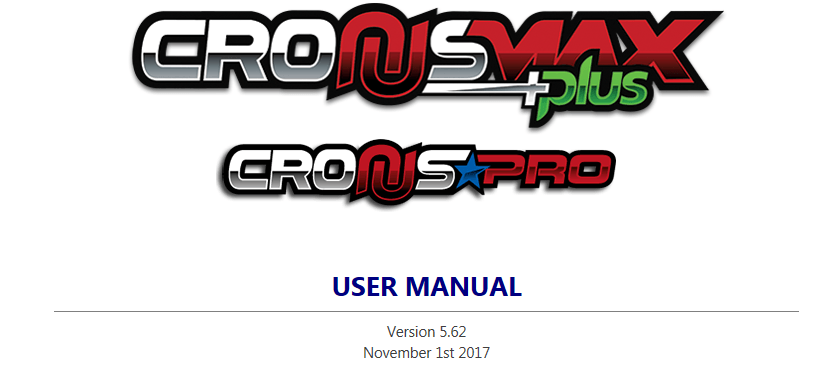
CRONUS MAX의 User Manual의 GPC Guide 부분을 번역했음. 번역하면서 느낀 건데, GPC 관련 매뉴얼이 좀 부실함. 우선 프로그래밍에 좀 익숙해야 이해가 되고, 예제가 부실함. 따라서, 사용자가 여러 테스트를 해 봐야 할 것임.

번역에 관련하여.

* 기본적으로 최대한 원문에 가깝게 해석을 하도록 노력 하였으나, 한국어로 매끄럽게 읽힐 수 있도록 어느 정도 의역을 하였음.
* 그러나, build나 compile과 같은 컴퓨터 과학 용어는 그대로 빌드, 컴파일과 같이 번역하였음
* 한국어는 단수/복수를 엄격히 구분하지 않고 심지어 문맥상으로 복수인지 단수인지를 이해하는 스타일 이므로 그냥 단수로 번역 하였음.
* 영어 원문으로 읽으면 이해가 직관적으로 되는 것들이, 다른 사람이 번역해 놓은 것을 읽으면 이해 안가는 것들이 있음. 그런 부분은 원문을 직접 읽어 볼 것.
* 근데 막상 번역하다 보니 귀찮아져서 나중 가서는 대충 했음.
* 해석 당시 USER MANUAL 버전은 아래 그림과 같이 Version 5.62임.



**GPC 안내서 (GPC Guide)**

GPC는 CronusMAX PLUS에서 사용되는 스크립트 언어이다. GPC 명령어(commands)는 명령어 자체가 그 기능을 설명하도록 디자인되어 있기 때문에, 초보자부터 전문 프로그래머까지 누구든 빠르게 익힐 수 있다. GPC의 기초 문법은 C언어를 차용했기 때문에, GPC의 기본 구조와 키워드들은 전에 C 프로그래밍을 해보았던 사람들에게 친숙할 것이다. 물론 C에는 없는 많은 고유한 GPC 함수들이 있지만, 그것들 역시 읽고 이해하기 쉽도록 디자인 되어 있다.

GPC는 단순히 말하면 CronusMAX PLUS에서 사용되는 코드의 사람이 읽을 수 있는 버전이다. CronousMAX는 실제로는 바이트코드 시스템을 사용한다. 당신이 프로그램을 작성하고, 그 스크립트를 빌드하거나 컴파일 할 때마다, Cronus PRO에 내장된 컴파일러는 스크립트를 장치로 보내기 전에 바이트코드로 전환한다.

바이트 코드는 CronusMAX PLUS가 작동시키는 가상머신(virtual machine, VM 또는 소프트웨어 해석기)을 위한 명령 세트(instruction set)이다. 컨트롤러(controller, 게임기 입력 패드나 조이스틱 등)의 입력(input)은 가상머신으로 보내지고, 그런 후 가상머신은 그 입력을 명령 세트에 따라 바이트 코드로 변환한다. 이렇게 변환된 출력(output)은 콘솔(console)로 보내진다. 가상머신은 가상의 컴퓨터(hypothetical computer)이며, 가상머신의 함수와 아키텍처는 컨트롤러 명령을 콘솔로 변환하고 출력하는 작업에 대해 특별히 최적화 되어 있다.

Cronus MAX의 가상머신은 stack machine이며 최적화 되어 있다. 따라서 지연(delay) 없이 콘솔로 데이터를 보낼 수 있다. CPU load는 80% 아래로 유지되며, CronusMAX PLUS의 CPU는 할당되는 모든 작업을 지연 없이 콘솔로 출력할 수 있다. 최적화 과정은 가상머신이 오류 검사를 최소로 하며, 딱 필요한 것만 한다는 의미이다. 이것은 프로그래머가 GPC 스크립트를 개발할 때 GPC의 제한(limitation)과 원칙(fundamental)을 잘 이해하는 것이 중요하다는 의미이다.

**CronusMAX PLUS 사양**

Real CPU: **Atmel Microcontroller @ 16MHz**

Real Memory: **8184 bytes**

Real Flash: **130944 bytes**

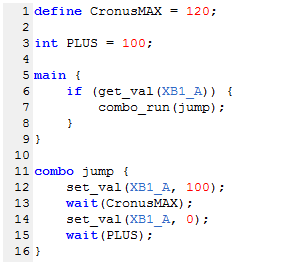
VM Type: **Stack Machine**

VM Max Bytecode: **4096 bytes**

VM Stack Memory: **128 words (256 bytes)**

**기초 문법 (Basic Syntax)**

GPC는 CronusMAX PLUS에서 사용되는 스크립트 언어이다.



**1. 명령 구분 (Instruction Separation)**

C처럼, GPC는 각각의 문(statement) 끝에 세미콜론(;)으로 끝나도록 해야 한다. 그러나 코드 block의 닫음 태그(여기서 닫음 중괄호 })는 자동적으로 세미콜론이 포함되는 것을 의미하기 때문에, GPC code block의 마지막 줄은 세미콜론이 없어도 된다.

**main** {

    sensitivity(XB1\_LY, NOT\_USE, 80);

    a = b \* ( c + 20 )

}

코드 block의 마지막 줄에 세미콜론을 안 넣어도 되지만, 나중에 코드를 확장할 때 실수하지 않으려면 항상 세미콜론을 쓰는 게 좋은 습관이다.

**2. 코드 내포하기 (Nesting Code)**

코드를 내포하는 것, 또는 logic block을 만드는 것은 코드를 함께 묶는 것이다. block은 {로 시작해서 }으로 끝난다. 이것은 그 문(statement)이 활성화되었을 때만 실행되는 의미로 {와 }으로 코드를 감싼다.

아래 예제를 보면,

**main**

{ //Main Start

**if**(get\_val(PS4\_R2))

    { //Block 1 Start

**if**(get\_val(PS4\_L2))

        { //Block 2 Start

            combo\_run(RAPID\_FIRE\_ADS);

        } //Block 2 End

**else**

        { //Block 3 Start

            combo\_run(RAPID\_FIRE);

        } //Block 3 End

    } //Block 1 End

} //Main End

Block 1이 활성화 되지 않으면 Block 2와 Block 3은 무시된다. 즉, R2 버튼을 누르지 않으면 아무 일도 안 일어난다. R2 버튼을 누르면, CronusMAX는 Block 2를 살펴보게 된다. 이때 L2 버튼이 눌려지면, RAPID\_FIRE\_ADS 콤보가 실행되며 Block 3는 무시된다. 그러나 L2 버튼이 눌려지지 않으면 Block 2가 무시되는 대신 Block 3에 있는 코드가 실행되며 따라서 RAPID\_FIRE 콤보가 실행된다.

아래 예제와 같이, 하나의 문(statement) 후에 코드가 한 줄만 있으면 암시적인 내포(nesting)를 의미한다.

**main** {

**if**(get\_val(XB1\_RT) > 95)

        combo\_run(RAPID\_FIRE);

}

컴파일 될 때, combo\_run(RAPID\_FIRE); 줄은 자동적으로 if 문 사이에 내포된다. if 문 다음에 두 줄 이상의 코드가 실행되게 하려면 반드시 {와 }을 사용해야 한다.

**3. 코드 주석 달기 (Commenting Code)**

주석(comment)은 컴파일러가 무시하는 문자들이다. 주석은 보통 향후 참고(future reference)에 대한 코드를 설명하거나 살펴볼 다른 코드에 대한 기록을 하는데 사용된다. 그러나, 주석은 스크립트를 디버깅할 때 특정 코드 부분을 비활성화 하는데도 사용된다. GPC 주석은 전에 C 프로그램을 사용해 본 경험이 있으면 같은 스타일이기 때문에 익숙할 것이다.

주석은 한 줄을 주석(Single Line Comment) 하는 것과 여러 줄을 주석(Multi Line Comment)하는 것의 두 가지 형태가 있다.

**한 줄 주석(Single Line Comment)**

한 줄 주석은 아래 예제와 같이 // (슬래시 두 개) 문자를 사용하며, 그 후에 임의의 문자들이 따라 온다.

**main** {

    // 한 줄 주석

**if**(get\_val(XB1\_RT) > 95)

        // 이것도 한 줄 주석

        combo\_run(RAPID\_FIRE);

}

**여러 줄 주석(Multi Line Comment)**

/\* (슬래시, 별표) 문자는 여러 줄 주석하기의 시작이고 이후에 일련의 여러 문자를 쓸 수 있다. 여러 줄 주석의 끝은 처음 발견되는 \*/ (별표, 슬래시) 이다.

아래 예제를 보자.

**main** {

    /\* 여러 줄 주석

    if(get\_val(XB1\_RT) > 95)

        combo\_run(RAPID\_FIRE);

    \*/

}

그러나, 아래 예제와 같이 여러 줄 주석을 이중으로 내포 될 수는 없다.

**main** {

    /\* 여러 줄 주석

    if(get\_val(XB1\_RT) > 95)

        combo\_run(RAPID\_FIRE); /\*이렇게 주석 달면 문제 발생\*/

    \*/

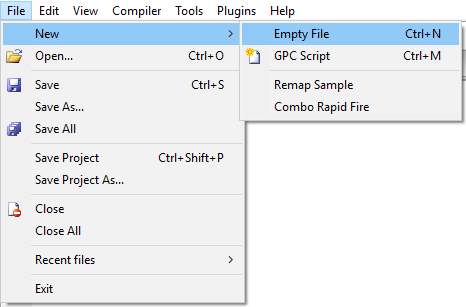
}

**간단한 사용지침 (A Simple Tutorial)**

본 기본 사용 지침은 당신의 첫 스크립트를 어떻게 작성하는지 와 그것을 어떻게 확장할 수 있는지 보여줄 것이다.

**단계 1**

Cronus PRO를 실행하고 File 메뉴를 클릭한 후, New → Empty File을 선택하라.



**단계 2**

Cronus PRO의 GPC code 편집기(editor)에 아래 코드를 타이핑 하거나 복사 및 붙여넣기를 하라.

**main**{

**if**(get\_val(XB1\_RT)){

        combo\_run(RAPID\_FIRE);

    }

}

**combo** RAPID\_FIRE{

    set\_val(XB1\_RT, 100);

**wait**(40);

    set\_val(XB1\_RT, 0);

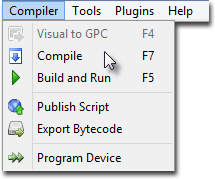
**wait**(30);

    set\_val(XB1\_RT, 0);

}

**단계 3**

오류를 체크하기 위해 코드를 컴파일 하라. 컴파일 하려면 키보드의 **F7**을 누르거나 Compiler 메뉴의 Compile을 선택하라.



그러면 GPC 편집기 아래쪽 출력 창(Output window)에 다음과 같은 메시지가 출력된다.

   ------ GPC: Build started ------

   > 1: New\* :

   > Bytecode size: 148 bytes (3.6%)

   > Stack memory: 3 words (2.7%)

   Build succeeded with 0 warnings...

PC PROG 포트 쪽으로 CronusMAX PLUS와 PC가 연결되어 있고, 컨트롤러도 연결되어 있으면 Build and Run 옵션을 사용해서 이 스크립트가 어떻게 동작하는지 볼 수 있다. **F5**를 누르거나 Compiler 메뉴의 Build and Run을 선택하라. Build and Run은 코드를 컴파일 한 후 CronusMAX PLUS에 컴파일 된 코드를 보내기 때문에 바로 그것을 테스트 할 수 있다.

**스크립트 분해(Script Breakdown)**

지금부터 설명하는 스크립트는 오른쪽 트리거 버튼이 값을 가지거나 또는 눌려졌을 때 언제든지 RAPID\_FIRE 라는 이름의 콤보를 실행한다. 콤보가 끝날 때 여전히 오른쪽 트리거를 누르고 있으면 그 콤보를 다시 시작한다.

CronusMAX PLUS가 어떻게 이것을 수행하는지 분석하기 위해, 우선 스크립트를 main과combo 두 섹션(section)으로 나눌 것이다.

