**ASPICE-stopWatchTimer**

목차

[1. 고객 요구사항 3](#_Toc34690850)

[1.1 정의 3](#_Toc34690851)

[2. SYSTEM요구사항 명세서 4](#_Toc34690852)

[2.1 정의 4](#_Toc34690855)

[3. FW요구사항 명세서 5](#_Toc34690856)

[3.1 정의 5](#_Toc34690860)

[4. FW아키텍처 설계서 6](#_Toc34690861)

[4.1 Architecture 6](#_Toc34690862)

[4.2 UML (State Diagram) 7](#_Toc34690863)

[4.3 FlowChart 8](#_Toc34690864)

[5. FW단위 설계서 30](#_Toc34690865)

[5.1 SOSC\_init\_16MHz, SPLL\_init\_16\_160MHz, NormalRUNMode\_80MHz 30](#_Toc34690866)

[5.2 init\_data\_bss 30](#_Toc34690867)

[5.3 IRQ\_Config 30](#_Toc34690868)

[5.4 LPIT0\_init 30](#_Toc34690869)

[5.5 PortResetOnInit, PortPwrOnInit 30](#_Toc34690870)

[5.6 SysInitHW 31](#_Toc34690871)

[5.7 Sys\_Main 31](#_Toc34690872)

[6. Source Code 33](#_Toc34690878)

[7. FW단위시험 34](#_Toc34690879)

[7.1 EncoderRotateDetect 34](#_Toc34690880)

[7.2 EncoderKeyDetect 47](#_Toc34690881)

[7.3 Chattering\_4ms\_Cycle 49](#_Toc34690882)

[7.4 StopWatchUpdateAction 51](#_Toc34690883)

[8. FW통합검증 56](#_Toc34690884)

[9. FW자격검증 56](#_Toc34690885)

1. 고객 요구사항
2. 정의
3. 0부터 **5분**까지 1초 간격으로 count한다
4. Timer **Start/Stop/Reset** 기능을 포함해야 한다
5. 현재 **count 시작 위치를 조절 가능**해야 한다
6. 5분까지 count되면 count가 완료되었음을 알려야 한다. 5분까지 display count가 완료되면 다시 0으로 초기화되고, count는 멈춰있다
7. 타이머 동작은 **Debug Terminal을 통해 확인**할 수 있다. **디스플레이 형식은 (XX:XX)**을 따른다
8. SYSTEM요구사항 명세서

11. 정의

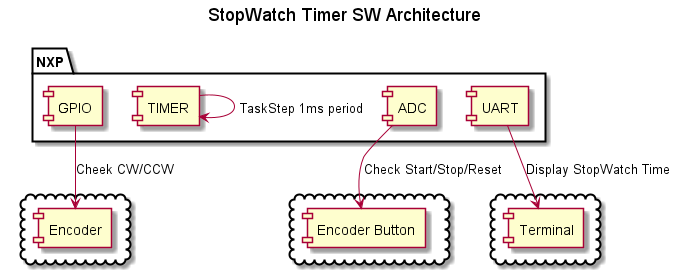
|  |  |  |  |
| --- | --- | --- | --- |
| **출처**  **(고객사 요구사항 ID)** | **SYS요구사항 ID** | **SYS요구사항 명** | **SYS요구사항 내용** |
| A | WATCH-SYS-001 | Timer Counter Range & Cycle | 타이머는 기본적으로 **0분0초에서 5분00초까지 동작**하며, **1초 간격**으로 count된다 |
| B | WATCH-SYS-002 | Timer Counter Start/Stop/Reset | **엔코더 내장 버튼**을 활용하여 타이머의**start/stop/reset 기능을 구현**한다 |
| C | WATCH-SYS-003 | Timer Control | **엔코더**를 활용하여 **시작되는 시간 값을 조절**할 수 있다 |
| D | WATCH-SYS-004 | Timer Endpoint | 시간이 05:00에 도달하면 디스플레이로 완료됨을 알리고, 시간 값을 00:00으로 초기화 한다 |

1. FW요구사항 명세서

4. 1. 정의

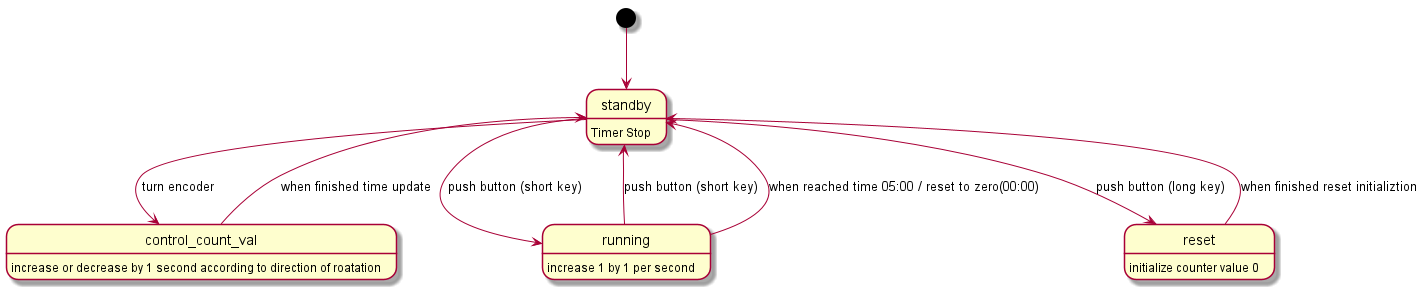
|  |  |  |  |
| --- | --- | --- | --- |
| **출처**  **(SYS 요구사항 ID)** | **FW요구사항 ID** | **FW요구사항 명** | **FW요구사항 내용** |
| WATCH-SYS-001 | WATCH-FW-001 | Timer Counter Range & Cycle | 초기값은 00:00을 시작으로, 최대 05:00 시간을 세는 스톱워치다. 증가하는 주기는 1초로 설정한다. |
| WATCH-SYS-002 | WATCH-FW-002 | Timer Counter Start/Stop/Reset | 스톱워치가 멈춰있을 땐 엔코더 버튼이 Start 기능으로 활성화되고, 반대로 스톱워치 동작 중에는 Stop 기능으로 활성화 된다.  추가적으로 스톱워치가 멈춰있을 때, 2초 이상의 Long Key 입력을 하면 시간을 00:00으로 초기화되는 Reset 기능을 수행한다 |
| WATCH-SYS-003 | WATCH-FW-003 | Timer Counter Control | 시간 조절 기능은 스톱워치가 동작하지 않을 때만 활성화된다. 엔코더를 회전할 때마다 회전 방향에 따라 시간을 ±1초씩 증감시키도록 한다. 증감 과정에서는 반드시 초기에 설정한 Timer Counter Range(00:00~05:00)를 지키도록 한다. |
| WATCH-SYS-004 | WATCH-FW-004 | Timer Counter Endpoint | 타이머가 동작 중이면서 05:00까지 센다면 별도의 Text을 디스플레이에 띄어서 나타내도록 한다. 그리고 00:00으로 카운터 값을 초기화 시키고, 스톱워치를 정지상태로 놓도록 한다. |

1. FW아키텍처 설계서
2. Architecture



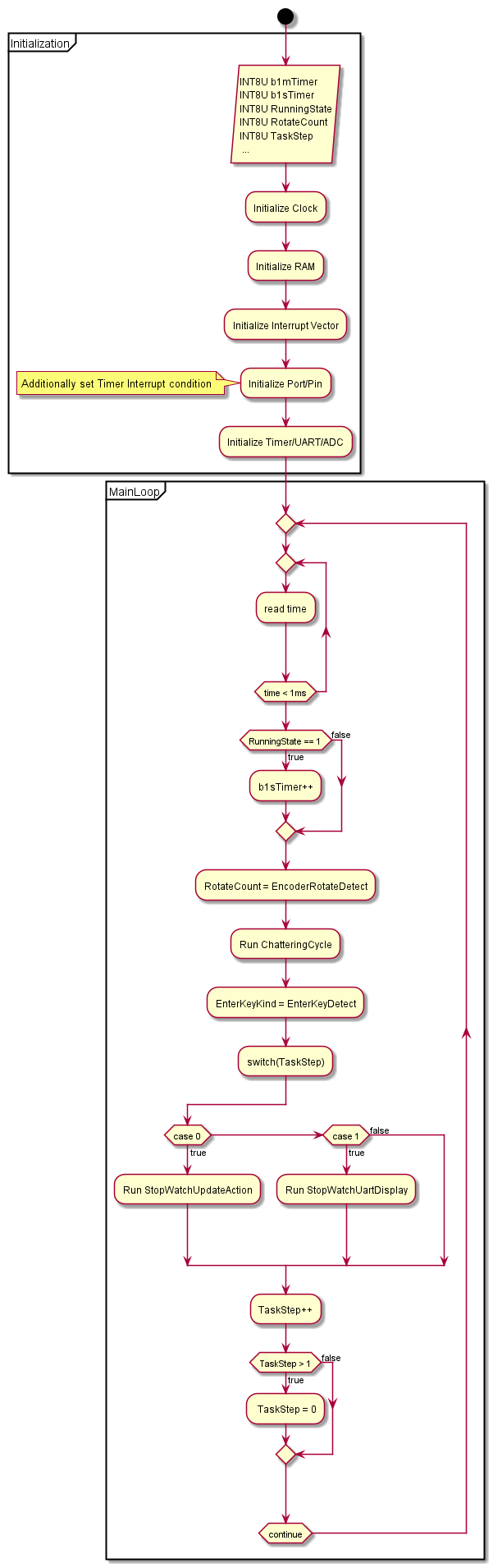
* **NXP** MCU를 사용할 것이며, **GPIO, TIMER, ADC, UART Peripheral**을 사용한다
* GPIO로 연결된 **Encoder의 2핀 입력**을 받아서 **CW/CCW 상태**를 검사한다.그리고 ADC에 연결된 엔터 키 버튼의 값을 판단해 **Start/Stop/Reset 상태를 결정**하게 된다.
* **UART는 스톱워치 시간을 Display하기 위한 도구**로 사용되며 **갱신되어야 할 상황에서만** PC 터미널에 시간을 변경하게 한다.
* **Timer는 1ms** 단위로 TaskStep을 확인할 뿐만 아니라 스톱워치 시간을 갱신하기 위한 주기인 **1s**를 측정하기 위해서 사용된다.

