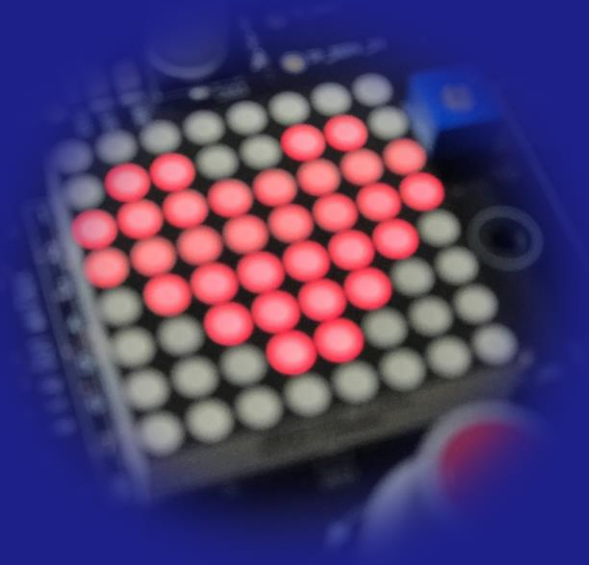
코딩키트

블록 코딩에서 사물인터넷 코딩까지
쉽고 재미있게 배울 수 있어요.

블록 코딩 / 아두이노 코딩 / 라즈베리파이 코딩 / 사물인터넷 코딩

코딩북4 - 라즈베리파이 코딩



SISO DREAM
Silicon to Software

(주)시소드림 SISODREAM Inc.

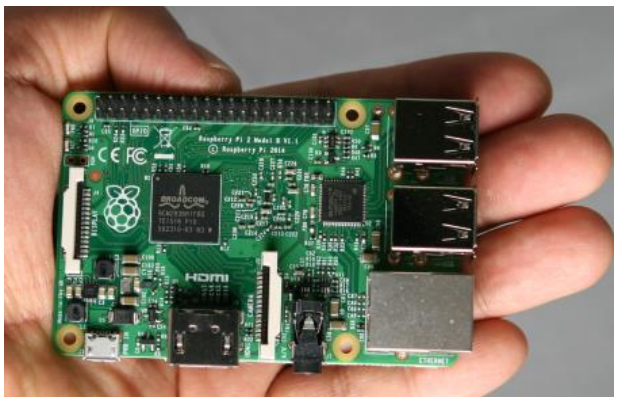
- 본 문서의 저작권 -

발행일 : 2015 년 8 월 28 일 (초판), 2017 년 3 월 8 일 (개정판)
발행처 : ㈜시소드림
연락처 : ck@sisodream.com / 031-757-7755

목 차

[라즈베리파이 소개 및 설치]

라즈베리파이(Raspberry Pi)는 영국의 라즈베리파이 재단이 학교의 기초 컴퓨터 교육을 위해 만든 소형 컴퓨터입니다. 손바닥 안에 쏙 들어올 정도로 매우 작은 컴퓨터입니다.

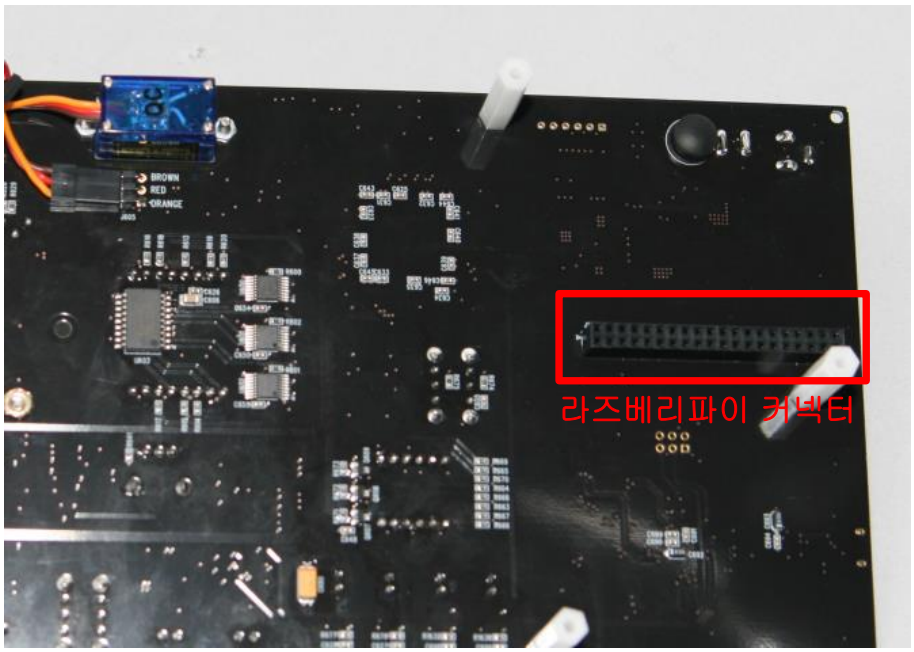


라즈베리파이를 아두이노와 비슷하게 생각하시는 분들이 계시는데, 하드웨어적으로는 두 시스템이 매우 큰 차이를 보입니다. 하지만 두 시스템을 이용하여 코딩을 배우는 입장에서는 비슷한 부분이 많이 있습니다. 그래서 코딩 키트에서는 아두이노와 라즈베리파이를 같이 다루고 있습니다. 또한 이 두 시스템이 **사물 인터넷(IoT : Internet Of Things)**을 구현하기에 매우 적합합니다.

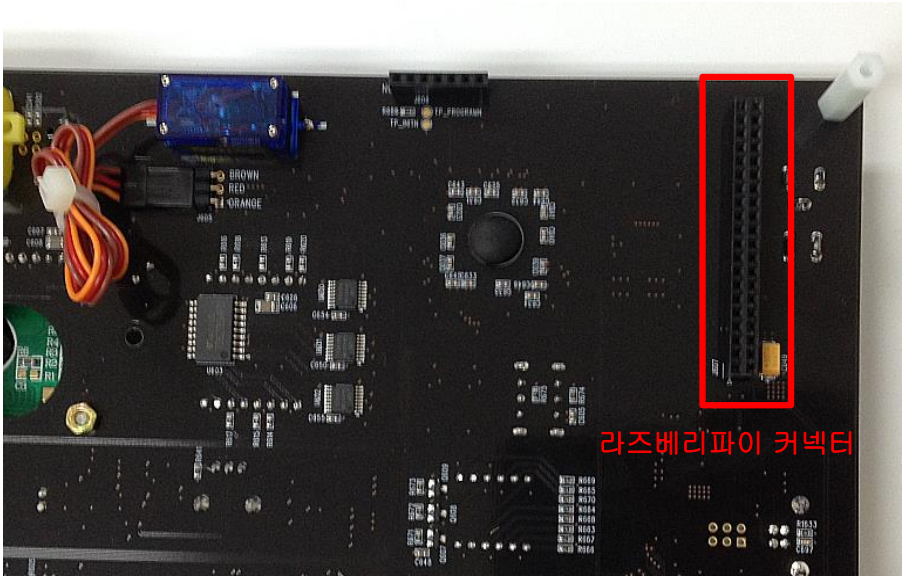
기술적으로만 본다면 라즈베리파이는 아두이노에 비해서 성능이 월등히 뛰어나고 운영체제(Operating System) 등을 사용함으로서 아두이노보다는 PC 와 더 가까운 형태입니다. 그래서 라즈베리파이는 키보드, 마우스, 모니터 같은 입출력 장치를 연결하고, 코딩된 코드를 컴파일하여 직접 실행합니다. 이것은 PC 에서 자신이 사용할 프로그램을 만들고 직접 컴파일하고 실행하여 사용하는 것과 같습니다. 예를 들어, PC 에서 동작하는 게임 프로그램은 PC 에서 코딩하고, 컴파일하여 만듭니다. 마찬가지로 라즈베리파이에서 동작하는 게임 프로그램은 라즈베리파이에서 코딩하고 컴파일하여 만듭니다. 그런데, 아두이노는 어떻게 했습니까? PC 에서 코딩하고 컴파일하여 PC 에서 실행한 것이 아니고 아두이노로 업로드하여 아두이노에서 실행하였습니다. 이것이 아두이노와 라즈베리파이의 가장 큰 차이입니다. 그래서 여러분이 라즈베리파이를 다눌 때는 기본적으로 키보드, 마우스, 모니터만 있으면 됩니다. PC 가 따로 필요하지 않습니다.

그리고 아두이노 프로그램처럼 PC 에서 컴파일하지만 실행을 PC 에서 하지 않는 PC 컴파일 프로그램을 크로스 컴파일러(Cross Compiler) 라고 합니다. 여러분이 앞으로 하드웨어를 기반으로 한 프로그래머가 된다면 많이 듣게될 용어이니 잘 알아 두십시요. 간단한 예로 스마트폰의 프로그램도 다 PC 에서 합니다. 그것은 PC 에 스마트폰용 크로스 컴파일러가 있다는 이야기입니다.

라즈베리파이를 사용하려면 몇 가지 준비 과정이 필요합니다. 먼저 라즈베리파이가 있어야겠지요. 코딩 키트에는 아두이노는 있지만 라즈베리파이는 장착되어 있지 않습니다. 코딩 키트를 뒤집어 보면 다음 그림과 같이 라즈베리파이를 연결하는 커넥터가 있습니다.

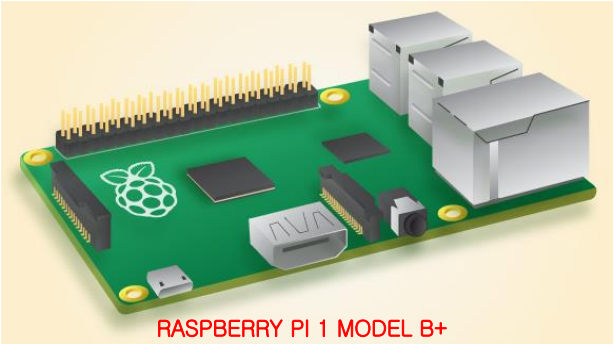


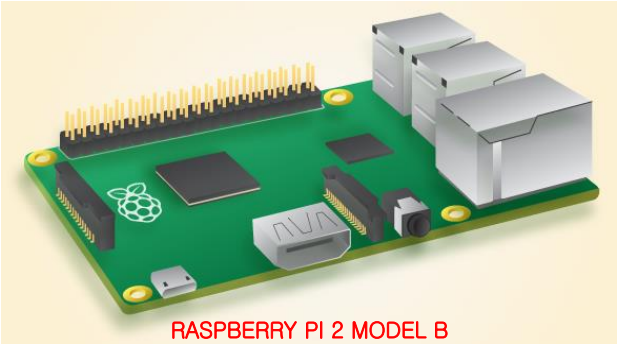
< 코딩 키트 Long Version >



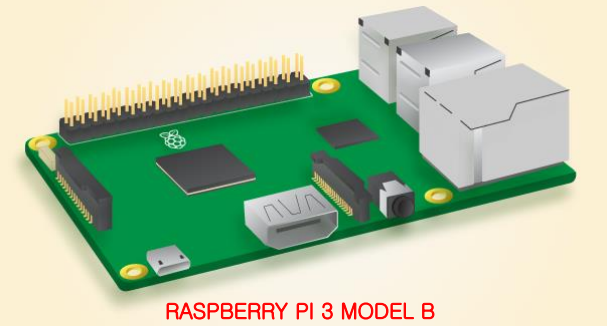
< 코딩 키트 Short Version >

현재 코딩 키트에서는 라즈베리파이 1 Model B+, 라즈베리파이 2 Model B, 라즈베리파이 3 MODEL B 를 지원합니다. 향후 라즈베리파이의 새로운 버전이 출시 된다면 이에 맞추어 코딩 키트도 업그레이드 할 것입니다.


RASPBERRY PI 1 MODEL B+


RASPBERRY PI 2 MODEL B

출처 : https://www.raspberrypi.org/


RASPBERRY PI 3 MODEL B

출처 : https://www.raspberrypi.org/

그래서 라즈베리파이를 구매하실 것이라면 두 모델 중 하나를 구매해 주십시요.

코딩 키트의 라즈베리파이 키트는 다음과 같이 두 가지 품목으로 구성되어 있습니다. 하나는 마이크로 SD 카드로 8GB 의 용량입니다.

이 마이크로 SD 카드는 비어 있는 것이 아니라 여러분이 라즈베리파이를 쉽게 다룰 수 있도록 라즈베리파이용 운영체제 및 여러 유용한 소프트웨어들, 개발 프로그램, 여러가지 설정들이 들어가 있습니다. 또한 코딩 키트를 이용한 여러가지 재미있는 예제 코드들이 있습니다.

여러분이 라즈베리파이를 이용하여 코딩 연습을 하려면 라즈베리파이 웹사이트에서 받은 운영체제에 여러가지 설정을 바꾸어 주어야 편하게 사용할 수 있습니다. 예를 들어 라즈베리파이에 있는 키보드

설정은 우리나라에서 주로 사용하고 있는 키보드 설정과는 다릅니다. 여러분이 샵(#) 키를 누르면 "#" 이 표시가 되지 않고 "￡"으로 표시가 됩니다. "#" 은 여러분이 앞으로 라즈베리파이에서 코딩할 때 코드의 주석 처리를 하는 기호여서 매우 많이 사용합니다. 그런데 "#" 으로 표시가 되지 않고 "￡"으로 표시가 되니 얼마나 불편하겠습니까? 그래서 키보드 설정을 바꾸어야 합니다. 이것처럼 몇 가지 더 설정을 바꾸어야 하는데, 이것이 꽤 귀찮고 복잡한 절차를 거쳐야 합니다. 게다가 중간에 실수라도 한다면 처음부터 다시 해야 합니다. 여러분의 이러한 수고를 덜기 위해서 코딩 키트에서는 모든 설정이 다 되어 있는 마이크로 SD 카드를 공급해 드립니다. 그러면 여러분은 귀찮은 작업을 안 하고 빠르게 라즈베리파이 코딩 세계로 들어갈 수 있습니다.

또한 이 마이크로 SD 카드에는 이 코딩 북에 친절하게 설명되어 있는 예제들의 원본 코드가 들어 있습니다. 그래서 여러분이 공부를 하시다가 원본 코드가 필요할 때 이 마이크로 SD 카드에 있는 예제들을 바로 사용하실 수가 있습니다.

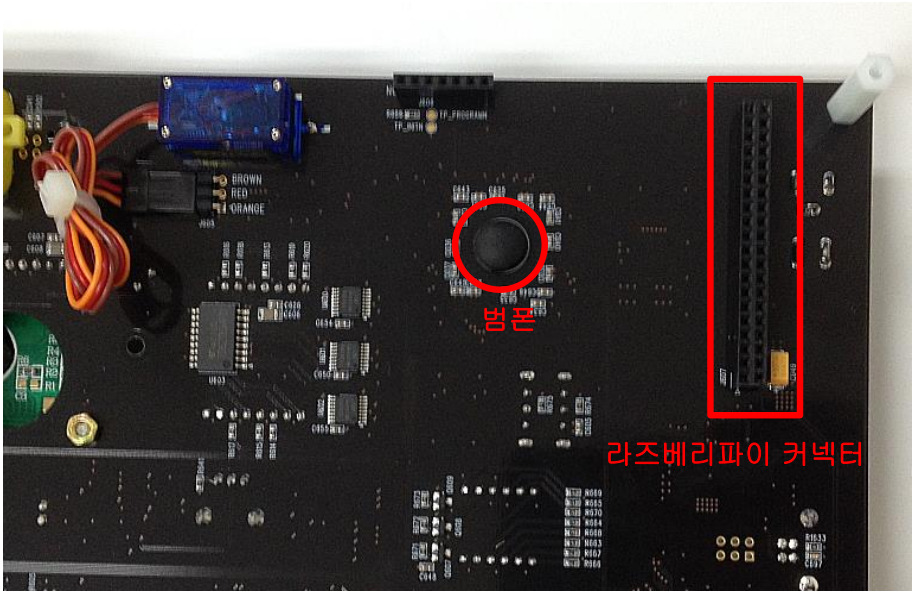
그럼 이 마이크로 SD 카드를 라즈베리파이에 장착해 주십시요. 라즈베리파이를 뒤집어 보면 마이크로 SD 카드 소켓이 보일 것입니다. 그곳에 장착해 주시면 됩니다. 마이크로 SD 카드를 제거하실 때는 마이크로 SD 카드를 소켓쪽으로 꾹 눌러 주시면 빠집니다.

라즈베리파이 키트의 또 하나의 구성품은 크로스(Cross) 랜 케이블입니다.

이 케이블은 라즈베리파이에 직접 모니터, 마우스, 키보드를 연결하지 않고 PC 에서 원격으로 라즈베리파이를 컨트롤할 때 사용합니다.

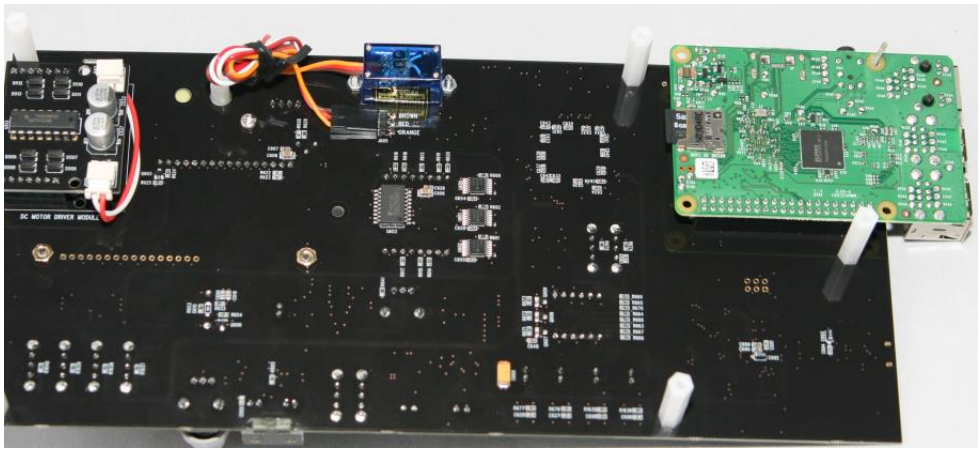
그럼 이제 마이크로 SD 카드가 장착된 라즈베리파이를 코딩 키트에 장착하겠습니다. 그 전에 코딩 키트를 뒤집어 아크릴 케이스를 제거해 주십시요. 그리고 라즈베리파이를 코딩 키트에 장착해 주시기 전에 코딩 키트의 뒷면에 다음 그림과 같이 범폰(Bumpon)이 장착되어 있는지를 꼭 확인해 주십시요.

< 코딩 키트 Long Version >

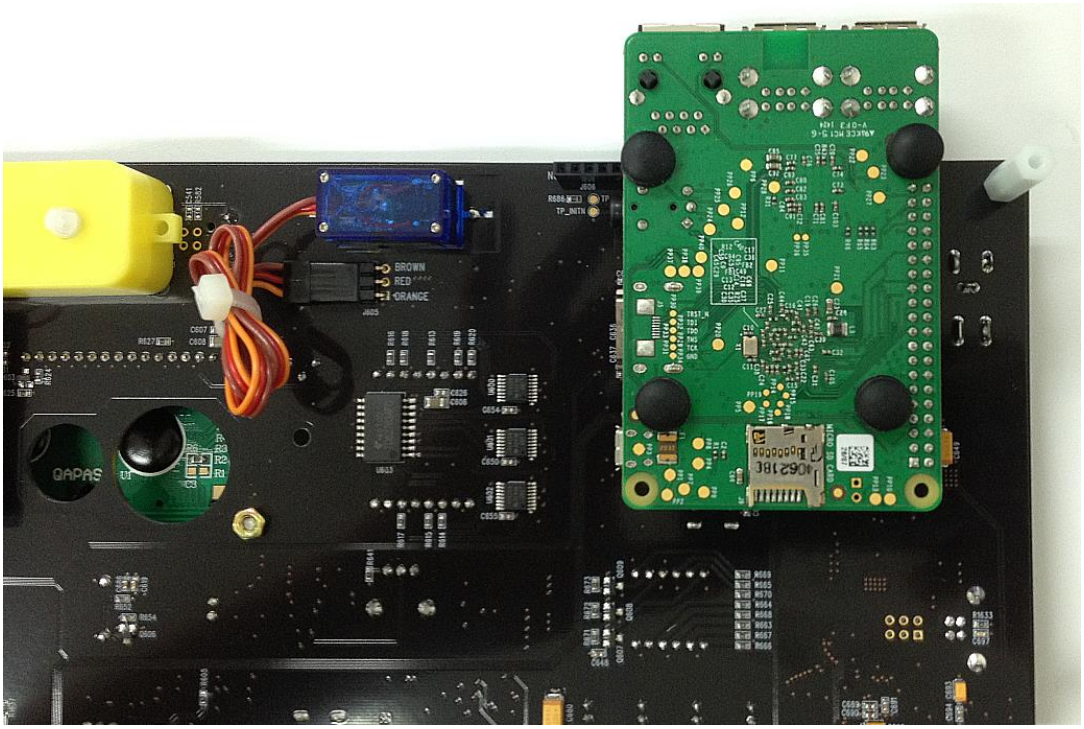


< 코딩 키트 Short Version >

이제 위 그림의 라즈베리파이 커넥터에 라즈베리파이를 연결하십시요. 연결은 아래 사진을 참조하여 방향을 잘 맞추어 연결해 주십시요.



< 코딩 키트 Long Version >



< 코딩 키트 Short Version >

이렇게 하여 라즈베리파이의 장착이 완료되었습니다.

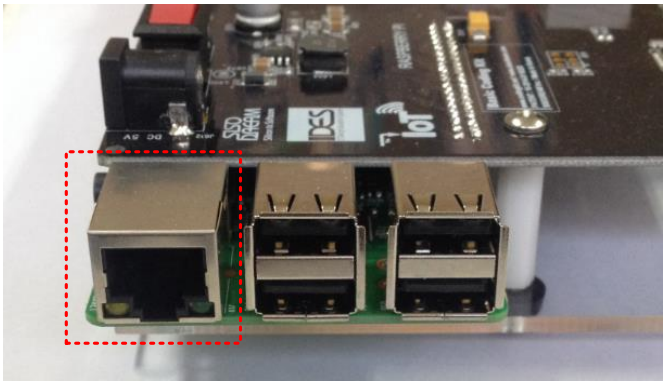
위에서 설명드린 것처럼 라즈베리파이는 아두이노와 다르게 라즈베리파이에 직접 키보드, 마우스, 모니터를 연결하여 사용한다고 설명드렸습니다. 그런데 이렇게 사용하는 방법 이외에 PC 에 연결하여 PC 에서 원격으로 사용하셔도 됩니다. PC 에서 원격으로 사용하게 되면 PC 와 라즈베리파이는 크로스 랜케이블로 연결이 되고, PC 의 키보드, 마우스, 모니터를 라즈베리파이의 입출력 장치로 활용할 수 있습니다. 그래서 라즈베리파이 키트에 크로스 랜케이블이 있는 것입니다.

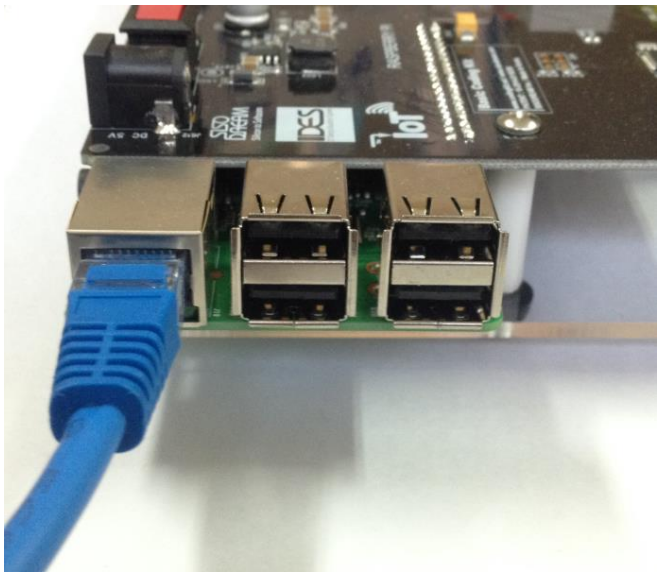
특히 PC 를 노트북으로 사용하시는 경우에는 따로 키보드를 사셔서 라즈베리파이에 연결하지 마시고 노트북에서 원격으로 하는 것이 좋습니다.

이제 여러분은 라즈베리파이에 직접 키보드, 마우스, 모니터를 연결할 것인가 아니면 PC 에서 원격으로 라즈베리파이를 연결하여 사용할 것인가를 결정하여야 합니다. 코딩 북에서는 먼저 PC 에서 원격으로 라즈베리파이를 연결하여 사용하는 방법을 설명하겠습니다. 라즈베리파이에 직접 키보드, 마우스, 모니터를 연결하여 사용할 것이면 이 부분은 건너 뛰셔도 됩니다.

[라즈베리파이 원격 제어]

다음과 같이 라즈베리파이의 유선랜 포트에 제공된 크로스랜 케이블을 연결하십시요.





그리고 이 케이블의 나머지 한쪽은 PC 의 유선랜 포트에 연결하십시요. 그리고 이제 전원 스위치를 켜 주십시요. 이렇게 하여 코딩 키트와 라즈베리파이는 준비가 다 되었습니다.

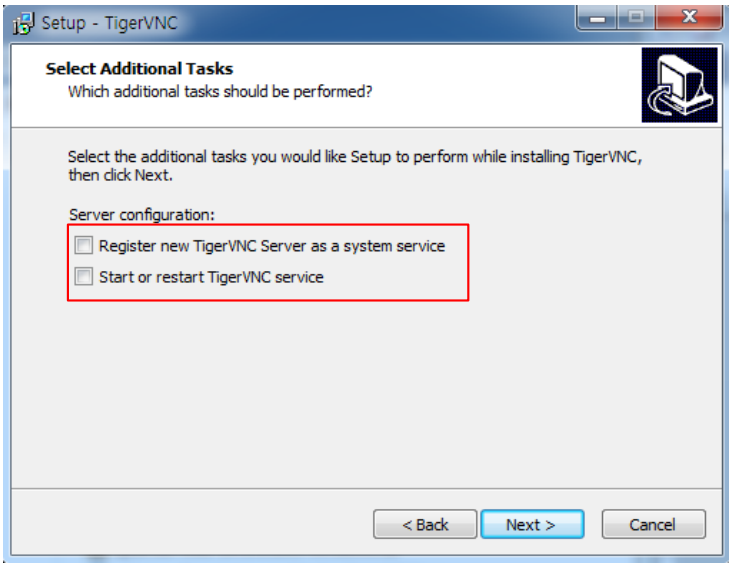
이제 PC 에 라즈베리파이를 원격으로 제어할 프로그램을 설치하여야 합니다. 이런 프로그램으로 주로 VNC (Virtual Network Computing, 가상 네트워크 컴퓨팅) 프로그램을 사용합니다. VNC 란 네트워크 연결을 통하여 한 장치에서 다른 장치를 원격으로 제어하는 데 사용하는 기술을 말합니다. 현재 여러 분야에 사용되고 있으며, 라즈베리파이의 원격제어에도 사용하기 위해 PC 에 이런 종류의 소프트웨어를 설치하는 것 입니다.

저희가 사용할 VNC 프로그램의 이름은 Tiger VNC 라고 합니다. 아래의 링크에서 Tiger VNC 를 다운로드 받습니다.

https://bintray.com/tigervnc/stable/tigervnc/1.3.1

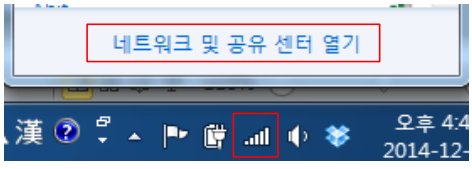
이 글을 작성할 시점에서는 1.3.1 버전이 가장 최신 버전입니다. 여러분이 코딩 키트를 사용할 시점에 다른 최신 버전이 있다면 그 버전을 사용하셔도 괜찮습니다. 다음 링크를 클릭하여 다운로드 받습니다.

설치할 때 아래와 같은 선택 창이 나타나면 두 개 모두 선택해제하고 진행합니다.



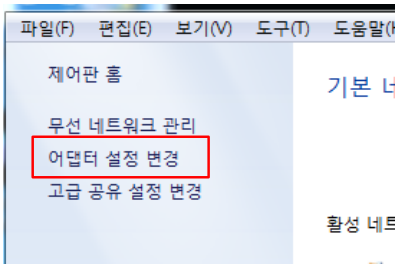
이제 PC 의 IP 주소를 설정하여 합니다. 만약 여러분이 유선으로 인터넷을 사용중인 경우라면 이 설정으로 인해서 인터넷 연결이 해제되므로 더 이상 인터넷을 사용할 수 없게 됩니다. 그래서 유선으로만 인터넷을 사용하시는 분들께는 라즈베리파이의 원격제어를 권장하진 않습니다. 유선과 무선 네트워크어댑터 둘 다 장착되어 있는 PC 의 경우에는 유선연결을 원격제어에 사용하고 무선을 인터넷 연결에 사용할 수 있습니다.

그럼 이제 유선 네트워크의 IP 주소를 설정하겠습니다. 다음과 같이 PC 우측 하단의 무선 네트워크 아이콘을 클릭 하시면 "네트워크 및 공유 센터 열기" 링크가 활성화 됩니다.



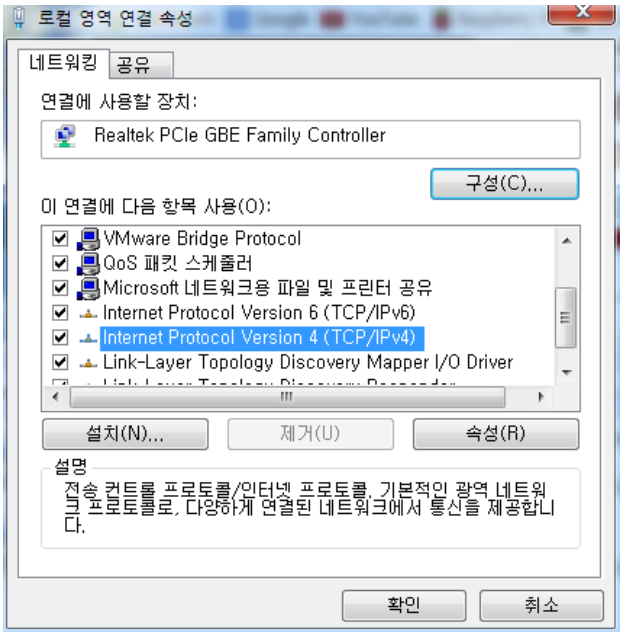
위 그림의 "네트워크 및 공유 센터 열기" 를 클릭하면 해당 창이 나타납니다. 그 창에서 "어댑터 설정 변경"

을 선택합니다.

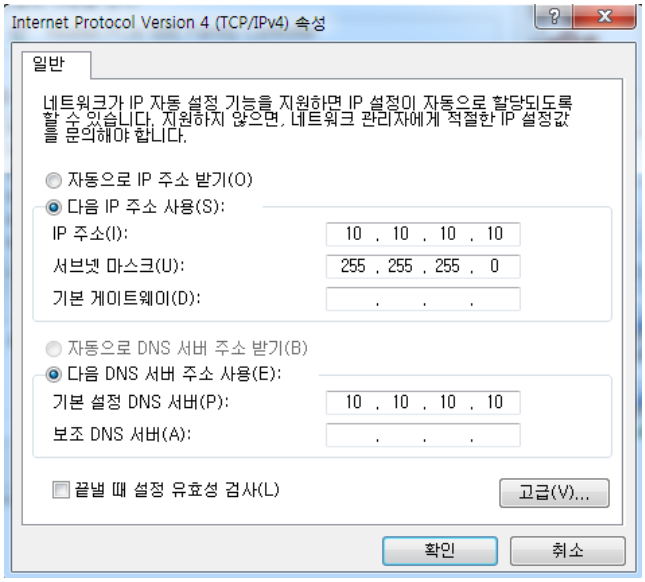


그리고 나타나는 창에서 아래 그림과 같은 "로컬 영역 연결"에 마우스 커서를 위치 시킨 후 마우스 오른쪽 버튼을 클릭한 후 "속성" 메뉴를 선택해 줍니다.

그러면 "로컬 영역 연결 속성" 창이 나타납니다. 이 창에서 "Internet Protocol Version 4(TCP/IPv4)" 를 선택한 후 속성 아이콘을 클릭합니다.



그리고 나타난 창에서 다음 그림과 같이 "다음 IP 주소 사용"을 선택하고 "IP 주소"를 10.10.10.10 으로 설정하여 줍니다. 그리고 서브넷 마스크는 255.255.255.0 으로 설정합니다. "다음 DNS 서버 주소 사용" 을 선택하고, "기본 설정 DNS 서버" 는 10.10.10.10 으로 설정합니다.



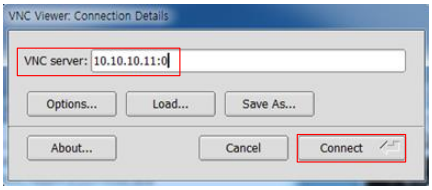
이렇게 설정한 후 확인을 클릭하여 창을 닫습니다. 설정과 관련하여 열여 있던 모든 창을 닫아 주십시요.

만약 네트워크 설정을 원래 상태로 되돌리려면 "자동으로 IP 주소 받기" 와 "자동으로 DNS 서버 주소 받기" 를 선택하시면 됩니다.

여기서 코딩 키트의 전원이 켜 있고, 크로스 랜 케이블이 라즈베리파이와 PC 에 연결되어 있는지 확인해 주십시요. 원격제어를 사용하기 위해서는 라즈베리파이의 부팅시간 1 ~ 2 분 정도가 필요하기 때문에 코딩 키트의 전원을 켜 준 이후에 1 ~ 2 분 정도 기다렸다가 PC 에서 원격제어 프로그램인 VNC 프로그램을 실행시켜야 합니다.

바탕화면의 Tiger VNC Viewer 아이콘을 더블클릭하거나 시작 → 모든 프로그램 → TigerVNC → Tiger VNC Viewer 를 선택하여 VNC viewer를 실행합니다.

VNC Viewer 가 실행되면 다음과 같은 창이 나타납니다.



여기서 10.10.10.11 은 PC가 VNC를 통해 라즈베리파이를 식별할 수 있는 라즈베리파이의 IP 주소이고 :0 은 라즈베리파이의 윈도우를 PC에서 얼마의 해상도로 표시할 것인지를 설정하는 해상도 설정 번호입니다. 이 해상도 설정 번호는 :0 ~:5 까지입니다. 각 번호에 따른 해상도는 라즈베리파이 1 모델과 라즈베리파이 2 모델은 다음과 같습니다.

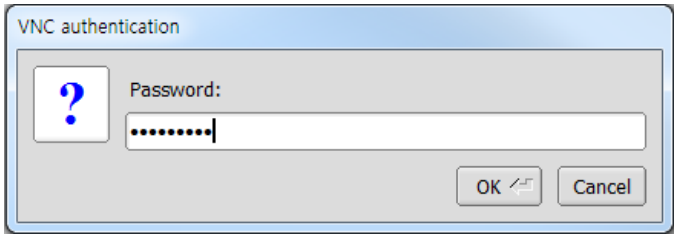
해상도 설정 번호	해상도
0	1920x1080
1	1680x1050
2	1600x900
3	1280x720
4	1024x768
5	800x600

라즈베리파이 3 모델은 다음과 같이 숫자 1 부터 시작합니다.

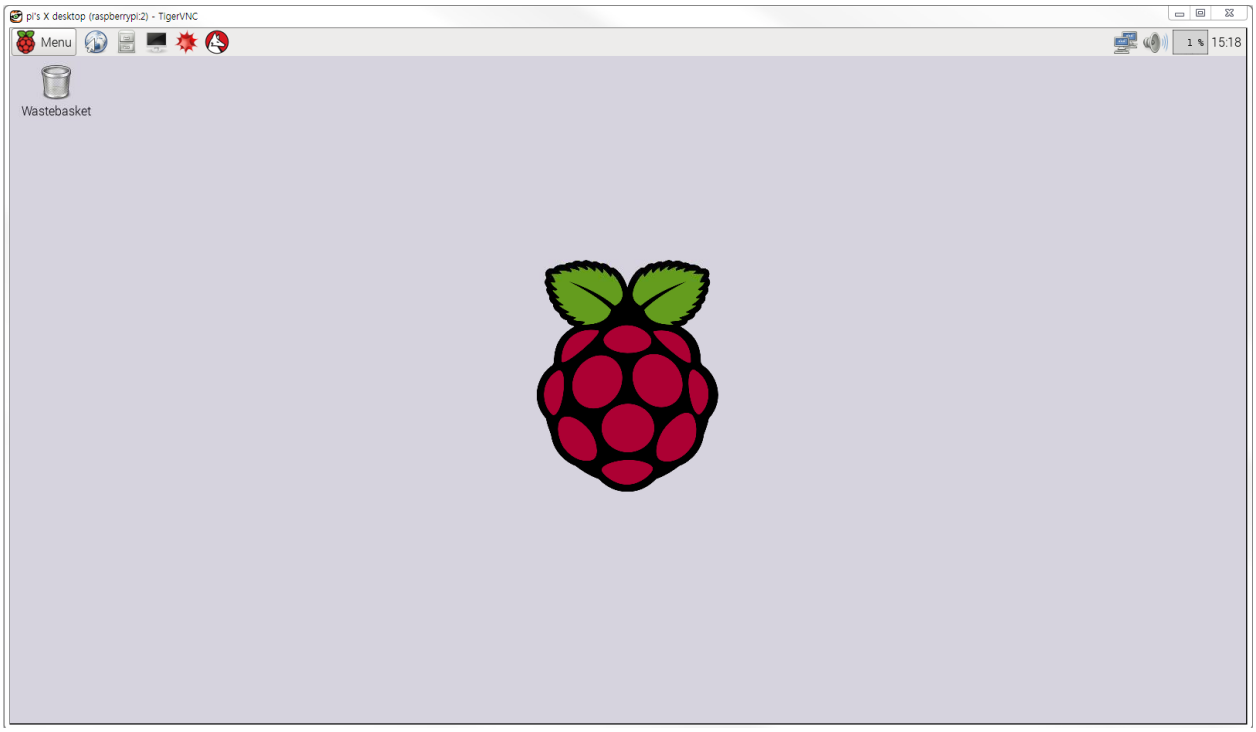
해상도 설정 번호	해상도
1	1920x1080
2	1680x1050
3	1600x900
4	1280x720
5	1024x768
6	800x600

VNC Viewer 의 해상도는 PC의 해상도 보다 약간 낮은 해상도를 선택하여 사용하는 것이 좋습니다. 예를 들어 내 컴퓨터 모니터 화면의 해상도가 1920x1080 이면 10.10.10.11:2 를 선택하여 VNC Viewer 의 해상도는 1600x900 정도로 사용하는 것이 좋습니다.

