ロボットで学ぶ

AVR C Programming



- ATmega128 マイクロコントローラ
- ロボットコントローラDRCファームウェア
  簡単に作成する

Dongbu Robot資料室

（マニュアルやアプリケーションを
ダウンロードしてください。）

www.dongburobot.com

Dongbu Robot

Dongbu Robot

ロボットで学ぶ
AVR C Programming

初版発行日：2012年10月05日
発行：(株)Dongbu Robot
住所：11th Floor, Bucheon Techno Park 401, Yakdae-dong, Wonmi-gu, Bucheon-city, Gyunggi-do 420-734, Korea
内容お問い合わせ ：+82-32-329-5551(ext.311)
FAX ： +82-32-329-5569

(株)Dongbu Robotで製作した教育制作物として本教材を事前の承認なしに
無断複写、複製、転載することを禁じます。

(株)Dongbu Robot All rights reserved

目次

PART 01 Hovis DRCとATMega128 学習

Chapter 01 Hovis DRCとセンサ―――― 12

PART 02 Visual Logic に関するC言語文法の習得

Chapter 01 C言語with Robot―――― 28
Chapter 02 C言語に目を通す―――― 29
Chapter 03 変数―――― 31
Chapter 04 配列、ポインタ、構造体―――― 41
Chapter 05 メモリ―――― 45
Chapter 06 構造体―――― 46
Chapter 07 繰り返しと分岐―――― 49
Chapter 08 関数―――― 57

PART 03 AVRロボット制御機Firmwareの習得

Chapter 00 AVR Studioのインストールと Hello Programming ---------- 66

Chapter 01 LED ---------- 84
AVRの原理とH/ Wの原理---------- 84
DRC＆H / Wの準備とAVR Studio＆S/ Wの実行---------- 85
例と説明---------- 88
Hex作成とDRCの実行 ---------- 90

Chapter 02 Button ---------- 93
AVRの原理とH/ Wの原理---------- 93
DRC＆H / Wの準備とAVR Studio＆S/ Wの実行---------- 94
例と説明---------- 97
Hex作成とDRCの実行 ---------- 101

Chapter 03 Buzzer ビーフ ---------- 104
AVRの原理とH/ Wの原理---------- 104
DRC＆H / Wの準備とAVR Studio＆S/ Wの実行---------- 274
例と説明---------- 278
Hex作成とDRCの実行 ---------- 281

Chapter 04 外部割り込み MIC ---------- 114
AVRの原理とH/ Wの原理---------- 114
DRC＆H / Wの準備とAVR Studio＆S/ Wの実行---------- 118
例と説明---------- 121
Hex作成とDRCの実行 ---------- 125

Chapter 05 Timer/Counter ---------- 128
AVRの原理とH/ Wの原理---------- 128
DRC＆H / Wの準備とAVR Studio＆S/ Wの実行---------- 135
例と説明---------- 138
Hex作成とDRCの実行 ---------- 142

Chapter 06 PSD (ADC) ---------- 145
AVRの原理とH/ Wの原理---------- 145
DRC＆H / Wの準備とAVR Studio＆S/ Wの実行---------- 151
例と説明---------- 154
Hex作成とDRCの実行 ---------- 157

Chapter 07 USART (Serial) ---------- 160
AVRの原理とH/ Wの原理---------- 160
DRC＆H / Wの準備とAVR Studio＆S/ Wの実行---------- 167
例と説明---------- 175
Hex作成とDRCの実行 ---------- 183

Chapter 08 Servo Motor ---------- 188
AVRの原理とH/ Wの原理---------- 188
DRC＆H / Wの準備とAVR Studio＆S/ Wの実行---------- 189
例と説明---------- 198
Hex作成とDRCの実行 ---------- 212

Chapter 09 Busser Advanced ---------- 216
AVRの原理とH/ Wの原理---------- 216
DRC＆H / Wの準備とAVR Studio＆S/ Wの実行---------- 220
例と説明---------- 222
Hex作成とDRCの実行 ---------- 228

Chapter 10 EEPROM ---------- 231
AVRの原理とH/ Wの原理---------- 231
DRC＆H / Wの準備とAVR Studio＆S/ Wの実行---------- 232
例と説明---------- 239
Hex作成とDRCの実行 ---------- 247

Chapter 11 加速度/ジャイロセンサ---------- 251
AVRの原理とH/ Wの原理---------- 251
DRC＆H / Wの準備とAVR Studio＆S/ Wの実行---------- 256
例と説明---------- 264
Hex作成とDRCの実行 ---------- 269

Chapter 12 IR リモコン---------- 272
AVRの原理とH/ Wの原理---------- 272
DRC＆H / Wの準備とAVR Studio＆S/ Wの実行---------- 274
例と説明---------- 282
Hex作成とDRCの実行 ---------- 289

PART 04 ロボットに応用する

Chapter 01

ロボットモーション制御 294
AVRの原理とH/ Wの原理 294
DRC＆H / Wの準備とAVR Studio＆S/ Wの実行 295
例と説明 303
Hex作成とDRCの実行 308

Chapter 02

ダンプトラックwithリモコン 318
AVRの原理とH/ Wの原理 318
DRC＆H / Wの準備とAVR Studio＆S/ Wの実行 319
例と説明 327
Hex作成とDRCの実行 350

おすすめの標準講義カリキュラム

1章：DRCの概要、Hovis概要、AVRの概要、AVRのインストール、Hello DRC AVR
2章：C言語 – 変数、制御文、関数、配列、ポインタ、構造体
3章～7章：AVR例 – LED、Button、Buzzer、加速度、MIC、Timer
8章：中間考査
9章～15章：AVR例–ヒューマノイドモーション、ダンプトラック、
PSD、EEPROM、USART、Motor、 ZigBee応用ロボット
16章：期末考査
別添：DRCレジスタ、有用な情報

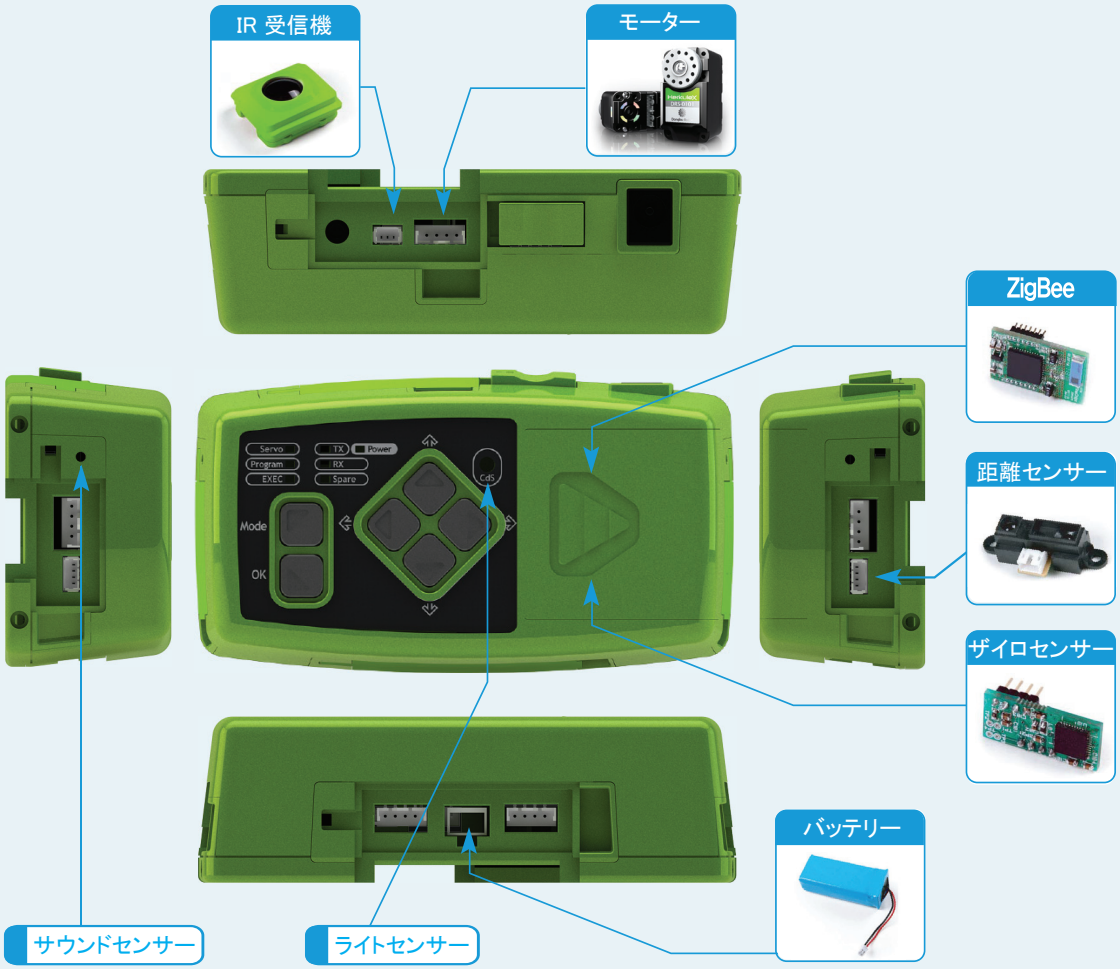
PART 01

Hovis DRCと ATMega128 学習

Hovis DRCとATMega128 学習

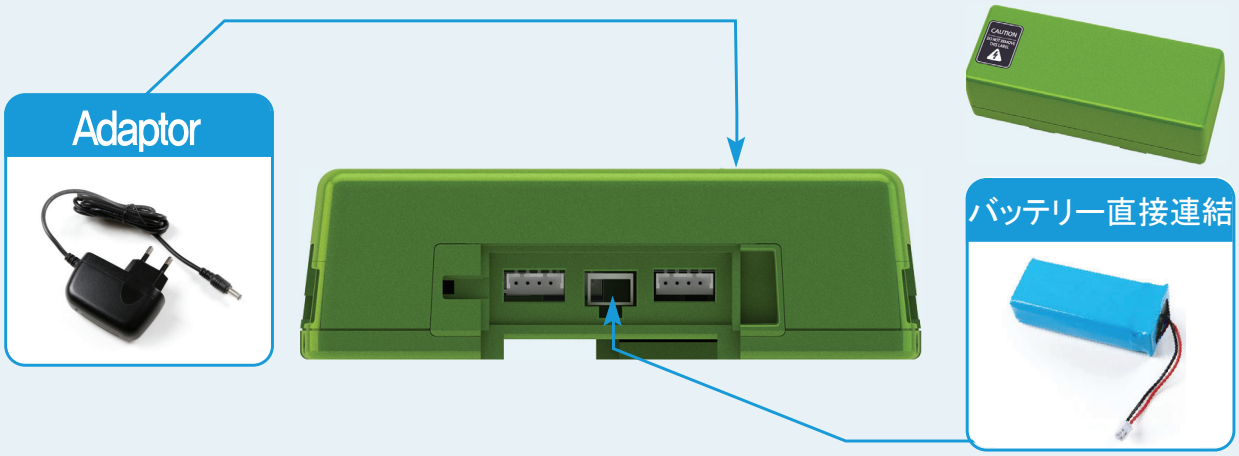
概要

コントローラーDRCはHovis　Liteの頭脳の役割をしているメイン部品です。コントローラーは6つのサーボモーターポート、2つのPSDセンサのポート、Gyroセンサ、ZigBeeを装着するためのコネクタなどのインタフェースで構成されています。また、光センサとサウンドセンサが組み込まれています。 DR-Visual Logicプログラムは本DRCコントローラーの機能を利用してプログラミングできるビジュアルロボットのプログラミング言語です。各種センサのテストやモーターを1～32個まで接続してプログラミングできます。



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バッテリーの取り付け方法



■ バッテリー

電源をDRCに供給するために、電池とコントローラーの下段の電源ポートに接続します。

■ 充電方法

アダプターとバッテリーケーブルを利用して直接充電できます。
もう一つの方法はコントローラーの上段にあるアダプタのコネクタを利用することができます。そのコネクタにアダプタを差し込むと、コントローラーと接続されたバッテリが自動充電されます。

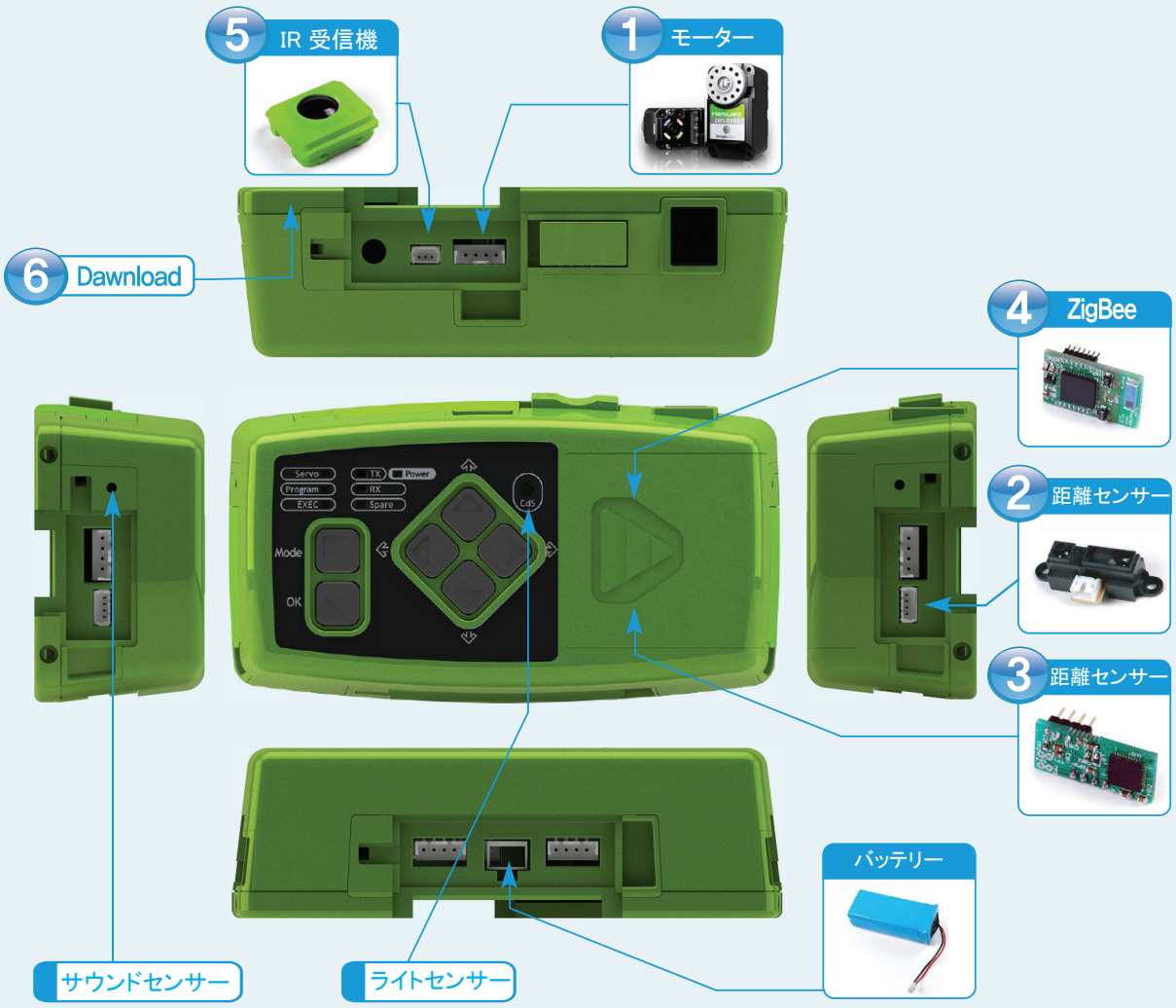


■ バッテリー

バッテリー残量が20%以下の時、PowerLEDライトが点滅します。

Hovis DRCとATMega128 学習

DRCの接続方法



1 モーターの接続
コントローラーの周辺に全5つのポートがあります。

2 距離センサの接続
左、右側に2つのポートがあります。

3 ジャイロセンサの接続
コントローラーのスロットカバーを開き、内部に装着します。

4 ZigBeeの接続
コントローラーのスロットカバーを開き、内部に装着します。

5 IR Receiverの接続
1つのポートがあります。リモコン受信用です。

6 ダウンロードリンク
Ear phone Jack でPCプログラムをロボットに
ダウンロードする際に使います。

DRCのインタフェース



DRCコントローラーはコントローラー正面にInput/Outputを示すボタンとLEDがあり、側面はモーターやセンサを接続するポートで構成されています。DRCのUserInterfaceは正面のボタンの操作でコントローラーにInput命令をすることができ、LEDでOutputのデータを表示できます。

	表記	ショートカットキー	基本Taskモード
Main Button	Mode	Task実行	
	Ok	確認ボタン	
Navi Key	(左)	バッテリーの残量表示	点検モード
	(上)	DRC自己診断	自律行動モード
	(右)	有線/無線の切り替え	リモコン操縦モード
	(下)	モーターのIDスキャン	サウンドデモモード
LED	Servo	HerkuleX 実行中	
	Program	DR-SIM/Visual Logic 実行中	
	EXEC	Task 実行中	
	TX	データ送信	
	RX	データ受信	
	Spare	ユーザーの割り当て	
	Power	電源表示	
Sensor	Cds	Light Sensor	

DRC Register Map & Protocol

■ レジスタ

DRCにはレジスタと呼ばれる領域があります。コントローラーの現在の状態、動作設定、センサに関連したいくつかのデータが入っています。例えば現在のコントローラーに接続しているサーボモーターの数、モーターのID、コントローラーのエラー状態、エラーコードなどはコントローラーの現在の状態に該当します。コントローラーが動作可能な最小/最大電圧、命令の応答設定はコントローラーの動作設定に該当します。また、光センサの現在の光量、サウンドセンサが感知した音の方向性などはセンサのデータに該当します。

コントローラーのレジスタは不揮発性(Non-Volatile、EEPROM)Registerと揮発性(Volatile、RAM)Registerに区別されます。不揮発性レジスタは電源がオフの状態でもデータは消去されない領域で、コントローラーの動作の基本設定値が保存されています。不揮発性レジスタの値は電源が入った瞬間に揮発性レジスタの対応する領域にコピーされます。揮発性レジスタはコントローラーの動作設定、コントローラーの状態、センサ値を保存しており、揮発性レジスタに保存した動作設定がコントローラーの動作に直接影響を与えます。

レジスタの各項目がどのように変化するかを知っていればDR-Visual Logicでロボットの行動をプログラムする時、さらに細かな動作条件を作成することができます。また、コントローラの状態を読み取ったり、設定を変えることでロボットを容易に使えるようになります。詳細についてはDRCレジスタとプロトコルの詳細なマニュアルを参照してください。

■ プロトコル

ユーザーがレジスタの値を読み取ったり、変更する場合はそれを指示する命令を送信する必要があります。その命令の決めているフォーマットをプロトコルと呼びます。プロトコルにはレジスタを読み書きする命令、コントローラーに保存されたモーションタスク、メロディなどを実行するコマンド、コントローラーを再起動するコマンドなど様々な命令が定義されています。

PCとコントローラーの通信時にはこのように定義されたプロトコルを通して要求パケットと応答パケットを送受信します。Dongbu Robotが提供するDR-SIMとDR-Visual Logicもこのプロトコルを利用して作られています。ユーザーはプロトコルを熟知することで、自分で作成したプログラムでDRCを制御することができます。

詳細についてはDRCレジスタとプロトコルの詳細なマニュアルを参照してください。

Hovis DRCとATMega128 学習

DRC機能の使い方

1 プログラムの定義

Firmware：ユーザーが変更できない内部プログラムです。

Task：Task Editor（DR-Visual Logic）を使用してユーザーが変更できるユーザー定義のプログラムです。

出荷時にはヒューマノイド専用の基本Task Programが定義されており、ユーザーが変更可能です。

2 動作方法

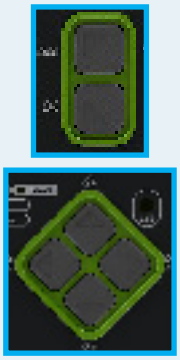
電源を入れNavi Keyを押すと、Firmwareに入っている基本機能が実行され、Modeボタンを押すと、保存されているTaskが実行されます。基本Taskの場合、Task実行後、再びNavi Key Okをクリックして、どのモードに入るか選択することができます。

Firmware（ユーザーが変更できません）

1 Navi クリック



2 Mode or Navi クリック

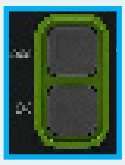


3 コントローラーの自己診断テスト



Task Mode（ユーザーが変更可能）

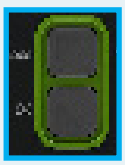
1 Mode クリック



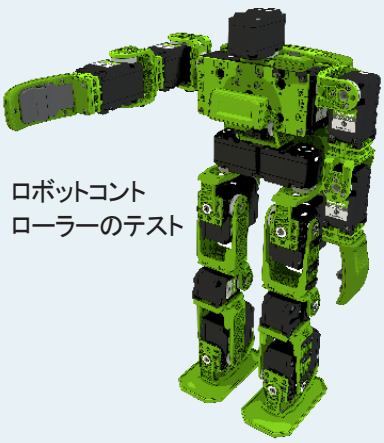
2 Navi クリック



3 Ok クリック



4 ロボットコントローラーのテスト



3 機能の使い方や動作

	表記	説明	機能や動作方法
Main Button	Mode	モード変更	Taskを始める 基本Taskの場合、Mode → Navi key → Okで動作モードを選択可能
	OK	確認ボタン	
Navi Key (Firmware)	(左)	バッテリーの残量表示	バッテリーの残量確認が可能→ 左LEDの3つの表示で最も低い時に1つ、中間の時に2つ、最も高い時に3つ
	(上)	テスト	コントローラーでモーターやセンサのテスト 使い方:(上)→ ボタン → センサによるモーターの反応でテスト テスト項目:モーター/光/音/距離/加速度/ジャイロセンサ
	(右)	有線/無線の切り替え	Ear phone jackを経由して有線通信/ ZigBeeによる無線通信の切り替え
	(下)	モーターのIDスキャン	現在接続しているモーターのIDを再スキャン
Navi Key (ヒューマノイド専用の基本Task)	(左)	点検モード	Mode → (左) → Ok : 点検モード:モーターを一つずつ選択し、そのIDのモーターが存在しない場合はビープ音を出す
	(上)	自律行動モード	Mode → (上) → Ok : 自律行動モード:ロボットが動きながら、自ら動作
	(右)	リモコン操縦モード	Mode → (右) → Ok : リモコン操縦モード:リモコンで定義されている動作を実行
	(下)	サウンドデモモード	Mode → (下) → Ok : サウンドデモモード:サウンド入力に応じて様々な動作をする
LED_mode	Servo	HerkuleX 実行中	HerkuleX Managerが実行中の時に点滅
	Program	DR-SIM/Visual Logic 実行中	DR-SIM/ Visual Logicを使用して、編集、使用中の時に点滅する。 データやファームウェアをダウンロードする時は、オンになる
	EXEC	モード変更/Task 実行中	Modeボタンを押して、Taskモードに入る時、Taskの実行中の場合オン
	TX	データ送信	外部へデータを送信する時、点滅する、task実行中の時はUser Spare空間
	RX	データ受信	外部からのデータを受信する時、点滅する、task実行中の時はUser Spare空間
	Spare	ユーザーの割り当て	
	LED点滅	エラー発生	実行中、エラーが生じた場合は右の3つのLEDが点滅しながら通知
LED_Power	Power	電源表示	残量が20%未満の時、点滅
Sensor	CdS	Light Sensor	光感知センサ

DRC基本テスト

DRCはロボットを組み立てなくてもテストモードを使用して基本テストが可能です。電源を入れ、コントローラーのショートカットキー(上)ボタンを押してモーターとセンサのテストを実行します。センサテストはモーターID1と2をそれぞれ左、右に挿入した状態でモーターの反応でテストを実行します。テスト項目はモーター、光センサ、音センサ、距離センサ、ジャイロセンサなどがあります。シナリオと操作方法の概要は以下の通りです。

光センサ、音センサが内蔵されている機能なので、menuとokでテストします。
PSD、Acc/ Gyroは(下)キーを押すと、まず、各センサが接続されているか一つずつチェックして、一つだけ接続されている場合は処理し、二つが同時に接続されている場合は処理されません。

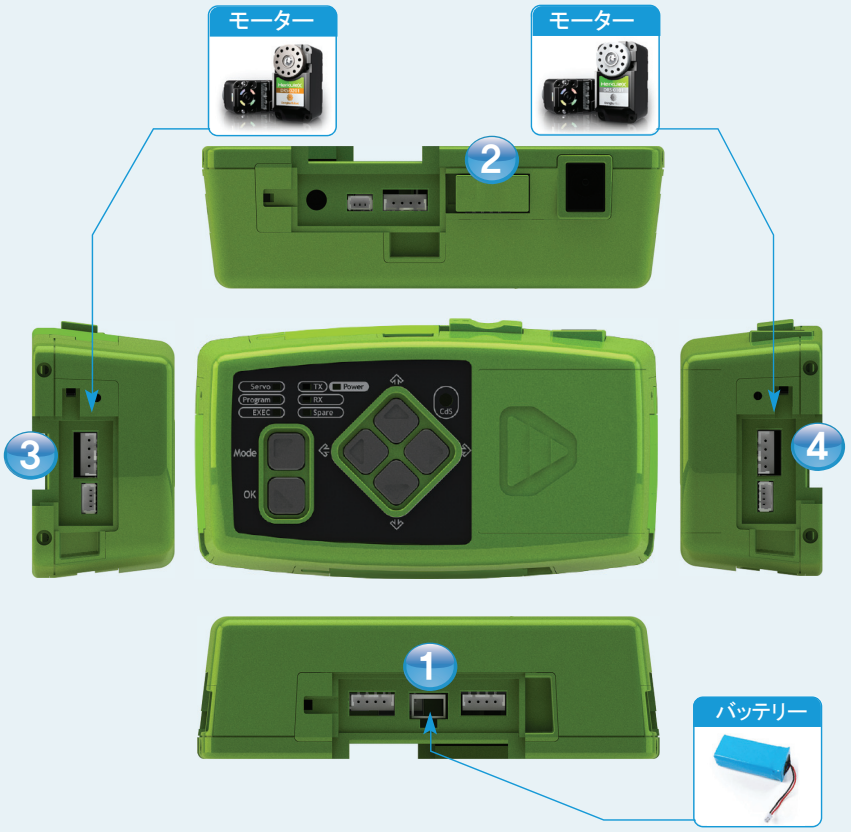
1. モーター :左、右に一つずつモーターを差し込む(左に1番、右に2番)
   (上) → (左) : 左側のモーターが繰り返し動く。(右) : 右のモーターが動く。(上) : モーターの二つが同時に動く。
2. 光センサ :(上) → (Menu) : Light機能が起動、Cdsウィンドウを遮ると、モーターの二つが同時に動く。
3. 音センサ :(上) → (OK) : Sound Sensor起動
   左側で拍手すると、左側のモーターが動き、
   右側で拍手すると、右側のモーターが動く。

以下の機能はセンサを接続後(下)ボタンを押すと動作します。テストするセンサを1台だけ接続します。2つ以上が接続されている場合は動作しません。

4. PSD Digital :(上) → デジタル距離センサの接続 → (下) PSD起動
   10cm以下の物体が接近すると → モーター二つが動き
   10cm以上の物体が遠くなれば(崖の検出) →モーター二つが停止
   → 崖の検出
5. PSD Analog : (上) →アナログPSDセンサの接続 → (下) PSD起動
   作動すると同時に二つのモーターが同じ方向に動く。物体との距離が遠いほど、より速く動いて、
   物体が近ければゆっくり動く → 物体が10 cm以内に接近するとモーターが逆に動き
   → 壁を検出し、避けるシナリオ
6. Acc :(上) → Acc/Gyro装着 → (下) Acc起動
   コントローラーのAccセンサがまっすぐに立った状態ではモーターが止まっている。コントローラーの傾きに応じてモーターは速度を変えて動く。コントローラーをさらに傾けると速く動く。
7. Gyro :(上) → Acc/Gyro 装着 → (下) Acc 起動 → (下) Gyro 起動
   コントローラーが停止している時には動かない。
   コントローラーが回る速度に合わせてモーターも一緒に動く。

次の詳細なテストページを参照して確実に実行してください。

DRC基本テスト：モーター



1 バッテリーの接続

2 電源を入れる

3 左モータの接続：必ずMotor ID1番を接続してください（他のモーターは動作しません。）

4 右モータの接続：必ずMotor ID2番を接続してください（他のモーターは動作しません。）

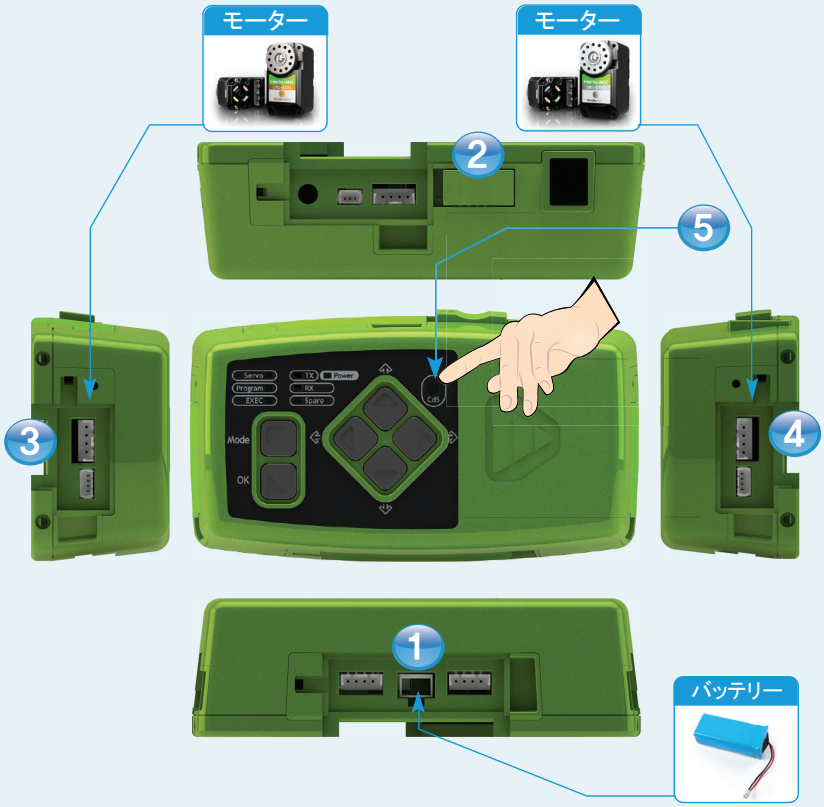
モーターは外に置いてテストします。車輪が回ることを想定したシナリオです。

→ テストの進行

- 電源を入れて（上）ボタンを押して、Test Mode切換後
- Navi Keyの（左）ボタンをクリックします。→ 左側のモーターが動作します。
- Navi Keyの（右）ボタンをクリックします。→ 右側のモーターが動作します。
- Navi Keyの（上）ボタンをクリックします。→ 二つのモーターが前進方向に動作します。

上と同じように動作した場合、モーターが異常なく動作したことを意味します。モーターテストを完了します。以降のすべてのテストの結果は上の二つのモーターが正常に動作するかどうかで判断します。モーターを分離せず、センサテストを続行します。

DRC基本テスト：光センサ



1. バッテリーの接続
2. 電源を入れる
3. **左モータの接続**：必ずMotor ID1番を接続してください（他のモーターは動作しません。）
4. **右モータの接続**：必ずMotor ID2番を接続してください（他のモーターは動作しません。）

光が消えた時、ロボットが腕で虚空を掴むの仮想シナリオです。

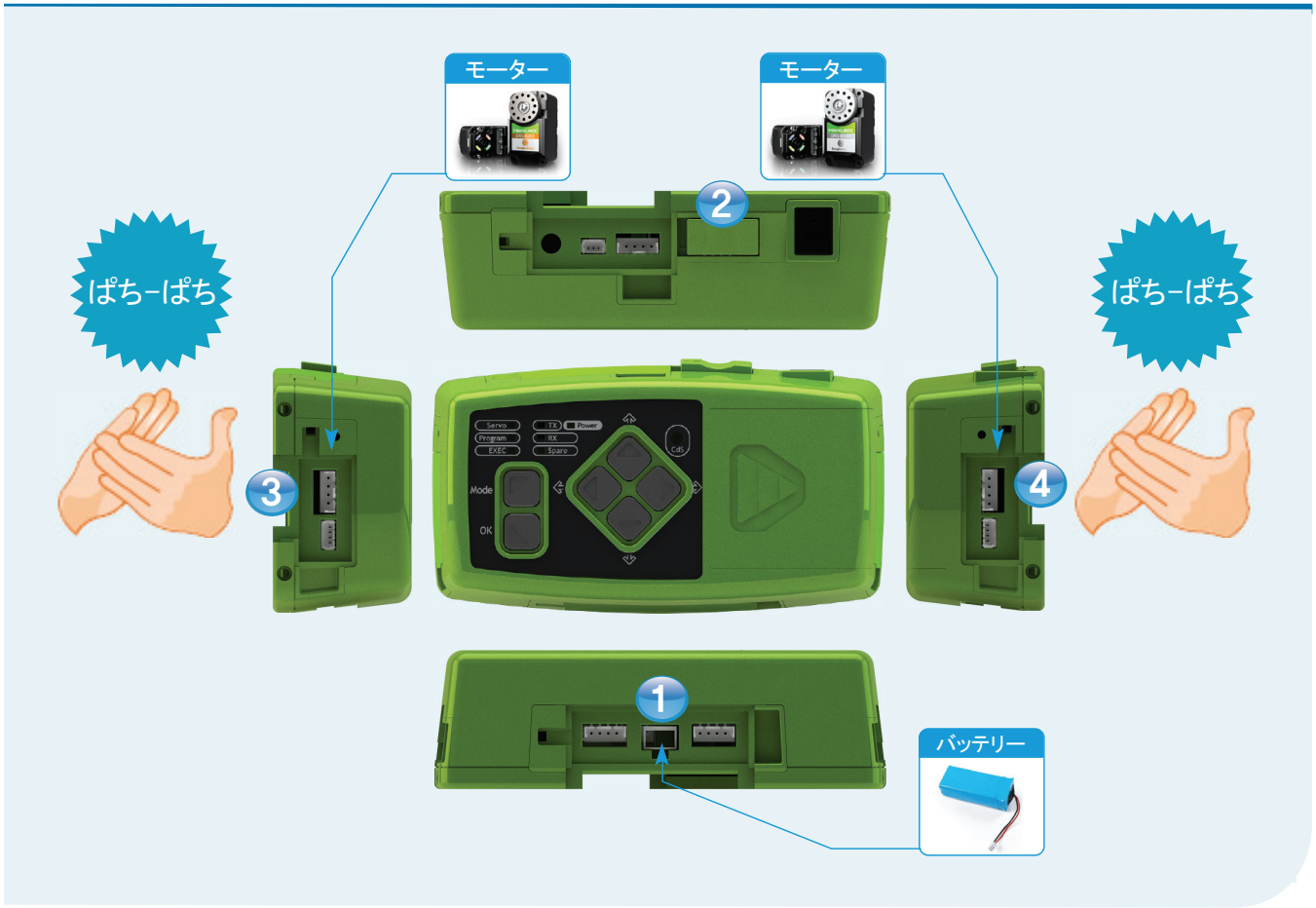
→ テストの進行

- 電源を入れて（上）ボタンを押して、Test Mode切換後
- Cdsウィンドウを手のひらで覆います。→左右のモーターが同時に動きます。

上と同じように動作した場合、光センサが異常なく動作したことを意味します。光センサのテストを完了します。

Hovis DRCとATMega128 学習

DRC基本テスト：音センサ



1 バッテリーの接続

2 電源を入れる

3 左モータの接続：必ずMotor ID1番を接続してください（他のモーターは動作しません。）

4 右モータの接続：必ずMotor ID2番を接続してください（他のモーターは動作しません。）

音センサはコントローラーの左右に設置されています。拍手する方向のモーターが動くシナリオです。

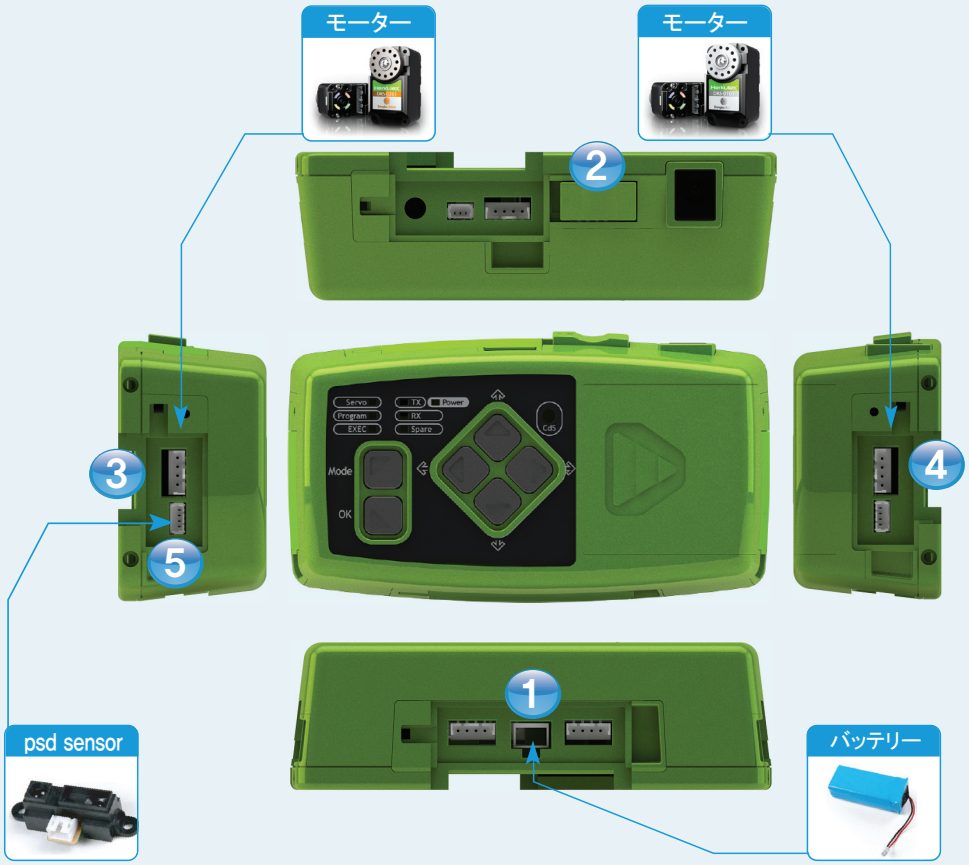
→ テストの進行

- 電源を入れて（上）ボタンを押して、Test Modeに切換後。
- （OK）ボタンをクリックします。→ 音センサが起動します。
- 左側で拍手をする。→ 左側のモーターが動きます。
- 右側で拍手をする。→ 右側のモーターが動きます。

上と同じように動作した場合、音センサが異常なく動作したことを意味します。音センサのテストを完了します。

Hovis DRCとATMega128 学習

DRC基本テスト : PSD デジタル距離センサ



1 バッテリーの接続

2 電源を入れる

3 左モータの接続 : 必ずMotor ID1番を接続してください(他のモーターは動作しません。)

4 右モータの接続 : 必ずMotor ID2番を接続してください(他のモーターは動作しません。)

5 PSDデジタルセンサの接続

PSDデジタルセンサは一定の距離を基準に距離を測定するセンサです。
一般的に地面の深さをチェックして崖を感知した時にロボットを停止させるシナリオです。

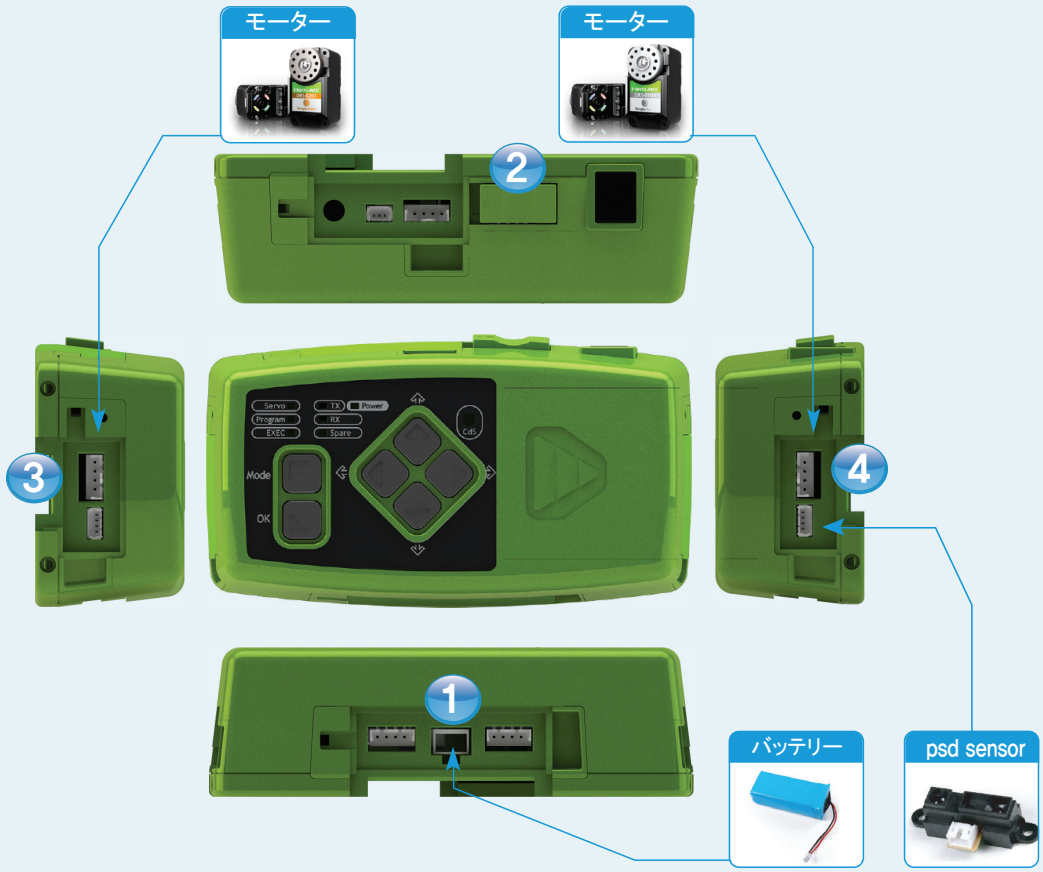
→ テストの進行

- 電源を入れて(上)ボタンを押して、Test Modeに切換後
- PSD線を接続して(下)ボタンをクリックします。→ PSDデジタルセンサが起動します。
- 手のひらや物体がセンサから10cm以下の場合→ 両方のモーターが動きます。
- 手のひらや物体がセンサから10cm以上の場合→ 両方のモーターが停止します。

＊PSDのデジタルセンサは基準の位置に基づいて、On/ Off形式でのみ動作します。

上と同じように動作した場合、PSDデジタルセンサが異常なく動作したことを意味します。PSDデジタルセンサのテストを完了します。

DRC基本テスト : PSD アナログ距離センサ



1 バッテリーの接続

2 電源を入れる

3 **左モータの接続** : 必ずMotor ID1番を接続してください(他のモーターは動作しません。)

4 **右モータの接続** : 必ずMotor ID2番を接続してください(他のモーターは動作しません。)

5 PSDのアナログセンサの接続

PSDのアナログセンサはリアルタイムで距離測定が可能です。接近距離別にモーターの速度を調整します。ほとんどの障害物の検出に使われ、障害物が近くに来た時に速度を落として、回避する動作に使用されます。

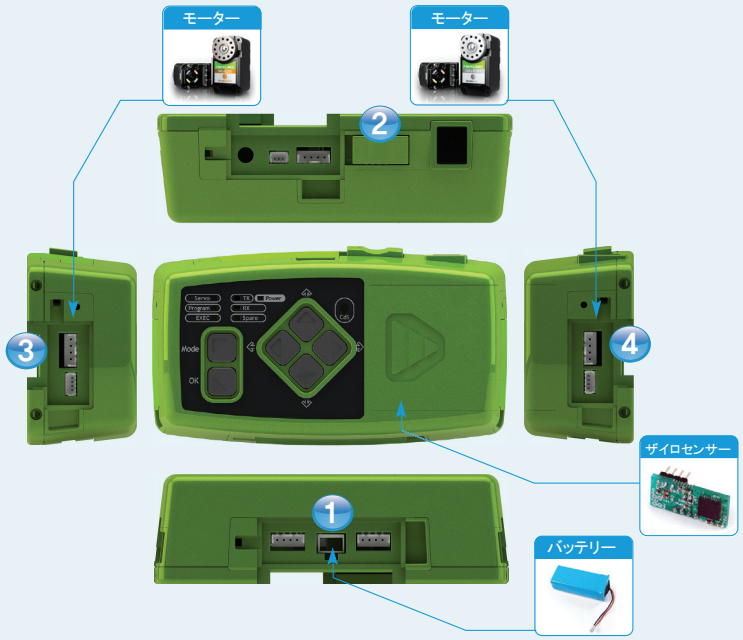
→ テストの進行

- 電源を入れて(上)ボタンを押して、Test Mode切換後、PSD線を接続して、(下)ボタンをクリックします。
  → PSDデジタルセンサが起動します。 → 二つのモーターが同じ方向に移動します。
- 手のひらや物体をPSDセンサの前に配置し、その後、遠くに移動させます。 → 物体が遠いほど、モーターは速く動きます。
  → 物体が近ければ近いほどモーターはゆっくり動きます。
- 物体が5cm以内に接近すると、モーターは逆に動きます。

上と同じように動作した場合、PSDアナログセンサが異常なく動作したことを意味します。PSDアナログセンサのテストを完了します。

Hovis DRCとATMega128 学習

DRC基本テスト：加速度/ジャイロセンサ



1 バッテリーの接続

2 電源を入れる

3 左モータの接続：必ずMotor ID1番を接続してください（他のモーターは動作しません。）

4 右モータの接続：必ずMotor ID2番を接続してください（他のモーターは動作しません。）

モーターは外に置いてテストします。車輪が回ることを想定したシナリオです。

5 Acc/ Gyroセンサの接続：ロボットコントローラーのふたを開け、加速度/ジャイロセンサを接続します。

→ テストの進行：ACC

- 電源を入れて（上）ボタンを押して、Test Mode切換後、Acc/Gyroを装着し、（下）ボタンを押します。
  → 加速度センサが起動します。
- コントローラーに搭載したAccセンサがまっすぐに立った状態であれば、モーターが停止します。
- コントローラーを徐々に傾けます → 傾きに応じた速度でモーターが動きます。さらに傾けると、速度が速くなります。

上と同じように動作した場合、Accセンサが異常なく動作したことを意味します。

→ テストの進行：Gyro

- ACCのテストモードの状態でもう一回（下）ボタンを押します。→ ジャイロセンサが起動します。
- コントローラーが止まっている時はモーターも停止します。
- コントローラーを動かします。→ コントローラーが回る速度に合わせてモーターの速度が変わります。

上と同じように動作した場合、加速度/ジャイロセンサが異常なく動作したことを意味します。 加速度/ジャイロセンサのテストを完了します。AccとGyroは1つのチップにモジュール化されています。

PART 02

Visual Logicに関する C言語文法の習得

PART 02

Visual Logicに関するC言語文法の習得 HOVIS

CH.1 : C言語with Robot

◆ロボットでC言語を学ぶということ

ロボットは動くコンピュータです。そのためロボットは頭脳の役割をするCPUを持っています。そのCPUはMCUと呼ばれ、MCUは性能に応じ8bit、16bit、32bitに分けられます。その中のロボットコントローラによく使われるMCUが8bit ATmel社のATmega128です。
Dongbu RobotヒューマノイドのコントローラのDRCはATmega128チップを使用した制御機です。

私たちは言語、つまりプログラミング言語を初めて学ぶ時に、C言語から習得します。しかし、まずロボットのメカニズムを習得してから、C言語を使用する必要があるので、最もよく使われるVisual C + +に比べて難しいです。

しかし、ロボットでC言語を習得するということは、PCアプリケーションの開発に使用されるMFCより難易度があり、高級なプログラミングができるということを意味し、ロボットにより興味を誘発できるという点で、大きな特徴があります。それはFirmwareを学ぶことができるからです。
ハードウェア設計が完了すると、Firmwareプログラミングをする必要があります。 Firmwareがないと、アプリケーションと他のアプリケーションを作成できません。そのためFirmwareエンジニアは一般のアプリケーションの開発者より待遇が良いです。もっと難易度の高い開発を行うことができるからです。
実務では、ファームウェアとアプリケーション開発者は厳密に区分されています。新入社員の時に一度決めれば一生の仕事になる可能性があります。そのため慎重な選択が必要であり、Dongbu Robotと共に行うC言語教育は良い指針となるでしょう。

ロボットで、C言語を学ぶということは、ハードウェアを知り、Firmwareを習得することであり、多くのエンジニア的な知識が必要です。C言語が難しいのではなく、ハードウェアとファームウェアが難しいのです。
そのため、ロボットで学ぶC言語にはもっと包括的な問題が内包されます。
Dongbu Robotは後にロボットを利用した入門者用のC言語の本を編纂する予定です。簡単な、Visual C + +より実用的で効果的な本になると思います。

今回のC言語の文法は要約型で進行する予定であり、例示は必要に応じてFirmwareとVisual C+ +の両方を使用します。C言語は、他のC言語入門書を見る方がいいとの判断で、ここではC文法は図表や総合的なデータを使用します。
また、DR-Visual Logicの例と一緒にどのようにC文法と連携されているのかを説明します。

CH.2 : C言語に目を通す

◆ C言語の例

※ Dongbu AVRの例

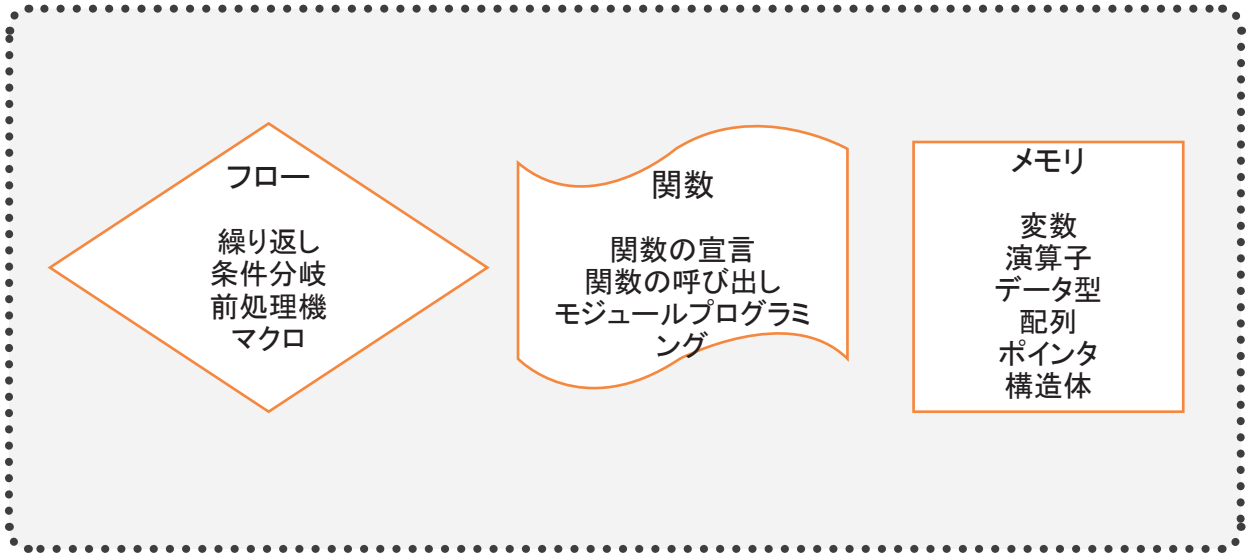
```
#include <avr/io.h>  //(1) 前処理機
#define BTN_MODE          0b00000001
int main(void)  // (2) 関数 - 機能の定義
{
        unsigned char ucButton;   // (3)変数 - メモリ管理 - ポインタ
PORTB &= ~0b10000000;

        while(1) //(4) 繰り返し
        {
                ucButton = (~PINA) & BTN_MASK;
                if(ucButton & BTN_OK){          //(5)条件分岐
                        PORTB ^= 0b10000000;
                        _delay_ms(1);
                }
        }
        return 1;
}
```

上記の例は、Dongbu　Robot　AVRのソース例の一部を抜粋したもので、Cプログラミングの文法の構成概要を示しています。

C言語には、大きく分けて二つの要素があります。それは変数と関数です。

(3)変数は、データを保存する目的でメモリを管理し、そのメモリには、それぞれアドレスが存在します。そのアドレスを保存するものをポインタと呼びます。ポインタも変数の一種です。一般的なC言語は、ポインタを難解な言語で規定しています。
(2)関数は、動作させる機能を持っています。プログラミングは関数の組み合わせです。
プログラミングをする理由は、コンピュータが人より計算の能力が優れているからです。この能力を最も有効に活用するために、制御や分岐や繰り返しを使います。
(4)繰り返しと(5)条件分岐はC言語だけでなく、すべてのプログラミング言語の文法がほぼ同じです。この二つの要素は、プログラミングの便利さを示す良い例です。

◆C言語の概要



Cプログラミング言語は、プログラムの流れを制御するフローとメモリにデータを管理するメモリのパートと機能を付与する関数で構成されます。

プログラムの作成と実行の手順は以下のとおりです。

1)プログラムを作成します。

2)コンパイルします。

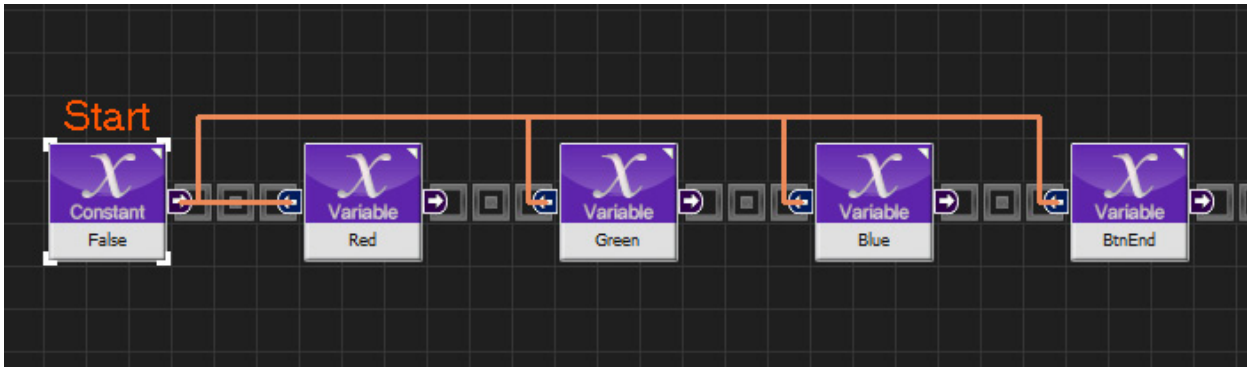
3)リンク:接続します。

4)実行ファイルを生成します。

CH.3：変数

◆ 変数

例) ledbutton.dtsの一部



C-like 見る

```
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                 }
```

※ 変数とは、データを保存できるメモリ領域の名前を意味します。

変更は可能です。ただし、定数は変更できません。

※ 様々な形（データ型）の変数は、メモリ領域の使い方に応じて以下のように簡単に分類されます。

整数型：char、int、long、3

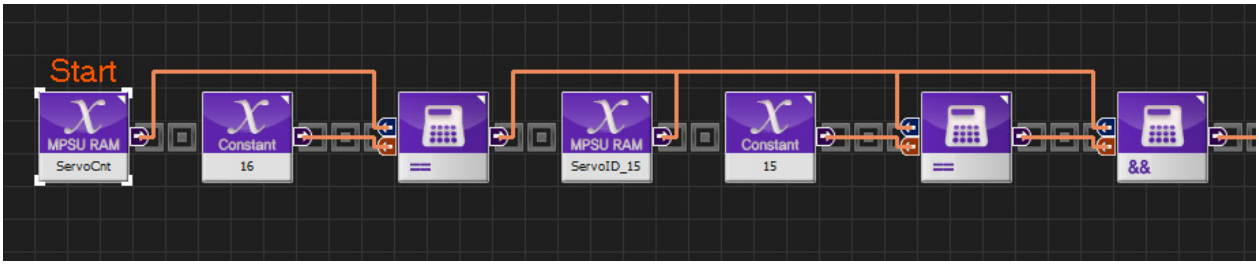
実数型：float、double3.1

※ 変数の宣言と代入例です。

```
Int main(void)
{
        Int val;    //int型の変数valの宣言
                    // → int型の変数にvalという名前を与え、メモリ領域を割り当て
        Val=20;//変数valに20を保存
```

変数

例）ledbutton.dtsの一部



C-like 見る

```
 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
```

◆リテラル定数の基本データ型

定数はメモリ領域に保存するためにデータ型を決定します。

Char c=’A’/文字定数の文字を書く場合char型として認識し、メモリに入れます。

Int I=5//整数の定数

Double d=3.15//実数の定数

左は変数、右は定数、二つは独立した関係です。

◆シンボリック（symbolic）定数

名前を持つ定数です。シンボリック定数を定義する方法は次のとおりです。

constキーワードを使用した変数の定数化

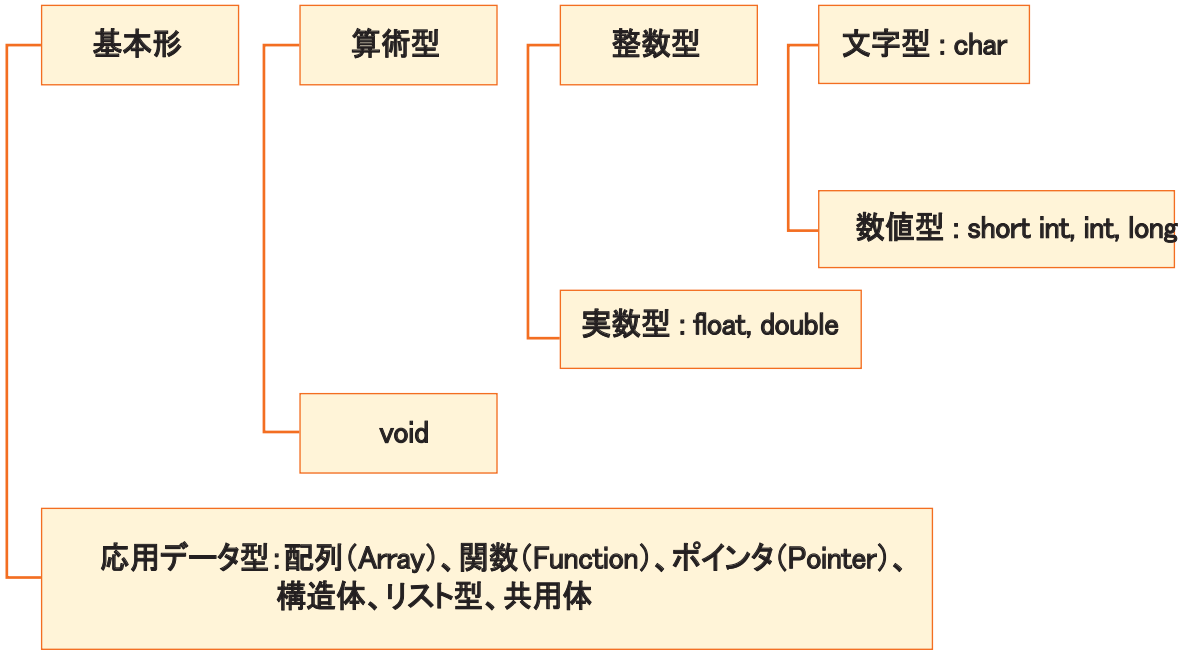
const int MAX=100//変数が定数となる

MAX=102//使用不可、MAXは定数であるため宣言をする時に大文字で書くのが慣例です。

◆接尾辞に応じた様々な定数の表現

接尾辞	データ型	使用例
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

◆ データ型の種類



◆ データ型範囲

区分	データ型	範囲	byte
文字型	char	-128~127	1
	unsigned char	0~255	
整数型	int	-2147483648 ~ 2147483637	4
	unsigned int	0 ~ 4294967295	
	short	-32768 ~ 32767	2
	unsigned short	0 ~ 65535	
	long	-2147483648 ~ 2147483637	4
	unsigned long	0 ~ 4294967295	
実数型	float	±3.4 x 10-38 ~ ±3.4 x 1038	4
	double	±1.7 x 10-308 ~ ±1.7 x 10308	8

◆拡張文字列

表記	意味	機能
¥n	New line	改行する。
¥t	Tab	次のタブ位置に移す。
¥b	Back space	1文字分戻る。
¥r	Carriage return	同じ行の先頭に戻る
¥a	Alert	警報音を出す
¥”	” 表示出力	二重引用符を出力
¥’	’ 表示出力	単一引用符を出力

◆ printf 変換文字列

区分	変換文字列	出力のタイプ
整数型	%d	符号付き10進数
	%u	符号なし10進数
	%o	符号なし8進数
	%x, %X	符号なし16進数
実数型	%lf, %f	符号付き小数点(double、float)
	%le, %e	符号付き指数型(double、float)
文字型	%c	一つの文字
文字列	%s	文字列

◆ printf 変換文字列のオプション

修飾子	意味	
数字	出力桁数の確保	
	%10d	画面に10桁を確保し、右揃え
	%-10d	画面に10桁を確保し、左揃え
	%10.2lf	画面に10桁を確保し、小数点以下2桁まで出力
h	Short型を意味例）%hu	
L , l	Long型を意味例）%ld	

◆ Scanf 変換文字列

区分	変換文字列	出力のタイプ
整数型	%d, %u	10進数
	%o	8進数
	%x	16進数
実数	%lf, %f	小数点または指数型
	%le, %e	符号付き指数型（double、float）
文字型	%c	一つの文字（char型の変数）
文字列	%s	文字列（char配列）

◆ C言語のキーワードと説明1

1.asm：インラインアセンブリコードを表すキーワード
2.auto：基本的な変数の保存方法を示すキーワード
3.break：for、while、switch、do... while文を条件なしに終える命令
4.case：switch文の中で使用される命令
5.char：最も簡単なデータ型
6.const：変数が変更されないように防止するデータ指定子
7.continue：for、while、switch、do... while文を次の繰り返し動作に進行させる命令
8.default：case文に一致しない場合を処理するためにswitch文で使用される命令
9.do：while文で一緒に使用される循環命令。ループ文は、少なくとも1回は実行される。
10.double：倍精度浮動小数点型の値を格納できるデータ型
11.else：if文FALSEに評価されるときに実行される選択的な文章を表す命令
12.enum：変数が特定の値のみを受け入れるようにするデータ型
13.extern：変数がプログラムの他の部分で宣言されることを知らせるデータ指定子
14.float：浮動小数点型数値を格納するために使用されるデータ型
15.for：初期化、増加、条件部分を持つ循環命令

◆ C言語のキーワードと説明2

16.goto：定義されているラベルに移動させる命令
17.if：TRUE/ FALSEの結果に基づいて、プログラムの制御を変更するために使用される命令
18.int：整数型の値を保存するために使用されるデータ型
19.long：int型よりも大きな整数型の値を保存するために使用されるデータ型
20.register：可能であれば、変数をレジスタに保存するように指定する保存型指定子
21.return：現在の関数を終えて呼び出した関数にプログラムの制御を返す命令。
関数で値を返すために使用される。
22.short：整数型の値を保存するために使用されるデータ型。ほとんどの
コンピュータでint型と同じサイズを持つ。
23.signed：変数が正数値と負数値をすべて保存できるということを指定するために使用される指定子
24.sizeof：項目のサイズをバイト単位で示す演算子
25.static：コンパイラが変数の値を保存するということを指定するために使用される指定子
26.struct：Cでデータ型の変数を一緒に結合させるために使用されるキーワード
27.switch：複数の条件を通したプログラムの流れを変更するために使用される命令。
case文と一緒に使用される。
28.typedef：既に存在する変数と関数の形を新しい名前に変更するために使用される指定子
29.union：複数の変数が同じメモリ領域を共有するようにしてくれるキーワード
30.unsigned：変数が正数値だけで保存できることを指定するために使用される。
31.void：関数が値を返さない場合、または使用されるポインタが汎用ポインタまたはすべての
データ型を指摘することができるという値を指定するために使用されるキーワード
32.volatile：変数が変更できないことを指定する。
33.while：指定された条件がTRUEに評価される限り継続して含まれている文を実行するループ文

◆文字型

文字表現のためにASCIIコードが出ます。

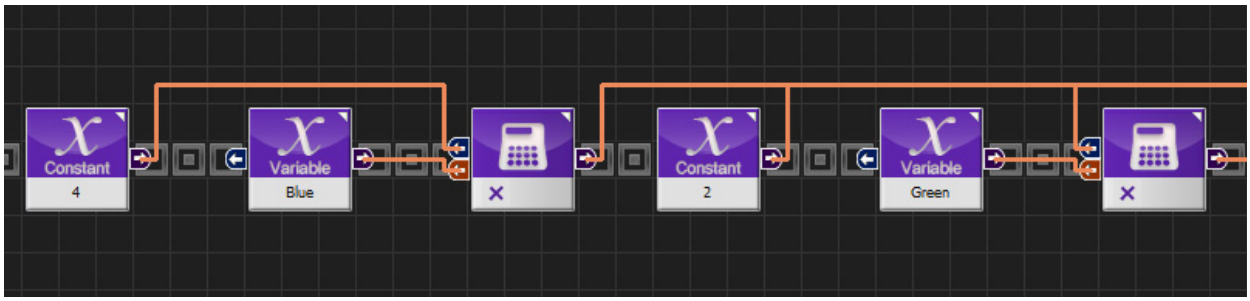
米国規格協会（ANSI）によって定義され、コンピュータを介して文字を表現するための標準として、文字と数字の接続関係を定義しました。文字Aは数字65、文字Bは数字66でマッチングさせます。コンピュータは2進数しかなかったため、コンピュータと人の間に約束を設けました。人は文字での表現を望んでいて、文字と数字の間のマッピング関係の維持を望んでいます。たとえば、A=65、B=55と約束し、人がAを入力すると、コンピュータは65と保存します。

ANSIで定義された数字と文字のマッピングは以下の通りです。

10進	16進	文字	10進	16進	文字	10進	16進	文字	10進	16進	文字	10進	16進	文字
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	空白	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			

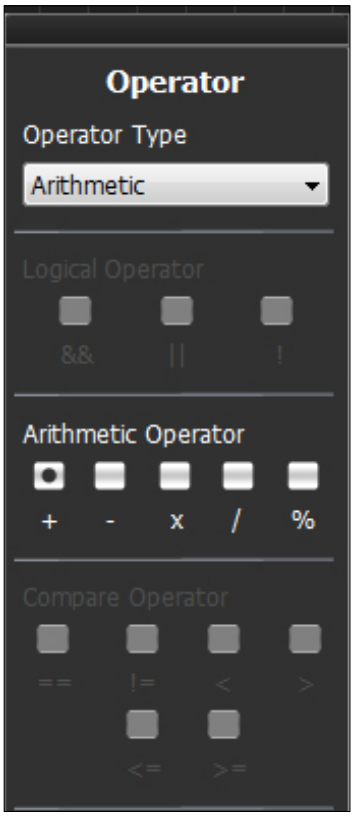
◆演算子

例）ledbutton.dtsの一部



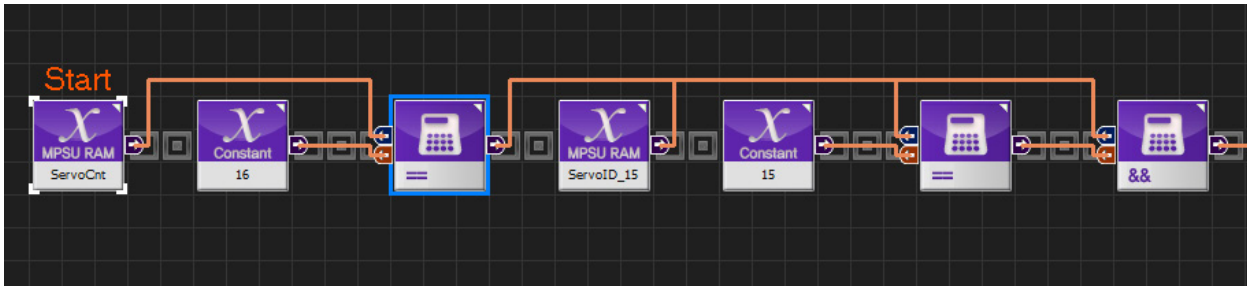
C-like 見る

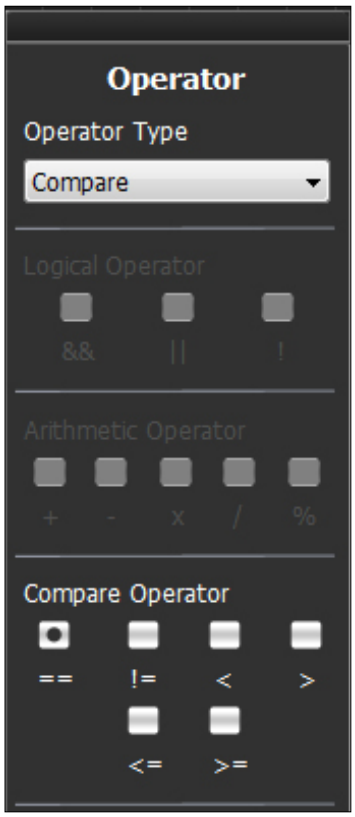
```
33  led( ( ( ( 4 * Blue ) + ( 2 * Green ) ) + Red ) )
34  if( ( ( MPSU_ButtonStat == 0x00 ) && BtnEnd ) )
35  {
36          BtnEnd=false
37  }
38  else
39  {
40  }
```

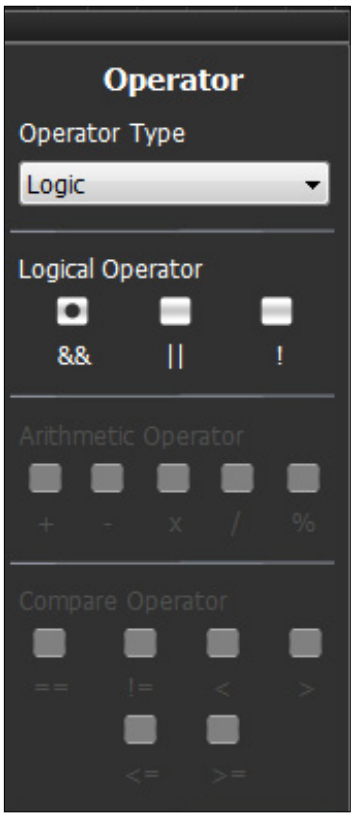


機能別種類	演算子
算術演算子	+ − * / %
符号演算子	+ −
代入演算子	= 複合代入演算子
関係演算子	== != <= < >= >
増減演算子	++ −−
ポインタ演算子	* & []
構造体の演算子	. ->
論理演算子	|| && !
ビット演算子	| & ~ >> <<
三項条件演算子	? :
コンマ演算子	,
sizeof演算子	sizeof
キャスト演算子	(type) type()
括弧演算子	()
C++ 演算子	new delete :: .* ->*

例) Remocon16.dtsの一部







C-like 見る

```
 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 )
```

◆演算子の優先順位

順位	名称	演算子の種類	結合方向
1	1次演算子	() {} . ->	→
2	単項演算子	+ - ! ~ (type) sizeof ++ -- & *	←
3	乗算、除算、剰余算	* / %	→
4	加減	+ -	
5	Shift演算子	<< >>	
6	関係演算子	< > <= >=	
7	等価演算子	== !=	
8	ビット積	&	
9	ビット差	^	
10	ビット和	|	
11	論理積	&&	
12	論理和	||	
13	条件演算子	? :	→
14	代入演算子	= += -= *= /= %= <<= >>=&= |=	←
15	順次演算子	,	→

◆ 記憶クラス



データ領域

ヒープ(Heap)領域
ランタイムにメモリの割り当て

スタック領域
コンパイルタイムにメモリの割り当て

◆メモリ領域

指定子		有効範囲	生存期間	メモリ位置	自動初期化
auto		宣言されたブロック内	ブロック終了時	Stack	X
extern		一つのモジュール全体のプログラム	プログラム終了時	(非)初期化領域	O
static	内部static	宣言されたブロック内	プログラム終了時	(非)初期化領域	O
	外部static	宣言されたモジュール内			
register			ブロック終了時	CPU内のレジスタ	X

CH.4 : 配列、ポインタ、構造体

DR-Visual LogicはC言語で扱う配列とポインタと構造体については説明しません。グラフィックロジックを構成するために、複雑な文法を最小限にするためです。

そのため、このパートでは、Visual Logicと対応せず、配列、ポインタ、構造体を簡単に説明します。

◆ 1次元配列

※ 配列とは

- 2つ以上の変数を同時に宣言する効果があります。
- 大量のデータを一括して処理する必要がある場合に便利です。
- 地域の特性をもつこともあり、グローバル属性をもつこともあります。

※ 配列の宣言

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

配列要素のデータ型　配列名、配列の長さ

➔ Int型のデータを10個保存できるメモリ空間を割り当てて、名前をarrayとします。

- 配列の長さ:アレイを構成する変数の数(必ず定数を使用)
- 配列要素のデータ型:配列を構成する変数のデータ型
- 配列名:配列にアクセスするときに使用される名前

◆ 2次元配列

※ 多次元配列の例

Int arr[100]1次元配列

Int arr[10][10]10x10、2次元配列

Int arr[5][5][5]5x5x5、3次元配列

1次元は線、2次元は面、3次元は直六面体です。

通常、多次元配列は、2次元配列を意味します。

※ 多次元配列の実際のメモリ構成

1次元配列と同じですが、アプローチを2次元的に解析するだけです。

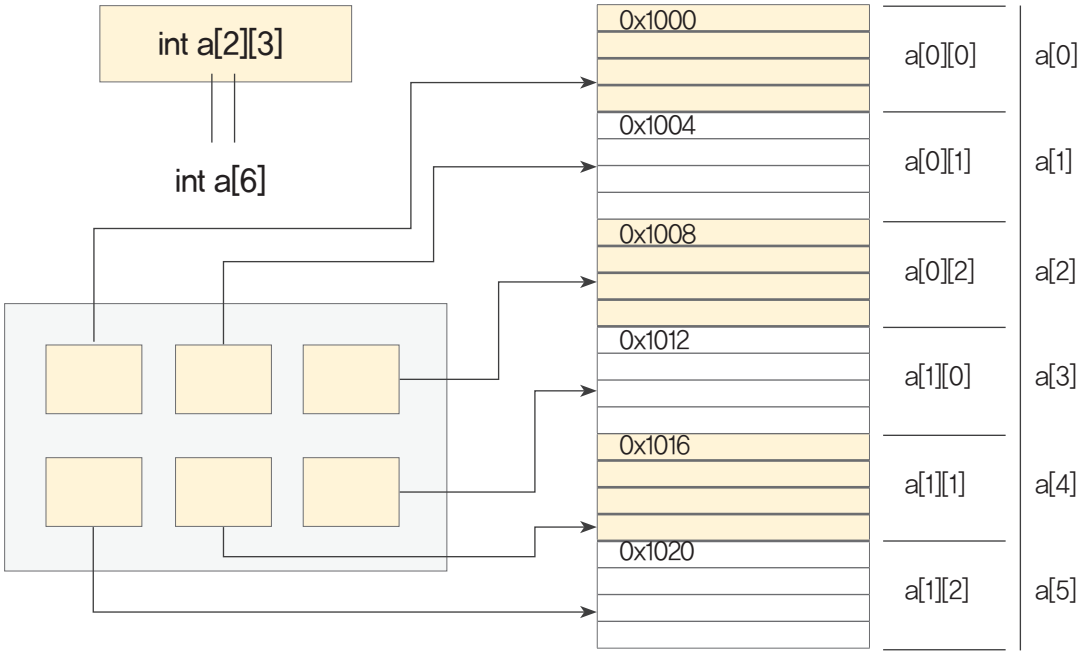
Int a[2][3];2次元のアプローチa[0][0]。縦が2、横が3

Int a[6];1次元のアクセスa [0]～a[5]は必ず2次元的に解析してください。

問題自体が2次元的な場合があります。

解決する問題は、2次元であるが、実際の構造は1次元的ではあるが、2次元的にする習慣が必要です。

◆多次元配列の実際のメモリ構成



※ 2次元配列の宣言と同時に初期化

中括弧の中に別の中括弧→2次元以上の初期化をする時に使用します。

case1 : 行単位ですべての要素を初期化

case 2 : 行単位で一部の要素だけを初期化

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

最初の中括弧 →最初の行の初期化確認、

int somang[3][3]={{1}、{4,5}、{7,8,9}};

行の一部だけを初期化→最初の行の初期化要素のリストは1つだけ、左から入れて、残りは0にします。

Case 3 : 内部中括弧なしの初期化

int somang[3][3]={1,2,3、4,5,6、7};

最初から3個は1行目の初期化、次の3個は2行目の初期化、残りの1個は3行目を初期化。

◆ポインタ

※ ポインタとは?

- ポインタとポインタ変数

  ポインタとは、変数を意味します。ポインタ変数と同じ概念です。

- メモリのアドレス値を保存するための変数です。通常の変数はデータを保存するためのものですが、

  データではなく、アドレスの値を保存します。

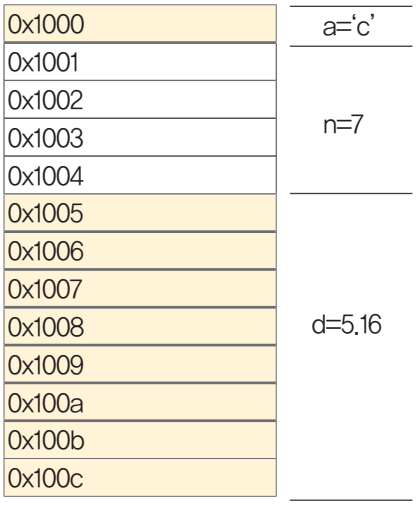
- “ポインタ”を一般的に”ポインタ変数”と呼びます。

  ポインタも場合によっては定数になる場合があります。しかし、定数は少ないです。その定数の半分は定数化変数です。

- アドレス値とポインタは異なります。

◆メモリ構造の表現方法

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



※ ポインタとは、アドレス値を保存するための、変数です。

8ビット→1バイトのアドレス値

16ビット→2バイトのアドレス値

32ビット→4バイトのアドレス値。

歴史的にポインタのアドレス値は変わり、現在は4バイトで表現します。→32ビットベースのアドレス値は4バイトです。

問題自体が2次元的な場合があります。

解決する問題は、2次元であるが、実際の構造は1次元的ではあるが、2次元的にする習慣が必要です。

◆ポインタと配列の関係

－ 配列の名前は、最初の要素のアドレス値を示します。

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

Ox1000

a[0]：aという配列の0番目の要素、&a[0]：0番目のインデックスに対応するアドレス値を返す →Ox1000

Ox1004

Ox1008

....

aというのは、配列の最初の要素のアドレスを表します。

◆ポインタ演算と配列

※arrがポインタまたは配列名である場合

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

* ptr+ +//* ptrの値を求め、ptrを1増加させます。

(* ptr)+ +//まず、* ptrの値を求めた後* ptrの値を1増加させます。

+ + ptr// ptrを1増加させた後 ptrを求めます。

+ +* ptr//* ptrの値を1増加させた後、その値を結果としてとります。

◆関数ポインタ

関数の名前は、関数の位置を指すポインタです。

関数名のポインタ型を決定する要素は、リターン型と伝達因子です。

int(* fun)(int);// int型のパラメータ1個を受け取ります。

// funはポインタの名前です。

//リターン型はint型です。

◆ void ポインタ

どのような形のアドレス値でも保存できるポインタです。

例）char c=' a' ;

int n=10;

void* vp;

vp=&c;

vp=&n;

ポインタ演算や値を変更すると参照することができません。

CH.5 : メモリ

◆メモリの動的割り当て

※ スタック、ヒープ、そしてデータ領域

- プログラムの実行のためにデフォルトで割り当てられているメモリ領域です。
- コンパイル時に関数が必要とするスタックのサイズを決定する必要があります。
- OSが管理してくれるメモリ構造であり、C+ +、JAVAなどが実行されると、以下のようにメモリが管理されます。

※ データ領域:グローバル変数、Static変数を保存するための場所です。

※スタック領域:ローカル変数、パラメータを保存するための場所です。

- コンパイル時に関数で要求するスタックのサイズを決定する必要があります。
- 必ずコンパイルタイムに決定されます。

● malloc関数

heap領域にメモリを割り当てます。

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

● free関数

heap領域に割り当てられたメモリを解放します。

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

CH.6：構造体

◆構造体の定義

－ 一つ（通常2つ）以上の基本型をベースにユーザー定義のデータ型を作成できる文法要素です。

```
struct point// pointという名前の構造体の宣言
{
Int x;//構造体メンバint x
Int y;//構造体メンバint y.
};
```

構造体の宣言と変数宣言を同時に！

```
struct 構造体名{
構造体メンバの宣言;
構造体メンバの宣言;
                :
}構造体変数名1、構造体変数名2…;
```



構造体の宣言と変数宣言を別々に！

```
struct 構造体名{
構造体メンバの宣言;
構造体メンバの宣言;
                :
};
struct 構造体名 構造体変数名1、構造体変数名2…;
```

◆構造体の特徴

構造体変数は、関数の伝達因子として使用可能です。

構造体は、基本のデータ型のように配列を宣言して使用できます。

```
例）struct phone list[10];//構造体の配列の宣言
    list[3]. age;// list配列の第四配列の要素の
                 // ageメンバを参照する
```

◆構造体ポインタ

- 構造体は、通常の変数のようにアドレス演算が可能で、構造体変数全体を指すポインタを持つことができる。

例）構造体ポインタ変数の宣言と初期化

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

例）構造体ポインタを利用したメンバを参照する

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

◆間接メンバ参照演算子

- 間接メンバ参照演算子 : (->)

例) 間接メンバ参照演算子使用法

```
(*sp).kor  == sp->kor
```

例) lp[i].name　　// 配列表現

```
==(*(lp+ i)). name;//ポインタ表現
==(lp+ i) - > name;//間接メンバ参照演算子を用いた表現
```

◆ 自己参照構造体

- 概念

構造体メンバーの中で自分自身を指すポインタを持っている構造体

例)

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

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

◆共用体

構造体の宣言と変数宣言を同時に！

```
union 共用体名{
共用体メンバの宣言;
共用体メンバの宣言;
        :
}共用体変数名1、共用体変数名2…;
```



構造体の宣言と変数宣言を別々に！

```
union 共用体名{
共用体メンバの宣言;
共用体メンバの宣言;
                :
};
union 共用体名 共用体変数名1、共用体変数名2…;
```

◆ typedef

構造体、共用体、列挙型のような応用データ型を使用する時に長くなる宣言を簡潔に記述できるようにデータ型の名前を再定義する。

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

◆列挙型

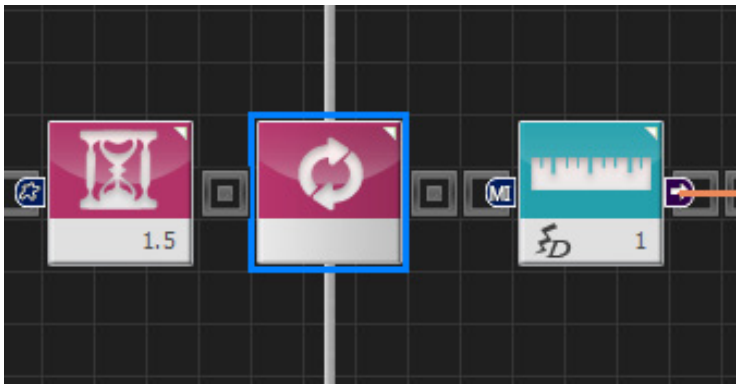
- 概念
  - 記憶に保存されるデータの集合を定義します。
  - 列挙型のメンバは、整数定数として扱われます。
  - プログラムの読みやすさを高めます。
  - 特定の整数値に意味を付与します。

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

CH.7 : 繰り返しと分岐

◆ 繰り返し

例) digital.dtsの一部



C-like 見る

```
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 ) )
```

反復文とは？

※反復文の機能

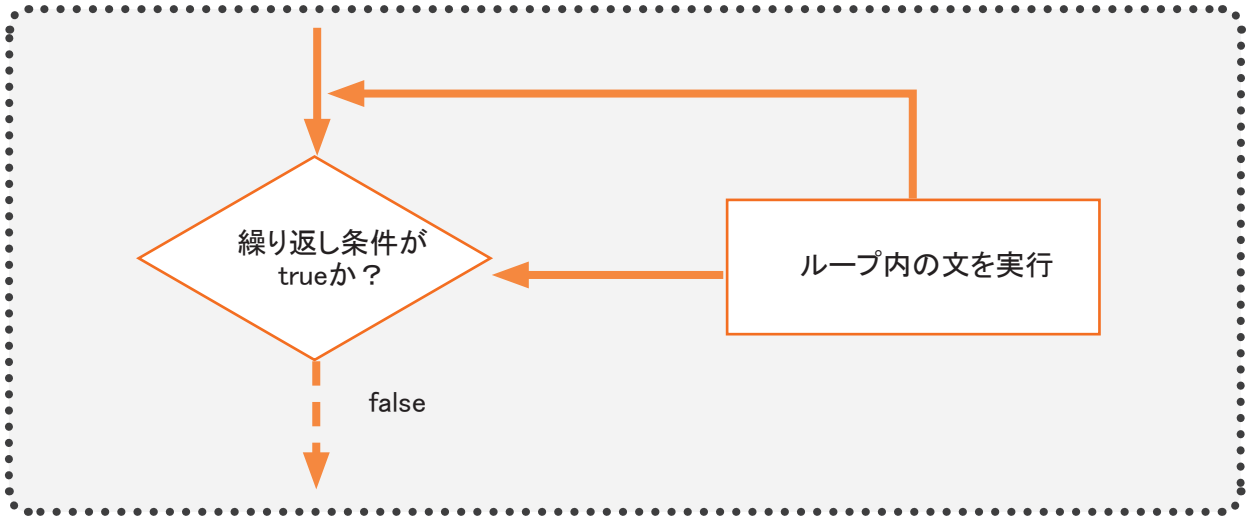
- 特定の領域を特定の条件が満たされる間だけ繰り返し実行するための文です。

※3つの形式の反復文

- while文による繰り返し
- do～while文による繰り返し
- for文による繰り返し

一つだけよく理解すると、すべてうまく理解できます。

while文の順序



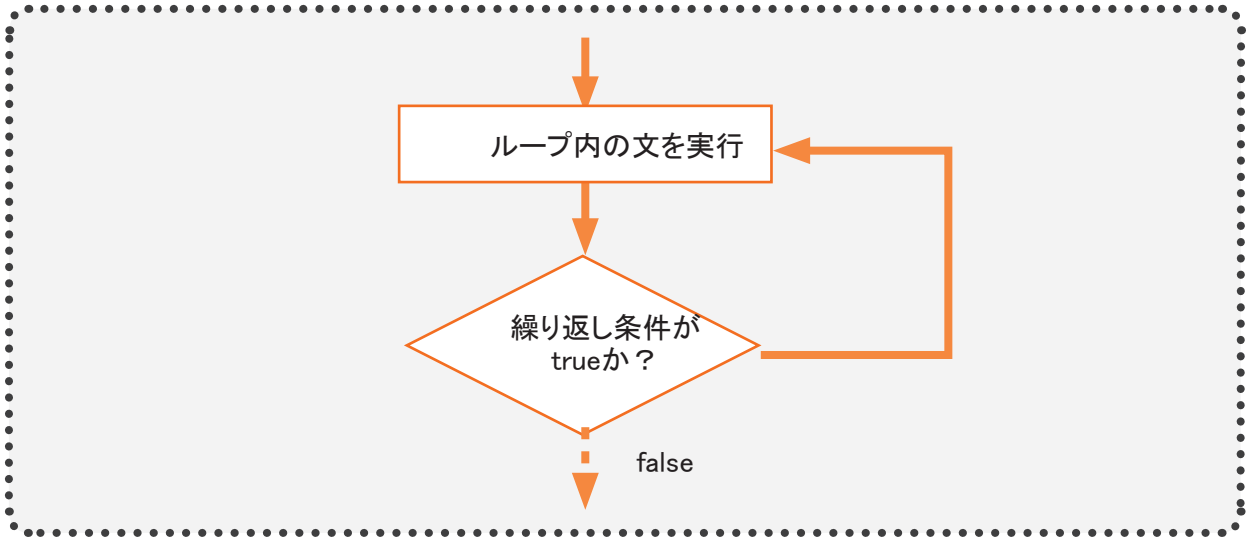
一般的な形式

```
While(条件式){
        繰り返し文;
}
```

● 先条件比較によるループです。（条件がtrueでない場合は、一度も実行できない場合があります。）

● 条件式が0(zero)以外の値になるとCはtrueと認識して無限loopしますので、必ず脱出文を繰り返すブロックの中に含める必要があります。

do~while文の順序

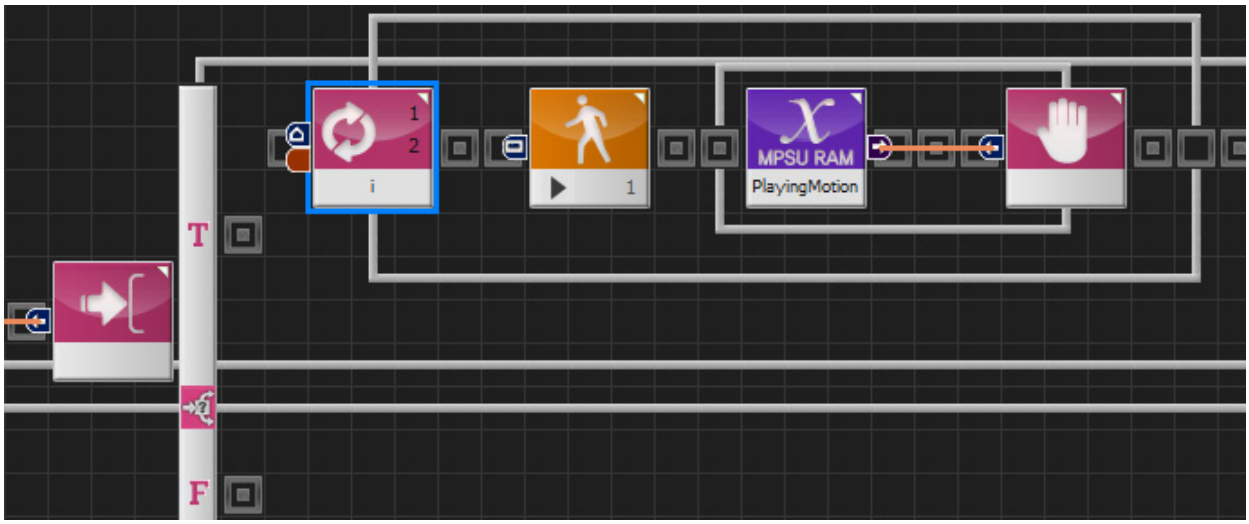


一般的な形式

```
do {
        繰り返し文;
} while(条件式);
```

●後条件の比較によるループです。（条件がtrueでない場合でも、一度実行する。）

例) digital.dtsの一部



C-like 見る

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

for文

一般的な形式

```
for (初期式;条件式;増減式){
        繰り返す文;
}
```

● 初期式
- Loop制御変数を初期化します。(最初に一度だけ実行)

● 条件式
- Loop制御変数の範囲をチェックして、繰り返しの有無を決定する。

● 増減式
- Loop制御変数を増加または減少させる。

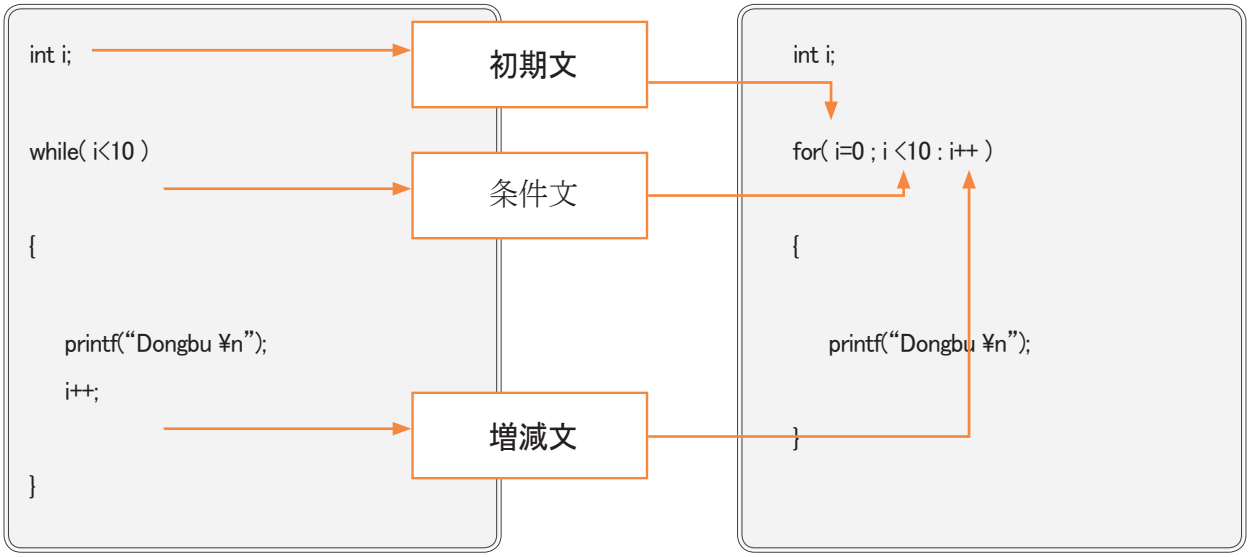
複数のfor文

```
for(初期式;条件式;増減式){
for(初期式;条件式;増減式){
        繰り返す文1;
    }
    繰り返す文2;
}
```