1. UML (State Diagram)



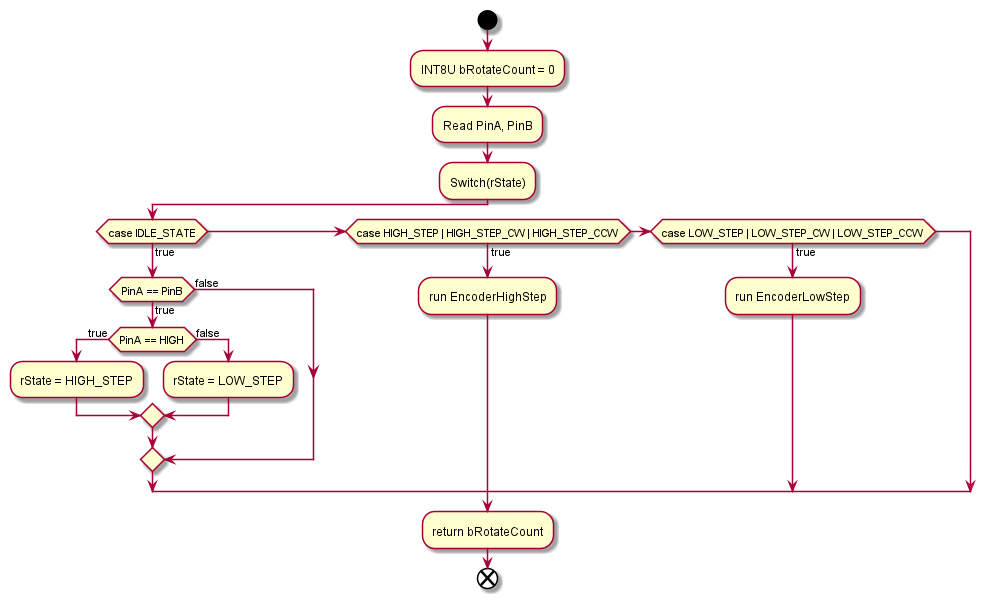
* 1. Stanby 상태
* 스톱워치가 멈춰있는 상태로 시간이 흘러가지 않는다
  1. Reset 상태
* Reset 상태는 엔코더 엔터 키를 2초 이상 길게 눌렀을 때 발생되며, **반드시 Stop 상태에서만 동작된다는 특징**이 있다. 그리고 리셋을 완료하면 자동으로 Standby 모드로 돌아가게 된다
  1. control\_count\_val 상태
* 엔코더를 돌려 시작 시간을 맞춰주는 상태다. **이 또한 반드시 Stop 상태에서만 동작된다는 특징**이 있다. 그리고 시간 갱신이 완료되었다면 즉시Standby 모드로 자동으로 돌아가게 된다
  1. Running 상태
* 스톱워치가 동작되는 상태를 말한다. 대기 모드 상태에서만 움직이며 **서로 엔터 키를 Short 키로 동작됐을 때 서로의 상태로 반전**된다. 1초에 1씩 증가를 하며, 만일 **최댓값인 5분에 도달하면 자동으로 Standby 모드로 돌아가게 된다.**

1. FlowChart
2. MainLoop



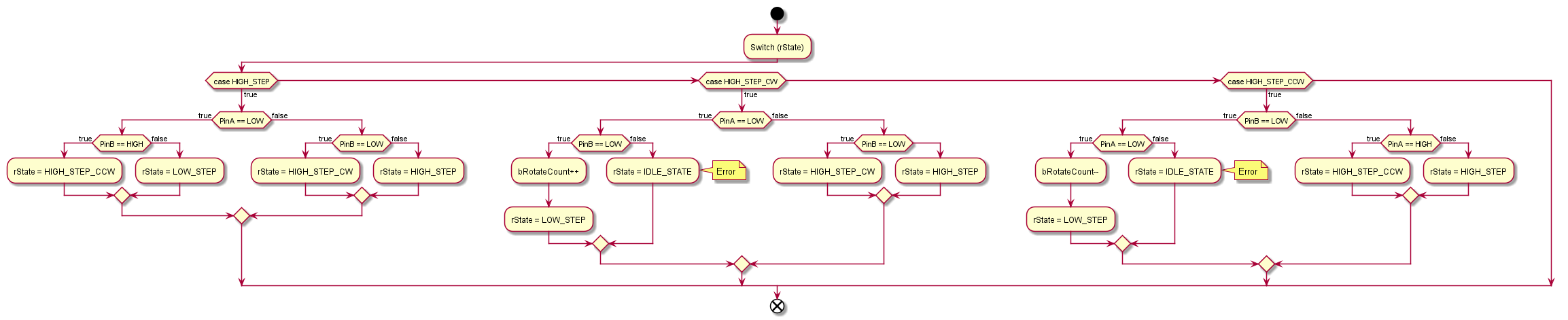
* MCU 사용을 위한 **Clock, RAM, 인터럽트 벡터(TIMER), 내부 Peripheral(GPIO, ADC)**을 초기화 한다
* 1ms 주기로 2가지의 TaskStep을 순서대로 검사하면서 일종의 감시 역할을 수행하게 된다. **따라서 전체 루프 주기는 약 2ms(시스템이 밀리지 않도록 반드시 2ms보다 여유 있는 시간에 종료되어야 한다)가 된다**
* 그리고 Task에 대해서 Detect와 Update을 분리해서 생각한다. Detect의 경우에는 빠른 입력이 들어오는 것을 대비하면 2ms 주기로 측정하는 것도 신호를 놓칠 불안한 면이 있다. 반면에 Update는 결국에는 Display 용도이기 때문에 빠른 전환은 필요가 없다. 축적된 데이터를 한 번에 표시해도 빠르기 때문에 시각적으로 큰 차이를 보이지 않는다. **따라서 Detect와 Update의 분리시켜서 동작을 시킬 것이다. 따라서 1ms마다 Detect를 2ms마다 Update 및 UART 송신을 하는 MainLoop로 구성해보려고 한다.**
* 1ms 주기 (중요도↑): 엔코더, 키 입력 Detect, 디바운싱 주기
* 2ms 주기: 엔코더, 키 입력 처리, 타이머 갱신, UART 터미널 송신

1. EncoderRotateDetect Loop

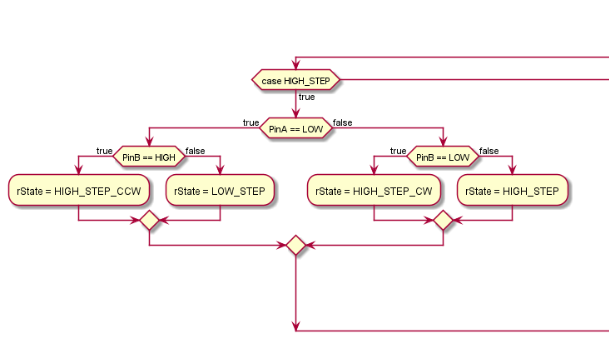


* 엔코더가 돌아갔을 때를 감지하는 Loop다. 엔코더의 상황은 총 5가지로 나눌 수 있다
  + - 1. **CW**
      2. **CCW**
      3. **Half CW**
      4. **Half CCW**
      5. **None**
* 그리고 EncoderRotateDetect에서 판단하는 상태는 총 7가지가 된다

1. **IDLE\_STATE :** 초기 상태 혹은 다른 상태에서 논리적 오류가 발생했을 때 들어온다
2. **HIGH\_STEP :** 두 핀 모두 1이 됐을 때 들어오며, CW 혹은 CCW을 판명한다
3. **HIGH\_STEP\_CW :** PinB가 0일 때 진입하게 된다. **PinB의 상태가 0이면서 PinA의 상태가 0임을 기대하면서** 최종적으로 CW 여부를 판명한다.
4. **HIGH\_STEP\_CCW :** PinA가 0일 때 진입하게 된다. **PinA의 상태가 0이면서 PinB의 상태가 0임을 기대하면서** 최종적으로 CCW 여부를 판명한다.
5. **LOW\_STEP :** 두 핀 모두 0이 됐을 때 들어오며, CW 혹은 CCW을 판명한다
6. **LOW\_STEP\_CW :** PinB가 1일 때 진입하게 된다. **PinB의 상태가 1이면서 PinA의 상태가 1임을 기대하면서** 최종적으로 CW 여부를 판명한다.
7. **LOW\_STEP\_CCW :** PinA가 1일 때 진입하게 된다. **PinA의 상태가 1이면서 PinB의 상태가 1임을 기대하면서** 최종적으로 CCW 여부를 판명한다.
8. EncoderHighStep

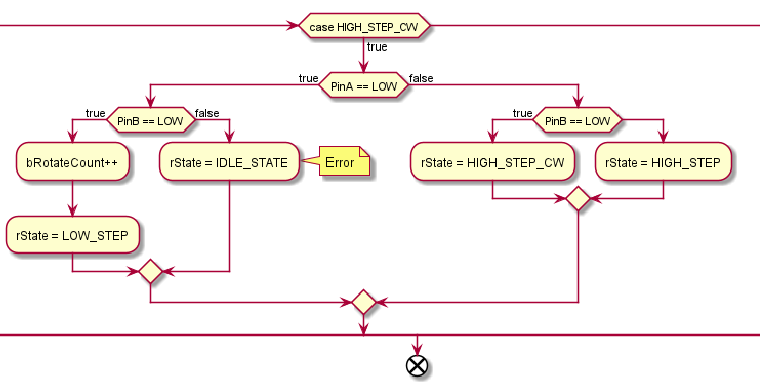


1. HIGH\_STEP



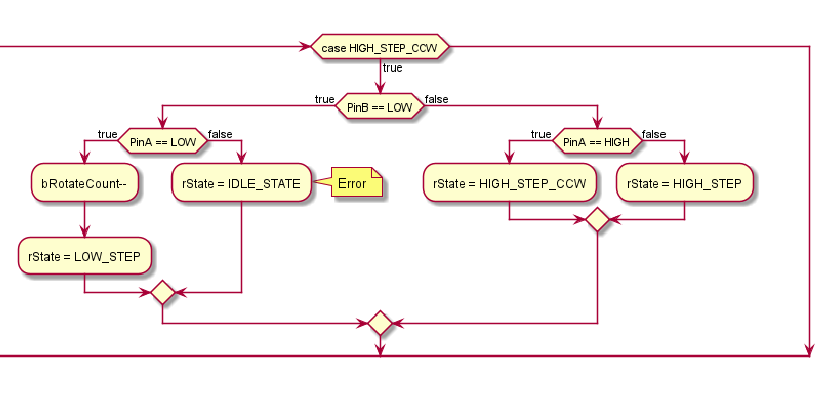
* IDLE\_STATE에서 두 핀이 모두 HIGH일 때 진입하게 된다
* **CW일 때는 PinB가 1에서 0으로 먼저 떨어지고, 후에 PinA가 1에서 0으로 떨어진다. 반면 CCW일 때는 PinA가 먼저 떨어진 후 PinB가 떨어진다.**
* 따라서 만일 PinA가 HIGH, PinB가 LOW라면 HIGH\_STEP\_CW 상태로 들어간다. 마찬가지로 PinA가 Low, PinB가 HIGH라면 HIGH\_STEP\_CW 상태로 들어간다
* 그리고 나머지 두 상태인 PinA==HIGH && PinB==HIGH는 아직 지속되는 HIGH\_STEP이다. **그리고 PinA==LOW && PinB==LOW는 LOW\_STEP 상태로 변환시킨다**