이제 "Connect" 아이콘을 클릭하면 암호(Password)를 입력하는 창이 나옵니다. 그 창에는 "codingkit" 를 입력하고 "OK" 버튼을 클릭하면 라즈베리파이가 연결됩니다.



연결이 잘 되었다면 다음과 같은 화면을 볼 수 있습니다.

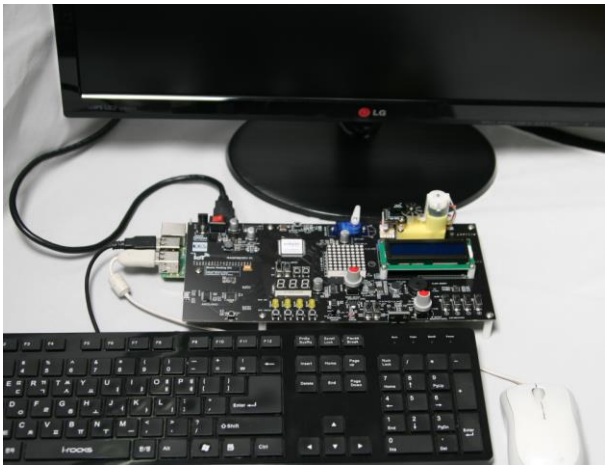


VNC 연결은 라즈베리파이를 켠 이후에 1 ~ 2분 정도 기다리셨다가 연결하십시요. 이유는 라즈베리파이가 완전히 부팅된 이후에 VNC 연결을 할 수 있기 때문입니다.

라즈베리파이를 끝낼 때는 좌측 상단 메뉴(Menu) 아이콘의 Shutdown 을 선택하면 됩니다. 라즈베리파이 3 부터는 원격제어로 라즈베리파이 3 를 끝낼 때는 해당 로그인 아이디의 암호(Password)를 물어 봅니다. 기본적으로 설정되어 있는 암호는 raspberry 입니다.

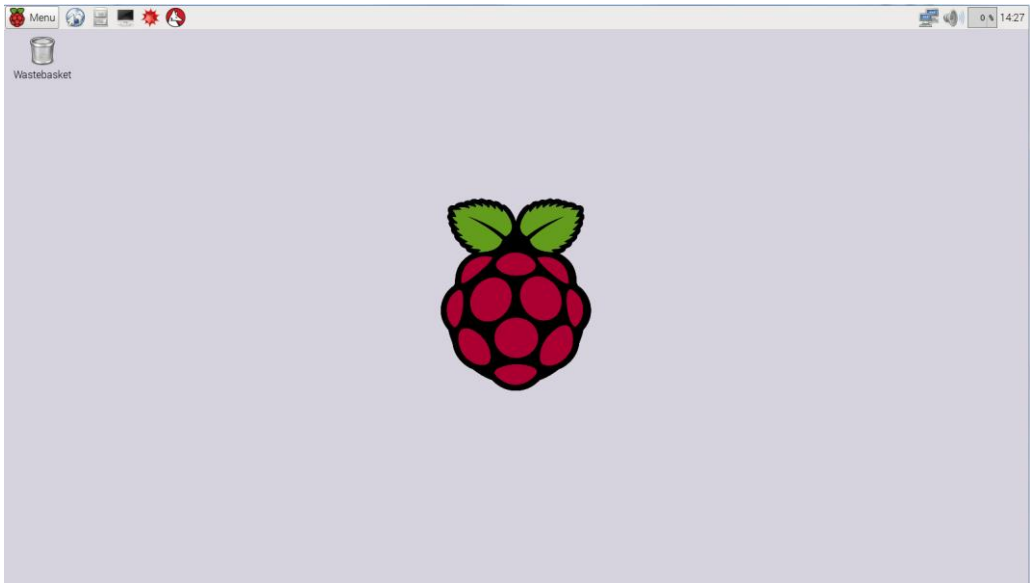
이제 코딩 준비는 완료되었습니다. 이제부터 즐겁게 라즈베리파이 코딩을 해 보겠습니다.

이번에는 라즈베리파이에 키보드, 마우스, 모니터를 직접 연결하는 것에 대해 설명드리겠습니다. 이것은 특별히 어려울 것이 전혀 없습니다. 그냥 다음과 같이 키보드, 마우스는 라즈베리파이의 USB 포트에 연결하고, 모니터는 HDMI 케이블로 연결하면 됩니다.



이제 코딩 키트의 전원 스위치를 올립니다. 그러면 모니터에 수많은 글자들이 올라가는 것이 보이면서 라즈베리파이가 부팅이 됩니다. 더 이상 글자가 올라가는 것이 보이지 않고 프롬프트가 뜨면 다음과 같이 startx 쓰고 엔터 키를 누릅니다.

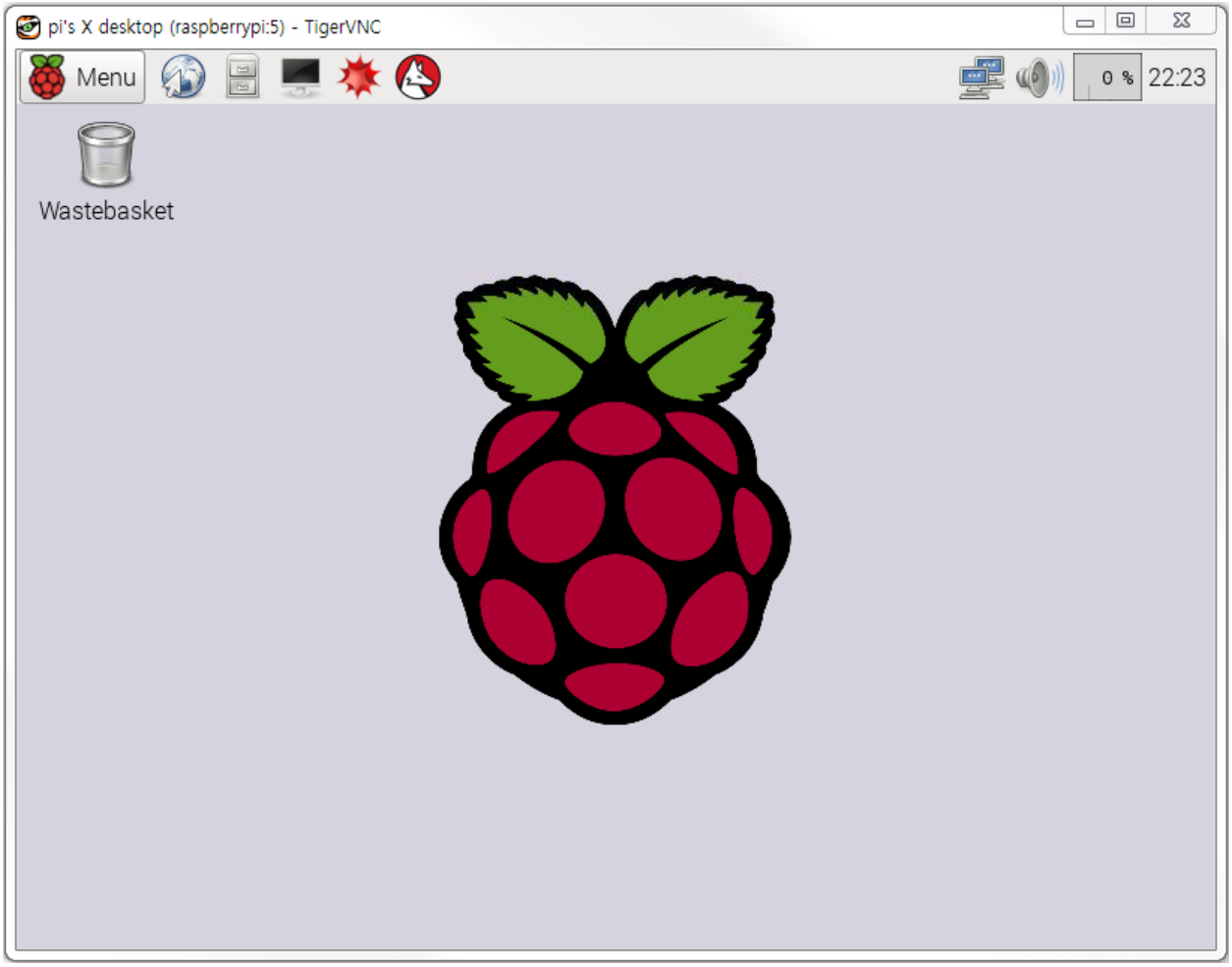
그러면 다음과 같은 화면이 보입니다.



이렇게 하여 키보드, 마우스, 모니터를 직접 연결하는 방법도 알아 보았습니다. 앞으로 코딩 북에서 설명드리는 모든 부분은 PC 에서 원격으로 하든, 키보드, 마우스, 모니터를 직접 연결하든 전혀 관계가 없으므로 이 부분에 대해서는 앞으로는 전혀 신경 쓰실 것이 없습니다.

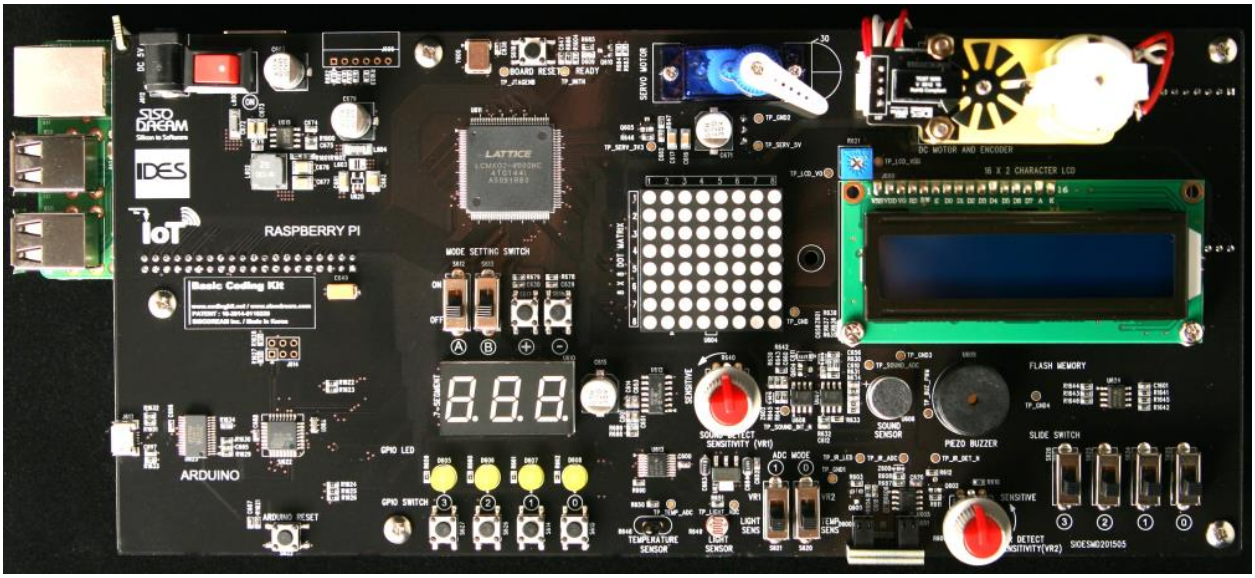
[라즈베리파이 운영체제 소개와 코딩 키트 초기 설정]

라즈베리파이에서 출력된 화면은 라즈비안 이라는 라즈베리파이 운영체제 소프트웨어 화면입니다. 이 라즈비안 운영체제는 리눅스에 그 바탕을 두고 있습니다. 리눅스를 라즈베리파이 하드웨어에 맞게 고친 것입니다. 그래서 리눅스를 사용해 보신 분이라면 쉽게 다룰 수 있을 것입니다. 그렇지 않은 분들도 크게 걱정하실 것은 없습니다. 우리는 라즈베리파이를 활용한 코딩을 공부할 것이지 리눅스를 공부할 것은 아니기 때문에 리눅스를 배워야 하나 하는 걱정은 전혀 하지 않아도 됩니다.



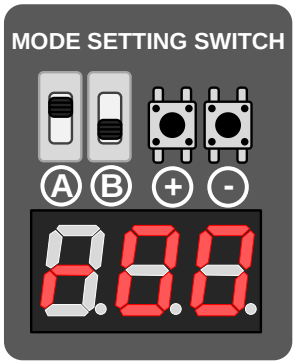
잠깐 라즈비안 운영체제를 살펴 보면, 바탕화면에는 산딸기 모양의 라즈베리파이 공식 로고가 보입니다. 좌측 상단에는 메뉴 버튼이 보입니다. 윈도우 운영체제의 사작 버튼과 비슷한 것입니다. 이 메뉴 버튼을 누르면 여러가지 프로그램을 열거나 라즈베리파이를 끌 수 있는 Shutdown 메뉴가 나옵니다. 그 옆으로는 여러가지 자주 쓰이는 아이콘들이 보입니다. 오른쪽 끝으로는 네트워크와 소리 관련 아이콘이 보이고 그 옆에 % 숫자는 CPU 사용량을 보여주는 것입니다. 제일 끝에는 시간을 표시해 줍니다. 더 자세한 사항은 차차 알아보도록 하겠습니다.

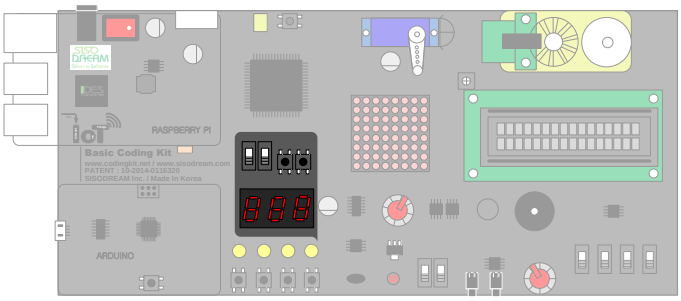
코딩 키트와 라즈베리파이가 장착된 앞면 모습은 다음과 같습니다.



코딩 키트는 아두이노도 사용할 수 있고 라즈베리파이도 사용할 수 있습니다. 라즈베리파이를 사용할 때는 다음과 같이 "Mode Setting Switch" 중 "A" 스위치를 위로 올려야 합니다. 이 스위치를 위로 올리면 세븐세그먼트 3 개 중 제일 왼쪽 세븐세그먼트에 "r" 자가 표시됩니다. 그리고 "+" 혹 "-" 버튼을 2 초 정도 계속 누르고 있으면 세븐세그먼트가 깜박이는 것을 멈추고 다음 그림과 같이 "r00" 을 표시합니다. 이것은 코딩 키트가 "라즈베리파이 00 동작모드" 에서 동작하고 있음을 보여주며, 이 동작모드에 따라 라즈베리파이를 통해 직접 구동할 수 있는 주변 장치들이 변경됩니다. 이 동작모드는 r00 부터 r03 까지 있으며 각 모드에 따른 라즈베리파이의 핀번호와 주변 장치의 연결 구조는 부록 A 에서 참고하실 수 있습니다.

위 사진의 스위치 위치나 세븐세그먼트의 값이 잘 확인이 안 되시면 아래 그림을 참조하십시요.





여기까지 하면 코딩 키트의 스위칭 시스템 설정이 완료되었고 이제부터는 라즈베리파이의 핀들이 코딩 키트의 디바이스에 연결되는 것입니다.

Note : 코딩 키트의 전원 스위치를 올리면 라즈베리파이에도 전원이 공급됩니다. 따로 아답터를 라즈베리파이에 연결하지 않으셔도 됩니다. 그래서 라즈베리파이를 코딩 키트에 장착해서만 사용하실 것이라면 라즈베리파이용 아답터를 따로 구매하지 않으셔도 됩니다. 그리고 라즈베리파이를 코딩 키트에 장착하고는 따로 라즈베리파이에 전원을 연결하지 마십시요. **라즈베리파이의 마이크로 USB 커넥터에는 아답터 및 USB 케이블을 절대로 연결하지 마십시요.**

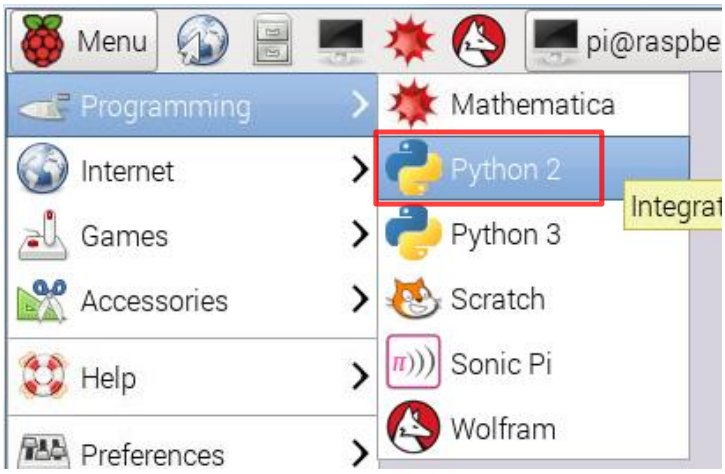
[파이썬과 라즈베리파이]

라즈베리파이(Rapberry Pi)의 파이(Pi)는 파이썬(Python)이라는 컴퓨터 프로그램 언어에서 따온 것입니다. 그런 만큼 라즈베리파이는 파이썬을 기본 프로그램 언어로 사용하고 있습니다. 파이썬은 매우 쉬운 언어입니다. 앞에서 코딩 키트를 활용하여 아두이노 코딩을 열심히 공부한 사람은 매우 쉽게 파이썬 코드를 작성할 수 있을 것입니다.

출처 : www.python.org

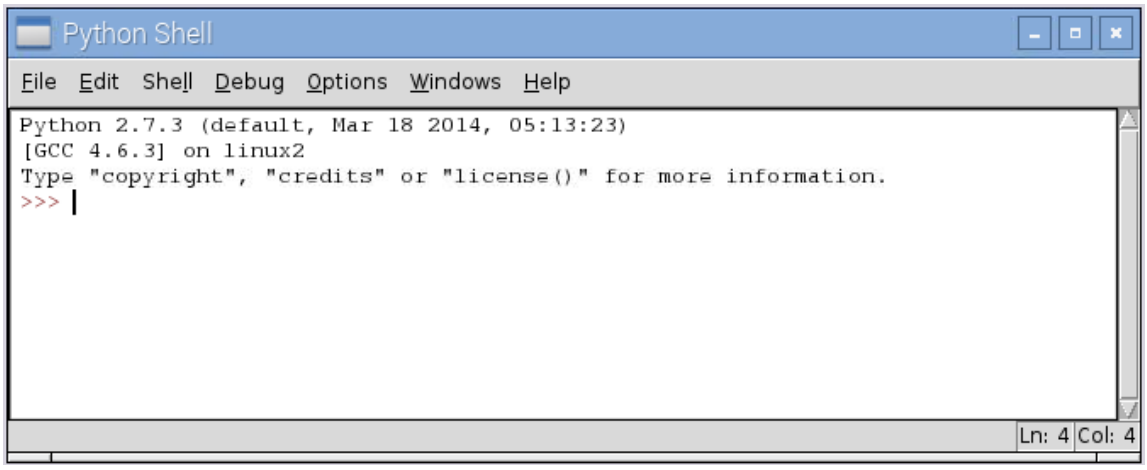
파이썬은 1991 년 귀도 반 로섬(Guido Van Rossum)이 발표한 프로그램 언어입니다. 이 프로그램 언어는 C 언어하고는 다르게 한 줄 한 줄씩 실행이 되는 인터프리터 방식입니다. 또한 특정 CPU 나 하드웨어에 국한되어 있지 않고 어떠한 플랫폼에서도 자유롭게 사용할 수 있는 프로그램 언어입니다. 파이썬은 현재 매우 많은 사람들이 사용하고 있습니다. 그러다 보니 라이브러리가 매우 풍부하여 여러분이 파이썬 코딩을 하실 때 더욱 더 쉽고 빠르게 코딩할 수 있습니다. 파이썬 공식 홈페이지(https://www.python.org/)를 방문하시면 더 많은 정보를 얻을 수 있습니다.

아두이노에서는 아두이노 프로그램을 이용하여 아두이노 코드를 컴파일하고 실행하였습니다. 라즈베리파이에서도 파이썬을 컴파일해 주고 실행해 주는 프로그램이 있습니다. 라즈베리파이 화면의 좌측상단의 "Menu → Programming → Python 2" 를 클릭하여 실행시키면 Python 2 프로그램이 실행됩니다.

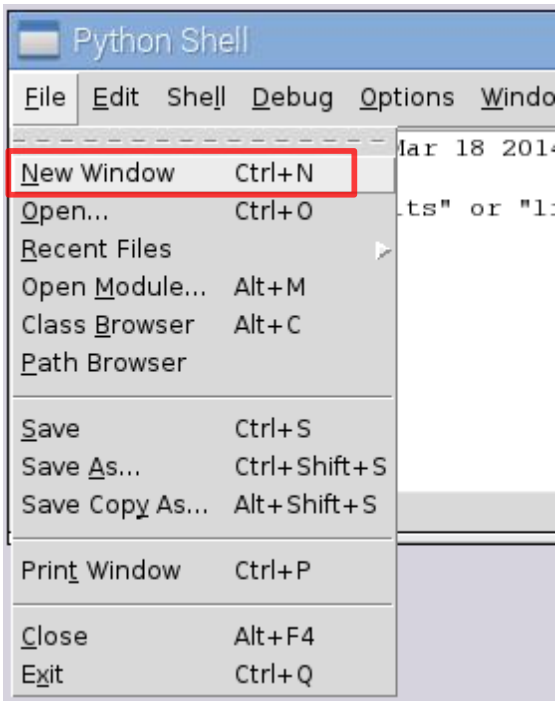


Python 3 도 있지만 코딩 키트에서는 Python 2 를 이용하겠습니다.

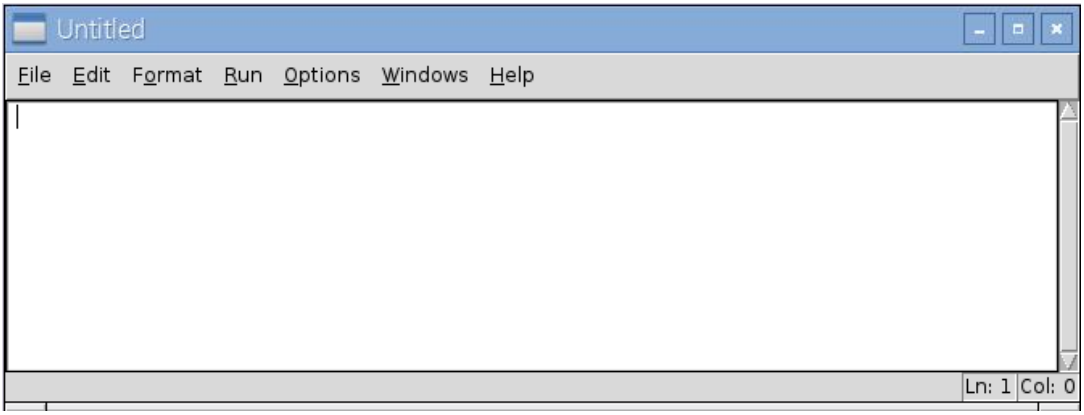
실행 시키면 다음과 같이 창이 나옵니다.



여기서 "File → New Window" 메뉴를 클릭합니다.



다음과 같은 새 창이 열립니다.



앞으로 여러분이 여기에 코딩을 할 것입니다. 그럼 여기에 간단하게 LED 한 개를 켜고 끄는 코드를 작성해보겠습니다.

아두이노에서 GPIO(General Purpose Input Output)라는 것을 배웠을 것입니다. 이것은 어떤 핀에 0 혹은 1 값을 쓰거나 읽는 것을 말합니다. 라즈베리파에서도 이 GPIO를 이용하여 장착 디바이스들을 컨트롤해보겠습니다. 아두이노에서는 바로 GPIO 를 사용하였는데, 라즈베리파이에서는 GPIO 라이브러리를 이용합니다. 이 라이브러리를 이용하려면 먼저 이것을 프로그램으로 불러내야 합니다. 아두이노에서는 라이브러리를 사용할 때 "#include" 를 썼는데, 파이썬에서는 다음과 같이 import 를 이용 합니다.

```
import RPi.GPIO as GPIO
```

"import" 라고 쓴 다음 라이브러리 이름을 씁니다. as 뒤는 이 라이브러리를 다른이름으로 고쳐 쓴 것입니다. 즉, RPi.GPIO 를 GPIO 로 이름 바꾸기를 한 것입니다. 그래서 이 이후의 코드에서는 GPIO 라고 씁니다. import 에 관해서는 추후 더 자세한 설명을 해 드리겠습니다.

다음으로는 입출력을 위한 핀을 지정합니다. 아두이노에서 "#define LED 22" 라고 정의한 부분과 같은 부분입니다.

```
LED = 22
```

파이썬에서는 간단하게 LED 라고 쓰고 22 를 등호(=)로 할당해 줍니다. 파이썬에서는 이렇게 변수를 선언합니다. 아두이노에서 변수 타입을 써서 변수를 선언하는 것과는 다르게 변수 타입 없이 변수를 선언합니다. 그렇다고 해서 변수 타입이 없는 것은 아니고 처음에 저장하는 데이터의 타입에 따라서 변수의 타입이 정해집니다. 또한 파이썬에서는 변수를 선언하는 것과 사용하는 것이 명확하게 구분되어 있지도 않습니다. 그래서 변수를 선언한다고 하기도 좀 애매합니다. 이 부분에 대해서는 앞으로 코딩 북을 따라서 코딩을 해 나가다 보면 감이 잡힐 것입니다. 그렇게 감이 잡힐 때쯤 변수에 대해서 정리하고 넘어가겠습니다.

아두이노의 pinMode() 함수와 같이 핀의 입출력을 정해 주어야 합니다. 라즈베리파이에서는 GPIO.setup 이라는 함수를 사용합니다. 사용되는 인자는 아두이노와 똑같이 핀번호와, 입출력 모드입니다. 그래서 LED 핀은 출력으로 다음과 같이 코딩합니다.

```
GPIO.setup(LED, GPIO.OUT)
```

만약 입력으로 사용한다면 GPIO.OUT 대신에 GPIO.IN을 사용하면 됩니다.

LED 를 출력용으로 설정하였으면, 이제 LED 가 켜지는 동작을 시킬 수 있습니다. LED 가 켜지는 것은 GPIO.output 함수에 핀 번호와 1 값을, 꺼지는 것은 핀 번호와 0 값을 주면 됩니다. 그래서 다음과 같이 하면 LED 가 켜지고 꺼집니다.

```
GPIO.output(LED, 1)
```

```
GPIO.output(LED, 0)
```

이것은 아두이노의 digitalWrite() 함수와 같습니다. 두 함수가 받는 인자도 같고 너무나도 똑같습니다. 이것은 컴퓨터 언어가 다르기는 하지만 언어의 구조가 비슷하고, 모두 다 컴퓨터에서 쓰는 언어이기 때문에 다른 언어에 같은 함수가 만들어지는 것입니다. 그러서 여러분이 컴퓨터 언어 하나를 제대로 배워 두면 다른 언어도 금방 쉽게 습득할 수 있습니다. 여러분이 앞에서 아두이노 코드를 잘해 두셔서 파이썬도 금방 하는 것입니다. 아두이노 코드는 C 언어라고 생각하시면 됩니다.

LED 가 켜졌다 꺼졌다를 계속해서 반복하려면 아두이노의 loop() 함수와 같은 함수가 있어야 합니다. while 문을 이용하겠습니다. while 문 안의 내용을 계속해서 반복합니다.

파이썬의 while 문은 다음과 같습니다.

```
while 조건식 :
    문장
```

C 언어의 while 과 비슷합니다. 조건식 다음에 콜론(:)을 사용합니다. 이것은 C 언어의 중괄호와 같습니다. 그 다음 문장은 while 문 안의 문장이라는 의미로 4 칸 들여쓰기를 합니다. 꼭 4 칸이 아니어도 되지만, 코딩 키트에서 들여쓰기는 4 칸으로 통일하겠습니다. while 문은 무한 반복을 해야 하기 때문에 조건식에 1 을 써 줍니다. 그러면 아두이노 코드의 loop() 함수와 같게 되는 것입니다.

```
while 1 :
    GPIO.output(LED, 1)
    GPIO.output(LED, 0)
```

그런데, 이렇게 코드를 작성하면 LED 가 너무 빠르게 켜졌다 꺼집니다. 사람 눈으로는 LED가 깜박이는지 알 수 없습니다. 1 초마다 또는 2 초마다 켜졌다 꺼졌다를 반복하게 하기 위해서 시간 라이브러리를 사용합니다. 이때도 import 문을 사용하면 됩니다. GPIO 라이브러리를 불러오는 코드 다음에 시간 라이브러리를 불러오는 코드를 다음과 같이 추가합니다.

```
import time
```

그리고, time 라이브러리의 sleep() 함수를 사용하여 LED를 켜고 끄는 동작에 시간 간격을 정해줍니다.

```
while 1 :
    GPIO.output(LED, 1)
    time.sleep(3)
    GPIO.output(LED, 0)
    time.sleep(2)
```

이 sleep() 함수는 아두이노 코드에서 delay() 함수와 같은 것입니다. 차이는 delay() 함수의 단위가 1000 분의 1 초였다면 이 sleep() 함수는 초 단위입니다. 그래서 1000 분의 1 초는 0.001 초로 씁니다. 그런데, 왜 함수 이름을 sleep (자다) 이라고 했을까요? 이것은 컴퓨터 입장에서 이름을 지은 것입니다. 컴퓨터는 쉬지 않고 일합니다. 계속해서 무언가를 하지요. 그래서 sleep 이라고 하면 자면서 잠시 쉬라는 뜻입니다.

컴퓨터 언어에서는 비슷한 용어로 idle(일하지 않는, 비어 있는) 이란 용어도 많이 사용합니다. 그리고 함수의 인자로 시간을 주면 그 시간동안 아무것도 하지말고 기다려라 하는 뜻입니다. 코딩하는 입장에서는 아두이노의 delay 라는 함수 이름이 좀 더 이해하기 쉽습니다.

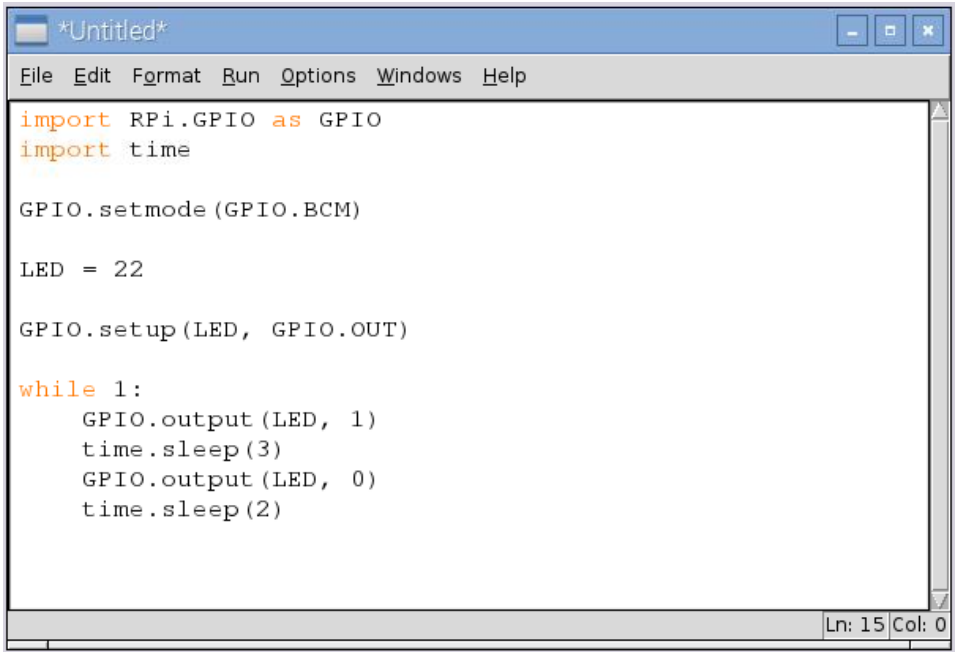
이제 위에서 설명한 것과 같이 LED 를 3 초간 켜고 2 초간 끄는 예제를 완성해 보겠습니다.

```
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BCM)

LED = 22

GPIO.setup(LED, GPIO.OUT)

while 1:
    GPIO.output(LED, 1)
    time.sleep(3)
    GPIO.output(LED, 0)
    time.sleep(2)
```

< 예제 → **led_3sec_on_2sec_off.py**>

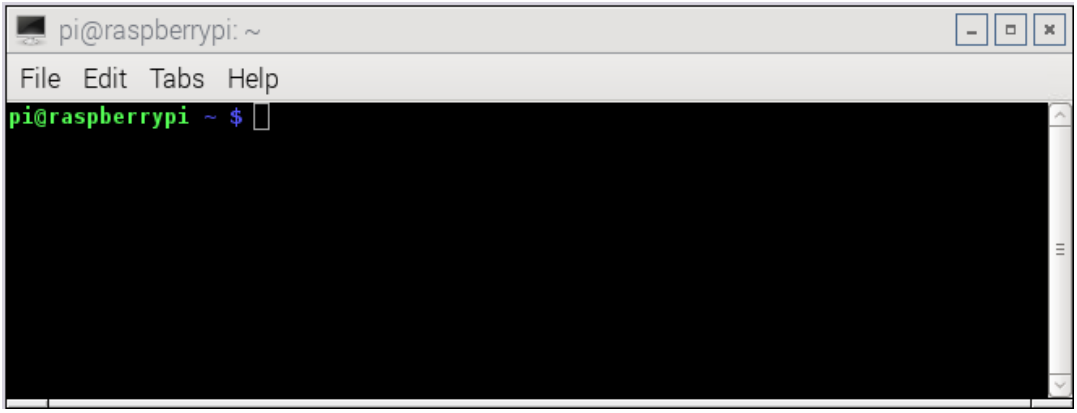
위 코드에서 설명드리지 않은 부분이 "GPIO.setmode(GPIO.BCM)" 입니다. 이것은 GPIO 모드를 설정하는 부분입니다. GPIO 번호가 실제 보드의 핀 번호가 아니라 라즈베리파이 CPU 칩의 GPIO 채널 번호라는 표시이다. 이 문장의 BCM 은 라즈베리파이 CPU 칩의 부품 번호가 BCM 으로 시작하기 때문입니다. 앞으로 이 문장은 라즈베리파이의 파이썬 코드에서 GPIO 를 사용할 때는 항상 같게 사용할 것입니다. 그래서 채널 번호 등의 이해하기 어려운 내용이 나와도 코딩하는데는 지장이 없습니다.

이 코드는 다음과 같이 Python 2 프로그램의 "New Window" 에 입력하셨을 것입니다.



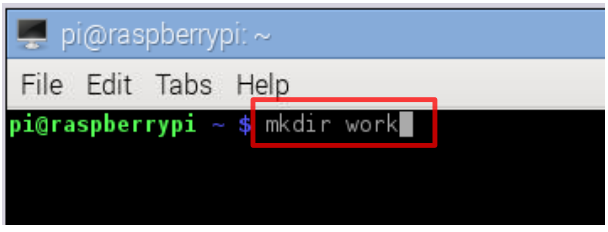
그러면 이제 코드를 저장하기 전에 터미널(Terminal) 프로그램에서 "work" 폴더를 하나 만들겠습니다. 다음 그림의 모니터 모양의 아이콘을 클릭하여 터미널 프로그램이 실행됩니다.

터미널을 실행시키면 다음과 같은 창이 나타납니다. 그러면 여러분은 여기에 리눅스 명령어를 입력합니다.



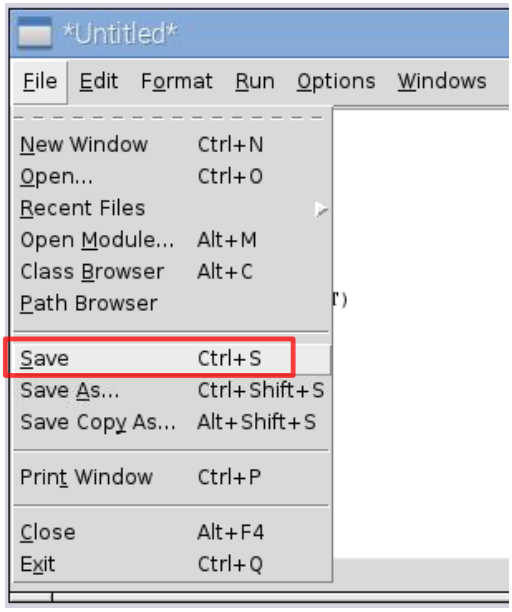
이 터미널이라는 프로그램은 리눅스 명령을 받거나 관련된 메시지를 출력해주는 프로그램입니다. 리눅스를 한 번도 다루어보지 않은 분이라면 많이 생소해 보이고 어려워 보일 수도 있습니다. 하지만 이러한 리눅스 관련 작업은 매우 간단한 몇 가지만 반복하여 사용하기 때문에 잘 모르신다고 하여 크게 걱정하지 않으셔도 됩니다. 그리고 몇 번 해 보면 금방 익숙해 지실 것입니다. 관련된 리눅스 명령어 몇 가지는 **[부록 B]** 에 정리해 두었습니다.

터미널 창에 "mkdir work" 이라고 타이핑 하고 키보드의 "Enter" 를 누르면 work 폴더가 생깁니다. mkdir 은 Make Directory 라는 의미로 현재 위치에 디렉토리 또는 폴더를 만듭니다.

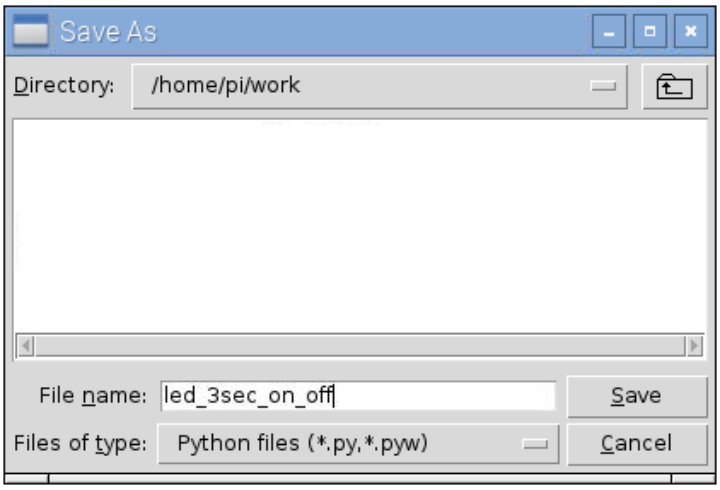


work 폴더가 생긴 것은 터미널에서 "ls"를 타이핑하면 확인할 수 있습니다. ls 는 List 의 약자로 현재 디렉토리에 있는 파일 및 하위 디렉토리의 이름을 보여줍니다.

이제 "File → Save" 메뉴를 이용하여 코드를 저장합니다.



다음과 같은 창이 나오면 work 폴더를 선택하고 "led_3sec_on_off.py" 라고 저장합니다. 여기서 py 라는 확장자는 생략하여도 자동으로 붙여줍니다.