● break文
- ループを出るときに使用するキーワード

● continue文
- 回ったループの残りの部分をそのままスキップする

forとwhile文の比較



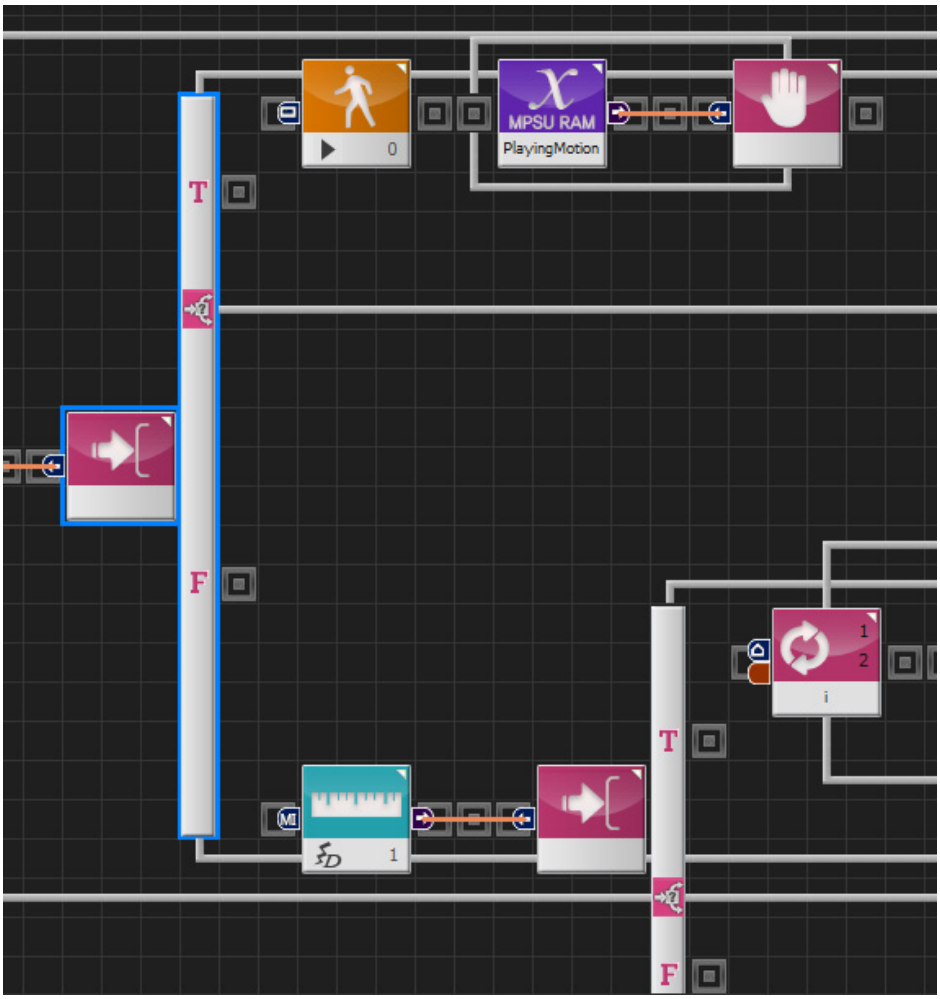

```
for(i=0;i<n+1;i++)
        total+=i;
```

while文に変更すると

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

◆条件分岐

例) digital.dtsの一部



C-like見る

```
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          {
```

◆ 条件分岐文

条件分岐文は、if、if else、switch caseの3つの形式がありますが、機能は同じなので、visual logicではif elseで表現しました。

◆ if

if文による条件付き実行：条件が満たされる場合に限り実行します。

if（実行の条件）→trueと仮定すると、
{ - →実行しようとする文章が一文の場合、中括弧を省略できます。
実行しようとする内容
}

```
int main(void)
{
  int val;
  printf("整数を1つ入力してください")
  scanf(%d, &val);

  if(val<0) //val <0 が trueの場合…
  {
    printf("入力値は、0より小さい"
  } → 文章が複数ある場合は、必ず入れる必要がある
  if(val>0)
    printf("入力値は、0より大きい"
  if(val==0)
    printf("入力値は0である。"

  return 0;
}
```

もし1を入力すると、2番目はTrueです。そのため3番目の条件の確認をする必要はありません。
→ このような問題点を解決するためにelseを使用します。

◆ if～else

前の条件が満足した場合でも、その下の文字の状態をチェックします。不要な比較演算をたくさんすることを防ぐためにif elseを使用します。

```
if（条件）
{
        条件を満たす時に実行一名"これ"
}
else
{
        条件に不満時に実行一名"あれ"
}
```

◆ if, else if, else

if(条件A) → 条件Aを満足する場合、“P1”を実行した後、else文まで飛び越える。

{

条件Aを満足した場合実行するプログラム名“P1”

}

else if(条件B) → 条件Aを満足しないが、条件Bを満足する場合、“P2”を実行した後、else文まで飛び越える。

{

条件Bを満足した場合実行するプログラム名“P2”

}

else if(条件C) → 条件A,Bを満足しないが、条件Cを満足する場合、“P3”を実行した後、else文まで飛び越える。

条件Cを満足した場合実行するプログラム名“P3”

}

else → case3: 条件A,B,Cを満足していない場合、else文の“P4”が実行される。

{

条件A,B,Cを満足していない場合実行するプログラム名”P4”

}

たとえば、条件Aが満足しないが、 条件Bを満足する場合、”P1”はスキップして、”P2”を実行した後、残りはすべてスキップします。→ 不要な演算を克服することができます。

◆ switch ~ case

int型char型の引数を一個転送できる

switch(n)

Case1: → nが1である場合

Break;

Case2:

Break;

Case3:

Break;

default:

n=2の場合、case1ではないのでスキップします。

case2は合うので実行します。break文は、switchを抜け出て行くという意味です。

もしbreakがない場合は、2からその下はすべて実行されます。

◆条件分岐全体の概要

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

```
if(条件)
{
  条件を満足する時に実行”
}
```

```
if(条件)
{
  条件を満足する時に実行”
}
else
{
条件を満足しない時に実行
}
```

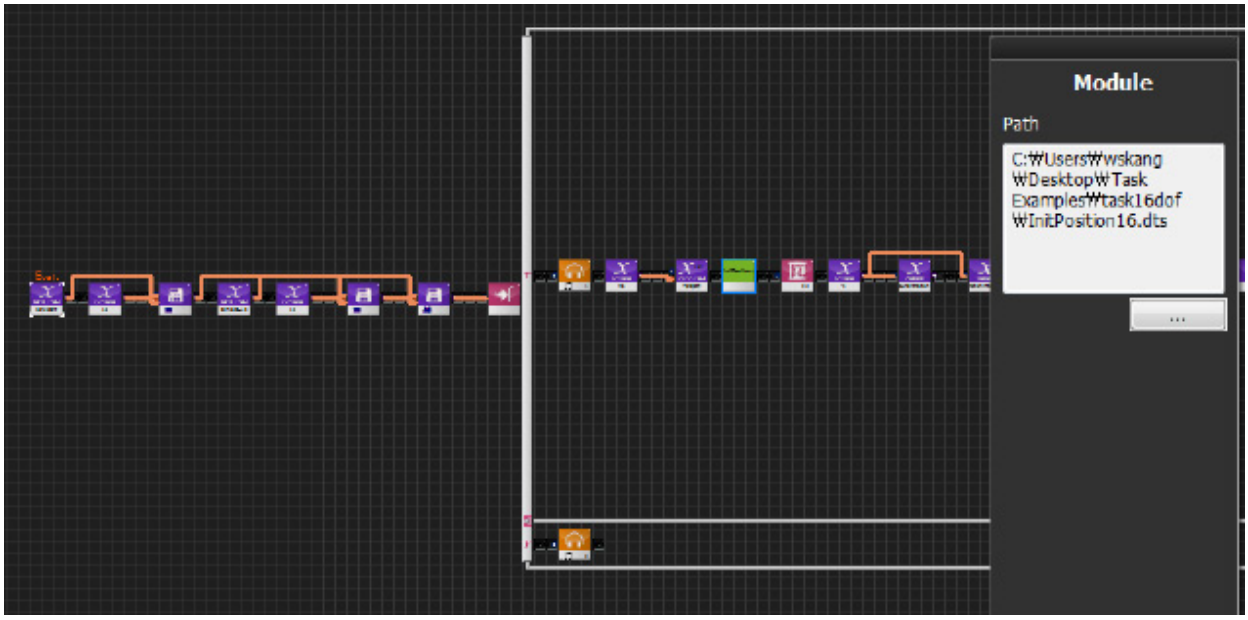
```
if(条件A)
{
      条件Aを満足する時に実行
}
else if(条件B)
{
      条件Bを満足する時に実行
}
else if(条件C)
{
      条件Cを満足する時に実行
}
else
{
      条件ABCを満足しない時に実行
}
```

```
switch(n)

  case1:  → nが 1である場合
          Break;
  case2:
          Break;
  case3:
    Break;

default:
```

CH.8 : 関数

◆ メイン 関数

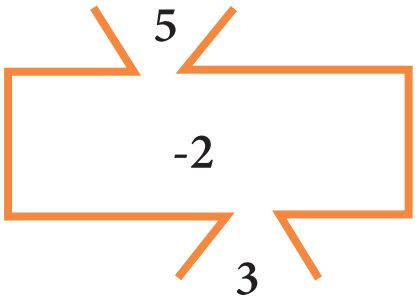
例) Remocon16.dtsの一部

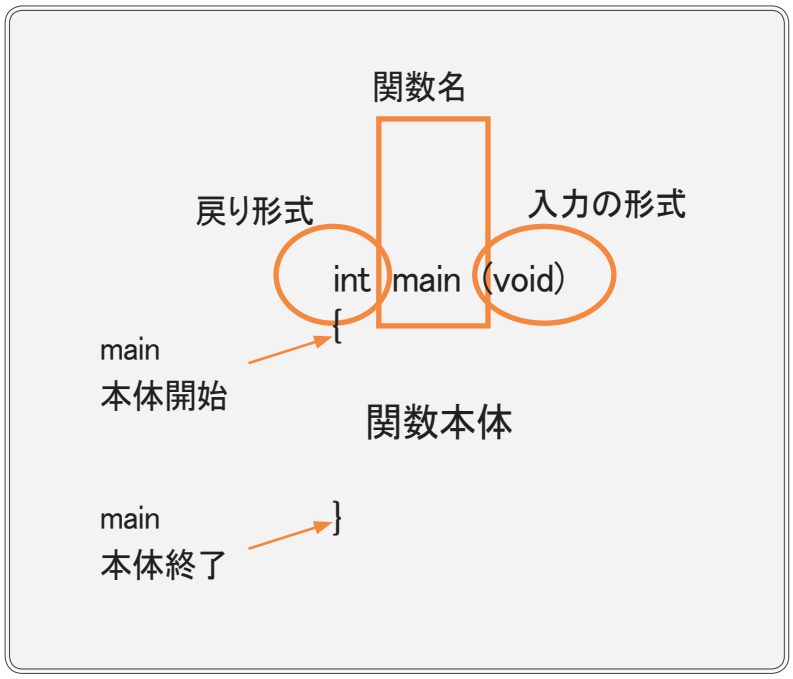


C-like見る

```
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 )
```

※Cという言語で実装されたプログラムは、関数から始めて、関数で終わります。関数は、C言語を理解することに最も重要です。ポインタが難しいのは事実ですが、関数がより重要です。ポインタは勉強するのが難しいのであって活用することは難しくありません。





1番目は、関数は名前を持ちます。→ main

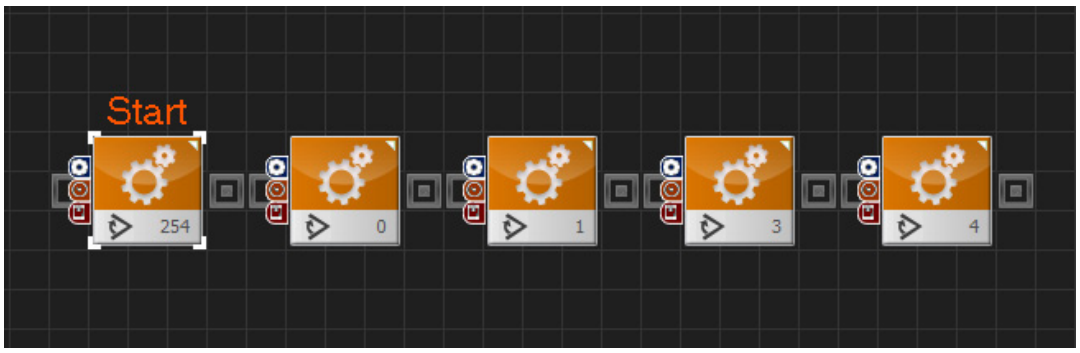
2番目は、入力の形式があります。→ void)

3番目は、出力の形態（戻り値の型、リターン型という言葉を多く使用する）があります。→ int

4番目は、関数の本体：関数の機能を意味します。

A、B、Cの関数をたくさん作っておいて、特定の順序に合わせて呼び出すことがプログラミングです。

例)InitPosition16.dtsの一部



C-like 見る

```
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  }
```

◆関数の宣言

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

(1)戻り型
(2)関数名
(3)パラメータ
(4)値の返り

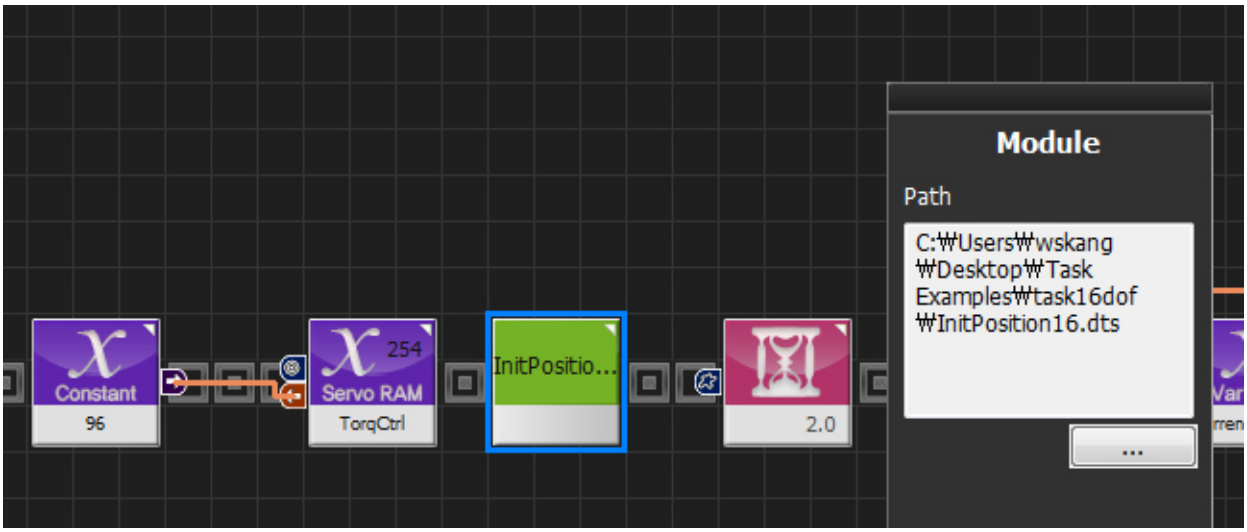
※ 4つのタイプの関数
入力と出力はないこともあります。重要なのは機能です。
伝達因子あり、戻り値あり
伝達因子あり、戻り値なし
伝達因子なし、戻り値あり
伝達因子なし、戻り値なし

変数の範囲(scope)
※変数の特性に応じた分類
ローカル変数(local variable):括弧内で宣言された変数
グローバル変数(Global variable):関数内で宣言されない変数
静的変数(Static variable):関数の内部、外部の両方の宣言が可能
レジスタ変数(Register variable):宣言に制限がたくさんあります

static変数
関数の内部と外部に宣言できます。
グローバル変数、ローカル変数の前にstaticキーワードを使用できます。→ファイルの範囲を持っています。↔ extern
一度だけ初期化されます。:グローバル変数の特徴
グローバル変数は、プログラムが開始すると上がって、終了すると下がります。
ローカル変数は、関数が呼び出されるたびに初期化されます。
関数の内部で宣言された場合、関数内でのみアクセスが可能です。:ローカル変数の特徴

◆関数呼び出しとモジュール化プログラミング

例) Remocon16.dtsの一部



C-like 見る

```
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
...

● モジュール化プログラミング

- 機能別にファイルを分けてプログラミングすることです。
- メンテナンス性が良くて管理するのが楽です。

たとえば、200～300行までは大丈夫ですが、500行～1000行以上であれば区分も難しく、メンテナンスも困難になります。

● ファイルの分割とコンパイル

- ファイルが分かれても完全に独立はありません。
- ファイルが分かれても、相互参照が発生することがありますが、これはグローバル変数とグローバル関数に制限されます。
- 同じコードを二つのファイルに分離し、分離されたファイルは、一つのプロジェクトとして分類されます。

◆関数の呼び出し

※ Call-By-ValueとCall-By-Reference

値による　　参照による

※ Call-By-Value

- 値のコピーによる関数の呼び出しです。
- 最も一般的な関数呼び出し形式です。

int main(void)

add(val1, val2)

→ int add(int a, int b)

add関数内でmain関数に入ってval1を操作できません。

コピーしたので、addでのみ使用が可能です。

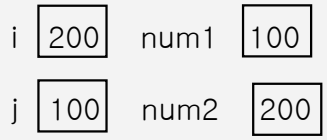
※ Call-By-Reference

- (参照を可能にするアドレス値)を引数として渡す関数呼び出しです。
- たとえば、変数valのアドレス値が0x10なら、adder(&val)はvalのアドレス値を渡します。pVal= 0x10です。

adderという関数は、アドレス値を取得して、(* pAvl)+ +;が指すメモリ領域移動し1を増加します。

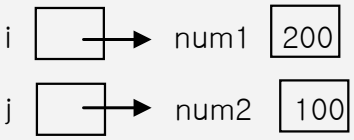
※ 値による呼び出しの例

```
#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;
}
```



※ 参照による呼び出しの例

```
#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;
}
```



◆関数のパラメータ

※ 関数のパラメータの種類

●実パラメータ

呼び出し関数側で伝達する実際の変数です。

●形式パラメータ

被呼び出し関数側で伝達を受ける変数です。

※関数のパラメータの特徴

- 実パラメータと形式パラメータの数は一致しています。
- 実パラメータの名前と形式パラメータの名前は同じでも構いません。

※ パラメータの引き渡し

– 配列を関数の引数として使用する時、配列のアドレスだけがパラメータに渡されます。

→ パラメータをポインタ型で宣言する必要があります。

```
#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;
}
```

→下の3つのすべての場合、1つのポインタをパラメータに使用

```
void a1(int num[5]) /*配列の使用*/
{
        int i;
        for(i=0; i<5;i++)    printf("%d", num[i]);
}
void a2(int num[])  /*アンサイズド配列の使用*/
{
        int i;
        for(i=0; i<5;i++)    printf("%d", num[i]);
}
void a3(int *num)  /*ポインタを使用*/
{
        int i;
        for(i=0; i<5;i++)   printf("%d", num[i]);
}
```

◆再帰関数

※ 再帰関数の基本的な理解

- 自分自身をもう一度呼び出す関数です。
- 再帰関数プログラムをよく理解するためには、コンピュータProcessをよく理解する必要があります。
- exeファイル作成するには、コンパイルされたバイナリが必要です。
- プログラムが開始されると、メモリCode areaに関数などすべてのものが上がります。
- メイン関数が実行されると、順次関数が実行されす。このコードを取り出して一つずつパッチを開始し、CPUがメイン関数コードを取り出して順番に実行します。
- リカーシブ関数を呼び出す→CPUがリカーシブをインポートした後に実行します。
- コード領域で再びリカーシブ関数を呼び出して、インポートした後に実行します。→ずっと繰り返す
- リカーシブ関数は終了されず、実行され、メモリに蓄積します。→問題となる
  （Stack overflowと呼ばれる：再帰的関数がずっと積み重なっていっぱいになること）

※ 脱出条件の必要性

無限再帰呼び出しを避けるためです。

データ構造（データを表現する方式）やアルゴリズム（問題解決）を勉強すると、再帰関数の便利さに気つきます。問題解決を悩んでいる時に、再帰的に解決することが多いです。

```
if(n==1)
return;
```
→ 1, 関数を抜け出ること、2、値を返すこと、二つの機能があります。

（リターンタイプがvoidだとしても、関数を抜け出すために使われます。）

```
recursive(n-1);
```

再帰関数において、脱出の条件があるかどうかが非常に重要です。

※ 再帰（recursive）呼び出し：循環（recursion）

：プロシージャ（procedure）が自分自身を呼び出すことです。

```
#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

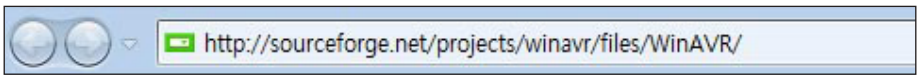
AVRロボット制御機Firmwareの習得

AVR の設置と使い方

AVRのプログラミングのためには、WinAVR compilerとAVR studioが必要です。

WinAVRは　http://sourceforge.net/projects/winavr/files/WinAVR/ ダウンロードします。

AVR studioは http://www.atmel.com/ からダウンロードします。

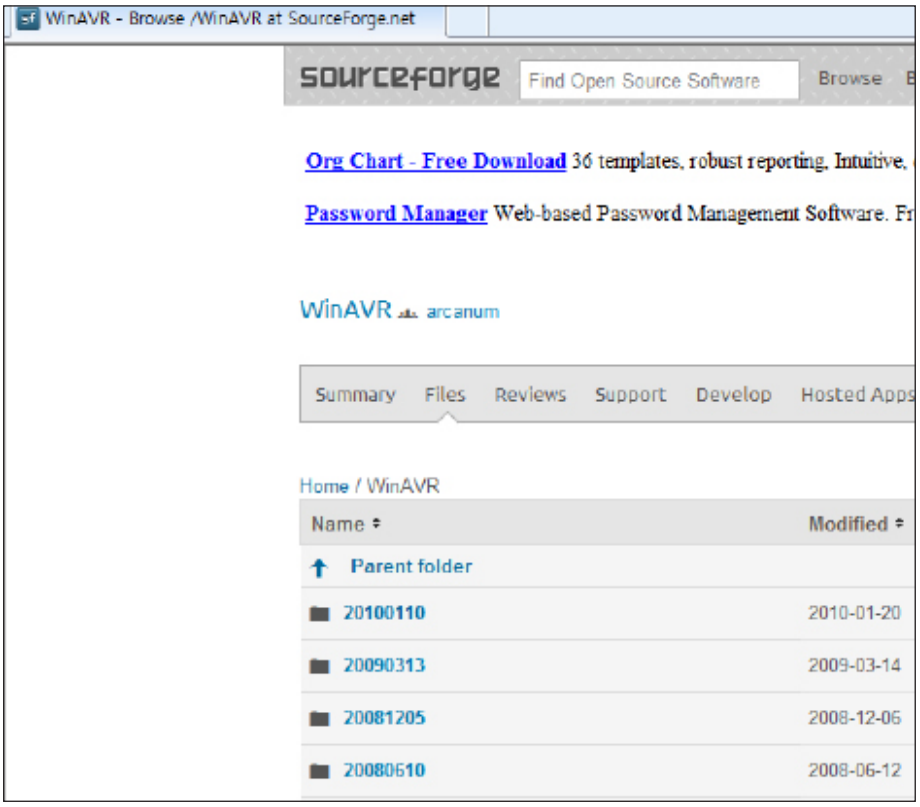


WinAVRを設置する

01 URL 入力

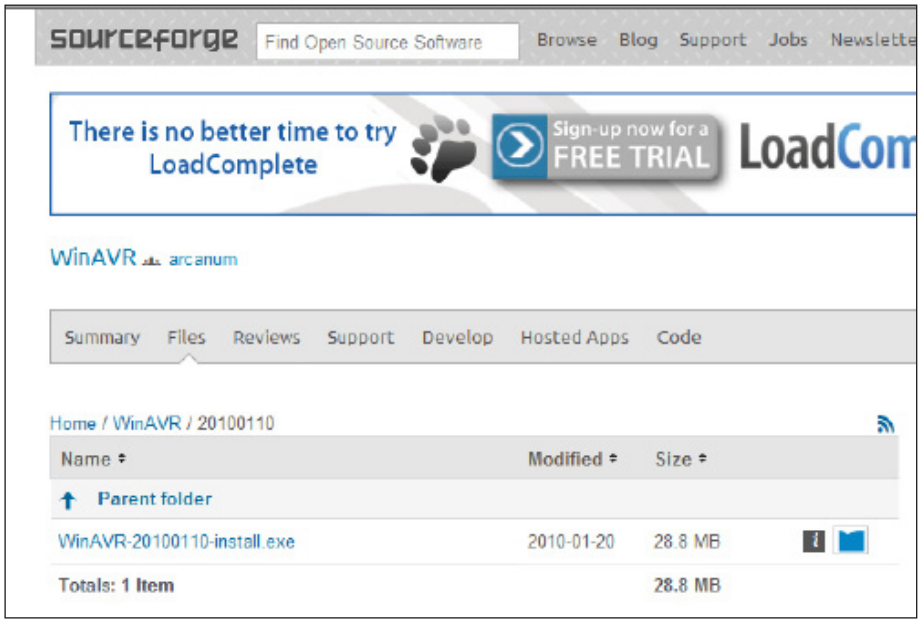
アドレスバーにURLを入力します。

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



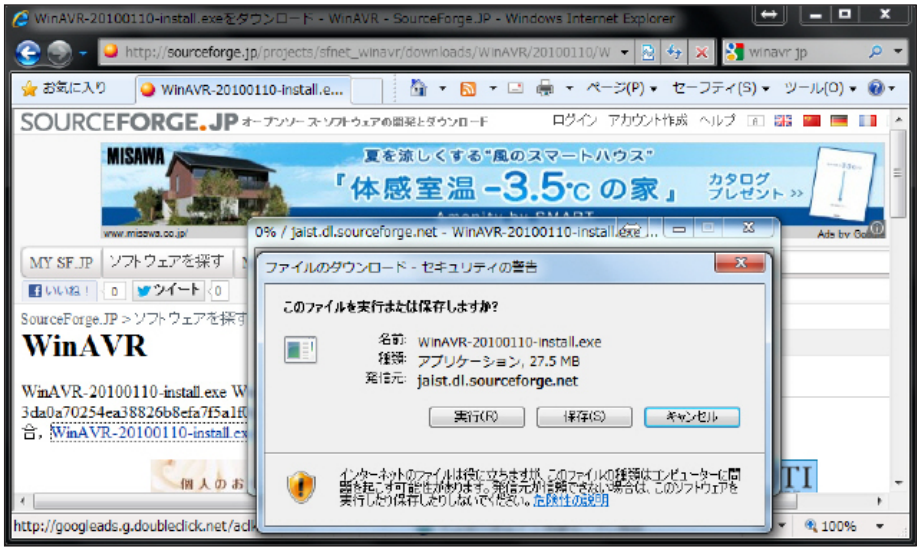
02 ダウンロードリンク

20100110の右側のチェックボックスをクリックします。



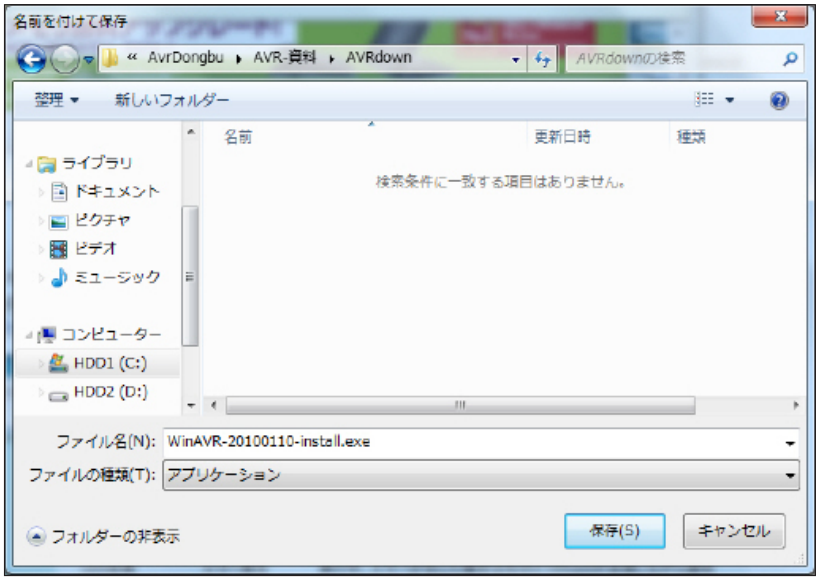
03 ダウンロード

WinAVT−20100110−install.exeのチェックボックスをクリックします。



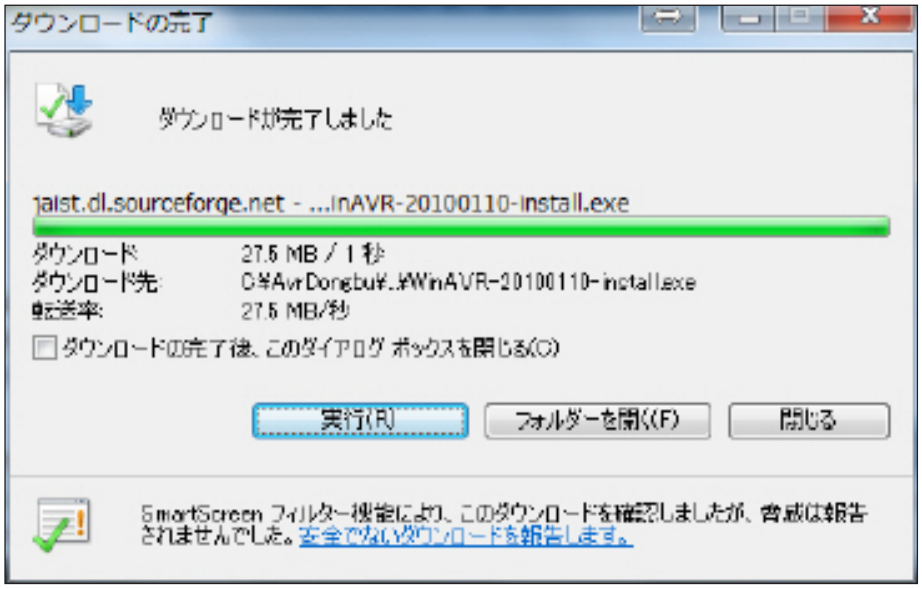
04 設置ファイルの保存

“保存”ボタンをクリックします。



05 保存フォルダを指定

設置ファイルを保存するフォルダを指定し、“保存”ボタンをクリックします 。



06 プログラムのダウンロード

“次へ”ボタンをクリックします。



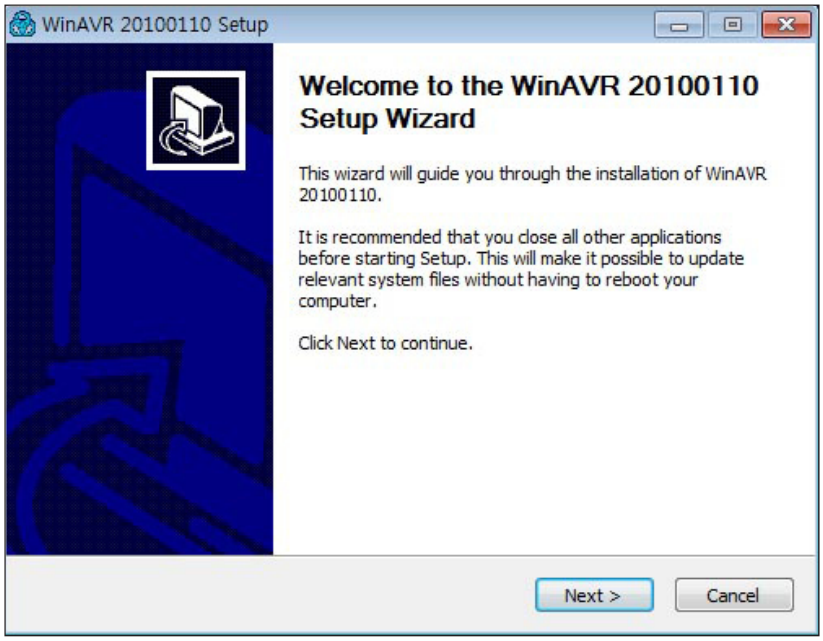
07 設置フォルダを選択

生成されたアイコンをダブルクリックします。



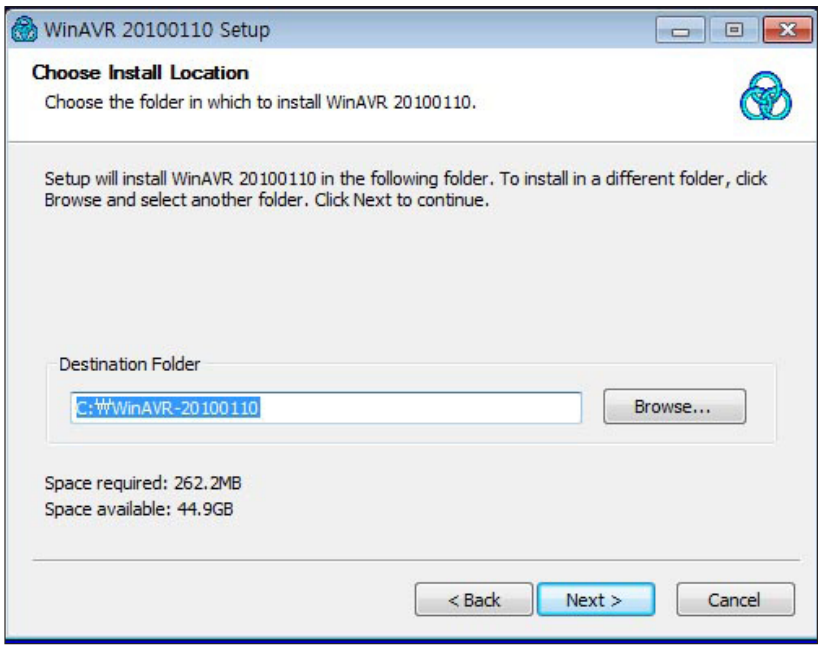
08 設置の確認

言語は“English”を選択し、“OK”をクリックします。



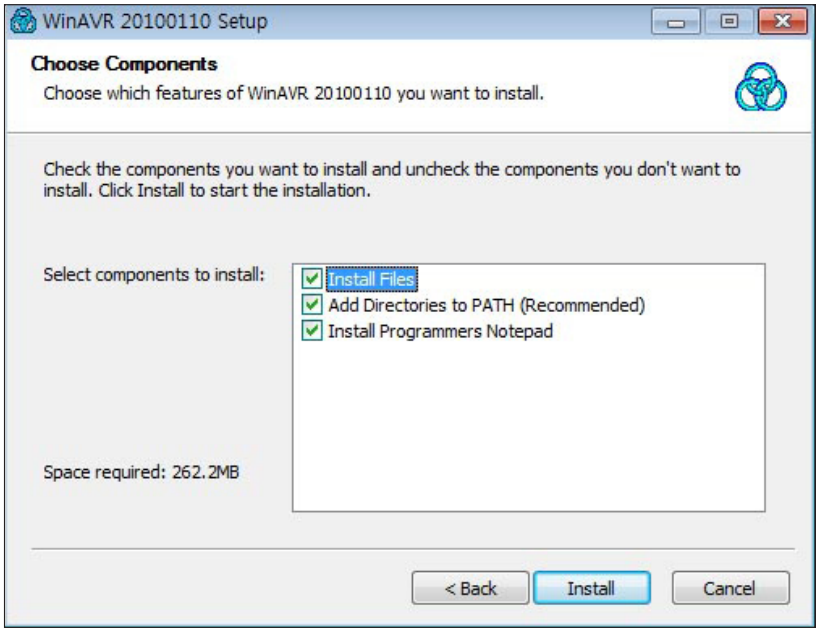
09　設置ウィザード開始

設置ウィザードが実行されます。“NEXT”をクリックします。



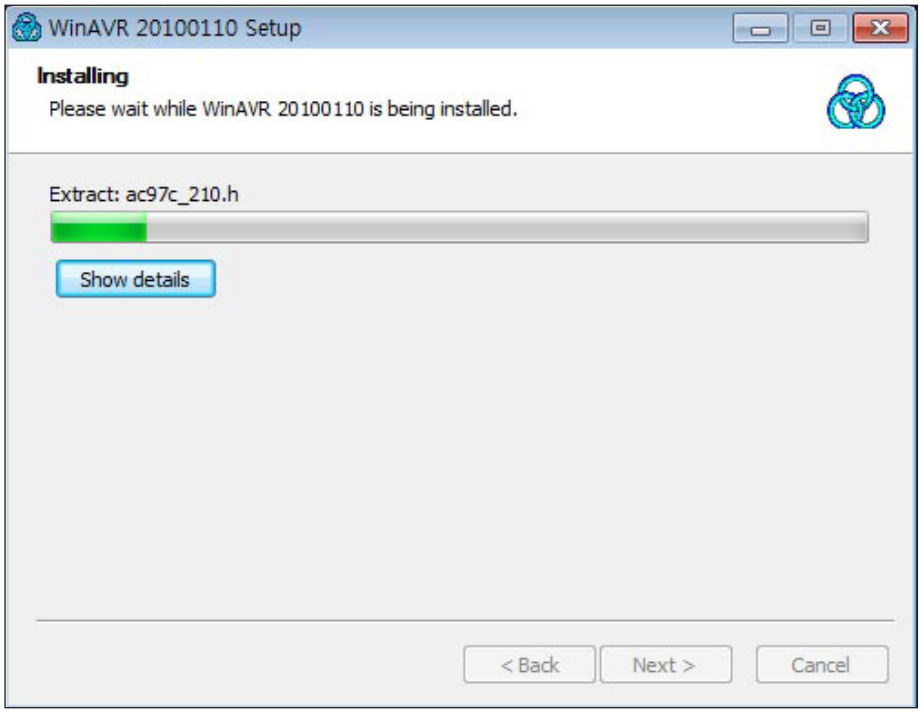
10　設置フォルダを確認

“NEXT” をクリックします。



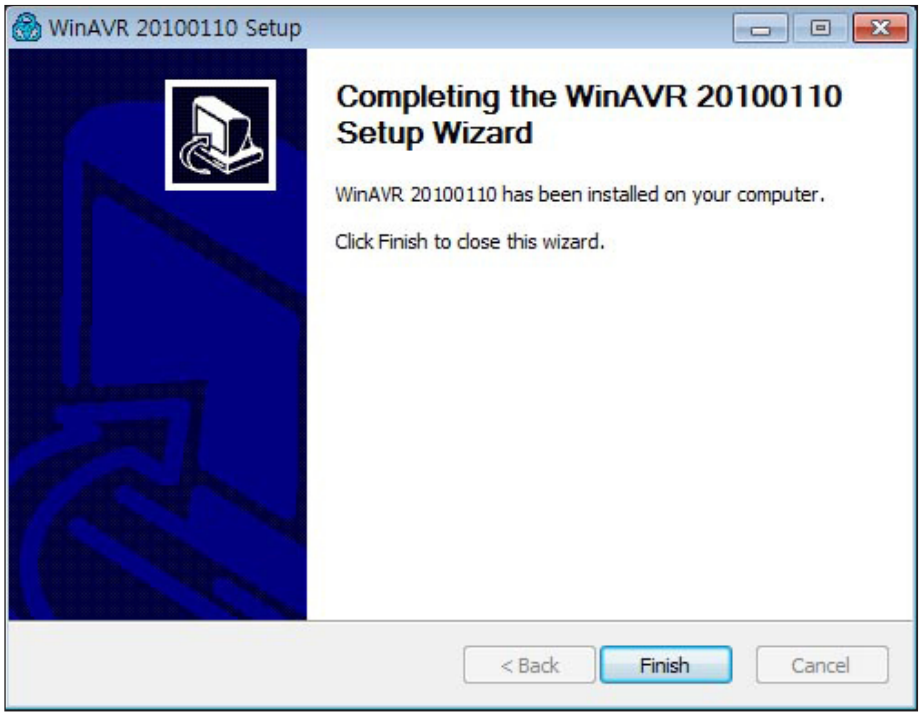
11　構成要素 の確認

“Install” をクリックします。



12　設置の開始

プログラムの設置を開始します。



13　設置の完了

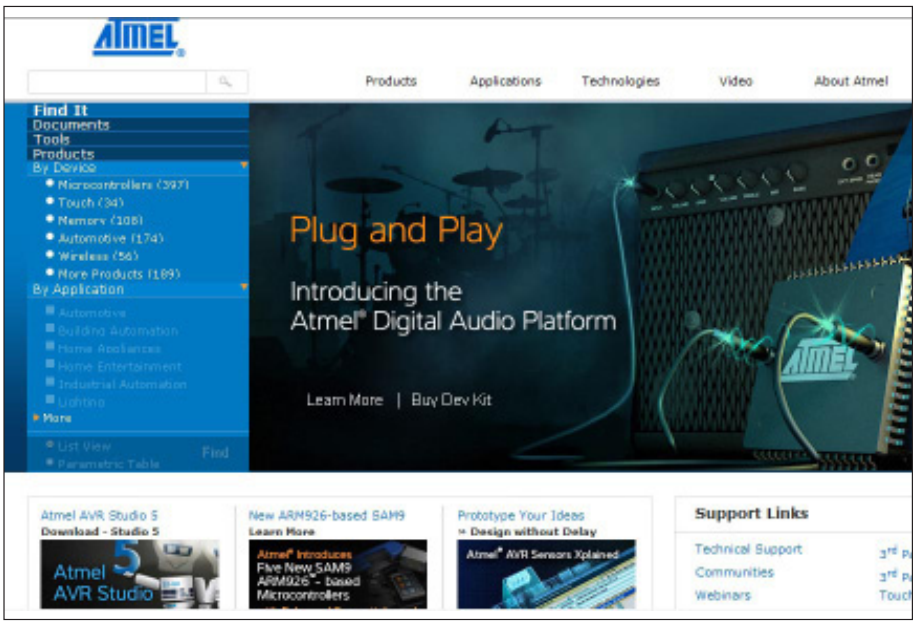
“Finish”をクリックすると設置が完了します。



AVR Studioを設置する

01 URL 入力

アドレスバーに
URL(www.atmel.com)
を入力します。



02 ダウンロードリンク

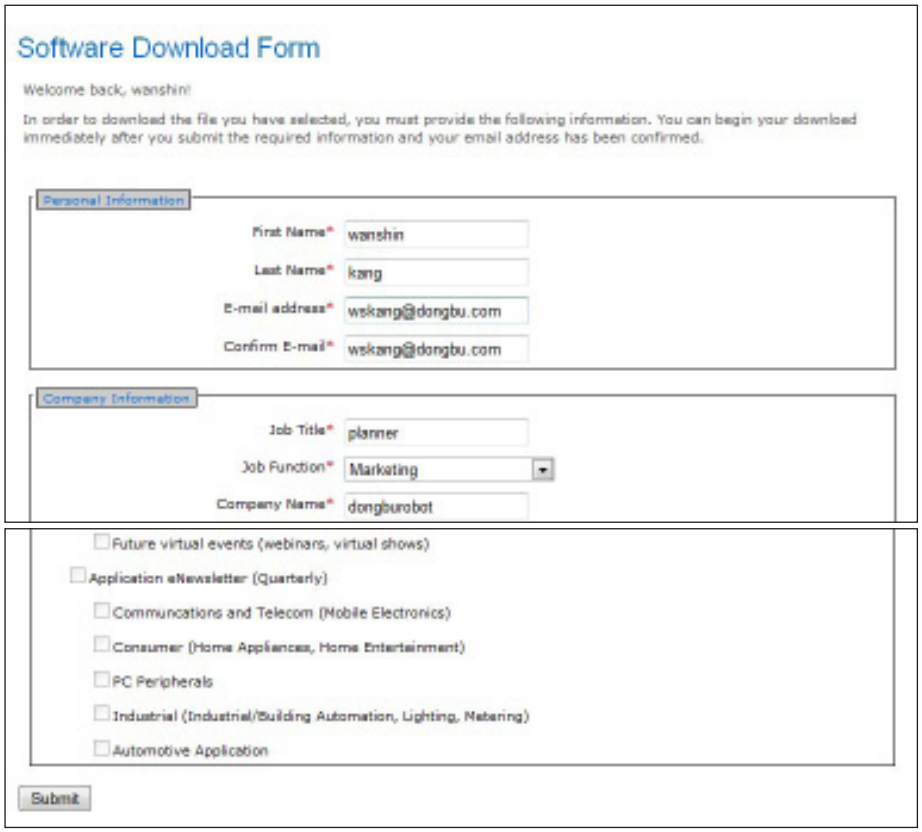
“Support Links”の
“Download AVR Studio 5”をクリックします。



03 ソフトウェアの選択

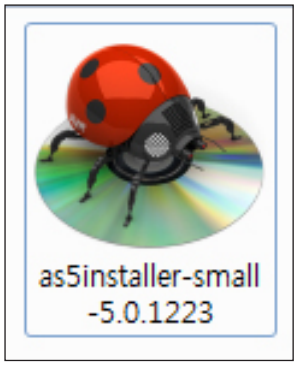
“AVR Studio 5.1 Installer–Full”をクリックします。





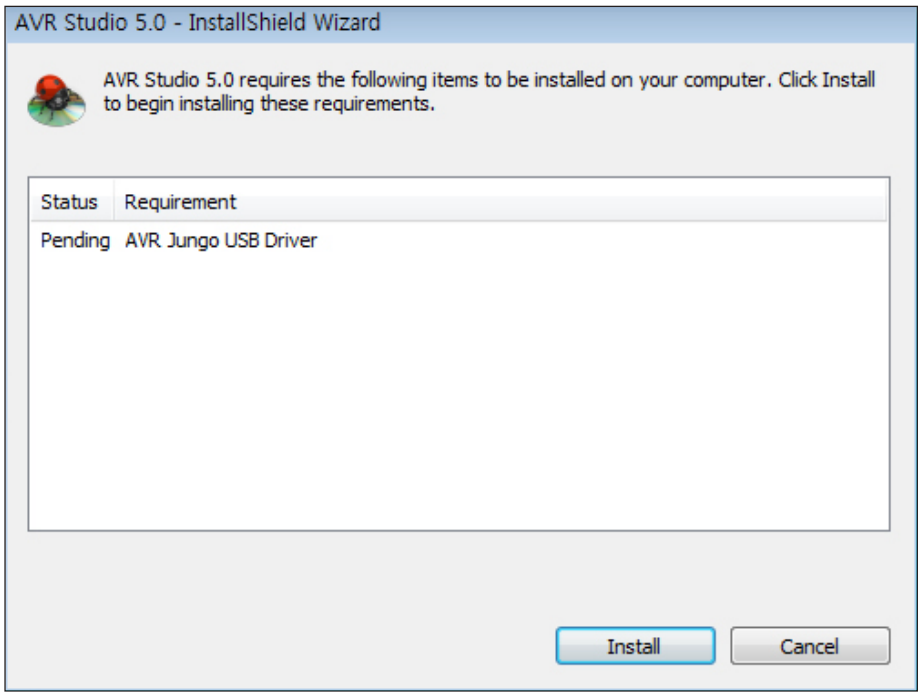
04 ソフトウェアのダウンロードフォーム

ユーザー情報を入力し、submitをクリックします。



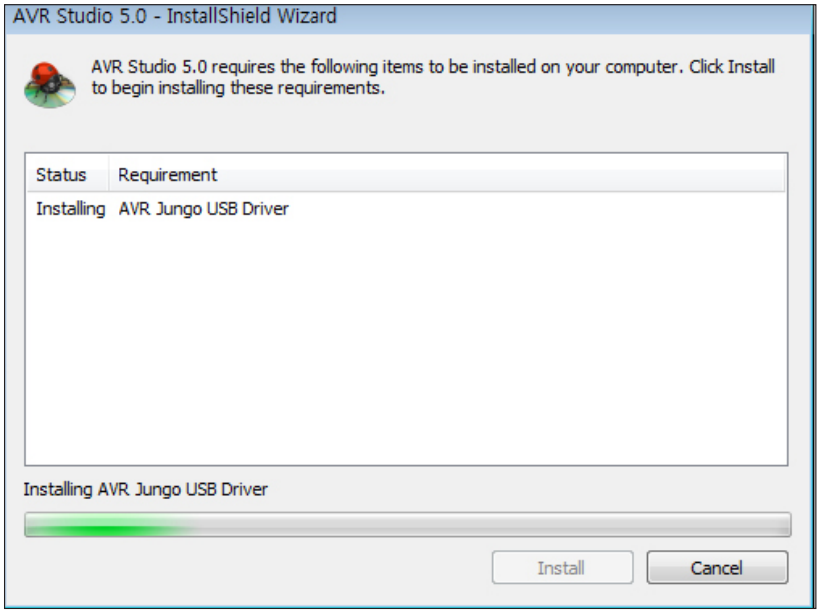
05 設置プログラムの実行

アイコンをダブルクリックします。



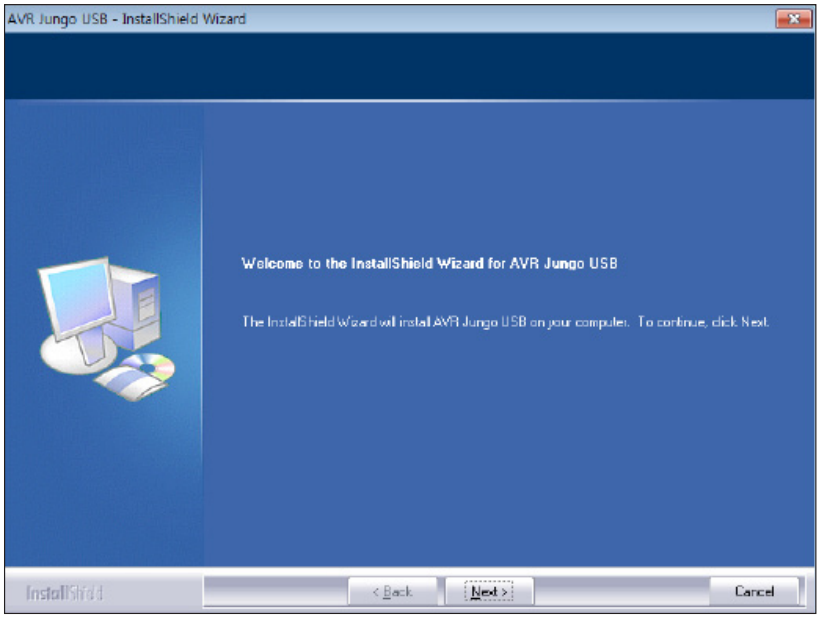
06 設置開始

“Install”をクリックします。



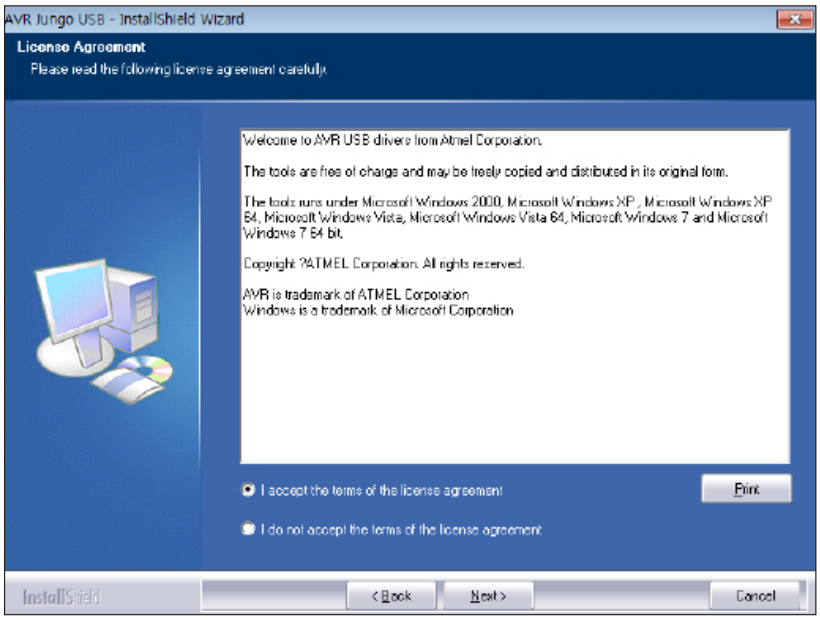
07 設置

プログラムの設置をします。



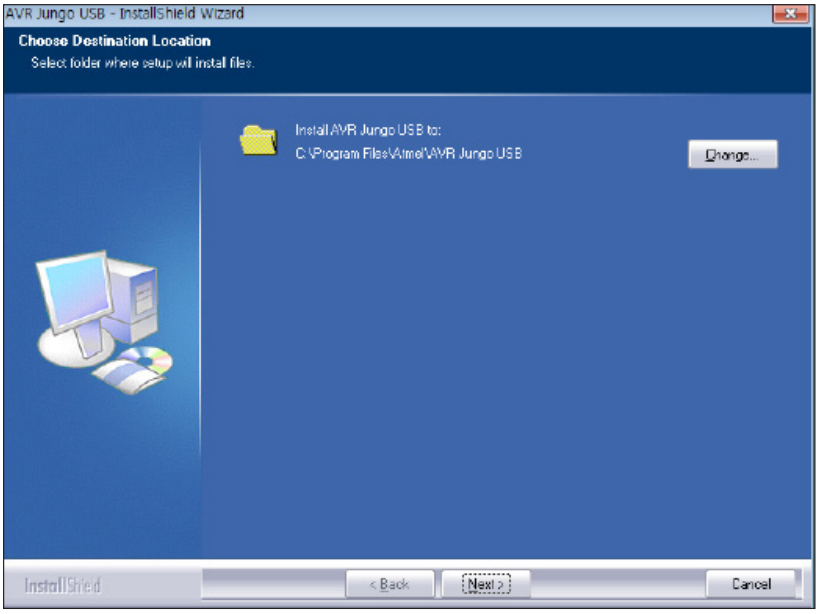
08 AVR Jungo USB設置確認

“NEXT”をクリックします。



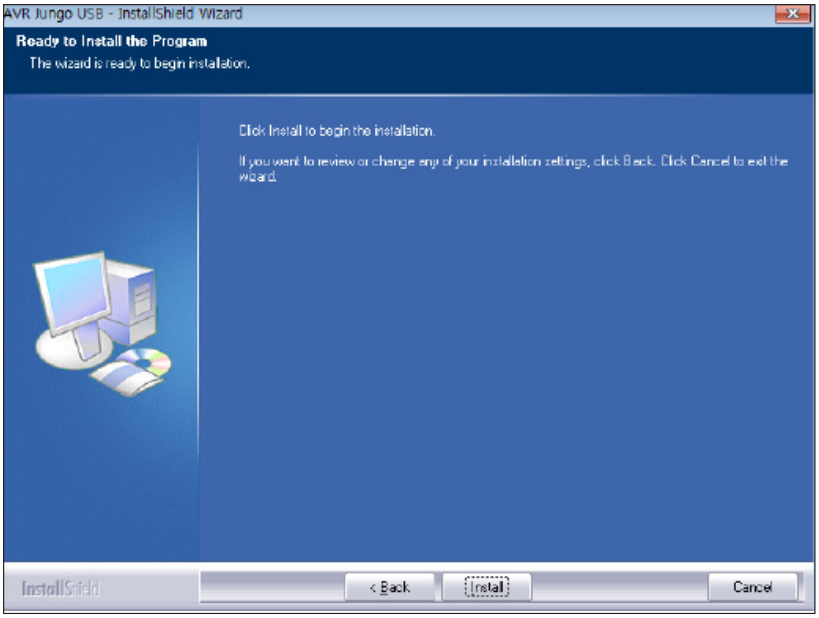
09 利用規約に同意

“I accept”を選択し “Next”をクリックします。



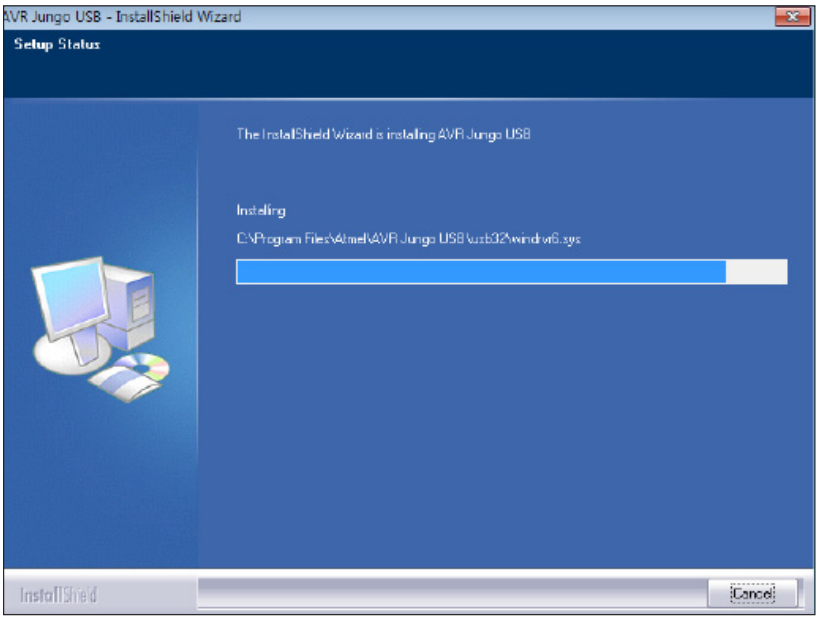
１０ 設置 フォルダの選択

プログラムの設置フォルダを確認した後、“Next”をクリックします 。



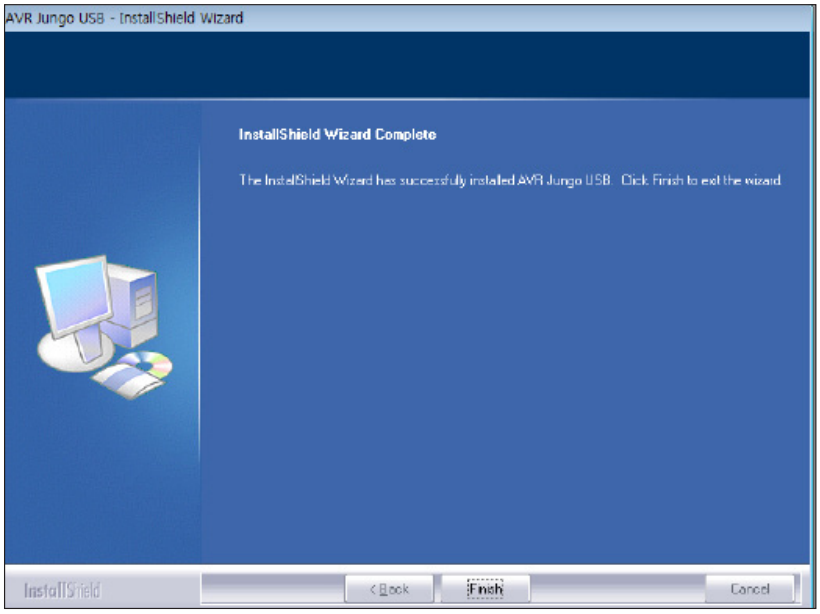
１１ 設置開始

“Install”をクリックすると設置を開始します。



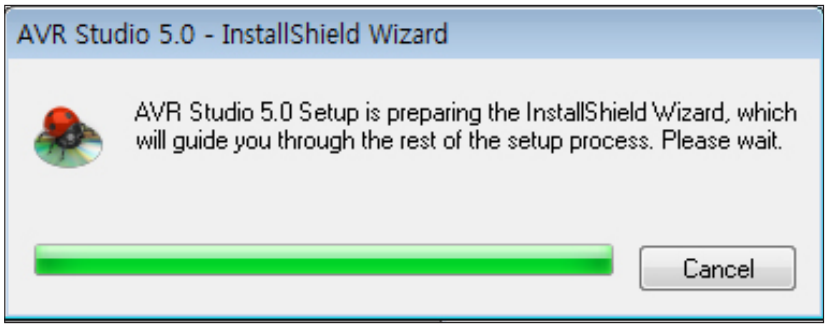
１２ 設置

プログラムが設置されます。



13 設置完了

“Finish”をクリックするとAVR Jungo USBの設置が完了します。



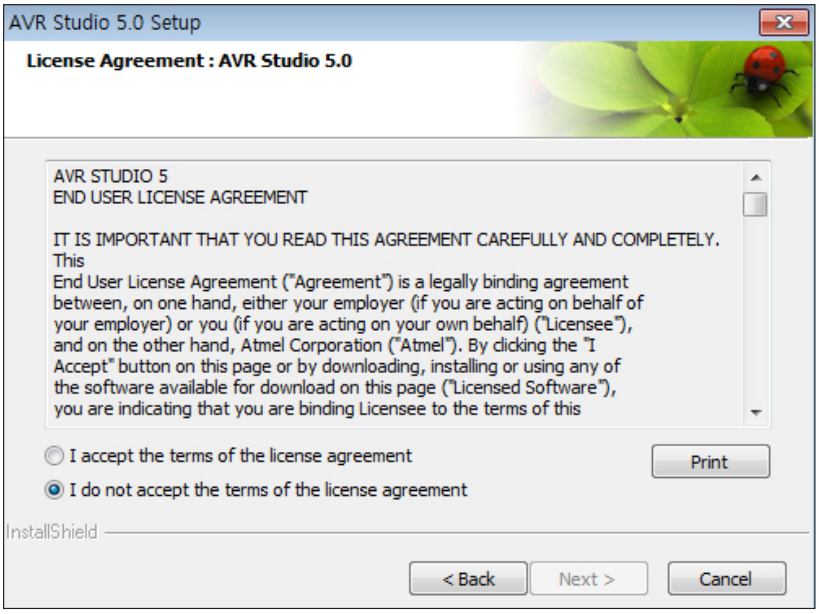
14 AVR Studio 5.0 設置準備の完了

AVR Studio 5.0を設置する準備が完了します。



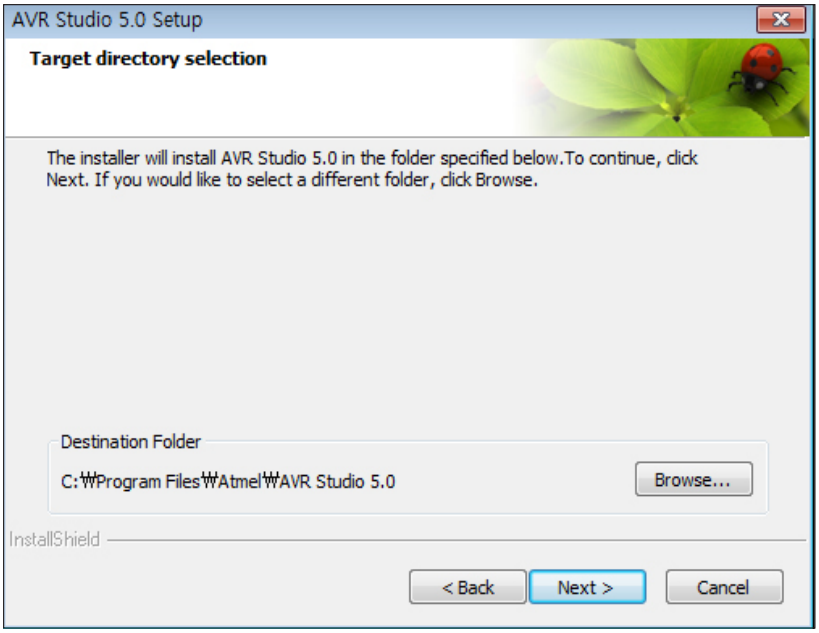
15 設置ウィザード開始

“Next”をクリックします。



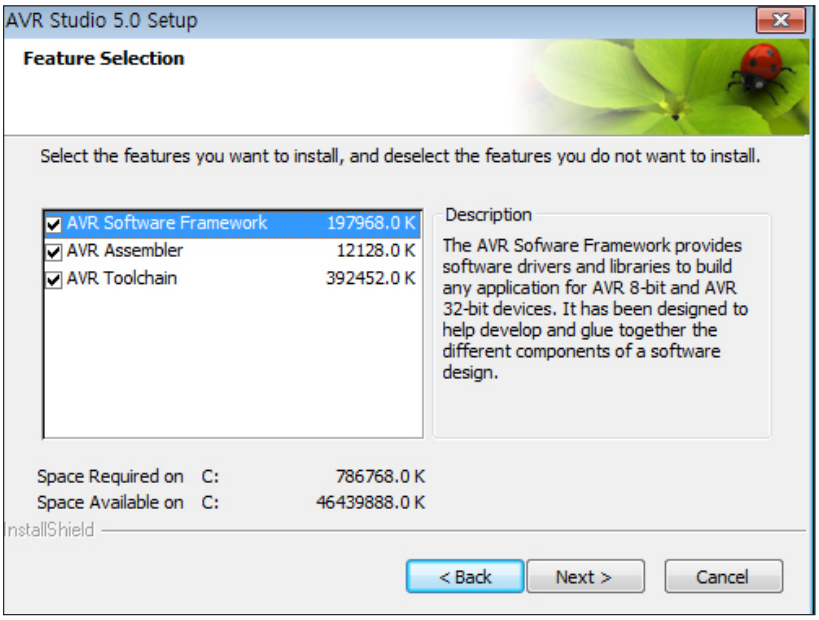
１６ 利用規約に同意

“I accept the terms of the licese agreement”を選択し“Next”をクリックします。



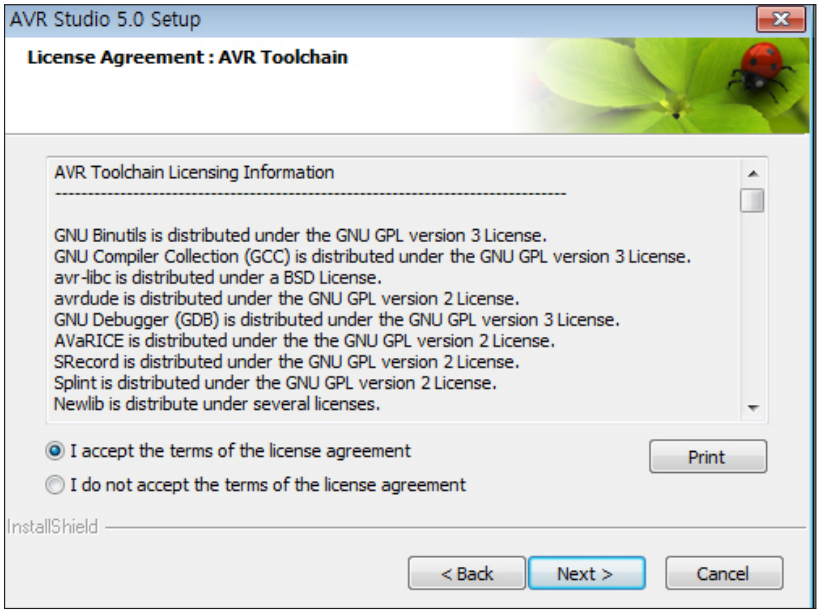
１７ 設置 フォルダの確認

設置フォルダを確認した後、“Next”をクリックします。



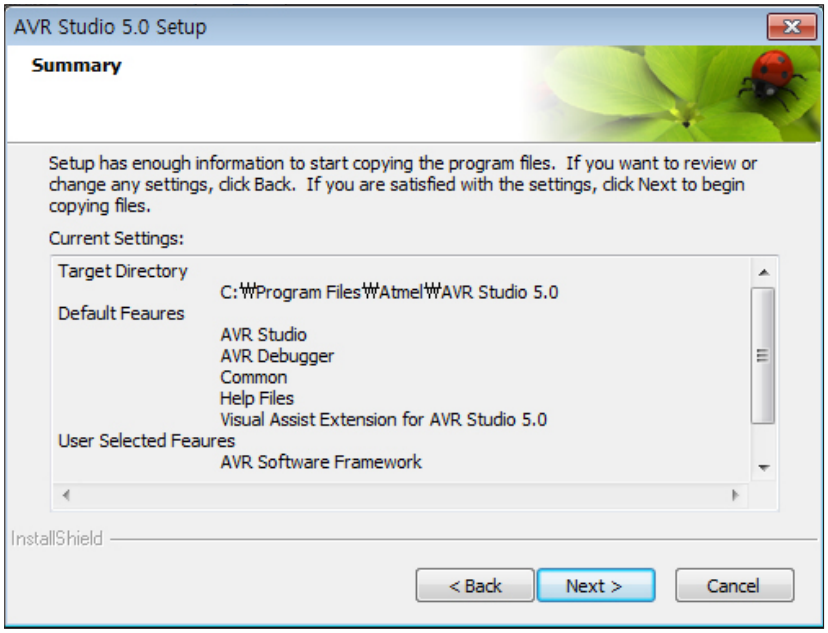
１８ 設置機能の選択

設置する機能を選択し“Next”をクリックします。



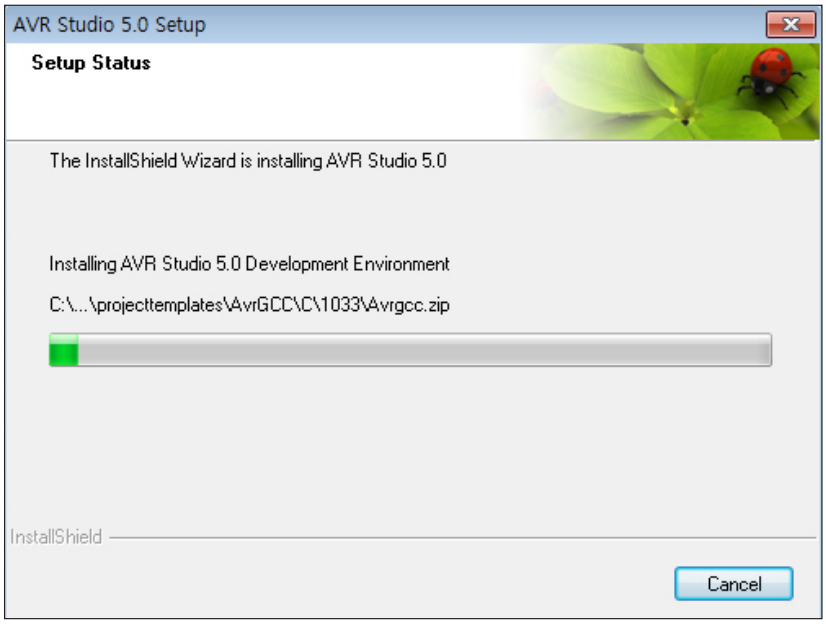
１９ 利用規約に同意

利用規約に同意した後、“Next”をクリックします。



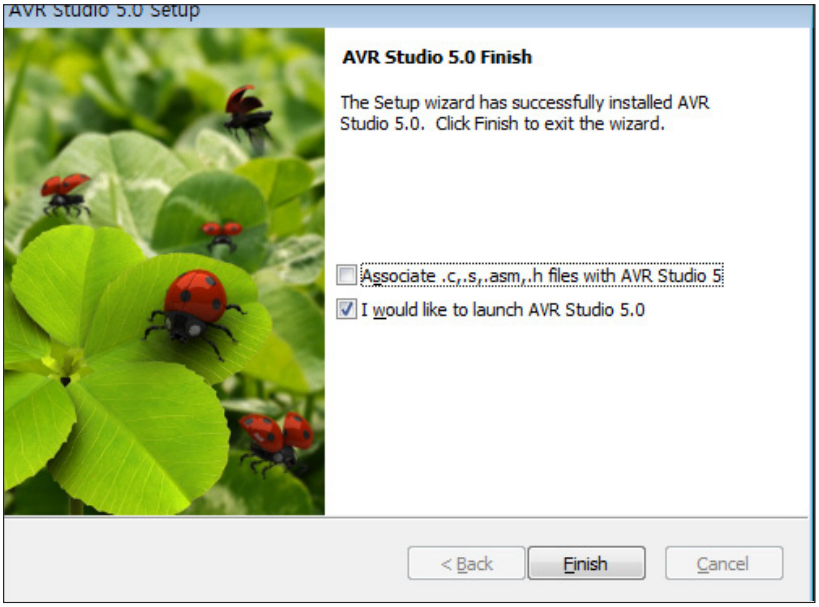
２０ 設置内容の確認

設置する内容を確認した後“Next”をクリックします。



２１ 設置開始

設置を開始します。



22 設置完了

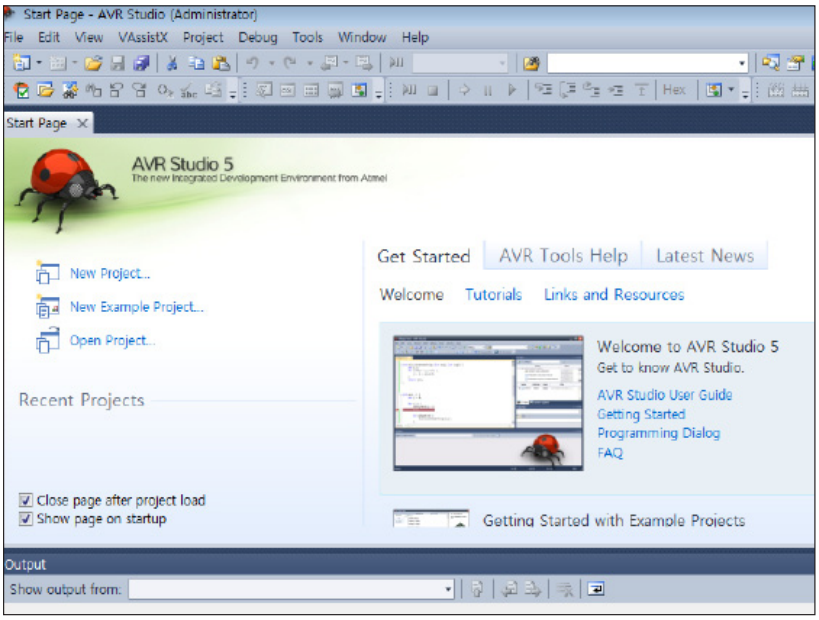
設置が完了しました。
“I would like to launch AVR Studio5.0”をチェックした後、“Finish”をクリックすると、プログラムが実行されます。



プロジェクトの作成やコーディング

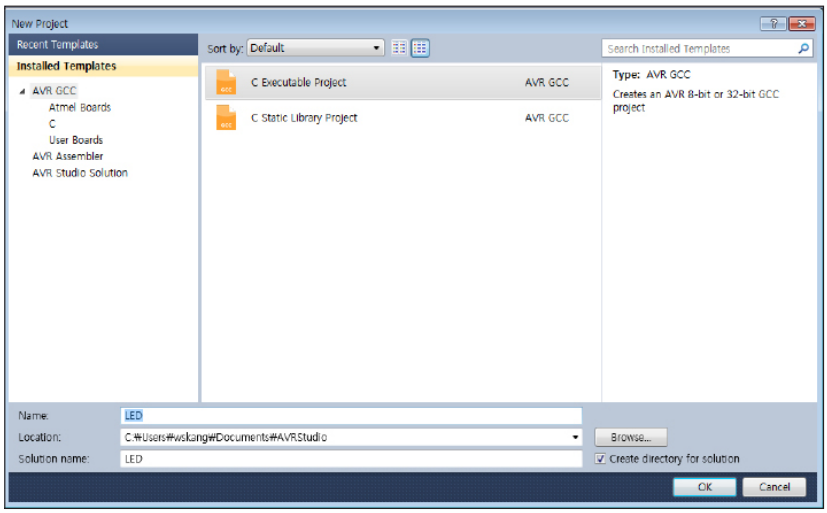
01 実行

AVR Stdioを実行します。



02 New Project

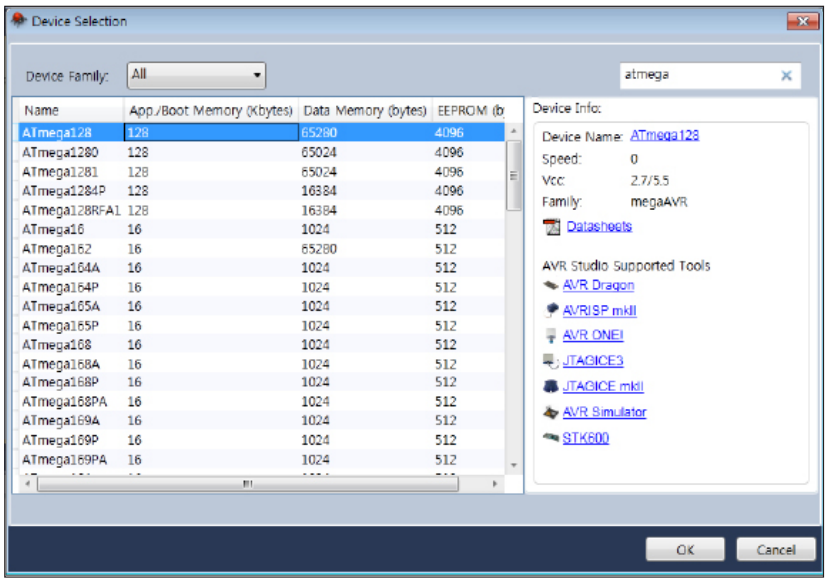
New Project をクリックします。



03 GCC 選択

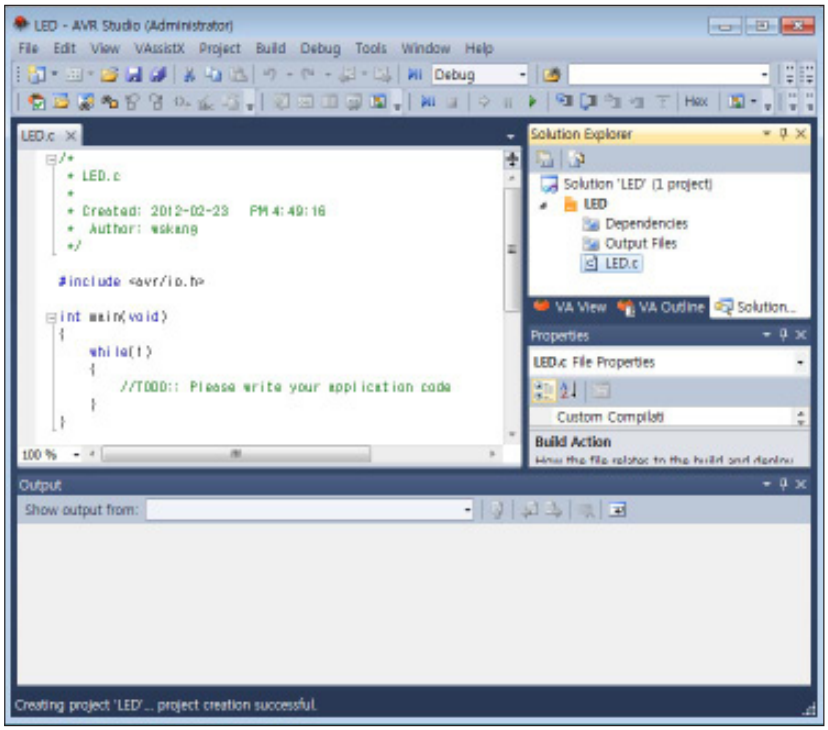
C Executable Project を選択します。

Nameに“LED”と入力し、Create directory for solutionをクリックし、OKボタンをクリックします。



04 ATmega 選択

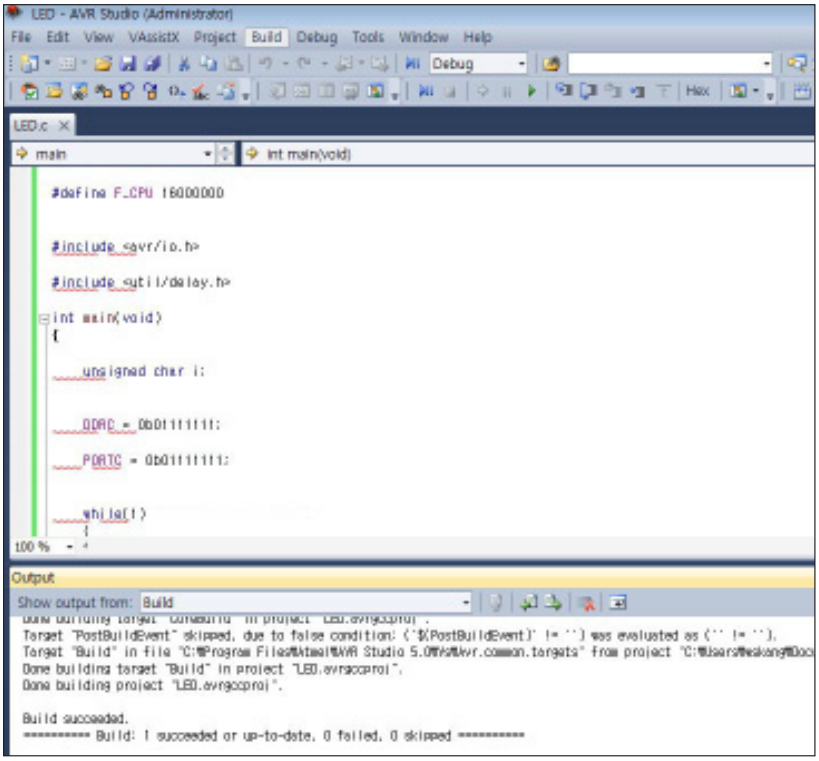
右上からatmegaを検索し、リスト中にATmega128を選択します。

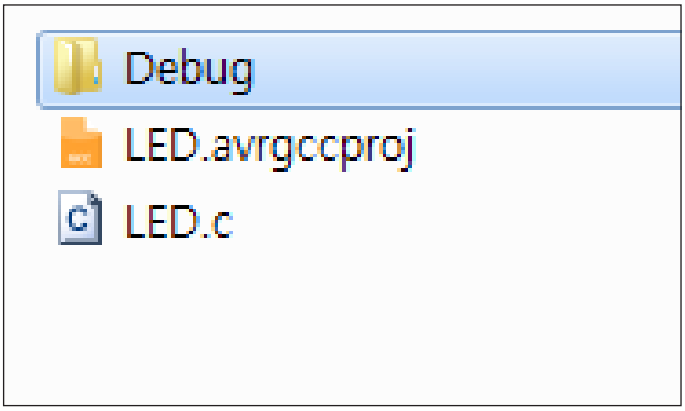


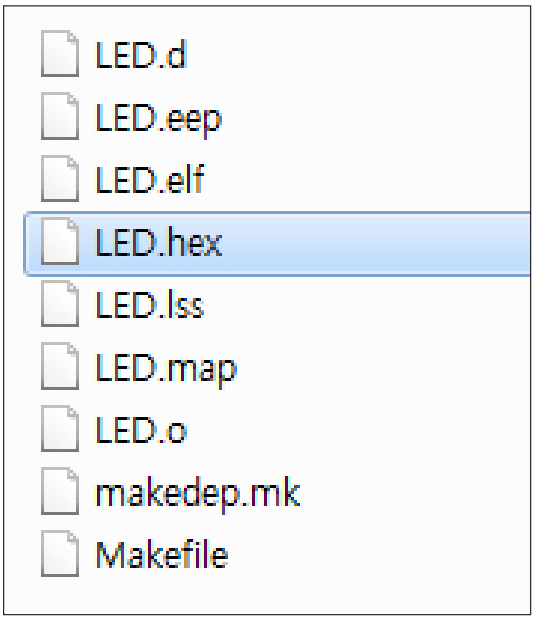
05 Editor

編集ウィンドウのコーディングを開始します。

該当する例を作成します。
次項の例は、LEDの例です。







Build や実行

01 Build

Buildメニューを選択してBuild SolutionをクリックしてBuildします。

1 succeded 0 failed が出ればエラーなしでコンパイルが完了したということです。

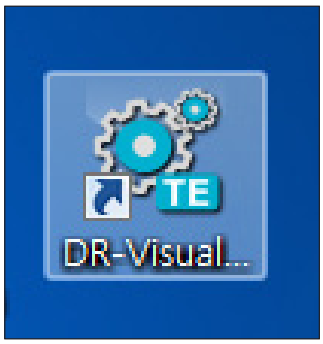
02 Debug フォルダ

Build後にソースがあるフォルダに入るとDebugフォルダが作成されています。クリックしてフォルダの中身を見ます。

03 hex 確認

Debugフォルダ内のhexファイルを確認します。Hexファイルはコントローラに移植するファイルです。

DR-Visual Logic のファームウェア修復を行います。



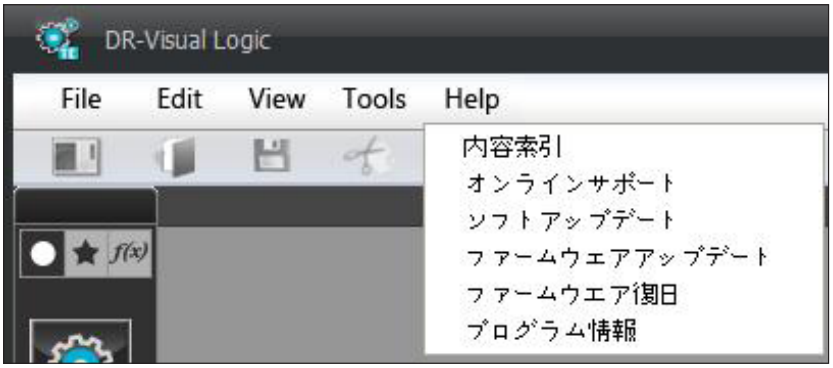
04 Visual Logic

DR–Visual Logic を実行します。



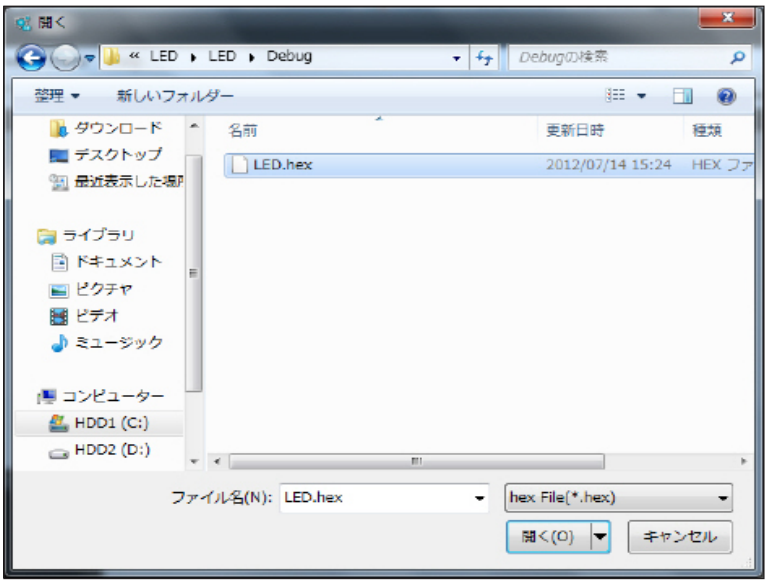
05 接続する

COM　番号を設定し、接続をクリックします。



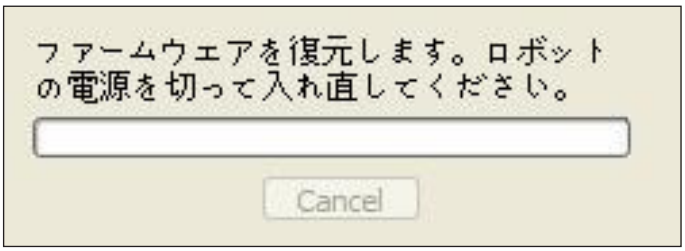
06 ファームウェアの修復

[ヘルプ]メニューから、ファームウェアの修復をクリックします。



07 ロード

DebugフォルダにHex ファイルをロードします。



08 ファームウェアの修復

“ロボットの電源を切って入れ直してください”というメッセージが出ます。
コントローラの電源ボタンを一度切ってから再度入れ直してください。



09 修復開始

ファームウェアの修復が開始されます。



10 実行

Servo LEDから Power LEDまでLEDが一つずつ点灯します。LED7個すべて点灯した後Servo LEDのみ点灯し、順番に一つずつ点灯することを繰り返します。

LED 制御する

ATmega128 のデジタル出力を使用してDRCに装着されたLEDを制御します。

AVR 原理

AVRには、デジタル入出力(I/O)ポートが組み込まれており、デジタル信号(HIGHとLOW)を入力したり送信したりできます。同じピンの入力を受け取るか出力するかを自由に設定できるので、LED制御、ボタンの入力、デジタルデータ転送など、さまざまに使用されます。 ATmega128に内蔵されているI/Oポートは、AからGまでの7個あります。また、AからFまでの8つのピンがあり、Gには、5つのピンがあるため合計53個のピンを自由にデジタル入出力として使用できます。この53個のピンは、汎用入出力機能に加え、それぞれ特殊な機能を持っており、レジスタの設定によっては、この機能を使用するかどうかを決定できます。
デジタル出力に関するレジスタは以下のとおりです。 ATmega128のレジスタはすべて8ビット(1バイト)で構成されています。 DDRXとPROTXでXは、ポート名(A～G)を示す文字です 。

■ DDRX：PORT X データ方向レジスタ

ビット	7	6	5	4	3	2	1	0
	DDX7	DDX6	DDX5	DDX4	DDX3	DDX2	DDX1	DDX0
読み取り/書き込み	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初期値	0	0	0	0	0	0	0	0

■ PORTX：PORT X データ方向レジスタ

ビット	7	6	5	4	3	2	1	0
	DDX7	DDX6	DDX5	DDX4	DDX3	DDX2	DDX1	DDX0
読み取り/書き込み	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初期値	0	0	0	0	0	0	0	0

DDRXレジスタは、ポートXの入出力方向を決定するレジスタであり、ビット値が0の場合は入力、1の場合は出力に設定されます。
PORTXレジスタは、ポートXに出力する論理値を格納するレジスタです。DDXn値が1(出力)に設定されている場合は、ポートXのn番ピンPORTXnに設定された値を出力します。

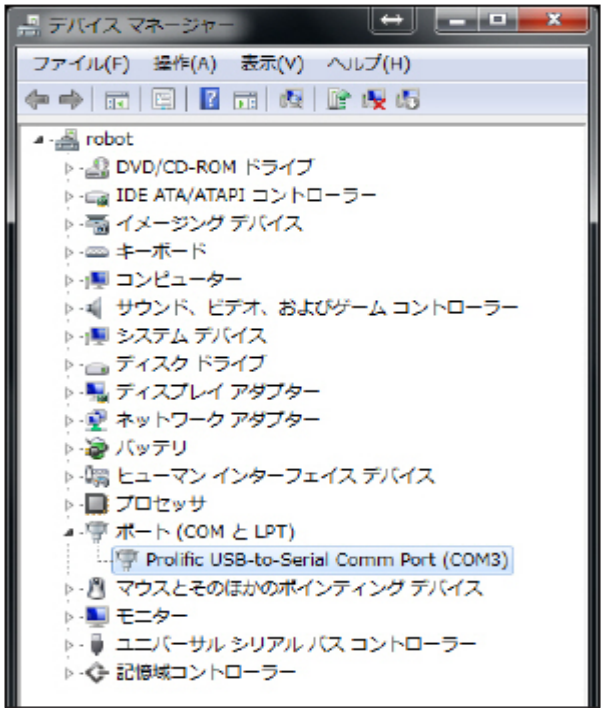
H/W 原理

ダイオードは、2つの電極を持ち、順方向電圧は電流が流れるが、逆方向電圧は電流がほとんど流れない特性を持っています。LEDはLight Emitting Diodeの略です。順方向に電圧が加わると電流が流れて点灯し(図00)、逆方向に電圧が加わると消灯します(図00)。ATmega128でLEDを制御する回路は、Active Low(図00)とActive High(図00)の二通りの方法があります。Active Low方式では、ピンの出力がLowのとき、LEDの順方向電圧がかかって点灯します。 Active High方式では、逆にHighのとき、LEDが点灯します。 DRCでは、PORT Cの0～6番ピンをLEDの制御に使用し、Active Lowの方法で回路が構成されています(図00)。ピン番号に対応するLEDの名前は、下表のとおりです。