1. HIGH\_STEP\_CW



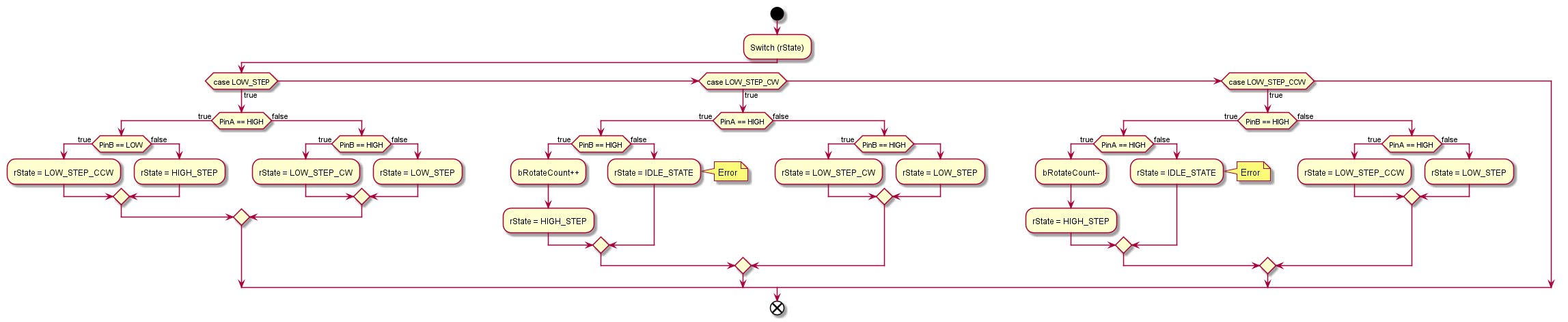
* PinA는 HIGH를 유지하면서 PinB가 HIGH에서 LOW로 먼저 떨어진 상태다. 이 때 PinA 마저 LOW라면 CW 회전을 한 상태고, 모두 0이기 때문에 LOW\_STEP으로 상태를 바꾼다
* **PinB를 다시 검사하는 것은 PinB의 상태가 1ms 이후에 다시 바뀌었을 지 모르기 때문이다. PinA는 LOW인데, PinB가 HIGH라면 엉뚱한 상황이기 때문에 에러로 판명하고 IDLE\_STATE로 상태를 변환한다.**
* 초기 진입 상황인 PinA == HIGH && PinB == LOW를 유지하면, 아직 PinA가 떨어지지 않았다고 판단하고(이땐 Half CW라고 생각할 수 있다) 상태를 유보한다
* 두 핀 모두 HIGH라면 Half CW에서 다시 원상태로 복귀한 상황으로 다시 HIGH\_STEP으로 복귀시킨다.

1. HIGH\_STEP\_CCW

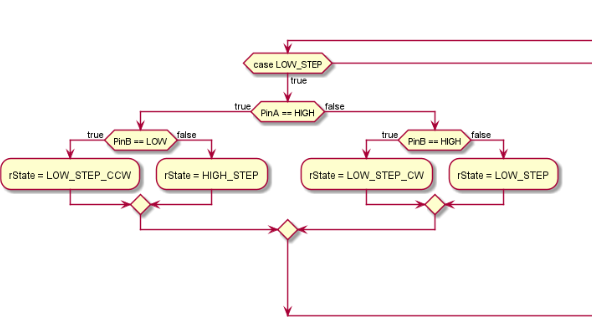


* PinB는 HIGH를 유지하면서 PinA가 HIGH에서 LOW로 먼저 떨어진 상태다. 이 때 PinB 마저 LOW라면 CCW 회전을 한 상태고, 모두 0이기 때문에 LOW\_STEP으로 상태를 바꾼다
* **PinA를 다시 검사하는 것은 PinA의 상태가 1ms 이후에 다시 바뀌었을 지 모르기 때문이다. PinB는 LOW인데, PinA가 HIGH라면 엉뚱한 상황이기 때문에 에러로 판명하고 IDLE\_STATE로 상태를 변환한다.**
* 초기 진입 상황인 PinA == LOW && PinB == HIGH를 유지하면, 아직 PinB가 떨어지지 않았다고 판단하고 상태를 유보한다.
* 만일 둘 다 상태라 HIGH라면, 초기 HIGH\_STEP 상태로 복귀시킨다.

1. EncoderLowStep

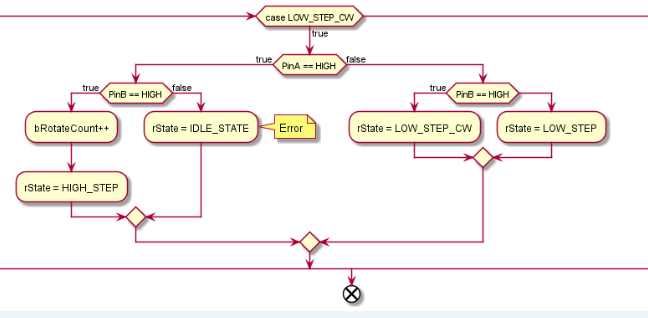


1. LOW\_STEP



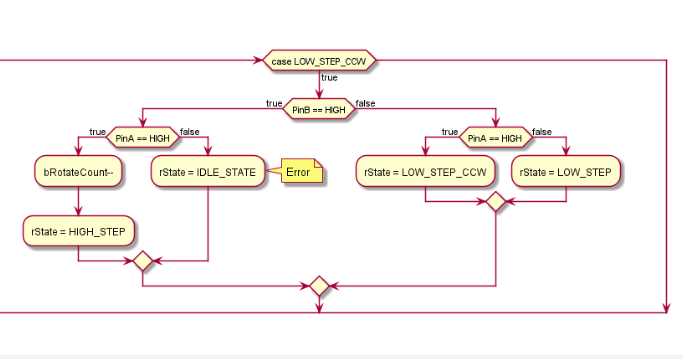
* IDLE\_STATE에서 두 핀이 모두 LOW일 때 진입하게 된다
* **CW일 때는 PinB가 0에서 1로 먼저 올라가고, 후에 PinA가 0에서 1로 올라간다. 반면 CCW일 때는 PinA가 먼저 올라간 후 PinB가 올라간다.**
* 따라서 만일 PinA가 LOW, PinB가 HIGH라면 LOW\_STEP\_CW 상태로 들어간다. 마찬가지로 PinA가 HIGH, PinB가 LOW라면 LOW\_STEP\_CCW 상태로 들어간다
* 그리고 나머지 두 상태인 PinA==LOW && PinB==LOW는 아직 지속되는 LOW\_STEP이다. 그리고 PinA==HIGH && PinB==HIGH는 HIGH\_STEP 상태를 전환시킨다

1. LOW\_STEP\_CW



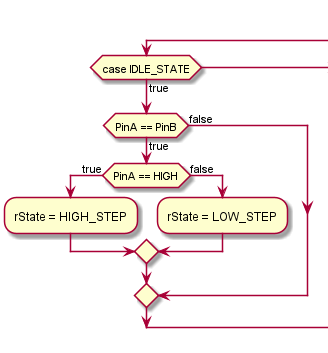
* PinA는 LOW를 유지하면서 PinB가 LOW에서 HIGH로 먼저 올라간 상태다. 이 때 PinA 마저 HIGH라면 CW 회전을 한 상태고, 모두 1이기 때문에 HIGH\_STEP으로 상태를 바꾼다
* **PinB를 다시 검사하는 것은 PinB의 상태가 1ms 이후에 다시 바뀌었을 지 모르기 때문이다. PinA는 HIGH인데, PinB가 LOW라면 엉뚱한 상황이기 때문에 에러로 판명하고 IDLE\_STATE로 상태를 변환한다.**
* 초기 진입 상황인 PinA == LOW && PinB == HIGH를 유지하면, 아직 PinA가 올라가지 않았다고 판단하고(이땐 Half CW라고 생각할 수 있다) 상태를 유보한다
* 두 핀 모두 LOW라면 Half CW에서 다시 원상태로 복귀한 상황으로 다시 LOW\_STEP으로 복귀시킨다.

1. LOW\_STEP\_CCW



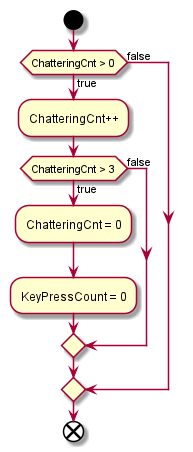
* PinB는 LOW를 유지하면서 PinA가 LOW에서 HIGH로 먼저 올라간 상태다. 이 때 PinB 마저 HIGH라면 CCW 회전을 한 상태고, 모두 1이기 때문에 HIGH\_STEP으로 상태를 바꾼다
* **PinA를 다시 검사하는 것은 PinA의 상태가 1ms 이후에 다시 바뀌었을 지 모르기 때문이다. PinB는 HIGH인데, PinA가 LOW라면 엉뚱한 상황이기 때문에 에러로 판명하고 IDLE\_STATE로 상태를 변환한다.**
* 초기 진입 상황인 PinA == HIGH && PinB == LOW를 유지하면, 아직 PinB가 올라가지 않았다고 판단하고(이땐 Half CCW라고 생각할 수 있다) 상태를 유보한다
* 두 핀 모두 LOW라면 Half CCW에서 다시 원상태로 복귀한 상황으로 다시 LOW\_STEP으로 복귀시킨다.

1. IDLE\_STATE



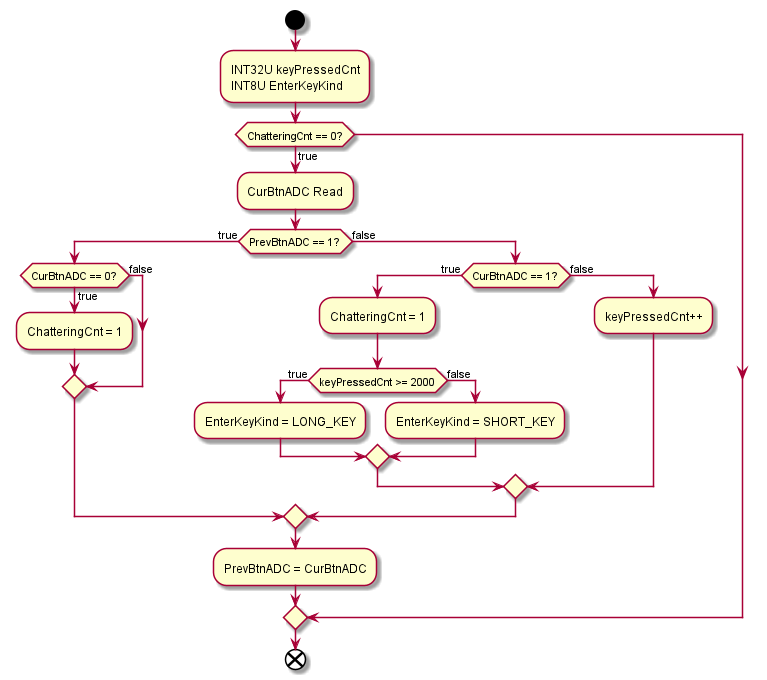
* IDLE\_STATE 상태로 초기에 진입했을 때와, 다른 상태에서 논리적 오류가 발생될 때 진입된다. 유휴 상태일 때 두 핀의 상태 값은 같으며 이에 따라 LOW\_STEP, HIGH\_STEP이 결정된다
* **초기 STATE로 IDLE\_STATE을 설정한 이유는 인터럽트 사용이 제한된 상황에서 현재 상태를 정의하고 가려고 하는 것이다. 그리고 현재 상태를 이전 상태로 사용해 상태를 파악하게 된다. 또한 엔코더가 반쯤 돌아간 상황에서 전원을 킨다면 시스템이 튀는 현상이 발생할 수도 있다. 따라서 이를 미연에 방지하고자 초기 IDLE\_STATE로 현재 상태를 명확하게 파악하기 위함에 있다.**