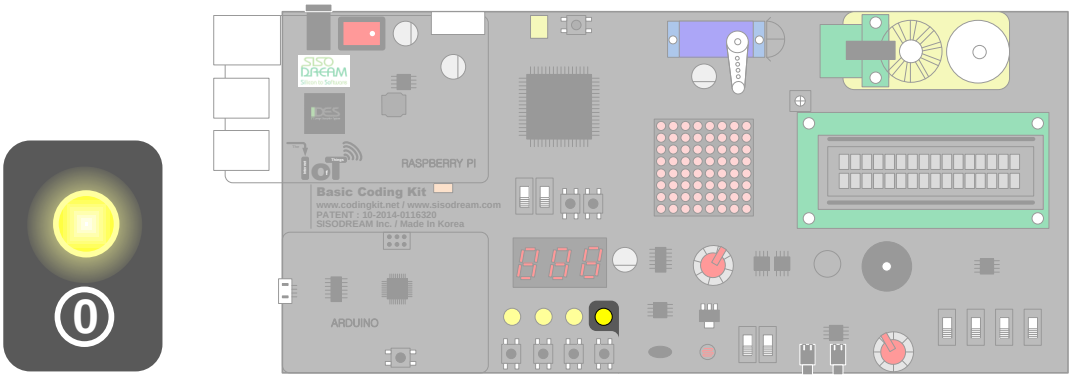


이제 파이썬 코드를 코딩 키트에서 실행시켜 보겠습니다. 터미널 프로그램에서 코드가 저장되어 있는 work 폴더로 이동하기 위해서 "cd work" 이라고 씁니다. 그러면 work 폴더로 이동이 됩니다. work 폴더에서 "sudo python led_3sec_on_off.py" 라고 타이핑하고 엔터키를 누르면 코드가 실행이 됩니다.

```
pi@raspberrypi ~ $ cd work
pi@raspberrypi ~/work $ sudo python led_3sec_on_off.py
```

여기서 "sudo" 라는 것은 "관리자 권한" 이라는 뜻입니다. 여기서 관리자 권한이 필요한 이유는 라즈베리파이에서 자신의 핀에 어떤 데이터를 쓰기 때문입니다. "python" 은 python 프로그램을 실행하라는 뜻입니다. "led_3sec_on_off.py" 는 python 프로그램을 실행할 때 이 코드를 사용하라는 뜻입니다.

이제 코딩 키트에서 LED 가 깜박이는 것을 확인할 수 있습니다.



< 예제 코드 : **LED 3 초간 켜고 2 초간 끄기** >

< 코드 위치 : Codingkit / **led_3sec_on_2sec_off.py** >

위의 코드 위치의 표시는 /home/pi/Codingkit/led_3sec_on_2sec_off.py 파일을 나타내는 것입니다. 코딩 키트의 라즈베리파이 키트를 구매하셔서 받은 마이크로 SD 카드로 라즈베리파이를 부팅하셨다면, /home/pi/Codingkit 디렉토리에 코딩 북에서 예제로 사용하는 모든 코드들이 들어 있습니다. 코딩 공부하시는데 많은 참고 자료가 될 것입니다.

파이썬 프로그램을 끝낼 때는 터미널에서 키보드의 Ctrl 키와 c 키를 동시에 누르면 프로그램을 끝낼 수 있습니다. PC 에서 복사와 같은 키 조합니다.

코드를 다시 실행 시키면 아래와 같은 "RuntimeWarning" 이라는 경고 메세지가 발생할 수도 있습니다.

```
pi@raspberrypi ~/Codingkit $ sudo python led_3sec_on_off.py
led_3sec_on_off.py:8: RuntimeWarning: This channel is already in use, continuing
 anyway.  Use GPIO.setwarnings(False) to disable warnings.
  GPIO.setup(LED, GPIO.OUT)
```

이것은 GPIO 핀이 이미 사용중이라는 메시지 입니다. 이전 프로그램에서 GPIO 를 정리하지 않고 끝내서 이런 메시지가 나오는 것입니다. 나중에 이것과 관련하여 GPIO 를 정리하는 방법을 알려드리겠습니다. 일단은 무시하고 넘어 가시면 됩니다.

< 문법 설명 : **4 칸 들여쓰기로 블럭 지정** >

위의 코드에서 while 문을 보면 다음과 같습니다.

```
while 1:
    GPIO.output(LED, 1)
    time.sleep(3)
    GPIO.output(LED, 0)
    time.sleep(2)
```

while 1 한 다음 콜론(:)을 씁니다. 이 콜론 다음의 문장들은 while 문에 포함되는 블럭입니다. 위의 코드에서 붉은색으로 되어 있는 부분이 모두 while 문 안의 블럭인 것입니다. C 언어에서는 중괄호로 처리했던 부분입니다. 이 문장들은 모두 4 칸 들여쓰기가 되어 있습니다. 꼭 4 칸이 아니어도 되지만 코딩 키트에서는 4 칸을 사용합니다. 이렇게 4 칸 들여쓰기가 되어 있는 부분은 모두 한 블럭이 되는 것입니다. 이 블럭을 끝내려면 4 칸 들여쓴 것을 제거하면 됩니다. 아래 코드에서 붉은색으로 쓰인 문장은 while 문 밖에 있는 것입니다.

```
while 1:
    GPIO.output(LED, 1)
    time.sleep(3)
    GPIO.output(LED, 0)
    time.sleep(2)
문장
```

정리하면 파이썬에서는 어느 구문에 포함된 블럭을 4 칸 들여쓰기로 표시합니다. 포함된 블럭을 끝내려면 4 칸 들여쓰기 한 것을 제거하면 됩니다. 어떤 구문의 시작은 콜론(:)으로 표시하고 4 칸 들여쓰기를 통하여 포함 관계를 표시합니다. 아래와 같이 중첩된 불록 관계도 코딩할 수 있습니다.

```
구문 1:
    구문 1 에_포함된_문장 1
    구문 1 에_포함된_문장 2
    구문 2:
        구문 2 에_포함된_문장 1
        구문 3:
            구문 3 에_포함된_문장 1
        구문 2 에_포함된_문장 2
    구문 1 에_포함된_문장 3
문장 1
문장 2
...
```

Python 2 프로그램에서는 자동으로 들여쓰기를 해 줍니다.

< 문법 설명 : **주석** >

파이썬에서 주석은 문법적으로 다음과 같이 표시 합니다.

```
"""
  This is the standard way to
  Include a multiple-line comment in
  Your code.
"""
```

파이썬 주석은 위와 같이 **큰따옴표 세 개**를 주석처리하려는 내용의 양 끝에 사용하여 만듭니다. C 언어에서 주석을 /* 과 */ 으로 처리하는 것과 비슷하게 여러 라인을 주석처리할 수 있습니다. 또는 한 라인 안에서 중간 부분만을 주석 처리할 수 있습니다.

그리고 다음과 같이 샵(#)을 넣으면 그 위치부터 그 라인의 끝까지는 주석으로 처리합니다.

```
# I love Python.
```

샵(#)이 큰따옴표 세 개가 있는 주석 안에 있다면 큰따옴표 셋이 우선하기 때문에 샵(#)도 주석이 됩니다.

< 문법 설명 : **while** >

파이썬에서 반복적인 작업을 수행해야 할 경우 while 문, for 문 등의 loop 문을 사용합니다. while 키워드는 반복문이 시작됨을 알려줍니다.

```
while 조건식:
  문장 1
  문장 2
  ...
```

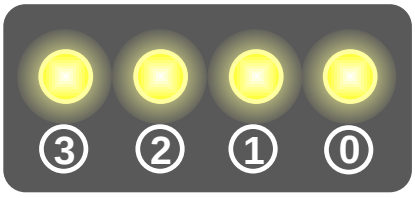
조건식이 참이면 while 문 안의 문장들을 수행합니다. 조건식이 거짓이 되면 while 문을 끝냅니다. 조건식에 1 값을 쓰면 조건식이 무조건 참이 되기 때문에 while 문을 계속해서 반복적으로 실행합니다.

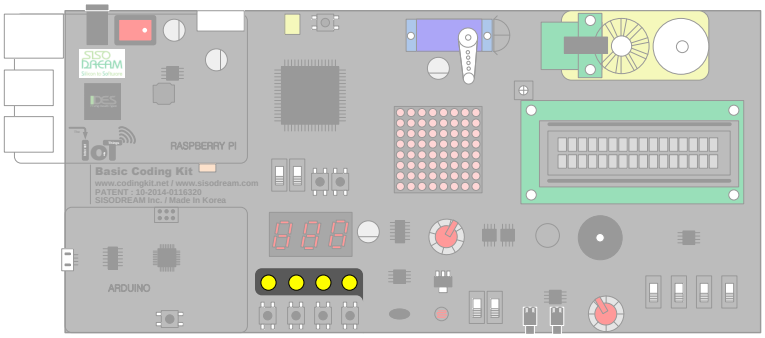
< 연습 문제 : **LED 4 개 1 초 간격으로 깜박이기** >

코딩 키트에 있는 LED 4개를 모두 1초 동안 켜고, 1초 동안 끄는 코딩을 해 봅니다. 즉, LED 모두 1초 간격으로 깜박이는 예제입니다. LED 각각의 핀 번호는 다음과 같습니다.

LED_0 = 22
LED_1 = 23
LED_2 = 24
LED_3 = 25

< 코드 위치 : Codingkit / **led4_on_off.py >**





[신호 입력 받아 출력하기]

이번에는 버튼을 누르면 LED 가 켜지는 코딩을 해 보겠습니다. 버튼의 핀 번호는 다음과 같이 설정합니다.

```
BUTTON = 4
```

버튼은 컨트롤 보드에 버튼이 눌렸는지 아닌지에 대한 값을 전달합니다. 컨트롤 보드 입장에서는 버튼의 출력은 자신에게 들어오는 입력이므로, 핀 모드는 다음과 같이 코딩합니다.

```
GPIO.setup(BUTTON, GPIO.IN)
```

버튼과 연결된 핀의 값을 읽는 것은 다음과 같이 코딩합니다. 즉, 입력되는 값을 읽는 것입니다.

```
GPIO.input(BUTTON)
```

이것은 GPIO.output() 함수가 어떤 핀에 값을 쓰라는 것과는 반대로 어떤 핀의 값을 읽어 오라는 뜻입니다. 이 값에 따라서 LED 를 켤지 끌지를 결정해야 합니다. 그것은 다음과 같이 코딩합니다.

```
if (GPIO.input(BUTTON) == BUTTON_PRESSED) :
   GPIO.output(LED, ON)
else :
    GPIO.output(LED, OFF)
```

GPIO.input가 버튼의 값을 읽어와서 그 값이 BUTTON_PRESSED 와 같다면, 즉 버튼이 눌려졌다면, LED를 켜고, 그렇치 않다면(else) LED를 끕니다. 그런데, 버튼은 눌리면 0 값을 주고, 눌리지 않으면 1 값을 줍니다. 그래서 BUTTON_PRESSED 는 다음과 같이 정의합니다.

```
BUTTON_PRESSED = 0
```

이렇게 하여 우리가 원하는 "버튼이 눌리면 LED 를 켜는" 코드가 완성된 것입니다. 여기에 다음과 같이 몇 가지를 정의해 주면 됩니다.

```
LED = 22
ON = 1
OFF = 0
```

LED 의 핀 번호는 22 번이고, ON, OFF 를 정의합니다.

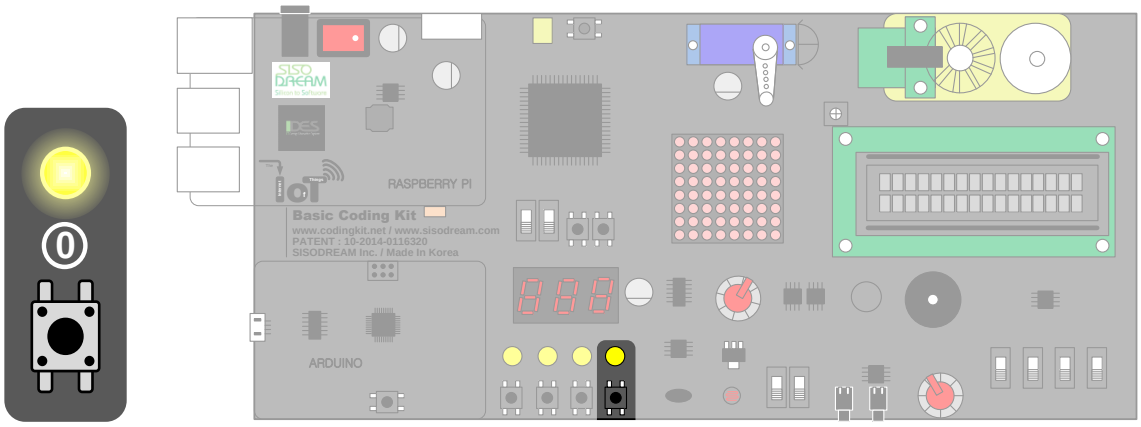
이상의 내용을 바탕으로 코딩하면 다음과 같습니다.

```
import RPi.GPIO as GPIO

GPIO.setmode(GPIO.BCM)

LED = 22
BUTTON = 4
BUTTON_PRESSED = 0
```

```
ON = 1
OFF = 0

GPIO.setup(LED, GPIO.OUT)
GPIO.setup(BUTTON, GPIO.IN)

while 1:
   if (GPIO.input(BUTTON) == BUTTON_PRESSED):
      GPIO.output(LED, ON)
   else:
      GPIO.output(LED, OFF)
```

< 예제 코드 : **버튼이 눌리면 LED 켜기** >

< 코드 위치 : Codingkit / **led_on_off_but.py** >



버튼을 누르면 LED 4개가 모두 켜지는 예제를 해 보겠습니다. 코드는 LED 한 개를 켜고 끄는 것과 매우 비슷합니다.

```
import RPi.GPIO as GPIO

GPIO.setmode(GPIO.BCM)

LED_0 = 22
LED_1 = 23
LED_2 = 24
LED_3 = 25

BUTTON = 4
BUTTON_PRESSED = 0
ON = 1
OFF = 0
```

```
GPIO.setup(LED_0, GPIO.OUT)
GPIO.setup(LED_1, GPIO.OUT)
GPIO.setup(LED_2, GPIO.OUT)
GPIO.setup(LED_3, GPIO.OUT)

GPIO.setup(BUTTON, GPIO.IN)

while 1:
  if (GPIO.input(BUTTON) == BUTTON_PRESSED):
    GPIO.output(LED_0, 1)
    GPIO.output(LED_1, 1)
    GPIO.output(LED_2, 1)
    GPIO.output(LED_3, 1)
  else:
    GPIO.output(LED_0, 0)
    GPIO.output(LED_1, 0)
    GPIO.output(LED_2, 0)
    GPIO.output(LED_3, 0)
```

이 코드에서 중요하게 봐 두어야 할 것은 들여 쓰기 부분입니다. 코드 중 붉은색 부분은 들여쓰기가 되어 있습니다. 이것은 if 뒤의 괄호 안의 값이 맞으면 if 문 아래 들여쓰기 된 모든 구문이 다 수행됩니다. 그리고 else 이후의 코드에도 들여쓰기가 되어 있습니다. 조건이 맞지 않다면 else 이후의 들여쓰기가 된 문장들이 모두 수행됩니다.

크게는 while 문 아래 문장들이 들여쓰기가 되어 있습니다. 버튼이 눌려지면 LED가 켜지고 그렇지 않으면 LED가 꺼지는 동작을 무한 반복한다는 의미입니다.

< 예제 코드 : **버튼이 눌리면 LED 4 개 켜기 >**

< 코드 위치 : Codingkit / **led4_on_off_but >**

< 문법 설명 : **변수** >

다음의 예제는 앞서 작성한 버튼에 의해 LED 를 켜고 끄는 코드입니다. 버튼이 눌리면 LED를 켜고, 아니면 LED 를 끄는 동작을 합니다.

```
if (GPIO.input(BUTTON) == BUTTON_PRESSED) :
  GPIO.output(LED, ON)
else :
  GPIO.output(LED, OFF)
```

여기서 버튼(BUTTON)에서 입력하는 값을 변수에 저장해보겠습니다. 파이썬에서는 변수를 선언을 따로 하지 않고 바로 사용합니다. 변수의 타입은 입력 되는 값에 의해서 결정됩니다.

```
button_value = GPIO.input(BUTTON)

if (button_value == BUTTON_PRESSED)  :
  GPIO.output(LED, ON)
else :
  GPIO.output(LED, OFF)
```

변수 button_value 에는 GPIO.input() 함수에서 전달되는 값이 저장됩니다. 그러면 button_value 는 버튼이 눌렸는지 안 눌렸는지를 저장하고 있습니다. 그리고 이 이후의 문장들에서는 GPIO.input(BUTTON) 대신 button_value 를 사용하면 됩니다. 코드는 다음과 같습니다.

```
import RPi.GPIO as GPIO

GPIO.setmode(GPIO.BCM)

LED = 22
BUTTON = 4
BUTTON_PRESSED = 0
ON = 1
OFF = 0

GPIO.setup(LED, GPIO.OUT)
GPIO.setup(BUTTON, GPIO.IN)

while 1:
   button_value = GPIO.input(BUTTON)
   if (button_value == BUTTON_PRESSED):
      GPIO.output(LED, ON)
   else:
      GPIO.output(LED, OFF)
```

< 예제 코드 : **버튼 값을 변수에 저장하기** >

< 코드 위치 : Codingkit / **led_on_off_but_var.py** >

< 문법 설명 : **if** >

앞의 코드에서 나왔던 "if 문" 에 대해서 알아 보겠습니다.

```
if (조건식) :
  문장1
  문장2
  ...
else :
  문장1
```

```
문장1
...
```

if 키워드는 if 문의 시작을 알려줍니다. if 키워드 다음에는 만족해야하는 "조건식"이 나옵니다. 이 조건식 다음에 콜론이 옵니다. 조건식을 감싸고 있는 괄호는 생략하여도 됩니다. 다음행에 한단계 들여쓰기를 하고 조건이 참인 경우 실행되는 문장들을 씁니다. 만약 조건식이 거짓일 경우에 실행되어야 하는 문장이 있다면 else 키워드를 사용합니다. else 키워드 다음에 콜론(:)이 옵니다. 그리고 다음행에 한단계 들여쓰기를 하고 조건식이 거짓일 경우 실행되는 문장들을 들여쓰기 하여 씁니다.

< 연습 문제 : **버튼이 눌리지 않았을 때 LED 켜기** >

다음은 앞에서 작성했던 코드입니다. if 문에 의해서 버튼이 눌려졌는지를 확인하여, 버튼이 눌려졌다면 (조건이 참이면) LED를 켜고, 버튼이 눌려지지 않았다면(조건이 거짓이면) LED를 끕니다.

```
button_value = GPIO.input(BUTTON)

if (button_value == BUTTON_PRESS) :
GPIO.output(LED, ON)
else :
GPIO.output(LED, OFF)
```

여기서 간단한 예제를 하나 해 볼까요? 버튼이 눌리지 않으면 LED 가 켜지는 예제를 해 보겠습니다. if 문에 "==" 기호 대신 "!=" 기호를 사용하면 간단히 해결될 것 같습니다. 직접 한 번 해 보시지요.

< 코드 위치 : Codingkit / **led_on_but_not.py** >

"==", "!=" 기호를 관계 연산자라고 합니다. 파이썬의 관계 연산자는 다음 표와 같이 C 언어와 동일합니다.

기호	의미
==	같다
!=	같지 않다
<	작다
<=	작거나 같다
>	크다
>=	크거나 같다

여기서 if 문에 대해서 더 자세히 알아 볼까요? if 문에는 다음과 같이 C 언어에서 보았던 "else if" 와 같은 "elif" 문을 쓸 수 있습니다.

```
if 조건식1 :
   문장1
   문장2
   …
elif 조건식2 :
   문장3
   문장4
   …
else :
   문장5
   문장6
   …
```

if 혹은 elif 다음의 조건식이 참이면 바로 다음행의 들여쓰기된 문장들을 수행합니다. 즉, 조건식1이 참이면, 문장1과 문장2가 실행이 되고, 조건식2가 참이면, 문장3과 문장4가 실행이됩니다. if, elif 다음 조건식들이 모두 거짓이면 else 다음의 들여쓰기된 문장5와 문장6이 수행됩니다. 여기서 elif 는 단 하나도 안 쓰일 수도 있고 매우 많이 쓰일 수도 있습니다. else 는 단 한번만 쓰일 수도 있고 안 쓰일 수도 있습니다.

다음과 같이 들여쓰기 단계를 사용하여 중복적으로 사용할 수도 있습니다.

```
if 조건식1 :
   문장1
   문장2
elif 조건식2 :
   if 조건식3 :
      문장3
      문장4
   elif 조건식4 :
      문장5
      문장6

   else :
      문장7
      문장8
```

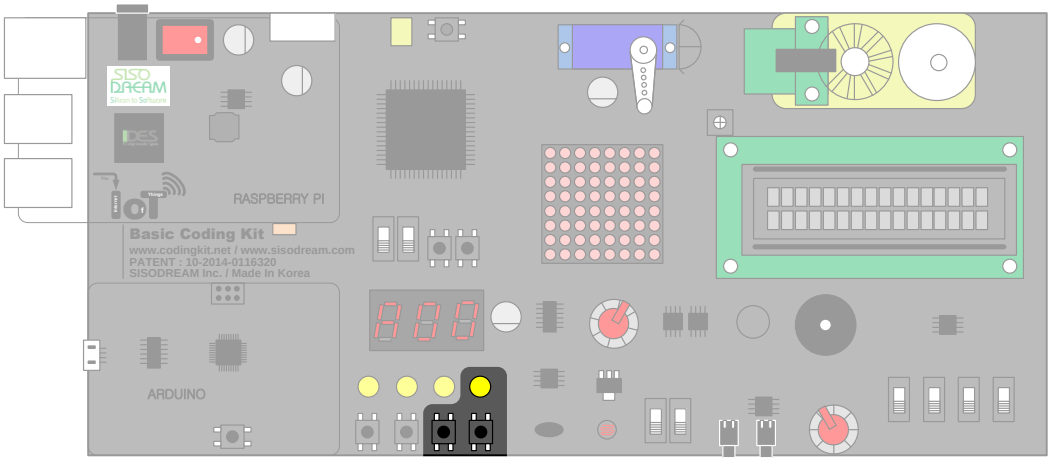
위의 코드에서 조건식1이 참이면 조건식1 다음행에서 들여쓰기를 한 문장1과 문장2가 수행됩니다. 조건식1이 거짓이고 조건식2가 참이면 조건식2 다음행에서 한단계 들여쓰기를 한 조건식문들이 수행됩니다. 그래서 조건식2가 참이고 조건식3이 참이면 문장3과 문장4가 수행이 됩니다. 모든 경우를 표로 나타내면 다음

고 같습니다.

조건식1	조건식2	조건식3	조건식4	수행문장
참	-	-	-	문장1, 문장2
거짓	참	참	-	문장3, 문장4
거짓	참	거짓	참	문장5, 문장6
거짓	참	거짓	거짓	문장7, 문장8

< 문법 설명 : **논리 연산자** >

이번에는 버튼 2개를 이용한 예제를 해 보겠습니다.



버튼 2개가 모두 눌려야만 LED 가 켜지는 예제입니다. 먼저 코드를 한 번 볼까요?

```
import RPi.GPIO as GPIO

GPIO.setmode(GPIO.BCM)

LED = 22
BUTTON_0 = 4
BUTTON_1 = 5
BUTTON_PRESSED = 0
ON = 1
OFF = 0

GPIO.setup(LED, GPIO.OUT)
GPIO.setup(BUTTON_0, GPIO.IN)
GPIO.setup(BUTTON_1, GPIO.IN)

while 1:
    but0_value = GPIO.input(BUTTON_0)
```

```
but1_value = GPIO.input(BUTTON_1)

if ((but0_value == BUTTON_PRESSED) and (but1_value == BUTTON_PRESSED)):
    GPIO.output(LED, ON)
else:
    GPIO.output(LED, OFF)
```

< 예제 코드 : **AND 연산자를 이용한 LED 켜기** >

< 코드 위치 : Codingkit / **led_on_but_and.py** >

LED 는 출력 모드로, 버튼 두개를 입력 모드로 하였습니다. but0_value 와 but1_value 에 각각 버튼 값을 읽어 저장해 두었습니다. 그 다음 문장부터가 중요합니다.

"if ((but0_value == BUTTON_PRESSED) and (but1_value == BUTTON_PRESSED))" 는 "버튼 0번이 눌렸고, 버튼 1번도 눌렸으면" 하는 뜻입니다. 여기서 and 표시는 "AND 연산자" 또는 "논리곱" 이라고 불리며, and 양쪽의 값이 모두 참이면 결과 값으로 참을 내어 줍니다.

조건1 and 조건2

조건1과 조건2가 모두 참이어야 결과 값이 참이 됩니다.

AND 가 있으면 OR 가 있고, 논리곱이 있으면 논리합이 있겠지요. OR 연산자는 **or** 키워드로 씁니다.

조건1 or 조건2

조건1과 조건2 둘중 한가지 조건만 참이어도 결과 값이 참이 됩니다.

< 연습 문제 : **OR 연산자를 이용한 LED 켜기** >

그러면 여기서 간단한 예제를 하나를 해 볼까요? 버튼 둘 중 하나라도 눌리면 LED 가 켜지는 예제는 어떻게 할까요? OR 연산자를 이용해서 직접 한 번 해 보십시요.

< 코드 위치 : Codingkit / **led_on_but_or.py** >

AND 와 OR 연산자 이외에 논리 연산자가 하나 더 있습니다. 바로 NOT 연산자 또는 "논리 부정" 입니다. 이 연산자는 AND 와 OR 연산자가 좌우에 두 개의 값을 연산하는 것과는 달리 자신의 오른쪽의 값만을 반대로 만듭니다. 키워드 **not**을 사용합니다.

not (조건 또는 값)

조건 또는 값의 반대값을 결과로 내어줍니다.

간단한 예제 하나 해 보겠습니다. 전에 버튼이 눌리면 LED 가 켜지는 아래와 같은 코드를 작성했었습니다. 이 중 붉은색으로 표시된 부분은 NOT 연산자를 사용하면 매우 간단하게 바꿀 수 있습니다.

```
import RPi.GPIO as GPIO

GPIO.setmode(GPIO.BCM)

LED = 22
BUTTON = 4
BUTTON_PRESSED = 0
ON = 1
OFF = 0

GPIO.setup(LED, GPIO.OUT)
GPIO.setup(BUTTON, GPIO.IN)

while 1:
   if (GPIO.input(BUTTON) == BUTTON_PRESSED):
      GPIO.output(LED, ON)
   else:
      GPIO.output(LED, OFF)
```

위의 코드에서는 if 문을 사용하였는데, NOT 연산자를 사용하여 다음과 같이 버튼의 값을 LED 로 바로 전달합니다.

```
import RPi.GPIO as GPIO

GPIO.setmode(GPIO.BCM)

LED = 22
BUTTON = 4
BUTTON_PRESSED = 0

GPIO.setup(LED, GPIO.OUT)
GPIO.setup(BUTTON, GPIO.IN)

while 1:
   button_value = GPIO.input(BUTTON)
   GPIO.output(LED, not button_value)
```

위의 코드에서 보면 버튼 값을 읽어 button_value 변수에 저장합니다. 아시다시피 버튼은 눌리면 0 이고 눌리지 않으면 1 입니다. 그래서 button_value 에 NOT 연산자를 취하면 버튼이 눌렸을 때 1 이 되고, 눌리지 않았을 때 0 이 됩니다. 이 값을 그대로 LED 전달하면 LED 는 버튼이 눌렸을 때 켜지고, 버튼이 눌리지 않았을 때 꺼집니다.

< 예제 코드 : **NOT 연산자를 이용하여 버튼이 눌렸을 때 LED 켜기** >

< 코드 위치 : Codingkit / **led_but_not.py** >

코딩 키트의 버튼 2개와 LED 4개를 이용한 예제입니다. 여기서 버튼 2개가 어떻게 눌리느냐에 따라서 켜지는 LED 수를 다르게 하는 예제를 해 보겠습니다. 다음은 버튼 2개가 눌리는 경우와 이에 따라 LED 가 켜지는 경우를 나타내는 표입니다.

BUTTON 1	BUTTON 0	LED 3	LED 2	LED 1	LED 0
안 눌림	안 눌림	OFF	OFF	OFF	ON
안 눌림	눌림	OFF	OFF	ON	ON
눌림	안 눌림	OFF	ON	ON	ON
눌림	눌림	ON	ON	ON	ON

표의 의미는 버튼이 모두 안 눌리면 LED 0 만 켜집니다. 모두 눌리면 LED 가 모두 켜집니다. 나머지도 이와 비슷한 의미입니다. 위의 표와 같이 동작하도록 코딩해 보겠습니다. 조금 어려운 것 같지만 "==", "!=", "and" 를 잘 이용하면 그렇게 어렵지 않습니다. 코드는 다음과 같습니다.

```
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)

LED_0 = 22
LED_1 = 23
LED_2 = 24
LED_3 = 25

BUTTON_0 = 4
BUTTON_1 = 5

BUTTON_PRESS = 0
ON = 1
OFF = 0

GPIO.setup(LED_0, GPIO.OUT)
GPIO.setup(LED_1, GPIO.OUT)
GPIO.setup(LED_2, GPIO.OUT)
GPIO.setup(LED_3, GPIO.OUT)

GPIO.setup(BUTTON_0, GPIO.IN)
GPIO.setup(BUTTON_1, GPIO.IN)

while 1 :
    but0_value = GPIO.input(BUTTON_0)
    but1_value = GPIO.input(BUTTON_1)
```

```
if ((but1_value != BUTTON_PRESS) and (but0_value != BUTTON_PRESS)) :
    GPIO.output(LED_0, ON)
    GPIO.output(LED_1, OFF)
    GPIO.output(LED_2, OFF)
    GPIO.output(LED_3, OFF)
elif ((but1_value != BUTTON_PRESS) and (but0_value == BUTTON_PRESS)) :
    GPIO.output(LED_0, ON)
    GPIO.output(LED_1, ON)
    GPIO.output(LED_2, OFF)
    GPIO.output(LED_3, OFF)
elif ((but1_value == BUTTON_PRESS) and (but0_value != BUTTON_PRESS)) :
    GPIO.output(LED_0, ON)
    GPIO.output(LED_1, ON)
    GPIO.output(LED_2, ON)
    GPIO.output(LED_3, OFF)
else :
    GPIO.output(LED_0, ON)
    GPIO.output(LED_1, ON)
    GPIO.output(LED_2, ON)
    GPIO.output(LED_3, ON)
```

보면 if 문 안에서 "!=" 는 눌리지 않았으면 이고, "==" 눌렸으면 입니다. 이 부분만 이해하면 전체적으로 어려운 부분은 없습니다. 코드가 좀 길지요. 하지만 대부분 반복되는 부분입니다.

< 예제 코드 : **버튼 2 개의 눌림에 따른 LED 4 개 켜기** >

< 코드 위치 : Codingkit / **led4_on_off_but2.py >**

< 연습 문제 : **버튼 2 개의 눌림에 따른 LED 4 개 켜기** >

버튼 2 개가 다음과 같이 눌렸을 때 LED 가 다음과 같이 켜지는 예제를 해 봅니다.

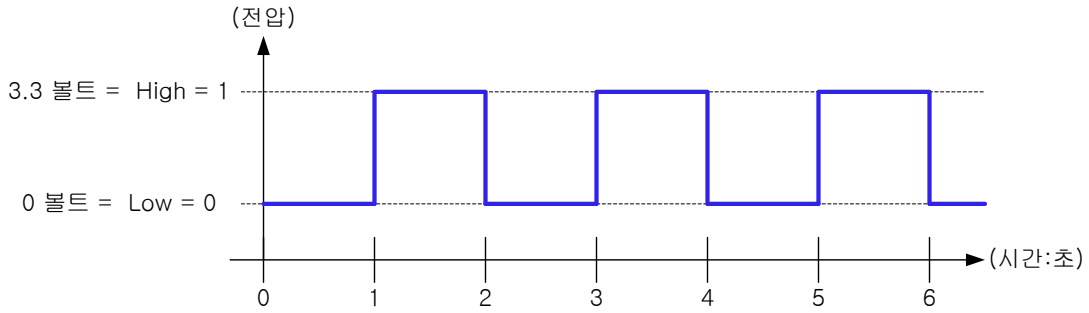
BUTTON 1	BUTTON 0	LED 3	LED 2	LED 1	LED 0
안 눌림	안 눌림	OFF	OFF	OFF	OFF
안 눌림	눌림	OFF	OFF	ON	ON
눌림	안 눌림	ON	ON	OFF	OFF
눌림	눌림	ON	ON	ON	ON

버튼 0 이 눌리면 LED 0 과 LED 1 번만 켜 집니다. 버튼 1 이 눌리면 LED 2 와 LED 3 번이 켜집니다. 한 번 잘 생각해 보시고, 나중에 제가 코딩한 것과도 비교해 보세요.

< 코드 위치 : Codingkit / **led4_on_left_right.py** >

[PWM(Pulse Width Modulation)]

코딩 키트의 아두이노 편을 공부하실 때 전기 신호를 어떻게 표시하는 가를 배웠습니다. 다음과 같은 파형으로 표시하였습니다.



그리고 이 파형이 주기적으로 반복되는 신호를 PWM 이라고 배웠습니다. 그리고 여기서 듀티비를 조정하여 어떤 의미 있는 값을 만들어 냈습니다. 이제부터 라즈베리파이에서는 어떻게 PWM 신호를 활용하는지 배워 보겠습니다.

버튼 0 번을 누르고 있으면 1초, 버튼 1 번을 누르고 있으면 0.1초, 버튼 2 번을 누르고 있으면 0.01 초, 버튼 3 번을 누르고 있으면 0.001 초 간격으로 LED 가 깜박이는 코딩을 해 보겠습니다.

```
import RPi.GPIO as GPIO
import time
GPIO.setmode(GPIO.BCM)

LED = 22

BUTTON_0 = 4
BUTTON_1 = 5
BUTTON_2 = 6
BUTTON_3 = 7

BUTTON_PRESSED = 0
ON = 1
OFF = 0

GPIO.setup(LED, GPIO.OUT)

GPIO.setup(BUTTON_0, GPIO.IN)
GPIO.setup(BUTTON_1, GPIO.IN)
GPIO.setup(BUTTON_2, GPIO.IN)
GPIO.setup(BUTTON_3, GPIO.IN)

while 1 :
    but0_value = GPIO.input(BUTTON_0)
    but1_value = GPIO.input(BUTTON_1)
    but2_value = GPIO.input(BUTTON_2)
```

```
but3_value = GPIO.input(BUTTON_3)

if (but0_value == BUTTON_PRESSED) :
   GPIO.output(LED, ON)
   time.sleep(1)
   GPIO.output(LED, OFF)
   time.sleep(1)
elif (but1_value == BUTTON_PRESSED) :
   GPIO.output(LED, ON)
   time.sleep(0.1)
   GPIO.output(LED, OFF)
   time.sleep(0.1)
elif (but2_value == BUTTON_PRESSED) :
   GPIO.output(LED, ON)
   time.sleep(0.01)
   GPIO.output(LED, OFF)
   time.sleep(0.01)
elif (but3_value == BUTTON_PRESSED) :
   GPIO.output(LED, ON)
   time.sleep(0.001)
   GPIO.output(LED, OFF)
   time.sleep(0.001)
else :
   GPIO.output(LED, OFF)
```

각 버튼이 눌렸을 때마다 time.sleep() 함수를 이용하여 시간 간격을 주면 쉽게 구현할 수 있는 코드입니다. 1 초와 0.1 초는 LED 가 깜박이는 것을 볼 수 있고, 0.01 초와 0.001 초는 LED 가 켜 있는 것처럼 보입니다. 그래도 자세히 보면 0.01 초는 LED 가 떨리는 것이 보일 것입니다.

< 예제 코드 : **LED 를 주기적으로 매우 빠르게 켜기 (신호 파형 이해)** >

< 코드 위치 : Codingkit / **led_wave_but4 >**

이번에는 for 문을 이용하여 PWM 신호를 만들어 LED 의 밝기를 조절해 보겠습니다. 다음 코드는 LED 의 밝기를 PWM 신호를 이용하여 서서히 단계적으로 밝아지게 하는 코드입니다.

```
import RPi.GPIO as GPIO
import time
GPIO.setmode(GPIO.BCM)

LED = 22

ON = 1
OFF = 0

GPIO.setup(LED, GPIO.OUT)

while 1 :
   # LED Off
```

```
GPIO.output(LED, OFF)
time.sleep(1)

# Duty Ratio : 20%
for i in range(0, 100) :
    GPIO.output(LED, ON)
    time.sleep(0.002)
    GPIO.output(LED, OFF)
    time.sleep(0.008)

# Duty Ratio : 40%
for i in range(0, 100) :
    GPIO.output(LED, ON)
    time.sleep(0.004)
    GPIO.output(LED, OFF)
    time.sleep(0.006)

# Duty Ratio : 60%
for i in range(0, 100) :
    GPIO.output(LED, ON)
    time.sleep(0.006)
    GPIO.output(LED, OFF)
    time.sleep(0.004)

# Duty Ratio : 80%
for i in range(0, 100) :
    GPIO.output(LED, ON)
    time.sleep(0.008)
    GPIO.output(LED, OFF)
    time.sleep(0.002)

# Duty Ratio : 100%
GPIO.output(LED, ON)
time.sleep(1)
```

< 예제 코드 : **for 문을 이용한 PWM 신호로 LED 켜기** >

< 코드 위치 : Codingkit / **led_pwm_for.py** >

코드를 보면 중간에 for 로 시작하는 구문이 있습니다. 이 for 문을 알기 위해서는 C 언어의 배열과 같은 리스트라는 것과 전에 for 문 안에 있는 range() 함수를 먼저 배워야 합니다. 그 이후에 나머지 코드들에 관해서도 설명하겠습니다.

< 문법 설명 : **List** >

C 언어에서의 배열을 파이썬에서는 리스트(List)라고 합니다. 파이썬에서는 리스트가 매우 유용하게 사용이 됩니다. 리스트는 다음과 같이 정의 합니다.

```
리스트_이름 = [원소1, 원소2, ..., 원소N]
```

위와 같이 리스트 정의는 대괄호 안에 원소들을 콤마로 분리하여 코딩해 주면 됩니다. 리스트 예제로 LED 4 개와 버튼 4 개를 리스트에 저장하고 각각의 버튼이 눌렸을 때 같은 번호의 LED 가 켜지는 코드를 작성해 보겠습니다. 다음과 같이 LED_0 에서 LED_4 번의 핀 번호를 정의하고 이를 리스트에 저장해 보겠습니다.

```
LED_0 = 22
LED_1 = 23
LED_2 = 24
LED_3 = 25

List_LED = [LED_0, LED_1, LED_2, LED_3]
```

이렇게 저장된 LED 들을 다음과 같이 OUT 으로 설정합니다.

```
GPIO.setup(List_LED[0], GPIO.OUT)
GPIO.setup(List_LED[1], GPIO.OUT)
GPIO.setup(List_LED[2], GPIO.OUT)
GPIO.setup(List_LED[3], GPIO.OUT)
```

여기서 List_LED[0] 은 List_LED 리스트의 첫번째 원소를 가리킵니다. List_LED[1] 은 두번째 원소입니다. 이렇게 리스트는 첫번째 원소부터 0 번의 번호를 붙여 나갑니다.

