로봇으로 배우는

AVR C Programming



- ATmega128 마이크로콘트롤러
- 로봇제어기 DRC 펌웨어 쉽게 만들기

동부로봇 자료실

(매뉴얼 및 어플리케이션을 다운로드 받으십시오.)

www.dongburobot.com

Dongbu Robot

Dongbu Robot

로봇으로 배우는
AVR C Programming

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

본 교재는 ㈜동부로봇에서 저작한 교육 제작물로서 사전 승인없이 무단 복사, 복제 및 전제를 금합니다.

㈜동부로봇 All rights reserved

목차

PART 01 Hovis DRC와 ATMega128 익히기

Chapter 01 Hovis DRC 및 센서 ---------- 12

PART 02 Visual Logic 연계된 C언어 문법익히기

Chapter 01 C 언어 with Robot ---------- 28
Chapter 02 C 언어 훑어보기---------- 29
Chapter 03 변수 ---------- 31
Chapter 04 배열, 포인터, 구조체 ---------- 41
Chapter 05 메모리 ---------- 45
Chapter 06 구조체 ---------- 46
Chapter 07 반복과 분기 ---------- 49
Chapter 08 함수 ---------- 57

PART 03 AVR 로봇제어기 Firmware 익히기

Chapter 00 **AVR Studio 설치 및 Hello Programming** ---------- 66

Chapter 01 **LED** ---------- 84
AVR 원리와 H/W 원리 ---------- 84
DRC & H/W 준비 및 AVR Studio & S/W 실행 ---------- 85
예제 및 설명 ---------- 88
Hex 만들기 및 DRC 실행 ---------- 90

Chapter 02 **Button** ---------- 93
AVR 원리와 H/W 원리 ---------- 93
DRC & H/W 준비 및 AVR Studio & S/W 실행 ---------- 94
예제 및 설명 ---------- 97
Hex 만들기 및 DRC 실행 ---------- 101

Chapter 03 **Buzzer 비프** ---------- 104
AVR 원리와 H/W 원리 ---------- 104
DRC & H/W 준비 및 AVR Studio & S/W 실행 ---------- 274
예제 및 설명 ---------- 278
Hex 만들기 및 DRC 실행 ---------- 281

Chapter 04 **외부인터럽트 MIC** ---------- 114
AVR 원리와 H/W 원리 ---------- 114
DRC & H/W 준비 및 AVR Studio & S/W 실행 ---------- 118
예제 및 설명 ---------- 121
Hex 만들기 및 DRC 실행 ---------- 125

Chapter 05 **Timer/Counter** ---------- 128
AVR 원리와 H/W 원리 ---------- 128
DRC & H/W 준비 및 AVR Studio & S/W 실행 ---------- 135
예제 및 설명 ---------- 138
Hex 만들기 및 DRC 실행 ---------- 142

Chapter 06 **PSD (ADC)** ---------- 145
AVR 원리와 H/W 원리 ---------- 145
DRC & H/W 준비 및 AVR Studio & S/W 실행 ---------- 151
예제 및 설명 ---------- 154
Hex 만들기 및 DRC 실행 ---------- 157

Chapter 07 **USART (Serial)** ---------- 160
AVR 원리와 H/W 원리 ---------- 160
DRC & H/W 준비 및 AVR Studio & S/W 실행 ---------- 167
예제 및 설명 ---------- 175
Hex 만들기 및 DRC 실행 ---------- 183

Chapter 08 **Servo Motor ---------- 188**
AVR 원리와 H/W 원리 ---------- 188
DRC & H/W 준비 및 AVR Studio & S/W 실행 ---------- 189
예제 및 설명 ---------- 198
Hex 만들기 및 DRC 실행 ---------- 212

Chapter 09 **Busser Advanced ---------- 216**
AVR 원리와 H/W 원리 ---------- 216
DRC & H/W 준비 및 AVR Studio & S/W 실행 ---------- 220
예제 및 설명 ---------- 222
Hex 만들기 및 DRC 실행 ---------- 228

Chapter 10 **EEPROM ---------- 231**
AVR 원리와 H/W 원리 ---------- 231
DRC & H/W 준비 및 AVR Studio & S/W 실행 ---------- 232
예제 및 설명 ---------- 239
Hex 만들기 및 DRC 실행 ---------- 247

Chapter 11 **가속도/자이로센서 ---------- 251**
AVR 원리와 H/W 원리 ---------- 251
DRC & H/W 준비 및 AVR Studio & S/W 실행 ---------- 256
예제 및 설명 ---------- 264
Hex 만들기 및 DRC 실행 ---------- 269

Chapter 12 **IR 리모콘 ---------- 272**
AVR 원리와 H/W 원리 ---------- 272
DRC & H/W 준비 및 AVR Studio & S/W 실행 ---------- 274
예제 및 설명 ---------- 282
Hex 만들기 및 DRC 실행 ---------- 289

PART 04 로봇에 응용하기

Chapter 01

로봇모션제어---------- 294
AVR 원리와 H/W 원리 ---------- 294
DRC & H/W 준비 및 AVR Studio & S/W 실행 ---------- 295
예제 및 설명 ---------- 303
Hex 만들기 및 DRC 실행 ---------- 308

Chapter 02

덤프트럭 with 리모콘 ---------- 318
AVR 원리와 H/W 원리 ---------- 318
DRC & H/W 준비 및 AVR Studio & S/W 실행 ---------- 319
예제 및 설명 ---------- 327
Hex 만들기 및 DRC 실행 ---------- 350

표준강의 커리큘럼 추천

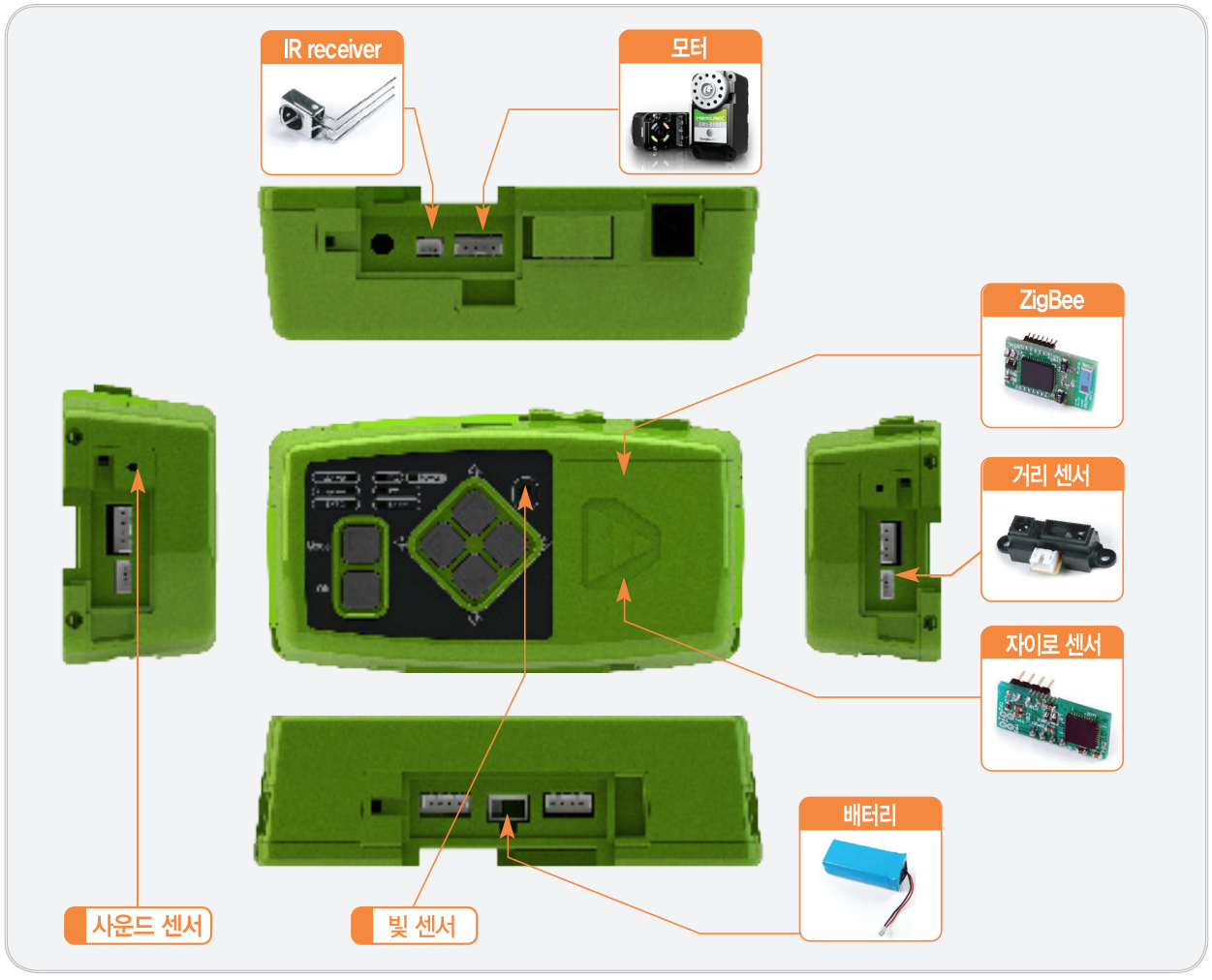
1강 : DRC 개요, Hovis 개요, AVR 개요, AVR 설치, Hello DRC AVR
2강 : C언어 – 변수, 제어문, 함수, 배열, 포인터, 구조체
3강 ~ 7강 : AVR 예제 – LED, Button, Buzzer,가속도,MIC,Timer
8강 : 중간고사
9강 ~ 15강 : AVR 예제 – 휴머노이드 모션, 덤프트럭, PSD, EEPROM, USART, Motor, ZigBee 응용로봇
16강 : 기말고사
별첨 : DRC 레지스터, 유용한 정보

PART 01

Hovis DRC와 ATMega128 익히기

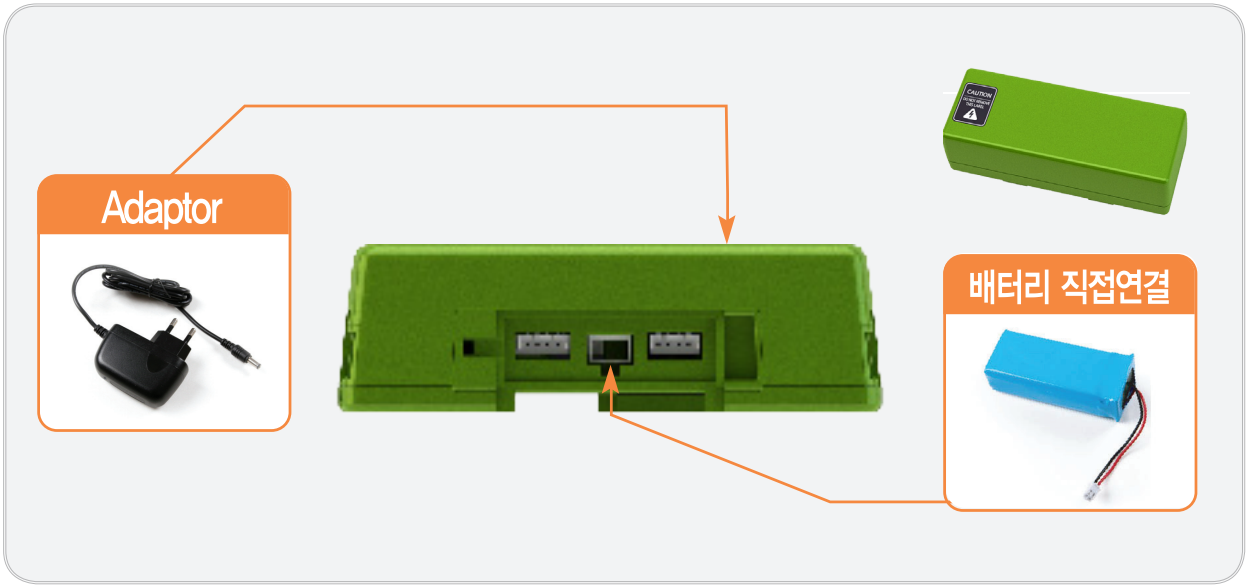
HOVIS DRC 개요

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**

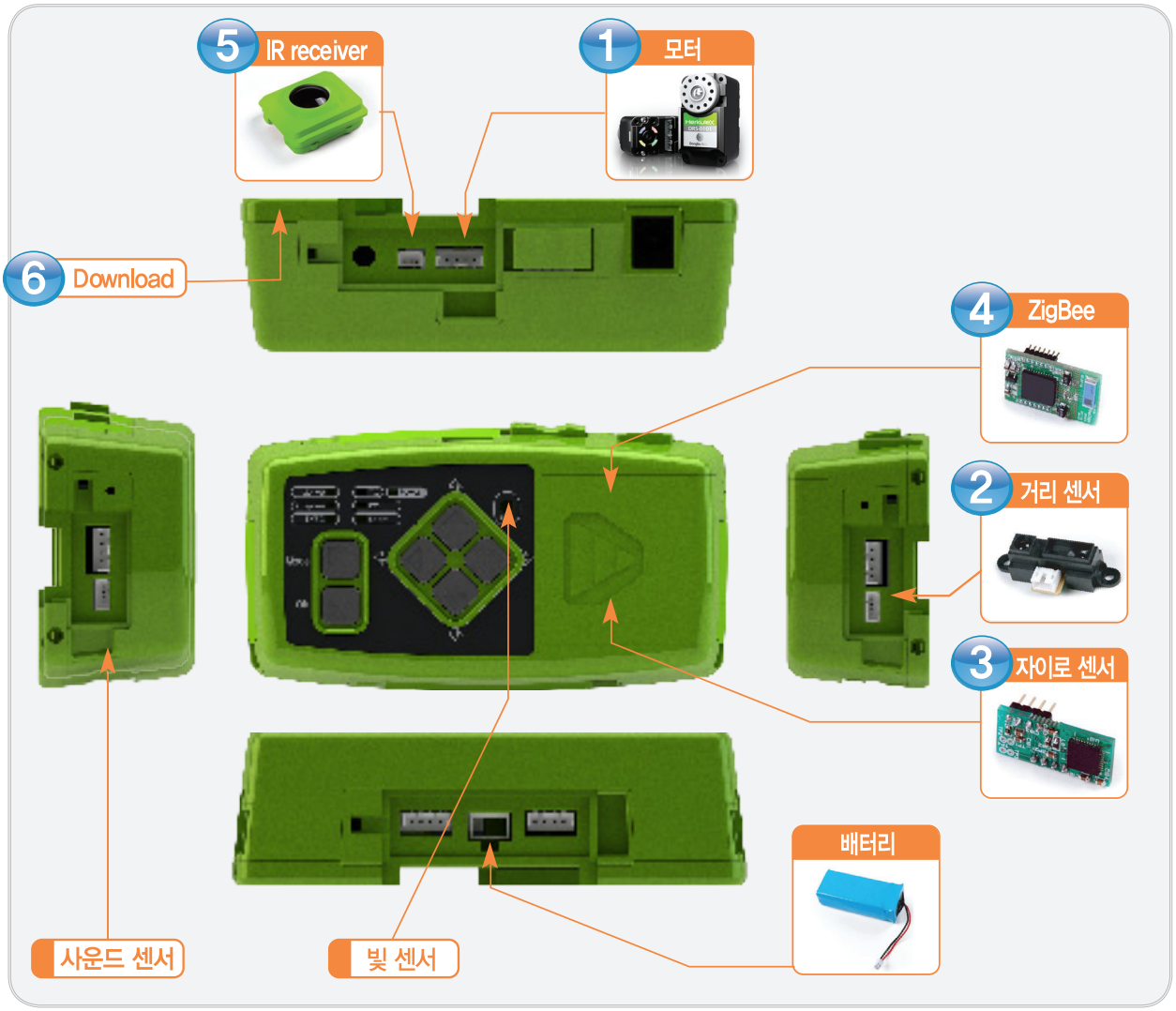
아답터 와 배터리 케이블을 이용하여 직접 충전할 수 있습니다.
또 다른 방법은 제어기 상단에 아답터 커넥터가 있습니다.
그 커넥터에 아답터를 꽂으면 제어기와 연결된 배터리에 자동 충전됩니다.



■ **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.

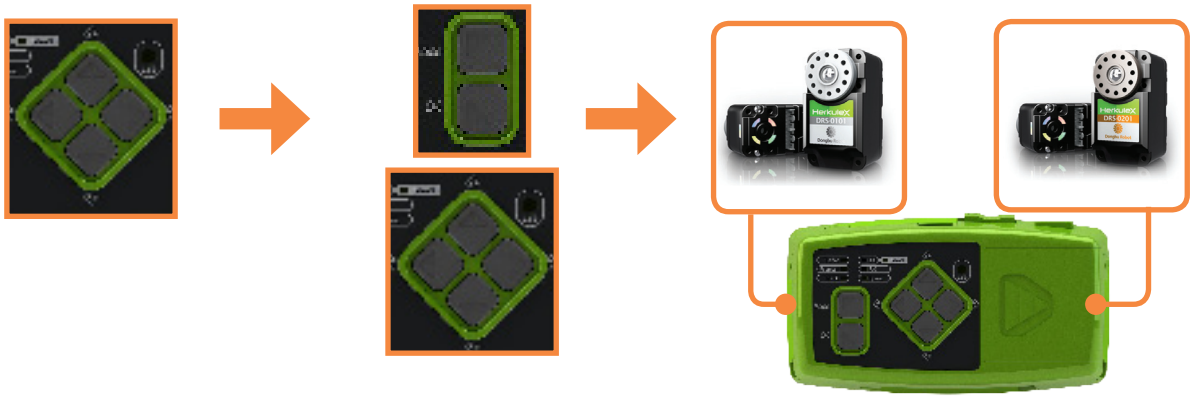
출시시 휴머노이드 전용 기본 Task Program 은 정의되어있고, User 가 수정 가능합니다.

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.

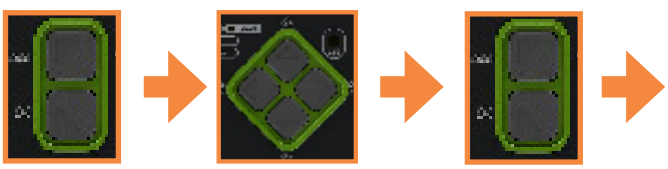
Firmware (사용자가 바꿀 수 없음)

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 켜져있음
	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

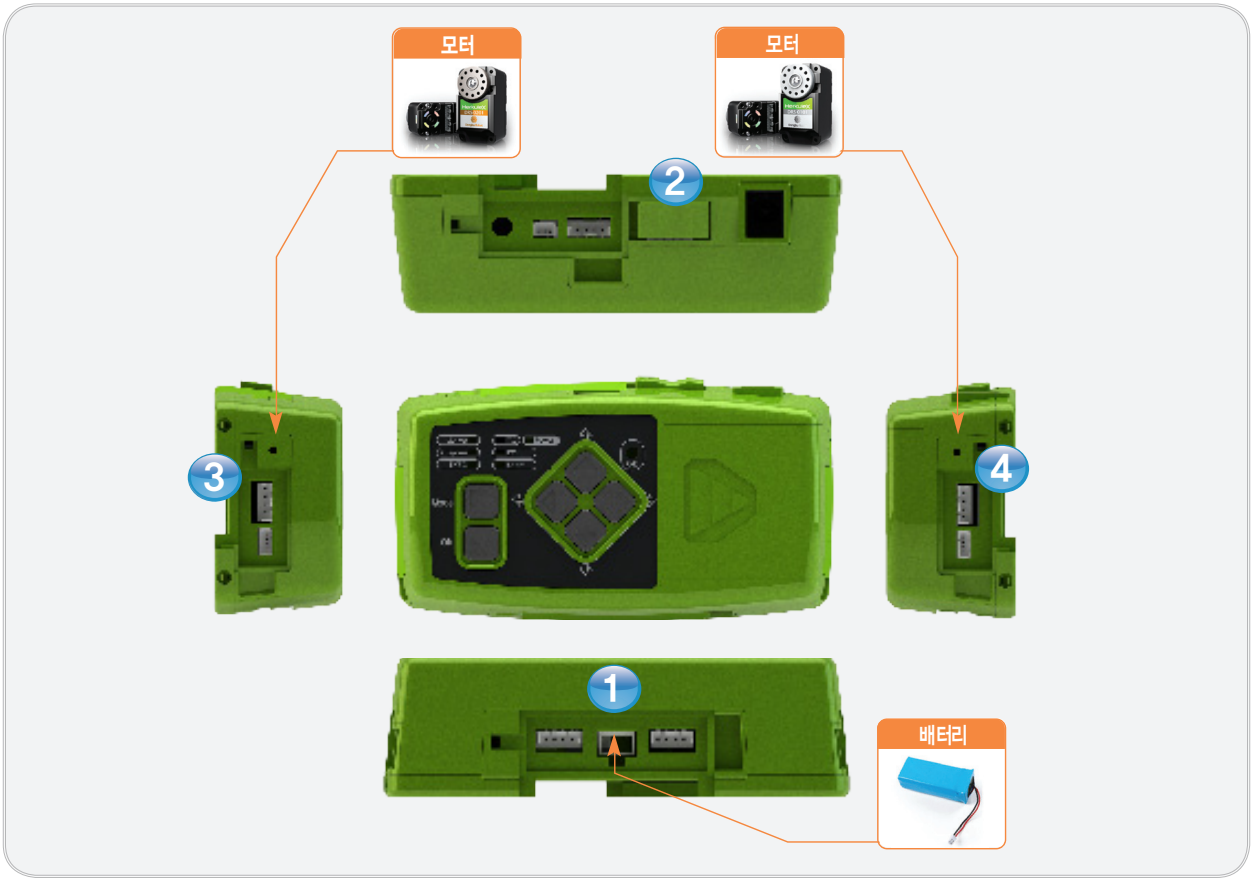
5 PSD Analog : (상) → 아날로그 PSD센서 연결 → (하) PSD 작동함
작동함과 동시에 모터 두개가 같은 방향으로 움직임. 물체의 거리가 멀 수록 더 빨리 움직이고
물체가 가까우면 느리게 움직임 → 물체가 10 cm 내로 접근하면 모터가 반대로 움직임
→ 벽 감지 후 피하는 시나리오임

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.)

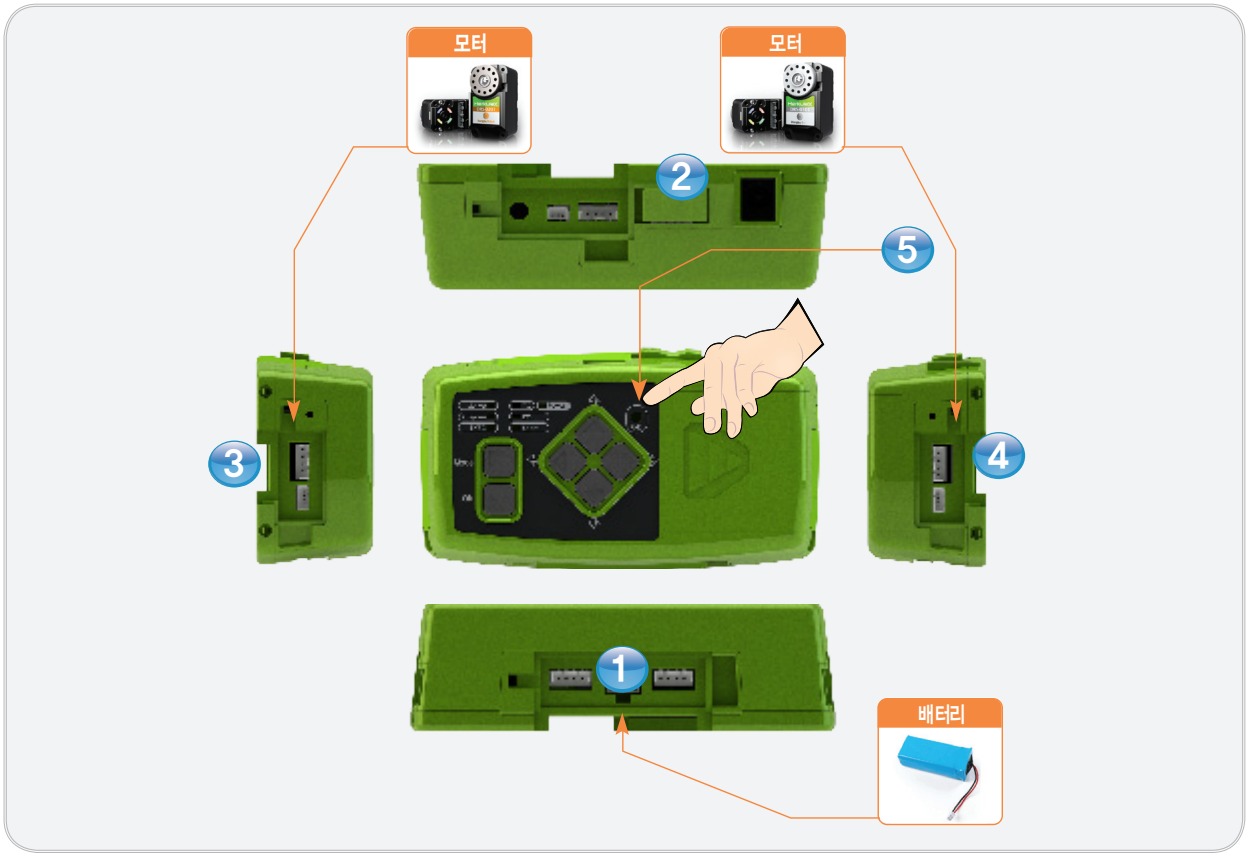
모터는 바깥쪽으로 놓고 테스트 합니다. 바퀴가 돌아가는 것을 가상한 시나리오입니다.

→ 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.)

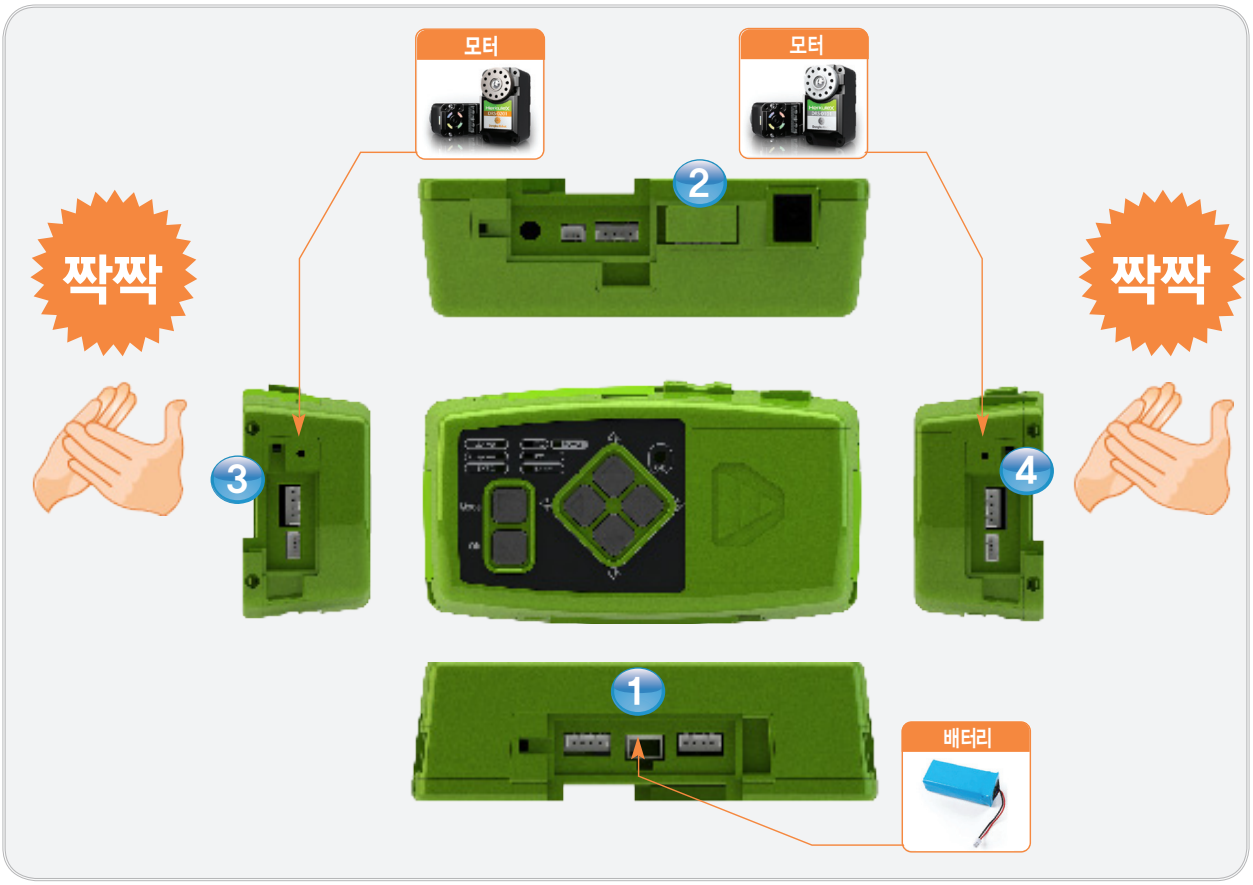
빛이 사라졌을 때, 로봇이 팔로 허공을 짚는 가상 시나리오입니다.

→ 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.)

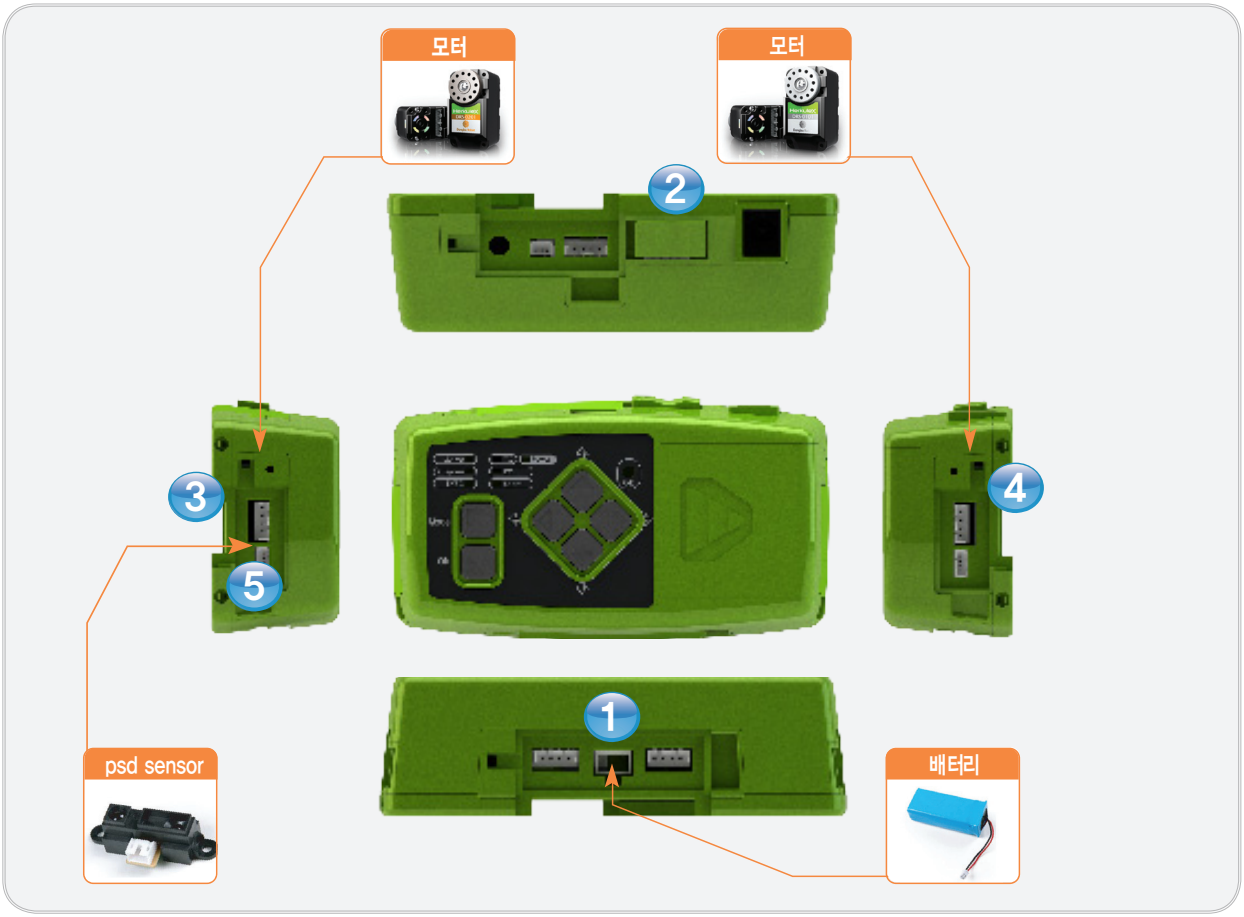
Sound Sensor 제어기 좌우에 위치합니다. 박수치는 방향의 모터가 움직이는 시나리오입니다.

→ 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

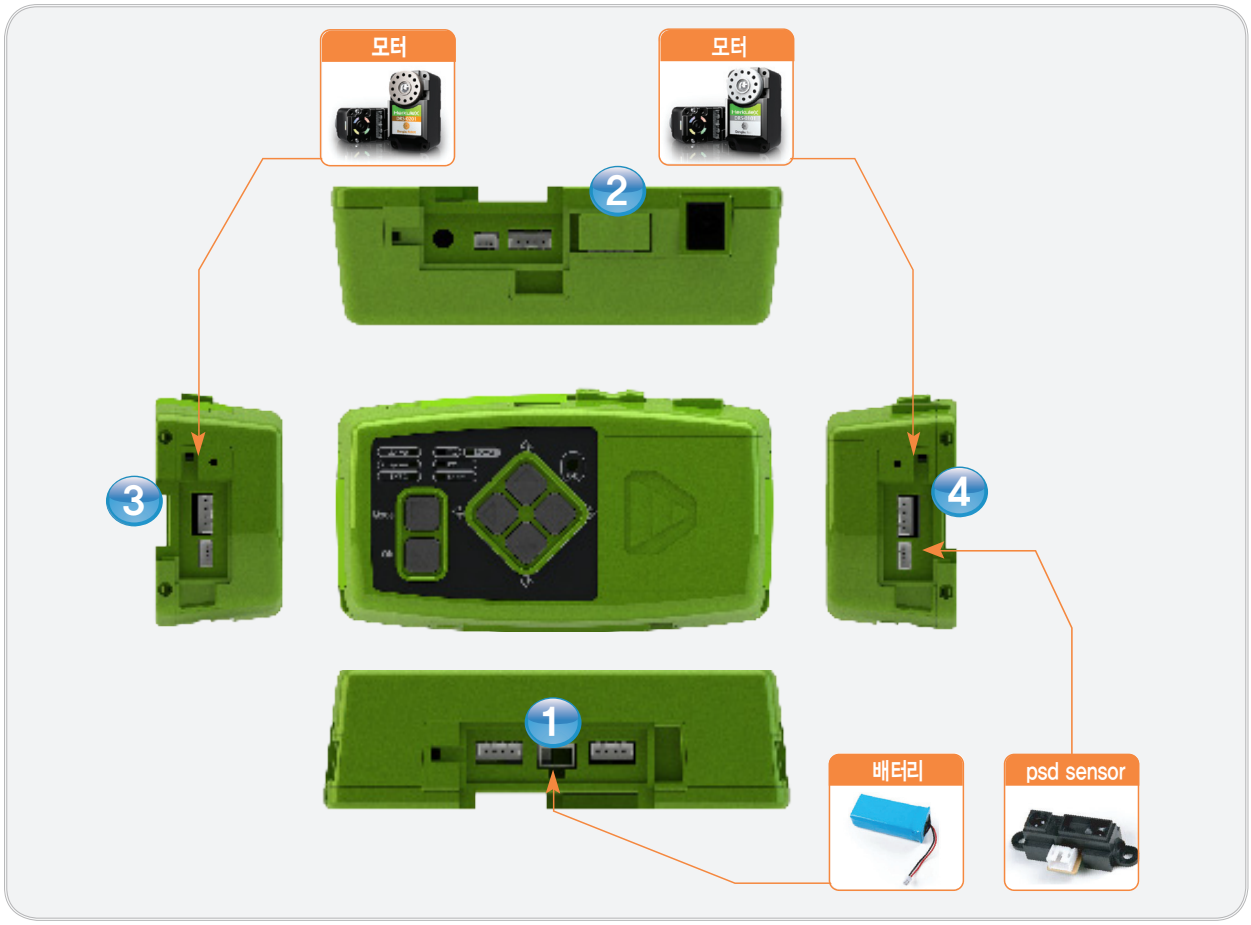
PSD Digital Sensor 는 일정거리를 기준으로 가깝냐 머냐를 체크하는 센서입니다. 흔히 땅의 깊이를 체크하여 낭떠러지를 감지했을 때 로봇을 멈추게 하는 시나리오입니다.

→ 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

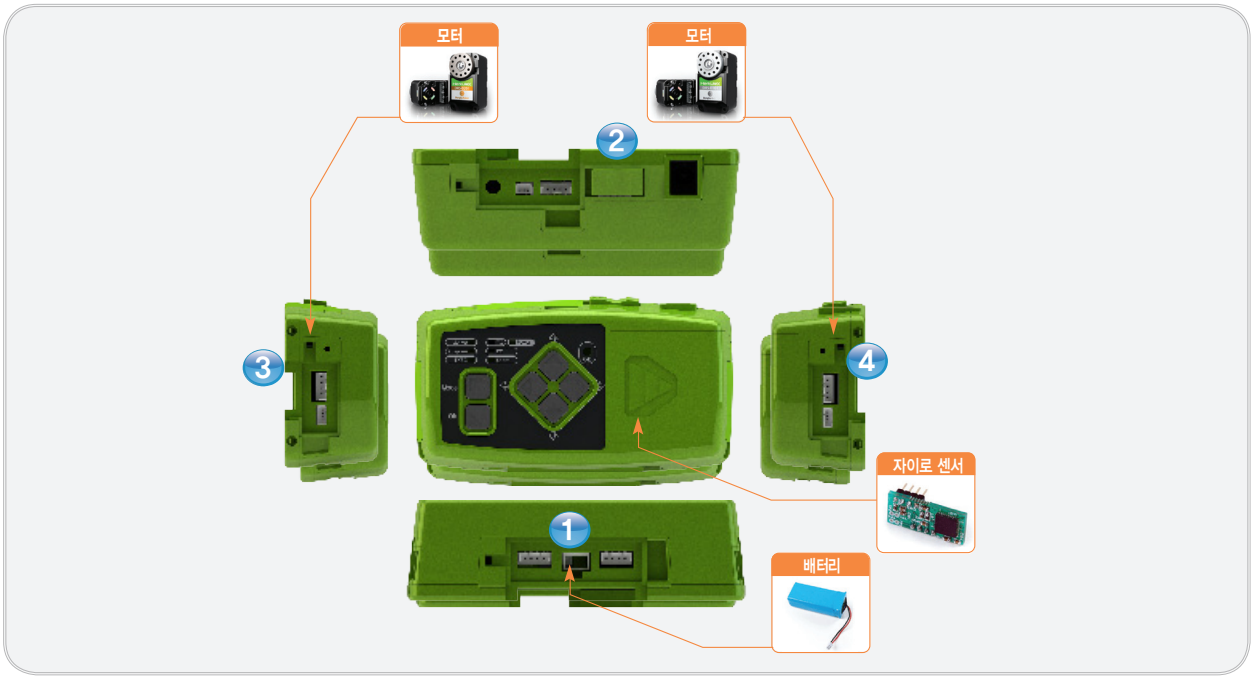
PSD Analog Sensor 는 실시간 거리 측정이 가능합니다. 접근거리별로 모터의 속도를 조절합니다. 대부분 장애물 감지에 쓰이며, 장애물이 가까이 왔을 때 속도를 줄이고, 회피하는 동작에 사용됩니다.

→ 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.

PART 02

Visual Logic 연계된 C 언어 문법 익히기

CH.1 : 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.

CH.2 : C 언어 훑어보기

◆ C 언어 예시

※ 동부로봇 AVR 예제

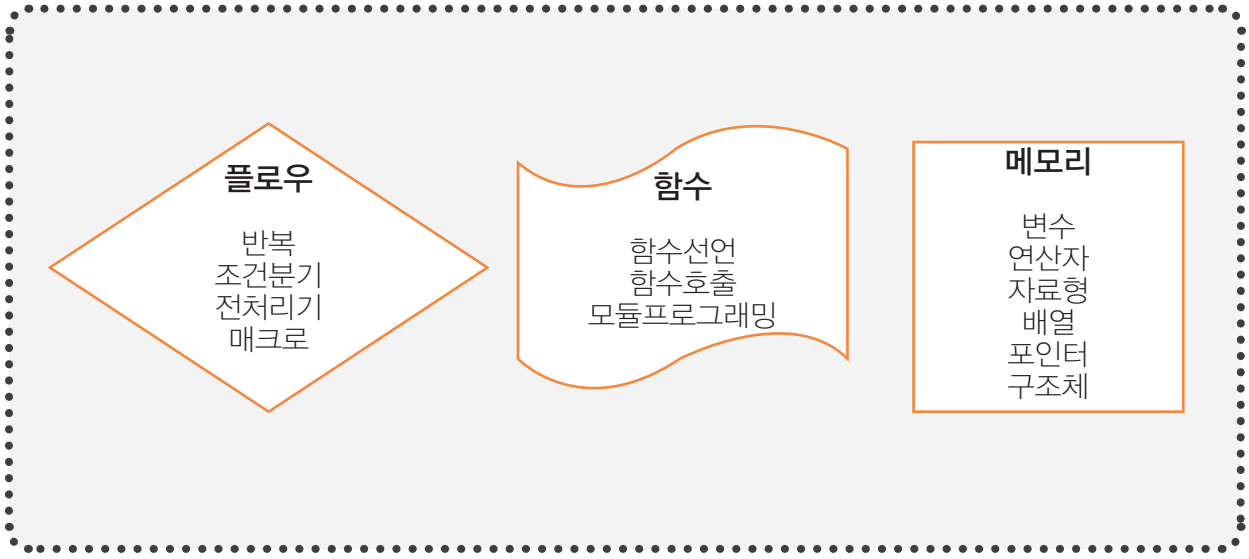
```
#include <avr/io.h>  //(1) 전처리기
#define BTN_MODE      0b00000001
int main(void)  // (2) 함수 - 기능정의
{
        unsigned char ucButton;   // (3) 변수 - 메모리 관리 - 포인터
        PORTB &= ~0b10000000;

        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.

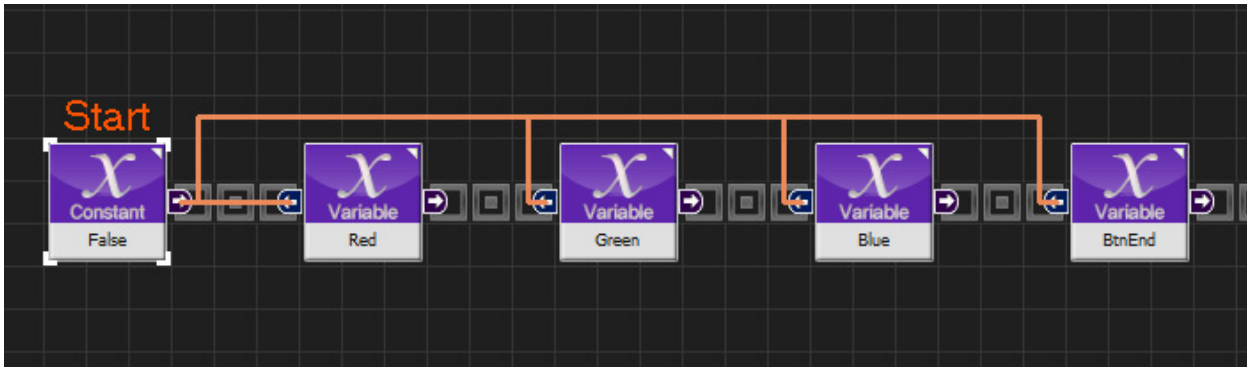
프로그램 작성 및 실행순서는 아래와 같습니다.

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

CH.3 : 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.

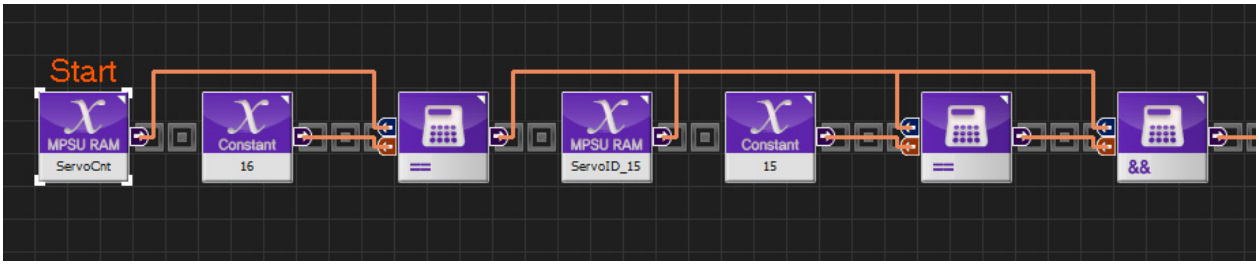
정수형 : char, int, long 3

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
```

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

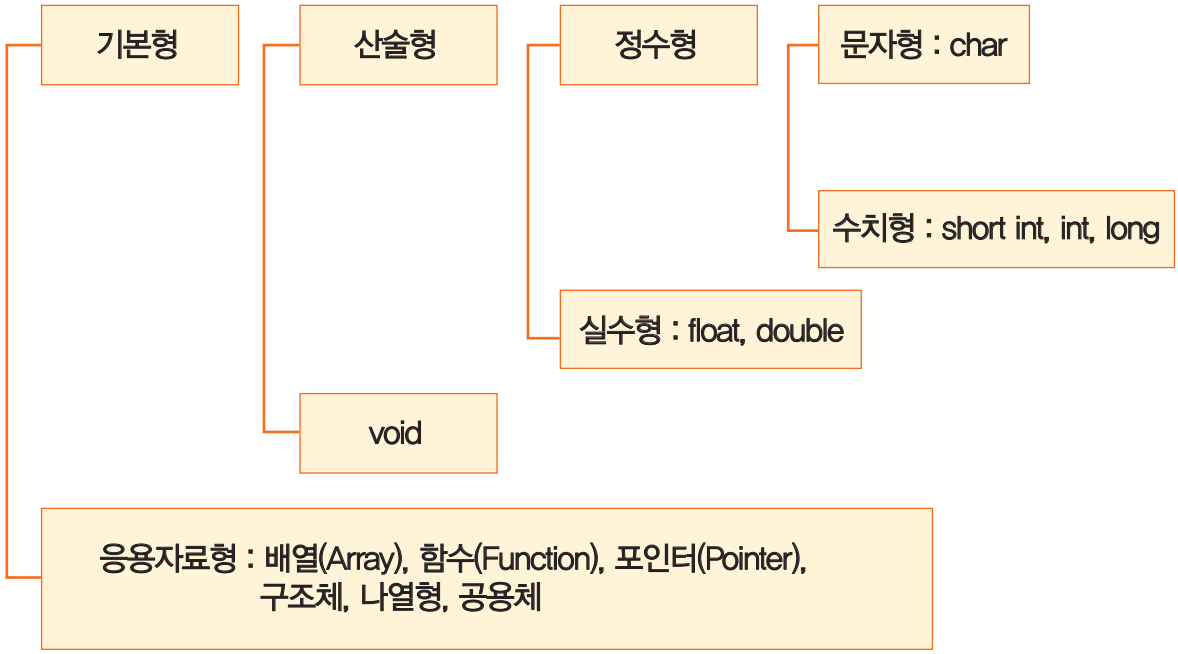
const int MAX = 100 // 변수가 상수가 됨

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.

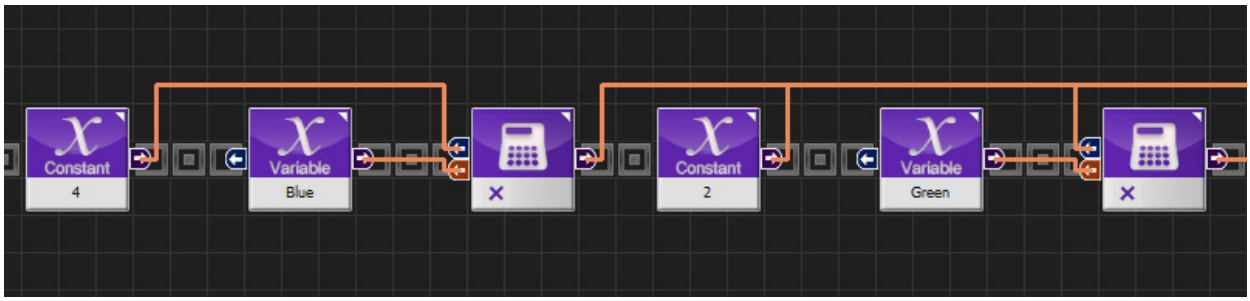
미국 표준 협회 (ANSI) 에 의해 정의, 컴퓨터를 통해서 문자를 표현하기 위한 표준으로서 컴퓨터는 문자를 표현하지 못하기 때문에, 문자와 숫자의 연결 관계를 정의하였습니다. 문자 A 는 숫자 65, 문자 B 는 숫자 66 으로 매칭시킵니다.
문자, 컴퓨터는 2진수밖에 받지 못하기 때문에, 컴퓨터와 사람사이에 약속을 하기 시작하였습니다. 사람은 문자 표현 원하고, 문자와 숫자 사이에 매핑 관계를 유지하기를 원합니다. 에를 들어 A=65, B = 55 라고 약속, 사람이 A 입력, 컴퓨터는 65라고 저장합니다.

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			

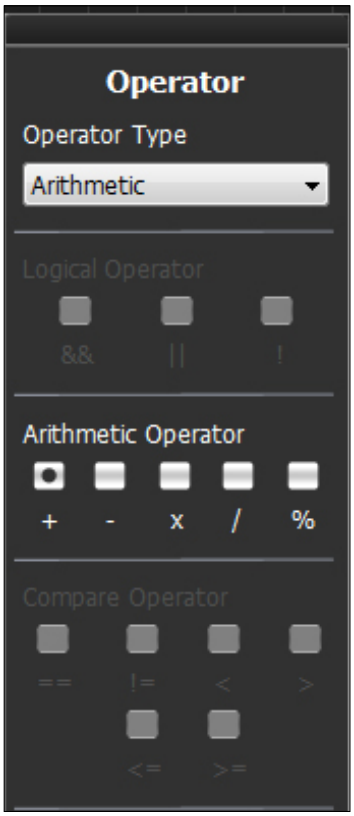
◆ 연산자

ex) ledbutton.dts 중 일부



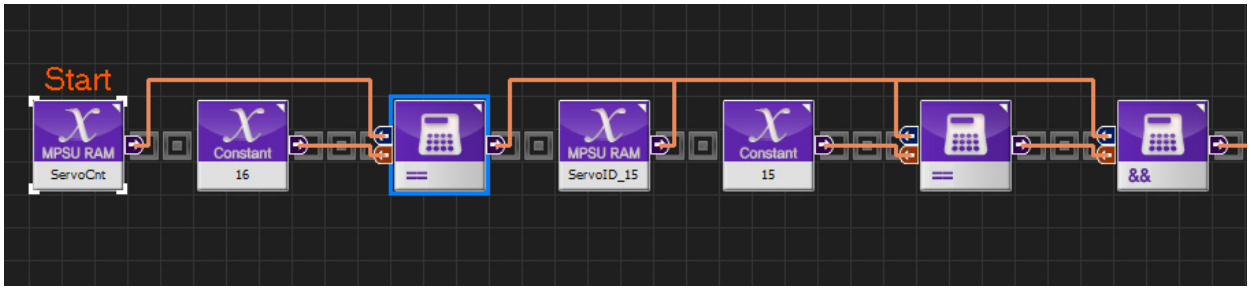
C-like 보기

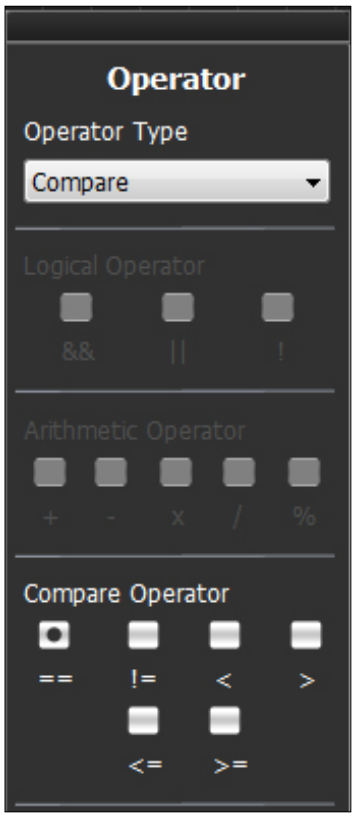
```
33  led( ( ( ( 4 * Blue ) + ( 2 * Green ) ) + Red ) )
34  if( ( ( MPSU_ButtonStat == 0x00 ) && BtnEnd ) )
35  {
36          BtnEnd=false
37  }
38  else
39  {
40  }
```

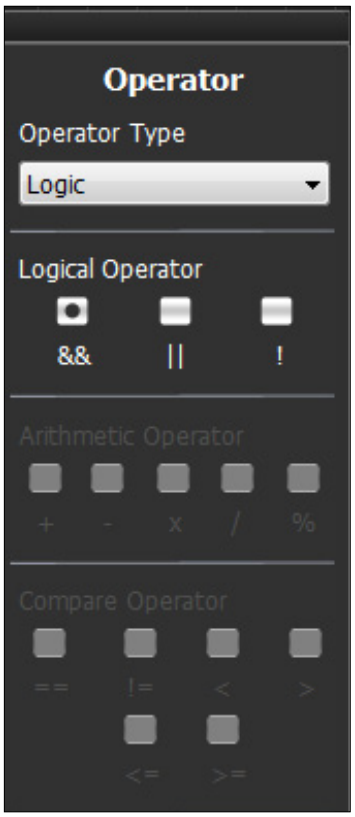


기능별 종류	연산자
산술 연산자	+ – * / %
부호 연산자	+ –
대입 연산자	= 복합 대입 연산자
관계 연산자	== != 〈= 〈 〉= 〉
증감 연산자	++ ––
포인터 연산자	* & []
구조체 연산자	. –〉
논리 연산자	|| && !
비트 연산자	| & ~ 〉〉 〈〈
삼항 조건 연산자	? :
쉼표 연산자	,
sizeof 연산자	sizeof
캐스트 연산자	(type) type()
괄호 연산자	()
C++ 연산자	new delete :: .* –〉*

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

CH.4 : 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.

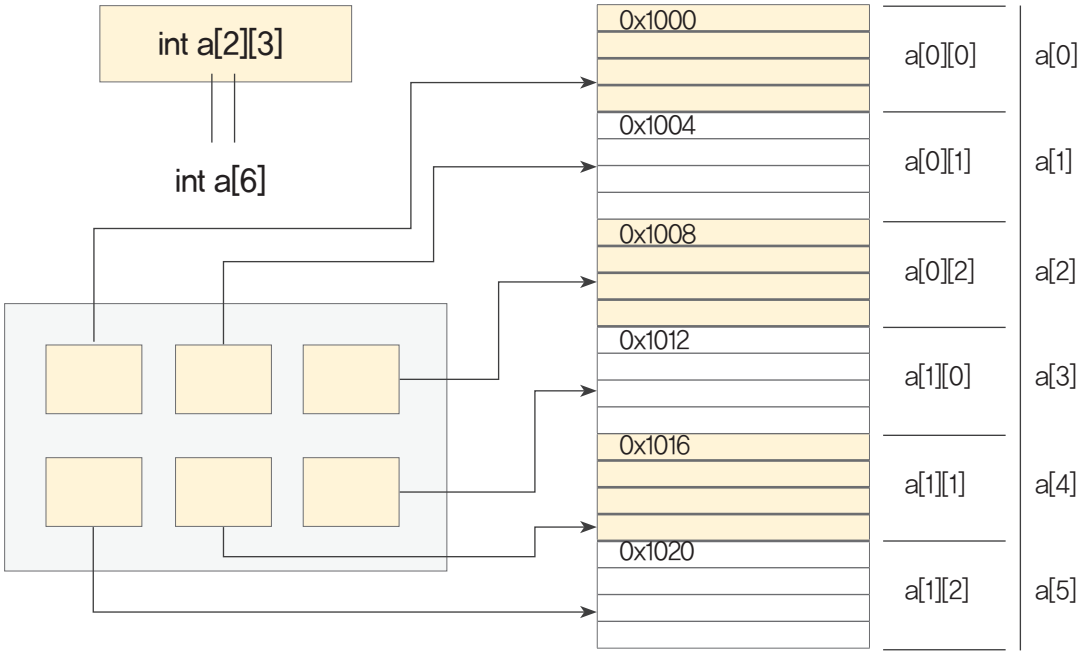
Int a[2][3]; → 2차원 접근 a[0][0]. 세로가 2, 가로가 3

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.

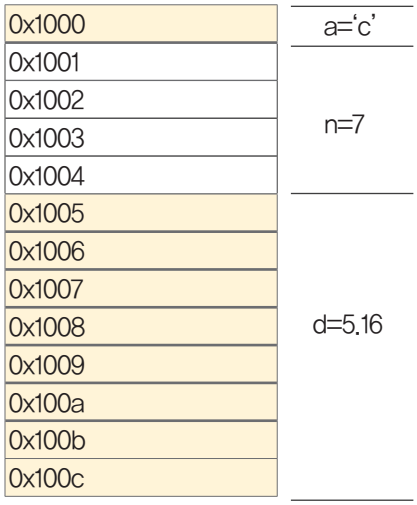
– “포인터”를 흔히 “포인터 변수”라고 합니다.

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.

CH.5 : 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)
```

CH.6 : 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

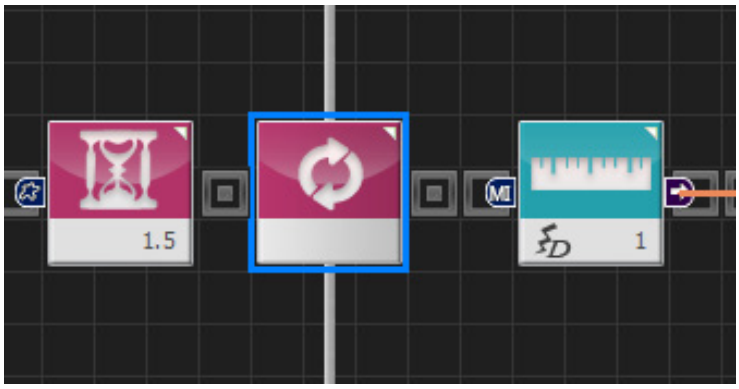
- 개념
  - 기억공간에 저장될 데이터의 집합을 정의한다.
  - 열거형 멤버들은 정수형 상수로 취급 당한다.
  - 프로그램 가독성을 높일 수 있다.
  - 특정 정수값에 의미를 부여할 수 있다.

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

CH.7 : 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

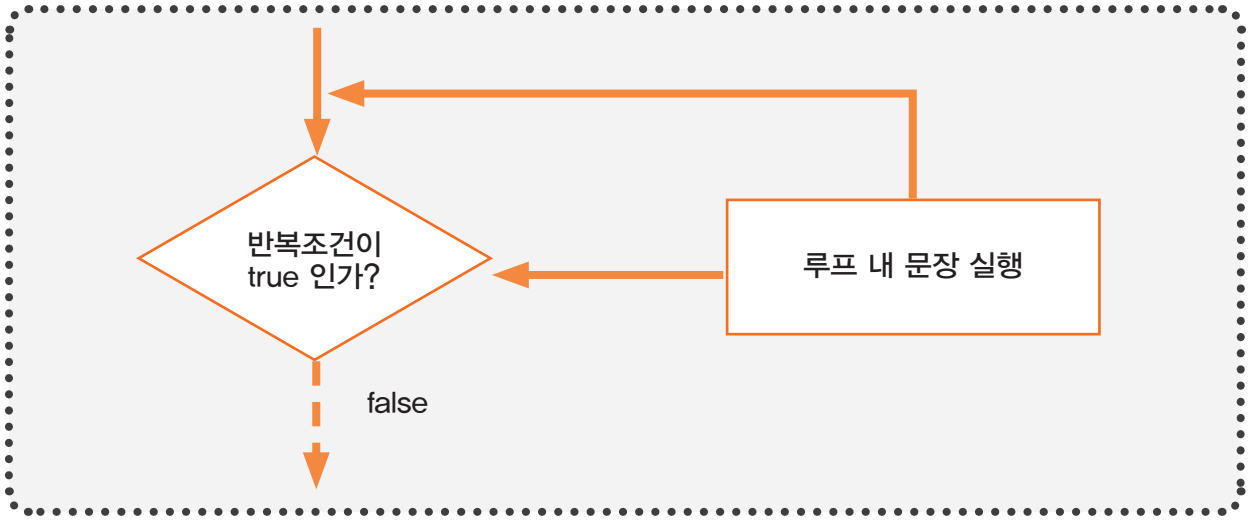
– while 문에 의한 반복

– do~while 문에 의한 반복

– for 문에 의한 반복

한가지만 다 이해해도 모두 잘 이해할 수 있습니다.

while 문의 순서도

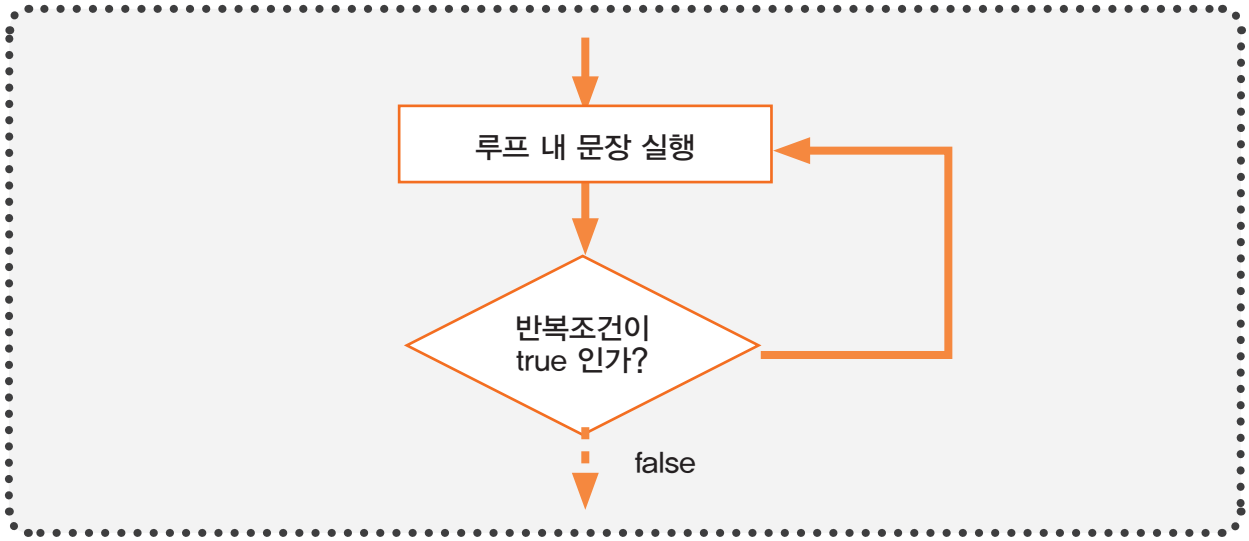


일반형식

```
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

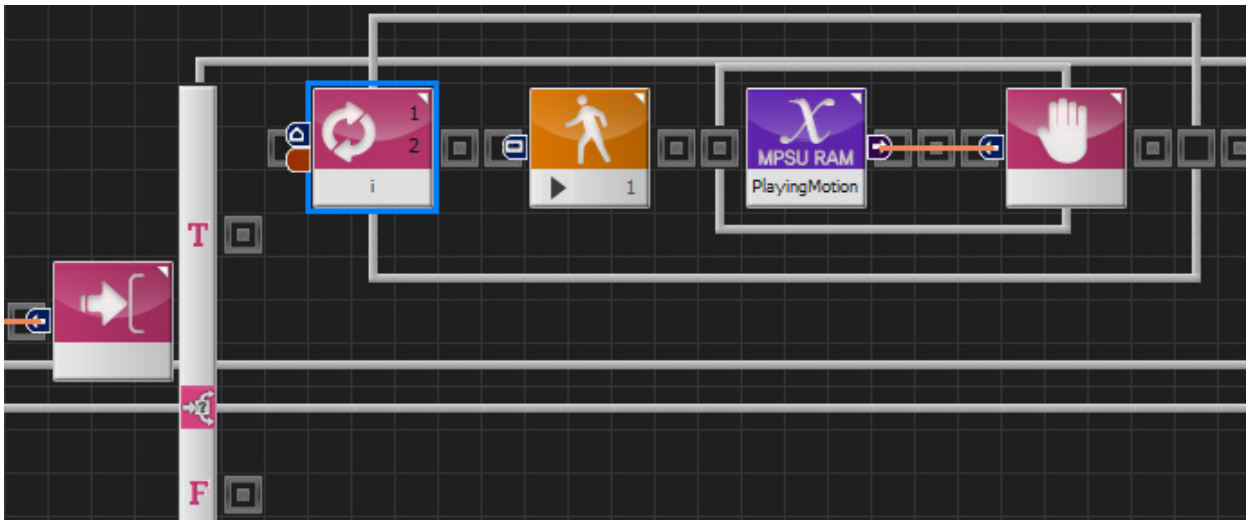


일반형식

```
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 )
```

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.