1. ChatteringCycle Loop



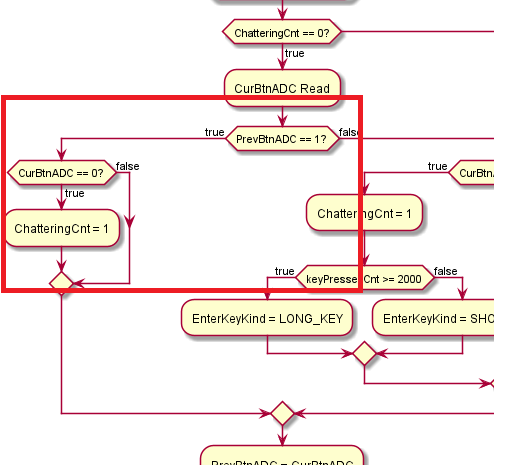
* **엔코더 키 값에서 발생한 채터링을 방지하기 위한 디바운싱 루프다. 오실로스코프 측정 결과 채터링은 1ms 내외에서 끝나는 것을 알았고, 최대 3ms로 디바운싱 시간을 잡아서 채터링을 방지하고자 한다.**
* 채터링이 끝나면 키 입력 시작과 끝에서 필요한 KeyPressCount 변수 초기화를 진행한다.
* EnterKeyDetect Loop보다 위에 놓은 것은 만일 키 입력이 시작된다면 바로 1ms 이후가 아닌 바로 ChatteringCnt 변수를 증가시키기 때문이다

1. EnterKeyDetect Loop



* EnterKeyDetect Loop는 엔터 키 입력을 확인하는 부분이다. 결과적으로는 SHORT\_KEY, LONG\_KEY, NONE을 검사하게 된다
* 현재, 이전 키 값을 비교해 총 4가지 상태를 확인할 수 있다
  + - * 1. 1 (안눌린 상태) -> 0 (눌린 상태): 막 키 입력이 들어간 상태
        2. 1 (안눌린 상태) -> 1 (안눌린 상태): 아무런 키 입력이 없는 상태
        3. 0 (눌린 상태) -> 0 (눌린 상태): 키 입력이 유지되는 상태
        4. 0 (눌린 상태) -> 1 (안눌린 상태): 막 키 입력이 떼진 상태

1. 1(안눌) ->0(눌)

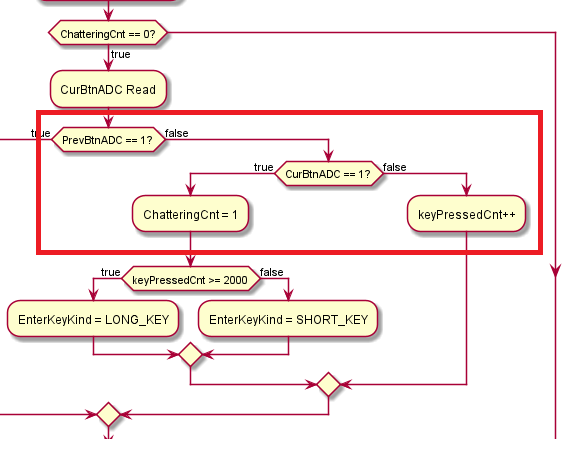
****

* PrevBtnADC == 0 && CurBtnADC == 1인 상황으로 막 버튼이 눌린 상태다**.** 채터링 발생에 대한 디바운싱을 위해 **ChatteringCnt 변수를 1로 초기화한다. 총 3ms 동안 디바운싱 시간을 가지게 된다**

1. 1(안눌)->1(안눌)

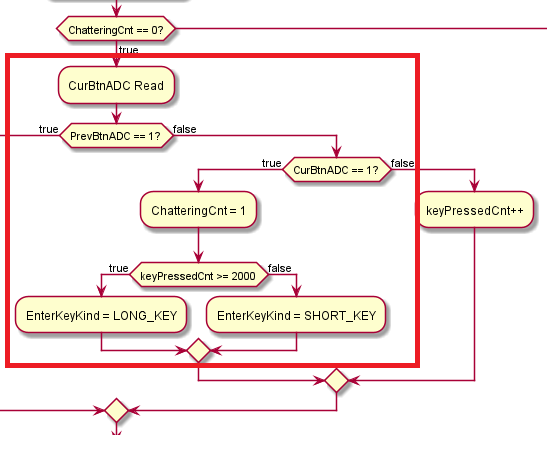
* 2번 상태 또한 위 그림에 해당되며, PrevBtnADC == 1 && CurBtnADC == 1인 상태다. 아무 변화가 없기 때문에 아무 동작도 안하고 버튼 상태만 초기화 하고 빠져 나오게 된다

1. 0(눌)->0(눌)

****

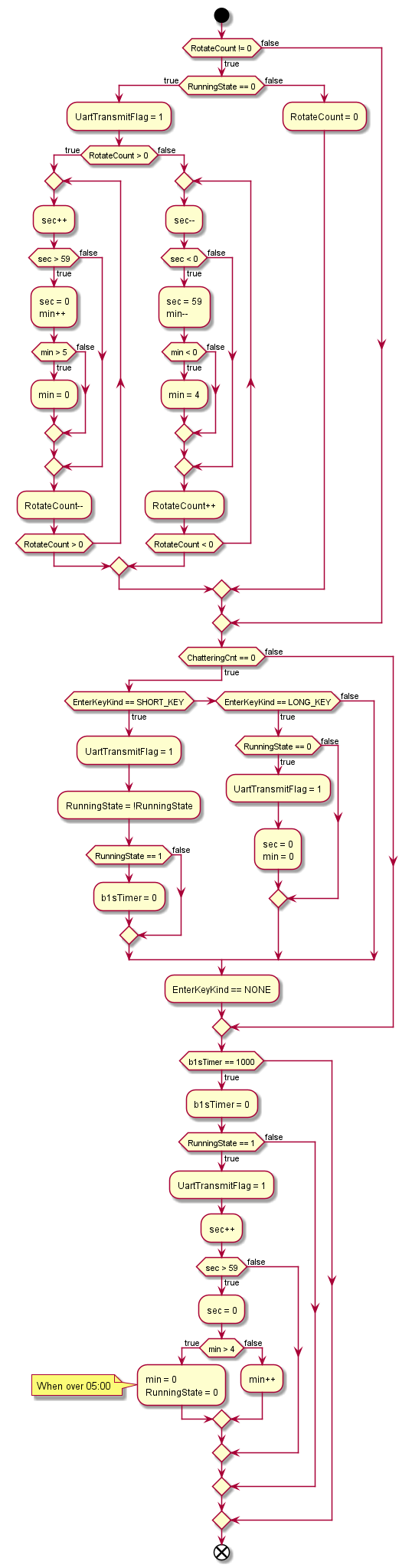
* PrevBtnADC == 0 && CurBtnADC == 0인 상황으로 버튼이 계속 눌려진 상태다. **이때는 keyPressedCnt 변수를 증가시키면서 키가 눌러진 시간을 확인하게 된다.**

1. 0(눌)->1(안눌)

****

* PrevBtnADC == 0 && CurBtnADC == 1인 상황으로 버튼이 막 떼진 상태다. 키 입력일 때와 마찬가지로 ChatteringCnt를 1로 초기화해서 디바운싱 시간을 가지게 된다. **키 입력의 주기는 1ms다. 따라서 리셋(LONG\_KEY)의 기준이 2초이므로 2000이상 값을 가지게 된다면 LONG\_KEY로 값을 결정하게 된다. 그리고 2초 이하라면 시작/멈춤(SHORT\_KEY)의 값인 SHORT\_KEY로 반환하게 된다**

1. StopWatchUpdateAction Loop

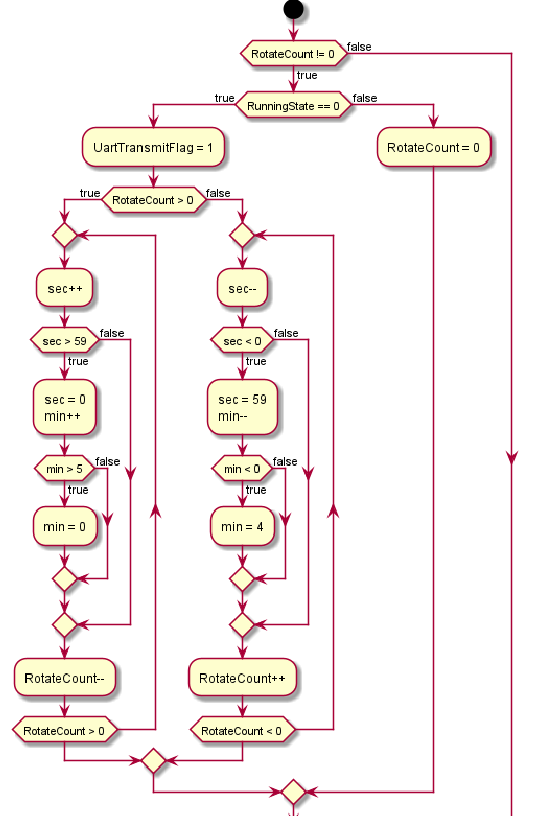


* StopWatchUpdateAction Loop에서는 총 3가지의 일을 연속적으로 실행하게 된다

1. 엔코더 값 처리
2. 키 값 처리
3. 1초 타이머 검사 및 처리

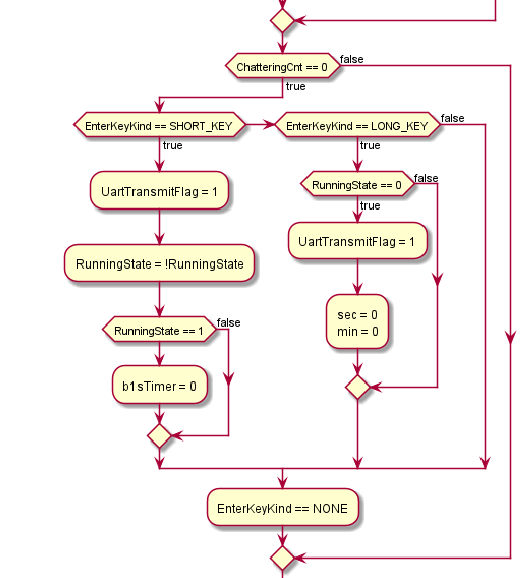
* **1ms 주기이므로 반드시 1), 2), 3)의 처리 시간은 총합 1초 미만이어야 한다**

