창의적 **공학 설계**를 위한

HOVIS DRC, 비주얼로직 & C로봇프로그래밍

- 창의적 학습 구현을 위한 Visual Logic 기본 사용법과 알고리즘 학습
- AVR 펌웨어 로봇 C프로그래밍 완벽가이드 (ATMega 128 마이크로 콘트롤러)

동부로봇 자료실

(매뉴얼 및 어플리케이션을 다운로드 받으십시오.)

www.dongburobot.com

Dongbu Robot

Dongbu Robot

창의적 공학 설계를 위한
Visual Logic & C
Robot Programming

초판 발행일 : 2012년 10월 04일
펴낸곳 : ㈜동부로봇
주소 : 경기도 부천시 원미구 약대동 부천테크노파크 11층
내용문의 : 032-329-5551
FAX : 032-329-5569

목차

PART 02 DR-Visual Logic Programming

PART 03 Visual Logic 연계된 C언어 문법익히기

PART 04 AVR 로봇제어기 Firmware 익히기

PART 01

Donbu Robot
DRC & Humanoid

PART 01

소개

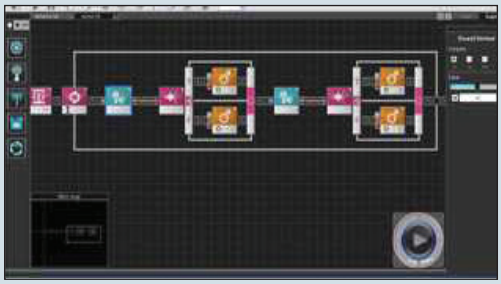
HOVIS

Hovis Lite 소개

호비스 라이트를 소개합니다.

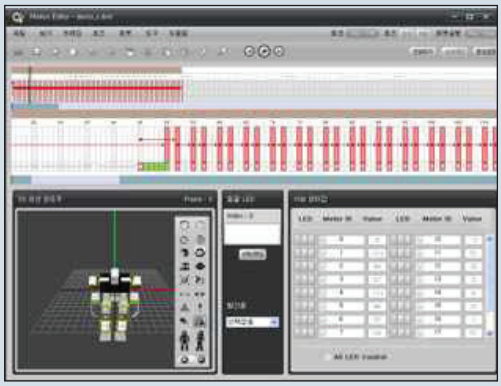
컬러풀한 브라켓

4가지 색상의 브라켓을 이용하여 컬러풀한 휴머노이드 제작이 가능합니다. 또한 케이스를 씌워 완성형 로봇과 옴니휠 부착, 안드로이드 단말기 부착 등 업그레이드가 가능한 세계 최초 로봇입니다.



DR-Visual Logic (Task Editor)

제어기 DRC 를 기반으로 프로그래밍할 수 있게, 총 24가지 모듈을 제공하고, Drag & Drop 방식으로 프로그래밍 언어를 전혀 모르는 초보자도 쉽게 접근할 수 있게 만든 그래픽 프로그래밍 언어툴입니다.



DR-SIM (Motion Editor)

로봇 모션 편집기에 동영상 편집기에 사용되는 타임프레임을 도입하여, 화면에서 직접 모션을 만들 수 있고, 로봇으로부터 캡쳐도 가능한 다기능 로봇 모션 제작 툴입니다. 만들어진 모션은 화면 시뮬레이션을 통해 미리 볼 수있고, 로봇에 다운로드 하여 즉시 실행이 가능합니다.

- 4가지 색깔 브라켓
- 27가지 로봇 조립 가능
- 케이스별 업그레이드 가능
- 안드로이드 단말기와 프로그래밍 인터페이스 제공
- 소스 제공
- 커리큘럼 제공

관련 사이트

학습자료 다운로드 :

www.hovis.co.kr/guide
www.dongburobot.com → 고객지원 → 서비스용 → 자료실
www.dongburobot.com → Quick Menu → 고객지원 → 서비스용 → 자료실

PART 01

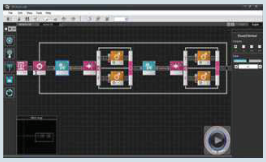
소개

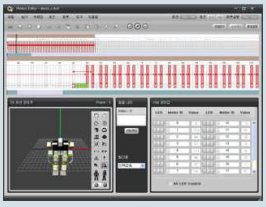
HOVIS

조립부터 테스트까지

조 립

프로그래밍





테스트

■ **조립**

로봇을 조립합니다. 로봇조립 방법은 조립도에 상세하게 설명되어 있습니다. 동부로봇은 모터 16축 ~ 20축 기본 휴머노이드 로봇의 조립도를 제공하며, 그 외에 26가지 이상의 확장형 로봇을 제작할 수 있는 키트로 구성되어있습니다. 각종 로봇 조립방법은 웹사이트에 웹매뉴얼로 제공하고 있습니다.

■ **프로그래밍**

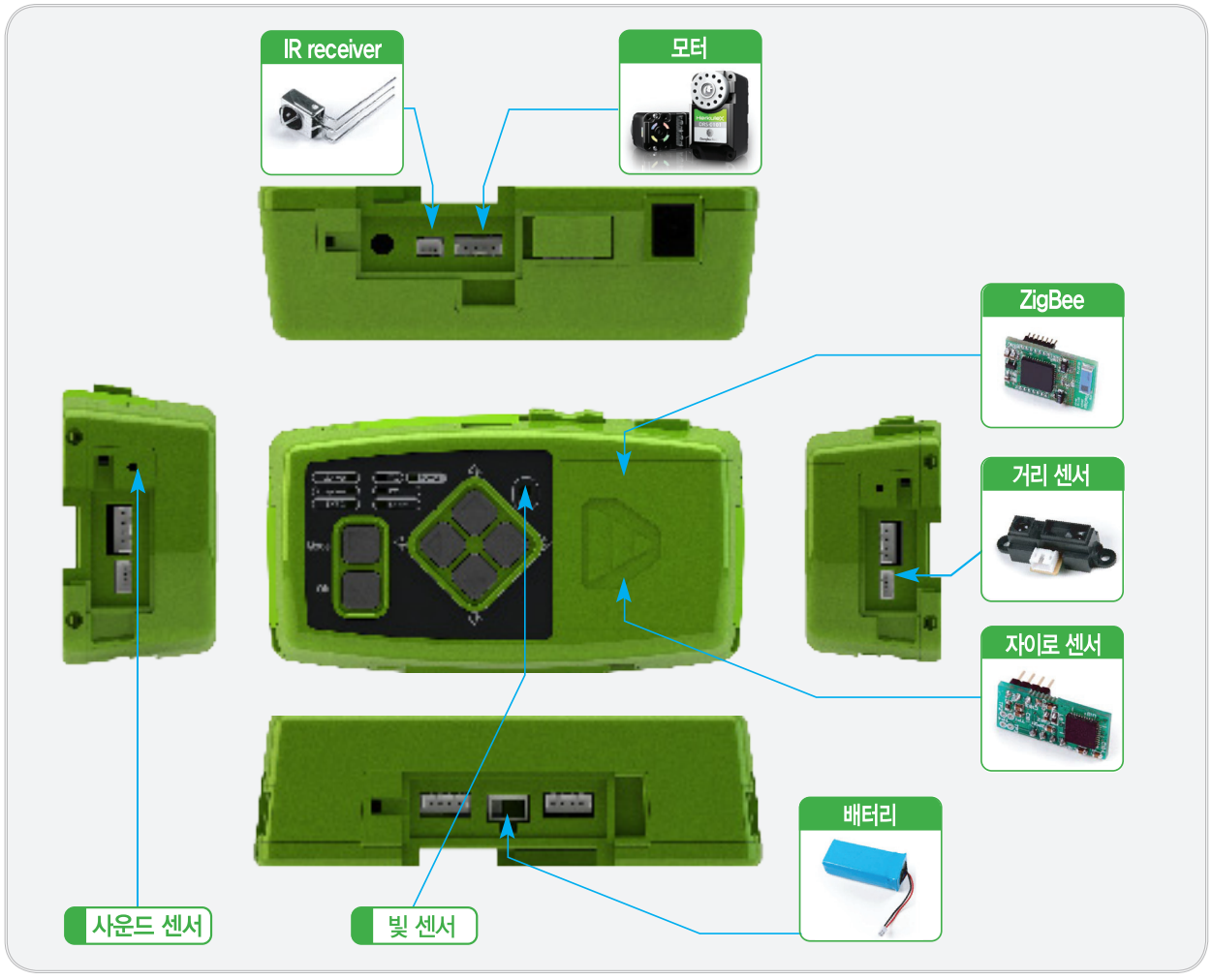
DR-Visual Logic 소프트웨어를 이용하여 로봇을 프로그래밍 합니다. 또한 DR-SIM 소프트웨어를 이용하여 로봇 동작을 편집하거나 캡쳐하여 모션을 만들 수 있습니다. 만들어진 모션은 로봇에 다운로드 하여 바로 실행 해 볼 수 있습니다. 또한 DRC 제어기에 저장하여 DR-Visual Logic 을 통해 불러와 프로그래밍 할 수 있습니다.

■ **테스트**

DR-Visual Logic 에서 작성한 프로그램을 컴파일하여 로봇에 다운로드 합니다. 원하는 동작을 실행시키지 못했다면, 다시 모듈별 속성값을 조정하고, 다운로드하여 실행한 후 테스트 해보세요.

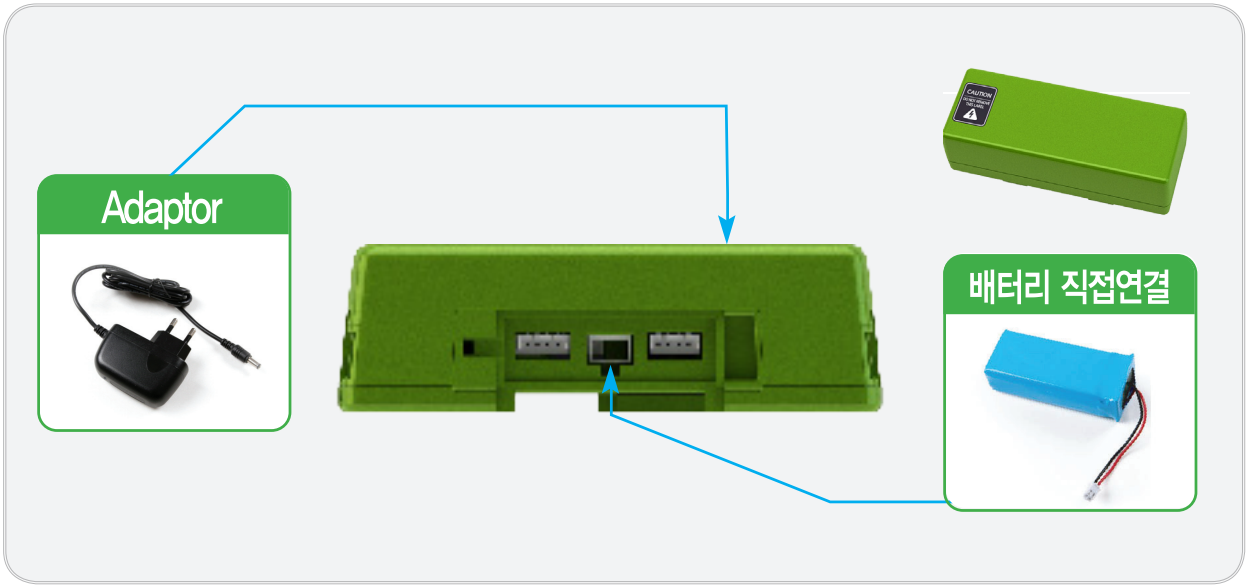
HOVIS DRC 개요

제어기 DRC 는 Hovis Lite 구성에서 두뇌 역할을 하는 메인 부품입니다. 제어기에는 6개의 서보모터 포트, 2개의 PSD 센서 포트, Gyro 센서, ZigBee 를 장착할 수 있는 커넥터등의 인터페이스를 가지고 있습니다, 또한 빛센서와 사운드 센서는 내장되어있습니다. DR-Visual Logic 프로그램은 본 제어기 DRC 의 기능을 이용하여 프로그래밍 할 수 있는 비쥬얼 로봇 프로그래밍 언어입니다. 각종 센서 테스트 및 모터를 1~32개까지 연결하여 프로그래밍이 가능합니다.



CPU	ATMega 128
Size, Weight	108 x 58.5 x 33 (mm), 82 g
Operating Voltage	Tolerance Range : 6.5V ~ 10V, Recommended Voltage : 7.4V
Serial Speed	115,200 bps ~ 666,667 bps
Consumed Current	When IDLE : 50mA, Overall Max Current : 3A (PTC Fuse)
Interface	Button : 6ea, MIC : 2ea, LED : 7ea
External I/O	Servo Motor : 6ea, PSD Sensor : 2ea
Back Cover I/O	ZigBee : 1ea, Gyro Sensor : 1ea
Internal I/O	Sound Sensor : 2ea, Light Sensor : 1ea

DRC 배터리 장착 방법



■ **배터리**

전원을 DRC에 공급하기 위하여 배터리와 제어기 하단 전원 포트를 연결합니다.

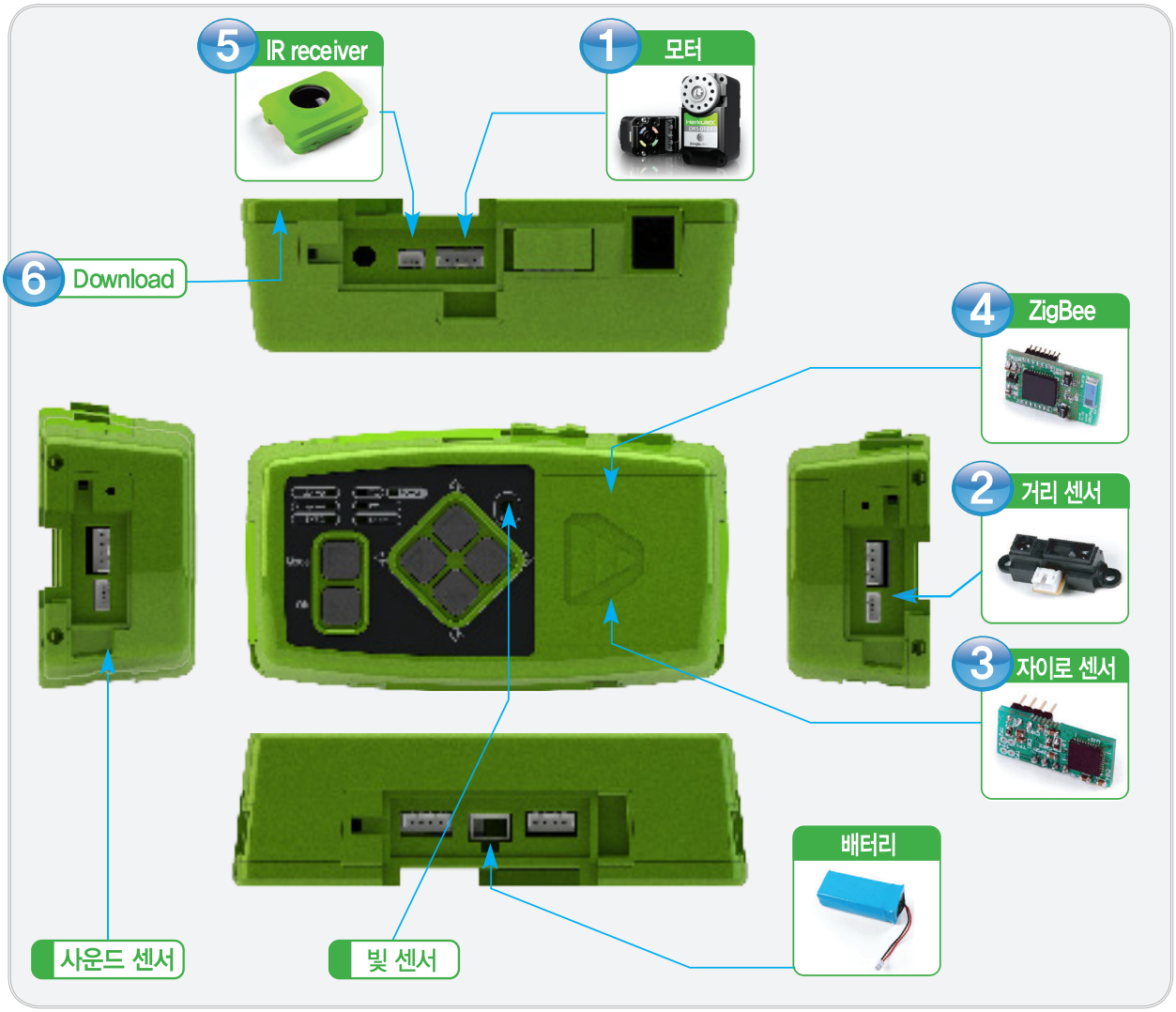
■ **충전방법**

아답터 와 배터리 케이블을 이용하여 직접 충전할 수 있습니다.
또 다른 방법은 제어기 상단에 아답터 커넥터가 있습니다.
그 커넥터에 아답터를 꽂으면 제어기와 연결된 배터리에 자동 충전됩니다.

■ **배터리 부족**

배터리 잔량이 20% 이하로 감소시, Power LED 불이 깜빡거립니다.

DRC 연결 방법



1 모터 연결
제어기 주변 총 5개의 포트가 있습니다.

2 거리 센서 연결
좌,우로 2개의 포트가 있습니다.

3 자이로 센서 연결
제어기 슬롯 커버를 연 후 내부에 장착합니다.

4 ZigBee 연결
제어기 슬롯 커버를 연 후 내부에 장착합니다.

5 IR Receiver 연결
1개 포트가 있습니다. 리모콘 수신용입니다.

6 다운로드 연결
Ear phone Jack 으로 PC 프로그램을 로봇에 다운로드할 때 쓰입니다.

DRC 인터페이스

제어기 DRC 는 제어기 정면에서 Input / Output을 나타내는 버튼과 LED 가 있고, 측면에는 모터나 센서를 연결하는 포트로 구성되어 있습니다. DRC 의 User Interface 는 정면의 버튼의 조작으로 제어기에 Input 명령을 할 수 있고, LED 를 통하여 Output 에 대한 데이터를 확인할 수 있습니다.

	표 기	단 축 키	기본 Task 모드
Main Button	Mode	Task실행	
	Ok	확인버튼	
Navi Key	(좌)	배터리 잔량표기	점검 모드
	(상)	DRC 자체 테스트	자율 행동 모드
	(우)	유/무선 통신전환	리모콘 조종 모드
	(하)	모터 ID 스캔	사운드 데모 모드
LED	Servo	HerkuleX 실행중	
	Program	DR-SIM/Visual Logic 실행중	
	EXEC	Task 실행중	
	TX	데이터 송신	
	RX	데이터 수신	
	Spare	사용자 할당	
	Power	파워표시	
Sensor	Cds	Light Sensor	

DRC Register Map & Protocol

레지스터

DRC에는 레지스터라는 영역이 있어서, 제어기의 현재 상태와 작동 설정, 그리고 센서에 관련한 여러 가지 데이터가 들어가 있습니다. 예를 들면 현재 제어기에 연결된 서보 모터의 수와 ID, 제어기의 에러 상태와 에러 코드 등은 제어기의 현재 상태에 해당하고, 제어기가 작동할 수 있는 최소/최대 전압, 제어기의 명령 응답 설정 등은 제어기의 작동 설정에 해당합니다. 또한 빛 센서에 들어오는 현재 광량, 사운드 센서가 감지한 소리의 방향 등은 센서 데이터에 해당합니다.

제어기의 레지스터는 비휘발성(Non-Volatile, EEPROM) Register와 휘발성(Volatile, RAM) Register로 구분됩니다. 비휘발성 레지스터는 전원이 꺼진 상태에서도 데이터가 지워지지 않는 영역으로, 제어기 동작의 기본적인 설정 값들이 저장됩니다. 비휘발성 레지스터의 값은 전원이 켜진 순간 휘발성 레지스터의 해당 영역으로 복사됩니다. 휘발성 레지스터는 제어기 동작 설정이나 제어기의 상태, 센서 값을 저장하고 있으며, 휘발성 레지스터에 저장된 동작 설정이 제어기 동작에 직접적인 영향을 미칩니다.

레지스터에 어떤 항목들이 있고 각 항목이 어떻게 변화하는지를 알면, DR-Visual Logic으로 로봇 행동을 프로그램 할 때 조금 더 섬세한 행동 조건을 만들 수 있습니다. 또한 제어기의 상태를 읽거나 작동 설정을 바꾸어 사용자의 로봇 사용을 편리하게 할 수 있습니다. 자세한 설명은 DRC 레지스터 및 프로토콜 상세 매뉴얼을 참조하시기 바랍니다.

프로토콜

사용자가 레지스터 값을 읽거나 변경하려면 그것을 지시하는 명령을 보내야 하는데, 그 명령의 정해진 형식을 프로토콜이라고 합니다. 프로토콜에는 레지스터를 읽고 쓰는 명령 외에도, 제어기에 저장된 모션이나 태스크, 멜로디 등을 실행하는 명령과 제어기를 재부팅하는 명령 등 다양한 명령이 정의되어 있습니다.

PC와 제어기가 통신할 때에는 이렇게 정의된 프로토콜을 통해서 요청 패킷과 응답 패킷을 주고 받게 됩니다. 동부로봇에서 제공하는 DR-SIM과 DR-Visual Logic도 이 프로토콜을 이용하여 만들어져 있습니다. 사용자는 프로토콜을 숙지하여 자신이 직접 만든 프로그램으로 DRC를 제어할 수 있습니다.

자세한 내용은 DRC 레지스터 및 프로토콜 상세 매뉴얼을 참조하시기 바랍니다.

DRC 기능 사용법

1 프로그램 정의

Firmware : 사용자가 변경할 수 없도록 Fix 된 내부 프로그램 입니다.

Task : Task Editor (DR-Visual Logic)을 통해 User 가 변경할 수 있는 사용자 정의 프로그램 입니다.

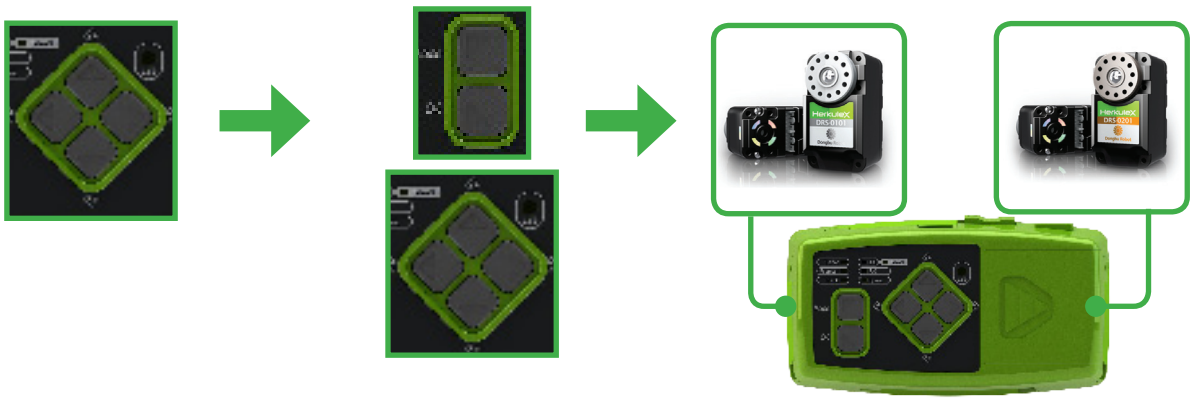
출시시 휴머노이드 전용 기본 Task Program 은 정의되어있고, User 가 수정 가능합니다.

2 동작 방법

전원을 켜고 Navi Key를 누르면 Firmware 에 들어 있는 기본 기능이 실행되고, Mode 버튼을 누르면 저장되어 있는 Task가 실행됩니다. 기본 Task의 경우, Task 실행후 다시 Navi Key → Ok 를 눌러 어떤 모드로 들어갈 지 선택할 수 있습니다.

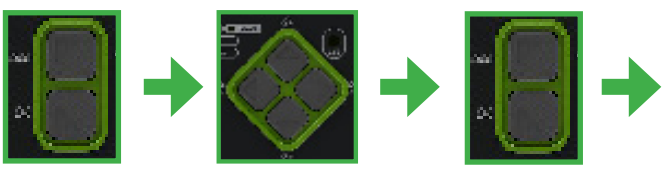
Firmware (사용자가 바꿀 수 없음)

1 Navi 버튼 클릭 → 2 Mode or Navi 클릭 → 3 제어기 자체 테스트



Task Mode (사용자가 바꿀 수 있음)

1 Mode 클릭 → 2 Navi 클릭 → 3 Ok 클릭



4 로봇 제어기 테스트

DRC 기능 사용법

3 기능 사용법 및 동작

	표기	설명	기능 및 동작 방법
Main Button	Mode	모드변경	Task를 시작함 기본 Task의 경우 Mode→ Navi key → Ok로 동작모드 선택 가능
	OK	확인 버튼	
Navi Key (Firmware)	(좌)	배터리 잔량표기	배터리 잔량 확인 가능 → 좌측 LED 3개로 표시 가장 낮을 때 1개, 중간일 때 2개, 가장 높을 때 3개
	(상)	테스트	제어기 자체로 모터 및 센서 테스트 사용법 : (상) → 버튼 → 센서에 따른 모터 반응으로 테스트 테스트항목 : 모터/빛/소리/거리/가속도/자이로센서
	(우)	유/무선 전환버튼	Ear phone jack을 통한 유선 통신 / ZigBee를 통한 무선 통신 전환
	(하)	모터 ID 스캔	현재 연결된 모터의 ID를 다시 스캔
Navi Key (휴머노이드 전용 기본 Task)	(좌)	점검 모드	Mode → (좌) → Ok : 모터확인모드 : 모터 하나씩 선택, 해당 ID의 모터가 존재하지않을 경우 경고음이 울림
	(상)	자율 행동 모드	Mode → (상) → Ok : 자율보행모드 : 로봇이 움직이면서 스스로 동작
	(우)	리모콘 조종 모드	Mode → (우) → Ok : 리모콘 조종 모드,리모콘에 정의된 동작하기
	(하)	사운드 데모 모드	Mode → (하) → Ok : 사운드 데모 모드 : 소리입력에 따라 다양한 동작하기
LED_mode	Servo	HerkuleX 실행중	HerkuleX Manager 가 실행중일 때 깜박거림
	Program	DR-SIM/Visual Logic 실행중	DR-SIM / Visual Logic 을 이용하여 편집사용중일 때 깜박거림 데이터나 펌웨어를 다운로드 할때 켜저있음
	EXEC	모드변경/Task실행	Mode 버튼을 눌러 Task 모드로 진입 시 Task가 실행중인 동안 켜져있음
	TX	데이터 송신	외부로 데이터 송신할 때 깜박거림, task 실행중 일때는 User Spare 공간
	RX	데이터 수신	외부로부터 데이터 수신할 때 깜박거림, task 실행중일때는 User Spare 공간
	Spare	사용자 할당	
	LED 점멸	에러발생	실행중 에러가 생긴 경우 우측 3개 LED 깜박거리면서 알림
LED_Power	Power	파워표시	잔여량이 20% 미만일때 깜박거림
Sensor	CdS	Light Sensor	빛 감지 센서

DRC 기본 테스트

DRC 는 로봇을 조립하지 않고, 테스트 모드를 통해 기본 테스트가 가능합니다. 전원을 켜고 제어기 단축키인 (상) 버튼을 누른 후 모터 및 센서 테스트를 진행합니다. 센서 테스트는 모터ID 1번과 2번을 각각 좌,우에 끼운 상태에서 모터 반응으로 테스트를 진행합니다. 테스트 항목은 모터, 빛센서, 소리센서, 거리센서, 자이로센서 등입니다. 시나리오와 작동법 요약은 아래와 같습니다.

빛센서, 소리센서는 내장되어있는 기능이라 menu와 ok 로 처리합니다.
PSD, Acc/Gyro 는 (하) 키를 누르면 먼저 각 센서가 하나씩만 접속이 되었는지 체크하여 한개만 꽂혀 있으면 처리하고 한번에 두개가 동시에 꽂혀 있으면 처리하지 않습니다.

1 **모터** : 좌,우에 하나씩 모터를 끼움(좌측에 1번, 우측에 2번)
(상) → (좌) : 좌측 모터가 반복적으로 움직임, (우) 우측 모터가 움직임, (상) 모터 두개가 동시에 움직임

2 **빛센서** : (상) → (Menu) : Light 기능이 동작함, Cds 창은 감추면 모터 두개가 동시에 움직임

3 **소리센서** : (상) → (OK) : Sound Sensor 가 동작함,
왼쪽에 박수치면 왼쪽 모터가 움직임,
오른쪽에 박수치면 오른쪽 모터가 움직임

아래기능은 센서 연결후 (하)버튼을 누르면 동작합니다. 테스트할 센서 하나만 연결하여야 하며 2개 이상이 연결되어 있으면 동작하지 않습니다.

4 **PSD Digital** : (상) → 디지털 거리센서 연결→ (하) PSD 작동함
10cm 이하로 물체가 접근하면 → 모터 두개가 움직임
10cm 이상으로 물체가 멀어지면(낭떠러지 감지) → 모터 두개가 멈춤
→ 낭떠러지 감지

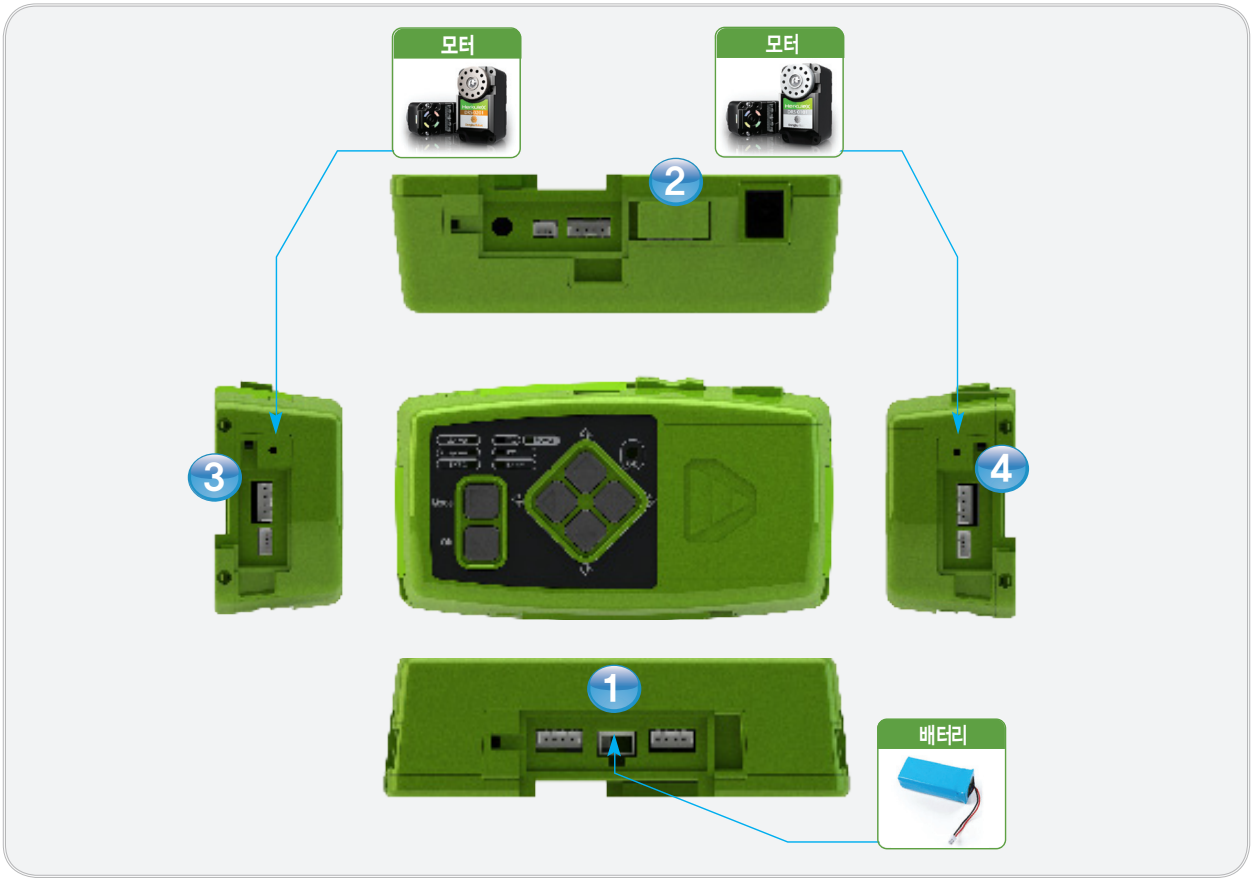
5 **PSD Analog** : (상) → 아날로그 PSD센서 연결 → (하) PSD 작동함
작동함과 동시에 모터 두개가 같은 방향으로 움직임. 물체의 거리가 멀 수록 더 빨리 움직이고
물체가 가까우면 느리게 움직임 → 물체가 10 cm 내로 접근하면 모터가 반대로 움직임
→ 벽 감지 후 피하는 시나리오임

6 **Acc** : (상) → Acc/Gyro 장착 → (하) Acc 동작함
제어기가 등에 붙은 각도대로 똑바로 선 상태에는 모터가 멈춰있음. 제어기의 기울기에 따라서
모터속도가 다르게 움직임. 더 많이 눕힐 수록 더 빠르게 움직임.

7 **Gyro** : (상) → Acc/Gyro 장착 → (하) Acc 동작함 → (하) Gyro 동작함
제어기가 멈춰 있을 때에는 움직이지 않음.
제어기가 돌아가는 속도와 비슷하게 모터도 함께 움직임.

다음 상세 테스트 페이지를 통하여 천천히 따라해보세요

DRC 기본 테스트 : 모터



1 배터리 연결

2 전원켜기

3 좌측 모터 연결 : 반드시 Motor ID 1 번을 연결하여야 합니다.(다른 모터는 동작하지 않습니다.)

4 우측 모터 연결 : 반드시 Motor ID 2 번을 연결하여야 합니다.(다른 모터는 동작하지 않습니다.)

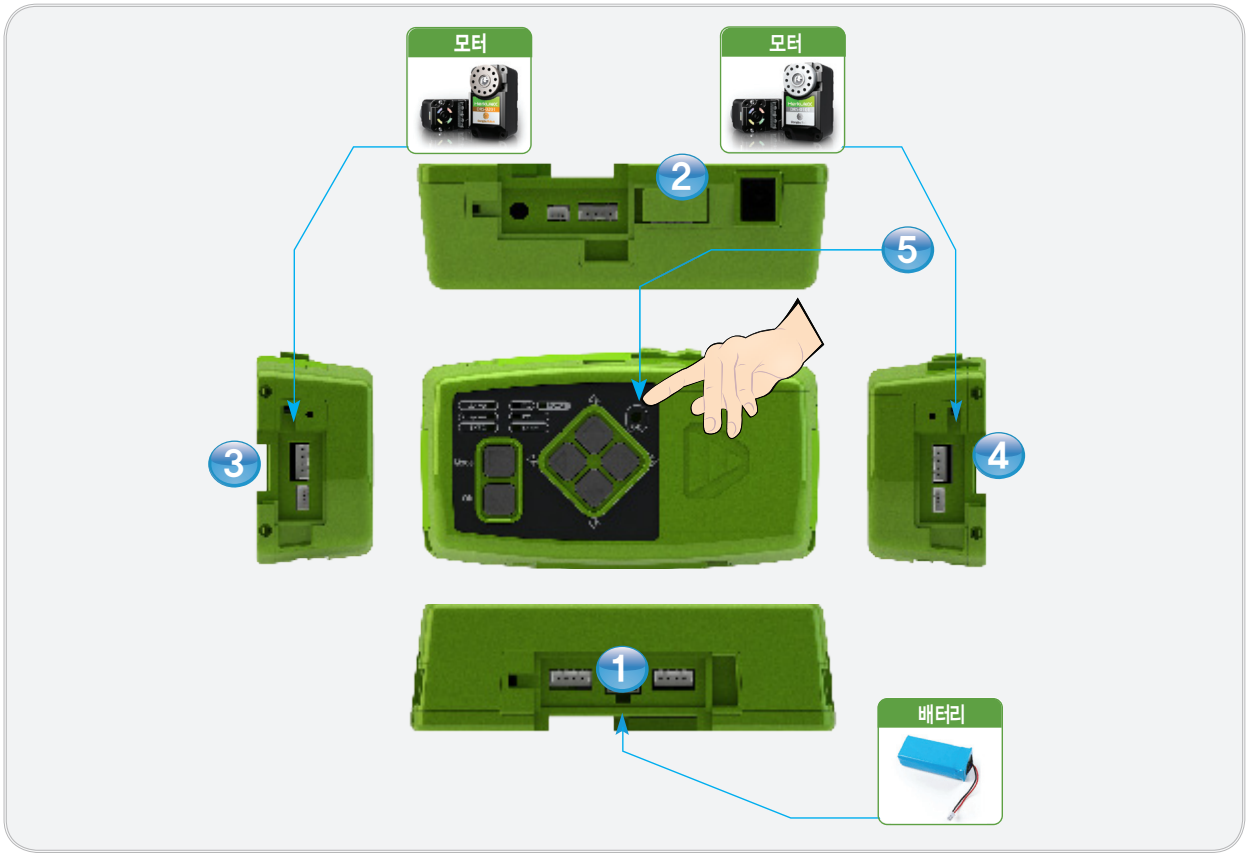
모터는 바깥쪽으로 놓고 테스트 합니다. 바퀴가 돌아가는 것을 가상한 시나리오입니다.

→ 테스트 진행

- 전원을 켜고 (상) 버튼을 눌러 Test Mode 진입후
- Navi Key 의 (좌) 버튼을 클릭합니다. → 왼쪽 모터가 동작합니다.
- Navi Key 의 (우) 버튼을 클릭합니다. → 오른쪽 모터가 동작합니다.
- Navi Key 의 (상) 버튼을 클릭합니다. → 모터 두개가 전진 방향으로 동작합니다.

상위와 같이 동작하였다면, 모터가 이상없이 동작하였음을 의미합니다. 모터테스트 완료합니다. 이후 모든 테스트의 결과는 위 두 모터가 제대로 동작하는지의 여부로 판단합니다. 모터를 분리하지 말고 센서 테스트를 계속 진행합니다.

DRC 기본 테스트 : 빛 센서



1 배터리 연결

2 전원켜기

3 좌측 모터 연결 : 반드시 Motor ID 1 번을 연결하여야 합니다.(다른 모터는 동작하지 않습니다.)

4 우측 모터 연결 : 반드시 Motor ID 2 번을 연결하여야 합니다.(다른 모터는 동작하지 않습니다.)

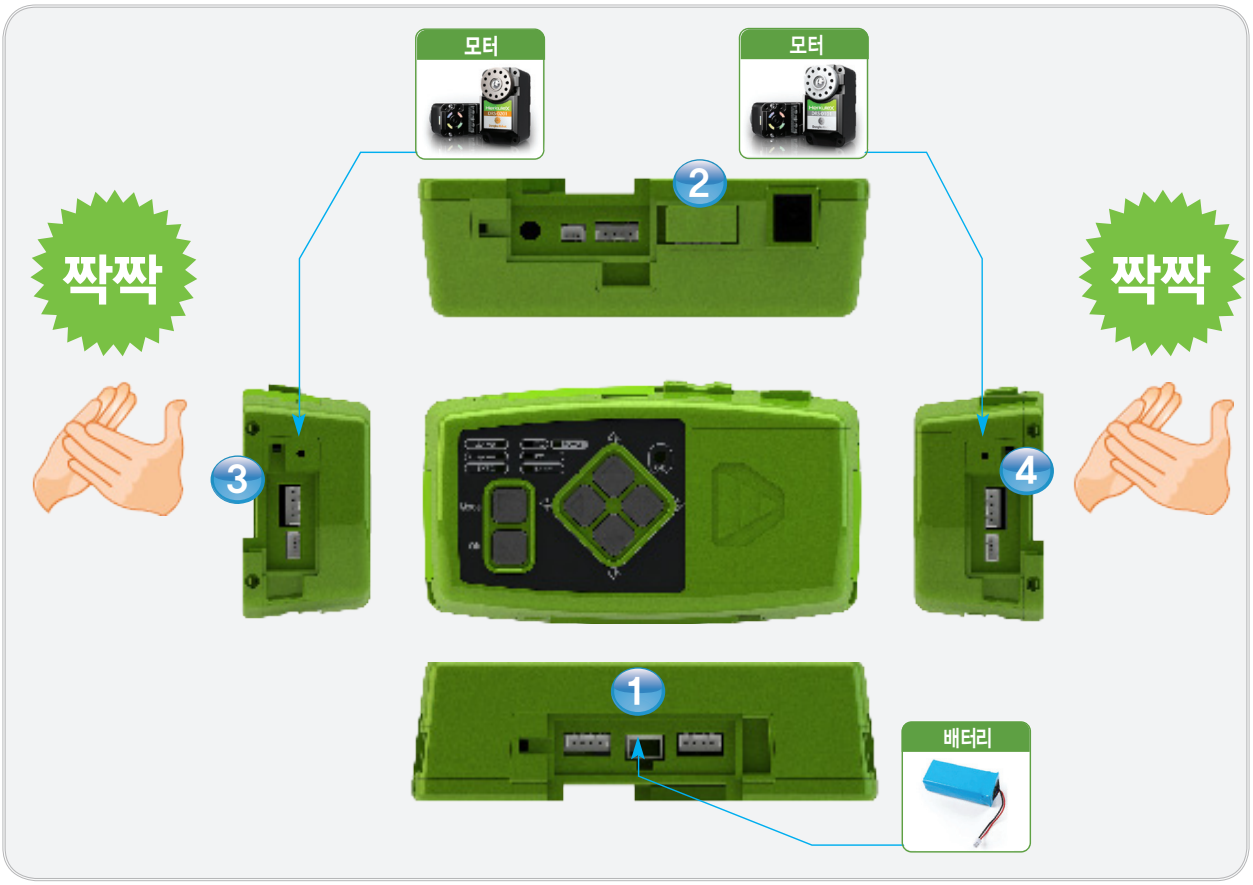
빛이 사라졌을 때, 로봇이 팔로 허공을 짚는 가상 시나리오입니다.

→ 테스트 진행

- 전원을 켜고 (상) 버튼을 눌러 Test Mode 진입후
- Cds 창을 손바닥으로 가립니다. → 좌우 모터가 동시에 움직입니다.

상위와 같이 동작하였다면, 빛센서가 이상없이 동작하였음을 의미합니다. 빛 센서 테스트를 완료합니다.

DRC 기본 테스트 : 소리센서



1 배터리 연결

2 전원켜기

3 좌측 모터 연결 : 반드시 Motor ID 1 번을 연결하여야 합니다.(다른 모터는 동작하지 않습니다.)

4 우측 모터 연결 : 반드시 Motor ID 2 번을 연결하여야 합니다.(다른 모터는 동작하지 않습니다.)

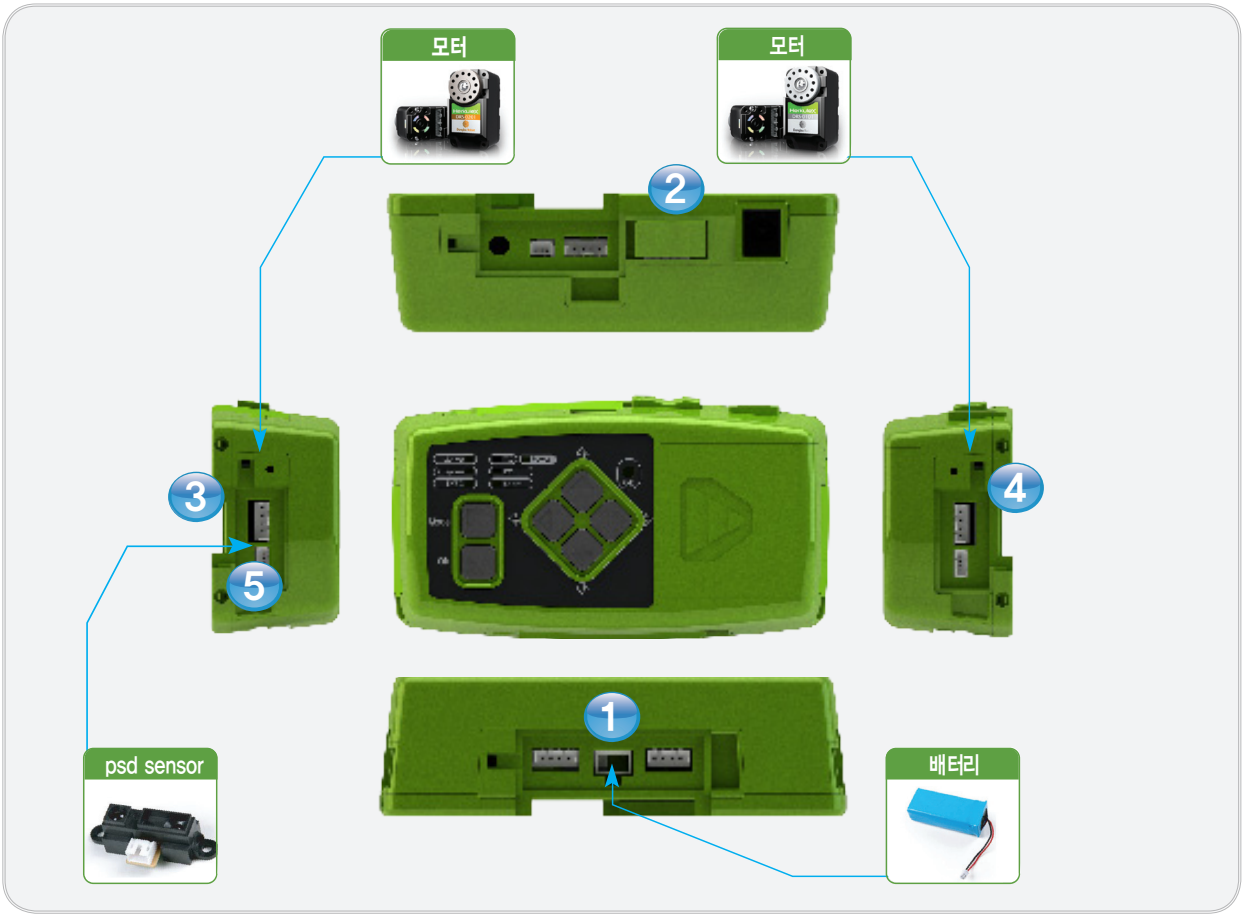
Sound Sensor 제어기 좌우에 위치합니다. 박수치는 방향의 모터가 움직이는 시나리오입니다.

→ 테스트 진행

- 전원을 켜고 (상) 버튼을 눌러 Test Mode 진입후 .
- (OK) 버튼을 클릭합니다. → Sound Sensor가 동작합니다.
- 왼쪽에 박수칩니다. → 왼쪽 모터가 움직입니다.
- 오른쪽에 박수칩니다. → 오른쪽 모터가 움직입니다.

상위와 같이 동작하였다면, 소리센서가 이상없이 동작하였음을 의미합니다. 소리센서테스트 완료합니다.

DRC 기본 테스트 : PSD Digital 거리센서



1 배터리 연결

2 전원켜기

3 좌측 모터 연결 : 반드시 Motor ID 1 번을 연결하여야 합니다.(다른 모터는 동작하지 않습니다.)

4 우측 모터 연결 : 반드시 Motor ID 2 번을 연결하여야 합니다.(다른 모터는 동작하지 않습니다.)

5 PSD Digital Sensor 연결

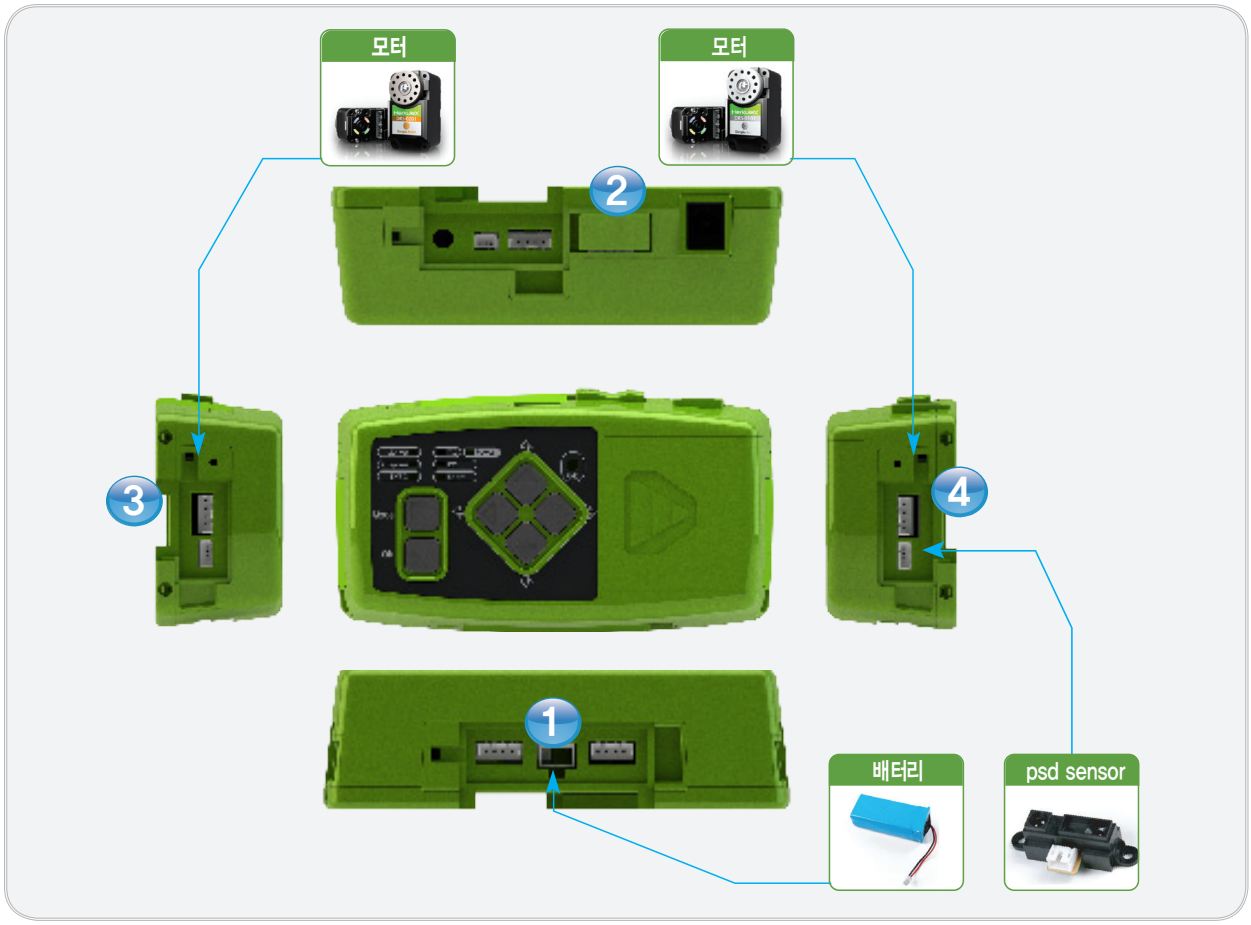
PSD Digital Sensor 는 일정거리를 기준으로 가깝냐 머냐를 체크하는 센서입니다. 흔히 땅의 깊이를 체크하여 낭떠러지를 감지했을 때 로봇을 멈추게 하는 시나리오입니다.

→ 테스트 진행

- 전원을 켜고 (상) 버튼을 눌러 Test Mode 진입후
- PSD 선을 연결하고 (하)버튼을 클릭합니다. → PSD Digital Sensor 가 동작합니다.
- 손바닥이나 물체를 센서 10cm 이하로 접근시킵니다. → 양쪽 모터가 모두 움직입니다.
- 손바닥이나 물체를 센서 10cm 이상으로 떨어뜨립니다. → 양쪽 모터가 모두 중지합니다.

* PSD Digital 은 기준위치를 기반으로 On/Off 형태로만 동작합니다.
상위와 같이 동작하였다면, PSD Digital Sensor 가 이상없이 동작하였음을 의미합니다. PSD Digital Sensor 테스트를 완료합니다.

DRC 기본 테스트 : PSD Analog 거리센서



1 배터리 연결

2 전원켜기

3 좌측 모터 연결 : 반드시 Motor ID 1 번을 연결하여야 합니다.(다른 모터는 동작하지 않습니다.)

4 우측 모터 연결 : 반드시 Motor ID 2 번을 연결하여야 합니다.(다른 모터는 동작하지 않습니다.)

5 PSD Analog Sensor 연결

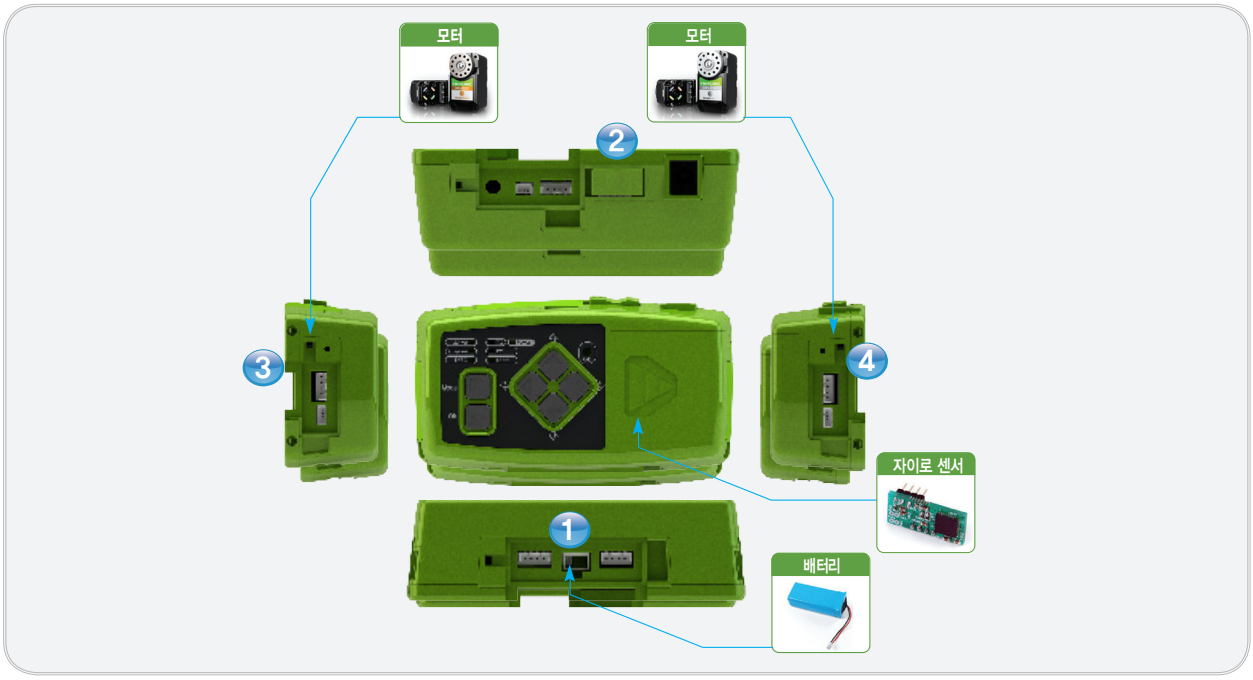
PSD Analog Sensor 는 실시간 거리 측정이 가능합니다. 접근거리별로 모터의 속도를 조절합니다. 대부분 장애물 감지에 쓰이며, 장애물이 가까이 왔을 때 속도를 줄이고, 회피하는 동작에 사용됩니다.

→ 테스트 진행

- 전원을 켜고 (상) 버튼을 눌러 Test Mode 진입후 PSD 선을 연결하고 (하)버튼을 클릭합니다. → PSD Digital Sensor 가 동작합니다. → 모터 두개가 같은 방향으로 움직입니다.
- 손바닥이나 물체를 PSD 센서 앞에 위치시켰다가 멀리 떨어뜨립니다. → 물체가 멀수록 모터는 더 빨리 움직입니다. → 물체가 가까우면 가까울수록 모터는 느리게 움직입니다.
- 물체가 5cm 이내로 접근하면 모터는 반대로 움직입니다.

상위와 같이 동작하였다면, PSD Analog Sensor 가 이상없이 동작하였음을 의미합니다. PSD Analog Sensor 테스트를 완료합니다.

DRC 기본 테스트 : 가속도/자이로센서



1 배터리 연결

2 전원켜기

3 좌측 모터 연결 : 반드시 Motor ID 1 번을 연결하여야 합니다.(다른 모터는 동작하지 않습니다.)

4 우측 모터 연결 : 반드시 Motor ID 2 번을 연결하여야 합니다.(다른 모터는 동작하지 않습니다.)

모터는 바깥쪽으로 놓고 테스트 합니다. 바퀴가 돌아가는 것을 가상한 시나리오입니다.

5 Acc/Gyro 센서 연결 : 로봇 제어기 뚜껑을 열고, 가속도/자이로 센서를 장착합니다.

→ 테스트 진행 : ACC

- 전원을 켜고 (상) 버튼을 눌러 Test Mode 진입후 Acc/Gyro 를 장착한 후 (하) 버튼을 누릅니다. → 가속도센서가 동작합니다.
- 제어기가 로봇 등에 붙은 각도데로 똑바로 선 상태이면 모터가 멈춰있습니다.
- 제어기를 서서히 기울입니다. → 기울기에 따라서 모터속도가 다르게 움직입니다. 더 많이 굽힐수록더 빠르게 움직입니다.

상위와 같이 동작하였다면, Acc Sensor 가 이상없이 동작하였음을 의미합니다.

→ 테스트 진행 : Gyro

- ACC테스트 모드에서 한번더 (하)버튼을 누릅니다. → 자이로센서 가 동작합니다.
- 제어기가 멈춰있을 때는 모터도 정지해있습니다.
- 제어기를 움직입니다. → 제어기가 돌아가는 속도와 비슷하게 모터 속도가 변합니다.

상위와 같이 동작하였다면, 가속도/자이로센서가 이상없이 동작하였음을 의미합니다. 가속도/자이로센서 테스트를 완료합니다.
Acc 와 Gyro 는 한 칩에 같이 모듈화 되어있습니다.

부품목록

HOVIS

Hovis Lite 구성도

1. 서보모터 : 허큘렉스 DRS-0101
2. 브라켓 : 서보모터간 연결 관절 역할
3. 조인트 : 모터와 브라켓간 연결
4. 하네스 : 케이블
5. PSD 센서 : 거리측정
6. IR수신기 : 리모콘 수신기
7. 제어기 DRC : 메인 컨트롤러
8. Gyro 센서 : 보행중 자세 교정 (옵션) - 별도판매
9. ZigBee : 통신모듈 (옵션) - 별도판매
10. 배터리 : 3,000m 리튬폴리머
11. 리모콘

Hovis Lite 부품목록

부품별 수량입니다.

아래 체크 박스(☐) 에 체크하면서 부품 수량을 확인하세요

그림은 뒷장 사진을 참조하세요.

메인 부품	Controller	1개	☐
	7.4V Li-Po Battery	1개	☐
	AC adaptor	1개	☐
	HerkuleX DRS-0101	16개	☐
브라켓	Front / Back	4개	☐
	Hand	4개	☐
	Foot	4개	☐
	Universal Plate	40개	☐
	Ankle Plate	8개	☐
	Dummy Servot	4개	☐
	U1-type Bracket	6개	☐
	U2-type Bracket	8개	☐
	U3-type Bracket	6개	☐
	U4-type Bracket	8개	☐
악세사리	Serial Cable (DSUB 9Pin/3p Audio Jack)	1개	☐
	USB to Serial Gender	1개	☐
	Wheel (White, Ø60)	4개	☐
	Horn(Plastic)	5개	☐

조인트	L-type Joint (Single Nut)	70개	☐
	L-type Joint (Double Nut)	10개	☐
	L-type Joint (Hole Only)	40개	☐
	I-type Joint (10.9mm,Hole only)	10개	☐
	I-type Joint (12.5mm,Double Nut)	60개	☐
	I-type Joint (16.0mm,Hole only)	20개	☐
	V-type Joint (12.0mm,Single Nut)	20개	☐
	V-type Joint (12.0mm,Double Nut)	20개	☐
	Bushing Set	30개	☐
볼트 & 너트 & 하네스	Nut (M2)	40개	☐
	Nut (M3)	10개	☐
	Bolt (PH/T 2.0X13)	25개	☐
	Bolt (PH/M 2.0X4)	50개	☐
	Bolt (PH/M 2.0X5)	20개	☐
	Bolt (PH/M 2.0X6)	150개	☐
	Bolt (PH/M 2.0X8)	20개	☐
	Bolt (PH/M 3.0X6)	10개	☐
	Bolt (PH/M 3.0X8)	20개	☐
	Bolt (PH/T 2.0X4)	40개	☐
	Bolt (PH/T 2.0X5)	60개	☐
	Harness (75mm)	4개	☐
	Harness (100mm)	4개	☐
	Harness (200mm)	6개	☐
	Harness (300mm)	2개	☐
	Harness Clamp	20개	☐

부품목록

HOVIS

Hovis Lite 부품목록

메인 부품

DRC-005T 1개입
Controller

DRL-0728 1개입
7.4V Li-Po Battery
(3,000mA)

DRQ-0001 1개입
AC Adaptor

DRS-0101 16개입
HerkuleX Smart Servo

볼트 & 너트 & 하네스

DRA-0002 4개입
Harness (75mm)

DRA-0003 4개입
Harness (100mm)

DRA-0006 6개입
Harness (200mm)

DRA-0007 2개입
Harness (300mm)

DRA-0051 40개입
Nut (M2)

DRA-0052 10개입
Nut (M3)

DRA-0054 25개입
Bolt (PH/T 2.0X13)

DRA-0056 50개입
Bolt (PH/M 2.0X4)

DRA-0057 20개입
Bolt (PH/M 2.0X5)

DRA-0058 50개입
Bolt (PH/M 2.0X6)

DRA-0059 20개입
Bolt (PH/M 2.0X8)

DRA-0061 10개입
Bolt (PH/M 3.0X6)

DRA-0062 20개입
Bolt (PH/M 3.0X8)

DRA-0063 40개입
Bolt (PH/T 2.0X4)

DRA-0064 60개입
Bolt (PH/T 2.0X5)

DRJ-0010 20개입
Harness Clamp

부품목록

HOVIS

Hovis Lite 부품목록

브라켓

DRB-0001 4개입
Front / Back

DRB-0002 4개입
Hand

DRB-0003 4개입
Foot

DRB-0004 40개입
Universal Plate

DRB-0005 8개입
Ankle Plate

DRB-0006 4개입
Dummy Servo

DRB-0008 6개입
U1-type Bracket

DRB-0009 8개입
U2-type Bracket

DRB-0010 6개입
U3-type Bracket

DRB-0011 8개입
U4-type Bracket

조인트

DRJ-0001 70개입
L-type Joint (Single Nut)

DRJ-0002 10개입
L-type Joint (Double Nut)

DRJ-0003 40개입
L-type Joint (Hole only)

DRJ-0004 10개입
I-type Joint
(10.9mm, Hole only)

DRJ-0006 60개입
I-type Joint
(12.5mm, Double Nut)

DRJ-0007 20개입
I-type Joint
(16.0mm, Hole only)

DRJ-0008 20개입
V-type Joint
(12.0mm, Single Nut)

DRJ-0009 20개입
V-type Joint
(12.0mm, Double Nut)

DRJ-0011 30개입
Bushing Set

악세사리

DCW-0001 4개입
Wheel (White, Ø60)

DRI-0002 1개입
Serial Cable
(DSUB 9Pin-3P Audio Jack)

DRI-0003 1개입
USB to Serial Gender

DRH-1001 5개입
Horn (Plastic)

Humanoid 조립도

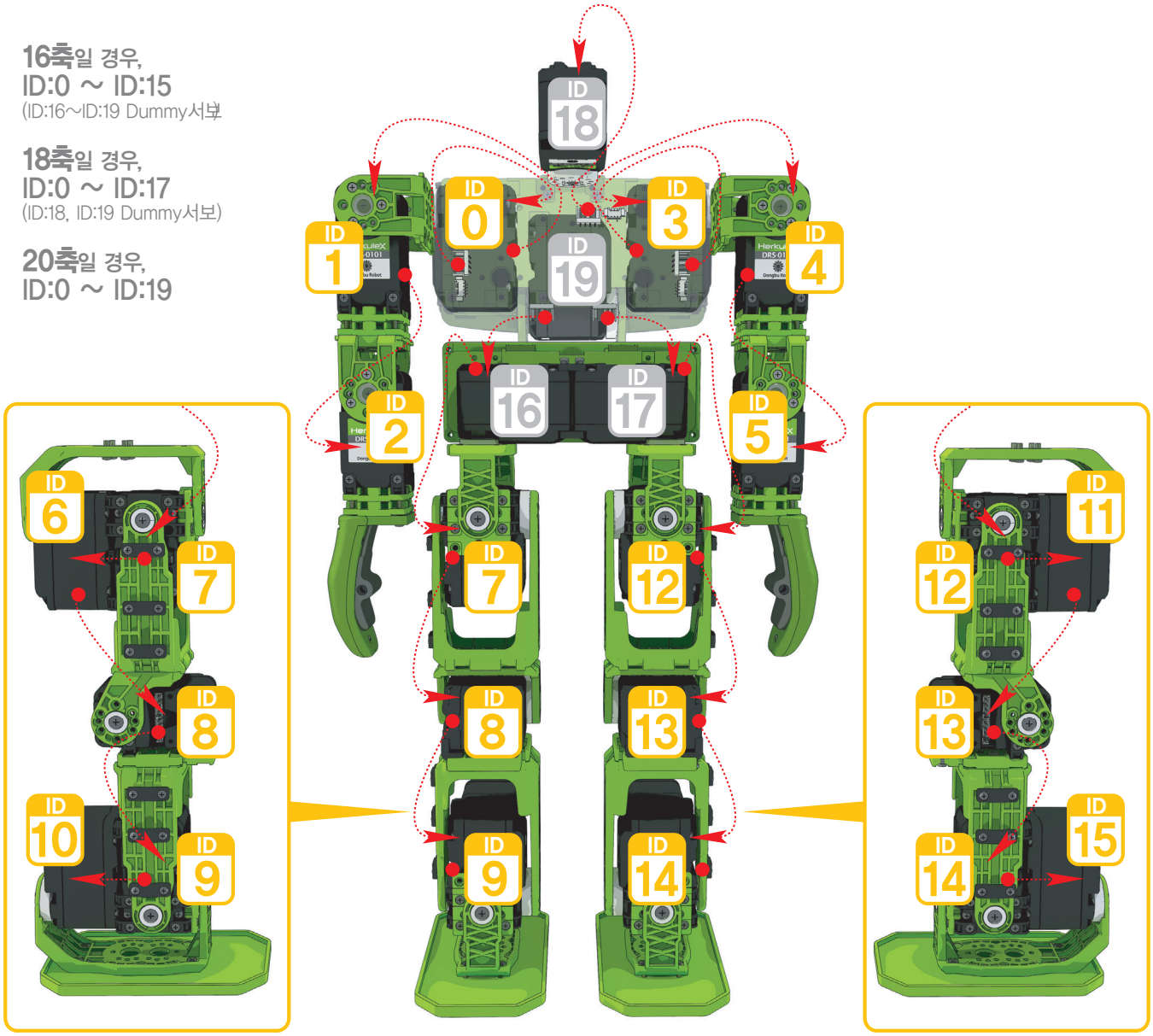
휴머노이드 ID 배치와 배선도입니다.

휴머노이드 ID는 제품 출하시 각각의 서보모터에 ID를 부여하여 제공됩니다. 모터 겉면에 쓰인 ID 번호데로 조립을 해야만 로봇이 제대로 동작할 수 있습니다. ID 배치 기준은 20축 기준이고, 16~19번은 더미모터와 교체용이기 때문에 가장 마지막 번호로 배치되었습니다. 휴머노이드 조립시 가장 난이도가 높은 파트가 서보간의 배선문제입니다. 전체 연결에 대한 개념을 익히신 후에 세부적으로 하나씩 시도해 보기 바랍니다. 상세 배선 연결은 휴머노이드 조립도를 참조하세요.

16축일 경우,
ID:0 ~ ID:15
(ID:16~ID:19 Dummy서보

18축일 경우,
ID:0 ~ ID:17
(ID:18, ID:19 Dummy서보)

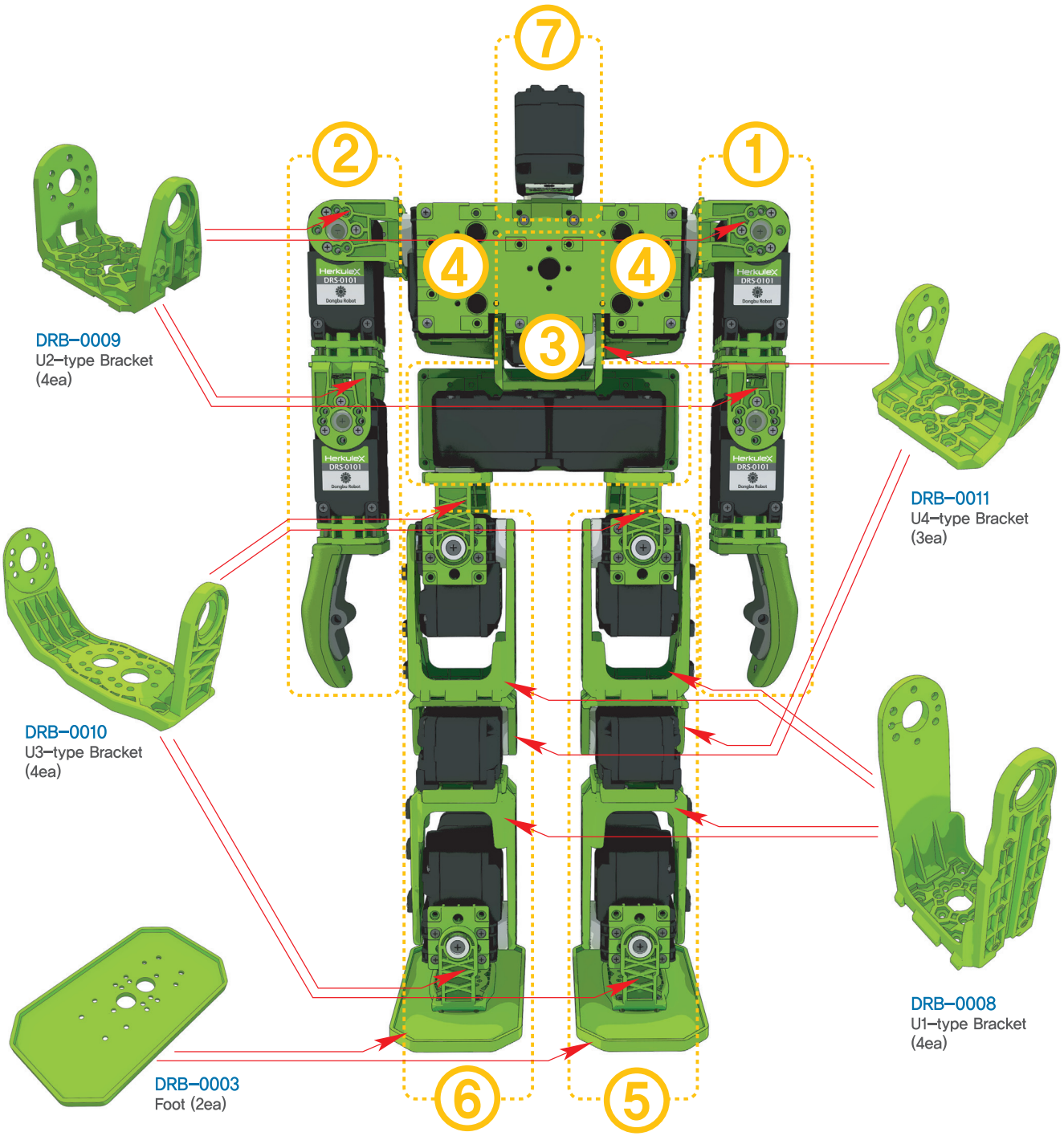
20축일 경우,
ID:0 ~ ID:19



Humanoid 조립도

휴머노이드 브라켓 배치도입니다.

브라켓의 용도는 서보간 연결 및 로봇의 관절역할을 합니다. 어깨,허리,엉덩이,무릎,발바닥으로 구성되며, 전체 브라켓 배치도를 익힌 후 세부적인 조립에 들어갑니다. 세부 조립방법은 휴머노이드 조립도를 참조하세요



Humanoid DRC 기능 사용법

제어기 켜기

파워표시	잔여량이 20% 미만일때 깜박거림
배터리 잔량표기	전원을 켜고 (좌)버튼을 눌러 배터리 잔량 확인 가능 → 좌측 LED 3 개로 표시 가장 낮을 때 1 개, 중간일 때 2 개, 가장 높을 때 3 개
Task진입	Mode버튼을 눌러 기본 Task를 시작함 Navi key → Ok버튼으로 원하는 모드 선택

로봇 동작

중간조립 점검모드	Mode → (하) → OK : 중간 조립 점검 모드조립 중간 중간마다 잘 조립되었는지 확인 오른팔, 왼팔, 오른다리, 왼다리, 센서
모터확인모드	Mode → (좌) → OK : 모터확인모드 – 실행하면 모터 힘이 풀림, 모터가 내려가면서 하나씩 선택됨. 선택된 모터는 LED가 깜박임다. – (상) 모터 ID 1씩 증가, (하) 모터 ID 1씩 감소 – 해당 ID의 모터가 존재하지 않을 경우 경고음이 울림.
자율보행모드	Mode → (상) → OK : 자율보행모드 : 로봇이 움직이면서 스스로 동작 – 박수를 치면 박수 치는 방향으로 횟수만큼 이동(동작중 박수치면 반응 안함) – 5초동안 일정한 반응 없으면 기본 동작 시작 – 기본동작 : 앉았다 일어남, 앞뒤로 걷기, 방향 틀며 걷기 – 장애물 피하기(PSD 센서 필요) – 넘어졌을때 스스로 일어남(자이로 센서 필요)
리모콘 조정 모드	Mode → (우) → OK : 리모콘 조정 모드 리모콘 번호(0~9번)와 방향키(상하좌우멈춤)에 정의된 동작하기

프로그램 다운로드

	LED 동작	LED 표기
HerkuleX 연결	HerkuleX Manager 가 실행중일 때 깜박거림	Servo
DR-SIM/ Visual Logic 연결	DR-SIM / Visual Logic 을 이용하여 편집 사용중일 때 깜박거림 데이터나 펌웨어를 다운로드 할 때 켜져있음	Program
Task를 로봇에실행	Mode 버튼을 눌러 Task모드로 진입시 Task가 실행중인 동안 켜져 있음	EXEC
데이터 송신	외부로 데이터 송신할 때 깜박거림, task 실행중일때는 User Spare 공간	TX
데이터 수신	외부로부터 데이터 수신할 때 깜박거림, task 실행중일때는 User Spare 공간	RX
사용자 할당		Spare
에러발생	실행중 에러가 생긴 경우 모든 LED 깜박거리면서 알림	모든 LED 깜박

Humanoid DRC 기능 사용법

로봇동작

점검 모드	**Mode → (좌) → OK로 점검 모드로 진입합니다.** **초록색 RX LED가 켜지면 (좌) 혹은 (우) 버튼을 눌러 어떤 점검을 할 것인지 선택합니다.** **(좌) : 모터 확인 모드** – 모터를 하나씩 선택하며 연결 여부와 조립 상태를 확인하는 모드입니다. – 모터 확인 모드 진입 시 빨간색 TX LED가 켜집니다. – 선택된 ID의 모터가 초록색 LED가 켜지며 중간 위치(512)로 움직입니다. 나머지 모터는 LED가 꺼지며 힘이 풀립니다. – (상) (하) 버튼으로 ID를 0~15까지 변경하며 각 모터의 연결 여부와 조립 위치를 확인합니다. 처음 선택된 ID는 0번(오른쪽 어깨)입니다. – (상) 버튼을 누르면 ID가 1 증가, (하) 버튼은 1 감소 합니다. – 해당 ID의 모터가 존재하지 않을 경우 경고 버저 음이 울립니다. **(우) : 중간 조립 점검 모드** – 팔, 다리 등을 알맞게 조립하였는지 모듈 별로 테스트할 수 있는 모드입니다. – 중간 조립 점검 모드 진입 시 파란색 Spare LED가 켜집니다. – 선택된 모듈에 해당하는 모터들이 곧게 편 자세와 구부린 자세를 왕복하며 천천히 움직입니다. – (상) (하) 버튼으로 어떤 팔과 다리를 움직일 것인지 선택할 수 있습니다. – 왼팔, 오른팔, 왼다리, 오른다리의 순서로 선택됩니다. 처음 선택된 모듈은 왼팔입니다. – 선택한 모듈에 있어야 할 ID가 모두 연결되지 않았다면, 연결되지 않은 ID 수와 같은 횟수만큼 버저 음을 울려 알려줍니다.
자율 행동 모드	**Mode → (상) → OK로 자율 행동 모드로 진입합니다.** – 로봇이 스스로 행동을 하며 작동하는 모드입니다. – 평상시 상태에서, 로봇은 직진, 앞구르기, 좌회전, 우회전 중 하나를 랜덤으로 선택하여 동작합니다. – 직진은 10/20/30걸음 중에서 랜덤으로 선택되며, 좌/우회전도 12/24/36걸음 중에서 랜덤으로 선택됩니다. – 선택된 동작을 마치면 잠시 멈추었다가 다음 동작을 선택하여 동작합니다. – PSD 센서가 ADC 포트 1번에 장착되어 있다면 장애물을 인식하고 회피할 수 있습니다. – 동작 중에 장애물을 인식하면 로봇은 후진 후 좌회전, 좌회전, 뒤구르기 후 좌회전 중 하나를 랜덤으로 선택하여 동작합니다. – 뒤구르기 후 좌회전은 직진을 10걸음 이상 하던 도중 장애물을 만났을 때만 가능합니다. – 좌회전을 많이 반복해도 장애물을 피하지 못한다면, 다시 후진 후 좌회전을 시도합니다. – 가속도 센서가 제어기에 장착되어 있다면 넘어짐을 감지하고 일어날 수 있습니다. – 동작 중에 넘어진 경우 자동으로 방향을 인식하여 기존 모션을 멈추고 일어나기모션으로 전환합니다.

Humanoid DRC 기능 사용법

로봇동작

리모컨 조종 모드	**Mode –〉 (우) –〉 OK로 리모컨 조종 모드로 진입합니다.** – 리모컨으로 로봇의 행동을 조종하는 모드입니다. 리모컨 리시버 모듈이 연결되어 있어야 동작합니다. – 상 : 전진 – 하 : 후진 – 좌 : 좌회전 – 우 : 우회전 – OK : 정지 – 1 : 앞구르기 – 2 : 뒤구르기 – 3 : 푸쉬업 – 4 : 복싱 – 5 : 앞으로 일어나기(가속도 센서가 장착되어 있고, 실제로 누운 상태에서만 가능) – 6 : 뒤로 일어나기(가속도 센서가 장착되어 있고, 실제로 엎드린 상태에서만 가능)
사운드 데모 모드	**Mode –〉 (하) –〉 OK로 사운드 데모 모드로 진입합니다.** – 소리 횟수와 방향 인식을 통해 로봇이 다른 반응을 보이는 모드입니다. – 제어기의 TX, RX, Spare LED가 켜진 상태가 소리 인식 중인 상태입니다. – 소리 인식 중인 상태에서, – 1번 소리 입력 시 : 앞구르기, 뒤구르기, 푸쉬업, 복싱 중 한 모션을 랜덤으로 실행합니다. – 2번 소리 입력 시 : 소리가 난 방향의 팔을 들어올려 웨이브를 합니다. 가령 왼쪽에서 한번, 오른쪽 에서 한번 입력 되었다면 왼쪽 팔, 오른쪽 팔 순으로 웨이브를 합니다. – 3번 소리 입력 시 : 소리가 난 방향으로 회전 후 10걸음 직진합니다. 왼쪽에서 소리가 나면 좌회전 후 직진합니다. **※ 소리 방향 인식은 주변 환경의 소음이나 벽에 의한 소리의 반사 등에 영향을 많이 받아 정확하게 동작하지 않을 수 있습니다. 또한, 박수 소리 같이 짧고 큰 소리가 인식하기 쉽습니다.**

PART 02

DR-Visual Logic Programming

PART 02

소프트웨어 : DR-SIM

HOVIS

설치하기

DR-SIM 소개

DR-SIM 은 로봇 모션을 쉽게 편집할 수 있는 소프트웨어 툴입니다. Motion Editor 로 불리어지는 DR-SIM은 초보자도 쉽게 사용할 수 있습니다. DR-SIM은 강력한 시뮬레이션 기능을 지원합니다. 사용자는 이 기능을 통해 실제 로봇에 적용된 모션을 사전에 시뮬레이션 할 수 있습니다. 이것이 기존 Motion Editor 와의는 가장 큰 차별점 입니다. 뿐만아니라, DR-SIM 기존의 Motion Editor 와 같이 실제 로봇을 조작하여 모션을 만들 수 있는 기능도 지원합니다. DR-SIM은 타임라인을 제공하여 동영상 편집기와 같은 직관적인 UI 를 제공합니다. 사용자는 시간에 따라 로봇 모션을 만드는게 가능합니다. 머리쪽의 LED 편집이나 사운드도 모션과 같이 타임라인에 삽입하여 편집할 수 있어 멀티미디어적인 편집이 가능합니다. 마지막으로, DR-SIM은 멀티미디어 적인 편집을 지원합니다. 로봇머리의 LED및 타임라인 상의 사운드 기능을 이용하여 로봇 모션의 멀티미디어 효과를 더하세요.

- **시스템 요건**
- **최소 인텔 팬티엄 800 Mhz**
- **Windows XP, Windows Vista, Windows 7**
- **최소 256 MB RAM**
- **하드디스크 설치 공간 300 MB 필요**
- **USB Port**
- **Macintosh(지원 예정)**

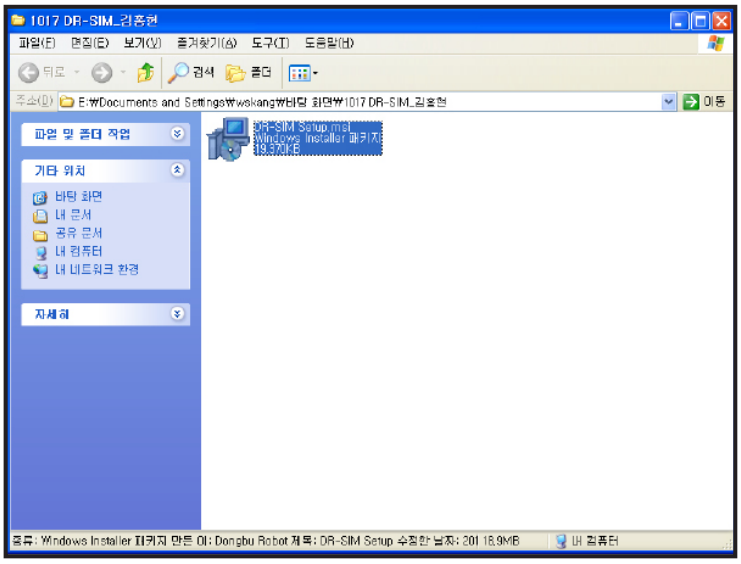
PART 02

소프트웨어 : DR-SIM

설치하기

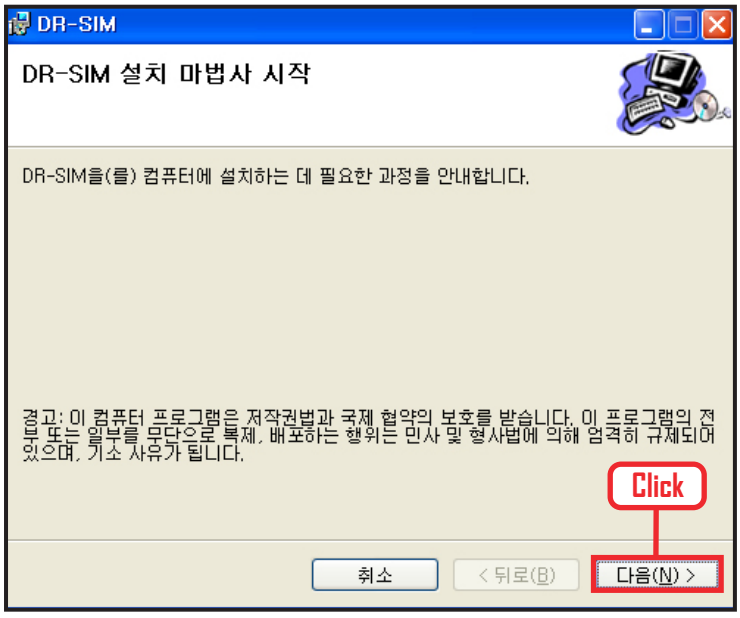
따라하기

설치부터 실행까지 따라해보세요



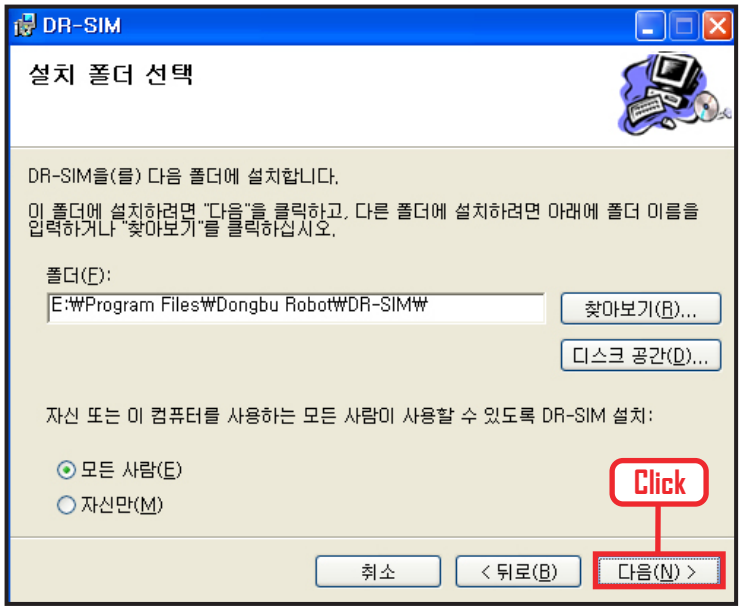
01 설치파일

설치파일을 클릭합니다.



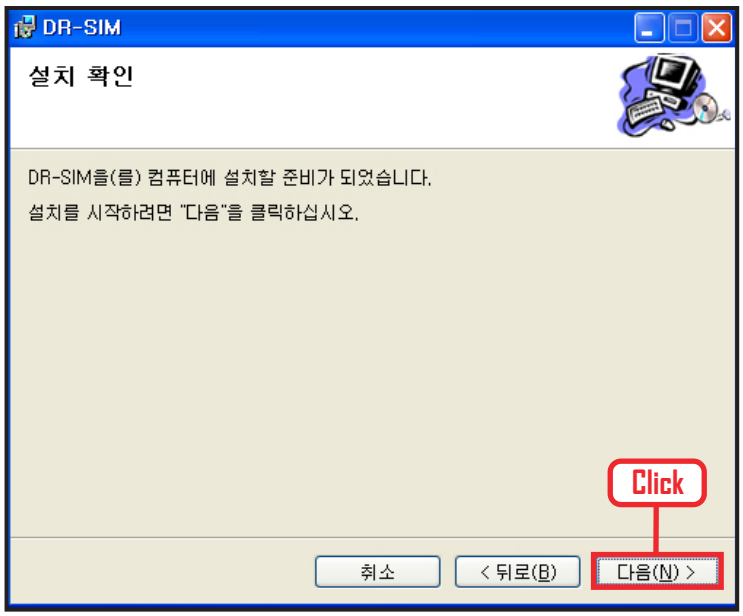
02 설치 마법사 시작

"다음" 버튼을 클릭합니다.



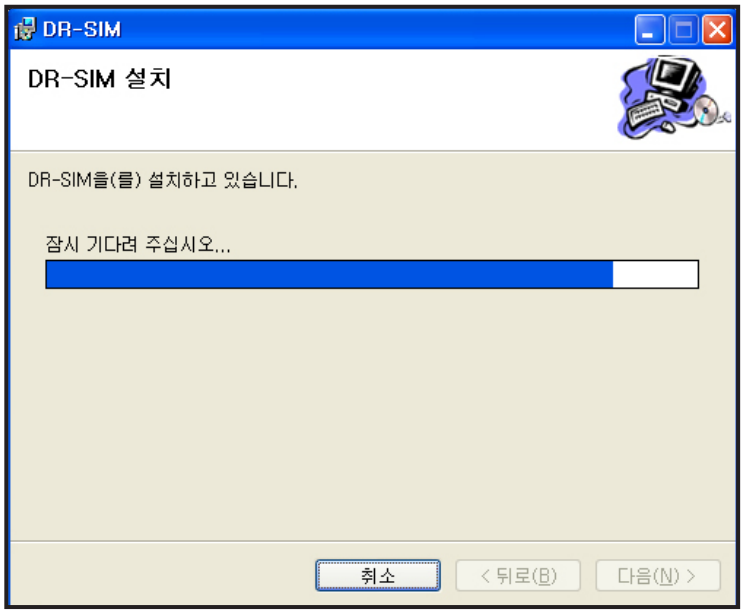
03 설치 폴더를 선택

"다음" 버튼을 클릭합니다.



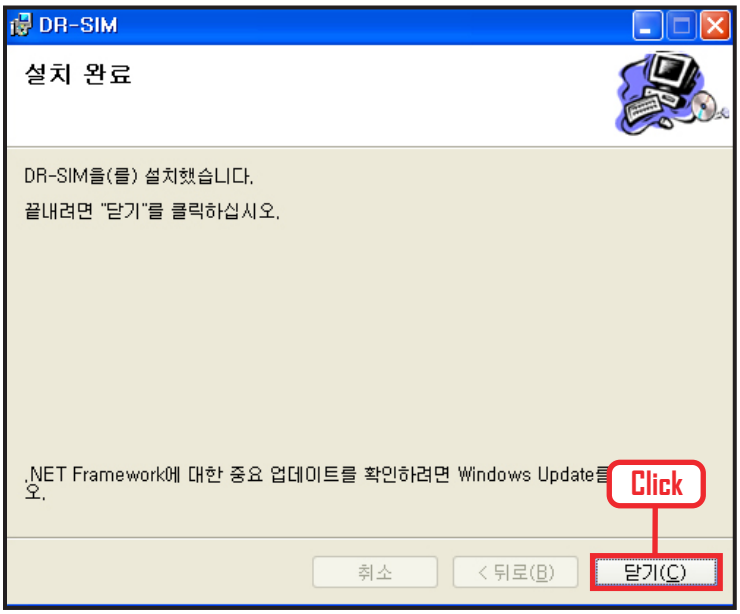
04 설치 확인

"다음" 버튼을 클릭합니다.



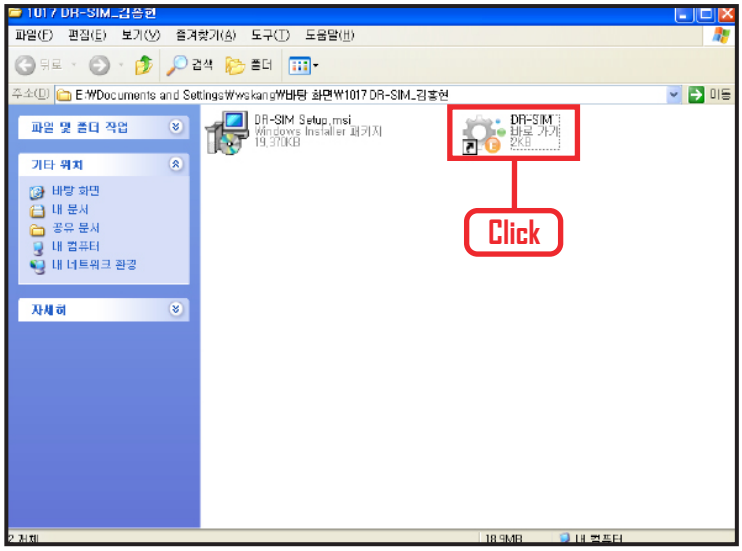
05 설치 시작

설치시작 합니다. 프로그레스 바가 끝날때 까지 기다려주세요



06 설치 완료 선택

"닫기" 를 클릭하세요
소프트웨어 설치가 완료되었습니다.



07 실행파일 확인

바탕화면의 바로가기 아이콘 및 Windows 시작 〉 모든프로그램 〉 Dongbu Robot 〉 DR-SIM에서 실행파일을 확인하세요.
실행파일을 클릭하면 프로그램이 실행됩니다.

설치가 되지 않을경우 Microsoft, Net Frame work 3.5를 설치하시고, 다시 시도해 보세요.

PART 02

소프트웨어 : DR-SIM

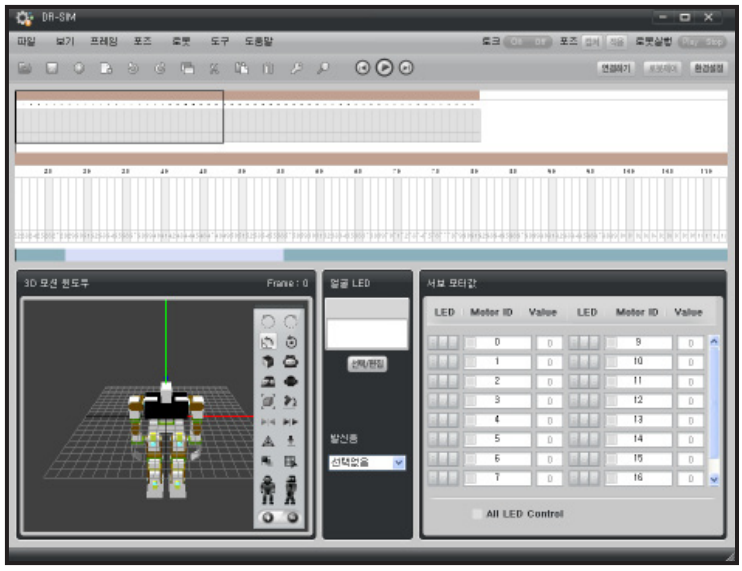
HOVIS

Hello DR-SIM

다음은 로봇 모션을 만드는 첫번째 예제입니다. DR-SIM을 이용하여 간단한 모션을 편집한 후 이를 시뮬레이션 합니다.
로봇에 연결하여 편집된 모션 File을 다운로드 합니다. 그리고, 로봇에서 수행되는 모션을 확인합니다.

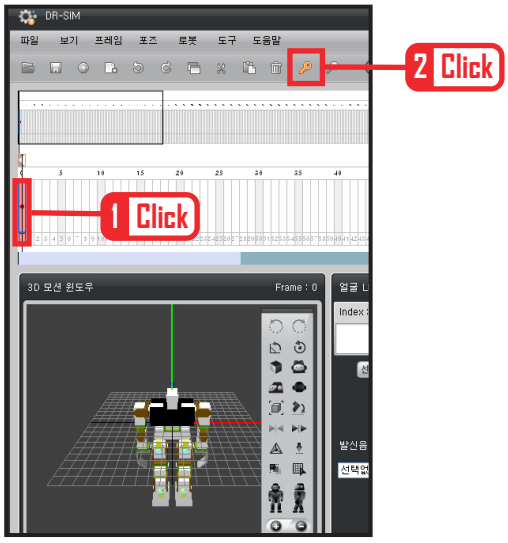
00 실행

DR-SIM 실행 아이콘을 클릭하여 프로그램을 실행합니다.



01 전체 화면

DR-SIM의 전체화면입니다.
중간에 타임라인과 하단의 모션 편집이 있습니다.
주로 화면 중간의 타임라인과 하단의 3D 모션윈도우를 통해 모션을 편집합니다.

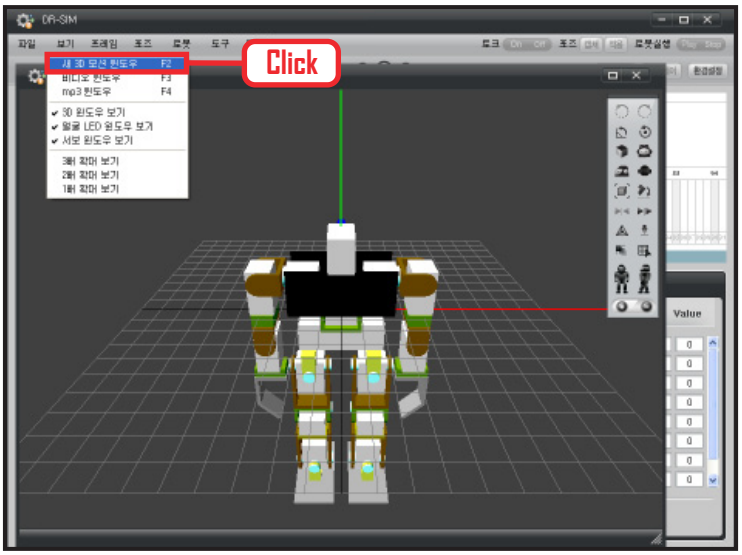


02 기본자세

로봇동작의 시작점이나, 첫 프레임에 로봇의 기본 동작을 삽입합니다.

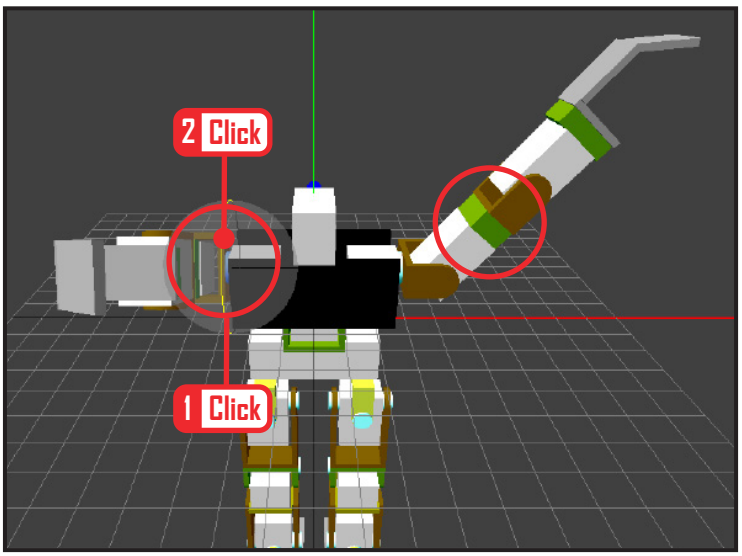
3D 모션 윈도우의 동작을 기본자세로 위치시킴 → 첫 프레임을 클릭 → 상단의 키 프레임 삽입(열쇠 아이콘)을 클릭
기본 자세가 첫 번째 키 프레임에 삽입되었습니다.

3D 모션 윈도우의 자세가 기본 자세가 아니면 우측 툴바에서 기본자세로 설정한 후 진행합니다.(단축키 Alt + I)



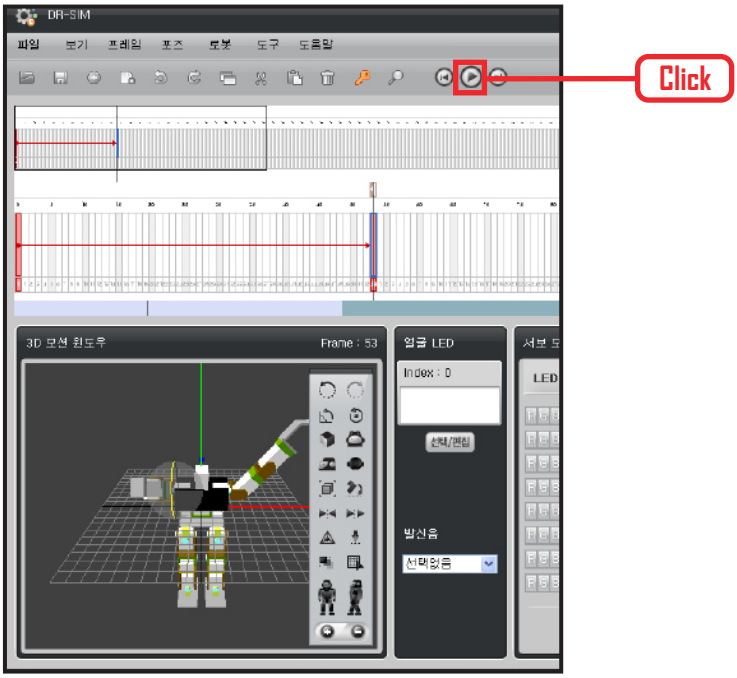
03 3D 윈도우 창

3D 윈도우 창을 크게 보기 위해서는 **메뉴 〉 보기 〉 새 3D 윈도우 보기** 를 클릭합니다.
새로운 3D 윈도우 창이 뜨면 화면을 드래그해서 확대하여 봅니다.



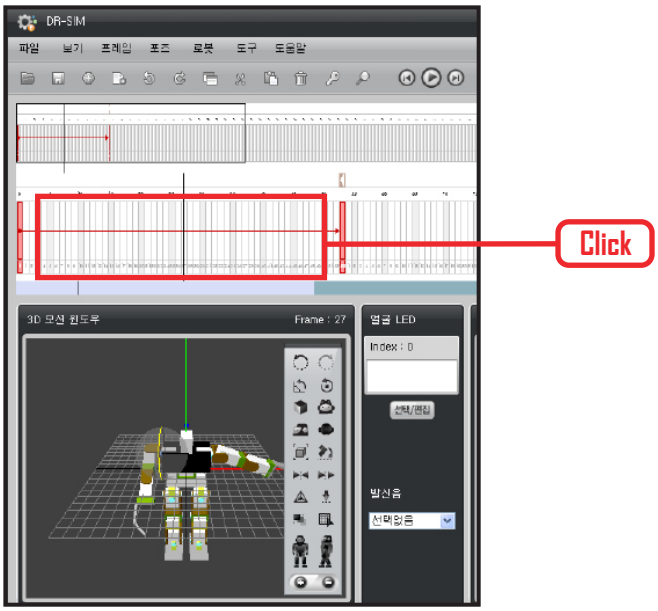
04 모션 편집

로봇관절을 클릭하면 관절 이동선이 노란색 실선으로 표시됩니다.
그 라인을 따라 마우스의 왼쪽를 누른 상태로 드래그합니다.
로봇의 왼손은 좌상측으로 올리고, 로봇은 오른손은 앞상측으로 올립니다.



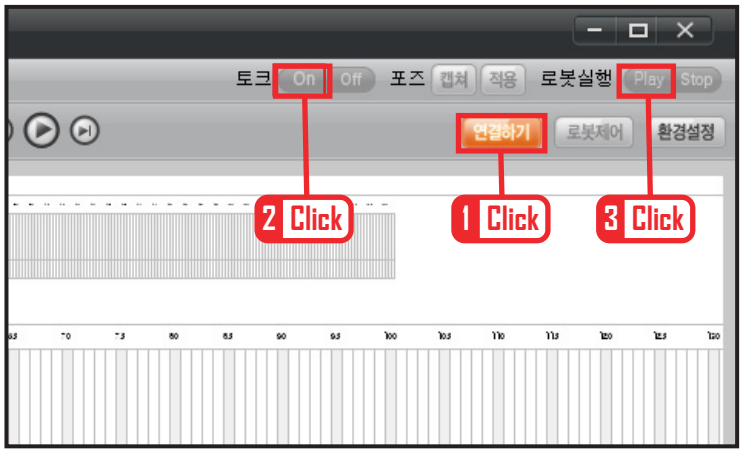
05 편집 모션 삽입

3D 창에서 편집된 로봇 모션을 원하는 타임 프레임에 삽입합니다.
삽입 후 상단의 "▶"버튼을 클릭합니다.
로봇이 동작하는 것을 3D 창에서 시뮬레이션으로 확인합니다.



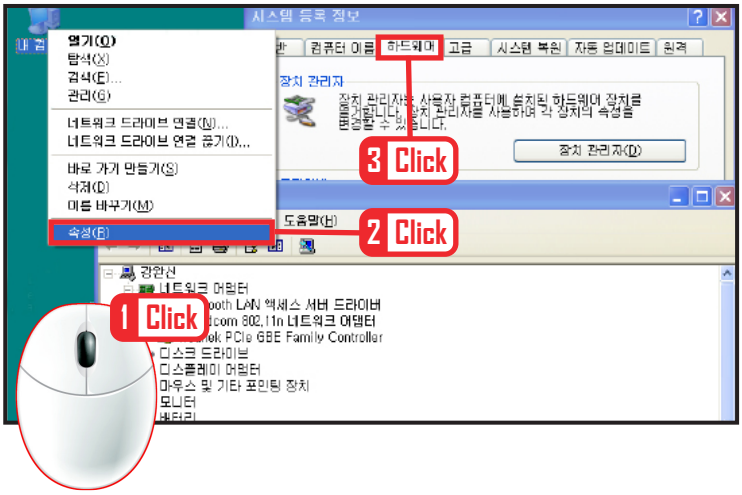
06 삽입 프레임 중간 확인

기본 모션과 편집된 모션 중간 타임라인을 클릭합니다.
두 프레임간의 중간동작을 확인해 볼 수있습니다.



07 로봇과 연결하기

PC나 노트북 USB 단자에서 USB to Serial 컨버터를 이용하여 로봇에 연결합니다.
"연결하기" 아이콘을 클릭하면 로봇과 연결됩니다.
연결되지 않으면 COM port 를 확인합니다.
"토크 ON" 버튼을 클릭한 후 로봇의 토크상태를 확인하여 움직이지 않으면 토크가 걸린 것입니다.
"로봇실행 Play" 버튼을 클릭하면 로봇이 편집 모션처럼 움직입니다.
이로써 첫 번째 모션 편집과 로봇에서의 실행을 완료했습니다.



08 참조 : COM Port 설정

로봇이 연결되지 않을 시에는 COM 포트 설정이 잘못되었기 때문입니다.
"내컴퓨터" 오른쪽 마우스 **"속성"** 을 클릭하면 **"시스템 등록정보"**가 나옵니다. **"하드웨어"** 탭을 클릭하여 장치관리자를 엽니다.
장치관리자의 COM 포트를 클릭하면 현재 설정 가능한 COM 포트 종류가 나옵니다.
그중에 USB 에 연결된 COM2 를 선택하고 저장하면 로봇과 연결된 COM 포트가 열리게 됩니다.

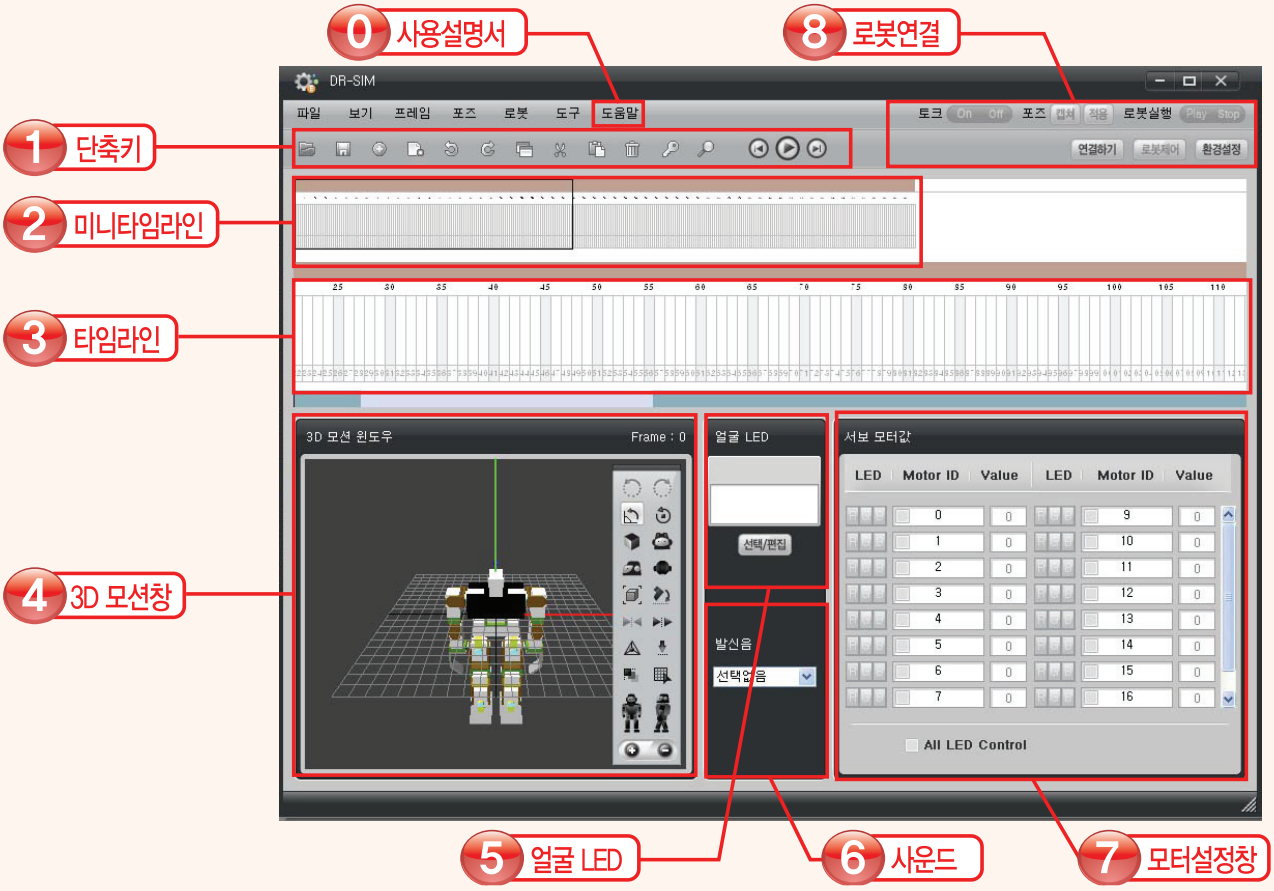
09 참조 : 환경설정 COM 포트 설정

통신설정에서 COM2 를 선택하고 저장합니다.
로봇과 PC 소프트웨어가 연결됩니다.

PART 02

소프트웨어 : DR-SIM

사용자 인터페이스



0 사용설명서 : DR-SIM 의 상세 사용법이 나와있습니다. F1 을 누르면 도움말 바로가기가 실행됩니다.

1 단축키 : 사용 빈도가 높은 메뉴를 모았습니다. 시뮬레이션 플레이, 키프레임 삽입 등이 있습니다.

2 미니 타임라인 : 타임프레임의 전체 아웃라인을 볼 수 있습니다.

3 타임라인 : 로봇 모션을 편집 후 시간별로 모션을 배치할 수 있습니다.

4 3D 모션창 : 시뮬레이션 창으로서 모션 편집 및 로봇 모션이 가상으로 움직이는 것을 볼 수 있습니다.

5 얼굴LED : 로봇 얼굴 쪽의 LED 를 편집할 수 있는 창입니다. 편집 후 타임프레임에 삽입합니다.

6 사운드 : 저장 된 사운드를 지정할 수 있습니다. 사운드 선택 후 타임프레임에 삽입합니다.

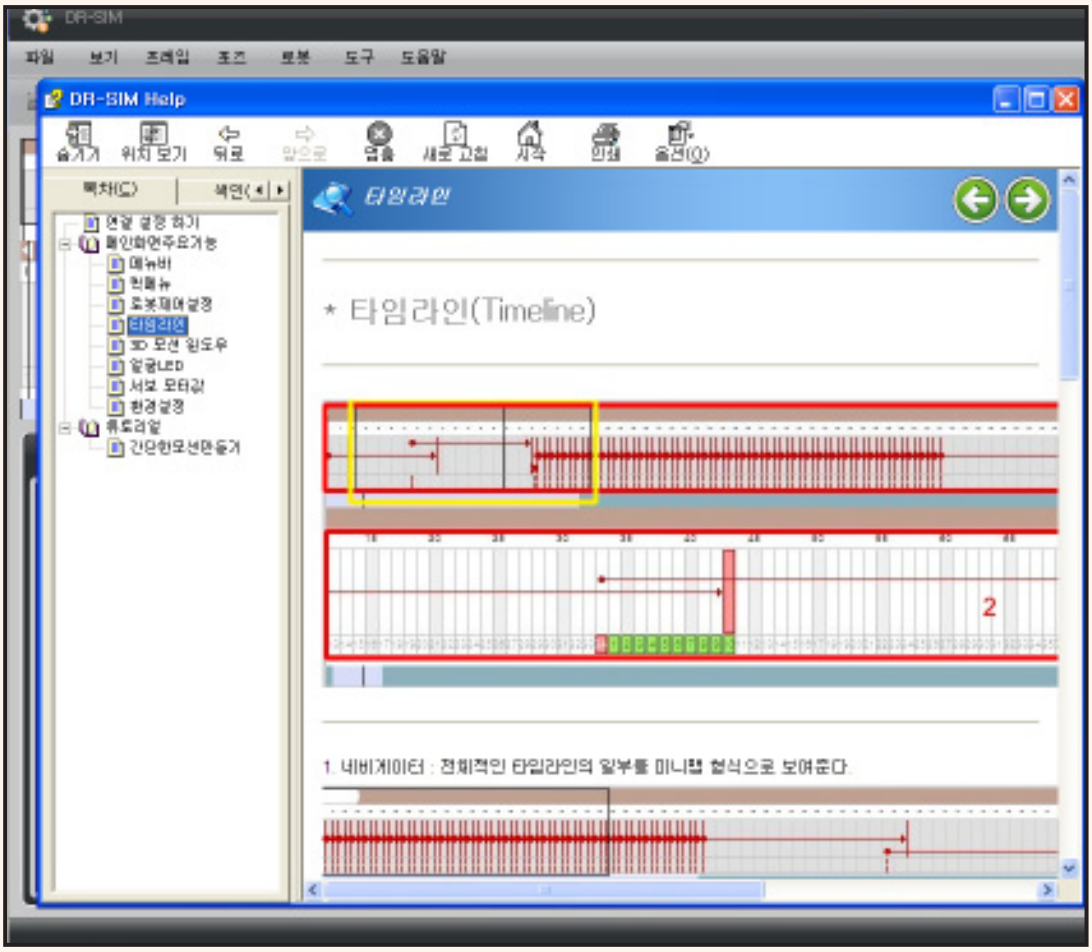
7 모터설정창 : 로봇에 사용된 모든 모터 아이디별로 값 및 LED 를 설정할 수 있습니다.

8 로봇연결 : 로봇과 연결하기 단축키입니다. 모션 편집된 파일을 로봇에 다운로드 하거나 실제 로봇이 취하고 있는 포즈를 캡쳐하는 용도로 사용되어집니다.

소프트웨어 : DR-SIM

도움말 보기

메뉴에서 도구 〉 도움말을 클릭하면 아래와 같은 도움말 팝업창이 뜹니다. DR-SIM 을 사용하기 위해서는 도움말을 먼저 읽어보신 후 사용하기를 권합니다. **(상단 도움말 클릭 〉 내용색인 클릭 〉 좌측메뉴 타임라인 클릭 → 아래 그림이 뜸)**



- **연결설정하기 :** COM 포트 연결하는 방법을 설명합니다.
- **메인화면 주요기능 :** 프로그램 기능에 대한 사용법을 설명합니다.

  메뉴바
  퀵메뉴
  로봇제어설정
  타임라인
  3D 모션 윈도우
  얼굴 LED
  서보 모터값
  환경설정

- **튜토리얼**

  간단한 모션만들기 : 기본 모션 만들기 예제에 대한 설명입니다.

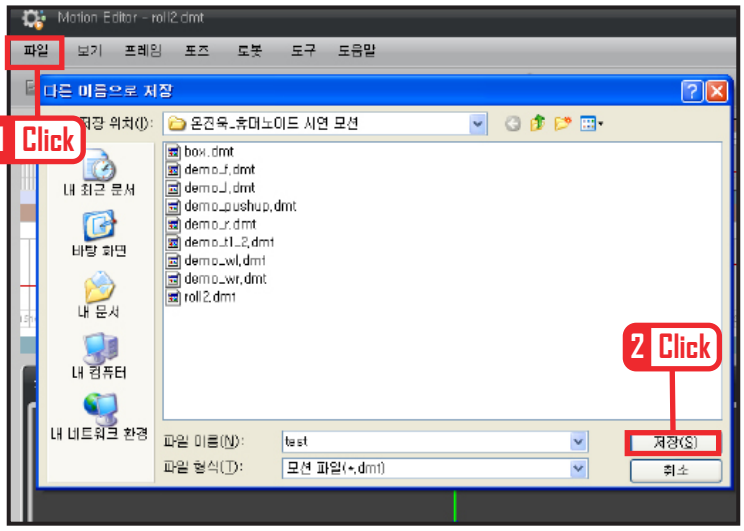
  다양한 모션 예제소스는 웹사이트에서 제공합니다.

PART 02

소프트웨어 : DR-SIM

다운로드 하기

편집한 로봇 모션은 파일로 저장됩니다, 저장된 로봇 모션 파일들은 한꺼번에 제어기 DRC에 다운로드가 가능합니다. (이때 제어기 DRC에 있던 이전 로봇 모션은 삭제됩니다.) 다운로드된 로봇 모션 파일들 제어기 DRC에 다운로드 된 순서에 따라 번호가 부여됩니다. 설정된 번호는 DR-Visual Logic(Task Editor) 프로그램에서 불러와 사용합니다.



로봇모션 저장

편집된 로봇 모션은 **파일 〉 다른이름**으로 저장 메뉴를 클릭하여 제목을 붙인 후 저장폴더에 저장합니다.

01 로봇제어

로봇과 DR-SIM 이 연결된 상태에서 **"로봇제어"** 아이콘을 클릭합니다.

02 로봇제어창

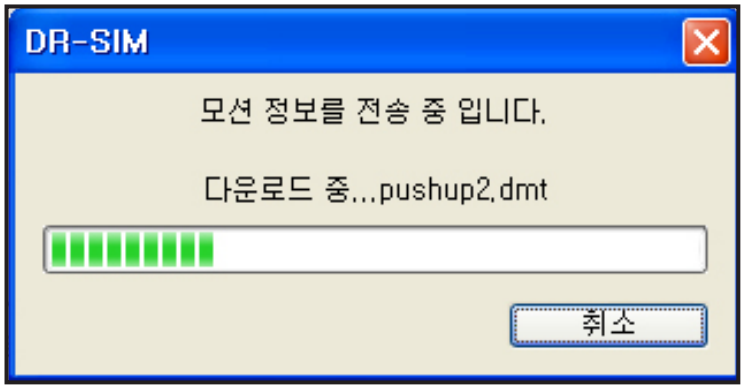
모션 정보 다운로드 팝업이 뜹니다. 모션이 저장된 폴더경로가 있고, 왼쪽은 PC 에 저장된 모션리스트가 있습니다. 오른쪽은 제어기에 다운로드하면 로봇모션이 번호순으로 리스트업됩니다.

03 저장 폴더 열기

폴더경로의 우측을 클릭하면 폴더 찾아보기 팝업창이 뜹니다. 저장된 폴더를 선택합니다.

04 모션 리스트

선택된 폴더에 있는 모션 리스트가 보여집니다. 리스트에 커서를 올려놓고 다운로드 아이콘을 클릭합니다.



05 다운로드

로봇모션이 하나씩 다운로드 됩니다.

06 컨트롤러 모션 목록

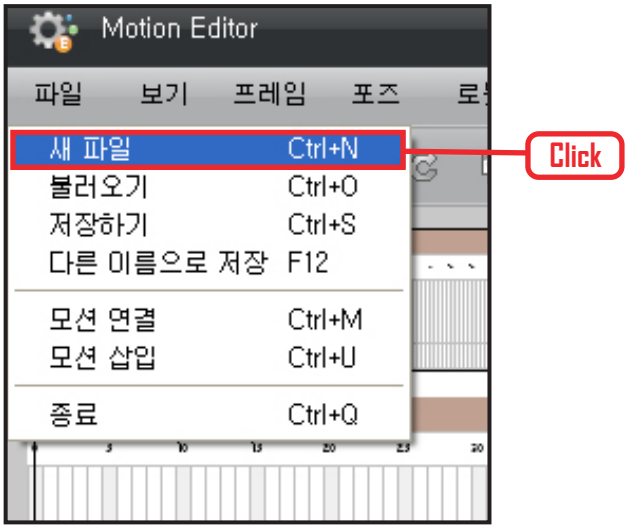
모든 모션이 다 다운로드되면, 오른쪽의 컨트롤러 모션 목록에 0번부터 모션이 리스트업됩니다.
번호는 DR-Visual Logic (Task Editor) 에서 프로그래밍시 인덱스로 불러올 수 있습니다.
이로써 로봇 모션의 다운로드가 완료되었습니다.

소프트웨어 : DR-SIM

모션 만들기

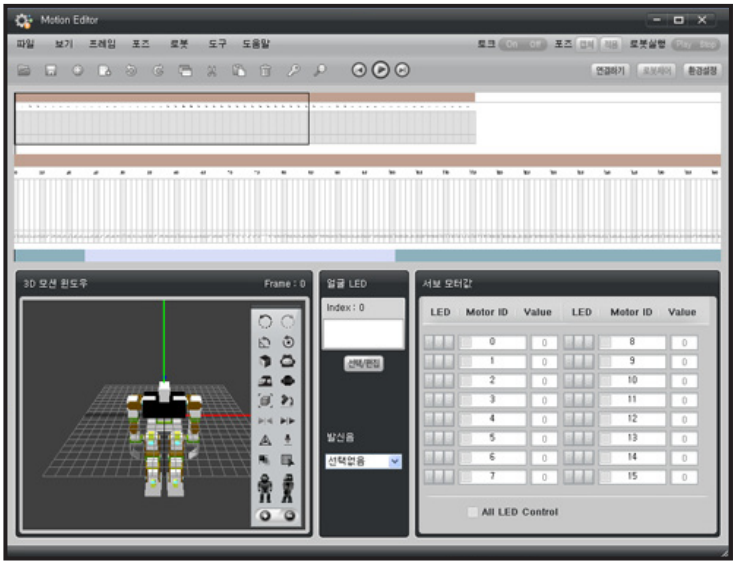
모션만들기 따라하기

모션만들기는 크게 두가지 방법이 있습니다. 3D 모션윈도우 창에서 관절마다 클릭하여 이동선 라인에 모션을 만드는 방법과 로봇 하드웨어 자체에 어떤 동작을 만들어서 캡쳐하는 방법이 있습니다. 이 두가지 방법을 활용하여 간단한 모션을 만들어 봅니다.



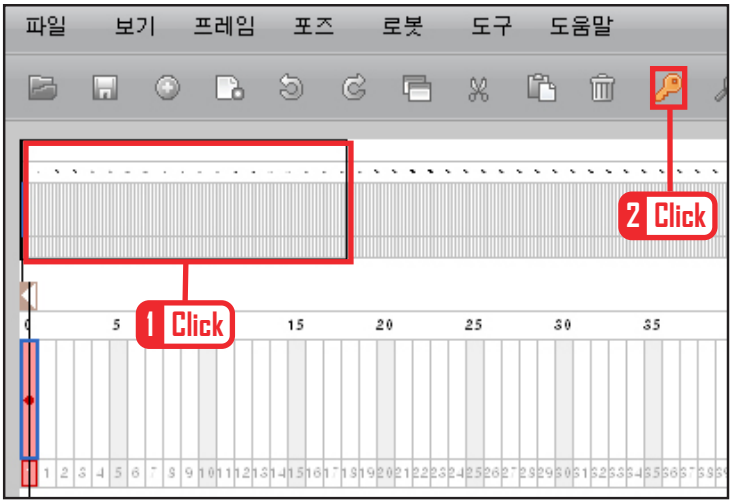
01 새파일

파일 〉 새파일 을 클릭합니다.



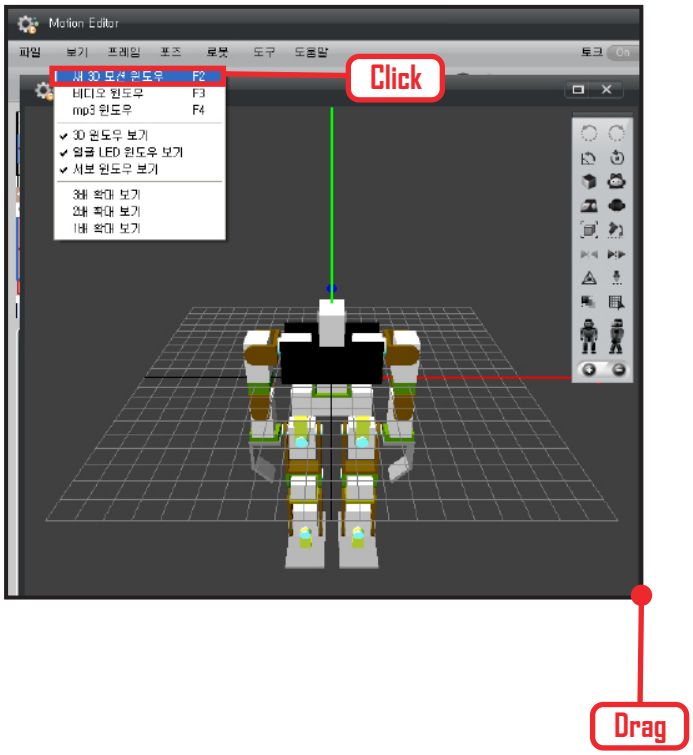
02 새 모션 편집창

이전 편집창이 없어지고, 새로운 로봇 3D모션창이 뜹니다.



03 첫 프레임

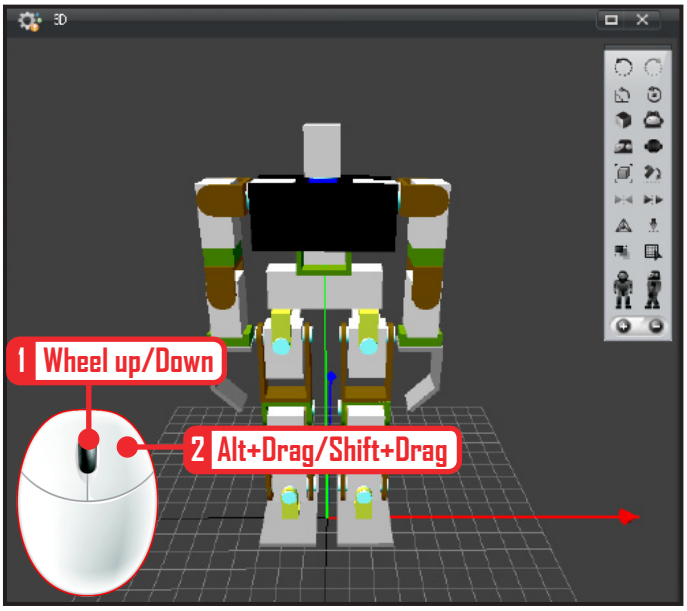
로봇 기본 자세를 첫 프레임에 삽입합니다.
첫번째 프레임을 클릭한 후 키프레임 삽입을 클릭합니다.



04 새 모션창

화면에서 모션을 편집하기 위해선 모션창을 확대할 필요가 있습니다. 모션창만 따로 띄워서 확대하여 편집하도록 하겠습니다. 3D 모션창은 최대 3개까지 동시에 띄울 수 있으면 각각 상,하,옆면 상태로 놓고 편집할 수 있습니다.

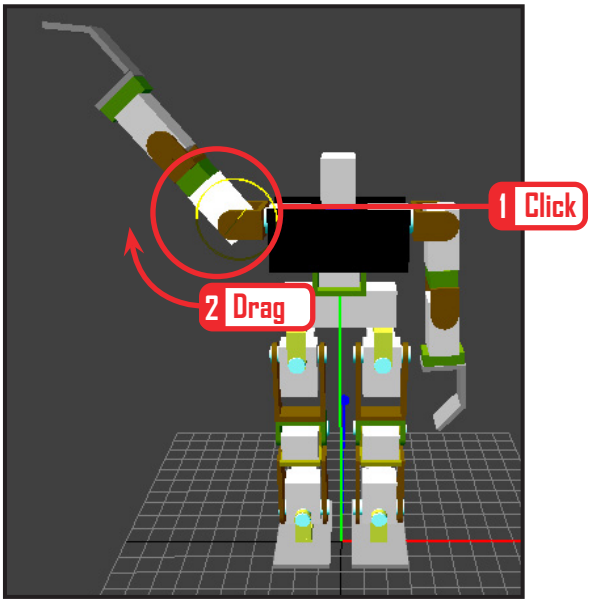
보기 〉 새 3D 모션윈도우 클릭합니다.
팝업 모션창이 뜨면 오른쪽 하단 끝 부분을 오른쪽 마우스로 잡고 드래그하여 화면을 확대합니다.



05 로봇 확대

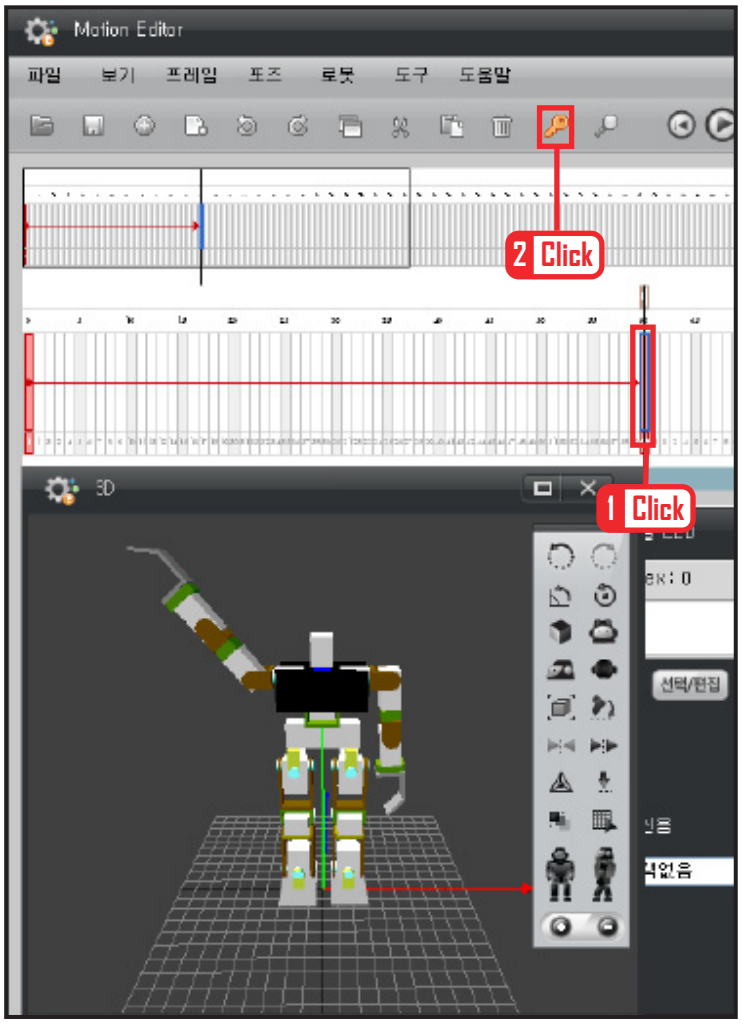
편집창 안의 로봇을 확대합니다.
빈 공간에 마우스를 클릭하고, 마우스 휠로 로봇을 확대 축소할 수 있습니다. 편집하기 좋게 로봇을 확대합니다.

원하는 각도를 조절하고 싶으면, 마우스 오른쪽 버튼을 누른 상태에서 드래그 합니다. 로봇의 위치는 (Shift+마우스우측버튼) 드래그하여 조절합니다.



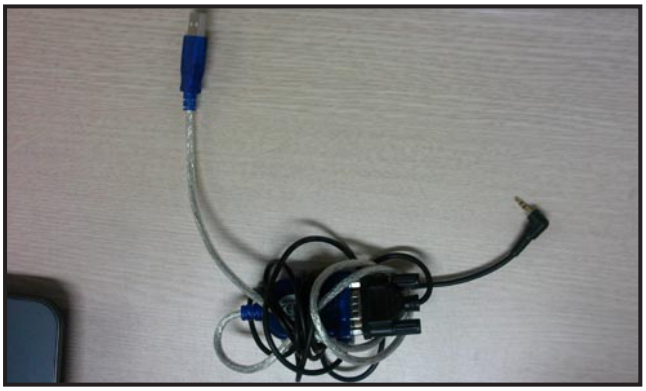
06 팔 모션 편집

팔을 올려봅니다. 어깨부분을 클릭합니다. 클릭하면 노란색으로 이동가능선이 표시됩니다. 이동선을 따라 드래그하면 팔이 올라갑니다.



07 키 프레임 삽입

팔 올린 모션을 프레임에 삽입합니다. 원하는 프레임을 클릭한 후 키 아이콘을 클릭하여 모션을 삽입합니다.



08-1 USB to Serial 컨버터

로봇과 연결을 시작합니다.
PC의 USB Port에서 로봇과 연결시켜주는 USB to Serial 컨버터 입니다.

08-2 USB port

PC는 뒤편에 Serial Port 가 있기 때문에 바로 연결 가능하지만, 노트북과 같이 Serial Port 가 없으면 USB to Serial 컨버터 를 이용하여 연결합니다.

08-3 로봇과 연결

RS232c 오디오잭을 로봇에 연결합니다.

08-4 로봇 Port

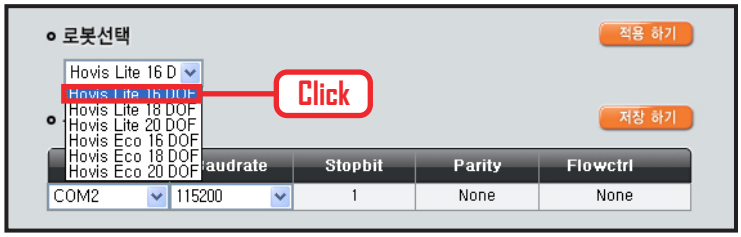
제어기 DRC 상단을 보면, PC와 연결하는 Serial Port 와 머리쪽 서보모터 Port, 그리고 전원Port가 있습니다.
모두 연결된 사진입니다.

08-5 로봇 연결 버튼

로봇과 연결하는 메뉴는 DR-SIM 오른쪽 상단에 있습니다.
COM Port를 설정하기 위해 환경설정을 클릭합니다.

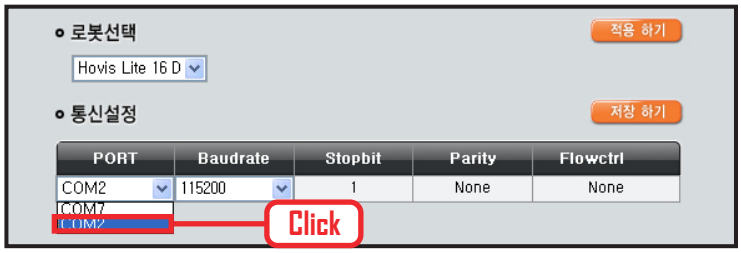
09 환경설정

DR-SIM version 과 로봇선택, 통신설정 등을 할 수 있는 설정창입니다.



10 로봇 선택

DR-SIM 은 총 6개 타입 휴머노이드를 제공합니다.
그 중 가장 기본은 16축 휴머노이드 입니다.
사용자가 조립한 로봇의 형태를 선택합니다.
Hovis Lite 16 DOF 를 선택합니다.

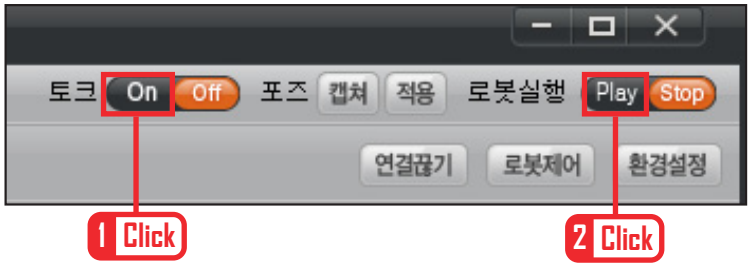


11 COM Port 선택

"PORT" 에 연결할 수 있는 COM Port 번호가 나옵니다. 그 중 한 개를 선택합니다. 연결이 되지 않으면 하드웨어 속성에서 활용할 수 있는 COM Port 번호를 확인합니다.

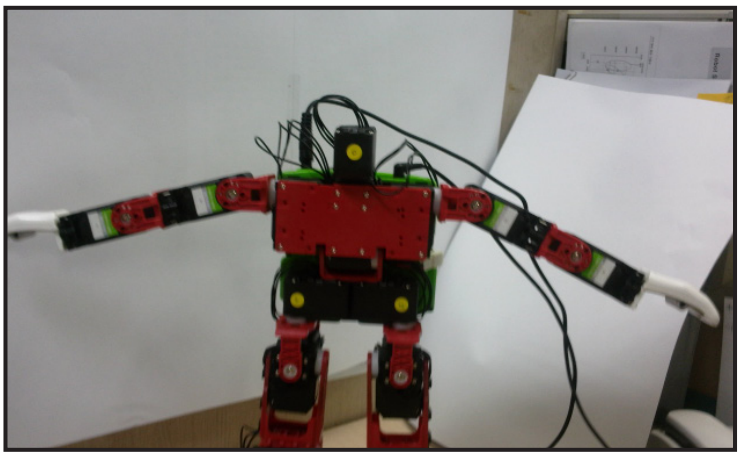
12 연결하기

연결하기 아이콘 클릭합니다.



13 연결

연결이 완료되면 좌측 그림과 같이 토크버튼이 활성화 됩니다.
로봇을 동작시키기 위해선 "토크 On" 버튼을 클릭합니다. 현재 모션 플레이 하기 위해선 play 버튼 클릭합니다.

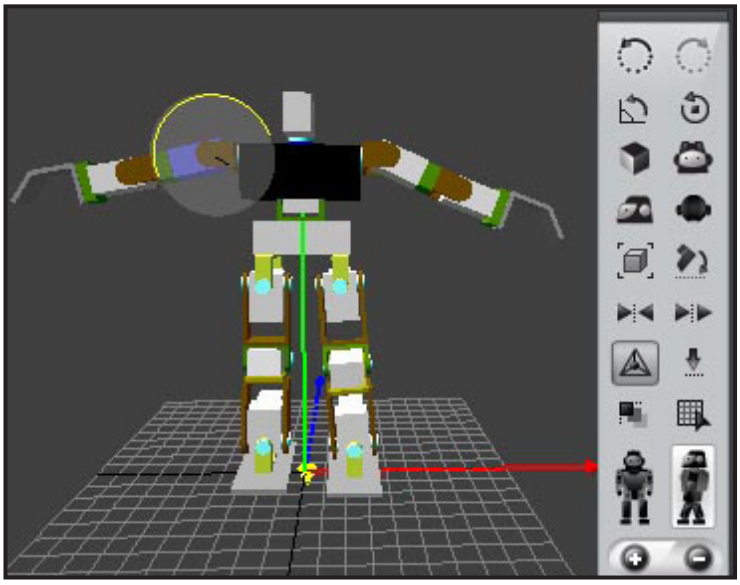


14 로봇 캡쳐를 위한 모션

이번에는 로봇자체를 캡처하여 편집하는 방법을 알아봅니다.
“토크 Off” 버튼을 눌러서 토크를 준 후 원하는 로봇의 모션을 손으로 직접 만듭니다.

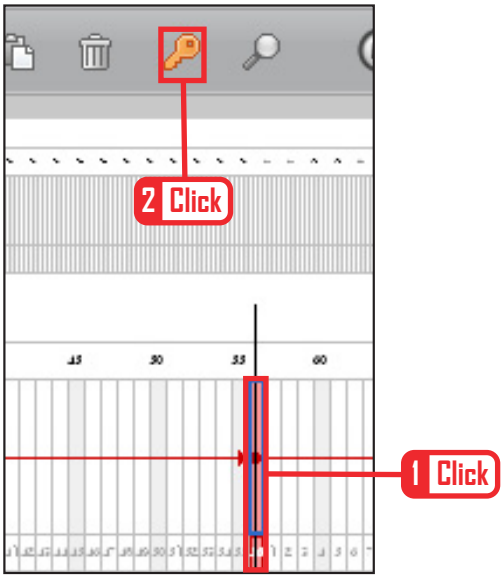
15-1 캡쳐

로봇 캡처 버튼을 클릭합니다.



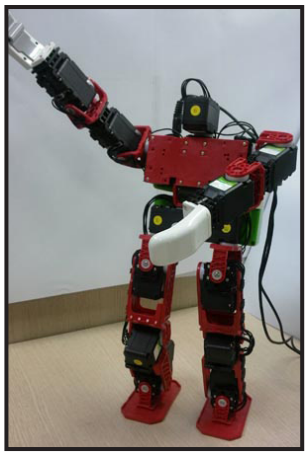
15-2 3D 모션창 캡처 보여지기

캡처버튼을 누르면 3D 모션창에 바로 캡처한 로봇 모양이 나옵니다.



15-3 키 프레임 삽입

캡처한 모션을 원하는 프레임에 삽입합니다.
프레임 클릭하고 키프레임 클릭하면 삽입 됩니다.

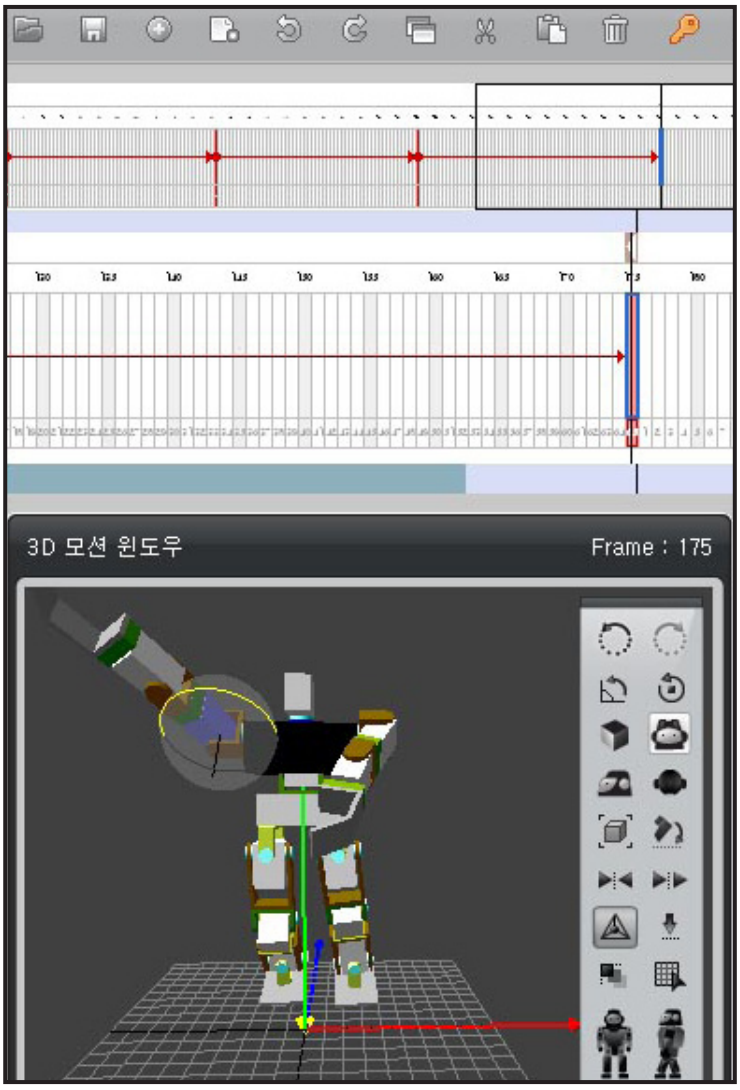


16-1 다른동작

로봇의 다른 동작을 손으로 직접 만듭니다.

16-2 캡쳐하기

캡처합니다.

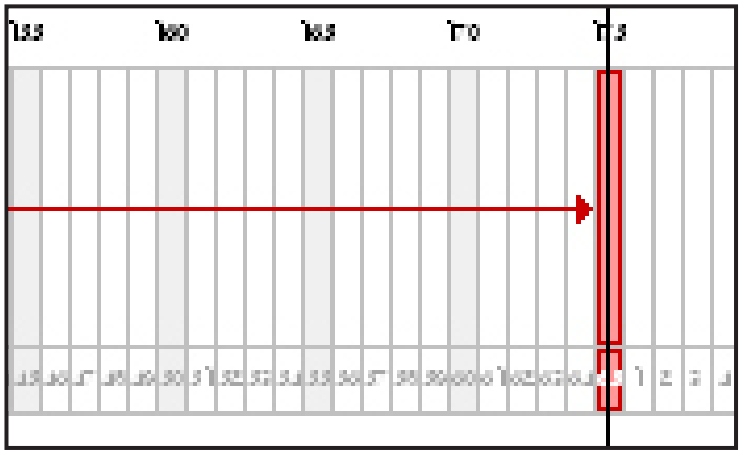


17 모션 확인

손으로 만든 동작과 3D모션창과 비교합니다.

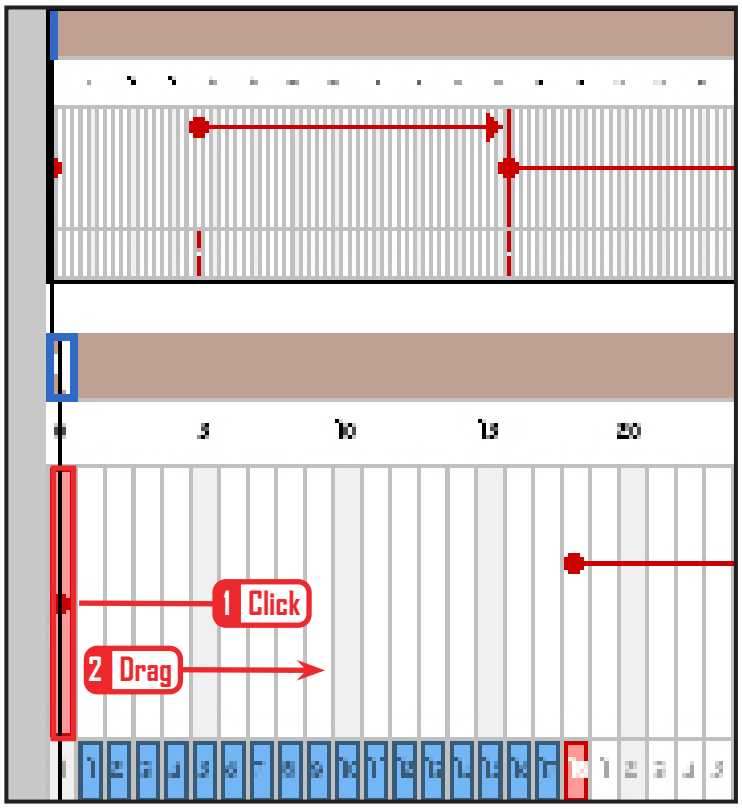
18 캡쳐하기

새로 만든 모션을 캡처합니다.



19 키 프레임 삽입

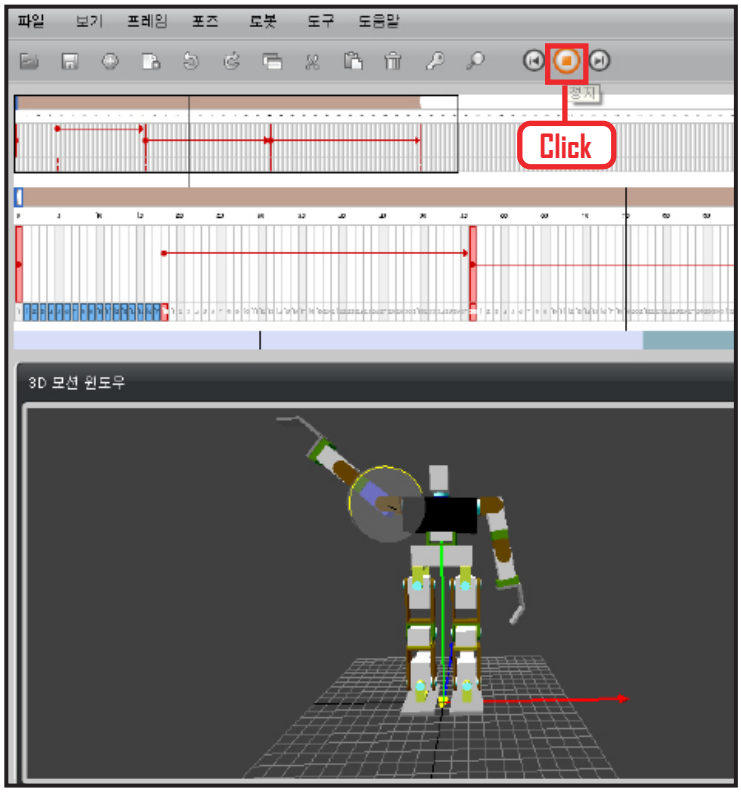
모션을 원하는 곳에 삽입합니다.



20 Delay 값

로봇의 첫 동작에서는 첫 동작 키 프레임과 두번째 동작 키 프레임간에 모션 차이가 많이 나면 로봇이 갑작스럽게 움직일 수 있습니다. 그것을 방지하기 위해어 첫 동작을 천천히 실행하게 하는 기능을 부여해 줍니다.

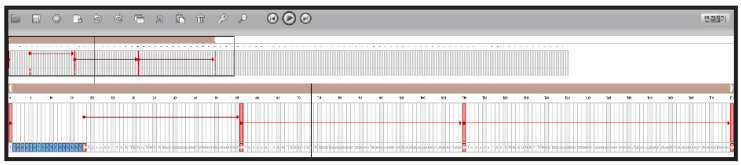
프레임하단을 클릭하여 왼쪽 마우스를 누른 상태로 오른쪽으로 드래그하면 왼쪽 그림과 같이 표기되고, Delay 값이 만들어집니다.



21 화면 플레이

현재까지 만들어진 모션을 화면에서 먼저 동작시켜봅니다.

플레이 아이콘을 클릭합니다.

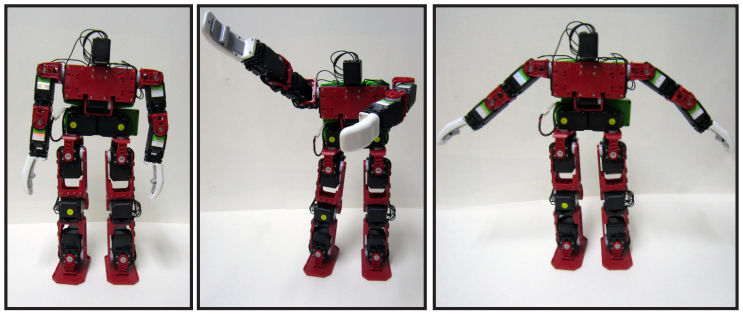


22 플레이

타임프레임에서 현재 플레이되고 있는 위치에 따라서 이동선이 표기됩니다.

23 로봇에서 플레이

로봇에서 직접 플레이합니다. 우측 로봇연결 아이콘에서 Play 버튼을 클릭합니다.



24 로봇에서 동작

왼쪽은 3D 모션창에서 만든 모션이고, 가운데는 로봇에서 직접 동작을 만들어 캡처한 것이고, 오른쪽도 로봇에서 만든 것을 캡처한 것입니다.
플레이 시키면 연속동작으로 움직입니다.

PART 02

소프트웨어 : DR-Visual Logic

설치하기

DR-Visual Logic 소개

DR-Visual Logic 은 동부로봇에서 개발된 고유의 로봇 프로그래밍 언어를 Graphic 모듈화시켜 Drag & Drop 방식을 채용한 로봇 프로그래밍 툴입니다. 동부로봇의 제어기인 DRC 의 기능에 맞춰서 각각의 기능을 모듈화 시켜놓았습니다.

마우스만을 활용하여 드래그 앤 드랍 방식으로 초보자도 쉽게 프로그래밍 할 수 있는 구조로 되어있으며, C-like 탭을 제공하여, 그래픽 프로그래밍이 Text 로 변환되는 코드를 바로 확인 할 수 있습니다. C 문법 과 유사한 코드이기 때문에 프로그래밍 입문자가 C 문법을 익히는데 많은 도움이 될 수 있습니다. 로봇 언어중에 가장 쉽고, 가장 강력한 기능을 발휘하는 프로그래밍 툴로서 초급자부터 고급자까지 그 활용폭이 넓어서, 로봇 교육 시장에서 가장 주목받는 로봇 언어로 사용되어지고 있습니다.

앞으로 업그레이드된 제어기 관련 모듈 추가 및 로봇 모션의 모듈화 및 시뮬레이션 결합 등으로 다채로운 기능을 발휘하는 프로그램으로 업그레이드 할 예정입니다.

- 시스템 요건
- 최소 인텔 팬티엄 800 Mhz
- Windows XP, Windows 7
- 최소 256 MB RAM
- 하드디스크 설치 공간 300 MB 필요
- USB Port
- Macintosh (지원 예정)
- Windows Vista

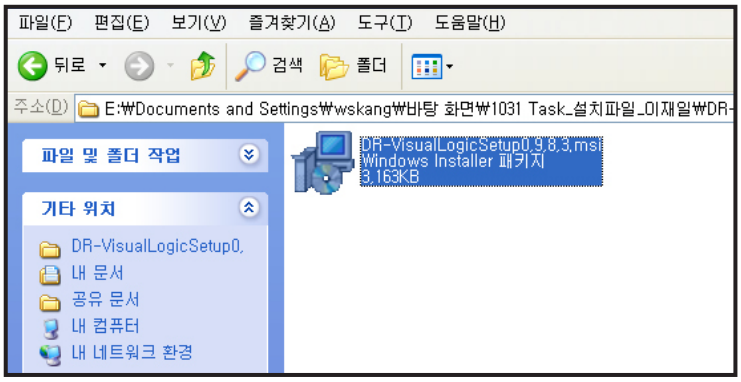
PART 02

소프트웨어 : DR-Visual Logic

설치하기

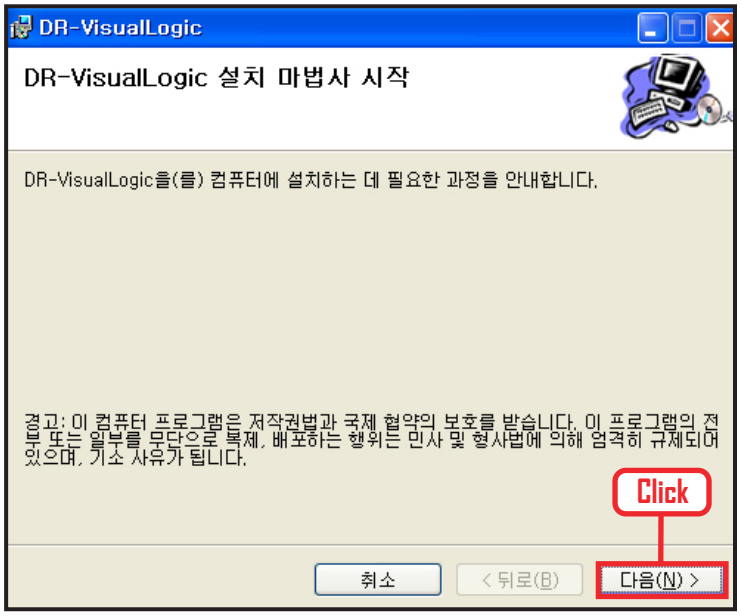
따라하기

설치부터 실행까지 따라해보세요



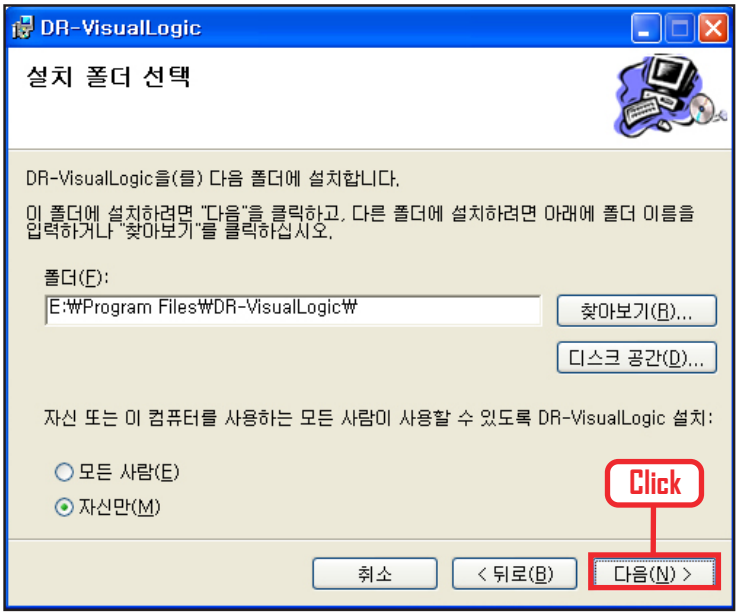
01 설치파일

설치파일을 클릭합니다.



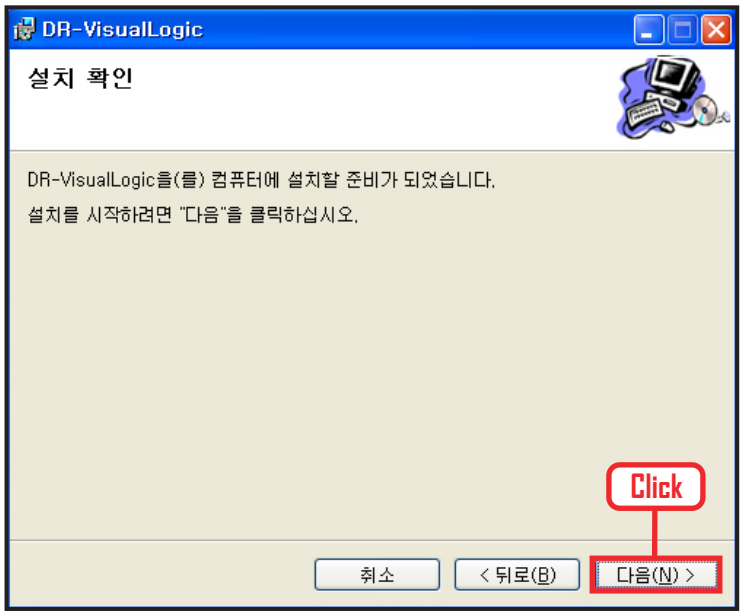
02 설치 마법사 시작

"다음" 버튼을 클릭합니다.



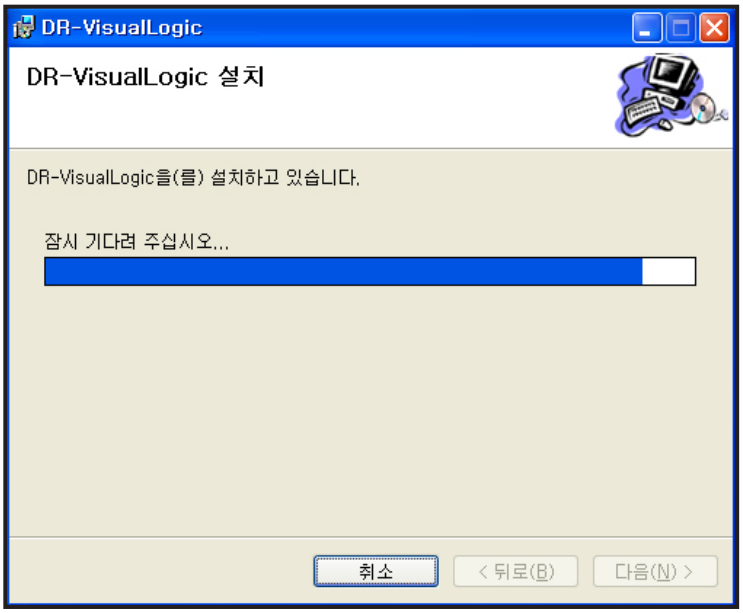
03 설치 폴더를 선택

"다음" 버튼을 클릭합니다.



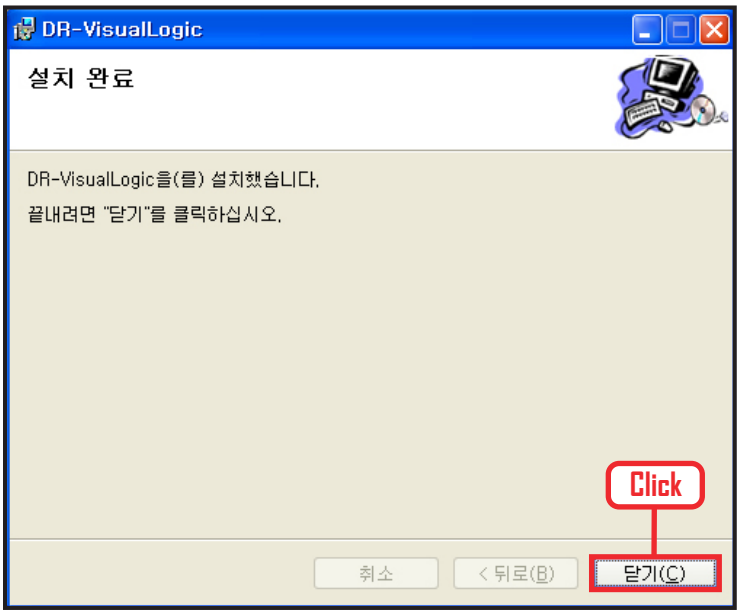
04 설치 확인

"다음" 버튼을 클릭합니다.



05 설치 시작

설치 시작 합니다. 프로그레스 바가 끝날때 까지 기다려주세요



06 설치 완료 선택

"닫기" 를 클릭하세요
소프트웨어 설치가 완료되었습니다.

07 실행파일 확인

바탕화면과 **Windows 시작 〉 모든프로그램 〉 Dongbu Robot 〉 DR-VisualLogic** 에서 실행파일을 확인하세요
실행파일을 클릭하면 프로그램이 실행됩니다.

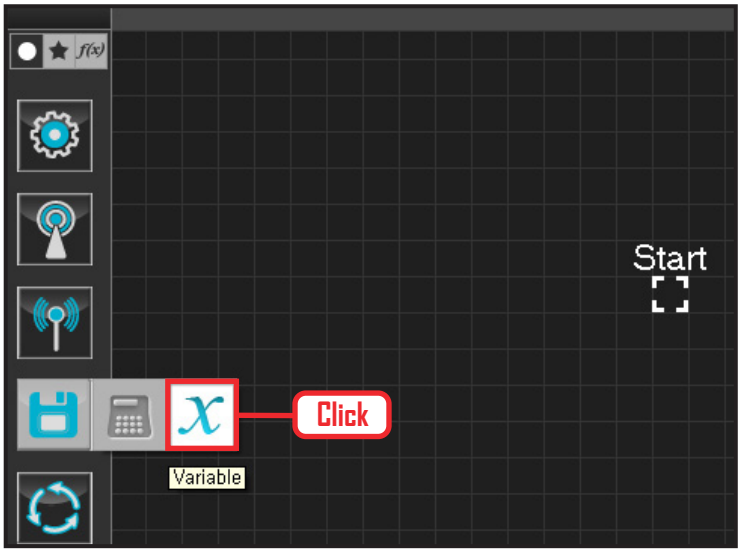
소프트웨어 : DR-Visual Logic

Hello DR-Visual Logic

첫 번째 프로그램 따라하기

예제설명

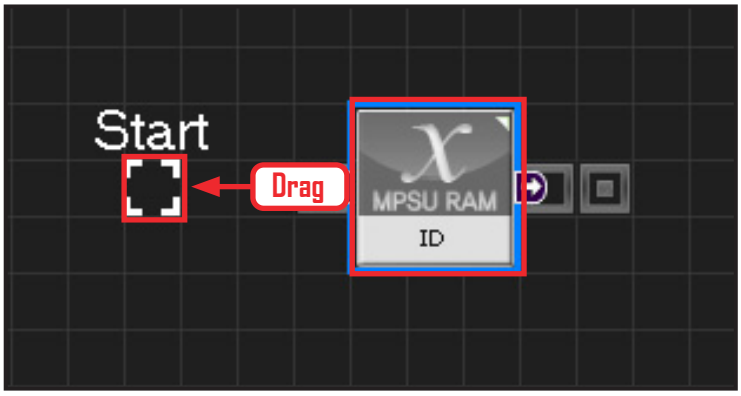
양팔 벌려있는 로봇의 모터를 한쪽팔만 차려자세로 내려봅니다. 16축 휴머노이드 로봇의 모터들을 모두 중앙정렬 시키면, 로봇은 양팔이 좌우로 팔벌려 자세가 됩니다. 그중 한쪽 팔을 차려자세로 내려봅니다.



01 변수 지정

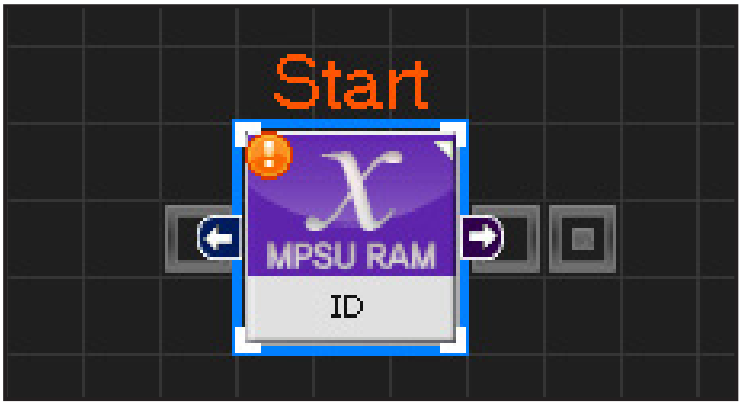
로봇을 동작시킨다는 것은 로봇의 서보 모터를 동작시킨다는 의미입니다. 서보가 스스로 움직일 수 있는 상태로 값을 지정해주어야 합니다.

Data 〉 Variable 모듈을 클릭합니다.



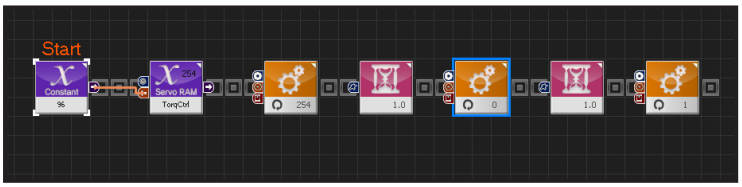
02 시작

모듈의 왼쪽 연결선을 Start Point 에 드래그하여 정확히 도킹을 시킵니다.



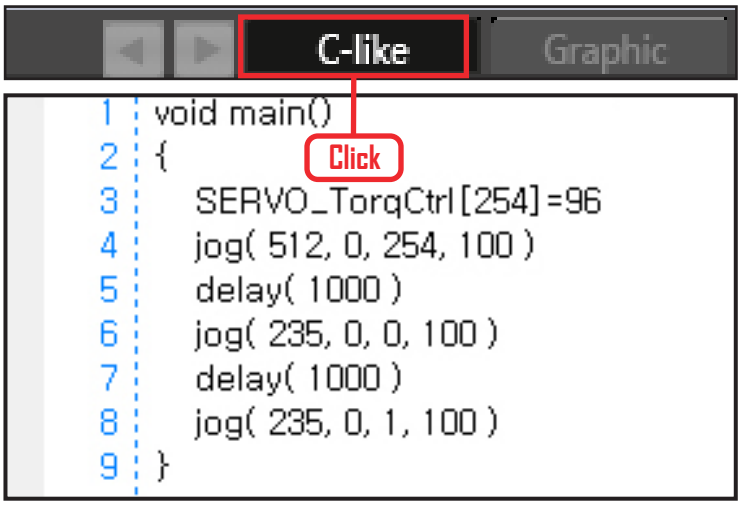
03 프로그래밍 시작

모듈과 Start Point 가 정확히 도킹하면 왼쪽과 같이 활성화된 칼라 이미지 모듈로 변합니다.
그럼 프로그래밍이 시작되었다는 의미입니다.



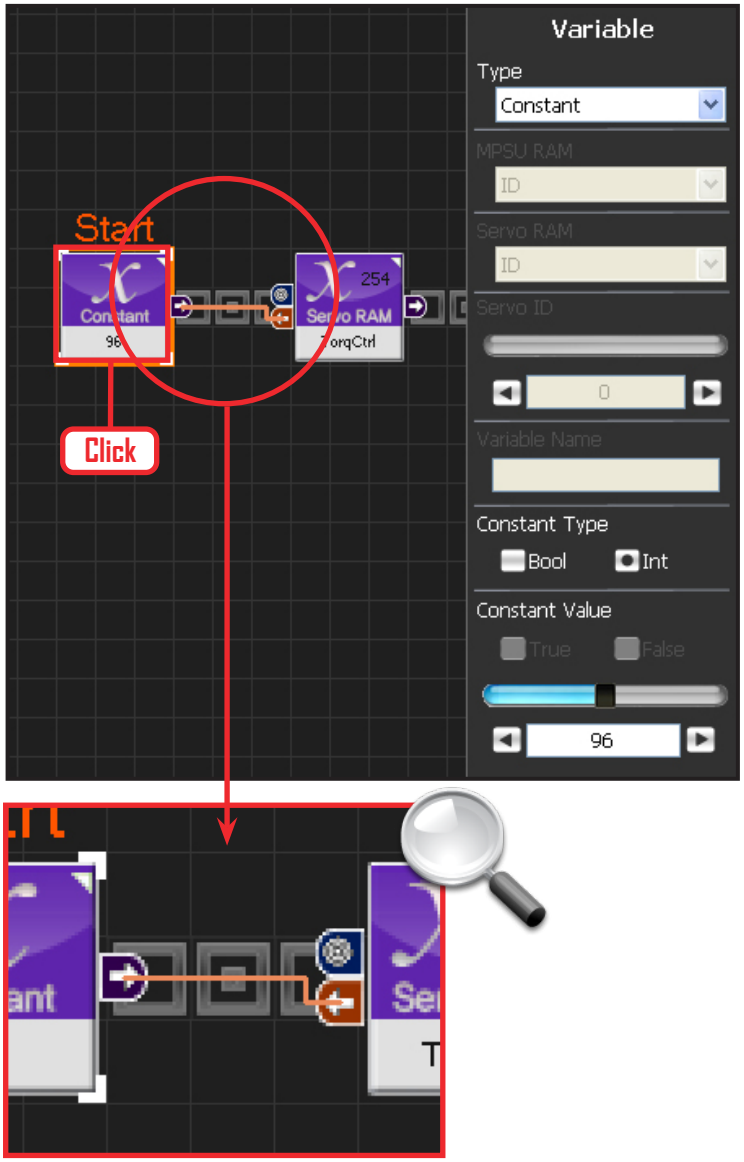
04 전체 프로그래밍

로봇의 모터를 움직여 한쪽팔을 내리는 전체 프로그래밍 전개 화면입니다.



05 C-Like 보기

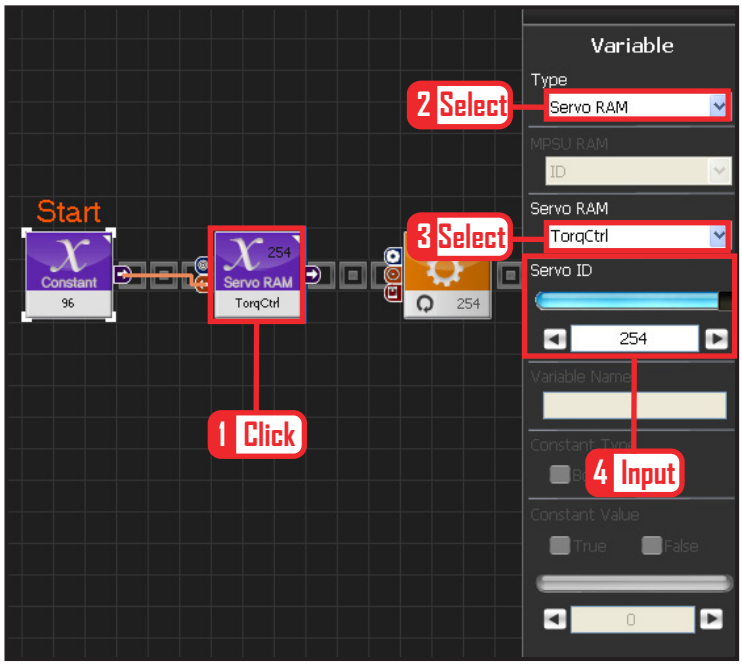
오른쪽 상단의 Graphic 탭에서 C-like 탭을 클릭하면 왼쪽과 같은 Task 프로그래밍 화면이 나옵니다.
전체 프로그래밍 Task 화면입니다.
C와 유사한 문법 구조를 가지고 있으므로 C 문법 선행학습 효과도 있습니다.
각 모듈별로 클릭하면 커서가 따라서 움직이므로 모듈이 Text로 어떻게 변환하는지 확인할 수 있습니다.



06 상수 설정

서보 모터를 스스로 움직일 수 있는 상태로 만드는 과정입니다.

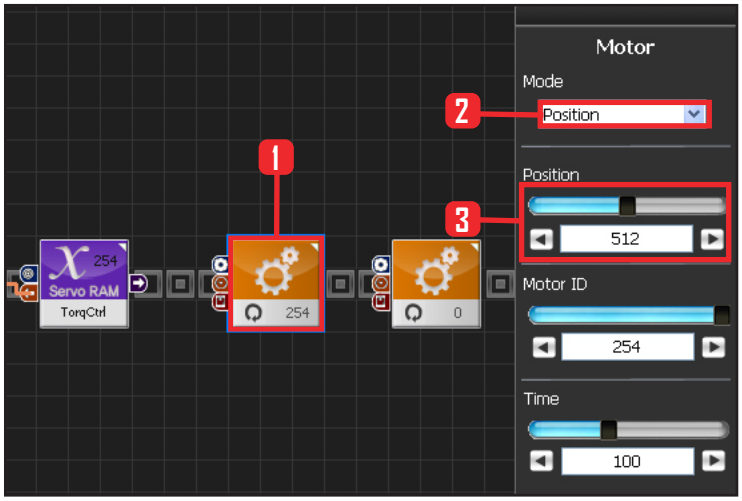
Variable Type 을 Constant 로 선택합니다.
속성중에 Constant Value 값을 96 으로 설정합니다.
서보의 TorgControl 레지스터에 96(0X60) 이라는 값이 들어가면 서보가 움직일 수 있는 상태가 됩니다.
그 값은 Output 커넥터를 통하여 뒤 모듈의 토크값에 전달합니다.



07 모든 서보에 적용

앞에서 받은 96 상수값을 모든 서보에 적용하는 과정입니다.

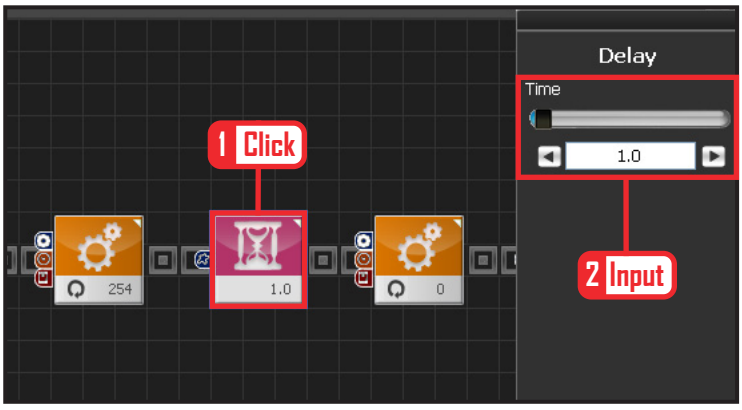
Variable 〉 Type : Servo RAM을 선택합니다.
Servo RAM : TorqCtrl 을 선택합니다.
Servo ID : 254 로 설정합니다. 254는 연결 되어있는 모든 서보에 적용하겠다는 의미입니다.



08 모든 서보모터 각도 설정

모든 서보모터의 각도를 중앙에 보내는 과정입니다.

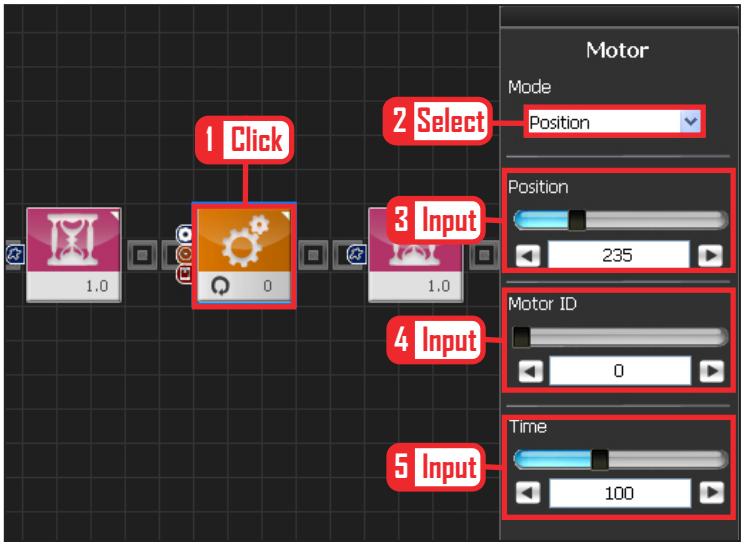
Motion 〉 Moter 를 선택합니다.
Mode : Positon 으로 선택합니다. 각도를 조절합니다. Position : 512 로 설정합니다. 512 번은 모터를 중앙으로 보낸다는 의미입니다
Motor ID : 254 로 설정합니다. 254 는 모든 모터에 적용하겠다는 의미입니다.
Time : 100 으로 설정합니다. 단위는 1당 11.2ms로, 100은 약 1.12초를 의미합니다.
1.12초동안 원하는 각도에 위치시킨다는 의미입니다.



09 Delay

다음 모터 동작 전에 1초를 기다린 후 시작하는 설정입니다.

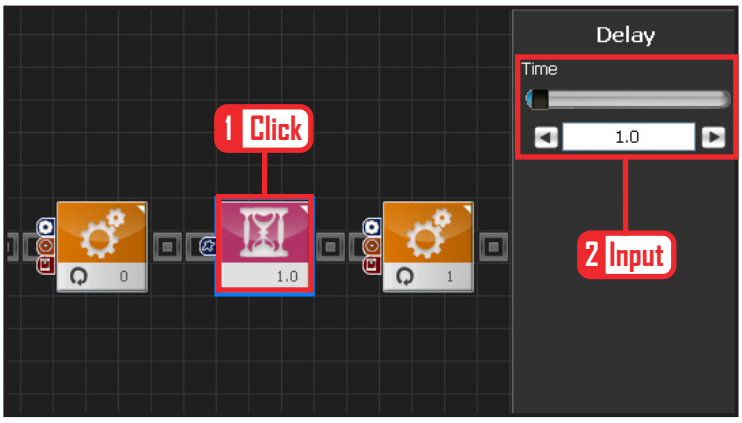
Flow 〉 Delay 모듈을 선택합니다.
Time : 1.0 을 설정합니다. 1초정도 Delay 하겠다는 의미입니다.



10 모터 0번 (오른쪽 어깨) 설정

차려자세(기본자세) 만들기
모든 로봇의 모터의 각도를 중앙으로 정렬 하면 휴머노이드에서는 팔을 좌우로 뻗게됩니다. 한쪽팔을 차려 자세로 되돌려 놓는 설정입니다.

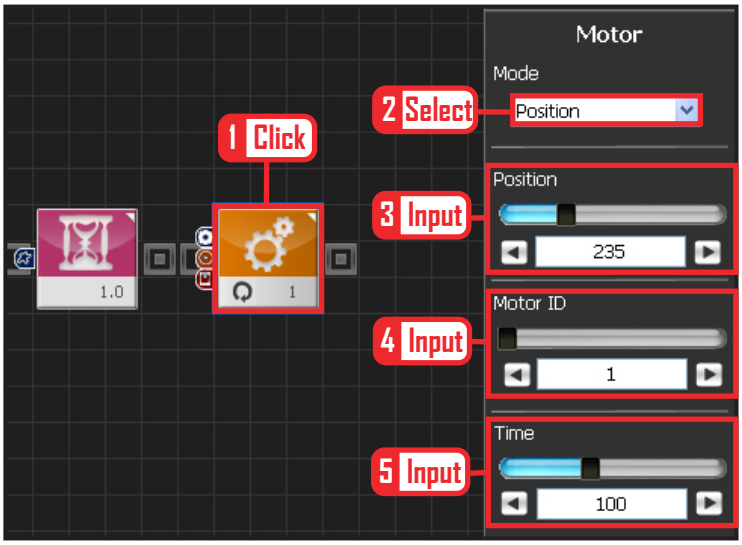
Motion 〉 Motor 를 선택합니다.
Mode : Position 으로 선택합니다.
Position : 235 로 설정합니다. 235 는 수평으로 들고 있던 오른손을 수직으로 내려 갈 수 있게 모터를 돌리게 됩니다.
Motor ID : 0 으로 설정합니다. 오른쪽 어깨 모터 ID 가 0번입니다.
Time : 100 으로 설정합니다. 약 1.12초동안 원하는 각도로 이동합니다.



11 Delay

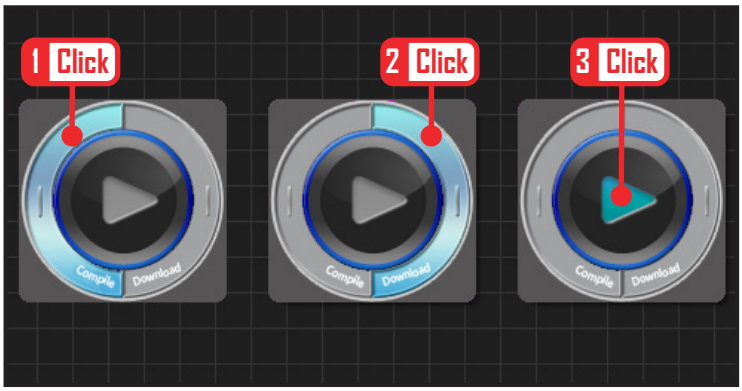
다음 모터 동작 전에 1초를 기다린 후 시작하는 설정입니다.

Flow 〉 Delay 모듈을 선택합니다.
Time : 1.0 을 설정합니다. 1초정도 Delay 하겠다는 의미입니다.



12 모터 1번 (오른쪽 팔) 설정

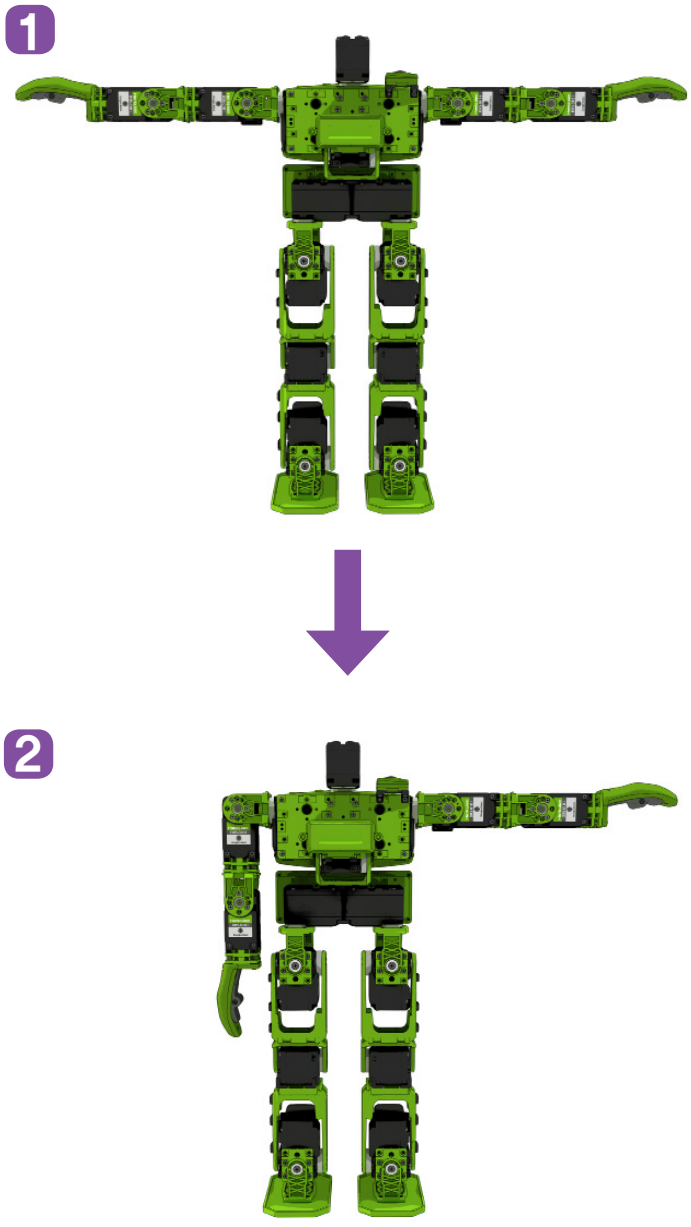
Mode : Postion 으로 선택합니다.
Position : 235 로 선택합니다. 235는 수평으로 되어 있던 팔을 수직으로 내리게 됩니다.
Motor ID : 1 로 설정합니다. 오른쪽 위쪽 팔 어깨와 닿는 모터가 1번 모터입니다.
Time : 100 으로 설정합니다. 약 1.12초동안 원하는 각도로 이동합니다.



13 다운로드

프로그래밍 후 컴파일 –〉 로봇에 다운로드 –〉 실행하는 과정을 거칩니다.

Compile을 클릭합니다. 에러가 없으면, 우측 Download를 클릭합니다. 로봇에 다운로드 합니다. 다운로드 완료된 후, 실행버튼(중간 화살표)을 클릭합니다.



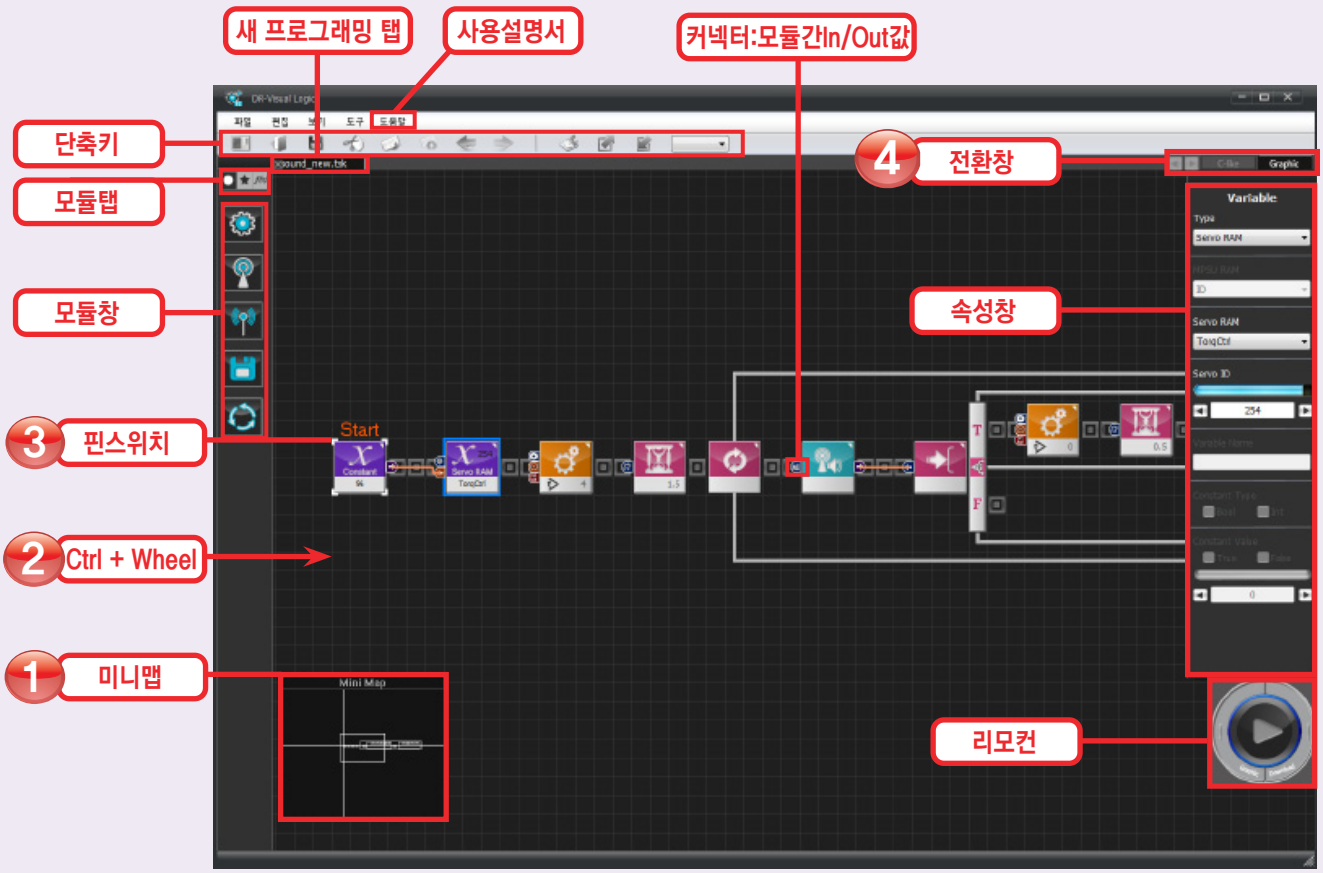
14 로봇동작

양팔 벌려 수평으로 유지하고 있던 로봇 팔중에 오른쪽 팔이 아래로 내려갑니다.

PART 02

소프트웨어 : DR-Visual Logic

사용자 인터페이스



❶ 미니맵 : 드래그방식으로 조정하면, 프로그래밍이 길어져도 현재의 위치를 알 수 있고, 원하는 위치에 단번에 갈 수 있습니다.

❷ Wheel Up/Down : 화면 확대 축소

❸ 커넥터 스위치 : 클릭하면 해당 모듈 핀의 타이틀이 나와 한눈에 알아 볼 수 있게 처리하고 다시 누르면 없어집니다.

❹ 전환창 : Graphic to Text 로 그래픽으로 프로그래밍 된 소스가 Text 형태의 소스로 변환됨, C문법과 유사합니다.

0 단축키 : 메뉴중에 많이 쓰이는 메뉴를 단축키로 처리

1 모듈탭 : 모든 모듈

2 모듈창 :

3 새 프로그래밍 탭 :

4 사용설명서 :

5 커넥터 :

6 속성창 :

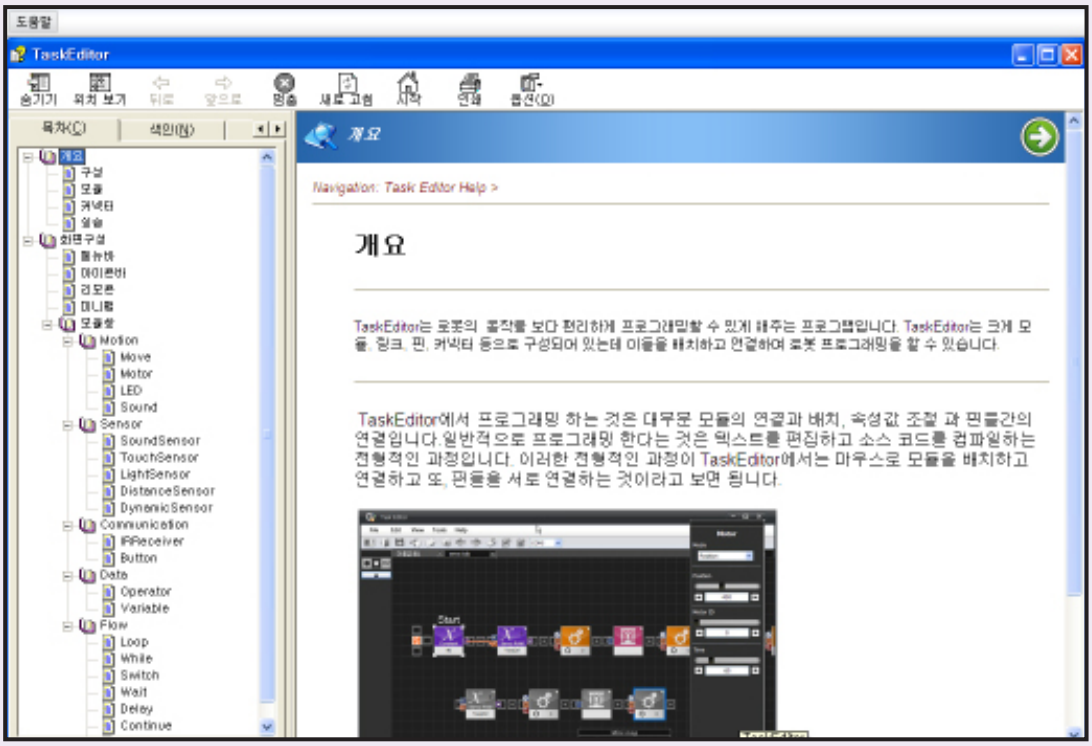
7 다운로더 :

PART 02

소프트웨어 : DR-Visual Logic

도움말 보기

메뉴에서 도구 〉 도움말을 클릭하면 아래와 같은 도움말 팝업창이 뜹니다. DR-Visual Logic 를 사용하기 위해서는 도움말을 먼저 읽어보신 후 사용하기를 권합니다. (**상단 도움말 클릭 〉 내용색인 클릭 〉 좌측메뉴 타임라인 클릭** → 아래 그림이 뜸)



■ 개요

구성
모듈
커넥터
실습

■ 화면구성

메뉴바
아이콘바
리모콘
미니맵
모듈창

■ 모듈창

Motion : Move(로봇저장모션), Motor(서보모터), LED, Sound(비프음)
Sensor : Sound Sensor, Touch Sensor, Light Sensor,
Distance Sensor(거리센서, PSD Digital, PSD Analog),
Dynamics Sensor(Accermeter, Gyro 센서)
Communication : IRReceive, ZigBee, Button
Data : Operator, Variable
Flow : Loop, While, Switch, Wait, Delay, Continue, Break

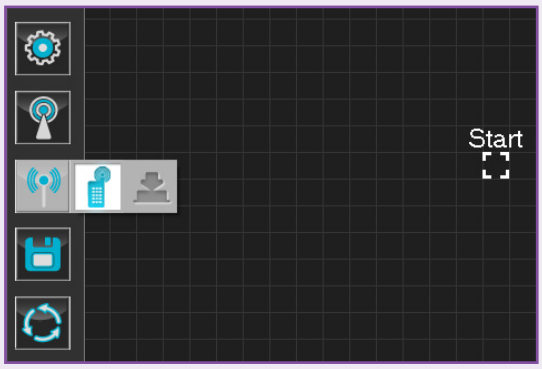
PART 02

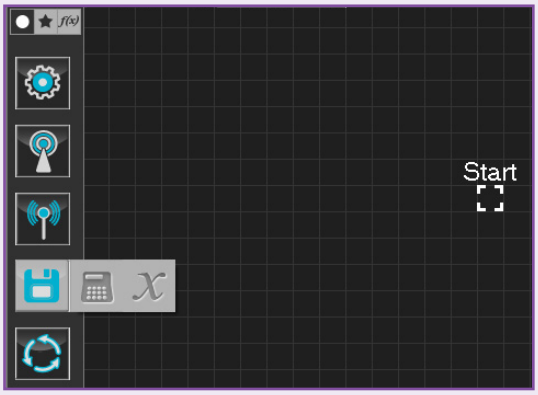
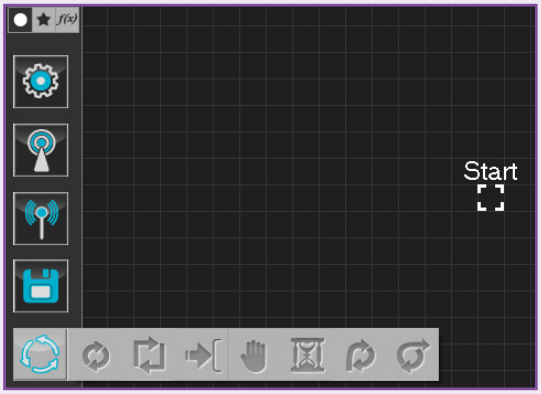
소프트웨어 : DR-Visual Logic

프로그래밍 모듈

DR-Visual Logic 는 아래와 같은 모듈로 구성됩니다.

Module Pack 에는 프로그램을 만들기 위해 필요한 모든 프로그래밍 모듈이 포함되어있습니다. 각각의 모듈은 제어기 DRC 가 지원해주는 기능으로 구성되어있습니다. Description 은 휴머노이드 조립시에 각각의 부품이 어느 위치에 있고, 그 부품에 해당한 모듈에 대해서 간략히 설명하였습니다.

Module Pack	Picture	Module	Description
Motion (모션)		Move	저장된 로봇 모션을 실행함
		Motor	모터별로 위치 / 속도제어
		LED	머리 – 저장된 LED를 실행함 등 – 제어기 LED를 제어함
		Sound	Buzzer 를 울림
Sensor (센서)		Sound Sensor	내장, 좌우구분
		Touch	머리모듈의 터치 인식 여부
		Light	내장, 등부위, 광량 측정
		Distance	거리측정
		Dynamics	역학, 가속도와 각속도 측정
Communication (통신)		IRRciever	리모컨 데이터 인식
		Button	후면 제어기 버튼 인식

Module Pack	Picture	Module	Description
Data (데이터)		Operator	연산자
		Variable	레지스터 데이터 사용자 선언변수/상수
Flow (흐름분기)		Loop	무한반복/for문
		While	특정 조건동안 반복
		Switch	제어 분기, if-else
		Wait	특정조건동안 대기
		Delay	지정시간동안 대기
		Continue	loop의 처음으로 돌아감
		Break	loop를 빠져나옴

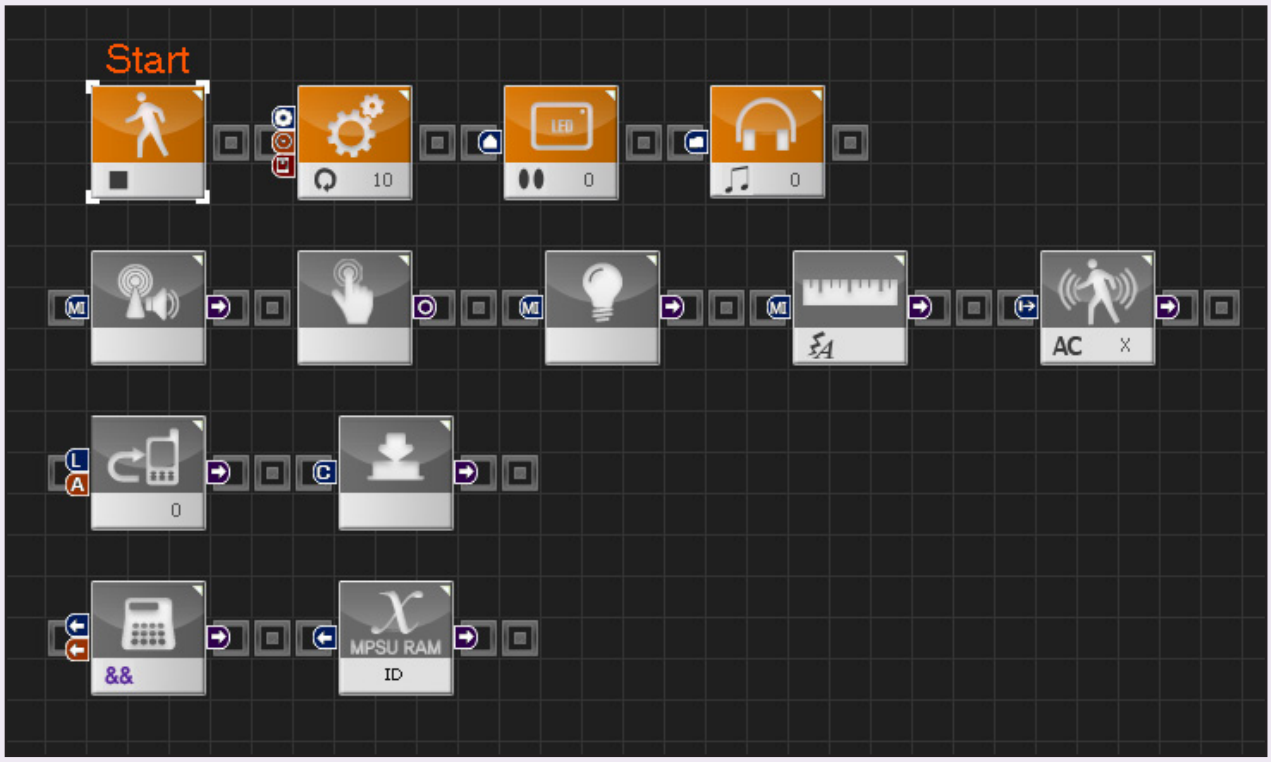
소프트웨어 : DR-Visual Logic

프로그래밍 모듈 〉 일반형 모듈

DR-Visual Logic의 모듈은 일반형 모듈과 Flow 형 모듈로 나뉩니다.

일반형 모듈

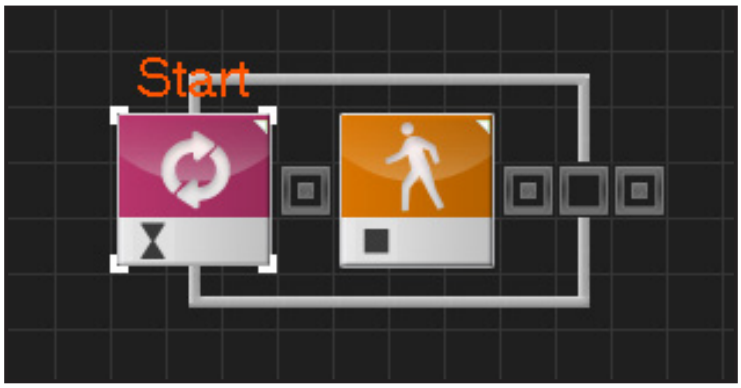
일반형 모듈은 순차적으로 모듈끼리 연결하여 사용됩니다. Flow 모듈을 제외한 모든 모듈이 여기에 해당합니다.



위부터 차례로 Motion, Sensor, Communicaiton, Data 를 나타내는 모듈 아이콘입니다.

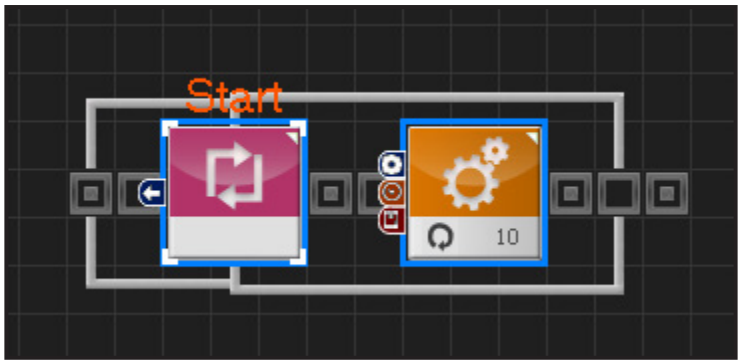
Flow 형 모듈

Flow 형 모듈은 일반 모듈과 연결하여 반복, 분기 등을 프로그래밍의 흐름을 연결시켜주는 모듈을 말합니다. 따라서 일반형 모듈과는 다르게 일반모듈과 결합시 외곽 아웃라인 형태의 띠가 형성되는 그래픽 구조를 갖습니다.



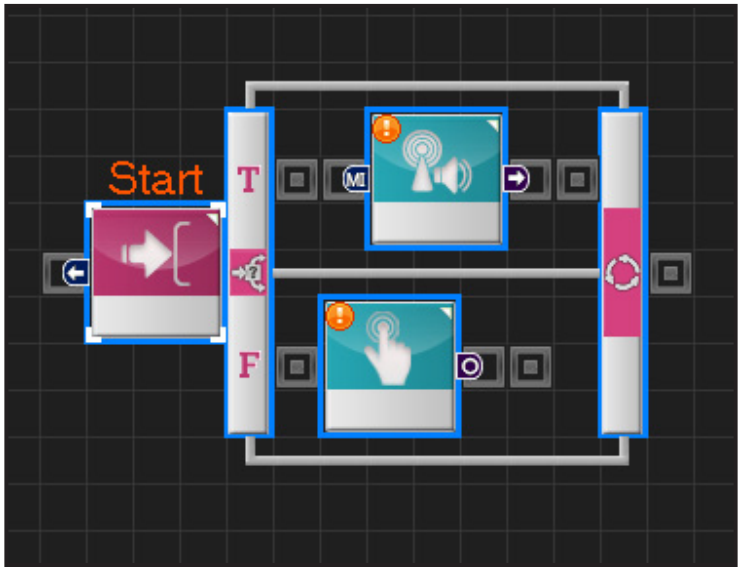
Loop

Loop 는 반복을 지시하는 모듈로서, 일정 횟수 동안 반복하는 For 문과 영원히 반복하는 Forever 문이 있습니다.



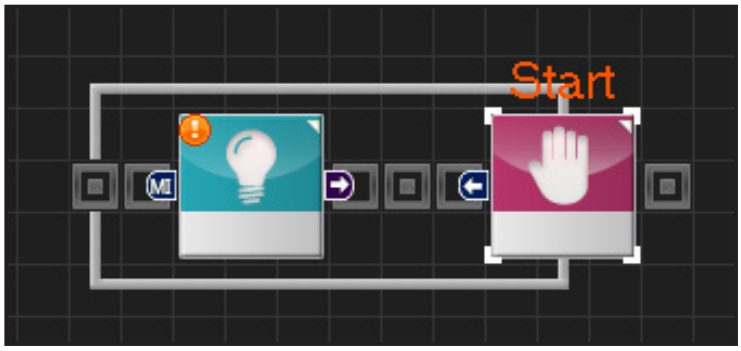
While

While 은 앞조건이 맞을시 뒤를 실행하라 는 의미로서 조건과 결합된 반복문입니다.



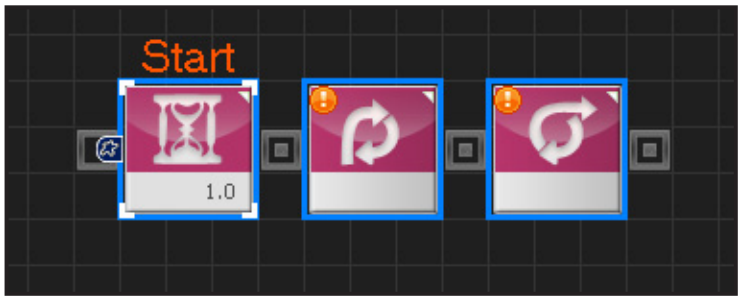
Switch

입력조건이 참일경우 위의것을, 거짓일 경우 아래의 것을 실행하는 if – else 문입니다.



Wait

입력 조건이 참일동안 프로그램의 실행을 멈추고, 거짓이 되면 다시 실행을 재개합니다.

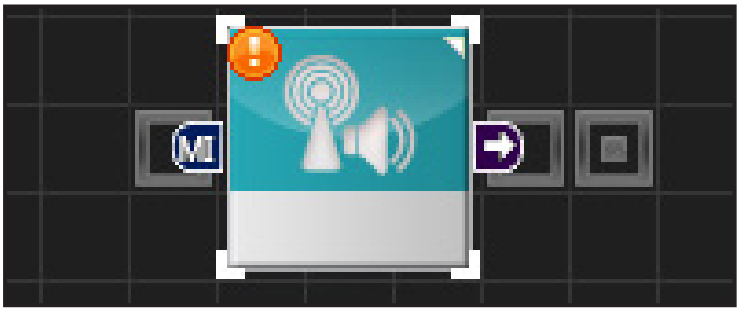


Delay, Continue, Break

좌 3개 모듈은 주변에 띠가 형성되는 않는 일반형 모듈처럼 나열됩니다.
Delay 는 일정 시간동안 지연시키는 모듈입니다.
Continue 는 반복문의 처음으로 돌아가 계속 실행하라는 의미입니다.
Break 는 반복문을 빠져나오라는 의미입니다.

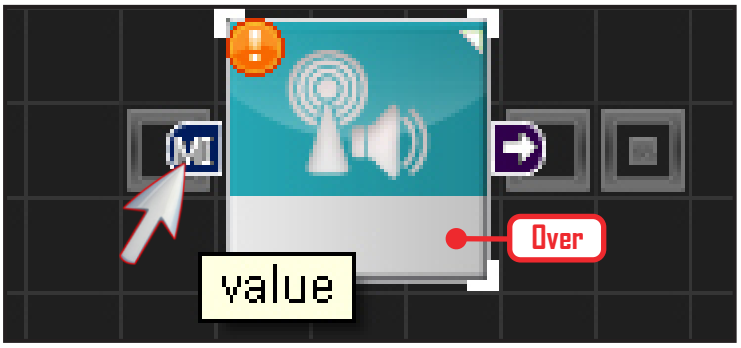
커넥터

모듈에 따라 Input 값과 Output 값을 같는 모듈이 있습니다. Output 커넥터의 결과값은 이것과 연결된 뒤 모듈의 Input 값에 입력되어 동작합니다. Input/Output값이 있는 모듈은 모듈상에 처음부터 좌우측에 커넥터 형태로 나와 있습니다. 아래는 커넥터의 예를 보여줍니다.



커넥터

팔 올린 모션을 프레임에 삽입합니다. 원하는 프레임을 클릭한 후 키 아이콘을 클릭하여 모션을 삽입합니다.

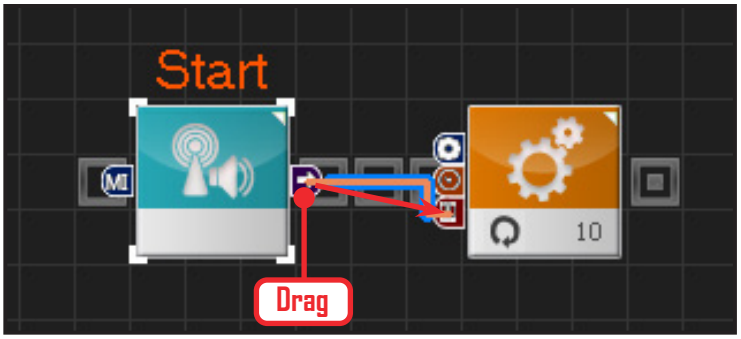


말풍선 도움말

커넥터의 아이콘만 보고는 어떤 기능의 커넥터인지 구분하기 어렵습니다. 커넥터 위에 마우스를 올리면 좌측과 같이 말풍선으로 핀 이름이 나옵니다.

말풍선 펼치기

여려개의 커넥터 이름을 한꺼번에 보고 싶으면 우측 상단의 세모모양의 아이콘을 클릭하면 커넥터 양쪽에 커넥터 이름 말풍선이 나옵니다. 다시 클릭하면 없어집니다.

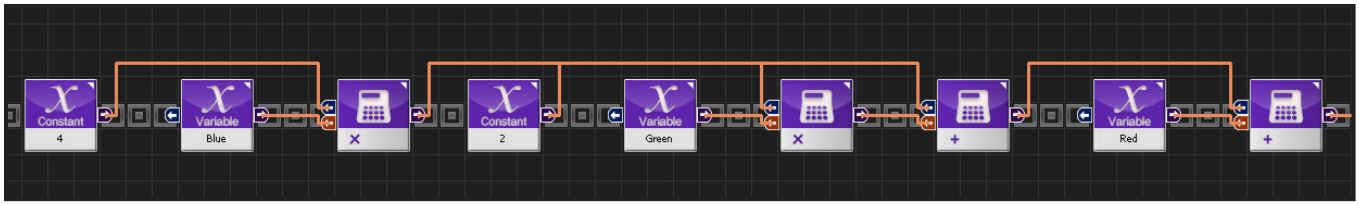


커넥터 연결

앞 모듈의 Output 값을 뒷 모듈의 Input값에 넣기 위해서는 마우스로 드래그해서 연결하면 좌측과 같이 연결선이 표기됩니다.

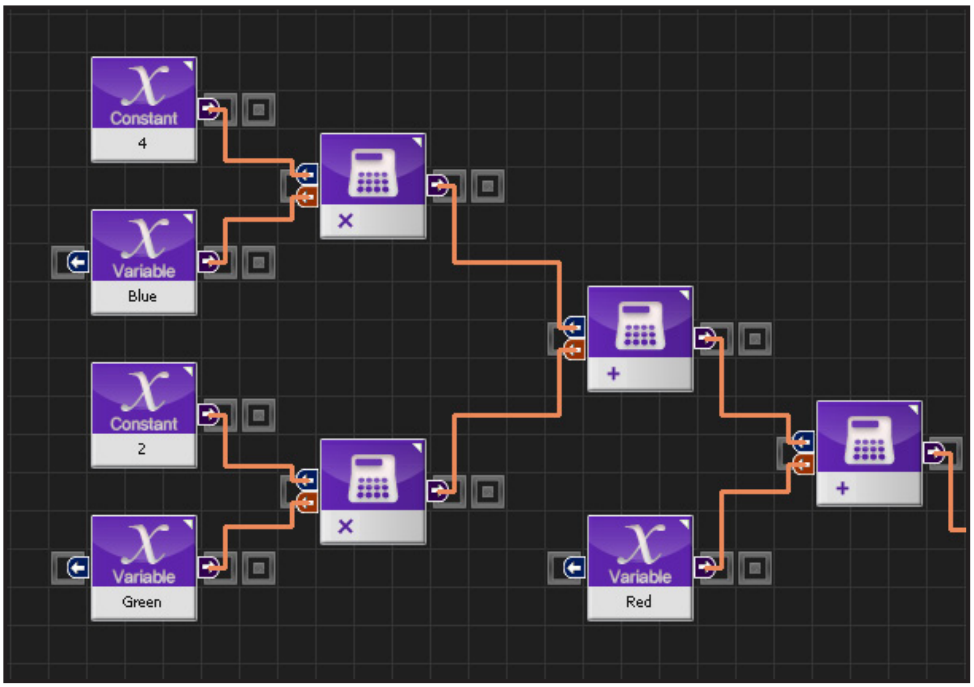
연결형 타입

모듈을 연결할 때 직렬형 연결과 병렬형 연결이 있습니다.