**The Main Section**

**main**{

**if**(get\_val(XB1\_RT)){

        combo\_run(RAPID\_FIRE);

    }

}

**Basic GPC Structure**를 설명할 때 다시 나오지만, main 섹션은 CronusMAX PLUS에서 하나의 반복(loop)으로 실행한다. CronusMAX의 가상머신은 순서대로 코드를 실행하며, 코드의 끝에 도달하면, 데이터가 콘솔로 전달되고 그런 후 가상 머신은 다음 반복(next loop)을 시작한다.

**if**(get\_val(XB1\_RT))

위 코드는 CronusMAX에 if 안이 TRUE이면, 내포된 코드를 실행하라고 지시한다. if 안이 TRUE 이려면 XB1\_RT (오른쪽 트리거 버튼)이 0(zero)보다 큰 값을 가지는 것, 즉 XB1\_RT 버튼을 누른 것이고, if 안이 FALSE인 경우는 XB1\_RT 버튼을 땐 것이다.

                       {

        combo\_run(RAPID\_FIRE);

    }

위 코드는 **if** 문 사이에 내포된 것이다. Nesting code는 계층 구조를 만든다. 열림 중괄호 ({)로 시작해서 닫힘 중괄호(})로 끝난다. **if** 문 사이의 nesting code는 **if** 문이 TRUE 일 때만 실행되는 코드이다.

        combo\_run(RAPID\_FIRE);

위 줄은 CronusMAX에 RAPID\_FIRE 라는 이름을 갖는 **combo**를 실행하도록 지시한다. CronusMAX가 이 지시를 받았을 때, 이 **combo**가 이미 실행 중이면, **combo**를 실행하라는 이 지시는 실행되지 않는다는 점이 중요하다. RAPID\_FIRE combo가 끝났을 때만 그 **combo**가 다시 실행된다. 즉, 당신이 오른쪽 트리거 버튼을 누르고 있으면 이 코드는 활성화 되고 CronusMAX가 **combo**를 시작하며 그런 후 **combo**가 끝나자 마자 다시 시작한다는 것을 의미한다. 그러므로 오른쪽 트리거 버튼을 땔 때까지 그 combo가 계속 실행된다.

**The Combo Section**

**combo** RAPID\_FIRE{

    set\_val(XB1\_RT, 100);

**wait**(40);

    set\_val(XB1\_RT, 0);

**wait**(30);

    set\_val(XB1\_RT, 0);

}

위 코드는 오른쪽 트리거 버튼을 누를 때 CronusMAX가 실행하는 **combo**다. 실행될 때, **combo**는 **wait** 문을 만날 때까지 코드를 실행한다. wait 문은 그 위에 있는 명령어를 지정된 밀리초 동안 실행한다.

    set\_val(XB1\_RT, 100);

**wait**(40);

위 코드는 오른쪽 트리거의 값을 40 밀리초 동안 100 (즉, 완전히 누른 것에 해당)으로 설정한다.

    set\_val(XB1\_RT, 0);

**wait**(30);

40 밀리초가 지난 후, 위 코드는 오른쪽 트리거의 값을 30 밀리초 동안 0 (트리거 버튼을 땐 것)으로 설정한다.

**코드 확장하기(Expanding the code)**

이제 스크립트가 작동하는 방식을 이해 했으니, 스크립트를 약간 복잡하게 변경해 보자.

main section에서 아래 코드 부분을 보자.

**if**(get\_val(XB1\_RT)){

위 코드를 다음과 같이 바꾸어 보자.

**if**(get\_val(XB1\_RT) && !get\_val(XB1\_LT)){

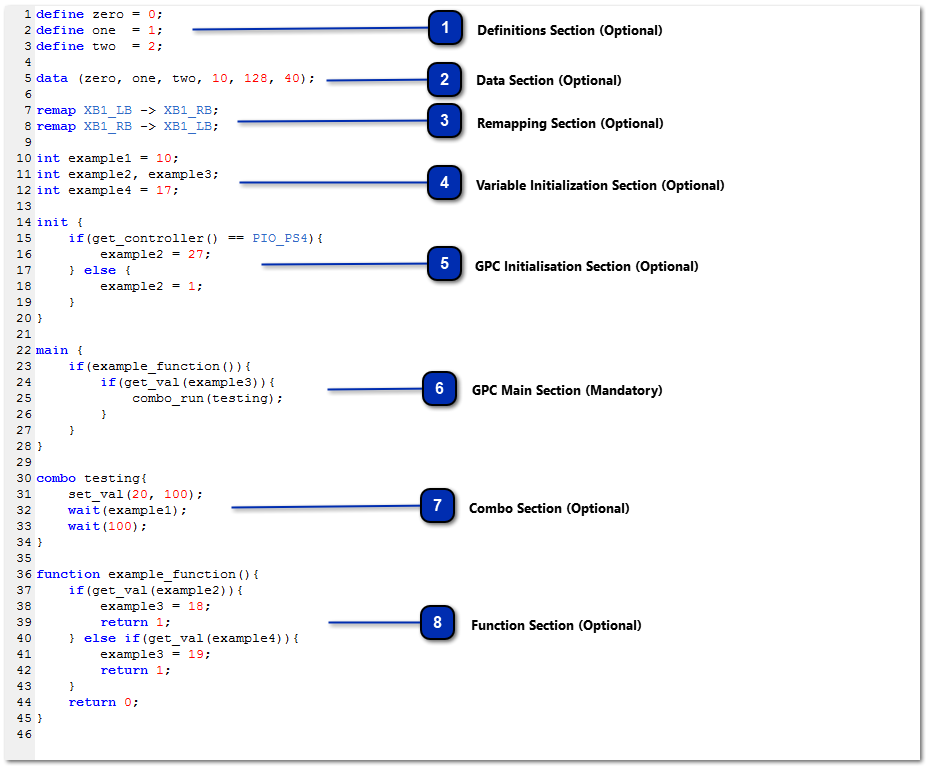
**if** 문에 추가된 &&과 !get\_val(XB1\_LT)를 도입한 것은 CronusMAX에 오른쪽 트리거는 0이 아닌 임의의 값을 가지면서 왼쪽 트리거는 0일 때만 combo가 실행되도록 지시하는 것이다.

&&는 ‘and’를 의미하고 !는 ‘not’을 의미한다. 그래서, **if** 문은 이제 ‘오른쪽 트리거는 0이 아닌 값을 가지면서 왼쪽 트리거는 0이면’을 의미한다. 즉, 이 코드는 게임 안에서 에임을 안 하면서(왼쪽 트리거를 누르지 않으면서) 오른쪽 트리거를 누르고 있을 때만 빠르게 사격(Rapid Fire)한다는 것을 의미한다.

Build and run을 사용하면 실제로 코드가 실행되어 동작하는 것을 확인할 수 있다.

**기본 GPC 구조 (Basic GPC Structure)**

GPC 스크립트는 섹션으로 분리된다. 8개 종류의 섹션이 있고, 각 섹션은 아래 예제와 같은 순서로 나열 되어야 한다. 스크립트를 작성할 때 main 섹션은 반드시 있어야 하며, main 섹션은 끊임없이 반복적으로 실행된다.



**1. Definitions Section (Optional)**



Definition 섹션은 단어(word)에 값을 할당하는데 사용되며, 따라서 명명된 상수(named constant)를 만든다. Definition은 반드시 **main** 섹션 전에 있어야 한다.

위 예제는 zero, one, two 라는 단어를 각각 0, 1, 2 값으로 정의한다. define으로 정의된 단어들은 고정된 값(static value)으로 실행 시간(run time)동안 값이 변경될 수 없다.

그래서, 스크립트 안에 ‘one’ 이라는 단어를 사용한 것은 숫자 ‘1’을 타이핑 한 것과 동일하다.

Definitions에 대한 더 자세한 것은 **Definitions** 카테고리를 참고 하라.

**2. Data Section (Optional)**



data 섹션은 GPC 바이트코드 안에 가상 주소 공간(virtual address space)의 시작에 위치하는 바이트 (8 bit unsigned integer) 배열이다. 그 값은 읽기 전용이고 실행 시간에 변경될 수 없다.

사용자는 0 기준 데이터 배열을 통해 이 값에 접근할 수 있다.

예를 들어 아래와 같은 방식으로 배열에 접근 할 수 있다.

example4 = **dbyte**(4);  //example4 = 128

좀 더 자세한 것은 **Data Section** 카테고리를 참고 하라.

**3. Remapping Section (Optional)**



Remapping 섹션에서는 조정 행위(behavior of the controls)을 바꿀 수 있다. 위 코드의 첫 번째 줄은 왼쪽 범퍼(XB1\_LB)에 할당된 값을 오른쪽 범퍼(XB1\_RB)에 보내도록 하는 것이고, 두 번째 줄은 그 역이다.

버튼 remap은 **main**이 끝난 후에 한번만 적용되며, 출력 보고(output report)가 콘솔로 보내지기 전에 적용된다는 것을 기억하라. 이것은 스크립트를 작성할 때 remapping을 고려하지 않고 작성해야 한다는 것을 의미한다.

예를 들어, 아래와 같이 스크립트 어느 지점에서 왼쪽 범퍼에 100을 지정했다면,

set\_val(XB1\_LB, 100);

**main** 과정이 끝날 때, 출력 보고에는 초기에 왼쪽 범퍼에 100의 값이 들어 있다. Remap이 처리될 때, 이 값은 오른쪽 범퍼에 대신 할당되며, 출력 보고는 수정된다. 최종 출력 보고가 콘솔로 보내질 때, 왼쪽 범퍼 대신 오른쪽 범퍼에 100의 값이 포함되어 있는 것이다.

더 자세한 내용은 **Remapping** 카테고리를 참고하라.

**4. Variable Initialization Section (Optional)**

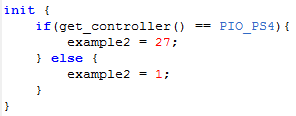


이 섹션은 사용자가 변수(variable)를 만들 수 있는 섹션이다. 하나의 변수는 stack memory안에 한 지점이고 거기에는 값이 위치한다. 변수는 **main** 또는 **init** 섹션 전에만 만들 수 있다. 이러한 변수는 전역(global)이며, 스크립트 어느 지점에서는 접근 할 수 있다. 변수는 고정(static)이 아니며, 따라서, 실행 시간 동안 값을 바꿀 수 있다.

Variable initialization 섹션에서 변수의 값을 할당하지 않으면, 0으로 초기화 된다. 위 코드에서 example2와 example3가 그 예이다.

더 자세한 사항은 **Variables** 카테고리를 참고하라.

**5. GPC Initialization Section (Optional)**

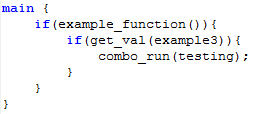


**init** 섹션은 가상머신에 스크립트가 적재될 때, 딱 한번만 실행된다는 것만 제외하고 **main**과 매우 유사하다. **init** 는 **main** 섹션에서와 똑같이 combo나 사용자 정의 함수 같은 명령어와 함수를 실행 할 수 있다.

**init**는 일반적으로 변수/배열 그리고 스크립트 셋업을 하는데 사용된다. 위 예제는 CronusMAX PLUS에 PS4 컨트롤러가 연결되어 있으면, example2에 27 값을 할당하고, 그렇지 않으면 1 값을 할당하라는 의미다.

더 자세한 사항은 **Init Section** 카테고리를 참고하라.

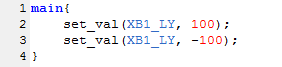
**6. GPC Main Section (Mandatory)**



**main** 섹션은 GPC 스크립트의 핵심이며, 모든 함수 및 콤보가 **main**에서 처음으로 실행된다. **main** 섹션만이 반드시 작성해야 하는 섹션이며, 어떤 GPC 스크립트이던 **main** 이 있어야 한다. 한 GPC 스크립트에 여러 **combo**나 여러 사용자 생성 **function**을 사용할 수 있는 것과는 달리, 한 GPC 스크립트에는 하나의 **main** 섹션만 있어야 한다. **main**은 반복적으로 실행된다. 가상머신은 순차적으로 코드를 실행하고 출력 보고를 생성한다. 가상머신이 **main**의 끝에 도달하면, 결과보고가 콘솔로 보내진다. 콘솔이 새 데이터를 요청하고, 출력 보고가 전달되고, **main**은 또 다른 실행을 시작한다.

명령어가 순차적으로 실행되기 때문에, 하나의 버튼에 두 번 이상 값을 설정하면, 마지막 설정한 명령어만 보내진다.

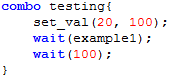
아래 예제를 보자.



XB1\_LY에 -100 값이 설정된 것으로 콘솔에 전달된다. 콘솔은 XB1\_LY 가 100으로 설정된 것은 파악할 수 없는데, 콘솔로 출력보고가 전달되기 전에 그 값이 수정되었기 때문이다.

더 자세한 사항은 **Main Section** 카테고리를 참고하라.

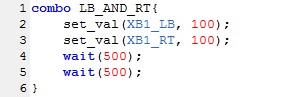
**7. Combo Section (Optional)**



**combo**는 명령어 세트가 순서대로 실행되는 함수이며, 명령어들 이후에 직접 wait 명령어를 지정하여 시간의 크기를 조절할 수 있다.

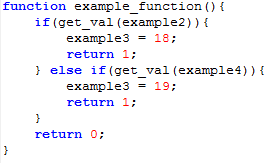
위 스크립트는 combo가 실행될 때, 지정자 20 (Xbox 컨트롤러에서는 X버튼이고, Playstation 컨트롤러에서는 네모 버튼)을 변수 example1에 지정된 밀리초 동안 100% (버튼을 완전히 누른 것)로 설정하고, 그런 후 100 밀리초 동안 아무것도 하지 않는 것이다.