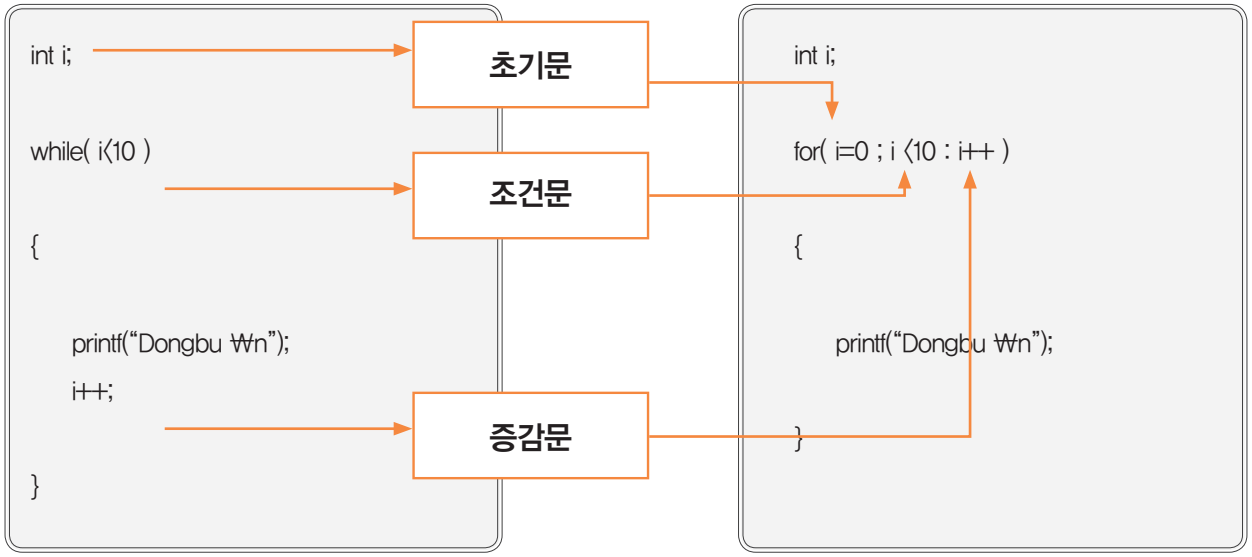
다중 for 문

```
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

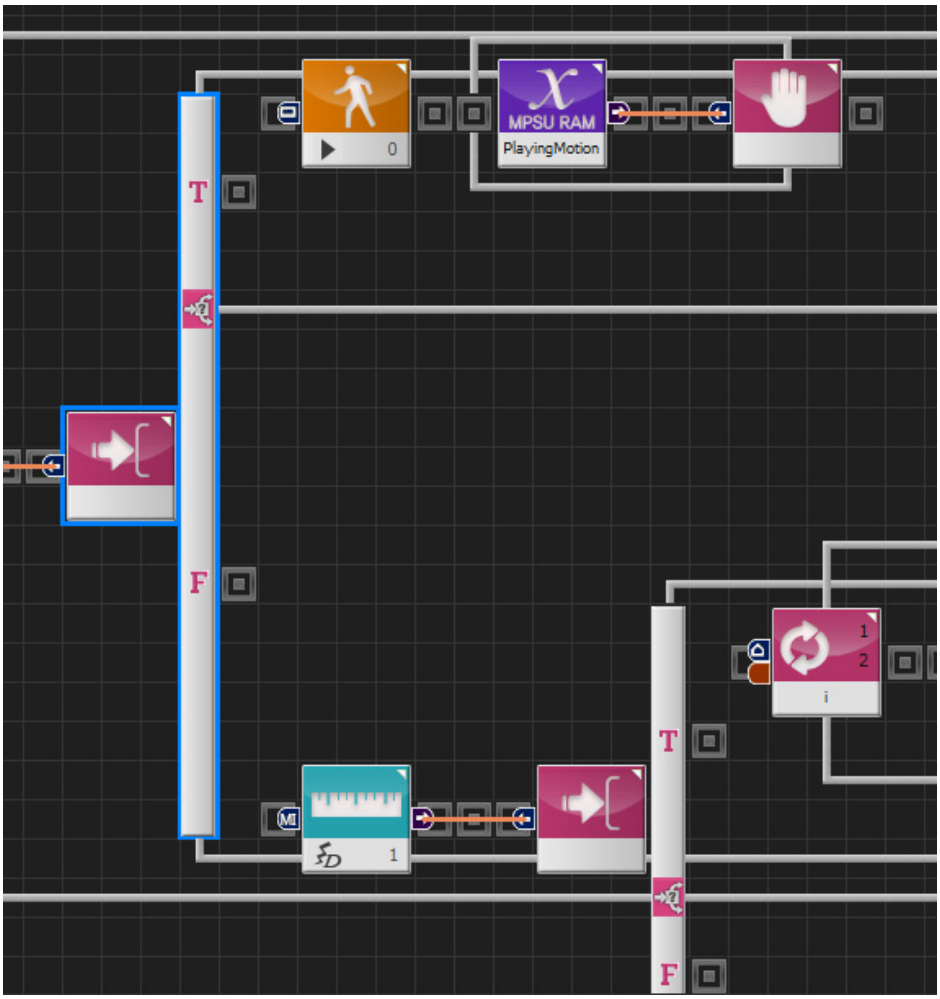
for 와 while 문 비교




```
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
}
```

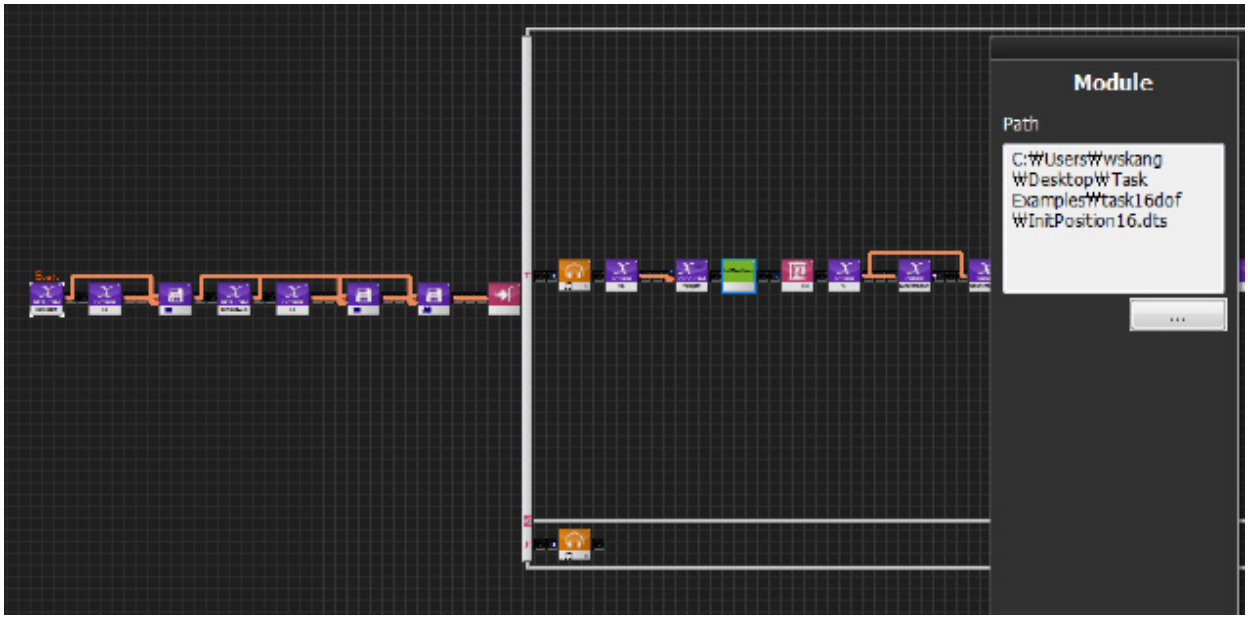
```
if(조건 A)
{
    조건 A 만족시 실행
}
else if(조건 B)
{
    조건 B만족시 실행
}
else if(조건 C)
{
    조건 C 만족시 실행
}
else
{
    조건 ABC 불만족시 실행
}
```

```
switch(n)

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

default:
```

CH.8 : 함수

◆ 메인함수

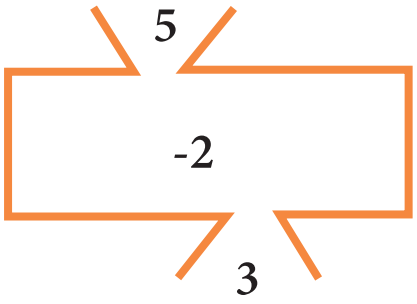
ex) Remocon16.dts 중 일부

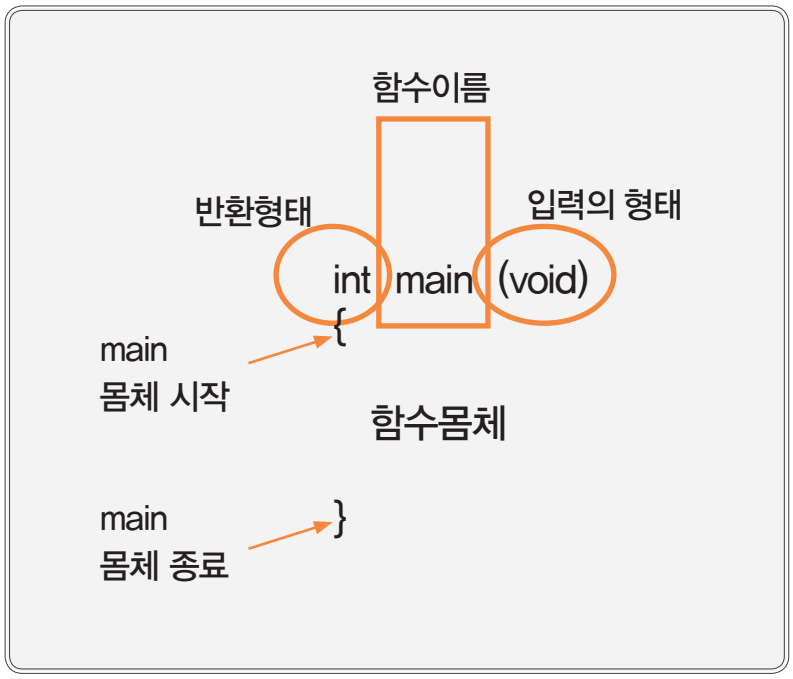


C-like 보기

```
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

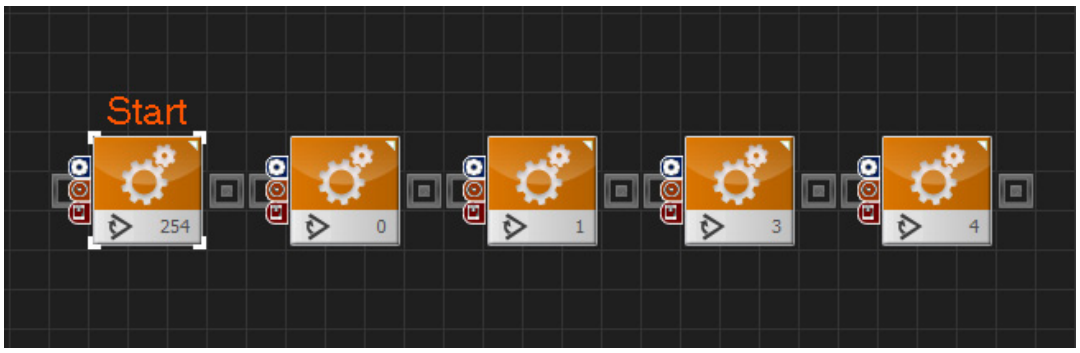
들째, 입력의 형태가 있습니다. → (void)

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

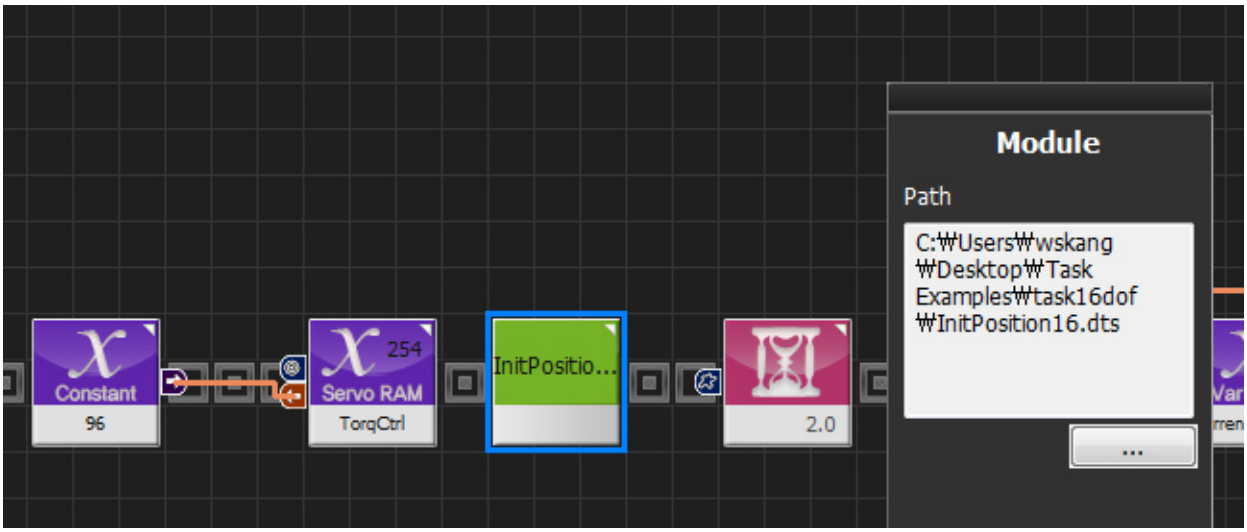
전역변수는 프로그램이 시작하면 올라갔다가, 종료되어야 내려갑니다.

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

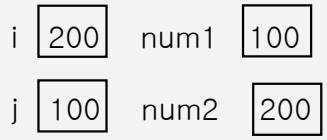
– 참조(참조를 가능테 하는 주소값)를 인자로 전달하는 형태의 함수 호출입니다.

– 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.

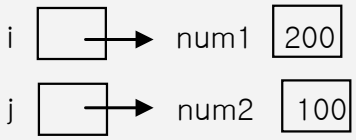
※ 값에 의한 호출 예제

```
#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 03

AVR 로봇 제어기 Firmware 익히기

CH.0 : 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.

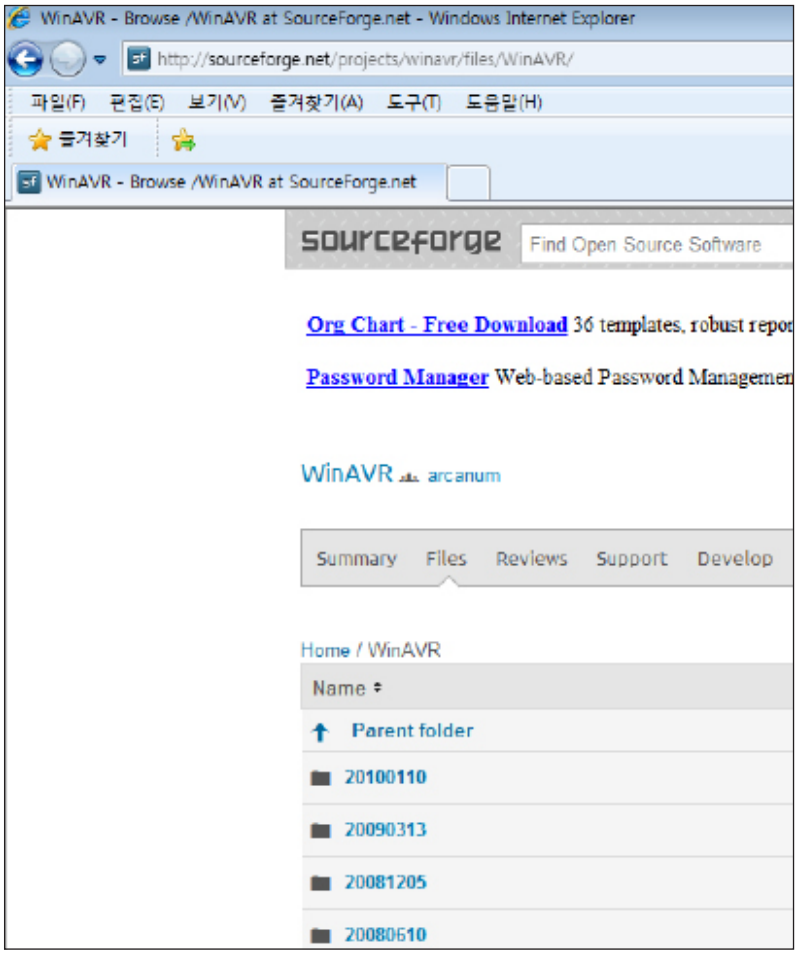
WinAVR 설치하기



01 URL 입력

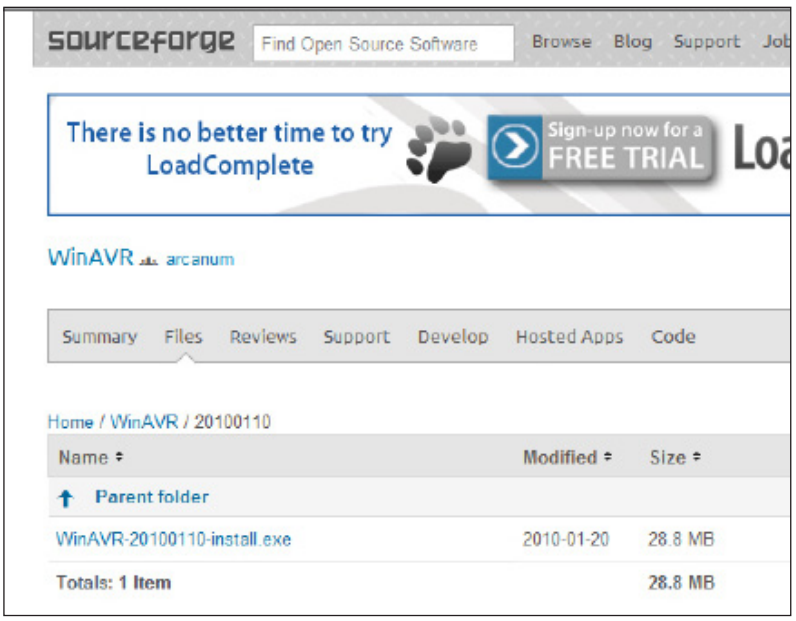
주소창에 URL을 입력합니다.

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



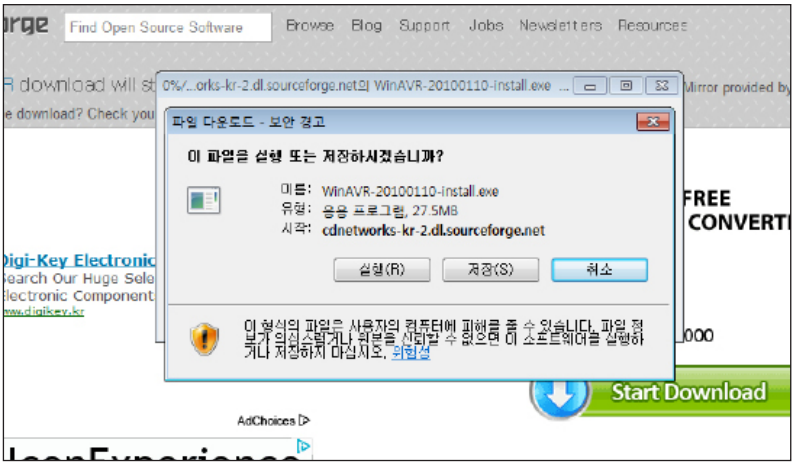
02 다운로드 링크

20100110의 우측 체크박스를 클릭합니다.



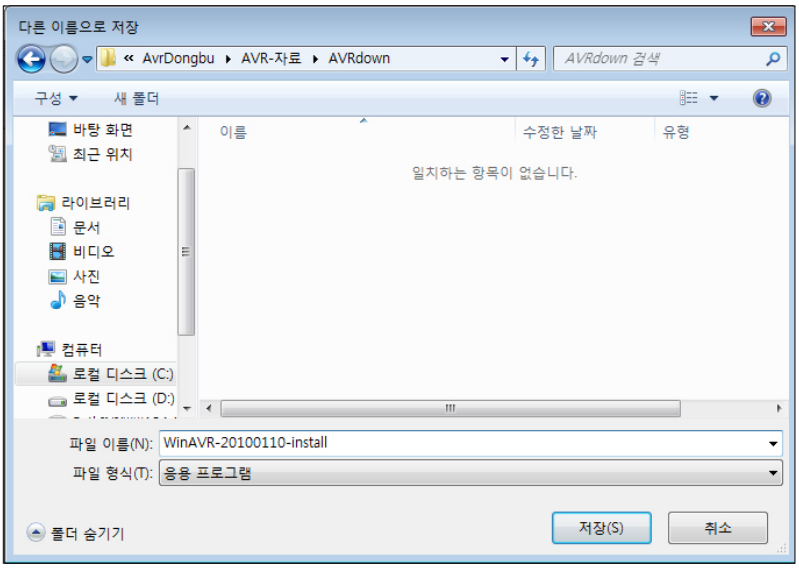
03 다운로드

WinAVT-20100110-install.exe의 체크박스를 클릭합니다.



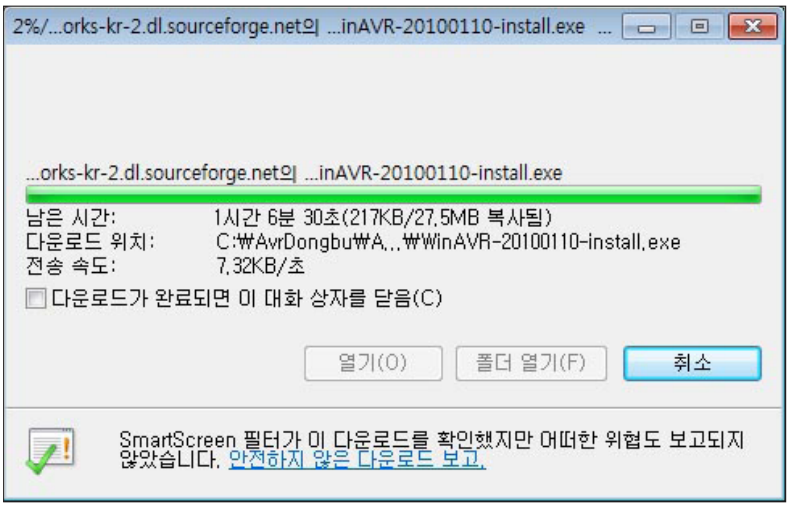
04 설치파일 저장

"저장"버튼을 클릭합니다.



05 저장 폴더 지정

설치파일을 저장할폴더 지정 후 "저장" 버튼을 클릭하세요



06 프로그램 다운로드

"다음" 버튼을 클릭합니다.



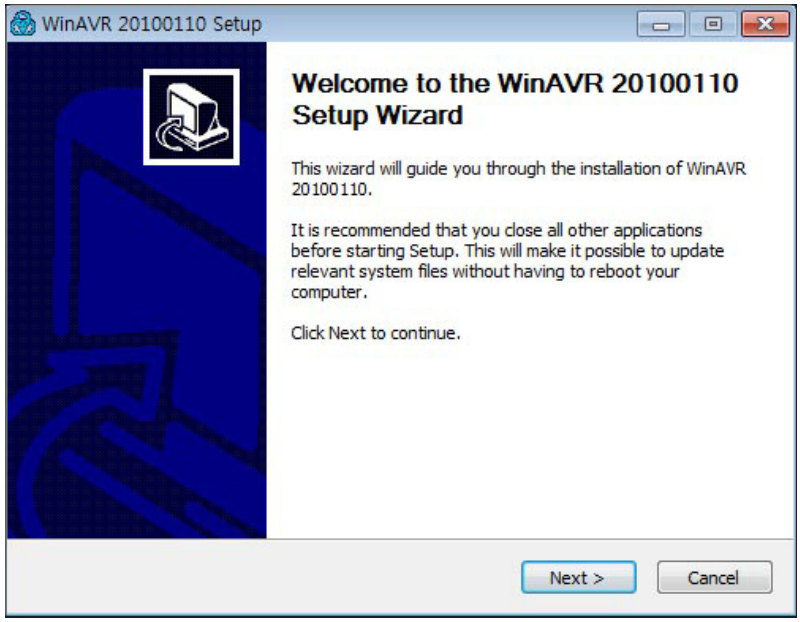
07 설치 폴더를 선택

생성된 아이콘을 더블클릭 하세요



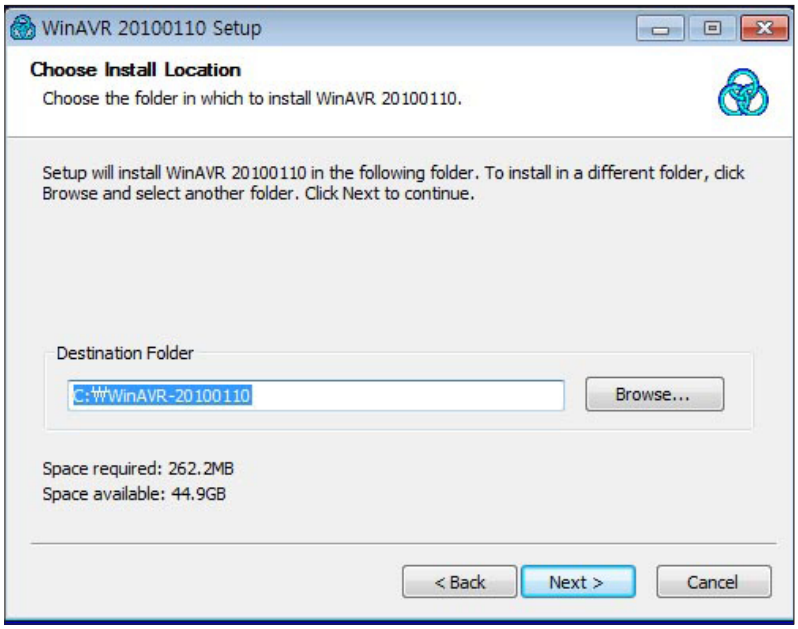
08 설치 확인

언어를 "English"로 선택 후 "OK"를 클릭합니다.



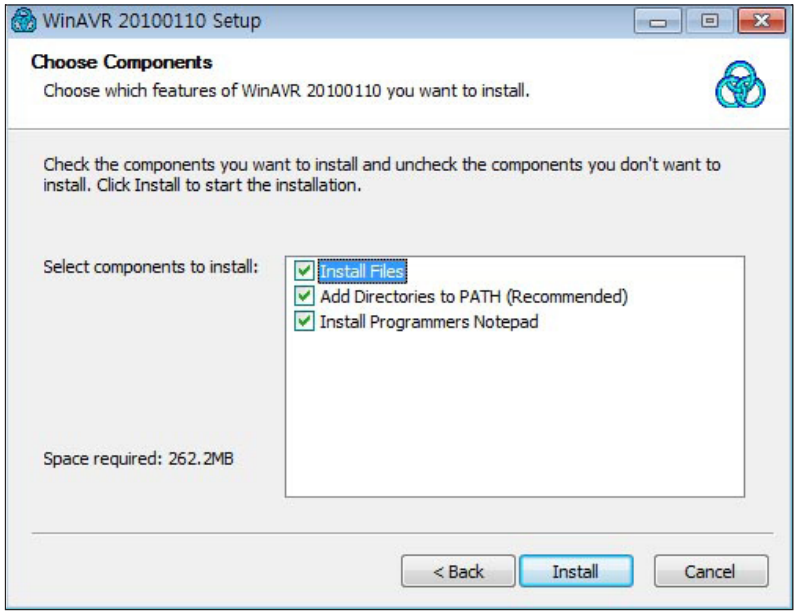
09 설치 마법사 시작

설치 마법사가 실행됩니다. "NEXT" 를 클릭하세요.



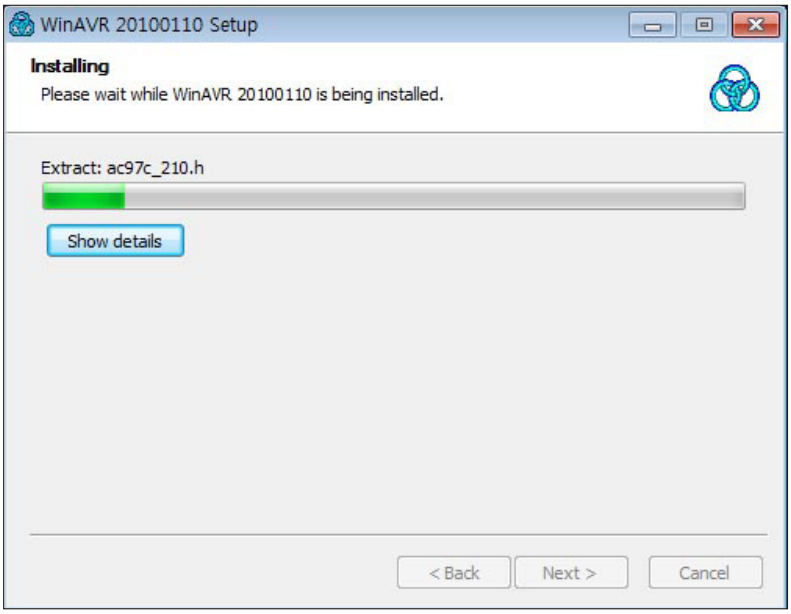
10 설치 폴더 확인

"NEXT" 를 클릭하세요



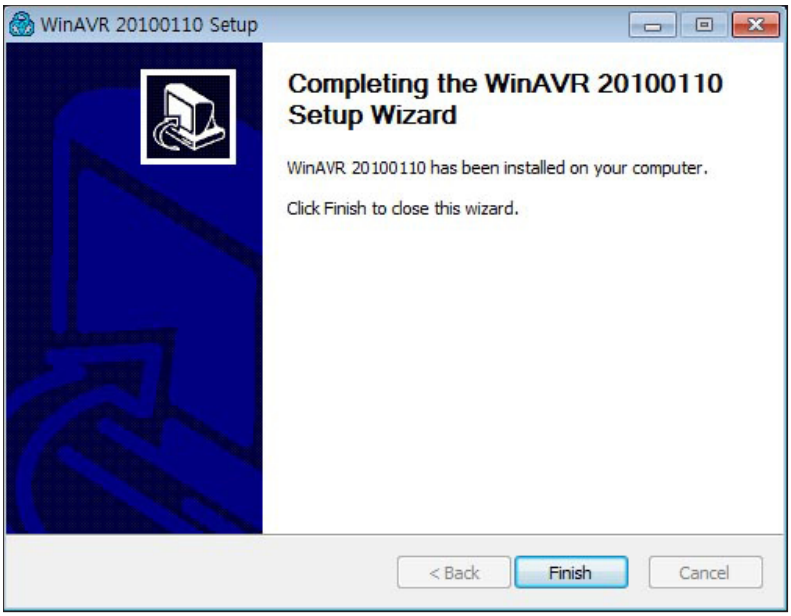
11 구성 요소 확인

"Install" 을 클릭하세요.



12 설치 시작

프로그램 설치를 시작합니다.



13 설치 완료

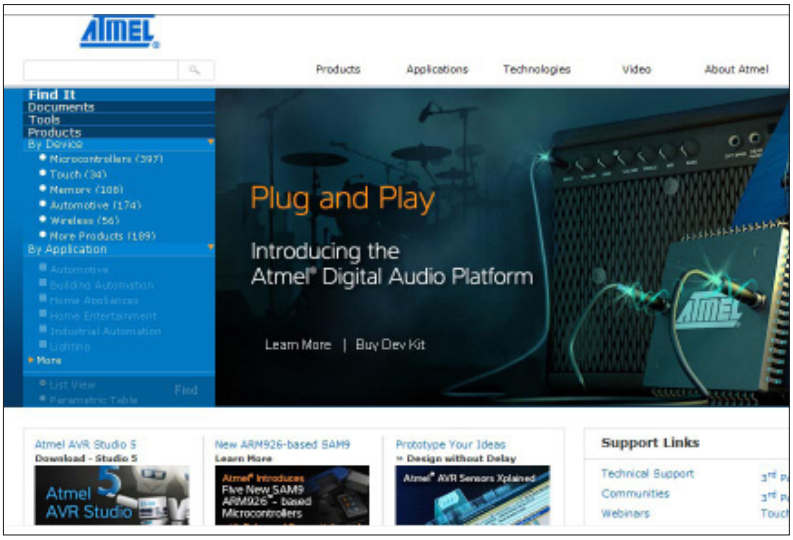
"Finish" 를 클릭하면 설치가 완료 됩니다.



AVR Studio 설치하기

01 URL 입력

주소창에 www.atmel.com URL을 입력합니다.



02 다운로드 링크

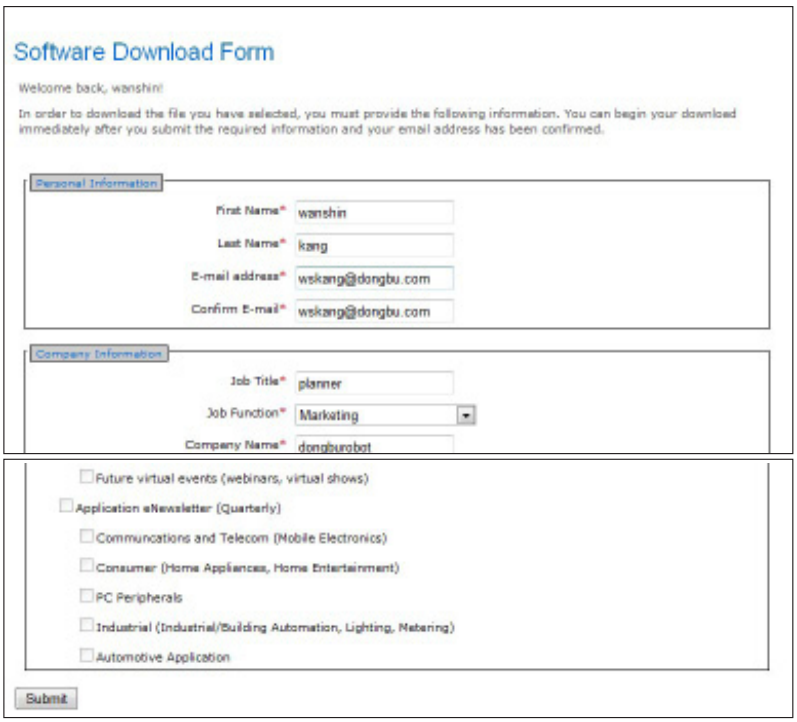
"Support Links"의
"Download AVR Studio 5"를 클릭합니다.





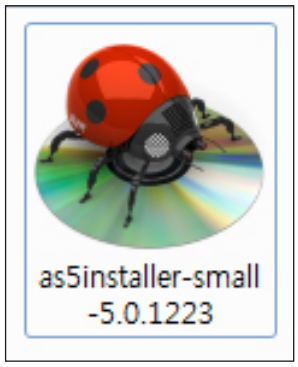
03 소프트웨어 선택

"AVR Studio 5.1 Installer–Full"을 클릭합니다.



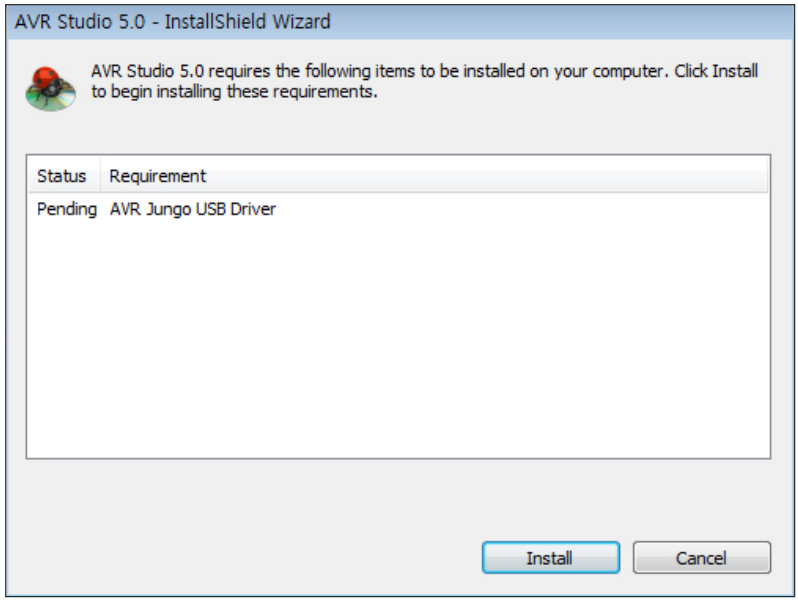
04 소프트웨어 다운로드 양식

사용자 정보를 입력한 뒤 submit을 클릭합니다.



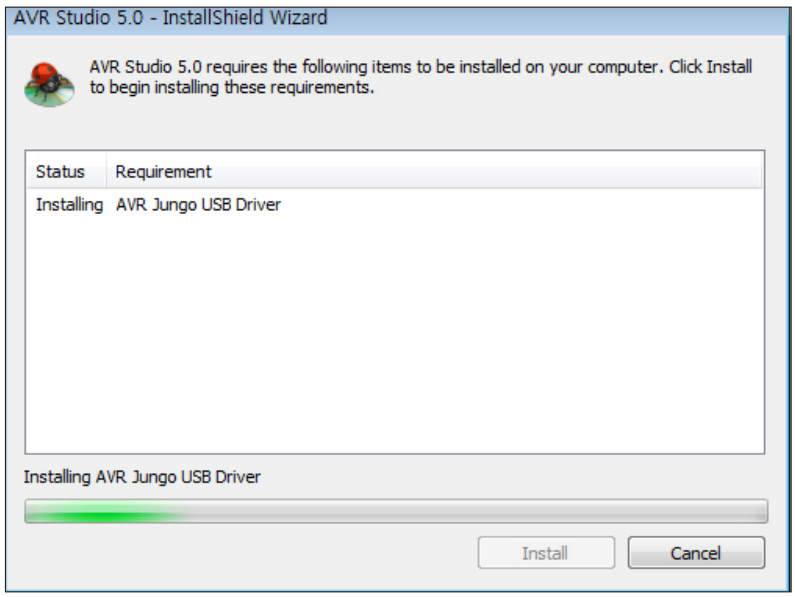
05 설치 프로그램 실행

아이콘을 더블클릭합니다.



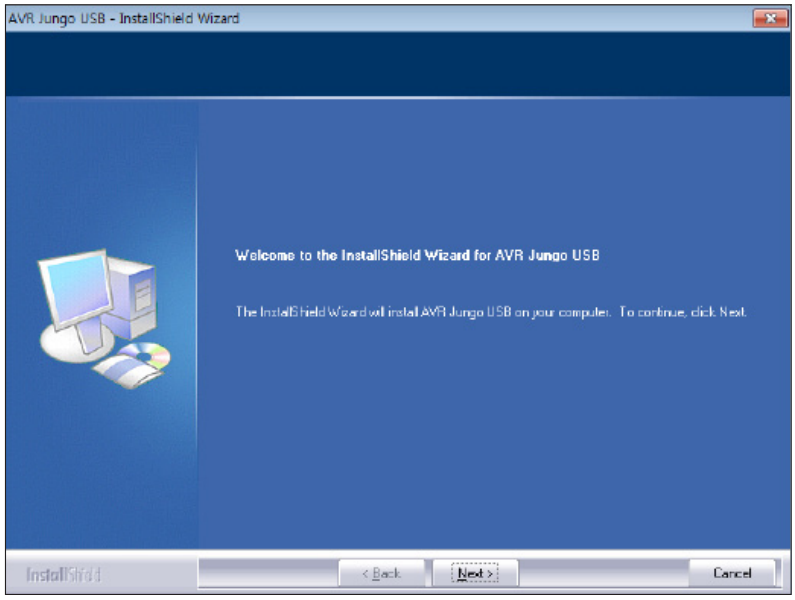
06 설치 시작

"Install" 버튼을 클릭하세요



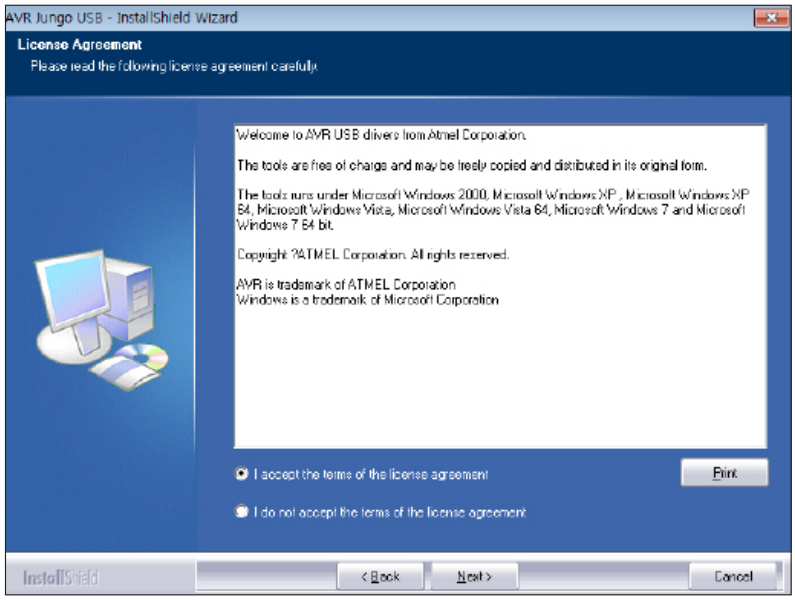
07 설치

프로그램이 설치됩니다.



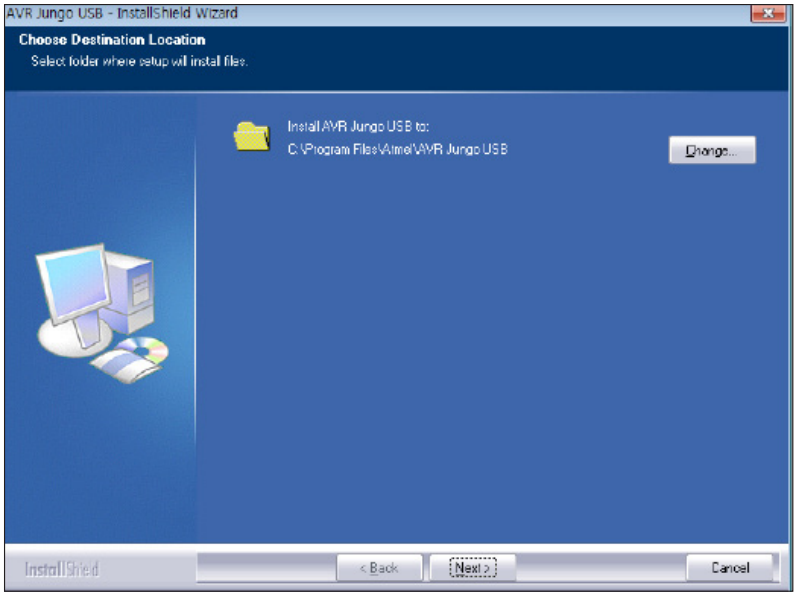
08 AVR Jungo USB 설치 확인

"NEXT"를 클릭하세요



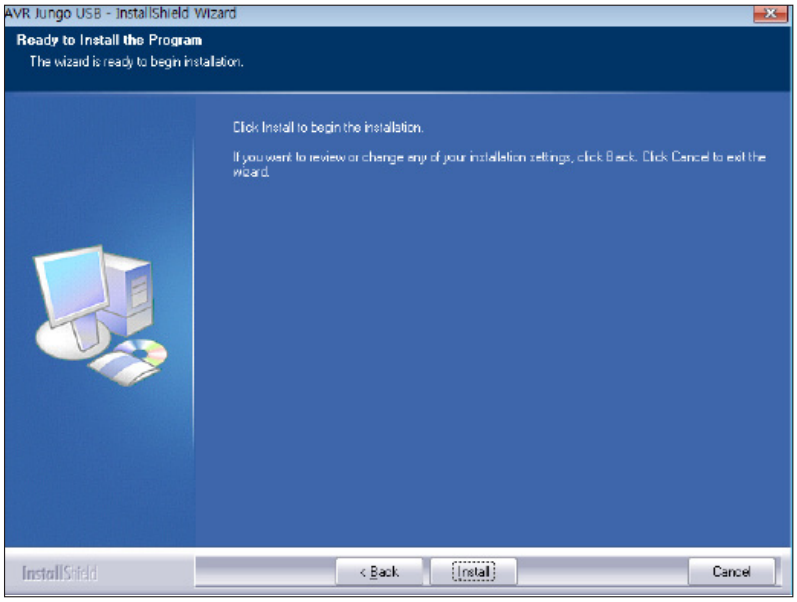
09 약관 동의

"I accept"를 선택하고 "Next"를 클릭하세요



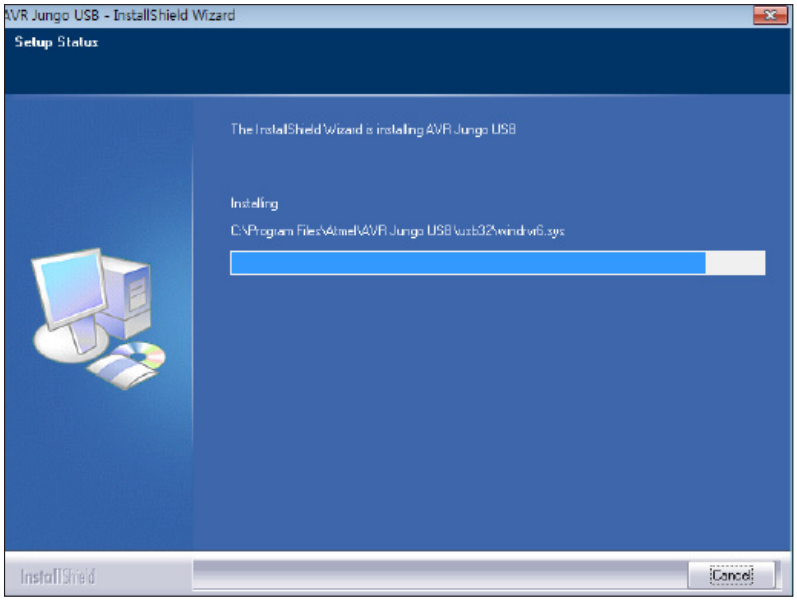
10 설치 폴더 선택

프로그램 설치폴더를 확인 후 “Next”를 클릭합니다.



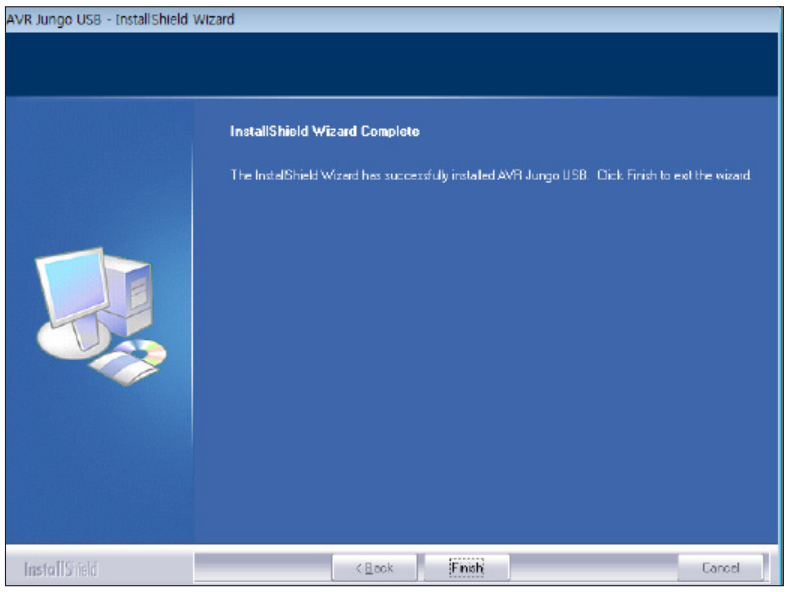
11 설치 시작

“Install”을 클릭하면 설치를 시작합니다.



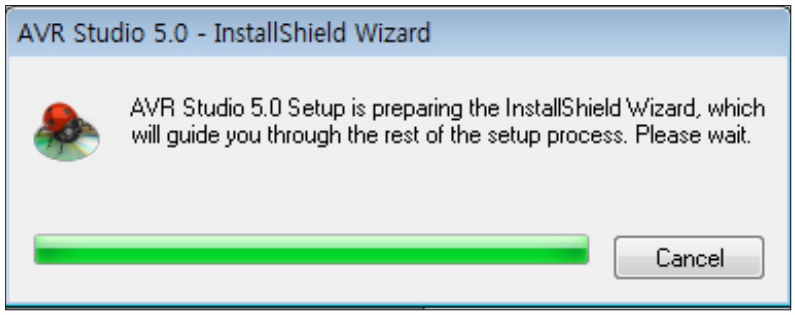
12 설치

프로그램이 설치됩니다.



13 설치완료

"Finish"를 클릭하면 AVR Jungo USB설치가 완료됩니다.



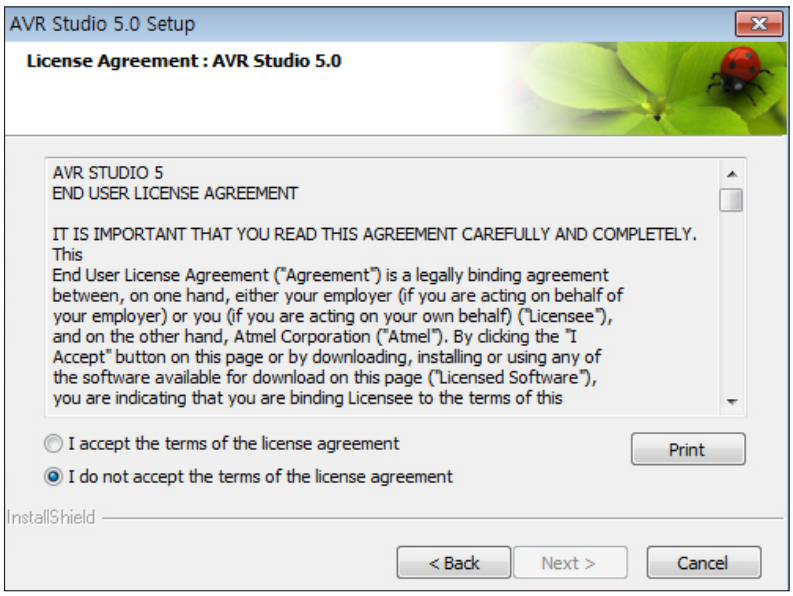
14 AVR Studio 5.0 설치 준비 완료

AVR Studio 5.0을 설치할 준비가 완료되었습니다.



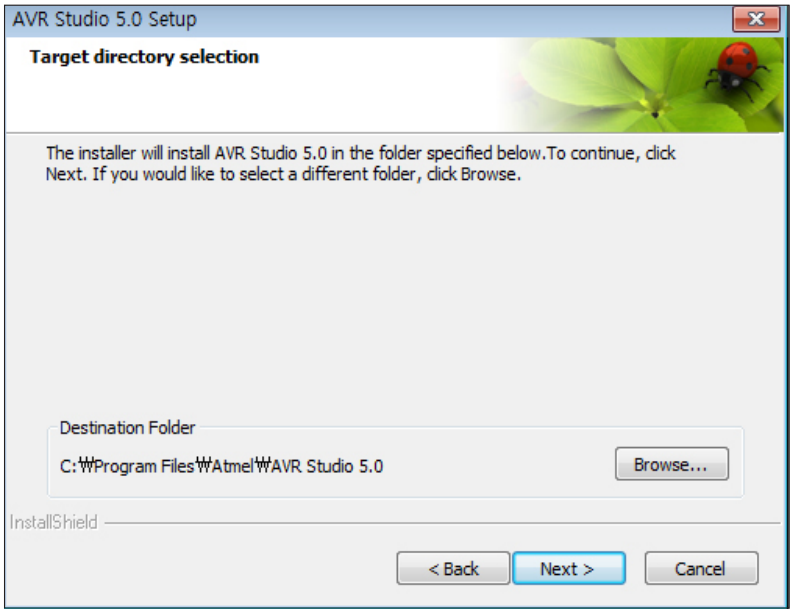
15 설치 마법사 시작

"Next"를 클릭합니다.



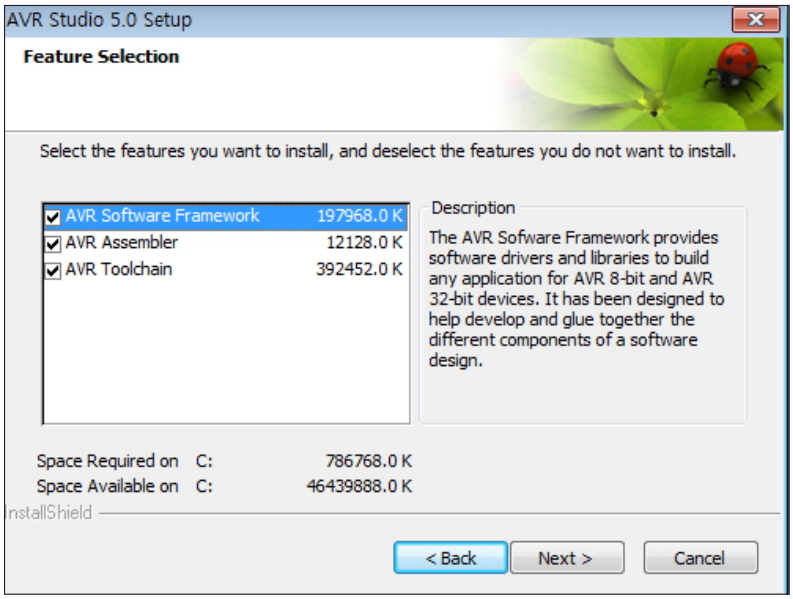
16 약관 동의

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



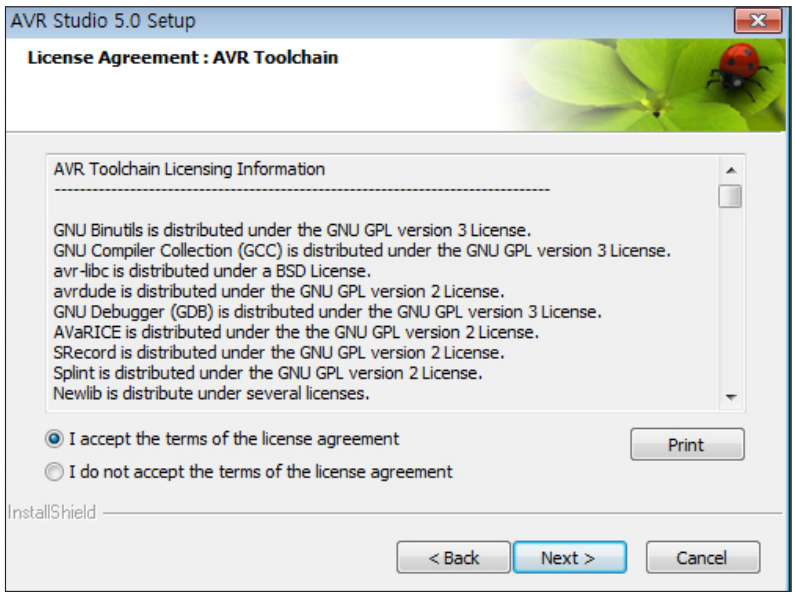
17 설치 폴더 확인

설치 폴더 확인 후 "Next"를 클릭합니다.



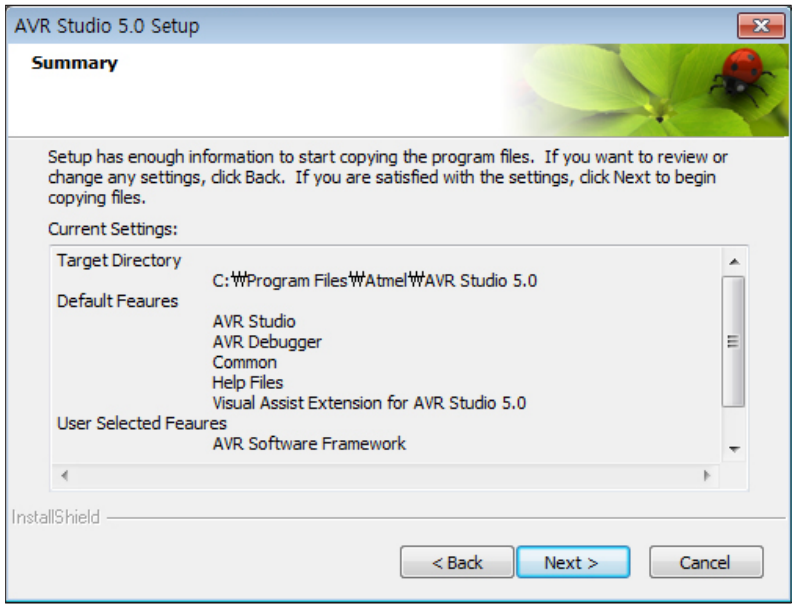
18 설치 기능 선택

설치할 기능 선택후 "Next"를 클릭합니다.



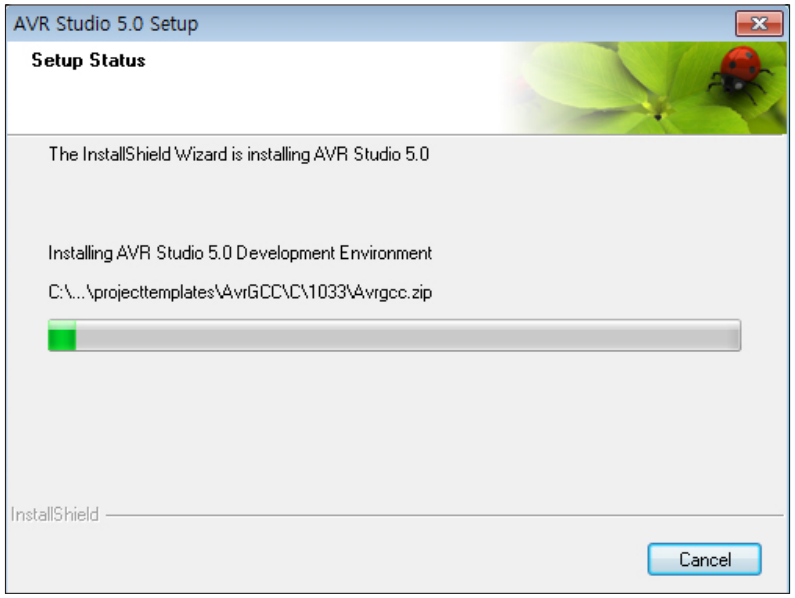
19 약관 동의

약관 동의 후 “Next”를 클릭합니다.



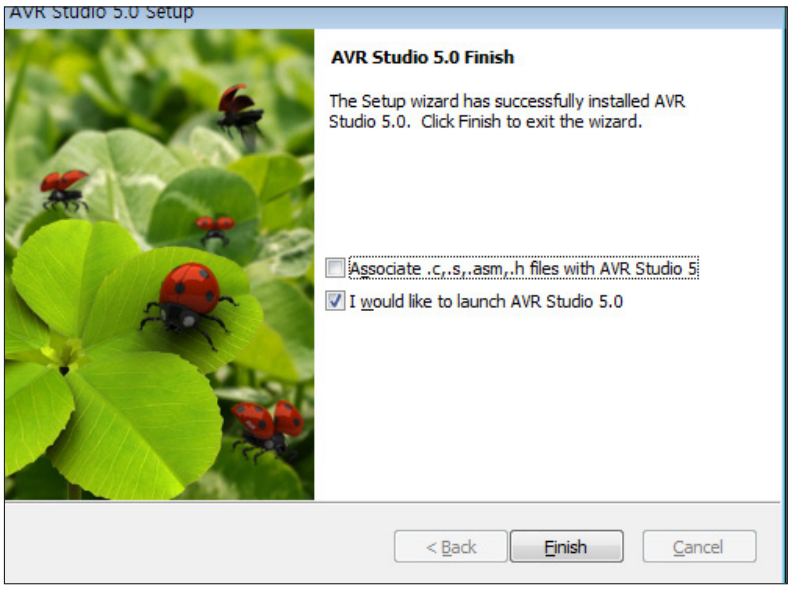
20 설치 내용 확인

설치할 내용 확인 후 “Next”를 클릭합니다.



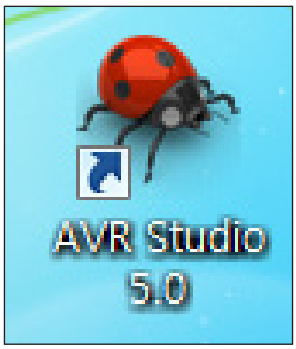
21 설치 시작

설치가 시작됩니다.



22 설치 완료

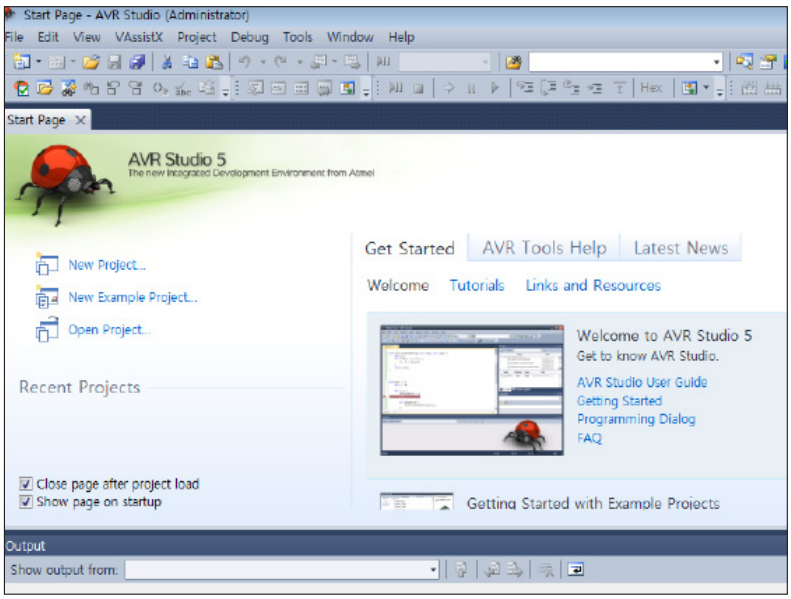
설치가 완료되었습니다.
"I would like to launch AVR Studio 5.0"을 체크한 뒤 "Finish"를 클릭하면 프로그램이 실행됩니다.



프로젝트 만들기 및 코딩

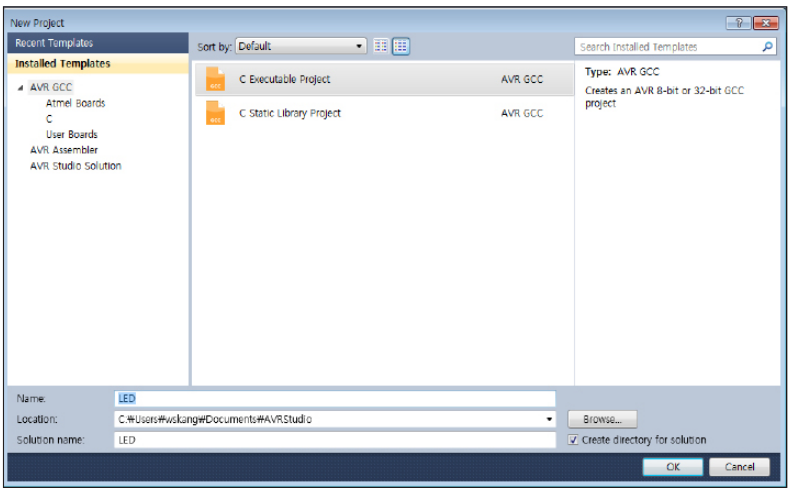
01 실행

AVR Studio 를 실행합니다.



02 New Project

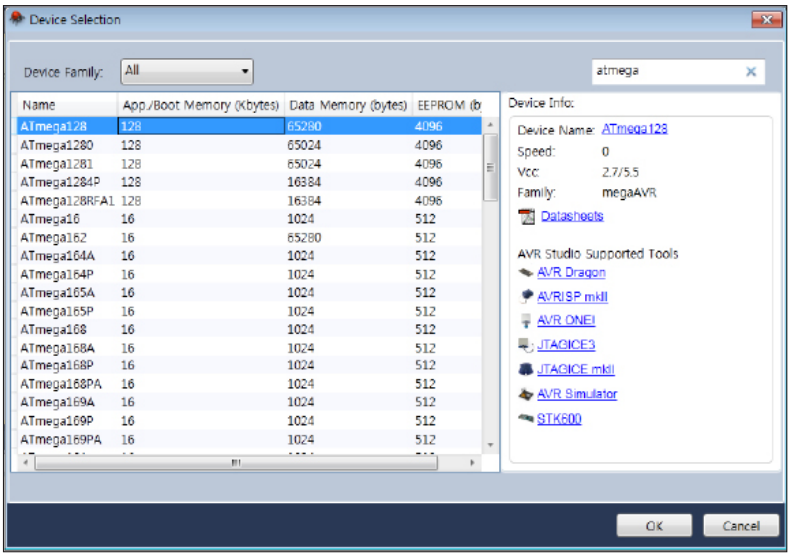
New Project 를 클릭합니다.



03 GCC 선택

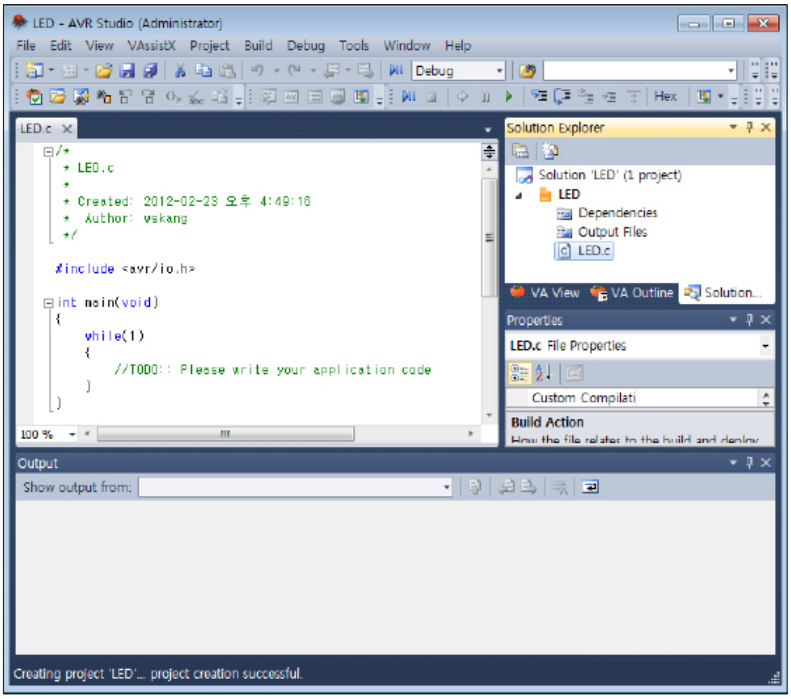
C Executable Project 를 선택합니다.

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



04 ATmega 선택

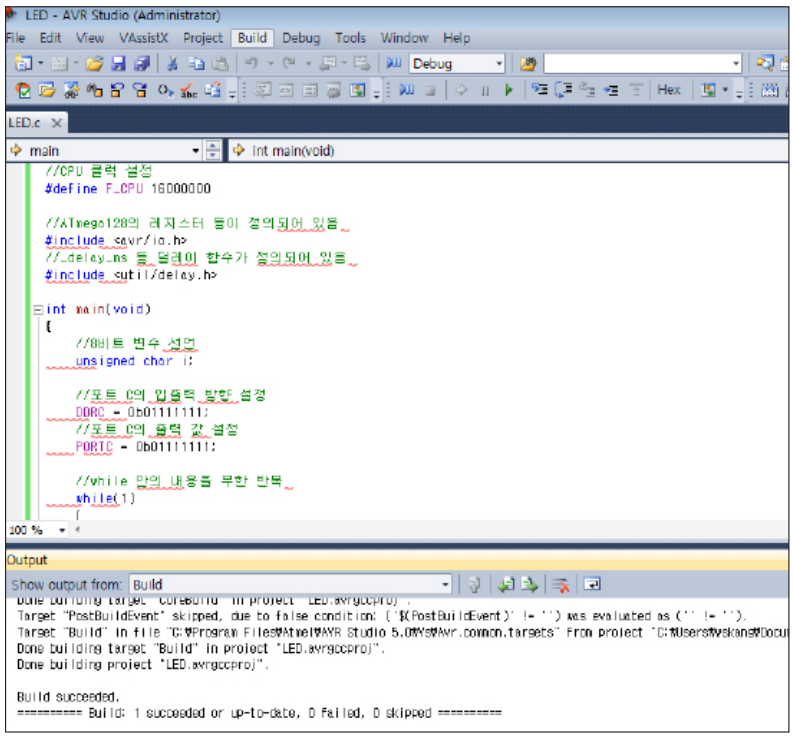
우측 상단에 atmega 를 검색하여, 리스트중에 ATmega128 을 선택합니다.

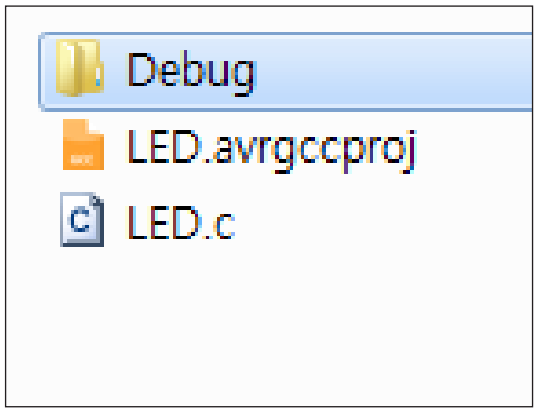


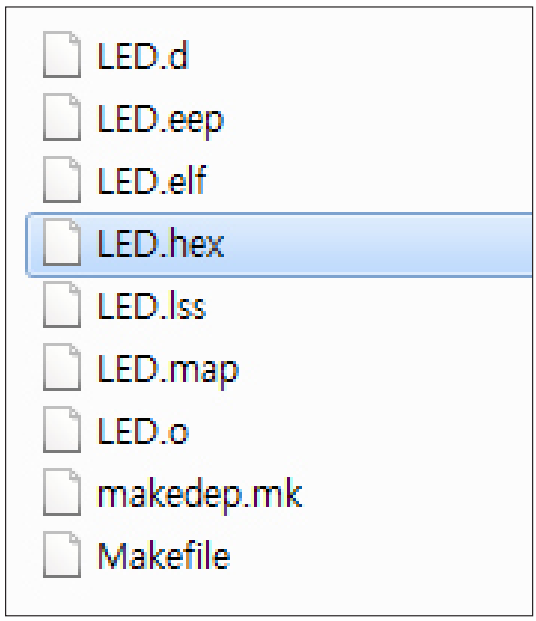
05 Editor

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.

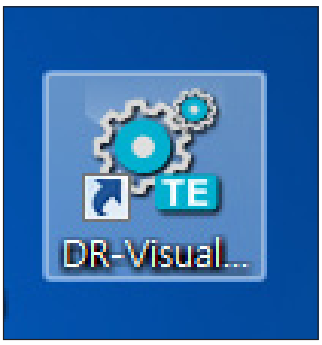
02 Debug 폴더

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.



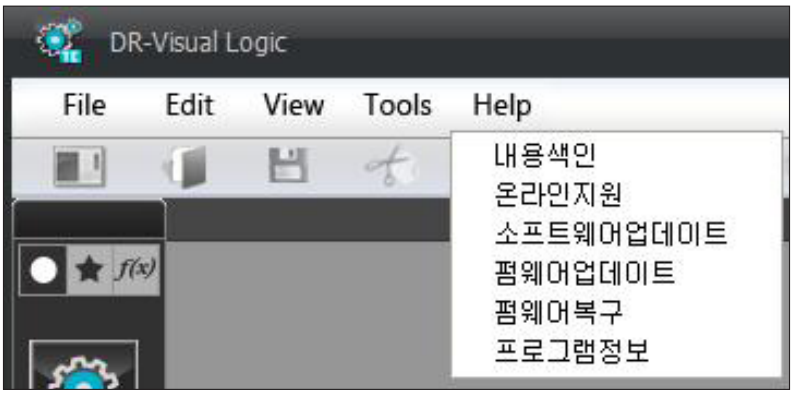
04 Visual Logic

DR-Visual Logic 을 실행시킵니다.



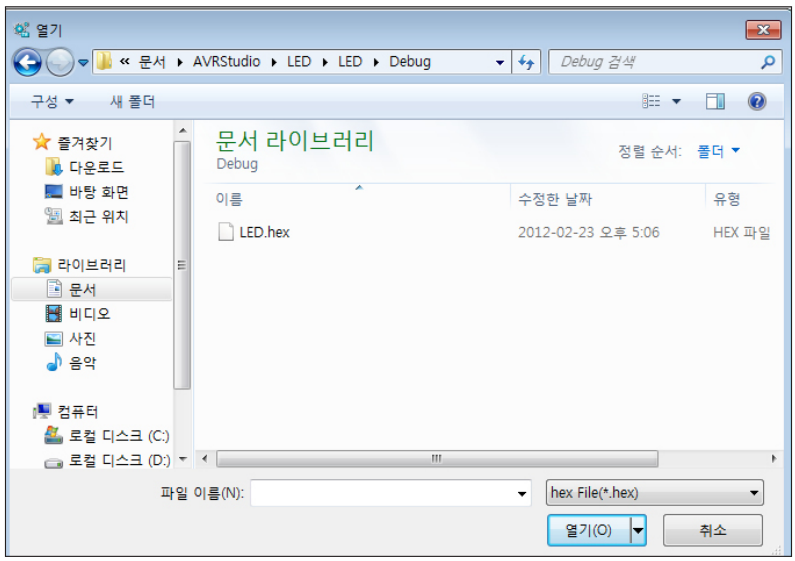
05 연결하기

COM 번호를 설정하고 연결하기를 클릭합니다.



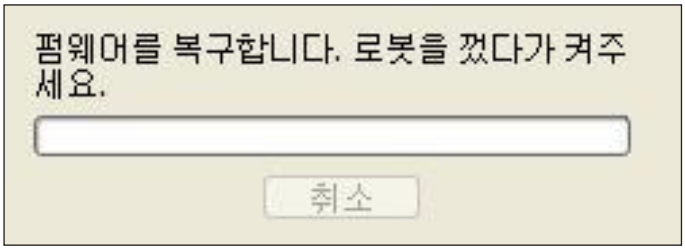
06 펌웨어 복구

도움말 메뉴에서 펌웨어복구를 클릭합니다.



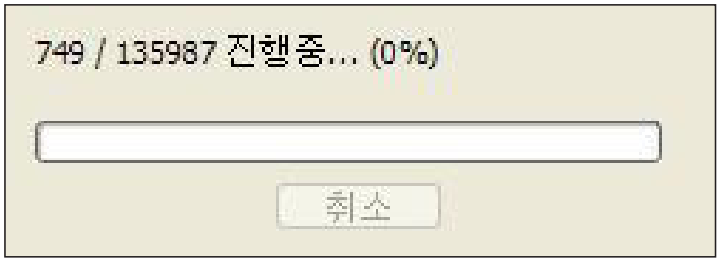
07 불러오기

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

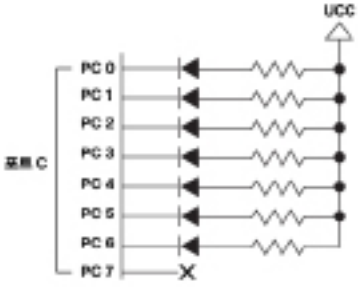
■ PORTX : PORT X 데이터 레지스터

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

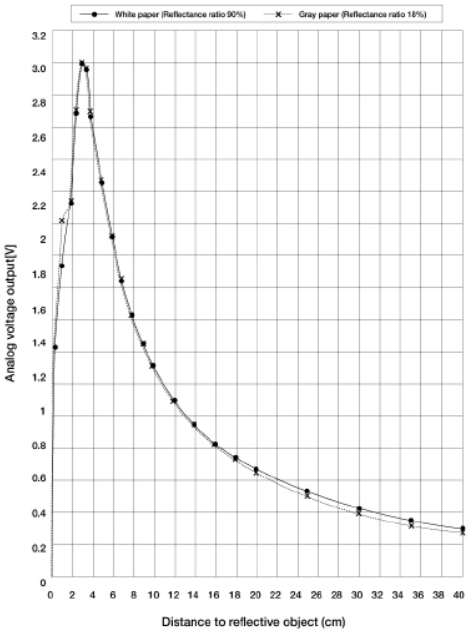
DDRX 레지스터는 포트 X의 입출력 방향을 결정하는 레지스터이며 비트 값이 0이면 입력, 1이면 출력으로 설정됩니다. PORTX 레지스터는 포트 X에 출력할 논리 값을 저장하는 레지스터입니다. DDXn값이 1(출력)으로 설정되어 있으면, 포트 X의 n번 핀에 PORTXn에 설정된 값을 출력하게 됩니다.

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.



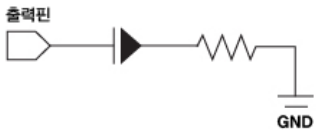
[그림1]



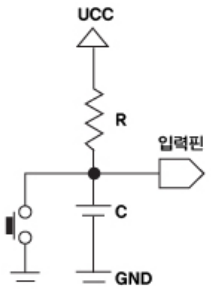
[그림2]



[그림3]



[그림4]



[그림5]

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

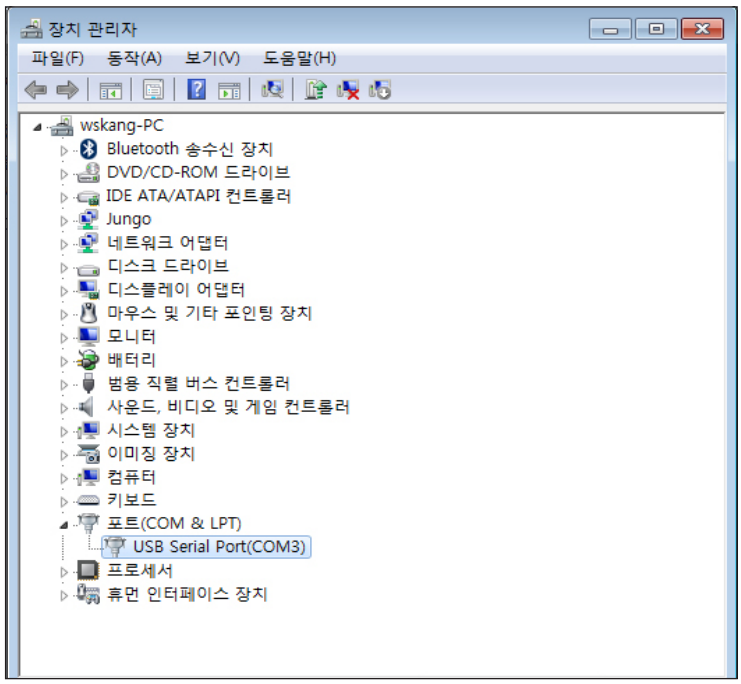
준비



하드웨어 준비하기

01 연결

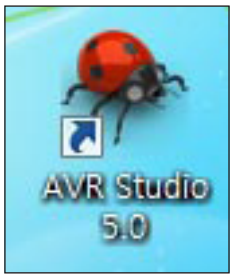
DRC 제어기 아답터로 전원을 연결하고, USB to Serial Cable 로 PC 와 DRC 를 연결합니다.



02 COM port 연결

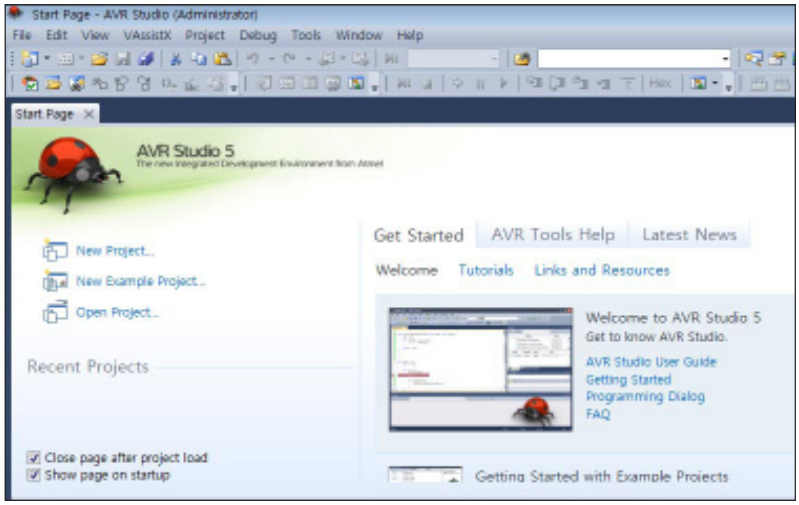
내컴퓨터 〉 속성 〉 장치관리자에서 COM 연결 확인합니다.

AVR Studio 실행 및 코딩 준비



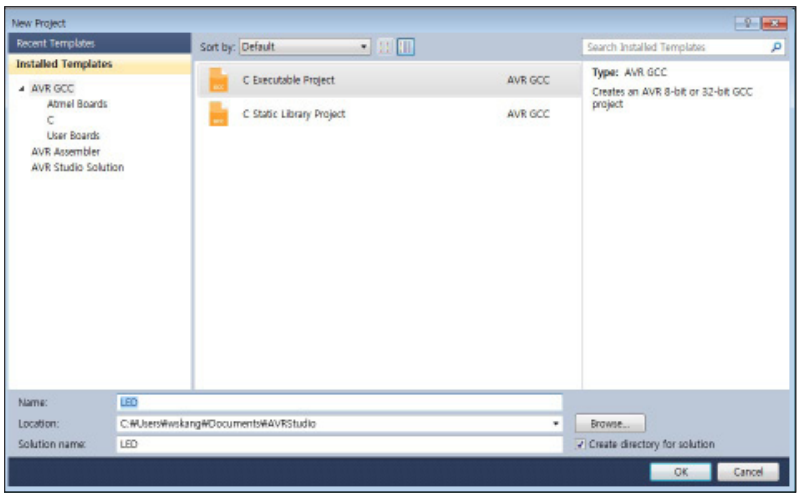
01 실행

AVR Studio 를 실행합니다.



02 New Project

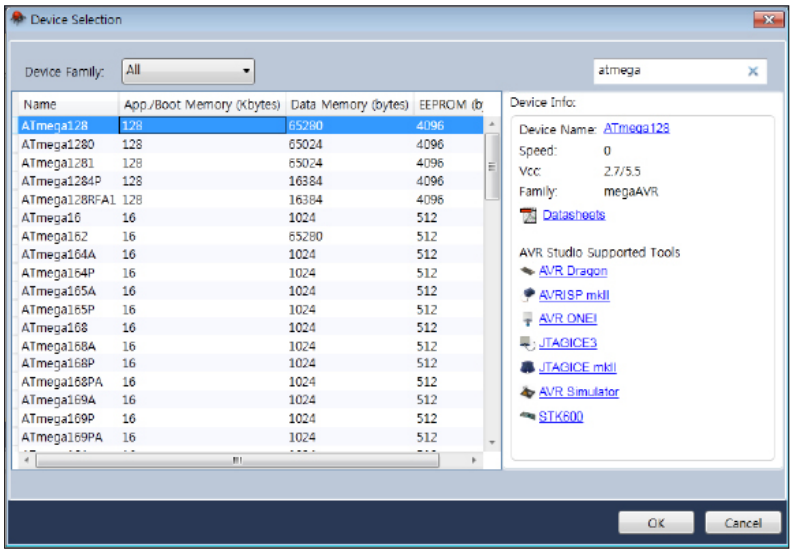
New Project 를 클릭합니다.



03 GCC 선택

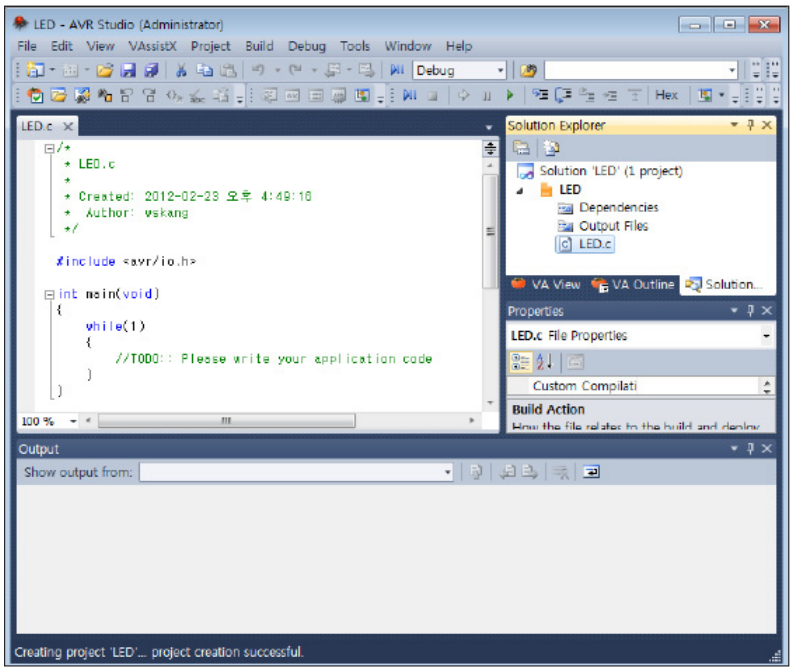
C Executable Project 를 선택합니다.

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;
```