직렬형 연결

직렬형 연결은 말 그대로 좌에서 우로 모듈을 순차적으로 연결하는 것입니다. 위에는 연산에 관한 프로그래밍입니다. ((4xBlue)+(2xGreen))+(1xRed) 연산식을 직렬형 연결로 표현한 것입니다.



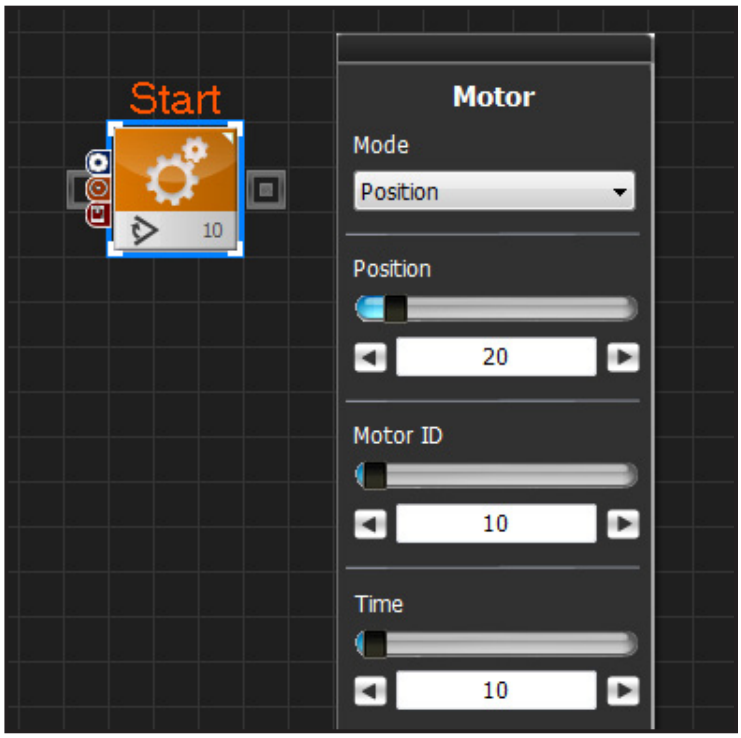
병렬형 연결

병렬형 연결은 상하 공간을 활용하여 모듈을 병렬적으로 연결하는 것입니다. 아래 예시는 위 직렬형 연결과 같은 프로그래밍입니다.

속성창

속성창

모듈마다 자신의 속성을 가지고 있고, 그 속성값을 설정해줘야만 프로그래밍을 수행할 수 있습니다. 리스트 팝업, 라디오 버튼, 숫자 설정 등의 UI 로 표현되며, 각 모듈별 속성 설명 및 속성값/제한값 등은 도움말을 참조하기 바랍니다.



속성창

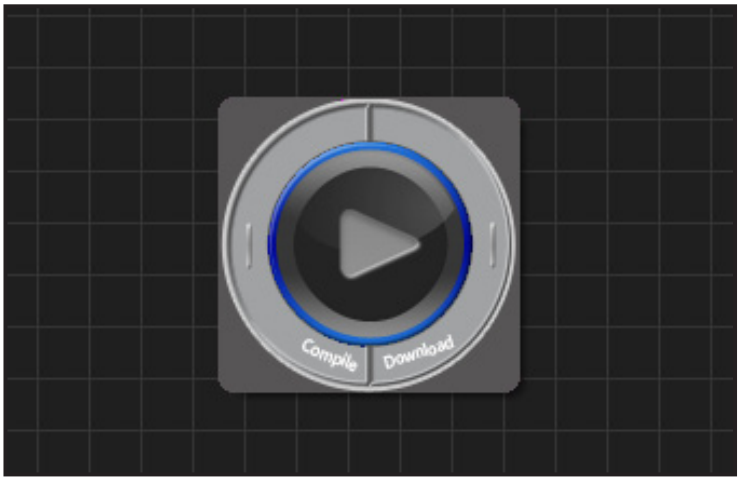
모터모듈을 클릭했을 때 우측과 같은 속성창이 나타납니다.
모터는 위치제어와 속도제어 속성을 가집니다. Mode 에서 위치제어를 위해서는 Postion 을 선택하고, 속도 제어를 위해서는 Velocity 를 선택합니다.
세부 설정에 Position, Motor ID, Time 등 의 값을 조절합니다.

소프트웨어 : DR-Visual Logic

컴파일/다운로드하기

다운로더

로봇의 프로그래밍을 완료하고, 컴파일, 다운로드, 로봇에서 실행을 거치게 됩니다. 프로그래밍 창 좌측 하단에 큰 아이콘으로 제공하며, 도구 메뉴에서 상세하게 활용할 수 있습니다.



다운로더 아이콘

왼쪽 아이콘은 다운로더 아이콘입니다. 좌측은 Compile, 우측은 Download를 나타내며, 가운데 화살표는 로봇에서의 실행을 의미합니다.

도구 메뉴

도구 메뉴에서는 좀 더 상세하게 실행이 가능합니다.

컴파일 : 작성한 Task를 컴파일 합니다.
다운로드 : 컴파일한 Task를 다운로드 합니다.
실행 : 다운로드 한 Task를 실행합니다.
중지 : 로봇 실행중 중지시킵니다.
중단점까지 실행 : 프로그래밍 중간마다 중단점을 지정하면 그곳까지 실행후 일시 중단합니다.
한단계씩 실행 : 모듈별로 실행합니다.

PART 02

소프트웨어 : DR-Visual Logic

모듈별 프로그래밍

기본 제공되는 샘플 프로그램은 제어기 DRC 플랫폼 기반 프로그램이며, 기본 16축 휴머노이드를 기반으로 작성되었습니다. 18,20축 휴머노이드나 변신 조립형 로봇에 따른 모션이나 모듈 변경이 있을 때에는 그것에 맞춰서 재 프로그래밍해야합니다.
프로그램 적용시 먼저 로봇 모터 ID를 확인하고, 로봇의 센서 위치를 확인합니다. 또한 DR-SIM을 통해서 로봇 저장모션 리스트를 확인하여 프로그래밍시 맞는 Index 값을 줍니다. 기본 제공하는 샘플예제는 아래와 같습니다

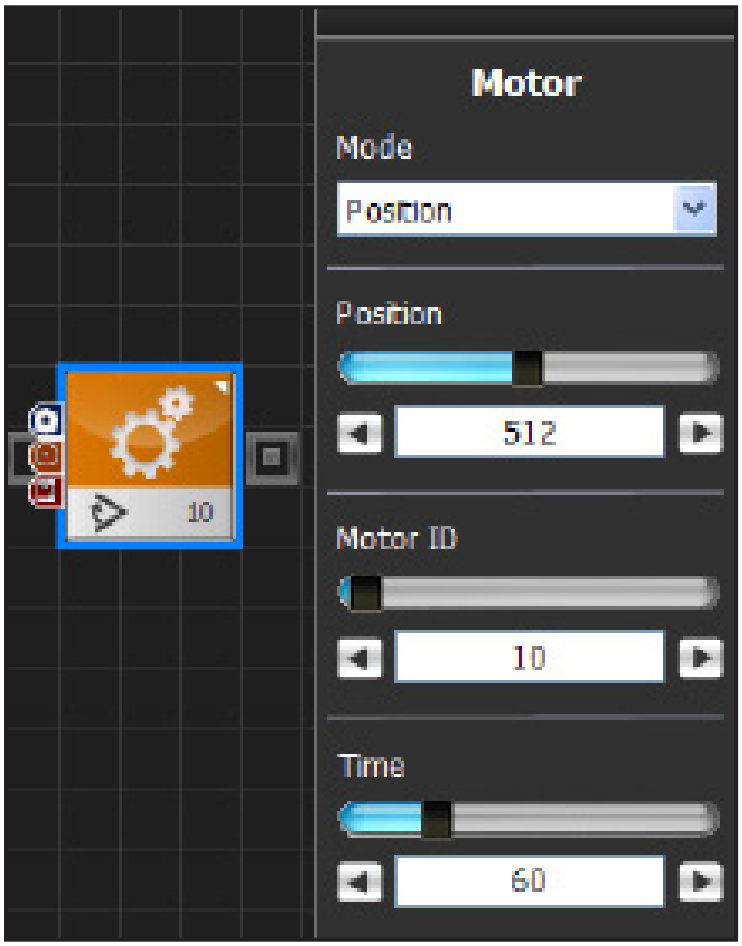
Module Pack	Module	예제 샘플
Motion (모션)	Move	Move 는 제어기 DRC 에 저장된 로봇의 모션을 가져와 프로그램에 적용하는 것입니다. 로봇 모션은 번호로 가져올 수 있으며, 번호별 모션 이름은 DR-SIM 에서 확인할 수 있습니다. DR-SIM 에서 편집된 특정 모션을 로봇에서 무한반복적으로 돌리는 프로그래밍을 해봅니다. 로봇 신뢰성 테스트나 전시등에서 활용도가 높고, 난이도가 있는 프로그램이므로 천천히 따라해보기 바랍니다.
	Motor	모터를 하나씩 제어하여 춤을 추는 프로그램 입니다.
	LED	제어기 DRC 의 버튼을 누르면 LED 가 켜지고 꺼지는 프로그램입니다. (With Button)
	Sound	리모콘의 버튼 입력(1 번~8 번)을 받아서 음을 울리도록 하는 프로그램입니다. (With IRRecieve)
Sensor (센서)	Sound Sensor	Sound Sensor 는 제어기 DRC 내부의 양쪽에 위치합니다. 왼쪽측면에서 박수를 치고 왼쪽 손을 들고, 오른쪽측면에서 박수를 치면 오른쪽 손을 드는 프로그래밍을 해봅니다. (Sample 두번째 안) 주변에 소음이 많으면, 각각 양쪽에서 소리를 구분하는게 어려워집니다. 한쪽에서 박수를 쳐도 양손을 모두 들거나, 불규칙적으로 반응합니다. 주변에 소음이 있더라도 로봇이 정확한 반응을 한다는 것을 보여주기위해 좀더 세분화된 프로그래밍을 할 필요가 있습니다. 처음 소리가 입력되었을시 다른 소리가 입력되지 않도록 강제로 Delay 를 줘서 한번 박수 칠 때 한번만 팔을 올리도록 정확도를 높이는 프로그램입니다.
	Light	빛의 세기에 따라 로봇의 모터를 동작하는 예제입니다. 외부의 빛이 어두워지면 로봇이 왼팔을 올립니다. (제어기 뒤쪽의 CDS센서를 손가락으로 가리면 빛의 들어오는 양이 없어 어두워지므로, 로봇이 왼팔을 올리도록 프로그래밍합니다.)
	Distance	PSD Digital(거리센서) : 벽이 일정거리 가까워지면 뒷 걸음질 치다가 우회전 한 후 전진하면 프로그램 입니다. PSD Analog(거리센서) : 벽이 가까워지면 좌회전 하면서 벽을 회피하는 프로그램 입니다.
	Dynamics	Accerlateration : 앞으로 넘어졌을 때 일어나는 프로그래밍, 뒤로 넘어졌을 때 일어나는 프로그램
Communication (통신)	IRRciever	리모콘의 1 번부터 8 번까지 음높이를 설정하여 DRC 제어기 사운드가 울리도록 하는 프로그래밍입니다. 1~8 번은 도~시까지 매칭합니다. (With Sound)
	Button	제어기 DRC 의 버튼을 누르면 제어기 뒤편의 LED 가 반응하는 프로그램입니다. (with LED)

소프트웨어 : DR-Visual Logic

모듈별 프로그래밍 : 모션(Motion) 〉 Motor

Motor 설명

Motor 모듈에는 두 가지 동작 모드가 있습니다. 하나는 위치 제어 모드, 하나는 속도 제어 모드입니다.



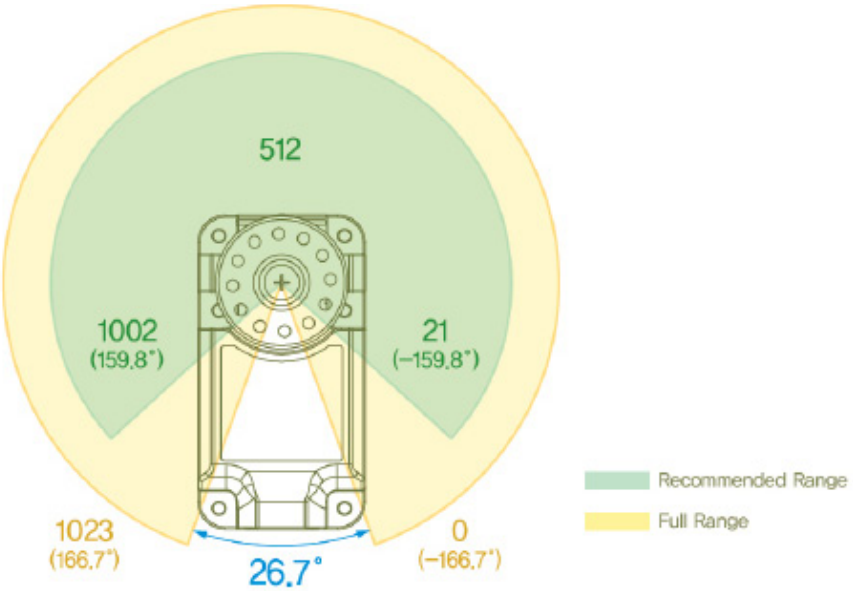
1 위치 제어 모드

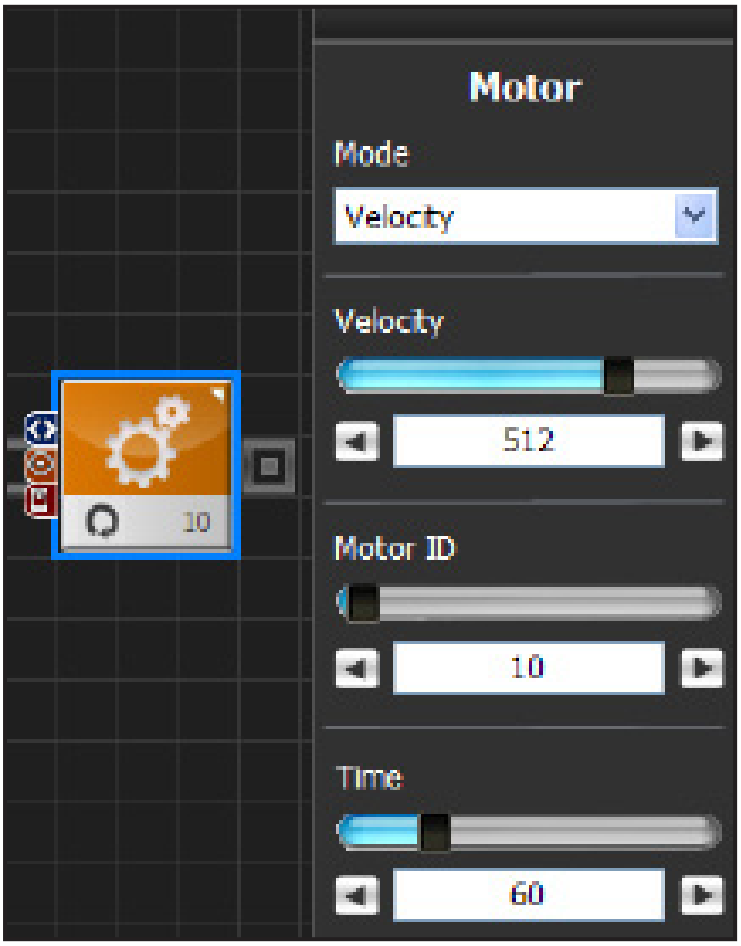
위치 제어 모드는 원하는 모터를 특정 위치로 이동 시키는 모드입니다.

Position은 −127~1152의 값을 가질 수 있는데, 서보의 초기 공장 설정에서 허용되는 값은 21~1002이며 이 범위를 벗어나는 값은 모터의 최소/최대 위치값과 위치 보정치가 변경되면 나올 수 있는 값들입니다. 모터의 정위치 값은 512로, 조립할 때의 기준이 되는 위치도 512입니다. 정상적으로 Hovis의 모든 모터 위치를 512로 보내면, 양팔을 90도로 벌리고 선 자세를 취하게 됩니다. 서보의 제어 범위와 정위치에 대한 정리된 그림은 아래를 참조 하세요.

Motor ID는 제어할 서보의 ID입니다.

Time은 서보가 목표 위치로 이동할 시간을 나타냅니다. 1당 11.2ms의 실제 시간을 의미하며, 가령 100으로 설정할 경우 1.12초 동안 원하는 위치로 이동하게 됩니다.





2 속도 제어 모드

속도 제어 모드는 원하는 모터를 특정 속도로 무한 회전 시키는 모드입니다.

Velocity는 -1023~1023의 값을 가질 수 있는데, 숫자가 클수록 큰 출력으로 빠르게 회전을 시키게 됩니다. 양수와 음수의 차이는 회전 방향입니다.

Motor ID는 제어할 서보의 ID입니다.

Time은 서보가 목표 속도에 도달할 시간을 시간을 나타냅니다. 1당 11.2ms의 실제 시간을 의미하며, 가령 100으로 설정할 경우 1.12초 동안 목표 속도로 서서히 속도가 변하게 됩니다.

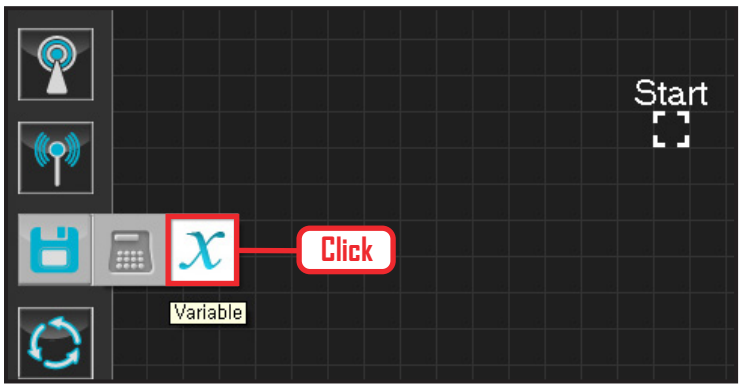
.

Motor 예제 따라하기

예제설명

흔히 로봇 모션은 각각의 모터를 일일히 제어하고 종합하여 동작을 만들게 됩니다. 그 복잡성으로 인해 모터를 일일히 제어하는 것보다는 DR-SIM(Motion Editor) 같은 툴을 제공하여 모션을 쉽게 만드는 게 보편적입니다.

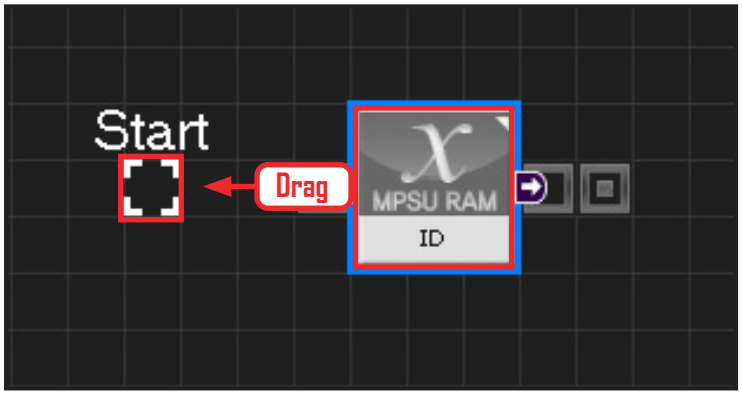
이번 프로그래밍은 Motion Editor 를 사용하지 않고, Task Editor 를 이용하여 모터를 하나하나 조작하여 연속되는 모션을 만들어보고자 합니다. 모션 목표는 웨이브 댄스를 추는 로봇입니다. 완료되었을 때 흥미로운 동작이니 천천히 따라해보세요.



01 변수 지정

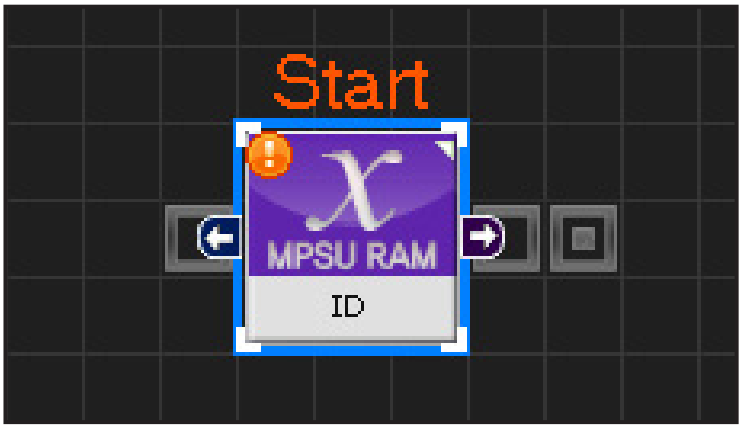
로봇을 동작시킨다는 것은 로봇의 서보 모터를 동작시킨다는 의미입니다. 서보가 스스로 움직일 수 있는 상태로 값을 지정해주어야 합니다.

Data 〉 Variable 모듈을 클릭합니다.



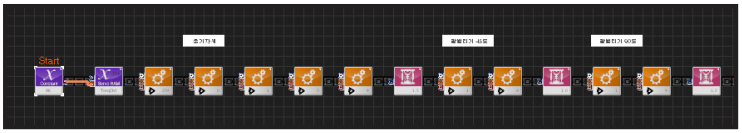
02 시작

모듈의 왼쪽 연결선을 Start Point 에 드래그하여 정확히 도킹을 시킵니다.



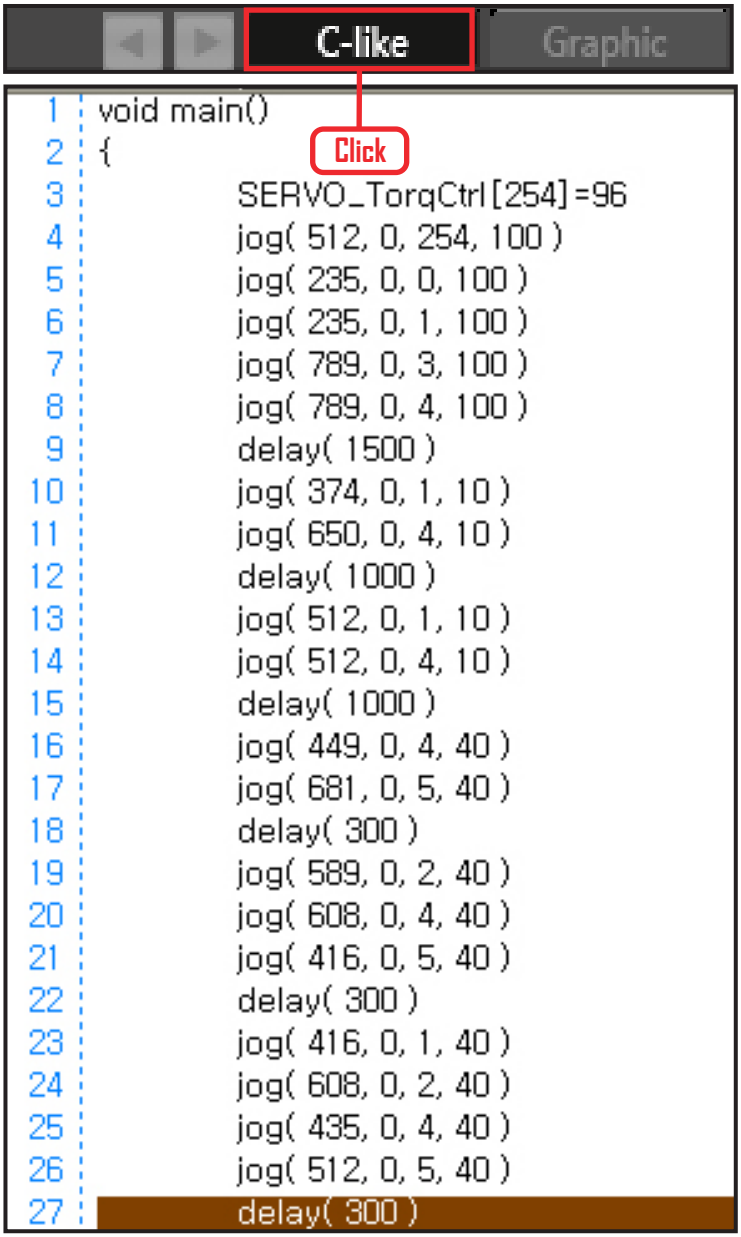
03 프로그래밍 시작

모듈과 Start Point 가 정확히 도킹하면 왼쪽과 같이 활성화된 칼라 이미지 모듈로 변합니다.
그럼 프로그래밍이 시작되었다는 의미입니다.



04 전체 프로그래밍

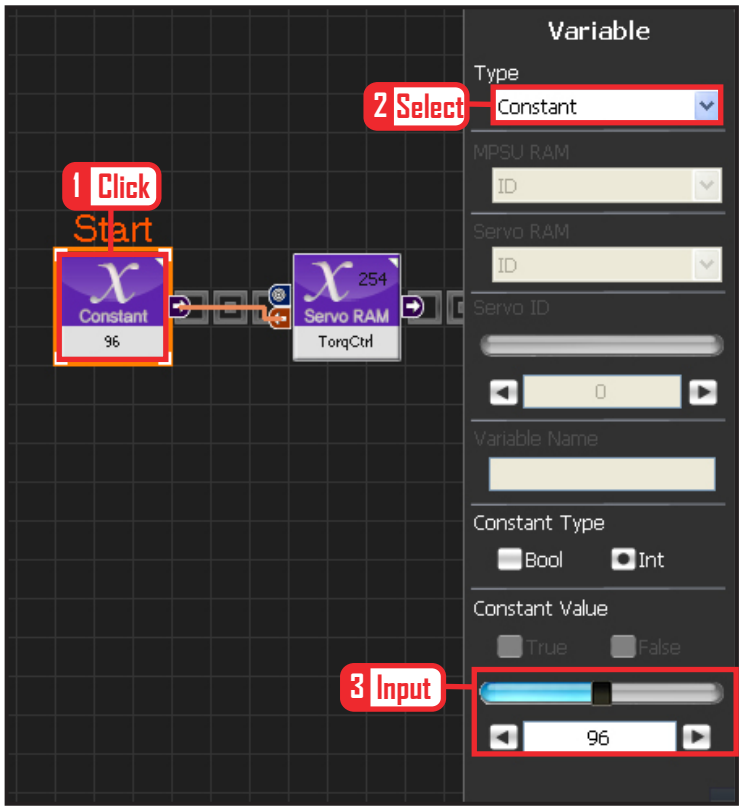
모터를 제어하는 전체 프로그래밍입니다.



05 C-Like 보기

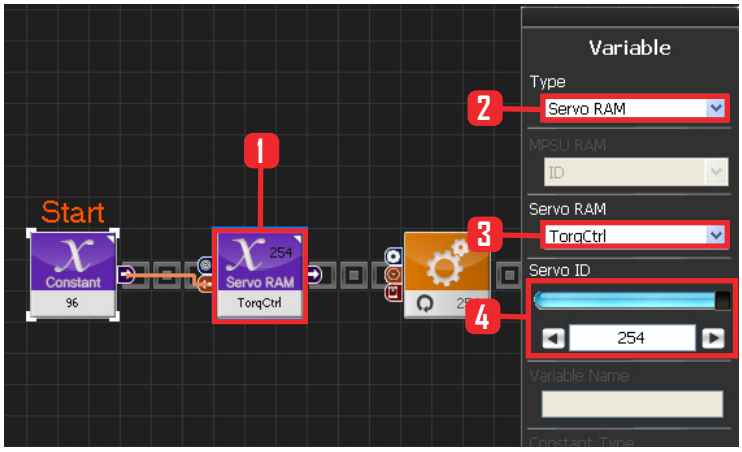
오른쪽 상단의 Graphic 탭에서 C-like 탭을 클릭하면 왼쪽과 같은 Task 프로그래밍 화면이 나옵니다.
모터모듈을 이용한 전체 프로그래밍 화면입니다.
C와 유사한 문법 구조를 가지고 있으므로 C 문법 선행학습 효과도 있습니다.
각 모듈별로 클릭하면 커서가 따라서 움직이므로 모듈별로 Text로 어떻게 변환하는지 확인할 수 있습니다.

```
28          jog( 575, 0, 1, 40 )
29          jog( 343, 0, 2, 40 )
30          jog( 512, 0, 4, 40 )
31          delay( 300 )
32          jog( 512, 0, 1, 40 )
33          jog( 512, 0, 2, 40 )
34          delay( 500 )
35          jog( 374, 0, 1, 10 )
36          jog( 650, 0, 4, 10 )
37          delay( 200 )
38          jog( 235, 0, 1, 10 )
39          jog( 789, 0, 4, 10 )
40          delay( 200 )
41  }
```



06 상수 설정

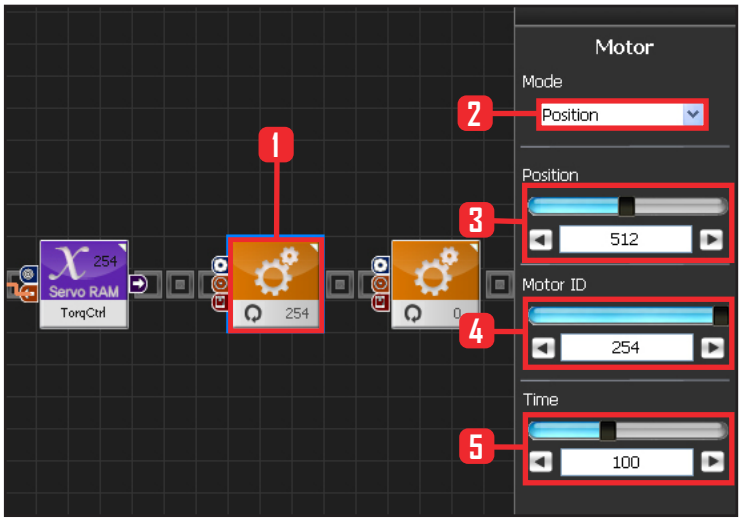
서보 모터를 스스로 움직일 수 있는 상태로 만드는 과정입니다.
Variable Type 을 Constant 로 선택합니다.
속성중에 Constant Value 값을 96 으로 설정합니다.
서보의 TorqControl 레지스터에 96(0x60) 이라는 값이 들어가면 서보가 움직일 수 있는 상태가 됩니다.
그 값은 Output 커넥터를 통하여 뒤 모듈의 토크값에 전달합니다.



07 모든 서보에 적용

앞에서 받은 96 상수값을 모든 서보에 적용하는 과정입니다.

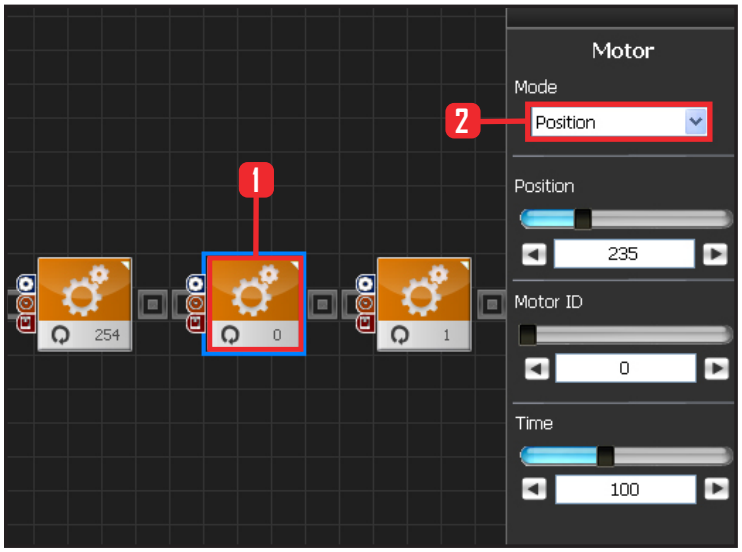
Variable 〉 Type : Servo RAM을 선택합니다.
Servo RAM : TorqCtrl 을 선택합니다.
Servo ID : 254 를 선택합니다. 254는 연결되어있는 모든 서보에 적용하겠다는 의미입니다.



08 모든 서보모터 각도 설정

모든 서보모터의 각도를 중앙에 보내는 과정입니다.

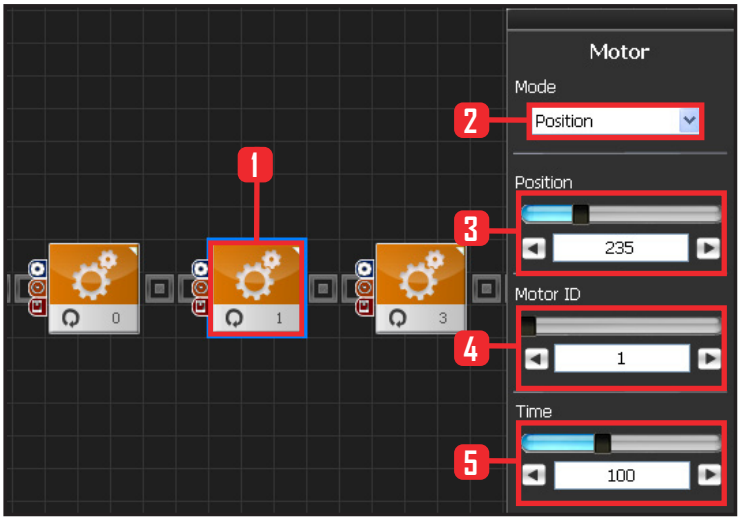
Motion 〉 Moter 를 선택합니다.
Mode : Positon 으로 선택합니다. 각도를 조절합니다.
Position : 512 로 설정합니다. 512 번은 모터를 중앙으로 보낸다는 의미입니다.
Motor ID : 254 로 설정합니다. 254 는 모든 모터에 적용하겠다는 의미입니다.
Time : 100 으로 설정합니다. 단위는 1당 11.2ms로, 100은 약 1.12초를 의미합니다.
1.12초동안 원하는 각도로 이동시킨다는 의미입니다.



09 모터 0번 (오른쪽 어깨) 설정

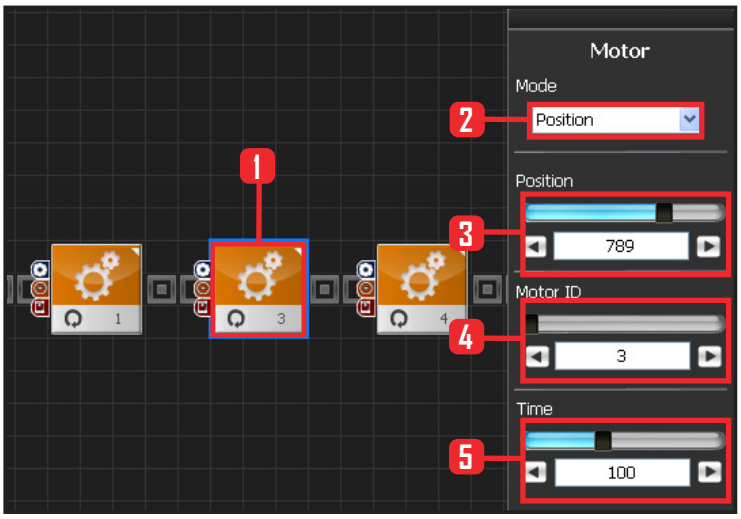
1단계 : 초기자세 잡기
차려자세(기본자세) 만들기
모든 로봇의 모터의 각도를 중앙으로 정렬하면 휴머노이드에서는 팔을 좌우로 뻗게 됩니다. 이것을 차려자세로 되돌려 놓아야만 기본 자세를 유지하여 동작시키기가 용이해집니다.

Motion 〉 Moter 를 선택합니다.
Mode : Position 으로 선택합니다.
Position : 235 로 설정합니다. 235 는 수평으로 들고 있던 오른손을 수직으로 내려갈 수 있게 모터를 돌리게 됩니다.
Motor ID : 0 으로 설정합니다. 오른쪽 어깨 모터 ID가 0번입니다.
Time : 100 으로 설정합니다. 약 1.12초동안 원하는 각도로 이동합니다.



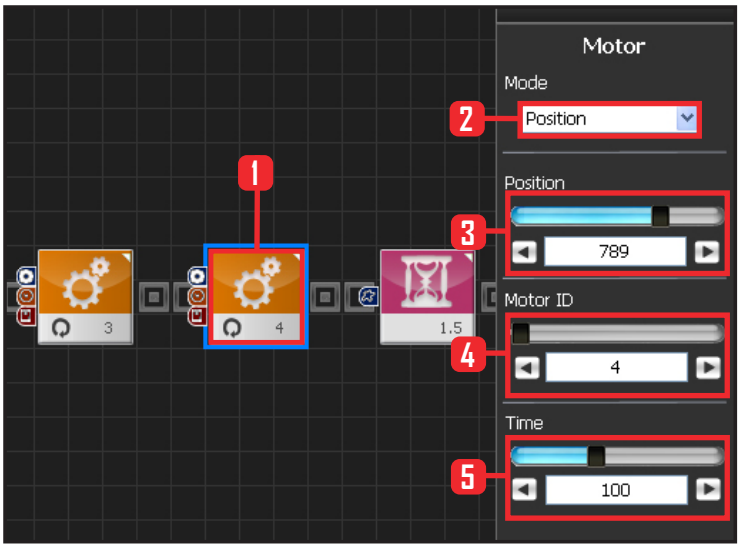
10 모터 1번 (오른쪽 팔) 설정

Mode : Postion 으로 선택합니다.
Position : 235 로 선택합니다. 235는 수평으로 되어 있던 팔을 수직으로 내리게 됩니다.
Motor ID : 1 로 설정합니다. 오른쪽 위쪽 팔 어깨와 닿는 모터가 1번 모터입니다.
Time : 100 으로 설정합니다. 약 1.12초동안 원하는 각도로 이동합니다.



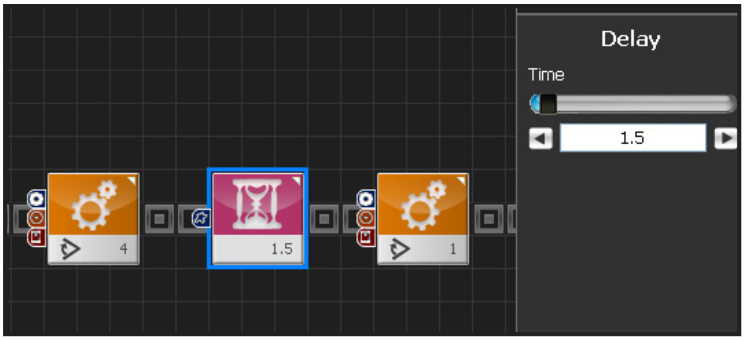
11 모터 3번(왼쪽 어깨) 설정

Mode : Position 으로 선택합니다.
Position : 789 로 설정합니다. 789 는 수평으로 들고 있던 왼손을 수직으로 내려갈 수 있게 모터를 돌리게 됩니다.
Motor ID : 3 으로 설정합니다. 왼쪽 어깨 모터 ID가 3번입니다.
Time : 100 으로 설정합니다. 약 1.12초동안 원하는 각도로 이동합니다.



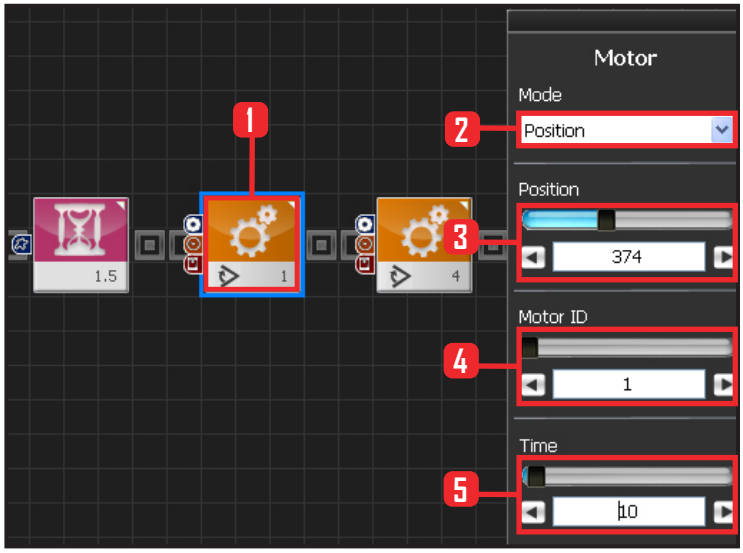
12 모터 4번(왼쪽 팔) 설정

Mode : Postion 으로 선택합니다.
Position : 789 로 선택합니다. 789는 수평으로 되어있던 팔을 수직으로 내리게 됩니다.
Motor ID : 4 로 설정합니다. 왼쪽 위쪽 팔 어깨와 닿는 모터가 4번 모터입니다.
Time : 100 으로 설정합니다. 약 1.12초동안 원하는 각도로 이동합니다.



13 Delay 지연시키기

1.5 초 동안 기다립니다.

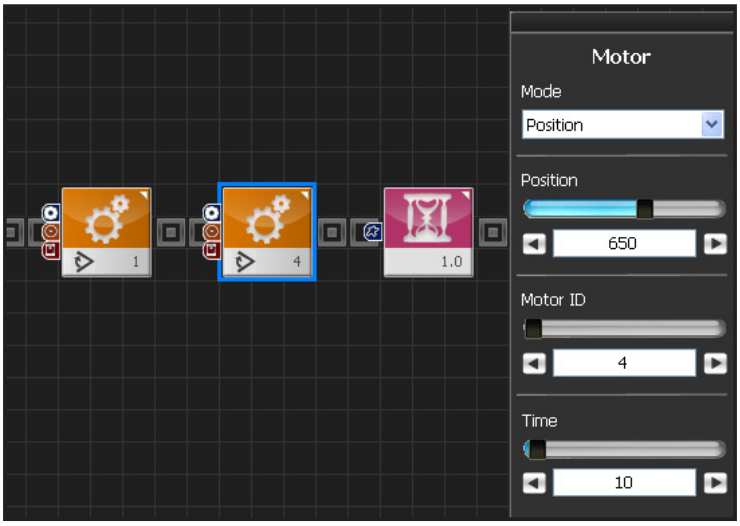


14 모터 1번(오른쪽 팔) 설정

2단계 : 45도 각도로 팔벌리기

로봇이 춤을 추기 위한 준비동작으로 팔을 45도 각도로 만듭니다.

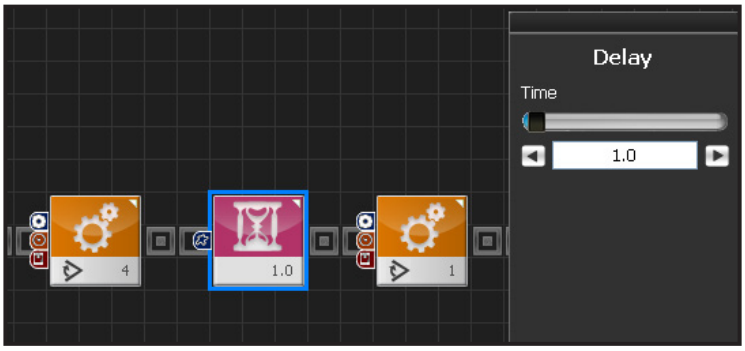
Motion 〉 Moter 를 선택합니다.
Mode : Position 으로 선택합니다.
Position : 374 로 설정합니다. 374 는 오른쪽 팔을 45도로 만듭니다.
Motor ID : 1 로 설정합니다. 오른쪽 팔 윗부분 모터 ID가 1번입니다.
Time : 10 으로 설정합니다. 약 0.112초동안 원하는 각도로 이동합니다.



15 모터 4번(오른쪽 팔) 설정

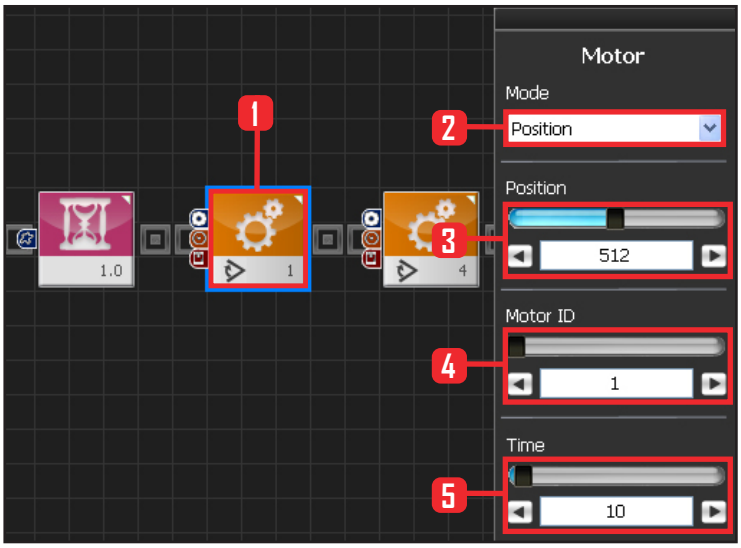
마찬가지로 왼쪽팔 윗부분 모터인 4번을 45도로 만듭니다.

Motion 〉 Moter 를 선택합니다.
Mode : Position 으로 선택합니다.
Position : 650 로 설정합니다. 650 는 왼쪽 팔을 45도로 만듭니다.
Motor ID : 4 로 설정합니다. 왼쪽 팔 윗부분 모터 ID가 4번입니다.
Time : 10 으로 설정합니다. 약 0.112초동안 원하는 각도로 이동합니다.



16 Delay 지연시키기

1.0 초 동안 기다립니다.

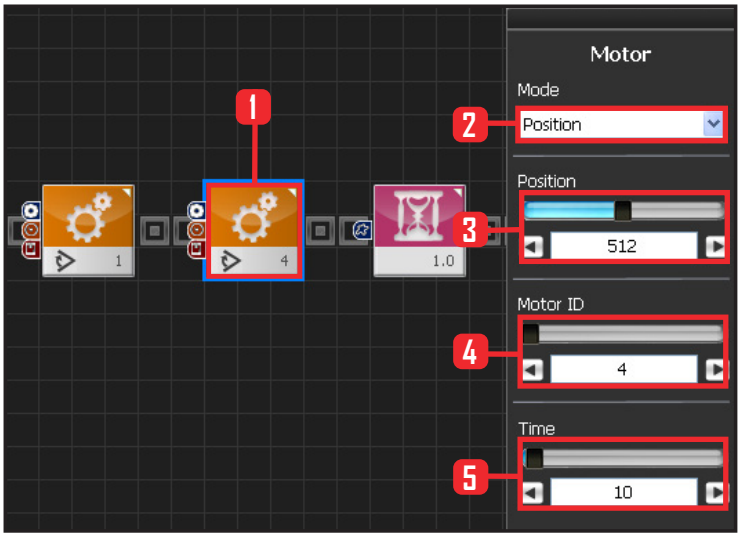


17 모터 1번(오른쪽 팔) 설정

3단계 : 90도 각도로 팔벌리기

로봇의 팔을 90도로 올려서 웨이브 댄스추기 시작단계로 접어듭니다.

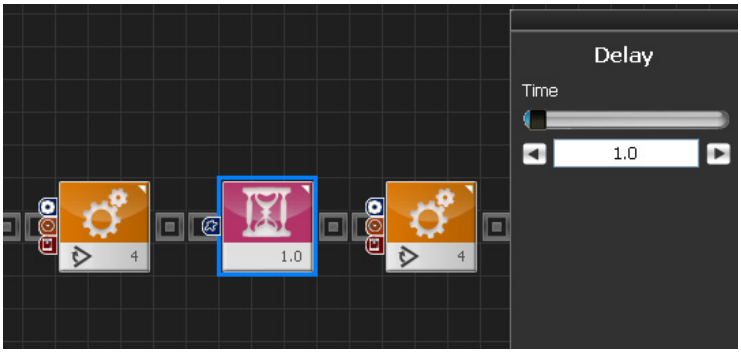
Motion 〉 Moter 를 선택합니다.
Mode : Position 으로 선택합니다.
Position : 512 로 설정합니다. 512 는 팔을 90각도로 만듭니다. 모터의 중앙값이기도 합니다. 모든 모터를 중앙값에 놓으면 로봇은 양팔벌려 자세가 됩니다.
Motor ID : 1 로 설정합니다. 오른쪽 팔 윗부분 모터 ID 가 1번입니다.
Time : 10 으로 설정합니다. 약 0.112초동안 원하는 각도로 이동합니다.



18 모터 4번(왼쪽 팔) 설정

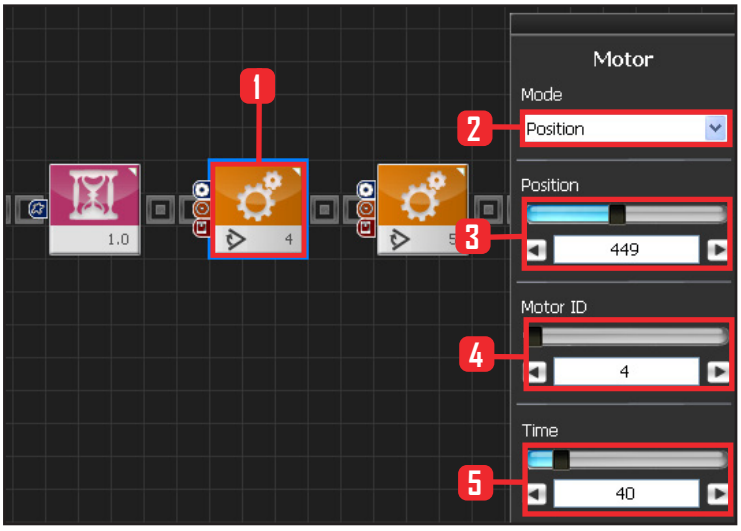
로봇의 팔을 90도로 올려서 웨이브 댄스추기 시작단계로 접어듭니다.

Motion 〉 Moter 를 선택합니다.
Mode : Position 으로 선택합니다.
Position : 512 로 설정합니다. 512 는 팔을 90도로 만듭니다. 모터의 중앙값이기도 합니다. 모든 모터를 중앙값에 놓으면 로봇은 양팔벌려 자세가 됩니다.
Motor ID : 4 로 설정합니다. 왼쪽 팔 윗부분 모터 ID 가 4번입니다.
Time : 10 으로 설정합니다. 약 0.112초동안 원하는 각도로 이동합니다.



19 Delay 지연시키기

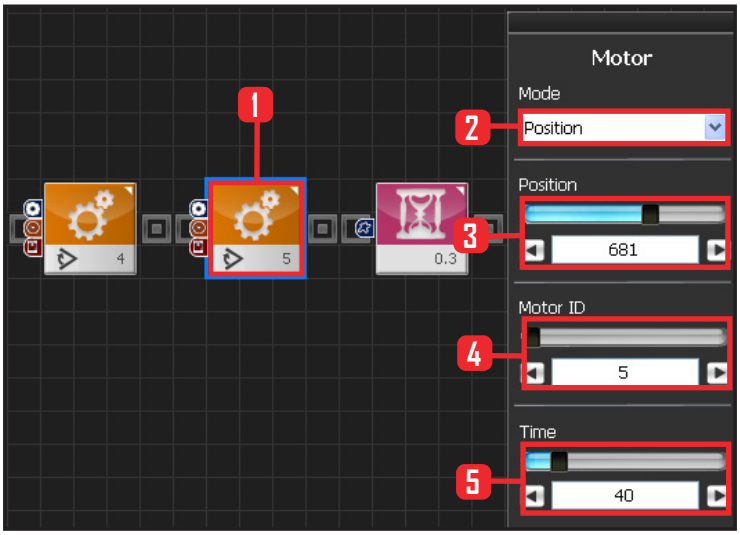
1.0 초 동안 기다립니다.



20 모터 4번(왼쪽 팔) 설정

4단계 : 웨이브 1단계
왼쪽팔부터 웨이브 시작합니다.

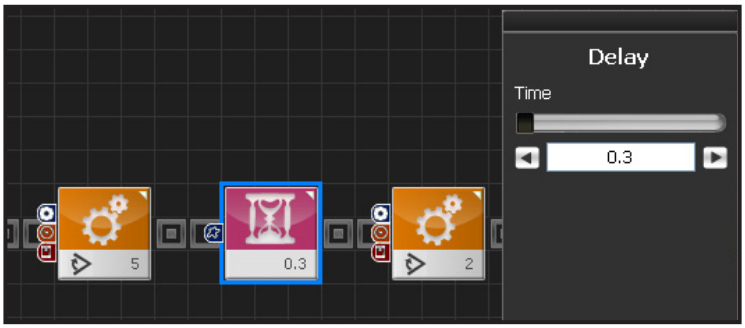
Motion 〉 Moter 를 선택합니다.
Mode : Position 으로 선택합니다.
Position : 449 로 설정합니다. 449 는 팔을 웨이브 시작단계로 만듭니다.
Motor ID : 4 로 설정합니다. 왼쪽 팔 윗부분 모터 ID 가 4번입니다.
Time : 40 으로 설정합니다. 약 0.448초동안 원하는 각도로 이동합니다.



21 모터 5번(왼쪽 팔 아래) 설정

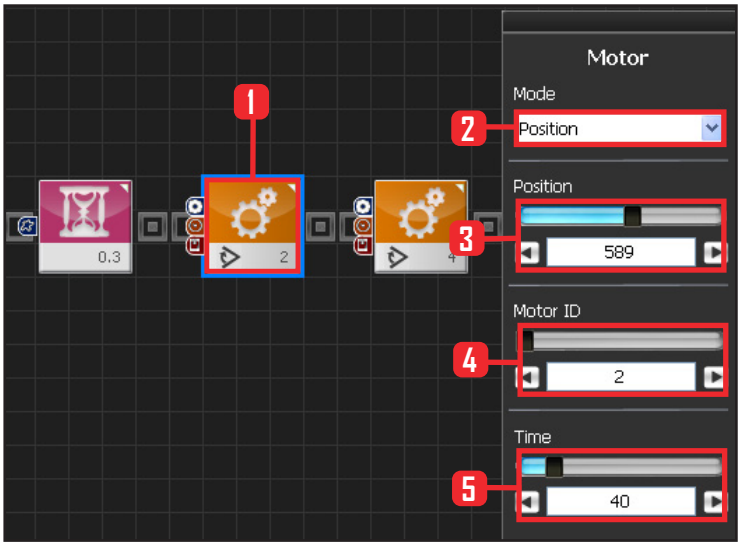
왼쪽팔 아랫부분 웨이브 시작합니다.

Motion 〉 Moter 를 선택합니다.
Mode : Position 으로 선택합니다.
Position : 681 로 설정합니다.
Motor ID : 5 로 설정합니다. 왼쪽 팔 아랫부분 모터 ID가 5번입니다.
Time : 40 으로 설정합니다. 약 0.448초동안 원하는 각도로 이동합니다.



22 Delay 지연시키기

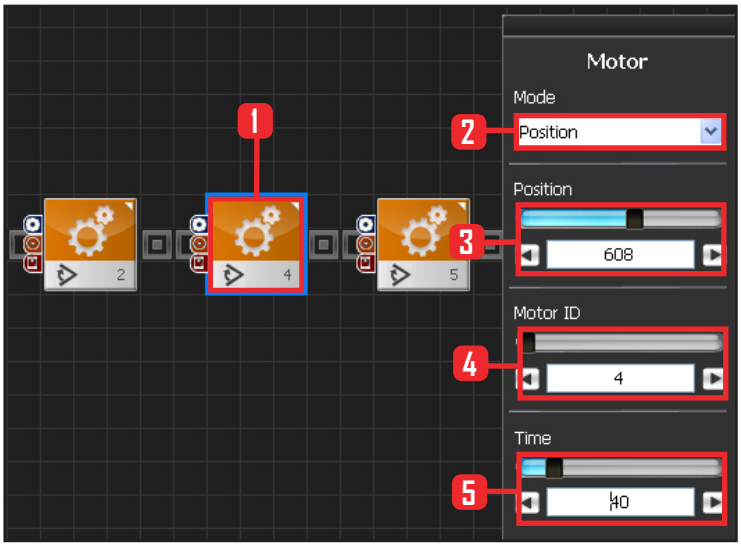
0.3 초 동안 기다립니다.
춤이 시작된 부분이기 때문에 Delay Time 을 적게 줍니다.



23 모터 2번(오른쪽 팔 아랫부분) 설정

5단계 : 웨이브 2단계
오른쪽 팔 아랫부분 웨이브 시작합니다.

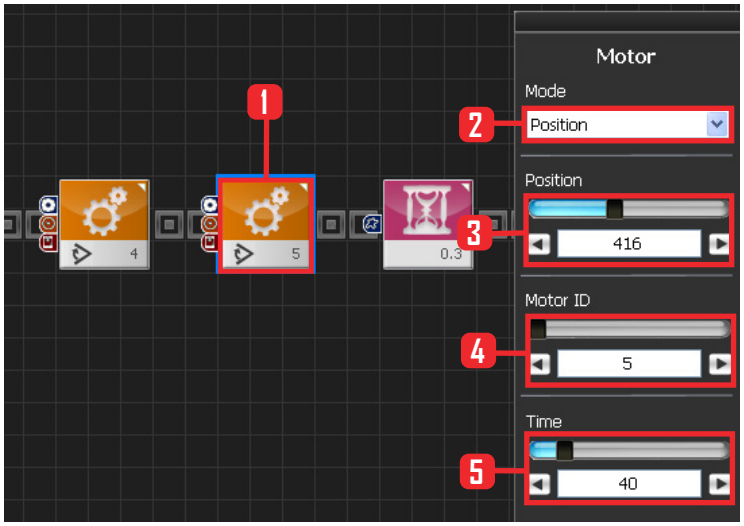
Motion 〉 Moter 를 선택합니다.
Mode : Position 으로 선택합니다.
Position : 589 로 설정합니다.
Motor ID : 2 로 설정합니다. 오른쪽 팔 아랫부분 모터 ID가 2번입니다.
Time : 40 으로 설정합니다. 약 0.448초동안 원하는 각도로 이동합니다.



24 모터 4번(왼쪽 팔 윗부분) 설정

모터의 각도를 조금씩 변화시켜 웨이브 효과를 줍니다.

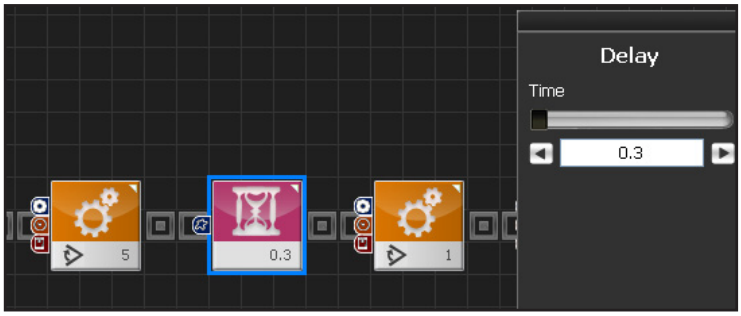
Motion 〉 Moter 를 선택합니다.
Mode : Position 으로 선택합니다.
Position : 608 로 설정합니다.
Motor ID : 4 로 설정합니다.
Time : 40 으로 설정합니다. 약 0.448초동안 원하는 각도로 이동합니다.



25 모터 5번(왼쪽 팔 아랫부분) 설정

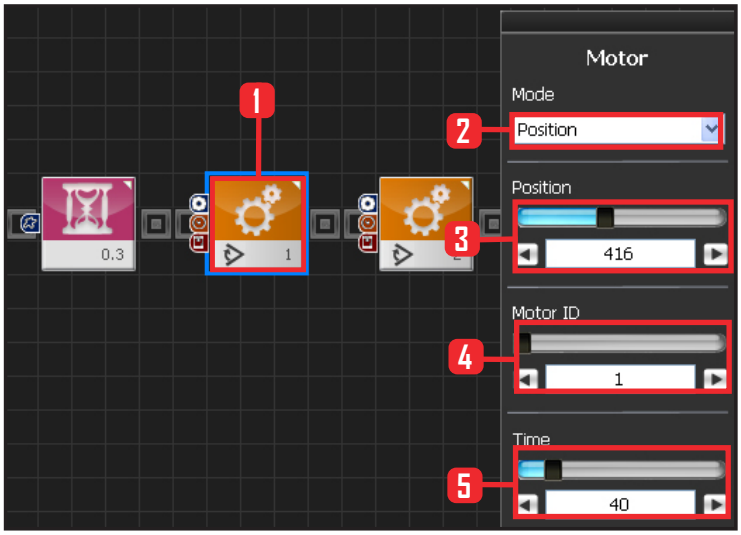
모터의 각도를 조금씩 변화시켜 웨이브 효과를 줍니다.

Motion〉Moter 를 선택합니다.
Mode : Position 으로 선택합니다.
Position : 416 로 설정합니다.
Motor ID : 5 로 설정합니다.
Time : 40 으로 설정합니다. 약 0.448초동안 원하는 각도로 이동합니다.



26 Delay 지연시키기

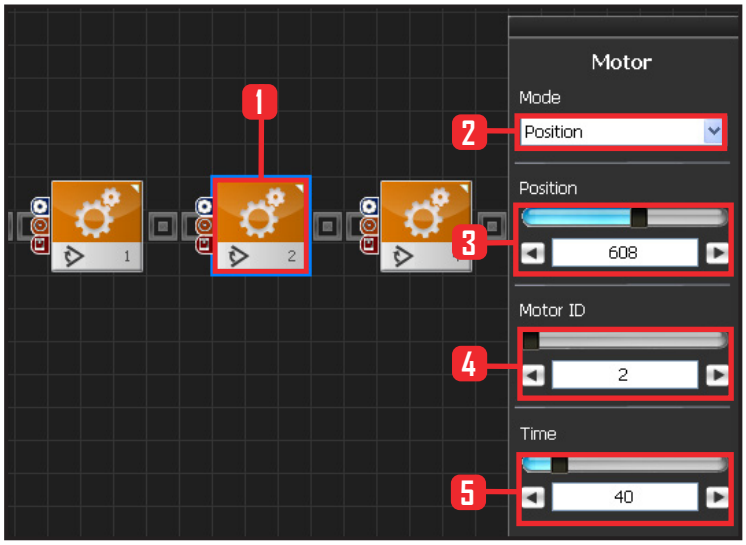
0.3 초 동안 기다립니다.
춤이 시작된 부분이기 때문에 Delay Time 을 적게 줍니다.



27 모터 1번(오른쪽 팔 윗부분) 설정

6단계 : 웨이브 3단계
웨이브 줬던 모터를 원상복귀 시킵니다.

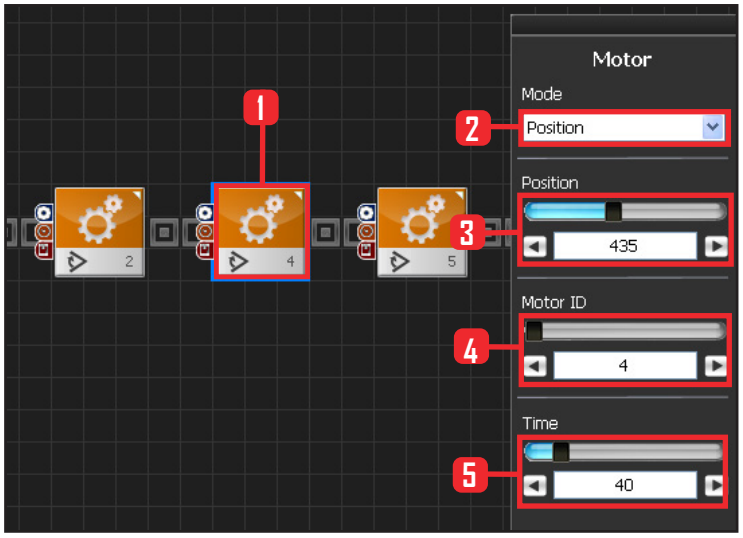
Motion〉Moter 를 선택합니다.
Mode : Position 으로 선택합니다.
Position : 416 로 설정합니다.
Motor ID : 1 로 설정합니다.
Time : 40 으로 설정합니다. 약 0.448초동안 원하는 각도로 이동합니다.



28 모터 2번(오른쪽 팔 아랫부분) 설정

웨이브 줬던 모터를 원상복귀 시킵니다.

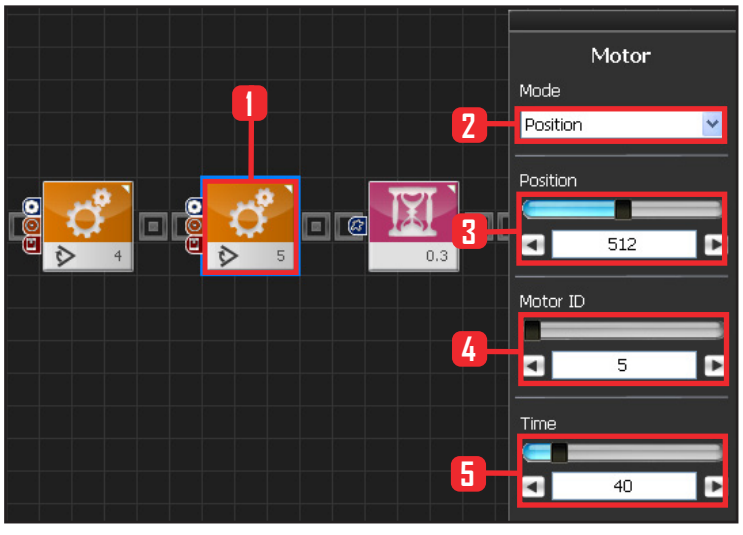
Motion〉Moter 를 선택합니다.
Mode : Position 으로 선택합니다.
Position : 608 로 설정합니다.
Motor ID : 2 로 설정합니다.
Time : 40 으로 설정합니다. 약 0.448초동안 원하는 각도로 이동합니다.



29 모터 4번(왼쪽 윗부분) 설정

웨이브 줬던 모터를 원상복귀 시킵니다.

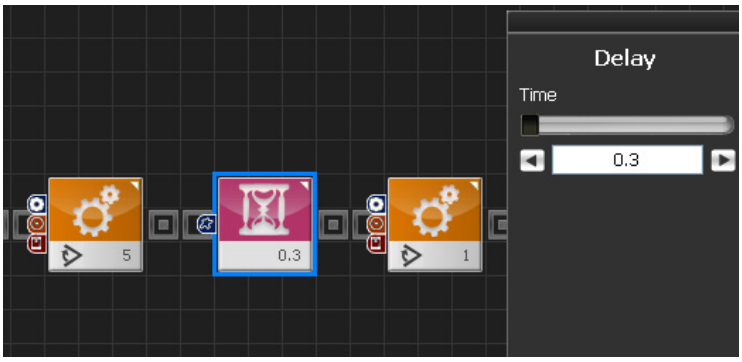
Motion〉Moter 를 선택합니다.
Mode : Position 으로 선택합니다.
Position : 435 로 설정합니다.
Motor ID : 4 로 설정합니다.
Time : 40 으로 설정합니다. 약 0.448초동안 원하는 각도로 이동합니다.



30 모터 5번(왼쪽팔 아랫부분) 설정

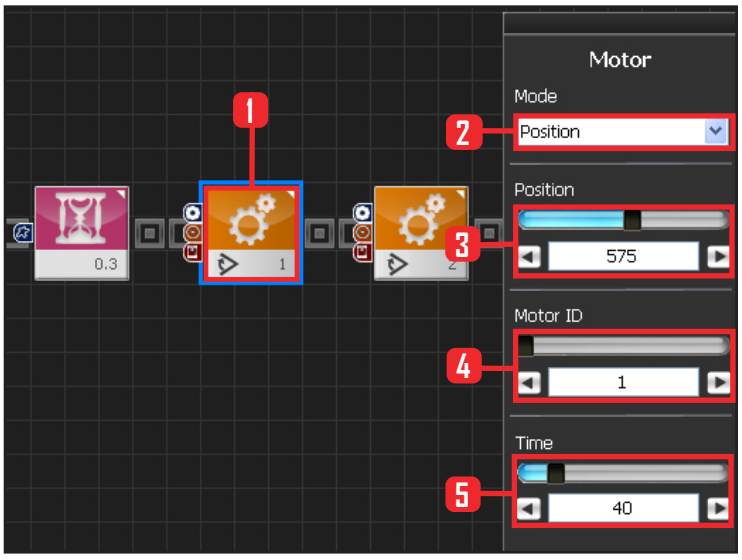
웨이브 줬던 모터를 원상복귀 시킵니다.

Motion〉Moter 를 선택합니다.
Mode : Position 으로 선택합니다.
Position : 512 로 설정합니다.
Motor ID : 5 로 설정합니다.
Time : 40 으로 설정합니다. 약 0.448초동안 원하는 각도로 이동합니다.



31 Delay 지연시키기

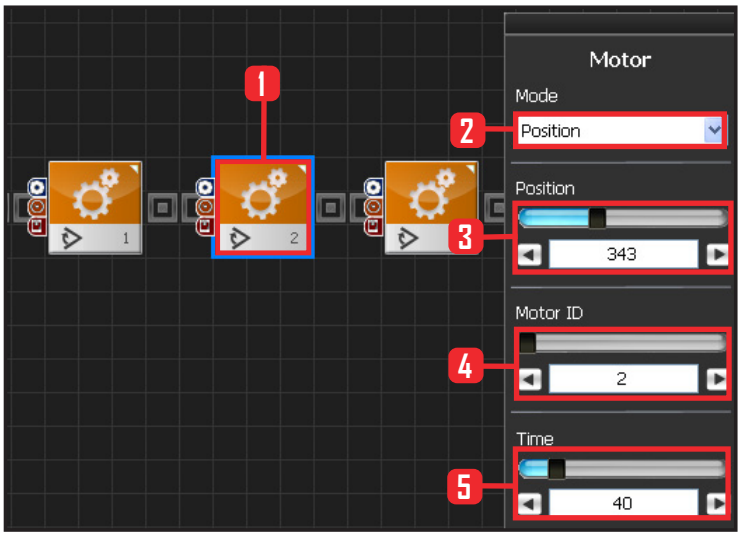
0.3 초 동안 기다립니다.
춤이 시작된 부분이기 때문에 Delay Time 을 적게 줍니다.



32 모터 1번(오른쪽 팔 윗부분) 설정

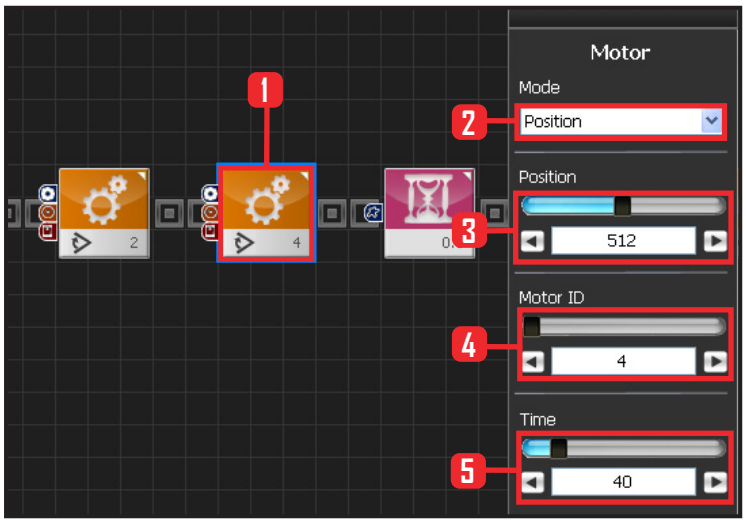
7단계 : 웨이브 4단계
웨이브를 마무리합니다.

Motion 〉Moter 를 선택합니다.
Mode : Position 으로 선택합니다.
Position : 575 로 설정합니다.
Motor ID : 1 로 설정합니다.
Time : 40 으로 설정합니다. 약 0.448초동안 원하는 각도로 이동합니다.



33 모터 2번(오른쪽 팔 아랫부분) 설정

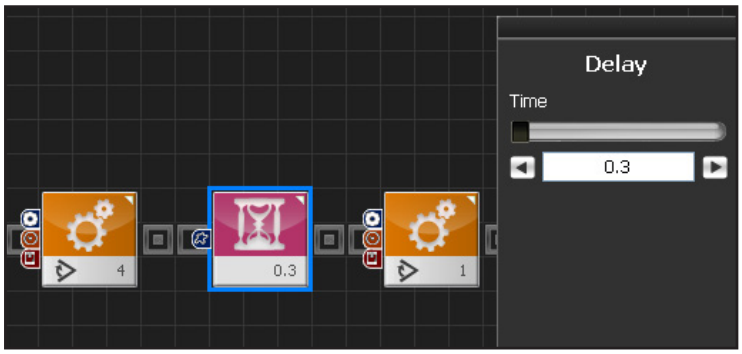
웨이브를 마무리합니다.
Motion 〉Moter 를 선택합니다.
Mode : Position 으로 선택합니다.
Position : 343 로 설정합니다.
Motor ID : 2 로 설정합니다.
Time : 40 으로 설정합니다. 약 0.448초동안 원하는 각도로 이동합니다.



34 모터 4번(왼쪽 팔 윗부분) 설정

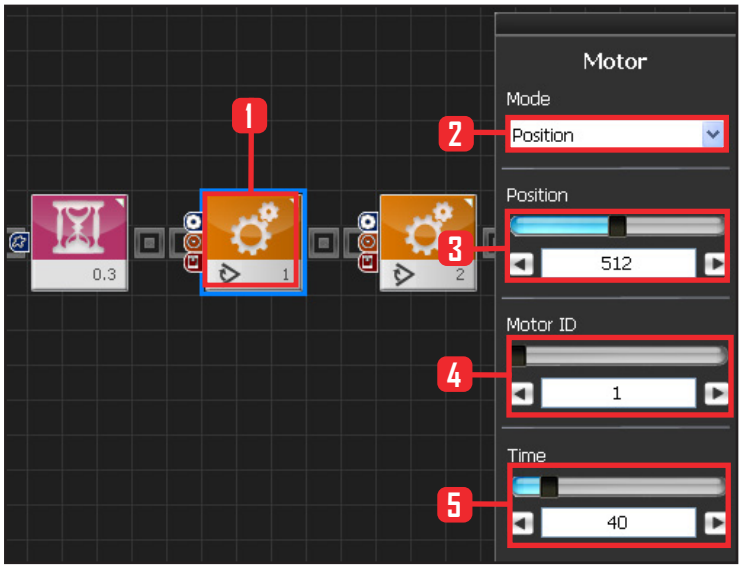
웨이브를 마무리 합니다.

Motion 〉 Moter 를 선택합니다.
Mode : Position 으로 선택합니다.
Position : 512 로 설정합니다.
Motor ID : 4 로 설정합니다.
Time : 40 으로 설정합니다. 약 0.448초동안 원하는 각도로 이동합니다.



35 Delay 지연시키기

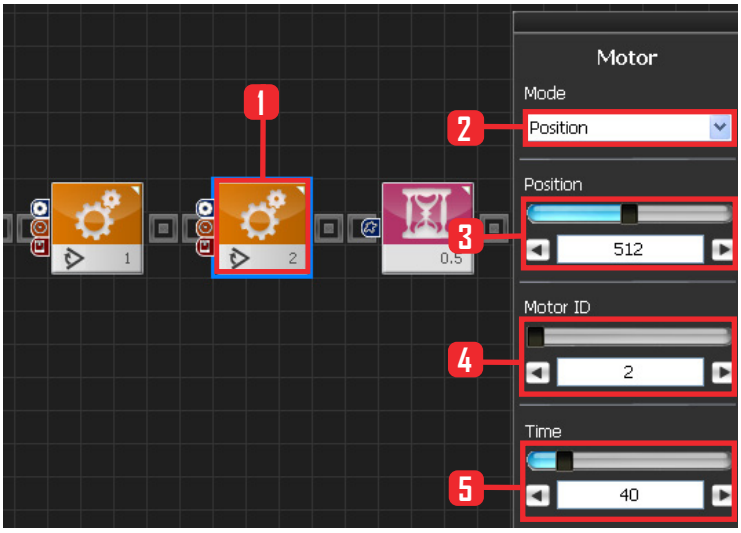
0.3 초 동안 기다립니다.
춤이 시작된 부분이기 때문에 Delay Time 을 적게 줍니다.



36 모터 1번(오른쪽 팔 윗부분) 설정

8단계 : 웨이브 5단계
로봇의 팔을 양팔 벌려 처음자세로 원상복귀시킵니다.

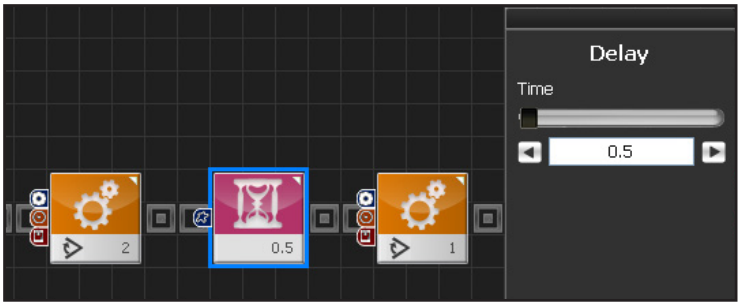
Motion〉Moter 를 선택합니다.
Mode : Position 으로 선택합니다.
Position : 512 로 설정합니다.
Motor ID : 1 로 설정합니다.
Time : 40 으로 설정합니다. 약 0.448초동안 원하는 각도로 이동합니다.



37 모터 2번(오른쪽 팔 아랫부분) 설정

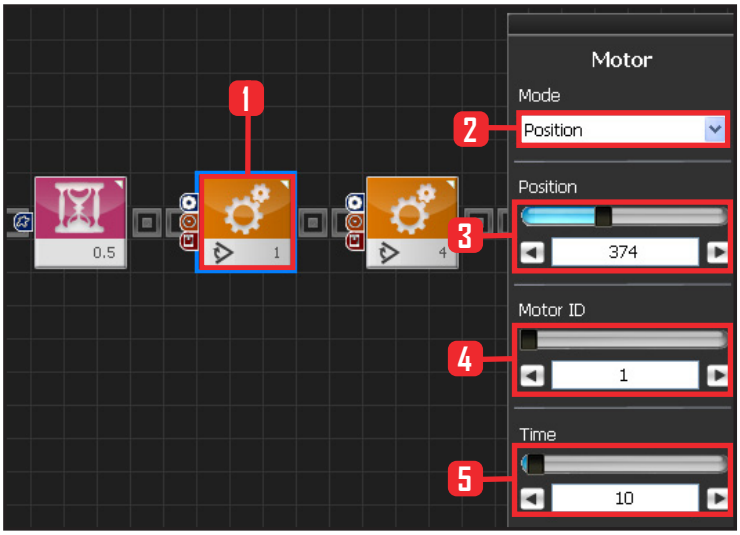
로봇의 팔을 양팔 벌려 처음자세로 원상복귀시킵니다.

Motion 〉 Moter 를 선택합니다.
Mode : Position 으로 선택합니다.
Position : 512 로 설정합니다.
Motor ID : 2 로 설정합니다.
Time : 40 으로 설정합니다. 약 0.448초동안 원하는 각도로 이동합니다.



38 Delay 지연시키기

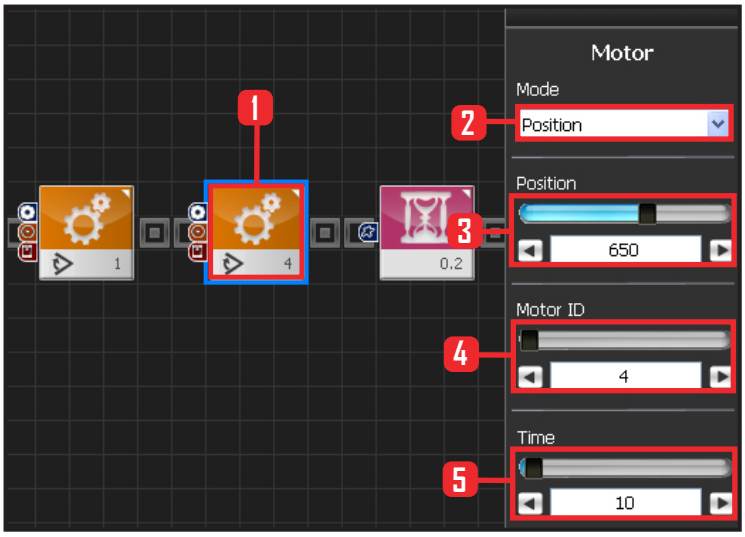
0.5 초 동안 기다립니다.



39 모터 1번(오른쪽 팔 윗부분) 설정

9단계 : 45도 각도로 팔내리기
차려자세로 복귀시킵니다. 우선 45도 각도로 이동합니다.

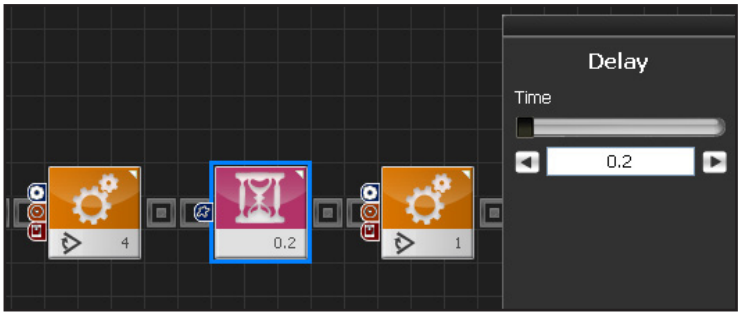
Motion 〉 Moter 를 선택합니다.
Mode : Position 으로 선택합니다.
Position : 374 로 설정합니다.
Motor ID : 1 로 설정합니다.
Time : 10 으로 설정합니다.



40 모터 4번(왼쪽 팔 윗부분) 설정

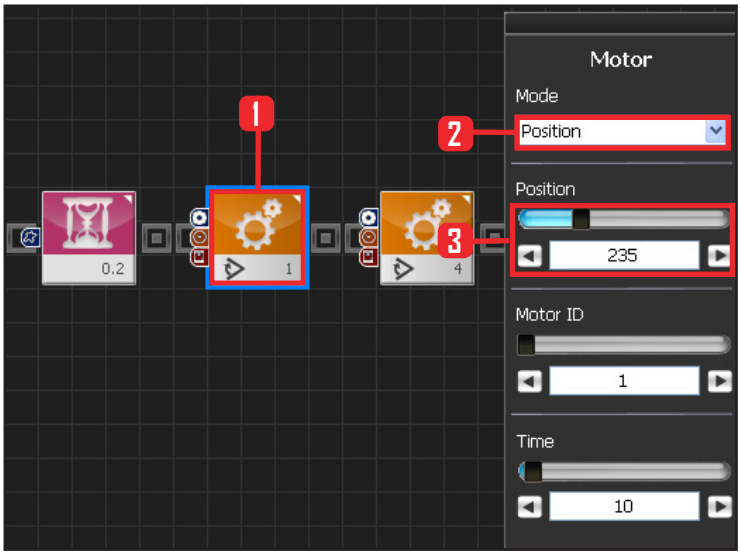
차려자세로 복귀시킵니다. 우선 45도 각도로 이동합니다.

Motion 〉 Moter 를 선택합니다.
Mode : Position 으로 선택합니다.
Position : 650 으로 설정합니다.
Motor ID : 4 로 설정합니다.
Time : 10 으로 설정합니다.



41 Delay 지연시키기

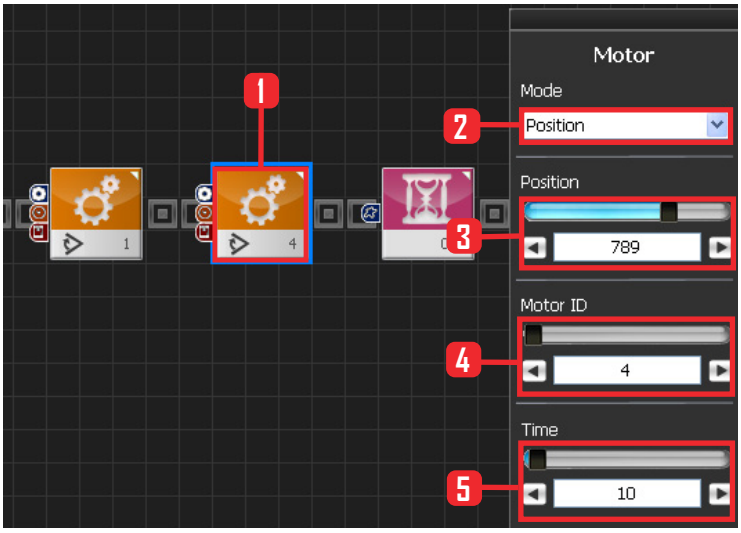
0.2 초 동안 기다립니다.



42 모터 1번(오른쪽 팔 윗부분) 설정

10단계 : 춤 완료
차려자세로 복귀시킵니다.

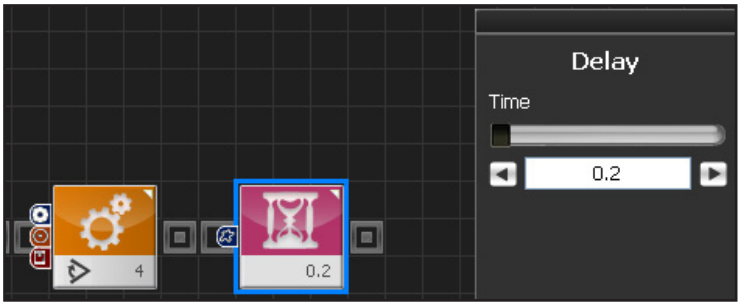
Motion 〉 Moter 를 선택합니다.
Mode : Position 으로 선택합니다.
Position : 235 로 설정합니다.
Motor ID : 1 로 설정합니다.
Time : 10 으로 설정합니다.



43 모터 4번(왼쪽 팔 아랫부분) 설정

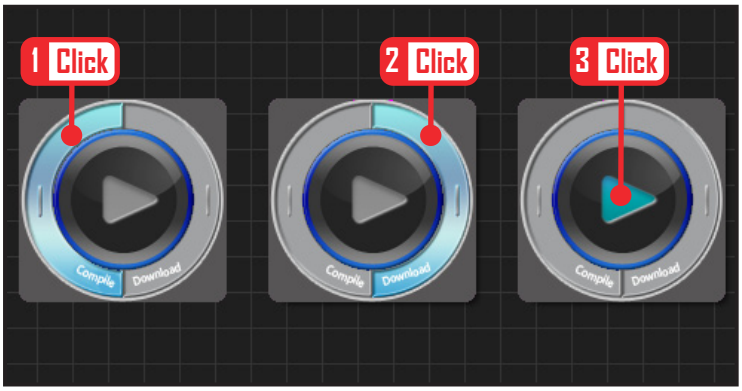
차려자세로 복귀시킵니다.

Motion 〉 Moter 를 선택합니다.
Mode : Position 으로 선택합니다.
Position : 235 로 설정합니다.
Motor ID : 4 로 설정합니다.
Time : 10 으로 설정합니다.



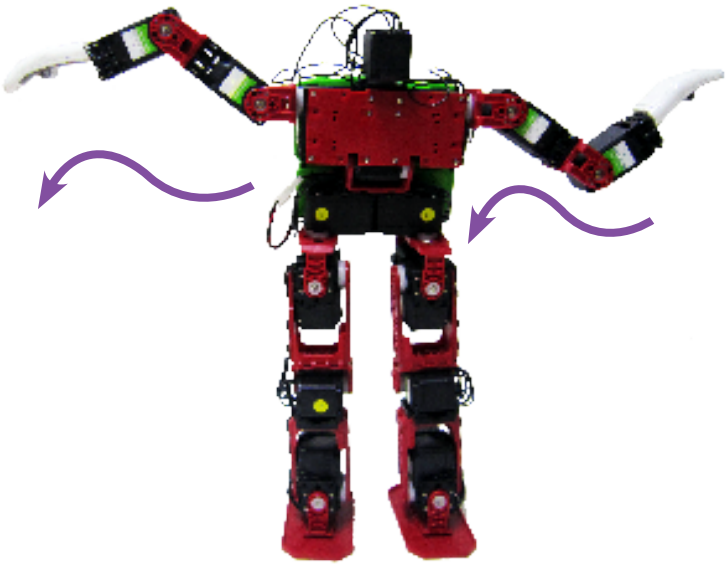
44 Delay 지연시키기

0.2 초 동안 기다립니다.



45 컴파일, 다운로드, 실행

왼쪽 클릭하여 컴파일 시킵니다. 에러가 없으면 오른쪽 클릭하여 로봇에 다운로드 시킵니다. 다운로드 완료되면 가운데 화살표 실행버튼을 눌러 로봇에서 실행시킵니다.



46 로봇동작

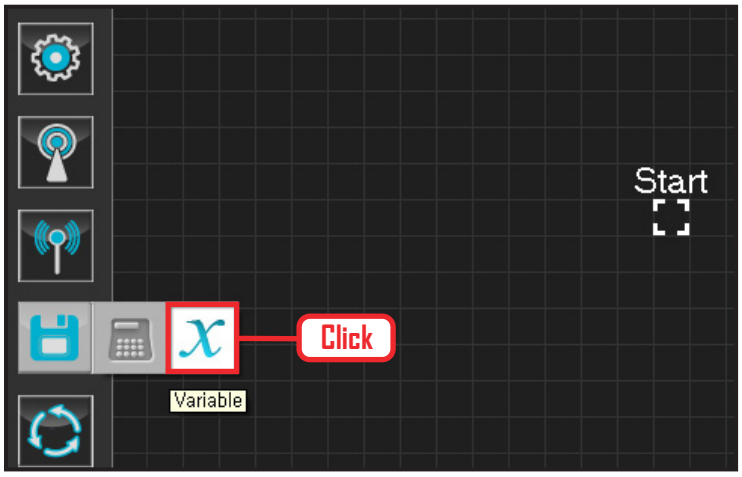
로봇이 왼팔부터 웨이브를 합니다.

Move 예제 따라하기

예제설명

Move 는 제어기 DRC에 저장된 로봇의 모션을 가져와 프로그램에 적용하는 것입니다. 로봇 모션은 번호로 가져올 수 있으며, 번호별 모션 이름은 DR-SIM 에서 확인할 수 있습니다. DR-SIM 에서 편집된 특정 모션을 로봇에서 무한반복적으로 돌리는 프로그래밍을 해봅니다. 로봇 신뢰성 테스트나 전시등에서 활용도가 높고, 난이도가 있는 프로그램이므로 천천히 따라해보기 바랍니다.

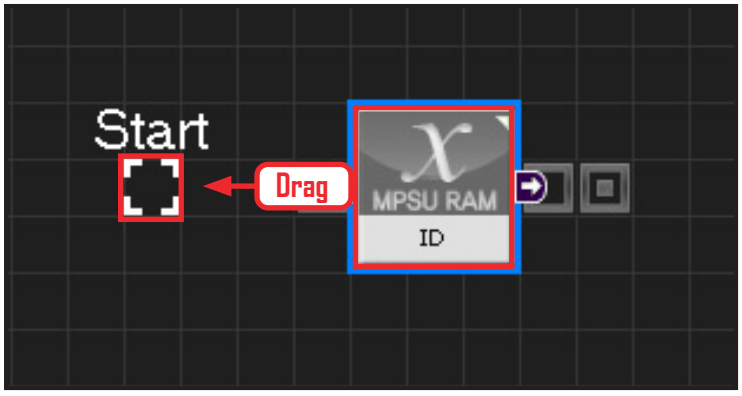
* 이 예제에서 사용하는 모션과 모션번호는 기본제공 모션과 다릅니다. DR-SIM 을 통해 편집한 모션을 DRC에 다운로드 한 경우를 가정하고 작성된 예제입니다.



01 변수 지정

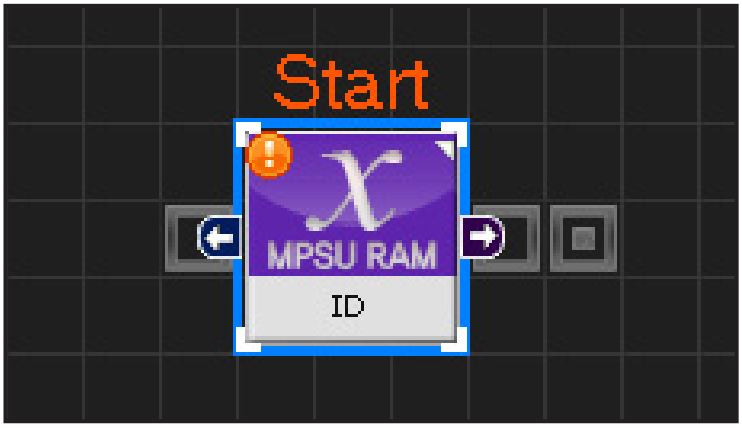
로봇을 동작시킨다는 것은 로봇의 서보 모터를 동작시킨다는 의미입니다. 서보가 스스로 움직일 수 있는 상태로 값을 지정해주어야 합니다.

Data 〉 Variable 모듈을 클릭합니다.



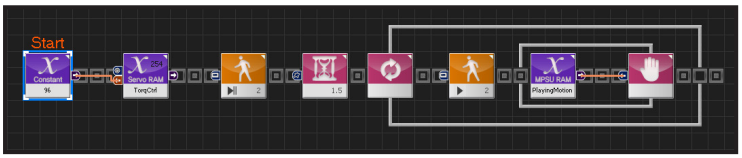
02 시작

모듈의 왼쪽 연결선을 Start Point 에 드래그하여 정확히 도킹을 시킵니다.



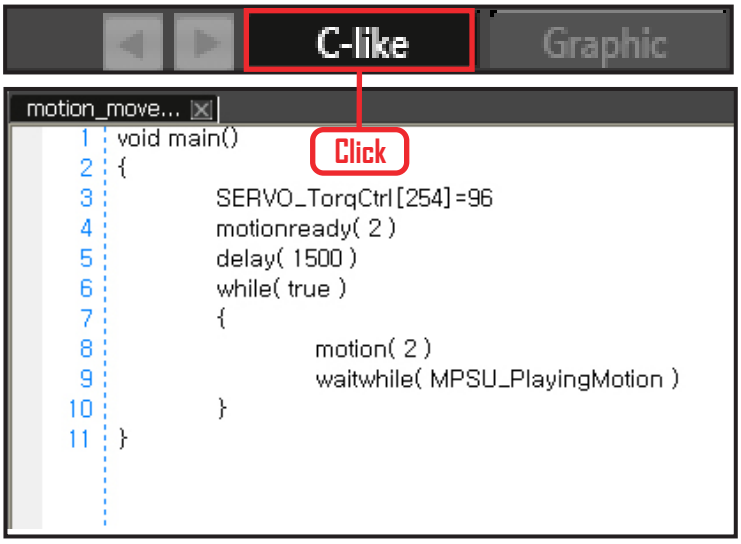
03 프로그래밍 시작

모듈과 Start Point 가 정확히 도킹하면 왼쪽과 같이 활성화된 칼라 이미지 모듈로 변합니다.
그럼 프로그래밍이 시작되었다는 의미입니다.



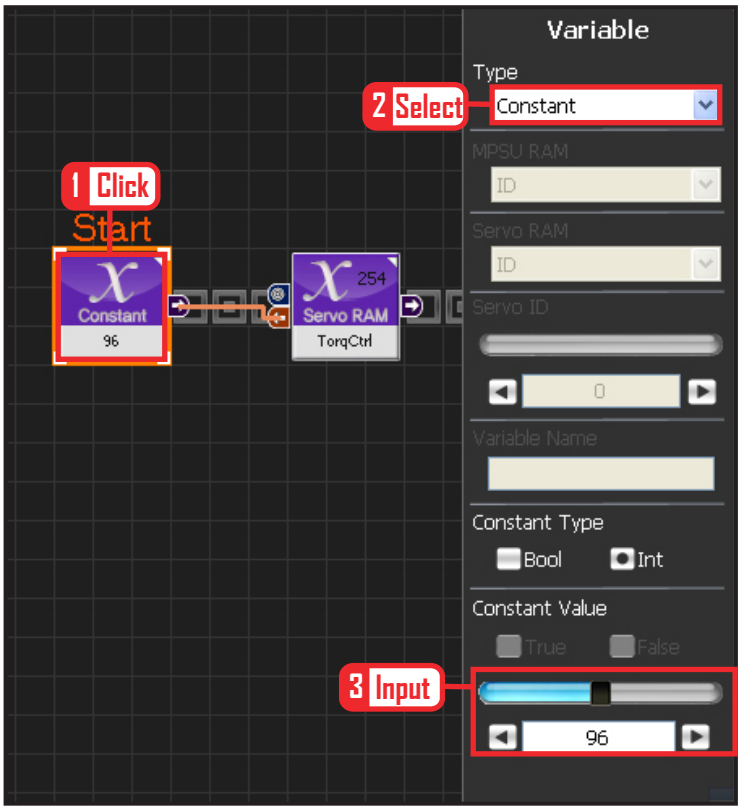
04 전체 프로그래밍

저장된 모션을 가져와 일정하게 반복 시키는 프로그래밍입니다. Motion ready 값에 주의합니다.



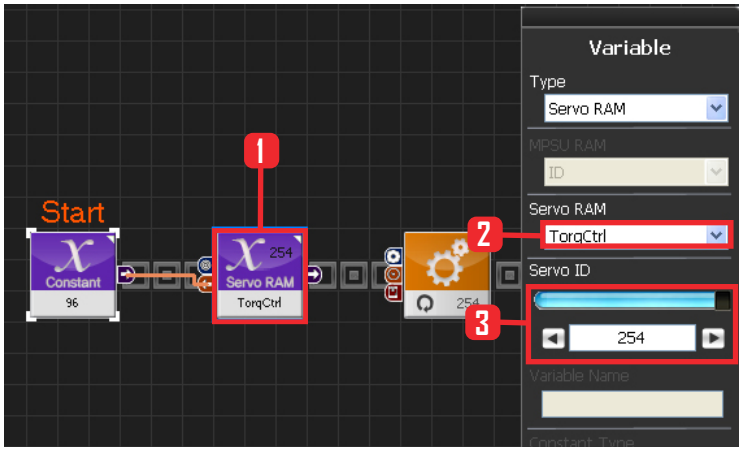
05 C-Like 보기

오른쪽 상단의 Graphic 탭에서 C-like 탭을 클릭하면 왼쪽과 같은 Task 프로그래밍 화면이 나옵니다.
저장모션을 이용한 전체 프로그래밍 화면입니다.
C와 유사한 문법 구조를 가지고 있으므로 C 문법 선행학습 효과도 있습니다.
각 모듈별로 클릭하면 커서가 따라서 움직이므로 모듈별로 Text로 어떻게 변환하는지 확인할 수 있습니다.



06 상수 설정

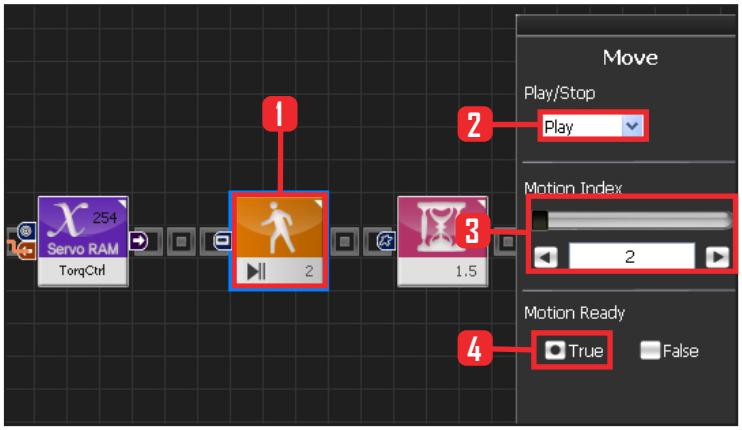
서보 모터를 스스로 움직일 수 있는 상태로 만드는 과정입니다.
Variable Type 을 Constant 로 선택합니다.
속성중에 Constant Value 값을 96 으로 설정합니다.
서보의 TorqControl 레지스터에 96(0x60) 이라는 값이 들어가면 서보가 움직일 수 있는 상태가 됩니다.
그 값은 Output 커넥터를 통하여 뒤 모듈의 토크값에 전달합니다.



07 모든 서보에 적용

앞에서 받은 96 상수값을 모든 서보에 적용하는 과정입니다.

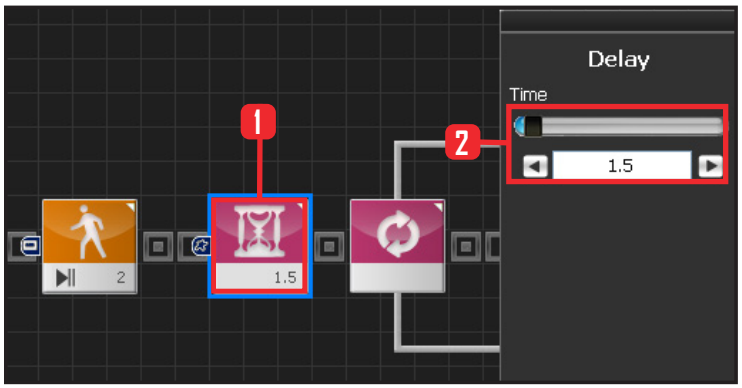
Variable 〉 Type : Servo RAM을 선택합니다.
Servo RAM : TorqCtrl 을 선택합니다.
Servo ID : 254 를 선택합니다. 254는 연결되어있는 모든 서보에 적용하겠다는 의미입니다.



08 Motion Ready

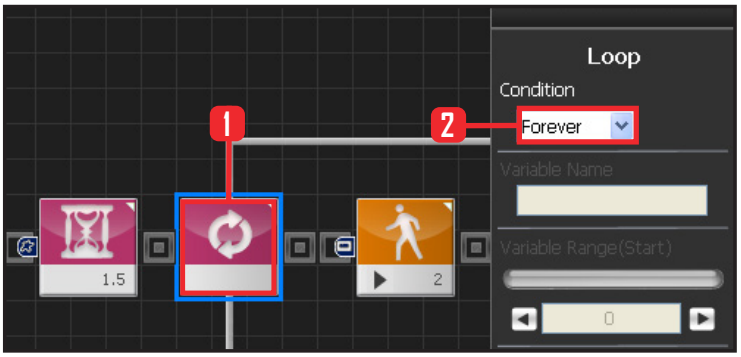
저장된 모션을 가져올 때 로봇의 현재 상태에서 갑작스럽게 모션이 변동하고 움직일 수가 있습니다. 현재 상태와 모션 시작상태가 너무 다르면 모터에 무리가 가거나 사용자에게 위험할 수도 있습니다. 그래서 Motion Ready 즉 모션이 동작할 준비시간을 줘야합니다.

Motion 〉 Move 를 선택합니다.
Play/Stop : Play 를 선택합니다.
Motion Index : 2를 선택합니다. 2번 모션을 가져오겠다는 뜻입니다. 참고로 이 프로그래밍의 2번 모션은 앉았다 일어나는 것입니다. 꼭 2번일 필요는 없으며, 사용자가 실행하고 싶은 모션 번호를 입력하면 됩니다.
Motion Ready : True 를 선택합니다. True 를 선택하면 동작하고자 하는 모션의 첫 번째 상태로 서서히 이동합니다.



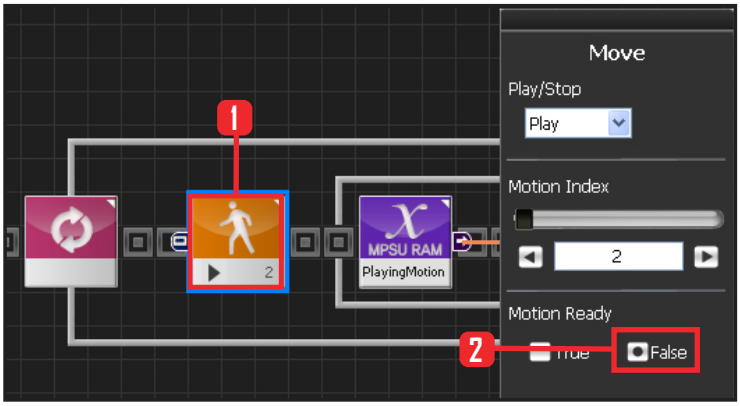
09 Delay

Motion Ready 동작이 끝나기 전에 진행하는 것을 방지하기위해 Delay 값을 1.5 초로 설정합니다.



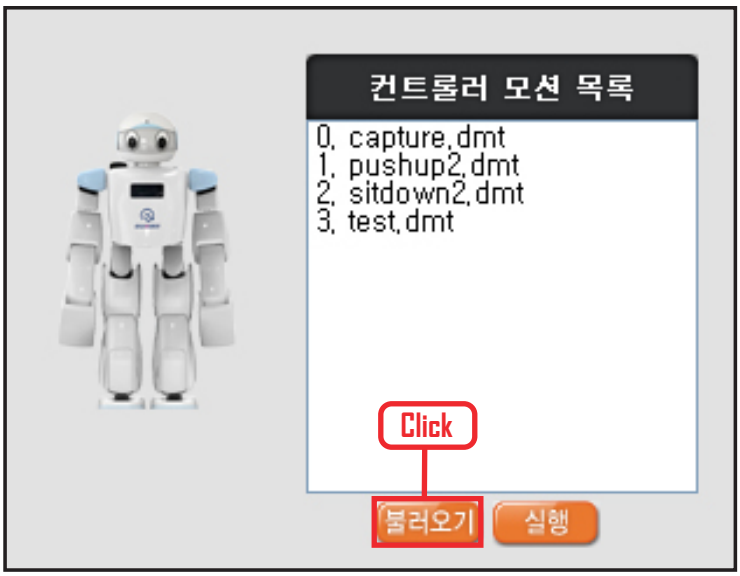
10 반복

Forever 무한 반복을 설정합니다.



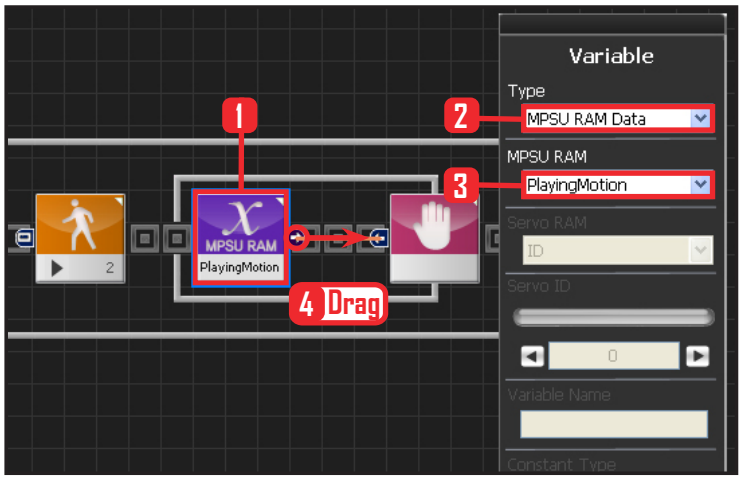
11 모션 동작

Motion Ready 값을 False 로 설정하면 모션을 처음부터 끝까지 동작시킨다는 의미입니다.
Motion Ready : False 로 선택합니다.



참조: 모션 보기

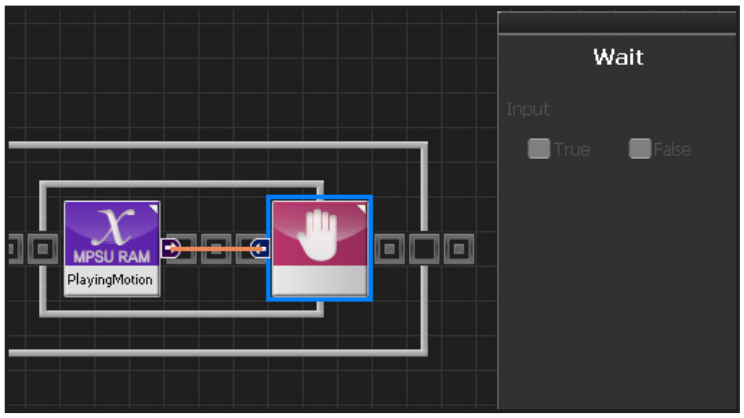
DR-SIM 프로그램에서 로봇과 연결하여 로봇설정을 클릭하면, 현재 제어기에 있는 로봇 모션을 확인할 수 있습니다.
현재 2번 모션은 양팔을 벌려 앉았다 일어나는 모션입니다.



12 모션 동작 확인

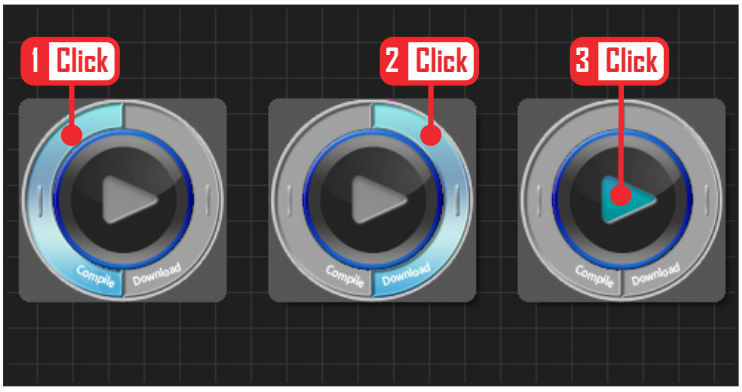
Loop 는 계속적인 반복을 의미합니다. Move 명령을 내리고 나서 실제 모션이 실행되어 완료되기 까지는 시간이 걸리므로 loop안에 Move모듈 하나만을 넣고 실행하면 모션을 이미 실행중임에도 loop를 계속 돌면서 모션실행 명령을 반복하게 됩니다.
이렇게 되면 Move모듈을 만난 횟수와 실제모션을 실행한 횟수가 달라집니다.
따라서 실행한 모션이 끝날때까지 기다렸다가 다시 loop의 처음으로 돌아가게 하는 편이 더 정확합니다.
Variable 〉 MPSU RAM Data 에 들어가면 Playing Motion 이 있습니다. Playing Motion 은 로봇이 모션을 실행중인지 확인하는 변수입니다. 그 Playing Motion에 Wait 를 걸어주면 로봇의 동작이 끝날 때까지 Loop는 기다려줍니다.

Data 〉 Variable 모듈을 선택합니다.
Type : MPSU RAM Data 를 선택합니다.
MPSU RAM : Playing Motion 을 선택합니다.
Output 커넥터 값을 뒤에 Wait 모듈에 연결합니다.
모션이 끝날 때까지 기다리겠다는 의미입니다.



13 Wait

모션이 끝날때까지 기다립니다.
모션이 끝나면 다시 처음으로 돌아가 모션을 반복합니다.



14 컴파일, 다운로드, 실행

왼쪽 클릭하여 컴파일 시킵니다. 에러가 없으면 오른쪽 클릭하여 로봇에 다운로드 시킵니다. 다운로드 완료되면 가운데 화살표 실행버튼을 눌러 로봇에서 실행시킵니다.

1

2

15 로봇동작

로봇이 앉았다 일어나는 모션을 반복적으로 수행합니다.

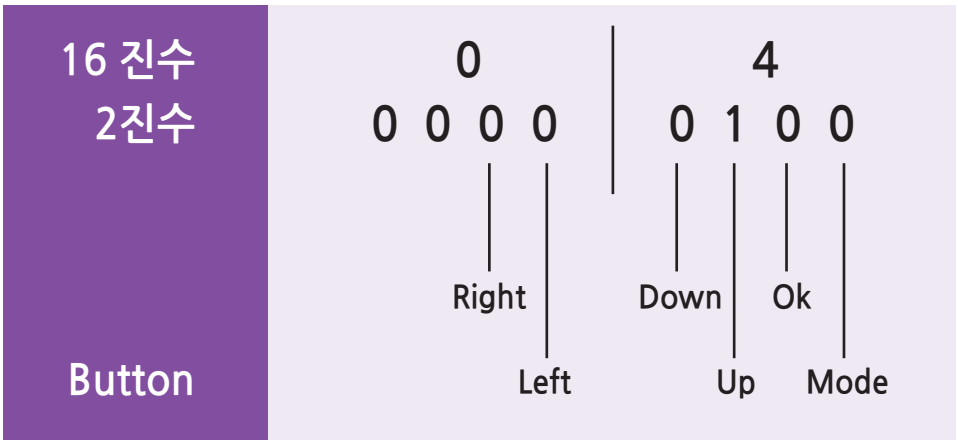
Button, LED 예제 따라하기

예제설명

DRC 제어기에 있는 Button 을 이용하여 LED 를 켰다 끄는 프로그래밍을 합니다.

Button 과 LED 프로그래밍을 하기위해선 Button 과 LED 가 켜지는 과정을 먼저 알아야 합니다.

$16 = 2^4$



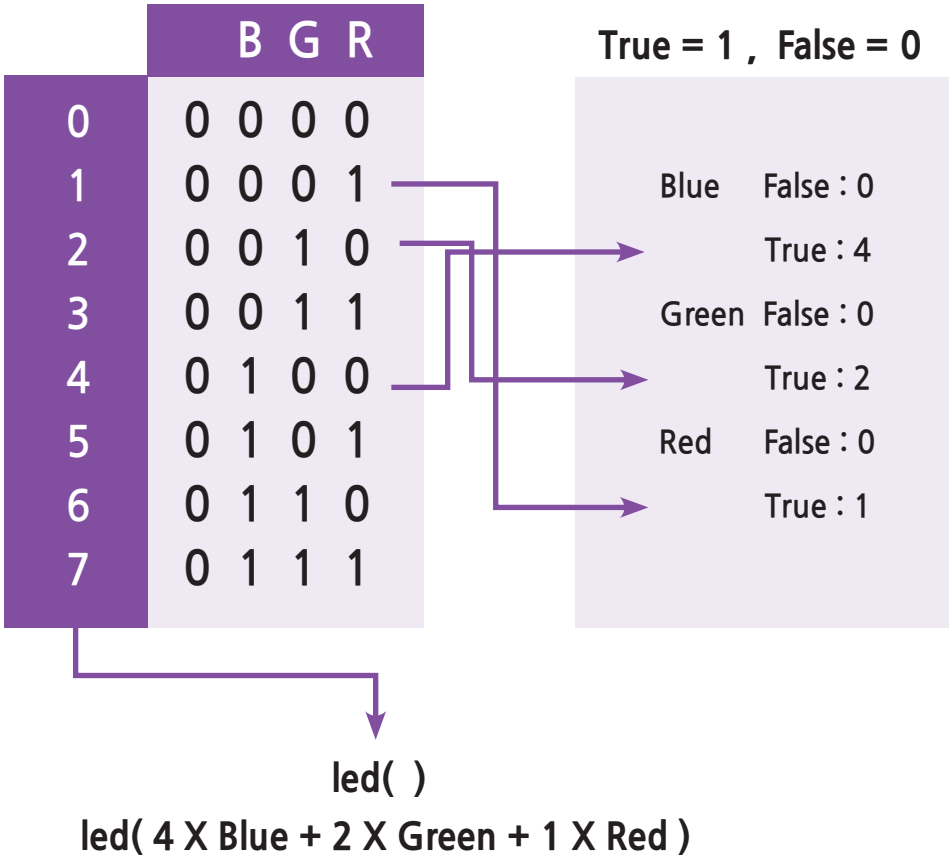
EX)

Right	2	0	h
	0 0 1 0	0 0 0 0	
Down	0	8	h
	0 0 0 0	1 0 0 0	

Button

DRC에는 6개의 버튼이 있으며, 버튼의 눌린 상태를 1Byte로 표시합니다. 1Byte는 8Bit로 이루어져 있으므로, 1Byte에는 8개의 1과 0을 저장할 수 있습니다. DRC의 버튼 6개가 눌린 상태(1)과 눌리지 않은 상태(0)을 표시하기 위해서는 6Bit가 필요합니다. 위 그림에서 보는 바와 같이, 각 버튼은 Bit 하나로 매칭되어 있습니다.

버튼이 눌린 상태는 1과 0으로 표시되며, 이는 Button 모듈 우측 하단에 16진수로 표시됩니다. Right 버튼이 눌린 경우 버튼 값은 00100000이 되며, 이를 16진수로 바꾸면 20h입니다(h는 16진수라는 표시입니다). Down 버튼이 눌린 경우 버튼 값은 00001000이 되며, 이를 16진수로 바꾸면 08h입니다. 조금 더 복잡하게, Up+Down 버튼이 눌린 경우에 버튼 값은 00001100이 되며, 이를 16진수로 바꾸면 0Ch가 됩니다.



LED

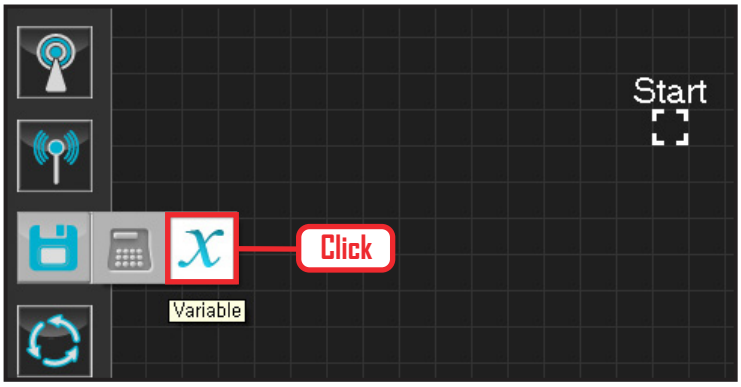
DRC에는 7개의 LED가 있지만, Task 모드에서 제어 가능한 것은 3개입니다. Red, Green, Blue의 LED 3개가 켜진 상태와 꺼진 상태를 표시하기 위해서는 3Bit가 필요합니다. 위 그림에서 보는 바와 같이, 각 LED는 바이트의 최하위 비트부터 Red, Green, Blue의 순서로 매칭되어 있습니다.

이 LED 값을 LED 모듈의 입력 값으로 넣으면, 해당하는 LED가 켜지게 됩니다. 가령 LED 모듈의 입력 값에 2를 넣었다면, 2는 2진수로 표시하면 00000010이므로 Green LED가 켜지게 됩니다. 비슷한 원리로 입력 값 0(00000000)은 LED가 다 꺼지게 만들며, 입력 값 7(00000111)은 LED가 켜지게 만듭니다.

Blue는 2진수에서 4의 자리에 해당하며, Green은 2의 자리, Red는 1의 자리에 해당합니다. 만약 Blue, Green, Red라는 변수가 있어서 이들의 값(true, false)에 따라 각 LED의 켜짐/꺼짐 여부가 결정된다면, LED 모듈의 입력 값으로 4 x Blue + 2 x Green + 1 x Red을 넣으면 변수 이름에 따라서 실제 LED를 제어할 수 있을 것입니다.

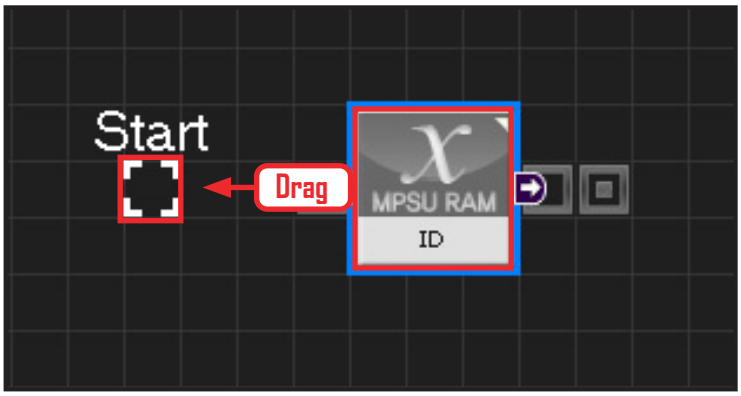
예를 들어서 Blue, Green이 true이고 Red가 false일 때 4 x Blue + 2 x Green + 1 x Red = 6이 됩니다. 6은 2진수로 표시하면 00000110이므로 이것을 LED 모듈의 입력 값으로 넣으면 Green, Blue LED가 켜지는 것입니다.

위 배경지식을 기반으로 Button 과 LED 프로그래밍을 해봅니다



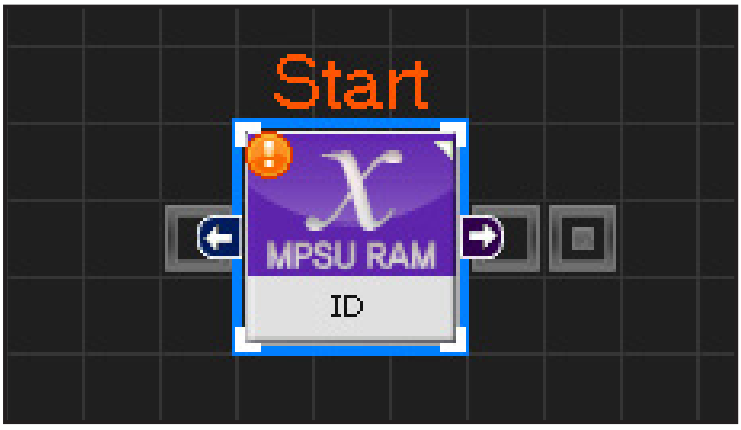
01 변수 지정

Data 〉 Variable 모듈을 클릭합니다.



02 시작

모듈의 왼쪽 연결선을 Start Point 에 드래그하여 정확히 도킹을 시킵니다.

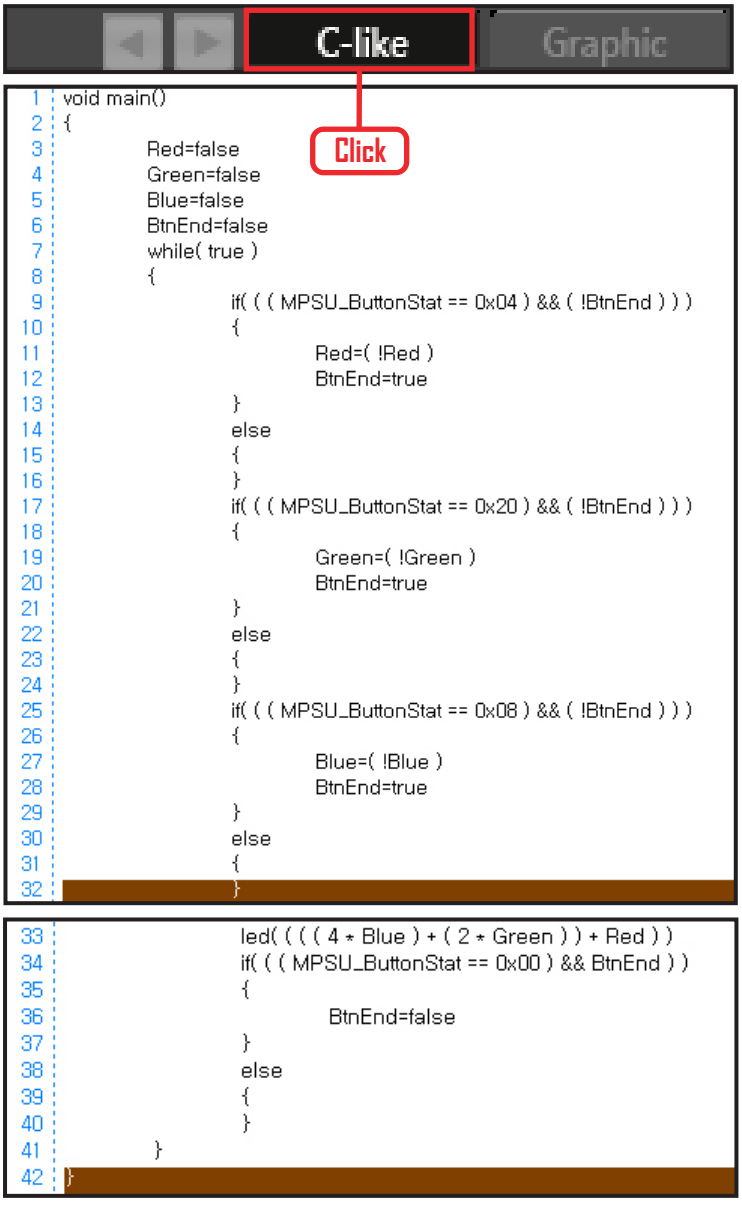


03 프로그래밍 시작

모듈과 Start Point 가 정확히 도킹하면 왼쪽과 같이 활성화된 칼라 이미지 모듈로 변합니다.
그럼 프로그래밍이 시작되었다는 의미입니다.

04 전체 프로그래밍

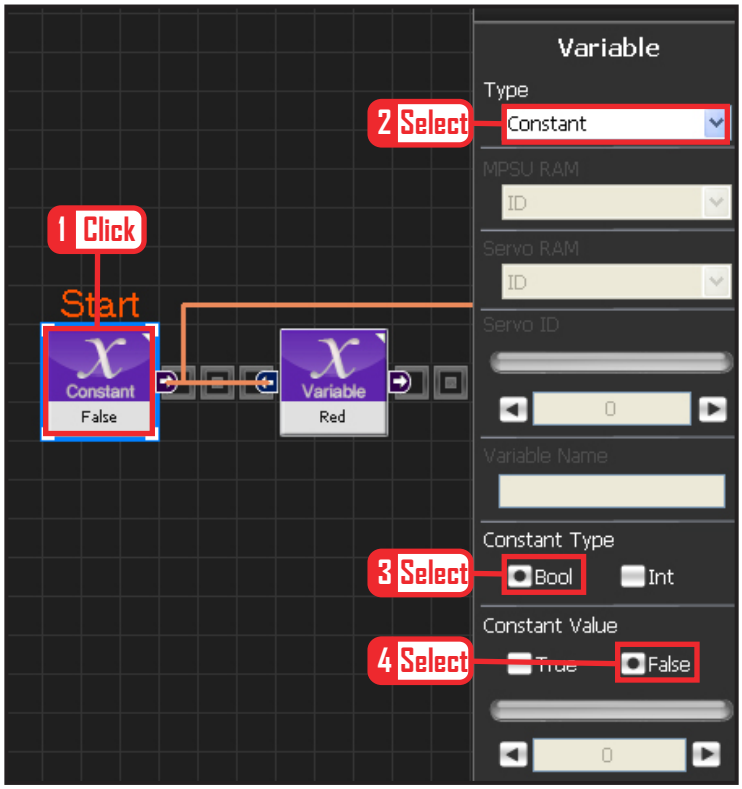
버튼과 LED를 이용한 전체 프로그래밍입니다.




```
1  void main()
2  {
3          Red=false
4          Green=false
5          Blue=false
6          BtnEnd=false
7          while( true )
8          {
9                  if( ( ( MPSU_ButtonStat == 0x04 ) && ( !BtnEnd ) ) )
10                 {
11                         Red=( !Red )
12                         BtnEnd=true
13                 }
14                 else
15                 {
16                 }
17                 if( ( ( MPSU_ButtonStat == 0x20 ) && ( !BtnEnd ) ) )
18                 {
19                         Green=( !Green )
20                         BtnEnd=true
21                 }
22                 else
23                 {
24                 }
25                 if( ( ( MPSU_ButtonStat == 0x08 ) && ( !BtnEnd ) ) )
26                 {
27                         Blue=( !Blue )
28                         BtnEnd=true
29                 }
30                 else
31                 {
32                 }
33                 led( ( ( ( 4 * Blue ) + ( 2 * Green ) ) + Red ) )
34                 if( ( ( MPSU_ButtonStat == 0x00 ) && BtnEnd ) )
35                 {
36                         BtnEnd=false
37                 }
38                 else
39                 {
40                 }
41         }
42 }
```

05 C-Like 보기

오른쪽 상단의 Graphic 탭에서 C-like 탭을 클릭하면 왼쪽과 같은 Task 프로그래밍 화면이 나옵니다.
Button과 LED를 이용한 전체 프로그래밍 화면입니다.
C와 유사한 문법 구조를 가지고 있으므로 C 문법 선행학습 효과도 있습니다.
각 모듈별로 클릭하면 커서가 따라서 움직이므로 모듈별로 Text로 어떻게 변환하는지 확인할 수 있습니다.

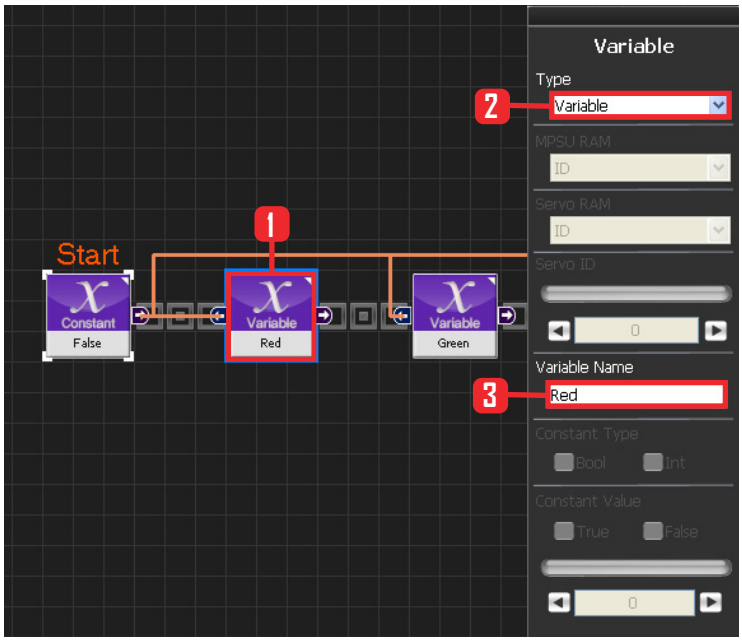


06 False 로 초기화

모든 LED 는 False 꺼진 상태로 초기화합니다.

Data 〉 Variable 모듈을 선택합니다.
Type : Contant 를 선택합니다.
Constant Type 은 Bool 로 설정합니다. Bool 은 참과 거짓을 나타내는 자료형입니다.
Constant Value : False 를 선택합니다

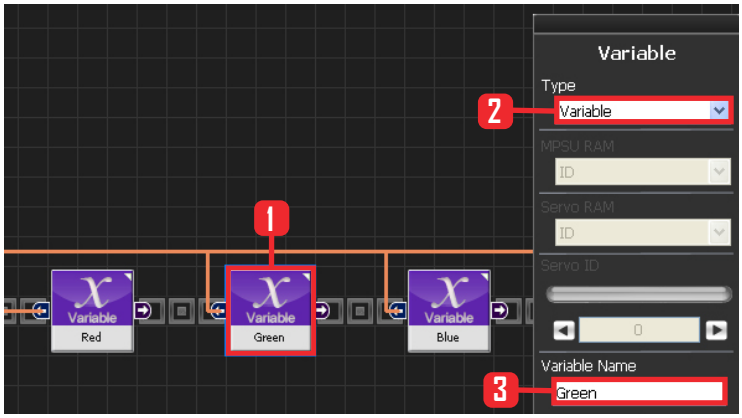
이후 커넥터를 이용하여 False를 각 변수에 연결합니다.



07 Red 변수

Data 〉 Variable 을 선택합니다.
Type : Variable 로 선택합니다.
Variable Name : Red 로 입력합니다.

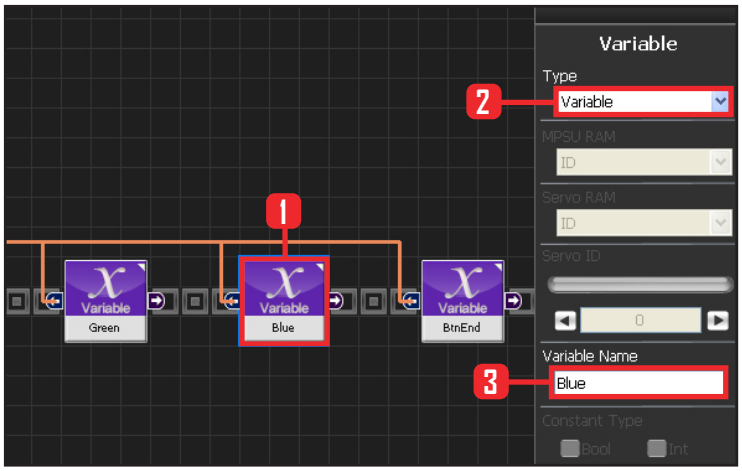
False 면 빨간색 LED가 꺼지고, True 면 켜집니다.



08 Green 변수

Data 〉 Variable 을 선택합니다.
Type : Variable 로 선택합니다.
Variable Name : Green 으로 입력합니다.

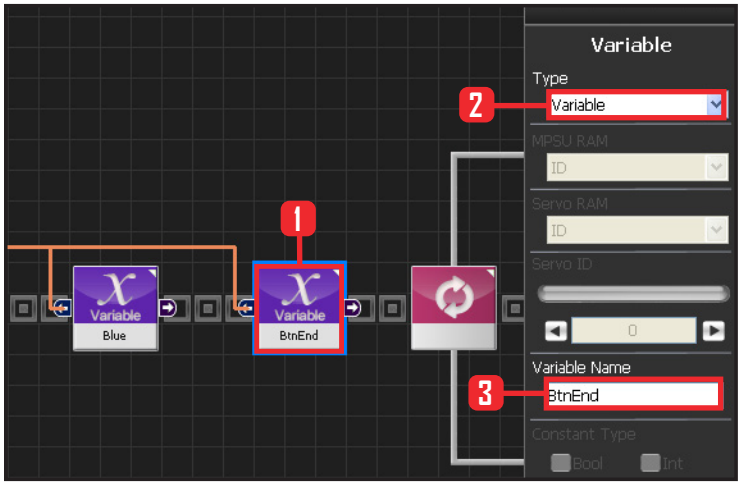
False 면 초록색 LED가 꺼지고, True 면 켜집니다.



09 Blue 변수

Data 〉 Variable 을 선택합니다.
Type : Variable 로 선택합니다.
Variable Name : Blue 로 입력합니다.

False 면 파란색 LED가 꺼지고, True 면 켜집니다.



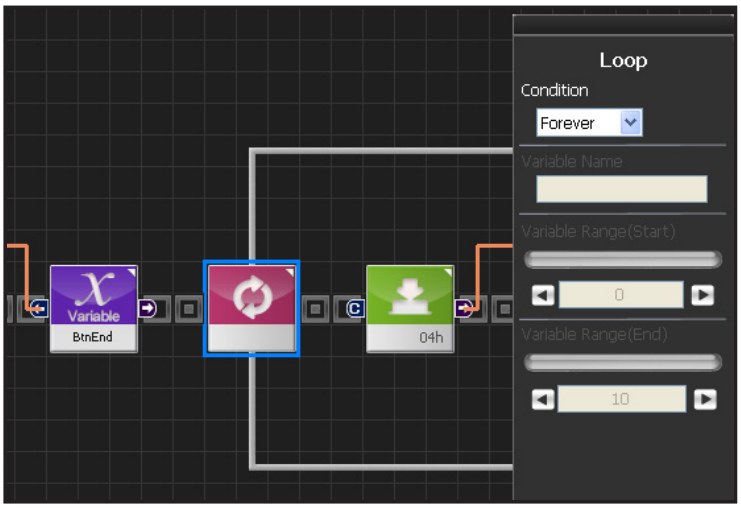
10 BtnEnd 변수

버튼이 안 눌렸을 때 False로 유지되다가
버튼이 눌리고 원하는 동작이 끝나는 순간
False –〉 True로 바뀌는 변수입니다.
또한 버튼을 떼는 순간 True 에서 False로 바뀌는 변수입니다.

Data 〉 Variable 을 선택합니다.
Type : Variable 로 선택합니다.
Variable Name : BtnEnd 로 입력합니다.

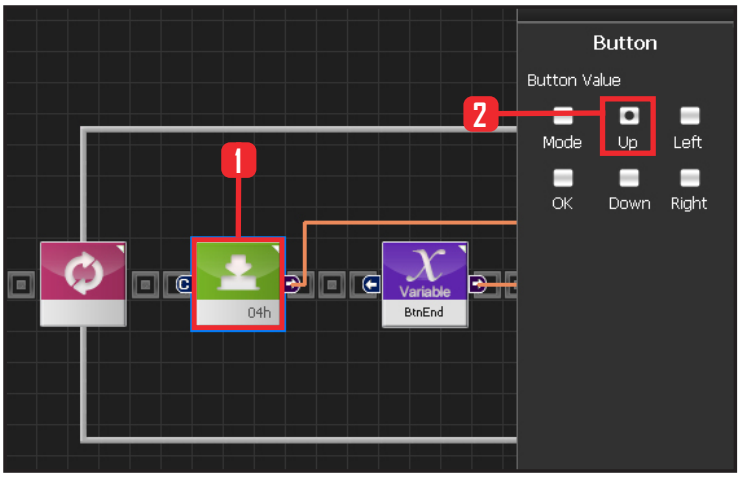
11 변수 선언

Red, Green, Blue, BtnEnd 4개 모두 초기값을 False 로 설정합니다.



12 Loop

Forever 무한 반복시킵니다.

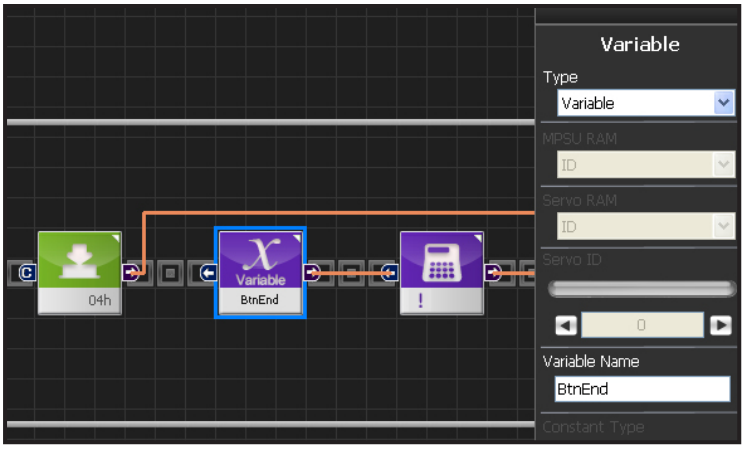


13 Up 버튼

버튼모듈을 만듭니다. 이모듈은 선택한 버튼이 눌렸을 경우 True가 되고 그 외에는 False가 됩니다. Up Button 을 선택시, Up Button이 눌린경우 true, 그외에 false 가 됩니다.

Communication 〉 Button 모듈을 선택합니다.
Button Value 를 Up 으로 선택합니다.

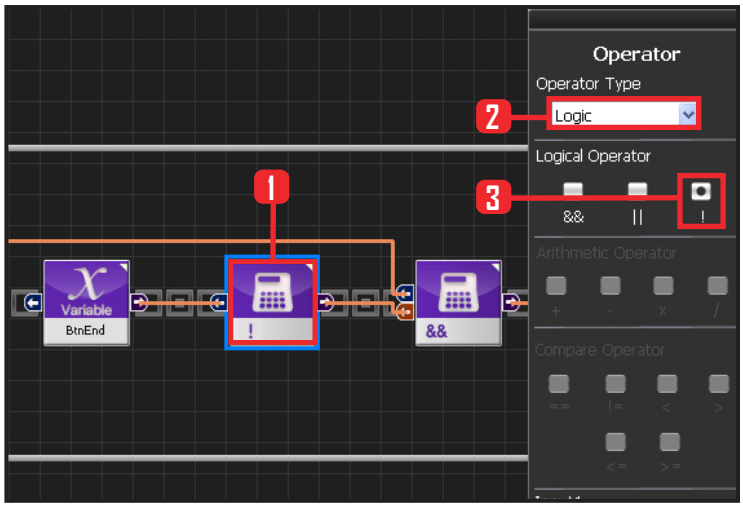
16진수로 04h 이므로 모듈 그림에 04h가 표기됩니다.



14 BtnEnd

BtnEnd 값은 false 로 초기화 되어있습니다. 뒤에 not 연산이 붙어서 True 로 변합니다.

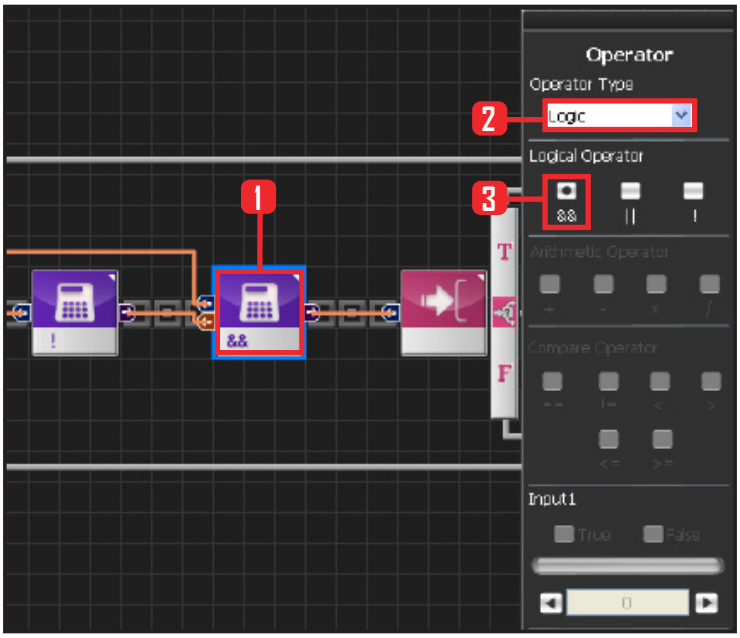
앞의 btnEnd 변수를 복사하여 붙입니다.



15 ! 연산

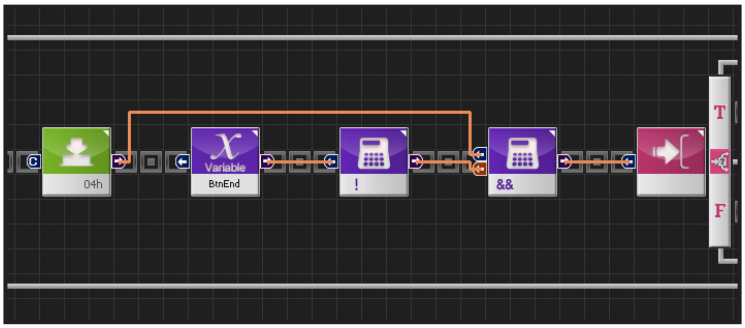
! 연산을 적용하여 BtnEnd 값을 반대로 바꿉니다.

Data 〉 Operator 모듈을 선택합니다.
Operartor Type : Logic으로 선택합니다.
Logical Operator : ! 로 선택합니다.



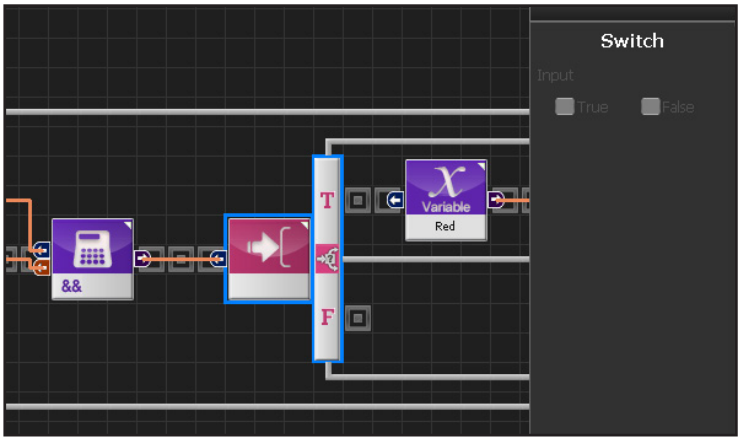
16 And 연산

Up 버튼이 눌러졌고, BtnEnd가 false (! 적용하여 True) 일때 True가 되어, 뒤의 조건문을 실행하도록 합니다.
Data 〉 Operator 모듈을 선택합니다.
Operartor Type : Logic으로 선택합니다.
Logical Operator : && 로 선택합니다.



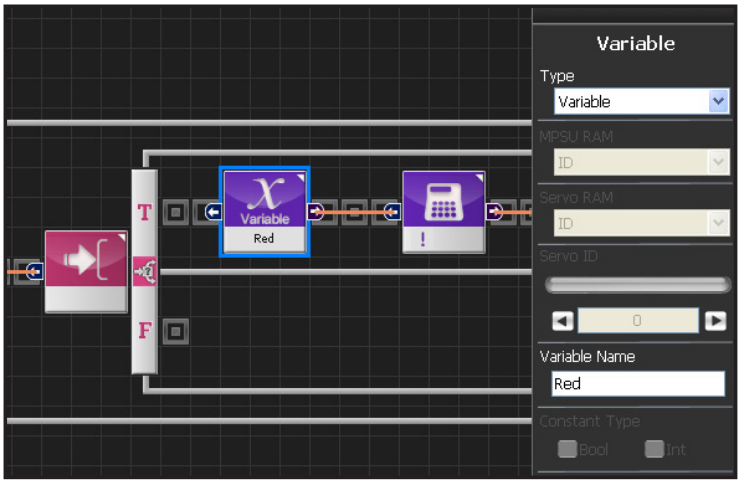
17 Up 버튼이 눌렸을 때

Up 버튼이 눌려졌고 BtnEnd가 false일때, 뒤 조건을 수행합니다.



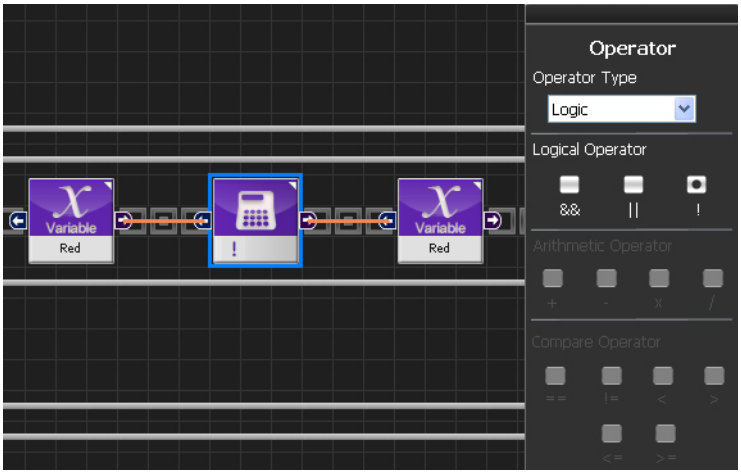
18 If 분기문

True 일 때 윗 부분을 수행합니다.



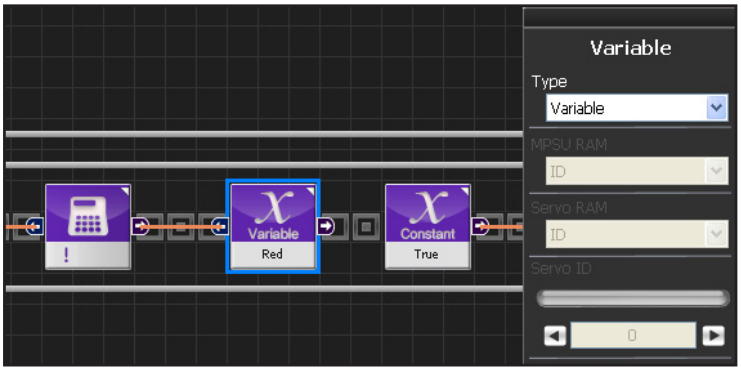
19 Red 출력

앞의 Red변수를 복사하여 붙입니다.



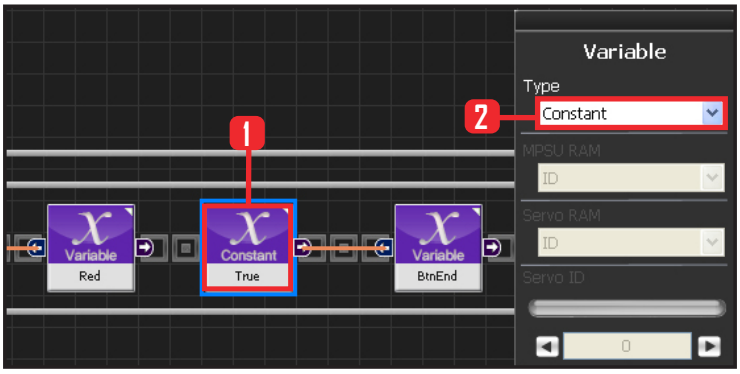
20 ! 연산

Red 가 True 이면 False 로 False 이면 True 로 변합니다.



21 Red 입력

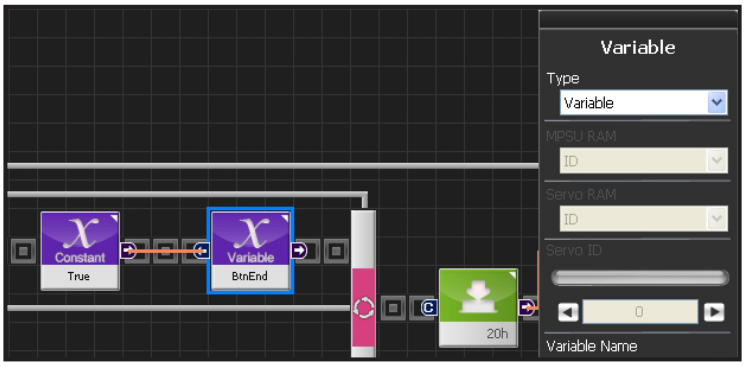
앞의 Red변수의 값이 true 일땐 false로, false일땐 true로 바뀌어 저장됩니다.



22 True 설정

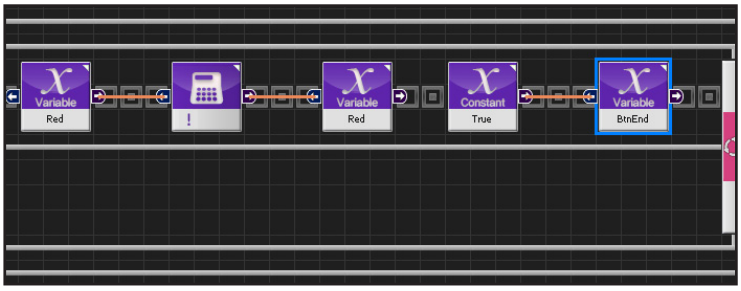
버튼이 눌리고 해야할 동작이 끝났으므로
BtnEnd가 false 에서 True로 바뀌어야 합니다.

Data 〉 Variable 모듈을 선택합니다.
Type : Contant 를 선택합니다.
Constant Type 은 Bool 로 설정합니다. Bool 은 참과 거짓을 나타내는 자료형입니다.
Constant Value : True 를 선택합니다



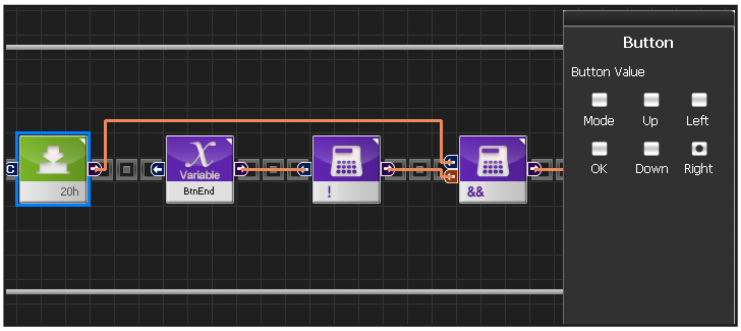
23 BtnEnd를 true로

True 값이 BtnEnd 에 입력됩니다.
BtnEnd가 true가 되면, loop를 반복할 때 up버튼이 계속 눌려 있더라도 조건문을 만족시키지 않으므로 Red변수의 값이 더이상 변하지 않게 됩니다.



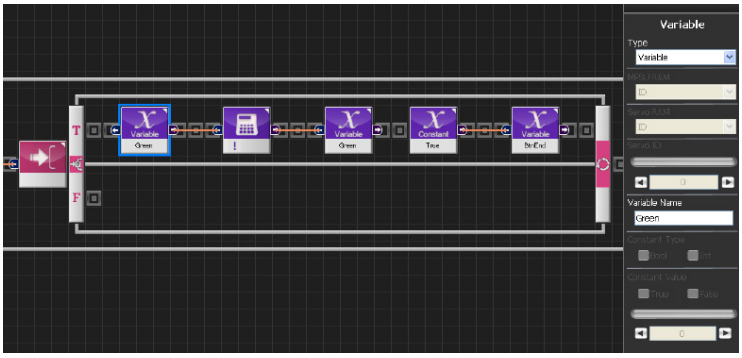
24 Red LED

Up버튼을 한번 누르면 Red가 켜지고, 다시 한번 누르면 Red 가 꺼집니다.



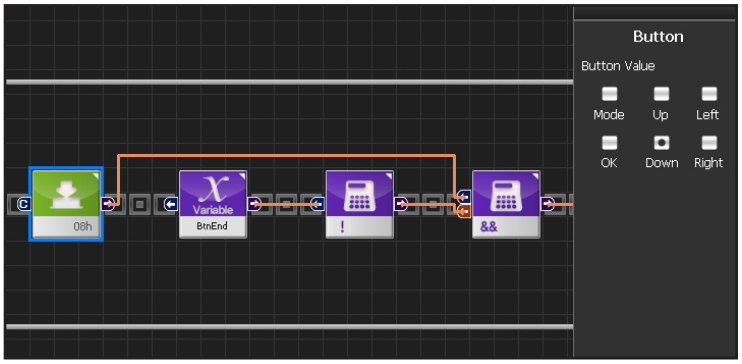
25 Right 버튼

Right 버튼을 눌렀을 때입니다.



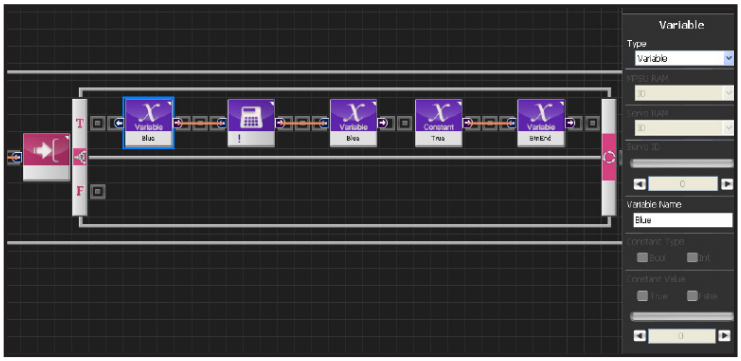
26 Green LED

Right 버튼을 한번 누르면 Green 이 켜졌다가, 한번 더 누르면 Green 이 꺼집니다.



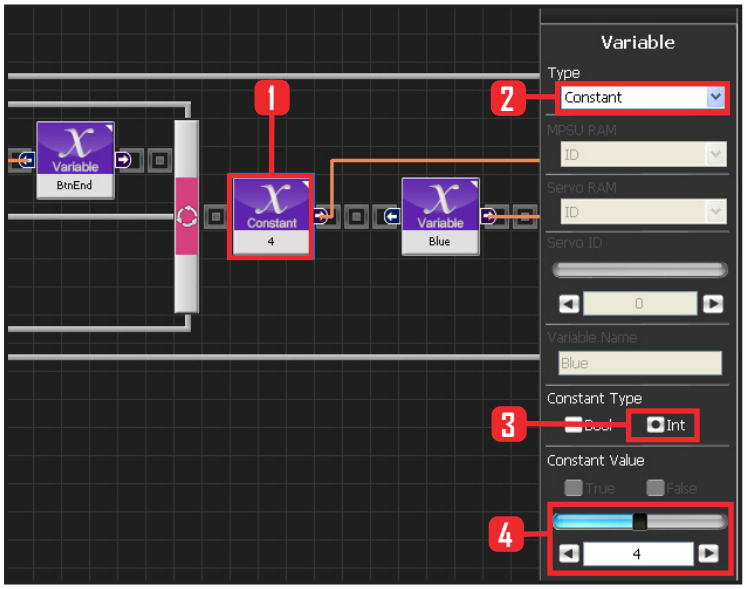
27 Down 버튼

Down 버튼을 눌렀을 때입니다.



28 Blue LED

Down 버튼을 한번 누르면 Blue 가 켜졌다가, 한번 더 누르면 Blue 가 꺼집니다.



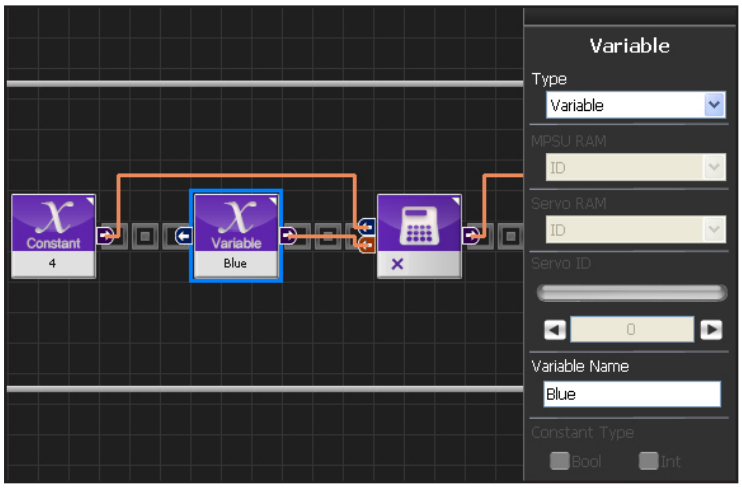
29 LED 값

위에서 설명했듯이 LED 모듈에 입력값이 들어가면, 그 값에 따라 LED가 켜집니다. 그 식은 (4 x Blue + 2 x Green + 1 x Red) 입니다.
위 식을 모듈별로 연결하여 표현한 것이 좌측 그림입니다.

상수 4 를 설정합니다.

(4 x Blue + 2 x Green + 1 x Red)

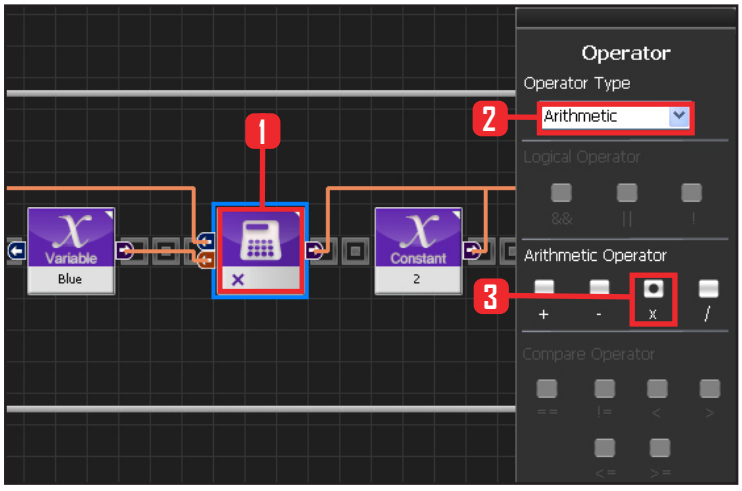
Data 〉 Variable 모듈을 선택합니다.
Type : Contant 를 선택합니다.
Constant Type 은 int 로 설정합니다.
Constant Value : 4로 설정합니다.



30 Blue

(4 x Blue + 2 x Green + 1 x Red)

앞의 Blue 변수를 복사합니다.

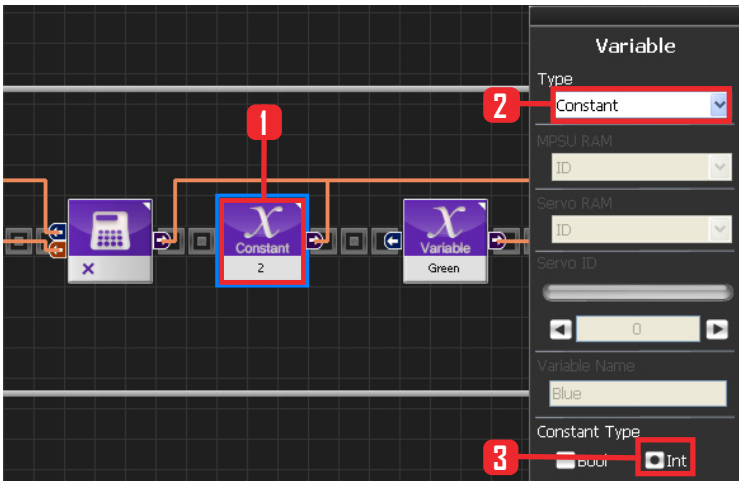


31 곱하기

(4 x Blue + 2 x Green + 1 x Red)

Data 〉 Operator 모듈을 선택합니다.
Operartor Type : Arithmetic으로 선택합니다.
Arithmetic Operator : X 로 선택합니다.

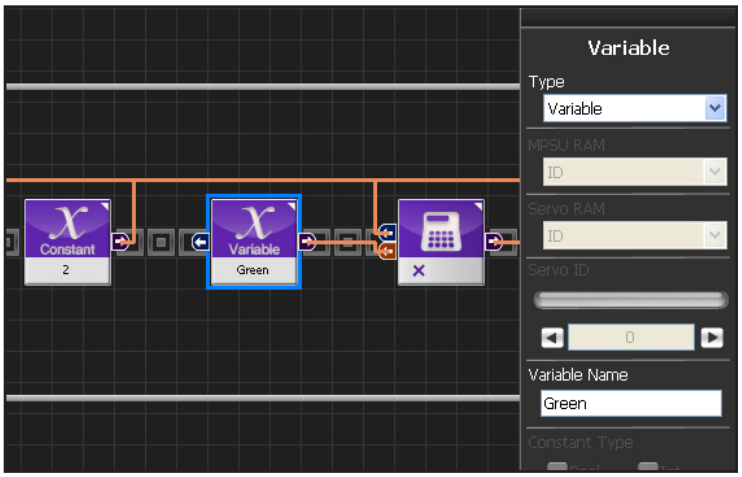
곱하기 모듈의 두 입력 커넥터에 상수 4와 Blue 변수 모듈을 이어 붙입니다.



32 상수2

(4 x Blue + 2 x Green + 1 x Red)

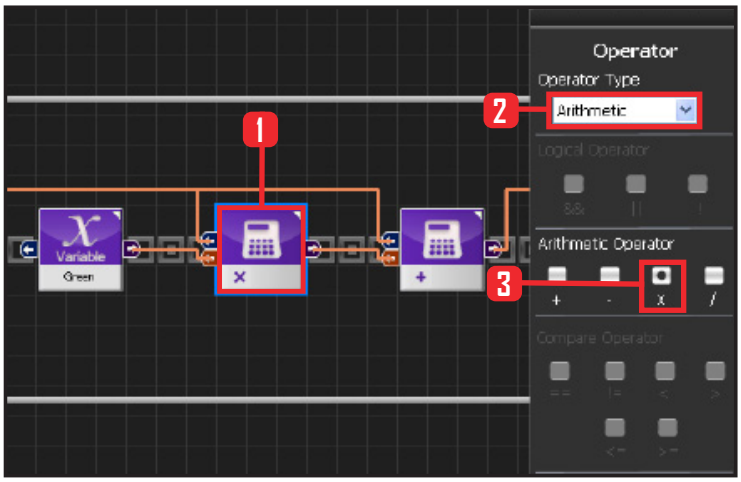
Data 〉 Variable 모듈을 선택합니다.
Type : Contant 를 선택합니다.
Constant Type 은 int 로 설정합니다.
Constant Value : 2로 설정합니다.



33 Green

(4 x Blue + 2 x Green + 1 x Red)

앞의 Green 변수를 복사합니다.

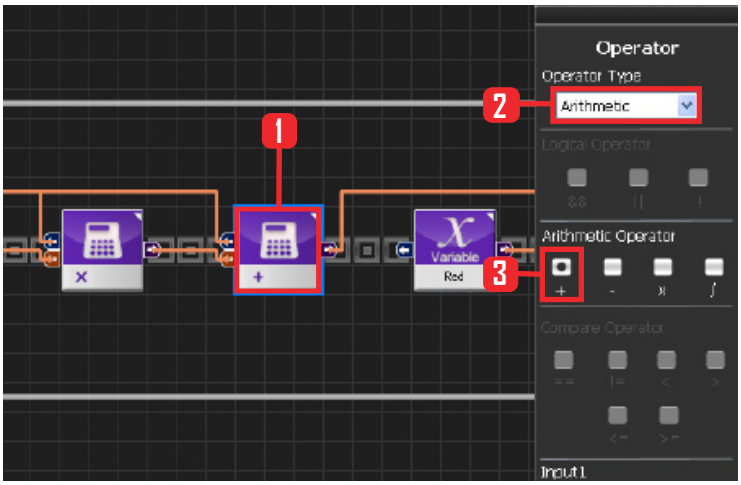


34 곱하기

(4 x Blue + 2 x Green + 1 x Red)

Data 〉 Operator 모듈을 선택합니다.
Operartor Type : Arithmetic으로 선택합니다.
Arithmetic Operator : X 로 선택합니다.

곱하기 모듈의 두 입력 커넥터에 상수 2와 Green 변수 모듈을 이어 붙입니다.

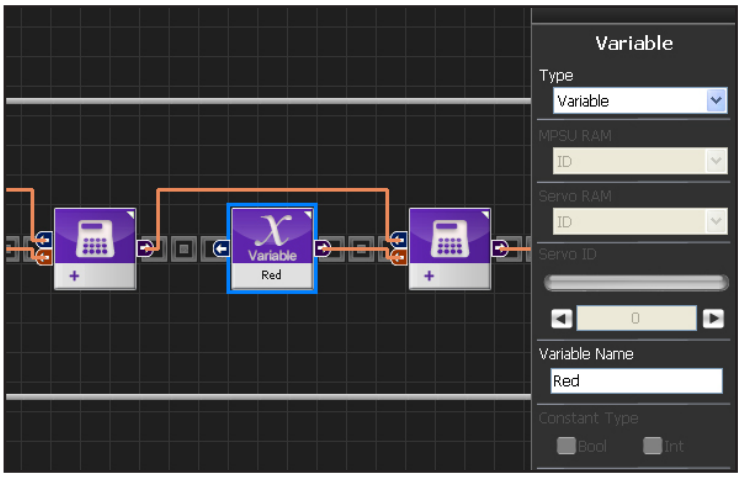


35 더하기

(4 x Blue + 2 x Green + 1 x Red)

Data 〉 Operator 모듈을 선택합니다.
Operartor Type : Arithmetic으로 선택합니다.
Arithmetic Operator : + 로 선택합니다.

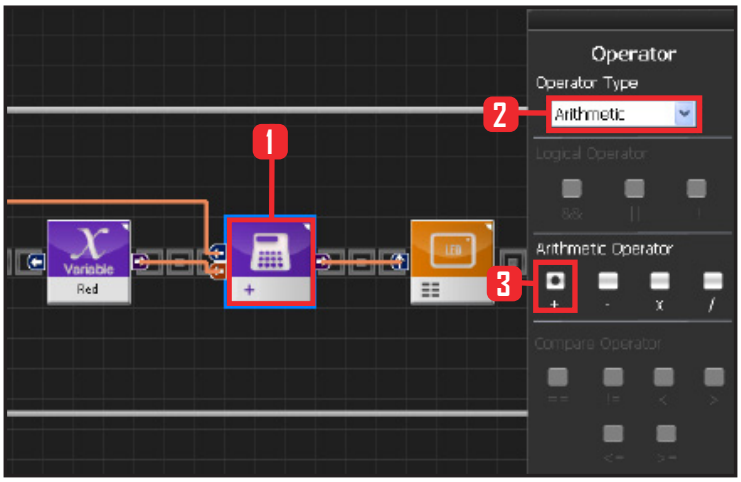
앞서 31, 34번에서 나온 두 곱하기 모듈의 출력을 더하기 모듈의 두 입력 커넥터에 연결합니다.



36 Red

(4 x Blue + 2 x Green + 1 x Red)

앞의 Red 변수를 복사해 옵니다.

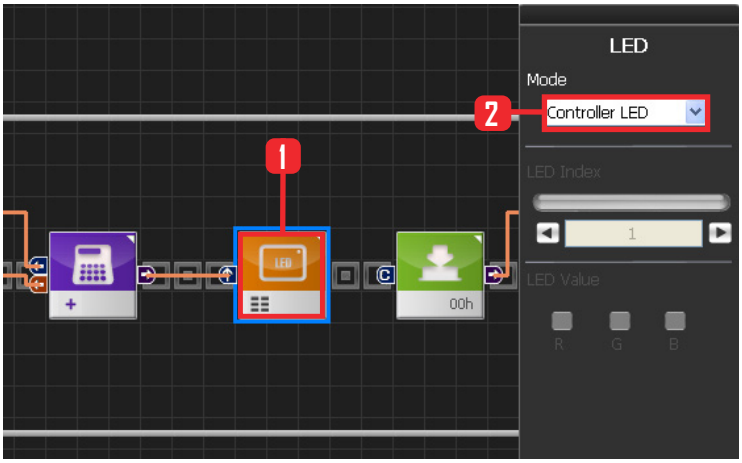


37 더하기

(4 x Blue + 2 x Green + 1 x Red)

Data 〉 Operator 모듈을 선택합니다.
Operartor Type : Arithmetic으로 선택합니다.
Arithmetic Operator : + 로 선택합니다.

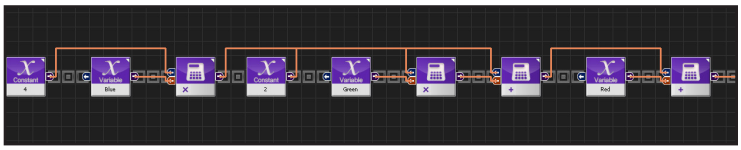
35번의 더하기 모듈과 Red 변수 모듈의 출력을 더하기 모듈의 두 입력 커넥터에 연결합니다.



38 LED

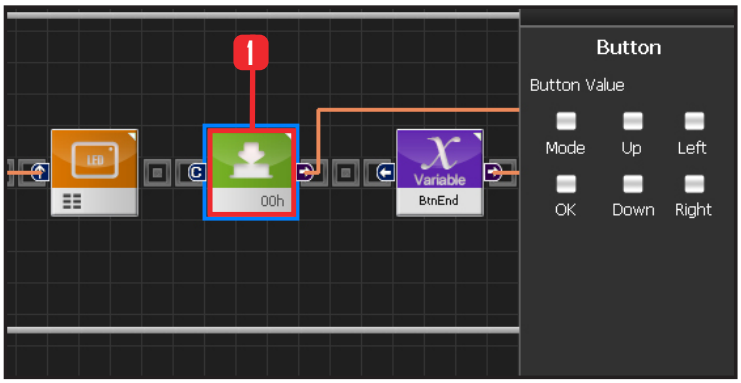
Motion 〉LED 모듈을 선택합니다.
Mode : Controller LED 를 선택합니다.

앞에서 연산한 값을 LED Value에 넣어서 각각의 LED 가 켜지게 합니다.



39 LED 값 산출 연산

(4 x Blue + 2 x Green + 1 x Red)
을 모듈로 표현한 것입니다.

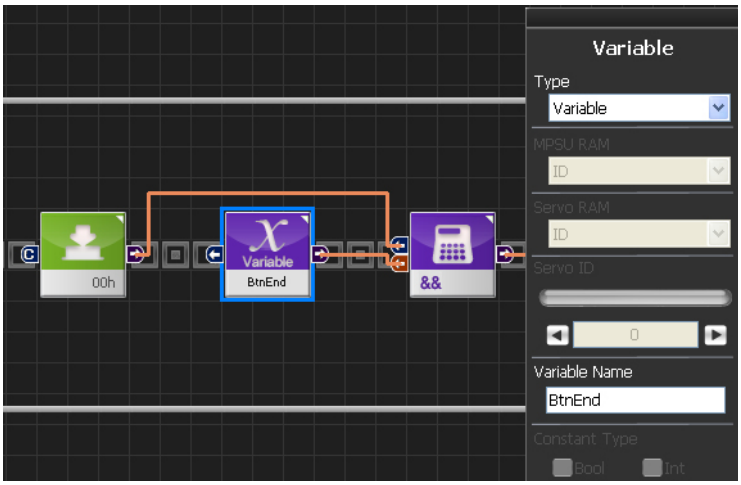


40 버튼이 안눌린 상태

버튼이 눌려서 원하는 동작을 마친후 BtnEnd 변수는 True로 바뀝니다. BtnEnd 변수가 True가 되면 버튼에 따른 동작을 더이상 하지 않으므로 한번 누를 때 원하는 동작이 한번만 실행되게 할 수 있습니다. 버튼을 떼면, BtnEnd를 false로 초기화 시켜야 합니다. 지금부터는 초기화 과정을 프로그래밍 합니다.

Motion 〉Button 모듈을 선택합니다.
Button Value : 아무것도 선택하지 않습니다.

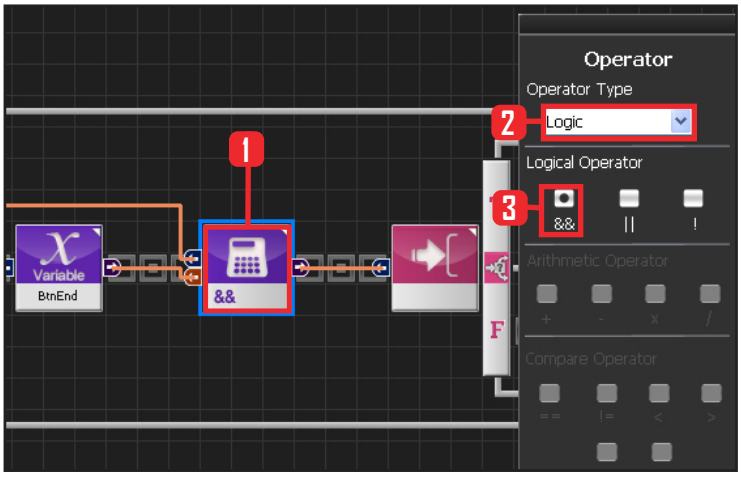
버튼이 안눌린 상태를 의미합니다.



41 BtnEnd 가 True

BtnEnd가 True 일때입니다.

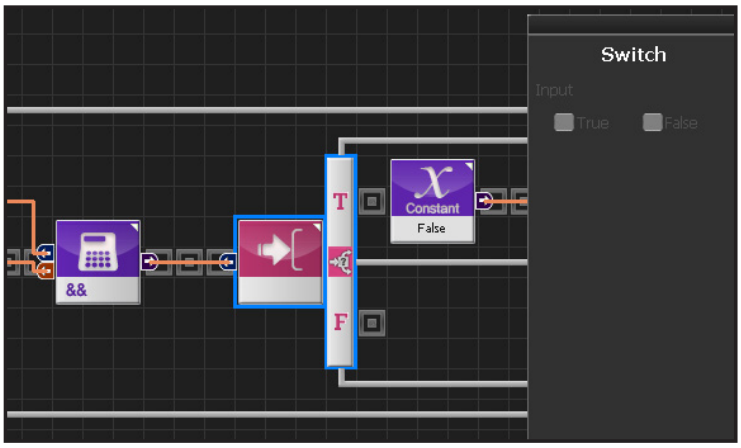
앞의 BtnEnd 변수를 복사합니다.



42 && 연산자

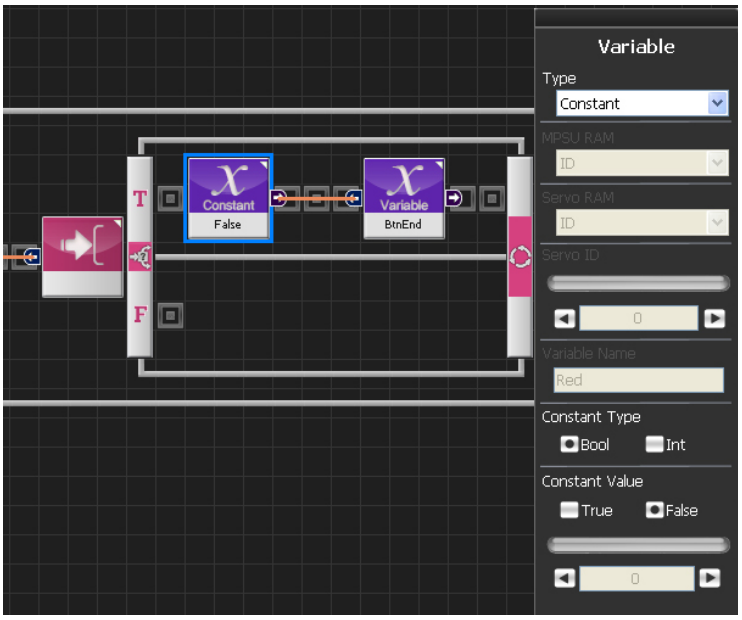
버튼이 안눌린 상태와 BtnEnd 가 True 를 동시에 만족하는 상태는 방금 버튼을 뗀 상태를 의미합니다.

Data 〉 Operator 모듈을 선택합니다.
Operartor Type : Logic으로 선택합니다.
Logical Operator : && 로 선택합니다



43 If 분기문

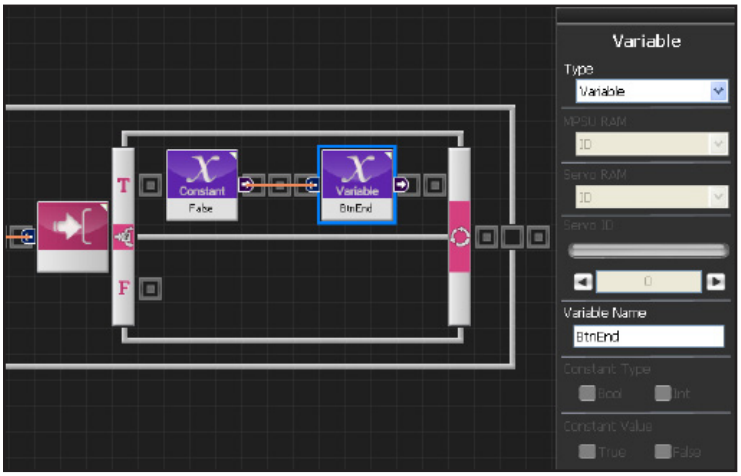
방금 버튼을 눌렀다가 뗀 상태가 True 이면 수행합니다.



44 False 값

BtnEnd 를 True 에서 False 로 만듭니다.

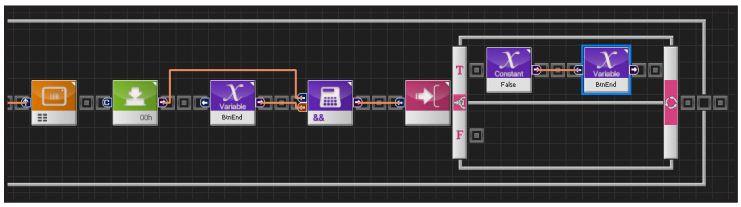
Data 〉 Variable 모듈을 선택합니다.
Type : Contant 를 선택합니다.
Constant Type 은 Bool 로 설정합니다. Bool 은 참과 거짓을 나타내는 자료형입니다.
Constant Value : False 를 선택합니다.



45 BtnEnd를 False로

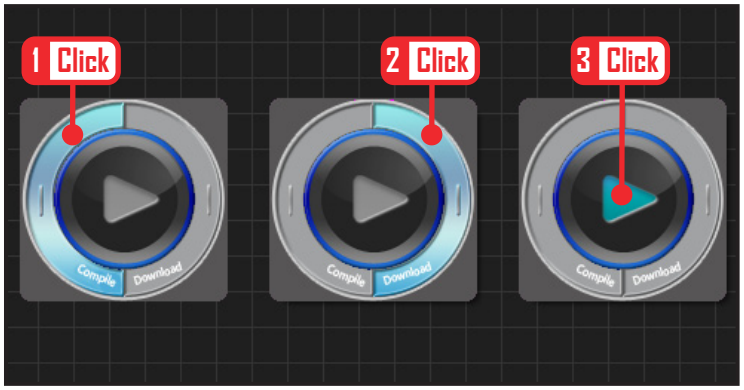
BtnEnd 에 False 값을 입력합니다.

앞의 BtnEnd 변수를 복사합니다.



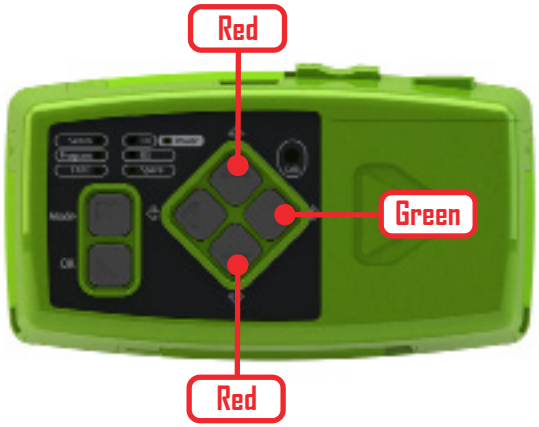
46 버튼 뗐을 때 초기화

방금 버튼을 뗀 상태에 BtnEnd를 false 로 초기화 시킵니다.



47 컴파일, 다운로드, 실행

왼쪽 클릭하여 컴파일 시킵니다. 에러가 없으면 오른쪽 클릭하여 로봇에 다운로드 시킵니다. 다운로드 완료되면 가운데 화살표 실행버튼을 눌러 로봇에서 실행시킵니다.



48 로봇동작

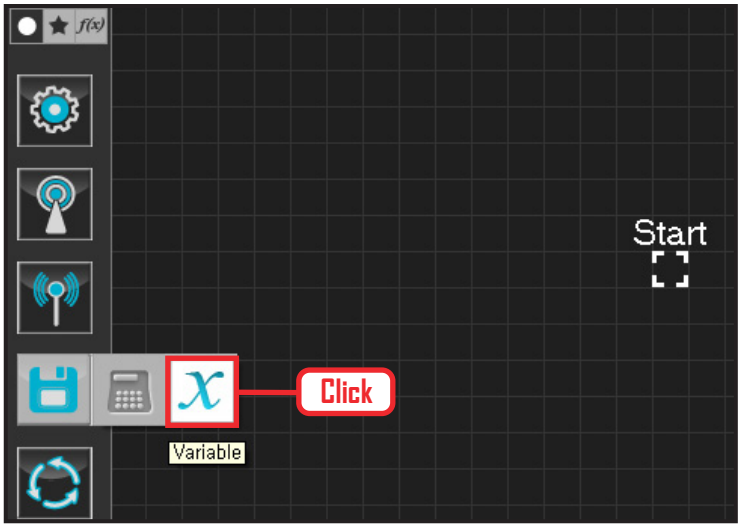
Up 버튼에 Red
Right 버튼에 Green
Down 버튼에 Blue 가 켜졌다 다시 한번 누르면 꺼집니다.

Light 예제 따라하기

예제설명

빛의 세기에 따라 로봇의 모터를 동작하는 예제입니다.

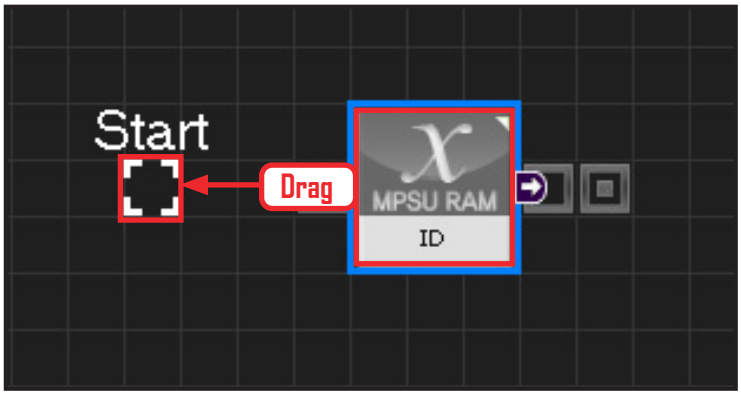
외부의 빛이 어두워지면 로봇이 왼팔을 올립니다. (제어기 뒤쪽의 CDS 센서를 손바닥으로 가리면 빛의 들어오는 양이 없어 어두워지므로, 로봇이 왼팔을 올리도록 프로그래밍합니다.)



01 변수 지정

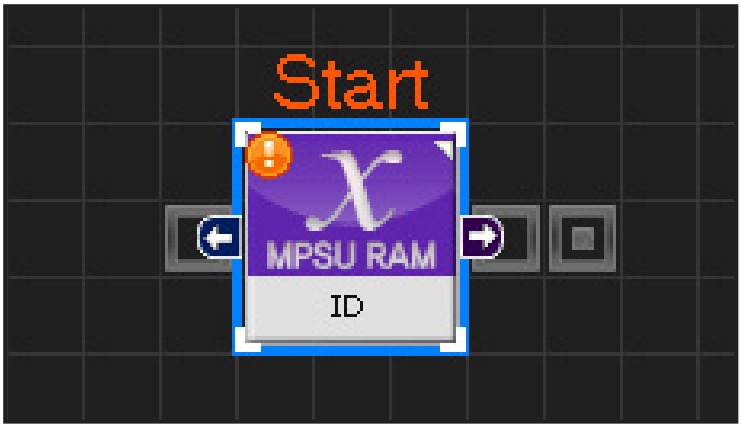
로봇을 동작시킨다는 것은 로봇의 서보 모터를 동작시킨다는 의미입니다. 서보가 스스로 움직일 수 있는 상태로 값을 지정해주어야 합니다.

Data 〉 Variable 모듈을 클릭합니다.



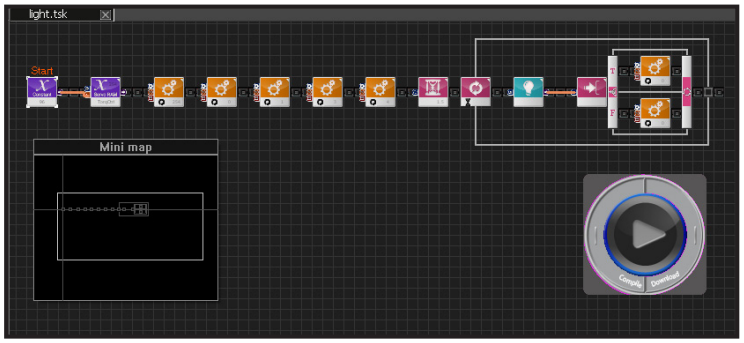
02 시작

모듈의 왼쪽 연결선을 Start Point 에 드래그하여 정확히 도킹을 시킵니다.



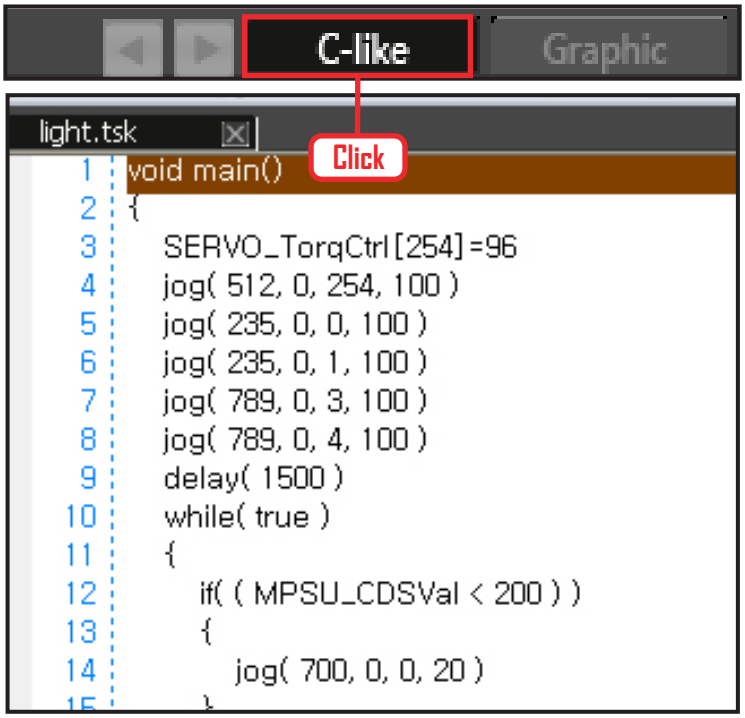
03 프로그래밍 시작

모듈과 Start Point 가 정확히 도킹하면 왼쪽과 같이 활성화된 칼라 이미지 모듈로 변합니다.
그럼 프로그래밍이 시작되었다는 의미입니다.



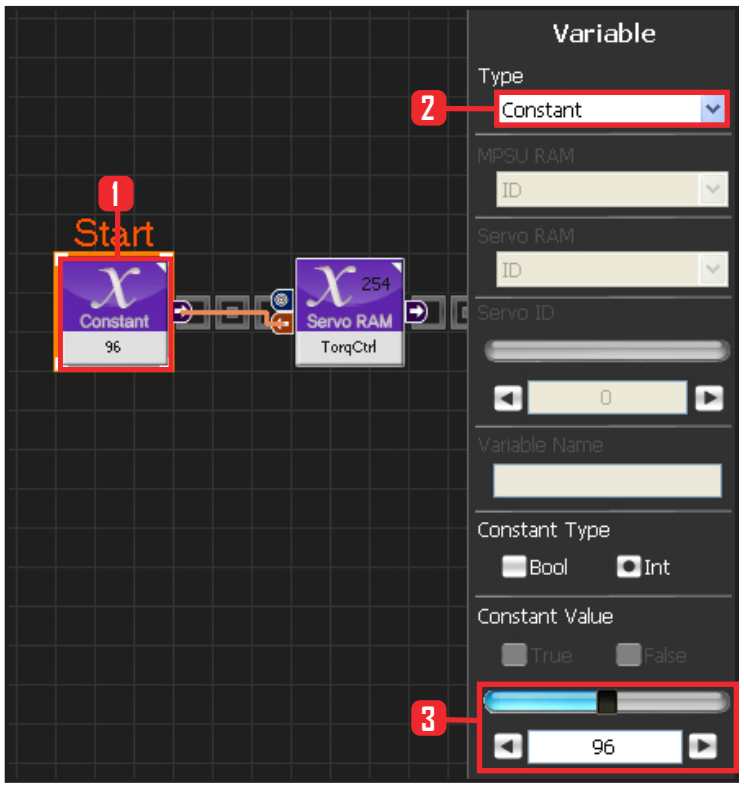
04 전체 프로그래밍

빛 센서를 이용하여 로봇의 모터를 움직이는 전체 프로그래밍 전개 화면입니다.



05 C-Like 보기

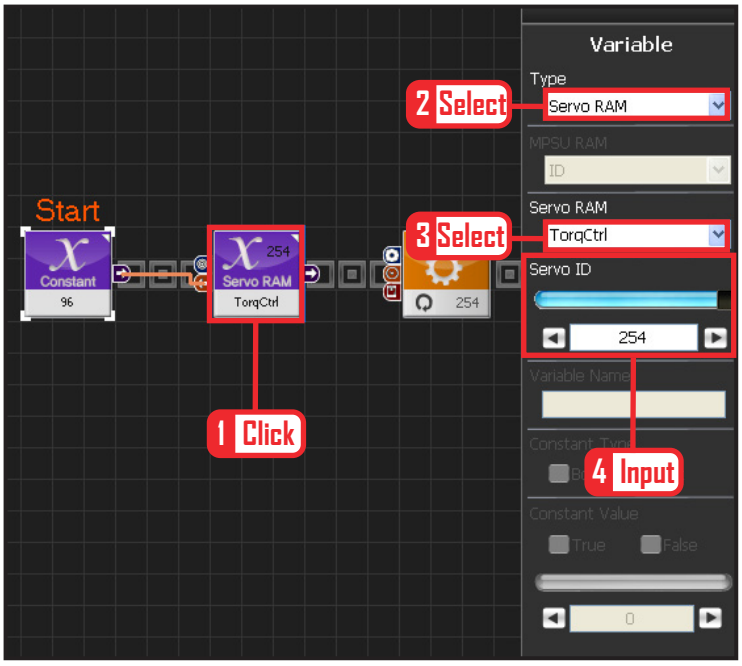
오른쪽 상단의 Graphic 탭에서 C-like 탭을 클릭하면 왼쪽과 같은 Task 프로그래밍 화면이 나옵니다.
빛센서를 이용한 전체 프로그래밍 화면입니다.
C와 유사한 문법 구조를 가지고 있으므로 C 문법 선행학습 효과도 있습니다.
각 모듈별로 클릭하면 커서가 따라서 움직이므로 모듈별로 Text로 어떻게 변환하는지 확인할 수 있습니다.



06 상수 설정

서보 모터를 스스로 움직일 수 있는 상태로 만드는 과정입니다.

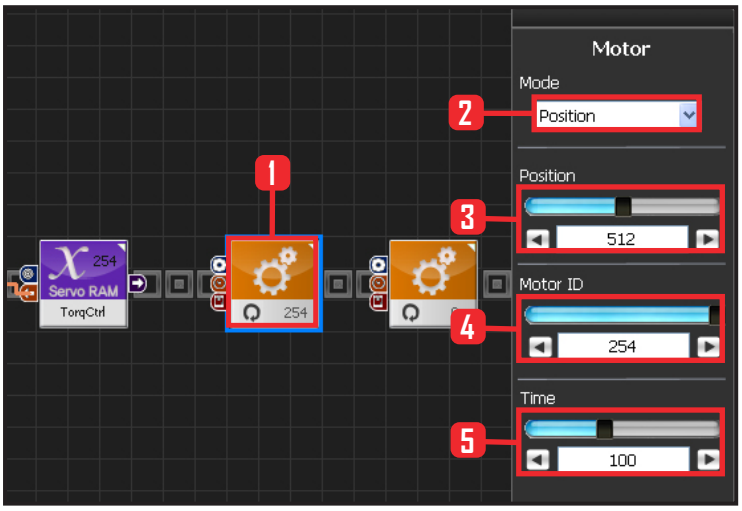
Variable Type 을 Constant 로 선택합니다.
속성중에 Constant Value 값을 96 으로 설정합니다.
서보의 TorqControl 레지스터에 96(0x60) 이라는 값이 들어가면 서보가 움직일 수 있는 상태가 됩니다.
그 값은 Output 커넥터를 통하여 뒤 모듈의 토크값에 전달합니다.



07 모든 서보에 적용

앞에서 받은 96 상수값을 모든 서보에 적용하는 과정입니다.

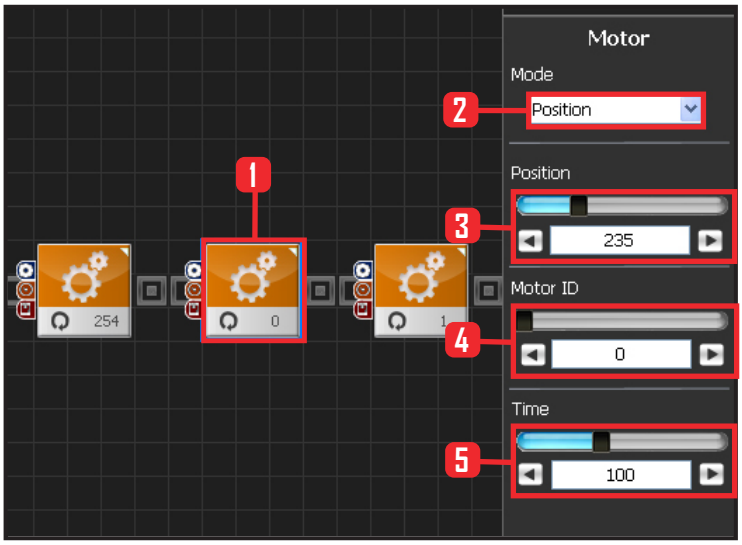
Variable 〉 Type : Servo RAM을 선택합니다.
Servo RAM : TorqCtrl 을 선택합니다.
Servo ID : 254 로 설정합니다. 254는 연결 되어있는 모든 서보에 적용하겠다는 의미입니다.



08 모든 서보모터 각도 설정

모든 서보모터의 각도를 중앙에 보내는 과정입니다.

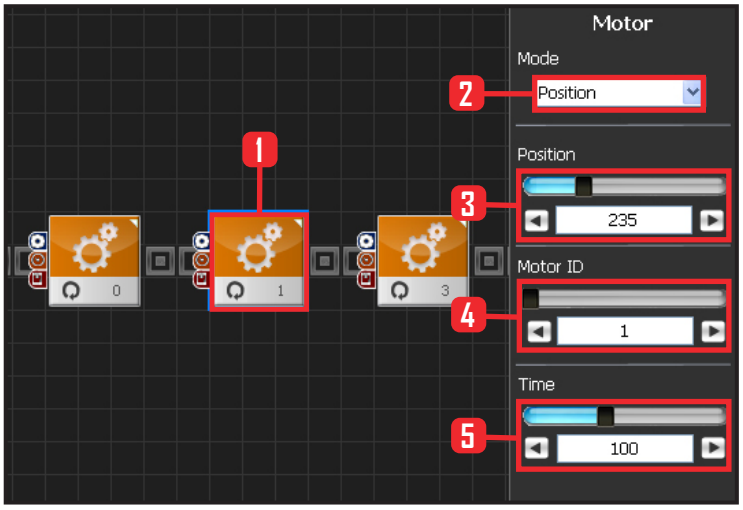
Motion 〉 Moter 를 선택합니다.
Mode : Positon 으로 선택합니다. 각도를 조절합니다.
Position : 512 로 설정합니다. 512 번은 모터를 중앙으로 보낸다는 의미입니다.
Motor ID : 254 로 설정합니다. 254 는 모든 모터에 적용하겠다는 의미입니다.
Time : 100 으로 설정합니다. 단위는 1당 11.2ms로, 100은 약 1.12초를 의미합니다.
1.12초동안 원하는 각도로 이동시킨다는 의미입니다.



09 모터 0번 (오른쪽 어깨) 설정

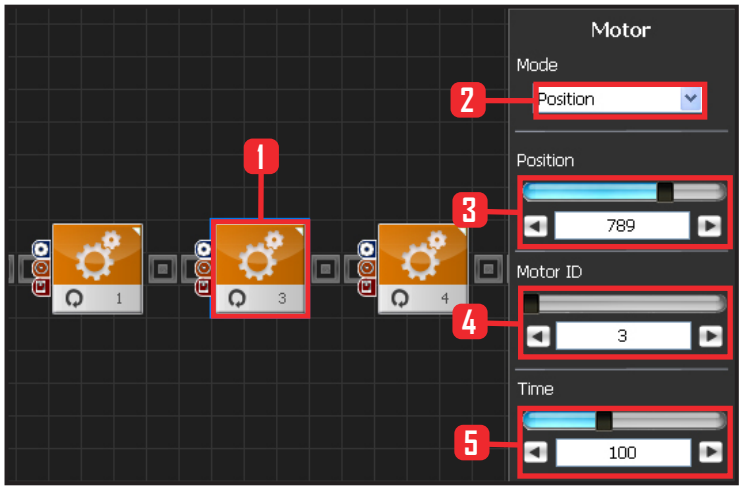
차려자세(기본자세) 만들기
모든 로봇의 모터의 각도를 중앙으로 정렬하면 휴머노이드에서는 팔을 좌우로 뻗게됩니다. 이것을 차려자세로 되돌려 놓아야만 기본 자세를 유지하여 동작시키기가 용이해집니다.

Motion 〉 Moter 를 선택합니다.
Mode : Position 으로 선택합니다.
Position : 235 로 설정합니다. 235 는 수평으로 들고 있던 오른손을 수직으로 내려 갈 수 있게 모터를 돌리게 됩니다.
Motor ID : 0 으로 설정합니다. 오른쪽 어깨 모터 ID가 0번입니다.
Time : 100 으로 설정합니다. 약 1.12초동안 원하는 각도로 이동합니다.



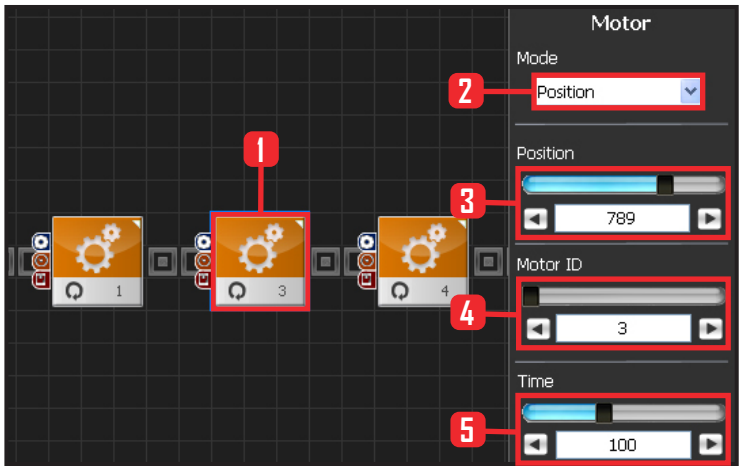
10 모터 1번 (오른쪽 팔) 설정

Mode : Postion 으로 선택합니다.
Position : 235 로 선택합니다. 235는 수평으로 되어 있던 팔을 수직으로 내리게 됩니다.
Motor ID : 1 로 설정합니다. 오른쪽 위쪽 팔 어깨와 닿는 모터가 1번 모터입니다.
Time : 100 으로 설정합니다. 약 1.12초동안 원하는 각도로 이동합니다.



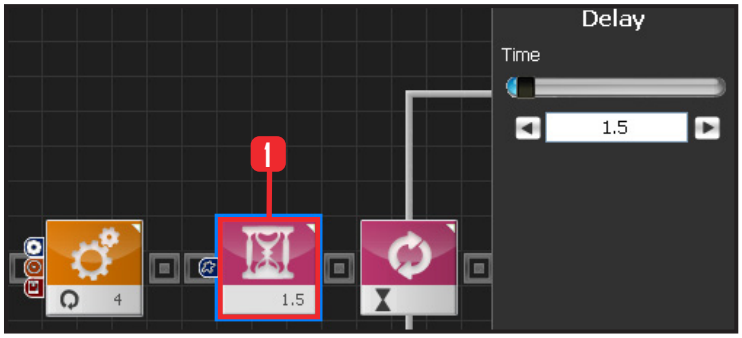
11 모터 3번(왼쪽 어깨) 설정

Mode : Position 으로 선택합니다.
Position : 789 로 설정합니다. 789 는 수평으로 들고 있던 왼손을 수직으로 내려 갈 수 있게 모터를 돌리게 됩니다.
Motor ID : 3 으로 설정합니다. 왼쪽 어깨 모터 ID가 3번입니다.
Time : 100 으로 설정합니다. 약 1.12초동안 원하는 각도로 이동합니다.



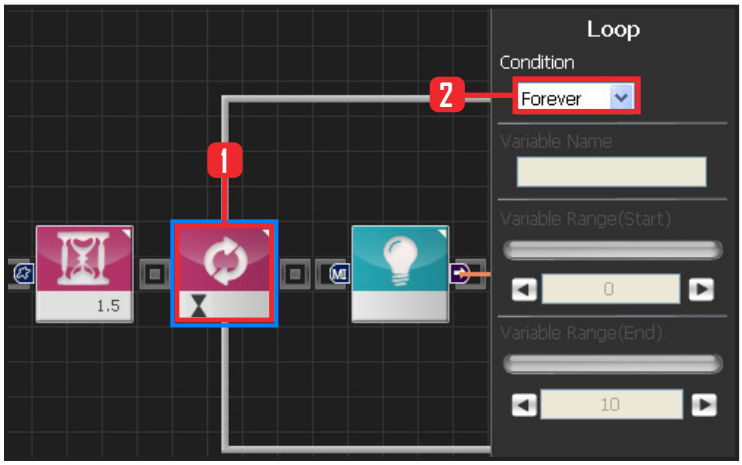
12 모터 4번(왼쪽 팔) 설정

Mode : Postion 으로 선택합니다.
Position : 789 로 선택합니다. 789 는 수평으로 되어 있던 팔을 수직으로 내리게 됩니다.
Motor ID : 4 로 설정합니다. 왼쪽 위쪽팔 어깨와 닿는 모터가 4번 모터입니다.
Time : 100 으로 설정합니다. 약 1.12초동안 원하는 각도로 이동합니다.



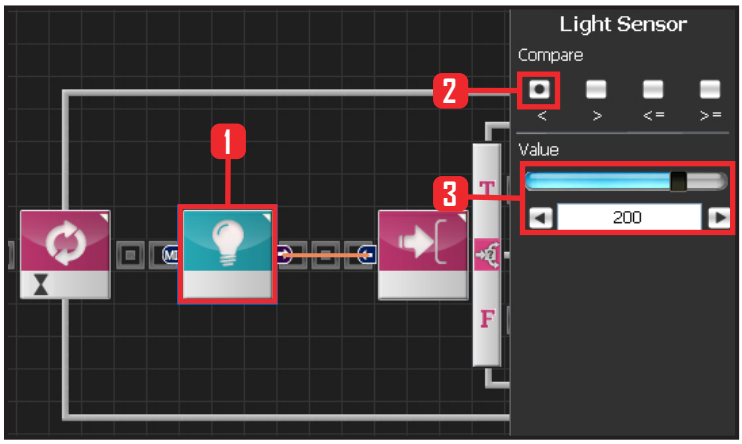
13 Delay 지연시키기

로봇이 차려자세가 되기까지 기다린 후에 다음 모듈을 실행시키기 위해 잠시 지연하기 위한 과정입니다.
Flow 〉 Delay 모듈을 선택합니다.
Time : 1.5 로 설정합니다. 여기에서 단위는 초입니다. 약 1.5초동안 Delay 시킵니다.



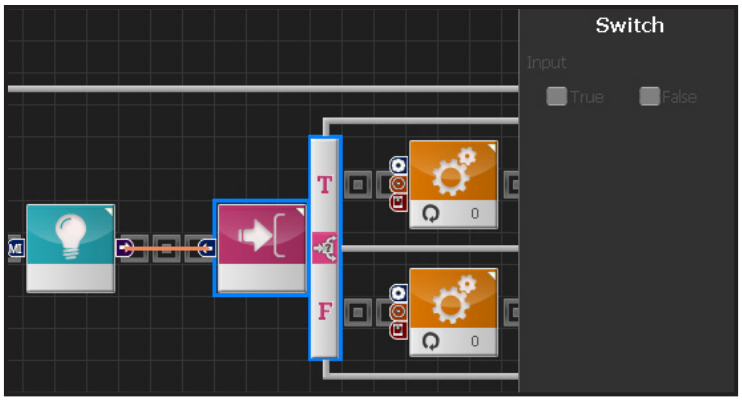
14 Loop 반복문

Flow 〉 Loop 모듈을 선택합니다.
Condition 은 Forever 를 선택합니다. 무한루프로 돌립니다.



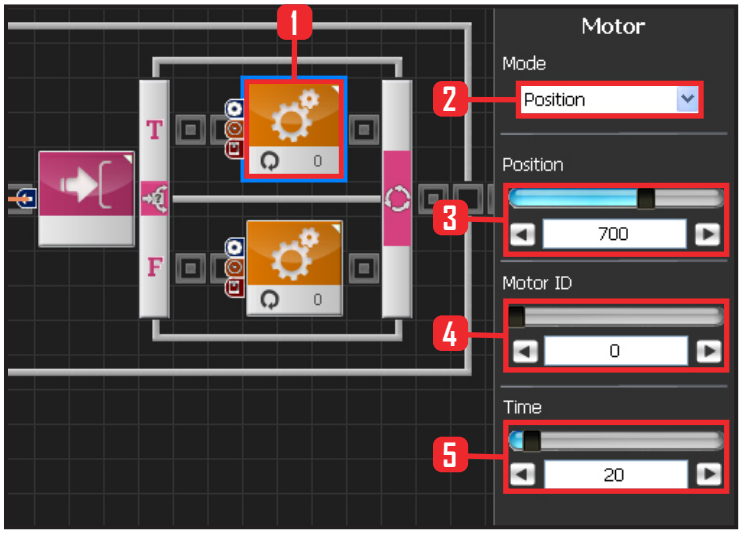
15 Light Sensor 빛 센서

Sensor 〉 Light 모듈을 선택합니다.
Compare : 〈 를 선택합니다. 어떤 값보다 작았을 때를 의미합니다.
Value : 200 으로 설정합니다. 광도값이 200 임을 의미합니다.
광도값이 200 보다 작으면 모듈의 output이 True 가 되고, 크거나 같으면 False가 됩니다.



16 Switch IF 분기문

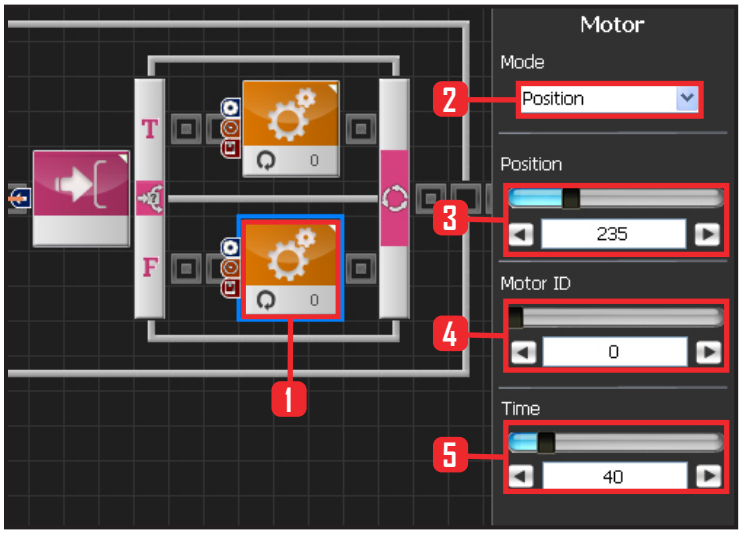
앞의 값이 True 이거나 False 일때 각각에 해당하는 결과를 실행합니다.



17 모터 0번(어깨) 설정

빛광도가 200보다 낮으면, 즉 어두우면(True) 오른쪽 팔을 올리게 되고, 환해지면(False) 오른쪽 팔이 내려진 상태로 그래도 유지됩니다.

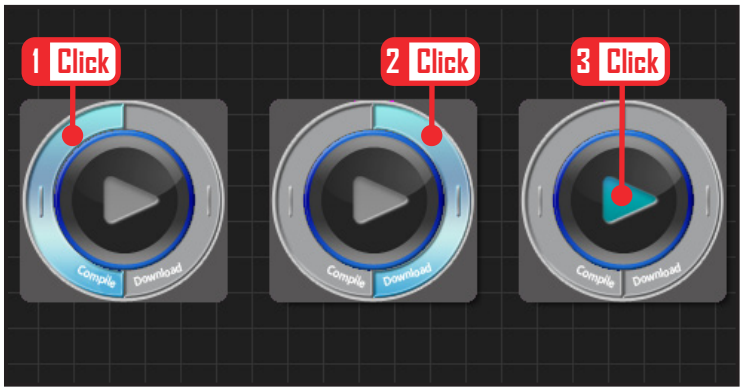
Motion 〉 Moter 를 선택합니다.
Mode : Position 으로 선택합니다.
Position : 700 으로 설정합니다. 700 은 수직 차려자세로 놓여있는 팔을 앞으로 올리게 됩니다.
Motor ID : 0 으로 설정합니다. 오른쪽 어깨 모터 ID 가 0번입니다.
Time : 20 으로 설정합니다.



18 모터 0번(어깨) 설정

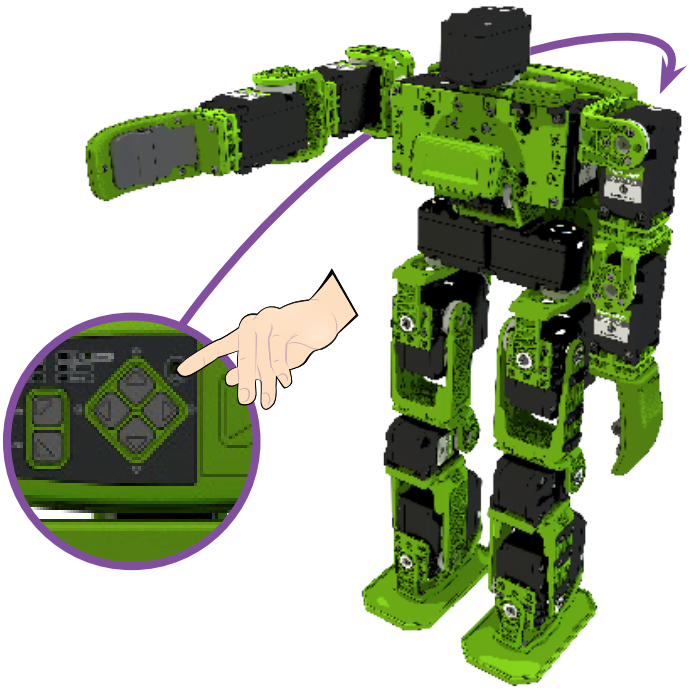
제어기 cds 센서에서 손가락을 떼면, 다시 환해지므로, 모터는 차려자세로 돌아갑니다.

Motion 〉 Moter 를 선택합니다.
Mode : Position 으로 선택합니다.
Position : 235 으로 설정합니다. 235 는 수직 차려자세로 돌아가게 합니다.
Motor ID : 0 으로 설정합니다. 오른쪽 어깨 모터 ID 가 0번입니다.
Time : 40 으로 설정합니다. 올라간 속도보다. 좀 더 느리고 내려옵니다.



18 다운로드

프로그래밍 후 컴파일 – 로봇에 다운로드 – 실행 하는 과정을 거칩니다.
Compile을 클릭합니다. 에러가 없으면, 우측 Download 를 클릭합니다. 로봇에 다운로드 합니다.
다운로드 완료된 후, 실행버튼(중간 화살표)을 클릭합니다.
로봇에서 외부 입력을 기다리게 됩니다.