하나의 wait 문 전에 여러 명령어를 지정할 수 있다. 아래 콤보 예제를 보면, 콤보가 실행되면, 왼쪽 범퍼와 오른쪽 트리거 둘다를 500 밀리초 동안 누른 후에, 500 밀리초동안 아무것도 하지 않는 것이다.



더 자세한 사항은 **Combo Section** 카테고리를 참고하라.

**8. Function Section (Optional)**



사용자가 만든 함수는 main 섹션과 유사하다. 명령어가 순차적으로 처리되고, main 섹션에 허용된 어떤 스크립트 명령어든 **function** 섹션에 사용할 수 있다. **function**은 반드시 GPC 스크립트의 끝에 위치해야 한다.

**function**의 다른 점은 **function**은 호출될 때만 실행되며 값을 반환할 수 있다는 것이다. function으로부터 값이 반환되면, **function**은 종료되며 그 지점 이후의 어떤 코드든 실행 되지 않는다.

GPC 사용자 함수는 전역(global)이며, 이것은 **init**, **main**, **combo** 에서 호출 될 수 있다는 것을 의미한다. 하나의 함수는 또 다른 함수 안에서도 호출 될 수 있다.

더 자세한 사항은 **User Created Functions Section** 카테고리를 참고하라.

**Definitions**

Definition의 유일한 목적은 값을 단어로 할당하는 것이다. 그러므로 스크립트를 사람이 더 읽기 쉽게 만든다. Definition은 스크립트의 bytecode 공간을 사용하지 않는다. 스크립트가 컴파일 될 때, definition문으로 정의된 단어들은 그 단어에 할당된 값으로 변경된다.

**문법**

**define** <name> = <value>;

* <name> : 상수의 이름
* <value> : 상수에 할당되는 값. 정수만 사용할 수 있다.

아래 예제와 같이 **define**으로 단어를 정의되고, 값을 할당하면, 값이 허용되는 어느 곳에서든 정의한 단어를 사용할 수 있다.

**define** my\_value = 50;

**int** myvar;

**main** {

    set\_val(XB1\_RT, my\_value);  // RT 를 50으로 지정

    myvar = my\_value;           //myvar 는 50과 같다.

//myvar가 50보다 크거나 같으면

**if**(myvar >= my\_value) {

        // 여기 실행

    }

//LT가 50보다 크면

**if**(get\_val(XB1\_LT) > my\_value) {

        //여기 실행

    }

}

아래 예제와 같이 **define**으로 정의한 단어는 고정된 값이고 따라서, 실행 시간동안 변경될 수 없다.

**define** my\_value = 50;

**main** {

    my\_value = 70;  // compiler error가 발생

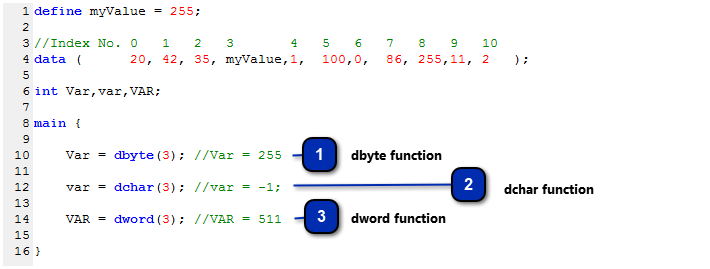
}

단어에 값을 할당하면서 실행 시간 동안 그 값을 변경하려면 **define** 대신 변수(**variable**)을 사용하라.

**Data Section**

**data** 섹션은 GPC 바이트코드의 가상 주소 공간의 처음 부분에 위치하며, 실행 시간 동안 변경될 수 없는 고정된 값을 포함한다.

**data** 섹션의 주 목적은 고정 정보를 저장하는 것이며, 크기는 안에 들어간 값들로 결정된다. 고정값들은 GPC에서 index를 사용해서 접근할 수 있고, data 섹션 전에 위치한 **definition**으로 고정값을 지정할 수 있다. 아래 예를 보자.



data 섹션에 위치한 값은 바이트 (8 bit unsigned integer)로 표현된다. Index는 0부터 시작이다. 위 예에서 볼 수 있는 것처럼, 첫 번째 값은 index가 0 (zero)이고, 11번째 값은 index가 10이다.

**1. dbyte function**



**dbyte** 함수는 지정된 index에 해당하는 배열 안의 byte 값(8 bit **unsigned** integer)을 반환한다.

**문법**

**dbyte** ( <index> );

**Parameters**

<index> : data 섹션 안의 한 원소의 index

**2. dchar function**



**dchar** 함수는 지정된 index에 해당하는 배열 안의 char 값(8 bit **signed** integer)을 반환한다.

**문법**

**dchar** ( <index> );

**Parameters**

<index> : data 섹션 안의 한 원소의 index

**3. dword function**



**dword** 함수는 지정된 index에 해당하는 배열 안의 word 값(**16** bit **signed** integer)을 반환한다.

**문법**

**dword** ( <index> );

**Parameters**

<index> : data 섹션 안의 한 원소의 index

**Remapping**

**remap**은 스크립트의 시작부분에 정의되며, **main** 전에 정의가 완료 되야 한다. **remap**은 실행 시간(run time) 동안 실행(excute)될 수 없다.

**remap**이 스크립트의 시작 부분에 정의 되지만, 가상머신은 현재 반복되고 있는 main 섹션이 끝나기 전까지 **remap**을 실행하지 않는다. 이것은 입력에 대해 remap되는 입력에 대한 어떤 스크립트 동작도 원래의 버튼에 대해서 **remap**을 고려하지 않으면서 프로그램 돼야만 한다는 것을 의미한다.

예를 들어, PS4\_CIRCLE와 PS4\_TRIANGLE 버튼을 바꾸기 위해 remap 명령어를 사용하고, 컨트롤러의 PS4\_CIRCLE 버튼을 combo의 시작 버튼으로 하기 원한다면, 아래와 같이 당신의 코드에 여전히 PS4\_CIRCLE 버튼을 사용해야 한다.

**remap** PS4\_CIRCLE -> PS4\_TRIANGLE;

**remap** PS4\_TRIANGLE -> PS4\_CIRCLE;

**main** {

//Input commands 는 remap에 영향 받지 않는다.

**if**(get\_val(PS4\_CIRCLE)) {

//따라서 이 combo를 실행하려면,

//물리적으로 CIRCLE button을 눌러야 한다.

        combo\_run(My\_Combo);

    }

}

**combo** My\_Combo{

    set\_val(PS4\_L3, 100);

**wait**(200);

    set\_val(PS4\_L3, 0);

**wait**(190);

}

그러나, main이 끝난 후에 remap이 처리되기 때문에, remap으로 지정한 출력 명령어는 remap이 된다는 것을 생각해야 한다. 따라서, remap을 이용해서 CIRCLE버튼을 눌렀을 때 TRIANGLE 버튼이 반복적으로 연타되도록 하는 예제가 아래와 같다. 아래 예제에서 input command의 값을 가져오는 get\_val 명령어는 remap에 영향이 없지만, output command의 값을 지정하는 set\_val 명령어로 지정된 값은 remap이 고려된다는 것을 생각해야 한다.

**remap** PS4\_CIRCLE -> PS4\_TRIANGLE;

**remap** PS4\_TRIANGLE -> PS4\_CIRCLE;

**main** {

**if**(get\_val(PS4\_CIRCLE)) {   //Input commands 는 remap에 영향 받지 않는다.

//따라서, 이 combo를 실행하려면, 물리적으로 CIRCLE 버튼을 눌러야 한다.

        combo\_run(My\_Combo);

    }

}

**combo** My\_Combo{

    set\_val(PS4\_CIRCLE, 100);   //combo가 실행되면, CIRCLE이 반복적으로 눌려진다.

**wait**(40);                   //그러나 현재 반복단계의 main이 끝나면

    set\_val(PS4\_CIRCLE, 0);     //가상머신은 remap을 실행하기 때문에

**wait**(30);                   //CIRCLE은 TRIANGLE로 바뀐다.

}

Remap은 input으로 입력된 값을 선택된 output값으로 지정한다. 따라서, 하나의 input으로 여러 input을 다룰 수 있다.

아래 예제를 보자.

**remap** PS4\_CROSS -> PS4\_SQUARE;

**remap** PS4\_CROSS -> PS4\_TRIANGLE;

**main** {

}

CROSS 버튼은 CROSS, SQUARE, TRIANGLE을 동시에 출력할 것이다. 따라서 SQUARE와 TRIANGLE 버튼은 더 이상 어떤 출력도 나오지 않는다.

**Commands and Syntax**

두 remap 명령어가 있다.

**remap**

**Remap: input identifier의 값을 output identifier에 지정**

**문법**

**remap** <input identifier> -> <output identifier>;

Remap은 아래 예제 처럼 output identifier가 동일하면 이전에 지정된 remap을 무시하고 마지막에 지정한 remap이 우선한다.



**unmap**

input으로 입력된 것을 output에 전달하지 않는다. 이것은 가상머신이 컨트롤러에서부터 받은 input을 인지하기는 하지만 output 보고를 통해 console로 전달하지 않는다는 의미이다. 따라서, unmap은 특정 버튼의 원래 기능을 console에 보내지 않으면서 combo의 시작버튼이나 code의 실행 버튼으로 사용할 수 있다. 아래 예제를 보자.

**unmap** PS4\_TRIANGLE;

**main** {

**if**(get\_val(PS4\_TRIANGLE)){

        combo\_run(MyCombo);

    }

}

다음과 같이 ALL\_REMAPS 상수를 이용하면 한번에 모든 remap을 unmap할 수 있다.

**unmap** ALL\_REMAPS;

**Variables**

GPC의 모든 변수는 16 bit signed integer다. GPC는 소수를 지원하지 않으며 소수점 이하는 0로 처리한다. 즉, 3.4는 3이 된다.

16 bit signed 는 변수가 -32768 부터 +32767까지 범위의 정수로 저장 할 수 있다는 의미다.

**변수 선언하기 (Declaring Variables)**

변수는 가상머신 메모리에 저장될 수 있는 데이터의 한 위치이다. 변수명은 밑줄(underscore, \_)이나 문자로 시작하며, 다음부터는 문자, 숫자, 밑줄로 조합할 수 있다. 대소문자를 가리기 때문에 cronusmax, Cronusmax, CRONUSMAX는 전부 다른 변수명이다.

GPC에서 이런 방식으로 정의된 변수는 global이며 따라서 변수는 **combo** 또는 **function** 뿐 아니라 **init** 또는 **main**에서도 접근 및 값의 변경이 가능하다. 사용자가 만든 함수에서 선언한 변수만이 local이다.

Global 변수는 **main** 또는 **init** 섹션 전에 선언 되야 만 하며, 따라서 **main**이나 **init** 이후 또는 안에 선언 할 수 없다. 다음 예제를 보자.

**int** myVar = 100, MYVar;

**int** MYVAR = -40;

**init** {

**int** incorrect; //이것은 error를 발생시킨다.

}

**int** Incorrect; //역시 error를 발생시킨다.

**main** {

**int** INCORRECT; //이것도 error를 발생시킨다.

}

변수는 항상 값이 할당된다. 선언 시 값이 할당되지 않으면, 0(zero)값으로 초기화 된다. 그리고 그 값들은 실행 시간 동안 변경될 수 있다. 아래 예제를 보자.

// myVar의 초기값은 100, MYVar 초기값은 0

**int** myVar = 100, MYVar;

// MYVAR 초기값은 -40

**int** MYVAR = -40;

**main** {

    MYVar = myVAR - MYVAR;  // 여기서 MYVar 값은 60이 된다.

}

**불 변수 (Boolean Variables)**

C# 같은 다른 프로그래밍 언어와는 달리, GPC는 불 값(논리 값)을 위한 변수 형태는 따로 없다. GPC에서는 정수로 논리 표현을 지원하도록 디자인 되어 있다. **TRUE**와 **FALSE** 키워드는 각각 1과 0으로 할당되도록 되어 있다. 아래 예제를 보자.

**int** myvar, MYVAR;

**main** {

    myvar = TRUE;   //myvar = 1

    MYVAR = FALSE;  //MYVAR = 0;

}

따라서, 아래와 같이 특정 코드를 활성화 하거나 비활성화하는 스위치를 만드는데 응용할 수 있다.

**int** myvar;

**main** {

  //A버튼을 누르면

**if**(event\_press(XB360\_A)) {

  //myvar에는 not myvar가 할당된다.

//따라서 myvar가 TRUE 이거나 0이 아닌 어떤 값을 가지면 myvar에 0이 할당된다.

//따라서 myvar가 FALSE 이거나 0값을 가지면 myvar에 1이 할당된다.

        myvar = !myvar;

    }

  // myvar 가 0이 아닌 값을 가지면

**if**(myvar) {

        //여기를 실행

    }

}

위 예제를 보면 myvar 의 값이 0이 아니면 if(myvar)는 항상 TRUE가 된다. 그러나 아래와 같이 변수의 값이 특정 값을 가질 때만 활성화 되는 nested code를 설계할 수도 있다.