ピン番号	7	6	5	4	3	2	1	0
LED 名	-	Power	Spare	RX	TX	EXEC	Program	Servo

準備





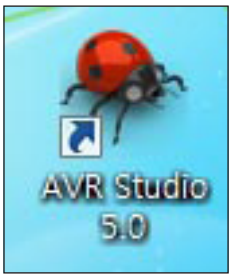
ハードウェアの準備

01 接続

DRC制御機アダプターで電源を接続し、USB to Serial CableにPCとDRCを接続します。

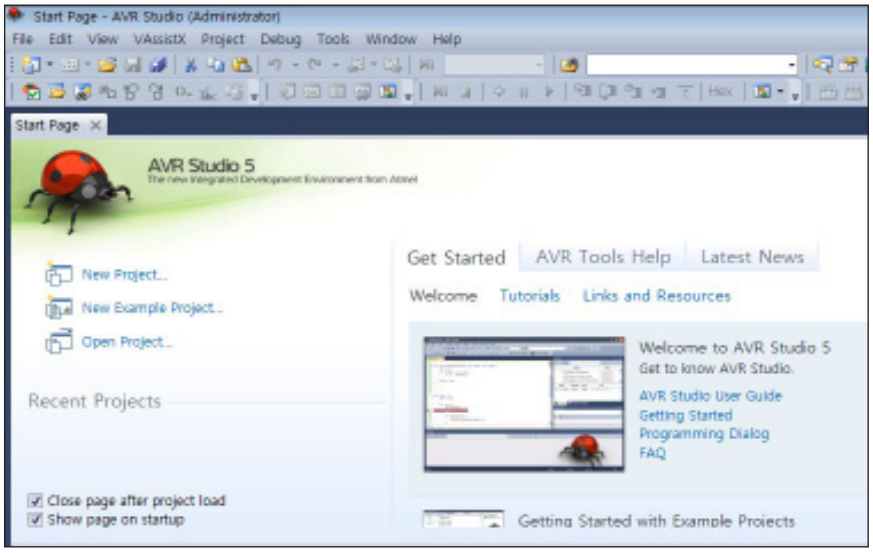
02 COM port 接続

マイコンピュータ>プロパティ>デバイスマネージャでCOM接続を確認します。



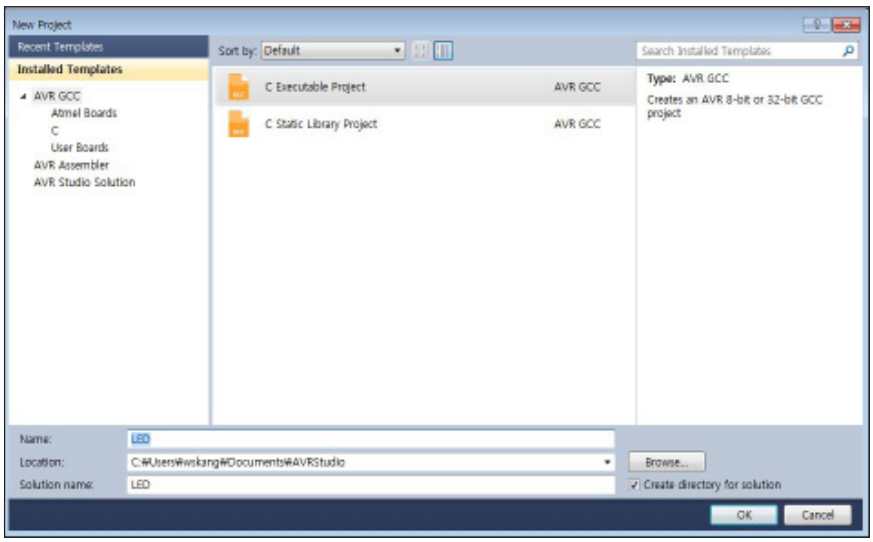
01 実行

AVR Stdioを実行します。



02 New Project

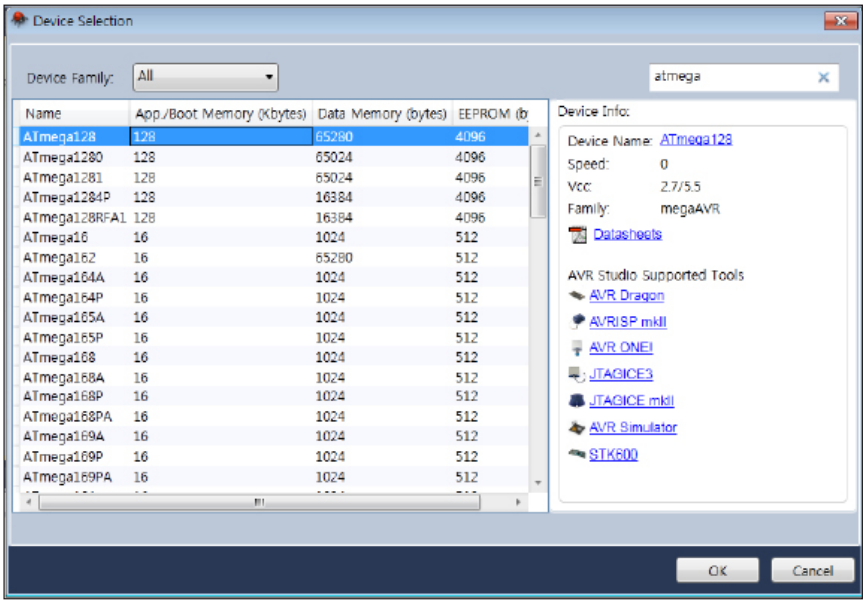
New Projectをクリックします。



03 GCC 選択

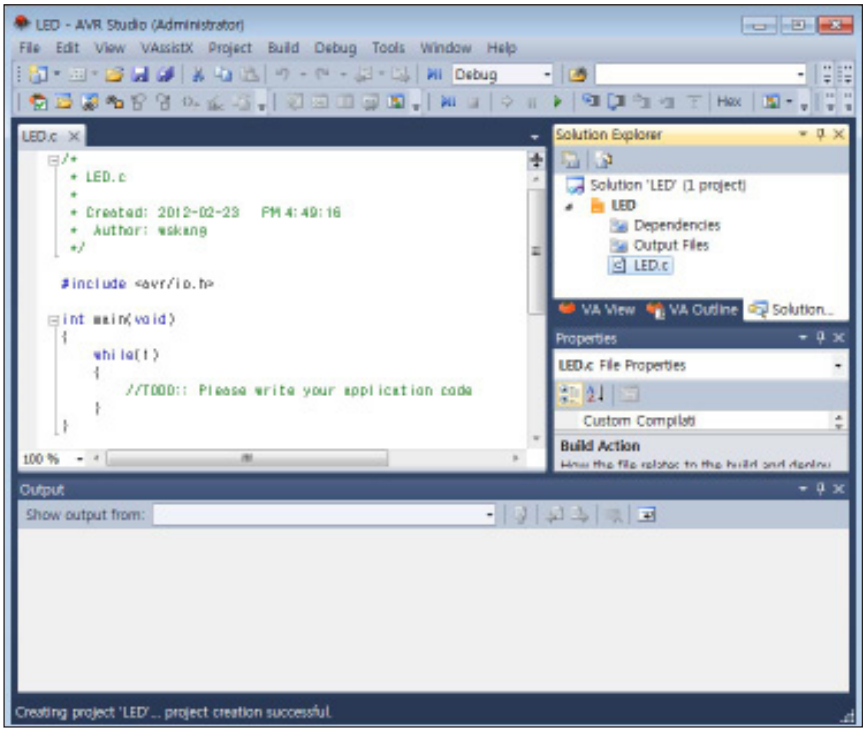
C Executable Projectを選択します。

Nameを“LED”に入力してCreate directory for solutionをクリックし OKバトンをクリックします。



04 ATmega 選択

右上からatmegaを検索し、リスト中のATmega128を選択します。



05 Editor

編集ウィンドウでのコーディングを開始します。

次の例を見てください。

例

DRCのLEDがServo(0番ピン)からPower(6番ピン)まで順に点灯していることを繰り返す例です。

```
//CPU クリック設定
#define F_CPU 16000000

//ATmega128のレジスタなどが定義されている
#include <avr/io.h>
//_delay_ms などディレイ が定義されている
#include <util/delay.h>

int main(void)
{
	//8ビット の宣言
	unsigned char i;

	//ポートCの入出力方向を設定する
	DDRC = 0b01111111;
	//ポートCの出力値を設定する
	PORTC = 0b01111111;

	//while の 容を無限に繰り返す
	while(1)
	{
		//LEDが順番に一つずつ点 するように動作を繰り返す
		for(i=0b00000001;i<=0b01111111;i=(i<<1)+1){
			PORTC = ~i;		//Active Lowなので、 iの値を反 して出力
			_delay_ms(200);		//200ms 待機
		}
	}

	// の形に合わせて整 値を返す
	return 1;
```

例の説明

```
#define F_CPU 16000000
```

#define文はコードの前処理命令の一つで、今後F_CPUという値が出ると16000000に置き換えられることを意味しています。DRCのATmega128は16MHzのクロック速度で動作します。クロック速度に比例して命令の処理速度が異なりますので、この命令を通じて、このプログラムが動作する速度を指定することでdelay関数などが正常に動作します。

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

avr/io.h ヘッダファイルにはDDRX、PORTXレジスタなどAVR制御のために必要なレジスタなどが定義されており、util/ delay.hヘッダファイルには、_delay_ms()などのよく使うディレイ関数が定義されているので # include文を使用していています。

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

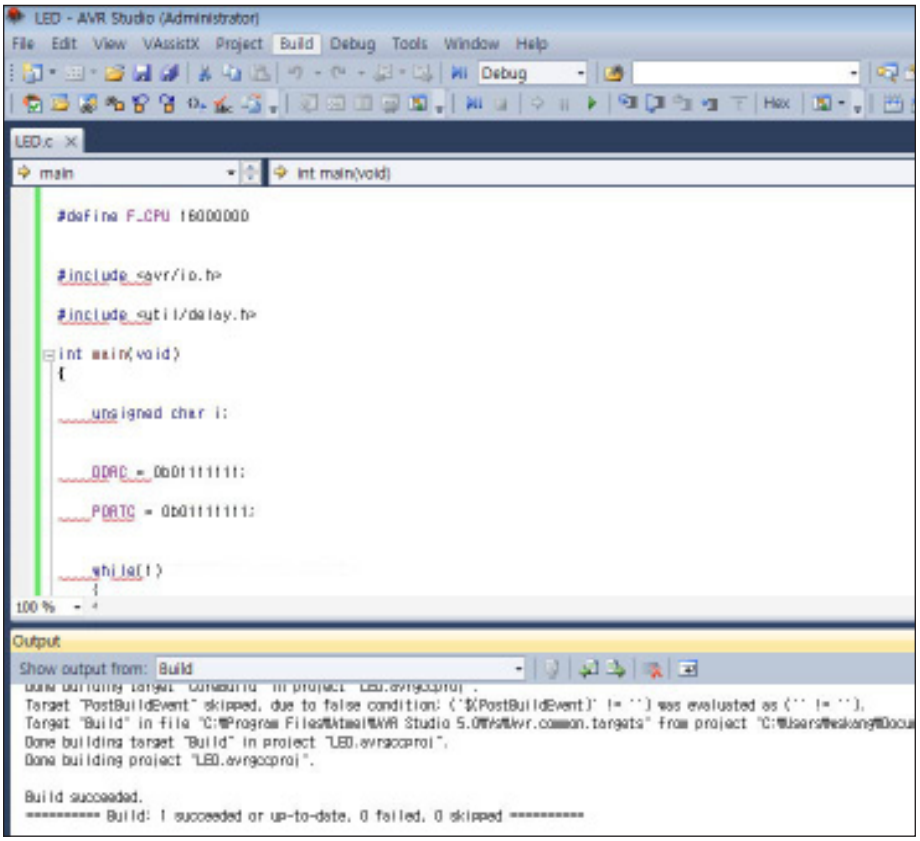
ポートCの0～6番ピンを出力に設定し、出力値を1にします。これにより、すべてのLEDが消灯します。ここで、0b01111111はC言語で2進数を表します。

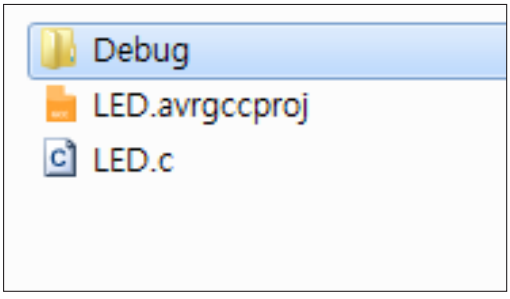
```
for(i=0b00000001;i<=0b01111111;i=(i<<1)+1){
        PORTC = ~i;
        _delay_ms(200);
```

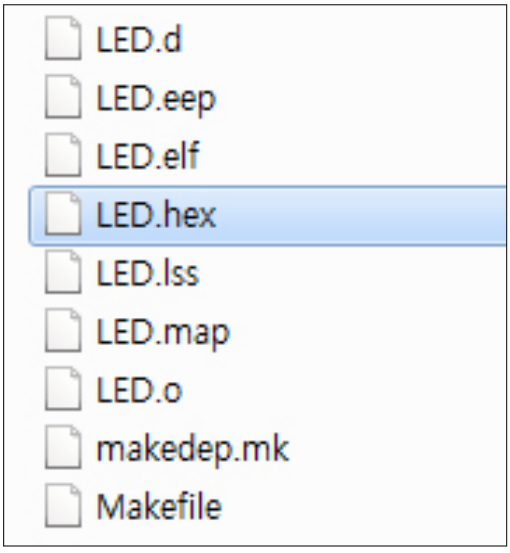
for文でiは0b00000001に初期化されます。これを～(bitwise not)演算子を用いて処理すると、0b11111110になり、これがPORTCの値に入ると0番のビットに対応するLEDが点灯します。_delay_ms()関数を通じて200msほど待った後iはi=(i<<1)+1に更新されます。この演算は、iの値を1ビット左にshiftして1を加えます。たとえば、0b00001111はshiftされ0b00011110になって1を加えると0x00011111になります。このようにiの値に1のビットを一つ追加します。したがって、for文を実行すると0番のビットから始めて順番にLEDが一つずつ点灯します。for文はi<=0x01111111を満たしている間に実行されるため、すべてのLEDが点灯したら終了します。このfor文はwhile(1)文に囲まれているので、再びfor文の最初から開始され、LEDは0番目のビットから順に点灯します。

実行

ダウンロードして実行するとServo LEDからPower LEDまでLEDが一つずつ点灯します。LED7個が点灯した後は再びServo LEDが点灯し、順番に一つずつ点灯することを繰り返します。







01 Build

Buildメニューを選択してBuild SolutionをクリックしてBuildします。

1 succeeded0 failedが出るとエラーなしでコンパイルが完了したということです。

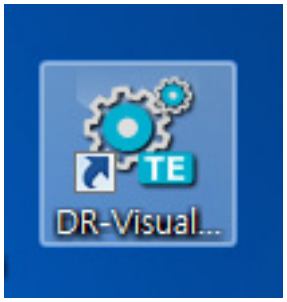
02 Debug フォルダ

Build後にソースがあるフォルダに入るとDebugフォルダが作成されています。クリックしてフォルダの中身を表示します。

03 hex 確認

Debugフォルダの中のhexファイルを確認します。Hexファイルは制御機に移植するファイルです。

DR-Visual Logicで、ファームウェアのリカバリーを行います。



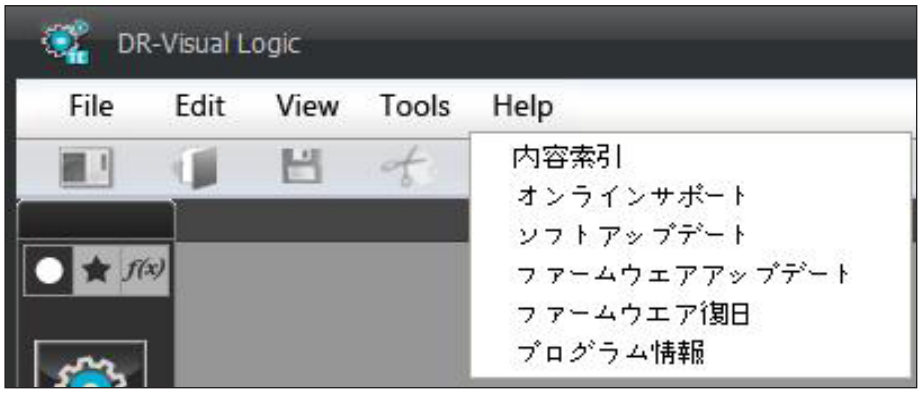
04 Visual Logic

DR-Visual Logicを実行する。



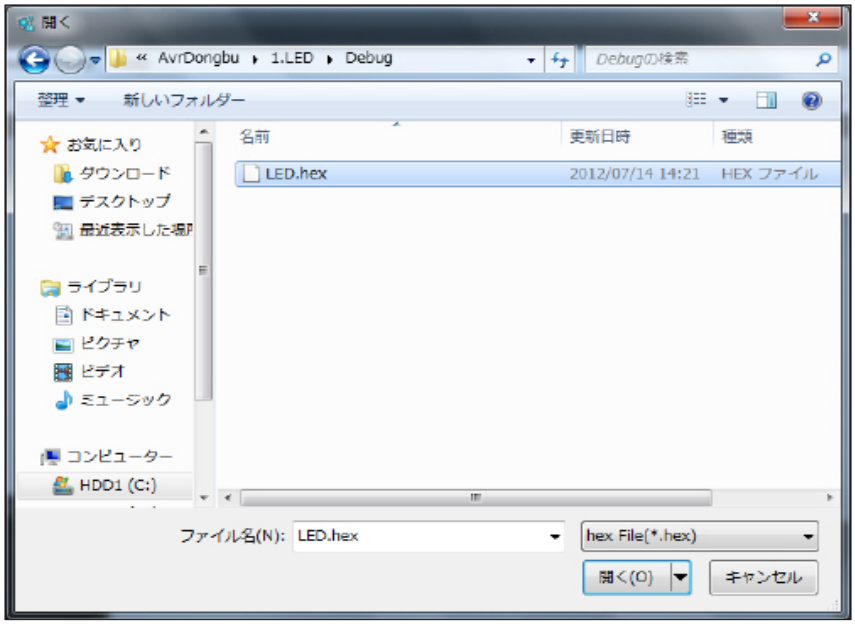
05 接続する

COM 番号を設定し、接続をクリックします。



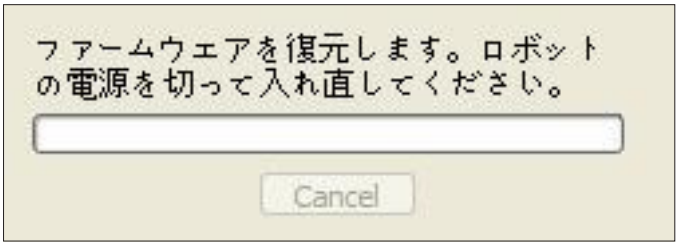
06 ファームウェアの修復

[ヘルプ]メニューからファームウェア修復をクリックします。



07 ロード

DebugフォルダにHexファイルをロードします。



08 ファームウェアの修復

Buildメニューを選択してBuild SolutionをクリックしてBuildします。

1 succeeded0 failedが出るとエラーなしでコンパイルが完了したということです。



09 修復の開始

ファームウェアの修復が開始されます。



10 実行

Servo LEDからPower LEDまでLEDが一つずつ点灯します。LED7個が点灯した後は再びServo LEDが点灯し、順番に一つずつ点灯することを繰り返します。

LED 制御する

ATmega128のデジタル出力を使用してDRCに装着されたLEDを制御します。

AVR 原理

前のChapter1では、デジタル入出力ポートの機能の出力を利用してLEDを制御しました。今回はボタンで入力を受け、その結果に応じてLEDを制御する方法を説明します。

デジタル入出力に関連するレジスタは以下のとおりです。ATmega128のレジスタはすべて8ビット(1バイト)で構成されています。DDRXとPROTXでXは、ポート名(A～G)を示す文字です。

■ DDRX : PORT X データ方向レジスタ

ビット	7	6	5	4	3	2	1	0
	DDX7	DDX6	DDX5	DDX4	DDX3	DDX2	DDX1	DDX0
読み取り/書き込み	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初期値	0	0	0	0	0	0	0	0

■ PORTX : PORT X データレジスタ

ビット	7	6	5	4	3	2	1	0
	PORTX7	PORTX6	PORTX5	PORTX4	PORTX3	PORTX2	PORTX1	PORTX0
読み取り/書き込み	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初期値	0	0	0	0	0	0	0	0

■ PINX : PORTX 入力ピンアドレス

ビット	7	6	5	4	3	2	1	0
	PINX7	PINX6	PINX5	PINX4	PINX3	PINX2	PINX1	PINX0
読み取り/書き込み	R	R	R	R	R	R	R	R
初期値	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Chapter1で説明したように、DDRXのレジスタは、ポートXの入出力方向を決定するレジスタであり、PORTXレジスタは、ポートXに出力する論理値を格納するレジスタです。 PINXレジスタは、現在のポートの論理値を読み取ることができるアドレスです。入出力するかどうかにかかわらず、PINXレジスタを読み取ることで、PORT Xの現在の論理値を読み取ることができます。PINXには値を書き込むことができず、ただ読むだけです。

H/W 原理

タクトスイッチ(tact switch)は、プッシュスイッチ(push switch)の一種で、動作時のカチッという感触に特徴があるため、英語で"触覚を利用した"という意味のtactileという言葉からタクトスイッチと呼ばれる名前を持つようになりました。 タクトスイッチの両方のピンは、普段は離れているがカチッと押された状態では、接続されて電流が流れます。タクトスイッチは2ピンと4ピンがありますが、4ピンのスイッチも内部的には2ピンずつ接続されており、2ピン スイッチ と動作の違いはありません。

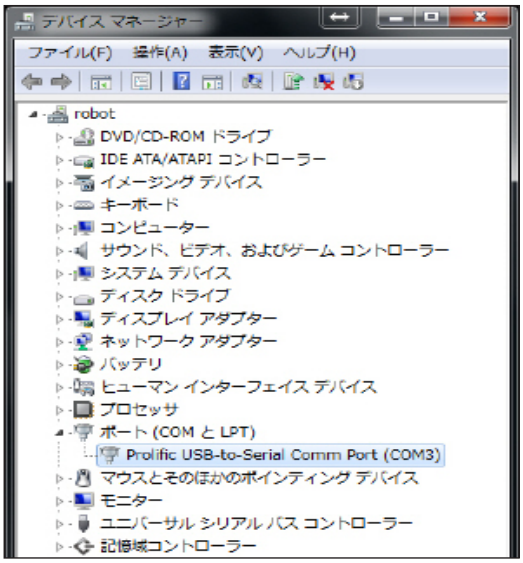
タクトスイッチをATmega128に入力する回路は以下のとおりです(図00)。スイッチが押されていない場合、回路のコンデンサCは、抵抗Rを通って充電され、デジタル入力ピンにVCCと同じ電圧が入ります(図00)。したがって、スイッチが押されていない場合、入力値がHIGHになります。スイッチが押された場合、デジタル入力ピンはGNDに直接接続されてコンデンサCが放電されます(図00)。デジタル入力ピンにGND電圧が入るので、入力値がLOWになります。この回路では抵抗Rがない場合は、スイッチが押されなかったら、入力ピンは接続された電圧がないフローティング状態になります。このような状態はHIGH、LOWどちらの状態もありません。

このときの抵抗Rは、VCCと入力ピンを接続することで、入力ピンの電圧値をHIGHに引き上げてくれる役目をします。したがって、この場合、抵抗Rをプルアップ(pull-up)抵抗といいます。DRCは、PORT Aの0～5番ピンのボタン入力に使用し、ピン番号に対応するLEDの名前は、下表のとおりです。

ピン番号	7	6	5	4	3	2	1	0
LED 名	-	-	Right	Left	Down	Up	OK	Mode

準備





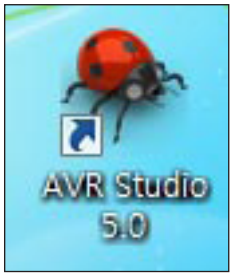
ハードウェアの準備

01 接続

DRC制御機アダプターで電源を接続し、USB to Serial CableでPCとDRCを接続します。

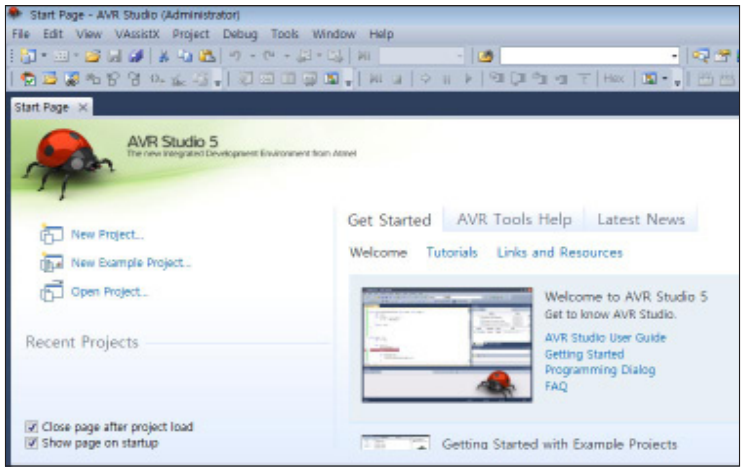
02 COM port 接続

マイコンピュータ>プロパティ>デバイスマネージャでCOM接続を確認します。



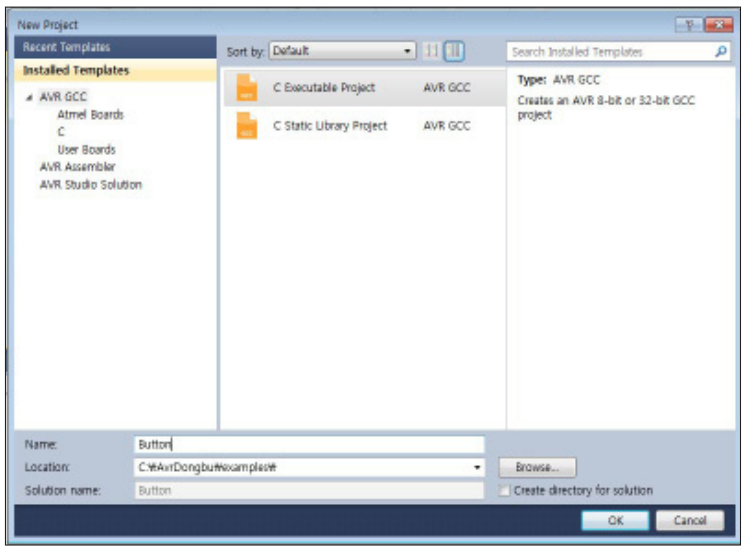
01 実行

AVR Stdioを実行します。



02 New Project

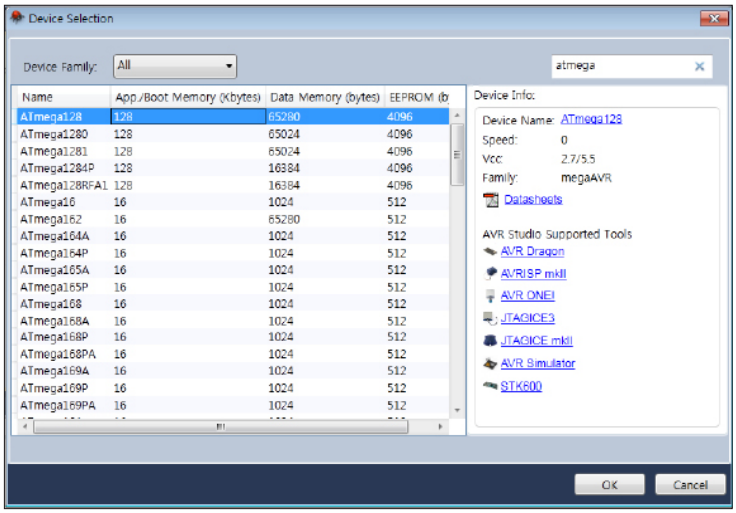
New Projectをクリックします。



03 GCC 選択

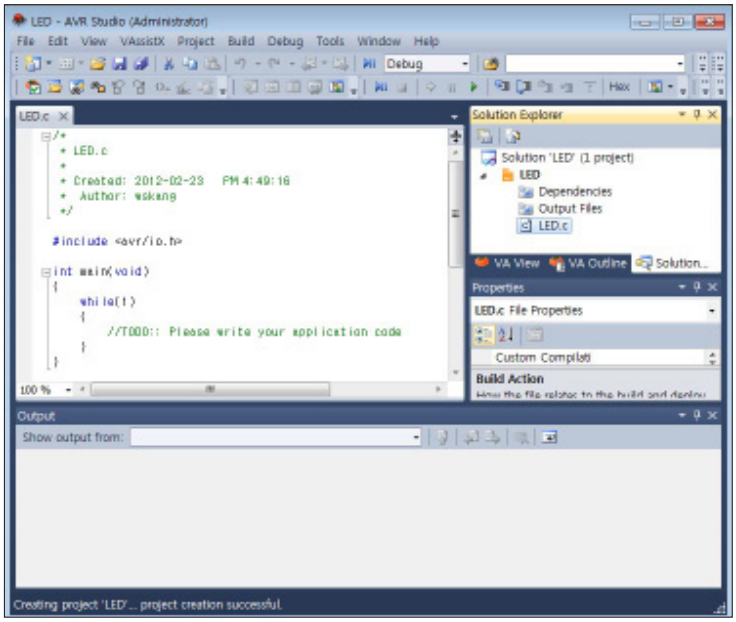
C Executable Projectを選択します。

Nameを“Button”に入力して,
Create directory for solutionをクリックし, OK ボタンを押します。



04 ATmega 選択

右上からatmegaを検索し、リスト中のATmega128を選択します。



05 Editor

編集ウィンドウでコーディングを開始します。

次の例を見てください。

例

DRCのボタンを押すと、そのボタンに対応する位置のLEDが点灯する例です。押すボタンに応じて点灯するLEDは、下表のとおりです。

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

```
//ATmega128のレジスタなどが定義されている
#include <avr/io.h>

//LEDの名前に  するポートCのピン定義
#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

//BTNの名前に  するポートAのピン定義
#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)
{
```

```
//8ビットの  の宣言
        unsigned char ucButton, ucLED;

        //ポートAの入出力方向を設定する
        DDRA = 0b00000000;

        //ポートCの入出力方向を設定する
        DDRC = 0b01111111;
        ///ポートCの出力値を設定する
        PORTC = 0b01111111;

        //while の 容を無限に繰り返す
        while(1)
        {
                //ポートAからボタンの値の み出し
                ucButton = (~PINA) & BTN_MASK;
                //ucLEDをLEDがすべてオフの 態に初期化
                ucLED = 0;

                //MODE ボタンを押したら PROGRAM LEDをオン
                if(ucButton & BTN_MODE){
                        ucLED |= LED_PROGRAM;
                }
                //OK ボタンを押したら EXEC LEDをオン
                if(ucButton & BTN_OK){
                        ucLED |= LED_EXEC;
                }
                //UP ボタンを押したら TX LEDをオン
                if(ucButton & BTN_UP){
                        ucLED |= LED_TX;
                }
                //DOWN ボタンを押したらRX LEDをオン
                if(ucButton & BTN_DOWN){
                        ucLED |= LED_RX;
```

```
    }
                    //LEFT ボタンを押したらSERVO LEDをオン
                    if(ucButton & BTN_LEFT){
                            ucLED |= LED_SERVO;
                    }
                    //RIGHT ボタンを押したら POWER LEDをオン
                    if(ucButton & BTN_RIGHT){
                            ucLED |= LED_POWER;
                    }

                    //ucLEDの値にあわせ PORTCの値を 更する
                    PORTC = (~ucLED) & LED_MASK;
            }

    //  の形に合わせて整 値を返す
            return 1;
    }
```

例の説明

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

avr/io.hヘッダファイルにはDDRX、PORTXレジスタなどAVR制御のために必要なレジスタなどが定義されているので#include文を使用しています。

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

#define文はコードの前処理命令の一つです。# defineLED_SERVO0b00000001は今後LED_SERVOが出れば0b00000001に置き換えてコンパイルするという意味です。DRCのボタンとLEDは、それぞれのビットに対応する名前がありますが、毎回そのボタンとLEDを扱うたびに0b00000001のような見苦しい数字を使用するよりもLED_SERVOに対応する名前を使用する方がプログラムを作成しやすいので、宣言します。

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

8ビットの変数ucButtonとucLEDを宣言します。これらの変数はそれぞれ、現在のボタンとLEDの状態を示す目的で使用されます。while文の内部でucButtonとucLEDは、実際のポートの入出力とは逆に保存されます。つまり、PINAとは異なり、ucButtonでボタンが押された状態が1で、押されていない状態が0です。また、PORTCとは異なり、ucLEDでLEDが点灯した状態が1であり、オフの状態が0です。

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

ポートAの0～5番ピンを入力に設定します。ボタンに対応するピンを入力に設定することです。ポートCの0～6番ピンを出力に設定し、出力値を1にします。これにより、すべてのLEDが消灯します。

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

while文はucButtonからボタンの値を読み取ることから始めます。PINAで読んだボタンの値は、ボタンが押されたとき0、押されていないときに1であるため、～演算子を処理して逆に変えてくれます。そしてBTN_MASK(0b00111111)と&(bitwise AND)処理をして、ボタンとは関係のない6、7番目のビットを0にしています。これにより、ucButtonには0b00XXXXXX値が入り、Xには、各ビットに対応するボタンが押されかどうかが保存されます。その後ucLEDをLEDがすべて消灯状態にある0に初期化します。

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

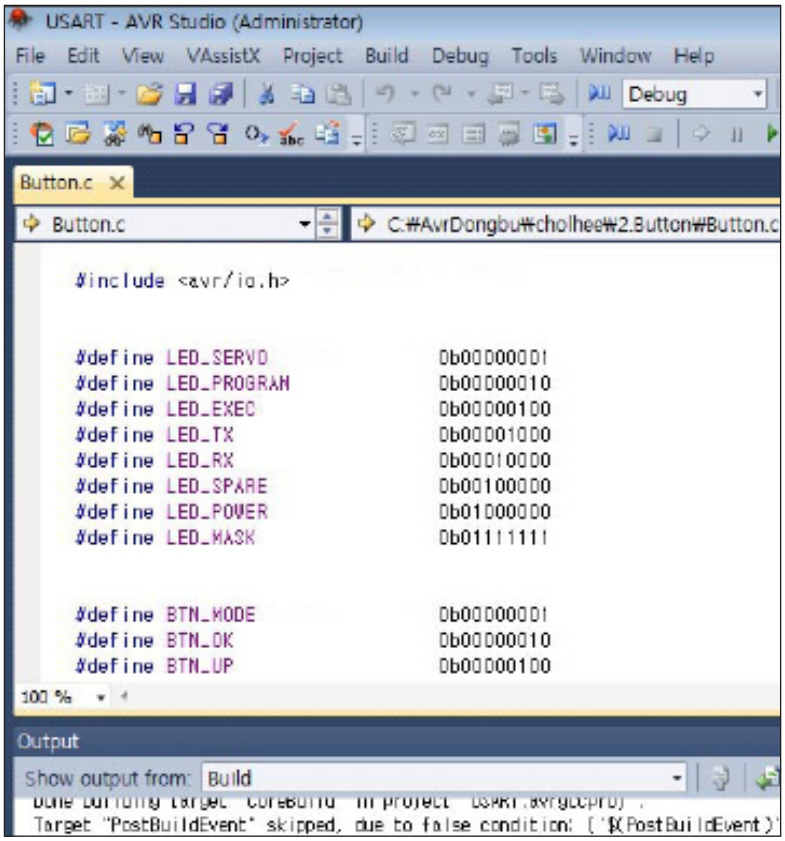
if(ucButton&BTN_MODE)という条件文は、ucButton値BTN_MODE(0b00000001)と&演算します。結果値はucButtonの0番のビットが1の場合（Modeボタンが押されたら）0b00000001、0であればそれだけで0になります。MODEボタンを押された場合if文の括弧内の値が真であるため、中身が実行されます。if文の中のucLED |= LED_PROGRAMはucLED= ucLED | LED_PROGRAMの簡単な表現です。ucLED値LED_PROGRAM(0b00000010)を|(bitwise OR)処理してucLED値に変えることを意味しています。この計算の結果としてucLEDのProgram LEDに対応する1番のビットが1になります。その後のif文も同じです。Modeボタンが押されたときucLEDの値ProgramLEDに対応するビットが1になるように、OKボタンを押した場合EXEC LED、UPボタンが押されたらTX LED、DOWNボタンが押されたらRX LED、...に対応するビットが1になります。

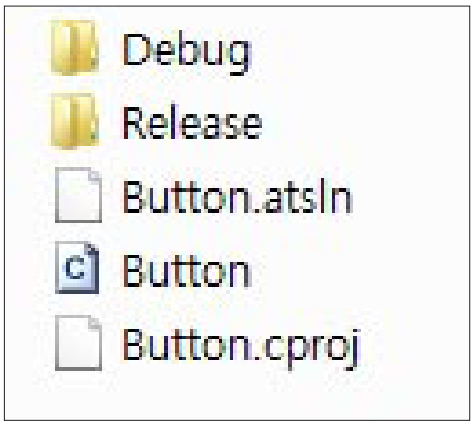
```
PORTC = (~ucLED) & LED_MASK;
```

while文の最後では、このボタン値に応じて変更されたucLED値を実際のポートCに適用します。ucLEDとPORTCのオンLEDを示す方法が逆なので～演算処理して、ボタンと同じようにLED_MASK(0b01111111)を&処理してLEDとは関係のない7番目のビットを0にしています。この値をPORTCに代入してLEDを制御します。while文が終わったら、再度、最初に戻って、ボタンの入力を読み取り、その値に応じてLEDを制御することを繰り返しています。

実行

ダウンロードして実行するとボタンを押すことによって対応するLEDが点灯します。







01 Build

Buildメニューを選択しBuild SolutionをクリックしてBuildします。

1 succeded0 failedが出るとエラーなしでコンパイルが完了したということです。

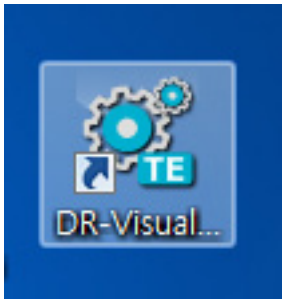
02 Debug フォルダ

Build後にソースがあるフォルダに入るとDebugフォルダが作成されています。クリックしてフォルダの中身を表示します。

03 hex 確認

Debugフォルダの中のhexファイルを確認します。Hexファイルは制御機に移植するファイルです。

DR-Visual Logicで、ファームウェアのリカバリーを行います。



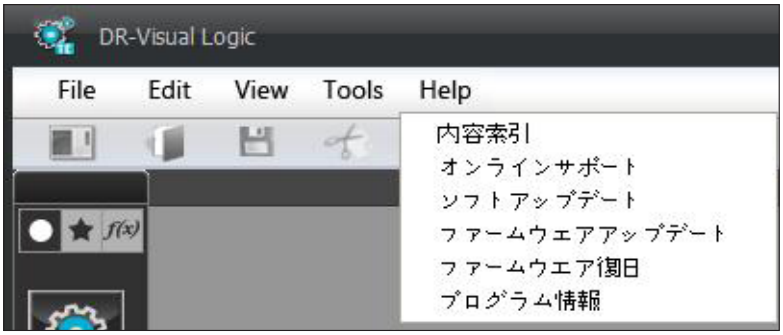
04 Visual Logic

DR-Visual Logicを実行します。



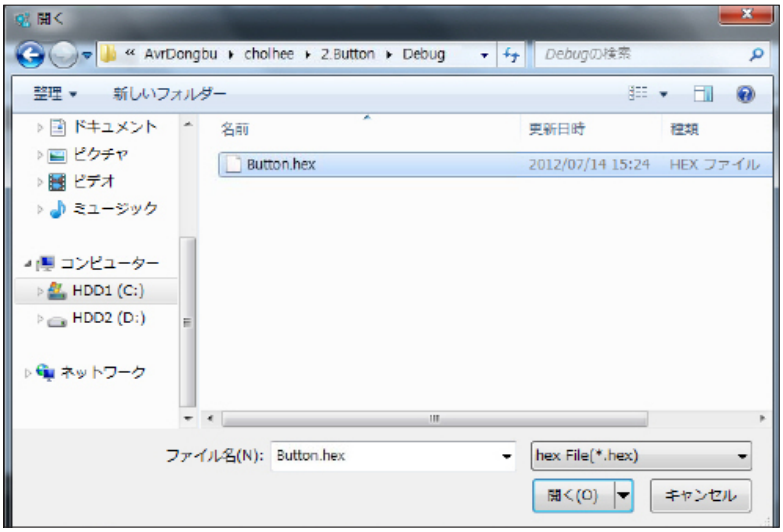
05 接続する

COM番号を設定して接続するをクリックします。



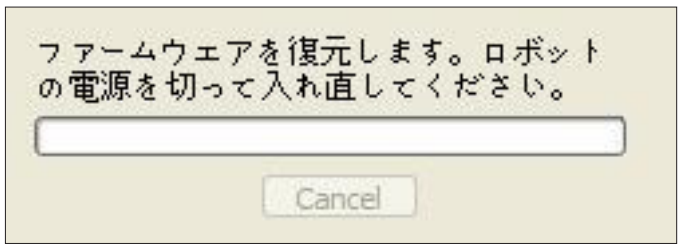
06 ファームウェアの修復

[ヘルプ]メニューからファームウェア修復をクリックします。



07 ロード

DebugフォルダにHexファイルをロードします。



08 ファームウェアの修復

“ロボットの電源を切って入れ直してください”というメッセージが出ます。コントローラの電源ボタンを一度切ってから再度入れ直してください。



09 修復の開始

ファームウェアの修復が開始されます。



10 実行

ボタンを押すことによって対応するLEDが点灯します。

ブザーを鳴らす

ATmega128のデジタル入出力を使用してDRCに装着されたブザーでビープ音を出します。

AVR 原理

前のChapter2では、ボタンで入力を受け、LEDを制御する方法を学びました。今回はLEDの代わりに、出力ピンでブザーを制御して音を鳴らすようにしてみます。デジタル入出力に関連するレジスタはDDRX、PORTX、PINXです。Chapter2の説明を参照して下さい。

H/W 原理

DRCに装着されたブザーの内部にはコイルが入っています。ブザーの電流が任意の周波数で周期的に変わると、コイルの磁場の強さも変わります。磁場の強さが変わるとブザーの振動板が鳴って、その震えによって音が出るようになります。ブザーをATmega128に接続する回路は以下のとおりです（図00）。ATmega128のポートBの7番ピン(PB7)がトランジスタを通ってブザーに接続されています。ブザーの消耗電流はATmega128が送信できる電流よりも大きいので、出力ピンにブザーを直接接続すると電流が不足して誤動作したり、ピンに損傷を与える可能性があります。そこで、出力ピンとブザーの間にトランジスタを挿入することで、PB7から少量の出力電流だけでブザーに多くの電流を流すことができます。
出力ピンの出力値を一定の周期で変えるとブザーに流れる電流も、そのような周期で変化することになり、その電流によってブザーの音が鳴ります。短い周期（高い周波数）で出力ピンを変えれば、ブザーの音も比例してより高い音を出します。
今回の例では、delay関数を使用してPB7を一定周期で変えて音を出しています。

準備



ハードウェアの準備

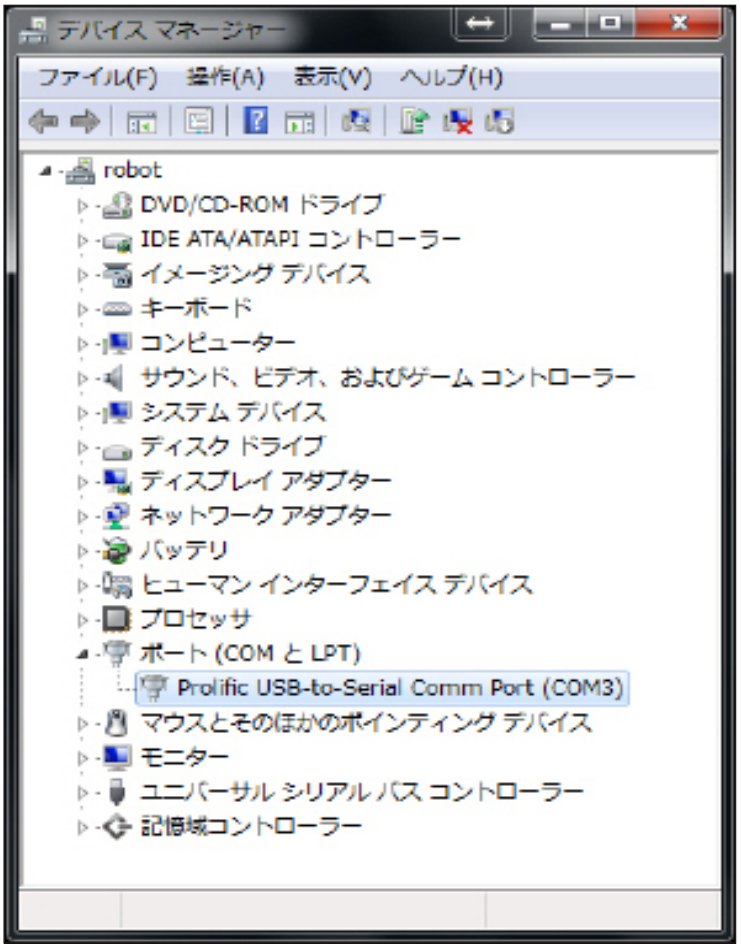
01 接続

DRC制御機アダプターで電源を接続し、USB to Serial CableでPCとDRCを接続します。



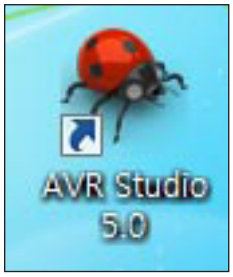
01 接続

DRC制御機アダプターで電源を接続し、USB to Serial CableでPCとDRCを接続します。



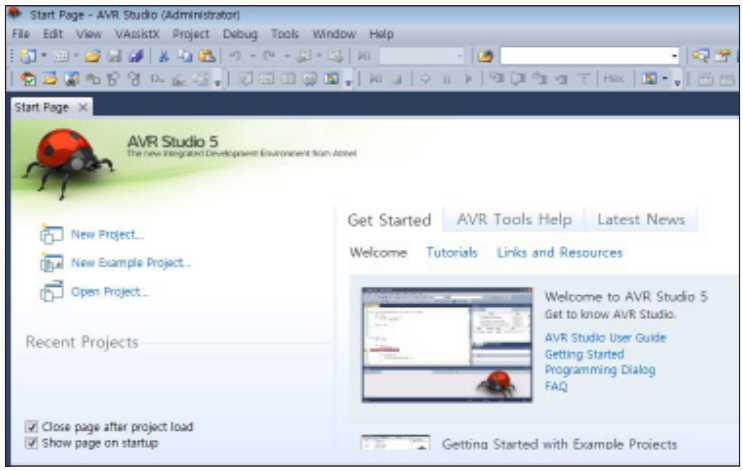
02 COM port 接続

マイコンピュータ>プロパティ>デバイスマネージャでCOM接続を確認します。



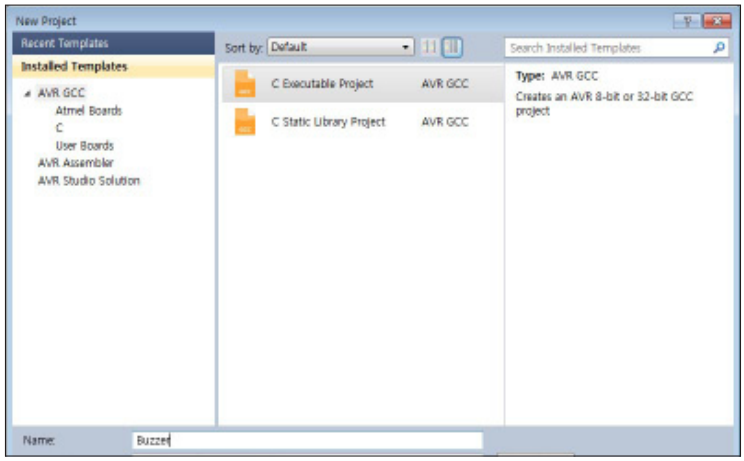
01 実行

AVR Stdioを実行します。



02 New Project

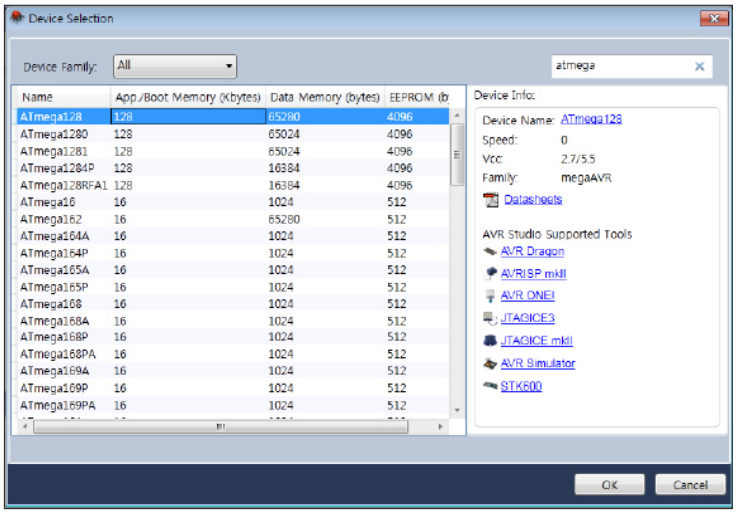
New Projectをクリックします。



03 GCC 選択

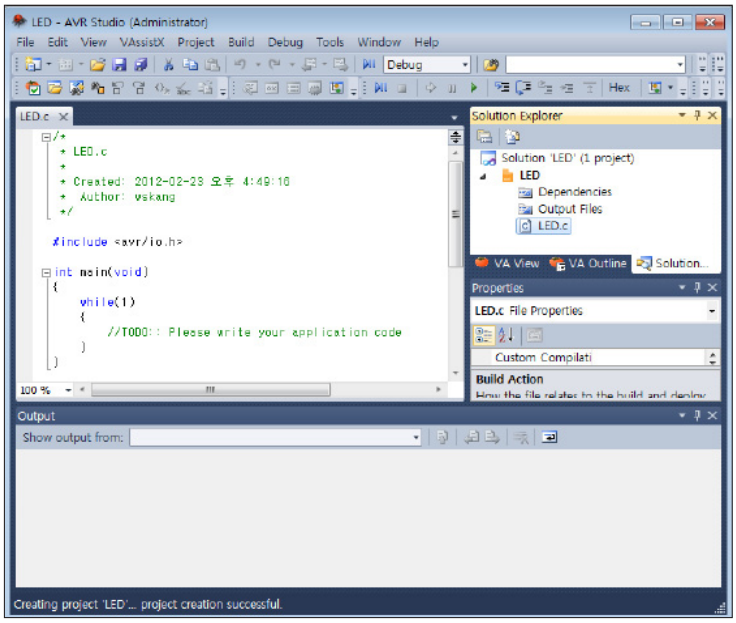
C Executable Projectを選択します。

Nameを“Button”に入力して,
Create directory for solutionをクリックし, OK ボタンを押します。



04 ATmega 選択

右上からatmegaを検索し、リスト中のATmega128を選択します。



05 Editor

編集ウィンドウでコーディングを開始します。

次の例を見てください。

例

DRCのOKボタンを押すとブザーが鳴る例です。

```
//CPU クロックの設定
#define F_CPU 16000000

//ATmega128のレジスタなどが定義されている
#include <avr/io.h>
//_delay_ms などディレイ が定義されている
#include <util/delay.h>

//BTNの名前に するポートAのピン定義
#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)
{
        //8ビットの の宣言
        unsigned char ucButton;

        //ポートAの入出力方向を設定する
        DDRA = 0b00000000;

        ///ブザーピン（PB7）の入出力方向を設定する
        DDRB |= 0b10000000;
        ///ブザーピン（PB7）の出力値を設定する
        PORTB &= ~0b10000000;
```

```
//ポートCの入出力方向を設定する
        DDRC = 0b01111111;
        ///ポートCの出力値を設定する
        PORTC = 0b01111111;

        //while の 容を無限に繰り返す
        while(1)
        {
                //ポートAからボタンの値の み出し
                ucButton = (~PINA) & BTN_MASK;

                //OKボタンを押されたらPB7の値を逆に えて1ms待機
                if(ucButton & BTN_OK){
                        PORTB ^= 0b10000000;
                        _delay_ms(1);
                }
        }

        // の形に合わせて整 値を返す
        return 1;
}
```

例の説明

```
#define F_CPU 16000000
```

#define文はコードの前処理命令の一つで、今後F_CPUという値が出れば16000000に置き換えられることを意味しています。DRCのATmega128は16MHzのクロック速度で動作します。クロック速度に比例して命令の処理速度が異なりますので、この命令を通じて、このプログラムが動作する速度を明示するとdelay関数が正常に動作します。

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

avr/io.hヘッダファイルにはDDRX、PORTXレジスタなどAVR制御のために必要なレジスタが定義されており、util/ delay.hヘッダファイルには、_delay_ms()などを使用してディレイ関数が定義されているので#include文を使用します。

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

#defineBTN_MODE 0b00000001は今後BTN_MODEが出れば0b00000001に置き換えてコンパイルするという意味です。DRCの各ボタンに対応するビットをボタンの名前に置き換えてプログラムを作成しやすいので、宣言します。

```
unsigned char ucButton;
```

8ビットの変数ucButtonを宣言します。この変数は、現在のボタンの状態を示す目的で使用されます。while文の内部でucButtonは、実際のポートの入出力とは逆に保存されます。つまり、PINAとは異なり、ucButtonでボタンが押された状態が1で、押されていない状態が0です。

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

ポートAの0～5番ピンを入力に設定します。ボタンに対応するピンを入力に設定します。そして、ポートBの7番ピンを出力に設定し、出力値を0にします。ポートCの0～6番ピンを出力に設定し、出力値を1にします。これにより、すべてのLEDが消灯します。

DDRBとPORTBには、他のレジスタとは異なり、=の代わりに|=、&=を使用しています。DDRBを0b10000000と|(bitwise OR)演算をすると、0のビットは、DDRBの元の値のまま維持され、1ビットだけ1に変わります。つまり、DDRBの7番ビットだけが1で作成し、残りはそのままにしたという意味です。一方では、PORTB &=～0b10000000;はPORTBを0b01111111(～0b10000000の結果)と&(bitwise AND)演算をするという意味です。1のビットは、PORTBの元の値のまま維持され、0のビットが0に変わります。つまり、PORTBの7番ビットを0にし、残りはそのままにしたという意味です。

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

while文はucButtonにボタンの値を読み取ることから始めます。PINAで読んだボタンの値は、ボタンが押されたとき0、押されないとき1であるため、～演算子を処理して逆に変えてくれます。そしてBTN_MASK(0b00111111)と&(bitwise AND)処理して、ボタンとは関係のない6、7番目のビットを0にしています。これにより、ucButtonには0b00XXXXXX値が入り、Xには、各ビットに対応するボタンが押されたかどうかが保存されます。

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

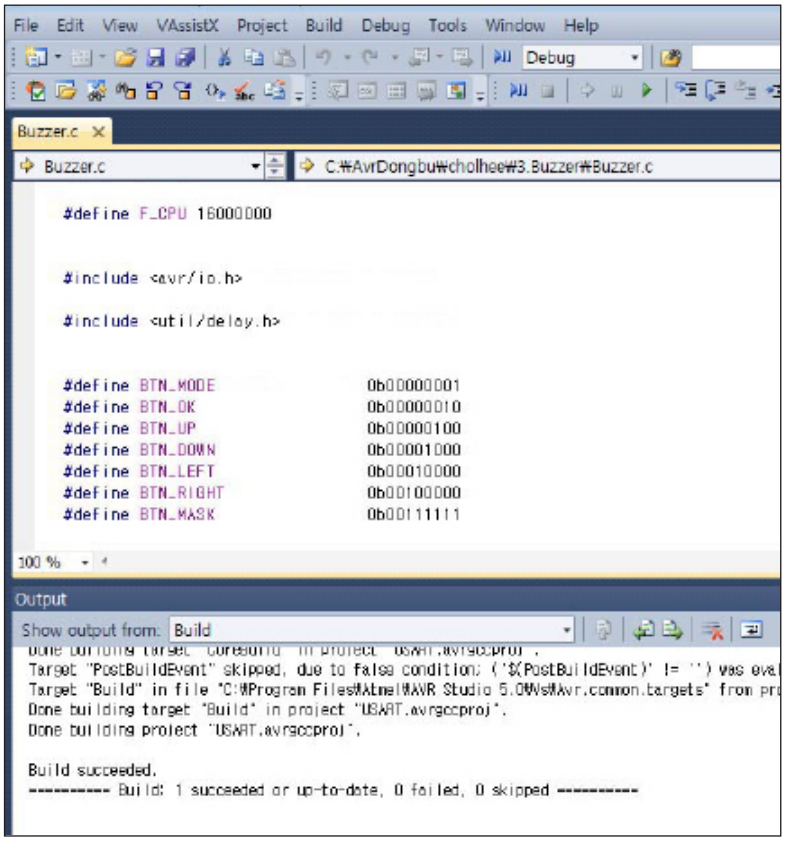
if(ucButton&BTN_OK)という条件文の結果は、ucButton値のOKボタンに対応するビットが1の場合正しく、0なら偽になります。つまり、OKボタンを押した場合、括弧の中身が実行されます。

if文の中のPORTB^=0b10000000;はPORTB=PORTB^0b10000000;の簡単な表現です。PORTBと0b10000000を^(bitwiseXOR)処理するという意味です。XOR(ExclusiveOR)演算は、両方のビットが異なる時に1になります。つまり、ある値1をXORすると、0のときは0XOR1であるため、1になり、1のときは1XOR1なので0になります。したがって、PORTB^=0b10000000はPORTBの7ビット目が0である場合に、1の場合は0に変えることを意味しています。

PB7の出力値を変更した後_delay_ms関数を使用して、約1msの間の動作を停止します。while文を繰り返すことで、OKボタンが押された時には約1msごとにPB7の出力値が変化します。したがって、ブザーは約2msの周期で電流が流れることになり、該当する周波数のビープ音を出します。

実行

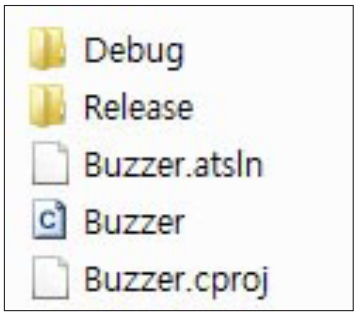
ダウンロードして実行するとServo LEDからPower LEDまでLEDが一つずつ追加で点灯します。LED7個も点灯した後、再度Servo LEDのみ点灯し、順番に一つずつ点灯することを繰り返します。

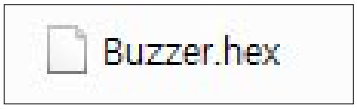


01 Build

Buildメニューを選択しBuild Solutionをクリックしてビルドします。

1 succeeded0 failedが出るとエラーなしでコンパイルが完了したということです。





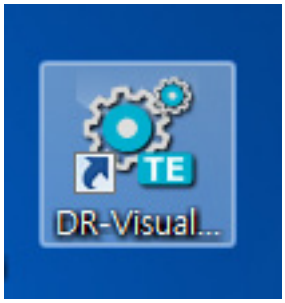
02 Debug フォルダ

Build後にソースがあるフォルダに入るとDebugフォルダが作成されています。クリックしてフォルダの中身を表示します。

03 hex 確認

Debugフォルダの中のhexファイルを確認します。Hexファイルは制御機に移植するファイルです。

DR–Visual Logicで、ファームウェアのリカバリーを行います。



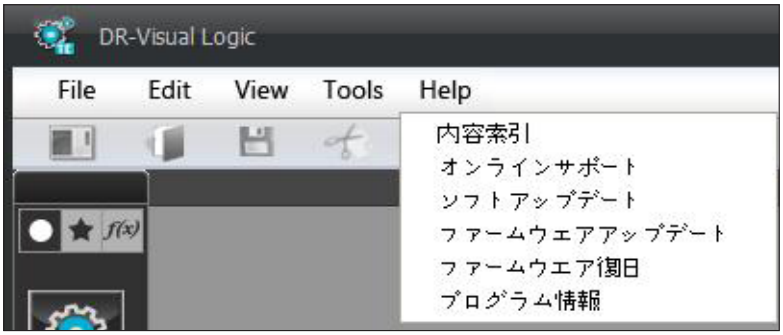
04 Visual Logic

DR–Visual Logicを実行します。



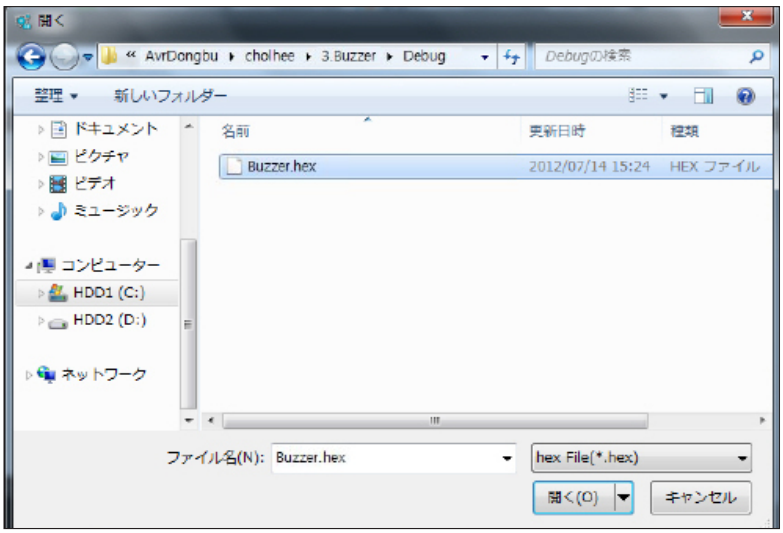
05 接続する

COM番号を設定して接続するをクリックします。



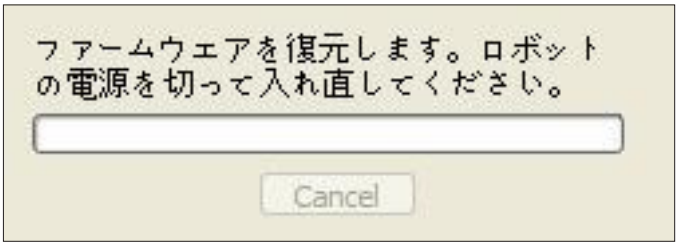
06 ファームウェアの修復

[ヘルプ]メニューからファームウェア修復をクリックします。



07 ロード

Debug フォルダにHexファイルをロードします。



08 ファームウェアの修復

“ロボットの電源を切って入れ直してください”というメッセージが出ます。コントローラの電源ボタンを一度切ってから再度入れ直してください。



09 修復の開始

ファームウェアの修復が開始されます。



10 実行

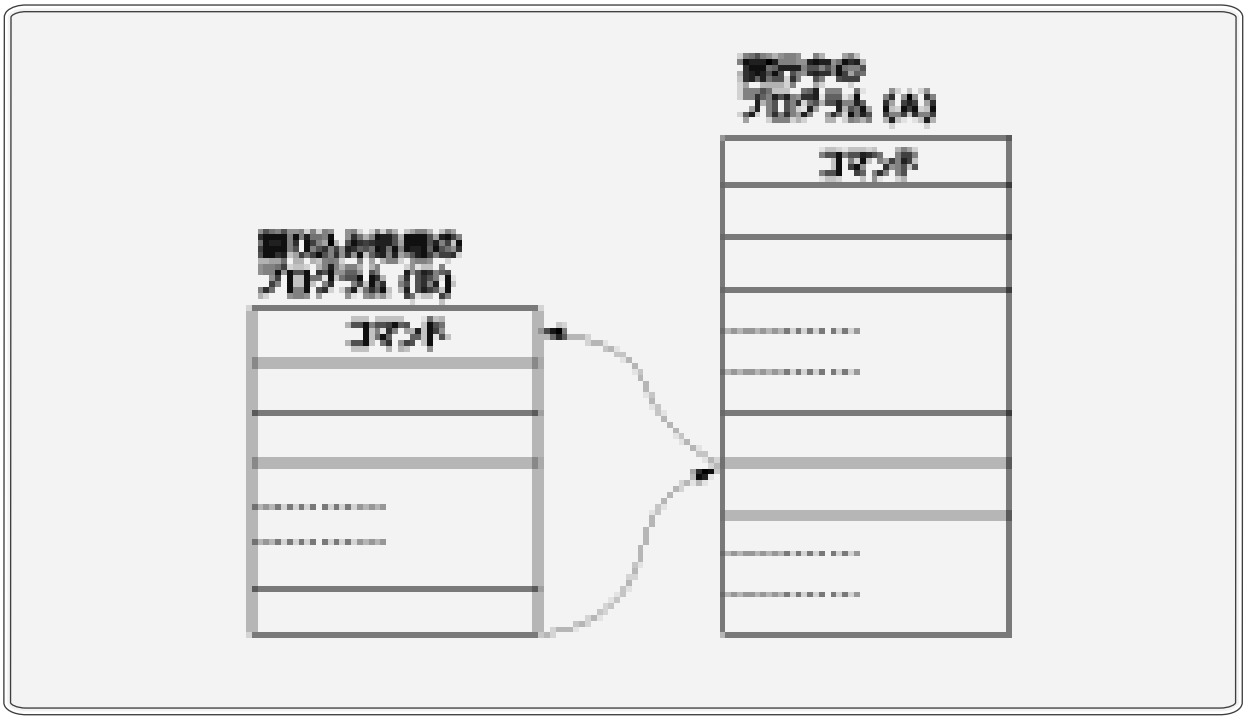
OKボタンを押すと、ブザーでビープ音が鳴ります。

外部インタラプトから入力を受け取る

ATmega128の外部インタラプトを使用してDRCに装着されたMICから入力を受け取ります。

AVR 原理

今回はATmega128の外部インタラプト機能を使用しています。インタラプトは実行中のプログラムを一時停止し、他のプログラムをはめ込み実行させることです。インタラプト要因となる条件があるときに実行中のプログラムを中断して強制的に特定のアドレスに移動し、用意されているインタラプト処理プログラムを実行させ、その処理が完了したら、元のプログラムに戻って実行します（図00）。このときインタラプトを処理する部分をインタラプトサービスルーチン(InterruptService Routine、ISR)といいます。インタラプトを使用すると、プログラムの処理をより効率的に行うことができ、インタラプト要因が発生する正確なタイミングで、その要因に必要な動作をすることができるという利点があります。



ATmega128で使用できるインタラプトソースは、以下のように非常に多いです。上に書かれたものほど優先順位が高いインタラプトです。このチャプターでは、外部インタラプトを使用してプログラミングし、残りのチャプターでは、タイマー/カウンタオーバーフローとUSART受信完了インタラプトなどを使用します。

ベクトルの番号	ベクトルのアドレス	インタラプトのソース	インタラプトの説明
1	$0000	RESET	リセット
2	$0002	INT0	外部インタラプト 0
3	$0004	INT1	外部インタラプト 1
4	$0006	INT2	外部インタラプト 2
5	$0008	INT3	外部インタラプト 3
6	$000A	INT4	外部インタラプト 4
7	$000C	INT5	外部インタラプト 5
8	$000E	INT6	外部インタラプト 6
9	$0010	INT7	外部インタラプト 7
10	$0012	TIMER2 COMP	タイマー/カウンタ2 比較マッチ
11	$0014	TIMER2 OVF	タイマー/カウンタ2 オーバーフロー
12	$0016	TIMER1 CAPT	タイマー/カウンタ1 入力のキャプチャ
13	$0018	TIMER1 COMPA	タイマー/カウンタ1 比較マッチ A
14	$001A	TIMER1 COMPB	タイマー/カウンタ1 比較マッチ B
15	$001C	TIMER1 OVF	タイマー/カウンタ1 オーバーフロー
16	$001E	TIMER0 COMP	タイマー/カウンタ0 比較マッチ
17	$0020	TIMER0 OVF	タイマー/カウンタ0 オーバーフロー
18	$0022	SPI, STC	SPI シリアル通信の完了
19	$0024	USART0, RX	USART0受信完了
20	$0026	USART0, UDRE	USART0 データレジスタ エンプティー
21	$0028	USART0, TX	USART0 送信完了
22	$002A	ADC	ADC 変換が完了
23	$002C	EE READY	EEPROMの準備完了
24	$002E	ANALOG COMP	アナログ比較器
25	$0030	TIMER1 COMPC	タイマー/カウンタ1 比較マッチ C
26	$0032	TIMER3 CAPT	タイマー/カウンタ3 入力のキャプチャ
27	$0034	TIMER3 COMPA	タイマー/カウンタ3 比較マッチ A
28	$0036	TIMER3 COMPB	タイマー/カウンタ3 比較マッチ B
29	$0038	TIMER3 COMPC	タイマー/カウンタ3 比較マッチ C
30	$003A	TIMER3 OVF	タイマー/カウンタ3 オーバーフロー
31	$003C	USART1 RX	USART1 受信完了
32	$003E	USART1 UDRE	USART1 データレジスタ エンプティー
33	$0040	USART1 TX	USART1 送信完了
34	$0042	TWI	TWI(I2C) 直列インターフェース
35	$0044	SPM READY	プログラムメモリ保存の準備完了

ATmega128の外部インタラプトは、ポートEが使用されます。ポートEの8つのピンPEnがそれぞれインタラプトソースINTnに対応します。外部インタラプトの設定に関連するレジスタは以下のとおりです。

■ SREG : AVR ステータスレジスタ

ビット	7	6	5	4	3	2	1	0
	I	T	H	S	V	N	Z	C
読み取り/書き込み	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初期値	0	0	0	0	0	0	0	0

レジスタは、AVRの全体的な状態を示すレジスタです。二重インタラプトに関連する部分は、7番目のビットです。

- ビット7番I: 全域インタラプト有効

ATmega128からのインタラプトを有効にするには、Iビットを1に設定する必要があります。それぞれのインタラプトが有効かどうかは、該当するレジスタのビットを設定して決定しますが、インタラプトを有効にさせてもIビットが0になっている場合は、実際にはインタラプトが 発生しません。

■ EICRA : 外部インタラプトステータスレジスタA

ビット	7	6	5	4	3	2	1	0
	ISC31	ISC30	ISC21	ISC20	ISC11	ISC10	ISC01	ISC00
読み取り/書き込み	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初期値	0	0	0	0	0	0	0	0

EICRA　レジスタは、外部インタラプト0～3の動作モードを選択するレジスタです。インタラプトソース1つで2つのビットが割り当てられています。ビットの設定に従った動作モードは下表のとおりです。

ISCn1	ISCn0	説明
0	0	INTnにLOW入力が入ってきたときにインタラプトが発生
0	1	使用しない
1	0	INTnに下降エッジ(HIGHからLOWに変わる時)が発生した場合、非同期インタラプト要求
1	1	INTnに上昇エッジ(LOWからHIGHに変わる時)が発生した場合、非同期インタラプト要求

■ EICRB : 外部インタラプトステータスレジスタ B

ビット	7	6	5	4	3	2	1	0
	ISC71	ISC70	ISC61	ISC60	ISC51	ISC50	ISC41	ISC40
読み取り/書き込み	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初期値	0	0	0	0	0	0	0	0

EICRA レジスタは、外部インタラプト0～3の動作モードを選択するレジスタです。インタラプトソース1つで2つのビットが割り当てられています。ビットの設定に従った動作モードは下表のとおりです。

ISCn1	ISCn0	説明
0	0	INTnに LOW 入力が入ってきたときにインタラプトが発生
0	1	INTnに下降エッジ と昇エッジが発生した場合インタラプタ 要求
1	0	INTnに下降エッジ(HIGHからLOWに変わる時)が発生した場合インタラプタ要求
1	1	INTnに上昇エッジ(LOWからHIGHに変わる時)が発生した場合インタラプタ要求

■ EIMSK : 外部インタラプトマスクレジスタ

ビット	7	6	5	4	3	2	1	0
	INT7	INT6	INT5	INT4	INT3	INT2	INT1	INT0
読み取り/書き込み	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初期値	0	0	0	0	0	0	0	0

EIMSKレジスタは、外部インタラプトを使用するかどうかを決めるレジスタです。ビットを1に設定すると対応する外部インタラプトを許可するものであり、0に設定すると許可されません。INTnを1に設定してSREGのIビットが1に設定されている場合、ISCn1 : 0ビットで設定した条件を満たす外部インタラプトが発生したときにEIFRのINTFnビットが1になり、インタラプトサービスルーチンを実行します。

EIMSK：外部インタラプトマスクレジスタ

ビット	7	6	5	4	3	2	1	0
	INTF7	INTF6	INTF5	INTF4	INTF3	INTF2	INTF1	INTF0
読み取り/書き込み	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初期値	0	0	0	0	0	0	0	0

EIFRレジスタは、外部インタラプトの個々のインタラプト要求の状態を示すレジスタです。PEnピンの状態変化がインタラプト要求を起こしたときは、該当するビットINTFnが1に設定され、このビットは、インターラプトサービスルーチンに入るときに、ハードウェアによって0に初期化されます。ソフトウェア的に手動でこのビットを0に変えには、このビットに1を書く必要があります。SREGのIビットとINTn、INTFnがすべて1の場合は、外部インタラプトnが実行します。

H/W 原理

DRCには、2つのマイクが左右に装備されており、回路を介して外部インタラプトピンに接続されます。一定サイズ以上の振動が検出されるとインタラプトピンに入る信号がLOWからHIGHに変化し、振動がなくなれば再びLOWに戻ります。拍手の音では一度にLOW-〉HIGH-〉LOWと外部インタラプトピンの入力が変化するため、拍手の音が鳴っている時、外部インタラプト入力がLOWからHIGHに変わる上昇エッジが発生します。

DRC～2つの外部インタラプトを使用して、左のマイクはINT5、右のマイクは、INT6入力を受け取ります。左から音が出た場合近くの左側のマイクでより速く上昇エッジが起こるので、INT5でINT6より先にインタラプトが発生します。逆に右から音がすればINT6でINT5より先にインタラプトが起こります。

今回の例では、INT5、INT6で上昇エッジが発生した場合インタラプトが発生するように設定してインタラプトサービスルーチンを作成してみます。

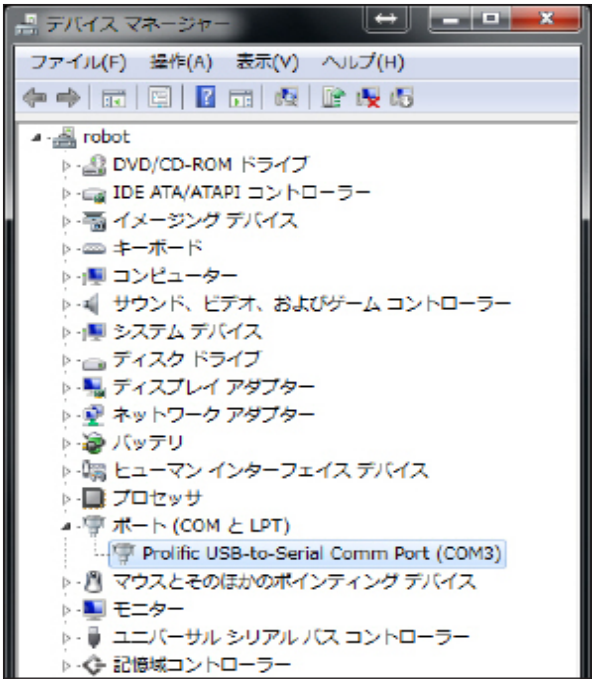
準備



ハードウェアの準備

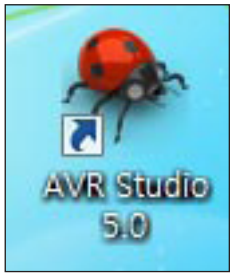
01 接続

DRC制御機アダプターで電源を接続し、USB to Serial CableでPCとDRCを接続します。



02 COM port 接続

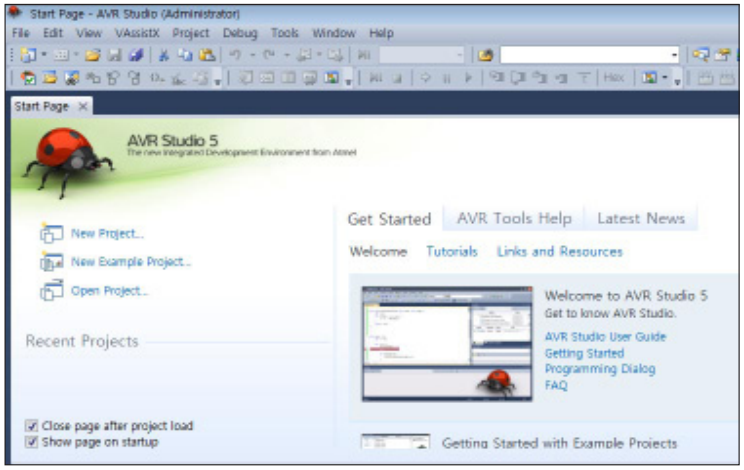
マイコンピュータ>プロパティ>デバイスマネージャでCOM接続を確認します。



AVR Studioの実行とコーディングの準備

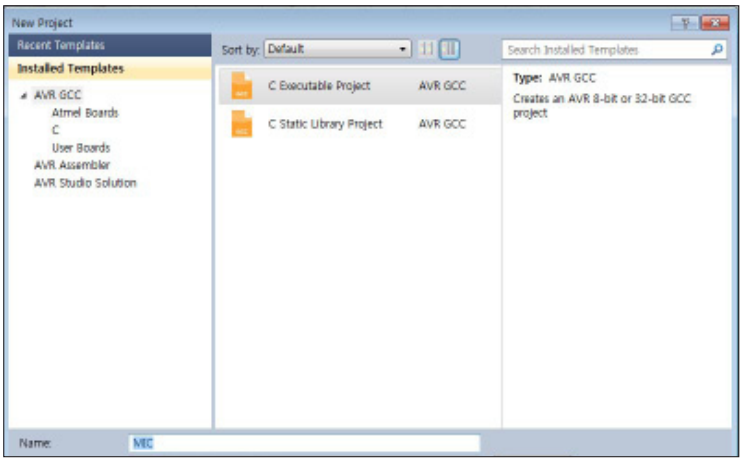
01 実行

AVR Stdio を実行します。



02 New Project

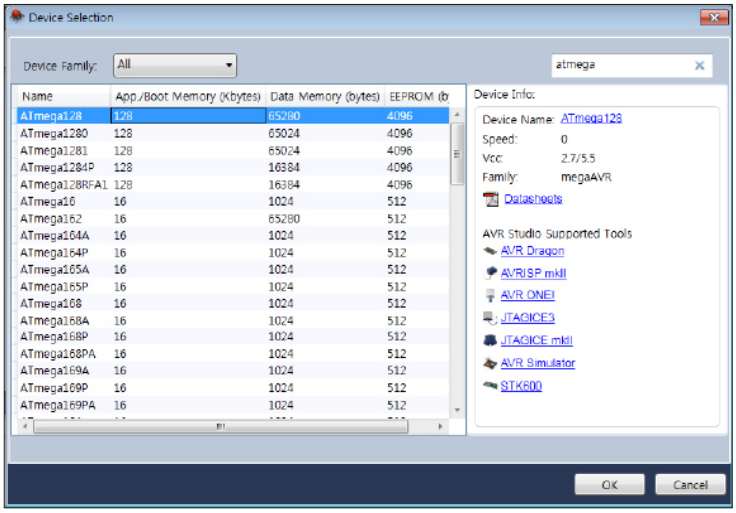
New Project をクリックします。



03 GCC 選択

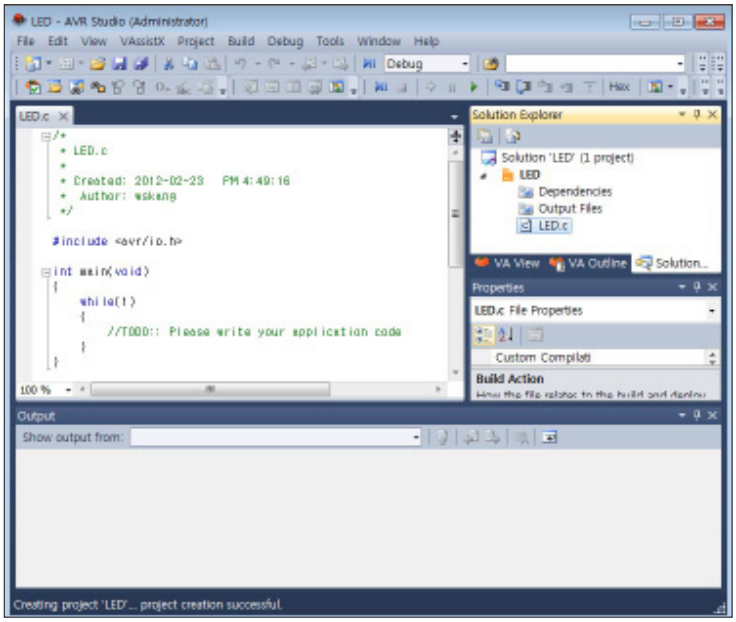
C Executable Projectを選択します。

Nameを”MIC”に入力して、Create directory for solutionをクリックし、OKボタンを押します。



04 ATmega 選択

右上からatmegaを検索し、リスト中のATmega128を選択します。



05 Editor

編集ウィンドウでのコーディングを開始します。

次の例を見てください。

例

DRCの左側のマイクで音を認識すると、Servo LEDを0.5秒間入れて切り、右のマイクで音を認識すると、Power LEDを0.5秒間入れて切る例です。

```
//CPU クロックの設定
#define F_CPU 16000000

//ATmega128のレジスタなどが定義されている
#include <avr/io.h>
//interruptを使用するために必要な情報が定義されている
#include <avr/interrupt.h>
//_delay_ms などディレイ が定義されている
#include <util/delay.h>

//LEDの名前に するポートCのピン定義
#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

//外部インタラプト5番のISR。 Servo LEDを0.5秒間入れて切る。
ISR(INT5_vect){
        PORTC &= ~LED_SERVO;
        _delay_ms(500);
        PORTC |= LED_SERVO;
}

//外部インタラプト6番の ISR. Power LEDを0.5秒間入れて切る。
ISR(INT6_vect){
        PORTC &= ~LED_POWER;
        _delay_ms(500);
        PORTC |= LED_POWER;
}
```

```
int main(void)
{
        //ポートC の入出力方向を設定する
        DDRC = 0b01111111;
        //ポートCの出力値を設定する
        PORTC = 0b01111111;

        //全 インタラプトを無 にする
        cli();

        //INT5, INT6を上昇エッジモード に設定する
        EICRB |= 0b00111100;
        //INT5, INT6を使用するように設定
        EIMSK |= 0b01100000;
        //INT5, INT6のインタラプトフラグを0に初期化
        EIFR |= 0b01100000;

        //全 インタラプトを有 にする
        sei();

        //while の 容を無限に繰り返す
        while(1)
        {

        }

        // の形に合わせて整 値を返す
        return 1;
}
```

例の説明

```
#define F_CPU 16000000
```

#define文はコードの前処理命令の一つで、今後F_CPUという値が表示されたら16000000に置き換えられることを意味しています。DRCのATmega128は16MHzのクロック速度で動作します。クロック速度に比例して命令の処理速度が異なりますので、この命令を通じて、このプログラムが動作する速度を明示するとディレイ関数が正常に動作します。

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

#define BTN_MODE 0b00000001は、今後BTN_MODEが出れば0b00000001に置き換えてコンパイルするという意味です。DRCの各ボタンに対応するビットをボタンの名前に置き換えてプログラムを作成しやすくするために宣言します。

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

avr/io.hヘッダファイルにはDDRX、PORTXレジスタなどAVR制御のために必要なレジスタが定義されており、avr/interrupt.hヘッダファイルには、インタラプトを使用するための関数が定義されています。 util/ delay.hヘッダファイルには、_delay_ms( )などのたびたび使用するディレイ関数が定義されているので # include文を使用します。

```
#define LED_SERVO                    0b00000001
…
#define LED_MASK                     0b01111111
```

#define LED_SERVO 0b00000001は、今後LED_SERVOが出れば0b00000001に置き換えてコンパイルするという意味です。DRCの各LEDに対応するビットをLEDの名前に置き換えてプログラムを作成しやすくするために宣言します。

```
ISR(INT5_vect){
        PORTC &= ~LED_SERVO;
        _delay_ms(500);
        PORTC |= LED_SERVO;
}
ISR(INT6_vect){
        PORTC &= ~LED_POWER;
        _delay_ms(500);
        PORTC |= LED_POWER;
```

外部インタラプト5番と6番のインタラプトサービスルーチンです。インタラプトが発生してEIFRレジスタの対応するビットが1になると、インタラプト要求に応じて実行していたプログラムを停止し、インタラプトサービスルーチンに入ります。進入する瞬間EIFRレジスタのビットは再び0に初期化されます。左のマイクの音が認識されてINT5に上昇エッジが発生すると、INT5のインタラプトサービスルーチンからPORTCのServo LEDに対応するビットを0にしてServo LEDを入れて、500ms待ってから、再度、そのビットを1にすることでServo LEDをオフにします。 同じように右のマイクに音が入ってINT6に上昇エッジが発生すると、INT6のインタラプトサービスルーチンでPower LEDを500msの間入れて切ります。

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

ポートCの0～6番ピンを出力に設定し、出力値を1にします。これにより、すべてのLEDが消灯します。

```
cli();

EICRB |= 0b00111100;
EIMSK |= 0b01100000;
EIFR |= 0b01100000;

sei();
```

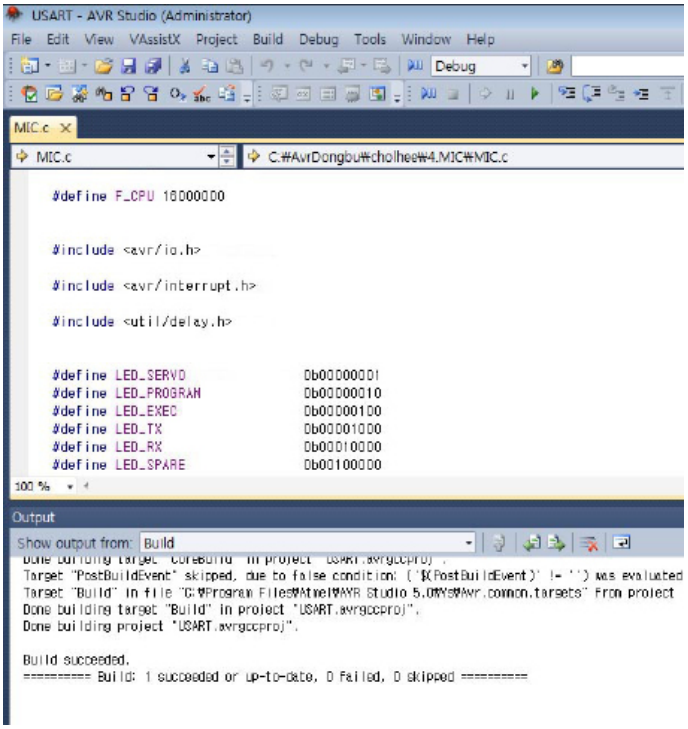
cli()とsei()は、それぞれ全体のインタラプトを無効にし、活性化させる命令です。ATmega128には、SREGというAVRのステータスレジスタがあり、このうち7番目のビットのIビットはインタラプト要求があったときインタラプトサービスルーチンにジャンプできるようにするビットです。レジスタの値を変えて、外部インタラプトを使用する設定にするには、このIビットを1に設定する必要があります。cli()命令は、Iビットを0に、sei()命令は、Iビットを1に設定します。
インタラプトの設定を変更中にインタラプトが発生して予想できない誤動作が発生することを防止するために、まずcli()命令を通じて全体のインタラプトを無効にします。その後、外部インタラプトに関連するレジスタに適切な値を入れて使用したい用途に合わせて、外部インタラプトを設定します。まず、外部インタラプト5、6番の上昇エッジモードで使用するため、EICRBのISC61、ISC60、ISC51、ISC50はすべて1にする必要があります。そのためEICRBを0b00111100と|(bitwise OR)演算して、そのビットを1にしています。また、EIMSKの5、6番目のビットを1にすることで、外部インタラプトを使用することを通知します。そのためEIMSKを0b01100000と|演算して5、6番目のビットを1にします。最後に、EIFRの5、6番目のビットを0に初期化します。前に説明したように、EIFRのビットを強制的に0にするためには、そのビットに1を使用します。 EIFRの5、6番目のビットに1を使って初期化します。
外部インタラプトの設定が終わった後にsei()命令を通じて全体のインタラプトを有効にしています。

```
while(1)
{

}
```

このプログラムは、while文の中には何も入っていません。このため、インタラプトが発生しない状況では、プログラムはwhile文を繰り返すだけ何もしません。しかし、インタラプトが発生した場合、プログラムはwhile文を回すことをしばらく止めてインタラプトサービスルーチンにジャンプして、ルーチンが終了すると元の場所に戻って続けてwhile文を繰り返します。

실행

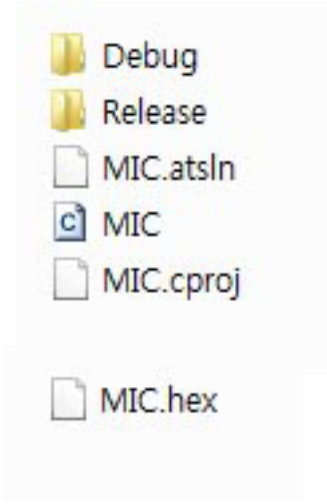
ダウンロードして実行し，音が出たら Servo LEDと Power LEDが 0.5秒間ONになった後、OFFになります。左から拍手を打てばServo LEDが最初に点滅してPower LEDが後に点滅します。逆に右から拍手を打つとPower LEDが点灯した後Servo LEDが点灯します。これは、音が発生した方向に沿って両側のマイクに音が到達する時間が異なっていて、そのインタラプトが発生するタイミングが微妙に異なっているからです。INT5が最初に発生してINT5のインタラプトサービスルーチンに入った後、INT6が発生した場合、EIFRの6番目のビットは、INT5のインタラプトサービスルーチンが終了するまで1になったまま維持され、INT6のインタラプトサービスルーチンに入り、ようやく0に初期化されます。



01 Build

Buildメニューを選択しBuild SolutionをクリックしてBuildします。

1 succeeded0 failedが出るとエラーなしでコンパイルが完了したということです。



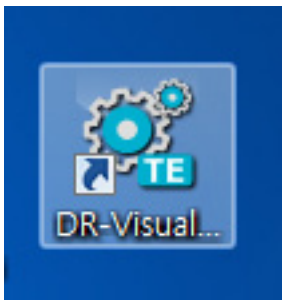
02 Debug フォルダ

Build後にソースがあるフォルダに入るとDebugフォルダが作成されています。クリックしてフォルダの中身を表示します。

03 hex 確認

Debugフォルダの中のhexファイルを確認します。Hexファイルは制御機に移植するファイルです。

DR-Visual Logicで、ファームウェアのリカバリーを行います。



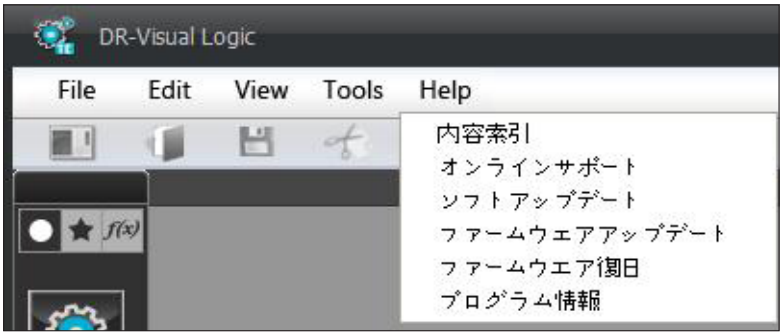
04 Visual Logic

DR-Visual Logicを実行します。



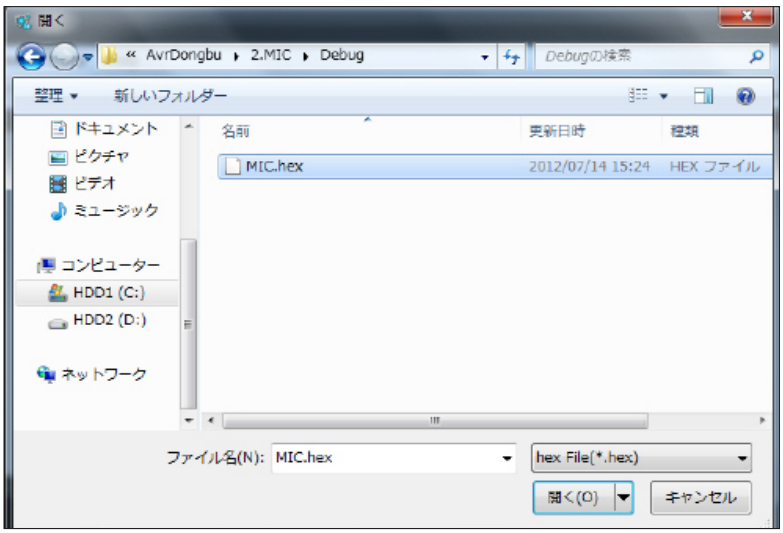
05 接続する

COM 番号を設定して接続するをクリックします。



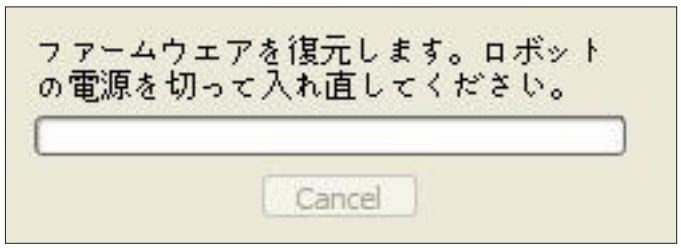
06 ファームウェアの修復

[ヘルプ]メニューからファームウェア修復をクリックします。



07 ロード

Debug フォルダにHexファイルをロードします。



08 ファームウェアの修復

“ロボットの電源を切って入れ直してください”というメッセージが出ます。コントローラの電源ボタンを一度切ってから再度入れ直してください。



09 修復の開始

ファームウェアの修復が開始されます。

10 実行

左から拍手を打てば Servo LED が最初に点滅して Power LEDが後に点滅します。逆に右から拍手を打つと Power LEDが点灯した後 Servo LEDが点灯します。

PART 03

AVRロボット制御機Firmwareの習得 HOVIS

タイマー/カウンタにディレイ機能を使用する

ATmega128のタイマー/カウンタを使用してディレイを実装し、Chaper1のLEDの例に使用します。

AVR 原理

ATmega128には、8ビットタイマー/カウンタはタイマー/カウンタ0とタイマー/カウンタ2があり、16ビット·タイマー/カウンタはタイマー/カウンタ1とタイマー/カウンタ3があり、計4つのタイマー/カウンタがあります。タイマー/カウンタは、特定の信号間の時間を計ったり、信号が出てきた数を数えたり、PWM波形を作成するなど、様々な用途に使用されます。タイマー/カウンタの様々な機能を整理すると以下のとおりです。

- 一般的なタイマー動作

タイマー/カウンタに印加されるクロックを受けて自分で一定時間ごとにカウントレジスタの値を増加させ、オーバーフローが発生すると、オーバーフローインタラプトフラグを1にしカウントレジスタを0に初期化します。オーバーフローが発生するたびにカウントレジスタを0以外の一定の特定の値で初期化すれば、必要な期間続けてオーバーフローインタラプトを発生させます。

- PWM 波形発生動作

PWM波形を発生するモードでは、一定の周期でオーバーフローが発生します。ここで初期値の0と最大値の間に比較値を設定して、0から比較値までの時間と比較する値からオーバーフローまでの時間をHIGH、LOWやLOW、HIGHで出力するPWM波形を作成できます。比較値の値を変更することにより、デューティ·サイクルを調整し、PWM制御ができます。

※ PWMとは?