19 로봇동작

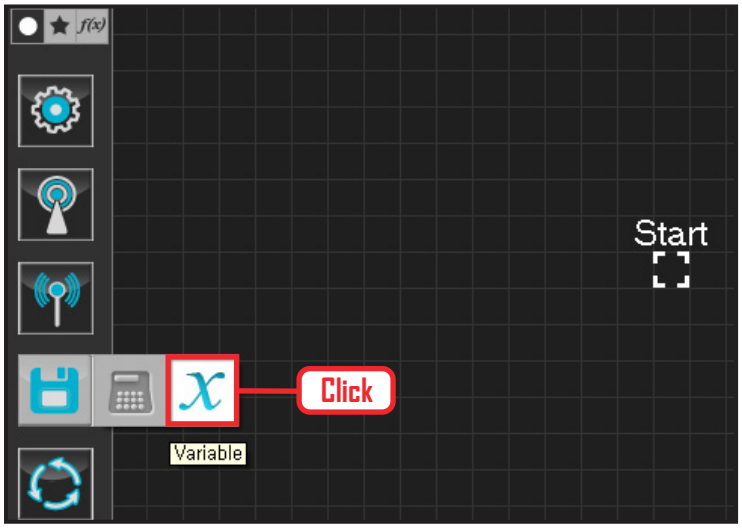
현재는 빛이 밝은 상태이므로 차려자세를 유지합니다.
손가락으로 cds 를 가리면, 로봇이 오른팔을 듭니다.
손가락을 떼면 로봇이 오른팔을 내립니다.
정확히 실행되는 것을 확인합니다.

Sound Sensor 예제 따라하기

예제설명

Sound Sensor 는 제어기 DRC 내부의 양쪽에 위치합니다.

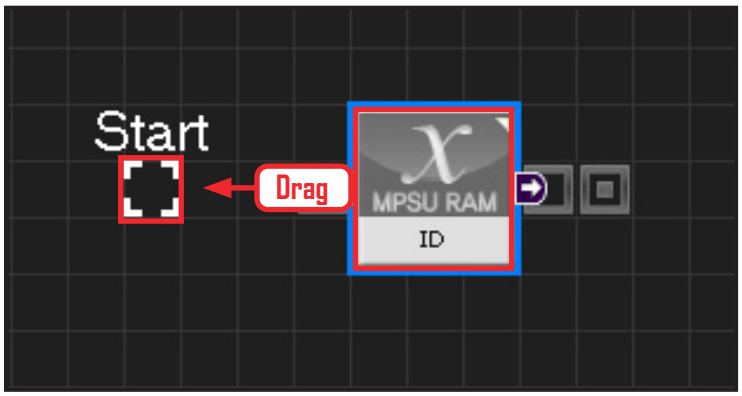
왼쪽측면에서 박수를 치고 왼쪽 손을 들고, 오른쪽측면에서 박수를 치면 오른쪽 손을 드는 프로그래밍을 해봅니다.



01 변수 지정

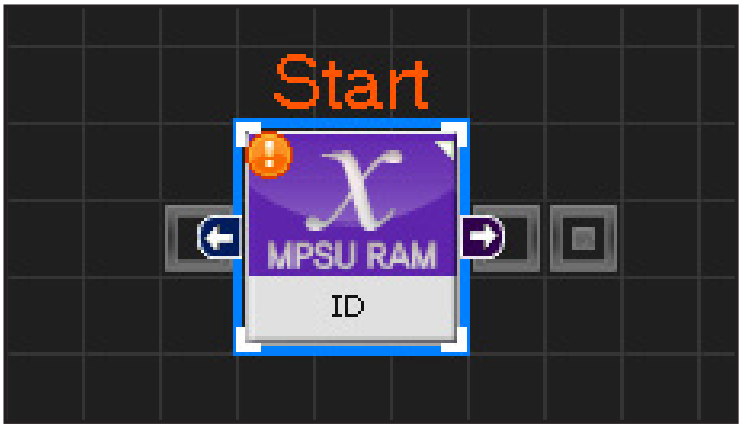
로봇을 동작시킨다는 것은 로봇의 서보 모터를 동작시킨다는 의미입니다. 서보가 스스로 움직일 수 있는 상태로 값을 지정해주어야 합니다.

Data〉Variable 모듈을 클릭합니다.



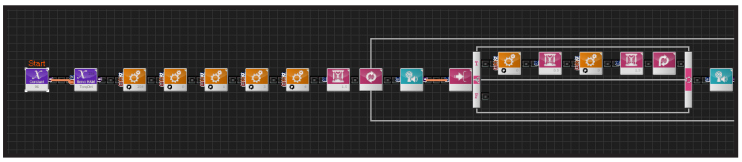
02 시작

모듈의 왼쪽 연결선을 Start Point 에 드래그하여 정확히 도킹을 시킵니다.



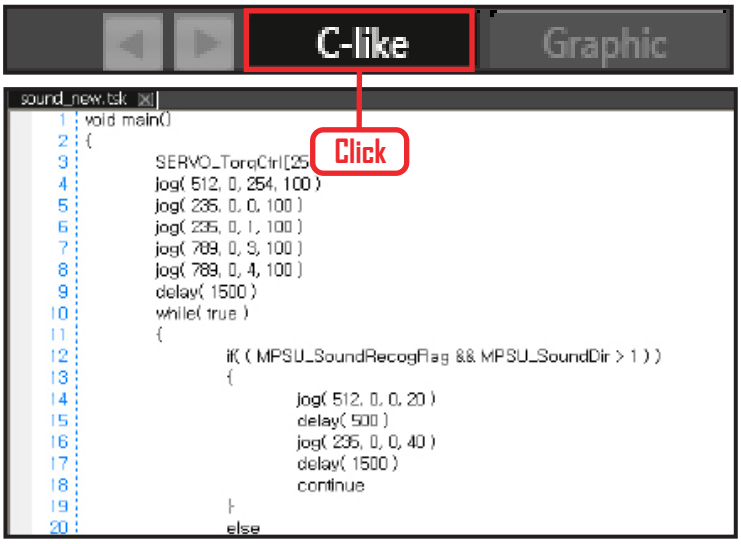
03 프로그래밍 시작

모듈과 Start Point 가 정확히 도킹하면 왼쪽과 같이 활성화된 칼라 이미지 모듈로 변합니다.
그럼 프로그래밍이 시작되었다는 의미입니다.



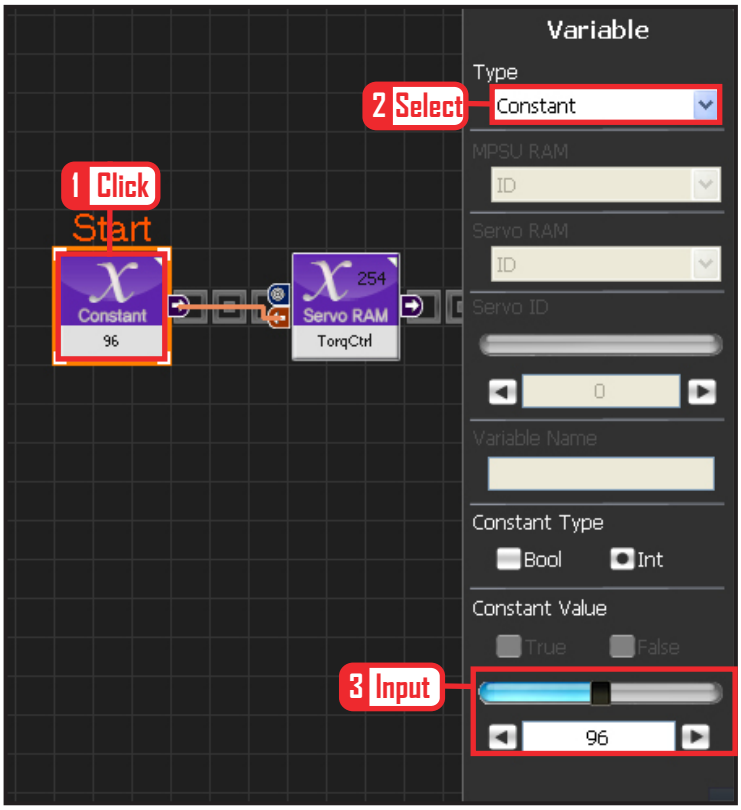
04 전체 프로그래밍

사운드 센서를 이용하여 각각의 모터 동작을 실행 시켜봅니다.



05 C-Like 보기

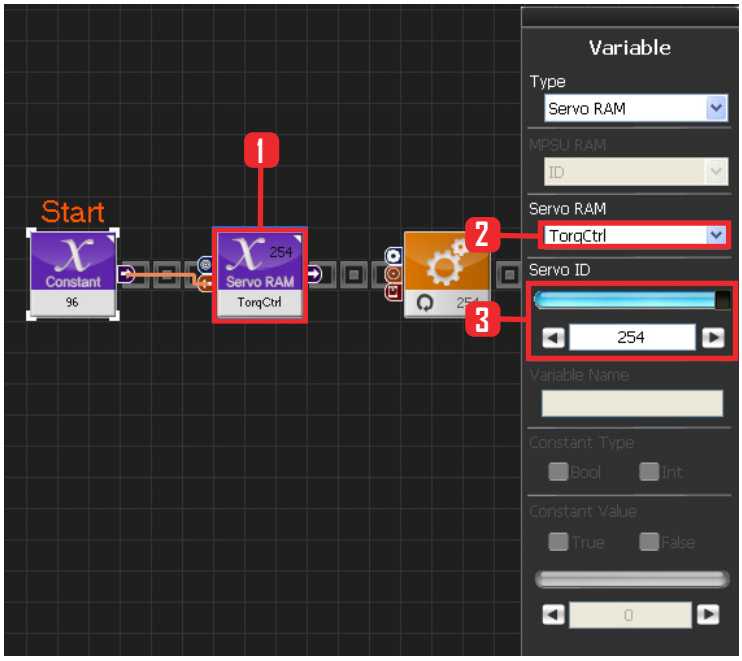
오른쪽 상단의 Graphic 탭에서 C-like 탭을 클릭하면 왼쪽과 같은 Task 프로그래밍 화면이 나옵니다.
사운드 센서를 이용한 전체 프로그래밍 화면입니다.
C와 유사한 문법 구조를 가지고 있으므로 C 문법 선행학습 효과도 있습니다.
각 모듈별로 클릭하면 커서가 따라서 움직이므로 모듈별로 Text로 어떻게 변환하는지 확인할 수 있습니다.



06 상수 설정

서보 모터를 스스로 움직일 수 있는 상태로 만드는 과정입니다.

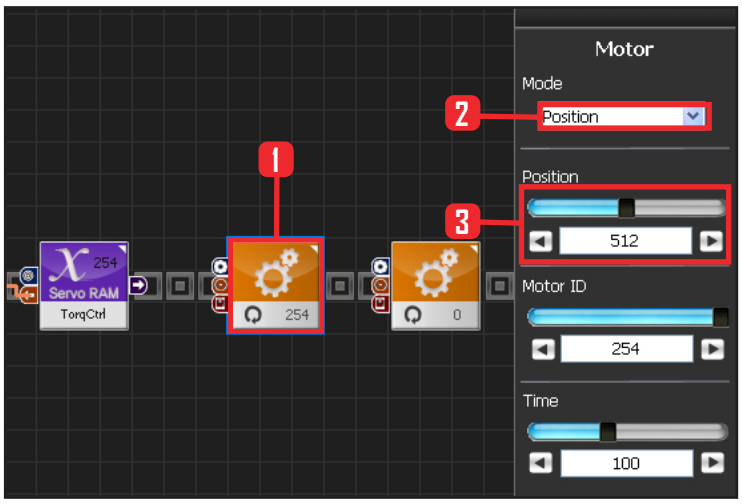
Variable Type 을 Constant 로 선택합니다.
속성중에 Constant Value 값을 96 으로 설정합니다.
서보의 TorqControl 레지스터에 96(0x60) 이라는 값이 들어가면 서보가 움직일 수 있는 상태가 됩니다.
그 값은 Output 커넥터를 통하여 뒤 모듈의 토크값에 전달합니다.



07 모든 서보에 적용

앞에서 받은 96 상수값을 모든 서보에 적용하는 과정입니다.

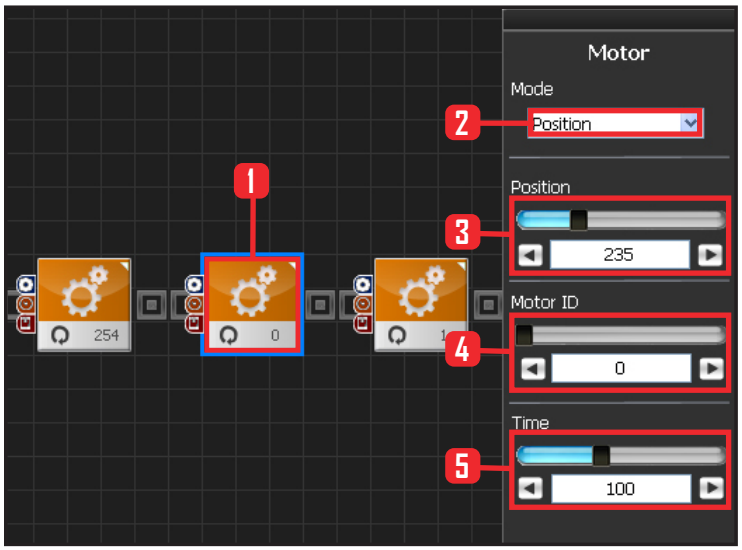
Variable 〉 Type : Servo RAM을 선택합니다.
Servo RAM : TorqCtrl 을 선택합니다.
Servo ID : 254 를 선택합니다. 254는 연결되어있는 모든 서보에 적용하겠다는 의미입니다.



08 모든 서보모터 각도 설정

모든 서보모터의 각도를 중앙에 보내는 과정입니다.

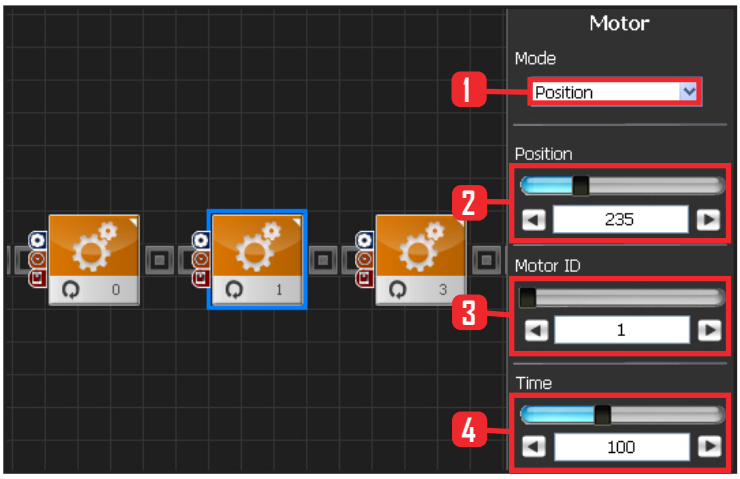
Motion 〉 Moter 를 선택합니다.
Mode : Positon 으로 선택합니다. 각도를 조절합니다. Position : 512 로 설정합니다. 512 번은 모터를 중앙으로 보낸다는 의미입니다
Motor ID : 254 로 설정합니다. 254 는 모든 모터에 적용하겠다는 의미입니다.
Time : 100 으로 설정합니다. 단위는 1당 11.2ms로, 100은 약 1.12초를 의미합니다.
1.12초동안 원하는 각도로 이동시킨다는 의미입니다.



09 모터 0번 (오른쪽 어깨) 설정

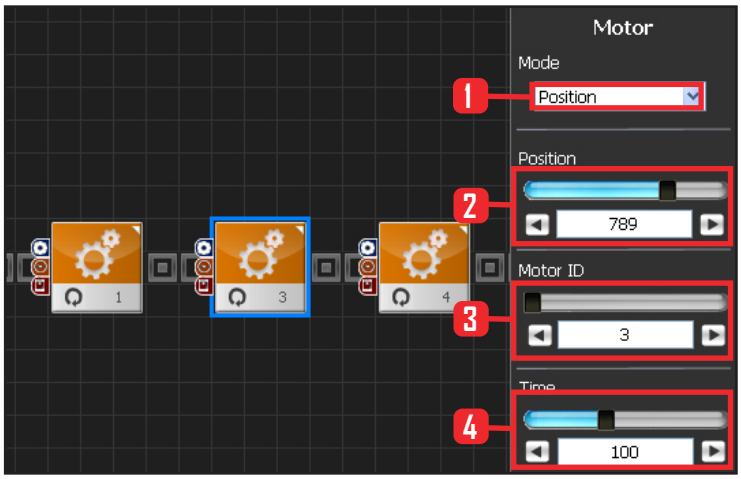
차려자세(기본자세) 만들기
모든 로봇의 모터의 각도를 중앙으로 정렬하면 휴머노이드에서는 팔을 좌우로 뻗게됩니다. 이것을 차려자세로 되돌려 놓아야만 기본 자세를 유지하여 동작시키기가 용이해집니다.

Motion〉Moter 를 선택합니다.
Mode : Position 으로 선택합니다.
Position : 235 로 설정합니다. 235 는 수평으로 들고 있던 오른손을 수직으로 내려갈 수 있게 모터를 돌리게 됩니다.
Motor ID : 0 으로 설정합니다. 오른쪽 어깨 모터 ID가 0번입니다.
Time : 100 으로 설정합니다. 약 1.12초동안 원하는 각도로 이동합니다.



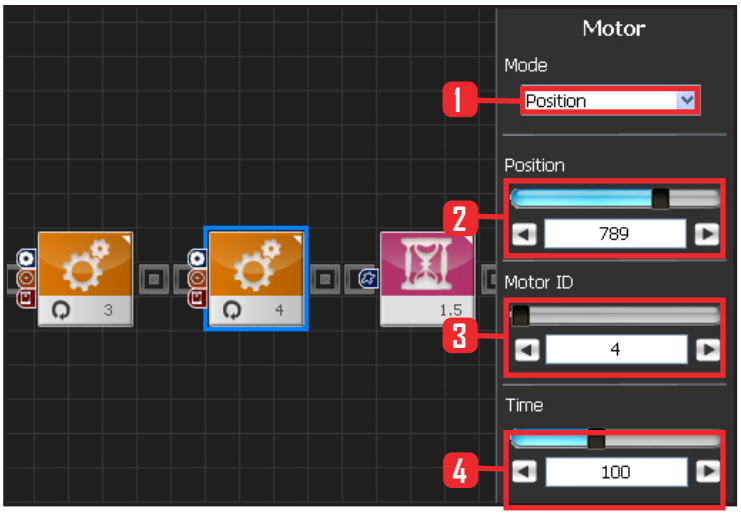
10 모터 1번 (오른쪽 팔) 설정

Mode : Postion 으로 선택합니다.
Position : 235 로 선택합니다. 235는 수평으로 되어 있던 팔을 수직으로 내리게 됩니다.
Motor ID : 1 로 설정합니다. 오른쪽 위쪽 팔 어깨와 닿는 모터가 1번 모터입니다.
Time : 100 으로 설정합니다. 약 1.12초동안 원하는 각도로 이동합니다.



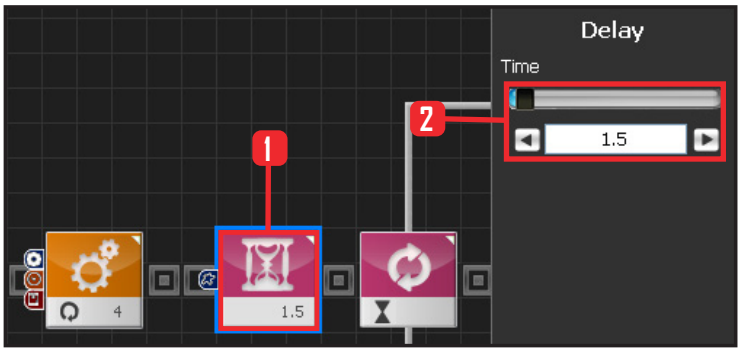
11 모터 3번(왼쪽 어깨) 설정

Mode : Position 으로 선택합니다.
Position : 789 로 설정합니다. 789 는 수평으로 들고 있던 왼손을 수직으로 내려갈 수 있게 모터를 돌리게 됩니다.
Motor ID : 3 으로 설정합니다. 왼쪽 어깨모터 ID가 3번입니다.
Time : 100 으로 설정합니다. 약 1.12초동안 원하는 각도로 이동합니다.



12 모터 4번(왼쪽 팔) 설정

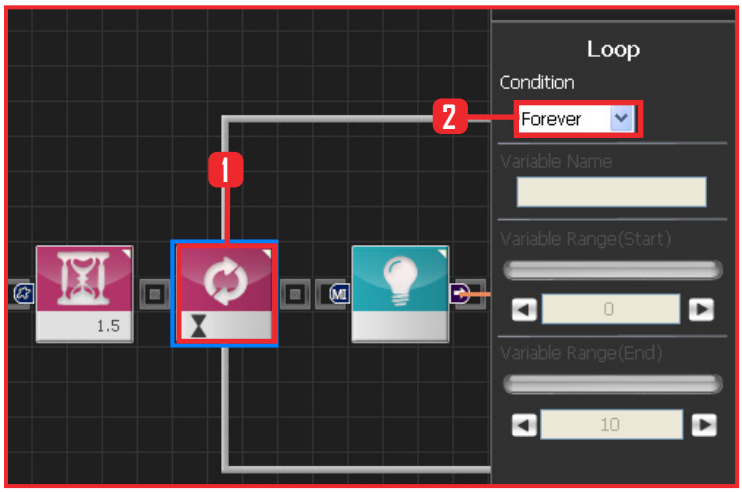
Mode : Postion 으로 선택합니다.
Position : 789 로 선택합니다. 789 는 수평으로 되어 있던 팔을 수직으로 내리게 됩니다.
Motor ID : 4 로 설정합니다. 왼쪽 위쪽팔 어깨와 닿는 모터가 4번 모터입니다.
Time : 100 으로 설정합니다. 약 1.12초동안 원하는 각도로 이동합니다.



13 Delay 지연시키기

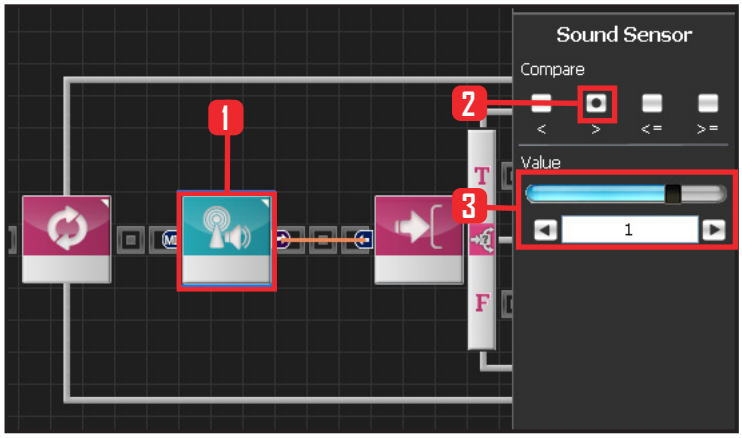
로봇이 차려자세가 되기까지 기다린 후에 다음 모듈을 실행시키기 위해 잠시 지연하기 위한 과정입니다.

Flow 〉 Delay 모듈을 선택합니다.
Time : 1.5 로 설정합니다. 여기에서 단위는 초입니다. 약 1.5초동안 Delay 시킵니다.



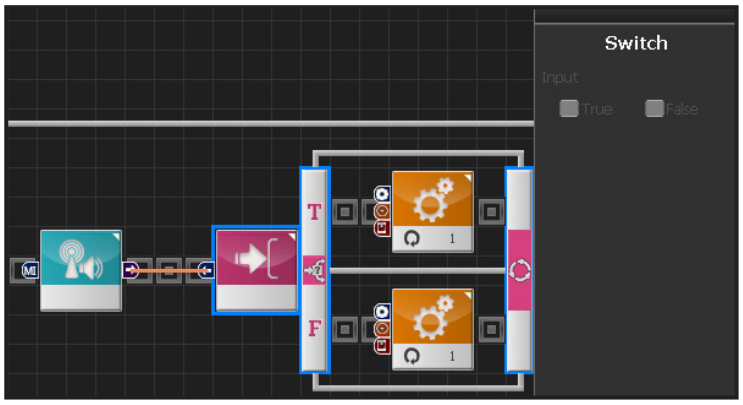
14 Loop 반복문

Flow 〉 Loop 모듈을 선택합니다.
Condition 은 Forever 를 선택합니다. 무한루프로 돌립니다.



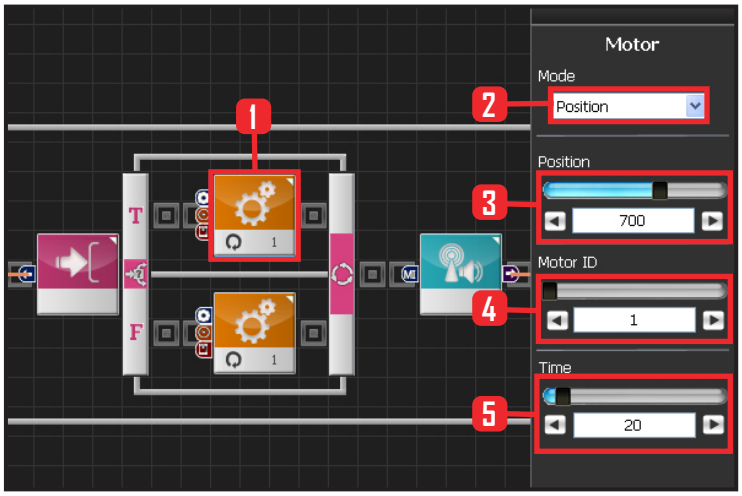
15 Sound Sensor

Sensor 〉 Sound Sensor 모듈을 선택합니다.
Compare : 〉 를 선택합니다. 어떤 값보다 클 때를 의미합니다.
Value : 0 으로 설정합니다. 소리의 방향은 –2 에서 2까지 이며 음수는 왼쪽에서 소리가 났음을, 양수는 오른쪽에서 소리가 났음을 의미합니다.
0보다 크다는 것은 오른쪽에서 소리가 들어왔음을 의미합니다. 소리가 났으며 소리의 방향이 오른쪽이면 Output이 True가 되고, 아니면 False가 됩니다.



16 Switch IF 분기문

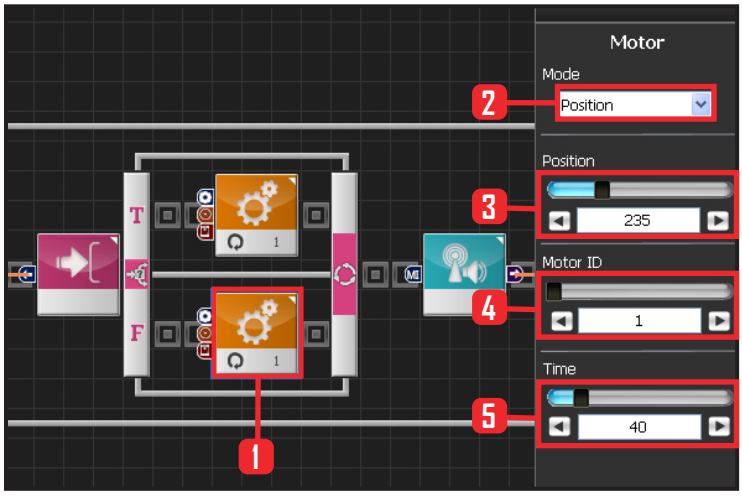
앞의 값이 True 이거나 False 일때 각각에 해당하는 결과를 실행합니다.
오른쪽에서 소리가 났으면 True, 그렇지 않으면 False 입니다.



17 모터 1번(팔) 설정

소리가 오른쪽에서 났으면 True 로 팔을 올립니다.

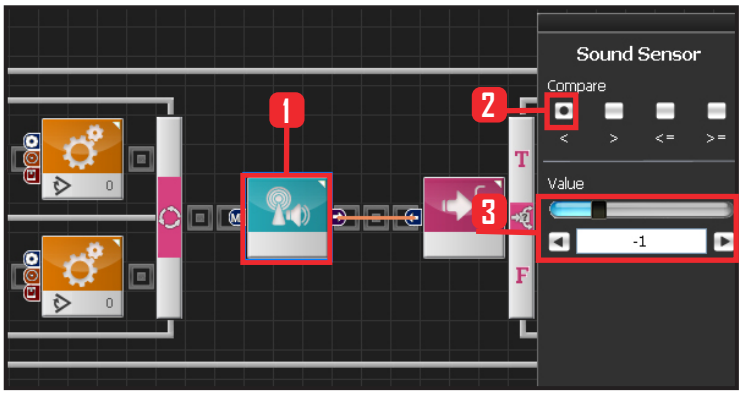
Motion 〉 Moter 를 선택합니다.
Mode : Position 으로 선택합니다.
Position : 700 으로 설정합니다. 700 은 수직 차려자세로 놓여있는 팔을 옆으로 올리게 됩니다.
Motor ID : 1 로 설정합니다. 오른쪽 팔 모터 ID가 1번입니다.
Time : 20 으로 설정합니다.



18 모터 1번(팔) 설정

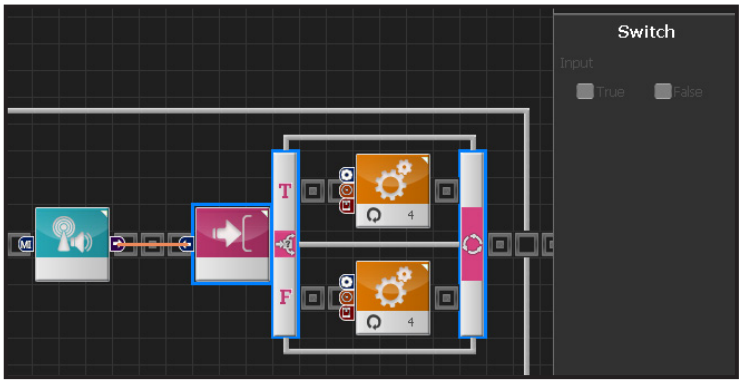
소리가 나지 않았거나, 오른쪽외의 방향에서 났으면 False 로 현재 내려져 있는 팔 상태를 유지합니다.

Motion 〉 Moter 를 선택합니다.
Mode : Position 으로 선택합니다.
Position : 235 으로 설정합니다. 235 는 수직 차려자세를 유지합니다. 팔이 올라가 있어다면 차려자세로 내려옵니다.
Motor ID : 1 로 설정합니다. 오른쪽 팔 모터 ID가 1번입니다.
Time : 40 으로 설정합니다. 올라간 속도보다 좀 더 느리게 내려옵니다.



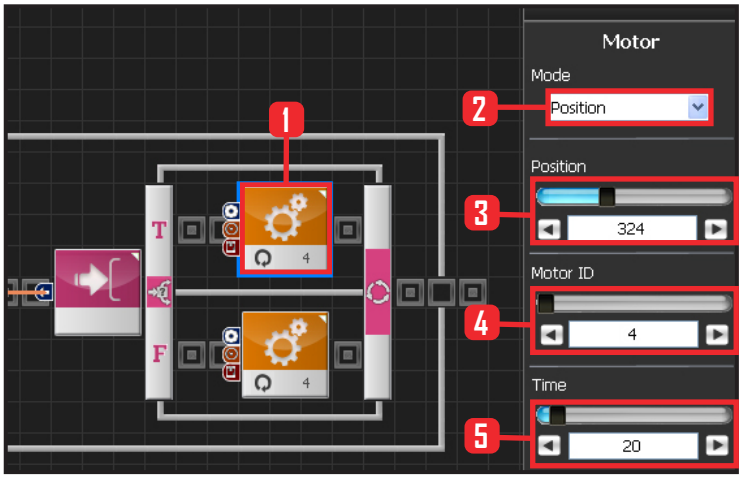
19 Sound Sensor

Sensor 〉Sound Sensor 모듈을 선택합니다.
Compare : 〈 를 선택합니다. 어떤 값보다 크기가 작을 때를 의미합니다.
Value : 0 으로 설정합니다. 소리의 크기는 -2 에서 2까지 이며 음수는 왼쪽, 양수는 오른쪽을 의미합니다. 0 보다 작다는 것은 왼쪽에서 소리가 들어있음을 의미합니다. 소리가 났으며 소리의 방향이 왼쪽인 경우 Output이 True가 되고, 아니면 False가 됩니다.



20 Switch IF 분기문

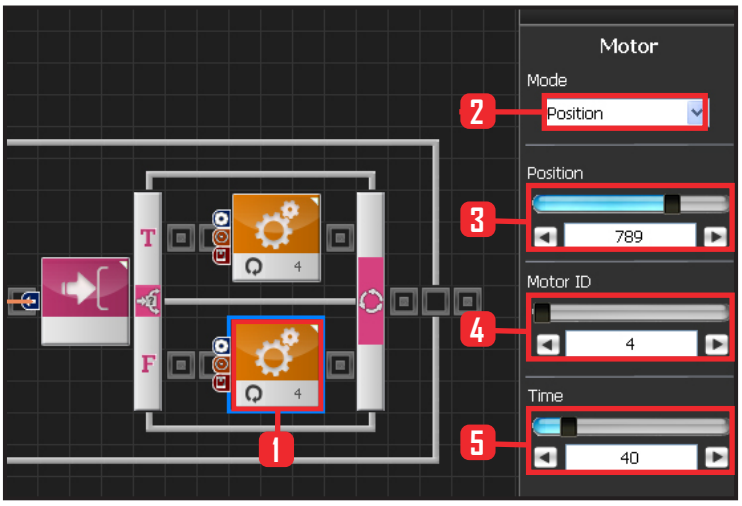
앞의 값이 True 이거나 False 일때 각각에 해당하는 결과를 실행합니다.
왼쪽에서 소리가 났으면 True, 그렇지 않으면 False 입니다.



21 모터 4번(팔) 설정

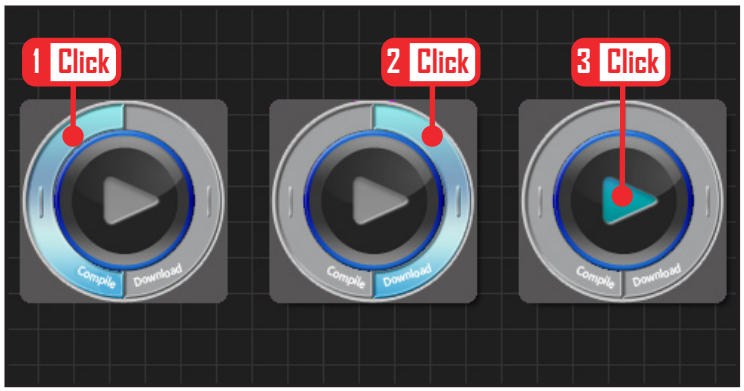
소리가 왼쪽에서 났으면 True 로 왼팔을 올립니다.

Motion 〉Moter 를 선택합니다.
Mode : Position 으로 선택합니다.
Position : 324 으로 설정합니다. 324 는 수직 차려자세로 놓여있는 팔을 옆으로 올리게 됩니다.
Motor ID : 4 로 설정합니다. 왼쪽 팔 모터 ID가 4번입니다.
Time : 20 으로 설정합니다.



22 모터 4번(팔) 설정

소리가 나지 않았거나, 왼쪽 외의 방향에서 났으면 면 False 로 현재 내려져 있는 팔 상태를 유지합니다.
Motion 〉 Moter 를 선택합니다.
Mode : Position 으로 선택합니다.
Position : 789 로 설정합니다. 789 는 수직 차려자세를 유지합니다. 팔이 올라가 있었다면 차려자세로 내려옵니다.
Motor ID : 4 로 설정합니다. 오른쪽 팔 모터 ID가 4번입니다.
Time : 40 으로 설정합니다. 올라간 속도보다. 좀 더 느리게 내려옵니다.



23 컴파일, 다운로드, 실행

왼쪽 클릭하여 컴파일 시킵니다. 에러가 없으면 오른쪽 클릭하여 로봇에 다운로드 시킵니다. 다운로드 완료되면 가운데 화살표 실행버튼을 눌러 로봇에서 실행시킵니다.

24 로봇동작

로봇 오른쪽에서 박수치면 오른쪽 팔을 올리고, 로봇 왼쪽에서 박수치면 왼쪽 팔을 올립니다.

1

2

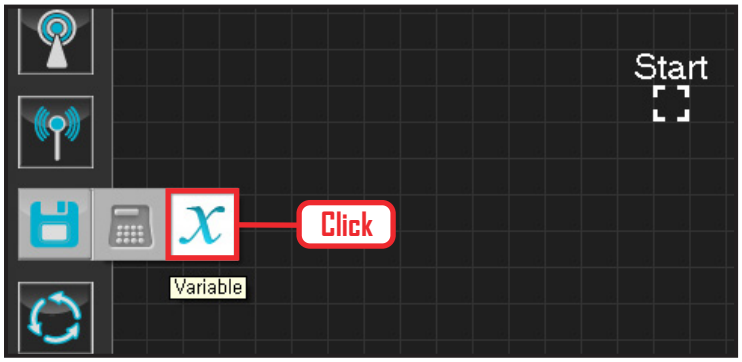
Sound Sensor(심화) 예제 따라하기

예제설명

Sound Sensor 는 제어기 DRC 내부의 양쪽에 위치합니다.

왼쪽측면에서 박수를 치고 왼쪽 손을 들고, 오른쪽측면에서 박수를 치면 오른쪽 손을 드는 프로그래밍이 1차 안 이였습니다.

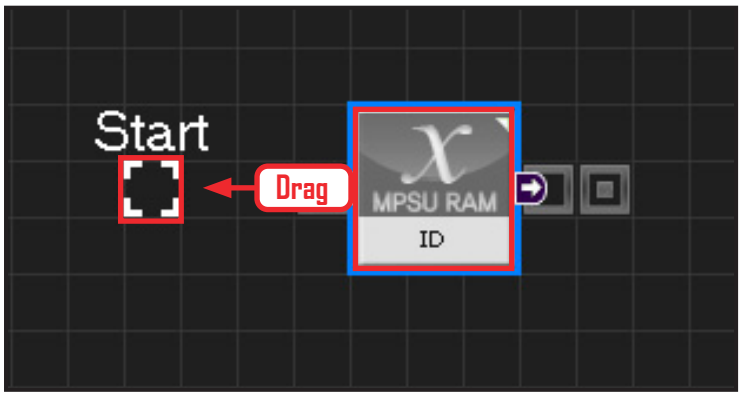
주변에 소음이 많으면, 각각 양쪽에서 소리를 구분하는게 어려워집니다. 한쪽에서 박수를 쳐도 양손을 모두 들거나, 불규칙적으로 반응합니다. 주변에 소음이 있더라도 로봇이 정확한 반응을 한다는 것을 보여주기위해 좀더 세분화된 프로그래밍을 할 필요가 있습니다. 처음 소리가 입력되었을 시 다른 소리가 입력되지 않도록 강제로 Delay 를 줘서 한번 박수 칠 때 한번만 팔을 올리도록 정확도를 높이는 프로그래밍입니다.



01 변수 지정

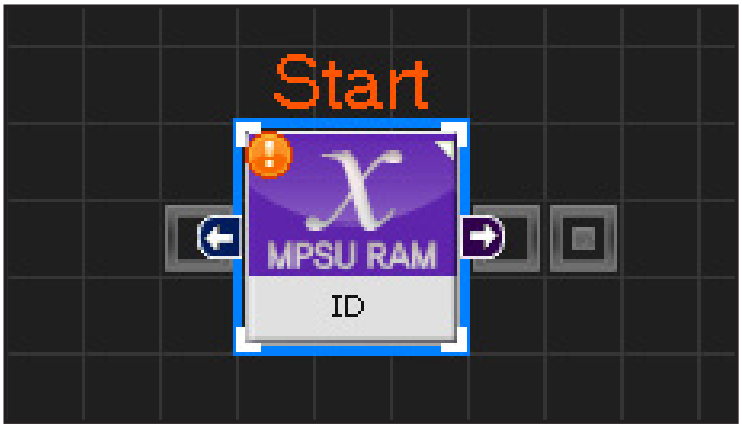
로봇을 동작시킨다는 것은 로봇의 서보 모터를 동작시킨다는 의미입니다. 서보가 스스로 움직일 수 있는 상태로 값을 지정해주어야 합니다.

Data 〉 Variable 모듈을 클릭합니다.



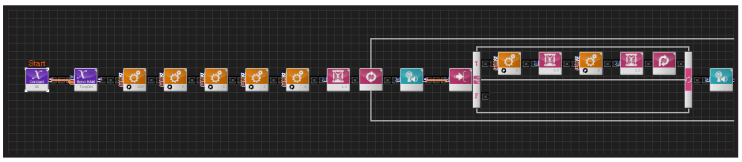
02 시작

모듈의 왼쪽 연결선을 Start Point 에 드래그하여 정확히 도킹을 시킵니다.



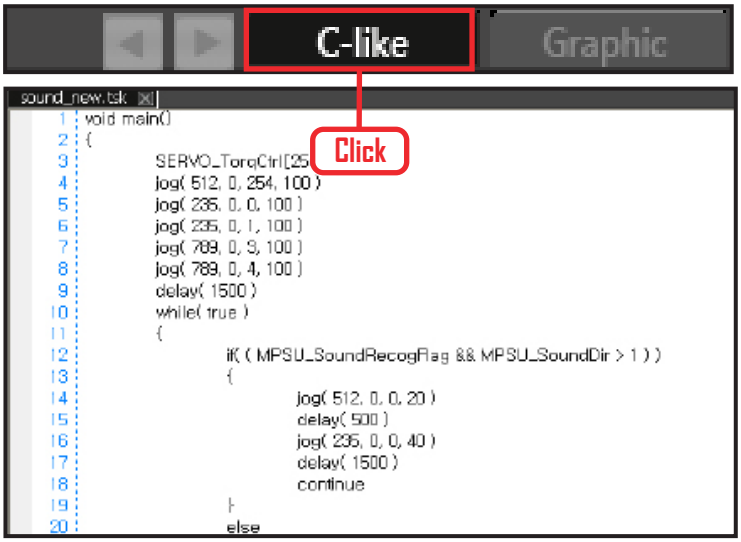
03 프로그래밍 시작

모듈과 Start Point 가 정확히 도킹하면 왼쪽과 같이 활성화된 칼라 이미지 모듈로 변합니다.
그럼 프로그래밍이 시작되었다는 의미입니다.



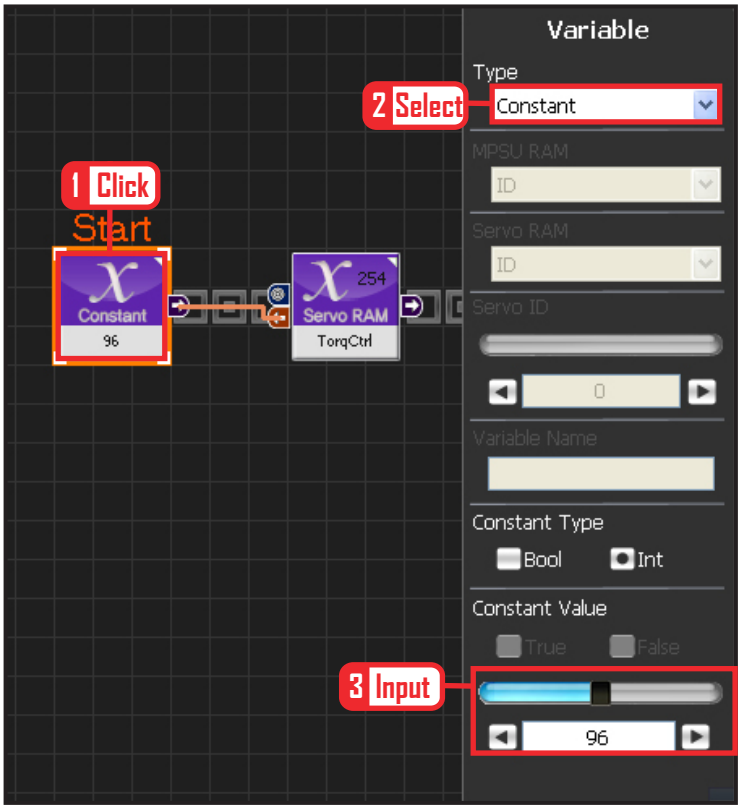
04 전체 프로그래밍

사운드 센서를 민감도를 높여 로봇이 좀더 정확하게 반응하게 프로그래밍합니다.



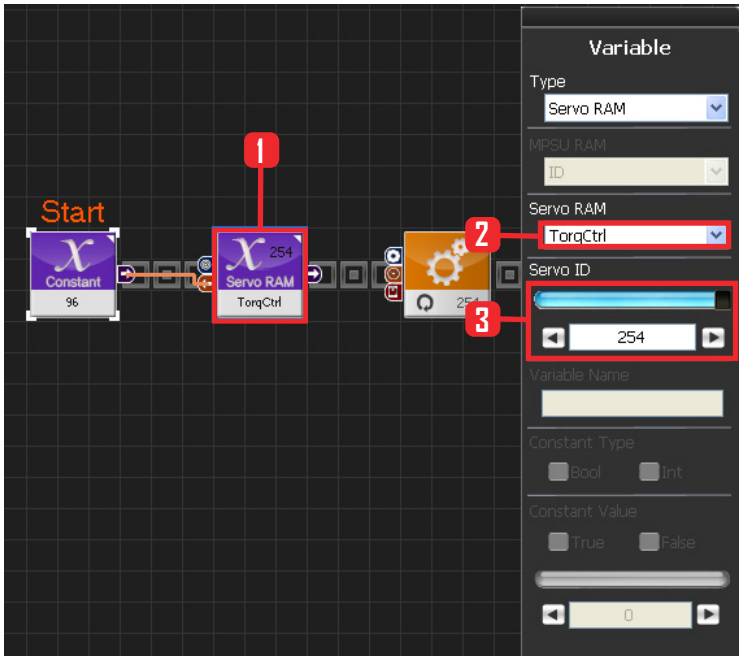
05 C-Like 보기

오른쪽 상단의 Graphic 탭에서 C-like 탭을 클릭하면 왼쪽과 같은 Task 프로그래밍 화면이 나옵니다.
사운드센서를 이용한 전체 프로그래밍 화면입니다.
C와 유사한 문법 구조를 가지고 있으므로 C 문법 선행학습 효과도 있습니다.
각 모듈별로 클릭하면 커서가 따라서 움직이므로 모듈별로 Text로 어떻게 변환하는지 확인할 수 있습니다.



06 상수 설정

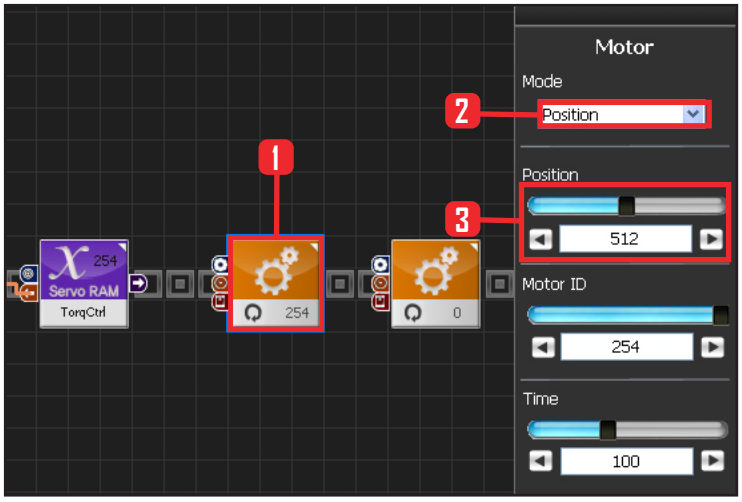
서보 모터를 스스로 움직일 수 있는 상태로 만드는 과정입니다.
Variable Type 을 Constant 로 선택합니다.
속성중에 Constant Value 값을 96 으로 설정합니다.
서보의 TorqControl 레지스터에 96(0x60) 이라는 값이 들어가면 서보가 움직일 수 있는 상태가 됩니다.
그 값은 Output 커넥터를 통하여 뒤 모듈의 토크값에 전달합니다.



07 모든 서보에 적용

앞에서 받은 96 상수값을 모든 서보에 적용하는 과정입니다.

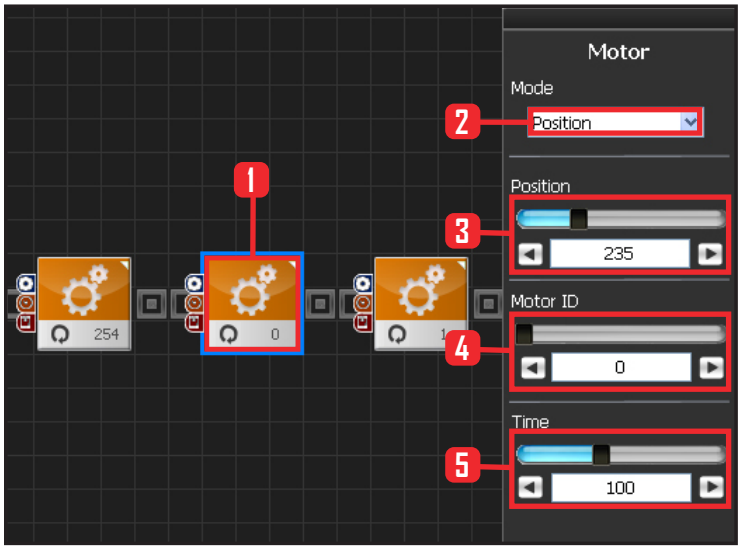
Variable 〉 Type : Servo RAM을 선택합니다.
Servo RAM : TorqCtrl 을 선택합니다.
Servo ID : 254 를 선택합니다. 254는 연결되어있는 모든 서보에 적용하겠다는 의미입니다.



08 모든 서보모터 각도 설정

모든 서보모터의 각도를 중앙에 보내는 과정입니다.

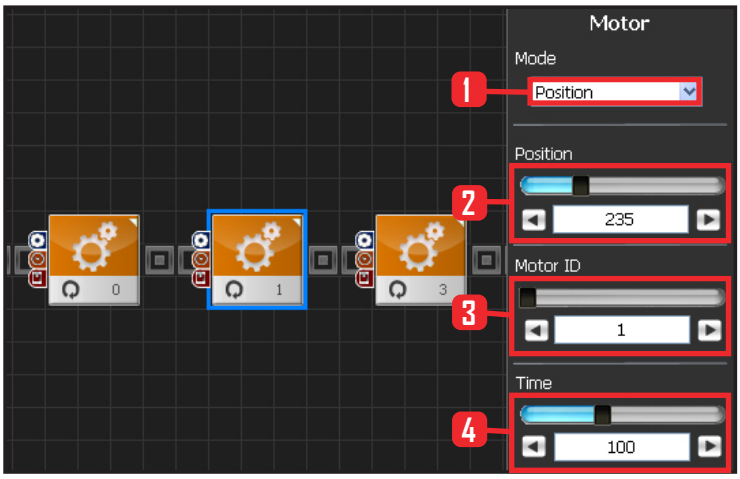
Motion 〉 Moter 를 선택합니다.
Mode : Positon 으로 선택합니다. 각도를 조절합니다.
Position : 512 로 설정합니다. 512 번은 모터를 중앙으로 보낸다는 의미입니다
Motor ID : 254 로 설정합니다. 254 는 모든 모터에 적용하겠다는 의미입니다.
Time : 100 으로 설정합니다. 단위는 1당 11.2ms로, 100은 약 1.12초를 의미합니다.
1.12초동안 원하는 각도로 이동시킨다는 의미입니다.



09 모터 0번 (오른쪽 어깨) 설정

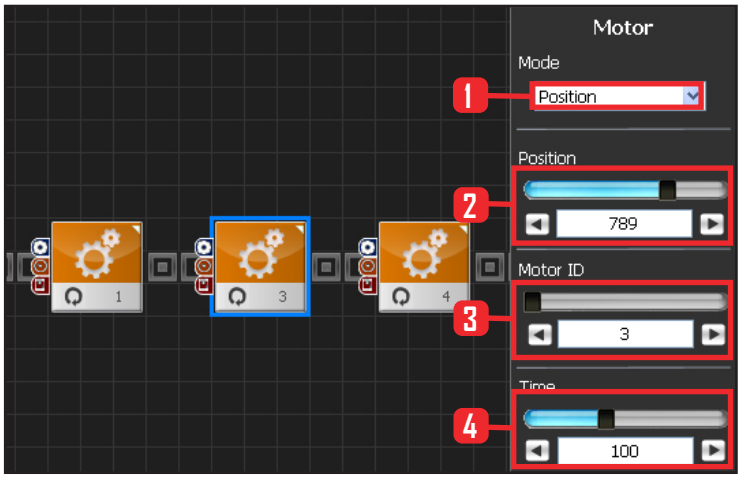
차려자세(기본자세) 만들기
모든 로봇의 모터의 각도를 중앙으로 정렬하면 휴머노이드에서는 팔을 좌우로 뻗게됩니다. 이것을 차려자세로 되돌려 놓아야만 기본 자세를 유지하여 동작시키기가 용이해집니다.

Motion〉Moter 를 선택합니다.
Mode : Position 으로 선택합니다.
Position : 235 로 설정합니다. 235 는 수평으로 들고 있던 오른손을 수직으로 내려갈 수 있게 모터를 돌리게 됩니다.
Motor ID : 0 으로 설정합니다. 오른쪽 어깨 모터 ID가 0번입니다.
Time : 100 으로 설정합니다. 약 1.12초동안 원하는 각도로 이동합니다.



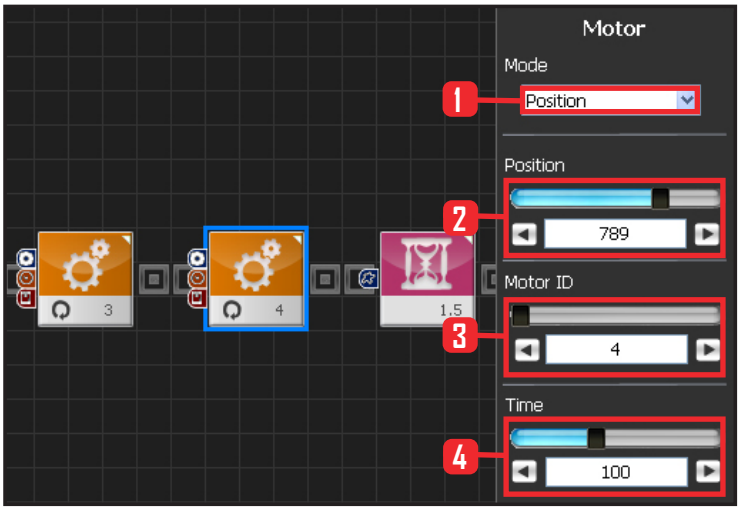
10 모터 1번 (오른쪽 팔) 설정

Mode : Postion 으로 선택합니다.
Position : 235 로 선택합니다. 235 는 수평으로 되어 있던 팔을 수직으로 내리게 됩니다.
Motor ID : 1 로 설정합니다. 오른쪽 위쪽 팔 어깨와 닿는 모터가 1번 모터입니다.
Time : 100 으로 설정합니다. 약 1.12초동안 원하는 각도로 이동합니다.



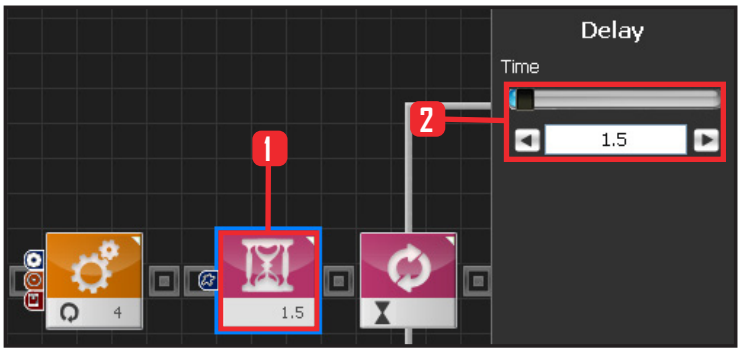
11 모터 3번(왼쪽 어깨) 설정

Mode : Position 으로 선택합니다.
Position : 789 로 설정합니다. 789 는 수평으로 들고 있던 왼손을 수직으로 내려갈 수 있게 모터를 돌리게 됩니다.
Motor ID : 3 으로 설정합니다. 왼쪽 어깨모터 ID가 3번입니다.
Time : 100 으로 설정합니다. 약 1.12초동안 원하는 각도로 이동합니다.



12 모터 4번(왼쪽 팔) 설정

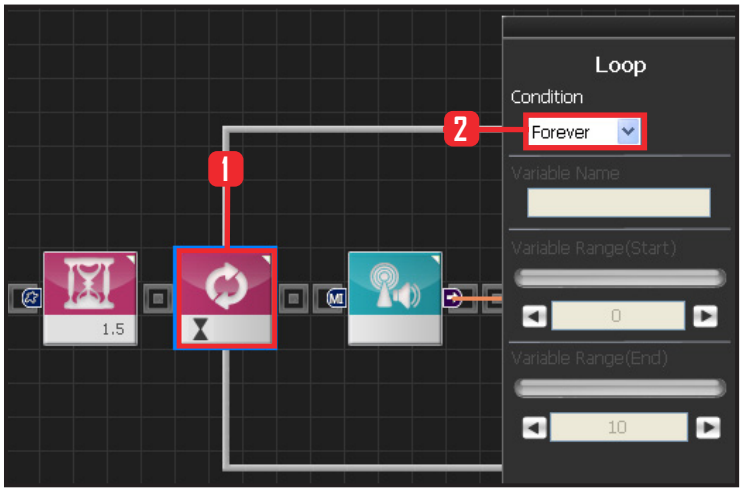
Mode : Postion 으로 선택합니다.
Position : 789 로 선택합니다. 789 는 수평으로 되어 있던 팔을 수직으로 내리게 됩니다.
Motor ID : 4 로 설정합니다. 왼쪽 위쪽팔 어깨와 닿는 모터가 4번 모터입니다.
Time : 100 으로 설정합니다. 약 1.12초동안 원하는 각도로 이동합니다.



13 Delay 지연시키기

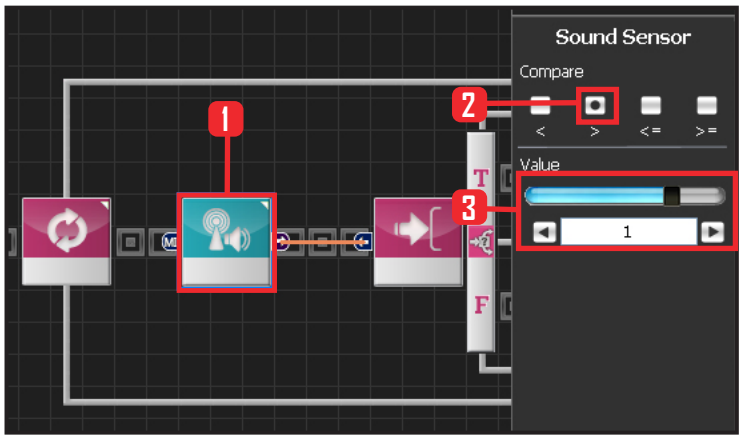
로봇이 차려자세가 되기까지 기다린 후에 다음 모듈을 실행시키기 위해 잠시 지연하기 위한 과정입니다.

Flow 〉 Delay 모듈을 선택합니다.
Time : 1.5 로 설정합니다. 여기에서 단위는 초입니다. 약 1.5초동안 Delay 시킵니다.



14 Loop 반복문

Flow 〉 Loop 모듈을 선택합니다.
Condition 은 Forever 를 선택합니다. 무한루프로 돌립니다.

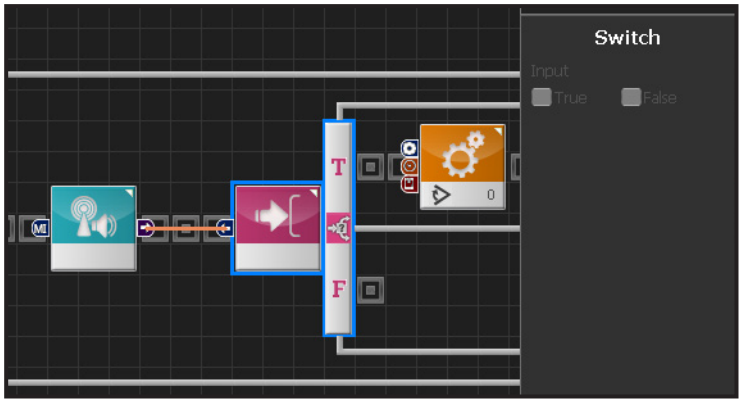


15 Sound Sensor

Sensor 〉 Sound Sensor 모듈을 선택합니다.

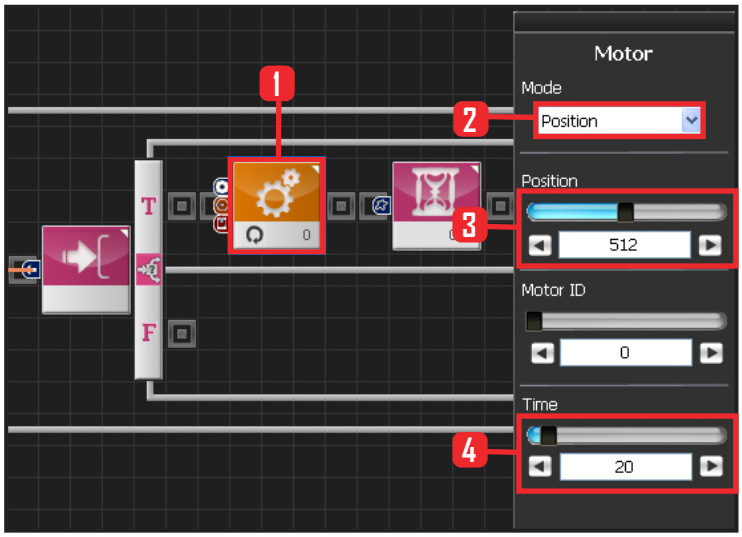
Compare : 〉 로 설정합니다.

Value : 1 로 설정합니다.
소리에 대한 중간값은 0 입니다. 하지만 소리 입력 감도를 더 낮춰서 1보다 컸을 때, 즉 확실히 오른쪽에서 입력받았을 때만 반응하도록 설정합니다. 주변의 소음입력이나 양손을 같이 드는 현상을 방지하기 위해서 입니다.



16 Switch IF 분기문

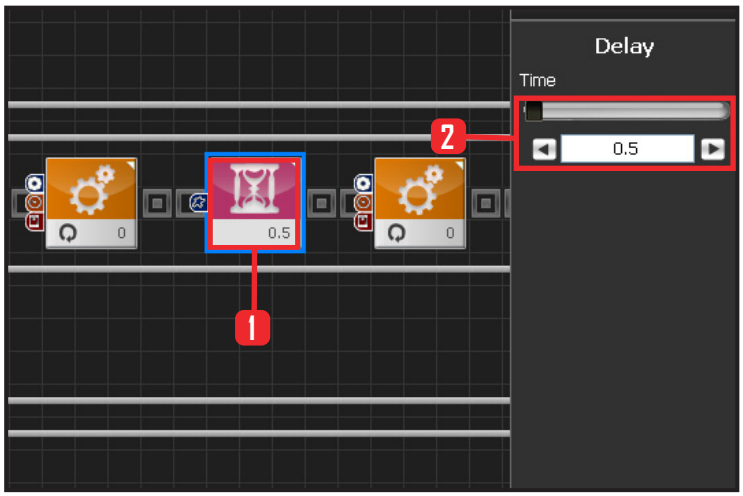
분기문입니다. 앞의 조건이 True 일 때 실행합니다.



17 모터 0번(어깨) 설정

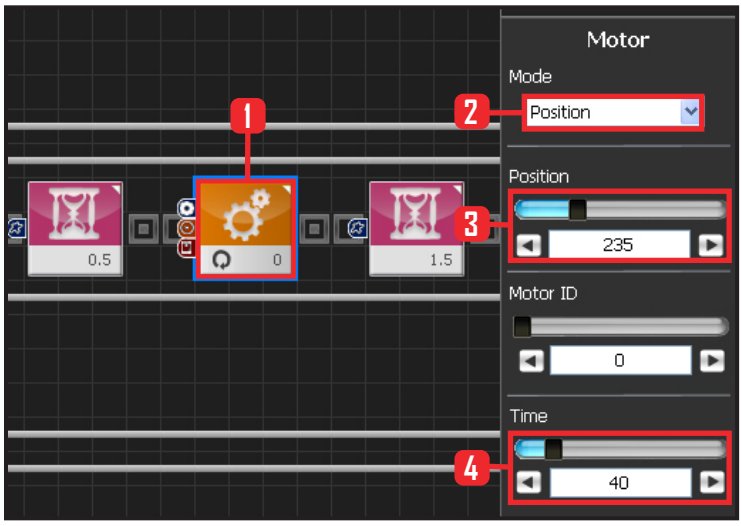
소리 방향값이 1보다 크면 True 로 팔을 올립니다.
팔 동작시에 외부의 잡음이나 모터 소리에 의해서 다시 다른 팔이 동작되는 경우가 있습니다. 이것을 방지하기 위해서 한 팔이 동작할 때는 다른 팔이 동작하지 못하도록 프로그래밍해봅니다.

Motion 〉 Moter 를 를 선택합니다.
Mode : Position 을 선택합니다.
Position 은 512에 위치시킵니다. 512는 양팔 벌리는 자세입니다.
Time 시간은 20으로 설정합니다.
로봇 오른쪽 팔이 차려자세에서 90도 각도로 올라갑니다.



18 Delay

팔 동작중에 외부의 잡음이 들리면 곧바로 다시 반응한다거나, 다른팔이 움직일 수 있습니다. 이러한 현상을 방지하기 위해서 현재 팔이 움직이는 동안 다른 현상이 일어나지 않게 Delay 값을 줍니다.
0.5초의 Delay 동안 오른팔을 움직이는 것 외에 아무 움직임도 없습니다.

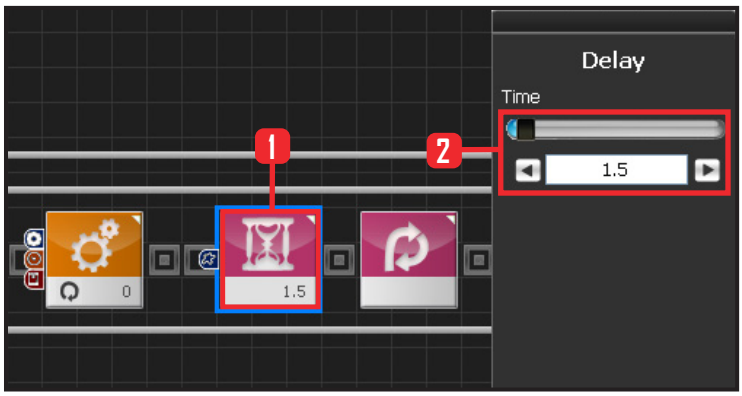


19 모터 0번(어깨) 차려자세 복귀

90도로 올라간 팔을 다시 차려자세로 내립니다.

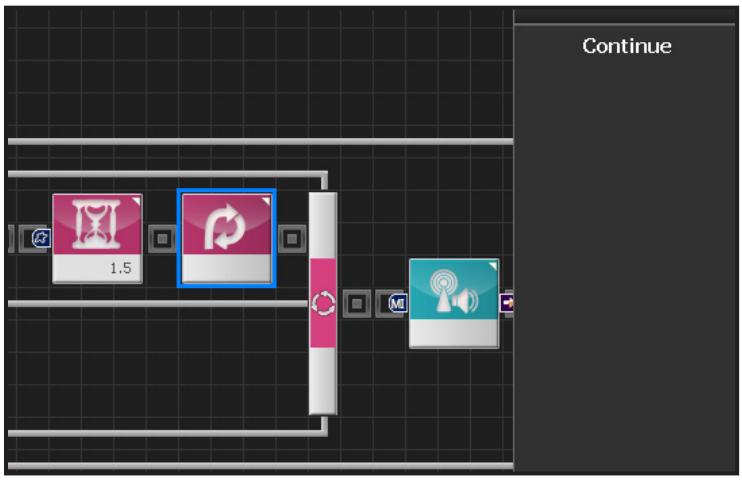
Motion 〉 Moter 를 를 선택합니다.
Mode : Position 을 선택합니다.
Position 은 235에 위치시켜 차려자세로 되돌립니다.
시간은 40으로 설정합니다.

로봇 오른쪽 팔이 차려자세로 돌아옵니다.



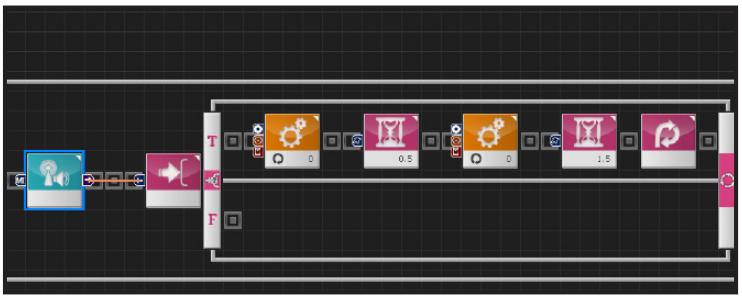
20 Delay

차려자세로 돌아와서 한동안 다른 동작을 못하게 Delay 값을 줍니다.
1.5초 동안 Delay 값을 주면, 로봇은 팔을 내린후 동작하지 않으며 소리입력 여부를 검사하지 않습니다.
그 이후 소리가 들리면 반복합니다.



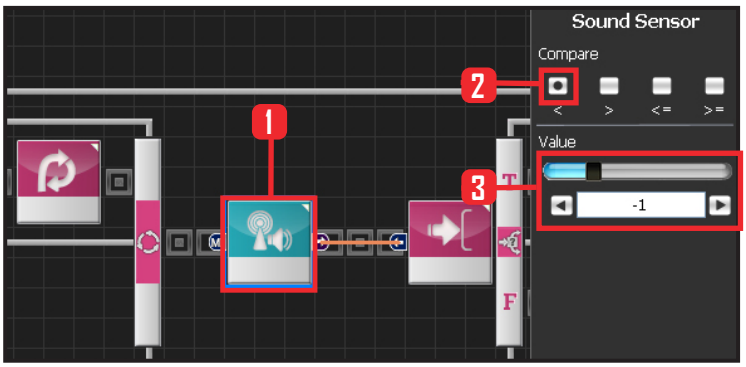
21 Continue

1.5초동안 Delay 이후 다시 Loop 의 처음으로 돌아갑니다.



22 요약

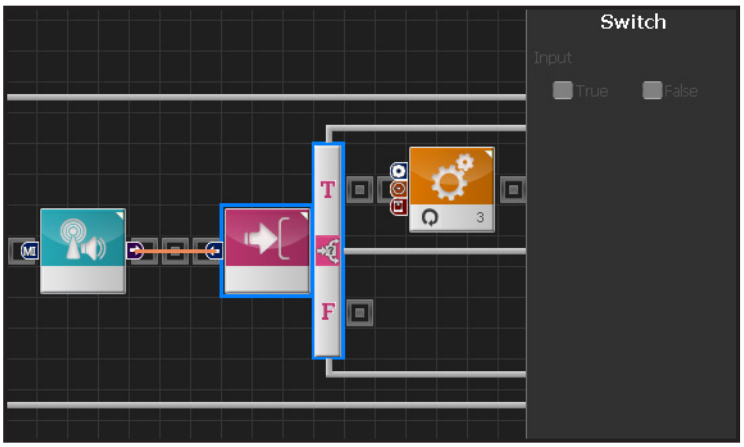
로봇의 오른쪽 소리에 따라서 로봇이 정확히 동작한다는 것을 보여주기 위해서 외부의 자극을 의도적으로 막는 프로그래밍을 해봤습니다.
이러한 프로그래밍은 단 한번의 박수로 단 한번의 손을 들게 하여서 정확도를 높일 수가 있습니다.



23 Sound Sensor (두번째)

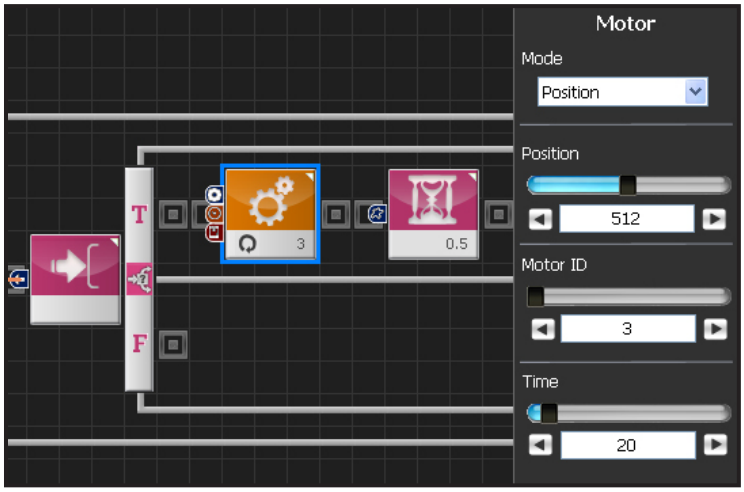
두번째 사운드 센서를 설정합니다. 왼쪽 손이 반응합니다.

Sensor 〉 Sound Sensor 모듈을 선택합니다.
Compare : 〈 로 설정합니다.
Value 값은 –1 입니다. –1보다 작을 때 반응합니다



24 Switch IF 분기문

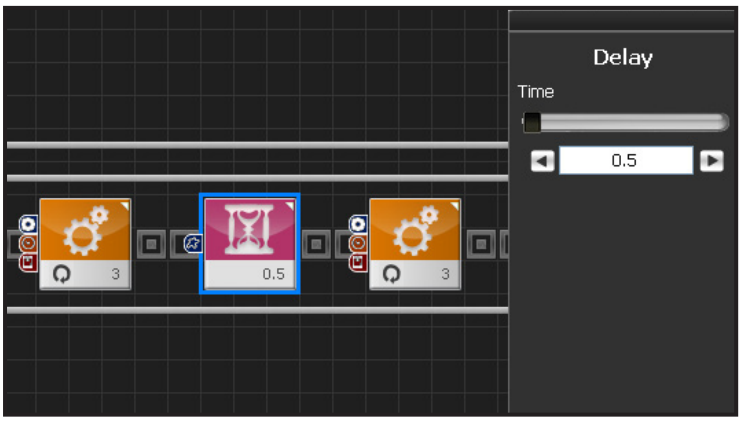
True 일 때 동작합니다.



25 모터 3번(어깨) 설정

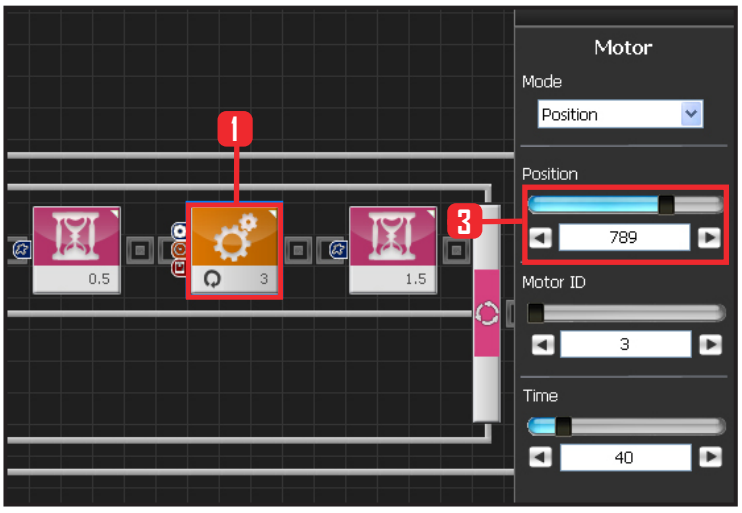
소리 방향값이 –1보다 작으면 True 로 왼팔을 양팔벌려 자세로 올립니다.

Motion〉Moter 를 선택합니다.
Mode : Position 으로 선택합니다.
Position 은 512 로 설정합니다.
왼팔 차려자세일 때 어깨 각도는 789 입니다. 789에서 512로 이동하면 양팔 벌려 자세가 됩니다. 시간은 20으로 설정합니다.



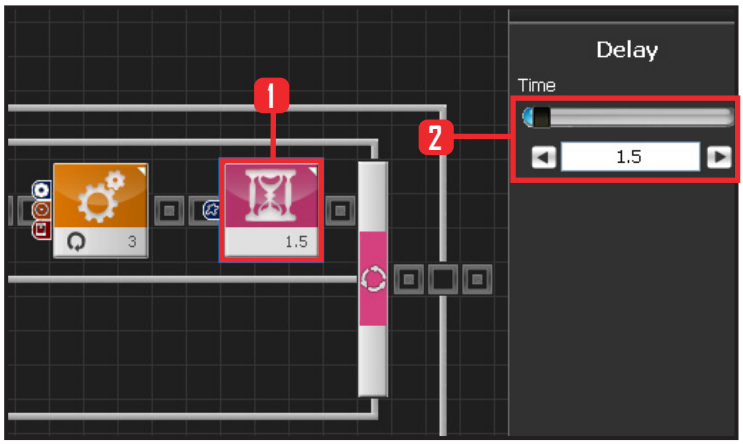
26 Delay

오른팔과 마찬가지로 외부의 자극(소리) 인식을 막기 위해 0.5초동안 Delay 값을 줍니다.



27 모터 3번(어깨) 차려자세

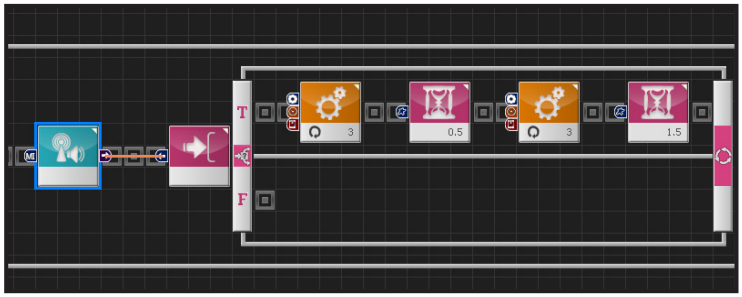
3번 모터 Position 을 789로 설정하여 다시 차려자세로 돌립니다.



28 Delay

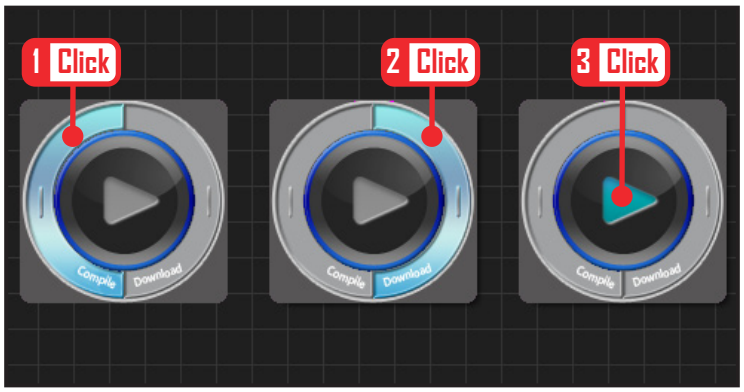
1.5초동안 Delay 값을 줘서, 다른 반응이 일어나지 않도록 합니다.

앞 0번 모터에는 Continue 가 있는데, 뒤의 3번 모터에는 없는 이유는 Loop 의 끝이기 때문에 자동으로 처음으로 돌아가기 때문입니다.



29 왼팔 반응

왼쪽에서 박수를 쳤을 때 일정한 간격으로 팔을 팔벌려 자세로 올렸다 내립니다.
박수를 연속으로 두 번을 쳐도 로봇은 한 번만 반응합니다. Delay 값을 줬기 때문입니다.
이러한 프로그래밍은 주변이 시끄럽거나 좌우 구분이 어려워서 로봇의 양쪽 팔을 혼란스럽게 움직일 때 유용하게 사용될 수 있습니다.



30 컴파일, 다운로드, 실행

왼쪽 클릭하여 컴파일 시킵니다. 에러가 없으면 오른쪽 클릭하여 로봇에 다운로드 시킵니다. 다운로드 완료되면 가운데 화살표 실행버튼을 눌러 로봇에서 실행시킵니다.

31 로봇동작

로봇 오른쪽에서 박수치면 오른쪽 팔을 올리고, 로봇 왼쪽에서 박수치면 왼쪽 팔을 올립니다.

1

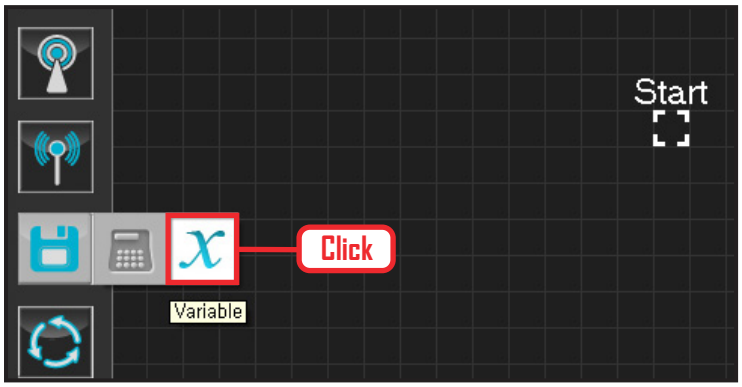
2

Digital Distance Sensor 예제 따라하기

예제설명

아날로그 거리센서는 거리별로 모두 측정이 가능하지만, 디지털 거리센서는 특정 거리를 기준으로 가깝냐 머냐 만을 판단합니다. 바퀴가 있는 로봇은 벽감지보다는 낭떠러지 감지용으로 많이 활용됩니다. 휴머노이드에서는 발 자체가 움직이기 때문에 낭떠러지 감지보다는 벽감지 프로그래밍을 해봅니다. 아날로그 거리센서와 비교해보기 바랍니다.

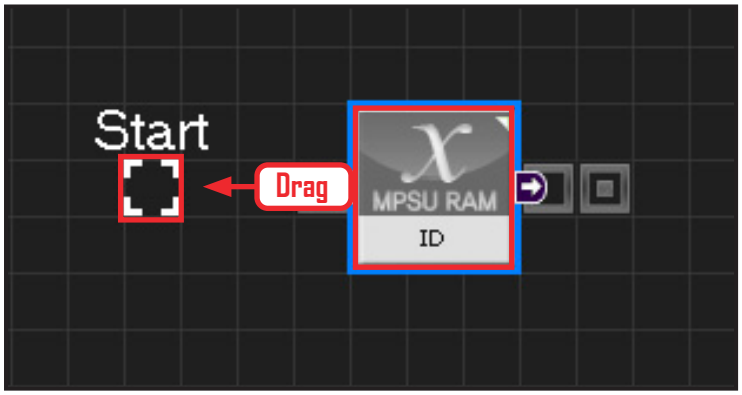
벽이 일정거리 가까워지면 뒷걸음질 하다가 방향을 틀고 전진하는 프로그래밍입니다. 이 예제를 실행하려면 ADC포트 1번 (좌측)에 디지털 거리 센서를 장착한 상태여야 합니다.



01 변수 지정

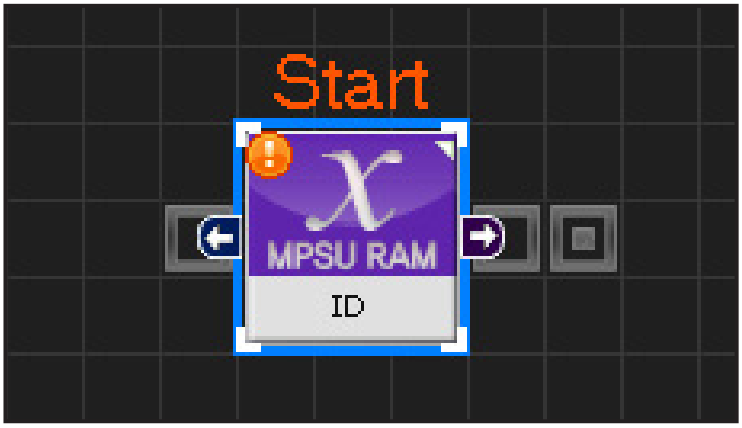
로봇을 동작시킨다는 것은 로봇의 서보 모터를 동작시킨다는 의미입니다. 서보가 스스로 움직일 수 있는 상태로 값을 지정해주어야 합니다.

Data 〉 Variable 모듈을 클릭합니다.



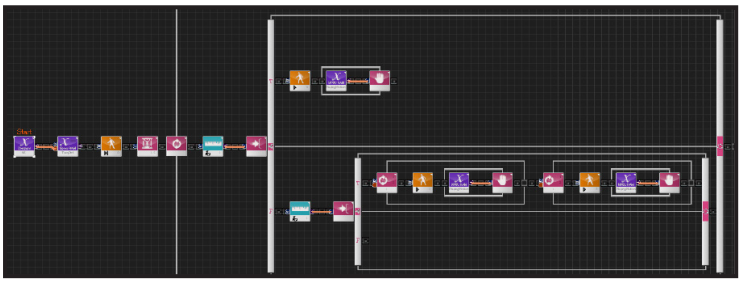
02 시작

모듈의 왼쪽 연결선을 Start Point 에 드래그하여 정확히 도킹을 시킵니다.



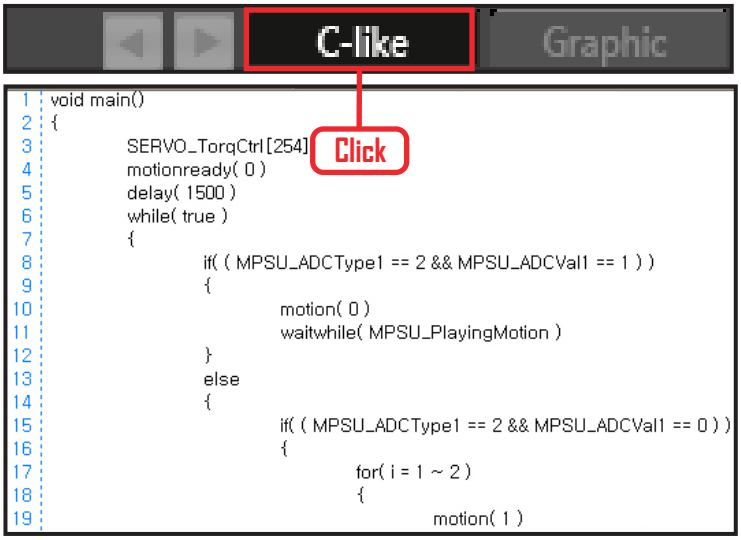
03 프로그래밍 시작

모듈과 Start Point 가 정확히 도킹하면 왼쪽과 같이 활성화된 칼라 이미지 모듈로 변합니다.
그럼 프로그래밍이 시작되었다는 의미입니다.



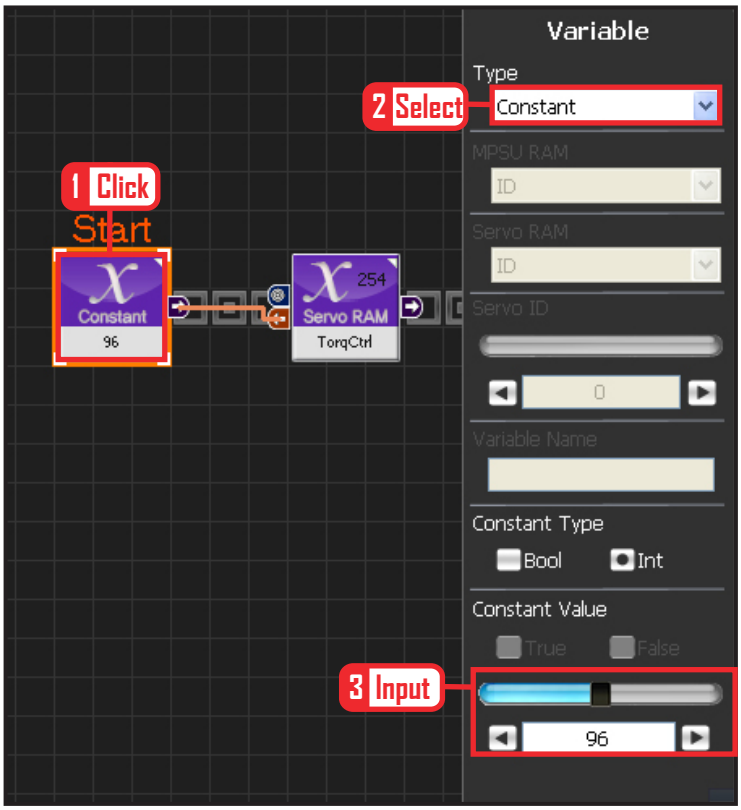
04 전체 프로그래밍

디지털 거리센서를 이용한 전체 프로그래밍입니다.



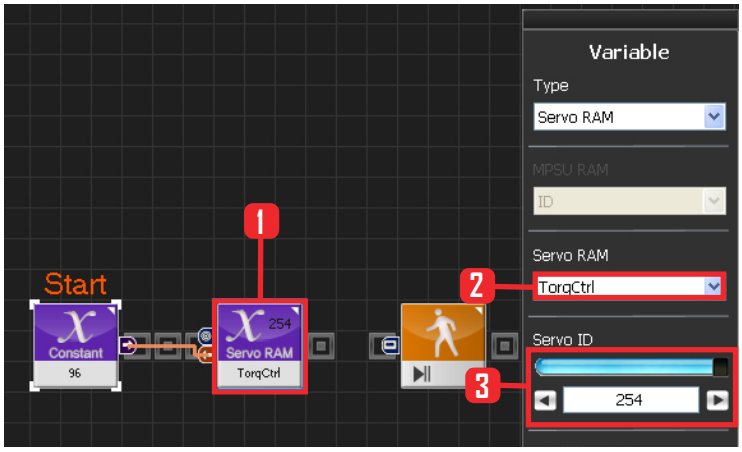
05 C-Like 보기

오른쪽 상단의 Graphic 탭에서 C-like 탭을 클릭하면 왼쪽과 같은 Task 프로그래밍 화면이 나옵니다.
디지털 거리센서를 이용한 전체 프로그래밍 화면입니다.
C와 유사한 문법 구조를 가지고 있으므로 C 문법 선행학습 효과도 있습니다.
각 모듈별로 클릭하면 커서가 따라서 움직이므로 모듈별로 Text로 어떻게 변환하는지 확인할 수 있습니다.



06 상수 설정

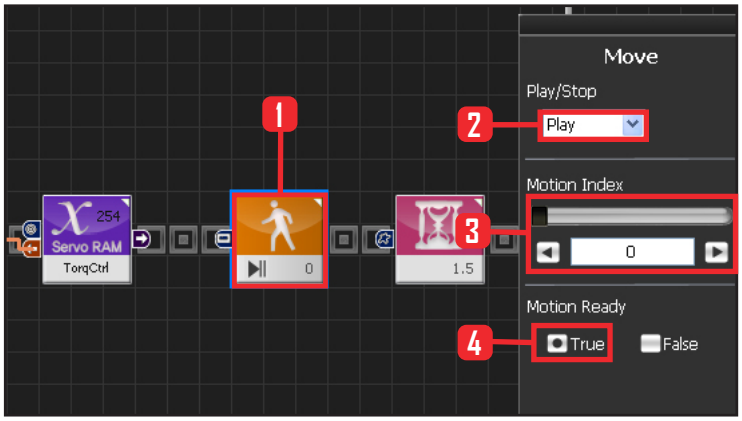
서보 모터를 스스로 움직일 수 있는 상태로 만드는 과정입니다.
Variable Type 을 Constant 로 선택합니다.
속성중에 Constant Value 값을 96 으로 설정합니다.
서보의 TorqControl 레지스터에 96(0x60) 이라는 값이 들어가면 서보가 움직일 수 있는 상태가 됩니다.
그 값은 Output 커넥터를 통하여 뒤 모듈의 토크값에 전달합니다.



07 모든 서보에 적용

앞에서 받은 96 상수값을 모든 서보에 적용하는 과정입니다.

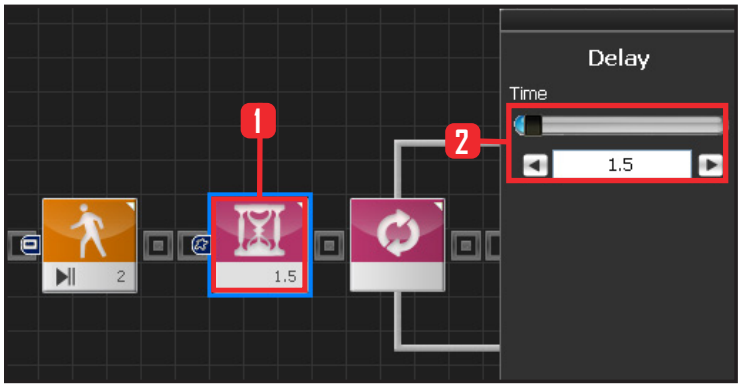
Variable 〉 Type : Servo RAM을 선택합니다.
Servo RAM : TorqCtrl 을 선택합니다.
Servo ID : 254 를 선택합니다. 254는 연결되어있는 모든 서보에 적용하겠다는 의미입니다.



08 모션 동작준비

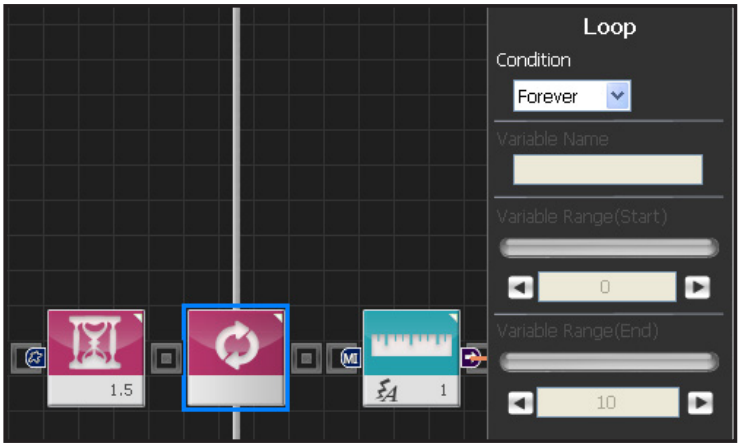
모션이 동작하기 위해서는 준비과정을 거칩니다. 이전 로봇 모션에서 갑작스럽게 변동하면 로봇에 무리가 가해질 수 있습니다. 따라서 현재 실행하고자 하는 로봇의 첫 모션으로 천천히 이동시키는 과정입니다. Motion Ready 가 True 이면 모션 첫 장면을 준비하는 것이고, False 이면 모션이 동작합니다.

Motion 〉 Move 모듈을 선택합니다.
Play/Stop : Play 를 선택합니다.
Motion Index : 0 번을 선택합니다. 0번은 전진하면서 걷기 모션입니다.
Motion Ready : True 를 선택합니다.
모션 동작 준비 과정입니다.



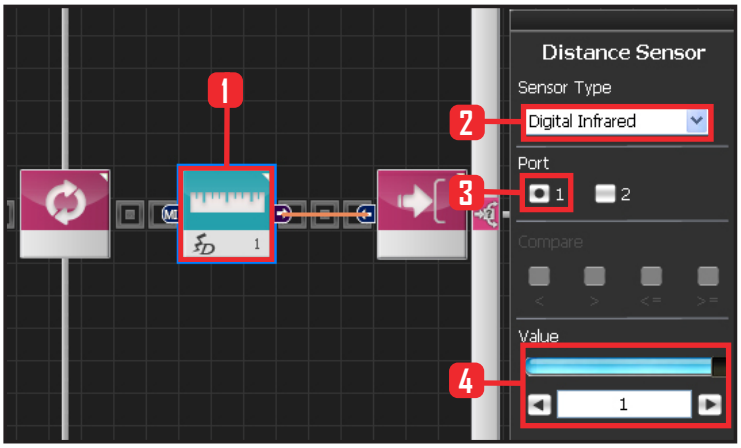
09 Delay

Motion Ready 동작이 끝나기 전에 진행하는 것을 방지하기위해 Delay 값을 1.5 초로 설정합니다.



10 반복

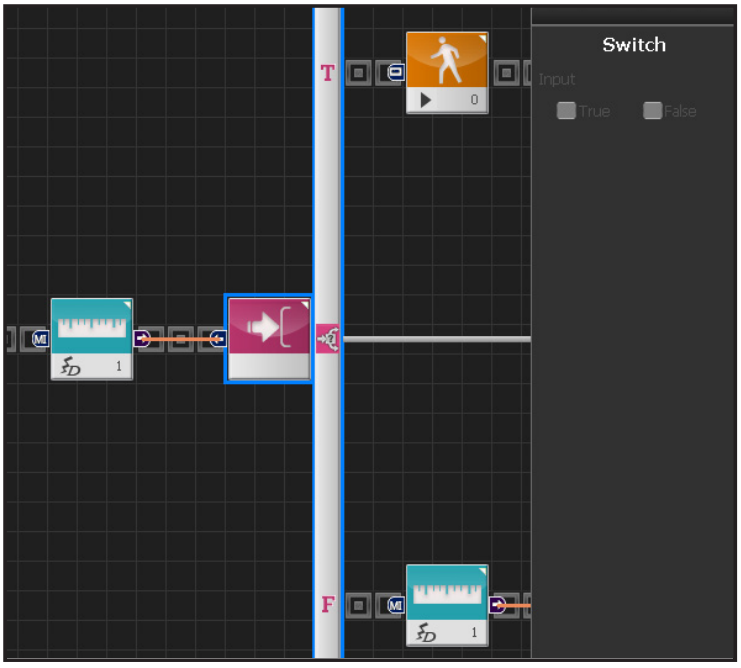
Forever 무한 반복을 설정합니다.



11 디지털 거리센서 설정

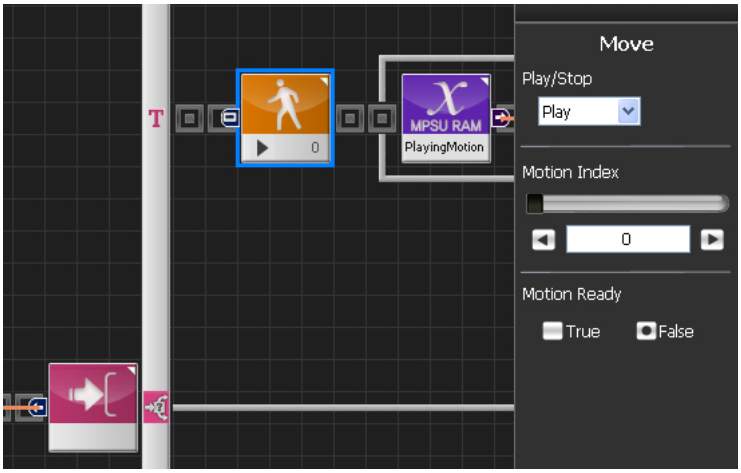
디지털 거리센서는 제품마다 측정 거리가 다릅니다. 여기서는 20cm 기준으로 설정됩니다.

Sensor〉Distance Sensor 모듈을 선택합니다.
Sensor Type : Digital Infrared 를 선택합니다.
Port : 1번을 선택합니다.
Value : 1 을 선택합니다. 10cm 보다 멀리 있다는 것을 의미합니다.



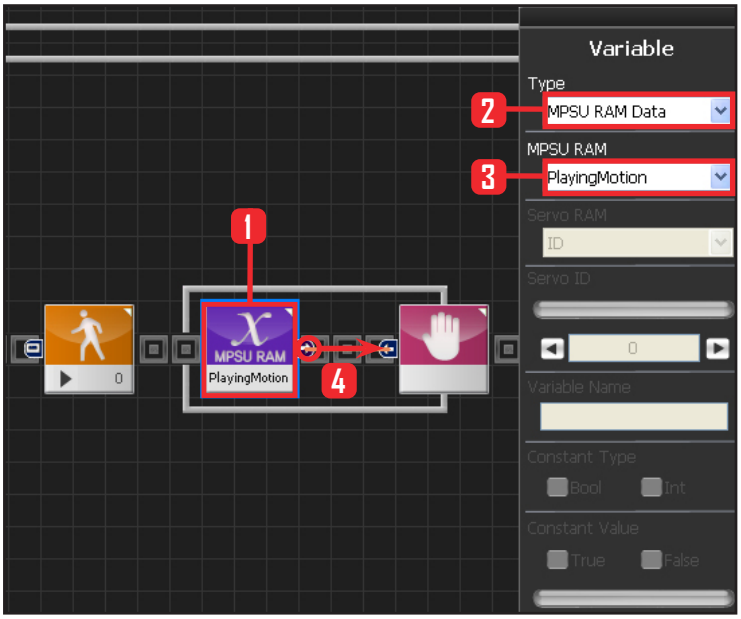
12 If 분기문

True 일때는 전진하고, False 일때는 다음 조건문으로 들어갑니다.



13 전진하기

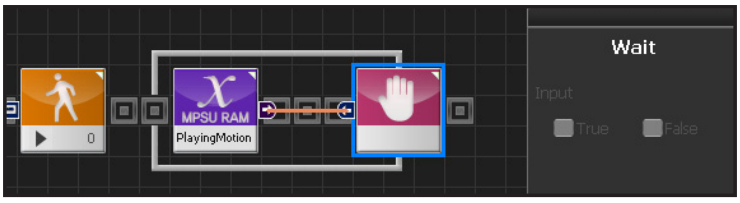
10cm 보다 먼거리이기 때문에 로봇은 전진합니다.
Motion Ready 값을 False 로 선택하면 로봇은 전진 모션을 실행합니다.



14 모션 동작 확인

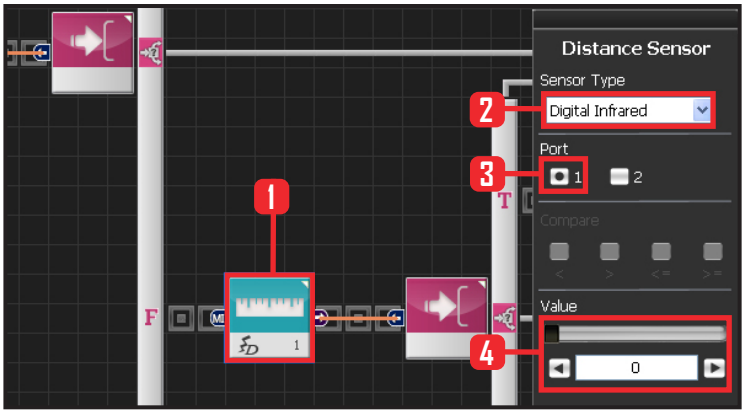
Loop 는 계속적인 반복을 의미합니다. Move 명령을 내리고 나서 실제 모션이 실행되어 완료되기 까지는 시간이 걸리므로 loop안에 Move모듈 하나만을 넣고 실행하면 모션을 이미 실행중임에도 loop를 계속 돌면서 모션실행 명령을 반복하게 됩니다.
이렇게 되면 Move모듈을 만난 횟수와 실제모션을 실행한 횟수가 달라집니다.
따라서 실행한 모션이 끝날때까지 기다렸다가 다시 loop의 처음으로 돌아가게 하는 편이 더 정확합니다.
Variable 〉 MPSU RAM Data 에 들어가면 Playing Motion 이 있습니다. Playing Motion 은 로봇이 모션을 실행중인지 확인하는 변수입니다. 그 Playing Motion에 Wait 를 걸어주면 로봇의 동작이 끝날 때까지 Loop는 기다려줍니다.

Data 〉 Variable 모듈을 선택합니다.
Type : MPSU RAM Data 를 선택합니다.
MPSU RAM : Playing Motion 을 선택합니다.
Output 커넥터 값을 뒤에 Wait 모듈에 연결합니다.
모션이 끝날 때까지 기다리겠다는 의미입니다



15 Wait

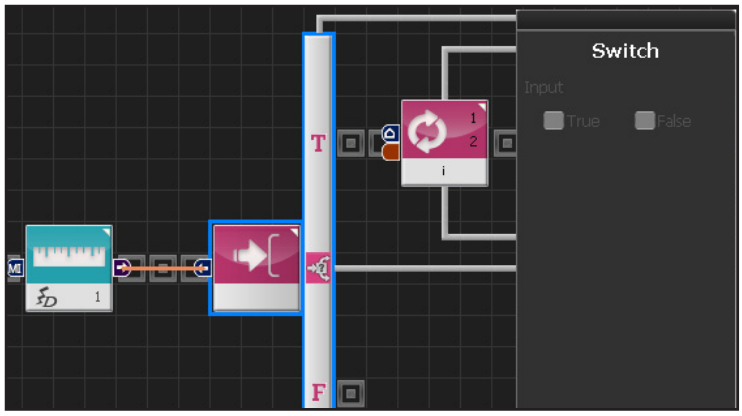
모션이 끝날때까지 기다립니다.
모션이 끝나면 다시 처음으로 돌아가 모션을 반복합니다.



16 벽 근접시 동작

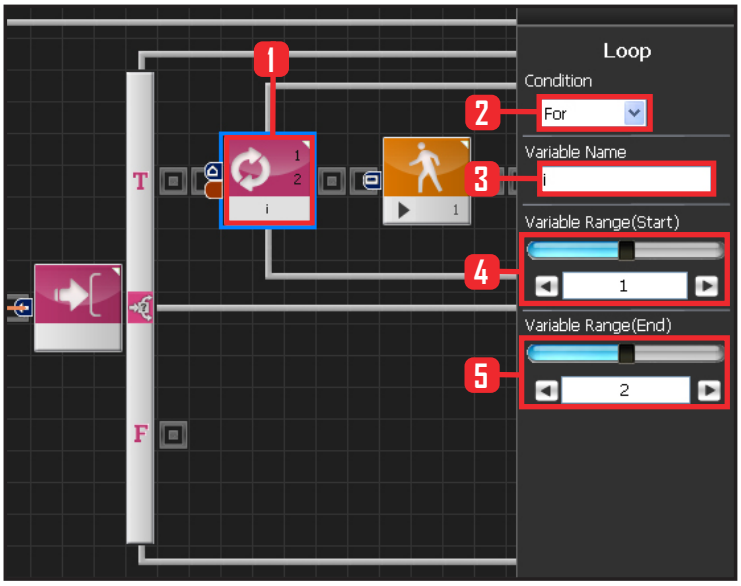
로봇이 벽과 10cm 이내로 근접시에는 로봇을 후진보행 시켰다가 우회전 시키는 프로그래밍입니다.

Sensor 〉 Distance Sensor 모듈을 선택합니다.
Sensor Type : Digital Infrared 를 선택합니다.
Port : 1번을 선택합니다.
Value : 0 을 선택합니다. 10cm 안에 들어왔다는 의미입니다.



17 If 분기문

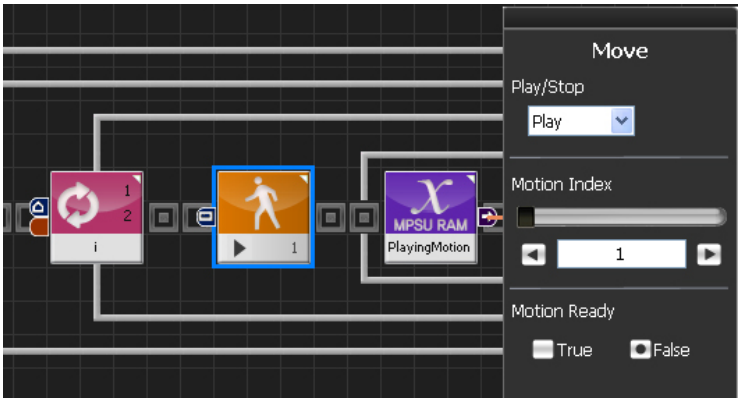
10cm 보다 더 가까울 때 True 안의 문을 실행시킵니다.



18 For 반복

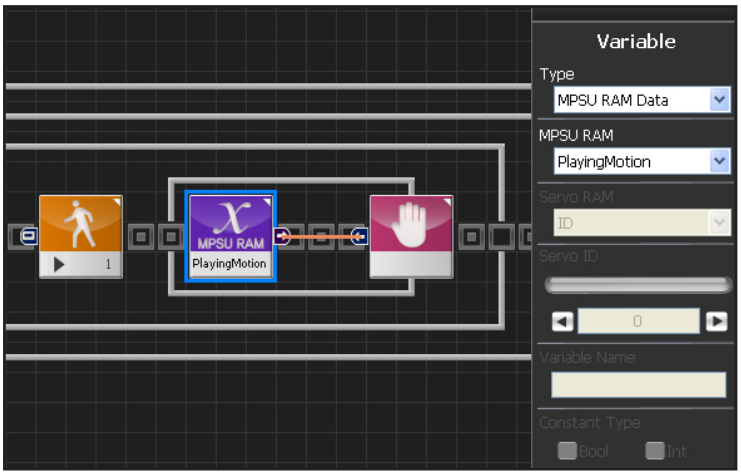
로봇의 모션은 같은 동작을 일정하게 반복하여 원하는 모션까지의 동작을 취하게 할 수 있습니다.
모션 1번은 로봇이 후진하는 모션입니다. 후진은 왼발 오른발 한번씩만 뒤로 이동하게 되어있습니다.
그 모션에 원하는 수만큼 반복을 줄 수 있는 For 문을 써서, 원하는 위치만큼 이동시킬 수 있습니다.

Flow 〉 Loop 모듈을 선택합니다.
Condition : For 를 선택합니다.
Valuable Name 은 i 로 입력합니다.
Valuable Range(Start) 1 로 입력합니다.
Valuable Range(End) 2 로 입력합니다.
모션을 두번을 반복하여 실행시킨다는 의미입니다.



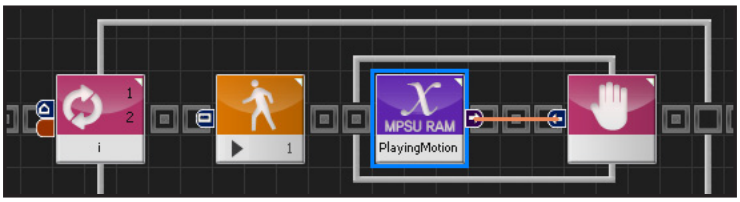
19 뒤로 물러서기

1번은 뒤로 물러서는 모션입니다.
False 를 선택하면 로봇이 후진모션을 실행합니다.



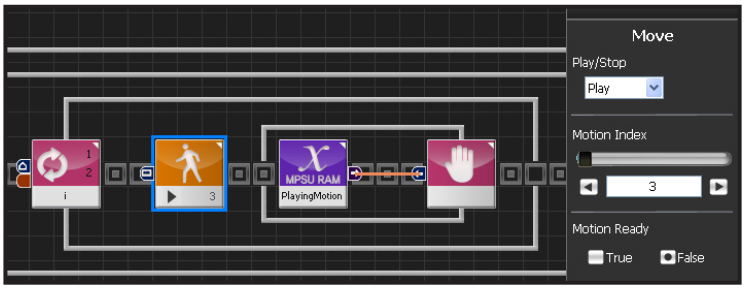
20 동작 감지

Playing Motion 으로 로봇의 동작을 감지하고, 끝나면 다시 for 문의 처음으로 돌아갑니다.



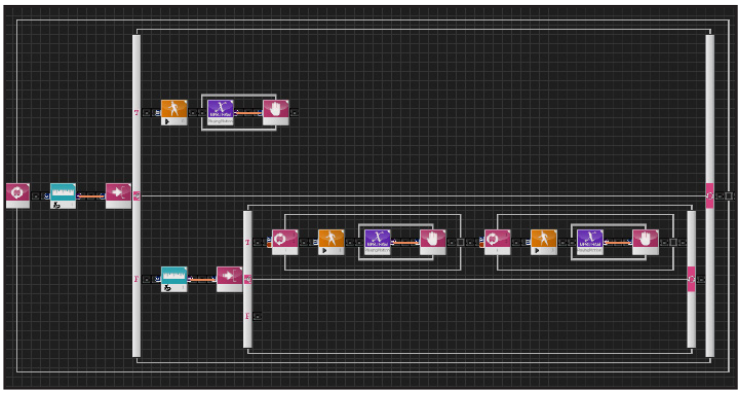
21 두번 반복 후진

로봇을 두번 반복해서 후진 시키는 프로그래밍이였습니다.



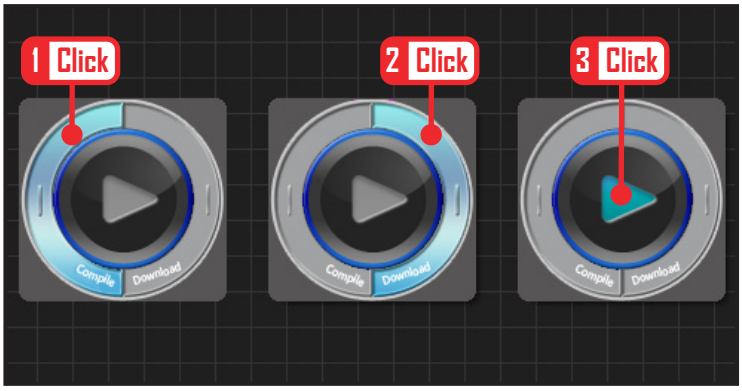
22 우회전 시키기

로봇 모션 3 번은 제자리에서 좌회전 하는 동작입니다. 우회전 동작도 마찬가지로 For 문을 써서 원하는 만큼 위치를 조절할 수 있습니다.
Motion 은 3번을 선택하고 For문은 1~3 번까지 설정하여 위와 같이 프로그래밍 합니다.



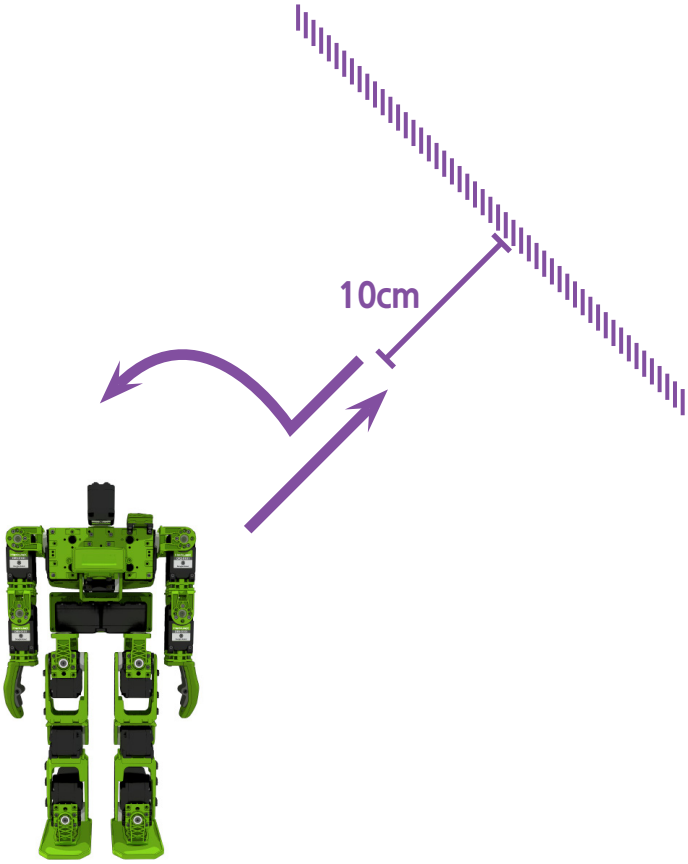
23 전체 살펴보기

10cm 보다 멀때는 로봇을 그대로 전진시키고, 10cm 보다 가까울 때 에는 For 문으로 후진과 우회전 모션을 일정하게 반복시켜 회피하는 프로그래밍입니다.



24 컴파일, 다운로드, 실행

왼쪽 클릭하여 컴파일 시킵니다. 에러가 없으면 오른쪽 클릭하여 로봇에 다운로드 시킵니다. 다운로드 완료되면 가운데 화살표 실행버튼을 눌러 로봇에서 실행시킵니다.



25 로봇동작

로봇이 10cm 앞의 벽을 감지하면 후진하였다가 우회전 한 후 전진합니다.

소프트웨어 : DR-Visual Logic

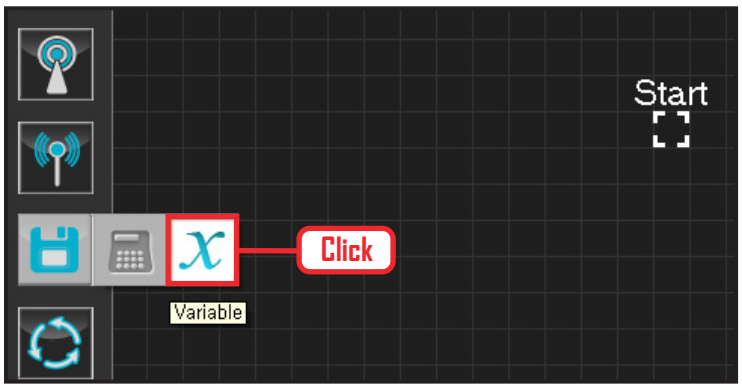
모듈별 프로그래밍 : 센서(Sensor) 〉 Analog Distance

Analog Distance Sensor 예제 따라하기

예제설명

아날로그 거리센서를 이용하여, 벽이 가까워지면 좌회전 하면서 벽을 회피하는 프로그래밍입니다

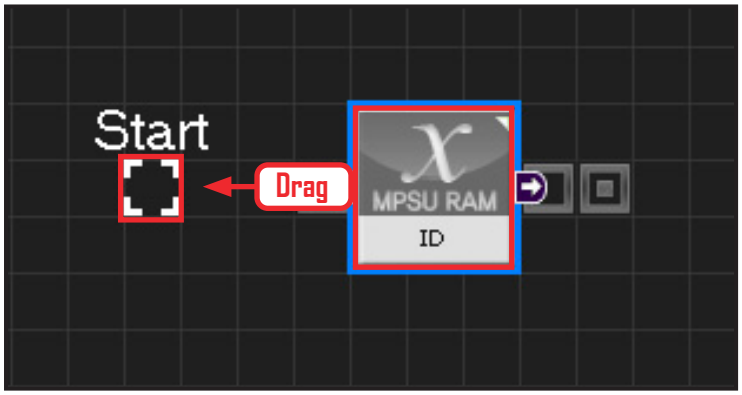
거리센서에는 아날로그 센서와 디지털 센서 가 있습니다. 디지털 센서는 일정거리(10cm) 를 기준으로 안쪽과 바깥쪽 두 영역만 감지하는 반면, 아날로그 센서는 거리별로(6~40cm) 위치를 감지할 수 있습니다. 이 예제를 실행하려면 ADC 포트 1번 (좌측)에 PSD센서를 장착한 상태여야 합니다.



01 변수 지정

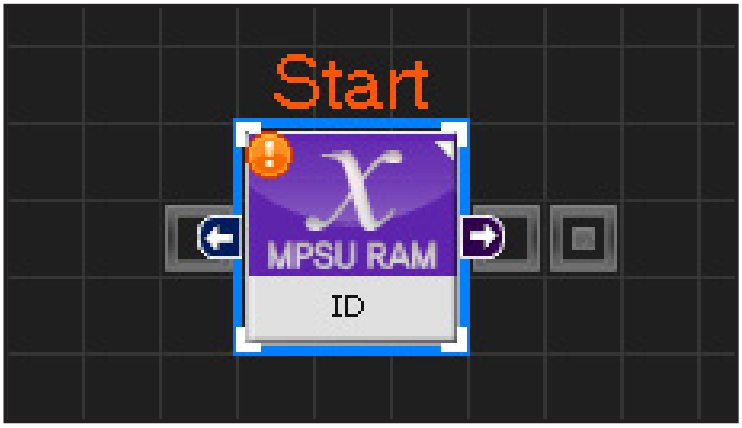
로봇을 동작시킨다는 것은 로봇의 서보 모터를 동작시킨다는 의미입니다. 서보가 스스로 움직일 수 있는 상태로 값을 지정해주어야 합니다.

Data 〉 Variable 모듈을 클릭합니다.



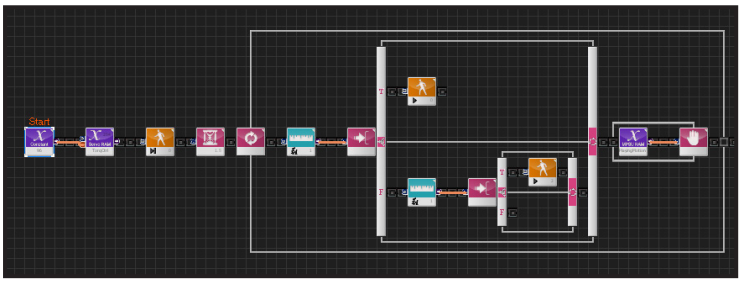
02 시작

모듈의 왼쪽 연결선을 Start Point 에 드래그하여 정확히 도킹을 시킵니다.



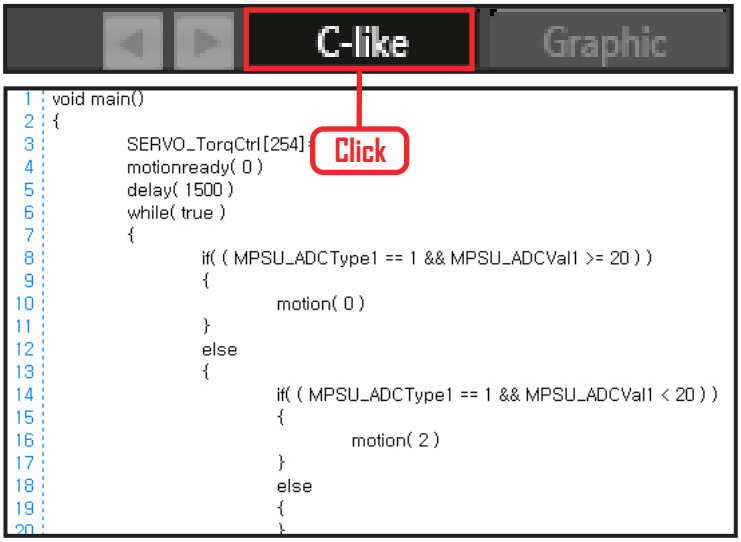
03 프로그래밍 시작

모듈과 Start Point 가 정확히 도킹하면 왼쪽과 같이 활성화된 칼라 이미지 모듈로 변합니다.
그럼 프로그래밍이 시작되었다는 의미입니다.



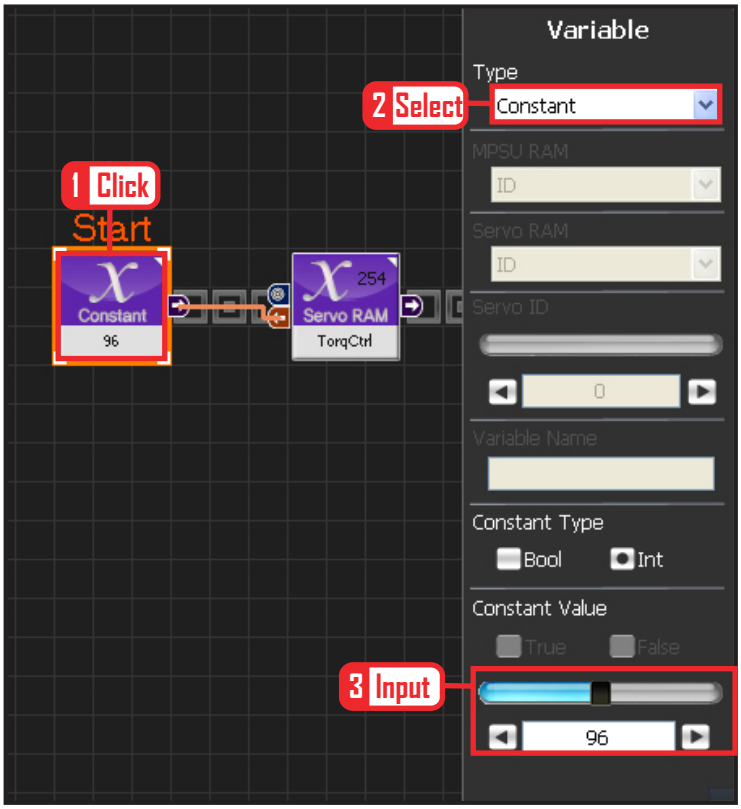
04 전체 프로그래밍

아날로그 거리센서와 전진걷기 모션, 좌로돌기 모션을 이용하여 전체 프로그래밍해봅니다.



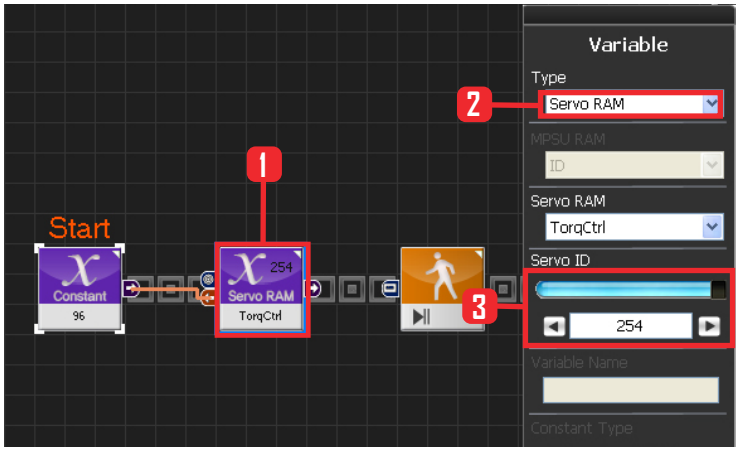
05 C-Like 보기

오른쪽 상단의 Graphic 탭에서 C-like 탭을 클릭하면 왼쪽과 같은 Task 프로그래밍 화면이 나옵니다.
아날로그 거리센서를 이용한 전체 프로그래밍 화면입니다.
C와 유사한 문법 구조를 가지고 있으므로 C 문법 선행학습 효과도 있습니다.
각 모듈별로 클릭하면 커서가 따라서 움직이므로 모듈별로 Text로 어떻게 변환하는지 확인할 수 있습니다.



06 상수 설정

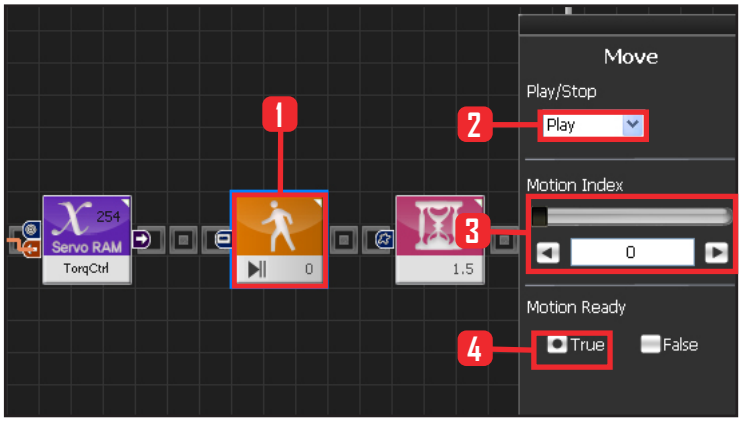
서보 모터를 스스로 움직일 수 있는 상태로 만드는 과정입니다.
Variable Type 을 Constant 로 선택합니다.
속성중에 Constant Value 값을 96 으로 설정합니다.
서보의 TorqControl 레지스터에 96(0x60) 이라는 값이 들어가면 서보가 움직일 수 있는 상태가 됩니다.
그 값은 Output 커넥터를 통하여 뒤 모듈의 토크값에 전달합니다.



07 모든 서보에 적용

앞에서 받은 96 상수값을 모든 서보에 적용하는 과정입니다.

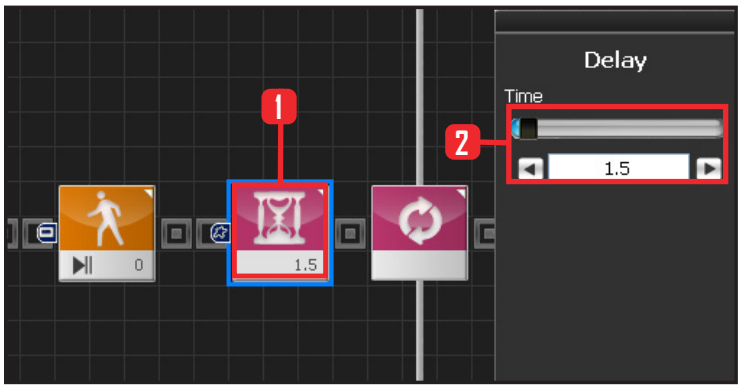
Variable 〉 Type : Servo RAM을 선택합니다.
Servo RAM : TorqCtrl 을 선택합니다.
Servo ID : 254 를 선택합니다. 254는 연결되어있는 모든 서보에 적용하겠다는 의미입니다.



08 모션 동작준비

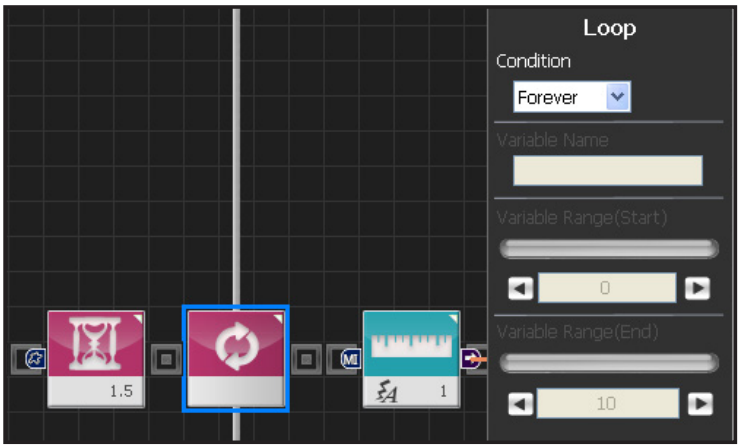
모션이 동작하기 위해서는 준비과정을 거칩니다. 이전 로봇 모션에서 갑작스럽게 변동하면 로봇에 무리가 가해질 수 있습니다. 따라서 현재 실행하고자 하는 로봇의 첫 모션으로 천천히 이동시키는 과정입니다. Motion Ready 가 True 이면 모션 첫 장면을 준비하는 것이고, False 이면 모션이 동작합니다.

Motion 〉 Move 모듈을 선택합니다.
Play/Stop : Play 를 선택합니다.
Motion Index : 0 번을 선택합니다. 0번은 전진하면서 걷기 모션입니다.
Motion Ready : True 를 선택합니다.
모션 동작 준비 과정입니다.



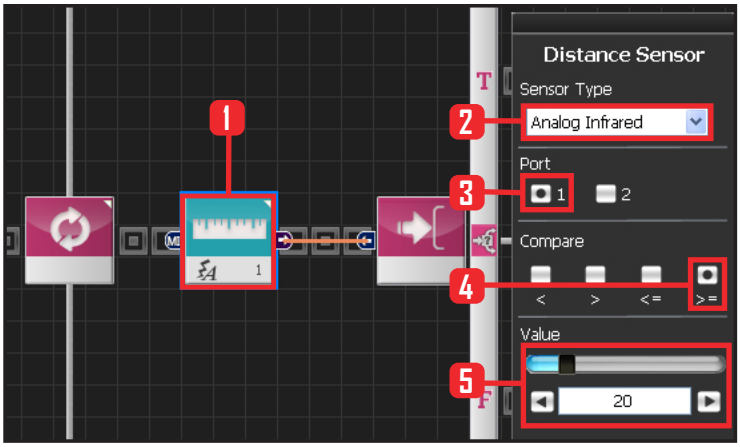
09 Delay

Motion Ready 동작이 끝나기 전에 진행하는 것을 방지하기위해 Delay 값을 1.5 초로 설정합니다.



10 반복

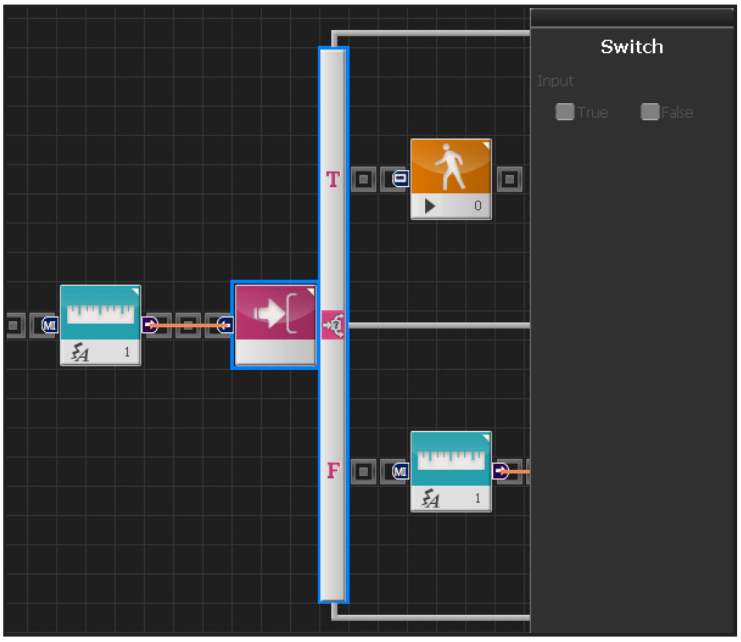
Forever 무한 반복을 설정합니다.



11 아날로그 센서 설정

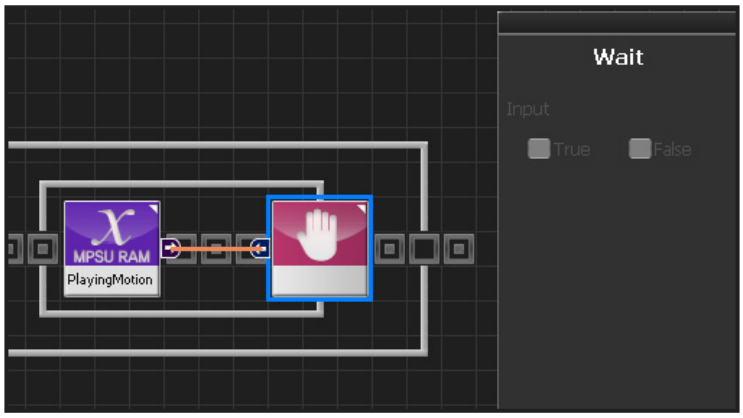
20cm 기준으로 설정합니다.

Sensor 〉 Distance Sensor 모듈을 선택합니다.
Sensor Type : Analog Infrared 를 선택합니다.
Port : 1번을 선택합니다.
Compare : 〉= 를 선택합니다. 기준보다 같거나 높았을 때 True 입니다.
Value : 20 을 선택합니다. 20cm 를 의미합니다.



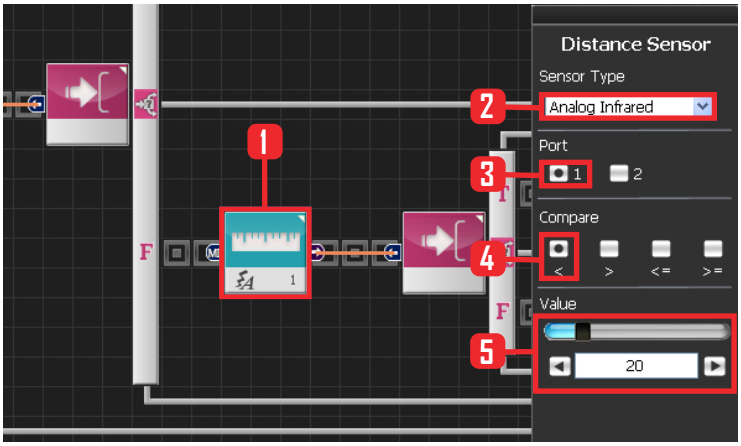
12 If 분기문

True 일때는 전진하고, False 일때는 다음 조건문으로 들어갑니다.



13 전진하기

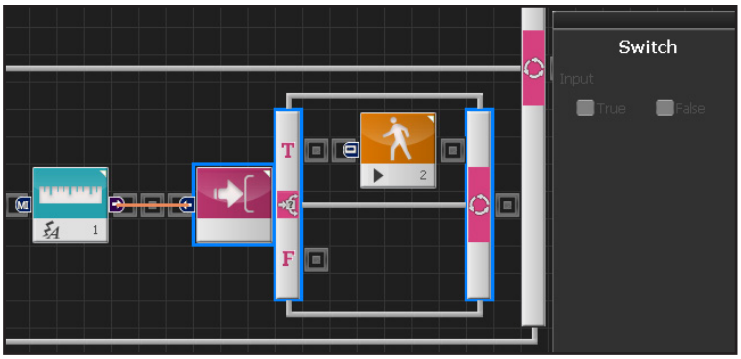
20cm 보다 먼거리이기 때문에 로봇은 전진합니다.
Motion Ready 값을 False 로 선택하면 로봇은 전진 모션을 실행합니다.



14 벽 근접시 동작

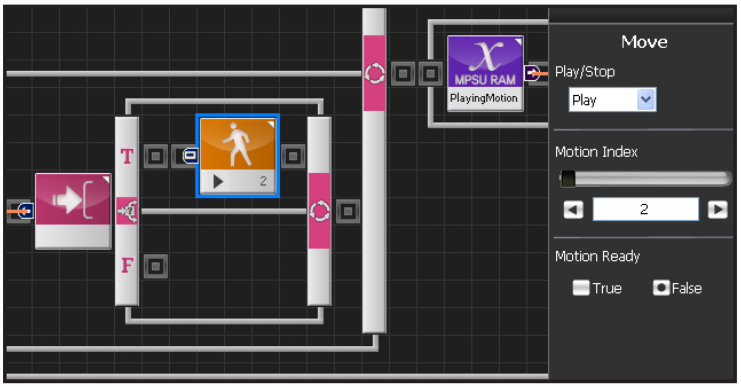
로봇이 벽과 20cm 이내로 근접시에는 좌회전 시킵니다.

Sensor 〉 Distance Sensor 모듈을 선택합니다.
Sensor Type : Analog Infrared 를 선택합니다.
Port : 1번을 선택합니다.
Compare : 〈를 선택합니다. 기준보다 낮을 때 True 입니다.
Value : 20 을 선택합니다. 20cm 를 의미합니다.



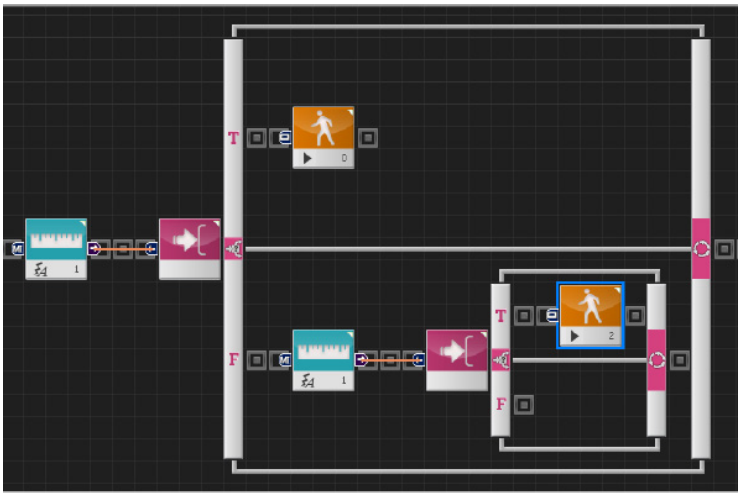
15 If 분기문

20cm 보다 더 가까울 때 True 안의 문을 실행시킵니다.



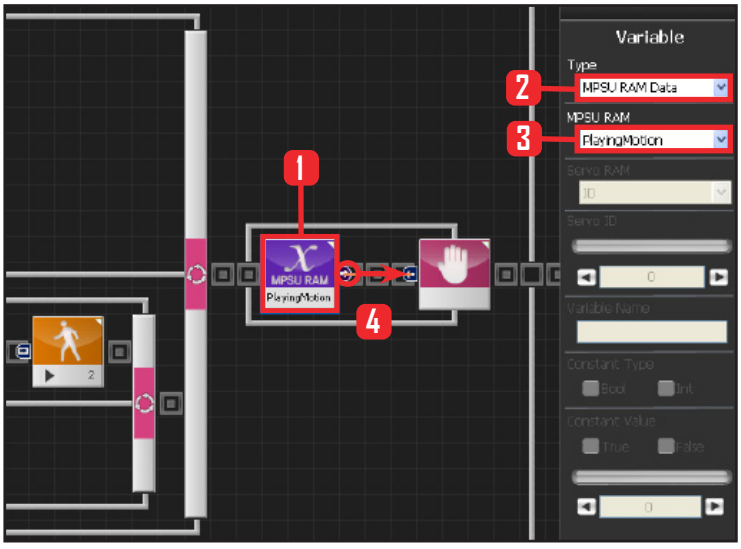
16 좌회전 시키기

Motion 2번은 좌회전 모션입니다.
Motion Ready 값이 False 이면 좌회전 모션을 실행합니다.



17 거리별 동작

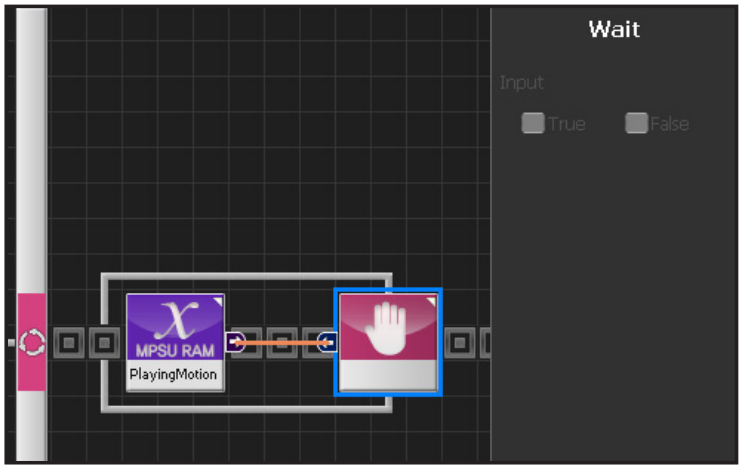
20cm 보다 멀면 로봇이 계속 앞으로 전진하고, 20cm 보다 작으면 로봇이 좌회전 합니다.



18 모션 동작 확인

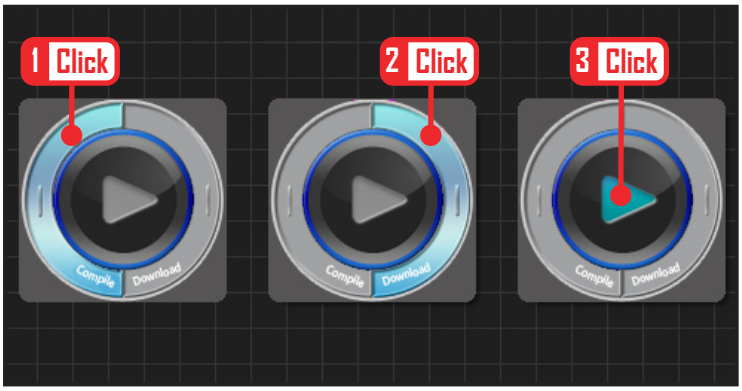
Loop 는 계속적인 반복을 의미합니다. Move 명령을 내리고 나서 실제 모션이 실행되어 완료되기 까지는 시간이 걸리므로 loop안에 Move모듈 하나만을 넣고 실행하면 모션을 이미 실행중임에도 loop를 계속 돌면서 모션실행 명령을 반복하게 됩니다.
이렇게 되면 Move모듈을 만난 횟수와 실제모션을 실행한 횟수가 달라집니다.
따라서 실행한 모션이 끝날때까지 기다렸다가 다시 loop의 처음으로 돌아가게 하는 편이 더 정확합니다.
Variable 〉 MPSU RAM Data 에 들어가면 Playing Motion 이 있습니다. Playing Motion 은 로봇이 모션을 실행중인지 확인하는 변수입니다. 그 Playing Motion에 Wait 를 걸어주면 로봇의 동작이 끝날 때까지 Loop는 기다려줍니다.

Data 〉 Variable 모듈을 선택합니다.
Type : MPSU RAM Data 를 선택합니다.
MPSU RAM : Playing Motion 을 선택합니다.
Output 커넥터 값을 뒤에 Wait 모듈에 연결합니다.
모션이 끝날 때까지 기다리겠다는 의미입니다.



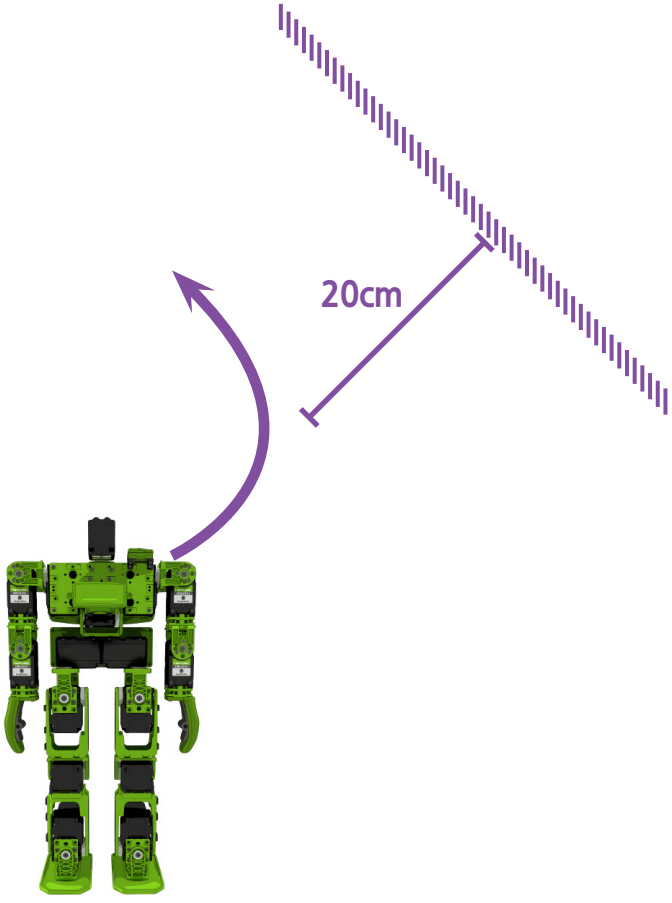
19 Wait

모션이 끝날때까지 기다립니다.
모션이 끝나면 다시 처음으로 돌아가 모션을 반복합니다.



20 컴파일, 다운로드, 실행

왼쪽 클릭하여 컴파일 시킵니다. 에러가 없으면 오른쪽 클릭하여 로봇에 다운로드 시킵니다. 다운로드 완료되면 가운데 화살표 실행버튼을 눌러 로봇에서 실행시킵니다.



21 로봇동작

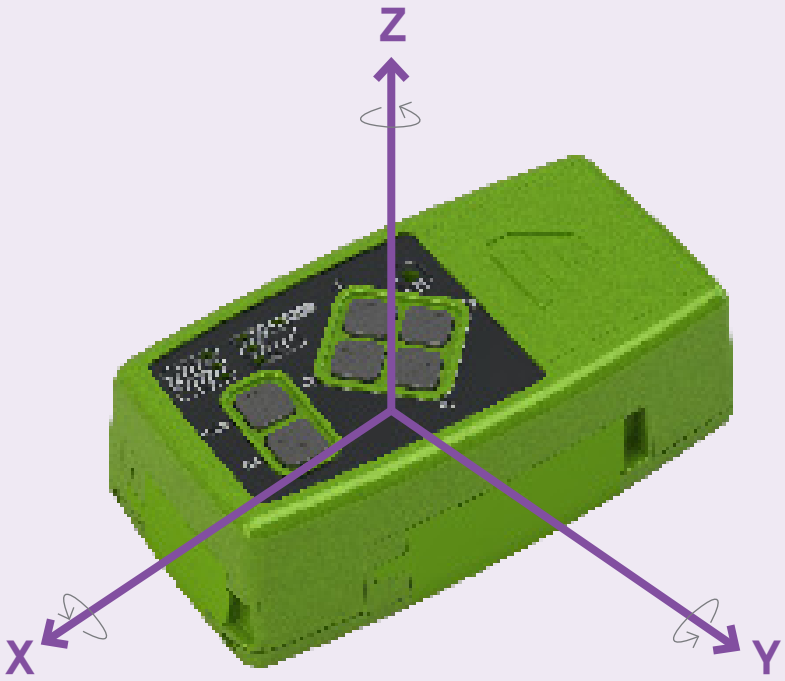
로봇이 벽을 향해 전진하다가 20cm 안으로 근접하면 좌회전 합니다.

Acceleration (가속도) 예제 따라하기

예제설명

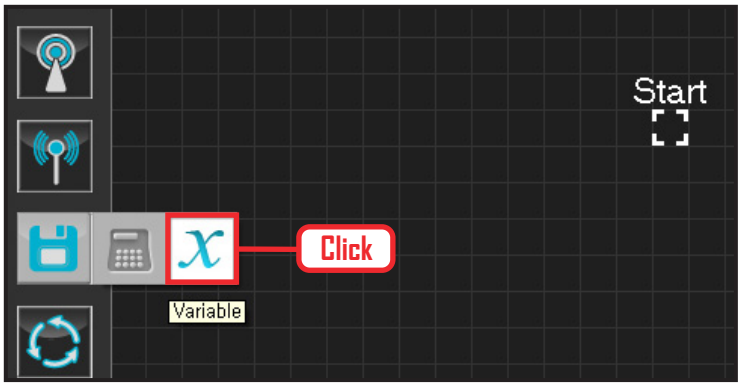
Acceleration (가속도) 센서를 이용하여 로봇이 각각 앞으로 넘어졌을 때와 뒤로 넘어졌을 때 일어나는 프로그래밍을 해봅니다.

가속도 센서는 제어기 뒷 커버를 열고 넣을 수 있는 모듈 형태로서 Gyro센서 모듈과 결합되어 있습니다. (Z축 표시되어있는 그림 삽입)



- 로봇이 엎드릴 때 Z축은 "-" 가속도가 붙고 그 값은 약 -4096 입니다.
- 로봇이 누을때 Z축은 "+" 가속도가 붙고 그 값은 약 4096 입니다.

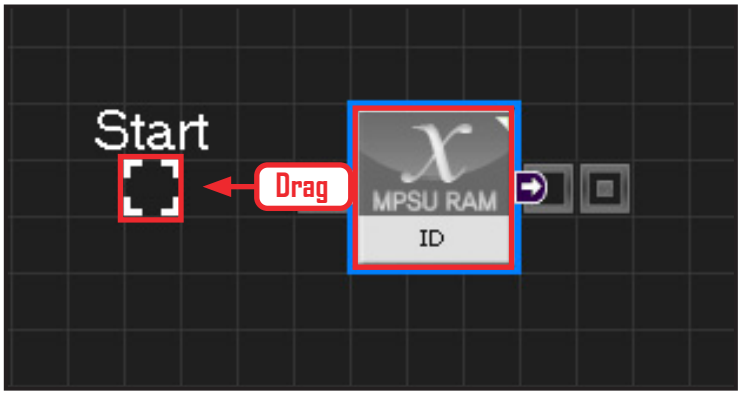
  (4096 은 약 1g 중력값을 나타냅니다.)



01 변수 지정

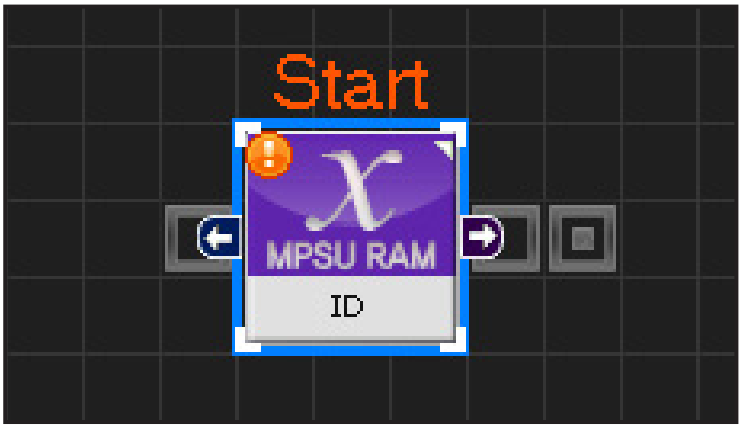
로봇을 동작시킨다는 것은 로봇의 서보 모터를 동작시킨다는 의미입니다. 서보가 스스로 움직일 수 있는 상태로 값을 지정해주어야 합니다.

Data 〉 Variable 모듈을 클릭합니다.



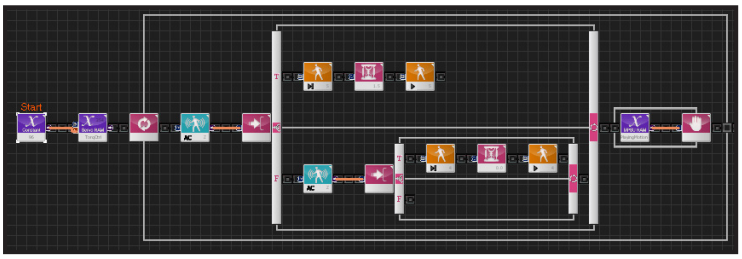
02 시작

모듈의 왼쪽 연결선을 Start Point 에 드래그하여 정확히 도킹을 시킵니다.



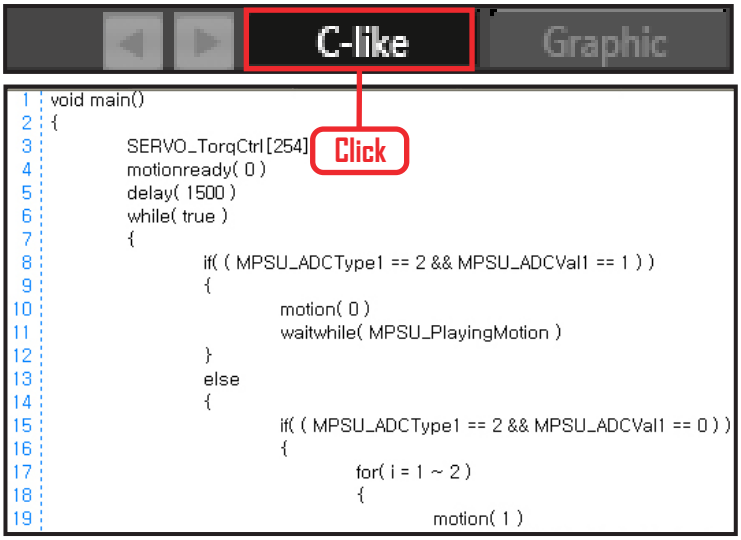
03 프로그래밍 시작

모듈과 Start Point 가 정확히 도킹하면 왼쪽과 같이 활성화된 칼라 이미지 모듈로 변합니다.
그럼 프로그래밍이 시작되었다는 의미입니다.



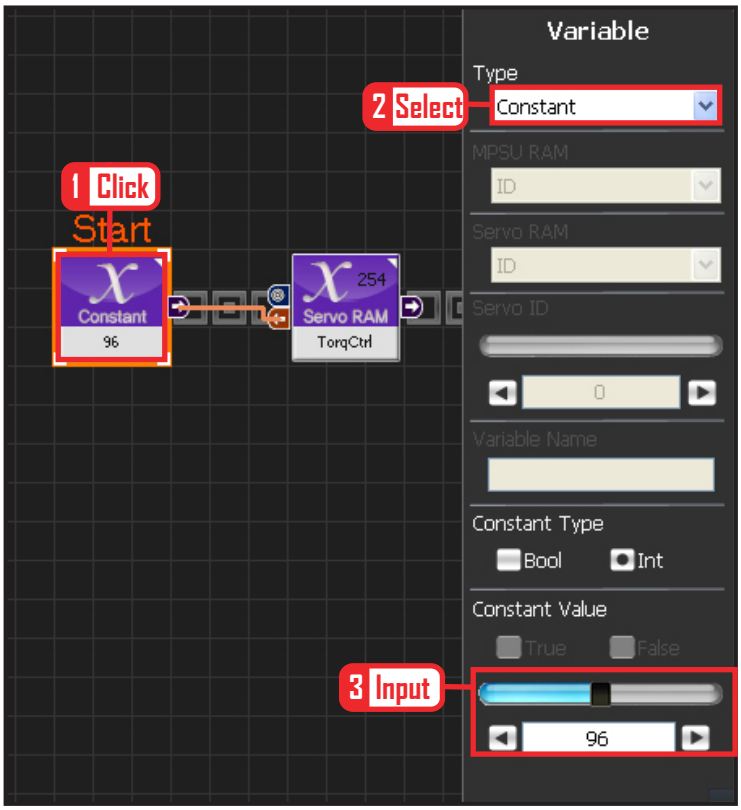
04 전체 프로그래밍

가속도 센서를 이용하여 넘어졌던 로봇이 일어나는 프로그래밍을 해봅니다.



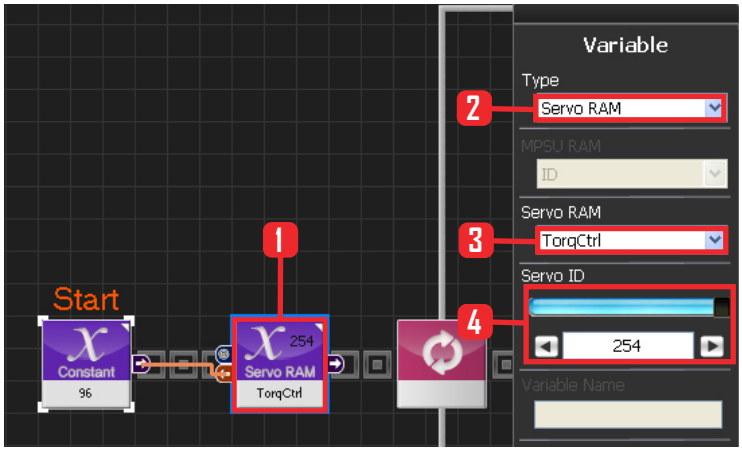
05 C-Like 보기

오른쪽 상단의 Graphic 탭에서 C-like 탭을 클릭하면 왼쪽과 같은 Task 프로그래밍 화면이 나옵니다.
중력가속도를 이용한 전체 프로그래밍 화면입니다.
C와 유사한 문법 구조를 가지고 있으므로 C 문법 선행학습 효과도 있습니다.
각 모듈별로 클릭하면 커서가 따라서 움직이므로 모듈별로 Text로 어떻게 변환하는지 확인할 수 있습니다,



06 상수 설정

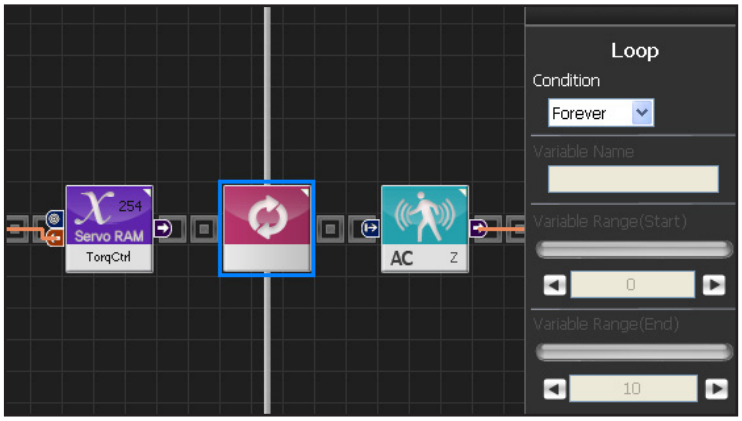
서보 모터를 스스로 움직일 수 있는 상태로 만드는 과정입니다.
Variable Type 을 Constant 로 선택합니다.
속성중에 Constant Value 값을 96 으로 설정합니다.
서보의 TorqControl 레지스터에 96(0x60) 이라는 값이 들어가면 서보가 움직일 수 있는 상태가 됩니다.
그 값은 Output 커넥터를 통하여 뒤 모듈의 토크값에 전달합니다.



07 모든 서보에 적용

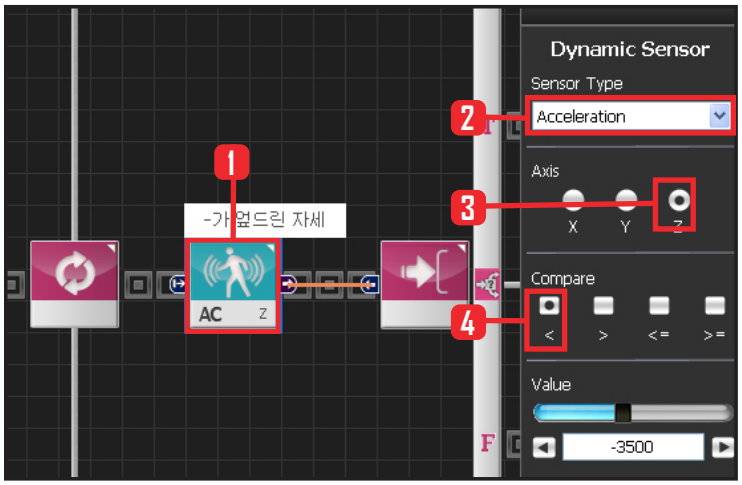
앞에서 받은 96 상수값을 모든 서보에 적용하는 과정입니다.

Variable 〉 Type : Servo RAM을 선택합니다.
Servo RAM : TorqCtrl 을 선택합니다.
Servo ID : 254 를 선택합니다. 254는 연결되어있는 모든 서보에 적용하겠다는 의미입니다.



08 반복

Forever 무한 반복을 설정합니다.

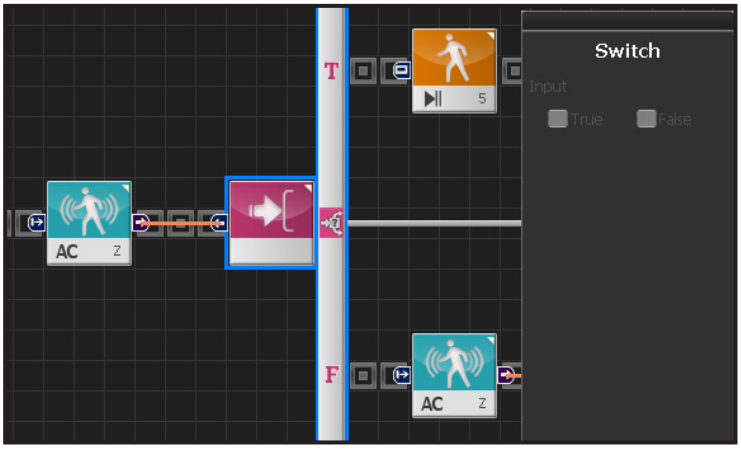


09 가속도 설정 (엎드려 넘어졌을 때)

가속도는 로봇이 반듯하게 서있을 때 0 값을 나타냅니다. 로봇이 앞으로 엎드려 넘어졌을 때 –4096값을 나타내고, 뒤로 누울 때는 +4096 값이 나타납니다.
따라서 –4096 값에 근접했을 때 로봇이 넘어졌다고 판단할 수 있습니다. 그 기준값을 –3500 으로 설정합니다.

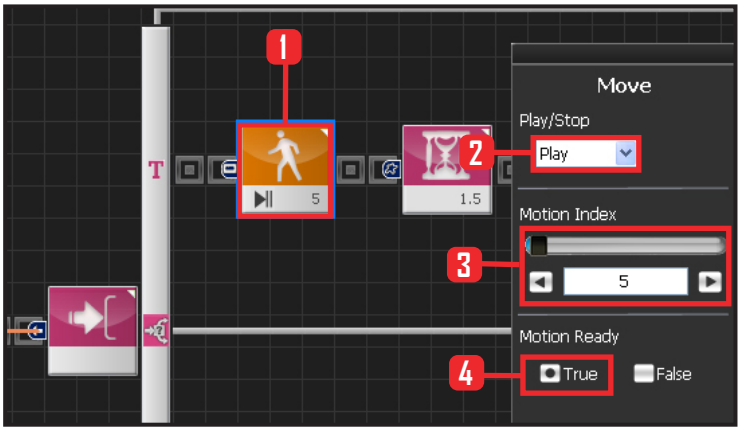
–3500 보다 작은 값이면 로봇이 엎드려 넘어졌다고 판단할 수 있습니다.

Sensor 〉 Dynamic Sensor 모듈을 선택합니다.
Sensor Type : Acceleration 를 선택합니다. 가속도 센서입니다.
Axis : Z 축으로 설정합니다.
Compass : 〈 로 설정합니다.
Value : –3500 으로 설정합니다.
–3500 보다 작으면 앞으로 엎드려 넘어졌다는 설정이 완료되었습니다.



10 If 분기문

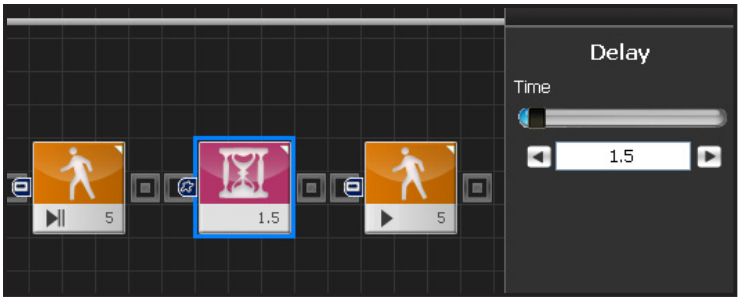
True 일때는 뒤로 일어나게 하고, False 일때는 다음 조건문으로 진행하도록 프로그래밍합니다.



11 뒤로 일어나기 모션

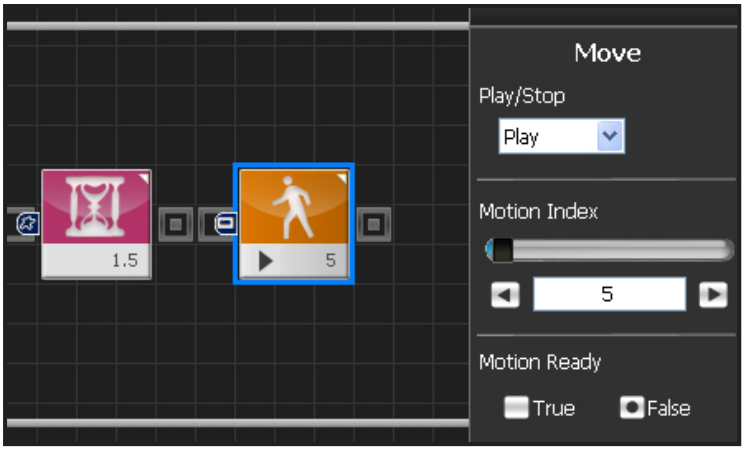
로봇이 앞으로 엎어져 있으므로 로봇의 뒤로 일어나는 동작을 삽입해야합니다.
5번 모션이 뒤로 일어나는 모션입니다.

Motion 〉 Move 모듈을 선택합니다.
Play/Stop : Play 를 선택합니다.
Motion Index : 5 번을 선택합니다. 5번은 뒤로 일어나는 모션입니다.
Motion Ready : True 를 선택합니다.
모션 동작 준비 과정입니다.



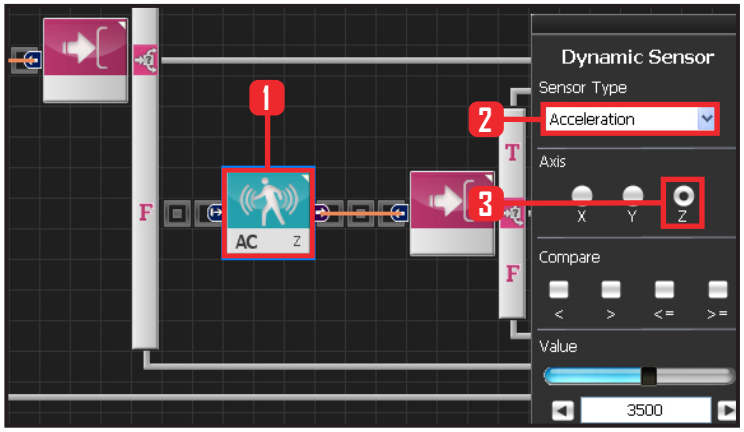
12 Delay

Motion Ready 동작이 끝나기 전에 진행하는 것을 방지하기위해 Delay 값을 1.5 초로 설정합니다



13 뒤로 일어나기 모션 실행

Motion Ready 를 False 로 선택하면 뒤로 일어나기 모션이 실행됩니다.



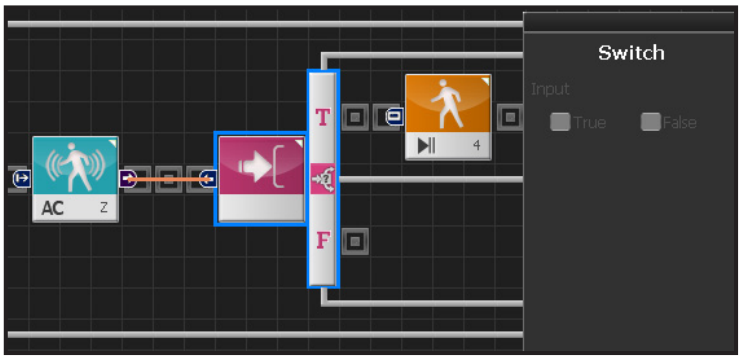
14 중력가속도 설정 (뒤로 넘어졌을 때)

중력가속도는 로봇이 반듯하게 서있을 때 0 값을 나타냅니다.
로봇이 앞으로 엎드려 넘어졌을 때 –4096값을 나타내고, 뒤로 누울 때는 +4096 값이 나타납니다.
따라서 4096 값에 근접했을 때 로봇이 넘어졌다고 판단할 수 있습니다. 그 기준값을 3500 으로 설정합니다.

3500 보다 큰 값이면 로봇이 뒤로 넘어져서 누워있다고 판단할 수 있습니다.

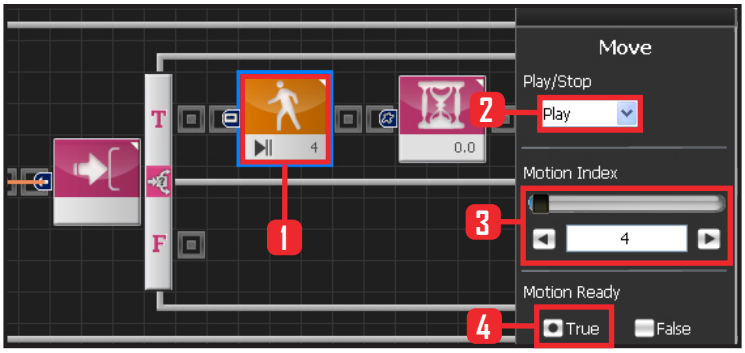
Sensor 〉 Dynamic Sensor 모듈을 선택합니다.

Sensor Type : Acceleration를 선택합니다. 가속도 센서입니다.
Axis : Z 축으로 설정합니다.
Compass : 〉로 설정합니다.
Value : 3500 으로 설정합니다.
3500 보다 크면 뒤로 넘어져서 누워졌다는 설정이 완료되었습니다.



15 If 분기문

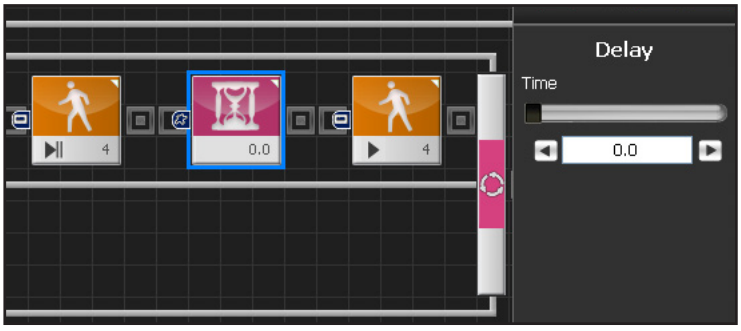
True 일때는 앞으로 일어나는 프로그래밍입니다.



16 모션 동작준비

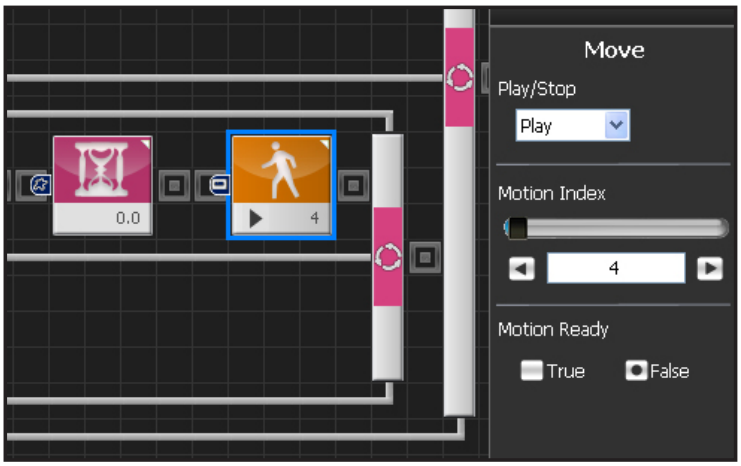
모션이 동작하기 위해서는 준비과정을 거칩니다. 이전 로봇 모션에서 갑작스럽게 변동하면 로봇에 무리가 가해질 수 있습니다. 따라서 현재 실행하고자 하는 로봇의 첫 모션으로 천천히 이동시키는 과정입니다.
Motion Ready 가 True 이면 모션 첫 장면을 준비하는 것이고, False 이면 모션이 동작합니다.

Motion 〉 Move 모듈을 선택합니다.
Play/Stop : Play 를 선택합니다.
Motion Index : 4 번을 선택합니다. 4번은 앞으로 일어나는 모션입니다.
Motion Ready : True 를 선택합니다.
모션 동작 준비 과정입니다.



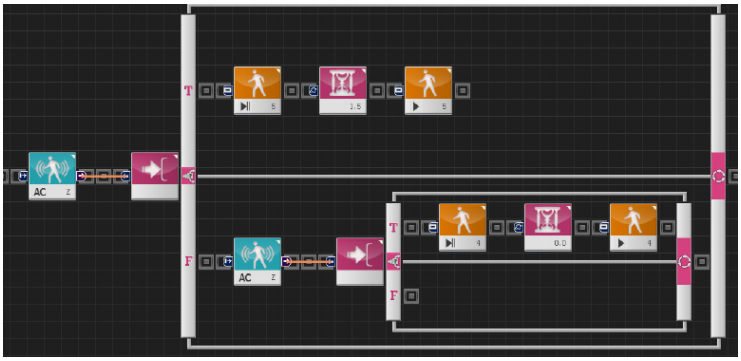
17 Delay

Motion Ready 동작이 끝나기 전에 진행하는 것을 방지하기 위해 Delay 값을 1.5 초로 설정합니다



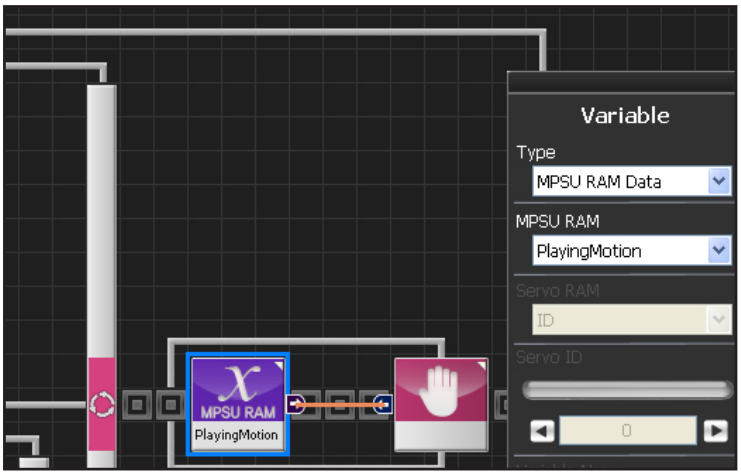
18 앞으로 일어나기 모션 실행

Motion Ready 를 False 로 선택하면 앞으로 일어나기 모션이 실행됩니다



19 넘어진 로봇 일어나기

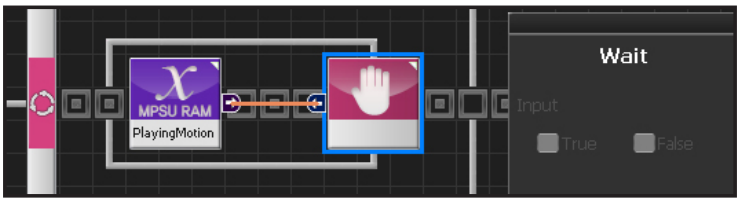
Z축 가속도 값을 기준으로 로봇의 넘어짐을 판단하고, 넘어진 위치에 따라 각각 일어나는 동작을 실행하는 프로그래밍입니다.



20 모션 동작 확인

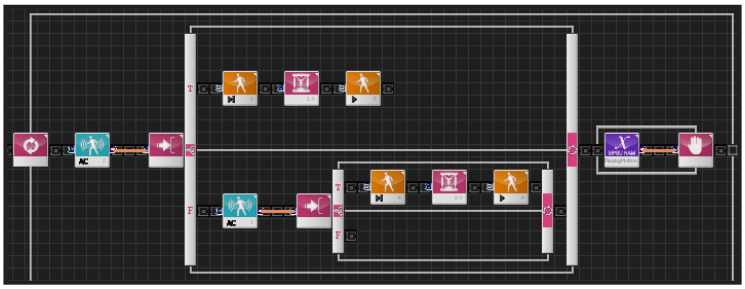
Loop 는 계속적인 반복을 의미합니다. Move 명령을 내리고 나서 실제 모션이 실행되어 완료되기 까지는 시간이 걸리므로 loop안에 Move모듈 하나만을 넣고 실행하면 모션을 이미 실행중임에도 loop를 계속 돌면서 모션실행 명령을 반복하게 됩니다.
이렇게 되면 Move모듈을 만난 횟수와 실제모션을 실행한 횟수가 달라집니다.
따라서 실행한 모션이 끝날때까지 기다렸다가 다시 loop의 처음으로 돌아가게 하는 편이 더 정확합니다.
Variable 〉 MPSU RAM Data 에 들어가면 Playing Motion 이 있습니다. Playing Motion 은 로봇이 모션을 실행중인지 확인하는 변수입니다. 그 Playing Motion에 Wait 를 걸어주면 로봇의 동작이 끝날 때까지 Loop는 기다려줍니다.

Data 〉 Variable 모듈을 선택합니다.
Type : MPSU RAM Data 를 선택합니다.
MPSU RAM : Playing Motion 을 선택합니다.
Output 커넥터 값을 뒤에 Wait 모듈에 연결합니다.
모션이 끝날 때까지 기다리겠다는 의미입니다.



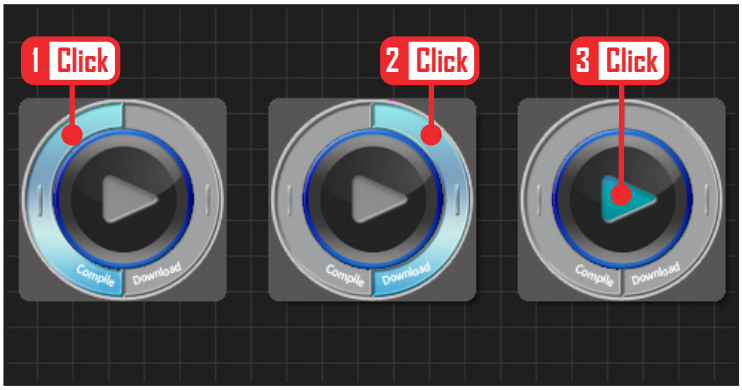
21 Wait

모션이 끝날때까지 기다립니다.
모션이 끝나면 다시 처음으로 돌아가 모션을 반복합니다.



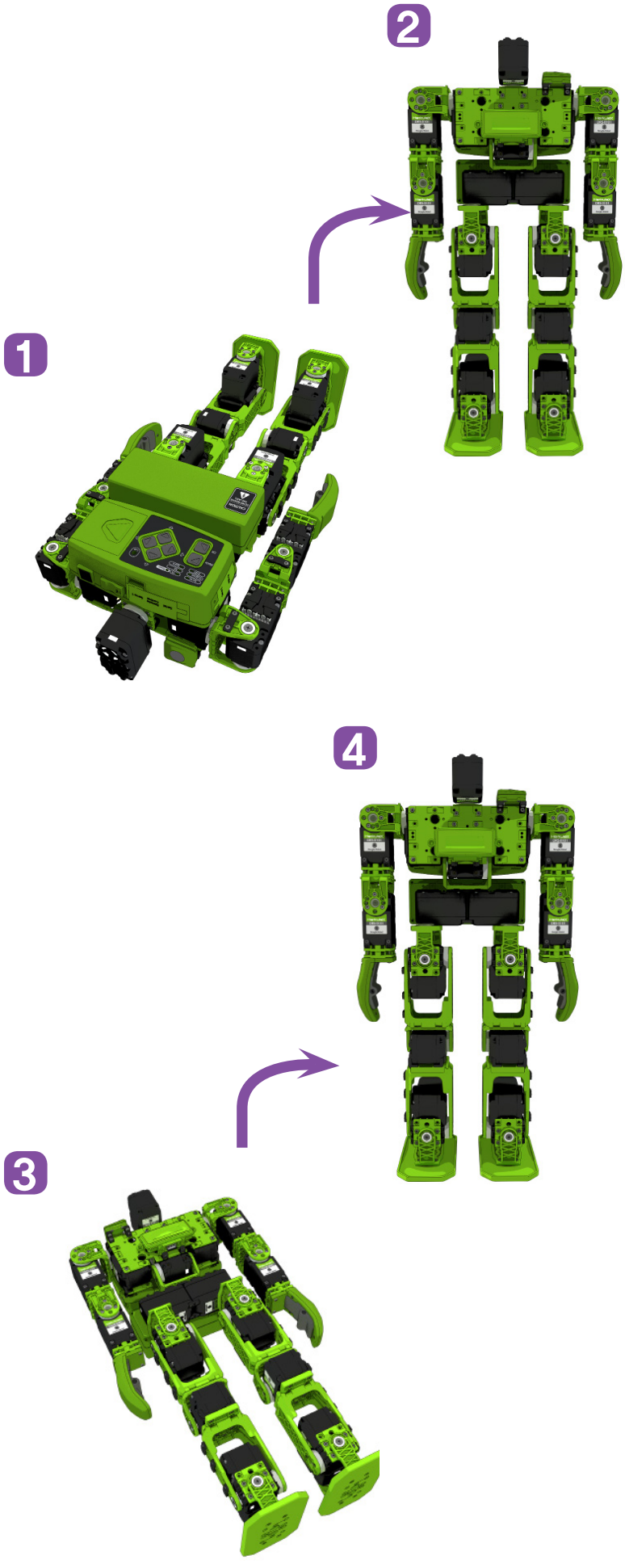
22 전체 프로그래밍

로봇이 앞으로 넘어졌을 때와 뒤로 넘어졌을 때를 판단하여 각각 뒤로 일어나기와 앞으로 일어나기 모션을 실행하는 프로그래밍 입니다.



23 컴파일, 다운로드, 실행

왼쪽 클릭하여 컴파일 시킵니다. 에러가 없으면 오른쪽 클릭하여 로봇에 다운로드 시킵니다. 다운로드 완료되면 가운데 화살표 실행버튼을 눌러 로봇에서 실행시킵니다.



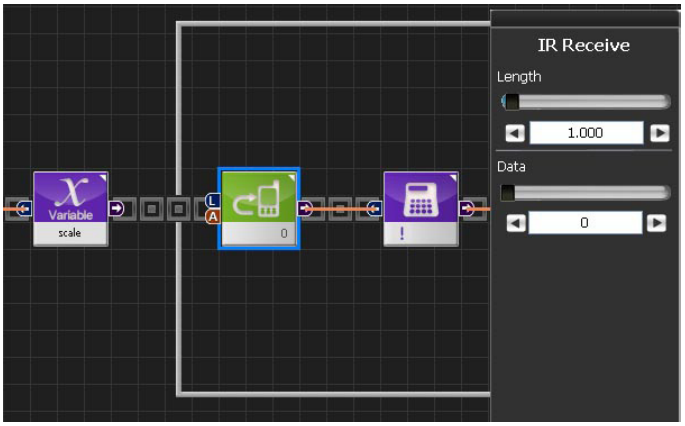
24 로봇동작

로봇이 앞으로 넘어져 있으면 뒤로 일어나고, 로봇이 뒤로 넘어져 있으면 앞으로 일어납니다.

IRReceive, Sound & Motion 예제 따라하기

(Sound 예제로 설명, Motion 예제 설명생략 리모콘 번호 매칭)

IRRecieve Module 의 Data 값의 번호는 오른쪽 리모콘의 키와 매칭됩니다.



Hovis 리모컨의 키맵은 오른쪽 그림과 같습니다. IR Receive 모듈의 Data가 왼쪽의 키 숫자에 대응됩니다.

가령 오른쪽 위의 전원 버튼을 누르면 Data 0이 DRC로 들어오게 됩니다. 이 때 사용자는 IR Receive 모듈에서 Data를 0으로 설정해서 Switch 모듈의 입력으로 연결하면, 전원 버튼을 눌렀을 때 로봇이 어떤 행동을 할 것인지를 프로그래밍 할 수 있습니다.

채널설정

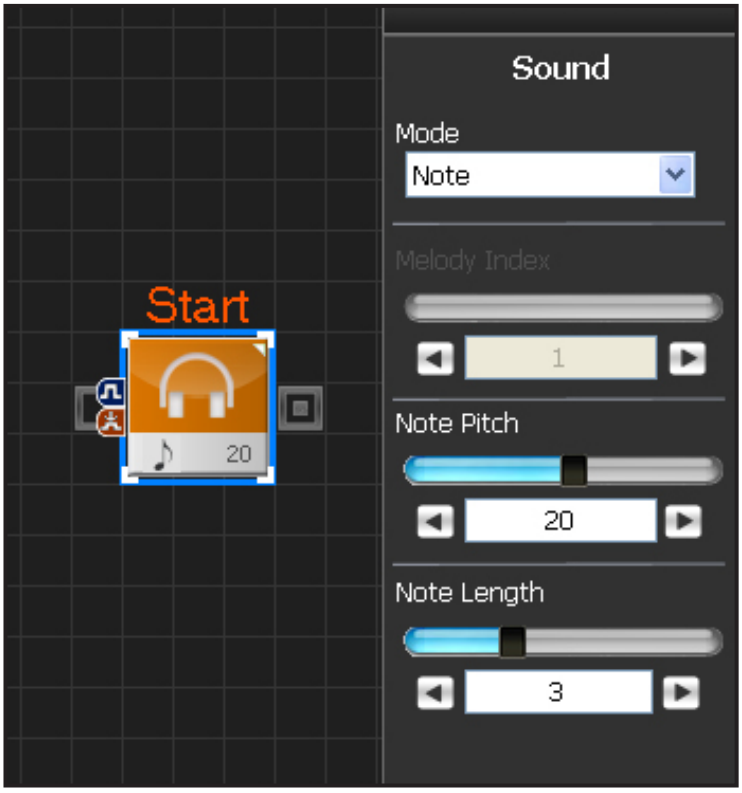
리모컨에는 채널이 있어서, 리모컨의 채널과 DRC에 설정 된 채널이 같아야만 리모컨 데이터를 받을 수 있습니다, 리모컨의 채널은 1~0의 숫자 + OK를 동시에 누름으로써 설정할 수 있으며, DRC의 채널은 MPSU Ram Data의 RmcChannel 값을 바꾸어 설정할 수 있습니다. 리모컨의 숫자에 대응되는 RmcChannel 값은 아래와 같습니다.

리모컨 버튼	RmcChannel 값
0+OK	97(0x61)
1+OK	98(0x62)
2+OK	99(0x63)
3+OK	100(0x64)
4+OK	101(0x65)
5+OK	102(0x66)
6+OK	103(0x67)
7+OK	104(0x68)
8+OK	105(0x69)
9+OK	106(0x6A)

예제설명

리모컨의 번호와 Sound 음을 매칭시켜서 도레미파솔라시도 (1번~8번) 음이 나오게 하는 프로그램입니다.

도레미 음은 Motion 〉 Sound 모듈에서 Note의 Pitch 값에 따라 음이 달라집니다. DRC 제어기 음높이는 총 0번에서 37번까지 있으며, 총 3옥타브까지 음을 낼 수 있습니다.



00 Sound 속성창

Motion 〉 Sound 모듈을 선택합니다.
Mode 에는 Melody 와 Note 가 있습니다. Melody 는 저장되어있는 편집음 리스트중에 하나를 선택하여 플레이합니다.
Mode 에서 Note 는 자체적인 36개의 음을 가져와서 사용할 때 선택합니다.
아래 표를 참조하세요

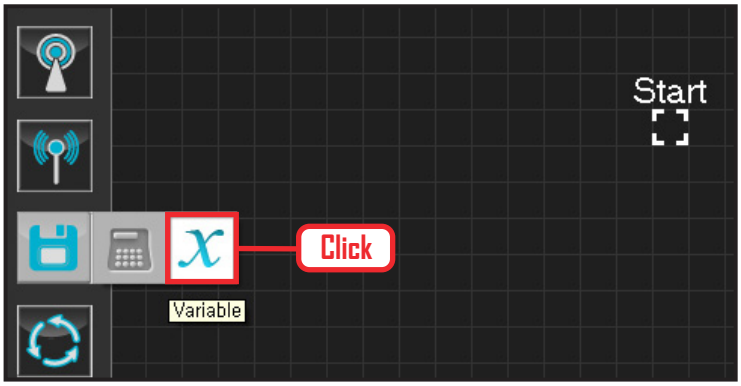
Note Pitch 는 0번에서 37번까지 선택합니다. 총 3옥타브로 구성됩니다.
Note Length 는 음의 박자를 의미합니다. 32분음표에서 온음표까지 선택할 수 있습니다.
아래 표를 참조하세요.

Note Pitch (음높이)

번호	0												
계명	무음												
번호	1	2	3	4	5	6	7	8	9	10	11	12	
계명	도	도#	레	레#	미	파	파#	솔	솔#	라	라#	시	
번호	13	14	15	16	17	18	19	20	21	22	23	24	
계명	도	도#	레	레#	미	파	파#	솔	솔#	라	라#	시	
번호	25	26	27	28	29	30	31	32	33	34	35	36	37
계명	도	도#	레	레#	미	파	파#	솔	솔#	라	라#	시	도

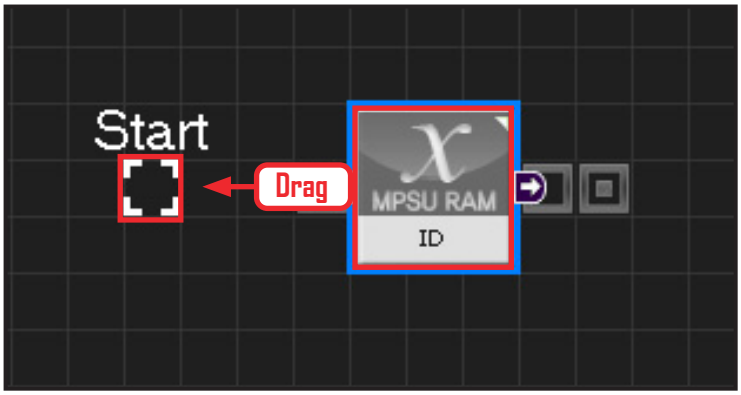
Note Length (음길이)

번호	0	1	2	3	4	5	6	7	8	9
Raw Data	6	12	18	24	36	48	72	96	144	192
시간 (ms)	38.4	76.8	115.2	153.6	230.4	307.2	460.8	614.4	921.6	1228.8
음표	32분 음표	16분 음표	점16분 음표	8분 음표	점8분 음표	4분 음표	점4분 음표	2분 음표	점2분 음표	온음표



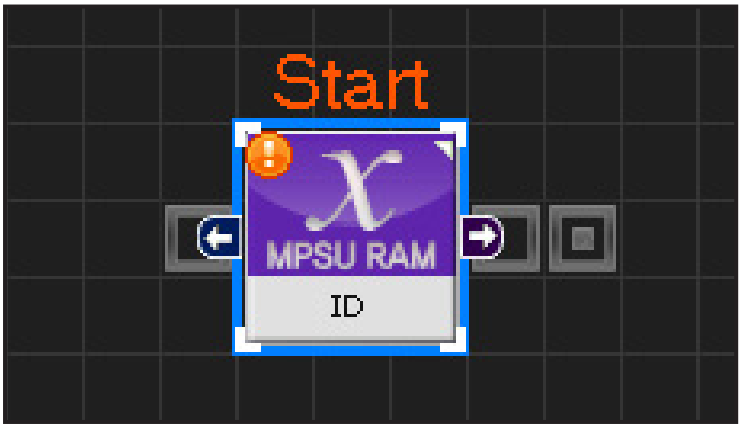
01 변수 지정

Data 〉 Variable 모듈을 클릭합니다.



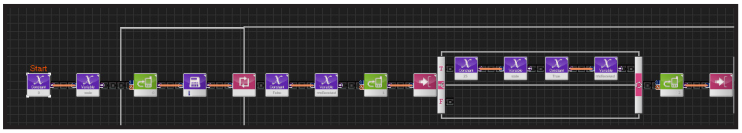
02 시작

모듈의 왼쪽 연결선을 Start Point 에 드래그하여 정확히 도킹을 시킵니다.



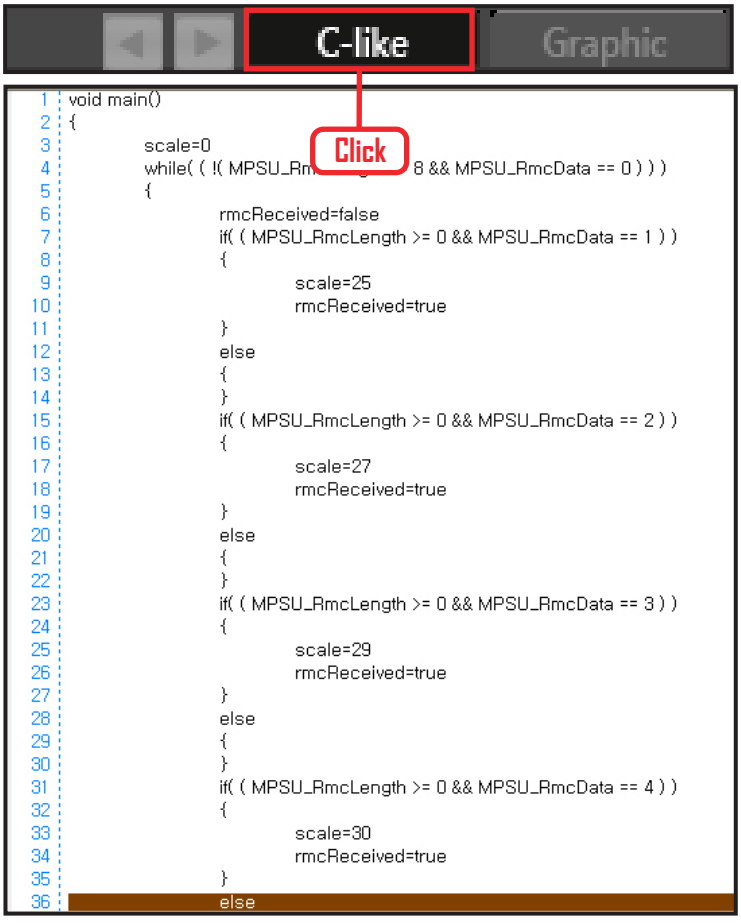
03 프로그래밍 시작

모듈과 Start Point 가 정확히 도킹하면 왼쪽과 같이 활성화된 칼라 이미지 모듈로 변합니다.
그럼 프로그래밍이 시작되었다는 의미입니다.



04 전체 프로그래밍

리모컨과 버저를 이용한 전체 프로그래밍입니다.

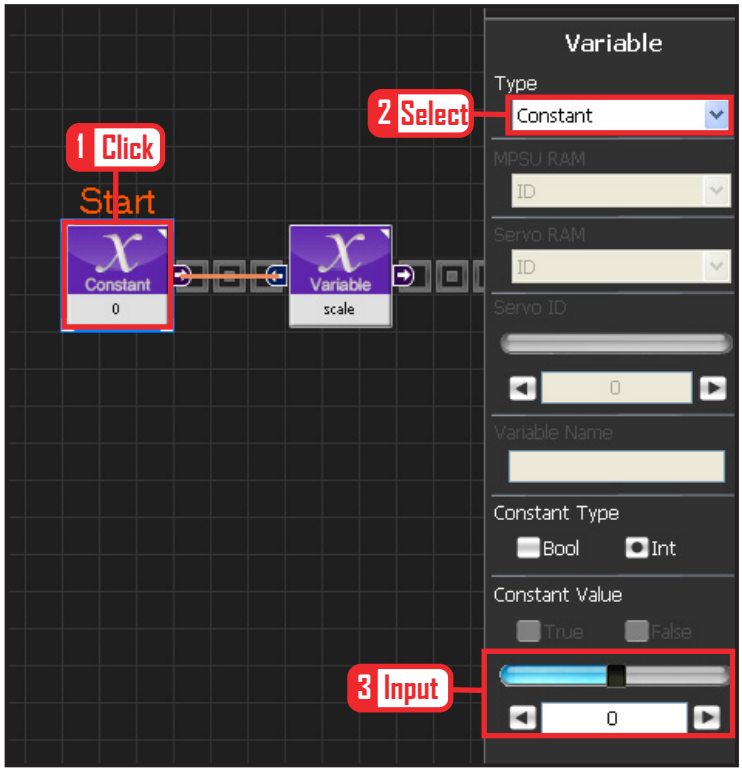


05 C-Like 보기

오른쪽 상단의 Graphic 탭에서 C-like 탭을 클릭하면 왼쪽과 같은 Task 프로그래밍 화면이 나옵니다.
IRReceive 리모컨과 사운드를 이용한 전체 프로그래밍 화면입니다.
C와 유사한 문법 구조를 가지고 있으므로 C 문법 선행학습 효과도 있습니다.
각 모듈별로 클릭하면 커서가 따라서 움직이므로 모듈별로 Text로 어떻게 변환하는지 확인할 수 있습니다

```
37                  {
38                  }
39                  if( ( MPSU_RmcLength >= 0 && MPSU_RmcData == 5 ) )
40                  {
41                          scale=32
42                          rmcReceived=true
43                  }
44                  else
45                  {
46                  }
47                  if( ( MPSU_RmcLength >= 0 && MPSU_RmcData == 6 ) )
48                  {
49                          scale=34
50                          rmcReceived=true
51                  }
52                  else
53                  {
54                  }
55                  if( ( MPSU_RmcLength >= 0 && MPSU_RmcData == 7 ) )
56                  {
57                          scale=36
58                          rmcReceived=true
59                  }
60                  else
61                  {
62                  }
63                  if( ( MPSU_RmcLength >= 0 && MPSU_RmcData == 8 ) )
64                  {
65                          scale=37
66                          rmcReceived=true
67                  }
68                  else
```

```
69                  {
70                  }
71                  if( ( true == rmcReceived ) )
72                  {
73                          note( scale, 3 )
74                          waitwhile( MPSU_BuzzTime )
75                  }
76                  else
77                  {
78                  }
79          }
80  }
```



06 상수 설정

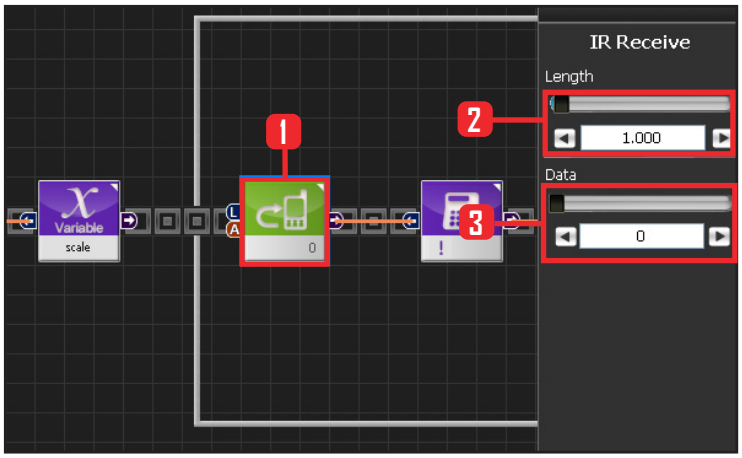
플레이할 음계의 변수를 선언하는 과정입니다.

Data〉Variable 을 선택합니다.
Type : Constant 로 선택합니다.
Constant Value : 0 으로 설정합니다.

07 변수명

Play할 음계의 변수 이름을 지정합니다.

Data 〉 Variable 을 선택합니다.
Type : Variable 로 선택합니다.
Variable Name : scale 로 입력합니다.

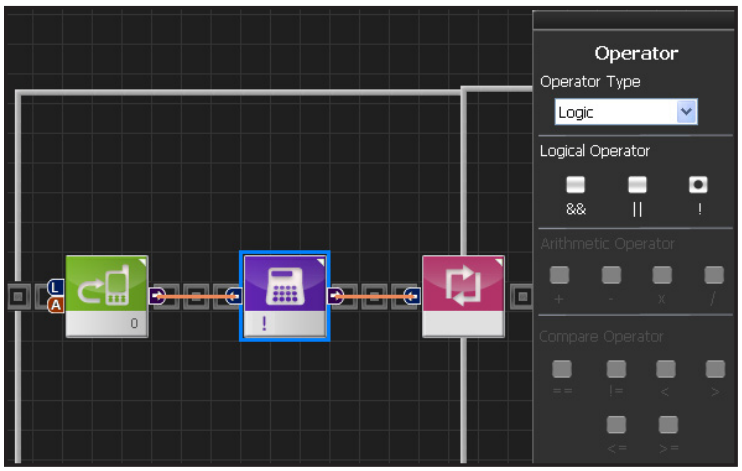


08 While 문 예외 조건

리모컨 0번을 길게 누르면 빠져나옵니다.

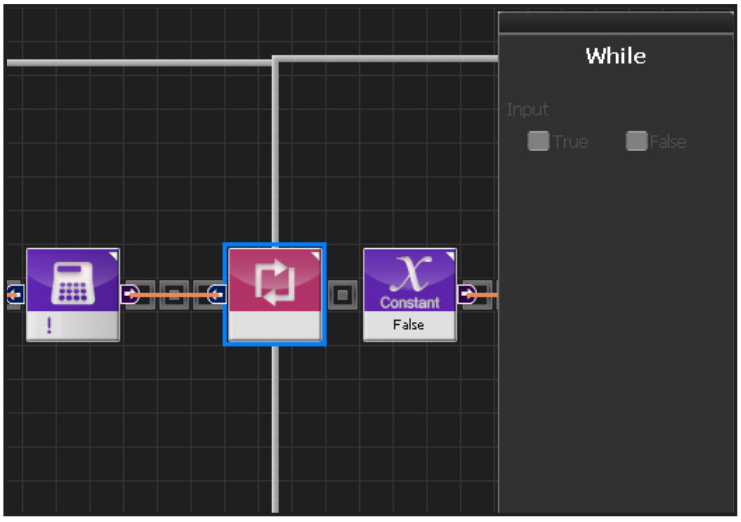
Communication〉IRReceive 모듈을 선택합니다.
Length : 1.000 으로 설정합니다. 1초동안 누른다는 의미입니다.
Data : 0 으로 설정합니다. 전원버튼을 누른다는 의미입니다.

전원버튼을 1초 이상 길게 누르면 이 모듈의 Output 값은 true가 되고, 아닌경우 false 입니다.



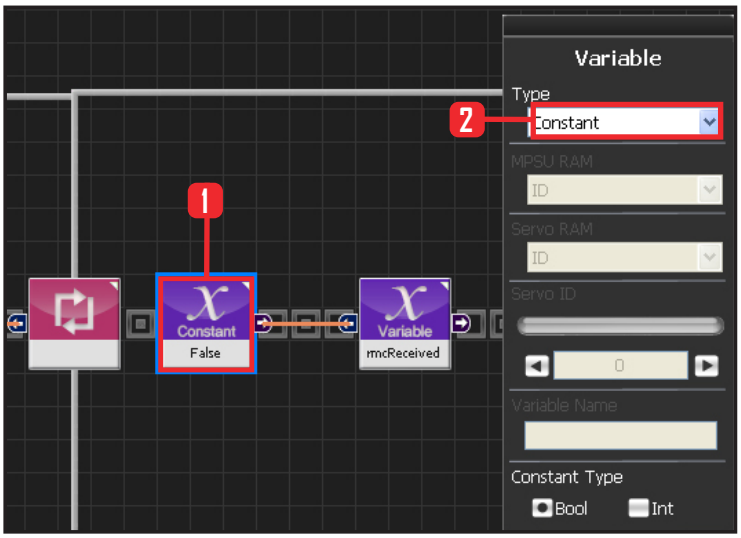
09 ! 연산자 설정

! 은 true / false 값을 반대로 만들어 줍니다. 따라서 IRReceive 모듈의 출력값이 반대가 되어 while문의 입력으로 들어갑니다.



10 While 조건반복

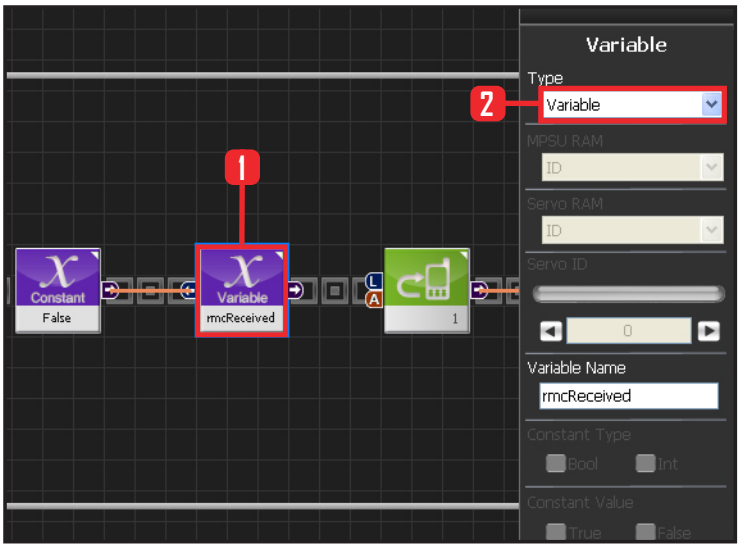
앞에 조건에 따라 반복을 합니다.
입력이 true이면 계속 뒤의 내용을 반복합니다. !연산자를 거치므로 IRReceive 모듈의 Output이 false이면 반복하고, true 이면 loop를 빠져 나옵니다. 즉 전원 버튼을 1초 이상 누르면 반복문을 빠져나오는 것 입니다.



11 리모컨 입력 변수 초기화

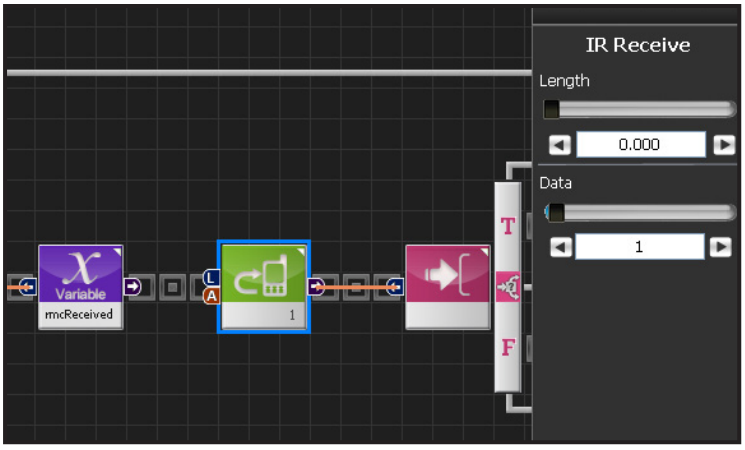
리모컨을 입력받았다는 변수를 설정합니다.

Data 〉 Variable 모듈을 선택합니다.
Type : Contant 를 선택합니다.
Constant Type 은 Bool 로 설정합니다. Bool 은 참과 거짓을 나타내는 자료형입니다.
Constant Value : False 를 선택합니다



12 리모컨 입력 초기 변수

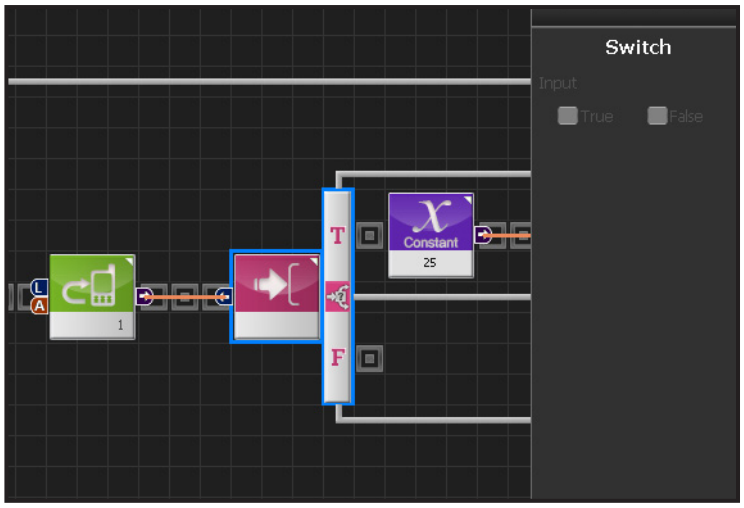
Data 〉 Variable 을 선택합니다.
Type : Variable 로 선택합니다.
Variable Name : rmcReceived 로 입력합니다
rmcReceived 는 loop에서 1~8의 리모콘 입력을 받았다는 것을 표시하는 변수명으로서, loop(반복) 처음에는 false로 초기화하고, loop(반복)이 끝날때쯤 검사해서 true 이면 음표가 플레이 됩니다.



13 1번 리모컨

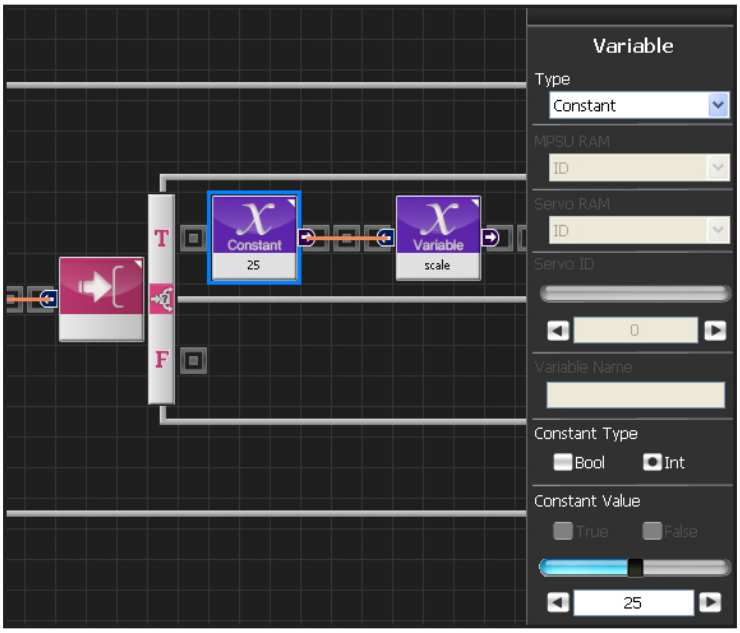
1번 리모컨이 눌렀는지 검사합니다.

Communication〉IRReceive 모듈을 선택합니다.
Length : 0.000 으로 설정합니다.
Data : 1 로 설정합니다. 1번 버튼을 의미합니다..



14 IF 조건문

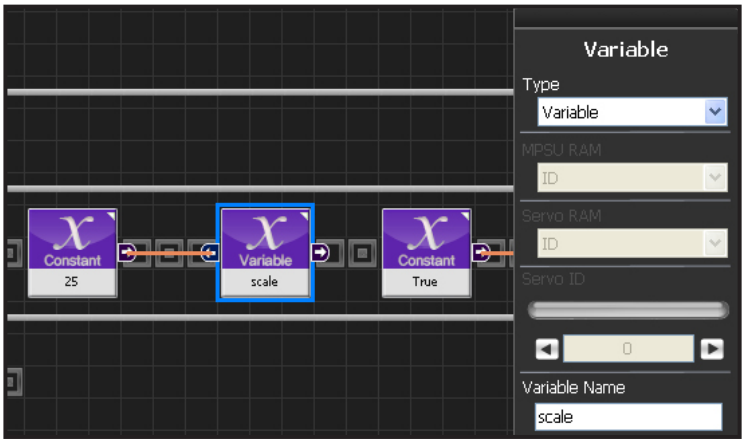
True 이면 실행합니다.



15 "도" 음 저장

앞에서 설명한 Note Pitch (3옥타브 음) 번호가 25 번이면 도 음을 의미합니다.
Scale 값을 도 로 바꿔줍니다.

Data〉Variable 모듈을 선택합니다.
Type : Contant 를 선택합니다.
Constant Type 은 int 로 설정합니다.
Constant Value : 25 를 선택합니다. 25는 음계 "도" 를 나타냅니다.

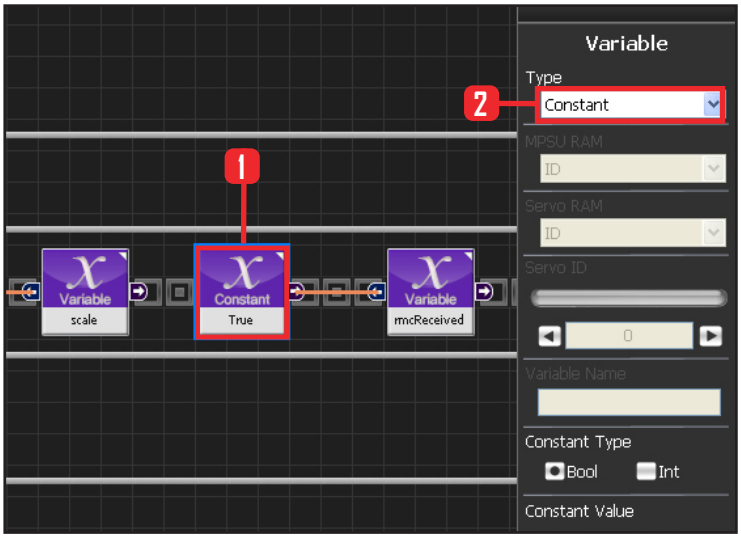


16 scale

Play할 음계의 변수 이름은 scale 로 동일합니다.