**int** myvar;

**main** {

 //A 버튼을 누르면

**if**(event\_press(XB360\_A)) {

        myvar = myvar + 1;  //myvar 변수에는 기존 myvar값에 1이 더해진 값이 할당

    }

 // X 버튼을 누르면

**if**(event\_press(XB360\_X)) {

        myvar = myvar - 1;  //myvar 변수에는 기존 myvar값에 1이 빼진 값이 할당

    }

 //mvar 의 값이 2면

**if**(myvar == 2) {

        //여기 실행

    }

  // mvar 의 값이 4보다 크면

**if**(myvar >= 4) {

        //여기 실행

    }

  // mvar 의 값이2보다 작으면

**if**(myvar < 2) {

        //여기 실행

    }

}

**배열 (Arrays)**

배열은 하나의 명령어로 여러 변수들을 만드는 방법이다. 배열의 index는 0에서 시작하기 때문에 아래 예제와 같이 원소가 5개인 배열의 index는 0부터 4까지 이다.

**int** myvar[5];

**main** {

    //int myvar[5]는 5 개 변수를 생성

    myvar[0];   // 첫 번째 변수

    myvar[1];   // 두 번째 변수

    myvar[2];   // 세 번째 변수

    myvar[3];   // 네 번째 변수

    myvar[4];   // 다섯 번째 변수

}

배열의 모든 값은 초기화 될 때 0으로 할당되며, **init** 섹션 전에 다른 값으로 할당 할 수 없다. 배열 역시 global 이며, 따라서 실행 시간 동안 어느 지점에서든 값을 할당하거나 변경할 수 있다. 아래 예제를 보자.

**int** myvar[5];

**init** {

    myvar[0] = 20;  //myvar[0]는 20

    myvar[1] = 10;  //myvar[1]는 10

}

**main** {

  //myvar[2]는 30 (20 + 10)

    myvar[2] = myvar[0] + myvar[1];

    myvar[3] = 15;  //myvar[3]은 15

//myvar[4]는 45 (30 + 15)

    myvar[4] = add\_two\_variables(myvar[2], myvar[3]);

}

**function** add\_two\_variables(\_1st\_variable, \_2nd\_variable) {

**return** \_1st\_variable + \_2nd\_variable;

}

**배열을 통해 임의의 변수에 접근하기 (Accessing any variable via an array)**

스크립트가 컴파일 될 때, 모든 global 변수는 배열 안에 위치하게 된다. 따라서 배열로 초기화 되지 않은 변수에서 그 다음 변수들을 호출하는데 배열을 사용할 수 있다. 예제를 보자.

**int** myvar;

**int** \_1st = 10;

**int** \_2nd = 20;

**int** \_3rd = 30;

**int** \_4th = 40, \_5th = 50;

**int** var\_to\_place\_value\_into;

**main** {

  //var\_to\_place\_value\_into는 10 (\_1st)

    var\_to\_place\_value\_into = myvar[1];

//var\_to\_place\_value\_into 는 20 (\_2nd)

    var\_to\_place\_value\_into = myvar[2];

//var\_to\_place\_value\_into 는 60 ((40 + 50) - 30)

    var\_to\_place\_value\_into = (myvar[4] + myvar[5]) - myvar[3];

//var\_to\_place\_value\_into 는 30 (\_3rd)

    var\_to\_place\_value\_into = \_2nd[1];

}

**Init Section**

GPC의 **init** 섹션은 반복적으로 실행되지 않는다는 점만 빼고 **main** 섹션과 동일하다. **init** 섹션은 스크립트가 처음 적재된 후 **main** 섹션의 첫 번째 반복 전에 한번만 실행되며, **combo** 나 **function**을 호출할 수 있다. 또한 **main** 섹션과 마찬가지로 **init**에서 변수의 값을 변경할 수 있다.

init 섹션은 스크립트의 셋업을 하는데 사용된다. 예를 들어, Playstation 3와 Xbox One에서 같은 Rapid Fire 스크립트를 사용하면서 Playstation 3에서는 발사 버튼으로 범퍼 버튼을, Xbox One에서는 발사 버튼으로 트리거 버튼을 사용하길 원한다고 하자. 그러면 다음 예제와 같이 **init** 섹션에서 그것을 자동으로 조정할 수 있다.

**int** FIRE\_BTN;

**init** {

**if**(get\_console() == PIO\_PS3) {  //PS3에 연결되어 있으면

        FIRE\_BTN = 3;   //3 = LB/L1

    } **else** {                        // PS3가 아닌 다른 콘솔이면

        FIRE\_BTN = 4;   //4 = RT/R1

    }

}

**main** {

**if**(get\_val(FIRE\_BTN)) {

        combo\_run(Rapid\_Fire);

    }

}

**combo** Rapid\_Fire {

    set\_val(FIRE\_BTN, 100);

**wait**(50);

    set\_val(FIRE\_BTN, 0);

**wait**(40);

    set\_val(FIRE\_BTN, 0);

}

**init** 섹션은 실행 시간 동안 값을 지속적으로 재호출(recall)하길 원하지 않을 때 사용하는 persistent 변수를 사용할 때 매우 유용하다.

**Main Section**

**main** 섹션은 gpc 스크립트의 가장 중요한 부분이다. main은 지속적으로 반복되며, 실행 시간 동안 진행된 코드는 다시 처음으로 돌아간다. 따라서, main 섹션은 반드시 작성해야 하며, main없은 GPC 스크립트는 허용되지 않는다.

**main** 섹션은 아래와 같이 {으로 시작해서 }으로 끝난다.

**main**

{   //Main Start

}   //Main End

**main** 섹션이 마지막 지점에 도달하면, 콘솔에 전달할 출력 보고(output report)가 만들어 지며, **remap**이 처리되고, 그런 후 콘솔에 보고를 전달한다. 그리고 **main** 섹션은 처음부터 다시 시작한다.

**main** 섹션 안에 코드는 작성된 순서대로 실행된다. GPC 스크립트를 작성할 때 가장 중요하게 기억해야 할 점은 콘솔에 전달할 결과 보고가 **main** 섹션이 끝날 때 만들어 진다는 것이다. 그래서 아래와 같이 작성하면 기대와 다른 결과가 나올 수 있다.

**int** press\_lt;

**main**{

**if**(get\_val(XB1\_RT)){        //RT를 누르면

        press\_lt = TRUE;        //변수 'press\_lt'를 TRUE로 지정

    }

    press\_lt = FALSE;           // 변수 'press\_lt'를 FALSE로 지정

**if**(press\_lt) {              //이 라인은 'press\_lt'가 TRUE인 상태를 절대 만나지 못함.

        set\_val(XB1\_LT, 100);   //따라서, 이 라인이 절대 실행 안됨

    }

}

위와 같이, RT 버튼을 누르면 press\_lt 변수가 **TRUE**가 되지만, 그 즉시 바로 아래 코드가 실행되면 **FALSE**가 된다. 그러므로 if(press\_lt)는 항상 **FALSE**가 되고 if 문 안에 있는 내포 코드는 절대 실행되지 않는다.

그러나 코드를 아래와 같이 수정하면 원하는 결과가 나온다.

**int** press\_lt;

**main**{

    press\_lt = FALSE;           // 변수 'press\_lt'를 FALSE로 지정

**if**(get\_val(XB1\_RT)){        // RT를 누르면

        press\_lt = TRUE;        // 변수 'press\_lt'를 TRUE로 지정

    }

**if**(press\_lt) {              // 이 라인은 'press\_lt'가 TRUE가 되면

        set\_val(XB1\_LT, 100);   //여기를 실행하게 된다.

    }

}

RT 버튼이 눌러지면, 이미 **FALSE**로 설정된 press\_lt 의 값이 **TRUE**로 변경되고, 따라서 if(press\_lt) 안의 내포 코드가 실행된다. RT 버튼을 누르지 않으면 press\_lt 변수는 TRUE가 되지 않기 때문에 if(press\_lt)에 내포된 코드는 실행되지 않는다.

아래 코드와 같이 ‘press\_lt = FALSE;’ 코드를 else 문을 써서 하는게 더 좋은 방법으로 보인다.

**int** press\_lt;

**main**{

**if**(get\_val(XB1\_RT)){        // RT를 누르면

        press\_lt = TRUE;        // 변수 'press\_lt'를 TRUE로 지정

    } **else** {

        press\_lt = FALSE;       // 변수 'press\_lt'를 FALSE로 지정

    }

**if**(press\_lt) {              // 이 라인은 'press\_lt'가 TRUE가 되면

        set\_val(XB1\_LT, 100);   //여기를 실행하게 된다.

    }

}

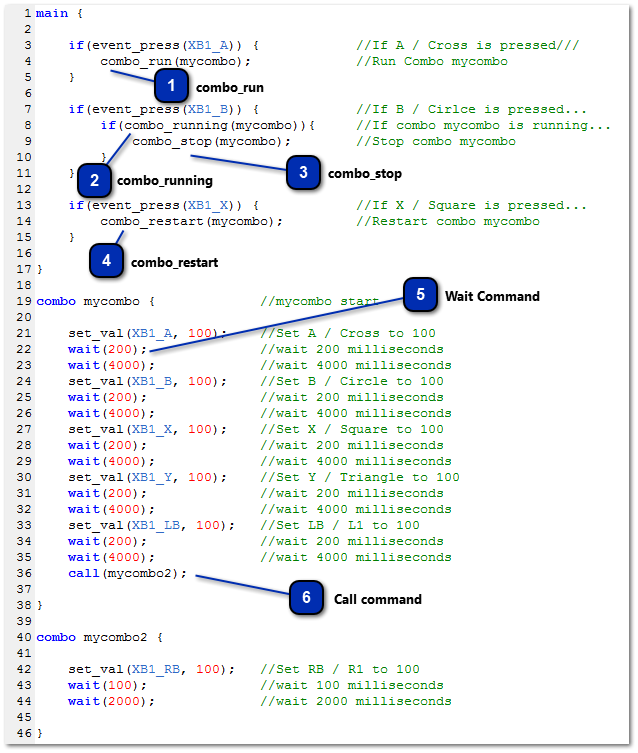
위 코드 역시 RT/버튼을 누를 때만 press\_lt 의 값이 **TRUE**가 되기 때문에 기대한 대로 동작할 것이다. 그러므로 press\_lt 변수는 RT 버튼을 안 눌렀을 때만 **FALSE**로 설정된다.

위 예제들에서 살펴본 것처럼, 코드는 작성한 순서대로 실행되며, 코드의 한 라인이라도 간단히 위치를 이동시키는 것에 의해 콘솔에 전달되는 출력(output)의 효과가 클 수 있다는 것을 기억하는 것이 중요하다.

**Combo Section**

**combo**는 순차적으로 실행될 지시들(instructions)의 조합이다. **main**과 마찬가지로, **combo**안의 명령어는 작성된 순서대로 수행된다. **main**에서 그랬던 것처럼 **combo**에서도 **function** 호출이나, 변수 값 지정 등을 할 수 있지만, 일반적으로는 필요하지 않으며, 이런 것을 하는 것은 stack memory나 bytecode 공간의 낭비만을 가져온다. **combo**는 **wait** 명령어를 사용함으로써 특정 길이의 시간 동안 버튼을 토글(눌렀다 때는)하도록 출력을 설정하는데 이상적이다. **wait** 명령어는 **combo** 안에서만 쓸 수 있고, 다른 곳에서는 사용할 수 없다.

**combo**의 이름은 variable의 이름을 정하는 규칙과 같다.



**1. combo\_run**



combo\_run은 주어진 이름의 combo를 실행한다. combo\_restart 명령과는 달리, combo가 현재 진행 중이면 효과가 없다. 즉, combo\_run은 주어진 이름의 combo가 안 돌아갈 때만 그 combo를 시작한다.

**문법**

**combo\_run** ( <Combo Name> );

**Parameters**

<Combo Name> : combo에 지정된 이름

**2. combo\_running**



combo\_running은 지정된 이름의 combo가 진행 중인지 아닌지를 체크할 때 사용하는 함수이다. 지정된 이름의 combo가 진행 중이면, TRUE를 반환하고 아니면 FALSE를 반환한다.

**문법**

**combo\_running** ( <Combo Name> );

**Parameters**

<Combo Name> : combo에 지정된 이름

**Returns**

Combo가 진행 중이면 TRUE, 아니면 FALSE

combo\_running는 combo가 끝났을 때만 특정 코드가 실행되길 원할 때 매우 유용하다. 아래 예제는 버튼 하나를 눌렀을 때 combo가 5번 실행되게 한다.

**int** run\_combo = 0;

**main** {

**if**(event\_press(19)) {       // XB1의 A나 PS4의 Cross를 누르면

        run\_combo = 5;          //변수 'run\_combo'의 값은 5가 된다.

    }

// 'run\_combo' 가 0이 아닌 값이면서 mycombo 가 실행 중이 아니면

**if**(run\_combo && !combo\_running(mycombo)) {

        run\_combo = run\_combo - 1;  //'run\_combo' 변수의 값을 1 뺀다.

        combo\_restart(mycombo);     // 그리고 mycombo를 재 시작한다.