PWMは Pulse Width Modulationの略で、パルス·ワイズ·モジュレーションとも呼ばれています。モーター制御やLEDの制御をアナログ電圧で制御するときには1V、2V、3Vなど好きな電圧を印加してモータの速度やLEDの明るさを制御ができます。ところが、デジタル信号で制御するときは、HIGHとLOWしか持てる値がないので可能な強さが制限されます。このような場合に使用するのがPWMです。一定の高速周期でデジタルパルスを発生させますが、パルスがHIGHに維持される幅を調整することで、まるでアナログ電圧をかけるのと同じ効果を出します。例えばHIGHが5VでLOWが0Vであるデジタル出力ピンからHIGHとLOWの長さが1:1のパルス信号を一定の速い周期で出すならば、まるで2.5Vの電圧が加えられたのと同じ効果が出ます。他の例として、長さが4:1の場合4Vの電圧がかかったのと同じ効果が出ます。ここでHIGHの長さと全体周期の比をデューティ·サイクル(Duty Cycle)と呼びます。 HIGH : LOWの時間の割合が1:1の場合、デューティ·サイクルは50%、4:1の場合80%になります。

- トグル動作

カウントレジスタの値と比較値が一致すると、出力ピンの状態を切り替える(HIGHであればLOWに、LOWの場合はHIGHにする)ように設定できます。比較値を変更する方法で出力ピンの出るデジタル波形の周波数を調整できます。

- パルス幅測定動作

入力キャプチャ機能を使用して、特定のピンの入力信号の変化を検出し、その変化が起きた時間をキャプチャできます。結果を利用してきた信号の幅を比較したり、周期を測定するために利用できます。

- カウンタ動作

外部からの信号の変化のクロック入力を受けて、信号の変化があるたびに、カウントレジスタが増加するようにして、特定の信号の変化が何度かあったのか数えられます。

今回の例では、タイマー/カウンタ0を使用して、一般的なタイマー動作を設定して、それに基づいChapter1のLEDの例で使用した_delay_msをタイマー/カウンタ0の機能を応用して交換します。

タイマー/カウンタ0に関連する代表的なレジスタは、下記のとおりです。

TCCR0 : タイマー/カウンタ0 制御レジスタ

ビット	7	6	5	4	3	2	1	0
	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00
読み取り/書き込み	W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初期値	0	0	0	0	0	0	0	0

- ビット2:0番 CS02:0 : クロック選択

タイマー/カウンタ0が動作するクロックを選択します。 ATmega128に入ってくる基本的なクロック周波数をどんな割合で分けてTCNT0で使用するかを決めます。DRCの基本クロック周波数は16Mhzです。したがって、CS02 : 00b101で設定すると、タイマー/カウンタ0の動作周波数は16Mhz/128=125khzになります。8usごとにTCNT0が1ずつ増加します。

CS02	CS01	CS00	説明
0	0	0	動作 しない
0	0	1	クロック/1
0	1	0	クロック/8
0	1	1	クロック/32
1	0	0	クロック/64
1	0	1	クロック/128
1	1	0	クロック/256
1	1	1	クロック/1024

- ビット6番、3番WGM01：0：波形生成モード

タイマー/カウンタ0が動作するモードを決定します。各モードに応じてカウントレジスタが増加する方向、カウンタの最大値、発生する出力波形の種類が異なります。タイマー/カウンタ0は、4つの動作モードがあり、通常モード、CTCモード、2種類のPWMモードがあります。それぞれのモードの詳細については、以下のレジスタの説明を参照してください。

ISCn1	ISCn0	動作モード	最大値	OCR0 レジスタの更新する時	TOV0 フラグが1となる時
0	0	一般的	255(0xFF)	設定するとすぐ	TCNT0=255
0	1	Phase Correct PWM	255(0xFF)	TCNT0=最大値	TCNT0=0
1	0	CTC	OCR0	設定するとすぐ	TCNT0=255
1	1	高速 PWM	255(0xFF)	TCNT0=0	TCNT0=255

- ビット5:4回COM01：0：比較マッチ出力モード

この2つのビットはOC0端子の動作を設定します。OC0は、ポートBの4番ピン(PB4)にあります。PB4はCOM01：0中　一つでも1である場合には、通常の入出力ポートの機能ではなく、タイマー/カウンタ0の比較出力ピンの機能に使用されます。動作モードによってCOM01：0の設定値に基づいてOC0ピンの機能が異なります。表に示した比較マッチとは、これから説明するOCR0（出力比較レジスタ）とTCNT0（タイマ/カウンタ0レジスタ）がお互いに同じ値の時を指しています。　COM01:0をOC0ピンとして使用することを設定しても、実際にDDRBの4番目のビットを出力ピンに設定するとOC0出力されます。

■ PWM モードでない場合（一般的に、CTC）の比較出力モード

COM01	COM00	OC0 ピンの機能
0	0	一般I/ Oポートで動作
0	1	比較マッチでOC0出力をトグル
1	0	比較マッチでOC0出力をLOWに設定
1	1	比較マッチでOC0出力をHIGHに設定

■ 高速PWMモードの比較出力モード

COM01	COM00	OC0 ピンの機能
0	0	一般I/ Oポートで動作
0	1	使用しない
1	0	比較マッチでOC0出力をLOWに設定して TCNT0=0のときOC0出力をHIGHにする
1	1	比較マッチでOC0出力をHIGHに設定して TCNT0=0のときOC0出力をLOWにする

■ Phase Correct PWM モードの比較出力モード

COM01	COM00	OC0 ピンの機能
0	0	一般I/ Oポートで動作
0	1	使用しない
1	0	アップカウントの比較マッチでOC0出力をLOWに設定して ダウンカウントの比較マッチでOC0出力をHIGHにする
1	1	アップカウントの比較マッチでOC0出力をHIGHに設定して ダウンカウントの比較マッチでOC0出力をLOWにする

- ビット7番FOC0 : 強制出力の比較

FOC0はPWMモードではない場合にのみ動作します。このビットに1を書き込むと、タイマーの値に関係なく、強制的に比較マッチを起こし、COM01 : 0の設定に応じてOC0ピン出力値を変更します。このビットが強制的に起こした比較マッチはインタラプトを発生させず、CTCモードでタイマーを初期化しません。また、このビットは、書き込みだけが可能であり、常に0に読み込まれます。

■ TCNT0 : タイマー/カウンタ0レジスタ

ビット	7	6	5	4	3	2	1	0
	TCNT0[7:0]							
読み取り/書き込み	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初期値	0	0	0	0	0	0	0	0

TCNT0はタイマー/カウンタ0の現在のカウント値を格納するレジスタです。 CS02 : 0ビットによって決定されたタイマー/カウンタ0の周期ごとに1ずつ増加します。このレジスタは、直接読み書きすることができますが、TCNT0の値を書いた直後のクロックでの比較マッチが起きないため、必要な比較マッチを逃す
可能性もありますので注意して使用する必要があります。

■ OCR0 : タイマー/カウンタ0の出力比較レジスタ

ビット	7	6	5	4	3	2	1	0
	OCR0[7:0]							
読み取り/書き込み	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初期値	0	0	0	0	0	0	0	0

OCR0 レジスタの中に入っている値はTCNT0の値が変更されるたびTCNT0と比較されます。 OCR0とTCNT0の値が同じになると比較マッチが起こり、動作モードとCOM02 : 0ビットの値に応じてOC0端子の出力が異なります。 OCR0の値を変えても、動作モードによっては、すぐに反映されるわけではなく、TCNT0が特定の値に到達したときに反映される場合があります。詳細については、WGM01 : 0ビットの表を参照してください。

■ TIMSK : タイマー/カウンタ0インタラプトマスクレジスタ

ビット	7	6	5	4	3	2	1	0
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIEB	TOIE1	OCIE0	TOIE0
読み取り/書き込み	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初期値	0	0	0	0	0	0	0	0

- ビット1番OCIE0 : タイマー/カウンタ0の出力を比較マッチインタラプトを有効にする。

このビットを1に設定するとタイマー/カウンタ0の比較マッチインタラプトが有効になります。このビットが1に設定されSREGレジスタのIビットが1に設定された状態で比較マッチが起こるとTIFRレジスタのOCF0ビットが1になり、対応するインタラプトサービスルーチンが実行されます。

- ビット0番TOIE0 : タイマー/カウンタ0オーバーフローインタラプトを有効にする。

このビットを1に設定するとタイマー/カウンタ0のオーバーフローインタラプトが有効になります。このビットが1に設定されSREGレジスタのIビットが1に設定された状態で、オーバーフローが起こるとTIFRレジスタのTOV0ビットが1になり、対応するインタラプトサービスルーチンが実行されます。.

■ TIFR :タイマー/カウンタ0インタラプトマスクレジスタ

ビット	7	6	5	4	3	2	1	0
	OCF2	TOF2	ICF1	OCF1A	OCFB	TOV1	OCIE0	TOIE0
読み取り/書き込み	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初期値	0	0	0	0	0	0	0	0

- -ビット1番OCF0 : タイマー/カウンタ0の出力を比較フラグ

このビットはTCNT0とOCR0間で比較マッチが起こると、1に設定されます。このビットは、対応するインタラプトサービスルーチンに入るときに、ハードウェアによって0に初期化されます。ソフトウェア的に手動でこのビットを0に変えるには、このビットに1を書いておく必要があります。 SREGのIビットとOCIE0、OCF0がすべて1の場合、タイマー/カウンタ0比較マッチインタラプトが実行されます。

-ビット0番TOV0 : タイマー/カウンタ0オーバーフローフラグ

このビットはタイマー/カウンタ0でオーバーフローが起こると、1に設定され、対応するインタラプトサービスルーチンに入るときに、ハードウェアによって0に初期化されます。ソフトウェア的に手動でこのビットを0に変えるには、このビットに1を書いておく必要があります。 SREGのIビットとTOIE0、TOV0がすべて1の場合、タイマー/カウンタ0オーバーフローインタラプトが実行されます。 Phase Correct PWMモードでは、他のモードと異なり、TCNT0の増減方向がダウンカウントからアップカウントに変わるとき(TCNT0=0のとき)TOV0ビットが1に設定されます。

この他にもASSR、SFIORなどレジスタがタイマー/カウンタ0の動作に関連しています。詳細はATmega128のデータシートまたは他の書籍を参照してください。

前のTCCR0の説明でも示したとおり、タイマー/カウンタ0の動作モードは4つあります。

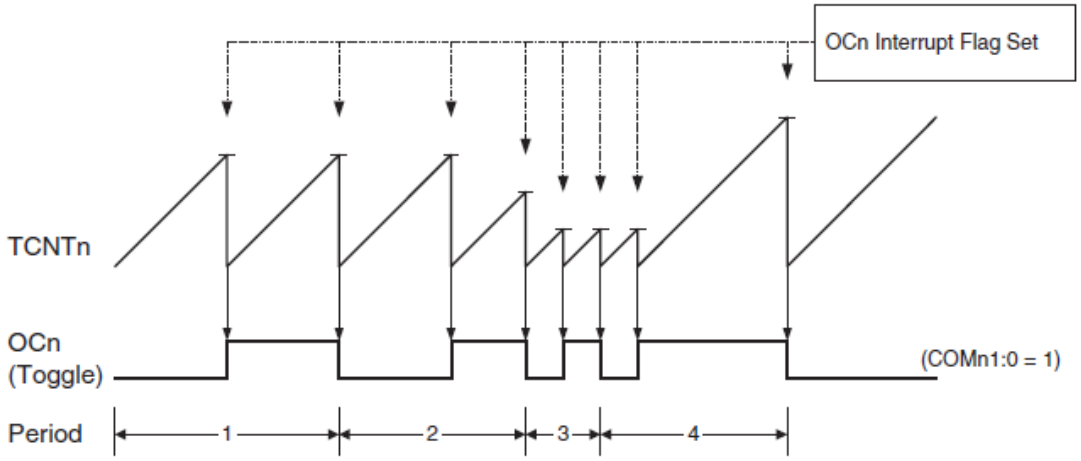
1.一般モード(WGM01：0=0b00)

最も簡単な動作モードでTCNT0がいつもアップカウンタ(増加するカウンタ)として動作し、最大値255で最小値0に変わる瞬間にTOV0が1になります。 SREGのIビットとTOIE0がすべて1であればタイマー/カウンタ0オーバーフローインタラプトサービスルーチンが実行されてTOV0は再び0に初期化されます。一般モードは、通常、タイマー/カウンタ0オーバーフローインタラプトを使用するか、外部クロック信号をカウントするために使用して、比較マッチインタラプトを使用できますが、CPU時間の占有を理由に推奨されません。

2. CTCモード(WGM01：0=0b10)

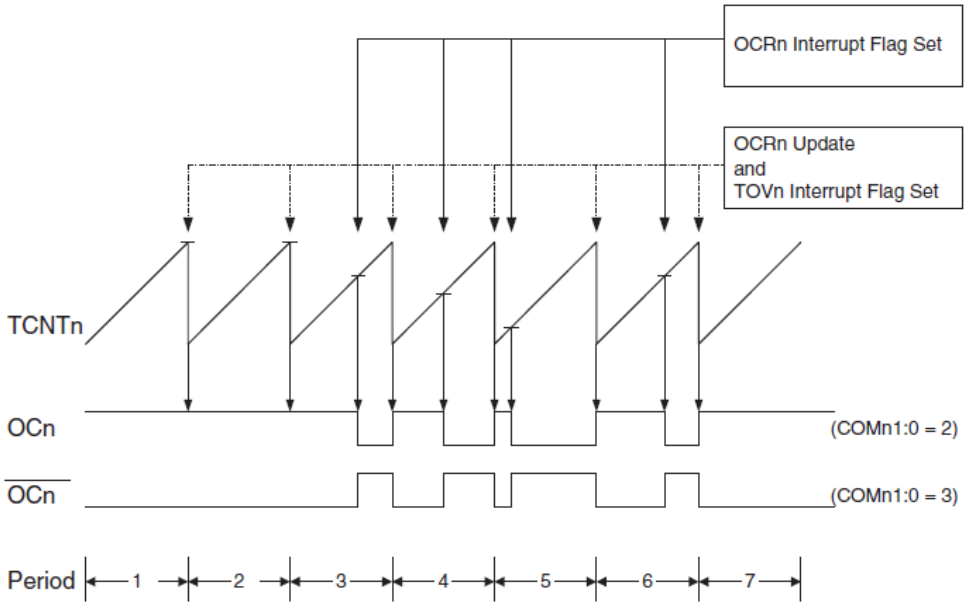
CTCモードは、Clear Timer on Compareモードの略で、OCR0レジスタがタイマーの初期化の周期を決定するために使用されます。 CTCモードでは、TCNT0レジスタがOCR0と同じとなる時、つまり比較マッチが起こる瞬間0に初期化されます。つまり、OCR0レジスタの値がカウンタの最大値を決定します。比較マッチが起こる瞬間OCF0ビットは1に変わり、タイマ/カウンタ0オーバーフローインタラプトサービスルーチンを実行すると、0に初期化されます。

COM01：0の値を0b01に設定すると、比較マッチが起こるたびにOC0ピンをトグルするように設定することができます。これを利用すると、OC0ピンに周期的なパルスを作成することができ、OCR0の値を変える度にパルスの周波数を変更できます.



3. 高速PWMモード(WGM01：0=0b11)

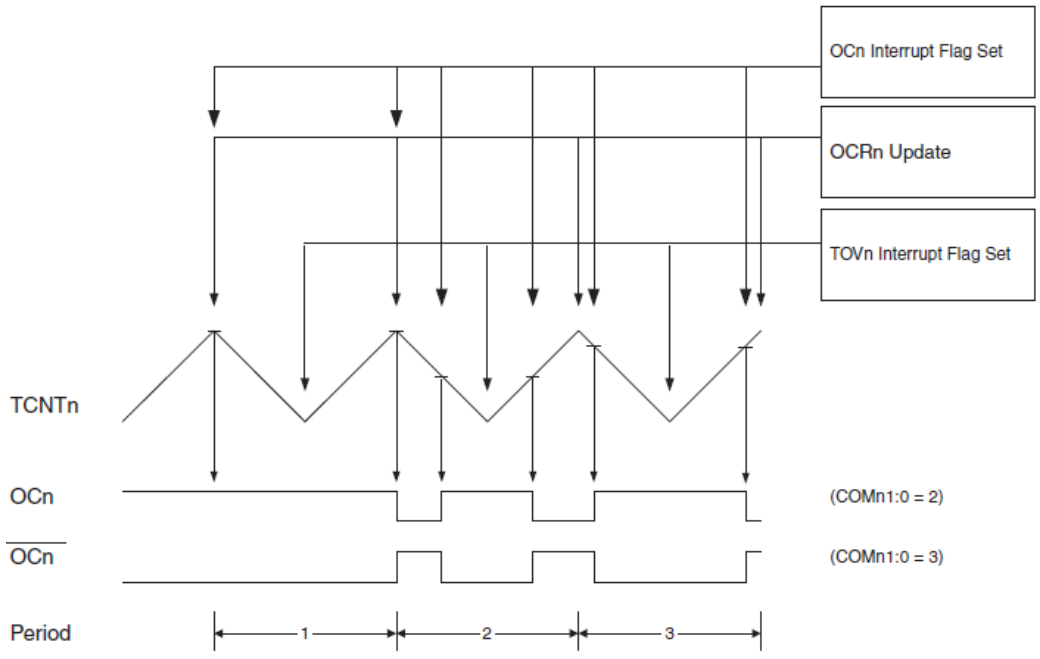
高速PWMモードでは、TCNT0が0から255まで増加して0に変化する鋸歯状に変わり、OCR0の値に応じてデューティ·サイクルが変化するPWM出力を生成するモードです。 COM01：00b10に設定した場合、OC0ピンは比較マッチが起きたときにLOWに変化し、TCNT0が0のときHIGHになります。そのため、OC0は0～OCR0までHIGH、OCR0～255まではLOWである波形が出てくる非反転出力として比較モードで動作します。 COM01：00b11に設定した場合には、逆に、比較マッチが起こった時にHIGH、TCNT0が0のときLOWになります。したがって、0～OCR0までLOW、OCR0～255まではHIGHである波形が出る反転出力比較モードで動作します。



4. Phase Correct PWM モード(WGM01:0=0b01)

Phase Correct PWM モードは、高速PWMモードと似ていますが、TCNT0は常に増加する方向に動作するのではなく0から255まで増加し、再び255から0に減少することを繰り返しているモードです。高速PWMモードのように0から255まで、1周期ではなく、0から255まで、そして255から0まで 1 周期であるため、高速PWMモードに比べて周期は2倍だがTCNT0=0を基準に対称な特性のため、モータ制御に使用されます。

COM01:0を0b10に設定した場合、OC0ピンはアップカウントでの比較マッチが起きたときにLOWに変化し、ダウンカウントでの比較マッチが起きたときにHIGHになります(非反転出力比較モード)。 COM01 : 00b11に設定した場合には、逆に、アップカウントでの比較マッチが起きた時にHIGH、ダウンカウントでの比較マッチが起きたときにLOWになります(反転出力比較モード)。



H/W 原理

今回の例では、ATmega128に内蔵されタイマー/カウンタ0を用いてLEDを制御します。LEDのハードウェアに関連した内容は、Chapter1を参照してください。

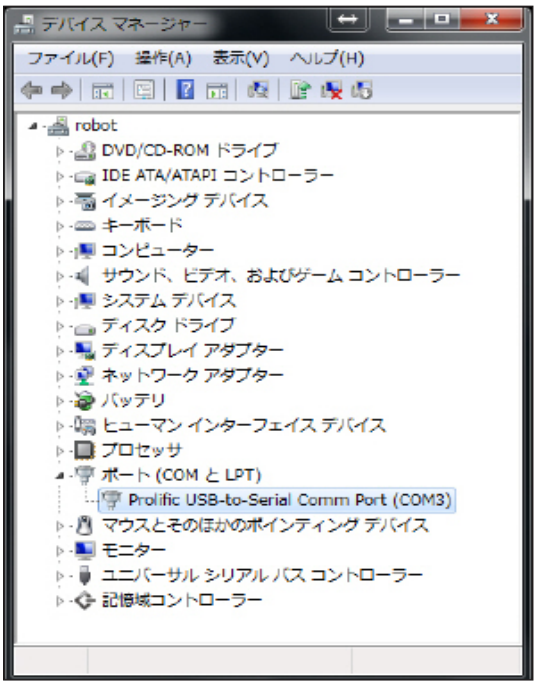
準備



ハードウェアの準備

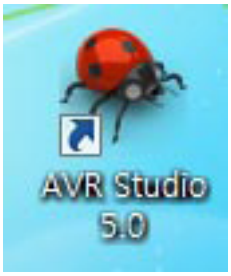
01 接続

DRC制御機アダプターで電源を接続し、USB to Serial CableでPCとDRCを接続します。



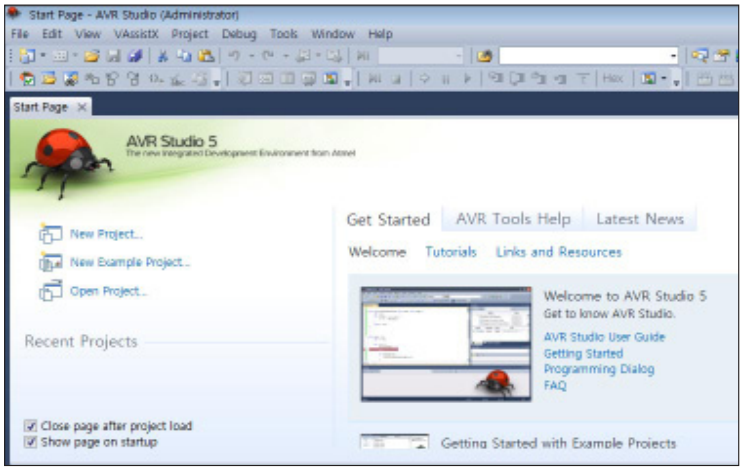
02 COM port 接続

マイコンピュータ>プロパティ>デバイスマネージャでCOM接続を確認します。



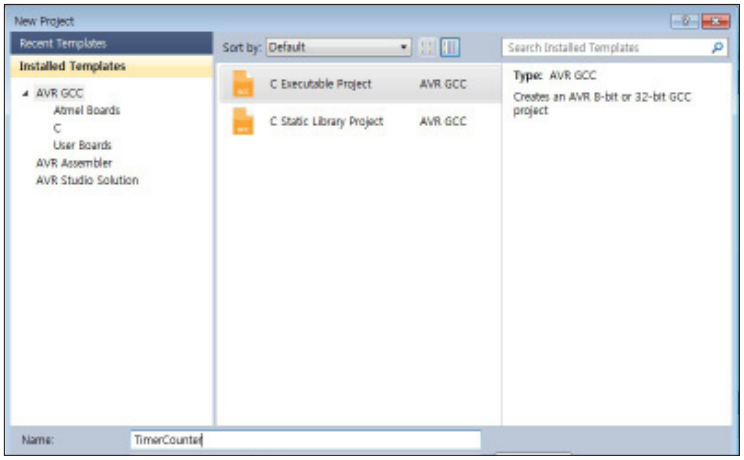
01 実行

AVR Stdioを実行します。



02 New Project

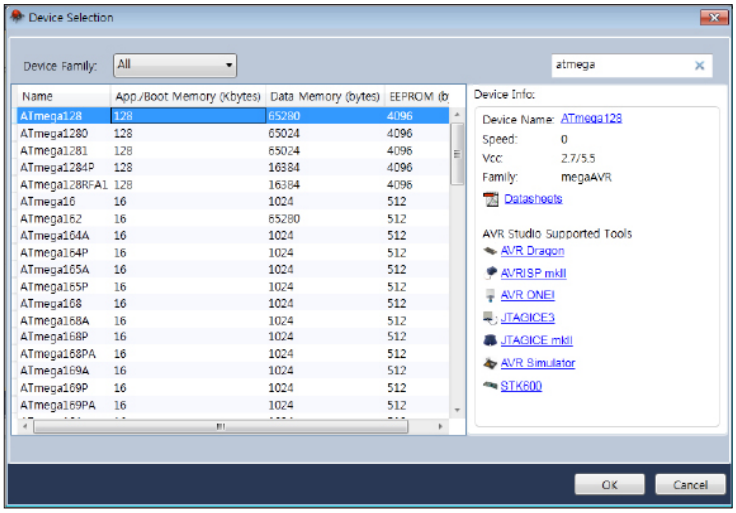
New Projectをクリックします。



03 GCC 選択

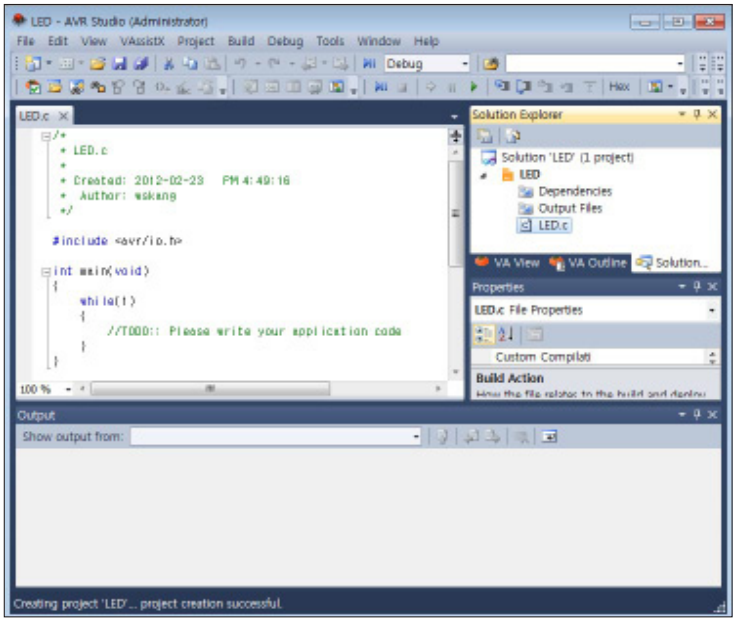
C Executable Projectを選択します。

Nameを“TimerCounter”に入力して，Create directory for solution をクリックし、OKボタンを押します。



04 ATmega 選択

右上からatmegaを検索し、リスト中のATmega128を選択します。



05 Editor

編集ウィンドウでコーディングを開始します。

次の例を見てください。

例

DRCのLEDがServo（0番ピン）からPower（6番ピン）まで順に点灯していることを繰り返す例です。ただし、ディレイ機能をタイマー/カウンタ0に実装します。

```
//ATmega128のレジスタが定義されている
#include <avr/io.h>
//interruptを使用するために必要な情報が定義されている
#include <avr/interrupt.h>

//タイマー/カウンタ0のクロック速度とTCNT0の初期値
#define TIMER0_DIVIDER                          0b00000101
#define TIMER0_TCNT0_PRELOAD                    6

//タイマー/カウンタ0で時間を計る時使用する全域 の宣言
volatile unsigned char gucTimerTick;

//タイマー/カウンタ0オーバーフローインタラプトサービスルーチン
ISR(TIMER0_OVF_vect){
        //TCNT0を初期値に再設定
        TCNT0 = TIMER0_TCNT0_PRELOAD;

        //全域 が0以外のとき1減少        if(gucTimerTick != 0){
                gucTimerTick--;
        }
}

int main(void)
{
        //8ビット の宣言
        unsigned char i;

        ///ポートCの入出力方向を設定する
        DDRC = 0b01111111;
        //ポートCの出力値を設定する
        PORTC = 0b01111111;
```

```
//全 インタラプトを無 にする
        cli();

        //TCNT0の初期値を設定
        TCNT0 = TIMER0_TCNT0_PRELOAD;
        //タイマ―/カウンタ0オ―バ―フロ―インタラプトを有 にする
        TIMSK |= (1<<TOIE0);
        //タイマ―/カウンタ0オ―バ―フロ―フラグを0にする
        TIFR |= (1<<TOV0);
        //一般モ―ド、OC0出力せずに設定したクロック速度でタイマ―/カウンタ0開始
        TCCR0 = TIMER0_DIVIDER;

        //全 インタラプトを有 にする
        sei();

        //while  の 容を無限に繰り返す
        while(1)
        {
                //LEDが順番に一つずつ点 するように動作を繰り返す
                for(i=0b00000001;i<=0b01111111;i=(i<<1)+1){
                        PORTC = ~i;             //Active Lowであるため、i の値を反 して出力

                        gucTimerTick = 100;     //200ms間を待ち
                        while(gucTimerTick);
                }
        }

        //  の形に合わせて整 値を返す
        return 1;

}
```

例の説明

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

avr/io.hヘッダファイルにはDDRX、PORTXレジスタなどAVR制御のために必要なレジスタが定義されており、avr/ interrupt.hヘッダファイルには、インタラプトを使用するための関数が定義されているので # includeステートメントを使用します。

```
#define TIMER0_DIVIDER                  0b00000101
#define TIMER0_TCNT0_PRELOAD                6
```

タイマー/カウンタ0オーバーフローインタラプトが発生する周期を設定する2項目を簡単に#defineで宣言します。この二つの意味と、この二つがどのようにインタラプト周期を決定するかは後で説明します。

```
volatile unsigned char gucTimerTick;
```

全域変数を宣言します。 main関数内で宣言した変数は、インタラプト サービス·ルーチンで使用することがないため、全域変数を宣言してmain関数とインタラプトサービスルーチンの両方で使用できるようにしています。

```
ISR(TIMER0_OVF_vect){
        TCNT0 = TIMER0_TCNT0_PRELOAD;

        if(gucTimerTick != 0){
                gucTimerTick--;
        }
}
```

タイマー/カウンタ0オーバーフローインタラプトサービスルーチンです。インタラプトが発生すると、TCNT0値は0に初期化されていますが、この値をTIMER0_TCNT0_PRELOADという任意に設定した初期値に再変更します。そして全域変数であるgucTimerTickの値が0でない場合、1を減らし、ルーチンを終了します。

```
unsigned char i;

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

LEDの値が格納される8ビットの変数iを宣言して、ポートCの0～6番ピンを出力に設定し、出力値を1にします。これにより、すべてのLEDが消灯します。

```
cli();

TCNT0 = TIMER0_TCNT0_PRELOAD;
TIMSK |= (1<<TOIE0);
TIFR |= (1<<TOV0);
TCCR0 = TIMER0_DIVIDER;

sei();
```

cli()を使用してSREGのIビットを0にすることで、全体のインタラプトを無効にします。設定を変える時にインタラプトが発生し誤動作することを防止するためです。

TCNT0の値をTIMER_TCNT0_PRELOAD値に置き換えます。前のインタラプトサービスルーチンでも見たように、TCNT0値は常にTIMER_TCNT0_PRELOAD値に初期化されます。これは、元のTCNT0が0から255に増加した後タイマー/カウンタ0オーバーフローインタラプトが発生していたのに対し、TIMER_TCNT0_PRELOADという値が255に増加した後インタラプトが発生するようにしてインタラプトが発生する周期を変えたものです。本来であれば256番タイマークロックが必要でありオーバーフローが起きたが、このようにTCNT0を初期化するとオーバーフロー周期は(256−TIMER0_TCNT0_PRELOAD)に減少します。この例ではオーバーフロー周期は250です。

TIMSKでTOIE0を1にしています。今まではレジスタの値を変更するときは、0b00000001などを直接入力していましたが、上記のような方法も可能です。avr/io.hヘッダファイルには、各ビットが何のビットに対応しているのかを格納しています。TOIE0はTIMSKの0番目のビットであるため、0に宣言されています。そのため、TIMSK|=(1<<TOIE0);のようなプログラムとTIMSKのTOIE0ビットを1に設定します。同じようにTIFRもTOV0ビットに1を使用してTOV0ビットを0に初期化します。

一般モードで動作することで、OC0出力は使用しないため、TCCR0のWGM01：0、COM01：0はすべて0になります。したがって、適切なCS02：0値TCCR0に入れると一般モードでタイマー/カウンタ動作が開始します。TIMER0_DIVIDERがそのCS02：0の値を表します。例では、0b101を使用して基本的なクロックの16Mhzを128で割っています。この場合、クロック周波数は125khzになって、8usごとにTCNT0が1ずつ増加します。8usごとにTCNT0が1ずつ増加し、250番ずつオーバーフローインタラプトが発生するので、オーバーフローインタラプト周期は2msになります。TIMER0_TCNT0_PRELOADとTIMER0_DIVIDERの値を変更すると、ユーザーがオーバーフローインタラプト周期を作成できます。

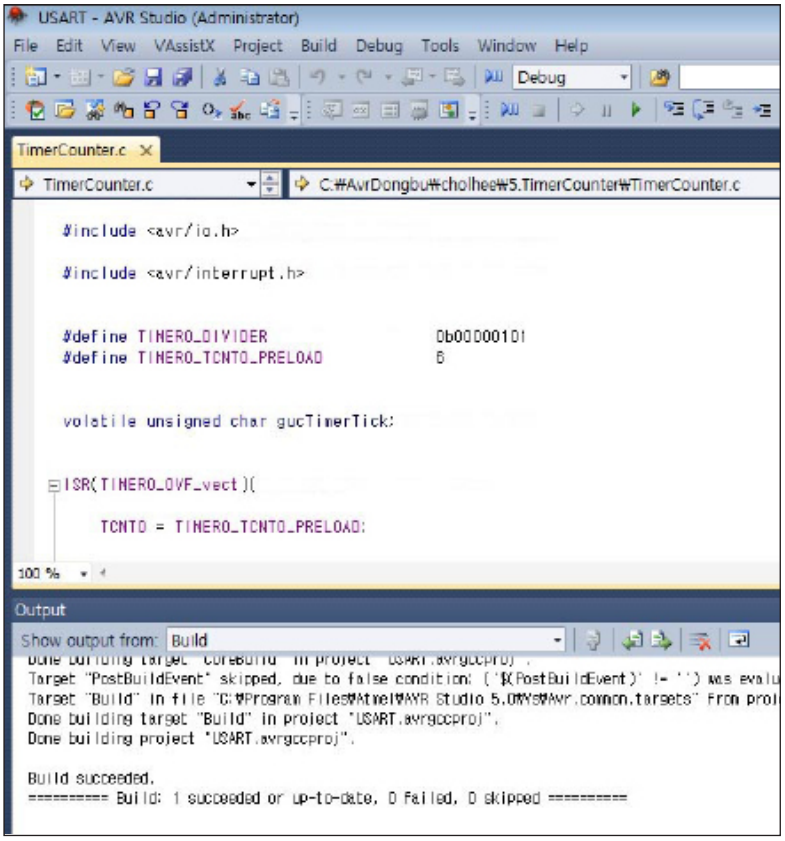
TCCR0のCS02：0の値を入れてタイマーが動作を開始すると、sei()命令を通じて全体のインタラプトを有効にしています。これからは2msによってタイマー/カウンタ0オーバーフローインタラプトが発生します。

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

        gucTimerTick = 100;
        while(gucTimerTick);
}
```

for文の内容は、Chapter1のLEDの内容と同じです。一つ違う点は、_delay_ms(200)がgucTimerTick=100;とwhile(gucTimerTick);に変わる点です。gucTimerTickを100に変えてやれば、gucTimerTickは2msごどにオーバーフローインタラプトサービスルーチンによって1つずつ減少します。したがって、gucTimerTickが0になるためには、100回のインタラプトサービスルーチンが必要です。while(gucTimerTick);はgucTimerTickが0でない場合はwhile文を反復して待つという意味なので、gucTimerTickが0でないとプログラムがwhile文を終了して再度for文の最初に戻りますが、これは200msの間待機している効果になります。_delay_ms関数の機能をタイマ/カウンタ0オーバーフローインタラプトで実装したものです。

実行

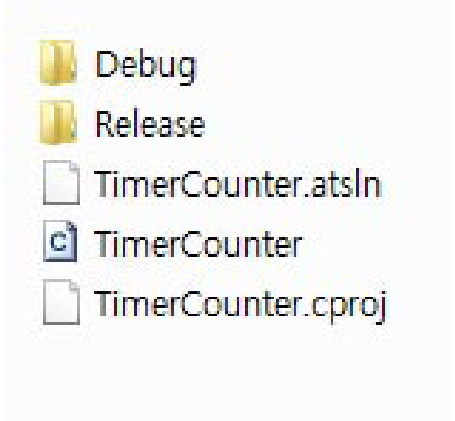
Chapter1のように、Servo LEDからPower LEDまでLEDが一つずつ追加で点灯します。LED7個も点灯した後、再度Servo LEDのみ点灯し、順番に一つずつ点灯することを繰り返します。



01 Build

Buildメニューを選択しBuild Solutionをクリックしてビルドします。

1 succeeded0 failedが出るとエラーなしでコンパイルが完了したということです。





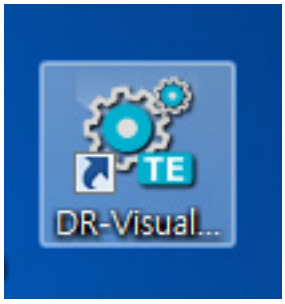
02 Debug フォルダ

Build後にソースがあるフォルダに入るとDebugフォルダが作成されています。クリックしてフォルダの中身を表示します。

03 hex 確認

Debugフォルダの中のhexファイルを確認します。Hexファイルは制御機に移植するファイルです。

DR–Visual Logicで、ファームウェアのリカバリーを行います。



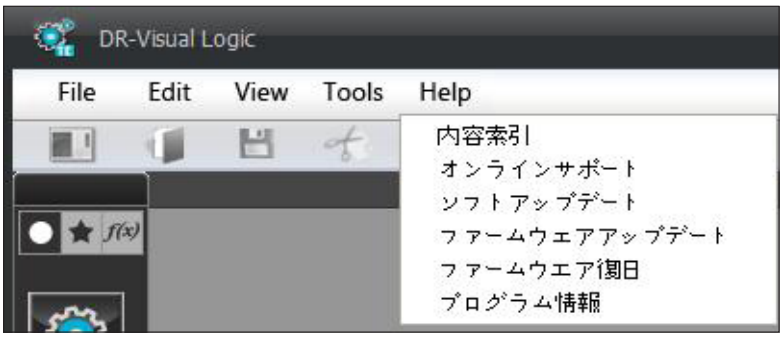
04 Visual Logic

DR–Visual Logicを実行します。



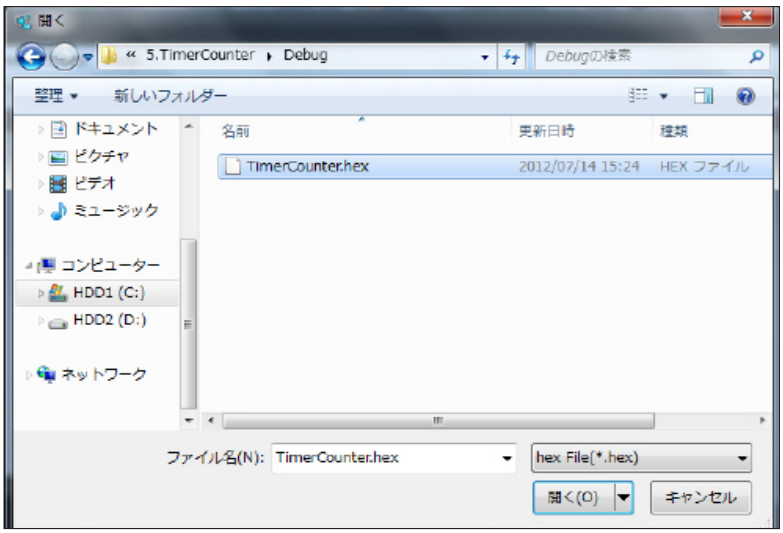
05 接続する

COM 番号を設定して接続するをクリックします。



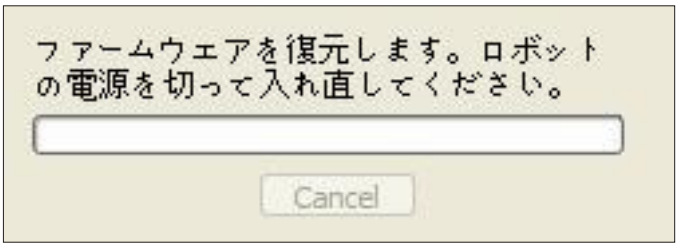
06 ファームウェアの修復

[ヘルプ]メニューからファームウェア修復をクリックします。



07 ロード

Debug フォルダにHexファイルをロードします。



08 ファームウェアの修復

“ロボットの電源を切って入れ直してください”というメッセージが出ます。コントローラの電源ボタンを一度切ってから再度入れ直してください。



09 修復の開始

ファームウェアの修復が開始されます。



10 実行

Servo LEDからPower LEDまでLEDが一つずつ追加で点灯します。LED7個も点灯した後、再度Servo LEDのみ点灯し、順番に一つずつ点灯していることを繰り返します。

ADCを使用してPSDセンサーの値を読み取る

ATmega128のアナログデジタル変換器(Analog-to-Digital Converter、ADC)を使用して、アナログPSDセンサーの出力値を読み取ります。

AVR 原理

ATmega128は、8つのチャンネル(ADC0～ADC7)に10ビットの分解能を持つ逐次比較型アナログデジタル変換器(successive approximation analog-to-digital converter)が内蔵されています。8つのチャネルはポートFと兼用で使用され、内部のマルチプレクサにより、どのポートを使用するかを選択できます。ADCの前段には、サンプル/ホールド(Sample and hold)回路が接続されており、変換が行われている間に入力されたアナログ電圧が一定に保たれるようになります。

8つのチャンネルから1つのチャンネルを単極性のアナログ入力として受け取り、変換する動作モードと、指定した二つのチャンネルの電圧差を変換する差動動作モードがあります。差動動作モードではADC1、ADC0とADC3、ADC2の二組は電圧の差を1倍、10倍、200倍に増幅して変換する機能もあります。

ADCは、ATmega128の電源ピンのVCCで動作するのではなく、ノイズを減らすために、独立した電源ピンのAVCCに電源入力を受けて動作します。しかし、AVCCがVCCより0.3V以上の差がなければなりません。

ADCに関連するレジスタは、以下のとおりです。

■ ADMUX : ADC マルチプレクサ選択レジスタ

ビット	7	6	5	4	3	2	1	0
	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0
読み取り/書き込み	W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初期値	0	0	0	0	0	0	0	0

-ビット7:6番REFS1 : 0 : 基準の選択ビット

ADCは、アナログ電圧を基準電圧のVREFと比較して相対的な大きさを10bitのデジタル値に変換するデバイスであるため、デジタル値を見てアナログ電圧を取得するには基準電圧のVREFを知る必要があります。ATmega128には、AREFというピンがあるので、通常の場合は、このピンに入力されたアナログ電圧が基準電圧VREFとなります。また、内部レジスタ(ADMUX)の設定を使用してAVCCに入ってきた入力電圧を基準電圧として使用するか、またはATmega128に内蔵されている2.56Vの基準電圧を使用します。

REFS1	REFS0	基準電圧の選択
0	0	AREFの入力電圧
0	1	AVCCの入力電圧
1	0	使用しない
1	1	2.56Vの内部電圧

- ビット5番ADLAR：ADCの結果を左に配置

ADCに変換された10ビットデータは、ADC Data RegisterのADCHとADCLに格納されます。 ADLARが0の場合は、データの下位8ビットがADCLと、上位2ビットがADCHの1:0番のビットに格納されます。ADLARが1の時は、データの上位8ビットがADCHに、下位2ビットがADCLの7:6番のビットに格納されます。

- ビット4:0番MUX4：0：アナログチャネルと利得を選択するビット

この5つのビットの値に応じてどのような入力を使用して、どのようなゲインに電圧を増幅して変換するか選ぶことができます。MUX4：0の値に応じてどのような入力を使用するのかは下表のとおりです。

MUX4:0	単極性入力	差動入力		
		+端子	-端子	ゲイン
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 制御、ステータスレジスタ

ビット	7	6	5	4	3	2	1	0
	ADEN	ADSC	ADFR	ADIF	ADIE	ADPS2	ADPS1	ADPS0
読み取り/書き込み	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初期値	0	0	0	0	0	0	0	0

-ビット7番ADEN : ADC有効

このビットは、ADCの動作を許可または禁止します。1に設定すると、ADCは動作します。0に設定すると、変換中でも、ADCを消灯します。

- ビット6番ADSC : ADC変換を開始

ADCの変換を開始するビットです。1に設定した場合にADCが変換を開始します。単一の変換モードでは1に設定すると、ADSCは変換が完了すると、0になります。フリーランニングモードでは、ADSCは1に設定すると、最初に変換が開始され、次から変換を繰り返します。普段ADC変換は13回のADCクロックサイクルが必要であり、活性化後初の変換では、ADCの初期化プロセスのために25回のADCクロックサイクルがかかります。ADCのクロックについては、以後のADPS2 : 0ビットで説明します。

- ビット5番ADFR : ADCフリーランニングの選択

ADCのフリーランニングモードを設定するかどうかを指定します。 1に設定すると、ADCはADSCが1である間は変換を繰り返してデータレジスタを更新します。0に設定するとフリーランニングモードが終了し、単一の変換モードに戻ります。

- ビット4番ADIF : ADCインタラプトフラグ

このビットは、ADC変換が終了してデータレジスタが更新されると1に設定されます。もしSREGのIビットとADIEビットを1に設定すると、ADC変換完了割り込みが発生します。インタラプトが発生して対応するISRに入ると、ADIFは自動的に0に初期化されます。ソフトウェア的に手動でこのビットを0に変えるには、このビットに1を使用する必要があります。

- ビット3番ADIE : ADCインタラプトを有効に

このビットは、ADC変換が完了したときインタラプトを許可するように設定するビットです。このビットとSREGのIビットが1の場合、ADC変換が完了したときADC変換完了インタラプトが発生します。

- ビット2:0番ADPS2 : 0 : ADCプリスケーラ選択ビット

このビットは、ATmega128のクロック周波数をいくつかに分けてADCに入力するかどうかを指定します。基本的にADCの回路は10ビットの解像度でADC変換を完了するために、50～200kHzの周波数が必要です。もし10ビットよりも低い解像度が必要な場合は、周波数を高くすると、サンプル周期を増やすことができます。ATmega128のADCには、100kHz以上のどんなクロック周波数でも分けて、50～200kHzの周波数を得ることができるようにプリスケーラを提供しており、ユーザーが自分のクロック周波数に合わせてADPS2 : 0を設定して、範囲内のADCクロック周波数を設定する必要があります。たとえば、DRCで使用されるクロック周波数は16MHzなので、128で割ると、125kHzを作成できます。

ADPS2	ADPS1	ADPS0	分周比
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 データレジスタ

ADLAR = 0の場合

ビット	15	14	13	12	11	10	9	8
ADCH	–	–	–	–	–	–	ADC9	ADC8
読み取り/書き込み	R	R	R	R	R	R	R	R
初期値	0	0	0	0	0	0	0	0
ビット	7	6	5	4	3	2	1	0
ADCL	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0
読み取り/書き込み	R	R	R	R	R	R	R	R
初期値	0	0	0	0	0	0	0	0

ADLAR = 0の場合

ビット	15	14	13	12	11	10	9	8
ADCH	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2
読み取り/書き込み	R	R	R	R	R	R	R	R
初期値	0	0	0	0	0	0	0	0
ビット	7	6	5	4	3	2	1	0
ADCL	ADC1	ADC0	–	–	–	–	–	–
読み取り/書き込み	R	R	R	R	R	R	R	R
初期値	0	0	0	0	0	0	0	0

– ADC9：0：ADC変換結果

ADCの変換が終わったら、ADCHとADCLの2つのレジスタから値を読み取ります。データは、単極性·モードでは、0～1023の符号なし値として格納され、差動モードでは、–512～+511の2の補数の値として格納されます。 ADCHとADCLが格納された10ビット値をADCとすると、動作モードに応じたADCの数式は以下のとおりです。

単極性·モード: $ADC = \frac{V_{IN} \times 1024}{V_{REF}}$

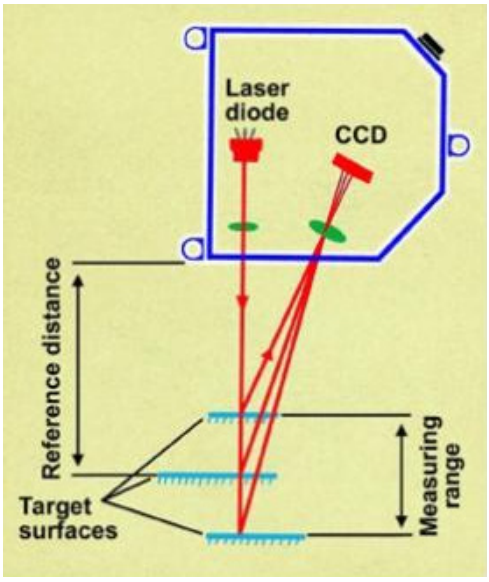
差動モード : $ADC = \frac{(V_{POS} - V_{NEG}) \times GAIN \times 512}{V_{REF}}$

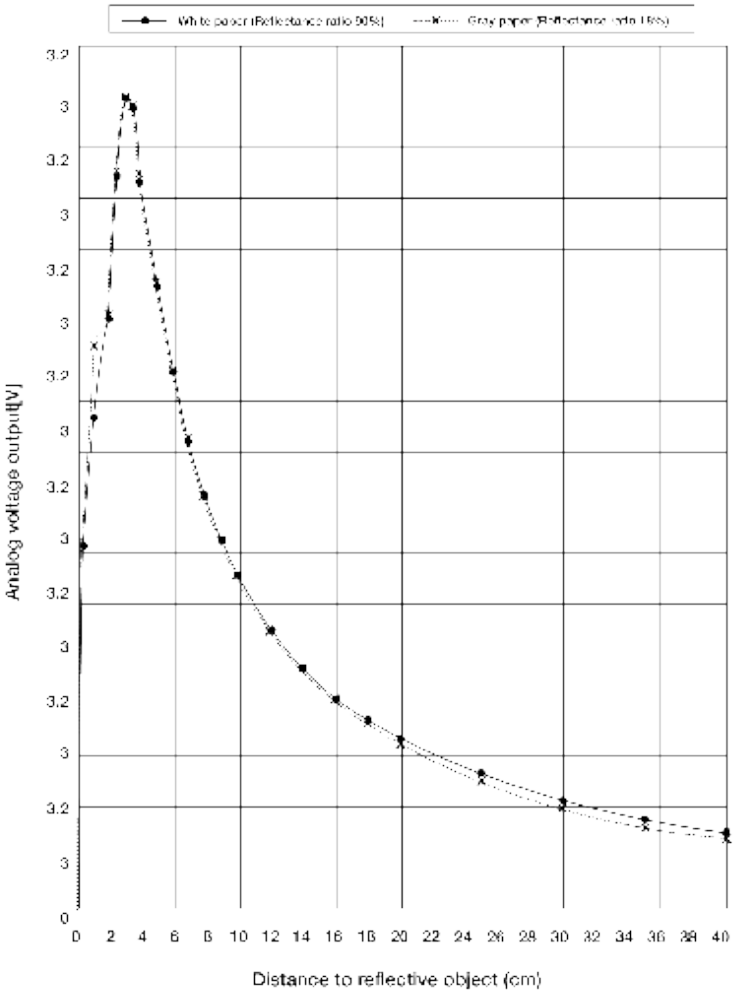
10ビットではなく、8ビット解像度で十分であればADLAR=1に設定し、ADCHだけを読んで使用します。しかし、2つのレジスタの内容をすべて使用するのであれば、常にADCLを先に読んでADCHを後に読むべきです。

H/W 原理

今回の例では、アナログPSDセンサーモジュールをDRCに接続してADC変換を介して、センサの出力値を読みます。PSDセンサーは赤外線を利用してセンサーの前の障害物までの距離を計るセンサーです。PSDセンサーは、2つの丸いウィンドウがありますが、一つは発光部であり、もう一つは受光部です。発光部は、定期的に赤外線を撃ち、手前の物体から反射され受光部に再表示します。図00に示すように物体との距離に応じて受光部で反射され入る光の角度が変わり受光部に結ばれる位置が変わります。PSDセンサーは、受光部に赤外線が入ってきた場所に応じて出力電圧を変化させて、この電圧でセンサーと障害物の距離を計っています。PSDセンサーは大きく分けて、アナログタイプとデジタルタイプがあり、モデルに応じて、測定距離も多様であり、距離の認識や障害物の検出に多く応用されています。私たちが使用するセンサーは、40cmまでの測定範囲を持つアナログタイプのPSDです。距離に応じた出力電圧は、次の図00に示します。

DRCの左側センサーポートはADC6(PF6)に対応しており、右側センサーポートはADC7(PF7)に対応しています。アナログPSDセンサーを左センサーポートに接続する場合は、ADC6を通ってセンサーの値を読み込む必要があります。





ハードウェアの準備



01 接続

DRC制御機アダプターで電源を接続し、USB to Serial CableでPCとDRCを接続します。



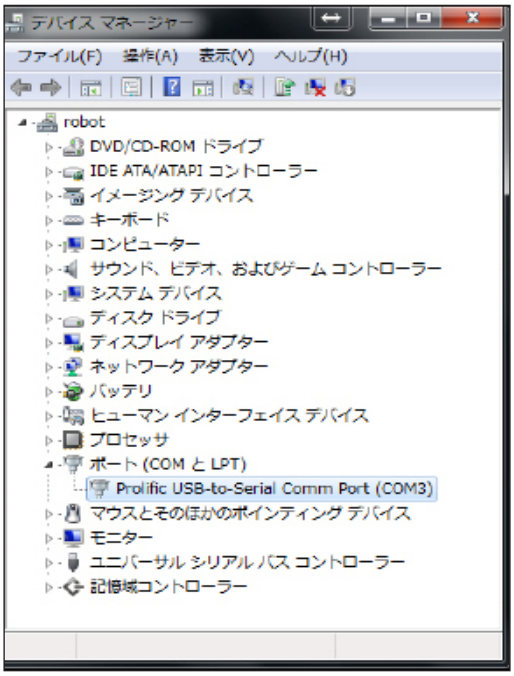
02 PSD センサーの準備

PSD センサーを準備します。



03 PSD センサーの接続

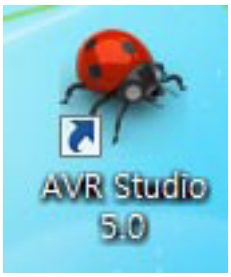
PSD センサーを制御機の左側のポートに接続します。



04 COM port 接続

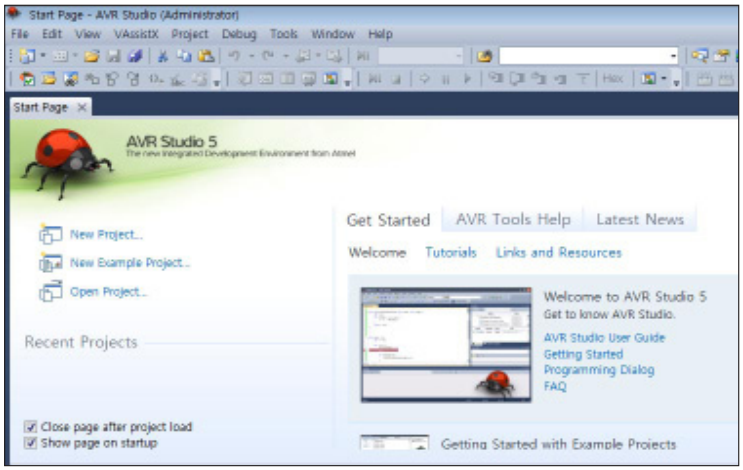
マイコンピュータ>プロパティ>デバイスマネージャでCOM接続を確認します。

AVR Studio の実行とコーディングの準備



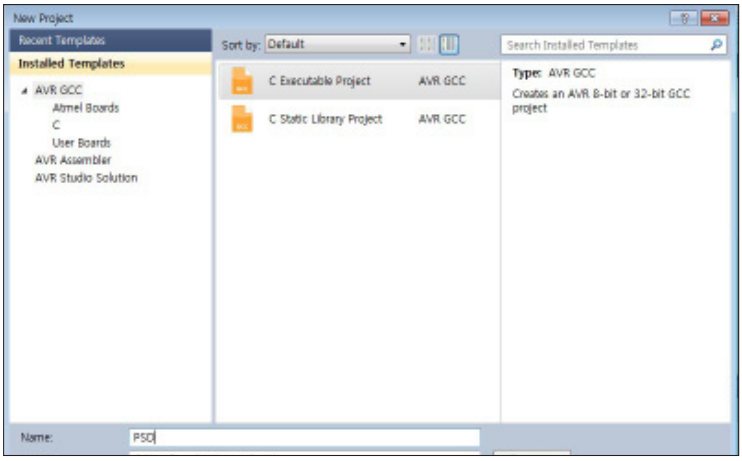
01 実行

AVR Stdio を実行します。



02 New Project

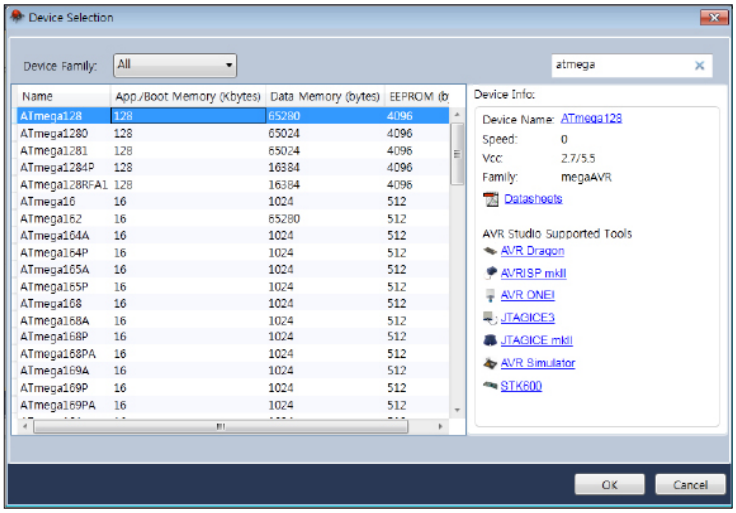
New Projectをクリックします。



03 GCC 選択

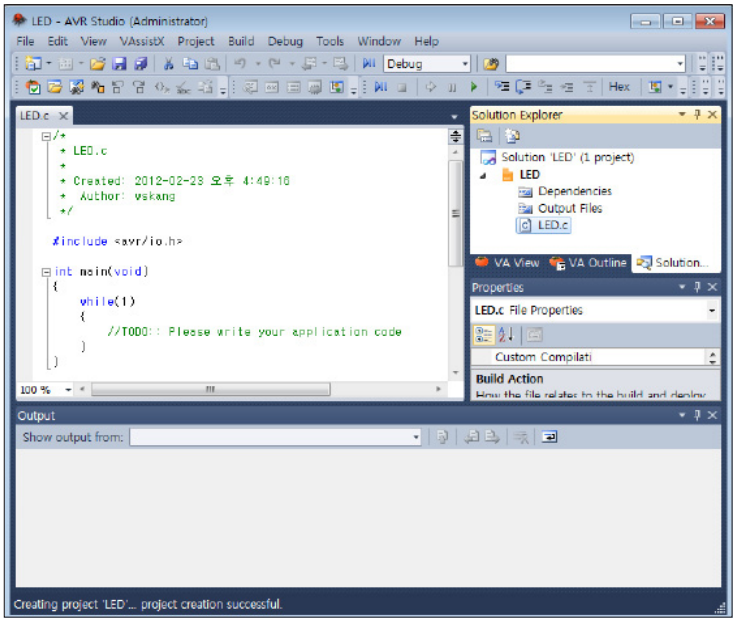
C Executable Project を選択します。

Nameを“PSD”に入力して, Create directory for solution をクリックし、OKボタンを押します。



04 ATmega 選択

右上からatmegaを検索し、リスト中のATmega128を選択します。



05 Editor

編集ウィンドウでコーディングを開始します。

次の例を見てください。

例

DRCの左側センサーポートに接続されたアナログPSDセンサーの出力値をADCを使用して変換し、距離の値が10cmより近い場合は、LEDをオンにする例です。

PSD.c（ソースファイル名と位置）

```
//ATmega128のレジスタが定義されている
#include <avr/io.h>

int main(void)
{
	//16ビット　の宣言
	int iADC;

	//ポートCの入出力方向を設定する
	DDRC = 0b01111111;
	//ポートCの出力値を設定する
	PORTC = 0b01111111;

	//VREF=AREF入力電 、左 えオフ、ADC6番 極性入力
	ADMUX=0b00000110;
	//ADC 有 、ADIFの初期化、クロック分周比 128
	ADCSRA=0b10010111;

	//while の 容を無限に繰り 返す
	while(1)
	{
		//ADC 換開始
		ADCSRA |= (1<<ADSC);
		//ADC 換が完了するまで待つ
		while(!(ADCSRA & (1<<ADIF)));
		//ADCLとADCHの値を み取って格納
		iADC = ADCL | (ADCH<<8);
		//ADIFの初期化
		ADCSRA |= (1<<ADIF);

		// 1.3/5*1024 = 266.24
		if(iADC > 266)
			PORTC = 0b00000000;
		else
			PORTC = 0b01111111;
	}

	// の形に合わせて整 値を返す
	return 1;
}
```

例の説明

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

avr/io.h ヘッダファイルにはDDRX、PORTXレジスタなどAVR制御のために必要なレジスタが定義されているので # include 文を使用しています。

```
int iADC;

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

ADC値が格納される16ビットの変数iADCを宣言して、ポートCの0～6番ピンを出力に設定し、出力値を1にします。これにより、すべてのLEDが消灯します。

```
ADMUX=0b00000110;
ADCSRA=0b10010111;
```

ADCを単極性･モードでADC6ポートからアナログ入力を読み込むためのレジスタ設定をします。ADMUXから見ると、DRCの回路上での基準電圧は、ATmega128の動作電圧の5Vと同じようにAREFピンが接続されているのでREFS1：0ビットは0b00に設定し、左側整列を使用していないため、ADLARを0に設定します。そしてADC6ポートは単極性･モードで読み取るものなのでMUX4：0ビットは0b00110に設定します。
ADCSRAでは、まずADCを有効にするためにADENビットを1に設定して、まだ変換開始を行うことなく、フリーランニングも使用しないのでADSC、ADFRは0に設定します。ADIFの場合は、もしかしたら1に設定されている可能性がありますので0に再初期化するために、1を使用して、インタラプトを使用していないため、ADIEは0に設定します。最後に、ATmega128の16MHzクロック周波数をADCに適した50～200kHzで
分割のためのADPS2：0ビットを0b111に設定して128で割った125kHzをADCクロックに供給するように設定します。

```
ADCSRA |= (1<<ADSC);
while(!(ADCSRA & (1<<ADIF)));
iADC = ADCL | (ADCH<<8);
ADCSRA |= (1<<ADIF);
```

while文では、ADCの変換を繰り返しています。まずADCSRAのADSCビットを1にしています。これにより、ADC変換が開始されます。
while(！(ADCSRA＆(1<< ADIF)));は！(ADCSRA＆(1<< ADIF))を満足するとき、何もせずに待機するよう命令します。
！(ADCSRA＆(1<< ADIF))を満足するのはADCSRAのADIFが0の時、つまり、ADIFビットが0の間、待って1になる瞬間while文の次に進むという意味です。

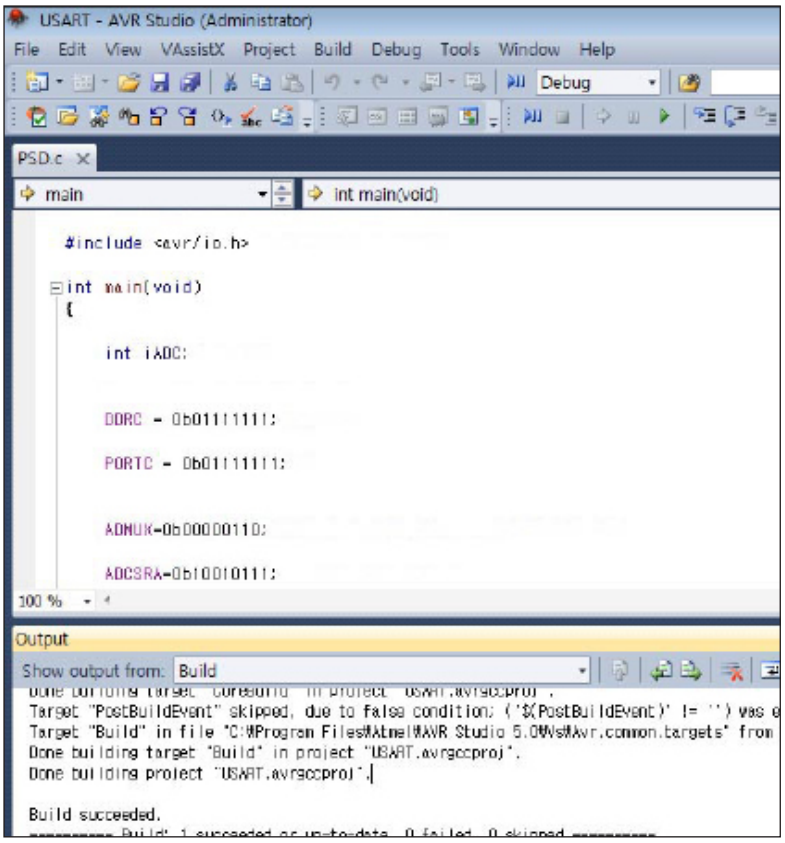
ADC変換が完了し、ADCLとADCHが更新されADIFビットが1になった後は、ADCLとADCHの値を読み取りiADCに保存します。ADCHはADC値の上位1バイトに対応するため、8ビット（1バイト）分だけ左に<<（shift left）演算をしてADCLと|（bitwise OR）演算iADCに保存すると、10ビットのデータをiADCに合わせて保存します。レジスタの説明で述べたように、ADCLを先に読んだ後にADCHを読むので、式ではADCLを先に使ってADCLレジスタの値を先に読むようにします。
ADC変換が完了インタラプトを使用していないため、ADIFビットは自動的に0に設定されていません。そのため、ADCの値を読み込んだ後にインタラプトフラグを0に初期化する必要があります。ADCSRAのADIFビットに1を使用して、このビットを0に初期化します。

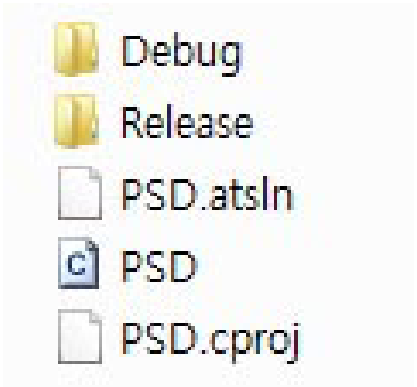
```
if(iADC > 266)
        PORTC = 0b00000000;
else
        PORTC = 0b01111111;
```

iADCの値に応じてLEDが点滅します。図00のPSDセンサーの距離に応じた電圧値のグラフを見ると、物体との距離が10cmのときセンサーの出力値は約1.3Vになります。1.3Vに対応するADC出力値は、$ADC = \frac{V_N \times 1024}{V_{REF}}$ ですので、式に沿って計算すると1.3×1024÷5=266.24となります。四捨五入した場合、入力電圧が1.3Vのとき、ADC値は266程度になります。
iADCが266より大きい場合、物体が10cmより近くにあるという意味で、小さければ10cmより遠くにあるという意味です。 if–else文では、物体が10cmより近いと、すべてのLEDをオンにし、10cmよりも遠いと、すべてのLEDをオフにします。

実行

Chapter1のように、Servo LEDからPower LEDまでLEDが一つずつ追加で点灯します。LED7個が点灯した後、再度Servo LEDのみ点灯し、順番に一つずつ点灯することを繰り返します。







01 Build

Buildメニューを選択しBuild Solutionをクリックして Buildします。

1 succeeded0 failedが出るとエラーなしでコンパイルが完了したということです。

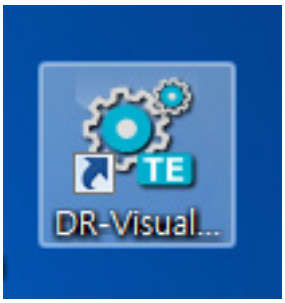
02 Debug フォルダ

Build後にソースがあるフォルダに入るとDebugフォルダが作成されています。クリックしてフォルダの中身を表示します。

03 hex 確認

Debugフォルダの中のhexファイルを確認します。Hexファイルは制御機に移植するファイルです。

DR-Visual Logicで、ファームウェアのリカバリーを行います。



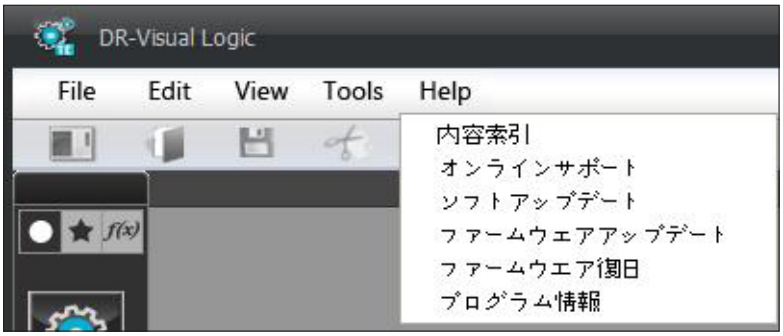
04 Visual Logic

DR–Visual Logicを実行します。



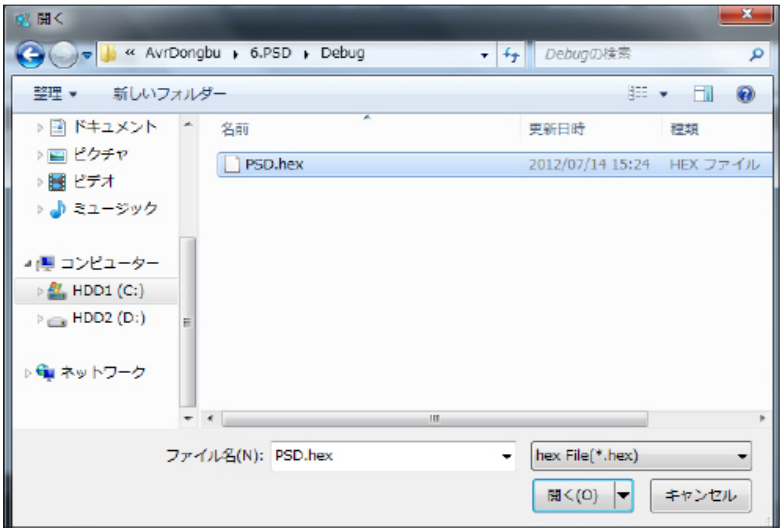
05 接続する

COM 番号を設定して接続するをクリックします。



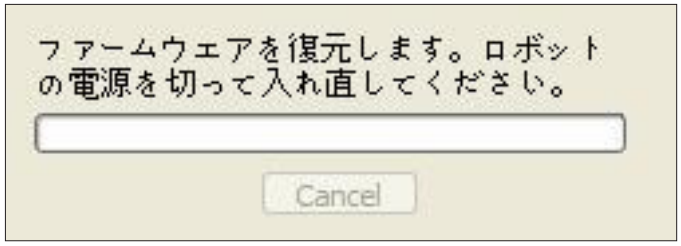
06 ファームウェアの修復

[ヘルプ]メニューからファームウェア修復をクリックします。



07 ロード

Debug フォルダにHexファイルをロードします。



08 ファームウェアの修復

“ロボットの電源を切って入れ直してください”というメッセージが出ます。コントローラの電源ボタンを一度切ってから再度入れ直してください。



09 修復の開始

ファームウェアの修復が開始されます。

（何もない場合，LEDが消灯）



10 実行

PSDセンサーの10cm以内に手を出すとDRCのすべてのLEDが点灯し、手を離したときにLEDが消灯します。

（手を近づけるとLEDが点灯）



USARTを使用してシリアル通信をする

ATmega128の USART(Universal Synchronous and Asynchronous serial Receiver and Transmitter)を使用して、PCとDRCの間でシリアル通信(直列通信)をします。

AVR 原理

ATmega128には、USART0、USART1の二つのUSARTが内蔵されています。USARTは一般非同期、倍速非同期、マスター同期、スレーブ同期の4つのモードで使用できます。名前からわかるように、非同期通信と同期通信の両方をサポートしています。私たちが使用するモードは、非同期通信で、その中の2倍速非同期を使用しています。ATmega128のUSARTはクロック生成ブロック、送信ブロック、受信ブロック、レジスタブロックの4つの部分から構成されています。クロック生成ブロックは、USARTの通信を行うときに必要なクロックを発生する回路です。非同期通信を行うときは、自主的に発生するクロックを使用しますが、同期通信を行うときは、マスターでは、独自のクロックを発生させてスレーブに出力し、スレーブでは、そのクロック信号を受けて同期して通信します。 送信ブロックは、USARTの送信部がレジスタに使用された8ビットの並列データを直列に送信する役割をします。受信ブロックは、USARTの受信部が直列に受信した直列データを並列データに変換してレジスタに保存する機能をします。レジスタブロックは、USARTを制御するための各種の制御ビットと設定ビット、そして一部のデータを保存するビットとして使用します。

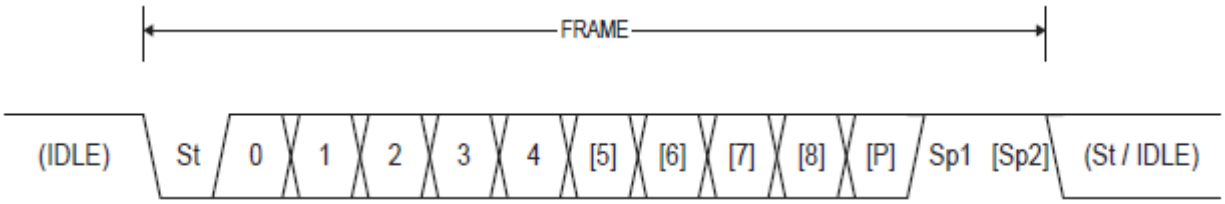


図00は、USARTの送受信フレームを示しています。フレームは、シリアル通信でデータビットと同期化ビット(開始ビット、停止ビット)、そして付加的なパリティビットまでを意味します。シリアルフレームの形式は、次の事項のうちの一つを選んで組み合わせることで、作成されます。

- 1つの開始ビット
- 5、6、7、8、または9個のデータビット
- 偶数パリティビット1つまたは奇数パリティビット1個または使用しない
- 1または2つの停止ビット

よく使用される形式は、8個のデータビット、パリティビット使用なし、1ストップビットであり、DRCとHerkuleXでもこの形式を使用しています。

通信が行われていない状態(IDLE)では、データラインはHIGHになります。開始ビット(St)は、常にLOWでなければならず、その後データビット(0~[8])が送信されます。データビットは、種類に応じて5~9ビットが送信されます。図00の括弧は種類に応じて括弧をしているビットはない可能性があるという意味です。その後の設定に応じてパリティビット([P])を送信することもあり、そうでない場合があります。その後停止ビット(Sp1~[Sp2])において1~2ビットのデータ転送とフレームが終了し、データラインは再びIDLE状態に戻るか、次データの開始ビットを開始します。

よく使用される形式である8個データビット、パリティビットは使用しません。1個停止ビットを使用すると、フレームの全ビット数開始ビット1+データビット8+停止ビット1で計10ビットになります。
USARTに関連するレジスタは、以下のとおりです。レジスタ名のnは0または1で、nの席に0と1を入れると、それぞれUSART0とUSART1に対抗するレジスタを意味します。

UDRn : USART 送信/受信データレジスタ

ビット	7	6	5	4	3	2	1	0
Read	RXBn[7:0]							
Write	TXBn[7:0]							
読み取り/書き込み	R	R/W	R	R	R	R	R/W	R/W
初期値	0	0	1	0	0	0	0	0

USARTn 送信データバッファレジスタとUSARTn受信データバッファレジスタは同じUDRnという名前のデータレジスタにアクセスします。UDRnにデータを使用して、その値を送信データレジスタに使用して、UDRnからデータを読み取ると、受信データレジスタの値が読み込まれます。
送信データレジスタは、UCSRAnのUDREnビットが1の場合にのみ値を使用しています。送信が有効になっているときに、データを使用すると、そのデータは、シフトレジスタに移動して1ビットずつ直列に送信されます。
受信データレジスタは、2段階FIFOで構成されており2つの文字データを受けます。FIFO動作の特性上、受信データレジスタは、一度アクセスされるたび、その次のデータの値が変わってしまいます。したがって、UDRnからデータを読み取るときやRXBn[7:0]のうち、特定のビットにアクセスする際にUDRnを読み取って別の変数に保存して処理する方が安全です。

UCSRnA : USART 制御、ステータスレジスタ A

ビット	7	6	5	4	3	2	1	0
	RXCn	TXCn	UDREn	FEn	DORn	UPEn	U2Xn	MPCMn
読み取り/書き込み	R	R/W	R	R	R	R	R/W	R/W
初期値	0	0	1	0	0	0	0	0

- ビット7番RXCn : USART受信完了

受信完了状態を示すビットで受信データレジスタにデータが入ってきて受信を終了すると1に設定され、データがない場合は0に設定されます。RXCnビットはUSARTn受信完了インタラプタを起こすこともあります。

-ビット6番TXCn : USART送信完了

送信完了の状態を示すビットで、送信データレジスタ内のデータがすべて送信された場合1に設定されます。 USARTn送信完了インターラプト を起こすのに使用でき、インターラプト サービスルーチンに入るときに0に初期化されます。また、このビットに1を使用すると、手動で0に初期化されます。

- ビット5番UDREn : USARTデータレジスタエンプティー

このビットは、送信バッファ(UDRn)が新しいデータを受け取る準備ができていることを通知します。 UDREnが1のときUDRnにデータを使用すると、データが送信されます。 UDREnはUSARTnデータレジスタエンプティー インターラプト を発生させることに使用できます。

- ビット4番FEn : フレームエラー

受信バッファに入ってきた次の文字にフレームエラーが起きたことを知らせるビットです。データフレームの最初の停止ビットは常に1でなければなりませんが、受信したデータの最初の停止ビットが0の場合は1に設定されます。

-ビット3番DORn : データオーバーラン

データが受信されている間、オーバーランエラーが発生したことを示すビットです。受信バッファの2段階FIFOにすでに文字データの2つが入っている状態で、別の開始ビットが受信ピンに検出されたときに設定されます。

-ビット2番UPEn : パリティエラー

データを受信中にパリティエラーが発生したことを示すビットです。このエラーは、UCSRnCレジスタのUPMn1 : 0ビットでパリティを使用するように設定した場合にのみ発生します。

- ビット1番U2Xn : 2倍速USART送信 速度

このビットは、非同期通信のみ効果があり、同期通信時には0に設定する必要があります。このビットに1を使用すると、クロック生成ブロックでUSARTのクロックを発生させると分周比を16から8にすることで、同じUBRRL、UBRRH値USARTの通信速度を2倍にできます。このビットが0に設定されている場合分周比はそのまま16です

- ビット0番MPCMn : マルチプロセッサ通信モード

このビットは、マルチプロセッサ通信モードを活性化します。 MPCMnビットを1に設定するとUSARTnから受信したデータのアドレス情報が含まれていないデータはすべて無視されます。複数のプロセッサを一つの通信回線で制御するときに使用されます。

■ UCSRnB : USART 制御、ステータスレジスタ B

ビット	7	6	5	4	3	2	1	0
	RXCIEn	TXCIEn	UDRIEn	RXENn	TXENn	UCSZn2	RXB8n	TXB8n
読み取り/書き込み	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W
初期値	0	0	0	0	0	0	0	0

- ビット7番RXCIEn : 受信完了インターラプト 活性化

このビットに1を使用すると、RXCnフラグがインターラプトを起こすように活性化されます。SREGのIビットとRXCIEnビットが1に設定され、UCSRnAのRXCnフラグが1になるとUSARTn受信完了インターラプト が発生しています。

- ビット6番TXCIEn : 送信完了インターラプト 活性化

このビットに1を使用すると、TXCnフラグがインターラプト を起こすように活性化されます。 SREGのIビットとTXCIEnビットが1に設定され、UCSRnAのTXCnフラグが1になるとUSARTn送信完了インターラプト が発生します。

- ビット5番UDRIEn：データレジスタエンプティー インターラプト 活性化

このビットに1を使用すると、UDREnフラグがインターラプトを起こすように活性化されます。SREGのIビットとUDRIEnビットが1に設定され、UCSRnAのUDREnフラグが1になるとUSARTnデータレジスタエンプティーインターラプトが発生します。

- ビット4番RXENn：受信活性化

このビットに1を使用すると、USARTnの受信ブロックが活性化されます。RXENnビットが1に設定すると、ATmega128のRxDnピンI/ Oポートの設定に関係なく、直列データ受信ピンが動作します。

- ビット3番TXENn：送信活性化

このビットに1を使用すると、USARTnの送信ブロックが活性化されます。TXENnビットを1に設定すると、ATmega128のTxDnピンI/ Oポートの設定に関係なく、直列データ送信ピンが動作します。データを送信しているときTXENnに0を使用すると、現在送信中のデータを送ってから送信ピンを無効にします。

- ビット2番UCSZn2：文字のサイズ

このビットは、UCSRnCのUCSZn1：0で、データのサイズを設定するために使用されます。詳細については、UCSZn1：0ビットで説明します。

- ビット1番RXB8n：受信データビット8

RXB8nビットはデータビット数が9個に設定されて動作するとき、受信したデータビットの9ビット目（8ビット）が格納されています。常にUDRnに近づくようにして受信データの下位ビットRXBn[7:0]を読む前にRXB8nを先に読み込み必要があります。

- ビット0番TXB8n：送信データビット8

TXB8nビットはデータビット数が9個に設定されて動作するとき、送信データのビットのうちの9番目のビット（8ビット）を表しています。常にUDRnに送信データの下位ビットTXBn[7:0]を使用する前に、TXB8nを最初に使用する必要があります。

■ UCSRnC：USART制御、ステータスレジスタ C

ビット	7	6	5	4	3	2	1	0
	-	UMSELn	UPMn1	UPMn0	USBSn	UCSZn1	UCSZn0	UCPOLn
読み取り/書き込み	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初期値	0	0	0	0	0	1	1	0

- ビット6番UMSELn：USARTモード選択

UMSELnはUSARTの非同期通信に使用する同期通信で使用するかどうかを選ぶのビットです。 0の場合、非同期通信、1の場合、同期通信に設定されます。

- ビット5:4番UPMn1：0：パリティモード

パリティビットの生成と検査の形式を設定するビットです。パリティを使用するように設定すると、送信ブロックは自動的にパリティビットを作成し、データビットの後に送信し、受信ブロックは、受信データのパリティを検査します。検査の結果、エラーが発生した場合には、UCSRnAレジスタのUPEnフラグが1に設定されます。

UPMn1	UPMn0	パリティモード
0	0	パリティビットなし
0	1	（使用しない）
1	0	偶数パリティビットを使用する
1	1	奇数パリティビットを使用する

- ビット3番USBSn：停止ビットを選択する

停止ビットの設定を選ぶビットです。0に設定すると停止ビットが1個、1に設定すると、2個に設定されます。

- ビット2:1番UCSZn1：0：文字のサイズ

UCSRnBレジスタのUCSZn2と、送信ブロックと受信ブロックで使用されるフレームのデータビット数（文字サイズ）を設定するビットです。

UCSZn2	UCSZn1	UCSZn0	文字サイズ
0	0	0	5ビット
0	0	1	6ビット
0	1	0	7ビット
0	1	1	8ビット
1	0	0	（使用しない）
1	0	1	（使用しない）
1	1	0	（使用しない）
1	1	1	9ビット

- ビット0番UCPOLn：クロック極性

このビットは同期モードでのみ使用され、非同期モードでは0に設定する必要があります。　UCPOLnビットは同期クロック(XCKn)とデータ出力の変化点/データ入力のサンプリング時点の関係を指定します。

UCPOLn	データ出力の変化時点	データ入力のサンプリング時点
0	上昇エッジの時	下降 エッジの時
1	下降 エッジの時	上昇エッジの時

UBRRnL, UBRRnH : USART ボーレート·レジスタ

ビット	15	14	13	12	11	10	9	8
UBRRnH	–	–	–	–	UBRRn[11:8]			
読み取り/書き込み	R	R	R	R	R/W	R/W	R/W	R/W
初期値	0	0	0	0	0	0	0	0
ビット	7	6	5	4	3	2	1	0
UBRRnL	UBRRn[7:0]							
読み取り/書き込み	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初期値	0	0	0	0	0	0	0	0

– ビット11:0番UBRRn11 : 0 : USARTnボーレート·レジスタ

この12ビットの値は、USARTnのボーレート(baud rate)、つまり、通信速度を設定します。USART通信のボーレートは、bps(bit per second)で表します。つまり、1秒間いくつのビットを送信するかどうかを、通信速度の尺度として使用します。よく使用するボーレートは2400、4800、9600、14400、19200、38400、57600、115200、230400、460800、921600 bpsなどがあり、DRCとHerkuleXは115200 bpsの標準ボーレートで使用します。通信モードとU2Xnビットの値に基づいたボーレートとUBRRの計算式は以下のとおりです。

動作モード	ボーレート計算式	UBRR計算式
一般的な非同期モード(U2Xn = 0)	$BAUD = \frac{f_{osc}}{16\ (UBRR+1)}$	$UBRR = \frac{f_{osc}}{16\ BAUD} - 1$
2倍速非同期モード(U2Xn = 1)	$BAUD = \frac{f_{osc}}{8(UBRR+1)}$	$UBRR = \frac{f_{osc}}{8BAUD} - 1$
マスター同期モード	$BAUD = \frac{f_{osc}}{2(UBRR+1)}$	$UBRR = \frac{f_{osc}}{2BAUD} - 1$

ここでBAUDはボーレート、fOSCはシステムクロック周波数、UBRRはUBRRレジスタの値です。

	f_{osc} = 16.0000 MHz			
	U2X = 0		U2X = 1	
Baud Rate (bps)	UBRR	Error	UBRR	Error
2400	416	-0.1%	832	0.0%
4800	207	0.2%	416	-0.1%
9600	103	0.2%	207	0.2%
14.4k	68	0.6%	138	-0.1%
19.2k	51	0.2%	103	0.2%
28.8k	34	-0.8%	68	0.6%
38.4k	25	0.2%	51	0.2%
57.6k	16	2.1%	34	-0.8%
76.8k	12	0.2%	25	0.2%
115.2k	8	-3.5%	16	2.1%
230.4k	3	8.5%	8	-3.5%
250k	3	0.0%	7	0.0%
0.5M	1	0.0%	3	0.0%
1M	0	0.0%	1	0.0%
Max (1)	1 Mbps		2 Mbps	

上の図は、DRCのシステムクロック周波数が16MHzの時のUBRR値とボーレートの関係を表した表です。DRCとHerkuleXの標準的なボードレートである115200bpsを設定するには、一般モードでは8を、2倍速モードで16を設定する必要があります。DRCとHerkuleXでは誤差が少ないので、2倍速モードで16を使用しています。

H/W 原理

前に見たように、ATmega128にはUSART0、USART1の二つのUSARTポートがあります。この中にUSART0はHerkuleXとの通信に使用する4ピンのポート5個が並列に接続されており、USART1はイヤホンジャックを通じてPCと接続されており、内部的にZigbeeモジュールと接続されています。

ATmega128からのUSART信号は、デジタルHIGHとLOWの信号なので、信号レベルはLOWのとき0V、HIGHのとき、5Vです。このようなレベルの信号をTTL信号であるとします。しかし普通のPCのCOMポートまたはUSB-to-Serialジェンダーからの信号は、通常的にLOWのとき-5～-10V、HIGHのとき、+5～+10 Vの信号レベルを持ちます。このような信号は、RS-232レベルにあると言われます。ただし、同じデータが送信されている場合でも、これらの信号をそのまま接続するとお互いの信号レベルが合わなくて通信を行うことができません。

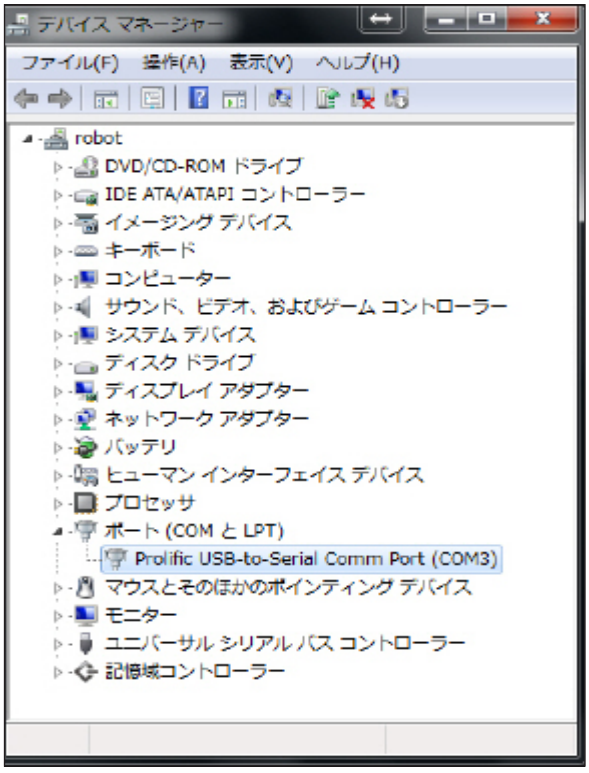
そのため、DRCの回路は、直列通信のレベル変換チップを使用して、ATmega128が送信される信号をTTLでRS-232に変えてPCに転送し、PCからの信号をRS-232でTTLに変えてATmega128に送信します。

ハードウェアの準備



01 接続

DRC制御機アダプターで電源を接続し、USB to Serial CableでPCとDRCを接続します。



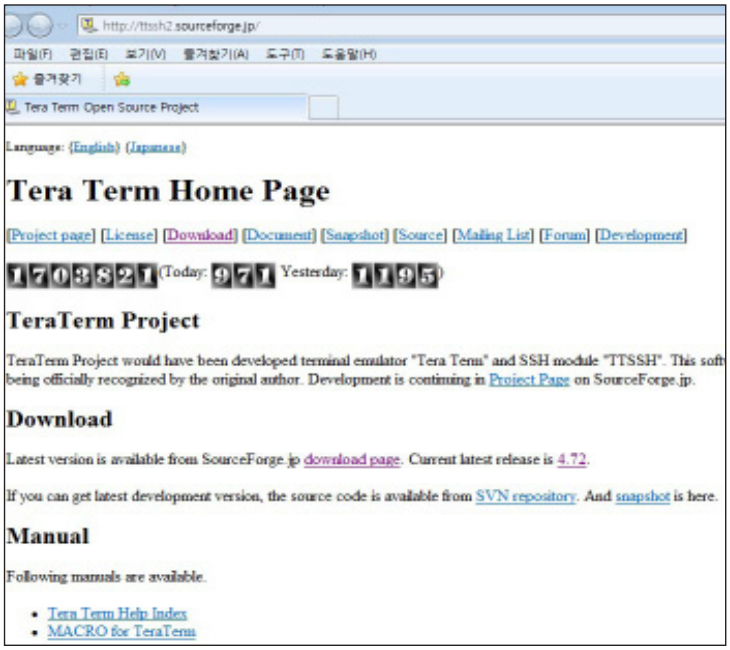
02 COM port 接続

マイコンピュータ>プロパティ>デバイスマネージャでCOM接続を確認します。



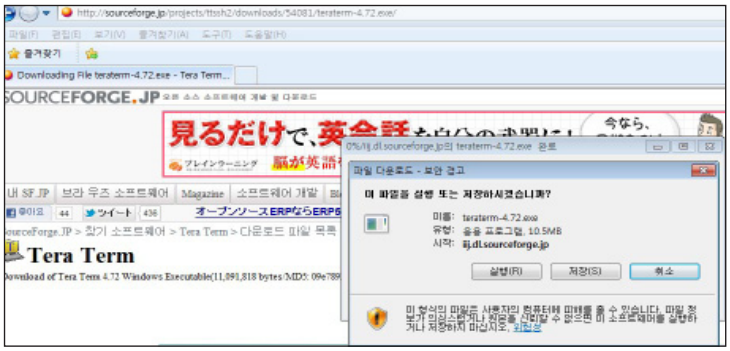
01 住所検索

インターネットポータルサイトで
“tera term”を検索します。



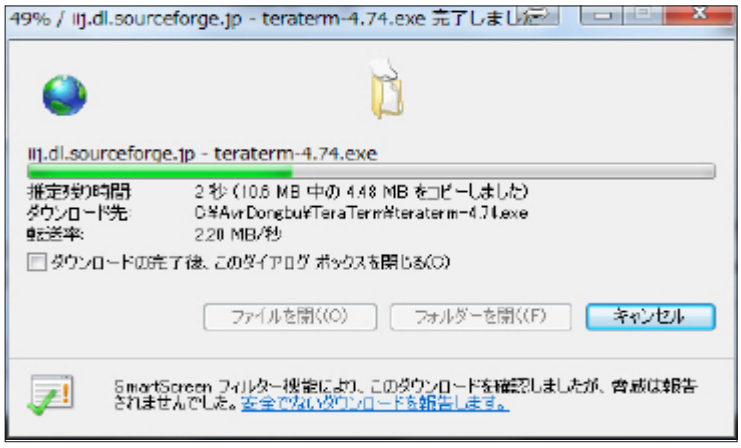
02 ホームページにアクセス

Tera Termのホームページにアクセスし、
“Download page”をクリックします。



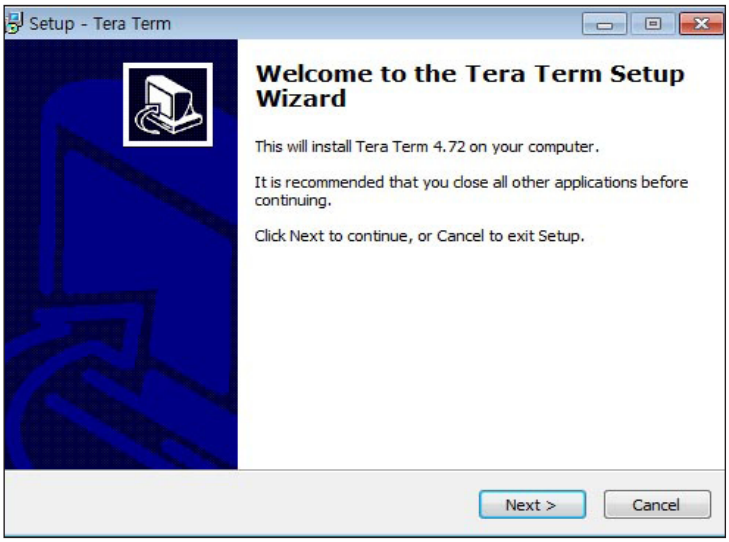
03 ファイルのダウンロード

“保存”をクリックしてファイルをダウンロードします。



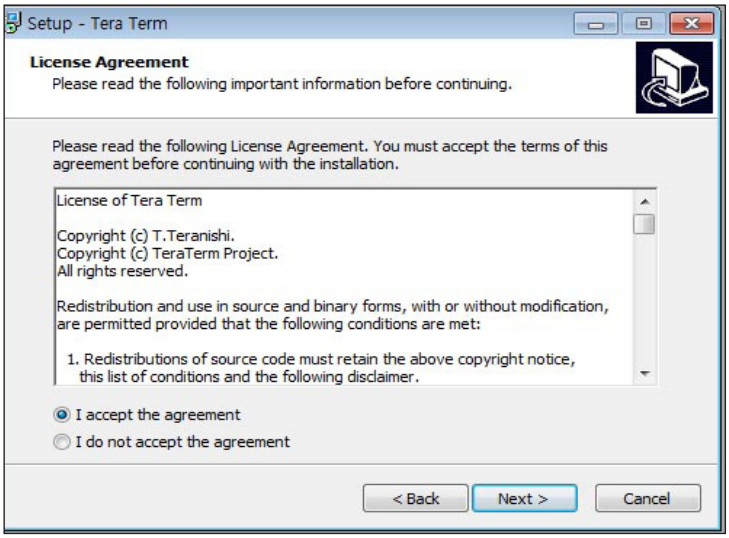
04 ファイルのダウンロード開始

インストールファイルをダウンロードします。



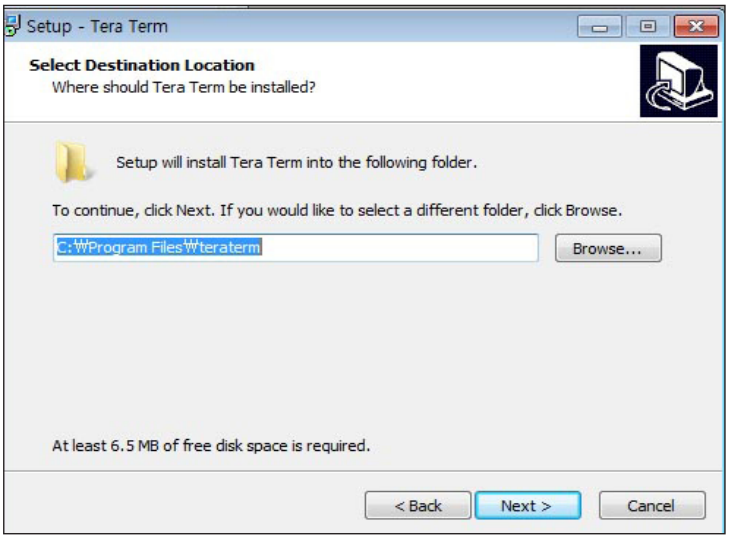
05 インストールウィザードの実行

“Next”をクリックするとインストールウィザードが実行します。



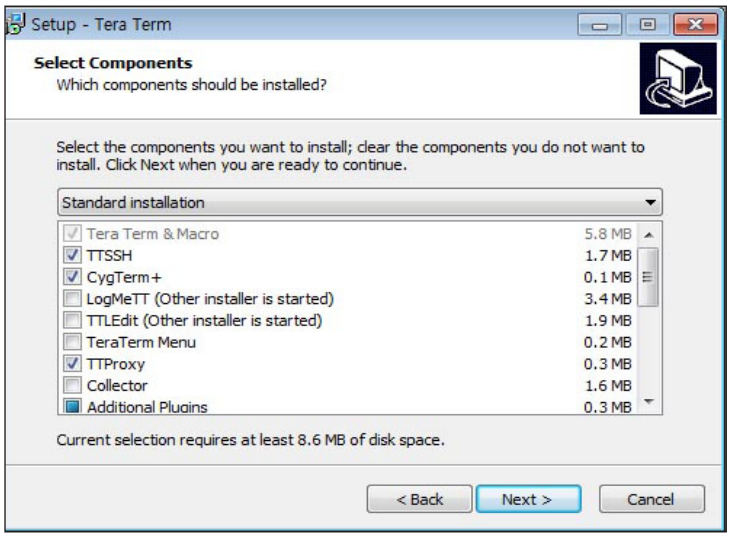
06 利用規約に同意

“I accept the agreement”をチェックし、“Next”をクリックします。



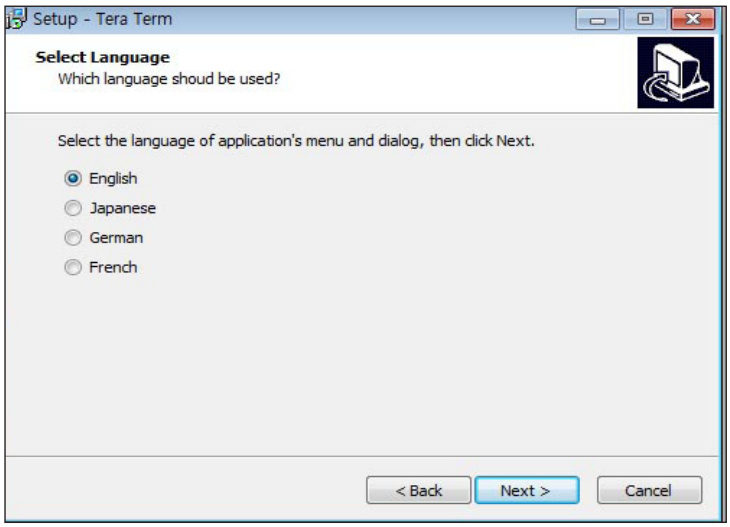
07 インストールフォルダを確認

プログラムをインストールするフォルダを確認し、“Next”をクリックします。



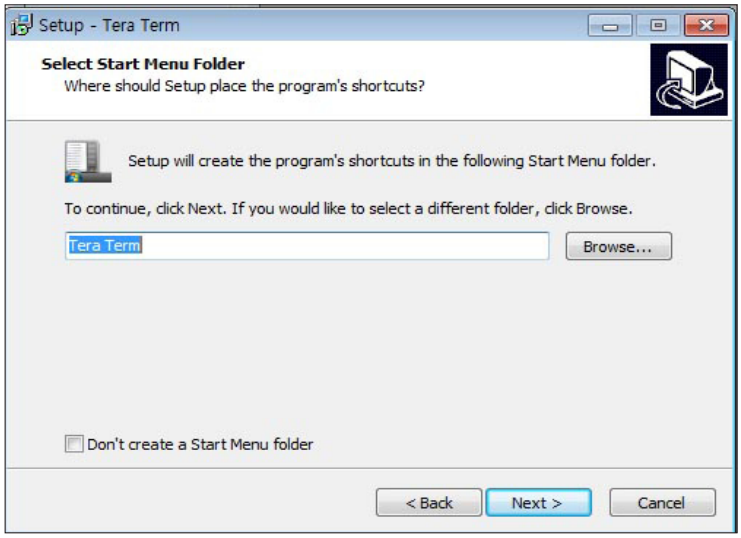
08 構成要素選択

“Next”をクリックします。



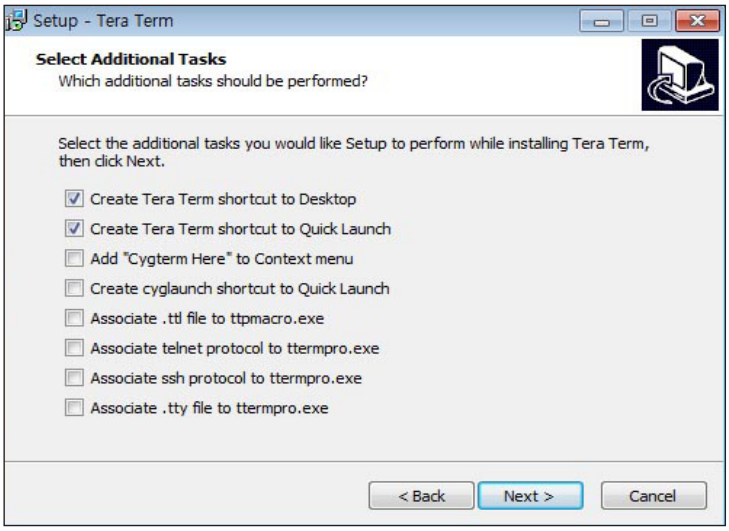
09 言語を選択

言語を選択し、
“Next”をクリックします。



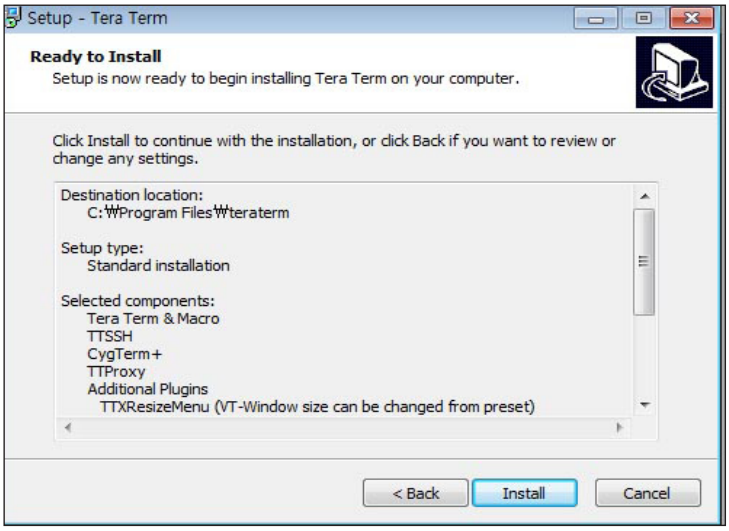
１０ 開始 メニューフォルダの選択

開始メニューフォルダを選択し、“Next”をクリックします。



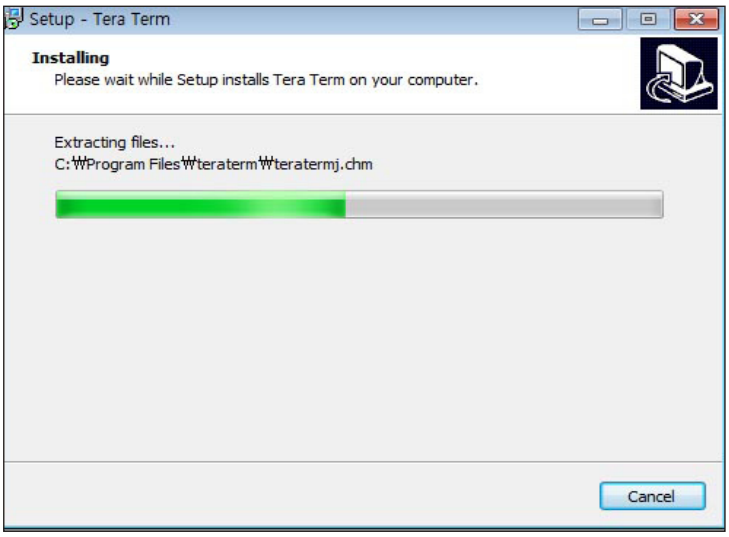
１１ 追加機能の選択

必要な追加機能を選択し、“Next”をクリックします。



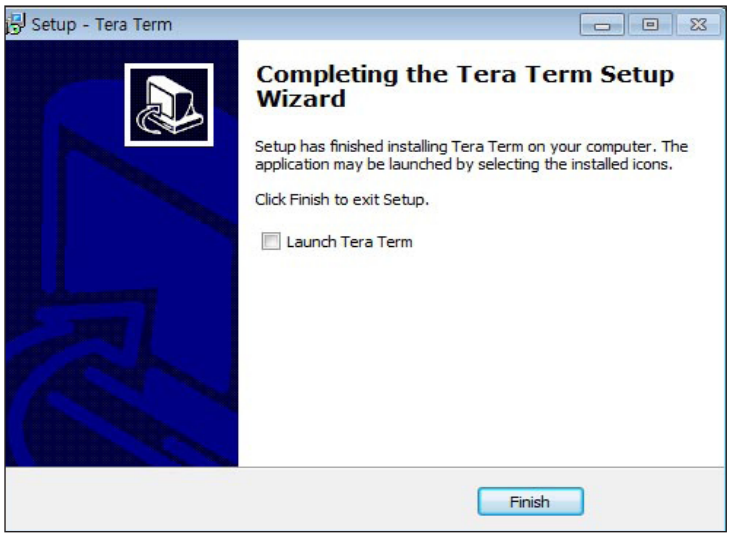
１２ インストールの開始

“Install”をクリックするとインストールが開始します。



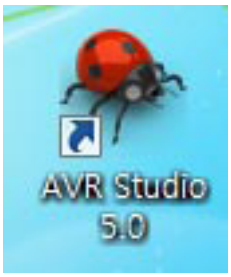
１３　インストールの確認

プログラムがインストールされます。



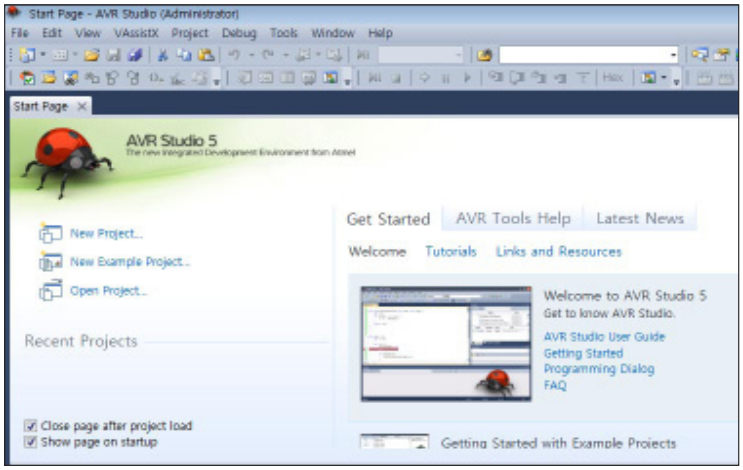
１４　インストールの完了

“Finish”をクリックすると、インストールが完了します。



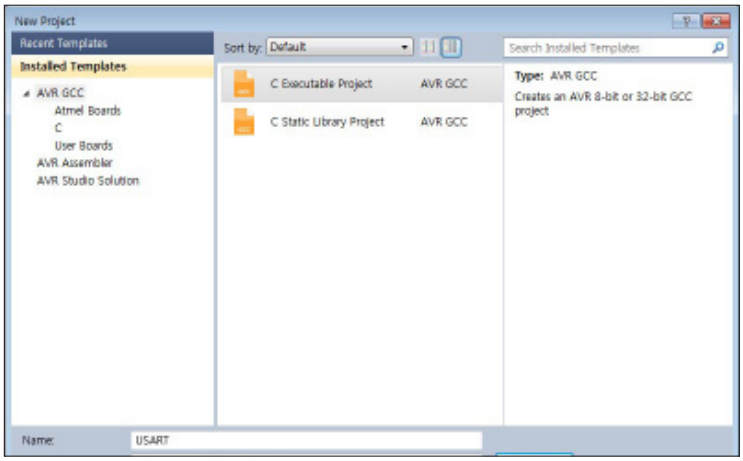
01 実行

AVR Stdio を実行します。



02 New Project

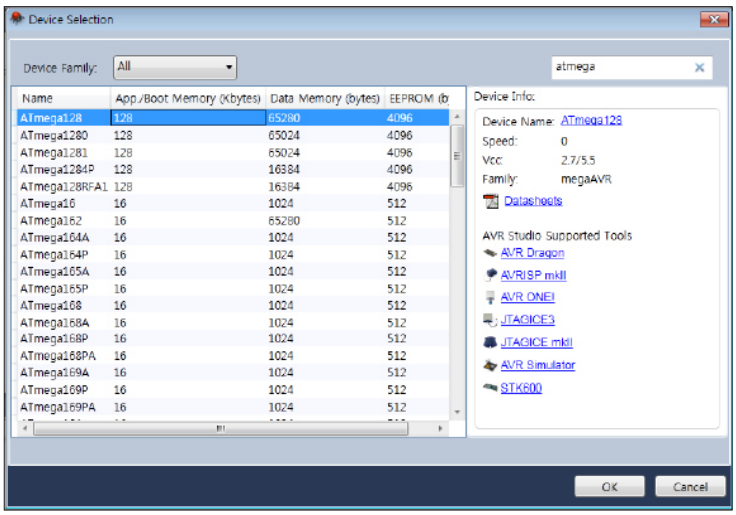
New Project をクリックします。



03 GCC 選択

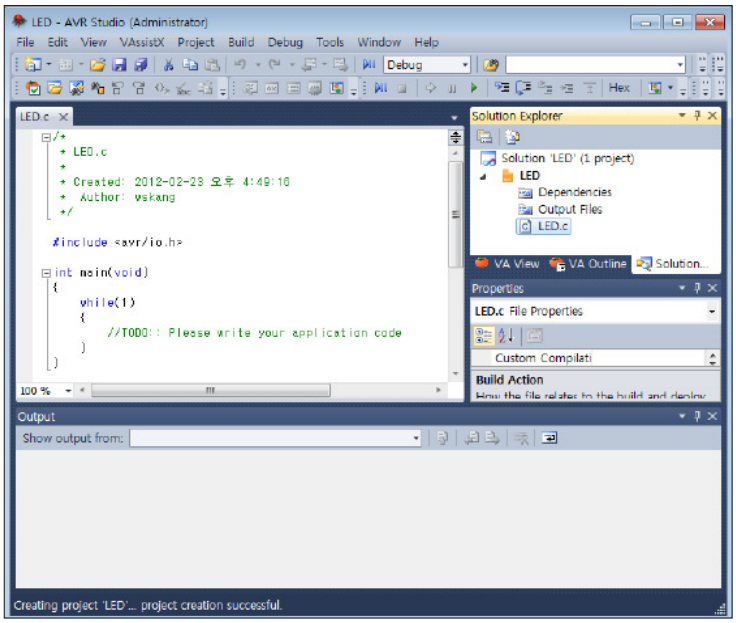
C Executable Project を選択します。

Nameを“USART”に入力して, Create directory for solution をクリックし、OK ボタンを押します。



04 ATmega 選択

右上からatmegaを検索し、リスト中のATmega128を選択します。



05 Editor

編集ウィンドウでのコーディングを開始します。

次の例を見てください。

例

USB–to–Serial　またはPCの背面のCOMPORTとDRCを通信ケーブルで接続し、ハイパーターミナルを利用して送信する文字に応じてDRCのLEDを制御する例です。

USART.c（ソースファイル名と場所）