Data 〉 Variable 을 선택합니다.
Type : Variable 로 선택합니다.
Variable Name : scale 로 입력합니다.

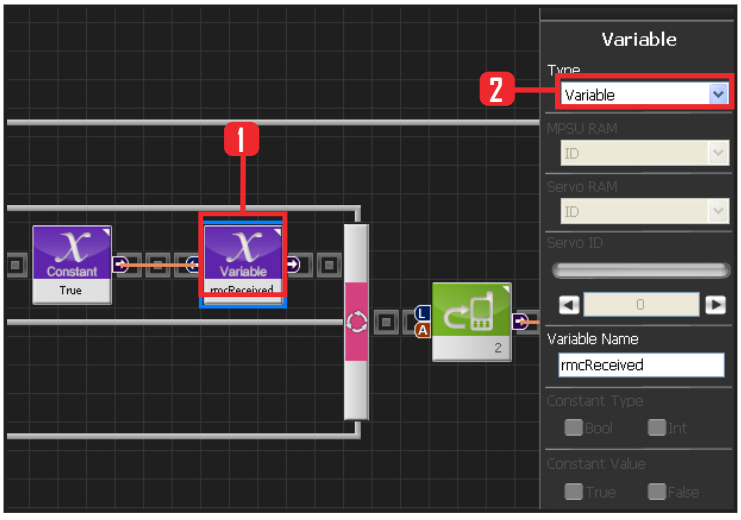
앞 상수 25 값을 input connecter 로 입력받습니다.



17 리모컨 받았음을 저장

변수 rmcRecieved 가 True 이면 1~8번 버튼중 하나가 눌렀다는 것을 의미합니다.

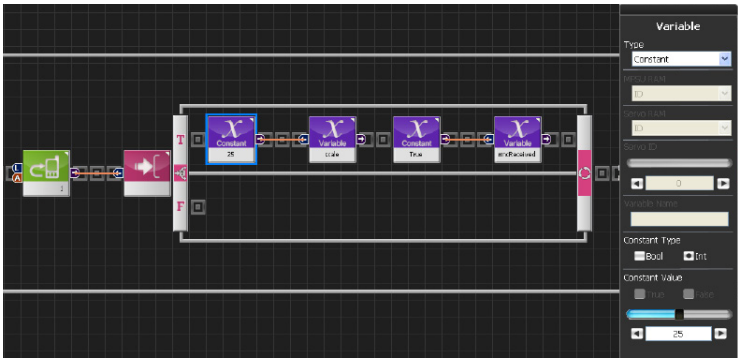
Data 〉 Variable 모듈을 선택합니다.
Type : Contant 를 선택합니다.
Constant Type 은 Bool 로 설정합니다.
Constant Value : True 를 선택합니다.



18 리모컨 받았음을 저장

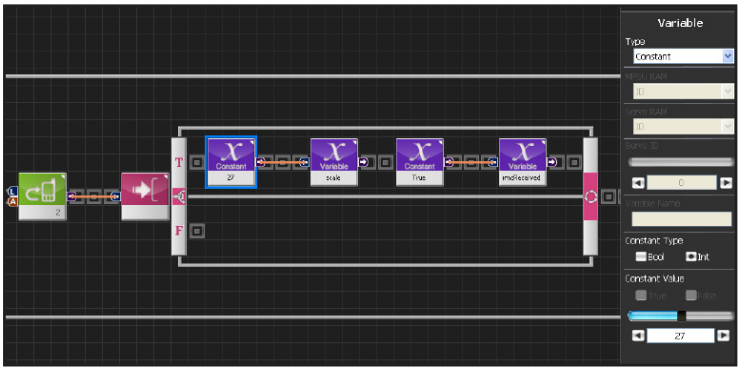
Data 〉 Variable 을 선택합니다.
Type : Variable 로 선택합니다.
Variable Name : rmcReceived로 동일합니다.

앞상수 true값을 input connertor로 입력 받습니다.



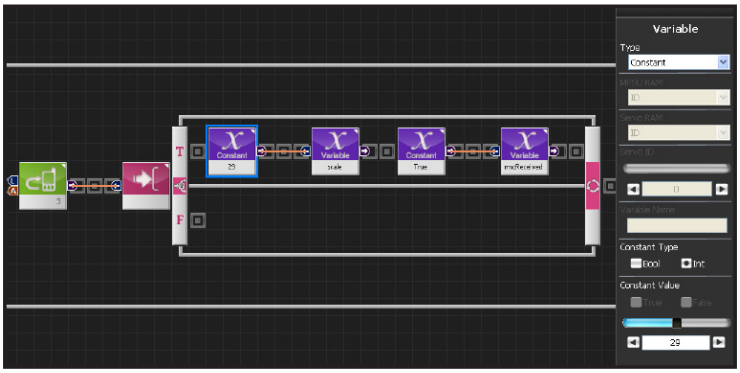
19 1번 -> "도" 음

리모컨 1번을 눌렀을 때 '도' 음을 Scale에 저장하는 프로그래밍 입니다.



20 2번 -> "레" 음

리모컨 2번을 눌렀을 때 '레' 음을 Scale에 저장하는 프로그래밍 입니다.
Scale = 27 번이 '레' 음입니다.

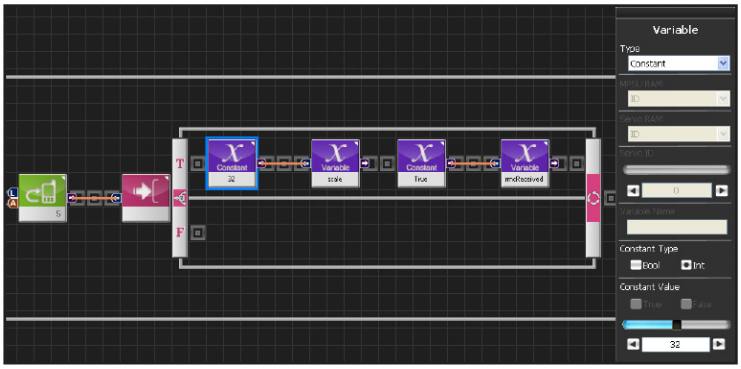


21 3번 -> "미" 음

리모컨 3번을 눌렀을 때 '미' 음을 Scale에 저장하는 프로그래밍 입니다.
Scale = 29 번이 '미' 음입니다.

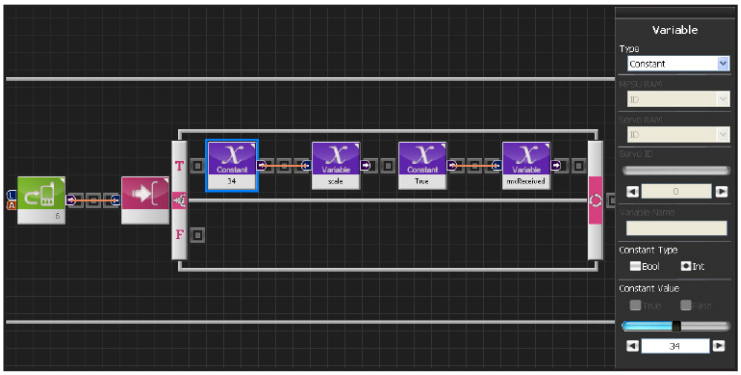
22 4번 -> “파” 음

리모컨 4번을 눌렀을 때 ‘파’ 음을 Scale에 저장하는 프로그래밍 입니다.
Scale = 30 번이 파’ 음입니다.



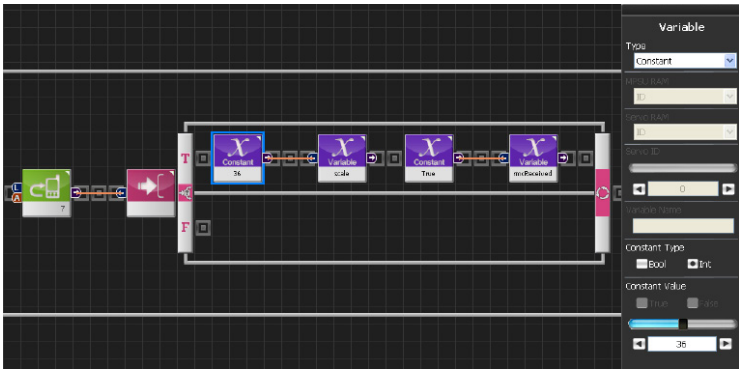
23 5번 -> “솔” 음

리모컨 5번을 눌렀을 때 ‘솔’ 음을 Scale에 저장하는 프로그래밍 입니다.
Scale = 32 번이 ‘솔’ 음입니다



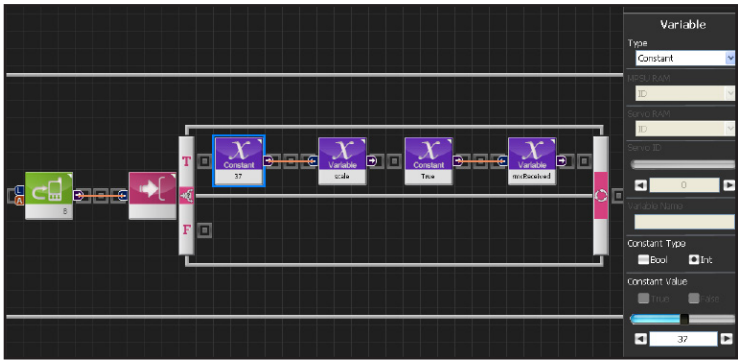
24 6번 -> “라” 음

리모컨 6번을 눌렀을 때 ‘라’ 음을 Scale에 저장하는 프로그래밍 입니다.
Scale = 34 번이 ‘라’ 음입니다 .



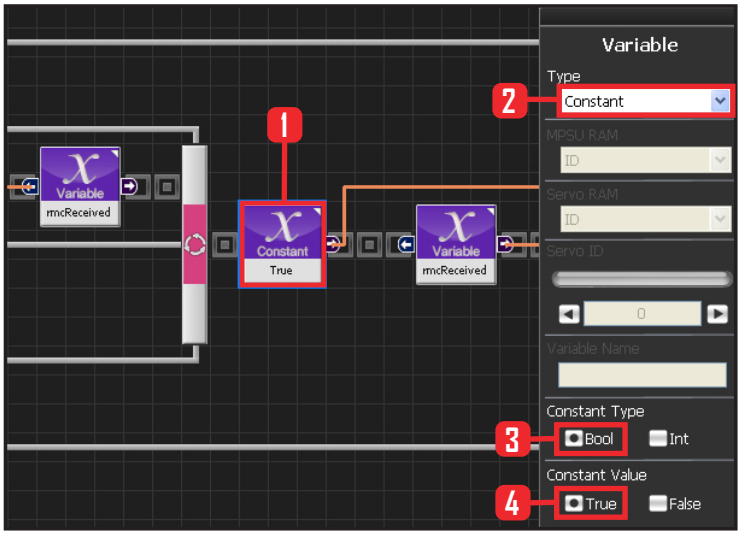
25 7번 -> "시" 음

리모컨 7번을 눌렀을 때 '시' 음을 Scale에 저장하는 프로그래밍 입니다.
Scale = 36 번이 '시' 음입니다



26 8번 -> "도" 음

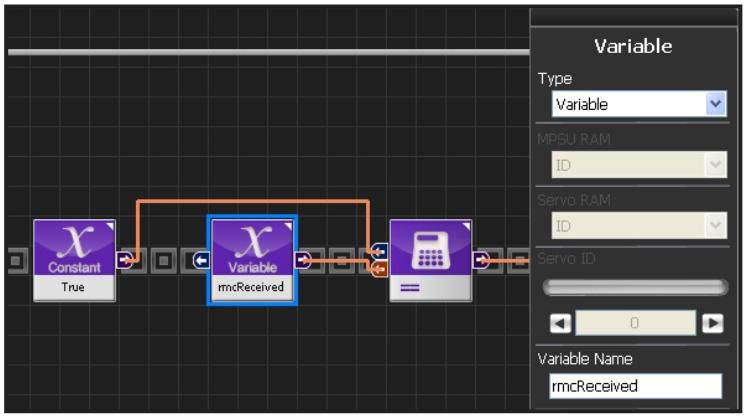
리모컨 8번을 눌렀을 때 '도' 음을 Scale에 저장하는 프로그래밍 입니다.
Scale = 37 번이 '도' 음입니다 .



27 rmcReceived 가 True 일 때

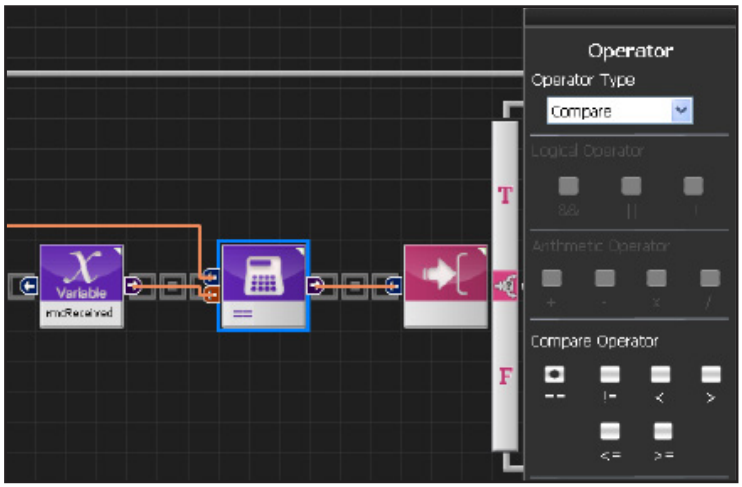
rmcReceived 가 True 일 때 앞서 음 높이 값을 저장한 scale 값을 note 에 넣어줘서 음이 나오게 합니다.

Data 〉 Variable 모듈을 선택합니다.
Type : Contant 를 선택합니다.
Constant Type 은 Bool 로 설정합니다. Bool 은 참과 거짓을 나타내는 자료형입니다.
Constant Value : True 를 선택합니다



28 rmcReceived 가 True 일 때

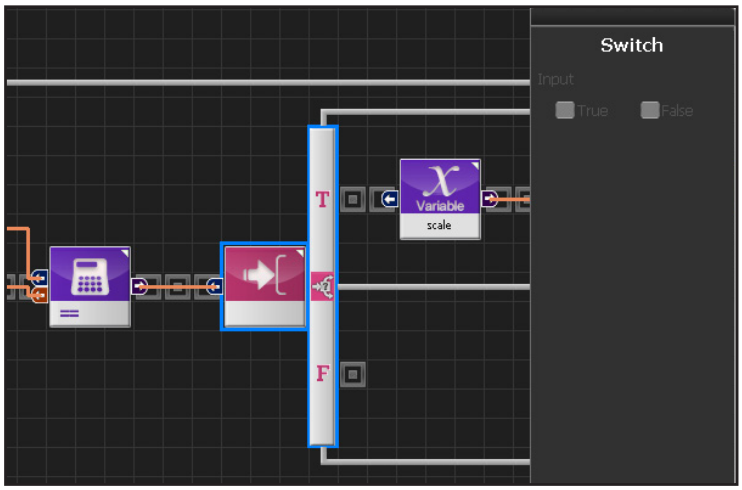
rmcReceived 변수 명은 동일합니다.



29 비교연산자 ==

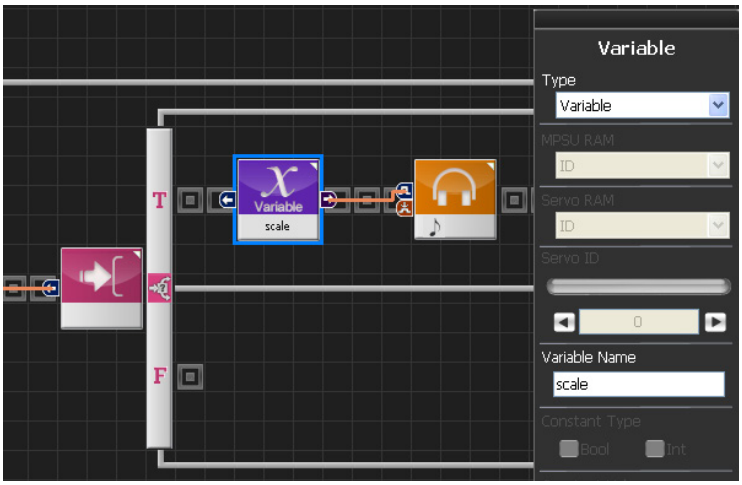
Data 〉 Operator 모듈을 선택합니다.
Operator Type : Compare 를 선택합니다.
Compare Operator 는 == 를 선택합니다.

rmcReceived = = true 라는 의미이며, “rmcReceived 가 true 와 같다” 를 나타냅니다.



30 Switch IF 분기문

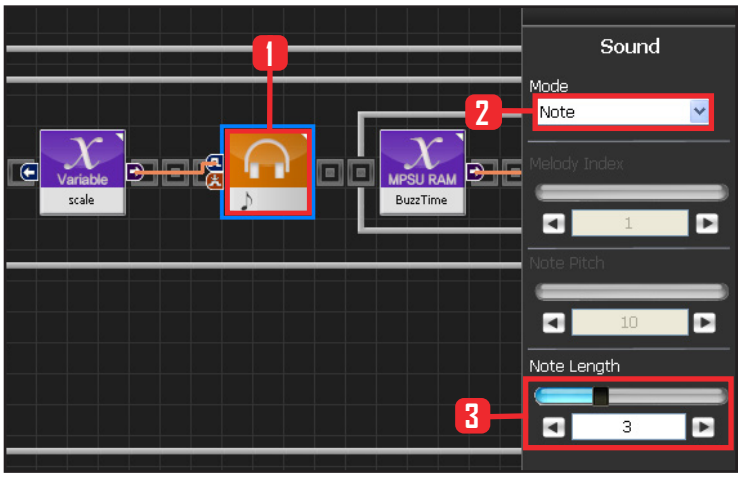
True 이면 실행합니다.



31 scale -> note

Scale 값을 Note 에 넣어주는 과정입니다.

변수 scale 모듈을 만듭니다.

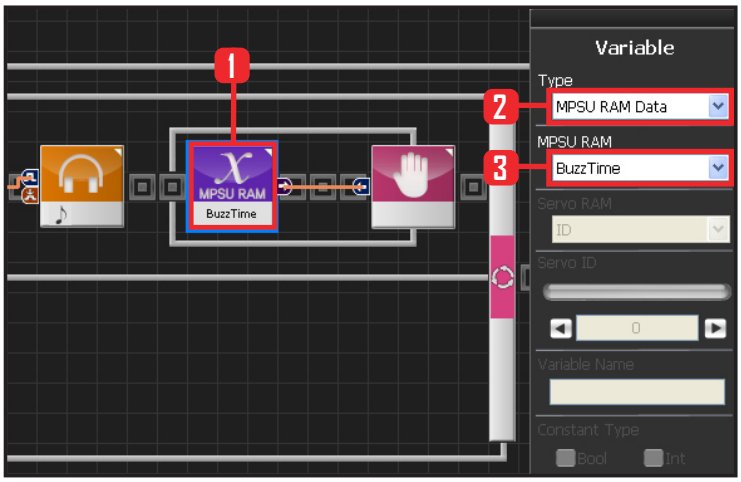


32 사운드 Play

Scale 값을 note 에 넣어줘 소리가 나오게 합니다.

Motion 〉 Sound 모듈을 선택합니다.
Note Length : 3을 선택합니다. 8분음표를 뜻하며 153.6ms 동안을 합니다.

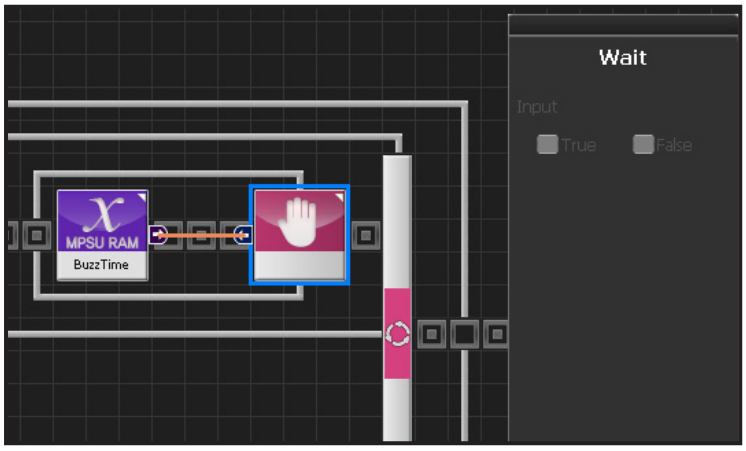
앞서 리모컨으로 부터 입력받은 버튼에 따라 scale 값을 다르게 저장하였고, 그 scale 값을 Note Pitch 넣어주면 거기에 해당하는 음계가 울립니다.



33 BuzzTime

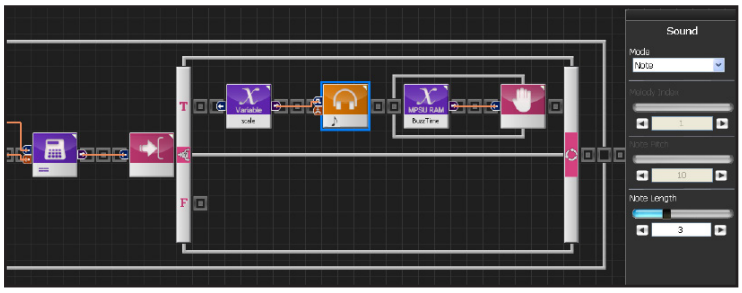
MPSU RAM Data 에 BuzzTime 은 음이 울리고 있는지 여부를 판단하고 기다립니다.
BuzzTime은 버져가 울리기 시작할때 특정값이 된 후에 6.4ms마다 1씩 줄어듭니다.
0이 아닐 때 버져가 울리고 있는 상태가 되고, 0일 때 버저가 안 울리고 있는 상태가 됩니다.
음표에 따른 초기 BuzeTime 값은 두번째 페이지의 음길이 표에서 “Raw Data”를 참조하시면 됩니다.

Data 〉 Variable 을 선택합니다.
Type : MPSU RAM Data 를 선택합니다.
MPSU RAM : BuzzTime 을 선택합니다.



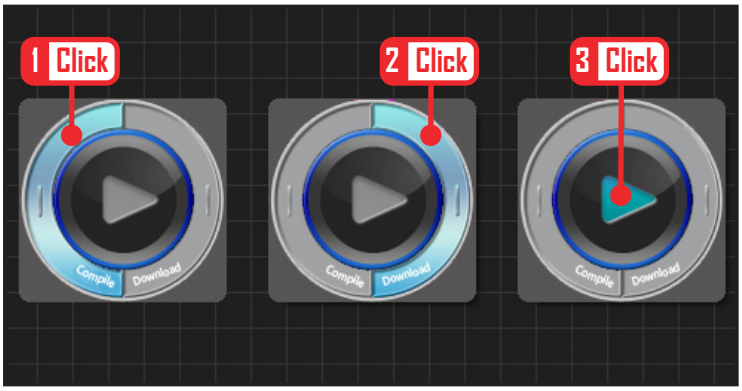
34 Wait

Buzztime이 0이 될때까지, 즉 소리가 끝날 때까지 기다립니다



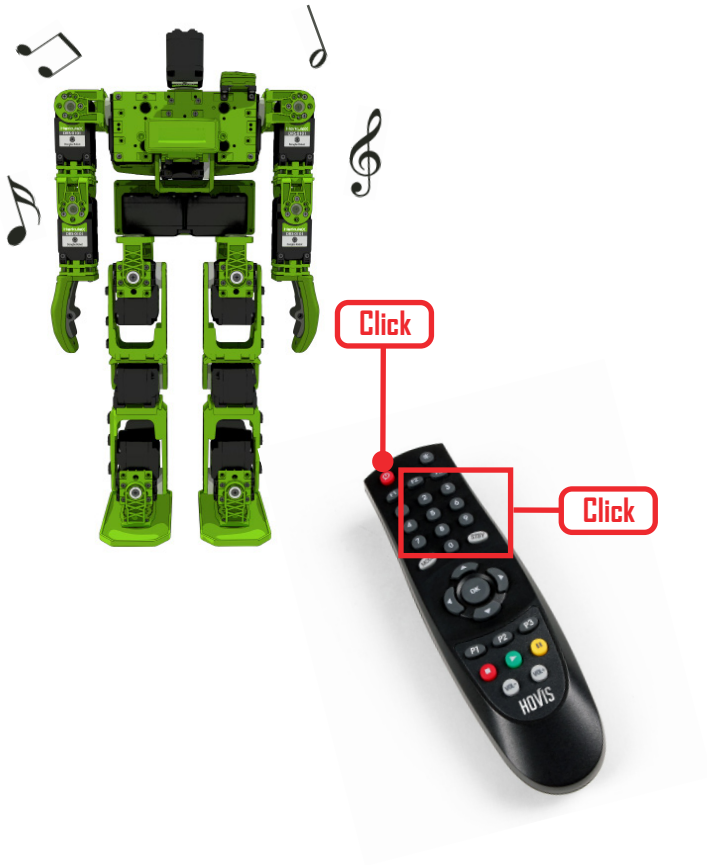
35 음이 나오는 과정

rmcReceived 가 True일때 scale에 저장된 값을 Sound 모듈에 입력하여 Sound 모듈에서 해당음이 발생합니다.
그 음은 BuzzTime 으로 종료를 확인하고 다시 처음으로 돌아갑니다.



36 컴파일, 다운로드, 실행

왼쪽 클릭하여 컴파일 시킵니다. 에러가 없으면 오른쪽 클릭하여 로봇에 다운로드 시킵니다. 다운로드 완료되면 가운데 화살표 실행버튼을 눌러 로봇에서 실행시킵니다.



37 로봇동작

리모컨 1~8버튼을 누르면 음계를 실행하고, 전원 버튼을 길게 누르면 Task가 종료됩니다.

PART 03

Visual Logic 연계된 C언어 문법익히기

C 언어 문법 익히기

C 언어 with Robot

◆ 로봇에서 C 언어를 배운다는 것

로봇은 움직이는 컴퓨터라고 보면 됩니다. 그렇다면 로봇은 두뇌 역할을 하는 CPU를 가지고 있을 것입니다. 그 CPU 를 MCU 라고 부르는데, MCU 는 성능에 따라서 8bit, 16bit, 32bit 로 나눠집니다. 그 중 로봇 컨트롤러로 가장 많이 쓰이는 MCU 가 8bit ATmel 사의 ATmega 128 입니다.
동부로봇 휴머노이드에 쓰이는 컨트롤러인 DRC 는 ATmega128 칩을 사용한 제어기입니다.

우리가 흔히 언어,즉 프로그래밍 랭귀지를 가장 처음 배울때 C 언어부터 익힙니다. 그렇다면, 로봇이 C 언어 입문 교재로 사용될 수 있나에 대한 의문점이 생깁니다. 현실은 그리 순탄치 않다가 정답입니다. 우선 로봇에 대한 메카니즘을 익히고, C 언어로 풀어가야하기 때문에, 가장 많이 쓰이는 Visual C++ 에 비해, 상대적으로 어렵다고 볼 수 있습니다.

하지만, 로봇으로 C 언어를 익힌다는 것은 PC 어플리케이션 개발에 사용되는 MFC 보다는 좀더 난이도 있고, 고급스러운 프로그래밍을 할 수 있다는 것을 의미하고, 로봇으로 인해 흥미를 유발할 수 있다는 점에서 더 큰 차별성을 둘 수 있습니다. 바로 Firmware 를 가르킬 수 있기 때문입니다.
하드웨어 설계가 끝나면 Firmware 프로그래밍을 해야합니다. Firmware 없이 어플리케이션을 만들 수 없고, 다른 응용 프로그램을 만들 수가 없습니다. 따라서 Firmware 엔지니어는 일반 어플리케이션 개발자보다 대우가 좋은게 현실입니다. 좀 더 난이도 있는 개발을 할 수 있기 때문입니다.
실무에서는 펌웨어와 어플리케이션 개발자는 엄격히 구분되어있습니다. 신입사원 때 한번 정하면 평생 직업이 될 가능성이 높습니다. 따라서 신중한 선택이 필요하며 동부로봇과 함께하는 C언어 교육은 좋은 지침이 될 것입니다.

로봇에서 C 언어를 배운다는 것은 하드웨어를 알아야 하고, Firmware 를 익혀야하므로 좀더 많은 엔지니어적인 지식을 필요로 하는게 사실입니다. C 언어가 어려운게 아니라, 하드웨어와 펌웨어가 더 어렵다고도 볼 수 있습니다.
따라서, 로봇으로 익히는 C 언어는 좀더 포괄적인 문제가 내포합니다.
동부로봇은 추후에 로봇을 이용한 입문자용 C 언어 책을 편찬할 예정입니다. 가장 쉽고, Visual C++ 보다 더 실용적이고, 효과적인 책이 될 것입니다.

본 C언어 문법은 요약형으로 진행할 예정이며, 예시는 필요에 따라 Firmware 와 Visual C++ 모두 사용 될 것입니다. 그리고, C 언어 자체는 다른 C언어 입문서를 보는게 더 낫다는 판단으로 여기에서 다루는 C 문법은 도표형태나 종합적인 데이타를 다루고, 유저는 그것을 참고하는 방식으로만 쓰일 수 있게 할 예정입니다.
또한 DR-Visual Logic 의 예시와 함께 어떻게 C 문법과 연계되는지도 같이 설명할 예정입니다.

C 언어 훑어보기

◆ C 언어 예시

※ 동부로봇 AVR 예제

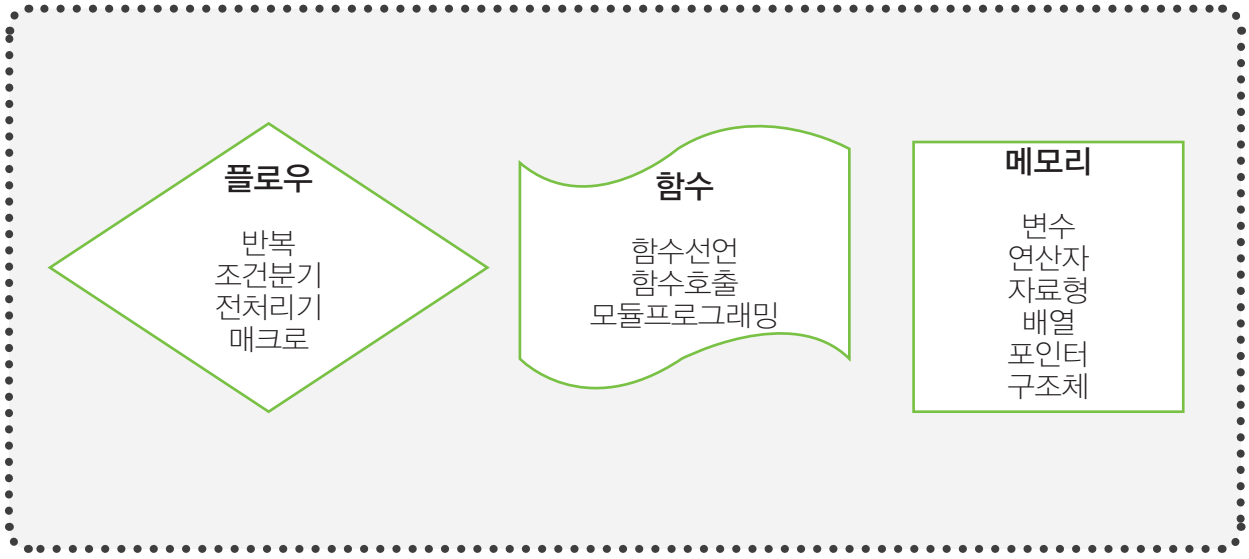
```
#include <avr/io.h>  //(1) 전처리기
#define BTN_MODE      0b00000001
int main(void)  // (2) 함수 - 기능정의
{
        unsigned char ucButton;   // (3) 변수 - 메모리 관리 - 포인터
        PORTB &= ~0b10000000;

        while(1) //(4) 반복
        {
                ucButton = (~PINA) & BTN_MASK;
                if(ucButton & BTN_OK){          //(5)조건분기
                        PORTB ^= 0b10000000;
                        _delay_ms(1);
                }
        }
        return 1;
}
```

위 예제는 Dongbu Robot AVR 소스 예제중에 일부를 발취한 것으로, C 프로그래밍 문법이 어떻게 구성되는지 간략히 요약되어있습니다.

C언어는 크게 두가지 요소를 가집니다.
바로 변수와 함수입니다.

(3) 변수는 데이터를 저장하고자 하는 목적으로 메모리를 관리하고, 그 메모리는 각각 주소가 존재하는데, 그 주소를 저장하는 것을 포인터라고 부릅니다. 포인터 또한 변수의 일종입니다. 흔히 C 언어하면 포인터에 대한 난해성이 많은 언어로 규정짓기도 합니다.
(2) 함수는 동작시키는 기능을 말하는 것으로서, 프로그래밍은 함수들의 결합이라고 말할 수 있습니다.
프로그래밍을 하는 이유는 컴퓨터가 인간이 할 수 있는 계산 능력을 훨씬 뛰어넘기 때문입니다. 그것을 가장 잘 활용하는 것이 바로, 제어와 분기, 반복입니다.
(4) 반복과 (5) 조건 분기는 C 언어뿐 아니라, 모든 프로그래밍 언어의 문법이 거의 동일하다고 보면 됩니다. 이 두가지 요소는 왜 프로그래밍을 하면 편리해지는지를 보여주는 가장 좋은 사례입니다.

◆ C 언어 요약



C 프로그래밍 언어는 프로그램의 흐름을 제어하는 플로우 와 메모리에 데이터를 관리하는 메모리 파트, 그리고 기능을 부여하는 함수로 구성됩니다.

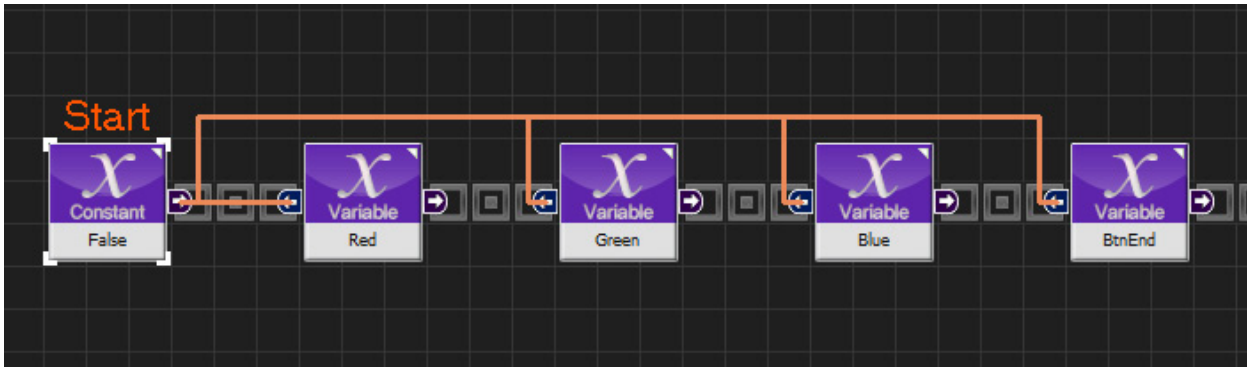
프로그램 작성 및 실행순서는 아래와 같습니다.

1) 프로그램 작성 합니다.
2) 컴파일 합니다.
3) 링크 : 연결합니다.
4) 실행파일이 생성됩니다.

변수

◆ 변수

ex) ledbutton.dts 중 일부



C-like 보기

```
1  void main()
2  {
3          Red=false
4          Green=false
5          Blue=false
6          BtnEnd=false
7          while( true )
8          {
9                  if( ( ( MPSU_ButtonStat == 0x04 ) && ( !BtnEnd ) ) )
10                 {
11                         Red=( !Red )
12                         BtnEnd=true
13                 }
14                 else
15                 {
16                 }
```

※ 변수란 데이터를 저장할 수 있는 메모리 공간에 붙여진 이름을 의미합니다.

변경이 가능합니다. 단, 상수는 변경이 불가능합니다.

※ 다양한 형태(자료형) 의 변수는 메모리 공간 쓰임새에 따라서 아래와 같이 간략히 나뉩니다.

정수형 : char, int, long 3

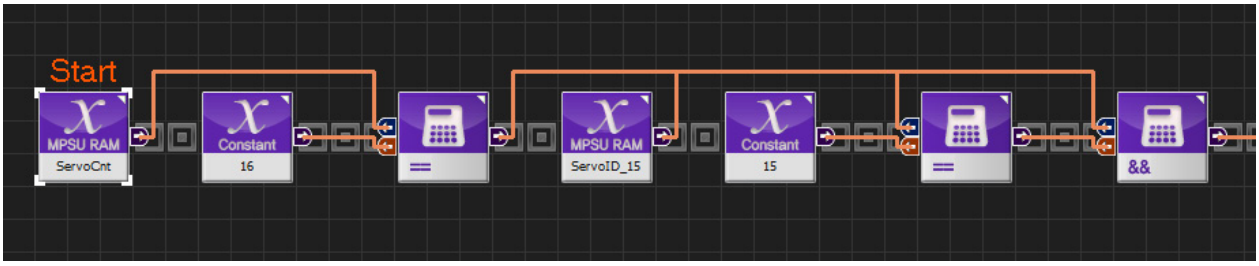
실수형 : float, double 3.1

※ 변수의 선언 및 대입 예시입니다.

```
Int main(void)
{
        Int val;    //int형 변수 val 의 선언
                        // → int 형 변수 val 이름 지어주고, 메모리 공간 할당
        Val = 20; //변수 val 에 20을 저장
```

◆ 변수

ex) ledbutton.dts 중 일부



C-like 보기

```
 9  void main()
10  {
11          if( ( ( MPSU_ServoCnt == 16 ) && ( MPSU_ServoID_15 == 15 ) ) )
12          {
13                  melody( 1 )
14                  SERVO_TorqCtrl[254]=96
15                  InitPosition16()
16                  delay( 2000 )
17                  CurrentMotion=-1
18                  MotionAfterStop=-1
19                  RmcEnd=false
20                  MtnReady=false
```

◆ 리터럴 상수의 기본 자료형

상수도 메모리 공간에 저장되기 위해서 자료형이 결정됩니다.

Char c = 'A' / 문자 상수 → 문자를 쓸 경우 char 형으로 인식하고, 메모리에 넣는다는 의미입니다.

Int I = 5 //정수상수

Double d = 3.15 // 실수상수

왼쪽은 변수, 오른쪽은 상수, 둘은 독립적인 관계입니다.

◆ 심볼릭(symbolic) 상수

이름을 지니는 상수입니다. 심볼릭 상수를 정의하는 방법은 아래와 같습니다.

const 키워드 를 통한 변수의 상수화

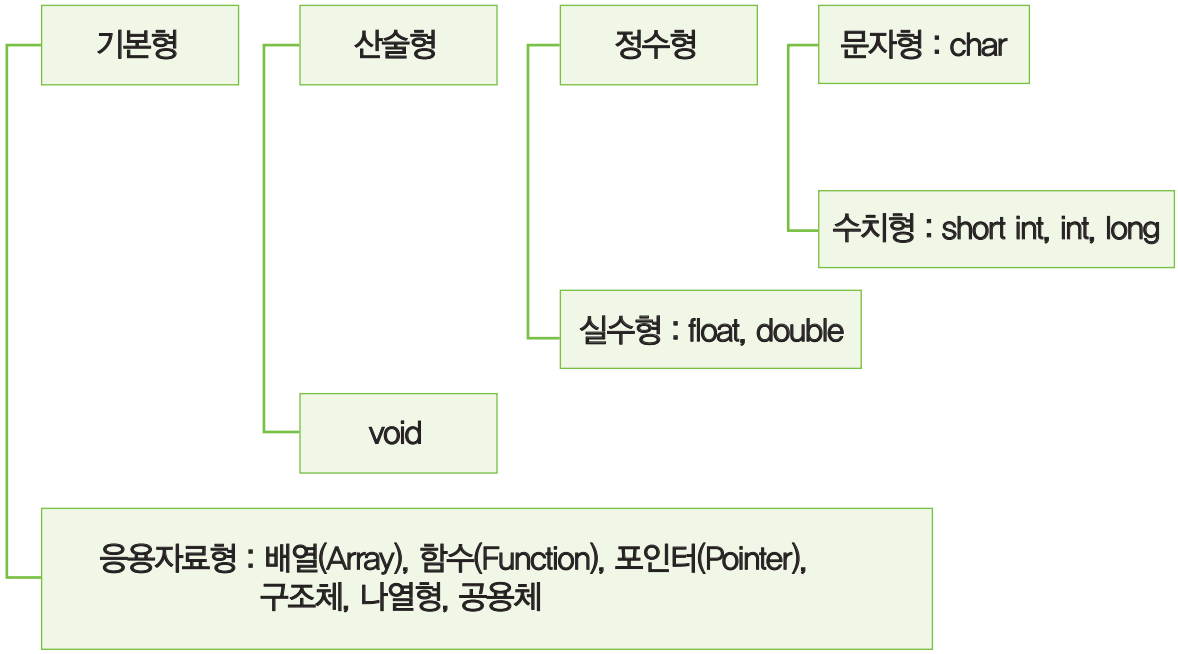
const int MAX = 100 // 변수가 상수가 됨

MAX = 102 // 허용 안됨, MAX 는 상수이기 때문에 상수 변수 선언시 대문자로 써주는게 관례입니다.

◆ 접미사에 따른 다양한 상수의 표현

접미사	자료형	사용 예
u or U	unsigned int	304U
l or L	long	304L
ul or UL	unsigned long	304UL
f or F	float	3.15F
l or L	long double	3.15L

◆ 자료형 종류



◆ 자료형 범위

구분	자료형	범위	byte
문자형	char	–128~127	1
	unsigned char	0~255	
정수형	int	–2147483648 ~ 2147483637	4
	unsigned int	0 ~ 4294967295	
	short	–32768 ~ 32767	2
	unsigned short	0 ~ 65535	
	long	–2147483648 ~ 2147483637	4
	unsigned long	0 ~ 4294967295	
실수형	float	±3.4 x 10–38 ~ ±3.4 x 1038	4
	double	±1.7 x 10–308 ~ ±1.7 x 10308	8

◆ 확장문자열

표기	의미	기능
₩n	New line	줄바꿈을 한다.
₩t	Tab	다음 탭 위치로 옮긴다.
₩b	Back space	현재 줄의 첫번째 칸으로 간다
₩r	Carriage return	한 칸 뒤로 옮긴다.
₩a	Alert	경보음을 낸다
₩”	” 표시 출력	큰 따옴표를 출력
₩’	’ 표시 출력	작은 따옴표를 출력

◆ printf 변환 문자열

구분	변환문자열	출력형태
정수형	%d	부호 있는 10진수
	%u	부호 없는 10진수
	%o	부호 없는 8진수
	%x, %X	부호 없는 16진수
실수형	%lf, %f	부호 있는 소수점(double, float)
	%le, %e	부호 있는 지수형(double, float)
문자형	%c	하나의 문자
문자열	%s	문자열

◆ printf 변환 문자열 옵션

변경자	의 미	
숫자	출력 자릿수 확보	
	%10d	화면에 10자리를 확보 오른쪽 정렬
	%–10d	화면에 10자리를 확보 왼쪽 정렬
	%10.2lf	화면에 10자리를 확보, 소수점 이하 2자리 까지 출력
h	Short 형임을 의미 예) %hu	
L , l	Long형임을 의미 예) %ld	

◆ Scanf 변환 문자열

구분	변환문자열	출력형태
정수형	%d, %u	10진수
	%o	8진수
	%x	16진수
실수형	%lf, %f	소수점 또는 지수형
	%le, %e	부호 있는 지수형(double, float)
문자형	%c	하나의 문자 (char 형 변수)
문자열	%s	문자열 (char 배열)

◆ C 언어의 키워드 와 설명 1

1.asm : 인라인 어셈블리 코드를 나타내는 키워드
2.auto : 기본적인 변수의 저장방식을 나타내는 키워드
3.break : for,while,switch,do...while문을 조건없이 마치는 명령
4.case : switch문 내에서 사용되는 명령
5.char : 가장 간단한 데이터형
6.const : 변수가 변경되지 않도록 방지하는 데이터 지정자
7.continue : for,while,switch,do...while문을 다음 반복동작으로 진행시키는 명령
8.default : case문에 일치하지 않는 경우를 처리하기 위해 switch문에서 사용되는 명령
9.do : while문과 함께 사용되는 순환명령.순환문은 최소한 한번 실행됨.
10.double : 배정도 부동 소수형값을 저장할 수 있는 데이터형
11.else : if문이 FALSE로 평가될 때 실행되는 선택적인 문장을 나타내는 명령
12.enum :변수가 특정값만을 받아들이도록 해주는 데이터형
13.extern : 변수가 프로그램의 다른 부분에서 선언된다는 것을 알려주는 데이터 지정자
14.float : 부동 소수형 숫자값을 저장하기 위해 사용되는 데이터형
15.for : 초기화,증가,조건 부분을 가지는 순환명령

◆ C 언어의 키워드 와 설명 2

16.goto :　　정의되어 있는 레이블로 이동시키는 명령
17.if : TRUE/FALSE의 결과에 따라 프로그램의 제어를 변경하는데 사용되는 명령
18.int : 정수형 값을 저장하는 데 사용되는 데이터형
19.long : int형보다 큰 정수형 값을 저장하는 데 사용되는 데이터형
20.register : 가능하다면 변수를 레지스터에 저장하도록 지정하는 저장형태 지정자
21.return : 현재의 함수를 마치고 호출한함수로 프로그램의 제어를 돌려주는 명령.
함수에서 값을 돌려주기 위해서 사용됨.
22.short : 정수형 값을 저장하는 데 사용되는 데이터형. 자주 사용되지는 않지만 대부분의
컴퓨터에서 int형과 동일한 크기를 가짐.
23.signed : 변수가 양수와 음수값을 모두 저장할 수 있다는 것을 지정하기 위해서 사용되는
지정자
24.sizeof : 항목의 크기를 바이트 단위로 알려주는 연산자
25.static :컴파일러가 변수의 값을 보존해야 한다는 것을 지정하는데 사용되는 지정자
26.struct :C에서 어떤 데이터형의 변수를 함께 결합시키는 데 사용되는 키워드
27.switch :여러 가지 조건을 통해서 프로그램의 흐름을 변경하는 데 사용되는 명령.
case문과 함께 사용됨.
28.typedef : 이미 존재하는 변수와 함수의 형태를 새로운 이름으로 변경하는 데 사용되는
지정자
29.union :여러 개의 변수가 동일한 메모리 영역을 공유하도록 해주는 데 사용되는 키워드
30.unsigned : 변수가 양수값만으르 저장할 수 있다는 것을 지정하는 데 사용되는 지정자.
31.void : 함수가 어떤 값을 돌려주지 않거나, 또는 사용되는 포인터가 범용 포인터이거나 모든 데이터
형을 지적할 수 있다는 갓을 지정하는 데 사용되는 키워드
32.volatile : 변수가 변경될 수 없다는 것을 지정하는 지정자.
33.while : 지정된 조건이 TRUE로 평가되는 한 계속해서 포함된 문장을 실행하는 순환문

◆ 문자형

문자 표현을 위하여 ASCII 코드라는 것이 등장합니다.

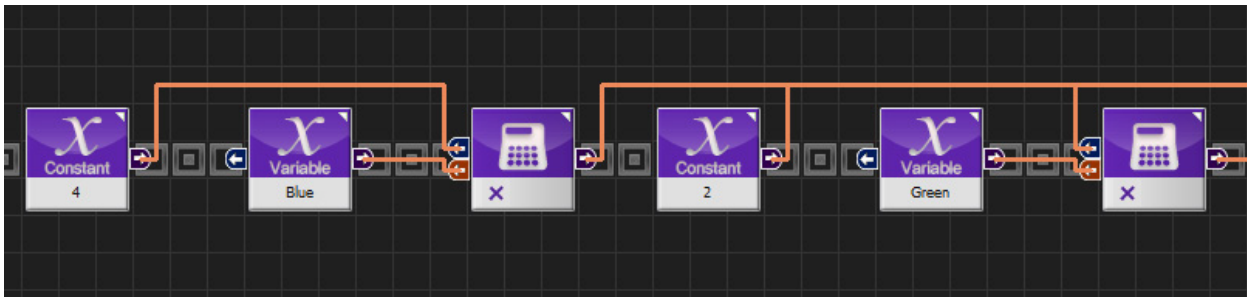
미국 표준 협회 (ANSI) 에 의해 정의, 컴퓨터를 통해서 문자를 표현하기 위한 표준으로서 컴퓨터는 문자를 표현하지 못하기 때문에, 문자와 숫자의 연결 관계를 정의하였습니다. 문자 A 는 숫자 65, 문자 B 는 숫자 66 으로 매칭시킵니다.
문자, 컴퓨터는 2진수밖에 받지 못하기 때문에, 컴퓨터와 사람사이에 약속을 하기 시작하였습니다. 사람은 문자 표현 원하고, 문자와 숫자 사이에 매핑 관계를 유지하기를 원합니다. 예를 들어 A=65, B = 55 라고 약속, 사람이 A 입력, 컴퓨터는 65라고 저장합니다.

ANSI 에서 정의되었고, 숫자와 문자 매핑은 아래와 같습니다.

10진	16진	문자	10진	16진	문자	10진	16진	문자	10진	16진	문자	10진	16진	문자
0	0	Null	47	2F	/	68	44	D	89	59	Y	110	6E	n
7	7	Bell	48	30	0	69	45	E	90	5A	Z	111	6F	o
8	8	BS	49	31	1	70	46	F	91	5B	[	112	70	p
9	9	Tab	50	32	2	71	47	G	92	5C	₩	113	71	q
10	A	LF	51	33	3	72	48	H	93	5D	]	114	72	r
13	D	CR	52	34	4	73	49	I	94	5E	^	115	73	s
32	20	공백	53	35	5	74	4A	J	95	5F	_	116	74	t
33	21	!	54	36	6	75	4B	K	96	60	`	117	75	u
34	22	"	55	37	7	76	4C	L	97	61	a	118	76	v
35	23	#	56	38	8	77	4D	M	98	62	b	119	77	w
36	24	$	57	39	9	78	4E	N	99	63	c	120	78	x
37	25	%	58	3A	:	79	4F	O	100	64	d	121	79	y
38	26	&	59	3B	;	80	50	P	101	65	e	122	7A	z
39	27	'	60	3C	<	81	51	Q	102	66	f	123	7B	{
40	28	(	61	3D	=	82	52	R	103	67	g	124	7C	|
41	29	)	62	3E	>	83	53	S	104	68	h	125	7D	}
42	2A	*	63	3F	?	84	54	T	105	69	i	126	7E	~
43	2B	+	64	40	@	85	55	U	106	6A	j	127	7F	Del
44	2C	,	65	41	A	86	56	V	107	6B	k			
45	2D	-	66	42	B	87	57	W	108	6C	l			
46	2E	.	67	43	C	88	58	X	109	6D	m			

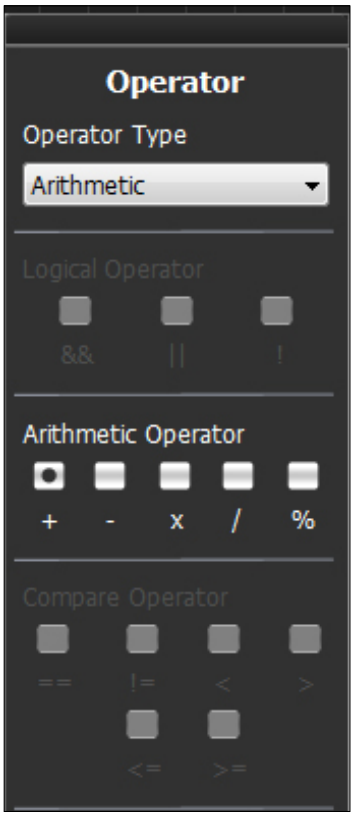
◆ 연산자

ex) ledbutton.dts 중 일부



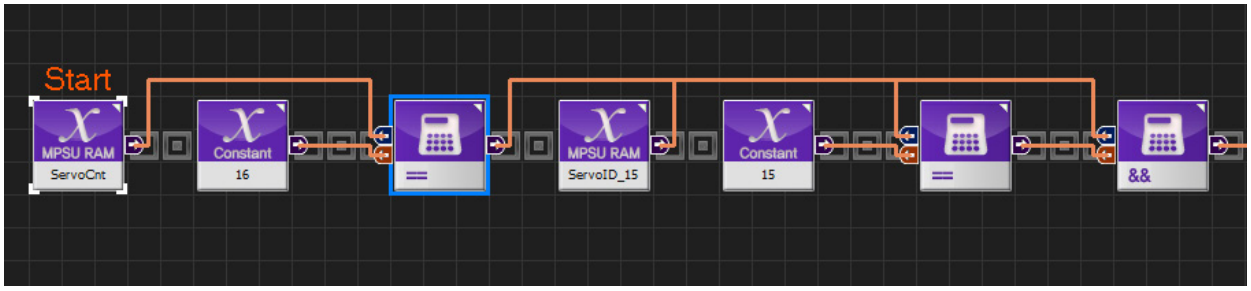
C-like 보기

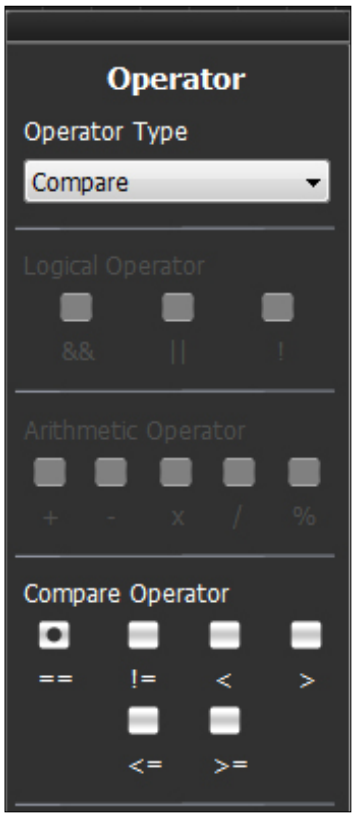
```
33    led( ( ( ( 4 * Blue ) + ( 2 * Green ) ) + Red ) )
34    if( ( ( MPSU_ButtonStat == 0x00 ) && BtnEnd ) )
35    {
36            BtnEnd=false
37    }
38    else
39    {
40    }
```

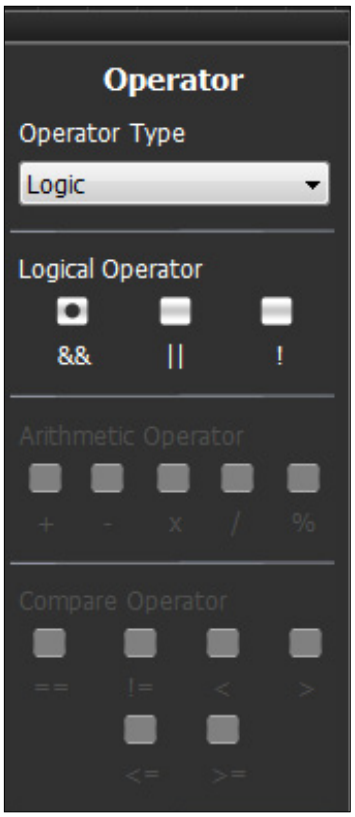


기능별 종류	연산자
산술 연산자	+ – * / %
부호 연산자	+ –
대입 연산자	= 복합 대입 연산자
관계 연산자	== != 〈= 〈 〉= 〉
증감 연산자	++ ––
포인터 연산자	* & []
구조체 연산자	. –〉
논리 연산자	|| && !
비트 연산자	| & ~ 〉〉 〈〈
삼항 조건 연산자	? :
쉼표 연산자	,
sizeof 연산자	sizeof
캐스트 연산자	(type) type()
괄호 연산자	()
C++ 연산자	new delete :: .* –〉*

ex) Remocon16.dts 중 일부







C-like 보기

```
 9  void main()
10  {
11          if( ( ( MPSU_ServoCnt == 16 ) && ( MPSU_ServoID_15 == 15 ) ) )
12          {
13                  melody( 1 )
14                  SERVO_TorqCtrl[254]=96
15                  InitPosition16()
16                  delay( 2000 )
```

◆ 연산자 우선순위

순위	명칭	연산자 종류	결합방향
1	1차 연산자	() {} . –>	→
2	단항 연산자	+ – ! ~ (type) sizeof ++ –– & *	←
3	승제	* / %	→
4	가감	+ –	
5	Shift 연산자	〈〈 〉〉	
6	관계연산자	〈 〉 〈= 〉=	
7	등가 연산자	== !=	
8	비트 곱	&	
9	비트 차	^	
10	비트 합	|	
11	논리곱	&&	
12	논리합	||	
13	조건 연산자	? :	→
14	대입연산자	= += –= *= /= %= 〈〈= 〉〉=&= |=	←
15	순차연산자	,	→

◆ 기억클래스

데이터 영역

힙(Heap) 영역
런타임에 메모리 할당

스택 영역
컴파일 타임에 메모리 할당

◆ 메모리 영역

지정자		유효범위	생존기간	메모리 위치	자동초기화
auto		선언된 블록 내	블록 종료 시	Stack	X
extern		한 모듈 전체 프로그램	프로그램 종료 시	(비)초기화 영역	O
static	내부 static	선언된 블록 내	프로그램 종료 시	(비)초기화 영역	O
	외부 static	선언된 모듈 내			
register			블록 종료 시	CPU 내 레지스터	X

배열, 포인터, 구조체

DR-Visual Logic 에는 C언어에서 다루는 배열과 포인터, 구조체는 다루지 않습니다. 그래픽 로직을 구성하기 위해서 복잡한 문법을 최소화 시키기 위함입니다.
따라서 이 파트에서는 Visual Logic 과 대응없이 배열,포인터, 구조체를 간략히 설명하겠습니다.

◆ 1차원 배열

※ 배열이란

- 둘 이상의 변수를 동시에 선언하는 효과를 지닙니다.
- 많은 양의 데이터를 일괄적으로 처리해야 하는 경우에 유용합니다.
- 지역적 특성을 지닐 수도 있고, 전역적 특성을 지닐 수도 있습니다.

※ 배열 선언

```
int  array  [10];
```

배열요소자료형 배열이름 배열길이

→ Int 형 데이터 10개를 저장할 수 메모리 공간을 할당하고, 이름을 array 라고 붙여줍니다.

- 배열 길이 : 배열을 구성하는 변수의 개수 (반드시 상수를 사용)
- 배열 요소 자료형 : 배열을 구성하는 변수의 자료형
- 배열 이름 : 배열에 접근할 때 사용되는 이름

◆ 2차원 배열

※ 다차원 배열의 예

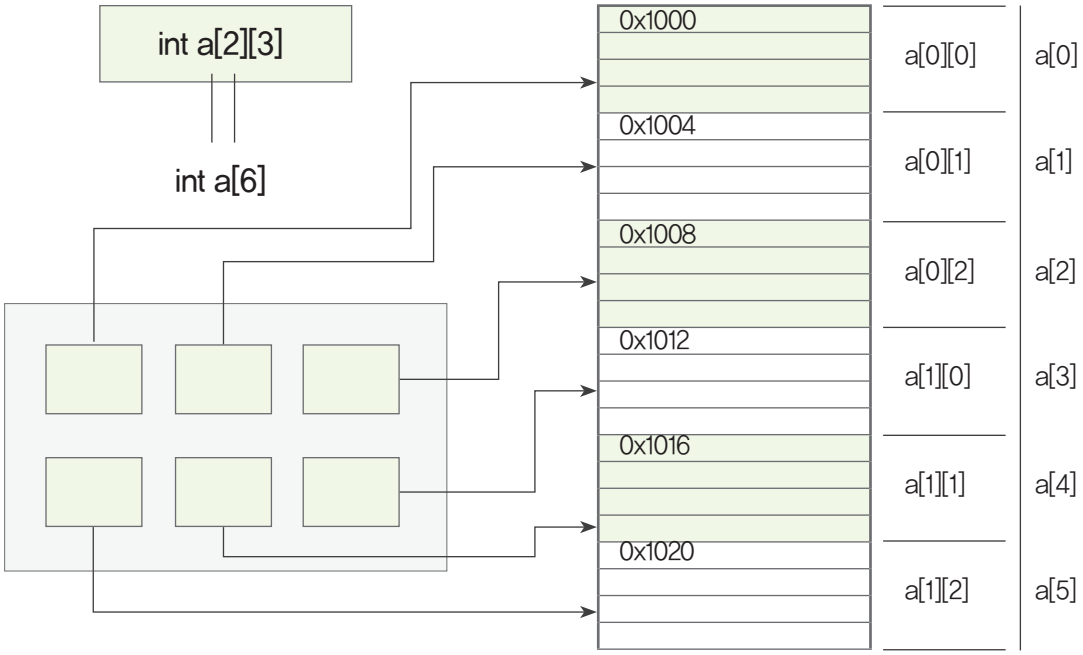
Int arr[100]	1차원배열
Int arr[10][10]	10x10, 2차원배열
Int arr[5][5][5]	5x5x5, 3차원배열

1차원은 선, 2차원은 면, 3차원은 직육면체입니다.
보통 다차원배열은 2차원 배열을 의미합니다.

※ 다차원 배열의 실제 메모리 구성

1차원 배열과 동일하나, 접근 방법을 2차원적으로 해석할 뿐입니다.
Int a[2][3]; → 2차원 접근 a[0][0]. 세로가 2, 가로가 3
Int a[6]; → 1차원 접근 a[0]~a[5] → 반드시 2차원적으로 해석하기 바랍니다.
문제자체가 2차원적인 경우가 많습니다.
내가 해결해야할 문제는 2차원인데, 실제 구조는 1차원적이기는 하지만 2차원적으로 하는 습관을 가져야 합니다.

◆ 다차원 배열의 실제 메모리 구성



※ 2차원 배열 ! 선언과 동시에 초기화

중괄호 안에 또 다른 중괄호 → 2차원 이상 초기화 할 때 사용합니다.

case 1 : 행 단위로 모든 요소들을 초기화

case 2 : 행 단위로 일부 요소들만 초기화

int somang[3][3]={{1,2,3},{4,5,6,},{7,8,9}};

첫번째 중괄호 → 첫번째 행 초기확, 두번째, 세번째

int somang[3][3]= {{1},{4,5},{7,8,9}};

행의 일부만을 초기화 → 첫번째 행 초기화 요소리스트는 1개뿐, 왼쪽부터 넣고, 나머지는 0으로 채워줍니다.

Case 3 : 내부 중괄호 없이 초기화

int somang[3][3]={1,2,3 ,4,5,6, 7};

처음부터 세개 채우고, 두번째 3개 채우고, 나머지 1개 채웁니다.

◆ 포인터

※ 포인터란 무엇인가?

– 포인터와 포인터 변수

포인터라고하면 변수를 의미합니다. 포인터변수와 같은 개념입니다.

– 메모리의 주소값을 저장하기 위한 변수를 말합니다. 일반적인 변수는 데이터를 저장하기 위한 것이지만,

데이터가 아닌 주소값을 저장하면 포인터입니다.

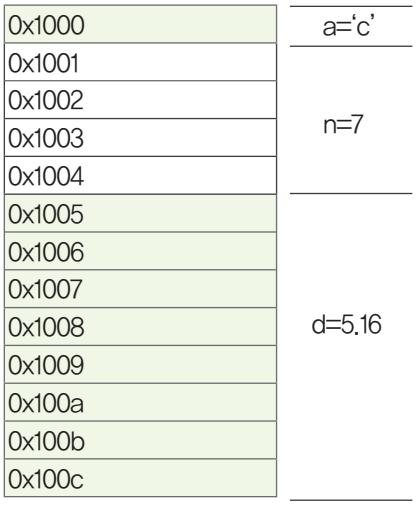
– “포인터”를 흔히 “포인터 변수”라고 합니다.

포인터도 경우에 따라 상수가 될 수도 있습니다. 하지만 상수는 매우 적습니다. 그 상수의 절반은 상수화 변수입니다.

– 주소값과 포인터는 다릅니다.

◆ 메모리 구조 표현 방식

```
int main(void)
{
   char a='c';
   int n=4;
   double d=5.16
```



※ 포인터란 주소값을 저장하기 위한, 변수입니다.

8비트 → 1바이트 주소값

16비트 → 2바이트 주소값

32비트 → 4바이트 주소값.

역사적으로 포인터의 주소값은 변해왔습니다. 오늘날은 4바이트로 표현됩니다. → 32비트기반은 4바이트 입니다.

문제자체가 2차원적인 경우가 많습니다.

내가 해결해야할 문제는 2차원인데, 실제 구조는 1차원적이기는 하지만 2차원적으로 하는 습관을 가져야 합니다.

◆ 포인터와 배열의 관계

– 배열 이름은 첫 번째 요소의 주소값을 나타냅니다.

Int a[5]={0,1,2,3,4}

Ox1000

a[0] : a라는 배열의 0번째 요소, &a[0] : 0번째 인덱스에 해당하는 주소값을 반환 → Ox1000

Ox1004

Ox1008

….

a 라는것은 배열 첫번째 요소의 주소를 나타냅니다.

◆ 포인터 연산과 배열

※ arr이 포인터이거나 배열이름인 경우

```
arr[i] == *(arr+i)
```

*ptr++ // *ptr의 값을 구하고 ptr을 1증가시킵니다.

(*ptr)++ // 먼저 *ptr의 값을 구한 후 *ptr의 값을 1 증가시킵니다.

*++ptr // ptr을 1증가시킨 후 *ptr을 구합니다.

++*ptr // *ptr의 값을 1증가 시킨 후 , 그 값을 결과로 취합니다.

◆ 함수포인터

함수의 이름은 함수의 위치를 가리키는 포인터입니다.

함수이름의 포인터 타입을 결정짓는 요소는 리턴형과 전달인자입니다.

```
int (*fun)(int);  // int형 매개변수 1개 받습니다.
                  // fun은 포인터 이름입니다.
                  // 리턴형은 int형입니다.
```

◆ void 포인터

어떤 형의 주소값 이라도 저장할 수 있는 포인터 입니다.

예) char c='a';

```
int n=10;
void* vp;
vp = &c;
vp = &n;
```

포인터 연산이나 값을 변경, 참조하는 것 등을 할 수 없습니다.

메모리

◆ 메모리 동적할당

※ 스택, 힙 그리고 데이터 영역

– 프로그램의 실행을 위해 기본적으로 할당하는 메모리 공간입니다.

– 컴파일 타임에 함수에서 요구하는 스택의 크기 결정되어야 합니다.

– OS 가 관리해주는 메모리 구조이고, C++, JAVA 등도 실행되면, 아래와 같이 메모리 관리됨

※ 데이터 영역 : 전역변수, Static 변수 저장하기 위한 장소입니다.

※ 스택 영역 : 지역변수, 매개 변수를 저장하기 위한 장소입니다.

– 컴파일 타임에 함수에서 요구하는 스택의 크기 결정되어야 합니다.

– 반드시 컴파일 타임에 결정됩니다.

● malloc함수

heap영역에 메모리를 할당합니다.

```
void* malloc(size_t size)
```

● free함수

heap영역에 할당된 메모리를 해제합니다

```
void free(void* ptr)
```

구조체

◆ 구조체의 정의

– 하나(보통 둘) 이상의 기본 자료형을 기반으로 사용자 정의 자료형을 만들 수 있는 문법 요소입니다.

```
struct point   // point 라는 이름의 구조체 선언
{
Int x;       // 구조체 멤버 int x
Int y;        // 구조체 멤버 int y.
};
```

구조체 선언과 변수 선언을 동시에!!

```
struct 구조체명 {
        구조체 멤버선언;
        구조체 멤버선언;
              :
} 구조체 변수명1, 구조체변수명2…;
```

구조체 맴버

구조체 선언과 변수 선언을 따로!!

```
struct 구조체명 {
        구조체 멤버선언;
        구조체 멤버선언;
              :
};
struct 구조체명 구조체변수명1, 구조체변수명2…;
```

◆ 구조체의 특징

구조체 변수는 함수의 전달인자로 사용가능합니다.

구조체는 기본자료형과 같이 배열을 선언하여 사용할 수 있습니다.

```
예) struct phone list[10];  // 구조체 배열의 선언
    list[3].age; // list배열의 네번째배열요소의
                 // age멤버 참조
```

◆ 구조체 포인터

● 구조체는 마치 일반변수와 같이 주소 연산이 가능하고 구조체 변수 전체를 가리키는 포인터를 가질 수 있다.

예) 구조체 포인터 변수의 선언과 초기화

```
struct score *sp = &a;
```

예) 구조체 포인터를 이용한 멤버 참조

```
(*sp).kor;
```

◆ 간접 멤버 참조 연산자

● 간접 멤버 참조 연산자 : (–〉)

예) 간접 멤버 참조 연산자 사용법

```
(*sp).kor  == sp–〉kor
```

예) lp[i].name // 배열표현

```
== (*(lp+i)).name;   // 포인터표현
== (lp+i)–〉name;   //  간접멤버 참조 연산자를 이용한 표현
```

◆ 자기참조 구조체

● 개념

구조체 멤버 중에서 자기 자신을 가리키는 포인터를 가지고 있는 구조체

예)

```
struct list{
        int l;
        struct list *ptr;
} s1, s2, s3;

s1.prt=&s2;
s2.prt=&s3;
s3.prt=&s1;
```

◆ 공용체

구조체 선언과 변수 선언을 동시에!!

```
union 공용체명 {
        공용체 멤버선언;
        공용체 멤버선언;
              :
} 공용체 변수명1, 공용체변수명2…;
```

구조체 선언과 변수 선언을 따로!!

```
union 공용체명 {
        공용체 멤버선언;
        공용체 멤버선언;
              :
};
union 공용체명 공용체변수명1, 공용체변수명2…;
```

◆ typedef

구조체, 공용체, 열거형과 같은 응용자료형을 사용할 때 길어지는 선언문을 간결하게 작성할 수 있도록 자료형의 이름을 재정의한다.

```
예) struct student{
      int num;
      double grade;
    };
    typeof struct student Student;
```

◆ 열거형

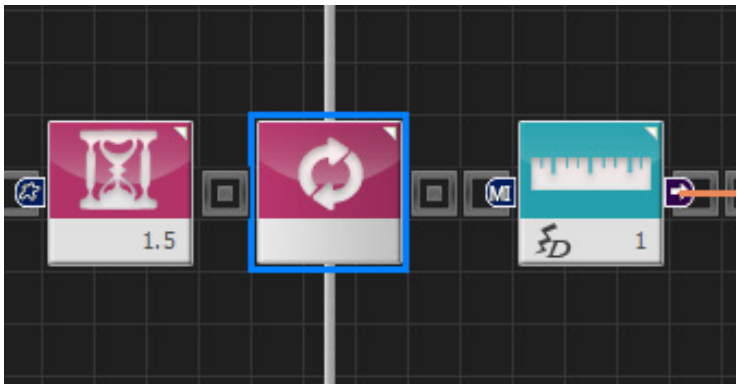
- 개념
  - 기억공간에 저장될 데이터의 집합을 정의한다.
  - 열거형 멤버들은 정수형 상수로 취급 당한다.
  - 프로그램 가독성을 높일 수 있다.
  - 특정 정수값에 의미를 부여할 수 있다.

```
예)  enum color {RED=1. GREEN=3, BLUE=5};
       enum color {sun=5, mon, tue, wed=10}
```

반복과 분기

◆ 반복

ex) digital.dts 중 일부



C-like 보기

```
1  void main()
2  {
3          SERVO_TorqCtrl[254]=96
4          motionready( 0 )
5          delay( 1500 )
6          while( true )
7          {
8                  if( ( MPSU_ADCType1 == 2 && MPSU_ADCVal1 == 1 ) )
```

반복문이란 ?

※ 반복문의 기능

– 특정 영역을 특정 조건이 만족하는 동안에 반복 실행하기 위한 문장입니다.

※ 세가지 형태의 반복문

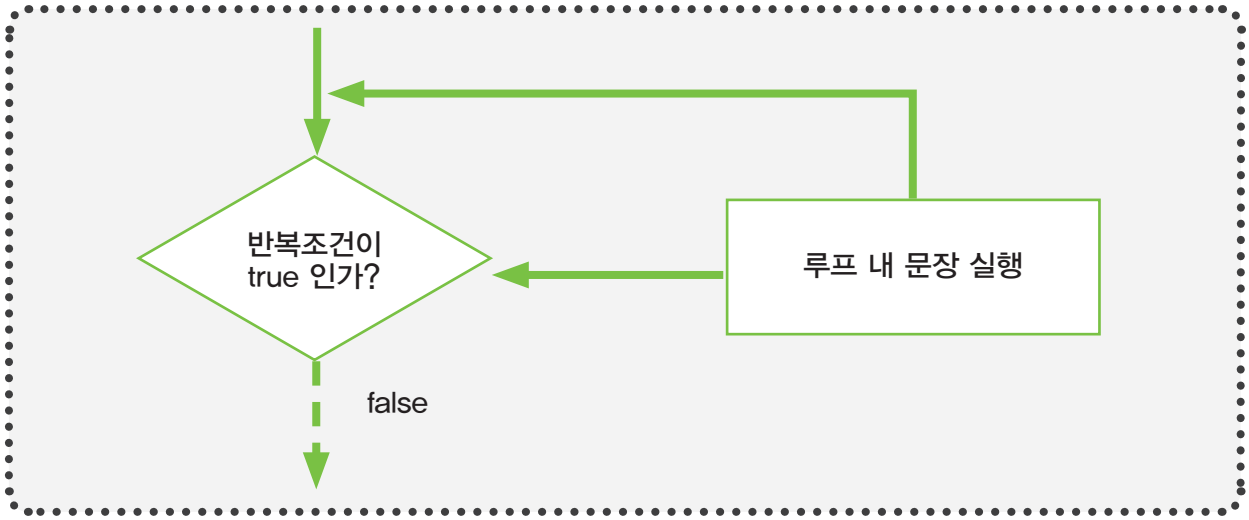
– while 문에 의한 반복

– do~while 문에 의한 반복

– for 문에 의한 반복

한가지만 다 이해해도 모두 잘 이해할 수 있습니다.

while 문의 순서도

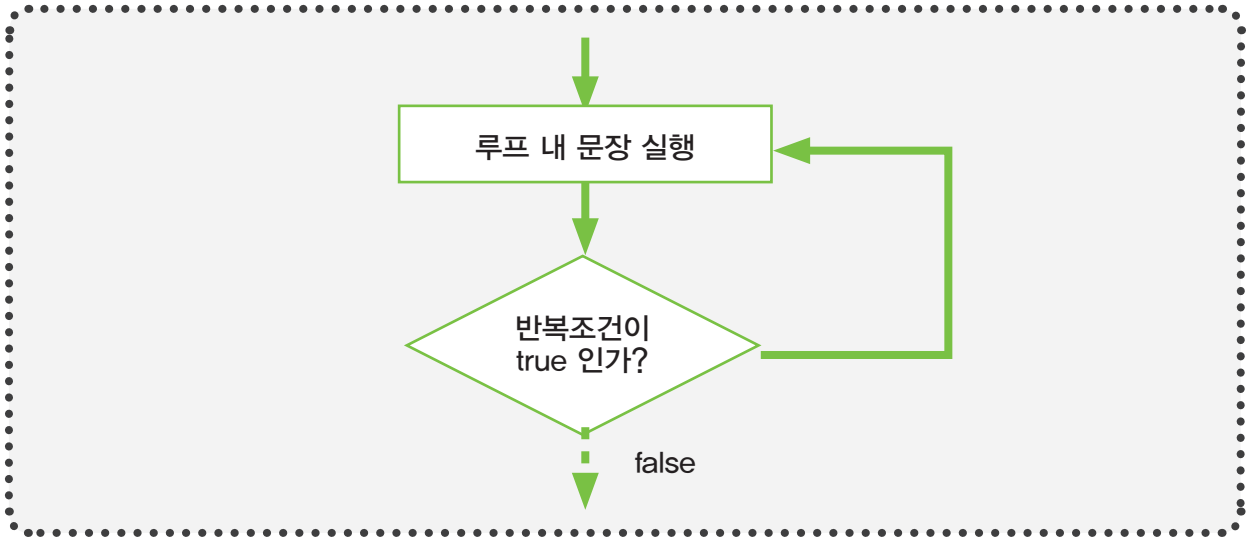


일반형식

```
While(조건식) {
        반복한 문장;
}
```

- 선조건 비교 순환문입니다. (조건이 참이 아닐 경우 한번도 실행하지 못할 수 있습니다.)
- 조건식에 0(zero)가 아닌 값이 오면 C는 참으로 인식하여 무한 loop에 빠지므로 반드시 탈출문을 반복 블록 안에 포함시켜야 합니다.

do~while 문의 순서도

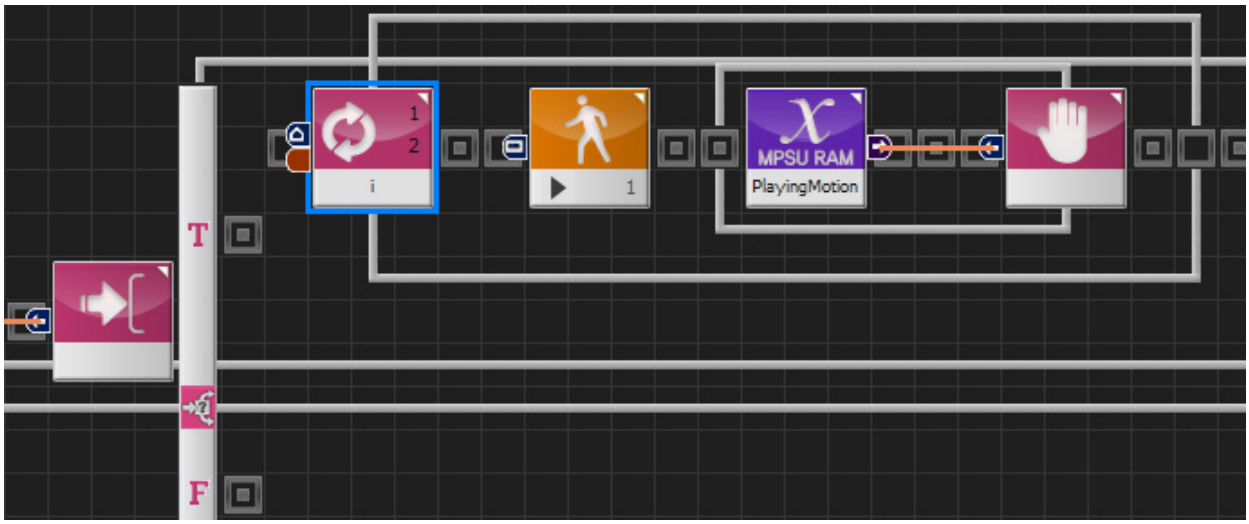


일반형식

```
do {
        반복한 문장;
} while(조건식);
```

- 후조건 비교 순환문이다. (조건이 참이 아닐 경우라도 한번은 실행한다.)

ex) digital.dts 중 일부



C-like 보기

```
if( ( MPSU_ADCType1 == 2 && MPSU_ADCVal1 == 0 ) )
{
        for( i = 1 ~ 2 )
        {
                motion( 1 )
                waitwhile( MPSU_PlayingMotion )
```

for 문

일반형식

```
for (초기식 ; 조건식 ; 증감식) {
        반복할 문장;
}
```

● 초기식
– Loop제어변수를 초기화 한다.(처음 한번만 수행)

● 조건식
– Loop제어변수의 범위를 검사하여 반복여부를 결정한다.

● 증감식
– Loop제어변수를 증가 또는 감소한다.

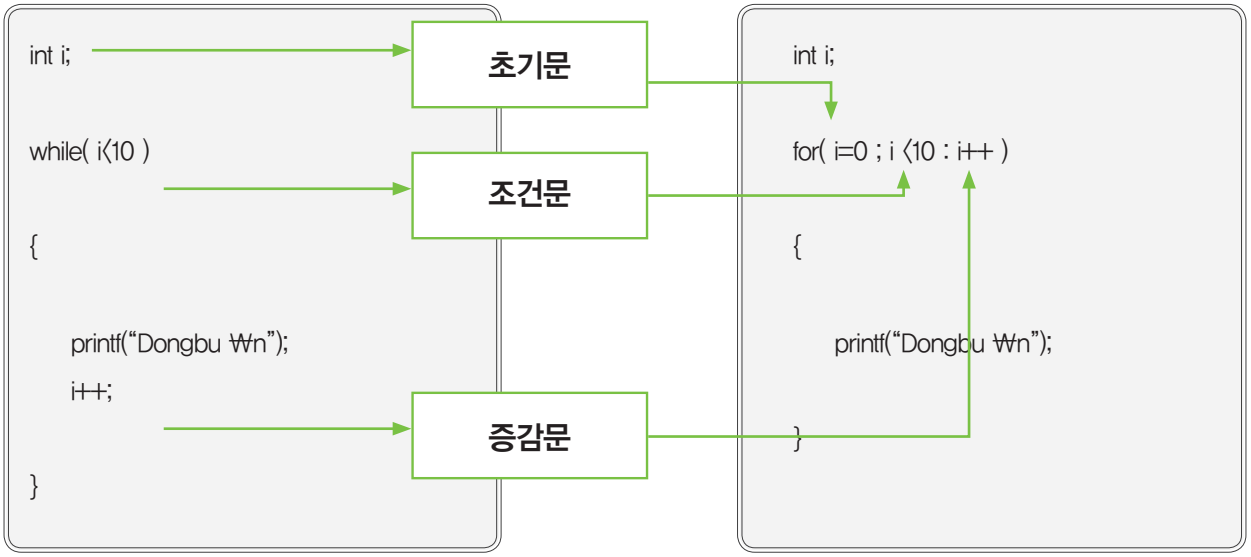
다중 for 문

```
for (초기식 ; 조건식 ; 증감식) {
        for (초기식 ; 조건식 ; 증감식) {
                반복할 문장 1;
        }
        반복할 문장 2;
}
```

● break文
– 반복문을 빠져 나올 때 사용하는 키워드

● continue文
– 돌던 루프의 남아있는 부분을 그냥 건너뛰는 것

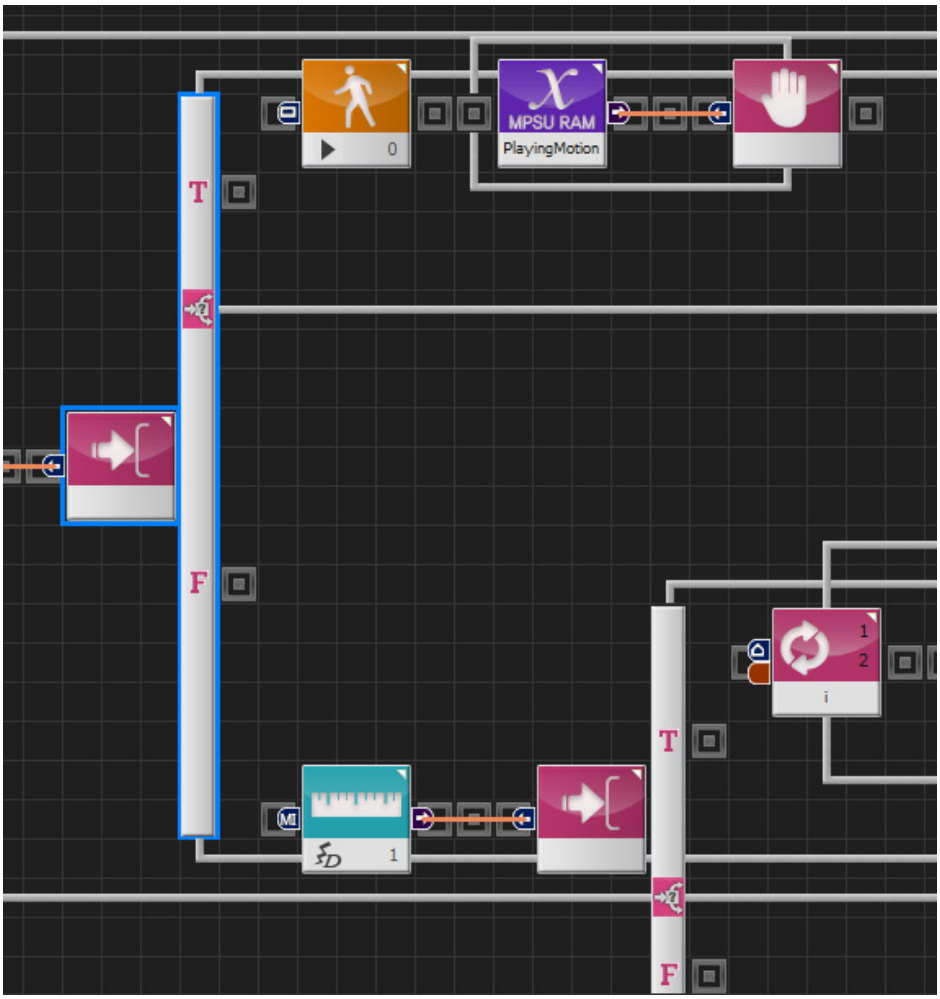
for 와 while 문 비교




```
for(i=0;i<n+1;i++)
        total+=i;

 while 문으로 바꾸면

i=0;
while(i<n+1)
{
        total+=i;
        i++;
}
```

◆ 조건분기

ex) digital.dts 중 일부



C-like 보기

```
8   if( ( MPSU_ADCType1 == 2 && MPSU_ADCVal1 == 1 ) )
9   {
10          motion( 0 )
11          waitwhile( MPSU_PlayingMotion )
12  }
13  else
14  {
15          if( ( MPSU_ADCType1 == 2 && MPSU_ADCVal1 == 0 ) )
16          {
```

◆ 조건 분기문

조건분기문은 if, if else, switch case 등 세가지 형태이지만, 기능은 동일하기 때문에 visual logic 에서는 if else 로보 표현했습니다.

◆ if

if문에 의한 조건적 실행 : 조건이 만족되는 경우에 한해서 실행합니다.

if(실행의조건) → true 라고 가정하면
{ – → 실행하고자 하는 문장이 한 문장일 때 중괄호 생략이 가능합니다.
실행하고자 하는 내용
}

```
int main(void)
{
  int val;
  printf("정수를 하나 입력하세요")
  scanf(%d, &val);

  if(val<0) //val <0 이 true이면…
  {
     printf("입력값은 0보다 작다"
  } → 문장이 둘 이상이면 반드시 넣어줘야함
  if(val>0)
     printf("입력값은 0보다 크다"
  if(val==0)
     printf("입력값은 0이다."

  return 0;
}
```

만약 1 을 입력하면, 두번째에서 True 이다, 따라서 세번째 조건 검사는 할 필요가 없습니다.
→ 이런 문제점을 해결하기 위해 else 를 넣습니다.

◆ if~else

앞조건이 만족하는데도, 그 아래 문자의 조건을 검사하게 됩니다. . 불필요한 비교연산을 많이 하게되고, 그것을 해결하기 위해서 if else 를 씁니다.

if(조건)
{
  조건 만족시 실행 일명 "이거"
}
else
{
  조건 불만족시 실행 일명"저거"
}

◆ if, else if, else

if(조건 A) → case1: 조건 A가 만족 “요거”실행후 마지막 else 까지도 완전히 건너뜁니다.

{

조건 A 만족시 실행 일명 ”요거”

}

else if(조건 B) → case2: 조건 A가 만족 하지 않았다. 따라서 조건 B 가 만족되는지 확인합니다.

조건B는 만족!”이거”실행후 마지막 else 까지도 완전히 건너뜁니다.

{

조건 B만족시 실행 일명 “이거”

}

else if(조건 C)

조건 C 만족시 실행 일명 ”그거”

}

else → case3: 조건 A,B,C가 모두 만족되지 않는다. Else 문 안에 있는 “저거”실행합니다.

{

조건 ABC 불만족시 실행 일명 “저거”

}

예를 들어 조건 B가 만족된다면, A는 건너뛰고, B 를 실행한후 나머지는 다 건너띄어
버립니다.
만약 조건 C 가 만족한다면, else 문장은 보지도 않고 건너뜀뜁니다.
→ 불필요한 연산을 극복할 수 있습니다.

◆ switch ~ case

int 형 char 형 인자를 한 개 전달할 수 있음

switch(n)

Case1: → n 이 1인 경우

Break;

Case2:

Break;

Case3:

Break;

default:

n=2 인 경우, case1 은 아니므로 건너뜁니다.
case2는 맞으니까 실행합니다. break 가 있으면 break 문은 switch 를 빠져나가라는 의미입니다.
만약 break 가 없으면, 2부터 그 아래는 모두 실행시킵니다.

◆ 조건분기 전체 요약

if, if~else, if~else if~else, switch~case

```
if(조건)
{
  조건 만족시 실행"
}
```

```
if(조건)
{
  조건 만족시 실행"
}
else
{
  조건 불만족시 실행
}
```

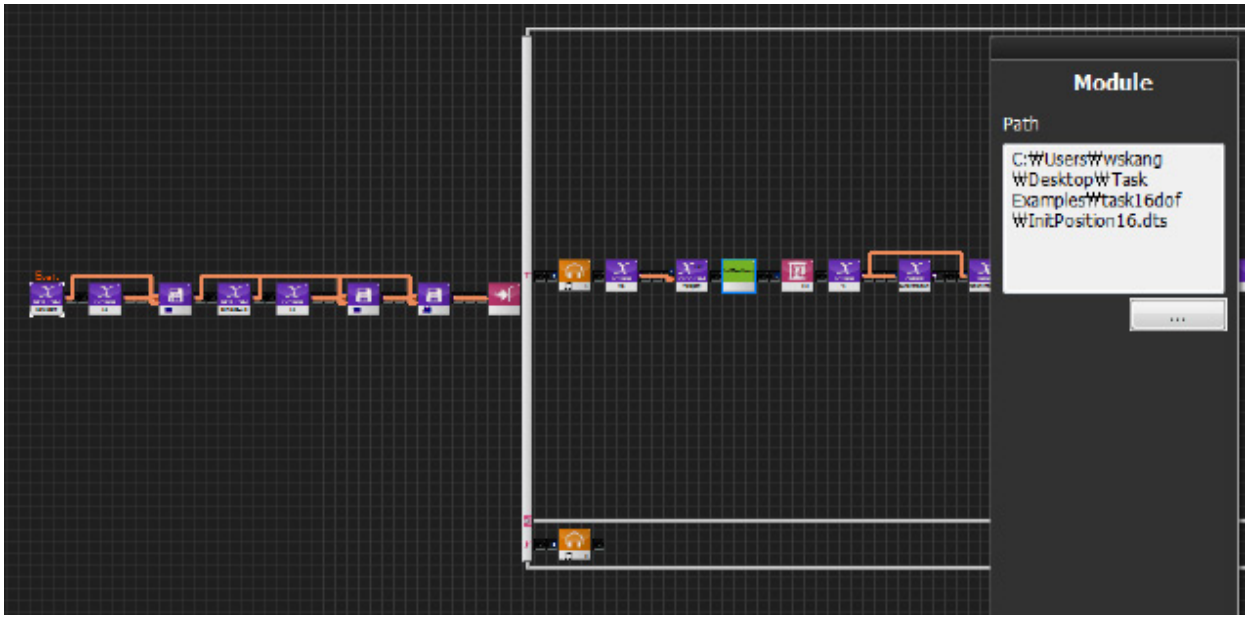
```
if(조건 A)
{
    조건 A 만족시 실행
}
else if(조건 B)
{
    조건 B만족시 실행
}
else if(조건 C)
{
    조건 C 만족시 실행
}
else
{
    조건 ABC 불만족시 실행
}
```

```
switch(n)

  case1:  → n 이 1인 경우
          Break;
  case2:
          Break;
  case3:
     Break;

default:
```

함수

◆ 메인함수

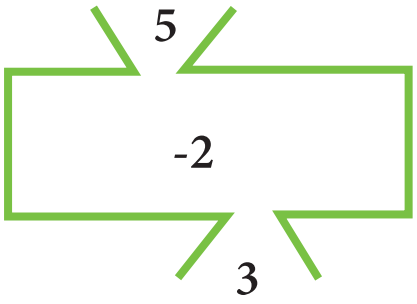
ex) Remocon16.dts 중 일부

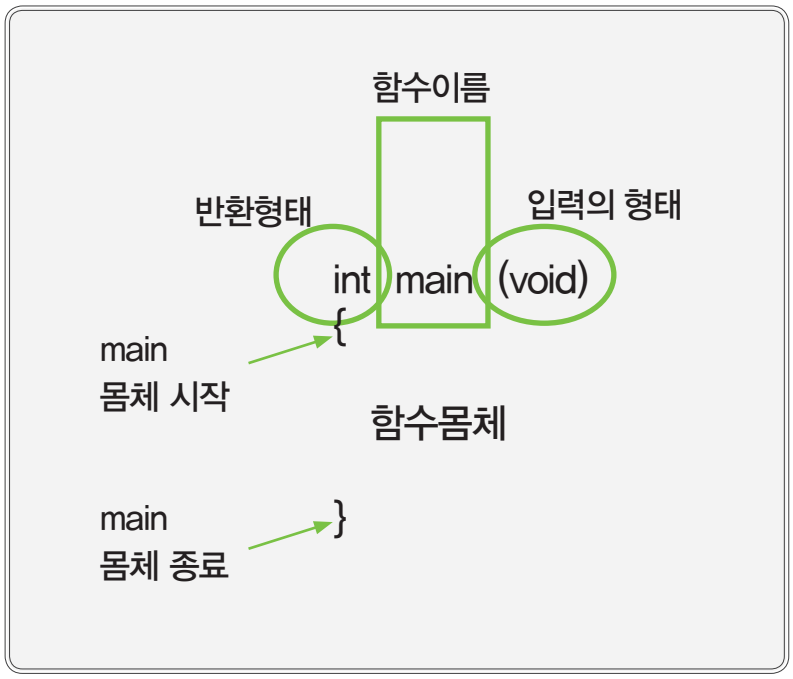


C-like 보기

```
1  void InitPosition16()
2  {
3          jog( 512, 0, 254, 120 )
4          jog( 235, 0, 0, 120 )
5          jog( 235, 0, 1, 120 )
6          jog( 789, 0, 3, 120 )
7          jog( 789, 0, 4, 120 )
8  }
9  void main()
10 {
11         if( ( ( MPSU_ServoCnt == 16 ) && ( MPSU_ServoID_15 == 15 ) ) )
12         {
13                 melody( 1 )
14                 SERVO_TorqCtrl[254]=96
15                 InitPosition16()
16                 delay( 2000 )
17                 CurrentMotion=-1
18                 MotionAfterStop=-1
19                 RmcEnd=false
20                 MtnReady=false
21                 MtnStop=false
22                 Interruptable=true
23                 while( true )
```

※ C라는 언어로 구현된 프로그램은 함수로 시작해서, 함수로 끝난다고 해도 과언이 아닙니다. 함수라는 것은 C 언어를 이해하는데, 가장 중요한 일을 합니다. 포인터 가 어려운건 사실이지만, 함수가 훨씬 더 중요하다고 할 수 있습니다. 포인터는 공부하기가 어려운것이지 활용하는 것은 어렵지 않습니다.





첫째, 함수는 이름을 지닙니다. → main

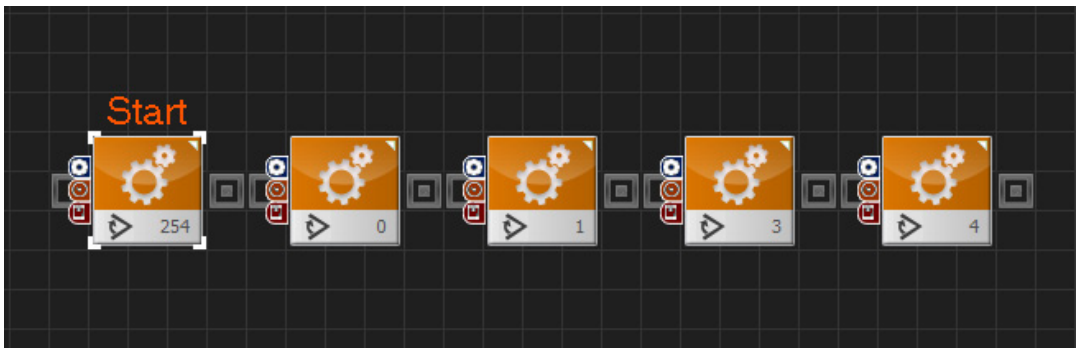
들째, 입력의 형태가 있습니다. → (void)

셋째, 출력의 형태(반환형, 리턴형 이라는 말을 더 많이 함)가 있습니다. → int

넷째, 함수의 몸체 : 함수의 기능을 의미합니다.

A,B,C 함수를 많이 만들어놓고, 특정한 순서에 맞게 호출하는 것이 프로그래밍입니다.

ex)InitPosition16.dts 중 일부



C-like 보기

```
1  void main()
2  {
3          jog( 512, 0, 254, 120 )
4          jog( 235, 0, 0, 120 )
5          jog( 235, 0, 1, 120 )
6          jog( 789, 0, 3, 120 )
7          jog( 789, 0, 4, 120 )
8  }
```

◆ 함수선언

```
(1)  (2)    (3)
int minus (int i, int j)
{
    int result = i-j;
(4) return result;
}
```

(1) 반환 형
(2) 함수 이름
(3) 매개 변수
(4) 값의 반환

※ 4가지 형태의 함수

입력과 출력은 있을 수도 있고, 없을 수도 있습니다. 중요한 것은 기능입니다.

전달 인자 있음, 반환 값 있음

전달 인자 있음, 반환 값 없음

전달 인자 없음, 반환 값 있음

전달 인자 없음, 반환 값 없음

변수의 범위(scope)

※ 변수의 특성에 따른 분류

지역변수 (local variable) : 중괄호 내에 선언되는 변수

전역변수 (Global variable) : 함수 내에 선언되지 않는 변수

정적변수 (Static variable) : 함수 내부, 외부 모두 선언 가능

레지스터 변수(Register variable) : 선언에 제한이 많이 따름

static 변수

함수 내부 및 외부에 선언 가능합니다.

전역변수, 지역변수 앞에 static 키워드를 붙일 수 있습니다. → 파일 범위를 갖습니다. ↔ extern

한번만 초기화된다됩니다.: 전역변수의 특징

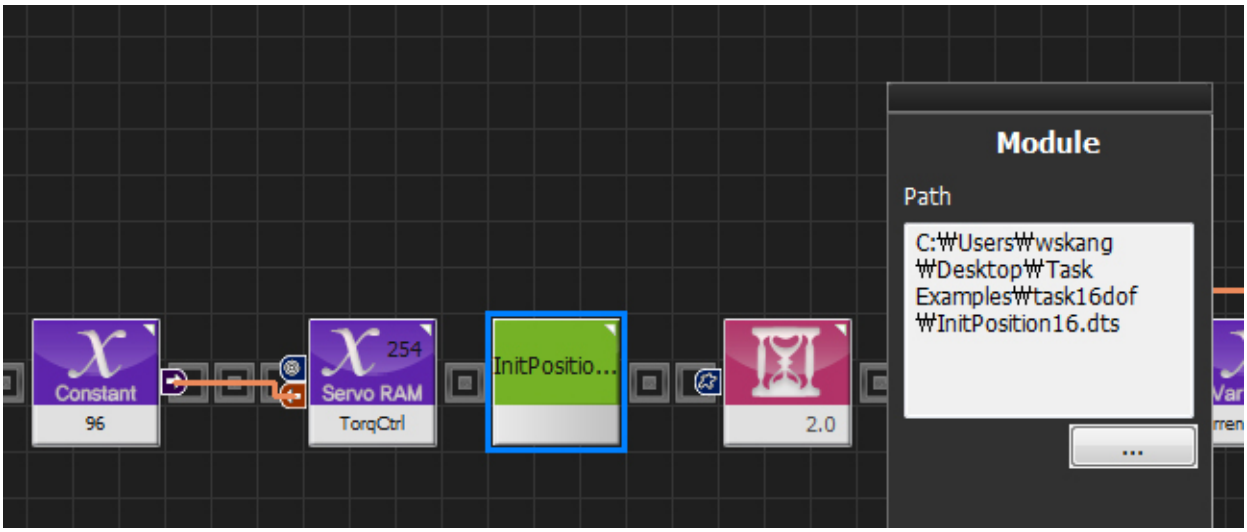
전역변수는 프로그램이 시작하면 올라갔다가, 종료되어야 내려갑니다.

지역변수는 함수가 호출될때마다 초기화됩니다.

함수 내부에서 선언될 경우 함수 내에서만 접근이 가능합니다. : 지역변수의 특징

◆ 함수 호출 및 모듈화 프로그래밍

ex) Remocon16.dts 중 일부



C-like 보기

```
1  void InitPosition16()
2  {
3          jog( 512, 0, 254, 120 )
4          jog( 235, 0, 0, 120 )
5          jog( 235, 0, 1, 120 )
6          jog( 789, 0, 3, 120 )
7          jog( 789, 0, 4, 120 )
8  }
9  void main()
10 {
11         if( ( ( MPSU_ServoCnt == 16 ) && ( MPSU_ServoID_15 == 15 ) ) )
12         {
13                 melody( 1 )
14                 SERVO_TorqCtrl[254]=96
15                 InitPosition16()
16                 delay( 2000 )
17                 CurrentMotion=-1
18                 MotionAfterStop=-1
19                 RmcEnd=false
20                 MtnReady=false
21                 MtnStop=false
22                 Interruptable=true
23                 while( true )
24                 {
25                         if( ( ( MPSU_RmcLength >= 0 && MPSU_RmcD
26                         {
```

Module
Path
C:₩Users₩wskang
₩Desktop₩Task
Examples₩task16dof
₩InitPosition16.dts
...

● 모듈화 프로그래밍

– 기능별로 파일을 나눠가며 프로그래밍하는 것을 말합니다.

– 유지 보수성이 좋아지고. 관리하기가 편해집니다.

예를 들어 200~300 라인은 괜찮지만, 500 라인~ 1000 라인 이상이면 구분도 어렵고, 유지보수도 어려워집니다.

● 파일의 분할 및 컴파일

– 파일을 나눌지라도 완전히 독립되는 것은 아닙니다.

– 파일이 나뉘어도 상호 참조가 발생할 수 있는데, 이는 전역 변수 및 전역 함수로 제한됩니다.

– 같은 코드를 두개의 파일로 분리, 분리된 파일은 하나의 프로젝트로 분류되어야 합니다.

◆ 함수호출

※ Call-By-Value와 Call-By-Reference

값에 의한　　참조에 의한

※ Call-By-Value

– 값의 복사에 의한 함수의 호출입니다.

– 가장 일반적인 함수 호출 형태입니다.

int main(void)

add(val1, val2)

→ int add(int a, int b)

add 라는 함수에서 main 함수의 들어가서 val1 을 조작할 수 없습니다.

왜냐하면, 복사한 것이기 때문에, add에서만 사용이 가능합니다.

※ Call-By-Reference

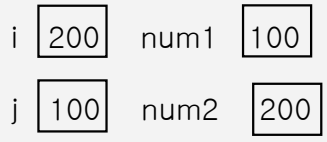
– 참조(참조를 가능테 하는 주소값)를 인자로 전달하는 형태의 함수 호출입니다.

– 예를 들어 변수 val의 주소값이 0x10 라면,adder(&val)는 val 의 주소값을 전달합니다. . pVal =0x10 입니다.

adder 라는 함수는 주소값을 알고 있고, (*pAvl)++; 가 가리키는 메모리 공간에 가서 1을 증가시켜라, 라는 의미입니다.

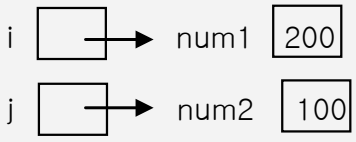
※ 값에 의한 호출 예제

```
#include <stdio.h>
void swap(int i, int j);

int main(void)
{
        int num1, num2;
        num1=100;
        num2=200;
        swap(num1, num2);
        printf(%d %d, num1, num2);

        return 0;
}

void swap(int i, int j)
{
        int temp;
        temp=i;  i=j;  j=temp;
}
```



※ 참조에 의한 호출 예제

```
#include <stdio.h>
void swap(int *i, int *j);

int main(void)
{
        int num1, num2;
        num1=100;
        num2=200;
        swap(&num1, &num2);
        printf(%d %d, num1, num2);

        return 0;
}

void swap(int *i, int *j)
{
        int temp;
        temp=*i; *i=*j; *j=temp;
}
```



◆ 함수의 매개변수

※ 함수 매개변수의 종류

- 실 매개변수

  호출 함수 쪽에서 전달하는 실제 변수입니다.

- 형식 매개 변수

  피호출함수 쪽에서 전달받는 변수입니다.

※ 함수 매개변수의 특징

- 실 매개변수와 형식매개변수의 개수는 일치합니다.
- 실 매개변수명과 형식매개변수의 이름은 같아도 상관없습니다.

※ 매개변수 전달

– 배열이 함수에 대한 인수로 사용될 때 배열의 주소만이 매개변수에게 전달 됩니다.

→ 매개변수가 포인터 형으로 선언되어야 합니다.

```
#include <stdio.h>
void a1(int num[5]), f2(int num[]), f3(int *num);

int main(void)
{
        int count[5]={1,2,3,4,5};
        a1(count);
        a2(count);
        a3(count);
        return 0;
}
```

→ 아래 세 가지 모든 경우 하나의 포인터 매개변수 사용

```
void a1(int num[5]) /*배열의 사용*/
{
        int i;
        for(i=0; i<5;i++)     printf("%d", num[i]);
}
void a2(int num[])  /*언사이즈드 배열 사용*/
{
        int i;
        for(i=0; i<5;i++)     printf("%d", num[i]);
}
void a3(int *num)  /*포인터를 사용*/
{
        int i;
        for(i=0; i<5;i++)    printf("%d", num[i]);
}
```

◆ 재귀함수

※ 재귀함수의 기본적 이해

– 자기 자신을 다시 호출하는 형태의 함수입니다.

– 재귀함수 프로그램을 잘 이해하기 위해서는 컴퓨터 Process 를 잘 이해해야합니다.

– exe 파일 만들려면, 컴파일된 바이너리가 필요합니다.

– 프로그램이 시작되면 메모리 Code area 에 함수등 모든 것이 올라갑니다.

– 메인함수가 실행되면, 순차적으로 함수 진행됩니다. 이 코드를 가져다가 하나씩 패치를 시작하고, CPU 가 메인함수 코드를 가져다가 순차적으로 실행하는 것입니다.

– 리커시브 함수 호출 → CPU 가 리커시브 가져다가 놓고 실행합니다.

– 코드 영역에서 다시 리커시브 함수 호출하여, 가져다가 놓고 실행합니다. →계속 반복함

– 리커시브 함수는 종료되지 않고 계속 실행되고, 메모리에 계속 쌓입니다. → 문제가됨

(Stack overflow 라고 함 : 재귀적 함수가 계속쌓여서 꽉 차는 것)

※ 탈출 조건의 필요성

무한 재귀 호출을 피하기 위함입니다.

재귀함수를 공부하는 이유는, 자료구조(데이터를 표현하는 방식) 나 알고리즘(문제해결)을 공부하게 되면, 재귀함수가 매우 유용합니다. 문제해결을 고민하는데 있어서, 재귀적으로 해결하는게 너무나 많습니다.

```
if(n==1)
return;     → 1, 함수를 빠져나오는 것, 2, 값을 반환하는 것, 두가지 기능이 있습니다.
    (리턴타입이 void 라고 해도, 함수를 빠져나오는 용도로 쓰입니다.)
recursive(n-1);
```

재귀함수 할 때는 중요한 것은 탈출 조건을 달아뒀냐가 매우 중요합니다.

※ 재귀(recursive) 호출 : 순환(recursion)

: 어떤 프로시저(procedure)가 자기 자신을 호출 하는 것입니다.

```
#include <stdio.h>

int facto1(int n);

int main(void)
{
        printf("5 factorial : %d", facto1(5));
        return 0;
}

int facto1(int n)
{
        if(n==0)
                return 1;
        else
                return n * facto1(n-1);
}
```

```
#include <stdio.h>

int facto2(int n);

int main(void)
{
        printf("5 factorial : %d", facto2(5));
        return 0;
}

int facto2(int n)
{
        int i, result;
        result=1;
        if(n==0)
                return 0;
        for(i=n; i>0; i--)
                result=result *i;
        return result;
}
```

PART 04

AVR 로봇제어기 Firmware 익히기

AVR 설치 및 사용법

AVR 프로그래밍을 위해서는 WinAVR compiler 와 AVR studio 가 필요합니다.

WinAVR 은 http://sourceforge.net/projects/winavr/files/WinAVR/ 다운받습니다.

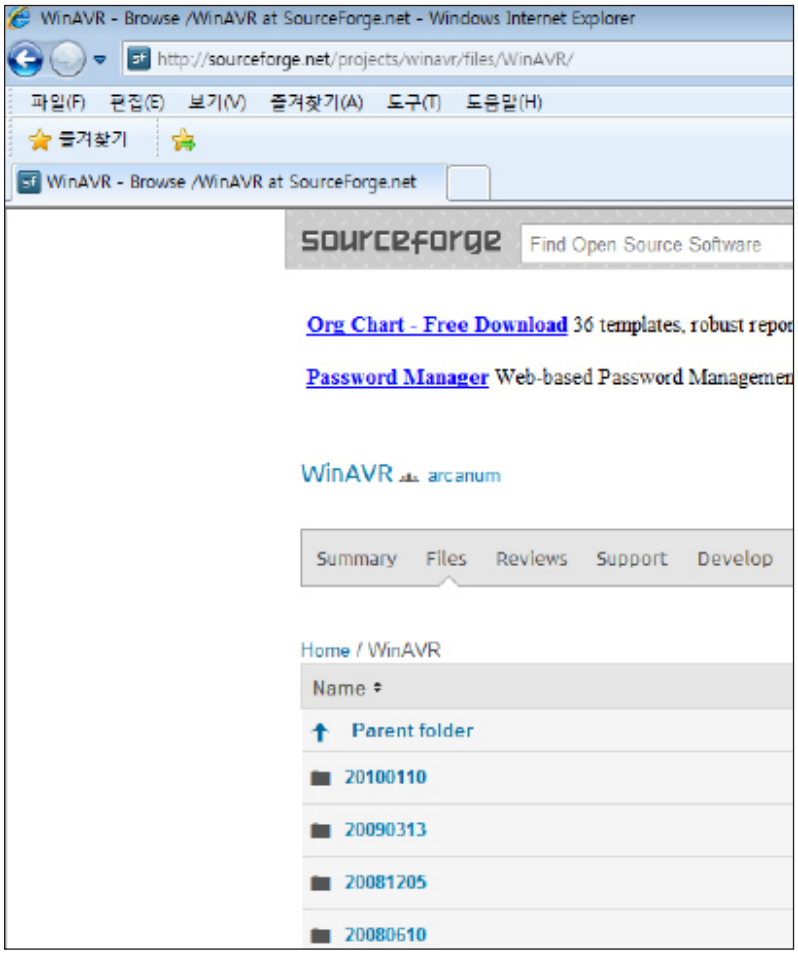
AVR studio 는 http://www.atmel.com/ 에서 다운받습니다.

WinAVR 설치하기

01 URL 입력

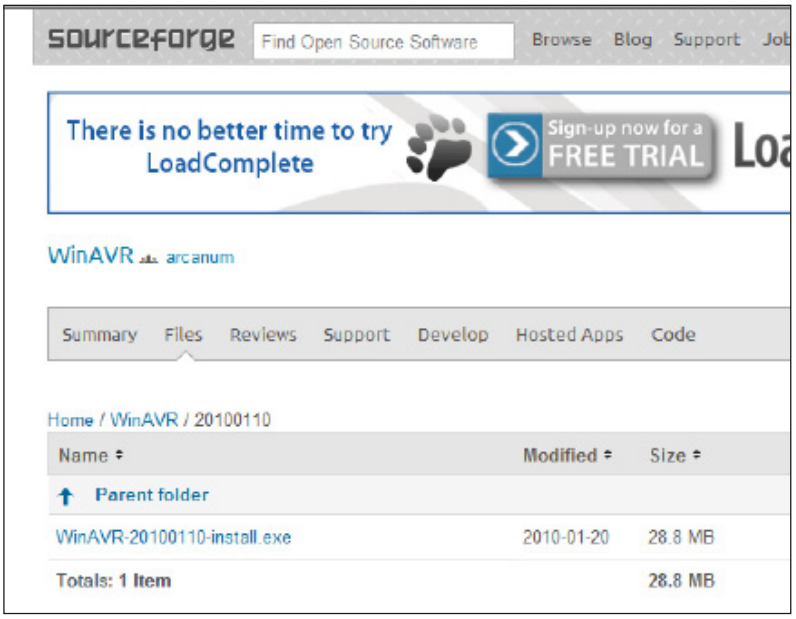
주소창에 URL을 입력합니다.

http://sourceforge.net/projects/winavr/files/WinAVR/



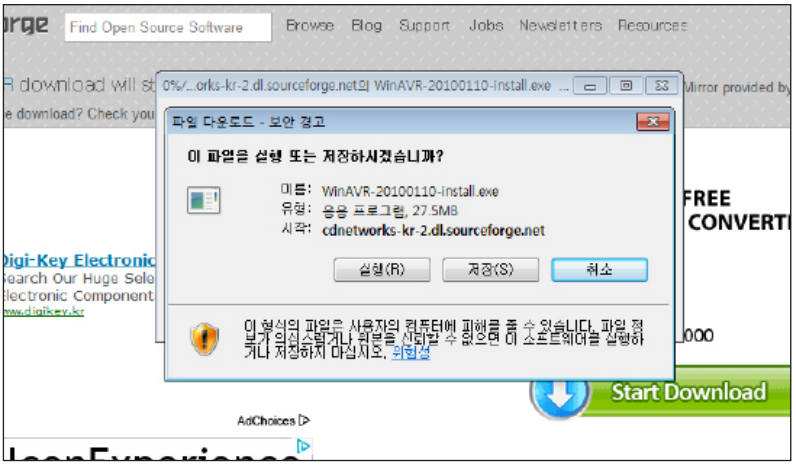
02 다운로드 링크

20100110의 우측 체크박스를 클릭합니다.



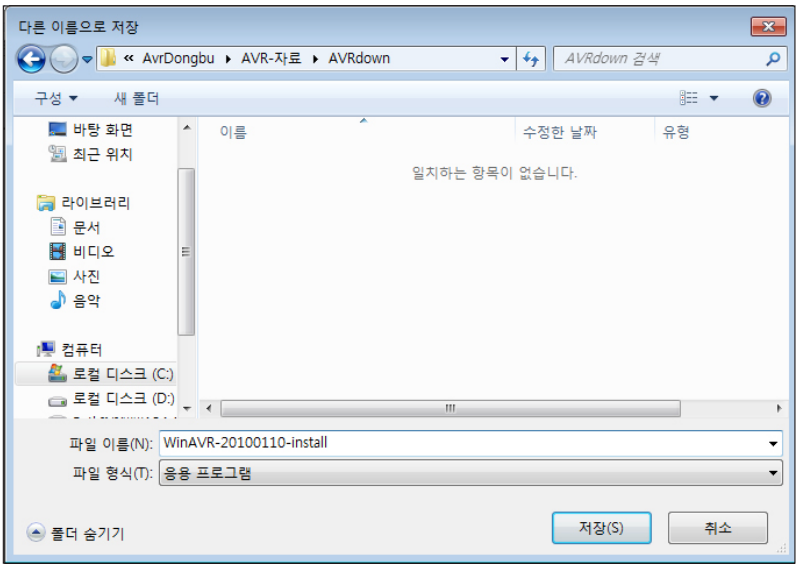
03 다운로드

WinAVT-20100110-install.exe의 체크박스를 클릭합니다.



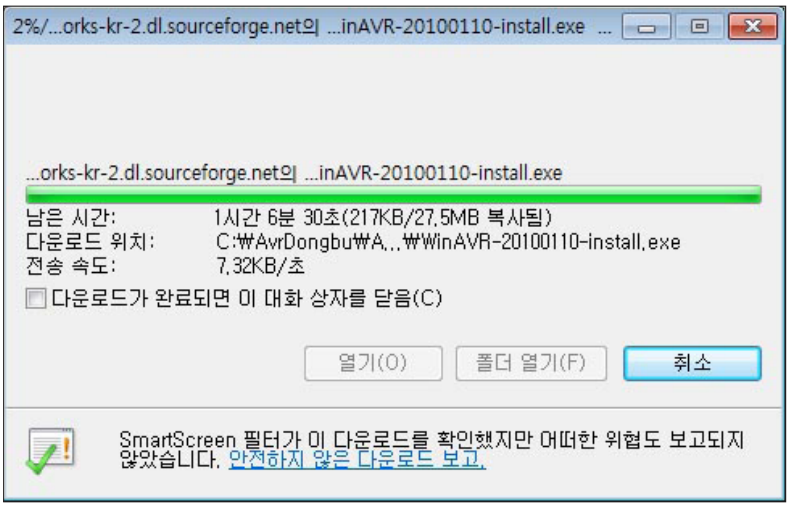
04 설치파일 저장

"저장"버튼을 클릭합니다.



05 저장 폴더 지정

설치파일을 저장할폴더 지정 후 "저장" 버튼을 클릭하세요



06 프로그램 다운로드

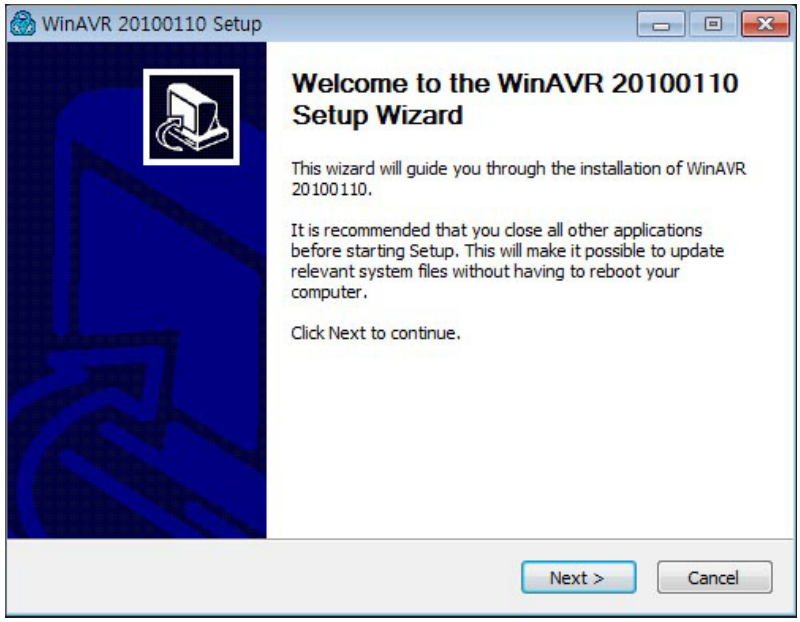
"다음" 버튼을 클릭합니다.

07 설치 폴더를 선택

생성된 아이콘을 더블클릭 하세요

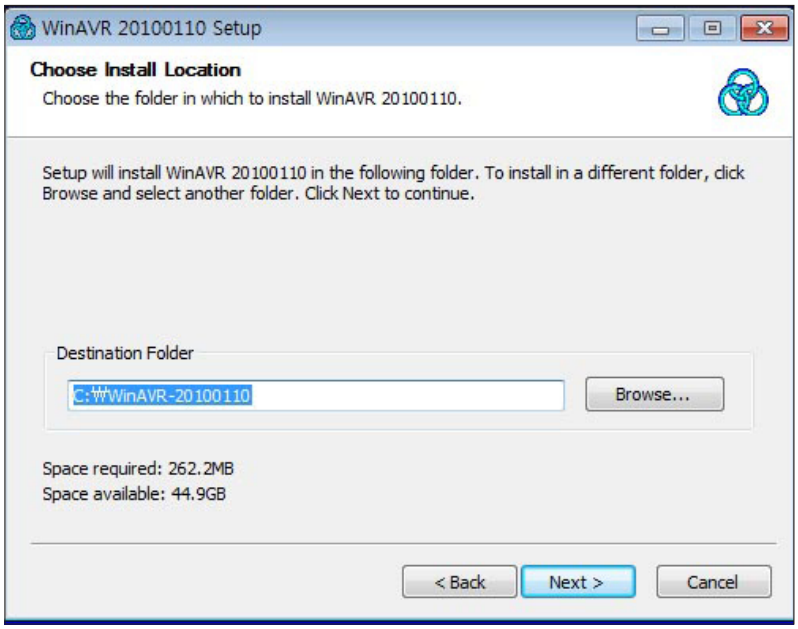
08 설치 확인

언어를 "English"로 선택 후 "OK"를 클릭합니다.



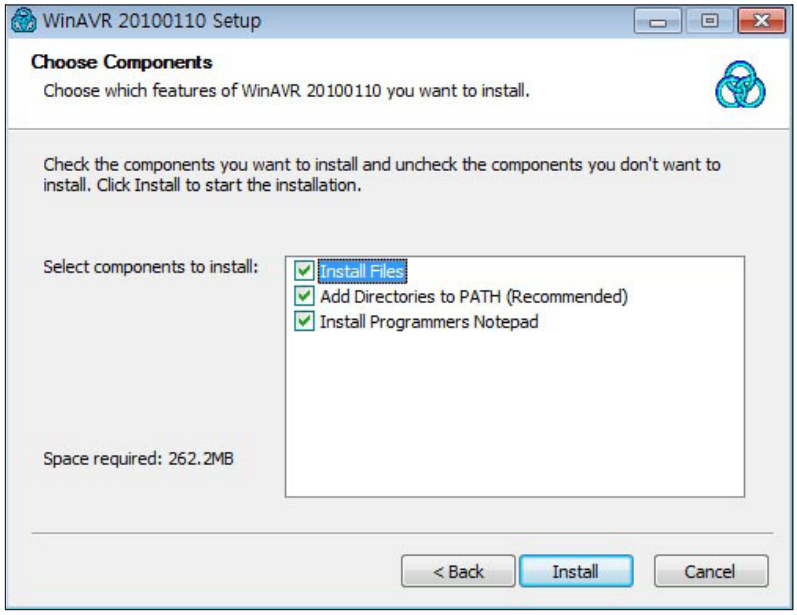
09 설치 마법사 시작

설치 마법사가 실행됩니다. "NEXT" 를 클릭하세요.



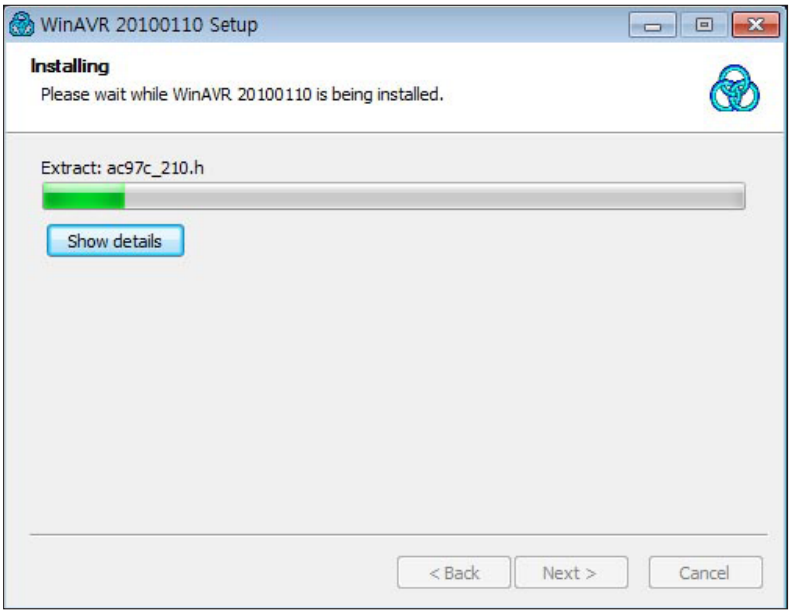
10 설치 폴더 확인

"NEXT" 를 클릭하세요



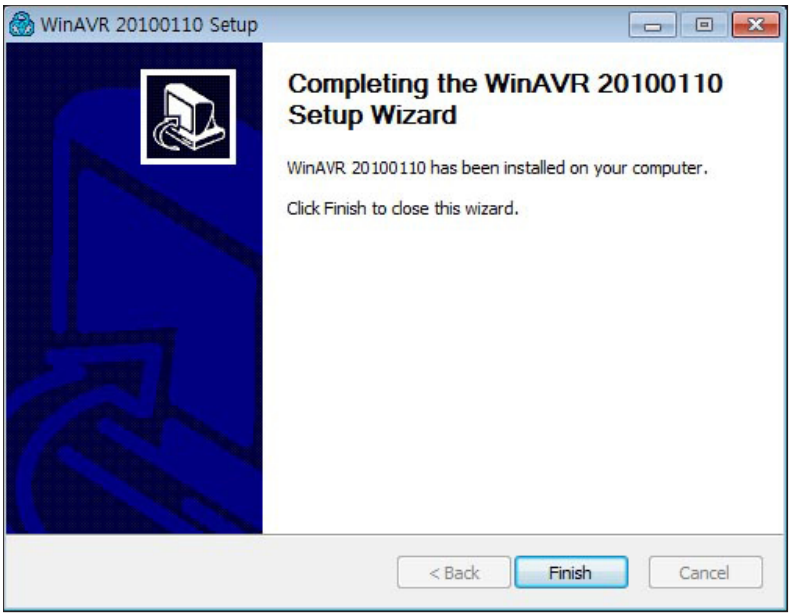
11 구성 요소 확인

"Install" 을 클릭하세요.



12 설치 시작

프로그램 설치를 시작합니다.



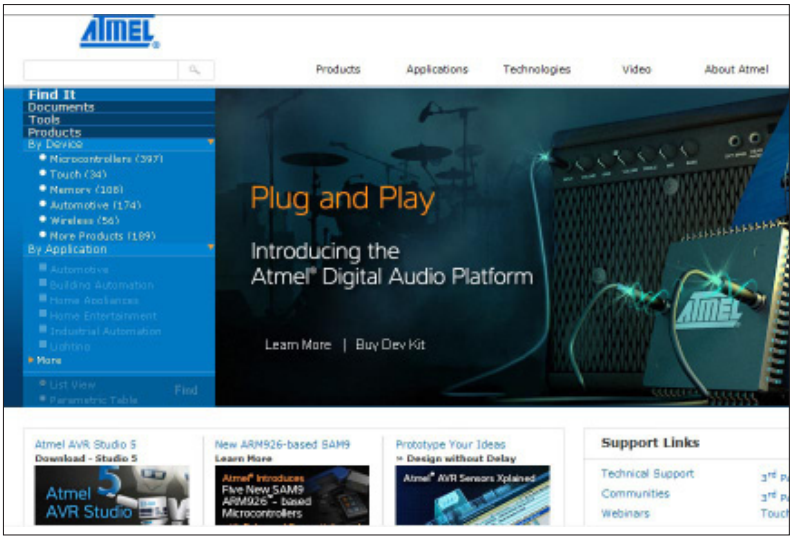
13 설치 완료

"Finish" 를 클릭하면 설치가 완료됩니다.

AVR Studio 설치하기

01 URL 입력

주소창에 www.atmel.com URL을 입력합니다.

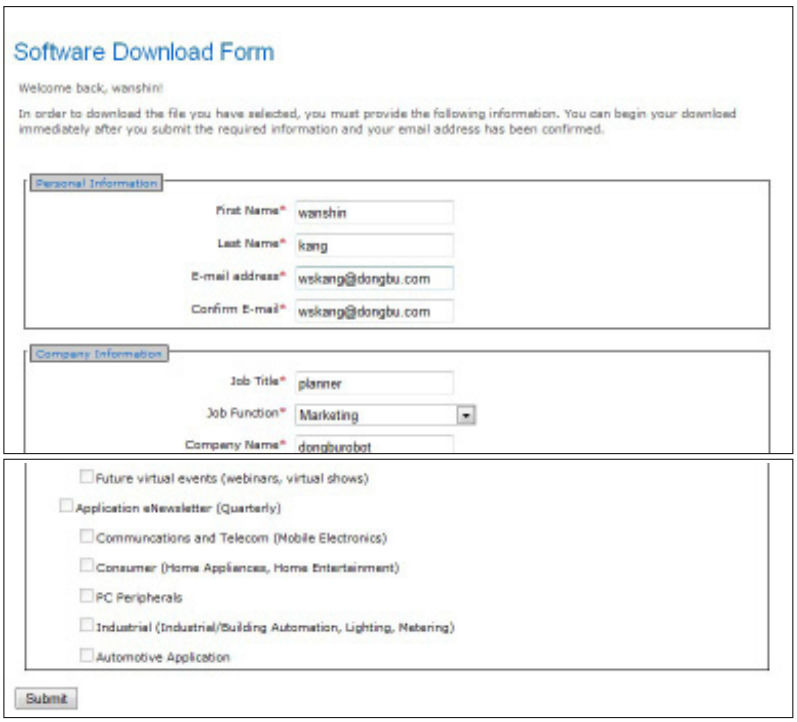


02 다운로드 링크

"Support Links"의
"Download AVR Studio 5"를 클릭합니다.

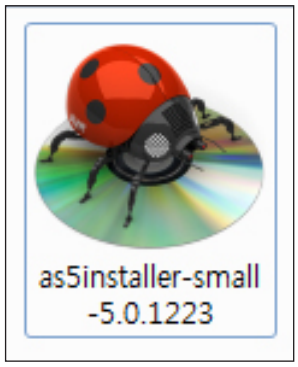
03 소프트웨어 선택

"AVR Studio 5.1 Installer–Full"을 클릭합니다.



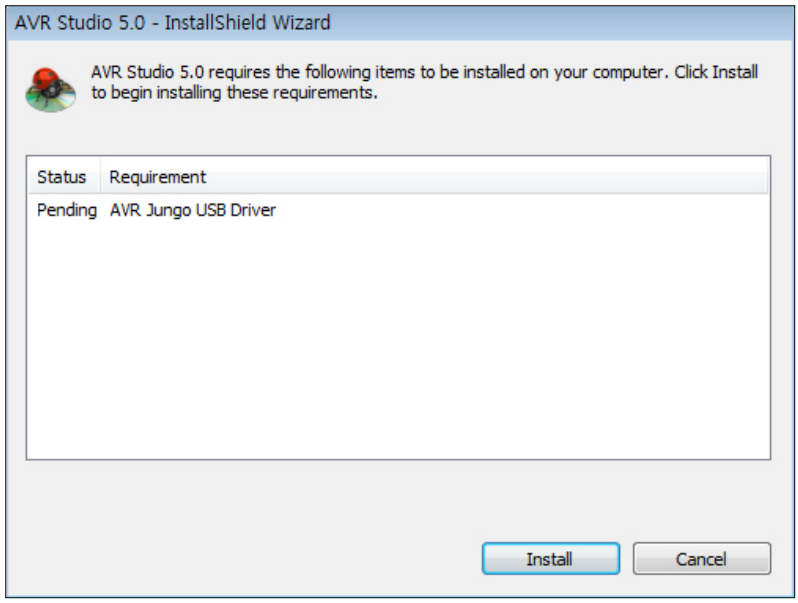
04 소프트웨어 다운로드 양식

사용자 정보를 입력한 뒤 submit을 클릭합니다.



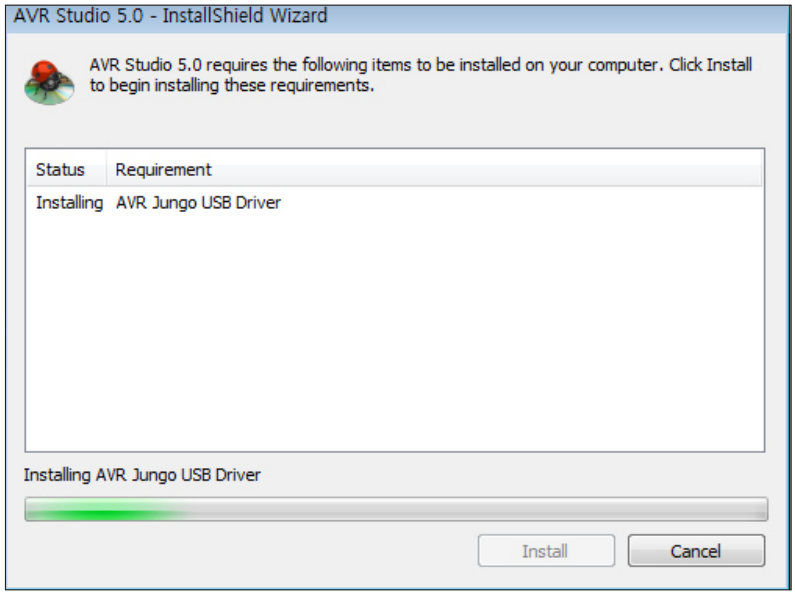
05 설치 프로그램 실행

아이콘을 더블클릭합니다.



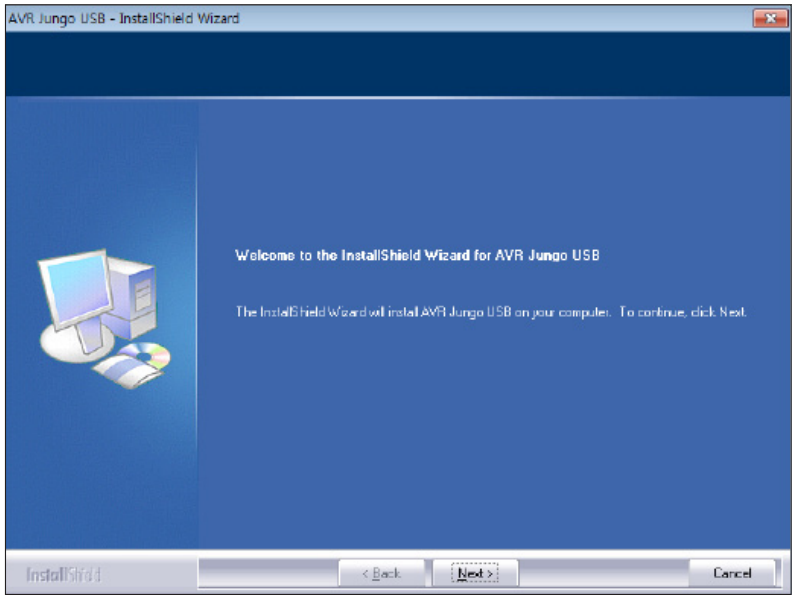
06 설치 시작

"Install" 버튼을 클릭하세요



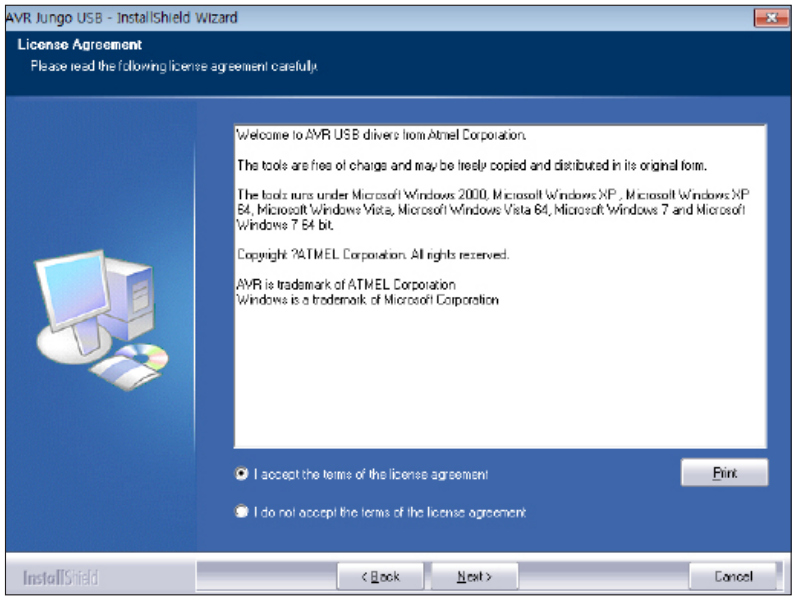
07 설치

프로그램이 설치됩니다.



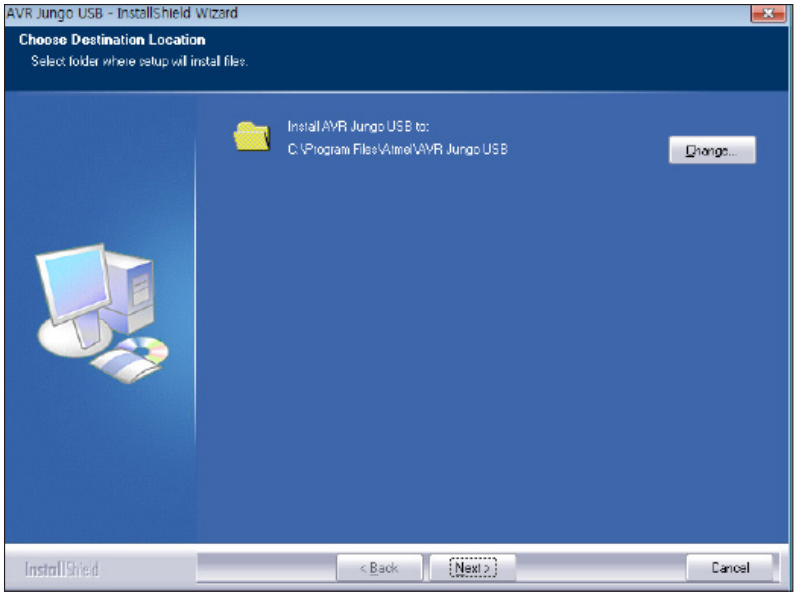
08 AVR Jungo USB 설치 확인

"NEXT"를 클릭하세요



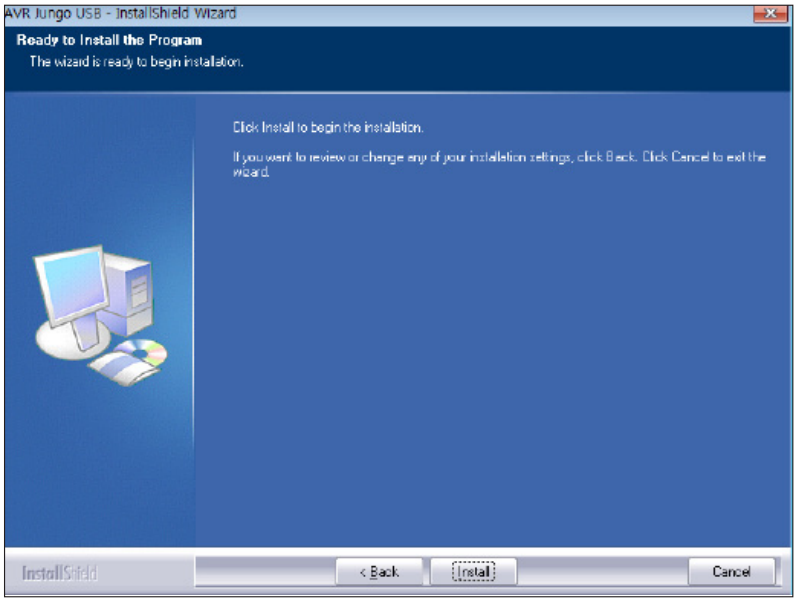
09 약관 동의

"I accept"를 선택하고 "Next"를 클릭하세요



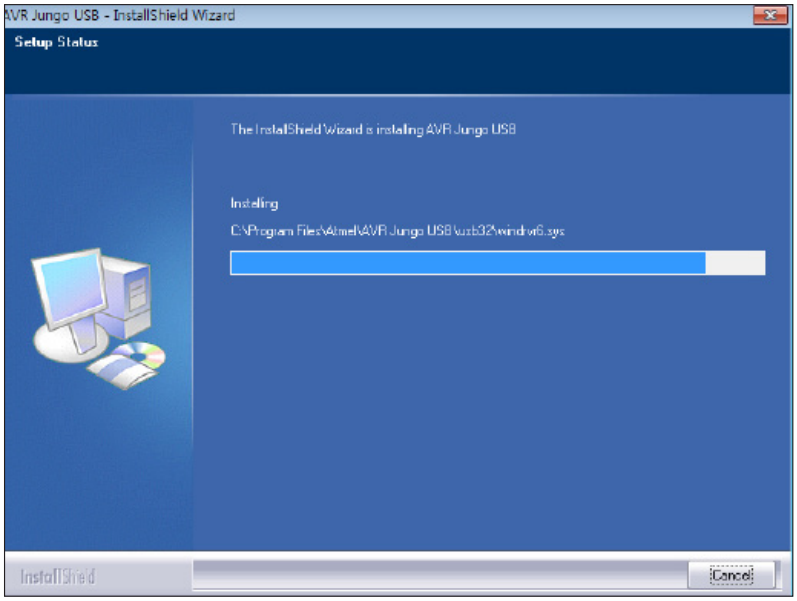
10 설치 폴더 선택

프로그램 설치폴더를 확인 후 “Next”를 클릭합니다.



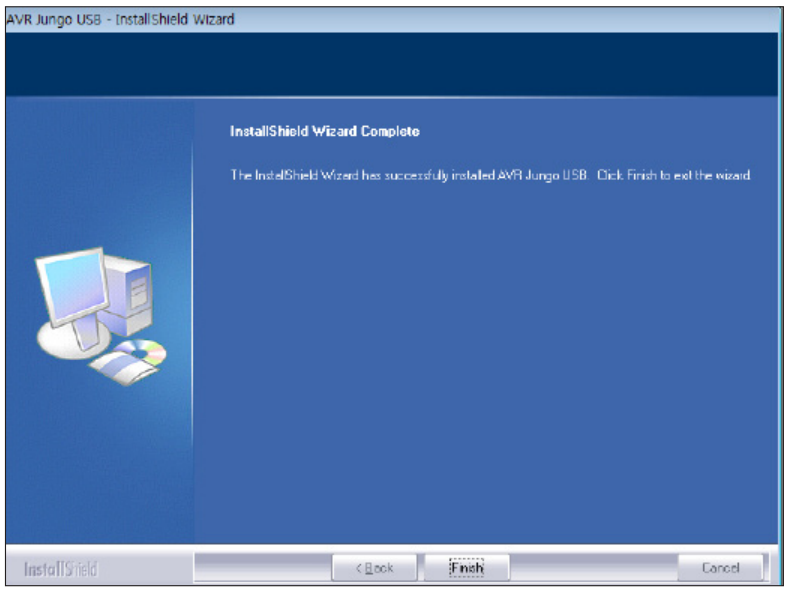
11 설치 시작

“Install”을 클릭하면 설치를 시작합니다.



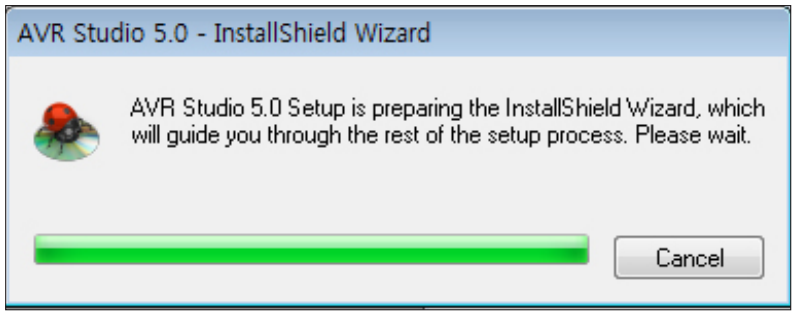
12 설치

프로그램이 설치됩니다.



13 설치완료

"Finish"를 클릭하면 AVR Jungo USB설치가 완료됩니다.

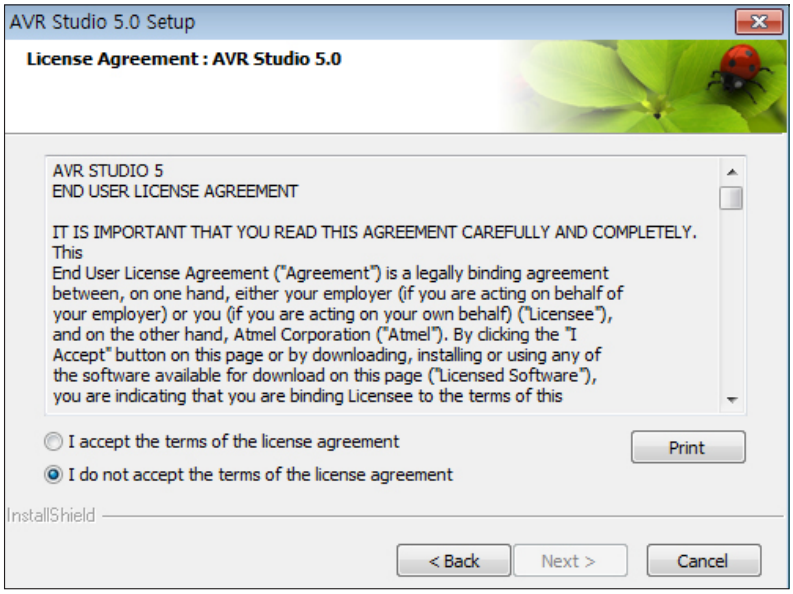


14 AVR Studio 5.0 설치 준비 완료

AVR Studio 5.0을 설치할 준비가 완료되었습니다.

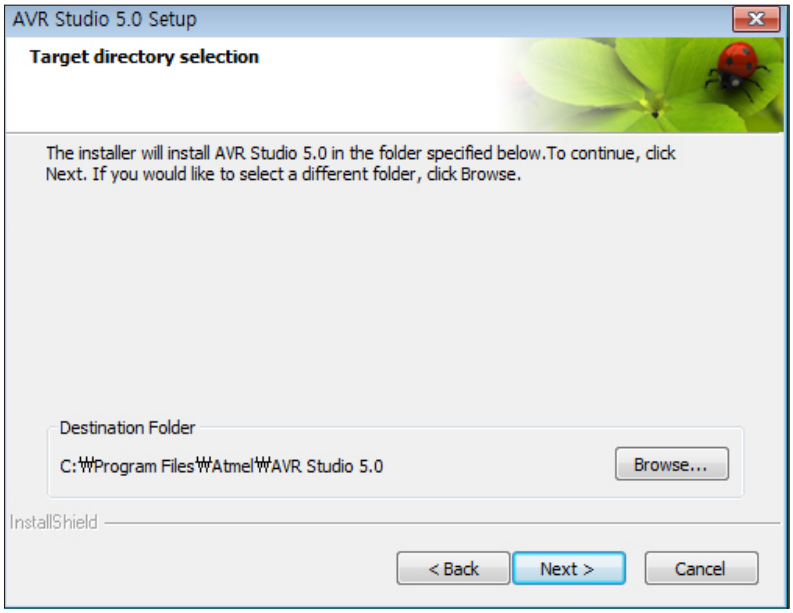
15 설치 마법사 시작

"Next"를 클릭합니다.



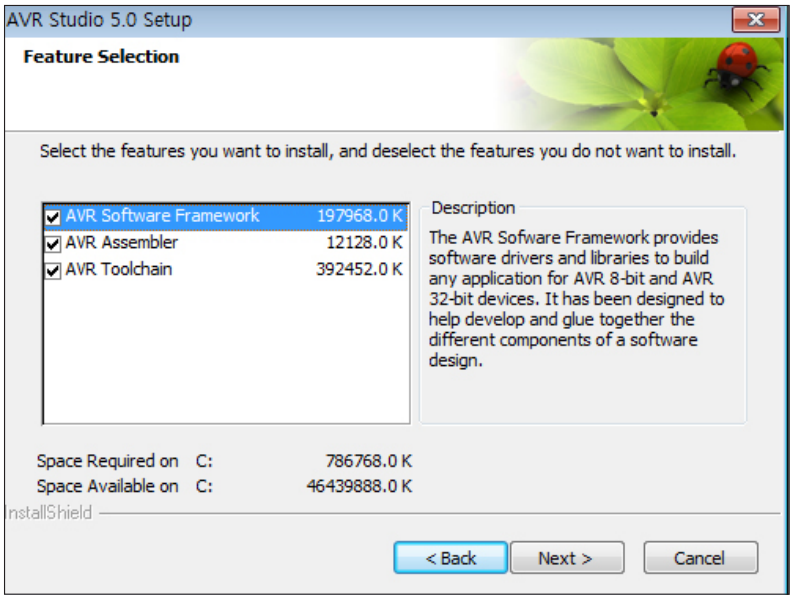
16 약관 동의

"I accept the terms of the licese agreement"를 체크한 뒤 "Next"를 클릭합니다.



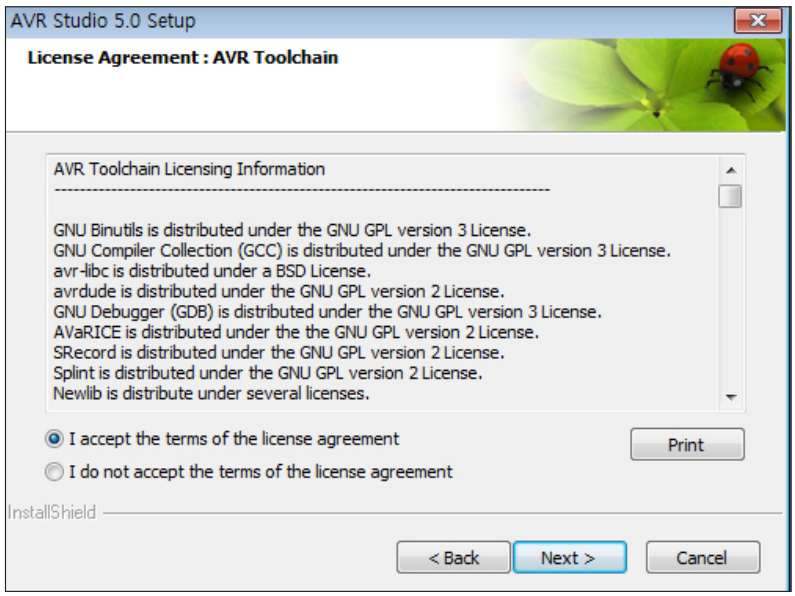
17 설치 폴더 확인

설치 폴더 확인 후 "Next"를 클릭합니다.



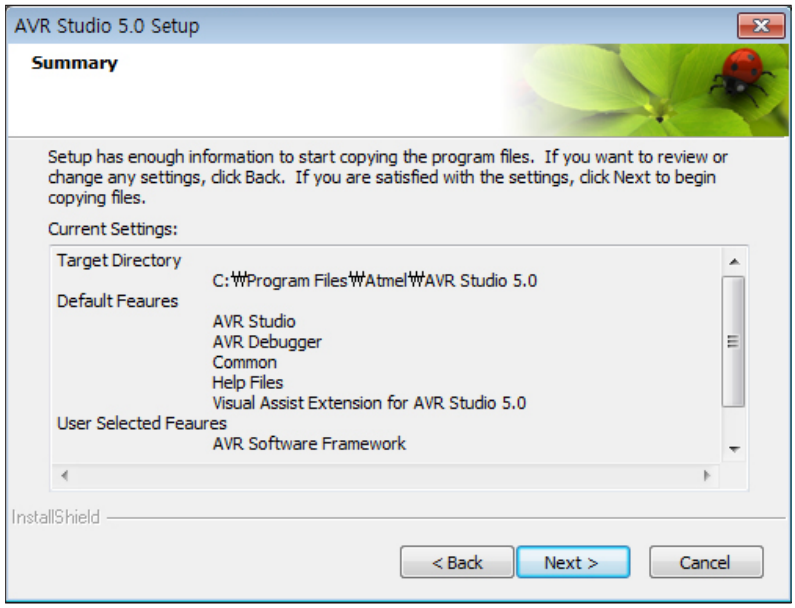
18 설치 기능 선택

설치할 기능 선택후 "Next"를 클릭합니다.



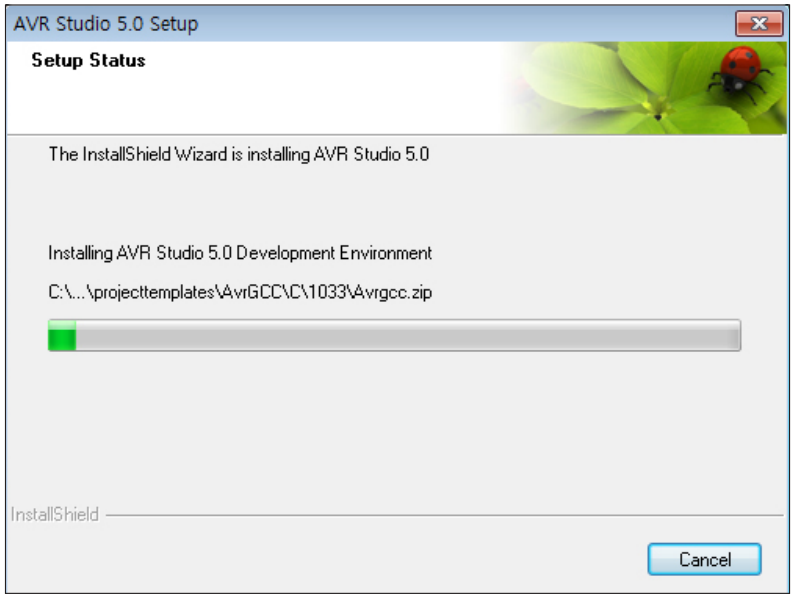
19 약관 동의

약관 동의 후 "Next"를 클릭합니다.



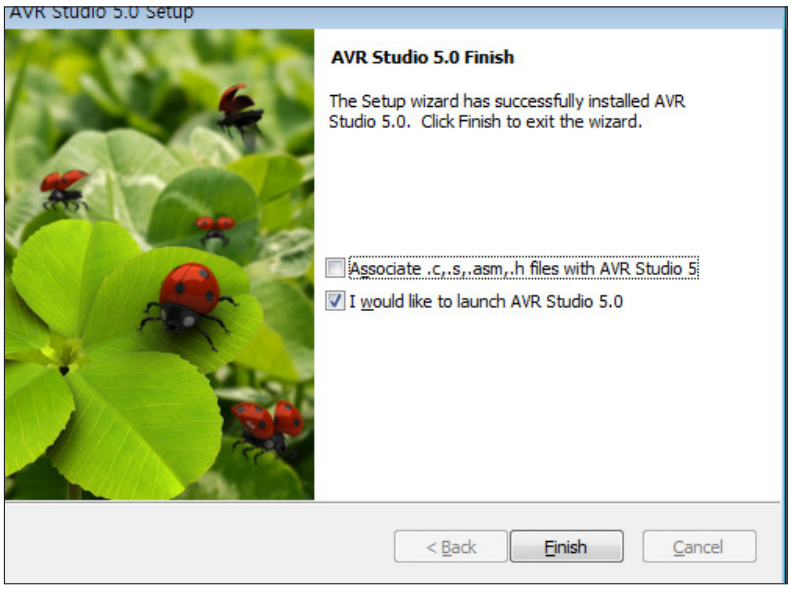
20 설치 내용 확인

설치할 내용 확인 후 "Next"를 클릭합니다.



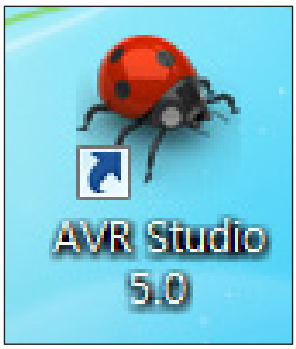
21 설치 시작

설치가 시작됩니다.



22 설치 완료

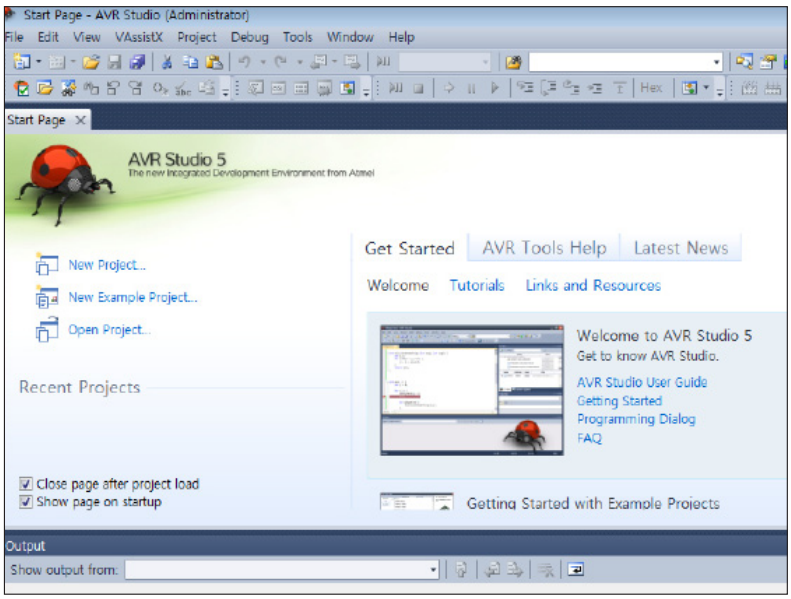
설치가 완료되었습니다.
"I would like to launch AVR Studio 5.0"을 체크한 뒤 "Finish"를 클릭하면 프로그램이 실행됩니다.



프로젝트 만들기 및 코딩

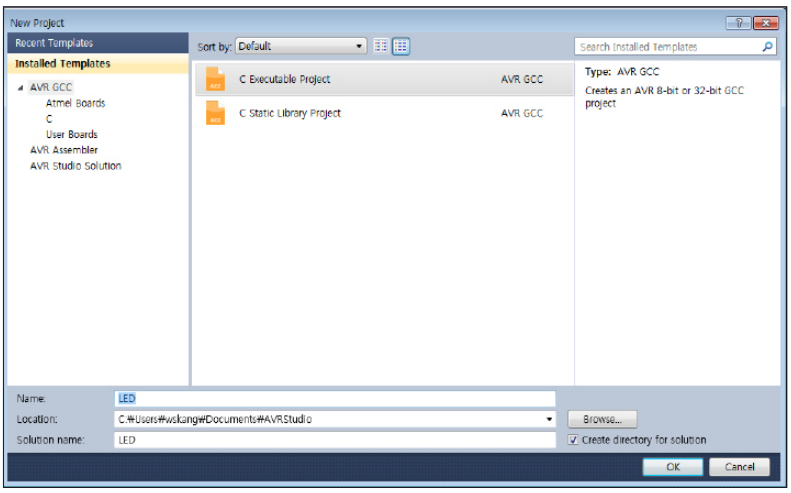
01 실행

AVR Studio 를 실행합니다.



02 New Project

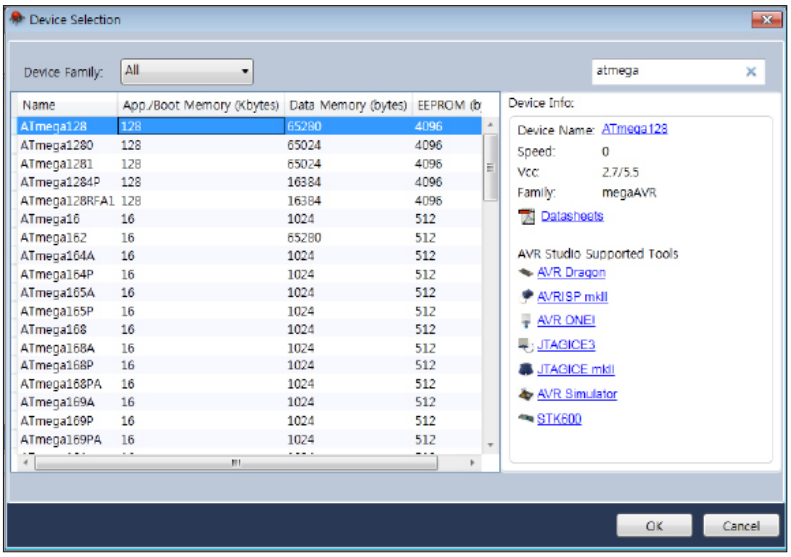
New Project 를 클릭합니다.



03 GCC 선택

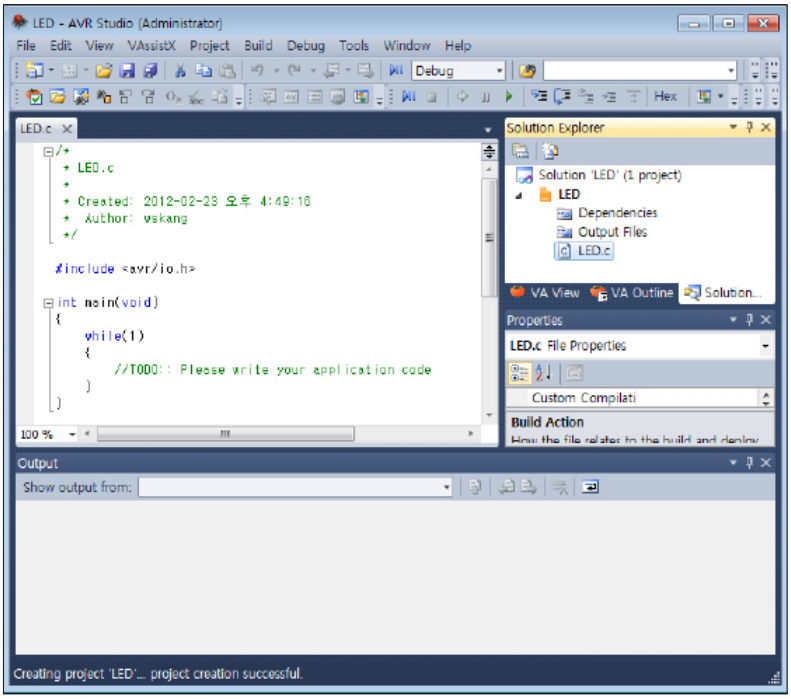
C Executable Project 를 선택합니다.

Name 을 "LED"로 입력하고, Create directory for solution 을 클릭하고, OK 버튼을 누릅니다.



04 ATmega 선택

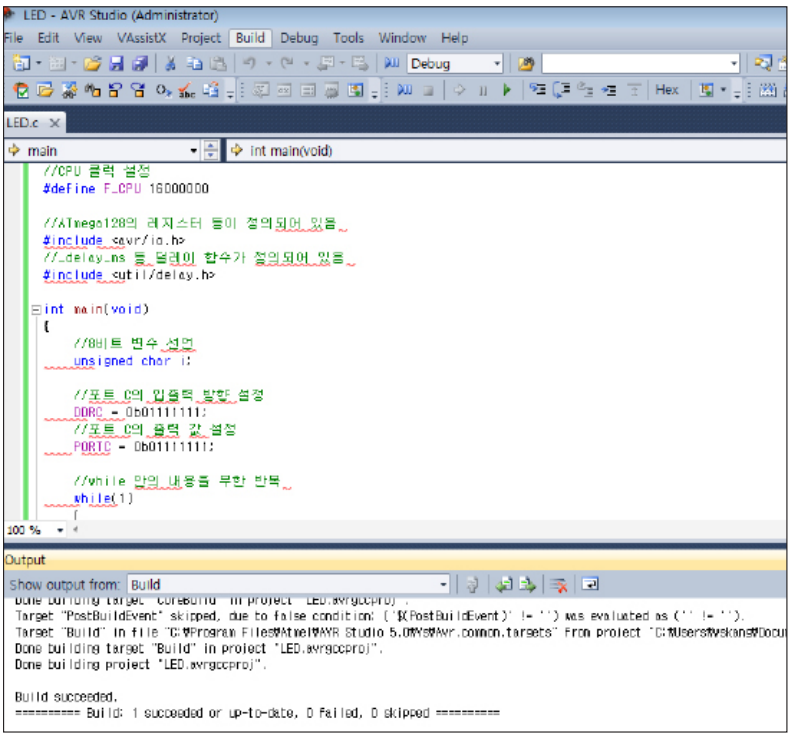
우측 상단에 atmega 를 검색하여, 리스트중에 ATmega128 을 선택합니다.



05 Editor

편집창에 코딩을 시작합니다.

해당 예제를 작성합니다. 아래는 LED 예제입니다.

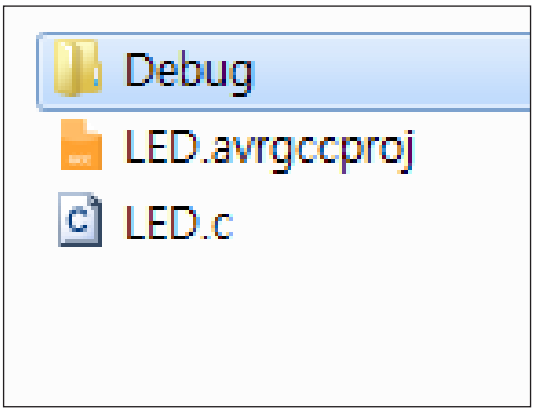


Build 및 실행

01 Build

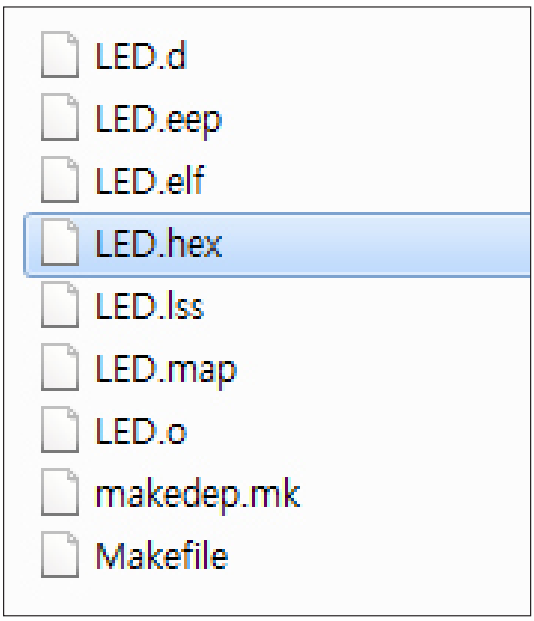
Build 메뉴를 선택하여 Build Solution 을 클릭하여 Build 시킵니다.

1 succeded 0 failed 가 나오면 에러없이 컴파일 완료됐다는 것을 의미합니다.



02 Debug 폴더

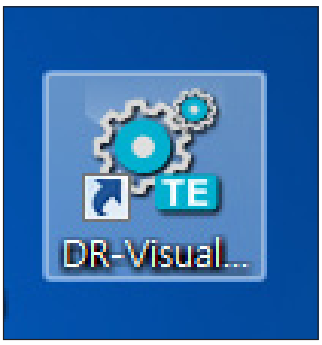
Build 후에 소스가 있는 폴더에 들어가면 Debug 폴더가 생성 되어있습니다. 클릭하여 폴더 안의 내용을 봅니다.



03 hex 확인

Debug 폴더 안의 hex 파일 확인합니다. Hex 파일이 제어기에 포팅시키는 파일입니다.

DR-Visual Logic 에서 펌웨어 복구를 실시합니다.

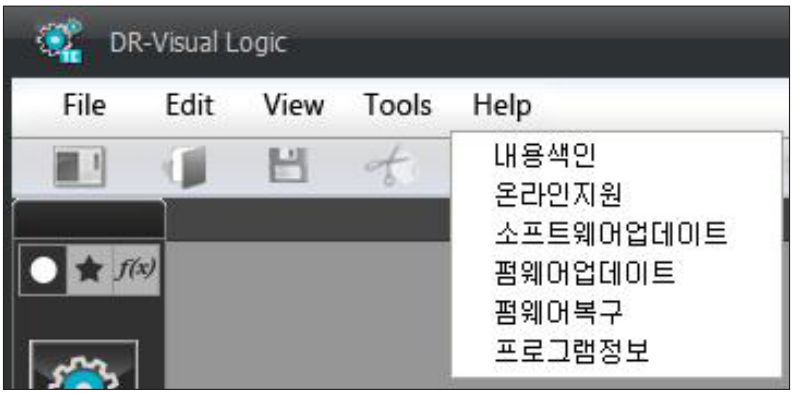


04 Visual Logic

DR-Visual Logic 을 실행시킵니다.

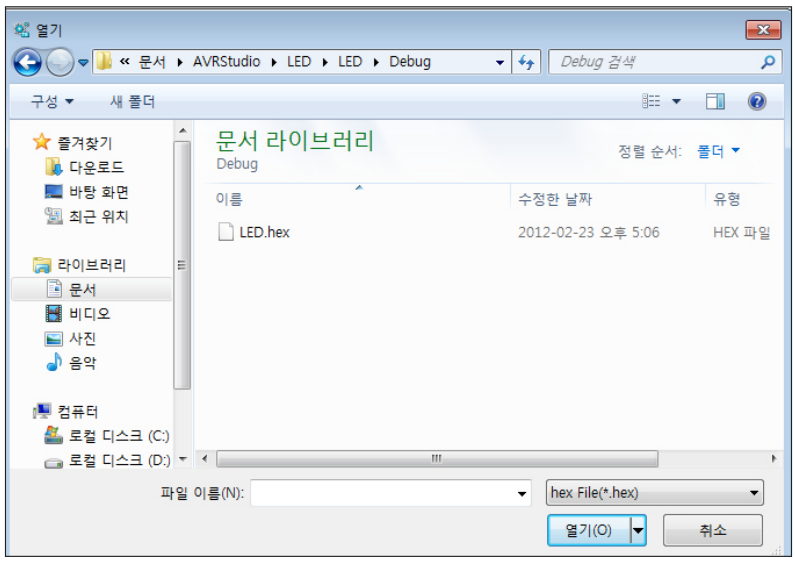
05 연결하기

COM 번호를 설정하고 연결하기를 클릭합니다.



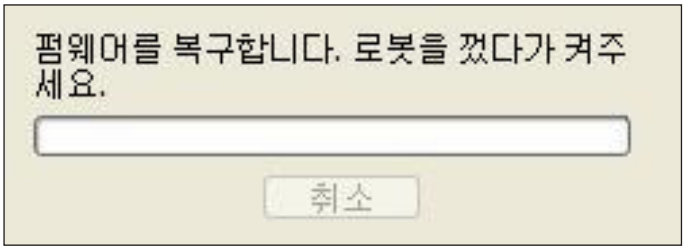
06 펌웨어 복구

도움말 메뉴에서 펌웨어복구를 클릭합니다.



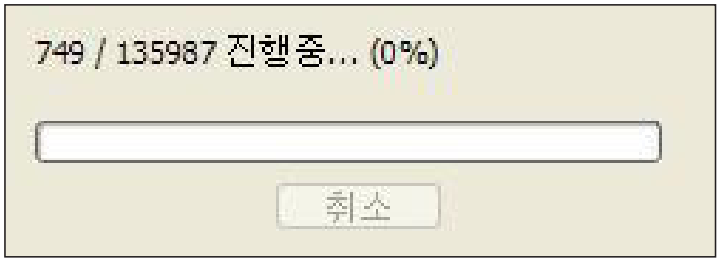
07 불러오기

Debug 폴더에서 Hex 파일을 불러옵니다.



08 펌웨어복구

로봇을 껐다가 켜주세요 라는 문구가 나옵니다.
제어기의 전원버튼을 한번 껐다가 다시 켜주세요.



09 복구시작

펌웨에 복구가 시작됩니다.

10 실행

Servo LED부터 Power LED까지 LED가 하나씩 추가로 켜집니다. LED 7개가 다 켜지고 난 후에는 다시 Servo LED만 켜지고, 차례로 하나씩 켜지는 것을 반복합니다.

CH.1 : LED 제어하기

ATmega128의 디지털 출력을 사용하여 DRC에 장착된 LED를 제어합니다.

AVR 원리

AVR에는 디지털 입출력(I/O) 포트가 내장되어 있으며, 디지털 신호(HIGH와 LOW)를 입력 받거나 내보낼 수 있습니다. 같은 핀에서 입력을 받을 것인지 출력을 할 것인지를 마음대로 설정할 수 있기 때문에, LED 제어, 버튼 입력 받기, 디지털 데이터 전송 등 여러 용도로 사용됩니다. ATmega128에 내장된 I/O 포트는 A부터 G까지 총 7개가 있으며, A부터 F까지는 8개의 핀이 있고 G는 5개의 핀이 있어 총 53개의 핀을 자유로이 디지털 입출력으로 사용할 수 있습니다. 이 53개의 핀은 또한, 범용 입출력 기능 외에 각각의 특수한 다른 기능을 가지고 있으며, 레지스터의 설정에 따라서 이 기능을 사용할지 여부를 결정할 수 있습니다. 각 핀이 어떤 특수한 기능을 가지고 있는지는 차차 공부하면서 알게 될 것입니다.
디지털 출력에 관련된 레지스터는 아래와 같습니다. ATmega128의 레지스터는 모두 8개의 비트(1바이트)로 구성되어 있습니다. DDRX와 PROTX에서 X는 포트 이름(A~G)을 가리키는 문자입니다.

■ DDRX : PORT X 데이터 방향 레지스터

비트	7	6	5	4	3	2	1	0
	DDX7	DDX6	DDX5	DDX4	DDX3	DDX2	DDX1	DDX0
읽기/쓰기	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
초기값	0	0	0	0	0	0	0	0

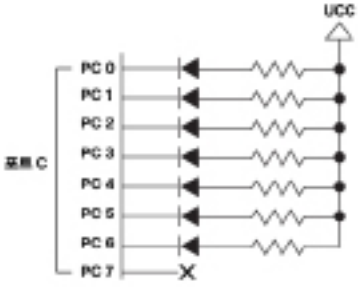
■ PORTX : PORT X 데이터 레지스터

비트	7	6	5	4	3	2	1	0
	PORTX7	PORTX6	PORTX5	PORTX4	PORTX3	PORTX2	PORTX1	PORTX0
읽기/쓰기	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
초기값	0	0	0	0	0	0	0	0

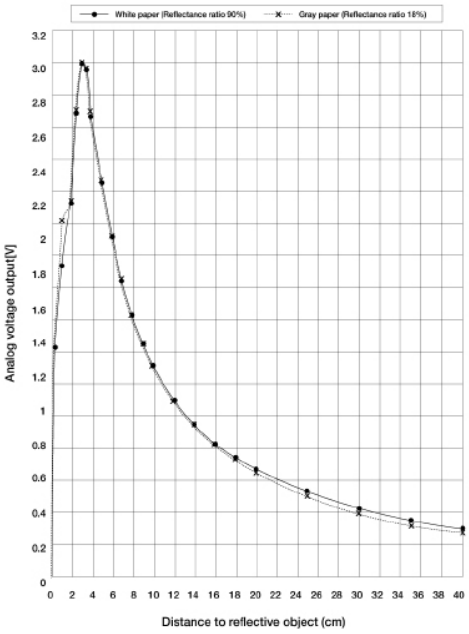
DDRX 레지스터는 포트 X의 입출력 방향을 결정하는 레지스터이며 비트 값이 0이면 입력, 1이면 출력으로 설정됩니다. PORTX 레지스터는 포트 X에 출력할 논리 값을 저장하는 레지스터입니다. DDXn값이 1(출력)으로 설정되어 있으면, 포트 X의 n번 핀에 PORTXn에 설정된 값을 출력하게 됩니다.

H/W 원리

다이오드는 2개의 전극을 가지며, 순방향 전압에서는 전류가 흐르나 역방향 전압에서는 전류가 거의 흐르지 않는 특성을 가지고 있습니다. LED는 Light Emitting Diode의 약자로, 순방향으로 전압이 가해지면 전류가 가해지면서 불이 켜지고(그림 01), 역방향으로 전압이 가해지면 불이 꺼집니다(그림 02). ATmega128로 LED를 제어하는 회로는 Active Low(그림 03)와 Active High(그림 04)의 두 가지 방식이 있습니다. Active Low 방식에서는 핀의 출력이 Low일 때 LED에 순방향 전압이 걸려 불이 켜집니다. Active High 방식에서는 반대로 High일 때 LED가 켜집니다. DRC에서는 PORT C의 0~6번 핀을 LED제어에 사용하며, Active Low 방식으로 회로가 구성되어 있습니다(그림 05). 핀 번호에 대응되는 LED 이름은 아래 표와 같습니다.

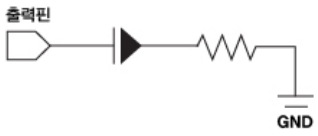


[그림1]

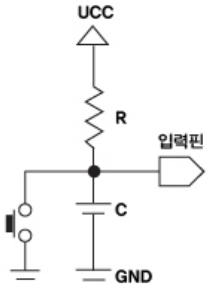


[그림2]

[그림3]



[그림4]



[그림5]

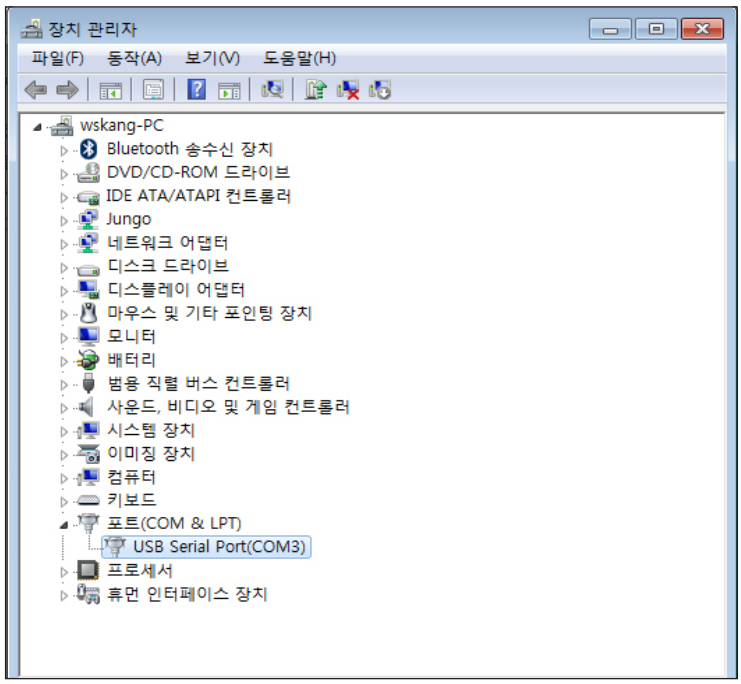
Pin 번호	7	6	5	4	3	2	1	0
LED 이름	–	Power	Spare	RX	TX	EXEC	Program	Servo

준비

하드웨어 준비하기

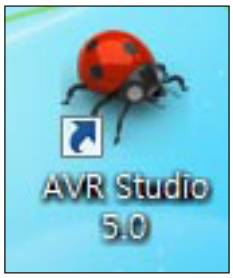
01 연결

DRC 제어기 아답터로 전원을 연결하고, USB to Serial Cable 로 PC 와 DRC 를 연결합니다.



02 COM port 연결

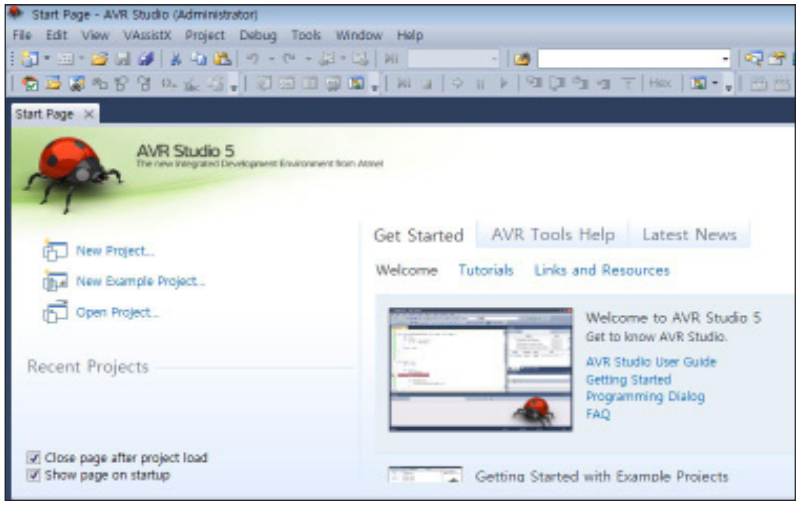
내컴퓨터 〉 속성 〉 장치관리자에서 COM 연결 확인합니다.



AVR Studio 실행 및 코딩 준비

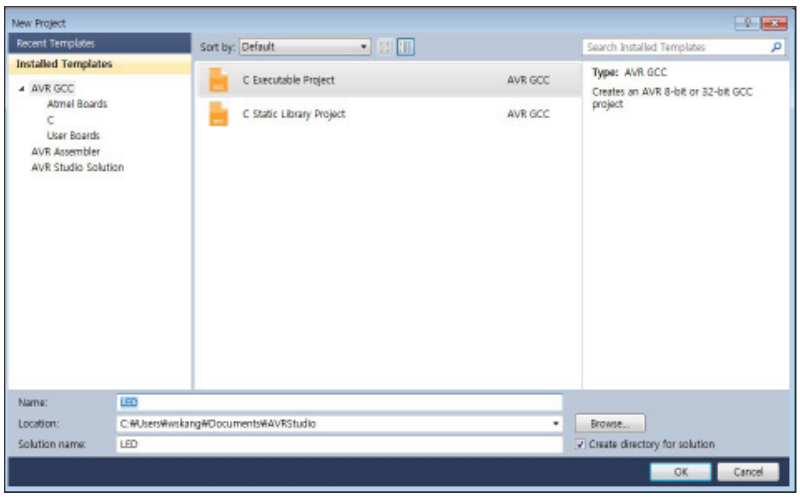
01 실행

AVR Studio 를 실행합니다.



02 New Project

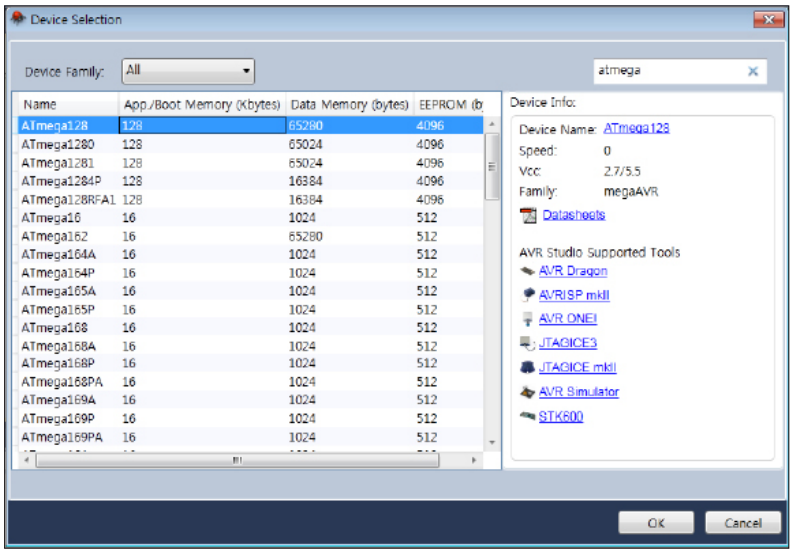
New Project 를 클릭합니다.



03 GCC 선택

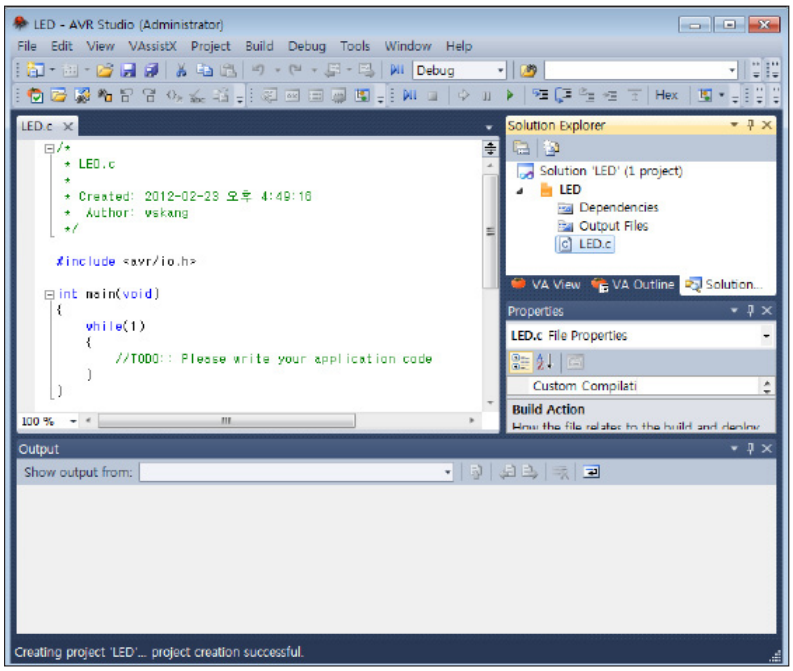
C Executable Project 를 선택합니다.

Name 을 "LED"로 입력하고, Create directory for solution 을 클릭하고, OK 버튼을 누릅니다.



04 ATmega 선택

우측 상단에 atmega 를 검색하여, 리스트중에 ATmega128 을 선택합니다.



05 Editor

편집창에 코딩을 시작합니다.

아래 예제를 따라하세요.

예제

DRC의 LED들이 Servo(0번핀)부터 Power(6번핀)까지 차례로 켜지는 것을 반복하는 예제입니다.

LED.c

```
//CPU 클럭 설정
#define F_CPU 16000000

//ATmega128의 레지스터 등이 정의되어 있음
#include <avr/io.h>
//_delay_ms 등 딜레이 함수가 정의되어 있음
#include <util/delay.h>

int main(void)
{
	//8비트 변수 선언
	unsigned char i;

	//포트 C의 입출력 방향 설정
	DDRC = 0b01111111;
	//포트 C의 출력 값 설정
	PORTC = 0b01111111;

	//while 안의 내용을 무한 반복
	while(1)
	{
		//LED가 차례로 하나씩 더 켜지도록 동작을 반복함
		for(i=0b00000001;i<=0b01111111;i=(i<<1)+1){
			PORTC = ~i;		//Active Low이므로 i 변수의 값을 반전하여 출력
			_delay_ms(200);		//200ms동안 대기
		}
	}

	//함수의 형태에 맞게 정수값 반환
	return 1;
```

예제설명

```
#define F_CPU 16000000
```

#define 문은 코드 전처리 명령 중 하나로, 앞으로 F_CPU라는 값이 나오면 16000000으로 대체한다는 뜻입니다. DRC의 ATmega128은 16MHz의 클럭 속도로 동작합니다. 클럭 속도에 비례해서 명령 처리 속도가 달라지므로, 이 명령을 통해서 이 프로그램이 동작할 속도를 명시해 주어야 delay 함수 등이 올바르게 동작합니다.

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

avr/io.h 헤더 파일에는 DDRX, PORTX 레지스터 등 AVR 제어를 위해 필요한 레지스터 등이 정의되어 있고, util/delay.h 헤더 파일에는 _delay_ms()등 자주 사용하는 딜레이 함수가 정의되어 있으므로 #include 문을 사용하여 포함시켜 줍니다.

```
DDRC = 0b01111111;
PORTC = 0b01111111;
```

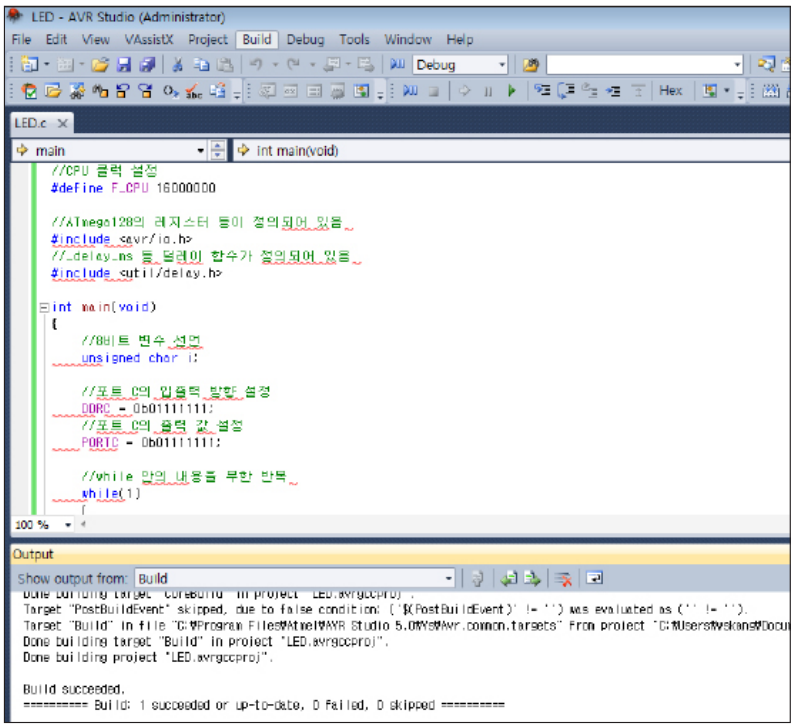
포트 C의 0~6번 핀을 출력으로 설정하고, 출력 값을 1로 만듭니다. 그러면 모든 LED가 꺼지게 됩니다. 여기서 0b01111111은 C언어에서 2진수를 나타내는 표기법입니다.

```
for(i=0b00000001;i<=0b01111111;i=(i<<1)+1){
        PORTC = ~i;
        _delay_ms(200);
```

for문에서 i는 0b00000001로 초기화 됩니다. 이를 ~(bitwise not) 연산자 처리하면 0b11111110이 되며, 이것이 PORTC의 값에 들어가면 0번 비트에 해당하는 LED가 켜지게 됩니다. _delay_ms() 함수를 통해 200ms만큼 기다린 후에 i는 i=(i<<1)+1로 업데이트 됩니다. 이 연산은 i 값을 한 비트만큼 왼쪽으로 shift하고 1을 더합니다. 예를 들어 0b00001111은 shift되어 0b00011110이 되고 1을 더하면 0x00011111이 되어 i 값에 1인 비트를 하나 추가하게 됩니다. 따라서 for문을 계속 실행하다 보면 0번 비트부터 시작해서 차례로 LED가 하나씩 더 켜집니다. for문은 i<=0x01111111을 만족하는 동안 실행되므로, 모든 LED가 다 켜진 후에는 for문이 끝납니다. 이 for문은 while(1)문에 둘러싸여 있으므로 다시 for문이 처음부터 시작되며, LED는 0번 비트부터 다시 차례로 켜지게 됩니다.

실행

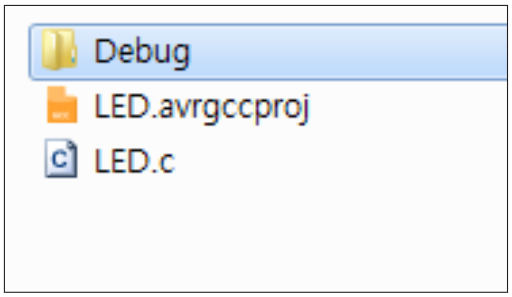
다운로드하고 실행 시 Servo LED부터 Power LED까지 LED가 하나씩 추가로 켜집니다. LED 7개가 다 켜지고 난 후에는 다시 Servo LED만 켜지고, 차례로 하나씩 켜지는 것을 반복합니다.



01 Build

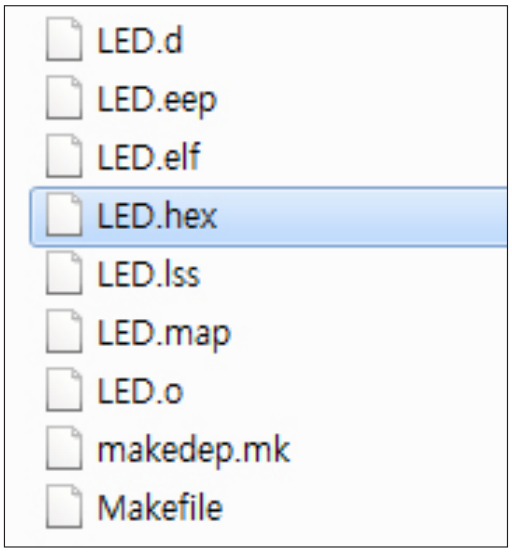
Build 메뉴를 선택하여 Build Solution을 클릭하여 Build 시킵니다.

1 succeeded 0 failed 가 나오면 에러없이 컴파일 완료됐다는 것을 의미합니다.



02 Debug 폴더

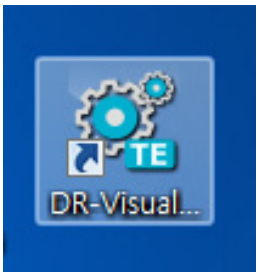
Build 후에 소스가 있는 폴더에 들어가면 Debug 폴더가 생성 되어있습니다. 클릭하여 폴더 안의 내용을 봅니다.



03 hex 확인

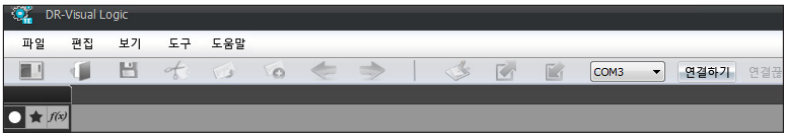
Debug 폴더 안의 hex 파일 확인합니다. Hex 파일이 제어기에 포팅시키는 파일입니다.

DR-Visual Logic 에서 펌웨어 복구를 실시합니다.



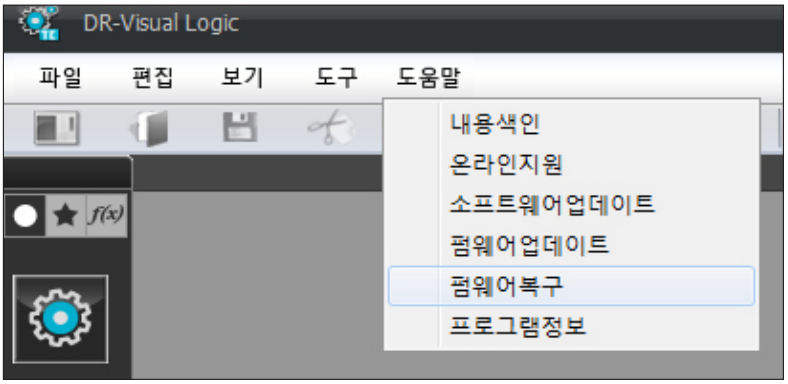
04 Visual Logic

DR-Visual Logic 을 실행시킵니다.



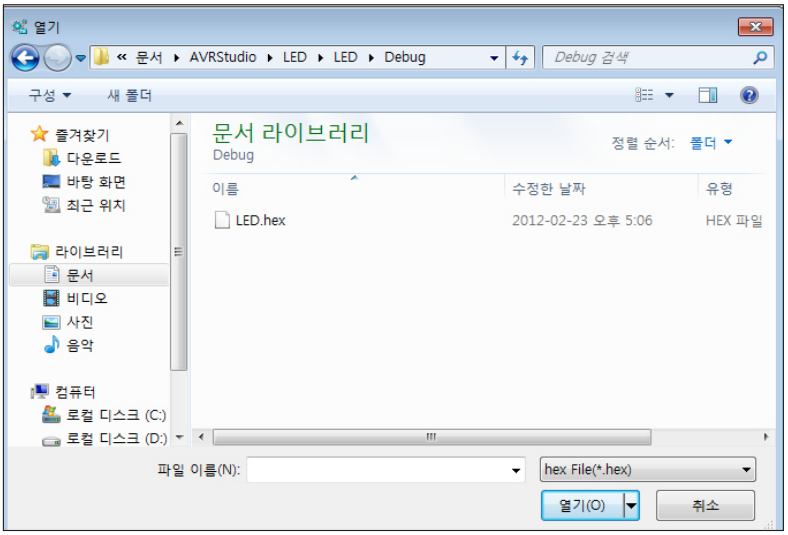
05 연결하기

COM 번호를 설정하고 연결하기를 클릭합니다.



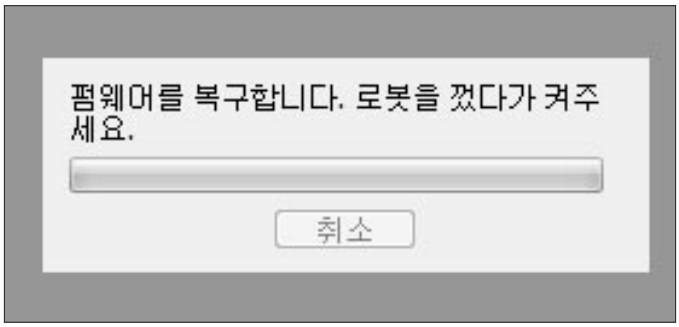
06 펌웨어 복구

도움말 메뉴에서 펌웨어복구를 클릭합니다.



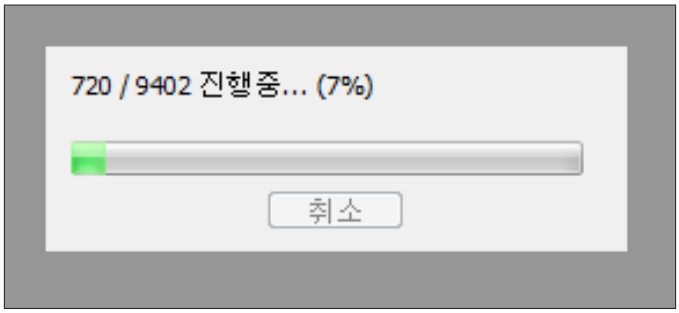
07 불러오기

Debug 폴더에서 Hex 파일을 불러옵니다



08 펌웨어복구

로봇을 껐다가 켜주세요 라는 문구가 나옵니다.
제어기의 전원버튼을 한번 껐다가 다시 켜주세요.



09 복구시작

펌웨어 복구가 시작됩니다.

10 실행

Servo LED부터 Power LED까지 LED가 하나씩 추가로 켜집니다. LED 7개가 다 켜지고 난 후에는 다시 Servo LED만 켜지고, 차례로 하나씩 켜지는 것을 반복합니다.

CH.2 : 버튼 입력하기

ATmega128의 디지털 입출력을 사용하여 DRC에 장착된 버튼을 입력 받아 LED를 제어합니다.

AVR 원리

앞서 Chapter 1에서는 디지털 입출력 포트의 기능 중 출력을 이용해 LED를 제어해 보았습니다. 이번에는 버튼으로 입력을 받아서 그 결과에 따라서 LED를 제어하는 방법을 알아봅니다.

디지털 입출력에 관련된 레지스터는 아래와 같습니다. ATmega128의 레지스터는 모두 8개의 비트(1바이트)로 구성되어 있습니다. DDRX와 PORT X에서 X는 포트 이름(A~G)을 가리키는 문자입니다.

■ DDRX : PORT X 데이터 방향 레지스터

비트	7	6	5	4	3	2	1	0
	DDX7	DDX6	DDX5	DDX4	DDX3	DDX2	DDX1	DDX0
읽기/쓰기	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
초기값	0	0	0	0	0	0	0	0

■ PORTX : PORT X 데이터 레지스터

비트	7	6	5	4	3	2	1	0
	PORTX7	PORTX6	PORTX5	PORTX4	PORTX3	PORTX2	PORTX1	PORTX0
읽기/쓰기	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
초기값	0	0	0	0	0	0	0	0

■ PINX : PORT X 입력 핀 어드레스

비트	7	6	5	4	3	2	1	0
	PINX7	PINX6	PINX5	PINX4	PINX3	PINX2	PINX1	PINX0
읽기/쓰기	R	R	R	R	R	R	R	R
초기값	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Chapter 1에서 설명한 대로 DDRX 레지스터는 포트 X의 입출력 방향을 결정하는 레지스터이며, PORTX 레지스터는 포트 X에 출력할 논리 값을 저장하는 레지스터입니다. PINX 레지스터는 현재 포트의 논리값을 읽을 수 있는 어드레스입니다. 입출력 여부와 상관 없이 PINX 레지스터를 읽음으로써 PORT X의 현재 논리 값을 읽을 수 있습니다. PINX에는 값을 쓸 수가 없으며, 오로지 읽을 수만 있습니다.

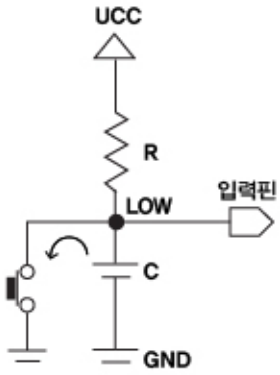
H/W 원리

택트 스위치(tact switch)는 푸시 스위치(push switch)의 일종으로, 동작 시 딸각거리는 느낌이 나는 특징 때문에 영어로 '촉각을 이용한'이라는 뜻의 tactile이라는 단어에서 택트 스위치라는 이름을 가지게 되었습니다. 택트 스위치의 두 핀은 평소에는 떨어져 있다가 딸각하고 눌린 상태에는 연결되어 전류가 흐르게 됩니다. 택트 스위치의 종류에는 2핀과 4핀이 있는데, 4핀 스위치도 내부적으로는 2핀씩 연결되어 있어 2핀 스위치랑 동작에 차이가 없습니다.

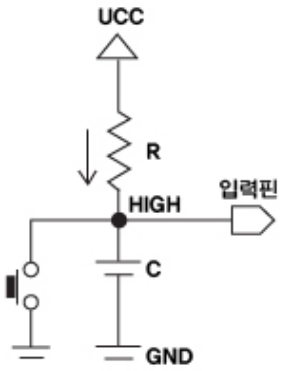
택트 스위치를 ATmega128에 입력하는 회로는 아래와 같습니다(그림 01). 스위치가 눌리지 않았을 때, 회로는 콘덴서 C가 저항 R을 통해서 충전되어 디지털 입력 핀에는 VCC와 같은 전압이 들어갑니다(그림 02). 따라서 스위치가 눌리지 않았을 때 입력 값은 HIGH가 됩니다. 스위치가 눌린 경우, 디지털 입력 핀은 GND와 직접 연결되어 콘덴서 C가 방전됩니다(그림 03). 디지털 입력 핀에는 GND 전압이 들어가므로, 입력 값이 LOW가 됩니다. 이 회로에서 저항 R이 없다면, 스위치가 안 눌린 경우에 입력 핀은 연결된 전압이 없는 플로팅 상태가 되어 HIGH라고 할 수도 LOW라고 할 수도 없는 상태가 됩니다. 이 때 저항 R이 VCC과 입력 핀을 연결해줌으로써, 입력 핀의 전압 값을 HIGH로 끌어올려 주는 역할을 합니다. 그래서 이 경우 저항 R을 풀업(pull-up) 저항이라고 합니다.

DRC에서는 PORT A의 0~5번 핀을 버튼 입력에 사용하며, 핀 번호에 대응되는 버튼 이름은 아래 표와 같습니다.

핀 번호	7	6	5	4	3	2	1	0
버튼 이름	–	–	Right	Left	Down	Up	OK	Mode



[그림1]



[그림2]

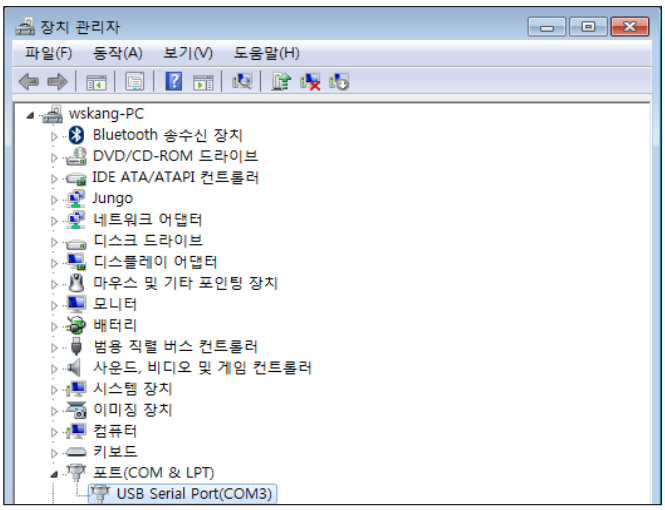
[그림3]

준비

하드웨어 준비하기

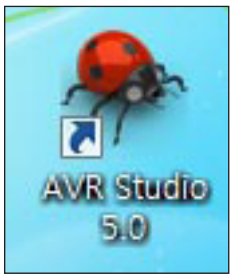
01 연결

DRC 제어기 아답터로 전원을 연결하고, USB to Serial Cable 로 PC 와 DRC 를 연결합니다.



02 COM port 연결

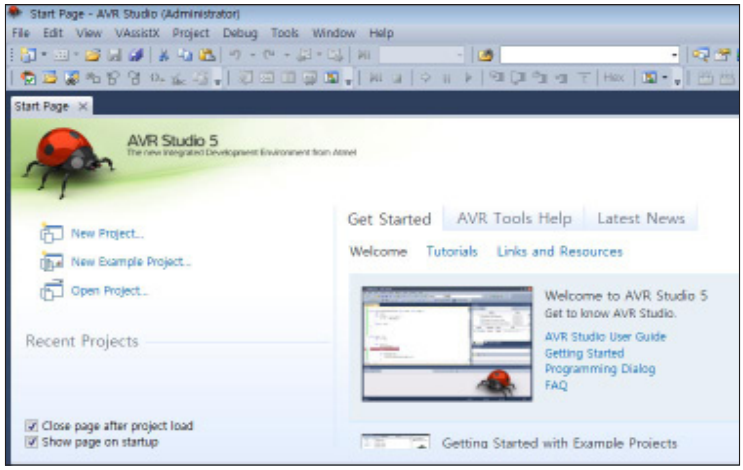
내컴퓨터 〉속성 〉장치관리자에서 COM 연결 확인합니다.



AVR Studio 실행 및 코딩 준비

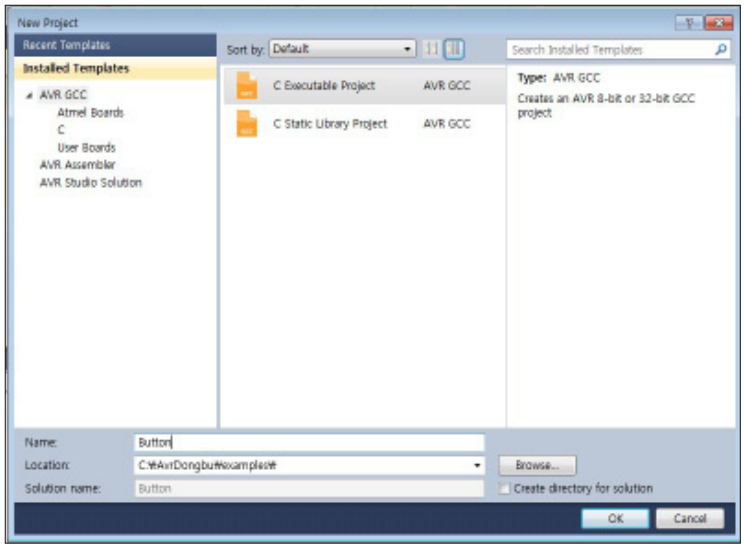
01 실행

AVR Studio 를 실행합니다.



02 New Project

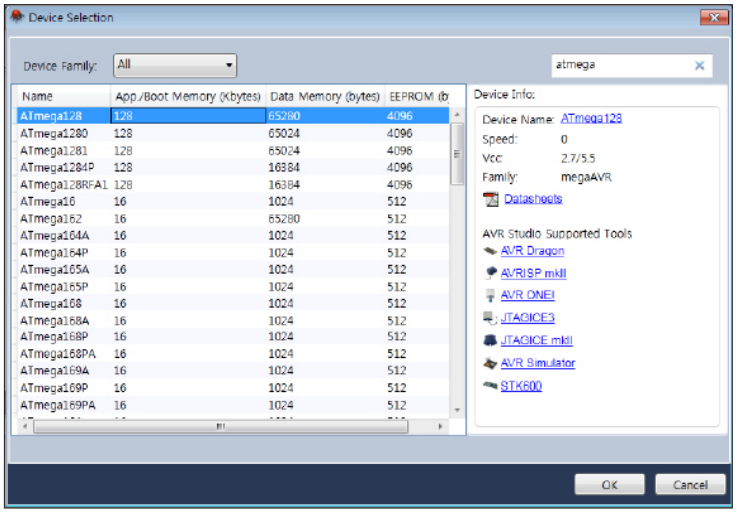
New Project 를 클릭합니다.



03 GCC 선택

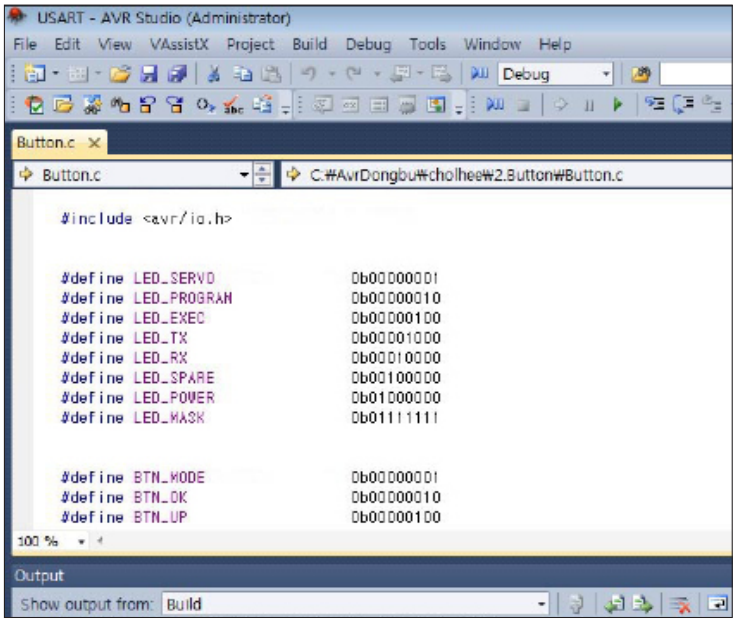
C Executable Project 를 선택합니다.

Name 을 “Button”로 입력하고, Create directory for solution 을 클릭하고, OK 버튼을 누릅니다.



04 ATmega 선택

우측 상단에 atmega 를 검색하여, 리스트중에 ATmega128 을 선택합니다.



05 Editor

편집창에 코딩을 시작합니다.

아래 예제를 따라하세요.

예제

DRC의 버튼을 누르면 그 버튼과 대응되는 위치의 LED가 켜지는 예제입니다. 누르는 각 버튼에 따라서 켜지는 LED는 아래 표와 같습니다.