    }

}

**combo** mycombo {

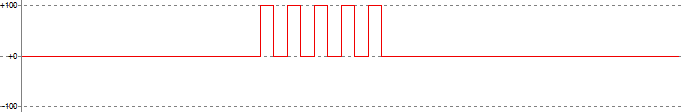
    set\_val(3, 100);    // RB 또는 R1 을 100으로 지정 (누른 상태)

**wait**(200);          // 200 milliseconds 를 누른 채로 기다리기

**wait**(200);          // 200 milliseconds 기다리기

}

위 코드의 장치 모니터 결과이다.



위 그림처럼, 코드의 결과는 A/Cross 버튼을 한 번 누르면 RB/R1 버튼을 5번 누른 게 된다. 위 코드처럼, combo\_running 함수와 변수를 사용 했을 때 약간의 코드로 여러 버튼을 누르도록 만드는 것이 가능하다. ‘run\_combo = 5;’ 에서 숫자를 0부터 32767까지 변경할 수 있다. 그러나 정의한 수만큼 콤보가 많이 실행될 것이다.

**3. combo\_stop**



이름처럼, combo\_stop은 현재 진행 중인 combo를 멈춘다. combo\_run과 마찬가지로 현재 combo가 진행 중이 아니면, 효과가 없다.

**문법**

**combo\_stop** ( <Combo Name> );

**Parameters**

<Combo Name> : combo에 지정된 이름

combo\_stop은 버튼을 누르고 있을 때만 combo가 실행되기를 원할 때 특히 유용하다. 아래 예제를 보자.

**int** run\_combo = 0;

**main** {

**if**(get\_val(19)) {           //A또는 Cross를 누르고 있으면

        combo\_run(mycombo);     //mycombo를 실행

    } **else** {                    //A또는 Cross를 누르지 않거나, 때고 있으면

        combo\_stop(mycombo);    //mycombo를 멈춤

    }

}

**combo** mycombo {

    set\_val(3, 100);    // RB 또는 R1 을 100으로 지정 (누른 상태)

**wait**(200);          // 200 milliseconds 를 누른 채로 기다리기

**wait**(200);          // 200 milliseconds 기다리기

}

**4. combo\_restart**

combo\_restart는 진행 중인 콤보를 재시작한다. 지정한 콤보가 현재 진행 중이면, 처음부터 다시 시작한다. 지정한 콤보가 진행 중이 아니면, 콤보가 시작된다.

**문법**

**combo\_restart** ( <Combo Name> );

**Parameters**

<Combo Name> : combo에 지정된 이름

**5. Wait Command**



wait 명령어는 CronusMAX 안의 가상머신에게 마지막 명령어 세트를 얼마나 오랫동안 실행할지 지정하는 명령어이다. 지정하는 시간은 밀리초이며, 1밀리초에서 32767밀리초(1밀리초에서 32초)까지 지정할 수 있다.

wait 에 지정된 시간 동안 실행되는 명령어는 현재 **wait**와 이전 **wait**사이에 위치하거나, 현재 **wait**와 이전 **call** 명령어, 또는 현재 **wait**와 **combo**의 시작 사이에 위치한다. 아래 예제를 보자.

**combo** mycombo {

    set\_val(19, 100);//¯¯|

    set\_val(18, 100);//  | 18번과 19번에 해당하는 두 버튼이

**wait**(1000);      //←\_| 1000 milliseconds (1 second)동안 누른 채로 유지된다.

    set\_val(3, 100);//¯¯|

    set\_val(7, 100);//  | 3번과 7번에 해당하는 두 버튼이

**wait**(1500);     //←\_| 1500 milliseconds (1.5 second)동안 누른 채로 유지된다.

    set\_val(9, -100);//¯¯| 9번에 해당하는 축이

**wait**(2000);      //←\_| 2000 milliseconds (2 second)동안 -100값으로 유지된다.

}

**wait** 명령어는 combo 안에서만 사용할 수 있고 combo 블록의 첫 번째 수준에서만 사용할 수 있다. 내포는 할 수 없다. 아래 예제를 보자.

**main** {

**wait**(100);  //잘못된 사용 – combo문 이외에서는 사용할 수 없다.

}

**combo** mycombo {

**if**(get\_console() == PIO\_PS4) {

**wait**(200);  //잘못된 사용 – combo block의 첫번째 수준에서만 쓸 수 있다.

    }

**wait**(400);      //올바른 사용

}

**function** myfunction() {

**wait**(300);      //잘못된 사용 – 역시 combo문 안에서만 사용할 수 있다. 에러 발생.

}

**문법**

**wait** ( <Time> );

**Parameters**

<Time> : 마지막 명령어를 실행한 밀리초 단위 시간, 범위는 10 ~ 4000

**6. Call command**



wait와 같이, call은 combo 안에서만 쓸 수 있고, combo의 첫 번째 수준에서만 사용할 수 있다.

**call** 명령어가 사용되면, 현재 **combo**는 잠시 멈추고, **call** 에 지정된 **combo**가 실행된다. **call** 에 지정된 **combo**가 끝나면, **call** 명령어가 포함된 **combo**가 다시 시작한다. 아래 예제를 보자.

**combo** \_1st\_combo {

    set\_val(XB1\_B, 100);

**wait**(100);

**wait**(200);

**call**(\_2nd\_combo);   //\_2nd\_combo 가 끝날 때까지 \_1st\_combo 는 멈춤

    set\_val(XB1\_Y, 100);

**wait**(100);

**wait**(200);

}

**combo** \_2nd\_combo {

    set\_val(XB1\_RB, 100);

**wait**(100);

**wait**(200);

}

위 예제는 \_1st\_combo가 실행 될 때, 아래 명령이 콘솔에 차례대로 보내진다.:

* B/Circle을 100 밀리초 동안 누른다.
* 200 밀리초 동안 아무 버튼도 누르지 않는다.
* RB/R1을 100 밀리초 동안 누른다.
* 200 밀리초 동안 아무 버튼도 누르지 않는다.
* Y/Triangle 을 100 밀리초 동안 누른다.
* 200 밀리초 동안 아무 버튼도 누르지 않는다.

**call** 명령어는 **combo**안에, 지정된 지점에서 **combo**를 주입하는 것이다. 동일한 동작을 하는 두 개의 콤보를 설정할 때 스크립트의 공간을 낭비하지 않고 싶을 때 **call** 명령어는 유용하다.

**문법**

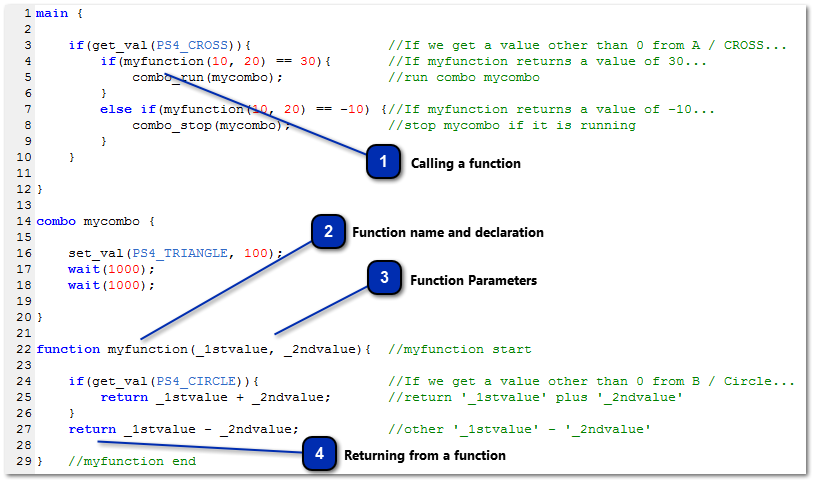
**call** ( <Combo Name> );

**Parameters**

<Combo Name> : 호출될 combo의 이름

**User Created Functions**

GPC는 많은 내장 함수(built in functions) 뿐 아니라, 사용자 스스로 함수를 만들 수 있다. function은 main 섹션 안에서 허용되는 어떤 코드도 동일하게 작성할 수 있으며, 역시 그 코드들이 정의된 순서대로 실행된다.



**1. 함수 호출하기 (Calling a function)**



function을 호출하려면, 간단히 그 function의 이름을 쓰고, (과 )사이에 적절한 parameter를 넣어주면된다.

function이 호출되면, function안의 코드가 실행되며 return 값이 그 function을 호출한 곳으로 보내진다.

사용자 function은 전역 범위(global scope)이기 때문에, function은 init, main, combo 섹션에서 호출될 수 있다. function은 또 다른 function안에서도 호출 될 수 있다. 그런, GPC는 function의 재귀 호출은 지원하지 않는다. 즉, 특정 function에서 그 자신의 function을 호출할 수 없다는 뜻이다.

**2. 함수 이름과 선언 (Function name and declaration)**



function을 선언하기 위해, function을 타이핑 하고 그 다음 그 function의 이름과 ()를 타이핑 한다. 괄호 ()안에는 그 function이 가져야 할 적절한 parameter의 이름을 써야 한다. Function의 이름과 parameters는 variable과 동일한 규칙을 따라야 한다. 즉, 시작은 밑줄(\_) 또는 문자로 시작해야 하며, 그 후로는 문자와 숫자, 밑줄 조합으로 쓸 수 있다.

Function은 main가 combo 섹션을 정의한 후 GPC 스크립트의 마지막에 선언(declare)되야 만 한다. (주: 사실 여기서 선언의 의미는 C에서 함수 정의의 의미이다. GPC에서 함수 원형 선언 같은 것은 없다. 원문 자체가 declare로 되어 있으므로 그냥 선언으로 해석하였다.)

**문법**

**function** function\_name( <Parameter(s)> );

**Parameters**

<Parameter(s)> : Optional parameters.

원하는 수 만큼 정의하거나 또는 없어도 된다.

각 파라메터는 콤마(,)로 구분 되야 한다.

**3. 함수 파라메터 (Function Parameters)**



함수 파라메터는 function의 바깥에서는 접근할 수 없기 때문에 지역 변수(local variable)라고 생각 할 수 있다. 값이 function 안으로 전달되고, function 안에서 변수처럼 사용할 수 있다.

GPC는 16bit 정수 형식의 데이터만 지원하기 때문에, function 안에서 파라메터의 데이터 형식을 지정할 필요가 없으며 파라메터의 이름은 function과 variable과 같이 동일한 규칙을 따른다. 시작은 밑줄(\_) 또는 문자로 시작해야 하며, 그 후로는 문자와 숫자, 밑줄 조합으로 쓸 수 있다.

Function 파라메터는 optional이다. 필요 없으면 정의하지 않아도 된다. 예를 들어 function myfunction () {}은 허용되는 표현이다.

**4. 함수로부터 리턴하기 (Returning from a function)**



return은 함수에만 있는 명령어이다. 모든 사용자 함수가 return값이 꼭 필수적인 것은 아니다. 함수 안에서 return이 없으면, 그 함수는 자동으로 0(zero)를 리턴한다.

함수 안에서 여러 return 지점을 가질 수 있다. 첫 번째 return명령어가 실행되면, function은 그 function이 호출된 곳으로 값을 리턴 하며, function은 종료된다. function에서 리턴된 지점 이후의 코드는 실행되지 않는다.

값을 리턴하는 것은 코드 섹션을 실행 할지 말지를 판단하는 불 값(Boolean value)으로 사용될 수 있는 면에서 유용하다. 다음 예제를 보자.

**int** RF\_HOLD = 40;

**int** RF\_NULL = 30;

**main** {

**if**(myfunction()) { //myfunction 0이외의 값을 리턴하면

**if**(get\_val(XB1\_RT)) { //RT를 누르고 있으면

            combo\_run(Rapid\_Fire);  //Rapid\_Fire 콤보 실행

        }

    }

}

**combo** Rapid\_Fire {

    set\_val(XB1\_RT, 100);

**wait**(RF\_HOLD);

    set\_val(XB1\_RT, 0);

**wait**(RF\_NULL);

    set\_val(XB1\_RT, 0);

}

**function** myfunction() {

**if**(get\_val(XB1\_VIEW)) { // View 버튼을 누르고 있으면

**if**(get\_val(XB1\_A))

            RF\_HOLD = adjust\_speed(RF\_HOLD, 10, 1000, 10);

**if**(get\_val(XB1\_B))

            RF\_NULL = adjust\_speed(RF\_NULL, 10, 1000, 10);

        set\_val(XB1\_A, 0);

        set\_val(XB1\_B, 0);

        set\_val(XB1\_LB, 0);

        set\_val(XB1\_RB, 0);

        set\_val(XB1\_VIEW, 0);

        set\_val(TRACE\_1, RF\_HOLD / 10);

        set\_val(TRACE\_2, RF\_NULL / 10);

**return** 0; //Return 0

    }

**return** 1; // View을 안 누르고 있으면, return 1

}

**function** adjust\_speed(var, min\_value, max\_value, adjustment\_increment) {

**if**(event\_press(XB1\_RB) && var < max\_value)

        var = var + adjustment\_increment;

**if**(event\_press(XB1\_LB) && var > min\_value)

        var = var - adjustment\_increment;

**return** var;

}

**위 예제가 어떻게 작동하는지 이해하기(Understanding how the example works)**

GPC 스크립트가 처음 적재(load)되면, 두 변수 RF\_HOLD, RF\_NULL이 각각 40과 30의 값을 가지고 만들어 진다.