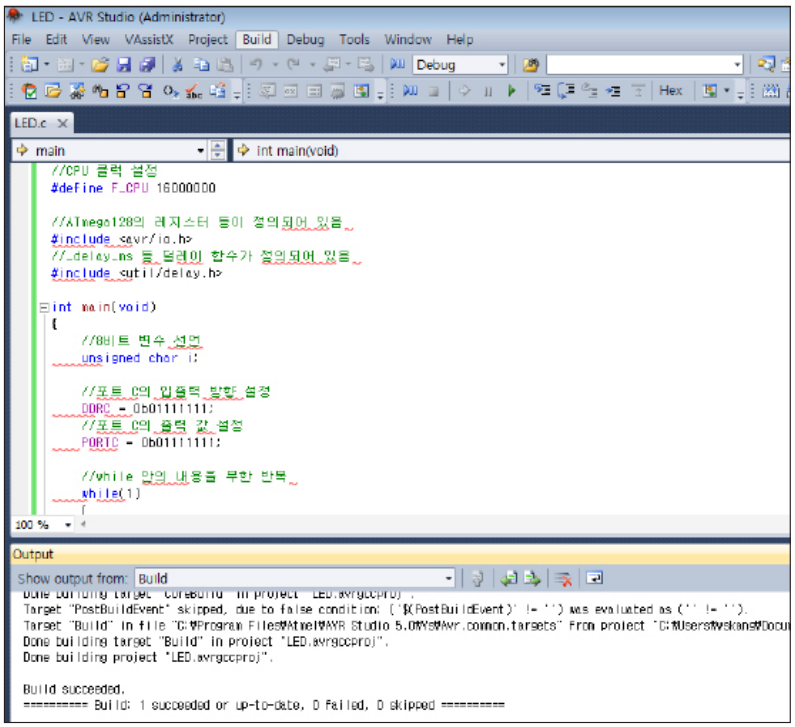
포트 C의 0~6번 핀을 출력으로 설정하고, 출력 값을 1로 만듭니다. 그러면 모든 LED가 꺼지게 됩니다. 여기서 0b01111111은 C언어에서 2진수를 나타내는 표기법입니다.

```
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

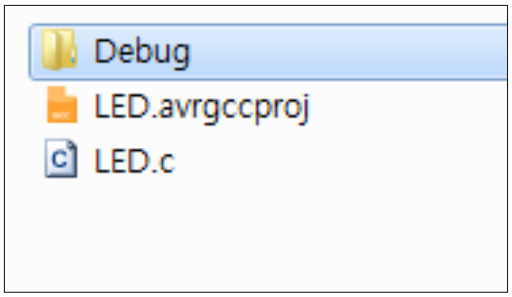
다운로드하고 실행 시 Servo LED부터 Power LED까지 LED가 하나씩 추가로 켜집니다. LED 7개가 다 켜지고 난 후에는 다시 Servo LED만 켜지고, 차례로 하나씩 켜지는 것을 반복합니다.



01 Build

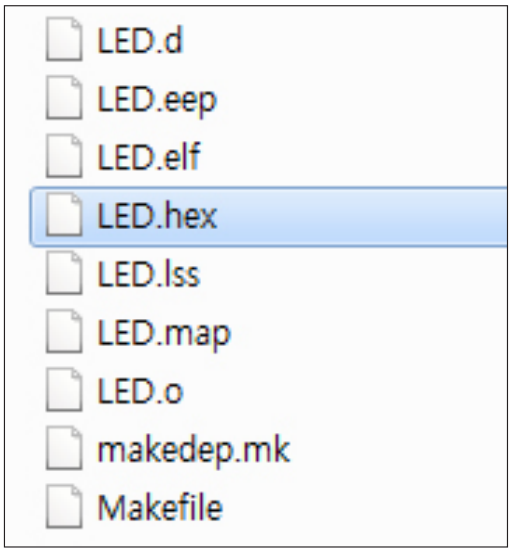
Build 메뉴를 선택하여 Build Solution을 클릭하여 Build 시킵니다.

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



02 Debug 폴더

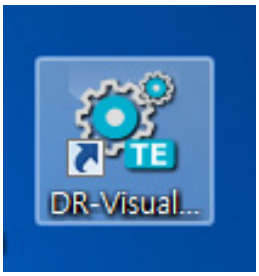
Build 후에 소스가 있는 폴더에 들어가면 Debug 폴더가 생성 되어있습니다. 클릭하여 폴더 안의 내용을 봅니다.



03 hex 확인

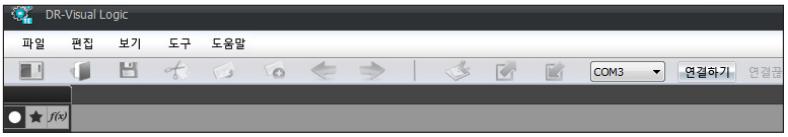
Debug 폴더 안의 hex 파일 확인합니다. Hex 파일이 제어기에 포팅시키는 파일입니다.

DR-Visual Logic 에서 펌웨어 복구를 실시합니다.



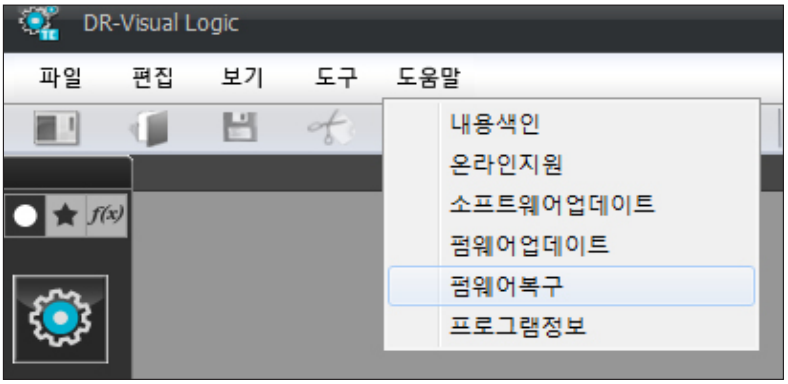
04 Visual Logic

DR-Visual Logic 을 실행시킵니다.



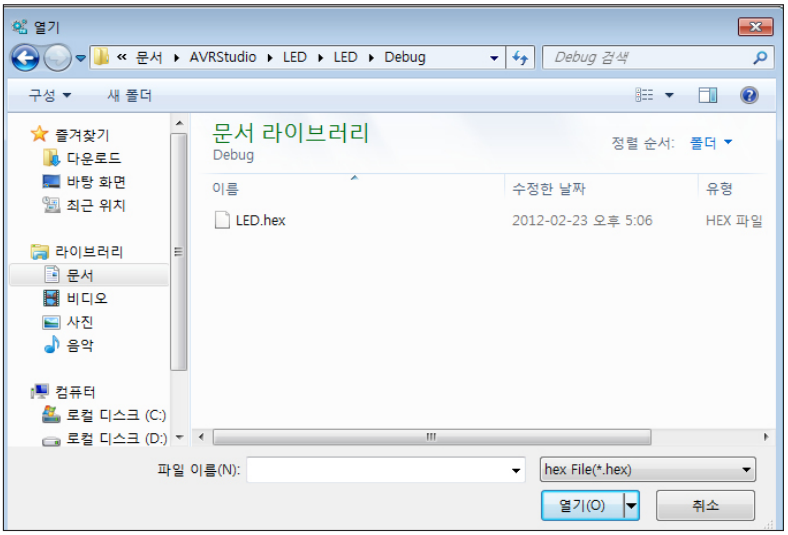
05 연결하기

COM 번호를 설정하고 연결하기를 클릭합니다.



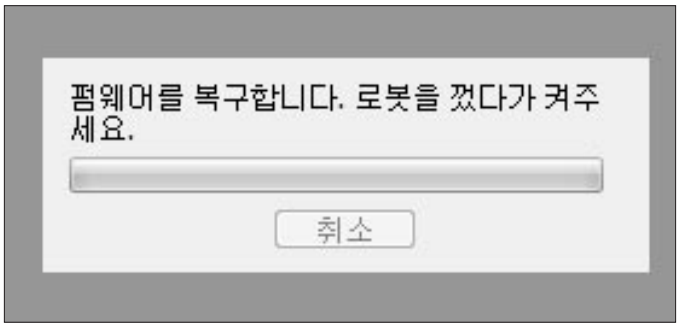
06 펌웨어 복구

도움말 메뉴에서 펌웨어복구를 클릭합니다.



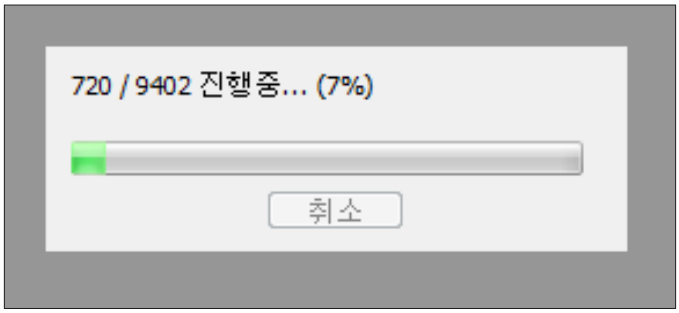
07 불러오기

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.

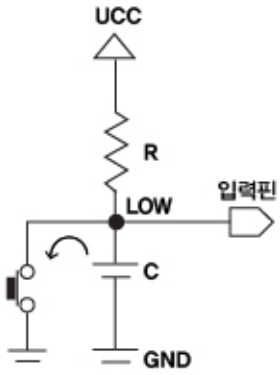
H/W 원리

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.

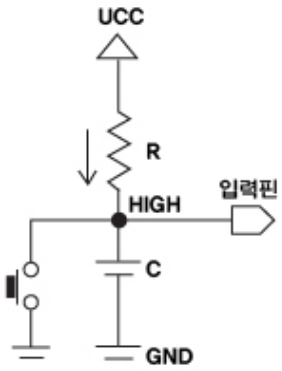
택트 스위치를 ATmega128에 입력하는 회로는 아래와 같습니다(그림 01). 스위치가 눌리지 않았을 때, 회로는 콘덴서 C가 저항 R을 통해서 충전되어 디지털 입력 핀에는 VCC와 같은 전압이 들어갑니다(그림 02). 따라서 스위치가 눌리지 않았을 때 입력 값은 HIGH가 됩니다. 스위치가 눌린 경우, 디지털 입력 핀은 GND와 직접 연결되어 콘덴서 C가 방전됩니다(그림 03). 디지털 입력 핀에는 GND 전압이 들어가므로, 입력 값이 LOW가 됩니다. 이 회로에서 저항 R이 없다면, 스위치가 안 눌린 경우에 입력 핀은 연결된 전압이 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]

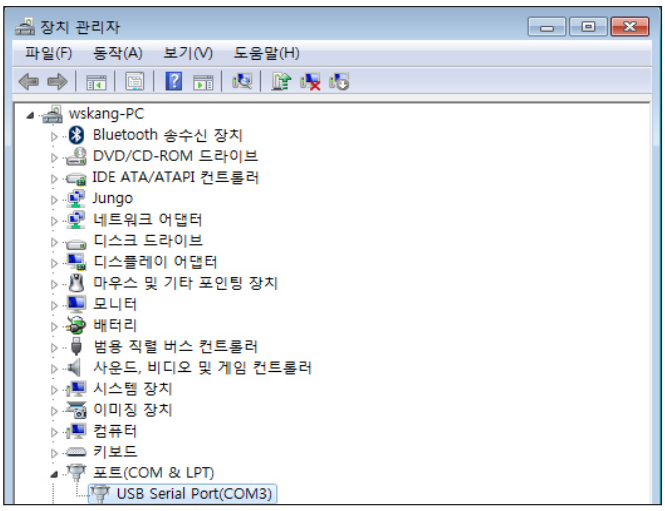
준비



하드웨어 준비하기

01 연결

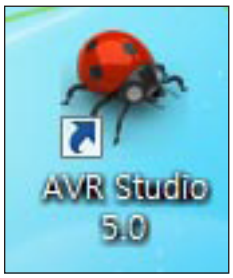
DRC 제어기 아답터로 전원을 연결하고, USB to Serial Cable 로 PC 와 DRC 를 연결합니다.



02 COM port 연결

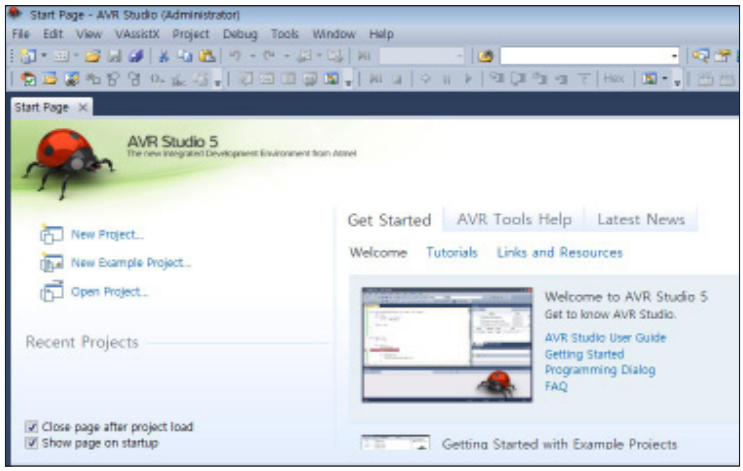
내컴퓨터 〉속성 〉장치관리자에서 COM 연결 확인합니다.

AVR Studio 실행 및 코딩 준비



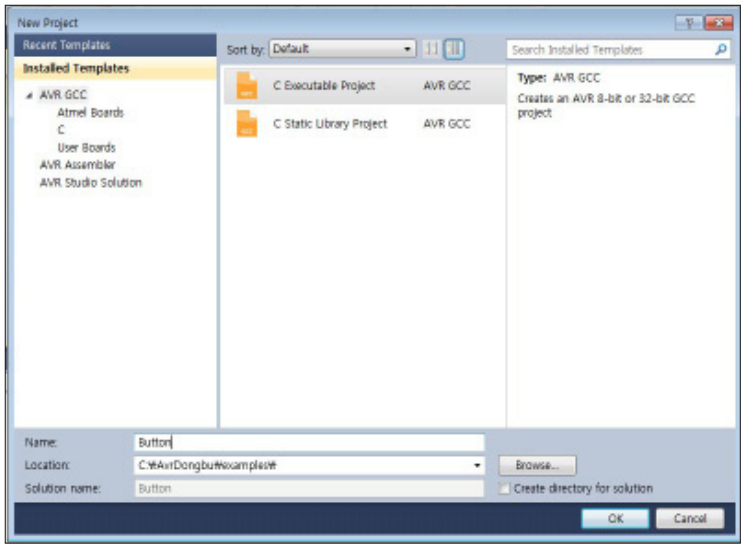
01 실행

AVR Studio 를 실행합니다.



02 New Project

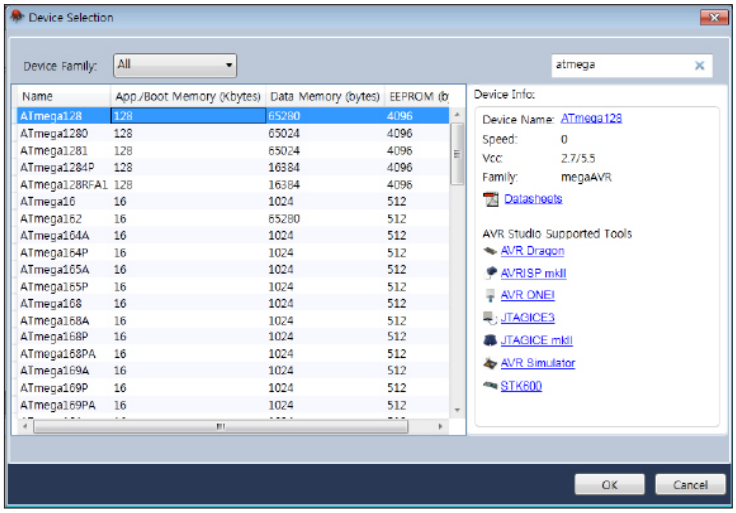
New Project 를 클릭합니다.



03 GCC 선택

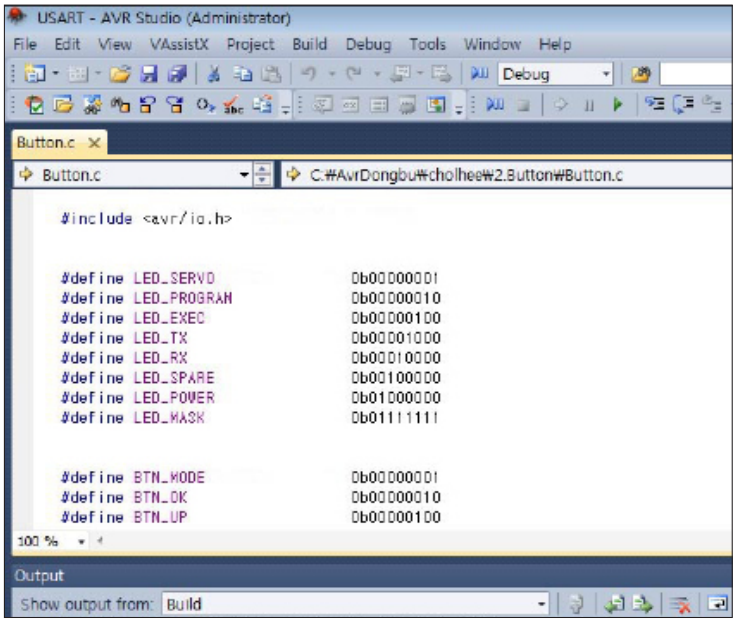
C Executable Project 를 선택합니다.

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;
}
```

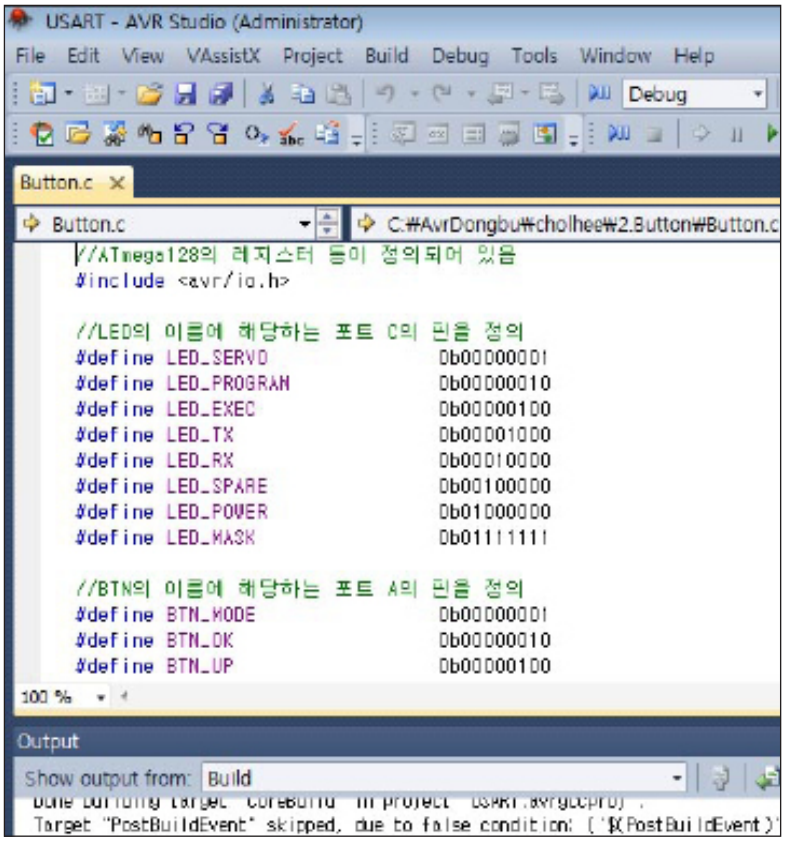
if(ucButton & BTN_MODE)라는 조건문은 ucButton 값을 BTN_MODE(0b00000001)와 & 연산합니다. 결과값은 ucButton의 0번 비트가 1이라면(Mode 버튼이 눌렸다면) 0b00000001, 0이라면 그냥 0이 됩니다. MODE 버튼이 눌렸다면 if문의 괄호 안 값이 참이 되므로 안의 내용이 실행됩니다. if문 안의 ucLED |= LED_PROGRAM은 ucLED = ucLED | LED_PROGRAM의 간략한 표현입니다. ucLED 값과 LED_PROGRAM(0b00000010)을 |(bitwise 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.

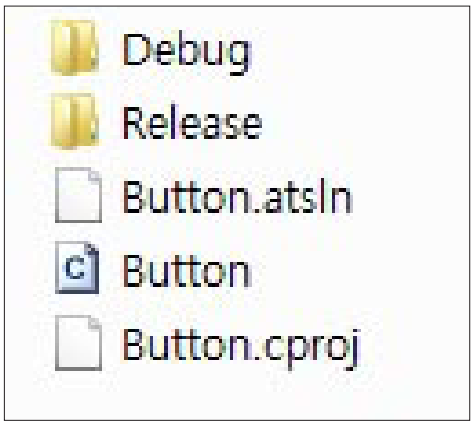
```
PORTC = (~ucLED) & LED_MASK;
```

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 Solution을 클릭하여 Build 시킵니다.

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

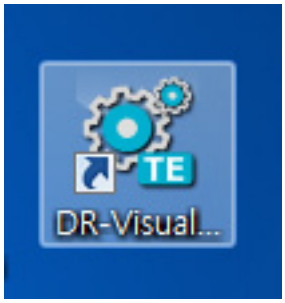
02 Debug 폴더

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

03 hex 확인

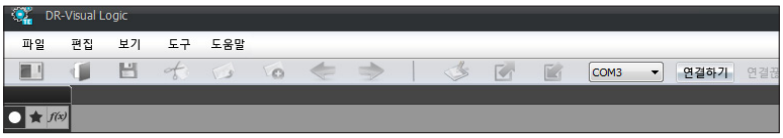
Debug 폴더 안의 hex 파일 확인합니다. Hex 파일이 제어기에 포팅시키는 파일입니다.

DR-Visual Logic 에서 펌웨어 복구를 실시합니다.



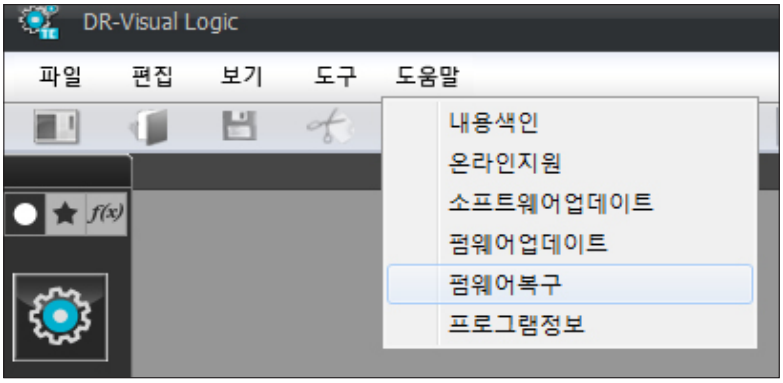
04 Visual Logic

DR-Visual Logic 을 실행시킵니다.



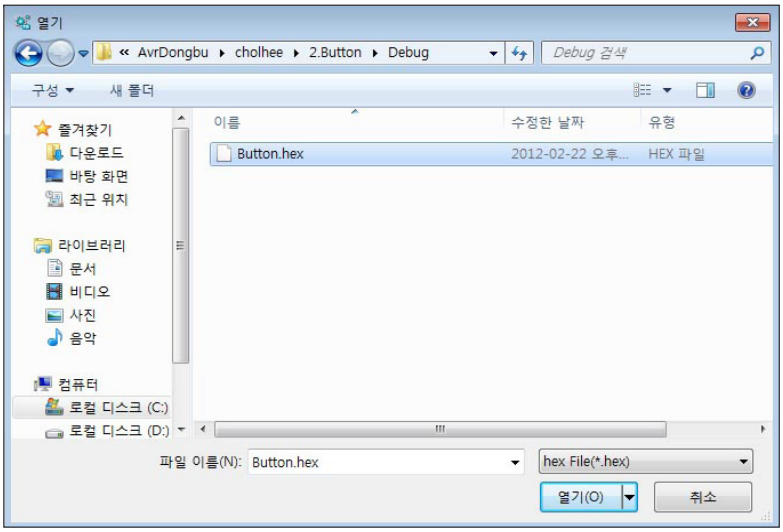
05 연결하기

COM 번호를 설정하고 연결하기를 클릭합니다.



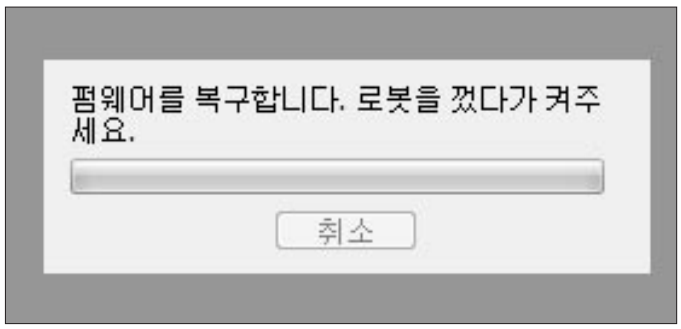
06 펌웨어 복구

도움말 메뉴에서 펌웨어복구를 클릭합니다.



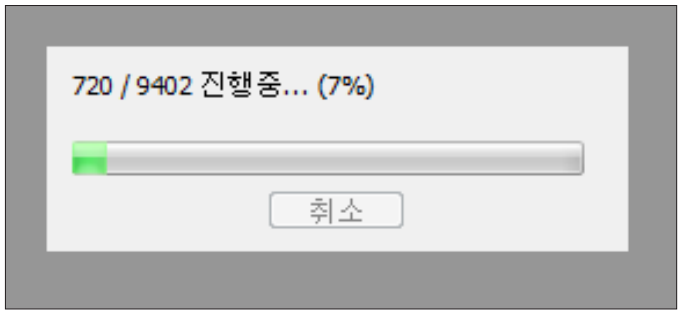
07 불러오기

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

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.

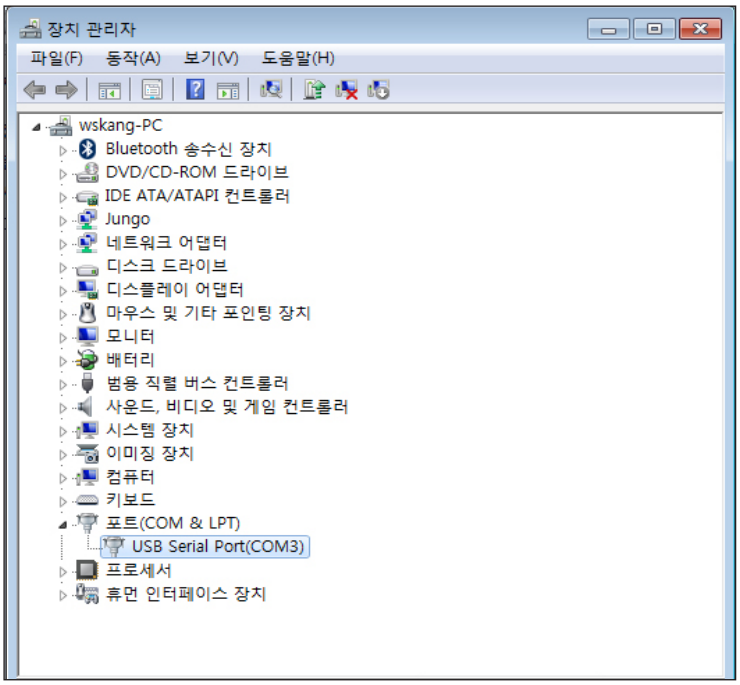
이번 예제에서는 delay 함수를 사용해서 PB7을 일정 주기로 바꾸어 음을 내 봅니다.

하드웨어 준비하기



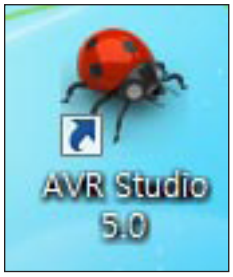
01 연결

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



02 COM port 연결

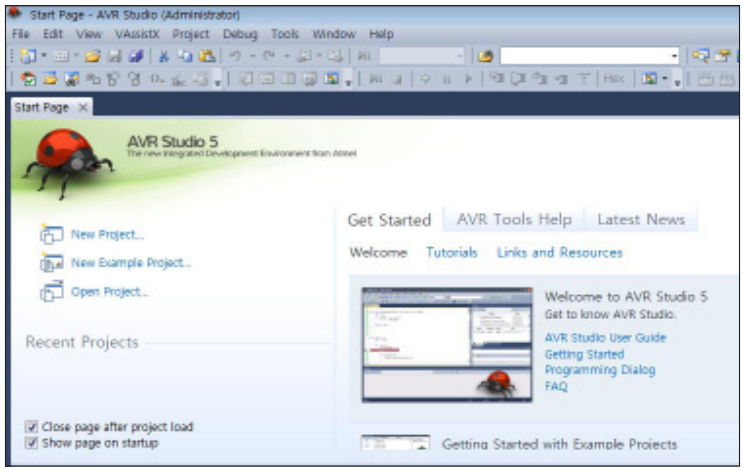
내컴퓨터 〉 속성 〉 장치관리자에서 COM 연결 확인합니다.



AVR Studio 실행 및 코딩 준비

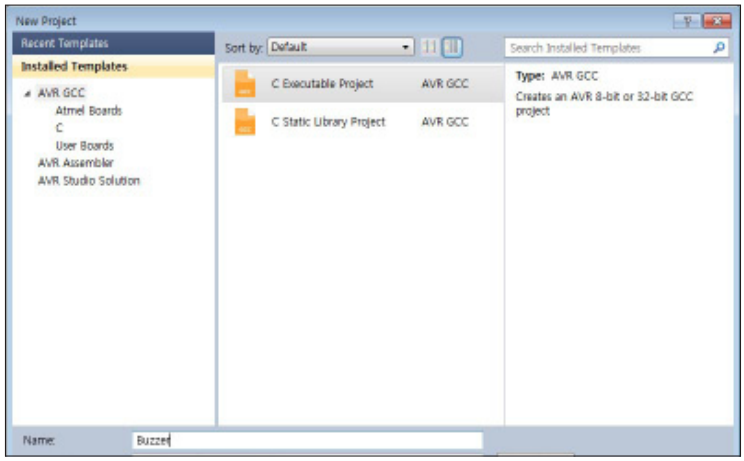
01 실행

AVR Studio 를 실행합니다.



02 New Project

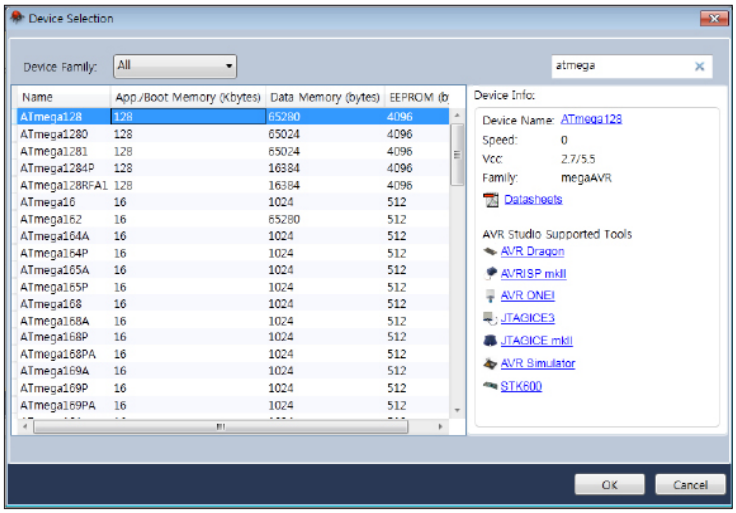
New Project 를 클릭합니다.



03 GCC 선택

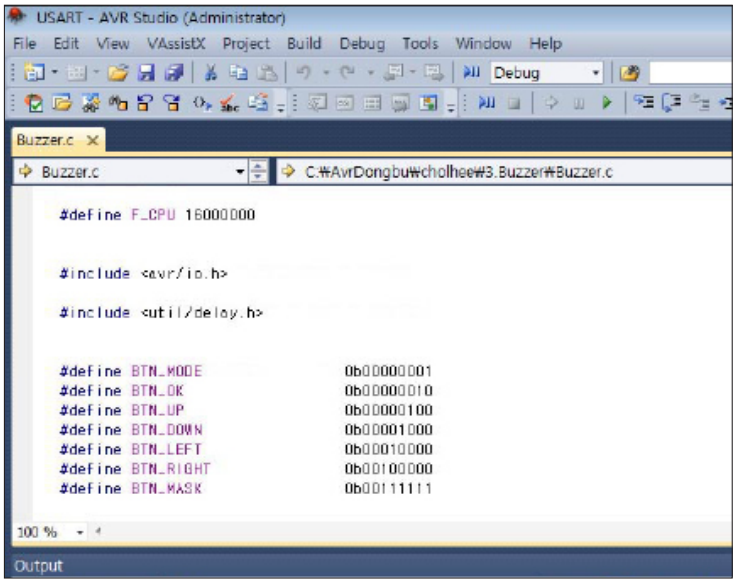
C Executable Project 를 선택합니다.

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);
}
```

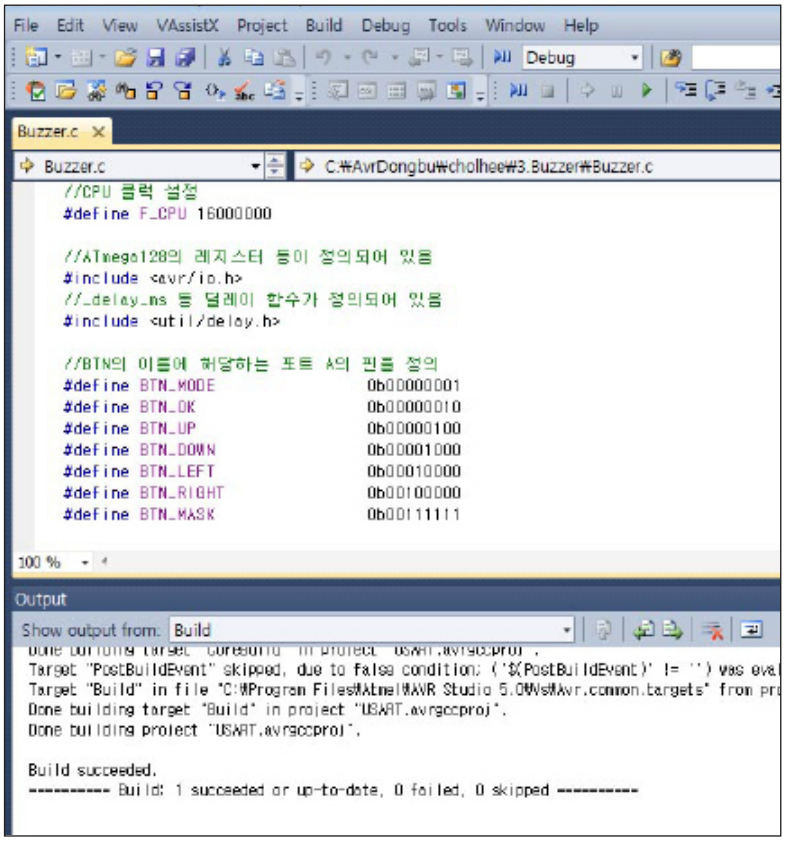
if(ucButton & BTN_OK)라는 조건문의 결과는 ucButton 값 중 OK 버튼에 해당하는 비트가 1이라면 참이 되고, 0이라면 거짓이 됩니다. 즉 OK 버튼이 눌렸다면 괄호 안의 내용이 실행됩니다.

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

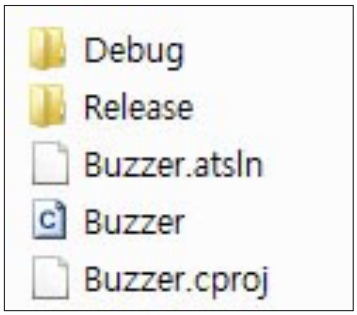
다운로드하고 실행 시 OK버튼을 누르면 버저가 울립니다.

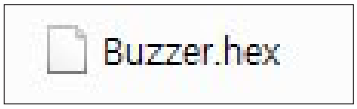


01 Build

Build 메뉴를 선택하여 Build Solution을 클릭하여 Build 시킵니다.

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





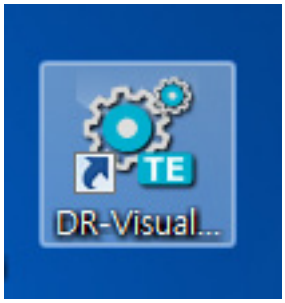
02 Debug 폴더

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

03 hex 확인

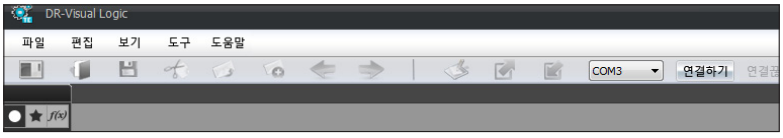
Debug 폴더 안의 hex 파일 확인합니다. Hex 파일이 제어기에 포팅시키는 파일입니다.

DR-Visual Logic 에서 펌웨어 복구를 실시합니다.



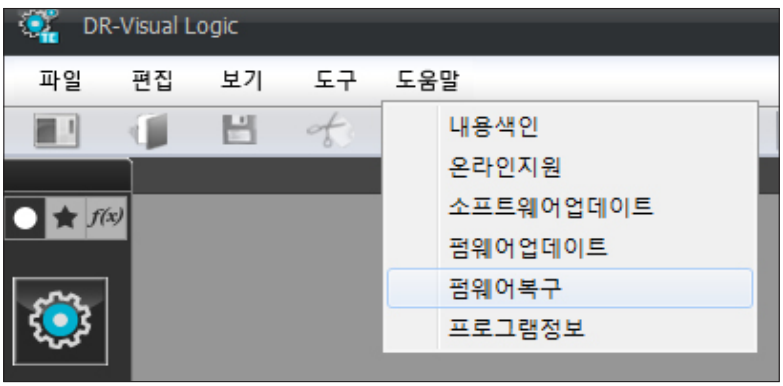
04 Visual Logic

DR-Visual Logic 을 실행시킵니다.



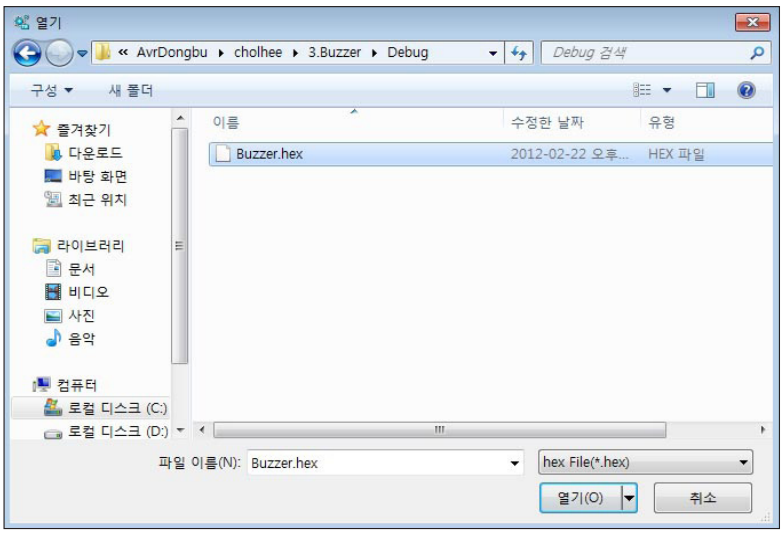
05 연결하기

COM 번호를 설정하고 연결하기를 클릭합니다.



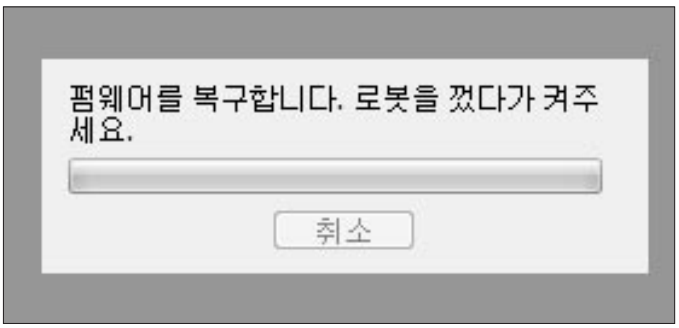
06 펌웨어 복구

도움말 메뉴에서 펌웨어복구를 클릭합니다.



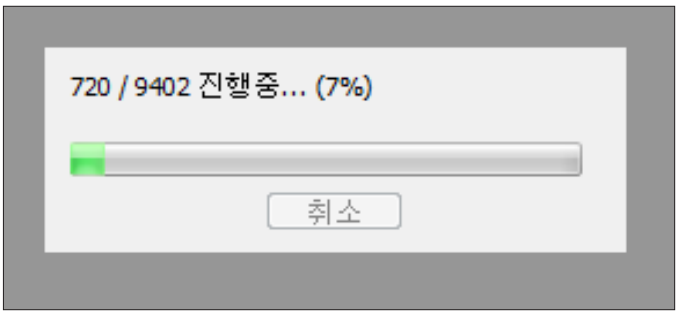
07 불러오기

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

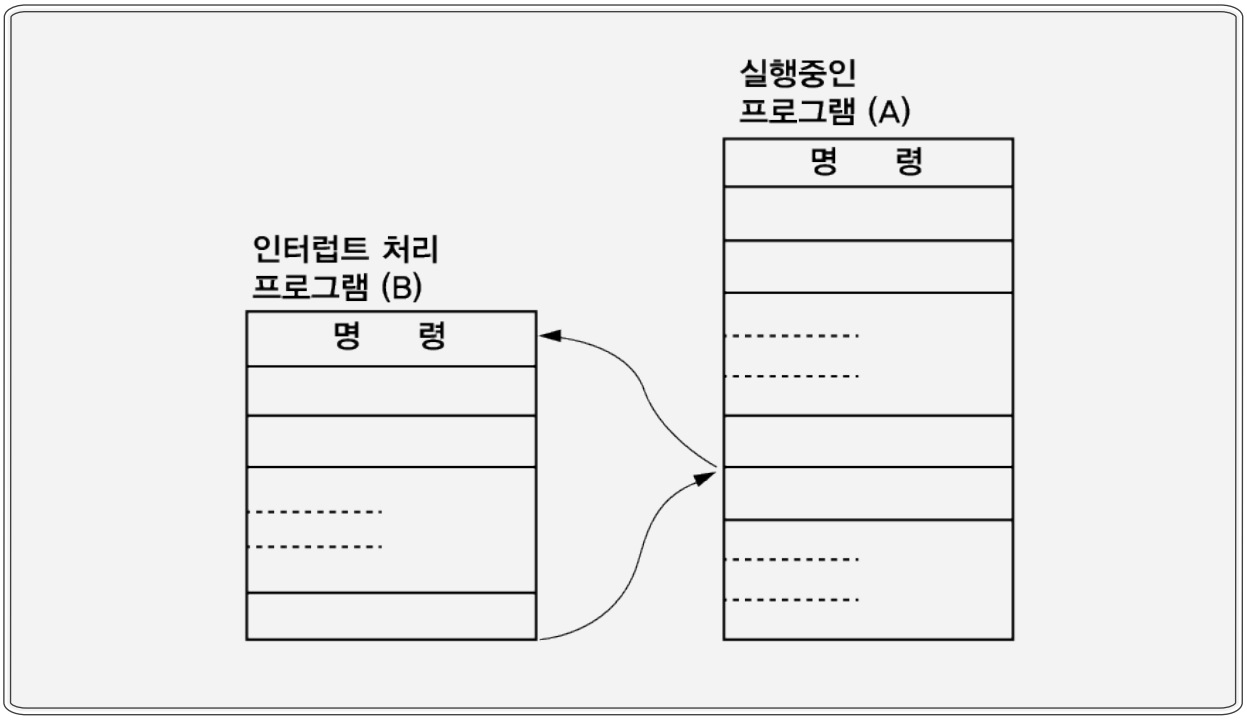
OK 버튼을 누르면 버저에서 비프음이 납니다.

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.

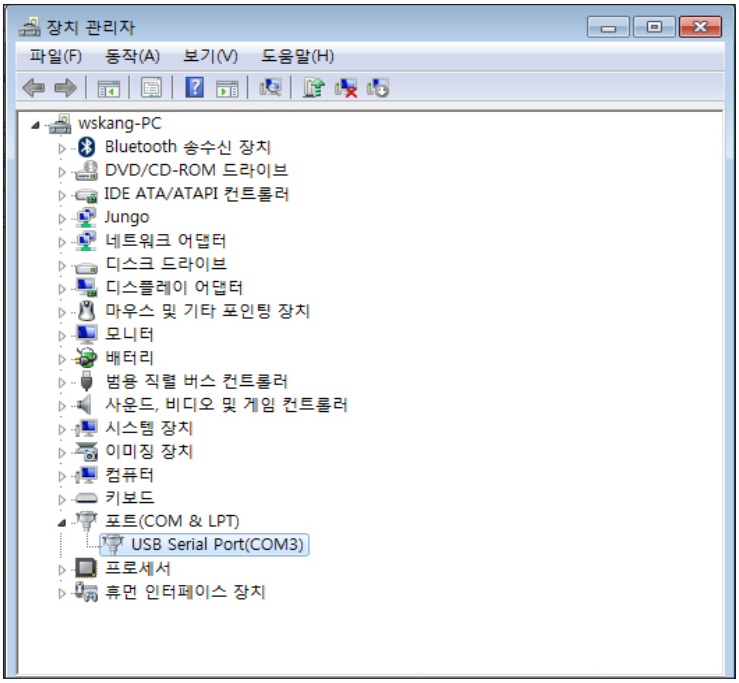
준비



하드웨어 준비하기

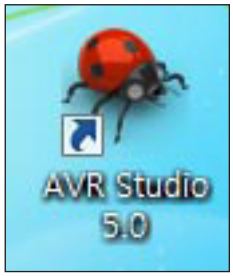
01 연결

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



02 COM port 연결

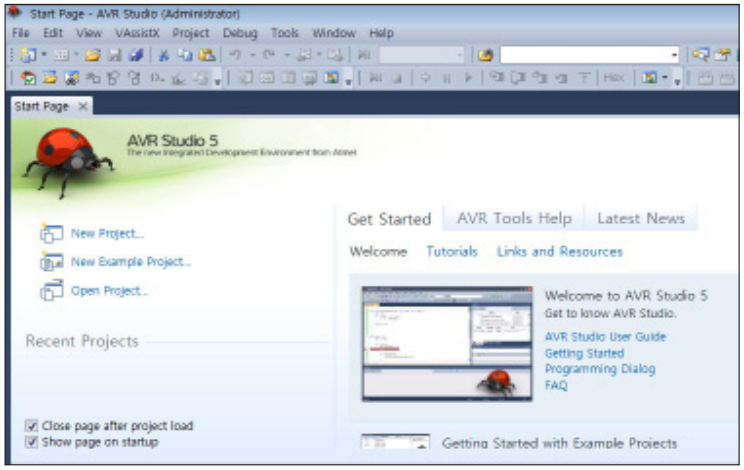
내컴퓨터 〉 속성 〉 장치관리자에서
COM 연결 확인합니다.



AVR Studio 실행 및 코딩 준비

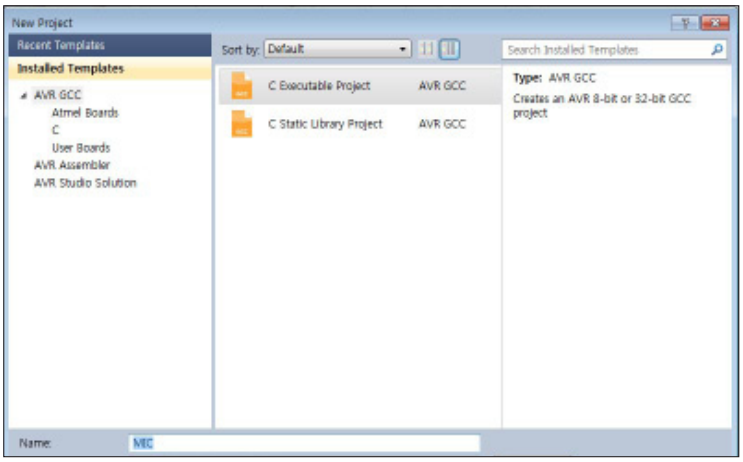
01 실행

AVR Studio 를 실행합니다.



02 New Project

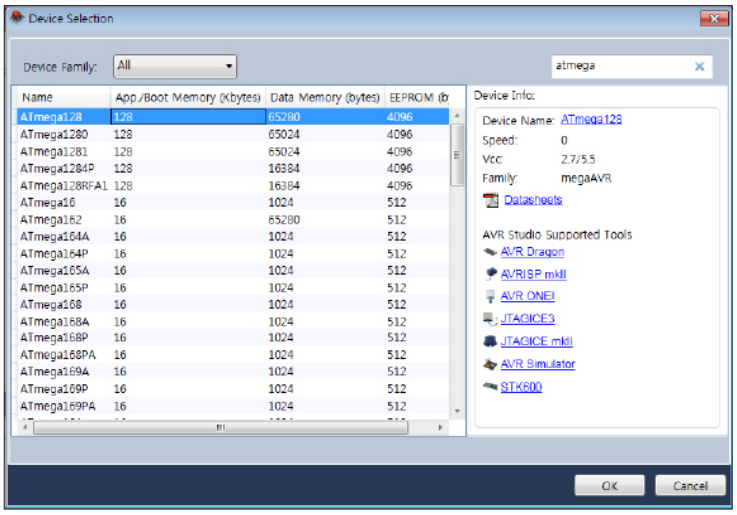
New Project 를 클릭합니다.



03 GCC 선택

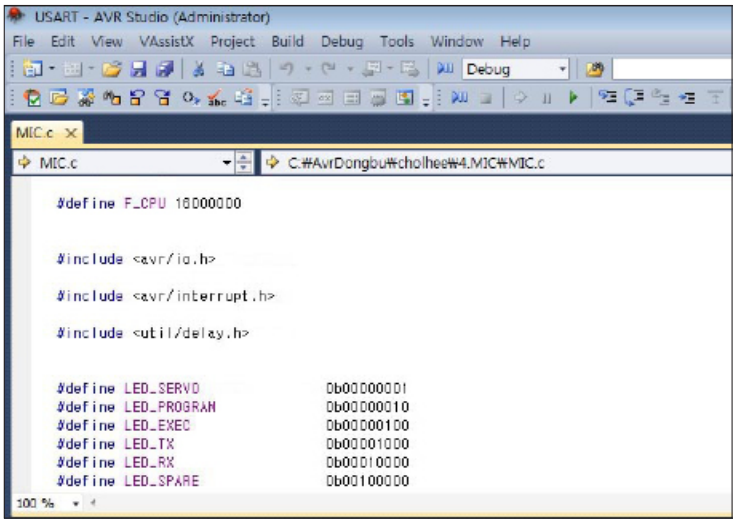
C Executable Project 를 선택합니다.

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();
```

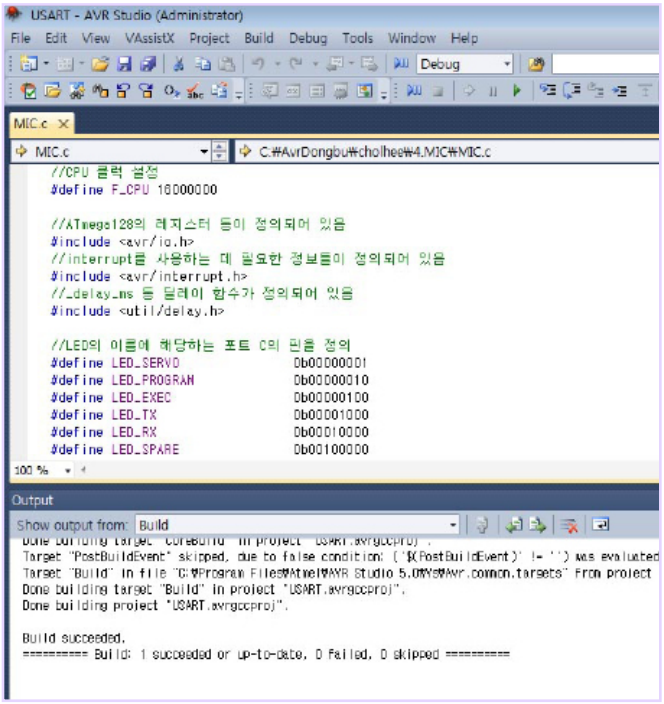
cli()와 sei()는 각각 전체 인터럽트를 비활성화, 활성화 시키는 명령어입니다. ATmega128에는 SREG라는 AVR 상태 레지스터가 있는데, 이 중 7번 비트인 I 비트는 인터럽트 요청이 있을 때 인터럽트 서비스 루틴으로 점프하도록 허용하는 비트입니다. 레지스터 값을 바꾸어서 외부 인터럽트를 사용하겠다고 설정을 해놓았더라도, 이 I 비트가 1로 설정되어야 실제로 인터럽트를 사용할 수 있습니다. cli() 명령은 I 비트를 0으로, sei() 명령은 I 비트를 1로 설정합니다.
인터럽트에 대한 설정을 바꾸는 중에 인터럽트가 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.