```
//CPU クロックの設定する
#define F_CPU 16000000

//ATmega128のレジスタが定義されている
#include <avr/io.h>
//interruptを使用するために必要な情報が定義されている
#include <avr/interrupt.h>
//_delay_msなどディレイ が定義されている
#include <util/delay.h>

//LEDに するポートCのピン定義
#define LED_MASK 0b01111111

//USART1からデータを受信するときに使用全域 の宣言
volatile unsigned char gucRxBuffer;
volatile unsigned char gucRxFlag;

//USART1 受信完了インターラプト サービスルーチン
ISR(USART1_RX_vect){
        gucRxBuffer = UDR1;
        gucRxFlag = 1;
}

//USART1を初期化する
void USART1_Init(unsigned long ulBaudRate){
        unsigned long ulTemp;

        //ブートローダーとの衝突を防ぐために、しばらく待機
        _delay_ms(10);

        //送信、受信、受信インターラプト 活性化
        UCSR1B = ((1 << TXEN1) | (1 << RXEN1) | (1 << RXCIE1));
        //2倍速 送モード の設定
        UCSR1A = (1 << U2X1);

        //クロックとBaud Rateを使用してUBRR1に入る値の計算
        ulTemp = F_CPU / (ulBaudRate * 8L) - 1;
```

```
	//UBRR1に計算した値を設定する
	UBRR1H = (ulTemp >> 8) & 0x0F;
	UBRR1L = ulTemp & 0xFF;

	//全域　の初期化
	gucRxBuffer = 0;
	gucRxFlag = 0;

	return;
}

//USART1で文字を送信する
void USART1_PutChar(char cData){
	//送信データレジスタが空になるまで待機
	while(!(UCSR1A & (1 << UDRE1)));
	//送信データレジスタにデータを入力
	UDR1 = cData;

	return;
}

//USART1文字列を送信する
void USART1_PutString(char* pcStr){
	//'¥0'に う時までUSART1_PutCharを繰り返す
	while(*pcStr)
		USART1_PutChar(*pcStr++);
}

int main(void)
{
	//8ビット　の宣言
	unsigned char ucLED;

	///ポートCの入出力方向を設定する
	DDRC = 0b01111111;
	//ポートCの出力値を設定する
	PORTC = 0b01111111;

	//全 のインターラプト を無 にする
	cli();

	//USART1 の初期化
	USART1_Init(115200);
```

```
        //全 インターラプト を活性化
        sei();

        ///初期化を完了メッセージを送信
        USART1_PutString("Serial Initialized¥r¥n");

        //Servo LEDをオン
        ucLED = 0b00000001;
        PORTC = (~ucLED) & LED_MASK;

        //while の 容を無限に繰り返す
        while(1)
        {
                //データを受け取った場合は 行
                if(gucRxFlag){
                        //受信したデータが"z"や "Z"であれば 行
                        if(gucRxBuffer == 'z' || gucRxBuffer == 'Z'){
                                //オンにするLEDのビットを右に移動
                                if(ucLED > 0b00000001){
                                        ucLED >>= 1;
                                }
                                else{
                                        ucLED = 0b01000000;
                                }
                                PORTC = (~ucLED) & LED_MASK;

                                //メッセージを送信
                                USART1_PutString("LED Moved by ¥'z¥'¥r¥n");
                        }
                        //受信したデータが"x"や "X"なら 行
                        else if(gucRxBuffer == 'x' || gucRxBuffer == 'X'){
                                //オンにするLEDのビットを左に移動
                                if(ucLED < 0b01000000){
                                        ucLED <<= 1;
                                }
                                else{
                                        ucLED = 0b00000001;
                                }
                                PORTC = (~ucLED) & LED_MASK;

                                //メッセージを送信
                                USART1_PutString("LED Moved by ¥'x¥'¥r¥n");
                        }
                        //データ受信フラグの初期化
                        gucRxFlag = 0;
                }
        }

        /// の形に合わせて整 値を返す
        return 1;
}
```

例の説明

```
#define F_CPU 16000000
```

#define文はコードの前処理命令の一つで、今後F_CPUという値が出ると16000000に置き換えられることを意味しています。DRCのATmega128は16MHzのクロック速度で動作します。クロック速度に比例して命令の処理速度が異なりますので、この命令を通じて、このプログラムが動作する速度を指定する必要があります。そうすることでdelay関数が正常に動作します。

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

avr/io.hヘッダファイルにはDDRX、PORTXレジスタなどAVR制御のために必要なレジスタが定義されており、avr/ interrupt.hヘッダファイルには、インターラプト を使用するための関数が定義されています。util/ delay.hヘッダファイルには、_delay_ms()などのよく使用するディレイ関数が定義されているので # include文を使用しています。

```
#define LED_MASK                    0b01111111
```

#define LED_MASK 0b01111111は、今後LED_MASKが出ると0b01111111に置き換えてコンパイルするという意味です。DRCのLED全体に対応するビットをLED_MASKという名前に置き換えてプログラムを作成しやすくするために宣言します。

```
volatile unsigned char gucRxBuffer;
volatile unsigned char gucRxFlag;
```

インターラプト サービスルーチンと関数で共通に使用するグローバル変数を宣言します。gucRxBufferは受けたUSART1データレジスタの値を保存する変数であり、gucRxFlagは受け取ったデータがあることを示すフラグ変数です。

```
ISR(USART1_RX_vect){
        gucRxBuffer = UDR1;
        gucRxFlag = 1;
}
```

USART1 受信完了インターラプト のインターラプト サービスルーチンです。UDR1から受け取ったデータを読み込んでgucRxBufferに保存し、gucRxFlagを1にしています。このデータとフラグは、メイン関数で使用されます。

```
void USART1_Init(unsigned long ulBaudRate)
```

PCとの通信に使用するUSART1を初期化する関数です。ボーレートをulBaudRateというパラメータで受けて、ulBaudRateの値を使用してUBRR1に入る値を計算します。

```
unsigned long ulTemp;
```

ulTempという変数を宣言します。ulBaudRateで計算したUBRR値を保存する変数です。

```
_delay_ms(10);
```

DRCのブートローダーでもUSART1を使用してファームウェアを接続するため、ブートローダーでの通信中にデータをすべて送信できないにもかかわらず、ファームウェアでUSART1の設定を変更する場合、通信エラーが生じます。そのため、ブートローダーで送信されるデータを全部送信できるように10ms程度の時間を待ちます。

```
UCSR1B = ((1 << TXEN1) | (1 << RXEN1) | (1 << RXCIE1));
UCSR1A = (1 << U2X1);
```

UCSR1BとUCSR1A レジスタを設定します。送信と受信、受信インターラプト を活性化 します。また、2倍速非同期モードを使用するので、U2X1を1にしています。UCSR1Cレジスタは初期値のまま使用するため、別々に設定はしません。

```
ulTemp = F_CPU / (ulBaudRate * 8L) - 1;

UBRR1H = (ulTemp >> 8) & 0x0F;
UBRR1L = ulTemp & 0xFF;
```

ulBaudRateを使用してUBRR1値を計算してulTempに保存します。計算式は、上で出た$UBRR = \frac{f_{OSC}}{8BAUD} - 1$を使用しています。 ulTemp値の上位4バイトと下位8バイトを、それぞれUBRR1HとUBRR1Lに保存します。これでUSART1は入力ulBaudRateのボーレートで動作します。

```
gucRxBuffer = 0;
gucRxFlag = 0;
```

全域変数を0に初期化します。

```
void USART1_PutChar(char cData){
        while(!(UCSR1A & (1 << UDRE1)));
        UDR1 = cData;
return;
}
```

USART1_PutCharはUSART1を使用して、1バイトのデータを送信する関数です。データ送信は、UCSR1AのUDRE1ビットが1でなければ出来るので、他のデータを既に送っているときは、UDRE1ビットが1になるまで待機する必要があります。while(!(UCSR1A&(1<< UDRE1)));でUDRE1が1になるまで待って、その後はパラメータで受け取ったcDataをUDR1に入れて新しくcDataを送信します。

```
void USART1_PutString(char* pcStr){
        while(*pcStr)
                USART1_PutChar(*pcStr++);
}
```

USART1_PutStringはUSART1を使用して文字列を送信する関数です。文字列の終わりは'₩0'(NULL文字)で終わって、その値は0なので、while(* pcStr)文ではUSART1_PutCharを利用してpcStrが指すデータを送り、pcStrの値を1ずつ増加させ、* pcStrがNULL文字になるまで繰り返すことになります。

```
unsigned char ucLED;

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

LEDの出力値が保存される8ビットの変数ucLEDを宣言して、ポートCの0〜6番ピンを出力に設定し、出力値を1にします。これにより、すべてのLEDが消灯します。

```
cli();
USART1_Init(115200);
sei();
```

インターラプトの設定を変更する中でインターラプトが発生して予期せぬ誤動作が発生することを防止するために、まずcli()命令を通じて全体のインターラプトを無効にします。その後USART1_Init関数を使用してUSART1を初期化します。使用するボーレートが115200であるため、パラメータとしてその値を使用しています。USART1の初期化が終わった後にsei()命令を通じて全体のインターラプト を活性化 します。

```
USART1_PutString("Serial Initialized₩r₩n");

ucLED = 0b00000001;
PORTC = (~ucLED) & LED_MASK;
```

USART1_PutStringを使用して、シリアル通信が初期化されたことを通知します。文字列の終わりの¥ r¥ nは、行を変えるための文字です。¥ rはCarriage Returnとしてカーソルを行の先頭に変える役目をして、¥ nはNewlineとして行を変更する場合に使用します。
ucLEDを0番ビットのみが1で作ってServoLEDのみオンに設定します。その後PORTCの値を変えて実際にucLEDに設定されたLEDを点灯します。実際のLEDがactivelowの方法で行動しているビットが0になるとLEDが点灯するので、ucLEDを～（bitwise NOT）演算してビットを反転した結果にLED_MASKを＆（bitwise AND）演算した値をPORTCに入力します。

```
if(gucRxFlag){
…
        gucRxFlag = 0;
}
```

while文の中には、if文が一つあります。このif文はgucRxFlagが0以外のときに実行され、if文が終わる場合は、再度gucRxFlagを0に初期化します。普段はgucRxFlagが0であるため、while文は何もせずに繰り返されます。そのうちに、USART1のデータが受信されると、USART1受信完了インターラプトのサービスルーチンでは、データをgucRxBufferに読んでgucRxFlagを1に設定します。インターラプトが終わって、またif文に出会ったときは、gucRxFlagが1であるため、プログラムは、if文の中に進入してルーチンを実行します。そして、その後gucRxFlagを0に初期化します。このようにif文を作成すれば、if文の内容をUSART1受信完了インターラプト が起こるたびに一回ずつ実行することができます。

```
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");
}
```

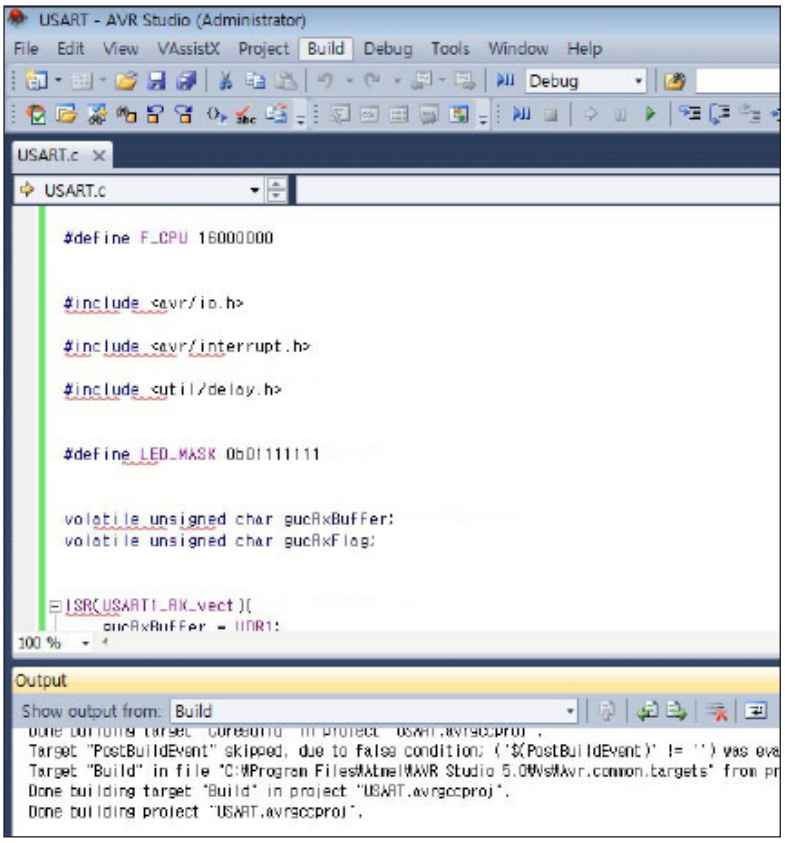
gucRxFlagが1なのでif文に入った後、上記のコードに出会います。gucRxBufferに保存されただけで入ってきたデータの値が' z' または' Z' の場合には、ucLEDに保存されたビットを右に一度移動します。もしucLED値が0b00000001となり、さらに移動するスペースがない場合は、一番左側の0b01000000に変えます。その後PORTCのucLEDに対応する値を入れて、実際にLEDをオンにします。LEDの電源を入れた後USART1_PutString関数を使用して、"z"キーでLEDが動いたことを出力します。

```
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");
}
```

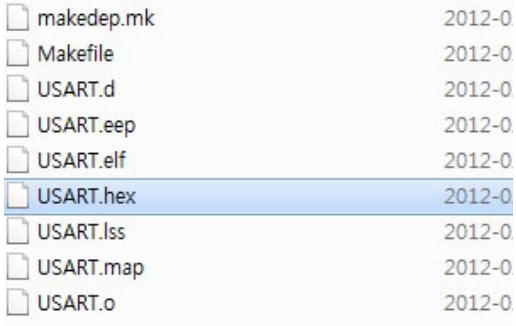
もし入力されたデータが"z"や"Z"でない場合は、上記のコードに出会います。データの値が' x' または' X' である場合には、"z"とは逆にucLEDのビットを左に移動します。もしucLED値が0b01000000となり移動するスペースがない場合は、一番右側である0b00000001に置き換えます。その後PORTCの値を変えてLEDをオンにし、"x"キーでLEDが動いたことを出力します。If–else文は、ここで終わるので、もし"z"、"Z"、 "x"、 "X"以外の入力が入って来たら何も行わず、gucRxFlagだけ0に初期化します。

実行

ファームウェアを実行して、ハイパーターミナルを利用してz、xキーを入力すると、点灯LEDが移動しながら、ターミナルウィンドウにメッセージが出力されることを確認します。







01 Build

Buildメニューを選択しBuild SolutionをクリックしてBuildします。

1 succeeded0 failedが出るとエラーなしでコンパイルが完了したということです。

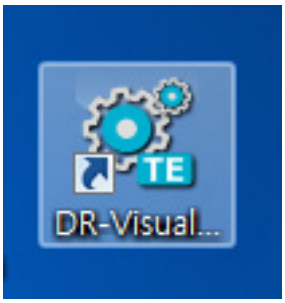
02 Debug フォルダ

Build後にソースがあるフォルダに入るとDebugフォルダが作成されています。クリックしてフォルダの中身を表示します。

03 hex 確認

Debugフォルダの中のhexファイルを確認します。Hexファイルは制御機に移植するファイルです。

DR-Visual Logicで、ファームウェアのリカバリーを行います。



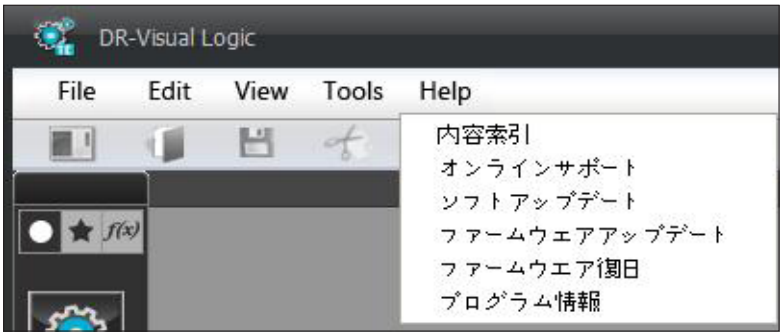
04 Visual Logic

DR–Visual Logicを実行します。



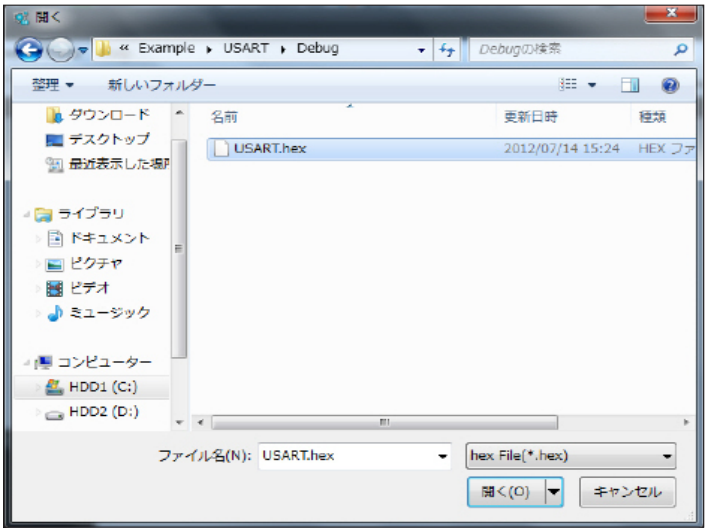
05 接続する

COM 番号を設定して接続するをクリックします。



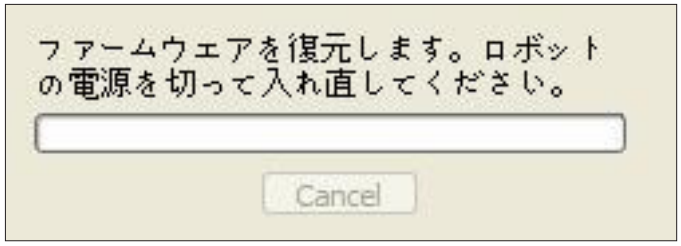
06 ファームウェアの修復

[ヘルプ]メニューからファームウェア修復をクリックします。



07 ロード

Debug フォルダにHexファイルをロードします。



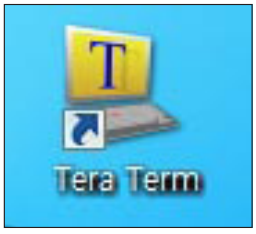
08 ファームウェアの修復

“ロボットの電源を切って入れ直してください”というメッセージが出ます。コントローラの電源ボタンを一度切ってから再度入れ直してください。



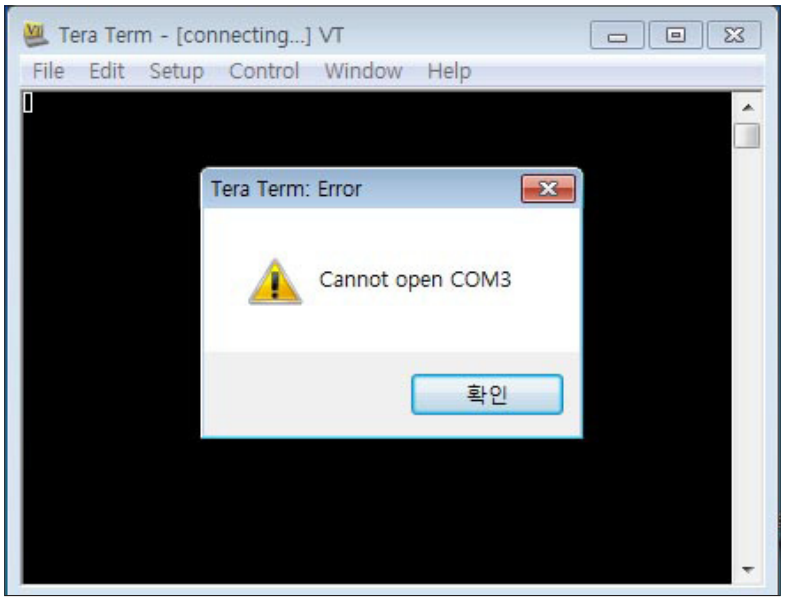
09 修復の開始

ファームウェアの復元が開始されます。プロセスが完了したら、ポップアップウィンドウが消えます。
Tera Term を利用して通信テストを行います。



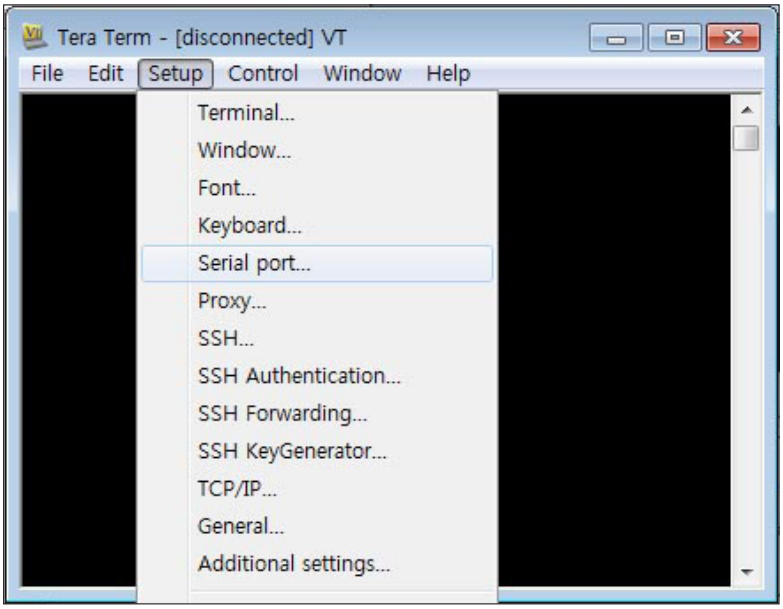
10 実行

Tera Term をクリックしてプログラムを実行します。



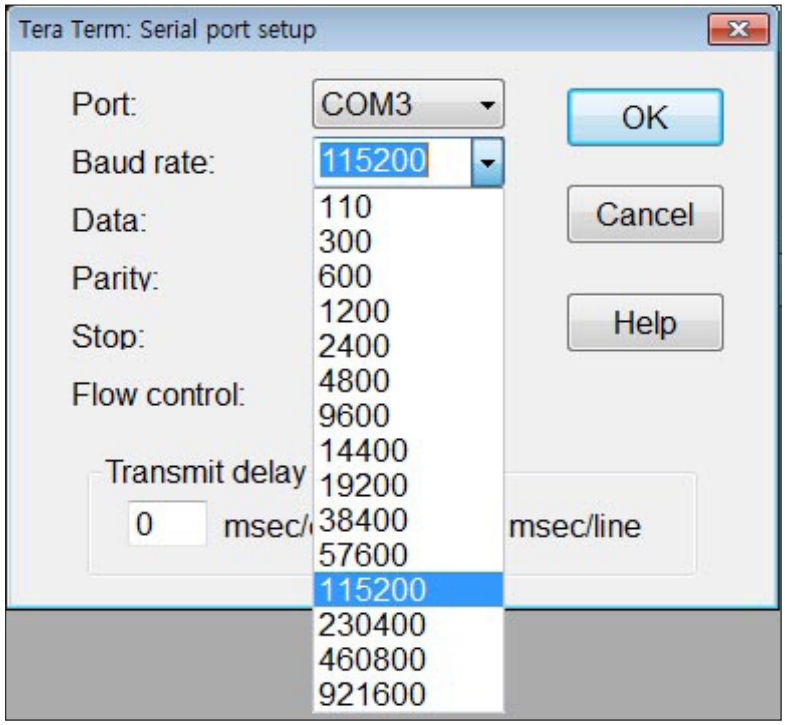
11 開く

実行すると、すぐには接続されていません。[OK]ボタンをクリックします。



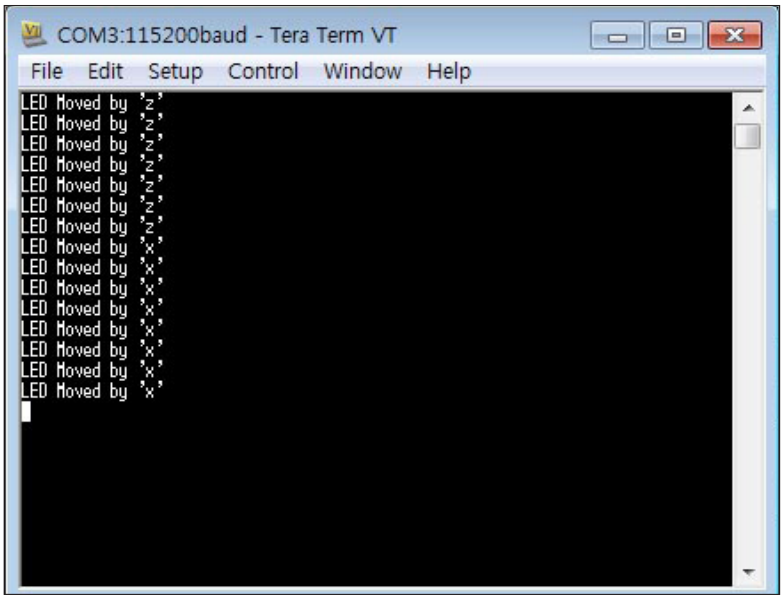
12 Set up

メニューの中のSetupメニューをクリックしてSerial portを選択します。



13 Baud rate

Baud rate を115200に選択します。
コントローラの最大通信速度です。



LEDが順番に移動する



14 実行

ファームウェアを実行して、Tera Term(ハイパーターミナル)を利用してz、xキーを入力すると、点灯LEDが移動しながら、ターミナルウィンドウにメッセージが出力されることを確認できます。

HerkuleX 制御する

herkulexライブラリを利用してHerkuleXに直接パケットを送信し制御します。

H/W原理

DRCに装着されたATmega128には、USART0、USART1の二つのUSARTポートがあり、このうちUSART0はHerkuleXとの通信に使用する4ピンのポート5個が並列に接続されています。制御したいHerkuleXをケーブルでDRCと接続すると、USART0を通って直接パケットを送受信することができます。

今回のChapterでHerkuleXと通信をするためには、HerkuleXのレジスタマッププロトコルの知識が必要です。今回の例で使用するパケットに使用された命令をまとめると以下のとおりです。

名	CMD	説明
RAM_WRITE	0x03	HerkuleXの揮発性レジスタ領域のデータを使用しています。
RAM_READ	0x04	HerkuleXの揮発性レジスタ領域のデータを読み取ります。
I_JOG	0x05	HerkuleXの位置/速度を制御して動かします。

今回の例で使用する揮発性レジスタ領域の項目をまとめると以下のとおりです。

名	アドレス	サイズ	説明
Torque Control	52	1	トルクの状態を制御します。 0x00 : 自由に動く力がない状態 0x40 : 動く力はないが、現在の位置を維持する状態 0x60 : トルクが印加され、自分で駆動する状態
Calibrated Position	58	2	現在の位置とGPIOの入力結果を表します。 ここでGPIOは、HerkuleXの一部のモデルにおいて 、右の通信ポートをスイッチの入力ポートとして使用する時の入力値です。 15番のビット : 使用しない 14:13番のビット : GPIO入力値を示しています。 12:0番のビット : 現在HerkuleXの位置を示します。

プロトコルとレジスタマップの詳細については、HerkuleXマニュアルを参照してください。

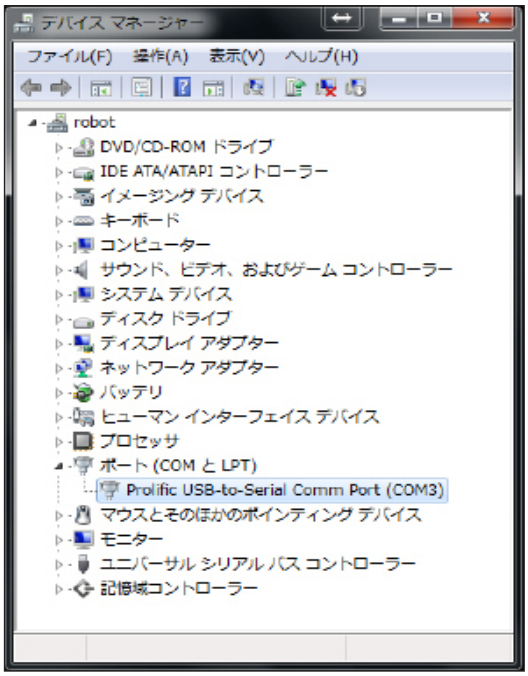
ハードウェアの準備



01 モータの準備

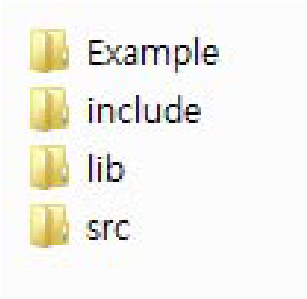
DRC制御機アダプターで電源を接続し、USB to Serial CableでPCとDRCを接続します。

サーボモータをコントローラの右側のポートに接続します。



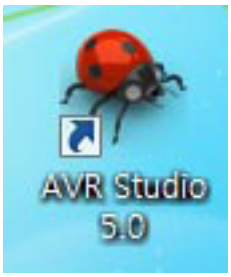
02 COM port 接続

マイコンピュータ>プロパティ>デバイスマネージャでCOM接続を確認します。



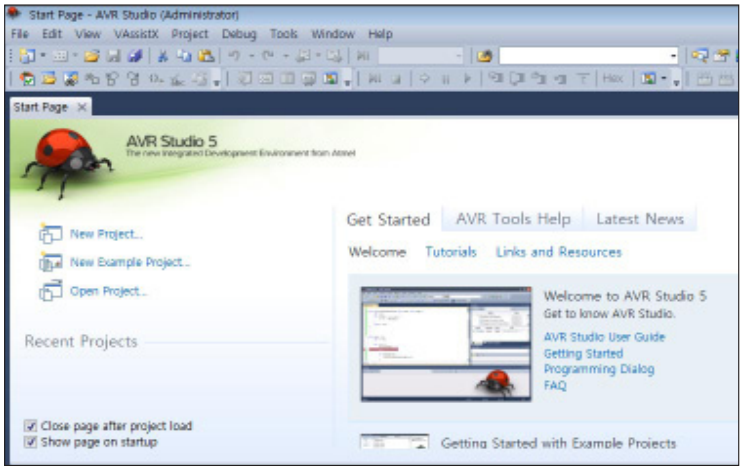
00 ライブラリ

サーボモータを制御するためには、基本的なライブラリファイルが必要です。ライブラリが含まれているフォルダ構造です



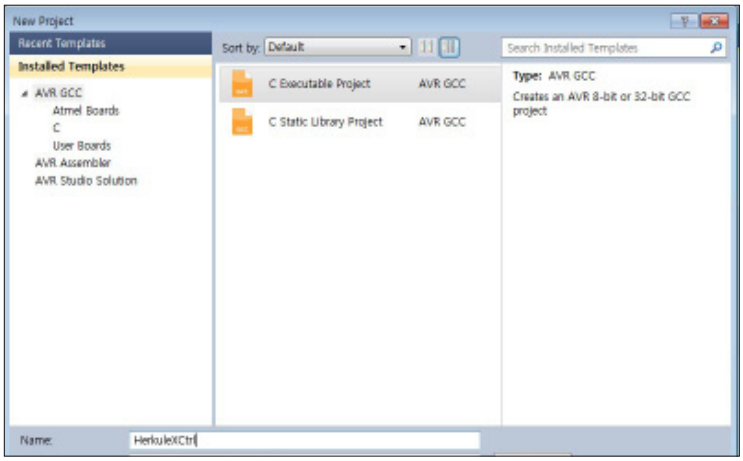
01 実行

AVR Studio を実行します。



02 New Project

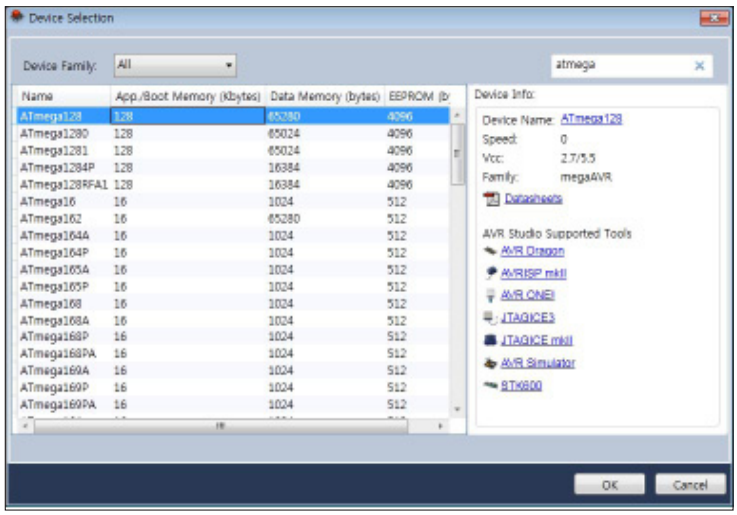
New Projectをクリックします。



03 GCC 選択

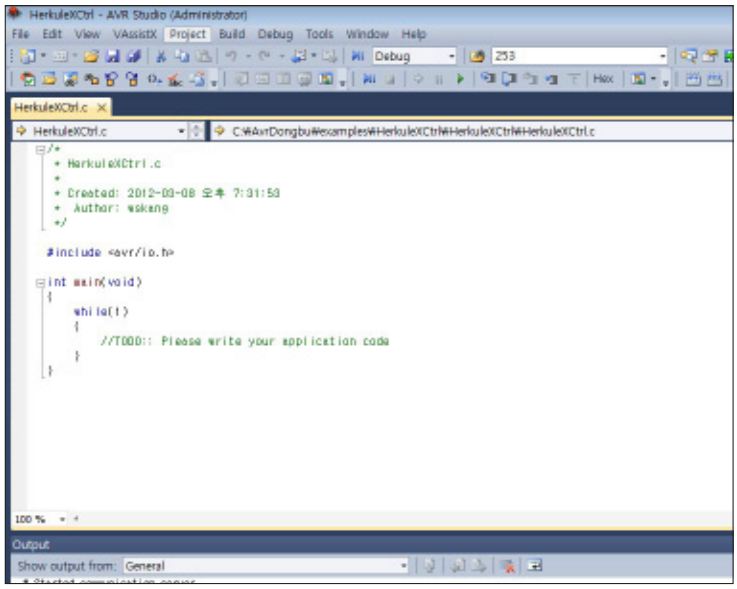
C Executable Project を選択します。

Name を“USART”に入力して，Create directory for solution をクリックし、OK ボタンを押します。



04 ATmega 選択

右上からatmegaを検索し、リスト中のATmega128を選択します。

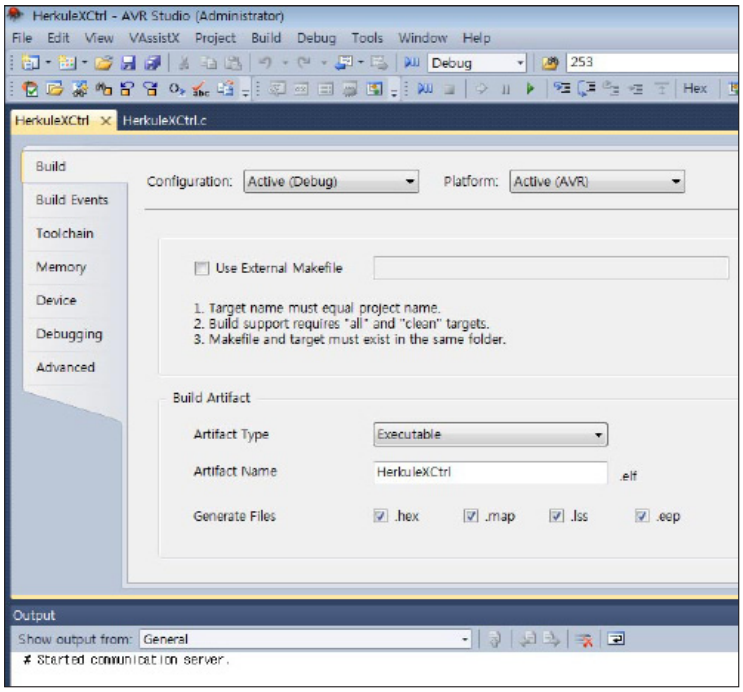


05 Editor

編集ウィンドウでコーディングを開始します。

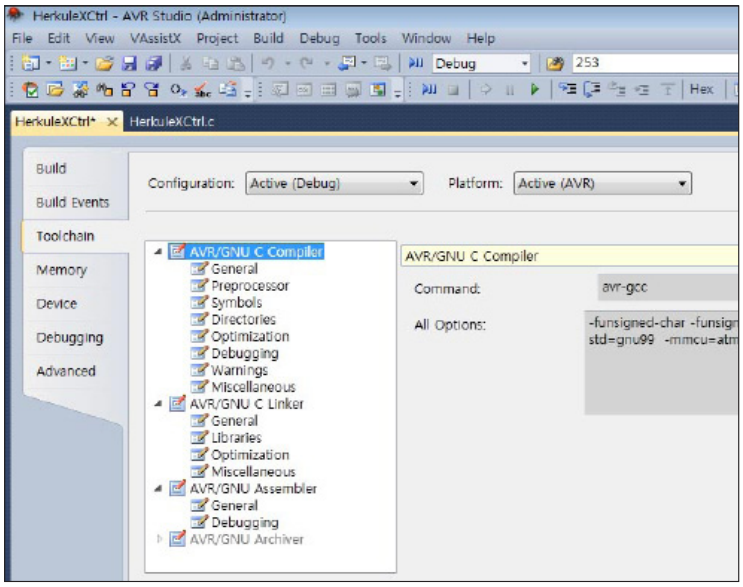
次の例を見てください。

ライブラリのリンクと設定



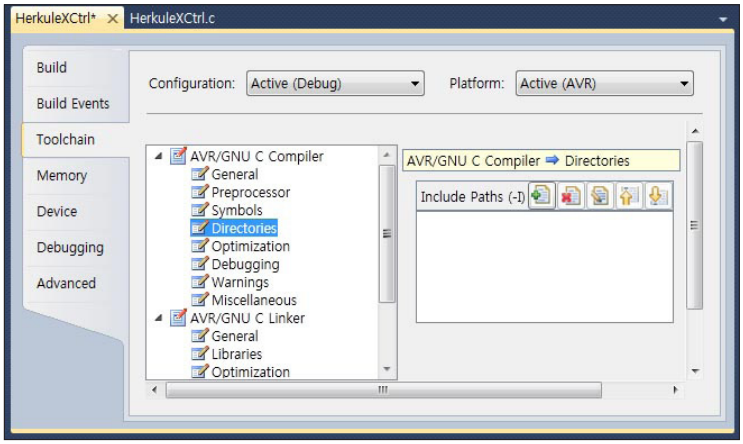
01 実行

AVR Studio 上部メニューの中にProjectをクリックすると、HerkuleXCtrl(プロジェクト名によって異なります)Propertyメニューが作成されています。クリックすると、左のように設定画面が現れます。



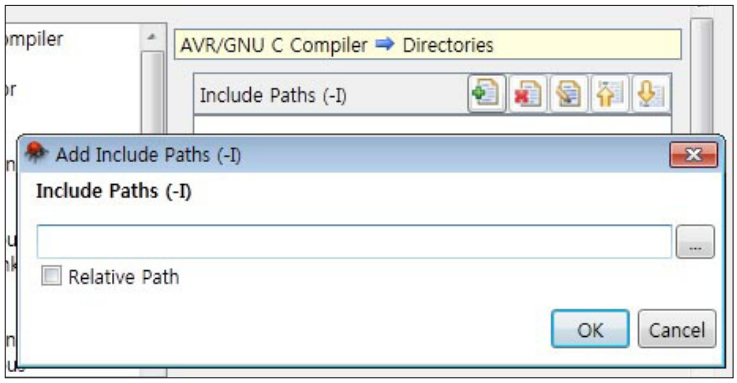
02 Toolchain

左のテブボタン中のToolchainをクリックして、AVR/ GNU C Compilerを選択します。



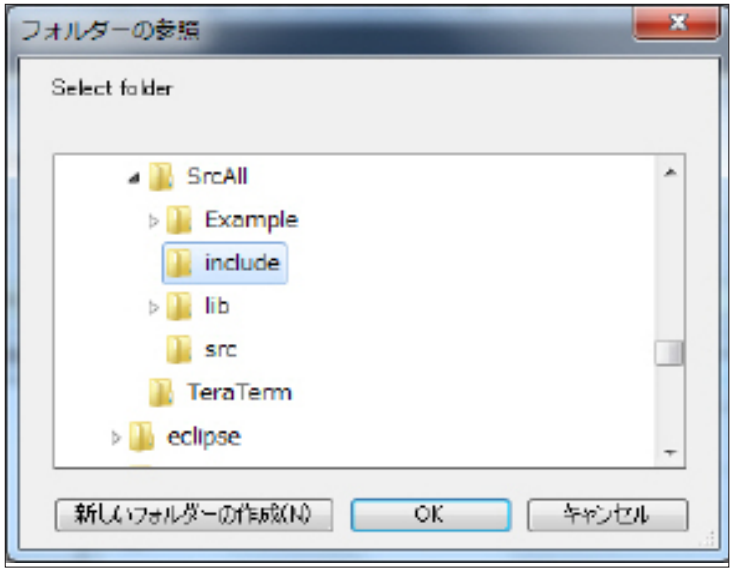
03 C Compiler

C Compilerフォルダの中のDirectoriesを選択します。



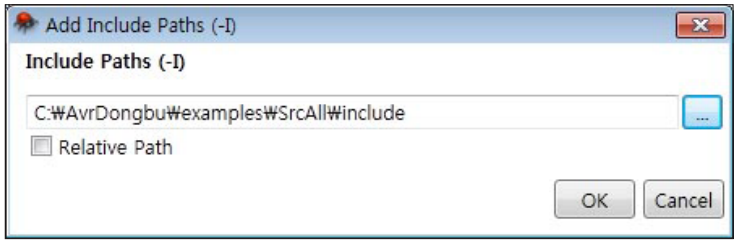
04 Path

Include Pathsの右の+ボタンをクリックするとポップアップ入力画面が現れます。Relative Path（相対パス）のチェックを解除して、右側のフォルダの検索ボタンをクリックします。



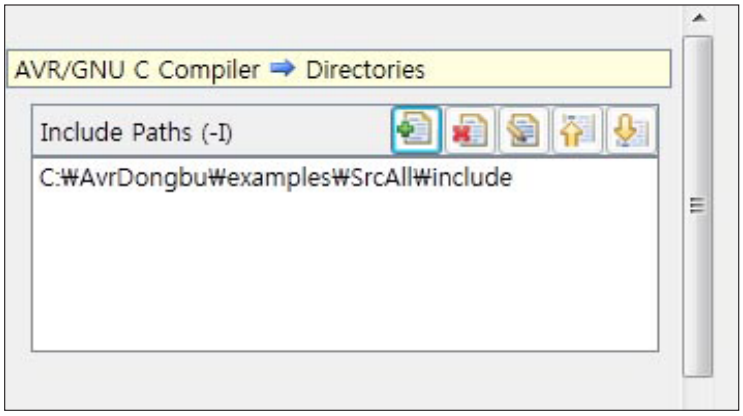
05 Include

提供ライブラリフォルダの中のincludeを選択します。



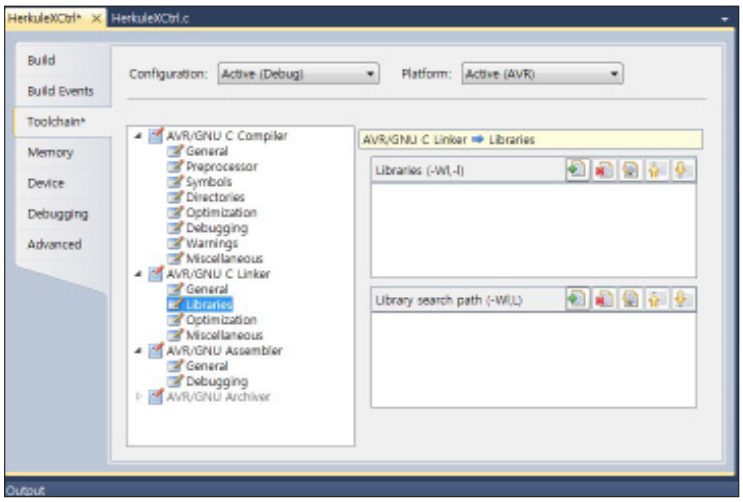
06 Include 完了

Includeフォルダの場所を指定します。OKボタンをクリックします。



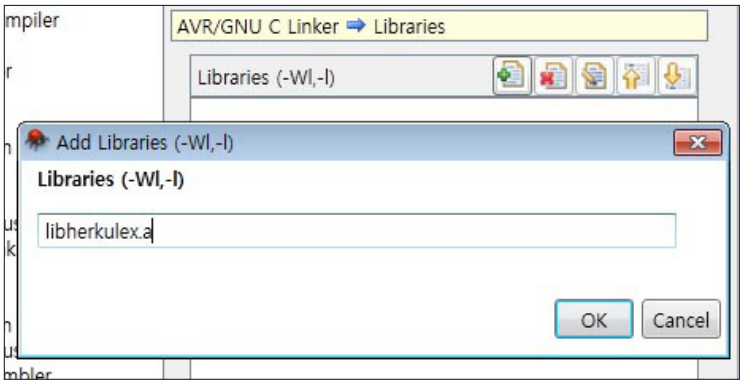
07 Path 完了

Includeパスの指定が完了しました。



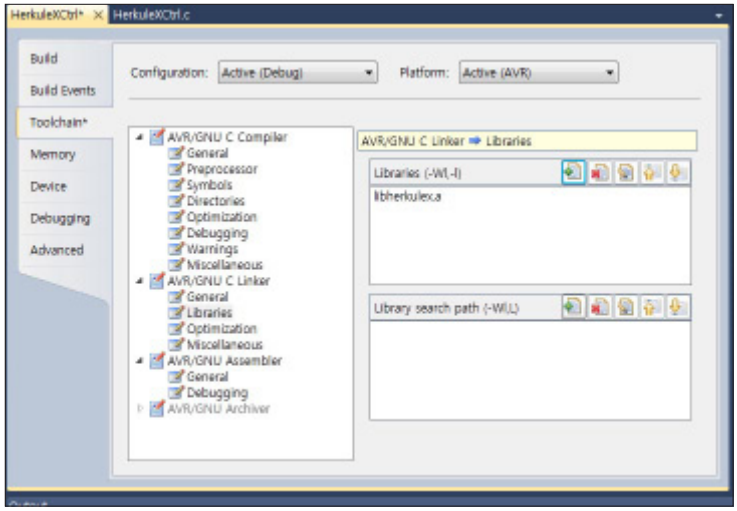
08 C Linker

C Linkerフォルダの中のLibrariesを選択します。



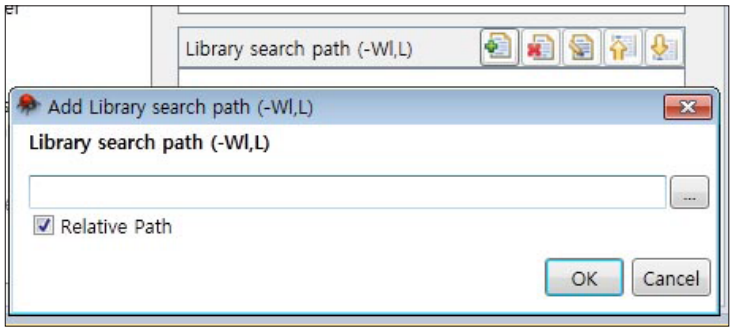
09 Libraries

Librariesウィンドウの右側+をクリックすると、Librariesファイル名入力画面が出ます。 libherkulex.aを直接入力します。
OKボタンをクリックします。



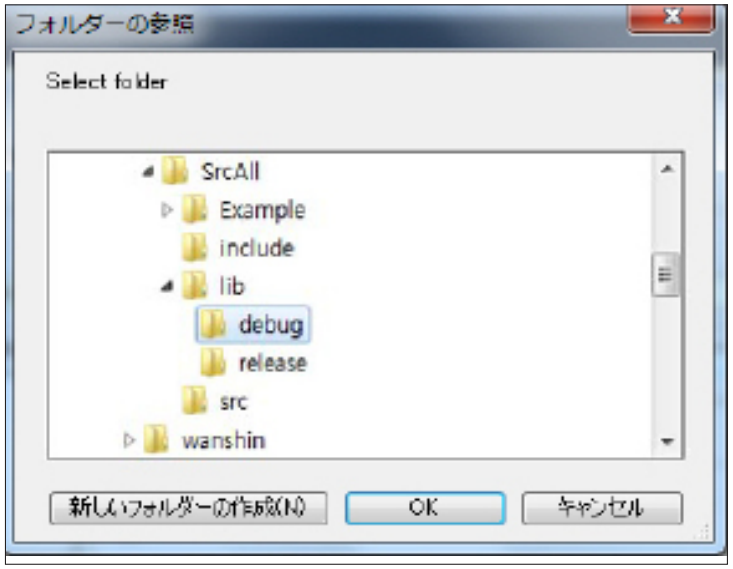
10 Search

サーボモータのライブラリファイル libherkulx.aリンクが完了しました。
次はLibrary search pathを設定します。
右の+ボタンをクリックします。



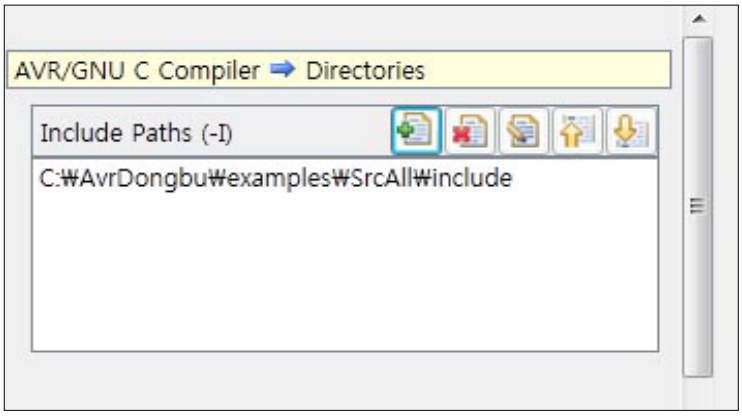
11 Path

Libray search pathのポップアップの右側のフォルダの検索ボタンをクリックします。



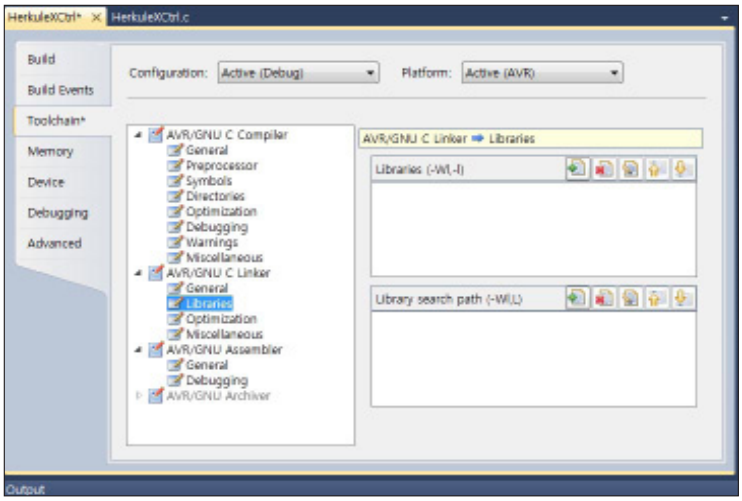
12 Debug

提供ライブラリフォルダのlibサブフォルダでdebugを選択します。



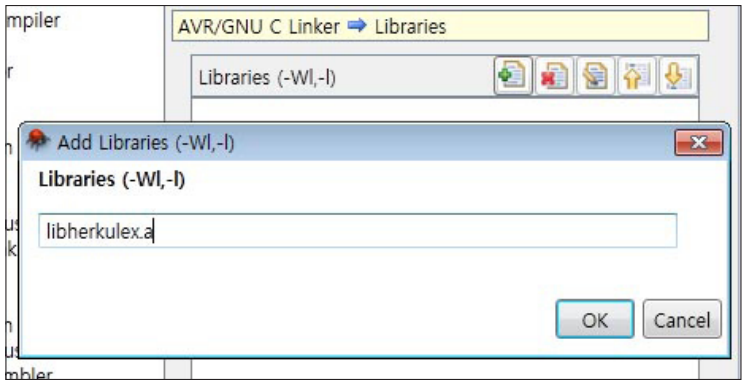
13 Path 完了

Relative Path(相対パス指定)チェックボックスを解除します。
Library search pathのパスの指定が完了しました。
OKボタンをクリックします。



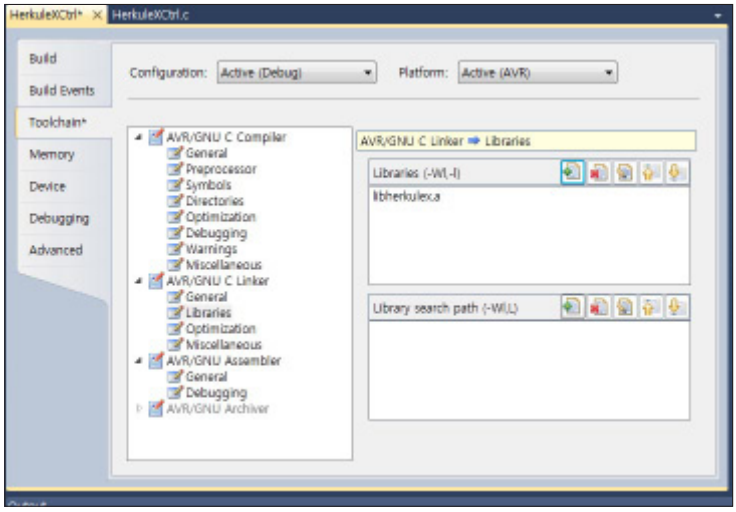
14 Link 完了

C Linkerライブラリファイルとパスの指定が完了しました。



15 保存

[保存]ボタンを押すと、現在の設定値が保存されます。タップ名がHerkuleXCtrl*)HerkuleXCtrlIに変更されます。



16 コーディング

HerkuleX.cタップを選択し、以下の示した例を入力します。

Servo IDは、基本的な253番です。サーボモータだけ単品で購買して時は基本的なIDは253番ですが、ヒューマノイドセットで購入の際にはサーボによってID値が表記されています。そのサーボモータを使用時にはそのIDを入力します。

例

IDが253のDRS-0101サーボモータを繰り返し動かしながら、位置の値を読み取って目的の場所に実際に到達しているかを確認する例です。

TIMER0.h

```
#ifndef TIMER0_H_
#define TIMER0_H_

#ifdef __TIMER0_C
	#define TIMER0_EXT
#else
	#define TIMER0_EXT extern
#endif

//タイマーを使って時間を測定する時の全域
TIMER0_EXT volatile unsigned char gucTimerTick;
//タイマ/カウンタ0の初期化
TIMER0_EXT void TIMER0_Init(void);

#endif /* TIMER0_H_ */
```

Herkulex.h

```
#ifndef HERKULEX_H_
#define HERKULEX_H_

////////////////////////////////プロトコルに する宣言////////////////////////////////
//各項目のインデックス
#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

//ヘッダ 連
#define HEADER				0xFF

//SIZE 連
#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 連
#define MAX_ID                          0xFD
#define BROADCAST_ID                    0xFE

//CMD 連 - 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 連 - 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 連
#define CHKSUM_MASK                     0xFE
```

```
//////////////////////////////プロトコルの構造 ////////////////////////////////
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 DrslJog
{
        DrsJog          stJog;
        DrsSet          stSet;
        unsigned char   ucId;
        unsigned char   ucPlayTime;
}DrslJog;

typedef struct DrsSJog
{
        DrsJog          stJog;
        DrsSet          stSet;
        unsigned char   ucId;
}DrsSJog;

typedef struct DrslJogData
{
        DrslJog         stlJog[CMD_I_JOG_MAX_DRS];
}DrslJogData;

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;

//////////////////////////////受信 態の結果の値//////////////////////////////
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

//HerkuleXを制御するために初期化する
HERKULEX_EXT void hklx_Init(unsigned long ulBaudRate);
//HerkuleXにパケットを送信する
HERKULEX_EXT void hklx_SendPacket(DrsPacket stPacket);
//HerkuleXからのパケットを受け取る
HERKULEX_EXT unsigned char hklx_ucReceivePacket(DrsPacket *pstPacket);

#endif /* HERKULEX_H_ */
```

HerkuleXCtrl.c

```
//CPU クロックを設定する
#define F_CPU 16000000

//ATmega128のレジスタが定義されている
#include <avr/io.h>
//interruptを使用するために必要な情報が定義されている
#include <avr/interrupt.h>
//_delay_ms などディレイ が定義されている
#include <util/delay.h>

//herkulex ライブラリのヘッダファイルを含む
#include <herkulex.h>
#include <TIMER0.h>

//DRS-0101의 Calibrated Position 情報を使用するための構造
typedef struct DrsCaliPos
{
	int		iPosition : 13;
	unsigned int	uiGPIO1 : 1;
	unsigned int	uiGPIO2 : 1;
	unsigned int	reserved : 1;
}DrsCaliPos;

//Calibrated Position 情報を容易にするための共用
typedef union DrsUnionCaliPos
{
	DrsCaliPos	stCaliPos;
	unsigned int	uiCaliPos;
}DrsUnionCaliPos;

int main(void)
{
	//送信するパケットと受け取るパケットが保存されているパケットの構造 の宣言
	DrsPacket stSendPacket, stRcvPacket;
	//Calibrated Positionを保存する共用　の宣言
	DrsUnionCaliPos unCaliPos;
	//使用する　の宣言
	unsigned char ucResult;
	int iPos = 400;

	//ポートCの入出力方向を設定する
	DDRC = 0b01111111;
	//ポートCの出力値を設定する
	PORTC = 0b01111111;
```

```
//全 インター ラプト を無 にする
        cli();

        //HerkuleXを使用するために初期化
        hklx_Init(115200);

        //全 のインター ラプト を活性化
        sei();

        //Torque Controlに0x60を使ってトルクをかけるパケット の構成
        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;

        //パケット を送信する
        hklx_SendPacket(stSendPacket);

        while(1)
        {
                //目標の場所の  の値を 更する
                if(iPos == 400){
                        iPos = 624;
                }
                else{
                        iPos = 400;
                }

                //DRS-0101を動かすI_JOGパケット を構成する
                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;

               //パケット を送信する
                hklx_SendPacket(stSendPacket);

                //すべて動いたときまで1秒待機
                _delay_ms(1000);
```

```
        //Calibrated Positionを みむRAM_READパケットの構成
                stSendPacket.ucPacketSize = MIN_PACKET_SIZE + 2;
                stSendPacket.ucChipID = 253;
                stSendPacket.ucCmd = CMD_RAM_READ;
                stSendPacket.unData.stRWData.ucAddress = 58;
                stSendPacket.unData.stRWData.ucLen = 2;

                //パケットを送信する
                hklx_SendPacket(stSendPacket);

                //TIMEOUT まで待つ時間を設定する。 2*15 = 30(ms)
                gucTimerTick=15;
                while(1){
                        //パケットを受け取る  を呼び出して結果をucResultに保存
                        ucResult = hklx_ucReceivePacket(&stRcvPacket);
                        //結果がDRS_RXWAITINGでなければ出る
                        if(ucResult != DRS_RXWAITING){
                                break;
                        }
                        //30msが過ぎてgucTimerTickが0になると出る
                        if(gucTimerTick==0){
                                ucResult = DRS_RXTIMEOUT;
                                break;
                        }
                }

                //パケットの受信が正常に完了した時
                if(ucResult == DRS_RXCOMPLETE){
                        //受け取ったデータの2バイトを、共用の  に保存
                        unCaliPos.uiCaliPos = stRcvPacket.unData.stRWData.ucData[0] |
                                        (stRcvPacket.unData.stRWData.ucData[1]<<8);

                        //目標の位置と5以で近接した場合はLEDをすべてオフ
                        if((iPos - unCaliPos.stCaliPos.iPosition) > -5 &&
                          (iPos - unCaliPos.stCaliPos.iPosition) < 5){
                                PORTC = 0b01111111;
                        }
                        //目標位置の5以外で外に出るとLEDをすべてオン
                        else{
                                PORTC = 0b00000000;
                        }
                }
                //パケットの受信が正常に行われていない場合LEDをすべてオン
                else{
                        PORTC = 0b00000000;
                }
        }

        //  の形に合わせて整 値を返す
        return 1;
}
```

例の説明

まず、HerkuleXCtrl.cで#include文を使用して含まれているヘッダファイルを説明します。

```
#ifndef TIMER0_H_
#define TIMER0_H_

...

#endif /* TIMER0_H_ */
```

```
#define __TIMER0_C
        #include "TIMER0.h"
#undef __TIMER0_C
```

（TIMER0.c の一部分）

一番上のコードは、すべての全域変数と関数原型を一度だけ宣言してくれるための部分です。その下にあるTIMER0.cの一部分のように、TIMERT0.cでTIMER0.hを # include文に含める場合は、_TIMERT0_Cを # defineで宣言して含んでいます。これにより、TIMER0.cでTIMER0.hをコンパイルするときは、

```
volatile unsigned char gucTimerTick;
void TIMER0_Init(void);
```

でコンパイルします。変数と関数の前のexternは他のソースファイルで宣言され、変数や関数を、現在のソースファイルでアクセスしたり、呼び出す際に使用されます。 TIMER0.c以外のファイルでTIMER0.cの変数と関数を使用するためにexternを付けてコンパイルします。

これまでに説明したコンパイラに関連した内容は、TIMER0.hではなくて、他のヘッダファイルでもをすべて同じように適用されるので、その後、他のヘッダファイルでは別々に宣言しません。

TIMER0.hの全域変数gucTimerTickはChapter5で使用する変数のように、0でない場合2msごとに1ずつ減る変数です。

TIMER0_Init()関数は、TIMER0の初期化関数で、TCNT0の初期値に設定してタイマー/カウンタ0オーバーフローインターラプト を活性化 して、指定された分周比でタイマー/カウンタ0を開始します。

herkulex.h

herkulex.hでHerkuleXを制御するために必要なプロトコルに関する宣言文と構造体が定義されています。最初のプロトコルの宣言部分では、パケットを構成するヘッダ、SIZE、ID、CMD、Check Sumに関連するさまざまな情報が宣言されています。

その次には、各CMDによるパケット構造体が定義されていて、ユーザーが送信するパケットの内容を簡単に構成できるようになっています。DrsJogはI_JOGとS_JOGパケットでHerkuleXが移動する位置/速度情報が入った構造体であり、DrsSetは同じようにI_JOGとS_JOGパケットで様々な設定情報が入った構造体です。DrsIJogとDrsSJogは、それぞれI_JOGとS_JOGパケットでサーボモータの一つに対応する情報が入った構造体であり、DrsIJogDataとDrsSJogDataは、サーボモータ全体に対応する情報が あります。

DrsRWDataはEEP_WRITE、EEP_READ、RAM_WRITE、RAM_READの4つの命令で使用します。データの最初のバイトはレジスタのアドレス、2番目のバイトは、読み取りまたは使用するデータの長さ、3番目のバイトは使用するデータ(WRITE命令の場合のみ)が入るパケットの形式を構造体に示しました。

DrsIJogData, DrsSJogData, DrsRWData すべてのパケットでOptional Data部分を表す構造体なので、DrsDataという共用体で囲んで、同じメモリ領域をCMDの種類によって異なるアクセスができるようにします。

最後に、DrsPacketは、ヘッダー、SIZE、ID、CMD、Check SumとOptional Dataを含むパケット全体に関する構造体です。HerkuleXCtrl.cでDrsPacket変数を宣言して、パケットを構成し、送信、受信、受信したパケットを解析します。

```
enum{
        DRS_RXWAITING,
        DRS_RXCOMPLETE,
        DRS_HEADERNOTFOUND,
        DRS_INVALIDSIZE,
        DRS_UNKNOWNCMD,
        DRS_INVALIDID,
        DRS_CHKSUMERROR,
        DRS_RXTIMEOUT
}DrsRxStatus;
```

パケット構造体hklx_ucReceivePacketから返される受信状態の結果の値が列挙型変数(enum)DrsRxStatusに宣言されています。

```
HERKULEX_EXT void hklx_Init(unsigned long ulBaudRate);
HERKULEX_EXT void hklx_SendPacket(DrsPacket stPacket);
HERKULEX_EXT unsigned char hklx_ucReceivePacket(DrsPacket *pstPacket);
```

上記のコードは、herkulex.hで宣言された関数です。hklx_Init()は、文字通りHerkuleXを制御するためにATmega128を初期化する関数です。タイマー/カウンタ0とUSART0を初期化します。hklx_SendPacket()はHerkuleXにパケットを送信する関数です。 SIZE、ID、CMD、Optional DataをDrsPacket変数に入力してhklx_SendPacket()を呼び出すと、この関数でstPacketにヘッダーとチェックサムを入力し、USART0に送信します。Hklx_ucReceivePacket()は、HerkuleXからパケットを受信する関数です。USART0の受信バッファの内容を検査して、完全なパケットをすべて受けたらpstPacketにコピーしてDRS_RXCOMPLETEを返します。そうでない場合は、現在の状況に合ったDrsRxStatusの他の値を返します。

HerkuleXCtrl.c

```
#define F_CPU 16000000
```

#define文はコードの前処理命令の一つで、今後F_CPUという値が出ると16000000に置き換えられることを意味しています。DRCのATmega128は16MHzのクロック速度で動作します。クロック速度に比例して命令の処理速度が異なりますので、この命令を通じて、このプログラムが動作する速度を明示するとdelay関数が正常に動作します。

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

avr/io.hヘッダファイルにはDDRX、PORTXレジスタなどAVR制御のために必要なレジスタが定義されており、avr/ interrupt.hヘッダファイルには、インターラプト を使用するための関数が定義されています。util/ delay.hヘッダファイルには、_delay_ms()などのよく使用するディレイ関数が定義されているので # include文を使用しています。

```
#include <herkulex.h>
#include <TIMER0.h>
```

herkulex.h ヘッダファイルには、パケットを構成して送信するための宣言と構造体、および関数が定義されているので、ぜひ含める必要があります。そしてgucTimerTickを使用してパケットのTimeoutを判定するために、TIMER0.hも含めます。

```
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;
```

前に説明したように、HerkuleXレジスタマップCalibrated Positionは位置情報とGPIOの入力情報が含まれています。必要なのは位置情報ですが、毎回Calibrated Positionの値を読み込むたびにGPIO入力情報も一緒に読み込まれてきますのでビット演算を通して位置の値のみを示す必要があります。このような不便さを解消するために宣言するのが上記の構造体と共用体です。DrsCaliPosはCalibrated Positionの各ビットがどのように使用されるかを示す構造体であり、DrsUnionCaliPosはこの構造体をunsigned intのようにまとめて、Calibrated Positionの2つのバイト情報をDrsCaliPosとunsigned intの二つの方法でアクセスできるようにしました。

```
DrsPacket stSendPacket, stRcvPacket;
DrsUnionCaliPos unCaliPos;
unsigned char ucResult;
int iPos = 400;
```

メイン関数で使用する変数を宣言します。DrsPacket構造体タイプで報告するための変数と受け取る時に使われる変数を宣言し、受信したデータを処理するためにDrsUnionCaliPos共用体のタイプにunCaliPosを宣言します。そしてhklx_ucReceivePacket()の結果の値が保存されるucResultを宣言し、モータの目標位置であるiPosを宣言し、400に初期化します。

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

ポートCの0～6番ピンを出力に設定し、出力値を1にします。これにより、すべてのLEDが消灯します。

```
cli();
hklx_Init(115200);
sei();
```

インターラプトに対する設定を変更中にインターラプトが発生して予期せぬ誤動作が発生することを防止するために、まずcli()命令を通じて全体のインターラプトを無効にします。その後hklx_Init関数を使用してHerkuleXを制御するためのタイマー/カウンタ0とUSART0を初期化します。使用するボーレートが115200であるため、パラメータとしてその値を使用しています。初期化が終わった後にsei()命令を通じて全体のインターラプトを活性化 します。

```
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;

hklx_SendPacket(stSendPacket);
```

モーターが自ら動くように制御するためには、モータのトルクを印加する必要があります。上記のコードは、トルクを印加させるためのRAM_WRITEパケットを構成する送信部分です。パケットのSIZEは基本MIN_PACKET_SIZE(7バイト)に加え、アドレス+長さ+データ(1バイト)で、総MIN_PACKET_SIZE+3(10バイト)であり、対象モータのIDが253であるため、IDは253です。CMDはCMD_RAM_WRITE(0x03)であり、unData共用体のメンバーの中でstRWData構造体のucAddress、ucLen、ucData[0]の値をそれぞれ52、1、0x60にしています。その後hklx_SendPacket()関数を呼び出してパケットを送信します。

```
if(iPos == 400){
        iPos = 624;
}
else{
        iPos = 400;
}
```

ここからはwhile文の中です。モーターの目標位置を示す変数であるiPosはwhile文を一度繰り返すたびに交互に400と624での計算が必要なので、上記のように値を変えています。

```
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;

hklx_SendPacket(stSendPacket);

_delay_ms(1000);
```

モーターがiPosの位置に動くようにI_JOGパケットを構成し、送信する部分です。パケットのSIZEは基本的にMIN_PACKET_SIZE(7バイト)に加え、モーター一つのデータがCMD_I_JOG_STRUCT_SIZE(5バイト)であるため、二つを合わせた長さ(12バイト)になります。I_JOGは複数のモーターにまとめて命令するのでIDはBROADCAST_ID(254)です。CMDはCMD_I_JOG(0x05)であり、unData共用体のメンバーの中でstIJogData構造体のstIJog[0]に情報を適切に入れています。

stJogのuiValueは目標の場所iPos値を入れており、位置制御モードなので、stSetのucModeは0、LED3個をすべてオンにするにstSetのucLedGreen、ucLedBlue、ucLedRedをすべて1に、ucIDは253で、ucPlayTimeは50(560ms)に設定します。その後hklx_SendPacket()関数を呼び出してパケットを送り、動きが終わるまで1秒間待機します。

```
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);
```

モーターの動きが終わった後に、モータの現在位置の値を読み出して、モーターが、実際に目標位置に到達したことを確認します。上記のコードは、Calibrated　Positionの値を読み取るためのRAM_READパケットを構成している送信部分です。パケットのSIZEは基本MIN_PACKET_SIZE(7バイト)に加え、アドレス+長さで総MIN_PACKET_SIZE+2(9バイト)であり、対象モータのIDが253であるため、IDは253です。 CMDはCMD_RAM_READ(0x04)であり、unData共用体のメンバーの中でstRW-Data構造体のucAddress、ucLenの値をそれぞれ58、2にしています。その後hklx_SendPacket()関数を呼び出してパケットを送信します。

```
gucTimerTick=15;
while(1){
        ucResult = hklx_ucReceivePacket(&stRcvPacket);
        if(ucResult != DRS_RXWAITING){
                break;
        }

        if(gucTimerTick==0){
                ucResult = DRS_RXTIMEOUT;
                break;
        }
}
```

Timeout t時間を設定してACK Packetが入ってくるの待っている関数です。libherkulex.aファイルには、タイマー/カウンタ0オーバーフローインターラプトサービスルーチンが含まれていますので、HerkuleXCtrl.cでgucTimerTick値15にすると、この変数は、タイマー/カウンタ0オーバーフローインターラプトが発生する2msごとに1ずつ減って、約30ms後にゼロになります。これまでwhile文の中ではずっとhklx_ucReceivePacket()関数を呼び出してucResultにその結果を保存します。
ACKパケットが入ってくるのを待つときは、結果の値がDRS_RXWAITINGであり、完成されたパケットを受け取ったり、そのほか、間違ったパケットが入ってきた状況では、DrsRxStatus中、それに合わせた状態の値が返されます。

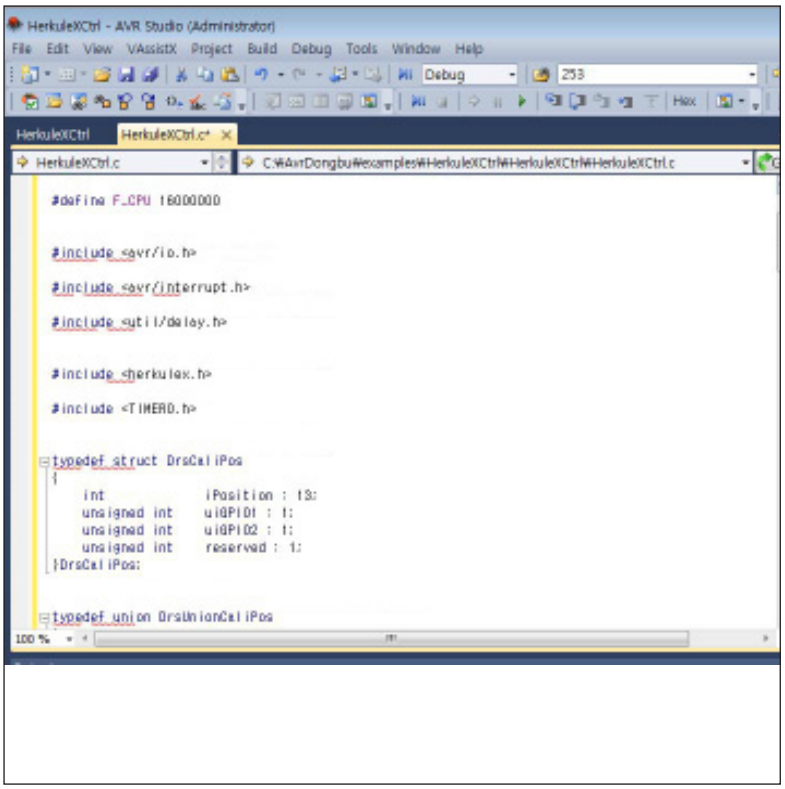
そのため、ucResultはDRS_RXWAITINGがない場合には、break文を使用してwhile文を出ます。もし30msが過ぎてgucTimerTickが0になったのにまだucResultがDRS_RXWAITINGの場合には、ucResultをDRS_RXTIMEOUTに作成しwhile文から脱出します。

```
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;
}
```

もしucResultがDRS_RXCOMPLETEに完全なACKパケットを返したならば、受けたRAM_READのACKパケットのCalibrated Positionデータが含まれている部分をunCaliPos共用体変数のメンバーuiCaliPosに保存します。そしてunCaliPosのstCaliPosのiPositionにアクセスすると、読み取ったデータから位置情報に対応する下位13ビットだけアクセスして使用ができます。if文を使用して、目標の場所のiPosが、現在の場所であるunCaliPos.stCaliPos.iPositionと5未満で差が出るのか検査し、もしそうならばDRCのLEDをオフにします。もし5以上の差が出た場合にはLEDをオンにします。また、パケットエラーが発生して完全なACKパケットを受け取っていない場合でも、LEDをオンにします。

実行

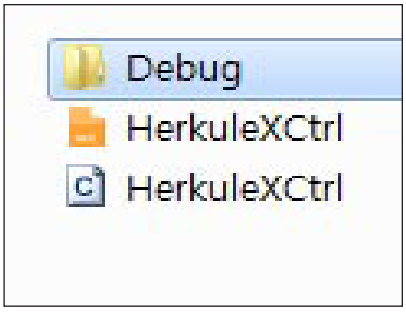
ファームウェアをダウンロードして実行すると、ID253のHerkuleXが400、624の位置を約1秒周期で往復します。通常のモーターであればDRCのLEDはオフになっている必要があります。



01 Build

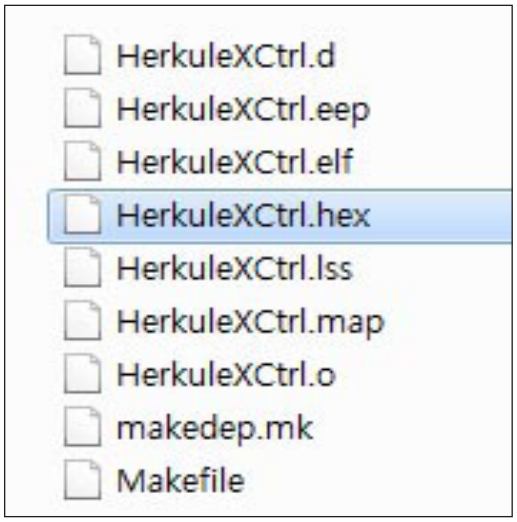
Buildメニューを選択しBuild Solutionをクリックしてビルドします。

1 succeeded0 failedが出るとエラーなしでコンパイルが完了したということです。



02 Debug フォルダ

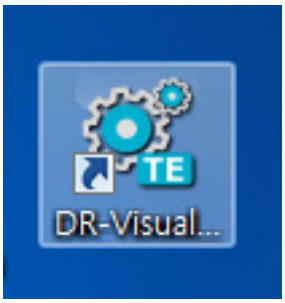
Build後にソースがあるフォルダに入るとDebugフォルダが作成されています。クリックしてフォルダの中身を表示します。