그런 후 main 섹션의 첫 번째 반복이 시작된다. 아래 라인에 도달하게 되면,

**if**(myfunction()) {

function인 ‘myfunction()’이 실행된다.

그런 후, ‘myfunction()’안에 코드가 실행된다. XB1\_VIEW 버튼이 눌러져 있지 않으면 아래 if 문 안의 값이 FALSE이기 때문에 if 아래 내포 코드는 무시된다.

**if**(get\_val(XB1\_RT)) { //RT를 누르고 있으면

따라서 function 안에 실행된 다음라인은 아래와 같다.

**return** 1; // View을 안 누르고 있으면, return 1

이 지점에서 값 1이 리턴 되기 때문에

**if**(myfunction()) {

위 문장은 TRUE가 되고, 다음 코드가 실행된다.

**if**(get\_val(XB1\_RT)) { //RT를 누르고 있으면

            combo\_run(Rapid\_Fire);  //Rapid\_Fire 콤보 실행

        }

main 섹션은 main 섹션의 끝까지 실행하고, 계속 반복한다.

그러나, ‘myfunction()’이 실행될 때, XB1\_VIEW가 눌러져 있으면 다음 코드가 실행된다.

**if**(get\_val(XB1\_A))

            RF\_HOLD = adjust\_speed(RF\_HOLD, 10, 1000, 10);

**if**(get\_val(XB1\_B))

            RF\_NULL = adjust\_speed(RF\_NULL, 10, 1000, 10);

        set\_val(XB1\_A, 0);

        set\_val(XB1\_B, 0);

        set\_val(XB1\_LB, 0);

        set\_val(XB1\_RB, 0);

        set\_val(XB1\_VIEW, 0);

        set\_val(TRACE\_1, RF\_HOLD / 10);

        set\_val(TRACE\_2, RF\_NULL / 10);

**return** 0; //Return 0

위 코드에서 볼 수 있듯이, XB1\_A 또는 XB1\_B가 눌려져 있지 않아도 몇 개 버튼의 값이 0으로 지정되고, 두 변수의 값이 TRACE 값으로 써지고, 아래와 같이 가장 중요한 라인에 다다른다.

**return** 0; //Return 0

이 지점에서 0 값이 리턴 되기 때문에

**if**(myfunction()) {

위 문장은 FALSE가 되고, 다음 코드는 무시되며 실행되지 않는다.

**if**(get\_val(XB1\_RT)) { //RT를 누르고 있으면

            combo\_run(Rapid\_Fire);  //Rapid\_Fire 콤보 실행

        }

XB1\_VIEW와 XB1\_A가 동시에 눌러지면, ‘myfunction()’이 실행되고, 다음 코드에 도달하고 실행된다.

RF\_HOLD = adjust\_speed(RF\_HOLD, 10, 1000, 10);

위 라인의 의미는 RF\_HOLD 변수의 값을 ‘adjust\_speed’ 함수가 리턴하는 값으로 할당하라 또는 저장하라 는 것이다. 이제 adjust\_speed 함수가 값을 어떻게 리턴하는지 살펴보자.

위 코드와 같이, ‘adjust\_speed’ 함수는 4개의 입력 파라메터를 가지며, 그것이 RF\_HOLD, 10, 1000, 10이다. 이제 ‘adjust\_speed’ 함수가 어떻게 선언되었는지 살펴보자.

**function** adjust\_speed(var, min\_value, max\_value, adjustment\_increment) {

function ‘adjust\_speed’ 는 4개의 인수가 요구되며, 조정될 변수 var, 최소값 min\_value, 최대값 max\_value, 얼마큼씩 조정될지 증분 adjustment\_increment이다.

변수의 값을 조정하기 위해, 함수는 다음 코드를 실행한다.

**if**(event\_press(XB1\_RB) && var < max\_value)

        var = var + adjustment\_increment;

**if**(event\_press(XB1\_LB) && var > min\_value)

        var = var - adjustment\_increment;

**return** var;

위 코드의 첫 부분에, XB1\_RB를 누르고, var 값이 max\_value보다 낮으면 var 에 10이 더해지게 된다. 그런 후, function이 호출된 곳으로 var의 값이 리턴된다. 따라서, RF\_HOLD는 전보다 10이 더 커지게 된다.

만약 XB1\_LB를 누르고, var 값이 min\_value보다 크면 var 에 10이 빼지게 된다. 그런 후, function이 호출된 곳으로 var의 값이 리턴된다. 따라서, RF\_HOLD는 전보다 10이 더 작아지게 된다.

XB1\_VIEW와 XB1\_B를 누르고 있으면 RF\_NULL의 변수가 RF\_HOLD에서 그랬던 것과 동일하게 적용된다.

**I/O Functions**

I/O는 입력/출력(Input/Output)을 말한다. 가상머신으로 입력을 주는 두 공급처는 연결된 컨트롤러와 콘솔이다. 가상머신은 컨트롤러와 콘솔로부터 입력을 받을 뿐 아니라, 컨트롤러와 콘솔에 출력을 주기도 한다. 이 섹션에서는 가상머신의 외부로부터 데이터를 받거나, 데이터를 주는 것에 관련된 GPC functions을 찾을 수 있다.

**Controller I/O Functions**

컨트롤러는 주 입력 장치이지만, CronusMAX PLUS는 rumble motors나 LEDs를 제어하기 위해 컨트롤러로 데이터를 보내기도 한다. 아래는 이것과 관련된 GPC 명령어이다.



|  |  |
| --- | --- |
| Function Name | Description |
| get\_val | 컨트롤러 버튼의 현재 값을 반환 |
| get\_lval | 컨트롤러 버튼의 이전 값을 반환 |
| get\_ptime | 컨트롤러 버튼의 상태 변화 경과 시간을 반환 |
| get\_controller | 입력포트에 연결된 컨트롤러의 종류(type)를 반환 |
| get\_battery | 무선 컨트롤러 배터리의 현재 상태를 반환 |
| set\_led | 입력포트에 연결된 컨트롤러의 LED 상태를 설정 |
| set\_rumble | %단위로 속도를 조절하는 럼블모터를 설정 |
| event\_press | 컨트롤러 버튼을 누를 때(press) TRUE를 반환 |
| event\_release | 컨트롤러 버튼을 땔 때(release) TRUE를 반환 |
| block | 지정한 시간 동안 컨트롤러 버튼의 출력을 막는다. |
| swap | 두 컨트롤러 버튼의 출력을 바꾼다. |
| sensitivity | 컨트롤러의 아날로그 스틱의 민감도를 조정 |
| deadzone | 두 컨트롤러 버튼의 데드존을 조정 |
| stickize | 마우스의 값 또는 Wiimote IR 입력을 아날로그 스틱으로 변환 |
| ps4\_touchpad | 듀얼쇼크4 터치패드 상태의 세부 정보를 반환 |
| ps4\_set\_touchpad | 듀얼쇼크 터치패드를 특정 (X,Y) 위치로 지정 |
| turn\_off | 입력포트에 연결된 무선 컨트롤러를 끈다. |
| wiir\_offscreen | Wiimote IR 센서가 off screent이면 TRUE를 반환 |

**1. get\_val**

get\_val은 **int** 형으로 컨트롤러 버튼의 현재 값을 반환한다. 반환되는 정수는 퍼센트%를 의미한다.

GPC에서는 **int**가 논리 값(Boolean value)으로 처리되므로, get\_val 명령어는 컨트롤러 버튼이 0의 값을 가지는지 아니면 0이 아닌 값을 가지는지 확인하는데 사용될 수 있다. 아래 예제를 보자.

**if**(get\_val(PS4\_R1))

위 코드는 R1/RB 버튼이 **0**(zero)이 아닌 값을 가지면, TRUE를 반환한다.

특정 컨트롤러 버튼이 특정 값 이상을 가지는지도 체크하는 사용할 수 있다. 다음 예제를 보자.

**if**(get\_val(PS4\_R2) > 50)

R2/RT 버튼이 50%보다 더 큰 값을 가지면 TRUE를 반환한다.

get\_val 이 **int**를 반환하기 때문에, 다음과 같이 variable에 리턴 값을 할당 할 수 있다.

**int** myvar;

**main** {

    myvar = get\_val(XB1\_LT);

}

**문법**

get\_val ( <identifier> );

**Parameters**

<identifier> : 컨트롤러 버튼의 identifier

**Returns**

지정된 identifier의 현재 값. 버튼 형태에 따라서 -100에서 +100의 값 범위를 가질 수 있다.

**2. get\_lval**

get\_lval은 지정된 identifier의 main 루프의 이전 반복에서의 값을 반환한다는 점을 제외하고 get\_val과 같다. 반환되는 값은 역시 **int** 형이며 퍼센트%를 나타낸다.

그러나, get\_lval은 get\_val과는 달리 입력 보고(input report)에서 데이터를 가져오기 때문에, get\_lval 이전의 코드에 영향을 받지 않는다. 예를 들어 set\_val을 통해 특정 identifier의 출력을 바꾸어도, 지정한 identifier의 콘트롤러에서 들어오는 이전 main 루프의 input 값을 반환한다. 아래 예제를 돌려보면 이해가 될 것이다.

**main** {

   // RT/R2 를 0으로 지정

    set\_val(XB1\_RT, 0);

// 이전 코드에서 RT / R2 가 0(zero)으로 지정됐지만

// RT/R2 트리거를 누른 값을 반환한다.

    set\_val(TRACE\_1, get\_lval(XB1\_RT));

   // 이전 코드에서 RT / R2 가 0 (zero)으로 지정됐기 때문에

// 항상 0을 반환한다.

    set\_val(TRACE\_2, get\_val(XB1\_RT));

}

**문법**

get\_lval ( <identifier> );

**Parameters**

<identifier> : 컨트롤러 버튼의 identifier

**Returns**

지정된 identifier의 이전 값. 버튼 형태에 따라서 -100에서 +100의 값 범위를 가질 수 있다.

**3. get\_ptime**

get\_ptime은 identifier 상태 변화를 밀리초에 해당하는 int 형으로 반환한다. 무슨 뜻인가 하면 identifiers의 값이 FALSE에서 TRUE 또는 TRUE에서 FALSE로 변할 때, 그 identifier에 해당하는 get\_ptime에 해당하는 카운터가 0으로 재설정된다.

따라서, get\_ptime을 위해 시계가 항상 돌아가고 있는 중이고 그래서 get\_val과 같은 명령어와 함께 사용해야 한다.

예를 들어, 아래와 같이 **if** 문 안에 get\_ptime을 사용하면,

**if**(get\_ptime(XB360\_A) < 200) // 원문에는 > 로 되어 있지만 문맥상 // < 게 맞는 것으로 보인다.

A버튼을 눌렀을 때 또는 **200** 밀리초보다 크지 않을 때만 TRUE가 된다.

아래와 같이 get\_ptime과 get\_val을 함께 사용하면 버튼을 누르고 있은 지 200 밀리초 이상일 때만 TRUE가 된다.

**if**(get\_val(XB360\_A) && get\_ptime(XB360\_A) > 200)

**문법**

get\_ptime ( <identifier> );

**Parameters**

<identifier> : 컨트롤러 버튼의 identifier

**Returns**

컨트롤러 버튼의 상태 변화의 경과 시간. 반환되는 값은 0 ~ 4000 밀리초 범위이다.

**4. get\_controller**

get\_controller는 CronusMAX PLUS 입력포트에 연결된 현재 컨트롤러 종류(type)를 표현하는 int 형 값을 반환한다.

컨트롤러가 연결되어 있지 않으면 **0**(zero)을 반환하고, 현재 연결된 컨트롤러 종류에 따라서 **1** ~ **5** 값을 반환한다.

기억하기 쉽게 하기 위해 5종류의 컨트롤러에 대해 할당되어 있는 값에 대한 다섯 개의 기호 상수를 정의해 두었다.

|  |  |  |
| --- | --- | --- |
| Name | Description | Value |
| PIO\_PS3 | Playstation 3 Controller | 1 |
| PIO\_XB360 | Xbox 360 Controller | 2 |
| PIO\_WII | Wii Controller | 3 |
| PIO\_PS4 | Playstation 4 Controller | 4 |
| PIO\_XB1 | Xbox One Controller | 5 |

사용 예

**main** {

**if**(get\_controller() == PIO\_XB1) {

        // 여기 실행

    }

}

**문법**

get\_controller ( );

**Parameters**

None

**Returns**

어떤 종류의 컨트롤러가 연결되어 있는지 나타내는 값

**5. get\_battery**

get\_battery는 배터리 수준을 반환하는데, **int** 형으로 **0** ~ **10** 의 값을 가진다. **0**은 충전이 되어 있지 않는 것이고 **10**은 완충된 것이다.

배터리가 연결되어 있지 않고, 유선 컨트롤러가 연결되어 있으면 **10**을 반환한다.