버튼도 같은 방법으로 리스트에 저장하고 IN 으로 설정합니다.

```
BUTTON_0 = 4
BUTTON_1 = 5
BUTTON_2 = 6
BUTTON_3 = 16

List_But = [BUTTON_0, BUTTON_1, BUTTON_2, BUTTON_3]

GPIO.setup(List_But[0], GPIO.IN)
GPIO.setup(List_But[1], GPIO.IN)
GPIO.setup(List_But[2], GPIO.IN)
GPIO.setup(List_But[3], GPIO.IN)
```

이렇게 설정된 LED 와 버튼은 다음과 같이 연결되어 버튼이 눌리면 같은 번호의 LED 가 켜집니다. 이 코드에서는 not 연산자를 사용하여 버튼이 눌린 경우에 1 값이 되도록 하였습니다.

```
while 1 :
    GPIO.output(List_LED[0], not GPIO.input(List_But[0]))
    GPIO.output(List_LED[1], not GPIO.input(List_But[1]))
    GPIO.output(List_LED[2], not GPIO.input(List_But[2]))
    GPIO.output(List_LED[3], not GPIO.input(List_But[3]))
```

전체 코드는 다음과 같습니다.

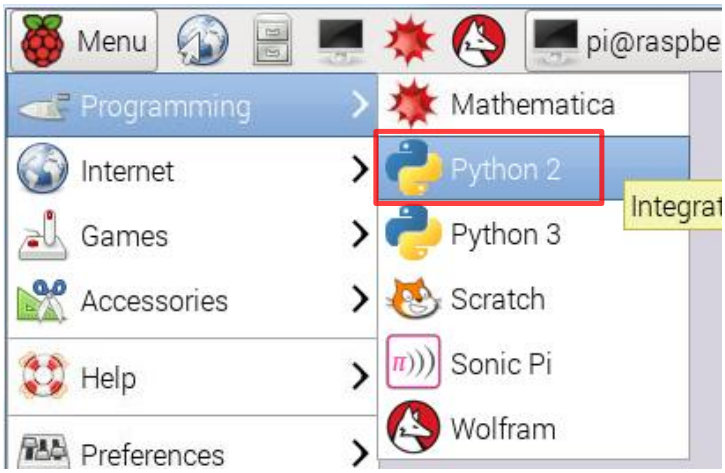
```
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)

LED_0 = 22
LED_1 = 23
LED_2 = 24
LED_3 = 25

BUTTON_0 = 4
BUTTON_1 = 5
BUTTON_2 = 6
BUTTON_3 = 7

List_LED = [LED_0, LED_1, LED_2, LED_3]
List_But = [BUTTON_0, BUTTON_1, BUTTON_2, BUTTON_3]

GPIO.setup(List_LED[0], GPIO.OUT)
GPIO.setup(List_LED[1], GPIO.OUT)
GPIO.setup(List_LED[2], GPIO.OUT)
GPIO.setup(List_LED[3], GPIO.OUT)

GPIO.setup(List_But[0], GPIO.IN)
GPIO.setup(List_But[1], GPIO.IN)
GPIO.setup(List_But[2], GPIO.IN)
GPIO.setup(List_But[3], GPIO.IN)

while 1 :
    GPIO.output(List_LED[0], not GPIO.input(List_But[0]))
    GPIO.output(List_LED[1], not GPIO.input(List_But[1]))
    GPIO.output(List_LED[2], not GPIO.input(List_But[2]))
    GPIO.output(List_LED[3], not GPIO.input(List_But[3]))
```

이제 코드를 코딩 키트에서 실행시켜 보겠습니다.

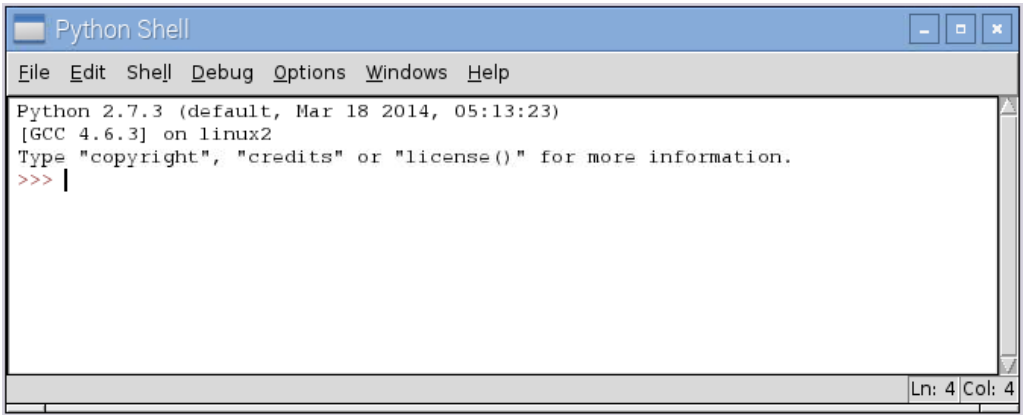
리스트에서 유용하게 사용되는 함수 몇 가지를 알아 보겠습니다.

- ✓ len(리스트_이름) : 리스트 원소의 개수를 반환 합니다.
- ✓ 리스트_이름.remove(원소_이름) : 인자의 원소를 리스트에서 제거합니다.
- ✓ 리스트_이름.append(원소_이름) : 인자의 원소를 리스트에 맨 마자막 원소로 추가합니다.
- ✓ 리스트_이름.insert(원소_번호, 원소_이름) : 원소_번호의 위치에 원소_이름의 원소를 추가합니다.

다음과 같이 Python 2 프로그램에 코딩해서 위의 함수들을 사용해 보겠습니다. 그럼 라즈베리파이 화면의 좌측상단의 "Menu → Programming → Python 2" 를 클릭하여 Python 2 프로그램을 실행시킵니다.



실행 시키면 다음과 같이 파이썬 쉘(Shell) 창이 나오는데, 여기에 리스트 관련 함수들을 사용해 보겠습니다.



위의 쉘 창에 다음과 같이 직접 코딩하고 엔터키를 누르면 각각의 코드 실행 결과가 바로 바로 나타납니다.

```
>>> List_LED = [22, 23, 24, 25]
>>> len(List_LED)
4
>>> List_LED.remove(23)
>>> List_LED
[22, 24, 25]
>>> List_LED.append(23)
>>> List_LED
[22, 24, 25, 23]
>>> List_LED.insert(0, 26)
>>> List_LED
[26, 22, 24, 25, 23]
```

위의 List_LED 의 각 원소들은 코딩 키트에 있는 LED 의 라즈베리파이 핀 번호입니다. len(List_LED) 하면 List_LED 리스트이 원소 개수가 출력됩니다. List_LED 라고 리스트 이름만 타이핑하면 리스트의 모든 원소들을 보여줍니다. 나머지 코드들도 의미를 잘 생각해 보세요.

< 예제 코드 : **리스트를 이용한 버튼과 LED 매핑** >

< 코드 위치 : Codingkit / **led4_but4_list.py** >

< 문법 설명 : **range() 함수** >

range() 함수를 다음과 같이 쓰면 어떤 범위 안의 모든 정수 값을 리스트 형태로 반환해 주는 함수입니다. 다음과 같은 식으로 쓸 수 있습니다.

range(최소값, 최대값)

이 값의 반환값은 [최소값, 최소값+1, 최소값+2, ..., 최대값-1] 값이 출력됩니다. 여기서 주의깊게 보셔야 할 것은 최소값에서 시작해서 최대값보다 1 작은 값까지 리스트를 만듭니다. Python 2 프로그램에 다음과 같이 입력해 보면 쉽게 알 수 있습니다.

```
>>> range(0, 10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

range 함수는 다음과 같이 최대값만을 설정할 수도 있습니다. 그러면 어떻게 될까요? 직접 Python 2 프로그램에 다음과 같이 입력해 보십시요.

```
>>> range(10)
```

해 보시면 위의 range(0, 10) 과 같은 결과를 보인다는 것을 알 수 있습니다. range() 함수에 하나의 인자만 쓰면 range() 함수는 0 부터 그 값까지를 리스트 형태로 반환합니다.

range() 함수에서는 다음과 같이 정수값이 증가하는 간격을 정해 줄 수도 있습니다.

range(최소값, 최대값, 증가값)

그래서 다음과 같이 range(0, 100, 10) 을 입력하면 0 부터 10 씩 증가한 값이 출력됩니다.

```
>>> range(0, 100, 10)
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
```

다음과 같이 증가값으로 마이너스 값을 줄 수도 있습니다.

```
>>> range(100, 0, -10)
[100, 90, 80, 70, 60, 50, 40, 30, 20, 10]
```

다음과 같이 최대값이 최소값보다 큰 값인데, 마이너스 증가값을 주면 아무런 항목도 없는 리스트를 출력합니다.

```
>>> range(0, 100, -10)
[]
```

이렇게 range() 함수에 증가값 인자를 이용하면 더 다양한 정수 리스트를 만들 수 있습니다.

GPIO.setup(), GPIO.output(), time.sleep() 같은 함수는 앞에 GPIO 혹은 time 이라는 라이브러리 이름을 붙여서 사용하였습니다. 그런데 range() 함수는 아무런 라이브러리 이름을 붙이지 않았습니다. 이것은 처음부터 파이썬 프로그램에 포함되어 있는 함수인 것입니다. 이런 함수를 내장 함수라고 합니다. 파이썬에는 내장 함수가 많이 있습니다. 이 내장 함수들은 파이썬 코딩을 할 때 가장 많이 쓰이는 함수들을 미리 만들어 둔 것입니다.

< 문법 설명 : **for 문** >

앞에서 설명한 range() 함수와 리스트는 for 문에서 매우 유용하게 사용됩니다. 그럼 이제 for 문에 대해서 알아보도록하고 for 문과 range() 함수, 리스트를 이용한 예제를 해 보도록 하겠습니다. 파이썬에서 for 문은 정해진 범위의 리스트나 어떤 나열된 것을 처리하기 위해 사용합니다.

파이썬의 for문의 기본 구조는 다음과 같습니다.

```
for 변수 in 리스트 :
    문장1
  문장2
```

루프가 순환하면서 리스트의 각 항목들이 순차적으로 변수에 대입됩니다. 모든 리스트의 원소를 나열하면 루프는 반복을 마칩니다.

C 언어에서 "for (i = 0; i < 100; i++)" 으로 주어진 반복문이라면 파이썬에서는 "for i in range(0, 100)" 으로 쓸 수 있습니다. 여기서 range() 함수는 인자로 주어진 범위 안의 순차적인 숫자들을 리스트로 반환합니다. range(0, 100)은 0 부터 99 까지 수를 가지는 리스트를 되돌려 줍니다. 그리고 리스트의 원소의 개수가 0 부터 99 까지 100 개이므로 이 for 문은 100 번을 반복합니다. 그래서 C 언어의 for 문과 같게 동작을 합니다. for 문에서의 인덱스를 range() 함수로 생성했을 때는 C 언어의 for 문과 같습니다. 하지만 for 문의 인덱스를 다른 특별한 리스트를 사용한다면 매우 다양한 형태의 for 문 코딩을 할 수 있습니다.

앞에서 작성한 다음과 같은 for 문을 분석해 보겠습니다.

```
# Duty Ratio : 20%
  for i in range(0, 100) :
    GPIO.output(LED, ON)
    time.sleep(0.002)
    GPIO.output(LED, OFF)
    time.sleep(0.008)
```

이것은 붉은색 부분의 코드를 100 번 반복하라는 것입니다. range() 함수에 의해서 인덱스 i 는 0 ~ 99 까지 변합니다. 붉은색 코드는 0.002 초 동안 LED 를 켜고, 0.008 초 동안 LED 를 끄는 코드입니다. 이것이 100 번을 반복하는 것입니다. 그러면 전체에서 20% 만 LED 를 켜는 것이므로 LED 가 조금 어둡겠지요. 이 for 문 이외에도 다음과 같이 40%, 60%, 80% 만큼의 시간 동안 LED 를 켜는 for 문이 있습니다.

```
# Duty Ratio : 40%
for i in range(0, 100) :
  GPIO.output(LED, ON)
  time.sleep(0.004)
  GPIO.output(LED, OFF)
  time.sleep(0.006)

# Duty Ratio : 60%
for i in range(0, 100) :
  GPIO.output(LED, ON)
  time.sleep(0.006)
  GPIO.output(LED, OFF)
  time.sleep(0.004)

# Duty Ratio : 80%
for i in range(0, 100) :
  GPIO.output(LED, ON)
  time.sleep(0.008)
  GPIO.output(LED, OFF)
  time.sleep(0.002)
```

각각의 for 문은 모두 같고 time.sleep() 함수의 시간 값만 다릅니다. 이렇게 하여 LED 가 단계적으로 밝아지는 코드가 완성되었습니다.

다음은 for 문을 2 중으로 중첩으로 사용하여 LED 의 밝기를 서서히 밝게 하는 코드입니다. 이 코드는 LED 의 밝기가 10 단계로 되어 있습니다. 위의 코드 (led_pwm_for.py) 에서 while 문 부분만 대체한 코드는 다음과 같습니다.

```
import RPi.GPIO as GPIO
import time
GPIO.setmode(GPIO.BCM)

LED = 22
```

```
ON = 1
OFF = 0

GPIO.setup(LED, GPIO.OUT)

while 1 :
   for i in range(0, 10) :
      for j in range(0, 100) :
         GPIO.output(LED, ON)
         time.sleep(i*0.001)
         GPIO.output(LED, OFF)
         time.sleep((10-i)*0.001)
      print "Index Number : ", i
   GPIO.output(LED, OFF)
   time.sleep(1)
```

첫번째 for 문이 0 부터 9 까지 i 에 차례로 대입되면서 반복을 하는 동안 매번 두번째 for 문이 0 에서 99 까지의 수가 j 에 대입되면서 두번째 for 문 안에 있는 문장들이 수행됩니다. i 가 0 일때, j 는 0 부터 100 까지 변하면서 다음행의 한단계 들여써진 붉은색 문장들이 반복됩니다. 여기서 i 가 0 이면 LED 는 켜지지 않고 10*0.001 초 동안 꺼져 있습니다. i 가 1 이면 1*0.001 초 동안 켜지고 (10-1)*0.001 초 동안 꺼집니다. 이렇게 i 가 0 에서 9 까지 변하는 동안 LED 가 켜지는 시간은 길어지고 꺼지는 시간은 짧아집니다. 그래서 LED 는 서서히 밝아집니다. 그리고 중첩된 for 문 중 두번째 for 문은 j 인덱스가 0 에서 99 까지 변하면서 100 번을 반복합니다. 그러면 두번째 for 문에 머물러 있는 시간은 얼마나 될까요? 두번째 for 문의 sleep() 함수에서 시간을 먼저 계산해 보면 다음과 같습니다.

$$i*0.001 + (10-i)*0.001 = (i + 10 - i)*0.001 = 10 * 0.001 = 0.01$$

위의 계산에 따르면 0.01 초가 됩니다. 이것이 100 번 반복이 되면 1 초가 되겠지요. 그래서 두번째 for 문에는 1 초 동안 머물게 됩니다. 첫번째 for 문이 LED 의 밝기를 10 단계로 나눈다면 각 단계는 두번째 for 문에 의해서 1 초 동안 머물게 됩니다. 즉, 해당 밝기로 1 초 동안을 유지하는 것입니다.

< 문법 설명 : **print() 함수** >

이 코드를 실행시키면 터미널에서 다음과 같은 메시지가 나옵니다.

이것은 다음과 같은 print() 함수 코드에 의해서 출력된 것입니다.

```
while 1 :
    for i in range(0, 10) :
        for j in range(0, 100) :
            GPIO.output(LED, ON)
            time.sleep(i*0.001)
            GPIO.output(LED, OFF)
            time.sleep((10-i)*0.001)
        print (i)
    GPIO.output(LED, OFF)
    time.sleep(1)
```

앞으로 print() 함수는 자주 보게 되실 것입니다. print() 함수는 기본적으로 큰 따옴표(" ") 안의 문자는 그대로 처리해 주고 변수는 변수의 값을 출력해 준다고 생각하시면 됩니다. 위의 코드에서는 print() 함수의 인자가 변수 i 이기 때문에 for 문 안에서 i 의 값을 0 ~ 9 까지 숫자로 출력해 주는 것입니다. 다음과 같이 코딩 하시면 어떻게 출력이 될까요?

```
while 1 :
    for i in range(0, 10) :
        for j in range(0, 100) :
            GPIO.output(LED, ON)
            time.sleep(i*0.001)
            GPIO.output(LED, OFF)
            time.sleep((10-i)*0.001)
        print "Index Number : ", i
    GPIO.output(LED, OFF)
    time.sleep(1)
```

다음과 같이 출력됩니다. 큰 따옴표 안의 문자는 그대로 출력해 줍니다.

```
Index Number :  0
Index Number :  1
Index Number :  2
Index Number :  3
Index Number :  4
Index Number :  5
Index Number :  6
Index Number :  7
Index Number :  8
Index Number :  9
Index Number :  0
```

print 문 다음에 괄호를 해 주셔도 되는데, 그러면 다음과 같이 전체를 괄호로 하면 괄호가 같이 출력이 됩니다.

```
print ("Index Number : ", i)
```

```
('Index Number : ', 0)
('Index Number : ', 1)
('Index Number : ', 2)
('Index Number : ', 3)
('Index Number : ', 4)
('Index Number : ', 5)
```

다음과 같이 각각의 인자만 괄호를 하면 괄호는 출력되지 않고 괄호를 전혀 쓰지 않은 것과 같은 결과를 보입니다.

```
print ("Index Number : "), (i)
```

이렇게 print 문만 조금씩 바꾸어서 여러가지 시도를 해 보십시요. 중요한 것은 각 단위는 콤마로 구분이 됩니다. 즉, "Index Number : " 와 i 는 콤마로 구분합니다.

위해서 했던 4 개의 버튼이 눌리면 같은 번호의 LED 가 켜지는 리스트를 이용한 코드를 for 문으로 바꾸어 보겠습니다. 매우 드라마틱하게 코드가 줄어드는 것을 보실 수 있을 것입니다.

```
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)

LED_0 = 22
LED_1 = 23
LED_2 = 24
LED_3 = 25

BUTTON_0 = 4
BUTTON_1 = 5
BUTTON_2 = 6
BUTTON_3 = 16

List_LED = [LED_0, LED_1, LED_2, LED_3]
List_But = [BUTTON_0, BUTTON_1, BUTTON_2, BUTTON_3]
```

```
for LED in List_LED :
   GPIO.setup(LED, GPIO.OUT)

for But in List_But :
   GPIO.setup(But, GPIO.IN)

while 1 :
   for i in range(0, 4) :
      GPIO.output(List_LED[i], not GPIO.input(List_But[i]))
```

위의 코드는 for 문과 리스트의 예제로는 매우 좋은 코드입니다. 특히 리스트는 파이썬에서 매우 많이 다루어지는 유용한 항목입니다. 잘 보고 익혀 두시면 코딩하시는데 많은 도움이 될 것입니다. 위의 코드 전체에는 3 개의 for 문이 있습니다. 2 개의 for 문은 미리 정의해 둔 List_LED 와 List_But 리스트를 이용하였고 마지막 while 문 안의 for 문 안에 있는 리스트는 range() 함수를 활용하였습니다. 왜 그렇게 하였는지도 for 문을 사용하지 않은 코드와 비교하면서 잘 봐 두도록 하면 좋을 것입니다.

< 예제 코드 : **리스트와 for 문을 이용한 버튼과 LED 매핑** >

< 코드 위치 : Codingkit / **led4_but4_list_for.py** >

아두이노의 analogWrite() 함수와 같이 라즈베리파이에서도 PWM 신호를 만들어주는 GPIO.PWM() 이라는 함수가 있습니다. 이 함수는 GPIO 라이브러리 안에 서브 함수로 있어서 GPIO.PWM() 으로 씁니다. 다음과 같이 GPIO.PWM() 함수를 생성하여 변수에 저장합니다.

```
ck_pwm = GPIO.PWM(핀_번호, 주파수)
```

이제 이 변수를 이용하여 첫번째 인자로 주어진 핀_번호에 두번째 인자로 주어진 주파수의 신호가 출력되도록 하겠습니다. 주파수의 단위는 헤르츠(Hertz) 입니다. 만약 1000 이라고 입력합니다. 1000 Hz 인 것입니다.

```
ck_pwm.start(듀티비)
```

start() 함수에 듀티비 인자를 주어 해당 듀티비로 PWM 신호의 출력이 시작됩니다. 여기서 듀티비의 범위는 0.0 에서 100.0 까지 입니다. 소수점 첫째 자리까지 자세히 설정할 수 있습니다. 다음과 같은 함수로 주파수와 듀티비를 바꿀 수 있습니다.

```
ck_pwm.ChangeFrequency(주파수)

ck_pwm.ChangeDutyCycle(듀티비)
```

더 이상 PWM 신호가 출력되는 것을 원치 않는다면 다음 함수로 PWM 신호의 출력을 끝낼 수 있습니다.

```
ck_pwm.stop()
```

이제 이러한 코드들을 이용하여 LED 에 PWM 신호를 인가하여 밝기를 바꾸는 코딩을 해 보겠습니다.

```
import RPi.GPIO as GPIO
import time
GPIO.setmode(GPIO.BCM)

LED = 22

ON = 1
OFF = 0

GPIO.setup(LED, GPIO.OUT)

ck_pwm = GPIO.PWM(LED, 1000)

while 1 :
   ck_pwm.start(0)
   for i in range(0, 10) :
      ck_pwm.ChangeDutyCycle(i*10)
      time.sleep(1)
   ck_pwm.stop()
   GPIO.output(LED, OFF)
   time.sleep(1)
```

먼저 주파수 1000 Hz 로 LED 에 출력될 신호를 "ck_pwm = GPIO.PWM(LED, 1000)" 코드로 설정하였습니다. while 함수에서 start() 함수로 신호 출력을 시작하였고 for 문을 이용하여 듀티비를 10 단계로 바꾸었습니다. 이를 위해서 "i*10" 을 해 주었습니다. 이 밝기를 1 초간 유지하기 위해서 time.sleep(1) 코드가 있습니다. 마지막에 stop() 함수를 이용하여 PWM 신호의 출력을 멈추어 주고 LED 를 끄고 1 초간 기다려 줍니다.

이제 코딩 키트로 확인해 보세요.

< 예제 코드 : **GPIO.PWM() 함수를 이용하여 LED 밝기 조절하기** >

< 코드 위치 : Codingkit / **led_pwm_func.py** >

< 연습 문제 : **GPIO.PWM() 함수를 이용하여 LED 밝기 더 세밀하게 조절하기** >

ChangeDutyCycle() 함수에서 듀티비는 0.0 에서 100.0 까지 설정할 수 있습니다. 그러면 LED 의 밝기 조절은 좀 더 세밀하게 조절할 수 있습니다. 그래서 이번 연습문제에서는 for 문의 인덱스를 0 에서 100 까지 바꾸어 LED 의 밝기를 100 단계로 해 보겠습니다. 이 인덱스를 ChangeDutyCycle() 함수의 인자로 사용할 수 있습니다. 그러면 다음 코드와 같은 역할을 할 것입니다.

```
p.ChangeDutyCycle(0)
p.ChangeDutyCycle(1)
.
.
.
p.ChangeDutyCycle(99)
p.ChangeDutyCycle(100)
```

이렇게 하면 밝기는 더 세밀하게 조절이 될 것입니다. 전체 코드는 여러분이 직접 작성해 보십시요. 여기서 sleep() 함수의 인자는 0.1 이 적당할 것 같습니다. 그래야 전체 밝기가 변하는데 10 초의 시간이 걸립니다. 1 초로 해주면 너무 길어집니다.

이 코드를 코딩 키트로 실행 시켜 보십시요. 이전에 10 단계로 조절되었던 밝기와 100 단계로 조절되었던 밝기를 코딩 키트에서 확인해 보십시요. 분명한 차이를 느낄 수 있을 것입니다. 시간 더 되시는 분은 1000 단계로 밝기가 조절되는 코드도 해 보십시요. 코딩 키트에서 실행했을 때 차이를 느끼지는 못 하겠지만 코딩하는 재미는 느끼실 수 있을 것입니다.

< 코드 위치 : Codingkit / **led_pwm_func_100.py** >

[부저(Buzzer) 소리 내기]

계속해서 LED 와 버튼 예제만 하시느라 조금 지루하셨을 것입니다. 지금부터는 새로운 디바이스인 부저(Buzzer) 예제를 해 보겠습니다.

부저는 아두이노 실습에서도 해 보았듯이 PWM 신호로 컨트롤이 됩니다. 그래서 여러분이 방금 배우셨던 GPIO.PWM() 함수를 이용하여 컨트롤해 보겠습니다.

```
import RPi.GPIO as GPIO
import time
GPIO.setmode(GPIO.BCM)

BUZ = 15

ON = 1
OFF = 0

GPIO.setup(BUZ, GPIO.OUT)

ck_pwm = GPIO.PWM(BUZ, 1000)
ck_pwm.start(50)
time.sleep(5)
ck_pwm.stop()
```

부저는 라즈베리의 15 번 핀에 연결되어 있습니다. 이 핀에 1000 Hz 주파수의 PWM 신호를 출력하겠습니다. 그리고 부저는 계속해서 울어대면 매우 시끄러우니깐 5 초만 울리게 했습니다. 코드를 실행해 보면 5 초간 삐~~~ 소리가 납니다.

< 예제 코드 : **부저 소리 내기** >

< 코드 위치 : Codingkit / **buz_pwm.py >**

그런데 삐~~~ 소리가 너무 시끄럽죠. 그래서 버튼을 연결해 보겠습니다. 버튼이 눌릴 때만 소리를 냅니다.

```
import RPi.GPIO as GPIO
import time
GPIO.setmode(GPIO.BCM)

BUZ = 15
BUTTON = 4

BUTTON_PRESSED = 0
ON = 1
OFF = 0

GPIO.setup(BUZ, GPIO.OUT)
```

```
GPIO.setup(BUTTON, GPIO.IN)

ck_pwm = GPIO.PWM(BUZ, 1000)
ck_pwm_on = 0;

while 1 :
  if (GPIO.input(BUTTON) == BUTTON_PRESSED) :
    if (ck_pwm_on == 0) :
      ck_pwm.start(50)
      ck_pwm_on = 1
  else :
    if (ck_pwm_on == 1) :
      ck_pwm.stop()
      ck_pwm_on = 0
```

버튼이 눌린 것을 체크해서 버튼이 눌렸으면 start() 함수를 이용해서 PWM 신호를 출력합니다. 그리고 눌리지 않았으면 stop() 함수로 PWM 신호의 출력을 끝냅니다. 여기서 ck_pwm_on 변수를 하나 추가해서 현재 PWM 신호가 출력되고 있으면 이 변수에 1 을 줘서 현재 PWM 신호가 출력되고 있음을 표시합니다. 그래서 ck_pwm_on 변수가 1 인지를 체크해서 1 이 아닐때만 start() 신호를 실행합니다. 이것은 PWM 신호가 켜 있는데, 또 켜 주지는 않겠다는 것입니다. PWM 신호를 켰다면 ck_pwm_on 신호에는 1 을 씁니다. 버튼이 눌리지 않았을 때는 ck_pwm_on 신호를 체크하여 stop() 함수를 수행합니다. 이렇게 상태를 표시해 주어 같은 작업을 반복하지 않게 하는 코드는 잘 봐 두십시요. 나중에 두루두루 사용될 것입니다.

< 예제 코드 : **버튼이 눌릴 때만 부저 소리 내기** >

< 코드 위치 : Codingkit / **buz_but.py** >

이제 코딩 키트에서 실행해 봅니다. 그런데, 이상하게 소리가 고르게 나지 않고 중간에 짧게 끊기는 것 같기도 하고, 좀 이상하지요. 이것은 GPIO.PWM() 함수를 이용해서는 정확한 PWM 신호를 만들수가 없기 때문입니다. PWM 신호라는 것은 주기와 듀티비가 계속해서 같아야 합니다. 그런데, 중간에 주기와 듀티비가 달라져 버리면 정확한 신호의 전달이 되지 않습니다. 그래서 부저의 소리가 고르지 않고 이상하게 들리는 것입니다.

그래서 이것을 해결하기 위해서 라즈베리파이 1 에서는 RPIO 라는 라이브러리를 사용합니다. 하지만 이 라이브러리는 라즈베리파이 2 에서는 동작하지 않습니다. 라즈베리파이 1 과 2 는 CPU 자체가 다른 매우 다른 하드웨어 입니다. 그래서 하드웨어와 밀접한 관계가 있는 PWM 신호는 제대로 동작하지 않는 것입니다. 이 글을 쓰고 있는 시점에서 라즈베리파이 2 가 출시된 지는 얼마 되지 않았습니다. 하지만 같은 가격에 월등히 성능히 좋은 라즈베리파이 2 가 앞으로 더 많은 사람들이 사용하게 될 것이고 라즈베리파이 1 은 그 사용이 급격히 줄어들 것입니다. 그래서 코딩 키트에서는 라즈베리파이 1 과 2 모두에서 사용할 수 있도록 하드웨어로 PWM 신호를 생성할 수 있도록 하였습니다. 바로 코딩 키트에 있는 스위칭 칩에서 SPI 신호를 받아서 PWM 신호를 재생해 주는 것입니다. SPI 신호는 코딩북의 아두이노 파트에서 배워서

잘 아실 것입니다. 그럼 라즈베리파이에서는 SPI 를 어떻게 사용하는지를 배우고 이 SPI 를 이용해서 어떻게 PWM 신호를 재생하는지를 배우겠습니다.

[SPI 인터페이스로 PWM 신호 만들기]

라즈베리파이에서는 SPI 를 2 개 사용할 수 있습니다. 완벽하게 2 개의 SPI 를 사용할 수 있는 것은 아니고요. SPI_SS (슬래이브 셀렉트 : Slave Select) 신호가 2 개 있습니다. 그래서 SPI 2 개를 동시에는 사용할 수는 없지만 SPI 슬래이브 2 개를 연결하여 사용할 수는 있습니다. 각각은 시간을 나누어 사용해야 겠지요. 참고로 한가지 더 말씀드리면 라즈베리파이는 아두이노와는 달리 ADC(Analog to Digital Converter)가 내장되어 있지 않습니다. 그래서 외부의 ADC 를 통하여 센서 등의 아날로그 신호 값을 입력 받아야 합니다. 코딩 키트에도 라즈베리파이를 위한 ADC 가 장착되어 있습니다. 그런데, 이 ADC 가 SPI 로 연결되어 있습니다. 그래서 라즈베리파이에 있는 2 개의 SPI 중 하나는 이 ADC 에 연결되어 있고 또 다른 하나는 지금부터 배워 볼 PWM 신호를 재생하는데 쓰입니다.

SPI 를 사용하려면 먼저 다음과 같이 spidev 라는 라이브러리를 import 합니다.

```
import spidev
```

이 라이브러리를 이용하여 spi 변수를 하나 선언합니다.

```
ck_spi = spidev.SpiDev()
```

SPI 를 사용하기 위해서 초기화를 해 주어야 하는데, 그 때 사용하는 함수가 open() 입니다.

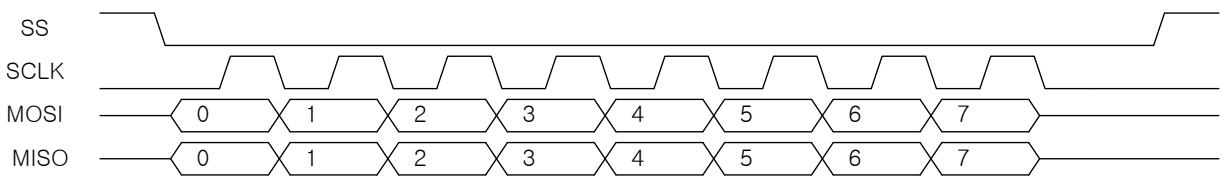
```
ck_spi.open(0, 1)
```

open() 함수의 첫번째 인자는 SPI 포트 번호인데, 라즈베리파이에서 SPI 의 신호 4 개를 전부 지원하는 포트는 1 개뿐이어서 첫번째 인자는 항상 0 을 씁니다. 두번째 인자는 SPI_SS 신호의 번호입니다. SPI_SS 신호 중 0 번은 ADC 에게 양보하고 여기서는 1 번을 사용합니다. 그래서 open(0, 1) 이 된 것입니다.

혹시 아두이노의 SPI 코드가 기억 나십니까? 기억나지 않는다면 다시 한번 더 보시면 다음과 같이 SPI_SS 신호는 따로 컨트롤해 준 것이 기억나실 것입니다.

```
digitalWrite(SPI_SS, 0);
SPI.transfer(cmd);
SPI.transfer(data);
digitalWrite(SPI_SS, 1);
```

라즈베리파이에서도 똑같습니다. SPI _SS 신호는 따로 컨트롤해 줍니다. 이것은 여러 개의 슬래이브를 사용하기 위해서 하드웨어에서 자동으로 해 주지 않고 이렇게 코딩으로 처리합니다. 이 참에 SPI 파형도 한번 더 봐 두시지요.



위의 그림에서 SS 신호를 따로 컨트롤해 주는 겁니다. 그래서 라즈베리파이 코드에서는 다음과 같이 SPI_SS 신호를 선언해 주고 그 신호에 GPIO.output() 함수를 이용하여 값을 써 줍니다.

아두이노 코드의 SPI.transfer() 라는 함수와 같은 함수가 라즈베리파이에도 있습니다. 이 함수도 SPI 슬래이브로 데이터를 전달하는 함수 입니다. 그런데 이 함수는 인자를 리스트로 받을 수 있어 여러 개의 항목을 리스트로 묶어 한꺼번에 보낼 수 있습니다. 리스트의 원소는 8 비트입니다. 그래서 다음과 같이 하면 SPI 로 커맨드와 데이터를 보낼 수 있습니다.

```
GPIO.output(SPI_SS, 0)
ck_spi.xfer([cmd, data])
GPIO.output(SPI_SS, 1)
```

xfer() 함수의 인자로 대괄호([])를 이용하여 리스트를 만들어 전달해 줍니다. 이제 전체 코드를 보면 다음과 같습니다.

```
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)
import time
import spidev

SPI_SS = 7

# SPI Command
SPI_CMD_CLEAR       = 0x00
SPI_CMD_PWM_BUZ_EN   = 0x20
SPI_CMD_PWM_BUZ_FR_H = 0x21
SPI_CMD_PWM_BUZ_FR_L = 0x22
SPI_CMD_PWM_BUZ_DUTY = 0x23

GPIO.setup(SPI_SS, GPIO.OUT)
GPIO.output(SPI_SS, 1)

ck_spi = spidev.SpiDev()
ck_spi.open(0, 1)

def ck_spiWr(cmd, data) :
    print(hex(cmd), hex(data))
    GPIO.output(SPI_SS, 0)
    ck_spi.xfer([cmd, data])
    GPIO.output(SPI_SS, 1)

ck_spiWr(SPI_CMD_CLEAR, 0)
ck_spiWr(SPI_CMD_PWM_BUZ_FR_H, 0x01)
ck_spiWr(SPI_CMD_PWM_BUZ_FR_L, 0x2c)
ck_spiWr(SPI_CMD_PWM_BUZ_DUTY, 50)
ck_spiWr(SPI_CMD_PWM_BUZ_EN, 1)
time.sleep(5)
ck_spiWr(SPI_CMD_PWM_BUZ_EN, 0)
```

코딩 키트에서는 아래와 같은 SPI 커맨드를 활용하여 부저를 동작시킬 수 있습니다. SPI 커맨드는 아두이노 파트에서 이미 설명드린 바와 같이 이 다음에 보낼 데이터의 종류를 의미합니다.

커맨드	코드	설명
SPI_CMD_PWM_BUZ_EN	0x20	부저로 PWM 신호를 출력합니다. 다른 모든 부저 관련 커맨드를 실행한 다음에 이 커맨드를 실행합니다. 데이터 값이 1 이면 활성화 됩니다.
SPI_CMD_PWM_BUZ_FR_H	0x21	부저 PWM 신호의 주파수 중 상위 8 비트를 설정합니다.
SPI_CMD_PWM_BUZ_FR_L	0x22	부저 PWM 신호의 주파수 중 하위 8 비트를 설정합니다.
SPI_CMD_PWM_BUZ_DUTY	0x23	부저 PWM 신호의 듀티비를 설정합니다. 0 ~ 100 의 범위를 갖습니다.
SPI_CMD_CLEAR	0x00	SPI 인터페이스로 연결된 디바이스들을 초기화합니다. 예를 들어 LED 같은 경우에는 0 값을 주어 LED 가 꺼지도록 합니다.

SPI_CMD_PWM_BUZ_FR_H 와 SPI_CMD_PWM_BUZ_FR_L 은 PWM 신호의 주파수를 결정합니다. PWM 신호의 주파수는 최대 300 KHz 이고 최소 1 Hz 입니다. 그런데, 그 값은 다음 공식을 따릅니다.

300,000 / 주파수 = 주파수_설정_입력값

이 공식에 의해서 도출된 주파수_설정_입력값을 SPI_CMD_PWM_BUZ_FR_H 와 SPI_CMD_PWM_BUZ_FR_L 커맨드의 데이터로 넣어 주시면 됩니다. 만약 여러분이 1000 Hz 를 만들고 싶으면 다음과 같이 계산이 됩니다.

300,000 / 1000 = 300

이 계산에 의해서 주파수_설정_입력값은 300 이 되고 이 값은 SPI_CMD_PWM_BUZ_FR_H 와 SPI_CMD_PWM_BUZ_FR_L 의 데이터가 되는데, 상위 8 비트는 300 / 256 이고 하위 8 비트는 300 을 256 으로 나눈 나머지 값입니다. 다음과 같이 계산이 됩니다.

300 / 256 = 1 (나머지 버림)

300 % 256 = 44 (% 는 모듈로 연산, 즉, 나머지 계산)

그래서 SPI_CMD_PWM_BUZ_FR_H 에는 1 을 입력하고 SPI_CMD_PWM_BUZ_FR_L 에는 44 를 입력합니다. 조금 복잡하지만 하나 하나 따져보면 그렇게 어렵지는 않습니다. 그리고 이것을 자동으로 계산하는 함수를 하나 만들어 코드에 넣는 것도 좋은 방법입니다. 이 함수는 잠시 후에 만들어 보도록 하겠습니다.

위의 공식에서 주파수_설정_값을 구한 이후에 SPI_CMD_PWM_BUZ_FR_H 와 SPI_CMD_PWM_BUZ_FR_L 에 나누어 값을 입력할 때 위와 같이 256 으로 나누고 나머지를 구하고 하는 방법도 있고 일단 주파수_설정_값 300 을 16 진수로 바꿉니다. 그러면 0x12C 가 계산됩니다. 이 값중 아래 두자리 0x2C 를 SPI_CMD_PWM_BUZ_FR_L 의 데이터로 입력하고 그 위의 한 자리 0x1 을 SPI_CMD_PWM_BUZ_FR_H 에 입력하면 됩니다. 그래서 위의 코드와 같은 다음과 같은 코드가 나온 것입니다.