1. EncoderProcess



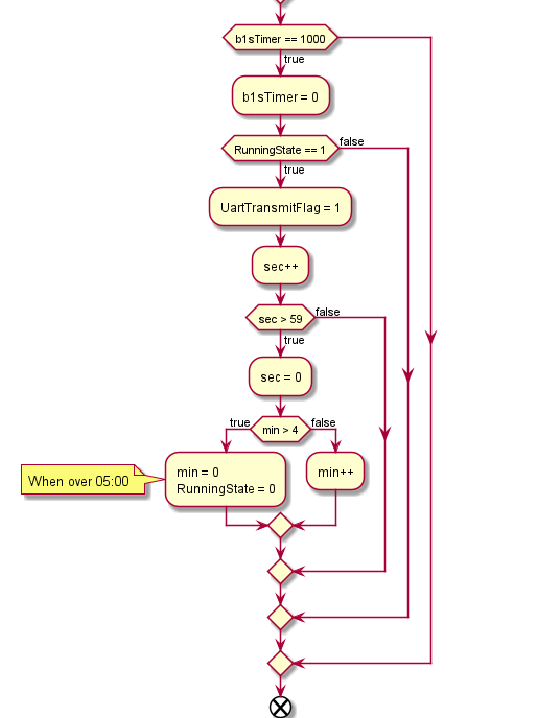
* 1ms 마다 감지한 엔코더 회전 수를 확인하고 처리하는 루프다. **위에서 엔코더 회전 값에 대해서 RotateCount에 CW면 +1, CCW면 -1로 처리하게 했다. 이는 1ms 감지 주기와 2ms 처리 주기가 맞지 않기 때문에 값을 누적시켰다. 따라서 예를 들어 2번 CW 회전을 했다고 하면 RotateCount에는 2 값이 들어갔을 것이다.**
* 그리고 RotateCount가 0이 아니라면 루프를 한 단계 들어가게 된다. 그리고 현재 정지 상태가 아니라면 Count를 0으로 초기화하고 빠져나오게 된다. 정지 상태고 회전 수가 존재하기 때문에 시간 갱신이 필요하다. 따라서 UartTransmitFlag를 1로 Set하게 된다. 그리고 회전 수만큼 시간을 갱신하게 된다

1. EnterKey Process



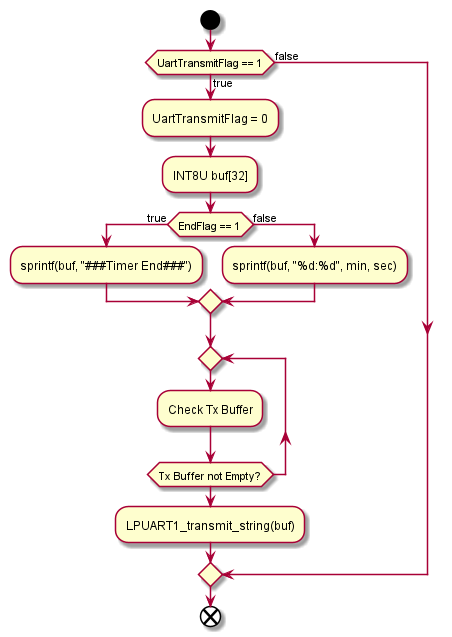
* 1ms 마다 엔터 키를 처리하는 부분이다. 먼저 현재 디바운싱 주기라면 처리를 잠시 미뤄야 하기 때문에 루프를 빠져나가게 된다.
* **만일 디바운싱 주기가 아니라면 EnterKeyKind 검사를 통해서 SHORT\_KEY, LONG\_KEY를 판단하게 된다. SHORT\_KEY는 시작/정지, LONG\_KEY는 리셋의 의미를 담고 있다. 따라서 SHORT\_KEY에서는 현재 RunningState를 반전시키고, LONG\_KEY에서는 시간 리셋 처리를 하도록 한다. 이 또한 시간 갱신이 이뤄지는 것이기 때문에 UartTransmitFlag를 1로 Set한다.**
* 그리고 SHORT\_KEY면서 상태가 Running으로 바뀌었을 때는 1초 타이머를 변수를 초기화해서 1초 카운트를 시작하게 된다

1. 1sTimer Process



* 1sTimer 변수가 1000이 되었는지 확인하는 부분이다. 아직 1초가 되지 않았다면 루프를 빠져나가게 된다.
* 1초가 되었고, 현재 Running 상태라면 갱신이 생기기 때문에 UartTransmitFlag를 1로 Set한다. 그리고 1초를 올리게 된다. 만일 5분이 되었다면 타이머 동작을 종료하게 된다

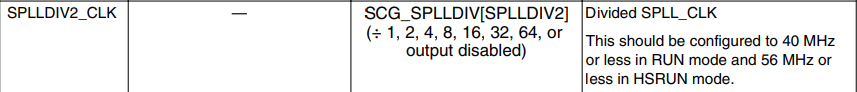
1. StopWatchUartDisplay Loop



* UART 통신을 통해 PC 터미널에 현재 시간을 보여주는 Loop다. **엔코더 혹은 타이머에 의해 갱신되는 값들을 약 2ms마다 한번씩 보여주게 된다.** **그리고 불필요한 전송을 위해 반드시 Flag가 1로 Set 되었을 때만 전송을 해야만 한다.**
* 최종적으로 사용하려는 함수는 LPUARTx\_transmit\_string(char \*)의 형태를 가지게 된다. 따라서 char 배열을 선언하고 이를 sprint 함수를 사용해서 문장 구성을 한다. 문장은 “분,시” 혹은 앞에 추가적인 수식어를 붙여서 구성된다
* **정보 손실을 막기 위해서 UART의 Tx 버퍼가 비었는지 확인하고, 비었을 때 송신하게 된다.** 비어있지 않다면 반복적으로 검사하게 된다. 그리고 타이머가 종료하게 되면 “Timer End” 문자열을 송신해서 종료됨을 알리게 한다

1. FW단위 설계서
2. SOSC\_init\_16MHz, SPLL\_init\_16\_160MHz, NormalRUNMode\_80MHz
3. 시스템 클럭으로 8MHz 크리스탈 오실레이터 사용
4. PLL 설정을 통해 클럭 160MHz로 주파수 체배
5. 시스템 클럭 Normal Run 모드로 Sys&Core 80MHz, Peripheral Bus 40MHz, Flash Memory 20MHz로 설정
6. **Boot Code가 없을 시 링커스크립트 Flash 영역 초기화와 같이 수행하도록 한다**
7. init\_data\_bss
8. ROM으로부터 벡터 테이블, 초기화 데이터, 코드를 RAM으로 복사
9. bss data section을 0으로 초기화한다
10. IRQ\_Config
11. 사용할 인터럽트에 대해서 NVIC Enable
12. 1ms, 250us 타이머를 위한 LPIT 관련해 총 2개의 NVIC Enable을 한다
13. 크게 순서는 상관이 없기 때문에 우선순위는 0x0A(10)로 동일하게 설정한다
14. 내부에서 사용하는 함수는 enable\_interrupts\_NVIC로 매개변수로 Interrupt Vector 번호와, 우선순위를 받게 된다.
15. LPIT0\_init
16. 어떠한 클럭에 동기화할지에 대해서 클럭 소스를 선택해야 한다. 실제 코드에서는 SPLLDIV2\_CLK으로 설정된다.

Clock definitions(P.535)에 따르면 RUN mode일 때 SPLLDIV2\_CLK는 40MHz 이하로 설정된다.



1. 타이머 채널에 대한 인터럽트 Enable을 설정한다
2. TVAL 레지스터에 동기화한 클럭 기준으로 카운터 주기를 설정한다. 예를 들어 버스 클럭이 40MHz고, TVAL이 40KHz라면 1ms 주기로 IRQ\_Handler가 불러진다
3. 사용할 채널은 총 2개로, 각각 250us(2번 채널)와 1ms(0번 채널) 카운터 역할을 한다.
4. PortResetOnInit, PortPwrOnInit
5. 사용할 핀에 대해서 MUX 레지스터에서 GPIO 혹은 다른 IO 입출력으로 사용할지 설정한다
6. 입출력 방향도 설정하고, 만일 외부 인터럽트로 사용한다면 인터럽트 조건(상승, 하강, 모두)에 대해서도 초기화한다
7. 핀이 동작할 하드웨어적 전압 범위도 설정이 가능하다
8. PortResetOnInit은 MCU가 리셋을 한 직후, PortPwrOnInit은 파워가 들어간 후 실행한다
9. SysInitHW
10. 만일 포트 초기화에서 GPIO가 아닌 Alternate 핀으로 지정했다면 해당 IO에 대해서 구체적으로 정의한다
11. Debug시 터미널 사용을 위해 UART를 활용할 것인데, Baud Rate와 프로토콜에 대해서 설정한다
12. 엔코더 엔터키가 ADC 핀으로 연결되어 있다. 연결된 ADC핀에 대한 Enable, 샘플링 주기, Resolution에 대한 설정을 진행한다
13. Sys\_Main
14. 사용할 변수에 대해서 초기화를 한다
15. 전원 및 시스템 관리를 위해서 WatchDog를 켰다가 이후 감지가 시작했을 때는 끄도록 한다
16. 위에서 설정한 타이머는 250us, 1ms 총 2개다. 1ms 타이머에서 변형된 타이머로 2ms도 있다고 할 수 있다. 따라서 실질적으로 돌아가는 타이머는 총 3개의 타이머라고 말할 수 있다
17. 250us 주기마다
    * 1. EncoderRotateDetect

엔코더가 돌아간 것을 검사하게 된다. 수십 번의 테스트 결과로 1ms 내에서도 엔코더 파형이 나타날 수 있으므로, 250us 주기마다 따로 검사를 해서 돌아간 횟수를 정확하게 파악하도록 한다

1. 1ms 주기마다
2. 1초 타이머를 위한 Sw1sTimer 변수를 1 증가시킨다
3. EncoderKeyDetect

ADC0 – CH1에 연결된 엔코더 키 값을 읽어 눌렀는지에 대해서 검사한다. 여러 번의 테스트 결과 약 130~180 값이 나온다면 눌렸다고 판단할 수 있다. 반대로 245~260 값은 키가 떼진 상태로 말할 수 있다. 따라서 감지 시간을 측정해 SHORT\_KEY, LONG\_KEY, NONE을 반환한다

1. Chattering\_4ms\_Cycle

EncoderKey가 눌렸을 때와 떼졌을 때의 디바운싱을 위한 부분이다. 1ms 주기를 4번을 돌면서 총 4ms의 디바운싱 시간을 가지게 된다

1. 2ms 주기마다
2. StopWatchUpdateAction

값이 발생했을 때의 엔코더 값 처리, 키 값 처리, 1초 타이머 처리에 대한 부분이다. 값에 대한 처리는 시각적으로 보이는 부분에 집중되기 때문에 감지보다는 느린 주기인 2ms마다 해도 무방하다