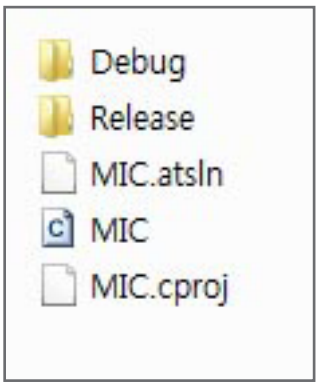
```
while(1)
{

}
```

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.

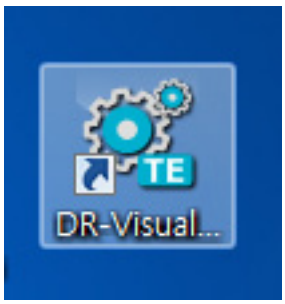
02 Debug 폴더

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

03 hex 확인

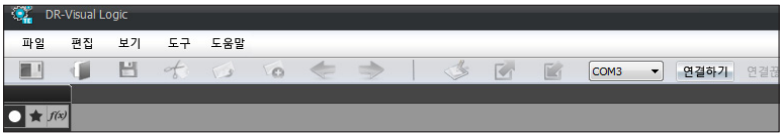
Debug 폴더 안의 hex 파일 확인합니다. Hex 파일이 제어기에 포팅시키는 파일입니다.

DR-Visual Logic 에서 펌웨어 복구를 실시합니다.



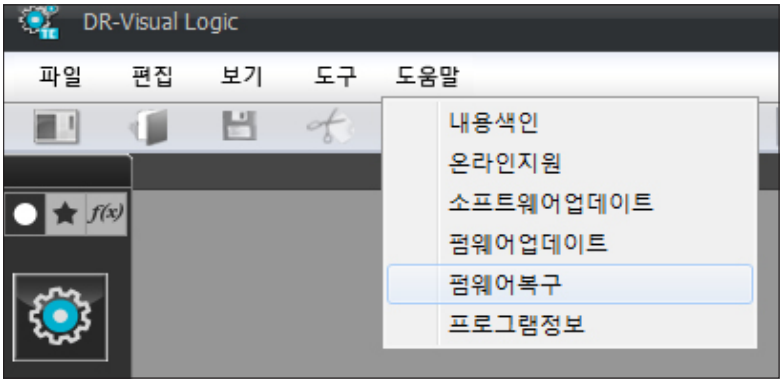
04 Visual Logic

DR-Visual Logic 을 실행시킵니다.



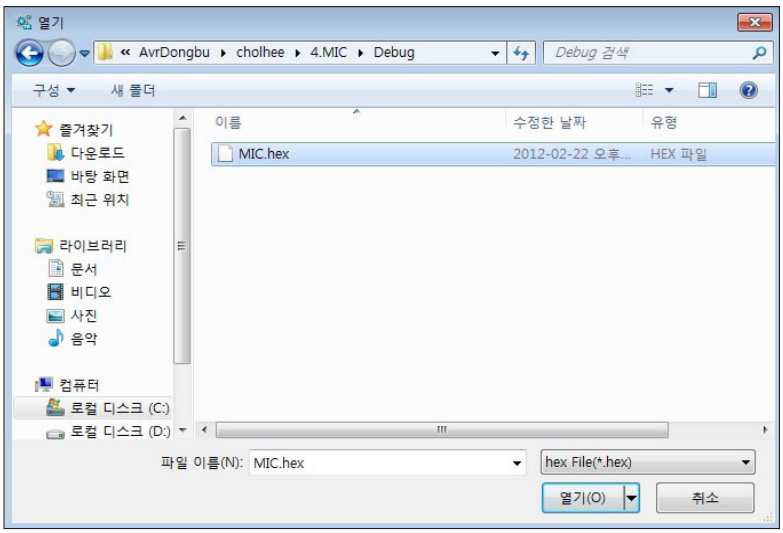
05 연결하기

COM 번호를 설정하고 연결하기를 클릭합니다.



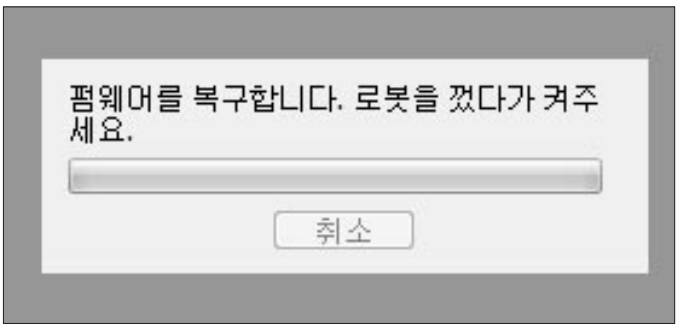
06 펌웨어 복구

도움말 메뉴에서 펌웨어복구를 클릭합니다.



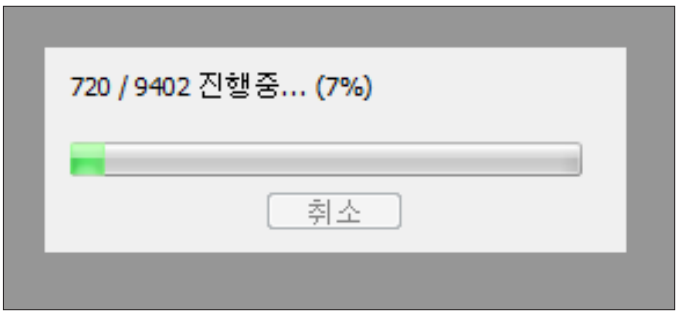
07 불러오기

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.

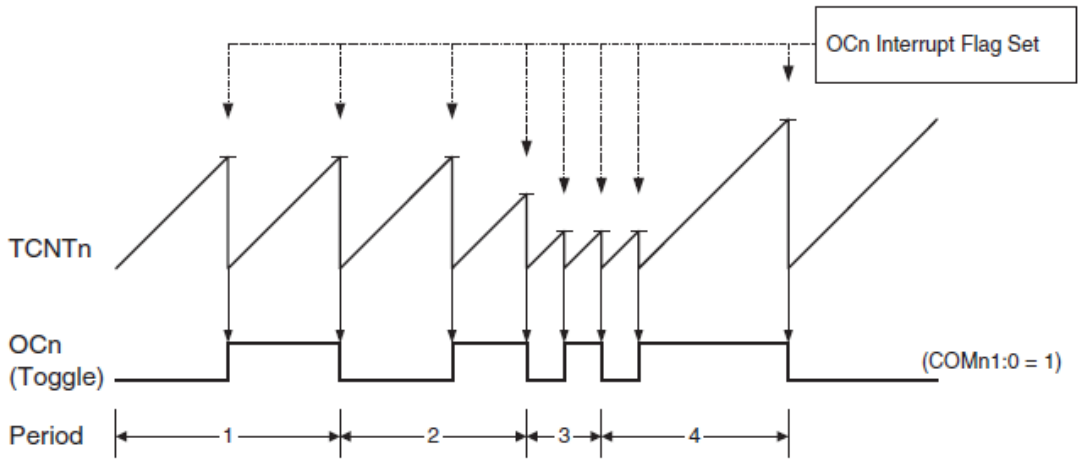
1. 일반 모드(WGM01:0 = 0b00)

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)

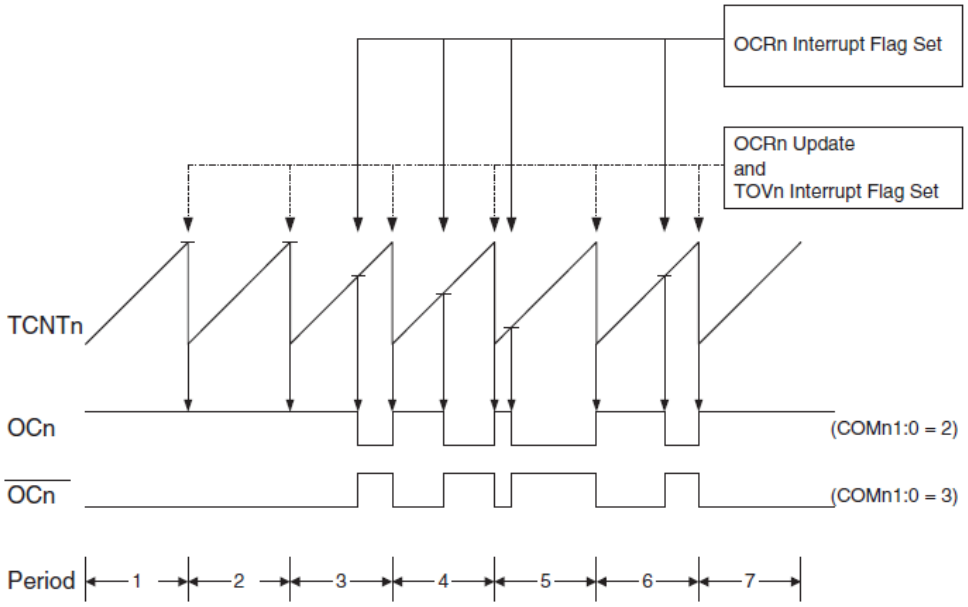
CTC모드는 Clear Timer on Compare 모드의 약자로, OCR0 레지스터가 타이머 초기화의 주기를 결정하는 데 사용됩니다. CTC 모드에서는 TCNT0 레지스터가 OCR0과 같아지는 순간, 즉 비교 매치가 일어나는 순간 0으로 초기화 됩니다. 즉 OCR0 레지스터의 값이 카운터의 최대값을 결정합니다. 비교 매치가 일어나는 순간 OCF0 비트는 1로 바뀌며, 타이머/카운터0 오버플로우 인터럽트 서비스 루틴이 실행되면 0으로 초기화 됩니다.

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)

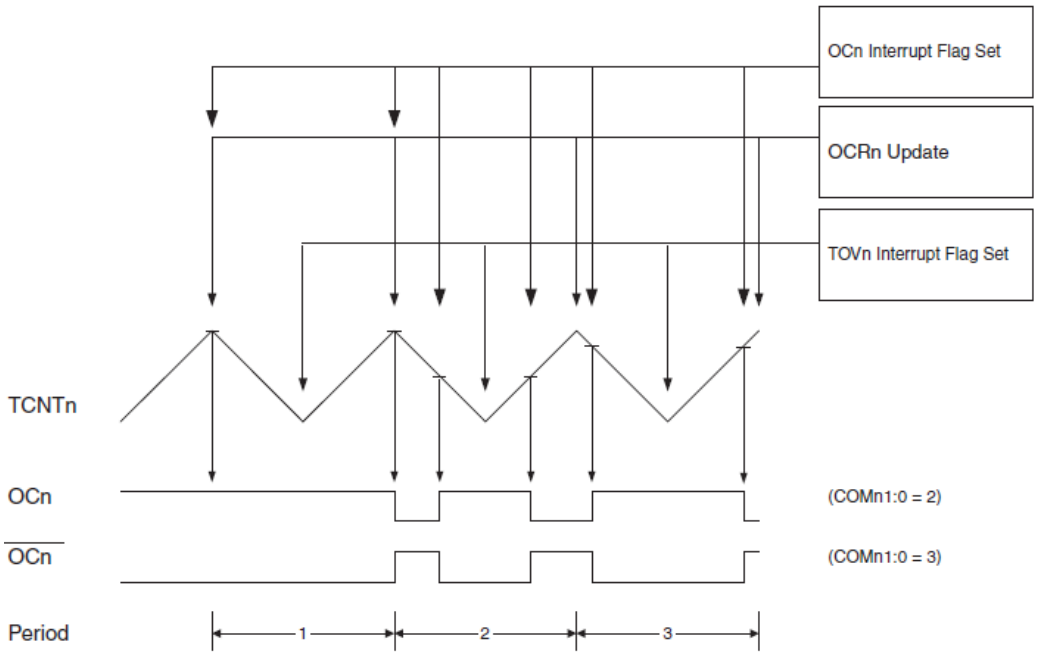
고속 PWM 모드는 TCNT0이 0에서 255까지 증가하고 다시 0으로 변하는 톱니 모양으로 변하며, OCR0의 값에 따라서 듀티 사이클이 변하는 PWM 출력을 만들어내는 모드입니다. COM01:0을 0b10으로 설정한 경우, OC0 핀은 비교 매치가 일어난 때에 LOW로 변하고, TCNT0이 0일 때 HIGH로 변합니다. 따라서 OC0은 0~OCR0 까지는 HIGH, OCR0~255까지는 LOW인 파형이 나오는 비반전 출력 비교 모드로 작동합니다. COM01:0을 0b11로 설정한 경우에는 반대로, 비교 매치가 일어난 때에 HIGH, 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.

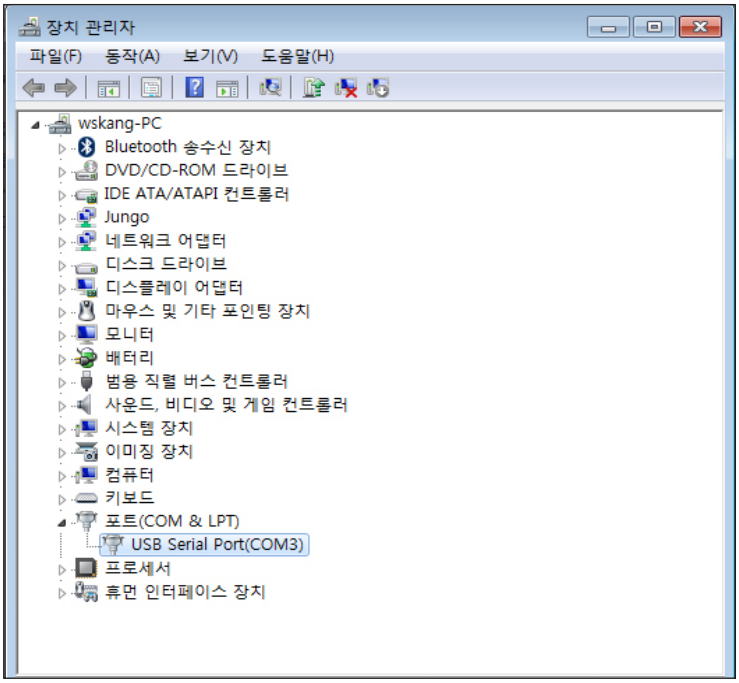
준비



하드웨어 준비하기

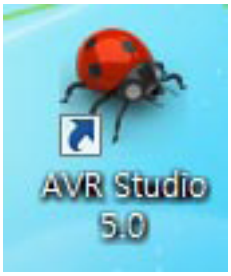
01 연결

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



02 COM port 연결

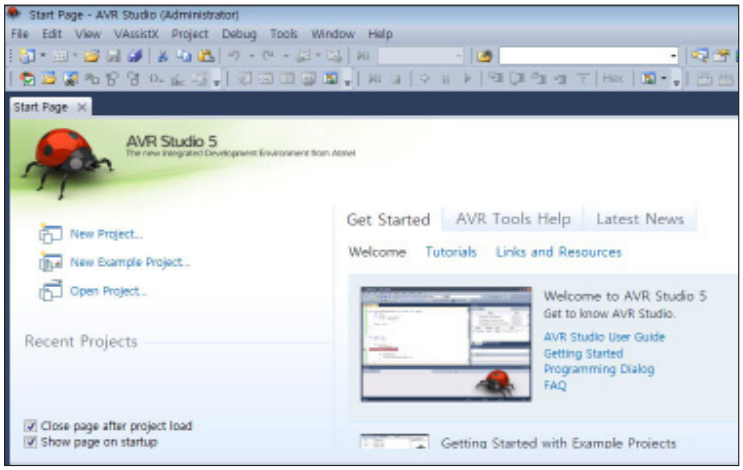
내컴퓨터 〉 속성 〉 장치관리자에서 COM 연결 확인합니다.



AVR Studio 실행 및 코딩 준비

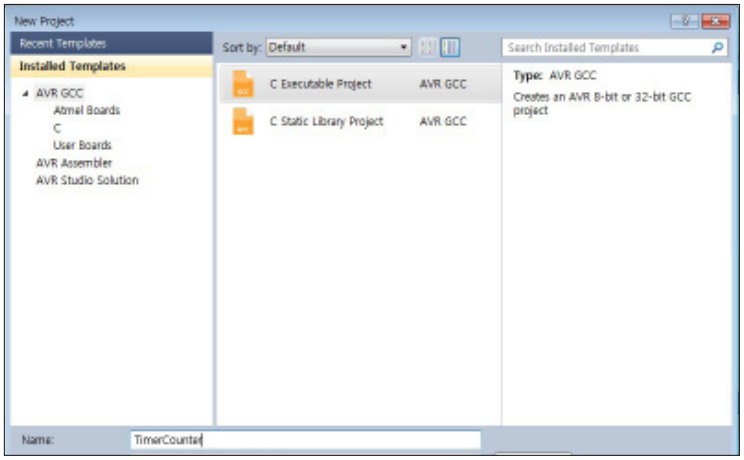
01 실행

AVR Studio 를 실행합니다.



02 New Project

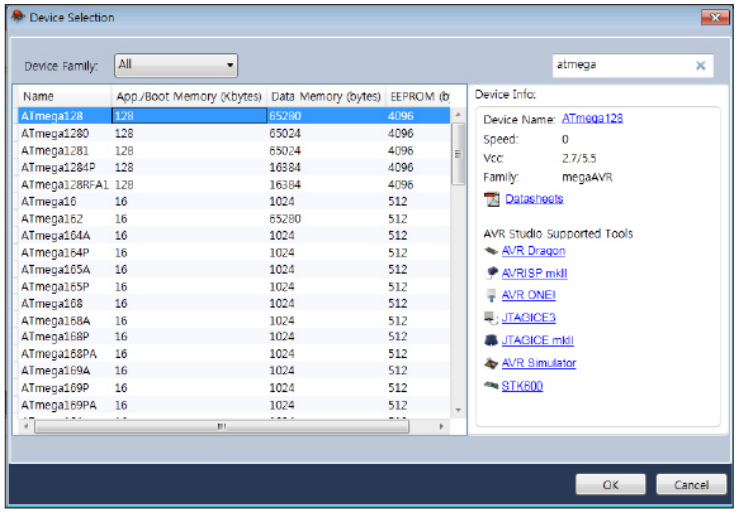
New Project 를 클릭합니다.



03 GCC 선택

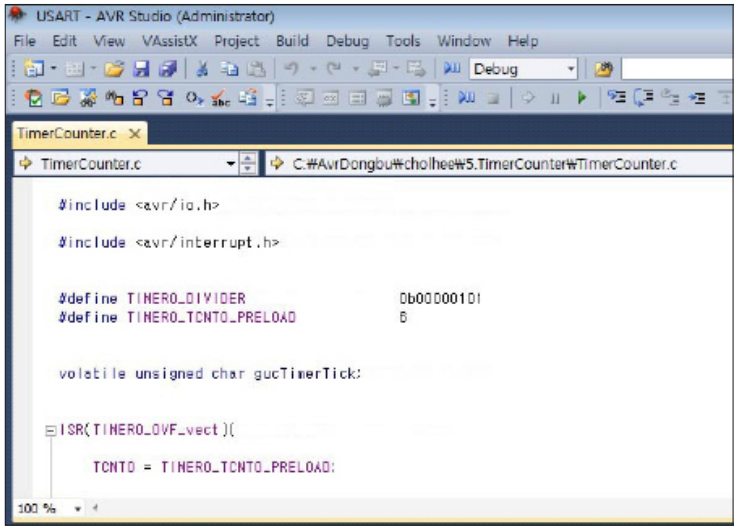
C Executable Project 를 선택합니다.

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.

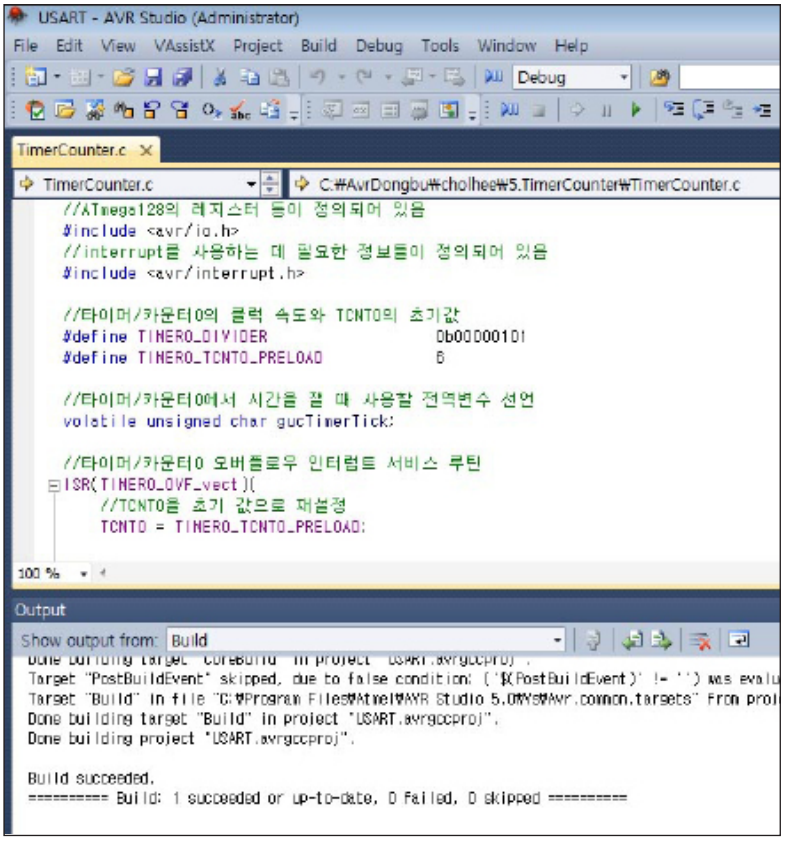
TCCR0의 CS02:0에 값을 넣어 타이머가 동작하기 시작하면, sei() 명령을 통해서 전체 인터럽트를 활성화 합니다. 이제부터는 2ms 마다 타이머/카운터0 오버플로우 인터럽트가 발생합니다.

```
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

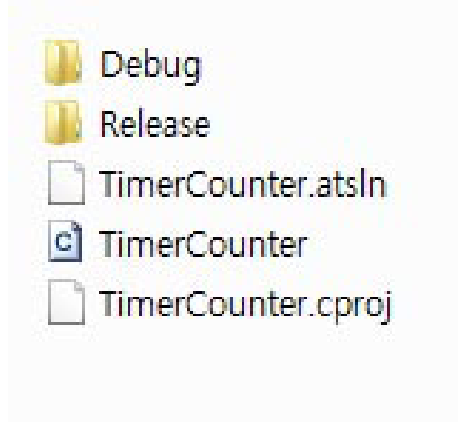
Chapter 1과 같이, Servo LED부터 Power LED까지 LED가 하나씩 추가로 켜집니다. LED 7개가 다 켜지고 난 후에는 다시 Servo LED만 켜지고, 차례로 하나씩 켜지는 것을 반복합니다.



01 Build

Build 메뉴를 선택하여 Build Solution을 클릭하여 Build 시킵니다.

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





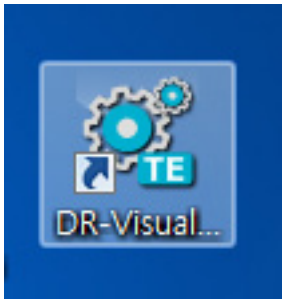
02 Debug 폴더

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

03 hex 확인

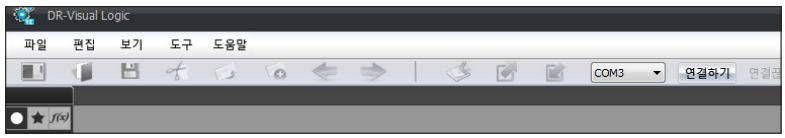
Debug 폴더 안의 hex 파일 확인합니다. Hex 파일이 제어기에 포팅시키는 파일입니다.

DR-Visual Logic 에서 펌웨어 복구를 실시합니다.



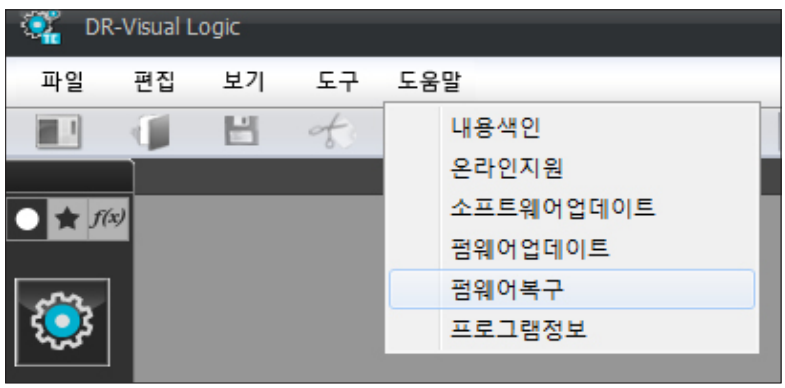
04 Visual Logic

DR-Visual Logic 을 실행시킵니다.



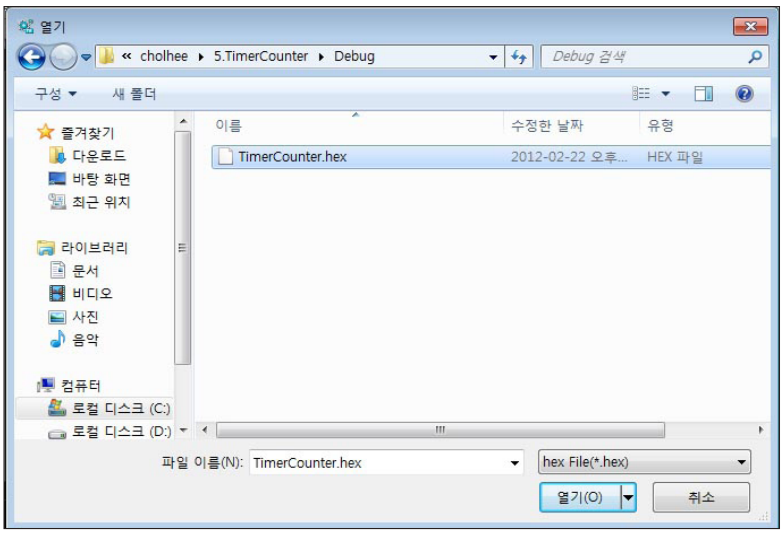
05 연결하기

COM 번호를 설정하고 연결하기를 클릭합니다.



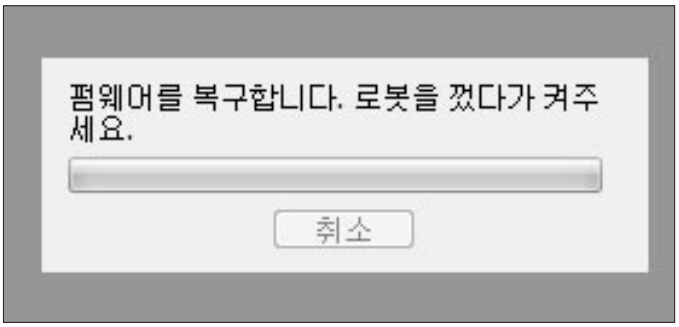
06 펌웨어 복구

도움말 메뉴에서 펌웨어복구를 클릭합니다.



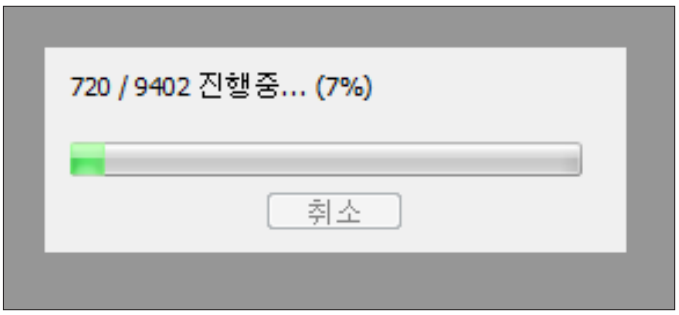
07 불러오기

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.

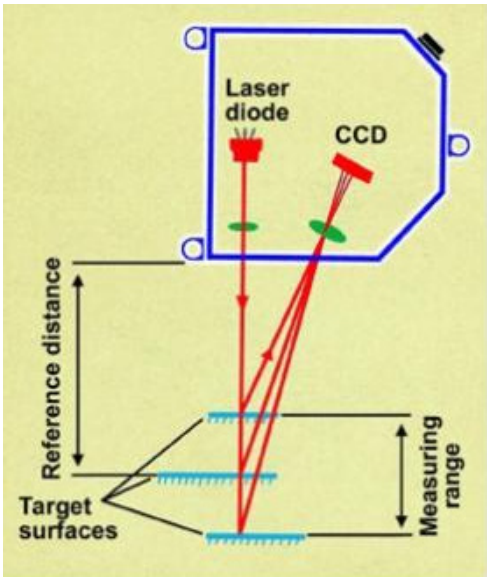
단극성 모드 : $$ADC = \frac{V_{IN} \times 1024}{V_{REF}}$$

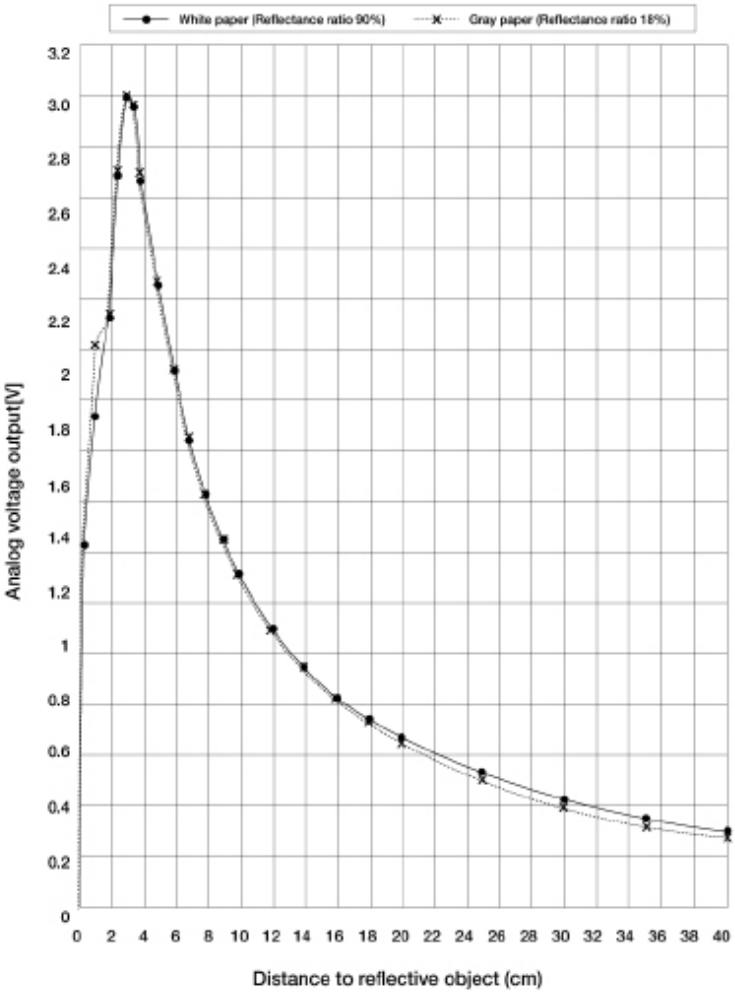
차동 모드 : $$ADC = \frac{(V_{POS} - V_{NEG}) \times GAIN \times 512}{V_{REF}}$$

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.





[그림.1]

하드웨어 준비하기



01 연결

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



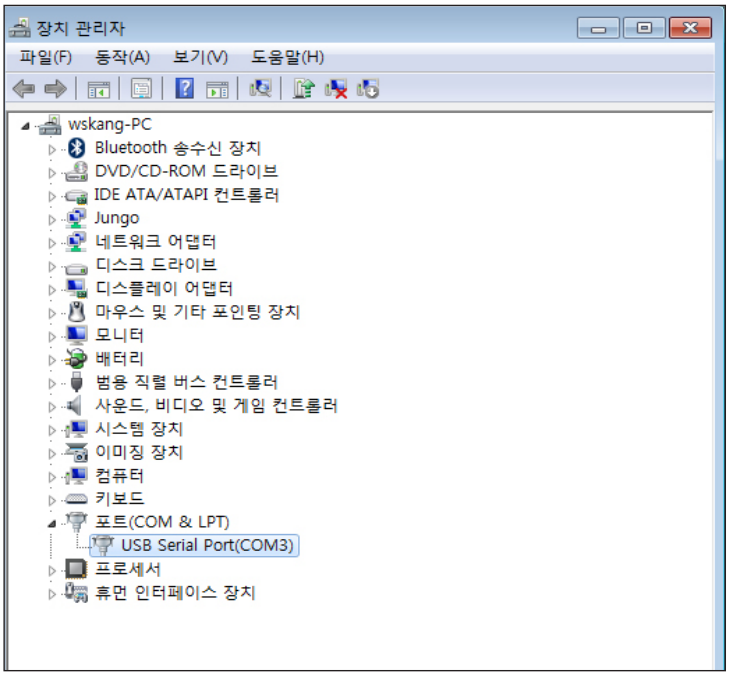
02 PSD 센서 준비

PSD 센서를 준비합니다.



03 PSD 센서 연결

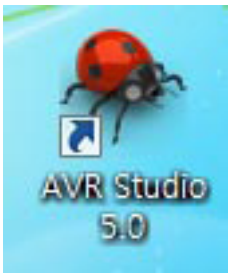
PSD 센서를 제어기 좌측 포트에 연결합니다.



04 COM port 연결

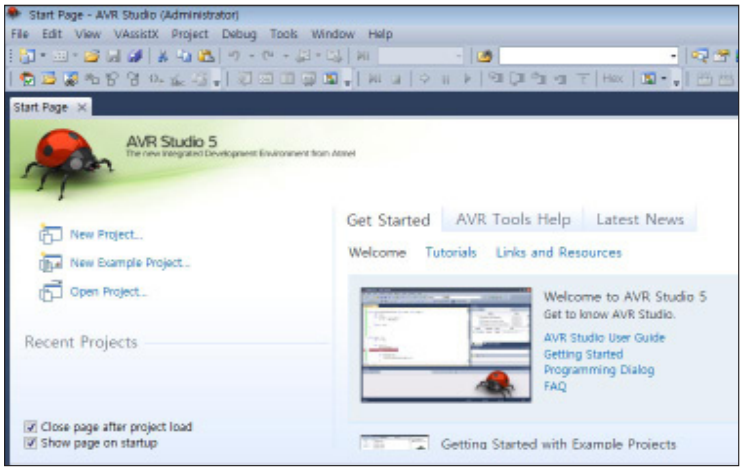
내컴퓨터 〉 속성 〉 장치관리자에서
COM 연결 확인합니다.

AVR Studio 실행 및 코딩 준비



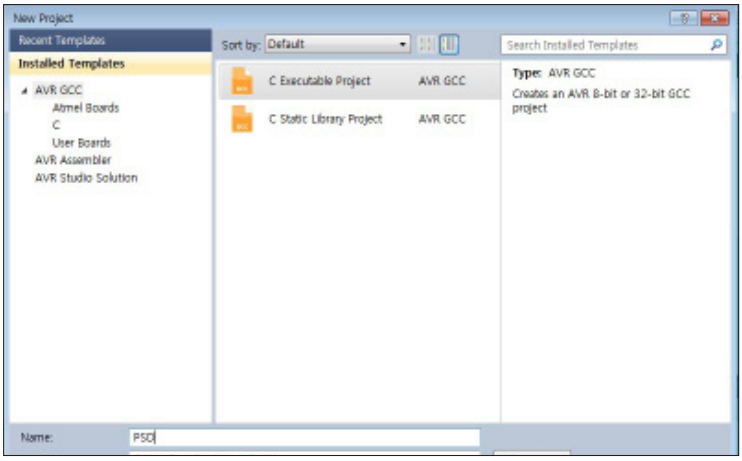
01 실행

AVR Studio 를 실행합니다.



02 New Project

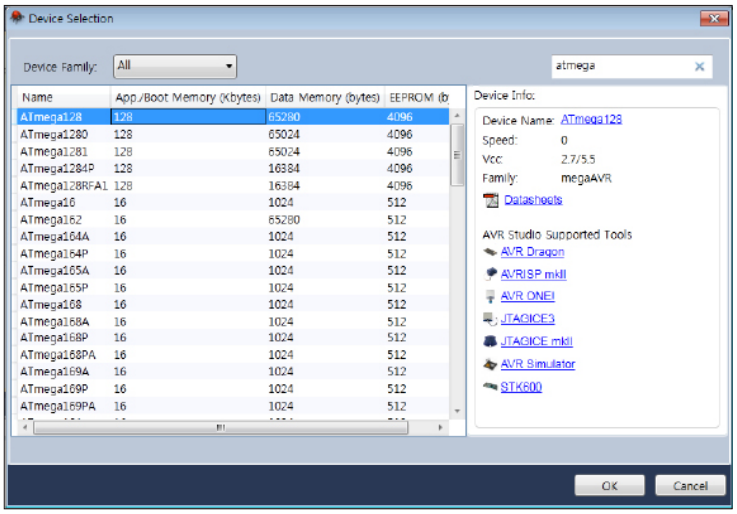
New Project 를 클릭합니다.



03 GCC 선택

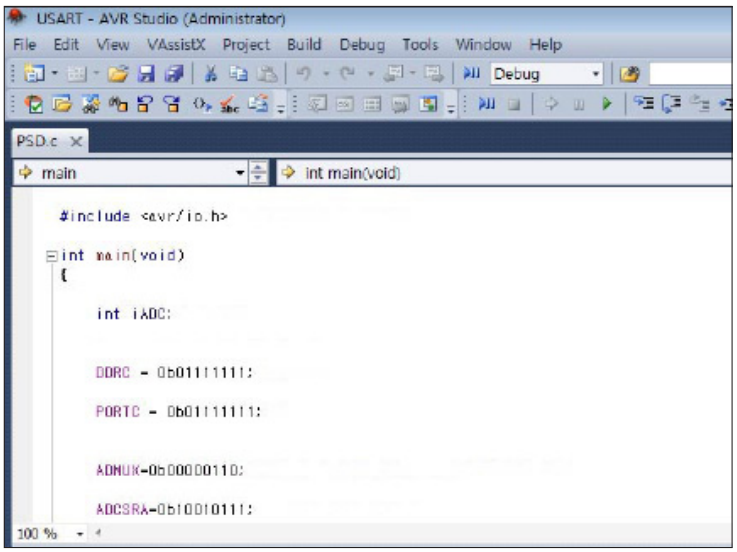
C Executable Project 를 선택합니다.

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;
```

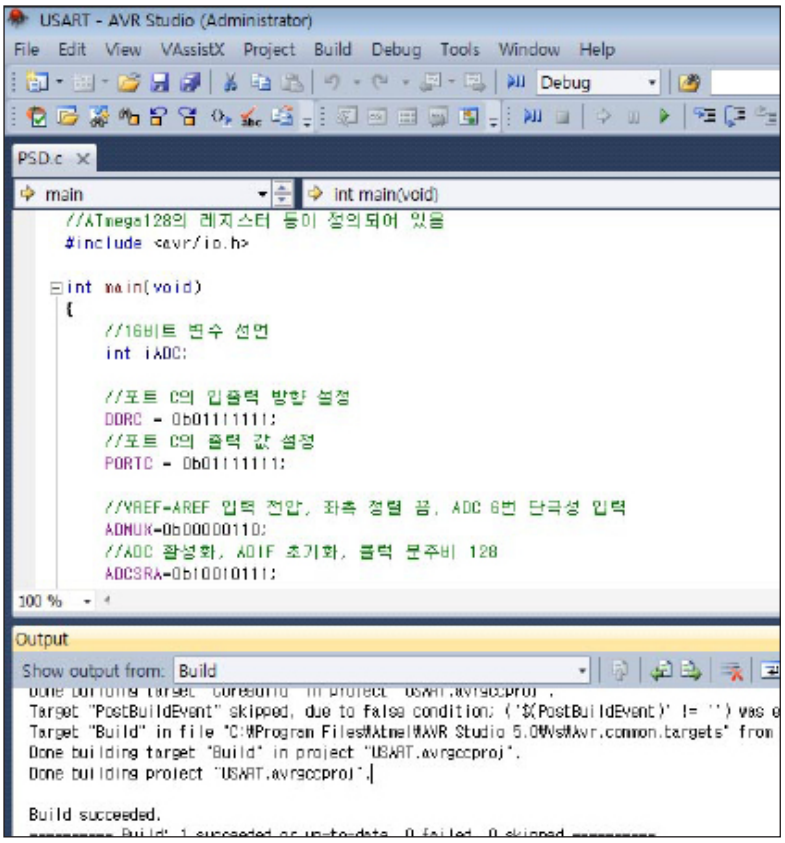
iADC의 값에 따라서 LED를 점멸합니다. 그림 00의 PSD 센서의 거리에 따른 전압 값 그래프를 보면, 물체와의 거리가 10cm 일 때 센서의 출력 값은 약 1.3V가 되는 것을 볼 수 있습니다. 1.3V에 해당하는 ADC 출력 값은 $ADC = \frac{V_{IN} \times 1024}{V_{REF}}$

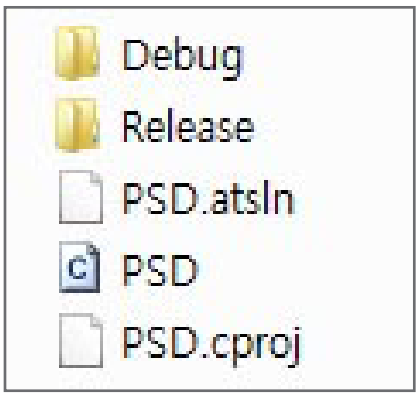
이므로, 수식을 따라서 계산하면 1.3×1024÷5 = 266.24가 됩니다. 반올림하면, 입력 전압이 1.3V일 때 ADC 값은 266 정도가 될 것입니다.

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.

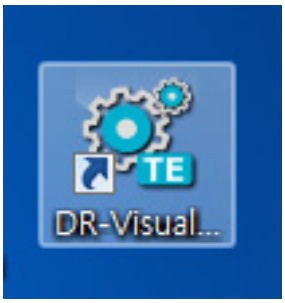
02 Debug 폴더

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

03 hex 확인

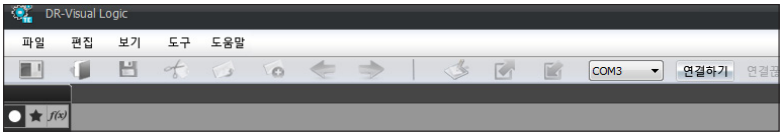
Debug 폴더 안의 hex 파일 확인합니다. Hex 파일이 제어기에 포팅시키는 파일입니다.

DR-Visual Logic 에서 펌웨어 복구를 실시합니다.



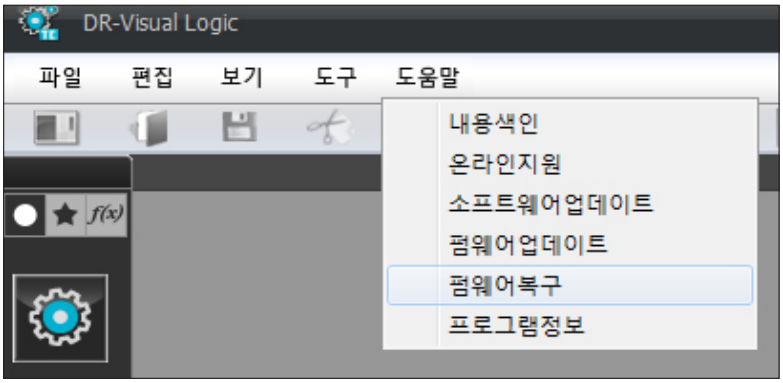
04 Visual Logic

DR-Visual Logic 을 실행시킵니다.



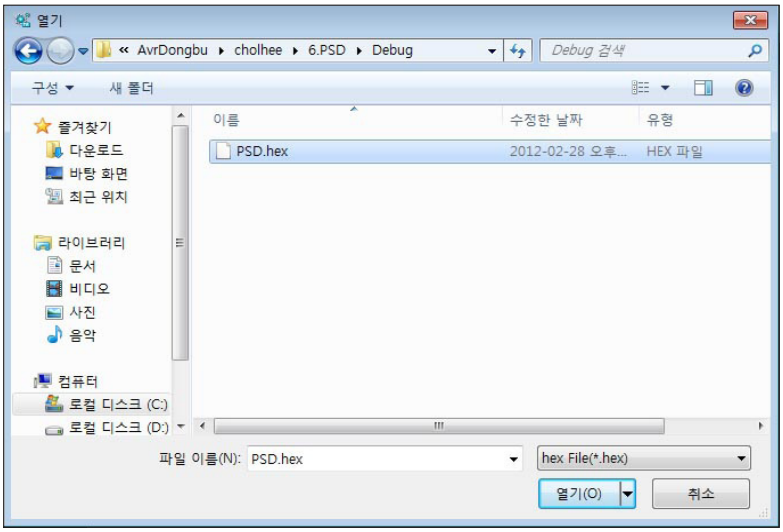
05 연결하기

COM 번호를 설정하고 연결하기를 클릭합니다.



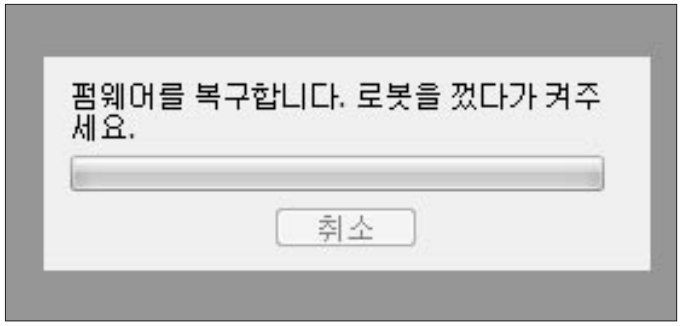
06 펌웨어 복구

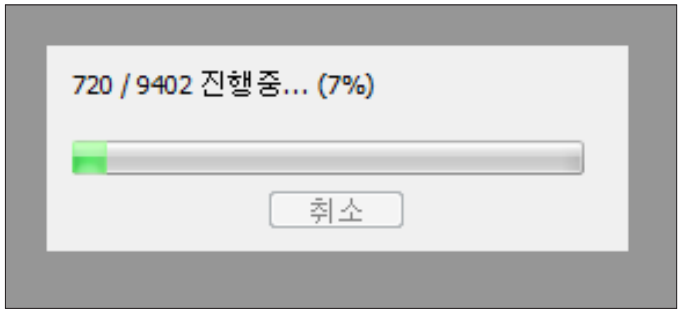
도움말 메뉴에서 펌웨어 복구를 클릭합니다.



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.

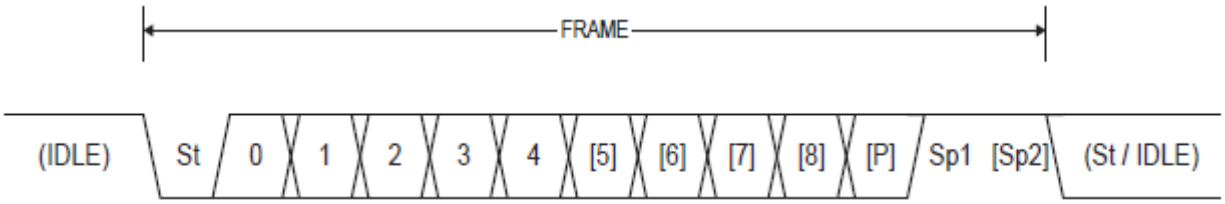
CH.7 : USART를 사용하여 시리얼 통신하기

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.

Baud Rate (bps)	f_{osc} = 16.0000 MHz			
	U2X = 0		U2X = 1	
	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.

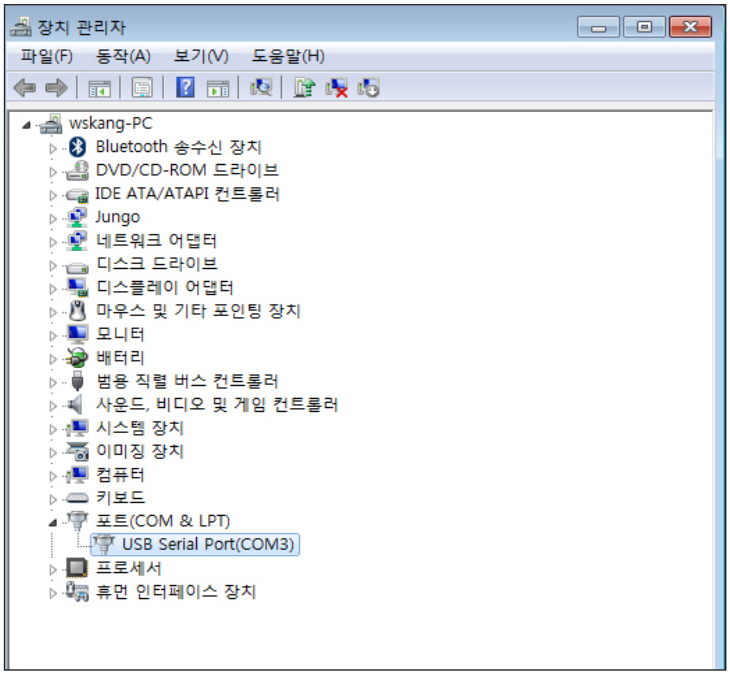
그래서 DRC의 회로에는 직렬 통신 레벨 변환 칩을 사용하여 ATmega128이 보내는 신호를 TTL에서 RS-232로 바꾸어 PC에 전송하고, PC에서 보내는 신호를 RS-232에서 TTL로 바꾸어 ATmega128로 전송합니다.

하드웨어 준비하기



01 연결

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



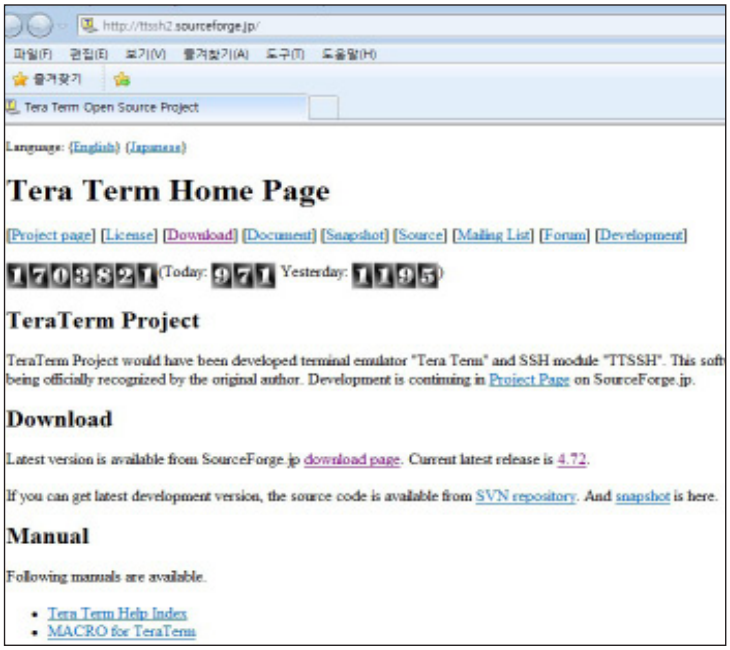
02 COM port 연결

내컴퓨터 〉 속성 〉 장치관리자에서 COM 연결 확인합니다.



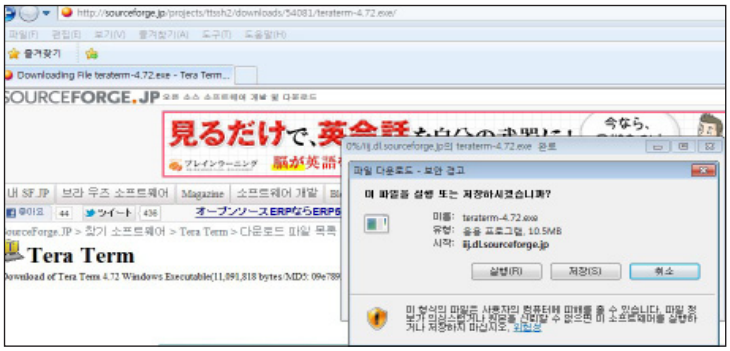
01 주소 검색

인터넷 포털 사이트에서 “tera term”을 검색합니다.



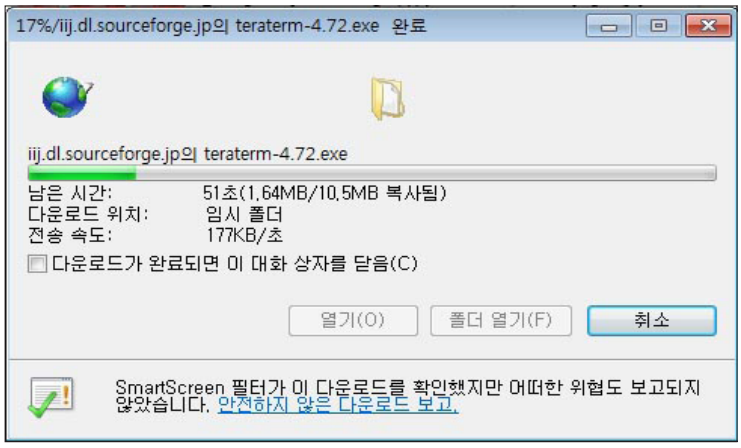
02 홈페이지 접속

Tera Term 홈페이지에 접속하여 “download page”를 클릭합니다.



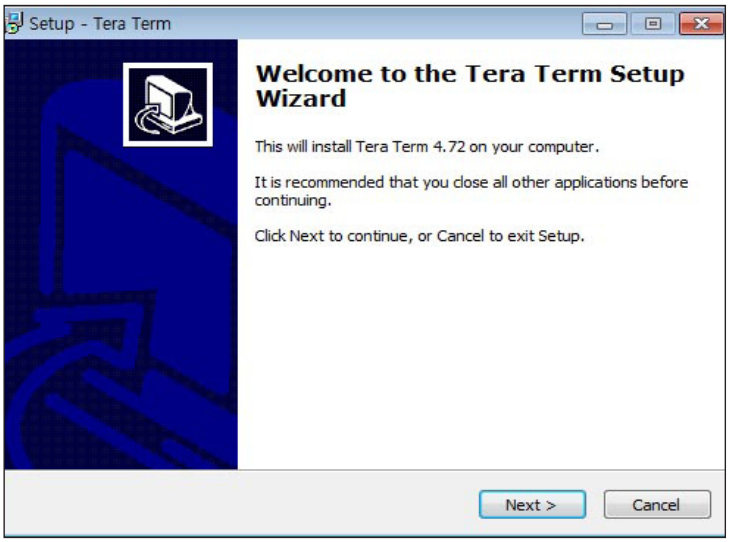
03 파일 다운로드

“저장”을 눌러 파일을 다운로드 받습니다.



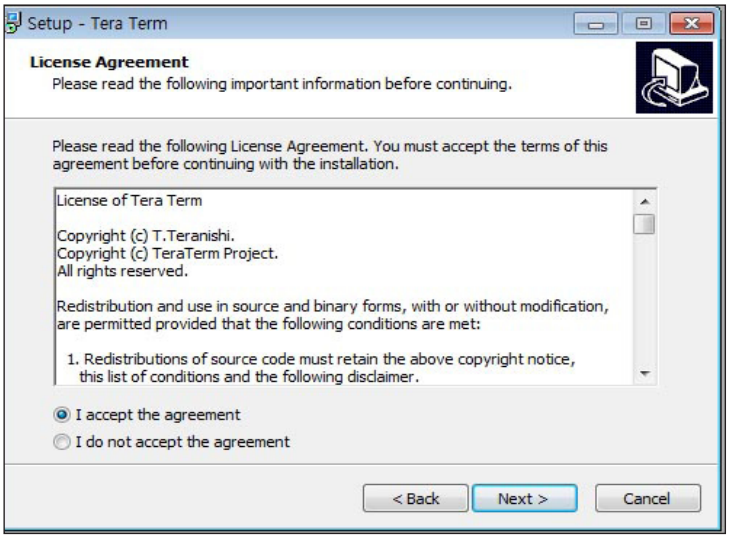
04 파일 다운로드 시작

설치 파일이 다운로드됩니다.



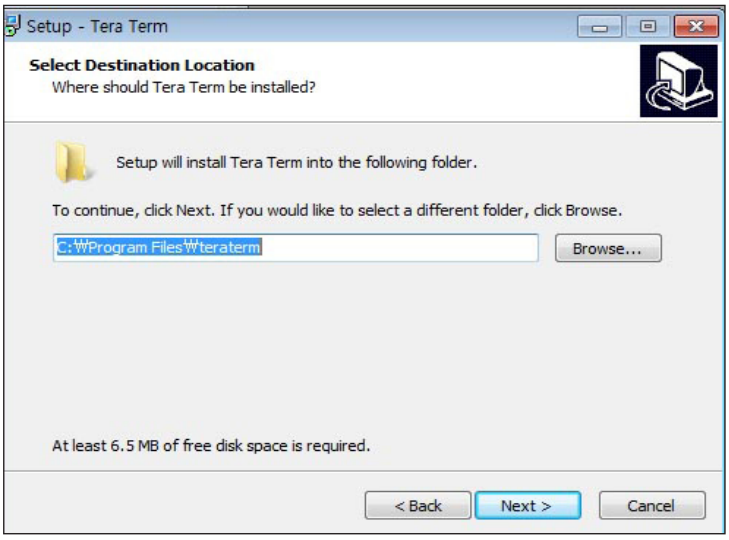
05 설치 마법사 실행

"Next"를 누르면 설치 마법사가 실행됩니다.



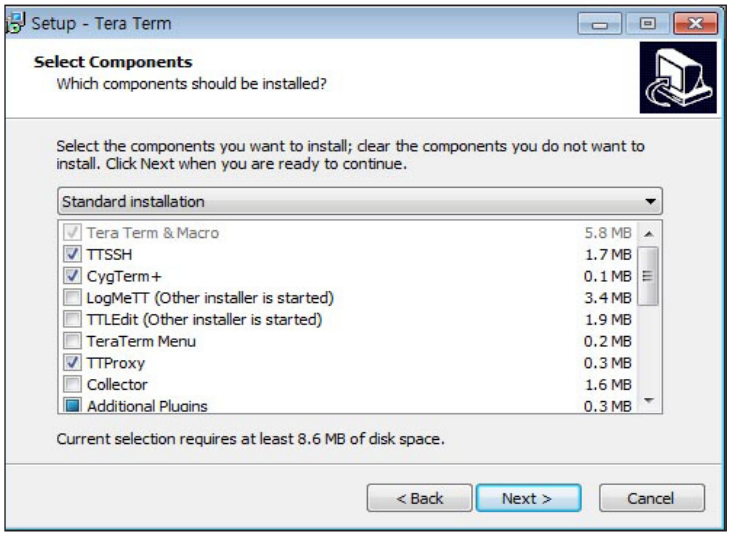
06 약관 동의

"I accept the agreement"를 체크하고 "Next"를 클릭합니다.



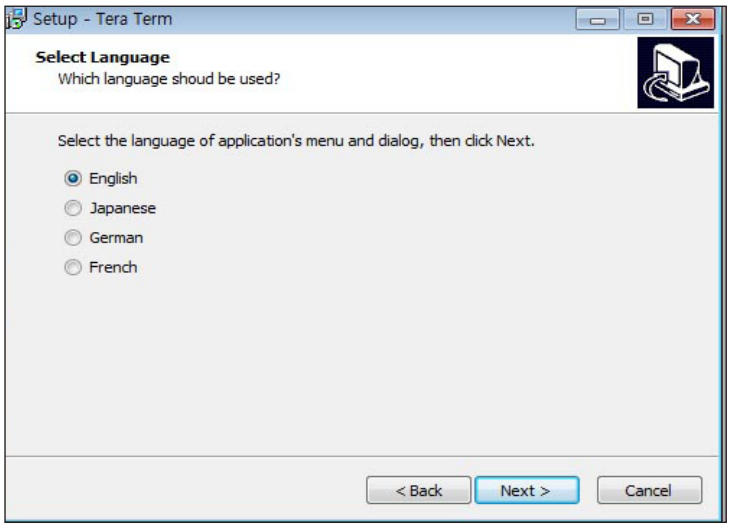
07 설치 폴더 확인

프로그램을 설치할 폴더를 확인하고 "Next"를 클릭합니다.



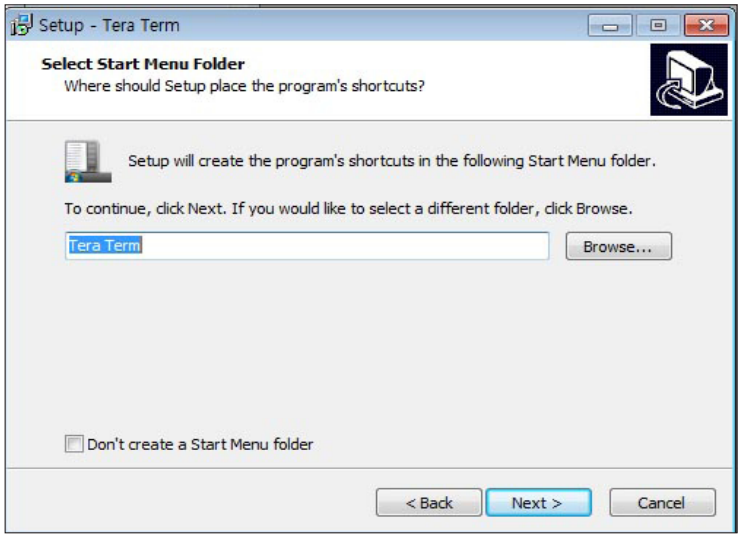
08 구성요소 선택

"Next"를 클릭합니다.



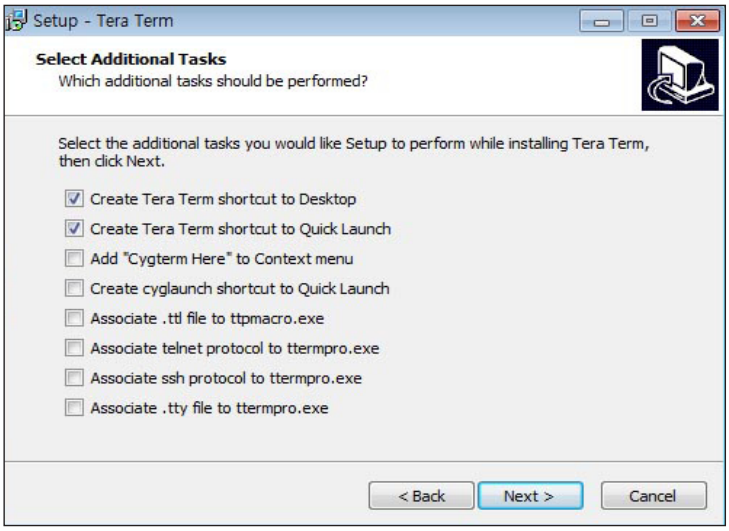
09 언어 선택

언어 선택 후 "Next"를 클릭합니다.



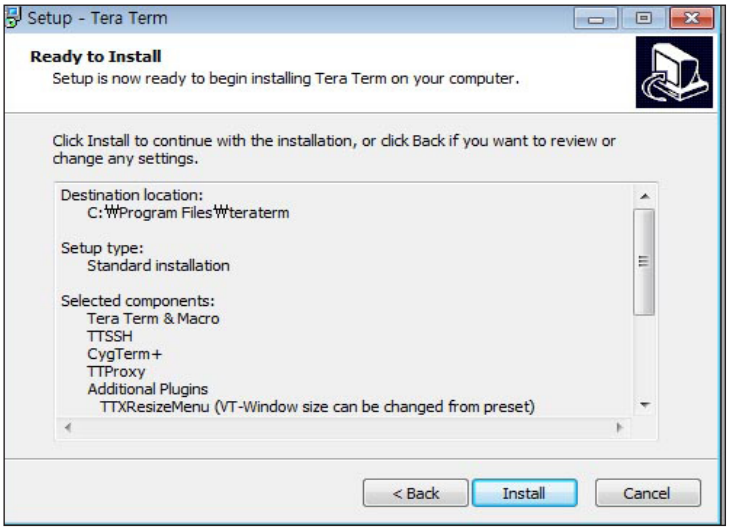
10 시작 메뉴 폴더 선택

시작 메뉴 폴더 선택 후 “Next”를 클릭합니다.



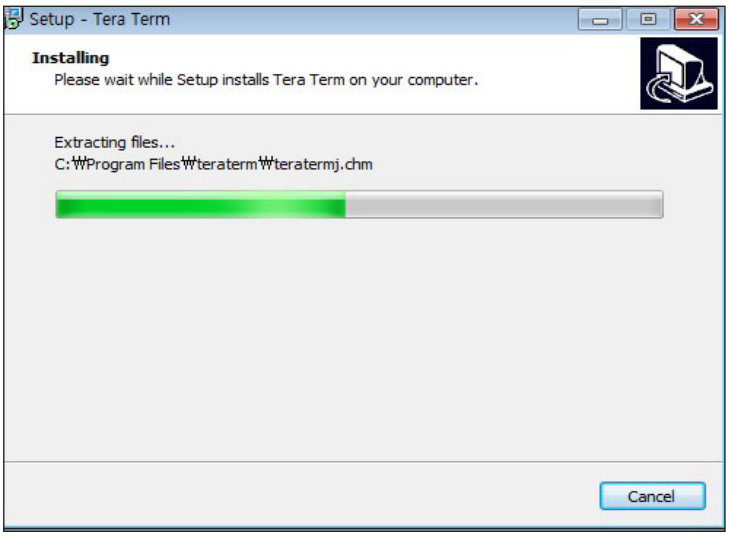
11 추가 기능 선택

원하는 추가 기능을 선택 후 “Next”를 클릭합니다.