```
ck_spiWr(SPI_CMD_PWM_BUZ_FR_H, 0x01)
ck_spiWr(SPI_CMD_PWM_BUZ_FR_L, 0x2c)
```

주파수 설정 방법이 조금 복잡하기는 한데, 정확한 PWM 신호를 만들기 위해서 이러한 방법을 사용하였습니다.

파이썬의 숫자 표현은 C 언어와 동일하게 2진수는 숫자 앞에 0b 를 붙이고 16 진수는 0x 를 붙입니다. 아두이노 파트의 < 문법 설명 : **2 진수, 8 진수, 10 진수, 16 진수 그리고 비트, 바이트** > 를 참고하시면 매우 자세히 나와 있습니다.

한 가지 알아두실 것은 print() 함수에서 다음과 같이 하면 16 진수나 2 진수로 출력할 수 있습니다.

```
print(hex(38))
print(bin(38))
```

이렇게 해서 위의 코드 설명은 거의 다 끝난 것 같은데, 중요한 하나가 빠졌습니다. def 구문입니다. 이것은 파이썬에서 함수를 정의하는 구문입니다.

< 예제 코드 : **SPI 인터페이스로 만들어진 PWM 신호를 이용하여 부저 소리 내기** >

< 코드 위치 : Codingkit / **buz_spi_pwm.py** >

< 문법 설명 : **함수** >

파이썬의 함수는 C 언어의 함수하고는 모양새는 많이 다르지만 실상은 똑같습니다.

```
def 함수이름 (매개변수) :
    문장 1
    문장 2
    …
    return(값)
```

파이썬의 함수는 def 이라는 키워드로 시작합니다. 그리고 함수 이름을 쓰고 괄호 안에 매개 변수를 씁니다. 만약 리턴(return) 값이 있다면 return 을 이용합니다.

이 정의에 따라 간단한 함수를 하나 만들어 보겠습니다. time.sleep 함수는 인자를 초단위로 받습니다. 그런데, 디바이스를 다루다보니 아두이노와 같이 1/1000 초(Mili Second : ms)의 시간값을 많이 사용하게 됩니다. 그래서 1/1000 초 단위로 시간을 처리하는 함수를 만들어 보겠습니다.

```
def sleep_ms(ms) :
    s = ms*0.001
    time.sleep(s)
```

이 함수를 이용하여 이전에 for 문을 이용하여 PWM 신호를 만들었던 다음 코드를 바꾸어 보겠습니다.

```
while 1 :
    # LED Off
    GPIO.output(LED, OFF)
    time.sleep(1)

    # Duty Ratio : 20%
    for i in range(0, 100) :
        GPIO.output(LED, ON)
        time.sleep(0.002)
        GPIO.output(LED, OFF)
        time.sleep(0.008)
```

위의 코드는 sleep_ms() 함수를 이용하여 다음과 같이 바꿀 수 있습니다.

```
while 1 :
    # LED Off
    GPIO.output(LED, OFF)
    time.sleep(1)

    # Duty Ratio : 20%
    for i in range(0, 100) :
        GPIO.output(LED, ON)
        sleep_ms(2)
        GPIO.output(LED, OFF)
        sleep_ms(8)
```

파이썬에서는 함수를 이렇게 활용합니다. 이 함수는 반환값이 없어 return 문은 사용하지 않았습니다.

함수는 사용하려는 곳보다 먼저 정의해 두어야 합니다. 즉, 위의 코드에서도 sleep_ms(2) 를 사용하는 while 문 보다 이전에 "def sleep_ms(ms)" 함수를 정의해야 합니다.

< 예제 코드 : **함수를 이용하여 LED 밝기 조절** >

< 코드 위치 : Codingkit / **led_pwm_for_def.py** >

위의 SPI 를 이용한 PWM 신호를 만드는 코드에서는 함수를 다음과 같이 썼습니다.

```
def ck_spiWr(cmd, data) :
    print(hex(cmd), hex(data))
    GPIO.output(SPI_SS, 0)
    ck_spi.xfer([cmd, data])
    GPIO.output(SPI_SS, 1)
```

먼저 ck_spiWr 라는 함수 이름을 쓰고 괄호 안에 cmd 와 data 매개변수를 받습니다. 이 변수들을 SPI 로 전송하기 위해 ck_spi.xfer() 함수를 이용합니다. 이 함수는 리스트를 인자로 받기 때문에 cmd와 data 를 리스트로 만들어 인자로 전달하였습니다.

이 함수는 반환값이 없어 return() 을 사용하지는 않았습니다.

함수의 사용은 다음과 같습니다.

```
반환값_저장_변수 = 함수이름(인자1, 인자2, ...)
```

C 언어와 똑같습니다. 함수이름으로 함수를 호출하고 반환값이 있으면 변수에 저장하거나 코드 내에서 바로 씁니다. 인자들을 괄호 안에 써서 전달합니다. 위의 코드는 다음과 같이 썼습니다.

```
ck_spiWr(SPI_CMD_PWM_BUZ_FR_H, 0x01)
ck_spiWr(SPI_CMD_PWM_BUZ_FR_L, 0x2c)
ck_spiWr(SPI_CMD_PWM_BUZ_DUTY, 50)
ck_spiWr(SPI_CMD_PWM_BUZ_EN, 1)
```

ck_spiWr 함수를 불러와 다양한 커맨드를 입력하고 해당 커맨드에 맞는 데이터를 인자로 전달해 주었습니다.

이제 이 코드를 코딩 키트에서 실행해 보겠습니다. SPI 를 사용하지 않고 했던 코드에서의 부저 소리가 고르지 않았던 것에 비해서 이번 코드는 매우 고르게 소리를 내는 것을 알 수 있습니다. 앞으로는 계속해서 이 SPI 를 이용하여 PWM 신호를 재생하겠습니다. 여러분이 코딩 키트가 없이 라즈베리파이 보드만으로는 PWM 신호를 제대로 재생하기는 힘듭니다. 그래서 일반적으로 라즈베리파이를 이용하여 PWM 신호를 재생할 때는 PWM 신호를 재생하는 칩을 따로 장착하거나 합니다. 여기서는 그런 칩의 역할을 코딩 키트의 스위칭 칩에서 해 주고 있는 것입니다.

< 문법 설명 : 모듈(**module**) >

파이썬에서는 함수를 모듈이라는 것으로 만들어 다른 파일의 코드에서 사용할 수 있도록 해 줍니다. C 언어에서의 라이브러리와 유사한 개념이라고 할 수 있겠습니다. 이 모듈에는 함수를 한 개 또는 여러 개를 포함할 수 있습니다. 이 모듈은 전에 보았던 import 를 구문을 이용하여 불러올 수 있습니다.

파이썬에서의 모듈의 정의는 조금 애매합니다. 모듈이라는 것은 파이썬에서 어떤 함수를 만들고 만들어진 함수를 배포(Distribute)하기 위해서 만든 단위입니다. 모듈은 모듈과 패키지(Package)로 나뉩니다. 이 말이 좀 이상하게 들리지요? 모듈을 **모듈**과 패키지로? 좀 이상하긴 하겠지만, 일단 계속 설명드리고 나중에 다시 정리 하겠습니다. 그래서 그렇게 나뉜 모듈과 패키지 중 패키지는 모듈과 패키지들이 모인 것을 패키지라고 합니다. 여기서 모듈은 어떤 다른 모듈이나 패키지를 포함할 수 없습니다. 그럼 처음 모듈과 나뉘어진 모듈을 구분하기 위해서 이렇게 나뉘어진 모듈을 "작은 모듈"이라고 하겠습니다. 그래서 어떤 함수를 배포하기 위해서 작은 모듈로 만들었습니다. 이렇게 만든 작은 모듈에는 어떤 작은 모듈이든 패키지든 포함할 수가 없습니다. 그리과 작은 모듈들이 모이면 패키지가 됩니다. 그런데 패키지는 작은 모듈뿐만 아니라 다른 패키지도 포함할 수 있습니다. 그리고 끝으로 이렇게 작은 모듈이나 패키지나 그냥 모두 통상적으로 모듈이라고 부릅니다. 이제 좀 이해가 되셨나요? 그래서 파이썬에서 일반적으로 모듈이라고 부르는 것은 함수를 배포하기 위해서 만든 작은 모듈이나, 이런 작은 모듈들이 모아서 패키지로 배포하는 것이나 다 모듈이라고 부릅니다.

이 이야기를 다음과 같이 정리할께요. 모듈이란 함수 하나나 여러 개 묶음을 다른 코드에서 불러다 쓰기 위해서 만든 것을 모듈이라고 부릅니다. 위에서 패키지 개념만 조금 이해하시고 "그냥 다 모듈이다" 라고 이해하시고 넘어 가시면 됩니다. 복잡한 이야기는 패쓰~~~ 하셔도 코딩하시는데 큰 지장은 없습니다.

그림으로 정리하면 다음과 같습니다.

그럼 모듈을 만들어 보겠습니다. 모듈은 함수를 이용하여 만듭니다. 그래서 위의 코드 중 함수를 이용하여 모듈을 만들어 보겠습니다. 모듈 코드는 다음과 같습니다.

```
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)
import spidev

SPI_SS = 7

# SPI Command
SPI_CMD_CLEAR        = 0x00
SPI_CMD_PWM_BUZ_EN   = 0x20
SPI_CMD_PWM_BUZ_FR_H = 0x21
SPI_CMD_PWM_BUZ_FR_L = 0x22
```

```
SPI_CMD_PWM_BUZ_DUTY = 0x23

GPIO.setup(SPI_SS, GPIO.OUT)
GPIO.output(SPI_SS, 1)

ck_spi = spidev.SpiDev()
ck_spi.open(0, 1)

def ck_spiWr(cmd, data) :
    print(hex(cmd), hex(data))
    GPIO.output(SPI_SS, 0)
    ck_spi.xfer([cmd, data])
    GPIO.output(SPI_SS, 1)
```

코딩이 다 되었으면 저장하기 전에 다음과 같이 터미널에서 CK_SPI 폴더를 만듭니다.

```
pi@raspberrypi ~/Codingkit $ mkdir CK_SPI
```

위의 코드를 위에서 만든 CK_SPI 폴더에 CK_SPI.py 라는 이름으로 저장합니다. 그리고 다음과 같이 setup.py 라는 파일을 만들어 같은 폴더에 저장합니다. 이 setup.py 파일은 모듈에 대한 정보를 담고 있는 파일입니다.

```
from distutils.core import setup

setup(
    name    = 'CK_SPI',
    version = '1.0.0',
    py_modules = ['CK_SPI'],
    author = 'SISODREAM - WH Lee',
    author_email = 'ck@sisodream.com',
    url = 'www.codingkit.net',
    description = 'Codingkit SPI Interface'
)
```

먼저 distutils.core 라는 모듈 중 setup 함수를 import 합니다. distutils.core 는 모듈을 배포하기 위한 유틸리티(Utility)들이 있는 모듈입니다. 이 중 setup() 함수는 모듈 설정 및 설치에 관한 함수입니다. 이 setup() 함수에는 먼저 이름과 버전 정보를 전달합니다. 다음으로 모듈의 리스트를 전달하고 그 다음은 저자, 저자 이메일, 관련 사이트 주소, 추가 정보 등을 전달해 줍니다.

이제 위에서 만든 CK_SPI 폴더로 가서 "python setup.py sdist" 라고 입력하여 모듈을 생성합니다.

```
pi@raspberrypi ~/Codingkit $ cd CK_SPI/
pi@raspberrypi ~/Codingkit/CK_SPI $ python setup.py sdist
running sdist
running check
warning: sdist: manifest template 'MANIFEST.in' does not exist (using default file list)

warning: sdist: standard file not found: should have one of README, README.txt

writing manifest file 'MANIFEST'
creating CK_SPI-1.0.0
making hard links in CK_SPI-1.0.0...
hard linking CK_SPI.py -> CK_SPI-1.0.0
hard linking setup.py -> CK_SPI-1.0.0
Creating tar archive
removing 'CK_SPI-1.0.0' (and everything under it)
```

경고(warning)이 2 개가 나오는데 무시하고 넘어가셔도 됩니다. 첫번째 경고는 'MANIFEST.in' 파일이 없다는 것인데, 이것은 현재 만드려는 모듈에 파이썬 파일(확장자가 .py 인 파일)의 목록을 적어두는 파일입니다. 이 파일은 없어도 되고, 이 모듈을 만드는 과정에서 자동적으로 "MANIFEST" 파일을 만듭니다. 두번째 경고는 이 모듈을 구체적으로 설명하는 "README" 나 "README.txt" 파일이 없다는 것입니다. 이렇게 하여 모듈이 만들어졌습니다. 이렇게 만들어진 모듈은 "sudo python setup.py install" 하여 설치합니다.

```
pi@raspberrypi ~/Codingkit/CK_SPI $ sudo python setup.py install
running install
running build
running build_py
running install_lib
running install_egg_info
Removing /usr/local/lib/python2.7/dist-packages/CK_SPI-1.0.0.egg-info
Writing /usr/local/lib/python2.7/dist-packages/CK_SPI-1.0.0.egg-info
```

이제 모듈을 만들고 설치하는 것까지 완료하였습니다. 그러면 이제 이 모듈을 사용하는 방법에 대해서 배워보겠습니다.

< 코드 위치 : Codingkit / **CK_SPI.py** >

< 코드 위치 : Codingkit / **setup.py** >

< 문법 설명 : **import** >

파이썬에서는 배포되는 모듈을 사용하기 위해서 import 문을 사용합니다. C언어의 #include 문과 유사합니다.

이제 위에서 만든 CK_SPI 모듈을 사용하여 위에서 만든 "SPI 인터페이스로 만들어진 PWM 신호를 이용하여 부저 소리 내기" 의 예제를 바꾸어 보겠습니다. 코드는 다음과 같습니다.

```
import time
import CK_SPI as spi
```

```
spi.ck_spiWr(spi.SPI_CMD_CLEAR, 0)
spi.ck_spiWr(spi.SPI_CMD_PWM_BUZ_FR_H, 0x01)
spi.ck_spiWr(spi.SPI_CMD_PWM_BUZ_FR_L, 0x2c)
spi.ck_spiWr(spi.SPI_CMD_PWM_BUZ_DUTY, 50)
spi.ck_spiWr(spi.SPI_CMD_PWM_BUZ_EN, 1)
time.sleep(5)
spi.ck_spiWr(spi.SPI_CMD_PWM_BUZ_EN, 0)
```

먼저 붉은색 코드와 같이 CK_SPI 를 import 합니다. 이름이 너무 길어 간단하게 spi 로 바꿉니다. CK_SPI 모듈에 있는 ck_spiWr() 함수를 사용하기 위해서 spi.sk_spiWr() 라고 써 줍니다. CK_SPI 에서 정의된 변수도 "spi.변수이름" 으로 사용할 수 있습니다.

모듈 및 import 에 대해서 몇 가지 더 알아 보고 가겠습니다.

파이썬 모듈이 어디에 설치되어 있는지 알고 싶으면 Python 2 프로그램 쉘에서 다음과 같이 타이핑합니다.

```
>>> import sys
>>> sys.path
['', '/home/pi', '/usr/bin', '/usr/lib/python3.2', '/usr/lib/python3.2/plat-linu
x2', '/usr/lib/python3.2/lib-dynload', '/usr/local/lib/python3.2/dist-packages',
 '/usr/lib/python3/dist-packages']
```

RPi.GPIO 모듈의 위치는 다음과 같이하면 알 수 있습니다.

```
>>> RPi.GPIO
<module 'RPi.GPIO' from '/usr/lib/python3/dist-packages/RPi/GPIO.cpython-32mu.so'>
```

위와 같이 from 다음의 path 에 위치해 있습니다.

import 는 다음과 같이 정의 됩니다.

```
import 모듈이름 as 사용이름
```

import 키워드 이후에 모듈이름을 써 줍니다. 이름이 너무 길거나 할 때는 as 를 사용하여 이름을 바꿀수 있습니다. as 이후는 생략할 수 있습니다. 위의 코드에서 as 이하를 생략하였다면 다음과 같이 씁니다.

```
import CK_SPI
CK_SPI.ck_spiWr(CK_SPI.SPI_CMD_CLEAR, 0)
```

다음과 같이 사용할 수도 있습니다.

```
from 모듈이름 import 함수_또는_변수
```

위의 코드에서 함수_또는_변수 는 모듈 안의 함수 또는 변수를 말하는 것입니다. 이렇게 쓰면 함수 또는 변수를 사용할 때 모듈 이름을 쓰지 않아도 됩니다. 다음과 같이 cs_spiWr 를 import 하면 모듈 이름을 쓰지 않고 사용할 수 있습니다.

```
import CK_SPI as spi
from CK_SPI import ck_spiWr
ck_spiWr(spi.SPI_CMD_CLEAR, 0)
```

이 때 주의하실 점은 현재 코드에서 ck_spiWr 라는 함수가 없어야 합니다. 다음과 같이 하면 모두 모듈 이름 없이 사용할 수 있습니다.

```
from CK_SPI import *
ck_spiWr(SPI_CMD_CLEAR, 0)
```

이렇게 import 는 다양한 방법으로 사용될 수 있습니다. 잘 봐 두셨다가 유용하게 사용하십시요.

한 가지 더 설명드리면 CK_SPI.py 파일에서 "import RPi.GPIO as GPIO" 했습니다. 그리고 위의 코드에서 "import CK_SPI" 를 했는데, 다음과 같이 LED 를 켜는 코드를 추가할 때 " import RPi.GPIO as GPIO" 를 해야 할까요? 말아야 할까요? 하는 것이 편리하게 사용할 수 있지만 안해도 사용할 수는 있습니다. GPIO 를 import 한 코드는 다음과 같습니다.

```
import time
import CK_SPI as spi
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)

LED = 22
GPIO.setup(LED, GPIO.OUT)
GPIO.output(LED, 1)

spi.ck_spiWr(spi.SPI_CMD_CLEAR, 0)
spi.ck_spiWr(spi.SPI_CMD_PWM_BUZ_FR_H, 0x01)
spi.ck_spiWr(spi.SPI_CMD_PWM_BUZ_FR_L, 0x2c)
spi.ck_spiWr(spi.SPI_CMD_PWM_BUZ_DUTY, 50)
spi.ck_spiWr(spi.SPI_CMD_PWM_BUZ_EN, 1)
time.sleep(5)
spi.ck_spiWr(spi.SPI_CMD_PWM_BUZ_EN, 0)

GPIO.output(LED, 0)
```

GPIO 를 import 하지 않고 사용하려면 GPIO.setup() 같이 GPIO 를 사용하는 코드를 spi.GPIO.setup() 이라고 써서 GPIO 의 위치가 spi 모듈 안에 있다는 것을 정의해 두어야 합니다.

위의 코드는 소리가 나는 동안 LED 가 켜져 있는 코드입니다.

< 예제 코드 : **import GPIO 해서 LED 켜기** >

< 코드 위치 : Codingkit / **buz_gpio_led.py** >

< 연습 문제 : **부저에서 소리가 나는 동안 LED 가 하나씩 꺼지기** >

처음 부저소리가 날때는 LED 를 모두 켭니다. 그리고 시간이 1 초 흐를 때마다 LED 를 하나씩 끕니다. 그리고 최종적으로는 LED 를 모두 끄고 부저도 끕니다.

```
import time
import CK_SPI as spi
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)

LED_0 = 22
LED_1 = 23
LED_2 = 24
LED_3 = 25

List_LED = [LED_0, LED_1, LED_2, LED_3]
```

모듈 import 하는 것과 LED 핀 정의는 위와 같습니다. LED 는 리스트로하면 코드가 더 쉽겠죠. 그리고 for 문을 꼭 이용하세요.

< 코드 위치 : Codingkit / **buz_led4_off.py** >

[가변저항을 통한 아날로그 값 입력 받기]

아두이노는 가변저항 또는 센서 출력 값 같은 아날로그 값을 입력받는 전용 핀이 있습니다. 이것을 가능하게 하는 것은 내장 ADC (Analog to Digital Converter) 가 이었서 그렇다고 배웠습니다. 하지만 라즈베리파이는 ADC가 내장되어 있지 않기 때문에 외부에 별도의 ADC 칩을 사용해야 합니다. 이 외장 ADC 와 라즈베리파이는 SPI 통신을 합니다. 앞에서 PWM 신호를 생성하는 SPI 의 SPI_SS (SPI Slave Select) 는 1 번을 사용했다면 외장 ADC 와 연결되는 SPI 의 SPI_SS 는 0 번을 사용합니다.

그럼 SPI_SS 핀 번호를 정의하고 입출력 모드를 설정한 후 1 값을 줍니다.

```
ADC_SPI_SS = 8
GPIO.setup(ADC_SPI_SS, GPIO.OUT)
GPIO.output(ADC_SPI_SS, 1)
```

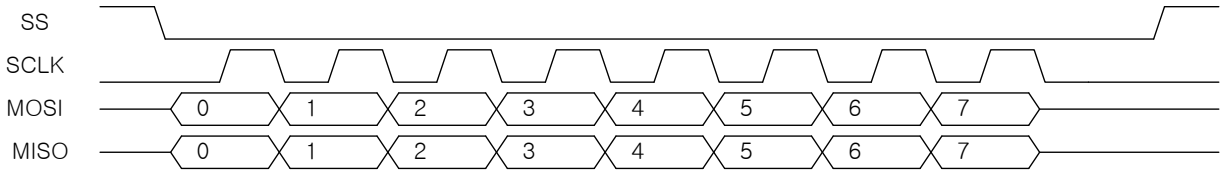
외장 ADC 는 4 개의 아날로그 값을 입력 받을 수 있습니다. 이런 것을 4 채널(Channel) 이라고 합니다. 각 채널의 아날로그 값은 디지털 값으로 바뀌는데, 이 디지털 값은 10 비트입니다. 그래서 이 아날로그 값이 디지털 값으로 변한 값의 범위는 0 ~ 1023 까지입니다. 아두이노의 ADC 값의 범위와 같습니다. 각 채널에는 다음과 같은 아날로그 신호들이 연결되어 있습니다.

채널 번호	채널 ID	연결 디바이스	ADC Mode Switch
0	0xC7	적외선 센서	X
1	0xCF	소리 센서	X
2	0xD7	온도 센서	Switch 0 Down
		가변저항 2	Switch 0 Up
3	0xDF	밝기 센서	Switch 1 Down
		가변저항 1	Switch 1 Up

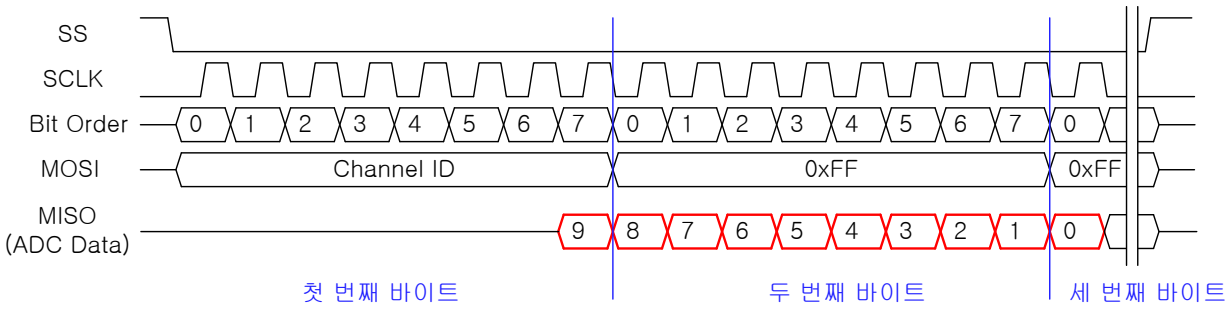
연결된 디바이스의 값을 읽어 오기 위해서는 채널 ID를 입력하고 값을 읽습니다. 그래서 다음과 같은 코드가 만들어 집니다. 이것은 가변저항 1 번의 값을 읽는 것입니다.

```
GPIO.output(ADC_SPI_SS, 0)
adcVal_list = spi.xfer([ADC_VR1, 0xFF, 0xFF])
GPIO.output(ADC_SPI_SS, 1)
adcVal = (adcVal_list[0] << 9) + (adcVal_list[1]<<1) + (adcVal_list[2] >> 7)
print "VR1 : ", adcVal, hex(adcVal)
time.sleep(0.5)
```

다음과 같은 SPI 인터페이스 파형을 만들어 주기 위해서 SPI_SS 에 0 을 써 주고 1 을 써 줍니다.



그리고 그 사이에 spi.xfer 함수로 채널 ID 와 0xFF 값을 두 번 슬래이브에 전달합니다. 그런데, 채널 ID 는 왜 전달하는지 알겠는데, 0xFF 값은 어떤 의미일까요? 실제로 이 값은 아무 의미가 없습니다. 0xFF 대신 다른 어떤 값을 전달해 주어도 상관이 없습니다. 이렇게 하는 이유는 위의 파형을 보시면 마스터에서 MOSI 신호가 나가면서 MISO 신호가 들어옵니다. 이것은 마스터에서 무엇인가를 받기 위해서는 의미없는 무엇이라도 줘야 하는 것입니다. 이 ADC 에서는 10 비트의 신호가 다음과 같이 입력이 됩니다.



위의 그림에서 붉은색 부분이 MISO 신호에 실려 오는 것으로 라즈베리파이에서 읽어야할 데이터입니다. 이제 위의 코드에서 spi.xfer() 함수와 위의 그림을 연결해 보겠습니다.

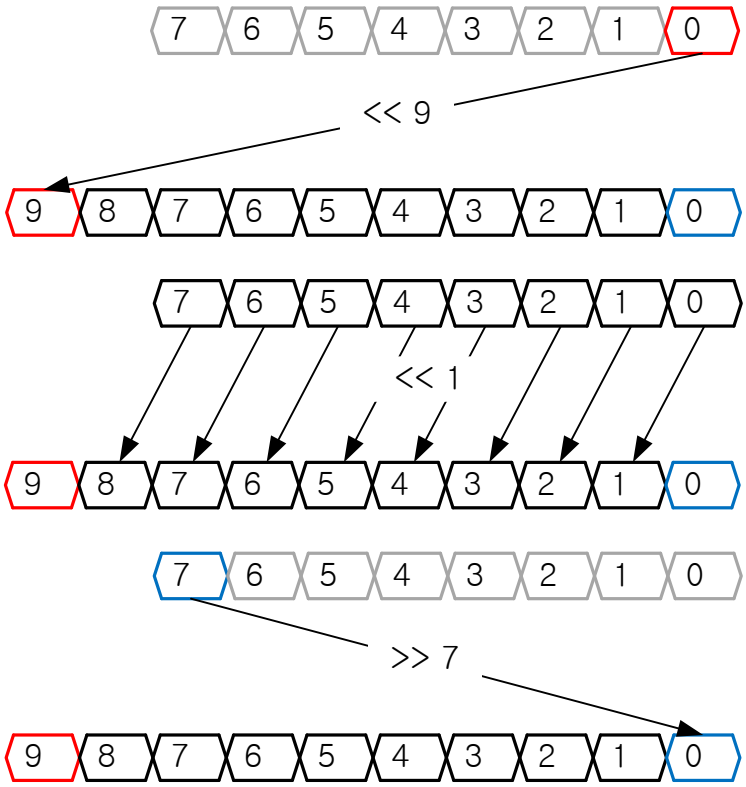
```
adcVal_list = spi.xfer([ADC_VR1, 0xFF, 0xFF])
```

spi.xfer() 함수의 인자는 채널 ID 와 슬래이브에서 값을 받기 위한 0xFF 값 두 개입니다. 이것은 파형의 MOSI 에서 3 바이트의 값을 전달한 것과 같습니다. 이렇게 값을 주면 MISO 로 위의 그림과 같이 세 바이트에 걸쳐서 ADC 값 10 비트를 받습니다. 이것은 adcVal_list 리스트 변수에 저장됩니다. 그런데 이렇게 전달되는 10 비트가 처음부터 오는 것은 아니고 첫번째 바이트의 마지막 비트부터 전달이 되어 세번째 바이트의 첫번째 비트까지입니다. spi.xfer() 함수는 반환 값이 리스트입니다. 이 리스트의 원소들은 8 비트로 구성이 됩니다. 즉, SPI 가 바이트 단위로 전송되기 때문에 리스트의 원소도 8 비트로 구성이 됩니다. 이 리스트의 첫번째 원소는 그림에서 MISO 의 첫번째 바이트가 저장됩니다. ADC 는 10 비트 값을 전달해 주기 때문에 adcVal_list 리스트의 첫번째 원소중 하위 1 비트만이 의미가 있고 나머지는 의미가 없습니다. 이 1 비트가 두번째 원소 8 비트하고 세번째 원소의 1 비트와 합쳐져서 ADC 값 10 비트를 구성하는 것입니다. 이렇게 구성된 10 비트가 합쳐진 값을 구하는 것은 다음 코드와 같습니다.

```
adcVal = (adcVal_list[0] << 9) + (adcVal_list[1]<<1) + (adcVal_list[2] >> 7)
```

첫번째 원소는 위의 파형 그림과 같이 9 번 비트이므로 9 만큼 왼쪽으로 쉬프트하고, 2 번째 원소는 1~8 번 비트이므로 1 만큼 왼쪽으로 쉬프트합니다. 세번째 원소는 0 번째 비트이므로 8 비트의 최상위에 있는 세번째 원소의 비트를 7 만큼 오른쪽으로 쉬프트 합니다. 이것을 그림으로 도식화하면 다음 그림과 같습

니다.

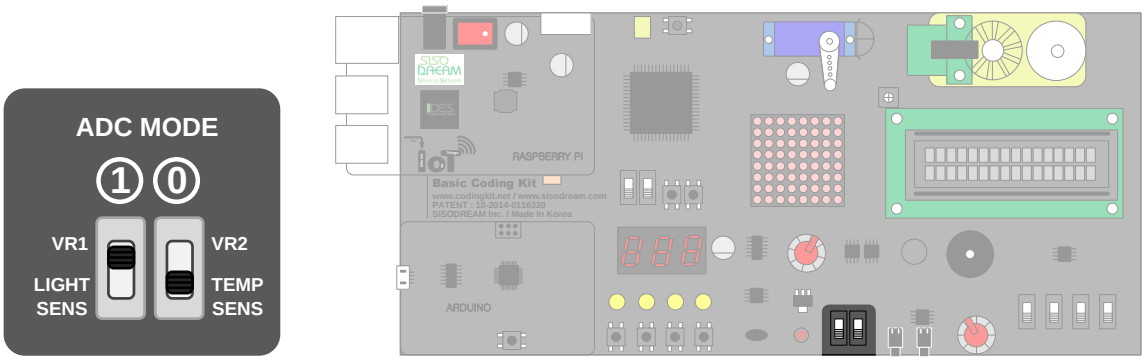


이렇게 하여 아날로그 값을 얻었습니다. 전체 코드는 다음과 같습니다.

```
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)
import time
import spidev

ADC_SPI_SS = 8
ADC_VR1 = 0xDF

GPIO.setup(ADC_SPI_SS, GPIO.OUT)
GPIO.output(ADC_SPI_SS, 1)

spi = spidev.SpiDev()
spi.open(0, 0)

while 1 :
    GPIO.output(ADC_SPI_SS, 0)
    adcVal_list = spi.xfer([ADC_VR1, 0xFF, 0xFF])
    GPIO.output(ADC_SPI_SS, 1)
    adcVal = (adcVal_list[0] << 9) + (adcVal_list[1]<<1) + (adcVal_list[2] >> 7)
    print "VR1 : ", adcVal, hex(adcVal)
    time.sleep(0.5)
```

이제 코딩 키트에서 실행 시켜 보십시요. 그리고 가변저항을 돌려서 값을 바꾸어 보십시요. 주의하실 것은

ADC Mode Switch 중 1 번은 위로 올려 주세요.



다음과 같이 가변저항 값이 출력되는 것을 터미널에서 확인할 수 있습니다.

```
VR1 :   0 0x0
VR1 :   0 0x0
VR1 :   2 0x2
VR1 :   107 0x6b
VR1 :   225 0xe1
VR1 :   342 0x156
VR1 :   464 0x1d0
VR1 :   595 0x253
VR1 :   711 0x2c7
VR1 :   849 0x351
VR1 :   933 0x3a5
VR1 :   978 0x3d2
VR1 :   978 0x3d2
VR1 :   1022 0x3fe
VR1 :   1022 0x3fe
VR1 :   1022 0x3fe
VR1 :   1022 0x3fe
VR1 :   1022 0x3fe
```

출력은 10 진수 값과 16 진수 값을 동시에 하도록 하였습니다.

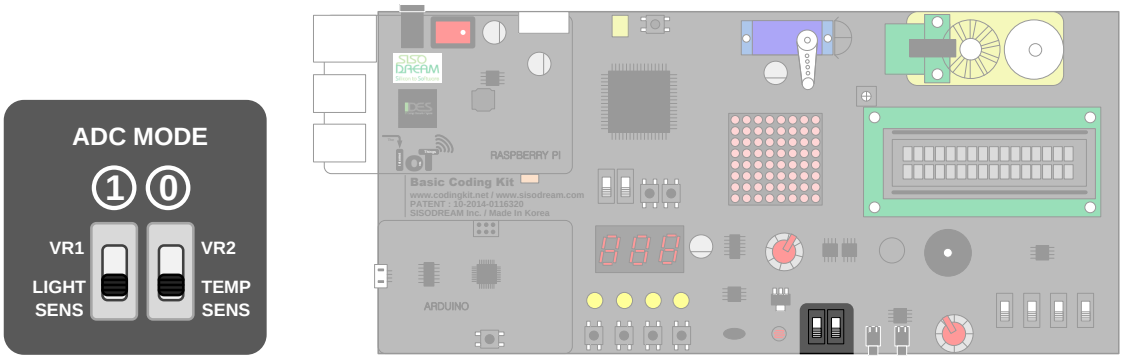
< 예제 코드 : **가변저항 값 읽기** >

< 코드 위치 : **Codingkit / vr_print.py** >

[센서 값 입력 받기 : 밝기, 온도, 소리, 적외선 센서]

센서 관련된 이론적인 부분들은 아두이노에서 많이 다루었기 때문에 여기서는 파이썬 코드로 어떻게 센서 값을 읽어 들이는지에 대한 것을 알아보도록 하겠습니다. 첫번째 예제에서는 밝기, 온도, 소리, 적외선 센서 각각을 스위치 4 개가 ON 되면 그 값을 각각 읽어 들이는 것입니다. 예를 들면 0 번 스위치가 ON 이 되면 밝기 센서의 값을 터미널에 print 하는 것입니다.

센서 값은 가변저항 예제와 마찬가지로 SPI 에 연결된 ADC 의 값을 읽어 오는 것입니다. 그래서 가변저항 값을 읽어 오는 것과 매우 비슷하게 코딩하면 됩니다. 그리고 일단 코딩 키트의 ADC 모드 스위치는 아래 그림과 같이 모두 아래로 내려 줍니다.



그러면 ADC 채널 ID 부터 다음과 같이 정의해 줍니다.

```
ADC_CH_LIGHT   = 0xDF  # ADC Channel ID - Light
ADC_CH_TEMP    = 0xD7  # ADC Channel ID - Temperature
ADC_CH_SOUND = 0xCF  # ADC Channel ID - Sound
ADC_CH_IR        = 0xC7  # ADC Channel ID - IR
```

이 채널을 이용하여 ADC 에서 값을 읽어 오는 함수는 다음과 같습니다.

```
def ck_adcRd(chID) :
    GPIO.output(ADC_SPI_SS, 0)
    adcVal_list = spi.xfer([chID, 0xFF, 0xFF])
    GPIO.output(ADC_SPI_SS, 1)
    adcVal = ((adcVal_list[0] << 9) + (adcVal_list[1]<<1) + (adcVal_list[2] >> 7)) & 0x3ff
    return(adcVal)
```

가변저항의 코드와 똑같은 코드를 함수로 만들었을 뿐입니다. 스위치의 핀을 정의하고 리스트로 저장합니다.

```
SWITCH_0 = 26
SWITCH_1 = 27
SWITCH_2 = 2
SWITCH_3 = 3

sensor_switch = [SWITCH_0, SWITCH_1, SWITCH_2, SWITCH_3]
```

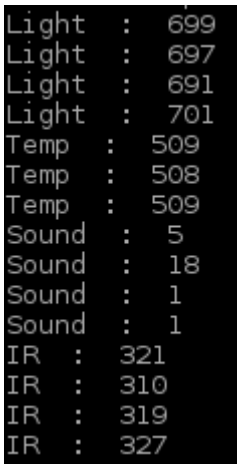
스위치의 리스트 원소에 맞춰 센서의 이름과 센서의 채널 ID 를 리스트로 저장합니다.

```
sensor_name = ["Light", "Temp", "Sound", "IR"]
sensor_chID = [ADC_CH_LIGHT, ADC_CH_TEMP, ADC_CH_SOUND, ADC_CH_IR]
```

이것은 다음과 같이 for 문을 활용하여 각 센서의 값을 출력하려고 하는 것입니다.

```
while 1 :
  for i in range(4) :
    if (GPIO.input(sensor_switch[i]) == SW_ON) :
      print sensor_name[i], " : ", ck_adcRd(sensor_chID[i])
  time.sleep(0.5)
```

while 문 안의 for 문은 0 ~ 4 까지 인덱스를 반복합니다. 어떤 스위치가 ON 이 되었는지 확인하고 ON 이 된 스위치에 해당하는 센서의 이름과 ADC 값을 출력해 줍니다. 코딩 키트에서 실행 시키면 다음과 같이 동작합니다.



위의 그림은 스위치를 하나씩 올렸다 내린 것입니다. 모든 스위치를 내리면 아무 것도 출력하지 않습니다. 전체 코드는 다음과 같습니다.