1. StopWatchUartDisplay

UartTransmitFlag가 1로 Set이 되었을 때만 PC 콘솔로 데이터를 보내게 된다

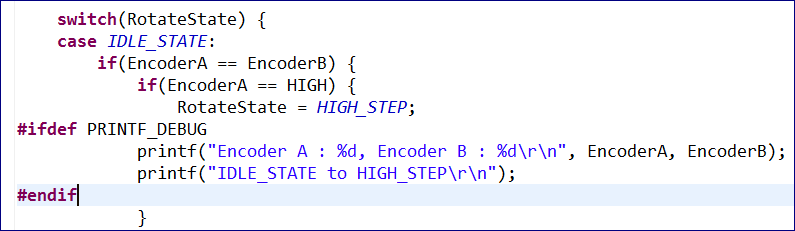
6. Source Code

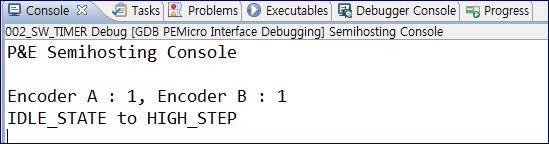
* 첨부파일에 추가하도록 하겠습니다

1. FW단위시험
2. EncoderRotateDetect

* 이전 RotateState와 현재 입력된 엔코더 키 값을 측정해서 다음 RotateState를 판단하게 된다
  1. IDLE\_STATE
     1. IDLE\_STATE -> HIGH\_STEP

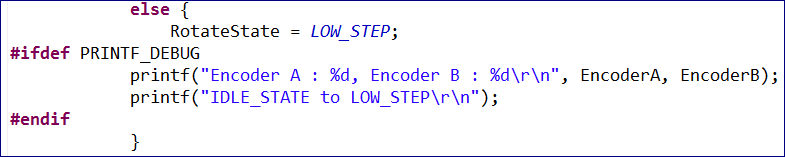
전원을 켰을 초기에 EncoderA, EncoderB 모두 1이 입력됐을 때

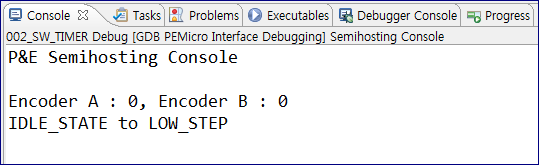




* + 1. IDLE\_STATE -> LOW\_STEP

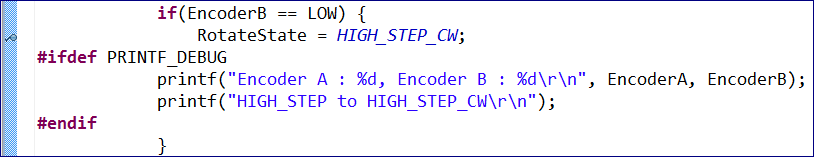
전원을 켰을 초기에 EncoderA, EncoderB 모두 0이 입력됐을 때

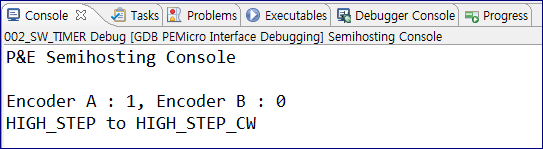




* 1. HIGH\_STEP
     1. HIGH\_STEP -> HIGH\_STEP\_CW

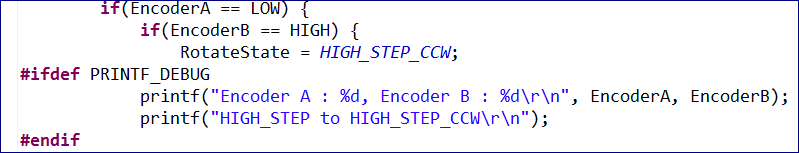
HIGH\_STEP 상태에서 CW 방향으로 Encoder가 돌았을 때

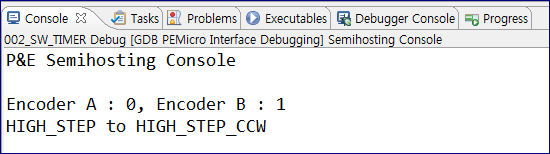




* + 1. HIGH\_STEP -> HIGH\_STEP\_CCW

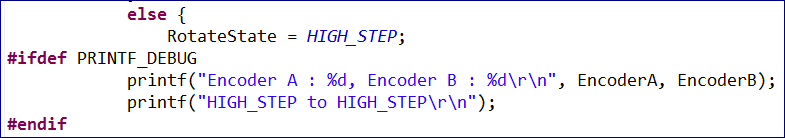
HIGH\_STEP 상태에서 CCW 방향으로 Encoder가 돌았을 때

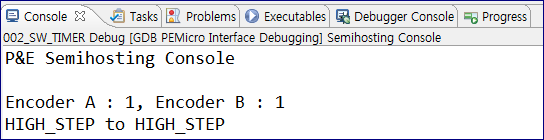




* + 1. HIGH\_STEP -> HIGH\_STEP

HIGH\_STEP 상태에서 상태가 계속 유지되는 상황



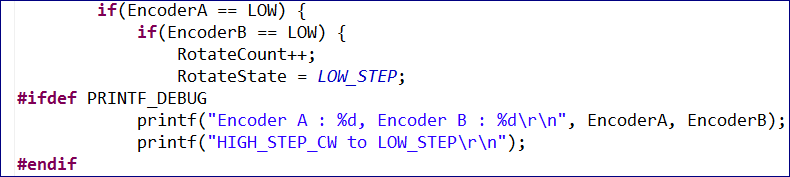


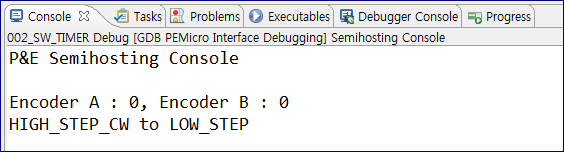
* + 1. HIGH\_STEP -> LOW\_STEP

HIGH\_STEP 상태에서 발생하는 일종의 에러 상황. BreakPoint를 건 후 1분 동안 Encoder을 돌렸지만 에러 상황이 발생하지 않음

* 1. HIGH\_STEP\_CW
     1. HIGH\_STEP\_CW -> LOW\_STEP

EncoderB 핀까지 모두 LOW로 떨어져 한 바퀴를 확실히 돈 상황이다



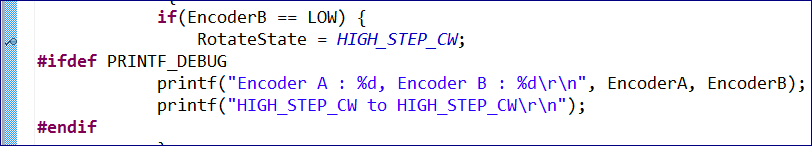


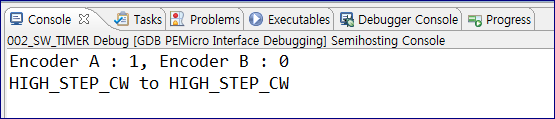
* + 1. HIGH\_STEP\_CW -> IDLE\_STATE

에러 상황으로 EncoderB 핀은 LOW로 떨어졌지만 EncoderA는 HIGH가 된 상황이다. 일종의 에러 상황으로 BreakPoint를 건 후 1분 동안 Encoder을 돌렸지만 에러 상황이 발생하지 않았다

* + 1. HIGH\_STEP\_CW -> HIGH\_STEP\_CW

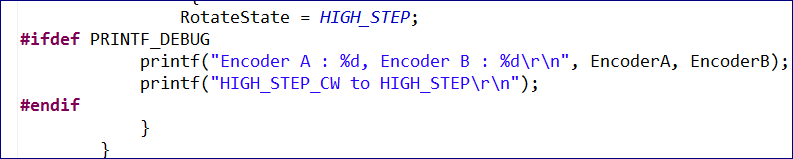
바로 EncoderB 핀이 LOW로 떨어지지 않기 때문에 반드시 이 루프에 걸리게 됐다

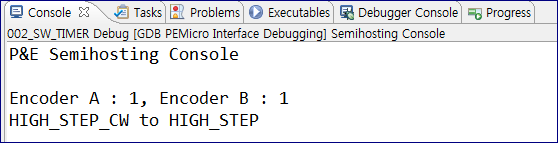




* + 1. HGIH\_STEP\_CW -> HIGH\_STEP

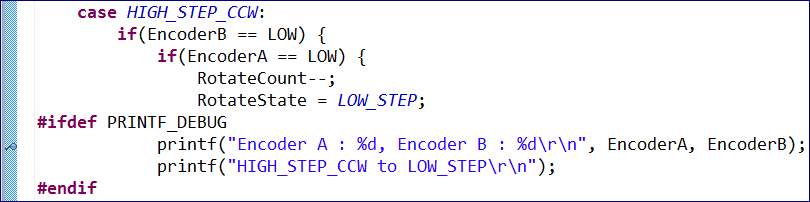
CW 방향으로 반 돌린 후 CCW 방향으로 반 돌려서 원 상태로 돌아온 상황을 말한다

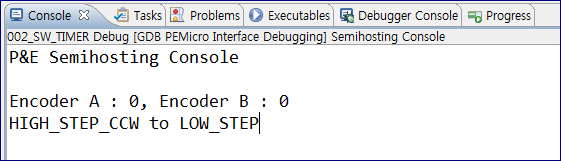




* 1. HIGH\_STEP\_CCW
     1. HIGH\_STEP\_CCW -> LOW\_STEP

EncoderA 핀까지 0으로 떨어지면서 확실한 CCW로 돌아간 상태다



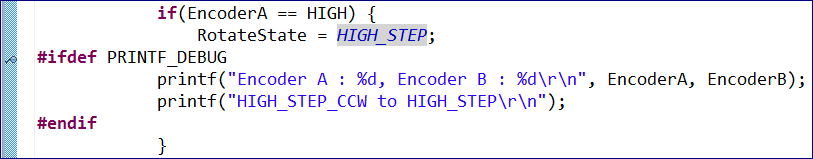


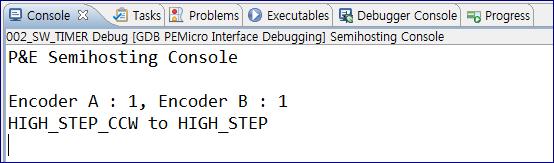
* + 1. HIGH\_STEP\_CCW -> IDLE\_STATE

에러 상황으로 EncoderA 핀은 LOW로 떨어졌지만, EncoderB는 HIGH가 된 상황이다. 일종의 에러 상황으로 BreakPoint를 건 후 1분 동안 Encoder를 돌렸지만 에러 상황이 발생하지 않았다