03 hex 確認

Debugフォルダの中のhexファイルを確認します。Hexファイルは制御機に移植するファイルです。

DR-Visual Logicで、ファームウェアのリカバリーを行います。



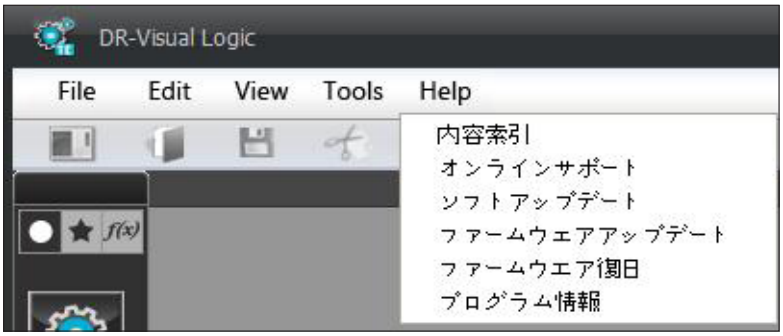
04 Visual Logic

DR-Visual Logicを実行します。



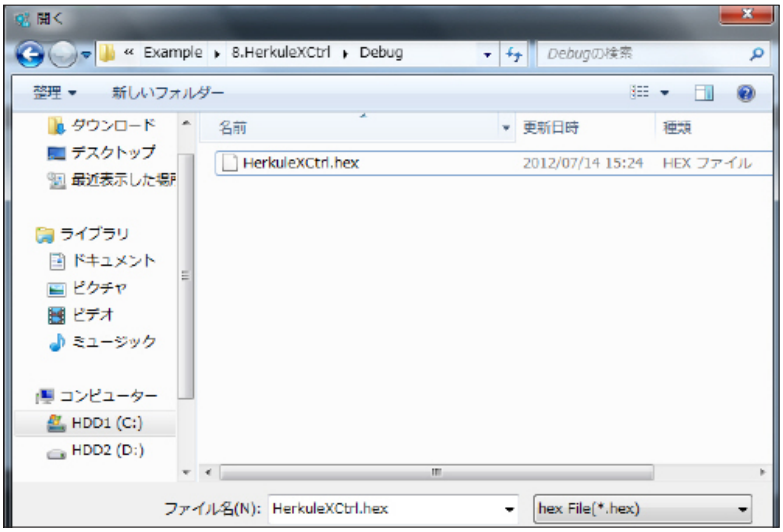
05 接続する

COM 番号を設定して接続するをクリックします。



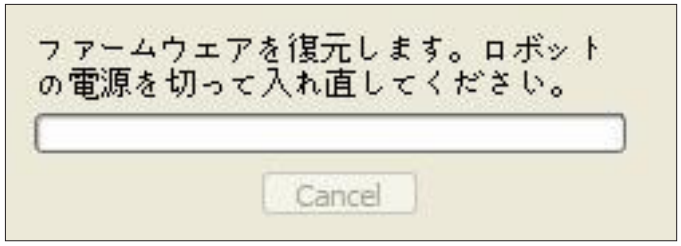
06 ファームウェアの修復

[ヘルプ]メニューからファームウェア修復をクリックします。



07 ロード

Debug　フォルダにHexファイルをロードします。



08 ファームウェアの修復

“ロボットの電源を切って入れ直してください”というメッセージが出ます。
コントローラの電源ボタンを一度切ってから再度入れ直してください。



09 修復の開始

ファームウェアの復元が開始されます。プロセスが完了したら、ポップアップウィンドウが消えます。
Tera Term を利用して通信テストを行います。



10 接続

ファームウェアのダウンロードが完了し、LEDに変化がなければ正常に移植された状態です。
もし移植ファームウェアがエラーの場合はLEDが点滅します。



11 実行

モーターが左右に移動します。
ファームウェアをダウンロードして実行すると、ID253のHerkuleXが400、624の位置を約1秒周期で往復します。通常のモーターであればDRCのLEDは、すべてオフになっている必要があります。



12 分離

モーターを分離すると、すべてのLEDが点灯します。コントローラとモーター間の通信が接続されていないことを意味します。

タイマー/カウンタでブザー音階を再生する

ATmega128のタイマー・カウンタを使用して様々な音をブザーで再生します。

AVR 原理

Chapter5で説明したように、ATmega128には8ビットのタイマー/カウンタであるタイマー/カウンタ0とタイマー/カウンタ2、16ビットのタイマー/カウンタであるタイマー/カウンタ1とタイマー/カウンタ3という合計4つのタイマー/カウンタがあります。今回のChapterでは、タイマー/カウンタ2を使用します。Chapter5で説明したタイマー/カウンタ0と同じように8ビットのタイマー/カウンタなので、だいたい似ています。今回のChapterでは、タイマー/カウンタに関する一般情報については省略して、タイマー/カウンタ2とタイマー/カウンタ0の違いだけ簡単に説明します。別に記載しない部分はタイマー/カウンタ0のレジスタ名やビット名をタイマー/カウンタ2と変更するとChapter5のタイマー/カウンタ0の説明と同じです。

今回のChapterでは、タイマー/カウンタ2をCTC(Clear Timer on Compare match)モードで使用して、適当なOCR2値に音階を再生します。

タイマー/カウンタ2に関連した代表的なレジスタは以下の通りです。

TCCR2:タイマー/カウンタ2制御レジスタ

ビット	7	6	5	4	3	2	1	0
	FOC2	WGM20	COM21	COM20	WGM21	CS22	CS21	CS20
読み取り/書き込み	W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初期値	0	0	0	0	0	0	0	0

- ビット2:0番CS22:0:クロック選択

タイマー/カウンタ2が動作するクロックを選択します。ATmega128に入る基本クロックの周波数を任意の割合で分けてTCNT2で使用するかどうかを決めます。DRCの基本クロックの周波数は16MHzです。そのためCS22:0を0b100に設定した場合、タイマー/カウンタ2の動作周波数は16Mhz/256=62.5kHzとなります。16usごとにTCNT2が1ずつ増加します。
タイマー/カウンタ0のCS02:0とタイマー/カウンタ2のCS22:0はクロックが分かれる割合が異なりますので、使用する時に注意が必要です。もしT2ピンの外部クロックを使用するように設定すると、T2ピン(ポートDの7番ピン、PD7)の入出力にかかわらず、値の変化が発生し、TCNT2値が変化します。これはすぐに外部信号を受けて、クロックとして使用できますが、ユーザーがPD7の値を直接変更してソフトウェア的にクロックを作り出すという意味です。

CS22	CS21	CS20	説明
0	0	0	動作しない
0	0	1	クロック/1
0	1	0	クロック/8
0	1	1	クロック/64
1	0	0	クロック/256
1	0	1	クロック/1024
1	1	0	T2ピンから外部クロック(立ち下がりエッジ)
1	1	1	T2ピンから外部クロック(立ち上がりエッジ)

■ TCNT2:タイマー/カウンタ2レジスタ

ビット	7	6	5	4	3	2	1	0
	TCNT2[7:0]							
読み取り/書き込み	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初期値	0	0	0	0	0	0	0	0

TCNT2はタイマー/カウンタ2の現在のカウント値を保存するレジスタです。TCNT0と同じように動作します。

■ OCR2 : タイマー/カウンタ2の出力比較レジスタ

ビット	7	6	5	4	3	2	1	0
	OCR2[7:0]							
読み取り/書き込み	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初期値	0	0	0	0	0	0	0	0

OCR2レジスタの中に入っている値は、TCNT2の値が変化するたびにTCN2と比較されます。 OCR0と同じように動作します。

■ TIMSK: タイマー/カウンタ2割り込みマスクレジスタ

ビット	7	6	5	4	3	2	1	0
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIEB	TOIE1	OCIE0	TOIE0
読み取り/書き込み	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初期値	0	0	0	0	0	0	0	0

– ビット7番OCIE2:タイマー/カウンタ2の出力の比較マッチ割り込みが有効になる

このビットが1に設定されるとタイマー/カウンタ2の比較マッチ割り込みが有効になります。このビットが1に設定されSREGレジスタのIビットが1に設定された状態で比較マッチが起こるとTIFRレジスタのOCF2ビットが1になり、該当する割り込みサービスルーチンが実行されます。

– ビット6番TOIE2:タイマー/カウンタ2オーバーフロー割り込みが有効になる

このビットが1に設定されるとタイマー/カウンタ2のオーバーフロー割り込みが有効になります。このビットが1に設定されSREGレジスタのIビットが1に設定された状態でオーバーフローが起こるとTIFRレジスタのTOV2ビットが1になり、該当する割り込みサービスルーチンが実行されます。

■ TIFR : タイマー/カウンタ2割り込みフラグレジスタ

ビット	7	6	5	4	3	2	1	0
	OCF2	TOV2	ICF1	OCF1A	OCFB	TOV1	OCF0	TOV0
読み取り/書き込み	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初期値	0	0	0	0	0	0	0	0

–ビット7番OCF2:タイマー/カウンタ2の出力の比較フラグ

このビットは、TCNT2とOCR2間に比較マッチが起こると1に設定されます。このビットは、対応する割り込みサービスルーチンに進入する時に、ハードウェアによって0に初期化されます。ソフトウェア的に手動でこのビットを0に変更するには、このビットに1を使用します。SREGのIビットとOCIE2、OCF2がすべて1の場合、タイマ/カウンタ2比較マッチ割り込みが実行されます。

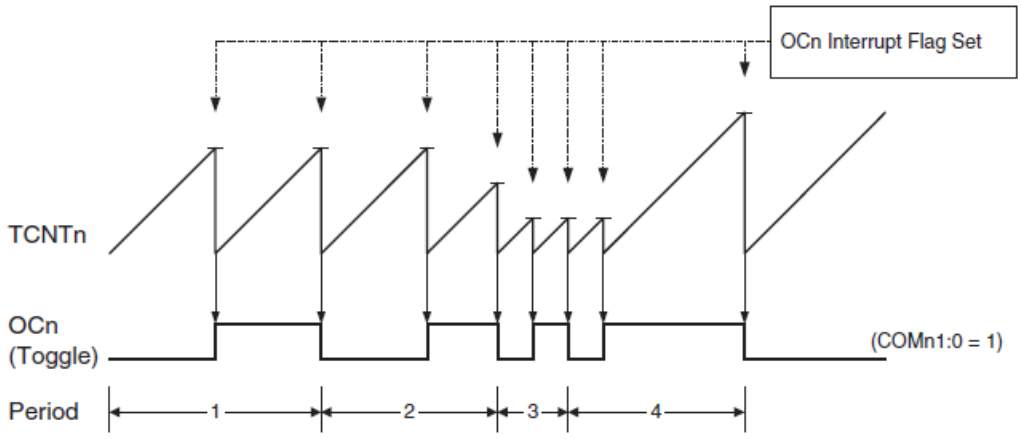
– ビット0番TOV2:タイマー/カウンタ2オーバーフローフラグ

このビットはタイマー/カウンタ2でオーバーフローが起こると1に設定され、該当する割り込みサービスルーチンに進入する時に、ハードウェアによって0に初期化されます。ソフトウェア的に手動でこのビットを0に変更するには、このビットに1を使用します。SREGのIビットとTOIE2、TOV2がすべて1の場合、タイマー/カウンタ2オーバーフロー割り込みが実行されます。Phase Correct PWMモードでは、他のモードとは異なり、TCNT2の増減方向がダウンカウントでアップカウントに変わる時に(TCNT2=0の時に)TOV2ビットが1に設定されます。

タイマー/カウンタ2はタイマー/カウンタ0と異なる非同期動作をサポートしていないため、ASSR、SFIORなどのレジスタは関係ありません。詳細はATmega128のデータシートや他の書籍を参照してください。

タイマー/カウンタ2の動作モードも4種類あり、タイマー/カウンタ0の4つのモードと同じです。この中で使用するモードはCTCモードです。 CTCモードは、Clear Timer on Compareモードの略で、OCR2レジスタがタイマーの初期化の周期を決定するために使用されます。CTCモードでは、TCNT2レジスタがOCR2と等しくなる瞬間、つまり比較マッチが起こる瞬間0に初期化されます。つまりOCR2レジスタの値がカウンタの最大値を決定します。比較マッチが起こる瞬間OCF2ビットは1に変わり、タイマー/カウンタ2のオーバーフロー割り込みサービスルーチンが実行されると0に初期化されます。COM21:0の値を0b01に設定すると、比較マッチが起こるたびにOC2ピンがトグルするように設定できます。これを利用するとOC2ピンに周期的なパルスを作成でき、OCR2の値を変えるに応じてパルスの周波数を変えることができます。

このような原理を利用して、様々な周波数でOC2ピンをトグルしてブザーから様々な音階を再生できます。



CTCモードでOC2ピンがトグルされる周波数を計算する式は次のとおりです。

$$f_{OC2} = \frac{f_{clk_I/O}}{2 \times N \times (1 + OCR2)}$$

$f_{clk_I/O}$ は、現在ATmega128のシステムクロックであるため、16MHzです。NはCS22:0によってクロックが割れる割合であり、例では、256です。OCR2は名前の通り、OCR2レジスタの値です。この計算式を通って出てくる周波数である f_{OC2} がブザーが鳴る音の周波数です。希望音に該当するOCR2の値を取得するには、この式を逆に解いて入手します。

$$OCR2 = \frac{f_{clk_I/O}}{2 \times N \times f_{OC2}} - 1$$

例えば、5オクターブのラ(A)音の周波数は880Hzです。5オクターブのラ音を出すためのOCR2の値は、

$$OCR2 = \frac{16{,}000{,}000}{2 \times 256 \times 880} - 1 = 34.51 \approx 35$$

です。

例で使用する周波数と、上記の計算式を使って計算したOCR2の値は下表のとおりです。

音階	周波数(Hz)	OCR2の値
5オクターブのド(C)	523.25	59
5オクターブのレ(D)	587.33	52
5オクターブのミ(E)	659.26	46
5オクターブのファ(F)	698.46	44
5オクターブのソ(G)	784.99	39
5オクターブのラ(A)	880	35
5オクターブのシ(B)	987.77	31
6オクターブのド(C)	1046.50	29

H/W の原理

今回の例では、タイマー/カウンタ2のOC2ピン(PB7)を利用してブザーを制御します。ブザー回路に関する内容は、Chapter3を参照してください。

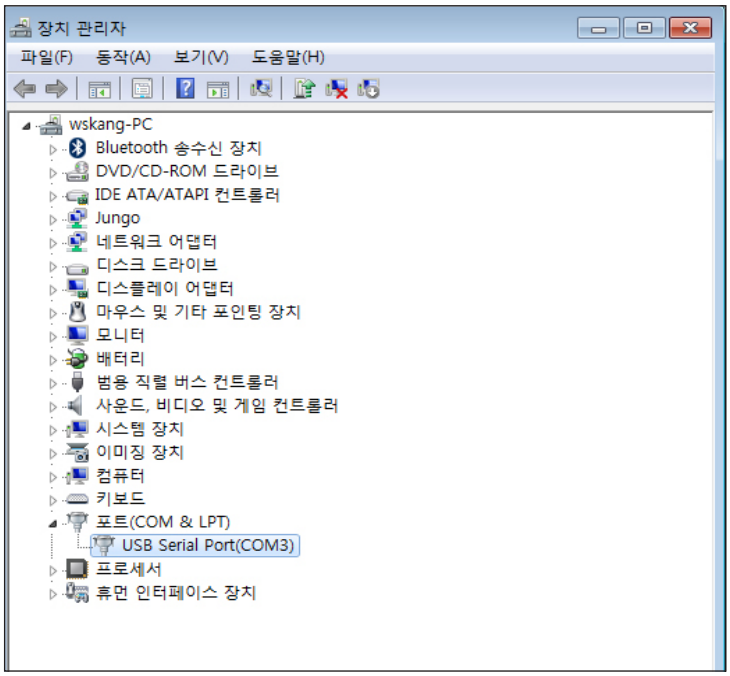
準備

ハードウェアを準備する



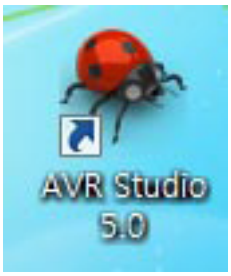
01 接続する

DRC制御機のアダプターで電源を接続して、USB to Serial Cable、PCとDRCを接続します。



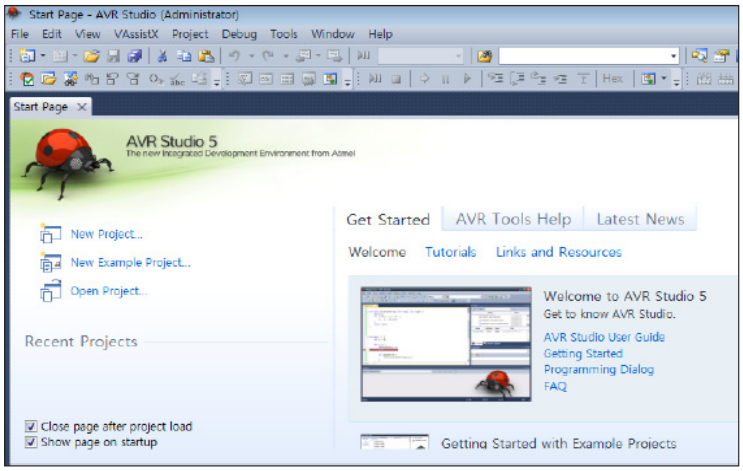
02 COM portを接続する

[マイコンピュータ]>[プロパティ]>[デバイスマネージャ]でCOM接続を確認します。



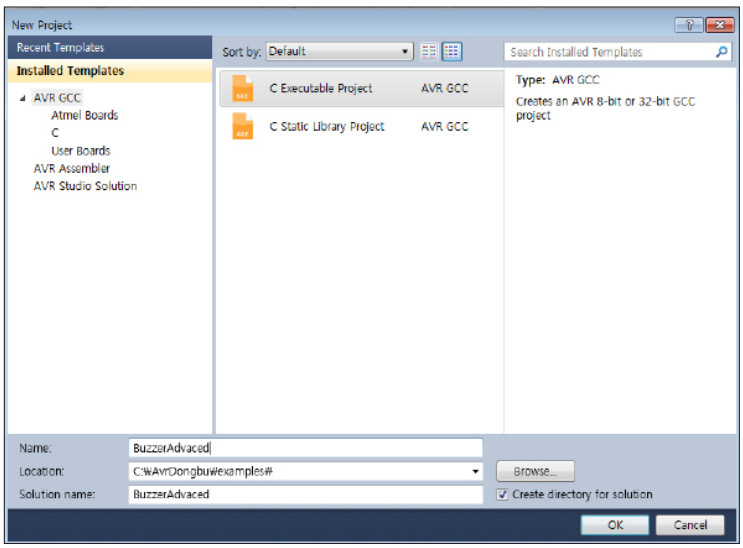
01 実行

AVRStudioをクリックして、実行します。



02 New Project

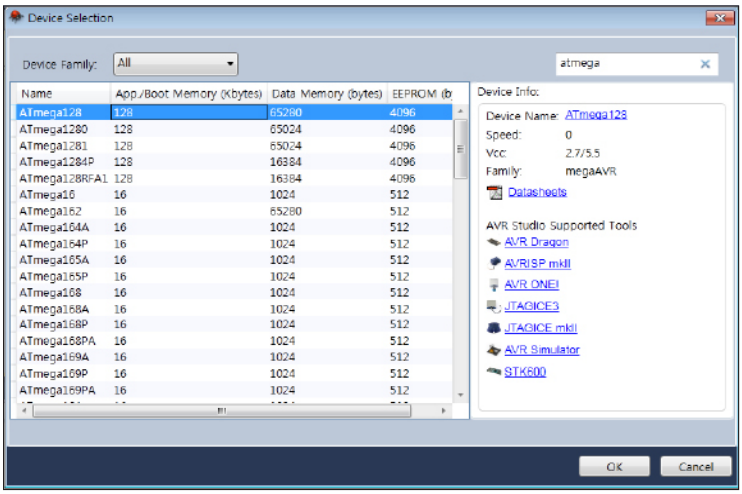
New Projectをクリックします。



03 GCC選択

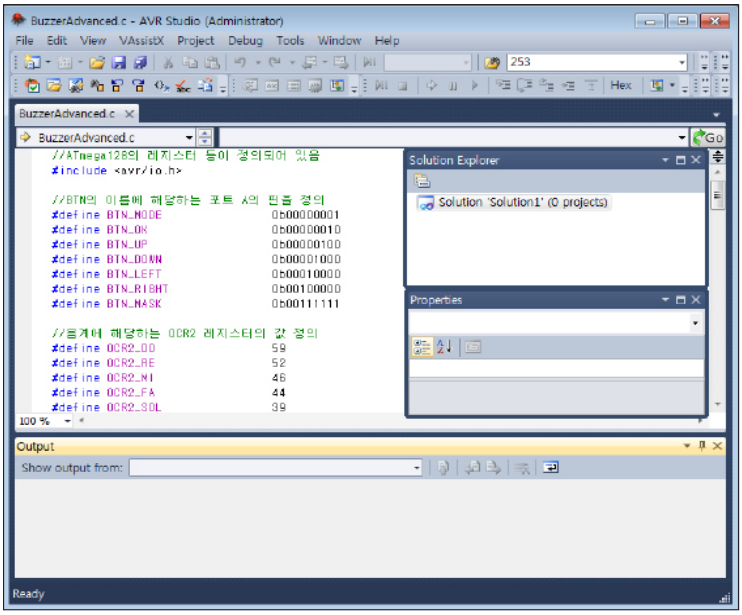
C Executable Projectを選択します。

Nameを“Buzzer Advaced”と入力し、Create directory for solution をクリックして、OK ボタンを押します。



04 ATmega 選択

右上からatmegaを検索して、
リスト中のATmega128を選択します。



05 Editor

編集ウィンドウでコーディングを開始します。
下の例を入力します。

例

DRCの6つのボタンを使用して、押すボタンによってド～ラまでのブザー音が鳴る例をタイマー/カウンタ2のCTCモードで実行します。ボタンによるブザー音は以下のように割り当てられます。

ボタン	音
Mode	ド
OK	レ
左	ミ
上	ファ
右	ソ
下	ラ

BuzzerAdvanced.c

```
//ATmega128のレジスタなどが定義されている
#include <avr/io.h>

//BTNの名前に対応するポートAのピンを定義
#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

//音階に対応するOCR2レジスタの値の定義
#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

//タイマー/カウンタ2のクロック速度
#define TIMER2_DIVIDER          0b00000100

int main(void)
{
        //8ビット変数の宣言
        unsigned char ucButton;
        unsigned char ucButtonOld;

        //ポートAの入出力方向を設定
        DDRA = 0b00000000;

         //ブザーピン(PB7)の入出力方向を設定
        DDRB |= 0b10000000;

        //ポートCの入出力方向を設定
        DDRC = 0b01111111;
        //ポートCの出力値を設定
        PORTC = 0b01111111
```

```
 //タイマー/カウンタ2の初期化
TCNT2 = 0;
OCR2 = 0;
TCCR2 = 0;

//ucButtonOldを現在の値に初期化
ucButtonOld = (~PINA) & BTN_MASK;

//while 内の内容を無限に繰り返す
while(1)
{
        //ポートAからボタンの値の読み出し
        ucButton = (~PINA) & BTN_MASK;

        //ボタンの値が直前の値と異なる場合
        if(ucButtonOld != ucButton){
                //MODEボタンが押されている時に"ド"音がする
                if(ucButton & BTN_MODE){
                        TCNT2 = 0;
                        OCR2 = OCR2_DO;
                        TCCR2 = (TIMER2_DIVIDER)|(1<<WGM21)|(1<<COM20);
                }
                //OK ボタンが押されている時に"レ"音がする
                else if(ucButton & BTN_OK){
                        TCNT2 = 0;
                        OCR2 = OCR2_RE;
                        TCCR2 = (TIMER2_DIVIDER)|(1<<WGM21)|(1<<COM20);
                }
                //LEFT ボタンが押されている時に"ミ"音がする
                else if(ucButton & BTN_LEFT){
                        TCNT2 = 0;
                        OCR2 = OCR2_MI;
                        TCCR2 = (TIMER2_DIVIDER)|(1<<WGM21)|(1<<COM20);
                }
                //UP ボタンが押されている時に"ファ"音がする
                else if(ucButton & BTN_UP){
                        TCNT2 = 0;
                        OCR2 = OCR2_FA;
                        TCCR2 = (TIMER2_DIVIDER)|(1<<WGM21)|(1<<COM20);
                }
```

```
                    //RIGHT ボタンが押されている時に"ソ"音がする
                    else if(ucButton & BTN_RIGHT){
                        TCNT2 = 0;
                        OCR2 = OCR2_SOL;
                        TCCR2 = (TIMER2_DIVIDER)|(1<<WGM21)|(1<<COM20);
                    }
                    //DOWN ボタンが押されている時に"ラ"音がする
                    else if(ucButton & BTN_DOWN){
                        TCNT2 = 0;
                        OCR2 = OCR2_LA;
                        TCCR2 = (TIMER2_DIVIDER)|(1<<WGM21)|(1<<COM20);
                    }
                    //ボタンが押されていない場合は、サウンド停止
                    else{
                        TCNT2 = 0;
                        OCR2 = 0;
                        TCCR2 = 0;
                    }
                }

                //現在のボタンの値を別の変数に保存
                ucButtonOld = ucButton;
            }

            //関数の形に合わせて整数値を返す
            return 1;
    }
```

例の説明

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

avr/　io.hヘッダファイルには、DDRX、PORTXレジスタなどAVR制御のために必要なレジスタなどが定義されています。現在のソースコードで使用できるように#include文を使用します。

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

コードの前処理命令である＃defineを使用し、各ボタンに対応するビットとボタン全体を表すマスクを宣言して、ボタンを使用するコードを書いたり読んだりする時に簡単に確認できるようにします。

```
#define OCR2_DO                    59
                                   …
#define OCR2_DO2                   29
```

上記の表で出た各音階に応じたOCR2値を＃defineで宣言します。この値は、音を再生するためにタイマー/カウンタ2のOCR2レジスタに値を入れる時に使用します。単純に59のような数字を代入すると、”ド”の音と接続できませんが、OCR2_DOを代入すると“ド”の音が再生されることを確認できます。

```
#define TIMER2_DIVIDER             0b00000100
```

タイマー/カウンタ2のクロックを任意の割合で分割するかどうかを決定するTCCR2レジスタのCS22:0ビットの値を＃defineで宣言します。このように宣言し、残りの部分でTCCR2レジスタに値を代入するたびにTIMER2_DIVIDERを使用して代入すれば、クロックの比率を変えることがある時にTIMER2_DIVIDERの値を変更すると、他の部分を修正するなく、全体が変更されるため便利です。

```
unsigned char ucButton;
unsigned char ucButtonOld;
```

メイン関数で使用する2つの変数を宣言します。ucButtonは現在、whileループで読んだボタンの値であり、ucButtonOldは直前のwhileループで読んだボタンの値です。毎回whileループの最後でucButtonOldはucButtonの値に更新されます。この二つの変数は、ボタンが押されたり、離してボタンの値が変わった瞬間だけタイマー/カウンタ2の値を変更するために使用します。

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

ポートA、B、Cを設定します。ポートAは、ボタンが接続されているポートでDDRAのすべてに0を使用して入力を設定します。ポートBの7番ピン(PB7)はタイマー/カウンタ2のOC2出力ピンです。タイマー/カウンタ2でOC2出力ピンを使用すると設定した場合でもDDRBの7番目のビットが出力に設定されている必要があり、実際に出力が出るので、DDRBの7番目のビットに1を使用して出力を設定します。ポートCはLEDが接続されたポートであるため、DDRCのLEDに対応する7つのビットに1を使用して出力を作成し、PORTCにも7つのビットに1を使用してLEDを切ります。

```
TCNT2 = 0;
OCR2 = 0;
TCCR2 = 0;
```

タイマー/カウンタ2のレジスタを初期化します。前に説明したレジスタのTCNT2、OCR2、TCCR2を初期値である0にします。この時点でタイマー/カウンタ2はまだ動作しません。

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

前に説明したように、ucButtonOld変数には、while文で直前に読んだボタンの値が保存されます。ところが、まだwhile文に進入する前とボタンの値を読む前なのでucButtonOldの値をucButtonに変えることには意味がなく、予期しない動作を起こすことがあります。そのためwhile文に進入する直前に一度ボタンの値を読んでucButtonOldに保存します。

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

ここからはwhile文の中の内容です。ポートAからボタンの値を読んで、ucButtonに保存します。　ucButtonに保存されたボタンの値は、ボタンに対応するビットが1の場合、ボタンが押された状態、0の場合は押されていない状態を示しています。

```
if(ucButtonOld != ucButton){
                    …
}
```

直前のwhileループで読み取ったボタンの値（ucButtonOld）と先ほど読んだボタンの値（ucButton）を比較して、異なる場合にif文の中の内容を実行します。ボタンが押された瞬間や押されていたボタンが離された瞬間だけタイマー/カウンタ2の設定を変更する部分です。

```
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_MODEはDRCのModeボタンが押されたかどうかを検査する部分です。ucButtonがBTN_MODE（0b00000001）とBitwise AND（&）演算の場合と、ucButtonの0番ビットが1の場合はtrue、0の場合は偽となります。 Modeボタンが押されている場合タイマー/カウンタ2の値を変えてブザーが”ド”の音を再生します。

TCNT2は0から開始するように0に初期化して、値に応じて周波数が変化するOCR2には”ド”に該当する事前定義された値を代入します。TCCR2はWGM21を1にしてCTCモードに設定します。COM20を1にして比較マッチが起こるたびにOC2ピンをトグルにして、TIMER2_DIVIDERに定義された値にCS22:0を設定し、必要な割合でクロックを分けるようにします。この例では、CS22:0を0b100に設定して256にクロックを分割します。

続いてelse　if文のボタンとOCR2の値だけが変わて、他は同じです。if文、else if文を通してMODEボタン－ド、OKボタン－レ、左のボタン－ミ、上のボタン－　ファ、右のボタン－ソ、下のボタン－ラを再生します。

```
else{
        TCNT2 = 0;
        OCR2 = 0;
        TCCR2 = 0;
}
```

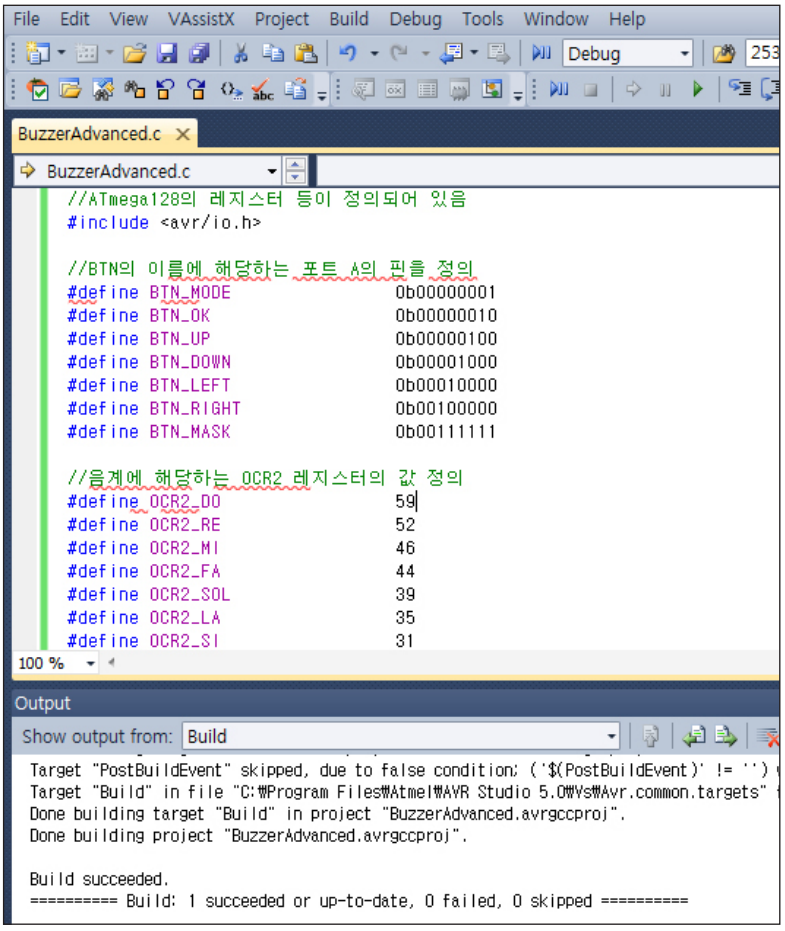
いずれのボタンも押されていない場合、タイマー/カウンタ2レジスタをすべて0にして停止させます。これにより、ブザーの音が停止します。

```
ucButtonOld = ucButton;
```

whileループが終了するたびに、現在のループで読んだucButtonの値をucButtonOldに代入して、次のループで比較に使用します。

実行

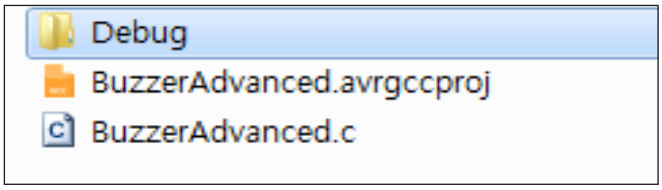
ファームウェアをダウンロードして、ボタンを押すと、該当するブザー音が再生されます。



01　Build

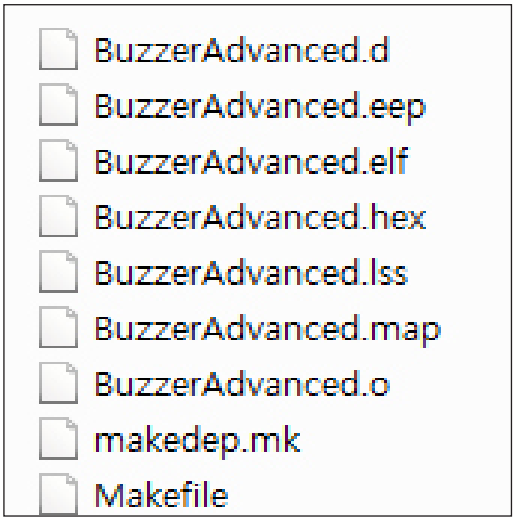
Buildメニューを選択しBuild Solutionをクリックしてビルドします。

1 succeeded0 failedが出るとエラーなしでコンパイルが完了したということです。



02 Debug フォルダ

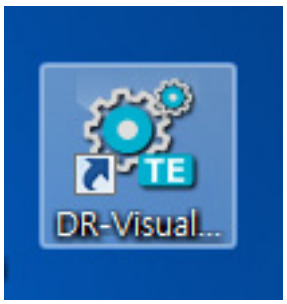
Build後にソースがあるフォルダに入るとDebugフォルダが作成されています。クリックしてフォルダの中身を見ます。



03 hex 確認

Debugフォルダの中のhexファイルを確認します。Hexファイルは制御機に移動するファイルです。

DR-Visual Logicでファームウェアの修復を行います。



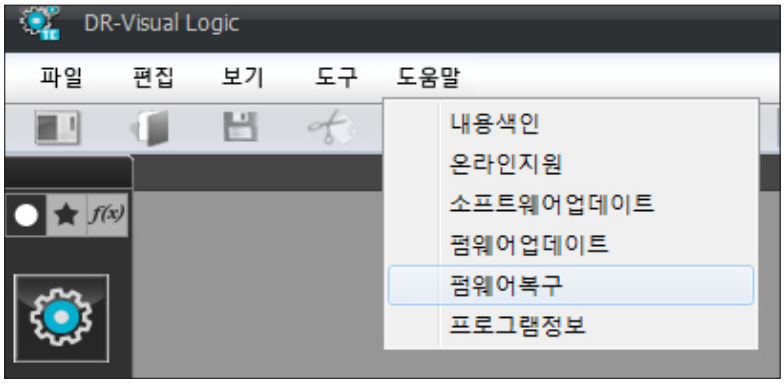
04 Visual Logic

DR-Visual Logicを実行します。



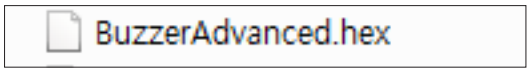
05 接続する

COM番号を設定して接続するをクリックします。



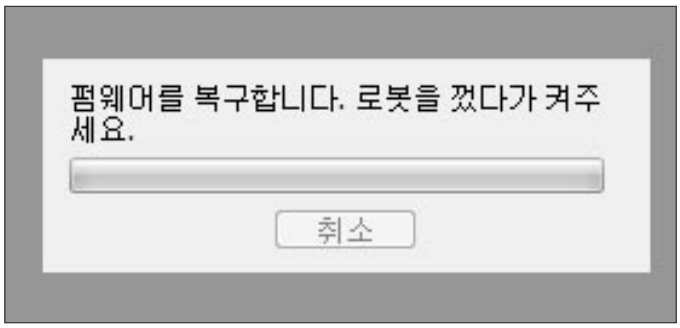
06 ファームウェアの修復

[ヘルプ]メニューからファームウェア修復をクリックします。



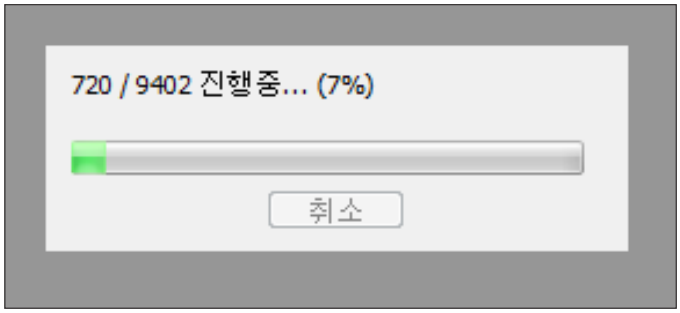
07 ロード

DebugフォルダにHexファイルをロードします。



08 ファームウェアの修復

“ロボットの電源を切って入れ直してください”というメッセージが出ます。制御機の電源ボタンを一度切ってから再度入れ直してください。



09 修復の開始

ファームウェアの修復が開始されます。



10 接続

それぞれのボタンごとに押すとブザーでドレミファソラなどの音の高さが違う音が出ます。

EEPROMを活用する

ATmega128のEEPROM(Electrically Erasable Programmable Read-Only Memory)を活用してデータを保存して読み込みます。

AVR 原理

メモリには、大きく揮発性メモリと不揮発性メモリがあります。揮発性メモリは電源がなくなれば、すべてのデータが失われるため、揮発性メモリとしています。不揮発性メモリは、その逆に、電源がなくなってもデータが残っているので、再度電源を入れ直しても、そのデータを再ロードさせ、使用できます。

ATmega128で通常使用される変数は、揮発性メモリであるSRAMに保存され、電源をオフにすると、すべての値が失われます。しかし、どのような場合でも、直前まで使用していたデータを保存して、電源を入れた時に使用したいと思うこともあります。このような場合ATmega128にEEPROMと呼ばれる一種の不揮発性メモリが内蔵されており、必要なデータを自由に読み取り、書き込むことができます。

ATmega128の内蔵EEPROMの容量は4kbyteで、内蔵SRAMと同じ大きさです。ただしEEPROMの場合100,000回以上の書き込み/読み取りを繰り返すと寿命がなくなるので、一般的な変数のように頻繁に値を変更すると長く使用できません。例えば、main関数のwhile文で一度繰り返すごとにEEPROMに値を保存すると、すぐにEEPROMが壊れて誤動作を起こします。したがって、EEPROMには必ず必要な状況でのみ値を保存するように注意してプログラムを作成する必要があります。

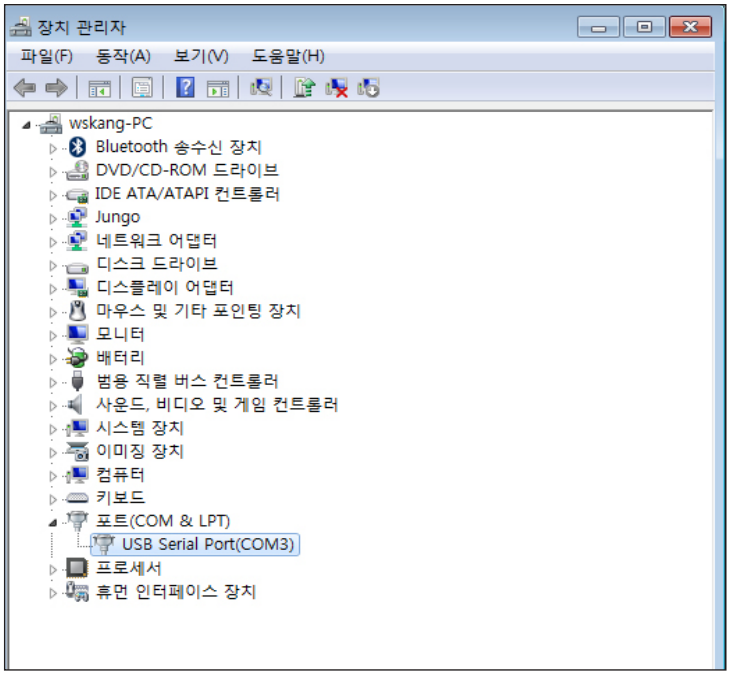
EEPROMを読み書きするために関連したレジスタにEEARH、EEARL、EEDR、EECRなどのレジスタがあります。ただし、ユーザーは、この細かいレジスタの設定について知る必要がありません。 AVR Studioは、ユーザーがeepromデータを簡単に読み書きできるようにavr/ eeprom.hヘッダファイルに関連する関数をすでに提供しています。

ハードウェアを準備する



01 接続する

DRC制御機のアダプターで電源を接続します。USB to Serial CableでPCとDRCを接続します。



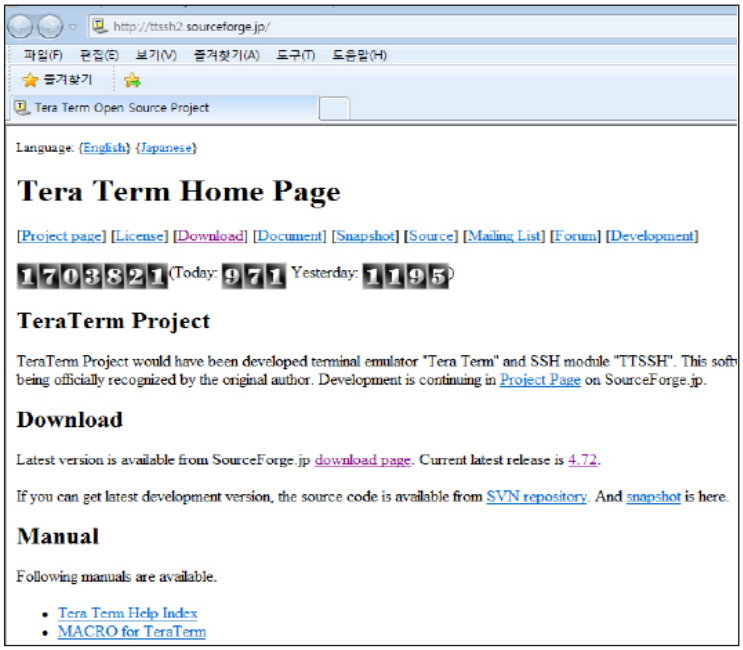
02 COM portを接続する

[マイコンピュータ]>[プロパティ]>[デバイスマネージャ]でCOM接続を確認します。



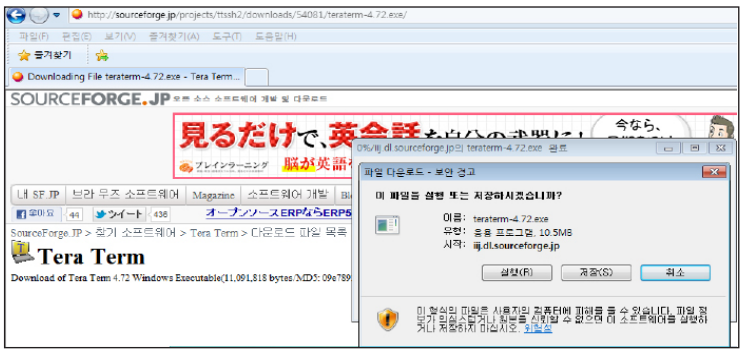
01 アドレスの検索

インターネットポータルサイトで”tera term”を検索します。



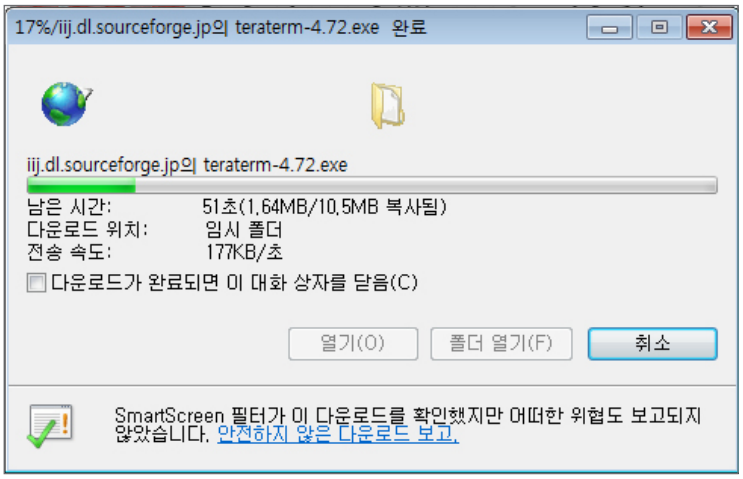
02 ホームページに接続する

Tera Termホームページに接続して“Download page”をクリックします。



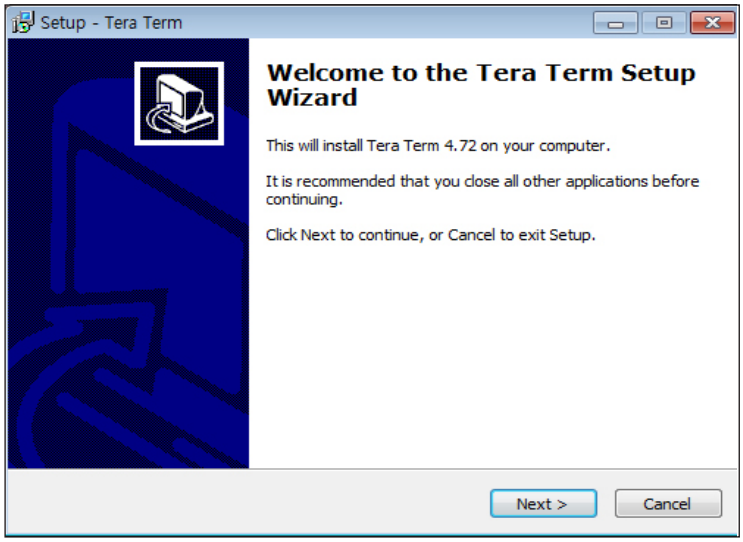
03 ファイルのダウンロード

“保存”をクリックしてファイルをダウンロードします。



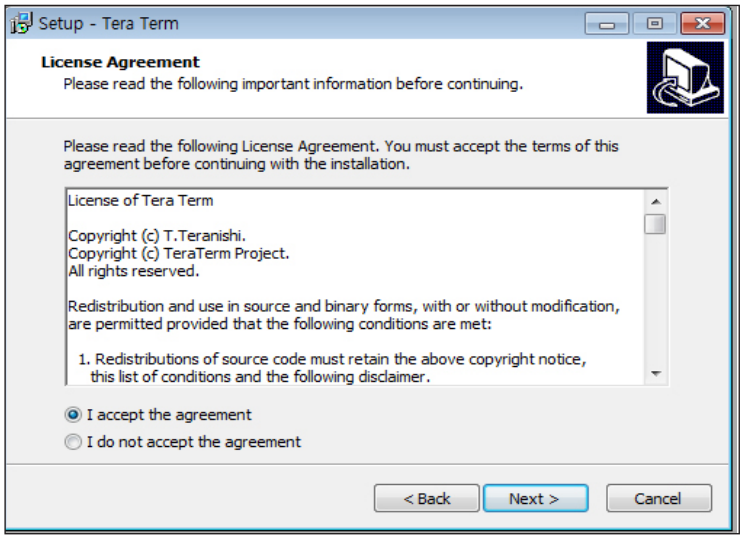
04 ファイルのダウンロードを開始

インストールファイルがダウンロードを開始します。



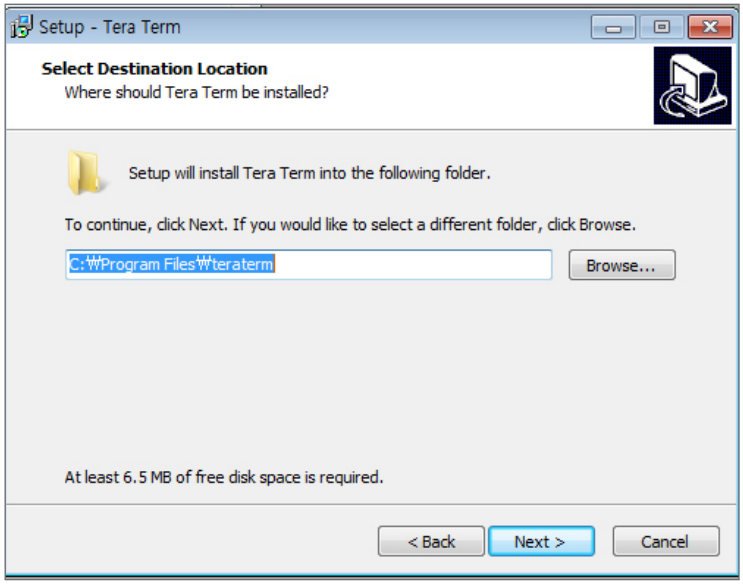
05 インストールウィザードの実行

“Next”をクリックすると、インストールウィザードが 実行されます。



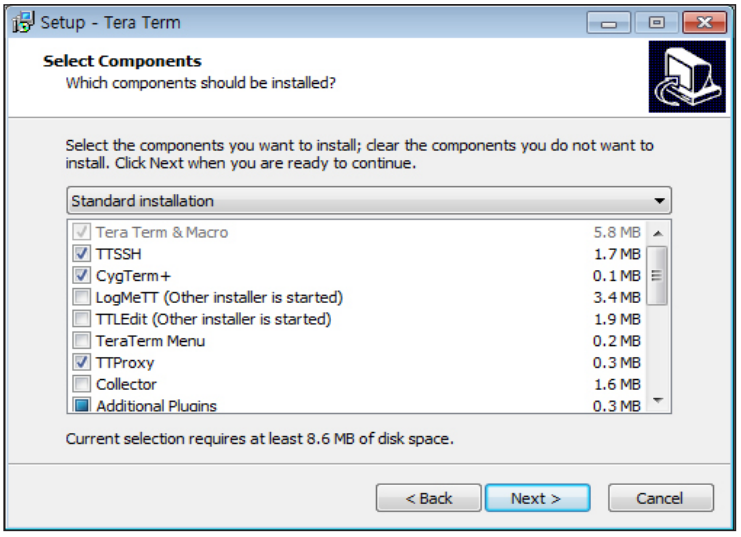
06 規約に同意

“I accept the agreement”をチェックして”Next”をクリックします。



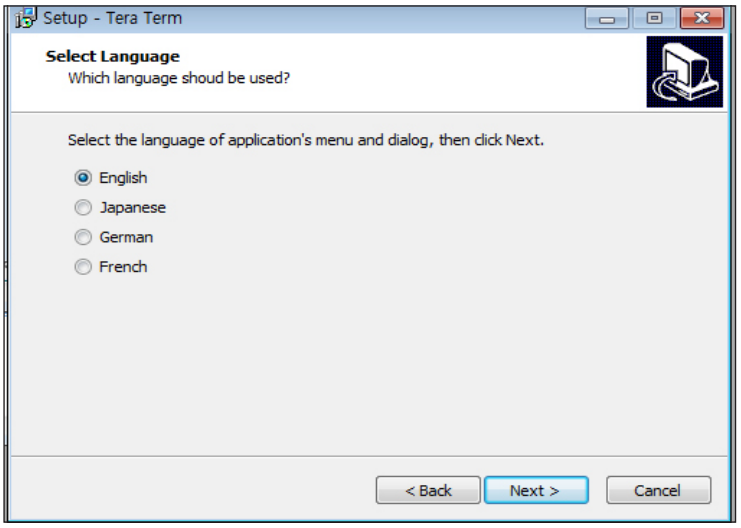
07 インストールフォルダを確認

プログラムをインストールするフォルダを確認して”Next”をクリックします。



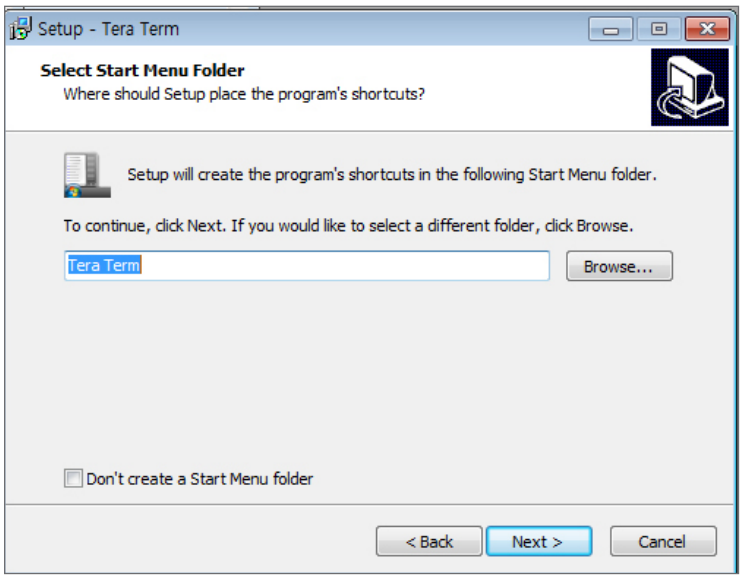
08 コンポーネントの選択

“Next”をクリックします。



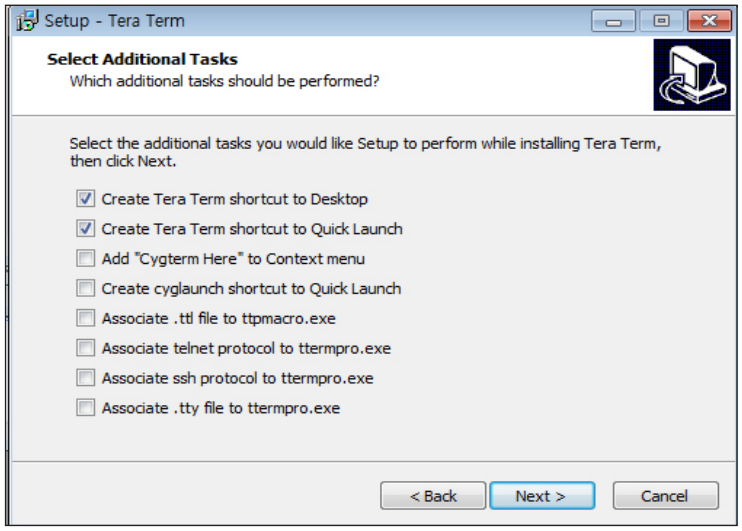
09 言語を選択

言語を選択して”Next”をクリックします。



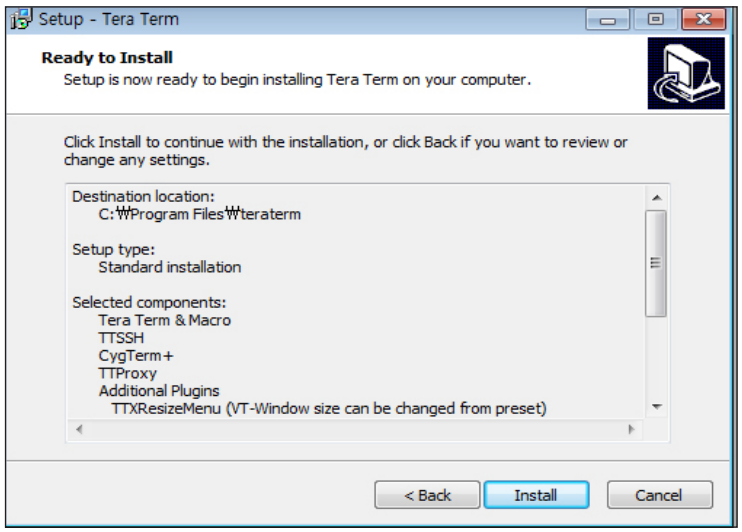
１０　スタートメニューフォルダの選択

[スタート]メニューフォルダを選択した後に “Next”をクリックします。



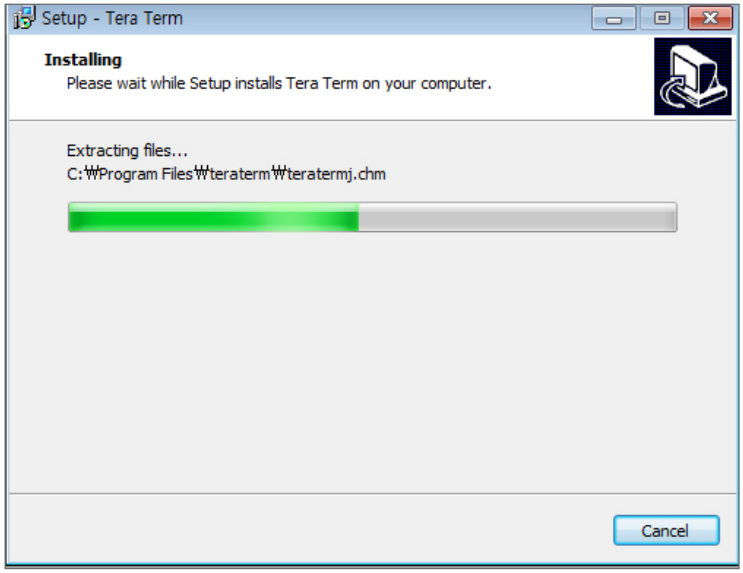
１１　追加機能の選択

必要な追加機能を選択した後、“Next”をクリックします。



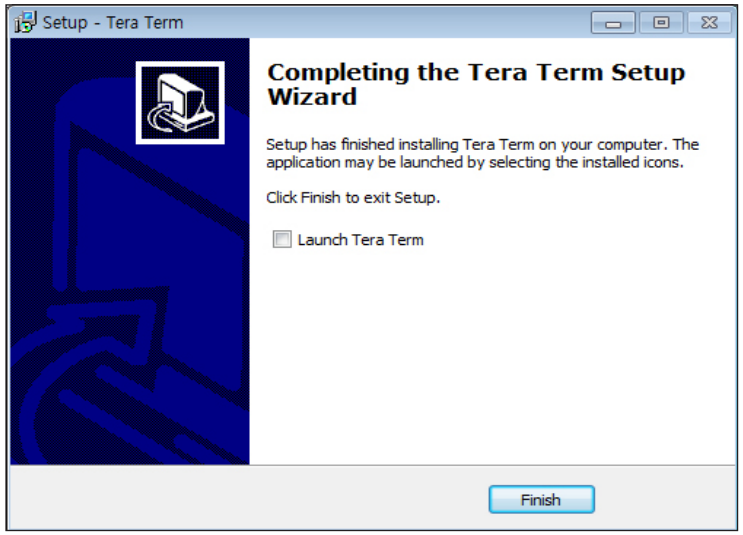
１２　インストールの開始

“Install”をクリックすると、インストールを開始します。



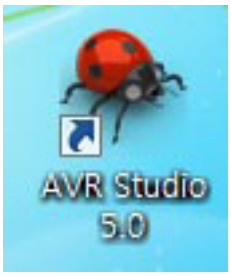
13 インストールの確認

プログラムをインストールします。



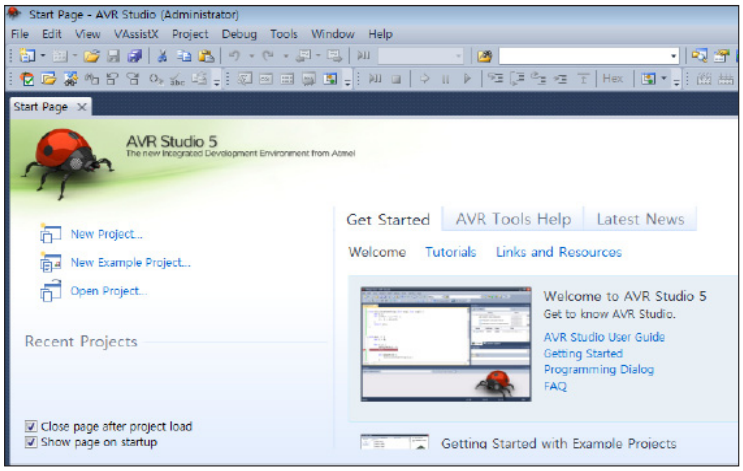
14 インストールの完了

“Finish”をクリックするとインストールが完了します。



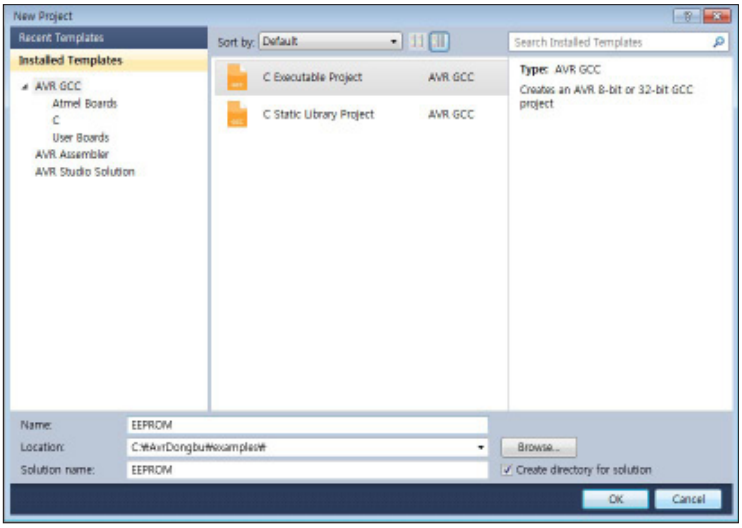
01 実行

AVRStudioをクリックして、実行します。



02 New Project

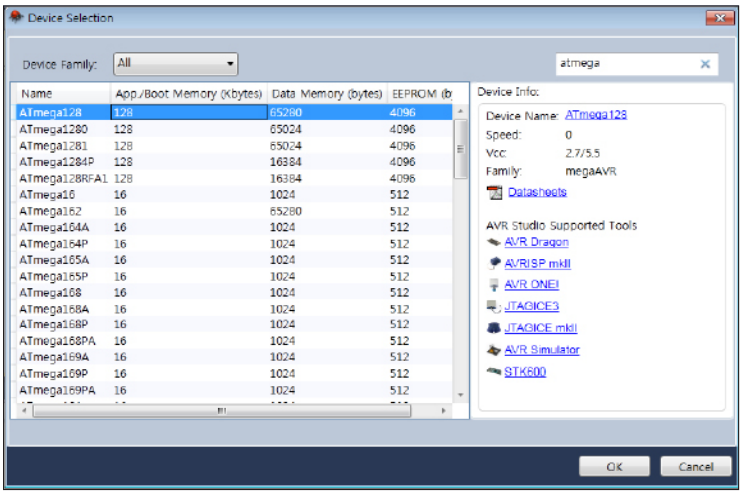
New Projectをクリックします。



03 GCC選択

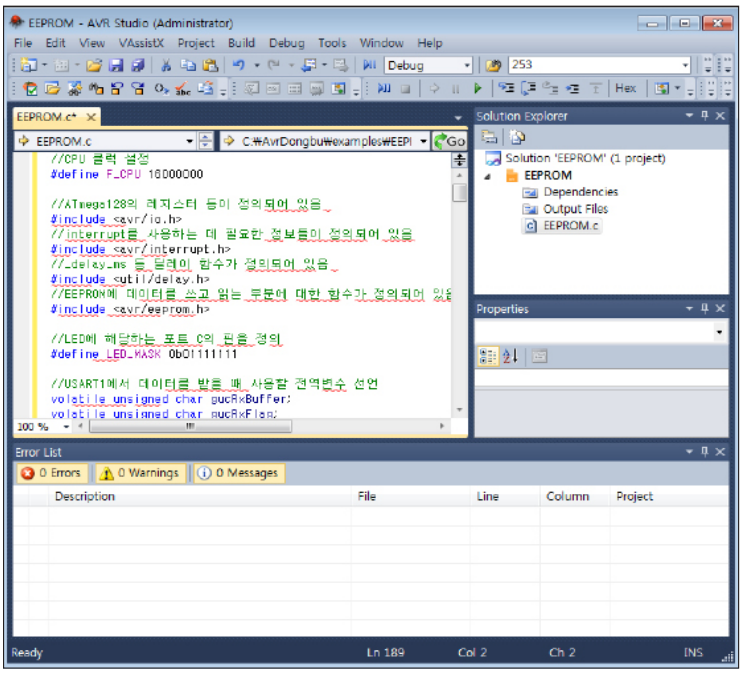
C Executable Projectを選択します。

Nameに“EEPROM”と入力し、
Create directory for solution をクリックして、OK ボタンを押します。



04 ATmega 選択

右上からatmegaを検索して、
リスト中のATmega128を選択します。



05 Editor

編集ウィンドウでコーディングを開始します。
下の例を入力します。

例

今回のChapterの例は、Chapter7の例とほぼ同じです。USB-to-SerialまたはPC背面のCOMPORTとDRCを通信ケーブルで接続し、ハイパーターミナルを利用して送信される文字に応じてDRCのLEDを制御する例を示します。ただし、違う点は、EEPROMに現在どのようなLEDが点灯しているかを保存し、電源をオフにして入れ直しても最後に点灯していたLEDが点灯することです。

ソースはChapter7のUSART.cとほぼ同じなので、USART.cから変更された部分を赤い文字で表示し、変更された部分のみを説明します。ソースコードの全体的な内容を知りたい場合は、Chapter 7の例の説明をもう一度読んでから今回のChapterを読んでください。

EEPROM.c

```
//CPU クロックを設定する
#define F_CPU 16000000

//ATmega128のレジスタが定義されている
#include <avr/io.h>
//interruptを使用するために必要な情報が定義されている
#include <avr/interrupt.h>
//_delay_ms などディレイ関数が定義されている
#include <util/delay.h>
//EEPROMにデータを読み書きする部分の関数が定義されている
#include <avr/eeprom.h>

//LEDに対応するポートCのピン定義
#define LED_MASK 0b01111111

//USART1からデータを受信するときに使用するグローバル変数の宣言
volatile unsigned char gucRxBuffer;
volatile unsigned char gucRxFlag;

/**
* USART1 受信完了割り込みのサービス･ルーチンです。
* UDR1からデータを読んでgucRxBufferに保存して
* gucRxFlagを1に設定します。
*
* @author       Dongbu Robot
* @param        なし
* @return       なし
* @version      1.0.0.0
*/
ISR(USART1_RX_vect){
        gucRxBuffer = UDR1;
        gucRxFlag = 1;
}

/**
* USART1の初期化関数である。USART1を有効にして
* ulBaudRateに応じてボーレートを設定した後、
*グローバル変数を初期化します。
*
* @author       Dongbu Robot
* @param        ulBaudRate: USART1を動作させるボーレート
* @return       なし
* @version      1.0.0.0
*/
```

```
void USART1_Init(unsigned long ulBaudRate){
        unsigned long ulTemp;

        //ブートローダーとの衝突を防ぐためにしばらく待機
        _delay_ms(10);

        //送信、受信、受信割り込みを有効に
        UCSR1B = ((1 << TXEN1) | (1 << RXEN1) | (1 << RXCIE1));
        //2倍速転送モードの設定
        UCSR1A = (1 << U2X1);

        //クロックとBaud Rateを使ってUBRR1に入る値の計算
        ulTemp = F_CPU / (ulBaudRate * 8L) - 1;

        //UBRR1に計算した値を設定
        UBRR1H = (ulTemp >> 8) & 0x0F;
        UBRR1L = ulTemp & 0xFF;

        //グローバル変数の初期化
        gucRxBuffer = 0;
        gucRxFlag = 0;

        return;
}

/**
* USART1に文字を送信します。
*
* @author       Dongbu Robot
* @param        cData: 送信する文字データ
* @return       なし
* @version      1.0.0.0
*/
void USART1_PutChar(char cData){
        //送信データレジスタが空になるまで待機
        while(!(UCSR1A & (1 << UDRE1)));
        //送信データレジスタにデータを入力
        UDR1 = cData;

        return;
}

/**
```

```
* USART1文字列を送信します。
*
* @author		Dongbu Robot
* @param		pcStr: 送信する文字列の最初の文字のアドレス
* @return		なし
* @version		1.0.0.0
*/
void USART1_PutString(char* pcStr){
	//'¥0'に会うまでUSART1_PutCharを繰り返す
	while(*pcStr)
		USART1_PutChar(*pcStr++);
}

int main(void)
{
	//8ビット変数の宣言
	unsigned char ucLED;

	//ポートCの入出力方向を設定
	DDRC = 0b01111111;
	//ポートCの出力値を設定
	PORTC = 0b01111111;

	//全体の割り込みを無効に
	cli();

	//USART1初期化
	USART1_Init(115200);

	//全体の割り込みを有効に
	sei();

	//初期化完了メッセージ送信
	USART1_PutString("Serial Initialized¥r¥n");

	 //EEPROMの0番アドレスからデータを読み込む
	ucLED = eeprom_read_byte((uint8_t*)0);

	//データが必要なものでなければ0b00000001に初期化
	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;
}

//ポートCのLEDをオン
PORTC = (~ucLED) & LED_MASK;

//while 内の内容を無限に繰り返す
while(1)
{
        //データを受け取ると実行
        if(gucRxFlag){
                //受信したデータが' z' や 'Z' であると実行
                if(gucRxBuffer == 'z' || gucRxBuffer == 'Z' ){
                        //オンにするLEDのビットを右に移動した後、EEPROMにデータを保存
                        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;

                        //メッセージの送信
                        USART1_PutString("LED Moved by ¥' z¥' ¥r¥n");
                }
                //受信したデータが"x"や "X"であると実行
                else if(gucRxBuffer == 'x' || gucRxBuffer == 'X' ){
                        //オンにするLEDのビットを左に移動した後、EEPROMにデータを保存
                        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;

                                        //メッセージの送信
                                        USART1_PutString("LED Moved by ¥' x¥' ¥r¥n");
                                }
                                //データ受信フラグの初期化
                                gucRxFlag = 0;
                        }
                }

                //関数の形に合わせて整数値を返す
                return 1;
        }
```

例の説明

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

前に説明したように、avr/ eeprom.hヘッダファイルには、EEPROMからデータを読み書きするための関数が定義されています。このヘッダで宣言されている主な関数は以下のとおりです。

```
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);
```

上の関数は、EEPROMの_ pというアドレスで、それぞれunsigned char(8ビット)、unsigned int(16ビット)、unsigned long int(32ビット)のデータを読み込むための関数です。関数原型のuint8_t、uint16_t、uint32_tは、それぞれ8ビット、16ビット、32ビット符号なし整数型を表すもので、unsigned char、unsigned int、unsigned long intと同じです。 uint8_tなどの変数の型は、その変数がいくつのビットで構成されているのかを確実に明示したい場合に使用されます。intとlong intなどの変数の型は、システムとコンパイラによって意味する長さが異なる可能性があるからです。

上の3つの関数のパラメータには、読みたいEEPROMのアドレス値、つまり、ポインタが入り、そのアドレスのデータ値がリターンされ戻ります。

```
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);
```

上の関数は、EEPROMの_ pというアドレスにunsigned char(8ビット)、unsigned int(16ビット)、unsigned long int(32ビット)のデータ_ valueを保存するための関数です。パラメータには、データを書き込み、EEPROMのアドレス値、つまりポインタが入り、その次に書き込まれるデータが入ります。

EEPROMを読み書きする関数がどのように使用されるかは、以下の実際の例で表示されます。

```
ucLED = eeprom_read_byte((uint8_t*)0);
```

eeprom_read_byte関数を使用して、実際にeepromのデータを読み取る部分です。パラメータに0を使用したのは、EEPROMの0番アドレスからunsigned charデータを読み取るためであり、(uint8_t*)に0を使用したのは0という値のタイプを、関数原型のデータ型に合わせて変換させる部分だからです。このコードを経れば、変数ucLEDにEEPROM0番アドレスのデータ値が保存されます。

```
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;
}
```

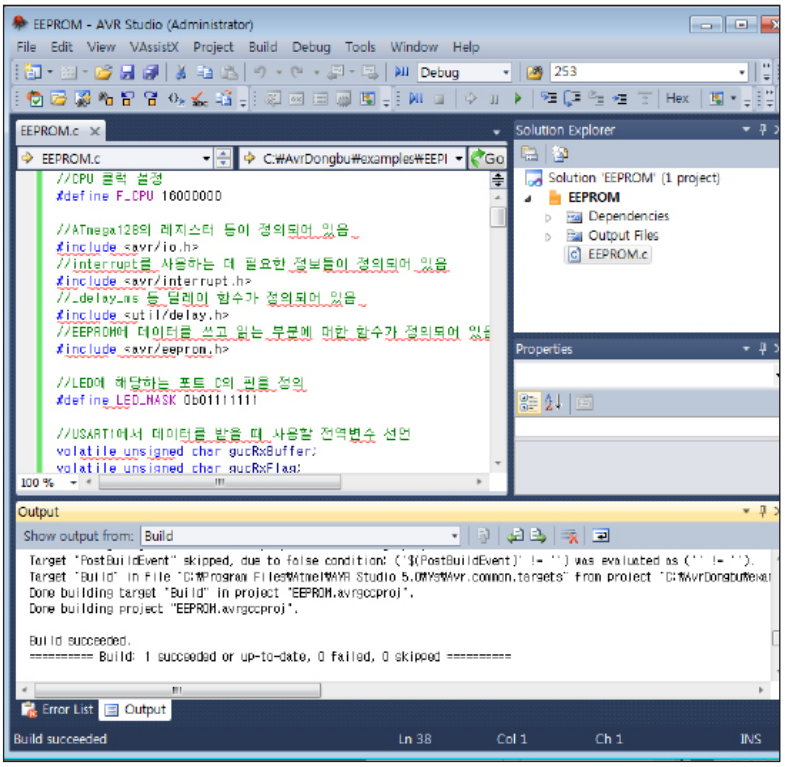
EEPROMから読み取ったデータを検査する部分です。ボードに初めて電源を入れた状況、またはEEPROMに他のプログラムやノイズによって意図しない値が入った場合には、ucLEDに予期しない値が保存されることがあります。これをそのままにしておくと、プログラムが誤動作を起こす可能性があります。そのためswitch-case文を使用してucLED値が正常のカテゴリ(0b01000000～0b00000001)内に入ると値を変えずにスキップし、その中に入ってなければucLEDを0b00000001に初期化してeeprom_write_byte関数を使用してEEPROM0番アドレスにucLED値を保存します。

```
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);
}
```

この部分はユーザーからのキーボード入力に応じて、特定の文字が入るとucLED値を変えて点灯するLEDの位置を変更する部分です。この部分でucLED値が変わる部分の下にすべてeeprom_write_byte((uint8_t*)0、ucLED);が追加されて、ucLED値が変わるたびにEEPROMの0番アドレスに現在のucLED値を保存するように変更しました。この部分が追加されたことで電源を切って入れると、無条件にServo　LEDから開始したこの前の例とは異なり、今回の例では、オフになる前に点灯していたLEDが再び点灯し、その位置から再起動します。

実行

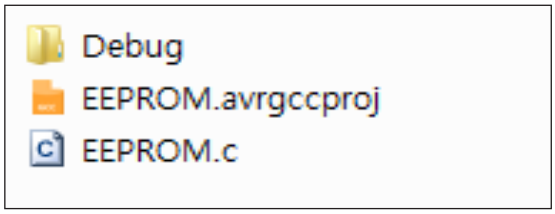
Tera Termを利用してファームウェアを実行します。Z、Xキーを入力すると、点灯したLEDが移動しながら、ターミナルウィンドウにメッセージが出力されます。この時、電源を切って入れ直しても最後の位置から再開します。



01 Build

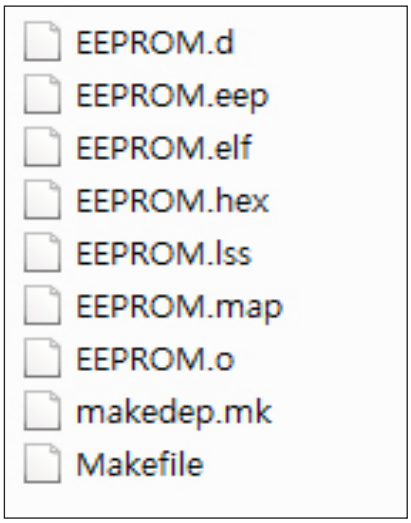
Buildメニューを選択しBuild Solutionをクリックしてビルドします。

1 succeeded0 failedが出るとエラーなしでコンパイルが完了したということです。



02 Debug フォルダ

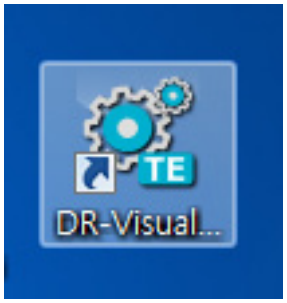
Build後にソースがあるフォルダに入るとDebugフォルダが作成されています。クリックしてフォルダの中身を見ます。



03 hex 確認

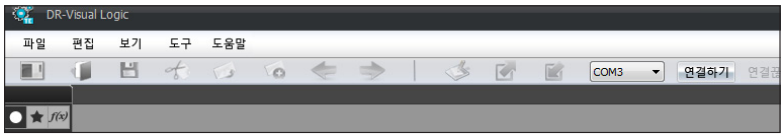
Debugフォルダの中のhexファイルを確認します。Hexファイルは制御機に移動するファイルです。

DR-Visual Logicでファームウェアの修復を行います。



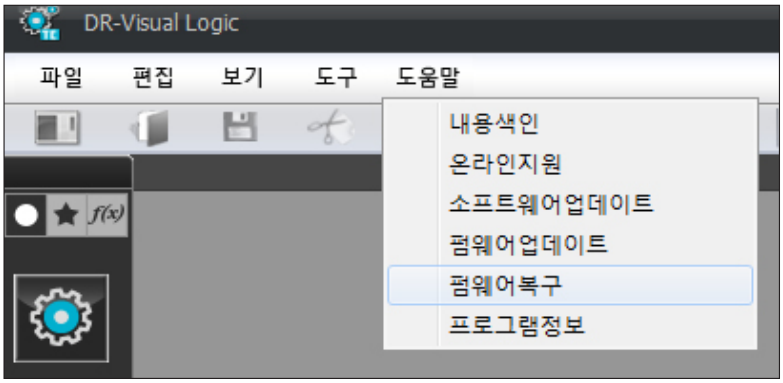
04 Visual Logic

DR-Visual Logicを実行します。



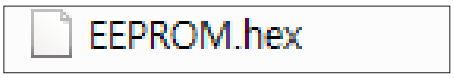
05 接続する

COM番号を設定して接続するをクリックします。



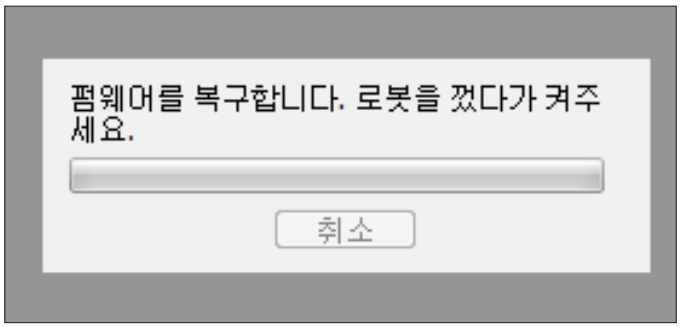
06 ファームウェアの修復

[ヘルプ]メニューからファームウェア修復をクリックします。



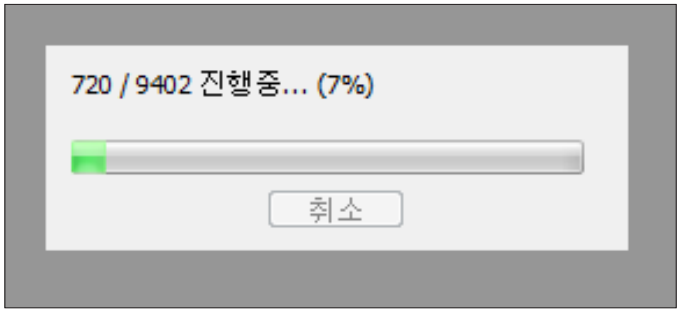
07 ロード

Debugフォルダ にHexファイルをロードします。



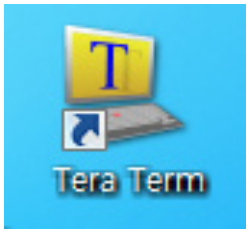
08 ファームウェアの修復

“ロボットの電源を切って入れ直してください”というメッセージが出ます。制御機の電源ボタンを一度切ってから再度入れ直してください。



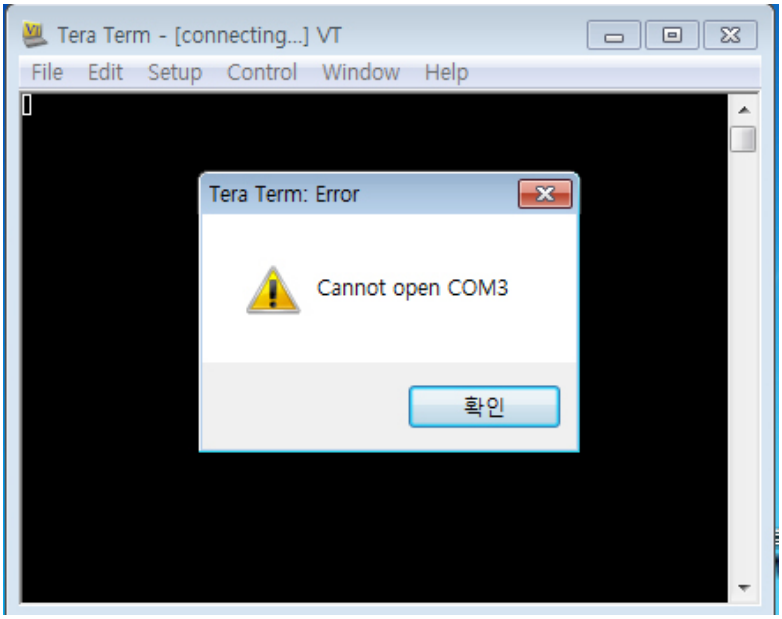
09 修復の開始

ファームウェアの修復が開始されます。



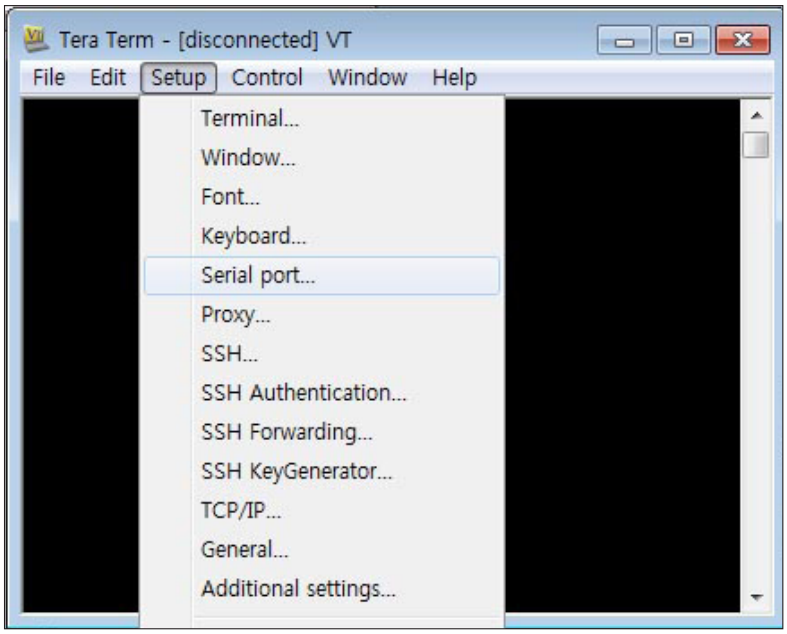
10 実行

Tera Termのアイコンをクリックしてプログラムを実行します。



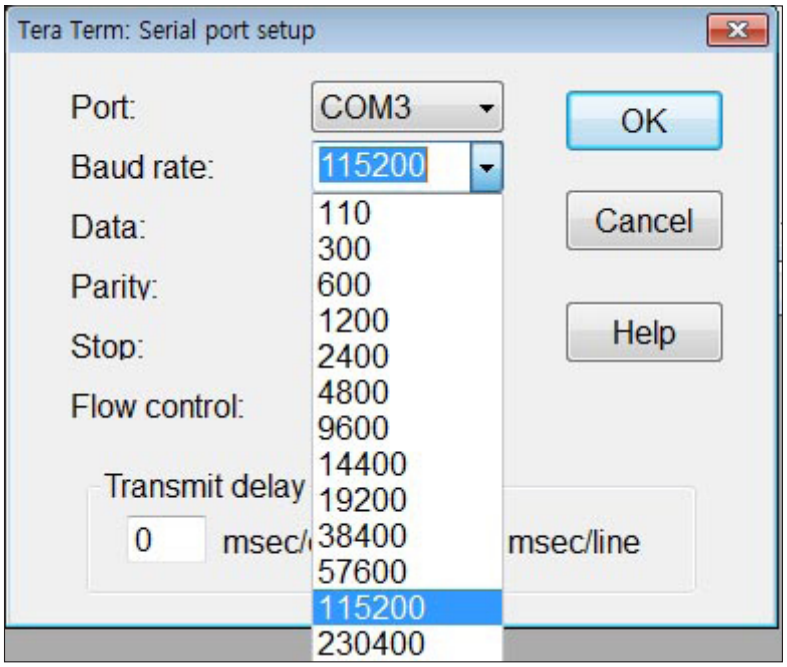
11 開く

実行すると、すぐには接続されません。　[OK]ボタンをクリックします。



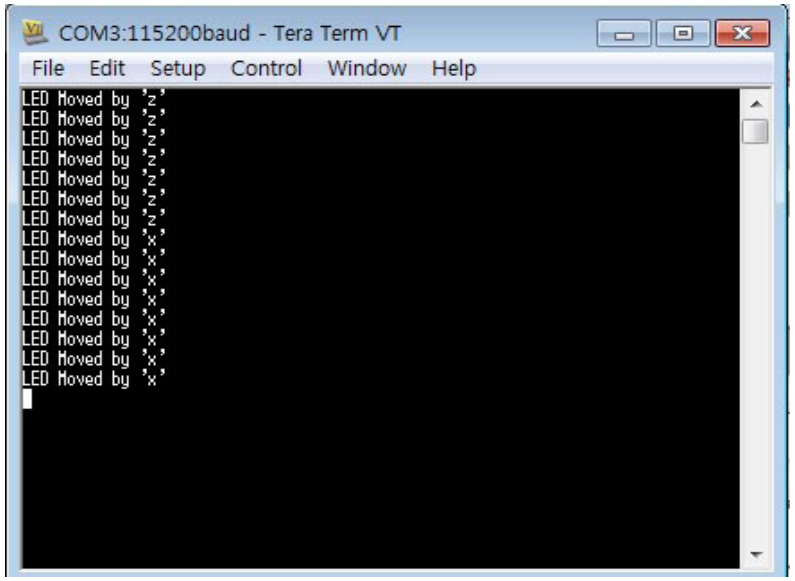
12 Setup

メニューの中のSetupメニューをクリックしてSerial portを選択します。



13 Baud rate

Baud rateは115200を選択します。制御機の最大通信速度です。



14 実行

ファームウェアを実行してTera Term（ハイパーターミナル）からz、xキーを入力すると、オンになったLEDが移動しながら、ターミナルウィンドウにメッセージが出力されることを確認できます。

加速度/ジャイロセンサを活用する

DRCに加速度/ジャイロを取り付けて、センサの値を読み取り、傾き、角速度の感知などに活用します。

AVR 原理

今回のChapterでは、ATmega128のGPIOピンを使用してI2C通信により、加速度/ジャイロセンサの値を読み取ります。I2C通信に関連した内容は、すべてlibaccgyro.aライブラリによって提供されるので、ユーザーはセンサの値を読み取って活用するだけです。

H/W 原理

I2C 通信

ATmega128のGPIOポートは、LEDのオン、ボタンの読み取りなどの簡単な入出力操作にも使用できますが、GPIOポートを使った通信も可能です。これらの通信の一つがI2C通信です。I2Cはフィリップスが開発した直列通信プロトコルであり、アイ-トゥ-シイまたはアイ-スクエア-シイと読みます。I2C通信は直列データ(SDA)と直列クロック(SCL)の2つのラインを利用して通信をして、通信電圧は最大5V、一般的には3.3Vです。DRCは、3.3Vの電圧でI2C通信をします。

I2C通信は、マスターとスレーブに分かれて行われ、通常命令を下し値を読み書きする方がマスター、その命令に応じて要求された値を返したり受け取った値を保存する方がスレーブです。ハードウェア構成から考えればDRCのATmega128チップがマスター、加速度/ジャイロモジュールの加速度センサチップとジャイロセンサチップがスレーブです。

SCLピンはマスターを制御し、0と1を繰り返しながらクロック信号をスレーブに下げます。それとともにクロックに合わせてSDAピンを決められた約束に沿ってマスターとスレーブのどちらが、入力、出力になるのかを決定しデータを送受信します。USARTではTXとRXピンは別々の役割を担っていますが、I2CはTXとRXを一つのピンで送受信を行います。

I2C通信のスレーブは、それぞれ7ビット長のスレーブアドレスを持ち、通信が行われる際にスレーブアドレスが異なる場合、応答しません。そのため、一つのマスターに互いのアドレスが異なる複数のスレーブを接続して、同じラインで通信することが可能であり、I2Cは一種のバス(bus)とも呼ばれています。実際にDongbu Robotの加速度/ジャイロモジュールも2つの別のアドレスが他のスレーブと同じラインに接続しています。7ビット長のスレーブアドレスと128個のアドレスが使用可能です。この中の16個は、特定の用途に予約されており、実際には112個のアドレスを使用できます。

I2Cを使用するチップは、スレーブアドレスの生産時に決まっています。ユーザーはアドレスを任意に変えることができません。しかし、同時に使用するチップのスレーブアドレスが重なった場合は、この両方のチップは仕方なく他のI2Cラインに接続されるます。このような状況を回避するために、I2C通信をサポートするほとんどのチップは、入力ピンを置くことによって、そのピンに入る入力が0か1かによってI2Cのスレーブアドレスが変わるようになっています。通常は、7ビットのスレーブアドレスのうちの上位6ビットを定めて、最後の1ビットは入力ピンに入った電圧に応じて0か1かが決定される方式を使用するか、入力ピンの値に応じてアドレスを最初から変える場合もあります。

I2C通信には、大きく分けて4つのプロトコルがあります。HerkuleXにRAM_READ、RAM_WRITEプロトコルがあるように、I2C通信もスレーブのレジスタを読み書きする方法があります。

■ 一つのバイトを読み取り

マスター	S	AD+W		RA		DATA		P
スレーブ			ACK		ACK		ACK	

■ 複数のバイトの読み取り

マスター	S	AD+W		RA		DATA		…	DATA		P
スレーブ			ACK		ACK		ACK			ACK	

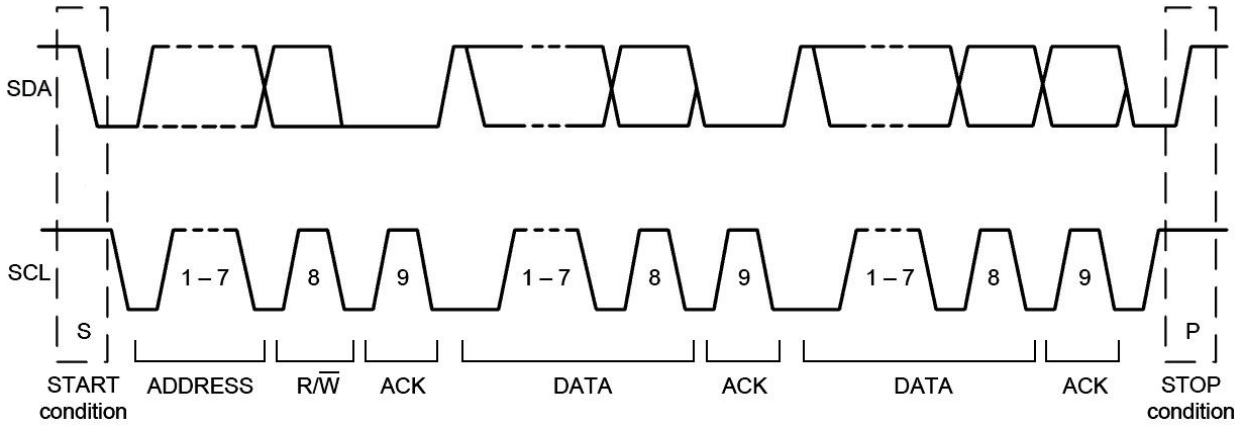
■ 一つのバイトの書き込み

マスター	S	AD+W		RA		S	AD+R			NACK	P
スレーブ			ACK		ACK			ACK	DATA		

■ 複数のバイトの書き込み

マスター	S	AD+W		RA		S	AD+R			ACK	…		NACK	P
スレーブ			ACK		ACK			ACK	DATA			DATA		

信号	説明
S	開始信号（SCLが1日の間SDAが1 - >0）
AD	I2Cスレーブアドレス（7ビット）
W	書き込みビット（0）
R	読み取りビット（1）
ACK	応答信号（9番目のクロックで、SCLが1の間SDAを0に維持）
NACK	非応答信号（9番目のクロックで、SCLが1の間SDAを1に維持）
RA	I2Cスレーブ内部のレジスタアドレス
DATA	転送されるデータ（8ビット）
P	停止信号（SCLが1日の間SDAを0 - >1）



I2C通信は開始信号を開始して停止信号で終わります。普段SCLとSDAピンは、1(HIGH)に維持されていますが、SCLが1なのにSDAが1から0(LOW)に変わったら、その時が通信の開始です。その後は8ビットを送信される方が1ビットのACKで応答することを繰り返します。SCLが1なのにSDAが0から1に変わる停止信号で終わります。 例外的な書き込み動作の場合、スレーブアドレスとレジスタアドレスを書いた後、再起動信号を送信します。この開始信号を再起動(restart)信号と呼びます。開始と停止信号以外のすべてSDAピンの値の変更は、SCLが0のとき行われます。

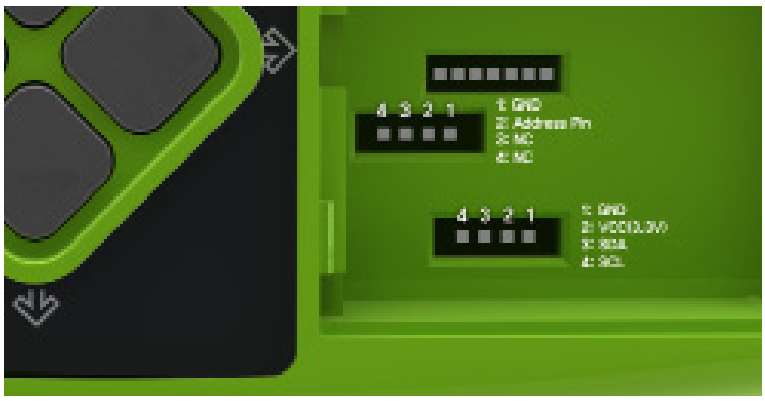
libaccgyro.aでは、細かな開始信号、停止信号、応答信号などを考える必要なく、I2C通信を介してスレーブにデータを読み書きできる関数を提供しています。