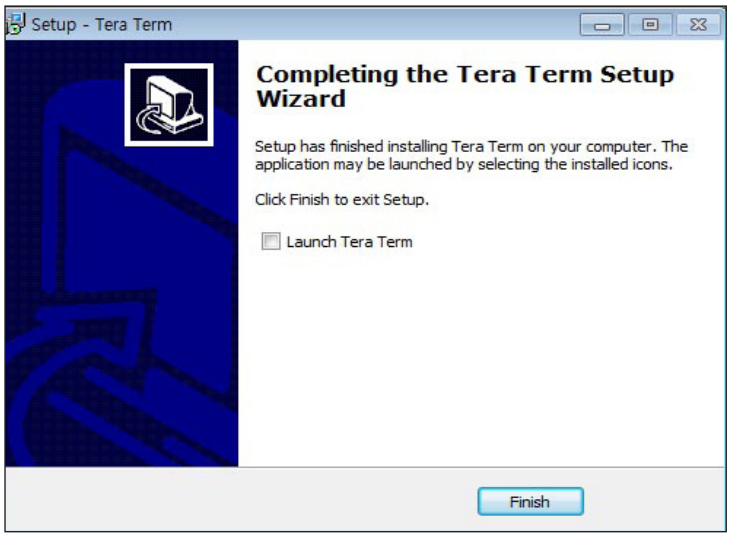
12 설치 시작

“Install”을 클릭하면 설치가 시작됩니다.



13 설치 확인

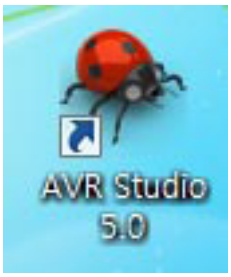
프로그램이 설치됩니다.



14 설치 완료

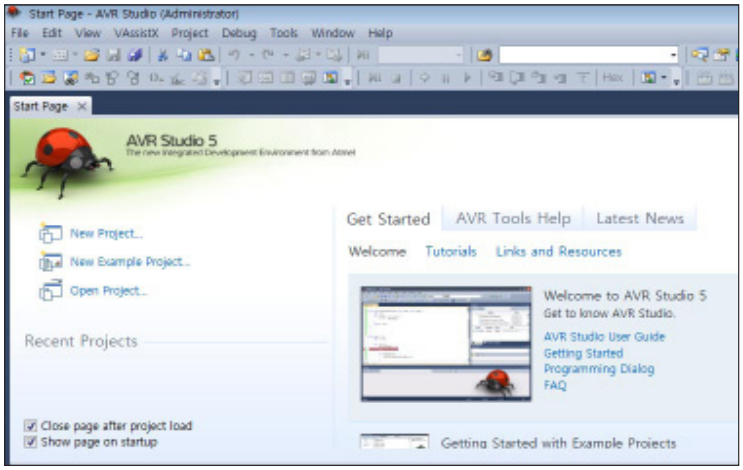
"Finish"를 클릭하면 설치가 완료됩니다.

AVR Studio 실행 및 코딩 준비



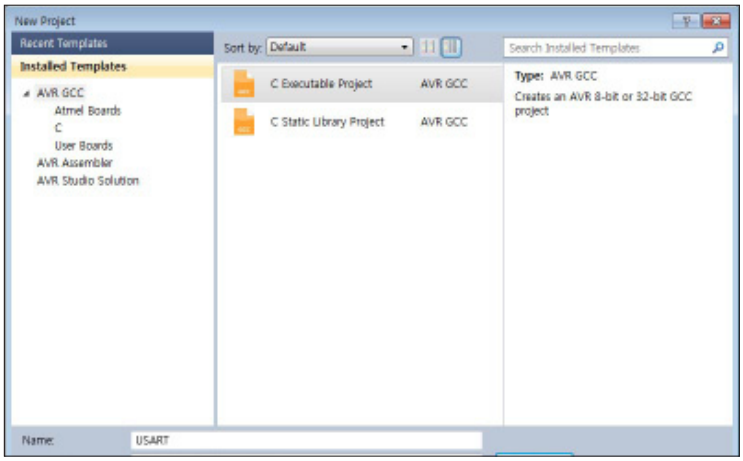
01 실행

AVR Studio 아이콘을 클릭하여 실행합니다.



02 New Project

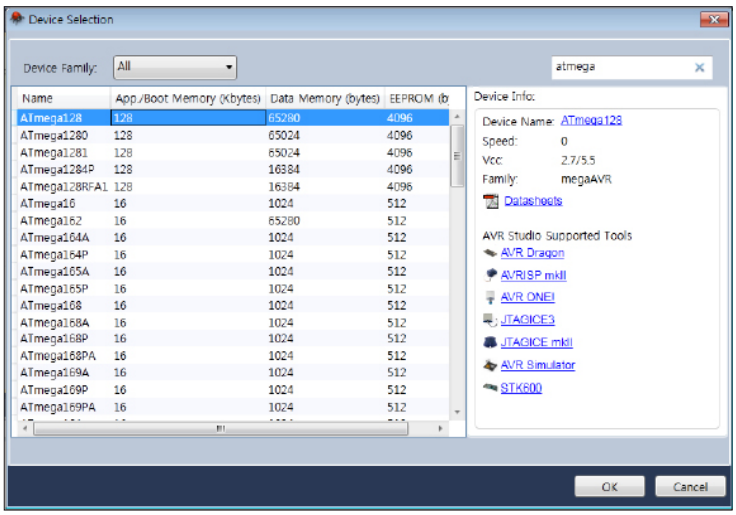
New Project 를 클릭합니다.



03 GCC 선택

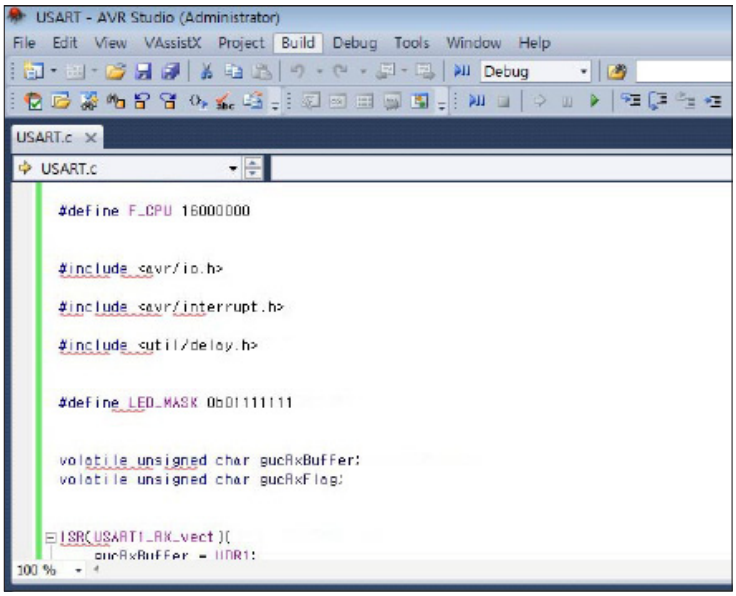
C Executable Project 를 선택합니다.

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");
}
```

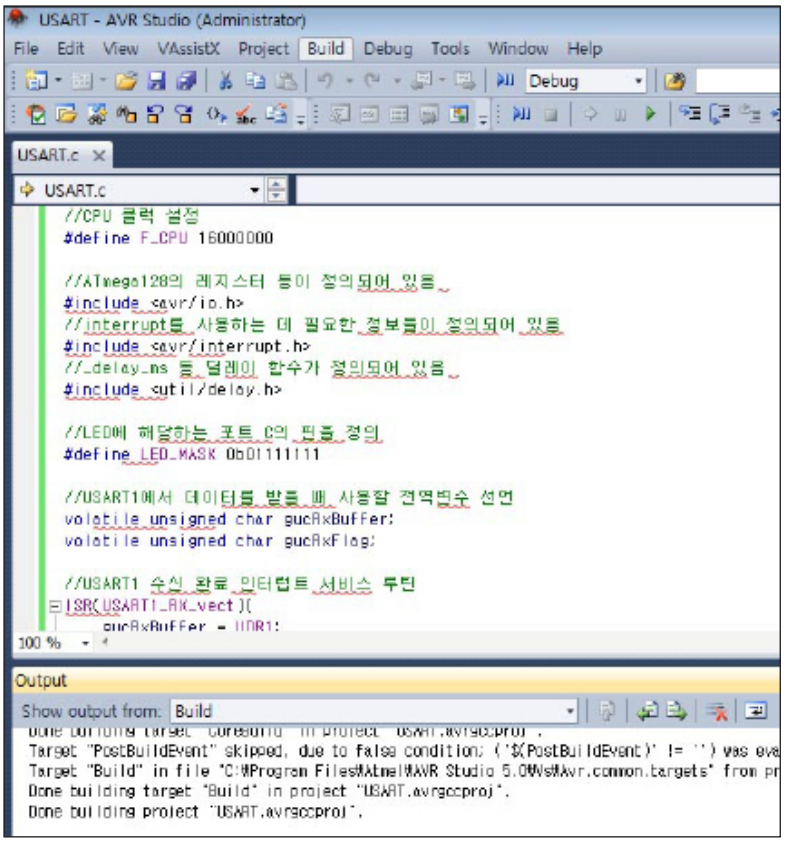
gucRxFlag가 1이라서 if문에 진입한 후에는 위의 코드를 만나게 됩니다. gucRxBuffer에 저장된 방금 들어온 데이터 값이 'z'이거나 'Z'인 경우에는 ucLED에 저장된 비트를 오른쪽으로 한번 이동합니다. 만약 ucLED 값이 0b00000001이라서 더 이동할 공간이 없다면 제일 왼쪽인 0b01000000으로 바꾸어 줍니다. 그 후에는 PORTC에 ucLED에 해당하는 값을 넣어 실제로 LED를 켭니다. LED를 켜고 나면 USART1_PutString() 함수를 사용해 'z' 키에 의해 LED가 움직였다는 것을 출력합니다.

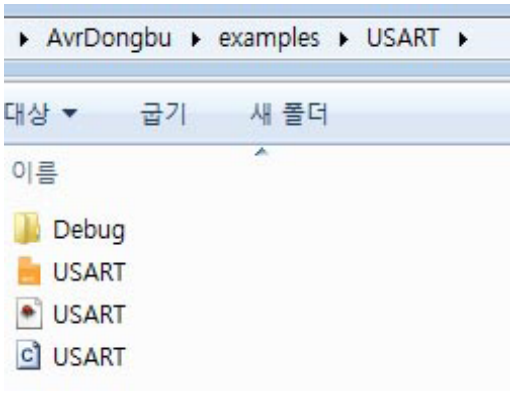
```
else if(gucRxBuffer == 'x' || gucRxBuffer == 'X'){
        if(ucLED < 0b01000000){
                ucLED <<= 1;
        }
        else{
                ucLED = 0b00000001;
        }
        PORTC = (~ucLED) & LED_MASK;

        USART1_PutString("LED Moved by ₩'x₩'₩r₩n");
}
```

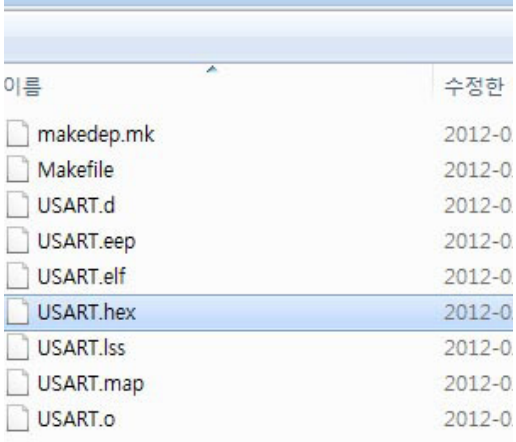
만약 입력된 데이터가 'z'나 'Z'가 아니라면 위의 코드를 만납니다. 데이터 값이 'x'이거나 'X'인 경우에는 'z'일 때와는 반대로 ucLED의 비트를 왼쪽으로 이동합니다. 만약 ucLED 값이 0b01000000이라서 더 이동할 공간이 없으면 제일 오른쪽인 0b00000001으로 바꿉니다. 그 후에는 PORTC의 값을 바꾸어 LED를 켜고, 'x' 키에 의해 LED가 움직였다는 것을 출력합니다. If–else 문은 여기서 끝나므로, 만약 'z', 'Z', 'x', 'X'가 아닌 다른 입력이 들어왔다면 아무 행동도 하지 않고 gucRxFlag만 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.

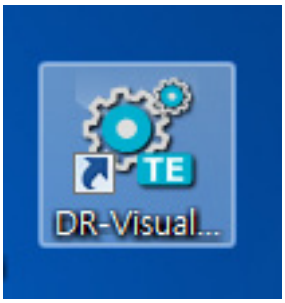
02 Debug 폴더

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

03 hex 확인

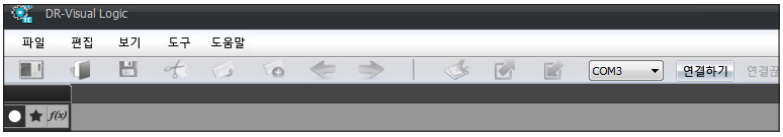
Debug 폴더 안의 hex 파일 확인합니다. Hex 파일이 제어기에 포팅시키는 파일입니다.

DR-Visual Logic 에서 펌웨어 복구를 실시합니다.



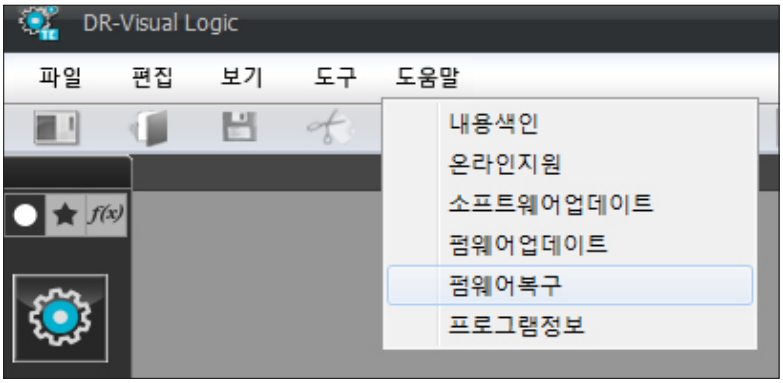
04 Visual Logic

DR-Visual Logic 을 실행시킵니다.



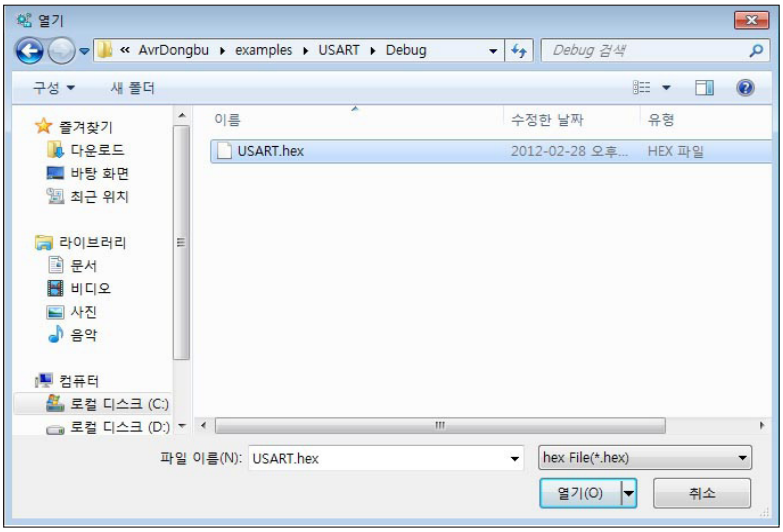
05 연결하기

COM 번호를 설정하고 연결하기를 클릭합니다.



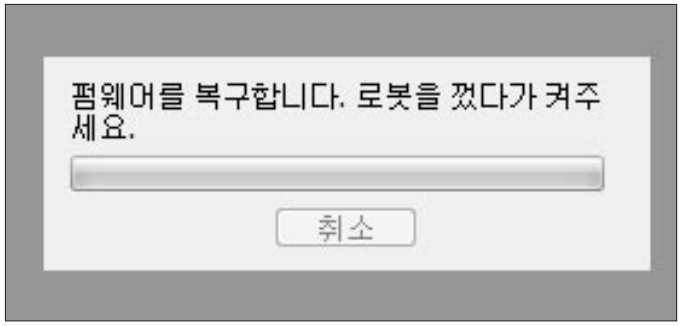
06 펌웨어 복구

도움말 메뉴에서 펌웨어 복구를 클릭합니다.



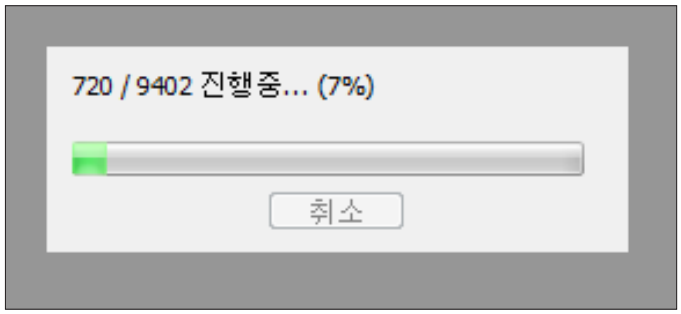
07 불러오기

Debug 폴더에서 Hex 파일을 불러옵니다



08 펌웨어복구

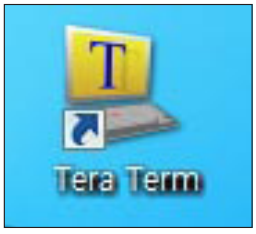
로봇을 껐다가 켜주세요 라는 문구가 나옵니다.
제어기의 전원버튼을 한번 껐다가 다시 켜주세요



09 복구시작

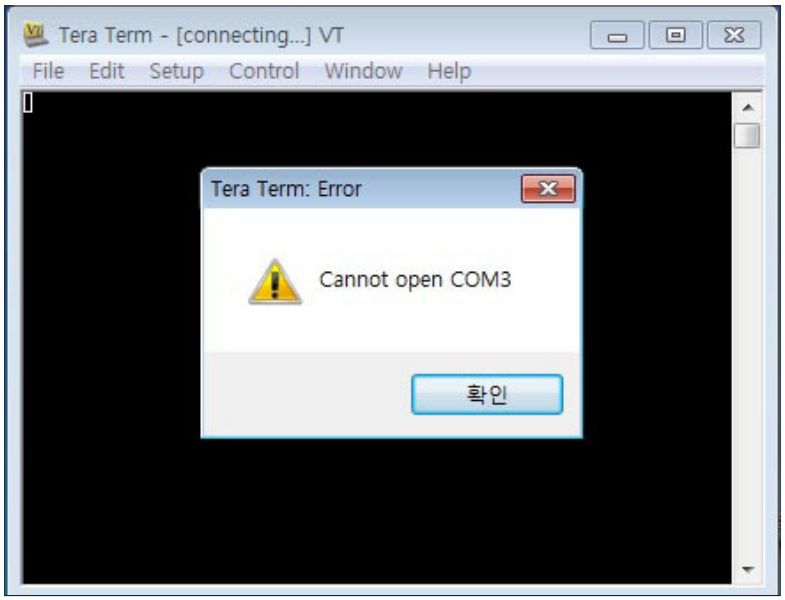
펌웨어 복구가 시작됩니다. 진행이 완료되면 팝업창이 내려갑니다.

Tera Term 을 활용하여 통신 테스트를 실시합니다.



10 실행

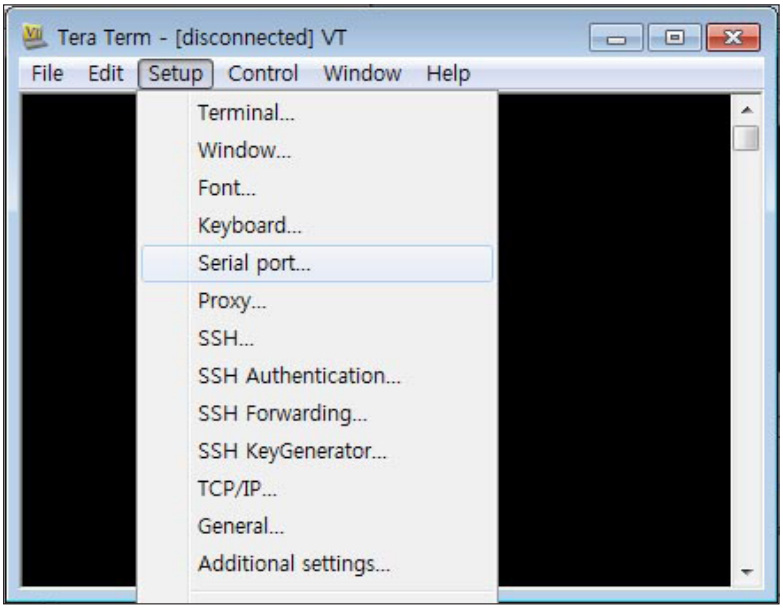
Tera Term 아이콘을 클릭하여 프로그램을 실행시킵니다.



11 열기

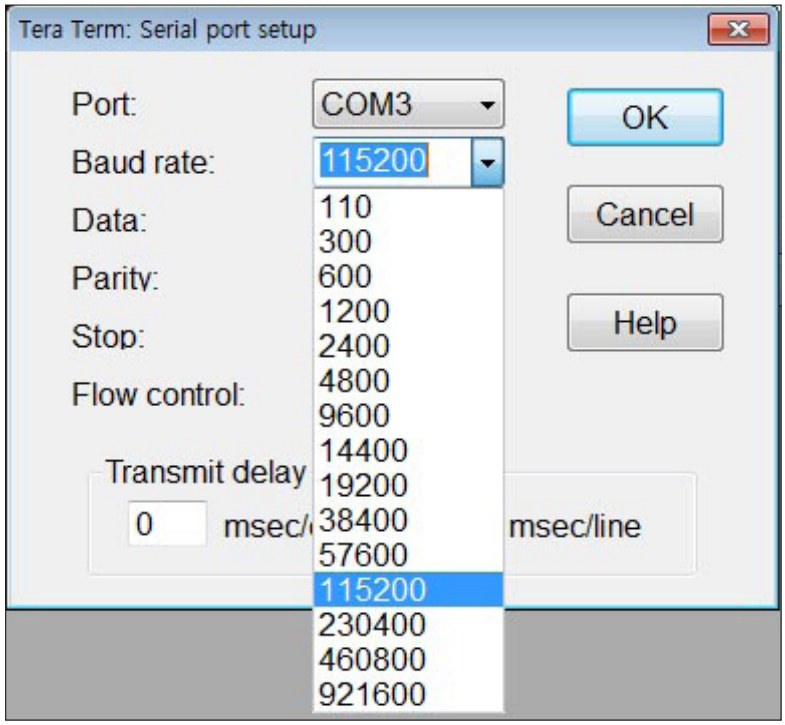
실행하면 바로 연결되지 않습니다.
확인버튼을 클릭합니다.

※바로 연결 될 수도 있습니다.
여기서 연결되지 않은 이유는
DR-Visual Logic 에서 연결끊기를
안눌렀기 때문입니다.



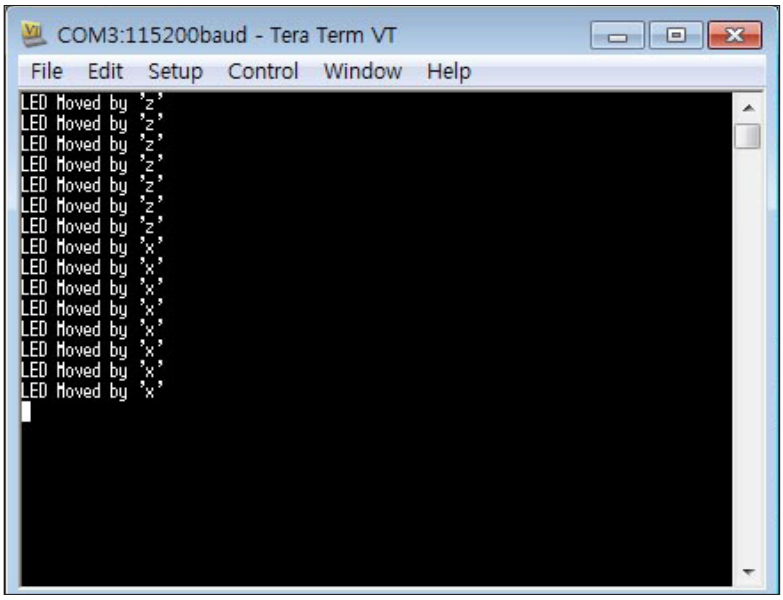
12 Set up

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.

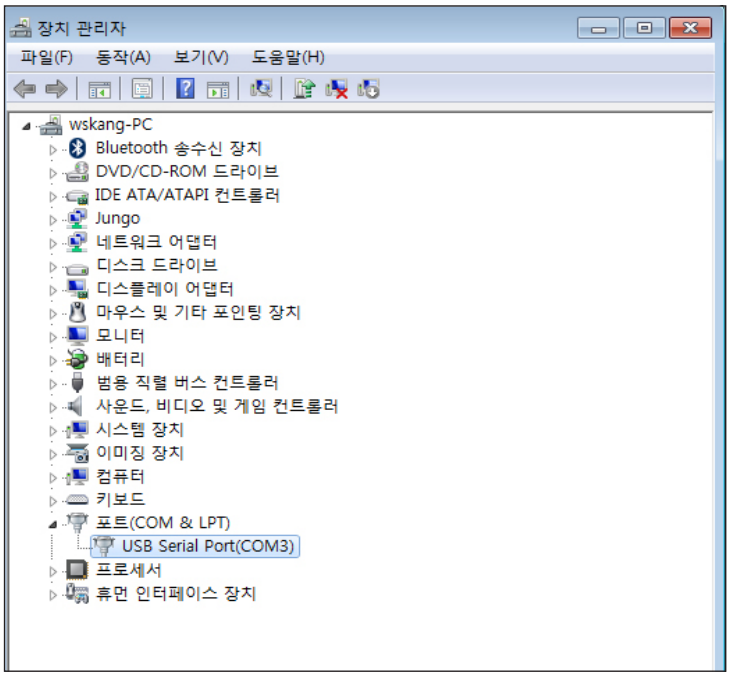
하드웨어 준비하기



01 모터 준비

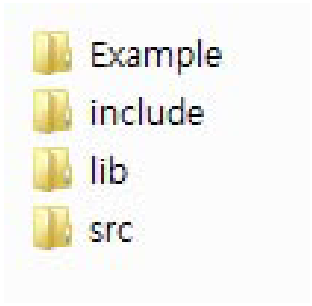
DRC 제어기 아답터로 전원을 연결하고, USB to Serial Cable 로 PC 와 DRC 를 연결합니다.

서보모터를 제어기 우측 포트에 연결합니다.



02 COM port 연결

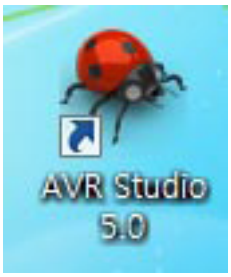
내컴퓨터 〉 속성 〉 장치관리자에서 COM 연결 확인합니다.



AVR Studio 실행 및 코딩 준비

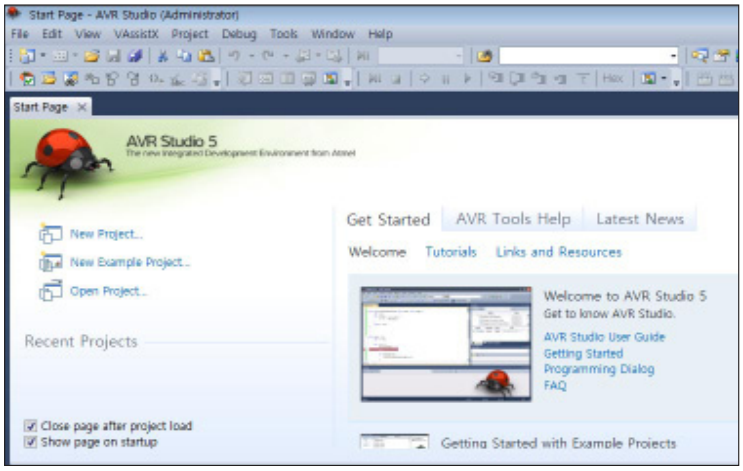
00 라이브러리

서보모터를 제어하기 위해서는 기본 라이브러리 파일이 필요합니다.
라이브러리가 포함된 폴더 구조입니다.



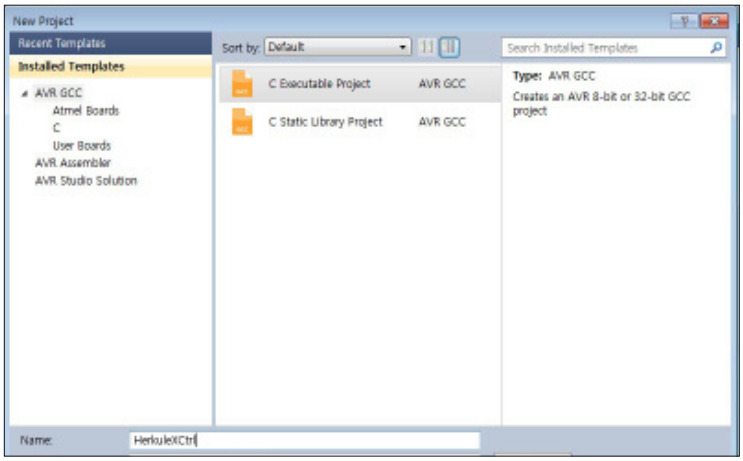
01 실행

AVR Studio 아이콘을 클릭하여 실행합니다.



02 New Project

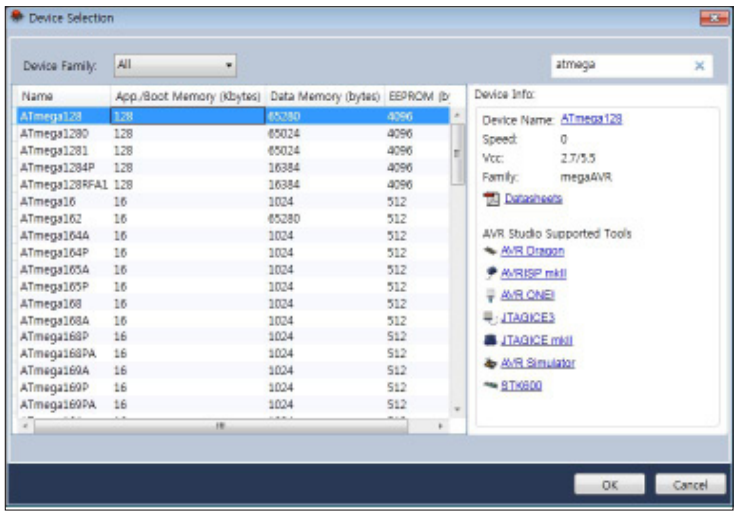
New Project 를 클릭합니다.



03 GCC 선택

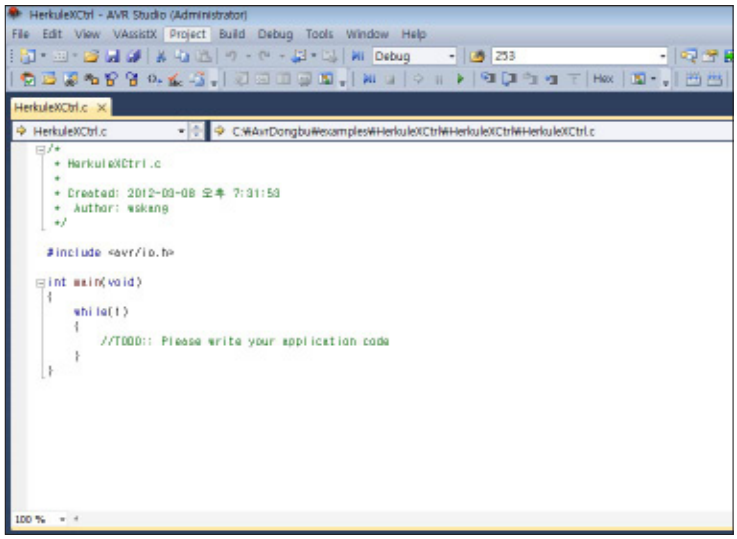
C Executable Project 를 선택합니다.

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



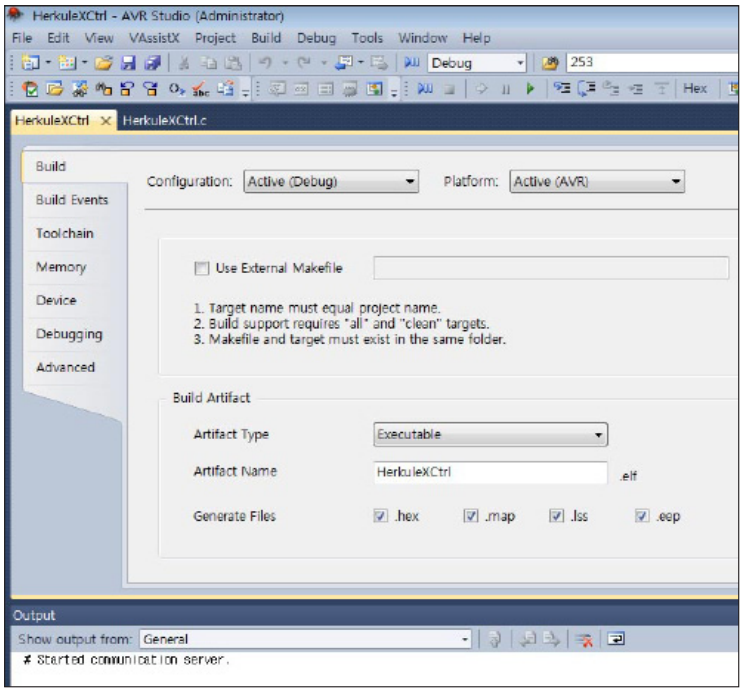
04 ATmega 선택

우측 상단에 atmega 를 검색하여, 리스트중에 ATmega128 을 선택합니다.



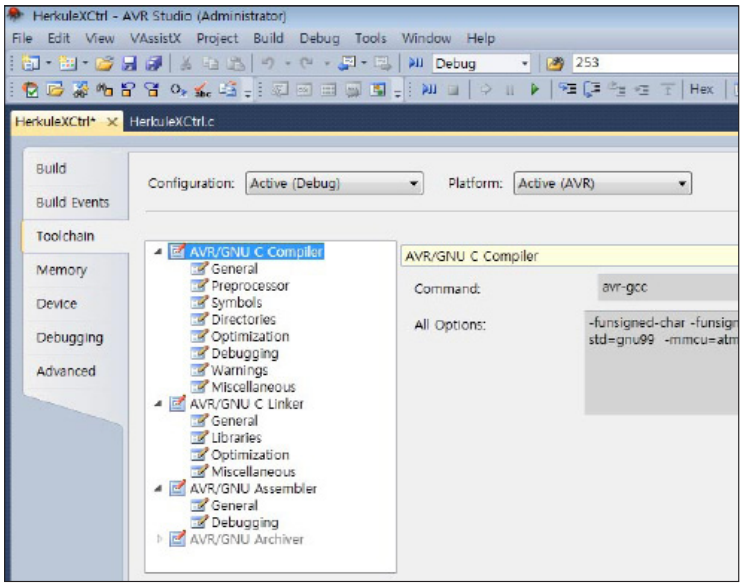
05 Editor

편집창에 코딩을 시작합니다.



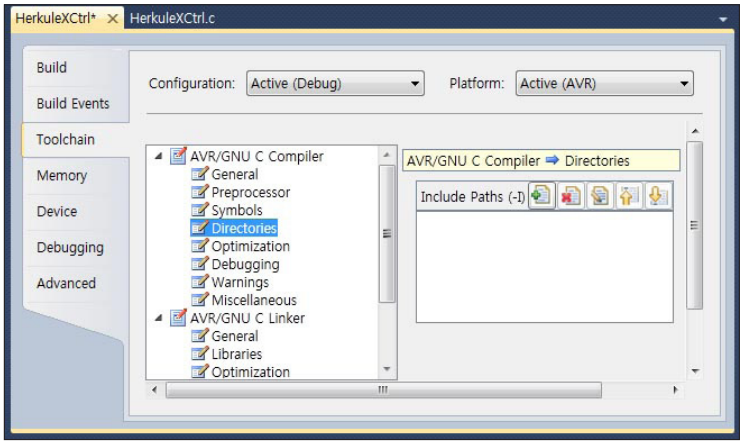
01 설정

AVR Studio 상단 메뉴중에 Project 를 클릭하면 HerkuleXCtrl (프로젝트 이름별로 다름) Property 메뉴가 생성되어 있습니다. 클릭하면 좌측과 같이 설정화면이 나옵니다.



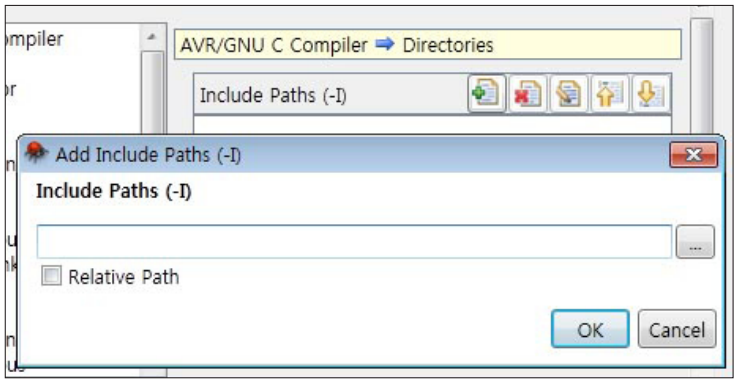
02 Toolchain

좌측 탭버튼중에 Toolchain 을 클릭하여, AVR/GNU C Compiler 을 선택합니다.



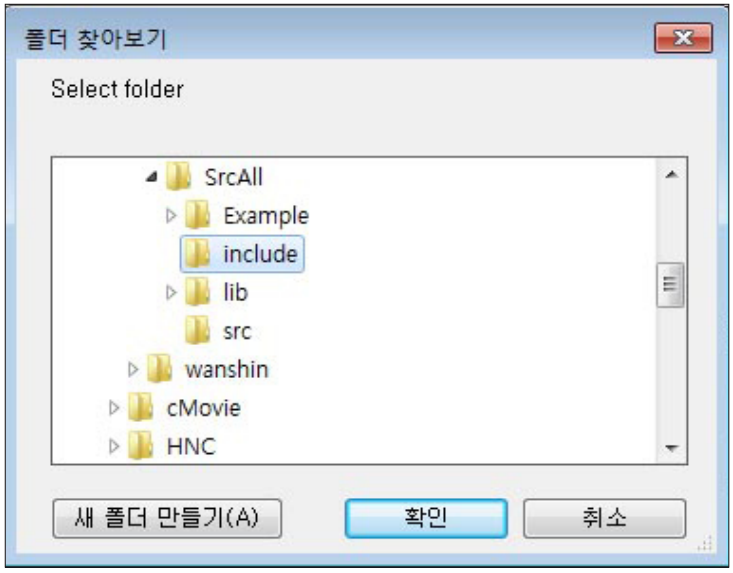
03 C Compiler

C Compiler 폴더중에 Directories 를 선택합니다.



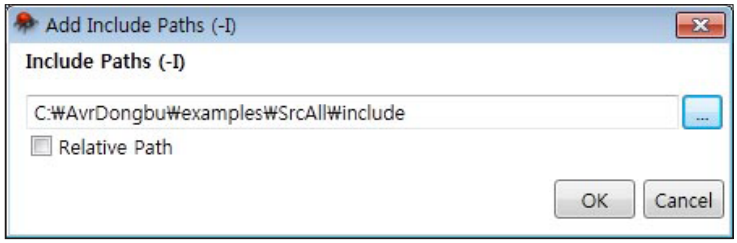
04 Path

Include Paths 오른쪽에 + 버튼을 클릭하면 팝업 입력창이 뜹니다.
Relative Path (상대경로) 체크를 해지하고, 우측 폴더 검색 버튼을 클릭합니다.
※5.1일경우 해지할 필요가 없습니다.



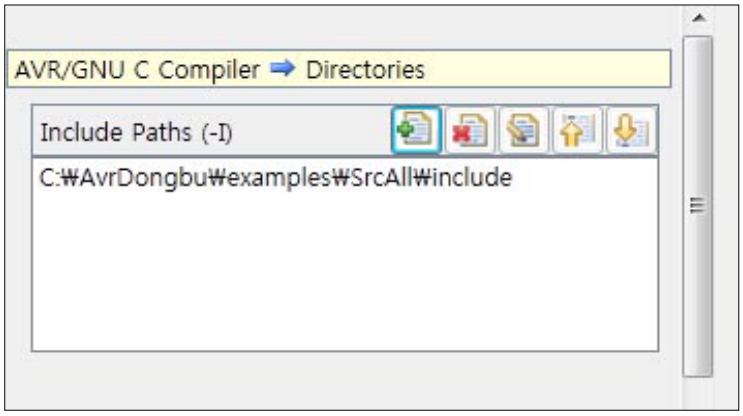
05 Include

제공 라이브러리 폴더중에 include 를 선택합니다.



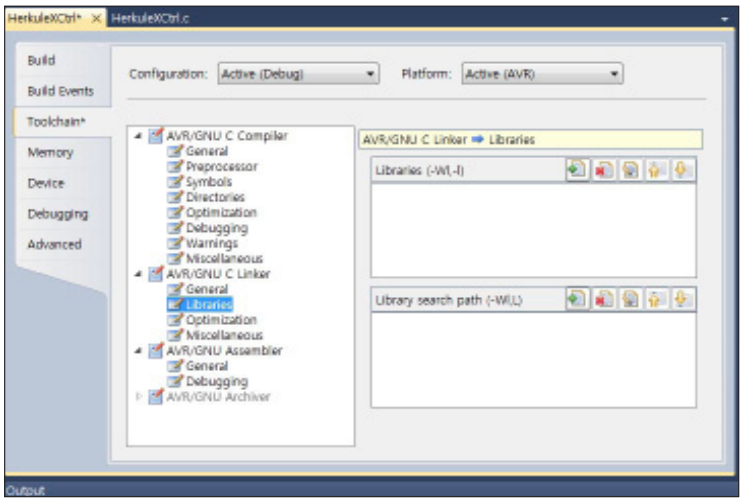
06 Include 완료

Include 폴더 위치가 지정됩니다. OK 버튼을 클릭합니다



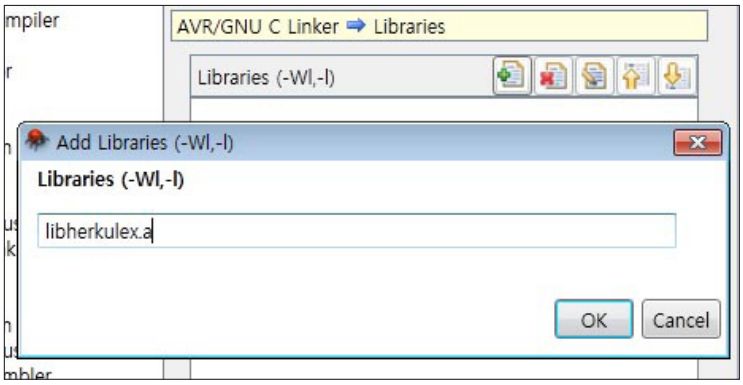
07 Path 완료

Include 경로지정이 완료되었습니다.



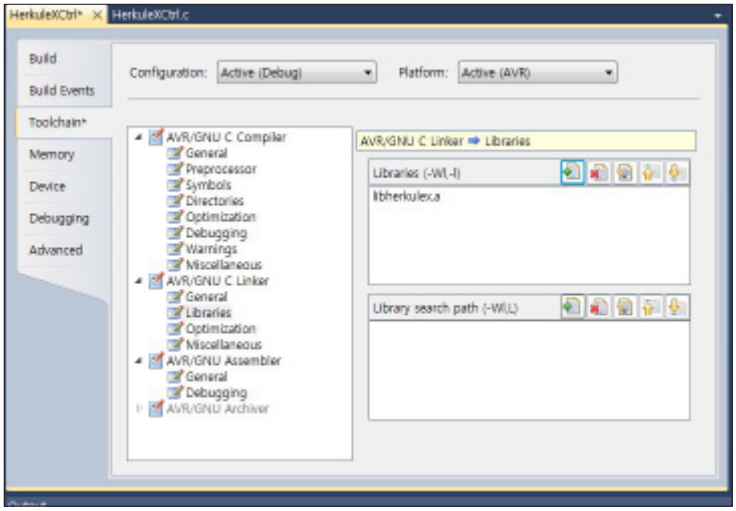
08 C Linker

C Linker 폴더중에 Libraries 를 선택합니다



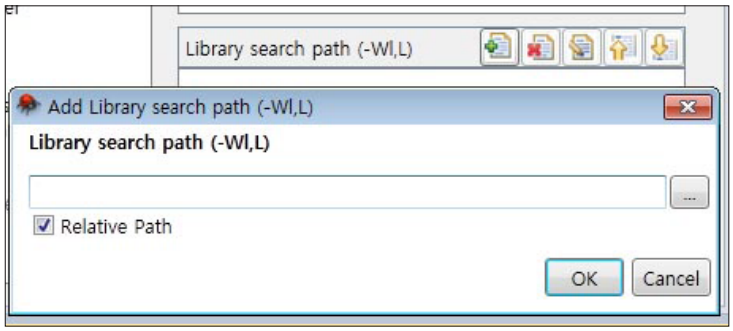
09 Libraries

Libraries 창 우측 + 를 클릭하면, Libraries 파일명 입력창이 나옵니다.
libherkulex.a 를 직접 입력합니다.
OK 버튼을 클릭합니다.



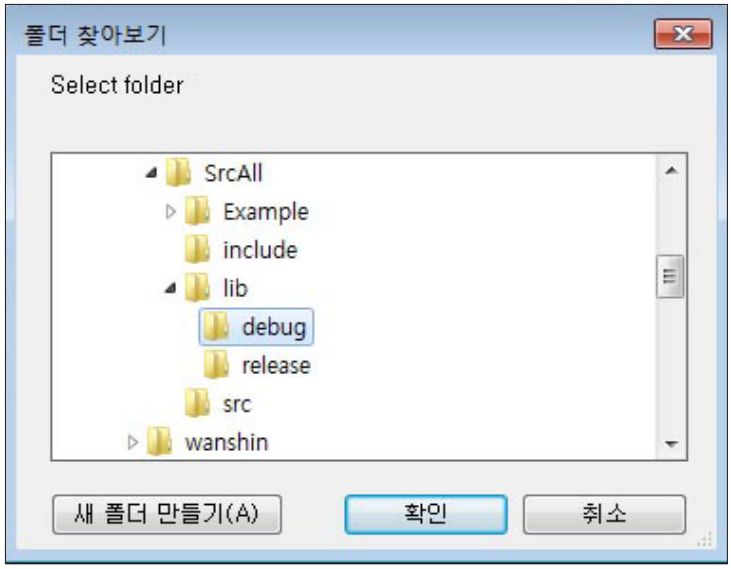
10 Search

서보모터 라이브러리 파일 libherkulx.a 링크가 완료되었습니다.
다음으로 Library search path 를 설정합니다.
우측 + 버튼을 클릭합니다.



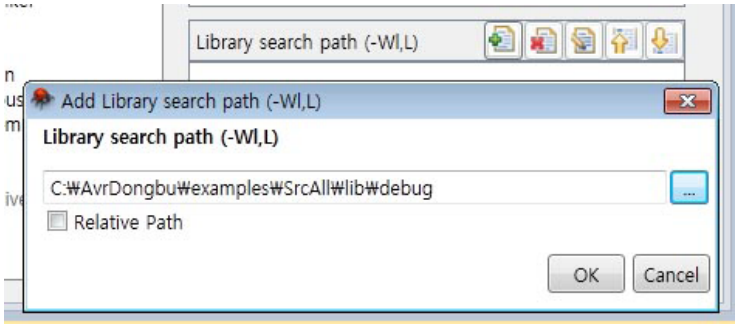
11 Path

Library search path 팝업창 우측에 폴더 검색 버튼을 클릭합니다.



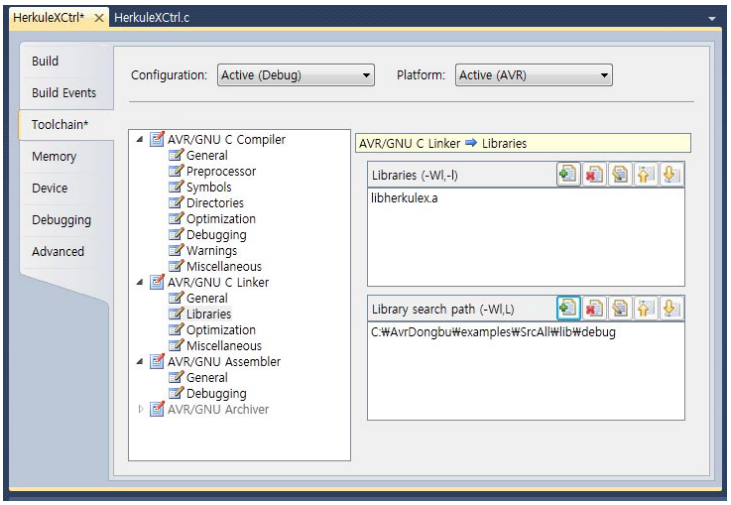
12 Debug

제공 라이브러리 폴더에서 lib 하위 폴더에 debug 를 선택합니다.



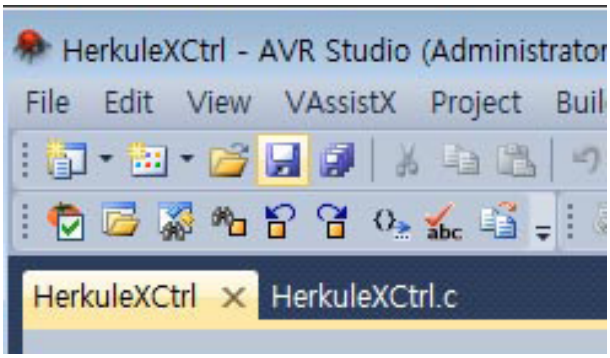
13 Path 완료

Relative Path(상대경로지정) 체크박스를 해지합니다.
Library search path 경로지정이 완료되었습니다.
OK 버튼을 클릭합니다.



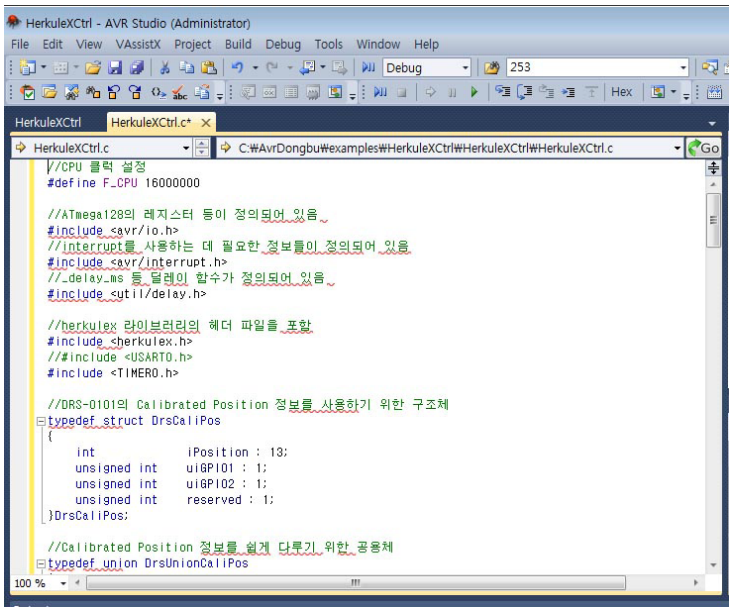
14 Link 완료

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;
        }
}
```

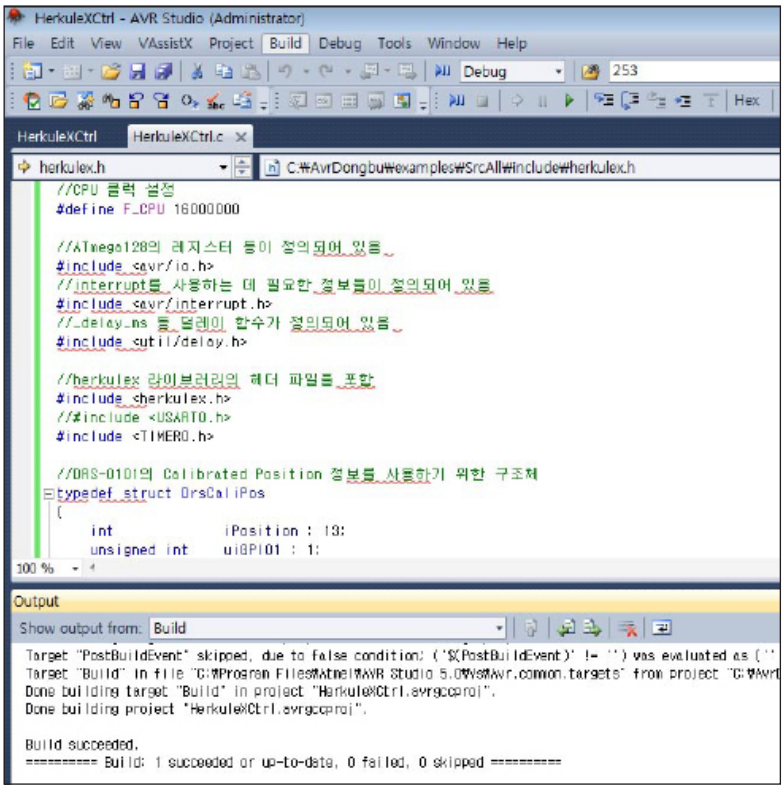
Timeout 시간을 설정해놓고 ACK Packet이 들어오기를 기다리는 함수입니다. libherkulex.a 파일에는 타이머/카운터0 오버플로우 인터럽트 서비스 루틴이 포함되어 있기 때문에, HerkuleXCtrl.c에서 gucTimerTick 값을 15로 만들면 이 변수는 타이머/카운터0 오버플로우 인터럽트가 발생하는 2ms마다 1씩 줄어들어서, 약 30ms 후에는 0이 됩니다. 그동안 while문 안에서는 계속 hklx_ucReceivePacket() 함수를 호출해서 ucResult에 그 결과 값을 저장합니다.
ACK 패킷이 들어오기를 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

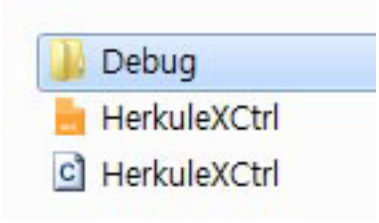
펌웨어를 다운로드하고 실행하면 ID 253인 HerkuleX가 400, 624 위치를 약 1초 주기로 왕복하는 것을 볼 수 있습니다. 정상적인 모터라면 DRC의 LED는 모두 꺼져 있어야 합니다.



01 Build

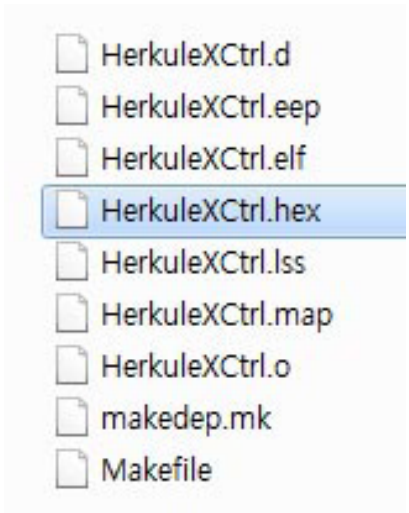
Build 메뉴를 선택하여 Build Solution을 클릭하여 Build 시킵니다.

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



02 Debug 폴더

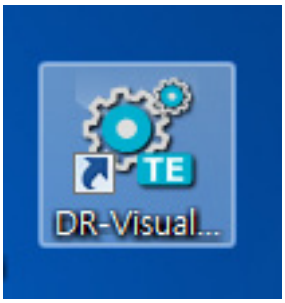
Build 후에 소스가 있는 폴더에 들어가면 Debug 폴더가 생성 되어있습니다. 클릭하여 폴더 안의 내용을 봅니다.



03 hex 확인

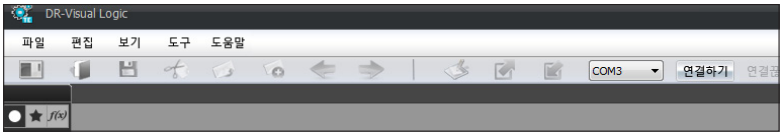
Debug 폴더 안의 hex 파일 확인합니다. Hex 파일이 제어기에 포팅시키는 파일입니다.

DR-Visual Logic 에서 펌웨어 복구를 실시합니다.



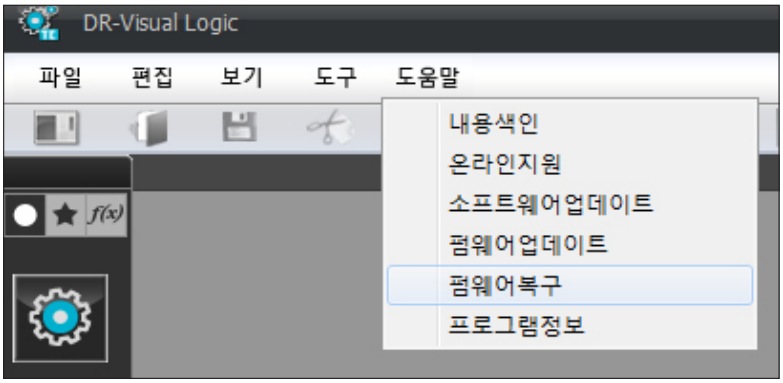
04 Visual Logic

DR-Visual Logic 을 실행시킵니다.



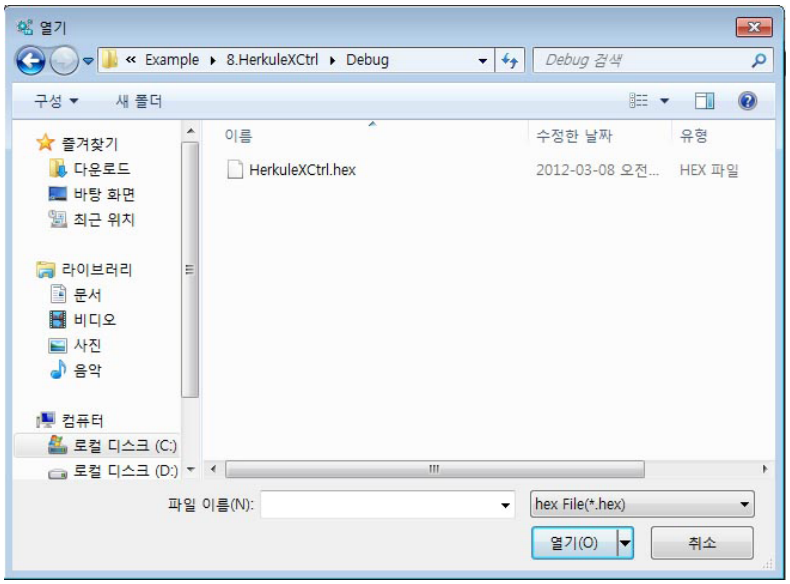
05 연결하기

COM 번호를 설정하고 연결하기를 클릭합니다.



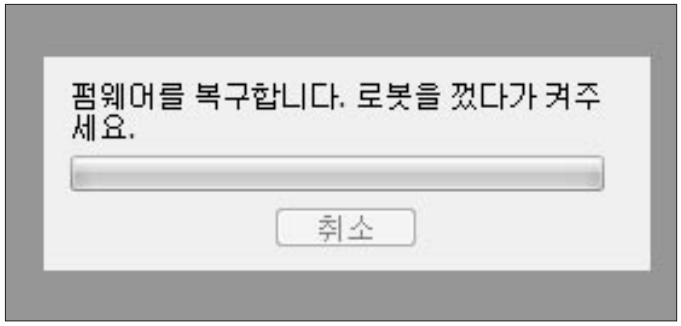
06 펌웨어 복구

도움말 메뉴에서 펌웨어 복구를 클릭합니다.



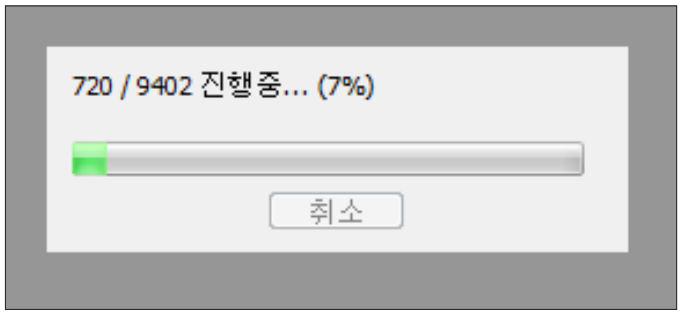
07 불러오기

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.

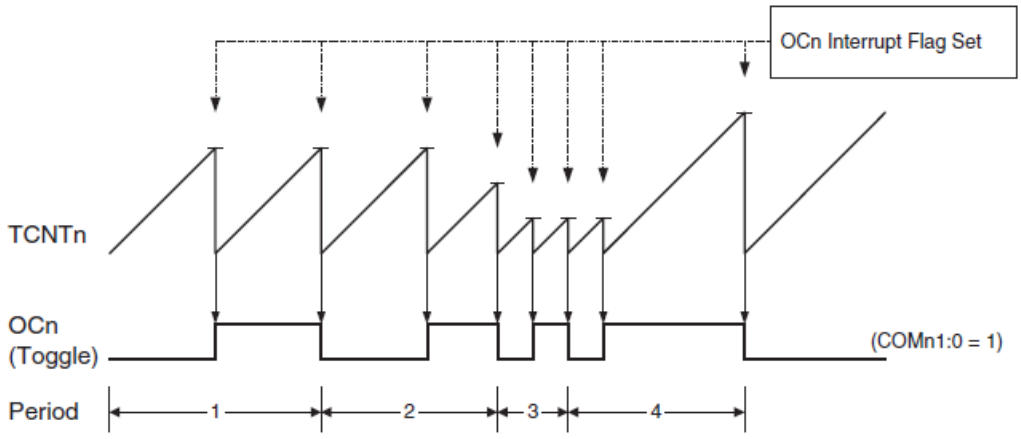
타이머/카운터2는 타이머/카운터0과 다르게 비동기 동작을 지원하지 않으므로, ASSR, SFIOR 등의 레지스터는 관련이 없습니다. 더 자세한 사항은 ATmega128의 데이터시트나 다른 서적을 참조하기 바랍니다.

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.

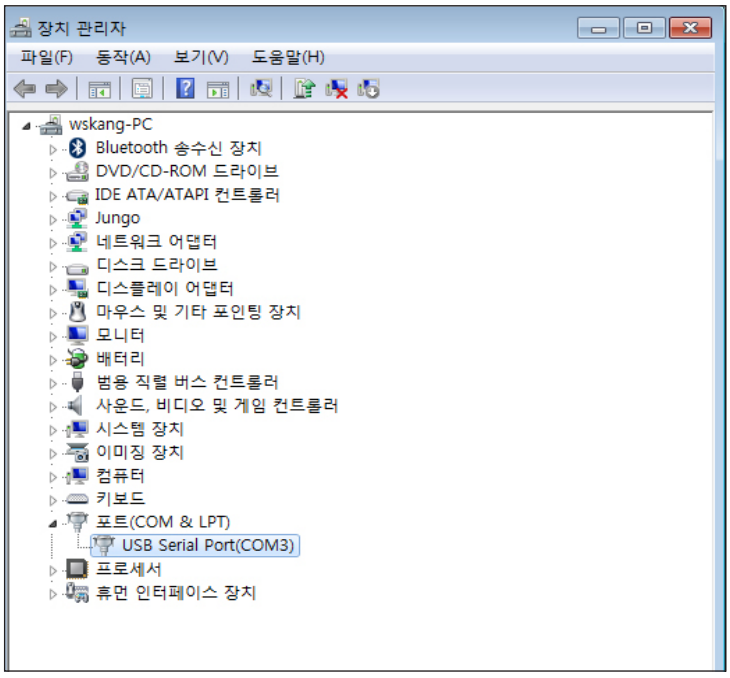
준비

하드웨어 준비하기



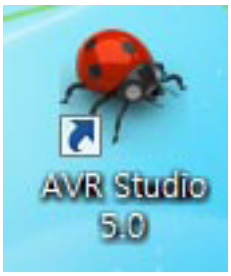
01 연결

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



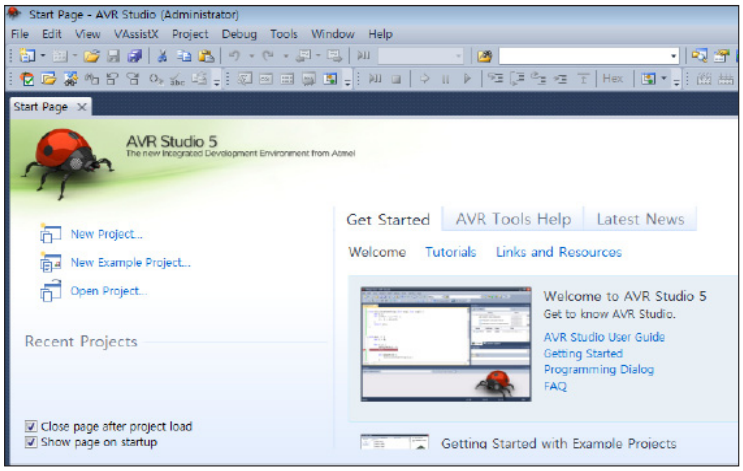
02 COM port 연결

내컴퓨터 〉 속성 〉 장치관리자에서 COM 연결 확인합니다.



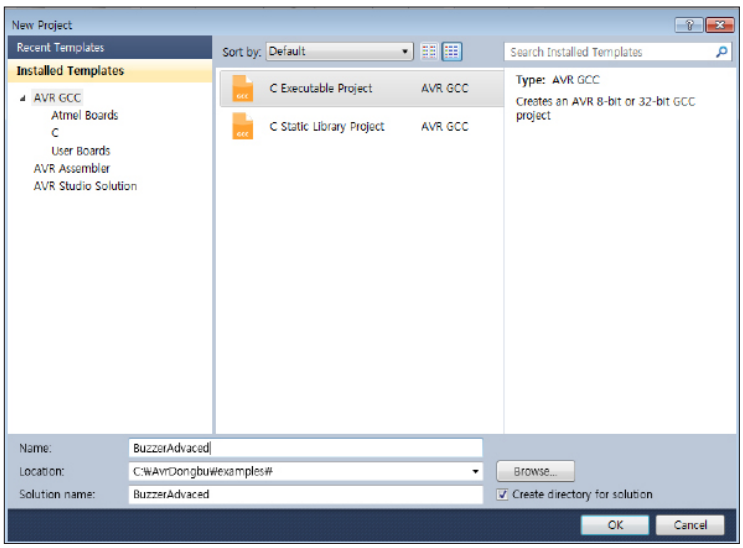
01 실행

AVR Studio 아이콘을 클릭하여 실행합니다.



02 New Project

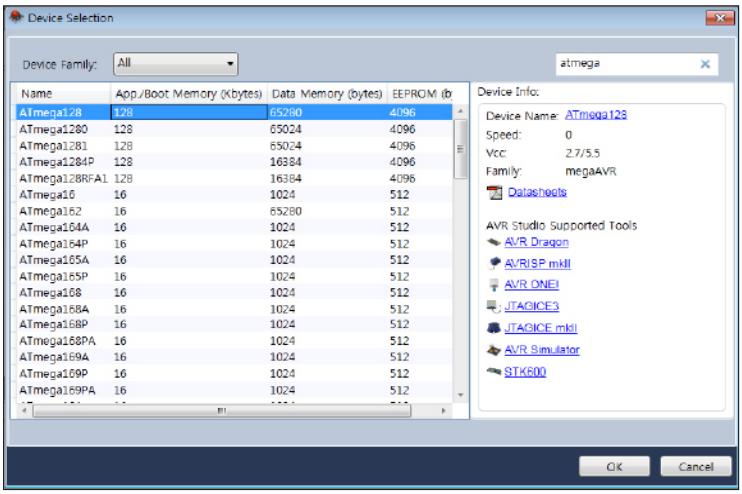
New Project 를 클릭합니다.