버튼	LED
Mode	Program
OK	EXEC
Up	TX
Down	RX
Left	Servo
Right	Power

Button.c

```
//ATmega128의 레지스터 등이 정의되어 있음
#include <avr/io.h>

//LED의 이름에 해당하는 포트 C의 핀을 정의
#define LED_SERVO                0b00000001
#define LED_PROGRAM              0b00000010
#define LED_EXEC                 0b00000100
#define LED_TX                   0b00001000
#define LED_RX                   0b00010000
#define LED_SPARE                0b00100000
#define LED_POWER                0b01000000
#define LED_MASK                 0b01111111

//BTN의 이름에 해당하는 포트 A의 핀을 정의
#define BTN_MODE                 0b00000001
#define BTN_OK                   0b00000010
#define BTN_UP                   0b00000100
#define BTN_DOWN                 0b00001000
#define BTN_LEFT                 0b00010000
#define BTN_RIGHT                0b00100000
#define BTN_MASK                 0b00111111

int main(void)
{
```

```
//8비트 변수 선언
unsigned char ucButton, ucLED;

//포트 A의 입출력 방향 설정
DDRA = 0b00000000;

//포트 C의 입출력 방향 설정
DDRC = 0b01111111;
//포트 C의 출력 값 설정
PORTC = 0b01111111;

//while 안의 내용을 무한 반복
while(1)
{
        //포트 A로 부터 버튼 값 읽어오기
        ucButton = (~PINA) & BTN_MASK;
        //ucLED를 LED가 모두 꺼진 상태로 초기화
        ucLED = 0;

        //MODE 버튼이 눌렸다면 PROGRAM LED를 켬
        if(ucButton & BTN_MODE){
                ucLED |= LED_PROGRAM;
        }
        //OK 버튼이 눌렸다면 EXEC LED를 켬
        if(ucButton & BTN_OK){
                ucLED |= LED_EXEC;
        }
        //UP 버튼이 눌렸다면 TX LED를 켬
        if(ucButton & BTN_UP){
                ucLED |= LED_TX;
        }
        //DOWN 버튼이 눌렸다면 RX LED를 켬
        if(ucButton & BTN_DOWN){
                ucLED |= LED_RX;
```

```
                }
                //LEFT 버튼이 눌렸다면 SERVO LED를 켬
                if(ucButton & BTN_LEFT){
                        ucLED |= LED_SERVO;
                }
                //RIGHT 버튼이 눌렸다면 POWER LED를 켬
                if(ucButton & BTN_RIGHT){
                        ucLED |= LED_POWER;
                }

                //ucLED의 값에 맞게 PORTC의 값을 바꿔주기
                PORTC = (~ucLED) & LED_MASK;
        }

    //함수의 형태에 맞게 정수값 반환
        return 1;
    }
```

예제설명

```
#include <avr/io.h>
```

avr/io.h 헤더 파일에는 DDRX, PORTX 레지스터 등 AVR 제어를 위해 필요한 레지스터 등이 정의되어 있으므로 #include 문을 사용하여 포함시켜 줍니다.

```
#define LED_SERVO                0b00000001
...
#define BTN_MASK                 0b00111111
```

#define 문은 코드 전처리 명령 중 하나로, #define LED_SERVO 0b00000001은 앞으로 LED_SERVO라는 것이 나오면 0b00000001로 대체해서 컴파일한다는 뜻입니다. DRC의 버튼과 LED에는 각각 비트에 해당하는 이름이 있는데, 매번 그 버튼과 LED를 다룰 때 마다 0b00000001 같은 알아보기 힘든 숫자를 사용하는 것 보다 LED_SERVO 같이 해당하는 이름을 사용하는 것이 프로그램을 작성하기 쉽기 때문에 선언합니다.

```
unsigned char ucButton, ucLED;
```

8비트 변수인 ucButton과 ucLED를 선언합니다. 이 변수들은 각각 현재 버튼과 LED 상태를 나타내는 용도로 사용됩니다. while문 내부에서 ucButton과 ucLED는 실제 포트의 입출력과는 반대로 저장됩니다. 즉 PINA와는 달리 ucButton에서는 버튼이 눌린 상태가 1이고, 눌리지 않은 상태가 0입니다. 또한 PORTC와는 달리 ucLED에서는 LED가 켜진 상태가 1이고, 꺼진 상태가 0입니다.

```
DDRA = 0b00000000;
DDRC = 0b01111111;
PORTC = 0b01111111;
```

포트 A의 0~5번 핀을 입력으로 설정합니다. 버튼에 해당하는 핀들을 입력으로 설정하는 것입니다. 포트 C의 0~6번 핀을 출력으로 설정하고, 출력 값을 1로 만듭니다. 그러면 모든 LED가 꺼지게 됩니다.

```
ucButton = (~PINA) & BTN_MASK;
ucLED = 0;
```

while문은 ucButton에 버튼 값을 읽어오는 것부터 시작합니다. PINA에서 읽은 버튼 값은 버튼이 눌렸을 때 0, 안 눌렸을 때 1이므로, ~ 연산자를 처리해 반대로 바꾸어 줍니다. 그리고 BTN_MASK(0b00111111)와 &(bitwise AND) 처리하여 버튼과는 관계 없는 6, 7번 비트를 0으로 만들어 줍니다. 그러면 ucButton에는 0b00XXXXXX값이 들어가게 되며, X에는 각 비트에 해당하는 버튼이 눌렸는지 여부가 저장됩니다. 그 후에는 ucLED를 LED가 모두 꺼진 상태인 0으로 초기화합니다.

```
if(ucButton & BTN_MODE){
        ucLED |= LED_PROGRAM;
}
```

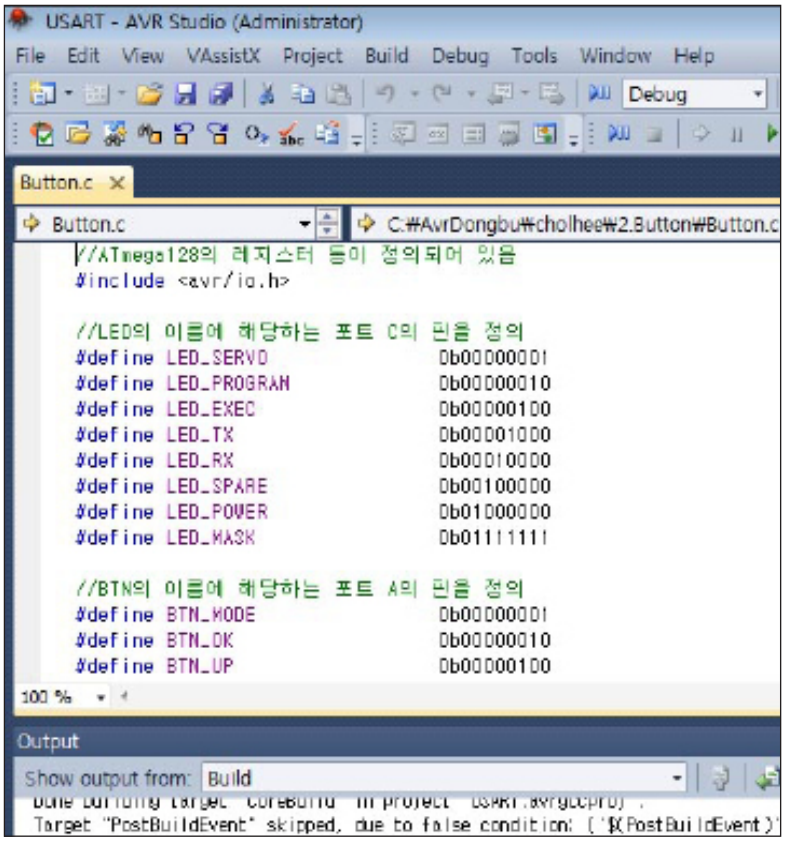
if(ucButton & BTN_MODE)라는 조건문은 ucButton 값을 BTN_MODE(0b00000001)와 & 연산합니다. 결과값은 ucButton의 0번 비트가 1이라면(Mode 버튼이 눌렸다면) 0b00000001, 0이라면 그냥 0이 됩니다. MODE 버튼이 눌렸다면 if문의 괄호 안 값이 참이 되므로 안의 내용이 실행됩니다. if문 안의 ucLED |= LED_PROGRAM은 ucLED = ucLED | LED_PROGRAM의 간략한 표현입니다. ucLED 값과 LED_PROGRAM(0b00000010)을 |(bitwise OR) 처리해서 ucLED 값을 그것으로 바꾼다는 뜻입니다. 이 계산의 결과로 ucLED의 Program LED에 해당하는 1번 비트가 1이 됩니다.
그 이후의 if문도 마찬가지입니다. Mode 버튼이 눌렸을 때 ucLED의 값이 Program LED에 해당하는 비트가 1이 되는 것과 같이, OK 버튼이 눌렸다면 EXEC LED, UP 버튼이 눌렸다면 TX LED, DOWN 버튼이 눌렸다면 RX LED, … 에 해당하는 비트가 1이 됩니다.

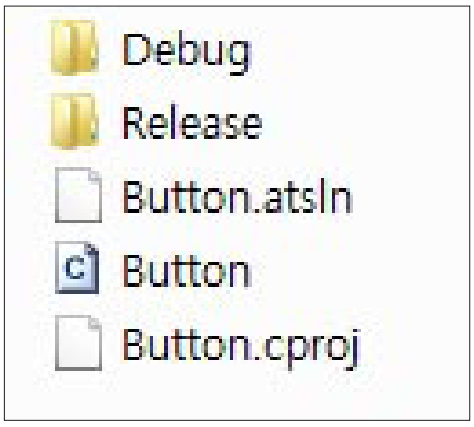
```
PORTC = (~ucLED) & LED_MASK;
```

while문의 마지막에서는 이렇게 버튼 값에 따라서 바꾼 ucLED 값을 실제 포트 C에 적용합니다. ucLED와 PORTC에서 켤 LED를 나타내는 방식이 반대이므로 ~연산 처리하고, 버튼과 마찬가지로 LED_MASK(0b01111111)를 &처리하여 LED와 상관없는 7번째 비트를 0으로 만들어줍니다. 이 값을 PORTC에 대입하여 LED를 제어하게 됩니다. while문이 끝나면 다시 처음으로 돌아가서, 버튼 입력을 읽고 그 값에 따라서 LED를 제어하는 것을 계속 반복합니다.

실행

다운로드하고 실행 시 버튼이 눌림에 따라서 해당하는 LED가 켜집니다.





01 Build

Build 메뉴를 선택하여 Build Solution 을 클릭하여 Build 시킵니다.

1 succeeded 0 failed 가 나오면 에러없이 컴파일 완료됐다는 것을 의미합니다.

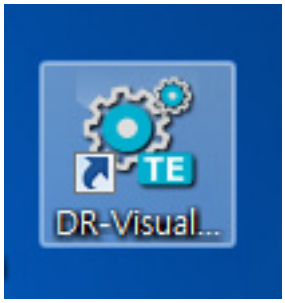
02 Debug 폴더

Build 후에 소스가 있는 폴더에 들어가면 Debug 폴더가 생성 되어있습니다. 클릭하여 폴더 안의 내용을 봅니다.

03 hex 확인

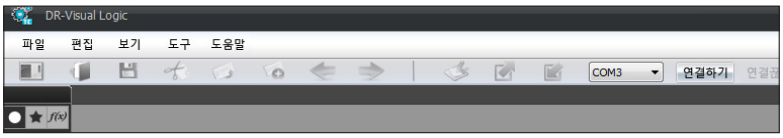
Debug 폴더 안의 hex 파일 확인합니다. Hex 파일이 제어기에 포팅시키는 파일입니다.

DR-Visual Logic 에서 펌웨어 복구를 실시합니다.



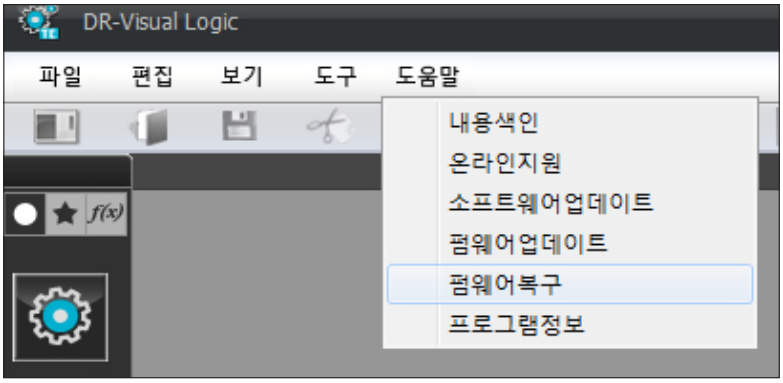
04 Visual Logic

DR-Visual Logic 을 실행시킵니다.



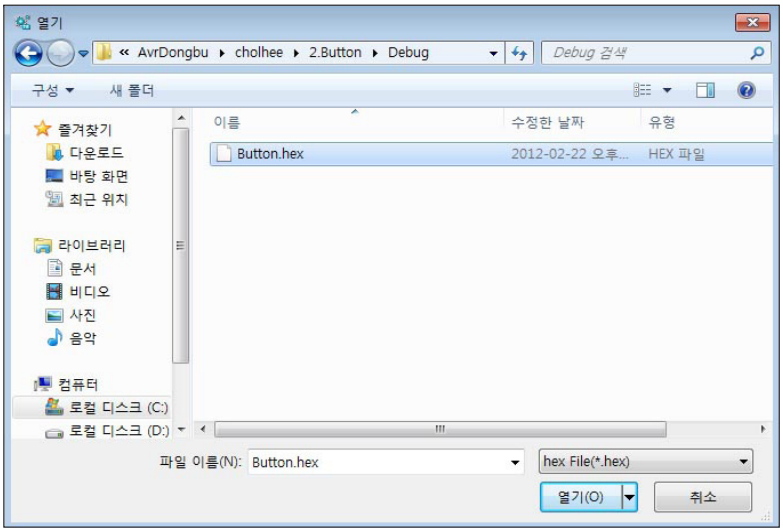
05 연결하기

COM 번호를 설정하고 연결하기를 클릭합니다.



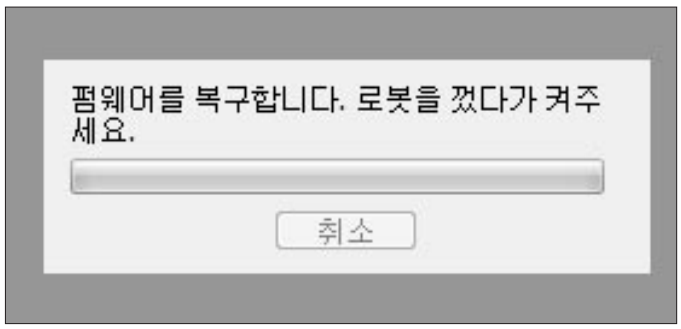
06 펌웨어 복구

도움말 메뉴에서 펌웨어복구를 클릭합니다.



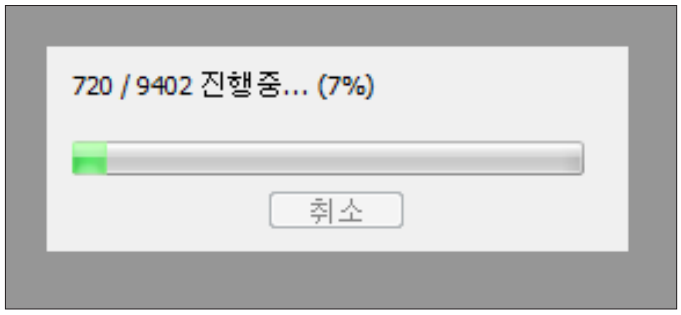
07 불러오기

Debug 폴더에서 Hex 파일을 불러옵니다



08 펌웨어복구

로봇을 껐다가 켜주세요 라는 문구가 나옵니다.
제어기의 전원버튼을 한번 껐다가 다시 켜주세요.



09 복구시작

펌웨어 복구가 시작됩니다.

10 실행

버튼이 눌림에 따라서 해당하는 LED가 켜집니다

CH.3 : 버저 울리기

ATmega128의 디지털 입출력을 사용하여 DRC에 장착된 버저로 비프음을 내봅니다.

AVR 원리

앞서 Chapter 2에서는 버튼으로 입력을 받아 LED를 제어하는 방법을 알아보았습니다. 이번에는 LED 대신에 출력 핀으로 버저를 제어해서 음을 울리도록 해봅니다. 디지털 입출력에 관련된 레지스터는 DDRX, PORTX, PINX로, Chapter 2의 설명을 참조하면 됩니다.

H/W 원리

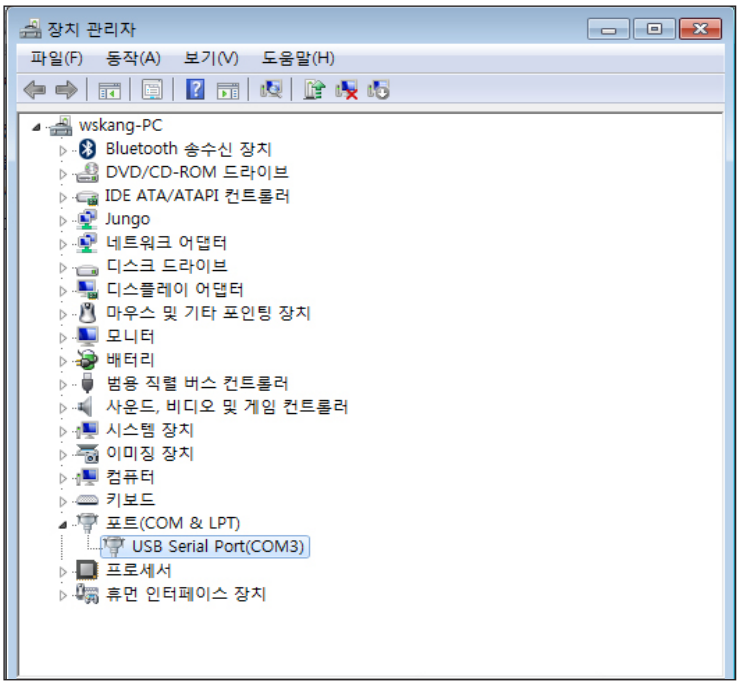
DRC에 장착된 버저의 내부에는 코일이 들어 있습니다. 버저의 흐르는 전류가 어떤 주파수로 주기적으로 변하면, 코일의 자기장 세기도 변하게 됩니다. 자기장 세기가 변하면 버저의 진동편이 울려서, 그 떨림에 의해서 소리가 나게 됩니다.
버저를 ATmega128에 연결하는 회로는 아래와 같습니다(그림 00). ATmega128의 포트 B의 7번 핀(PB7)이 트랜지스터를 통해서 버저와 연결되어 있습니다. 버저의 소모 전류가 ATmega128이 내보낼 수 있는 전류보다 더 크므로, 출력 핀에 버저를 바로 연결하면 전류가 부족해서 오작동하거나 핀에 손상을 줄 수도 있습니다. 그래서 출력 핀과 버저 사이에 트랜지스터를 삽입해서, PB7에서 소량의 출력 전류 만으로도 버저에 많은 전류를 흘릴 수 있도록 합니다.
출력 핀의 출력 값을 일정한 주기로 바꾸어주면 버저에 흐르는 전류도 그와 같은 주기로 변화하게 되며, 그 전류에 의해서 버저에서 소리가 납니다. 더 짧은 주기(높은 주파수)로 출력 핀을 바꾸어주면 버저에서 나는 소리도 그와 비례해서 더 높은 음이 납니다.
이번 예제에서는 delay 함수를 사용해서 PB7을 일정 주기로 바꾸어 음을 내 봅니다.

준비

하드웨어 준비하기

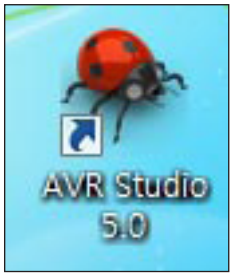
01 연결

DRC 제어기 아답터로 전원을 연결하고, USB to Serial Cable 로 PC 와 DRC 를 연결합니다.



02 COM port 연결

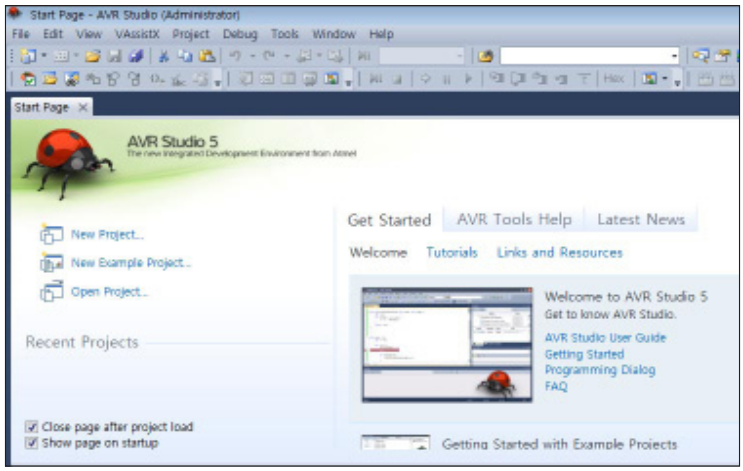
내컴퓨터 〉 속성 〉 장치관리자에서 COM 연결 확인합니다.



AVR Studio 실행 및 코딩 준비

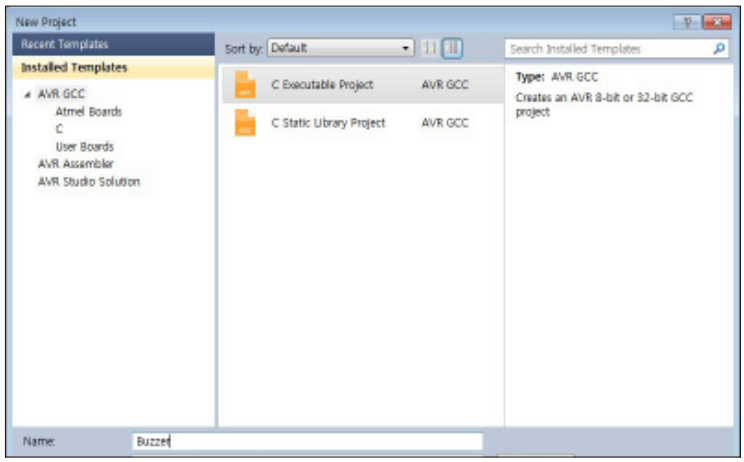
01 실행

AVR Studio 를 실행합니다.



02 New Project

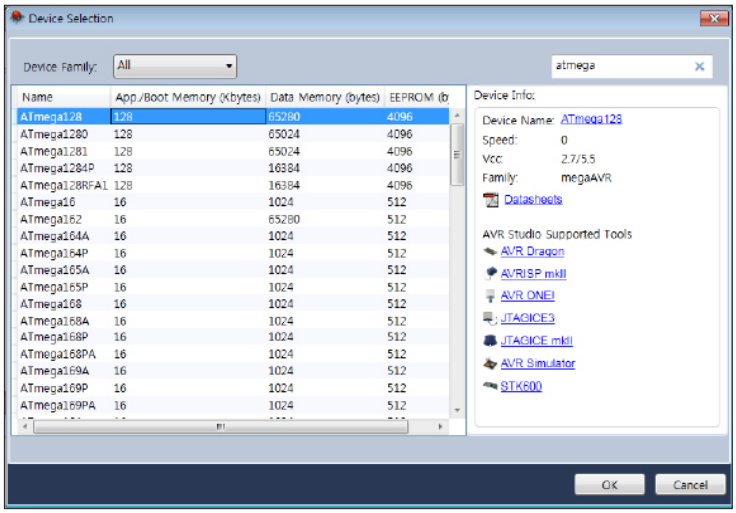
New Project 를 클릭합니다.



03 GCC 선택

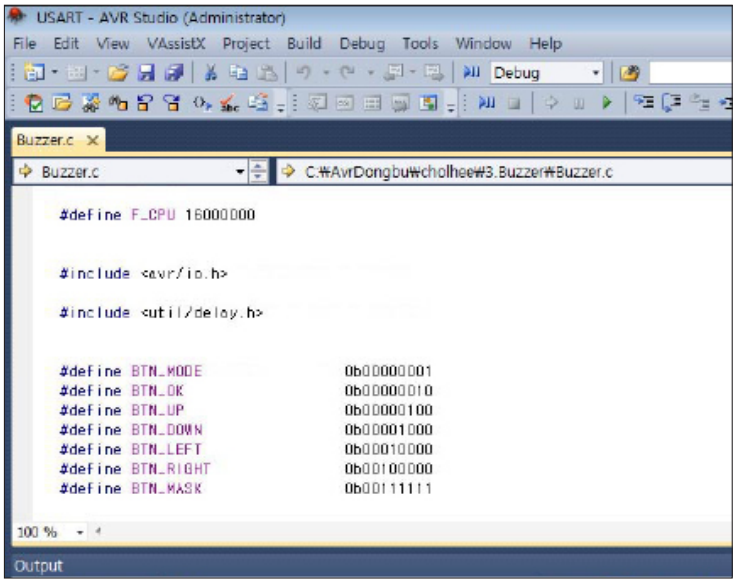
C Executable Project 를 선택합니다.

Name 을 “Buzzer”로 입력하고, Create directory for solution 을 클릭하고, OK 버튼을 누릅니다.



04 ATmega 선택

우측 상단에 atmega 를 검색하여, 리스트중에 ATmega128 을 선택합니다.



05 Editor

편집창에 코딩을 시작합니다.

아래 예제를 따라하세요.

예제

DRC의 OK 버튼을 누르면 버저가 울리는 예제입니다.

Buzzer.c

```
//CPU 클럭 설정
#define F_CPU 16000000

//ATmega128의 레지스터 등이 정의되어 있음
#include <avr/io.h>
//_delay_ms 등 딜레이 함수가 정의되어 있음
#include <util/delay.h>

//BTN의 이름에 해당하는 포트 A의 핀을 정의
#define BTN_MODE                0b00000001
#define BTN_OK                  0b00000010
#define BTN_UP                  0b00000100
#define BTN_DOWN                0b00001000
#define BTN_LEFT                0b00010000
#define BTN_RIGHT               0b00100000
#define BTN_MASK                0b00111111

int main(void)
{
        //8비트 변수 선언
        unsigned char ucButton;

        //포트 A의 입출력 방향 설정
        DDRA = 0b00000000;

        //버저 핀(PB7)의 입출력 방향 설정
        DDRB |= 0b10000000;
        //버저 핀(PB7)의 출력 값 설정
        PORTB &= ~0b10000000;
```

```
        //포트 C의 입출력 방향 설정
        DDRC = 0b01111111;
        //포트 C의 출력 값 설정
        PORTC = 0b01111111;

        //while 안의 내용을 무한 반복
        while(1)
        {
                //포트 A로 부터 버튼 값 읽어오기
                ucButton = (~PINA) & BTN_MASK;

                //OK 버튼이 눌렸다면 PB7의 값을 반대로 바꾸고 1ms 대기
                if(ucButton & BTN_OK){
                        PORTB ^= 0b10000000;
                        _delay_ms(1);
                }
        }

        //함수의 형태에 맞게 정수값 반환
        return 1;
}
```

예제설명

```
#define F_CPU 16000000
```

#define 문은 코드 전처리 명령 중 하나로, 앞으로 F_CPU라는 값이 나오면 16000000으로 대체한다는 뜻입니다. DRC의 ATmega128은 16MHz의 클럭 속도로 동작합니다. 클럭 속도에 비례해서 명령 처리 속도가 달라지므로, 이 명령을 통해서 이 프로그램이 동작할 속도를 명시해 주어야 delay 함수 등이 올바르게 동작합니다.

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

avr/io.h 헤더 파일에는 DDRX, PORTX 레지스터 등 AVR 제어를 위해 필요한 레지스터 등이 정의되어 있고, util/delay.h 헤더 파일에는 _delay_ms()등 자주 사용하는 딜레이 함수가 정의되어 있으므로 #include 문을 사용하여 포함시켜 줍니다.

```
#define BTN_MODE                0b00000001
…
#define BTN_MASK                0b00111111
```

#define BTN_MODE 0b00000001은 앞으로 BTN_MODE라는 것이 나오면 0b00000001로 대체해서 컴파일한다는 뜻입니다. DRC의 각 버튼에 해당하는 비트를 버튼의 이름으로 대체해서 프로그램을 작성하기 쉽게 하려고 선언합니다.

```
unsigned char ucButton;
```

8비트 변수인 ucButton을 선언합니다. 이 변수는 현재 버튼 상태를 나타내는 용도로 사용됩니다. while문 내부에서 ucButton은 실제 포트의 입출력과는 반대로 저장됩니다. 즉 PINA와는 달리 ucButton에서는 버튼이 눌린 상태가 1이고, 눌리지 않은 상태가 0입니다.

```
DDRA = 0b00000000;
DDRB |= 0b10000000;
PORTB &= ~0b10000000;
DDRC = 0b01111111;
PORTC = 0b01111111;
```

포트 A의 0~5번 핀을 입력으로 설정합니다. 버튼에 해당하는 핀들을 입력으로 설정하는 것입니다. 그리고 포트 B의 7번 핀을 출력으로 설정하고 출력 값을 0으로 만듭니다. 포트 C의 0~6번 핀을 출력으로 설정하고, 출력 값을 1로 만듭니다. 그러면 모든 LED가 꺼지게 됩니다.

DDRB와 PORTB에는 다른 레지스터와는 달리 = 대신 |=, &=을 사용하고 있습니다. DDRB를 0b10000000과 |(bitwise OR) 연산을 하면, 0인 비트는 DDRB의 원래 값 그대로 유지되며 1인 비트만 1로 바뀌게 됩니다. 즉 DDRB의 7번 비트만 1로 만들고, 나머지는 그대로 두겠다는 뜻입니다. 이와 반대로, PORTB &= ~0b10000000; 은 PORTB를 0b01111111(~0b10000000의 결과)과 &(bitwise AND) 연산을 하겠다는 뜻입니다. 1인 비트는 PORTB의 원래 값 그대로 유지되며, 0인 비트만 0으로 바뀌게 됩니다. 즉 PORTB의 7번 비트만 0으로 만들고, 나머지는 그대로 두겠다는 뜻입니다.

```
ucButton = (~PINA) & BTN_MASK;
```

while문은 ucButton에 버튼 값을 읽어오는 것부터 시작합니다. PINA에서 읽은 버튼 값은 버튼이 눌렸을 때 0, 안 눌렸을 때 1이므로, ~ 연산자를 처리해 반대로 바꾸어 줍니다. 그리고 BTN_MASK(0b00111111)와 &(bitwise AND) 처리하여 버튼과는 관계 없는 6, 7번 비트를 0으로 만들어 줍니다. 그러면 ucButton에는 0b00XXXXXX값이 들어가게 되며, X에는 각 비트에 해당하는 버튼이 눌렸는지 여부가 저장됩니다.

```
if(ucButton & BTN_OK){
        PORTB ^= 0b10000000;
        _delay_ms(1);
}
```

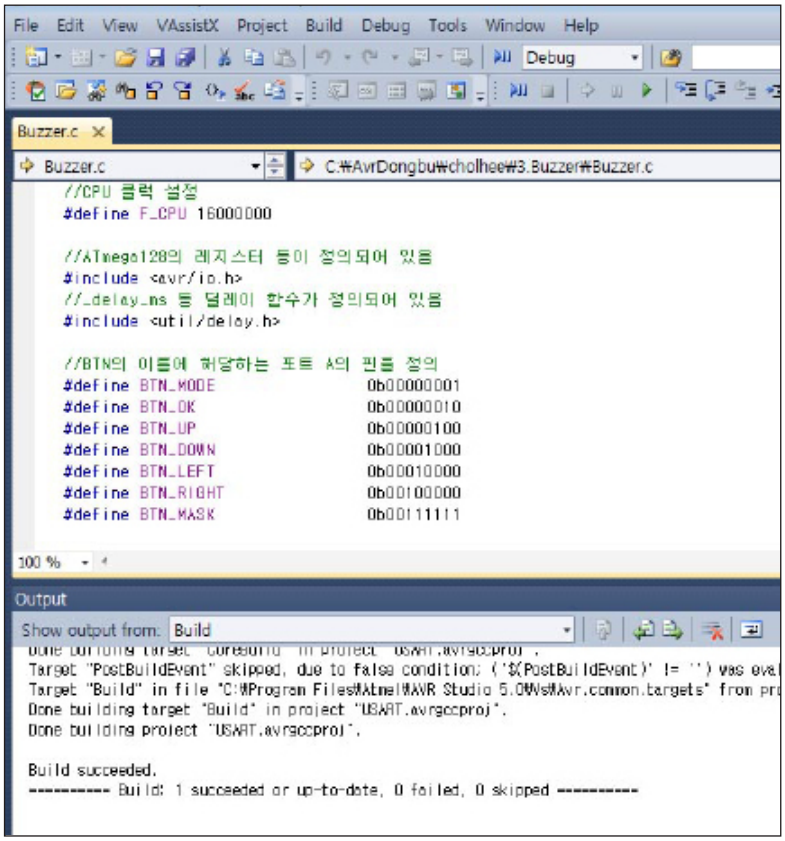
if(ucButton & BTN_OK)라는 조건문의 결과는 ucButton 값 중 OK 버튼에 해당하는 비트가 1이라면 참이 되고, 0이라면 거짓이 됩니다. 즉 OK 버튼이 눌렸다면 괄호 안의 내용이 실행됩니다.

if문 안의 PORTB ^= 0b10000000; 은 PORTB = PORTB ^ 0b10000000; 의 간략한 표현입니다. PORTB와 0b10000000을 ^(bitwise XOR) 처리하겠다는 뜻입니다. XOR(Exclusive OR) 연산은 두 비트가 서로 다를 때 1이 됩니다. 그래서 어떤 값과 1을 XOR하면, 0일 때는 0 XOR 1이므로 1이 되고, 1일 때는 1 XOR 1이므로 0이 됩니다. 따라서, PORTB ^= 0b10000000은 PORTB의 7번 비트를 0이면 1로, 1이면 0으로 바꾼다는 뜻입니다.

PB7의 출력 값을 바꾼 후에는 _delay_ms 함수를 사용해 약 1ms 동안 동작을 멈춥니다. while문이 계속 반복되므로, OK 버튼이 눌린 동안에는 약 1ms 마다 PB7의 출력 값이 변할 것입니다. 따라서 버저에는 약 2ms의 주기로 전류가 흐르게 되며, 해당하는 주파수의 비프음을 낼 것입니다.

실행

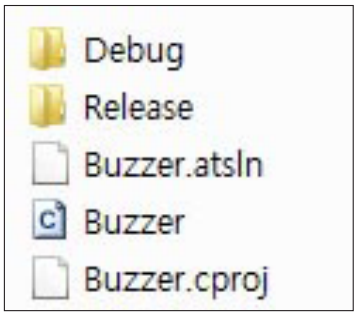
다운로드하고 실행 시 OK버튼을 누르면 버저가 울립니다.



01 Build

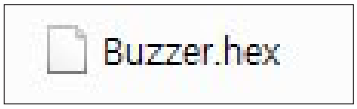
Build 메뉴를 선택하여 Build Solution을 클릭하여 Build 시킵니다.

1 succeeded 0 failed 가 나오면 에러없이 컴파일 완료됐다는 것을 의미합니다.



02 Debug 폴더

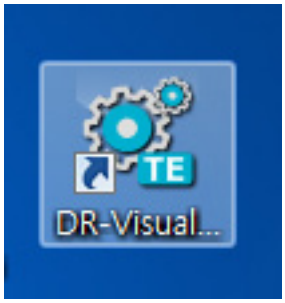
Build 후에 소스가 있는 폴더에 들어가면 Debug 폴더가 생성 되어있습니다. 클릭하여 폴더 안의 내용을 봅니다.



03 hex 확인

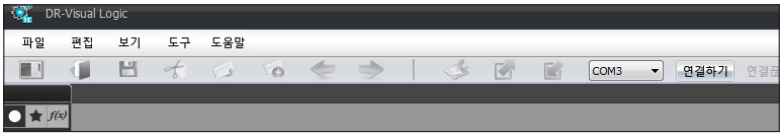
Debug 폴더 안의 hex 파일 확인합니다. Hex 파일이 제어기에 포팅시키는 파일입니다.

DR-Visual Logic 에서 펌웨어 복구를 실시합니다.



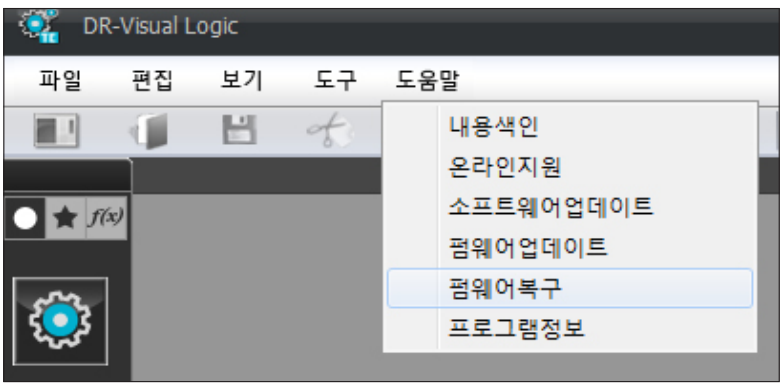
04 Visual Logic

DR-Visual Logic 을 실행시킵니다.



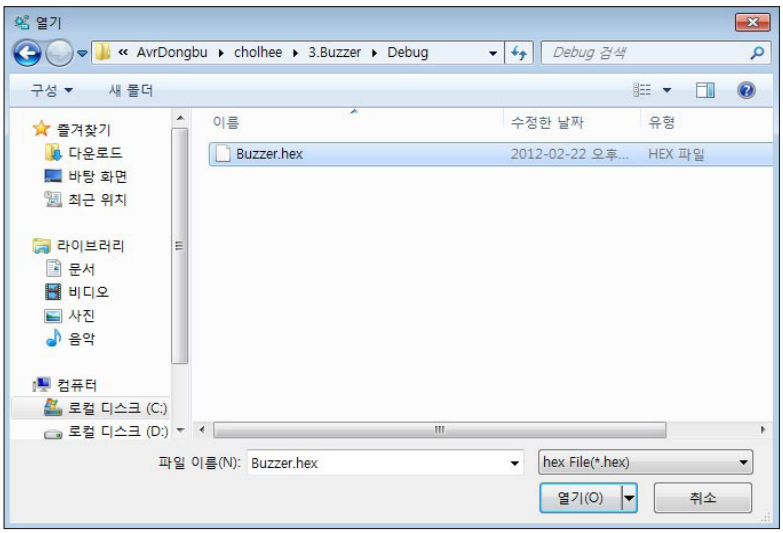
05 연결하기

COM 번호를 설정하고 연결하기를 클릭합니다.



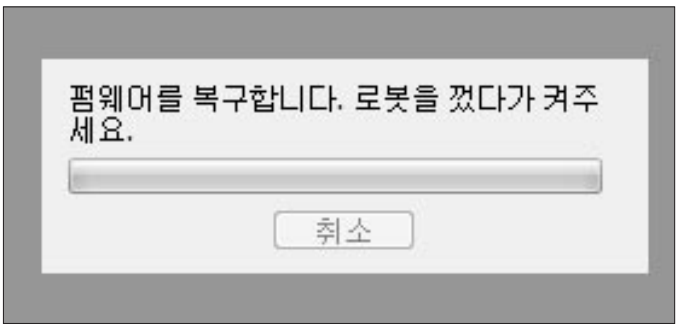
06 펌웨어 복구

도움말 메뉴에서 펌웨어복구를 클릭합니다.



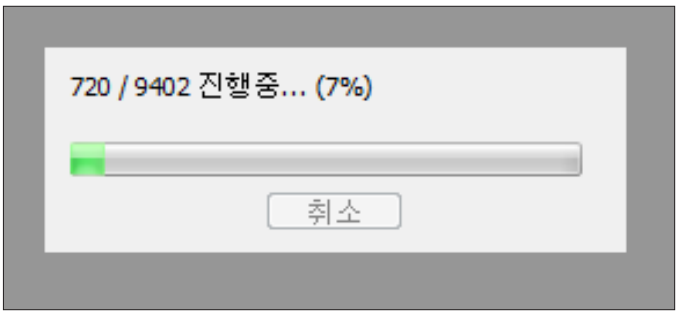
07 불러오기

Debug 폴더에서 Hex 파일을 불러옵니다



08 펌웨어복구

로봇을 껐다가 켜주세요 라는 문구가 나옵니다.
제어기의 전원버튼을 한번 껐다가 다시 켜주세요



09 복구시작

펌웨어 복구가 시작됩니다.

10 실행

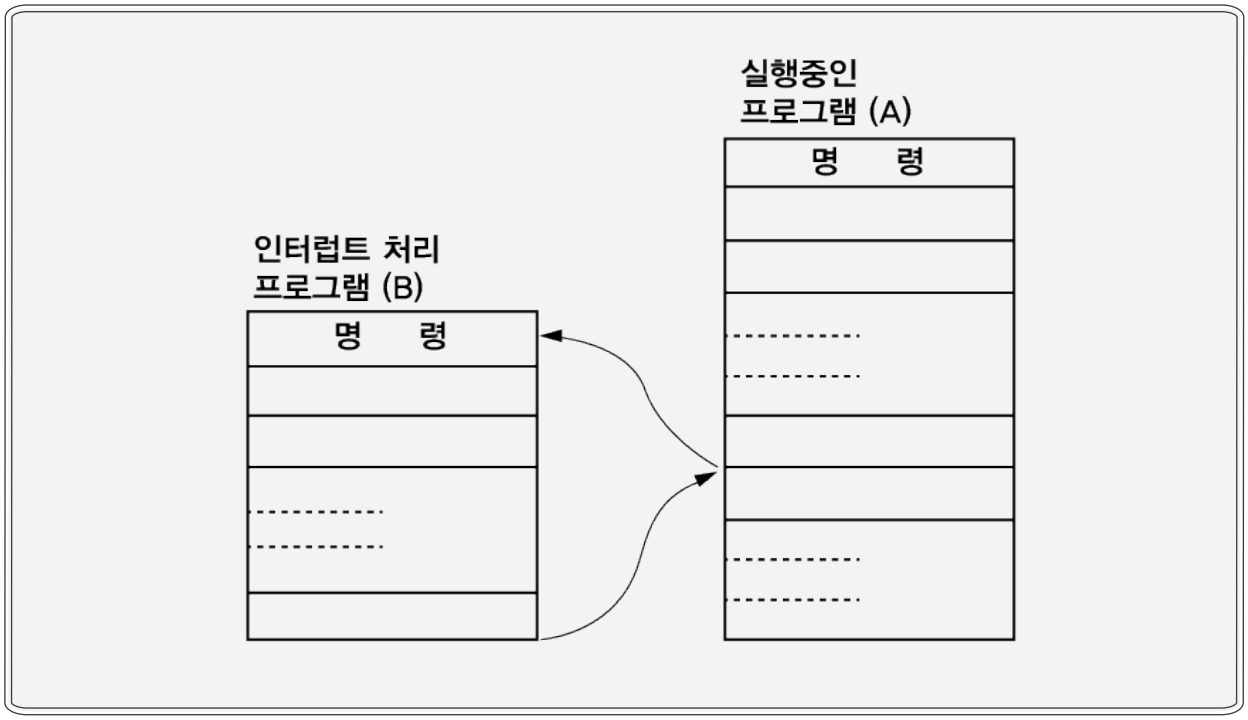
OK 버튼을 누르면 버저에서 비프음이 납니다.

CH.4 : 외부 인터럽트로 MIC 입력 받기

ATmega128의 외부 인터럽트를 사용하여 DRC에 장착된 MIC로부터 입력을 받습니다.

AVR 원리

이번에는 ATmega128의 외부 인터럽트 기능을 사용해봅니다. 인터럽트란 실행 중인 프로그램을 일시 중단하고 다른 프로그램을 끼워 넣어 실행시키는 것입니다. 인터럽트 요인이 되는 조건이 생겼을 때 실행 중인 프로그램을 중단하여 강제적으로 특정한 주소로 이동하고, 준비되어 있는 인터럽트 처리 프로그램을 실행시키며, 그 처리가 끝나면 원래의 프로그램으로 되돌아가서 계속 실행시킵니다(그림 01). 이 때 인터럽트를 처리하는 부분을 인터럽트 서비스 루틴(Interrupt Service Routine, ISR)이라고 합니다. 인터럽트를 사용하면 프로그램 처리를 좀 더 효율적으로 할 수 있으며, 인터럽트 요인이 발생하는 정확한 타이밍에 그 요인에 필요한 동작을 해줄 수 있다는 장점이 있습니다.



[그림 1]

ATmega128에서 사용할 수 있는 인터럽트 소스는 아래와 같이 매우 많습니다. 위에 쓰여진 것일수록 더 우선 순위가 높은 인터럽트입니다. 이번 챕터에서는 외부 인터럽트를 사용하여 프로그래밍을 해보고, 앞으로 남은 챕터에서는 타이머/카운터 오버플로우와 USART 수신 완료 인터럽트 등을 사용해볼 것입니다.

벡터 번호	벡터 어드레스	인터럽트 소스	인터럽트 설명
1	$0000	RESET	리셋
2	$0002	INT0	외부 인터럽트 0
3	$0004	INT1	외부 인터럽트 1
4	$0006	INT2	외부 인터럽트 2
5	$0008	INT3	외부 인터럽트 3
6	$000A	INT4	외부 인터럽트 4
7	$000C	INT5	외부 인터럽트 5
8	$000E	INT6	외부 인터럽트 6
9	$0010	INT7	외부 인터럽트 7
10	$0012	TIMER2 COMP	타이머/카운터2 비교 매치
11	$0014	TIMER2 OVF	타이머/카운터2 오버플로우
12	$0016	TIMER1 CAPT	타이머/카운터1 입력 캡처
13	$0018	TIMER1 COMPA	타이머/카운터1 비교 매치 A
14	$001A	TIMER1 COMPB	타이머/카운터1 비교 매치 B
15	$001C	TIMER1 OVF	타이머/카운터1 오버플로우
16	$001E	TIMER0 COMP	타이머/카운터0 비교 매치
17	$0020	TIMER0 OVF	타이머/카운터0 오버플로우
18	$0022	SPI, STC	SPI 시리얼 통신 완료
19	$0024	USART0, RX	USART0 수신 완료
20	$0026	USART0, UDRE	USART0 데이터 레지스터 비움
21	$0028	USART0, TX	USART0 송신 완료
22	$002A	ADC	ADC 변환 완료
23	$002C	EE READY	EEPROM 준비 완료
24	$002E	ANALOG COMP	아날로그 비교기
25	$0030	TIMER1 COMPC	타이머/카운터1 비교 매치 C
26	$0032	TIMER3 CAPT	타이머/카운터3 입력 캡처
27	$0034	TIMER3 COMPA	타이머/카운터3 비교 매치 A
28	$0036	TIMER3 COMPB	타이머/카운터3 비교 매치 B
29	$0038	TIMER3 COMPC	타이머/카운터3 비교 매치 C
30	$003A	TIMER3 OVF	타이머/카운터3 오버플로우
31	$003C	USART1 RX	USART1 수신 완료
32	$003E	USART1 UDRE	USART1 데이터 레지스터 비움
33	$0040	USART1 TX	USART1 송신 완료
34	$0042	TWI	TWI(I2C) 직렬 인터페이스
35	$0044	SPM READY	프로그램 메모리 저장 준비 완료

ATmega128의 외부 인터럽트는 포트 E가 사용됩니다. 포트 E의 8개 핀 PEn이 각각 인터럽트 소스 INTn에 대응됩니다. 외부 인터럽트의 설정과 관련된 레지스터는 아래와 같습니다.

SREG : AVR 상태 레지스터

비트	7	6	5	4	3	2	1	0
	I	T	H	S	V	N	Z	C
읽기/쓰기	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
초기값	0	0	0	0	0	0	0	0

SREG 레지스터는 AVR의 전체적인 상태를 나타내는 레지스터입니다. 이중 인터럽트와 관련된 부분은 7번 비트입니다.

– 비트 7번 I : 전역 인터럽트 활성화

ATmega128에서 어떤 인터럽트를 활성화 하기 위해서는 I 비트를 1로 설정해야 합니다. 각각의 인터럽트에 대한 활성화 여부는 해당하는 레지스터의 비트를 설정해서 결정하지만, 인터럽트를 활성화 시키더라도 I 비트가 0으로 되어 있다면 실제로 인터럽트가 발생하지 않습니다.

EICRA : 외부 인터럽트 상태 레지스터 A

비트	7	6	5	4	3	2	1	0
	ISC31	ISC30	ISC21	ISC20	ISC11	ISC10	ISC01	ISC00
읽기/쓰기	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
초기값	0	0	0	0	0	0	0	0

EICRA 레지스터는 외부 인터럽트 0부터 3까지의 동작 모드를 선택하는 레지스터입니다. 인터럽트 소스 하나당 2개의 비트가 할당되어 있습니다. 비트 설정에 따른 동작 모드는 아래 표에 나와 있습니다.

ISCn1	ISCn0	설명
0	0	INTn에 LOW 입력이 들어왔을 때 인터럽트가 발생
0	1	사용 안함
1	0	INTn에 하강 에지(HIGH에서 LOW로 변할 때)가 발생 시 비동기 인터럽트 요청
1	1	INTn에 상승 에지(LOW에서 HIGH로 변할 때)가 발생 시 비동기 인터럽트 요청

EICRB : 외부 인터럽트 상태 레지스터 B

비트	7	6	5	4	3	2	1	0
	ISC71	ISC70	ISC61	ISC60	ISC51	ISC50	ISC41	ISC40
읽기/쓰기	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
초기값	0	0	0	0	0	0	0	0

EICRA 레지스터는 외부 인터럽트 0부터 3까지의 동작 모드를 선택하는 레지스터입니다. 인터럽트 소스 하나당 2개의 비트가 할당되어 있습니다. 비트 설정에 따른 동작 모드는 아래 표에 나와 있습니다.

ISCn1	ISCn0	설명
0	0	INTn에 LOW 입력이 들어왔을 때 인터럽트가 발생
0	1	INTn에 하강 에지 또는 상승 에지가 발생 시 인터럽트 요청
1	0	INTn에 하강 에지(HIGH에서 LOW로 변할 때)가 발생 시 인터럽트 요청
1	1	INTn에 상승 에지(LOW에서 HIGH로 변할 때)가 발생 시 인터럽트 요청

EIMSK : 외부 인터럽트 마스크 레지스터

비트	7	6	5	4	3	2	1	0
	INT7	INT6	INT5	INT4	INT3	INT2	INT1	INT0
읽기/쓰기	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
초기값	0	0	0	0	0	0	0	0

EIMSK 레지스터는 각 외부 인터럽트를 사용할지 여부를 정하는 레지스터입니다. 비트가 1로 설정되면 해당하는 외부 인터럽트를 허용하는 것이며, 0으로 설정되면 허용하지 않는 것입니다. INTn을 1로 설정하고 SREG의 I 비트가 1로 설정되었다면, ISCn1:0 비트에서 설정된 조건과 맞는 외부 인터럽트가 발생했을 때 EIFR의 INTFn 비트가 1이 되고, 인터럽트 서비스 루틴이 실행됩니다.

EIFR : 외부 인터럽트 플래그 레지스터

비트	7	6	5	4	3	2	1	0
	INTF7	INTF6	INTF5	INTF4	INTF3	INTF2	INTF1	INTF0
읽기/쓰기	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
초기값	0	0	0	0	0	0	0	0

EIFR 레지스터는 외부 인터럽트의 개별적인 인터럽트 요청 상태를 나타내는 레지스터입니다. PEn 핀의 상태 변화가 인터럽트 요청을 일으켰을 때에는 해당하는 비트인 INTFn이 1로 설정되며, 이 비트는 인터럽트 서비스 루틴에 진입할 때 하드웨어에 의해서 0으로 초기화 됩니다. 소프트웨어적으로 수동으로 이 비트를 0으로 변하게 하려면, 이 비트에 1을 써주어야 합니다. SREG의 I 비트와 INTn, INTFn이 모두 1이라면, 외부 인터럽트 n이 실행됩니다.

H/W 원리

DRC에는 두 개의 마이크가 좌우로 장착되어 있고, 회로를 통해 외부 인터럽트 핀과 연결됩니다. 일정 크기 이상의 진동이 감지되면 인터럽트 핀에 들어가는 신호가 LOW에서 HIGH로 바뀌며, 진동이 사라지면 다시 LOW로 돌아옵니다. 박수 소리 한번에 LOW->HIGH->LOW로 외부 인터럽트 핀의 입력이 변하므로, 박수 소리가 나는 순간 외부 인터럽트 입력이 LOW에서 HIGH로 바뀌는 상승 에지가 일어나게 됩니다.

DRC에서는 두 개의 외부 인터럽트를 사용하며, 왼쪽 마이크는 INT5, 오른쪽 마이크는 INT6로 입력을 받습니다. 왼쪽에서 소리가 났을 경우 더 가까운 왼쪽 마이크에서 더 빨리 상승 에지가 일어나므로, INT5에서 INT6보다 더 먼저 인터럽트가 발생하게 됩니다. 반대로 오른쪽에서 소리가 나면 INT6에서 INT5보다 더 먼저 인터럽트가 일어납니다.

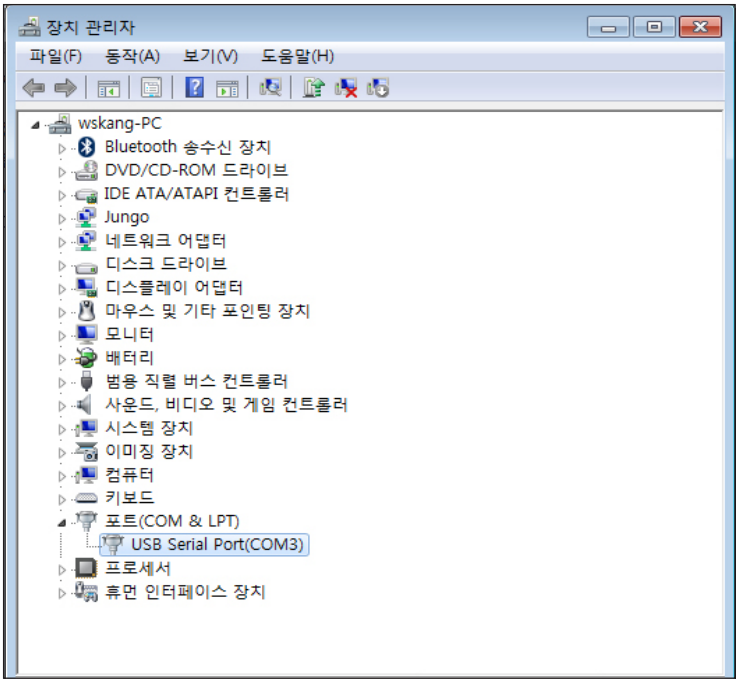
이번 예제에서는 INT5, INT6에서 상승 에지가 발생하는 경우 인터럽트가 발생하도록 설정하고 인터럽트 서비스 루틴을 작성해봅니다.

준비

하드웨어 준비하기

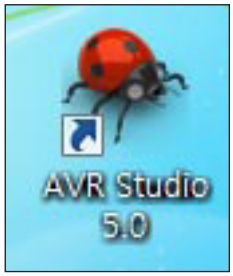
01 연결

DRC 제어기 아답터로 전원을 연결하고, USB to Serial Cable 로 PC 와 DRC 를 연결합니다.



02 COM port 연결

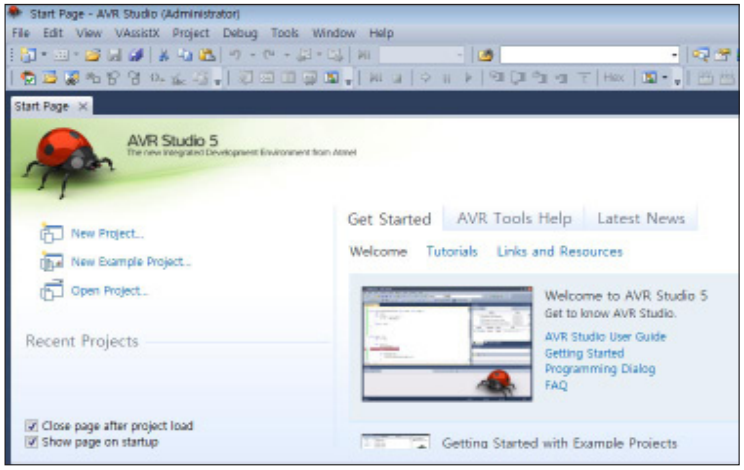
내컴퓨터 〉 속성 〉 장치관리자에서
COM 연결 확인합니다.



AVR Studio 실행 및 코딩 준비

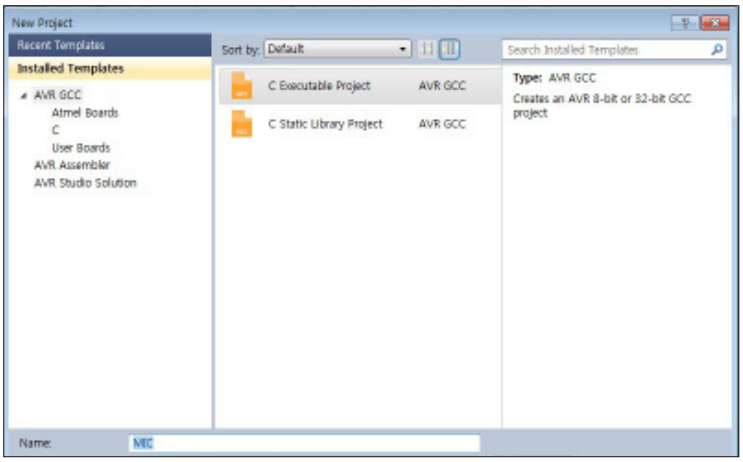
01 실행

AVR Studio 를 실행합니다.



02 New Project

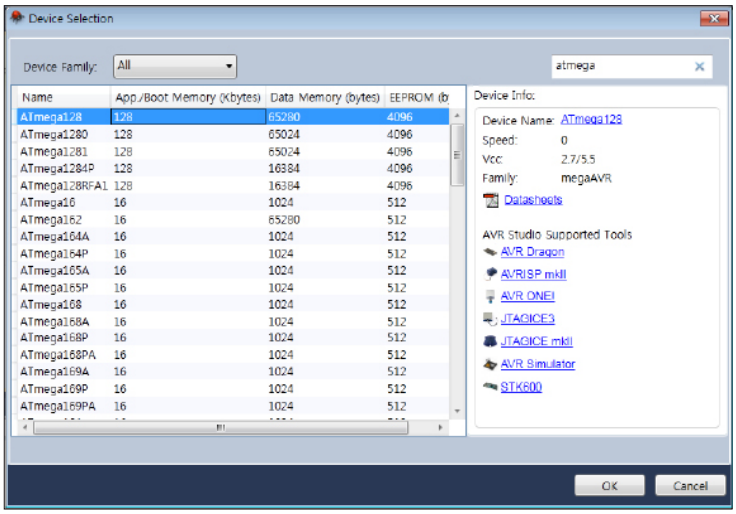
New Project 를 클릭합니다.



03 GCC 선택

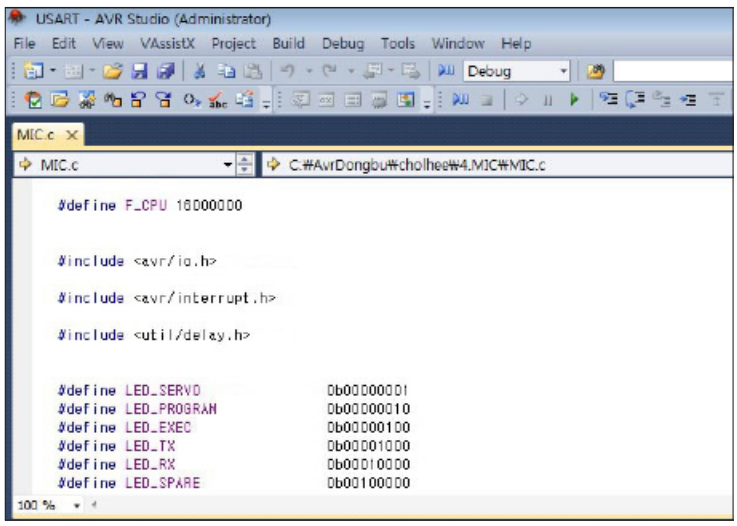
C Executable Project 를 선택합니다.

Name 을 "MIC"로 입력하고, Create directory for solution 을 클릭하고, OK 버튼을 누릅니다.



04 ATmega 선택

우측 상단에 atmega 를 검색하여, 리스트중에 ATmega128 을 선택합니다.



05 Editor

편집창에 코딩을 시작합니다.

아래 예제를 따라하세요.

예제

DRC의 좌측 마이크에서 소리를 인식하면 Servo LED를 0.5초간 켰다가 끄고, 우측 마이크에서 소리를 인식하면 Power LED를 0.5초간 켰다가 끄는 예제입니다.

MIC.c

```
//CPU 클럭 설정
#define F_CPU 16000000

//ATmega128의 레지스터 등이 정의되어 있음
#include <avr/io.h>
//_delay_ms 등 딜레이 함수가 정의되어 있음
#include <util/delay.h>

int main(void)
{
	//8비트 변수 선언
	unsigned char i;

	//포트 C의 입출력 방향 설정
	DDRC = 0b01111111;
	//포트 C의 출력 값 설정
	PORTC = 0b01111111;

	//while 안의 내용을 무한 반복
	while(1)
	{
		//LED가 차례로 하나씩 더 켜지도록 동작을 반복함
		for(i=0b00000001;i<=0b01111111;i=(i<<1)+1){
			PORTC = ~i;		//Active Low이므로 i 변수의 값을 반전하여 출력
			_delay_ms(200);		//200ms동안 대기
		}
	}

	//함수의 형태에 맞게 정수값 반환
	return 1;
```

```
int main(void)
{
        //포트 C의 입출력 방향 설정
        DDRC = 0b01111111;
        //포트 C의 출력 값 설정
        PORTC = 0b01111111;

        //전체 인터럽트를 비활성화
        cli();

        //INT5, INT6을 상승 에지 모드로 설정
        EICRB |= 0b00111100;
        //INT5, INT6을 사용하도록 설정
        EIMSK |= 0b01100000;
        //INT5, INT6의 인터럽트 플래그를 0으로 초기화
        EIFR |= 0b01100000;

        //전체 인터럽트를 활성화
        sei();

        //while 안의 내용을 무한 반복
        while(1)
        {

        }

        //함수의 형태에 맞게 정수값 반환
        return 1;
}
```

예제설명

```
#define F_CPU 16000000
```

#define 문은 코드 전처리 명령 중 하나로, 앞으로 F_CPU라는 값이 나오면 16000000으로 대체한다는 뜻입니다. DRC의 ATmega128은 16MHz의 클럭 속도로 동작합니다. 클럭 속도에 비례해서 명령 처리 속도가 달라지므로, 이 명령을 통해서 이 프로그램이 동작할 속도를 명시해 주어야 delay 함수 등이 올바르게 동작합니다.

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

avr/io.h 헤더 파일에는 DDRX, PORTX 레지스터 등 AVR 제어를 위해 필요한 레지스터 등이 정의되어 있고, avr/interrupt.h 헤더 파일에는 인터럽트를 사용하기 위한 함수들이 정의되어 있습니다. util/delay.h 헤더 파일에는 _delay_ms()등 자주 사용하는 딜레이 함수가 정의되어 있으므로 #include 문을 사용하여 포함시켜 줍니다.

```
#define LED_SERVO                    0b00000001
…
#define LED_MASK                     0b01111111
```

#define LED_SERVO 0b00000001은 앞으로 LED_SERVO라는 것이 나오면 0b00000001로 대체해서 컴파일한다는 뜻입니다. DRC의 각 LED에 해당하는 비트를 LED의 이름으로 대체해서 프로그램을 작성하기 쉽게 하려고 선언합니다.

```
ISR(INT5_vect){
        PORTC &= ~LED_SERVO;
        _delay_ms(500);
        PORTC |= LED_SERVO;
}
ISR(INT6_vect){
        PORTC &= ~LED_POWER;
        _delay_ms(500);
        PORTC |= LED_POWER;
```

외부 인터럽트 5번과 6번의 인터럽트 서비스 루틴입니다. 인터럽트가 발생해서 EIFR 레지스터의 해당 비트가 1이 되면, 인터럽트 요청에 따라서 실행하던 프로그램을 멈추고 인터럽트 서비스 루틴으로 진입합니다. 진입하는 순간 EIFR 레지스터의 비트는 다시 0으로 초기화됩니다. 왼쪽 마이크에 소리가 인식되어서 INT5에 상승 에지가 발생하면, INT5의 인터럽트 서비스 루틴에서 PORTC의 Servo LED에 해당하는 비트를 0으로 만들어 Servo LED를 켜고, 500ms 동안 대기한 후, 다시 그 비트를 1로 만들어서 Servo LED를 끕니다. 마찬가지로 오른쪽 마이크에 소리가 들어와 INT6에 상승 에지가 발생하면, INT6의 인터럽트 서비스 루틴에서 Power LED를 500ms 동안 켰다가 끕니다.

```
DDRC = 0b01111111;
PORTC = 0b01111111;
```

포트 C의 0~6번 핀을 출력으로 설정하고, 출력 값을 1로 만듭니다. 그러면 모든 LED가 꺼지게 됩니다.

```
cli();

EICRB |= 0b00111100;
EIMSK |= 0b01100000;
EIFR |= 0b01100000;

sei();
```

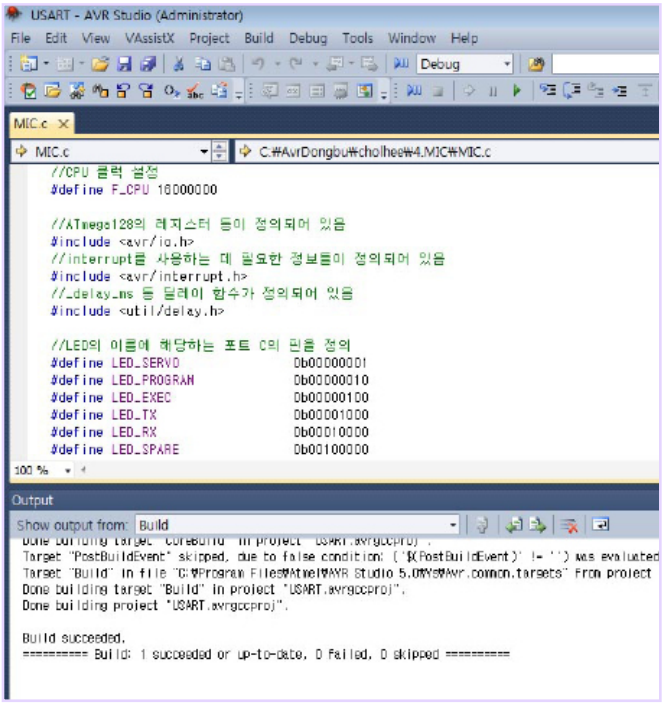
cli()와 sei()는 각각 전체 인터럽트를 비활성화, 활성화 시키는 명령어입니다. ATmega128에는 SREG라는 AVR 상태 레지스터가 있는데, 이 중 7번 비트인 I 비트는 인터럽트 요청이 있을 때 인터럽트 서비스 루틴으로 점프하도록 허용하는 비트입니다. 레지스터 값을 바꾸어서 외부 인터럽트를 사용하겠다고 설정을 해놓았더라도, 이 I 비트가 1로 설정되어야 실제로 인터럽트를 사용할 수 있습니다. cli() 명령은 I 비트를 0으로, sei() 명령은 I 비트를 1로 설정합니다.
인터럽트에 대한 설정을 바꾸는 중에 인터럽트가 발생해 예상치 못한 오동작이 발생하는 것을 방지하기 위해서 우선 cli() 명령을 통해서 전체 인터럽트를 비활성화 시킵니다.
그 후 외부 인터럽트에 관련된 레지스터에 알맞은 값을 넣어 우리가 사용하고자 하는 용도에 맞게 외부 인터럽트를 설정해 줍니다. 우선, 외부 인터럽트 5, 6번을 상승 에지 모드로 사용할 것이므로 EICRB의 ISC61, ISC60, ISC51, ISC50은 모두 1이 되어야 합니다. 그래서 EICRB를 0b00111100과 |(bitwise OR)연산 해주어 해당 비트를 1로 만들어 줍니다. 또한 EIMSK의 5, 6번 비트를 1로 만들어서 외부 인터럽트를 사용할 것임을 알려야 합니다. 그래서 EIMSK를 0b01100000과 | 연산 해주어 5, 6번 비트를 1로 만듭니다. 마지막으로 EIFR의 5, 6번 비트를 0으로 초기화 합니다. 앞서 설명한대로, EIFR의 비트를 강제로 0으로 만들기 위해서는 그 비트에 1을 써야 합니다. EIFR의 5, 6번 비트에 1을 써서 초기화 해줍니다.
외부 인터럽트에 대한 설정이 끝난 후에는 sei() 명령을 통해서 전체 인터럽트를 활성화 합니다.

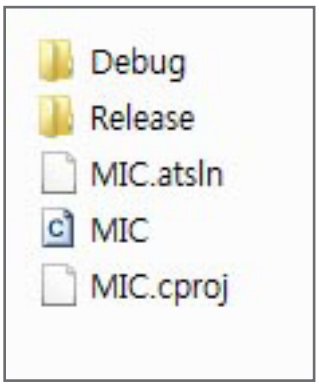
```
while(1)
{

}
```

이 프로그램에는 while문 안에 아무것도 들어 있지 않습니다. 따라서 인터럽트가 발생하지 않는 상황에서는 프로그램은 계속 while문을 반복할 뿐 아무런 일도 하지 않습니다. 그러나 인터럽트가 발생하면 프로그램은 while문을 도는 것을 잠시 멈추고 인터럽트 서비스 루틴으로 점프했다가, 루틴이 끝나면 다시 원래 있던 자리로 돌아와 계속 while문을 반복하게 됩니다

실행

다운로드하고 실행 시 소리가 나면 Servo LED나 Power LED가 0.5초간 꺼졌다가 꺼집니다. 왼쪽에서 박수를 치면 Servo LED가 먼저 점멸되고 Power LED가 나중에 점멸됩니다. 반대로 오른쪽에서 박수를치면 Power LED가 켜진 후에 Servo LED가 켜집니다. 이것은 소리가 발생한 방향에 따라서 양쪽 마이크에 소리가 도달하는 시간이 차이가 나고, 그래서 인터럽트가 발생하는 타이밍이 미세하게 차이가 나기 때문입니다. INT5가 먼저 발생해서 INT5의 인터럽트 서비스 루틴에 진입한 후 INT6이 발생하면, EIFR의 6번 비트는 INT5의 인터럽트 서비스 루틴이 끝날 때까지 1이 된 채로 유지되다가, INT6의 인터럽트 서비스 루틴으로 진입하면서 비로소 0으로 초기화 됩니다.





01 Build

Build 메뉴를 선택하여 Build Solution을 클릭하여 Build 시킵니다.

1 succeeded 0 failed 가 나오면 에러없이 컴파일 완료됐다는 것을 의미합니다.

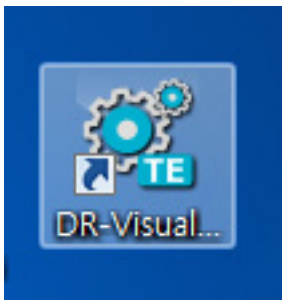
02 Debug 폴더

Build 후에 소스가 있는 폴더에 들어가면 Debug 폴더가 생성 되어있습니다. 클릭하여 폴더 안의 내용을 봅니다.

03 hex 확인

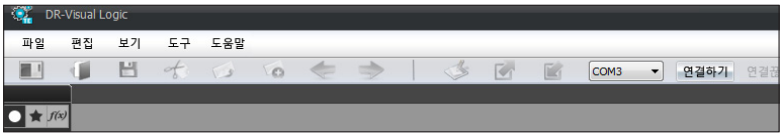
Debug 폴더 안의 hex 파일 확인합니다. Hex 파일이 제어기에 포팅시키는 파일입니다.

DR-Visual Logic 에서 펌웨어 복구를 실시합니다.



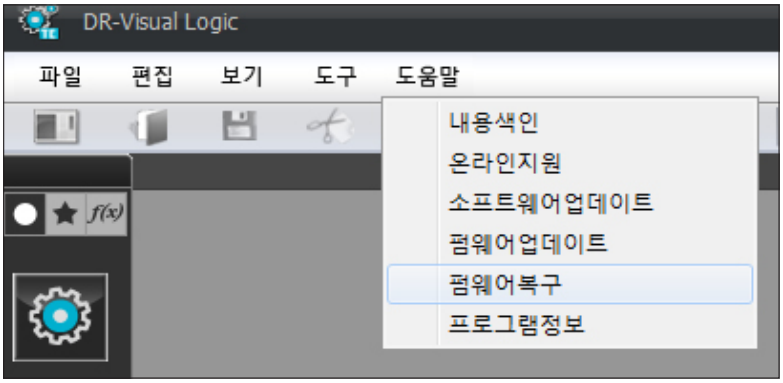
04 Visual Logic

DR-Visual Logic 을 실행시킵니다.



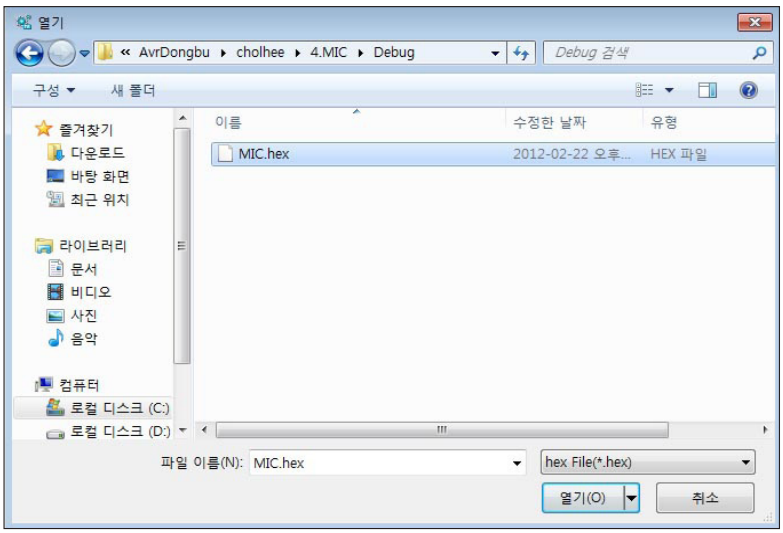
05 연결하기

COM 번호를 설정하고 연결하기를 클릭합니다.



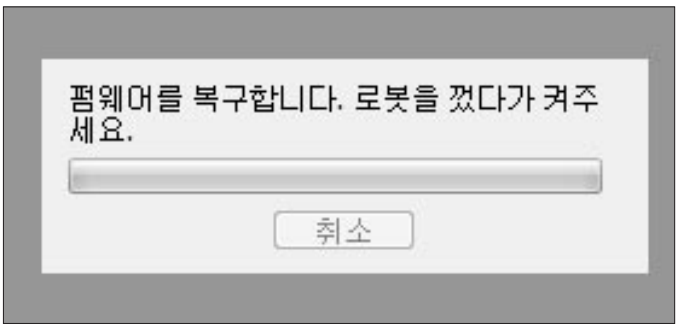
06 펌웨어 복구

도움말 메뉴에서 펌웨어복구를 클릭합니다.



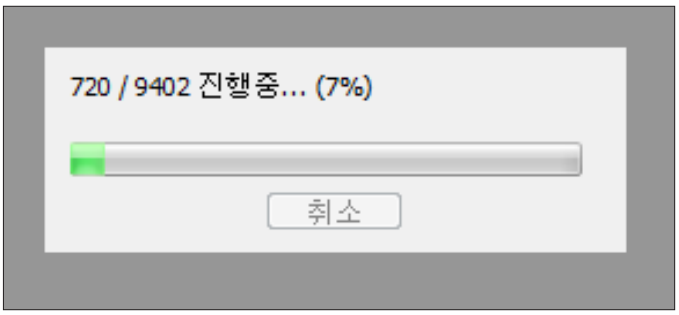
07 불러오기

Debug 폴더에서Hex 파일을 불러옵니다



08 펌웨어복구

로봇을 껐다가 켜주세요 라는 문구가 나옵니다.
제어기의 전원버튼을 한번 껐다가 다시 켜주세요.



09 복구시작

펌웨어 복구가 시작됩니다.

10 실행

왼쪽에서 박수를 치면 Servo LED가 먼저 점멸되고 Power LED가 나중에 점멸됩니다. 반대로 오른쪽에서 박수를치면 Power LED가 켜진 후에 Servo LED가 켜집니다.

CH.5 : 타이머/카운터로 딜레이 기능 사용하기

ATmega128의 타이머/카운터를 사용하여 딜레이를 구현하고, Chaper 1의 LED 예제에 활용합니다.

AVR 원리

ATmega128에는 8비트 타이머/카운터인 타이머/카운터0과 타이머/카운터2, 16비트 타이머/카운터인 타이머/카운터1과 타이머/카운터3으로 총 4개의 타이머/카운터가 있습니다. 타이머/카운터는 특정 신호사이의 시간을 재거나, 신호가 나온 개수를 세거나, PWM 파형를 만드는 등 여러 용도로 사용됩니다. 타이머/카운터의 여러 기능을 정리하면 아래와 같습니다.

– 일반적인 타이머 동작

타이머/카운터에 인가되는 클럭을 받아서 스스로 일정 시간마다 카운트 레지스터의 값을 증가시키며, 오버플로우가 발생하면 오버플로우 인터럽트 플래그를 1로 만들고 카운트 레지스터를 0으로 초기화 시킵니다. 오버플로우가 발생할 때마다 카운트 레지스터를 0이 아닌 일정한 특정 값으로 초기화 시키면, 원하는 주기로 계속 오버플로우를 인터럽트를 발생시킬 수 있습니다.

– PWM 파형 발생 동작

PWM 파형을 발생하는 모드에서는 일정한 주기로 오버플로우가 발생합니다. 여기서 초기값인 0과 최대값 사이에 비교값을 설정해서 0에서부터 비교값까지의 시간과 비교값부터 오버플로우까지의 시간을 HIGH, LOW나 LOW, HIGH로 출력하는 PWM 파형을 만들 수 있습니다. 비교값의 값을 변경함으로서 듀티 사이클을 조절하며 PWM 제어를 할 수 있는 것입니다.

※ PWM이란?

PWM은 Pulse Width Modulation의 약자로 펄스 폭 변조라고도 합니다. 모터 제어나 LED 제어를 아날로그 전압으로 제어할 때에는 1V, 2V, 3V 등 원하는 전압을 인가하여 모터의 속도나 LED의 밝기를 제어할 수 있습니다. 그런데 디지털 신호로 제어할 때에는 HIGH와 LOW밖에 가질 수 있는 값이 없으므로 가능한 세기가 제한됩니다. 이런 경우 사용하는 것이 PWM입니다. 일정한 빠른 주기로 디지털 펄스를 발생시키되, 펄스가 HIGH로 유지되는 폭을 조절함으로써 마치 아날로그 전압을 가하는 것과 같은 효과를 내는 것입니다. 가령 HIGH가 5V이고 LOW가 0V인 디지털 출력 핀에서 HIGH와 LOW의 길이가 1:1인 펄스 신호를 일정한 빠른 주기로 내보낸다면, 마치 2.5V의 전압이 가해진 것과 같은 효과를 내게 합니다. 다른 예로 길이가 4:1이라면 4V 전압이 가해진 것과 같은 효과가 납니다. 여기서 HIGH의 길이와 전체 주기의 비율을 듀티 사이클(Duty Cycle)이라고 합니다. HIGH:LOW의 시간 비율이 1:1인 경우 듀티 사이클은 50%, 4:1인 경우 80%가 될 것입니다.

– 토글 동작

카운트 레지스터 값과 비교값이 일치하면 출력 핀의 상태를 토글(HIGH이면 LOW로, LOW이면 HIGH로)하도록 설정할 수 있습니다. 비교값을 변경하는 방법으로 출력 핀에서 나오는 디지털 파형의 주파수를 조절할 수 있습니다.

– 펄스폭 측정 동작

입력 캡쳐 기능을 사용해서 특정 핀에 들어오는 신호의 변화를 감지해서 그 변화가 일어난 시간을 캡쳐할 수 있습니다. 결과값을 이용해서 들어온 신호의 폭을 비교하거나 주기를 측정하는 데 이용할 수 있습니다.

– 카운터 동작

외부에서 들어오는 신호 변화를 클럭으로 입력 받아서, 신호 변화가 일어날 때마다 카운트 레지스터가 증가하게 함으로써 특정 신호 변화가 몇 번 일어났는지 셀 수 있습니다.

이번 예제에서는 타이머/카운터0을 사용하여 일반적인 타이머 동작으로 설정해보고, 그에 따라서 Chapter 1의 LED 예제에서 사용했던 _delay_ms를 타이머/카운터0의 기능을 응용해 대체해봅니다.

타이머/카운터0에 관련된 대표적인 레지스터는 아래와 같습니다.

TCCR0 : 타이머/카운터0 제어 레지스터

비트	7	6	5	4	3	2	1	0
	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00
읽기/쓰기	W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
초기값	0	0	0	0	0	0	0	0

– 비트 2:0번 CS02:0 : 클럭 선택

타이머/카운터0이 동작할 클럭을 선택합니다. ATmega128에 들어오는 기본 클럭 주파수를 어떤 비율로 나누어서 TCNT0에서 사용할지가 정해집니다. DRC의 기본 클럭 주파수는 16MHz입니다. 따라서 CS02:0을 0b101으로 설정했다면, 타이머/카운터0의 동작 주파수는 16MHz/128 = 125kHz가 됩니다. 8us마다 TCNT0이 1씩 증가하는 것입니다.

CS02	CS01	CS00	설명
0	0	0	동작 안 함
0	0	1	클럭/1
0	1	0	클럭/8
0	1	1	클럭/32
1	0	0	클럭/64
1	0	1	클럭/128
1	1	0	클럭/256
1	1	1	클럭/1024

– 비트 6번, 3번 WGM01:0 : 파형 생성 모드

타이머/카운터0이 동작할 모드를 결정합니다. 각 모드에 따라서 카운트 레지스터가 증가하는 방향, 카운터의 최대값, 발생하는 출력 파형 종류가 달라집니다. 타이머/카운터0은 네 가지 동작 모드가 있으며 일반 모드, CTC 모드, 그리고 2가지 종류의 PWM 모드가 있습니다. 각각의 모드에 대한 자세한 설명은 레지스터에 대한 설명 다음부분을 참고하면 됩니다.

WGM01	WGM01	동작 모드	최대값	OCR0 레지스터의 업데이트 시점	TOV0 플래그가 1이 되는 시점
0	0	일반	255(0xFF)	설정하는 즉시	TCNT0=255
0	1	Phase Correct PWM	255(0xFF)	TCNT0=최대값	TCNT0=0
1	0	CTC	OCR0	설정하는 즉시	TCNT0=255
1	1	고속 PWM	255(0xFF)	TCNT0=0	TCNT0=255

– 비트 5:4번 COM01:0 : 비교 매치 출력 모드

이 2개 비트는 OC0 단자의 동작을 설정합니다. OC0은 포트 B의 4번 핀(PB4)에 있습니다. PB4는 COM01:0 중 하나라도 1인 경우에는 일반 입출력 포트의 기능이 아닌 타이머/카운터0의 비교 출력 핀의 기능으로 사용됩니다. 동작 모드마다 COM01:0의 설정 값에 따른 OC0 핀의 기능이 다릅니다. 표에서 말하는 비교 매치란, 앞으로 설명할 OCR0(출력 비교 레지스터)과 TCNT0(타이머/카운터0 레지스터)가 서로 값이 같아진 순간을 말합니다. COM01:0을 OC0핀을 사용하는 것으로 설정하더라도, 실제로 DDRB의 4번 비트를 출력 핀으로 설정해야만 OC0 출력이 나옵니다.

■ PWM 모드가 아닌 경우(일반, CTC)의 비교 출력 모드

COM01	COM00	OC0 핀의 기능
0	0	일반 I/O 포트로 동작
0	1	비교 매치에서 OC0 출력을 토글
1	0	비교 매치에서 OC0 출력을 LOW로 설정
1	1	비교 매치에서 OC0 출력을 HIGH로 설정

■ 고속 PWM 모드의 비교 출력 모드

COM01	COM00	OC0 핀의 기능
0	0	일반 I/O 포트로 동작
0	1	사용하지 않음
1	0	비교 매치에서 OC0 출력을 LOW로 설정하고 TCNT0=0일 때 OC0 출력을 HIGH로 함
1	1	비교 매치에서 OC0 출력을 HIGH로 설정하고 TCNT0=0일 때 OC0 출력을 LOW로 함

Phase Correct PWM 모드의 비교 출력 모드

COM01	COM00	OC0 핀의 기능
0	0	일반 I/O 포트로 동작
0	1	사용하지 않음
1	0	업 카운트의 비교 매치에서 OC0 출력을 LOW로 설정하고 다운 카운트의 비교 매치에서 OC0 출력을 HIGH로 함
1	1	업 카운트의 비교 매치에서 OC0 출력을 HIGH로 설정하고 다운 카운트의 비교 매치에서 OC0 출력을 LOW로 함

– 비트 7번 FOC0 : 강제 출력 비교

FOC0은 PWM 모드가 아닐 때만 동작합니다. 이 비트에 1을 쓰면, 타이머 값에 상관 없이 강제로 비교 매치를 일으켜서, COM01:0의 설정에 따라서 OC0 핀의 출력 값을 변경합니다. 이 비트에서 강제로 일으킨 비교 매치는 인터럽트를 발생시키지 않으며, CTC 모드에서 타이머를 초기화 시키지도 않습니다. 또한 이 비트는 쓰기만 가능하며, 항상 0으로 읽힙니다.

TCNT0 : 타이머/카운터0 레지스터

비트	7	6	5	4	3	2	1	0
	TCNT0[7:0]							
읽기/쓰기	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
초기값	0	0	0	0	0	0	0	0

TCNT0은 타이머/카운터0의 현재 카운트 값을 저장하는 레지스터입니다. CS02:0 비트에 따라서 결정된 타이머/카운터0의 주기마다 1씩 증가합니다. 이 레지스터는 직접 읽고 쓸 수 있습니다만, TCNT0의 값을 쓴 직후 클럭에서는 비교 매치가 일어나지 않으므로 필요한 비교 매치를 놓칠 가능성도 있으니 주의해서 사용해야 합니다.

OCR0 : 타이머/카운터0 출력 비교 레지스터

비트	7	6	5	4	3	2	1	0
	OCR0[7:0]							
읽기/쓰기	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
초기값	0	0	0	0	0	0	0	0

OCR0 레지스터 안에 들어 있는 값은 TCNT0의 값이 변할 때마다 TCNT0과 비교됩니다. OCR0과 TCNT0의 값이 같아지면 비교 매치가 일어나며, 동작 모드와 COM02:0 비트들의 값에 따라서 OC0 단자의 출력이 달라집니다. OCR0의 값을 바꾸더라도 동작 모드에 따라서는 그 즉시 반영되는 것이 아니라 TCNT0이 특정 값에 도달했을 때 반영되는 경우도 있습니다. 자세한 내용은 WGM01:0 비트의 표를 참조하시기 바랍니다.

■ TIMSK : 타이머/카운터0 인터럽트 마스크 레지스터

비트	7	6	5	4	3	2	1	0
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIEB	TOIE1	OCIE0	TOIE0
읽기/쓰기	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
초기값	0	0	0	0	0	0	0	0

– 비트 1번 OCIE0 : 타이머/카운터0 출력 비교 매치 인터럽트 활성화

이 비트가 1로 설정되면 타이머/카운터0의 비교 매치 인터럽트가 활성화됩니다. 이 비트가 1로 설정되고 SREG 레지스터의 I 비트가 1로 설정된 상태에서 비교 매치가 일어나면 TIFR 레지스터의 OCF0 비트가 1이 되고, 해당하는 인터럽트 서비스 루틴이 실행됩니다.

– 비트 0번 TOIE0 : 타이머/카운터0 오버플로우 인터럽트 활성화

이 비트가 1로 설정되면 타이머/카운터0의 오버플로우 인터럽트가 활성화됩니다. 이 비트가 1로 설정되고 SREG 레지스터의 I 비트가 1로 설정된 상태에서 오버플로우가 일어나면 TIFR 레지스터의 TOV0 비트가 1이 되고, 해당하는 인터럽트 서비스 루틴이 실행됩니다.

■ TIFR : 타이머/카운터0 인터럽트 플래그 레지스터

비트	7	6	5	4	3	2	1	0
	OCF2	TOV2	ICF1	OCF1A	OCFB	TOV1	OCE0	TOV0
읽기/쓰기	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
초기값	0	0	0	0	0	0	0	0

– 비트 1번 OCF0 : 타이머/카운터0 출력 비교 플래그

이 비트는 TCNT0과 OCR0 사이에서 비교 매치가 일어나면 1로 설정됩니다. 이 비트는 해당하는 인터럽트 서비스 루틴으로 진입할 때 하드웨어에 의해서 0으로 초기화됩니다. 소프트웨어적으로 수동으로 이 비트를 0으로 변하게 하려면, 이 비트에 1을 써주어야 합니다. SREG의 I 비트와 OCIE0, OCF0이 모두 1이라면, 타이머/카운터0 비교 매치 인터럽트가 실행됩니다.

– 비트 0번 TOV0 : 타이머/카운터0 오버플로우 플래그

이 비트는 타이머/카운터0에서 오버플로우가 일어나면 1로 설정되고, 해당하는 인터럽트 서비스 루틴으로 진입할 때 하드웨어에 의해서 0으로 초기화됩니다. 소프트웨어적으로 수동으로 이 비트를 0으로 변하게 하려면, 이 비트에 1을 써주어야 합니다. SREG의 I 비트와 TOIE0, TOV0이 모두 1이라면, 타이머/카운터0 오버플로우 인터럽트가 실행됩니다. Phase Correct PWM 모드에서는 다른 모드와 다르게, TCNT0의 증감 방향이 다운 카운트에서 업 카운트로 바뀔 때(TCNT0=0일 때) TOV0 비트가 1로 설정됩니다.

이 외에도 ASSR, SFIOR 등의 레지스터가 타이머/카운터0의 동작과 관련되어 있습니다. 더 자세한 사항은 ATmega128의 데이터시트나 다른 서적을 참조하기 바랍니다.

앞서 TCCR0의 설명에서도 나왔듯이 타이머/카운터0의 동작 모드는 4가지가 있습니다.

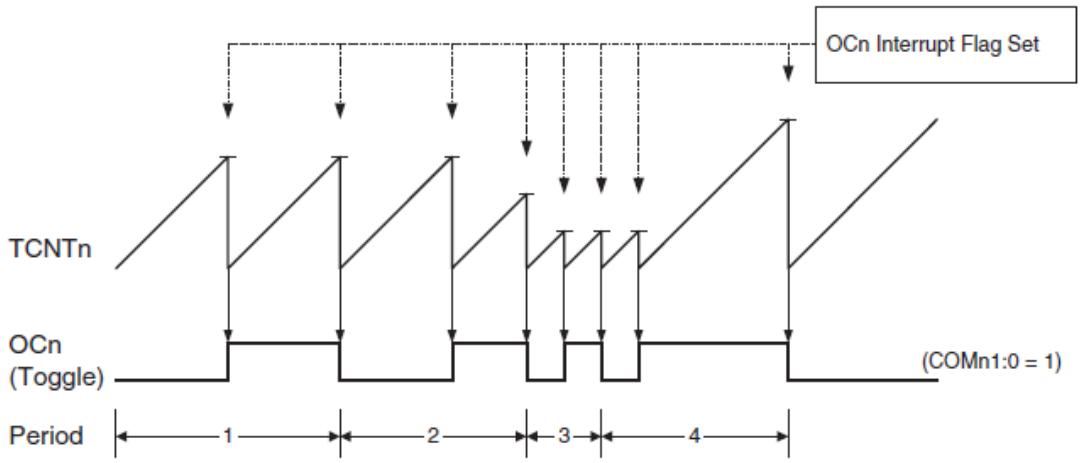
1. 일반 모드(WGM01:0 = 0b00)

가장 간단한 동작모드로 TCNT0이 항상 업 카운터(증가하는 카운터)로만 동작하고 최대값인 255에서 최소값인 0으로 변하는 순간에 TOV0이 1이 됩니다. SREG의 I 비트와 TOIE0이 모두 1이라면 타이머/카운터0 오버플로우 인터럽트 서비스 루틴이 실행되고 TOV0은 다시 0으로 초기화됩니다. 일반 모드는 대개 타이머/카운터0 오버플로우 인터럽트를 사용하거나 외부 클럭 신호를 세는 데 사용하며, 비교 매치 인터럽트를 사용할 수는 있으나 CPU 시간 점유를 이유로 권장하지 않습니다.

2. CTC모드(WGM01:0 = 0b10)

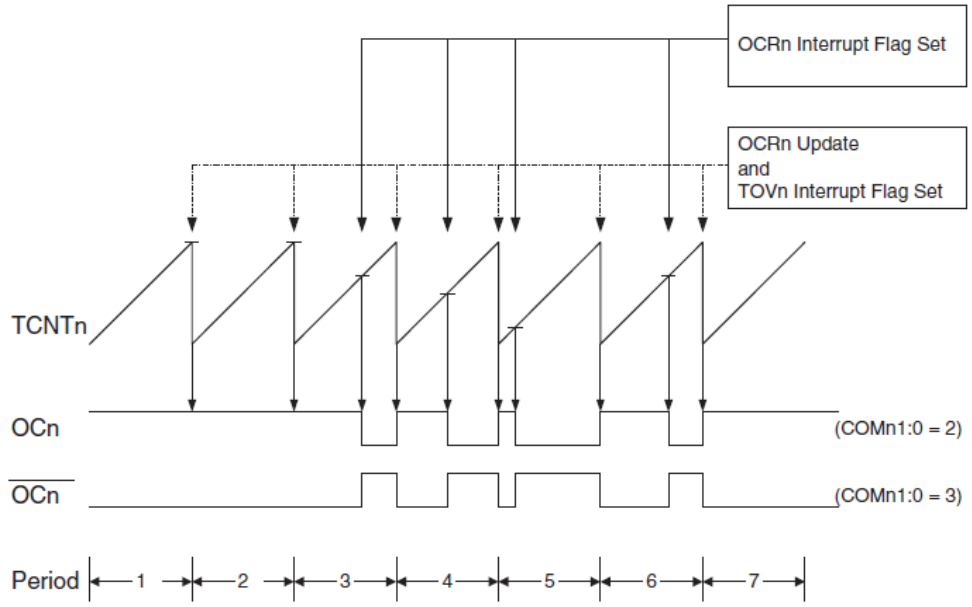
CTC모드는 Clear Timer on Compare 모드의 약자로, OCR0 레지스터가 타이머 초기화의 주기를 결정하는 데 사용됩니다. CTC 모드에서는 TCNT0 레지스터가 OCR0과 같아지는 순간, 즉 비교 매치가 일어나는 순간 0으로 초기화 됩니다. 즉 OCR0 레지스터의 값이 카운터의 최대값을 결정합니다. 비교 매치가 일어나는 순간 OCF0 비트는 1로 바뀌며, 타이머/카운터0 오버플로우 인터럽트 서비스 루틴이 실행되면 0으로 초기화 됩니다.

COM01:0 값을 0b01로 설정하면 비교 매치가 일어날 때마다 OC0 핀이 토글되도록 설정할 수 있습니다. 이를 이용하면 OC0 핀에 주기적인 펄스를 만들 수 있으며, OCR0의 값을 바꿈에 따라서 펄스의 주파수를 바꿀 수 있습니다.



3. 고속 PWM 모드(WGM01:0 = 0b11)

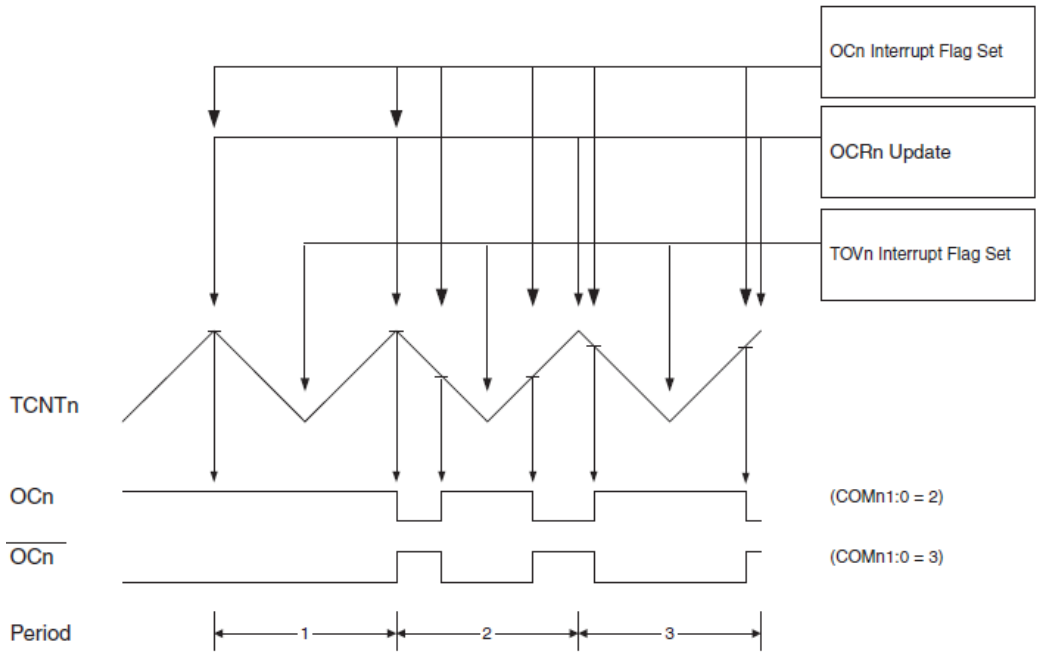
고속 PWM 모드는 TCNT0이 0에서 255까지 증가하고 다시 0으로 변하는 톱니 모양으로 변하며, OCR0의 값에 따라서 듀티 사이클이 변하는 PWM 출력을 만들어내는 모드입니다. COM01:0을 0b10으로 설정한 경우, OC0 핀은 비교 매치가 일어난 때에 LOW로 변하고, TCNT0이 0일 때 HIGH로 변합니다. 따라서 OC0은 0~OCR0 까지는 HIGH, OCR0~255까지는 LOW인 파형이 나오는 비반전 출력 비교 모드로 작동합니다. COM01:0을 0b11로 설정한 경우에는 반대로, 비교 매치가 일어난 때에 HIGH, TCNT0이 0일 때 LOW로 변합니다. 따라서 0~OCR0 까지는 LOW, OCR0~255까지는 HIGH인 파형이 나오는 반전 출력 비교 모드로 작동합니다.



4. Phase Correct PWM 모드(WGM01:0=0b01)

Phase Correct PWM 모드는 고속 PWM 모드와 유사하나, TCNT0이 항상 증가하는 방향으로 작동하는 것이 아니라 0에서 255까지 증가했다가 다시 255에서 0으로 감소하는 것을 반복하는 모드입니다. 고속 PWM 모드처럼 0에서 255까지가 한 주기가 아니고, 0에서 255까지, 그리고 255에서 0까지가 한 주기이므로 고속 PWM 모드에 비해서 주기는 2배이지만 TCNT0=0을 기준으로 대칭인 특성 때문에 모터 제어에 많이 사용됩니다.

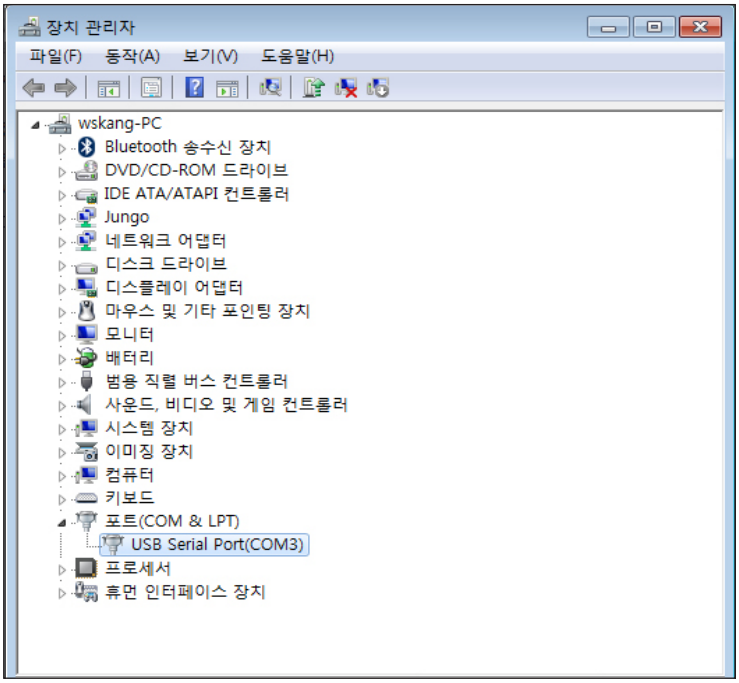
COM01:0을 0b10으로 설정한 경우, OC0 핀은 업 카운트에서 비교 매치가 일어난 때에 LOW로 변하고, 다운 카운트에서 비교 매치가 일어난 때에 HIGH로 변합니다(비반전 출력 비교 모드). COM01:0을 0b11로 설정한 경우에는 반대로, 업 카운트에서 비교 매치가 일어난 때에 HIGH, 다운 카운트에서 비교 매치가 일어난 때에 LOW로 변합니다 (반전 출력 비교 모드).



H/W 원리

이번 예제는 ATmega128에 내장된 타이머/카운터0을 이용해 LED를 제어합니다. LED의 하드웨어에 관련된 내용은 Chapter 1을 참조하면 됩니다.

준비



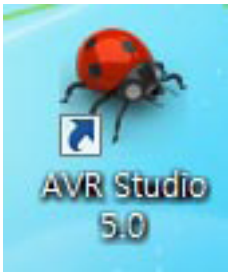
하드웨어 준비하기

01 연결

DRC 제어기 아답터로 전원을 연결하고, USB to Serial Cable 로 PC 와 DRC 를 연결합니다.

02 COM port 연결

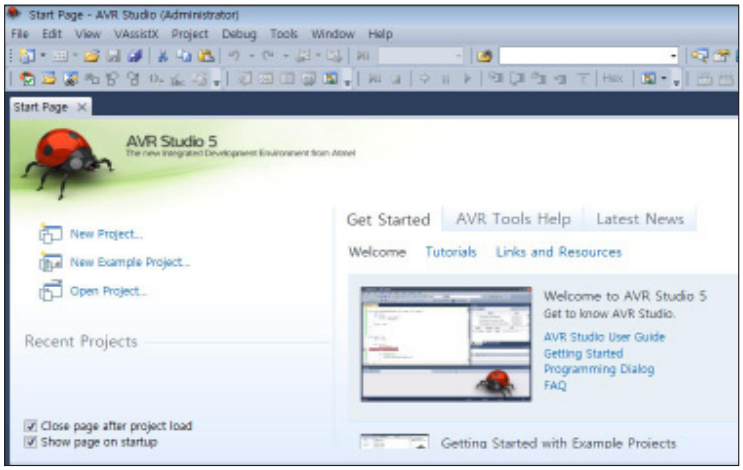
내컴퓨터 〉 속성 〉 장치관리자에서 COM 연결 확인합니다.



AVR Studio 실행 및 코딩 준비

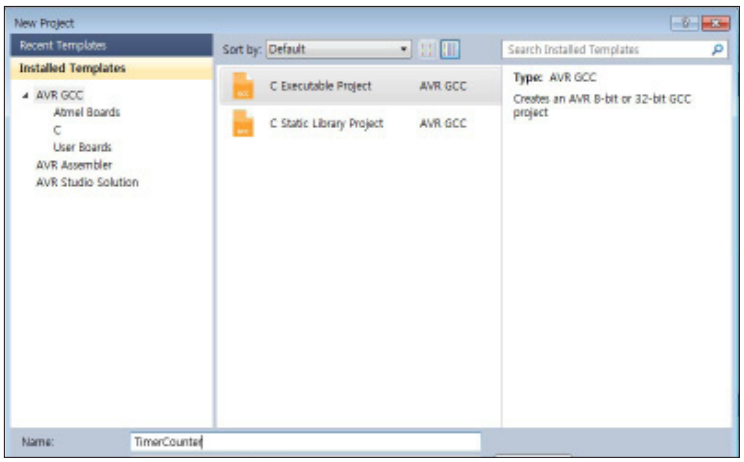
01 실행

AVR Studio 를 실행합니다.



02 New Project

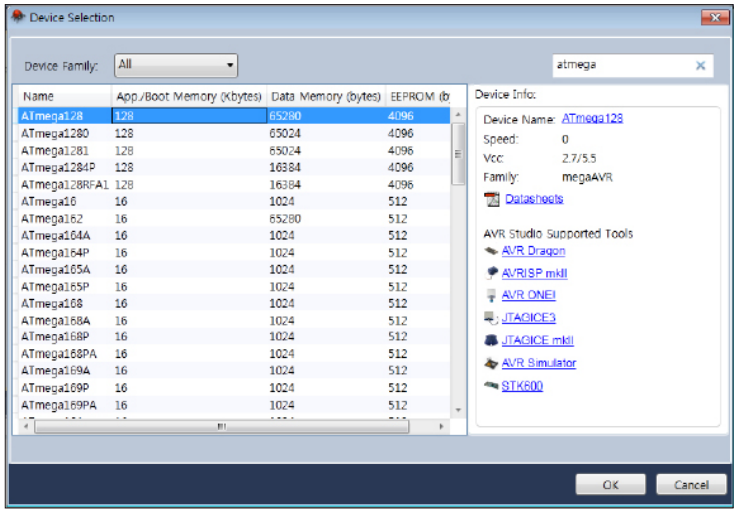
New Project 를 클릭합니다.



03 GCC 선택

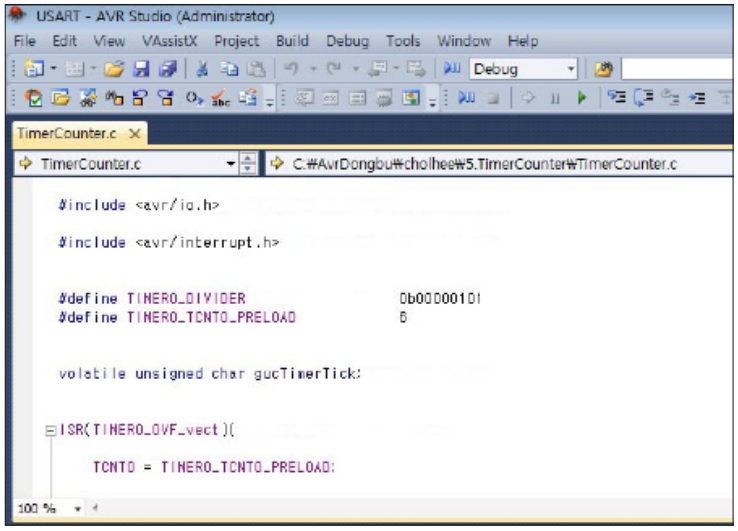
C Executable Project 를 선택합니다.

Name 을 “TimerCounter”로 입력하고, Create directory for solution 을 클릭하고, OK 버튼을 누릅니다.



04 ATmega 선택

우측 상단에 atmega 를 검색하여, 리스트중에 ATmega128 을 선택합니다.



05 Editor

편집창에 코딩을 시작합니다.

아래 예제를 따라하세요.

예제

DRC의 LED들이 Servo(0번핀)부터 Power(6번핀)까지 차례로 켜지는 것을 반복하는 예제입니다. 단, 딜레이 기능을 타이머/카운터0으로 구현합니다.

TimerCounter.c

```
//ATmega128의 레지스터 등이 정의되어 있음
#include <avr/io.h>
//interrupt를 사용하는 데 필요한 정보들이 정의되어 있음
#include <avr/interrupt.h>

//타이머/카운터0의 클럭 속도와 TCNT0의 초기값
#define TIMER0_DIVIDER                  0b00000101
#define TIMER0_TCNT0_PRELOAD            6

//타이머/카운터0에서 시간을 잴 때 사용할 전역변수 선언
volatile unsigned char gucTimerTick;

//타이머/카운터0 오버플로우 인터럽트 서비스 루틴
ISR(TIMER0_OVF_vect){
        //TCNT0을 초기 값으로 재설정
        TCNT0 = TIMER0_TCNT0_PRELOAD;

        //전역변수가 0이 아니면 1 감소
        if(gucTimerTick != 0){
                gucTimerTick--;
        }
}

int main(void)
{
        //8비트 변수 선언
        unsigned char i;

        //포트 C의 입출력 방향 설정
        DDRC = 0b01111111;
        //포트 C의 출력 값 설정
        PORTC = 0b01111111;
```

```
	//전체 인터럽트를 비활성화
	cli();

	//TCNT0의 초기값 설정
	TCNT0 = TIMER0_TCNT0_PRELOAD;
	//타이머/카운터0 오버플로우 인터럽트 활성화
	TIMSK |= (1<<TOIE0);
	//타이머/카운터0 오버플로우 플래그를 0으로 만듦
	TIFR |= (1<<TOV0);
	//일반모드, OC0출력 없이 설정한 클럭 속도로 타이머/카운터0 시작
	TCCR0 = TIMER0_DIVIDER;

	//전체 인터럽트를 활성화
	sei();

	//while 안의 내용을 무한 반복
	while(1)
	{
		//LED가 차례로 하나씩 더 켜지도록 동작을 반복함
		for(i=0b00000001;i<=0b01111111;i=(i<<1)+1){
			PORTC = ~i;		//Active Low이므로 i 변수의 값을 반전하여 출력

			gucTimerTick = 100;	//200ms동안 대기
			while(gucTimerTick);
		}
	}

	//함수의 형태에 맞게 정수값 반환
	return 1;

}
```

예제설명

```
#include <avr/io.h>
#include <avr/interrupt.h>
```

avr/io.h 헤더 파일에는 DDRX, PORTX 레지스터 등 AVR 제어를 위해 필요한 레지스터 등이 정의되어 있고, avr/interrupt.h 헤더 파일에는 인터럽트를 사용하기 위한 함수들이 정의되어 있으므로 #include 문을 사용하여 포함시켜 줍니다.

```
#define TIMER0_DIVIDER                  0b00000101
#define TIMER0_TCNT0_PRELOAD            6
```

타이머/카운터0 오버플로우 인터럽트가 발생할 주기를 설정하는 두 항목을 알아보기 쉽게 #define으로 선언해줍니다. 이 둘의 의미와, 이 둘이 어떻게 인터럽트 주기를 결정하는지는 추후에 설명합니다.

```
volatile unsigned char gucTimerTick;
```

전역변수를 선언합니다. main 함수 내부에서 선언한 변수는 인터럽트 서비스 루틴에서 사용할 수가 없으므로, 전역변수를 선언하여 main 함수와 인터럽트 서비스 루틴 모두에서 사용할 수 있도록 합니다.

```
ISR(TIMER0_OVF_vect){
        TCNT0 = TIMER0_TCNT0_PRELOAD;

        if(gucTimerTick != 0){
                gucTimerTick--;
        }
}
```

타이머/카운터0 오버플로우 인터럽트 서비스 루틴입니다. 인터럽트가 발생하면 TCNT0 값은 0으로 초기화되어 있는데, 이 값을 TIMER0_TCNT0_PRELOAD라는 임의로 설정한 초기값으로 다시 바꿉니다. 그리고 전역변수인 gucTimerTick값이 0이 아니라면, 1을 감소 시키고 루틴을 마칩니다.

```
unsigned char i;

DDRC = 0b01111111;
PORTC = 0b01111111;
```

LED 값이 저장될 8비트 변수 i를 선언하고, 포트 C의 0~6번 핀을 출력으로 설정하고, 출력 값을 1로 만듭니다. 그러면 모든 LED가 꺼지게 됩니다.

```
cli();

TCNT0 = TIMER0_TCNT0_PRELOAD;
TIMSK |= (1<<TOIE0);
TIFR |= (1<<TOV0);
TCCR0 = TIMER0_DIVIDER;

sei();
```

cli()를 통해 SREG의 I 비트를 0으로 만듦으로써 전체 인터럽트를 비활성화 시킵니다. 설정을 바꾸는 도중에 인터럽트가 발생해 오동작이 하는 것을 방지하기 위함입니다.

TCNT0 값을 TIMER_TCNT0_PRELOAD 값으로 바꿉니다. 앞서 인터럽트 서비스 루틴에서도 보았듯이, TCNT0 값은 항상 TIMER_TCNT0_PRELOAD 값으로 초기화 됩니다. 이것은 원래 TCNT0이 0에서 255로 증가한 후에 타이머/카운터0 오버플로우 인터럽트가 발생하던 것에 반해, TIMER_TCNT0_PRELOAD라는 값에서 255로 증가한 후에 인터럽트가 발생하도록 해서 인터럽트가 발생하는 주기를 바꾼 것입니다. 원래대로라면 256번의 타이머 클럭이 있어야 오버플로우가 일어났는데, 이렇게 TCNT0를 초기화하면 오버플로우 주기가 (256-TIMER0_TCNT0_PRELOAD)로 줄어들게 됩니다. 우리 예제에서 오버플로우 주기는 250입니다.

TIMSK에서 TOIE0을 1로 만들어줍니다. 지금까지는 레지스터 값을 바꿀 때 0b00000001 등 직접 값을 입력했는데, 위와 같은 방법도 가능합니다. avr/io.h 헤더 파일에는 각 비트가 몇 번 비트에 해당하는지 저장되어 있습니다. TOIE0은 TIMSK의 0번 비트이므로 0으로 선언되어 있습니다. 때문에, TIMSK |= (1<<TOIE0); 같이 프로그램하면 TIMSK의 TOIE0 비트를 1로 설정하게 됩니다. 마찬가지로 TIFR도 TOV0 비트에 1을 써줌으로써 TOV0 비트를 0으로 초기화 시킵니다.

우리는 일반 모드로 동작 시킬 것이고, OC0 출력은 사용하지 않을 것이므로 TCCR0의 WGM01:0, COM01:0은 모두 0이 됩니다. 따라서 알맞은 CS02:0 값만 TCCR0에 넣어주면 일반 모드로 타이머/카운터 작동이 시작됩니다. TIMER0_DIVIDER가 그 CS02:0 값을 나타냅니다. 우리 예제에서는 0b101을 사용하여 기본 클럭인 16Mhz를 128로 나누어 줍니다. 이 경우 클럭 주파수는 125khz가 되고, 8us마다 TCNT0이 1씩 증가하게 됩니다. 8us마다 TCNT0이 1씩 증가하고, 250번마다 오버플로우 인터럽트가 발생하므로, 오버플로우 인터럽트 주기는 2ms가 됩니다. TIMER0_TCNT0_PRELOAD와 TIMER0_DIVIDER의 값을 바꾸면, 사용자 마음대로 오버플로우 인터럽트 주기를 만들 수 있습니다.

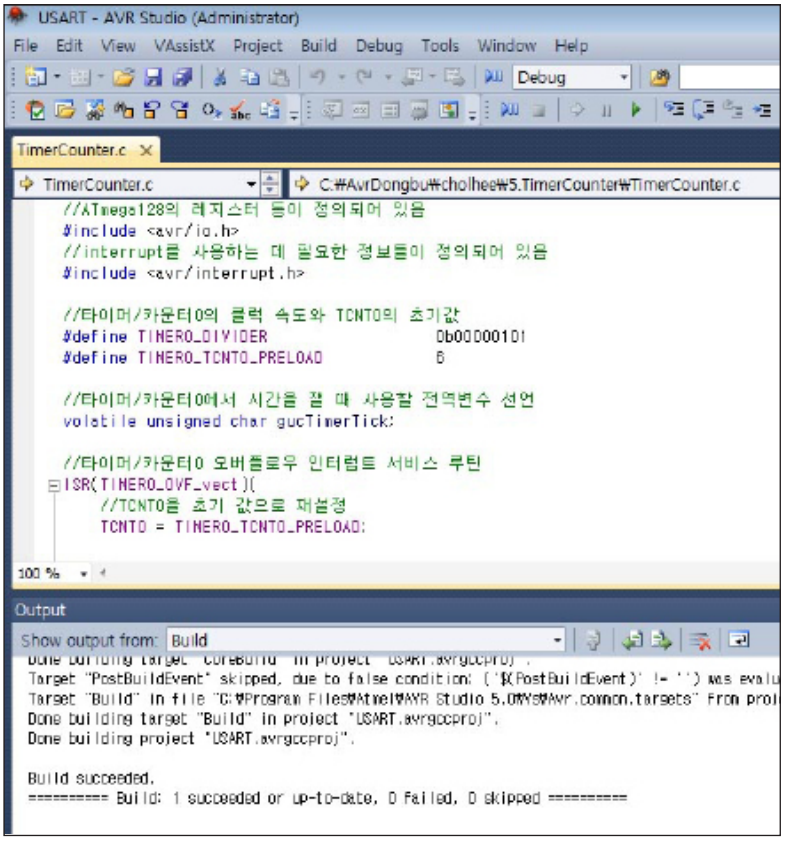
TCCR0의 CS02:0에 값을 넣어 타이머가 동작하기 시작하면, sei() 명령을 통해서 전체 인터럽트를 활성화 합니다. 이제부터는 2ms 마다 타이머/카운터0 오버플로우 인터럽트가 발생합니다.

```
for(i=0b00000001;i<=0b01111111;i=(i<<1)+1){
        PORTC = ~i;

        gucTimerTick = 100;
        while(gucTimerTick);
}
```

for문의 내용은 Chapter 1의 LED 내용과 같습니다. 하나 다른 점은 _delay_ms(200);이 gucTimerTick=100;과 while(gucTimerTick);으로 바뀌었다는 것입니다. gucTimerTick을 100으로 바꾸어주면, gucTimerTick은 2ms 마다 오버플로우 인터럽트 서비스 루틴에 의해서 1씩 감소합니다. 따라서 gucTimerTick이 0이 되기 위해서는 100번의 인터럽트 서비스 루틴이 필요합니다. while(gucTimerTick);은 gucTimerTick이 0이 아니면 while문을 반복하며 기다린다는 뜻이므로, gucTimerTick이 0이 되어야만 프로그램이 while문에서 빠져 나와 다시 for문의 처음으로 돌아가게 되며, 이것이 곧 200ms 동안 대기하는 효과가 됩니다. _delay_ms 함수의 기능을 타이머/카운터0 오버플로우 인터럽트로 구현한 것입니다.

실행

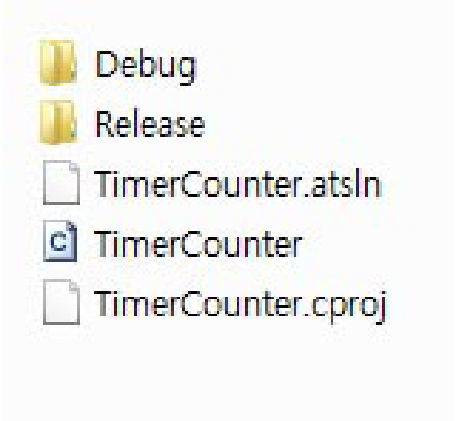
Chapter 1과 같이, Servo LED부터 Power LED까지 LED가 하나씩 추가로 켜집니다. LED 7개가 다 켜지고 난 후에는 다시 Servo LED만 켜지고, 차례로 하나씩 켜지는 것을 반복합니다.



01 Build

Build 메뉴를 선택하여 Build Solution을 클릭하여 Build 시킵니다.

1 succeeded 0 failed 가 나오면 에러없이 컴파일 완료됐다는 것을 의미합니다.



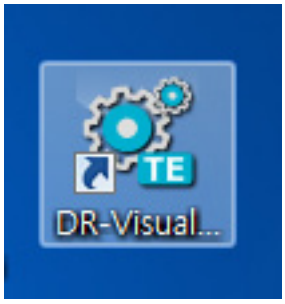
02 Debug 폴더

Build 후에 소스가 있는 폴더에 들어가면 Debug 폴더가 생성 되어있습니다. 클릭하여 폴더 안의 내용을 봅니다.

03 hex 확인

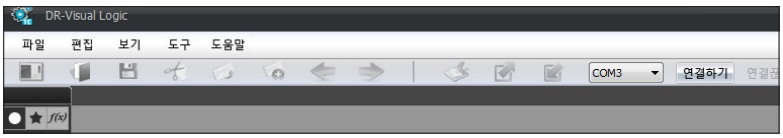
Debug 폴더 안의 hex 파일 확인합니다. Hex 파일이 제어기에 포팅시키는 파일입니다.

DR-Visual Logic 에서 펌웨어 복구를 실시합니다.



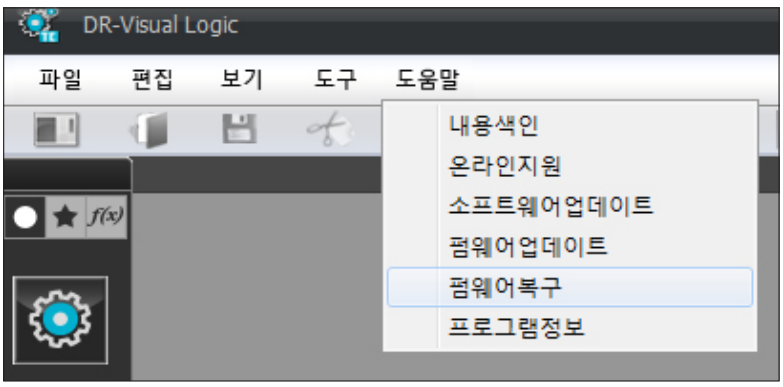
04 Visual Logic

DR-Visual Logic 을 실행시킵니다.



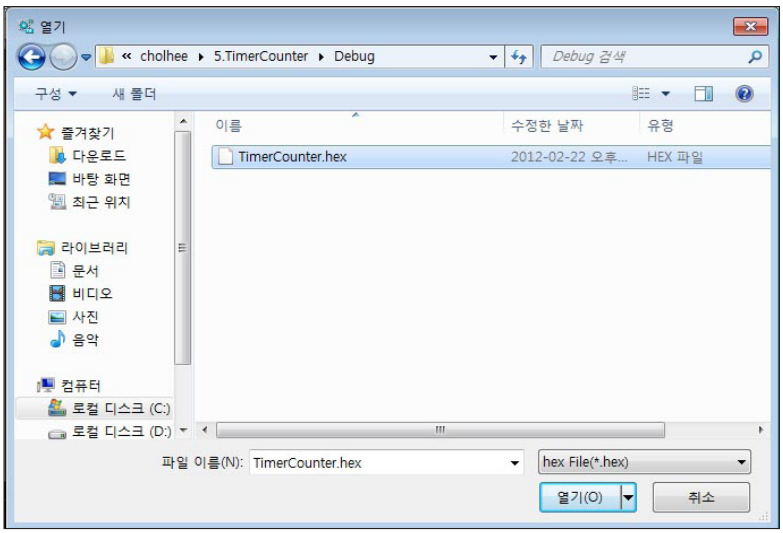
05 연결하기

COM 번호를 설정하고 연결하기를 클릭합니다.



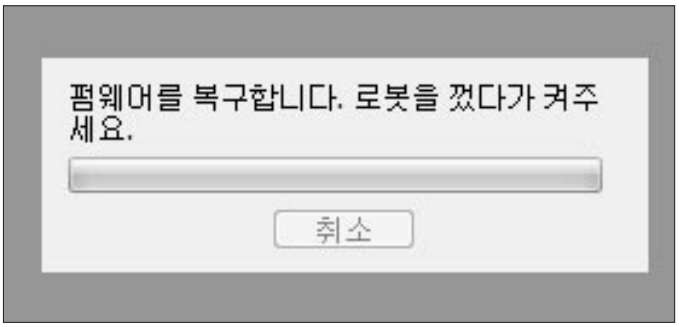
06 펌웨어 복구

도움말 메뉴에서 펌웨어복구를 클릭합니다.



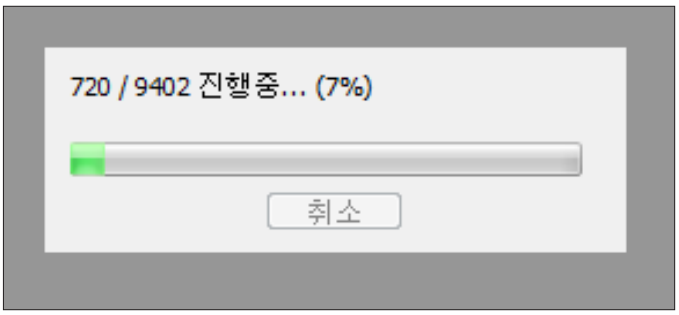
07 불러오기

Debug 폴더에서 Hex 파일을 불러옵니다



08 펌웨어복구

로봇을 껐다가 켜주세요 라는 문구가 나옵니다.
제어기의 전원버튼을 한번 껐다가 다시 켜주세요



09 복구시작

펌웨어 복구가 시작됩니다.

10 실행

Servo LED부터 Power LED까지 LED가 하나씩 추가로 켜집니다. LED 7개가 다 켜지고 난 후에는 다시 Servo LED만 켜지고, 차례로 하나씩 켜지는 것을 반복합니다.

CH.6 : ADC를 사용하여 PSD 센서 값 읽어오기

ATmega128의 아날로그 디지털 변환기(Analog-to-Digital Converter, ADC)를 사용하여 아날로그 PSD 센서의 출력 값을 읽어봅니다.

AVR 원리

ATmega128에는 8개 채널(ADC0~ADC7)에 10비트 분해능을 가지는 축차 비교형 아날로그 디지털 변환기(successive approximation analog-to-digital converter)가 내장되어 있습니다. 8개의 채널은 포트 F와 겸용으로 사용되며, 내부의 멀티플렉서에 의해서 어떤 포트를 사용할 것인지 선택할 수 있습니다. ADC 앞단에는 샘플/홀드(Sample and hold) 회로가 연결되어 있어, 변환이 수행되고 있는 동안에 입력된 아날로그 전압이 일정하게 유지되도록 해줍니다.

8개의 채널에서 하나의 채널을 단극성 아날로그 입력으로 받아 변환하는 동작 모드와 지정된 두 개의 채널의 전압 차이를 변환하는 차동 동작 모드가 있습니다. 차동 동작 모드에서 ADC1, ADC0과 ADC3, ADC2의 두 쌍은 전압 차이를 1배, 10배, 200배로 증폭해서 변환하는 기능도 있습니다.

ADC는 ATmega128의 전원 핀인 VCC로 작동하는 것이 아니라, 노이즈를 줄이기 위해서 독립적인 전원 핀인 AVCC로 전원 입력을 받아 작동됩니다. 그러나 AVCC가 VCC보다 0.3V이상 차이가 나서는 안됩니다.

ADC와 관련된 레지스터들은 아래와 같습니다.

ADMUX : ADC 멀티플렉서 선택 레지스터

비트	7	6	5	4	3	2	1	0
	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0
읽기/쓰기	W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
초기값	0	0	0	0	0	0	0	0

– 비트 7:6번 REFS1:0 : 기준 선택 비트

ADC는 아날로그 전압을 기준 전압인 VREF와 비교해서 그 상대적인 크기를 10bit의 디지털 값으로 변환하는 장치이므로, 디지털 값을 보고 아날로그 전압을 구하려면 기준 전압인 VREF를 알아야 합니다. ATmega128에는 AREF라는 핀이 있는데, 통상적인 경우에는 이 핀에 입력된 아날로그 전압이 기준 전압 VREF가 됩니다. 또한, 내부 레지스터(ADMUX) 설정을 통해 AVCC에 들어온 입력 전압을 기준 전압으로 사용하거나, ATmega128에 내장되어 있는 2.56V의 기준 전압을 사용할 수도 있습니다.

REFS1	REFS0	기준 전압 선택
0	0	AREF의 입력 전압
0	1	AVCC의 입력 전압
1	0	사용 안 함
1	1	2.56V의 내부 전압

– 비트 5번 ADLAR : ADC 결과 좌측 정렬

ADC로 변환된 10비트 데이터는 ADC Data Register인 ADCH와 ADCL에 저장됩니다. ADLAR이 0일 때에는 데이터의 하위 8비트가 ADCL에, 상위 2비트가 ADCH의 1:0번 비트에 저장됩니다. ADLAR이 1일 때에는 데이터의 상위 8비트가 ADCH에, 하위 2비트가 ADCL의 7:6번 비트에 저장됩니다.

– 비트 4:0번 MUX4:0 : 아날로그 채널 & 이득 선택 비트

이 다섯 비트의 값에 따라서 어떤 입력을 사용하고 어떤 이득으로 전압을 증폭해서 변환할지를 고를 수 있습니다. MUX4:0의 값에 따라서 어떤 입력을 사용하는지는 아래 표와 같습니다.

MUX4:0	단극성 입력	차동 입력		
		+단자	–단자	이득
00000	ADC0	–		
00001	ADC1			
00010	ADC2			
00011	ADC3			
00100	ADC4			
00101	ADC5			
00110	ADC6			
00111	ADC7			
01000	–	ADC0	ADC0	10x
01001		ADC1	ADC0	10x
01010		ADC0	ADC0	200x
01011		ADC1	ADC0	200x
01100		ADC2	ADC2	10x
01101		ADC3	ADC2	10x
01110		ADC2	ADC2	200x
01111		ADC3	ADC2	200x
10000		ADC0	ADC1	1x
10001		ADC1	ADC1	1x
10010		ADC2	ADC1	1x
10011		ADC3	ADC1	1x
10100		ADC4	ADC1	1x
10101		ADC5	ADC1	1x
10110		ADC6	ADC1	1x
10111		ADC7	ADC1	1x
11000		ADC0	ADC2	1x
11001		ADC1	ADC2	1x
11010		ADC2	ADC2	1x
11011		ADC3	ADC2	1x
11100		ADC4	ADC2	1x
11101		ADC5	ADC2	
11110	1.23V	–		
11111	0V			

■ ADCSR : ADC 제어, 상태 레지스터

비트	7	6	5	4	3	2	1	0
	ADEN	ADSC	ADFR	ADIF	ADIE	ADPS2	ADPS1	ADPS0
읽기/쓰기	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
초기값	0	0	0	0	0	0	0	0

– 비트 7번 ADEN : ADC 활성화

이 비트는 ADC의 동작을 허용하거나 금지합니다. 1로 설정하면 ADC가 동작합니다. 0으로 설정하면 변환 중이라도 ADC 꺼집니다.

– 비트 6번 ADSC : ADC 변환 시작

ADC의 변환을 시작하는 비트입니다. 1로 설정된 경우에 ADC가 변환을 시작합니다. 단일 변환 모드에서 1로 설정되면 ADSC는 변환이 진행되는 동안 1이다가 완료되면 0이 됩니다. 프리 러닝 모드에서는 ADSC가 1로 설정되면 첫 번째 변환이 시작되며 그 다음부터 계속 변환을 반복합니다. 평소 ADC 변환은 13번의 ADC 클럭 사이클이 필요하며, 활성화 후 첫 변환에서는 ADC의 초기화 과정 때문에 25번의 ADC 클럭 사이클이 소요됩니다. ADC의 클럭에 대해서는 이후의 ADPS2:0 비트에서 다룹니다.

– 비트 5번 ADFR : ADC 프리 러닝 선택

ADC의 프리 러닝 모드를 설정할 것인지 정합니다. 1로 설정되면 ADC는 ADSC가 1인 동안 계속 변환을 반복하며 데이터 레지스터를 업데이트 합니다. 0으로 설정하면 프리 러닝 모드가 종료되며 단일 변환 모드로 돌아갑니다.

– 비트 4번 ADIF : ADC 인터럽트 플래그

이 비트는 ADC 변환이 끝나서 데이터 레지스터가 업데이트 되면 1로 설정됩니다. 만약 SREG의 I 비트와 ADIE 비트가 1로 설정되어 있으면 ADC 변환 완료 인터럽트가 발생됩니다. 인터럽트가 발생해서 해당하는 ISR에 진입하면 ADIF는 자동으로 0으로 초기화 됩니다. 소프트웨어적으로 수동으로 이 비트를 0으로 변하게 하려면, 이 비트에 1을 써주어야 합니다.

– 비트 3번 ADIE : ADC 인터럽트 활성화

이 비트는 ADC 변환이 완료되었을 때 인터럽트를 허용하도록 설정하는 비트입니다. 이 비트와 SREG의 I 비트가 1이면, ADC 변환이 끝났을 때 ADC 변환 완료 인터럽트가 발생됩니다.

– 비트 2:0번 ADPS2:0 : ADC 프리스케일러 선택 비트

이 비트들은 ATmega128의 클럭 주파수를 몇으로 나누어 ADC에 입력할지를 정합니다. 기본적으로 ADC의 회로는 10비트의 최대 해상도로 ADC 변환을 마치기 위해서 50~200kHz의 주파수가 필요합니다. 만약 10비트보다 낮은 해상도가 필요하다면, 그보다 주파수를 높이면 샘플 주기를 높일 수 있습니다. ATmega128의 ADC에는 100kHz 이상의 어떤 클럭 주파수라도 나누어서 50~200kHz의 주파수를 얻을 수 있도록 프리스케일러를 제공하고 있으며 사용자가 자신의 클럭 주파수에 맞게 ADPS2:0을 설정하여 범위 이내의 ADC 클럭 주파수를 설정해야 합니다. 가령 예를 들어 DRC에서 사용하는 클럭 주파수는 16MHz이므로, 128로 나누면 125kHz를 만들어낼 수 있습니다.

ADPS2	ADPS1	ADPS0	분주비
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

ADCL, ADCH : ADC 데이터 레지스터

ADLAR = 0인 경우

비트	15	14	13	12	11	10	9	8
ADCH	–	–	–	–	–	–	ADC9	ADC8
읽기/쓰기	R	R	R	R	R	R	R	R
초기값	0	0	0	0	0	0	0	0
비트	7	6	5	4	3	2	1	0
ADCL	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0
읽기/쓰기	R	R	R	R	R	R	R	R
초기값	0	0	0	0	0	0	0	0

ADLAR = 0인 경우

비트	15	14	13	12	11	10	9	8
ADCH	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2
읽기/쓰기	R	R	R	R	R	R	R	R
초기값	0	0	0	0	0	0	0	0
비트	7	6	5	4	3	2	1	0
ADCL	ADC1	ADC0	–	–	–	–	–	–
읽기/쓰기	R	R	R	R	R	R	R	R
초기값	0	0	0	0	0	0	0	0

– ADC9:0 : ADC 변환 결과

ADC의 변환이 끝나면, ADCH와 ADCL의 두 레지스터에서 값을 읽을 수 있습니다. 데이터는 단극성 모드에서는 0~1023의 부호 없는 값으로 저장되고, 차동 모드에서는 –512~+511의 2의 보수 값으로 저장됩니다. ADCH와 ADCL의 저장된 10비트 값을 ADC라고 하면, 작동 모드에 따른 ADC 수식은 아래와 같습니다.

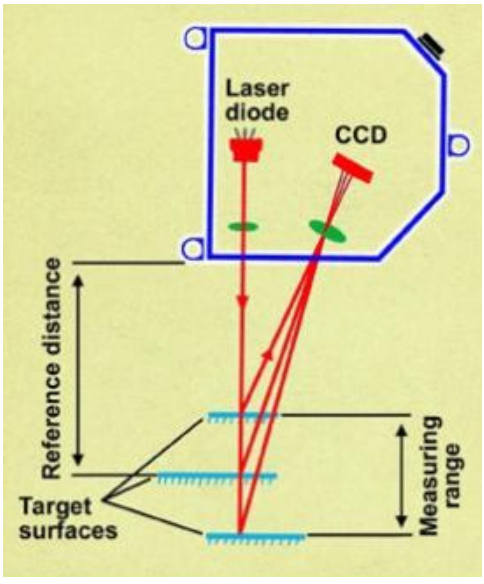
단극성 모드 : $$ADC = \frac{V_{IN} \times 1024}{V_{REF}}$$

차동 모드 : $$ADC = \frac{(V_{POS} - V_{NEG}) \times GAIN \times 512}{V_{REF}}$$

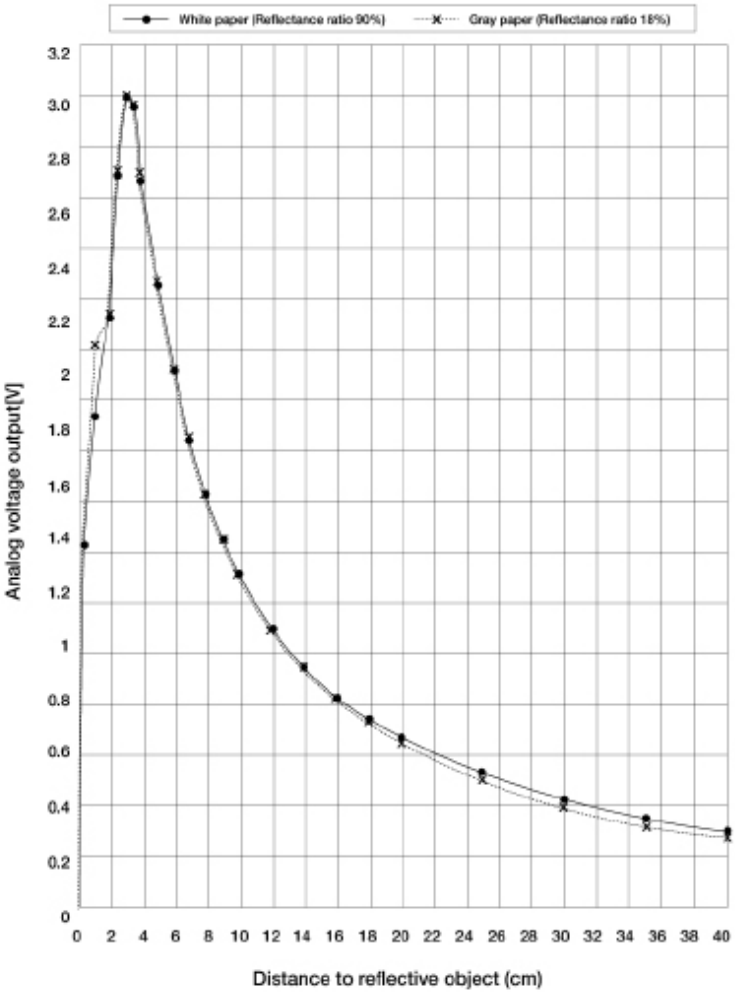
10비트가 아닌 8비트 해상도만으로 충분하다면 ADLAR = 1로 설정하고 ADCH만 읽어서 사용하면 됩니다. 그러나 두 레지스터의 내용 모두를 사용할 것이라면 항상 ADCL을 먼저 읽고 ADCH를 나중에 읽어야 합니다.

H/W 원리

이번 예제는 아날로그 PSD 센서 모듈을 DRC에 연결해 ADC 변환을 통해 센서의 출력 값을 읽어봅니다. PSD 센서는 적외선을 이용하여 센서 앞의 장애물까지의 거리를 재는 센서입니다. PSD 센서에는 두 개의 동그란 창이 있는데, 하나는 발광부이고 다른 하나는 수광부입니다. 발광부에서는 주기적으로 적외선을 쏘며, 적외선은 앞의 물체에서 반사되어 수광부로 다시 들어갑니다. 그림 01에서와 보는 바와 같이 물체와의 거리에 따라서 수광부로 반사되어 들어가는 빛의 각도가 달라져 수광부에 맺히는 위치가 달라지게 됩니다. PSD 센서는 수광부에 적외선이 들어온 위치에 따라서 출력 전압을 다르게 하고, 이 전압을 통해서 우리는 센서와 장애물의 거리를 알 수 있습니다. PSD 센서는 크게 아날로그 타입과 디지털 타입으로 나뉘며, 모델에 따라서 측정 거리도 다양하여 거리 인식 및 장애물 감지에 많이 응용되고 있습니다. 우리가 사용할 센서는 40cm까지의 측정 범위를 가지는 아날로그 타입의 PSD입니다. 거리에 따른 출력 전압은 아래 그림 02과 같습니다. DRC의 좌측 센서 포트는 ADC6(PF6)에 해당하며, 우측 센서 포트는 ADC7(PF7)에 해당합니다. 아날로그 PSD 센서를 좌측 센서 포트에 연결할 경우 ADC6을 통해서 센서 값을 읽어와야 합니다.



[그림1]



[그림2]

하드웨어 준비하기

01 연결

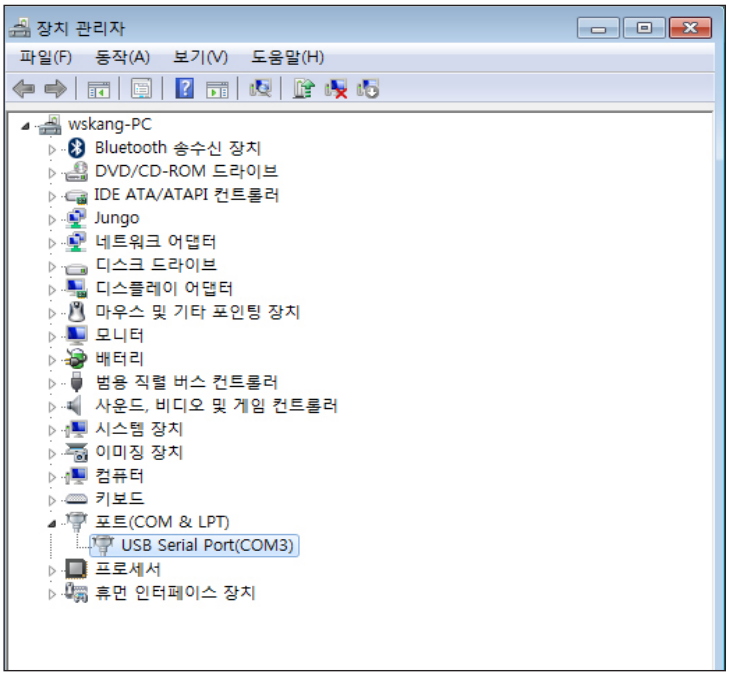
DRC 제어기 아답터로 전원을 연결하고, USB to Serial Cable 로 PC 와 DRC 를 연결합니다.

02 PSD 센서 준비

PSD 센서를 준비합니다.

03 PSD 센서 연결

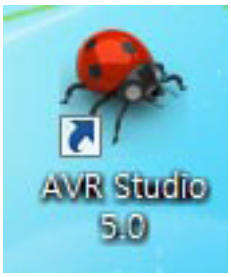
PSD 센서를 제어기 좌측 포트에 연결합니다.



04 COM port 연결

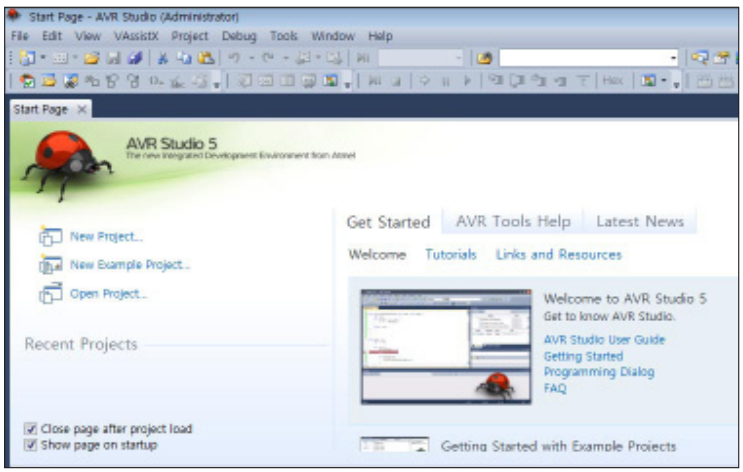
내컴퓨터 〉 속성 〉 장치관리자에서
COM 연결 확인합니다.

AVR Studio 실행 및 코딩 준비



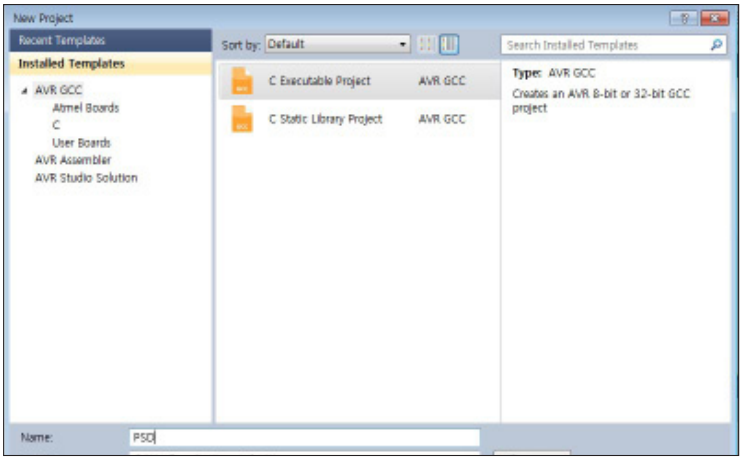
01 실행

AVR Studio 를 실행합니다.



02 New Project

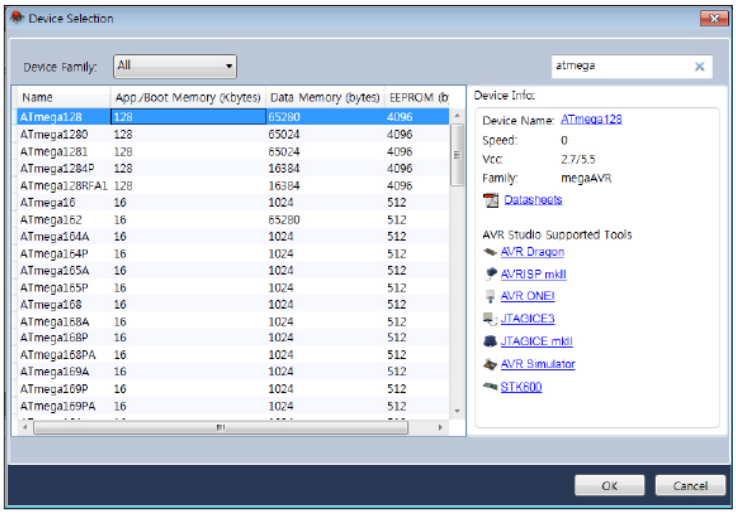
New Project 를 클릭합니다.



03 GCC 선택

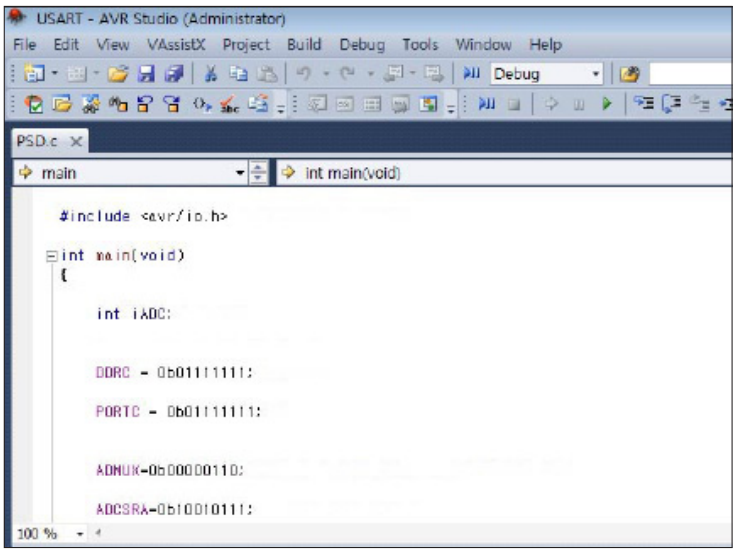
C Executable Project 를 선택합니다.

Name 을 "PSD"로 입력하고, Create directory for solution 을 클릭하고, OK 버튼을 누릅니다.



04 ATmega 선택

우측 상단에 atmega 를 검색하여, 리스트중에 ATmega128 을 선택합니다.



05 Editor

편집창에 코딩을 시작합니다.

아래 예제를 따라하세요.

예제

DRC의 좌측 센서 포트에 연결된 아날로그 PSD 센서의 출력 값을 ADC를 사용해 변환하고, 거리 값이 10cm보다 가까우면 LED를 켜는 예제입니다.

PSD.c

```
//ATmega128의 레지스터 등이 정의되어 있음
#include <avr/io.h>

int main(void)
{
	//16비트 변수 선언
	int iADC;

	//포트 C의 입출력 방향 설정
	DDRC = 0b01111111;
	//포트 C의 출력 값 설정
	PORTC = 0b01111111;

	//VREF=AREF 입력 전압, 좌측 정렬 끔, ADC 6번 단극성 입력
	ADMUX=0b00000110;
	//ADC 활성화, ADIF 초기화, 클럭 분주비 128
	ADCSRA=0b10010111;

	//while 안의 내용을 무한 반복
	while(1)
	{
		//ADC 변환 시작
		ADCSRA |= (1<<ADSC);
		//ADC 변환 완료까지 기다림
		while(!(ADCSRA & (1<<ADIF)));
		//ADCL과 ADCH 값을 읽어서 저장
		iADC = ADCL | (ADCH<<8);
		//ADIF 초기화
		ADCSRA |= (1<<ADIF);

		// 1.3/5*1024 = 266.24
		if(iADC > 266)
			PORTC = 0b00000000;
		else
			PORTC = 0b01111111;
	}

	//함수의 형태에 맞게 정수값 반환
	return 1;
}
```

예제설명

```
#include <avr/io.h>
```

avr/io.h 헤더 파일에는 DDRX, PORTX 레지스터 등 AVR 제어를 위해 필요한 레지스터 등이 정의되어 있으므로 #include 문을 사용하여 포함시켜 줍니다.

```
int iADC;

DDRC = 0b01111111;
PORTC = 0b01111111;
```

ADC 값이 저장될 16비트 변수 iADC를 선언하고, 포트 C의 0~6번 핀을 출력으로 설정하고, 출력 값을 1로 만듭니다. 그러면 모든 LED가 꺼지게 됩니다.

```
ADMUX=0b00000110;
ADCSRA=0b10010111;
```

ADC를 단극성 모드로 ADC6 포트로부터 아날로그 입력을 읽어오기 위한 레지스터 설정을 합니다. ADMUX부터 살펴보면, DRC의 회로상에서 기준 전압은 ATmega128의 동작전압인 5V와 동일하게 AREF 핀으로 연결되어 있으므로 REFS1:0 비트는 0b00으로 설정하며, 좌측 정렬을 사용하지 않으므로 ADLAR도 0으로 설정합니다. 그리고 ADC6 포트를 단극성 모드로 읽을 것이므로 MUX4:0 비트는 0b00110으로 설정합니다.
ADCSRA에서는 우선 ADC를 활성화 하기 위해 ADEN 비트를 1로 설정하고, 아직 변환 시작을 할 것이 아니고 프리 러닝도 사용하지 않으므로 ADSC, ADFR은 0으로 설정합니다. ADIF의 경우는 혹시라도 1로 설정되어 있을 수 있으므로 0으로 다시 초기화하기 위해 1을 써주며, 인터럽트를 사용하지 않을 것이므로 ADIE는 0으로 설정합니다. 마지막으로 ATmega128의 16MHz 클럭 주파수를 ADC에 적합한 50~200kHz로 나누어주기 위해 ADPS2:0 비트를 0b111로 설정해서 128로 나눈 125kHz를 ADC 클럭으로 공급하도록 설정합니다.

```
ADCSRA |= (1<<ADSC);
while(!(ADCSRA & (1<<ADIF)));
iADC = ADCL | (ADCH<<8);
ADCSRA |= (1<<ADIF);
```

while문에서는 ADC 변환을 계속 반복합니다. 우선 ADCSRA의 ADSC 비트를 1로 만들어 줍니다. 그러면 ADC 변환이 시작됩니다.
while(!(ADCSRA & (1<<ADIF)));은 !(ADCSRA & (1<<ADIF))를 만족하는 동안 아무 일도 하지 않고 대기하라는 명령입니다. !(ADCSRA & (1<<ADIF))를 만족하는 것은 ADCSRA의 ADIF가 0일 때이므로, 이는 즉 ADIF 비트가 0인 동안 기다리고 1이 되는 순간 while문을 빠져 나와 다음으로 진행하라는 의미입니다.

ADC 변환이 완료되고 ADCL과 ADCH가 업데이트 되어 ADIF 비트가 1이 된 후에는, ADCL과 ADCH 값을 읽어서 iADC에 저장합니다. ADCH는 ADC값의 상위 1바이트에 해당하므로 8비트(1바이트)만큼 왼쪽으로 〈〈(shift left)연산을 해서 ADCL과 |(bitwise OR) 연산해 iADC에 저장하면 10비트 데이터를 iADC에 알맞게 저장할 수 있습니다. 레지스터 설명에서 언급하였듯이, ADCL을 먼저 읽은 후 ADCH를 읽어야 하므로 식에서 ADCL을 더 먼저 써서 ADCL 레지스터 값을 먼저 읽어오도록 했습니다.
우리는 ADC 변환 완료 인터럽트를 사용하지 않으므로, ADIF 비트가 자동으로 0으로 설정되지 않습니다. 그래서 ADC값을 읽은 후에는 인터럽트 플래그를 다시 0으로 초기화 시켜야 합니다. ADCSRA의 ADIF 비트에 1을 써주어 이 비트를 0으로 초기화 시킵니다.

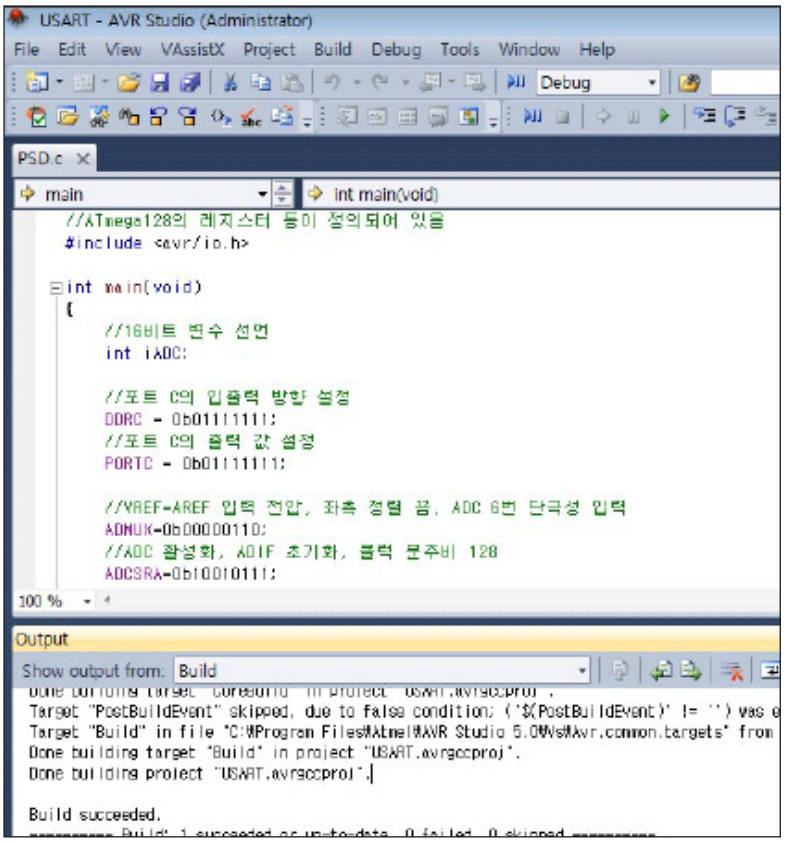
```
if(iADC > 266)
        PORTC = 0b00000000;
else
        PORTC = 0b01111111;
```

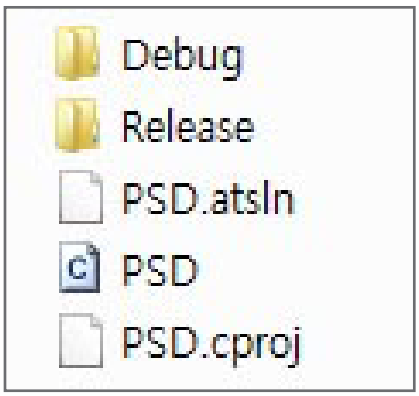
iADC의 값에 따라서 LED를 점멸합니다. 그림 00의 PSD 센서의 거리에 따른 전압 값 그래프를 보면, 물체와의 거리가 10cm 일 때 센서의 출력 값은 약 1.3V가 되는 것을 볼 수 있습니다. 1.3V에 해당하는 ADC 출력 값은 $ADC = \frac{V_{IN} \times 1024}{V_{REF}}$

이므로, 수식을 따라서 계산하면 1.3×1024÷5 = 266.24가 됩니다. 반올림하면, 입력 전압이 1.3V일 때 ADC 값은 266 정도가 될 것입니다.
iADC가 266보다 크면 물체가 10cm보다 가까이 있다는 뜻이고, 작으면 10cm보다 멀리 있다는 뜻입니다. if–else 문에서는 물체가 10cm보다 가까우면 모든 LED를 켜고, 10cm보다 멀면 모든 LED를 끕니다.

실행

DRC의 좌측 센서 포트에 연결된 아날로그 PSD 센서의 출력 값을 ADC를 사용해 변환하고, 거리 값이 10cm보다 가까우면 LED를 켜는 예제입니다.





01 Build

Build 메뉴를 선택하여 Build Solution을 클릭하여 Build 시킵니다.

1 succeeded 0 failed 가 나오면 에러없이 컴파일 완료됐다는 것을 의미합니다.

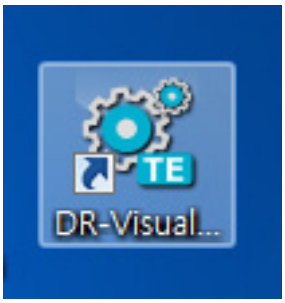
02 Debug 폴더

Build 후에 소스가 있는 폴더에 들어가면 Debug 폴더가 생성 되어있습니다. 클릭하여 폴더 안의 내용을 봅니다.

03 hex 확인

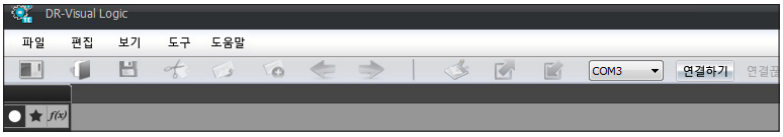
Debug 폴더 안의 hex 파일 확인합니다. Hex 파일이 제어기에 포팅시키는 파일입니다.

DR-Visual Logic 에서 펌웨어 복구를 실시합니다.



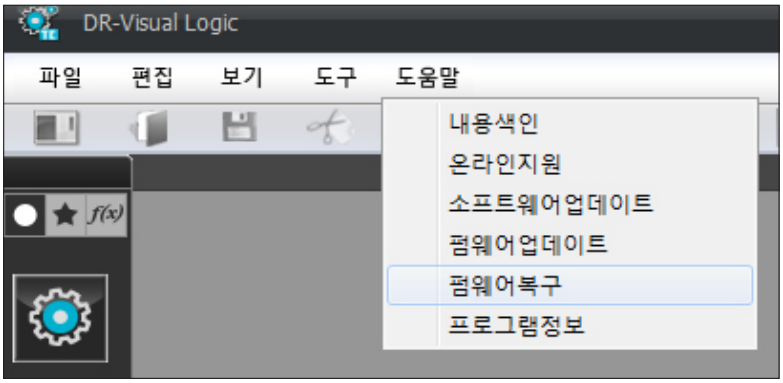
04 Visual Logic

DR-Visual Logic 을 실행시킵니다.



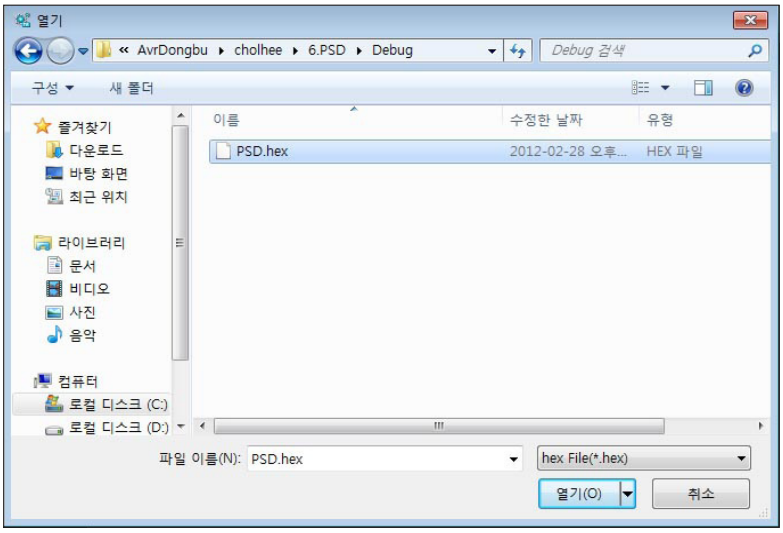
05 연결하기

COM 번호를 설정하고 연결하기를 클릭합니다.



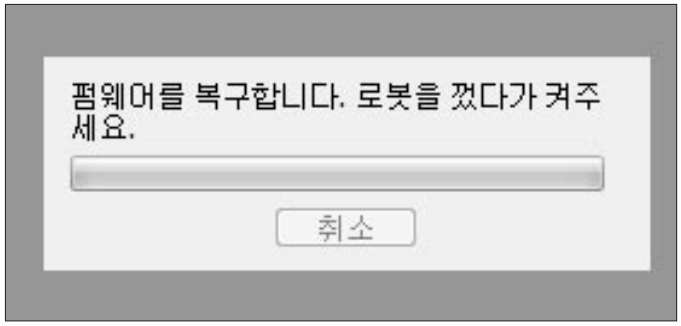
06 펌웨어 복구

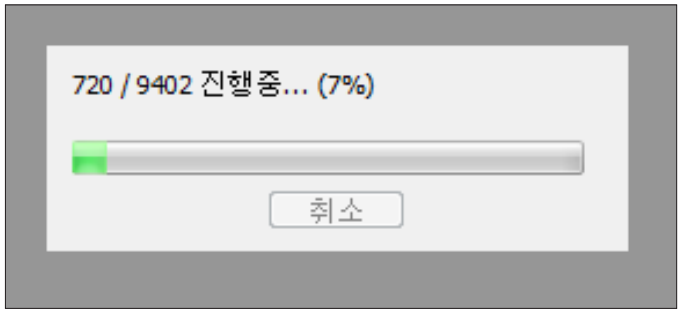
도움말 메뉴에서 펌웨어 복구를 클릭합니다.



07 불러오기

Debug 폴더에서 Hex 파일을 불러옵니다





(아무것도 없으면 불이 안 켜짐)

(손을 가까이 가져가면 불이 켜짐)

08 펌웨어복구

로봇을 껐다가 켜주세요 라는 문구가 나옵니다.
제어기의 전원버튼을 한번 껐다가 다시 켜주세요

09 복구시작

펌웨어 복구가 시작됩니다.

10 실행

PSD 센서에 10cm 이내로 손을 가져다 대면 DRC의 모든 LED가 켜지고, 손을 멀리하면 LED가 꺼집니다.

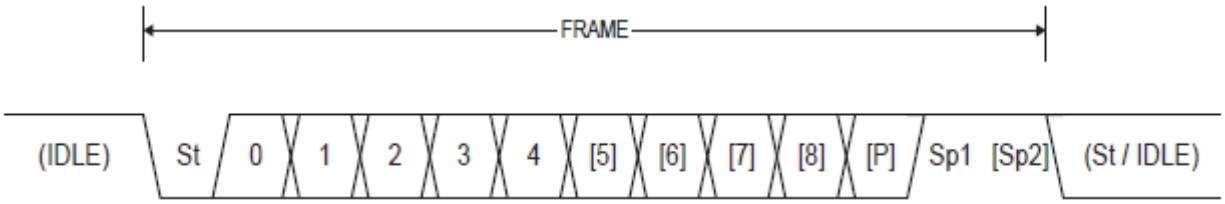
CH.7 : USART를 사용하여 시리얼 통신하기

ATmega128의 USART(Universal Synchronous and Asynchronous serial Receiver and Transmitter)를 사용하여 PC와 DRC 간에 시리얼 통신(직렬 통신)을 해봅니다.

AVR 원리

ATmega128에는 USART0, USART1의 두 USART가 내장되어 있습니다. USART는 일반 비동기, 2배속 비동기, 마스터 동기, 슬레이브 동기의 네 가지 모드로 사용할 수 있습니다. 이름에서 알 수 있듯이 비동기 통신과 동기 통신을 모두 지원합니다. 우리가 사용할 모드는 비동기 통신으로, 그 중 2배속 비동기를 사용합니다.

ATmega128의 USART는 클럭 생성 블록, 송신 블록, 수신 블록, 레지스터 블록의 네 가지 부분으로 이루어져 있습니다. 클럭 생성 블록은 USART가 통신을 할 때 필요한 클럭을 발생하는 회로입니다. 비동기 통신을 할 때에는 자체적으로 발생하는 클럭을 사용하지만, 동기 통신을 할 때는 마스터에서는 자체 발생 클럭을 발생해 슬레이브에 출력하고, 슬레이브에서는 그 클럭 신호를 받아 동기화 해서 통신을 합니다. 송신 블록은 USART의 송신부로, 레지스터에 써진 8비트 병렬 데이터를 직렬로 송신하는 역할을 합니다. 수신 블록은 USART의 수신부로, 직렬로 수신 받은 직렬 데이터를 병렬 데이터로 변환해 레지스터에 저장하는 기능을 합니다. 레지스터 블록은 USART를 제어하기 위한 각종 제어 비트와 설정 비트, 그리고 일부 데이터를 저장하는 비트로 사용됩니다.



[그림.1]

그림 01은 USART의 송수신 프레임을 나타내고 있습니다. 프레임은 시리얼 통신에서 데이터 비트와 동기화 비트(시작 비트, 정지 비트), 그리고 부가적으로 패리티 비트까지를 의미합니다. 시리얼 프레임의 형식은 아래 사항 중에 하나를 골라서 조합함으로써 만들어집니다.

– 1개의 시작 비트

– 5, 6, 7, 8 혹은 9개의 데이터 비트

– 짝수 패리티 비트 1개 혹은 홀수 패리티 비트 1개 혹은 사용하지 않음

– 1 혹은 2개의 정지 비트

가장 많이 사용되는 형식은 8개의 데이터 비트, 패리티 비트 사용 안 함, 1개의 정지 비트이며 DRC와 HerkuleX도 이 형식을 사용합니다.

통신이 일어나지 않는 상태(IDLE)에서는 데이터 라인은 HIGH로 유지됩니다. 시작 비트(St)는 항상 LOW 여야 하고, 그 후 데이터 비트(0~[8])가 전송됩니다. 데이터 비트는 형식에 따라 5~9비트가 전송 됩니다. 그림 01의 대괄호는 형식에 따라 대괄호 친 비트는 없을 수도 있다는 뜻입니다. 그 후 설정에 따라서 패리티 비트([P])가 전송될 수도 있고, 아닐 수도 있습니다. 그 후 정지 비트(Sp1~[Sp2])가 1~2비트 전송되면 프레임이 끝나며, 데이터 라인은 다시 IDLE 상태로 돌아가거나 다음 데이터의 시작 비트를 시작합니다.

가장 많이 사용되는 형식인 8개의 데이터 비트, 패리티 비트 사용 안 함, 1개의 정지 비트를 사용하면, 한 프레임의 총 비트 수는 시작 비트 1 + 데이터 비트 8 + 정지 비트 1로 총 10비트가 됩니다.
USART와 관련된 레지스터들은 아래와 같습니다. 레지스터 이름의 n은 0 혹은 1로, n의 자리에 0과 1을 넣으면 각각 USART0과 USART1에 대항하는 레지스터임을 뜻합니다.

UDRn : USART 송/수신 데이터 레지스터

비트	7	6	5	4	3	2	1	0
Read	RXBn[7:0]							
Write	TXBn[7:0]							
읽기/쓰기	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
초기값	0	0	1	0	0	0	0	0

USARTn 송신 데이터 버퍼 레지스터와 USARTn 수신 데이터 버퍼 레지스터는 같은 UDRn라는 데이터 레지스터로 접근됩니다. UDRn에 데이터를 쓰면 그 값이 송신 데이터 레지스터에 써지고, UDRn으로부터 데이터를 읽어오면 수신 데이터 레지스터의 값이 읽힙니다.
송신 데이터 레지스터는 UCSRAn의 UDREn 비트가 1일 때만 값을 쓸 수 있습니다. 송신이 활성화 되어 있을 때 데이터를 쓰면 그 데이터는 시프트 레지스터로 이동해 한 비트 씩 직렬로 전송됩니다.
수신 데이터 레지스터는 2단계 FIFO로 구성되어 있어서 두 개의 문자 데이터를 받는데, FIFO 동작의 특성상 수신 데이터 레지스터는 한 번 접근될 때마다 그 다음 데이터로 값이 바뀌어 버립니다. 따라서 UDRn으로부터 데이터를 읽을 때나 RXBn[7:0]중 특정 비트를 접근할 때에는 UDRn을 읽어와서 다른 변수에 저장한 후 처리하는 편이 안전합니다.

UCSRnA : USART 제어, 상태 레지스터 A

비트	7	6	5	4	3	2	1	0
	RXCn	TXCn	UDREn	FEn	DORn	UPEn	U2Xn	MPCMn
읽기/쓰기	R	R/W	R	R	R	R	R/W	R/W
초기값	0	0	1	0	0	0	0	0

– 비트 7번 RXCn : USART 수신 완료

수신 완료 상태를 나타내는 비트로 수신 데이터 레지스터에 데이터가 들어와 수신을 마치면 1로 설정되며, 데이터가 없으면 0으로 설정됩니다. RXCn 비트는 USARTn 수신 완료 인터럽트를 일으키는 데에 사용될 수도 있습니다.

– 비트 6번 TXCn : USART 송신 완료

송신 완료 상태를 나타내는 비트로 송신 데이터 레지스터에 있는 데이터가 모두 송신되면 1로 설정됩니다. USARTn 송신 완료 인터럽트를 일으키는 데에 사용될 수 있으며, 인터럽트 서비스 루틴으로 진입할 때 0으로 초기화 됩니다. 또는 이 비트에 1을 쓰면 수동으로 0으로 초기화할 수 있습니다.

– 비트 5번 UDREn : USART 데이터 레지스터 비움

이 비트는 송신 버퍼(UDRn)이 새 데이터를 받을 준비가 되었음을 알립니다. UDREn이 1일 때 UDRn에 데이터를 쓰면 데이터가 전송됩니다. UDREn은 USARTn 데이터 레지스터 비움 인터럽트를 발생시키는 데에 사용될 수 있습니다.

– 비트 4번 FEn : 프레임 에러

수신 버퍼에 들어온 다음 문자에 프레임 에러가 일어났음을 알리는 비트입니다. 데이터 프레임에서 첫 번째 정지 비트는 항상 1이어야 하는데, 받은 데이터의 첫 정지 비트가 0일 경우 1로 설정됩니다.

– 비트 3번 DORn : 데이터 오버런

데이터가 수신되는 동안 오버런 오류가 발생하였음을 나타내는 비트입니다. 수신 버퍼의 2단계 FIFO에 이미 문자 데이터 2개가 들어와 있는 상태에서 또 다른 시작 비트가 수신 핀에 감지되었을 때 설정됩니다.

– 비트 2번 UPEn : 패리티 오류

데이터를 수신하는 동안 패리티 오류가 발생했음을 나타내는 비트 입니다. 이 오류는 UCSRnC 레지스터의 UPMn1:0 비트에서 패리티를 사용하도록 설정했을 때만 발생합니다.

– 비트 1번 U2Xn : 2배속 USART 전송 속도

이 비트는 비동기 통신 시에만 효과가 있으며, 동기 통신 시에는 0으로 설정해야 합니다. 이 비트에 1을 쓰면 클럭 생성 블록에서 USART를 위한 클럭을 발생시킬 때 분주비를 16에서 8로 만들어서 같은 UBRRL, UBRRH 값에서 USART의 통신 속도를 두 배로 만듭니다. 이 비트가 0으로 설정되어 있을 때 분주비는 그대로 16입니다.

– 비트 0번 MPCMn : 다중 프로세서 통신 모드

이 비트는 다중 프로세서 통신 모드를 활성화 시킵니다. MPCMn 비트가 1로 설정되면 USARTn에서 수신하는 데이터 중 주소 정보를 포함하지 않는 데이터는 모두 무시됩니다. 여러 프로세서를 한 통신라인으로 제어할 때 사용됩니다.

■ UCSRnB : USART 제어, 상태 레지스터 B

비트	7	6	5	4	3	2	1	0
	RXCIEn	TXCIEn	UDRIEn	RXENn	TXENn	UCSZn2	RXB8n	TXB8n
읽기/쓰기	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W
초기값	0	0	0	0	0	0	0	0

– 비트 7번 RXCIEn : 수신 완료 인터럽트 활성화

이 비트에 1을 쓰면 RXCn 플래그가 인터럽트를 일으키도록 활성화합니다. SREG의 I 비트와 RXCIEn 비트가 1로 설정되어 있고 UCSRnA의 RXCn 플래그가 1이 되면 USARTn 수신 완료 인터럽트가 발생합니다.

– 비트 6번 TXCIEn : 송신 완료 인터럽트 활성화

이 비트에 1을 쓰면 TXCn 플래그가 인터럽트를 일으키도록 활성화합니다. SREG의 I 비트와 TXCIEn 비트가 1로 설정되어 있고 UCSRnA의 TXCn 플래그가 1이 되면 USARTn 송신 완료 인터럽트가 발생합니다.

– 비트 5번 UDRIEn : 데이터 레지스터 비움 인터럽트 활성화

이 비트에 1을 쓰면 UDREn 플래그가 인터럽트를 일으키도록 활성화합니다. SREG의 I 비트와 UDRIEn 비트가 1로 설정되어 있고 UCSRnA의 UDREn 플래그가 1이 되면 USARTn 데이터 레지스터 비움 인터럽트가 발생합니다.

– 비트 4번 RXENn : 수신 활성화

이 비트에 1을 쓰면 USARTn의 수신 블록이 활성화 됩니다. RXENn 비트가 1로 설정되면 ATmega128의 RxDn 핀이 I/O 포트 설정에 상관없이 직렬 데이터 수신 핀으로 동작합니다.

– 비트 3번 TXENn : 송신 활성화

이 비트에 1을 쓰면 USARTn의 송신 블록이 활성화 합니다. TXENn 비트가 1로 설정되면 ATmega128의 TxDn 핀이 I/O 포트 설정에 상관없이 직렬 데이터 송신 핀으로 동작합니다. 데이터를 송신 중일 때 TXENn에 0을 쓰면 현재 송신 중인 데이터를 모두 보내고 나서 송신 핀이 비활성화 됩니다.

– 비트 2번 UCSZn2 : 문자 크기

이 비트는 UCSRnC의 UCSZn1:0 과 함께 데이터의 크기를 설정하는데 사용됩니다. 자세한 내용은 UCSZn1:0 비트에서 다룹니다.

– 비트 1번 RXB8n : 수신 데이터 비트 8

RXB8n 비트는 데이터 비트 수가 9개로 설정되어 동작할 때 수신된 데이터 비트 중 9번째 비트(8번 비트)가 저장되어 있습니다. 항상 UDRn에 접근해서 수신 데이터의 하위 비트 RXBn[7:0]을 읽기 전에 RXB8n을 먼저 읽어야 합니다.

– 비트 0번 TXB8n : 송신 데이터 비트 8

TXB8n 비트는 데이터 비트 수가 9개로 설정되어 동작할 때 송신할 데이터 비트 중 9번째 비트(8번 비트)를 나타냅니다. 항상 UDRn에 송신 데이터의 하위 비트 TXBn[7:0]을 쓰기 전에 TXB8n을 먼저 써야 합니다.

■ UCSRnC : USART 제어, 상태 레지스터 C

비트	7	6	5	4	3	2	1	0
	–	UMSELn	UPMn1	UPMn0	USBSn	UCSZn1	UCSZn0	UCPOLn
읽기/쓰기	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
초기값	0	0	0	0	0	1	1	0

– 비트 6번 UMSELn : USART 모드 선택

UMSELn은 USART를 비동기 통신으로 사용할지 동기 통신으로 사용할지 고르는 비트입니다. 0일 때 비동기 통신, 1일 때 동기 통신으로 설정됩니다.

– 비트 5:4번 UPMn1:0 : 패리티 모드

패리티 비트의 발생과 검사 형태를 설정하는 비트들입니다. 패리티를 사용하도록 설정하면, 송신 블록에서는 자동으로 패리티 비트를 만들어 데이터 비트 뒤에 송신하고, 수신 블록에서는 들어오는 데이터의 패리티를 검사합니다. 검사 결과 오류가 발생한 경우에는 UCSRnA 레지스터의 UPEn 플래그가 1로 설정됩니다.

UPMn1	UPMn0	패리티 모드
0	0	패리티 비트 없음
0	1	(사용 안함)
1	0	짝수 패리티 비트 사용
1	1	홀수 패리티 비트 사용

– 비트 3번 USBSn : 정지 비트 선택

정지 비트의 설정을 고르는 비트입니다. 0으로 설정되면 정지 비트가 1개, 1로 설정되면 2개로 설정됩니다.

– 비트 2:1번 UCSZn1:0 : 문자 크기

UCSRnB 레지스터의 UCSZn2와 함께, 송신 블록과 수신 블록에서 사용하는 프레임의 데이터 비트 수(문자 크기)를 설정하는 비트입니다.

UCSZn2	UCSZn1	UCSZn0	문자 크기
0	0	0	5비트
0	0	1	6비트
0	1	0	7비트
0	1	1	8비트
1	0	0	(사용 안함)
1	0	1	(사용 안함)
1	1	0	(사용 안함)
1	1	1	9비트

– 비트 0번 UCPOLn : 클럭 극성

이 비트는 동기 모드에서만 사용되며, 비동기 모드에서는 0으로 설정해야 합니다. UCPOLn 비트는 동기 클럭(XCKn)과 데이터 출력 변화 시점/데이터 입력 샘플링 시점의 관계를 정합니다.

UCPOLn	데이터 출력 변화 시점	데이터 입력 샘플링 시점
0	상승 에지일 때	하강 에지일 때
1	하강 에지일 때	상승 에지일 때

UBRRnL, UBRRnH : USART 보드 레이트 레지스터

비트	15	14	13	12	11	10	9	8
UBRRnH	–	–	–	–	UBRRn[11:8]			
읽기/쓰기	R	R	R	R	R/W	R/W	R/W	R/W
초기값	0	0	0	0	0	0	0	0
비트	7	6	5	4	3	2	1	0
UBRRnL	UBRRn[7:0]							
읽기/쓰기	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
초기값	0	0	0	0	0	0	0	0

– 비트 11:0번 UBRRn11:0 : USARTn 보드 레이트 레지스터

이 12비트의 값은 USARTn의 보드 레이트(baud rate), 즉 통신 속도를 설정합니다. USART 통신의 보드 레이트는 bps(bit per second)로 나타내집니다. 즉, 1초 동안 몇 비트를 보내는 지를 통신 속도의 척도로 사용합니다. 자주 사용되는 보드 레이트에는 2400, 4800, 9600, 14400, 19200, 38400, 57600, 115200, 230400, 460800, 921600 bps 등이 있으며, DRC와 HerkuleX는 115200 bps를 표준 보드 레이트로 사용합니다. 통신 모드와 U2Xn 비트의 값에 따른 보드 레이트 및 UBRR 계산식은 아래와 같습니다.

동작 모드	보드 레이트 계산식	UBRR 계산식
일반 비동기 모드(U2Xn = 0)	$BAUD = \frac{f_{osc}}{16\ (UBRR+1)}$	$UBRR = \frac{f_{osc}}{16\ BAUD} - 1$
2배속 비동기 모드(U2Xn = 1)	$BAUD = \frac{f_{osc}}{8(UBRR+1)}$	$UBRR = \frac{f_{osc}}{8BAUD} - 1$
마스터 동기 모드	$BAUD = \frac{f_{osc}}{2(UBRR+1)}$	$UBRR = \frac{f_{osc}}{2BAUD} - 1$

여기서 BAUD는 보드 레이트, fOSC는 시스템 클럭 주파수, UBRR은 UBRR 레지스터의 값입니다.

	f_{osc} = 16.0000 MHz			
	U2X = 0		U2X = 1	
Baud Rate (bps)	UBRR	Error	UBRR	Error
2400	416	-0.1%	832	0.0%
4800	207	0.2%	416	-0.1%
9600	103	0.2%	207	0.2%
14.4k	68	0.6%	138	-0.1%
19.2k	51	0.2%	103	0.2%
28.8k	34	-0.8%	68	0.6%
38.4k	25	0.2%	51	0.2%
57.6k	16	2.1%	34	-0.8%
76.8k	12	0.2%	25	0.2%
115.2k	8	-3.5%	16	2.1%
230.4k	3	8.5%	8	-3.5%
250k	3	0.0%	7	0.0%
0.5M	1	0.0%	3	0.0%
1M	0	0.0%	1	0.0%
Max (1)	1 Mbps		2 Mbps	

상단 그림 은 DRC의 시스템 클럭 주파수인 16MHz에서 어떤 UBRR값을 설정해야 자주 사용하는 보드 레이트를 얻을 수 있는지 표로 나타낸 것입니다. DRC와 HerkuleX의 표준 보드 레이트인 115200bps를 설정하려면 일반 모드에서 8을, 2배속 모드에서 16을 설정해야 합니다. DRC와 HerkuleX에서는 이 중 더 오차가 적은 쪽으로, 2배속 모드에서 16을 사용하고 있습니다.

H/W 원리

앞서 알아보았듯이, ATmega128에는 USART0, USART1의 두 USART 포트가 있습니다. 이 중에 USART0은 HerkuleX와의 통신에 사용되는 4핀 포트 5개와 병렬적으로 연결되어 있으며, USART1은 이어폰 잭을 통해서 PC와 연결되어 있으며 내부적으로 Zigbee 인터페이스와 연결되어 있습니다.

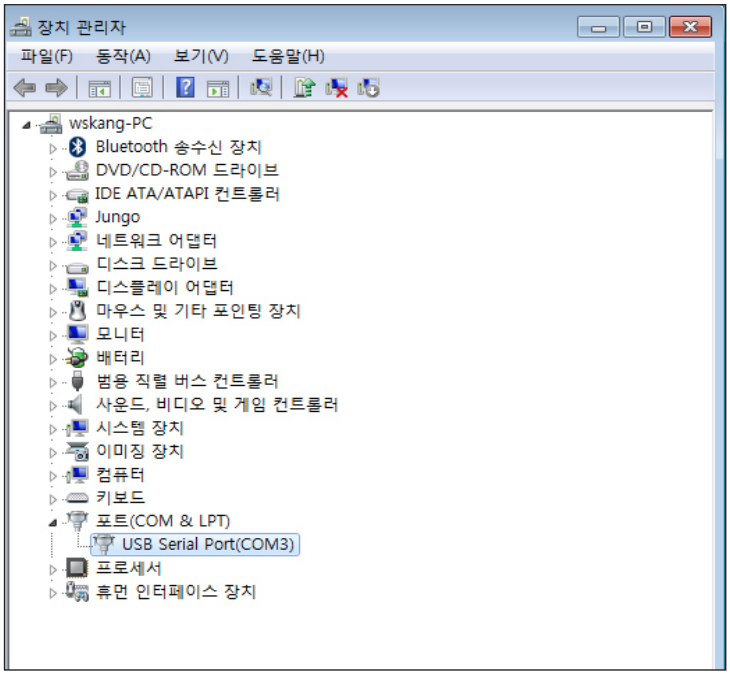
ATmega128에서 내보내는 USART 신호는 디지털 HIGH와 LOW 신호이므로, 신호 레벨은 LOW일 때 0V, HIGH일 때 5V입니다. 이런 레벨을 가진 신호를 TTL 신호라고 합니다. 그런데 보통 PC의 COM 포트나 USB-to-Serial 젠더에서 나오는 신호는 통상적으로 LOW일 때 -5~-10V, HIGH일 때 +5~+10V의 신호 레벨을 가집니다. 이런 신호는 RS-232 레벨에 있다고 이야기 합니다. 비록 똑같은 데이터가 전송되고 있다고 하더라도, 이 신호들을 그대로 연결했다가는 서로의 신호 레벨이 맞지 않아서 통신을 제대로 할 수 없습니다.

그래서 DRC의 회로에는 직렬 통신 레벨 변환 칩을 사용하여 ATmega128이 보내는 신호를 TTL에서 RS-232로 바꾸어 PC에 전송하고, PC에서 보내는 신호를 RS-232에서 TTL로 바꾸어 ATmega128로 전송합니다.

하드웨어 준비하기

01 연결

DRC 제어기 아답터로 전원을 연결하고, USB to Serial Cable 로 PC 와 DRC 를 연결합니다.

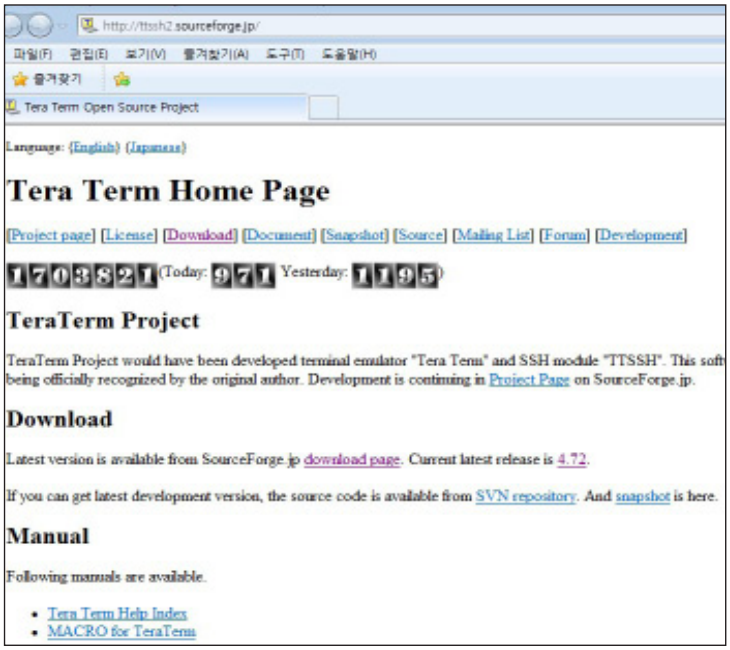


02 COM port 연결

내컴퓨터 〉 속성 〉 장치관리자에서 COM 연결 확인합니다.

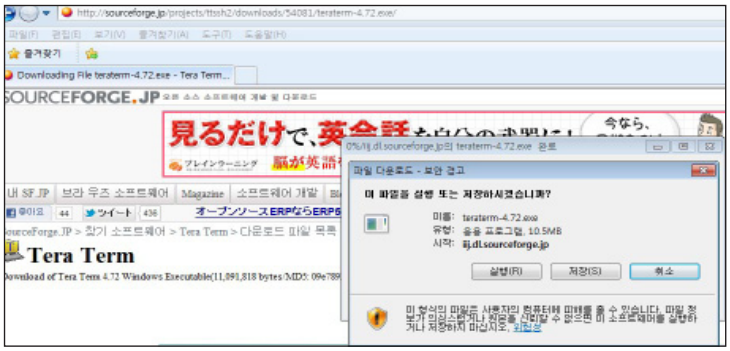
01 주소 검색

인터넷 포털 사이트에서 “tera term”을 검색합니다.



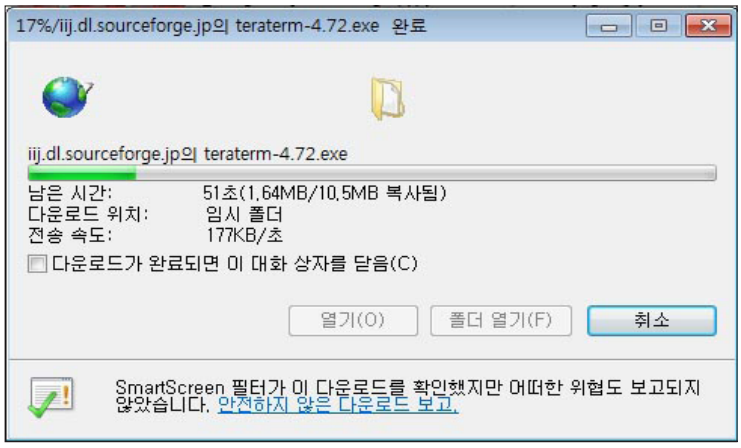
02 홈페이지 접속

Tera Term 홈페이지에 접속하여 “download page”를 클릭합니다.



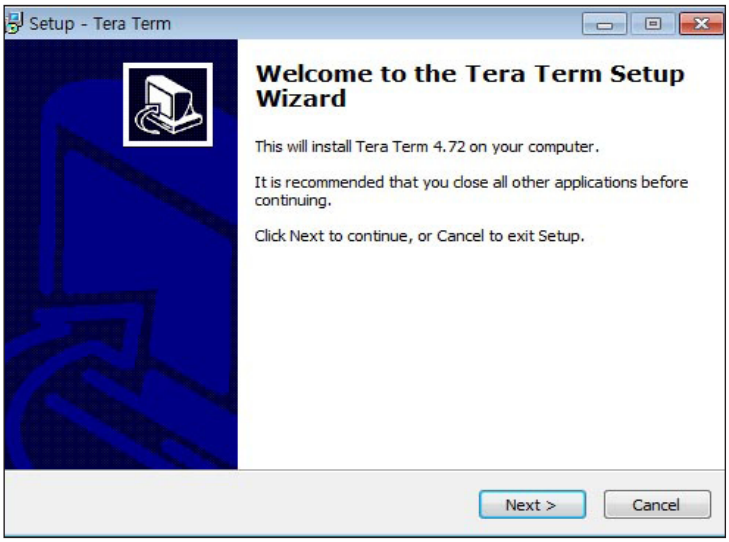
03 파일 다운로드

“저장”을 눌러 파일을 다운로드 받습니다.



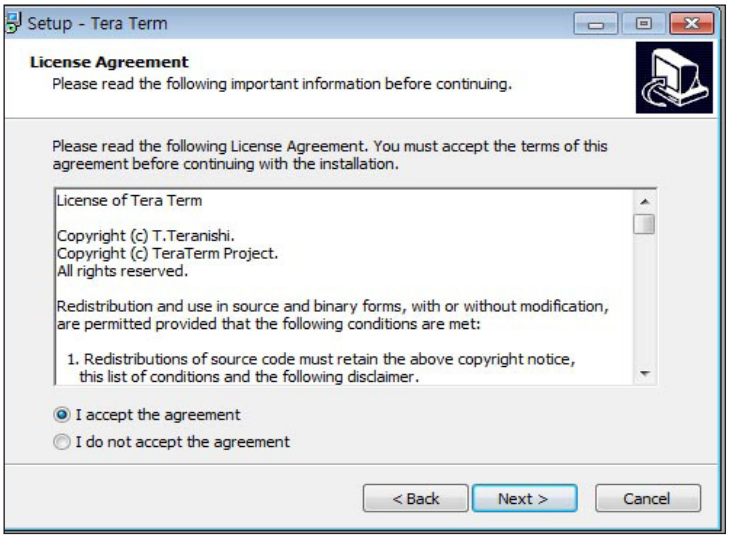
04 파일 다운로드 시작

설치 파일이 다운로드됩니다.



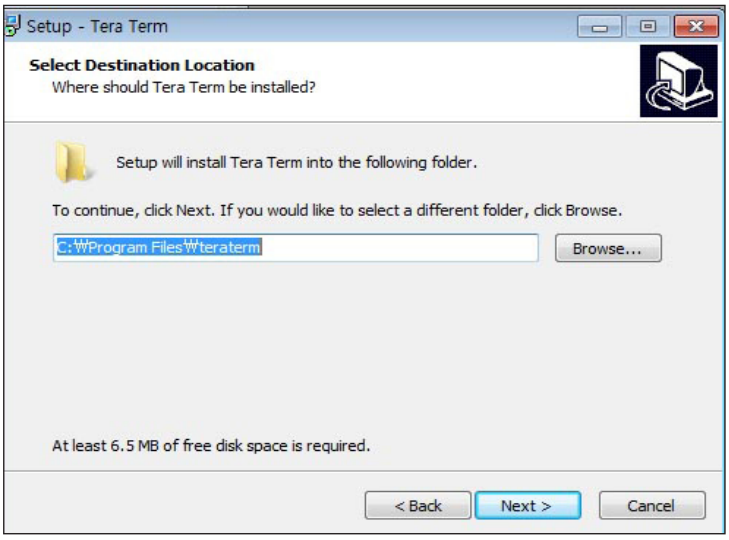
05 설치 마법사 실행

"Next"를 누르면 설치 마법사가 실행됩니다.



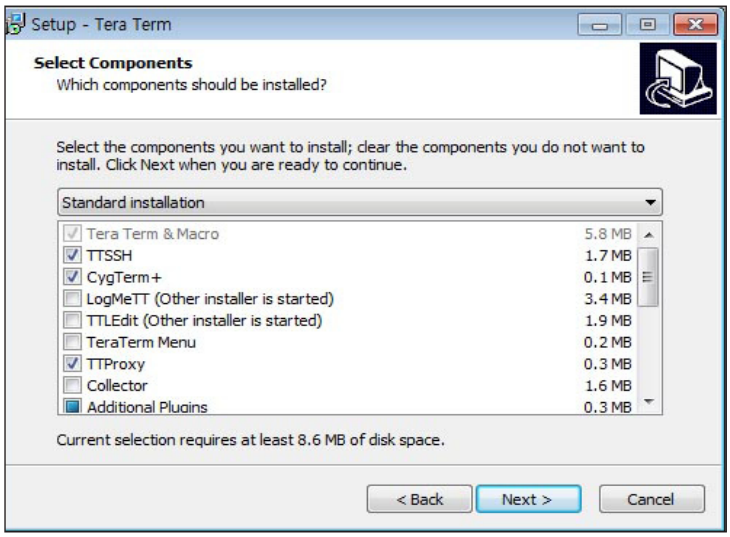
06 약관 동의

"I accept the agreement"를 체크하고 "Next"를 클릭합니다.



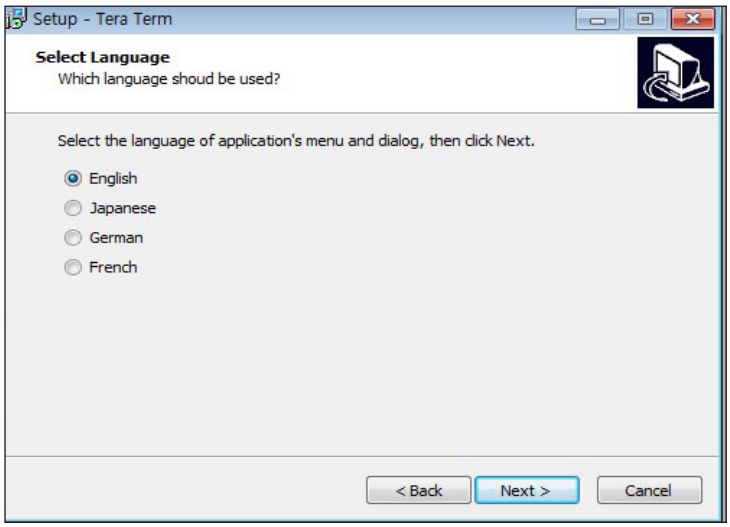
07 설치 폴더 확인

프로그램을 설치할 폴더를 확인하고 "Next"를 클릭합니다.



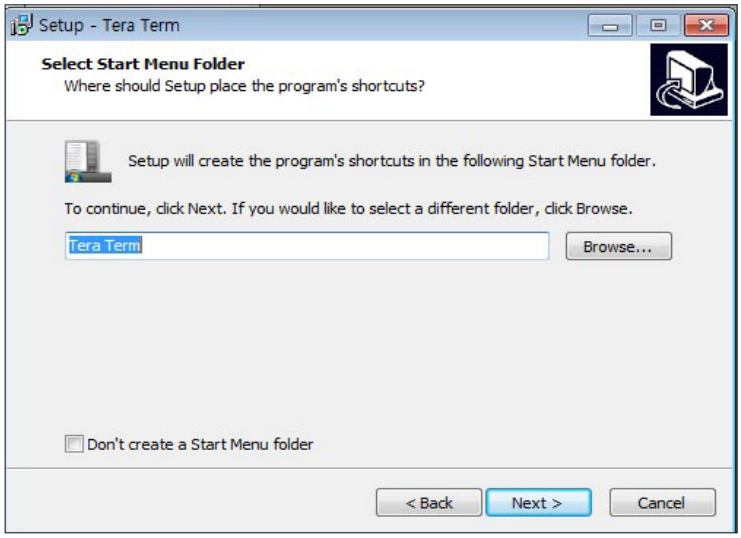
08 구성요소 선택

"Next"를 클릭합니다.



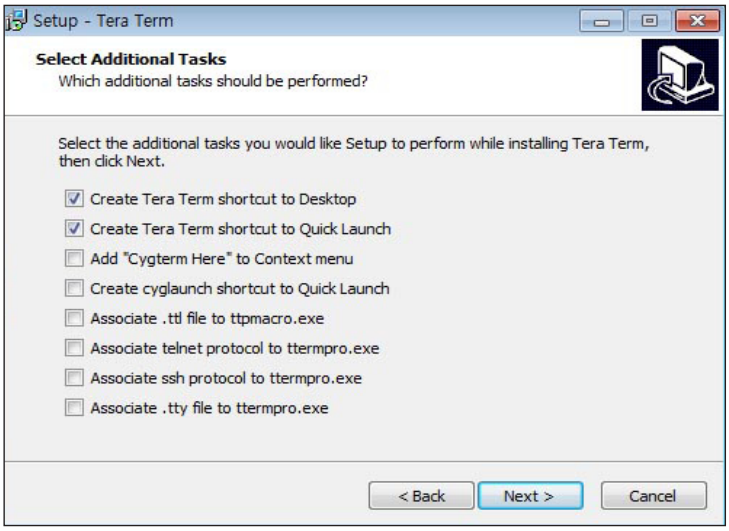
09 언어 선택

언어 선택 후 "Next"를 클릭합니다.



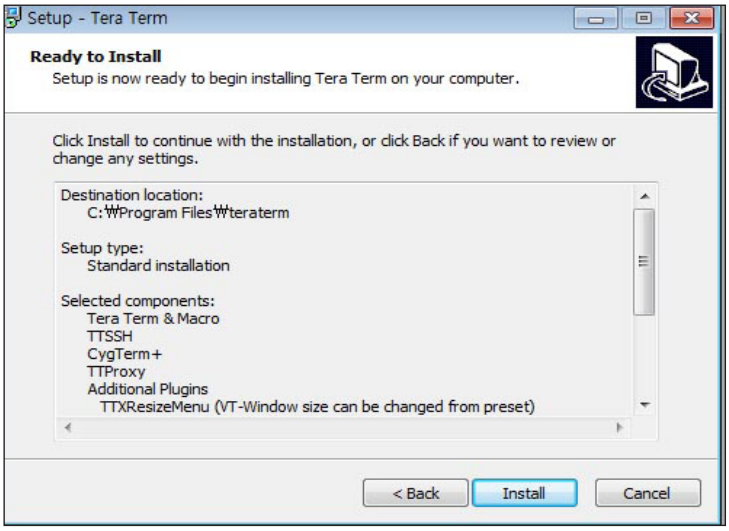
10 시작 메뉴 폴더 선택

시작 메뉴 폴더 선택 후 “Next”를 클릭합니다.



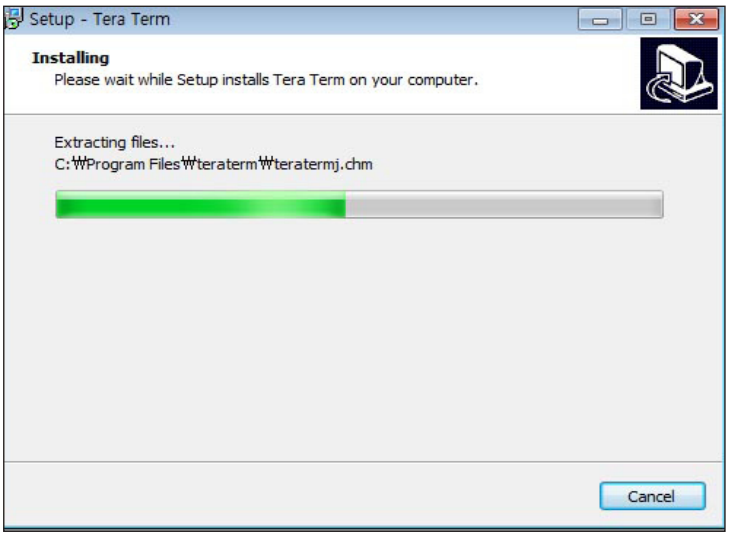
11 추가 기능 선택

원하는 추가 기능을 선택 후 “Next”를 클릭합니다.



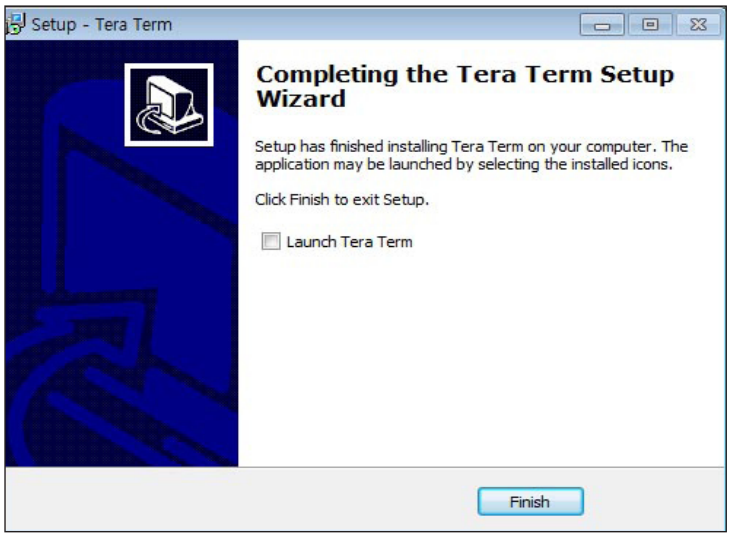
12 설치 시작

“Install”을 클릭하면 설치가 시작됩니다.



13 설치 확인

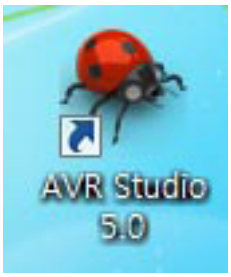
프로그램이 설치됩니다.



14 설치 완료

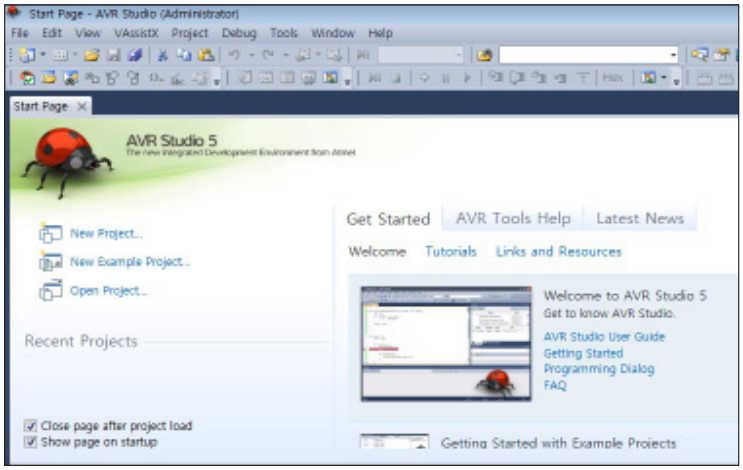
"Finish"를 클릭하면 설치가 완료됩니다.

AVR Studio 실행 및 코딩 준비



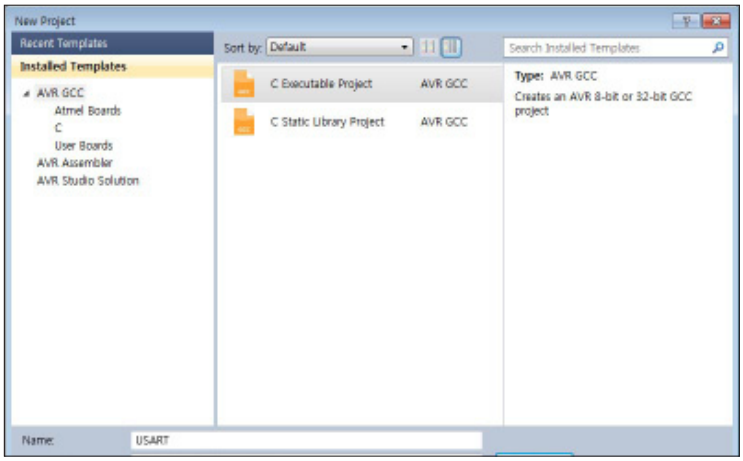
01 실행

AVR Studio 아이콘을 클릭하여 실행합니다.



02 New Project

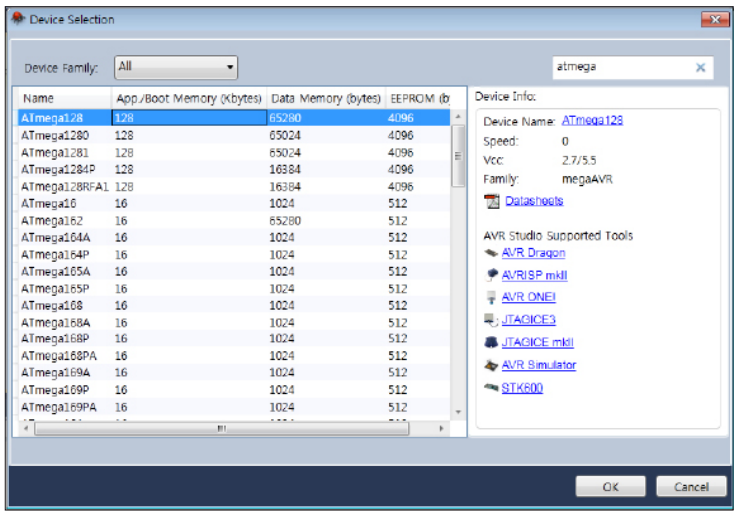
New Project 를 클릭합니다.



03 GCC 선택

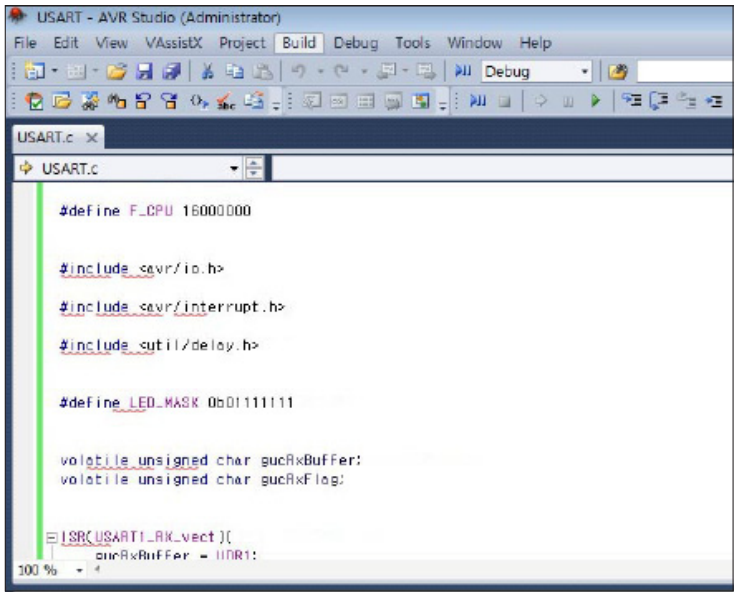
C Executable Project 를 선택합니다.

Name 을 “USART”로 입력하고, Create directory for solution 을 클릭하고, OK 버튼을 누릅니다.



04 ATmega 선택

우측 상단에 atmega 를 검색하여, 리스트중에 ATmega128 을 선택합니다.



05 Editor

편집창에 코딩을 시작합니다.

아래 예제를 따라하세요.

예제

USB-to-Serial 혹은 PC 뒷면의 COMPORT와 DRC를 통신 케이블로 연결하고, 하이퍼터미널을 이용해서 보내는 글자에 따라서 DRC의 LED를 제어하는 예제입니다.

USART.c

```
//CPU 클럭 설정
#define F_CPU 16000000

//ATmega128의 레지스터 등이 정의되어 있음
#include <avr/io.h>
//interrupt를 사용하는 데 필요한 정보들이 정의되어 있음
#include <avr/interrupt.h>
//_delay_ms 등 딜레이 함수가 정의되어 있음
#include <util/delay.h>

//LED에 해당하는 포트 C의 핀을 정의
#define LED_MASK 0b01111111

//USART1에서 데이터를 받을 때 사용할 전역변수 선언
volatile unsigned char gucRxBuffer;
volatile unsigned char gucRxFlag;

/**
* USART1 수신 완료 인터럽트의 서비스 루틴이다.
* UDR1으로부터 데이터를 읽어 gucRxBuffer에 저장하고
* gucRxFlag를 1로 설정한다.
*
* @author       Dongbu Robot
* @param        없음
* @return       없음
* @version      1.0.0.0
*/
ISR(USART1_RX_vect){
        gucRxBuffer = UDR1;
        gucRxFlag = 1;
}

/**
* USART1의 초기화 함수다. USART1을 활성화하고
* ulBaudRate에 따라서 보드 레이트를 설정한 후
* 전역 변수를 초기화 한다.
*
* @author       Dongbu Robot
* @param        ulBaudRate: USART1을 작동시킬 보드 레이트
* @return       없음
* @version      1.0.0.0
*/
void USART1_Init(unsigned long ulBaudRate){
        unsigned long ulTemp;

        //부트로더와의 충돌을 막기 위해 잠시 대기
        _delay_ms(10);
```

```
        //송신, 수신, 수신 인터럽트 활성화
        UCSR1B = ((1 << TXEN1) | (1 << RXEN1) | (1 << RXCIE1));
        //2배속 전송 모드 설정
        UCSR1A = (1 << U2X1);

        //클럭과 Baud Rate을 사용해 UBRR1에 들어갈 값 계산
        ulTemp = F_CPU / (ulBaudRate * 8L) - 1;

        //UBRR1에 계산한 값 설정
        UBRR1H = (ulTemp >> 8) & 0x0F;
        UBRR1L = ulTemp & 0xFF;

        //전역 변수 초기화
        gucRxBuffer = 0;
        gucRxFlag = 0;

        return;
}

//USART1로 문자 하나를 전송하는 함수
void USART1_PutChar(char cData){
        //송신 데이터 레지스터가 빌 때 까지 대기
        while(!(UCSR1A & (1 << UDRE1)));
        //송신 데이터 레지스터에 데이터 입력
        UDR1 = cData;

        return;
}

//USART1로 문자열을 전송하는 함수
void USART1_PutString(char* pcStr){
        //₩'0'을 만날 때 까지 USART1_PutChar를 반복
        while(*pcStr)
                USART1_PutChar(*pcStr++);
}

int main(void)
{
        //8비트 변수 선언
        unsigned char ucLED;

        //포트 C의 입출력 방향 설정
        DDRC = 0b01111111;
        //포트 C의 출력 값 설정
        PORTC = 0b01111111;

        //전체 인터럽트를 비활성화
        cli();

        //USART1 초기화
        USART1_Init(115200);
```

```
        //전체 인터럽트를 활성화
        sei();

        //초기화 완료 메시지 전송
        USART1_PutString("Serial Initialized₩r₩n");

        //Servo LED를 켬
        ucLED = 0b00000001;
        PORTC = (~ucLED) & LED_MASK;

        //while 안의 내용을 무한 반복
        while(1)
        {
                //데이터를 받았다면 실행
                if(gucRxFlag){
                        //받은 데이터가 'z'나 'Z'라면 실행
                        if(gucRxBuffer == 'z' || gucRxBuffer == 'Z'){
                                //켤 LED 비트를 오른쪽으로 이동
                                if(ucLED > 0b00000001){
                                        ucLED >>= 1;
                                }
                                else{
                                        ucLED = 0b01000000;
                                }
                                PORTC = (~ucLED) & LED_MASK;

                                //메시지 전송
                                USART1_PutString("LED Moved by ₩'z₩'₩r₩n");
                        }
                        //받은 데이터가 'x'나 'X'라면 실행
                        else if(gucRxBuffer == 'x' || gucRxBuffer == 'X'){
                                //켤 LED 비트를 왼쪽으로 이동
                                if(ucLED < 0b01000000){
                                        ucLED <<= 1;
                                }
                                else{
                                        ucLED = 0b00000001;
                                }
                                PORTC = (~ucLED) & LED_MASK;

                                //메시지 전송
                                USART1_PutString("LED Moved by ₩'x₩'₩r₩n");
                        }
                        //데이터 수신 플래그 초기화
                        gucRxFlag = 0;
                }
        }

        //함수의 형태에 맞게 정수값 반환
        return 1;
}
```

예제설명

```
#define F_CPU 16000000
```

#define 문은 코드 전처리 명령 중 하나로, 앞으로 F_CPU라는 값이 나오면 16000000으로 대체한다는 뜻입니다. DRC의 ATmega128은 16MHz의 클럭 속도로 동작합니다. 클럭 속도에 비례해서 명령 처리 속도가 달라지므로, 이 명령을 통해서 이 프로그램이 동작할 속도를 명시해 주어야 delay 함수 등이 올바르게 동작합니다.