사용 예

**main** {

**if**(get\_battery() <= 2) {

        // Do Something

    }

}

**문법**

get\_battery ( );

**Parameters**

None

**Returns**

**0** (충전 안됨) ~ **10** (완충) 범위의 값

**6. set\_led**

set\_led 는 컨트롤러의 LED 상태를 설정한다.

LED는 0 ~ 3 사이의 범위이다. 4개의 기호상수가 정의되어 있다.

|  |  |  |
| --- | --- | --- |
| Name | Description | Value |
| LED\_1 | LED 1 / Xbox 360 Quadrant 1 | 0 |
| LED\_2 | LED 2 / Xbox 360 Quadrant 2 | 1 |
| LED\_3 | LED 3 / Xbox 360 Quadrant 3 | 2 |
| LED\_4 | LED 4 / Xbox 360 Quadrant 4 | 3 |

LED는 4개의 상태 중 하나로 설정할 수 있다.

|  |  |
| --- | --- |
| Value | Description |
| 0 | LED Off |
| 1 | LED On |
| 2 | LED Blink Fast |
| 3 | LED Blink Slowly |

사용 예

set\_led(LED\_1**,** 3);  // LED 1을 천천히 깜빡이게 한다.

**문법**

set\_led ( <led\_identifier> , <state> );

**Parameters**

<led\_identifier> : LED의 identifier

<state> : 위 표에 있는 상태를 표시하는 값

더 자세한 사항은 **LEDs** 섹션을 참고하라.

**7. set\_rumble**

set\_rumble는 컨트롤러에 럼블모터들 중 선택된 하나의 속도를 설정한다.

럼블모터들은 0 ~ 3 의 번호를 갖는다. 각 럼블모터의 번호를 쉽게 기억하기 위해 4개의 기호상수가 정의되어 있다.

|  |  |  |
| --- | --- | --- |
| Name | Description | Value |
| RUMBLE\_A | 강한 럼블모터 (보통 왼쪽 모터) | 0 |
| RUMBLE\_B | 약한 럼블모터 (보통 오른쪽 모터) | 1 |
| RUMBLE\_RT | 오른쪽 트리거 모터 (Xbox One 컨트롤러만 있다) | 2 |
| RUMBLE\_LT | 왼쪽 트리거 모터 (Xbox One 컨트롤러만 있다) | 3 |

사용 예

set\_rumble(RUMBLE\_A**,** 50);

**문법**

set\_rumble ( <rumble\_identifier> , <speed as %> );

**Parameters**

<rumble\_identifier> : 럼블모터의 identifier

<speed as %> : 0 ~ 100 범위의 값

더 자세한 사항은 **Rumble** 섹션을 참고하라.

**8. event\_press**

event\_press는 버튼의 상태가 FALSE에서 TRUE로 변할 때 TRUE를 반환한다. 그러므로 버튼을 누른 후 누른 채로 유지하고 있어도, event\_press는 누른 순간에만 TRUE를 반환하고 누르고 있을 때에는 TRUE를 반환하지 않는다. 이것은 버튼을 누를 때만 콤보가 실행되기를 원할 때 사용하면 좋다.

예를 들어, 1인칭 슈팅게임을 하고 있는데, 스타이퍼 라이플을 사용한다고 하자. 그리고 조준버튼을 누를 때(보통 LT/L2) 자동으로 줌(scope)이 땡 겨 지길 원한다 치자. 그렇게 하려면 다음과 같이 하면 된다.

**main** {

**if**(event\_press(XB1\_LT)){ // LT / L2 를 누르는 순간

        combo\_run(scope\_in);    // 콤보 scope\_in 을 실행

    }

}

**combo** scope\_in {

**wait**(400);  // 조준 애니메이션이 끝나길 기다리자.

    set\_val(XB1\_RS, 100); // Zoom scope를 누른다.

**wait**(200);  // Zoom scope를 누른 채로 200ms 대기

}

**문법**

event\_press ( <identifier> );

**Parameters**

<identifier> : 컨트롤러 버튼의 identifier

**9. event\_release**

event\_release는 event\_press의 반대이다. 버튼의 상태가 TRUE에서 FALSE로 변할 때, TRUE를 반환한다. 이것은 버튼을 땔 때만 콤보가 실행되기를 원할 때 사용하면 좋다.

예를 들어, 슈팅 게임을 하는데 사격을 멈추면 언제든지 자동으로 재장전 되기를 원한다 치자. 그러면 다음과 같이 하면 된다.

**main** {

**if**(event\_release(XB1\_LT)){ // LT / L2 를 때는 순간

        combo\_run(reload);    // 콤보 reload 을 실행

    }

}

**combo** reload {

**wait**(200);  // 총 애니메이션이 끝나길 기다리자.

    set\_val(XB1\_X, 100); // 재장전을 위한 X버튼

**wait**(200);  // X를 누른 채로 200ms 대기

}

**문법**

event\_release ( <identifier> );

**Parameters**

<identifier> : 컨트롤러 버튼의 identifier

**10. block**

block은 지정된 밀리초 시간동안 컨트롤러 버튼의 신호 진행을 방해한다. 이때 지정 시간은 **20**에서 **4000** 밀리초 범위로 지정할 수 있다. block은 하나의 버튼을 두 개의 용도로 사용할 때 매우 유용하다.

예를 들어, 점프 게임(platform game)을 한다고 치고, 버튼을 누르고 있으면 더블 점프가 자동으로 나가게 하려면 다음과 같이 하면 된다.

**main** {

**if**(get\_val(XB1\_A)){  // A를 누르고 있으면

**if**(event\_press(XB1\_A)){  // 처음 A를 눌렀을 때

            combo\_run(single\_jump); // 단일 점프가 실행

        }

// 500 밀리초 동안 A 버튼이 눌러짐을 막음

        block(XB1\_A, 500);

    }

}

**combo** single\_jump {

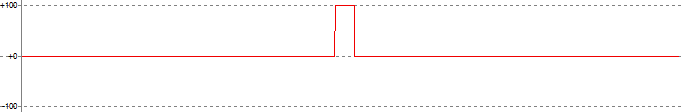
    set\_val(XB1\_A, 100);

**wait**(300);

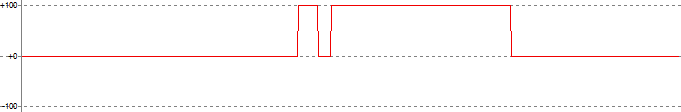
}

위 코드는 A 버튼이 눌러지자 마자 single\_jump 콤보가 실행된다. 그리고 **500** 밀리초 안에 A 버튼을 때면, 콘솔로 더 이상의 버튼 출력이 나가지 않는다. 그러나 A 버튼을 누르고 난 후 **500** 밀리초 이상 계속 누른 채 유지하면, 버튼 출력이 정상적으로 나간다.

버튼을 빠르게 눌렀다 500 밀리초 안에 때면 나타나는 출력



버튼을 누르고 500 밀리초 이상 누르고 있으면 나타나는 출력



**문법**

block ( <identifier> , <milliseconds> );

**Parameters**

<identifier> : 컨트롤러 버튼의 identifier

<milliseconds> : 버튼 출력을 막을 밀리초. **20** ~ **4000** 범위

**11. swap**

swap는 이름이 암시하는 것처럼, 버튼 두 개의 값을 바꾼다. 이것은 실행시간에 버튼을 remap하는데 유용하다.

예를 들어, 어떤 슈팅 게임을 하는데 그 게임에서 조준은 왼쪽 트리거 버튼이고(LT) 조준 시 줌은 오른쪽 아날로그 스틱 클릭(RS)이라고 하자. 이 줌 기능을 조준할 때만 RS에서 RB로 바꾸고 싶다면 다음과 같이 하면 된다.

**main** {

**if**(get\_val(XB1\_LT)) { // LT를 누르고 있으면

        swap(XB1\_RS, XB1\_RB); // RS와 RB를 바꾼다.

    }

}

위 코드는 LT를 누르고 있으면, RB가 오른쪽 아날로그 스틱 클릭이 되고, 오른쪽 아날로그 스틱 클릭이 RB를 누른 것이 된다.

**문법**

swap ( <identifier1> , <identifier2> );

**Parameters**

<identifier1> : 컨트롤러 버튼의 identifier

<identifier2> : 컨트롤러 버튼의 identifier

**12. sensitivity**

Sensitivity는 아날로그 스틱의 민감도를 조정한다.

이 함수는 세 개의 파라메터를 입력해야 하는데, 첫 번째는 조정할 아날로그 스틱의 axis 축, 두 번째는 midpoint이고, 세 번째는 sensitivity multiplier이다.

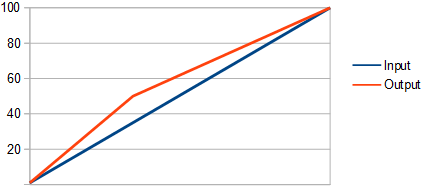
Midpoint는 아날로그 스틱의 중앙지점 값을 지정한다. 디폴트 값은 50%이다. 이 값을 바꿈으로써 두 민감도 범위를 지정할 수 있다. 50%보다 더 낮은 값으로 지정하면 높은 민감도 범위가 정류 위치(rest position)에 가까운 쪽에 만들어 지고, 낮은 민감도 범위가 정류 위치에 먼 쪽에 만들어 진다. 50%보다 높은 값은 반대 효과를 만든다.

Sensitivity multiplier는 input값(input vale)에 곱해지는 양이다. Multiplier 값은 퍼센트에 해당한다. 따라서 **40**은 **0.40**을 곱한 것이고, **100**은 **1.00**을 **140**은 **1.40**을 곱한 것에 해당한다.

아래 그림을 보면 sensitivity 함수를 사용하면 **input**과 **output**에 어떤 차이가 있는지 볼 수 있다.

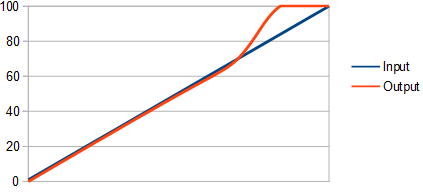
Midpoint를 35%로 하고, sensitivity multiplier를 100으로 하는 명령어는 다음과 같다.

sensitivity(XB1\_LX, 35, 100);



Midpoint를 70%로 하고, sensitivity multiplier를 140으로 하는 명령어는 다음과 같다.

sensitivity(XB1\_LX, 70, 140);



**문법**

sensitivity ( <identifier1> , <midpoint> , <sensitivity> );

**Parameters**

<identifier> : 컨트롤러 버튼의 identifier

<midpoint> : midpoint 값 지정

<sensitivity> : sensitivity 조정 비율

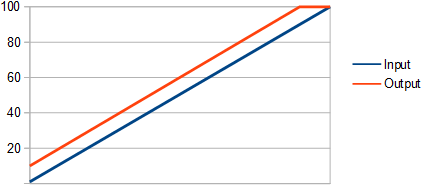
**13. deadzone**

deadzone은 두 축의 deadzone의 값을 조정한다. 콘솔에 프로그램된 디폴트 deadzone은 20%인데, 이것은 20% 이하의 아날로그 스틱 신호는 콘솔이 무시한다는 의미이다. CronusMAX PLUS는 input에 상대적인 output 신호를 조정할 수 있다.

예를 들어, 데드존의 10%를 없애려면, 아래 명령어로 10%를 추가해서 출력해야 한다.

deadzone(XB1\_LX, XB1\_LY , 10, 10);

위 명령어는 수평, 수직 축으로 물리적으로 10%만 움직이면 출력으로는 20%의 효과가 난다. 10% 추가 효과는 90%까지 추가되며 90% 이상은 아래 그림처럼 100%로 취급된다.



위 코드는 사각 deadzone을 만드는 것이고, LX, LY의 두 축에 항상 10%가 더해진다. 그러나, deadzone 함수는 원형 데드존을 사용할 수 있도록 미리 정의된 기호상수 DZ\_CIRCLE(또는 **101**)를 사용할 수 있다.

세 번째 파라메터에 값 대신 DZ\_CIRCLE(또는 **101**)를 사용하면 네 번째 파라메터는 Y axis에 대한 값 대신에 원의 반지름이 지정된다.

DZ\_CIRCLE 기호 상수를 사용한 예는 다음과 같다.

deadzone(XB1\_LX, XB1\_LY , DZ\_CIRCLE , 10);

**문법**

deadzone ( <identifier\_x> , <identifier\_y> ,

<dzone\_x> / DZ\_CIRCLE ,

<dzone\_y> / <radius> );

**Parameters**

< identifier\_x > : X축에 해당하는 컨트롤러 버튼

< identifier\_y > : Y축에 해당하는 컨트롤러 버튼

<dzone\_x> / DZ\_CIRCLE : X축 데드존 값 또는 DZ\_CIRCLE 상수

<dzone\_y> / <radius> : Y축 데드존 값 또는 원의 반지름 값

**14. stickize**

stickize는 Wiimote IR 또는 마우스 입력의 값을 아날로그 스틱으로 변환한다.