03 GCC 선택

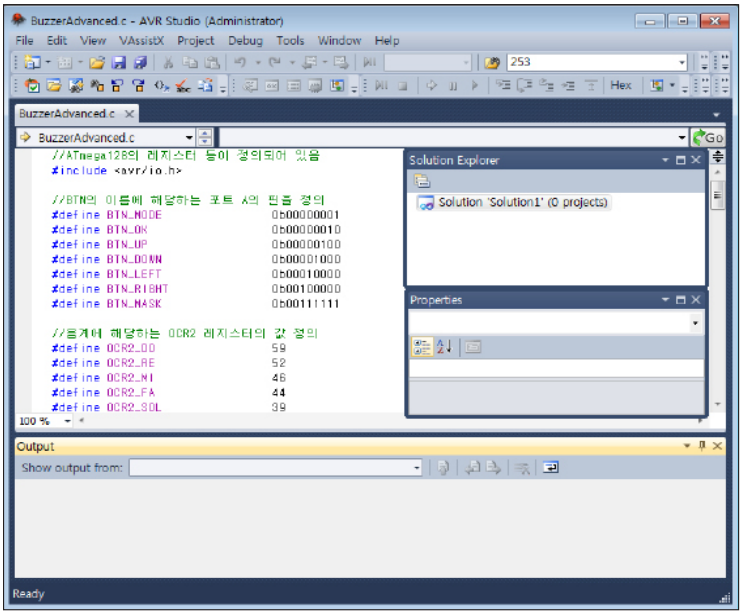
C Executable Project 를 선택합니다.

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;
}
```

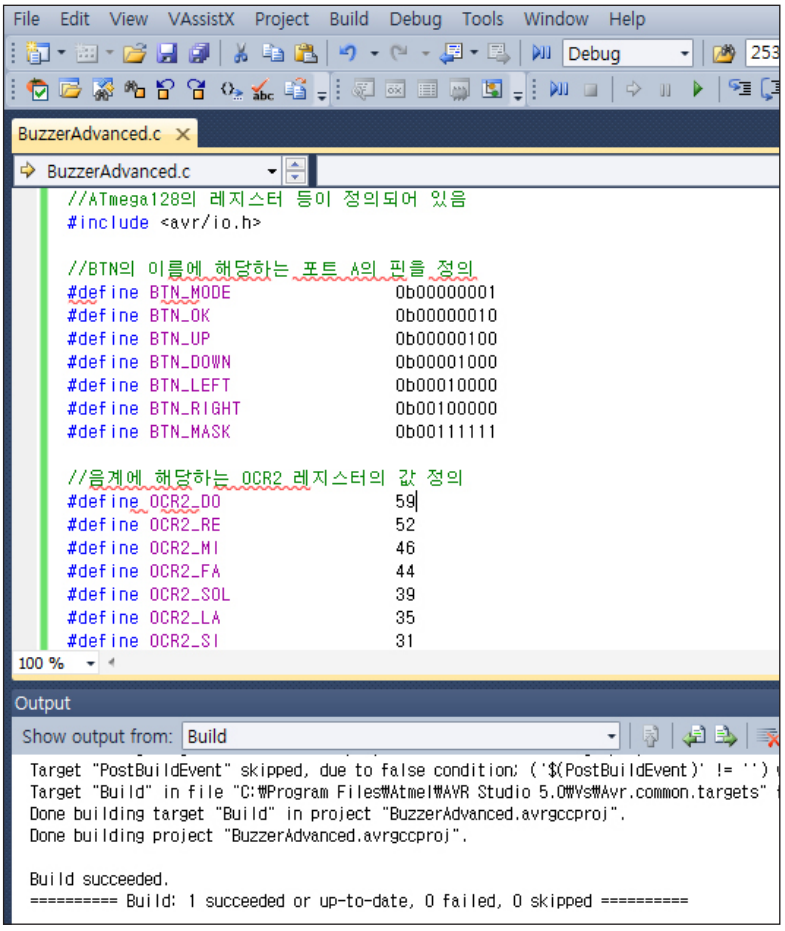
아무 버튼도 눌려있지 않다면, 타이머/카운터2 레지스터를 모두 0으로 만들어서 중지시킵니다. 이렇게 하면 버저의 울림이 멈추게 됩니다.

```
ucButtonOld = ucButton;
```

while 루프가 끝날 때마다 현재 루프에서 읽은 ucButton의 값을 ucButtonOld에 대입해, 다음 루프에서 비교하는 데에 사용합니다.

실행

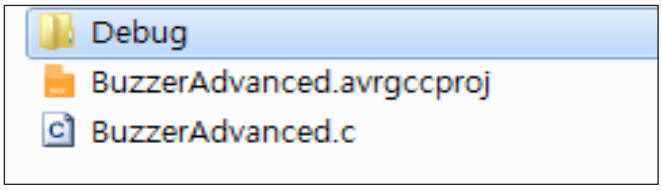
펌웨어를 다운로드하면, 버튼을 누를 때 해당하는 버저 음이 재생됩니다.



01 Build

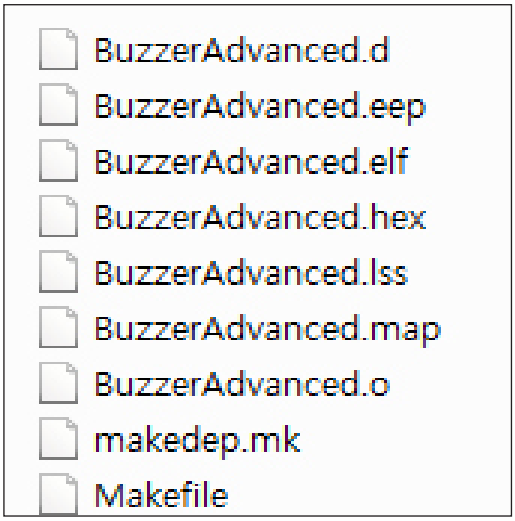
Build 메뉴를 선택하여 Build Solution을 클릭하여 Build 시킵니다.

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



02 Debug 폴더

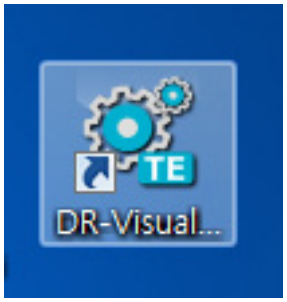
Build 후에 소스가 있는 폴더에 들어가면 Debug 폴더가 생성 되어있습니다. 클릭하여 폴더 안의 내용을 봅니다.



03 hex 확인

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

DR-Visual Logic 에서 펌웨어 복구를 실시합니다.



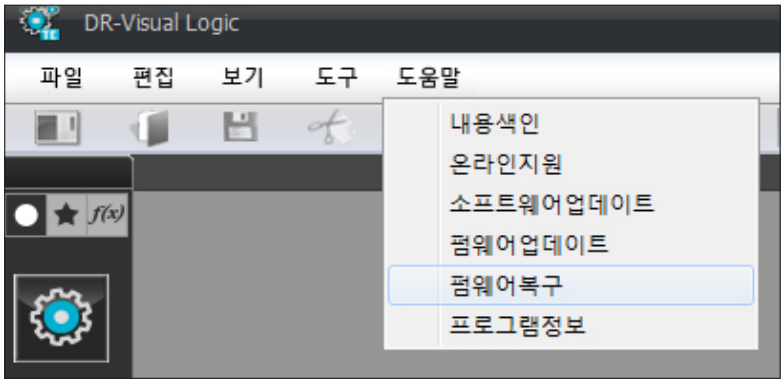
04 Visual Logic

DR-Visual Logic 을 실행시킵니다.



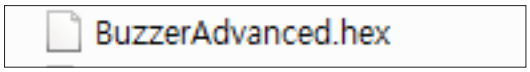
05 연결하기

COM 번호를 설정하고 연결하기를 클릭합니다.



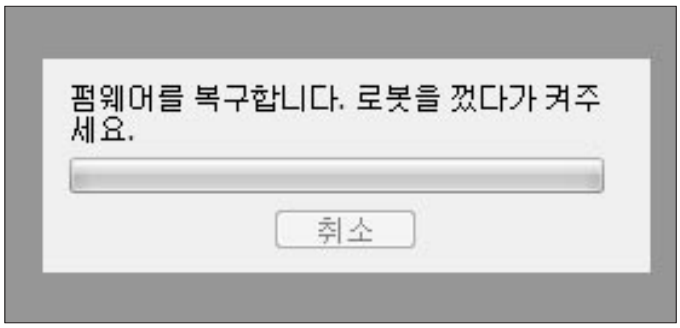
06 펌웨어 복구

도움말 메뉴에서 펌웨어 복구를 클릭합니다.



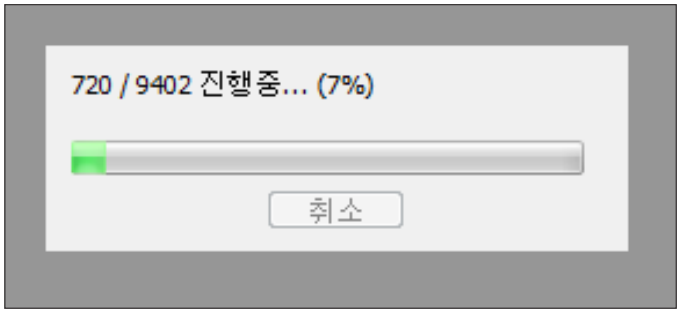
07 불러오기

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 도 레미파솔라 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.

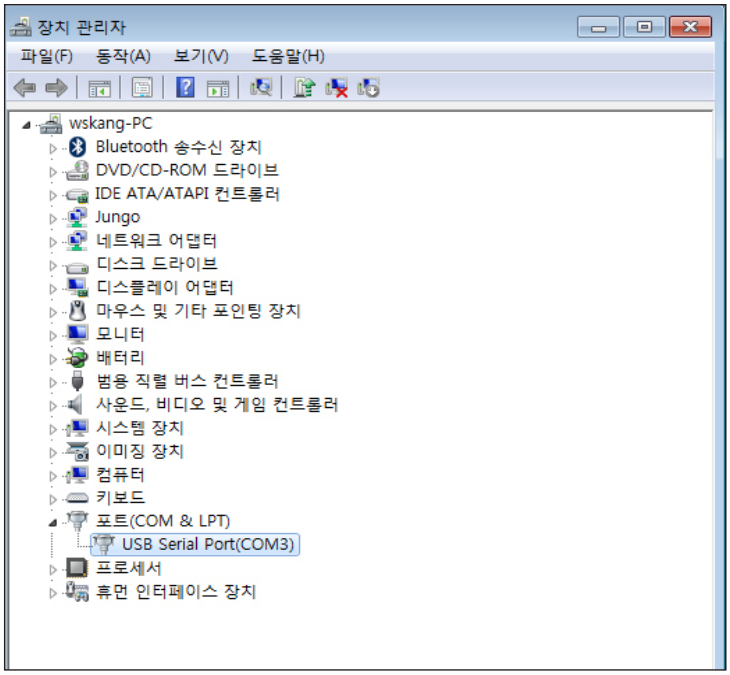
EEPROM을 읽고 쓰는 데에 관련된 레지스터로 EEARH, EEARL, EEDR, EECR 등의 레지스터가 있습니다. 그러나 사용자는 이 세세한 레지스터 설정에 대해서 알 필요가 없습니다. AVR Studio에서는 사용자가 eeprom 데이터를 쉽게 읽고 쓸 수 있도록 avr/eeprom.h 헤더 파일로 관련 함수를 이미 제공하기 때문입니다.

하드웨어 준비하기



01 연결

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



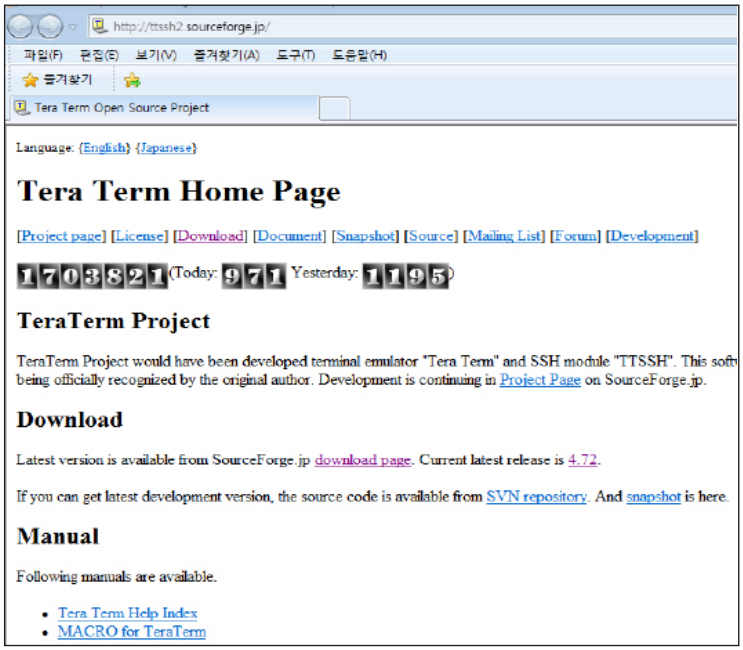
02 COM port 연결

내컴퓨터 〉 속성 〉 장치관리자에서 COM 연결 확인합니다.



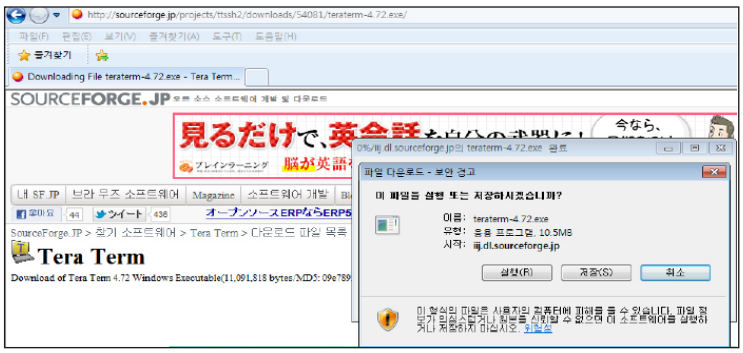
01 주소 검색

인터넷 포털 사이트에서 “tera term”을 검색합니다.



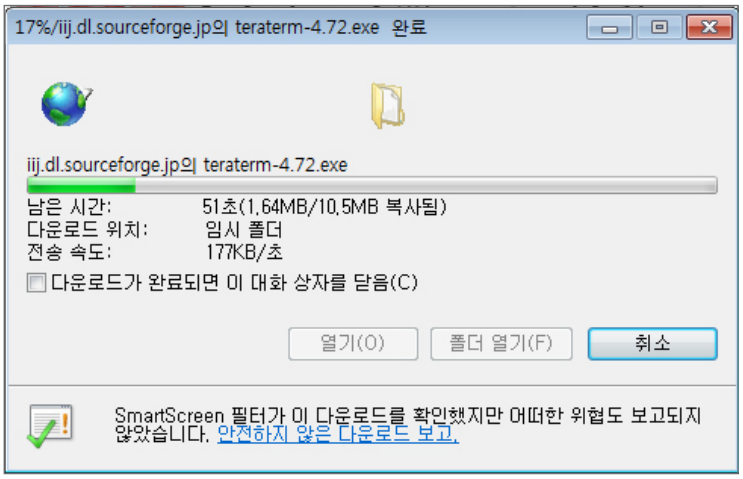
02 홈페이지 접속

Tera Term 홈페이지에 접속하여 “Download page”를 클릭합니다.



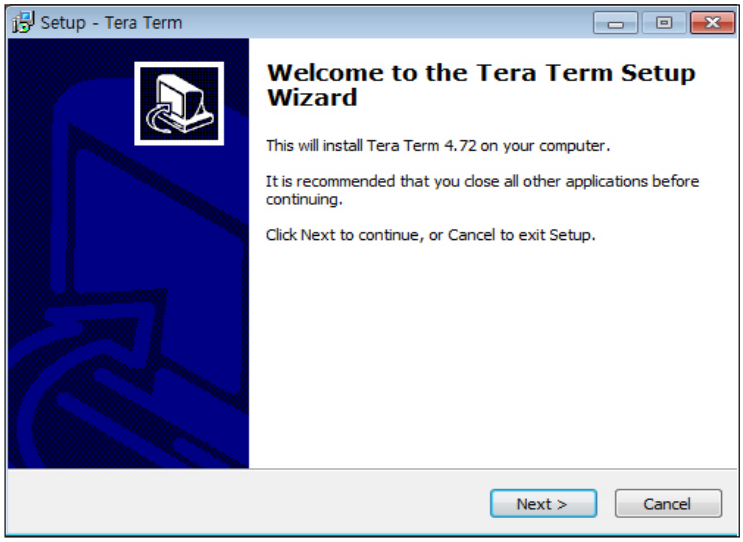
03 파일 다운로드

“저장”을 눌러 파일을 다운로드 받습니다.



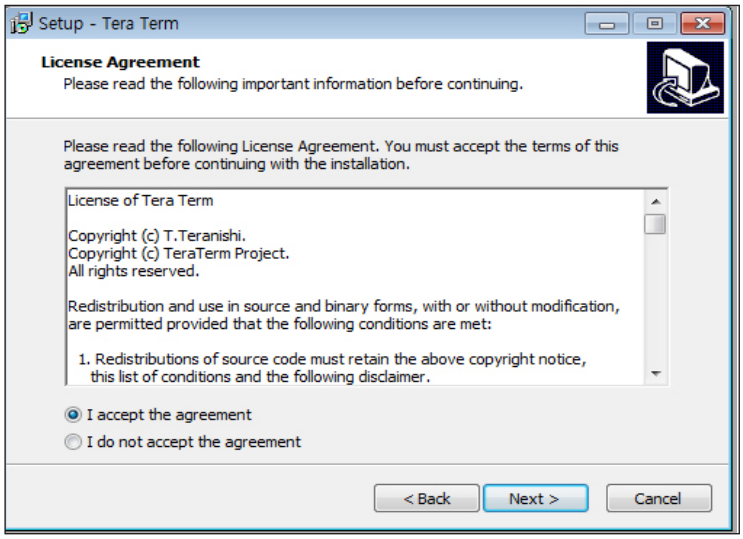
04 파일 다운로드 시작

설치 파일이 다운로드됩니다.



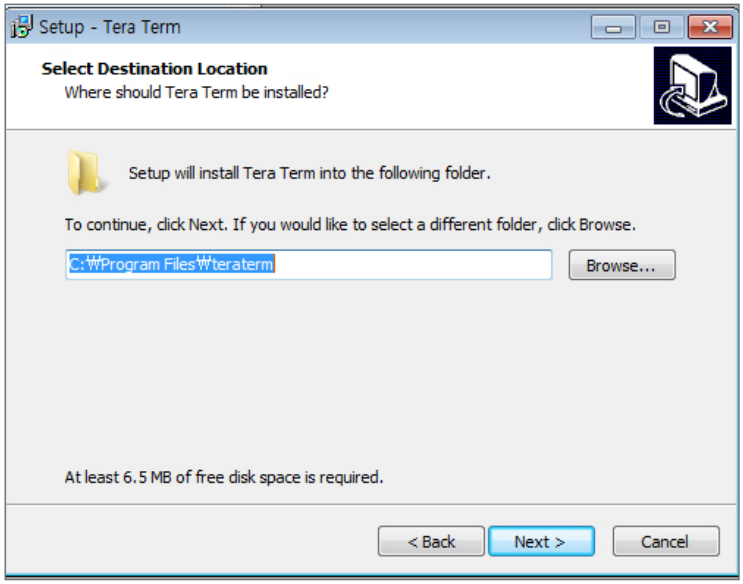
05 설치 마법사 실행

"Next"를 누르면 설치 마법사가 실행됩니다.



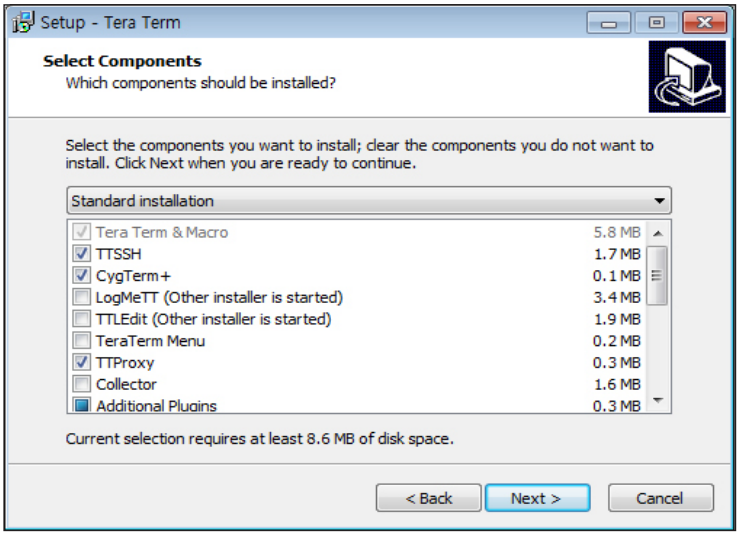
06 약관 동의

"I accept the agreement"를 체크하고 "Next"를 클릭합니다.



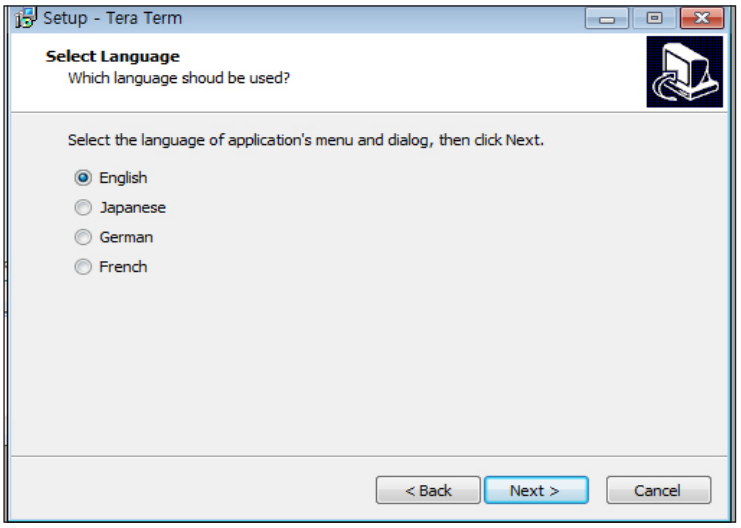
07 설치 폴더 확인

프로그램을 설치할 폴더를 확인하고 "Next"를 클릭합니다.



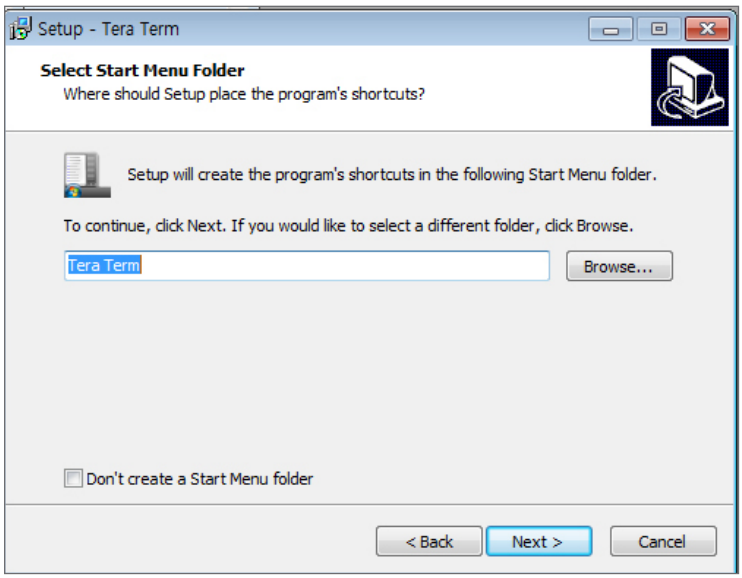
08 구성요소 선택

"Next"를 클릭합니다.



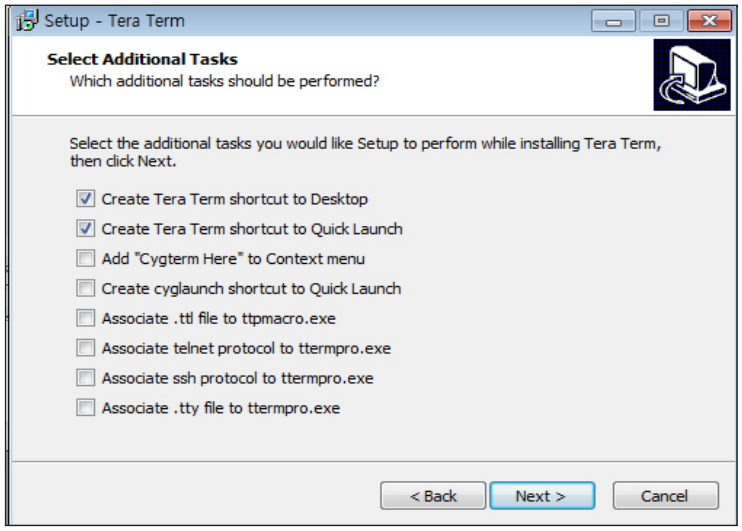
09 언어 선택

언어 선택 후 "Next"를 클릭합니다.



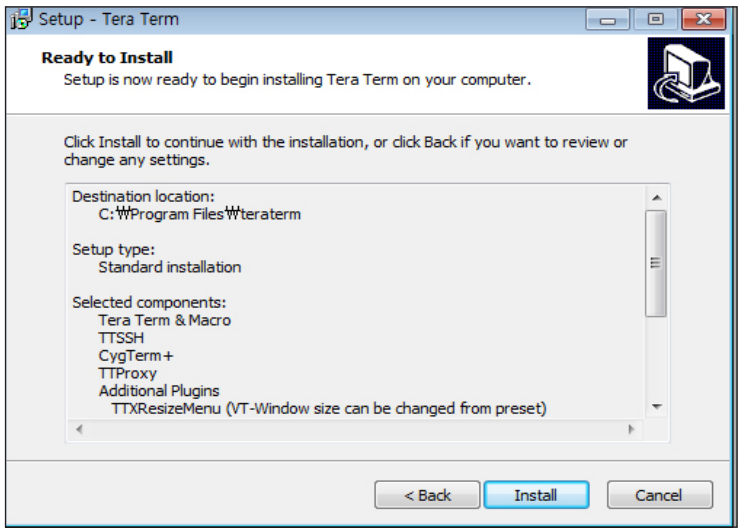
10 시작 메뉴 폴더 선택

시작 메뉴 폴더 선택 후 “Next”를 클릭합니다.



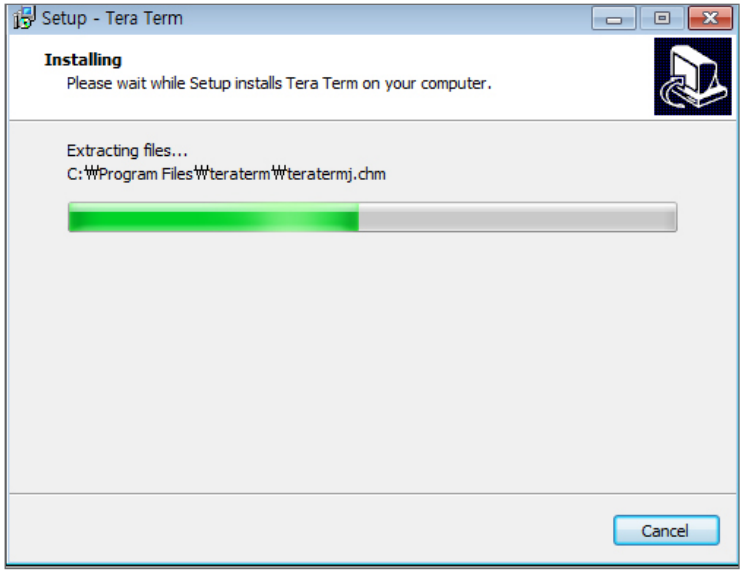
11 추가 기능 선택

원하는 추가 기능을 선택 후 “Next”를 클릭합니다.



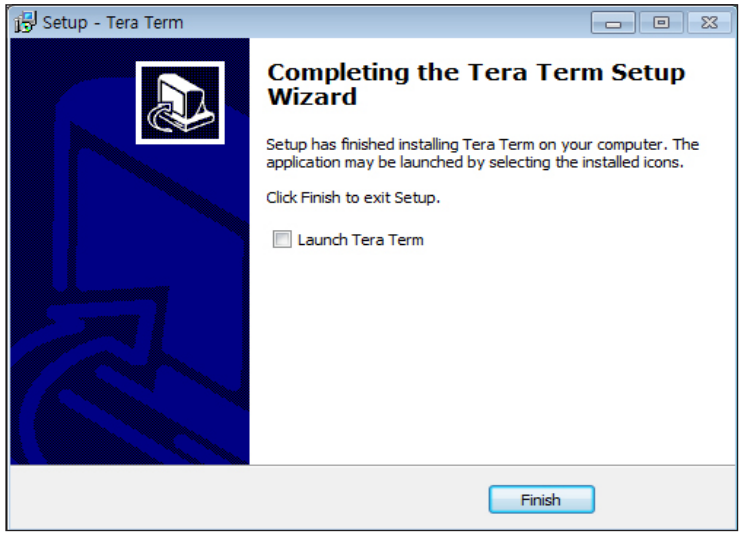
12 설치 시작

“Install”을 클릭하면 설치가 시작됩니다.



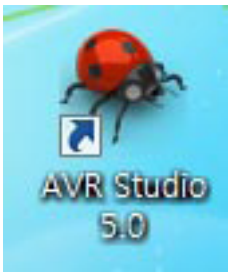
13 설치 확인

프로그램이 설치됩니다



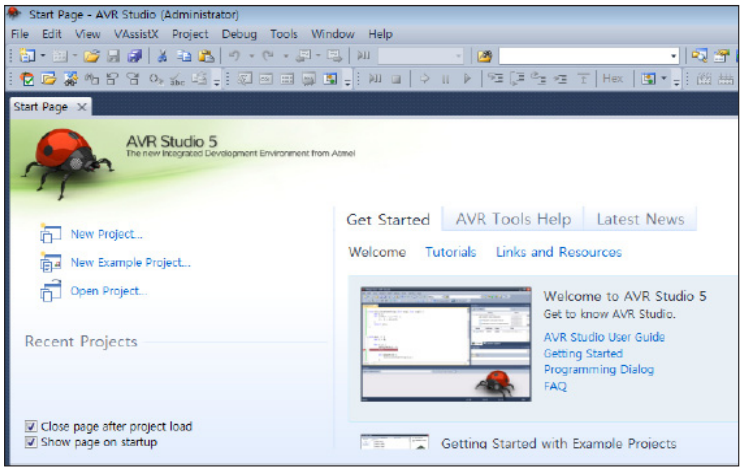
14 설치 완료

"Finish"를 클릭하면 설치가 완료됩니다.



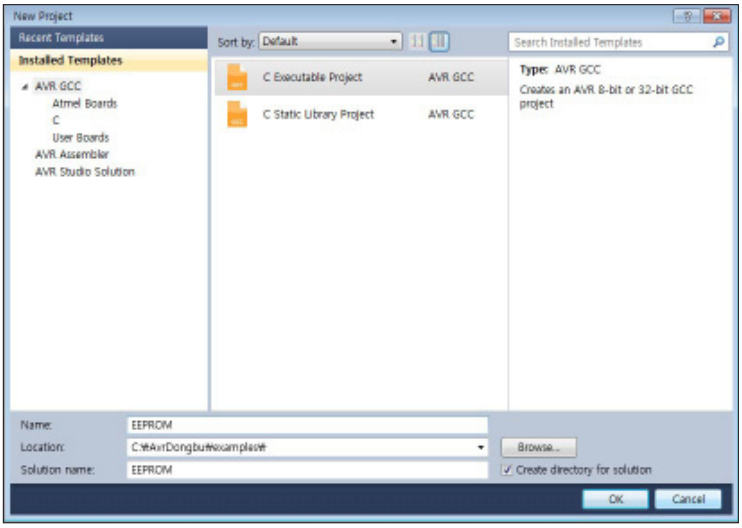
01 실행

AVR Studio 아이콘을 클릭하여 실행합니다.



02 New Project

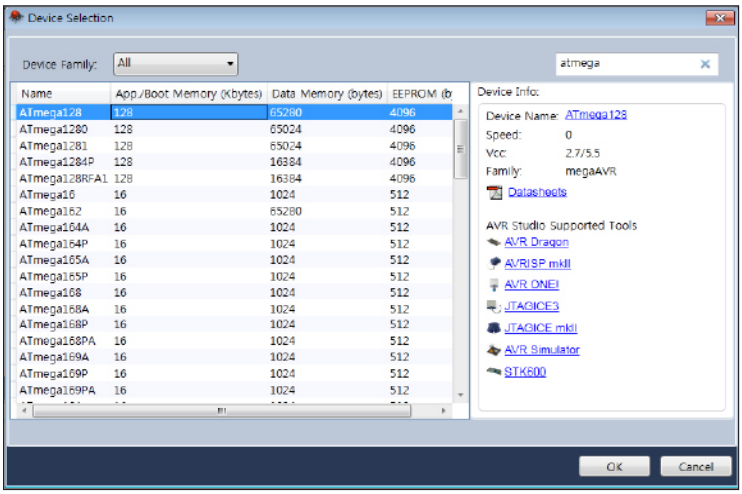
New Project 를 클릭합니다.



03 GCC 선택

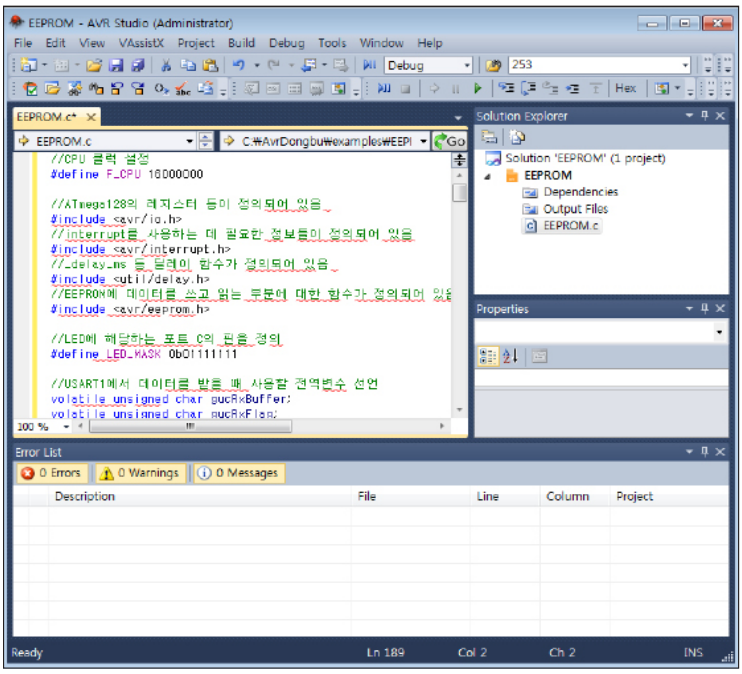
C Executable Project 를 선택합니다.

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;
}
```

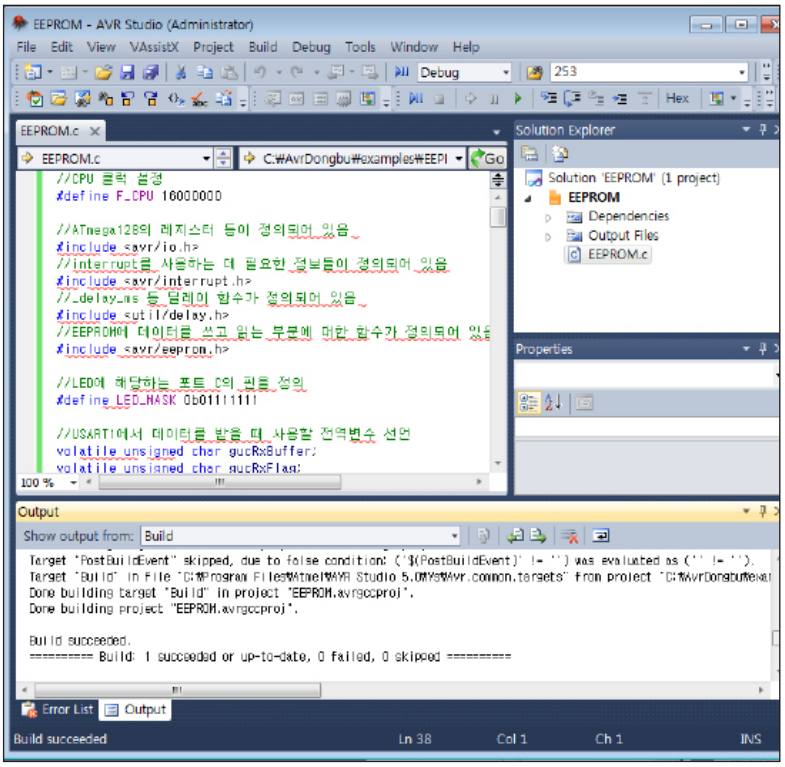
EEPROM으로부터 읽어온 데이터를 검사하는 부분입니다. 프로그램을 퓨징하고 처음 켠 상황이거나 EEPROM에 다른 프로그램이나 노이즈에 의해서 의도치 않은 값이 들어간 경우에는 ucLED에 예상치 못한 값이 저장될 것이고, 이것을 그대로 두면 프로그램이 오동작할 수 있습니다. 그래서 switch-case문을 사용하여 ucLED 값이 정상 범주(0b01000000~0b00000001) 안에 들면 값을 바꾸지 않고 넘어가고, 그 안에 들지 못하면 ucLED를 0b00000001로 초기화 하고 eeprom_write_byte 함수를 사용해 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

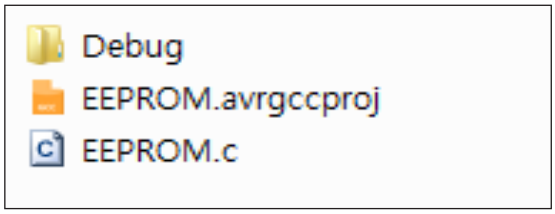
펌웨어를 실행하고 Tera Term을 이용해 Z,X키를 입력하면 켜진 LED가 이동하면서 터미널 창에 메시지가 출력됩니다. 이때 전원을 껐다가 켜도 마지막 위치에서 다시 시작합니다. ㅏ



01 Build

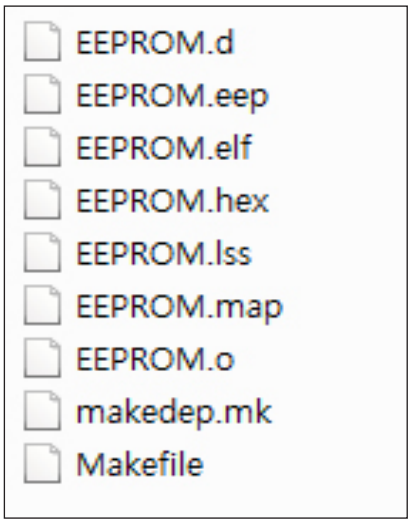
Build 메뉴를 선택하여 Build Solution 을 클릭하여 Build 시킵니다.

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



02 Debug 폴더

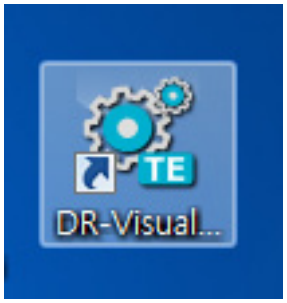
Build 후에 소스가 있는 폴더에 들어가면 Debug 폴더가 생성 되어있습니다. 클릭하여 폴더 안의 내용을 봅니다.



03 hex 확인

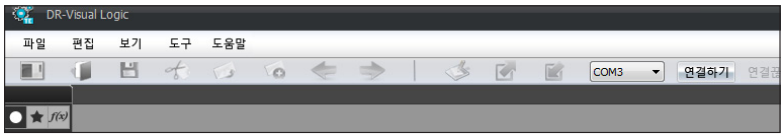
Debug 폴더 안의 hex 파일 확인합니다. Hex 파일이 제어기에 포팅시키는 파일입니다.

DR-Visual Logic 에서 펌웨어 복구를 실시합니다.



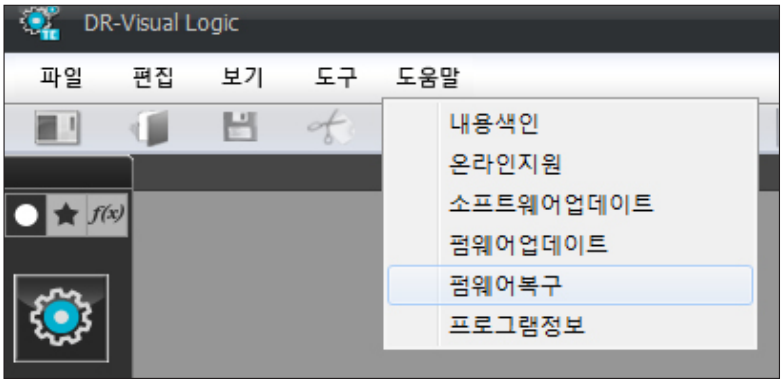
04 Visual Logic

DR-Visual Logic 을 실행시킵니다.



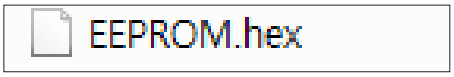
05 연결하기

COM 번호를 설정하고 연결하기를 클릭합니다.



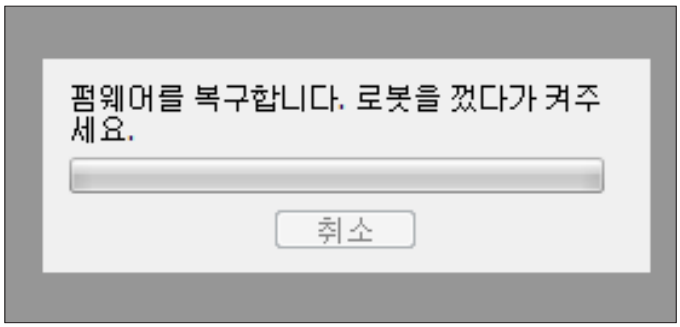
06 펌웨어 복구

도움말 메뉴에서 펌웨어 복구를 클릭합니다.



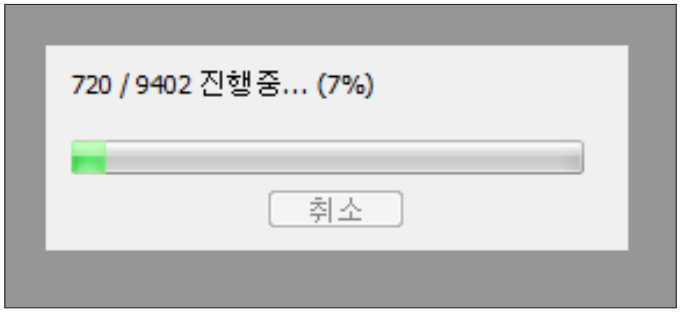
07 불러오기

Debug 폴더에서 Hex 파일을 불러옵니다



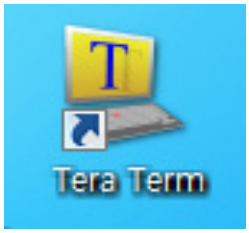
08 펌웨어복구

로봇을 껐다가 켜주세요 라는 문구가 나옵니다.
제어기의 전원 버튼을 한번 껐다가 다시 켜주세요.



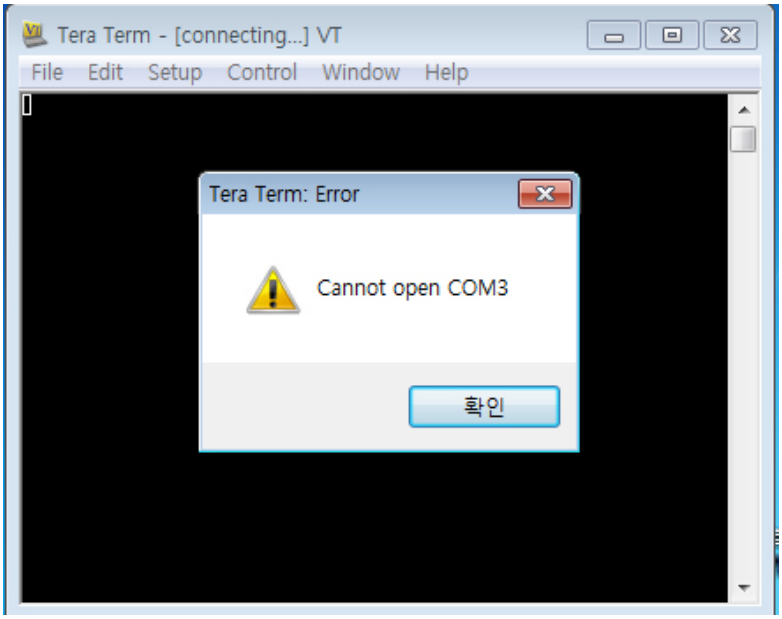
09 복구시작

펌웨어 복구가 시작됩니다.



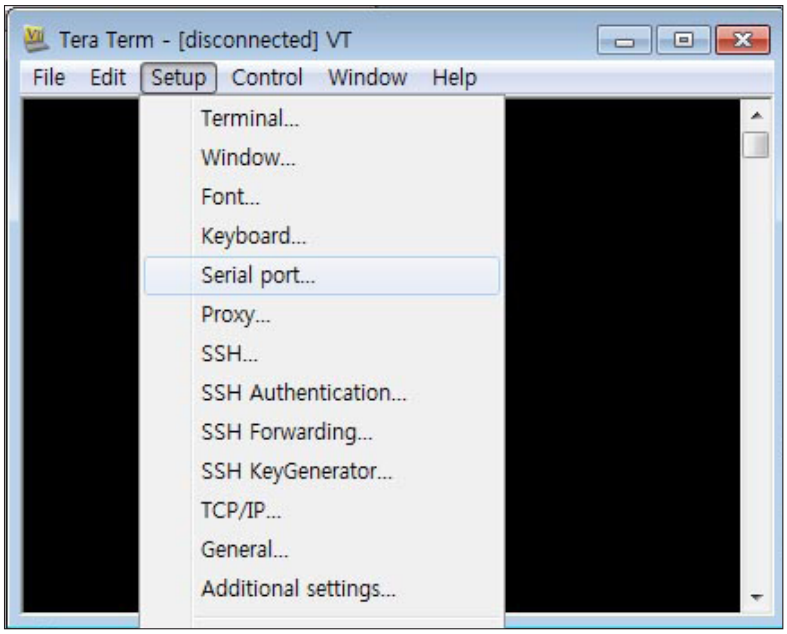
10 실행

Tera Term 아이콘을 클릭하여 프로그램을 실행시킵니다.



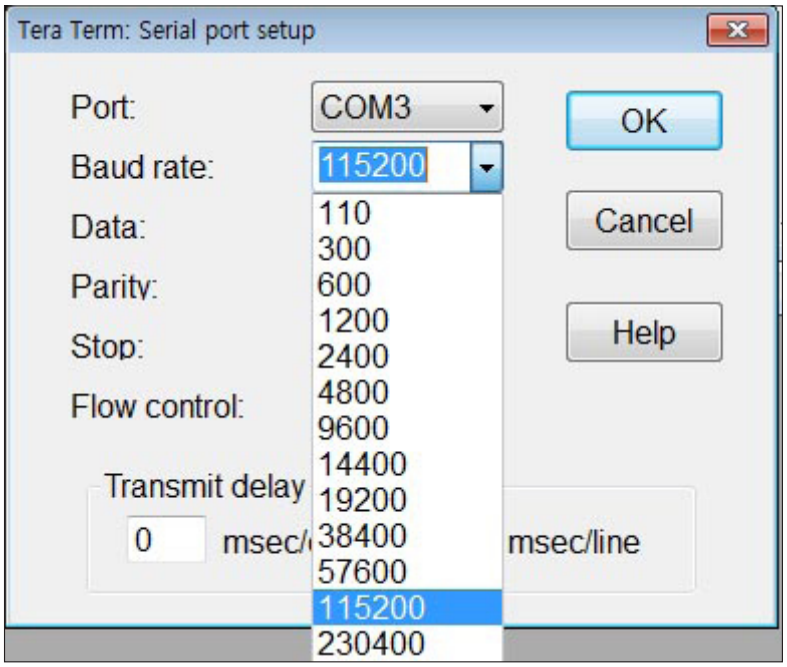
11 열기

실행하면 바로 연결되지 않습니다.
확인버튼을 클릭합니다.



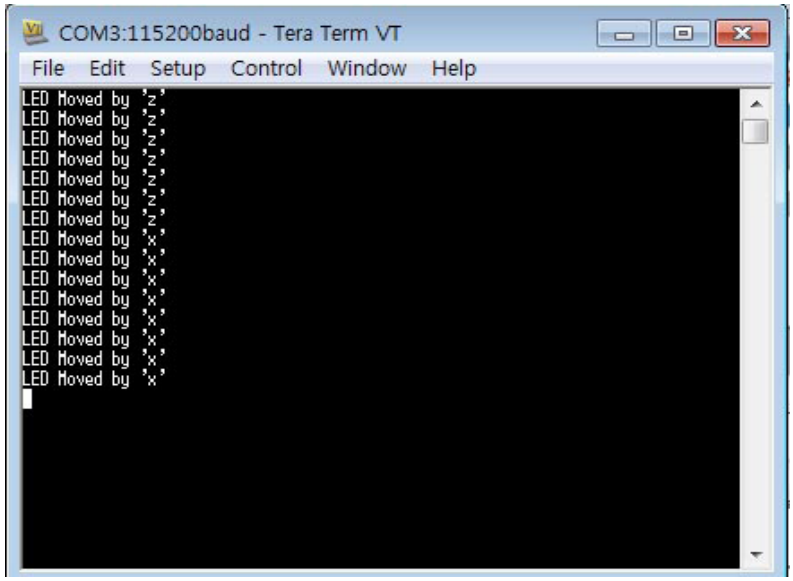
12 Setup

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.

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	

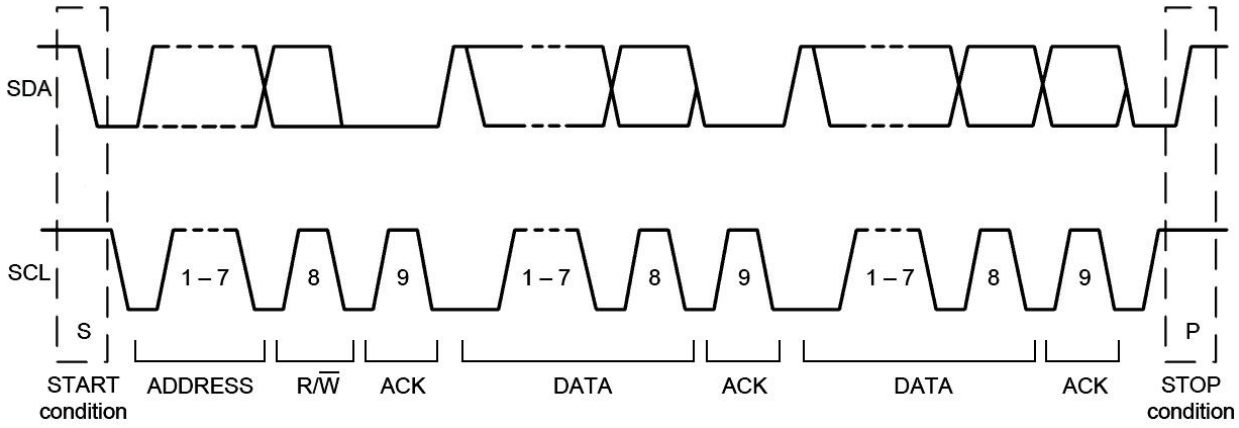
한 바이트 쓰기

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.