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

avr/io.h 헤더 파일에는 DDRX, PORTX 레지스터 등 AVR 제어를 위해 필요한 레지스터 등이 정의되어 있고, avr/interrupt.h 헤더 파일에는 인터럽트를 사용하기 위한 함수들이 정의되어 있습니다. util/delay.h 헤더 파일에는 _delay_ms()등 자주 사용하는 딜레이 함수가 정의되어 있으므로 #include 문을 사용하여 포함시켜 줍니다.

```
#define LED_MASK                    0b01111111
```

#define LED_MASK 0b01111111은 앞으로 LED_MASK라는 것이 나오면 0b01111111로 대체해서 컴파일한다는 뜻입니다. DRC의 LED 전체에 해당하는 비트들을 LED_MASK라는 이름으로 대체해서 프로그램을 작성하기 쉽게 하려고 선언합니다.

```
volatile unsigned char gucRxBuffer;
volatile unsigned char gucRxFlag;
```

인터럽트 서비스 루틴과 함수들에서 공통으로 사용할 전역변수를 선언합니다. gucRxBuffer는 받은 USART1 데이터 레지스터 값을 저장할 변수이며, gucRxFlag는 받은 데이터가 있음을 알리는 플래그 변수입니다.

```
ISR(USART1_RX_vect){
        gucRxBuffer = UDR1;
        gucRxFlag = 1;
}
```

USART1 수신 완료 인터럽트의 인터럽트 서비스 루틴입니다. UDR1로부터 받은 데이터를 읽어와 gucRxBuffer에 저장하고, gucRxFlag를 1로 만들어줍니다. 이 데이터와 플래그는 메인 함수에서 사용될 것 입니다.

```
void USART1_Init(unsigned long ulBaudRate)
```

PC와의 통신에 사용할 USART1을 초기화하는 함수입니다. 보드 레이트를 ulBaudRate라는 파라미터로 받으며, ulBaudRate의 값을 사용해 UBRR1에 들어갈 값을 계산합니다.

```
unsigned long ulTemp;
```

ulTemp라는 변수를 선언합니다. ulBaudRate로 계산한 UBRR 값을 저장할 변수입니다.

```
_delay_ms(10);
```

DRC의 부트로더에서도 USART1을 사용해서 펌웨어를 퓨징하기 때문에, 부트로더에서 통신 중에 데이터를 미처 다 보내지 못했는데 펌웨어에서 USART1의 설정을 바꿀 경우 통신 에러가 날 수 있습니다. 따라서 부트로더에서 보낸 데이터를 마저 보낼 수 있도록 10ms 정도의 시간을 기다립니다.

```
UCSR1B = ((1 << TXEN1) | (1 << RXEN1) | (1 << RXCIE1));
UCSR1A = (1 << U2X1);
```

UCSR1B와 UCSR1A 레지스터를 설정합니다. 송신과 수신, 수신 인터럽트를 활성화합니다. 또한 2배속 비동기 모드를 사용할 것이므로 U2X1을 1로 만들어 줍니다. UCSR1C 레지스터는 초기값 그대로 사용하면 되므로 따로 설정하지 않습니다.

```
ulTemp = F_CPU / (ulBaudRate * 8L) - 1;

UBRR1H = (ulTemp >> 8) & 0x0F;
UBRR1L = ulTemp & 0xFF;
```

ulBaudRate를 사용해 UBRR1 값을 계산하고 ulTemp에 저장합니다. 계산식은 위에서 나온 $UBRR = \frac{f_{OSC}}{8BAUD} - 1$ 를 사용합니다. ulTemp 값의 상위 4바이트와 하위 8바이트를 각각 UBRR1H와 UBRR1L에 저장합니다. 이제 USART1은 입력한 ulBaudRate의 보드 레이트로 동작할 것입니다.

```
gucRxBuffer = 0;
gucRxFlag = 0;
```

전역 변수를 0으로 초기화 합니다.

```
void USART1_PutChar(char cData){
        while(!(UCSR1A & (1 << UDRE1)));
        UDR1 = cData;
        return;
}
```

USART1_PutChar는 USART1을 사용해 1바이트 데이터를 송신하는 함수입니다. 데이터 송신은 UCSR1A의 UDRE1 비트가 1이어야만 할 수 있으므로, 다른 데이터를 이미 보내고 있을 때에는 UDRE1 비트가 1이 될 때까지 기다려야 합니다. while(!(UCSR1A & (1 << UDRE1)));으로 UDRE1이 1이 될 때까지 기다리고, 그 후에는 파라미터로 받은 cData를 UDR1에 넣음으로써 새로 cData를 전송합니다.

```
void USART1_PutString(char* pcStr){
        while(*pcStr)
                USART1_PutChar(*pcStr++);
}
```

USART1_PutString()은 USART1을 이용해 문자열을 보내는 함수입니다. 문자열의 끝은 '₩0'(NULL 문자)으로 끝나고 그 값은 0이므로, while(*pcStr)문에서는 USART1_PutChar()를 이용해 pcStr이 가리키는 데이터를 보내고, pcStr의 값을 1씩 증가시키면서 *pcStr이 NULL 문자가 될 때까지 반복하게 됩니다.

```
unsigned char ucLED;

DDRC = 0b01111111;
PORTC = 0b01111111;
```

LED의 출력 값이 저장될 8비트 변수 ucLED를 선언하고, 포트 C의 0~6번 핀을 출력으로 설정하고 출력 값을 1로 만듭니다. 그러면 모든 LED가 꺼지게 됩니다.

```
cli();
USART1_Init(115200);
sei();
```

인터럽트에 대한 설정을 바꾸는 중에 인터럽트가 발생해 예상치 못한 오동작이 발생하는 것을 방지하기 위해서 우선 cli() 명령을 통해서 전체 인터럽트를 비활성화 시킵니다. 그 후 USART1_Init() 함수를 사용해서 USART1을 초기화합니다.
사용할 보드 레이트가 115200이므로 파라미터로 그 값을 사용합니다. USART1의 초기화가 끝난 후에는 sei() 명령을 통해서 전체 인터럽트를 활성화 합니다.

```
USART1_PutString("Serial Initialized₩r₩n");

ucLED = 0b00000001;
PORTC = (~ucLED) & LED_MASK;
```

USART1_PutString()을 사용해서 시리얼 통신이 초기화 되었다는 것을 알립니다. 문자열 끝의 ₩r₩n은 줄을 바꾸기 위한 문자들입니다. ₩r은 Carriage Return으로 커서를 줄의 처음으로 돌리는 역할을 하고, ₩n은 Newline으로 줄을 바꿀 때 사용합니다.

ucLED를 0번 비트만 1로 만들어 Servo LED만 켜는 것으로 설정합니다. 그 후 PORTC의 값을 바꾸어 실제로 ucLED에 설정된 LED를 켭니다. 실제 LED가 active low 방식으로 작동해 비트가 0이 되어야 LED가 켜지기 때문에, ucLED를 ~(bitwise NOT)연산해 비트들을 토글한 결과에 LED_MASK를 &(bitwise AND)연산한 값을 PORTC에 입력합니다.

```
if(gucRxFlag){
        …
        gucRxFlag = 0;
}
```

while문 안에는 if문이 하나 있습니다. 이 if문은 gucRxFlag가 0이 아닐 때 실행되고, if문이 끝날 때에는 다시 gucRxFlag를 0으로 초기화 시킵니다. 평소에는 gucRxFlag가 0이므로 while문은 아무 일도 하지 않고 계속 반복합니다. 그러던 중 USART1로 데이터가 수신되면, USART1 수신 완료 인터럽트의 서비스 루틴에서는 데이터를 gucRxBuffer에 읽어오고 gucRxFlag를 1로 설정합니다. 인터럽트가 끝나고 다시 if문을 만났을 때에는 gucRxFlag가 1이기 때문에 프로그램은 if문 안으로 진입해서 루틴을 실행합니다. 그리고 그 후에는 gucRxFlag를 다시 0으로 초기화 시킵니다. 이렇게 if문을 작성하면, if문 안의 내용을 USART1 수신 완료 인터럽트가 일어날 때마다 한번씩 실행할 수 있게 됩니다.

```
If(gucRxBuffer == 'z' || gucRxBuffer == 'Z'){
        if(ucLED > 0b00000001){
                ucLED >>= 1;
        }
        else{
                ucLED = 0b01000000;
        }
        PORTC = (~ucLED) & LED_MASK;

        USART1_PutString("LED Moved by ₩'z₩'₩r₩n");
}
```

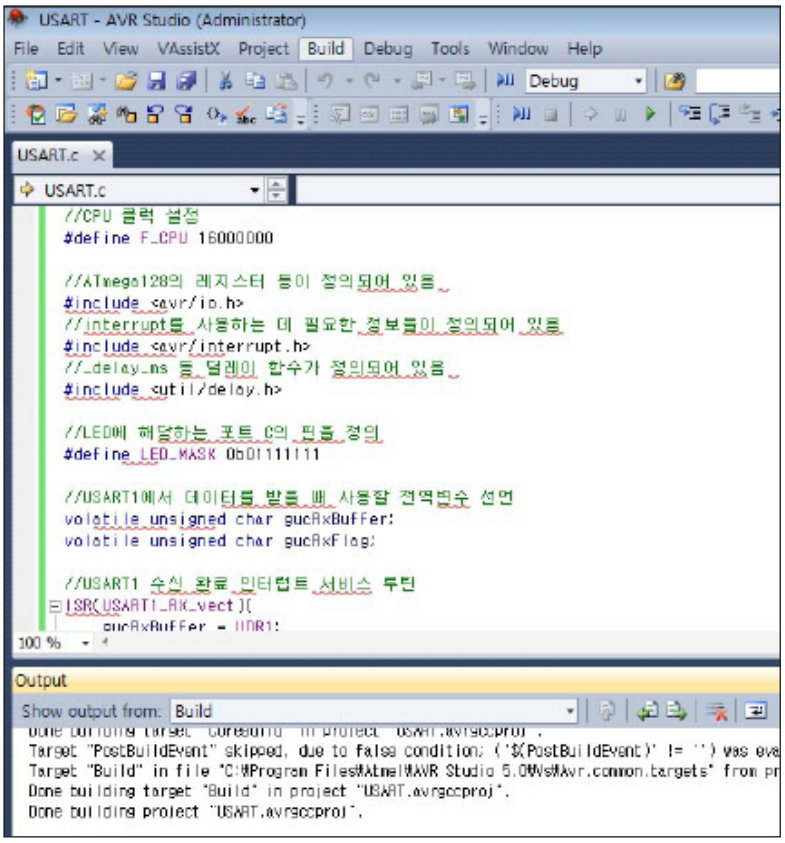
gucRxFlag가 1이라서 if문에 진입한 후에는 위의 코드를 만나게 됩니다. gucRxBuffer에 저장된 방금 들어온 데이터 값이 'z'이거나 'Z'인 경우에는 ucLED에 저장된 비트를 오른쪽으로 한번 이동합니다. 만약 ucLED 값이 0b00000001이라서 더 이동할 공간이 없다면 제일 왼쪽인 0b01000000으로 바꾸어 줍니다. 그 후에는 PORTC에 ucLED에 해당하는 값을 넣어 실제로 LED를 켭니다. LED를 켜고 나면 USART1_PutString() 함수를 사용해 'z' 키에 의해 LED가 움직였다는 것을 출력합니다.

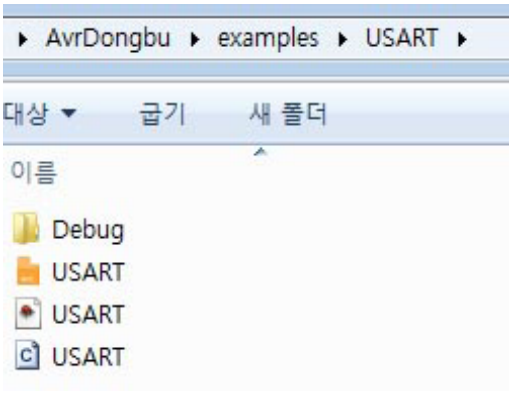
```
else if(gucRxBuffer == 'x' || gucRxBuffer == 'X'){
        if(ucLED < 0b01000000){
                ucLED <<= 1;
        }
        else{
                ucLED = 0b00000001;
        }
        PORTC = (~ucLED) & LED_MASK;

        USART1_PutString("LED Moved by ₩'x₩'₩r₩n");
}
```

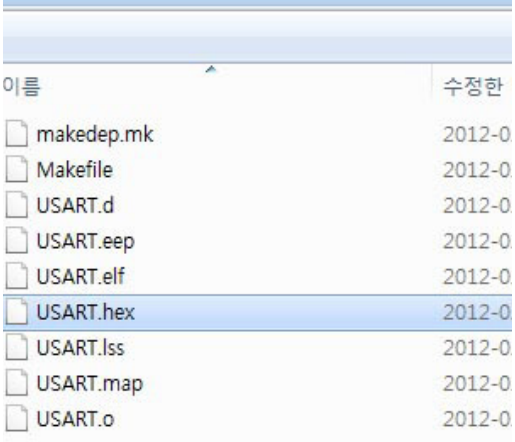
만약 입력된 데이터가 'z'나 'Z'가 아니라면 위의 코드를 만납니다. 데이터 값이 'x'이거나 'X'인 경우에는 'z'일 때와는 반대로 ucLED의 비트를 왼쪽으로 이동합니다. 만약 ucLED 값이 0b01000000이라서 더 이동할 공간이 없으면 제일 오른쪽인 0b00000001으로 바꿉니다. 그 후에는 PORTC의 값을 바꾸어 LED를 켜고, 'x' 키에 의해 LED가 움직였다는 것을 출력합니다. If–else 문은 여기서 끝나므로, 만약 'z', 'Z', 'x', 'X'가 아닌 다른 입력이 들어왔다면 아무 행동도 하지 않고 gucRxFlag만 0으로 초기화합니다.

실행

펌웨어를 실행하고 Term Term을 이용해 z, x키를 입력하면 켜진 LED가 이동하면서 터미널 창에 메시지가 출력되는 것을 확인할 수 있습니다..







01 Build

Build 메뉴를 선택하여 Build Solution을 클릭하여 Build 시킵니다.

1 succeeded 0 failed 가 나오면 에러없이 컴파일 완료됐다는 것을 의미합니다.

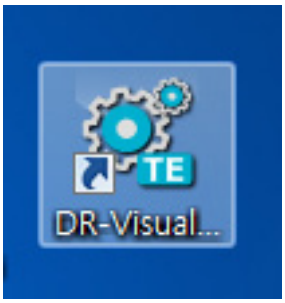
02 Debug 폴더

Build 후에 소스가 있는 폴더에 들어가면 Debug 폴더가 생성 되어있습니다. 클릭하여 폴더 안의 내용을 봅니다.

03 hex 확인

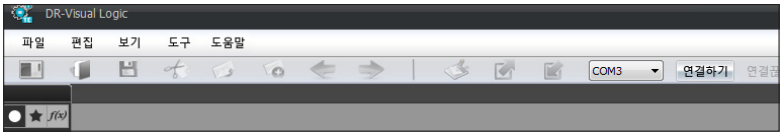
Debug 폴더 안의 hex 파일 확인합니다. Hex 파일이 제어기에 포팅시키는 파일입니다.

DR-Visual Logic 에서 펌웨어 복구를 실시합니다.



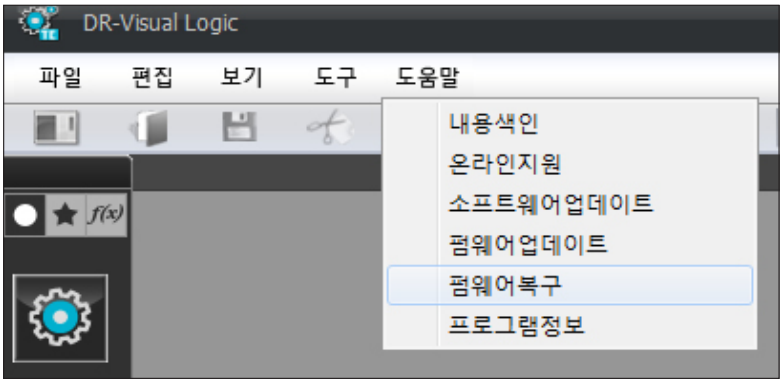
04 Visual Logic

DR-Visual Logic 을 실행시킵니다.



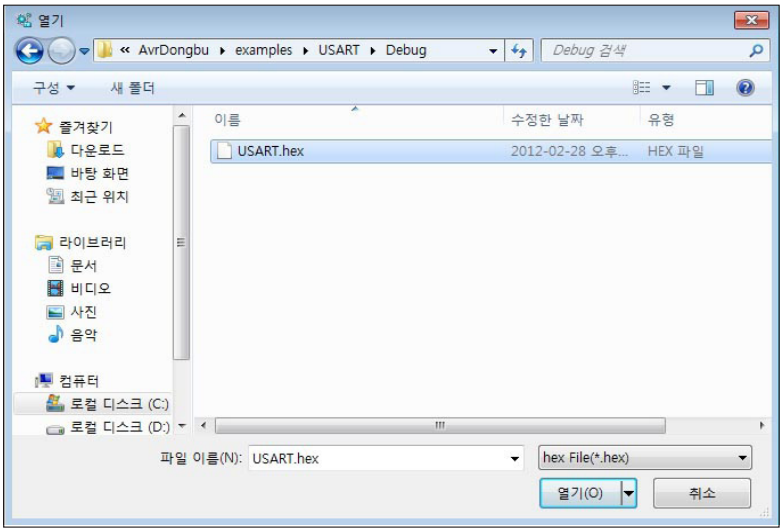
05 연결하기

COM 번호를 설정하고 연결하기를 클릭합니다.



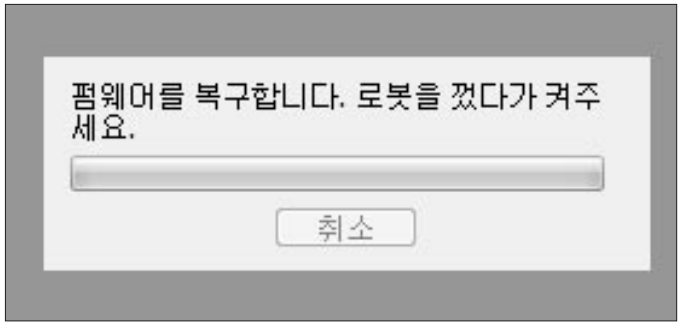
06 펌웨어 복구

도움말 메뉴에서 펌웨어 복구를 클릭합니다.



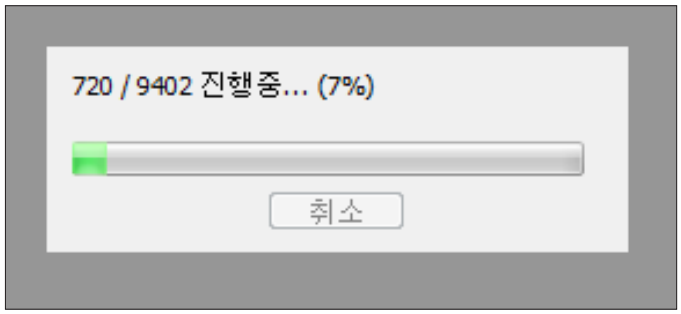
07 불러오기

Debug 폴더에서 Hex 파일을 불러옵니다



08 펌웨어복구

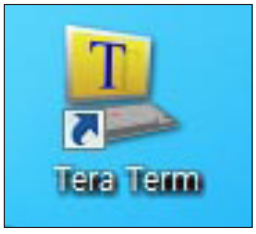
로봇을 껐다가 켜주세요 라는 문구가 나옵니다.
제어기의 전원버튼을 한번 껐다가 다시 켜주세요



09 복구시작

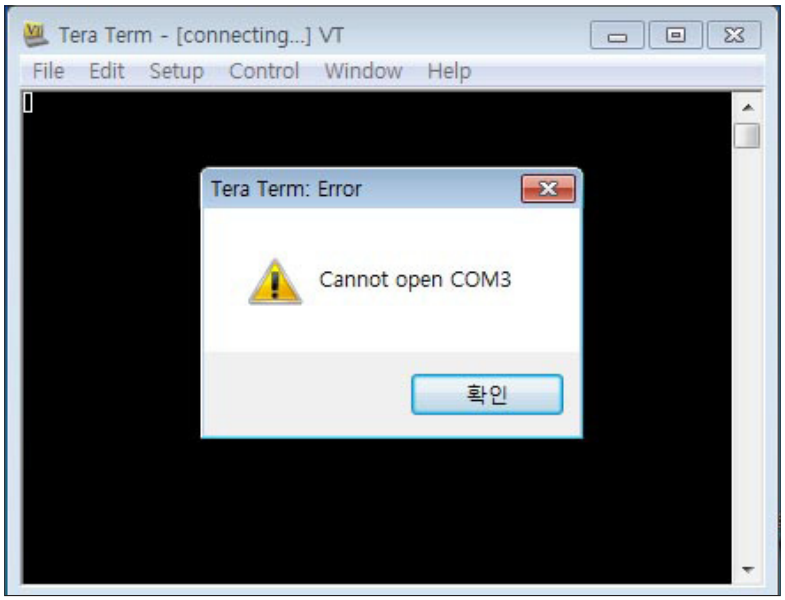
펌웨어 복구가 시작됩니다. 진행이 완료되면 팝업창이 내려갑니다.

Tera Term 을 활용하여 통신 테스트를 실시합니다.



10 실행

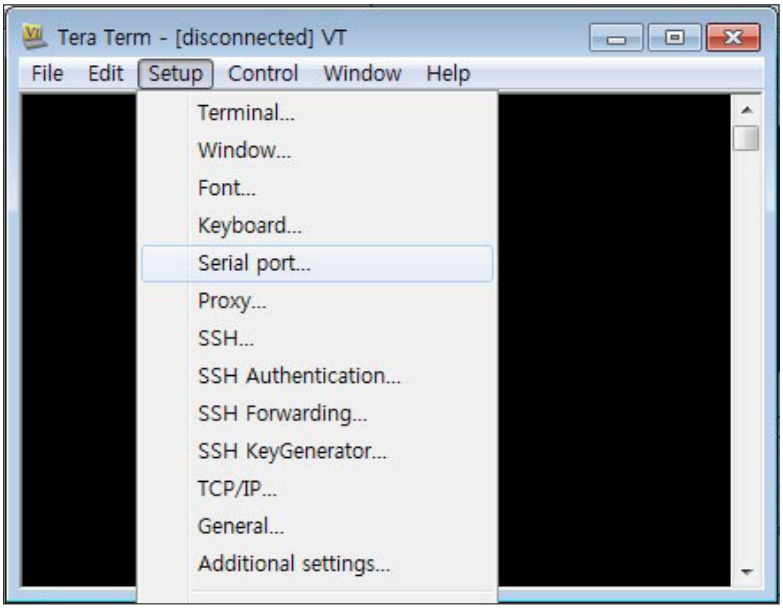
Tera Term 아이콘을 클릭하여 프로그램을 실행시킵니다.



11 열기

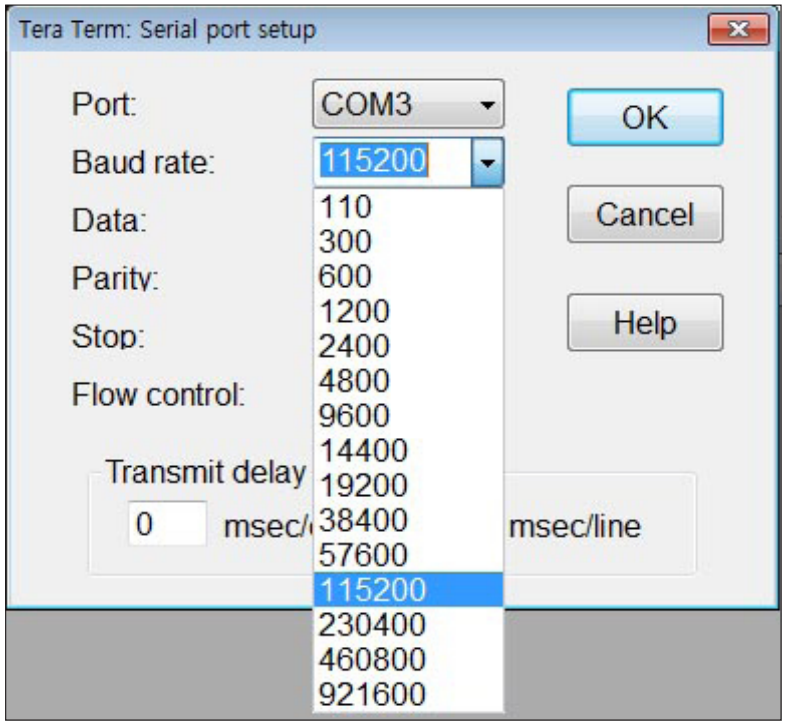
실행하면 바로 연결되지 않습니다.
확인버튼을 클릭합니다.

※바로 연결 될 수도 있습니다.
여기서 연결되지 않은 이유는
DR-Visual Logic 에서 연결끊기를
안눌렀기 때문입니다.



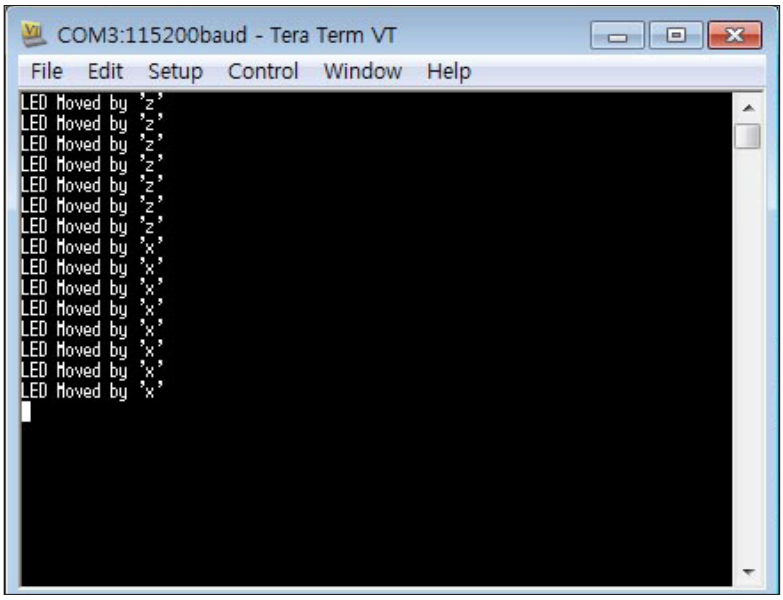
12 Set up

메뉴중에 Setup 메뉴를 클릭하여 Serial port 를 선택합니다.



13 Baud rate

Baud rate 를 115200 으로 선택합니다. 제어기 기본 통신속도입니다.



14 실행

펌웨어를 실행하고 Tera Term(하이퍼터미널)을 이용해 z, x키를 입력하면 켜진 LED가 이동하면서 터미널 창에 메시지가 출력되는 것을 확인할 수 있습니다.

HerkuleX 제어하기

herkulex 라이브러리를 이용해서 HerkuleX에 직접 패킷을 보내고 제어해 봅니다.

H/W 원리

DRC에 장착된 ATmega128에는 USART0, USART1의 두 USART 포트가 있으며 이 중 USART0은 HerkuleX와의 통신에 사용되는 4핀 포트 5개와 병렬적으로 연결되어 있습니다. 제어하고 싶은 HerkuleX를 케이블로 DRC와 연결하면 USART0을 통해서 직접 패킷을 보내고 받을 수 있습니다.

이번 Chapter에서는 HerkuleX와 통신을 하기 때문에 HerkuleX의 레지스터 맵과 프로토콜에 대한 어느 정도의 지식이 필요합니다. 이번 예제에서 사용하는 패킷에 사용된 명령을 요약하면 아래와 같습니다.

이름	CMD	설명
RAM_WRITE	0x03	HerkuleX의 휘발성 레지스터 영역에 데이터를 씁니다.
RAM_READ	0x04	HerkuleX의 휘발성 레지스터 영역의 데이터를 읽습니다.
I_JOG	0x05	HerkuleX의 위치/속도를 제어하여 움직이게 합니다. HerkuleX의 위치/속도를 제어하여 움직이게 합니다. 각 모터마다 이동 시간을 다르게 설정할 수 있습니다. I_JOG와 S_JOG 패킷에서 시간을 나타내기 위해서는 1바이트 크기의 데이터를 사용하는데, 이때 데이터는 1당 11.2ms를 나타냅니다. 즉 I_JOG 패킷에 50의 시간 데이터가 들어가 있다면 실제로 모터 50*11.2=560ms동안 목표 위치로 이동합니다.

이번 예제에서 사용된 휘발성 레지스터 영역의 항목을 요약하면 아래와 같습니다.

이름	주소	크기	설명
Torque Control	52	1	토크 인가 상태를 제어합니다. 0x00 : 자유롭게 움직이는 힘이 풀린 상태 0x40 : 힘은 풀렸으나 돌아가는 데에 저항이 있는 상태 0x60 : 토크가 인가되어 스스로 구동하는 상태
Calibrated Position	58	2	현재 위치와 GPIO 입력 결과를 나타냅니다. 여기서 GPIO란 HerkuleX의 일부 모델에서 오른쪽 통신 포트를 스위치 입력 포트로 사용하는데, 이 때의 입력 값입니다. 15번 비트 : 사용하지 않음 14:13번 비트 : GPIO 입력 값을 나타냅니다. 12:0번 비트 : 현재 HerkuleX의 위치를 나타냅니다.

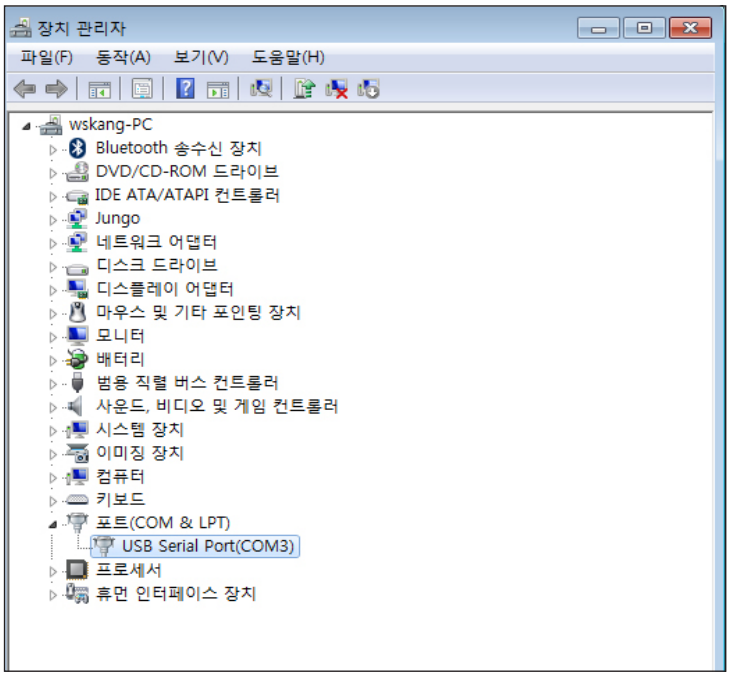
프로토콜과 레지스터 맵에 대한 더 자세한 내용은 HerkuleX 매뉴얼을 참조하세요.

하드웨어 준비하기

01 모터 준비

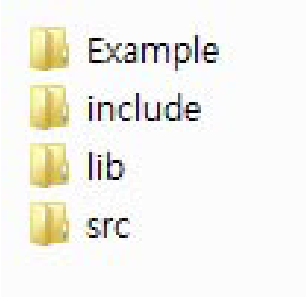
DRC 제어기 아답터로 전원을 연결하고, USB to Serial Cable 로 PC 와 DRC 를 연결합니다.

서보모터를 제어기 우측 포트에 연결합니다.



02 COM port 연결

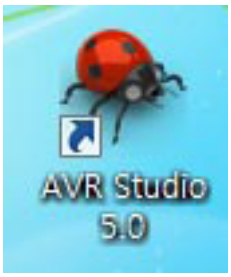
내컴퓨터 〉 속성 〉 장치관리자에서 COM 연결 확인합니다.



AVR Studio 실행 및 코딩 준비

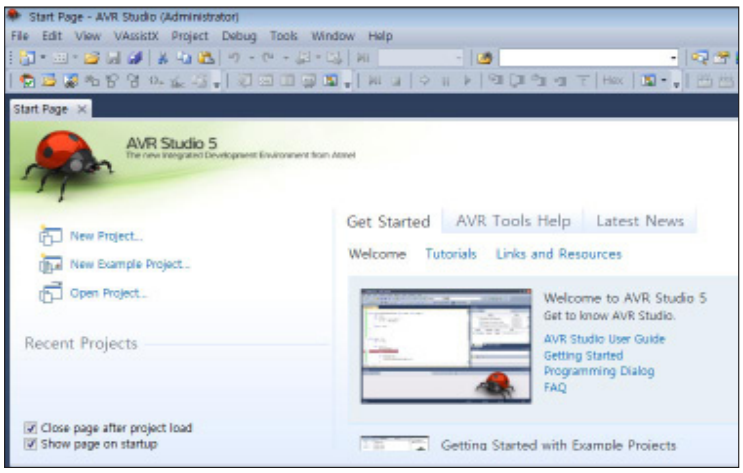
00 라이브러리

서보모터를 제어하기 위해서는 기본 라이브러리 파일이 필요합니다.
라이브러리가 포함된 폴더 구조입니다.



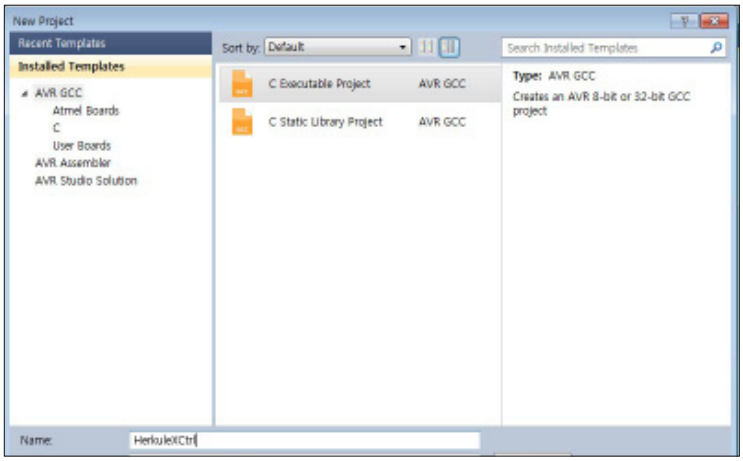
01 실행

AVR Studio 아이콘을 클릭하여 실행합니다.



02 New Project

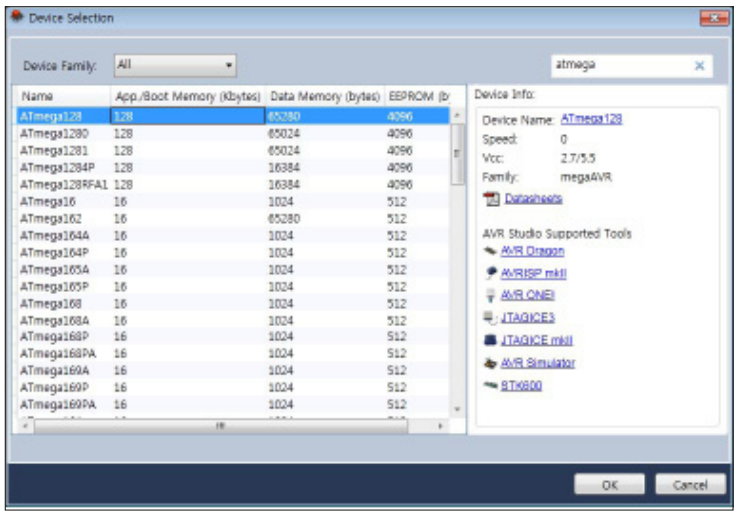
New Project 를 클릭합니다.



03 GCC 선택

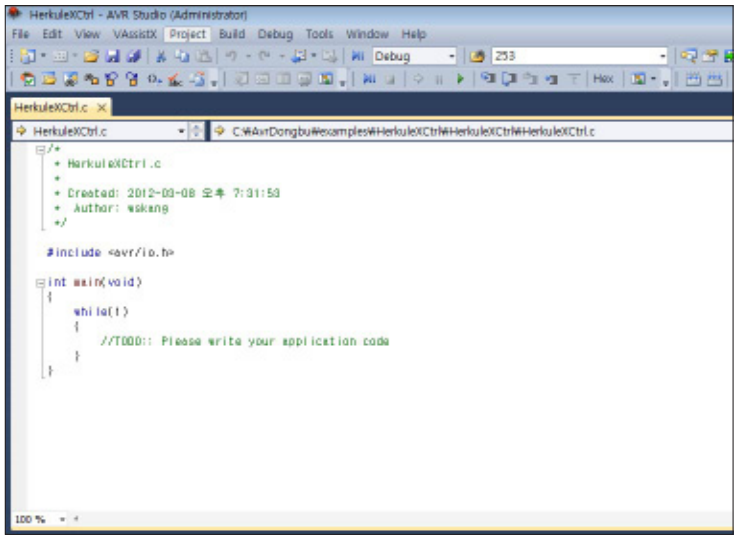
C Executable Project 를 선택합니다.

Name 을 “HerkuleX Ctrl”로 입력하고, Create directory for solution 을 클릭하고, OK 버튼을 누릅니다.



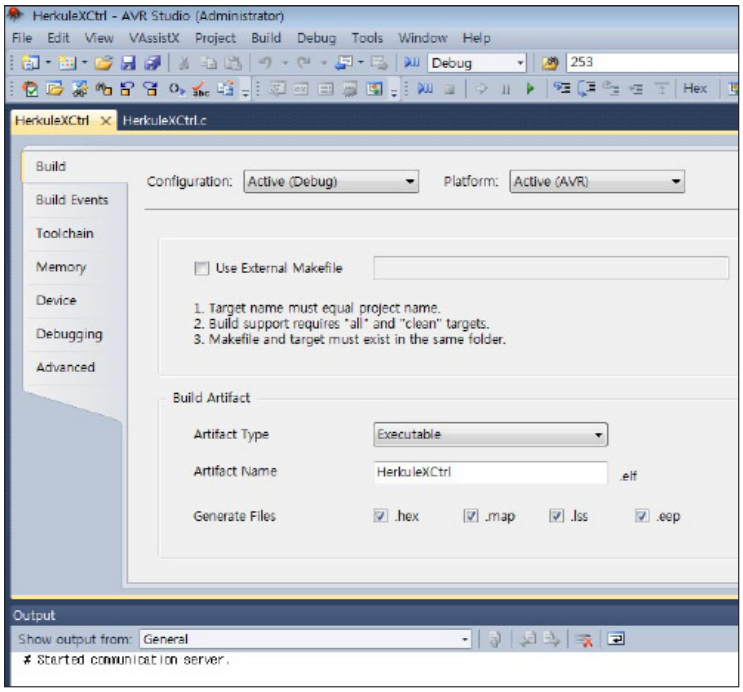
04 ATmega 선택

우측 상단에 atmega 를 검색하여, 리스트중에 ATmega128 을 선택합니다.



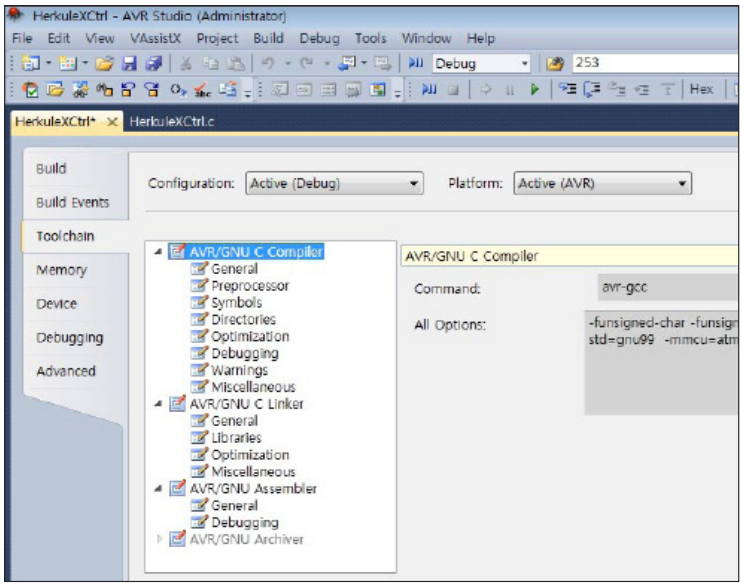
05 Editor

편집창에 코딩을 시작합니다.



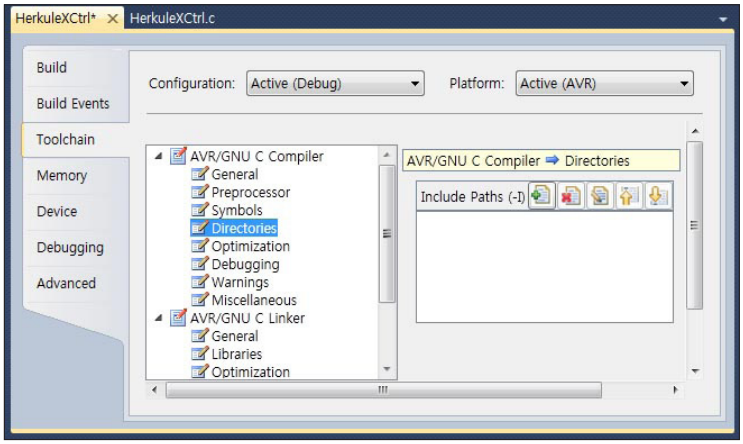
01 설정

AVR Studio 상단 메뉴중에 Project 를 클릭하면 HerkuleXCtrl (프로젝트 이름별로 다름) Property 메뉴가 생성되어 있습니다. 클릭하면 좌측과 같이 설정화면이 나옵니다.



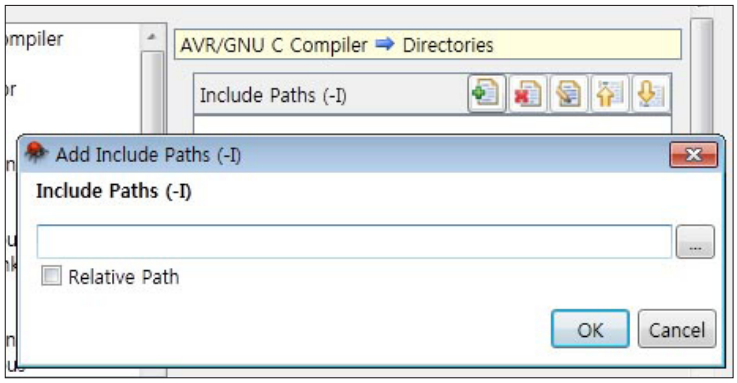
02 Toolchain

좌측 탭버튼중에 Toolchain 을 클릭하여, AVR/GNU C Compiler 을 선택합니다.



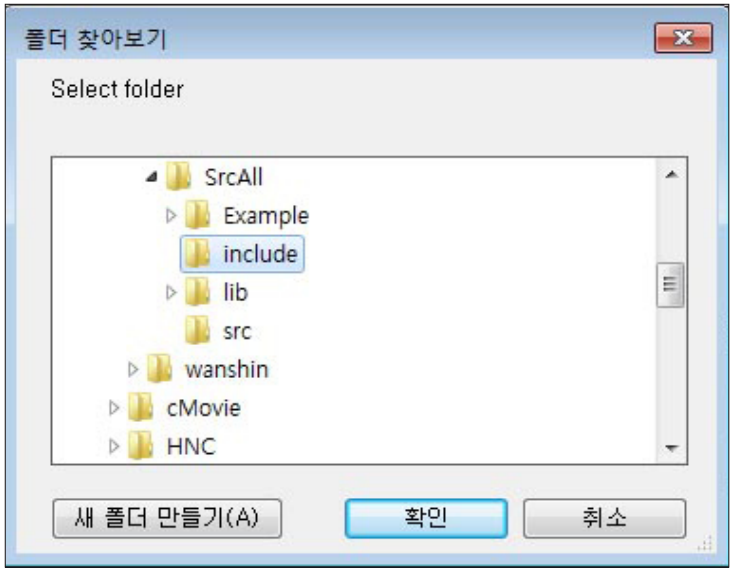
03 C Compiler

C Compiler 폴더중에 Directories 를 선택합니다.



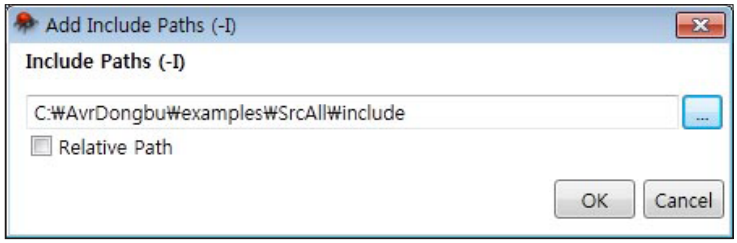
04 Path

Include Paths 오른쪽에 + 버튼을 클릭하면 팝업 입력창이 뜹니다.
Relative Path (상대경로) 체크를 해지하고, 우측 폴더 검색 버튼을 클릭합니다.
※5.1일경우 해지할 필요가 없습니다.



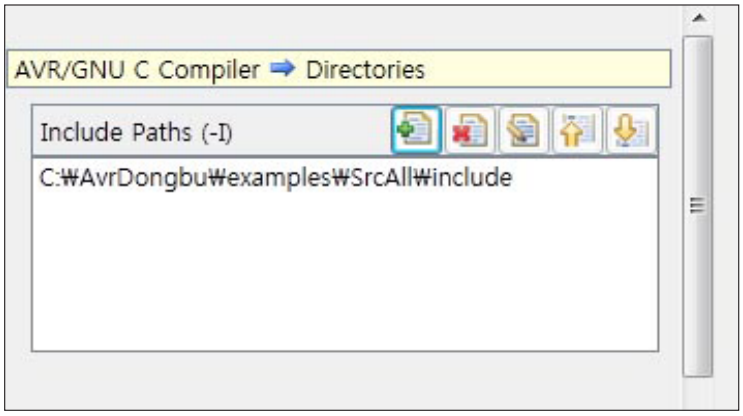
05 Include

제공 라이브러리 폴더중에 include 를 선택합니다.



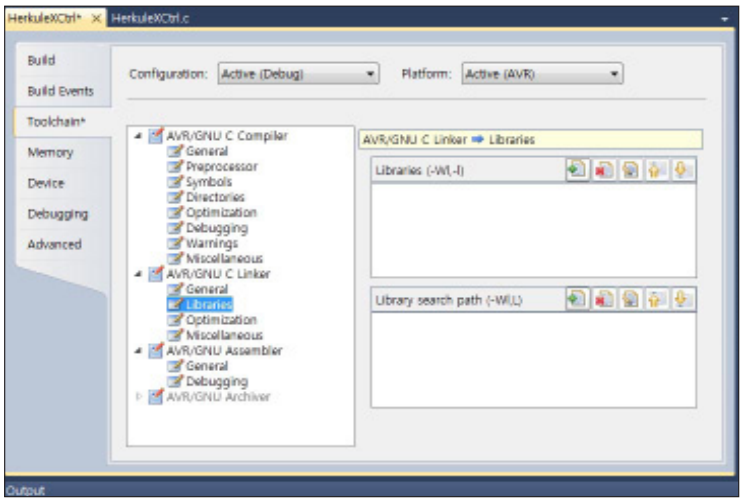
06 Include 완료

Include 폴더 위치가 지정됩니다. OK 버튼을 클릭합니다



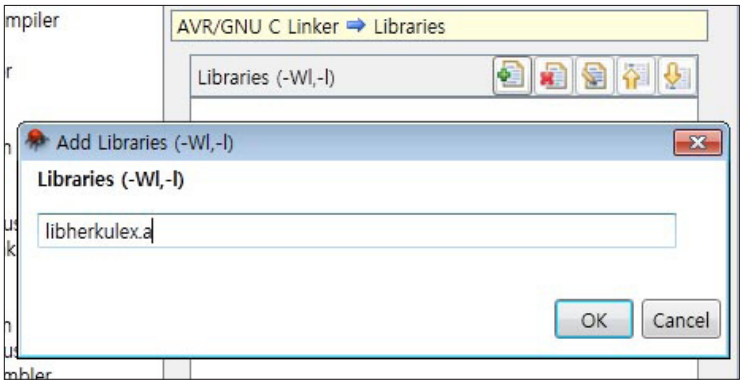
07 Path 완료

Include 경로지정이 완료되었습니다.



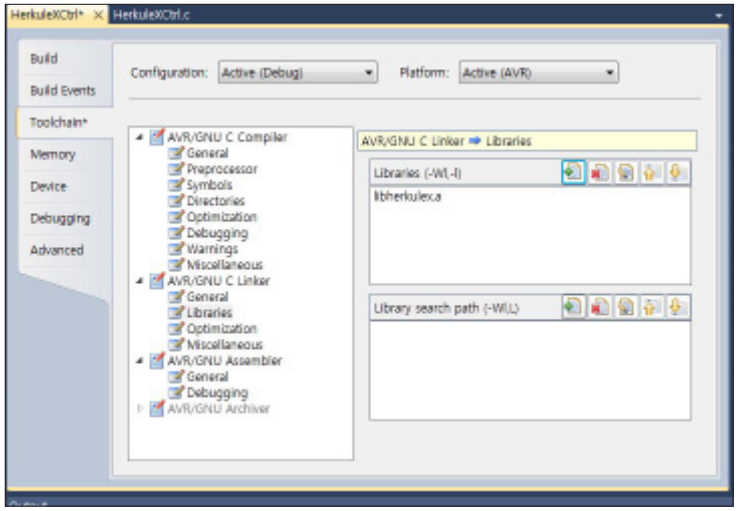
08 C Linker

C Linker 폴더중에 Libraries 를 선택합니다



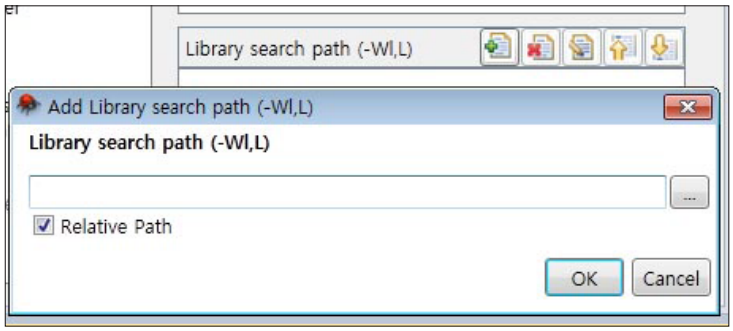
09 Libraries

Libraries 창 우측 + 를 클릭하면, Libraries 파일명 입력창이 나옵니다.
libherkulex.a 를 직접 입력합니다.
OK 버튼을 클릭합니다.



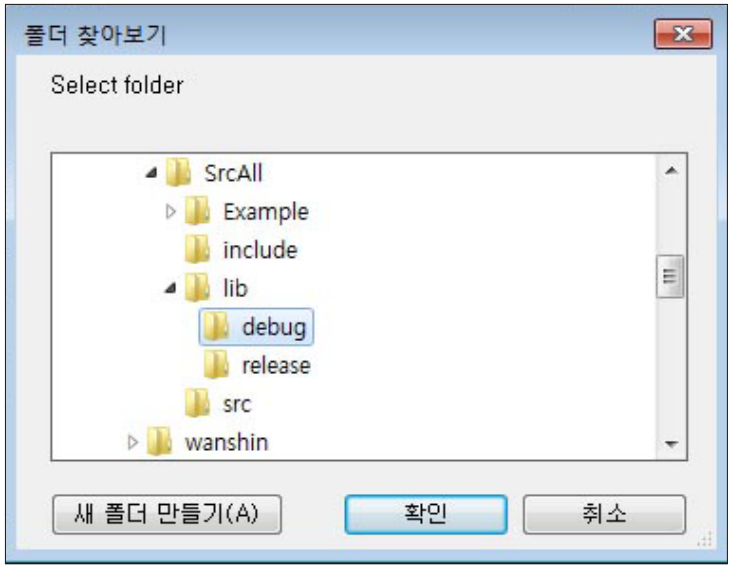
10 Search

서보모터 라이브러리 파일 libherkulx.a 링크가 완료되었습니다.
다음으로 Library search path 를 설정합니다.
우측 + 버튼을 클릭합니다.



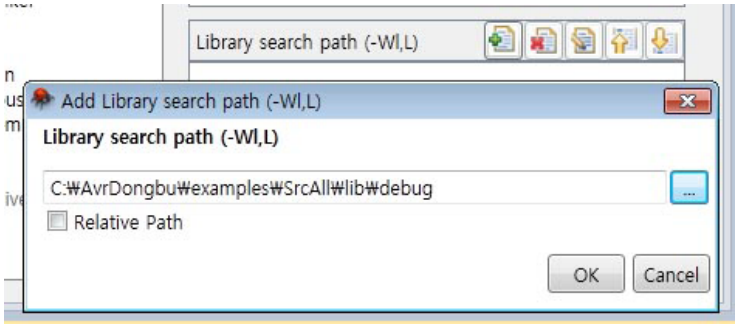
11 Path

Library search path 팝업창 우측에 폴더 검색 버튼을 클릭합니다.



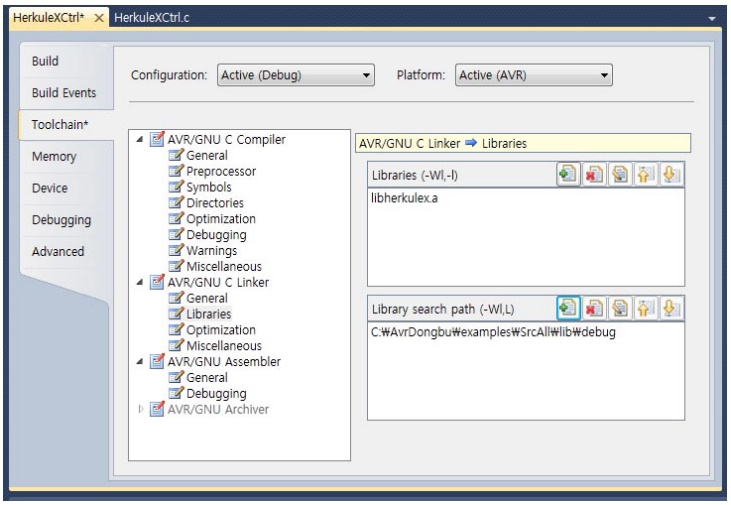
12 Debug

제공 라이브러리 폴더에서 lib 하위 폴더에 debug 를 선택합니다.



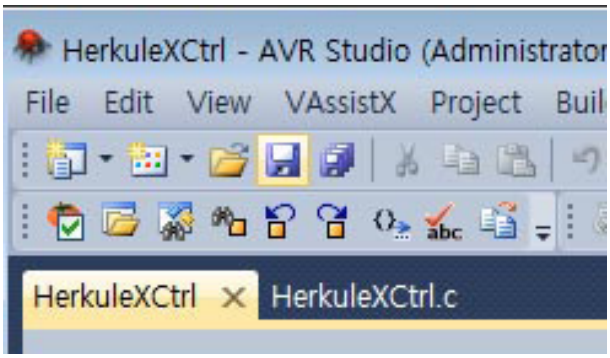
13 Path 완료

Relative Path(상대경로지정) 체크박스를 해지합니다.
Library search path 경로지정이 완료되었습니다.
OK 버튼을 클릭합니다.



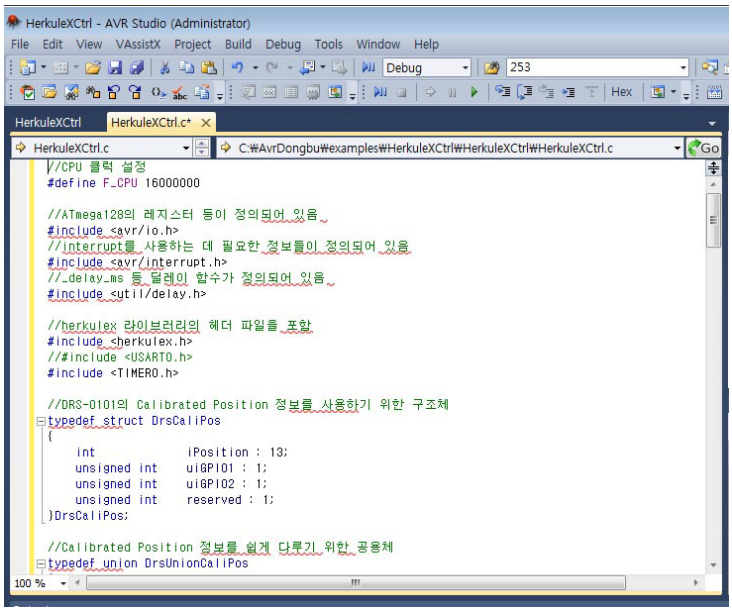
14 Link 완료

C Linker 라이브러리파일과 경로지정이 완료되었습니다.



15 저장

저장 버튼을 누르면 현재 설정값이 저장됩니다. 탭바 이름이 HerkuleXCtrl *〉 HerkuleXCtrl 로 변경됩니다.



16 코딩

HerkuleX.c 탭바를 선택하고, 아래 제공된 예제를 입력합니다.

Servo ID 는 기본 253번입니다. 서보모터만 단품으로 구매시 기본 ID 는 253번이지만, 휴머노이드 세트로 구매시에는 서보마다 ID 값이 표기되어있습니다. 그 서보모터를 사용시에는 그 ID 를 입력합니다

예제

ID가 253인 DRS-0101 서보 모터를 반복적으로 움직이면서, 위치 값을 읽어서 목표 위치에 실제로 도달하였는지 확인하는 예제입니다.

TIMER0.h

```
#ifndef TIMER0_H_
#define TIMER0_H_

#ifdef __TIMER0_C
	#define TIMER0_EXT
#else
	#define TIMER0_EXT extern
#endif

//타이머를 사용해 시간을 측정할 때의 전역변수
TIMER0_EXT volatile unsigned char gucTimerTick;
//타이머/카운터0의 초기화 함수
TIMER0_EXT void TIMER0_Init(void);

#endif /* TIMER0_H_ */
```

herkulex.h

```
#ifndef HERKULEX_H_
#define HERKULEX_H_

//////////////////////////////프로토콜에 대한 선언//////////////////////////////
//각 항목의 인덱스
#define PROTOCOL_SIZE_IDX		2
#define PROTOCOL_ID_IDX			3
#define PROTOCOL_CMD_IDX		4
#define PROTOCOL_CS1_IDX		5
#define PROTOCOL_CS2_IDX		6
#define PROTOCOL_DATA_IDX		7

//헤더 관련
#define HEADER				0xFF

//SIZE 관련
#define MIN_PACKET_SIZE			7
#define MIN_ACK_PACKET_SIZE		9
#define MAX_PACKET_SIZE			223
#define MAX_DATA_SIZE			(MAX_PACKET_SIZE-MIN_PACKET_SIZE)
```

```
//ID 관련
#define MAX_ID                          0xFD
#define BROADCAST_ID                    0xFE

//CMD 관련 - Request Packet
#define CMD_EEP_WRITE                   0x01
#define CMD_EEP_READ                    0x02
#define CMD_RAM_WRITE                   0x03
#define CMD_RAM_READ                    0x04
        #define CMD_RW_DATA_ADDR_IDX    7
        #define CMD_RW_DATA_LEN_IDX     8
#define CMD_I_JOG                       0x05
        #define CMD_I_JOG_STRUCT_SIZE   5
        #define CMD_I_JOG_MAX_DRS       (MAX_DATA_SIZE/CMD_I_JOG_STRUCT_SIZE)
#define CMD_S_JOG                       0x06
        #define CMD_S_JOG_STRUCT_SIZE   4
        #define CMD_S_JOG_MAX_DRS       (MAX_DATA_SIZE/CMD_S_JOG_STRUCT_SIZE)
#define CMD_STAT                        0x07
#define CMD_ROLLBACK                    0x08
#define CMD_REBOOT                      0x09

#define CMD_MIN                         (CMD_EEP_WRITE)
#define CMD_MAX                         (CMD_REBOOT)

//CMD 관련 - ACK Packet
#define CMD_ACK_MASK                    0x40

#define CMD_EEP_WRITE_ACK               (CMD_EEP_WRITE|CMD_ACK_MASK)
#define CMD_EEP_READ_ACK                (CMD_EEP_READ|CMD_ACK_MASK)
#define CMD_RAM_WRITE_ACK               (CMD_RAM_WRITE|CMD_ACK_MASK)
#define CMD_RAM_READ_ACK                (CMD_RAM_READ|CMD_ACK_MASK)
#define CMD_I_JOG_ACK                   (CMD_I_JOG|CMD_ACK_MASK)
#define CMD_S_JOG_ACK                   (CMD_S_JOG|CMD_ACK_MASK)
#define CMD_STAT_ACK                    (CMD_STAT|CMD_ACK_MASK)
#define CMD_ROLLBACK_ACK                (CMD_ROLLBACK|CMD_ACK_MASK)
#define CMD_REBOOT_ACK                  (CMD_REBOOT|CMD_ACK_MASK)

#define CMD_ACK_MIN                     (CMD_EEP_WRITE_ACK)
#define CMD_ACK_MAX                     (CMD_REBOOT_ACK)

//CheckSum 관련
#define CHKSUM_MASK                     0xFE
```

```
//////////////////////////////프로토콜 구조체//////////////////////////////
typedef struct DrsJog
{
        unsigned int    uiValue : 15;
        unsigned int    reserved : 1;
}DrsJog;

typedef struct DrsSet
{
        unsigned char   ucStopFlag : 1;
        unsigned char   ucMode : 1;
        unsigned char   ucLedGreen : 1;
        unsigned char   ucLedBlue : 1;
        unsigned char   ucLedRed : 1;

        unsigned char   ucJogInvalid : 1;
        unsigned char   reserved : 2;
}DrsSet;

typedef struct DrsIJog
{
        DrsJog          stJog;
        DrsSet          stSet;
        unsigned char   ucId;
        unsigned char   ucPlayTime;
}DrsIJog;

typedef struct DrsSJog
{
        DrsJog          stJog;
        DrsSet          stSet;
        unsigned char   ucId;
}DrsSJog;

typedef struct DrsIJogData
{
        DrsIJog         stIJog[CMD_I_JOG_MAX_DRS];
}DrsIJogData;

typedef struct DrsSJogData
{
        unsigned char   ucPlayTime;
        DrsSJog         stSJog[CMD_S_JOG_MAX_DRS];
}DrsSJogData;

typedef struct DrsRWData
{
        unsigned char   ucAddress;
```

```
        unsigned char   ucLen;
        unsigned char   ucData[MAX_DATA_SIZE-2];
}DrsRWData;

typedef union DrsData
{
        unsigned char   ucData[MAX_PACKET_SIZE-MIN_PACKET_SIZE];

        DrsRWData       stRWData;
        DrsIJogData     stIJogData;
        DrsSJogData     stSJogData;
}DrsData;

typedef struct
{
        unsigned char   ucHeader[2];
        unsigned char   ucPacketSize;
        unsigned char   ucChipID;
        unsigned char   ucCmd;
        unsigned char   ucCheckSum1;
        unsigned char   ucCheckSum2;
        DrsData         unData;
}DrsPacket;

//////////////////////////////수신 상태 결과 값////////////////////////////////
enum{
        DRS_RXWAITING,
        DRS_RXCOMPLETE,
        DRS_HEADERNOTFOUND,
        DRS_INVALIDSIZE,
        DRS_UNKNOWNCMD,
        DRS_INVALIDID,
        DRS_CHKSUMERROR,
        DRS_RXTIMEOUT
}DrsRxStatus;

#ifdef __HERKULEX_C
        #define HERKULEX_EXT
#else
        #define HERKULEX_EXT extern
#endif

//HerkuleX를 제어하기 위해 초기화하는 함수
HERKULEX_EXT void hklx_Init(unsigned long ulBaudRate);
//HerkuleX로 패킷을 보내는 함수
HERKULEX_EXT void hklx_SendPacket(DrsPacket stPacket);
//HerkuleX로부터 패킷을 받는 함수
HERKULEX_EXT unsigned char hklx_ucReceivePacket(DrsPacket *pstPacket);

#endif /* HERKULEX_H_ */
```

HerkuleXCtrl.c

```
//CPU 클럭 설정
#define F_CPU 16000000

//ATmega128의 레지스터 등이 정의되어 있음
#include <avr/io.h>
//interrupt를 사용하는 데 필요한 정보들이 정의되어 있음
#include <avr/interrupt.h>
//_delay_ms 등 딜레이 함수가 정의되어 있음
#include <util/delay.h>

//herkulex 라이브러리의 헤더 파일을 포함
#include <herkulex.h>
#include <TIMER0.h>

//DRS-0101의 Calibrated Position 정보를 사용하기 위한 구조체
typedef struct DrsCaliPos
{
	int		iPosition : 13;
	unsigned int	uiGPIO1 : 1;
	unsigned int	uiGPIO2 : 1;
	unsigned int	reserved : 1;
}DrsCaliPos;

//Calibrated Position 정보를 쉽게 다루기 위한 공용체
typedef union DrsUnionCaliPos
{
	DrsCaliPos	stCaliPos;
	unsigned int	uiCaliPos;
}DrsUnionCaliPos;

int main(void)
{
	//보낼 패킷과 받을 패킷이 저장될 패킷 구조체 선언
	DrsPacket stSendPacket, stRcvPacket;
	//Calibrated Position을 저장할 공용체 변수 선언
	DrsUnionCaliPos unCaliPos;
	//사용할 변수들 선언
	unsigned char ucResult;
	int iPos = 400;

	//포트 C의 입출력 방향 설정
	DDRC = 0b01111111;
	//포트 C의 출력 값 설정
	PORTC = 0b01111111;
```

```
//전체 인터럽트를 비활성화
cli();

//HerkuleX를 사용하기 위해 초기화
hklx_Init(115200);

//전체 인터럽트를 활성화
sei();

//Torque Control에 0x60을 써서 토크를 거는 패킷 구성
stSendPacket.ucPacketSize = MIN_PACKET_SIZE + 3;
stSendPacket.ucChipID = 253;
stSendPacket.ucCmd = CMD_RAM_WRITE;
stSendPacket.unData.stRWData.ucAddress = 52;
stSendPacket.unData.stRWData.ucLen = 1;
stSendPacket.unData.stRWData.ucData[0] = 0x60;

//패킷 보내기
hklx_SendPacket(stSendPacket);

while(1)
{
        //목표 위치 변수 값 변경
        if(iPos == 400){
                iPos = 624;
        }
        else{
                iPos = 400;
        }

        //DRS-0101을 움직일 I_JOG 패킷 구성
        stSendPacket.ucPacketSize = MIN_PACKET_SIZE + CMD_I_JOG_STRUCT_SIZE;
        stSendPacket.ucChipID = BROADCAST_ID;
        stSendPacket.ucCmd = CMD_I_JOG;
        stSendPacket.unData.stIJogData.stIJog[0].stJog.uiValue = iPos;
        stSendPacket.unData.stIJogData.stIJog[0].stSet.ucStopFlag = 0;
        stSendPacket.unData.stIJogData.stIJog[0].stSet.ucMode = 0;
        stSendPacket.unData.stIJogData.stIJog[0].stSet.ucLedGreen = 1;
        stSendPacket.unData.stIJogData.stIJog[0].stSet.ucLedBlue = 1;
        stSendPacket.unData.stIJogData.stIJog[0].stSet.ucLedRed = 1;
        stSendPacket.unData.stIJogData.stIJog[0].stSet.ucJogInvalid = 0;
        stSendPacket.unData.stIJogData.stIJog[0].ucId = 253;
        stSendPacket.unData.stIJogData.stIJog[0].ucPlayTime = 50;

        //패킷 보내기
        hklx_SendPacket(stSendPacket);

        //다 움직일 때 까지 1초 대기
        _delay_ms(1000);
```

```
                //Calibrated Position을 읽어올 RAM_READ 패킷 구성
                stSendPacket.ucPacketSize = MIN_PACKET_SIZE + 2;
                stSendPacket.ucChipID = 253;
                stSendPacket.ucCmd = CMD_RAM_READ;
                stSendPacket.unData.stRWData.ucAddress = 58;
                stSendPacket.unData.stRWData.ucLen = 2;

                //패킷 보내기
                hklx_SendPacket(stSendPacket);

                //TIMEOUT 까지 기다릴 시간 설정. 2*15 = 30(ms)
                gucTimerTick=15;
                while(1){
                        //패킷을 받는 함수를 호출해 결과를 ucResult에 저장
                        ucResult = hklx_ucReceivePacket(&stRcvPacket);
                        //결과 값이 DRS_RXWAITING이 아니면 빠져나옴
                        if(ucResult != DRS_RXWAITING){
                                break;
                        }
                        //30ms가 지나서 gucTimerTick이 0이 되면 빠져나옴
                        if(gucTimerTick==0){
                                ucResult = DRS_RXTIMEOUT;
                                break;
                        }
                }

                //패킷 수신이 정상적으로 완료 시
                if(ucResult == DRS_RXCOMPLETE){
                        //받은 데이터 2바이트를 공용체 변수에 저장
                        unCaliPos.uiCaliPos = stRcvPacket.unData.stRWData.ucData[0] |
                                              (stRcvPacket.unData.stRWData.ucData[1]<<8);

                        //목표 위치와 5 이내로 근접했으면 LED 모두 끔
                        if((iPos - unCaliPos.stCaliPos.iPosition) > -5 &&
                           (iPos - unCaliPos.stCaliPos.iPosition) < 5){
                                PORTC = 0b01111111;
                        }
                        //목표 위치의 5 밖으로 나갔으면 LED 모두 켬
                        else{
                                PORTC = 0b00000000;
                        }
                }
                //패킷 수신이 정상적으로 이루어지지 않았을 때 LED 모두 켬
                else{
                        PORTC = 0b00000000;
                }
        }

        //함수의 형태에 맞게 정수값 반환
        return 1;
}
```

예제설명

먼저 HerkuleXCtrl.c에서 #include문을 사용해 포함시켜주는 헤더 파일에 대한 설명을 합니다.

TIMER0.h

```
#ifndef TIMER0_H_
#define TIMER0_H_

...

#endif /* TIMER0_H_ */
```

#ifndef 문은 코드 컴파일러 지시자 중 하나로, #ifndef TIMER0_H_ 는 지금까지 컴파일을 하면서 TIMER0_H_가 #define을 사용해서 선언된 적이 있는지 판단하여, TIMER0_H_가 선언된 적이 없다면 #endif 까지의 내용을 컴파일에 반영합니다. 만약 이미 TIMER0_H_가 선언이 되어 있다면 #endif까지의 내용은 반영되지 않습니다. 비슷한 기능을 하는 지시자로 #ifdef가 있습니다. #ifdef는 #ifndef와는 반대로, 이후에 나오는 것이 선언된 적이 있으면 #endif까지 컴파일에 반영하며 없으면 반영하지 않습니다.

보통 c언어의 헤더 파일은 위 코드와 같이 #ifndef, #define으로 시작해서 #endif로 끝납니다. 이렇게 코딩을 하면, TIMER0.h가 처음 #include문에 의해 포함되었을 때는 TIMER0_H_가 선언된 적이 없으므로 안의 내용을 컴파일에 반영하게 되며, 이때 #define TIMER0_H_에 의해서 TIMER0_H_가 선언됩니다. 이렇게 해놓으면, 만약 TIMER0.h가 다시 또 #include문에 의해 포함되는 일이 생기면 이미 TIMER0_H_가 선언되어 있기 때문에 TIMER0.h의 내용을 컴파일하지 않게 됩니다. 한 헤더 파일을 두 번 컴파일 하는 경우, 전역 변수나 함수 원형이 두 번 이상 선언되어서 컴파일 에러를 일으킬 수 있습니다. 위와 같이 헤더 파일을 작성하면, 한 헤더파일이 한번만 컴파일 되기 때문에 에러를 방지할 수 있습니다.

```
#ifdef __TIMER0_C
        #define TIMER0_EXT
#else
        #define TIMER0_EXT extern
#endif

TIMER0_EXT volatile unsigned char gucTimerTick;
TIMER0_EXT void TIMER0_Init(void);
```

```
#define __TIMER0_C
        #include “TIMER0.h”
#undef __TIMER0_C
```

(TIMER0.c의 일부분)

제일 위 코드는 모든 전역 변수와 함수 원형을 한번씩만 선언해주기 위한 부분입니다. 그 아래에 있는 TIMER0.c의 일부분에서 보듯이, TIMERT0.c에서 TIMER0.h를 #include문으로 포함할 때는 __TIMERT0_C를 #define으로 미리 선언하고 포함합니다. 그러면 TIMER0.c에서 TIMER0.h를 컴파일 할 때에는

```
volatile unsigned char gucTimerTick;
void TIMER0_Init(void);
```

로 컴파일 합니다. 변수와 함수 앞의 extern은 다른 소스 파일에 선언된 변수나 함수를 현재 소스 파일에서 접근하거나 호출할 때에 사용됩니다. TIMER0.c외의 다른 파일에서 TIMER0.c의 변수와 함수를 사용하기 위해 extern을 붙여서 컴파일 하는 것입니다. 지금까지 설명한 컴파일러 지시자에 관련된 내용은 TIMER0.h말고도 다른 헤더 파일에서도 모두 똑같이 적용되므로, 이후 다른 헤더 파일에서는 따로 언급하지 않을 것입니다.

TIMER0.h의 전역변수 gucTimerTick은 Chapter 5에서 사용한 변수와 같이, 0이 아닐 경우 2ms마다 1씩 줄어드는 변수입니다. TIMER0_Init() 함수는 TIMER0의 초기화 함수로, TCNT0을 초기 값으로 설정하고 타이머/카운터0 오버플로우 인터럽트를 활성화 하고, 지정된 분주비로 타이머/카운터0을 시작합니다.

herkulex.h

herkulex.h에서는 HerkuleX를 제어하는 데에 필요한 프로토콜에 대한 선언문과 구조체 등이 정의되어 있습니다. 처음의 프로토콜에 대한 선언 부분에서는 패킷을 이루는 헤더, SIZE, ID, CMD, Check Sum에 관련된 여러 가지 정보가 선언되어 있습니다. 그 다음에는 각 CMD에 따른 패킷 구조체가 정의되어 있어서, 사용자가 자신이 보낼 패킷의 내용을 손쉽게 구성할 수 있도록 되어 있습니다. DrsJog은 I_JOG와 S_JOG 패킷에서 HerkuleX가 이동할 위치/속도 정보가 들어간 구조체이고, DrsSet은 마찬가지로 I_JOG와 S_JOG 패킷에서 여러 설정 정보가 들어간 구조체입니다. DrsIJog와 DrsSJog는 각각 I_JOG와 S_JOG 패킷에서 서보 모터에 하나에 해당하는 정보가 들어간 구조체이며, DrsIJogData와 DrsSJogData는 서보 모터 전체에 해당하는 정보를 가집니다. DrsRWData는 EEP_WRITE, EEP_READ, RAM_WRITE, RAM_READ의 네 가지 명령에서 사용합니다. 데이터의 첫 바이트는 레지스터 주소, 두 번째 바이트는 읽거나 쓸 데이터 길이, 세 번째 바이트부터는 쓸 데이터(WRITE 명령에만 해당)가 들어가는 패킷 형식을 구조체로 나타냈습니다. DrsIJogData, DrsSJogData, DrsRWData 모두가 패킷에서 Optional Data 부분을 나타내는 구조체들이므로, DrsData라는 공용체로 묶어서 같은 메모리 영역을 CMD의 종류에 따라서 다르게 접근할 수 있도록 합니다.

마지막으로 DrsPacket은 헤더, SIZE, ID, CMD, Check Sum과 Optional Data를 모두 포함하는 패킷 전체에 대한 구조체입니다. 우리는 HerkuleXCtrl.c에서 DrsPacket 변수를 선언해서 패킷을 구성하고, 보내고, 받고, 받은 패킷을 해석합니다.

```
enum{
        DRS_RXWAITING,
        DRS_RXCOMPLETE,
        DRS_HEADERNOTFOUND,
        DRS_INVALIDSIZE,
        DRS_UNKNOWNCMD,
        DRS_INVALIDID,
        DRS_CHKSUMERROR,
        DRS_RXTIMEOUT
}DrsRxStatus;
```

패킷 구조체 다음에는 hklx_ucReceivePacket에서 반환되는 수신 상태 결과 값이 열거형 변수(enum) DrsRxStatus로 선언되어 있습니다.

```
HERKULEX_EXT void hklx_Init(unsigned long ulBaudRate);
HERKULEX_EXT void hklx_SendPacket(DrsPacket stPacket);
HERKULEX_EXT unsigned char hklx_ucReceivePacket(DrsPacket *pstPacket);
```

위 코드는 herkulex.h에서 선언된 함수들입니다. hklx_Init()은 말그대로 HerkuleX를 제어하기 위해서 ATmega128을 초기화하는 함수입니다. 타이머/카운터0과 USART0을 초기화합니다. hklx_SendPacket()은 HerkuleX로 패킷을 보내는 함수입니다. SIZE, ID, CMD, Optional Data를 DrsPacket 변수에 입력하고 hklx_SendPacket()을 호출하면, 이 함수에서 stPacket에 헤더와 체크섬을 입력한 후 USART0으로 송신합니다. hklx_ucReceivePacket()은 HerkuleX로부터 패킷을 받아오는 함수입니다. USART0의 수신 버퍼 내용을 검사해서, 온전한 패킷을 모두 받았으면 pstPacket에 복사하고 DRS_RXCOMPLETE를 반환합니다. 그렇지 않은 경우에는 현재 상황에 맞는 DrsRxStatus의 다른 값을 반환합니다.

HerkuleXCtrl.c

```
#define F_CPU 16000000
```

#define 문은 코드 전처리 명령 중 하나로, 앞으로 F_CPU라는 값이 나오면 16000000으로 대체한다는 뜻입니다. DRC의 ATmega128은 16MHz의 클럭 속도로 동작합니다. 클럭 속도에 비례해서 명령 처리 속도가 달라지므로, 이 명령을 통해서 이 프로그램이 동작할 속도를 명시해 주어야 delay 함수 등이 올바르게 동작합니다.

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

avr/io.h 헤더 파일에는 DDRX, PORTX 레지스터 등 AVR 제어를 위해 필요한 레지스터 등이 정의되어 있고, avr/interrupt.h 헤더 파일에는 인터럽트를 사용하기 위한 함수들이 정의되어 있습니다. util/delay.h 헤더 파일에는 _delay_ms()등 자주 사용하는 딜레이 함수가 정의되어 있으므로 #include 문을 사용하여 포함시켜 줍니다.

```
#include <herkulex.h>
#include <TIMER0.h>
```

herkulex.h 헤더 파일에는 패킷을 구성하고 보내기 위한 선언들과 구조체, 그리고 함수가 정의되어 있으므로 꼭 포함해야 합니다. 그리고 gucTimerTick을 사용해 패킷의 Timeout을 판정하기 위하여, TIMER0.h도 포함시킵니다.

```
typedef struct DrsCaliPos
{
        int             iPosition : 13;
        unsigned int    uiGPIO1 : 1;
        unsigned int    uiGPIO2 : 1;
        unsigned int    reserved : 1;
}DrsCaliPos;

typedef union DrsUnionCaliPos
{
        DrsCaliPos      stCaliPos;
        unsigned int    uiCaliPos;
}DrsUnionCaliPos;
```

앞서 언급했듯이, HerkuleX 레지스터 맵의 Calibrated Position은 위치 정보와 GPIO 입력 정보를 포함하고 있습니다. 우리가 필요한 것은 위치정보인데, 이렇게 되면 매번 Calibrated Position 값을 읽어올 때 마다 GPIO 입력 정보도 같이 읽혀오기 때문에 비트 연산을 통해서 위치 값만 남겨야 합니다. 이런 불편함을 해소하기 위해서 선언하는 것이 위 구조체와 공용체입니다. DrsCaliPos는 Calibrated Position의 각 비트가 어떻게 사용되는지를 나타내는 구조체이며, DrsUnionCaliPos는 이 구조체와 unsigned int를 같이 묶어서, Calibrated Position의 두 바이트 정보를 DrsCaliPos와 unsigned int의 두 방법으로 접근 가능하도록 했습니다.

```
DrsPacket stSendPacket, stRcvPacket;
DrsUnionCaliPos unCaliPos;
unsigned char ucResult;
int iPos = 400;
```

메인 함수에서 사용할 변수들을 선언합니다. DrsPacket 구조체 타입으로 보낼 때 쓸 변수와 받을 때 쓸 변수를 선언하고, 받은 데이터를 처리하기 위해 DrsUnionCaliPos 공용체 타입으로 unCaliPos를 선언합니다. 그리고 hklx_ucReceivePacket()의 결과 값이 저장될 ucResult를 선언하고, 모터의 목표 위치인 iPos를 선언하고 400으로 초기화 합니다.

```
DDRC = 0b01111111;
PORTC = 0b01111111;
```

포트 C의 0~6번 핀을 출력으로 설정하고 출력 값을 1로 만듭니다. 그러면 모든 LED가 꺼지게 됩니다.

```
cli();
hklx_Init(115200);
sei();
```

인터럽트에 대한 설정을 바꾸는 중에 인터럽트가 발생해 예상치 못한 오동작이 발생하는 것을 방지하기 위해서 우선 cli() 명령을 통해서 전체 인터럽트를 비활성화 시킵니다. 그 후 hklx_Init() 함수를 사용해서 HerkuleX를 제어하기 위한 타이머/카운터0과 USART0을 초기화합니다. 사용할 보드 레이트가 115200이므로 파라미터로 그 값을 사용합니다. 초기화가 끝난 후에는 sei() 명령을 통해서 전체 인터럽트를 활성화 합니다.

```
stSendPacket.ucPacketSize = MIN_PACKET_SIZE + 3;
stSendPacket.ucChipID = 253;
stSendPacket.ucCmd = CMD_RAM_WRITE;
stSendPacket.unData.stRWData.ucAddress = 52;
stSendPacket.unData.stRWData.ucLen = 1;
stSendPacket.unData.stRWData.ucData[0] = 0x60;

hkIx_SendPacket(stSendPacket);
```

모터가 스스로 움직이게 제어하기 위해서는 모터에 토크가 인가되어야 합니다. 위의 코드가 토크를 인가시키기 위한 RAM_WRITE 패킷을 구성하고, 보내는 부분입니다. 패킷의 SIZE는 기본 MIN_PACKET_SIZE(7바이트)에다가, 주소 + 길이 + 데이터(1바이트)로 총 MIN_PACKET_SIZE+3(10바이트)이며, 대상 모터의 ID가 253이므로 ID는 253입니다. CMD는 CMD_RAM_WRITE(0x03)이며, unData 공용체의 멤버 중 stRWData 구조체의 ucAddress, ucLen, ucData[0]의 값을 각각 52, 1, 0x60으로 만들어줍니다. 그 후에는 hkIx_SendPacket() 함수를 호출해 패킷을 보냅니다.

```
if(iPos == 400){
        iPos = 624;
}
else{
        iPos = 400;
}
```

여기서부터는 while문 안의 내용입니다. 모터의 목표 위치를 나타내는 변수인 iPos는 while문을 한번 반복할 때마다 번갈아서 400나 624로 값이 바뀌어야 하므로 위와 같이 값을 바꾸어 줍니다.

```
stSendPacket.ucPacketSize = MIN_PACKET_SIZE + CMD_I_JOG_STRUCT_SIZE;
stSendPacket.ucChipID = BROADCAST_ID;
stSendPacket.ucCmd = CMD_I_JOG;
stSendPacket.unData.stIJogData.stIJog[0].stJog.uiValue = iPos;
stSendPacket.unData.stIJogData.stIJog[0].stSet.ucStopFlag = 0;
stSendPacket.unData.stIJogData.stIJog[0].stSet.ucMode = 0;
stSendPacket.unData.stIJogData.stIJog[0].stSet.ucLedGreen = 1;
stSendPacket.unData.stIJogData.stIJog[0].stSet.ucLedBlue = 1;
stSendPacket.unData.stIJogData.stIJog[0].stSet.ucLedRed = 1;
stSendPacket.unData.stIJogData.stIJog[0].stSet.ucJogInvalid = 0;
stSendPacket.unData.stIJogData.stIJog[0].ucId = 253;
stSendPacket.unData.stIJogData.stIJog[0].ucPlayTime = 50;

hkIx_SendPacket(stSendPacket);

_delay_ms(1000);
```

모터가 iPos의 위치로 움직이도록 I_JOG 패킷을 구성하고, 보내는 부분입니다. 패킷의 SIZE는 기본 MIN_PACKET_SIZE(7바이트)에다가, 모터 하나당 데이터가 CMD_I_JOG_STRUCT_SIZE(5바이트)이므로 이 둘을 합친 길이(12바이트)가 됩니다. I_JOG는 여러 모터들에게 한꺼번에 명령을 날리는 것이므로 ID는 BROADCAST_ID(254)입니다. CMD는 CMD_I_JOG(0x05)이며, unData 공용체의 멤버 중 stIJogData 구조체의 stIJog[0]에 정보를 알맞게 넣어줍니다.
stJog의 uiValue에는 목표 위치인 iPos 값을 넣고, 위치 제어 모드이므로 stSet의 ucMode는 0, LED 3개를 다 켜기 위해서 stSet의 ucLedGreen, ucLedBlue, ucLedRed를 모두 1로, ucID는 253으로, ucPlayTime은 50(560ms)으로 설정합니다. 그 후에는 hklx_SendPacket() 함수를 호출해 패킷을 보내고, 움직임이 끝날 때까지 1초간 대기합니다.

```
stSendPacket.ucPacketSize = MIN_PACKET_SIZE + 2;
stSendPacket.ucChipID = 253;
stSendPacket.ucCmd = CMD_RAM_READ;
stSendPacket.unData.stRWData.ucAddress = 58;
stSendPacket.unData.stRWData.ucLen = 2;

hklx_SendPacket(stSendPacket);
```

모터의 움직임이 끝난 후에는 모터의 현재 위치 값을 읽어와서 모터가 실제로 목표 위치에 도달했는지 확인해야 합니다. 위의 코드가 Calibrated Position 값을 읽기 위한 RAM_READ 패킷을 구성하고, 보내는 부분입니다. 패킷의 SIZE는 기본 MIN_PACKET_SIZE(7바이트)에다가, 주소 + 길이로 총 MIN_PACKET_SIZE+2(9바이트)이며, 대상 모터의 ID가 253이므로 ID는 253입니다. CMD는 CMD_RAM_READ(0x04)이며, unData 공용체의 멤버 중 stRWData 구조체의 ucAddress, ucLen의 값을 각각 58, 2로 만들어줍니다. 그 후에는 hklx_SendPacket() 함수를 호출해 패킷을 보냅니다.

```
gucTimerTick=15;
while(1){
        ucResult = hklx_ucReceivePacket(&stRcvPacket);
        if(ucResult != DRS_RXWAITING){
                break;
        }

        if(gucTimerTick==0){
                ucResult = DRS_RXTIMEOUT;
                break;
        }
}
```

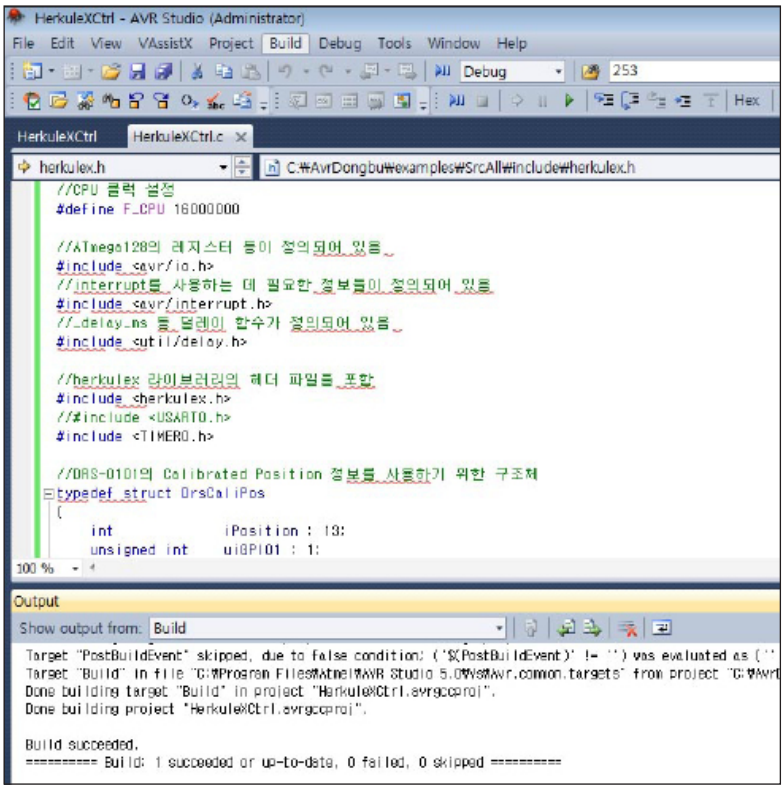
Timeout 시간을 설정해놓고 ACK Packet이 들어오기를 기다리는 함수입니다. libherkulex.a 파일에는 타이머/카운터0 오버플로우 인터럽트 서비스 루틴이 포함되어 있기 때문에, HerkuleXCtrl.c에서 gucTimerTick 값을 15로 만들면 이 변수는 타이머/카운터0 오버플로우 인터럽트가 발생하는 2ms마다 1씩 줄어들어서, 약 30ms 후에는 0이 됩니다. 그동안 while문 안에서는 계속 hklx_ucReceivePacket() 함수를 호출해서 ucResult에 그 결과 값을 저장합니다.
ACK 패킷이 들어오기를 기다릴 때에는 결과 값이 DRS_RXWAITING이며, 완성된 패킷을 받았거나 그 외에 잘못된 패킷이 들어온 상황에는 DrsRxStatus 중에서 그에 맞는 상태 값이 반환됩니다. 그렇기 때문에, ucResult가 DRS_RXWAITING이 아닌 경우에는 break문을 사용해서 while문을 빠져 나옵니다. 만약 30ms가 지나서 gucTimerTick이 0이 되었는데도 아직 ucResult가 DRS_RXWAITING일 경우에는, ucResult를 DRS_RXTIMEOUT으로 만들고 while문을 빠져 나옵니다.

```
if(ucResult == DRS_RXCOMPLETE){
        unCaliPos.uiCaliPos = stRcvPacket.unData.stRWData.ucData[0] |
                              (stRcvPacket.unData.stRWData.ucData[1]<<8);

        if((iPos - unCaliPos.stCaliPos.iPosition) > -5 &&
          (iPos - unCaliPos.stCaliPos.iPosition) < 5){
                PORTC = 0b01111111;
        }
        else{
                PORTC = 0b00000000;
        }
}
else{
        PORTC = 0b00000000;
}
```

만약 ucResult가 DRS_RXCOMPLETE로 완전한 ACK 패킷을 돌려받았다면, 받은 RAM_READ의 ACK 패킷 중 Calibrated Position 데이터가 들어 있는 부분을 unCaliPos 공용체 변수의 멤버 uiCaliPos에 저장합니다. 그리고 나서 unCaliPos의 stCaliPos 중 iPosition을 접근하면, 읽어온 데이터에서 위치 정보에 해당하는 하위 13비트만 접근해서 사용할 수 있습니다. if문을 사용해서 목표 위치인 iPos가 현재 위치인 unCaliPos.stCaliPos.iPosition과 5 미만으로 차이 나는지 검사하고, 만약 그렇다면 DRC의 LED를 모두 끕니다. 만약 5 이상 차이가 난다면 LED를 모두 켭니다. 또한, 패킷 에러가 발생해서 온전한 ACK 패킷을 받지 못한 경우에도 LED를 모두 켭니다.

실행

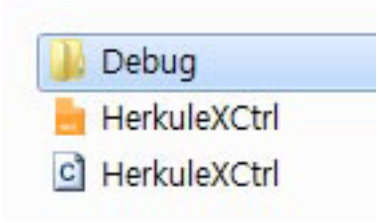
펌웨어를 다운로드하고 실행하면 ID 253인 HerkuleX가 400, 624 위치를 약 1초 주기로 왕복하는 것을 볼 수 있습니다. 정상적인 모터라면 DRC의 LED는 모두 꺼져 있어야 합니다.



01 Build

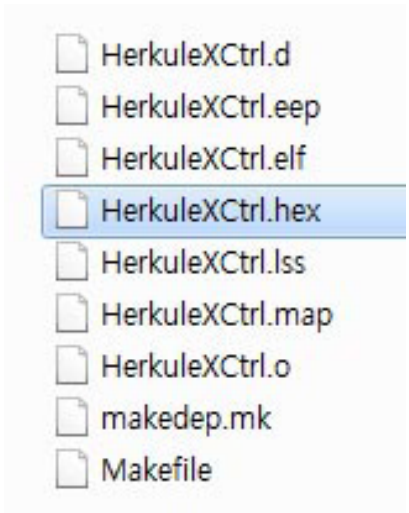
Build 메뉴를 선택하여 Build Solution을 클릭하여 Build 시킵니다.

1 succeeded 0 failed 가 나오면 에러없이 컴파일 완료됐다는 것을 의미합니다.



02 Debug 폴더

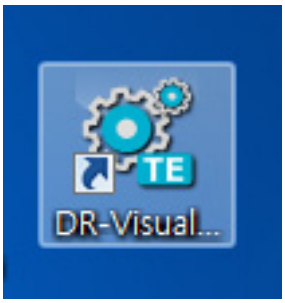
Build 후에 소스가 있는 폴더에 들어가면 Debug 폴더가 생성 되어있습니다. 클릭하여 폴더 안의 내용을 봅니다.



03 hex 확인

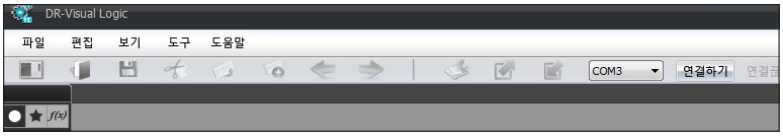
Debug 폴더 안의 hex 파일 확인합니다. Hex 파일이 제어기에 포팅시키는 파일입니다.

DR-Visual Logic 에서 펌웨어 복구를 실시합니다.



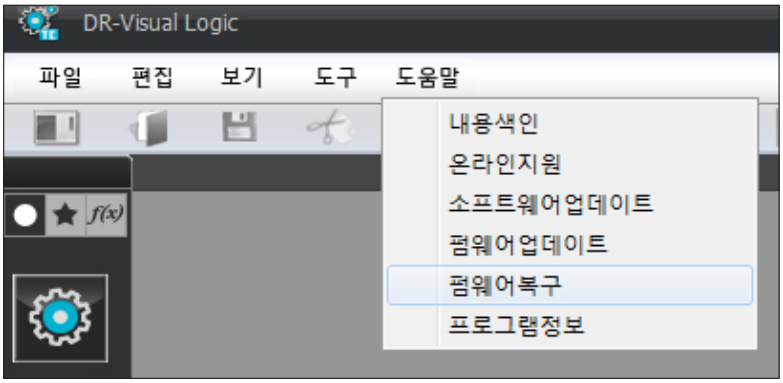
04 Visual Logic

DR-Visual Logic 을 실행시킵니다.



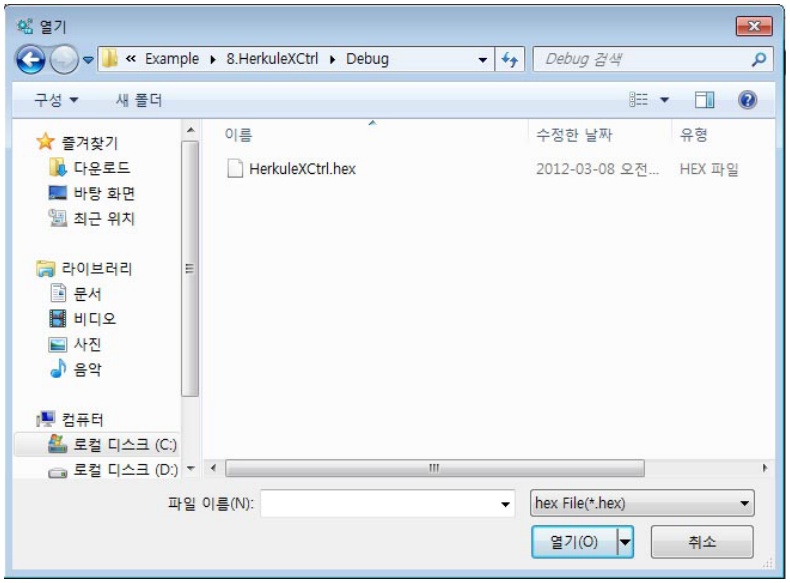
05 연결하기

COM 번호를 설정하고 연결하기를 클릭합니다.



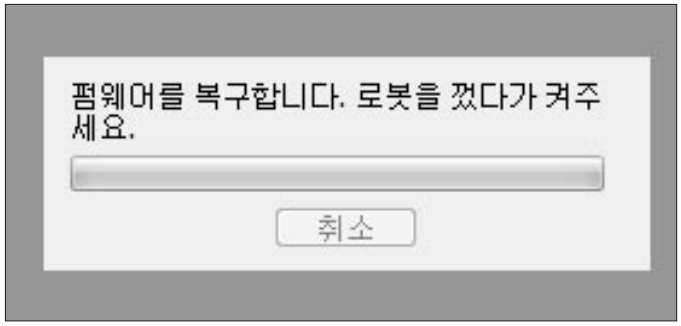
06 펌웨어 복구

도움말 메뉴에서 펌웨어 복구를 클릭합니다.



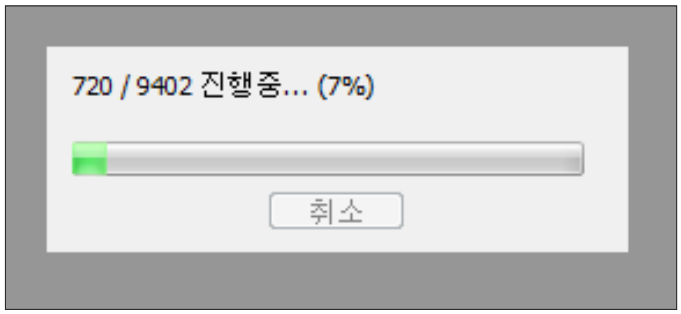
07 불러오기

Debug 폴더에서 Hex 파일을 불러옵니다



08 펌웨어복구

로봇을 껐다가 켜주세요 라는 문구가 나옵니다.
제어기의 전원버튼을 한번 껐다가 다시 켜주세요.



09 복구시작

펌웨어 복구가 시작됩니다.

10 실행

모터가 좌우로 움직입니다.
펌웨어를 다운로드하고 실행하면 ID 253인 HerkuleX가 400, 624 위치를 약 1초 주기로 왕복하는 것을 볼 수 있습니다. 정상적인 모터라면 DRC의 LED는 모두 꺼져 있어야 합니다.

11 분리

모터를 분리하면 모든 LED가 켜집니다. 제어기와 모터간 통신이 끊겼음을 의미합니다.

타이머/카운터로 버저 음계 재생하기

ATmega128의 타이머/카운터를 사용하여 여러 가지 음을 버저로 재생해봅니다.

AVR 원리

앞서 Chapter 5에서 설명한대로, ATmega128에는 8비트 타이머/카운터인 타이머/카운터0과 타이머/카운터2, 16비트 타이머/카운터인 타이머/카운터1과 타이머/카운터3으로 총 4개의 타이머/카운터가 있습니다. 이번 Chapter에서는 타이머/카운터2를 사용합니다. Chapter 5에서 설명한 타이머/카운터0과는 둘 다 8비트 타이머/카운터이므로 대체로 비슷합니다. 이번 Chapter에서는 타이머/카운터에 대한 전반적 내용은 생략하고, 타이머/카운터2와 타이머/카운터0의 차이점만 간단히 짚고 넘어갑니다. 별도의 언급이 없는 부분은 타이머/카운터0의 레지스터 이름과 비트 이름만 타이머/카운터2의 것으로 바꾸면 Chapter 5의 타이머/카운터0에 대한 설명과 동일합니다.

이번 Chapter 에서는 타이머/카운터2을 CTC(Clear Timer on Compare match) 모드로 사용하고, 알맞은 OCR2 값으로 음계를 재생합니다.

타이머/카운터2에 관련된 대표적인 레지스터는 아래와 같습니다.

TCCR2 : 타이머/카운터2 제어 레지스터

비트	7	6	5	4	3	2	1	0
	FOC2	WGM20	COM21	COM20	WGM21	CS22	CS21	CS20
읽기/쓰기	W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
초기값	0	0	0	0	0	0	0	0

– 비트 2:0번 CS22:0 : 클럭 선택

타이머/카운터2이 동작할 클럭을 선택합니다. ATmega128에 들어오는 기본 클럭 주파수를 어떤 비율로 나누어서 TCNT2에서 사용할지가 정해집니다. DRC의 기본 클럭 주파수는 16MHz입니다. 따라서 CS22:0을 0b100으로 설정했다면, 타이머/카운터2의 동작 주파수는 16Mhz/256 = 62.5kHz가 됩니다. 16us마다 TCNT2이 1씩 증가하는 것입니다.

타이머/카운터0의 CS02:0과 타이머/카운터2의 CS22:0은 클럭이 나뉘는 비율이 다르므로 사용시에 유의해야 합니다. 만약 T2핀에서 외부 클럭을 사용하도록 설정한다면, T2핀(포트 D의 7번 핀, PD7)이 입력이든 출력이든 상관없이 값의 변화가 발생하면 TCNT2 값이 변하게 됩니다. 이것은 곧 외부 신호를 받아서 클럭으로 사용할 수도 있지만, 사용자가 PD7의 값을 직접 바꿈을 통해 소프트웨어적으로 클럭을 만들어낼 수도 있다는 뜻입니다.

CS22	CS21	CS20	설명
0	0	0	동작 안 함
0	0	1	클럭/1
0	1	0	클럭/8
0	1	1	클럭/64
1	0	0	클럭/256
1	0	1	클럭/1024
1	1	0	T2핀에서 외부 클럭(하강 엣지)
1	1	1	T2핀에서 외부 클럭(상승 엣지)

TCNT2 : 타이머/카운터2 레지스터

비트	7	6	5	4	3	2	1	0
	TCNT2[7:0]							
읽기/쓰기	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
초기값	0	0	0	0	0	0	0	0

TCNT2은 타이머/카운터2의 현재 카운트 값을 저장하는 레지스터입니다. TCNT0과 동일하게 동작합니다.

OCR2 : 타이머/카운터2 출력 비교 레지스터

비트	7	6	5	4	3	2	1	0
	OCR2[7:0]							
읽기/쓰기	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
초기값	0	0	0	0	0	0	0	0

OCR2 레지스터 안에 들어있는 값은 TCNT2의 값이 변할 때마다 TCN2와 비교됩니다. OCRO와 동일하게 동작합니다.

TIMSK: 타이머/카운터2 인터럽트 마스크 레지스터

비트	7	6	5	4	3	2	1	0
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIEB	TOIE1	OCIE0	TOIE0
읽기/쓰기	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
초기값	0	0	0	0	0	0	0	0

– 비트 7번 OCIE2 : 타이머/카운터2 출력 비교 매치 인터럽트 활성화

이 비트가 1로 설정되면 타이머/카운터2의 비교 매치 인터럽트가 활성화됩니다. 이 비트가 1로 설정되고 SREG 레지스터의 I 비트가 1로 설정된 상태에서 비교 매치가 일어나면 TIFR 레지스터의 OCF2 비트가 1이 되고, 해당하는 인터럽트 서비스 루틴이 실행됩니다.

– 비트 6번 TOIE2 : 타이머/카운터2 오버플로우 인터럽트 활성화

이 비트가 1로 설정되면 타이머/카운터2의 오버플로우 인터럽트가 활성화됩니다. 이 비트가 1로 설정되고 SREG 레지스터의 I 비트가 1로 설정된 상태에서 오버플로우가 일어나면 TIFR 레지스터의 TOV2 비트가 1이 되고, 해당하는 인터럽트 서비스 루틴이 실행됩니다.

TIFR : 타이머/카운터2 인터럽트 플래그 레지스터

비트	7	6	5	4	3	2	1	0
	OCF2	TOV2	ICF1	OCF1A	OCFB	TOV1	OCF0	TOV0
읽기/쓰기	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
초기값	0	0	0	0	0	0	0	0

– 비트 7번 OCF2 : 타이머/카운터2 출력 비교 플래그

이 비트는 TCNT2과 OCR2 사이에서 비교 매치가 일어나면 1로 설정됩니다. 이 비트는 해당하는 인터럽트 서비스 루틴으로 진입할 때 하드웨어에 의해서 0으로 초기화됩니다. 소프트웨어적으로 수동으로 이 비트를 0으로 변하게 하려면, 이 비트에 1을 써주어야 합니다. SREG의 I 비트와 OCIE2, OCF2이 모두 1이라면, 타이머/카운터2 비교 매치 인터럽트가 실행됩니다.

– 비트 0번 TOV2 : 타이머/카운터2 오버플로우 플래그

이 비트는 타이머/카운터2에서 오버플로우가 일어나면 1로 설정되고, 해당하는 인터럽트 서비스 루틴으로 진입할 때 하드웨어에 의해서 0으로 초기화됩니다. 소프트웨어적으로 수동으로 이 비트를 0으로 변하게 하려면, 이 비트에 1을 써주어야 합니다. SREG의 I 비트와 TOIE2, TOV2이 모두 1이라면, 타이머/카운터2 오버플로우 인터럽트가 실행됩니다. Phase Correct PWM 모드에서는 다른 모드와 다르게, TCNT2의 증감 방향이 다운 카운트에서 업 카운트로 바뀔 때 (TCNT2=0일 때) TOV2 비트가 1로 설정됩니다.

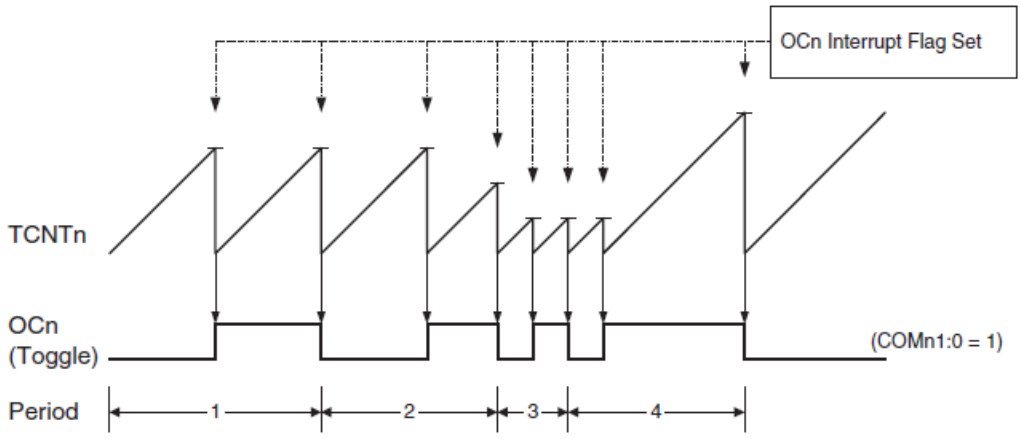
타이머/카운터2는 타이머/카운터0과 다르게 비동기 동작을 지원하지 않으므로, ASSR, SFIOR 등의 레지스터는 관련이 없습니다. 더 자세한 사항은 ATmega128의 데이터시트나 다른 서적을 참조하기 바랍니다.

타이머/카운터2의 동작 모드도 4가지가 있으며, 타이머/카운터0의 4가지 모드와 동일합니다. 이 중 우리가 사용할 모드는 CTC 모드입니다.

CTC모드는 Clear Timer on Compare 모드의 약자로, OCR2 레지스터가 타이머 초기화의 주기를 결정하는 데 사용됩니다. CTC 모드에서는 TCNT2 레지스터가 OCR2와 같아지는 순간, 즉 비교 매치가 일어나는 순간 0으로 초기화 됩니다. 즉 OCR2 레지스터의 값이 카운터의 최대값을 결정합니다. 비교 매치가 일어나는 순간 OCF2 비트는 1로 바뀌며, 타이머/카운터2 오버플로우 인터럽트 서비스 루틴이 실행되면 0으로 초기화 됩니다.

COM21:0 값을 0b01로 설정하면 비교 매치가 일어날 때마다 OC2 핀이 토글되도록 설정할 수 있습니다. 이를 이용하면 OC2 핀에 주기적인 펄스를 만들 수 있으며, OCR2의 값을 바꿈에 따라서 펄스의 주파수를 바꿀 수 있습니다.

이러한 원리를 이용해서 다양한 주파수로 OC2 핀이 토글되도록 하여 버저로 여러 가지 음계를 재생할 수 있습니다.



CTC 모드에서 OC2 핀이 토글되는 주파수를 계산하는 공식은 아래와 같습니다.

$$f_{OC2} = \frac{f_{clk\_I/O}}{2 \times N \times (1 + OCR2)}$$

$f_{clk\_I/O}$ 는 현재 ATmega128의 시스템 클럭이므로 16MHz입니다. N은 CS22:0에 의해서 클럭이 나누어지는 비율이며 우리 예제에서는 256입니다. OCR2는 이름 그대로 OCR2 레지스터의 값이며, 이 계산식을 통해서 나오는 주파수인 f_{OC2} 가 바로 버저가 울리는 음의 주파수입니다. 원하는 음에 해당하는 OCR2의 값을 알려면, 이 식을 거꾸로 풀어서 구해야 할 것입니다.

$$OCR2 = \frac{f_{clk\_I/O}}{2 \times N \times f_{OC2}} - 1$$

예를 들면, 5옥타브 라(A)라는 음의 주파수는 880Hz입니다. 5옥타브 라의 음을 내기 위한 OCR2의 값은,

$$OCR2 = \frac{16{,}000{,}000}{2 \times 256 \times 880} - 1 = 34.51 \approx 35$$

입니다.

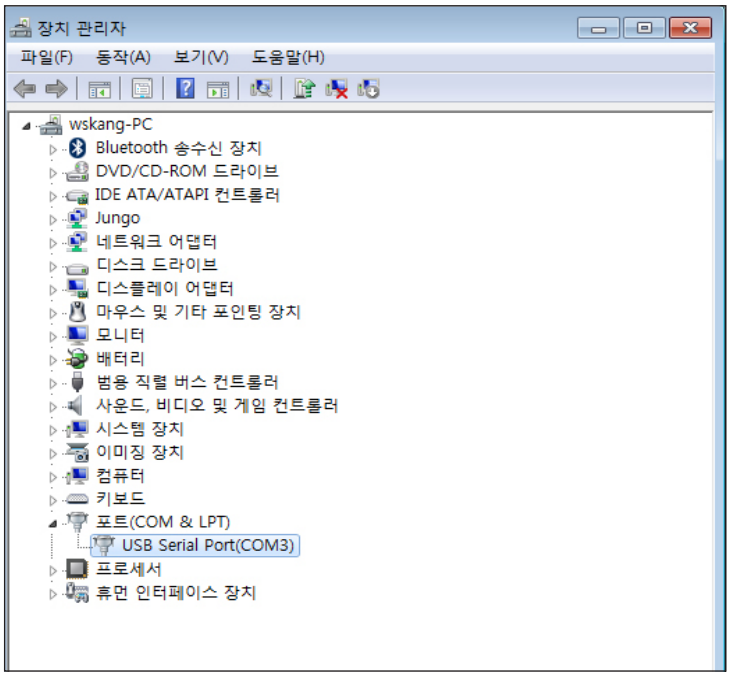
예제에서 사용할 음들의 주파수와, 위의 계산식을 통해서 계산한 OCR2의 값은 아래 표와 같습니다.

음계	주파수(Hz)	OCR2의 값
5옥타브 도(C)	523.25	59
5옥타브 레(D)	587.33	52
5옥타브 미(E)	659.26	46
5옥타브 파(F)	698.46	44
5옥타브 솔(G)	784.99	39
5옥타브 라(A)	880	35
5옥타브 시(B)	987.77	31
6옥타브 도(C)	1046.50	29

H/W 원리

이번 예제는 타이머/카운터2의 OC2핀(PB7)을 이용해 버저를 제어합니다. 버저 회로에 관련된 내용은 Chapter 3을 참고하면 됩니다.

준비



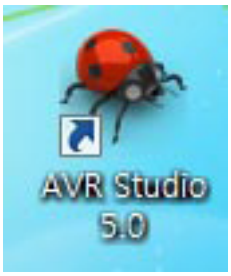
하드웨어 준비하기

01 연결

DRC 제어기 아답터로 전원을 연결하고, USB to Serial Cable 로 PC 와 DRC 를 연결합니다.

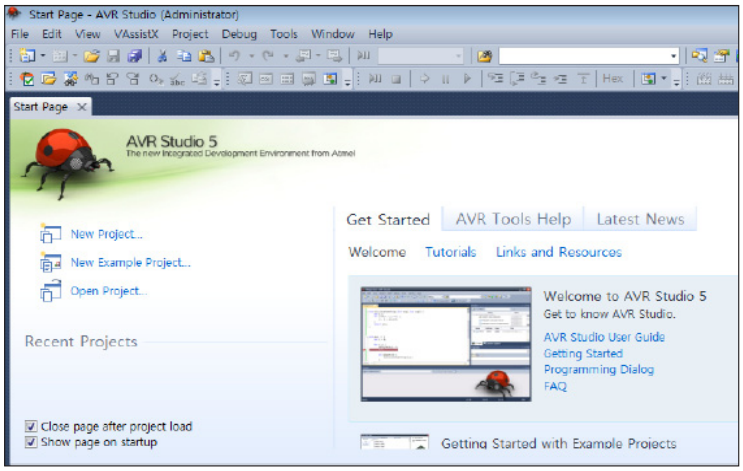
02 COM port 연결

내컴퓨터 〉 속성 〉 장치관리자에서 COM 연결 확인합니다.



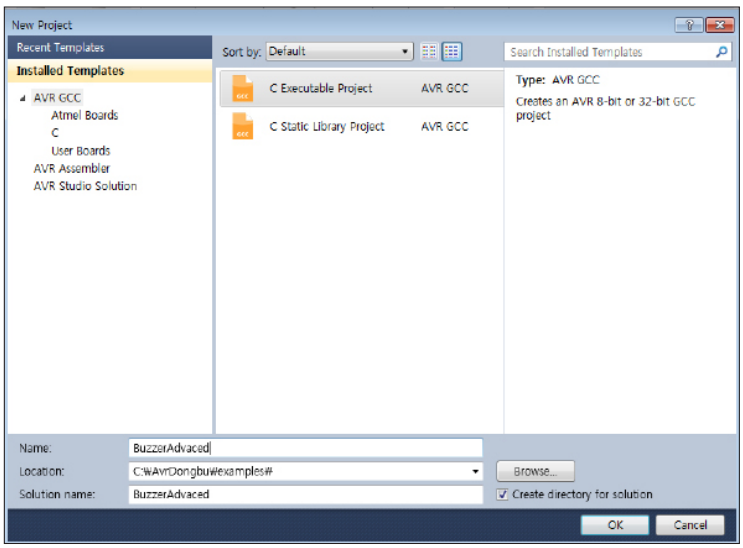
01 실행

AVR Studio 아이콘을 클릭하여 실행합니다.



02 New Project

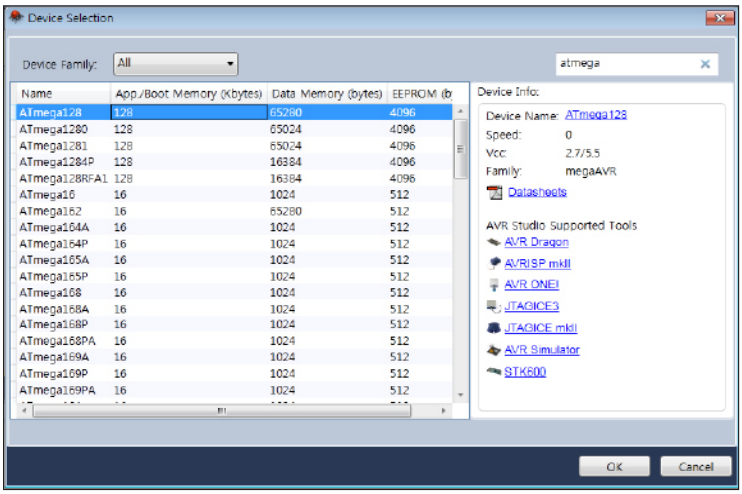
New Project 를 클릭합니다.



03 GCC 선택

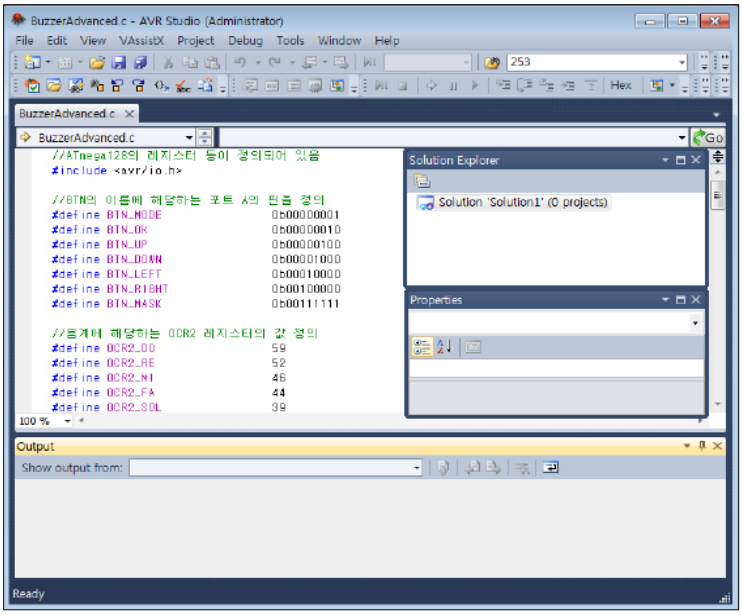
C Executable Project 를 선택합니다.

Name 을 “Buzzer Advaced”로 입력하고, Create directory for solution 을 클릭하고, OK 버튼을 누릅니다.



04 ATmega 선택

우측 상단에 atmega 를 검색하여,
리스트중에 ATmega128 을 선택합니다.



05 Editor

편집창에 코딩을 시작합니다.
아래 예제를 따라하세요.

예제

DRC의 6개의 버튼을 사용해서, 누르는 버튼에 따라서 도~라까지의 버저 음이 울리는 예제를 타이머/카운터2의 CTC 모드로 구현합니다. 버튼에 따른 버저 음은 아래와 같이 할당합니다.

버튼	음
Mode	도
OK	레
좌	미
상	파
우	솔
하	라

BuzzerAdvanced.c

```
//ATmega128의 레지스터 등이 정의되어 있음
#include <avr/io.h>

//BTN의 이름에 해당하는 포트 A의 핀을 정의
#define BTN_MODE                0b00000001
#define BTN_OK                  0b00000010
#define BTN_UP                  0b00000100
#define BTN_DOWN                0b00001000
#define BTN_LEFT                0b00010000
#define BTN_RIGHT               0b00100000
#define BTN_MASK                0b00111111

//음계에 해당하는 OCR2 레지스터의 값 정의
#define OCR2_DO                 59
#define OCR2_RE                 52
#define OCR2_MI                 46
#define OCR2_FA                 44
#define OCR2_SOL                39
#define OCR2_LA                 35
#define OCR2_SI                 31
#define OCR2_DO2                29

//타이머/카운터2의 클럭 속도
#define TIMER2_DIVIDER          0b00000100

int main(void)
{
        //8비트 변수 선언
        unsigned char ucButton;
        unsigned char ucButtonOld;

        //포트 A의 입출력 방향 설정
        DDRA = 0b00000000;

        //버저 핀(PB7)의 입출력 방향 설정
        DDRB |= 0b10000000;

        //포트 C의 입출력 방향 설정
        DDRC = 0b01111111;
        //포트 C의 출력 값 설정
        PORTC = 0b01111111
```

```
//타이머/카운터 2 초기화
TCNT2 = 0;
OCR2 = 0;
TCCR2 = 0;

//ucButtonOld를 현재 값으로 초기화
ucButtonOld = (~PINA) & BTN_MASK;

//while 안의 내용을 무한 반복
while(1)
{
        //포트 A로 부터 버튼 값 읽어오기
        ucButton = (~PINA) & BTN_MASK;

        //버튼 값이 바로 전 값과 다르다면
        if(ucButtonOld != ucButton){
                //MODE 버튼이 눌렸을 때 '도'음을 냄
                if(ucButton & BTN_MODE){
                        TCNT2 = 0;
                        OCR2 = OCR2_DO;
                        TCCR2 = (TIMER2_DIVIDER)|(1<<WGM21)|(1<<COM20);
                }
                //OK 버튼이 눌렸을 때 '레'음을 냄
                else if(ucButton & BTN_OK){
                        TCNT2 = 0;
                        OCR2 = OCR2_RE;
                        TCCR2 = (TIMER2_DIVIDER)|(1<<WGM21)|(1<<COM20);
                }
                //LEFT 버튼이 눌렸을 때 '미'음을 냄
                else if(ucButton & BTN_LEFT){
                        TCNT2 = 0;
                        OCR2 = OCR2_MI;
                        TCCR2 = (TIMER2_DIVIDER)|(1<<WGM21)|(1<<COM20);
                }
                //UP 버튼이 눌렸을 때 '파'음을 냄
                else if(ucButton & BTN_UP){
                        TCNT2 = 0;
                        OCR2 = OCR2_FA;
                        TCCR2 = (TIMER2_DIVIDER)|(1<<WGM21)|(1<<COM20);
                }
```

```
                    //RIGHT 버튼이 눌렸을 때 '솔'음을 냄
                    else if(ucButton & BTN_RIGHT){
                        TCNT2 = 0;
                        OCR2 = OCR2_SOL;
                        TCCR2 = (TIMER2_DIVIDER)|(1<<WGM21)|(1<<COM20);
                    }
                    //DOWN 버튼이 눌렸을 때 '라'음을 냄
                    else if(ucButton & BTN_DOWN){
                        TCNT2 = 0;
                        OCR2 = OCR2_LA;
                        TCCR2 = (TIMER2_DIVIDER)|(1<<WGM21)|(1<<COM20);
                    }
                    //버튼이 안 눌린 경우 소리 멈춤
                    else{
                        TCNT2 = 0;
                        OCR2 = 0;
                        TCCR2 = 0;
                    }
                }

                //현재 버튼 값을 다른 변수에 저장
                ucButtonOld = ucButton;
            }

            //함수의 형태에 맞게 정수값 반환
            return 1;
    }
```

예제설명

```
#include <avr/io.h>
```

avr/io.h 헤더 파일에는 DDRX, PORTX 레지스터 등 AVR 제어를 위해 필요한 레지스터 등이 정의되어 있습니다. 현재 소스 코드에서 사용할 수 있도록 #include 문을 사용하여 포함시켜 줍니다.

```
#define BTN_MODE                0b00000001
...
#define BTN_MASK                0b00111111
```

코드 전처리 명령인 #define을 사용해, 각 버튼에 해당하는 비트와 버튼 전체를 나타내는 마스크를 선언해줌으로써 버튼을 사용할 때 코드를 작성하거나 읽을 때 쉽게 알아볼 수 있도록 합니다.

```
#define OCR2_DO                    59
                    ...
#define OCR2_DO2                   29
```

위의 표에서 나온 각 음계에 따른 OCR2 값을 #define으로 선언합니다. 이 값들은 음을 재생하기 위해서 타이머/카운터2의 OCR2 레지스터에 값을 넣을 때 사용됩니다. 단순하게 59 같은 숫자를 대입하면 쉽게 '도'라는 음과 연결시킬 수 없지만, OCR2_DO를 대입한다면 '도'음이 재생될 거라고 쉽게 알아볼 수 있습니다.

```
#define TIMER2_DIVIDER                    0b00000100
```

타이머/카운터2에서 클럭을 어떤 비율로 나눌 것인지 결정하는 TCCR2 레지스터의 CS22:0 비트의 값을 #define으로 선언합니다. 이렇게 선언해두고 나머지 부분에서 TCCR2 레지스터에 값을 대입할 때마다 TIMER2_DIVIDER를 사용해 대입한다면, 클럭의 비율을 바꿀 일이 있을 때 TIMER2_DIVIDER의 값만 바꾸면 다른 부분을 고칠 필요 없이 전체가 수정되므로 유용합니다.

```
unsigned char ucButton;
unsigned char ucButtonOld;
```

메인 함수에서 사용할 두 변수를 선언합니다. ucButton은 현재 while 루프에서 읽은 버튼의 값이고, ucButtonOld는 직전의 while 루프에서 읽은 버튼의 값입니다. 매번 while 루프의 마지막에서 ucButtonOld는 ucButton의 값으로 업데이트 됩니다. 이 두 변수는 타이머/카운터2 값을 버튼이 눌리거나 떼져서 버튼의 값이 바뀐 순간에만 바꾸기 위해서 사용합니다.

```
DDRA = 0b00000000;
DDRB |= 0b10000000;
DDRC = 0b01111111;
PORTC = 0b01111111;
```

포트 A, B, C를 설정합니다. 포트 A는 버튼이 연결되어 있는 포트이므로 DDRA에 모두 0을 넣어서 입력으로 설정합니다. 포트 B의 7번 핀(PB7)은 타이머/카운터2의 OC2 출력핀입니다. 타이머/카운터2에서 OC2 출력핀을 사용한다고 설정했더라도 DDRB의 7번 비트가 출력으로 설정되어 있어야 실제로 출력이 나오기 때문에, DDRB의 7번 비트를 1로 만들어주어 출력으로 설정합니다. 포트 C는 LED가 연결된 포트이므로 DDRC의 LED에 해당하는 7개 비트에 1을 넣어 출력으로 만들고, PORTC에도 7개 비트에 1을 넣어 LED를 모두 끕니다.

```
TCNT2 = 0;
OCR2 = 0;
TCCR2 = 0;
```

타이머/카운터2의 레지스터를 초기화합니다. 앞서 설명에서 언급된 레지스터 중 TCNT2, OCR2, TCCR2를 초기값인 0으로 만들어 줍니다. 이 시점에서 타이머/카운터2는 아직 동작하지 않습니다.

```
ucButtonOld = (~PINA) & BTN_MASK;
```

앞서 설명했듯이, ucButtonOld 변수에는 while문에서 직전에 읽어온 버튼의 값이 저장됩니다. 그런데 아직 while문에 진입하기 전이라 버튼 값을 읽기 전이기 때문에 ucButtonOld의 값을 ucButton로 바꾸는 것은 의미가 없고 예기치 않은 동작을 초래할 수 있습니다. 그래서 while문에 진입하기 직전에 한번 버튼 값을 읽어서 ucButtonOld에 저장합니다.

```
ucButton = (~PINA) & BTN_MASK;
```

이제부터는 while문 안쪽의 내용입니다. 포트 A로부터 버튼 값을 읽어와, ucButton에 저장합니다. ucButton에 저장된 버튼 값은 버튼에 해당하는 비트가 1이면 버튼이 눌린 상태, 0이면 눌리지 않은 상태임을 나타냅니다.

```
if(ucButtonOld != ucButton){
                    …
}
```

직전 while 루프에서 읽어온 버튼 값(ucButtonOld)과 방금 읽은 버튼 값(ucButton)을 비교해서, 서로 다른 경우에 if문의 안쪽 내용을 실행합니다. 버튼이 눌리는 순간이나 눌렸던 버튼이 떼지는 순간에만 타이머/카운터2의 설정을 바꾸기 위한 부분입니다.

```
if(ucButton & BTN_MODE){
        TCNT2 = 0;
        OCR2 = OCR2_DO;
        TCCR2 = (TIMER2_DIVIDER)|(1<<WGM21)|(1<<COM20);
}
                …
else if(ucButton & BTN_DOWN){
        TCNT2 = 0;
        OCR2 = OCR2_LA;
        TCCR2 = (TIMER2_DIVIDER)|(1<<WGM21)|(1<<COM20);
}
```

ucButton & BTN_MODE는 DRC의 Mode 버튼이 눌렸는지를 검사하는 부분입니다. ucButton을 BTN_MODE(0b00000001)와 Bitwise AND(&) 연산하면, ucButton의 0번 비트가 1일 경우 참, 0일 경우 거짓이 됩니다. Mode 버튼이 눌려있다면 타이머/카운터2의 값을 바꾸어 버저가 '도'음을 재생하게 합니다.

TCNT2는 0부터 시작하도록 0으로 초기화 하고, 값에 따라서 주파수가 달라지는 OCR2에는 '도'에 해당하는 미리 정의된 값을 대입합니다. TCCR2에서는 WGM21을 1로 만들어 CTC 모드로 설정하고, COM20을 1로 만들어 비교 매치가 일어날 때마다 OC2 핀을 토글하게 하고, TIMER2_DIVIDER에 정의된 값으로 CS22:0을 설정해 원하는 비율로 클럭을 나누도록 합니다. 우리 예제에서는 CS22:0을 0b100으로 설정해서 256으로 클럭을 나눕니다.

밑에 이어지는 else if문도 버튼과 OCR2 값만 바뀌었을 뿐 동일합니다. if문과 else if문을 통해서 MODE버튼 – 도, OK버튼 – 레, 좌 버튼 – 미, 상 버튼 – 파, 우 버튼 – 솔, 하 버튼 – 라를 재생하도록 합니다.

```
else{
        TCNT2 = 0;
        OCR2 = 0;
        TCCR2 = 0;
}
```

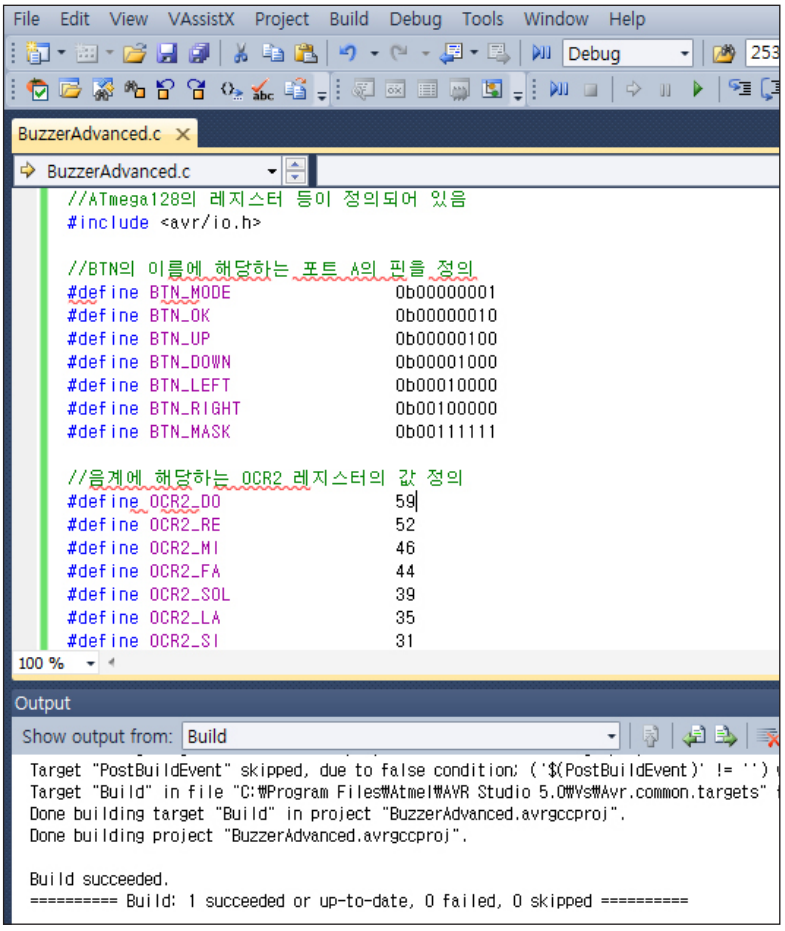
아무 버튼도 눌려있지 않다면, 타이머/카운터2 레지스터를 모두 0으로 만들어서 중지시킵니다. 이렇게 하면 버저의 울림이 멈추게 됩니다.

```
ucButtonOld = ucButton;
```

while 루프가 끝날 때마다 현재 루프에서 읽은 ucButton의 값을 ucButtonOld에 대입해, 다음 루프에서 비교하는 데에 사용합니다.

실행

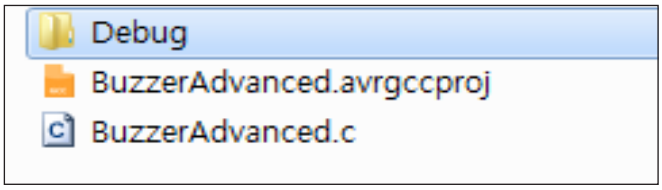
펌웨어를 다운로드하면, 버튼을 누를 때 해당하는 버저 음이 재생됩니다.



01 Build

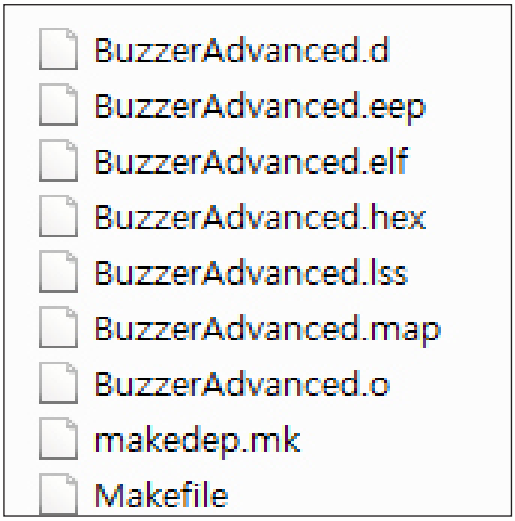
Build 메뉴를 선택하여 Build Solution을 클릭하여 Build 시킵니다.

1 succeeded 0 failed 가 나오면 에러없이 컴파일 완료됐다는 것을 의미합니다.



02 Debug 폴더

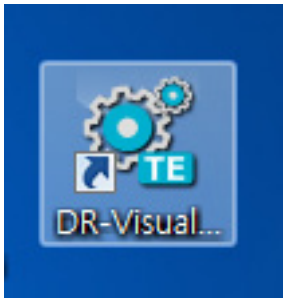
Build 후에 소스가 있는 폴더에 들어가면 Debug 폴더가 생성 되어있습니다. 클릭하여 폴더 안의 내용을 봅니다.



03 hex 확인

Debug 폴더 안의 hex 파일 확인합니다. Hex 파일이 제어기에 포팅시키는 파일입니다.

DR-Visual Logic 에서 펌웨어 복구를 실시합니다.

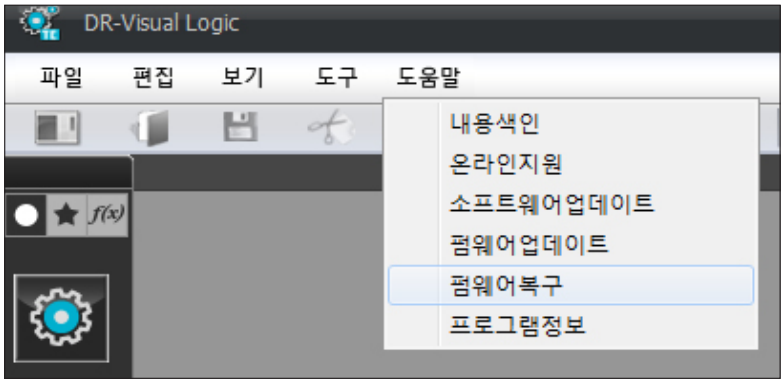


04 Visual Logic

DR-Visual Logic 을 실행시킵니다.

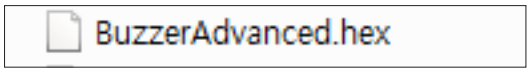
05 연결하기

COM 번호를 설정하고 연결하기를 클릭합니다.



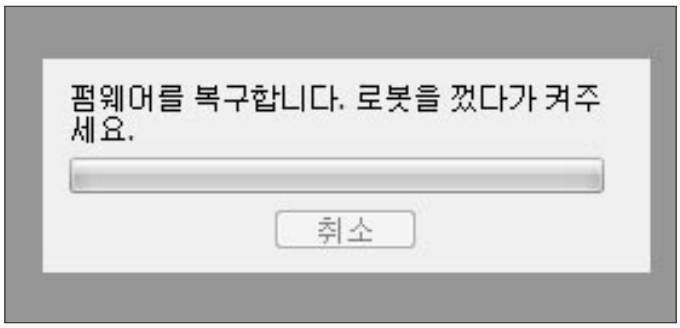
06 펌웨어 복구

도움말 메뉴에서 펌웨어 복구를 클릭합니다.



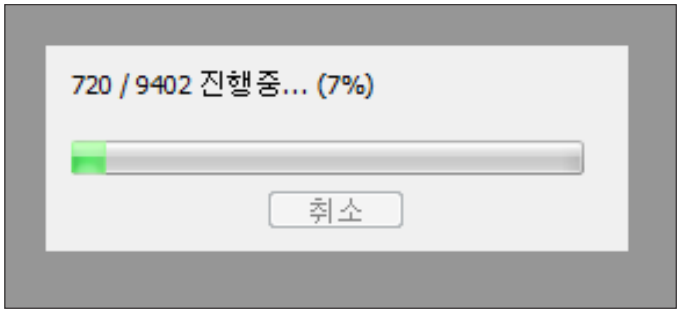
07 불러오기

Debug 폴더에서 Hex 파일을 불러옵니다



08 펌웨어복구

로봇을 껐다가 켜주세요 라는 문구가 나옵니다.
제어기의 전원버튼을 한번 껐다가 다시 켜주세요.



09 복구시작

펌웨어 복구가 시작됩니다.

10 연결

각각 버튼별로 누르면 버저에서 도레미파솔라 등의 음높이가 다른 음이 나옵니다.

EEPROM 활용하기

ATmega128의 EEPROM(Electrically Erasable Programmable Read-Only Memory)을 활용해서 데이터를 저장하고 읽어옵니다.

AVR 원리

메모리에는 크게 휘발성 메모리와 비휘발성 메모리가 있습니다. 휘발성 메모리는 전원이 없어지면 모든 데이터가 없어지므로 휘발성 메모리라고 합니다. 비휘발성 메모리는 그 반대로, 전원이 없어져도 데이터가 남아있기 때문에 다시 전원을 켜도 그 데이터를 다시 읽어와서 사용할 수 있습니다.

ATmega128에서 우리가 보통 사용하는 변수들은 휘발성 메모리인 SRAM에 저장되며, 전원을 끄면 모든 값이 사라지게 됩니다. 하지만 어떤 경우에는 직전까지 사용하던 데이터를 저장해두었다가 다음 전원을 켤 때 사용하고 싶을 수도 있을 것입니다. 이런 경우 ATmega128에는 EEPROM이라는 일종의 비휘발성 메모리가 내장되어 있어, 필요한 데이터를 자유롭게 읽고 쓰고 할 수 있습니다.

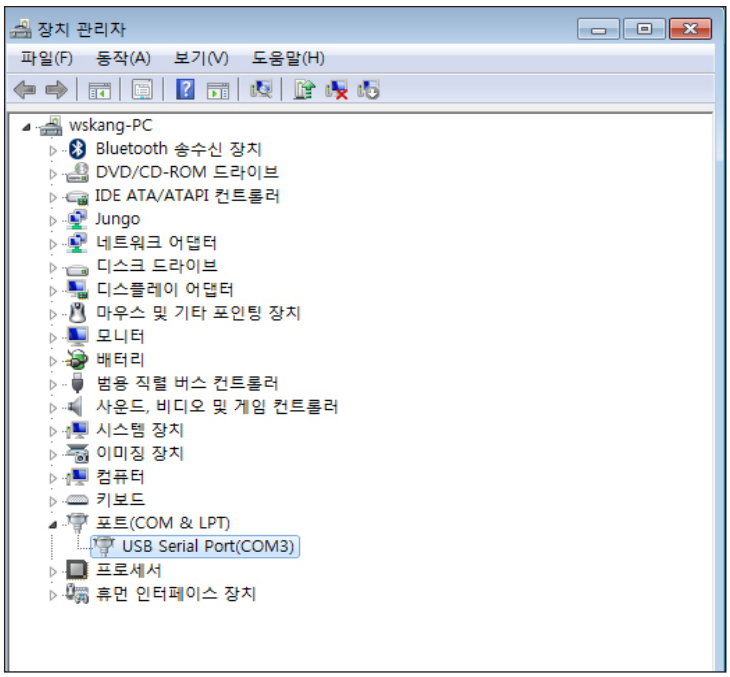
ATmega128의 내장 EEPROM 용량은 4kbyte로, 내장 SRAM과 동일한 크기입니다. 다만 EEPROM의 경우 100,000번 이상 쓰기/읽기를 반복하면 수명이 다 될 수 있으므로, 일반 변수처럼 자주 값을 바꾸면 오래 사용할 수 없습니다. 가령 main 함수의 while문에서 한번 반복마다 EEPROM에 값을 저장한다면 얼마 지나지 않아 EEPROM이 망가져서 오동작할 것입니다. 따라서, EEPROM에는 꼭 필요한 상황에만 값을 저장하도록 신경 써서 프로그램을 작성해야 합니다.

EEPROM을 읽고 쓰는 데에 관련된 레지스터로 EEARH, EEARL, EEDR, EECR 등의 레지스터가 있습니다. 그러나 사용자는 이 세세한 레지스터 설정에 대해서 알 필요가 없습니다. AVR Studio에서는 사용자가 eeprom 데이터를 쉽게 읽고 쓸 수 있도록 avr/eeprom.h 헤더 파일로 관련 함수를 이미 제공하기 때문입니다.

하드웨어 준비하기

01 연결

DRC 제어기 아답터로 전원을 연결하고, USB to Serial Cable 로 PC 와 DRC 를 연결합니다.

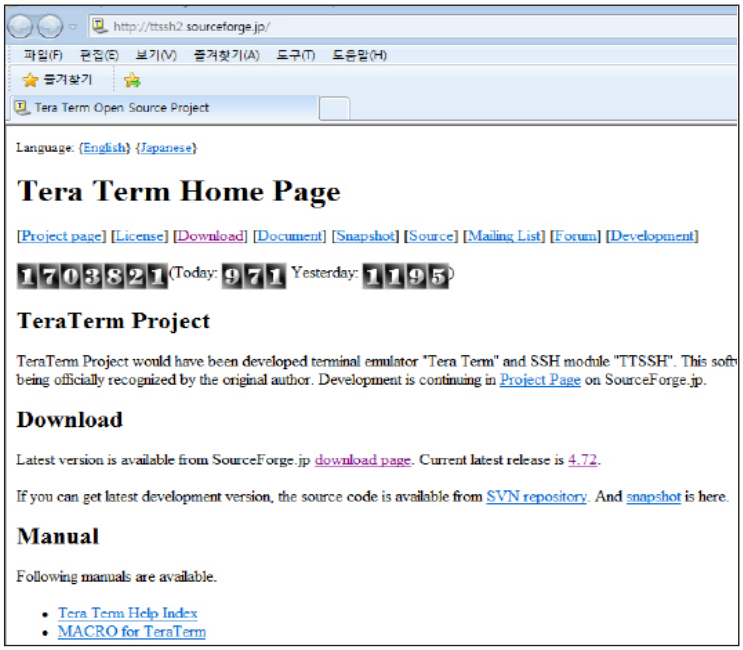


02 COM port 연결

내컴퓨터 〉 속성 〉 장치관리자에서 COM 연결 확인합니다.

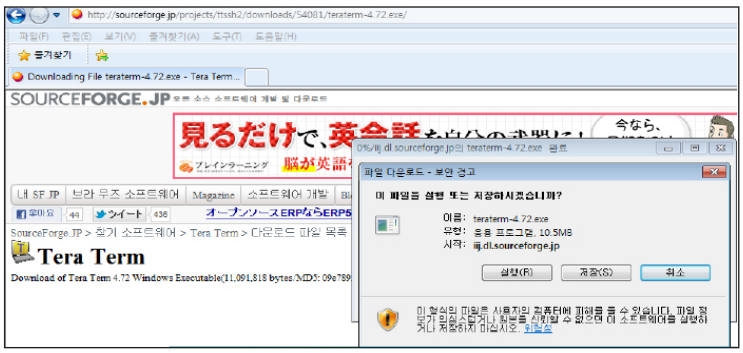
01 주소 검색

인터넷 포털 사이트에서 “tera term”을 검색합니다.



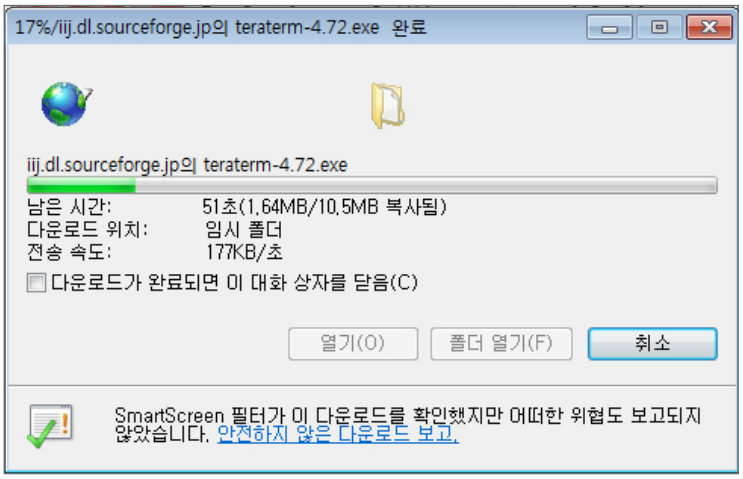
02 홈페이지 접속

Tera Term 홈페이지에 접속하여 “Download page”를 클릭합니다.



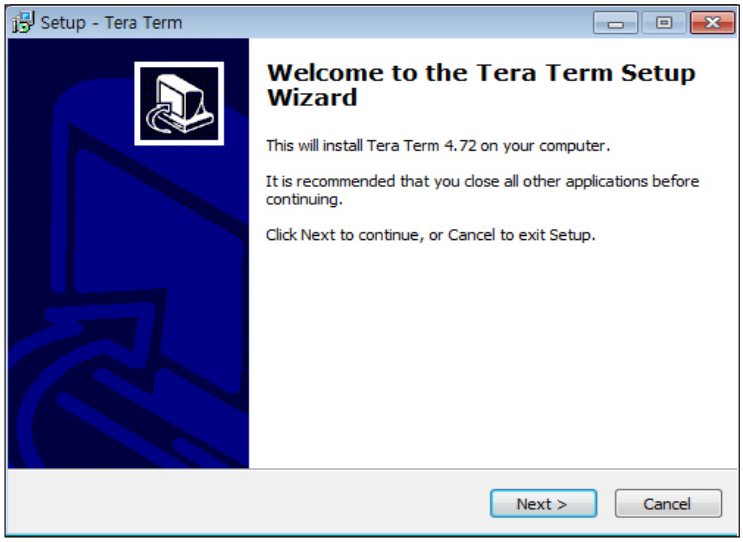
03 파일 다운로드

“저장”을 눌러 파일을 다운로드 받습니다.



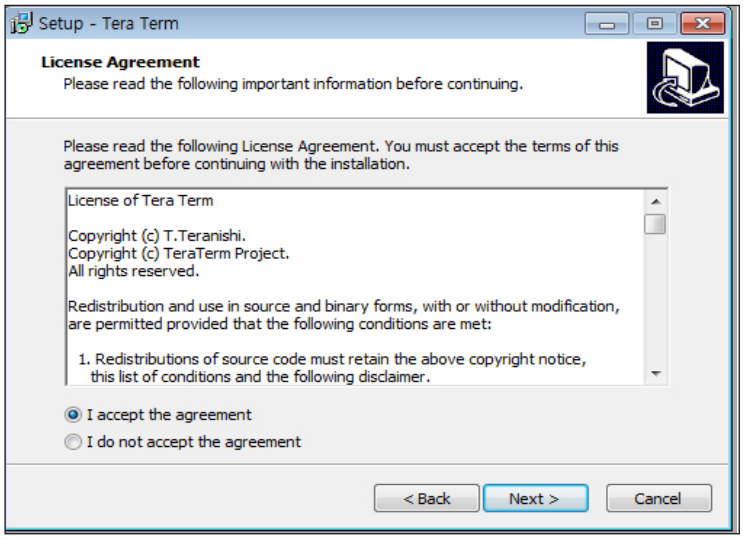
04 파일 다운로드 시작

설치 파일이 다운로드됩니다.



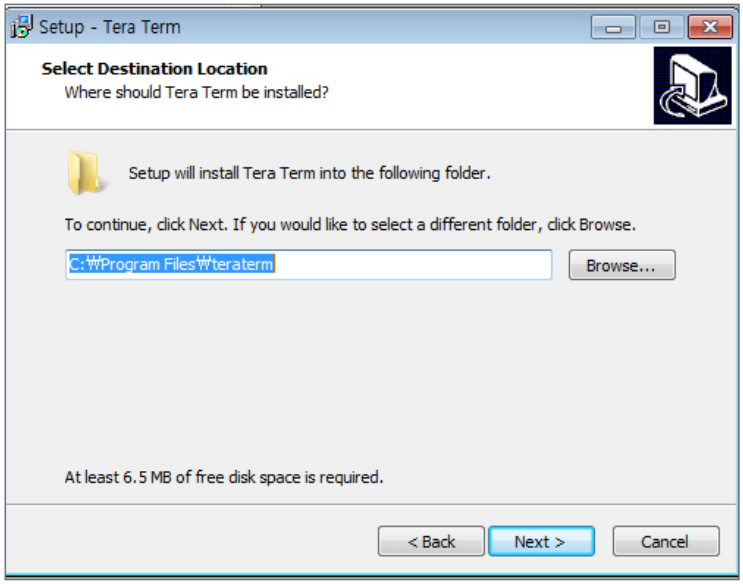
05 설치 마법사 실행

"Next"를 누르면 설치 마법사가 실행됩니다.



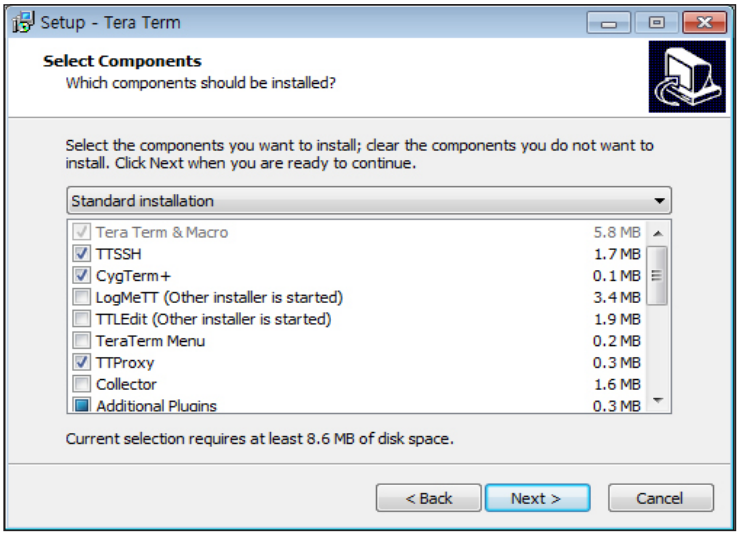
06 약관 동의

"I accept the agreement"를 체크하고 "Next"를 클릭합니다.



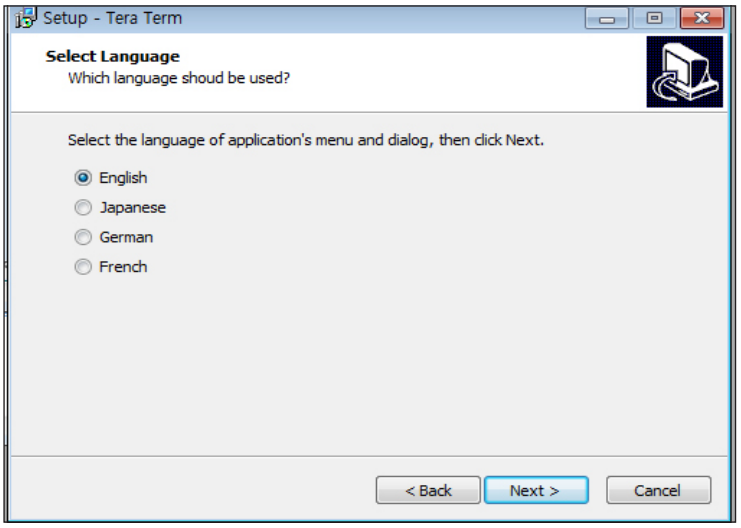
07 설치 폴더 확인

프로그램을 설치할 폴더를 확인하고 "Next"를 클릭합니다.



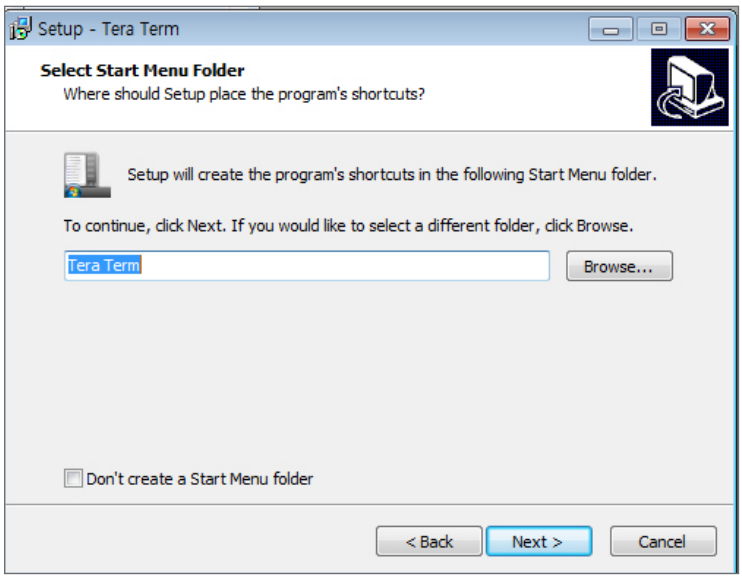
08 구성요소 선택

"Next"를 클릭합니다.



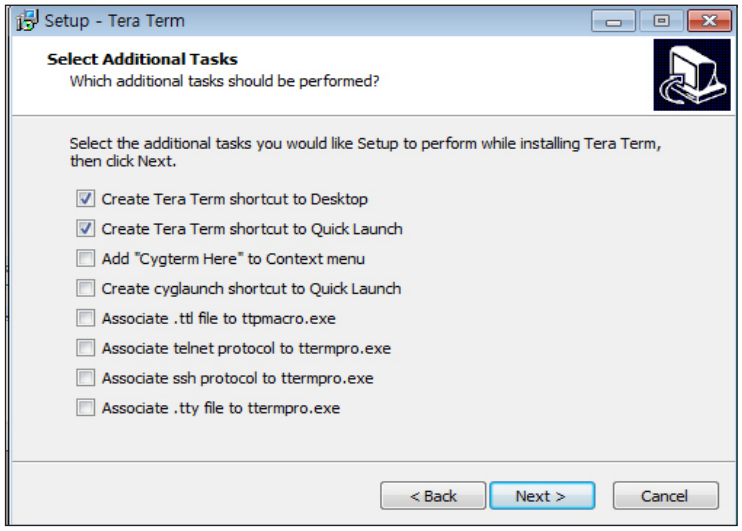
09 언어 선택

언어 선택 후 "Next"를 클릭합니다.



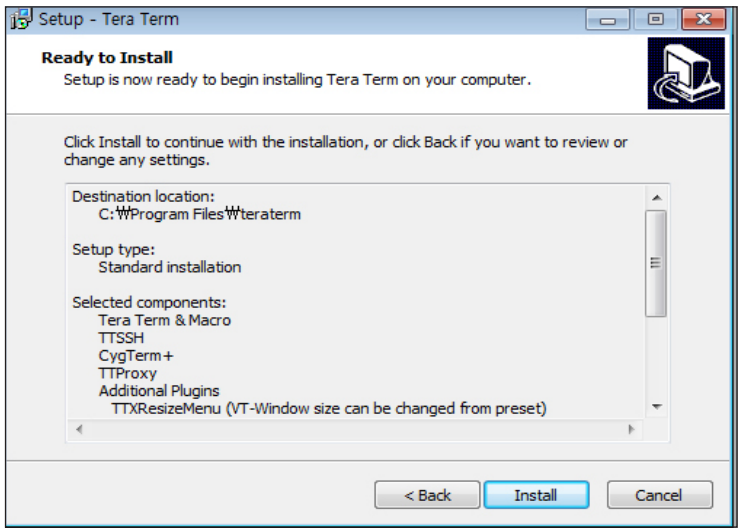
10 시작 메뉴 폴더 선택

시작 메뉴 폴더 선택 후 “Next”를 클릭합니다.



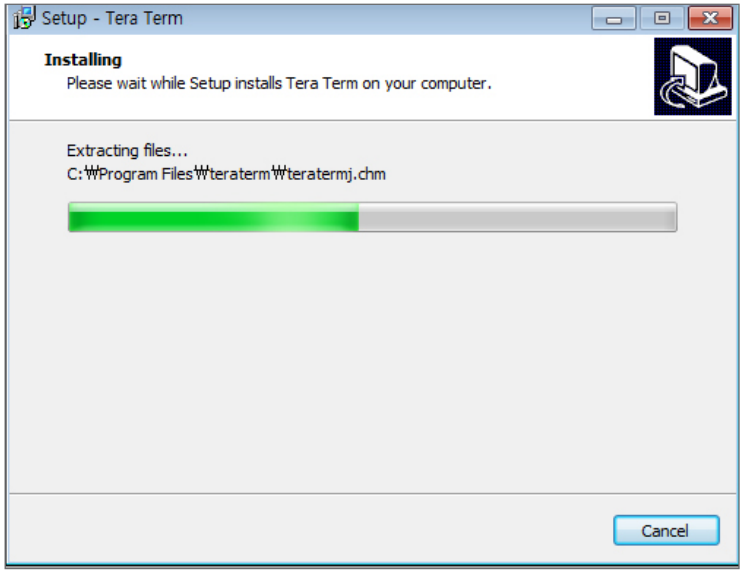
11 추가 기능 선택

원하는 추가 기능을 선택 후 “Next”를 클릭합니다.



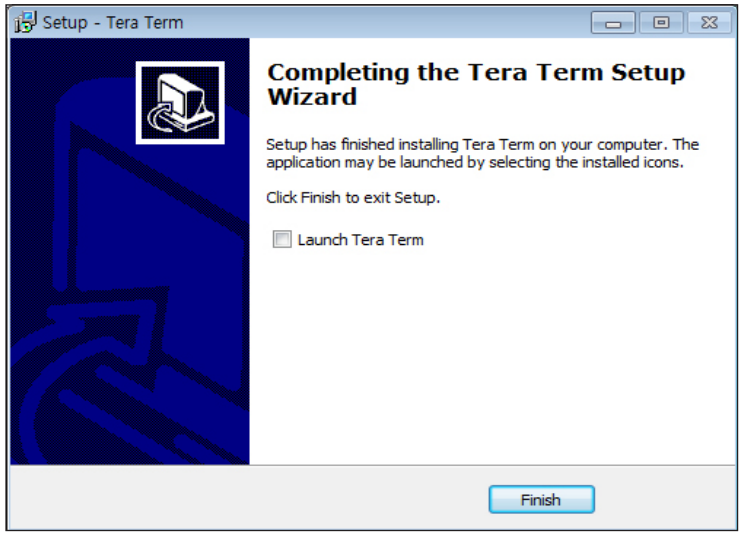
12 설치 시작

“Install”을 클릭하면 설치가 시작됩니다.



13 설치 확인

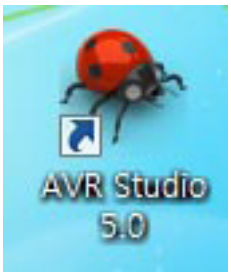
프로그램이 설치됩니다



14 설치 완료

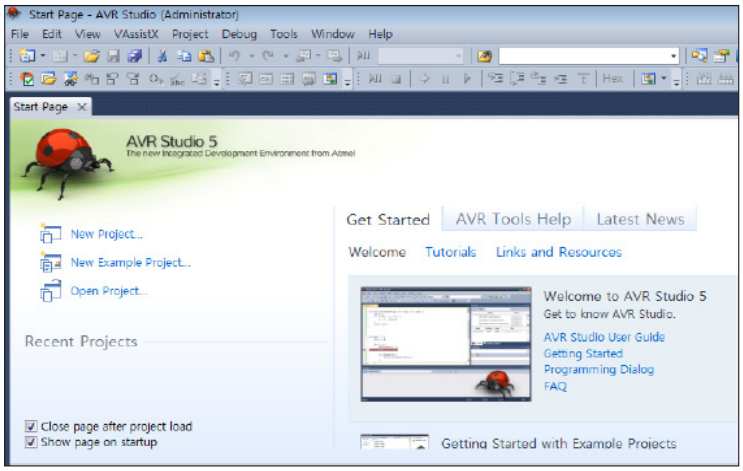
"Finish"를 클릭하면 설치가 완료됩니다.

AVR Studio 실행 및 코딩 준비



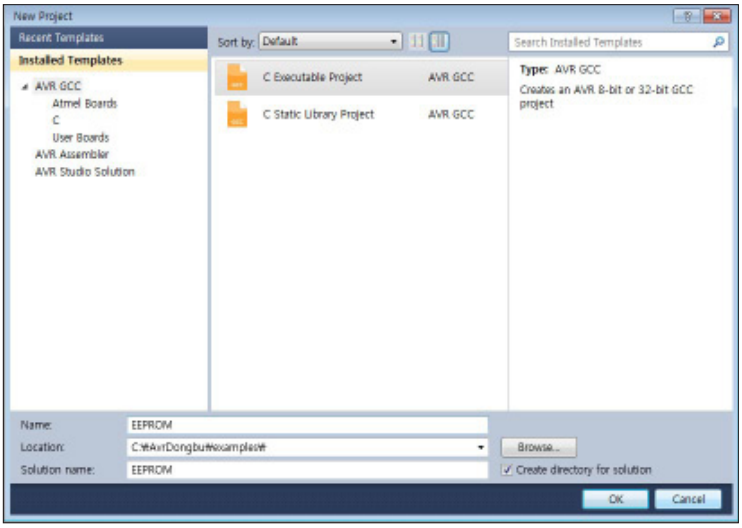
01 실행

AVR Studio 아이콘을 클릭하여 실행합니다.



02 New Project

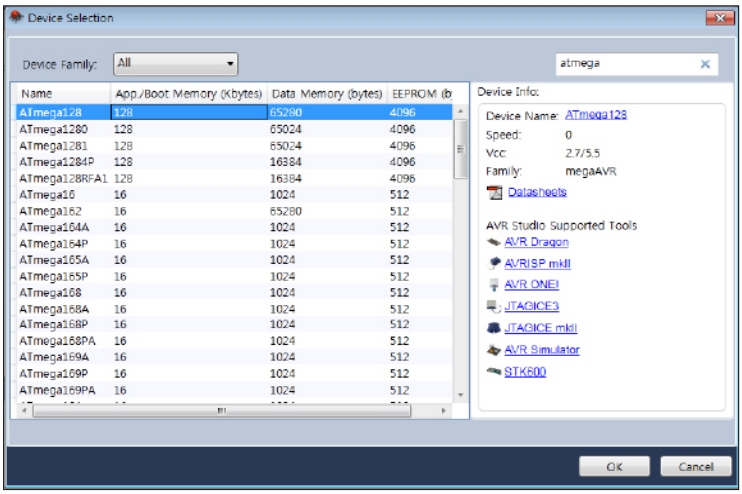
New Project 를 클릭합니다.



03 GCC 선택

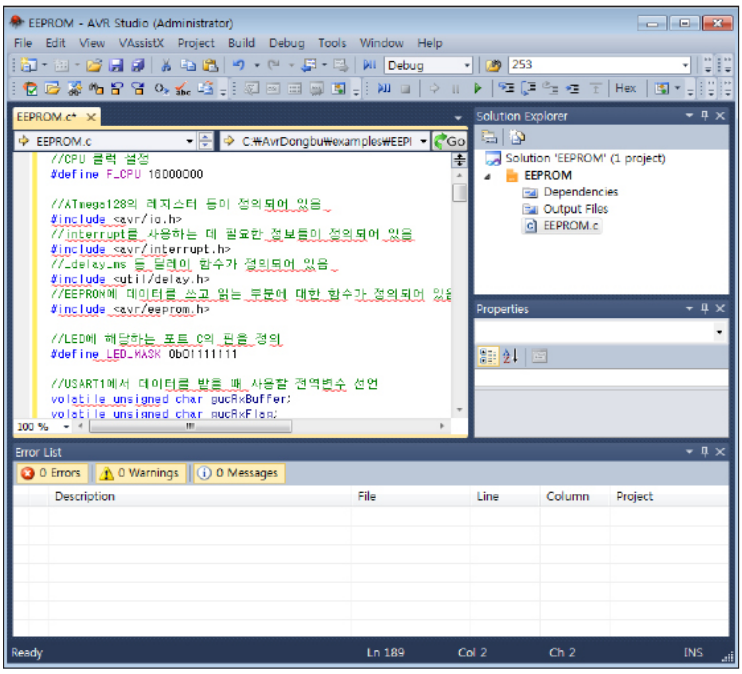
C Executable Project 를 선택합니다.

Name 을 “EEPROM”로 입력하고, Create directory for solution 을 클릭하고, OK 버튼을 누릅니다.



04 ATmega 선택

우측 상단에 atmega 를 검색하여,
리스트중에 ATmega128 을 선택합니다.



05 Editor

편집창에 코딩을 시작합니다.
아래 예제를 따라하세요.

예제

이번 Chapter의 예제는 Chapter 7의 예제와 거의 동일합니다. USB-to-Serial 혹은 PC 뒷면의 COMPORT와 DRC를 통신 케이블로 연결하고, 하이퍼터미널을 이용해서 보내는 글자에 따라서 DRC의 LED를 제어하는 예제입니다. 다만 다른 점은, EEPROM에 현재 어떤 LED가 켜져 있는지 저장하며, 전원을 껐다가 켜도 마지막에 켜져 있던 LED가 켜진다는 것입니다.

소스가 Chapter 7의 USART.c와 거의 같기 때문에, USART.c에 비교해서 변경된 부분을 빨간 글씨로 표시하고, 변경된 부분만 설명합니다. 소스 코드의 전체적인 내용을 알고 싶으시다면 Chapter 7의 예제 설명을 한번 더 읽고 이번 Chapter를 읽으시면 됩니다.

EEPROM.c

```
//CPU 클럭 설정
#define F_CPU 16000000

//ATmega128의 레지스터 등이 정의되어 있음
#include <avr/io.h>
//interrupt를 사용하는 데 필요한 정보들이 정의되어 있음
#include <avr/interrupt.h>
//_delay_ms 등 딜레이 함수가 정의되어 있음
#include <util/delay.h>
//EEPROM에 데이터를 쓰고 읽는 부분에 대한 함수가 정의되어 있음
#include <avr/eeprom.h>

//LED에 해당하는 포트 C의 핀을 정의
#define LED_MASK 0b01111111

//USART1에서 데이터를 받을 때 사용할 전역변수 선언
volatile unsigned char gucRxBuffer;
volatile unsigned char gucRxFlag;

/**
* USART1 수신 완료 인터럽트의 서비스 루틴이다.
* UDR1으로부터 데이터를 읽어 gucRxBuffer에 저장하고
* gucRxFlag를 1로 설정한다.
*
* @author       Dongbu Robot
* @param        없음
* @return       없음
* @version      1.0.0.0
*/
ISR(USART1_RX_vect){
        gucRxBuffer = UDR1;
        gucRxFlag = 1;
}

/**
* USART1의 초기화 함수다. USART1을 활성화하고
* ulBaudRate에 따라서 보드 레이트를 설정한 후
* 전역 변수를 초기화 한다.
*
* @author       Dongbu Robot
* @param        ulBaudRate: USART1을 작동시킬 보드 레이트
* @return       없음
* @version      1.0.0.0
*/
```

```
void USART1_Init(unsigned long ulBaudRate){
        unsigned long ulTemp;

        //부트로더와의 충돌을 막기 위해 잠시 대기
        _delay_ms(10);

        //송신, 수신, 수신 인터럽트 활성화
        UCSR1B = ((1 << TXEN1) | (1 << RXEN1) | (1 << RXCIE1));
        //2배속 전송 모드 설정
        UCSR1A = (1 << U2X1);

        //클럭과 Baud Rate을 사용해 UBRR1에 들어갈 값 계산
        ulTemp = F_CPU / (ulBaudRate * 8L) - 1;

        //UBRR1에 계산한 값 설정
        UBRR1H = (ulTemp >> 8) & 0x0F;
        UBRR1L = ulTemp & 0xFF;

        //전역 변수 초기화
        gucRxBuffer = 0;
        gucRxFlag = 0;

        return;
}

/**
* USART1으로 문자 하나를 전송한다.
*
* @author       Dongbu Robot
* @param        cData: 전송할 문자 데이터
* @return       없음
* @version      1.0.0.0
*/
void USART1_PutChar(char cData){
        //송신 데이터 레지스터가 빌 때 까지 대기
        while(!(UCSR1A & (1 << UDRE1)));
        //송신 데이터 레지스터에 데이터 입력
        UDR1 = cData;

        return;
}

/**
```

```
* USART1으로 문자열을 전송한다.
*
* @author       Dongbu Robot
* @param        pcStr: 전송할 문자열 중 첫 문자의 주소
* @return       없음
* @version      1.0.0.0
*/
void USART1_PutString(char* pcStr){
        //'₩0'을 만날 때 까지 USART1_PutChar를 반복
        while(*pcStr)
                USART1_PutChar(*pcStr++);
}

int main(void)
{
        //8비트 변수 선언
        unsigned char ucLED;

        //포트 C의 입출력 방향 설정
        DDRC = 0b01111111;
        //포트 C의 출력 값 설정
        PORTC = 0b01111111;

        //전체 인터럽트를 비활성화
        cli();

        //USART1 초기화
        USART1_Init(115200);

        //전체 인터럽트를 활성화
        sei();

        //초기화 완료 메시지 전송
        USART1_PutString("Serial Initialized₩r₩n");

        //EEPROM의 0번 주소로부터 데이터를 읽어옴
        ucLED = eeprom_read_byte((uint8_t*)0);

        //데이터가 원하는 것 중 하나가 아니면 0b00000001로 초기화
        switch(ucLED){
                case 0b01000000:
                case 0b00100000:
                case 0b00010000:
                case 0b00001000:
```

```
            case 0b00000100:
            case 0b00000010:
            case 0b00000001:
                    break;
            default:
                    ucLED = 0b00000001;
                    eeprom_write_byte((uint8_t*)0, ucLED);
                    break;
}

//포트 C의 LED를 켬
PORTC = (~ucLED) & LED_MASK;

//while 안의 내용을 무한 반복
while(1)
{
        //데이터를 받았다면 실행
        if(gucRxFlag){
                //받은 데이터가 'z'나 'Z'라면 실행
                if(gucRxBuffer == 'z' || gucRxBuffer == 'Z'){
                        //켤 LED 비트를 오른쪽으로 이동 후 EEPROM에 데이터 저장
                        if(ucLED > 0b00000001){
                                ucLED >>= 1;
                                eeprom_write_byte((uint8_t*)0, ucLED);
                        }
                        else{
                                ucLED = 0b01000000;
                                eeprom_write_byte((uint8_t*)0, ucLED);
                        }
                        PORTC = (~ucLED) & LED_MASK;

                        //메시지 전송
                        USART1_PutString("LED Moved by ₩'z₩'₩r₩n");
                }
                //받은 데이터가 'x'나 'X'라면 실행
                else if(gucRxBuffer == 'x' || gucRxBuffer == 'X'){
                        //켤 LED 비트를 왼쪽으로 이동 후 EEPROM에 데이터 저장
                        if(ucLED < 0b01000000){
                                ucLED <<= 1;
                                eeprom_write_byte((uint8_t*)0, ucLED);
                        }
                        else{
                                ucLED = 0b00000001;
                                eeprom_write_byte((uint8_t*)0, ucLED);
                        }
```

```
                                        PORTC = (~ucLED) & LED_MASK;

                                        //메시지 전송
                                        USART1_PutString("LED Moved by ₩'x₩'₩r₩n");
                                }
                                //데이터 수신 플래그 초기화
                                gucRxFlag = 0;
                        }
                }

                //함수의 형태에 맞게 정수값 반환
                return 1;
        }
```

예제설명

```
#include <avr/eeprom.h>
```

앞서 언급한대로, avr/eeprom.h 헤더 파일에는 EEPROM에서 데이터를 읽고 쓰기 위한 함수들이 정의되어 있습니다. 이 헤더에서 선언되어 있는 주요 함수들은 아래와 같습니다.

```
uint8_t eeprom_read_byte (const uint8_t *__p);
uint16_t eeprom_read_word (const uint16_t *__p);
uint32_t eeprom_read_dword (const uint32_t *__p);
```

위 함수들은 EEPROM의 __p라는 주소에서 각각 unsigned char(8비트), unsigned int(16비트), unsigned long int(32비트) 데이터를 읽어오기 위한 함수입니다. 함수 원형에서 나오는 uint8_t, uint16_t, uint32_t는 각각 8비트, 16비트, 32비트 부호 없는 정수형을 나타내는 것으로, unsigned char, unsigned int, unsigned long int와 동일합니다. uint8_t 등의 변수 타입은 이 변수가 몇 비트로 이루어져 있는지 확실히 명시하고 싶을 때 사용됩니다. int와 long int 등의 변수 타입은 시스템과 컴파일러에 따라서 뜻하는 길이가 서로 다를 수도 있기 때문입니다.
위 세 함수들의 파라미터에는 읽고자 하는 EEPROM의 주소 값, 즉 포인터가 들어가며, 그 주소의 데이터 값이 리턴 되어 돌아옵니다.

```
void eeprom_write_byte (uint8_t *__p, uint8_t __value);
void eeprom_write_word (uint16_t *__p, uint16_t __value);
void eeprom_write_dword (uint32_t *__p, uint32_t __value);
```

위 함수들은 EEPROM의 __p라는 주소에 unsigned char(8비트), unsigned int(16비트), unsigned long int(32비트) 데이터 __value를 저장하기 위한 함수입니다. 파라미터에는 자료를 쓰고자 하는 EEPROM의 주소 값, 즉 포인터가 들어가며, 그 다음 기록될 데이터가 들어갑니다.
EEPROM을 읽고 쓰는 함수가 어떻게 사용되는 지는 아래에서 실제 예로 볼 수 있습니다.

```
ucLED = eeprom_read_byte((uint8_t*)0);
```

eeprom_read_byte 함수를 사용해 실제로 eeprom의 데이터를 읽는 부분입니다. 파라미터로 0을 사용한 것은 EEPROM의 0번 주소에서 unsigned char 데이터를 읽어올 것을 뜻하고, (uint8_t*)를 0에 붙인 것은 0이라는 값의 타입을 함수 원형의 데이터 타입에 맞게 변환시켜주는 부분입니다. 이 코드를 거치면 변수 ucLED에 EEPROM 0번 주소의 데이터 값이 저장됩니다.

```
switch(ucLED){
        case 0b01000000:
        case 0b00100000:
        case 0b00010000:
        case 0b00001000:
        case 0b00000100:
        case 0b00000010:
        case 0b00000001:
                break;
        default:
                ucLED = 0b00000001;
                eeprom_write_byte((uint8_t*)0, ucLED);
                break;
}
```

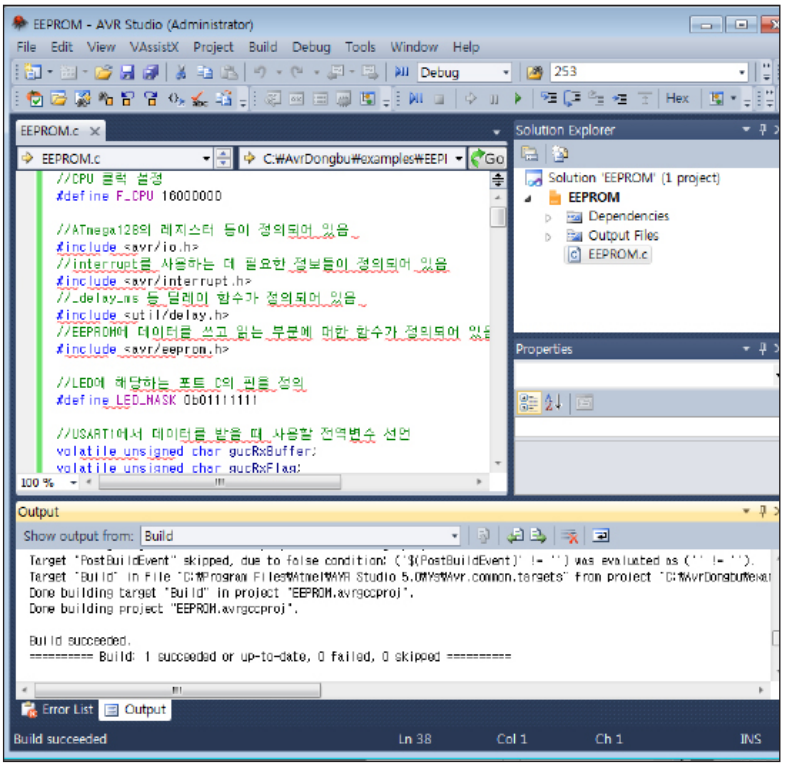
EEPROM으로부터 읽어온 데이터를 검사하는 부분입니다. 프로그램을 퓨징하고 처음 켠 상황이거나 EEPROM에 다른 프로그램이나 노이즈에 의해서 의도치 않은 값이 들어간 경우에는 ucLED에 예상치 못한 값이 저장될 것이고, 이것을 그대로 두면 프로그램이 오동작할 수 있습니다. 그래서 switch-case문을 사용하여 ucLED 값이 정상 범주(0b01000000~0b00000001) 안에 들면 값을 바꾸지 않고 넘어가고, 그 안에 들지 못하면 ucLED를 0b00000001로 초기화 하고 eeprom_write_byte 함수를 사용해 EEPROM 0번 주소에 ucLED 값을 저장합니다.

```
if(ucLED > 0b00000001){
        ucLED >>= 1;
        eeprom_write_byte((uint8_t*)0, ucLED);
}
else{
        ucLED = 0b01000000;
        eeprom_write_byte((uint8_t*)0, ucLED);
}
            …
if(ucLED < 0b01000000){
        ucLED <<= 1;
        eeprom_write_byte((uint8_t*)0, ucLED);
}
else{
        ucLED = 0b00000001;
        eeprom_write_byte((uint8_t*)0, ucLED);
}
```

이 부분은 사용자로부터의 키보드 입력에 따라서 특정 문자가 들어오면 ucLED 값을 바꾸어 켜질 LED의 위치를 변경하는 부분입니다. 이 부분에서 ucLED 값이 바뀌는 부분 밑에 모두 eeprom_write_byte((uint8_t*)0, ucLED); 가 추가되어, ucLED 값이 바뀔 때마다 EEPROM의 0번 주소에 현재 ucLED 값을 저장하도록 변경 되었습니다. 이 부분이 추가됨으로써, 전원을 껐다가 켜면 무조건 Servo LED부터 시작했던 저번 예제와 달리 이번 예제에서는 꺼지기 직전에 켜져 있던 LED가 다시 켜지면서 그 위치에서 다시 시작하게 됩니다.

실행

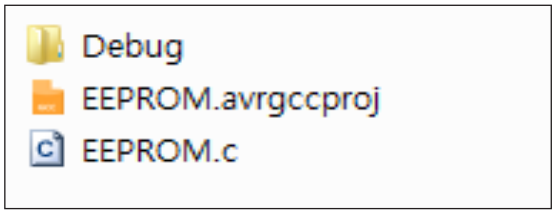
펌웨어를 실행하고 Tera Term을 이용해 Z,X키를 입력하면 켜진 LED가 이동하면서 터미널 창에 메시지가 출력됩니다. 이때 전원을 껏다가 켜도 마지막 위치에서 다시 시작합니다. ㅏ



01 Build

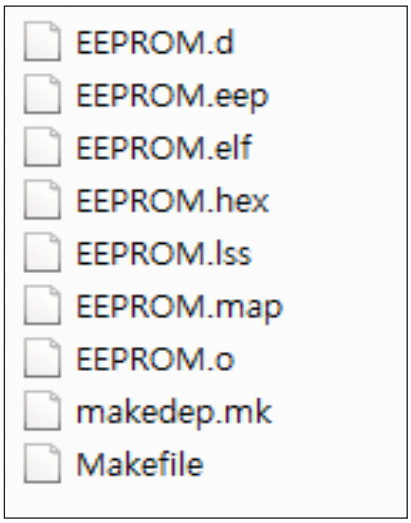
Build 메뉴를 선택하여 Build Solution 을 클릭하여 Build 시킵니다.

1 succeeded 0 failed 가 나오면 에러없이 컴파일 완료됐다는 것을 의미합니다.



02 Debug 폴더

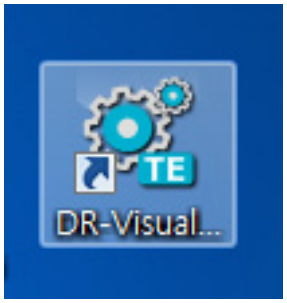
Build 후에 소스가 있는 폴더에 들어가면 Debug 폴더가 생성 되어있습니다. 클릭하여 폴더 안의 내용을 봅니다.



03 hex 확인

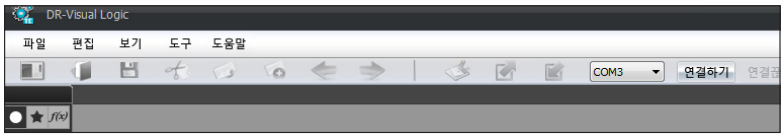
Debug 폴더 안의 hex 파일 확인합니다. Hex 파일이 제어기에 포팅시키는 파일입니다.

DR-Visual Logic 에서 펌웨어 복구를 실시합니다.



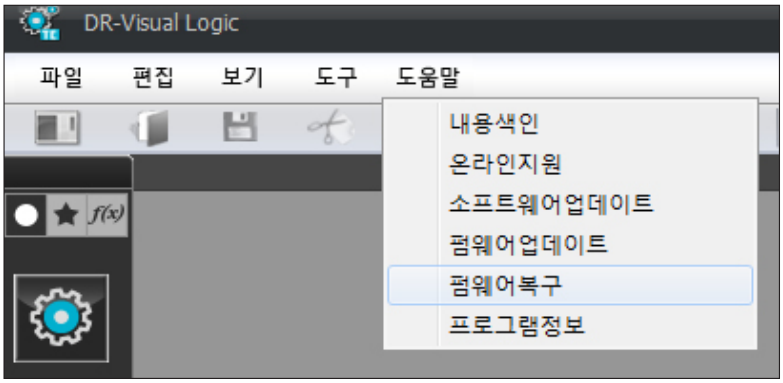
04 Visual Logic

DR-Visual Logic 을 실행시킵니다.



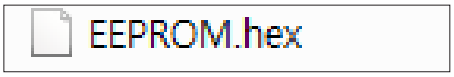
05 연결하기

COM 번호를 설정하고 연결하기를 클릭합니다.



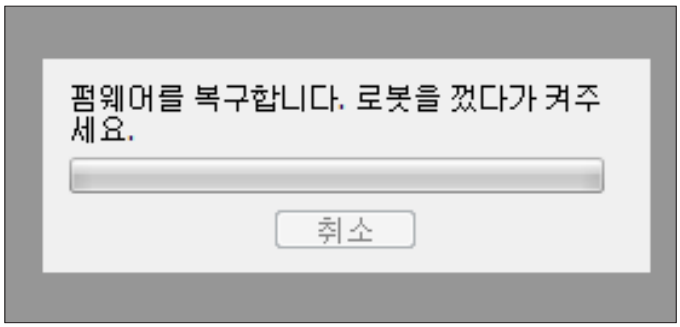
06 펌웨어 복구

도움말 메뉴에서 펌웨어 복구를 클릭합니다.



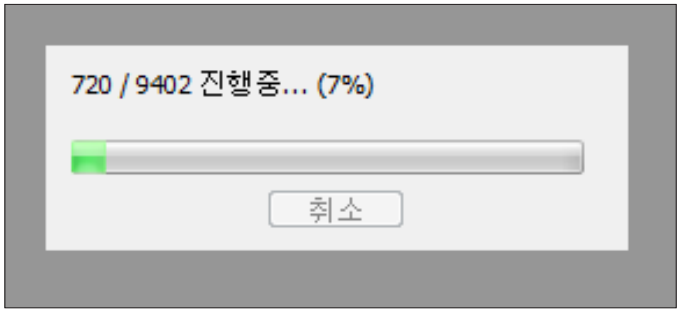
07 불러오기

Debug 폴더에서 Hex 파일을 불러옵니다



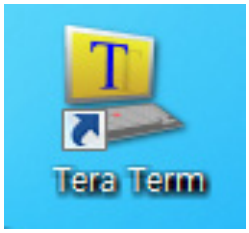
08 펌웨어복구

로봇을 껐다가 켜주세요 라는 문구가 나옵니다.
제어기의 전원 버튼을 한번 껐다가 다시 켜주세요.



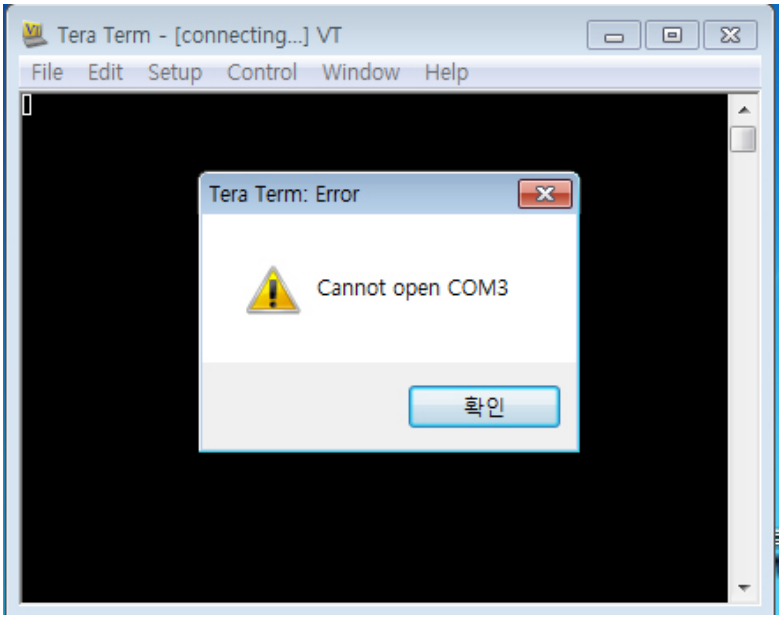
09 복구시작

펌웨어 복구가 시작됩니다.



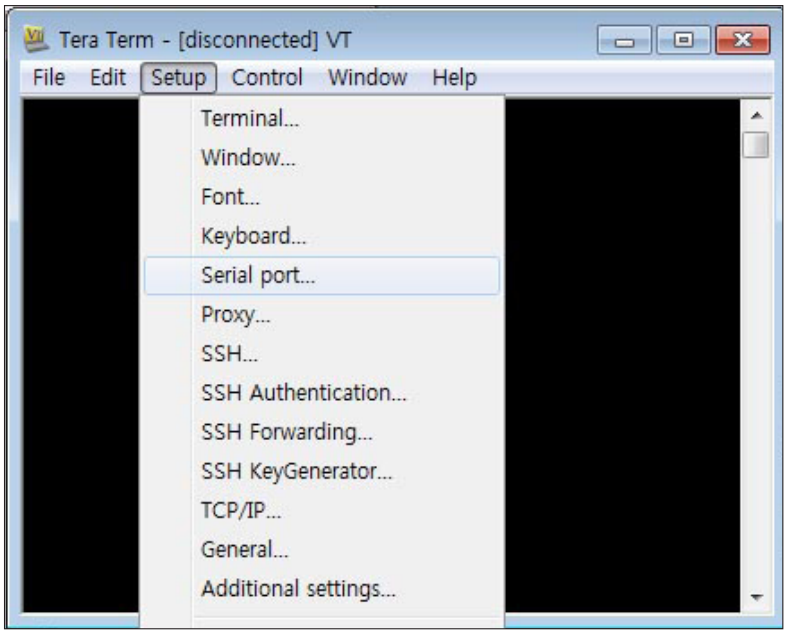
10 실행

Tera Term 아이콘을 클릭하여 프로그램을 실행시킵니다.



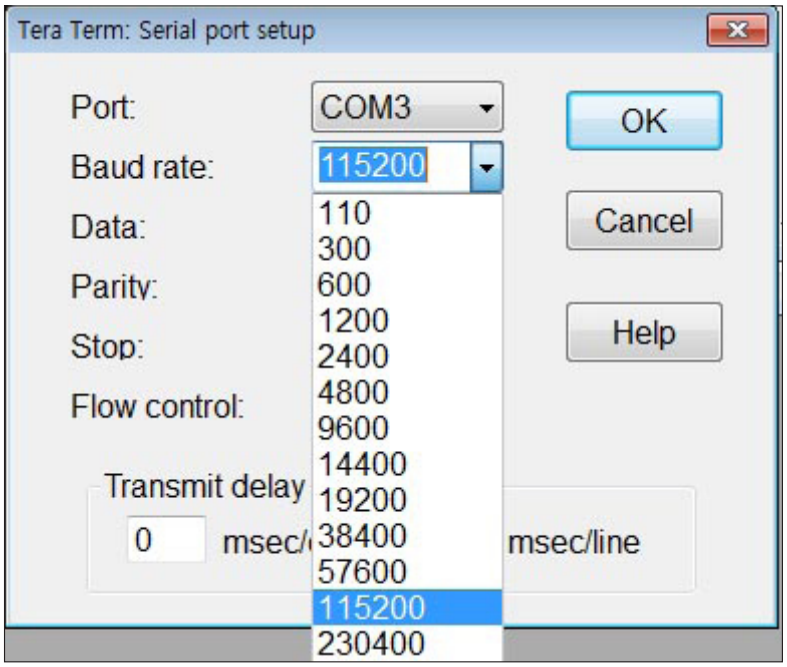
11 열기

실행하면 바로 연결되지 않습니다.
확인버튼을 클릭합니다.



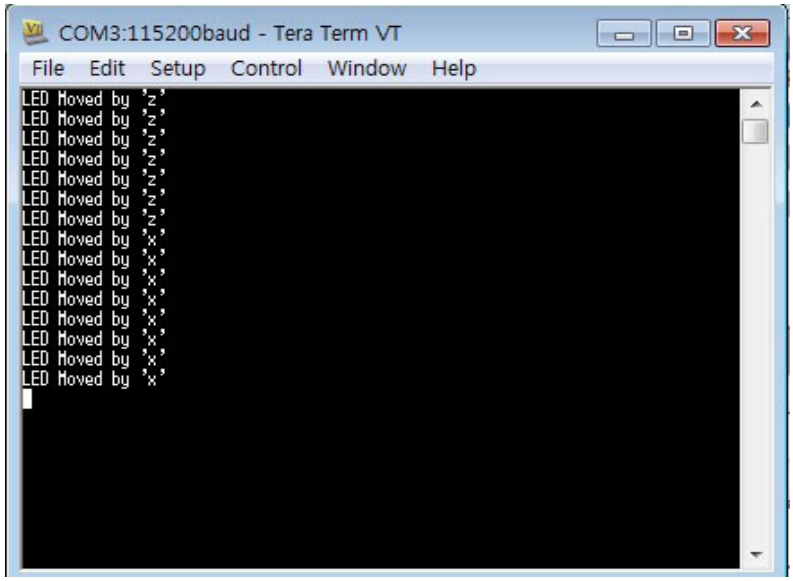
12 Setup

메뉴중에 Setup 메뉴를 클릭하여 Serial port 를 선택합니다.



13 Baud rate

Baud rate 를 115200 으로 선택합니다. 제어기 최대 통신속도입니다.



14 실행

펌웨어를 실행하고 Tera Term(하이퍼터미널)을 이용해 z, x키를 입력하면 켜진 LED가 이동하면서 터미널 창에 메시지가 출력되는 것을 확인할 수 있습니다.

PART 03

Firmware 익히기

HOVIS

가속도/자이로 센서 활용하기

DRC에 가속도/자이로를 부착하고 센서 값을 읽어 기울기, 각속도 감지 등에 활용해봅니다.

AVR 원리

이번 Chapter에서는 ATmega128의 GPIO 핀을 사용해서 I2C 통신을 하여 가속도/자이로 센서 값을 읽습니다. I2C 통신에 관련된 내용은 모두 libaccgyro.a 라이브러리에 의해서 제공되며, 사용자는 센서 값을 읽어서 활용하기만 하면 됩니다.

H/W 원리

I2C 통신

ATmega128의 GPIO 포트는 LED 켜기, 버튼 읽기 등의 단순한 입출력 작업에도 사용될 수 있지만, GPIO 포트를 사용한 통신도 가능합니다. 이러한 통신 중에 하나가 I2C 통신입니다. I2C는 필립스에서 개발한 직렬 통신 프로토콜이며, 아이-투-씨 내지는 아이-스퀘어-씨라고 읽습니다. I2C 통신은 직렬 데이터(SDA)와 직렬 클럭(SCL)이라는 두 개의 라인을 이용해 통신을 하며, 통신 전압은 최대 5V, 일반적으로는 3.3V입니다. DRC에서는 3.3V 전압으로 I2C 통신을 합니다.

I2C 통신은 마스터와 슬레이브로 나누어져 이루어지며, 보통 명령을 내리고 값을 읽거나 쓰는 쪽이 마스터, 그 명령에 따라서 요청한 값을 돌려주거나 받은 값을 저장하는 쪽이 슬레이브입니다. 우리 하드웨어 구성으로 따지자면 DRC의 ATmega128 칩이 마스터, 가속도/자이로 모듈의 가속도 센서 칩과 자이로 센서 칩이 슬레이브라고 할 수 있습니다.

SCL 핀은 마스터가 제어하며, 0과 1을 반복하면서 클럭 신호를 슬레이브에게 내립니다. 그러면서 클럭에 맞추어 SDA 핀을 정해진 약속에 따라서 어떤 때는 마스터가 출력, 슬레이브가 입력으로, 어떤 때는 슬레이브가 출력, 마스터가 입력으로 받는 식으로 데이터를 주고 받습니다. USART에서 TX와 RX핀이 따로따로 있는 것을 생각하면, I2C는 TX와 RX를 한 핀으로 해결하는 셈입니다.

I2C 통신의 슬레이브들은 각자 고유한 7비트 길이의 슬레이브 주소를 가지며, 통신이 이루어질 때에 슬레이브 주소가 다르면 응답하지 않습니다. 그렇기 때문에 하나의 마스터에 서로 주소가 다른 여러 슬레이브를 연결해서 같은 라인에서 통신하는 것이 가능하며, 그래서 I2C를 일종의 버스(bus)라고도 합니다. 실제로 동부로봇의 가속도/자이로 모듈도 두 개의 서로 주소가 다른 슬레이브를 같은 라인에 연결해 놓았습니다. 7비트 길이의 슬레이브 주소면 128개의 주소가 가능한데, 이중 16개는 특정한 용도로 예약이 되어 있어서 실제로는 112개의 주소를 사용할 수 있습니다.

대부분 I2C를 사용하는 칩들은 슬레이브 주소가 생산 시에 정해져서 나옵니다. 사용자가 주소를 임의대로 바꿀 수 없다는 뜻입니다. 그런데 만약 동시에 사용해야 하는 칩이 서로 슬레이브 주소가 겹친다면, 이 두 칩은 어쩔 수 없이 다른 I2C 라인에 연결되어야 할 것입니다. 이런 상황을 방지하기 위해서, I2C 통신을 지원하는 대부분의 칩에는 입력 핀을 하나 두어서, 그 핀에 들어가는 입력이 0인지 1인지에 따라서 I2C의 슬레이브 주소가 바뀌도록 되어 있습니다. 보통은 7비트 슬레이브 주소 중 상위 6비트를 정해놓고, 마지막 1비트는 입력 핀에 들어온 전압에 따라서 0인지 1인지가 결정되는 방식을 사용하나, 입력 핀의 값에 따라서 주소를 아예 바꾸는 경우도 있습니다.

I2C 통신에는 크게 4가지 프로토콜이 있습니다. 마치 HerkuleX에 RAM_READ, RAM_WRITE 프로토콜이 있듯이, I2C 통신도 슬레이브의 레지스터를 읽어오거나 쓰는 방식으로 이루어집니다.

한 바이트 읽기

마스터	S	AD+W		RA		DATA		P
슬레이브			ACK		ACK		ACK	

여러 바이트 읽기

마스터	S	AD+W		RA		DATA		...	DATA		P
슬레이브			ACK		ACK		ACK			ACK	

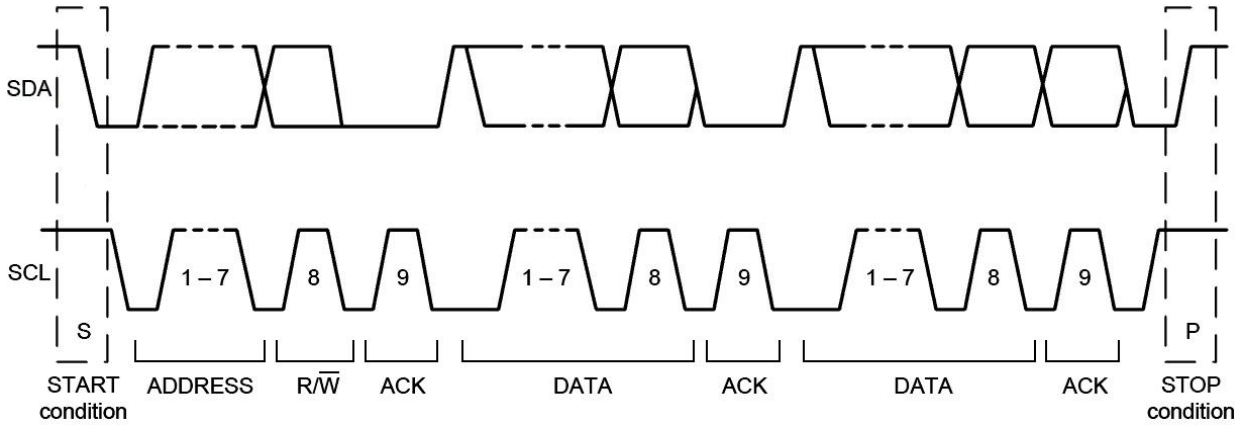
한 바이트 쓰기

마스터	S	AD+W		RA		S	AD+R			NACK	P
슬레이브			ACK		ACK			ACK	DATA		

여러 바이트 쓰기

마스터	S	AD+W		RA		S	AD+R			ACK	...		NACK	P
슬레이브			ACK		ACK			ACK	DATA			DATA		

신호	설명
S	시작 신호(SCL이 1일 동안 SDA가 1–>0)
AD	I2C 슬레이브 주소(7비트)
W	쓰기 비트(0)
R	읽기 비트(1)
ACK	응답 신호(9번째 클럭에서 SCL이 1인 동안 SDA가 0으로 유지)
NACK	비 응답 신호(9번째 클럭에서 SCL이 1인 동안 SDA가 1로 유지)
RA	I2C 슬레이브 내부의 레지스터 주소
DATA	전송되는 데이터(8비트)
P	정지 신호(SCL이 1일 동안 SDA가 0–>1)



I2C 통신은 시작 신호로 시작해서 정지 신호로 끝납니다. 평소 SCL과 SDA 핀은 1(HIGH)로 유지되고 있는데, SCL이 1인데 SDA가 1에서 0(LOW)으로 바뀌면 그 때가 통신의 시작입니다. 그 후에는 8비트를 보내면 받는 쪽이 1비트의 ACK로 응답하는 것을 반복하다가, SCL이 1인데 SDA가 0에서 1로 바뀌는 정지 신호로 끝이 납니다. 예외적으로 쓰기 동작의 경우 슬레이브 주소와 레지스터 주소를 쓴 후에 다시 시작 신호를 보내는데, 이 시작 신호를 재시작(restart) 신호라고 하기도 합니다. 시작과 정지 신호 외에 모든 SDA 핀의 값 변경은 SCL이 0일 때 이루어집니다.

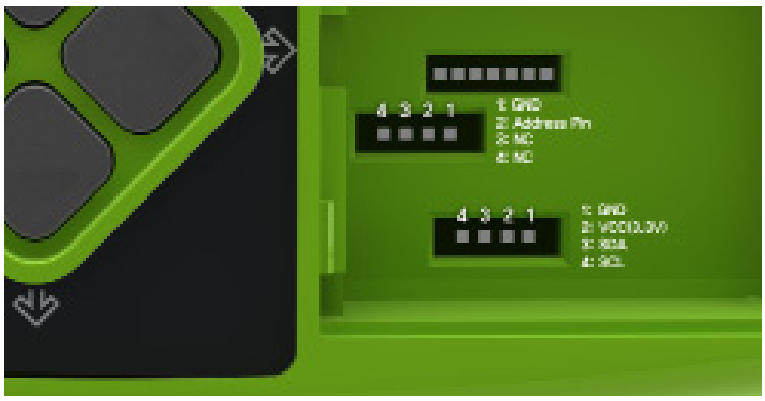
libaccgyro.a에서는 자잘한 시작 신호, 정지 신호, 응답 신호 등을 생각할 필요 없이, I2C 통신을 통해서 슬레이브에 데이터를 읽고 쓸 수 있는 함수들을 제공합니다.

```
void I2C_InitPin(unsigned char ucAddressPin);
unsigned char I2C_ucReadReg(unsigned char ucSlaveAddr, unsigned char ucRegAddr,
                            unsigned char *ucData);
unsigned char I2C_ucReadNReg(unsigned char ucSlaveAddr, unsigned char ucRegAddr,
                            unsigned char *ucData, unsigned char ucCnt);
unsigned char I2C_ucWriteReg(unsigned char ucSlaveAddr, unsigned char ucRegAddr,
                            unsigned char ucData);
unsigned char I2C_ucWriteNReg(unsigned char ucSlaveAddr, unsigned char ucRegAddr,
                            unsigned char *ucData, unsigned char ucCnt);
```

I2C_InitPin 함수는 DRC에서 I2C 통신에 사용하는 3가지 핀을 초기화 하는 함수입니다. SCL, SDA가 초기화 되고, 나머지 하나는 앞서 언급한 주소 출력 핀입니다. 주소 출력 핀을 원하는 칩의 주소 입력 핀과 연결해서 슬레이브 주소를 설정하는 데 사용할 수 있습니다. 매개변수인 ucAddressPin이 0이면 출력은 0(LOW), 1이면 출력이 1(HIGH)가 됩니다.

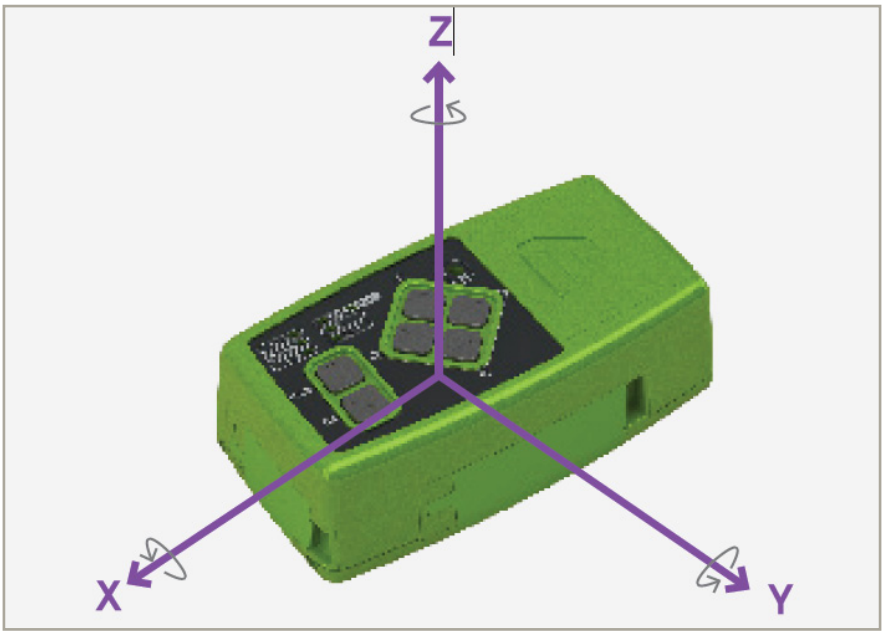
I2C_ucReadReg 함수는 I2C 통신으로 ucSlaveAddr이라는 슬레이브 주소의 ucRegAddr라는 레지스터 주소에서 데이터 1바이트를 읽어와 ucData라는 주소에 저장하는 함수입니다. I2C_ucReadNReg 함수도 마찬가지이지만 ucData라는 주소부터 ucCnt개만큼 읽어온다는 점이 다릅니다. I2C_ucWriteReg 함수는 I2C 통신으로 ucSlaveAddr이라는 슬레이브 주소의 ucRegAddr라는 레지스터 주소에 ucData라는 데이터를 쓰는 함수입니다. I2C_ucWriteNReg 함수도 마찬가지이지만 ucData라는 주소부터 ucCnt개만큼의 데이터를 쓴다는 점이 다릅니다. 위의 네 함수는 모두 unsigned char 라는 리턴 값을 가지는데, 통신이 끝까지 성공했으면 1, 실패했으면 0이 리턴 됩니다.

사용자가 가속도/자이로 센서 모듈 외의 다른 칩과 통신을 하고 싶다면, 핀 맵에 맞게 칩과 DRC를 연결하고, I2C_InitPin 함수로 핀을 초기화한 후 네 가지 통신 함수를 사용해 칩과 통신하면 어떤 칩과도 DRC와 연결하여 활용할 수 있을 것입니다. I2C 통신에 사용되는 핀의 핀 맵은 아래 그림과 같습니다.



가속도/자이로 센서 모듈

동부로봇의 가속도/자이로 센서 모듈은 위 그림에 나온 4핀 핀헤더 소켓 2개에 장착됩니다. 앞서 언급한대로, 센서 모듈에는 가속도 센서 칩과 자이로 센서 칩이 들어가 있어서 DRC와 같은 라인을 통해 I2C 통신으로 연결됩니다. 가속도/자이로 센서 칩은 각각 X, Y, Z 3축에 대한 선가속도와 각속도 데이터를 제공합니다. 이 센서의 데이터를 통해 DRC, 나아가서는 로봇이 어떤 기울기로 놓여 있는지, 어떤 속도로 회전하고 있는지를 알 수 있습니다.



가속도 센서에서는 ±16g(1g는 중력가속도, 즉 9.8m/s2)의 가속도를 13bit(−4096~4095)의 값으로 읽을 수 있습니다. 따라서 1g에 해당하는 값은 256입니다. 또한 자이로 센서에서는 ±2000dps(degree per second, °/s)의 각속도를 16bit(−32768~32767)의 값으로 읽을 수 있습니다. 예를 들어, 초당 90도를 회전하는 속도에 해당하는 값은 약 1475입니다.

DRC가 책상에 버튼 부분을 위로 바라보고 놓여 있을 때, 중력에 의해서 Z축의 가속도 값은 +1g를 나타내며, 따라서 Z축 가속도 값은 약 256이 나옵니다. 그 상태에서 DRC의 한쪽을 들어 DRC를 Θ° 정도 기울인다면, Z축 가속도 값은 256에 cosΘ°가 곱해진 값이 나옵니다. 예를 들어 45°를 기울인다면 Z축 가속도 값은 256×√2÷2=181이라는 값이 나올 것입니다.

가속도 센서에서는 ±16g(1g는 중력가속도, 즉 9.8m/s2)의 가속도를 13bit(–4096~4095)의 값으로 읽을 수 있습니다. 따라서 1g에 해당하는 값은 256입니다. 또한 자이로 센서에서는 ±2000dps(degree per second, °/s)의 각속도를 16bit (–32768~32767)의 값으로 읽을 수 있습니다. 예를 들어, 초당 90도를 회전하는 속도에 해당하는 값은 약 1475입니다.
DRC가 책상에 버튼 부분을 위로 바라보고 놓여 있을 때, 중력에 의해서 Z축의 가속도 값은 +1g를 나타내며, 따라서 Z축 가속도 값은 약 256이 나옵니다. 그 상태에서 DRC의 한쪽을 들어 DRC를 Θ° 정도 기울인다면, Z축 가속도 값은 256에 cosΘ°가 곱해진 값이 나옵니다. 예를 들어 45°를 기울인다면 Z축 가속도 값은 256×√2÷2=181이라는 값이 나올 것입니다.

다시 DRC를 책상에 놓고 회전 시키면 Z축을 회전 축으로 하는 회전 각속도가 생깁니다. 자이로 센서는 회전 각속도를 감지해서 데이터로 변환 시킵니다. 반시계 방향으로 회전 시키면 자이로 센서에서는 양(+)의 Z축 각속도 값이 나오며, 시계 방향으로 회전 시키면 반대로 음(–)의 Z축 각속도 값이 나올 것입니다.

libaccgyro.a에서는 앞서 설명한 일반적인 I2C 통신에 대한 함수 뿐 아니라, 초기화 함수만 한번 실행하면 손쉽게 가속도와 자이로 센서 값을 읽을 수 있는 함수들도 제공합니다.

```
unsigned char AccGyro_ucInit(void);
unsigned char AccGyro_ucReadAccX(int *iData);
unsigned char AccGyro_ucReadAccY(int *iData);
unsigned char AccGyro_ucReadAccZ(int *iData);
unsigned char AccGyro_ucReadAccXYZ(int *iData);
unsigned char AccGyro_ucReadGyroX(int *iData);
unsigned char AccGyro_ucReadGyroY(int *iData);
unsigned char AccGyro_ucReadGyroZ(int *iData);
unsigned char AccGyro_ucReadGyroXYZ(int *iData);
```

AccGyro_ucInit 함수는 DRC에서 I2C 통신에 사용하는 3가지 핀을 초기화 하고, 가속도/자이로 센서 모듈도 동작하도록 초기화 함수입니다. 가속도 센서와 자이로 센서가 제대로 통신 되는지 확인하고, 최대 감지 범위를 설정하고, 변환을 시작합니다. 통신과 설정이 끝까지 성공했으면 1, 실패했으면 0이 리턴 됩니다.

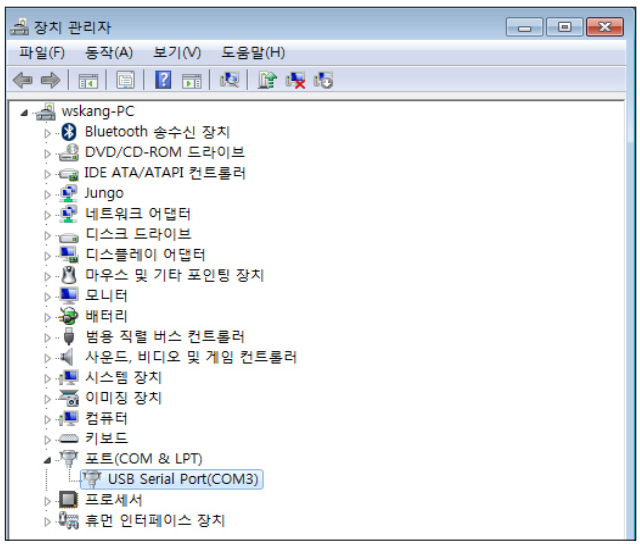
나머지 함수들은 이름대로 가속도 센서와 자이로 센서의 특정 축 값을 읽거나 XYZ축 값을 한꺼번에 읽는 함수입니다. iData로 넘긴 포인터 변수의 주소에 값이 저장되고, 읽는 데 성공한 경우 1, 실패한 경우 0이 리턴 됩니다. 예제에서는 위 함수를 사용해서 가속도/자이로 센서 모듈의 값을 읽습니다.

준비

하드웨어 준비하기

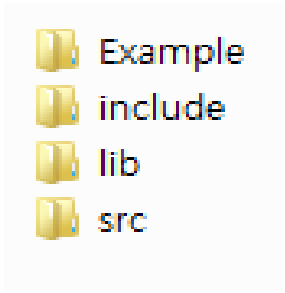
01 연결

DRC 제어기 아답터로 전원을 연결하고, USB to Serial Cable 로 PC 와 DRC 를 연결합니다.
가속도/자이로 모듈을 장착합니다.



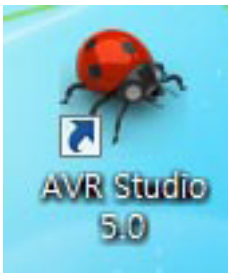
02 COM port 연결

내컴퓨터 〉 속성 〉 장치관리자에서 COM 연결 확인합니다.



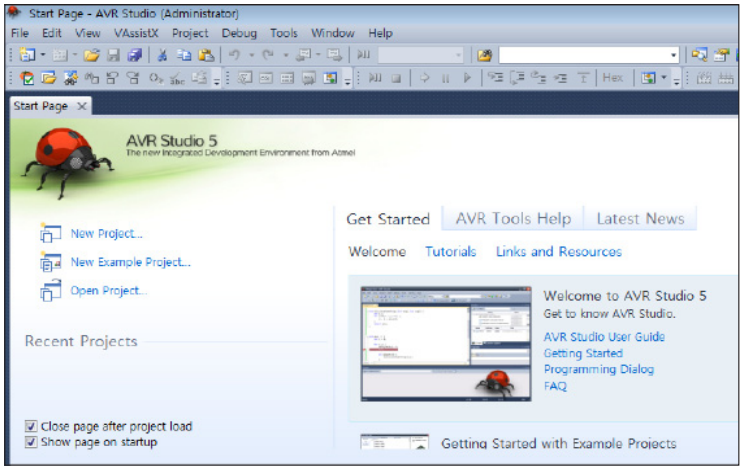
00 라이브러리

가속도/자이로 모듈을 제어하기 위해서는 기본 라이브러리 파일이 필요합니다. 라이브러리가 포함된 폴더 구조입니다.



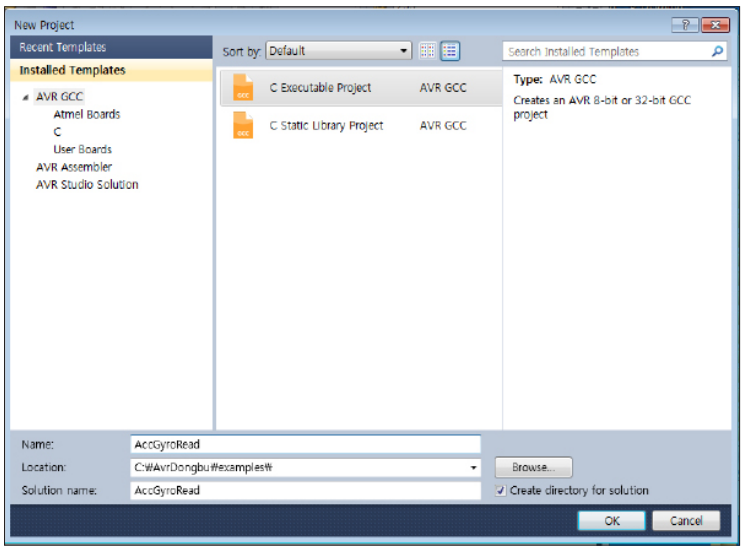
01 실행

AVR Studio 아이콘을 클릭하여 실행합니다.



02 New Project

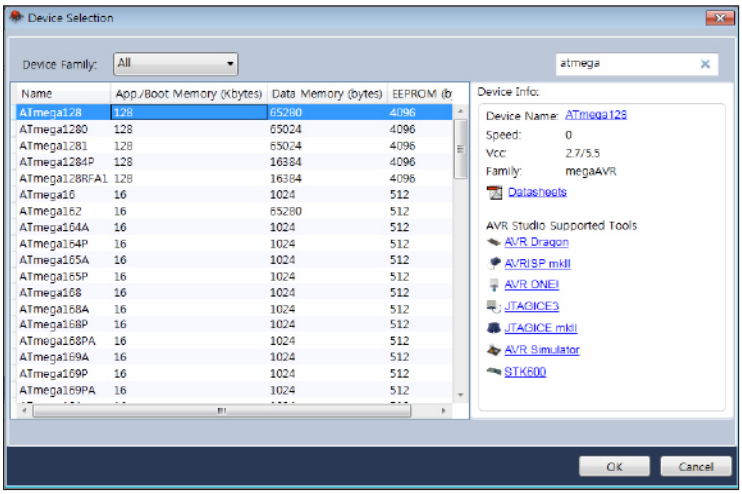
New Project 를 클릭합니다.