```
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)
import time
import spidev

ADC_SPI_SS = 8

ADC_CH_LIGHT = 0xDF  # ADC Channel ID - Light
ADC_CH_TEMP  = 0xD7  # ADC Channel ID - Temperature
ADC_CH_SOUND = 0xCF  # ADC Channel ID - Sound
ADC_CH_IR    = 0xC7  # ADC Channel ID - IR

GPIO.setup(ADC_SPI_SS, GPIO.OUT)
GPIO.output(ADC_SPI_SS, 1)
```

```
spi = spidev.SpiDev()
spi.open(0, 0)

def ck_adcRd(chID) :
    GPIO.output(ADC_SPI_SS, 0)
    adcVal_list = spi.xfer([chID, 0xFF, 0xFF])
    GPIO.output(ADC_SPI_SS, 1)
    adcVal = ((adcVal_list[0] << 9) + (adcVal_list[1]<<1) + (adcVal_list[2] >> 7)) & 0x3ff
    return(adcVal)

SWITCH_0 = 26
SWITCH_1 = 27
SWITCH_2 = 2
SWITCH_3 = 3

SW_ON = 1
SW_OFF = 0

sensor_name = ["Light", "Temp", "Sound", "IR"]
sensor_switch = [SWITCH_0, SWITCH_1, SWITCH_2, SWITCH_3]
sensor_chID = [ADC_CH_LIGHT, ADC_CH_TEMP, ADC_CH_SOUND, ADC_CH_IR]

for SWITCH in sensor_switch:
    GPIO.setup(SWITCH, GPIO.IN)

IR_LED = 13
GPIO.setup(IR_LED, GPIO.OUT)
GPIO.output(IR_LED, 1)

while 1 :
    for i in range(4) :
        if (GPIO.input(sensor_switch[i]) == SW_ON) :
            print sensor_name[i], " : ", ck_adcRd(sensor_chID[i])
    time.sleep(0.5)
```

< 예제 코드 : **센서 값 읽어 출력하기** >

< 코드 위치 : **Codingkit / sensor_print.py** >

[DC 모터]

아두이노 파트에서 보았듯이 DC 모터를 컨트롤하는데는 다음과 같이 3 개의 신호가 사용됩니다.

신호명	설명
DCMOT_EN	Enable 신호
DCMOT_FWD	정방향 Enable 신호
DCMOT_BWD	역방향 Enable 신호

이 중 Enable 신호는 GPIO.output() 함수를 이용하여 0 과 1 값을 설정하여 줍니다. 정방향 신호와 역방향 신호는 PWM 신호를 전달하여 회전 속도를 조정할 수 있습니다. PWM 신호는 부저와 마찬가지로 SPI 커맨드로 처리합니다. 다음은 정방향으로 회전하는 DC 모터 코드입니다.

```
import time
import CK_SPI_DEV as spi
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)

DCMOT_EN = 17

GPIO.setup(DCMOT_EN, GPIO.OUT)
GPIO.output(DCMOT_EN, 1)
spi.ck_spiWr(spi.SPI_CMD_CLEAR, 0)

spi.ck_pwmDCMot_fwd(1000, 50)

time.sleep(5)

# Disable
GPIO.output(DCMOT_EN, 0)
spi.ck_spiWr(spi.SPI_CMD_PWM_DM_FWD_EN, 0)
```

CK_SPI_DEV 모듈을 import 하였고 이름은 spi 로 했습니다. DCMOT_EN 신호는 GPIO.setup() 함수와 GPIO.output() 함수로 컨트롤 합니다. DCMOT_EN 에는 1 을 써 주어 DC 모터를 Enable 합니다. spi.ck_spiWr(spi.SPI_CMD_CLEAR, 0) 는 SPI 로 컨트롤되는 신호들을 초기화 해 줍니다. spi.ck_pwmDCMot_fwd(1000, 50) 함수는 DCMOT_FWD 신호의 주파수와 듀티비를 설정하고 PWM 신호를 생성하는 함수입니다. 첫번째 인자는 주파수로, 1000 을 입력하면 1000 Hz 의 주파수가 생성됩니다. 두번째 인자는 듀티비입니다. 50 으로 입력하면 50 % 의 듀티비가 생성됩니다. 이렇게 주파수와 듀티비를 인자로 전달하고 함수를 호출하면 1000 Hz, 50 % 의 PWM 신호가 생성이 됩니다.

time.sleep(5) 로 5 초간 동작을 시키고 DC 모터를 끕니다. 끄기 위해서 DCMOT_EN 신호에 0 을 써 주고, ck_spiWr() 함수를 이용하여 DCMOT_FWD 신호에 0 을 줍니다. DCMOT_FWD 신호에 PWM 신호를 인가하기 위해서는 다음과 같은 커맨드들이 사용됩니다.

커맨드	코드	설명
SPI_CMD_PWM_DM_FWD_EN	0x20	DCMOT_FWD 로 PWM 신호를 출력합니다. 다른 모든 DCMOT_FWD 관련 커맨드를 실행한 다음에 이 커맨드를 실행합니다.
SPI_CMD_PWM_ DM_FWD_FR_H	0x21	DCMOT_FWD PWM 신호의 주파수 중 상위 8 비트를 설정합니다.
SPI_CMD_PWM_ DM_FWD_FR_L	0x22	DCMOT_FWD PWM 신호의 주파수 중 하위 8 비트를 설정합니다.
SPI_CMD_PWM_ DM_FWD_DUTY	0x23	DCMOT_FWD PWM 신호의 듀티비를 설정합니다. 0 ~ 100 의 범위를 갖습니다.

ck_pwmDCMot_fwd() 함수는 위의 커맨드들을 이용하여 구현되었습니다. 이 함수는 위에서 import 한 CK_SPI_DEV 모듈에 있습니다. 이 함수의 코드는 다음과 같습니다.

```
def ck_pwm(cmd, freq, duty=50) :
    freq_set_val = 300000 / freq
    ck_spiWr(cmd+1, (freq_set_val >> 8) & 0xff) # Freq H
    ck_spiWr(cmd+2, freq_set_val & 0xff)        # Freq L
    ck_spiWr(cmd+3, duty)                       # DUTY
    ck_spiWr(cmd, 1)                            # EN

def ck_pwmDCMot_fwd(freq, duty=50) :
    ck_pwm(SPI_CMD_PWM_DM_FWD_EN, freq, duty)
```

위의 ck_pwm 함수는 초기 커맨드와 주파수, 듀티비를 입력으로 받아 SPI 커맨드를 이용하여 PWM 신호를 생성합니다. 먼저 입력 받은 주파수를 부저에서 설명했던 다음 공식을 이용하여 실제 주파수 설정 입력값으로 바꾸어 줍니다.

300,000 / 주파수 = 주파수_설정_입력값

여기서는 주파수가 1000 이므로 다음 공식으로 실제 주파수 설정 입력값은 300 이 됩니다.

300,000 / 1000 = 300

이 공식은 " freq_set_val = 300000 / freq" 로 코딩하였습니다. 다음은 주파수의 상위 8 비트를 ck_spiWr() 함수를 이용하여 설정하고 다음으로 주파수의 하위 8 비트를 설정합니다. 그리고 듀티비를 설정합니다. 끝으노 Enable 신호를 설정하여 PWM 신호를 생성합니다. ck_spiWr() 함수에 cmd 인자를 전달하는데, 순서가 +1, +2, +3, +0 으로 되어 있습니다. 이유는 DCMOT_FWD 커맨드가 다음과 같이 1씩 차이나는 값이기 때문에 SPI_CMD_PWM_DM_FWD_EN 에 +1, +2, +3 을 하면 각각 SPI_CMD_PWM_DM_FWD_FR_H, SPI_CMD_PWM_DM_FWD_FR_L, SPI_CMD_PWM_DM_FWD_DUTY 커맨드가 되기 때문입니다.

```
SPI_CMD_PWM_DM_FWD_EN   = 0x28
SPI_CMD_PWM_DM_FWD_FR_H = 0x29
SPI_CMD_PWM_DM_FWD_FR_L = 0x2a
SPI_CMD_PWM_DM_FWD_DUTY = 0x2b
```

이 ck_pwm() 함수를 이용하여 ck_pwmDCMot_fwd() 함수를 만듭니다. 이 함수에서는 ck_pwm() 함수를 호출하는데, 첫번째 인자로는 DCMOT_FWD 커맨드 중 가장 값이 작은 SPI_CMD_PWM_DM_FWD_EN 커맨드를 줍니다. 두번째 인자는 주파수이고 세번째 인자는 듀티비입니다. 이 세번째 인자를 "duty = 50" 이라고 쓰면 이 함수를 호출할 때 세번째 인자인 듀티비를 써 주지 않으면 기본 값으로 50 이 된다는 의미입니다.

이제 코딩 키트에서 실행해 보십시요. 정방향으로 DC 모터가 회전하는 것을 볼 수 있을 것입니다.

< 예제 코드 : **SPI 모듈을 이용하여 DC 모터 정방향으로 돌리기** >

< 코드 위치 : Codingkit / **dcmot_spi_pwm.py** >

< 연습 문제 : **DC 모터 역방향으로 돌리기** >

이번에는 DC 모터를 역방향으로 돌리겠습니다. 역방향으로 돌리는 SPI 커맨드 함수는 다음과 같습니다.

```
def ck_pwmDCMot_bwd(freq, duty=50) :
  ck_pwm(CMD_PWM_DM_BWD_EN, freq, duty)
```

역방향 커맨드는 다음과 같습니다.

```
SPI_CMD_PWM_DM_BWD_EN   = 0x2C
SPI_CMD_PWM_DM_BWD_FR_H = 0x2d
SPI_CMD_PWM_DM_BWD_FR_L = 0x2e
SPI_CMD_PWM_DM_BWD_DUTY = 0x2f
```

이 함수와 커맨드를 이용하여 DC 모터를 역방향으로 돌려 보십시요.

< 코드 위치 : Codingkit / **dcmot_spi_pwm_inv.py** >

다음은 가변 저항을 이용하여 DC 모터의 속도를 제어해 보겠습니다. DCMOT_FWD 의 PWM 신호의 듀티비를 조정하면 속도를 바꿀 수 있다고 아두이노 파트에서 배웠습니다. 그럼 이제 라즈베리파이에서 가변저항 값을 입력받아 이것으로 PWM 신호를 만들어 보겠습니다. PWM 신호는 위에서 사용했던 ck_pwmDCMot_fwd() 함수를 이용하여 만들겠습니다. 이 PWM 신호의 주파수는 1000 Hz 로 하겠습니다. 인자 중 듀티비는 가변저항의 값을 입력받아 대입해 주어야 합니다. 그런데, 가변저항 값은 0 ~ 1023 까

지 변하고 듀티비는 0 ~ 100 까지 변합니다. 이렇게 값의 범위가 다를 때는 아두이노에서는 map() 함수를 이용하여 쉽게 바꾸었습니다. 그래서 라즈베리파이에서도 map() 함수와 비슷한 ck_map() 함수를 다음과 같이 만들었습니다. 이 함수는 두고 두고 유용하게 사용될 것입니다.

```
def ck_map(x, in_min, in_max, out_min, out_max) :
    out_val = (((x - in_min) * (out_max - out_min)) / (in_max - in_min)) + out_min
    return out_val
```

이 ck_map() 함수를 이용하여 가변저항의 입력 값의 범위를 0 ~ 100 으로 변환한 다음 ck_pwmDCMot_fwd() 함수에 듀티비 인자로 전달하는 것입니다. 이 코드는 다음과 같습니다.

```
speed = spi_adc.ck_adcRd(spi_adc.ADC_CH_VR1)
speed = ck_map(speed, 0, 1023, 0, 100)
spi_dev.ck_pwmDCMot_fwd(1000, speed)
```

가변저항 값을 읽어 speed 변수에 저장합니다. 이 변수는 ck_map() 함수를 통하여 0 ~ 100 범위로 변환되고 이 값은 ck_pwmDCMot_fwd() 함수의 듀티비 인자로 전달됩니다. 이렇게 하면 가변저항 값을 바꿈에 따라서 듀티비가 바뀌고 듀티비가 바뀌면 DC 모터의 회전 속도도 바뀝니다.

위의 코드를 보면 spi_adc 와 spi_dev 모듈을 사용했음을 알 수 있습니다. 이 모듈들은 제공해드리는 마이크로 SD 카드에 이미 설치가 되어 있습니다. 이것은 다음과 같이 import 합니다.

```
import CK_SPI_ADC as spi_adc
import CK_SPI_DEV as spi_dev
```

이 두 모듈은 SPI 관련 함수들이 모여 있는 모듈입니다. 먼저 CK_SPI_ADC 를 설명드리겠습니다. 이 모듈에서는 코딩 키트에 있는 센서와 가변저항의 아날로그 값을 SPI 인터페이스를 통하여 읽어들이는 함수들이 모여 있는 모듈입니다. 이전에 SPI 를 통하여 센서 값과 가변저항 값을 읽어 들이는 설명 및 예제는 충분히 보았습니다. 이것을 정리하여 CK_SPI_ADC 모듈로 만들었습니다.

이 모듈에 내장된 ADC 채널 ID 변수는 다음과 같습니다.

디바이스	ADC 채널 ID 변수	채널 ID	ADC Mode Switch
적외선 센서	**ADC_CH_IR**	0xC7	X
소리 센서	**ADC_CH_SOUND**	0xCF	X
온도 센서	**ADC_CH_TEMP**	0xD7	Down
가변저항 2	**ADC_CH_VR2**		Up
밝기 센서	**ADC_CH_LIGHT**	0xDF	Down
가변저항 1	**ADC_CH_VR1**		Up

이 모듈에 내장된 ADC 값을 읽는 함수는 다음과 같습니다.

```
def ck_adcRd(chID) :
    GPIO.output(ADC_SPI_SS, 0)
    adcVal_list = spi.xfer([chID, 0xFF, 0xFF])
    GPIO.output(ADC_SPI_SS, 1)
    adcVal = ((adcVal_list[0] << 9) + (adcVal_list[1]<<1) + (adcVal_list[2] >> 7)) & 0x3ff
    return(adcVal)
```

전에 다 설명드렸던 함수입니다. 채널 ID 를 인자로 입력 받아 SPI 인터페이스를 통해서 해당 채널에 연결된 디바이스 값을 읽어오는 함수입니다. CK_SPI_ADC 모듈은 이렇게 구성이 되어 있습니다.

더 자세한 내용은 이 모듈 소스 코드를 참고해 주십시요. 코드는 위치는 다음과 같습니다.

< 코드 위치 : Codingkit / CodingkitSPI / **CK_SPI_ADC.py** >

CK_SPI_DEV 모듈은 이전에 DC 모터의 예제에서 사용했습니다. 그 때는 다음과 같이 import 했습니다.

```
import CK_SPI_DEV as spi
```

그런데 이번에는 SPI 관련 모듈 2 개를 import 해야 해서 이름을 spi_dev 로 바꾼 것입니다. 이 모듈은 SPI 를 이용하여 여러 디바이스를 컨트롤하는 모듈입니다. 아직까지는 PWM 신호를 만드는 용도로 주로 사용하였는데, 앞으로는 아두이노와 마찬가지로 도트매트릭스나 FND 등도 컨트롤할 것입니다. PWM 신호를 만드는 함수에 대해서는 앞에서 설명을 드렸고 다른 디바이스들에 관해서 차차 설명드리겠습니다. 궁금하신 것이 있거나 자세한 내용을 알고 싶으시면 소스 코드를 참고해 주세요. 코드는 다음 위치에 있습니다.

< 코드 위치 : Codingkit / CodingkitSPI / **CK_SPI_DEV.py** >

전체 코드는 다음과 같습니다.

```
import time
import CK_SPI_ADC as spi_adc
import CK_SPI_DEV as spi_dev
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)

DCMOT_EN = 17

GPIO.setup(DCMOT_EN, GPIO.OUT)
GPIO.output(DCMOT_EN, 1)
spi_dev.ck_spiWr(spi_dev.SPI_CMD_CLEAR, 0)


def ck_map(x, in_min, in_max, out_min, out_max) :
    out_val = (((x - in_min) * (out_max - out_min)) / (in_max - in_min)) + out_min
```

```
    return out_val

while (1) :
    for i in range(20) :
        speed = spi_adc.ck_adcRd(spi_adc.ADC_CH_VR1)
        speed = ck_map(speed, 0, 1023, 0, 100)
        spi_dev.ck_pwmDCMot_fwd(1000, speed)
        print i, ": SPEED : ", speed
        time.sleep(0.5)
    key = raw_input('Quit?? (y/n) : ')
    if (key == 'y') :
        break;

# Off
spi_dev.ck_spiWr(spi_dev.SPI_CMD_CLEAR, 0)
```

위의 코드 중 대부분은 위에서 설명드렸습니다. 루프문과 break, 키 입력 등에 대해서 설명 드리고 넘어가겠습니다.

< 문법 설명 : **키 입력 raw_input() 함수** >

raw_input() 내장 함수는 화면에 어떤 문구를 출력하고 값을 입력 받을 수 있는 프롬프트를 보여 줍니다. 여기에 사용자가 문자를 입력하면 raw_input() 함수는 이 문자를 반환합니다. Python2 프로그램에서 다음과 같이 입력하면 사용자가 지정한 값이 출력되는 것을 알 수 있습니다.

```
>>> raw_input('Quit?? (y/n) : ')
Quit?? (y/n) : y
'y'
>>> raw_input('Quit?? (y/n) : ')
Quit?? (y/n) : n
'n'
```

위의 예제 코드에서는 "key = raw_input('Quit?? (y/n) : ')" 하면 사용자에게 루프를 빠져 나갈 (Quit) 것이냐고 질문을 하고 입력한 값은 key 변수에 저장이 됩니다. 이 값이 'y' 이면 break 문을 실행하여 while 루프를 빠져 나갑니다. 이 값이 'n' 이 면 while 루프는 계속해서 반복하게 됩니다. while 문 안에 for 문이 있는데, 이것은 가변저항 값으로 DC 모터의 속도를 제어하는 코드를 20 번 반복해서 수행하겠다는 것입니다.

raw_input 함수는 문자만을 반환합니다. 여러분이 숫자를 입력하여도 문자로 반환을 합니다. 즉 여러분이 20 이라는 숫자를 입력하면 이것은 문자의 20 이지 숫자의 20 은 아닙니다. Python2 프로그램에 다음과 같은 테스트를 해 보겠습니다.

먼저 raw_input('Input Number : ') 하여 숫자를 입력하라는 메시지를 출력하고 숫자 20 을 입력해 보겠습

니다.

```
>>> raw_input('Input Number : ')
Input Number : 20
'20'
>>> key = raw_input('Input Number : ')
Input Number : 20
>>> key
'20'
>>> int(key)
20
>>> key
'20'
>>> key + 10

Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    key + 10
TypeError: cannot concatenate 'str' and 'int' objects
>>> int(key) + 10
30
```

작은 따옴표가 붙은 ('20') 20 이 출력이 됩니다. 이것은 문자라는 의미입니다. 이 값을 key 변수에 저장했습니다. 그리고 int(key) 하여 key 를 정수로 바꾸어 주었습니다. 그러니까 작은 따옴표가 생략된 20 이 출력이 됩니다. key 에 10 을 더해 보겠습니다. 파이썬 프로그램에서는 문자('str' : string)와 숫자('int' : integer) 는 더할 수 없다고 에러를 냅니다. 그래서 int(key) + 10 을 합니다. 그러면 30 이라고 값이 정상적으로 출력이 됩니다. 이렇게 raw_input() 함수는 문자만을 반환한다는 것을 명심하고 가도록 하겠습니다. 그리고 한 가지 더 여기서 계속 문자라고 했는데, 일반적으로 컴퓨터 언어에서는 한 문자는 문자라고 하고 여러 문자가 모여 있는 것은 보통 문자열이라고 합니다. 문자열이란 문자가 모인 배열이란 뜻입니다. 여기서도 raw_input 함수는 문자열을 반환합니다. 이 문자열은 int() 내장 함수를 이용하여 숫자로 바꿉니다. 이 숫자를 컴퓨터 언어에서는 보통 정수라고 합니다. 전문 용어도 잘 알아두면 좋습니다. 프로그램 전문가의 말을 알아 들을 수 있고 관련 서적을 볼 수도 있고 코딩 관련 질문도 멋지게 전문 용어로 할 수 있습니다.

< 문법 설명 : **break** >

파이썬의 break 는 C 언어의 break 와 같습니다. break 구문을 실행시키면 해당 루프문을 빠져나가게 되어 있습니다. 위의 예제 코드에서도 'y' 값을 입력 받으면 while 문을 빠져 나갑니다. 그리고 DC 모터를 끄는 코드를 실행합니다.

```
while (1) :
  for i in range(20) :
    speed = spi_adc.ck_adcRd(spi_adc.ADC_CH_VR1)
    speed = ck_map(speed, 0, 1023, 0, 100)
    spi_dev.ck_pwmDCMot_fwd(1000, speed)
    print i, ": SPEED : ", speed
    time.sleep(0.5)
  key = raw_input('Quit?? (y/n) : ')
```

```
    if (key == 'y') :
        break;

# Off
spi_dev.ck_spiWr(spi_dev.SPI_CMD_CLEAR, 0)
```

그런데 파이썬의 break 는 C 언어와는 다른 한 가지 기능이 더 있습니다. else 문을 수행할 수 있다는 것입니다.

```
루프문 :
  문장1
  break
else :
  문장2
```

루프문을 break 로 빠져 나오게 되면 else 문을 수행 안 합니다. 하지만 루프문을 모두 끝내고 나면 else 문을 수행합니다. 다음 예제를 한 번 보겠습니다.

```
>>> for i in range(5) :
        print(i)
        if (i == 3) :
            print("break")
            break
else :
        print("no break and else")

0
1
2
3
break
```

위와 같이 break 에 의해서 for 문을 빠져 나오게 되면 else 문을 수행하지 않습니다. 파란색의 출력들을 보면 0 에서 3 까지를 출력하고 "if (i == 3)" 안에 있는 "print("break")" 에 의해서 "break" 를 출력합니다. 이렇게 break 에 의해서 for 루프문을 빠져 나왔기 때문에 else 문은 수행하지 않습니다. 다음은 루프문을 정상적으로 끝내는 예입니다.

```
>>> for i in range(5) :
        print(i)
        if (i == 10) :
            print("break")
            break
else :
        print("no break and else")

0
1
2
3
4
no break and else
```

위의 예에서는 for 문은 0 ~ 4 로 인덱스가 변하는데 if 문은 10 일때 break 가 걸립니다. 그래서 파란색의 출력을 보면 0 ~ 4 가 출력이 되고 for 문을 빠져 나옵니다. break 로 빠져 나온 것이 아니기 때문에 else 문을 수행하여 "no break and else" 문장을 출력합니다.

파이썬에서도 아두이노에서 다루었던 break, continue 문과 같은 break, continue 문이 존재합니다. break 문은 앞에서 설명드렸고 이제 continu 문을 설명드리겠습니다. 파이썬에서의 continue 문은 C 언어와 동일하게 사용이 됩니다. 다음 예를 보겠습니다.

```
>>> for i in range(5) :
        if (i == 3) :
            print("continue")
            continue
        print(i)

0
1
2
continue
4
```

위의 예를 보면 if 문에 의해서 i 가 3 일 때 continue 문이 수행이 됩니다. continue 문이 수행이 되면 루프문 안의 문장 중 continue 다음 문장들은 수행하지 않고 루프문을 반복합니다. 그래서 파란색의 출력을 보면 3 은 출력이 되지 않고 대신 continue 라는 문자가 출력된 것을 볼 수 있습니다. 이것은 continue 문 다음의 "print(i)" 가 수행되지 않은 것입니다.

이제 가변저항으로 DC 모터의 속도를 제어하는 예제를 코딩 키트에서 실행해 보십시요. 그리고 가변저항을 돌려서 DC 모터의 회전 속도를 바꾸어 보십시요. 다음과 같은 내용도 같이 출력이 됩니다.

```
0 : SPEED :  19
1 : SPEED :  19
2 : SPEED :  19
3 : SPEED :  21
4 : SPEED :  32
5 : SPEED :  38
6 : SPEED :  46
7 : SPEED :  52
8 : SPEED :  62
9 : SPEED :  70
10 : SPEED :  76
11 : SPEED :  83
12 : SPEED :  92
13 : SPEED :  99
14 : SPEED :  99
15 : SPEED :  99
16 : SPEED :  99
17 : SPEED :  87
18 : SPEED :  84
19 : SPEED :  80
Quit?? (y/n) :
```

여기서 n 을 누르면 계속 파이썬 코드가 실행이 되고 y 를 누르면 끝을 냅니다. 20 번 반복이 너무 짧은 것 같으면 range() 함수의 인자를 바꾸면 됩니다.

< 예제 코드 : **가변저항으로 DC 모터의 회전 속도 제어하기** >

< 코드 위치 : Codingkit / **dcmot_speed.py** >

< 연습 문제 : **스위치를 이용하여 DC 모터 방향 바꾸기** >

위에서 가변저항으로 DC 모터의 회전 속도를 바꾸어 보았습니다. 이번에는 회전 속도 뿐만 아니라 스위치 값에 따라서 DC 모터의 회전 방향도 바꾸어 보도록 하겠습니다. 스위치 값이 0 이면 정방향으로 회전하고 스위치 값이 1 이면 역박향으로 회전하는 것입니다. 초기 정의 코드는 다음과 같습니다.

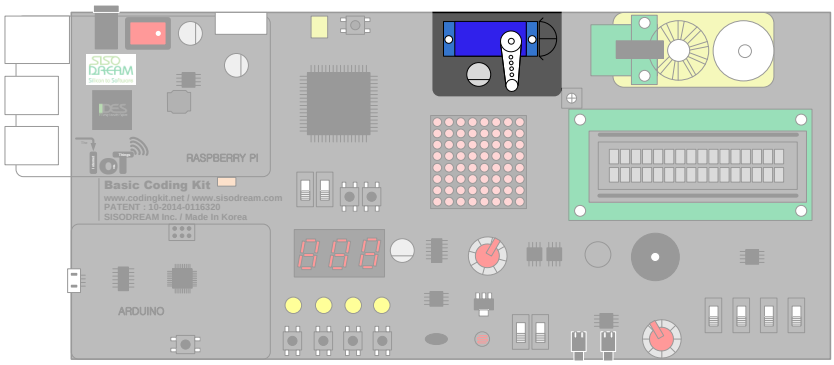
```
import time
import CK_SPI_ADC as spi_adc
import CK_SPI_DEV as spi_dev
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)

DCMOT_EN = 17
GPIO.setup(DCMOT_EN, GPIO.OUT)
GPIO.output(DCMOT_EN, 1)

SWITCH_0 = 26
GPIO.setup(SWITCH_0, GPIO.IN)

DCMOT_DIR_SW_FWD = 0
DCMOT_DIR_SW_BWD = 1
```

나머지 코드는 직접 작성해 보십시요.

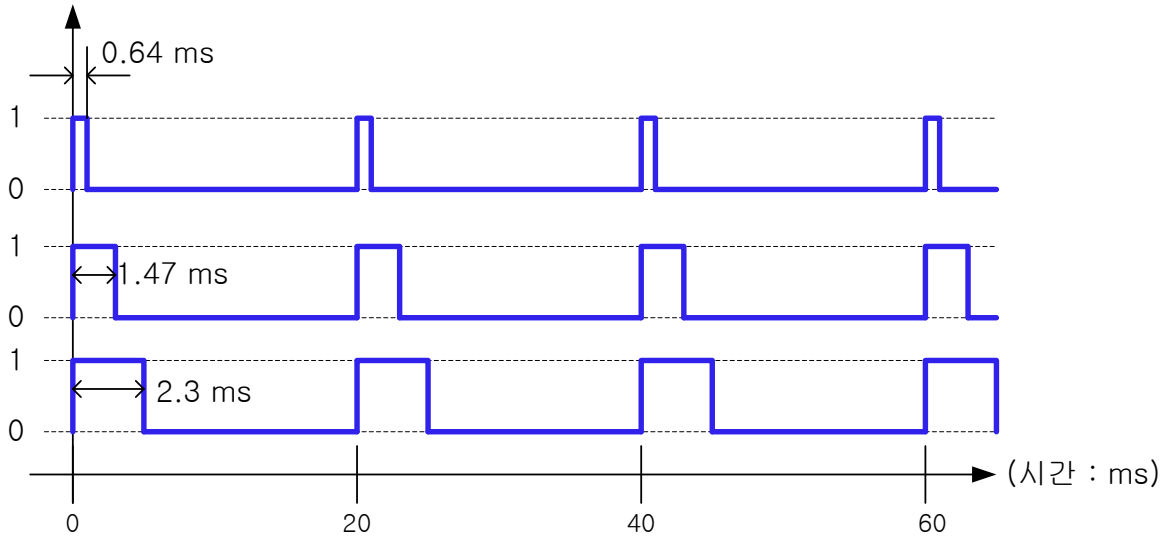
< 코드 위치 : Codingkit / **dcmot_speed_dir.py** >

[서보 모터]

이제부터는 서보 모터를 동작시켜 보겠습니다. 서보 모터는 다음 사진과 같고 PWM 신호를 받아서 그 신호에 따라 서보 모터의 혼(사진 참조)을 특정 각도로 움직입니다. 이 서버 모터의 혼을 서버혼이라고 부릅니다.



PWM 신호의 주기는 20 ms 이고 PWM 신호 중 1 인 구간이 0.64 ms 에서 2.3 ms 범위 내에서 움직입니다. 여기서 ms 는 밀리세크(Mili-second)의 약자이고 1000 분의 1 초입니다.



0.64 ms 는 0 도이고 2.3 ms 는 180 도입니다. 서보 모터는 이렇게 PWM 신호의 1 인 구간의 값에 따라서

0 도 ~ 180 도까지 변합니다. 그러면 1.47 ms 는 90 도가 될 것입니다.

부저와 DC 모터 관련 코딩에서 PWM 신호에 대해서 배웠습니다. 이 때 CK_SPI_DEV 라는 모듈을 import 하여 매우 쉽게 구현하였습니다. 서보 모터도 이 모듈을 import 하여 코딩하겠습니다. 그리고 이 모듈에 있는 ck_servoMot() 함수를 이용하여 서보 모터를 제어합니다. 이 함수는 인자로 각도를 받습니다. 그래서 서보혼을 해당 각도만큼 돌려 줍니다. 코드는 다음과 같습니다.

```
import time
import CK_SPI_DEV as spi_dev

spi_dev.ck_spiWr(spi_dev.SPI_CMD_CLEAR, 0)

DELAY = 0.5

try :
   while(1) :
      for i in range(0, 181, 5) :
         print(i)
         spi_dev.ck_servoMot(i)
         time.sleep(DELAY)

except KeyboardInterrupt :
   spi_dev.ck_spiWr(spi_dev.SPI_CMD_CLEAR, 0)
```

위의 코드에서 for 문의 range() 함수를 보면 인덱스는 0 ~ 181 범위에서 5 씩 증가하면서 변합니다. 이것은 0 도에서 180 도까지 5 도 간격으로 변함을 뜻합니다. 이 각도를 ck_servoMot() 함수에 입력하여 해당 각도만큼 서보혼이 움직이게 합니다.

< 예제 코드 : **서보 모터 컨트롤 하기** >

< 코드 위치 : Codingkit / **smot.py >**

< 문법 설명 : **try 와 except** >

이 코드에는 try 와 except 가 있습니다. 이 구문은 예외 처리라고 하는 것인데요. try 로 묶인 코드에서 에러가 발생했을 때 그 해당 에러에 대한 처리를 어떻게 할지를 except 에 적어둔 것입니다. 다음과 같이 정의합니다.

```
try :
   문장1
   문장2
   ...
```

```
except 에러_타입 :
    예외_처리_문장
```

위에 정의한 파이썬 코드를 실행하면 먼저 try 문을 실행하기 때문에 문장1, 문장2, ... 를 실행합니다. 그런데 이 중간에 에러가 발생하면 except 문을 실행합니다. 물론 에러가 발생하지 않으면 except 문은 실행되지 않습니다. except 문에는 에러_타입이 있을 수도 있고 없을 수도 있습니다. 만약 에러_타입이 없다면 try 문 안에서 에러가 발생하면 무조건 except 문이 실행이 됩니다. 만약 에러_타입이 있다면 try 문 안에서 해당 에러_타입에 대한 에러가 발생했을 때만 except 문을 실행합니다. 정리하면 여러분은 에러가 발생할 수 있을 것 같은 코드를 try 문으로 묶어 둡니다. 그리고 그 try 문에서 에러가 발생했을 때 처리할 코드를 except 문에 둡니다.

그럼 위의 예제 코드를 볼까요?

```
try :
    while(1) :
        for i in range(0, 181, 5) :
            print(i)
            spi_dev.ck_servoMot(i)
            time.sleep(DELAY)

except KeyboardInterrupt :
    spi_dev.ck_spiWr(spi_dev.SPI_CMD_CLEAR, 0)
```

try 문은 while(1) 로 되어 있어 계속해서 수행이 됩니다. 그러던 중 에러가 발생합니다. 에러 중에서도 except 문에 있는 KeyboardInterrupt 에러가 발생하면 그 다음 문장을 수행합니다. 그럼 이 KeyboardInterrupt 란 무슨 에러일까요? 짐작하신 분들도 계실 것 같은데요. 이것은 터미널에서 파이썬 코드가 실행 중일 때 끝내기 위해서 Ctrl + C (컨트롤 버튼과 C 키를 같이 누름) 키를 누른 것입니다. 즉 Ctrl + C 를 누르면 KeyboardInterrupt 에러가 발생하여 except 문이 실행이 됩니다. except 문에는 "spi_dev.ck_spiWr(SPI_CMD_CLEAR, 0)" 로 SPI 이 인터페스에 있는 모든 디바이스를 초기화 하는 것입니다. 즉 이 문장이 수행되면 서보 모터도 동작을 끝내게 됩니다.

그래서 이 코드에서 try, except 문이 하는 일을 정리하여 말씀드리면 코드가 while(1) 문을 통해서 무한 반복적으로 실행이됩니다. 그러던 중에 Ctrl + C 를 누르면 서보 모터의 동작을 끝내고 파이썬 프로그램은 끝이 납니다. 그럼 try, except 없이 코딩을 하고 실행시킨 다음 Ctrl + C 로 파이썬 프로그램을 끝낸다면 어떤 일이 발생할까요? 실제 try, except 없는 코드를 실행시키고 Ctrl + C 로 프로그램을 끝내도 서보 모터는 더 이상 움직이지는 않습니다. 하지만 PWM 신호는 계속해서 서보 모터로 공급이 되고 있습니다. 그래서 다음과 같이 DC 모터를 동작시키는 코드를 실행해 보면 바로 알 수 있습니다. 다음 코드는 이전에 가변저항으로 DC 모터의 회전 속도를 제어했던 코드에 try, except 를 추가한 코드입니다.

```
try :
    while (1) :
        speed = spi_adc.ck_adcRd(spi_adc.ADC_CH_VR1)
        speed = ck_map(speed, 0, 1023, 0, 100)
        spi_dev.ck_pwmDCMot_fwd(1000, speed)
        print "SPEED : ", speed
        time.sleep(0.5)

except KeyboardInterrupt :
    spi_dev.ck_spiWr(spi_dev.SPI_CMD_CLEAR, 0)
```

실행시켜 보시면 Ctrl + C 로 프로그램을 끝내고 DC 모터가 잘 멈추는 것을 볼 수 있습니다. 그럼 다음과 같이 try, except 가 없는 코드를 실행시켜 보신 다음 Ctrl + C 로 프로그램을 멈추어 보십시요. DC 모터는 계속해서 같은 속도로 돌고 있을 것입니다. 그럼 이 코드를 어떻게 끝낼까요? 더 이상 가변저항을 돌리는 것도 말을 듣지 않습니다. 아~~ 어렵게 됐네요. 그럼 다시 try 와 except 가 있는 코드를 실행시키십시요. 그리고 Ctrl + C 를 눌러 끝내세요. 휴~~ 드디서 DC 모터가 멈췄네요. 수고하셨습니다.

< 예제 코드 : **try, except 를 이용한 코드 끝내기** >

< 코드 위치 : Codingkit / **dcmot_try_except.py** >

전에 버튼과 LED 를 연결하여 켜는 코드에서 한번 실행하고 나서 다시 실행하면 다음과 같은 경고(Warning)가 발생했었습니다.

```
pi@raspberrypi ~/Codingkit $ sudo python led_on_off_but.py
led_on_off_but.py:12: RuntimeWarning: This channel is already in use, continuing
 anyway.  Use GPIO.setwarnings(False) to disable warnings.
  GPIO.setup(LED, GPIO.OUT)
```

이 경고는 "그 채널은 (GPIO 는) 이미 사용중이다." 라는 경고입니다. 이 경고는 이전에 GPIO 를 사용하고 정리하지 않고 끝을 내서 그럽니다. 즉, 정리하는 코드 없이 Ctrl + C 로 끝을 낸 것입니다. 그럼 이 경고를 바로 잡아 보도록 하겠습니다. 먼저 이전에 했던 코드는 다음과 같습니다.

```
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BCM)

LED = 22
BUTTON = 4
BUTTON_PRESSED = 0
ON = 1
OFF = 0

GPIO.setup(LED, GPIO.OUT)
GPIO.setup(BUTTON, GPIO.IN)
```

```
while 1:
    if (GPIO.input(BUTTON) == BUTTON_PRESSED):
        GPIO.output(LED, ON)
    else:
        GPIO.output(LED, OFF)
```

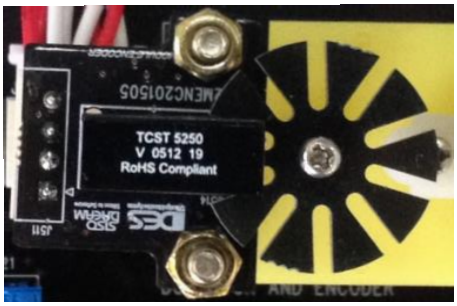
이 코드는 버튼을 누르면 LED 가 켜지는 코드입니다. 이 코드의 while 문을 감싸고 try, except 코드를 추가해 보겠습니다.

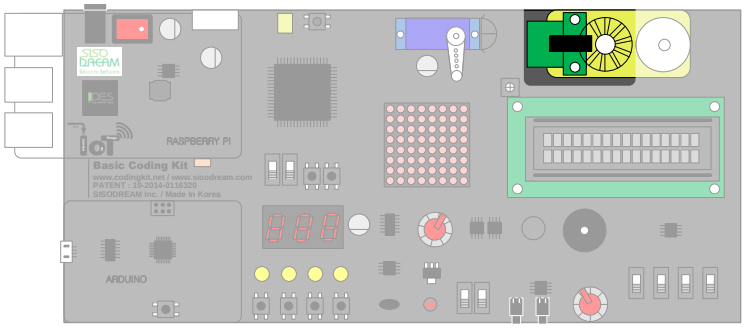
```
try :
    while 1:
        if (GPIO.input(BUTTON) == BUTTON_PRESSED):
            GPIO.output(LED, ON)
        else:
            GPIO.output(LED, OFF)

except KeyboardInterrupt :
    GPIO.cleanup()
```

except 문에 GPIO.cleanup() 함수를 추가하였습니다. 이 함수는 GPIO 로 사용하였던 핀들을 정리하는 함수입니다. 이제 이 코드를 실행시켜 보십시요. 더 이상 위에서 봤던 경고는 나오지 않습니다.

[DC 모터 인코더로 RPM 측정하기]

코딩 키트의 DC 모터에는 인코더 (Encoder) 가 붙어 있습니다.





이제 인코더에서 값을 입력 받아 보도록 하겠습니다. 인코더 하드웨어 관련 자세한 내용은 아두이노 파트를 참고해 주세요.

CK_SPI_DEV 모듈을 이용하면 인코더 휠이 한 바퀴 회전하는데 걸리는 시간을 얻을 수 있습니다. 이 시간을 얻기 위해서 다음 함수를 이용합니다.

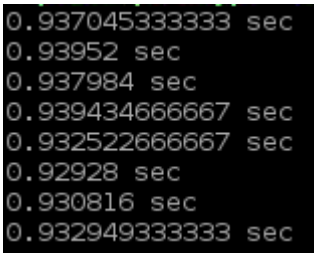
```
ck_getDCMotEncTime()
```

이 함수를 이용하여 DC 모터 인코더 휠이 한 바퀴 회전하는데 걸리는 시간을 얻을 수 있습니다. 이 시간을 변수 저장하여 출력하는 코드는 다음과 같습니다.