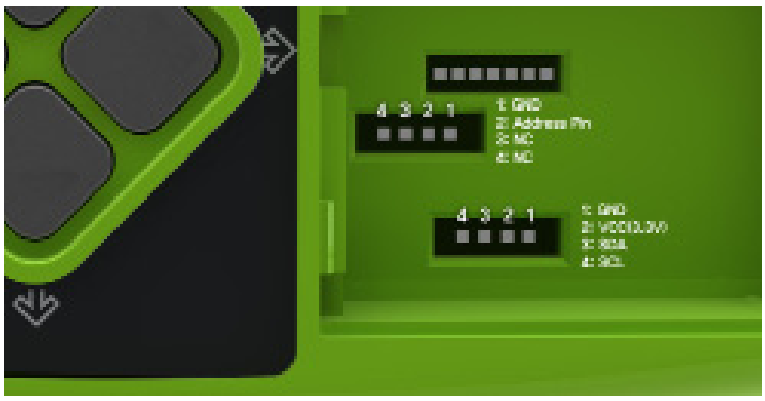
libaccgyro.a에서는 자잘한 시작 신호, 정지 신호, 응답 신호 등을 생각할 필요 없이, I2C 통신을 통해서 슬레이브에 데이터를 읽고 쓸 수 있는 함수들을 제공합니다.

```
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.

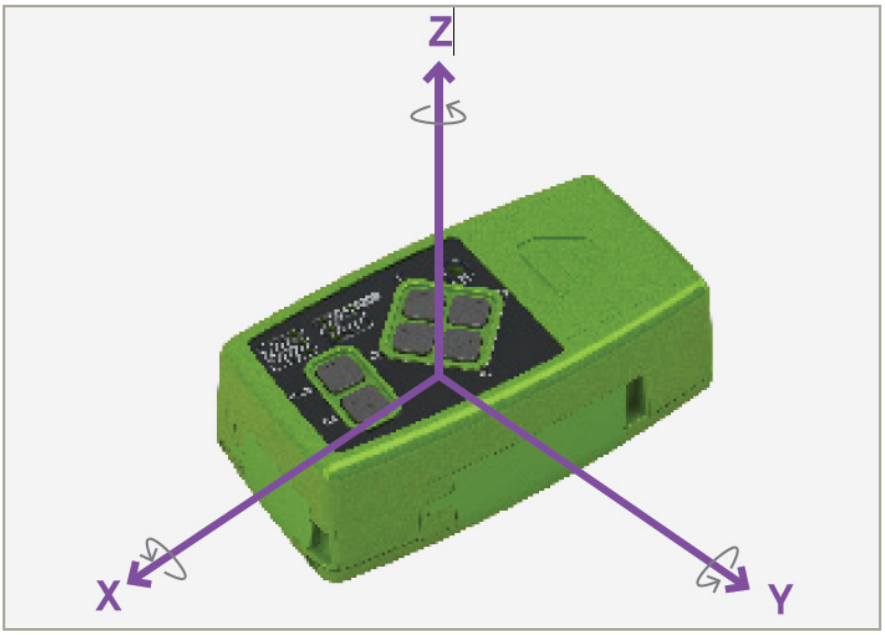
I2C_ucReadReg 함수는 I2C 통신으로 ucSlaveAddr이라는 슬레이브 주소의 ucRegAddr라는 레지스터 주소에서 데이터 1바이트를 읽어와 ucData라는 주소에 저장하는 함수입니다. I2C_ucReadNReg 함수도 마찬가지이지만 ucData라는 주소부터 ucCnt개만큼 읽어온다는 점이 다릅니다. I2C_ucWriteReg 함수는 I2C 통신으로 ucSlaveAddr이라는 슬레이브 주소의 ucRegAddr라는 레지스터 주소에 ucData라는 데이터를 쓰는 함수입니다. I2C_ucWriteNReg 함수도 마찬가지이지만 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.

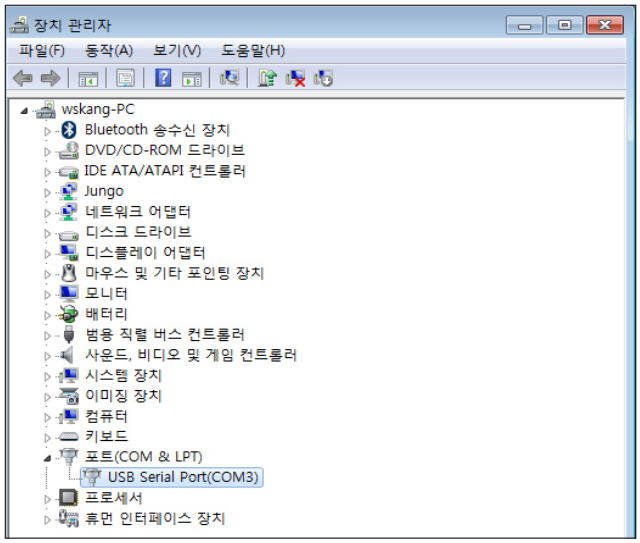
준비

하드웨어 준비하기



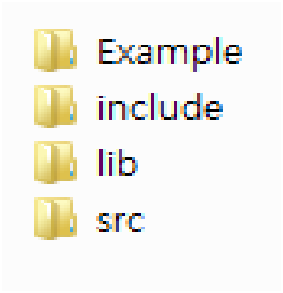
01 연결

DRC 제어기 아답터로 전원을 연결하고, USB to Serial Cable 로 PC 와 DRC 를 연결합니다.
가속도/자이로 모듈을 장착합니다.



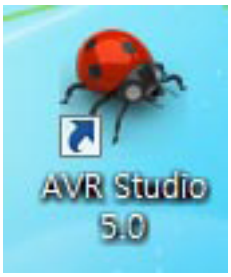
02 COM port 연결

내컴퓨터 〉 속성 〉 장치관리자에서 COM 연결 확인합니다.



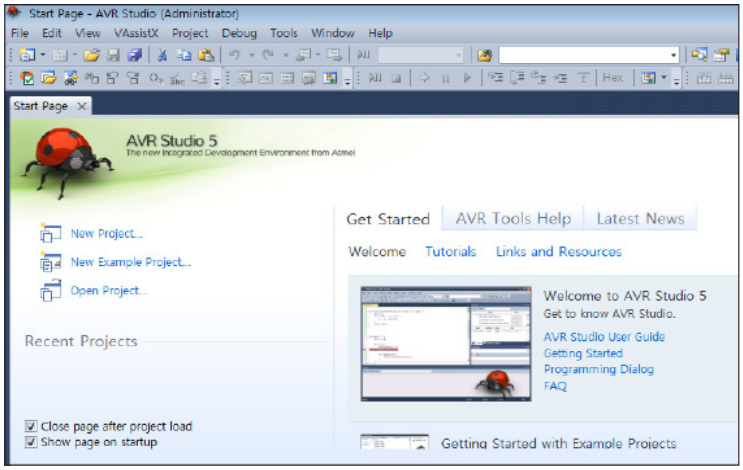
00 라이브러리

가속도/자이로 모듈을 제어하기 위해서는 기본 라이브러리 파일이 필요합니다. 라이브러리가 포함된 폴더 구조입니다.



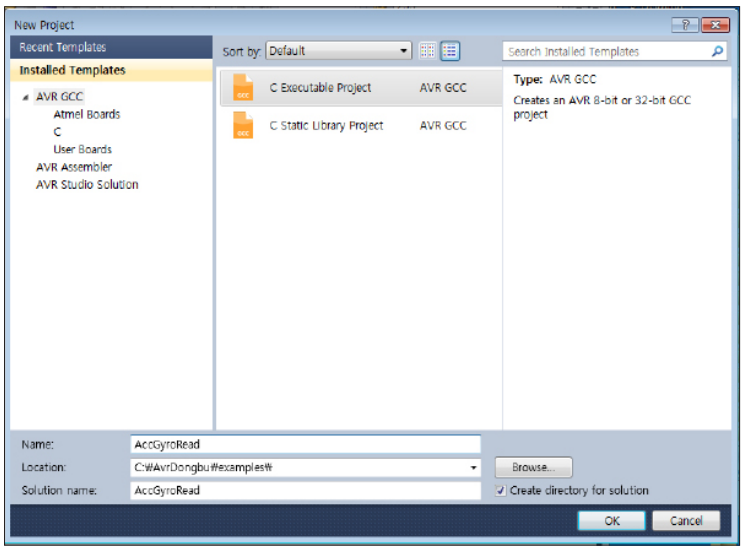
01 실행

AVR Studio 아이콘을 클릭하여 실행합니다.



02 New Project

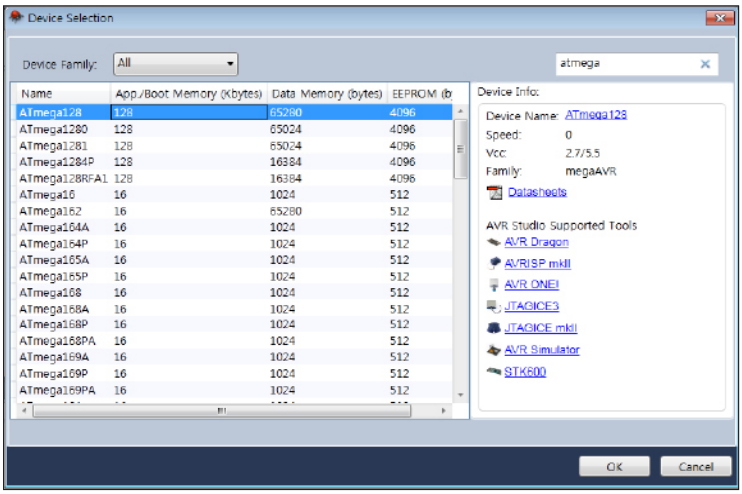
New Project 를 클릭합니다.



03 GCC 선택

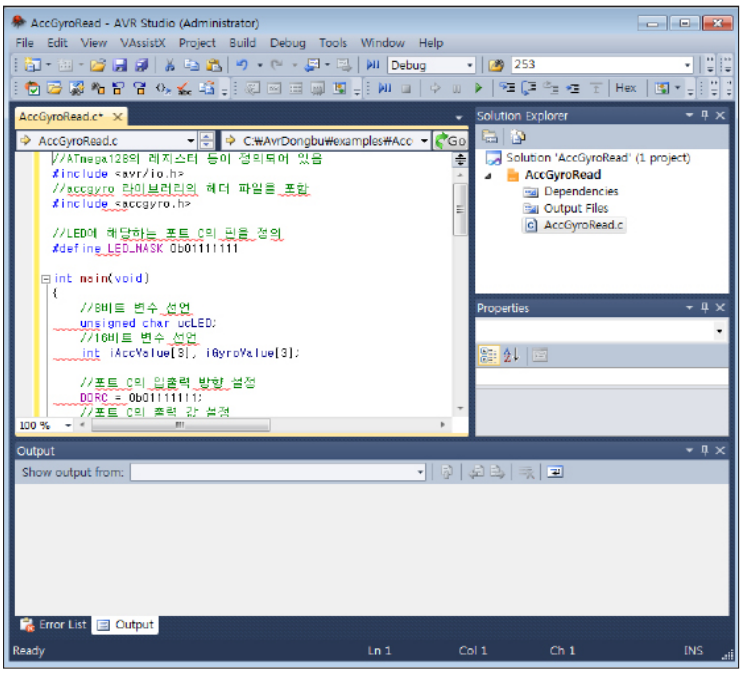
C Executable Project 를 선택합니다.

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



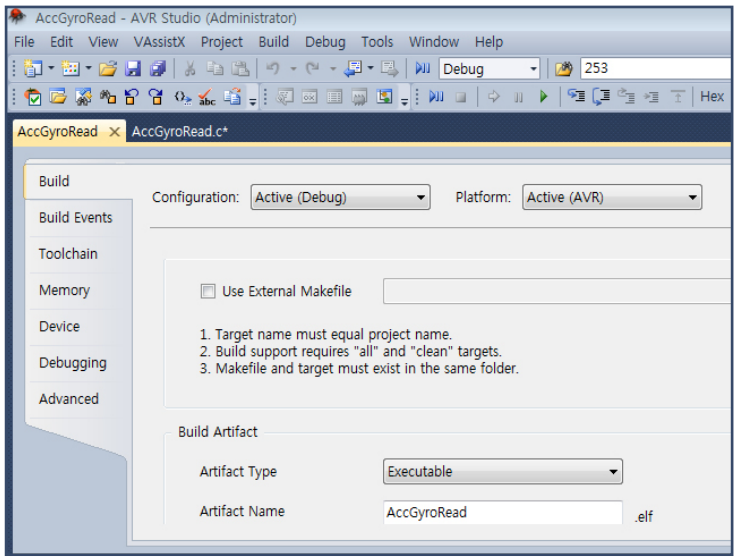
04 ATmega 선택

우측 상단에 atmega 를 검색하여,
리스트중에 ATmega128 을 선택합니다.



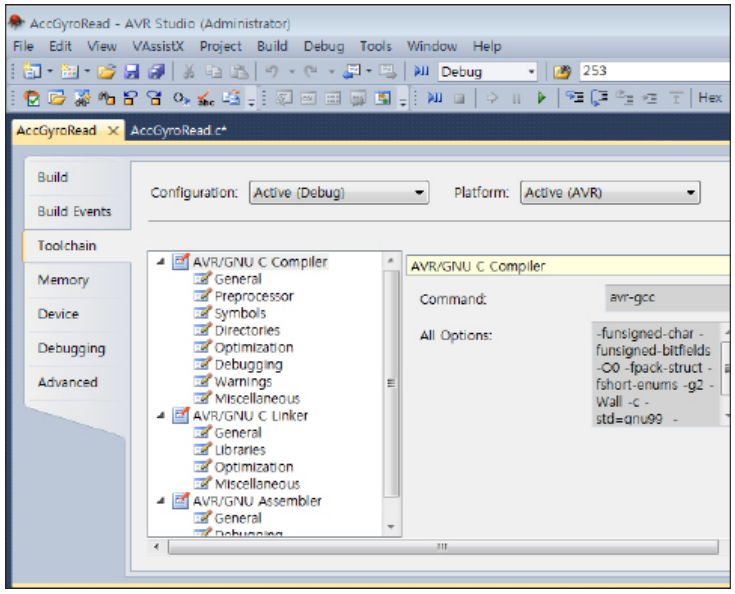
05 Editor

편집창에 코딩을 시작합니다.



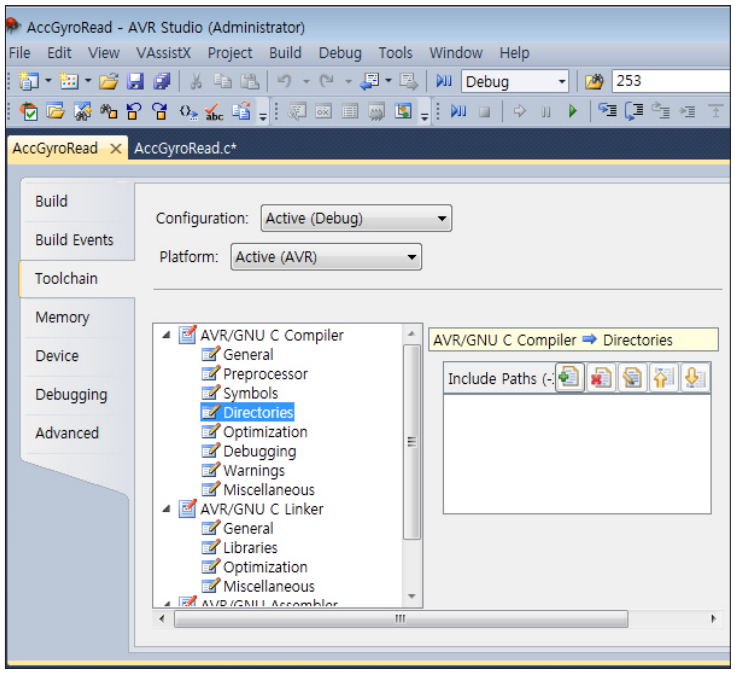
01 설정

AVR Studio 상단 메뉴중에 Project 를 클릭하면 AccGyroRead (프로젝트 이름별로 다름) Property 메뉴가 생성되어 있습니다. 클릭하면 좌측과 같이 설정화면이 나옵니다.



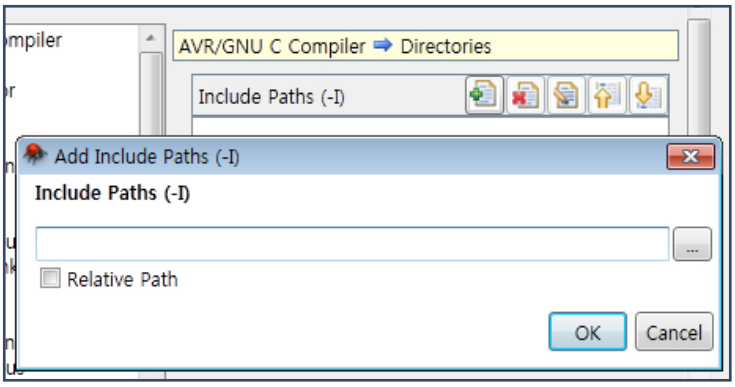
02 Toolchain

좌측 탭버튼중에 Toolchain 을 클릭하여, AVR/GNU C Compiler 을 선택합니다



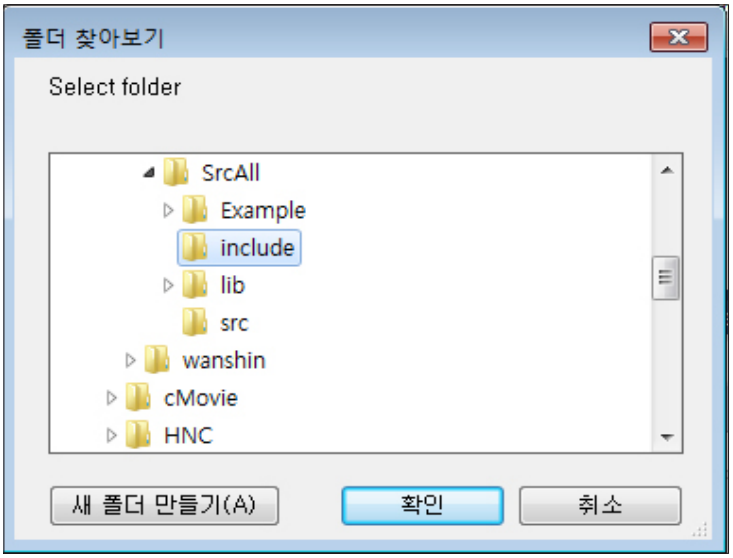
03 C Compiler

C Compiler 폴더중에 Directories 를 선택합니다.



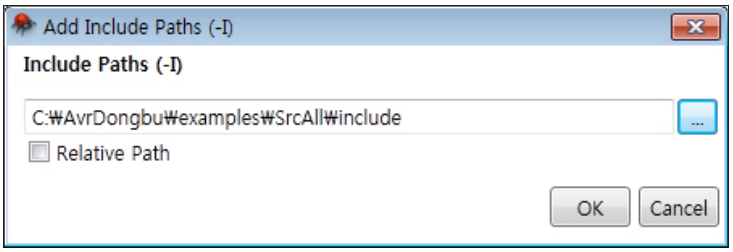
04 Path

Include Paths 오른쪽에 + 버튼을 클릭하면 팝업 입력창이 뜹니다.
Relative Path (상대경로) 체크를 해지하고(5.1에서는 안해도됨),
우측 폴더 검색 버튼을 클릭합니다.



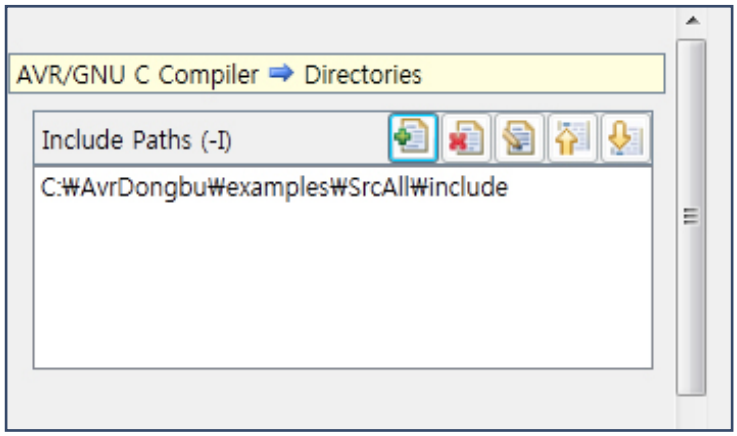
05 Include

제공 라이브러리 폴더중에 include 를 선택합니다.



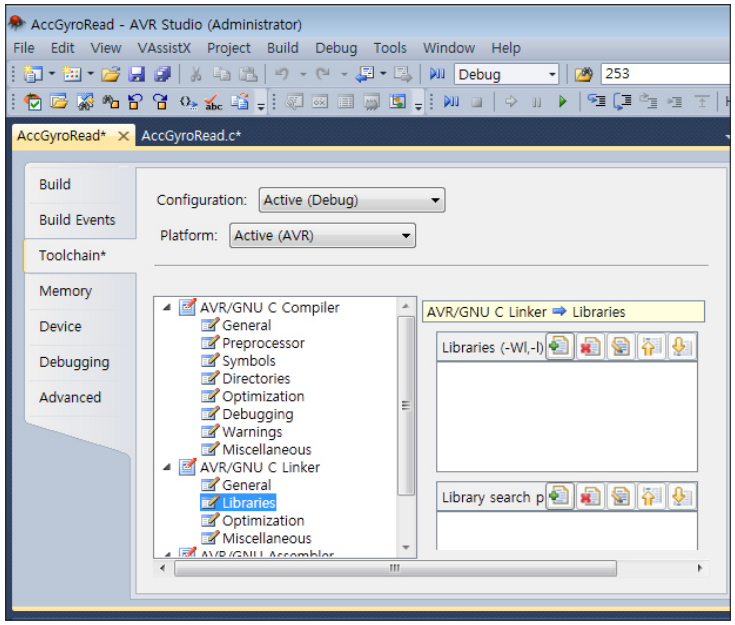
06 Include 완료

Include 폴더 위치가 지정됩니다. OK 버튼을 클릭합니다.



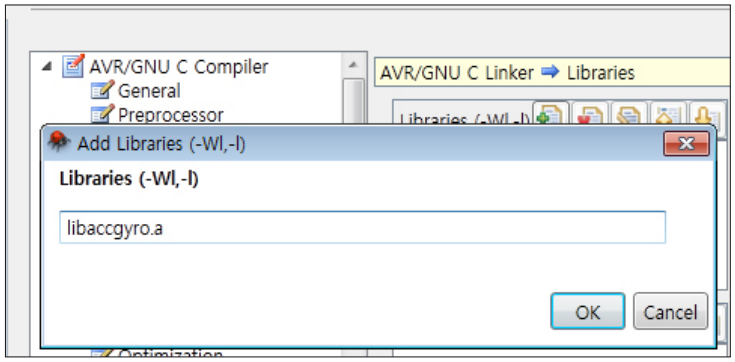
07 Path 완료

Include 경로지정이 완료되었습니다.



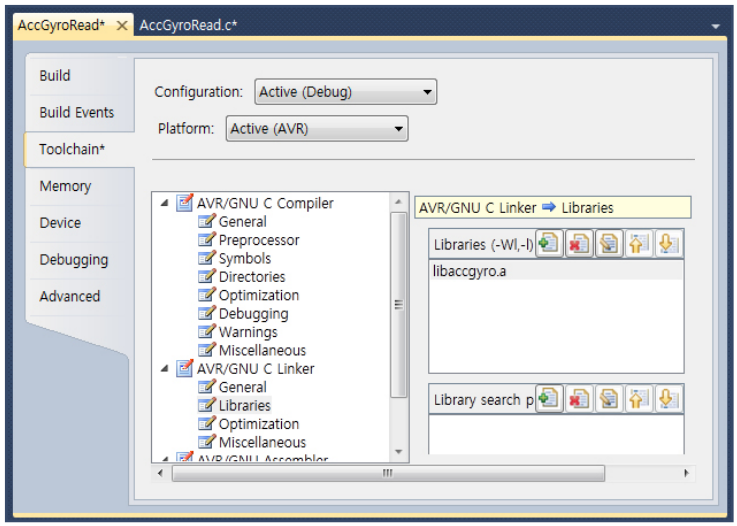
08 C Linker

C Linker 폴더중에 Libraries 를 선택합니다.



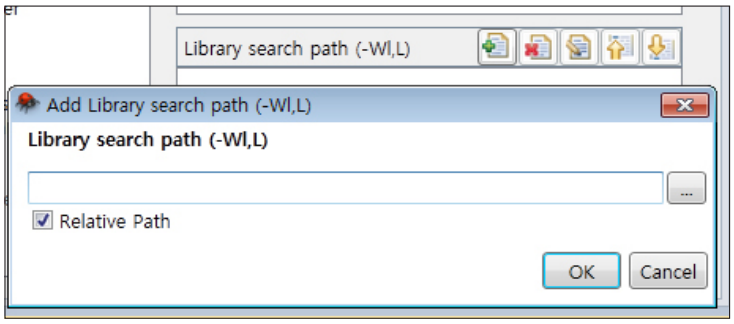
09 Libraries

Libraries 창 우측 + 를 클릭하면, Libraries 파일명 입력창이 나옵니다. libaccgyro.a 를 직접 입력합니다.
OK 버튼을 클릭합니다.



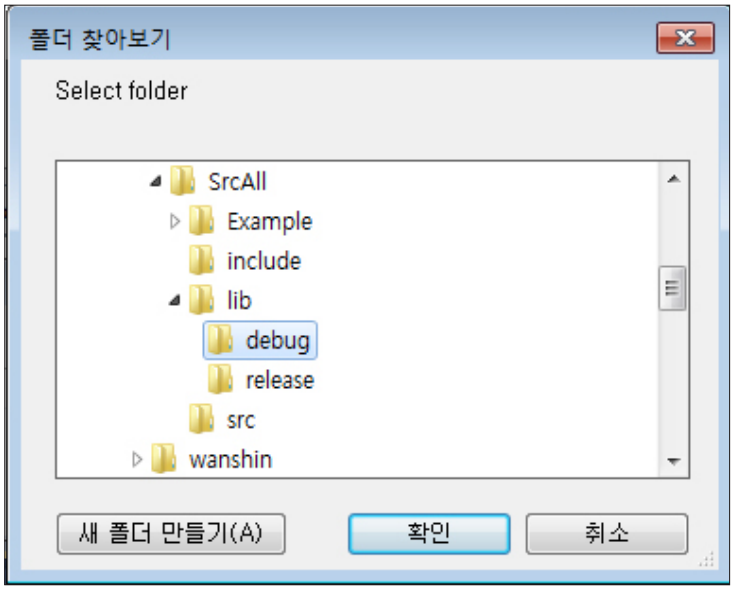
10 Search

가속도/자이로모듈 라이브러리 파일 libaccgyro.a 링크가 완료되었습니다.
다음으로 Library search path 를 설정합니다.
우측 + 버튼을 클릭합니다.



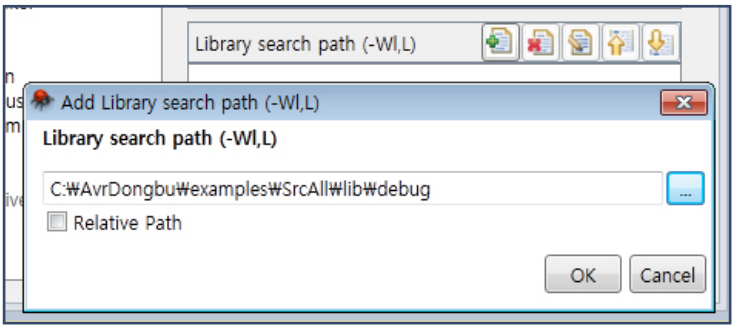
11 Path

Library search path 팝업창 우측에 폴더 검색 버튼을 클릭합니다.



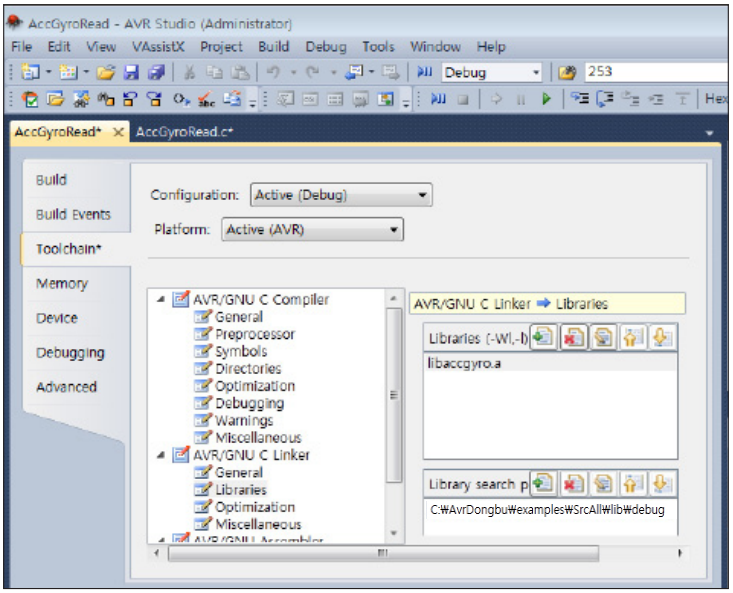
12 Debug

제공 라이브러리 폴더에서 lib 하위 폴더에 debug 를 선택합니다.



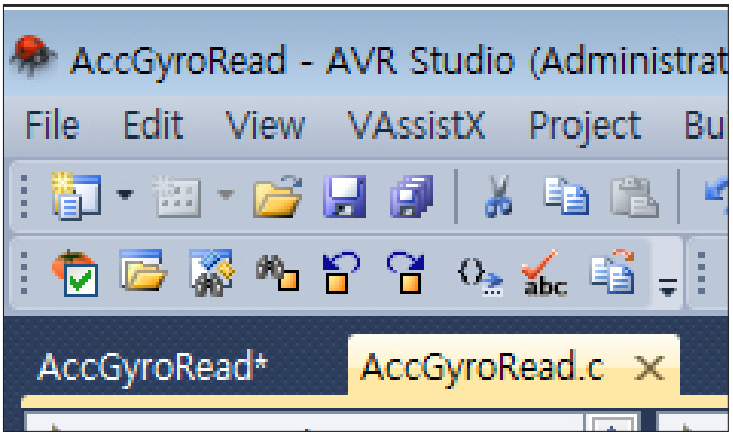
13 Path 완료

Library search path 경로지정이 완료되었습니다.
OK 버튼을 클릭합니다.



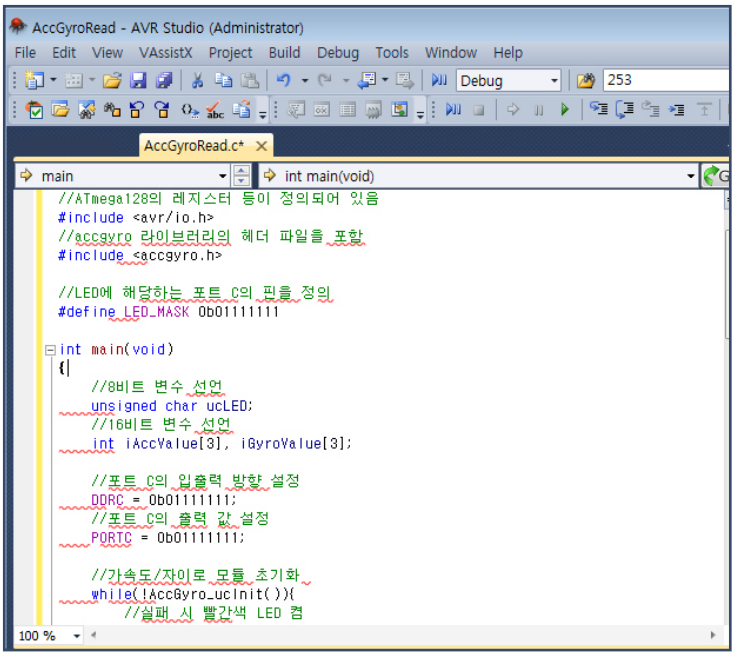
14 Link 완료

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.

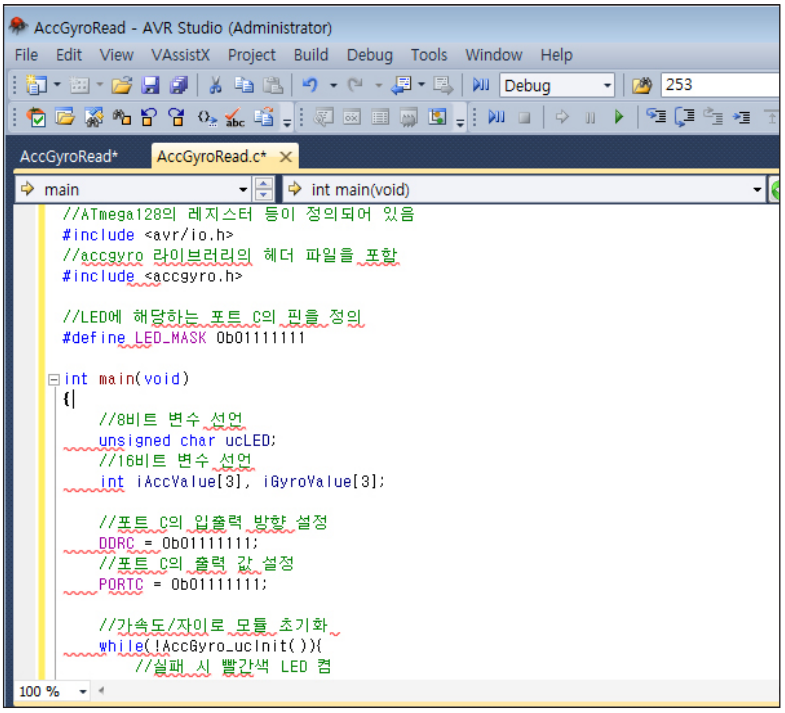
If문 안에서는 우선 ucLED 값이 모두 꺼지도록 초기화합니다. 그리고 iAccValue[2], 즉 Z축의 가속도 값이 180이하면 ucLED의 초록색 LED(Program, RX)가 켜지도록 ucLED의 해당 비트를 1로 만듭니다. Z축의 가속도 값이 180이하라는 것은, H/W 원리에서 언급했듯이 DRC가 약 45° 이상 기울어져 있다는 뜻입니다. 또, iGyroValue[2]가 1500보다 크거나 –1500보다 작으면 ucLED의 파란색 LED(EXEC, Spare)가 켜지도록 ucLED의 해당 비트를 1로 만듭니다. iGyroValue[2]의 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

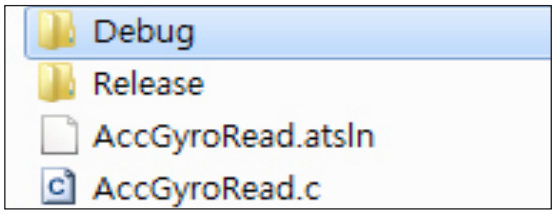
가속도/자이로 센서를 DRC에 장착한 후 펌웨어를 실행하면 가만히 수평으로 놓인 상태에서는 LED가 모두 꺼져 있으며, 기울이면 초록색 LED가, 회전 시키면 파란색 LED가 켜지는 것을 확인할 수 있습니다. 만약 가속도/자이로 센서를 장착하지 않거나 작동 중에 제거하면 빨간색 LED가 켜집니다.



01 Build

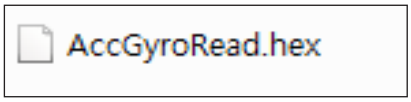
Build 메뉴를 선택하여 Build Solution을 클릭하여 Build 시킵니다.

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



02 Debug 폴더

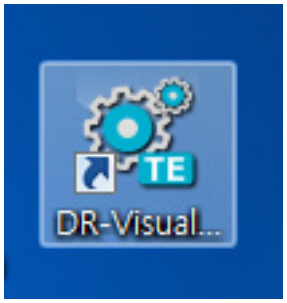
Build 후에 소스가 있는 폴더에 들어가면 Debug 폴더가 생성 되어있습니다. 클릭하여 폴더 안의 내용을 봅니다.



03 hex 확인

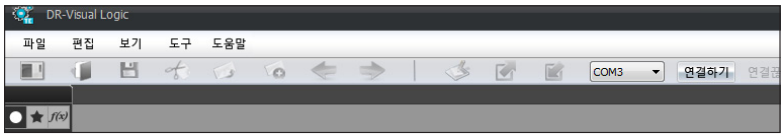
Debug 폴더 안의 hex 파일 확인합니다. Hex 파일이 제어기에 포팅시키는 파일입니다.

DR-Visual Logic 에서 펌웨어 복구를 실시합니다.



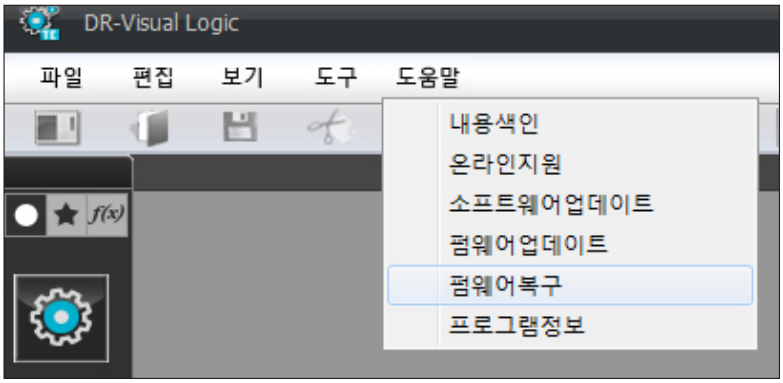
04 Visual Logic

DR-Visual Logic 을 실행시킵니다.



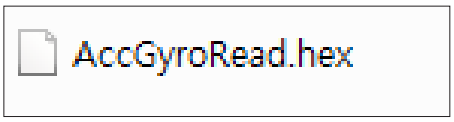
05 연결하기

COM 번호를 설정하고 연결하기를 클릭합니다.



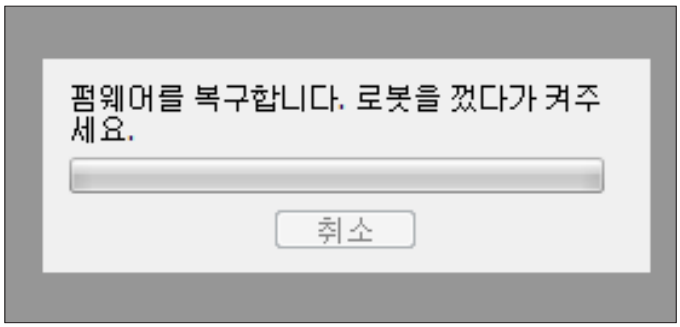
06 펌웨어 복구

도움말 메뉴에서 펌웨어복구를 클릭합니다.



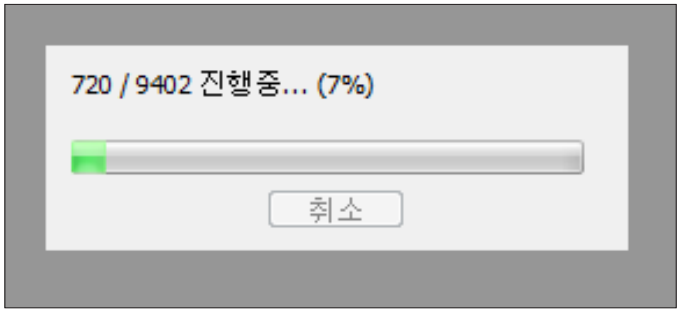
07 불러오기

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.

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

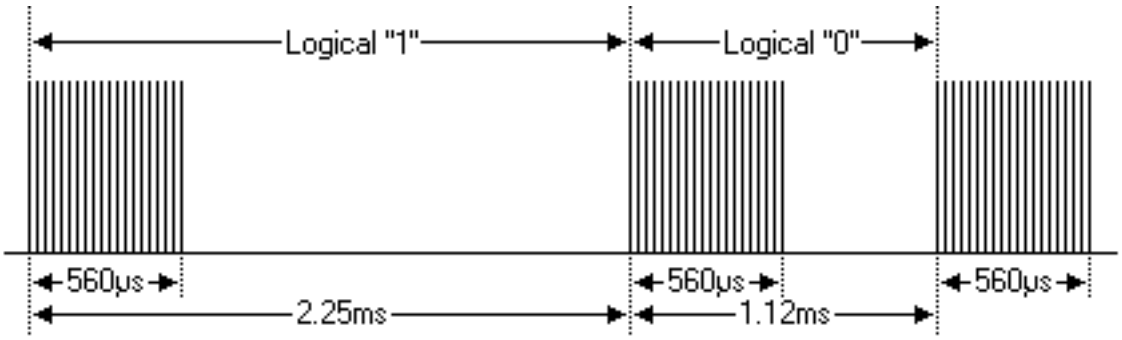
적외선 리모컨은 말 그대로 적외선을 사용해서 신호를 보내서 원격으로 다른 기기를 조종하는 장치입니다. 리모컨의 키를 누르면 리모컨 내부의 마이크로프로세서는 정해진 프로토콜에 맞게 HIGH와 LOW로 적외선 LED를 껐다가 켜서 데이터 코드를 앞의 기기에 쏩니다. 이 때 빛을 쏠 때는 단순히 HIGH이면 계속 적외선 LED를 켜고 LOW면 끄는 단순한 방식이 아니라, 38kHz의 반송파(Carrier Wave)로 변조되어 출력됩니다. 신호 내용이 HIGH일 때 38kHz의 구형파로 1과 0을 왔다갔다하는 신호를 쏜다는 뜻입니다. 리모컨 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.

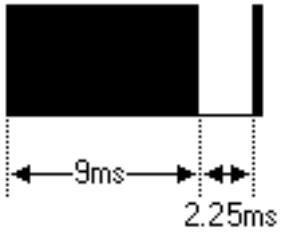
NEC 프로토콜(D6121)



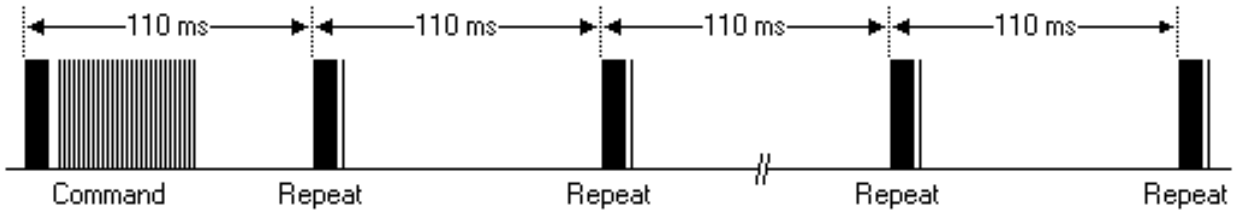
위 그림은 NEC 프로토콜의 전형적인 신호를 보여줍니다. 리모컨의 신호가 시작될 때는 9밀리초 동안의 HIGH 신호와 4.5밀리초 동안의 LOW 신호를 내보내는데, 이를 리더 코드(Leader Code)라고 합니다. 그 후에는 낮은 번호의 비트(LSB)부터 8비트의 Address와 Command를 보내는데, 각각 두 번씩 전송됩니다. 처음 보낼 때는 원래의 값을 보내고, 두 번째 보낼 때에는 모든 비트를 뒤집어서 보냅니다. 위 그림에서 보아도 처음 Address는 0b01011001인데 그 다음 ~Address는 0b10100110임을 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.



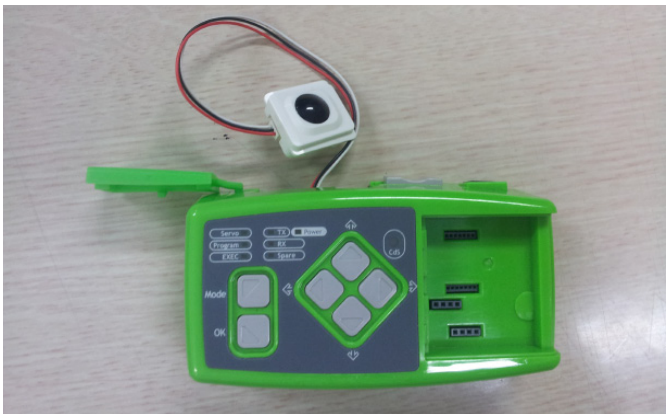
Hovis 리모컨의 경우, Address는 0x61부터 0x6A까지의 값을 가집니다. DRC 기본 펌웨어에서는 이 값을 채널이라는 값으로 나타냅니다. 리모컨의 채널은 1~0의 숫자 + OK를 동시에 누름으로써 설정할 수 있으며, 각 숫자에 대응되는 Address 값은 아래와 같습니다.

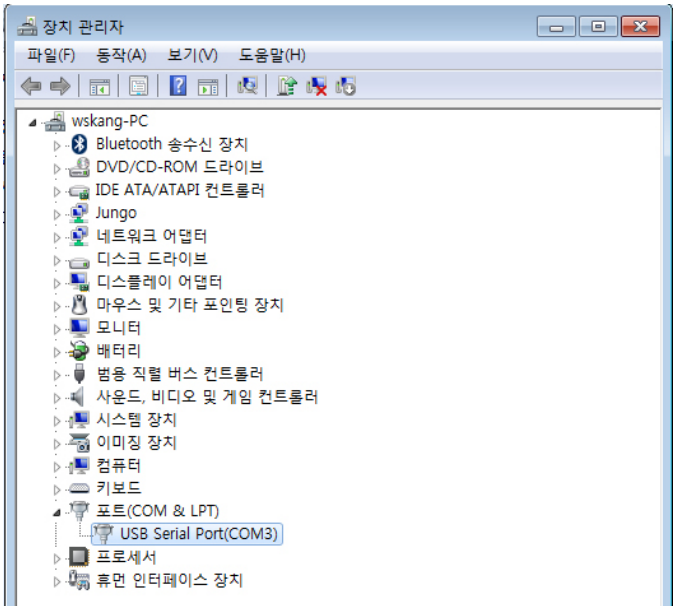
리모컨 버튼	RmcChannel 값
0+OK	97(0x61)
1+OK	98(0x62)
2+OK	99(0x63)
3+OK	100(0x64)
4+OK	101(0x65)
5+OK	102(0x66)
6+OK	103(0x67)
7+OK	104(0x68)
8+OK	105(0x69)
9+OK	106(0x6A)

또한 리모컨의 각 키마다 해당되는 버튼의 키 값은 아래 그림과 같습니다.



준비





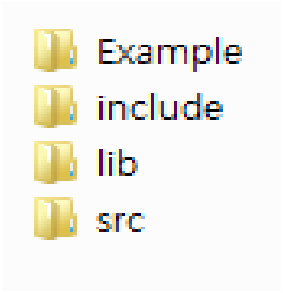
하드웨어 준비하기

01 연결

DRC 제어기 아답터로 전원을 연결하고, USB to Serial Cable 로 PC 와 DRC 를 연결합니다.
적외선 수신기를 연결합니다.리모컨도 같이 준비합니다.

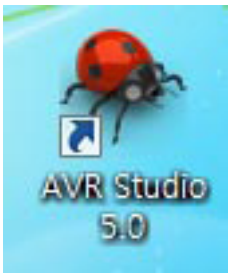
02 COM port 연결

내컴퓨터 〉 속성 〉 장치관리자에서 COM 연결 확인합니다.



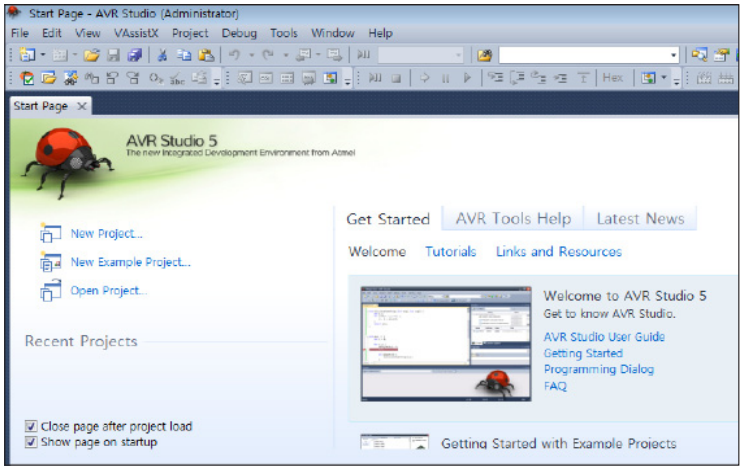
00 라이브러리

리모컨 입력을 받기 위해서는 기본 라이브러리 파일이 필요합니다.
라이브러리가 포함된 폴더 구조입니다.



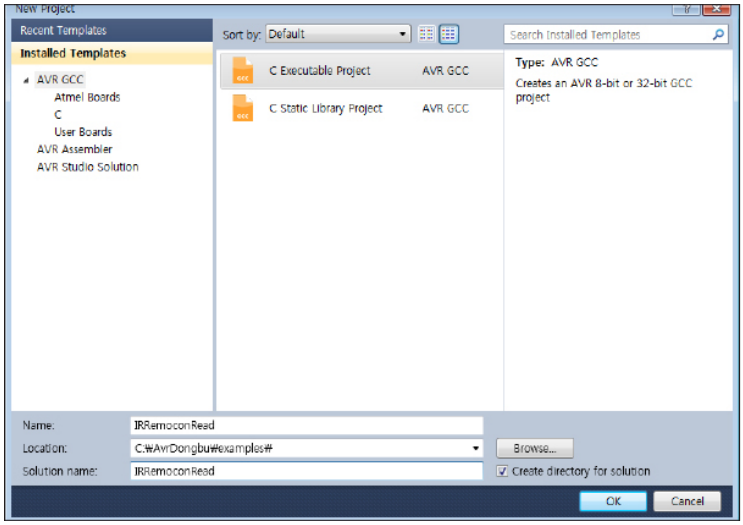
01 실행

AVR Studio 를 실행합니다.



02 New Project

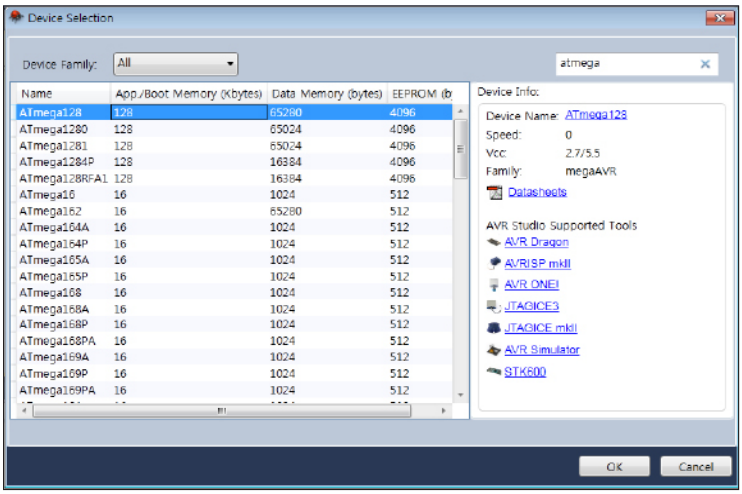
New Project 를 클릭합니다.



03 GCC 선택

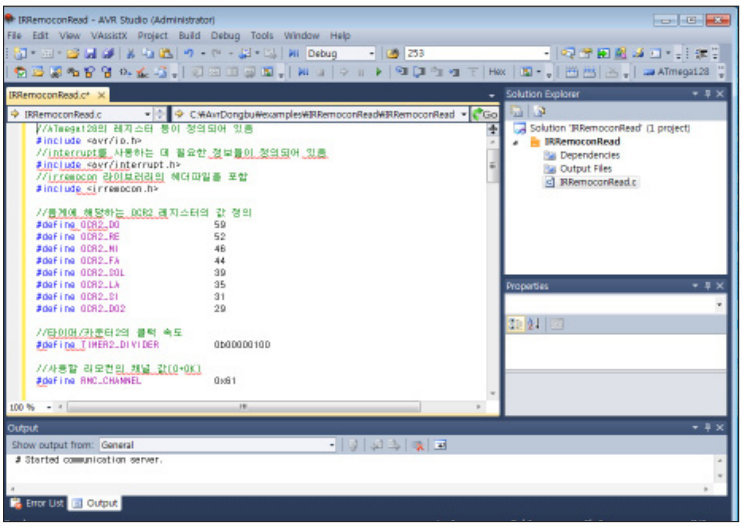
C Executable Project 를 선택합니다.

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



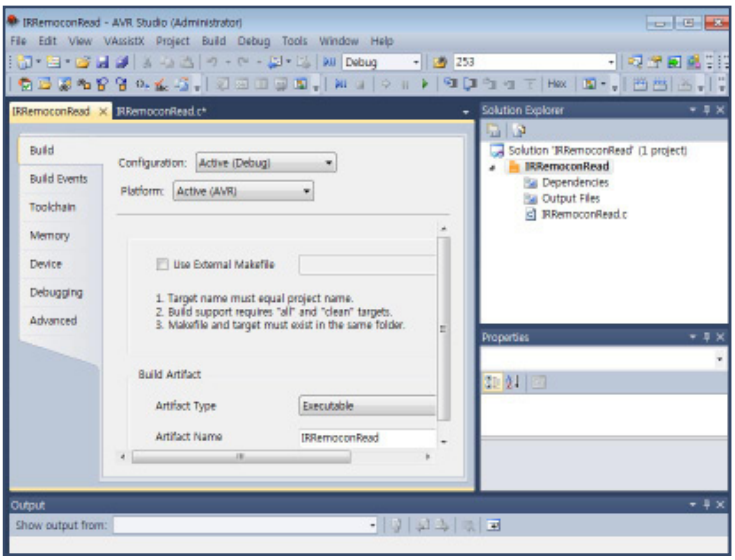
04 ATmega 선택

우측 상단에 atmega 를 검색하여,
리스트중에 ATmega128 을 선택합니다.



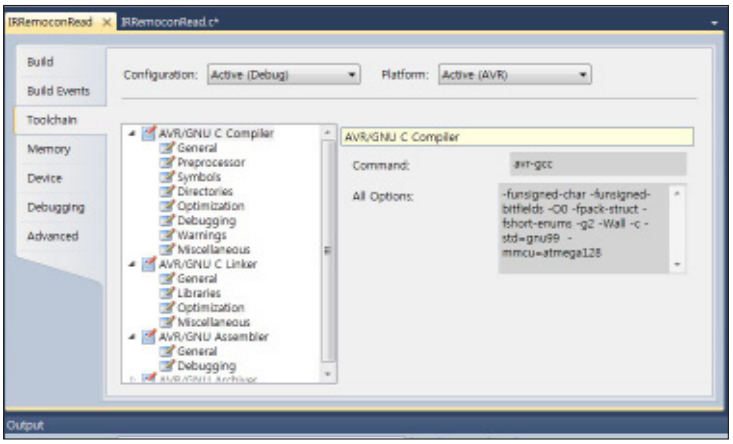
05 Editor

편집창에 코딩을 시작합니다.



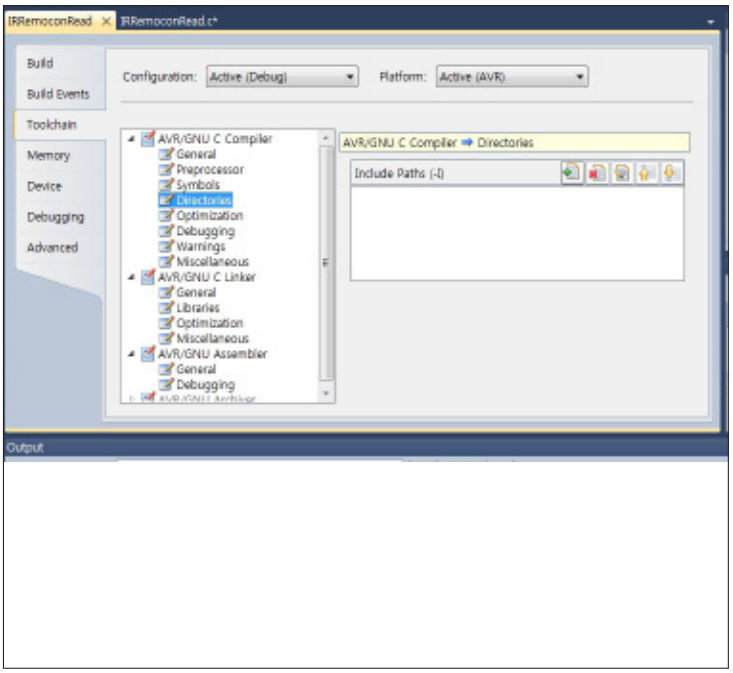
01 설정

AVR Studio 상단 메뉴중에 Project 를 클릭하면 IRRemoconRead (프로젝트 이름별로 다름) Property 메뉴가 생성되어 있습니다. 클릭하면 좌측과 같이 설정화면이 나옵니다.



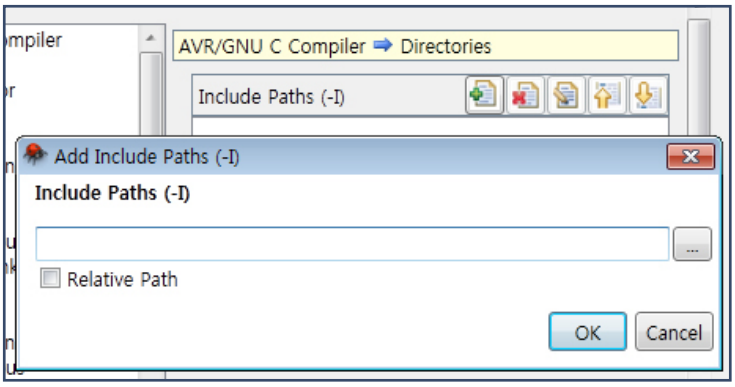
02 Toolchain

좌측 탭버튼중에 Toolchain 을 클릭하여, AVR/GNU C Compiler 을 선택합니다



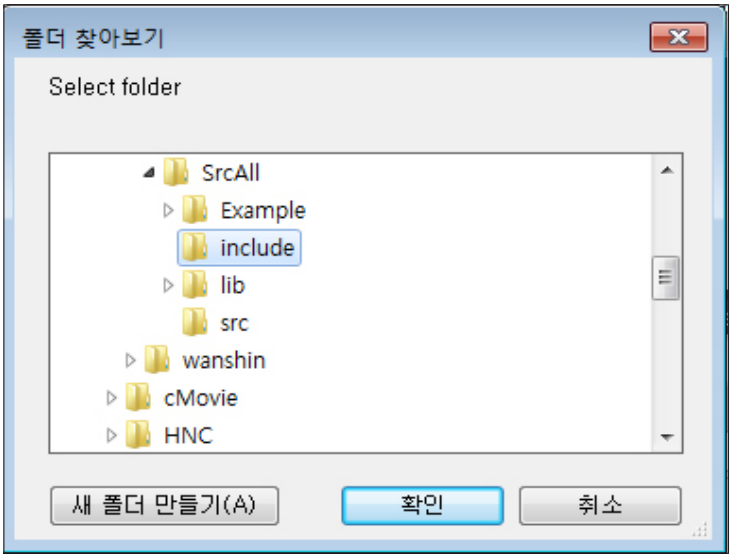
03 C Compiler

C Compiler 폴더중에 Directories 를 선택합니다.



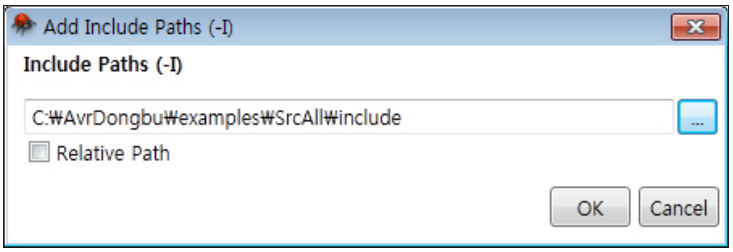
04 Path

Include Paths 오른쪽에 + 버튼을 클릭하면 팝업 입력창이 뜹니다.
Relative Path (상대경로) 체크를 해지하고(5.1에서는 안해도 됨), 우측 폴더 검색 버튼을 클릭합니다.



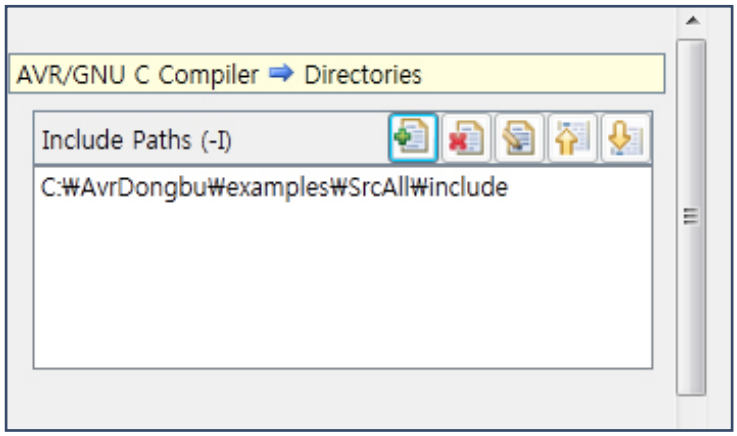
05 Include

제공 라이브러리 폴더중에 include 를 선택합니다.



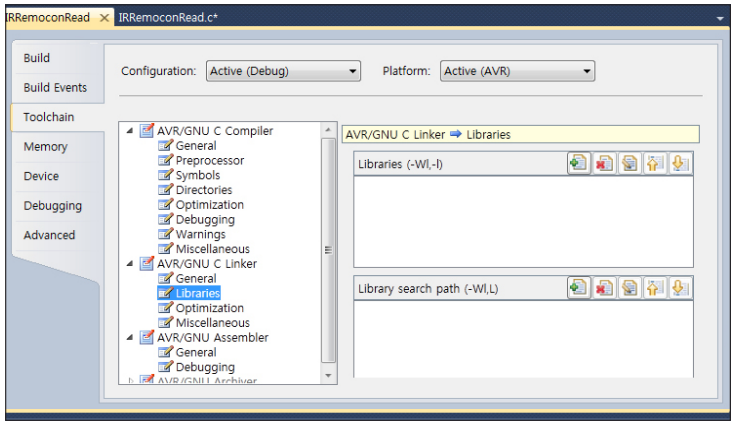
06 Include 완료

Include 폴더 위치가 지정됩니다. OK 버튼을 클릭합니다.



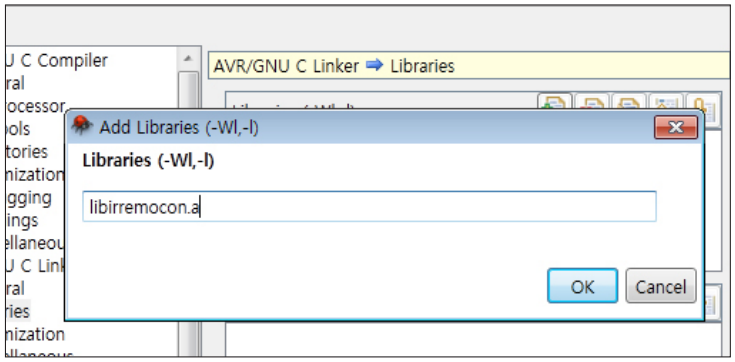
07 Path 완료

Include 경로지정이 완료되었습니다.



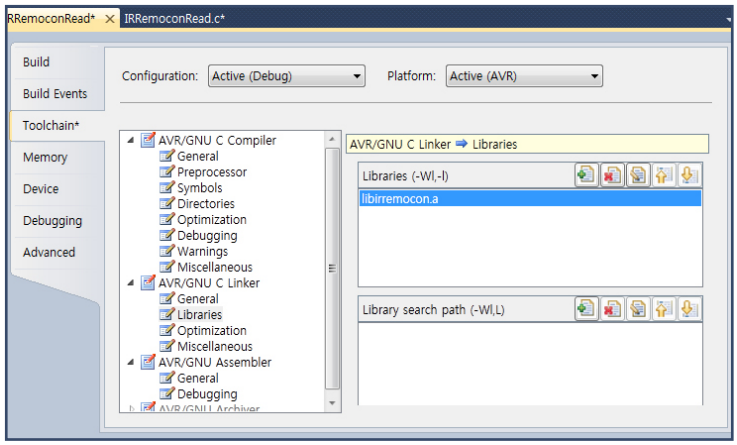
08 C Linker

C Linker 폴더중에 Libraries 를 선택합니다.



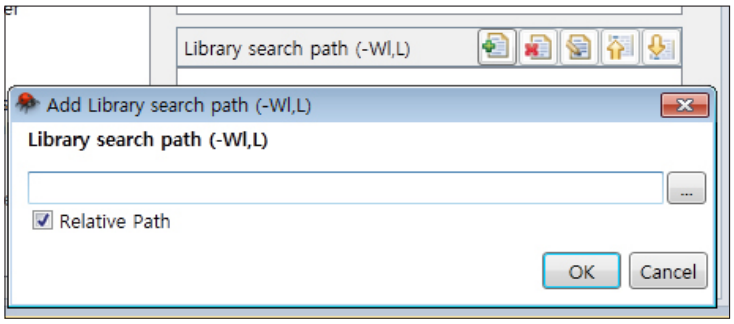
09 Libraries

Libraries 창 우측 + 를 클릭하면,
Libraries 파일명 입력창이 나옵니다.
libirremocon.a 를 직접 입력합니다.
OK 버튼을 클릭합니다.



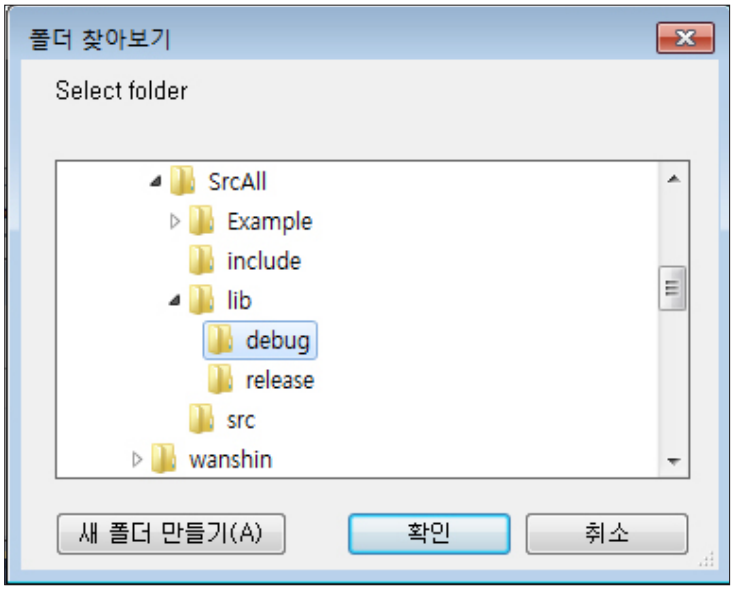
10 Search

리모컨 라이브러리 파일 libaccgyro.a 링크가 완료되었습니다.
다음으로 Library search path 를 설정합니다. 우측 + 버튼을 클릭합니다.



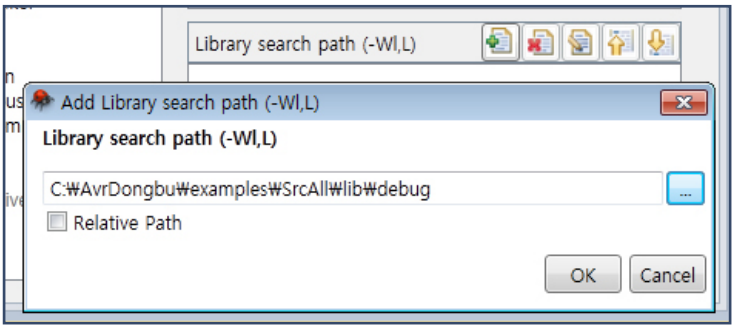
11 Path

Library search path 팝업창 우측에 폴더 검색 버튼을 클릭합니다.



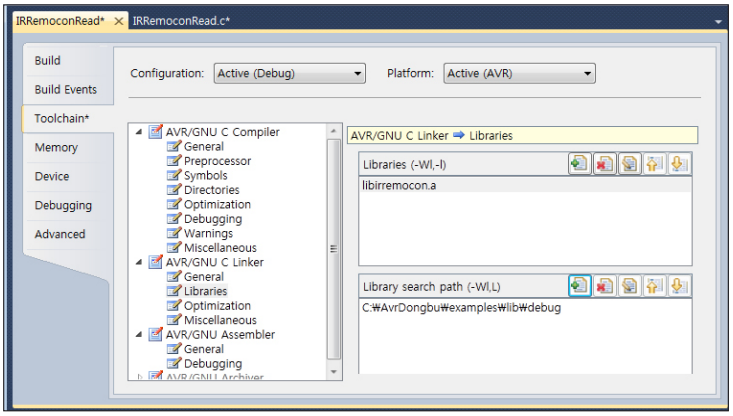
12 Debug

제공 라이브러리 폴더에서 lib 하위 폴더에 debug 를 선택합니다.



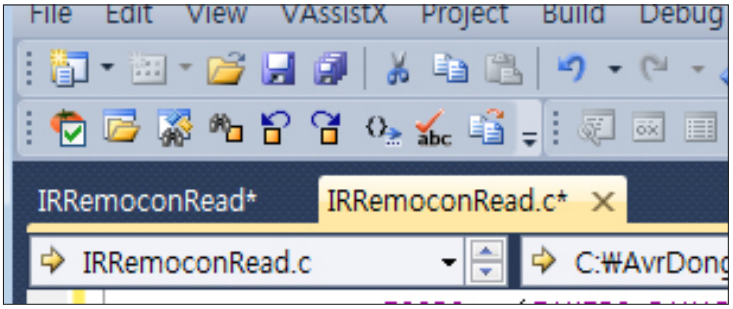
13 Path 완료

Relative Path(상대경로지정) 체크박스를 해지합니다.
Library search path 경로지정이 완료되었습니다.
OK 버튼을 클릭합니다.



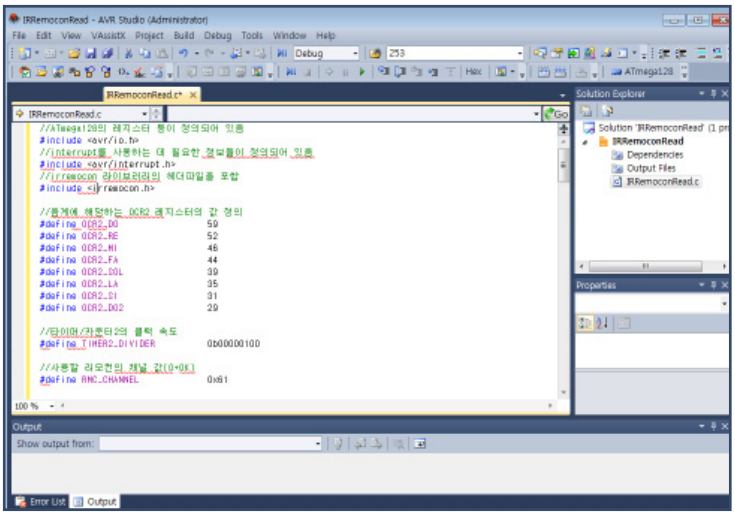
14 Link 완료

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;
}
```

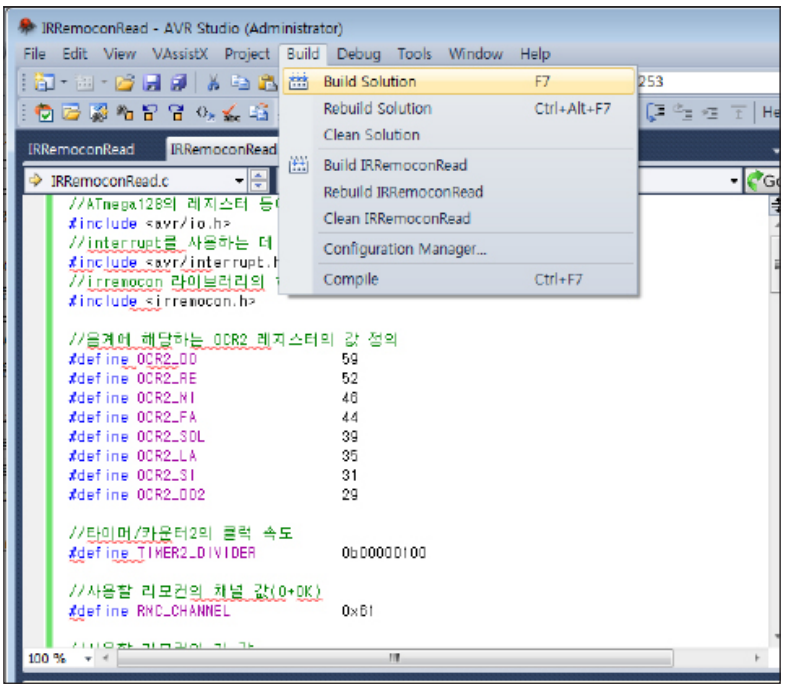
리모컨이 눌려있지 않거나 눌려 있지만 1~8번 키가 아닐 경우에는, 타이머/카운터2 레지스터를 모두 0으로 만들어서 중지시킵니다. 이렇게 하면 버저의 울림이 멈추게 됩니다.

```
ucDataOld = ucData;
```

while 루프가 끝날 때마다 현재 루프에서 읽은 ucData의 값을 ucDataOld에 대입해, 다음 루프에서 비교하는 데에 사용합니다.

실행

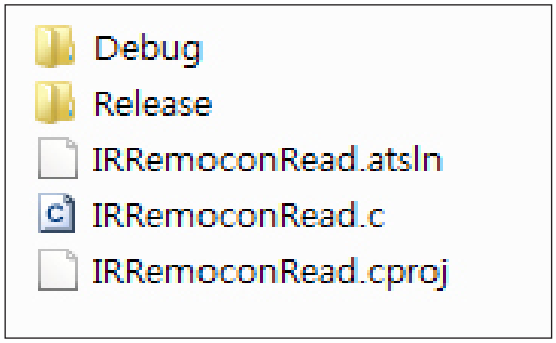
적외선 리모컨 수신 모듈을 DRC에 장착한 후 펌웨어를 실행하면, 리모컨 키를 누를 때 해당하는 버저 음이 재생됩니다.



01 Build

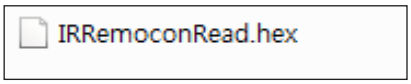
Build 메뉴를 선택하여 Build Solution을 클릭하여 Build 시킵니다.

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



02 Debug 폴더

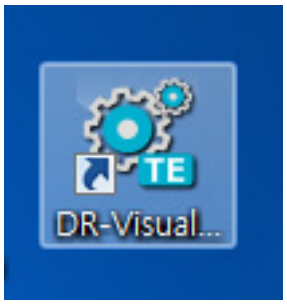
Build 후에 소스가 있는 폴더에 들어가면 Debug 폴더가 생성 되어있습니다. 클릭하여 폴더 안의 내용을 봅니다.



03 hex 확인

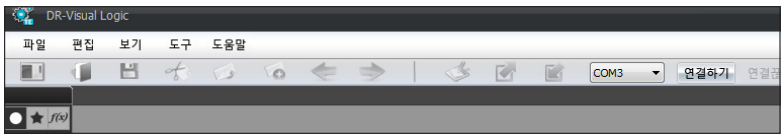
Debug 폴더 안의 hex 파일 확인합니다. Hex 파일이 제어기에 포팅시키는 파일입니다.

DR-Visual Logic 에서 펌웨어 복구를 실시합니다.



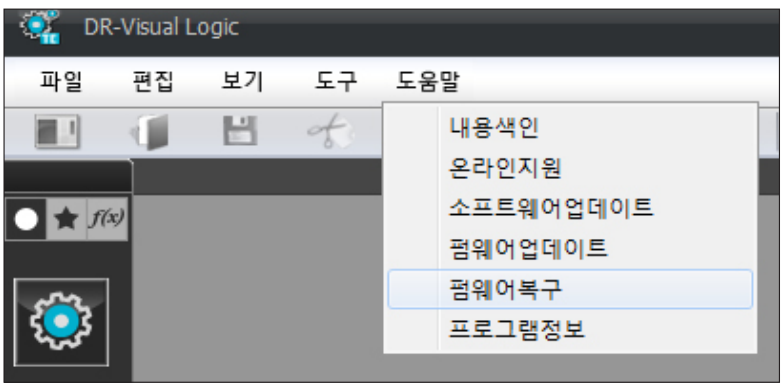
04 Visual Logic

DR-Visual Logic 을 실행시킵니다.



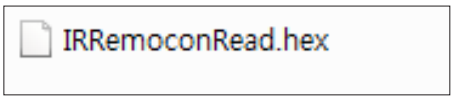
05 연결하기

COM 번호를 설정하고 연결하기를 클릭합니다.



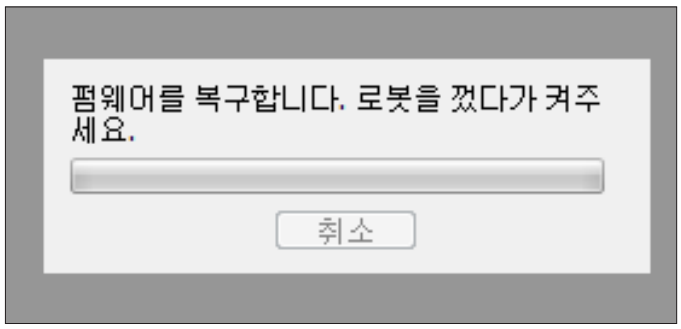
06 펌웨어 복구

도움말 메뉴에서 펌웨어복구를 클릭합니다.



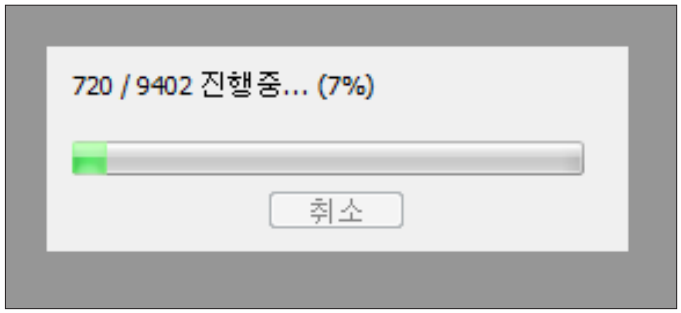
07 불러오기

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 04

로봇에 응용하기

CH.1 : 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.

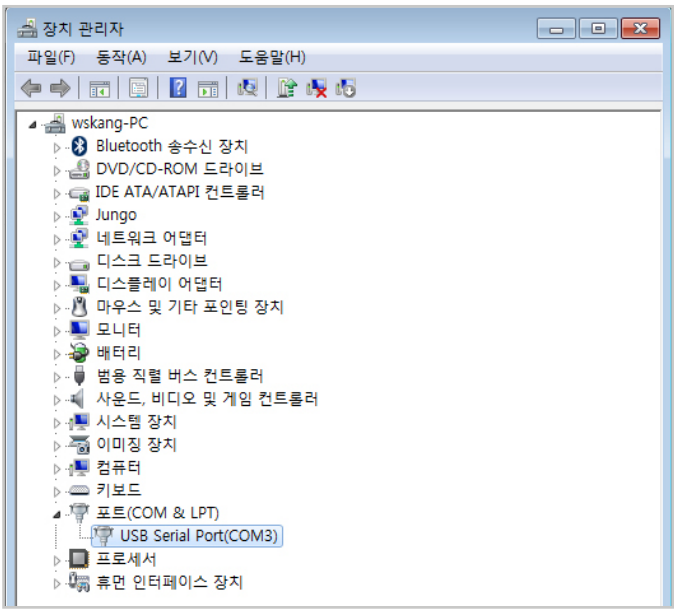
이름	CMD	설명
RAM_WRITE	0x03	HerkuleX의 휘발성 레지스터 영역에 데이터를 씁니다.
S_JOG	0x06	HerkuleX의 위치/속도를 제어하여 움직이게 합니다. I_JOG는 각 모터마다 이동 시간을 다르게 할 수 있는 반면, S_JOG는 모든 시간을 같은 시간동안 움직여서 목표 위치에 도달합니다. I_JOG와 S_JOG 패킷에서 시간을 나타내기 위해서는 1바이트 크기의 데이터를 사용하는데, 이때 데이터는 1당 11.2ms를 나타냅니다. 즉 I_JOG 패킷에 50의 시간 데이터가 들어가 있다면 실제로 모터는 50*11.2=560ms 동안 목표 위치로 이동합니다.

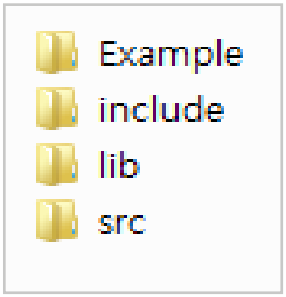
이번 예제에서 사용된 휘발성 레지스터 영역의 항목을 요약하면 아래와 같습니다.

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.

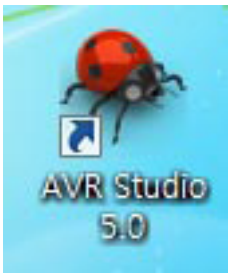
02 COM port 연결

내컴퓨터 〉 속성 〉 장치관리자에서 COM 연결 확인합니다.

AVR Studio 실행 및 코딩 준비

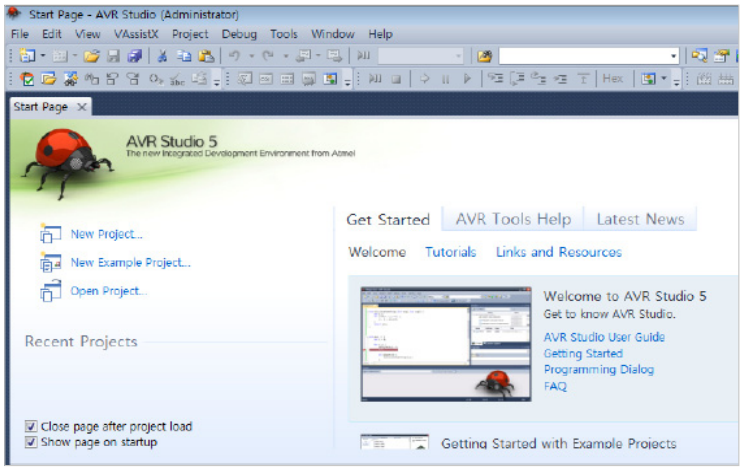
00 라이브러리

Hovis Lite 를 제어하기 위해서는 기본 라이브러리 파일이 필요합니다.
라이브러리가 포함된 폴더 구조입니다.



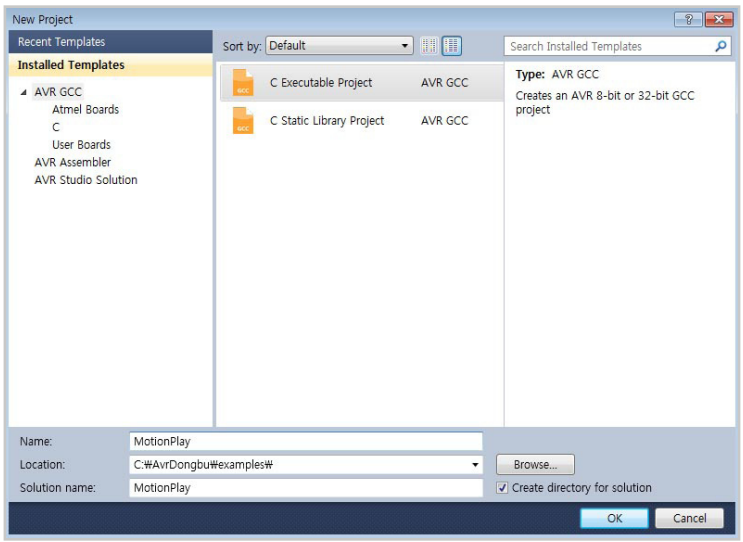
01 실행

AVR Studio 를 실행합니다.