* + 1. HIGH\_STEP\_CCW -> HIGH\_STEP

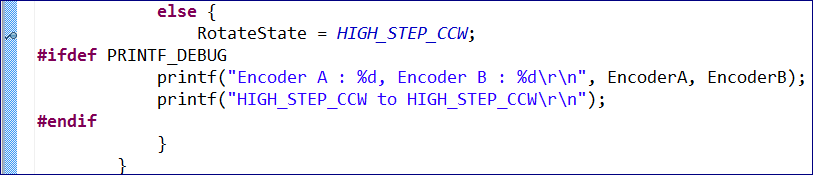
CCW로 반이 돈 상태에서 다시 CW로 반이 돌아 HIGH\_STEP으로 돌아간 상황이다

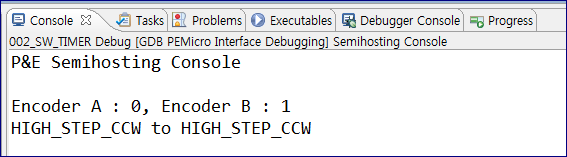




* + 1. HIGH\_STEP\_CCW -> HIGH\_STEP\_CCW

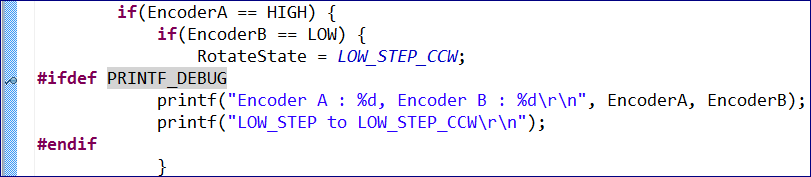
대체적인 Encoder Rotate에서 바로 EncoderA 핀이 LOW로 떨어지지 않기 때문에 반드시 이 루프에 걸리게 됐다

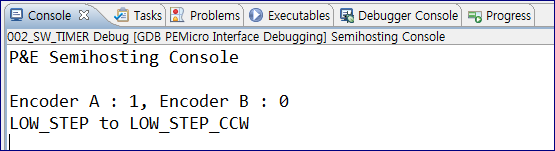




* 1. LOW\_STEP
     1. LOW\_STEP -> LOW\_STEP\_CCW

EncoderA 핀이 먼저 HIGH로 올라가면서 반 CCW가 돌아간 상황이다



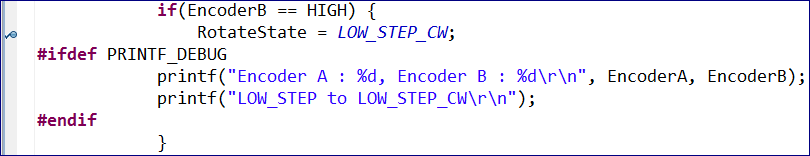


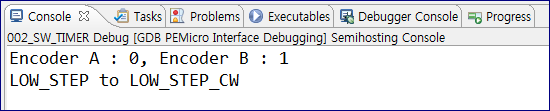
* + 1. LOW\_STEP -> HIGH\_STEP

EncoderA 핀만 먼저 HIGH로 올라간 상황이 아닌 250us 주기 내에서 동시에 EncoderB 핀도 HIGH로 올라간 상황으로 일종의 에러 상황이다. BreakPoint를 건 후 1분 동안 테스트 결과 점프하지 않았다

* + 1. LOW\_STEP -> LOW\_STEP\_CW

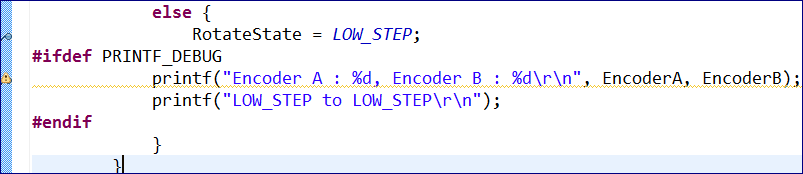
EncoderB 핀이 먼저 HIGH로 올라가면서 반 CW로 돌아간 상황이다

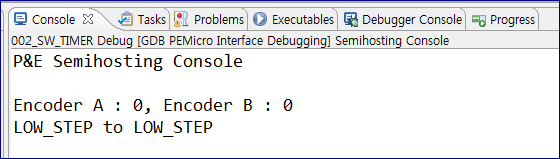




* + 1. LOW\_STEP -> LOW\_STEP

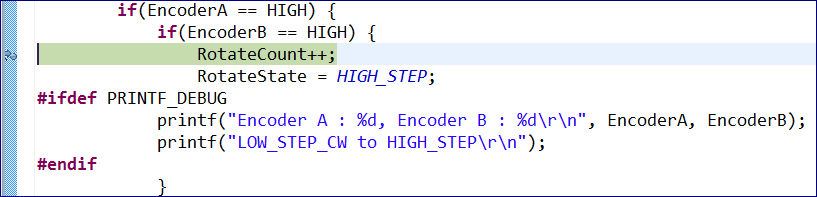
LOW\_STEP이 지속적으로 유지되는 상황이다

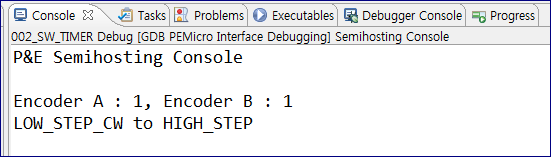




* 1. LOW\_STEP\_CW
     1. LOW\_STEP\_CW -> HIGH\_STEP

EncoderA 핀이 HIGH로 올라가면서 CW 방향으로 돌아간 상태



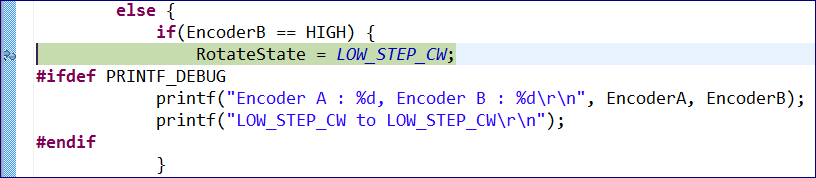


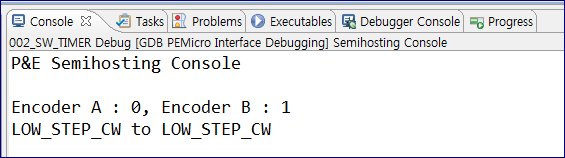
* + 1. LOW\_STEP\_CW -> IDLE\_STATE

EncoderA 핀은 HIGH로 올라갔지만, EncoderB 핀은 LOW로 떨어진 에러 상황. BreakPoint를 걸고 1분 동안 테스트 해본 결과, 루프에 잡히지 않았다

* + 1. LOW\_STEP\_CW -> LOW\_STEP\_CW

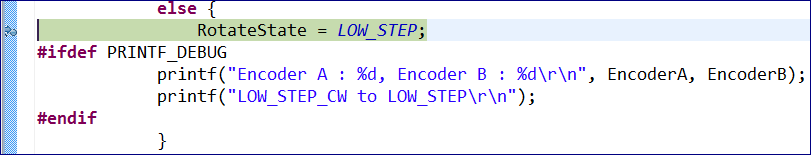
EncoderA 핀이 HIGH로 올라가기를 기대하는 상황이다

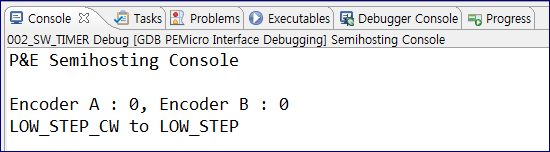




* + 1. LOW\_STEP\_CW -> LOW\_STEP

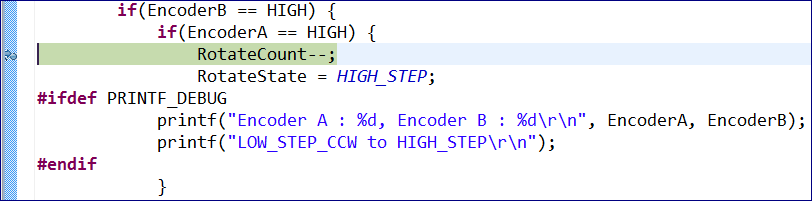
반 CW로 돌았다가 원 위치로 되돌아 온 상황이다

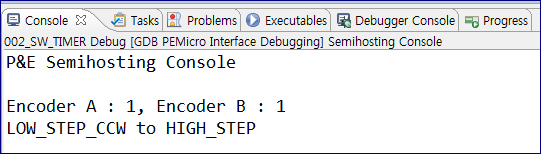




* 1. LOW\_STEP\_CCW
     1. LOW\_STEP\_CCW -> HIGH\_STEP

EncoderA가 HIGH가 되어 CCW로 돌아가진 상태



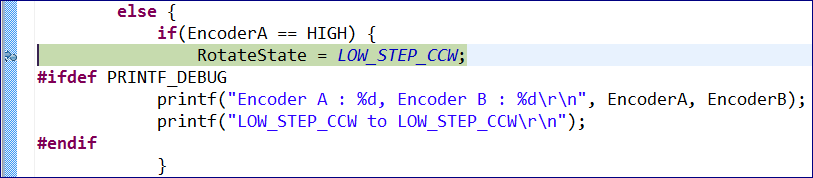


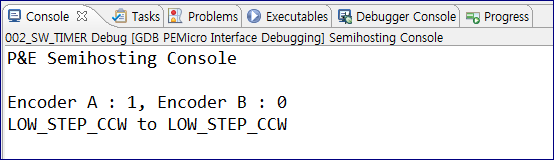
* + 1. LOW\_STEP\_CCW -> IDLE\_STATE

EncoderA가 HIGH가 되었지만, EncoderB가 LOW가 된 에러 상황이다. BreakPoint를 걸고 1분동안 테스트한 결과 루프로 빠지지 않았다