```
import RPi.GPIO as GPIO
import time
import CK_SPI_DEV as spi_dev

DCMOTENC_LED_ON   = 20

GPIO.setup(DCMOTENC_LED_ON, GPIO.OUT)
GPIO.output(DCMOTENC_LED_ON, 1)

DCMOT_EN = 17

GPIO.setup(DCMOT_EN, GPIO.OUT)
GPIO.output(DCMOT_EN, 1)

spi_dev.ck_pwmDCMot_fwd(1000, 100)

spi_dev.ck_spiWr(spi_dev.SPI_CMD_DCMOTENC_EN, 1)

try :
   while(1) :
      time.sleep(1)
      encData_sec = spi_dev.ck_getDCMotEncTime()
      print encData_sec, "sec"
```

```
except KeyboardInterrupt :
    spi_dev.ck_spiWr(spi_dev.SPI_CMD_CLEAR, 0)
    GPIO.cleanup()
```

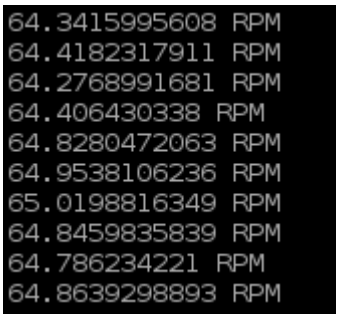
코드를 실행해 보면 다음과 같이 한 바퀴 도는데 걸리는 시간이 출력됩니다. 시간의 단위는 초입니다.



RPM 은 Revolutions Per Minute 라고 해서 1 분동안 몇 바퀴를 회전하는지를 숫자로 나타낸 것입니다. 그럼 1 분(60 초)을 위에서 구한 한 바퀴 도는데 걸리는 시간으로 나누면 RPM 이 나올 것입니다. 그래서 코드를 다음과 같이 살짝 바꾸어 주면 DC 모터의 RPM 을 출력할 수 있습니다

```
print (60/encData_sec), "RPM"
```

위와 같이 코드를 수정해 주고 실행을 시키면 다음과 같이 출력됩니다.



그런데 위의 코드에서는 한 가지 문제가 발생할 수 있습니다. 60/encData_sec 를 할 때 encData_sec 값이 0 일 수 있습니다. 파이썬에서는 0 으로 나누어지는 코드에서는 "ZeroDivisionError" 라는 에러를 발생합니다. 그래서 다음과 같이 예외 처리 코드를 추가해야 합니다.

```
try :
    while(1) :
        time.sleep(1)
        encData_sec = spi_dev.ck_getDCMotEncTime()
        try :
            print (60/encData_sec), "RPM"
        except ZeroDivisionError :
            print "Encoder Time is Zero."

except KeyboardInterrupt :
    spi_dev.ck_spiWr(spi_dev.SPI_CMD_CLEAR, 0)
    GPIO.cleanup()
```

예제 코드에서 다음과 같이 DCMOTENC_LED_ON 이라는 신호가 있습니다.

```
DCMOTENC_LED_ON   = 20

GPIO.setup(DCMOTENC_LED_ON, GPIO.OUT)
GPIO.output(DCMOTENC_LED_ON, 1)
```

아두이노 파트에서 인코더에는 적외선 센서가 있다고 했습니다. 적외선 센서는 발광부와 수광부가 있어서 발광부의 LED 를 켜 주면 그 값이 수광부로 들어 간다고 했습니다. 여기서 DCMOTENC_LED_ON 이라는 신호는 발광부의 LED 를 켜 주는 신호입니다. 아두이노 파트에서는 간단하게 DCMOTENC_EN 이라고 썼던 신호입니다.

< 예제 코드 : **DC 모터 인코더 휠의 한 바퀴 도는 시간 측정** >

< 코드 위치 : Codingkit / **dcmotenc.py >**

< 연습 문제 : **인코더를 이용하여 DC 모터의 속도 구하기** >

이번에는 가변저항으로 DC 모터의 속도를 제어하고 그 속도를 인코더를 이용하여 구해 보겠습니다. 속도는 RPM 으로 표시합니다.

다음과 같이 초기화 하는 부분이 있습니다.

```
#import

import RPi.GPIO as GPIO
import time
import CK_SPI_ADC as spi_adc
import CK_SPI_DEV as spi_dev

# Init DC Motor

DCMOT_EN = 17

GPIO.setup(DCMOT_EN, GPIO.OUT)
GPIO.output(DCMOT_EN, 1)

# Init DC Motor Encoder

DCMOTENC_LED_ON   = 20

GPIO.setup(DCMOTENC_LED_ON, GPIO.OUT)
GPIO.output(DCMOTENC_LED_ON, 1)
```

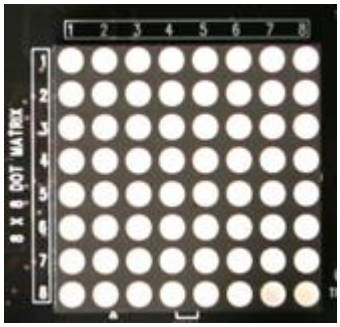
```
spi_dev.ck_spiWr(spi_dev.SPI_CMD_DCMOTENC_EN, 1)

# Map Function

def ck_map(x, in_min, in_max, out_min, out_max) :
    out_val = (((x - in_min) * (out_max - out_min)) / (in_max - in_min)) + out_min
    return out_val
```

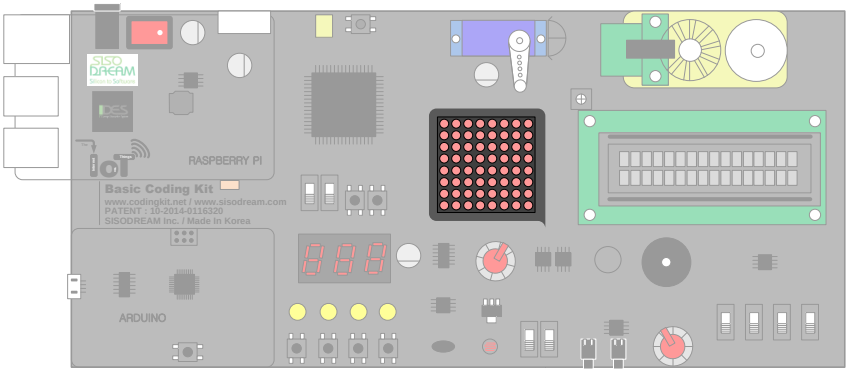
나머지 코드는 직접 작성해 보시기 바랍니다.

< 코드 위치 : Codingkit / **dcmotenc_speed.py >**

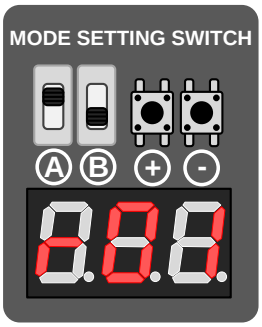
[도트매트릭스(Dotmatrix)에 하트 그리기]

도트매트릭스는 아래 사진과 같이 LED 를 여러 개 촘촘히 모아둔 모양입니다.

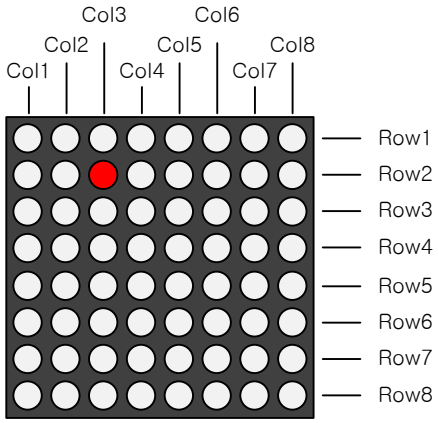




이 LED 를 하나 하나씩 별도로 켤 수 있기 때문에 원하는 문자나 그림을 표시할 수 있습니다. 라즈베리파이와 도트매트릭스의 연결은 스위칭 ID r01 입니다. 스위치와 버튼을 이용하여 스위칭 ID 를 r01 로 바꾸겠습니다.



아두이노 파트에서 도트메트릭스는 LED 하나 하나를 컨트롤하는 것은 아니고 가로줄 8 개와 세로줄 8 개로 컨트롤한다고 배웠습니다.



그래서 위 그림의 붉은색 LED 는 Col3 번과, Row2 번을 켜 주면 된다고 배웠습니다.

다음은 라즈베리파이에서 도트메트릭스에 연결된 핀을 정의한 것입니다.

```
DM_ROW_1 = 3
DM_ROW_2 = 19
DM_ROW_3 = 22
DM_ROW_4 = 18
DM_ROW_5 = 16
DM_ROW_6 = 5
DM_ROW_7 = 24
DM_ROW_8 = 25

DM_COL_1 = 2
DM_COL_2 = 23
DM_COL_3 = 6
DM_COL_4 = 4
DM_COL_5 = 21
DM_COL_6 = 15
DM_COL_7 = 26
DM_COL_8 = 27
```

그리고 이것은 리스트로 묶습니다. 그러면 for 문을 이용하여 쉽게 컨트롤할 수 있습니다.

```
dmRow = [DM_ROW_1, DM_ROW_2, DM_ROW_3, DM_ROW_4,
         DM_ROW_5, DM_ROW_6, DM_ROW_7, DM_ROW_8]
dmCol = [DM_COL_1, DM_COL_2, DM_COL_3, DM_COL_4,
         DM_COL_5, DM_COL_6, DM_COL_7, DM_COL_8]
```

다음은 for 문을 이용하여 GPIO 를 출력으로 설정합니다.

```
for x in dmRow:
  GPIO.setup(x, GPIO.OUT)
for y in dmCol:
  GPIO.setup(y, GPIO.OUT)
```

다음과 같이 for 문을 이용하여 도트매트릭스를 끄는 함수를 코딩합니다.

```
def dmOff():
  for x in dmRow:
    GPIO.output(x, 0)
  for y in dmCol:
    GPIO.output(y, 1)
```

다음과 같이 for 문을 이용하여 도트매트릭스를 켜는 함수를 코딩합니다.

```
def dmOn():
  for x in dmRow:
    GPIO.output(x, 1)
  for y in dmCol:
    GPIO.output(y, 0)
```

위의 함수들을 이용하여 도트매트릭스 전체를 1 초 간격으로 켜고 끄는 함수를 다음과 같이 구현합니다.

```
Delay = 1

try :
    while 1 :
        dmOn()
        time.sleep(Delay)
        dmOff()
        time.sleep(Delay)

except KeyboardInterrupt :
    GPIO.cleanup()
```

이제 이 코드를 코딩 키트에서 실행시켜 봅니다. 그러면 도트매트릭스가 1 초 간격으로 깜박이는 것을 확인할 수 있습니다.

< 예제 코드 : **도트매트릭스 깜박이기** >

< 코드 위치 : Codingkit / **dotmatrix_on_off.py >**

이 코드를 실행시키면 다음과 같은 경고 메시지가 터미널에 출력됩니다.

```
pi@raspberrypi ~/Codingkit $ sudo python dotmatrix_on_off.py
dotmatrix_on_off.py:31: RuntimeWarning: A physical pull up resistor is fitted on this channel!
  GPIO.setup(x, GPIO.OUT)
dotmatrix_on_off.py:33: RuntimeWarning: A physical pull up resistor is fitted on this channel!
  GPIO.setup(y, GPIO.OUT)
```

이것은 도트매트릭스 핀 번호 2, 3 번에 풀업(Pull-up) 저항이 연결되어 있다는 뜻입니다. 위의 메시지를 보고 어떻게 핀 번호 2, 3 번임을 알았을까요? 그것은 위의 메시지 중 "dot-matrix_on_off:31:" 에서 31 이 코드의 줄번호 입니다. 그래서 실제 코드를 보면 다음과 같이 줄번호 31 번에 "GPIO.setup(x, GPIO.OUT)" 코드가 있습니다. 33 번에는 "GPIO.setup(y, GPIO.OUT)" 코드가 있습니다.

```
30 for x in dmRow:
31     GPIO.setup(x, GPIO.OUT)
32 for y in dmCol:
33     GPIO.setup(y, GPIO.OUT)
34
```

위의 코드에서 x, y 는 dmRow 와 dmCol 리스트를 사용하였습니다. 이 dmRow 와 dmCol 의 원소에는 핀 번호 2, 3 번이 포함되어 있습니다. 라즈베리파이의 핀 2, 3 번은 풀업 저항이 연결되어 있습니다. 그래서 그 부분에 대한 경고 메시지인 것입니다.

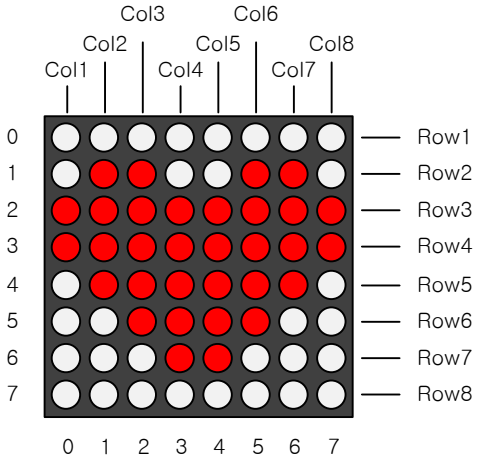
이 풀업 저항에 대해서는 코딩 키트 하드웨어 파트에서 자세히 다루겠습니다.

이번에는 도트매트릭스에 하트를 그려 보도록 하겠습니다. 다음과 같이 하트 모양이 되도록 각 줄을 정의

해 줍니다.

```
heart_col_0 = [1, 1, 1, 1, 1, 1, 1, 1]
heart_col_1 = [1, 0, 0, 1, 1, 0, 0, 1]
heart_col_2 = [0, 0, 0, 0, 0, 0, 0, 0]
heart_col_3 = [0, 0, 0, 0, 0, 0, 0, 0]
heart_col_4 = [1, 0, 0, 0, 0, 0, 0, 1]
heart_col_5 = [1, 1, 0, 0, 0, 0, 1, 1]
heart_col_6 = [1, 1, 1, 0, 0, 1, 1, 1]
heart_col_7 = [1, 1, 1, 1, 1, 1, 1, 1]
```

도트매트릭스는 다음 그림과 같이 가로로 8 줄이 있습니다. 이 각각의 줄에 있는 8 개의 LED 를 어떻게 켤지를 위의 코드와 같이 8 개의 리스트에 정의해 둔 것입니다. 켜는 값이 0 입니다.



이 값을 다음과 같이 각각의 핀에 할당합니다.

```
try :
    while (1) :
        for idx_row in range(8) :
            dmOff()
            GPIO.output(dmRow[idx_row], 1);

            if (idx_row == 0) :
                for idx_col in range(8) :
                    GPIO.output(dmCol[idx_col], heart_col_0[idx_col]);
            elif (idx_row == 1) :
                for idx_col in range(8) :
                    GPIO.output(dmCol[idx_col], heart_col_1[idx_col]);
            elif (idx_row == 2) :
                for idx_col in range(8) :
                    GPIO.output(dmCol[idx_col], heart_col_2[idx_col]);
            elif (idx_row == 3) :
                for idx_col in range(8) :
                    GPIO.output(dmCol[idx_col], heart_col_3[idx_col]);
            elif (idx_row == 4) :
                for idx_col in range(8) :
```

```
            GPIO.output(dmCol[idx_col], heart_col_4[idx_col]);
        elif (idx_row == 5) :
            for idx_col in range(8) :
                GPIO.output(dmCol[idx_col], heart_col_5[idx_col]);
        elif (idx_row == 6) :
            for idx_col in range(8) :
                GPIO.output(dmCol[idx_col], heart_col_6[idx_col]);
        elif (idx_row == 7) :
            for idx_col in range(8) :
                GPIO.output(dmCol[idx_col], heart_col_7[idx_col]);
        elif (idx_row == 8) :
            for idx_col in range(8) :
                GPIO.output(dmCol[idx_col], heart_col_8[idx_col]);

        time.sleep(Delay)

    except KeyboardInterrupt :
        GPIO.cleanup()
```

for 문을 이용하여 idx_row 를 0 ~ 7 이 되도록 합니다. 이것은 현재 0 ~ 7 번 줄 중 어떤 줄이 켜질지를 결정하는 것입니다. 일단 dmOff() 함수를 이용하여 도트매트릭스를 전부 끕니다. 그리고 0 ~ 7 번 중 해당 줄을 켭니다. 이렇게 켜진 줄의 8 개의 LED 중 어떤 것이 켜지고 꺼질지에 대해서는 미리 리스트로 정의해 두었습니다. 그 리스트는 heart_col_0 ~ hear_col_7 입니다. 이 값들은 다음과 같은 for 문을 이용하여 각 핀에 할당합니다.

```
for idx_col in range(8) :
    GPIO.output(dmCol[idx_col], heart_col_0[idx_col]);
```

LED 를 켜는 8 개의 리스트 중 어떤 리스트를 사용할지는 idx_row 로 결정을 합니다. 이 값에 따라서 heart_col_0 ~ hear_col_7 중 하나를 선택합니다.

이제 이 코드를 코딩 키트에서 실행시켜 봅니다.

< 예제 코드 : **도트매트릭스에 하트 그리기** >

< 코드 위치 : Codingkit / **dotmatrix_heart.py >**

코드가 좀 길지요. 이 코드를 리스트 안에 리스트를 넣는 2 차원 리스트를 이용하여 확 줄여 보겠습니다. 이번에는 다음과 같이 리스트를 정의합니다.

```
heart_col = [ [1, 1, 1, 1, 1, 1, 1, 1],
              [1, 0, 0, 1, 1, 0, 0, 1],
              [0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0],
              [1, 0, 0, 0, 0, 0, 0, 1],
```

```
[1, 1, 0, 0, 0, 0, 1, 1],
[1, 1, 1, 0, 0, 1, 1, 1],
[1, 1, 1, 1, 1, 1, 1, 1] ]
```

리스트 안에 리스트는 넣는 방법은 간단합니다. 대괄호를 이용하여 리스트를 만들고 이 리스트를 쉼표로 구분하여 리스트 안에 하나의 원소로 넣으면 됩니다. 이렇게 2 차원 리스트는 대괄호 2 개를 이용하여 값을 얻습니다. 즉, heart_col[2][3] 하면 이 값은 2 번줄의 3 번째 값입니다. 위의 코드에서는 붉은색으로 표시된 값입니다. 이 2 차원 리스트를 이용하면 위 코드의 많은 for 문은 다음 코드와 같이 짧게 줄일 수 있습니다.

```
for idx_row in range(8) :
    dmOff()
    GPIO.output(dmRow[idx_row], 1);

    for idx_col in range(8) :
        GPIO.output(dmCol[idx_col], heart_col[idx_row][idx_col]);

    time.sleep(Delay)
```

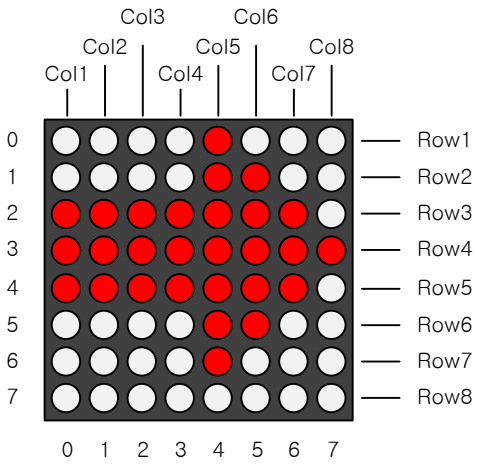
첫번째 for 문은 가로줄을 선택하는 for 문입니다. 그래서 일단 dmOff() 함수를 이용하여 모두 끈 이후에 GPIO.output(dmRow[idx_row], 1) 를 이용하여 해당 줄을 켜 줍니다. 그리고 두번째 for 문으로 2 차원 리스트의 값을 그 줄에 있는 8 개의 LED 에 할당합니다. 그리고 0.002 초의 Delay 를 주어 한 줄씩 계속 켜지도록 합니다. 그러면 하트 모양이 켜지는 것처럼 보입니다. 여기서 Delay 는 1 로 정의해 두었습니다.

< 예제 코드 : **2 차원 리스트를 이용하여 도트매트릭스에 하트 그리기** >

< 코드 위치 : Codingkit / **dotmatrix_heart_list_2.py** >

< 연습 문제 : **도트매트릭스에 화살표 그리기** >

도트매트릭스에 다음과 같은 화살표 모양을 그리는 연습 문제를 해 보겠습니다. 어렵지 않고 매우 쉽습니다.

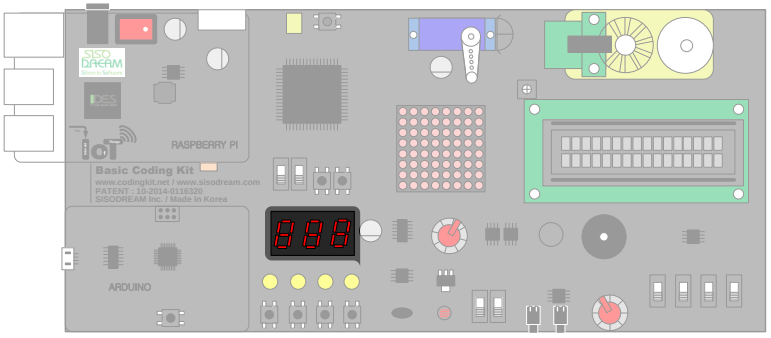


< 코드 위치 : Codingkit / **dotmatrix_arrow.py >**

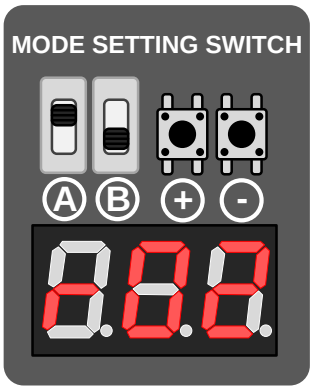
도트매트릭스에 하트를 그리는 예제와 화살표를 그리는 예제를 실행시키고 도트매트릭스를 잘 관찰하면 도트매트릭스가 간혹 빠르게 깜박이는 것을 볼 수 있을 것입니다. 이것은 라즈베리파이에 리눅스라는 복잡한 운영체제가 동작하고 있어서 모든 코드를 끊김 없이 처리해 줄 수 없기 때문에 생기는 문제입니다. 이 라즈베리파이에서는 실제로 많은 프로그램이 동작하고 있습니다. 기본적으로 리눅스 운영체제가 수행되고 있고 여기에 여러분이 파이썬으로 작성한 프로그램 이외에도 다른 여러가지 프로그램이 수행되고 있습니다. 이 여러가지 프로그램들은 아주 짧은 시간동안 수행하고 다른 프로그램이 수행할 수 있도록 자신은 잠시 쉬어줍니다. 그래서 여러분의 파이썬 프로그램도 수행하다가 다른 프로그램이 수행될 수 있도록 잠시 쉬는 시간이 있습니다. 이렇게 프로그램이 수행되는 동안 잠시 쉬는 시간이 끼어들다 보니 여러분이 작성한 도트매트릭스 프로그램이 수행되지 못하는 일이 생기는 것입니다. 그래서 도트매트릭스가 간혹 깜박이게 되는 것입니다. 이렇게 어떤 프로그램이 수행하고 쉬고 하는 것을 결정해는 주는 것은 리눅스 운영체제가 해 주는 것입니다. 그래서 앞으로 공부하게 될 세븐세그먼트, 캐릭터 LCD 에서도 간혹 깜박이는 것을 볼 수 있을 것입니다. 그래서 실제로 산업 현장에서는 이런 디바이스를 다눌 수 있는 별도의 칩을 라즈베리파이와 디바이스 사이에 장착을 합니다. 그렇게 되면 라즈베리파이에서는 어떤 모양을 그리라고 별도의 칩에 명령을 보내면 그 칩에서 디바이에서 명령 받은 모양을 그릴 수 있도록 다시 컨트롤해 주는 것입니다. 여기서 별도의 칩을 아두이노로 사용할 수도 있습니다. 그래서 코딩 키트의 고급 과정에서는 라즈베리파이에서 아두이노로 도트매트릭스에 어떤 모양을 그리라는 명령을 내리면 아두이노에서는 그 명령을 받아 도트매트릭스에 어떤 모양을 그리는 그런 예제를 실습해 볼 것입니다. 그런 실습은 조금 어렵습니다. 그런 조금 어려운 실습도 잘 할 수 있도록 지금 열심히 공부해 두십시요.

[세븐세그먼트에 숫자 표시 하기]

이번에는 세븐세그먼트(7-Segmnet)를 다루어 보겠습니다. 세븐세그먼트는 FND 라고 부르기도 하지요.



세븐세그먼트의 스위칭 ID 는 r02 입니다.

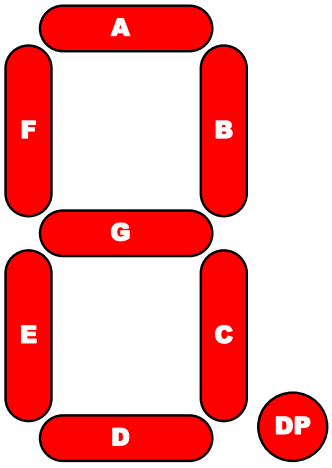


+ 혹은 - 버튼을 눌러서 스위칭 ID 를 고정하고 나면 FND 는 모두 꺼지거나 이상한 글자가 표시될 것입니다. 그것은 앞으로 여러분이 코딩하여 FND 를 컨트롤할 것이기 때문에 일단 FND 를 꺼지게 하거나 아니면 이전에 코딩된 부분이 적용되는 것입니다.

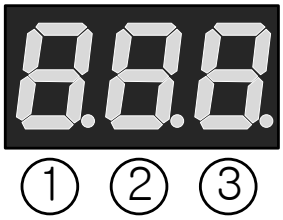
FND 를 모두 켰다 끄기를 반복하는 예제를 해 보도록 하겠습니다. r02 모드에서 FND 의 핀 번호는 다음과 같이 정의합니다.

```
FND_A    = 2
FND_B    = 3
FND_C    = 4
FND_D    = 5
FND_E    = 16
FND_F    = 12
FND_G    = 18
FND_DP   = 19
FND_SEL_1 = 14
FND_SEL_2 = 21
FND_SEL_3 = 22
```

위의 FND_A ~ FND_DP 는 다음 그림과 같이 각각의 FND 의 A 부터 DP 까지의 세그먼트에 연결이 됩니다.



코딩 키트에는 3 개의 FND 가 있습니다. 이 각각의 FND 는 FND_SEL_1, FND_SEL_2, FND_SEL_3 에 연결이 됩니다. 각각은 다음과 같이 할당이 됩니다.



이렇게 정의된 핀들의 GPIO 설정을 위해서 리스트로 묶어 주고 for 문을 이용하여 출력으로 설정해 줍니다.

```
FND_SEG = [FND_A, FND_B, FND_C, FND_D, FND_E, FND_F, FND_G, FND_DP]
FND_SEL = [FND_SEL_1, FND_SEL_2, FND_SEL_3]

for fndSeg in FND_SEG :
   GPIO.setup(fndSeg, GPIO.OUT)

for fndSel in FND_SEL :
   GPIO.setup(fndSel, GPIO.OUT)
```

FND 를 켜려면 세그먼트와 SEL 신호 모두에 0 을 주어야 합니다. 그래서 FND 를 켜는 함수는 다음과 같습니다.

```
FND_SEG_ON = 0
FND_SEL_ON = 0

def fndOn() :
   for fndSeg in FND_SEG :
      GPIO.output(fndSeg, FND_SEG_ON)
```

```
    for fndSel in FND_SEL :
      GPIO.output(fndSel, FND_SEL_ON)
```

이와는 반대로 끄는 함수는 FND_SEG_ON 과 FND_SEL_ON 변수에 not 연산자를 취하고 할당해 주면 됩니다.

```
def fndOff() :
  for fndSeg in FND_SEG :
    GPIO.output(fndSeg, not(FND_SEG_ON))
  for fndSel in FND_SEL :
    GPIO.output(fndSel, not(FND_SEL_ON))
```

이 함수들을 이용하여 1 초 간격으로 켜고 끄는 코드는 다음과 같습니다.

```
DELAY = 1

try :
  while(1) :
    fndOn()
    time.sleep(DELAY)
    fndOff()
    time.sleep(DELAY)

except KeyboardInterrupt:
  GPIO.cleanup()
```

이제 코딩 키트에서 실행해 봅니다. 그러면 FND 전체가 깜박일 것입니다.

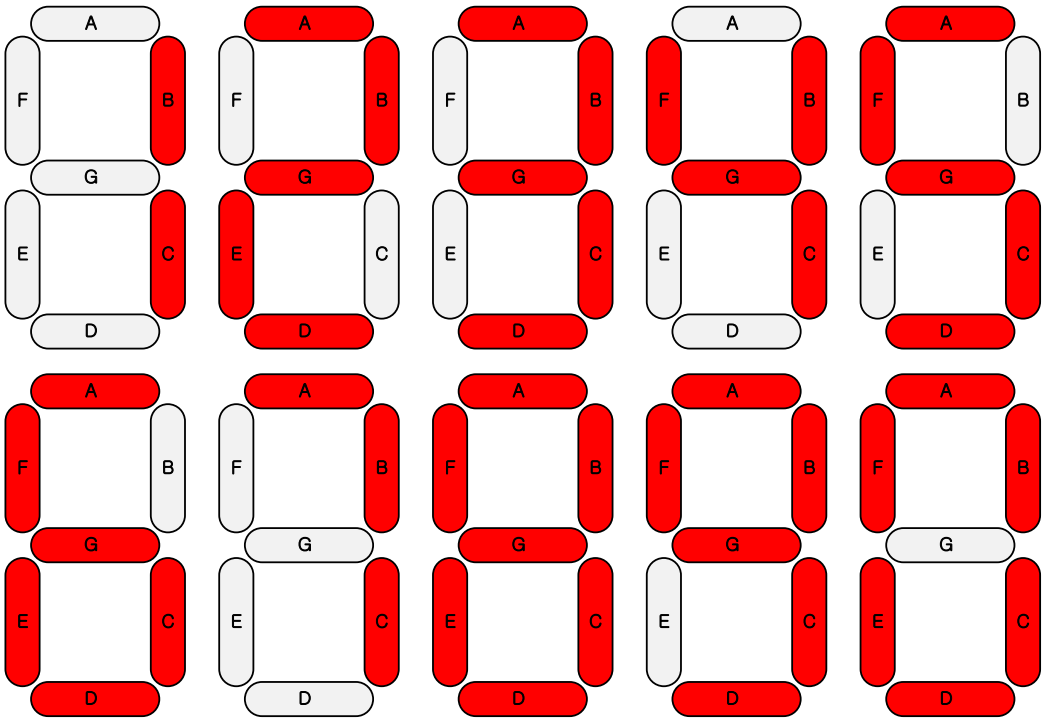
< 예제 코드 : **세븐세그먼트 깜박이기** >

< 코드 위치 : Codingkit / **fnd_on_off.py** >

이번에는 가변 저항 값을 FND 에 표시하는 예제를 해 보겠습니다. 가변 저항을 사용하기 위해서는 CK_SPI_ADC 모듈을 이용합니다.

import CK_SPI_ADC as spi_adc

가변 저항 값을 FND 에 표시하려면 0 ~ 9 까지 숫자를 FND 에 표시해야 합니다. 숫자는 다음 그림과 같이 표시합니다.



이 숫자들을 표시하기 위해서 다음과 같이 2 차원 리스트를 정의하였습니다.

```
fnd_num = [ [0, 0, 0, 0, 0, 0, 1, 1],
            [1, 0, 0, 1, 1, 1, 1, 1],
            [0, 0, 1, 0, 0, 1, 0, 1],
            [0, 0, 0, 0, 1, 1, 0, 1],
            [1, 0, 0, 1, 1, 0, 0, 1],
            [0, 1, 0, 0, 1, 0, 0, 1],
            [0, 1, 0, 0, 0, 0, 0, 1],
            [0, 0, 0, 1, 1, 1, 1, 1],
            [0, 0, 0, 0, 0, 0, 0, 1],
            [0, 0, 0, 0, 1, 0, 0, 1] ]
```

각 원소는 숫자 0 ~ 9 를 FND 에 표시하는 값입니다.

가변저항 값은 다음과 같이 읽어 들입니다.

```
vr_num = spi_adc.ck_adcRd(spi_adc.ADC_CH_VR1)
```

3 개의 FND 에 최대한 표시할 수 있는 숫자는 999 입니다. 그래서 가변저항 값이 999 보다 큰 경우에는 모두 다 999 가 되도록하는 코드를 작성합니다.

```
if (vr_num > 999) :
        vr_num = 999;
```

가변저항 값에서 100 의 자리, 10 의 자리, 1 의 자리 값을 구하여 3 개의 FND 에 표시합니다.

```
fnd1_num = vr_num / 100
    fnd2_num = (vr_num % 100) / 10
```

```
fnd3_num = vr_num % 10
```

100 의 자리는 가변저항 값을 100 으로 나눈 몫만을 취합니다. 10 의 자리는 가변 저항 값을 100 으로 나눈 나머지 값만을 취합니다. 1 의 자리는 가변저항 값을 10 으로 나눈 나머지 값입니다.

이렇게 구한 각 자릿수 값을 각각의 FND 에 표시합니다. 다음은 1 번 FND 에 100 의 자리 값을 표시하는 코드입니다.

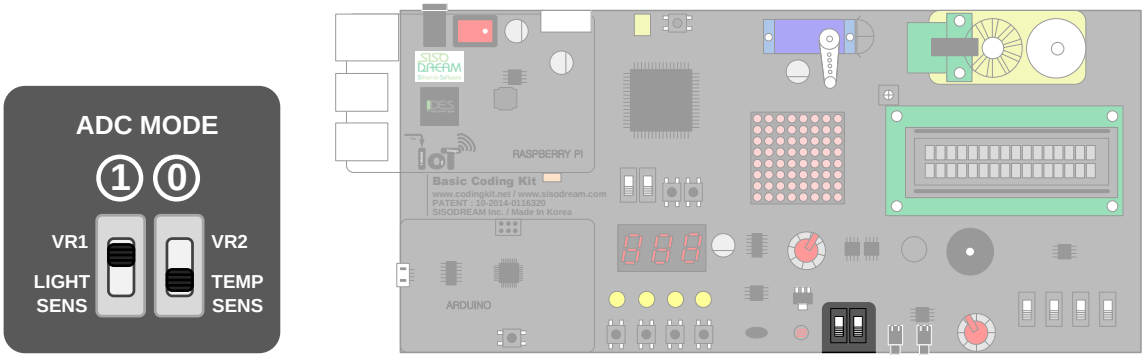
```
fndOff()
    GPIO.output(FND_SEL_1, FND_SEL_ON)
    for i in range(8) :
        GPIO.output(FND_SEG[i], fnd_num[fnd1_num][i])
    time.sleep(DELAY)
```

위 코드들을 잘 정리하여 전체 코드를 완성해 보세요. 그것도 좋은 공부가 될 것입니다.

< 예제 코드 : **세븐세그먼트에 가변저항 값 표시하기** >

< 코드 위치 : Codingkit / **fnd_vr.py** >

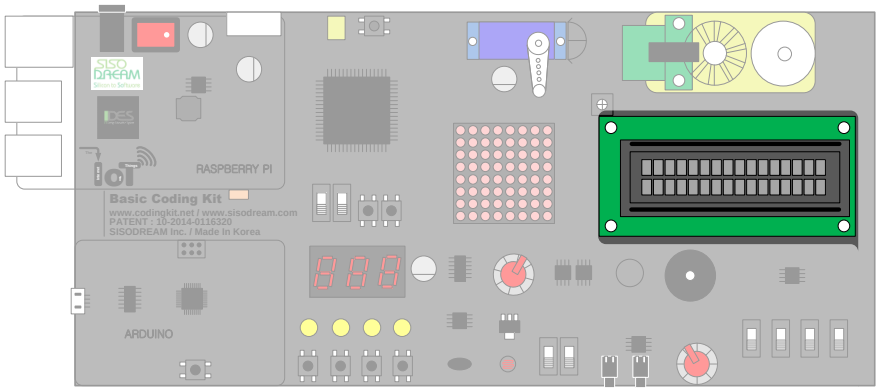
코드를 완성하였다면 코딩 키트에서 실행해 보십시요. 실행시키기 전에 ADC 모드 스위치 중 1 번은 위로 올려 주세요.



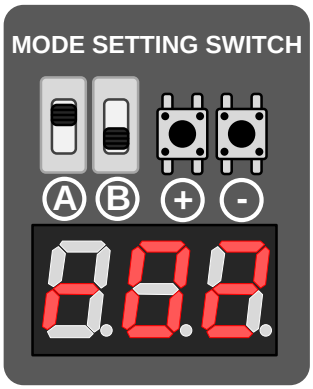
가변저항을 돌리면서 FND 에 값이 잘 표시되는지 확인하세요.

[Character LCD 에 문자 쓰기]

캐릭터 LCD(Character Liquid Crystal Display)라는 것은 문자를 출력하기 알맞게 제작된 LCD 입니다. 코딩키트의 캐릭터 LCD 는 16 개의 문자를 2 줄에 출력할 수 있는 캐릭터 LCD 가 장착되어 있습니다. 보통 16x2 캐릭터 LCD 라고 합니다.



라즈베리파이를 통해 캐릭터 LCD를 사용하려면 스위칭 ID 를 r02 로 맞추어야 합니다.



캐릭터 LCD 를 아두이노 파트에서는 LiquidCrystal.h 라는 라이브러리를 이용하여 코딩하였습니다. 하지만 라즈베리파이에서는 적당한 모듈이 없어 GPIO 를 이용하여 바로 코딩하겠습니다.

LCD 핀들은 다음과 같습니다.

핀 이름	용도
LCD_EN	LCD Eanble
LCD_RS	LCD 의 설정 신호 표시
LCD_D0 ~ 3	LCD 데이터 0 ~ 3
LCD_B_LIGHT	LCD Back Light

이 핀들을 다음과 같이 정의하였습니다.

```
LCD_EN      = 6
LCD_RS      = 15
LCD_D0      = 23
LCD_D1      = 24
LCD_D2      = 25
LCD_D3      = 26
LCD_B_LIGHT = 27

LCD_DATA = [LCD_D0, LCD_D1, LCD_D2, LCD_D3]
```

이렇게 정의된 핀들의 모드를 설정합니다.

```
# Pin Mode Setting
GPIO.setup(LCD_EN, GPIO.OUT)
GPIO.setup(LCD_RS, GPIO.OUT)
for data in LCD_DATA :
   GPIO.setup(data, GPIO.OUT)
GPIO.setup(LCD_B_LIGHT, GPIO.OUT)
```

LCD 백라이트(Back-light)는 바로 켜 줍니다. 그러면 코딩 키트의 LCD 에 파란색 백라이트가 켜집니다.