03 GCC 선택

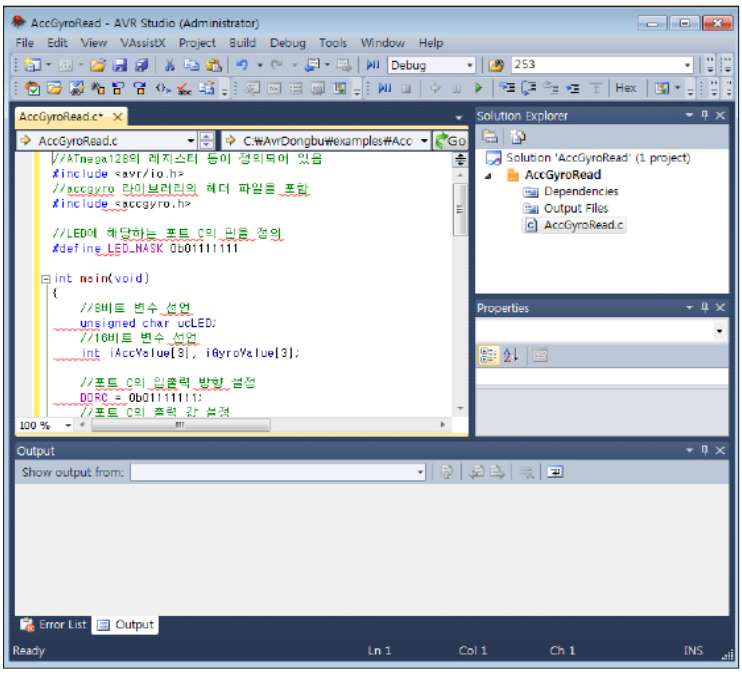
C Executable Project 를 선택합니다.

Name 을 “AccGyroRead”로 입력하고, Create directory for solution 을 클릭하고, OK 버튼을 누릅니다.



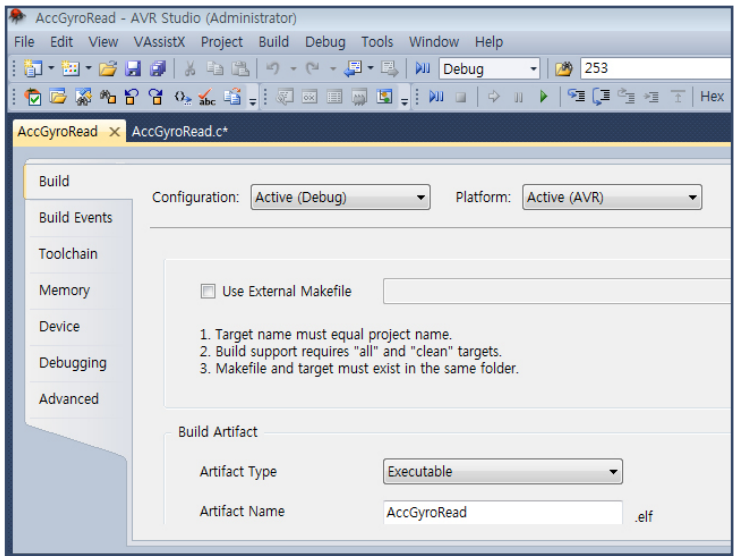
04 ATmega 선택

우측 상단에 atmega 를 검색하여,
리스트중에 ATmega128 을 선택합니다.



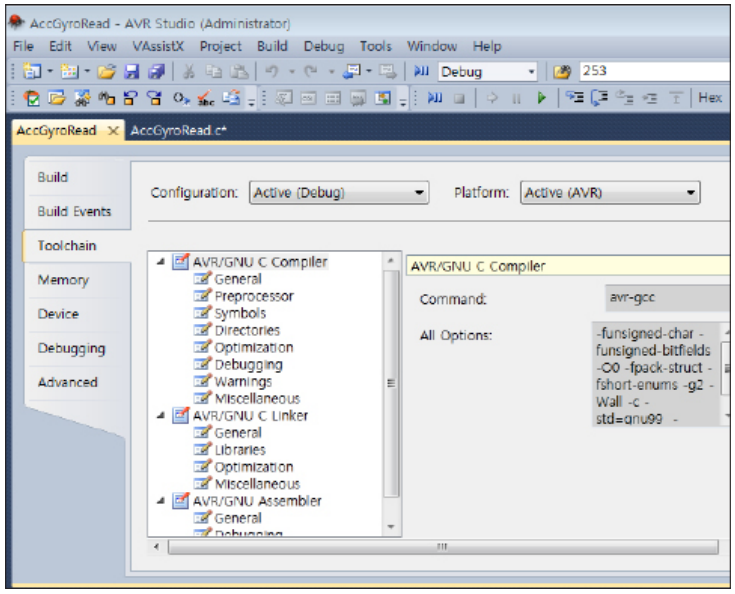
05 Editor

편집창에 코딩을 시작합니다.



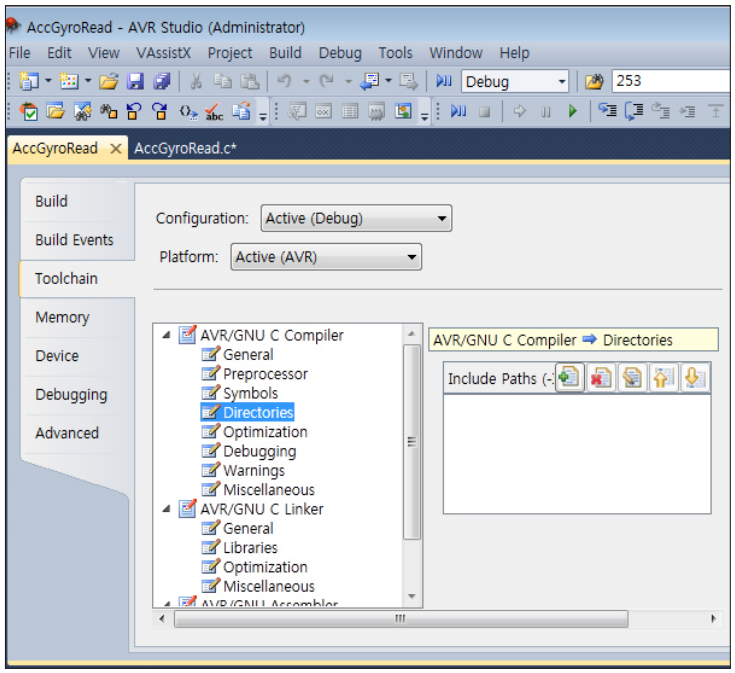
01 설정

AVR Studio 상단 메뉴중에 Project 를 클릭하면 AccGyroRead (프로젝트 이름별로 다름) Property 메뉴가 생성되어 있습니다. 클릭하면 좌측과 같이 설정화면이 나옵니다.



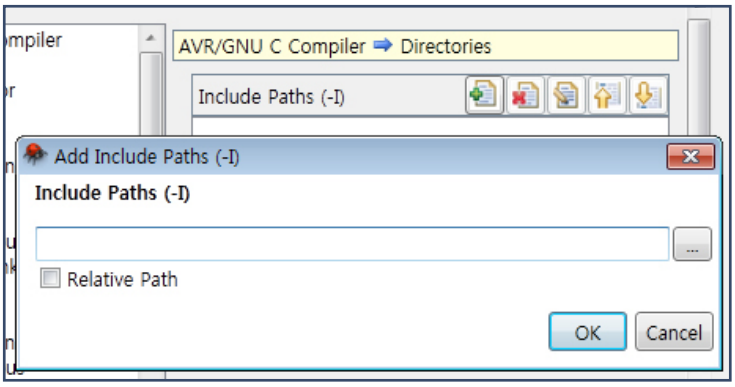
02 Toolchain

좌측 탭버튼중에 Toolchain 을 클릭하여, AVR/GNU C Compiler 을 선택합니다



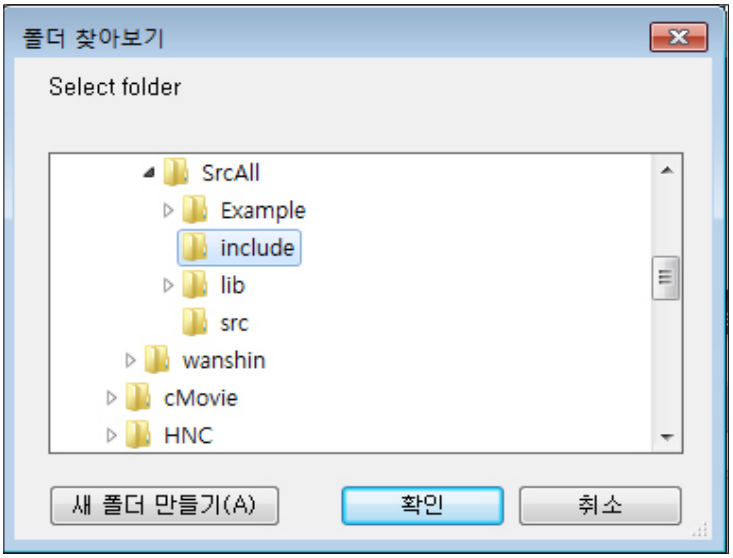
03 C Compiler

C Compiler 폴더중에 Directories 를 선택합니다.



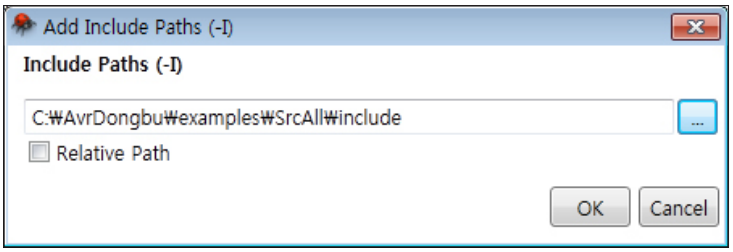
04 Path

Include Paths 오른쪽에 + 버튼을 클릭하면 팝업 입력창이 뜹니다.
Relative Path (상대경로) 체크를 해지하고(5.1에서는 안해도됨),
우측 폴더 검색 버튼을 클릭합니다.



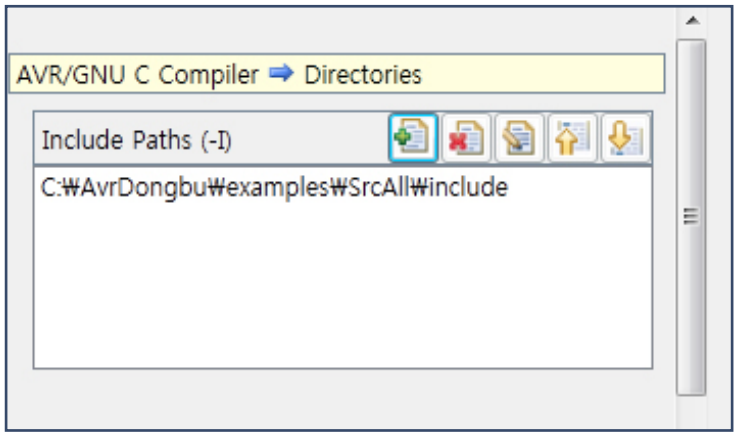
05 Include

제공 라이브러리 폴더중에 include 를 선택합니다.



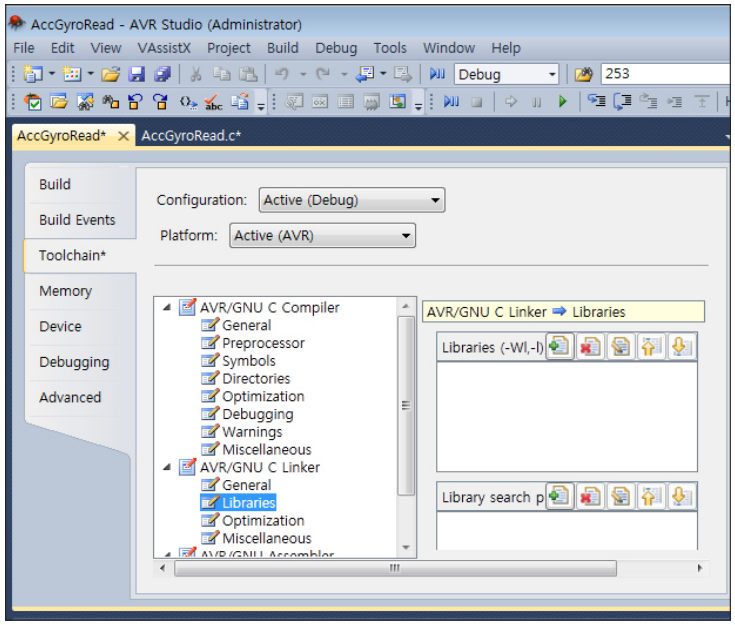
06 Include 완료

Include 폴더 위치가 지정됩니다. OK 버튼을 클릭합니다.



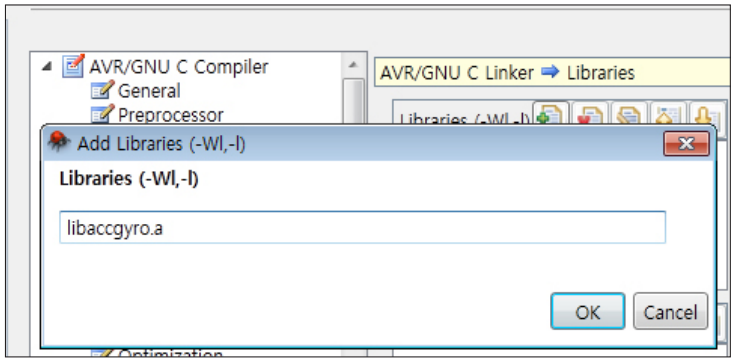
07 Path 완료

Include 경로지정이 완료되었습니다.



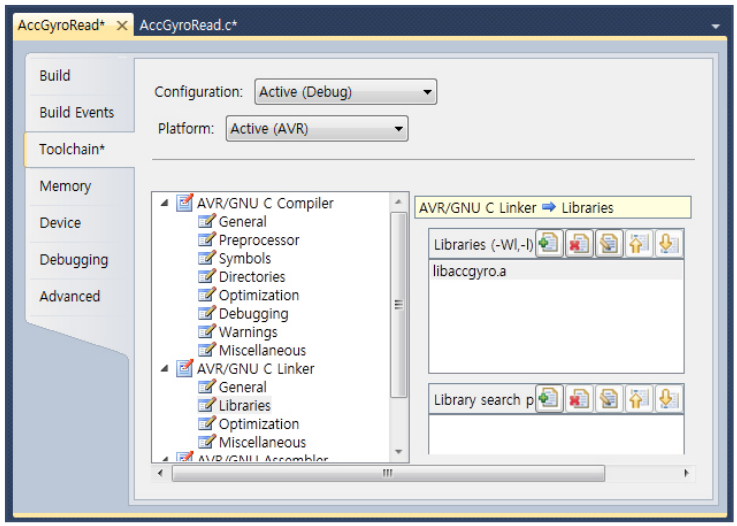
08 C Linker

C Linker 폴더중에 Libraries 를 선택합니다.



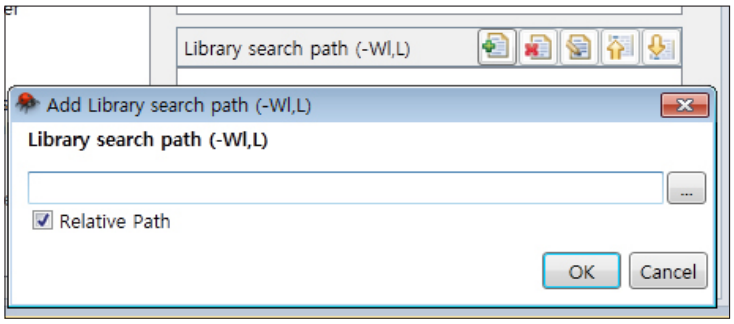
09 Libraries

Libraries 창 우측 + 를 클릭하면, Libraries 파일명 입력창이 나옵니다. libaccgyro.a 를 직접 입력합니다.
OK 버튼을 클릭합니다.



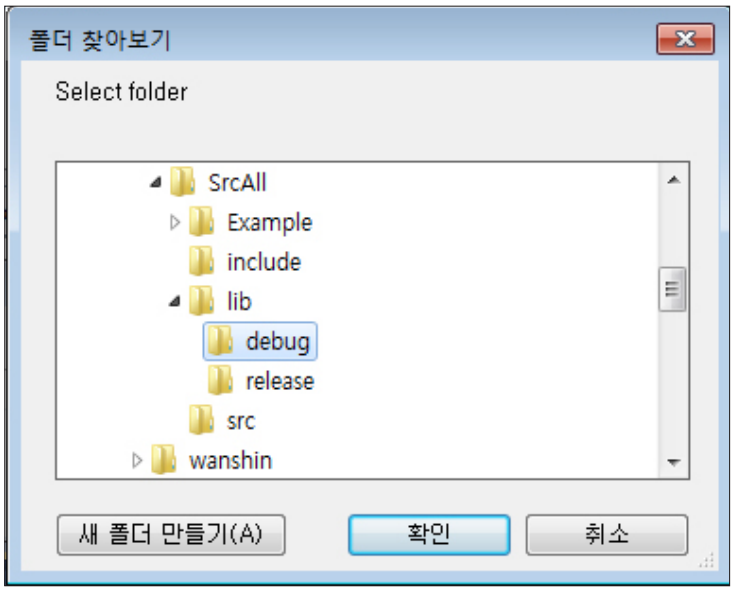
10 Search

가속도/자이로모듈 라이브러리 파일 libaccgyro.a 링크가 완료되었습니다.
다음으로 Library search path 를 설정합니다.
우측 + 버튼을 클릭합니다.



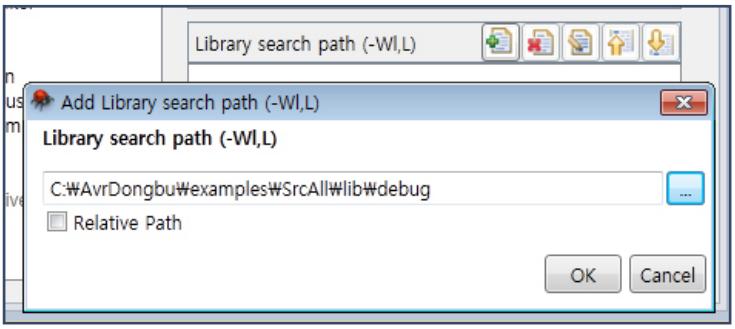
11 Path

Library search path 팝업창 우측에 폴더 검색 버튼을 클릭합니다.



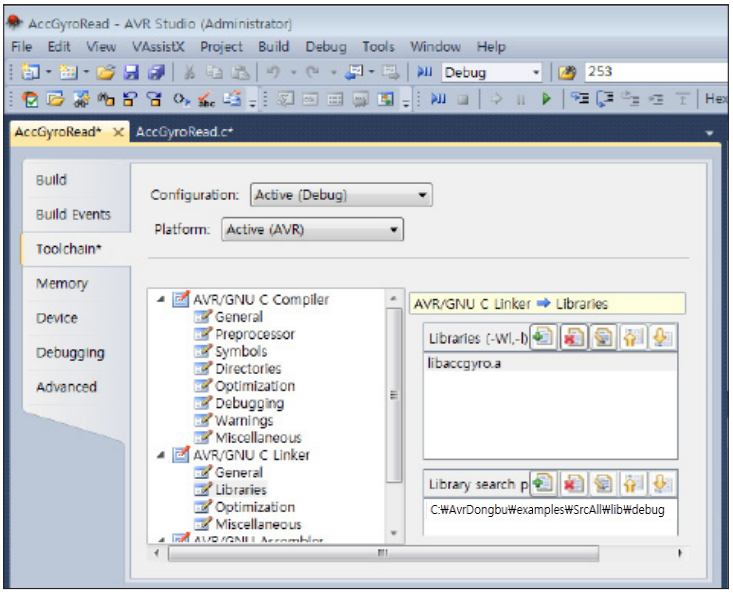
12 Debug

제공 라이브러리 폴더에서 lib 하위 폴더에 debug 를 선택합니다.



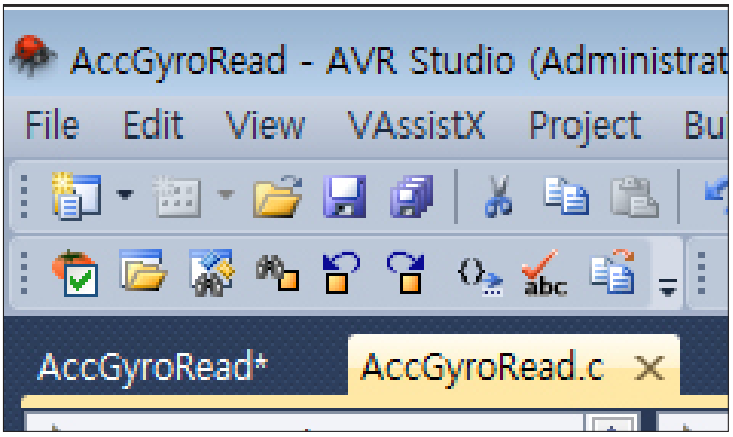
13 Path 완료

Library search path 경로지정이 완료되었습니다.
OK 버튼을 클릭합니다.



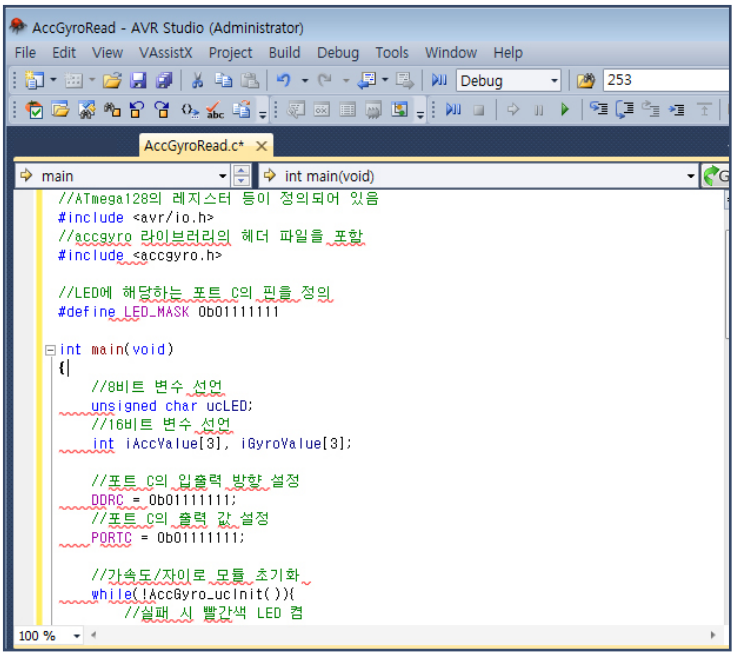
14 Link 완료

C Linker 라이브러리파일과 경로지정이 완료되었습니다



15 저장

저장 버튼을 누르면 현재 설정값이 저장됩니다. 탭바 이름이 AccGyroRead * –〉 AccGyroRead 로 변경됩니다.



16 코딩

AccGyroRead.c 탭바를 선택하고, 아래 제공된 예제를 입력합니다.

예제

DRC에 장착된 가속도/자이로 센서를 초기화 하고 가속도 센서와 자이로 센서 값을 지속적으로 읽어오면서, DRC가 45도 이상 기울어지면 초록색 LED를 켜고, DRC가 90dps 이상으로 회전하면 파란색 LED를 켜는 예제입니다.

accgyro.h

```
#ifndef ACCGYRO_H_
#define ACCGYRO_H_

#ifdef __ACCGYRO_C
        #define ACCGYRO_EXT
#else
        #define ACCGYRO_EXT extern
#endif

//I2C 통신을 위해 SCL, SDA 핀을 초기화 하고 주소 출력 값을 ucAddressPin으로 초기화 하는 함수.
ACCGYRO_EXT void I2C_InitPin(unsigned char ucAddressPin);
//I2C 통신으로 ucSlaveAddr의 ucRegAddr에서 값을 ucData로 읽어오는 함수. 성공 시 1, 실패 시 0 리턴.
ACCGYRO_EXT unsigned char I2C_ucReadReg(unsigned char ucSlaveAddr, unsigned char ucRegAddr, unsigned char *ucData);
//I2C 통신으로 ucSlaveAddr의 ucRegAddr에서 값을 ucData부터 ucCnt개 읽어오는 함수. 성공 시 1, 실패시 0 리턴.
```

```
ACCGYRO_EXT unsigned char I2C_ucReadNReg(unsigned char ucSlaveAddr, unsigned char ucRegAddr, unsigned char *ucData, unsigned char ucCnt);
//I2C 통신으로 ucSlaveAddr의 ucRegAddr에 ucData라는 값을 쓰는 함수. 성공 시 1, 실패시 0 리턴.
ACCGYRO_EXT unsigned char I2C_ucWriteReg(unsigned char ucSlaveAddr, unsigned char ucRegAddr, unsigned char ucData);
//I2C 통신으로 ucSlaveAddr의 ucRegAddr에 ucData부터 ucCnt개의 값을 쓰는 함수. 성공 시 1, 실패시 0 리턴.
ACCGYRO_EXT unsigned char I2C_ucWriteNReg(unsigned char ucSlaveAddr, unsigned char ucRegAddr, unsigned char *ucData, unsigned char ucCnt);

//가속도/자이로 센서 모듈을 초기화 하는 함수. 성공 시 1, 실패시 0 리턴.
ACCGYRO_EXT unsigned char AccGyro_ucInit(void);
//가속도 센서의 X축 값을 iData로 읽어오는 함수. 성공 시 1, 실패시 0 리턴.
ACCGYRO_EXT unsigned char AccGyro_ucReadAccX(int *iData);
//가속도 센서의 Y축 값을 iData로 읽어오는 함수. 성공 시 1, 실패시 0 리턴.
ACCGYRO_EXT unsigned char AccGyro_ucReadAccY(int *iData);
//가속도 센서의 Z축 값을 iData로 읽어오는 함수. 성공 시 1, 실패시 0 리턴.
ACCGYRO_EXT unsigned char AccGyro_ucReadAccZ(int *iData);
//가속도 센서의 X, Y, Z축 값을 iData 배열로 읽어오는 함수. 성공 시 1, 실패시 0 리턴.
ACCGYRO_EXT unsigned char AccGyro_ucReadAccXYZ(int *iData);
//자이로 센서의 X축 값을 iData로 읽어오는 함수. 성공 시 1, 실패시 0 리턴.
ACCGYRO_EXT unsigned char AccGyro_ucReadGyroX(int *iData);
//자이로 센서의 Y축 값을 iData로 읽어오는 함수. 성공 시 1, 실패시 0 리턴.
ACCGYRO_EXT unsigned char AccGyro_ucReadGyroY(int *iData);
//자이로 센서의 Z축 값을 iData로 읽어오는 함수. 성공 시 1, 실패시 0 리턴.
ACCGYRO_EXT unsigned char AccGyro_ucReadGyroZ(int *iData);
//자이로 센서의 X, Y, Z축 값을 iData 배열로 읽어오는 함수. 성공 시 1, 실패시 0 리턴.
ACCGYRO_EXT unsigned char AccGyro_ucReadGyroXYZ(int *iData);

#endif /* ACCGYRO_H_ */
```

AccGyro Read.c

```
//ATmega128의 레지스터 등이 정의되어 있음
#include <avr/io.h>
//accgyro 라이브러리의 헤더 파일을 포함
#include <accgyro.h>

//LED에 해당하는 포트 C의 핀을 정의
#define LED_MASK 0b01111111
```

```
int main(void)
{
	//8비트 변수 선언
	unsigned char ucLED;
	//16비트 변수 선언
	int iAccValue[3], iGyroValue[3];

	//포트 C의 입출력 방향 설정
	DDRC = 0b01111111;
	//포트 C의 출력 값 설정
	PORTC = 0b01111111;

	//가속도/자이로 모듈 초기화
	while(!AccGyro_ucInit()){
		//실패 시 빨간색 LED 켬
		ucLED = 0b01001001;
		PORTC = (~ucLED) & LED_MASK;
	}

	//while 안의 내용을 무한 반복
	while(1)
	{
		//가속도 센서, 자이로 센서 값 읽어서 배열에 저장
		if(AccGyro_ucReadAccXYZ(iAccValue) && AccGyro_ucReadGyroXYZ(iGyroValue)){
			ucLED = 0b00000000;

			//Z축이 45도 이상 기울어져 있으면 초록색 LED 켬
			if(iAccValue[2] < 180){
				ucLED |= 0b00010010;
			}

			//Z축 기준으로 ±90dps 이상 회전하고 있으면 파란색 LED 켬
			if(iGyroValue[2] > 1500 || iGyroValue[2] < -1500){
				ucLED |= 0b00100100;
			}
		}
		//센서 값 읽기 실패 시 빨간색 LED 켬
		else{
			ucLED = 0b01001001;
		}

		//ucLED 값에 해당하는 LED가 켜지도록 PORTC에 값 대입
		PORTC = (~ucLED) & LED_MASK;
	}

	//함수의 형태에 맞게 정수값 반환
	return 1;
}
```

예제설명

accgyro.h의 내용에 대한 설명은 위의 H/W 원리에서 I2C 통신, 가속도/자이로 모듈에 대해 설명하면서 함께 설명되었으므로 함수를 사용하는 데 무리가 없을 것이라 생각되어 생략합니다.

```
#include <avr/io.h>
#include <accgyro.h>
```

avr/io.h 헤더 파일에는 DDRX, PORTX 레지스터 등 AVR 제어를 위해 필요한 레지스터 등이 정의되어 있습니다. 또 accgyro.h 헤더 파일에는 libaccgyro.a 라이브러리 파일에 들어 있는 함수의 원형이 정의되어 있습니다. 현재 소스 코드에서 사용할 수 있도록 #include 문을 사용하여 포함시켜 줍니다.

```
#define LED_MASK 0b01111111
```

ucLED 변수를 사용해서 PORTC를 통해 LED를 제어할 때 실제 LED가 연결된 7개 핀만 값을 바꾸기 위해서 LED_MASK를 선언합니다.

```
unsigned char ucLED;
int iAccValue[3], iGyroValue[3];
```

변수를 선언합니다. ucLED는 현재 LED 상태를 저장하는 변수이며, 7개의 비트가 7개 LED와 대응됩니다. 0일 때 LED가 꺼진 것을 나타내고 1일 때는 켜진 것을 나타냅니다. iAccValue[3]과 iGyroValue[3]은 부호 있는 2바이트 정수 배열로, 각각 가속도와 자이로 센서의 X, Y, Z축 데이터가 순서대로 저장됩니다.

```
DDRC = 0b01111111;
PORTC = 0b01111111;
```

포트 C를 설정합니다. DDRC의 LED에 해당하는 7개 비트에 1을 넣어 출력으로 만들고, PORTC에도 7개 비트에 1을 넣어 LED를 모두 끕니다.

```
while(!AccGyro_ucInit()){
        ucLED = 0b01001001;
        PORTC = (~ucLED) & LED_MASK;
}
```

AccGyro_ucInit 함수를 사용해서 I2C 통신을 위한 핀과 가속도/자이로 모듈을 초기화 합니다. 성공하면 AccGyro_ucInit 함수의 리턴 값은 1일 것이므로 while문으로 진입하지 않고 빠져나갈 것이고, 실패하면 리턴 값이 0일 것이므로 while문 내로 진입해서 빨간색 LED(Servo, TX, Power LED)를 모두 켭니다. LED를 켠 후에는 AccGyro_ucInit를 다시 실행해서 그 결과 값에 따라서 while문을 다시 반복할지 빠져나올지가 결정됩니다. 이렇게 코드를 작성하면 AccGyro_ucInit가 계속 실패할 경우 이 while문을 반복하게 되며, AccGyro_ucInit가 성공해야만 다음으로 넘어가게 됩니다.

```
if(AccGyro_ucReadAccXYZ(iAccValue) && AccGyro_ucReadGyroXYZ(iGyroValue)){
        ucLED = 0b00000000;

        if(iAccValue[2] < 180){
                ucLED |= 0b00010010;
        }

        if(iGyroValue[2] > 1500 || iGyroValue[2] < -1500){
                ucLED |= 0b00100100;
        }
}
else{
        ucLED = 0b01001001;
}
```

while문 안의 if문 내용입니다. 이 if문의 조건문에서는, AccGyro_ucReadAccXYZ 함수와 AccGyro_ucReadGyroXYZ 함수를 실행해서 둘 다 리턴 값이 1로 성공한 경우에만 if문 안의 내용을 실행하고 하나라도 실패한 경우는 else의 내용을 실행합니다. 초기화 때와 마찬가지로 실패한 경우에는 ucLED 값을 빨간색 LED가 모두 켜지도록 변경합니다.

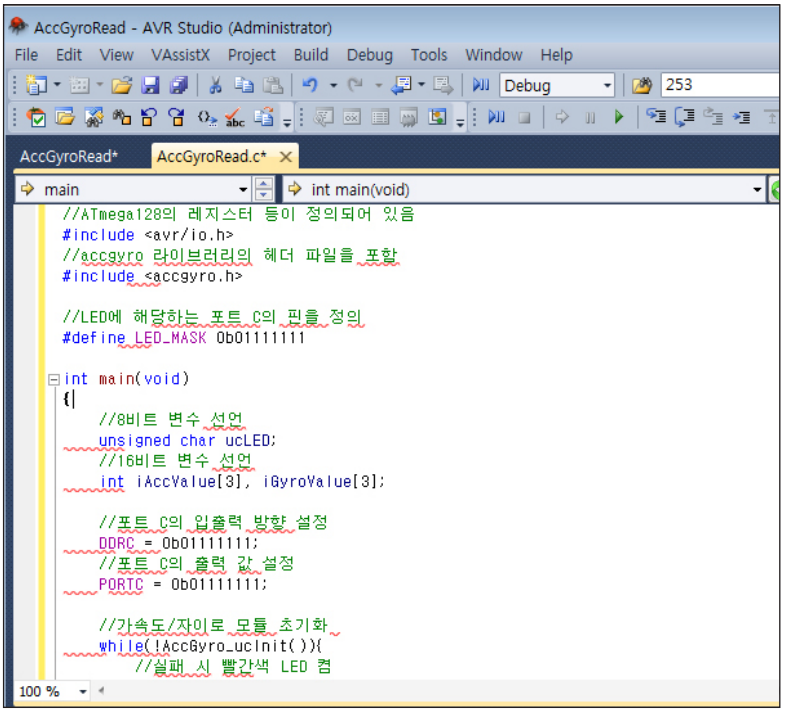
If문 안에서는 우선 ucLED 값이 모두 꺼지도록 초기화합니다. 그리고 iAccValue[2], 즉 Z축의 가속도 값이 180이하면 ucLED의 초록색 LED(Program, RX)가 켜지도록 ucLED의 해당 비트를 1로 만듭니다. Z축의 가속도 값이 180이하라는 것은, H/W 원리에서 언급했듯이 DRC가 약 45° 이상 기울어져 있다는 뜻입니다. 또, iGyroValue[2]가 1500보다 크거나 –1500보다 작으면 ucLED의 파란색 LED(EXEC, Spare)가 켜지도록 ucLED의 해당 비트를 1로 만듭니다. iGyroValue[2]의 절대값이 1500보다 크다는 것은, H/W 원리에서 언급했듯이 DRC가 Z축을 중심으로 약 90dps 이상의 속도로 회전하고 있다는 뜻입니다.

```
PORTC = (~ucLED) & LED_MASK;
```

위의 if–else 문에서 정한 ucLED 값을 사용해 실제로 PORTC의 값을 바꾸어 LED를 제어합니다.

실행

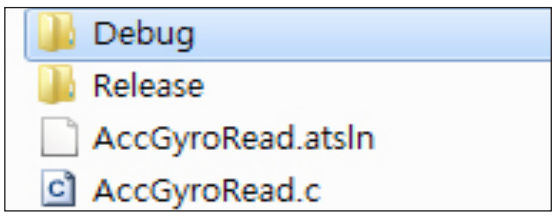
가속도/자이로 센서를 DRC에 장착한 후 펌웨어를 실행하면 가만히 수평으로 놓인 상태에서는 LED가 모두 꺼져 있으며, 기울이면 초록색 LED가, 회전 시키면 파란색 LED가 켜지는 것을 확인할 수 있습니다. 만약 가속도/자이로 센서를 장착하지 않거나 작동 중에 제거하면 빨간색 LED가 켜집니다.



01 Build

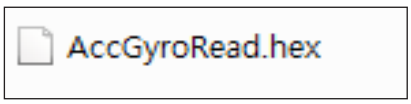
Build 메뉴를 선택하여 Build Solution을 클릭하여 Build 시킵니다.

1 succeeded 0 failed 가 나오면 에러없이 컴파일 완료됐다는 것을 의미합니다.



02 Debug 폴더

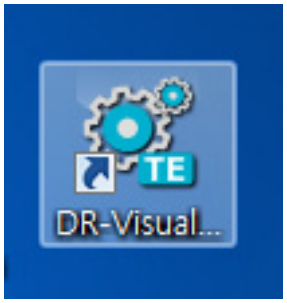
Build 후에 소스가 있는 폴더에 들어가면 Debug 폴더가 생성 되어있습니다. 클릭하여 폴더 안의 내용을 봅니다.



03 hex 확인

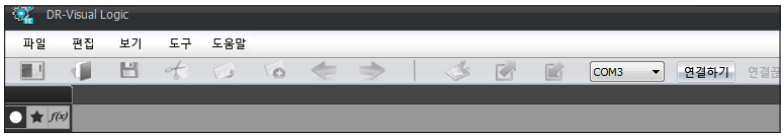
Debug 폴더 안의 hex 파일 확인합니다. Hex 파일이 제어기에 포팅시키는 파일입니다.

DR-Visual Logic 에서 펌웨어 복구를 실시합니다.



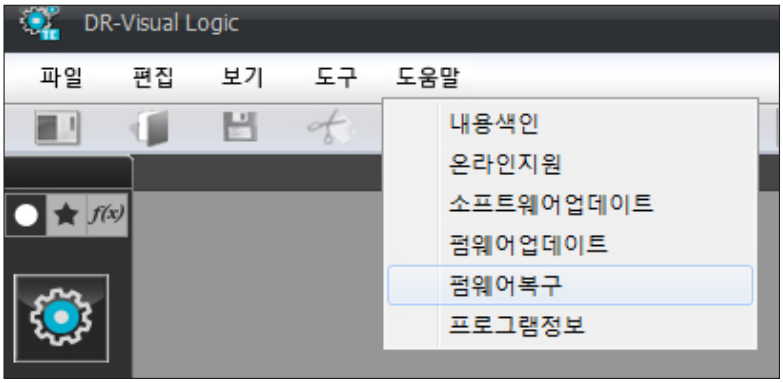
04 Visual Logic

DR-Visual Logic 을 실행시킵니다.



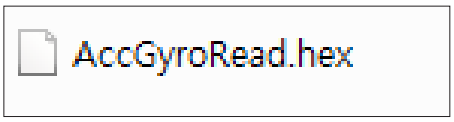
05 연결하기

COM 번호를 설정하고 연결하기를 클릭합니다.



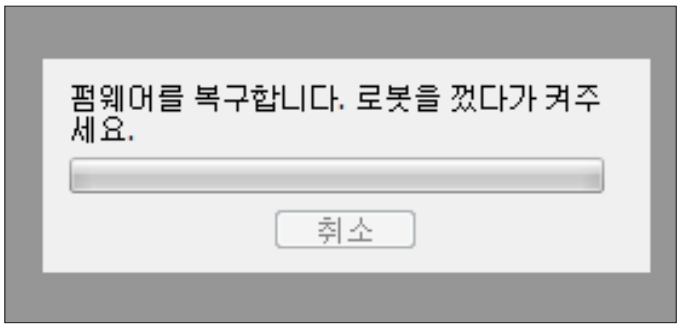
06 펌웨어 복구

도움말 메뉴에서 펌웨어복구를 클릭합니다.



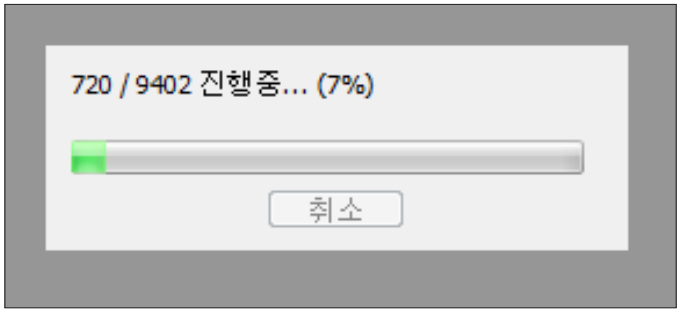
07 불러오기

Debug 폴더에서 Hex 파일을 불러옵니다



08 펌웨어복구

로봇을 껐다가 켜주세요 라는 문구가 나옵니다.
제어기의 전원버튼을 한번 껐다가 다시 켜주세요.



09 복구시작

펌웨어 복구가 시작됩니다.

10 실행

수평으로 놓인 상태에서는 LED가 모두 꺼져 있으며, 기울이면 초록색 LED가, 회전 시키면 파란색 LED가 켜지는 것을 확인할 수 있습니다.

PART 03

Firmware 익히기

HOVIS

적외선 리모컨 신호 읽기

DRC에 부착된 적외선 리모컨 수신 모듈로 리모컨 입력을 받아서 DRC를 리모컨으로 제어해봅니다

AVR 원리

이번 Chapter에서는 ATmega128의 외부 인터럽트 핀을 사용해 적외선 리모컨의 신호를 받아, 누른 리모컨의 키에 따라서 DRC가 다른 동작을 하는 예제를 작성해봅니다. 외부 인터럽트 기능과 관련된 레지스터에 대한 내용은 Chapter 4에서 자세히 다루었습니다. Chapter 4에서는 외부 인터럽트 5, 6번 핀(EXT5, EXT6)을 사용해 소리 입력을 받았었는데, 이번 Chapter에서는 외부 인터럽트 4번 핀을 사용해 적외선 리모컨 수신 모듈에서 리모컨 신호를 받습니다.

H/W 원리

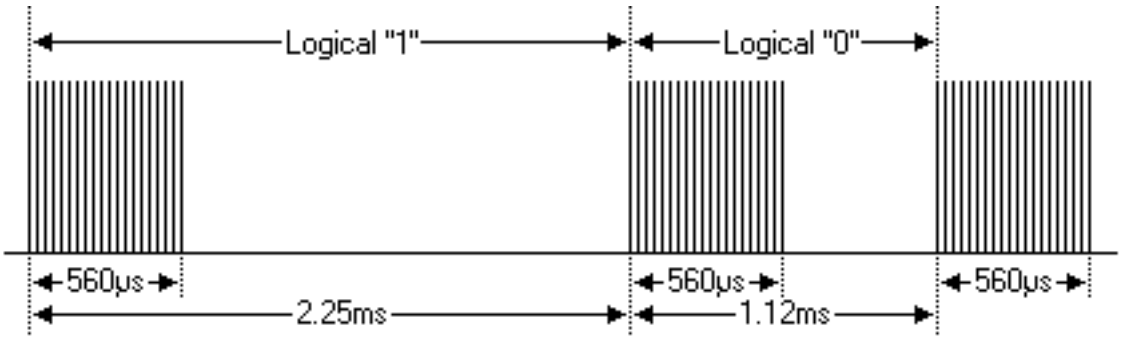
적외선 리모컨의 동작 원리

적외선 리모컨은 말 그대로 적외선을 사용해서 신호를 보내서 원격으로 다른 기기를 조종하는 장치입니다. 리모컨의 키를 누르면 리모컨 내부의 마이크로프로세서는 정해진 프로토콜에 맞게 HIGH와 LOW로 적외선 LED를 껐다가 켜서 데이터 코드를 앞의 기기에 쏩니다. 이 때 빛을 쏠 때는 단순히 HIGH이면 계속 적외선 LED를 켜고 LOW면 끄는 단순한 방식이 아니라, 38kHz의 반송파(Carrier Wave)로 변조되어 출력됩니다. 신호 내용이 HIGH일 때 38kHz의 구형파로 1과 0을 왔다갔다하는 신호를 쏜다는 뜻입니다. 리모컨 송신 시에 반송파를 쓰는 이유는 수신부에서 다른 광선과의 간섭을 방지하고 멀리 전송하기 위해서 입니다. 이렇게 반송파를 쓰면 수신부에서 리모컨 신호를 받을 때 주파수 필터링을 통해 38kHz 주파수로 진동하는 신호를 골라내서 더 정확하게 수신할 수 있습니다. 적외선 수신 모듈에서는 신호를 받아서 ATmega128의 인터럽트 핀에 데이터 코드를 보내줍니다. 이 때는 반송파가 제거된 상태로 HIGH과 LOW가 출력됩니다.

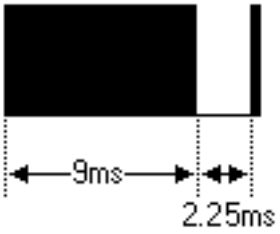
리모컨 프로토콜에는 D6121, TC9012, Sony, Panasonic, RC-5 등 국가와 회사마다 다양한 프로토콜이 존재합니다. 이 중 Hovis의 리모컨에서 사용하는 포맷은 D6121입니다. D6121은 NEC 프로토콜이라는 이름으로 더 많이 알려져 있습니다.

NEC 프로토콜(D6121)

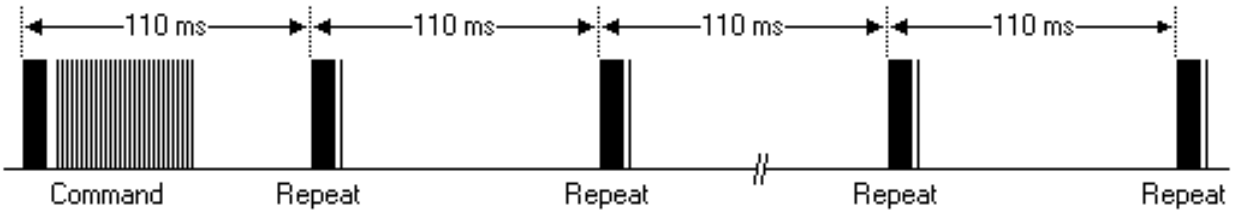
위 그림은 NEC 프로토콜의 전형적인 신호를 보여줍니다. 리모컨의 신호가 시작될 때는 9밀리초 동안의 HIGH 신호와 4.5밀리초 동안의 LOW 신호를 내보내는데, 이를 리더 코드(Leader Code)라고 합니다. 그 후에는 낮은 번호의 비트(LSB)부터 8비트의 Address와 Command를 보내는데, 각각 두 번씩 전송됩니다. 처음 보낼 때는 원래의 값을 보내고, 두 번째 보낼 때에는 모든 비트를 뒤집어서 보냅니다. 위 그림에서 보아도 처음 Address는 0b01011001인데 그 다음 ~Address는 0b10100110임을 알 수 있습니다.



Address와 Command를 보낼 때, 1과 0은 HIGH 후에 다음 HIGH가 나올 때까지의 시간으로 구분합니다. 1을 나타낼 경우에는 560마이크로초동안 HIGH가 유지되고 다음 신호가 나올 때까지 총 2.25밀리초가 소요됩니다. 0을 나타내는 경우에는 HIGH가 유지되는 시간은 같지만 다음 신호까지의 시간은 1.12밀리초밖에 되지 않습니다.



Address와 Command는 버튼을 처음 눌렀을 때 한번밖에 전송되지 않으며, 그 후에 계속 누르고 있을 때에는 110밀리초마다 리피트 코드(Repeat Code)가 전송됩니다. 리피트 코드는 위 그림과 같이 9밀리초의 HIGH 신호와 2.25밀리초동안의 LOW 신호, 그리고 560마이크로초 동안의 짧은 HIGH 신호로 이루어져 있습니다. 리피트 코드는 아래 그림과 같이 버튼이 계속 눌려 있는 동안 110밀리초마다 반복되어 현재 버튼이 계속 눌려 있음을 나타냅니다.

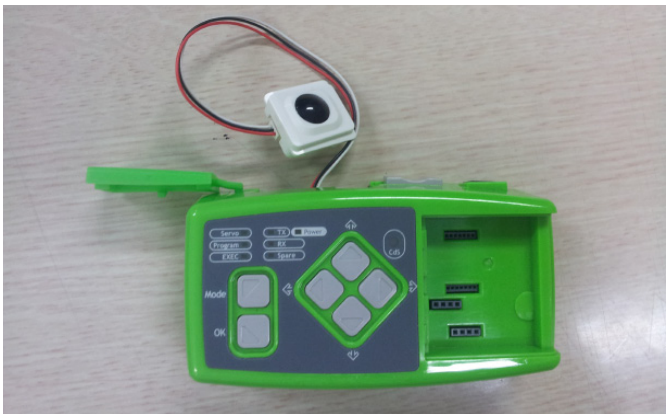


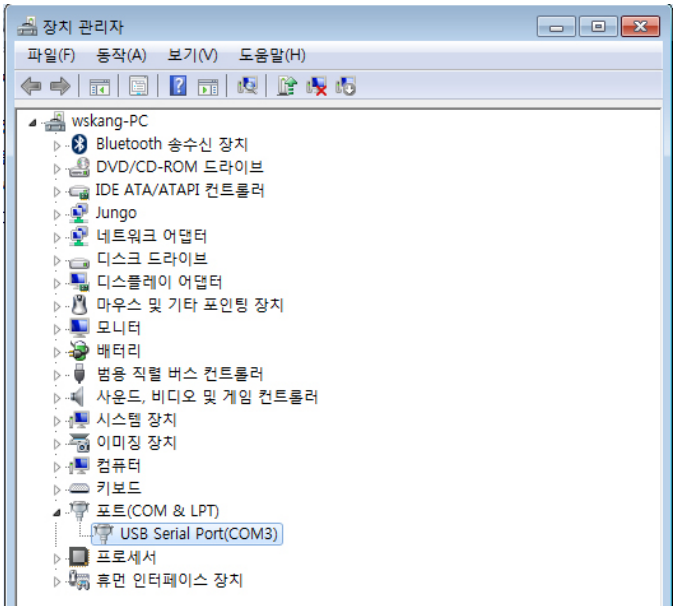
Hovis 리모컨의 경우, Address는 0x61부터 0x6A까지의 값을 가집니다. DRC 기본 펌웨어에서는 이 값을 채널이라는 값으로 나타냅니다. 리모컨의 채널은 1~0의 숫자 + OK를 동시에 누름으로써 설정할 수 있으며, 각 숫자에 대응되는 Address 값은 아래와 같습니다.

리모컨 버튼	RmcChannel 값
0+OK	97(0x61)
1+OK	98(0x62)
2+OK	99(0x63)
3+OK	100(0x64)
4+OK	101(0x65)
5+OK	102(0x66)
6+OK	103(0x67)
7+OK	104(0x68)
8+OK	105(0x69)
9+OK	106(0x6A)

또한 리모컨의 각 키마다 해당되는 버튼의 키 값은 아래 그림과 같습니다.

준비





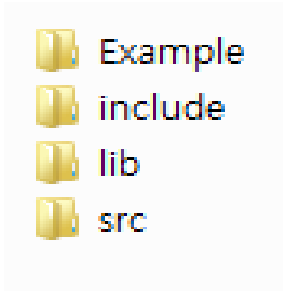
하드웨어 준비하기

01 연결

DRC 제어기 아답터로 전원을 연결하고, USB to Serial Cable 로 PC 와 DRC 를 연결합니다.
적외선 수신기를 연결합니다.리모컨도 같이 준비합니다.

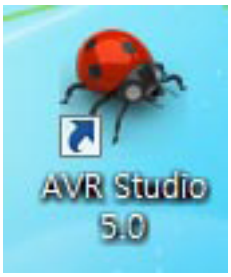
02 COM port 연결

내컴퓨터〉속성〉장치관리자에서 COM 연결 확인합니다.



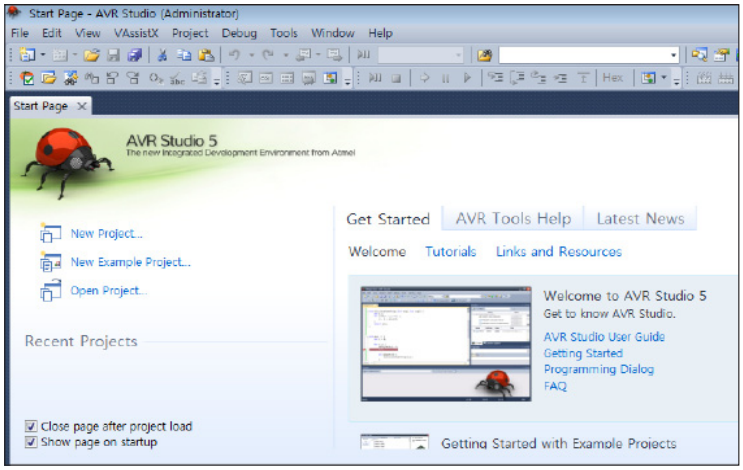
00 라이브러리

리모컨 입력을 받기 위해서는 기본 라이브러리 파일이 필요합니다.
라이브러리가 포함된 폴더 구조입니다.



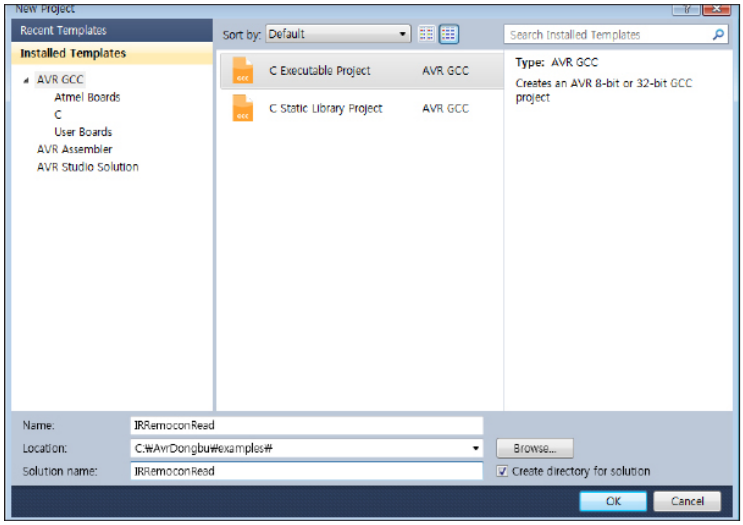
01 실행

AVR Studio 를 실행합니다.



02 New Project

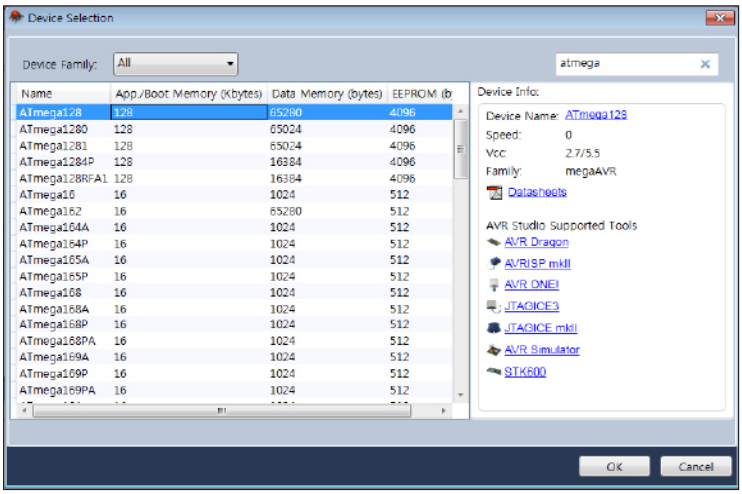
New Project 를 클릭합니다.



03 GCC 선택

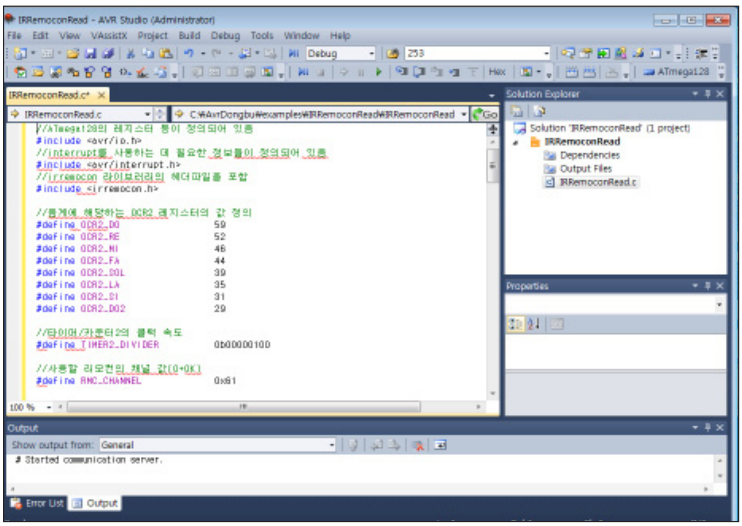
C Executable Project 를 선택합니다.

Name 을 “IRRemoconRead”로 입력하고, Create directory for solution 을 클릭하고, OK 버튼을 누릅니다.



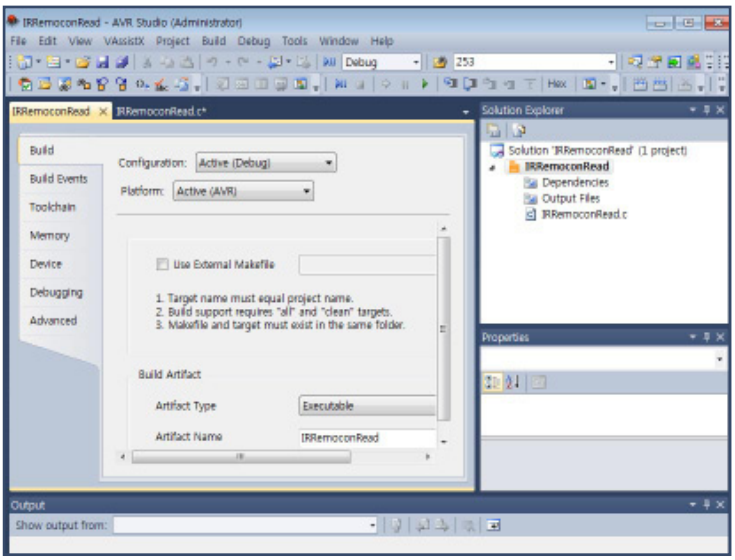
04 ATmega 선택

우측 상단에 atmega 를 검색하여,
리스트중에 ATmega128 을 선택합니다.



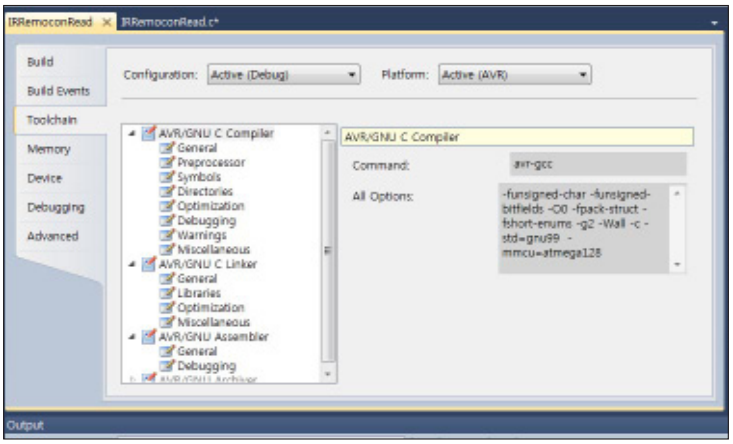
05 Editor

편집창에 코딩을 시작합니다.



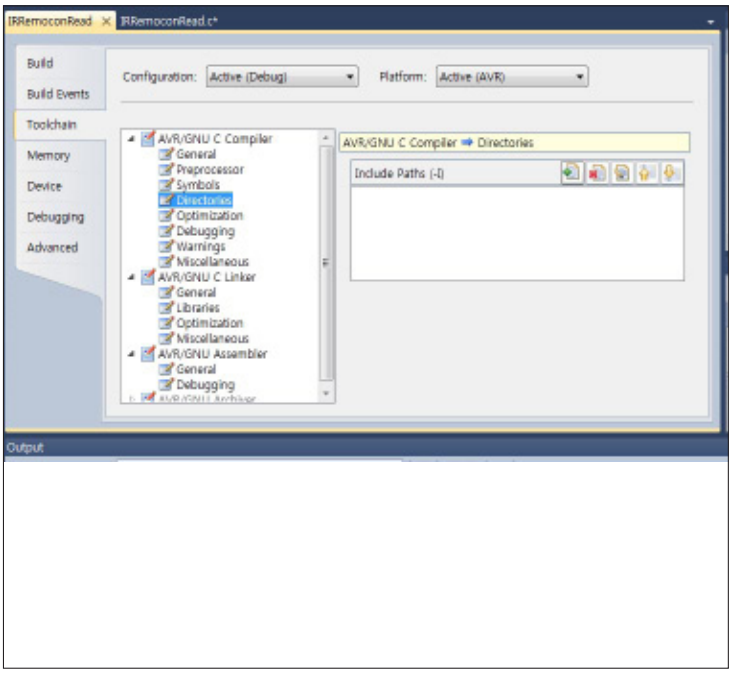
01 설정

AVR Studio 상단 메뉴중에 Project 를 클릭하면 IRRemoconRead (프로젝트 이름별로 다름) Property 메뉴가 생성되어 있습니다. 클릭하면 좌측과 같이 설정화면이 나옵니다.



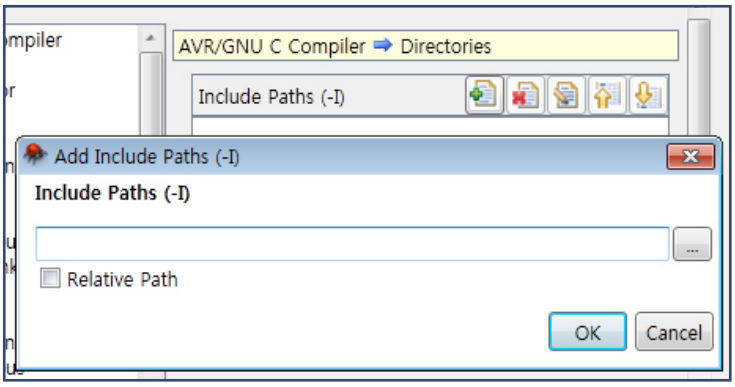
02 Toolchain

좌측 탭버튼중에 Toolchain 을 클릭하여, AVR/GNU C Compiler 을 선택합니다



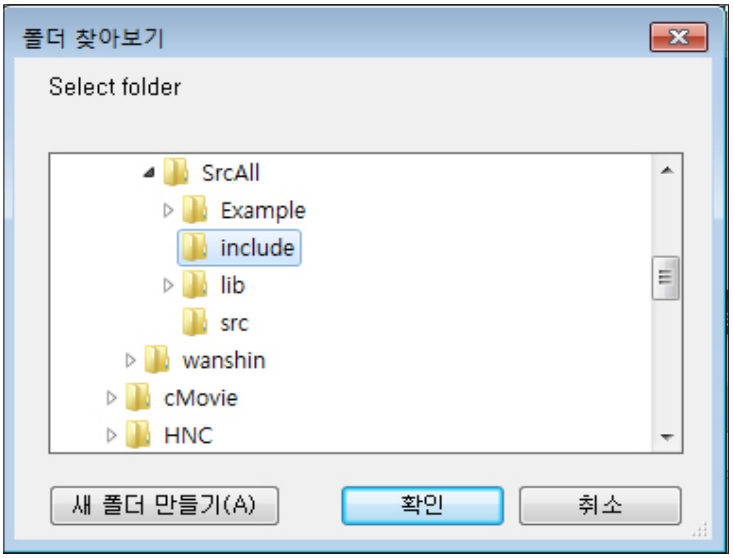
03 C Compiler

C Compiler 폴더중에 Directories 를 선택합니다.



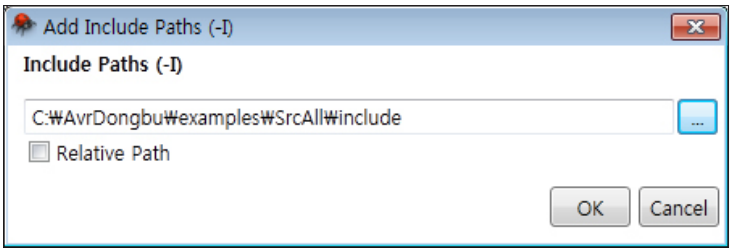
04 Path

Include Paths 오른쪽에 + 버튼을 클릭하면 팝업 입력창이 뜹니다.
Relative Path (상대경로) 체크를 해지하고(5.1에서는 안해도 됨), 우측 폴더 검색 버튼을 클릭합니다.



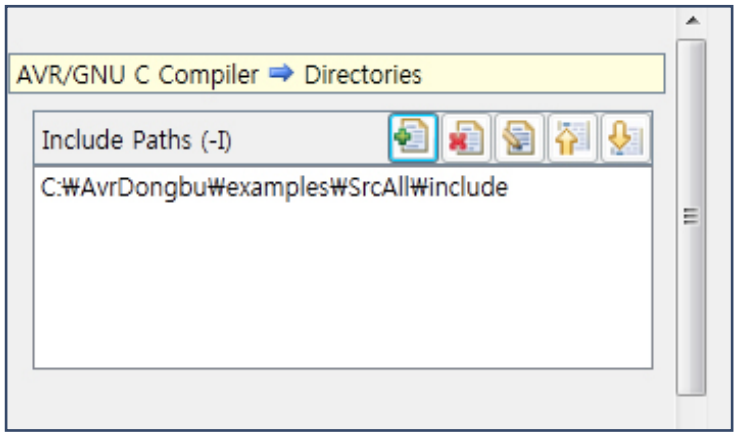
05 Include

제공 라이브러리 폴더중에 include 를 선택합니다.



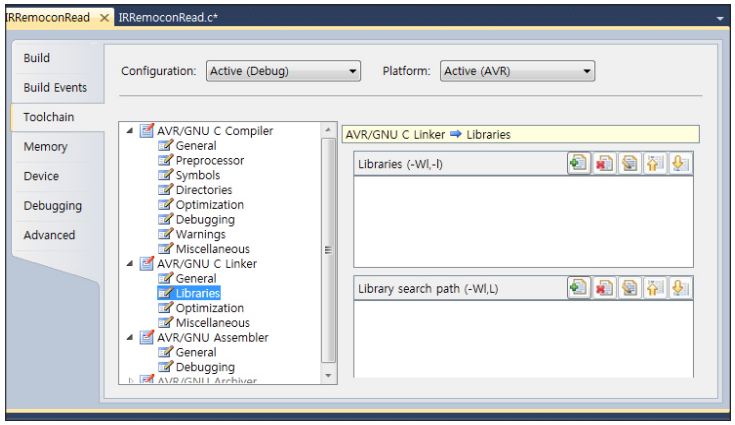
06 Include 완료

Include 폴더 위치가 지정됩니다. OK 버튼을 클릭합니다.



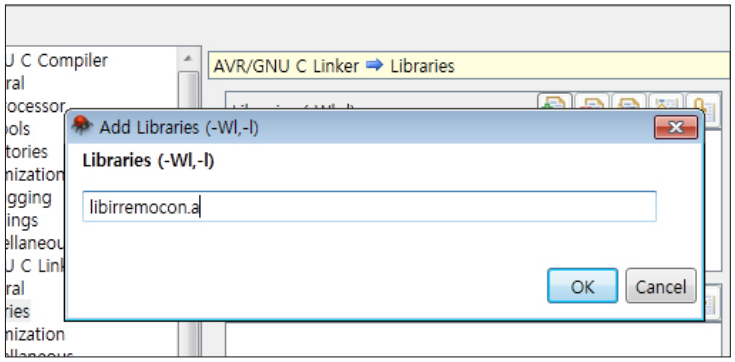
07 Path 완료

Include 경로지정이 완료되었습니다.



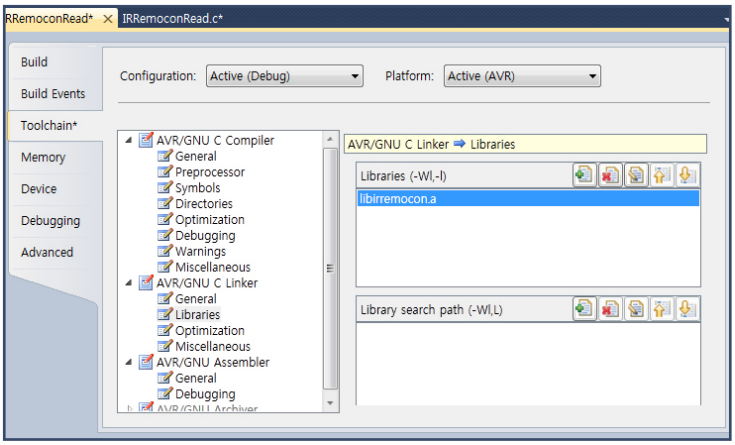
08 C Linker

C Linker 폴더중에 Libraries 를 선택합니다.



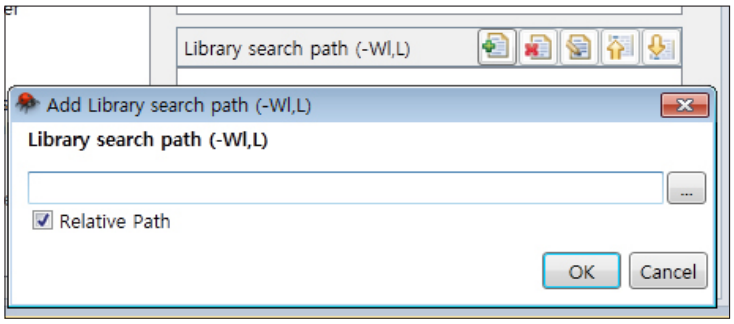
09 Libraries

Libraries 창 우측 + 를 클릭하면,
Libraries 파일명 입력창이 나옵니다.
libirremocon.a 를 직접 입력합니다.
OK 버튼을 클릭합니다.



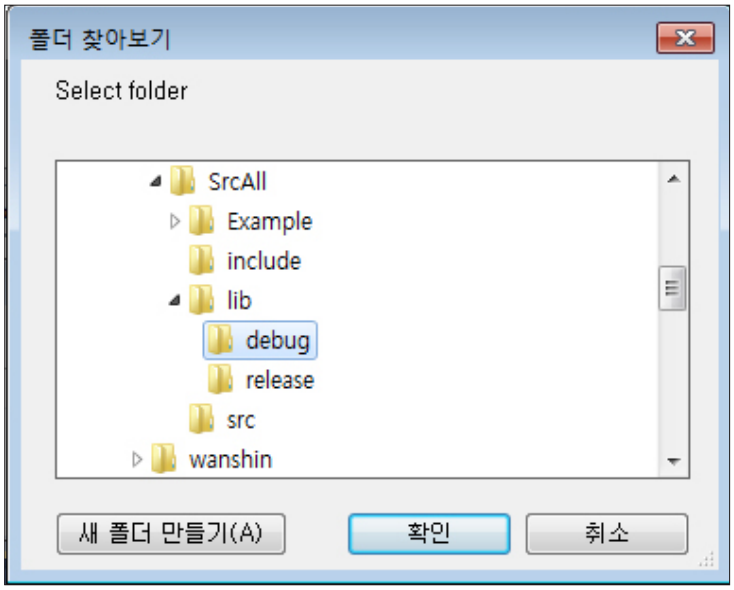
10 Search

리모컨 라이브러리 파일 libaccgyro.a 링크가 완료되었습니다.
다음으로 Library search path 를 설정합니다. 우측 + 버튼을 클릭합니다.



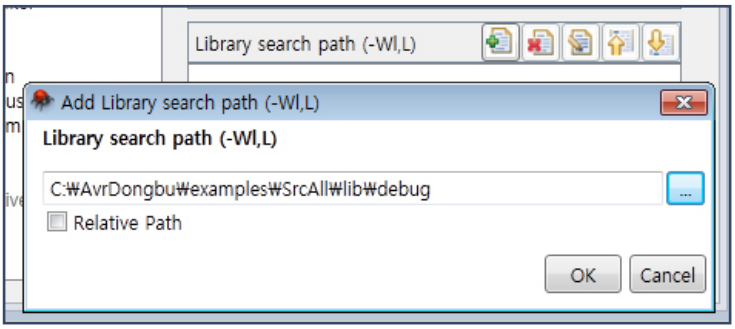
11 Path

Library search path 팝업창 우측에 폴더 검색 버튼을 클릭합니다.



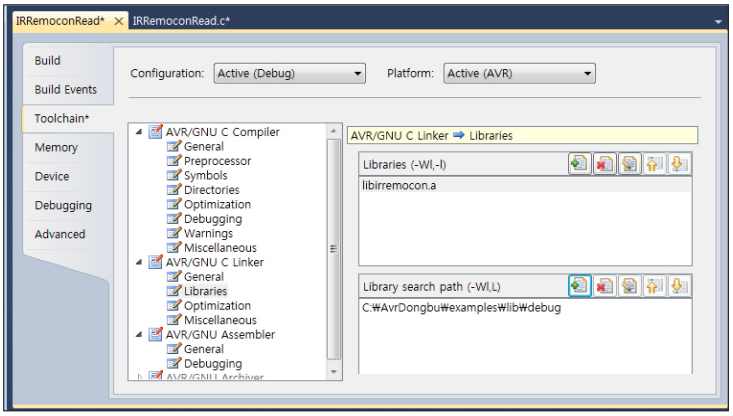
12 Debug

제공 라이브러리 폴더에서 lib 하위 폴더에 debug 를 선택합니다.



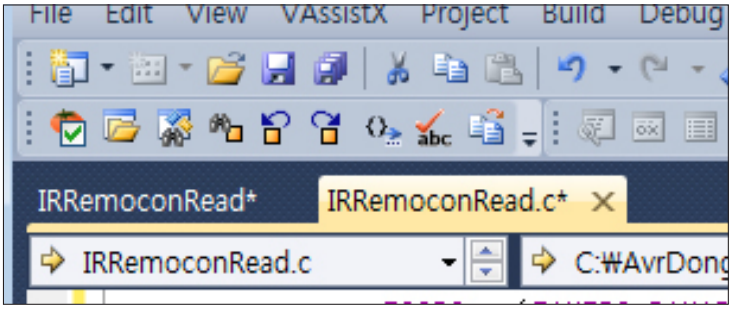
13 Path 완료

Relative Path(상대경로지정) 체크박스를 해지합니다.
Library search path 경로지정이 완료되었습니다.
OK 버튼을 클릭합니다.



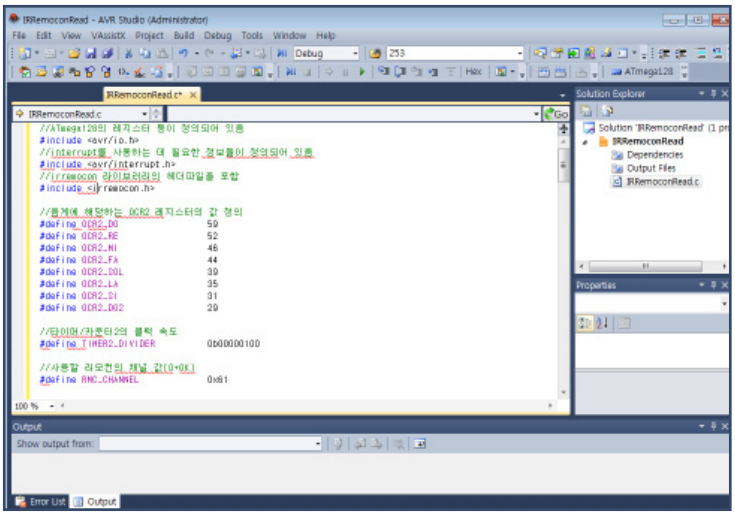
14 Link 완료

C Linker 라이브러리파일과 경로지정이 완료되었습니다



15 저장

저장 버튼을 누르면 현재 설정값이 저장됩니다. 탭바 이름이 IRRemoconRead * -> IRRemoconRead 로 변경됩니다.



16 코딩

IRRemocon.c 탭바를 선택하고, 아래 제공된 예제를 입력합니다.

예제

DRC에 장착된 적외선 리모컨 모듈을 통해서 리모컨 값을 받아, 들어온 키 값에 따라서 버저 음계를 연주하는 예제입니다. 버저에 관련된 내용은 Chapter 9를 참조하세요.

irremocon.h

```
#ifndef IRREMOCON_H_
#define IRREMOCON_H_

#ifdef __IRREMOCON_C
        #define IRREMOCON_EXT
#else
        #define IRREMOCON_EXT extern
#endif

//리모컨 데이터의 기본 값
#define RMC_ERROR_DATA                  0xFE
//리모컨 눌린 길이의 최대 값
#define RMC_LENGTH_MAX                  240

//적외선 리모컨 수신을 위해 외부 인터럽트 핀과 타이머/카운터1을 설정하는 함수.
IRREMOCON_EXT void RMC_Init(void);
//적외선 리모컨의 채널 ucChannel에서 들어오는 키의 값과 눌린 길이를 ucData, ucLength에 저장하는 함수.
IRREMOCON_EXT void RMC_ReadRemocon(unsigned char ucChannel, unsigned char *ucData, unsigned char *ucLength);

#endif /* IRREMOCON_H_ */
```

IRRemoconRead.c

```
//ATmega128의 레지스터 등이 정의되어 있음
#include <avr/io.h>
//interrupt를 사용하는 데 필요한 정보들이 정의되어 있음
#include <avr/interrupt.h>
//irremocon 라이브러리의 헤더파일을 포함
#include <irremocon.h>

//음계에 해당하는 OCR2 레지스터의 값 정의
#define OCR2_DO                 59
#define OCR2_RE                 52
#define OCR2_MI                 46
#define OCR2_FA                 44
#define OCR2_SOL                39
#define OCR2_LA                 35
#define OCR2_SI                 31
#define OCR2_DO2                29

//타이머/카운터2의 클럭 속도
#define TIMER2_DIVIDER          0b00000100

//사용할 리모컨의 채널 값(0+OK)
#define RMC_CHANNEL             0x61

//사용할 리모컨의 키 값
#define RMC_KEY_1               1
#define RMC_KEY_2               2
#define RMC_KEY_3               3
#define RMC_KEY_4               4
#define RMC_KEY_5               5
#define RMC_KEY_6               6
#define RMC_KEY_7               7
#define RMC_KEY_8               8

int main(void)
{
        //8비트 변수 선언
        unsigned char ucData, ucDataOld, ucLength;

        //버저 핀(PB7)의 입출력 방향 설정
        DDRB |= 0b10000000;

        //포트 C의 입출력 방향 설정
```

```
DDRC = 0b01111111;
//포트 C의 출력 값 설정
PORTC = 0b01111111;

//타이머/카운터 2 초기화
TCNT2 = 0;
OCR2 = 0;
TCCR2 = 0;

//전체 인터럽트를 비활성화
cli();

//리모컨 수신 모듈을 사용하기 위해 초기화
RMC_Init();

//전체 인터럽트를 활성화
sei();

//ucDataOld를 기본 값으로 초기화
ucDataOld = RMC_ERROR_DATA;

//while 안의 내용을 무한 반복
while(1)
{
        //ucData, ucLength로 리모컨 값 읽어오기
        RMC_ReadRemocon(RMC_CHANNEL, &ucData, &ucLength);

        //리모컨 값이 전 값과 다르다면
        if(ucDataOld != ucData){
                //1번 키가 눌렸을 때 '도'음을 냄
                if(ucData == RMC_KEY_1){
                        TCNT2 = 0;
                        OCR2 = OCR2_DO;
                        TCCR2 = (TIMER2_DIVIDER)|(1<<WGM21)|(1<<COM20);
                }
                //2번 키가 눌렸을 때 '레'음을 냄
                else if(ucData == RMC_KEY_2){
                        TCNT2 = 0;
                        OCR2 = OCR2_RE;
                        TCCR2 = (TIMER2_DIVIDER)|(1<<WGM21)|(1<<COM20);
                }
                //3번 키가 눌렸을 때 '미'음을 냄
                else if(ucData == RMC_KEY_3){
                        TCNT2 = 0;
                        OCR2 = OCR2_MI;
                        TCCR2 = (TIMER2_DIVIDER)|(1<<WGM21)|(1<<COM20);
                }
```

```
                    //4번 키가 눌렸을 때 '파'음을 냄
                    else if(ucData == RMC_KEY_4){
                            TCNT2 = 0;
                            OCR2 = OCR2_FA;
                            TCCR2 = (TIMER2_DIVIDER)|(1<<WGM21)|(1<<COM20);
                    }
                    //5번 키가 눌렸을 때 '솔'음을 냄
                    else if(ucData == RMC_KEY_5){
                            TCNT2 = 0;
                            OCR2 = OCR2_SOL;
                            TCCR2 = (TIMER2_DIVIDER)|(1<<WGM21)|(1<<COM20);
                    }
                    //6번 키가 눌렸을 때 '라'음을 냄
                    else if(ucData == RMC_KEY_6){
                            TCNT2 = 0;
                            OCR2 = OCR2_LA;
                            TCCR2 = (TIMER2_DIVIDER)|(1<<WGM21)|(1<<COM20);
                    }
                    //7번 키가 눌렸을 때 '시'음을 냄
                    else if(ucData == RMC_KEY_7){
                            TCNT2 = 0;
                            OCR2 = OCR2_SI;
                            TCCR2 = (TIMER2_DIVIDER)|(1<<WGM21)|(1<<COM20);
                    }
                    //8번 키가 눌렸을 때 '도'음을 냄
                    else if(ucData == RMC_KEY_8){
                            TCNT2 = 0;
                            OCR2 = OCR2_DO2;
                            TCCR2 = (TIMER2_DIVIDER)|(1<<WGM21)|(1<<COM20);
                    }
                    //1~8번 키가 안 눌린 경우 소리 멈춤
                    else{
                            TCNT2 = 0;
                            OCR2 = 0;
                            TCCR2 = 0;
                    }
            }

            //현재 데이터 값을 다른 변수에 저장
            ucDataOld = ucData;
    }

    //함수의 형태에 맞게 정수값 반환
    return 1;
}
```

예제설명

먼저 libirremocon.a의 헤더 파일인 irremocon.h에 대해서 설명합니다.

irremocon.h

```
#define RMC_ERROR_DATA          0xFE
#define RMC_LENGTH_MAX          240
```

리모컨에서 아무런 입력이 없을 때나 입력을 받던 중 예상치 못한 신호가 들어왔을 경우, 리모컨의 데이터는 0xFE라는 기본 값으로 설정됩니다. 즉 아무 리모컨 입력이 없을 때 리모컨 값을 읽은 경우 ucData에는 0xFE라는 값이 들어 있습니다. 그 기본 값을 매크로로 선언해서 사용하기 위해서 RMC_ERROR_DATA라는 값을 선언합니다.
리모컨 값을 읽을 때 ucLength에 들어가는 값은 1당 125밀리초의 시간을 나타냅니다. 그런데 ucLength는 부호 없는 8비트 정수이기 때문에 최대 값을 255까지밖에 가질 수 없습니다. 그래서 libirremocon.a에서는 최대 값을 240(30초)으로 제한하여, 30초 이상 리모컨이 눌려 있는 경우 무조건 키 길이가 240으로 읽히도록 했습니다. 그 최대값을 매크로로 선언해서 사용하기 위해 RMC_LENGTH_MAX를 선언합니다.

```
IRREMOCON_EXT void RMC_Init(void);
IRREMOCON_EXT void RMC_ReadRemocon(unsigned char ucChannel, unsigned char *ucData, unsigned char *ucLength);
```

RMC_Init은 적외선 리모컨 수신 모듈이 연결되는 외부인터럽트 4번 핀(PE4)과, 리모컨 신호의 시간을 재는 데에 사용되는 타이머/카운터1을 초기화 하는 함수입니다. RMC_ReadRemocon은 현재 리모컨 수신 모듈로 들어온 데이터 값을 읽는 함수입니다. 매개 변수로 들어가는 ucChannel은 데이터를 받을 채널을 지정해주는 매개 변수이며, 만약 ucChannel과 같은 채널에서 데이터가 들어왔다면 포인터 ucData와 ucLength에 그 키 값과 키가 눌린 길이가 저장됩니다. 만약 데이터가 들어오지 않은 상태라면 ucData에는 RMC_ERROR_DATA(0xFE), ucLength에는 0이 저장될 것입니다.

IRRemoconRead.c

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <irremocon.h>
```

avr/io.h 헤더 파일에는 DDRX, PORTX 레지스터 등 AVR 제어를 위해 필요한 레지스터 등이 정의되어 있습니다. avr/interrupt.h 파일에는 인터럽트를 사용하기 위한 정보가 정의되어 있습니다. 또 irremocon.h 헤더 파일에는 libirremocon.a 라이브러리 파일에 들어 있는 함수의 원형이 정의되어 있습니다. 현재 소스 코드에서 사용할 수 있도록 #include 문을 사용하여 포함시켜 줍니다.

```
#define OCR2_DO                 59
                    ...
#define OCR2_DO2                29
```

사용할 음계에 따른 OCR2 값을 #define으로 선언합니다. 이 값들은 음을 재생하기 위해서 타이머/카운터2의 OCR2 레지스터에 값을 넣을 때 사용됩니다. 단순하게 59 같은 숫자를 대입하면 쉽게 '도'라는 음과 연결시킬 수 없지만, OCR2_DO를 대입한다면 '도'음이 재생될 거라고 쉽게 알아볼 수 있습니다. 왜 이런 값들이 선언되는지는 Chapter 9의 H/W 원리를 참조하세요.

```
#define RMC_CHANNEL                    0x61
```

H/W 원리에서 다루었듯이, 리모컨을 설정함에 따라서 채널 값은 0x61부터 0x6A까지가 될 수 있습니다. 이번 예제에서는 리모컨의 0번 버튼 + OK를 눌렀을 때 설정되는 기본값인 0x61을 채널 값으로 사용합니다.

```
#define RMC_KEY_1                      1
                              ...
#define RMC_KEY_8                      8
```

사용할 리모컨의 키와 그에 해당하는 키 값을 선언합니다. 1이라고 쓰인 리모컨 키를 누르면 리모컨 데이터 1이 읽히게 됩니다. 지금 사용할 키들은 키와 데이터 값이 일치해서 별로 선언하는 의미가 없어보이지만, 방향키나 OK 등의 다른 키를 사용할 때는 키와 데이터 값을 선언해두면 매번 찾아보는 일 없이 사용할 수 있어 편리합니다.

```
unsigned char ucData, ucDataOld, ucLength;
```

메인 함수에서 사용할 변수를 선언합니다. ucData는 현재 while 루프에서 읽은 리모컨의 값이고, ucDataOld는 직전의 while 루프에서 읽은 리모컨의 값입니다. 매번 while 루프의 마지막에서 ucDataOld는 ucData의 값으로 업데이트 됩니다. 이 두 변수는 타이머/카운터2 리모컨이 눌리거나 떼져서 리모컨 값이 바뀐 순간에만 값을 바꾸기 위해서 사용합니다. ucLength는 리모컨이 눌린 길이를 읽어올 변수입니다.

```
DDRB |= 0b10000000;
DDRC = 0b01111111;
PORTC = 0b01111111;
```

포트 B, C를 설정합니다. 포트 B의 7번 핀(PB7)은 타이머/카운터2의 OC2 출력핀입니다. 타이머/카운터2에서 OC2 출력핀을 사용한다고 설정했더라도 DDRB의 7번 비트가 출력으로 설정되어 있어야 실제로 출력이 나오기 때문에, DDRB의 7번 비트를 1로 만들어주어 출력으로 설정합니다. 포트 C는 LED가 연결된 포트이므로 DDRC의 LED에 해당하는 7개 비트에 1을 넣어 출력으로 만들고, PORTC에도 7개 비트에 1을 넣어 LED를 모두 끕니다.

```
TCNT2 = 0;
OCR2 = 0;
TCCR2 = 0;
```

타이머/카운터2의 레지스터를 초기화합니다. 앞서 설명에서 언급된 레지스터 중 TCNT2, OCR2, TCCR2를 초기값인 0으로 만들어 줍니다. 이 시점에서 타이머/카운터2는 아직 동작하지 않습니다.

```
cli();
RMC_Init();
sei();
```

인터럽트에 대한 설정을 바꾸는 중에 인터럽트가 발생해 예상치 못한 오동작이 발생하는 것을 방지하기 위해서 우선 cli() 명령을 통해서 전체 인터럽트를 비활성화 시킵니다. 그 후 RMC_Init 함수를 사용해서 리모컨 신호를 받기 위한 외부 인터럽트 4번 핀과 타이머/카운터1을 초기화합니다. 초기화가 끝난 후에는 sei() 명령을 통해서 전체 인터럽트를 활성화 합니다.

```
ucDataOld = RMC_ERROR_DATA;
```

앞서 설명했듯이, ucDataOld 변수에는 while문에서 직전에 읽어온 리모컨의 값이 저장됩니다. 그런데 아직 while문에 진입하기 전이라 리모컨 값을 읽기 전이기 때문에, 기본 값인 RMC_ERROR_DATA를 ucDataOld에 저장합니다.

```
RMC_ReadRemocon(RMC_CHANNEL, &ucData, &ucLength);
```

이제부터는 while문 안쪽의 내용입니다. RMC_ReadRemocon 함수를 사용해서 리모컨의 값을 읽어옵니다. 첫 번째 매개변수에는 원하는 리모컨 채널인 RMC_CHANNEL을 넣어주고, 두 번째, 세 번째 매개 변수에는 ucData와 ucLength의 주소를 넣어서 RMC_CHANNEL 채널에서 들어온 데이터와 눌린 길이가 ucData와 ucLength에 저장되게 합니다.

```
if(ucDataOld != ucData){
        …
}
```

직전 while 루프에서 읽어온 리모컨 값(ucDataOld)과 방금 읽은 리모컨 값(ucData)을 비교해서, 서로 다른 경우에 if문의 안쪽 내용을 실행합니다. 리모컨 키가 눌리는 순간이나 눌렸던 키가 떼지는 순간에만 타이머/카운터2의 설정을 바꾸기 위한 부분입니다.

```
if(ucData == RMC_KEY_1){
        TCNT2 = 0;
        OCR2 = OCR2_DO;
        TCCR2 = (TIMER2_DIVIDER)|(1<<WGM21)|(1<<COM20);
}
                                        …
else if(ucData == RMC_KEY_8){
        TCNT2 = 0;
        OCR2 = OCR2_DO2;
        TCCR2 = (TIMER2_DIVIDER)|(1<<WGM21)|(1<<COM20);
}
```

ucData가 RMC_KEY_1과 같다면, 즉 1번 키가 눌렸다면 타이머/카운터2의 값을 바꾸어 버저가 '도'음을 재생하게 합니다. TCNT2는 0으로 초기화 하고, 값에 따라서 주파수가 달라지는 OCR2에는 '도'에 해당하는 미리 정의된 값을 대입합니다. TCCR2에서는 WGM21을 1로 만들어 CTC 모드로 설정하고, COM20을 1로 만들어 비교 매치가 일어날 때마다 OC2 핀을 토글하게 하고, TIMER2_DIVIDER에 정의된 값으로 CS22:0을 설정해 원하는 비율로 클럭을 나누도록 합니다. 우리 예제에서는 CS22:0을 0b100으로 설정해서 256으로 클럭을 나눕니다.

밑에 이어지는 else if문도 리모컨 키와 OCR2 값만 바뀌었을 뿐 동일합니다. if문과 else if문을 통해서 1번 키 – 도, 2번 키 – 레, 3번 키 – 미, 4번 키 – 파, 5번 키 – 솔, 6번 키 – 라, 7번 키 – 시, 8번 키 – 높은 도를 재생하도록 합니다.

```
else{
        TCNT2 = 0;
        OCR2 = 0;
        TCCR2 = 0;
}
```

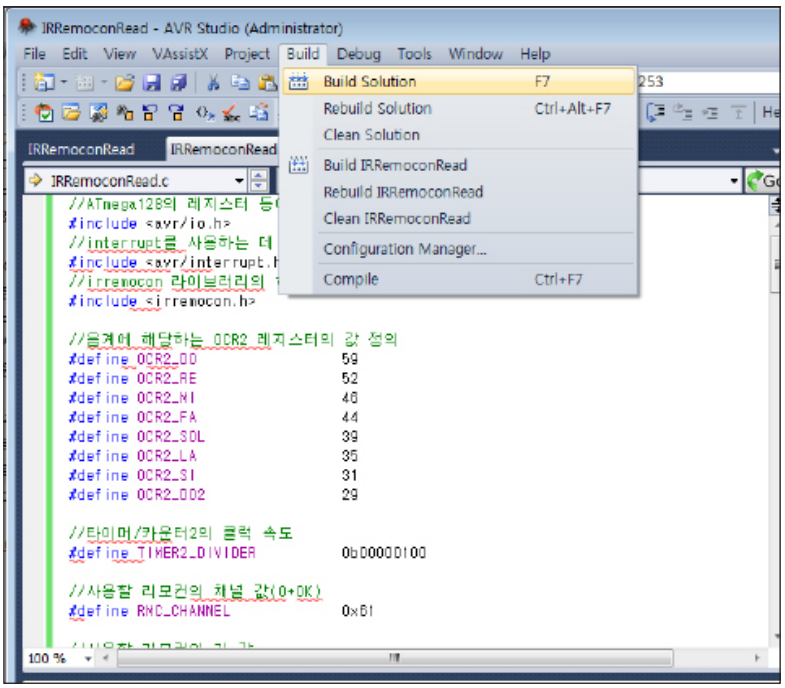
리모컨이 눌려있지 않거나 눌려 있지만 1~8번 키가 아닐 경우에는, 타이머/카운터2 레지스터를 모두 0으로 만들어서 중지시킵니다. 이렇게 하면 버저의 울림이 멈추게 됩니다.

```
ucDataOld = ucData;
```

while 루프가 끝날 때마다 현재 루프에서 읽은 ucData의 값을 ucDataOld에 대입해, 다음 루프에서 비교하는 데에 사용합니다.

실행

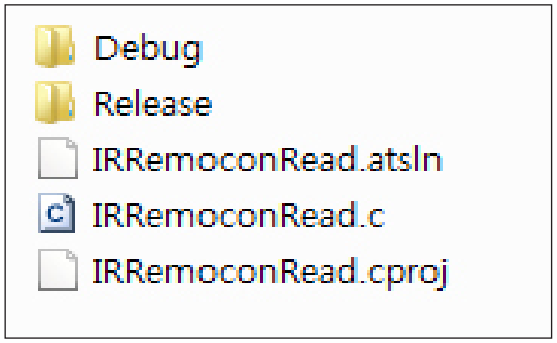
적외선 리모컨 수신 모듈을 DRC에 장착한 후 펌웨어를 실행하면, 리모컨 키를 누를 때 해당하는 버저 음이 재생됩니다.



01 Build

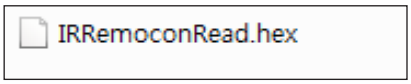
Build 메뉴를 선택하여 Build Solution을 클릭하여 Build 시킵니다.

1 succeeded 0 failed 가 나오면 에러없이 컴파일 완료됐다는 것을 의미합니다.



02 Debug 폴더

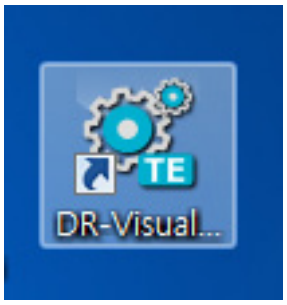
Build 후에 소스가 있는 폴더에 들어가면 Debug 폴더가 생성 되어있습니다. 클릭하여 폴더 안의 내용을 봅니다.



03 hex 확인

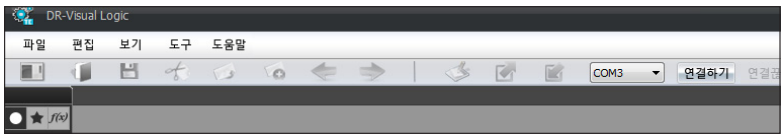
Debug 폴더 안의 hex 파일 확인합니다. Hex 파일이 제어기에 포팅시키는 파일입니다.

DR-Visual Logic 에서 펌웨어 복구를 실시합니다.



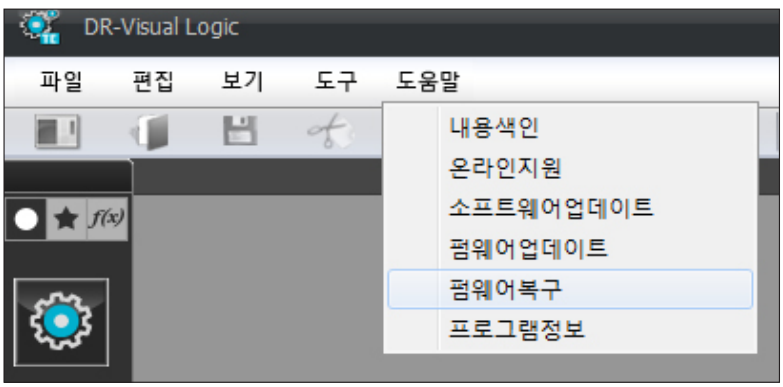
04 Visual Logic

DR-Visual Logic 을 실행시킵니다.



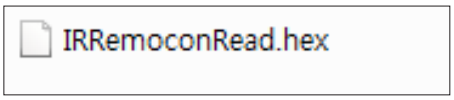
05 연결하기

COM 번호를 설정하고 연결하기를 클릭합니다.



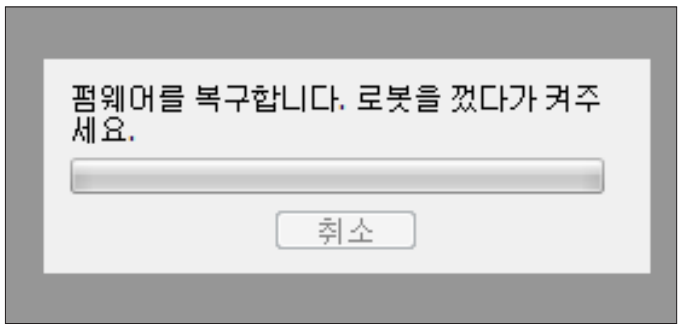
06 펌웨어 복구

도움말 메뉴에서 펌웨어복구를 클릭합니다.



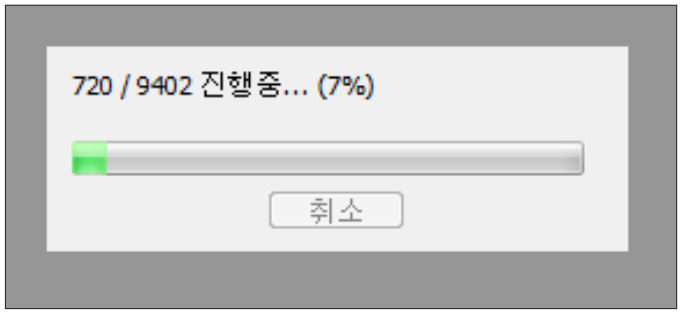
07 불러오기

Debug 폴더에서 Hex 파일을 불러옵니다



08 펌웨어복구

로봇을 껐다가 켜주세요 라는 문구가 나옵니다.
제어기의 전원버튼을 한번 껐다가 다시 켜주세요.



09 복구시작

펌웨어 복구가 시작됩니다.

10 실행

적외선 리모컨 수신 모듈을 DRC에 장착한 후 펌웨어를 실행하면, 리모컨 키를 누를 때 해당하는 버저 음이 재생됩니다.

PART 05

로봇에 응용하기

16축 휴머노이드 모션 실행하기

herkulex 라이브러리와 모션 데이터 파일을 사용해 Hovis Lite 16축 로봇에서 모션을 실행해봅니다.

H/W 원리

앞서 Chapter 8에서 다루었듯이, DRC에 장착된 ATmega128에는 USART0, USART1의 두 USART 포트가 있으며 이 중 USART0은 HerkuleX와의 통신에 사용되는 4핀 포트 5개와 병렬적으로 연결되어 있습니다. 제어하고 싶은 HerkuleX를 케이블로 DRC와 연결하면 USART0을 통해서 직접 패킷을 보내고 받을 수 있습니다.
이번 예제에서 사용하는 패킷에 사용된 명령을 요약하면 아래와 같습니다.

이름	CMD	설명
RAM_WRITE	0x03	HerkuleX의 휘발성 레지스터 영역에 데이터를 씁니다.
S_JOG	0x06	HerkuleX의 위치/속도를 제어하여 움직이게 합니다. I_JOG는 각 모터마다 이동 시간을 다르게 할 수 있는 반면, S_JOG는 모든 시간을 같은 시간동안 움직여서 목표 위치에 도달합니다. I_JOG와 S_JOG 패킷에서 시간을 나타내기 위해서는 1바이트 크기의 데이터를 사용하는데, 이때 데이터는 1당 11.2ms를 나타냅니다. 즉 I_JOG 패킷에 50의 시간 데이터가 들어가 있다면 실제로 모터는 50*11.2=560ms 동안 목표 위치로 이동합니다.

이번 예제에서 사용된 휘발성 레지스터 영역의 항목을 요약하면 아래와 같습니다.

이름	주소	크기	설명
Torque Control	52	1	토크 인가 상태를 제어합니다. 0x00 : 자유롭게 움직이는 힘이 풀린 상태 0x40 : 힘은 풀렸으나 돌아가는 데에 저항이 있는 상태 0x60 : 토크가 인가되어 스스로 구동하는 상태

프로토콜과 레지스터 맵에 대한 더 자세한 내용은 HerkuleX 매뉴얼을 참조하세요.

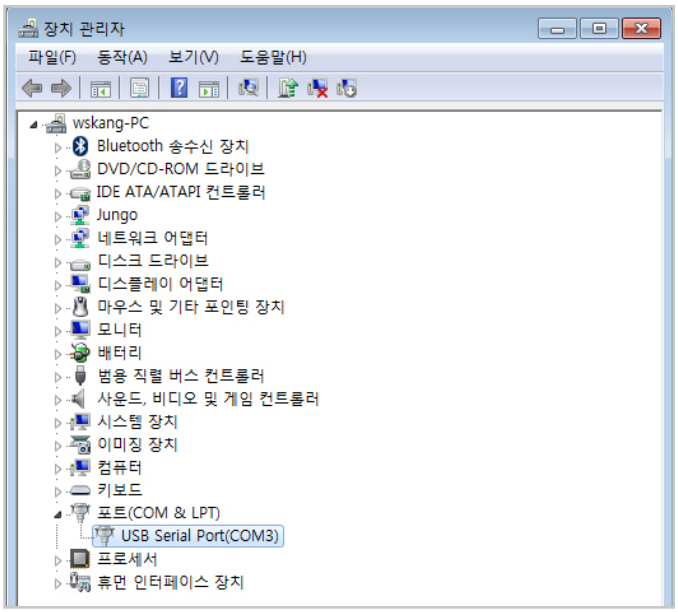
하드웨어 준비하기

01 연결

DRC 제어기 의 USB to Serial Cable 로 PC 와 DRC 를 연결합니다.
16축 휴머노이드 Hovis Lite 를 PC 와 연결합니다.

02 COM port 연결

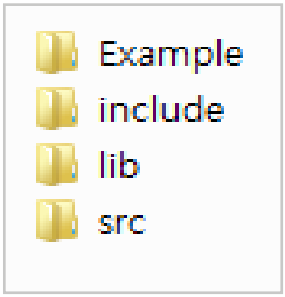
내컴퓨터 〉 속성 〉 장치관리자에서 COM 연결 확인합니다.

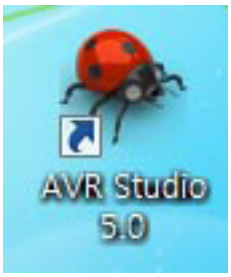


AVR Studio 실행 및 코딩 준비

00 라이브러리

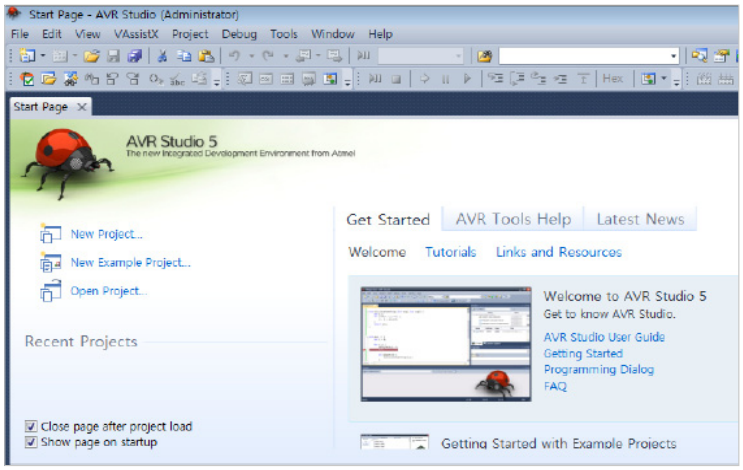
Hovis Lite 를 제어하기 위해서는 기본 라이브러리 파일이 필요합니다.
라이브러리가 포함된 폴더 구조입니다.





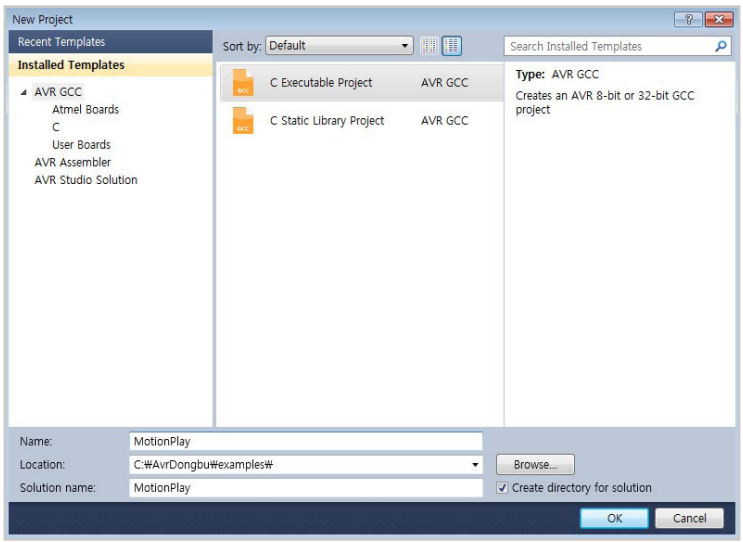
01 실행

AVR Studio 를 실행합니다.



02 New Project

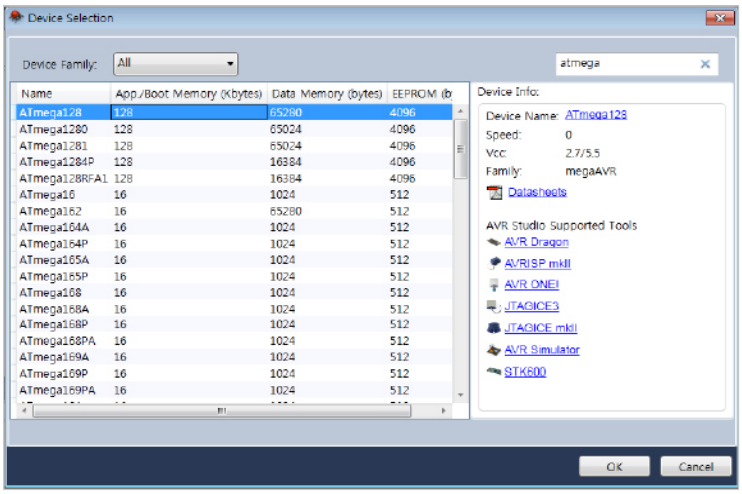
New Project 를 클릭합니다.



03 GCC 선택

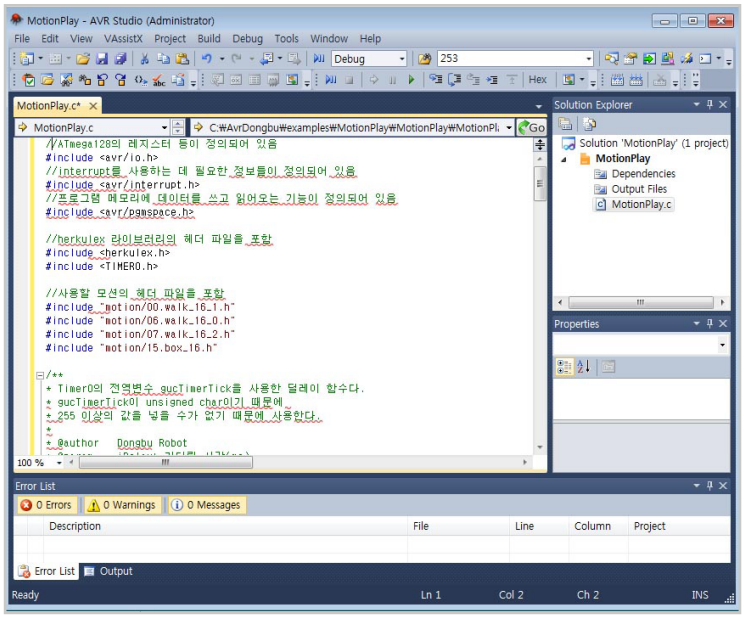
C Executable Project 를 선택합니다.

Name 을 “Motionplay”로 입력하고, Create directory for solution 을 클릭하고, OK 버튼을 누릅니다.



04 ATmega 선택

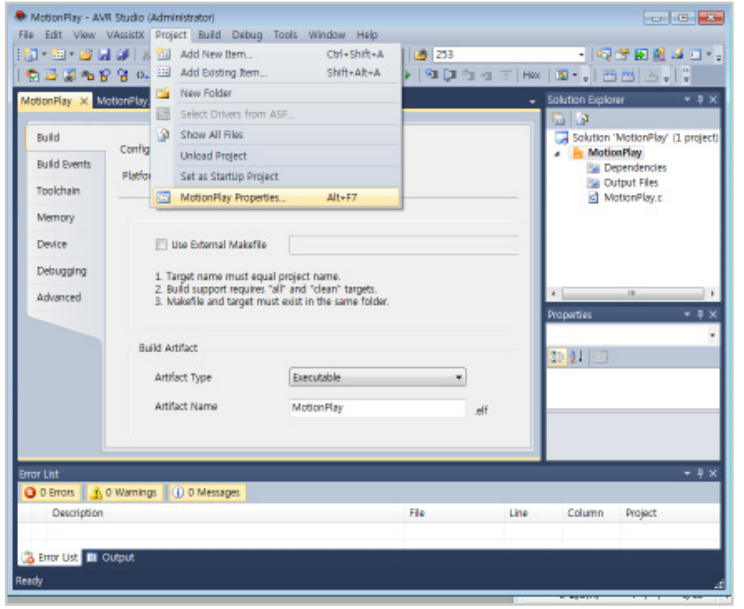
우측 상단에 atmega 를 검색하여, 리스트중에 ATmega128 을 선택합니다.



05 Editor

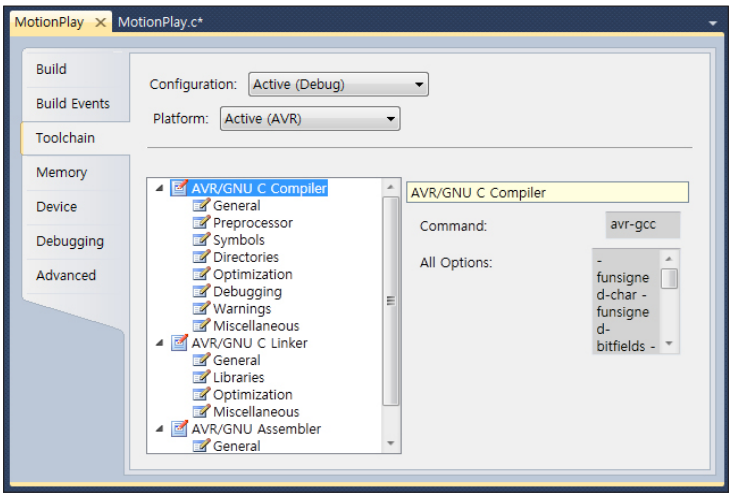
편집창에 코딩을 시작합니다.

라이브러리 링크 및 설정



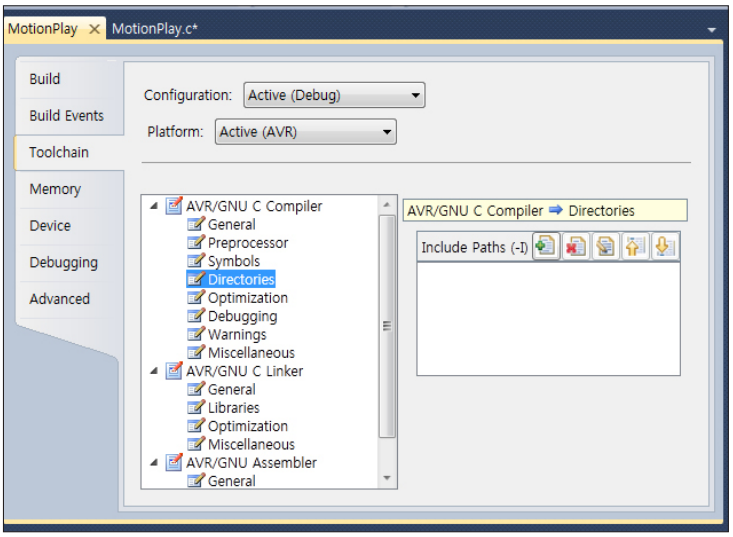
01 설정

AVR Studio 상단 메뉴중에 Project 를 클릭하면 MotionPlay (프로젝트 이름별로 다름) Property 메뉴가 생성되어 있습니다. 클릭하면 좌측과 같이 설정화면이 나옵니다.



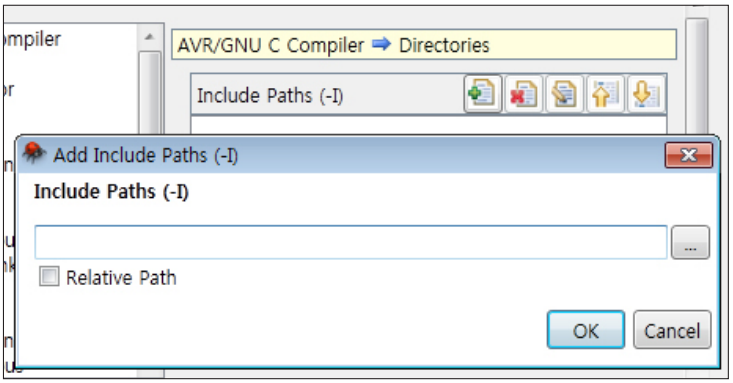
02 Toolchain

좌측 탭버튼중에 Toolchain 을 클릭하여, AVR/GNU C Compiler 을 선택합니다.



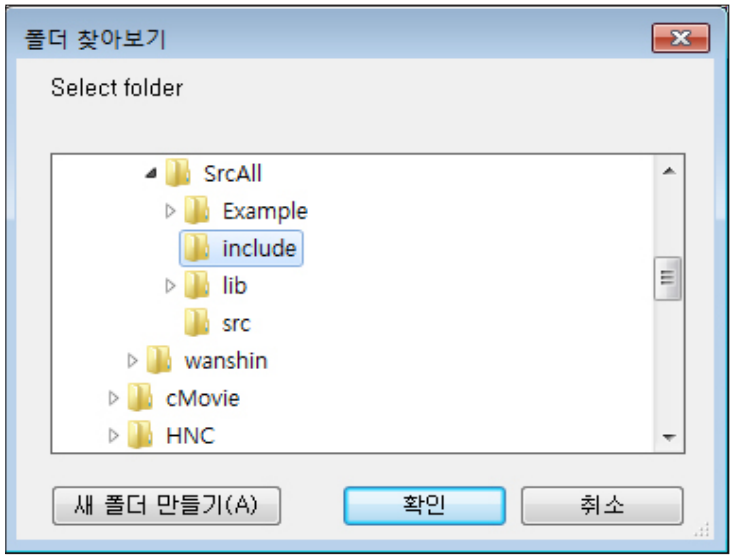
03 C Compiler

C Compiler 폴더중에 Directories 를 선택합니다.



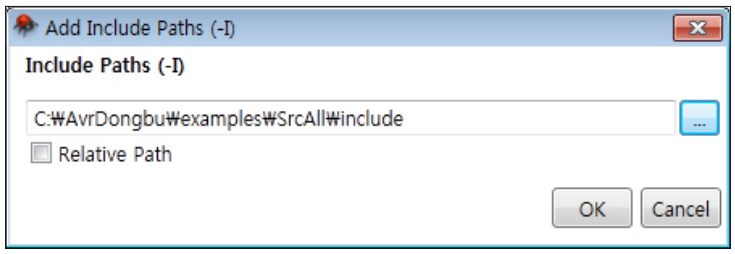
04 Path

Include Paths 오른쪽에 + 버튼을 클릭하면 팝업 입력창이 뜹니다.
Relative Path (상대경로) 체크를 해지하고,
우측 폴더 검색 버튼을 클릭합니다.



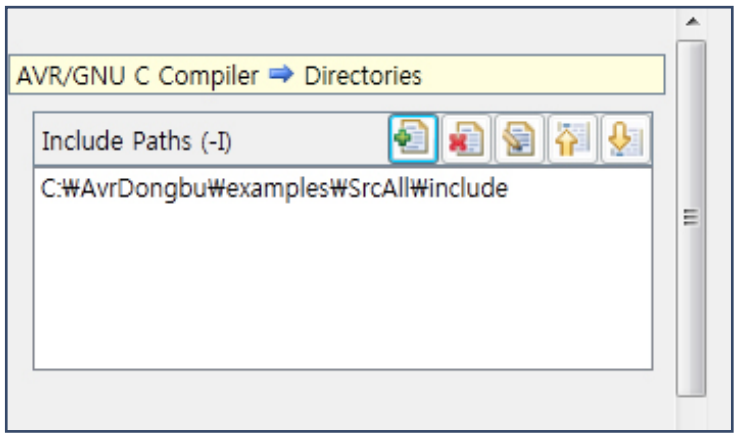
05 Include

제공 라이브러리 폴더중에 include 를 선택합니다.



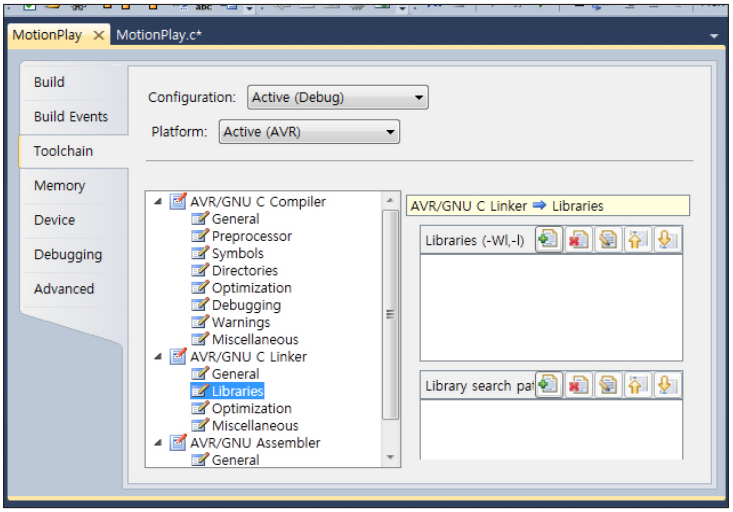
06 Include 완료

Include 폴더 위치가 지정됩니다.
OK 버튼을 클릭합니다.



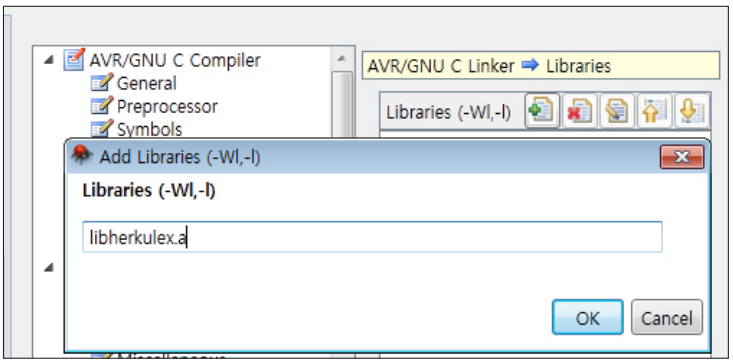
07 Path 완료

Include 경로지정이 완료되었습니다.



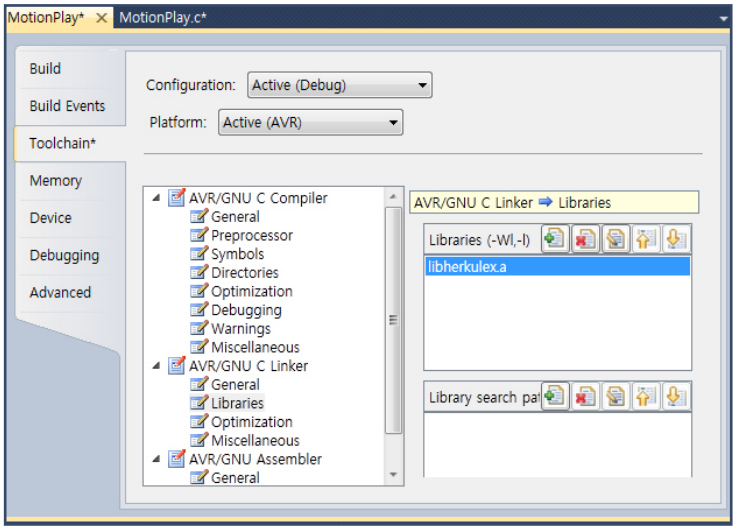
08 C Linker

C Linker 폴더중에 Libraries 를 선택합니다.



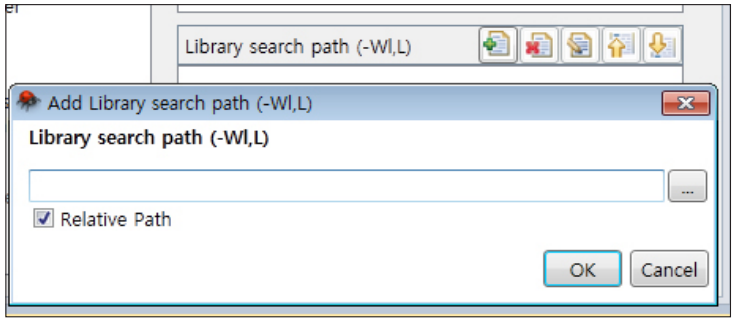
09 Libraries

Libraries 창 우측 + 를 클릭하면, Libraries 파일명 입력창이 나옵니다. libherkulex.a 를 직접 입력합니다.
OK 버튼을 클릭합니다.



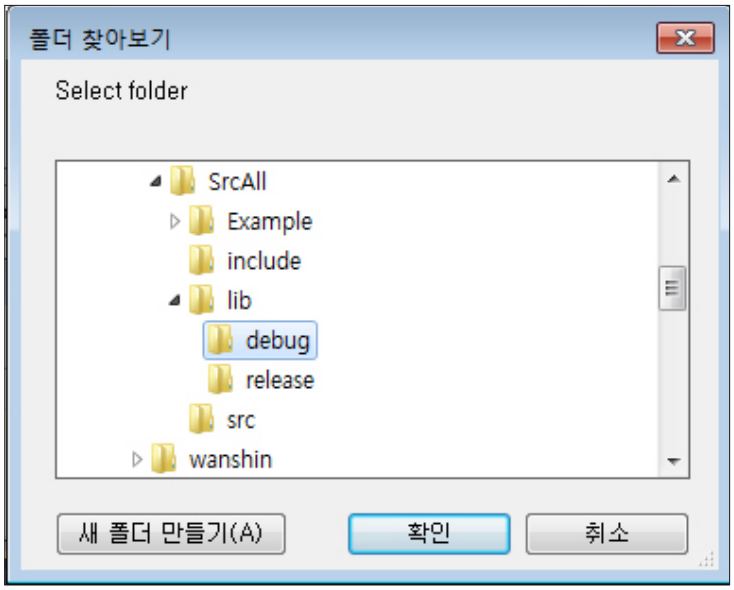
10 Search

서보모터 라이브러리 파일 libherkulex.a 링크가 완료되었습니다.
다음으로
Library search path 를 설정합니다.
우측 + 버튼을 클릭합니다.



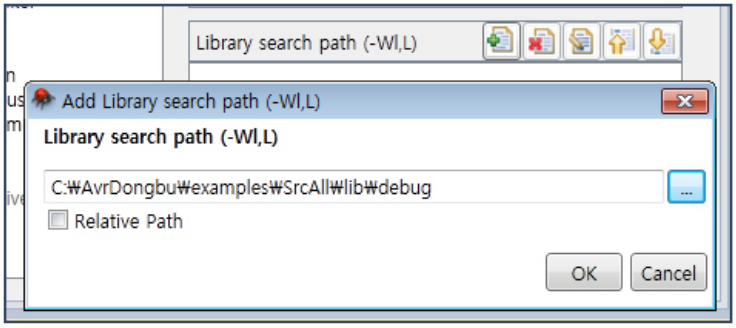
11 Path

Library search path 팝업창 우측에 폴더 검색 버튼을 클릭합니다.



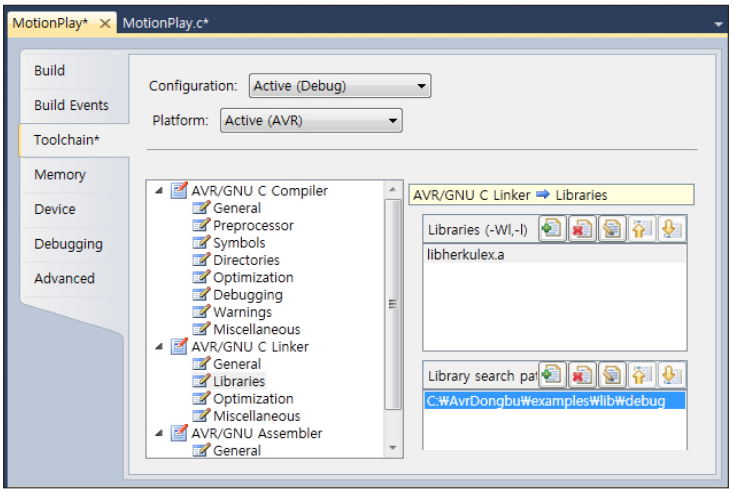
12 Debug

제공 라이브러리 폴더에서 lib 하위 폴더에 debug 를 선택합니다.



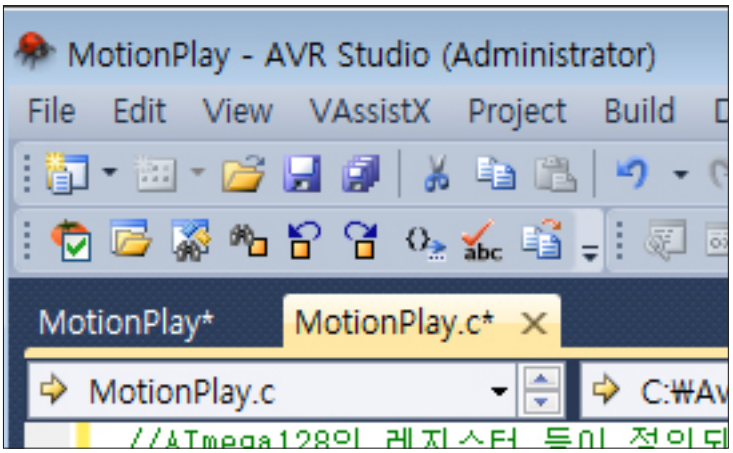
13 Path 완료

Relative Path(상대경로지정) 체크박스를 해지합니다.
Library search path 경로지정이 완료되었습니다.
OK 버튼을 클릭합니다.



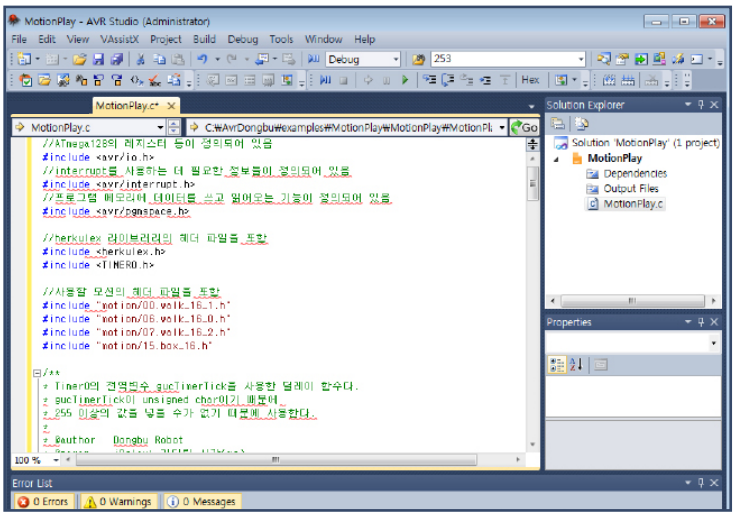
14 Link 완료

C Linker 라이브러리파일과 경로지정이 완료되었습니다.



15 저장

저장 버튼을 누르면 현재 설정값이 저장됩니다. 탭바 이름이 MotionPlay * –〉 MotionPlay 로 변경됩니다.



16 코딩

MotionPlay.c 탭바를 선택하고, 아래 제공된 예제를 입력합니다.

예제

로봇의 전원을 켜면 10발자국 전진하고 복싱을 하는 것을 반복하는 예제입니다.

00.Walk_16_1.h

```
#ifndef WALK_16_1_H_
#define WALK_16_1_H_

//현재 모션에서 플레이할 프레임의 수
prog_int16_t walk_16_1_FrameCnt = 48;

//현재 모션에서 제어할 모터의 수
prog_int16_t walk_16_1_MotorCnt = 16;

//프레임에 해당하는 모터의 목표 위치 값
prog_int16_t walk_16_1_MotorData[48][16] = {
        {269,398,362,828,730,559,532,398,261,671,533,544,699,759,430,533},
        {267,381,379,832,713,575,523,402,262,674,526,536,697,761,426,526},
        {264,364,396,836,696,593,515,406,263,676,519,527,695,762,423,519},
        {262,346,414,840,678,610,506,410,265,678,512,518,692,764,419,512},
        {260,328,431,844,660,628,497,414,267,680,505,509,690,765,415,505},
        {257,311,449,848,643,645,488,417,269,682,498,501,687,766,411,498},
        {254,294,465,852,626,662,480,421,271,684,491,492,685,767,408,491},
        {252,279,481,856,610,678,472,425,273,686,484,484,682,768,404,484},
        {249,264,496,860,596,693,465,430,276,687,478,477,679,769,401,478},
        {246,251,509,864,582,706,458,434,278,689,473,470,676,769,397,473},
        {239,248,512,867,582,706,457,437,274,689,468,470,669,762,397,468},
        {232,248,512,866,585,703,457,436,268,688,464,472,663,755,398,464},
        {226,248,512,860,587,701,457,430,260,685,463,473,656,749,398,463},
        {220,248,512,851,589,699,457,421,250,680,463,474,650,743,398,463},
        {215,248,512,837,591,698,457,407,240,673,463,475,645,738,397,463},
        {210,248,512,826,592,697,457,396,240,659,463,475,641,736,396,463},
        {207,248,512,815,592,696,457,385,246,641,463,475,637,734,393,463},
        {204,248,512,800,592,697,457,370,249,625,463,475,634,735,390,463},
        {202,248,512,784,591,698,457,354,247,613,463,475,632,737,386,463},
        {201,248,512,767,589,699,457,337,241,605,464,474,631,741,381,464},
        {200,248,512,754,587,701,457,324,238,600,468,473,631,746,375,468},
        {200,251,509,750,588,701,458,320,248,593,473,473,630,752,369,473},
        {200,264,496,750,598,690,465,320,257,589,478,478,630,758,362,478},
        {199,279,481,753,610,678,472,323,267,590,484,484,630,765,355,484},
        {196,294,465,755,626,662,480,325,265,594,491,492,626,763,353,491},
        {192,311,449,757,643,645,488,327,263,598,498,501,622,762,350,498},
        {188,328,431,760,660,628,497,329,262,601,505,509,618,761,348,505},
```

```
        {184,346,414,762,678,610,506,332,260,605,512,518,614,759,346,512},
        {180,364,396,764,696,593,515,334,259,609,519,527,610,757,344,519},
        {176,381,379,767,713,575,523,337,258,613,526,536,607,755,342,526},
        {172,398,362,770,730,559,532,339,257,616,533,544,603,753,340,533},
        {168,414,346,772,745,543,540,342,256,620,540,552,599,751,338,540},
        {164,428,331,775,760,528,547,345,255,623,546,559,594,748,337,546},
        {160,442,318,778,773,515,554,348,255,627,551,566,590,746,335,551},
        {157,442,318,785,776,512,554,355,262,627,556,567,587,750,335,556},
        {158,439,321,792,776,512,552,361,269,626,560,567,588,756,336,560},
        {164,437,323,798,776,512,551,368,275,626,561,567,594,764,339,561},
        {173,435,325,804,776,512,550,374,281,626,561,567,603,774,344,561},
        {187,433,326,809,776,512,549,379,286,627,561,567,617,784,351,561},
        {198,432,327,814,776,512,549,383,288,628,561,567,628,784,365,561},
        {209,432,328,817,776,512,549,387,290,631,561,567,639,778,383,561},
        {224,432,327,820,776,512,549,390,289,634,561,567,654,775,399,561},
        {240,433,326,822,776,512,549,392,287,638,561,567,670,777,411,561},
        {257,435,325,823,776,512,550,393,283,643,560,567,687,783,419,560},
        {270,437,323,824,776,512,551,393,278,649,556,567,700,786,424,556},
        {274,436,323,824,773,515,551,394,272,655,551,566,704,776,431,551},
        {274,426,334,824,760,528,546,394,266,662,546,559,704,767,435,546},
        {271,414,346,825,745,543,540,394,259,669,540,552,701,757,434,540}
};

//각 프레임의 모터 이동 시간
prog_int16_t walk_16_1_PlayTimeData[48] = {
        70,70,70,70,70,70,70,70,70,70,
        70,70,70,70,70,70,70,70,70,70,
        70,70,70,70,70,70,70,70,70,70,
        70,70,70,70,70,70,70,70,70,70,
        70,70,70,70,70,70,70,70
};

//각 프레임의 다음 프레임까지의 딜레이
prog_int16_t walk_16_1_DelayData[48] = {
        -14,-14,-14,-14,-14,-14,-14,-14,-14,-14,
        -14,-14,-14,-14,-14,-14,-14,-14,-14,-14,
        -14,-14,-14,-14,-14,-14,-14,-14,-14,-14,
        -14,-14,-14,-14,-14,-14,-14,-14,-14,-14,
        -14,-14,-14,-14,-14,-14,-14,-14
};

#endif /* WALK_16_1_H_ */
```

MotionPlay.c

```
//CPU 클럭 설정
#define F_CPU 16000000

//ATmega128의 레지스터 등이 정의되어 있음
#include <avr/io.h>
//interrupt를 사용하는 데 필요한 정보들이 정의되어 있음
#include <avr/interrupt.h>
//프로그램 메모리에 데이터를 쓰고 읽어오는 기능이 정의되어 있음
#include <avr/pgmspace.h>

//herkulex 라이브러리의 헤더 파일을 포함
#include <herkulex.h>
#include <TIMER0.h>

//사용할 모션의 헤더 파일을 포함
#include "motion/00.walk_16_1.h"
#include "motion/06.walk_16_0.h"
#include "motion/07.walk_16_2.h"
#include "motion/15.box_16.h"

/**
* Timer0의 전역변수 gucTimerTick을 사용한 딜레이 함수다.
* gucTimerTick이 unsigned char이기 때문에
* 255 이상의 값을 넣을 수가 없기 때문에 사용한다.
*
* @author       Dongbu Robot
* @param        iDelay: 기다릴 시간(ms)
* @return       없음
* @version      1.0.0.0
*/
void delay_ms(int iDelay){
        //iDelay가 0보다 크면 반복
        while(iDelay>0){
                //iDelay가 500ms 보다 큰 경우
                if(iDelay>500){
                        //gucTimerTick에 250을 넣고 iDelay를 500 뺀다.
                        gucTimerTick=250;
                        iDelay -= 500;
                        //Timer0에 의해 gucTimerTick이 0이 될 때 까지 대기
                        while(gucTimerTick>0);
                }
                //iDelay가 500ms 이하인 경우
                else{
                        //gucTimerTick에 iDelay를 2로 나누어 넣고 iDelay를 0으로 만든다.
                        gucTimerTick=iDelay/2;
                        iDelay = 0;
                        //Timer0에 의해 gucTimerTick이 0이 될 때 까지 대기
                        while(gucTimerTick>0);
                }
        }
}
```

```
/**
* 모션을 실행하기 위한 함수다.
* 모션 헤더 파일에 선언된 여러 정보 프로그램 메모리로부터 읽어와서
* 알맞은 S_JOG 패킷을 구성해서 보내어 모션을 실행한다.
*
* @author       Dongbu Robot
* @param        FrameCnt: 실행할 모션 프레임의 수
*               MotorCnt: 실행할 모션의 축 수
*               MotorData[FrameCnt][MotorCnt]: 각 프레임의 모터 목표 위치 값
*               PlayTimeData[FrameCnt]: 각 프레임의 모터 이동 시간 값
*               DelayData[FrameCnt]: 각 프레임의 딜레이 시간 값
* @return       없음
* @version      1.0.0.0
*/
void PlayMotion(        prog_int16_t FrameCnt, prog_int16_t MotorCnt,
                        prog_int16_t MotorData[FrameCnt][MotorCnt],
                        prog_int16_t PlayTimeData[FrameCnt],
                        prog_int16_t DelayData[FrameCnt])
{
        //변수 및 구조체 선언
        unsigned char i;
        unsigned int uiCurrentFrame;
        DrsPacket stSendPacket;

        //첫 프레임부터 마지막 프레임까지 반복
        for(uiCurrentFrame=0;uiCurrentFrame<FrameCnt;uiCurrentFrame++){
                //패킷 사이즈 설정. 최소 크기(7) + (모터 수 * 모터 당 데이터 크기(4)) + 이동시간(1)
                stSendPacket.ucPacketSize = MIN_PACKET_SIZE + CMD_S_JOG_STRUCT_SIZE * MotorCnt + 1;
                //여러 서보에게 주는 명령이므로 0xFE로 설정
                stSendPacket.ucChipID = BROADCAST_ID;
                //S_JOG 명령으로 설정
                stSendPacket.ucCmd = CMD_S_JOG;
                //현재 프레임에 해당하는 PlayTimeData를 읽어와서, 11.2로 나누고 반올림해서 대입
                stSendPacket.unData.stSJogData.ucPlayTime =
                                (pgm_read_word(PlayTimeData+uiCurrentFrame)*10+56)/112;
                //첫 모터부터 마지막 모터까지 반복
                for(i=0;i<MotorCnt;i++){
                        //현재 프레임과 모터에 해당하는 MotorData를 읽어와서 대입
                        stSendPacket.unData.stSJogData.stSJog[i].stJog.uiValue =
                                pgm_read_word(*(MotorData+uiCurrentFrame)+i);
                        stSendPacket.unData.stSJogData.stSJog[i].stJog.reserved = 0;
                        stSendPacket.unData.stSJogData.stSJog[i].stSet.ucStopFlag = 0;
                        stSendPacket.unData.stSJogData.stSJog[i].stSet.ucMode = 0;
                        stSendPacket.unData.stSJogData.stSJog[i].stSet.ucLedGreen = 0;
                        stSendPacket.unData.stSJogData.stSJog[i].stSet.ucLedBlue = 0;
                        stSendPacket.unData.stSJogData.stSJog[i].stSet.ucLedRed = 0;
                        stSendPacket.unData.stSJogData.stSJog[i].stSet.ucJogInvalid = 0;
                        stSendPacket.unData.stSJogData.stSJog[i].stSet.reserved = 0;
                        //모션 데이터는 모터 ID 0~15번 순으로 되어 있으므로 i를 대입
                        stSendPacket.unData.stSJogData.stSJog[i].ucId = i;
                }
```

```
                //구성한 패킷 보내기
                hklx_SendPacket(stSendPacket);

                //현재 프레임에 해당하는 PlayTimeData+DelayData만큼 대기 후 다음 프레임 실행
                delay_ms(pgm_read_word(PlayTimeData+uiCurrentFrame)+
                                pgm_read_word(DelayData+uiCurrentFrame));
        }

        return;
}

int main(void)
{
        //보낼 패킷과 받을 패킷이 저장될 패킷 구조체 선언
        DrsPacket stSendPacket;

        //포트 C의 입출력 방향 설정
        DDRC = 0b01111111;
        //포트 C의 출력 값 설정
        PORTC = 0b01111111;

        //전체 인터럽트를 비활성화
        cli();

        //HerkuleX를 사용하기 위해 초기화
        hklx_Init(115200);

        //전체 인터럽트를 활성화
        sei();

        //Torque Control에 0x60을 써서 토크를 거는 패킷 구성
        stSendPacket.ucPacketSize = MIN_PACKET_SIZE + 3;
        stSendPacket.ucChipID = BROADCAST_ID;
        stSendPacket.ucCmd = CMD_RAM_WRITE;
        stSendPacket.unData.stRWData.ucAddress = 52;
        stSendPacket.unData.stRWData.ucLen = 1;
        stSendPacket.unData.stRWData.ucData[0] = 0x60;

        //패킷 보내기
        hklx_SendPacket(stSendPacket);

        while(1)
        {
                //걷기 모션 실행(전진 시작 - 전진 * 3 - 전진 끝)
                PlayMotion(walk_16_0_FrameCnt, walk_16_0_MotorCnt,
                    walk_16_0_MotorData, walk_16_0_PlayTimeData, walk_16_0_DelayData);
                PlayMotion(walk_16_1_FrameCnt, walk_16_1_MotorCnt,
                    walk_16_1_MotorData, walk_16_1_PlayTimeData, walk_16_1_DelayData);
                PlayMotion(walk_16_1_FrameCnt, walk_16_1_MotorCnt,
                    walk_16_1_MotorData, walk_16_1_PlayTimeData, walk_16_1_DelayData);
                PlayMotion(walk_16_1_FrameCnt, walk_16_1_MotorCnt,
                    walk_16_1_MotorData, walk_16_1_PlayTimeData, walk_16_1_DelayData);
                PlayMotion(walk_16_2_FrameCnt, walk_16_2_MotorCnt,
                    walk_16_2_MotorData, walk_16_2_PlayTimeData, walk_16_2_DelayData);

                //1초 간 멈춤
                delay_ms(1000);
```

```
                //복싱 모션 실행
                PlayMotion(box_16_FrameCnt, box_16_MotorCnt,
                        box_16_MotorData, box_16_PlayTimeData, box_16_DelayData);

                //1초 간 멈춤
                delay_ms(1000);
        }

        //함수의 형태에 맞게 정수값 반환
        return 1;
}
```

예제설명

MotionPlay.c에서 #include문을 사용해 포함시켜주는 헤더 파일 중 libherkulex.a와 관련된 TIMER0.h와 herkulex.h에 대한 설명은 Chapter 8. HerkuleX 제어하기를 참조하세요. #include 문을 사용해 포함시킨 00.Walk_16_1.h 등의 모션 파일은 모두 형식이 동일하므로 00.Walk_16_1.h만 설명하도록 하겠습니다.

00.Walk_16_1.h

```
#ifndef WALK_16_1_H_
#define WALK_16_1_H_

...

#endif /* WALK_16_1_H_ */
```

00.Walk_16_1.h를 비롯한 모션 헤더 파일들은 위와 같이 #ifndef문과 #endif로 전체가 둘러싸여 있습니다. 일반적인 c 언어의 헤더파일과 같이 기존에 WALK_16_1_H_가 #define을 사용해서 선언되었는지 확인하고, 선언되지 않았으면 WALK_16_1_H_를 선언하고 헤더파일의 내용을 컴파일 합니다. 이렇게 하는 것은 같은 모션 헤더 파일이 여러 번 포함되어 컴파일 되는 것을 막기 위한 것입니다.

```
prog_int16_t walk_16_1_FrameCnt = 48;
```

모션 파일에서 로봇의 한 자세, 또는 한 순간을 나타내는 것을 프레임이라고 합니다. 프레임을 순차적으로 시간에 맞추어 재생을 하면 하나의 모션이 실행되는 것입니다. 현재 00.Walk_16_1 모션은 48개의 프레임으로 이루어진 모션입니다.
변수 앞에 붙은 prog_int16_t는 unsigned char, unsigned int 등과 같이 변수 타입을 나타내는 것입니다. prog_int16_t는 특수한 타입으로, 변수를 RAM 영역이 아닌 프로그램 메모리, 즉 flash 영역에 선언하겠다는 뜻입니다. 모션 헤더 파일에서 선언하는 모션들은 데이터의 양이 방대하기 때문에 모션 데이터를 모두 일반 변수로 선언해서 RAM에다가 저장해놓으면, 4kByte 밖에 되지 않는 ATmega128의 RAM 용량이 부족하게 됩니다. 그래서 모션 데이터를 프로그램 메모리에 저장해놓고 필요할 때 읽어다가 쓰는 것입니다.

한편 prog_int16_t 등의 타입을 사용해 변수를 프로그램 메모리에 저장해 놓으면, 프로그램 실행 중에 변수 값을 바꿀 수가 없으며 처음 선언한 값으로 그대로 유지됩니다. 프로그램 메모리는 실행되는 프로그램이 저장되는 곳이기 때문에, 실행 중에 프로그램 메모리의 값을 바꿀 수 있다면 프로그램 자체가 손상되어 오작동을 일으킬 수 있기 때문입니다. 모션 데이터의 경우 정해져 있는 숫자들이기 때문에 변할 염려가 없고, 따라서 프로그램 메모리 영역에 저장하는 것이 적합합니다.
prog_int16_t 등의 타입은 〈avr/pgmspace.h〉 파일이 포함되어 있을 때만 사용 가능합니다.

```
prog_int16_t walk_16_1_MotorCnt = 16;
```

모션을 실행하려면 현재 모션이 몇 개의 모터로 구성된 로봇의 것인지 알아야 합니다. 00.Walk_16_1 모션은 16축으로 이루어진 모션입니다.

```
prog_int16_t walk_16_1_MotorData[48][16] = {
        {269,398,362,828,730,559,532,398,261,671,533,544,699,759,430,533},
…
        {271,414,346,825,745,543,540,394,259,669,540,552,701,757,434,540}
};
```

모션의 각 프레임에서 각각의 모터가 어떤 위치에 있는지, 모터의 위치 값을 저장하는 2차원 배열입니다. 48개의 프레임으로 구성된 모션이고 모터가 16개가 있으므로, [48][16]의 2차원 배열이 필요합니다.

```
prog_int16_t walk_16_1_PlayTimeData[48] = {
        70,70,70,70,70,70,70,70,70,70,
…
        70,70,70,70,70,70,70,70
};
```

모션 파일의 각 프레임에 해당하는 S_JOG 패킷을 HerkuleX에 보낼 때, 얼마만큼의 시간 동안 해당 자세로 움직여야 하는 지를 나타내는 데이터입니다. 이 데이터는 밀리초(ms) 단위로, S_JOG 패킷에 들어가려면 11.2ms 단위로 바뀌어 들어가야 합니다. 00.Walk_16_1 모션의 실행 시간은 모두 70ms이므로, 11.2ms 단위로 환산하면 70/11.2 = 6.25, 즉 반올림하여 6이라는 데이터가 S_JOG 패킷에 들어가야 합니다.

```
prog_int16_t walk_16_1_DelayData[48] = {
        -14,-14,-14,-14,-14,-14,-14,-14,-14,-14,
…
        -14,-14,-14,-14,-14,-14,-14,-14
};
```

모션 파일의 실행 시간이 끝난 후, 다음 프레임을 얼마나 더 기다렸다가 보낼지 결정하는 것이 딜레이입니다. 이 데이터도 역시 밀리초(ms) 단위입니다. 가령 00.Walk_16_1 모션에서 실행 시간이 70ms일 때 딜레이가 0ms라면, 프레임을 실행하는 S_JOG 패킷을 보내고 나서 70ms가 지나면 바로 다음 프레임의 패킷을 보냅니다. 가령 딜레이가 14ms라면, 70ms+14ms = 84ms가 지나고 나서 다음 패킷을 보냅니다. 로봇의 움직임이 끝나고 나서도 14ms를 더 기다렸다가 보내는 셈입니다. 한편 이 모션처럼 딜레이가 –14ms라면, 70ms–14ms = 56ms가 지나고 나서 다음 패킷을 보내므로 로봇의 움직임이 끝나기도 전에 다음 프레임 패킷을 보냅니다. 이렇게 마이너스(–) 딜레이를 주었을 때의 장점은 로봇의 동작이 딱딱 끊어지는 것이 아니라 부드럽게 이어지는 모션을 만들 수 있다는 것입니다.

MotionPlay.c

```
#define F_CPU 16000000
```

#define 문은 코드 전처리 명령 중 하나로, 앞으로 F_CPU라는 값이 나오면 16000000으로 대체한다는 뜻입니다. DRC의 ATmega128은 16MHz의 클럭 속도로 동작합니다. 클럭 속도에 비례해서 명령 처리 속도가 달라지므로, 이 명령을 통해서 이 프로그램이 동작할 속도를 명시해 주어야 delay 함수 등이 올바르게 동작합니다.

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/pgmspace.h>
```

avr/io.h 헤더 파일에는 DDRX, PORTX 레지스터 등 AVR 제어를 위해 필요한 레지스터 등이 정의되어 있고, avr/interrupt.h 헤더 파일에는 인터럽트를 사용하기 위한 함수들이 정의되어 있습니다. avr/pgmspace.h 헤더 파일에는 프로그램 메모리에 데이터를 쓰고 읽어오는 기능이 정의되어 있습니다. 앞서 설명한 prog_int16_t 등의 변수 타입, 뒤에 나올 pgm_read_word() 등의 프로그램 메모리를 읽어오는 함수 등이 선언되어 있으므로 포함시켜 줘야 합니다.

```
#include <herkulex.h>
#include <TIMER0.h>
```

herkulex.h 헤더 파일에는 패킷을 구성하고 보내기 위한 선언들과 구조체, 그리고 함수가 정의되어 있으므로 꼭 포함해야 합니다. 그리고 gucTimerTick을 사용해 delay를 구현하기 위하여, TIMER0.h도 포함시킵니다.

```
#include "motion/00.walk_16_1.h"
#include "motion/06.walk_16_0.h"
#include "motion/07.walk_16_2.h"
#include "motion/15.box_16.h"
```

현재 예제에서 실행할 모션의 데이터가 담긴 헤더 파일입니다. 앞서 설명한 00.walk_16_1.h 처럼 모션의 프레임 수, 축 수, 모터 위치 데이터, 실행시간 데이터, 딜레이 데이터가 선언되어 있습니다.

```
void delay_ms(int iDelay){
        while(iDelay>0){
            if(iDelay>500){
```

```
                gucTimerTick=250;
                iDelay -= 500;
                while(gucTimerTick>0);
        }
        else{
                gucTimerTick=iDelay/2;
                iDelay = 0;
                while(gucTimerTick>0);
        }
    }
}
```

TIMER0을 이용한 딜레이 함수입니다. 매개변수로는 iDelayTick이 있는데, 얼마만큼의 시간동안 대기할 것인지 밀리초(ms) 단위로 나타낸 것입니다. gucTimerTick은 TIMER0.h에서 선언된 전역변수로, 평소에는 0으로 유지되다가 0이 아닌 값을 대입하면 libherkulex.a에 정의된 TIMER0 오버플로우 인터럽트 서비스 루틴에 의해서 2밀리초마다 1씩 줄어듭니다. 그래서 가령 10밀리초를 기다리고 싶으면 gucTimerTick에 5라는 값을 넣고 gucTimerTick이 0이 될 때까지 기다리면 됩니다. 이것을 이용해서 delay_ms 함수를 구현합니다.

gucTimerTick은 unsigned char, 즉 부호 없는 1바이트 변수입니다. 그래서 0~255 까지 밖에 값을 가질 수 없습니다. gucTimerTick의 255는 510밀리초에 해당하는 시간입니다. gucTimerTick에 한번의 대입만으로는 약 0.5초 밖에 딜레이를 구현할 수 없는 것입니다. 그래서 이 함수에서는 딜레이 값 iDelay를 500밀리초 씩 끊어서 gucTimerTick에 대입합니다.

iDelay가 500보다 크면, gucTimerTick에는 250을 대입하고 iDelay를 500 뺀 값으로 바꿉니다. 그리고 while(gucTimerTick>0); 을 사용해 gucTimerTick이 0보다 클 동안 무한 반복을 하면서 대기합니다. gucTimerTick이 0이 되면 while문에서 빠져나옵니다. iDelay가 500보다 작거나 같으면, gucTimerTick에는 iDelay를 2로 나누어 대입하고 iDelay를 0으로 만듭니다. 그리고 마찬가지로 while문을 사용해 gucTimerTick이 0이 될 때 까지 기다립니다. 이 과정을 while(iDelay>0)으로 감싸서 iDelay가 0보다 클 동안 반복하므로, iDelay가 0이 되었을 때 while문이 끝나고 delay_ms 함수가 끝나게 됩니다.

```
void PlayMotion(        prog_int16_t FrameCnt, prog_int16_t MotorCnt,
                prog_int16_t MotorData[FrameCnt][MotorCnt],
                prog_int16_t PlayTimeData[FrameCnt],
                prog_int16_t DelayData[FrameCnt])
```

다음 함수는 PlayMotion 함수입니다. 매개변수로는 모션 헤더 파일에 선언된 변수들이 그대로 들어갑니다. 차례로 프레임 수, 축 수, 모터 데이터, 실행시간 데이터, 딜레이 데이터입니다. 여기서 MotorData, PlayTimeData, DelayData는 배열의 이름이고, 배열의 이름은 그 배열의 첫 요소에 대한 포인터임을 유념하세요. 프로그램 메모리에 선언된 배열 데이터는 pgm_read_word() 등의 특별한 함수를 사용해서 데이터를 읽어와야 합니다.

```
unsigned char i;
unsigned int uiCurrentFrame;
DrsPacket stSendPacket;
```

변수를 선언합니다. i와 uiCurrentFrame은 for문에서 각각 모터 수, 프레임 수만큼 반복하기 위해서 선언된 변수입니다. 그리고 stSendPacket은 보낼 S_JOG 패킷이 저장될 공간입니다.

```
for(uiCurrentFrame=0;uiCurrentFrame<FrameCnt;uiCurrentFrame++){
                                    …
}
```

프레임 수만큼 반복되는 for문입니다. for문의 내부에서는 프레임을 하나씩 진행하면서 각 프레임에 해당하는 모터 값과 실행 시간을 읽어서 S_JOG 패킷을 구성해서 모터에 보내고, 실행 시간과 딜레이를 더한 만큼을 delay_ms 함수에 넣어서 그만큼의 시간을 기다립니다. delay_ms 함수가 끝나면 다음 프레임으로 넘어갑니다.

```
stSendPacket.ucPacketSize = MIN_PACKET_SIZE + CMD_S_JOG_STRUCT_SIZE * MotorCnt + 1;
stSendPacket.ucChipID = BROADCAST_ID;
stSendPacket.ucCmd = CMD_S_JOG;
```

현재 프레임에서 보내야 할 패킷 중에서 패킷 크기, 아이디, 명령을 설정하는 부분입니다. 패킷 사이즈는 S_JOG의 패킷 구조에 따라서, 기본 7바이트(MIN_PACKET_SIZE)에 모터 당 4바이트*모터 수가 더해지고, 실행 시간 1바이트가 더해집니다. 그래서 MotorCnt가 16일 경우에는 7+4*16+1이 되어서 총 72바이트 크기의 패킷을 모터에 보냅니다.

```
stSendPacket.unData.stSJogData.ucPlayTime = (pgm_read_word(PlayTimeData+uiCurrentFrame)*10+56)/112;
```

S_JOG 패킷의 실행시간을 설정하는 부분입니다. 식이 복잡하지만 한 단계씩 설명하겠습니다. 우선 pgm_read_word()는 프로그램 메모리의 값을 읽는 함수이며, avr/pgmspace.h에서 선언됩니다. 매개변수에는 읽을 프로그램 메모리의 주소가 들어갑니다. PlayTimeData는 이 함수의 매개변수로 받은 배열의 이름이며, 앞서 배열의 이름은 배열의 첫 요소의 주소임을 언급했습니다. 즉 PlayTimeData는 &PlayTimeData[0]와 같습니다. 이와 같은 원리로, (PlayTimeData+i)는 &PlayTimeData[i]와 같습니다. 즉 pgm_read_word에 매개변수로 넣는 (PlayTimeData+uiCurrentFrame)은 &PlayTimeData[uiCurrentFrame]과 같은 것입니다. 따라서 pgm_read_word (PlayTimeData+uiCurrentFrame)는 PlayTimeData[uiCurrentFrame]의 값을 읽어오는 부분입니다.

이 식에서는 PlayTimeData[uiCurrentFrame]의 값을 읽어와서 10을 곱하고, 56을 더하고, 112로 나눕니다. 이것은 PlayTimeData[uiCurrentFrame]를 11.2로 나눈 후 반올림하는 부분을 오로지 정수 계산으로 나타낸 것입니다.

PlayTimeData에는 실행 시간이 밀리초 단위로 저장되어 있기 때문에, 실제 S_JOG 패킷에 넣을 때에는 11.2밀리초 단위로 나누어서 보내야 합니다. 이 때 무턱대고 11.2로 나누는 방법도 있지만, 이 경우 실수 연산이 되기 때문에 AVR 프로세서의 특성상 연산 속도가 정수 연산보다 수십 배 느려집니다. 그래서 PlayTimeData[uiCurrentFrame]/11.2 대신에, 11.2 = 112/10 임을 이용해서 PlayTimeData[uiCurrentFrame]*10/112를 하면 두 번의 정수 연산 만으로도 11.2로 나누는 실수 연산을 구현할 수 있게 됩니다.

하지만 이렇게 하는 경우에는 나눗셈의 결과 값 중 소수 부분이 자동적으로 없어지기 때문에, PlayTimeData를 읽어와서 11.2로 나눈 뒤 소수점 이하를 모두 버림한 것과 같게 됩니다. 즉 11.2로 나눈 실행 시간 값이 6.0~6.9999… 의 범위에 있으면 모두 6이 되는 것입니다. 미세한 오차지만, 모션 실행 시 로봇이 의도치 않게 작동할 수도 있습니다. 이를 해결하기 위해 나누기 전에 56을 더한 후 112로 나누면, 원래 실행 시간 값 6.0~6.9999…의 범위가 계산 결과 값이 6.5~7.4999…가 되어 원래 실행 시간 기준 6.0~6.4999…은 6, 6.5~6.9999…은 7이라는 결과가 나오므로, 반올림한 것과 같은 효과를 얻게 됩니다.

요약하면 위 코드는 프로그램 메모리로부터 PlayTimeData[uiCurrentFrame]의 값을 읽어서, 11.2밀리초로 나눈 후 반올림하여 S_JOG 패킷의 실행시간에 대입하는 부분입니다.

```
for(i=0;i<MotorCnt;i++){
        stSendPacket.unData.stSJogData.stSJog[i].stJog.uiValue =
                        pgm_read_word(*(MotorData+uiCurrentFrame)+i);
        stSendPacket.unData.stSJogData.stSJog[i].stJog.reserved = 0;
        stSendPacket.unData.stSJogData.stSJog[i].stSet.ucStopFlag = 0;
        stSendPacket.unData.stSJogData.stSJog[i].stSet.ucMode = 0;
        stSendPacket.unData.stSJogData.stSJog[i].stSet.ucLedGreen = 0;
        stSendPacket.unData.stSJogData.stSJog[i].stSet.ucLedBlue = 0;
        stSendPacket.unData.stSJogData.stSJog[i].stSet.ucLedRed = 0;
        stSendPacket.unData.stSJogData.stSJog[i].stSet.ucJogInvalid = 0;
        stSendPacket.unData.stSJogData.stSJog[i].stSet.reserved = 0;
        stSendPacket.unData.stSJogData.stSJog[i].ucId = i;
}
```

i를 0부터 MotorCnt-1까지 증가시켜가면서, S_JOG의 모터 데이터 부분을 채워나갑니다. 현재 우리가 필요로 하는 모터 값은 MotorData[uiCurrentFrame][i]입니다. pgm_read_word 에서는 매개변수로 포인터를 필요로 하므로, 우리는 &MotorData[uiCurrentFrame][i]를 넣어주어야 합니다. &MotorData[uiCurrentFrame][i]는 곧 (*(MotorData+uiCurrentFrame)+i)와 같습니다. 왜 그런지 단계별로 설명하겠습니다.

MotorData[FrameCnt][MotorCnt]라는 2차원 배열은 다르게 생각하면 MotorCnt 개의 정수로 이루어진 배열이 FrameCnt 개만큼 있는 배열이므로, MotorData는 포인터의 포인터라고 할 수 있습니다. 앞서서 (PlayTimeData+uiCurrentFrame)이 &PlayTimeData[uiCurrentFrame]와 같았듯이,

(MotorData+uiCurrentFrame) = &MotorData[uiCurrentFrame]

와 같고, 양쪽에 *를 붙이면

***(MotorData+uiCurrentFrame)이 MotorData[uiCurrentFrame]**

이 됩니다. MotorData[uiCurrentFrame]은 다르게 보면 ucMotorCnt 개의 int로 이루어진 배열의 시작을 나타내는 포인터이므로, 이 값에 i를 더하면, 마찬가지로 (PlayTimeData+uiCurrentFrame)이 &PlayTimeData[uiCurrentFrame]와 같았듯이

(*(MotorData+uiCurrentFrame)+i) = &MotorData[uiCurrentFrame][i]

가 됩니다. 따라서, pgm_read_word(*(MotorData+uiCurrentFrame)+i) 부분은 프레임 번호 uiCurrentFrame과 모터 번호 i에 해당하는 모터 값을 프로그램 메모리로부터 읽어와서 S_JOG 패킷의 해당하는 위치에 넣어주는 부분입니다.

그 밑의 부분은 패킷의 나머지 플래그를 설정하는 부분이며, 별로 중요하지는 않습니다. 마지막의 stSendPacket.unData.stSJogData.stSJog[i].ucId = i;는 제어할 모터의 ID를 i로 대입해주는 부분입니다.

```
hklx_SendPacket(stSendPacket);
```

hklx_SendPacket() 함수를 호출해 만든 S_JOG 패킷을 보냅니다.

```
delay_ms(pgm_read_word(PlayTimeData+uiCurrentFrame)+pgm_read_word(DelayData+uiCurrentFrame));
```

다음 프레임의 패킷을 보낼 때까지 대기하는 부분입니다. pgm_read_word(PlayTimeData+uiCurrentFrame)로 현재 프레임의 실행 시간을 다시 읽어오고, pgm_read_word(DelayData+uiCurrentFrame)로 현재 프레임의 딜레이를 읽어와 둘을 더해서 delay_ms 함수에 넘겨주어 다음 프레임의 패킷을 보내기까지 기다립니다.
지금까지의 과정을 uiCurrentFrame이 0일 때부터 FrameCnt-1일 때까지, 모든 프레임에 대해서 반복하고 나면 모션 실행이 모두 종료되고, PlayMotion 함수가 끝납니다.

이제부터는 main 함수에 대한 설명입니다.

```
DrsPacket stSendPacket;
```

메인 함수에서 사용할 변수를 선언합니다. DrsPacket 구조체 타입으로 패킷을 보낼 때 쓸 변수를 선언합니다.

```
DDRC = 0b01111111;
PORTC = 0b01111111;
```

포트 C의 0~6번 핀을 출력으로 설정하고 출력 값을 1로 만듭니다. 그러면 모든 LED가 꺼지게 됩니다.

```
cli();
hklx_Init(115200);
sei();
```

인터럽트에 대한 설정을 바꾸는 중에 인터럽트가 발생해 예상치 못한 오동작이 발생하는 것을 방지하기 위해서 우선 cli() 명령을 통해서 전체 인터럽트를 비활성화 시킵니다. 그 후 hklx_Init() 함수를 사용해서 HerkuleX를 제어하기 위한 타이머/카운터0과 USART0을 초기화합니다. 사용할 보드 레이트가 115200이므로 파라미터로 그 값을 사용합니다. 초기화가 끝난 후에는 sei() 명령을 통해서 전체 인터럽트를 활성화 합니다.

```
stSendPacket.ucPacketSize = MIN_PACKET_SIZE + 3;
stSendPacket.ucChipID = BROADCAST_ID;
stSendPacket.ucCmd = CMD_RAM_WRITE;
stSendPacket.unData.stRWData.ucAddress = 52;
stSendPacket.unData.stRWData.ucLen = 1;
stSendPacket.unData.stRWData.ucData[0] = 0x60;

hklx_SendPacket(stSendPacket);
```

모터가 스스로 움직이게 제어하기 위해서는 모터에 토크가 인가되어야 합니다. 위의 코드가 토크를 인가시키기 위한 RAM_WRITE 패킷을 구성하고, 보내는 부분입니다. 패킷의 SIZE는 기본 MIN_PACKET_SIZE(7바이트)에다가, 주소 + 길이 + 데이터(1바이트)로 총 MIN_PACKET_SIZE+3(10바이트)이며, 모든 연결된 모터에 보내야 하므로 ID는 BROADCAST_ID, 즉 254 입니다. CMD는 CMD_RAM_WRITE(0x03)이며, unData 공용체의 멤버 중 stRWData 구조체의 ucAddress, ucLen, ucData[0]의 값을 각각 52, 1, 0x60으로 만들어줍니다. 그 후에는 hklx_SendPacket() 함수를 호출해 패킷을 보냅니다.

```
PlayMotion(walk_16_0_FrameCnt, walk_16_0_MotorCnt, walk_16_0_MotorData, walk_16_0_PlayTimeData, walk_16_0_DelayData);
PlayMotion(walk_16_1_FrameCnt, walk_16_1_MotorCnt, walk_16_1_MotorData, walk_16_1_PlayTimeData, walk_16_1_DelayData);
PlayMotion(walk_16_1_FrameCnt, walk_16_1_MotorCnt, walk_16_1_MotorData, walk_16_1_PlayTimeData, walk_16_1_DelayData);
PlayMotion(walk_16_1_FrameCnt, walk_16_1_MotorCnt, walk_16_1_MotorData, walk_16_1_PlayTimeData, walk_16_1_DelayData);
PlayMotion(walk_16_2_FrameCnt, walk_16_2_MotorCnt, walk_16_2_MotorData, walk_16_2_PlayTimeData, walk_16_2_DelayData);
```

여기서부터는 while문 안의 내용입니다. while문에서는 별다른 일이 없이, 모션을 계속 실행시킵니다. 처음에 전진 시작 모션(walk_16_0)을 실행하고, 그 후 전진 모션(walk_16_1)을 세번 실행하고, 전진 종료 모션(walk_16_2)을 실행합니다. 이렇게 하면 로봇이 10발자국을 앞으로 걸어간 후 다시 멈추게 됩니다.

```
delay_ms(1000);
```

전진이 끝난 후 1초 동안 멈추어 대기합니다.

```
PlayMotion(box_16_FrameCnt, box_16_MotorCnt, box_16_MotorData, box_16_PlayTimeData, box_16_DelayData);
```

복싱 모션(box_16)을 실행합니다.

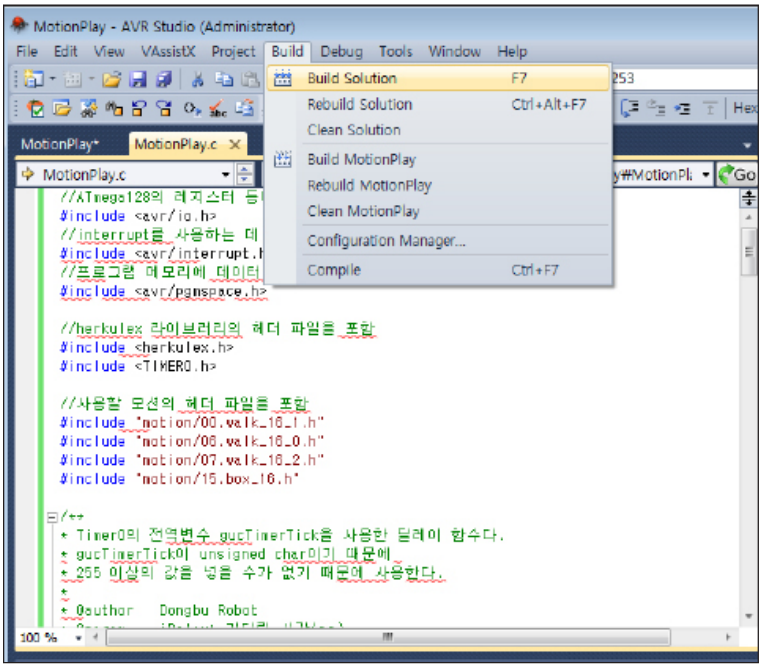
```
delay_ms(1000);
```

복싱 모션이 끝난 후 1초 동안 멈추어 대기합니다.

while문에서는 위 내용을 계속 반복하게 되어 있습니다. 로봇은 열 발자국 전진 후 복싱 모션을 계속 반복합니다.

실행

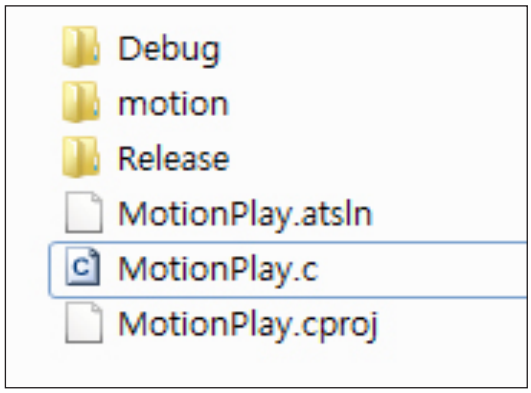
DRC를 16축 Hovis Lite에 연결, 펌웨어를 다운로드하고 실행하면 로봇이 열 발자국 전진 후 복싱 모션을 계속 반복합니다.



01 Build

Build 메뉴를 선택하여 Build Solution을 클릭하여 Build 시킵니다.

1 succeeded 0 failed 가 나오면 에러없이 컴파일 완료됐다는 것을 의미합니다.



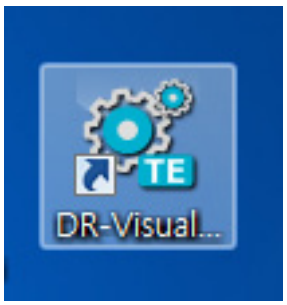
02 Debug 폴더

Build 후에 소스가 있는 폴더에 들어가면 Debug 폴더가 생성 되어있습니다. 클릭하여 폴더 안의 내용을 봅니다.

03 hex 확인

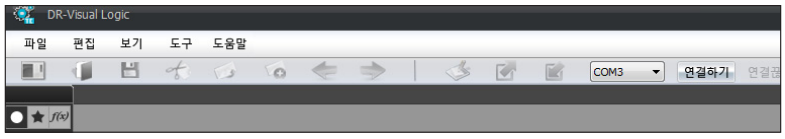
Debug 폴더 안의 hex 파일 확인합니다. Hex 파일이 제어기에 포팅시키는 파일입니다.

DR-Visual Logic 에서 펌웨어 복구를 실시합니다.



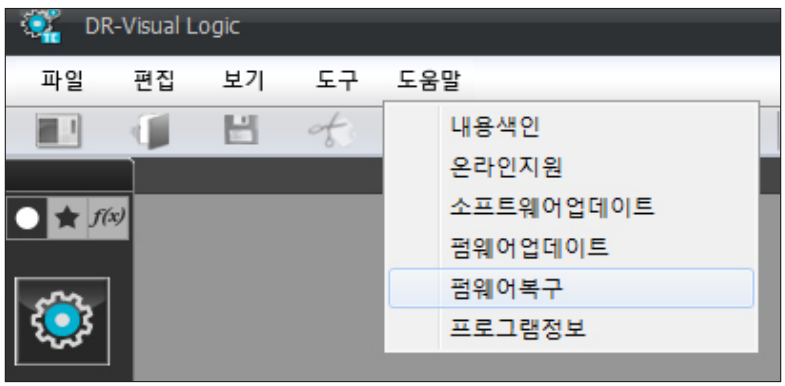
04 Visual Logic

DR-Visual Logic 을 실행시킵니다.



05 연결하기

COM 번호를 설정하고 연결하기를 클릭합니다.

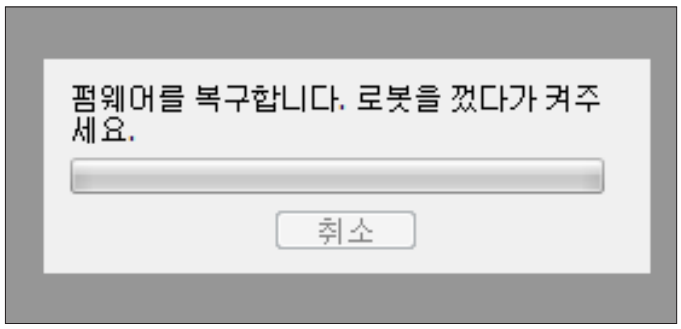


06 펌웨어 복구

도움말 메뉴에서 펌웨어복구를 클릭합니다.

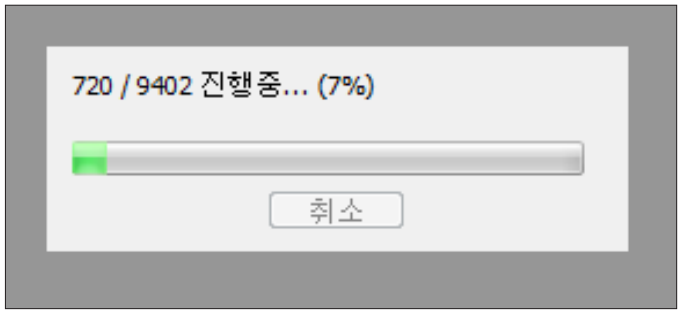
07 불러오기

Debug 폴더에서 Hex 파일을 불러옵니다



08 펌웨어복구

로봇을 껐다가 켜주세요 라는 문구가 나옵니다.
제어기의 전원버튼을 한번 껐다가 다시 켜주세요.



09 복구시작

펌웨어 복구가 시작됩니다.

10 실행

DRC를 16축 Hovis Lite에 연결, 펌웨어를 다운로드하고 실행하면 로봇이 열 발자국 전진 후 복싱 모션을 계속 반복합니다.

적외선 리모컨으로 덤프트럭 제어하기

herkulex 라이브러리, irremocon 라이브러리를 사용해 덤프트럭 로봇을 제어합니다.

H/W 원리

이번 예제에서는 HerkuleX를 제어하는 라이브러리와 적외선 리모컨을 읽어오는 라이브러리를 동시에 사용해 덤프트럭 로봇을 제어합니다. HerkuleX 제어에 관련된 내용은 Chapter 8을, 적외선 리모컨과 관련된 내용은 Chapter 12를 참조하세요. 이번 예제에서 사용하는 패킷에 사용된 명령을 요약하면 아래와 같습니다.

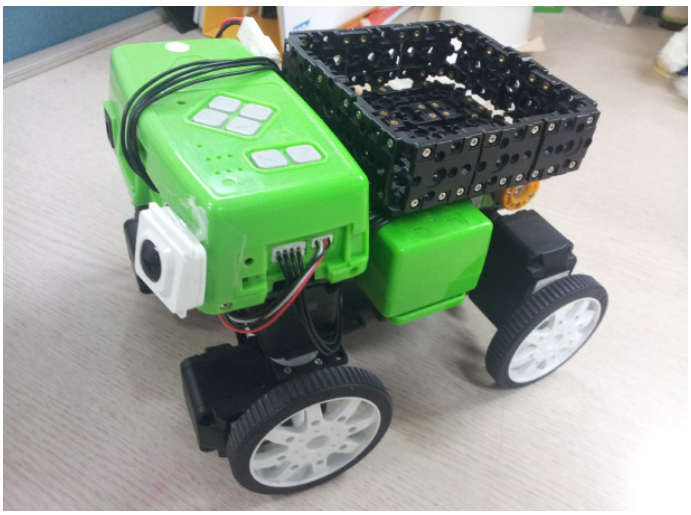
이름	CMD	설명
RAM_WRITE	0x03	HerkuleX의 휘발성 레지스터 영역에 데이터를 씁니다.
I_JOG	0x05	HerkuleX의 위치/속도를 제어하여 움직이게 합니다. 각 모터마다 이동 시간을 다르게 설정할 수 있습니다. I_JOG와 S_JOG 패킷에서 시간을 나타내기 위해서는 1바이트 크기의 데이터를 사용하는데, 이때 데이터는 1당 11.2ms를 나타냅니다. 즉 I_JOG 패킷에 50의 시간 데이터가 들어가 있다면 실제로 모터는 50*11.2=560ms 동안 목표 위치로 이동합니다.

이번 예제에서 사용된 휘발성 레지스터 영역의 항목을 요약하면 아래와 같습니다.

이름	주소	크기	설명
Torque Control	52	1	토크 인가 상태를 제어합니다. 0x00 : 자유롭게 움직이는 힘이 풀린 상태 0x40 : 힘은 풀렸으나 돌아가는 데에 저항이 있는 상태 0x60 : 토크가 인가되어 스스로 구동하는 상태

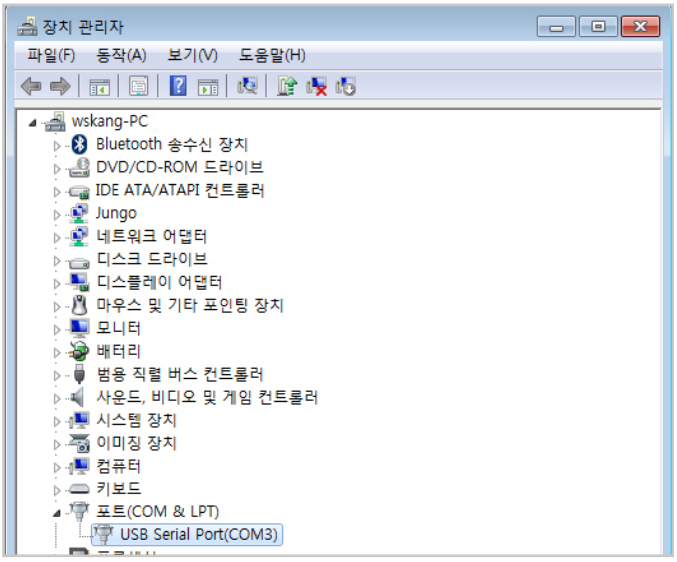
프로토콜과 레지스터 맵에 대한 더 자세한 내용은 HerkuleX 매뉴얼을 참조하세요.

하드웨어 준비하기



01 연결

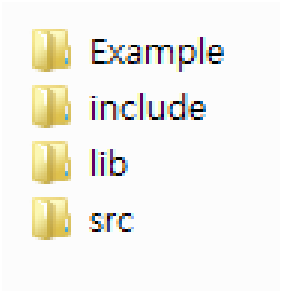
DRC 제어기 의 USB to Serial Cable 로 PC 와 DRC 를 연결합니다.
16축 휴머노이드 Hovis Lite 를 PC 와 연결합니다.



02 COM port 연결

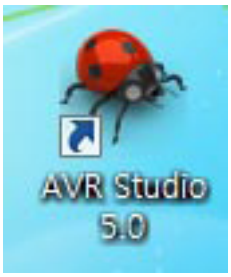
내컴퓨터 〉 속성 〉 장치관리자에서 COM 연결 확인합니다.

AVR Studio 실행 및 코딩 준비



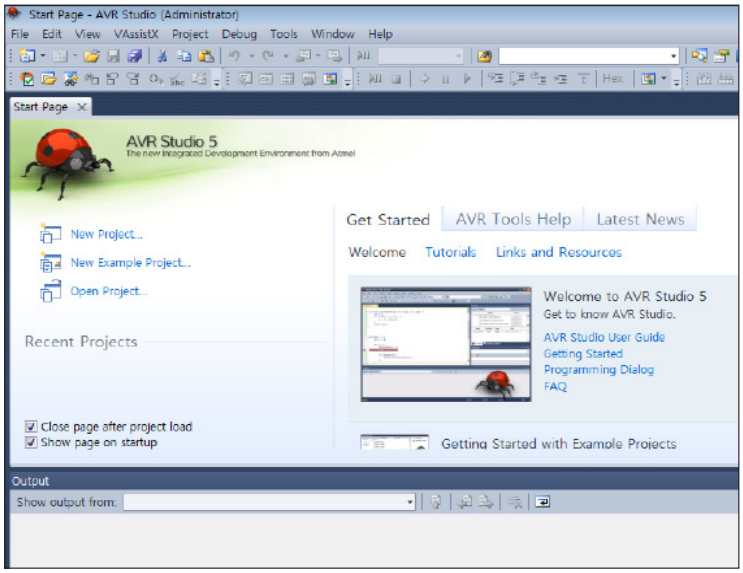
00 라이브러리

덤프트럭을 제어하기 위해서는 기본 라이브러리 파일이 필요합니다.
라이브러리가 포함된 폴더 구조입니다.



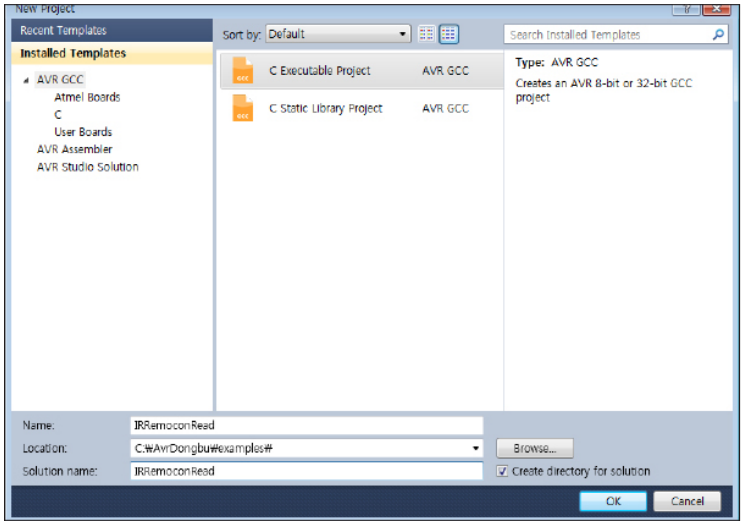
01 실행

AVR Studio 를 실행합니다.



02 New Project

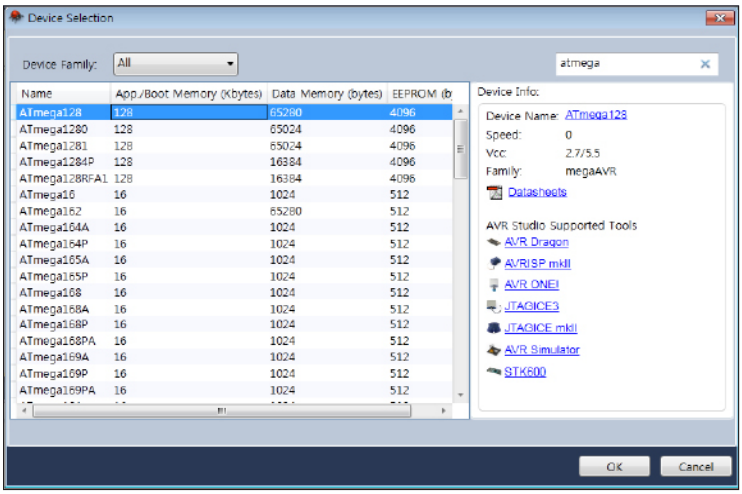
New Project 를 클릭합니다.



03 GCC 선택

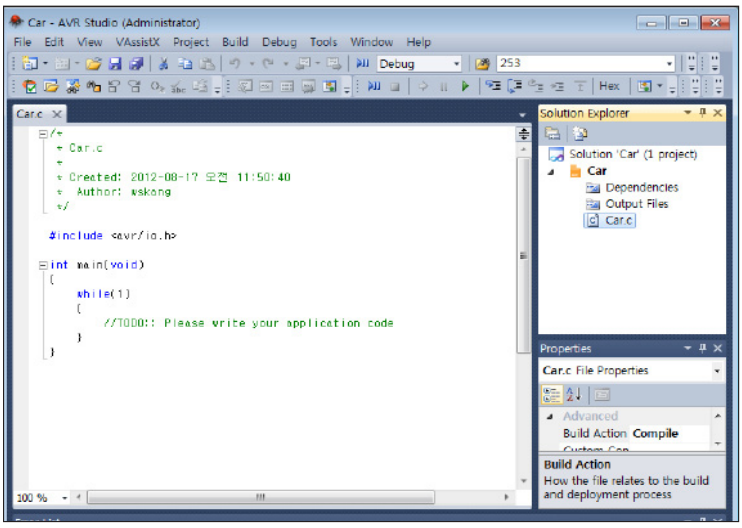
C Executable Project 를 선택합니다.

Name 을 “Car”로 입력하고, Create directory for solution 을 클릭하고, OK 버튼을 누릅니다.



04 ATmega 선택

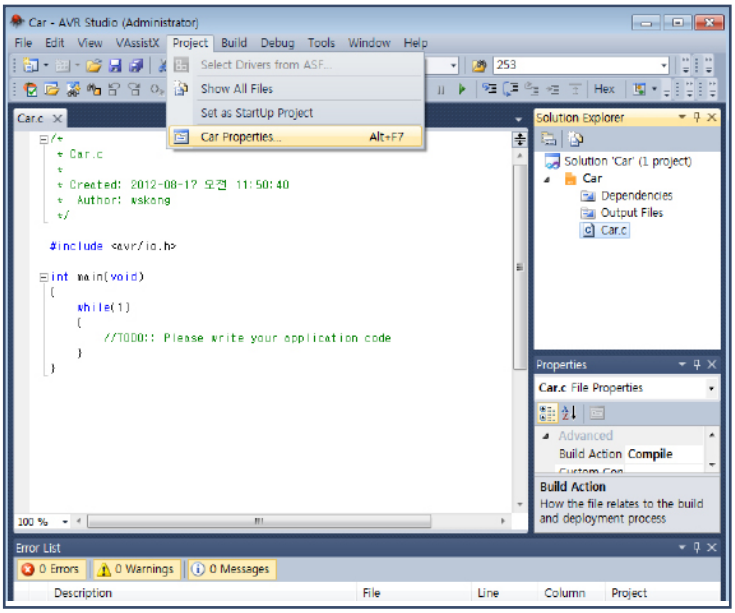
우측 상단에 atmega 를 검색하여, 리스트중에 ATmega128 을 선택합니다.



05 Editor

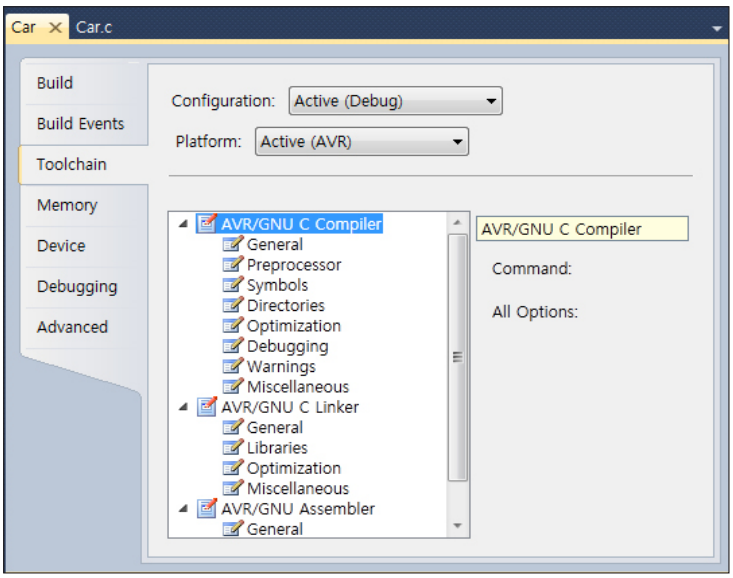
편집창에 코딩을 시작합니다.

라이브러리 링크 및 설정



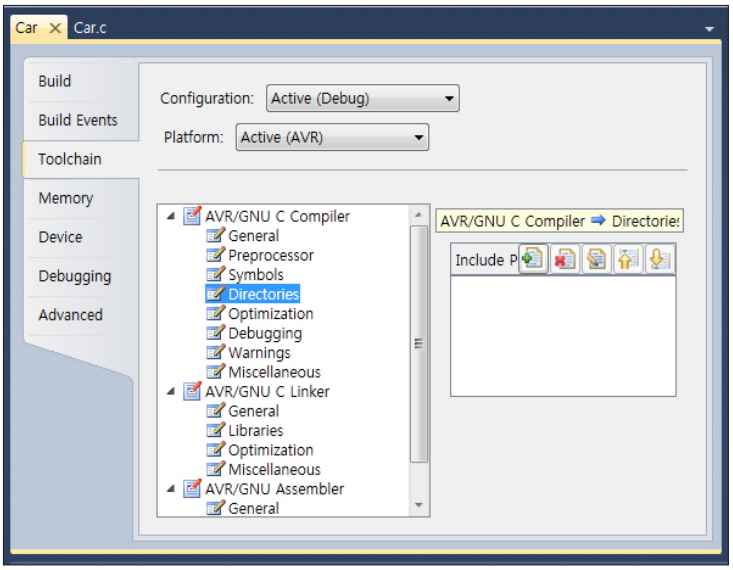
01 설정

AVR Studio 상단 메뉴중에 Project 를 클릭하면 Car (프로젝트 이름별로 다름) Property 메뉴가 생성되어 있습니다. 클릭하면 좌측과 같이 설정화면이 나옵니다.



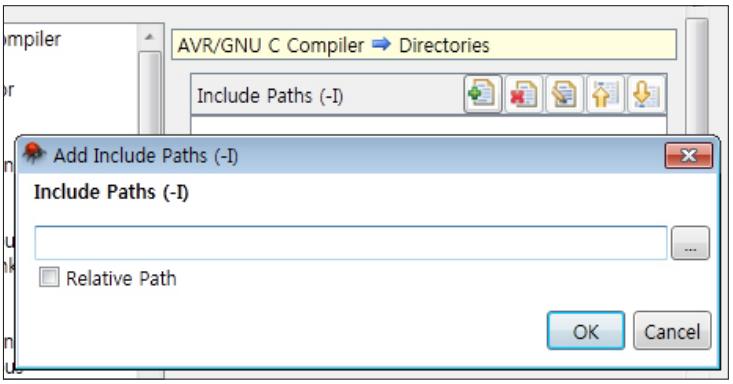
02 Toolchain

좌측 탭버튼중에 Toolchain 을 클릭하여, AVR/GNU C Compiler 을 선택합니다.



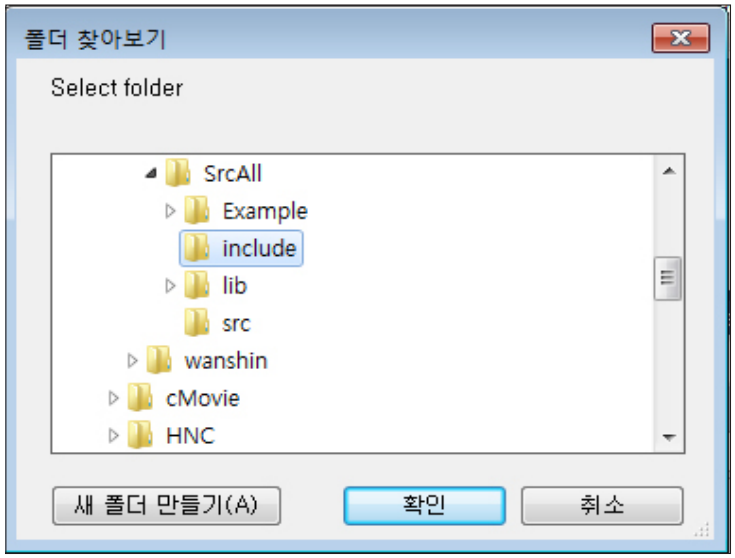
03 C Compiler

C Compiler 폴더중에 Directories 를 선택합니다.



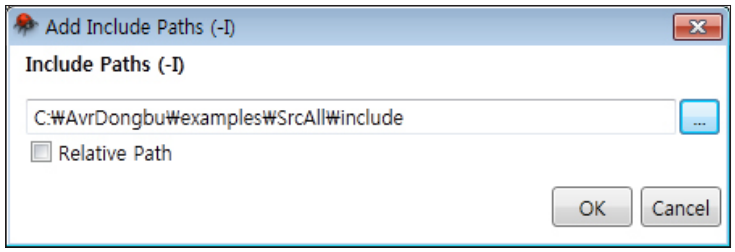
04 Path

Include Paths 오른쪽에 + 버튼을 클릭하면 팝업 입력창이 뜹니다.
Relative Path (상대경로) 체크를 해지하고,
우측 폴더 검색 버튼을 클릭합니다.



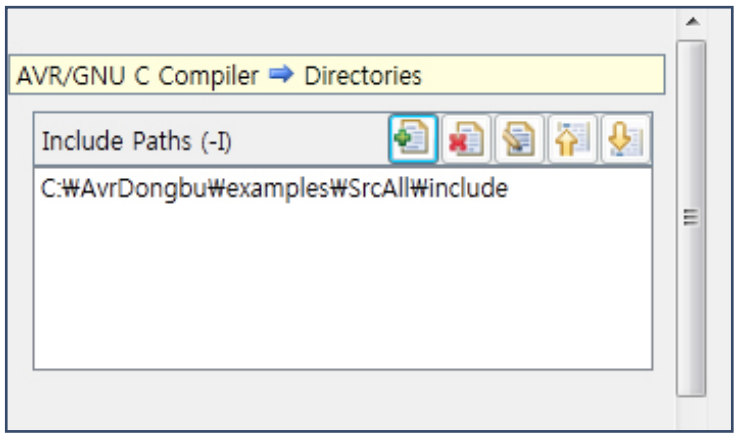
05 Include

제공 라이브러리 폴더중에 include 를 선택합니다.



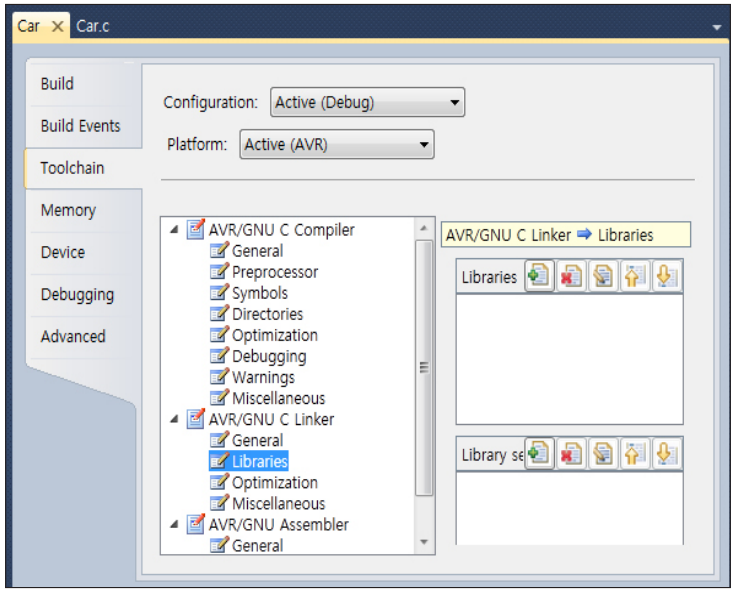
06 Include 완료

Include 폴더 위치가 지정됩니다. OK 버튼을 클릭합니다.



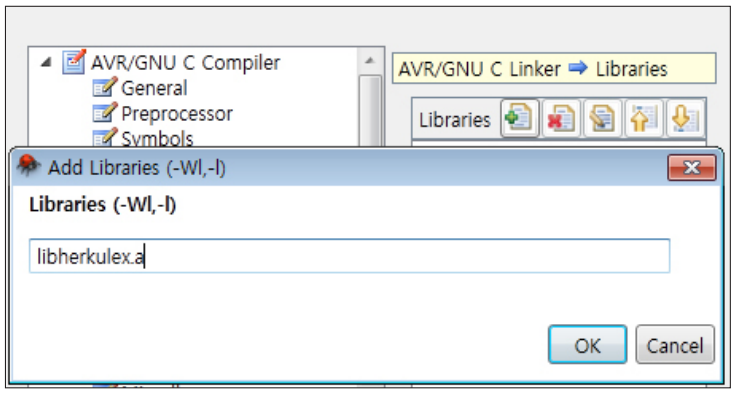
07 Path 완료

Include 경로지정이 완료되었습니다.



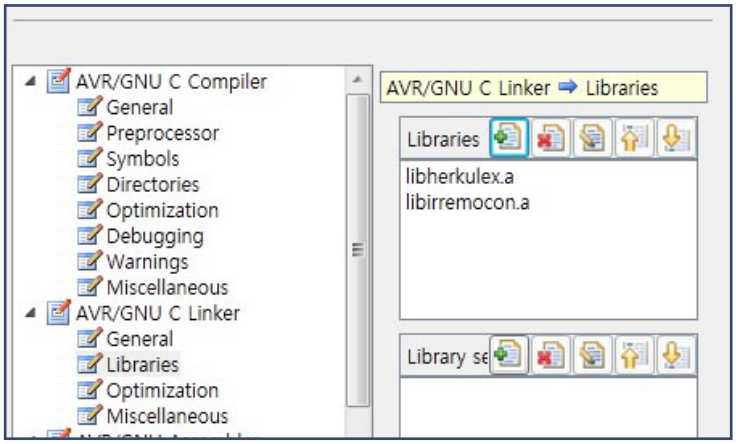
08 C Linker

C Linker 폴더중에 Libraries 를 선택합니다.



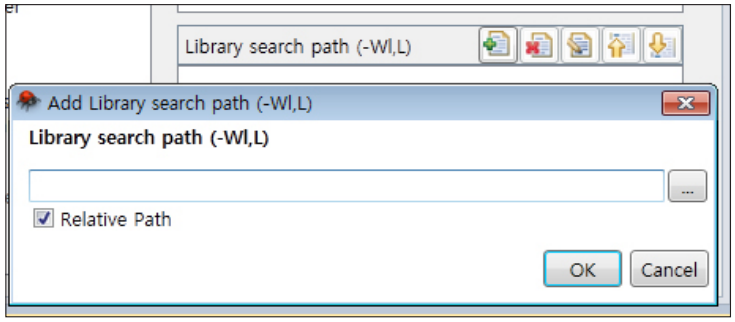
09 Libraries

Libraries 창 우측 + 를 클릭하면, Libraries 파일명 입력창이 나옵니다. libherkulex.a 를 직접 입력합니다.
OK 버튼을 클릭합니다.



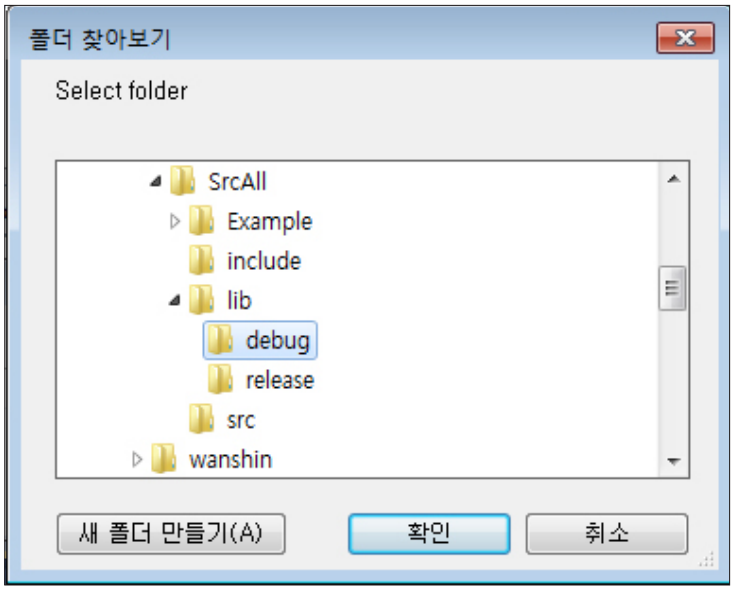
10 Search

서보모터 라이브러리 파일 libherkulex.a 링크가 완료되었습니다.
다음으로
Library search path 를 설정합니다.
우측 + 버튼을 클릭합니다.



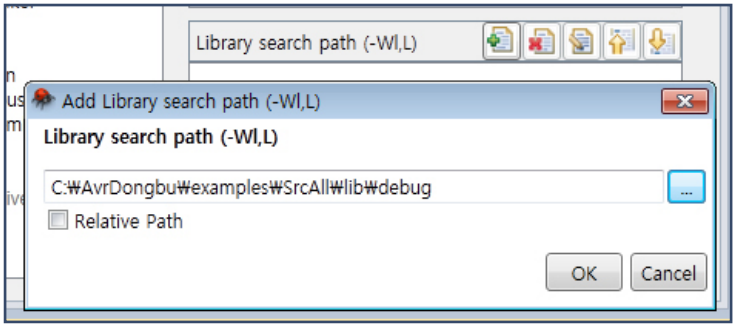
11 Path

Libray search path 팝업창 우측에 폴더 검색 버튼을 클릭합니다.



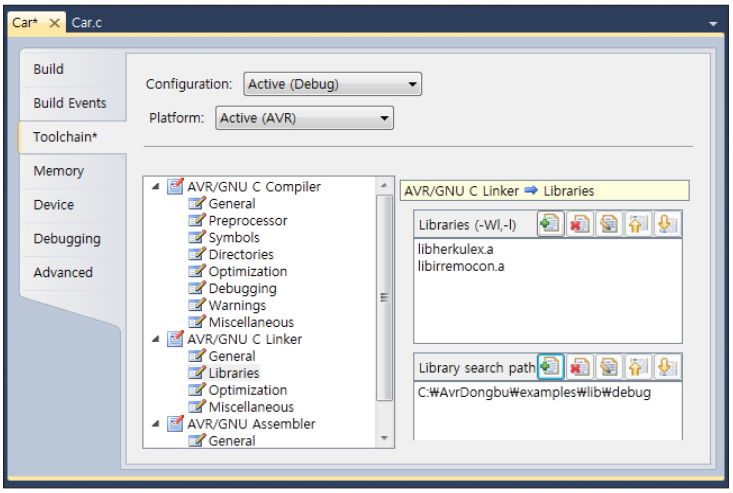
12 Debug

제공 라이브러리 폴더에서 lib 하위 폴더에 debug 를 선택합니다.



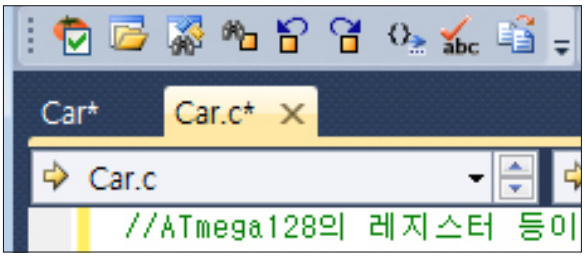
13 Path 완료

Relative Path(상대경로지정) 체크박스를 해지합니다.
Library search path 경로지정이 완료되었습니다.
OK 버튼을 클릭합니다.



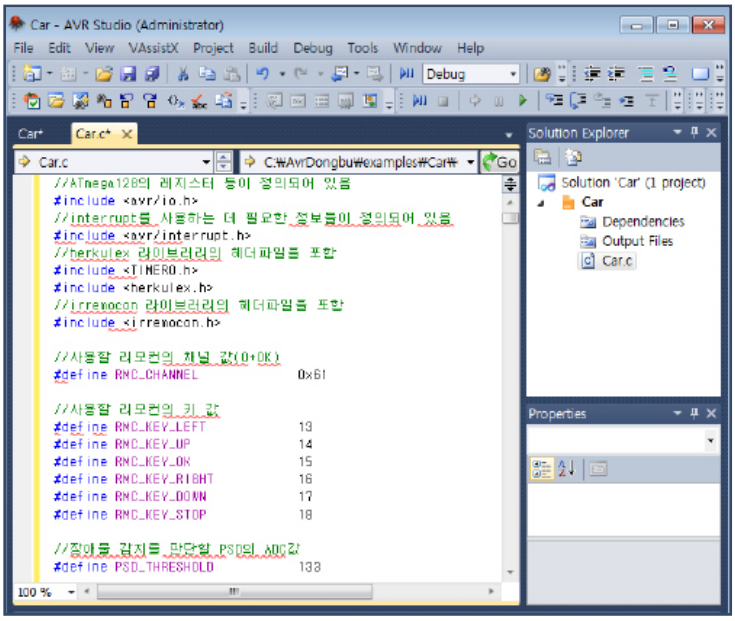
14 Link 완료

C Linker 라이브러리파일과 경로지정이 완료되었습니다.



15 저장

저장 버튼을 누르면 현재 설정값이 저장됩니다. 탭바 이름이 car* -> car로 변경됩니다.



16 코딩

Car.c 탭바를 선택하고, 아래 제공된 예제를 입력합니다.

예제

리모컨으로 덤프트럭 로봇을 제어하는 예제입니다. 위 키를 누르면 전진, 아래 키를 누르면 후진하며, 좌우 키에 따라서 좌회전, 우회전합니다. OK를 누르면 덤프 트럭의 짐받이를 세웠다가 다시 원위치로 돌아옵니다. 움직이는 중에 정지 버튼을 누르면 멈추며, 전방의 PSD로 장애물을 감지해서 전진 중에 장애물을 만나면 제자리에서 오른쪽으로 돌고 다시 전진합니다.