02 New Project

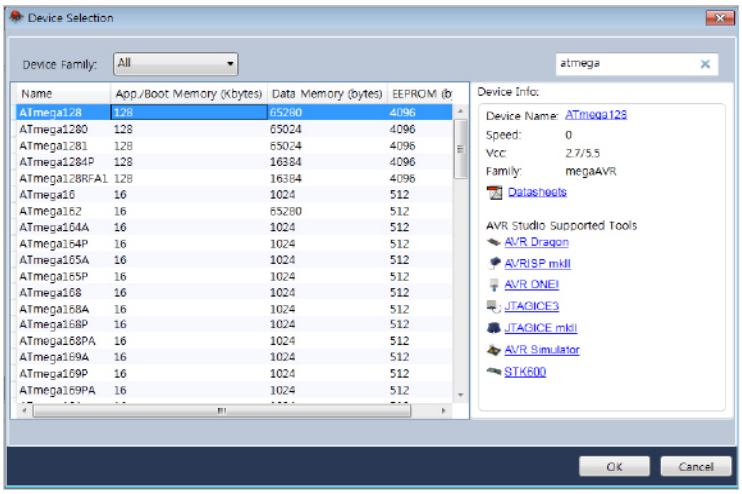
New Project 를 클릭합니다.



03 GCC 선택

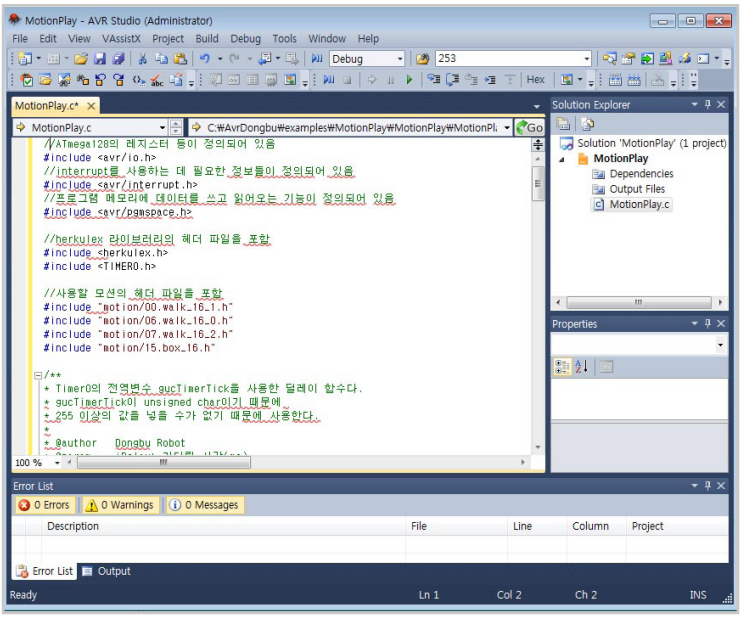
C Executable Project 를 선택합니다.

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



04 ATmega 선택

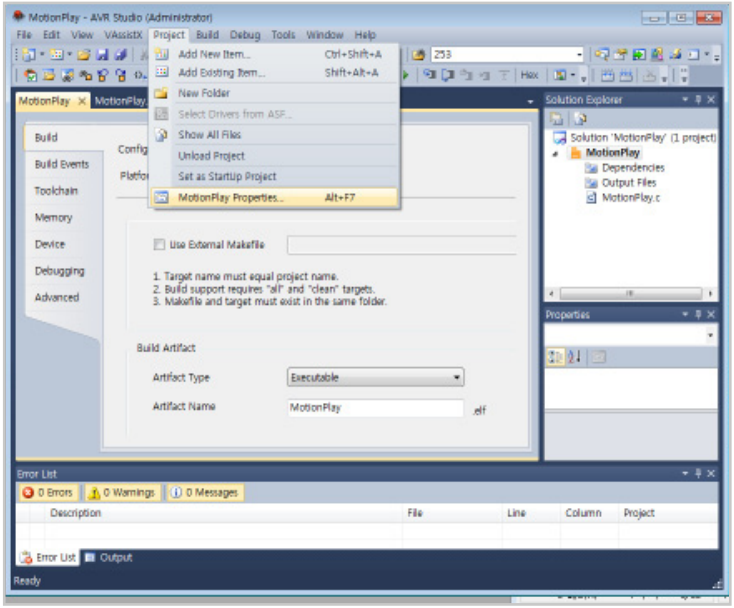
우측 상단에 atmega 를 검색하여, 리스트중에 ATmega128 을 선택합니다.



05 Editor

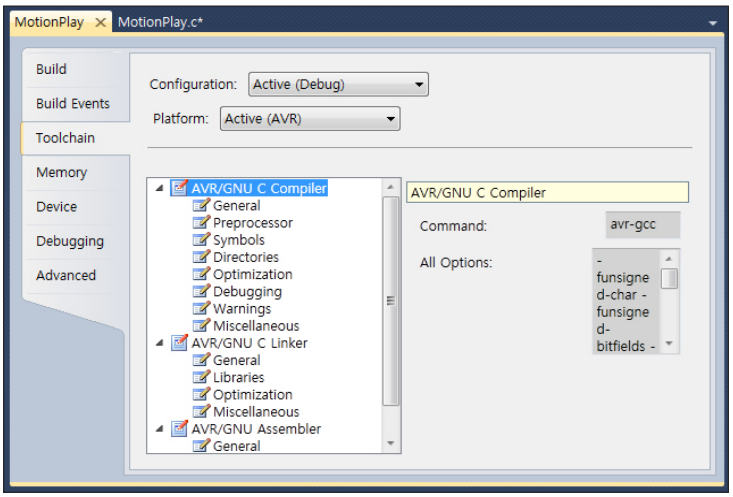
편집창에 코딩을 시작합니다.

라이브러리 링크 및 설정



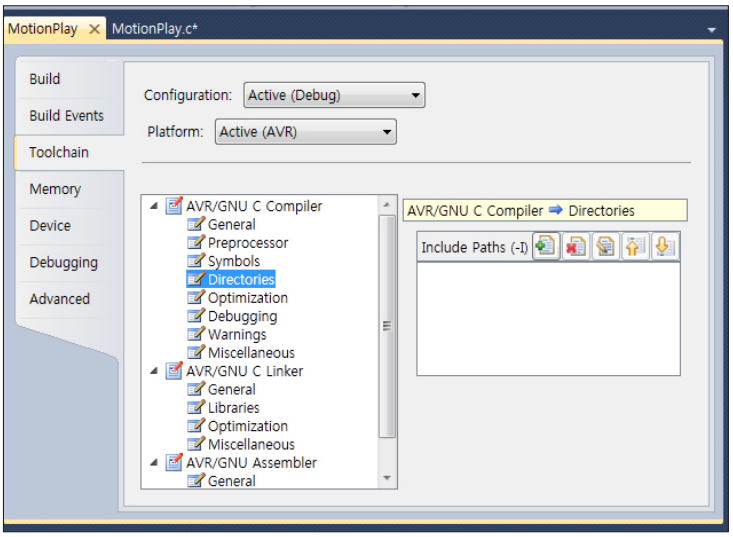
01 설정

AVR Studio 상단 메뉴중에 Project 를 클릭하면 MotionPlay (프로젝트 이름별로 다름) Property 메뉴가 생성되어 있습니다. 클릭하면 좌측과 같이 설정화면이 나옵니다.



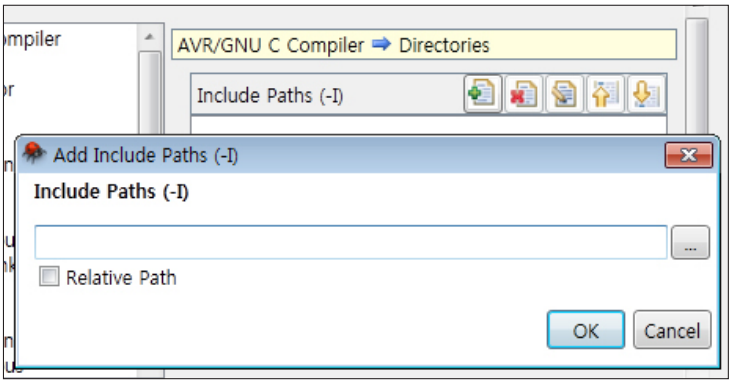
02 Toolchain

좌측 탭버튼중에 Toolchain 을 클릭하여, AVR/GNU C Compiler 을 선택합니다.



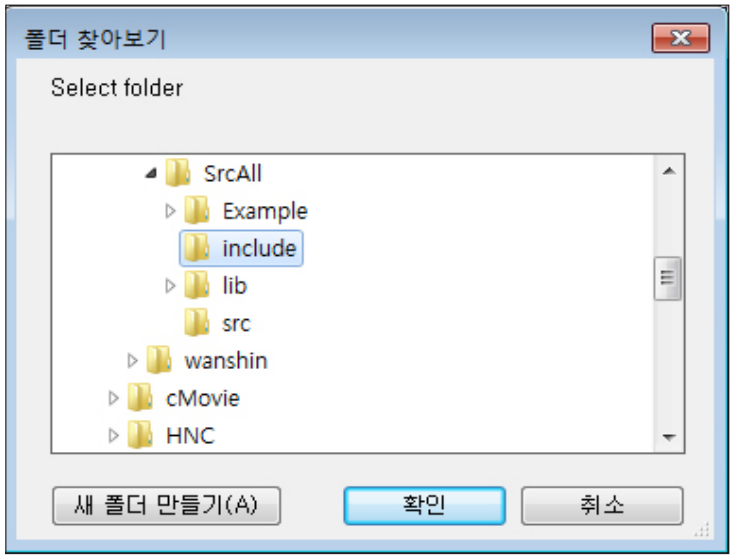
03 C Compiler

C Compiler 폴더중에 Directories 를 선택합니다.



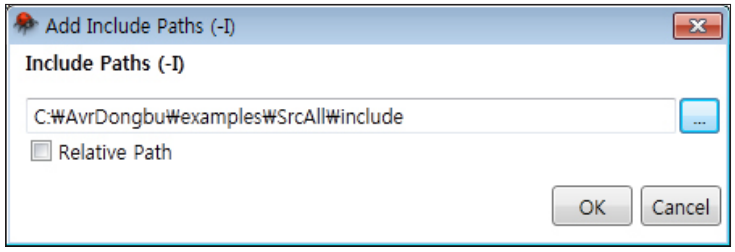
04 Path

Include Paths 오른쪽에 + 버튼을 클릭하면 팝업 입력창이 뜹니다.
Relative Path (상대경로) 체크를 해지하고,
우측 폴더 검색 버튼을 클릭합니다.



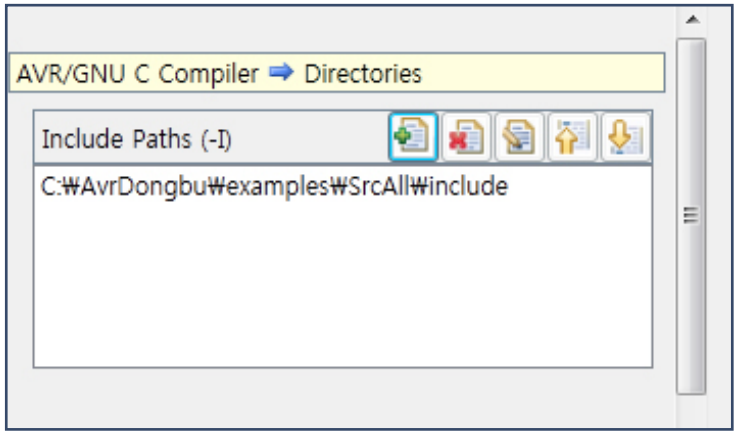
05 Include

제공 라이브러리 폴더중에 include 를 선택합니다.



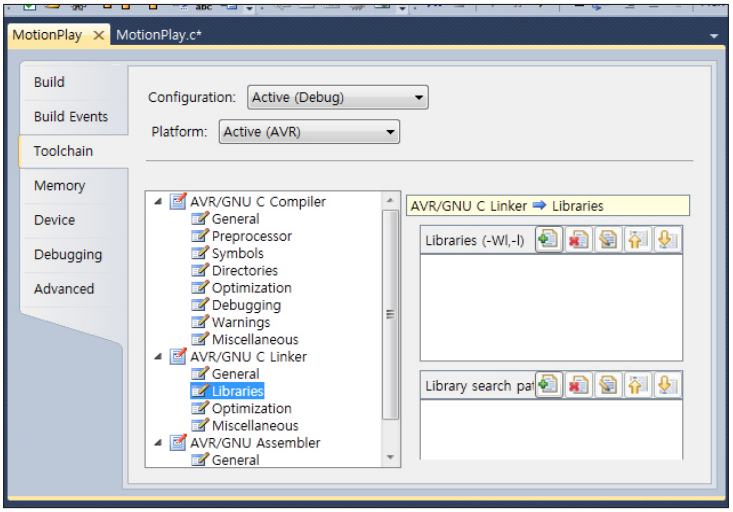
06 Include 완료

Include 폴더 위치가 지정됩니다.
OK 버튼을 클릭합니다.



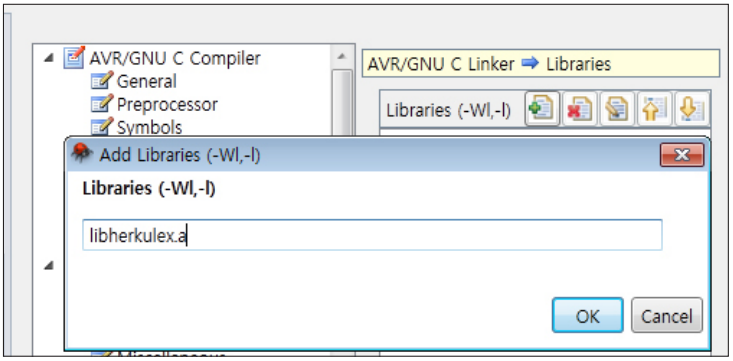
07 Path 완료

Include 경로지정이 완료되었습니다.



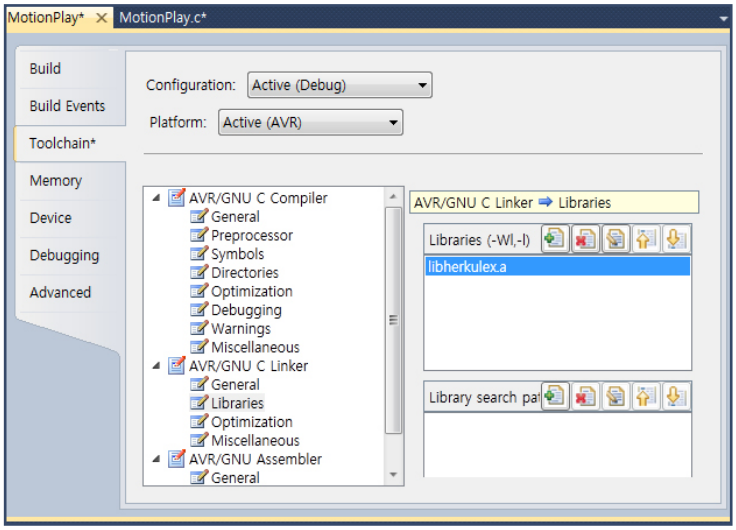
08 C Linker

C Linker 폴더중에 Libraries 를 선택합니다.



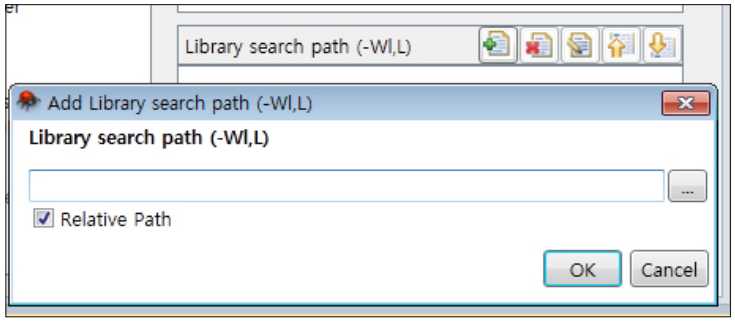
09 Libraries

Libraries 창 우측 + 를 클릭하면, Libraries 파일명 입력창이 나옵니다. libherkulex.a 를 직접 입력합니다.
OK 버튼을 클릭합니다.



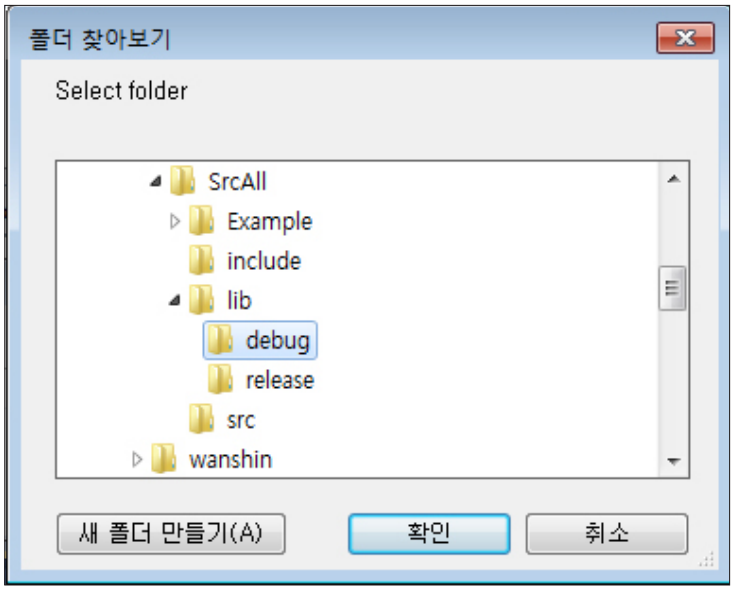
10 Search

서보모터 라이브러리 파일 libherkulex.a 링크가 완료되었습니다.
다음으로
Library search path 를 설정합니다.
우측 + 버튼을 클릭합니다.



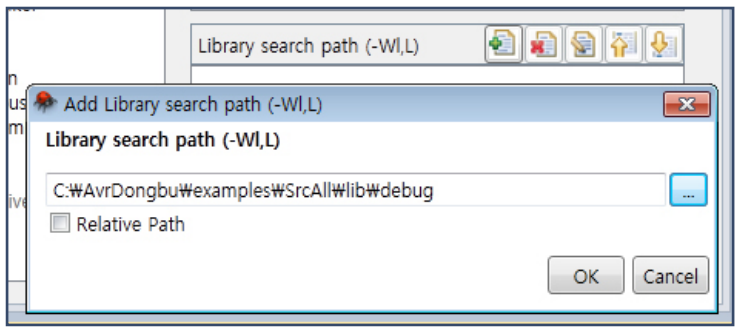
11 Path

Library search path 팝업창 우측에 폴더 검색 버튼을 클릭합니다.



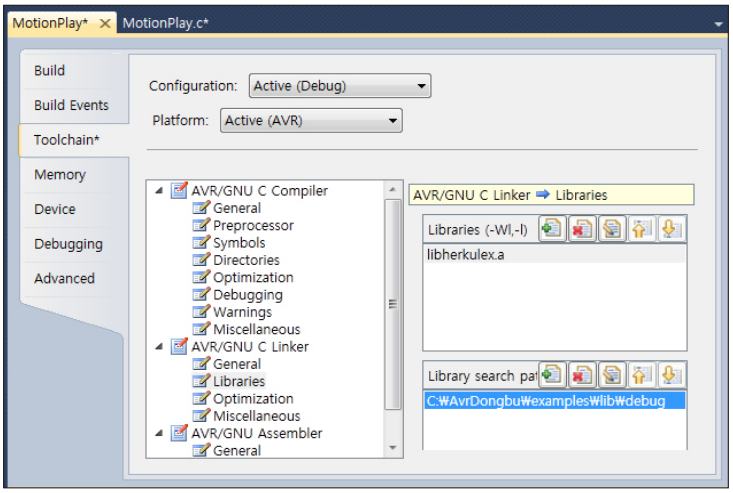
12 Debug

제공 라이브러리 폴더에서 lib 하위 폴더에 debug 를 선택합니다.



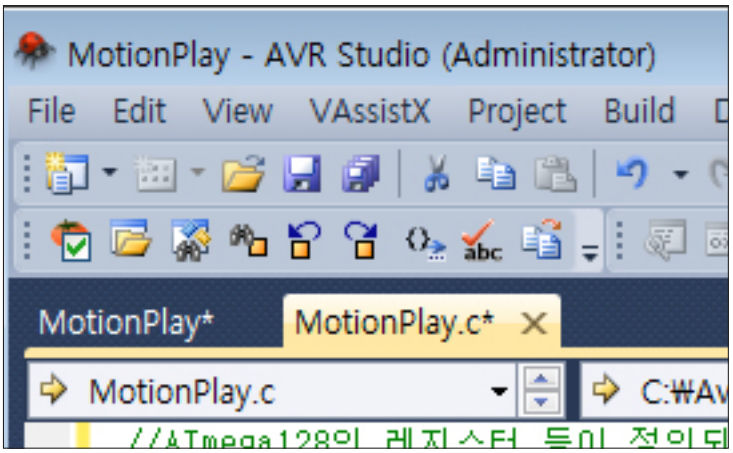
13 Path 완료

Relative Path(상대경로지정) 체크박스를 해지합니다.
Library search path 경로지정이 완료되었습니다.
OK 버튼을 클릭합니다.



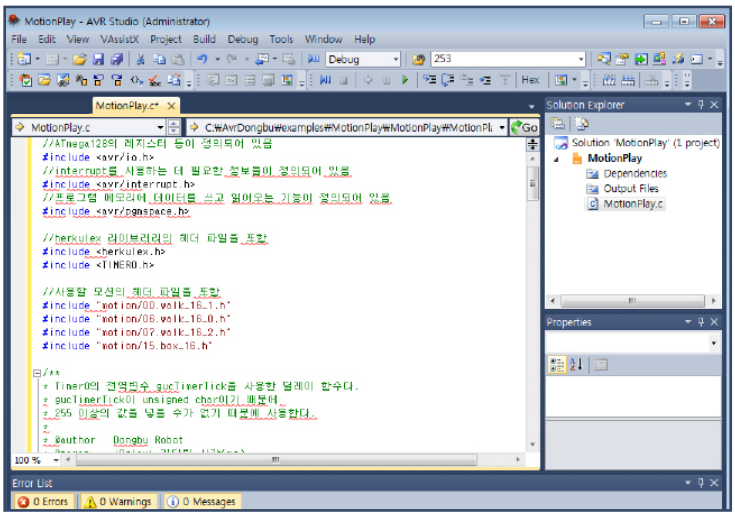
14 Link 완료

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.

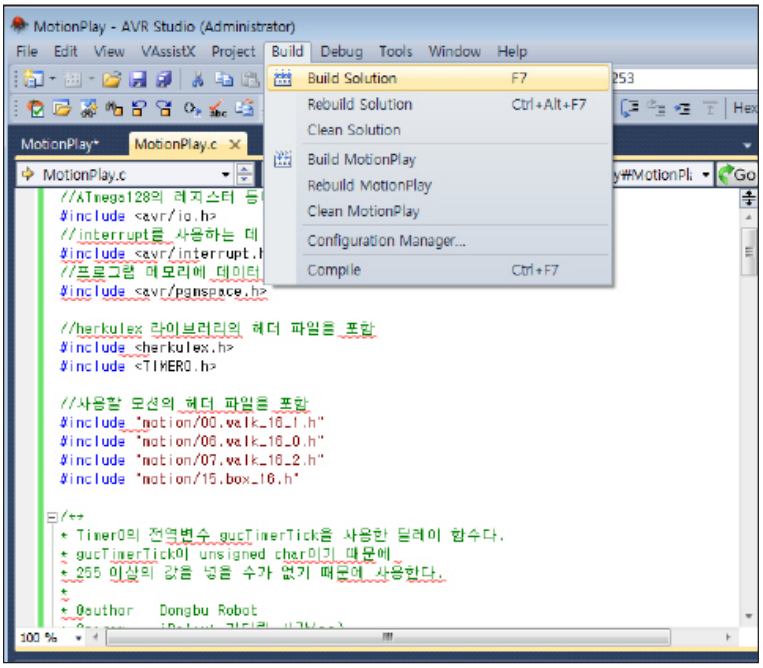
```
delay_ms(1000);
```

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

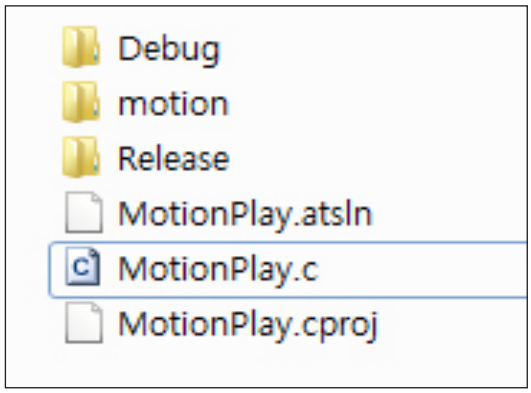
DRC를 16축 Hovis Lite에 연결, 펌웨어를 다운로드하고 실행하면 로봇이 열 발자국 전진 후 복싱 모션을 계속 반복합니다.



01 Build

Build 메뉴를 선택하여 Build Solution을 클릭하여 Build 시킵니다.

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



02 Debug 폴더

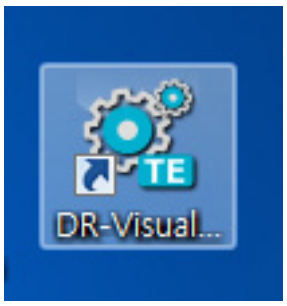
Build 후에 소스가 있는 폴더에 들어가면 Debug 폴더가 생성 되어있습니다. 클릭하여 폴더 안의 내용을 봅니다.



03 hex 확인

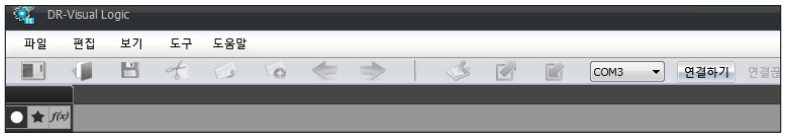
Debug 폴더 안의 hex 파일 확인합니다. Hex 파일이 제어기에 포팅시키는 파일입니다.

DR-Visual Logic 에서 펌웨어 복구를 실시합니다.



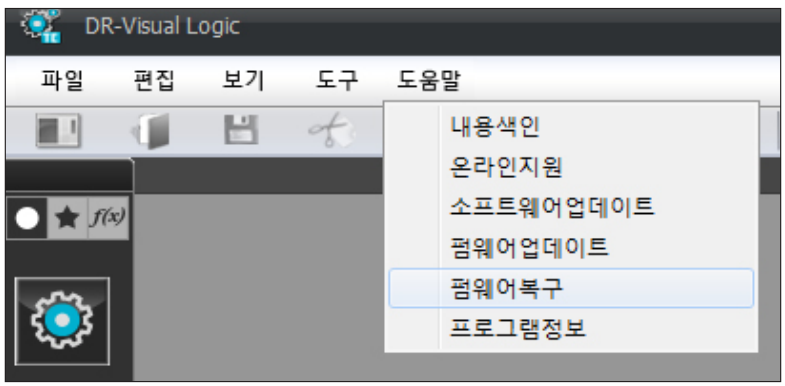
04 Visual Logic

DR-Visual Logic 을 실행시킵니다.



05 연결하기

COM 번호를 설정하고 연결하기를 클릭합니다.



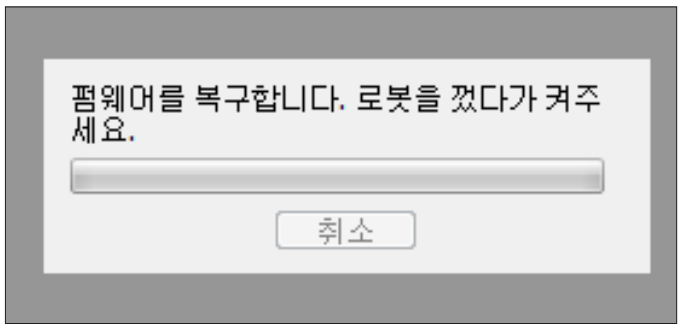
06 펌웨어 복구

도움말 메뉴에서 펌웨어복구를 클릭합니다.



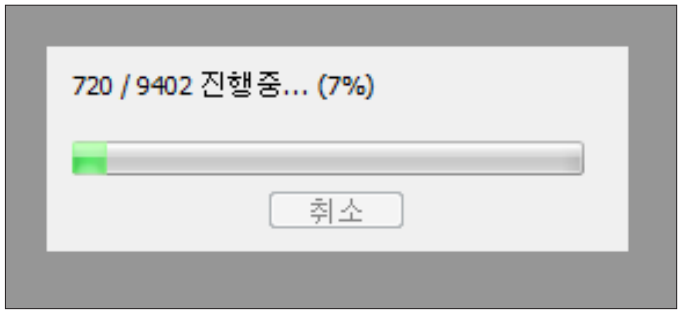
07 불러오기

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.

CH.2 : 적외선 리모컨으로 덤프트럭 제어하기

herkulex 라이브러리, irremocon 라이브러리를 사용해 덤프트럭 로봇을 제어합니다.

H/W 원리

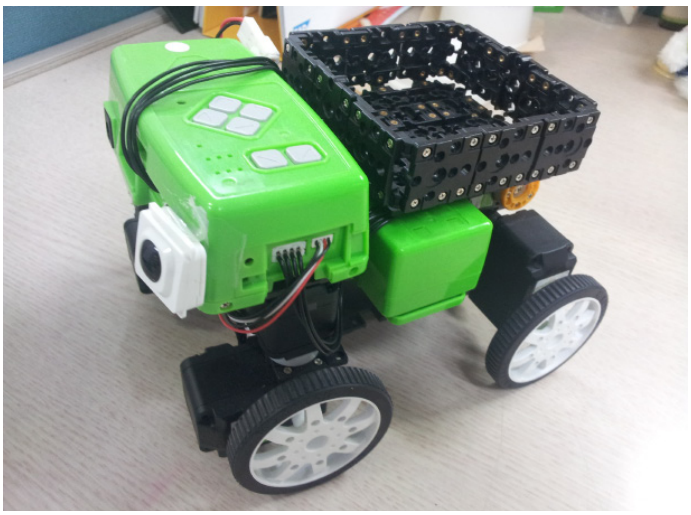
이번 예제에서는 HerkuleX를 제어하는 라이브러리와 적외선 리모컨을 읽어오는 라이브러리를 동시에 사용해 덤프트럭 로봇을 제어합니다. HerkuleX 제어에 관련된 내용은 Chapter 8을, 적외선 리모컨과 관련된 내용은 Chapter 12를 참조하세요. 이번 예제에서 사용하는 패킷에 사용된 명령을 요약하면 아래와 같습니다.

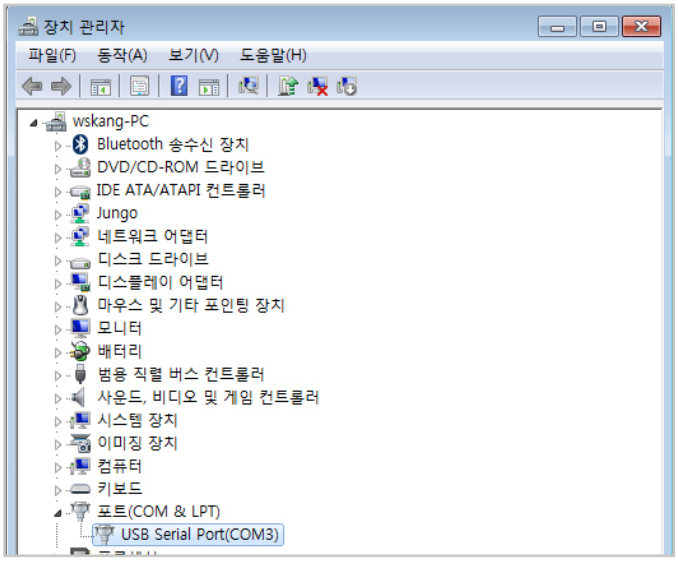
이름	CMD	설명
RAM_WRITE	0x03	HerkuleX의 휘발성 레지스터 영역에 데이터를 씁니다.
I_JOG	0x05	HerkuleX의 위치/속도를 제어하여 움직이게 합니다. 각 모터마다 이동 시간을 다르게 설정할 수 있습니다. I_JOG와 S_JOG 패킷에서 시간을 나타내기 위해서는 1바이트 크기의 데이터를 사용하는데, 이때 데이터는 1당 11.2ms를 나타냅니다. 즉 I_JOG 패킷에 50의 시간 데이터가 들어가 있다면 실제로 모터는 50*11.2=560ms 동안 목표 위치로 이동합니다.

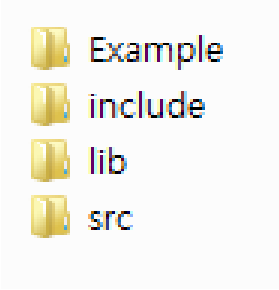
이번 예제에서 사용된 휘발성 레지스터 영역의 항목을 요약하면 아래와 같습니다.

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.

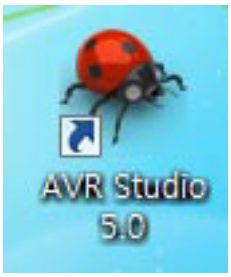
02 COM port 연결

내컴퓨터〉속성〉장치관리자에서 COM 연결 확인합니다.

AVR Studio 실행 및 코딩 준비

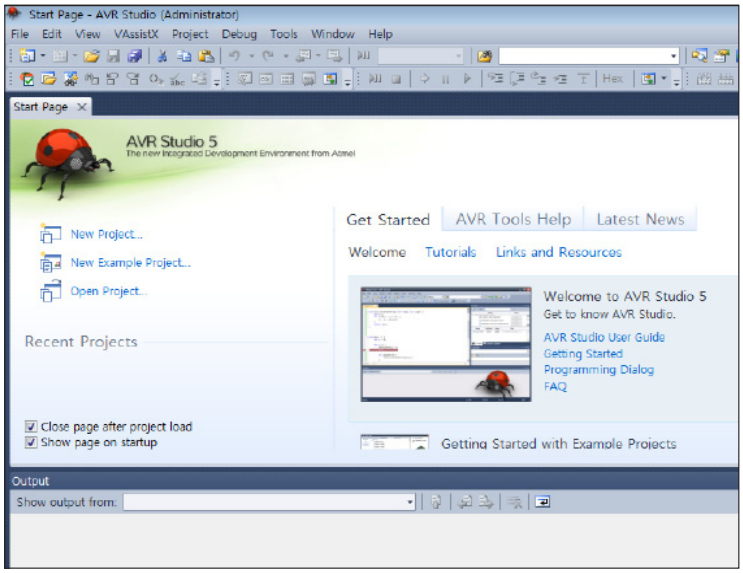
00 라이브러리

덤프트럭을 제어하기 위해서는 기본 라이브러리 파일이 필요합니다.
라이브러리가 포함된 폴더 구조입니다.



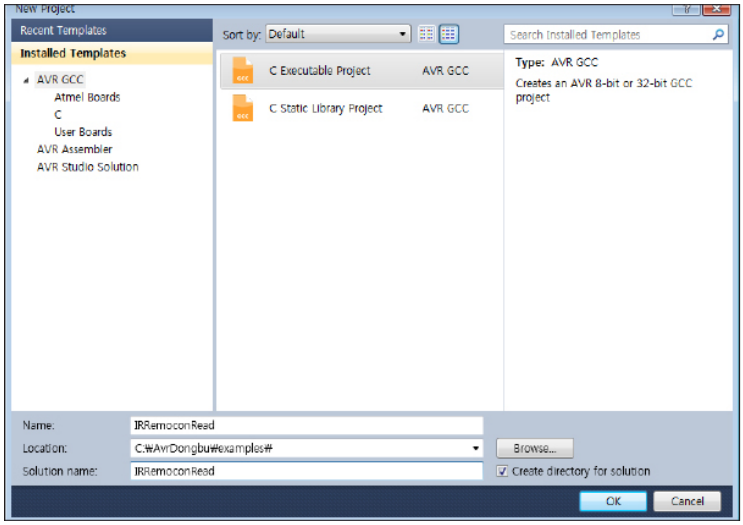
01 실행

AVR Studio 를 실행합니다.



02 New Project

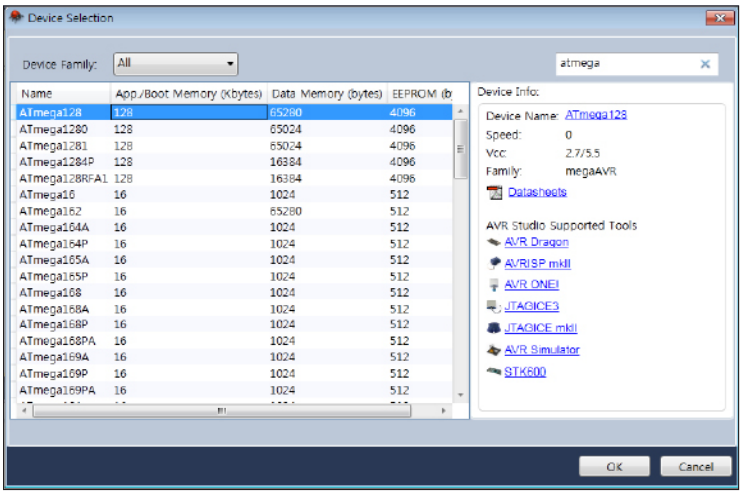
New Project 를 클릭합니다.



03 GCC 선택

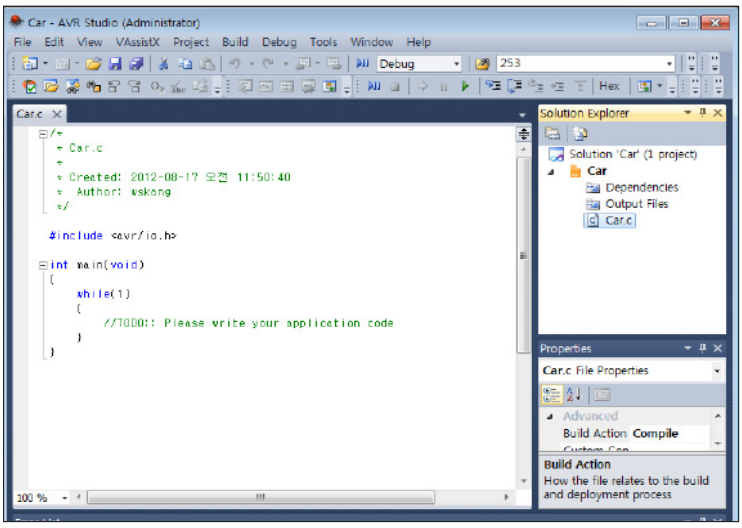
C Executable Project 를 선택합니다.

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



04 ATmega 선택

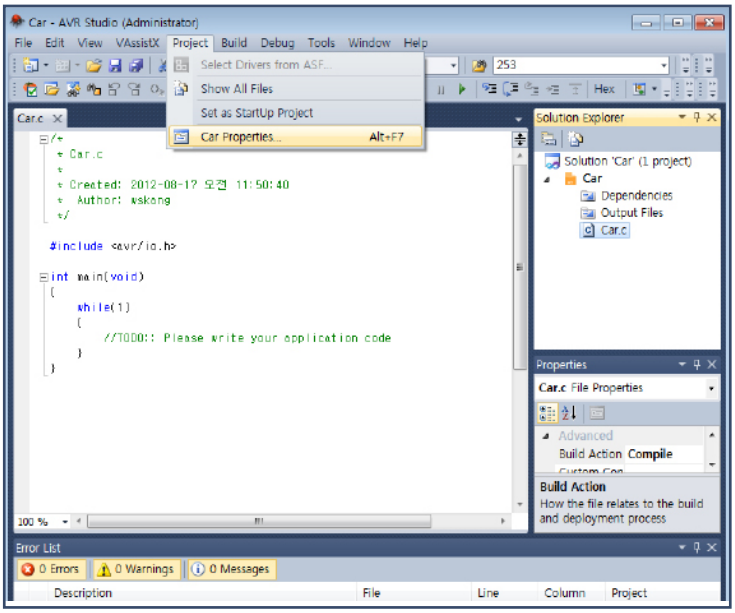
우측 상단에 atmega 를 검색하여, 리스트중에 ATmega128 을 선택합니다.



05 Editor

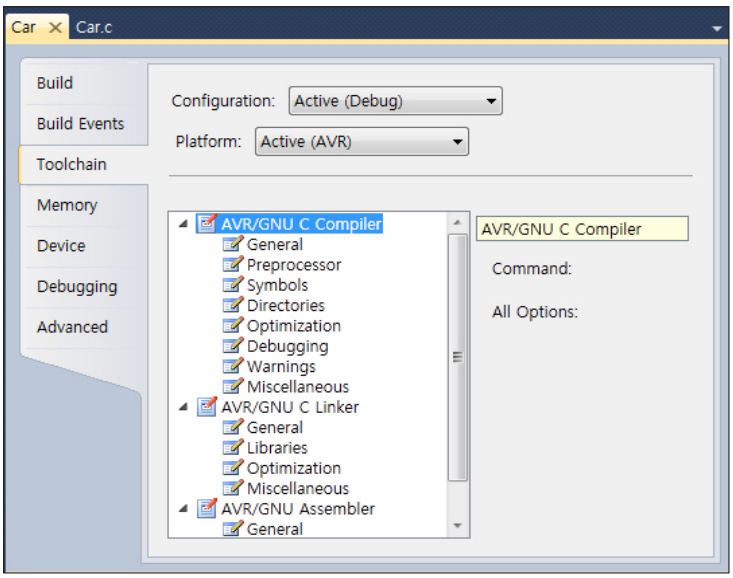
편집창에 코딩을 시작합니다.

라이브러리 링크 및 설정



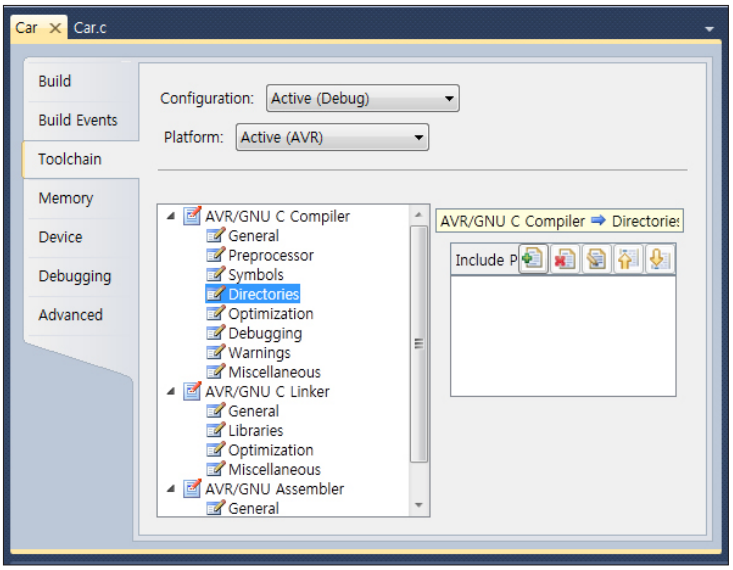
01 설정

AVR Studio 상단 메뉴중에 Project 를 클릭하면 Car (프로젝트 이름별로 다름) Property 메뉴가 생성되어 있습니다. 클릭하면 좌측과 같이 설정화면이 나옵니다.



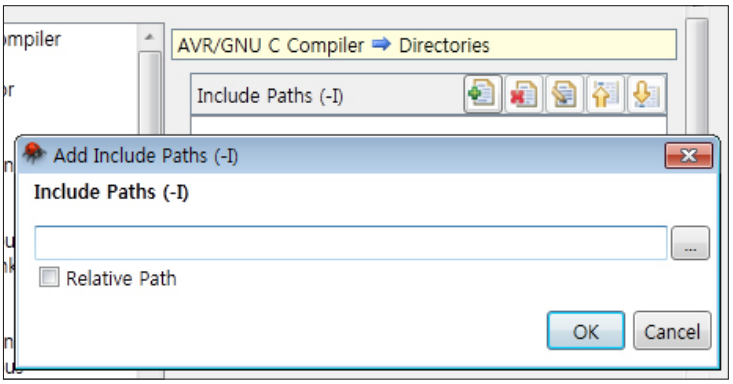
02 Toolchain

좌측 탭버튼중에 Toolchain 을 클릭하여, AVR/GNU C Compiler 을 선택합니다.



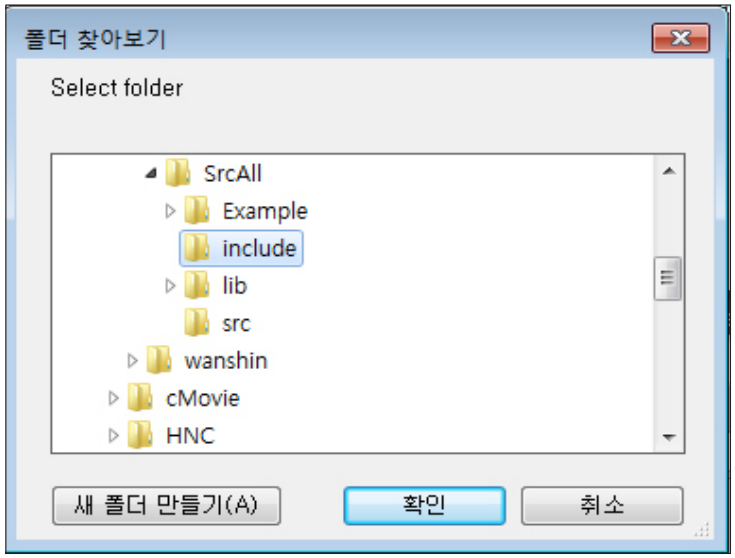
03 C Compiler

C Compiler 폴더중에 Directories 를 선택합니다.



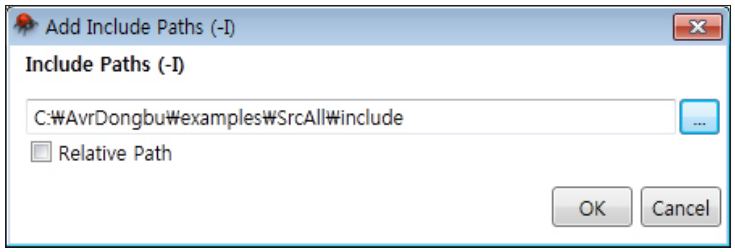
04 Path

Include Paths 오른쪽에 + 버튼을 클릭하면 팝업 입력창이 뜹니다.
Relative Path (상대경로) 체크를 해지하고,
우측 폴더 검색 버튼을 클릭합니다.



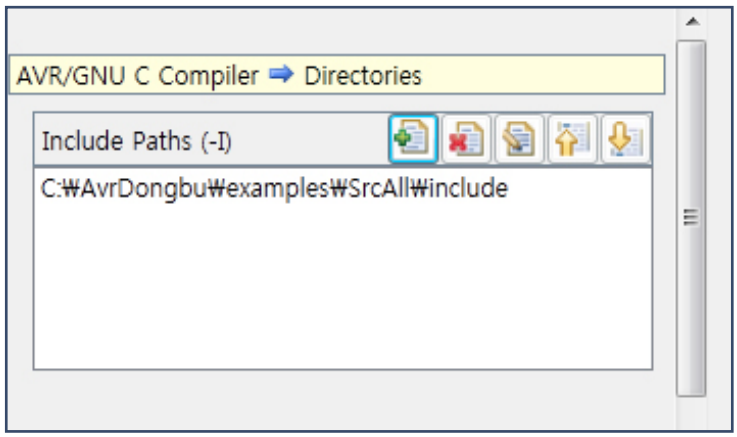
05 Include

제공 라이브러리 폴더중에 include 를 선택합니다.



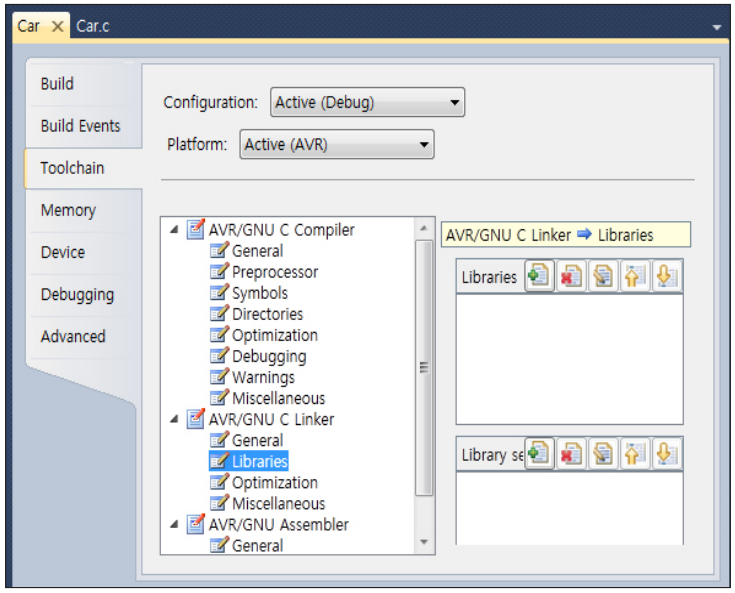
06 Include 완료

Include 폴더 위치가 지정됩니다. OK 버튼을 클릭합니다.



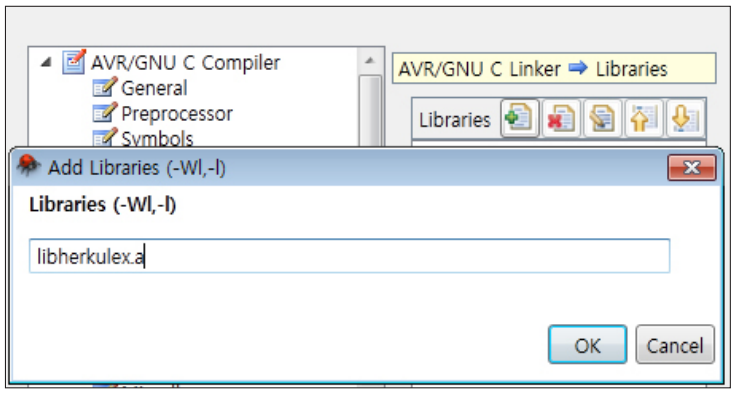
07 Path 완료

Include 경로지정이 완료되었습니다.



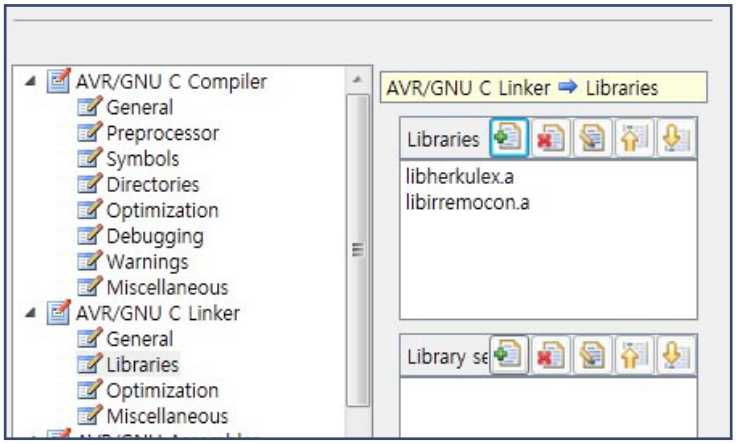
08 C Linker

C Linker 폴더중에 Libraries 를 선택합니다.



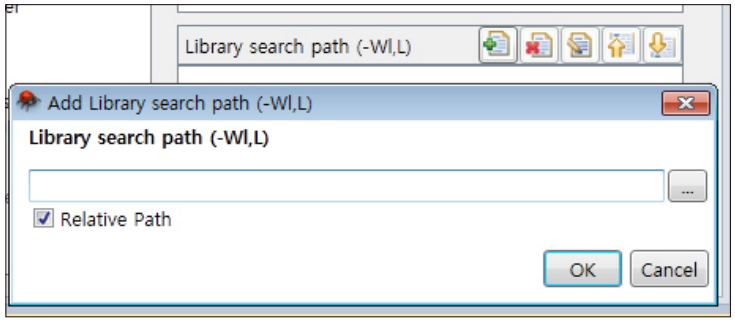
09 Libraries

Libraries 창 우측 + 를 클릭하면, Libraries 파일명 입력창이 나옵니다. libherkulex.a 를 직접 입력합니다.
OK 버튼을 클릭합니다.



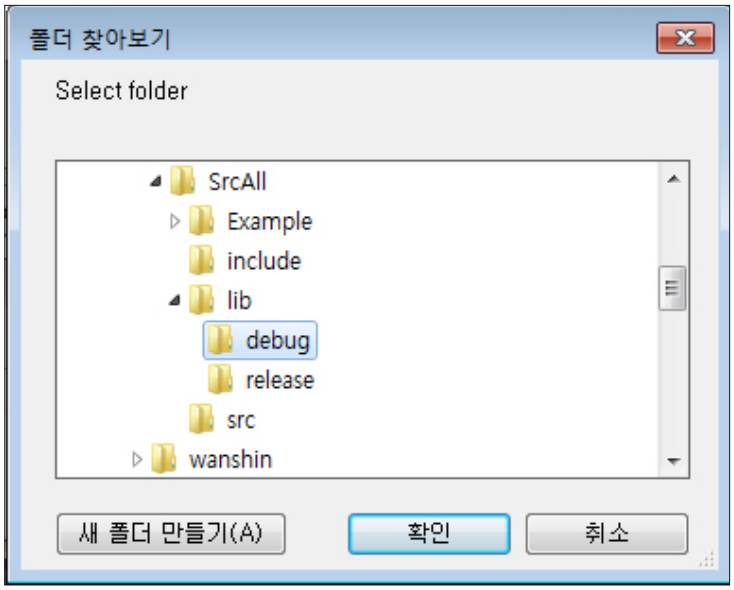
10 Search

서보모터 라이브러리 파일 libherkulex.a 링크가 완료되었습니다.
다음으로
Library search path 를 설정합니다.
우측 + 버튼을 클릭합니다.



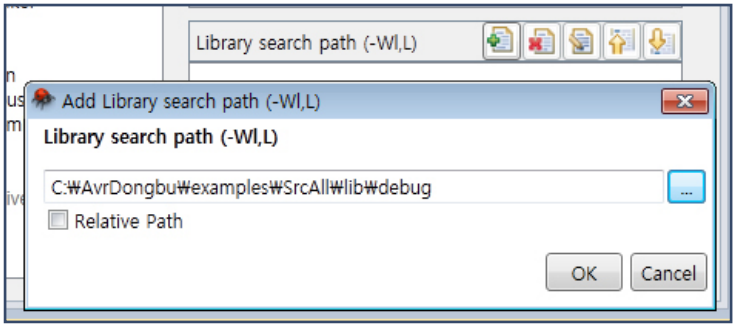
11 Path

Libray search path 팝업창 우측에 폴더 검색 버튼을 클릭합니다.



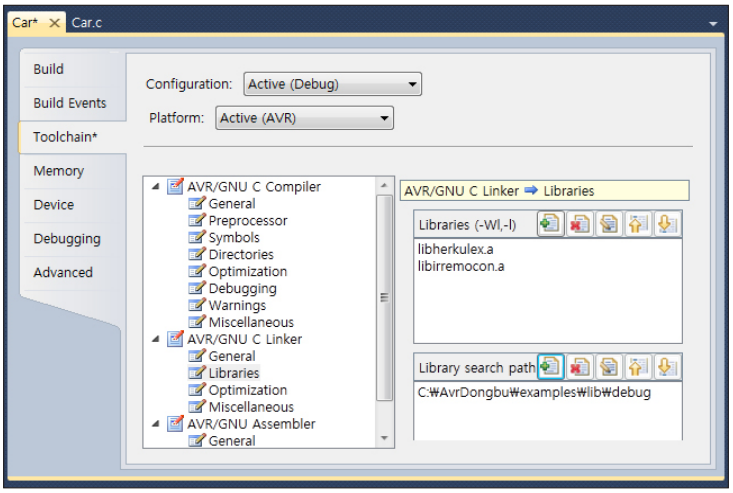
12 Debug

제공 라이브러리 폴더에서 lib 하위 폴더에 debug 를 선택합니다.



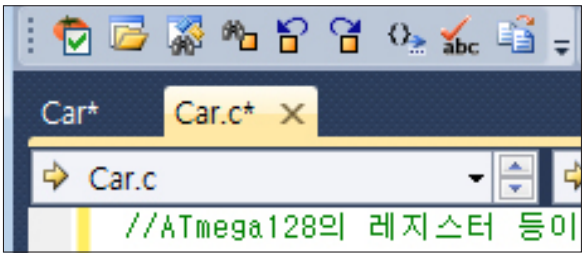
13 Path 완료

Relative Path(상대경로지정) 체크박스를 해지합니다.
Library search path 경로지정이 완료되었습니다.
OK 버튼을 클릭합니다.



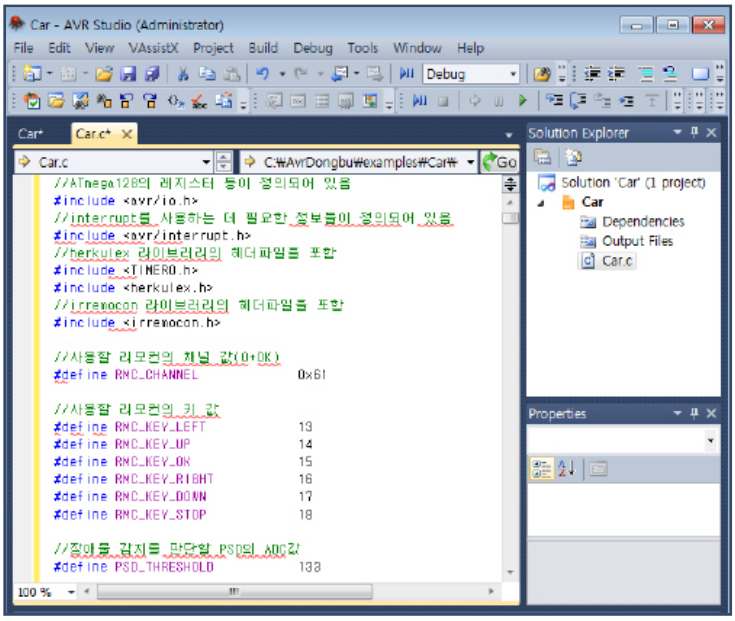
14 Link 완료

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;
```

MotionPlay.c

```
        //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;
}
```

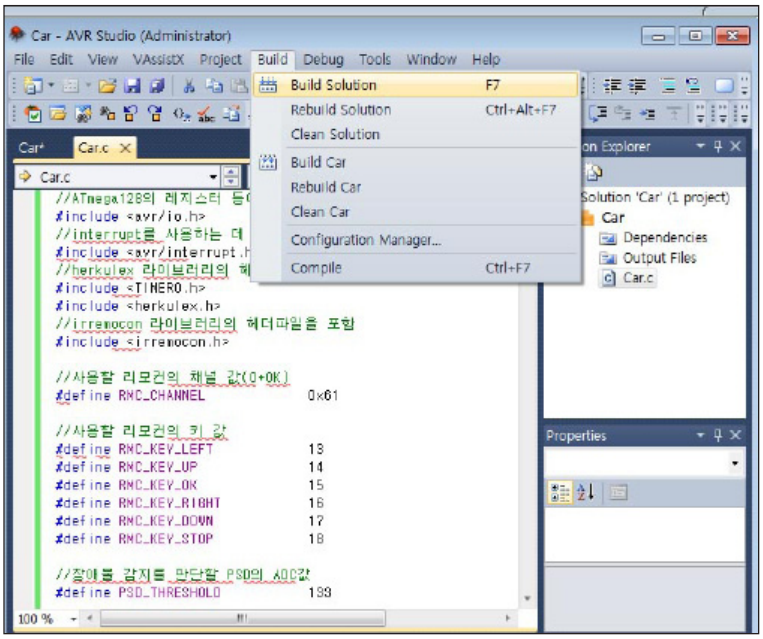
ucData의 값에 따라서 해당하는 동작을 수행하는 부분입니다. ucData가 RMC_KEY_UP과 같다면, 즉 위쪽 방향키가 눌렸다면 forward 함수를 호출해 트럭이 앞으로 가도록 하고 ucForward를 1로 바꿉니다. 이어지는 else if문도 리모컨 키와 그에 해당하는 동작을 나타냅니다. 아래쪽 방향키가 눌렸다면 backward 함수를 호출하고, 왼쪽 방향키가 눌렸다면 left_turn, 오른쪽 방향키가 눌렸다면 right_turn을 호출합니다. OK 키가 눌렸다면 load_off 함수를 호출해 짐을 내리는 동작을 취하게 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

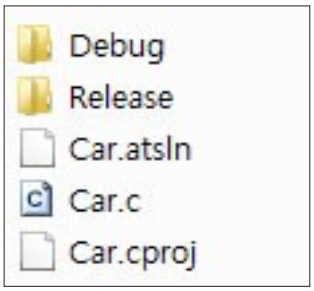
덤프트럭 로봇에 연결, 펌웨어를 다운로드 하고 실행하면 리모컨 입력에 따라서 덤프트럭 로봇을 제어할 수 있습니다.



01 Build

Build 메뉴를 선택하여 Build Solution을 클릭하여 Build 시킵니다.

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



02 Debug 폴더

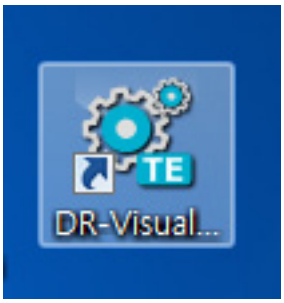
Build 후에 소스가 있는 폴더에 들어가면 Debug 폴더가 생성 되어있습니다. 클릭하여 폴더 안의 내용을 봅니다.



03 hex 확인

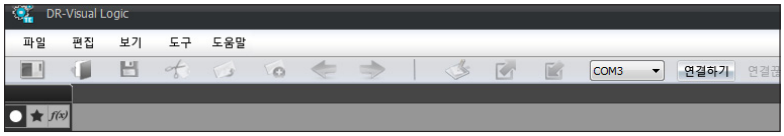
Debug 폴더 안의 hex 파일 확인합니다. Hex 파일이 제어기에 포팅시키는 파일입니다.

DR-Visual Logic 에서 펌웨어 복구를 실시합니다.



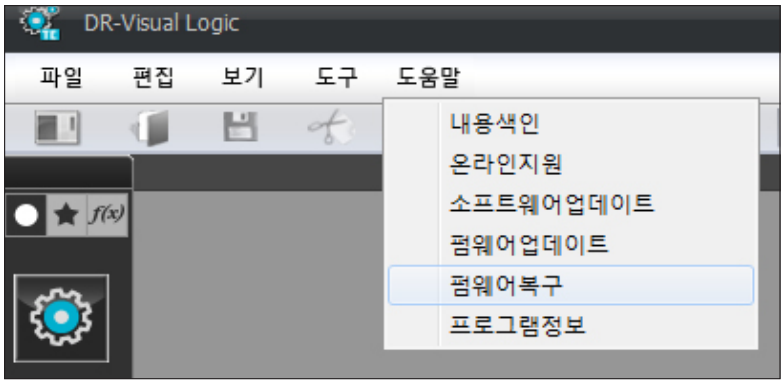
04 Visual Logic

DR-Visual Logic 을 실행시킵니다.



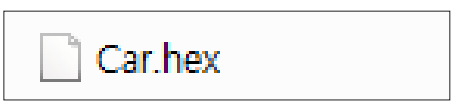
05 연결하기

COM 번호를 설정하고 연결하기를 클릭합니다.



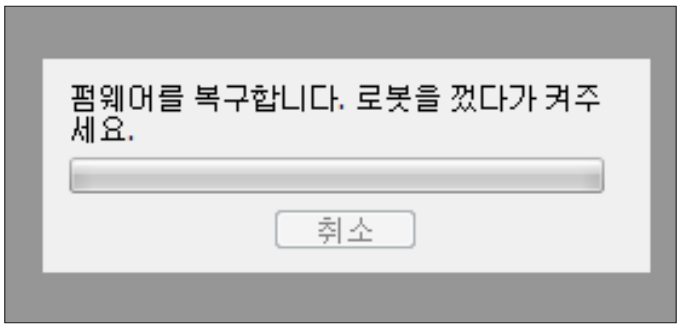
06 펌웨어 복구

도움말 메뉴에서 펌웨어복구를 클릭합니다.



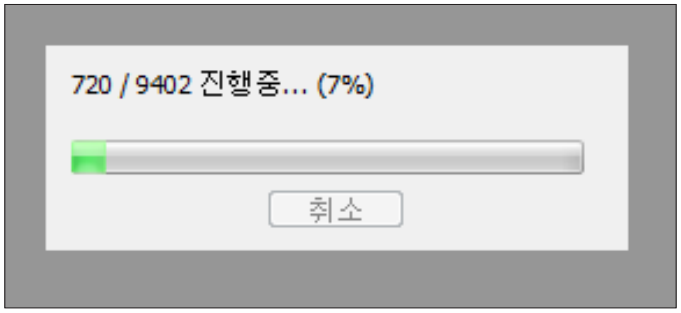
07 불러오기

Debug 폴더에서 Hex 파일을 불러옵니다



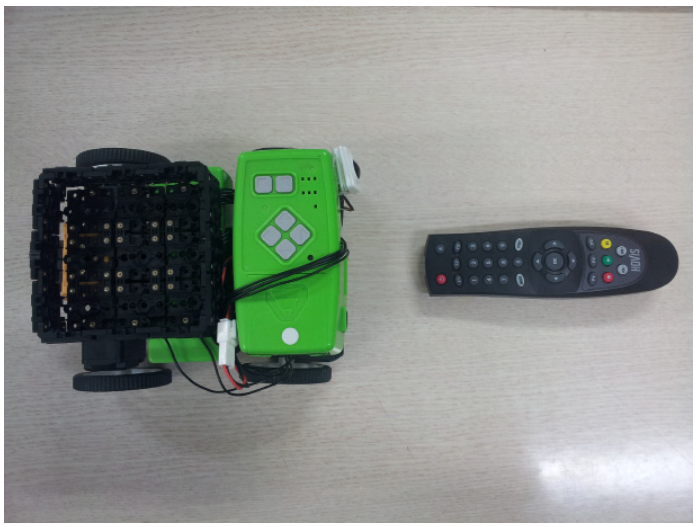
08 펌웨어복구

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



09 복구시작

펌웨어 복구가 시작됩니다.



10 실행

DRC를 덤프트럭 로봇에 연결, 펌웨어를 다운로드하고 실행하면 리모컨 입력에 따라서 덤프트럭 로봇을 제어할 수 있습니다.

로봇으로 배우는

AVR C Programming



ATmega128 마이크로프로세서가 적용된 로봇 제어기용 AVR C 프로그래밍의 진수

PART 01. HOVIS DRC와 ATmega128 익히기
PART 02. Visual Logic과 연계된 C언어 문법 익히기
PART 03. AVR 로봇 제어기 펌웨어 익히기
PART 04. 로봇에 응용하기

Dongbu Robot

(주) 동부로봇 www.dongburobot.com

본 사 : 경기도 부천시 원미구 약대동 193 부천테크노파크 401동 11층, TEL : 032-329-5551 (내선 103),
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

93550

ISBN 978-89-98873-04-2