```
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);
```

2C_InitPin関数はDRCでI2C通信に使用する3つのピンを初期化する関数です。SCL、SDAが初期化されます。残りの一つは、前で説明したアドレス出力ピンです。アドレス出力ピンを、希望するチップのアドレスの入力ピンに接続してスレーブアドレスを設定するために使用できます。パラメータのucAddressPinが0の場合、出力は0(LOW)、1であれば出力が1(HIGH)になります。

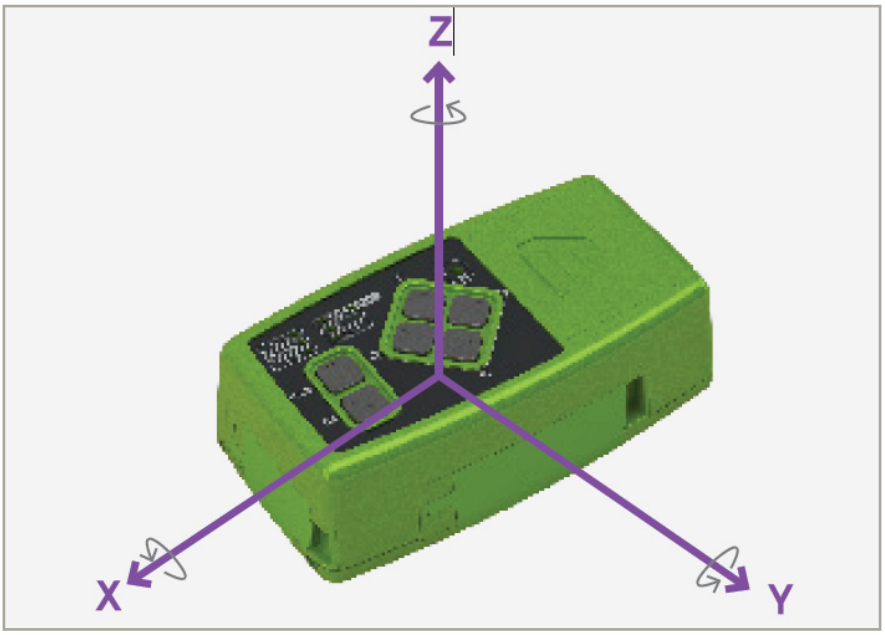
I2C_ucReadReg関数は、I2C通信でucSlaveAddrというスレーブアドレスのucRegAddrというレジスタアドレスからデータ1バイトを読み込み、ucDataというアドレスに保存する関数です。I2C_ucReadNReg関数も同じです。ucDataというアドレスからucCntだけ読み込むことが異なります。I2C_ucWriteReg関数は、I2C通信でucSlaveAddrというスレーブアドレスのucRegAddrというレジスタアドレスにucDataというデータを書き込む関数です。2C_ucWriteNReg関数も同じです。ucDataというアドレスからucCntだけのデータを使うことが異なります。上記の4関数はすべてunsigned charというリターン値を持ちますが、通信が最後まで成功したら1、失敗した場合は0が返されます。

ユーザーが加速度/ジャイロセンサモジュール以外のチップと通信をしたい場合は、ピンマップに合わせてチップとDRCを接続して、I2C_InitPin関数でピンを初期化した後、4つの通信機能を使用してチップと通信すると、どのチップにもDRCと接続して利用できます。 I2C通信に使用されるピンのピンマップは下の図のとおりです。



加速度/ジャイロセンサモジュール

Dongbu Robotの加速度/ジャイロセンサモジュールは、上の図のように4ピンのピンヘッダ、ソケット2つに装着されます。前で説明したように、センサモジュールには、加速度センサチップとジャイロセンサチップが入っていてDRCと同じラインを使用してI2C通信で接続されます。加速度/ジャイロセンサチップは、それぞれX、Y、Zの3軸に対する線加速度と角速度データを提供します。このセンサのデータを使用してDRCやロボットがどのような傾きで置かれているのか、どのような速度で回転しているかを知ることができます。



加速度センサでは±16g(1gは重力加速度、つまり9.8m/s2)の加速度を13bit(-4096～4095)の値として読み取ります。そのため1gに相当する値は256です。また、ジャイロセンサでは±2000dps(degree per second、° / s)の角速度を16bit(-32768～32767)の値として読み取ります。たとえば、毎秒90度回転する速度に相当する値は約1475です。

DRCが机の上にボタン部分を上にして置かれている時に、重力によって、Z軸の加速度値は1gを示し、Z軸加速度の値は、約256です。その状態でDRCの片側を持ち上げてDRCをΘ° 程度傾けると、Z軸加速度の値は256にcosΘ° が乗算した値です。たとえば45° 傾けると、Z軸加速度値は、256×√2÷2=181という値です。



またDRCを机の上に置いて回転させると、Z軸回転軸とする回転角速度が出てきます。ジャイロセンサは回転角速度を感知してデータに変換します。反時計方向に回転させると、ジャイロセンサでは、プラス(+)のZ軸角速度値が出て、時計方向に回転させると、逆に負(－)のZ軸角速度値が出ます。



libaccgyro.aでは、前に説明した一般的なI2C通信に対する関数だけでなく、一度だけ実行すれば簡単に加速度とジャイロセンサの値を読み取ることができる関数も提供します。

```
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関数はDRCでI2C通信に使用する3つのピンを初期化して、加速度/ジャイロセンサモジュールも動作するように初期化する関数です。加速度センサとジャイロセンサが正常に通信するか確認して、最大感知範囲を設定し、変換を開始します。通信と設定が最後まで成功したら1、失敗した場合は0が返されます。

残りの関数は名前の通り、加速度センサとジャイロセンサの特定の軸の値を読み取り、XYZ軸の値を一度に読み取る関数です。 iDataに渡したポインタ変数のアドレスに値が保存され、読み取りに成功した場合1、失敗した場合は0が返されます。例では、上記の関数を使用して加速度/ジャイロセンサモジュールの値を読み取ります。

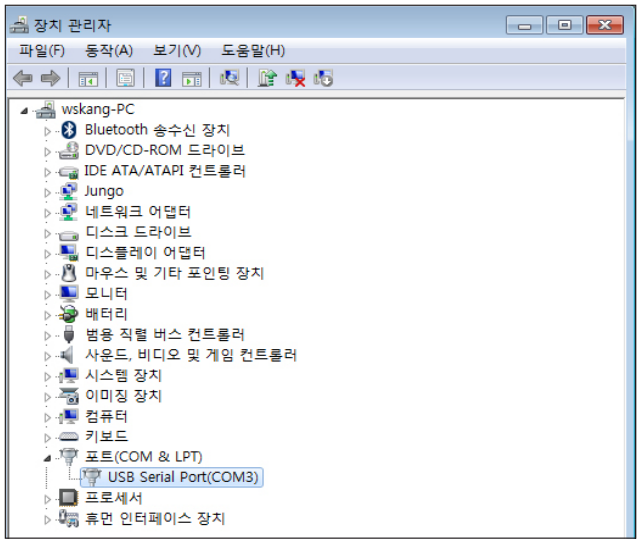
準備

ハードウェアを準備する



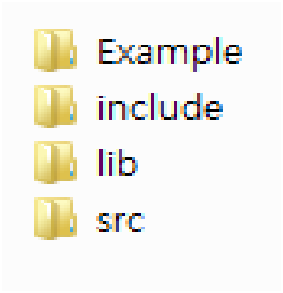
01 接続する

DRC制御機のアダプターで電源を接続します。USB to Serial CableでPCとDRCを接続します。
加速度/ジャイロモジュールを装着します。



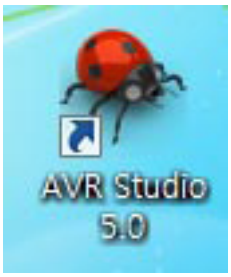
02 COM portを接続する

[マイコンピュータ]>[プロパティ]>[デバイスマネージャ]でCOM接続を確認します。



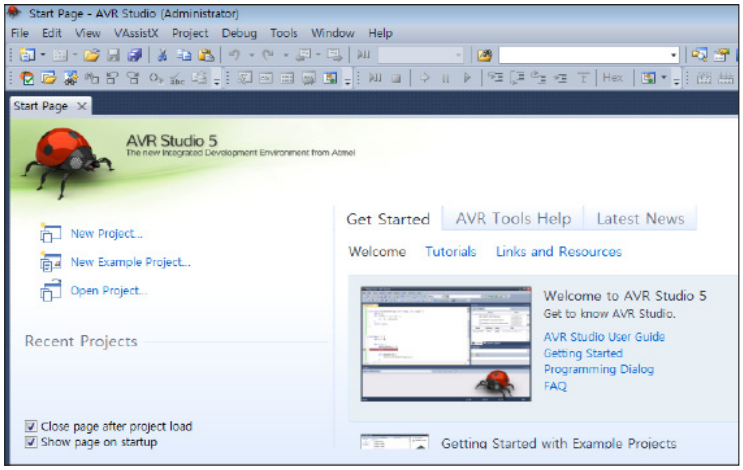
00 ライブラリ

加速度/ジャイロモジュールを制御するためには、基本ライブラリファイルが必要です。
ライブラリが含まれているフォルダ構造です。



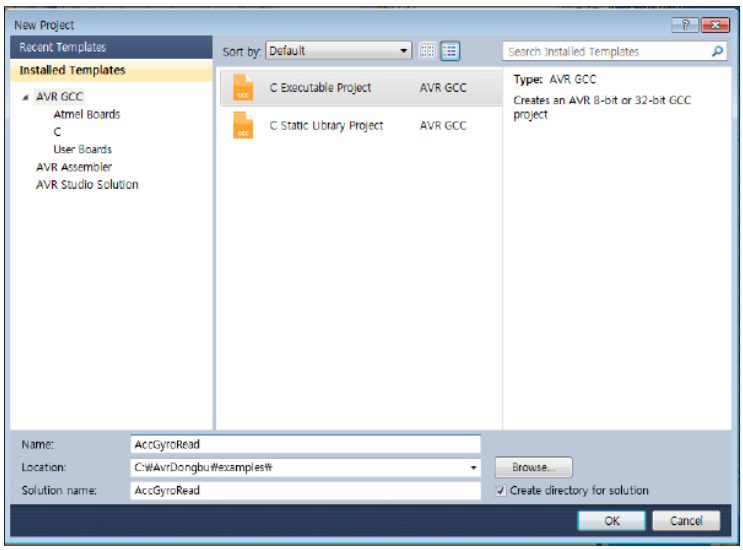
01 実行

AVRStudioをクリックして、実行します。



02 New Project

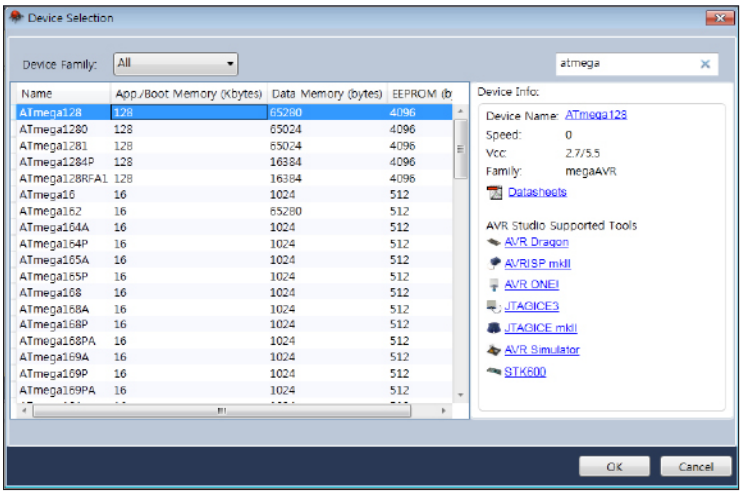
New Projectをクリックします。



03 GCC選択

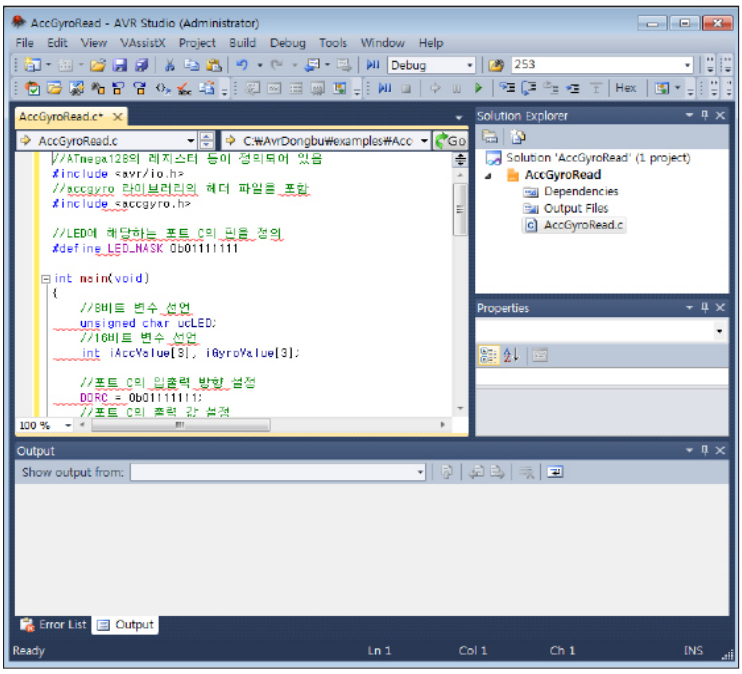
C Executable Projectを選択します。

Nameに“AccGyroRead”と入力し、Create directory for solution をクリックして、OK ボタンを押します。



04 ATmega 選択

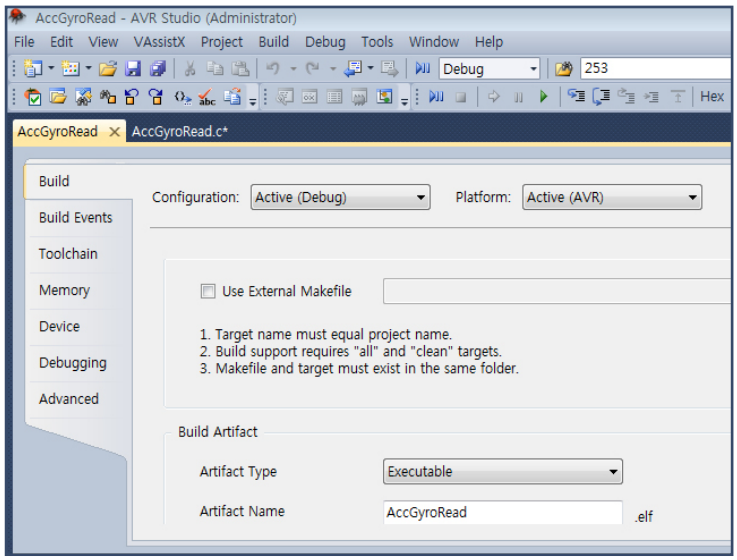
右上からatmegaを検索して、
リスト中のATmega128を選択します。



05 Editor

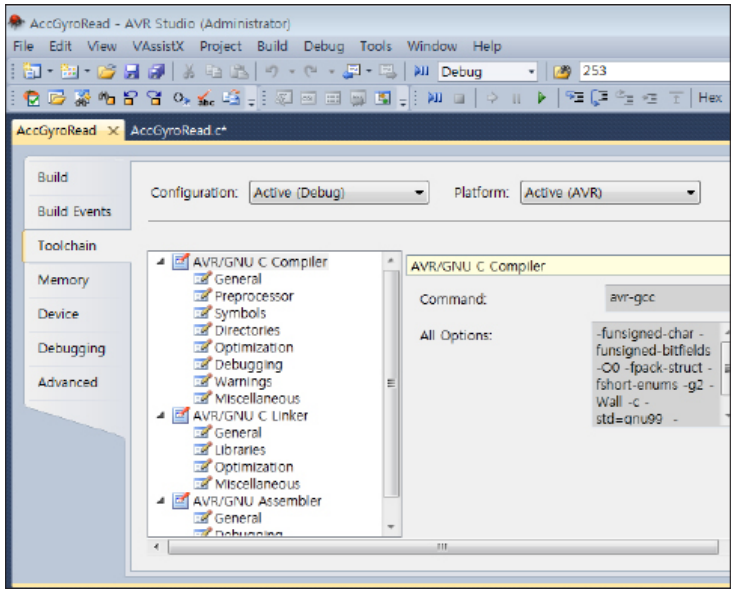
編集ウィンドウでコーディングを開始します。

ライブラリのリンクと設定



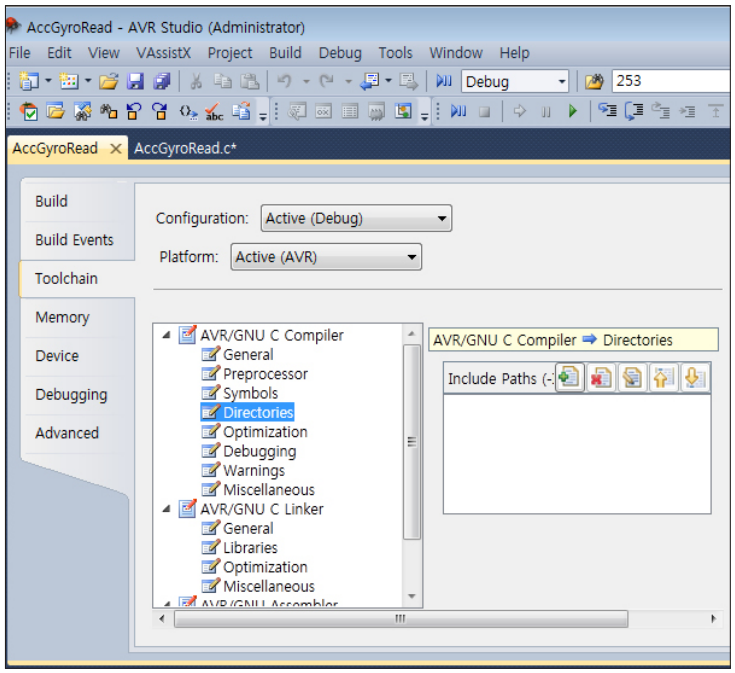
01 設定

AVR Studioの上部メニューの中のProjectをクリックすると、AccGyroRead（プロジェクト名により異なります）Propertyメニューが生成されています。クリックすると、左側のように設定画面が出ます。



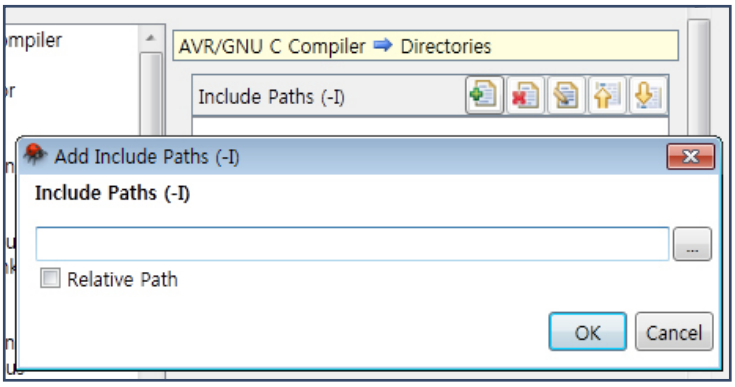
02 Toolchain

左側のタブボタンの中のToolchainをクリックして、AVR/ GNU C Compilerを選択します。



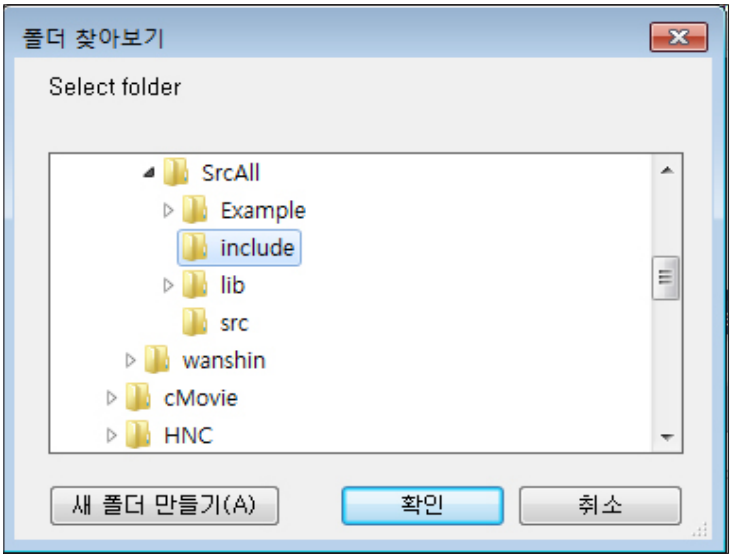
03 C Compiler

C Compiler フォルダ中のDirectoriesを選択します。



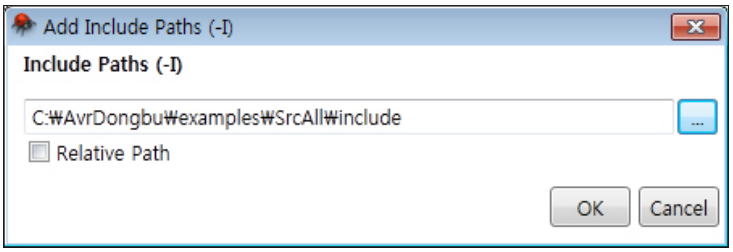
04 Path

Include Pathsの右にある+ボタンをクリックすると、ポップアップ入力画面が表示されます。
Relative Path（相対パス）のチェックを解除して（5．1ではしなくても良い）、右側のフォルダの検索ボタンをクリックします。



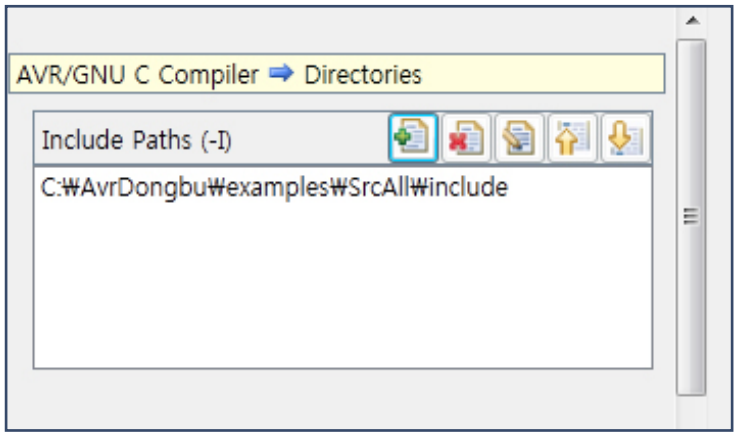
05 Include

提供ライブラリフォルダ中のincludeを選択します。



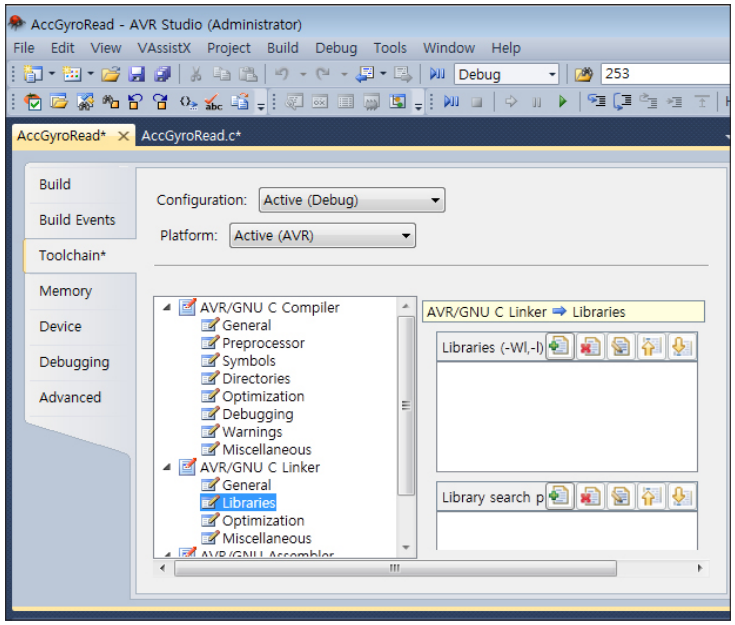
06 Include 完了

Includeフォルダの場所が指定されます。
OKボタンをクリックします。



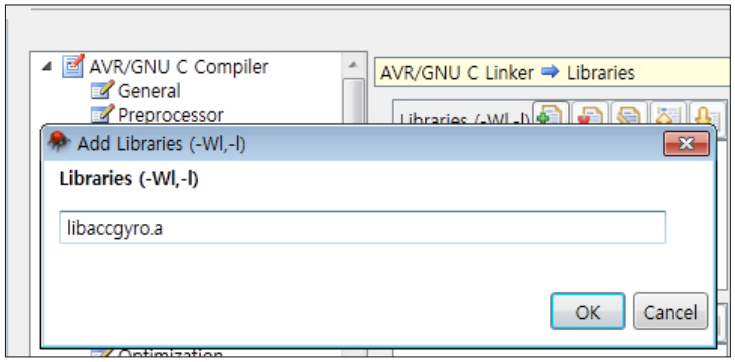
07 Path 完了

Includeパスの指定が完了しました。



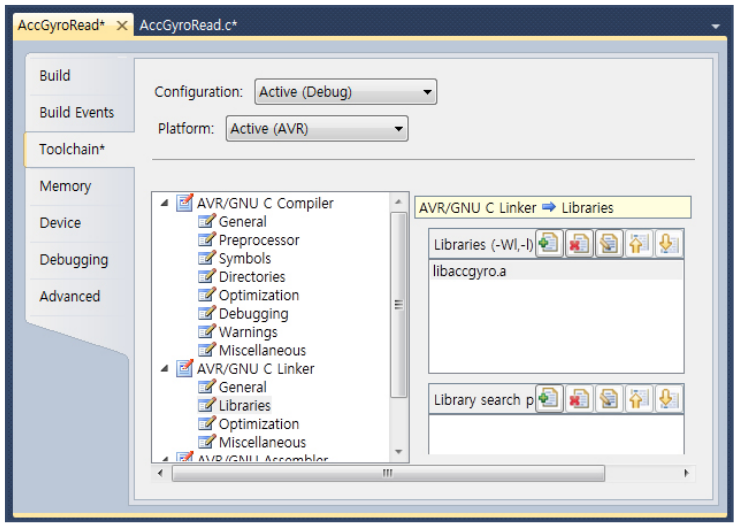
08 C Linker

C Linker フォルダ中のLibrariesを選択します。



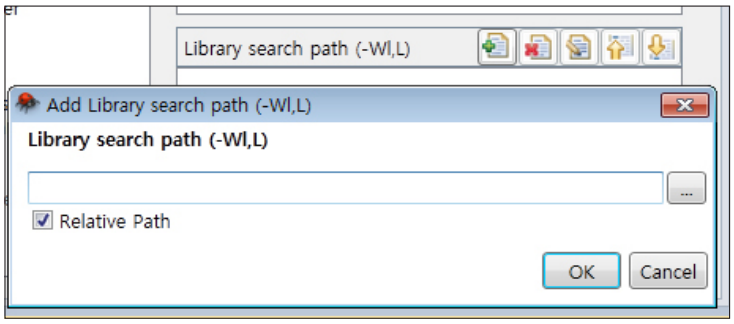
09 Libraries

Librariesウィンドウ右側にある+ボタンをクリックすると、Librariesファイル名入力画面が表示されます。libherkulex.aを直接入力します。
OKボタンをクリックします。



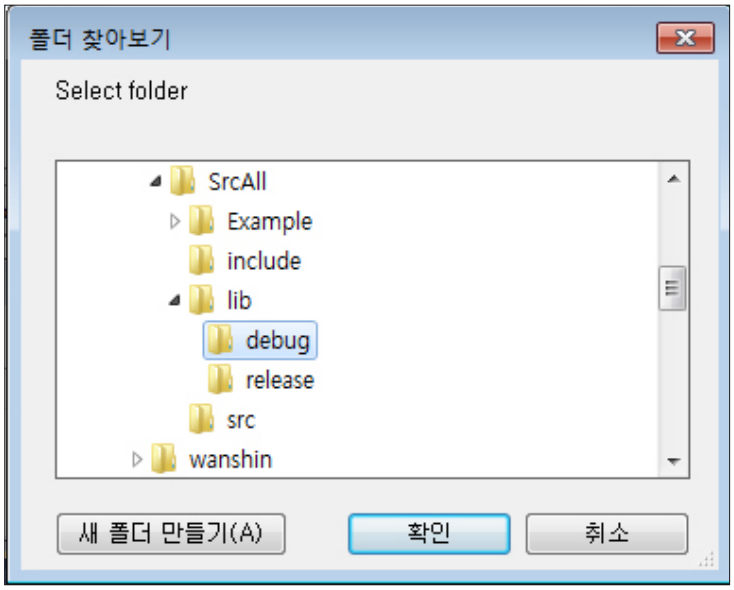
10 Search

加速度/ジャイロモジュールのライブラリファイルlibaccgyro.aリンクが完了しました。
次に、Library search pathを設定します。
右の+ボタンをクリックします。



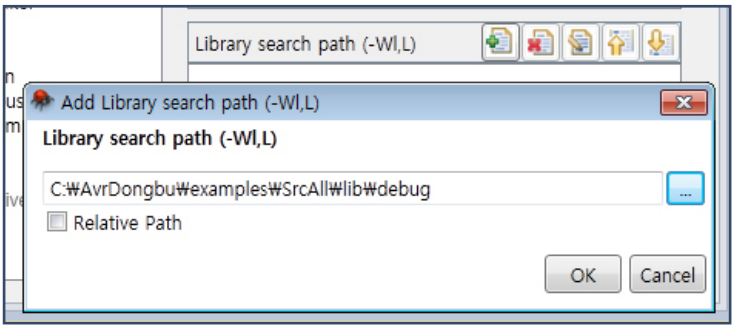
11 Path

Library search pathのポップアップ右側にあるフォルダの検索ボタンをクリックします。



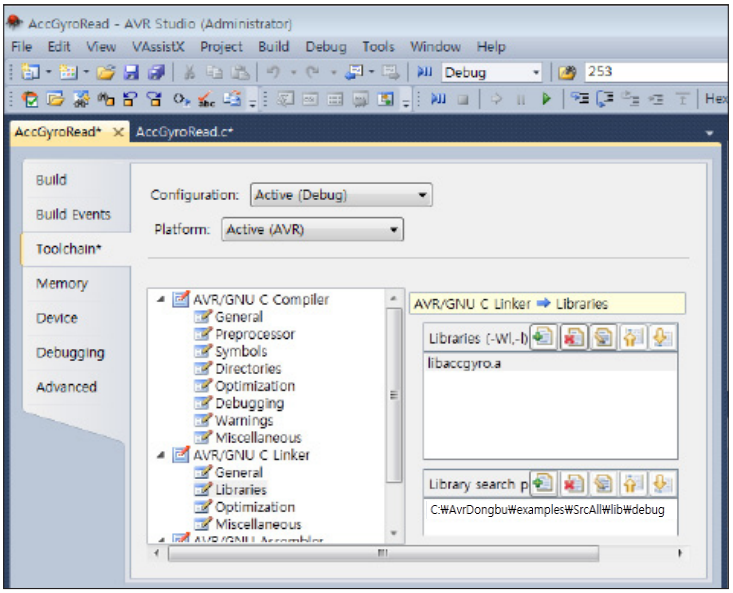
12 Debug

提供ライブラリフォルダのlibサブフォルダにあるdebugを選択します。



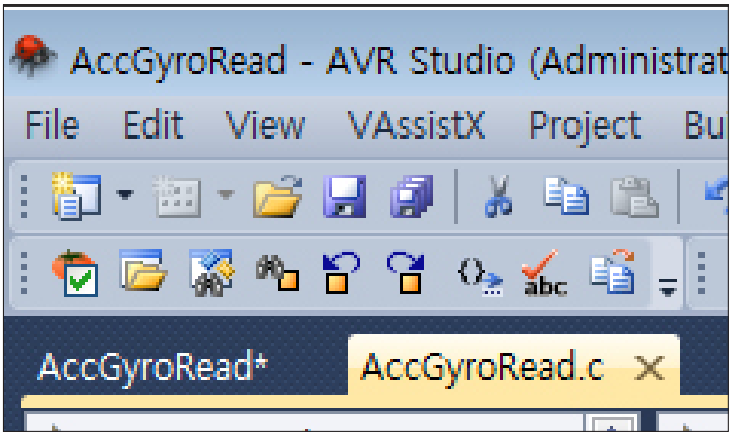
１３ Path 完了

Library search pathのパスの指定が完了しました。
OKボタンをクリックします。



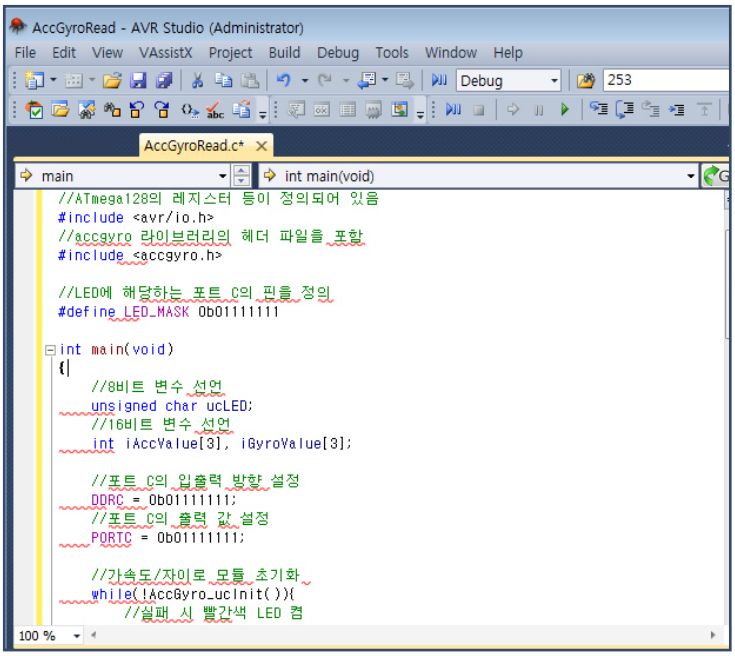
１４ Link 完了

C Linker ライブラリファイルとパスの指定が完了しました。



１５ 保存

保存ボタンを押すと現在の設定値が保存されます。タブの名前が AccGyroRead * -> AccGyroRead に変更されます。



16 コーディング

AccGyroRead.c　のタブを選択して、下の例を入力します。

例

DRCに装着された加速度/ジャイロセンサを初期化して加速度センサとジャイロセンサの値を持続的に読み取りながら、DRCが45度以上に傾いた場合、緑色のLEDを点灯し、DRCが90dps以上回転すると、青色のLEDを点灯する例です。

accgyro.h

```
#ifndef ACCGYRO_H_
#define ACCGYRO_H_

#ifdef _ACCGYRO_C
        #define ACCGYRO_EXT
#else
        #define ACCGYRO_EXT extern
#endif

// I2C通信のためにSCL、SDAピンを初期化してアドレス出力値をucAddressPinに初期化する関数。
ACCGYRO_EXT void I2C_InitPin(unsigned char ucAddressPin);
// IucSlaveAddrが指定する側のucRegAddrの値をucDataに読み込む関数です。成功した時は1、失敗した時は0にリターン。
ACCGYRO_EXT unsigned char I2C_ucReadReg(unsigned char ucSlaveAddr、unsigned char ucRegAddr、unsigned char* ucData);
// IucSlaveAddrが指定する側のucRegAddrからucCnt個のデータをusDataに読み込む関数。成功した時は1、失敗した時は0にリターン。
```

```
ACCGYRO_EXT unsigned char I2C_ucReadNReg(unsigned char ucSlaveAddr, unsigned char ucRegAddr, unsigned char *ucData, unsigned char ucCnt);
// I2C通信からucSlaveAddrのucRegAddrにucDataという値を使う関数です。成功した時は1、失敗した時は0にリターン。
ACCGYRO_EXT unsigned char I2C_ucWriteReg(unsigned char ucSlaveAddr、unsigned char ucRegAddr、unsigned char ucData);
// I2C通信からucSlaveAddrのucRegAddrにucDataからucCnt個の値を使う関数です。成功した時1、失敗した時は0にリターン。
ACCGYRO_EXT unsigned char I2C_ucWriteNReg(unsigned char ucSlaveAddr, unsigned char ucRegAddr, unsigned char *ucData, unsigned char ucCnt);

//加速度/ジャイロセンサモジュールを初期化する関数。成功した時は1、失敗した時は0にリターン。
ACCGYRO_EXT unsigned char AccGyro_ucInit(void);
//加速度センサのX軸の値をiDataに読み込む関数。成功した時は1、失敗した時は0にリターン。
ACCGYRO_EXT unsigned char AccGyro_ucReadAccX(int *iData);
//加速度センサのY軸の値をiDataに読み込む関数。成功した時は1、失敗した時は0にリターン。
ACCGYRO_EXT unsigned char AccGyro_ucReadAccY(int *iData);
//加速度センサのZ軸の値をiDataに読み込む関数。成功した時は1、失敗した時は0にリターン。
ACCGYRO_EXT unsigned char AccGyro_ucReadAccZ(int *iData);
//加速度センサのX、Y、Z軸の値をiData配列に読み込む関数。 成功した時は1、失敗した時は0にリターン。
ACCGYRO_EXT unsigned char AccGyro_ucReadAccXYZ(int *iData);
//ジャイロセンサのX軸の値をiDataに読み込む関数。 成功した時は1、失敗した時は0にリターン。
ACCGYRO_EXT unsigned char AccGyro_ucReadGyroX(int *iData);
//ジャイロセンサのY軸の値をiDataに読み込む関数。成功した時は1、失敗した時は0にリターン。
ACCGYRO_EXT unsigned char AccGyro_ucReadGyroY(int *iData);
//ジャイロセンサのZ軸の値をiDataに読み込む関数。成功した時は1、失敗した時は0にリターン。
ACCGYRO_EXT unsigned char AccGyro_ucReadGyroZ(int *iData);
//ジャイロセンサのX、Y、Z軸の値をiData配列に読み込む関数。成功した時は1、失敗した時は0にリターン。
ACCGYRO_EXT unsigned char AccGyro_ucReadGyroXYZ(int *iData);

#endif /* ACCGYRO_H_ */
```

AccGyro Read.c

```
// ATmega128のレジスタなどが定義されている
#include <avr/io.h>
// accgyroライブラリのヘッダファイルを含む
#include <accgyro.h>

// LEDに対応するポートCのピンを定義
#define LED_MASK 0b01111111
```

```
int main(void)
{
	//8ビット変数の宣言
	unsigned char ucLED;
	//16ビット変数の宣言
	int iAccValue[3], iGyroValue[3];

	//ポートCの入出力方向の設定
	DDRC = 0b01111111;
	//ポートCの出力値設定
	PORTC = 0b01111111;

	//加速度/ジャイロモジュールの初期化
	while(!AccGyro_ucInit()){
		//失敗した時に赤色のLEDをオン
		ucLED = 0b01001001;
		PORTC = (~ucLED) & LED_MASK;
	}

	//while 内の内容を無限に繰り返す
	while(1)
	{
		//加速度センサ、ジャイロセンサの値を読んで配列に保存
		if(AccGyro_ucReadAccXYZ(iAccValue) && AccGyro_ucReadGyroXYZ(iGyroValue)){
			ucLED = 0b00000000;

			//Z軸が45度以上傾いている場合は、緑色のLEDをオン
			if(iAccValue[2] < 180){
				ucLED |= 0b00010010;
			}

			//Z軸基準で±90dps以上回転していると、青色のLEDをオン
			if(iGyroValue[2] > 1500 || iGyroValue[2] < -1500){
				ucLED |= 0b00100100;
			}
		}
		//センサの値の読み取りに失敗した時に赤色のLEDをオン
		else{
			ucLED = 0b01001001;
		}

		//ucLED 値に対応するLEDが点灯するようにPORTCに値を代入
		PORTC = (~ucLED) & LED_MASK;
	}

	//関数の形に合わせて整数値を返す
	return 1;
}
```

例の説明

accgyro.hの内容についての説明は、上記のH/Wの原理でI2C通信、加速度/ジャイロモジュールについての説明と一緒に説明したので、省略しています。

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

avr/ io.hヘッダファイルには、DDRX、PORTXレジスタなどAVR制御のために必要なレジスタなどが定義されています。またaccgyro.hヘッダファイルには、libaccgyro.aライブラリファイルに含まれている関数の原型が定義されています。現在のソースコードで使用できるように#include文を使用します。

```
#define LED_MASK 0b01111111
```

ucLED変数を使用してPORTCを介してLEDを制御する時に、実際のLEDが接続された7つのピンだけ値を変えるためにLED_MASKを宣言します。

```
unsigned char ucLED;
int iAccValue[3], iGyroValue[3];
```

変数を宣言します。ucLEDは現在のLEDの状態を保存する変数であり、7つのビットが7つのLEDと対応します。0である時はLEDがオフになっていることを示し、1である時は点灯していることを示します。 iAccValue[3]とiGyroValue[3]は、符号付き2バイト整数の配列で、それぞれ加速度とジャイロセンサのX、Y、Z軸のデータを順番に保存します。

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

ポートCを設定します。 DDRCのLEDに対応する7つのビットに1を入れて、出力し、PORTCにも7つのビットに1を入れてLEDをオフにします。

```
while(!AccGyro_ucInit()){
        ucLED = 0b01001001;
        PORTC = (~ucLED) & LED_MASK;
}
```

AccGyro_ucInit関数を使用してI2C通信のためのピンと加速度/ジャイロモジュールを初期化します。成功すると、AccGyro_ucInit関数のリターン値は1なので、while文に進入せずに抜けます。失敗すると、リターン値が0なので、while文内に進入して、赤のLED(Servo、TX、Power　LED)を点灯します。 LEDが点灯した後は、AccGyro_ucInitを再実行してその結果の値に応じてwhile文を繰り返すか抜けるかが決定されます。このコードを作成すると、AccGyro_ucInitが失敗する場合はwhile文を繰り返して、AccGyro_ucInitが成功すると次に進みます。

```
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;
}
```

while文の中のif文の内容です。このif文の条件では、AccGyro_ucReadAccXYZ関数とAccGyro_ucReadGyroXYZ関数を実行して両方のリターン値が1であり、成功した場合にのみif文の中の内容を実行して、一つでも失敗した場合はelseの内容を実行します。初期化時と同じように失敗した場合には、ucLED値を赤色のLEDがすべて点灯するように変更します。

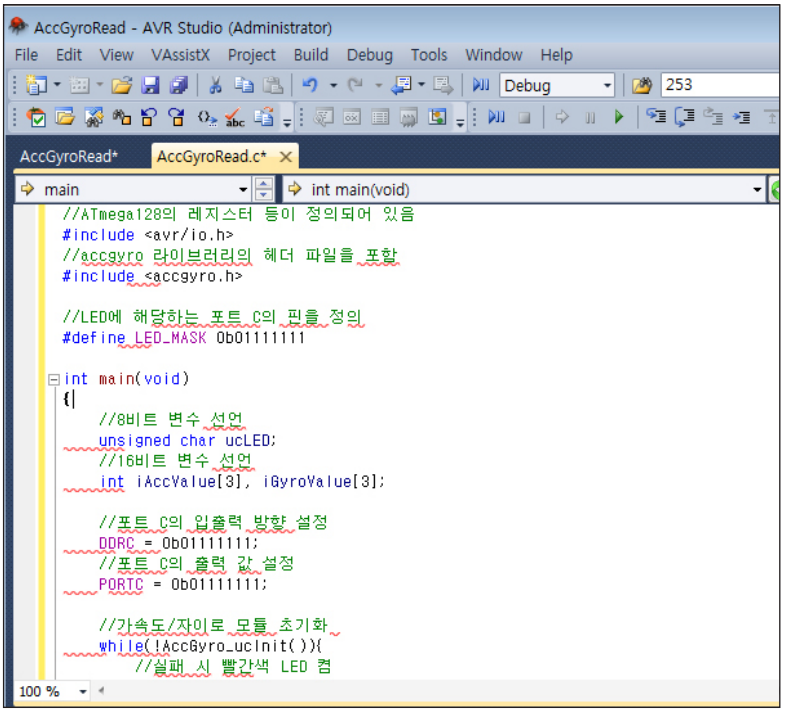
If文の中では、まずucLED値がすべてオフになるように初期化します。そしてiAccValue[2]、つまり、Z軸の加速度値が180以下であればucLEDの緑色のLED（Program、RX）がオンになるようにucLEDの対応するビットを1にします。Z軸の加速度値が180以下というのは、H/Wの原理で説明したようにDRCが約45° 以上傾いているということです。また、iGyroValue[2]が1500以上-1500未満の場合ucLEDの青色のLED（EXEC、Spare）が点灯するようにucLEDの対応するビットを1にします。iGyroValue[2]の絶対値が1500より大きいということは、H/Wの原理で説明したようにDRCがZ軸を中心に約90dps以上の速度で回転しているということです。

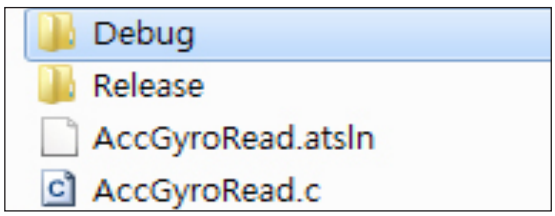
```
PORTC = (~ucLED) & LED_MASK;
```

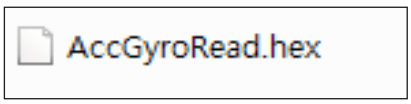
上記のif-else文で定めたucLED値を使用して実際にPORTCの値を変えてLEDを制御します。

実行

加速度/ジャイロセンサをDRCに装着した後、ファームウェアを実行すると、静止させ水平に置かれた状態では、LEDがすべてオフになっており、傾けると緑色のLEDが、回転させると、青色のLEDが点灯したことを確認できます。もし加速度/ジャイロセンサを搭載していないか、動作中に削除すると、赤いLEDが点灯します。







01 Build

Buildメニューを選択しBuild Solutionをクリックしてビルドします。

1 succeeded0 failedが出るとエラーなしでコンパイルが完了したということです。

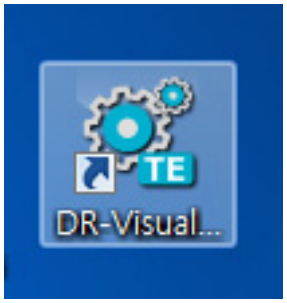
02 Debug フォルダ

Build後にソースがあるフォルダに入るとDebugフォルダが作成されています。クリックしてフォルダの中身を見ます。

03 hex 確認

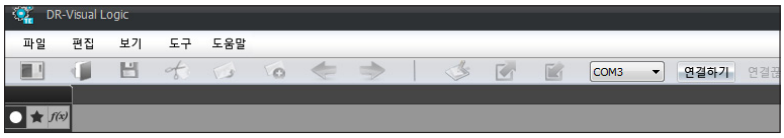
Debugフォルダの中のhexファイルを確認します。Hexファイルは制御機に移動するファイルです。

DR-Visual Logicでファームウェアの修復を行います。



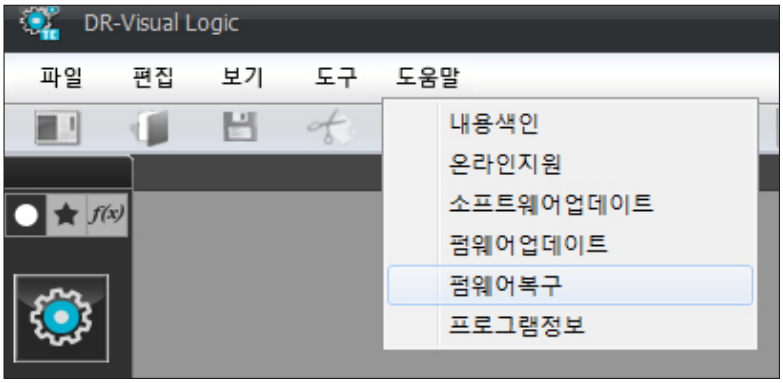
04 Visual Logic

DR-Visual Logicを実行します。



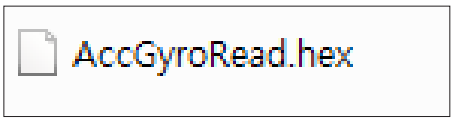
05 接続する

COM番号を設定して接続するをクリックします。



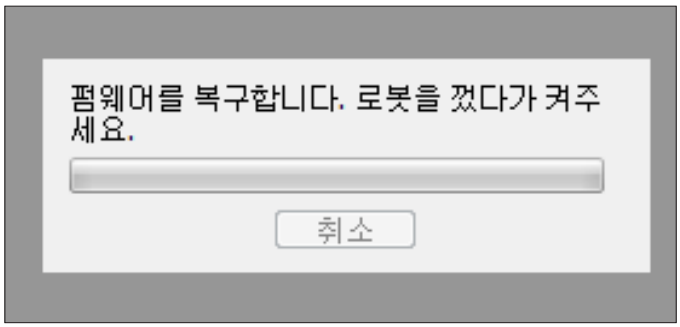
06 ファームウェアの修復

[ヘルプ]メニューからファームウェア修復をクリックします。



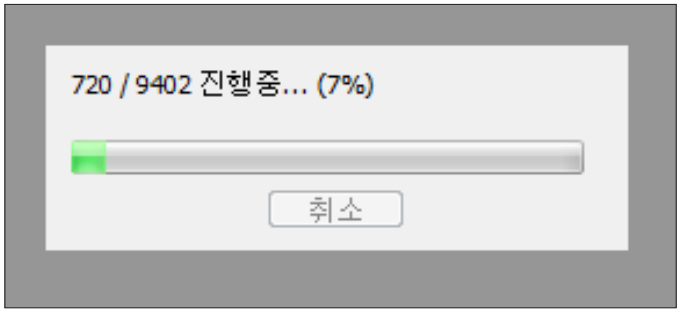
07 ロード

Debugフォルダ にHexファイルをロードします。



08 ファームウェアの修復

“ロボットの電源を切って入れ直してください”というメッセージが出ます。制御機の電源ボタンを一度切ってから再度入れ直してください。



09 修復の開始

ファームウェアの修復が開始されます。



10 実行

水平に置かれた状態では、LEDがすべてオフになっており、斜めにすると緑色のLEDが、回転させると、青色のLEDが点灯することを確認できます。

赤外線リモコン信号を読む

DRCに装着された赤外線リモコン受信モジュールでリモコンの入力を受けてDRCをリモコンで制御します。

AVRの原理

今回のChapterでは、ATmega128の外部割り込みピンを使用して赤外線リモコンの信号を受けて、リモコンのキーによってDRCが異なる動作をする例を作成します。外部割り込み機能に関連するレジスタについては、Chapter4で詳しく説明しました。Chapter4では、外部割り込み5、6番ピン(EXT5、EXT6)を使用して音入力を受けましたが、今回のChapterでは、外部割り込み4番ピンを使用して赤外線リモコン受信モジュールでリモコン信号を受信します。

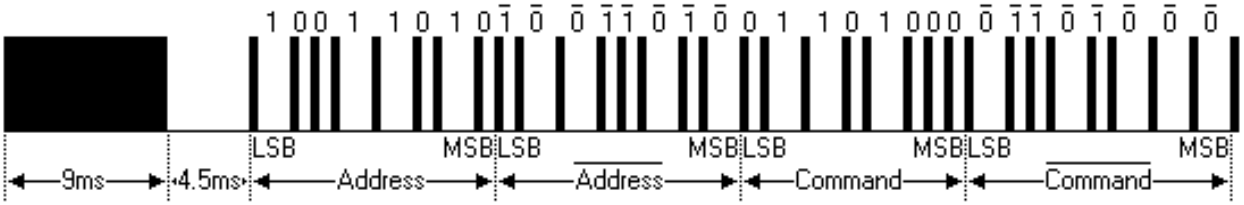
H/Wの原理

赤外線リモコンの動作原理

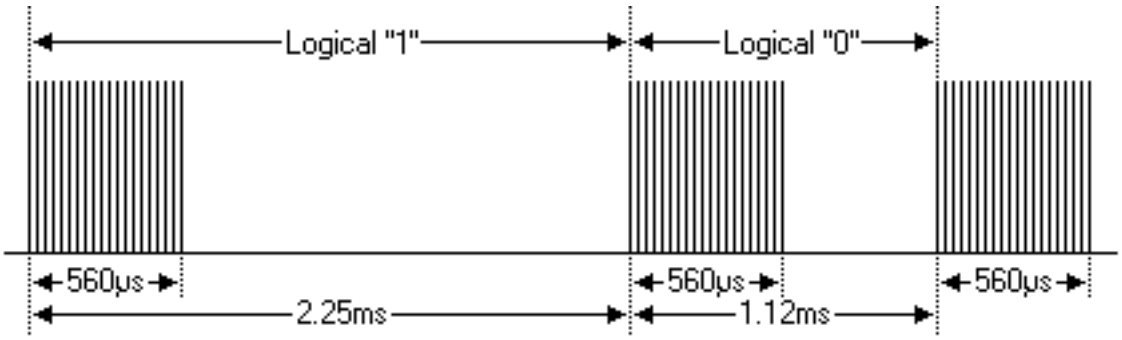
赤外線リモコンは、文字の通り赤外線を使用して信号を送ることによってリモートで他の機器を操る装置です。リモコンのキーを押すと、リモコン内部のマイクロプロセッサは、定められたプロトコルに合わせてHIGHとLOWに赤外線LEDを切るかオンにして、データコードを前の機器に放ちます。この時、光を放つのは、HIGHであると継続赤外線LEDをオンにして、LOWであるとオフにする単純な方法ではなく、38kHzの搬送波(Carrier　Wave)に変調されて出力します。信号の内容がHIGHのときに38kHzの矩形波で1と0を行き来する信号を放つという意味です。リモコン送信時に搬送波を使う理由は受信部で他の光線との干渉を防止し、ずっと送信するためです。搬送波を使うと、受信部でリモコン信号を受信した時に周波数のフィルタリングで38kHzの周波数で振動する信号を選ぶことによって、より正確に受信できます。赤外線受信モジュールでは、信号を受けてATmega128の割り込みピンにデータコードを送ります。このときは、搬送波が削除された状態でHIGHとLOWが出力されます。

リモコンのプロトコルには、D6121、TC9012、Sony、Panasonic、RC-5などの国と会社ごとに様々なプロトコルが存在します。Hovisのリモコンで使用するフォーマットは、D6121です。D6121は、NECプロトコルという名前で使用しています。

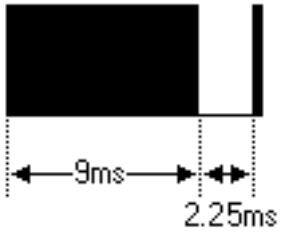
NECプロトコル(D6121)



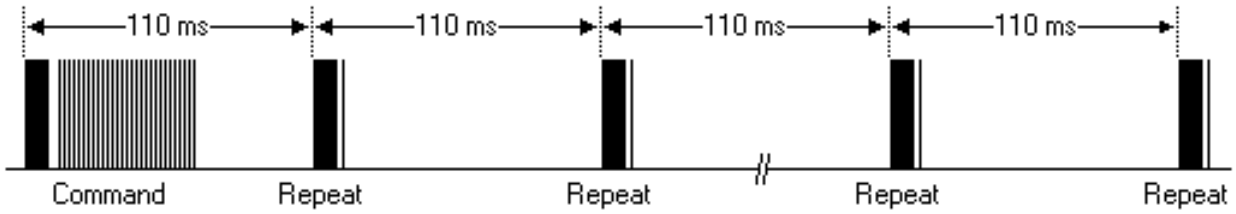
上の図は、NECプロトコルの典型的な信号を示します。リモコンの信号が開始する時は、9ミリ秒の間にHIGH信号と4.5ミリ秒の間にLOW信号を送信します。これをリーダーコード(Leader　Code)と呼びます。その後は低い番号のビット(LSB)から8ビットのAddressとCommandを送信して、それぞれを二回ずつ送信します。初めて送信する時は、元の値を送り、2番目に送信する時は、すべてのビットを裏返して送信します。上の図では、最初Addressは0b01011001であるが、その次の～Addressは0b10100110であることを示します。



AddressとCommandを送信する時に、1と0はHIGHの後に次のHIGHが出るまでの時間で区切ります。1を示している場合には、560マイクロ秒の間にHIGHが維持されて、次の信号が出るまで総2.25ミリ秒掛かります。0を表す場合には、HIGHが維持される時間は同じです。次の信号までの時間は1.12ミリ秒掛かります。



AddressとCommandは、ボタンを最初に押す時に一度だけ送信され、その後ずっと押されている時は、110ミリ秒ごとにリピートコード（Repeat　Code）が送信されます。リピートコードは上図のように9ミリ秒のHIGH信号と2.25ミリ秒間のLOW信号、560マイクロ秒間の短いHIGH信号で構成されています。リピートコードは以下の図のようにボタンがずっと押されている間、110ミリ秒ごとに繰り返されて、現在ボタンがずっと押されていることを示します。



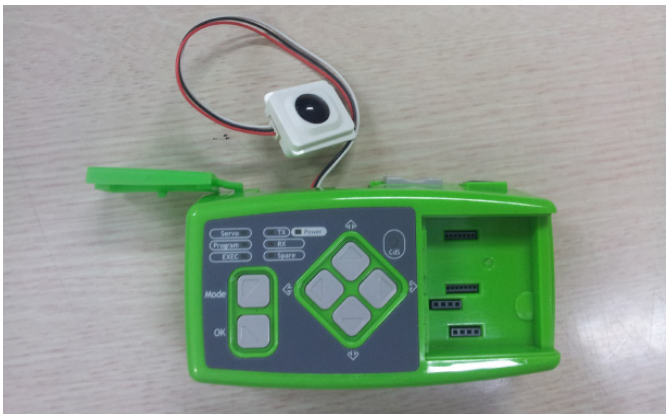
Hovisリモコンの場合は、Addressは0x61から0x6Aまでの値を持ちます。DRCの基本のファームウェアでは、この値をチャネルという値で表します。リモコンのチャンネルは1～0の数字+　OKボタンを同時に押して設定でき、それぞれの数字に対応するAddressの値は以下のとおりです。

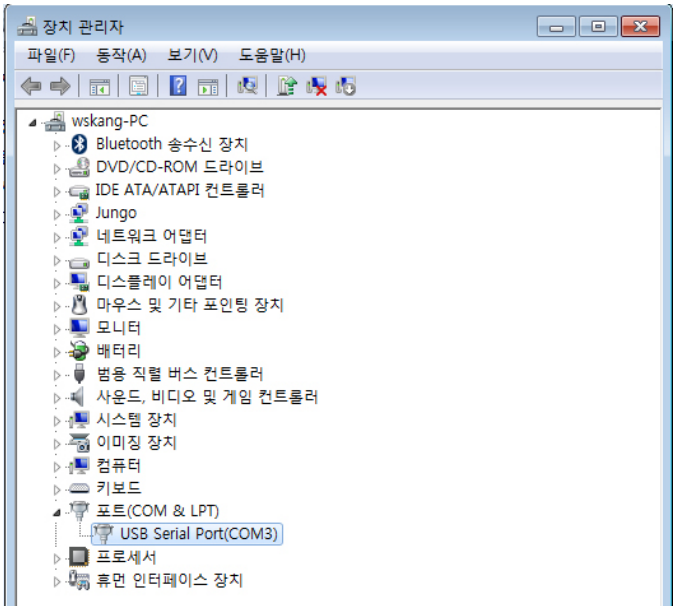
リモコンのボタン	RmcChannelの値
0+OK	97(0x61)
1+OK	98(0x62)
2+OK	99(0x63)
3+OK	100(0x64)
4+OK	101(0x65)
5+OK	102(0x66)
6+OK	103(0x67)
7+OK	104(0x68)
8+OK	105(0x69)
9+OK	106(0x6A)

また、リモコンの各キーごとに、該当するボタンのキー値は以下の図のとおりです。



準備





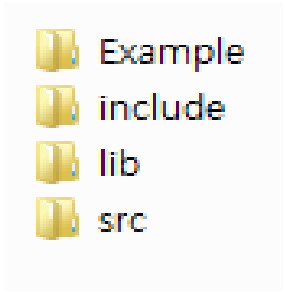
ハードウェアを準備する

01 接続する

DRC制御機のアダプターで電源を接続します。USB to Serial CableでPCとDRCを接続します。
赤外線受信機を接続します。リモコンも一緒に準備します。

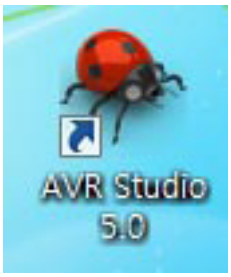
02 COM portを接続する

[マイコンピュータ]>[プロパティ]>[デバイスマネージャ]でCOM接続を確認します。



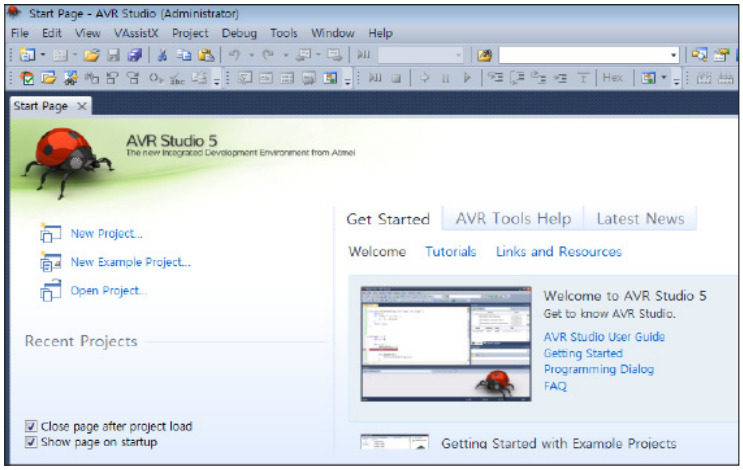
00 ライブラリ

リモコンの入力を受け取るためには、基本ライブラリファイルが必要です。ライブラリが含まれているフォルダ構造です。



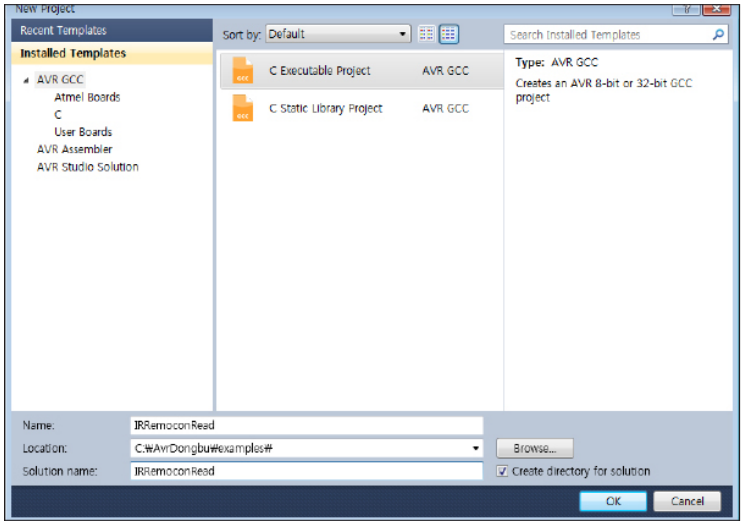
01 実行

AVRStudioをクリックして、実行します。



02 New Project

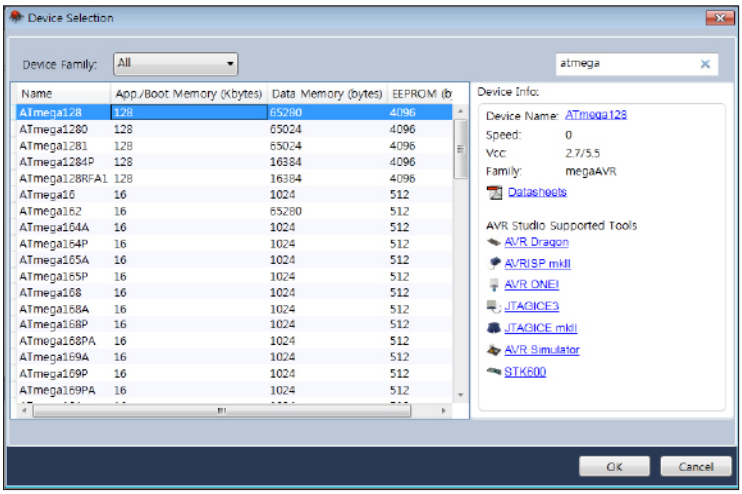
New Projectをクリックします。



03 GCC選択

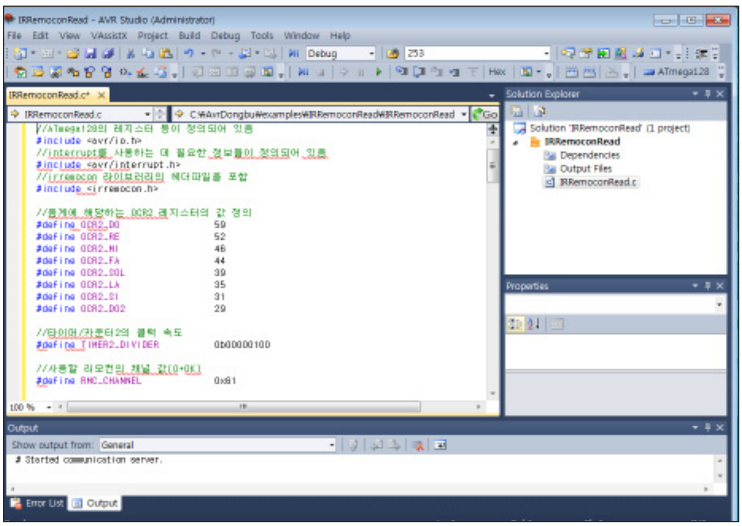
C Executable Projectを選択します。

Nameに“IRRemoconRead”と入力し、Create directory for solution をクリックして、OK ボタンを押します。



04 ATmega 選択

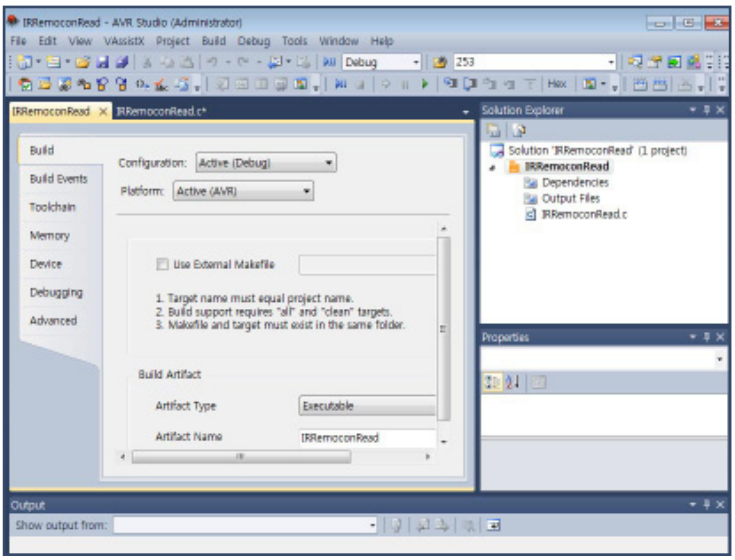
右上からatmegaを検索して、
リスト中のATmega128を選択します。



05 Editor

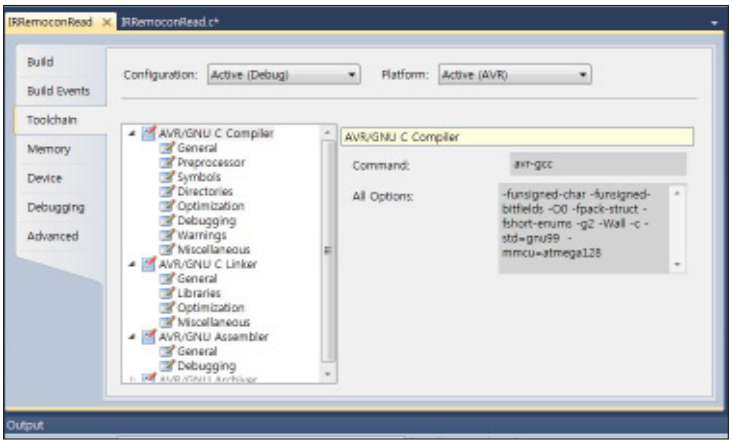
編集ウィンドウでコーディングを開始します。

ライブラリのリンクと設定



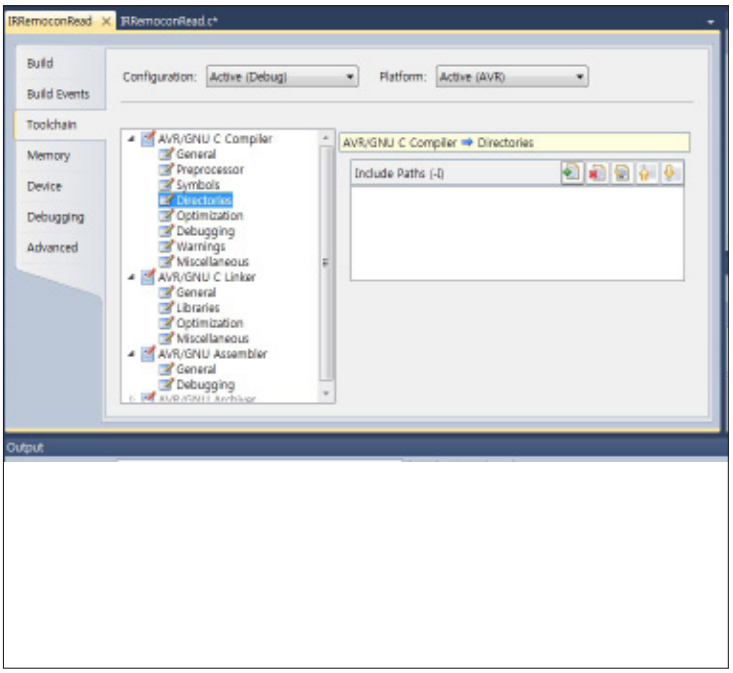
01 設定

AVR Studioの上部メニューの中のProjectをクリックすると、IRRemoconRead（プロジェクト名により異なります）Propertyメニューが生成されています。クリックすると、左側のように設定画面が出ます。



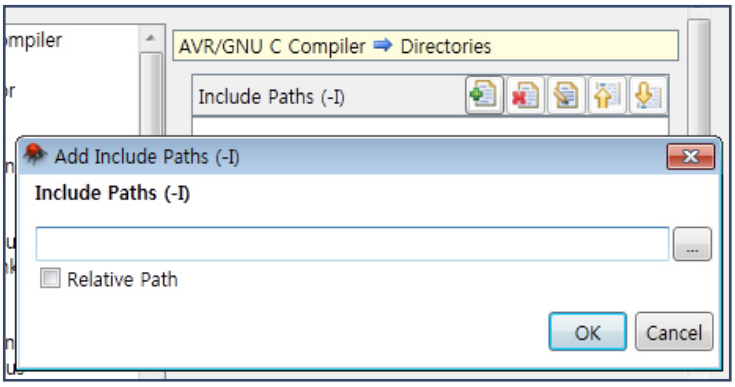
02 Toolchain

左側のタブボタンの中のToolchainをクリックして、AVR/ GNU C Compilerを選択します。



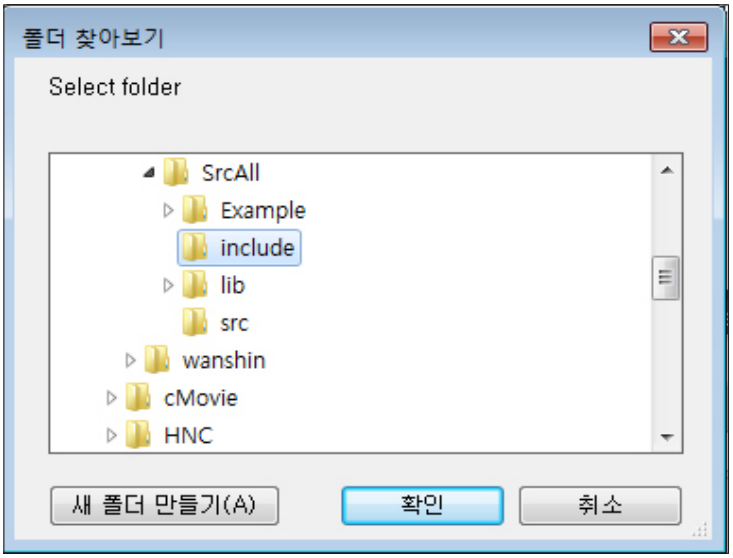
03 C Compiler

C Compiler フォルダ中のDirectoriesを選択します。



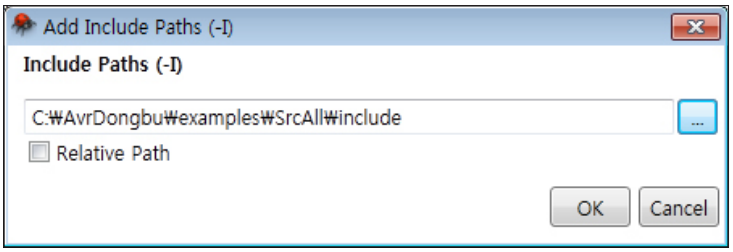
04 Path

Include Pathsの右にある+ボタンをクリックすると、ポップアップ入力画面が表示されます。
Relative Path（相対パス）のチェックを解除して（5．1ではしなくても良い）、右側のフォルダの検索ボタンをクリックします。



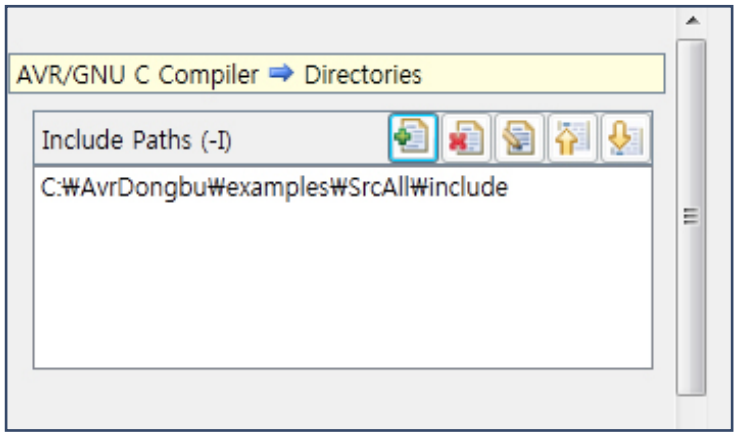
05 Include

提供ライブラリフォルダ中のincludeを選択します。



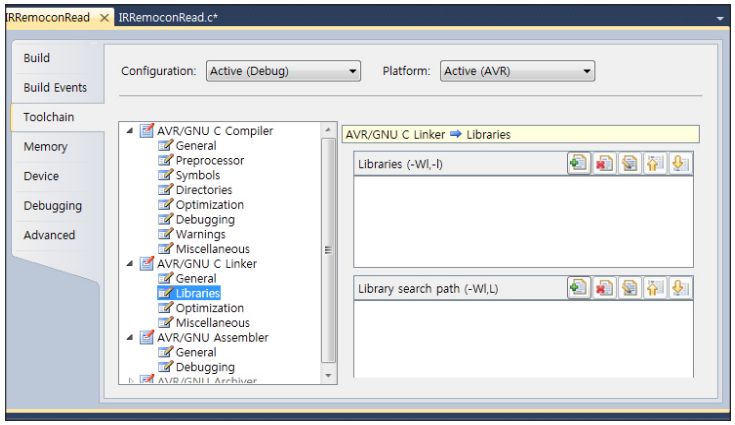
06 Include 完了

Includeフォルダの場所が指定されます。
OKボタンをクリックします。



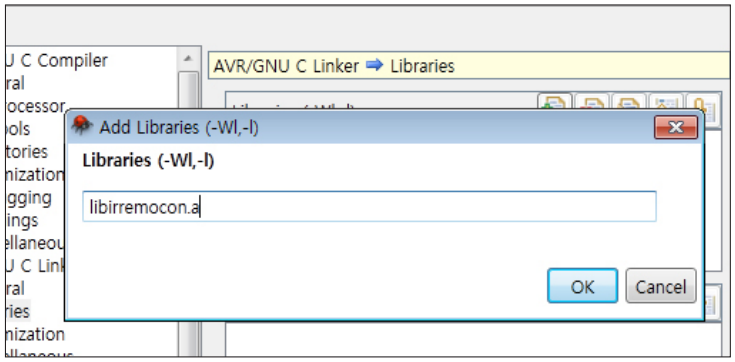
07 Path 完了

Includeパスの指定が完了しました。



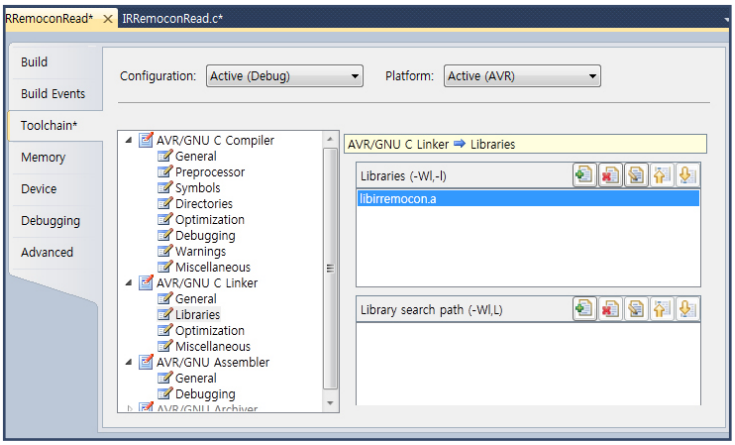
08 C Linker

C Linker フォルダ中のLibrariesを選択します。



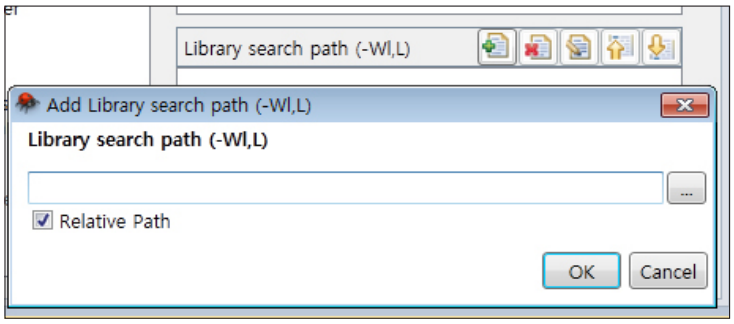
09 Libraries

Librariesウィンドウ右側にある+ボタンをクリックすると、Librariesファイル名入力画面が表示されます。libherkulex.aを直接入力します。
OKボタンをクリックします。



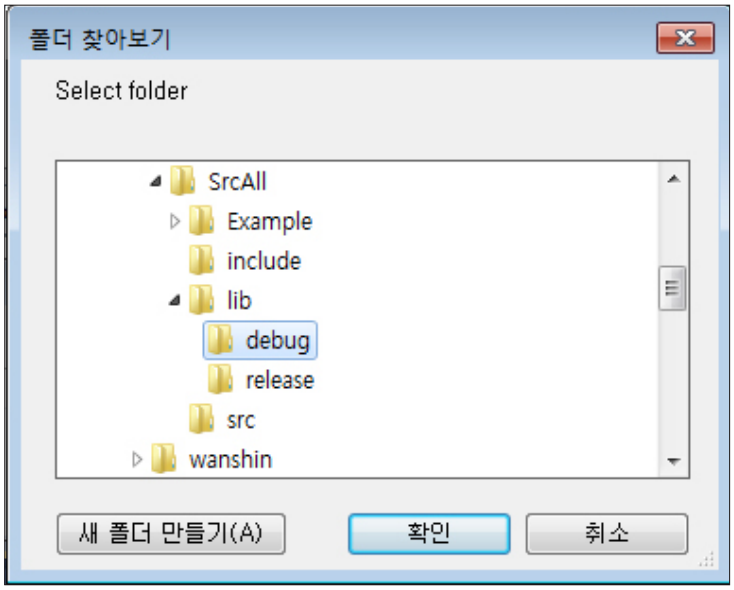
10 Search

リモコンのライブラリファイルlibaccgyro.aリンクが完了しました。
次に、Library search pathを設定します。右の+ボタンをクリックします。



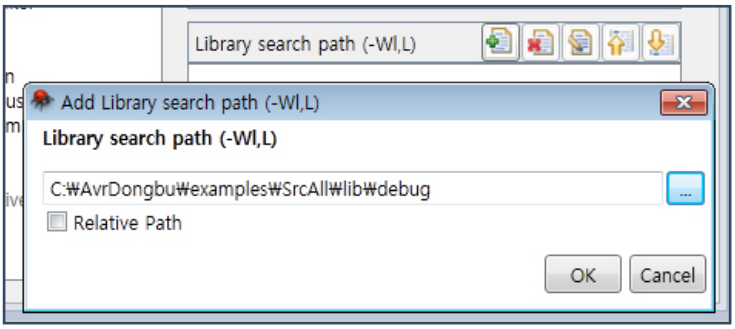
11 Path

Library search pathのポップアップ右側にあるフォルダの検索ボタンをクリックします。



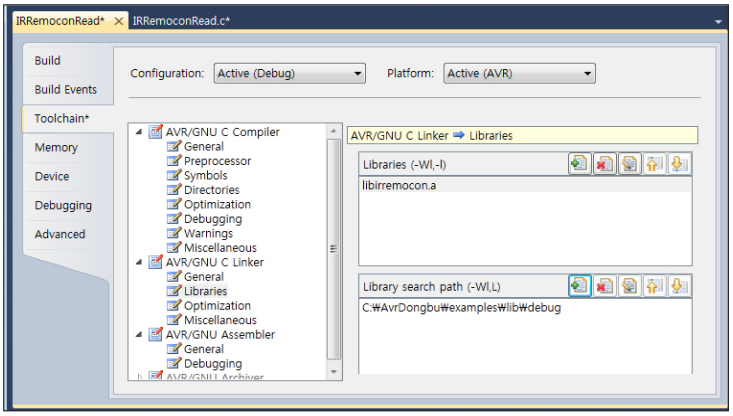
12 Debug

提供ライブラリフォルダのlibサブフォルダにあるdebugを選択します。



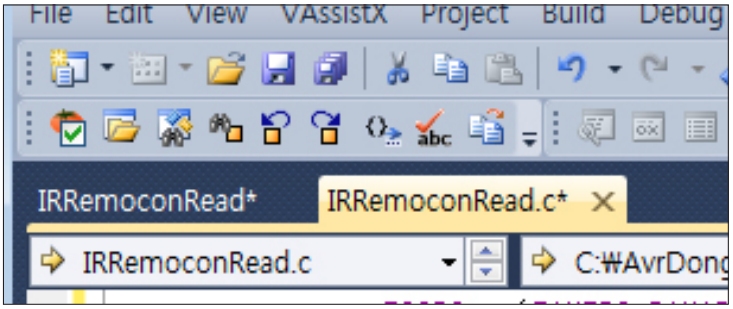
１３ Path 完了

Relative Path(相対パス指定)チェックボックスを解除します。
Library search pathのパスの指定が完了しました。
OKボタンをクリックします。



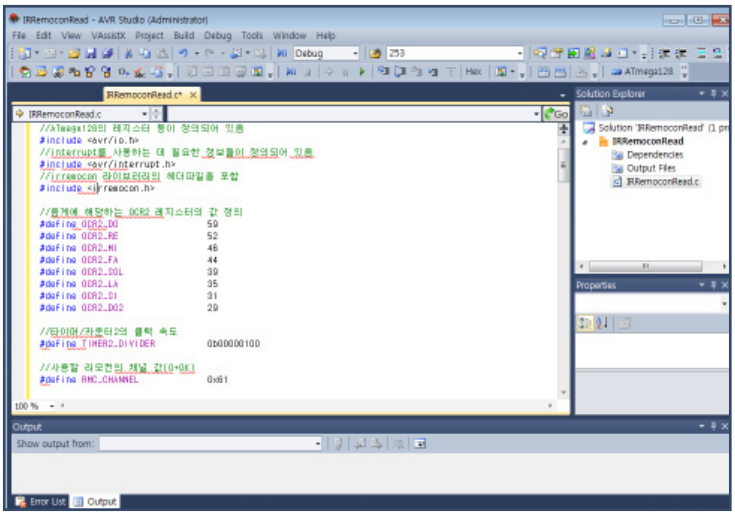
１４ Link 完了

C Linker ライブラリファイルとパスの指定が完了しました。



１５ 保存

保存ボタンを押すと現在の設定値が保存されます。タブの名前が IRRemocon-Read＊ -> IRRemoconRead に変更されます。



16 コーディング

IRRemocon.cのタブを選択して、下の例を入力します。

例

DRCに装着された赤外線リモコンモジュールを使って、リモコン値を受けて、入ったキーの値に応じてブザー音階を演奏する例です。ブザーに関する内容は、Chapter9を参照してください。

irremocon.h

```
#ifndef IRREMOCON_H_
#define IRREMOCON_H_

#ifdef _IRREMOCON_C
        #define IRREMOCON_EXT
#else
        #define IRREMOCON_EXT extern
#endif

//リモコンデータの基本値
#define RMC_ERROR_DATA                          0xFE
//リモコンが押された長さの最大値
#define RMC_LENGTH_MAX                          240

//赤外線リモコン受信のために、外部割り込みピンとタイマー/カウンタ1を設定する関数。
IRREMOCON_EXT void RMC_Init(void);
//赤外線リモコンのチャンネルucChannelで入ったキーの値と押された長さをucData、ucLengthに保存する関数。
IRREMOCON_EXT void RMC_ReadRemocon(unsigned char ucChannel, unsigned char *ucData, unsigned char *ucLength);

#endif /* IRREMOCON_H_ */
```

IRRemoconRead.c

```
//ATmega128のレジスタなどが定義されている
#include <avr/io.h>
//interruptを使用するために必要な情報が定義されている
#include <avr/interrupt.h>
//irremocon ライブラリのヘッダファイルを含む
#include <irremocon.h>

//音階に対応するOCR2レジスタの値の定義
#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

//タイマー/カウンタ2のクロック速度
#define TIMER2_DIVIDER          0b00000100

//使用するリモコンのチャンネル値(0+ OK)
#define RMC_CHANNEL             0x61

//使用するリモコンのキー値
#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)
{
        //8ビット変数の宣言
        unsigned char ucData, ucDataOld, ucLength;

        //ブザーピン(PB7)の入出力方向の設定
        DDRB |= 0b10000000;

        //ポートCの入出力方向の設定
```

```
DDRC = 0b01111111;
//ポートCの出力値設定
PORTC = 0b01111111;

//タイマー/カウンタ2の初期化
TCNT2 = 0;
OCR2 = 0;
TCCR2 = 0;

//全体の割り込みを無効に
cli();

//リモコン受信モジュールを使用するために初期化
RMC_Init();

//全体の割り込みを有効に
sei();

//ucDataOldを基本値に初期化
ucDataOld = RMC_ERROR_DATA;

//while内の内容を無限に繰り返す
while(1)
{
        //ucData, ucLengthからリモコン値の読み出し
        RMC_ReadRemocon(RMC_CHANNEL, &ucData, &ucLength);

        //リモコン値が前の値と異なる場合
        if(ucDataOld != ucData){
                //1番のキーが押された時に"ド"音がする
                if(ucData == RMC_KEY_1){
                        TCNT2 = 0;
                        OCR2 = OCR2_DO;
                        TCCR2 = (TIMER2_DIVIDER)|(1<<WGM21)|(1<<COM20);
                }
                //2番のキーが押された時に"レ"音がする
                else if(ucData == RMC_KEY_2){
                        TCNT2 = 0;
                        OCR2 = OCR2_RE;
                        TCCR2 = (TIMER2_DIVIDER)|(1<<WGM21)|(1<<COM20);
                }
                //3番のキーが押された時に"ミ"音がする
                else if(ucData == RMC_KEY_3){
                        TCNT2 = 0;
                        OCR2 = OCR2_MI;
                        TCCR2 = (TIMER2_DIVIDER)|(1<<WGM21)|(1<<COM20);
                }
```

```
                    //4番のキーが押された時に”ファ”音がする
                    else if(ucData == RMC_KEY_4){
                            TCNT2 = 0;
                            OCR2 = OCR2_FA;
                            TCCR2 = (TIMER2_DIVIDER)|(1<<WGM21)|(1<<COM20);
                    }
                    //5番のキーが押された時に”ソ”音がする
                    else if(ucData == RMC_KEY_5){
                            TCNT2 = 0;
                            OCR2 = OCR2_SOL;
                            TCCR2 = (TIMER2_DIVIDER)|(1<<WGM21)|(1<<COM20);
                    }
                    //6番のキーが押された時に”ラ”音がする
                    else if(ucData == RMC_KEY_6){
                            TCNT2 = 0;
                            OCR2 = OCR2_LA;
                            TCCR2 = (TIMER2_DIVIDER)|(1<<WGM21)|(1<<COM20);
                    }
                    //7番のキーが押された時に”シ”音がする
                    else if(ucData == RMC_KEY_7){
                            TCNT2 = 0;
                            OCR2 = OCR2_SI;
                            TCCR2 = (TIMER2_DIVIDER)|(1<<WGM21)|(1<<COM20);
                    }
                    //8番のキーが押された時に”ド”音がする
                    else if(ucData == RMC_KEY_8){
                            TCNT2 = 0;
                            OCR2 = OCR2_DO2;
                            TCCR2 = (TIMER2_DIVIDER)|(1<<WGM21)|(1<<COM20);
                    }
                    //1～8番キーが押されてない場合は音を止める
                    else{
                            TCNT2 = 0;
                            OCR2 = 0;
                            TCCR2 = 0;
                    }
            }

            //現在のデータ値を別の変数に保存
            ucDataOld = ucData;
    }

    //関数の形に合わせて整数値を返す
    return 1;
}
```

例の説明

まず、libirremocon.aのヘッダファイルであるirremocon.hについて説明します。

irremocon.h

```
#define RMC_ERROR_DATA                    0xFE
#define RMC_LENGTH_MAX                    240
```

リモコンで何も入力がない時や入力を受けている間に予期せぬ信号が入った場合は、リモコンのデータは、0xFEという基本値に設定します。つまり、何もリモコンの入力がない時に、リモコンの値を読み取る場合ucDataは0xFEという値が入っています。その基本値をマクロとして宣言して使用するためにRMC_ERROR_DATAという値を宣言します。
リモコン値を読み取る時にucLengthに入る値は、1当たり125ミリ秒の時間を表します。しかし、ucLengthは符号なし8ビット整数であるため、最大値を255までしか持つことができません。そのためlibirremocon.aでは、最大値を240（30秒）に制限して、30秒以上リモコンが押されている場合、無条件にキーの長さが240に読み込まれるようにしました。その最大値をマクロとして宣言して使用するためにRMC_LENGTH_MAXを宣言します。

```
IRREMOCON_EXT void RMC_Init(void);
IRREMOCON_EXT void RMC_ReadRemocon(unsigned char ucChannel, unsigned char *ucData, unsigned char *ucLength);
```

RMC_Initは赤外線リモコン受信モジュールが接続された外部割り込み4番ピン（PE4）と、リモコン信号の時間を計るのに使用されるタイマー/カウンタ1を初期化する関数です。　RMC_ReadRemoconは現在、リモコン受信モジュールに入ったデータ値を読み取る関数です。パラメータに入るucChannelは、データを受け取るチャンネルを指定するパラメータであり、もしucChannelと同じチャンネルでデータが入ると、ポインタucDataとucLengthにそのキーの値とキーが押された長さを保存します。もしデータが入っていない状態であればucDataはRMC_ERROR_DATA（0xFE）、ucLengthには0を保存します。

IRRemoconRead.c

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

avr/　io.hヘッダファイルには、DDRX、PORTXレジスタなどAVR制御のために必要なレジスタなどが定義されています。
avr/　interrupt.hファイルには、割り込みを使用するための情報が定義されています。またirremocon.hヘッダファイルには、libirremocon.aライブラリファイルに含まれている関数の原型が定義されています。現在のソースコードで使用できるように#include文を使用します。

```
#define OCR2_DO                    59
                                   …
#define OCR2_DO2                   29
```

使用する音階によるOCR2値を#defineで宣言します。この値は、音を再生するためにタイマー/カウンタ2のOCR2レジスタに値を入れる時に使用します。単純に59のような数字を代入すると、"ド"の音とは接続できませんが、OCR2_DOを代入すると"ド"の音が簡単に再生できます。なぜこのような値が宣言されるかはChapter9のH/ Wの原理を参照してください。

```
#define RMC_CHANNEL                    0x61
```

H/Wの原理で説明したように、リモコンを設定することによって、チャネルの値は0x61から0x6Aまでになります。今回の例では、リモコンの0番ボタン+ OKを押した時に設定される基本値である0x61をチャンネル値に使用します。

```
#define RMC_KEY_1                      1
                              …
#define RMC_KEY_8                      8
```

使用リモコンのキーとそれに対応するキーの値を宣言します。 1というリモコンキーを押すと、リモコンのデータ1が読み込まみます。今回使用するキーは、キーとデータの値が一致しているためあまり宣言する意味がありませんが、方向キーやOKなどの他のキーを使用する時は、キーとデータの値を宣言しておくと、毎回調べることなく使用できるので便利です。

```
unsigned char ucData, ucDataOld, ucLength;
```

メイン関数で使用する変数を宣言します。ucDataは現在、whileループで読んだリモコンの値であり、ucDataOldは直前のwhileループで読んだリモコンの値です。毎回whileループの最後でucDataOldはucDataの値に更新されます。この二つの変数は、タイマー/カウンタ2のリモコンを押したり離したりしてリモコン値が変わった瞬間だけ値を変更するために使用します。 ucLengthはリモコンが押された長さを読み取る変数です。

```
DDRB |= 0b10000000;
DDRC = 0b01111111;
PORTC = 0b01111111;
```