Car.c

```
//ATmega128의 레지스터 등이 정의되어 있음
#include <avr/io.h>
//interrupt를 사용하는 데 필요한 정보들이 정의되어 있음
#include <avr/interrupt.h>
//herkulex 라이브러리의 헤더파일을 포함
#include <TIMER0.h>
#include <herkulex.h>
//irremocon 라이브러리의 헤더파일을 포함
#include <irremocon.h>

//사용할 리모컨의 채널 값(0+OK)
#define RMC_CHANNEL                     0x61

//사용할 리모컨의 키 값
#define RMC_KEY_LEFT                    13
#define RMC_KEY_UP                      14
#define RMC_KEY_OK                      15
#define RMC_KEY_RIGHT                   16
#define RMC_KEY_DOWN                    17
#define RMC_KEY_STOP                    18

//장애물 감지를 판단할 PSD의 ADC값
#define PSD_THRESHOLD                   133

//I_JOG 패킷에서 역방향 회전 비트를 나타내는 마스크
#define INVERSE_TURN_MASK               0x4000

/**
* Timer0의 전역변수 gucTimerTick을 사용한 딜레이 함수다.
* gucTimerTick이 unsigned char이기 때문에
* 255 이상의 값을 넣을 수가 없기 때문에 사용한다.
*
* @author       Dongbu Robot
* @param        iDelay: 기다릴 시간(ms)
* @return       없음
* @version      1.0.0.0
*/
```

```
void delay_ms(int iDelay){
        //iDelay가 0보다 크면 반복
        while(iDelay>0){
                //iDelay가 500ms 보다 큰 경우
                if(iDelay>500){
                        //gucTimerTick에 250을 넣고 iDelay를 500 뺀다.
                        gucTimerTick=250;
                        iDelay -= 500;
                        //Timer0에 의해 gucTimerTick이 0이 될 때 까지 대기
                        while(gucTimerTick>0);
                }
                //iDelay가 500ms 이하인 경우
                else{
                        //gucTimerTick에 iDelay를 2로 나누어 넣고 iDelay를 0으로 만든다.
                        gucTimerTick=iDelay/2;
                        iDelay = 0;
                        //Timer0에 의해 gucTimerTick이 0이 될 때 까지 대기
                        while(gucTimerTick>0);
                }
        }
}

/**
* 덤프 트럭을 멈추는 함수다.
* 바퀴 서보(0, 2, 4, 5번)는 속도제어로 0을 보내고
* 조향 서보(1, 3번)는 정면을 바라보게 하고
* 짐받이 서보(6번)는 원래 위치로 보내는
* I_JOG 패킷을 구성해 서보로 보낸다.
*
* @author       Dongbu Robot
* @param        없음
* @return       없음
* @version      1.0.0.0
*/
void stop(void){
        //보낼 패킷이 저장될 패킷 구조체 선언
        DrsPacket stSendPacket;
        //for문에서 사용할 변수 선언
        unsigned char i;
        //I_JOG 패킷 크기, ID, CMD 설정
        stSendPacket.ucPacketSize = MIN_PACKET_SIZE + 7 * CMD_I_JOG_STRUCT_SIZE;
        stSendPacket.ucChipID = BROADCAST_ID;
        stSendPacket.ucCmd = CMD_I_JOG;
```

```
	//7개 서보에 공통적인 파라미터 설정 + ucId 설정
	for(i=0;i<7;i++){
		stSendPacket.unData.stIJogData.stIJog[i].stJog.reserved = 0;
		stSendPacket.unData.stIJogData.stIJog[i].stSet.ucStopFlag = 0;
		stSendPacket.unData.stIJogData.stIJog[i].stSet.ucLedGreen = 0;
		stSendPacket.unData.stIJogData.stIJog[i].stSet.ucLedBlue = 0;
		stSendPacket.unData.stIJogData.stIJog[i].stSet.ucLedRed = 0;
		stSendPacket.unData.stIJogData.stIJog[i].stSet.ucJogInvalid = 0;
		stSendPacket.unData.stIJogData.stIJog[i].stSet.reserved = 0;
		stSendPacket.unData.stIJogData.stIJog[i].ucId = i;
	}

	//0번 서보(왼쪽 앞 바퀴)를 정지(속도 제어)
	stSendPacket.unData.stIJogData.stIJog[0].stJog.uiValue = 0;
	stSendPacket.unData.stIJogData.stIJog[0].stSet.ucMode = 1;
	stSendPacket.unData.stIJogData.stIJog[0].ucPlayTime = 20;

	//1번 서보(왼쪽 조향)를 직진 방향으로(위치 제어)
	stSendPacket.unData.stIJogData.stIJog[1].stJog.uiValue = 789;
	stSendPacket.unData.stIJogData.stIJog[1].stSet.ucMode = 0;
	stSendPacket.unData.stIJogData.stIJog[1].ucPlayTime = 150;

	//2번 서보(오른쪽 앞 바퀴)를 정지(속도 제어)
	stSendPacket.unData.stIJogData.stIJog[2].stJog.uiValue = 0;
	stSendPacket.unData.stIJogData.stIJog[2].stSet.ucMode = 1;
	stSendPacket.unData.stIJogData.stIJog[2].ucPlayTime = 20;

	//3번 서보(오른쪽 조향)를 직진 방향으로(위치 제어)
	stSendPacket.unData.stIJogData.stIJog[3].stJog.uiValue = 235;
	stSendPacket.unData.stIJogData.stIJog[3].stSet.ucMode = 0;
	stSendPacket.unData.stIJogData.stIJog[3].ucPlayTime = 150;

	//4번 서보(왼쪽 뒤 바퀴)를 정지(속도 제어)
	stSendPacket.unData.stIJogData.stIJog[4].stJog.uiValue = 0;
	stSendPacket.unData.stIJogData.stIJog[4].stSet.ucMode = 1;
	stSendPacket.unData.stIJogData.stIJog[4].ucPlayTime = 20;

	//5번 서보(오른쪽 뒤 바퀴)를 정지(속도 제어)
	stSendPacket.unData.stIJogData.stIJog[5].stJog.uiValue = 0;
	stSendPacket.unData.stIJogData.stIJog[5].stSet.ucMode = 1;
	stSendPacket.unData.stIJogData.stIJog[5].ucPlayTime = 20;

	//6번 서보(짐받이)를 기본 위치로(위치 제어)
	stSendPacket.unData.stIJogData.stIJog[6].stJog.uiValue = 970;
	stSendPacket.unData.stIJogData.stIJog[6].stSet.ucMode = 0;
	stSendPacket.unData.stIJogData.stIJog[6].ucPlayTime = 100;

	//패킷 보내기
	hklx_SendPacket(stSendPacket);
}
```

```
/**
* 짐받이 서보(6번)를 기본 위치로 보내는
* I_JOG 패킷을 구성해 서보로 보낸다.
*
* @author       Dongbu Robot
* @param        없음
* @return       없음
* @version      1.0.0.0
*/
void load_on(void){
        //보낼 패킷이 저장될 패킷 구조체 선언
        DrsPacket stSendPacket;

        //I_JOG 패킷 크기, ID, CMD 설정
        stSendPacket.ucPacketSize = MIN_PACKET_SIZE + CMD_I_JOG_STRUCT_SIZE;
        stSendPacket.ucChipID = BROADCAST_ID;
        stSendPacket.ucCmd = CMD_I_JOG;

        //6번 서보(짐받이)를 위치 제어로 기본 위치로 보내는 파라미터 설정
        stSendPacket.unData.stIJogData.stIJog[0].stJog.uiValue = 970;
        stSendPacket.unData.stIJogData.stIJog[0].stJog.reserved = 0;
        stSendPacket.unData.stIJogData.stIJog[0].stSet.ucStopFlag = 0;
        stSendPacket.unData.stIJogData.stIJog[0].stSet.ucMode = 0;
        stSendPacket.unData.stIJogData.stIJog[0].stSet.ucLedGreen = 0;
        stSendPacket.unData.stIJogData.stIJog[0].stSet.ucLedBlue = 0;
        stSendPacket.unData.stIJogData.stIJog[0].stSet.ucLedRed = 0;
        stSendPacket.unData.stIJogData.stIJog[0].stSet.ucJogInvalid = 0;
        stSendPacket.unData.stIJogData.stIJog[0].stSet.reserved = 0;
        stSendPacket.unData.stIJogData.stIJog[0].ucId = 6;
        stSendPacket.unData.stIJogData.stIJog[0].ucPlayTime = 100;

        //패킷 보내기
        hklx_SendPacket(stSendPacket);
}

/**
* 짐받이 서보(6번)를 짐을 내리는 위치로 보내는
* I_JOG 패킷을 구성해 서보로 보낸다.
*
* @author       Dongbu Robot
* @param        없음
* @return       없음
* @version      1.0.0.0
*/
void load_off(void){
        //보낼 패킷이 저장될 패킷 구조체 선언
        DrsPacket stSendPacket;

        //I_JOG 패킷 크기, ID, CMD 설정
        stSendPacket.ucPacketSize = MIN_PACKET_SIZE + CMD_I_JOG_STRUCT_SIZE;
        stSendPacket.ucChipID = BROADCAST_ID;
        stSendPacket.ucCmd = CMD_I_JOG;
```

```
	//6번 서보(짐받이)를 위치 제어로 짐 내리는 위치로 보내는 파라미터 설정
	stSendPacket.unData.stIJogData.stIJog[0].stJog.uiValue = 700;
	stSendPacket.unData.stIJogData.stIJog[0].stJog.reserved = 0;
	stSendPacket.unData.stIJogData.stIJog[0].stSet.ucStopFlag = 0;
	stSendPacket.unData.stIJogData.stIJog[0].stSet.ucMode = 0;
	stSendPacket.unData.stIJogData.stIJog[0].stSet.ucLedGreen = 0;
	stSendPacket.unData.stIJogData.stIJog[0].stSet.ucLedBlue = 0;
	stSendPacket.unData.stIJogData.stIJog[0].stSet.ucLedRed = 0;
	stSendPacket.unData.stIJogData.stIJog[0].stSet.ucJogInvalid = 0;
	stSendPacket.unData.stIJogData.stIJog[0].stSet.reserved = 0;
	stSendPacket.unData.stIJogData.stIJog[0].ucId = 6;
	stSendPacket.unData.stIJogData.stIJog[0].ucPlayTime = 100;

	//패킷 보내기
	hklx_SendPacket(stSendPacket);
}

/**
* 조향 서보(1, 3번)를 틀어 좌회전 하게 하는
* I_JOG 패킷을 구성해 서보로 보낸다.
*
* @author		Dongbu Robot
* @param		없음
* @return		없음
* @version		1.0.0.0
*/
void left_turn(void){
	//보낼 패킷이 저장될 패킷 구조체 선언
	DrsPacket stSendPacket;

	//I_JOG 패킷 크기, ID, CMD 설정
	stSendPacket.ucPacketSize = MIN_PACKET_SIZE + 2 * CMD_I_JOG_STRUCT_SIZE;
	stSendPacket.ucChipID = BROADCAST_ID;
	stSendPacket.ucCmd = CMD_I_JOG;

	//1번 서보(왼쪽 조향)를 위치 제어로 좌회전 위치로 보내는 파라미터 설정
	stSendPacket.unData.stIJogData.stIJog[0].stJog.uiValue = 689;
	stSendPacket.unData.stIJogData.stIJog[0].stJog.reserved = 0;
	stSendPacket.unData.stIJogData.stIJog[0].stSet.ucStopFlag = 0;
	stSendPacket.unData.stIJogData.stIJog[0].stSet.ucMode = 0;
	stSendPacket.unData.stIJogData.stIJog[0].stSet.ucLedGreen = 0;
	stSendPacket.unData.stIJogData.stIJog[0].stSet.ucLedBlue = 0;
	stSendPacket.unData.stIJogData.stIJog[0].stSet.ucLedRed = 0;
	stSendPacket.unData.stIJogData.stIJog[0].stSet.ucJogInvalid = 0;
	stSendPacket.unData.stIJogData.stIJog[0].stSet.reserved = 0;
	stSendPacket.unData.stIJogData.stIJog[0].ucId = 1;
	stSendPacket.unData.stIJogData.stIJog[0].ucPlayTime = 50;

	//3번 서보(오른쪽 조향)를 위치 제어로 좌회전 위치로 보내는 파라미터 설정
	stSendPacket.unData.stIJogData.stIJog[1].stJog.uiValue = 135;
	stSendPacket.unData.stIJogData.stIJog[1].stJog.reserved = 0;
	stSendPacket.unData.stIJogData.stIJog[1].stSet.ucStopFlag = 0;
	stSendPacket.unData.stIJogData.stIJog[1].stSet.ucMode = 0;
	stSendPacket.unData.stIJogData.stIJog[1].stSet.ucLedGreen = 0;
	stSendPacket.unData.stIJogData.stIJog[1].stSet.ucLedBlue = 0;
```

```
	stSendPacket.unData.stIJogData.stIJog[1].stSet.ucLedRed = 0;
	stSendPacket.unData.stIJogData.stIJog[1].stSet.ucJogInvalid = 0;
	stSendPacket.unData.stIJogData.stIJog[1].stSet.reserved = 0;
	stSendPacket.unData.stIJogData.stIJog[1].ucId = 3;
	stSendPacket.unData.stIJogData.stIJog[1].ucPlayTime = 50;

	//패킷 보내기
	hklx_SendPacket(stSendPacket);
}

/**
* 조향 서보(1, 3번)를 틀어 우회전 하게 하는
* I_JOG 패킷을 구성해 서보로 보낸다.
*
* @author		Dongbu Robot
* @param		없음
* @return		없음
* @version		1.0.0.0
*/
void right_turn(void){
	//보낼 패킷이 저장될 패킷 구조체 선언
	DrsPacket stSendPacket;

	//I_JOG 패킷 크기, ID, CMD 설정
	stSendPacket.ucPacketSize = MIN_PACKET_SIZE + 2 * CMD_I_JOG_STRUCT_SIZE;
	stSendPacket.ucChipID = BROADCAST_ID;
	stSendPacket.ucCmd = CMD_I_JOG;

	//1번 서보(왼쪽 조향)를 위치 제어로 우회전 위치로 보내는 파라미터 설정
	stSendPacket.unData.stIJogData.stIJog[0].stJog.uiValue = 889;
	stSendPacket.unData.stIJogData.stIJog[0].stJog.reserved = 0;
	stSendPacket.unData.stIJogData.stIJog[0].stSet.ucStopFlag = 0;
	stSendPacket.unData.stIJogData.stIJog[0].stSet.ucMode = 0;
	stSendPacket.unData.stIJogData.stIJog[0].stSet.ucLedGreen = 0;
	stSendPacket.unData.stIJogData.stIJog[0].stSet.ucLedBlue = 0;
	stSendPacket.unData.stIJogData.stIJog[0].stSet.ucLedRed = 0;
	stSendPacket.unData.stIJogData.stIJog[0].stSet.ucJogInvalid = 0;
	stSendPacket.unData.stIJogData.stIJog[0].stSet.reserved = 0;
	stSendPacket.unData.stIJogData.stIJog[0].ucId = 1;
	stSendPacket.unData.stIJogData.stIJog[0].ucPlayTime = 50;

	//3번 서보(오른쪽 조향)를 위치 제어로 우회전 위치로 보내는 파라미터 설정
	stSendPacket.unData.stIJogData.stIJog[1].stJog.uiValue = 335;
	stSendPacket.unData.stIJogData.stIJog[1].stJog.reserved = 0;
	stSendPacket.unData.stIJogData.stIJog[1].stSet.ucStopFlag = 0;
	stSendPacket.unData.stIJogData.stIJog[1].stSet.ucMode = 0;
	stSendPacket.unData.stIJogData.stIJog[1].stSet.ucLedGreen = 0;
	stSendPacket.unData.stIJogData.stIJog[1].stSet.ucLedBlue = 0;
	stSendPacket.unData.stIJogData.stIJog[1].stSet.ucLedRed = 0;
	stSendPacket.unData.stIJogData.stIJog[1].stSet.ucJogInvalid = 0;
	stSendPacket.unData.stIJogData.stIJog[1].stSet.reserved = 0;
	stSendPacket.unData.stIJogData.stIJog[1].ucId = 3;
	stSendPacket.unData.stIJogData.stIJog[1].ucPlayTime = 50;
```

```
	//패킷 보내기
	hklx_SendPacket(stSendPacket);
}

/**
* 덤프 트럭을 전진하게 하는 함수다.
* 바퀴 서보(0, 2, 4, 5번)는 앞으로 굴러가도록 하고
* 조향 서보(1, 3번)는 정면을 바라보게 하는
* I_JOG 패킷을 구성해 서보로 보낸다.
*
* @author		Dongbu Robot
* @param		없음
* @return		없음
* @version		1.0.0.0
*/
void forward(void){
	//보낼 패킷과 받을 패킷이 저장될 패킷 구조체 선언
	DrsPacket stSendPacket;
	//for문에서 사용할 변수 선언
	unsigned char i;

	//I_JOG 패킷 크기, ID, CMD 설정
	stSendPacket.ucPacketSize = MIN_PACKET_SIZE + 6 * CMD_I_JOG_STRUCT_SIZE;
	stSendPacket.ucChipID = BROADCAST_ID;
	stSendPacket.ucCmd = CMD_I_JOG;

	//6개 서보에 공통적인 파라미터 설정 + ucId 설정
	for(i=0;i<6;i++){
		stSendPacket.unData.stIJogData.stIJog[i].stJog.reserved = 0;
		stSendPacket.unData.stIJogData.stIJog[i].stSet.ucStopFlag = 0;
		stSendPacket.unData.stIJogData.stIJog[i].stSet.ucLedGreen = 0;
		stSendPacket.unData.stIJogData.stIJog[i].stSet.ucLedBlue = 0;
		stSendPacket.unData.stIJogData.stIJog[i].stSet.ucLedRed = 0;
		stSendPacket.unData.stIJogData.stIJog[i].stSet.ucJogInvalid = 0;
		stSendPacket.unData.stIJogData.stIJog[i].stSet.reserved = 0;
		stSendPacket.unData.stIJogData.stIJog[i].ucId = i;
	}

	//0번 서보(왼쪽 앞 바퀴)를 전진 방향으로 회전(속도 제어)
	stSendPacket.unData.stIJogData.stIJog[0].stJog.uiValue = 800;
	stSendPacket.unData.stIJogData.stIJog[0].stSet.ucMode = 1;
	stSendPacket.unData.stIJogData.stIJog[0].ucPlayTime = 10;

	//1번 서보(왼쪽 조향)를 직진 방향으로(위치 제어)
	stSendPacket.unData.stIJogData.stIJog[1].stJog.uiValue = 789;
	stSendPacket.unData.stIJogData.stIJog[1].stSet.ucMode = 0;
	stSendPacket.unData.stIJogData.stIJog[1].ucPlayTime = 150;

	//2번 서보(오른쪽 앞 바퀴)를 전진 방향으로 회전(속도 제어)
	stSendPacket.unData.stIJogData.stIJog[2].stJog.uiValue = (800 | INVERSE_TURN_MASK);
	stSendPacket.unData.stIJogData.stIJog[2].stSet.ucMode = 1;
	stSendPacket.unData.stIJogData.stIJog[2].ucPlayTime = 10;

	//3번 서보(오른쪽 조향)를 직진 방향으로(위치 제어)
	stSendPacket.unData.stIJogData.stIJog[3].stJog.uiValue = 235;
	stSendPacket.unData.stIJogData.stIJog[3].stSet.ucMode = 0;
	stSendPacket.unData.stIJogData.stIJog[3].ucPlayTime = 150;
```

```
	//4번 서보(왼쪽 뒤 바퀴)를 전진 방향으로 회전(속도 제어)
	stSendPacket.unData.stIJogData.stIJog[4].stJog.uiValue = 800;
	stSendPacket.unData.stIJogData.stIJog[4].stSet.ucMode = 1;
	stSendPacket.unData.stIJogData.stIJog[4].ucPlayTime = 10;

	//5번 서보(오른쪽 뒤 바퀴)를 전진 방향으로 회전(속도 제어)
	stSendPacket.unData.stIJogData.stIJog[5].stJog.uiValue = (800 | INVERSE_TURN_MASK);
	stSendPacket.unData.stIJogData.stIJog[5].stSet.ucMode = 1;
	stSendPacket.unData.stIJogData.stIJog[5].ucPlayTime = 10;

	//패킷 보내기
	hklx_SendPacket(stSendPacket);
}

/**
* 덤프 트럭을 후진하게 하는 함수다.
* 바퀴 서보(0, 2, 4, 5번)는 뒤로 굴러가도록 하고
* 조향 서보(1, 3번)는 정면을 바라보게 하는
* I_JOG 패킷을 구성해 서보로 보낸다.
*
* @author	Dongbu Robot
* @param	없음
* @return	없음
* @version	1.0.0.0
*/
void backward(void){
	//보낼 패킷이 저장될 패킷 구조체 선언
	DrsPacket stSendPacket;
	//for문에서 사용할 변수 선언
	unsigned char i;

	//I_JOG 패킷 크기, ID, CMD 설정
	stSendPacket.ucPacketSize = MIN_PACKET_SIZE + 6 * CMD_I_JOG_STRUCT_SIZE;
	stSendPacket.ucChipID = BROADCAST_ID;
	stSendPacket.ucCmd = CMD_I_JOG;

	//6개 서보에 공통적인 파라미터 설정 + ucId 설정
	for(i=0;i<6;i++){
		stSendPacket.unData.stIJogData.stIJog[i].stJog.reserved = 0;
		stSendPacket.unData.stIJogData.stIJog[i].stSet.ucStopFlag = 0;
		stSendPacket.unData.stIJogData.stIJog[i].stSet.ucLedGreen = 0;
		stSendPacket.unData.stIJogData.stIJog[i].stSet.ucLedBlue = 0;
		stSendPacket.unData.stIJogData.stIJog[i].stSet.ucLedRed = 0;
		stSendPacket.unData.stIJogData.stIJog[i].stSet.ucJogInvalid = 0;
		stSendPacket.unData.stIJogData.stIJog[i].stSet.reserved = 0;
		stSendPacket.unData.stIJogData.stIJog[i].ucId = i;
	}

	//0번 서보(왼쪽 앞 바퀴)를 후진 방향으로 회전(속도 제어)
	stSendPacket.unData.stIJogData.stIJog[0].stJog.uiValue = (800 | INVERSE_TURN_MASK);
	stSendPacket.unData.stIJogData.stIJog[0].stSet.ucMode = 1;
	stSendPacket.unData.stIJogData.stIJog[0].ucPlayTime = 10;
```

```
	//1번 서보(왼쪽 조향)를 직진 방향으로(위치 제어)
	stSendPacket.unData.stIJogData.stIJog[1].stJog.uiValue = 789;
	stSendPacket.unData.stIJogData.stIJog[1].stSet.ucMode = 0;
	stSendPacket.unData.stIJogData.stIJog[1].ucPlayTime = 150;

	//2번 서보(오른쪽 앞 바퀴)를 후진 방향으로 회전(속도 제어)
	stSendPacket.unData.stIJogData.stIJog[2].stJog.uiValue = 800;
	stSendPacket.unData.stIJogData.stIJog[2].stSet.ucMode = 1;
	stSendPacket.unData.stIJogData.stIJog[2].ucPlayTime = 10;

	//3번 서보(오른쪽 조향)를 직진 방향으로(위치 제어)
	stSendPacket.unData.stIJogData.stIJog[3].stJog.uiValue = 235;
	stSendPacket.unData.stIJogData.stIJog[3].stSet.ucMode = 0;
	stSendPacket.unData.stIJogData.stIJog[3].ucPlayTime = 150;

	//4번 서보(왼쪽 뒤 바퀴)를 후진 방향으로 회전(속도 제어)
	stSendPacket.unData.stIJogData.stIJog[4].stJog.uiValue = (800 | INVERSE_TURN_MASK);
	stSendPacket.unData.stIJogData.stIJog[4].stSet.ucMode = 1;
	stSendPacket.unData.stIJogData.stIJog[4].ucPlayTime = 10;

	//5번 서보(오른쪽 뒤 바퀴)를 후진 방향으로 회전(속도 제어)
	stSendPacket.unData.stIJogData.stIJog[5].stJog.uiValue = 800;
	stSendPacket.unData.stIJogData.stIJog[5].stSet.ucMode = 1;
	stSendPacket.unData.stIJogData.stIJog[5].ucPlayTime = 10;

	//패킷 보내기
	hklx_SendPacket(stSendPacket);
}

/**
* 장애물을 만났을 때 제자리에서 빠르게 회전하게 하는 함수다.
* 바퀴 서보(0, 2, 4, 5번) 중 왼쪽 바퀴는 앞으로,
* 오른쪽 바퀴는 뒤로 굴러가게 해서 오른쪽으로 도는
* I_JOG 패킷을 구성해 서보로 보낸다.
*
* @author	Dongbu Robot
* @param	없음
* @return	없음
* @version	1.0.0.0
*/
void quick_turn(void){
	//보낼 패킷이 저장될 패킷 구조체 선언
	DrsPacket stSendPacket;
	//for문에서 사용할 변수 선언
	unsigned char i;

	//I_JOG 패킷 크기, ID, CMD 설정
	stSendPacket.ucPacketSize = MIN_PACKET_SIZE + 4 * CMD_I_JOG_STRUCT_SIZE;
	stSendPacket.ucChipID = BROADCAST_ID;
	stSendPacket.ucCmd = CMD_I_JOG;
```

```
	//4개 서보에 공통적인 파라미터 설정(0, 3번은 전진 방향, 2, 5번은 후진 방향)
	for(i=0;i<4;i++){
		stSendPacket.unData.stIJogData.stIJog[i].stJog.uiValue = 900;
		stSendPacket.unData.stIJogData.stIJog[i].stJog.reserved = 0;
		stSendPacket.unData.stIJogData.stIJog[i].stSet.ucStopFlag = 0;
		stSendPacket.unData.stIJogData.stIJog[i].stSet.ucMode = 1;
		stSendPacket.unData.stIJogData.stIJog[i].stSet.ucLedGreen = 0;
		stSendPacket.unData.stIJogData.stIJog[i].stSet.ucLedBlue = 0;
		stSendPacket.unData.stIJogData.stIJog[i].stSet.ucLedRed = 0;
		stSendPacket.unData.stIJogData.stIJog[i].stSet.ucJogInvalid = 0;
		stSendPacket.unData.stIJogData.stIJog[i].stSet.reserved = 0;
		stSendPacket.unData.stIJogData.stIJog[i].ucPlayTime = 10;
	}

	//ucId 설정
	stSendPacket.unData.stIJogData.stIJog[0].ucId = 0;
	stSendPacket.unData.stIJogData.stIJog[1].ucId = 2;
	stSendPacket.unData.stIJogData.stIJog[2].ucId = 4;
	stSendPacket.unData.stIJogData.stIJog[3].ucId = 5;

	//패킷 보내기
	hklx_SendPacket(stSendPacket);
}

int main(void)
{
	//보낼 패킷과 받을 패킷이 저장될 패킷 구조체 선언
	DrsPacket stSendPacket;
	//8비트 변수 선언
	unsigned char ucData, ucDataOld, ucLength, ucForward;
	//16비트 변수 선언
	int iADC;

	//포트 C의 입출력 방향 설정
	DDRC = 0b01111111;
	//포트 C의 출력 값 설정
	PORTC = 0b01111111;

	//VREF=AREF 입력 전압, 좌측 정렬 끔, ADC 6번 단극성 입력
	ADMUX=0b00000110;
	//ADC 활성화, ADIF 초기화, 클럭 분주비 128
	ADCSRA=0b10010111;

	//전체 인터럽트를 비활성화
	cli();

	//HerkuleX를 사용하기 위해 초기화
	hklx_Init(115200);

	//리모컨 수신 모듈을 사용하기 위해 초기화
	RMC_Init();

	//전체 인터럽트를 활성화
	sei();
```

```
//Torque Control에 0x60을 써서 토크를 거는 패킷 구성
stSendPacket.ucPacketSize = MIN_PACKET_SIZE + 3;
stSendPacket.ucChipID = 254;
stSendPacket.ucCmd = CMD_RAM_WRITE;
stSendPacket.unData.stRWData.ucAddress = 52;
stSendPacket.unData.stRWData.ucLen = 1;
stSendPacket.unData.stRWData.ucData[0] = 0x60;

//패킷 보내기
hklx_SendPacket(stSendPacket);

//ucDataOld를 기본 값으로 초기화
ucDataOld = RMC_ERROR_DATA;

//ucForward를 0으로 초기화
ucForward = 0;

//while 안의 내용을 무한 반복
while(1)
{
        //ADC 변환 시작
        ADCSRA |= (1<<ADSC);
        //ADC 변환 완료까지 기다림
        while(!(ADCSRA & (1<<ADIF)));
        //ADCL과 ADCH 값을 읽어서 저장
        iADC = ADCL | (ADCH<<8);
        //ADIF 초기화
        ADCSRA |= (1<<ADIF);

        // 전진 중에 장애물을 만난 경우 (0.65/5*1024 = 133.12)
        if(iADC > PSD_THRESHOLD && ucForward == 1){
                //잠시 멈추고 제자리에서 1초간 돈 후 전진
                stop();
                delay_ms(500);
                quick_turn();
                delay_ms(1000);
                forward();
                ucForward = 1;
        }

        //ucData, ucLength로 리모컨 값 읽어오기
        RMC_ReadRemocon(RMC_CHANNEL, &ucData, &ucLength);

        //리모컨 값이 전 값과 다르다면
        if(ucDataOld != ucData){
                //UP 키가 눌렸을 때 전진
                if(ucData == RMC_KEY_UP){
                        forward();
                        ucForward = 1;
                }
                //DOWN 키가 눌렸을 때 후진
                else if(ucData == RMC_KEY_DOWN){
                        backward();
                        ucForward = 0;
```

```
                    }
                    //LEFT 키가 눌렸을 때 좌측으로 방향 틀기
                    else if(ucData == RMC_KEY_LEFT){
                            left_turn();
                    }
                    //RIGHT 키가 눌렸을 때 우측으로 방향 틀기
                    else if(ucData == RMC_KEY_RIGHT){
                            right_turn();
                    }
                    //OK 키가 눌렸을 때 짐받이를 세웠다가 원래대로 돌아옴
                    else if(ucData == RMC_KEY_OK){
                            load_off();
                            delay_ms(1000);
                            load_on();
                    }
                    //정지 키가 눌렸을 때 정지
                    else if(ucData == RMC_KEY_STOP){
                            stop();
                            ucForward = 0;
                    }
            }

            //현재 데이터 값을 다른 변수에 저장
            ucDataOld = ucData;
    }

    //함수의 형태에 맞게 정수값 반환
    return 1;
}
```

예제설명

Car.c에서 #include문을 사용해 포함시켜주는 헤더 파일 중 libherkulex.a와 관련된 TIMER0.h와 herkulex.h에 대한 설명은 Chapter 8. HerkuleX 제어하기를 참조하세요. 또한 libirremocon.a와 관련된 irremocon.h에 대한 설명은 Chapter 12. 적외선 리모컨 신호 읽기를 참조하세요.

```
#include <avr/io.h>
#include <avr/interrupt.h>
```

avr/io.h 헤더 파일에는 DDRX, PORTX 레지스터 등 AVR 제어를 위해 필요한 레지스터 등이 정의되어 있고, avr/interrupt.h 헤더 파일에는 인터럽트를 사용하기 위한 함수들이 정의되어 있습니다.

```
#include <herkulex.h>
#include <TIMER0.h>
#include <irremocon.h>
```

herkulex.h 헤더 파일에는 패킷을 구성하고 보내기 위한 선언들과 구조체, 그리고 함수가 정의되어 있으므로 꼭 포함해야 합니다. 그리고 gucTimerTick을 사용해 delay를 구현하기 위하여, TIMER0.h도 포함시킵니다. irremocon.h에는 적외선 리모컨 수신 모듈을 초기화 하고 리모컨 데이터를 읽어오기 위한 함수가 정의되어 있으므로 포함시킵니다.

```
#define RMC_CHANNEL                    0x61
```

리모컨을 설정함에 따라서 채널 값은 0x61부터 0x6A까지가 될 수 있습니다. 이번 예제에서는 리모컨의 0번 버튼 + OK를 눌렀을 때 설정되는 기본값인 0x61을 채널 값으로 사용합니다.

```
#define RMC_KEY_LEFT                   13
                        …
#define RMC_KEY_STOP                   18
```

사용할 리모컨의 키와 그에 해당하는 키 값을 선언합니다. 상하좌우의 방향키와 OK, 그리고 빨간색 정지 키를 사용합니다. 키와 데이터 값을 선언해두면 키 값을 매번 찾아보는 일 없이 사용할 수 있어 편리합니다.

```
#define PSD_THRESHOLD                  133
```

PSD의 값을 읽어서 장애물이 앞에 있다고 판단할 때의 기준 값입니다. Chapter 6의 H/W 원리에 나온 거리와 PSD 출력 값의 그래프 상에서, 거리 20센티미터에서 PSD 전압은 약 0.65V가 나옵니다. 0.65V는 ADC 값으로 변환하면 0.65/5*1024=133.12 이므로 133을 기준으로 잡은 것입니다. 이보다 더 거리가 가까우면 0.65V보다 높은 전압이 출력되므로, 133 이상의 값이 ADC에서 읽혔다면 20센티미터 이내에 장애물이 있다는 뜻입니다.

```
#define INVERSE_TURN_MASK              0x4000
```

I_JOG와 S_JOG를 사용해서 서보를 속도제어 할때는 정방향 회전과 역방향 회전이 있습니다. 역방향 회전 시에는 stJog 구조체의 uiValue에 속도 값 외에도 14번 비트를 1로 설정해주어야 합니다. 그래서 위와 같은 마스크를 선언해서 속도 값과 마스크를 함께 Bitwise OR 연산하여 uiValue에 대입합니다.

```
void delay_ms(int iDelay){
        while(iDelay>0){
                if(iDelay>500){
                        gucTimerTick=250;
                        iDelay -= 500;
                        while(gucTimerTick>0);
                }
                else{
                        gucTimerTick=iDelay/2;
                        iDelay = 0;
                        while(gucTimerTick>0);
                }
        }
}
```

TIMER0을 이용한 딜레이 함수입니다. 매개변수로는 iDelayTick이 있는데, 얼마만큼의 시간동안 대기할 것인지 밀리초(ms) 단위로 나타낸 것입니다. gucTimerTick은 TIMER0.h에서 선언된 전역변수로, 평소에는 0으로 유지되다가 0이 아닌 값을 대입하면 libherkulex.a에 정의된 TIMER0 오버플로우 인터럽트 서비스 루틴에 의해서 2밀리초마다 1씩 줄어듭니다. 그래서 가령 10밀리초를 기다리고 싶으면 gucTimerTick에 5라는 값을 넣고 gucTimerTick이 0이 될 때까지 기다리면 됩니다. 이것을 이용해서 delay_ms() 함수를 구현합니다.

gucTimerTick은 unsigned char, 즉 부호 없는 1바이트 변수입니다. 그래서 0~255 까지 밖에 값을 가질 수 없습니다. gucTimerTick의 255는 510밀리초에 해당하는 시간입니다. gucTimerTick에 한번의 대입만으로는 약 0.5초 밖에 딜레이를 구현할 수 없는 것입니다. 그래서 이 함수에서는 딜레이 값 iDelay를 500밀리초 씩 끊어서 gucTimerTick에 대입합니다.

iDelay가 500보다 크면, gucTimerTick에는 250을 대입하고 iDelay를 500 뺀 값으로 다시 대입합니다. 그리고 while(gucTimerTick>0);을 사용해 gucTimerTick이 0보다 클 동안 무한 반복을 하면서 대기합니다. gucTimerTick이 0이 되면 while문에서 빠져나옵니다. iDelay가 500보다 작거나 같으면, gucTimerTick에는 iDelay를 2로 나누어 대입하고 iDelay를 0으로 만듭니다. 그리고 마찬가지로 while문을 사용해 gucTimerTick이 0이 될 때 까지 기다립니다. 이 과정을 while(iDelay>0)으로 감싸서 iDelay가 0보다 클 동안 반복하므로, iDelay가 0이 되었을 때 while문이 끝나고 delay_ms() 함수가 끝나게 됩니다.

```
void stop(void){
...
}
```

다음 함수는 stop() 함수입니다. 덤프트럭을 멈추는 I_JOG 패킷을 HerkuleX에 보내는 함수입니다.

```
DrsPacket stSendPacket;
unsigned char i;
```

변수를 선언합니다. stSendPacket은 보낼 I_JOG 패킷이 저장될 공간입니다. i는 for문을 제어할 모터 수만큼 반복하기 위해서 선언된 변수입니다.

```
stSendPacket.ucPacketSize = MIN_PACKET_SIZE + 7 * CMD_I_JOG_STRUCT_SIZE;
stSendPacket.ucChipID = BROADCAST_ID;
stSendPacket.ucCmd = CMD_I_JOG;
```

패킷의 크기와 ID, CMD를 설정하는 부분입니다. 패킷 크기는 I_JOG의 패킷 구조에 따라서, 기본 7바이트(MIN_PACKET_SIZE)에 모터 당 5바이트*모터 수가 더해집니다. 이 함수에서는 ID 0~6의 7개 모터를 제어할 것이므로 패킷 사이즈는 42가 될 것입니다. I_JOG 패킷이므로 ID는 BROADCAST_ID(0xFE), CMD는 CMD_I_JOG(0x05)입니다.

```
for(i=0;i<7;i++){
        stSendPacket.unData.stIJogData.stIJog[i].stJog.reserved = 0;
        stSendPacket.unData.stIJogData.stIJog[i].stSet.ucStopFlag = 0;
        stSendPacket.unData.stIJogData.stIJog[i].stSet.ucLedGreen = 0;
        stSendPacket.unData.stIJogData.stIJog[i].stSet.ucLedBlue = 0;
        stSendPacket.unData.stIJogData.stIJog[i].stSet.ucLedRed = 0;
        stSendPacket.unData.stIJogData.stIJog[i].stSet.ucJogInvalid = 0;
        stSendPacket.unData.stIJogData.stIJog[i].stSet.reserved = 0;
        stSendPacket.unData.stIJogData.stIJog[i].ucId = i;
}
```

Jog 명령은 여러가지 설정할 수 있는 파라미터가 많기 때문에, 모터를 하나하나씩 파라미터 설정을 하다가는 코드가 너무 길어집니다. 그래서 값이 공통되는 파라미터를 for문을 통해서 한꺼번에 설정해줍니다. 그리고 마지막에 ucId도 i로 설정해서, stIJog 배열의 인덱스와 모터 ID가 같게 합니다.

```
stSendPacket.unData.stIJogData.stIJog[0].stJog.uiValue = 0;
stSendPacket.unData.stIJogData.stIJog[0].stSet.ucMode = 1;
stSendPacket.unData.stIJogData.stIJog[0].ucPlayTime = 20;
                        ...
stSendPacket.unData.stIJogData.stIJog[6].stJog.uiValue = 970;
stSendPacket.unData.stIJogData.stIJog[6].stSet.ucMode = 0;
stSendPacket.unData.stIJogData.stIJog[6].ucPlayTime = 100;
```

0번 모터부터 6번 모터까지 각자 제어 모드와 속도/위치 값, 그리고 실행 시간을 설정합니다. 바퀴에 해당하는 0, 2, 4, 5번 모터는 속도 제어 모드로 속도 값은 0으로 정지시키고, 조향 축에 해당하는 1, 3번 서보는 위치 제어 모드로 각각 789, 235로 보내 나란히 직진하는 방향을 향하게 합니다. 짐받이인 6번 서보는 위치 제어 모드로 970으로 보내 기본 위치에 있게 합니다.

```
hklx_SendPacket(stSendPacket);
```

hklx_SendPacket() 함수를 사용해서 위에서 구성한 패킷을 HerkuleX로 전송합니다.

```
void load_on(void){
...
}
```

다음 함수는 load_on() 함수입니다. 덤프트럭의 짐받이 서보(6번)을 기본 위치로 보냅니다.

```
DrsPacket stSendPacket;

stSendPacket.ucPacketSize = MIN_PACKET_SIZE + CMD_I_JOG_STRUCT_SIZE;
stSendPacket.ucChipID = BROADCAST_ID;
stSendPacket.ucCmd = CMD_I_JOG;
```

보낼 패킷을 저장하는 stSendPacket 구조체를 선언하고 패킷 크기, ID, CMD를 설정합니다. 6번 서보만 제어할 것이므로 이 때 패킷 크기는 기본 7바이트(MIN_PACKET_SIZE)에 5바이트가 더해져 12가 됩니다.

```
stSendPacket.unData.stIJogData.stIJog[0].stJog.uiValue = 970;
                                    …
stSendPacket.unData.stIJogData.stIJog[0].stSet.ucMode = 0;
                                    …
stSendPacket.unData.stIJogData.stIJog[0].ucId = 6;
stSendPacket.unData.stIJogData.stIJog[0].ucPlayTime = 100;

hklx_SendPacket(stSendPacket);
```

stJog의 uiValue는 기본 위치인 970으로, stSet의 ucMode는 0(위치 제어), ucId는 6, 실행 시간은 100(약 1.12초)로 설정한 후 패킷을 보냅니다. 짐받이 서보가 기본 위치로 이동합니다.

```
void load_off(void){
…
}
```

다음 함수는 load_off 함수입니다. 덤프트럭의 짐받이 서보(6번)을 짐을 내리는 위치로 보냅니다. 이 함수의 내용은 load_on 함수와 거의 동일합니다.

```
stSendPacket.unData.stIJogData.stIJog[0].stJog.uiValue = 700;
```

load_off 함수가 load_on 함수와 다른 부분입니다. 6번 서보를 보낼 위치만 다르고 모두 동일합니다. 700으로 짐받이 서보를 보내면 덤프트럭이 화물을 내리듯이 짐받이를 비스듬히 들게 됩니다.

```
void left_turn(void){
…
}
```

조향 서보(1, 3번)을 움직여 좌회전 하는 방향으로 앞바퀴를 트는 함수입니다.

```
DrsPacket stSendPacket;

stSendPacket.ucPacketSize = MIN_PACKET_SIZE + 2 * CMD_I_JOG_STRUCT_SIZE;
stSendPacket.ucChipID = BROADCAST_ID;
stSendPacket.ucCmd = CMD_I_JOG;
```

보낼 패킷을 저장하는 stSendPacket 구조체를 선언하고 패킷 크기, ID, CMD를 설정합니다. 1, 3번 서보를 제어할 것이므로 이때 패킷 크기는 기본 7바이트(MIN_PACKET_SIZE)에 5*2바이트가 더해져 17이 됩니다.

```
stSendPacket.unData.stIJogData.stIJog[0].stJog.uiValue = 689;
…
stSendPacket.unData.stIJogData.stIJog[0].stSet.ucMode = 0;
…
stSendPacket.unData.stIJogData.stIJog[0].ucId = 1;
stSendPacket.unData.stIJogData.stIJog[0].ucPlayTime = 50;

stSendPacket.unData.stIJogData.stIJog[1].stJog.uiValue = 135;
…
stSendPacket.unData.stIJogData.stIJog[1].stSet.ucMode = 0;
…
stSendPacket.unData.stIJogData.stIJog[1].ucId = 3;
stSendPacket.unData.stIJogData.stIJog[1].ucPlayTime = 50;

hklx_SendPacket(stSendPacket);
```

1번 모터와 3번 모터를 위치제어 하여 좌회전 방향으로 앞바퀴를 틀도록 파라미터를 설정 합니다. stJog의 uiValue는 기본 위치인 789, 235보다 100 작게 각각 689, 135으로, stSet의 ucMode는 0(위치 제어), ucId는 각각 1, 3, 실행 시간은 50(약 0.56초)로 설정한 후 패킷을 보냅니다.

```
void right_turn(void){
…
}
```

조향 서보(1, 3번)을 움직여 우회전 하는 방향으로 앞바퀴를 트는 함수입니다. 이 함수의 내용은 left_turn() 함수와 거의

```
SendPacket.unData.stIJogData.stIJog[0].stJog.uiValue = 889;
                                …
stSendPacket.unData.stIJogData.stIJog[1].stJog.uiValue = 335;
```

load_off 함수가 load_on 함수와 다른 부분입니다. 1, 3번 서보를 보낼 위치만 다르고 모두 동일합니다. 이번에는 기본 위치인 789, 235보다 100 크게 각각 889, 335로 설정합니다.

```
void forward(void){
…
}
```

이 함수는 트럭을 전진하게 하는 함수입니다. 바퀴 서보(0, 2, 4, 5번)는 앞으로 굴러가는 방향으로 속도 제어하고 조향 서보(1, 3번)는 정면을 바라보도록 I_JOG 패킷을 보냅니다.

```
DrsPacket stSendPacket;
unsigned char i;
```

변수를 선언합니다. stSendPacket은 보낼 I_JOG 패킷이 저장될 공간입니다. i는 for문을 제어할 모터 수만큼 반복하기 위해서 선언된 변수입니다.

```
stSendPacket.ucPacketSize = MIN_PACKET_SIZE + 6 * CMD_I_JOG_STRUCT_SIZE;
stSendPacket.ucChipID = BROADCAST_ID;
stSendPacket.ucCmd = CMD_I_JOG;
```

패킷의 크기와 ID, CMD를 설정하는 부분입니다. 이 함수에서는 ID 0~5의 6개 모터를 제어할 것이므로 패킷 크기는 I_JOG의 패킷 구조에 따라 기본 7바이트(MIN_PACKET_SIZE)에 5*6바이트가 더해져 37이 될 것입니다.

```
for(i=0;i<6;i++){
…
}
```

값이 공통되는 파라미터를 for문을 통해서 한꺼번에 설정해줍니다. 그리고 마지막에 ucId도 i로 설정해서, stIJog 배열의 인덱스와 모터 ID가 같게 합니다.

```
stSendPacket.unData.stIJogData.stIJog[0].stJog.uiValue = 800;
stSendPacket.unData.stIJogData.stIJog[0].stSet.ucMode = 1;
stSendPacket.unData.stIJogData.stIJog[0].ucPlayTime = 10;

stSendPacket.unData.stIJogData.stIJog[5].stJog.uiValue = (800 | INVERSE_TURN_MASK);
stSendPacket.unData.stIJogData.stIJog[5].stSet.ucMode = 1;
stSendPacket.unData.stIJogData.stIJog[5].ucPlayTime = 10;
```

0번 모터부터 5번 모터까지 각자 제어 모드와 속도/위치 값, 그리고 실행 시간을 설정합니다. 바퀴에 해당하는 0, 2, 4, 5번 모터는 속도 제어 모드로 속도 값은 좌측의 0, 4번은 정방향 800, 우측의 2, 5번은 역방향 800으로 설정해서 전진 방향으로 모든 바퀴가 돌아가도록 합니다. 2, 5번의 stJog.uiValue를 설정할 때 800과 INVERSE_TURN_MASK를 Bitwise OR 연산한 값을 집어 넣는 것은, 역방향 회전을 나타내는 것입니다. 조향 축에 해당하는 1, 3번 서보는 위치 제어 모드로 각각 789, 235로 보내 나란히 직진하는 방향을 향하게 합니다.

```
hklx_SendPacket(stSendPacket);
```

hklx_SendPacket() 함수를 사용해서 위에서 구성한 패킷을 HerkuleX로 전송합니다.

```
void backward(void){
…
}
```

이 함수는 트럭을 후진하게 하는 함수입니다. 바퀴 서보(0, 2, 4, 5번)는 뒤로 굴러가는 방향으로 속도 제어하고 조향 서보(1, 3번)는 정면을 바라보도록 I_JOG 패킷을 보냅니다. 이 함수는 forward() 함수와 내용이 거의 동일합니다.

```
stSendPacket.unData.stIJogData.stIJog[0].stJog.uiValue = (800 | INVERSE_TURN_MASK);
                                        …
stSendPacket.unData.stIJogData.stIJog[2].stJog.uiValue = 800;
                                        …
stSendPacket.unData.stIJogData.stIJog[4].stJog.uiValue = (800 | INVERSE_TURN_MASK);
                                        …
stSendPacket.unData.stIJogData.stIJog[5].stJog.uiValue = 800;
```

backward() 함수가 forward() 함수와 다른 부분입니다. 정방향/역방향을 직진과 반대로 해서 후진 방향으로 모든 바퀴 모터가 움직이도록 설정합니다.

```
void guide turn(void){
…
}
```

quick_turn() 함수는 장애물을 만났을 때 사용됩니다. 장애물을 피하기 위해서 왼쪽 바퀴는 전진, 오른쪽 바퀴는 후진하여 제자리에서 오른쪽으로 도는 패킷을 보냅니다.

```
DrsPacket stSendPacket;
unsigned char i;

stSendPacket.ucPacketSize = MIN_PACKET_SIZE + 4 * CMD_I_JOG_STRUCT_SIZE;
stSendPacket.ucChipID = BROADCAST_ID;
stSendPacket.ucCmd = CMD_I_JOG;
```

보낼 패킷을 저장하는 stSendPacket 구조체를 선언하고, for문에서 사용할 변수 i를 선언하고, 패킷 크기, ID, CMD를 설정합니다. 0, 2, 4, 5번 서보를 제어할 것이므로 이 때 패킷 크기는 기본 7바이트(MIN_PACKET_SIZE)에 5*4바이트가 더해져 27이 됩니다.

```
for(i=0;i<4;i++){
        stSendPacket.unData.stIJogData.stIJog[i].stJog.uiValue = 900;
                                ...
        stSendPacket.unData.stIJogData.stIJog[i].stSet.ucMode = 1;
                                ...
        stSendPacket.unData.stIJogData.stIJog[i].ucPlayTime = 10;
}
```

값이 공통되는 파라미터를 for문을 통해서 한꺼번에 설정해줍니다. 이 함수에서는 모든 서보의 제어 모드, 속도 값, 실행 시간이 같으므로 for문에서 한꺼번에 설정합니다. 단 이번에는 서보의 ID가 듬성듬성 있기 때문에 ucId를 i로 설정하는 부분은 빠졌습니다.

```
stSendPacket.unData.stIJogData.stIJog[0].ucId = 0;
stSendPacket.unData.stIJogData.stIJog[1].ucId = 2;
stSendPacket.unData.stIJogData.stIJog[2].ucId = 4;
stSendPacket.unData.stIJogData.stIJog[3].ucId = 5;

hklx_SendPacket(stSendPacket);
```

ucId에 대상 서보의 ID를 집어 넣고 패킷을 보냅니다. 이 패킷을 보내면 덤프 트럭은 제자리에서 오른쪽으로 회전합니다.

이제부터는 main 함수에 대한 설명입니다.

```
DrsPacket stSendPacket;
unsigned char ucData, ucDataOld, ucLength, ucForward;
int iADC;
```

메인 함수에서 사용할 변수를 선언합니다. DrsPacket 구조체 타입으로 패킷을 보낼 때 쓸 변수를 선언합니다. ucData는 현재 while 루프에서 읽은 리모컨의 값이고, ucDataOld는 직전의 while 루프에서 읽은 리모컨의 값입니다. 매번 while 루프의 마지막에서 ucDataOld는 ucData의 값으로 업데이트 됩니다. 이 두 변수는 리모컨이 눌리거나 떼져서 리모컨 값이 바뀐 순간에만 타이머/카운터2 값을 바꾸기 위해서 사용합니다. ucLength는 리모컨이 눌린 길이를 읽어올 변수입니다. ucForward는 현재 로봇이 전진 중인지를 나타내는 변수입니다. 1일 경우 전진 중이라는 뜻입니다. iADC는 읽은 ADC 값이 저장될 16비트 정수 변수입니다.

```
DDRC = 0b01111111;
PORTC = 0b01111111;
```

포트 C의 0~6번 핀을 출력으로 설정하고 출력 값을 1로 만듭니다. 그러면 모든 LED가 꺼지게 됩니다.

```
ADMUX=0b00000110;
ADCSRA=0b10010111;
```

ADC를 단극성 모드로 ADC6 포트로부터 아날로그 입력을 읽어오기 위한 레지스터 설정을 합니다. ADMUX부터 살펴보면, DRC의 회로상에서 기준 전압은 ATmega128의 동작전압인 5V와 동일하게 AREF 핀으로 연결되어 있으므로 REFS1:0 비트는 0b00으로 설정하며, 좌측 정렬을 사용하지 않으므로 ADLAR도 0으로 설정합니다. 그리고 ADC6 포트를 단극성 모드로 읽을 것이므로 MUX4:0 비트는 0b00110으로 설정합니다.
ADCSRA에서는 우선 ADC를 활성화 하기 위해 ADEN 비트를 1로 설정하고, 아직 변환 시작을 할 것이 아니고 프리 러닝도 사용하지 않으므로 ADSC, ADFR은 0으로 설정합니다. ADIF의 경우는 혹시라도 1로 설정되어 있을 수 있으므로 0으로 다시 초기화하기 위해 1을 써주며, 인터럽트를 사용하지 않을 것이므로 ADIE는 0으로 설정합니다. 마지막으로 ATmega128의 16MHz 클럭 주파수를 ADC에 적합한 50~200kHz로 나누어주기 위해 ADPS2:0 비트를 0b111로 설정해서 128로 나눈 125kHz를 ADC 클럭으로 공급하도록 설정합니다.

ADC 레지스터에 대한 자세한 내용은 Chapter 6을 참조하세요.

```
cli();
hklx_Init(115200);
RMC_Init();
sei();
```

인터럽트에 대한 설정을 바꾸는 중에 인터럽트가 발생해 예상치 못한 오동작이 발생하는 것을 방지하기 위해서 우선 cli() 명령을 통해서 전체 인터럽트를 비활성화 시킵니다. 그 후 hklx_Init() 함수를 사용해서 HerkuleX를 제어하기 위한 타이머/카운터0과 USART0을 초기화합니다. 사용할 보드 레이트가 115200이므로 파라미터로 그 값을 사용합니다. 그리고 RMC_Init() 함수를 사용해 리모컨 신호를 받기 위한 외부 인터럽트 4번 핀과 타이머/카운터1을 초기화합니다. 초기화가 끝난 후에는 sei() 명령을 통해서 전체 인터럽트를 활성화 합니다.

```
stSendPacket.ucPacketSize = MIN_PACKET_SIZE + 3;
stSendPacket.ucChipID = BROADCAST_ID;
stSendPacket.ucCmd = CMD_RAM_WRITE;
stSendPacket.unData.stRWData.ucAddress = 52;
stSendPacket.unData.stRWData.ucLen = 1;
stSendPacket.unData.stRWData.ucData[0] = 0x60;

hklx_SendPacket(stSendPacket);
```

모터가 스스로 움직이게 제어하기 위해서는 모터에 토크가 인가되어야 합니다. 위의 코드가 토크를 인가시키기 위한 RAM_WRITE 패킷을 구성하고, 보내는 부분입니다. 패킷의 SIZE는 기본 MIN_PACKET_SIZE(7바이트)에다가, 주소 + 길이 + 데이터(1바이트)로 총 MIN_PACKET_SIZE+3(10바이트)이며, 모든 연결된 모터에 보내야 하므로 ID는 BROADCAST_ID, 즉 254 입니다. CMD는 CMD_RAM_WRITE(0x03)이며, unData 공용체의 멤버 중 stRWData 구조체의 ucAddress, ucLen, ucData[0]의 값을 각각 52, 1, 0x60으로 만들어줍니다. 그 후에는 hklx_SendPacket() 함수를 호출해 패킷을 보냅니다.

```
ucDataOld = RMC_ERROR_DATA;
ucForward = 0;
```

앞서 설명했듯이, ucDataOld 변수에는 while문에서 직전에 읽어온 리모컨의 값이 저장됩니다. 그런데 아직 while문에 진입하기 전이라 리모컨 값을 읽기 전이기 때문에, 기본 값인 RMC_ERROR_DATA를 ucDataOld에 저장합니다. 또한 지금은 덤프 트럭이 멈추어 있으므로, 전진 중인지 여부를 나타내는 ucForward 변수도 0으로 초기화 합니다.

```
ADCSRA |= (1<<ADSC);
while(!(ADCSRA & (1<<ADIF)));
iADC = ADCL | (ADCH<<8);
ADCSRA |= (1<<ADIF);
```

이제부터는 while문에 대한 설명입니다. while문의 처음 부분에서는 ADC 변환을 계속 반복합니다. 우선 ADCSRA의 ADSC 비트를 1로 만들어 줍니다. 그러면 ADC 변환이 시작됩니다. while(!(ADCSRA & (1<<ADIF))); 명령을 통해서 ADCSRA의 ADIF가 0인 동안 계속 대기합니다. ADC 변환이 끝나고 ADIF 비트가 1이 되는 순간 while문을 빠져 나와 다음으로 진행합니다.
ADC 변환이 완료되고 ADCL과 ADCH가 업데이트 되면, ADCL과 ADCH 값을 읽어서 iADC에 저장합니다. ADCH는 ADC 값의 상위 1바이트에 해당하므로 8비트(1바이트)만큼 왼쪽으로 <<(shift left)연산을 해서 ADCL과 |(bitwise OR) 연산해 iADC에 저장하면 10비트 데이터를 iADC에 알맞게 저장할 수 있습니다.
우리는 ADC 변환 완료 인터럽트를 사용하지 않으므로, ADIF 비트가 자동으로 0으로 설정되지 않습니다. 그래서 ADC값을 읽은 후에는 인터럽트 플래그를 다시 0으로 초기화 시켜야 합니다. ADCSRA의 ADIF 비트에 1을 써주어 이 비트를 0으로 초기화 시킵니다.

```
if(iADC > PSD_THRESHOLD && ucForward == 1){
        stop();
        delay_ms(500);
        quick_turn();
        delay_ms(1000);
        forward();
        ucForward = 1;
}
```

iADC 값이 PSD_THRESHOLD보다 커서 20센티미터 이내에 장애물이 있는 것으로 판단되며, 현재 전진하고 있는 경우 if문 안쪽으로 진입합니다. 우선 stop() 함수를 호출해 멈춘 후 0.5초 동안 잠시 대기합니다. 그리고 quick_turn() 함수를 호출해 제자리에서 우회전 하는 패킷을 보내고, 1초 동안 우회전하도록 기다립니다. 그 후에는 forward() 함수를 호출해 앞으로 가게 합니다. 그리고 ucForward를 1로 만들어 전진 중임을 나타냅니다.

```
RMC_ReadRemocon(RMC_CHANNEL, &ucData, &ucLength);
```

RMC_ReadRemocon() 함수를 사용해서 리모컨의 값을 읽어옵니다. 첫 번째 매개 변수에는 원하는 리모컨 채널인 RMC_CHANNEL을 넣어주고, 두 번째, 세 번째 매개 변수에는 ucData와 ucLength의 주소를 넣어서 RMC_CHANNEL 채널에서 들어온 데이터와 눌린 길이가 ucData와 ucLength에 저장되게 합니다.

```
if(ucDataOld != ucData){
…
}
```

직전 while 루프에서 읽어온 리모컨 값(ucDataOld)과 방금 읽은 리모컨 값(ucData)을 비교해서, 서로 다른 경우에 if문의 안쪽 내용을 실행합니다. 리모컨 키가 눌리는 순간에만 명령을 내리기 위한 부분입니다.

```
If(ucData == RMC_KEY_UP){
        forward();
        ucForward = 1;
}
else if(ucData == RMC_KEY_DOWN){
        backward();
        ucForward = 0;
}
else if(ucData == RMC_KEY_LEFT){
        left_turn();
}
else if(ucData == RMC_KEY_RIGHT){
        right_turn();
}
else if(ucData == RMC_KEY_OK){
        load_off();
        delay_ms(1000);
        load_on();
}
else if(ucData == RMC_KEY_STOP){
        stop();
        ucForward = 0;
}
```

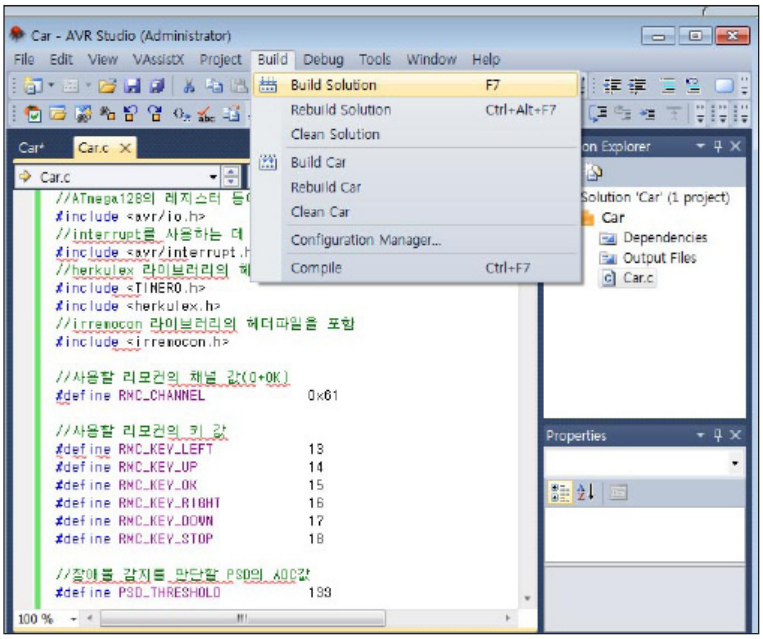
ucData의 값에 따라서 해당하는 동작을 수행하는 부분입니다. ucData가 RMC_KEY_UP과 같다면, 즉 위쪽 방향키가 눌렸다면 forward 함수를 호출해 트럭이 앞으로 가도록 하고 ucForward를 1로 바꿉니다. 이어지는 else if문도 리모컨 키와 그에 해당하는 동작을 나타냅니다. 아래쪽 방향키가 눌렸다면 backward 함수를 호출하고, 왼쪽 방향키가 눌렸다면 left_turn, 오른쪽 방향키가 눌렸다면 right_turn을 호출합니다. OK 키가 눌렸다면 load_off 함수를 호출해 짐을 내리는 동작을 취하게 하고, 1초 후에 다시 load_on 함수를 호출해 짐받이를 원위치 시킵니다. 정지 키가 눌렸다면 stop 함수를 호출합니다. 후진할 때와 정지할 때에는 ucForward를 0으로 만들어서, 후진 중일 때와 정지 상태에서는 PSD로 인식한 장애물에 대해서 반응하지 않도록 합니다.

```
ucDataOld = ucData;
```

while 루프가 끝날 때마다 현재 루프에서 읽은 ucData의 값을 ucDataOld에 대입해, 다음 루프에서 비교하는 데에 사용합니다.

실행

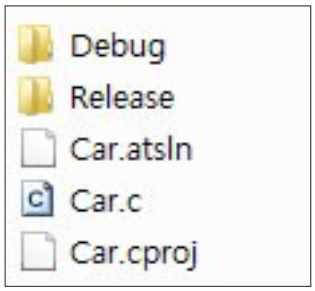
덤프트럭 로봇에 연결, 펌웨어를 다운로드 하고 실행하면 리모컨 입력에 따라서 덤프트럭 로봇을 제어할 수 있습니다.



01 Build

Build 메뉴를 선택하여 Build Solution 을 클릭하여 Build 시킵니다.

1 succeeded 0 failed 가 나오면 에러없이 컴파일 완료됐다는 것을 의미합니다.



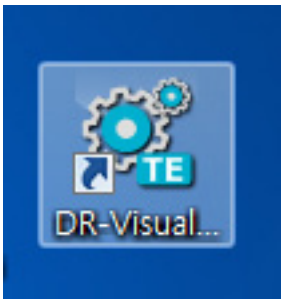
02 Debug 폴더

Build 후에 소스가 있는 폴더에 들어가면 Debug 폴더가 생성 되어있습니다. 클릭하여 폴더 안의 내용을 봅니다.

03 hex 확인

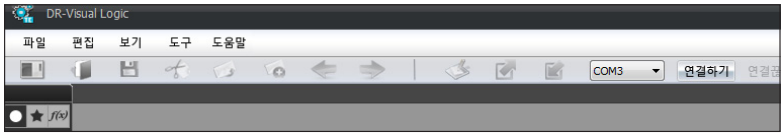
Debug 폴더 안의 hex 파일 확인합니다. Hex 파일이 제어기에 포팅시키는 파일입니다.

DR-Visual Logic 에서 펌웨어 복구를 실시합니다.



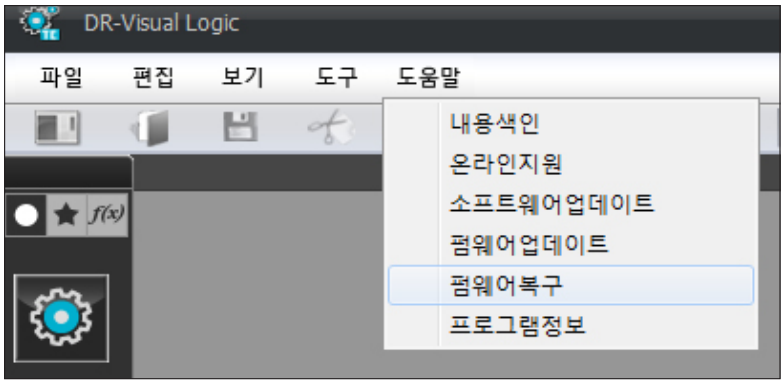
04 Visual Logic

DR-Visual Logic 을 실행시킵니다.



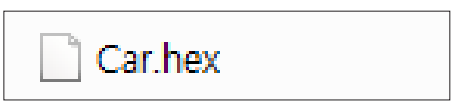
05 연결하기

COM 번호를 설정하고 연결하기를 클릭합니다.



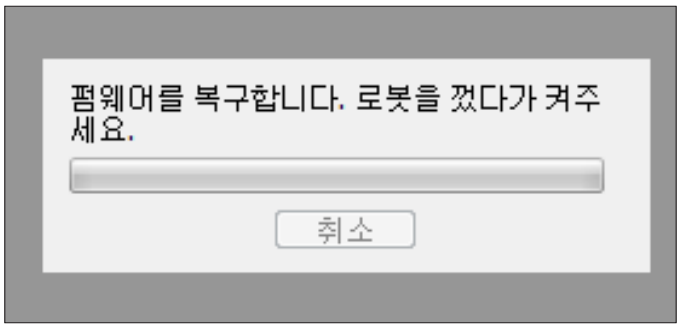
06 펌웨어 복구

도움말 메뉴에서 펌웨어복구를 클릭합니다.



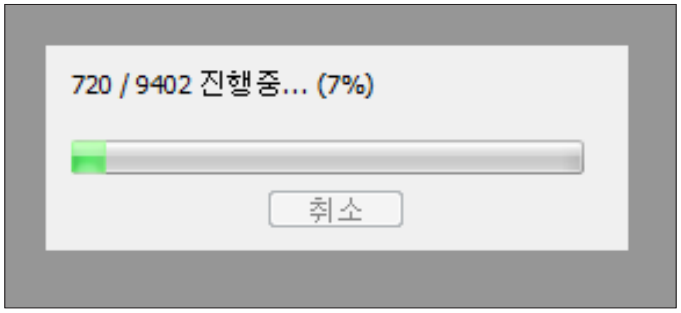
07 불러오기

Debug 폴더에서 Hex 파일을 불러옵니다



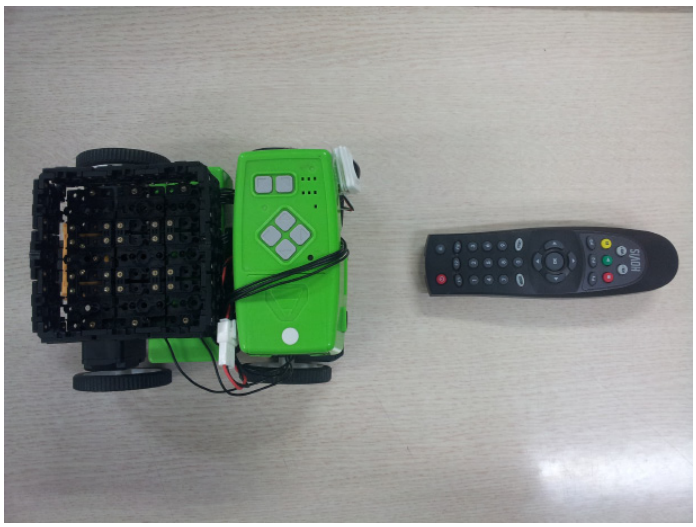
08 펌웨어복구

로봇을 껐다가 켜주세요 라는 문구가 나옵니다.
제어기의 전원버튼을 한번 껐다가 다시 켜주세요.



09 복구시작

펌웨어 복구가 시작됩니다.



10 실행

DRC를 덤프트럭 로봇에 연결, 펌웨어를 다운로드하고 실행하면 리모컨 입력에 따라서 덤프트럭 로봇을 제어할 수 있습니다.

별첨

Register

HOVIS

비휘발성 Register Map

제어기의 레지스터는 Non-Volatile(비휘발성) Register와 Volatile(휘발성) Register로 구분되며, 제어기의 현재 상태와 구동에 대한 Data로 구성되어 있습니다. 사용자는 프로토콜의 명령을 통해서 Register의 Data를 읽거나 변경함으로써 제어기를 제어할 수 있고, DR-Visual Logic 프로그램을 통하여 로봇의 행동을 프로그램할 때도 사용할 수 있습니다.

Non-Volatile Register (EEP Register, 비휘발성) Map

비휘발성 레지스터는 전원이 꺼진 상태에서도 데이터가 지워지지 않는 영역으로, 제어기 동작의 기본적인 설정 값들이 저장됩니다. 비휘발성 레지스터의 값은 전원이 켜진 순간 휘발성 레지스터의 해당 영역으로 복사됩니다. 비휘발성 레지스터의 값을 변경하면 그 당시에는 동작에 직접적인 영향을 주지 않지만, 제어기를 Reboot 하거나 다시 전원을 켤 때에는 변경된 값이 휘발성 레지스터로 복사되면서 동작에 영향을 미치게 됩니다.

- **Address**

  Address는 Register의 주소로서, Register를 읽거나 쓰기 위해서는 Packet에 해당 레지스터의 주소를 지정해주어야 합니다.

- **Default**

  공장에서 최초 출하 시의 설정 값입니다. Rollback 명령을 사용하면 비휘발성 레지스터를 Default 값으로 되돌릴 수 있습니다.

- **Valid Range**

  레지스터가 가질 수 있는 데이터의 유효 범위입니다. 유효 범위를 초과하여 입력된 경우에는 휘발성 메모리에 복사되는 과정에서 에러가 발생하며, 이 때 휘발성 메모리의 데이터는 유효 범위 내의 값으로 변경됩니다.

- **RW**

  RO(Read Only)는 읽기만 가능하고 쓰기는 할 수 없는 Register를 의미합니다. RO Register에 Write할 경우 에러가 발생합니다. RO Register는 제어기의 모델 명이나 Firmware 버전 같은 고유한 값이거나, 제어기의 센서 정보가 들어있는 항목입니다. RW는 읽기와 쓰기가 모두 가능한 Register입니다.

※ e(Reg_Name) : 비휘발성 Register(EEP Register)의 Reg_Name 항목을 지칭합니다.
※ r(Reg_Name) : 휘발성 Register(RAM Register)의 Reg_Name 항목을 지칭합니다.

Addr	구분	Bytes	Default	Valid Range	RW	Comments
0	Model No1	1	0x05	–	RO	제어기 모델 표시
1	Model No2	1	0x54	–	RO	
2	Version1	1	0x01	–	RO	Firmware version
3	Version2	1	0x01	–	RO	
4	Baud Rate	1	0x10	8page참고	RW	PC–제어기, 제어기–서보 간 통신 속도
5	Reserved	1	0x00	–	–	
6	Reserved	1	0x00	–	–	
7	ID	1	0xFD	0x00~0xFD	RW	제어기 ID(0xFE: Broadcasting ID로 사용 가능, ID 할당은 불가)
8	Ack Policy	1	0x01	0x00~0x02	RW	Policy에 의해 패킷에 응답
9	Torque Off Policy	1	0x2B	0x00~0x7F	RW	Policy에 의해 서보의 Torque를 품
10	Alarm LED Policy	1	0x7F	0x00~0x7F	RW	Policy에 의해 경보 LED 점멸
11	Status Check Policy	1	0x01	0x00~0x01	RW	서보의 각도 값을 체크할 지 여부
12	Min. Voltage	1	0x5F	0x00~0xFE	RW	최소 허용 전압(0x5F : 7.1V)
13	Max. Voltage	1	0x88	0x00~0xFE	RW	최대 허용 전압(0x88 : 10.0V)
14	Max.Temperature	1	0xDF	0x00~0xFE	RW	최대 허용 온도(0xDF : 85 ° C)
15	Remocon Channel	1	0x61	0x61~0x6A	RW	적외선 리모콘의 채널 코드
16	Servo Ack Wait Tick	1	0x0A	0x00~0xFE	RW	Servo Ack를 기다릴 시간 (0x0A : 16ms)
17	Zigbee Ack Wait Tick	1	0x50	0x00~0xFE	RW	Zigbee Ack를 기다릴 시간 (0x50 : 128ms)
18	LED Blink Period	1	0xBB	0x00~0xFE	RW	경보 LED 점멸 주기 (0xBB : 약 300ms)
19	ADC Fault Check Period	2	0x0138	0x0000 ~ 0x7FFF	RW	온도/전압 Error 판정 주기 (0x0138 : 약 500ms)
21	Packet Garbage Check Period	2	0x007D	0x0000 ~0x7FFF	RW	오류 패킷 판정 주기 (0x7D : 약 200ms)

0번부터 6번 주소까지는 제어기의 기본적인 정보와 통신에 관련된 Data이며, 7번부터 22번 주소까지는 제어기의 기능과 관련된 설정 Data입니다. 7번부터 22번 주소까지의 Data는 제어기를 부팅할 때 휘발성 레지스터에 복사됩니다.

Register

HOVIS

휘발성 Register Map

Volatile Register(RAM Register MAP)

휘발성 레지스터는 제어기 동작 설정이나 제어기의 상태, 센서 값을 저장하고 있으며, 휘발성 레지스터에 저장된 동작 설정이 제어기 동작에 직접적인 영향을 미칩니다., 제어기가 꺼졌다 켜지거나 Reboot 되면 초기화 됩니다. 휘발성 레지스터를 변경하여 제어기의 동작 설정을 변경했더라도 제어기가 Reboot되면 비휘발성 레지스터의 값으로 초기화됩니다.

Addr	구분	Bytes	Valid Range	RW	Comments
0	ID	1	0x00~0xFD	RW	
1	Ack Policy	1	0x00~0x02	RW	
2	Torque Off Policy	1	0x00~0x7F	RW	
3	Alarm LED Policy	1	0x00~0x7F	RW	
4	Status Check Policy	1	0x00~0x01	RW	
5	Min. Voltage	1	0x00~0xFE	RW	
6	Max. Voltage	1	0x00~0xFE	RW	비휘발성 레지스터 영역에서 복사되어 비휘발성 레지스터 영역과 동일하게 초기화 됨.
7	Max. Temperature	1	0x00~0xFE	RW	
8	Remocon Channel	1	0x61~0x6A	RW	
9	Servo Ack Wait Tick	1	0x00~0xFE	RW	
10	Zigbee Ack Wait Tick	1	0x00~0xFE	RW	
11	LED Blink Period	1	0x00~0xFE	RW	
12	ADC Fault Check Period	2	0x0000~0x7FFF	RW	
14	Packet Garbage Check Period	2	0x0000~0x7FFF	RW	
16	Status Error	1	0x00~0x7F	RW	상태 에러, 11 페이지 참조
17	Error Codes [0] ~ [4]	5	50 페이지 참조	RW	상태 에러 코드 중 최근 5개
22	LED Control	1	0x00~0x07	RW	Task 실행 시 LED 값 (0x01:Red, 0x02:Green, 0x04: Blue)
23	User Timer Tick	1	0x00~0xFF	RW	사용자 설정 가능한 타이머(100[ms]/tick)
24	Connected Program	1	0~3	RW	현재 연결된 PC 프로그램
25	Reserved	1	–	–	예비용
26	Zigbee PANID	2	0x0000~0xFFFF	RW	Zigbee가 속한 WPAN ID
28	Zigbee SADDR	2	0x0000~0xFFFF	RW	Zigbee의 ID
30	Zigbee DSTADDR	2	0x0000~0xFFFF	RW	상대 Zigbee의 ID
32	Zigbee ACKREQ	1	0~2	RW	RF 통신시 Ack를 요구할 것인지 여부
33	Zigbee BACKOFF	1	0~2	RW	RF 통신시 Random Delay 적용 여부
34	Servo Count	1	0~32	RO	현재 연결된 모터의 수
35	Servo ID[0]~[32]	33	0x00~0xFE	RO	각 모터의 ID (0xFE는 모터가 없다는 것을 의미)

Addr	구분	Bytes	Valid Range	RW	Comments
68	Playing Motion	1	0x00~0x01	RO	Motion이 실행 중인지 여부
69	Playing Task	1	0, 1, 3	RO	Task가 실행 중인지 여부
70	Charger Connected	1	0~1	RO	충전기가 연결 되었는지 여부
71	Buzzer Scale	1	0x00~0x25	RO	버저 음의 음높이
72	Buzzer Time	1	0~192	RO	버저 음의 남은 시간(6.4[ms]/tick)
73	Button Status	1	0x00~0x3F	RO	버튼의 상태
74	Remocon Length	1	0~240	RO	리모콘 버튼이 눌린 시간(125[ms]/tick)
75	Remocon Data	1	0x00~0x1D,0xFE	RO	리모콘 버튼 번호
76	Input Voltage Value	1	0x00~0xFE	RO	입력 전압의 Raw Data, 8bit
77	Temperature Value	1	0x00~0xFE	RO	현재 온도의 Raw Data, 8bit
78	Light Sensor Value	1	0x00~0xFE	RO	빛 센서 값의 Raw Data, 8bit
79	ADC Port 1 Sensor Type	1	0~2	RO	ADC 포트1에 연결된 센서 종류
80	ADC Port 2 Sensor Type	1	0~2	RO	ADC 포트2에 연결된 센서 종류
81	ADC Port 1 Sensor Value	2	0x0000~0xFFFF	RO	ADC 포트1에 연결된 센서 출력 값
83	ADC Port 2 Sensor Value	2	0x0000~0xFFFF	RO	ADC 포트2에 연결된 센서 출력 값
85	ACC/GYRO Connected	1	0~1	RO	가속도/자이로 센서 연결 여부
86	ACC X Value	2	-4096~4095	RO	가속도 센서 X축 Raw Data, 13bit
88	ACC Y Value	2	-4096~4095	RO	가속도 센서 Y축 Raw Data, 13bit
90	ACC Z Value	2	-4096~4095	RO	가속도 센서 Z축 Raw Data, 13bit
92	GYRO X Value	2	-32768~32767	RO	자이로 센서 X축 Raw Data, 16bit
94	GYRO Y Value	2	-32768~32767	RO	자이로 센서 Y축 Raw Data, 16bit
96	GYRO Z Value	2	-32768~32767	RO	자이로 센서 Z축 Raw Data, 16bit
98	Sound Detection Flag	1	0~250	RO	소리감지횟수
99	Sound Direction	1	-2~2	RO	소리를 감지한 방향(-는 왼쪽, +는 오른쪽)
100	Reserved	1	-	-	예비용
101	Tick	2	0~60000	RO	System tick, 1.6[ms]/INT
103	Reserved	1	-	-	예비용
104	Reserved	1	-	-	예비용
105	Reserved	1	-	-	예비용
106	Servo Status Error & Detail [0]~[31]	64	0x00~0x7F * 64	RO	연결된 서보의 상태값
170	Servo Position[0]~[31]	64	-127~1151* 32	RO	연결된 서보의 Position

Register 상세 설명

Model No 1, Model No 2(EEP Register의 0, 1번 Address)

DRC의 모델명을 2바이트의 Binary로 표시한 것입니다. 사용자가 바꿀 수 없습니다.

Version 1, Version 2(EEP Register의 2, 3번 Address)

DRC 내부에 들어가 있는 현재 펌웨어의 버전입니다. 펌웨어 버전이 최신이 아니라면 홈페이지에서 다운로드 받아서 업데이트 하여야 합니다. 사용자가 바꿀 수 없습니다.

Baud Rate(EEP Register의 4번 Address)

PC와 DRC 사이, 그리고 DRC와 DRS 사이의 UART 통신 속도를 설정하는 데이터입니다. 값에 따른 통신 속도는 아래 표와 같습니다. 기본 설정 속도는 115,200 bps이며, 표에 없는 값이 레지스터에 들어 있으면 기본 설정 속도인 115,200 bps로 설정됩니다.

Baud Rate	Register Value
57,600	34
115,200	16
200,000	9
250,000	7
400,000	4
500,000	3
666,667	2

ID(EEP Register의 7번 Address, RAM Register의 0번 Address)

DRC의 ID입니다. 253(0xFD)가 기본 설정이며, 여러 대의 DRC를 서로 ID를 다르게 하면 DRS 여러 개를 제어하듯 같은 통신 라인에 여러 대의 DRC를 연결하여 제어할 수 있습니다. 같은 통신 라인에 서로 ID가 같은 두 대 이상의 DRC가 연결되어 있다면, 통신 시 오동작할 가능성이 큽니다.

ACK Policy(EEP Register의 8번 Address, RAM Register의 1번 Address)

PC에서 DRC로 요청 패킷(Request Packet)을 수신 시, 응답 패킷(ACK Packet)을 보낼지 여부에 대한 설정 Data입니다.

- 0 : 어떠한 요청 패킷에도 응답하지 않습니다.
- 1 : Read Command와 몇몇 응답을 요구하는 패킷에 대해서만 응답합니다.
- 2 : 모든 패킷에 대하여 응답합니다.

※ STAT 요청 패킷에 대해서는 ACK Policy에 무관하게 항상 ACK Packet을 보냅니다.

※ REMOCON 요청 패킷에 대해서는 ACK Policy에 무관하게 항상 응답하지 않습니다.

※ 패킷의 pID가 254(0xFE, Broadcast pID)인 경우 응답하지 않습니다. 단, STAT 명령에는 응답합니다.

※ 각 요청 패킷의 ACK Policy에 따른 응답 여부는 22page의 명령 상세 설명에서 더 자세히 다룹니다.

Torque Off Policy(EEP Register의 9번 Address, RAM Register의 2번 Address)

상태 에러가 발생 시 제어기에 연결된 서보 모터의 토크를 풀 것인지 여부에 대한 정책을 결정합니다.

- (r(Torque Off Policy) & r(Status Error))이 참일 경우 연결된 모든 서보 모터의 토크가 풀려, 서보 모터가 더 이상 움직이지 않게 됩니다.
- r(Status Error)의 에러 상태를 해제해야 다시 서보 모터를 Torque On 상태로 변경할 수 있습니다.

※ &는 Bitwise AND 연산자로, A & B 연산을 수행할 시 A와 B의 각 비트를 비교하여 A, B 모두에서 1인 비트만 결과가 1이 됩니다. 예시) 00101110 & 10110110 의 연산 결과는 00100110이 됩니다.

Alarm LED Policy(EEP Register의 10번 Address, RAM Register의 3번 Address)

상태 에러가 발생 시 제어기의 경보 LED를 점멸할 것인지 여부에 대한 정책을 결정합니다.

- (r(Alarm LED Policy) & r(Status Error))이 참일 경우 제어기의 TX, RX, Spare LED가 점멸하며, 경보 LED의 점멸 주기는 r(LED Blink Period)에 의해 결정됩니다.
- 경보 LED가 점멸 중일 때는 TX, RX, Spare LED의 본래 기능은 무시 됩니다.
- TX, RX, Spare LED가 원래대로 정상 동작하기 위해서는 r(Status Error)의 에러 상태를 해제해야 합니다.

Status Check Policy(EEP Register의 11번 Address, RAM Register의 4번 Address)

제어기가 지속적으로 자신에 연결된 서보의 현재 위치 값을 읽어올지 여부를 설정합니다. r(Status Check Policy)가 1로 설정되어 있으면, 제어기는 자신과 연결된 서보의 위치 값과 서보의 상태를 지속적으로 r(Servo Status Error & Status Detail[0]~[31])과 r(Servo Position[0]~[31])에 업데이트 합니다. r(Status Check Policy)가 0로 설정되어 있으면, 이런 업데이트를 수행하지 않습니다.

Minimum Voltage(EEP Register의 12번 Address, RAM Register의 5번 Address)

최소 입력 전압의 Raw Data를 의미합니다. DRC의 입력 전압 r(Input Voltage Value)가 r(Minimum Voltage) 미만일 경우 r(Status Error)에 “Exceed Input Voltage Limit”(0번 비트)이 설정되며, r(Error Codes[0]~[4])에 0x01(전압이 너무 낮음)이 추가 됩니다.

- Default 설정치는 0x5F(약 7.1V)입니다. 실제 전압 값과의 관계는 환산표(page 48)를 참조 바랍니다.

Maximum Voltage(EEP Register의 13번 Address, RAM Register의 6번 Address)

최대 입력 전압의 Raw Data를 의미합니다. DRC의 입력 전압 r(Input Voltage Value)가 r(Maximum Voltage)를 초과할 경우 r(Status Error)에 “Exceed Input Voltage Limit”(0번 비트)이 설정되며, r(Error Codes[0]~[4])에 0x02(전압이 너무 높음)이 추가 됩니다.

- Default 설정치는 0x88(약 10.0V)입니다. 실제 전압 값과의 관계는 환산표(page 48)를 참조 바랍니다.

Maximum Temperature(EEP Register의 14번 Address, RAM Register의 7번 Address)

최대 동작 온도의 Raw Data를 의미합니다. DRC의 온도 r(Temperature Value)가 r(Maximum Temperature)를 초과할 경우 r(Status Error)에 "Exceed Temperature Limit"(1번 비트)이 설정되며, r(Error Codes[0]~[4])에 0x03(온도가 너무 높음)이 추가 됩니다.

- Default 설정치는 0xDF(약 85℃)입니다. 실제 온도 값과의 관계는 환산표(page 49)를 참조 바랍니다.

Remocon Channel(EEP Register의 15번 Address, RAM Register의 8번 Address)

리모컨의 채널을 의미합니다. 리모컨의 채널 값은 0x61부터 0x6A까지 10개의 채널 설정이 가능하며, 현재 사용하는 리모컨의 채널과 r(Remocon Channel)이 서로 일치해야 리모컨을 인식할 수 있습니다.

Servo Ack Wait Tick(EEP Register의 16번 Address, RAM Register의 9번 Address)

DRC에 연결된 서보에 응답 패킷(ACK Packet)을 필요로 하는 요청 패킷을 보낸 후, 응답 패킷을 받기 까지 기다리는 최대 시간입니다. 최대 시간까지 응답 패킷이 돌아오지 않는다면, 응답이 오지 않은 것으로 판단합니다. 1tick은 1.6ms를 의미하며, Default 설정치는 0x0A(약 16ms)입니다.

Zigbee Ack Wait Tick (EEP Register의 17번 Address, RAM Register의 10번 Address)

DRC에 연결된 Zigbee 모듈에 응답 패킷(ACK Packet)을 필요로 하는 요청 패킷을 보낸 후, 응답 패킷을 받기 까지 기다리는 최대 시간입니다. 최대 시간까지 응답 패킷이 돌아오지 않는다면, 응답이 오지 않은 것으로 판단합니다. 1tick은 1.6ms를 의미하며, Default 설정치는 0x50(약 128ms)입니다.

LED Blink Period(EEP Register의 18번 Address, RAM Register의 11번 Address)

상태 에러 발생 시 r(Alarm LED Policy)에 따라 LED가 점멸할 때, 경보 LED의 점멸 주기를 나타냅니다. LED는 r(LED Blink Period)만큼 켜져 있고, r(LED Blink Period)만큼 꺼져 있는 것을 계속 반복합니다. 1tick은 1.6ms를 의미하며, Default 설정치는 0xBB(약 300ms)입니다.

ADC Fault Check Period(EEP Register의 19번 Address, RAM Register의 12번 Address)

입력 전압과 온도의 오류 판정 주기입니다. 입력 전압과 온도가 허용 범위를 벗어나 r(ADC Fault Check Period)에 해당하는 시간만큼 지속될 경우 오류로 판정합니다. 1tick은 1.6ms를 의미하며, Default 설정치는 0x0138(약 500ms)입니다.

Packet Garbage Check Period(EEP Register의 21번 Address, RAM Register의 14번 Address)

미완성 패킷 판정 주기입니다. 패킷 형식에 어긋나는 데이터를 받았거나, 완성된 패킷을 미처 다 받지 못한 상태가 r(Packet Garbage Check Period)만큼 유지되면 미완성 패킷을 삭제하고 r(Status Error)의 "Invalid Packet"(2번 비트)이 설정합니다. r(Error Codes[0]~[4])에는 패킷을 어디서 받았느냐에 따라서 0x41(Zigbee 모듈의 응답 패킷이 미완성) 혹은 0x51(Servo의 응답 패킷이 미완성) 혹은 0x61(PC측 요청 패킷이 미완성)이 추가 됩니다.

Status Error(RAM Register의 16번 Address)

제어기 상태의 오류 내역을 나타냅니다. 총 7개의 비트로 각 오류의 상태 값을 나타냅니다. r(Alarm LED Policy)와 r(Torque Off Policy)도 아래의 오류 내역과 동일한 구조이며, r(Alarm LED Policy)에서 1로 설정된 비트에 해당하는 오류가 발생 시에는 경보 LED가 점멸하고, r(Torque Off Policy)에서 1로 설정된 비트에 해당하는 오류가 발생 시에는 모든 서보의 토크를 해제합니다.

비트	값	내역
0	0x01	Exceed Input Voltage limit
1	0x02	Exceed Temperature limit
2	0x04	Invalid Packet
3	0x08	Servo Missing
4	0x10	EEP REG distorted
5	0x20	Servo Status Error
6	0x40	Flash Data Distorted
7	0x80	Reserved

Error Codes[0]~[4](RAM Register의 17번 Address)

에러 발생 시 상세 에러 코드를 나타냅니다. 총 5바이트의 공간에 마지막으로 발생한 5가지의 에러가 저장됩니다. 에러가 발생하면 해당하는 에러 코드를 [0]에 저장하고, 기존 [0]~[3]에 저장되어 있던 에러 코드를 [1]~[4]로 한 바이트씩 밀어냅니다. 에러 코드의 상세 내역은 에러 코드 목록(page 50)을 참조 바랍니다.

LED Control(RAM Register의 22번 Address)

Task를 실행하고 있을 때 LED를 제어하는 레지스터입니다. 총 0x00~0x07까지의 값을 가질 수 있으며, 각 비트가 1일 때는 LED를 켜고, 0일 때는 LED를 끕니다. 각 비트가 제어하는 LED는 표와 같습니다. Task 실행 중이 아닐 때에는 LED Control은 의미를 가지지 않으며 항상 0입니다.

비트	값	LED
0	0x01	TX(Red)
1	0x02	RX(Green)
2	0x04	Spare(Blue)

User Timer Tick(RAM Register의 23번 Address)

사용자가 컨트롤할 수 있는 Timer로, 0이 아닌 값을 쓰면 100ms마다 1씩 줄어듭니다. Task 실행 시에 일정 시간만큼 Delay를 주는 용도로 사용합니다.

Connected Program(RAM Register의 24번 Address)

현재 어떤 프로그램으로 PC와 연결되어 통신하고 있는지 알려주는 레지스터 입니다.

- 0 : 프로그램과 연결되지 않았음
- 1 : HerkuleX Manager와 연결된 상태
- 2 : DR-SIM과 연결된 상태
- 3 : DR-Visual Logic과 연결된 상태

Zigbee PANID(RAM Register의 26번 Address)

Zigbee 모듈이 현재 접속해 있는 WPAN(Wireless Personal Area Network)의 ID를 나타냅니다. Zigbee 모듈을 처음 구입하여 DRC와 연결하였을 때 공장 초기값은 0xBADA이며, Zigbee 모듈이 연결되지 않은 상태라면 이 레지스터는 0xFFFF의 값을 가집니다.

Zigbee SADDR(RAM Register의 28번 Address)

Zigbee 모듈은 2바이트의 Short Address와 8바이트의 Long Address를 가집니다. DRC는 그 중 Short Address를 사용합니다. 이 Address는 한 WPAN내에서 각 Zigbee 모듈을 구분할 수 있도록 합니다. Zigbee 모듈을 처음 구입하여 DRC와 연결하였을 때 공장 초기값은 0xBEAD이며, Zigbee 모듈이 연결되지 않은 상태라면 이 레지스터는 0xFFFF의 값을 가집니다.

Zigbee DSTADDR(RAM Register의 30번 Address)

Zigbee 모듈을 통해서 무선으로 WPAN내의 다른 모듈로 패킷을 전송할 때, 패킷을 받을 Zigbee 모듈의 Short Address를 뜻합니다. Zigbee 모듈을 처음 구입하여 DRC와 연결하였을 때 공장 초기값은 0xBEAD이며, Zigbee 모듈이 연결되지 않은 상태라면 이 레지스터는 0xFFFF의 값을 가집니다.

※ 0xFFFF로 설정하고 패킷을 전송하면, Broadcasting이 되어 같은 WPAN 내의 모든 Zigbee 모듈이 패킷을 받게 됩니다.

Zigbee ACKREQ(RAM Register의 32번 Address)

Zigbee 모듈에서 다른 모듈로 무선 신호를 보낼 때, 다른 무선 통신 장비에 의해 방해를 받거나 장해물이 있다면 통신이 원활하게 이루어지지 않을 수도 있습니다. 이 때, 받는 쪽의 모듈이 받은 패킷에 대한 응답 패킷(ACK Packet)을 보내도록 요청한다면, 보내는 입장에서는 응답 패킷을 받지 못할 경우 재전송하면 되므로 통신의 신뢰성을 높일 수 있습니다. 그러나 응답 패킷을 받게 할 경우 신뢰성이 높아지는 대신에 통신에 필요한 시간은 그만큼 증가하기 때문에, 100ms 이하의 짧은 주기로 계속 패킷을 보내야 하는 경우에는 적합하지 않습니다. r(Zigbee ACKREQ)가 1일 경우에는 통신 시에 응답 패킷을 받도록 하고, 0일 경우에는 응답 패킷을 받지 않도록 합니다. Zigbee 모듈에 저장된 공장 초기값은 1이며, Zigbee 모듈이 연결되지 않은 상태라면 이 레지스터는 2의 값을 가집니다.

Zigbee BACKOFF(RAM Register의 33번 Address)

Zigbee 모듈에서 다른 모듈로 무선 신호를 보낼 때, 주변에 WIFI가 있거나 다른 Zigbee 모듈이 통신 중이라면 사용하려는 주파수 대역을 이미 다른 기기가 사용하고 있을 수 있습니다. 이 경우 통신을 할 수 없고 기존의 통신이 끝난 후에야 통신을 할 수 있습니다. r(Zigbee BACKOFF)가 1일 경우에는 이런 경우 랜덤한 시간 동안 기다린 후에 다시 통신을 시도합니다. r(Zigbee ACKREQ)와 마찬가지로, 신뢰성이 높아지는 대신 통신에 필요한 시간은 증가합니다. r(Zigbee BACKOFF)가 0일 경우에는 기다림 없이 바로 다시 통신을 시도합니다. Zigbee 모듈에 저장된 공장 초기값은 1이며, Zigbee 모듈이 연결되지 않은 상태라면 이 레지스터는 2의 값을 가집니다.

Servo Count(RAM Register의 34번 Address)

현재 제어기에 연결된 서로 다른 ID의 서보 모터 개수를 나타냅니다. 최대 32개까지 서보를 연결할 수 있으며, 32개를 넘는 서보 모터가 연결된 경우 r(Status Error)에 “Servo Status Error”(5번 비트)이 설정되며, r(Error Codes[0]~[4])에 0x33(너무 많은 서보가 연결됨)이 추가 됩니다

Servo ID[0]~[32](RAM Register의 35번 Address)

현재 제어기에 연결된 서보 모터의 ID들을 저장하는 33바이트의 공간입니다. Servo ID[0]부터 Servo ID[r(Servo Count)-1]까지 총 r(Servo Count) 바이트에 모터의 ID가 각각 저장되며, 그 외의 공간에는 0xFE(Broadcasting ID)가 저장됩니다. 서보 모터의 최대 개수는 32개이지만, Servo ID[r(Servo Count)]에 0xFE가 저장되는 규칙을 r(Servo Count)가 32일 때에도 만족시키기 위해서 33바이트의 공간을 사용합니다.

Playing Motion(RAM Register의 68번 Address)

현재 DRC에 저장되어 있는 Motion을 실행하고 있는지 여부를 나타내는 Flag입니다. 실행 중일 때에는 1, 그렇지 않을 때에는 0입니다.

Playing Task(RAM Register의 69번 Address)

현재 DRC에 저장되어 있는 Task를 실행하고 있는지 여부를 나타내는 Flag입니다. 실행 중일 때에는 1, 디버깅 모드로 실행 중일 때에는 3, 그렇지 않을 때에는 0입니다.

Charger Connected(RAM Register의 70번 Address)

현재 DRC의 DC잭을 통해 충전기가 연결 되었는지 여부를 나타내는 Flag입니다. DC잭이 연결되었을 때에는 1, 그렇지 않을 때에는 0입니다.

Buzzer Scale(RAM Register의 71번 Address)

현재 실행중인 버저 음의 음 높이를 나타냅니다. 총 3옥타브 대역의 버저 음을 반음 단위로 낼 수 있습니다. 버저가 울리고 있지 않을 때에는 0으로 유지됩니다. 표에서 음 높이 앞의 숫자는 옥타브를 나타냅니다.

값	음 높이	값	음 높이	값	음 높이	값	음 높이
0	쉼표	10	3라	20	4솔	30	5파
1	3도	11	3라#	21	4솔#	31	5파#
2	3도#	12	3시	22	4라	32	5솔
3	3레	13	4도	23	4라#	33	5솔#
4	3레#	14	4도#	24	4시	34	5라
5	3미	15	4레	25	5도	35	5라#
6	3파	16	4레#	26	5도#	36	5시
7	3파#	17	4미	27	5레	37	6도
8	3솔	18	4파	28	5레#		
9	3솔#	19	4파#	29	5미		

Buzzer Time(RAM Register의 72번 Address)

현재 실행중인 버저 음의 남은 시간을 나타냅니다. 1tick은 6.4ms에 해당합니다. 버저 멜로디를 만들거나 Task에서 음표를 실행하거나 하는 경우에 사용 가능한 음 길이의 수는 총 10개가 있습니다. 예를 들어 8분 음표를 실행하면, Buzzer Time에는 24라는 데이터가 쓰여지고 이 값은 6.4ms마다 1씩 감소하여 0이 될 때까지 총 153.6ms만큼 버저가 울리게 됩니다.

Button Status(RAM Register의 73번 Address)

6가지 버튼의 상태를 의미합니다. 버튼 하나당 한 비트로 상태를 나타내며, 눌린 상태가 1, 눌리지 않은 상태가 0입니다. 가령 OK버튼과 Left 버튼이 동시에 눌렸을 경우, r(Button Status)는 0x12가 됩니다.

비트	값	Button
0	0x01	Mode
1	0x02	OK
2	0x04	Up
3	0x08	Down
4	0x10	Left
5	0x20	Right

Remocon Length(RAM Register의 74번 Address)

리모컨 버튼을 얼마나 길게 누르고 있는지를 나타내는 레지스터입니다. 평소에는 0이나, 리모컨 버튼 신호가 들어온 순간부터 약 125ms마다 1씩 증가합니다. 예를 들어, 리모컨 버튼을 3초 동안 누르고 있었다면 r(Remocon Length)의 값은 24가 될 것입니다. r(Remocon Length)의 값은 최대 240까지 이므로, 최대 30초까지 누른 시간을 인식할 수 있습니다.

Remocon Data(RAM Register의 75번 Address)

현재 눌러져 있는 리모컨 버튼의 Key 값입니다. 리모컨 버튼에는 고유의 Key 값이 할당되어 있습니다. 리모컨 신호가 들어오지 않는 때에는 254(0xFE)의 값을 가집니다.

Input Voltage Value(RAM Register의 76번 Address)

입력 전압의 ADC(Anlog-to-Digital Conversion)값, 즉 Raw Data를 나타냅니다. 실제 전압 값과의 관계는 환산표(page 48)를 참조 바랍니다.

Temperature Value(RAM Register의 77번 Address)

현재 온도의 ADC(Anlog-to-Digital Conversion)값, 즉 Raw Data를 나타냅니다. 실제 온도 값과의 관계는 환산표(page 49)를 참조 바랍니다.

Light Sensor Value(RAM Register의 78번 Address)

DRC에 부착되어 있는 조도 센서로 들어오는 빛의 양을 나타냅니다. r(Light Sensor Value)의 크기가 클수록 더 밝은 환경에서 DRC가 동작하고 있다는 뜻입니다.

ADC Port 1 Sensor Type(RAM Register의 79번 Address)

현재 ADC Port 1 로 연결된 센서의 종류를 나타냅니다.

- 0 : 센서가 연결되지 않았음을 뜻합니다.
- 1 : 아날로그 적외선 거리 센서(PSD)가 연결 되었음을 뜻합니다.
- 2 : 디지털 거리 센서가 연결 되었음을 뜻합니다.

ADC Port 2 Sensor Type(RAM Register의 80번 Address)

현재 ADC Port 2로 연결된 센서의 종류를 나타냅니다.

- 0 : 센서가 연결되지 않았음을 뜻합니다.
- 1 : 아날로그 적외선 거리 센서(PSD)가 연결 되었음을 뜻합니다.
- 2 : 디지털 거리 센서가 연결 되었음을 뜻합니다.

ADC Port 1 Sensor Value(RAM Register의 81번 Address)

현재 ADC Port 1로 연결된 센서의 값을 나타냅니다.

- r(ADC Port 1 Sensor Type)이 0일 경우 : 센서가 연결되지 않았으므로 0입니다.
- r(ADC Port 1 Sensor Type)이 1일 경우 : 측정된 거리(cm 단위)가 3~40의 값으로 표시 됩니다.
- r(ADC Port 1 Sensor Type)이 2일 경우 : 디지털 거리 센서의 출력 값이 0이나 1로 표시 됩니다. 10cm보다 물체가 멀리 있으면 1, 가까이 있으면 0의 값이 출력됩니다.

ADC Port 2 Sensor Value(RAM Register의 83번 Address)

현재 ADC Port 2로 연결된 센서의 값을 나타냅니다.

- r(ADC Port 2 Sensor Type)이 0일 경우 : 센서가 연결되지 않았으므로 0입니다.
- r(ADC Port 2 Sensor Type)이 1일 경우 : 측정된 거리(cm 단위)가 3~40의 값으로 표시 됩니다.
- r(ADC Port 2 Sensor Type)이 2일 경우 : 디지털 거리 센서의 출력 값이 0이나 1로 표시 됩니다. 10cm보다 물체가 멀리 있으면 1, 가까이 있으면 0의 값이 출력됩니다.

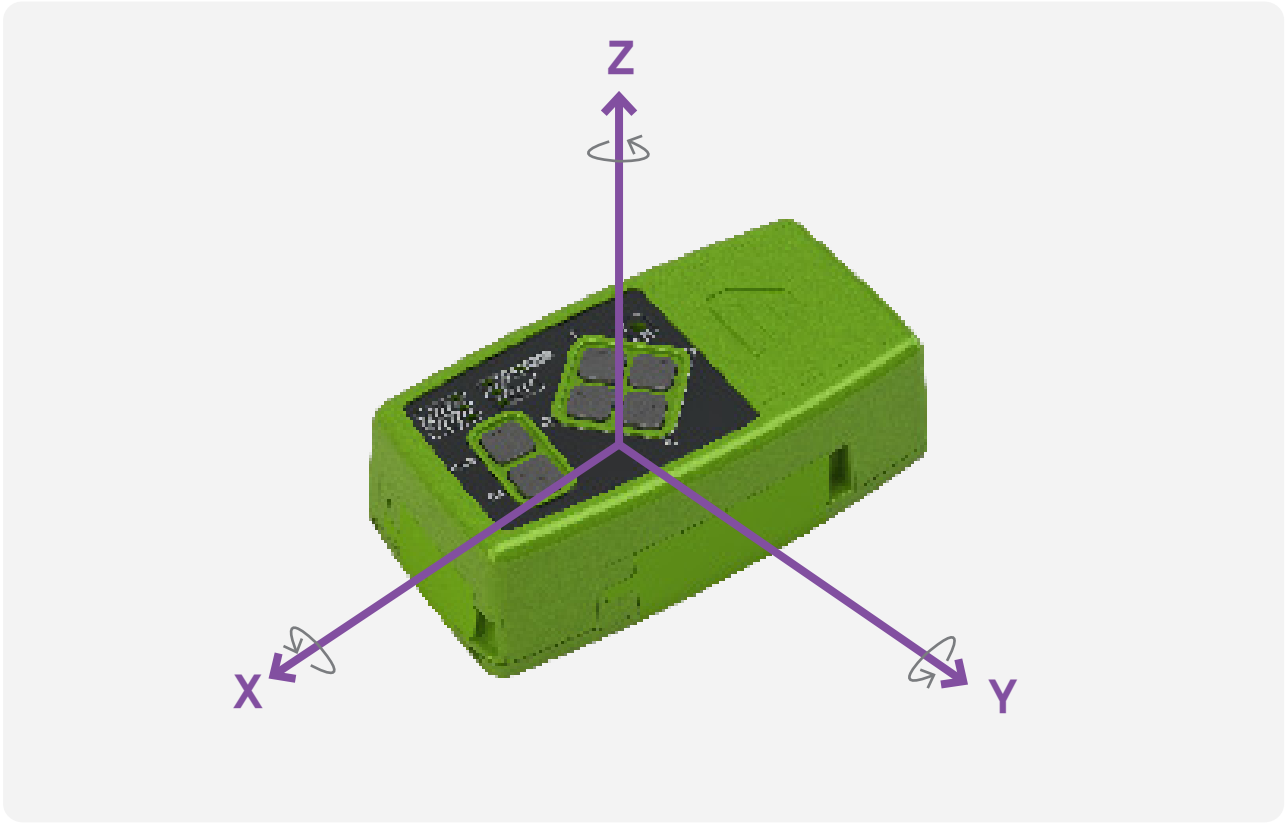
Acc/Gyro Connected(RAM Register의 85번 Address)

가속도/자이로 센서 모듈이 장착되었는지를 나타내는 Flag입니다. 모듈이 장착되었다면 1이 되고, 장착되지 않은 상태라면 0의 값을 가집니다.

Acc X Value, Acc Y Value, Acc Z Value(RAM Register의 86번, 88번, 90번 Address)

가속도 센서의 X, Y, Z축 값의 Raw Data입니다. 가속도 센서는 현재 제어기에 작용하는 가속도 값을 측정합니다. 가속도 센서 각 축의 방향은 아래 그림과 같습니다. 가속도 센서의 각 축은 –4096~4095 의 값을 가질 수 있으며, 256이 1g(중력가속도, 9.8m/s2)에 해당합니다. 따라서 Raw Data를 g의 단위로 바꾸는 계산 식은 아래와 같습니다.

- 가속도(g) = (Raw Data) / 256



Gyro X Value, Gyro Y Value, Gyro Z Value(RAM Register의 92, 94, 96번 Address)

자이로 센서의 X, Y, Z축 값의 Raw Data입니다. 자이로 센서는 현재 제어기가 돌아가는 회전 속도를 측정합니다. 오른손 엄지를 가속도 센서의 축의 방향으로 놓고, 오른손의 나머지 손가락을 접을 때의 방향이 해당 축의 (+) 회전 방향입니다. 다르게 말하면, 축 방향에서 원점을 내려다볼 때 반시계 방향이 (+) 회전 방향입니다. 자이로 센서의 각 축은 –32768~32767의 값을 가질 수 있으며, 약 16.38 이 1°/s(1초당 1° 회전하는 속도)에 해당합니다. 따라서 Raw Data를 °/s의 단위로 바꾸는 계산 식은 아래와 같습니다.

- 각속도(°/s) = (Raw Data) / 32768 X 2000

Sound Detection Flag(RAM Register의 98번 Address)

소리를 감지한 횟수 여부를 나타냅니다. DRC가 소리를 감지하면 r(Sound Detection Flag)가 1증가하고, 그 후 1초내로 다시 소리가 감지되지 않으면 0이 됩니다. 1초내로 또 다시 소리가 감지되었다면, 다시 r(Sound Detection Flag)는 1증가하고, 그 후 1초 동안 소리가 감지되지 않으면 0이 됩니다.

Sound Direction(RAM Register의 99번 Address)

r(Sound Detection Flag)가 1이상 일 때, 가장 늦게 감지한 소리의 방향을 나타내는 레지스터입니다. 값은 -2~2의 범위를 가집니다. 가운데 방향을 기준으로 잡았을 때, 각 값의 의미는 아래와 같습니다.

- -2 : 왼쪽 약 90°에서 소리가 났음을 의미합니다.
- -1 : 왼쪽 약 45°에서 소리가 났음을 의미합니다.
- 0 : 가운데 방향에서 소리가 났음을 의미합니다.
- 1 : 오른쪽 약 45°에서 소리가 났음을 의미합니다.
- 2 : 오른쪽 약 90°에서 소리가 났음을 의미합니다.

r(Sound Detection Flag)가 0일 때는 0의 값을 가집니다.

Tick(RAM Register의 101번 Address)

DRC 전반의 모든 시간에 관련된 작동의 기준이 되는 타이머의 Tick입니다. 0부터 시작하여 60000이 되면 다시 0으로 바뀝니다.

Servo Position[0]~[31](RAM Register의 170번 Address)

현재 연결되어 있는 서보 모터의 위치가 저장되는 2*32바이트짜리 공간입니다. r(Servo ID[n])에 해당하는 ID를 가진 서보의 위치 값이 r(Servo Position[n])에 저장되며, r(Status Check Policy)가 1이라면 실시간으로 계속 업데이트 됩니다. r(Servo Count) 이상의 공간은 0으로 채워집니다.

프로토콜 형식

개요

DRC를 제어하기 위한 패킷은 크게 두 가지로 나뉩니다. 하나는 PC에서 DRC로 보내는 요청 패킷(Request Packet)이며 다른 하나는 DRC에서 PC로 돌려주는 응답 패킷(ACK Packet)입니다.

설정

통신을 하기 위한 설정은 아래와 같습니다.

Baud Rate : 57,600 / 115,200 / 0.2M / 0.25M / 0.4M / 0.5M / 0.667M

Data Bit : 8

Stop Bit : 1

Parity : None

Flow Control : None

※ PC에 장착된 통신 Port나 USB to Serial Cable의 통신 속도는 Hardware 또는 Driver에 의해서 제한될 수 있습니다. 통신이 안 되는 경우 Baud Rate를 확인해야 합니다. DRC의 공장 초기값은 115,200bps입니다.

패킷 구조

구분	Header		Packet Size	pID	CMD	Check Sum1	Check Sum2	Optional Data
value	0xFF	0xFF	7~223	0~0xFE	세부 참조	세부 참조	세부 참조	세부 참조
bytes	1	1	1	1	1	1	1	MAX 216

1. Header(2바이트)

통신 패킷의 시작을 알립니다. 0xFF, 0xFF의 2바이트로 이루어져 있습니다.

2. Packet Size(1바이트)

Header부터 Optional Data까지 Packet의 총 바이트 수를 의미합니다. Packet Size의 최대 값은 223이므로 Packet Size가 223보다 크면 패킷 에러가 발생합니다.

3. pID(1바이트)

제어할 DRC의 ID를 뜻합니다. pID가 254(0xFE)일 경우 패킷을 받은 모든 DRC가 제어 대상이 되므로 사용에 주의해야 합니다. pID가 254보다 크면 패킷 에러가 발생합니다.

※ 레지스터의 r(ID)와 구분하기 위해, 패킷 내부의 ID 영역은 pID로 표기합니다.

4. CMD(1바이트)

요청 패킷에서는 DRC가 수행할 명령, 응답 패킷에서는 DRC가 받은 명령을 의미합니다. PC에서 DRC로보내는 요청 패킷의 명령은 총 14가지가 있습니다. 응답 패킷은 13가지의 명령이 있으며, 요청 패킷과 구별하기 위하여 요청 패킷의 CMD에 0x40을 Bitwise OR연산 합니다. 예를 들어, 요청 패킷의 EEP_WRITE(0x11) CMD에 대한 응답 패킷의 CMD는 0x51이 됩니다. CMD의 목록은 page 20의 Command Set을, 각 CMD에 대한 자세한 설명은 page 22의 명령 상세 설명을 참조하면 됩니다.

이외에도 DRC에 연결된 서보를 향한 요청 패킷 9가지를 보낼 수 있습니다. DRC는 패킷을 검사한 후 서보로 그 패킷을 전달해주며, 서보로부터 응답 패킷이 오면 다시 그것을 PC로 전달해줍니다. 서보의 요청 패킷과 응답 패킷에 대한 내용은 서보 매뉴얼을 참조하시기 바랍니다.

5. Check Sum1, Check Sum2(2바이트)

Check Sum1, 2는 패킷이 손상 없이 전송되었는지 확인하기 위한 2바이트의 공간입니다. Optional Data가 n바이트 있을 때, Check Sum의 계산식은 아래와 같습니다.

Check Sum1 = (Packet Size ^ pID ^ CMD ^ Data[0] ^ ··· ^ Data[n–1]) & 0xFE

Check Sum2 = (~(Packet Size ^ pID ^ CMD ^ Data[0] ^ ··· ^ Data[n–1])) & 0xFE

※ ~는 Bitwise NOT 연산자로, ~A 연산을 수행할 시 A 의 모든 비트를 반전합니다. 예시) ~(01101101)의 연산 결과는 10010010이 됩니다.

※ ^는 Bitwise AND 연산자로, A ^ B 연산을 수행할 시 A와 B의 각 비트를 비교하여 서로 다른 비트만 결과가 1이 됩니다.

예시) 00101110 ^ 10110110 의 연산 결과는 10011000이 됩니다.

6. Optional Data(0~216바이트)

CMD의 종류에 따라서 달라지는 부가적인 데이터입니다. 각 CMD에 따른 Optional Data의 내용은 page 22의 명령 상세 설명을 참조하면 됩니다.

Protocol

HOVIS

Command Set

프로토콜 구조의 CMD부분에 들어가는 Command들의 목록입니다. DRC를 제어하기 위하여 요청 패킷(Request Packet)에는 14가지의 CMD가 존재하며, 이에 응답하는 응답 패킷(ACK Packet)에는 13가지의 CMD가 존재합니다. PC에서 DRC로 요청 패킷을 보내면 DRC는 수신한 패킷에 따른 명령을 수행한 후, 그 결과나 자신의 상태를 응답 패킷의 형태로 PC에 되돌려줍니다. 각 요청 패킷과 응답 패킷의 형태와 형식은 page 22의 명령 상세 설명을 참조하시기 바랍니다.

1. 요청 패킷(PC to DRC)

항목	Cmd	Remark
EEP_WRITE	0x11	EEP Register Address 위치에서 Length개의 값 변경
EEP_READ	0x12	EEP Register Address 위치에서 Length개의 값 요청
RAM_WRITE	0x13	RAM Register Address 위치에서 Length개의 값 변경
RAM_READ	0x14	RAM Register Address 위치에서 Length개의 값 요청
CON_CHECK	0x15	현재 제어기에 어떤 ID의 Servo가 연결되어 있는지 스캔
PLAY_MOTION	0x16	저장되어 있는 Motion을 실행
PLAY_TASK	0x17	저장되어 있는 Task를 실행
PLAY_BUZZ	0x18	저장되어 있는 머리 LED와 Buzzer를 실행
STAT	0x19	제어기의 에러 상태와 가장 최근에 발생한 에러 코드를 요청
ROLLBACK	0x1A	공장 초기화값으로 모든 설정 변수를 변경 변경된 설정은 전원이 꺼진후 다시 기동될때 적용
REBOOT	0x1B	재부팅을 요청
ZIGBEE	0x1C	제어기에 연결된 Zigbee와 관련하여 제어 명령을 전송
REMOCON	0x1D	리모컨 데이터를 보냄
SERVO_FW_UPDATE	0x1E	Servo의 F/W를 업데이트하는 모드에 진입

2. 응답 패킷(DRC to PC)

항목	Cmd	Remark
EEP_WRITE	0x51	r(Status Error)와 r(Status Error Codes[0]) 회신 r(Ack Policy)가 All일 경우에 응답
EEP_READ	0x52	EEP Register Address 위치에서 Len개의 값 회신 r(Ack Policy)가 Read Only, All일 경우에 응답
RAM_WRITE	0x53	r(Status Error)와 r(Status Error Codes[0]) 회신 r(Ack Policy)가 All일 경우에 응답
RAM_READ	0x54	RAM Register Address 위치에서 Len개의 값 회신 r(Ack Policy)가 Read Only, All일 경우에 응답
CON_CHECK	0x55	스캔에 따른 결과로 발견된 서보의 ID들을 회신 r(Ack Policy)가 Read Only, All일 경우에 응답
PLAY_MOTION	0x56	r(Status Error)와 r(Status Error Codes[0]) 회신 r(Ack Policy)가 All일 경우에 응답
PLAY_TASK	0x57	세부 Instruction에 따라서 응답 여부와 형식이 다름(34page 참조)
PLAY_BUZZ	0x58	r(Status Error)와 r(Status Error Codes[0]) 회신 r(Ack Policy)가 All일 경우에 응답
STAT	0x59	r(Status Error)와 r(Status Error Codes[0]) 회신 r(Ack Policy)와 무관하게 항상 응답
ROLLBACK	0x5A	r(Status Error)와 r(Status Error Codes[0]) 회신 r(Ack Policy)가 All일 경우에 응답
REBOOT	0x5B	r(Status Error)와 r(Status Error Codes[0]) 회신 r(Ack Policy)가 All일 경우에 응답
ZIGBEE	0x5C	세부 Instruction에 따라서 응답 여부와 형식이 다름(43page 참조)
REMOCON	–	응답 패킷이 없음.
SERVO_FW_UPDATE	0x5E	r(Status Error)와 r(Status Error Codes[0]) 회신 r(Ack Policy)가 All일 경우에 응답

명령 상세 설명 – EEP_WRITE

1–1. EEP_WRITE – 요청 패킷(0x11)

구분	Packet Size	pID	CMD	Data[0]	Data[1]	Data[2]	···	Data[Length+1]
Value	7+2+Length	0~0xFE	0x11	Address	Length	EEP Data[0]	···	EEP Data[Length–1]

EEP Register의 Address 위치로부터 Length개의 값을 변경합니다. Optional Data에는 Address와 Length, 그리고 Length 개 만큼의 데이터가 들어가므로 Optional Data의 길이는 (2+Length)바이트가 됩니다. 따라서 Packet Size는 기본 7바이트에 (2+Length)바이트를 더해 (9+Length)바이트가 됩니다.
DRC는 이 패킷을 받으면, 비휘발성 레지스터의 주소 Address번부터 (Address+Length–1)번까지의 값을 EEP Data[0]부터 EEP Data[Length–1]의 값으로 변경합니다.

※ DRC의 작동에 직접적으로 영향을 미치는 것은 휘발성 레지스터이므로, EEP_WRITE로 변경한 값은 바로 작동에 반영되지 않습니다. REBOOT CMD를 보내거나 전원을 다시 켜서 다시 부팅 과정을 거쳐야 휘발성 레지스터에 반영됩니다.

예시

■ r(ID)가 253인 DRC의 e(Alarm LED Policy)를 0x3F로 변경하는 패킷

구분	Header		Packet Size	pID	CMD	CS1	CS2	Data[0]	Data[1]	Data[2]
Val–ue	0xFF	0xFF	0x0A(10)	0xFD	0x11	0xD2	0x2C	0x0A	0x01	0x3F

※ CS1, CS2는 각각 Check Sum1, Check Sum2의 약자입니다.
e(Alarm LED Policy)의 주소가 100이고 1바이트짜리 데이터이므로 Address는 10, Length는 1, 그리고 EEP Data[0]은 0x3F가 됩니다. Packet Size는 (9+Length)이므로 100이 되며, page 19의 계산식에 따라 Check Sum1과 Check Sum2가 계산됩니다.

1-2. EEP_WRITE – 응답 패킷(0x51)

구조

구분	Packet Size	pID	CMD	Data[0]	Data[1]
Value	7+2	r(ID)	0x51	r(Status Error)	r(Error Codes[0])

r(Status Error)와 r(Error Codes[0])의 값을 응답 패킷에 실어서 반환합니다. Optional Data의 크기는 2바이트로 고정 되어 있기 때문에 Packet Size는 9바이트로 고정됩니다. pID는 현재 응답하는 DRC의 r(ID)가 실리게 되며, CMD는 요청 패킷 CMD 0x11에 0x40을 Bitwise OR 연산한 0x51이 됩니다.

응답 조건

EEP_WRITE는 r(ACK Policy)가 2(모든 패킷에 대하여 응답)일 경우에만 응답합니다. 단, 요청 패킷의 pID가 254(Broadcasting ID)일 경우에는 r(ACK Policy)가 2라도 응답하지 않습니다.

예시

- r(ID)가 253인 DRC의 e(Alarm LED Policy)를 0x3F로 변경하는 패킷을 받은 후의 응답 패킷

구분	Header		Packet Size	pID	CMD	CS1	CS2	Data[0]	Data[1]
Value	0xFF	0xFF	0x09(9)	0xFD	0x51	0xA4	0x5A	0x00	0x00

현재 상태와 가장 최근의 에러 코드를 실어서 보냅니다. 에러가 없는 상태이므로 둘 다 0x00입니다.

명령 상세 설명 – EEP_READ

2-1. EEP_READ – 요청 패킷(0x12)

구조

구분	Packet Size	pID	CMD	Data[0]	Data[1]
Value	7+2	0~0xFE	0x12	Address	Length

EEP Register의 Address 위치로부터 Length개의 값을 읽습니다. Optional Data에는 Address와 Length가 들어가므로 Optional Data의 길이는 2바이트가 됩니다. 따라서 Packet Size는 기본 7바이트에 2바이트를 더해 9바이트가 됩니다.
DRC는 이 패킷을 받으면, 비휘발성 레지스터의 주소 Address번부터 (Address+Length–1)번까지의 값을 응답 패킷을 통해서 보내줍니다.

예시

- r(ID)가 253인 DRC의 e(Min Voltage), e(Max Voltage), e(Max Temperature)를 읽어오는 패킷

구분	Header		Packet Size	pID	CMD	CS1	CS2	Data[0]	Data[1]
Value	0xFF	0xFF	0x09(9)	0xFD	0x12	0xE8	0x16	0x0C	0x03

e(Min Voltage)의 주소가 120이고 그 이후로 3바이트를 읽어야 하므로 Address는 12, Length는 30이 됩니다. Packet Size는 90이며, page 19의 계산식에 따라 Check Sum1과 Check Sum2가 계산됩니다.

2-2. EEP_READ – 응답 패킷(0x52)

구조

구분	Packet Size	pID	CMD	Data[0]	Data[1]
Value	7+2+Length+2	r(ID)	0x52	Address	Length

구분	Data[2]	···	Data[Length+1]	Data[Length+2]	Data[Length+3]
Value	EEP Data[0]	···	EEP Data[Length-1]	r(Status Error)	r(Error Codes[0])

비휘발성 레지스터의 주소 Address번부터 (Address+Length-1)번까지의 값을 EEP Data[0]부터 EEP Data[Length-1]에 실어서 보냅니다. 또한 r(Status Error)와 r(Error Codes[0])의 값도 반환합니다. Optional Data에는 Address와 Length, 그리고 Length 개 만큼의 데이터, 또한 r(Status Error)와 r(Error Codes[0])이 들어가므로 Optional Data의 길이는 (2+Length+2)바이트가 됩니다. 따라서 Packet Size는 기본 7바이트에 (4+Length)바이트를 더해 (11+Length)바이트가 됩니다. pID는 현재 응답하는 DRC의 r(ID)가 실리게 되며, CMD는 요청 패킷 CMD 0x12에 0x40을 Bitwise OR 연산한 0x52이 됩니다.

응답 조건

EEP_READ는 r(ACK Policy)가 1(Read 명령일 때만 응답), 2(모든 패킷에 대하여 응답)일 경우에 응답합니다. 단, 요청 패킷의 pID가 254(Broadcasting ID)일 경우에는 응답하지 않습니다.

예시

■ r(ID)가 253인 DRC의 e(Min Voltage), e(Max Voltage), e(Max Temperature)를 읽어오는 패킷에 대한 응답 패킷

구분	Header		Packet Size	pID	CMD	CS1	CS2	Data[0]	Data[1]
Value	0xFF	0xFF	0x0E(14)	0xFD	0x52	0xA6	0x58	0x0C	0x03

구분	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]
Value	0x5F	0x88	0xDF	0x00	0x00

Address 12번부터 3바이트의 데이터를 Data[2]~Data[4]에 실어서 보냅니다. Data[2]에는 e(Min Voltage), Data[3]에는 e(Max Voltage), Data[4]에는 e(Max Temperature)가 각각 실려 있습니다. 그리고 Data[5]와 Data[6]에 현재 상태와 가장 최근의 에러 코드를 실어서 보냅니다. 에러가 없는 상태이므로 둘 다 0x00입니다.

명령 상세 설명 – RAM_WRITE

3-1. RAM_WRITE – 요청 패킷(0x13)

구조

구분	Packet Size	pID	CMD	Data[0]	Data[1]	Data[2]	…	Data[Length+1]
Value	7+2+Length	0~0xFE	0x13	Address	Length	RAM Data[0]	…	RAM Data[Length–1]

RAM Register의 Address 위치로부터 Length개의 값을 변경합니다. Optional Data에는 Address와 Length, 그리고 Length 개 만큼의 데이터가 들어가므로 Optional Data의 길이는 (2+Length)바이트가 됩니다. 따라서 Packet Size는 기본 7바이트에 (2+Length)바이트를 더해 (9+Length)바이트가 됩니다.
DRC는 이 패킷을 받으면, 휘발성 레지스터의 주소 Address번부터 (Address+Length–1)번까지의 값을 RAM Data[0]부터 RAM Data[Length–1]의 값으로 변경합니다.

예시

■ r(ID)가 253인 DRC의 r(Status Error)와 r(Error Codes[0]~[4])를 모두 0x00로 변경하는 패킷

구분	Header		Packet Size	pID	CMD	CS1	CS2	Data[0]	Data[1]
Value	0xFF	0xFF	0x0F(15)	0xFD	0x13	0xF6	0x08	0x10	0x06

구분	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]	Data[7]
Value	0x00	0x00	0x00	0x00	0x00	0x00

r(Status Error)의 주소가 160이고, 그 이후로 6바이트의 값을 변경해야 하므로 Address는 16, Length는 6, 그리고 RAM Data[0]~RAM Data[5]는 0x000이 됩니다. Packet Size는 (9+Length)이므로 15가 되며, page 19의 계산식에 따라 Check Sum1과 Check Sum2가 계산됩니다.

※ r(Status Error)와 r(Error Codes[0]~[4])는 쓰고 읽기가 모두 가능한 레지스터이지만, MPSU의 현재 상태를 나타내기 때문에 임의로 값을 수정할 수 없습니다. 이 6바이트의 레지스터에는 현재 값을 그대로 쓰거나, 모두 0x00을 쓰거나 둘 중 하나만 할 수 있습니다. 그렇지 않으면 r(Status Error)에 “Invalid Packet”(2번 비트)이 설정되며, r(Error Codes[0]~[4])에 0x73(r(Status Error)와 r(Error Codes[0]~[4])에 잘못된 쓰기 명령)가 추가됩니다.

3-2. RAM_WRITE – 응답 패킷(0x53)

구조

구분	Packet Size	pID	CMD	Data[0]	Data[1]
Value	7+2	r(ID)	0x53	r(Status Error)	r(Error Codes[0])

r(Status Error)와 r(Error Codes[0])의 값을 응답 패킷에 실어서 반환합니다. Optional Data의 크기는 2바이트로 고정 되어 있기 때문에 Packet Size는 9바이트로 고정됩니다. pID는 현재 응답하는 DRC의 r(ID)가 실리게 되며, CMD는 요청 패킷 CMD 0x13에 Bitwise OR 연산한 0x53이 됩니다.

응답 조건

RAM_WRITE는 r(ACK Policy)가 2(모든 패킷에 대하여 응답)일 경우에만 응답합니다. 단, 요청 패킷의 pID가 254(Broadcasting ID)일 경우에는 r(ACK Policy)가 2라도 응답하지 않습니다.

예시

- r(ID)가 253인 DRC의 r(Status Error)와 r(Error Codes[0]~[4])를 모두 0x00로 변경하는 패킷

구분	Header		Packet Size	pID	CMD	CS1	CS2	Data[0]	Data[1]
Value	0xFF	0xFF	0x09(9)	0xFD	0x53	0xA6	0x58	0x00	0x00

현재 상태와 가장 최근의 에러 코드를 실어서 보냅니다. 에러가 없는 상태이므로 둘 다 0x00입니다.

명령 상세 설명 – EEP_READ

4-1. RAM_READ – 요청 패킷(0x14)

구조

구분	Packet Size	pID	CMD	Data[0]	Data[1]
Value	7+2	0~0xFE	0x14	Address	Length

RAM Register의 Address 위치로부터 Length개의 값을 읽습니다. Optional Data에는 Address와 Length가 들어가므로 Optional Data의 길이는 2바이트가 됩니다. 따라서 Packet Size는 기본 7바이트에 2바이트를 더해 9바이트가 됩니다.
DRC는 이 패킷을 받으면, 휘발성 레지스터의 주소 Address번부터 (Address+Length-1)번까지의 값을 응답 패킷을 통해서 보내줍니다.

예시

- r(ID)가 253인 DRC의 r(Min Voltage), r(Max Voltage), r(Max Temperature)를 읽어오는 패킷

구분	Header		Packet Size	pID	CMD	CS1	CS2	Data[0]	Data[1]
Value	0xFF	0xFF	0x09(9)	0xFD	0x14	0xE6	0x18	0x05	0x03

r(Min Voltage)의 주소가 50이고 그 이후로 3바이트를 읽어야 하므로 Address는 5, Length는 30이 됩니다. Packet Size는 90이며, page 00의 계산식에 따라 Check Sum1과 Check Sum2가 계산됩니다.

4-2. RAM_READ – 응답 패킷(0x54)

구조

구분	Packet Size	pID	CMD	Data[0]	Data[1]
Value	7+2+Length+2	r(ID)	0x54	Address	Length

구분	Data[2]	···	Data[Length+1]	Data[Length+2]	Data[Length+3]
Value	RAM Data[0]	···	RAM Data[Length-1]	r(Status Error)	r(Error Codes[0])

휘발성 레지스터의 주소 Address번부터 (Address+Length-1)번까지의 값을 RAM Data[0]부터 RAM Data[Length-1]에 실어서 보냅니다. 또한 r(Status Error)와 r(Error Codes[0])의 값도 반환합니다. Optional Data에는 Address와 Length, 그리고 Length 개 만큼의 데이터, 또한 r(Status Error)와 r(Error Codes[0])이 들어가므로 Optional Data의 길이는 (2+Length+2)바이트가 됩니다. 따라서 Packet Size는 기본 7바이트에 (4+Length)바이트를 더해 (11+Length)바이트가 됩니다. pID는 현재 응답하는 DRC의 r(ID)가 실리게 되며, CMD는 요청 패킷 CMD 0x14에 0x40을 Bitwise OR 연산한 0x54이 됩니다.

응답 조건

RAM_WRITE는 r(ACK Policy)가 2(모든 패킷에 대하여 응답)일 경우에만 응답합니다. 단, 요청 패킷의 pID가 254(Broadcasting ID)일 경우에는 r(ACK Policy)가 2라도 응답하지 않습니다.

예시

- r(ID)가 253인 DRC의 r(Min Voltage), r(Max Voltage), r(Max Temperature)를 읽어오는 패킷에 대한 응답 패킷

구분	Header		Packet Size	pID	CMD	CS1	CS2	Data[0]	Data[1]
Value	0xFF	0xFF	0x09(9)	0xFD	0x54	0xA8	0x56	0x05	0x03

구분	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]
Value	0x5F	0x88	0xDF	0x00	0x00

Address 5번부터 3바이트의 데이터를 Data[2]~Data[4]에 실어서 보냅니다. Data[2]에는 r(Min Voltage), Data[3]에는 r(Max Voltage), Data[4]에는 r(Max Temperature)가 각각 실려 있습니다. 그리고 Data[5]와 Data[6]에 현재 상태와 가장 최근의 에러 코드를 실어서 보냅니다. 에러가 없는 상태이므로 둘 다 0x00입니다.

명령 상세 설명 – CON_CHECK

5–1. CON_CHECK – 요청 패킷(0x15)

구조

구분	Packet Size	pID	CMD	Data[0]	Data[1]	···	Data[Length]
Value	7+1+Length	0~0xFE	0x15	Length	ID[0]	···	ID[Length–1]

현재 DRC에 ID[0]~ID[Length–1]의 ID를 가지는 서보가 연결되어 있는지 확인합니다. Optional Data에는 Length와 Length개 만큼의 ID가 들어가므로 Optional Data의 길이는 (1+Length)바이트가 됩니다. 따라서 Packet Size는 기본 7바이트에 (1+Length)바이트를 더해 (8+Length)바이트입니다.
DRC는 이 패킷을 받으면, ID[0]부터 ID[Length–1]까지 서보들에 통신을 시도하고, 이 중에서 실제로 연결되어 통신에 성공한 개수와 ID를 응답 패킷을 통해 보내줍니다.

※ Length가 0인 경우에는 0~253까지 모든 ID를 스캔합니다.

예시

■ r(ID)가 253인 DRC에 ID가 0, 1, 2, 3, 4인 서보가 연결되었는지 확인하는 패킷

구분	Header		Packet Size	pID	CMD	CS1	CS2	Data[0]
Value	0xFF	0xFF	0x0D(13)	0xFD	0x15	0xE4	0x1A	0x05

구분	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]
Value	0x00	0x01	0x02	0x03	0x04

서보의 개수가 5개이므로 Length는 5가 되고, Data[1]부터 Data[5]에 각각 검사할 ID인 0, 1, 2, 3, 4가 각각 들어갑니다. Packet Size는 (8+Length)이므로 13이 되며, page 19의 계산식에 따라 Check Sum1과 Check Sum2가 계산됩니다.

5-2. CON_CHECK – 응답 패킷(0x55)

구조

구분	Packet Size	pID	CMD	Data[0]
Value	7+1+Length+2	r(ID)	0x55	Length

구분	Data[1]	···	Data[Length]	Data[Length+1]	Data[Length+2]
Value	ID[0]	···	ID[Length-1]	r(Status Error)	r(Error Codes[0])

요청 패킷을 받아서 ID스캔을 수행한 후, 실제로 통신이 되었던 서보의 개수를 Length에, 그 ID값들을 ID[0]부터 ID[Length-1]에 실어서 보냅니다. 또한 r(Status Error)와 r(Error Codes[0])의 값도 반환합니다. Optional Data에는 Length, 그리고 Length개 만큼의 ID, 또한 r(Status Error)와 r(Error Codes[0])이 들어가므로 Optional Data의 길이는 (1+Length+2)바이트가 됩니다. 따라서 Packet Size는 기본 7바이트에 (3+Length)바이트를 더해 (10+Length)바이트가 됩니다. pID는 현재 응답하는 DRC의 r(ID)가 실리게 되며, CMD는 요청 패킷 CMD 0x15에 0x40을 Bitwise OR 연산한 0x55이 됩니다.

응답 조건

CON_CHECK는 r(ACK Policy)가 1(Read 명령일 때만 응답), 2(모든 패킷에 대하여 응답)일 경우에 응답합니다. 단, 요청 패킷의 pID가 254(Broadcasting ID)일 경우에는 응답하지 않습니다.

예시

■ r(ID)가 253인 DRC에 ID가 0, 1, 2, 3, 4인 서보가 연결되었는지 확인하는 패킷을 받았는데, 검사 결과 실제로는 0, 1, 2만 연결되어 있는 경우의 응답 패킷

구분	Header		Packet Size	pID	CMD	CS1	CS2	Data[0]
Value	0xFF	0xFF	0x0D(13)	0xFD	0x55	0xA4	0x5A	0x03

구분	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]
Value	0x00	0x01	0x02	0x00	0x00

ID 스캔 결과 0, 1, 2만 연결되어 있으므로, Length를 나타내는 Data[0]는 3이 되며 Data[1]~Data[3]에 0, 1, 2가 차례로 들어갑니다. 그리고 Data[4]와 Data[5]에 현재 상태와 가장 최근의 에러 코드를 실어서 보냅니다. 에러가 없는 상태이므로 둘 다 0x00입니다. Packet Size는 (10+Length)이므로 13이 됩니다.

명령 상세 설명 – PLAY_MOTION

6-1. PLAY_MOTION – 요청 패킷(0x16)

구조

구분	Packet Size	pID	CMD	Data[0]	Data[1]
Value	7+2	0~0xFE	0x16	Motion No.	Motion Ready Flag

현DR-SIM을 사용하여 DRC에 모션을 저장하면, 모션은 0부터 127까지의 번호를 가지게 됩니다. PLAY_MOTION 패킷은 현재 DRC에 저장된 모션을 실행하며, Motion No.는 실행할 모션의 번호를 나타냅니다. Motion Ready Flag는 모션 준비 자세를 취할지 여부를 결정합니다. Motion Ready Flag를 1로 해서 패킷을 보내면, 해당 모션의 첫 프레임만을 느리게 실행하게 됩니다. 실행하려는 모션을 Motion Ready Flag를 1로 해서 한번 요청한 후, 조금 기다렸다가 0으로 해서 다시 요청하면, 걷기 등의 모션을 실행할 때 급격한 자세 변화로 인한 넘어짐과 모터 손상을 예방할 수 있습니다. 또한, 모션 번호를 254(0xFE)로 해서 패킷을 보내면 현재 실행중인 모션이 중지됩니다.

Motion No.와 Motion Ready Flag에 따른 동작을 정리하면 아래와 같습니다.

Motion No	Motion Ready Flag	동작
0~127	0	모션 실행
0~127	1	모션의 첫 프레임만 실행
254	0~1	모션 정지

예시

■ r(ID)가 253인 DRC에서 모션 1번을 실행하는 패킷

구분	Header		Packet Size	pID	CMD	CS1	CS2	Data[0]	Data[1]	Data[1]
Value	0xFF	0xFF	0x09(9)	0xFD	0x16	0xE2	0x1C	0x01	0x00	0x00

모션 번호가 1번이므로 Motion No.에는 1이 들어가며 Motion Ready Flag은 0입니다.

■ r(ID)가 253인 DRC에서 모션 2번을 첫 프레임만 실행하는 패킷

구분	Header		Packet Size	pID	CMD	CS1	CS2	Data[0]	Data[1]
Value	0xFF	0xFF	0x09(9)	0xFD	0x16	0xE0	0x1E	0x02	0x01

모션 번호가 2번이므로 Motion No.에는 2가 들어가며 Motion Ready Flag은 1입니다.

6-2. PLAY_MOTION – 응답 패킷(0x56)

구조

구분	Packet Size	pID	CMD	Data[0]	Data[1]
Value	7+2	r(ID)	0x56	r(Status Error)	r(Error Codes[0])

r(Status Error)와 r(Error Codes[0])의 값을 응답 패킷에 실어서 반환합니다. Optional Data의 크기는 2바이트로 고정 되어 있기 때문에 Packet Size는 9바이트로 고정됩니다. pID는 현재 응답하는 DRC의 r(ID)가 실리게 되며, CMD는 요청 패킷 CMD 0x16에 Bitwise OR 연산한 0x560이 됩니다

응답 조건

PLAY_MOTION은 r(ACK Policy)가 2(모든 패킷에 대하여 응답)일 경우에만 응답합니다. 단, 요청 패킷의 pID가 254(Broadcasting ID)일 경우에는 r(ACK Policy)가 2라도 응답하지 않습니다.

예시

- r(ID)가 253인 DRC의 2번 모션을 실행하는 것에 대한 응답 패킷

구분	Header		Packet Size	pID	CMD	CS1	CS2	Data[0]	Data[1]
Value	0xFF	0xFF	0x09(9)	0xFD	0x56	0xA2	0x5C	0x00	0x00

현재 상태와 가장 최근의 에러 코드를 실어서 보냅니다. 에러가 없는 상태이므로 둘 다 0x00입니다.

명령 상세 설명 – PLAY_TASK

7-1. PLAY_TASK – 요청 패킷(0x17)

구조

구분	Packet Size	pID	CMD	Data[0]	Data[1]
Value	7+1	0~0xFE	0x17	Instruction	Motion Ready Flag

DR-Visual Logic을 사용하여 DRC에 저장한 Task를 실행합니다. PLAY_TASK는 Instruction에 따라서 4개의 명령으로 나뉘는데, Instruction에 따라서 수행하는 일이 다릅니다.

- Instruction이 0일 때에는 일반 모드로 Task를 실행합니다.
- Instruction이 1일 때에는 디버깅 모드로 Task를 실행합니다.
- Instruction이 2일 때에는 Task를 한 Step만 진행한 후 멈춥니다. 이 Instruction은 디버깅 모드일 때에만 의미가 있습니다.
- Instruction이 254일 때에는 Task를 정지합니다. 일반 모드와 디버깅 모드에 관계 없이 Task가 정지됩니다.

예시

- r(ID)가 253인 DRC에서 저장된 Task를 실행하는 패킷

구분	Header		Packet Size	pID	CMD	CS1	CS2	Data[0]
Value	0xFF	0xFF	0x08(8)	0xFD	0x17	0xE2	0x1C	0x00

Task 실행이므로 Instruction은 0입니다.

- r(ID)가 253인 DRC에서 저장된 Task를 디버깅 모드로 실행하는 패킷

구분	Header		Packet Size	pID	CMD	CS1	CS2	Data[0]
Value	0xFF	0xFF	0x08(8)	0xFD	0x17	0xE2	0x1C	0x01

디버깅 모드에서 Task 실행이므로 Instruction은 1입니다.

- r(ID)가 253인 DRC가 디버깅 모드일 때, Task를 한 Step 진행하는 패킷

구분	Header		Packet Size	pID	CMD	CS1	CS2	Data[0]
Value	0xFF	0xFF	0x08(8)	0xFD	0x17	0xE0	0x1E	0x02

디버깅 모드 중에 Task를 한 Step 진행하므로 Instruction은 2입니다.

7-2. PLAY_TASK – 응답 패킷(0x57)

구조 – 디버깅 응답 패킷

구분	Packet Size	pID	CMD	Data[0]	Data[1]	Data[2]	Data[3]
Value	7+4	r(ID)	0x57	Program Counter L	Program Counter H	r(Status Error)	r(Error Codes[0])

구조 – 상태 응답 패킷

구분	Packet Size	pID	CMD	Data[0]	Data[1]
Value	7+2	r(ID)	0x57	r(Status Error)	r(Error Codes[0])

PLAY_TASK의 응답 패킷은 Instruction에 따라서 두 종류로 나뉩니다.
디버깅 응답 패킷은 현재 Task의 어떤 부분을 실행하고 있는지 Program Counter L과 Program Counter H의 두 바이트를 통해서 알려줍니다. 이 정보를 바탕으로 DR-Visual Logic에서 Task를 디버깅할 때 현재 어느 위치의 코드가 실행 중인지 알 수 있습니다. 디버깅 응답 패킷에는 추가로 r(Status Error)와 r(Error Codes[0])의 값도 포함됩니다. Optional Data의 크기가 4바이트로 고정되기 때문에 Packet Size는 11바이트입니다. pID는 현재 응답하는 DRC의 r(ID)가 실리게 되며, CMD는 요청 패킷 CMD 0x17에 Bitwise OR 연산한 0x570이 됩니다.
상태 응답 패킷은 r(Status Error)와 r(Error Codes[0])의 값을 응답 패킷에 실어서 반환합니다. Optional Data의 크기는 2바이트로 고정 되어 있기 때문에 Packet Size는 9바이트로 고정됩니다.
디버깅 응답 패킷은 디버깅에 관련된 상황일 때, 그리고 상태 응답 패킷은 나머지 상황일 때 사용됩니다. Instruction에 따라서 어떤 응답 패킷을 보내는지는 아래와 같습니다.

Instruction	성공 시	실패 시
0	상태	상태
1	디버깅	상태
2	디버깅	상태
254	상태	상태

Task 실행(Instruction 0)이나 Task 중지(Instruction 254)같은 경우에는 상태 응답 패킷으로 응답합니다. 디버깅 모드로 Task 실행(Instruction 1)이나 한 Step 진행 후 멈춤(Instruction 2)과 같이 디버깅과 관련된 Instruction에는 디버깅 응답 패킷으로 응답합니다. 그러나 이미 Task가 실행 중인데 Instruction 1을 보냈다거나, Task가 실행 중이 아닌데 Instruction 2를 보냈다거나 등과 같이 명령이 제대로 수행되지 않은 경우에는 상태 응답 패킷으로 응답합니다.

응답 조건

상태 응답 패킷을 보내야 할 경우에는 r(ACK Policy)가 2(모든 패킷에 대하여 응답)일 경우에만 응답합니다. 단, 요청 패킷의 pID가 254(Broadcasting ID)일 경우에는 r(ACK Policy)가 2라도 응답하지 않습니다. 디버깅 응답 패킷을 보내야 할 경우에는 r(ACK Policy)가 1(Read 명령일 때만 응답), 2(모든 패킷에 대하여 응답)일 경우에 응답합니다. 단, 요청 패킷의 pID가 254(Broadcasting ID)일 경우에는 응답하지 않습니다.

예시

■ r(ID)가 253인 DRC에서 저장된 Task를 실행하는 패킷

구분	Header		Packet Size	pID	CMD	CS1	CS2	Data[0]	Data[1]
Value	0xFF	0xFF	0x09(9)	0xFD	0x57	0xA2	0x5C	0x00	0x00

요청 패킷의 Instruction이 0이므로, 현재 상태와 가장 최근의 에러 코드를 실어서 보냅니다. 에러가 없는 상태이므로 둘 다 0x00입니다.

■ r(ID)가 253인 DRC에서 저장된 Task를 디버깅 모드로 실행하는 패킷

구분	Header		Packet Size	pID	CMD	CS1	CS2	Data[0]	Data[1]	Data[2]	Data[3]
Value	0xFF	0xFF	0x0B(11)	0xFD	0x57	0xAA	0x54	0x0B	0x00	0x00	0x00

요청 패킷의 Instruction이 1이므로, 디버깅 응답 패킷을 보냅니다. Data[0]과 Data[1]에는 현재 프로그램 카운터의 위치가 들어가 있습니다. 디버깅을 시작한 상태에서 현재 위치는 0x000B입니다. Data[2]와 Data[3]은 현재 상태와 가장 최근의 에러 코드를 나타냅니다.

■ r(ID)가 253인 DRC가 디버깅 모드일 때, Task를 한 Step 진행하는 패킷

구분	Header		Packet Size	pID	CMD	CS1	CS2	Data[0]	Data[1]	Data[2]	Data[3]
Value	0xFF	0xFF	0x0B(11)	0xFD	0x57	0x86	0x78	0x26	0x00	0x00	0x00

요청 패킷의 Instruction이 2이므로, 디버깅 응답 패킷을 보냅니다. Data[0]과 Data[1]를 보면 프로그램 카운터의 현재 위치는 0x0026임을 알 수 있습니다. Data[2]와 Data[3]은 현재 상태와 가장 최근의 에러 코드를 나타냅니다.

명령 상세 설명 – PLAY_BUZZ

8-1. PLAY_BUZZ – 요청 패킷(0x18)

구조

구분	Packet Size	pID	CMD	Data[0]	Data[1]
Value	7+2	0~0xFE	0x18	Reserved	Buzz No.

DRC에 저장된 Buzzer를 실행합니다. Buzzer의 번호는 1부터 63까지가 가능하며, Data[1]의 Buzz No.에 실행할 버저의 번호를 담아서 패킷을 보내게 됩니다. Data[0]은 다른 데이터가 들어가기 위해서 Reserve 되어 있는 공간이므로 무조건 0을 넣으면 됩니다.

예시

■ r(ID)가 253인 DRC에서 5번 버저를 실행하는 패킷

구분	Header		Packet Size	pID	CMD	CS1	CS2	Data[0]	Data[1]
Value	0xFF	0xFF	0x09(9)	0xFD	0x18	0xE8	0x16	0x00	0x05

5번 버저를 실행해야 하므로 Data[1]에는 5가 들어갑니다.

8-2. PLAY_BUZZ – 응답 패킷(0x58)

구조

구분	Packet Size	pID	CMD	Data[0]	Data[1]
Value	7+2	r(ID)	0x58	r(Status Error)	r(Error Codes[0])

r(Status Error)와 r(Error Codes[0])의 값을 응답 패킷에 실어서 반환합니다. Optional Data의 크기는 2바이트로 고정 되어 있기 때문에 Packet Size는 9바이트로 고정됩니다. pID는 현재 응답하는 DRC의 r(ID)가 실리게 되며, CMD는 요청 패킷 CMD 0x18에 Bitwise OR 연산한 0x58이 됩니다.

응답 조건

PLAY_Buzz은 r(ACK Policy)가 2(모든 패킷에 대하여 응답)일 경우에만 응답합니다. 단, 요청 패킷의 pID가 254(Broadcasting ID)일 경우에는 r(ACK Policy)가 2라도 응답하지 않습니다.

예시

- r(ID)가 253인 DRC에 보낸 5번 버저 실행 요청 패킷에 대한 응답 패킷

구분	Header		Packet Size	pID	CMD	CS1	CS2	Data[0]	Data[1]
Value	0xFF	0xFF	0x09(9)	0xFD	0x58	0xAC	0x52	0x00	0x00

현재 상태와 가장 최근의 에러 코드를 실어서 보냅니다. 에러가 없는 상태이므로 둘 다 0x00입니다.

명령 상세 설명 – STAT

9–1. STAT – 요청 패킷(0x19)

구조

구분	Packet Size	pID	CMD
Value	7+2	0~0xFE	0x18

DRC의 현재 상태를 요청합니다. DRC는 이 패킷을 받으면 r(Status Error)와 r(Error Codes[0])의 값을 응답 패킷에 실어서 반환합니다.

예시

■ r(ID)가 253인 DRC에 STAT 명령을 요청하는 패킷

구분	Header		Packet Size	pID	CMD	CS1	CS2
Value	0xFF	0xFF	0x07(7)	0xFD	0x19	0xE2	0x1C

9–2. STAT – 응답 패킷(0x59)

구조

구분	Packet Size	pID	CMD	Data[0]	Data[1]
Value	7+2	r(ID)	0x59	r(Status Er–ror)	r(Error Codes[0])

r(Status Error)와 r(Error Codes[0])의 값을 응답 패킷에 실어서 반환합니다. Optional Data의 크기는 2바이트로 고정 되어 있기 때문에 Packet Size는 9바이트로 고정됩니다. pID는 현재 응답하는 DRC의 r(ID)가 실리게 되며, CMD는 요청 패킷 CMD 0x19에 Bitwise OR 연산한 0x59이 됩니다.

응답 조건

STAT은 r(ACK Policy)와 관계 없이 모든 경우에 응답합니다. 요청 패킷의 pID가 254(Broadcasting ID)일 경우에도 응답합니다.

예시

구분	Header		Packet Size	pID	CMD	CS1	CS2	Data[0]	Data[1]
Value	0xFF	0xFF	0x09(9)	0xFD	0x59	0xAC	0x52	0x00	0x00

현재 상태와 가장 최근의 에러 코드를 실어서 보냅니다. 에러가 없는 상태이므로 둘 다 0x00입니다.

명령 상세 설명 – ROLLBACK

10-1. ROLLBACK – 요청 패킷(0x1A)

구조

구분	Packet Size	pID	CMD	Data[0]	Data[1]
Value	7+2	0~0xFE	0x1A	ID Skip	Baud Skip

DRC에 저장되어 있는 공장 초기 값으로 비휘발성 레지스터를 초기화합니다. 초기화된 비휘발성 레지스터는 다음에 전원을 켜거나 리셋을 건 후부터 동작에 영향을 미치게 됩니다. Data[0]과 Data[1]의 ID Skip, Baud Skip은 각각 초기화 시 e(ID)와 e(Baud Rate)를 초기화 대상에서 제외할 것인지 여부를 결정합니다. ID Skip이 1이면 e(ID)를 초기화하지 않고, Baud Skip이 1이면 e(Baud Rate)를 초기화하지 않습니다.

예시

■ r(ID)가 253인 DRC를 e(ID)만 빼고 공장 초기화 하는 패킷에 대한 응답패킷

구분	Header		Packet Size	pID	CMD	CS1	CS2	Data[0]	Data[1]
Value	0xFF	0xFF	0x09(9)	0xFD	0x1A	0xEE	0x10	0x01	0x00

e(ID)만을 빼고 초기화 하려는 패킷이므로 ID Skip은 1, Baud Skip은 0이 됩니다.

■ r(ID)가 253인 DRC를 e(ID), e(Baud Rate)만 빼고 공장 초기화 하는 패킷

구분	Header		Packet Size	pID	CMD	CS1	CS2	Data[0]	Data[1]
Value	0xFF	0xFF	0x09(9)	0xFD	0x1A	0xEE	0x10	0x01	0x01

e(ID)와 e(Baud Rate)를 빼고 초기화 하려 하므로 ID Skip은 1, Baud Skip도 1이 됩니다.

10-2. ROLLBACK – 응답 패킷(0x5A)

구조

구분	Packet Size	pID	CMD	Data[0]	Data[1]
Value	7+2	r(ID)	0x5A	r(Status Error)	r(Error Codes[0])

r(Status Error)와 r(Error Codes[0])의 값을 응답 패킷에 실어서 반환합니다. Optional Data의 크기는 2바이트로 고정 되어 있기 때문에 Packet Size는 9바이트로 고정됩니다. pID는 현재 응답하는 DRC의 r(ID)가 실리게 되며, CMD는 요청 패킷 CMD 0x1A에 Bitwise OR 연산한 0x5A이 됩니다.

응답 조건

ROLLBACK은 r(ACK Policy)가 2(모든 패킷에 대하여 응답)일 경우에만 응답합니다. 단, 요청 패킷의 pID가 254(Broadcasting ID)일 경우에는 r(ACK Policy)가 2라도 응답하지 않습니다.

예시

- r(ID)가 253인 DRC를 e(ID)만 빼고 공장 초기화 하는 패킷에 대한 응답패킷

구분	Header		Packet Size	pID	CMD	CS1	CS2	Data[0]	Data[1]
Value	0xFF	0xFF	0x09(9)	0xFD	0x5A	0xAE	0x50	0x00	0x00

현재 상태와 가장 최근의 에러 코드를 실어서 보냅니다. 에러가 없는 상태이므로 둘 다 0x00입니다.

명령 상세 설명 – REBOOT

11-1. REBOOT – 요청 패킷(0x1B)

구조

구분	Packet Size	pID	CMD
Value	7	0~0xFE	0x1B

DRC에 SW 리셋을 요청합니다. DRC는 이 패킷을 받으면 자신을 리셋 하여 초기 부팅을 다시 진행합니다.

예시

■ r(ID)가 253인 DRC에 REBOOT 명령을 요청하는 패킷

구분	Header		Packet Size	pID	CMD	CS1	CS2
Value	0xFF	0xFF	0x07(7)	0xFD	0x1B	0xE0	0x1E

11-2. REBOOT – 응답 패킷(0x5B)

구조

구분	Packet Size	pID	CMD	Data[0]	Data[1]
Value	7+2	r(ID)	0x5B	r(Status Er–ror)	r(Error Codes[0])

r(Status Error)와 r(Error Codes[0])의 값을 응답 패킷에 실어서 반환합니다. Optional Data의 크기는 2바이트로 고정 되어 있기 때문에 Packet Size는 9바이트로 고정됩니다. pID는 현재 응답하는 DRC의 r(ID)가 실리게 되며, CMD는 요청 패킷 CMD 0x1B에 Bitwise OR 연산한 0x5B이 됩니다.

응답 조건

REBOOT은 r(ACK Policy)가 2(모든 패킷에 대하여 응답)일 경우에만 응답합니다. 단, 요청 패킷의 pID가 254(Broadcasting ID)일 경우에는 r(ACK Policy)가 2라도 응답하지 않습니다.

예시

■ r(ID)가 253인 DRC에 보낸 REBOOT 요청 패킷에 대한 응답 패킷

구분	Header		Packet Size	pID	CMD	CS1	CS2	Data[0]	Data[1]
Value	0xFF	0xFF	0x09(9)	0xFD	0x5B	0xAE	0x50	0x00	0x00

현재 상태와 가장 최근의 에러 코드를 실어서 보냅니다. 에러가 없는 상태이므로 둘 다 0x00입니다.

명령 상세 설명 – ZIGBEE

12-1. ZIGBEE – 요청 패킷(0x1C)

구조

구분	Packet Size	pID	CMD	Data[0]
Value	7+1	0~0xFE	0x1C	Instruction

DRC에 장착되어 있는 Zigbee 모듈에 관련된 제어 명령을 수행합니다. ZIGBEE는 Instruction에 따라서 6개의 명령으로 나뉩니다. 휘발성 레지스터 맵에는 r(Zigbee PANID), r(Zigbee SADDR), r(Zigbee DSTADDR), r(Zigbee ACKREQ), r(Zigbee BACKOFF) 등 5가지 항목(총 8바이트)의 Zigbee와 관련된 레지스터가 있습니다. 각 레지스터는 Zigbee 모듈 내부에 저장된 속성값에 대응됩니다. Zigbee를 통한 무선 통신과 연결 케이블을 통한 유선 통신은 동시에 이루어질 수 없으므로 주의해야 합니다.

- Instruction이 0일 때에는 Zigbee 모듈의 속성값을 휘발성 레지스터로 읽어옵니다.
- Instruction이 1일 때에는 휘발성 레지스터의 속성값을 Zigbee 모듈로 덮어씁니다.
- Instruction이 2일 때에는 Zigbee 모듈의 속성값을 출고 상태로 초기화 합니다.
- Instruction이 3일 때에는 Zigbee 모듈을 리셋합니다.
- Instruction이 4일 때에는 유선 통신 모드(연결 케이블과 COM PORT를 통한 통신)로 전환합니다.
- Instruction이 5일 때에는 무선 통신 모드(Zigbee를 통한 통신)로 전환합니다.

예시

- r(ID)가 253인 DRC에서 Zigbee의 속성값을 읽어오는 패킷

구분	Header		Packet Size	pID	CMD	CS1	CS2	Data[0]
Value	0xFF	0xFF	0x08(8)	0xFD	0x1C	0xE8	0x16	0x00

모듈로부터 RAM으로 속성값을 읽어오는 것이므로 Instruction은 0입니다.

■ r(ID)가 253인 DRC에서 Zigbee 모듈을 출고 상태로 초기화하는 패킷

구분	Header		Packet Size	pID	CMD	CS1	CS2	Data[0]
Value	0xFF	0xFF	0x08(8)	0xFD	0x1C	0xEA	0x14	0x02

출고 상태로 초기화이므로 Instruction은 2입니다.

■ r(ID)가 253인 DRC를 무선 통신으로 전환하는 패킷

구분	Header		Packet Size	pID	CMD	CS1	CS2	Data[0]
Value	0xFF	0xFF	0x08(8)	0xFD	0x1C	0xEC	0x12	0x05

무선 통신 모드 전환의 Instruction은 5입니다.

12-2. ZIGBEE – 응답 패킷(0x1C)

구조

구분	Packet Size	pID	CMD	Data[0]	Data[1]	Data[2]
Value	7+3	r(ID)	0x5C	Success	r(Status Error)	r(Error Codes[0])

ZIGBEE의 응답 패킷에는 해당 명령의 성공 여부를 나타내는 Success라는 항목이 Data[0]에 실립니다. Success는 Zigbee 명령이 성공했을 시에 1, Zigbee 모듈이 장착되어 있지 않거나 통신 상의 문제로 실패했을 시에 0이 됩니다. Optional Data에는 추가로 r(Status Error)와 r(Error Codes[0])의 값도 포함됩니다. Optional Data의 크기가 3바이트로 고정되기 때문에 Packet Size는 10바이트입니다. pID는 현재 응답하는 DRC의 r(ID)가 실리게 되며, CMD는 요청 패킷 CMD 0x1C에 Bitwise OR 연산한 0x5C이 됩니다.

응답 조건

ZIGBEE는 r(ACK Policy)가 1(Read 명령일 때만 응답), 2(모든 패킷에 대하여 응답)일 경우에 응답합니다. 단, 요청 패킷의 pID가 254(Broadcasting ID)일 경우에는 응답하지 않습니다.

예시

■ r(ID)가 253인 DRC에서 Zigbee의 속성값을 읽어오는 패킷에 대한 응답 패킷(Zigbee 장착 시)

구분	Header		Packet Size	pID	CMD	CS1	CS2	Data[0]	Data[1]	Data[2]
Value	0xFF	0xFF	0x0A(10)	0xFD	0x5C	0xAA	0x54	0x01	0x00	0x00

Zigbee가 장착되어 있으므로 통신이 정상적으로 이루어졌을 것이고, Success의 값은 1입니다.

예시

- r(ID)가 253인 DRC에서 Zigbee를 출고 상태 초기화하는 패킷에 대한 응답 패킷(Zigbee 장착 시)

구분	Header		Packet Size	pID	CMD	CS1	CS2	Data[0]	Data[1]	Data[2]
Value	0xFF	0xFF	0x0A(10)	0xFD	0x5C	0xAA	0x54	0x01	0x00	0x00

마찬가지로 Zigbee 값이 정상적으로 초기화 되었고, Success의 값은 1입니다.

- r(ID)가 253인 DRC를 무선 통신으로 전환하는 패킷에 대한 응답 패킷(Zigbee 미 장착 시)

구분	Header		Packet Size	pID	CMD	CS1	CS2	Data[0]	Data[1]	Data[2]
Value	0xFF	0xFF	0x0A(10)	0xFD	0x5C	0xAA	0x54	0x00	0x00	0x00

Zigbee가 장착되어 있지 않으므로, 무선 통신 전환 시도는 실패합니다. 따라서 Success의 값은 0입니다.

명령 상세 설명 – REMOCON

13-1. REMOCON – 요청 패킷(0x1D)

구조

구분	Packet Size	pID	CMD	Data[0]	Data[1]	Data[2]
Value	7+3	0~0xFE	0x1D	Channel	Length	Data

DRC에는 리모컨 리시버 모듈을 붙일 수 있어서, IR 리모컨으로 제어 명령을 내릴 수 있습니다. 하지만 IR 리모컨이 없는 환경이나 Zigbee 모듈을 통한 무선 환경에서 IR 리모컨 대신에 패킷을 통해 제어를 할 수 있도록 하는 것이 REMOCON 명령입니다. Data[0]에는 리모컨의 Channel(0x61~0x6A)가 들어가며, Data[1]에는 리모컨 버튼을 누른 시간에 해당하는 Length(0~240, 1당 125ms)가 들어가며, Data[2]에는 리모컨 버튼의 키 Data가 들어갑니다. DRC가 리모컨 값을 받으면 Channel과 r(Remocon Channel)을 비교한 후, 일치한다면 약 250ms 동안 r(Remocon Length)값과 r(Remocon Data)값을 각각 Length, Data로 바꿉니다. 250ms 후에는 r(Remocon Length)값과 r(Remocon Data)값이 다시 각각 0과 254로 돌아가므로, REMOCON 요청 패킷을 사용할 때에는 125ms의 주기로 Length 값을 1씩 증가시키면서 보내는 것을 권장합니다.

예시

- 모든 DRC(Broadcasting)에 0x61의 채널로 1초 동안 버튼 0x21이 눌렸음을 알리는 요청 패킷

구분	Header		Packet Size	pID	CMD	CS1	CS2	Data[0]	Data[1]	Data[2]
Value	0xFF	0xFF	0x0A(10)	0xFE	0x1D	0xA0	0x5E	0x61	0x08	0x21

모든 DRC를 향해서 보내는 것이므로 pID는 0xFE이며, 채널이 0x61이므로 Data[0]에는 0x61이 들어갑니다. Length 값은 1당 125ms이므로, 1초는 8에 해당됩니다. 따라서 Data[1]에는 8의 값이 들어가며, Data[2]에는 리모컨 키 값인 0x21이 들어갑니다.

13-2. REMOCON – 응답 패킷(0x1D)

REMOCON 명령은 응답 패킷이 없습니다.

명령 상세 설명 – SERVO_FW_UPDATE

14-1. SERVO_FW_UPDATE – 요청 패킷(0x1E)

구조

구분	Packet Size	pID	CMD
Value	7	0~0xFE	0x1Z

DRC에 연결된 서보의 펌웨어(Firmware)를 업데이트해야 할 때 사용하는 패킷입니다. 서보의 펌웨어를 업데이트할 때는 평소에 사용하는 프로토콜과 다른 것을 사용하기 때문에, SERVO_FW_UPDATE 요청 패킷을 보내 특수한 모드로 넘어가야 합니다. 이 모드에서는 PC와 제어기 간의 통신은 일체 할 수 없고, 오로지 PC와 서보가 직접적으로 연결된 것처럼 동작합니다.

예시

■ r(ID)가 253인 DRC를 서보 펌웨어 업데이트 모드로 바꾸는 패킷

구분	Header		Packet Size	pID	CMD	CS1	CS2
Value	0xFF	0xFF	0x09(9)	0xFD	0x1E	0xE4	0x1A

14-2. SERVO_FW_UPDATE – 응답 패킷(0x5E)

구조

구분	Packet Size		pID	CMD	Data[0]	Data[1]
Value	7+2	r(ID)	0x5E	r(Status Error)	r(Error Codes[0])	0xA0

r(Status Error)와 r(Error Codes[0])의 값을 응답 패킷에 실어서 반환합니다. Optional Data의 크기는 2바이트로 고정 되어 있기 때문에 Packet Size는 9바이트로 고정됩니다. pID는 현재 응답하는 DRC의 r(ID)가 실리게 되며, CMD는 요청 패킷 CMD 0x1E에 Bitwise OR 연산한 0x5E이 됩니다.

응답 조건

SERVO_FW_UPDATE은 r(ACK Policy)가 2(모든 패킷에 대하여 응답)일 경우에만 응답합니다. 단, 요청 패킷의 pID가 254(Broadcasting ID)일 경우에는 r(ACK Policy)가 2라도 응답하지 않습니다.

예시

■ r(ID)가 253인 DRC를 서보 펌웨어 업데이트 모드로 바꾸는 요청 패킷에 대한 응답 패킷

구분	Header		Packet Size	pID	CMD	CS1	CS2	Data[0]	Data[1]
Value	0xFF	0xFF	0x09(9)	0xFD	0x5E	0xAA	0x54	0x00	0x00

현재 상태와 가장 최근의 에러 코드를 실어서 보냅니다. 에러가 없는 상태이므로 둘 다 0x00입니다.

ADC Lookup Table – Voltage

ADC		VIN	ADC		VIN	ADC		VIN	ADC		VIN
Decimal	HEX		Decimal	HEX		Decimal	HEX		Decimal	HEX	
0	0	0.000	64	40	4.722	128	80	9.444	192	C0	14.167
1	1	0.074	65	41	4.796	129	81	9.518	193	C1	14.240
2	2	0.148	66	42	4.870	130	82	9.592	194	C2	14.314
3	3	0.221	67	43	4.944	131	83	9.666	195	C3	14.388
4	4	0.295	68	44	5.017	132	84	9.740	196	C4	14.462
5	5	0.369	69	45	5.091	133	85	9.813	197	C5	14.536
6	6	0.443	70	46	5.165	134	86	9.887	198	C6	14.609
7	7	0.516	71	47	5.239	135	87	9.961	199	C7	14.683
8	8	0.590	72	48	5.313	136	88	10.035	200	C8	14.757
9	9	0.664	73	49	5.386	137	89	10.109	201	C9	14.831
10	A	0.738	74	4A	5.460	138	8A	10.182	202	CA	14.905
11	B	0.812	75	4B	5.534	139	8B	10.256	203	CB	14.978
12	C	0.885	76	4C	5.608	140	8C	10.330	204	CC	15.052
13	D	0.959	77	4D	5.681	141	8D	10.404	205	CD	15.126
14	E	1.033	78	4E	5.755	142	8E	10.477	206	CE	15.200
15	F	1.107	79	4F	5.829	143	8F	10.551	207	CF	15.273
16	10	1.181	80	50	5.903	144	90	10.625	208	D0	15.347
17	11	1.254	81	51	5.977	145	91	10.699	209	D1	15.421
18	12	1.328	82	52	6.050	146	92	10.773	210	D2	15.495
19	13	1.402	83	53	6.124	147	93	10.846	211	D3	15.569
20	14	1.476	84	54	6.198	148	94	10.920	212	D4	15.642
21	15	1.549	85	55	6.272	149	95	10.994	213	D5	15.716
22	16	1.623	86	56	6.345	150	96	11.068	214	D6	15.790
23	17	1.697	87	57	6.419	151	97	11.141	215	D7	15.864
24	18	1.771	88	58	6.493	152	98	11.215	216	D8	15.938
25	19	1.845	89	59	6.567	153	99	11.289	217	D9	16.011
26	1A	1.918	90	5A	6.641	154	9A	11.363	218	DA	16.085
27	1B	1.992	91	5B	6.714	155	9B	11.437	219	DB	16.159
28	1C	2.066	92	5C	6.788	156	9C	11.510	220	DC	16.233
29	1D	2.140	93	5D	6.862	157	9D	11.584	221	DD	16.306
30	1E	2.214	94	5E	6.936	158	9E	11.658	222	DE	16.380
31	1F	2.287	95	5F	7.010	159	9F	11.732	223	DF	16.454
32	20	2.361	96	60	7.083	160	A0	11.806	224	E0	16.528
33	21	2.435	97	61	7.157	161	A1	11.879	225	E1	16.602
34	22	2.509	98	62	7.231	162	A2	11.953	226	E2	16.675
35	23	2.582	99	63	7.305	163	A3	12.027	227	E3	16.749
36	24	2.656	100	64	7.378	164	A4	12.101	228	E4	16.823
37	25	2.730	101	65	7.452	165	A5	12.174	229	E5	16.897
38	26	2.804	102	66	7.526	166	A6	12.248	230	E6	16.970
39	27	2.878	103	67	7.600	167	A7	12.322	231	E7	17.044
40	28	2.951	104	68	7.674	168	A8	12.396	232	E8	17.118
41	29	3.025	105	69	7.747	169	A9	12.470	233	E9	17.192
42	2A	3.099	106	6A	7.821	170	AA	12.543	234	EA	17.266
43	2B	3.173	107	6B	7.895	171	AB	12.617	235	EB	17.339
44	2C	3.247	108	6C	7.969	172	AC	12.691	236	EC	17.413
45	2D	3.320	109	6D	8.043	173	AD	12.765	237	ED	17.487
46	2E	3.394	110	6E	8.116	174	AE	12.839	238	EE	17.561
47	2F	3.468	111	6F	8.190	175	AF	12.912	239	EF	17.635
48	30	3.542	112	70	8.264	176	B0	12.986	240	F0	17.708
49	31	3.615	113	71	8.338	177	B1	13.060	241	F1	17.782
50	32	3.689	114	72	8.411	178	B2	13.134	242	F2	17.856
51	33	3.763	115	73	8.485	179	B3	13.207	243	F3	17.930
52	34	3.837	116	74	8.559	180	B4	13.281	244	F4	18.003
53	35	3.911	117	75	8.633	181	B5	13.355	245	F5	18.077
54	36	3.984	118	76	8.707	182	B6	13.429	246	F6	18.151
55	37	4.058	119	77	8.780	183	B7	13.503	247	F7	18.225
56	38	4.132	120	78	8.854	184	B8	13.576	248	F8	18.299
57	39	4.206	121	79	8.928	185	B9	13.650	249	F9	18.372
58	3A	4.280	122	7A	9.002	186	BA	13.724	250	FA	18.446
59	3B	4.353	123	7B	9.076	187	BB	13.798	251	FB	18.520
60	3C	4.427	124	7C	9.149	188	BC	13.872	252	FC	18.594
61	3D	4.501	125	7D	9.223	189	BD	13.945	253	FD	18.668
62	3E	4.575	126	7E	9.297	190	BE	14.019	254	FE	18.741
63	3F	4.648	127	7F	9.371	191	BF	14.093	255	FF	18.815

ADC Lookup Table – Temperature

ADC		VIN	ADC		VIN	ADC		VIN	ADC		VIN
Decimal	HEX		Decimal	HEX		Decimal	HEX		Decimal	HEX	
0	0	–80.57	64	40	–1.34	128	80	25.00	192	C0	56.99
1	1	–72.89	65	41	–0.89	129	81	25.41	193	C1	57.67
2	2	–64.26	66	42	–0.44	130	82	25.82	194	C2	58.36
3	3	–58.84	67	43	0.01	131	83	26.24	195	C3	59.05
4	4	–54.80	68	44	0.46	132	84	26.65	196	C4	59.76
5	5	–51.55	69	45	0.90	133	85	27.07	197	C5	60.48
6	6	–48.81	70	46	1.34	134	86	27.49	198	C6	61.21
7	7	–46.43	71	47	1.78	135	87	27.91	199	C7	61.96
8	8	–44.32	72	48	2.21	136	88	28.33	200	C8	62.71
9	9	–42.41	73	49	2.64	137	89	28.75	201	C9	63.48
10	A	–40.68	74	4A	3.07	138	8A	29.18	202	CA	64.27
11	B	–39.08	75	4B	3.50	139	8B	29.60	203	CB	65.06
12	C	–37.59	76	4C	3.93	140	8C	30.03	204	CC	65.88
13	D	–36.20	77	4D	4.35	141	8D	30.46	205	CD	66.71
14	E	–34.89	78	4E	4.77	142	8E	30.89	206	CE	67.55
15	F	–33.66	79	4F	5.19	143	8F	31.32	207	CF	68.41
16	10	–32.49	80	50	5.61	144	90	31.76	208	D0	69.29
17	11	–31.37	81	51	6.03	145	91	32.20	209	D1	70.19
18	12	–30.31	82	52	6.45	146	92	32.64	210	D2	71.11
19	13	–29.29	83	53	6.86	147	93	33.08	211	D3	72.05
20	14	–28.31	84	54	7.27	148	94	33.52	212	D4	73.01
21	15	–27.36	85	55	7.68	149	95	33.97	213	D5	74.00
22	16	–26.45	86	56	8.09	150	96	34.42	214	D6	75.01
23	17	–25.57	87	57	8.50	151	97	34.87	215	D7	76.04
24	18	–24.72	88	58	8.91	152	98	35.33	216	D8	77.10
25	19	–23.89	89	59	9.32	153	99	35.78	217	D9	78.19
26	1A	–23.09	90	5A	9.72	154	9A	36.24	218	DA	79.31
27	1B	–22.31	91	5B	10.13	155	9B	36.71	219	DB	80.46
28	1C	–21.54	92	5C	10.53	156	9C	37.17	220	DC	81.65
29	1D	–20.80	93	5D	10.94	157	9D	37.64	221	DD	82.87
30	1E	–20.08	94	5E	11.34	158	9E	38.11	222	DE	84.13
31	1F	–19.37	95	5F	11.74	159	9F	38.59	223	DF	85.44
32	20	–18.68	96	60	12.14	160	A0	39.07	224	E0	86.78
33	21	–18.00	97	61	12.55	161	A1	39.55	225	E1	88.17
34	22	–17.34	98	62	12.95	162	A2	40.04	226	E2	89.62
35	23	–16.69	99	63	13.35	163	A3	40.53	227	E3	91.12
36	24	–16.05	100	64	13.75	164	A4	41.02	228	E4	92.67
37	25	–15.42	101	65	14.15	165	A5	41.52	229	E5	94.29
38	26	–14.81	102	66	14.54	166	A6	42.02	230	E6	95.98
39	27	–14.20	103	67	14.94	167	A7	42.52	231	E7	97.75
40	28	–13.61	104	68	15.34	168	A8	43.03	232	E8	99.59
41	29	–13.02	105	69	15.74	169	A9	43.55	233	E9	101.53
42	2A	–12.45	106	6A	16.14	170	AA	44.07	234	EA	103.57
43	2B	–11.88	107	6B	16.54	171	AB	44.59	235	EB	105.71
44	2C	–11.32	108	6C	16.94	172	AC	45.12	236	EC	107.98
45	2D	–10.76	109	6D	17.34	173	AD	45.65	237	ED	110.38
46	2E	–10.22	110	6E	17.74	174	AE	46.19	238	EE	112.93
47	2F	–9.68	111	6F	18.13	175	AF	46.74	239	EF	115.65
48	30	–9.15	112	70	18.53	176	B0	47.29	240	F0	118.57
49	31	–8.62	113	71	18.93	177	B1	47.84	241	F1	121.72
50	32	–8.10	114	72	19.33	178	B2	48.40	242	F2	125.12
51	33	–7.59	115	73	19.73	179	B3	48.97	243	F3	128.83
52	34	–7.08	116	74	20.13	180	B4	49.54	244	F4	132.89
53	35	–6.58	117	75	20.54	181	B5	50.12	245	F5	137.38
54	36	–6.08	118	76	20.94	182	B6	50.71	246	F6	142.40
55	37	–5.59	119	77	21.34	183	B7	51.30	247	F7	148.06
56	38	–5.10	120	78	21.74	184	B8	51.90	248	F8	154.56
57	39	–4.62	121	79	22.15	185	B9	52.51	249	F9	162.13
58	3A	–4.14	122	7A	22.55	186	BA	53.13	250	FA	171.18
59	3B	–3.66	123	7B	22.96	187	BB	53.75	251	FB	182.34
60	3C	–3.19	124	7C	23.36	188	BC	54.38	252	FC	196.72
61	3D	–2.72	125	7D	23.77	189	BD	55.02	253	FD	216.58
62	3E	–2.26	126	7E	24.18	190	BE	55.67	254	FE	247.46
63	3F	–1.80	127	7F	24.59	191	BF	56.33	255	FF	310.08
63	3F	4.648	127	7F	9.371	191	BF	14.093	255	FF	18.815

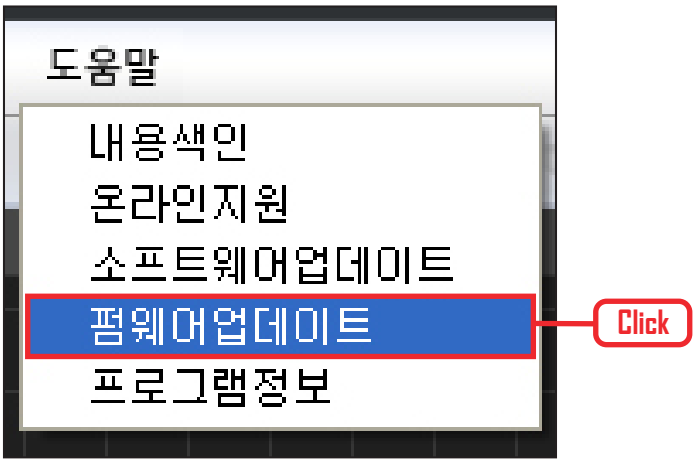
Error Code 상세 명세

Status Error Flag	Error Code	상세
Exceed Input Voltage limit	0x01	전압이 너무 낮음
	0x02	전압이 너무 높음
Exceed Temperature limit	0x03	온도가 너무 높음
Servo Missing	0x11	상태 자동 확인에 의해 서보 레지스터 읽던 중 서보 응답 없음
	0x12	Task 실행에 의해 서보 레지스터 읽던 중 서보 응답 없음
EEP REG distorted	0x21	EEPROM의 모델명이 잘못 됨
	0x22	EEPROM의 ID가 잘못 됨
	0x23	EEPROM 데이터가 손상됨
Servo Status Error	0x31	서보 상태 에러
	0x33	너무 많은 서보가 제어기에 연결됨
Invalid Packet	0x41	Zigbee Ack가 온전히 오지 않거나 Noise가 들어옴
	0x42	Zigbee Ack에서 Check Sum Error
	0x43	Zigbee Ack에서 Unknown Command
	0x44	Zigbee Ack가 들어왔으나 ID가 0xFC가 아님
	0x45	Zigbee Ack에서 사이즈가 너무 큰 패킷이 들어옴
	0x46	Zigbee Ack에서 명령과 안 맞는 사이즈의 패킷이 들어옴
	0x47	Zigbee Ack가 오지 않음
	0x51	Servo Ack가 온전히 오지 않거나 Noise가 들어옴
	0x52	Servo Ack에서 Check Sum Error
	0x53	Servo Ack에서 Unknown Command
	0x54	Servo Ack에서 Invalid ID의 패킷이 들어옴
	0x56	Servo Ack에서 사이즈가 너무 큰 패킷이 들어옴
	0x57	Servo Ack에서 명령과 안 맞는 사이즈의 패킷이 들어옴
	0x58	Servo Ack를 받는 UART 버퍼가 꽉참
	0x59	Servo에 보낼 패킷을 저장하는 버퍼가 꽉참
	0x61	PC측 패킷이 온전히 오지 않거나 Noise가 들어옴
	0x62	PC측 패킷에서 Check Sum Error
	0x63	PC측 패킷에서 Unknown Command
	0x64	PC측 패킷에서 Invalid ID의 패킷이 들어옴
	0x66	PC측 패킷에서 사이즈가 너무 큰 패킷이 들어옴
	0x67	PC측 패킷에서 명령에 안 맞는 사이즈의 패킷이 들어옴
	0x68	PC측 패킷을 받는 UART 버퍼가 꽉참
	0x71	EEP/RAM의 WRITE/READ 명령이 레지스터 범위를 벗어남
	0x72	RAM_WRITE에서 잘못된 값을 씀
	0x73	RAM_WRITE에서 Status에 잘못된 값을 씀
	0x74	CON_CHECK에서 패킷에 잘못된 ID가 들어가 있음
	0x75	PLAY_MOTION에서 잘못된 모션 번호가 들어가 있음
	0x76	PLAY_TASK에서 잘못된 Instruction이 들어가 있음
	0x77	REMOCON에서 잘못된 Channel이나 Length가 들어가 있음
	0x78	ZIGBEE에서 잘못된 Instruction이 들어가 있음
	0x79	PLAY_BUZZ에서 잘못된 버저 번호가 들어가 있음

Error Code 상세 명세

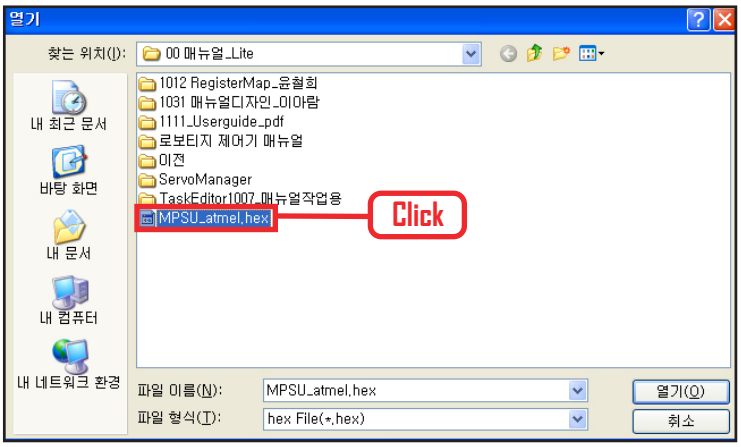
Status Error Flag	Error Code	상세
Flash Data Distorted	0x81	없는 Motion을 실행하라고 함
	0x82	Motion 데이터에 문제 있음
	0x83	Motion 데이터의 축 수와 현재 축 수가 다름
	0x84	다음 프레임까지 시간이 음수인 프레임을 만남
	0x85	너무 많은 Repeat 명령이 중첩되었음(최대 3개까지 가능)
	0x91	Task 데이터에 문제 있음
	0x92	계산 식 처리 중 이상
	0x93	프로그램 스택 오버플로우
	0x94	MPSU RAM에 Load 중 잘못된 레지스터 주소
	0x95	MPSU RAM에 Load 중 잘못된 레지스터 길이
	0x96	Servo RAM에 Load 중 잘못된 레지스터 주소
	0x97	Servo RAM에 Load 중 잘못된 레지스터 길이
	0x98	Servo RAM에 Load 중 잘못된 ID
	0x99	MPSU RAM에서 Read 중 잘못된 레지스터 길이
	0x9A	Servo RAM에서 Read 중 잘못된 레지스터 길이
	0x9B	Servo RAM에서 Read 중 잘못된 ID
	0xA1	Motion 명령에 범위 벗어난 값
	0xA2	Motion Ready 명령에 범위 벗어난 값
	0xA3	서보 제어 명령에 범위 벗어난 값
	0xA5	제어기 LED 제어 명령에 범위 벗어난 값
	0xA6	버저 멜로디 명령에 범위 벗어난 값
	0xA7	버저 음표 명령에 범위 벗어난 값
	0xB2	없는 Buzzer를 실행하라고 함

펌웨어 업데이트



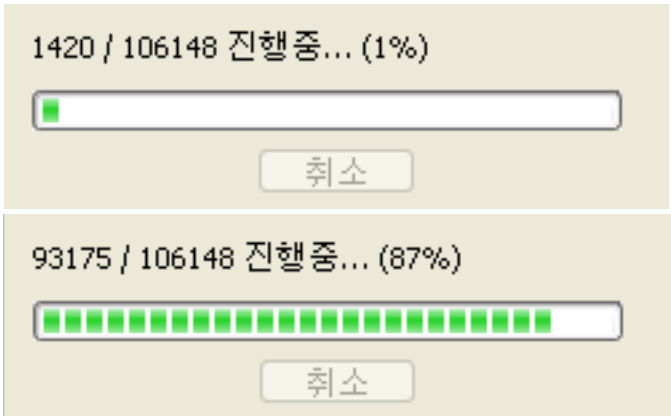
01 도움말 > 펌웨어업데이트

제어기 펌웨어 업데이트는 DR-Visual Logic 에서 할 수 있습니다.
제어기와 PC가 연결된 상태에서
도움말 > 펌웨어데이트를 클릭합니다.



02 파일 불러오기

펌웨어 파일 확장명은 .hex 입니다. 최신 hex 파일이 저장된 폴더로 이동하여 .hex 파일을 선택합니다.



03 펌웨어 업데이트

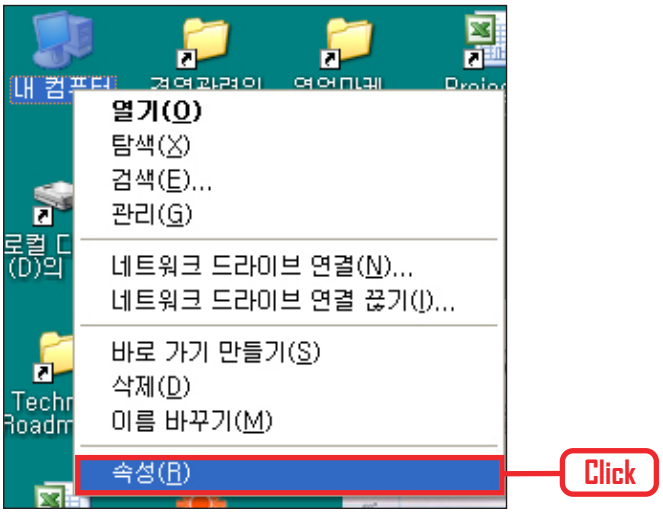
펌웨어를 업데이트합니다.

〈 주의 〉
펌웨어 업데이트 중에 업데이트를 중지한다거나 PC가 꺼지지 않도록 주의해주세요. DRC 작동이 안될 수 있습니다. 만약 업데이트중에 꺼져서 제어기가 작동이 안될시에는 DRC 복구를 해주거나 문의바랍니다.

COM 포트 설정 및 속도 개선하기

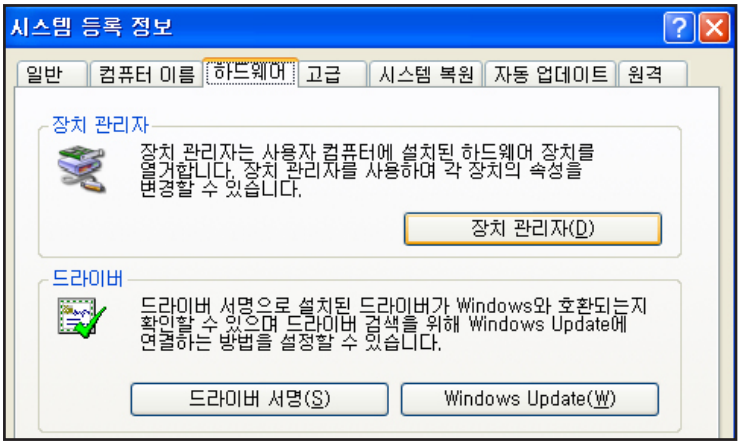
예제설명

PC와 로봇이 연결이 안되면 COM port 를 설정해주어야합니다. 펌웨어 업데이트 를 원활히 하기위해 속도도 개선해 주어야 사용하는데 편리합니다. COM Port 설정 및 속도개선을 알아봅니다.



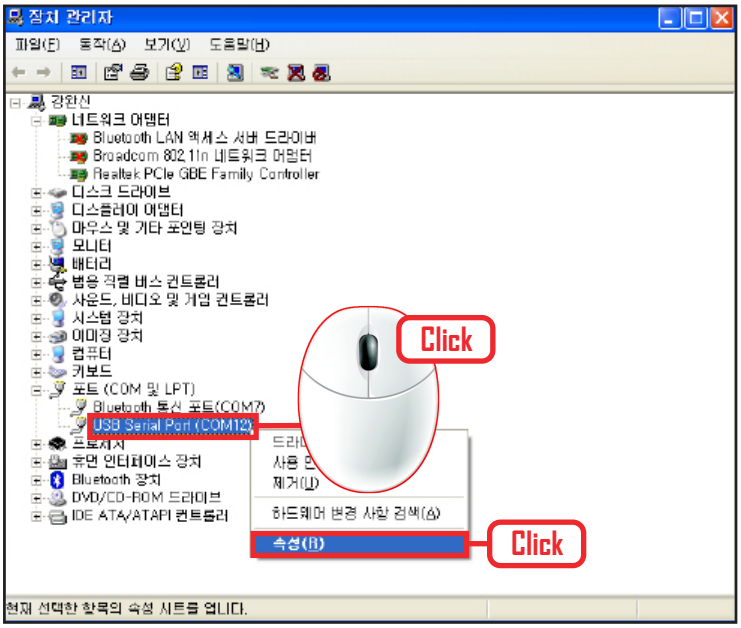
01 속성

내 컴퓨터에서 우측 마우스를 클릭하여 속성을 클릭합니다.



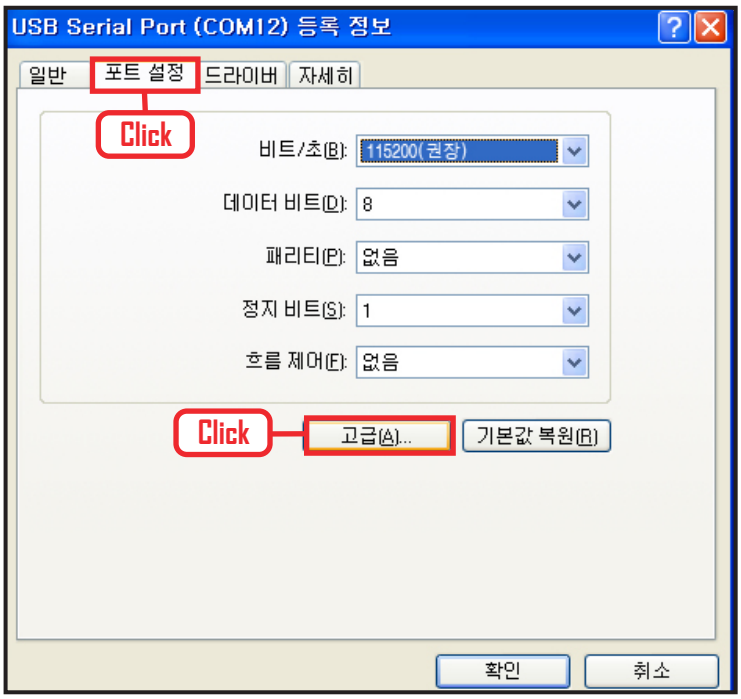
02 하드웨어 장치관리자

내 컴퓨터 속성을 클릭하면, 시스템 등록 정보가 나옵니다. 시스템 등록정보 탭중에 하드웨어를 선택하면 장치관리자 아이콘이 나옵니다.



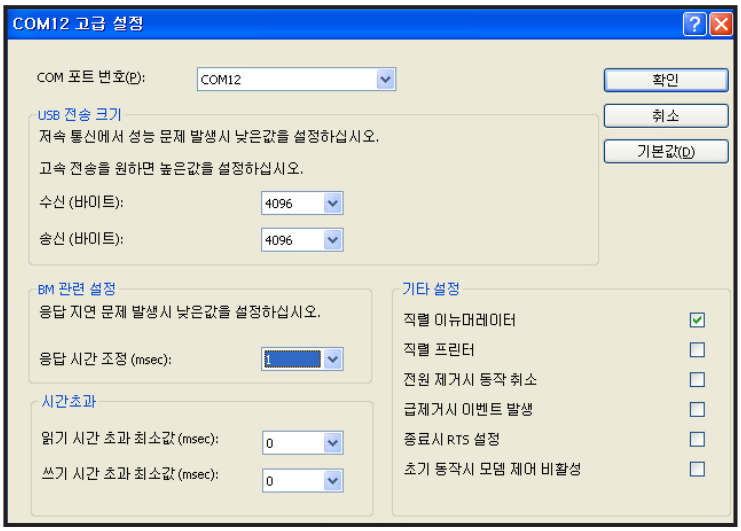
03 장치관리자 포트

장치관리자 트리중에 포트를 클릭합니다.
COM 포트 사용리스트가 보입니다.
USB Serial Port (COM12)에 우측 마우스를 클릭하면 속성이 뜹니다.



04 USB Serial Port 등록정보

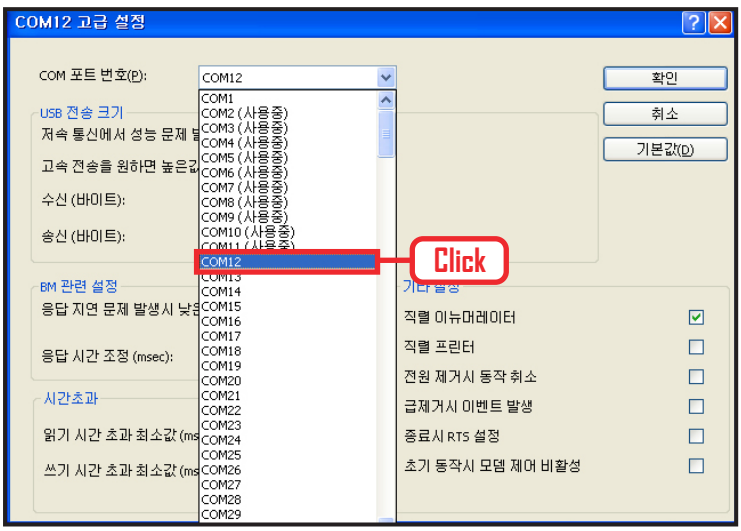
USB Serial Port(COM12) 등록정보의 포트 설정을 클릭합니다.
비트/초(B) 115200 (권장)



05 COM12 고급 설정

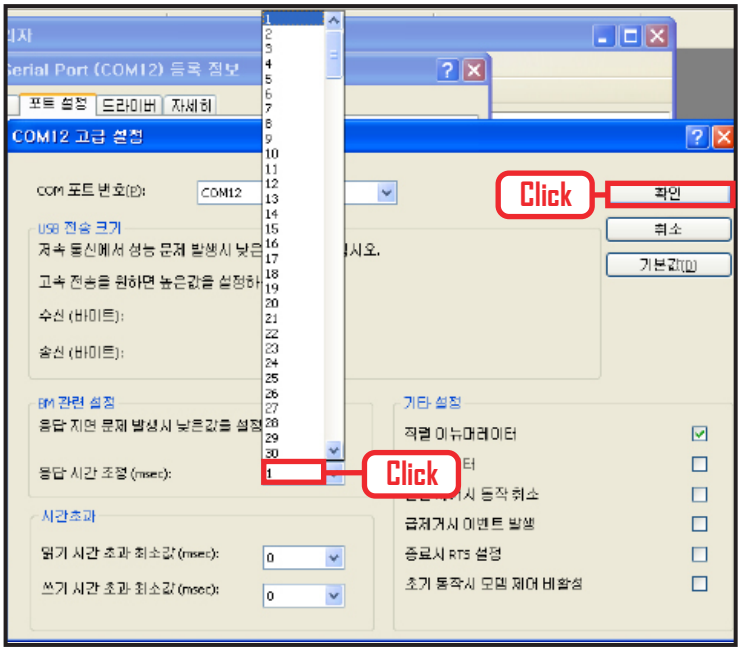
고급설정창에서 COM port 변경 및 속도개선을 할 수 있습니다.

현재 설정해 놓은 포트가 COM12 입니다.



06 COM 설정

COM 포트 번호를 클릭하면, COM1~30 까지 나열됩니다.
그 중에 원하는 포트를 선택하고, DR-SIM 이나 DR-Visual Logic 환경설정에서 그 포트 번호를 선택하여 로봇과 연결합니다.



07 속도개선

BM 관련 설정에 응답시간 조정을 클릭합니다. 기본 14로 설정되어있는 값을 1로 조정합니다. 그럼 COM 포트 속도는 14배가 빨라집니다.
1을 선택하고 확인을 클릭합니다.

유용한 정보

HOVIS

캘리브레이션(로봇 영점 잡기)

로봇을 조립한 후에 정확히 조립이 되었는지 확인하고 수정하는 작업입니다. 만약 로봇이 정확히 조립되지 않고 사용시에는 오작동이나 원하지 않는 동작이 발생할 수 있습니다.

DR-SIM 의 로봇제어를 클릭한 후 실제 로봇 모터의 정확한 위치를 바꾸면서 조정해봅니다.

01 연결하기

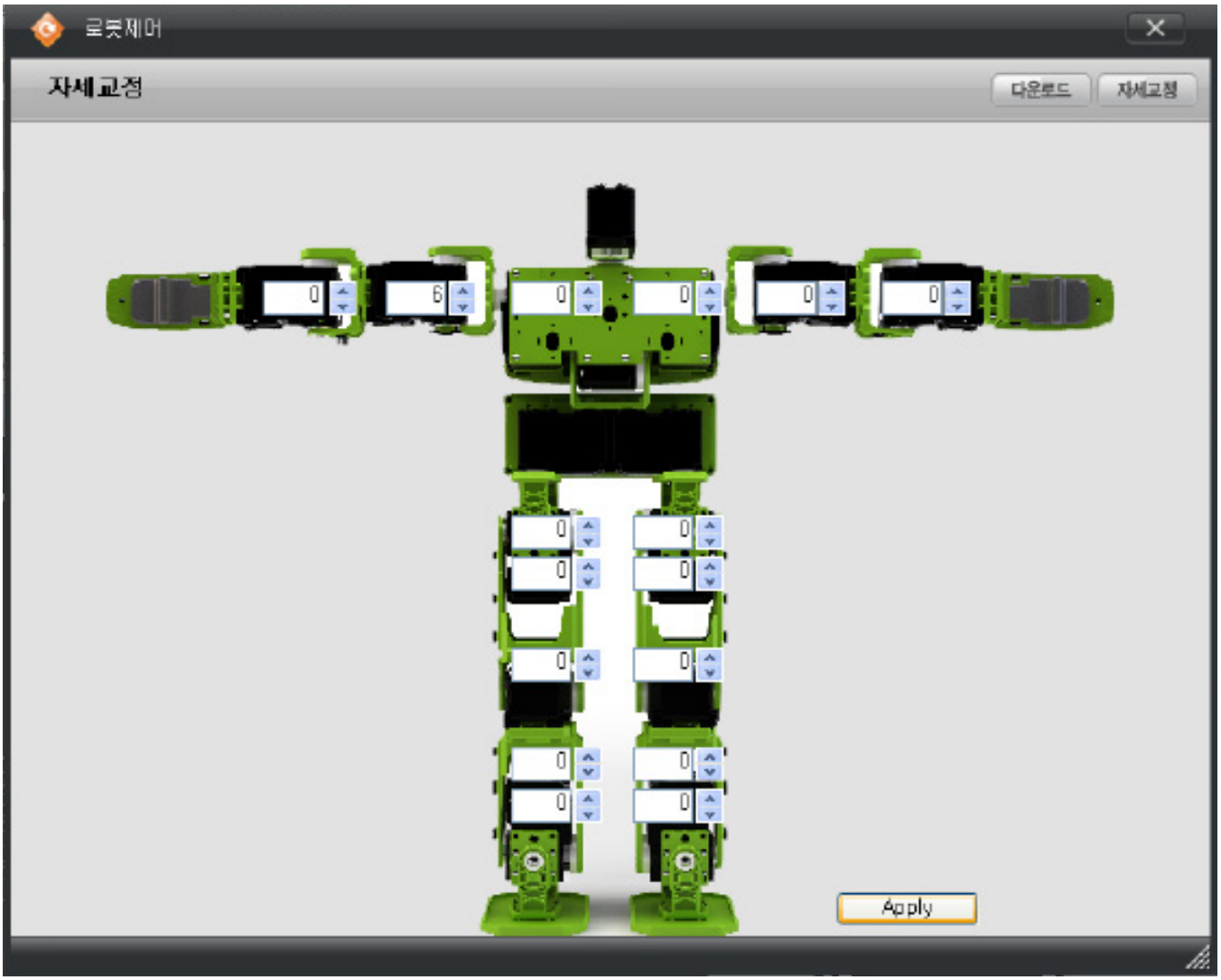
로봇을 연결합니다.
연결하기를 클릭합니다.

02 로봇제어

로봇의 전원을 킵니다.
토크 ON을 클릭합니다.
캘리브레이션은 로봇제어에서 합니다.
로봇제어를 클릭합니다.

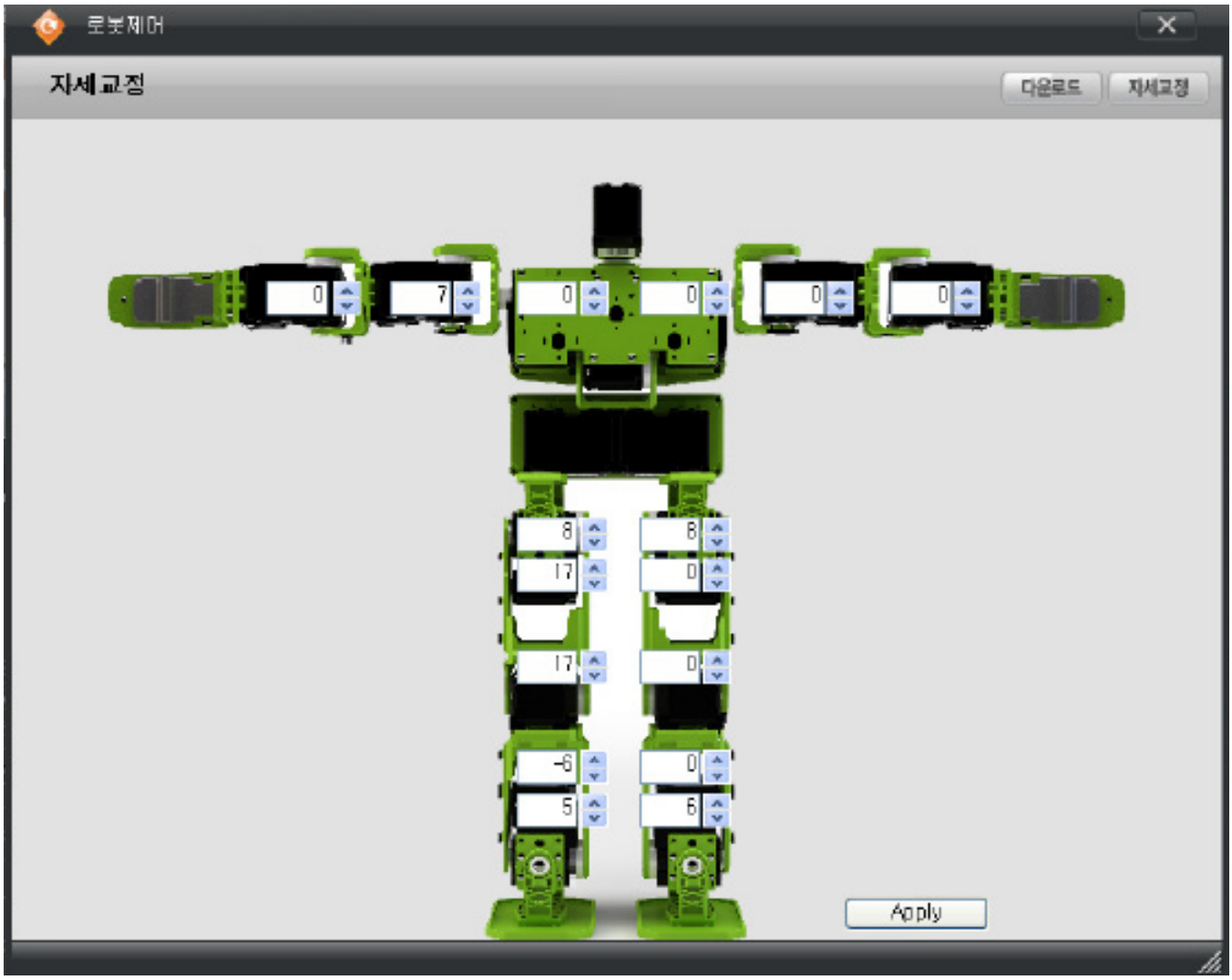
03 자세교정

로봇제어 창은 모션 다운로드와 자세교정으로 나뉩니다.
자세교정을 클릭합니다.



자세 교정을 클릭하면 현재 로봇의 캘리브레이션이 값이 나옵니다.
실제 로봇과 비교하면서 각 캘리브레이션값을 조절합니다.

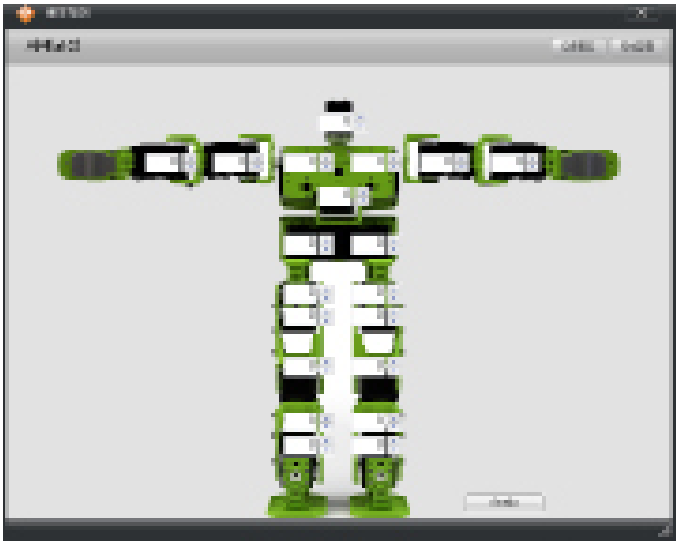
캘리브레이션은 127 ~ -128 까지 설정이 가능합니다. 상하아이콘을 이용하여 숫자를 변경하면 실제 로봇에서 조금씩 움직이는 것을 확인할 수 있습니다.

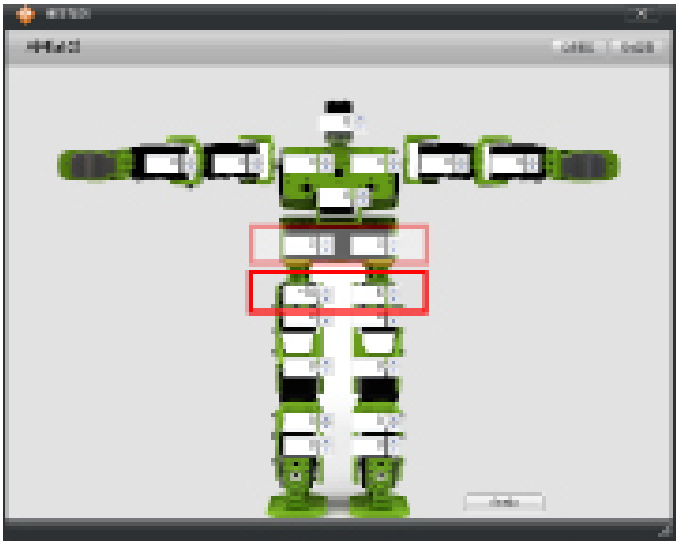


실제로봇을 보면서 움직임을 확인하고, 설정이 맞으면, Apply 를 클릭합니다.
Apply 를 클릭하면 로봇에 저장됩니다. 이후 로봇을 연결하면 조정된 값으로 숫자가 표기 됩니다.

캘리브레이션 TIP

로봇의 초기 자세는 조립상태에 따라 달라질 수 있습니다. 유저는 로봇을 정확히 동작시키기 위하여 초기 자세를 화면과 같이 보정하여야 합니다. 그러므로, 자세교정(캘리브레이션) 과정이 필요합니다.
로봇의 자세교정은 하체에서 상체로 하는 것이 유리합니다. 여러방법으로 로봇의 자세교정을 할 수 있으나, 일반적으로 다음과 같은 순서에 따릅니다.





〈설정전〉

〈설정후〉

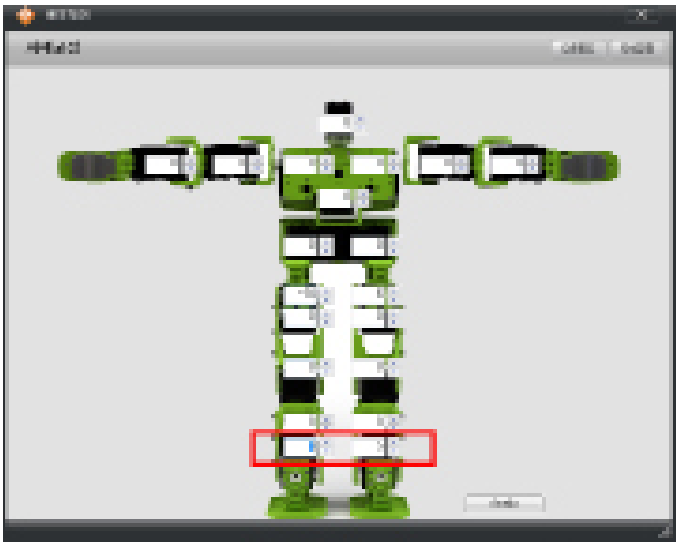
01

로봇제어 〉 자세교정 버튼을 클릭하여 자세교정창이 뜨면, 로봇을 살짝 들어올려 조립상태를 확인합니다.

02

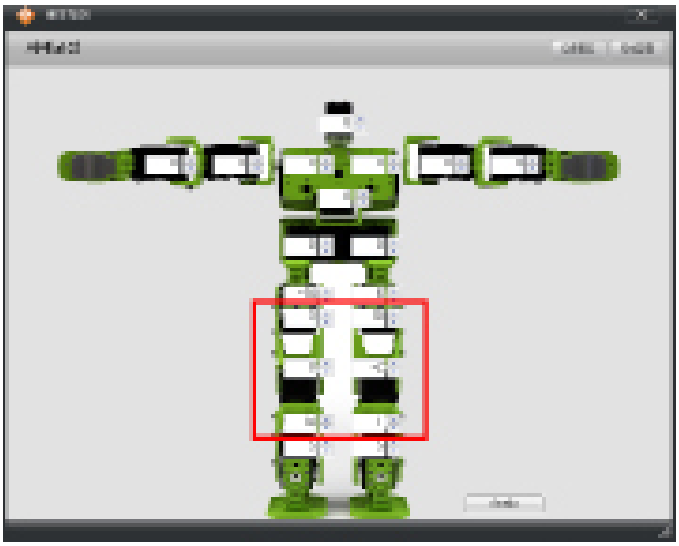
정면에서 로봇을 보면서, 로봇의 양 다리의 균형을 맞춥니다.

* 희미한 네모 박스는 18축, 20축 로봇의 경우 해당



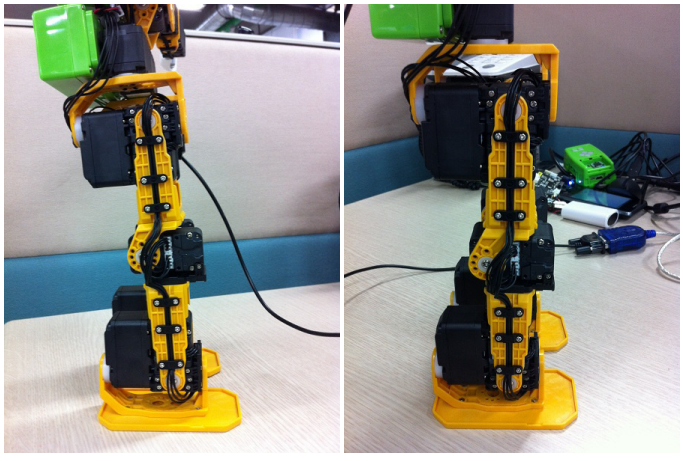
03

로봇의 양발이 바닥에 접지되도록 조절합니다.

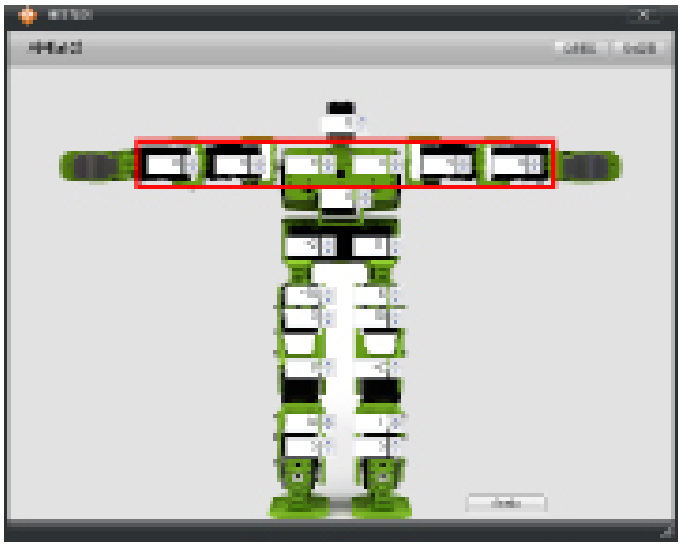


04

측면에서 로봇을 보면서, 각 다리의 수직을 맞춥니다.

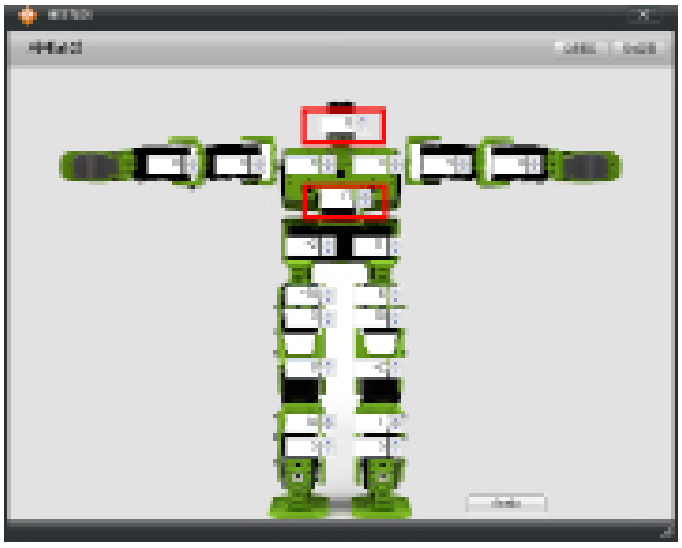


〈설정전〉 〈설정후〉



05

위에서 로봇을 보면서, 양팔이 일자형태가 되도록 조절합니다.



06

20축 로봇의 경우에 로봇의 허리와 머리를 조절합니다.

07

마지막으로 부분적으로 값을 조절하며 캘리브레이션을 마칩니다.

모터의 ID 변경하기

제어기(DRC)는 모터의 ID를 통해 각각의 모터를 식별합니다. 그러므로 로봇 조립시 모터를 ID값에 따라 올바른 위치에 조립하는 것이 중요합니다.
그러나 16축의 로봇을 18축 또는 20축의 로봇으로 재조립하거나, 로봇 조립시 모터를 잘 못 삽입한 경우, 모터의 ID를 재조정해야 합니다. 이 때 모터 ID 변경이 필요합니다.

- 16축에서 18축 또는 20축 변경시 재조립전에 반드시 조립할 모터의 ID를 변경한 후 조립하세요.
- 조립시 모터를 잘못된 위치에 조립한 경우, 다음과 같이 변경합니다.

  ex) 9번과 10번 모터의 위치가 바뀐경우

  9번 → 100번 (20번 부터 254까지 여유 ID입니다.)

  10번 → 9번

  100번 → 10번

다음은 HerkuleX Manager 프로그램을 통해 253번 모터의 ID를 15번으로 변경하는 예제입니다.
(HerkuleX Manager는 동부로봇 홈페이지에서 다운로드할 수 있습니다.)
http://www.dongburobot.com/jsp/cms/view.jsp?code=100122

01

모터를 제어기(DRC)에 연결하고 HerkuleX Manager를 실행합니다. COM Port설정 후, 연결하기 버튼을 클릭합니다.

02

연결하기 버튼을 클릭하면, 제어기(DRC)에 연결 된 모터가 왼쪽화면에 나타납니다. ID가 253인 모터에 ID를 15로 변경하기 위하여, [ID: 253] DRS-0101을 클릭합니다. 이어서 기본속성설정을 클릭하고, 스크롤바를 이용하여 ID&Policy화면이 보이도록 위치합니다.

03

Servo ID항목에 원하는 값을 입력합니다. 이 예제에서는 15를 입력합니다. 그리고 설정을 클릭합니다. 설정을 클릭하면 자동으로 모터의 ID Scan이 수행됩니다.

04

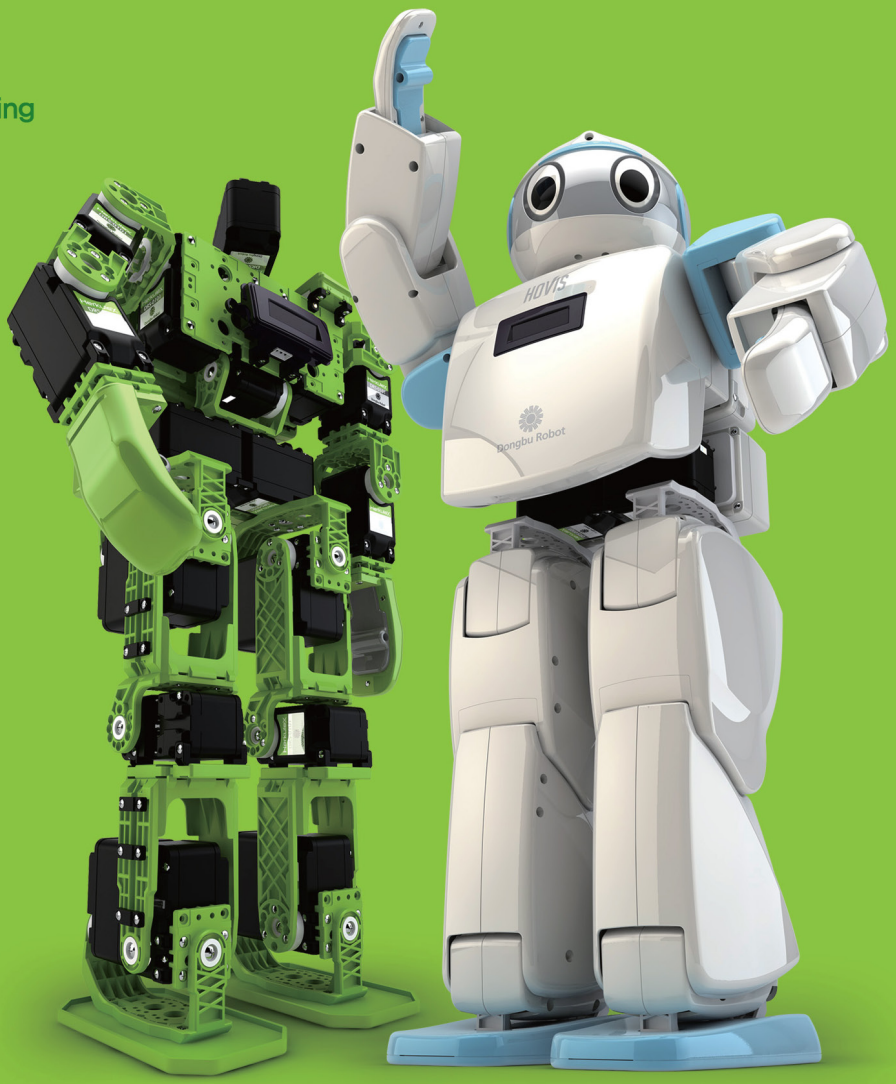
자동으로 ID Scan이 수행되면, 모터의 ID가 253에서 15로 변경된 걸 볼 수 있습니다. 마지막으로 [ID: 015]DRS-0101을 클릭하고, Save 버튼을 클릭하여 모터의 ID를 최종적으로 변경합니다. (ID Scan시 변경된 모터의 ID는 RAM Register에서 변경된 값입니다. RAM Register의 ID값은 전원이 공급이 되지 않으면 지워집니다. Save 버튼을 이용하여 변경된 모터의 ID를 EEP Register에 설정하여야만 최종적으로 모터 ID 변경이 완료됩니다.)

05

모터의 전원을 끊었다가 다시 연결합니다. 이후 HerkuleX Manager에서 ID Scan을 수행하여 모터의 ID가 제대로 변경되었는지 확인합니다.

HOVIS DRC, 비주얼로직 & C로봇프로그래밍

그래피컬 프로그래밍 툴인 **Visual Logic**을 통한 **알고리즘 학습**과
AVR C 프로그래밍을 통한 **마이크로프로세서 학습** 및 **로봇**을
제어할 수 있습니다.



Dongbu Robot

(주) 동부로봇 www.dongburobot.com
본 사 : 경기도 부천시 원미구 약대동 193 부천테크노파크 401동 11층, TEL : 032-329-5551 (내선 112),
FAX : 032-329-5569, E-MAIL : robotsales@dongbu.com
공 장 : 충청남도 천안시 서북구 직산읍 4산단 6길 27, TEL : 041-590-1700, FAX : 041-590-1701

9 788998 873035 93550
ISBN 978-89-98873-03-5