<이 부분은 우선 Skip>

**문법**

stickize ( <identifier\_x> , <identifier\_y> , <radius> );

**Parameters**

< identifier\_x > : X축에 해당하는 컨트롤러 버튼

< identifier\_y > : Y축에 해당하는 컨트롤러 버튼

<radius> : 원의 반지름 값

**15. ps4\_touchpad**

ps4\_touchpad는 터치패드의 현재 상태 세부 정보를 반환한다. get\_val처럼 **int** 형을 반환한다. ps4\_touchpad는 터치패드 위에 있는 두 손가락에 대한 정보를 X/Y 좌표 정보로 준다.

다음 표는 미리 정의된 6개의 기호 상수이다.

|  |  |  |
| --- | --- | --- |
| PS4T\_Constant | Value | Description |
| PS4T\_P1 | 0 | 최소 손가락 하나가 터치패드에 있으면 TRUE 아니면 FALSE |
| PS4T\_P1X | 2 | 터치패드 위에 첫 번째 손가락의 X축 값 |
| PS4T\_P1Y | 3 | 터치패드 위에 첫 번째 손가락의 Y축 값 |
| PS4T\_P2 | 16 | 손가락 두개가 터치패드에 있으면 TRUE 아니면 FALSE |
| PS4T\_P2X | 32 | 터치패드 위에 두 번째 손가락의 X축 값 |
| PS4T\_P2Y | 48 | 터치패드 위에 두 번째 손가락의 Y축 값 |

위 상수 중 하나의 값을 얻기 위해, 가단히 ps4\_touchpad 파라메터를 사용하면 된다. 예를 들어, 모든 상수의 상태를 읽기 위해서 다음과 같이 한다.

**main** {

  // 최소 손가락 하나가 터치패드 위에 있으면,

**if**(ps4\_touchpad(PS4T\_P1)) {

        set\_val(TRACE\_1, 1); // TRACE\_1에 1지정

// TRACE\_2에 X 좌표 지정

        set\_val(TRACE\_2, ps4\_touchpad(PS4T\_P1X));

// TRACE\_2에 X 좌표 지정

        set\_val(TRACE\_3, ps4\_touchpad(PS4T\_P1Y));

    } **else** { // 터치패드 위에 손가락 없으면,

        set\_val(TRACE\_1, 0); // TRACE\_1에 0지정

    }

// 손가락 두 개가 터치패드 위에 있으면,

**if**(ps4\_touchpad(PS4T\_P2)) {

        set\_val(TRACE\_4, 1); // TRACE\_4에 1지정

       // TRACE\_5에 두 번째 손가락 X 좌표 지정

set\_val(TRACE\_5, ps4\_touchpad(PS4T\_P2X));

// TRACE\_6에 두 번째 손가락 Y 좌표 지정

        set\_val(TRACE\_6, ps4\_touchpad(PS4T\_P2Y));

    } **else** { // 터치패드에 손가락 두 개 보다 적으면

        set\_val(TRACE\_4, 0); // TRACE\_4에 0지정

    }

}

**문법**

ps4\_touchpad ( <PS4T\_constant> );

**Parameters**

<PS4T\_constant> : 위 표의 상수

**Returns**

PS4T\_constant에 관련된 **int** 값

**16. ps4\_set\_touchpad**

ps4\_set\_touchpad는 듀얼쇼크4 터치패드를 특정 (X,Y) 위치를 터치한 효과를 나게 한다.

**문법**

ps4\_set\_touchpad ( <X Value> , <Y Value> );

**Parameters**

<X Value> : -100에서 +100 사이의 터치패드 X 위치 값

<Y Value> : -100에서 +100 사이의 터치패드 Y 위치 값

**Returns**

없음

**combo** Touch\_And\_Click\_DS4Touchpad {

**ps4\_set\_touchpad**(90, 45);

    set\_val(PS4\_TOUCH, 100);

**wait**(80);

}

**17. turn\_off**

turn\_off는 CronusMAX PLUS 입력 포트에 연결된 무선 컨트롤러의 스위치를 끈다.

사용 예:

**main** {

**if**(event\_press(XB1\_RS)) { // RS를 누르면

        turn\_off(); // 무선 컨트롤러 끔

    }

}

**문법**

trun\_off ( );

**Parameters**

없음

**18. wiir\_offscreen**

wii\_offscreen은 Wiimote 컨트롤러가 off screen을 가리키는 중인지 체크한다.

사용 예:

**main** {

**if**(wii\_offscreen()) {

        // 여기 실행

    }

}

**문법**

wii\_offscreen ( );

**Parameters**

없음

**Returns**

Wiimote IR이 off screen이면 TRUE이고 아니면 FALSE

**Console I/O Functions**

CronusMAX PLUS는 데이터를 주로 콘솔로 출력한다. 그러나, 럼블이나 led 상태 같은 데이터를 받기도 하며, GPC scripts에서 이것을 읽을 수 있다. 아래는 이러한 데이터에 관련된 GPC 명령어다.



|  |  |
| --- | --- |
| Function Name | Description |
| set\_val | 컨트롤러 버튼의 현재 값을 지정(덮어 씌운다, overwrite) |
| get\_console | 출력포트에 연결된 현재 콘솔의 종류를 반환 |
| output\_reconnection | 출력포트에 USB 커넥션을 끈고 재연결 |
| ps4\_authtimeout | PS4 authentication timeout(ps4 재인증) 상태를 반환 |
| get\_led | 지정된 LED의 현재 상태를 반환 |
| get\_rumble | 럼블모터의 현재 상태를 반환 |

**1. set\_val**

set\_val은 두 번째 파라메터로 지정된 값으로 첫 번째 파라메터로 지정된 컨트롤러 버튼의 현재 값을 덮어 씌운다. 이것은 지정된 버튼/축에 대한 컨트롤러의 입력이 무엇이든 set\_val 명령어는 지정한 값으로 덮어 씌운다는 의미이다.

set\_val은 주로 **combo**에서 버튼 값을 순차적으로 지정할 때 사용되지만, **main**이나 사용자 지정 함수에서도 사용될 수 있다.

예를 들어, 슈팅게임을 하고 있고, 트리거 버튼을 살짝만 눌러도 완전히 누른 것으로 하고 싶다고 하자. 그러면 아래와 같이 **main** 섹션에서 set\_val 명령어를 이용하면 된다.

**main** {

**if**(get\_val(XB1\_LT)) { // LT를 조금이라도 누르고 있으면

        set\_val(XB1\_LT, 100); // LT를 완전히 누른 것으로 설정

    }

**if**(get\_val(XB1\_RT)) { // RT를 조금이라도 누르고 있으면

        set\_val(XB1\_RT, 100); // RT를 완전히 누른 것으로 설정

    }

}

**문법**

Set\_val ( <identifier> , <percentage> );

**Parameters**

<identifier> : 컨트롤러 버튼의 identifier

<percentage> : 버튼의 %

**2. get\_console**

get\_console은 CronusMAX PLUS의 출력포트에 현재 연결된 콘솔의 종류를 나타내는 **int** 값을 반환한다.

아무런 콘솔도 연결되어 있는 않으면 **0**(zero)을 반환하고, **1**, **2**, **4**, **5**는 연결된 콘솔의 종류에 따라서 반환되는 값이다.

기억하기 쉽게 4개의 기호상수를 정의해 두었다.

|  |  |  |
| --- | --- | --- |
| Name | Description | Value |
| PIO\_PS3 | Playstation 3 | 1 |
| PIO\_XB360 | Xbox 360 | 2 |
| PIO\_PS4 | Playstation 4 | 4 |
| PIO\_XB1 | Xbox One | 5 |

사용 예:

**main** {

**if**(get\_console() == PIO\_XB1) {

        // 여기 실행

    }

}

**문법**

get\_console ( );

**Parameters**

None

**Returns**

현재 연결된 콘솔의 종류를 나타내는 값

**3. output\_reconnection**

output\_reconnection은 CronusMAX PLUS를 출력포트에 연결된 콘솔로부터 전기적으로 끈었다가 재연결 한다. 이 함수는 주로 PS4 authentication timeout(8분 재인증을 말하는 것으로 생각됨)을 재설정 하는데 사용된다. 그러나 Firmware 1.20이상이면 USB HUB를 사용하면 timeout이 없으므로 필요 없어졌다. 이 함수는 이전 CronusMAX 사용자를 위해 남겨 둔 것이다.

사용 예:

**main** {

  // MENU버튼을 누른 채로 XBOX 버튼을 누르면

**if**(get\_val(XB1\_MENU) && event\_press(XB1\_XBOX)) {

        output\_reconnection();  // 끈었다가 재연결

       // XBOX 버튼은 game 안에서 XBOX 버튼을 누르지 않은 것으로

// 하기 위해 0으로 설정

set\_val(XB1\_XBOX, 0);

    }

}

**문법**

output\_reconnection ( );

**Parameters**

None

**4. ps4\_authtimeout**

ps4\_authtimeout은 PS4의 authentications timeout 상태를 **int** 형으로 반환한다. 이 함수는 펌웨어 1.20 이상에서 더 이상 필요 없어졌다. 그러나 이전 CronusMAX 사용자들과 USB Hub를 사용하지 않는 사람들을 위해 남겨두었다.

ps4\_authtimeout의 주 기능은 PS4 authentication timeout일 때, 자동으로 끈고 재연결하는 시간이 가까워졌다고 경고를 주는 것이다. 아래 예제를 보자.

**int** authcount;

//1 = 30 secs, 2 = 1 min, 3 = 1 min 30 secs, 그리고 16까지

**int** NOTIFY = 3;

**main**{

    //PS4 Cross Over Gaming Section

**if**(get\_console() == PIO\_PS4  && get\_controller() != PIO\_PS4){

        authcount = ps4\_authtimeout();

        swap(1,27);   // Back/Select와 Touchpad 누르기를 바꿈

// Back/Select 누르면서 RS/R3 를 누른다.

**if**(get\_val(27) && get\_val(5)) {

            set\_val(27, 0);

            set\_val(5, 0);

            set\_val(1, 100);

        }

**if**(get\_val(27)) { // Back/Select 누르면서

            if(event\_press(19)){ A/Cross를 누르면

                output\_reconnection(); // Auth Timeout 재설정

            }

            set\_val(19, 0);

        }

**if**(authcount <= NOTIFY + 1) {

            combo\_run(notify);

        }

    }

    //PS4 Cross Over Gaming Section End

}

**combo** notify{

    set\_rumble(RUMBLE\_A, 100);

    set\_rumble(RUMBLE\_B, 100);

**wait**(150);

    reset\_rumble();

**wait**(250\*authcount);

}

**문법**

Ps4\_authtimeout ( );

**Parameters**

None

**Returns**

PS4 authentication timeout 상태.

**5. get\_led**

get\_led는 선택된 LED의 현재 상태를 나타내는 값을 **int** 형으로 반환한다.

LED 범위는 0 ~ 3이다. 기억하기 쉽게 LED에 대한 4개의 기호상수를 정의해 두었다.

|  |  |  |
| --- | --- | --- |
| Name | Description | Value |
| LED\_1 | LED 1 / Xbox 360 Quadrant 1 | 0 |
| LED\_2 | LED 2 / Xbox 360 Quadrant 2 | 1 |
| LED\_3 | LED 3 / Xbox 360 Quadrant 3 | 2 |
| LED\_4 | LED 4 / Xbox 360 Quadrant 4 | 3 |

이 함수가 반환하는 값은 선택된 LED의 현재 상태를 알려준다. 반환 값은 **0** ~ **3**이다.

|  |  |
| --- | --- |
| Return Value | Description |
| 0 | LED Off |
| 1 | LED On |
| 2 | LED Blinking Fast |
| 3 | LED Blinking Slowly |

사용 예:

**main** {

**if**(get\_led(LED\_2) == 1) { // LED 2가 On이면

        // 여기 실행

    }

}

**문법**

get\_led ( <Led\_Identifier> );

**Parameters**

<Led\_Identifier> : LED의 identifier

**Returns**

현재 상태를 나타내는 0 ~ 3 의 int 값

더 자세한 사항은 **LED** 부분을 참고하라.

**6. get\_rumble**

get\_rumble은 선택된 럼블모터의 속도를 int 형으로 반환한다. 반환되는 값의 범위는 **0** ~ **100**으로 속도를 %단위로 나타낸 것이다.

럼블모터는 0 ~ 3의 값을 가진다. 기억하기 쉽도록 4개의 기호 상수가 정의되어 있다.

|  |  |  |
| --- | --- | --- |
| Name | Description | Value |
| RUMBLE\_A | 강한 럼블모터 (보통 왼쪽 모터) | 0 |
| RUMBLE\_B | 약한 럼블모터 (보통 오른쪽 모터) | 1 |
| RUMBLE\_RT | 오른쪽 트리거 모터 (Xbox One 컨트롤러만 있다) | 2 |
| RUMBLE\_LT | 왼쪽 트리거 모터 (Xbox One 컨트롤러만 있다) | 3 |

사용 예:

**main** {

  // Rumble Motor A 가 50% 보다 큰 속도이면,

**if**(get\_rumble(RUMBLE\_A) > 50) {

        // 여기 실행

    }

}

**문법**

get\_rumble ( <rumble\_identifier> );

**Parameters**

<rumble\_identifier> : 럼블모터의 identifier

**Returns**

현재 선택된 모터의 속도를 나타내는 0 ~ 100 의 **int** 값

더 자세한 사항은 **Rumble** 부분을 참고하라.

**LEDs**