* + 1. LOW\_STEP\_CCW -> LOW\_STEP\_CCW

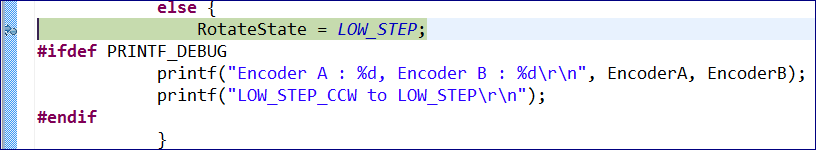
EncoderA가 HIGH로 변화됨을 기대하는 상황이다

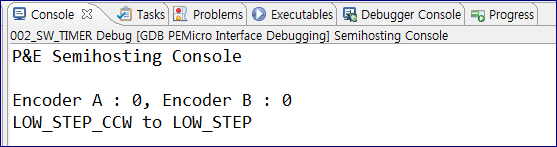




* + 1. LOW\_STEP\_CCW -> LOW\_STEP

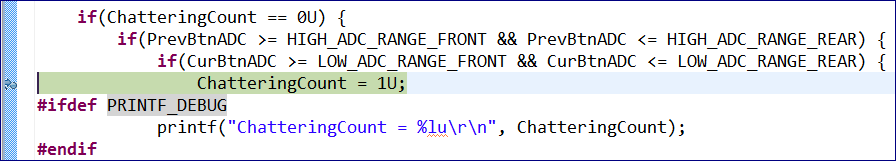
반 CCW 돌아간 상태에서 원래대로 돌아온 상황이다

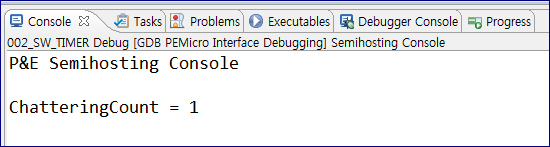


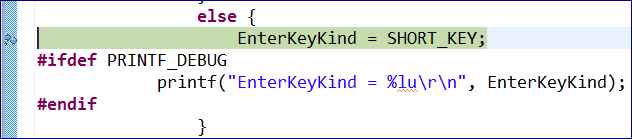


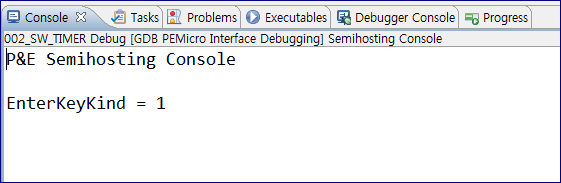
1. EncoderKeyDetect
   1. SHORTKEY
      * 1. 정지상황에서 시작버튼을 눌렀을 때, Running상태에서 정지버튼을 눌렀을 때

채터링 디바운싱이 시작되면서, SHORT\_KEY 값을 반환한다



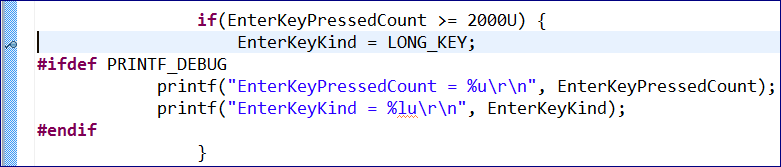


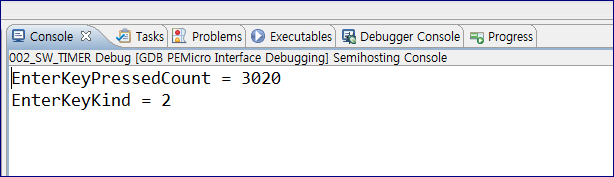




* 1. LONGKEY
     + 1. 2초 이상 EnterKey를 눌렀을 때

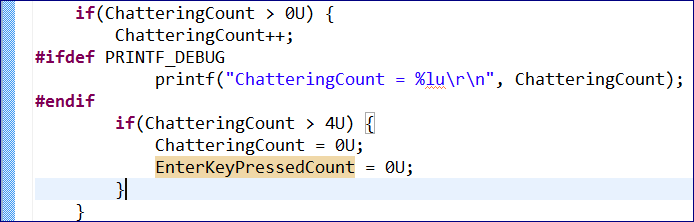
정지상황에서만 가능하며, SHORT\_KEY와 마찬가지로 디바운싱 시간을 가지고, 눌린 시간을 EnterKeyCount로 측정을 한다. 그리고 2000이상(2ms) 값이 반환되면 LONG\_KEY로 반환한다

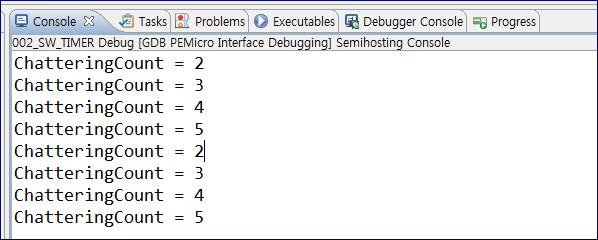




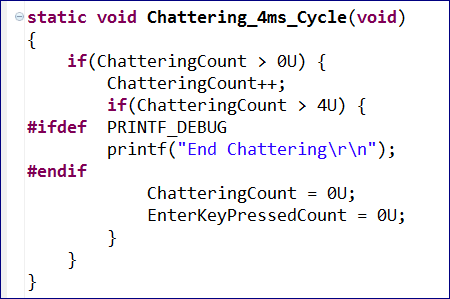
1. Chattering\_4ms\_Cycle

* EnterKey가 눌리고 떼졌을 때 발생하는 채터링을 막기 위한 구문. EnterKey가 눌리고 떼졌을 때, ChatteringCount를 1로 초기화되며, 5까지 총 4ms 동안 디바운싱 시간을 가지게 된다

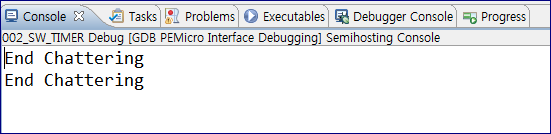




* 다음은 채터링 디바운싱 시간이 종료된 이후의 상황을 테스트해본다



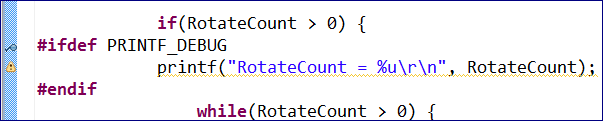
눌렀을 때와 떼졌을 때 모두 채터링 방지가 필요하므로, 한 번 누를 때 총 2번에 printf문이 동작을 한다

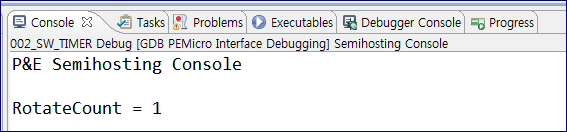


1. StopWatchUpdateAction

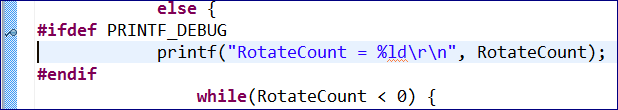
* Encoder, EnterKey에 대해서 감지된 값을 가지고 처리하는 부분

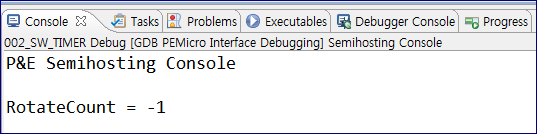
1. Encoder가 시계방향으로 돌아갔을 때



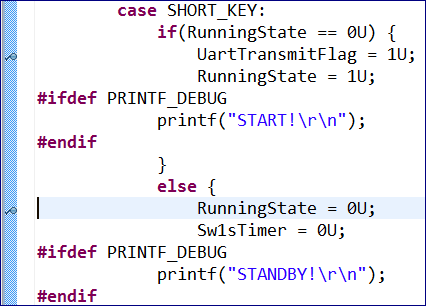


1. Encoder가 반시계방향으로 돌아갔을 때

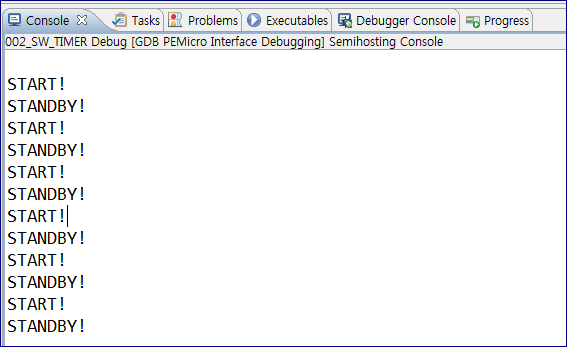




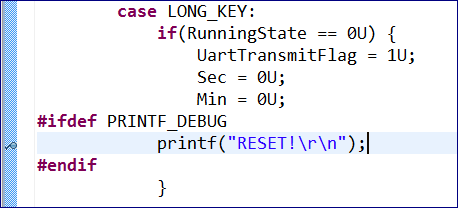
1. SHORT\_KEY를 눌러 시작, 정지 상태는 전환할 때



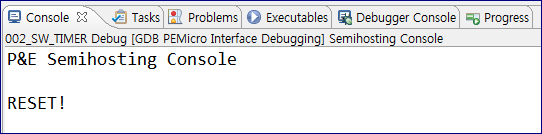
지속적으로 SHORT\_KEY를 동작시켜 START, STANDBY가 적절하게 바뀌는 확인하는 상황



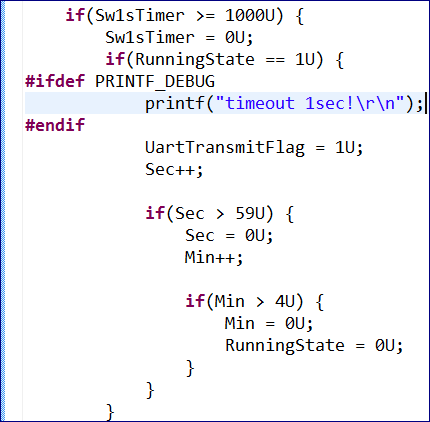
1. LONG\_KEY를 눌러 StopWatch Reset을 했을 때

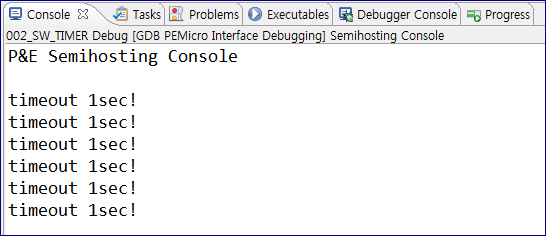


Standby 상태일 때 한 번 LONG\_KEY를 눌러 RESET을 하는 상황

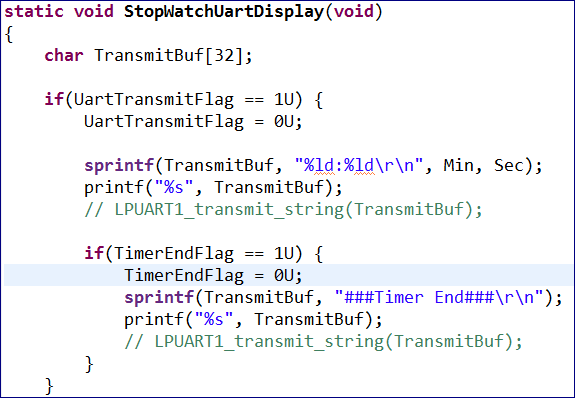


1. RunningTime 때 1초가 걸렸을 때

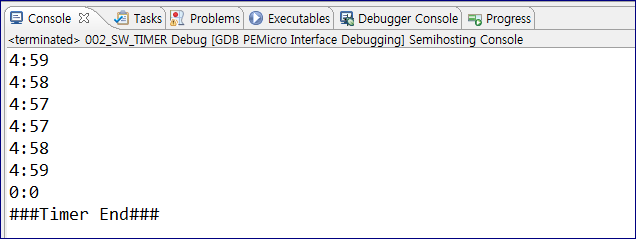




1. StopWatch가 동작할 때, UART를 통해 시간을 보낼 때



상황은 Encoder를 돌려 4:55초에 맞춘 후 동작버튼을 누른 상태다



1. FW통합검증
2. FW자격검증