```
GPIO.output(LCD_B_LIGHT, 1)
```

LCD 에는 커맨드와 데이터를 전달하여 컨트롤합니다. 대표적인 커맨드 몇 개를 설명드리면 커맨드가 무엇인지를 알 수 있을 것입니다.

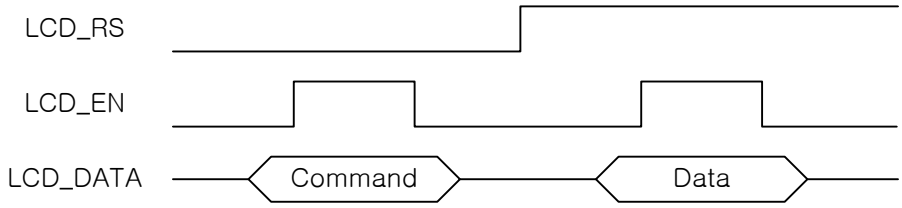
- LCD_CMD_CLEAR : LCD 에서 이 커맨드를 받으면 LCD 에 표시된 모든 문자를 지우고 커서를 좌측 상단으로 이동시킵니다.

- LCD_CMD_SET_CURSOR : LCD 에서 이 커맨드를 받으면 커서를 해당 위치로 옮깁니다.

LCD 에 전달되는 데이터의 문자의 아스키(ASCII - American Standard Code for Information Interchange, 미국정보교환 표준부호) 코드 값 입니다. 이 아스키 코드라는 것은 컴퓨터와 통신 장비 등 문자를 사용하는 장치에서 문자를 표현하기 위한 숫자 표현을 말합니다. 예를 들어 컴퓨터에서는 문자를 그대로 인식할 수 없으므로 a 는 97번, b 는 98 번 과 같은 식으로 문자나 기호에 번호를 붙인 것입니다. 표준화 되어 있

는 코드이므로 어떤 장치나 프로그램을 통해서도 공통적으로 사용할 수 있습니다. 캐릭터 LCD 에서는 이 아스키 코드 값을 받으면 해당 문자를 LCD 에 표시합니다. 즉, 데이터로 97 을 받으면 LCD 에 a 를 표시하는 것입니다.

그럼 이 커맨드나 데이터는 어떻게 전달하는지 알아 보겠습니다. 다음 파형과 같이 전달이 됩니다.

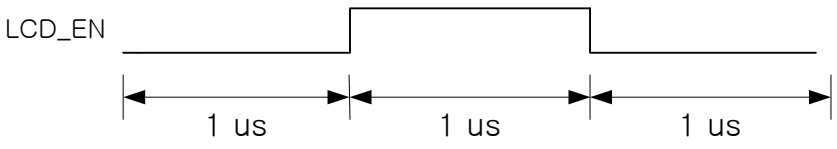


LCD_RS 신호는 현재 LCD_DATA 핀으로 전달되는 신호가 커맨드인지 데이터인지를 구분해 줍니다. LCD_RS 가 0 이면 커맨드이고 1 이면 데이터입니다. 그리고 커맨드와 데이터는 LCD_EN 신호가 1 일 때만 유효합니다. 그래서 LCD_DATA 핀에 커맨드 혹은 데이터를 실은 뒤에 LCD_EN 신호를 0 에서 1 로 주었다가 다시 0 으로 떨어뜨립니다. 이렇게 잡은 시간 동안 1 이 되고 나머지는 0 으로 유지되는 것을 펄스(Pulse) 라고 합니다.

그래서 커맨드나 데이터를 LCD 로 전달하는 코드는 다음과 같습니다.

```
D_MS = 0.001        # Delay : 1 Milisecond
D_US = 0.000001     # Delay : 1 Microsencod

def enablePluse() :
    GPIO.output(LCD_EN, 0)
    sleep(D_US)
    GPIO.output(LCD_EN, 1)
    sleep(D_US)
    GPIO.output(LCD_EN, 0)
    sleep(D_US)
```

먼저 위와 같이 펄스를 만들어주는 함수를 작성합니다. 이 함수는 다음 그림과 같은 펄스를 만들어 줍니다.



LCD 는 데이터 핀으로 4 핀을 사용합니다. 그래서 8 비트의 데이터를 전달하기 위해서는 2 번에 나누어 전달합니다. 먼저 4 ~ 7 번 데이터를 전달하고 다음으로 0 ~ 3 번 데이터를 전달합니다. 그럼 이렇게 나누어진 4 비트를 전달하는 코드는 다음과 같습니다.

```
def wrData_4bits(data4) :
```

```
for i in range(4) :
    if (((data4 >> i) & 0x1) == 1) :
        GPIO.output(LCD_DATA[i], 1)
    else :
        GPIO.output(LCD_DATA[i], 0)
enablePluse()
```

위의 코드에서 "(data4 >> i) & 0x1" 은 4 비트를 i 값만큼 오른쪽으로 쉬트프한 이후에 0x1 과 AND 비트 연산을 한 것입니다. 이렇게 하면 data4 의 0 번 비트부터 하나씩 취할 수 있습니다. 위의 코드의 i 값에 0 ~ 3 까지를 대입해서 계산해 보면 0 번 비트부터 하나씩 취한다는 것을 알 수 있을 것입니다. 비트 연산은 아두이노의 C 언어와 같습니다.

이렇게 하여 4 비트를 LCD 에 전달하는 함수를 만들었습니다. 다음은 이 함수를 이용하여 8 비트를 전달하는 함수입니다.

```
def wrData_8bits(data8, data_en = 0) :
    GPIO.output(LCD_RS, data_en)
    wrData_4bits(data8 >> 4)
    wrData_4bits(data8)
```

위의 함수의 첫번째 인자는 data8 이라는 8 비트 데이터입니다. 두번째 인자는 지금 전달하려고 하는 8 비트의 데이터가 커맨드인지 데이터인지를 구분하는 변수입니다. 이 값이 0 이면 커맨드이고 1 이면 데이터 입니다. 함수의 정의의 매개 변수에서 "data_en = 0" 이라고 값을 할당하면 이 data_en 의 초기값은 0 인 것입니다. 그리고 이 함수를 호출할 때 data_en 인자를 쓰지 않으면 그 값은 자동으로 0 이 됩니다. 이것은 이 함수를 호출하는 부분에서 다시 한 번 더 보겠습니다.

LCD_RS 신호는 커맨드와 데이터를 구분하는 신호라고 했습니다. 그래서 data_en 이 0 이면 커맨드이고 1 이면 데이터인 것입니다. 이 data_en 값을 그대로 LCD_RS 신호 값으로 사용합니다. 다음은 data8 을 오른쪽으로 4 비트 쉬프트하여 4 ~ 7 번 비트를 wrData_4bits() 함수로 전달합니다. 다음은 data8 을 그대로 전달하여 0 ~ 3 번 비트를 wrData_4bits() 함수로 전달합니다.

다음은 LCD 를 초기화하는 함수입니다. 이 함수를 이용하여 LCD 를 초기화해 줍니다.

```
def initLCD() :
    # Reset LCD
    sleep(15*D_MS)     # Wait Time > 15 ms
    wrData_4bits(0x3)
    sleep(5*D_MS)      # Wait Time > 4.1 ms
    wrData_4bits(0x3)
    sleep(200*D_US)  # Wait Time > 100 us
    wrData_8bits(0x32)
    # Config LCD
    wrData_8bits(0x28)  # Mode : 4-bit, Line : 2, Matrix : 5x7
    wrData_8bits(0x0C)  # Display : On, Cursor : Off, Cursor Blink : Off
    wrData_8bits(0x06)  # Cursor Shift : Right
```

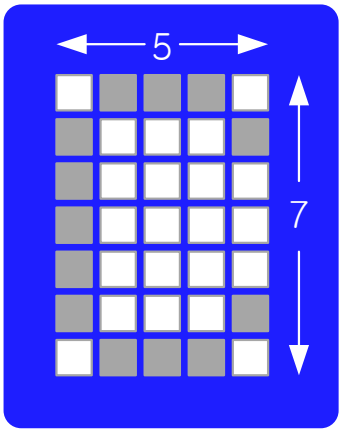
clearLCD()

위의 붉은색 부분은 LCD 를 내부적으로 리셋해 주어 LCD 의 상태를 초기화해 주는 코드입니다. 중간에 삽입된 sleep 시간은 캐릭터 LCD의 설명서를 참고하여 설정 된 것입니다. 이 시간이 제대로 지켜지지 않는 경우 LCD가 정상적인 동작을 하지 못 할 수도 있습니다. 그 다음 역시 LCD의 동작 설정의 용도로 LCD 설명서에서 미리 정의된 설정코드 입니다. 이 코드들은 LCD 를 다음과 같이 설정해 줍니다.

- 4 비트 모드 (Mode : 4-bit) : LCD 는 4 비트 혹은 8 비트의 데이터 신호를 사용할 수 있습니다. 실제적으로 4 핀 혹은 8 핀을 사용한다는 뜻입니다. 그런데 여기서는 4 비트 신호를 사용하겠다고 설정합니다.

- 문자(Character) 표시 줄 수 (Line : 2) : LCD 에 문자를 표시할 수 있는 줄 수가 몇 줄인지를 의미합니다. 여기서는 2 줄을 사용합니다.

- 한 문자 표시 매트릭스 (Matrix : 5x7) : 한 문자를 표시할 때 다음과 같이 5x7 만큼의 점을 사용한다는 뜻입니다.



- 디스플레이 (Display : On) : LCD 를 켜거나 꺼 주는 설정입니다.

- 커서 표시 (Cursor : Off) : 이것은 커서를 표시할지 안 할지에 대한 설정입니다. 여기서는 하지 않겠습니다.

- 커서 깜박임 (Cursor Blink : Off) : 만약 커서가 표시된다면 깜박이게 한다는 뜻입니다. 여기서는 커서를 표시하지 않으므로 깜박이지도 않습니다.

- 커서가 움직이는 방향 (Cursor Shift : Right) : 커서가 움직이는 방향을 설정합니다. 여기서는 오른쪽으로 움직이겠다는 뜻입니다. 커서를 표시하지도 않는데 무슨 커서를 움직이는 방향을 설정하나" 라고 의문을 가지실 수도 있는데요. 여기서 말하는 커서는 현재 문자가 표시될 위치를 말하는 것입니다. 그리고 커서를 오른쪽으로 움직이라는 뜻은 현재 문자를 표시 했으면 다음 문자는 현재 문자의 오른쪽에 표시하는 것입니다.

이렇게 초기화를 다 해 준 다음에는 LCD 를 클리어 해 줍니다. 이 클리어 함수는 다음과 같습니다.

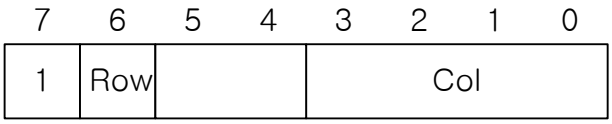
```
def clearLCD() :
  wrData_8bits(LCD_CMD_CLEAR)  # Clear
  sleep(3*D_MS)
```

이 함수는 LCD_CMD_CLEAR 커맨드를 LCD 에 전달하여 LCD 에 표시된 문자를 모두 지우고 커서의 위치를 좌측 상단에 둡니다. 이 커맨드를 LCD 에 전달한 뒤에는 3 ms 를 기다립니다.

다음은 커서의 위치를 설정하는 함수입니다.

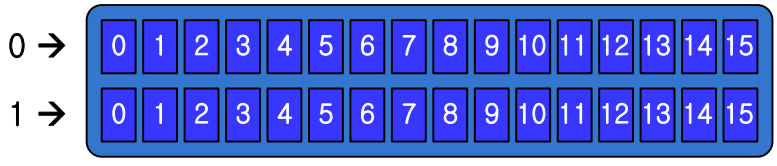
```
def setCursor(row, col) :
  if (row == 0) :
    rowBits = 0x0
  else :
    rowBits = 0x40  # row >= 1
    col = col & 0xf
  wrData_8bits(LCD_CMD_SET_CURSOR | rowBits | col)
```

LCD_CMD_SET_CURSOR 커맨드의 각 비트는 다음과 같이 구성됩니다.



7 번 비트에는 1 을 써 주어야 LCD_CMD_SET_CURSOR 커맨드임을 LCD 가 알 수 있습니다. 6 번 비트는 가로줄을 표시하는 값으로 이 값이 0 이면 첫번째 줄에 문자를 표시하고 이 값이 1 이면 두번째 줄에 문자를 표시합니다. 그래서 setCursor() 함수의 첫번째 인자인 row 값이 0 이면 0 을, 그렇지 않으면 6 번 비트를 1 로 해 주기 위해서 0x40 을 rowBits 변수에 할당해 줍니다.

0 ~ 3 번 비트는 각 줄의 16 개의 문자를 표시하는 위치입니다. 위치는 다음 그림과 같고 0 ~ 15 번의 위치 번호를 부여합니다. 0 ~ 15 번 번호를 0 ~ 3 번 비트에 할당하기 위해서 col = col & 0xf 로 마스킹합니다. 그래서 혹시라도 15 이상의 값이 들어가더라도 다른 비트 값들을 망가트리지 않도록 합니다.



다음은 문자열 (문자 여러 개가 모여 있는 묶음)을 받아 wrData_8bits() 함수에 전달해 주는 코드입니다.

```
def printToLCD(string) :
  for c in string :
    wrData_8bits(ord(c), 1)
```

for 문은 문자열에서 문자 하나씩을 추출하여 c 변수에 저장합니다. 이 c 변수는 ord(c) 라고 해 주었는데요. 이것은 문자를 아스키 코드로 바꾸어주는 함수입니다. 즉, 변수 c 에 'a' 문자를 전달하면 97 이라는 숫자로 바꾸어 줍니다. 이 숫자가 wrData_8bits() 함수에 전달이 됩니다.

이제 이 함수들을 이용하여 첫번째 줄에는 "Coding Kit!!" 라고 쓰고 두번째 줄에는 "Easy & Fun!!" 이라고 쓰는 코드를 작성해 보겠습니다.

```
initLCD()
printToLCD("Coding Kit!!")
setCursor(1, 0)
printToLCD("Easy & Fun!!")
```

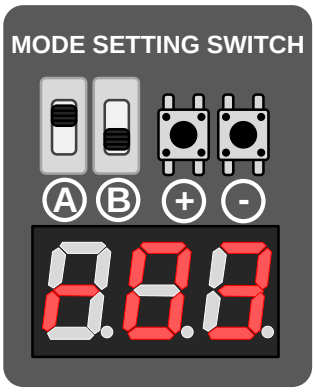
설명이 좀 길었습니다. 잘 알아두셨다가 캐릭터 LCD 를 다양하게 활용해 보세요.

< 예제 코드 : **캐릭터 LCD 에 문자 쓰기**>

< 코드 위치 : Codingkit / **charlcd.py >**

[SPI 핀 확장 모드를 사용하여 여러 디바이스 연결하기]

아두이노와 마찬가지로 라즈베리파이에서도 SPI 확장 모드를 사용할 수 있습니다. 이번장의 스위칭 ID 는 r03 입니다. 이 스위칭 ID 는 SPI 핀 확장 모드를 사용할 때 선택합니다.



SPI 핀 확장 모드를 이용하면 SPI 인터페이스를 통하여 코딩 키트의 대부분의 디바이스를 연결할 수 있습니다. 다음은 핀 확장 모드의 SPI 커맨드들입니다.

SPI Command	코드 (16 진수)	설 명
SPI_CMD_CLEAR	0x00	디바이스들을 초기화 합니다.
SPI_CMD_DM_CL_SEL	0x03	도트매트릭스(0)와 캐릭터 LCD(1) 중 선택
SPI_CMD_DM_ROW	0x04	도트매트릭스의 가로줄 설정
SPI_CMD_DM_COL	0x05	도트매트릭스의 세로줄 설정
SPI_CMD_FND_SEG	0x06	FND 세그먼트 값 설정
SPI_CMD_FND_SEL	0x07	각 FND 선택 신호 설정
SPI_CMD_LCD_BL	0x08	캐릭터 LCD Back-ligth 켜기
SPI_CMD_LCD_RS	0x09	캐릭터 LCD RS 신호 설정
SPI_CMD_LCD_EN	0x0A	캐릭터 LCD EN 신호 설정
SPI_CMD_LCD_DATA	0x0B	캐릭터 LCD Data 신호 설정
SPI_CMD_DCMOTENC_EN	0x12	DC 모터 인코더 활성화 신호
SPI_CMD_DCMOTENC_1	0x13	DC 모터 인코더의 각 휠 간의 시간 중 16~23 비트
SPI_CMD_DCMOTENC_2	0x14	DC 모터 인코더의 각 휠 간의 시간 중 8~15 비트
SPI_CMD_DCMOTENC_3	0x15	DC 모터 인코더의 각 휠 간의 시간 중 0~7 비트
SPI_CMD_PWM_BUZ_EN	0x20	부저 활성화 신호
SPI_CMD_PWM_BUZ_FR_H	0x21	부저 주파수 8~15 비트
SPI_CMD_PWM_BUZ_FR_L	0x22	부저 주파수 0~7 비트
SPI_CMD_PWM_BUZ_DUTY	0x23	부저 듀티비
SPI_CMD_PWM_SMOT_EN	0x24	서보 모터 활성화 신호
SPI_CMD_PWM_SMOT_DUTY_H	0x25	서보 모터 듀티비 8~15 비트 (범위 : 9000~36500)

SPI_CMD_PWM_SMOT_DUTY_L	0x26	서보 모터 듀티비 0~7 비트 (범위 : 9000~36500)
SPI_CMD_PWM_DM_FWD_EN	0x28	DC 모터 정방향 활성화 신호
SPI_CMD_PWM_DM_FWD_FR_H	0x29	DC 모터 정방향 주파수 8~15 비트
SPI_CMD_PWM_DM_FWD_FR_L	0x2a	DC 모터 정방향 주파수 0~7 비트
SPI_CMD_PWM_DM_FWD_DUTY	0x2b	DC 모터 정방향 듀티비
SPI_CMD_PWM_DM_BWD_EN	0x2c	DC 모터 역방향 활성화 신호
SPI_CMD_PWM_DM_BWD_FR_H	0x2d	DC 모터 역방향 주파수 8~15 비트
SPI_CMD_PWM_DM_BWD_FR_L	0x2e	DC 모터 역방향 주파수 0~7 비트
SPI_CMD_PWM_DM_BWD_DUTY	0x2f	DC 모터 역방향 듀티비

위의 커맨드들 중 대부분은 이전 예제에서 다루어졌고, 다루어지지 않았던 도트매트릭스, FND, 캐릭터 LCD 만 이번 장에서 예제로 해 보도록 하겠습니다. 그럼 먼저 도트매트릭스에 하트를 표시하는 예제를 해 보도록 하겠습니다.

SPI 핀 확장 모드를 사용하기 위해서는 CK_SPI_DEV 모듈을 import 합니다.

```
import CK_SPI_DEV as spi_dev
```

도트매트릭스를 사용하기 위해서는 다음 3 개의 커맨드를 이용합니다.

- SPI_CMD_DM_CL_SEL : 코딩 키트에서는 도트매트릭스와 캐릭터 LCD 를 같이 사용할 수 없습니다. 둘 중 하나를 선택하여 사용해야 합니다. 이 커맨드가 도트매트릭스와 캐릭터 LCD 중 하나를 선택하는 커맨드 입니다. 이 커맨들의 데이터가 0 이면 도트매트릭스가 선택되고, 1 이면 캐릭터 LCD 가 선택됩니다. 이번 예제에서는 다음과 같이 코딩하여 도트매트릭스가 선택되도록 합니다.

```
SEL_DOTMATRIX = 0
SEL_CAHR_LCD  = 1
spi_dev.ck_spiWr(spi_dev.SPI_CMD_DM_CL_SEL, SEL_DOTMATRIX)
```

- SPI_CMD_DM_ROW : 가로줄 값들을 설정하는 커맨드입니다. 이 커맨드의 데이터 8 비트가 가로줄 각각의 LED 를 켜고 꺼 줍니다.

- SPI_CMD_DM_COL : 세로줄 값들을 설정하는 커맨드입니다. 이 커맨드의 데이터 8 비트가 세로줄 각각의 LED 를 켜고 꺼 줍니다.

이 커맨드들을 이용한 전체 코드는 다음과 같습니다.

```
import CK_SPI_DEV as spi_dev
import RPi.GPIO as GPIO
from time import sleep
GPIO.setmode(GPIO.BCM)

SEL_DOTMATRIX = 0
```

```
SEL_CAHR_LCD  = 1

spi_dev.ck_spiWr(spi_dev.SPI_CMD_CLEAR, 0)
spi_dev.ck_spiWr(spi_dev.SPI_CMD_DM_CL_SEL, SEL_DOTMATRIX)

heart_col = [0b11111111,
             0b10011001,
             0b00000000,
             0b00000000,
             0b10000001,
             0b11000011,
             0b11100111,
             0b11111111]

try :
  while (1) :
    for i in range(8) :
      spi_dev.ck_spiWr(spi_dev.SPI_CMD_DM_ROW, 1 << i)
      spi_dev.ck_spiWr(spi_dev.SPI_CMD_DM_COL, heart_col[i])
      sleep(0.002)

except KeyboardInterrupt :
  spi_dev.ck_spiWr(spi_dev.SPI_CMD_CLEAR, 0)
  GPIO.cleanup()
```

heart_col 변수를 2 진수 값으로 정의하여 0 값이 하트 모양이 되도록 하였습니다. 위 코드 중 붉은색의 for 문은 도트매트릭스를 한 줄씩 켜 주는 코드입니다. "1 << i" 를 하면 값이 0b00000001, 0b00000010, … 이렇게 변하여 가로줄이 한 줄씩 선택이 됩니다. 그 선택된 줄에 맞추어 세로 값을 할당해 주기 위해서 다음과 같이 코딩한 것입니다.

```
spi_dev.ck_spiWr(spi_dev.SPI_CMD_DM_COL, heart_col[i])
```

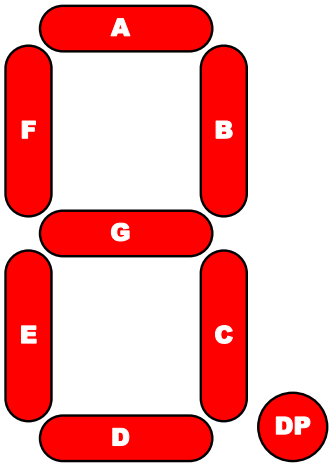
이제 코드를 코딩 키트에서 실행해 봅니다.

< 예제 코드 : **SPI 핀 확장 모드를 이용하여 도트매트릭스에 하트 그리기** >

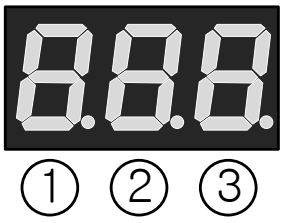
< 코드 위치 : Codingkit / **spi_dm_heart.py** >

다음으로는 FND 를 해 보겠습니다. 가변저항 값을 FND 에 표시하는 예제를 해 보도록 하겠습니다. FND 를 이용하기 위해서는 다음 커맨드를 활용합니다.

- SPI_CMD_FND_SEG : 다음 그림의 A ~ G, GP 를 컨트롤합니다. 8 비트의 데이터 중 7 번이 A, 6 번이 B, … , 1 번이 G, 0 번이 GP 입니다.



- SPI_CMD_FND_SEL : 3 개의 FND 선택 신호입니다. 데이터는 2 번 비트가 다음 그림의 1 번 FND, 1 번 비트가 2 번 FND, 0 번 비트가 3 번 FND 를 선택하는 신호입니다.



이 커맨드들을 이용하여 코딩하기 위해서는 CK_SPI_DEV 모듈을 import 합니다.

```
mport CK_SPI_DEV as spi_dev
```

가변저항 값을 ADC 를 통해서 읽기 위해서는 CK_SPI_ADC 모듈을 import 합니다.

```
import CK_SPI_ADC as spi_adc
```

FND 를 모두 끄는 함수는 다음과 같이 정의합니다.

```
FND_SEG_OFF = 0xff
FND_SEL_OFF = 0x7

def fndOff() :
  spi_dev.ck_spiWr(spi_dev.SPI_CMD_FND_SEL, FND_SEL_OFF)
  spi_dev.ck_spiWr(spi_dev.SPI_CMD_FND_SEG, FND_SEG_OFF)
```

세그먼트와 셀렉트 모두에 1 을 쓰면 FND 는 꺼집니다.

다음과 같이 2 진수로 0 ~ 9 까지 숫자를 표시하기 위한 FND 의 각 세그먼트 값을 정의합니다.

```
nd_num = [0b00000011,
      0b10011111,
      0b00100101,
      0b00001101,
```

```
0b10011001,
0b01001001,
0b01000001,
0b00011111,
0b00000001,
0b00001001]
```

셀렉트 신호도 다음과 같이 2 진수로 정의 합니다.

```
FND_SEL_1 = 0b011
FND_SEL_2 = 0b101
FND_SEL_3 = 0b110
```

가변저항 값으로 각 자릿수의 값을 구하는 것은 이전에 GPIO 를 이용하여 FND 를 컨트롤했던 예제 설명을 참과합니다.

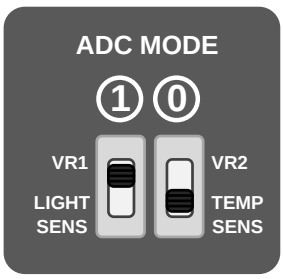
다음과 같이 1 번 FND 에 가변저항 값 중 100 의 자리 값을 표시합니다.

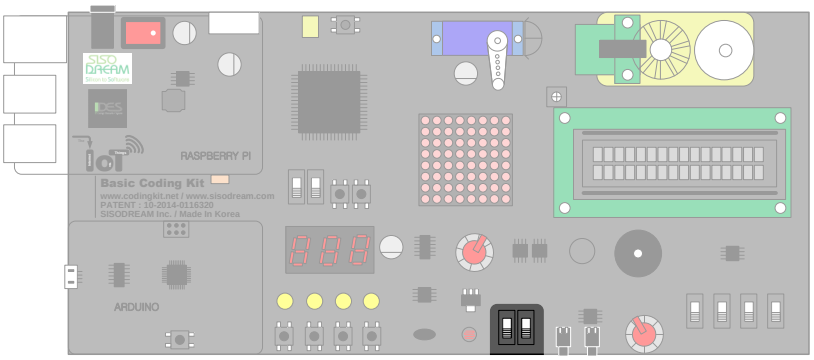
```
fndOff()
spi_dev.ck_spiWr(spi_dev.SPI_CMD_FND_SEL, FND_SEL_1)
spi_dev.ck_spiWr(spi_dev.SPI_CMD_FND_SEG, fnd_num[fnd1_num])
sleep(DELAY)
```

먼저 FND 를 꺼줍니다. 그리고 셀렉트 값을 SPI_CMD_FND_SEL 커멘드를 이용하여 설정해 줍니다. 세그먼트 값은 미리 정의해 둔 숫자 리스트에서 가변 저항의 100 의 자리 값에 해당하는 원소를 사용합니다.

2번, 3번 FND 도 위의 1 번 FND 코드와 같습니다.

코딩 키트에서 실행하시기 전에 ADC 모드 스위치 중 1 번은 위로 올려 줍니다.





가변저항을 돌리면서 FND 에 값이 잘 표시되는지 확인하세요.

< 예제 코드 : **SPI 핀 확장 모드를 이용하여 세븐세그먼트에 가변저항 값 표시하기** >

< 코드 위치 : Codingkit / **spi_fnd_vr.py >**

이번에는 캐릭터 LCD 를 컨트롤해 보겠습니다. 이전에 했던 GPIO 를 이용한 캐릭터 LCD 컨트롤과 코드가 매우 비슷합니다. GPIO 로 컨트롤했던 신호들을 SPI 커맨드로 바꾼 것 뿐입니다. 전체 코드는 다음과 같습니다.

```
import CK_SPI_DEV as spi_dev
import RPi.GPIO as GPIO
from time import sleep
GPIO.setmode(GPIO.BCM)

LCD_CMD_CLEAR = 0x01
LCD_CMD_SET_CURSOR = 0x80

D_MS = 0.001      # Delay : 1 Milisecond
D_US = 0.000001   # Delay : 1 Microsencod

SEL_DOTMATRIX = 0
SEL_CHAR_LCD  = 1

spi_dev.ck_spiWr(spi_dev.SPI_CMD_CLEAR, 0)
spi_dev.ck_spiWr(spi_dev.SPI_CMD_DM_CL_SEL, SEL_CHAR_LCD)
spi_dev.ck_spiWr(spi_dev.SPI_CMD_LCD_BL, 1)
spi_dev.ck_spiWr(spi_dev.SPI_CMD_LCD_EN, 0)
spi_dev.ck_spiWr(spi_dev.SPI_CMD_LCD_RS, 0)

def enablePluse() :
    spi_dev.ck_spiWr(spi_dev.SPI_CMD_LCD_EN, 0)
    sleep(D_US)
    spi_dev.ck_spiWr(spi_dev.SPI_CMD_LCD_EN, 1)
    sleep(D_US)
    spi_dev.ck_spiWr(spi_dev.SPI_CMD_LCD_EN, 0)
    sleep(D_US)

def wrData_4bits(data4) :
    spi_dev.ck_spiWr(spi_dev.SPI_CMD_LCD_DATA, data4 & 0xf)
    enablePluse()

def wrData_8bits(data8, data_en = 0) :
    spi_dev.ck_spiWr(spi_dev.SPI_CMD_LCD_RS, data_en)
    wrData_4bits(data8 >> 4)
    wrData_4bits(data8)

def clearLCD() :
    wrData_8bits(LCD_CMD_CLEAR)  # Clear
    sleep(3*D_MS)

def printToLCD(string) :
    for c in string :
        wrData_8bits(ord(c), 1)

def initLCD() :
```

```
    # Reset LCD
    sleep(15*D_MS)     # Wait Time > 15 ms
    wrData_4bits(0x3)
    sleep(5*D_MS)      # Wait Time > 4.1 ms
    wrData_4bits(0x3)
    sleep(200*D_US)  # Wait Time > 100 us
    wrData_8bits(0x32)
    # Init LCD
    wrData_8bits(0x28)  # Mode : 4-bit, Line : 2, Matrix : 5x7
    wrData_8bits(0x0C)  # Display : On, Cursor : Off, Cursor Blink : Off
    wrData_8bits(0x06)  # Cursor Shift : Right
    clearLCD()

def setCursor(row, col) :
    if (row == 0) :
        rowBits = 0x0
    else :
        rowBits = 0x40  # row >= 1
    col = col & 0xf
    wrData_8bits(LCD_CMD_SET_CURSOR | col | rowBits)

initLCD()
printToLCD("Coding Kit!!")
setCursor(1, 0)
printToLCD("Easy & Fun!!")
setCursor(0, 0)

try :
    while(1) :
        sleep(1)

except KeyboardInterrupt :
    clearLCD()
    spi_dev.ck_spiWr(spi_dev.SPI_CMD_CLEAR, 0)
    GPIO.cleanup()
```

< 예제 코드 : **SPI 핀 확장 모드를 이용하여 캐릭터 LCD 에 문자 쓰기**>

< 코드 위치 : Codingkit / **spi_charlcd.py >**

[부록 A : 스위칭(Switching) ID]

코딩 키트는 스위칭 시스템을 이용하여 여러가지 연결을 설정할 수 있습니다. 이렇게 연결된 각각의 설정은 스위칭 ID 로 표시합니다. 다음은 코딩 키트의 라즈베리파이에서 스위칭 ID 에 따른 디바이스 연결 표입니다. 표에서 붉은색으로 표시된 숫자는 파란색으로 표시한 스위칭 ID 에서 각각의 디바이스에 연결된 라즈베리파이의 핀 번호입니다. 여러분은 라즈베리파이에서 이 번호로 코딩을 하시면 됩니다. 확인(√) 표시는 SPI 확장 모드로 연결하여 더 이상 핀 수는 소요되지 않지만 연결은 가능하다는 표시입니다. Device Control SPI 항목이 SPI 확장 모드를 의미합니다.

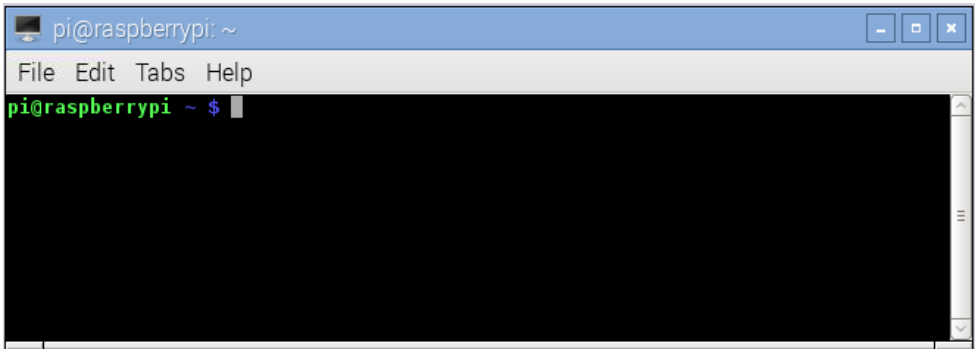
Raspberry Pi 디바이스	핀수	Switching ID 핀 번호			
		r00	r01	r02	r03
LED 0	1	22	√	√	22
LED 1	1	23	√	√	23
LED 2	1	24	√	√	24
LED 3	1	25	√	√	25
Button 0	1	4	√	√	4
Button 1	1	5	√	√	5
Button 2	1	6	√	√	6
Button 3	1	16	√	√	16
Buzzer	1	15	√	√	15
DC Motor Enable	1	17	17	17	17
DC Motor Forward	1	√	√	√	√
DC Motor Backward	1	√	√	√	√
DC Motor Encoder Enable	1	20	20	20	20
DC Motor Encoder Data	1	√	√	√	√
Sound Sensor Detect	1	14	14		14
IR Sensor LED	1	13	13	13	13
IR Sensor Detect	1	12	12		12
Servo Motor	1	√	√	√	√
Dotmatrix	16		*3	√	√
7-Segment	11		√	*4	√
Device Control SPI	4	*1	*1	*1	*1
Character LCD EN, RS, Data	6		√	*5	√
Character LCD Back Light	1		√	27	√
Switch 0	1	26	√	√	26
Switch 1	1	27	√	√	27
Switch 2	1	2	√	√	2
Switch 3	1	3	√	√	3
ADC SPI	4	*2	*2	*2	*2

위의 표에서 *1, *2, *3, *4, *5 는 여러핀이 연결된 것을 나타내며 그 내용은 다음과 같습니다.

*1 : SPI(Device)	*2 : SPI (ADC)	*3 : Dotmatrix	*4 : FND	*5 : Char LCD
SPI_SS : 5	SPI_SS : 6	DM_ROW_1 : 3	FND_A : 2	LCD_EN : 6
SPI_MOSI : 8	SPI_MOSI : 8	DM_ROW_2 : 19	FND_B : 3	LCD_RS : 15
SPI_MISO : 7	SPI_MISO : 7	DM_ROW_3 : 22	FND_C : 4	LCD_D0 : 23
SPI_SCLK : 9	SPI_SCLK : 9	DM_ROW_4 : 18	FND_D : 5	LCD_D1 : 24
		DM_ROW_5 : 16	FND_E : 16	LCD_D2 : 25
		DM_ROW_6 : 5	FND_F : 12	LCD_D3 : 26
		DM_ROW_7 : 24	FND_G : 18	
		DM_ROW_8 : 25	FND_DP : 19	
		DM_COL_1 : 2	FND_SEL_1 : 14	
		DM_COL_2 : 23	FND_SEL_2 : 21	
		DM_COL_3 : 6	FND_SEL_3 : 22	
		DM_COL_4 : 4		
		DM_COL_5 : 21		
		DM_COL_6 : 15		
		DM_COL_7 : 26		
		DM_COL_8 : 27		

[부록 B : 리눅스 명령어]

리눅스 명령어는 다음과 같은 터미널 프로그램에서 실행됩니다.



명령어를 입력할 위치에 위의 그림에서 초록색으로 "pi@raspberrypi ~ $" 라고 쓰여진 부분을 프롬프트라고 부릅니다. 이것이 표시가 되면 새로운 명령어를 받을 준비가 되었다고 알려주는 것입니다. 그리고 이 프롬프트는 다음과 같은 의미입니다.

아이디@컴퓨터이름 디렉토리 $

위와 같은 구성을 "pi@raspberrypi ~ $" 라고 쓰면 아이디는 pi 인 것이고 컴퓨터 이름은 raspberrypi 인 것입니다. 디렉토리는 "~" 인데 이것은 홈디렉토리를 말합니다. 리눅스에서는 각각의 사용자에게 읽고 쓰고 하는 권한이 부여된 디렉토르를 할당해 줍니다. 이것을 그 사용자의 홈디렉토리라고 합니다. 그래서 사용자가 로그인 하면 홈디렉토리에 위치해 있습니다. 그리고 그 디렉토리에 새로운 디렉토리를 만들거나 파일을 만들 수 있습니다.

다음은 자주 사용되는 명령어들입니다.

- ls (listing) : 해당 디렉토리의 항목을 리스트함

  pi@raspberrypi ~ $ ls –F : -F 옵션은 디렉토리, 실행 파일등을 표시

- cd (change directory)

  pi@raspberrypi ~ $ cd : 홈디렉토리로 이동 :

  pi@raspberrypi ~ $ cd directory_name : 해당 디렉토리로 이동

  - 상대경로와 절대 경로를 사용할 수 있음

- mv (move)

  - 파일 이름 변경 : pi@raspberrypi ~ $ mv file_name1 file_name2

  - 파일 및 디렉토리의 위치 이동 : pi@raspberrypi ~ $ mv file_name1 directory_name

- rm (remove) : 파일 삭제

  - 디렉토리까지 모두 삭제 : pi@raspberrypi ~ $ rm –rf

- mkdir (make directory) : 디렉토리 생성

- pwd (print working directory) : 현재 디렉토리를 보여 줌

- cp (copy)

  - 복사 : pi@raspberrypi ~ $ cp file_name1 file_name2

  - 디렉토리 복사 : pi@raspberrypi ~ $ cp –r directory_name1 directory_name2

- 키보드 위 아래 방향키 : 이전 커맨드 탐색

- Tab 키 : 커맨드 자동 완성

[Release Note]

V1.0 : 2015. 8. 28

-. 첫번째 버전 Release

V1.1 : 2016. 6. 12

-. Short Version 보드 사진 추가

-. 라즈베리파이에 모니터, 키보드, 마우스 직접 연결할 때 startx 입력하는 내용 추가

-. 오타 수정 : < 예제 → **led_3sec_on_off.py> → led_3sec_on_2sec_off.py**

V1.2 : 2016. 8. 24

-. 지원 보드 중 라즈베리파이 3 추가 (P.8)

-. 라즈베리파이 3 일 때 원격제어로 연결하는 해상도 (P.17)

-. 라즈베리파이 3 원격제어에서 Shutdown 할 때 암호 물어 봄 (P.18)

V2.0 : 2017. 3. 16

-. 2.0 버전 Release

-. 서보모터 범위 조정 – 관련 라이브러리 수정(CK_SPI_DEV.py)