ポートB、Cを設定します。ポートBの7番ピン(PB7)はタイマー/カウンタ2のOC2出力ピンです。タイマー/カウンタ2でOC2出力ピンを使用すると設定した場合でもDDRBの7番目のビットが出力に設定されている必要があり、実際に出力されるので、DDRBの7番目のビットを1にして出力を設定します。ポートCはLEDが接続されたポートであるため、DDRCのLEDに対応する7つのビットに1を入れて、出力し、PORTCにも7つのビットに1を入れてLEDを切ります。

```
TCNT2 = 0;
OCR2 = 0;
TCCR2 = 0;
```

タイマー/カウンタ2のレジスタを初期化します。前で説明したレジスタのTCNT2、OCR2、TCCR2を初期値である0にしています。この時点でタイマー/カウンタ2はまだ動作しません。

```
cli();
RMC_Init();
sei();
```

割り込みの設定を変える時に割り込みが発生して予期せぬ誤動作が発生することを防止するために、まずcli()命令を使用して全体の割り込みを無効にします。その後RMC_Init関数を使用してリモコン信号を受信するための外部割り込み4番ピンとタイマー/カウンタ1を初期化します。初期化が終わった後は、sei()命令を使用して全体の割り込みを有効にします。

```
ucDataOld = RMC_ERROR_DATA;
```

前に説明したように、ucDataOld変数には、while文で直前に読み取ったリモコンの値が保存されます。しかし、まだwhile文に進入する前とリモコンの値を読む前なので、基本値RMC_ERROR_DATAをucDataOldに保存します。

```
RMC_ReadRemocon(RMC_CHANNEL, &ucData, &ucLength);
```

ここからはwhile文の中の内容です。RMC_ReadRemocon関数を使用して、リモコンの値を読み込みます。最初のパラメータには、希望するリモコンのチャンネルであるRMC_CHANNELを入れて、2番目、3番目のパラメータには、ucDataとucLengthのアドレスを入れてRMC_CHANNELチャンネルで入ったデータと押された長さがucDataとucLengthに保存されます。

```
if(ucDataOld != ucData){
        …
}
```

直前のwhileループで読み取ったリモコン値(ucDataOld)と先ほど読んだリモコン値(ucData)を比較して、異なる場合にif文の中の内容を実行します。リモコンキーが押されている瞬間や、押されているキーが離れる瞬間のみタイマー/カウンタ2の設定を変更するための部分です。

```
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);
}
```

cDataがRMC_KEY_1と同じであると、つまり、1番のキーが押されているとタイマー/カウンタ2の値を変えてブザーが"ド"の音を再生します。TCNT2は0に初期化します。値に応じて周波数が変化するOCR2は"ド"に該当する事前に定義された値を代入します。TCCR2はWGM21を1にしてCTCモードに設定します。COM20を1にして比較マッチが起こるたびにOC2ピンをトグルさせます。TIMER2_DIVIDERに定義された値にCS22:0を設定して必要な割合でクロックを分割します。例では、CS22:0を0b100に設定して256にクロックを分割します。

下のelseif文もリモコンキーとOCR2の値だけが変わっており他は同じです。if文、else if文を通して1番キー –ド、2番のキー – レ、3番キー – ミ、4番キー – ファ、5番キー – ソ、6番キー – ラ、7番キー – シ、8番キー – 高いドを再生します。

```
else{
        TCNT2 = 0;
        OCR2 = 0;
        TCCR2 = 0;
}
```

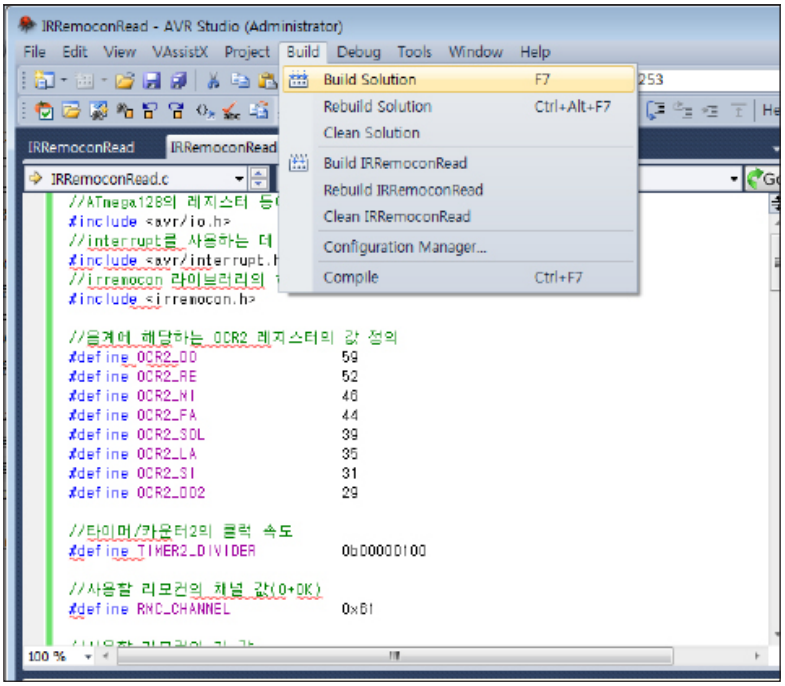
リモコンが押されていない場合と押されているが1～8番キーがない場合には、タイマー/カウンタ2レジスタをすべて0にして停止させます。それで、ブザーの音が停止します。

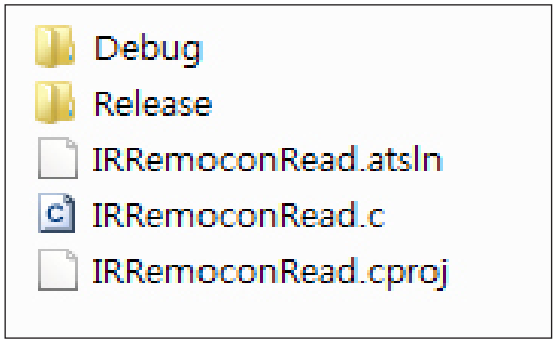
```
ucDataOld = ucData;
```

whileループが終了するたびに、現在のループで読んだucDataの値をucDataOldに代入して、次のループの比較に使用します。

実行

赤外線リモコン受信モジュールをDRCに装着した後、ファームウェアを実行すると、リモコンのキーを押す時に該当するブザー音が再生されます。





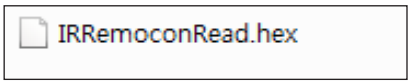
01 Build

Buildメニューを選択しBuild Solutionをクリックしてビルドします。

1 succeeded0 failedが出るとエラーなしでコンパイルが完了したということです。

02 Debug フォルダ

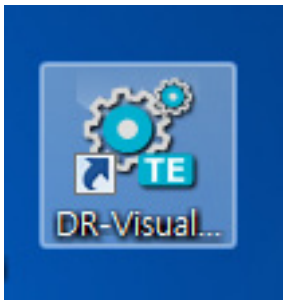
Build後にソースがあるフォルダに入るとDebugフォルダが作成されています。クリックしてフォルダの中身を見ます。



03 hex 確認

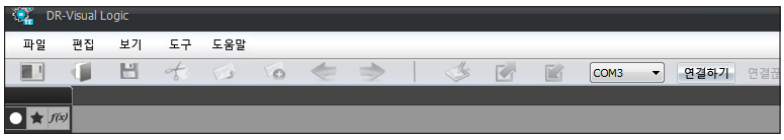
Debugフォルダの中のhexファイルを確認します。Hexファイルは制御機に移動するファイルです。

DR-Visual Logicでファームウェアの修復を行います。



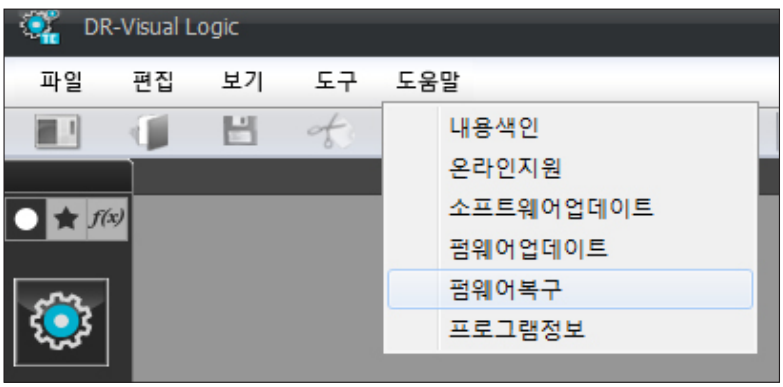
04 Visual Logic

DR-Visual Logicを実行します。



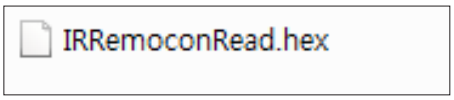
05 接続する

COM番号を設定して接続するをクリックします。



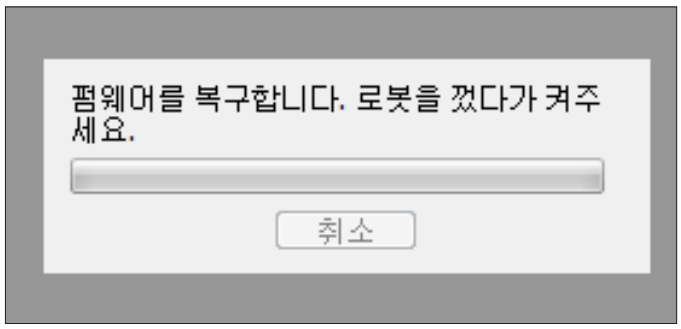
06 ファームウェアの修復

[ヘルプ]メニューからファームウェア修復をクリックします。



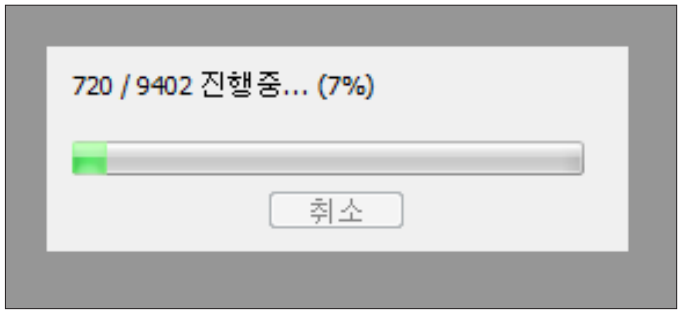
07 ロード

DebugフォルダにHexファイルをロードします。



08 ファームウェアの修復

“ロボットの電源を切って入れ直してください”というメッセージが出ます。制御機の電源ボタンを一度切ってから再度入れ直してください。



09 修復の開始

ファームウェアの修復が開始されます。



10 実行

赤外線リモコン受信モジュールをDRCに装着した後、ファームウェアを実行します。リモコンのキーを押すと、該当するブザー音が再生されます。

PART 04

ロボットに応用する

CH.1 : 16軸ヒューマノイドモーションを実行する

herkulex ライブラリとモーションデータファイルを使用してHovis Liteの16軸ロボットのモーションを実行します。

H/W 原理

Chapter8で説明したように、DRCに装着されたATmega128には、USART0、USART1の二つのUSARTポートがあり、このうちUSART0はHerkuleXとの通信に使用される4ピンのポート5個が並列に接続されています。制御したいHerkuleXをケーブルでDRCと接続すると、USART0を通って直接パケットを送受信できます。

今回の例で使用するパケットに使用された命令を要約すると以下の通りです。

名	CMD	説明
RAM_WRITE	0x03	HerkuleXの揮発性レジスタ領域にデータを使用します。
S_JOG	0x06	HerkuleXの位置/速度を制御して動くようにします。I_JOGは各モーター毎に移動時間を変更することができますが、S_JOGはすべての時間で同じ時間だけ動かして、目的の場所に到達します。 I_JOGとS_JOG パケットで時間を表すためには、1バイトのデータを使用します。この時のデータは1バイトあたり11.2msを示しています。つまりI_JOGパケットに50の時間データが入っている場合、実際にモーターは50*11.2=560msの時間をかけて、目標位置に移動します。

今回の例で使用されている揮発性レジスタ領域の項目を要約すると以下の通りです。

名	アドレス	サイズ	説明
Torque Control	52	1	トークの認可の状態を制御します。 0x00:自由に動く力がない状態 0x40:動く力はないが、現在の位置を維持する状態 0x60:トルクが認可され自分で駆動する状態

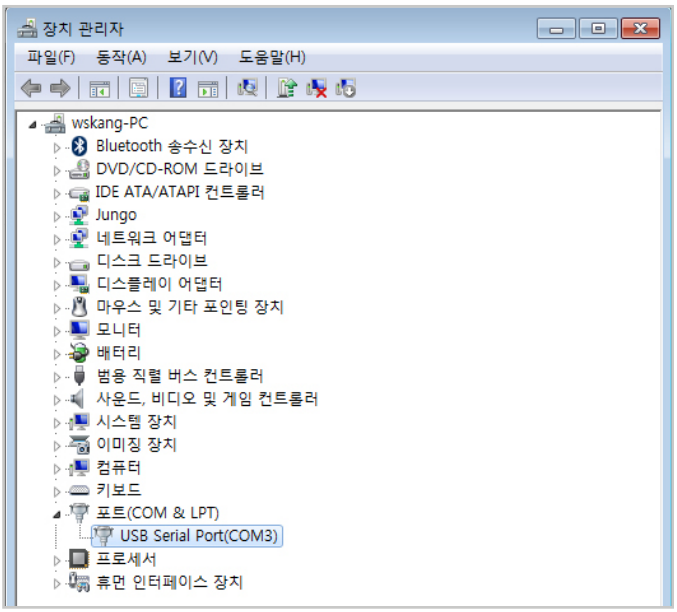
プロトコルとレジスタマップの詳細については、HerkuleXマニュアルを参照してください。

ハードウェアを準備する



01 接続 する

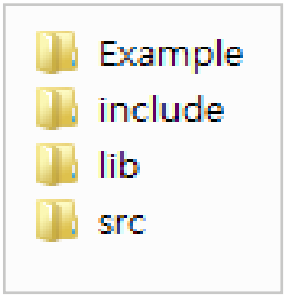
DRC 制御機のUSB to Serial CableにPCとDRCを接続します。
16軸ヒューマノイドHovis Liteを PCと接続します。



02 COM portを接続する

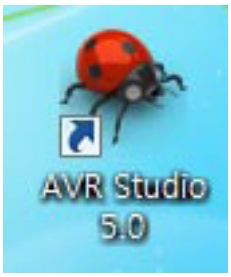
[マイコンピュータ]>[プロパティ]>[デバイスマネージャ]でCOM接続を確認します。

AVR Studio 実行とコーディングの準備



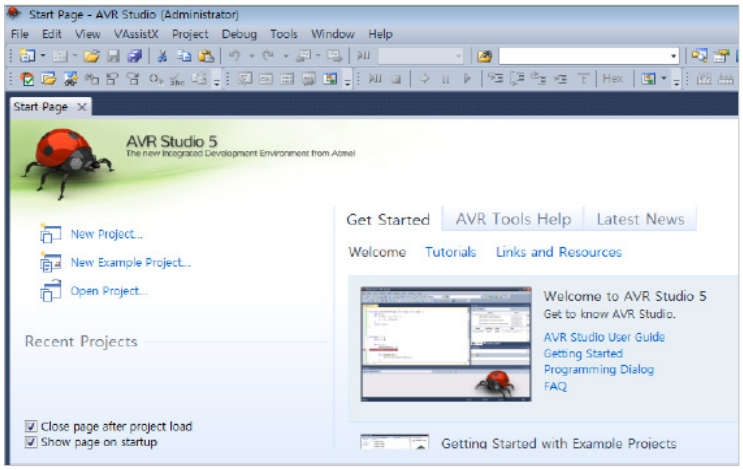
00 ライブラリ

Hovis Liteを制御するためには、基本ライブラリファイルが必要です。
ライブラリが含まれているフォルダ構造です。



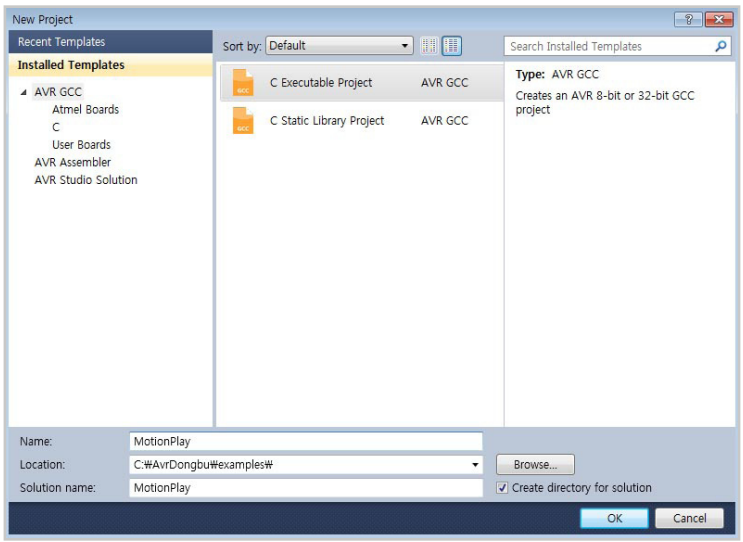
01 実行

AVR Studioを実行します。



02 New Project

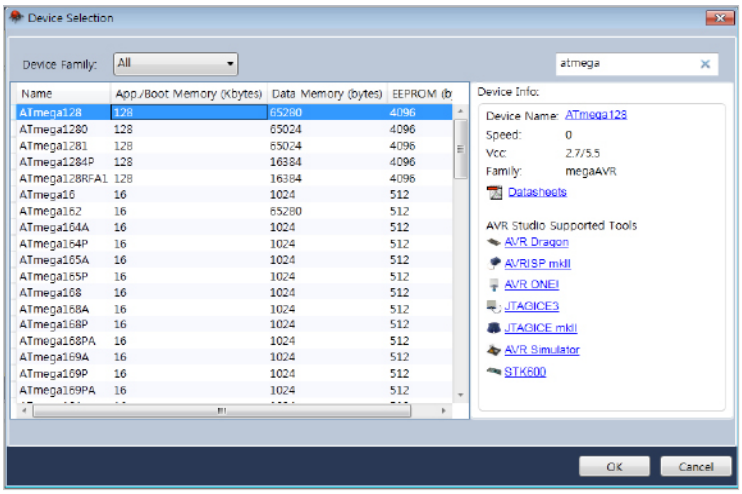
New Projectをクリックします。



03 GCC選択

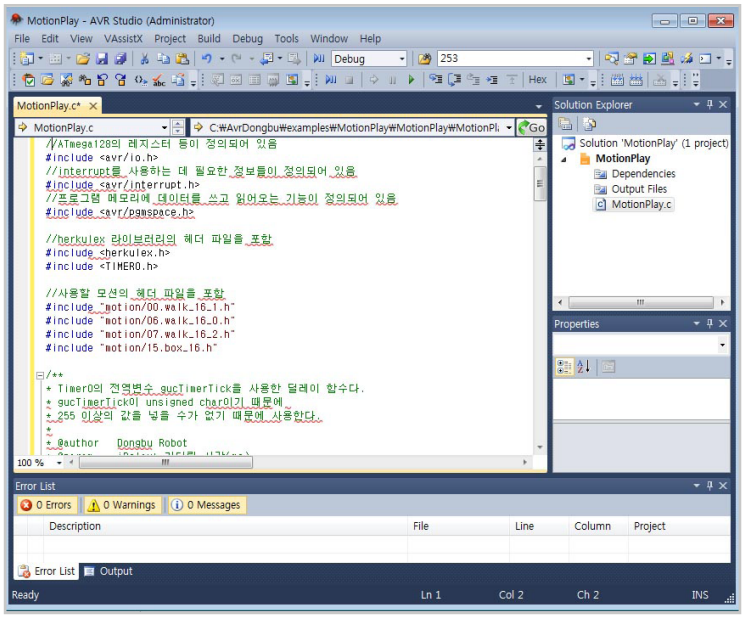
C Executable Projectを選択します。

Nameに“Motionplay”と入力し、 Create directory for solution をクリックして、OK ボタンを押します。



04 ATmega 選択

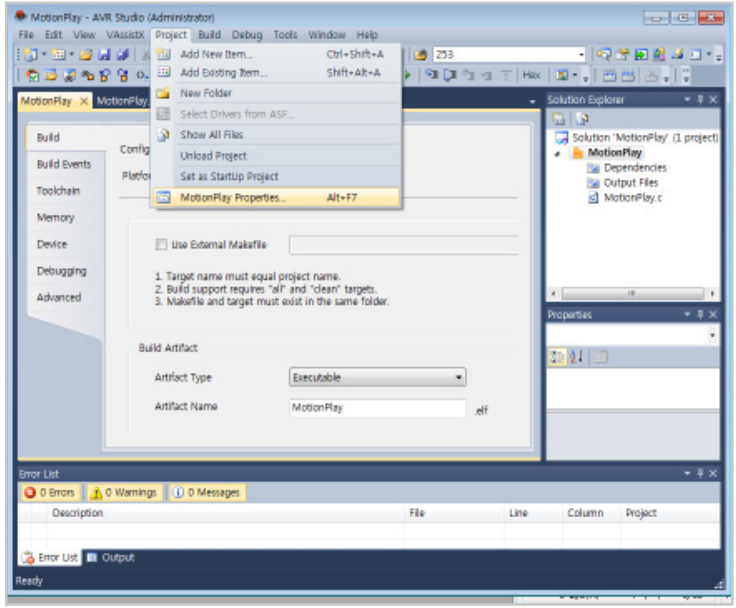
右上からatmegaを検索して、
リスト中のATmega128を選択します。



05 Editor

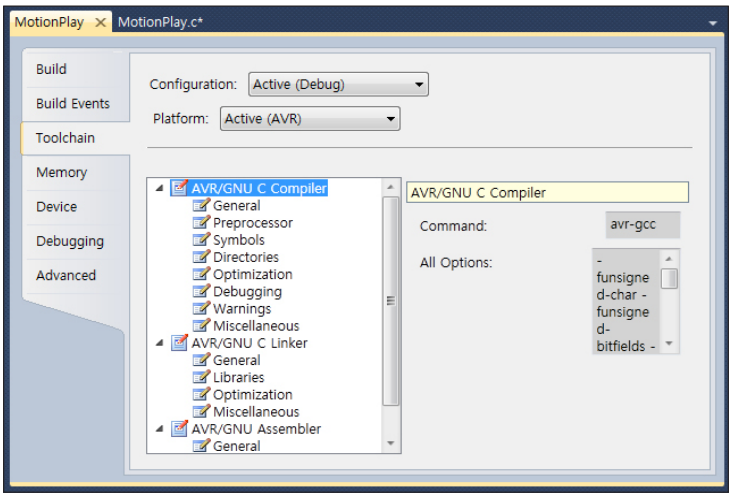
編集ウィンドウでコーディングを開始します。

ライブラリのリンクと設定



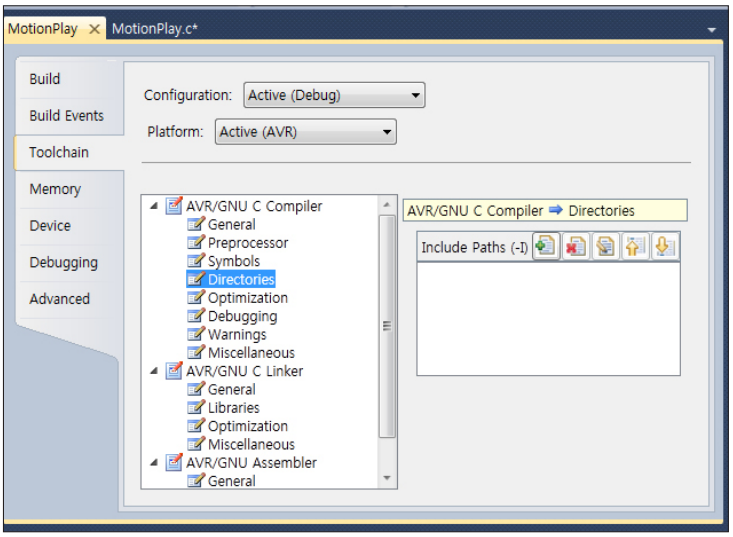
01 設定

AVR Studioの上部メニューの中からProjectをクリックすると、MotionPlay（プロジェクト名により異なります）Propertyメニューが生成されています。クリックすると、左側のように設定画面が出ます。



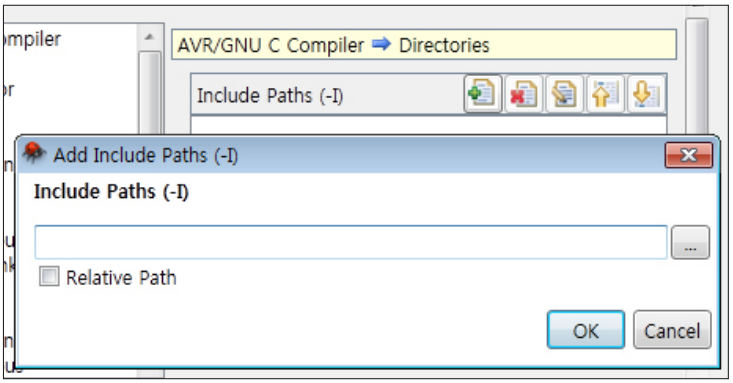
02 Toolchain

左側のタブボタンの中のToolchainをクリックして、AVR/ GNU C Compilerを選択します。



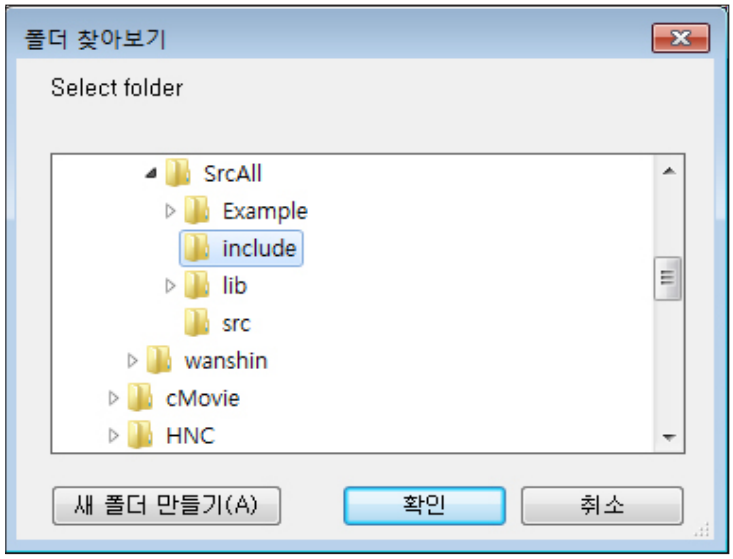
03 C Compiler

C Compiler フォルダ中のDirectoriesを選択します。



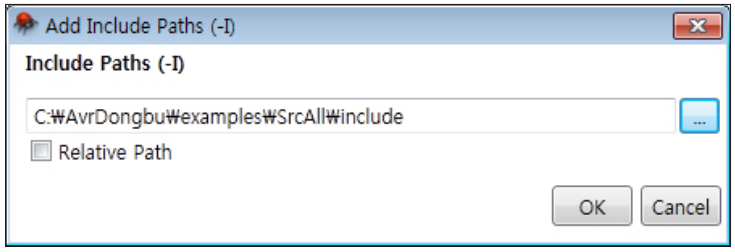
04 Path

Include Pathsの右にある+ボタンをクリックすると、ポップアップ入力画面が表示されます。
Relative Path（相対パス）のチェックを解除して、右側のフォルダの検索ボタンをクリックします。



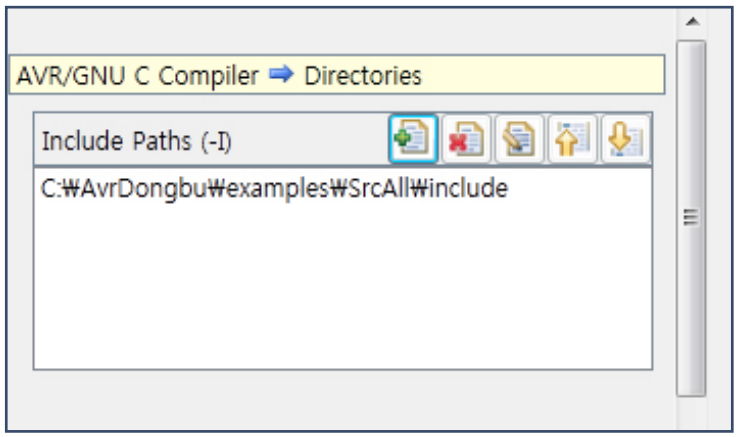
05 Include

提供ライブラリフォルダ中のincludeを選択します。



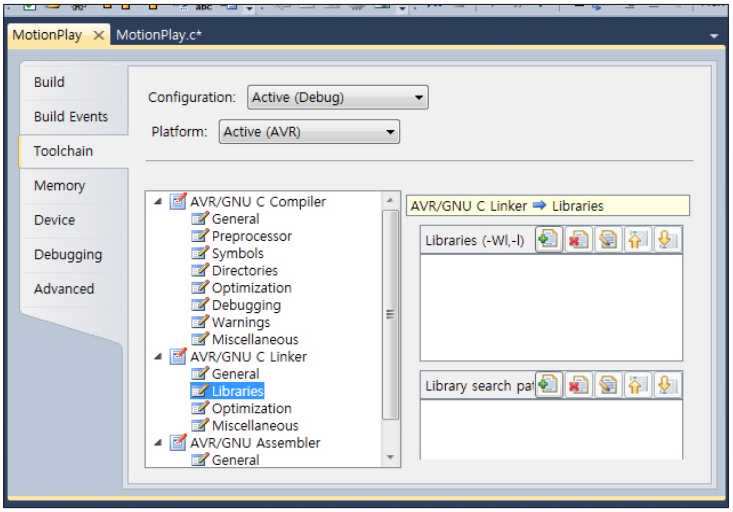
06 Include 完了

Includeフォルダの場所が指定されます。
OKボタンをクリックします。



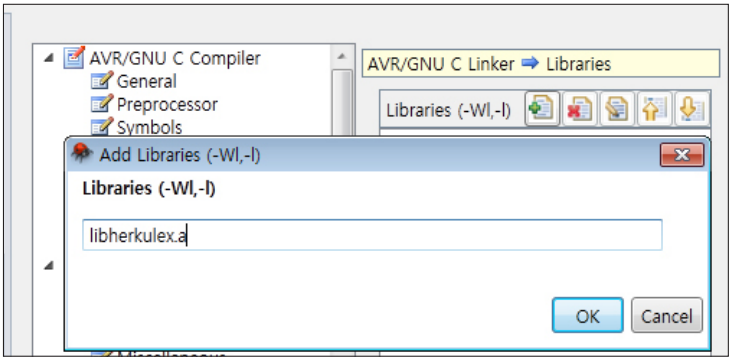
07 Path 完了

Includeパスの指定が完了しました。



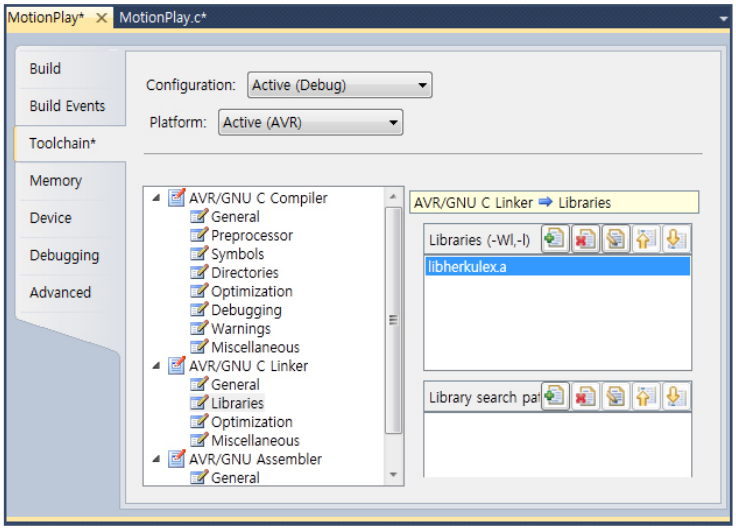
08 C Linker

C Linker フォルダ中のLibrariesを選択します。



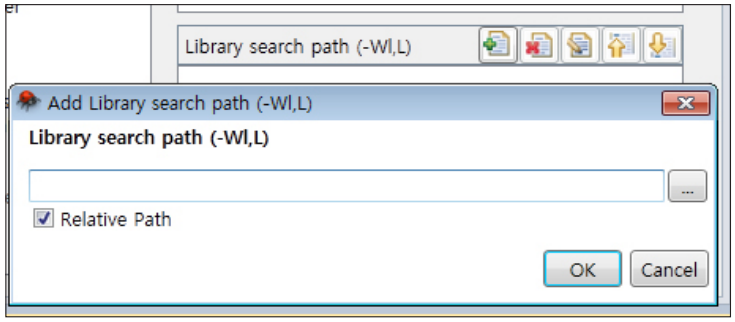
09 Libraries

Librariesウィンドウ右側にある+ボタンをクリックすると、Librariesファイル名入力画面が表示されます。 libherkulex.aを直接入力します。
OKボタンをクリックします。



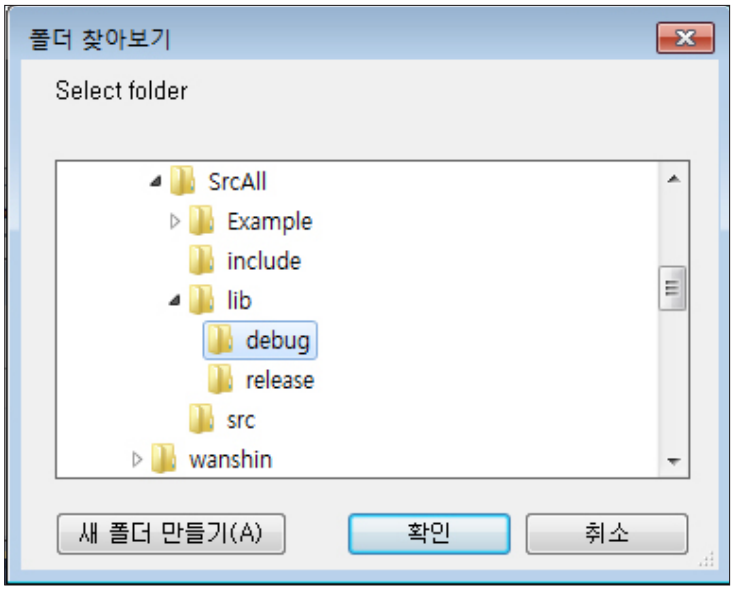
10 Search

サーボモーターのライブラリファイルlibherkulex.aリンクが完了しました。
次に、Library search pathを設定します。
右の+ボタンをクリックします。



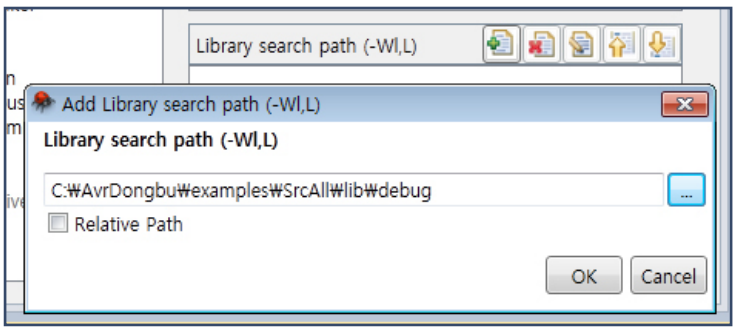
11 Path

Library search pathのポップアップ右側にあるフォルダの検索ボタンをクリックします。



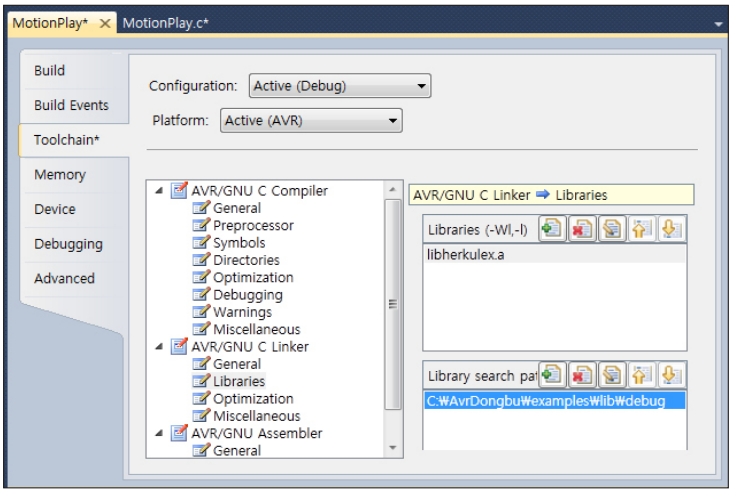
12 Debug

提供ライブラリフォルダのlibサブフォルダにあるdebugを選択します。



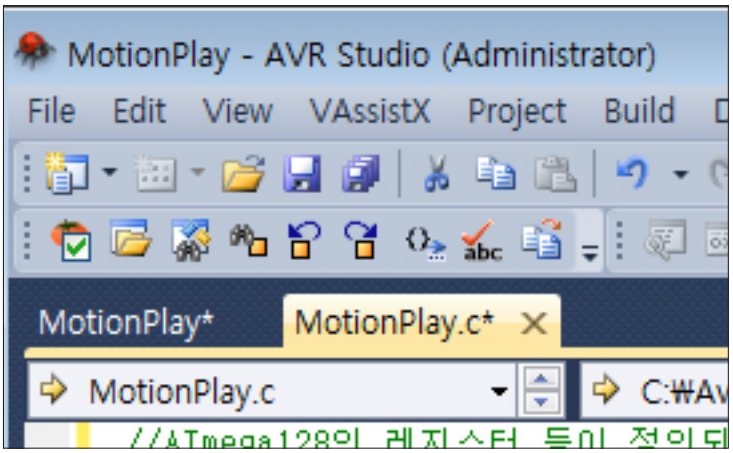
13 Path 完了

Relative Path(相対パス指定)チェックボックスを解除します。
Library search pathのパスの指定が完了しました。
OKボタンをクリックします。



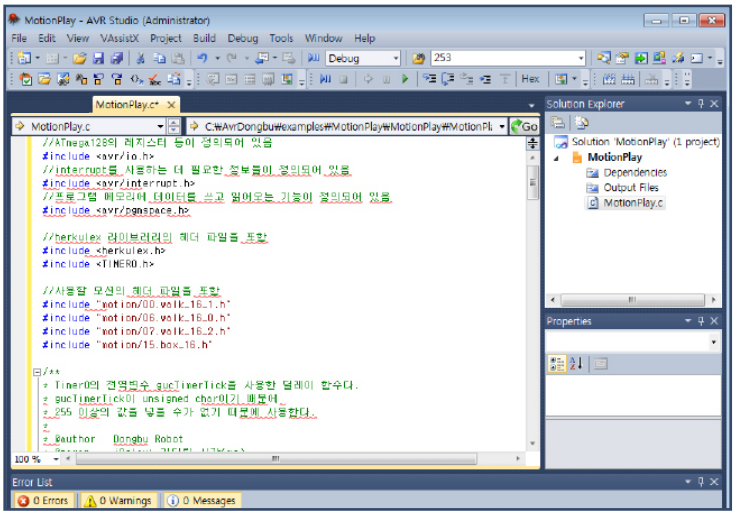
14 Link 完了

C Linker ライブラリファイルとパスの指定が完了しました。



15 保存

保存ボタンを押すと現在の設定値が保存されます。タブの名前がMotionPlay* ->MotionPlayに変更されます。



16 コーディング

MotionPlay.cのタブを選択して、下の例を入力します。

例

ロボットの電源を入れると、10歩前進してボクシングをすることを繰り返す例です。

00.Walk_16_1.h

```
#ifndef WALK_16_1_H_
#define WALK_16_1_H_

//現在のモーションでプレイするフレームの数
prog_int16_t walk_16_1_FrameCnt = 48;

//現在のモーションで制御するモーター数
prog_int16_t walk_16_1_MotorCnt = 16;

//フレームに対応するモーターの目標位置の値
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}
};

//各フレームのモーター移動時間
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
};

//各フレームの次のフレームまでのディレイ
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 クロックの設定
#define F_CPU 16000000

//ATmega128のレジスタなどが定義されている
#include <avr/io.h>
//interruptを使用するために必要な情報が定義されている
#include <avr/interrupt.h>
//プログラムメモリにデータを読み書きする機能が定義されている
#include <avr/pgmspace.h>

//herkulex ライブラリのヘッダファイルを含む
#include <herkulex.h>
#include <TIMER0.h>

//使用モーションのヘッダファイルを含む
#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”

/**
* Timer0のグローバル変数gucTimerTickを使用したディレイ関数です。
* gucTimerTickがunsigned charなので
* 255以上の値を入れないようにして使用する。
*
* @author          Dongbu Robot
* @param           iDelay: 待ち時間(ms)
* @return          なし
* @version         1.0.0.0
*/
void delay_ms(int iDelay){
        //iDelayが0より大きい場合、繰り返す
        while(iDelay>0){
                //iDelayが500msより大きい場合
                if(iDelay>500){
                        //gucTimerTick250を入れてiDelayから500引く。
                        gucTimerTick=250;
                        iDelay -= 500;
                        //Timer0によってgucTimerTickが0になるまで待機
                        while(gucTimerTick>0);
                }
                //iDelayが500ms以下の場合
                else{
                        //gucTimerTickにiDelayを2で割って入れてiDelayを0にする。
                        gucTimerTick=iDelay/2;
                        iDelay = 0;
                        //Timer0によってgucTimerTickが0になるまで待機
                        while(gucTimerTick>0);
                }
        }
}
```

```
/**
* モーションを実行するための関数である。
*モーションヘッダファイルに宣言された複数の情報プログラムメモリから読み出して、
*適当なS_JOGパケットを構成して送りモーションを実行します。
*
* @author          Dongbu Robot
* @param           FrameCnt: 実行するモーションフレームの数
*                  MotorCnt: 実行するモーションの軸数
*                  MotorData[FrameCnt][MotorCnt]: 各フレームのモーター目標位置の値
*                  PlayTimeData[FrameCnt]: 各フレームのモーター移動時間値
*                  DelayData[FrameCnt]: 各フレームのディレイ時間の値
* @return          なし
* @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])
{
    //変数と構造体の宣言
    unsigned char i;
    unsigned int uiCurrentFrame;
    DrsPacket stSendPacket;

    //最初のフレームから最後のフレームまで繰り返す
    for(uiCurrentFrame=0;uiCurrentFrame<FrameCnt;uiCurrentFrame++){
        //パケットサイズの設定。最小サイズ(7)+(モーター数*モーター当りのデータのサイズ
(4))+移動時間(1)
        stSendPacket.ucPacketSize = MIN_PACKET_SIZE + CMD_S_JOG_STRUCT_SIZE * MotorCnt + 1;
        //複数のサーボに与える命令なので、0xFEに設定
        stSendPacket.ucChipID = BROADCAST_ID;
        //S_JOG 命令で設定
        stSendPacket.ucCmd = CMD_S_JOG;
        //現在のフレームに該当するPlayTimeDataを読んで、11.2に分けて四捨五入して代入
        stSendPacket.unData.stSJogData.ucPlayTime =
                        (pgm_read_word(PlayTimeData+uiCurrentFrame)*10+56)/112;
        //最初モーターから最後のモーターまで繰り返す
        for(i=0;i<MotorCnt;i++){
            //現在のフレームとモーターに対応するMotorDataを読んで代入
            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;
            //モーションデータはモーターIDが0～15番の順になっているのでiを代入
            stSendPacket.unData.stSJogData.stSJog[i].ucId = i;
        }
```

```
                //構成したパケットを送信する
                hklx_SendPacket(stSendPacket);

                //現在のフレームに該当するPlayTimeData+ DelayDataほど待ってから、次のフレーム
を実行
                delay_ms(pgm_read_word(PlayTimeData+uiCurrentFrame)+
                                        pgm_read_word(DelayData+uiCurrentFrame));
        }

        return;
}

int main(void)
{
        //送信するパケットと受信するパケットが保存されるパケットの構造体宣言
        DrsPacket stSendPacket;

        //ポートCの入出力方向の設定
        DDRC = 0b01111111;
        //ポートCの出力値の設定
        PORTC = 0b01111111;

        //全体の割り込みを無効にする
        cli();

        //HerkuleXを使用するために初期化
        hklx_Init(115200);

        //全体の割り込みを有効にする
        sei();

        //Torque Control□ 0x60を使ってトークをかけるパケットの構成
        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;

        //パケットを送信する
        hklx_SendPacket(stSendPacket);

        while(1)
        {
                //歩くモーション実行(前進開始 - 前進*3 - 前進終了)
                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);

                //1秒間停止
                delay_ms(1000);
```

```
                //ボクシングモーションを実行
                PlayMotion(box_16_FrameCnt, box_16_MotorCnt,
                                box_16_MotorData, box_16_PlayTimeData, box_16_DelayData);

                //1秒間停止
                delay_ms(1000);
        }

        //関数の形に合わせて定数値を返す
        return 1;
}
```

例の説明

MotionPlay.cで、＃include文を使って含めるヘッダファイル中のlibherkulex.aに関するTIMER0.hとherkulex.hの説明はChapter8. HerkuleXを制御するを参照してください。 ＃include文を使用して含めた00.Walk_16_1.hなどのモーションファイルはすべての形式が同じなので、00.Walk_16_1.hのみ説明します。

00.Walk_16_1.h

```
#ifndef WALK_16_1_H_
#define WALK_16_1_H_

…

#endif /* WALK_16_1_H_ */
```

00.Walk_16_1.hをはじめとするモーションヘッダファイルは、上記のように＃ifndef文＃endifで全体が囲まれています。一般的なc言語のヘッダファイルのように、既存のWALK_16_1_H_が＃defineを使用して宣言されていることを確認して、宣言されていない場合WALK_16_1_H_を宣言して、ヘッダーファイルの内容をコンパイルします。そうすることで、同じモーションのヘッダーファイルが複数含まれた状態でコンパイルされることを防ぎます。

```
prog_int16_t walk_16_1_FrameCnt = 48;
```

モーションファイルでロボットの一つの姿勢、または一つの瞬間を示すことをフレームと呼びます。フレームを順に時間に合わせて再生すると一つのモーションが実行されます。現在00.Walk_16_1モーションは48個のフレームで構成されたモーションです。変数の前に付いたprog_int16_tはunsigned char、unsigned intなどの変数のタイプを示すものです。 prog_int16_tは特殊なタイプで、変数をRAM領域ではなく、プログラムメモリ、つまりflash領域に宣言するという意味です。モーションのヘッダーファイルで宣言するモーションは、データの量が膨大なため、モーションデータをすべて一般変数として宣言してRAMに保存しすると、4kByteしかないATmega128のRAM容量が不足します。そのためモーションデータをプログラムメモリに保存しておいて必要な時に読んで使用します。

一方prog_int16_tなどのタイプを使って変数をプログラムメモリに保存すると、プログラムの実行中に変数の値を変えることができず、最初に宣言した値で維持されます。プログラムメモリは実行されるプログラムが保存されているところであるため、実行中のプログラムメモリの値を変更できれば、プログラム自体が破損し、誤動作を起こすことがあります。モーションデータの場合数が決まっておる変わることがないためプログラムメモリ領域に保存することが適しています。prog_int16_tなどのタイプは<avr/pgmspace.h>ファイルが含まれている場合にのみ使用可能です。

```
prog_int16_t walk_16_1_MotorCnt = 16;
```

モーションを実行するには、現在のモーションがいくつのモーターで構成されているロボットなのか知る必要があります。00.Walk_16_1モーションは16軸で構成されるモーションです。

```
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}
};
```

モーションの各フレームで、それぞれのモーターがどの位置にいるのか、モーターの位置の値を保存する2次元配列です。48個のフレームで構成され、モーションにはモーターが16個あるので、[48][16]の2次元配列が必要です。

```
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
};
```

モーションファイルの各フレームに対応するS_JOGパケットをHerkuleXに送信するときに、どれだけの時間の間、該当姿勢で動くべきであるかを示すデータです。このデータは、ミリ秒(ms)単位です。S_JOGパケットに入る場合11.2ms単位で変化しながら入ります。　00.Walk_16_1モーションの実行時間は、すべて70msなので、11.2ms単位に換算すると70/11.2=6.25、つまり四捨五入して6というデータがS_JOGパケットに入ります。

```
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
};
```

モーションファイルの実行時間が終わった後、次のフレームをどのくらい待って送信するのかを決定するのがディレイです。このデータも、やはりミリ秒(ms)単位です。例えば00.Walk_16_1モーションで実行時間が70msでディレイが0msの場合、フレームを実行するS_JOGパケットを送ってから70msが経過すると、すぐに次のフレームのパケットを送信します。例えば、ディレイが14msであれば、70ms+14 ms=84msが経ってから、次のパケットを送信します。ロボットの動きが終わってからも14msを待って送信します。一方、このモーションのようにディレイが-14msであれば、70ms-14ms=56msが経ってから、次のパケットを送信するので、ロボットの動きが終わる前に次のフレームのパケットを送信します。マイナス(-)ディレイを与えた時の利点は、ロボットの動作が切れずにスムーズにつながるモーションを作成できるということです。

MotionPlay.c

```
#define F_CPU 16000000
```

#define文はコードの前処理命令の一つで、今後F_CPUという値が出たら16000000に置き換えられることを意味しています。DRCのATmega128は16MHzのクロック速度で動作します。クロック速度に比例して命令の処理速度が異なりますので、この命令を通してこのプログラムが動作する速度を明示するとdelay関数などが正しく動作します。

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

avr/io.hヘッダファイルには、DDRX、PORTXレジスタなどAVR制御のために必要なレジスタなどが定義されていて、avr/interrupt.hヘッダファイルには、割り込みを使用するための関数が定義されています。 avr/ pgmspace.hヘッダファイルには、プログラムメモリにデータを読み書きする機能が定義されています。前に説明したprog_int16_tなどの変数タイプや後ろに出てくるpgm_read_word()などのプログラムメモリを読み出す関数などが宣言されているので含める必要があります。

```
#include <herkulex.h>
#include <TIMER0.h>
```

herkulex.hヘッダファイルには、パケットを構成して送信するための宣言と構造体、関数が定義されているので、ぜひ含める必要があります。そしてgucTimerTickを使ってdelayを実行するために、TIMER0.hも含めます。

```
#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”
```

現在の例では、実行するモーションのデータが入ったヘッダファイルです。前に説明した00.walk_16_1.hのようにモーションのフレーム数、軸数、モーターの位置データ、実行時間のデータ、ディレイデータが宣言されています。

```
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);
                }
            }
    }
```

TIMER0を利用したディレイ関数です。パラメータとしては、iDelayTickがあり、どれだけの時間待機するかどうかミリ秒（ms）単位で表したものです。 gucTimerTickはTIMER0.hで宣言されたグローバル変数で、普段は0に維持され、0以外の値を代入すると、libherkulex.aに定義されたTIMER0オーバーフロー割り込みサービスルーチンによって2ミリ秒ごとに1ずつ減少します。例えば、10ミリ秒を待ちたい場合、gucTimerTickに5という値を入れてgucTimerTickが0になるまで待ちます。これを利用してdelay_ms関数を実行します。

gucTimerTickはunsignedchar、つまり符号なし1バイトの変数です。そのため0～255までしか値を持つことができません。gucTimerTickの255は510ミリ秒に相当する時間です。gucTimerTickに一度だけの代入では約0.5秒しかディレイを実行できません。そのためこの関数では、ディレイ値iDelayを500ミリ秒ずつに切ってgucTimerTickに代入します。

iDelayが500より大きい場合は、gucTimerTickには250を代入してiDelayを500引いた値に置き換えます。そしてwhile（gucTimerTick>0）;を使ってgucTimerTickが0より大きい間、無限ループをしながら待機します。gucTimerTickが0になるとwhile文から抜け出します。 iDelayが500以下の場合、gucTimerTickはiDelayを2で割って代入してiDelayを0にします。そして同じようにwhile文を使ってgucTimerTickが0になるまで待ちます。この過程をwhile（iDelay>0）に入れることによりiDelayが0より大きい間繰り返しているので、iDelayが0になった時にwhile文が終わってdelay_ms関数が終了します。

```
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])
```

次の関数は、PlayMotion関数です。パラメータとしては、モーションのヘッダーファイルで宣言された変数がそのまま入ります。順番にフレーム数、軸数、モーターデータ、実行時間のデータ、ディレイデータです。ここでMotorData、PlayTimeData、DelayDataは配列の名前であり、配列の名前は、その配列の最初の要素へのポインタであることを注意してください。プログラムメモリに宣言された配列データは、pgm_read_word（）などの特別な関数を使用してデータを読み込む必要があります。

```
unsigned char i;
unsigned int uiCurrentFrame;
DrsPacket stSendPacket;
```

変数を宣言します。iとuiCurrentFrameはfor文の中で、それぞれのモーターがフレーム数ほど繰り返すために宣言された変数です。そしてstSendPacketは送信するS_JOGパケットが保存される領域です。

```
for(uiCurrentFrame=0;uiCurrentFrame<FrameCnt;uiCurrentFrame++){
                                        …
}
```

フレーム数分を繰り返すfor文です。for文の内部では、フレームを一つずつ進めながら、各フレームに対応するモーターの値と実行時間を読んでS_JOGパケットを構成してモーターに送り、実行時間とディレイを加えた値をdelay_ms関数に入れて、その分の時間を待ちます。 delay_ms関数が終了すると、次のフレームに進みます。

```
stSendPacket.ucPacketSize = MIN_PACKET_SIZE + CMD_S_JOG_STRUCT_SIZE * MotorCnt + 1;
stSendPacket.ucChipID = BROADCAST_ID;
stSendPacket.ucCmd = CMD_S_JOG;
```

現在のフレームで送信されるパケットの中でパケットサイズ、ID、命令を設定する部分です。パケットサイズはS_JOGのパケット構造に沿って、基本的に7バイト(MIN_PACKET_SIZE)にモーターあたり4バイト*モーターが加わって、実行時間の1バイトが追加されます。MotorCntが16の場合には、7+4*16+1になって72バイトのサイズのパケットをモーターに送信します。

```
stSendPacket.unData.stSJogData.ucPlayTime = (pgm_read_word(PlayTimeData+uiCurrentFrame)*10+56)/112;
```

S_JOGパケットの実行時間を設定する部分です。式が複雑ですが一段階ずつ説明します。まずpgm_read_word()は、プログラムメモリの値を読み取る関数であり、avr/pgmspace.hで宣言されます。パラメータには、読みプログラムメモリのアドレスが入ります。PlayTimeDataは、この関数のパラメータとして受け取った配列の名前であり、先に配列の名前は、配列の最初の要素のアドレスであることを述べました。さらにPlayTimeDataは&PlayTimeData[0]と同じです。このような原理で、(PlayTimeData+ i)は&PlayTimeData[i]と同じです。つまりpgm_read_wordにパラメータとして入れる(PlayTimeData+uiCurrentFrame)は&PlayTimeData[uiCurrentFrame]と同じです。したがって、pgm_read_word(PlayTimeData+uiCurrentFrame)はPlayTimeData[uiCurrentFrame]の値を読み出す部分です。

この式では、PlayTimeData[uiCurrentFrame]の値を読み出して、10を掛けて、56を加えて、112で割ります。これはPlayTimeData[uiCurrentFrame]を11.2で割った後に四捨五入している部分をただ定数の計算で表したものです。PlayTimeDataには実行時間がミリ秒単位で保存されているため、実際のS_JOGパケットに入れるときには11.2ミリ秒単位に分けて送信します。このとき強引に11.2に分割する方法もありますが、この場合実数演算になるので、AVRプロセッサの特性上、演算速度が定数演算よりも数十倍遅くなります。そのためPlayTimeData[uiCurrentFrame]/11.2の代わりに、11.2=112/10を利用してPlayTimeData[uiCurrentFrame]*10/112をすると二度の定数演算だけで11.2で割る実数演算を実行できるようになります。

しかし、そうする場合には、除算の結果の値のうち小数部分が自動的に消えるので、PlayTimeDataを読んで11.2で割った後、小数点以下を切り捨てたものと同じになります。つまり11.2で割った実行時間の値が6.0～6.9999…の範囲にあれば全て6になります。微細な誤差によってモーション実行時のロボットが意図せずに動作することもあります。これを解決するために、除算する前に56を加えた後に112で割ると、元の実行時間の値6.0～6.9999…の範囲が計算結果の値では6.5～7.4999…となって、元の実行時間基準6.0～6.4999…は6、6.5～6.9999…は7という結果が出てくるので、四捨五入したのと同じ効果が得られます。要約すると、上記のコードは、プログラムメモリからPlayTimeData[uiCurrentFrame]の値を読んで、11.2ミリ秒割った後に四捨五入してS_JOGパケットの実行時間に代入する部分です。

```
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;
}
```

iを0からMotorCnt-1まで増加させながら、S_JOGのモーターデータ部分を満たしていきます。現在必要なモーターの値はMotorData[uiCurrentFrame][i]です。pgm_read_wordでは、パラメータとしてポインタを必要とするので、&MotorData[uiCurrentFrame][i]を付け加える必要があります。&MotorData[uiCurrentFrame][i]はすぐに(*(MotorData+uiCurrentFrame)+ i)と同じです。なぜそうなのかは順を追って説明します。

MotorData[FrameCnt][MotorCnt]という2次元配列は、分けて考えるとMotorCnt個の定数の配列がFrameCnt個ある配列なので、MotorDataはポインタのポインタとすることができます。前に(PlayTimeData+uiCurrentFrame)が&PlayTimeData[uiCurrentFrame]と同じだったように、

(MotorData+uiCurrentFrame) = &MotorData[uiCurrentFrame]

と同じで、両側に＊を付けると

*(MotorData+uiCurrentFrame)が MotorData[uiCurrentFrame]

になります。MotorData[uiCurrentFrame]はucMotorCnt個のintの配列の開始を示すポインタであるため、この値にiを加えると、(PlayTimeData+ uiCurrentFrame)が&PlayTimeData[uiCurrentFrame]と同じだったように

(*(MotorData+uiCurrentFrame)+i) = &MotorData[uiCurrentFrame][i]

になります。したがって、pgm_read_word(*(MotorData+ uiCurrentFrame)+ i)は、フレーム番号uiCurrentFrameとモーター番号iに対応するモーターの値をプログラムメモリから読み出して、S_JOGパケットの対応する位置に入れる部分です。

その下の部分は、パケットの残りのフラグを設定する部分であり、あまり重要ではありません。最後のstSendPacket.unData.stSJogData.stSJog[i]. ucId= i;は制御するモーターのIDをiに代入する部分です。

```
hklx_SendPacket(stSendPacket);
```

hklx_SendPacket()関数を呼び出して作ったS_JOGパケットを送信します。

```
delay_ms(pgm_read_word(PlayTimeData+uiCurrentFrame)+pgm_read_word(DelayData+uiCurrentFrame));
```

次のフレームのパケットを送信するまで待機する部分です。pgm_read_word(PlayTimeData+ uiCurrentFrame)で、現在のフレームの実行時間をもう一度読み込み、pgm_read_word(DelayData+ uiCurrentFrame)で、現在のフレームのディレイを読んで来て二つを加えてdelay_ms関数に渡して、次のフレームのパケットを送信するまで待機します。
今までの過程をuiCurrentFrameが0のときからFrameCnt-1まで、すべてのフレームに対して繰り返すと、モーション実行がすべて終了して、PlayMotion関数が終了します。

ここからはmain関数の説明です。

```
DrsPacket stSendPacket;
```

メイン関数で使用する変数を宣言します。DrsPacket構造体タイプでパケットを送信するときに使う変数を宣言します。

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

ポートCの0～6番ピンを出力に設定し、出力値を1にします。これにより、すべてのLEDが消灯します。

```
cli();
hklx_Init(115200);
sei();
```

割り込みの設定を変える時に割り込みが発生して予期せぬ誤動作が発生することを防止するために、まずcli()命令を通して全体の割り込みを無効にします。その後hklx_Init()関数を使用してHerkuleXを制御するためのタイマー/カウンタ0とUSART0を初期化します。使用ボーレートが115200であるため、パラメータとしてその値を使用します。初期化が終わった後は、sei()命令を通して全体の割り込みを有効にします。

```
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);
```

モーターが自分で動くように制御するためにはモーターのトルクの認可が必要です。上記のコードがトークを認可させるためのRAM_WRITEパケットを構成して、送信する部分です。パケットのSIZEは基本MIN_PACKET_SIZE(7バイト)に加え、アドレス+長さ+データ(1バイト)で総MIN_PACKET_SIZE+3(10バイト)であり、接続されたすべてのモーターに送信するためのIDはBROADCAST_ID、つまり254です。 CMDはCMD_RAM_WRITE(0x03)であり、unData共用体のメンバーの中でstRWData構造体のucAddress、ucLen、ucData[0]の値をそれぞれ52、1、0x60にしています。その後はhklx_SendPacket()関数を呼び出してパケットを送信します。

```
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);
```

ここからwhile文の中の内容です。while文では、特別なことはなく、モーションを実行します。最初に前進開始モーション(walk_16_0)を実行して、その後の前進モーション(walk_16_1)を三回実行して、前進終了モーション(walk_16_2)を実行します。これにより、ロボットが10歩前進した後、再び停止します。

```
delay_ms(1000);
```

前進が終わった後、1秒間止まって待機します。

```
PlayMotion(box_16_FrameCnt, box_16_MotorCnt, box_16_MotorData, box_16_PlayTimeData, box_16_DelayData);
```

ボクシングモーション(box_16)を実行します。

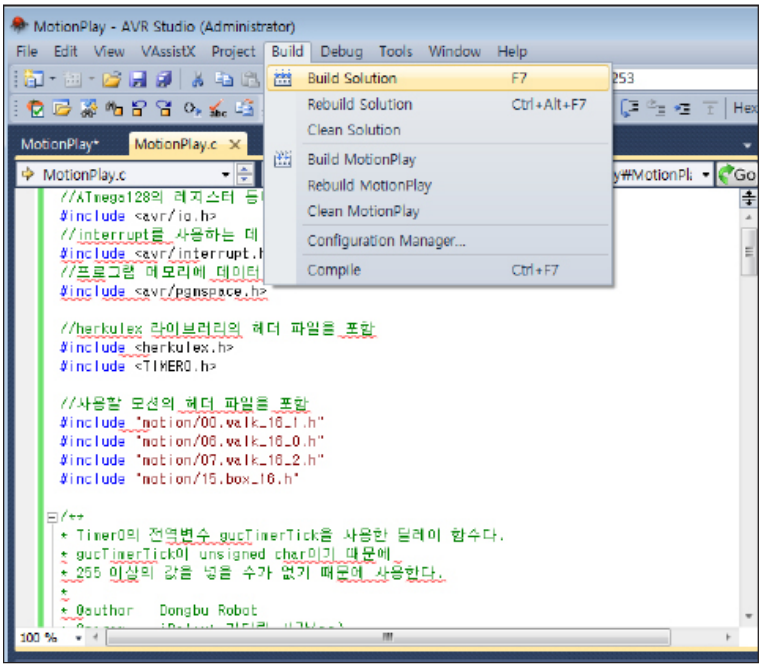
```
delay_ms(1000);
```

ボクシングモーションが終わった後、1秒間止まって待機しています。

while文では、上記の内容の繰り返しになっています。ロボットは10歩前進した後にボクシングモーションを繰り返します。

実行

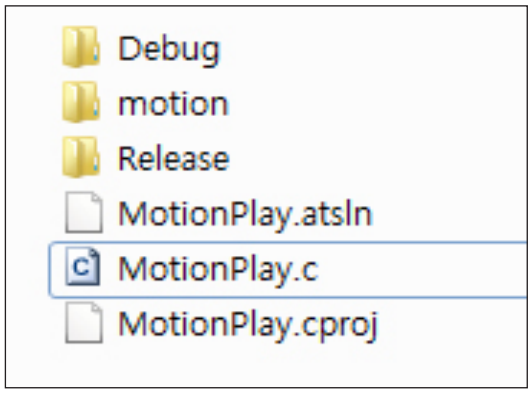
DRCを16軸Hovis Liteに接続し、ファームウェアをダウンロードして実行すると、ロボットが10歩前進した後にボクシングモーションを繰り返します。



01 Build

Buildメニューを選択しBuild Solutionをクリックしてビルドします。

1 succeeded0 failedが出るとエラーなしでコンパイルが完了したということです。



02 Debug フォルダ

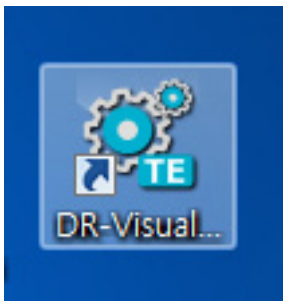
Build後にソースがあるフォルダに入るとDebugフォルダが作成されています。クリックしてフォルダの中身を見ます。



03 hex 確認

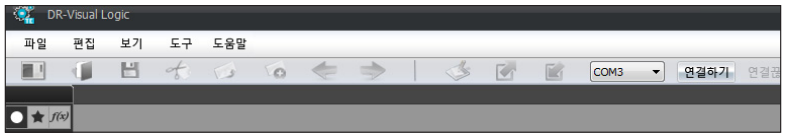
Debugフォルダの中のhexファイルを確認します。Hexファイルは制御機に移動するファイルです。

DR-Visual Logicでファームウェアの修復を行います。



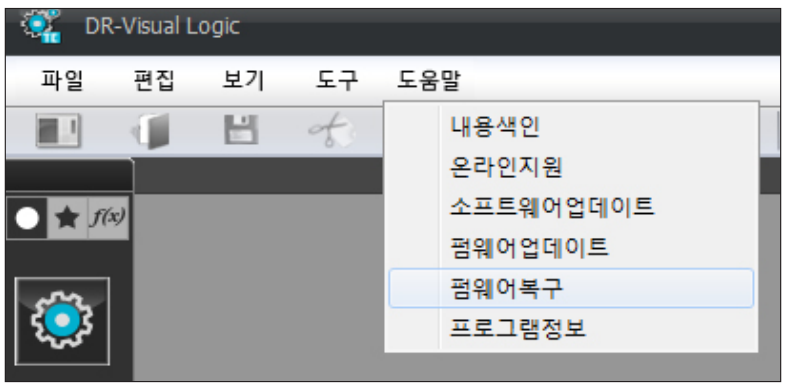
04 Visual Logic

DR-Visual Logicを実行します。



05 接続する

COM番号を設定して接続するをクリックします。



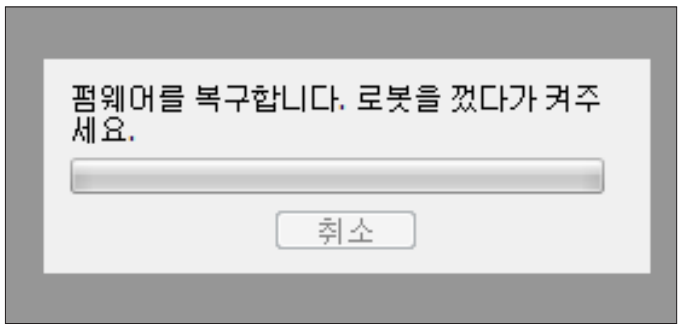
06 ファームウェアの修復

[ヘルプ]メニューからファームウェア修復をクリックします。



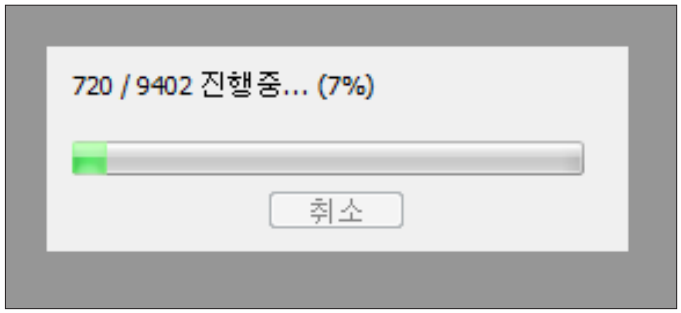
07 ロード

DebugフォルダにHexファイルをロードします。



08 ファームウェアの修復

“ロボットの電源を切って入れ直してください”というメッセージが出ます。制御機の電源ボタンを一度切ってから再度入れ直してください。



09 修復の開始

ファームウェアの修復が開始されます。



10 実行

DRCを16軸Hovis　Liteに接続し、ファームウェアをダウンロードして実行すると、ロボットが10歩前進した後にボクシングモーションを繰り返します。

CH.2 : 赤外線リモコンでダンプトラックを制御する

herkulexライブラリ、irremoconライブラリを使用してダンプトラックロボットを制御します。

H/W 原理

今回の例では、HerkuleXを制御するライブラリと赤外線リモコンを読み込むライブラリを同時に使用してダンプトラックロボットを制御します。 HerkuleX制御に関連した内容は、Chapter8を、赤外線リモコンに関連した内容は、Chapter12を参照してください。

今回の例で使用するパケットに使用された命令を要約すると以下の通りです。

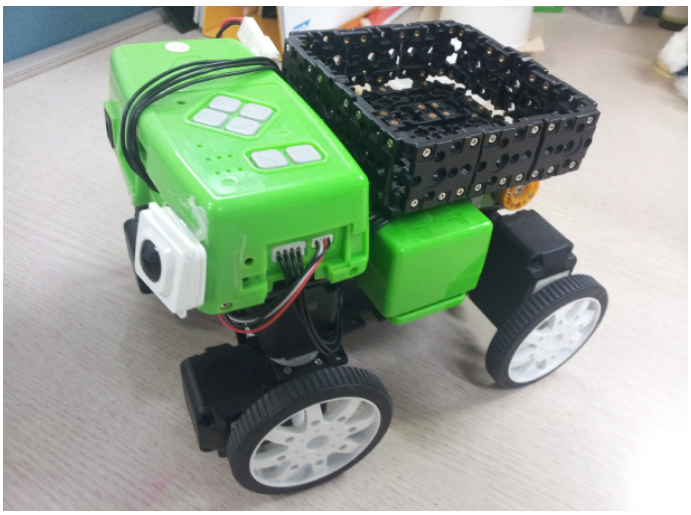
名	CMD	説明
RAM_WRITE	0x03	HerkuleXの揮発性レジスタ領域のデータを使用します。
I_JOG	0x05	HerkuleXの位置/速度を制御して動くようにします。各モーターごとに異なる 移動時間を設定できます。 I_JOGとS_JOGパケットで時間を表すためには、1バイトのデータを使用します。この時のデータは1バイトあたり11.2msを示しています。つまりI_JOGパケットに50の時間データが入っている場合、実際のモーターは50*11.2=560msの時間をかけて目標位置に移動します。

今回の例で使用した揮発性レジスタ領域の項目を要約すると以下の通りです。

名	アドレス	サイズ	説明
Torque Control	52	1	トークの認可の状態を制御します。 0x00:自由に動く力がない状態 0x40:動く力はないが、現在の位置を維持する状態 0x60:トルクが認可され自分で駆動する状態

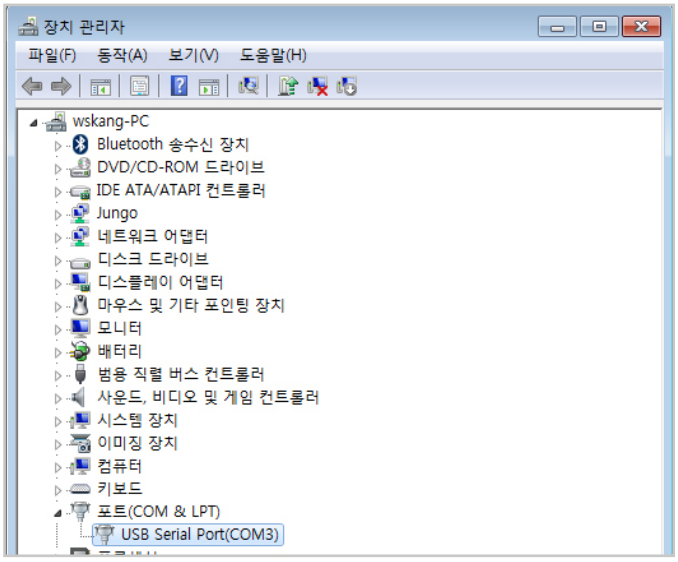
プロトコルとレジスタマップの詳細については、HerkuleXマニュアルを参照してください。

ハードウェアを準備する



01 接続する

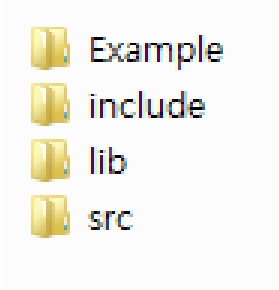
DRC 制御機のUSB to Serial CableにPCとDRCを接続します。
16軸ヒューマノイドHovis Liteを PCと接続します。



02 COM portを接続する

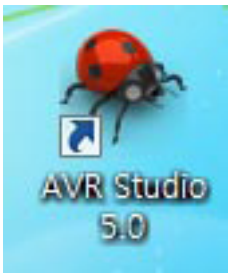
[マイコンピュータ]>[プロパティ]>[デバイスマネージャ]でCOM接続を確認します。

AVR Studio 実行とコーディングの準備



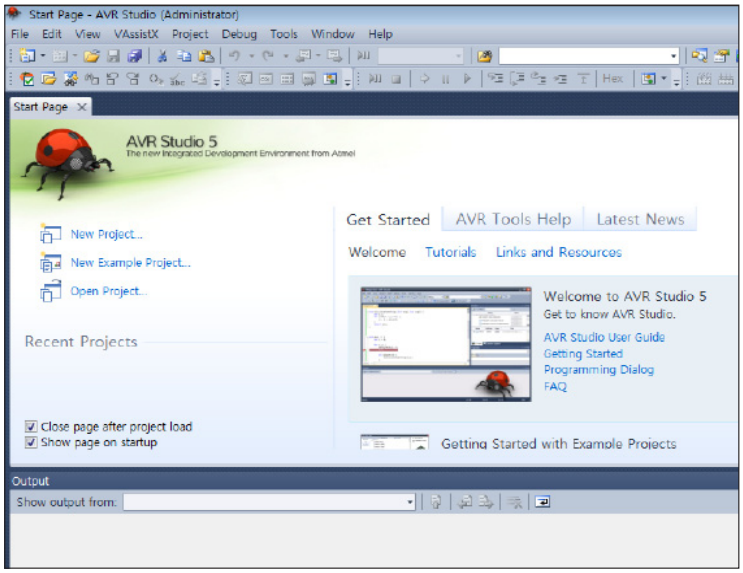
00 ライブラリ

ダンプトラックを制御するためには、基本ライブラリファイルが必要です。
ライブラリが含まれているフォルダ構造です。



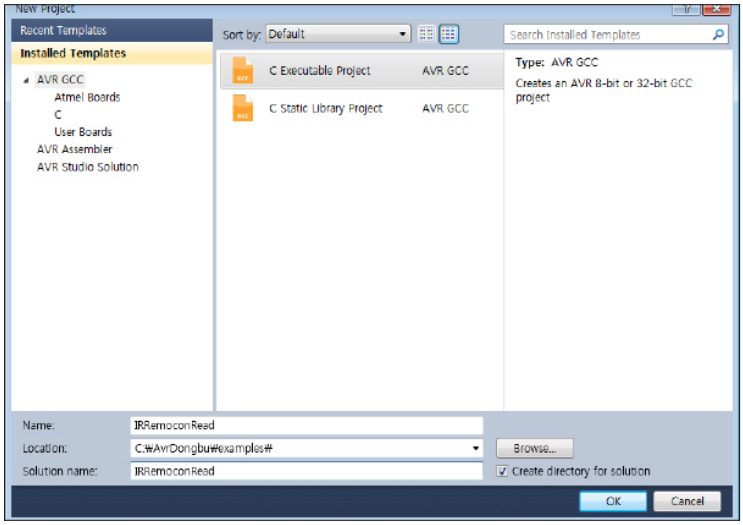
01 実行

AVR Studioを実行します。



02 New Project

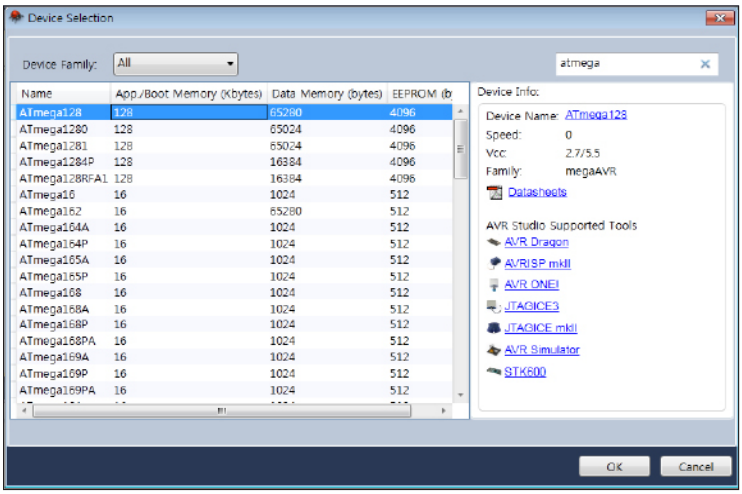
New Projectをクリックします。



03 GCC 選択

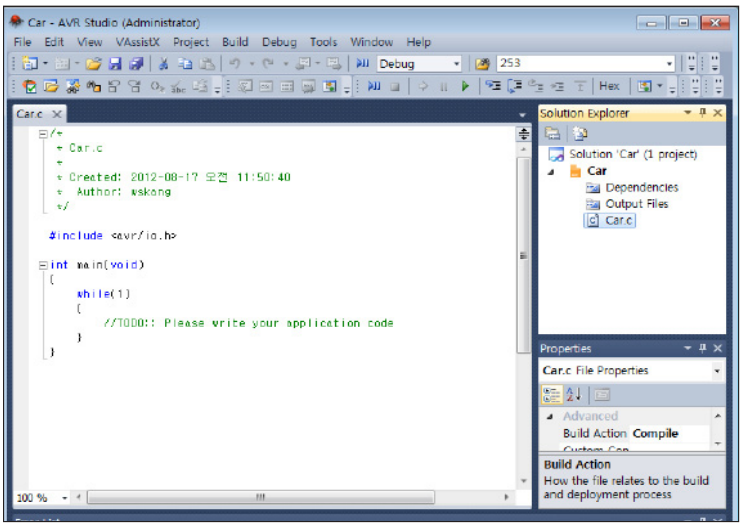
C Executable Projectを選択します。

Nameに“Car”と入力し、 Create directory for solution をクリックして、OK ボタンを押します。



04 ATmega 選択

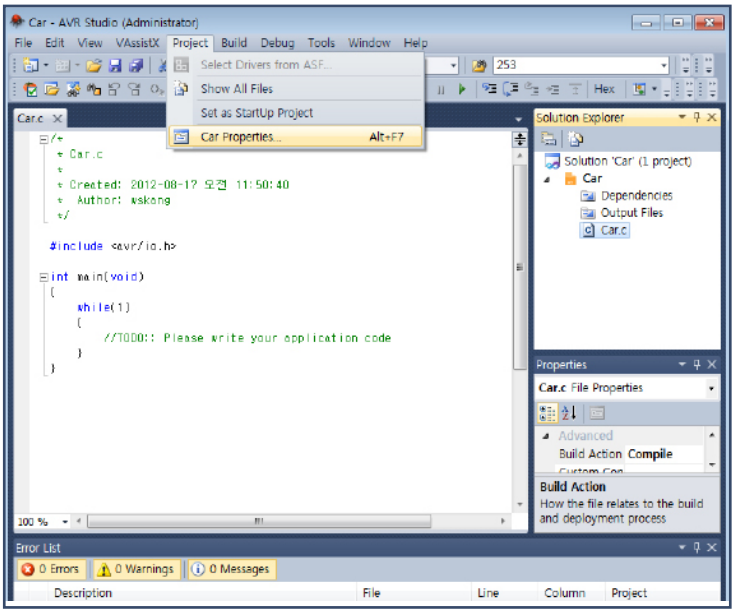
右上からatmegaを検索して、
リスト中のATmega128を選択します。



05 Editor

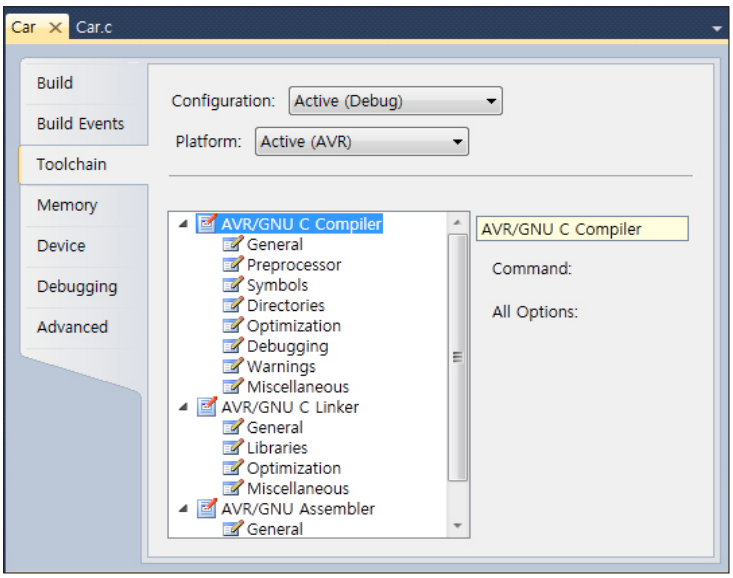
編集ウィンドウでコーディングを開始します。

ライブラリのリンクと設定



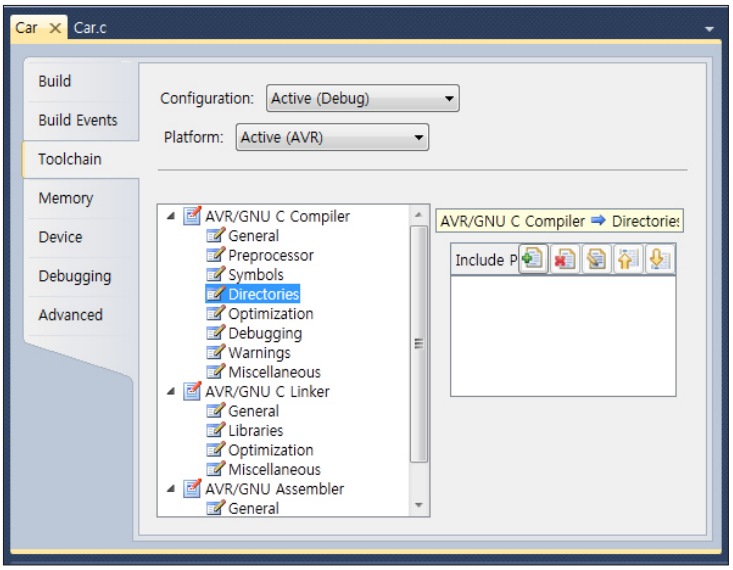
01 設定

AVR Studioの上部メニューの中にあるProjectをクリックすると、Car（プロジェクト名により異なります）Propertyメニューが生成されています。クリックすると、左側のように設定画面が出ます。



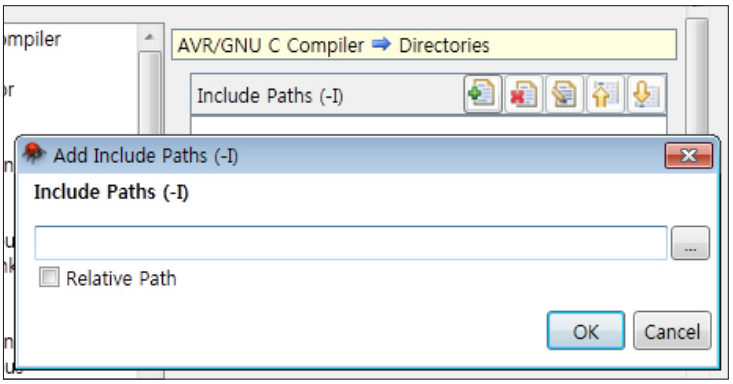
02 Toolchain

左側のタブボタンの中のToolchainをクリックして、AVR/ GNU C Compilerを選択します。



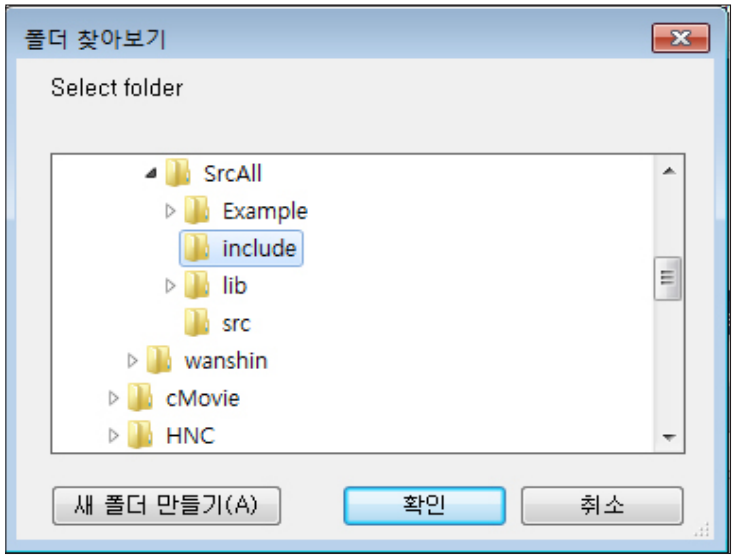
03 C Compiler

C　Compilerフォルダ中のDirectoriesを選択します。



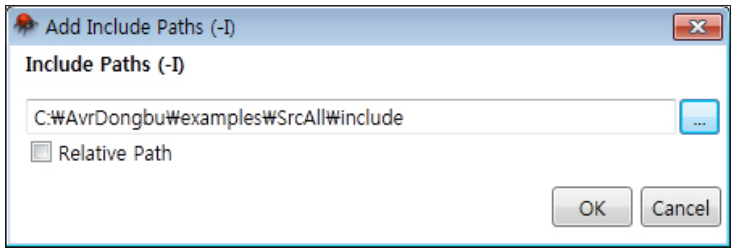
04 Path

Include　Pathsの右にある+ボタンをクリックすると、ポップアップ入力画面が表示されます。
Relative　Path（相対パス）のチェックを解除して、右側のフォルダを検索するボタンをクリックします。



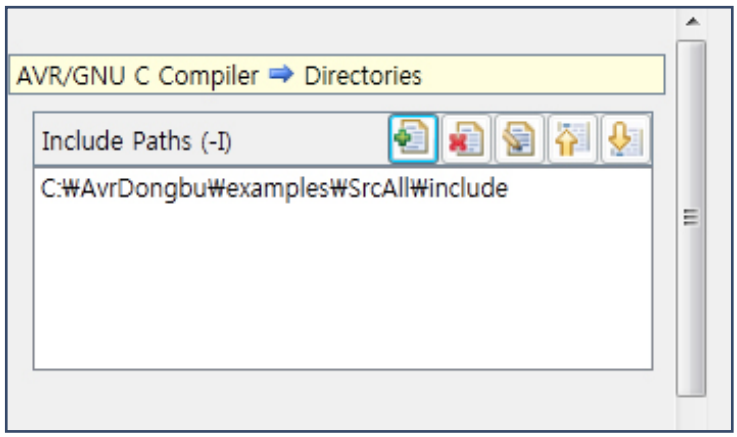
05 Include

提供ライブラリフォルダ中のincludeを選択します。



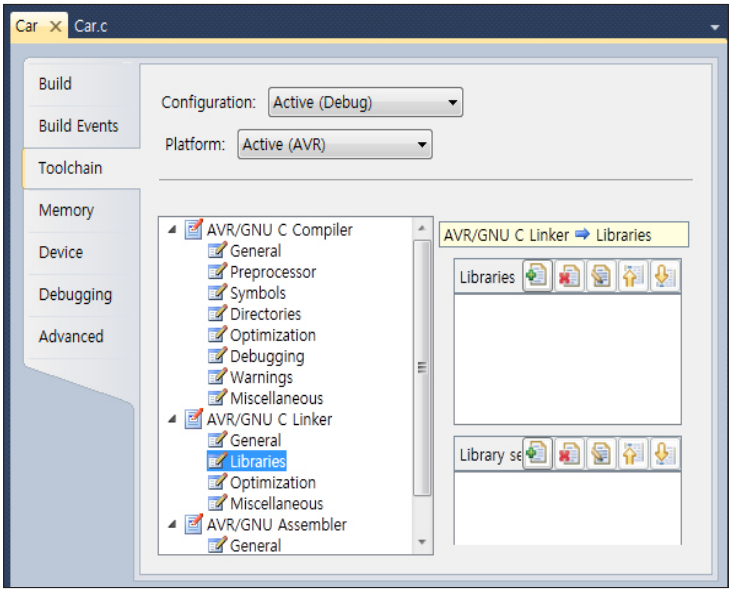
06 Include 完了

Includeフォルダの場所が指定されます。
OKボタンをクリックします。



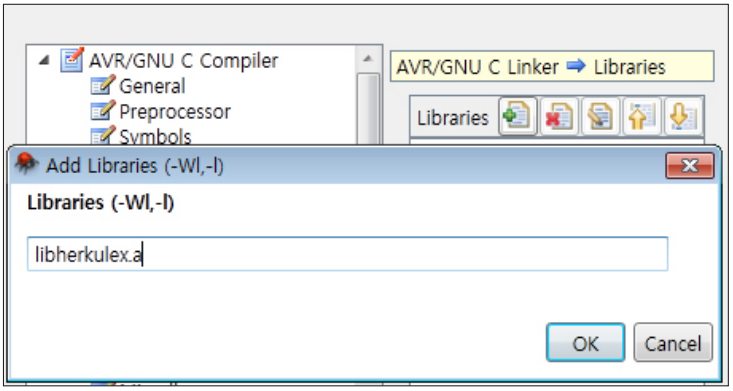
07 Path 完了

Includeパスの指定が完了しました。



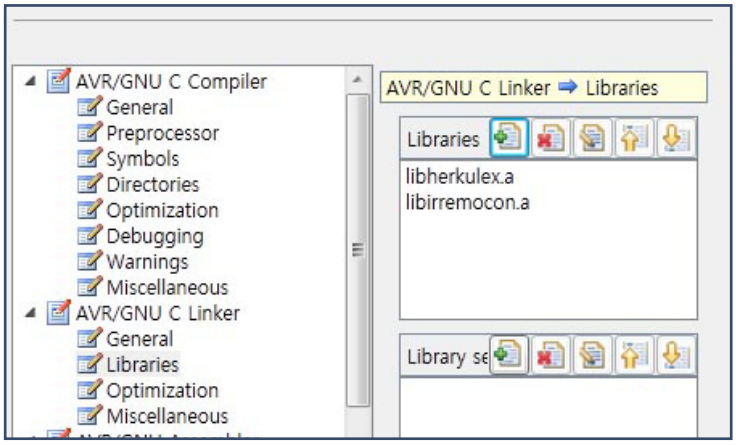
08 C Linker

C Linker フォルダ中のLibrariesを選択します。



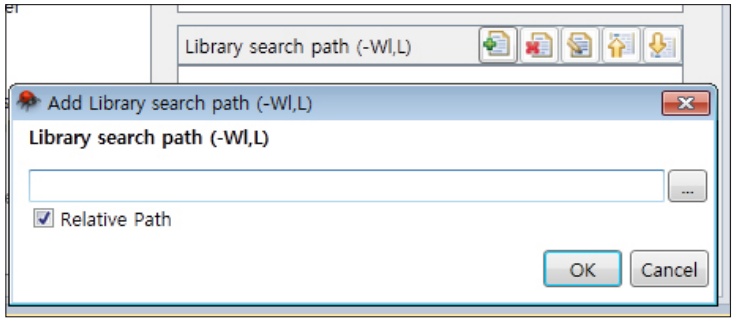
09 Libraries

Librariesウィンドウ右側にある+ボタンをクリックすると、Librariesファイル名入力画面が表示されます。libherkulex.aを直接入力します。
OKボタンをクリックします。



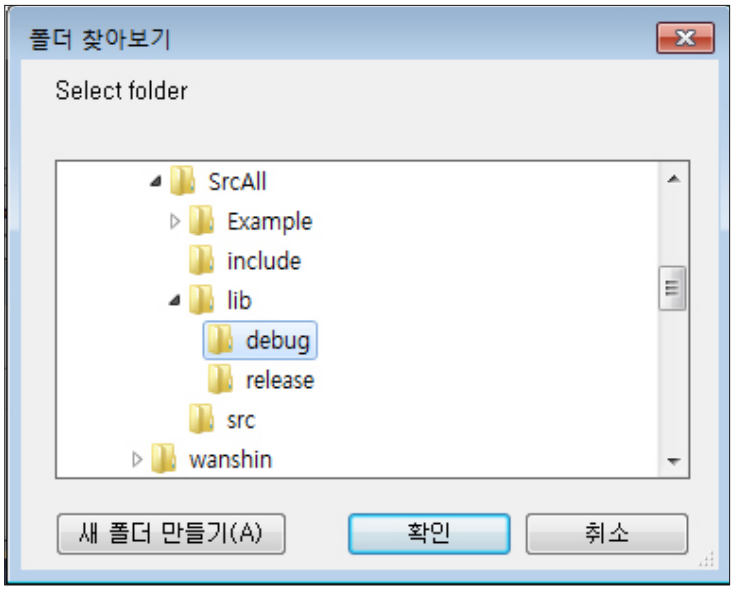
10 Search

サーボモーターのライブラリファイルlibherkulex.aリンクが完了しました。
次に、Library search pathを設定します。
右+ボタンをクリックします。



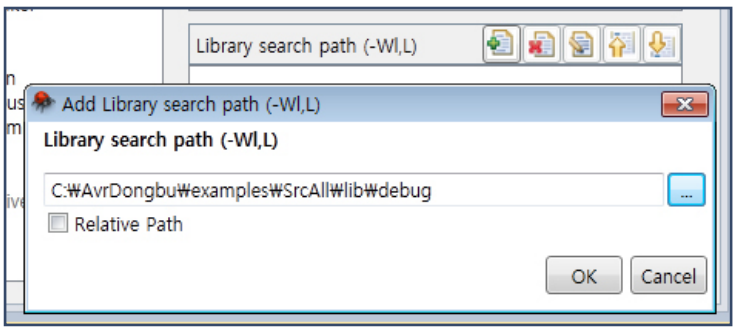
11 Path

Library search pathのポップアップ右側にあるフォルダの検索ボタンをクリックします。



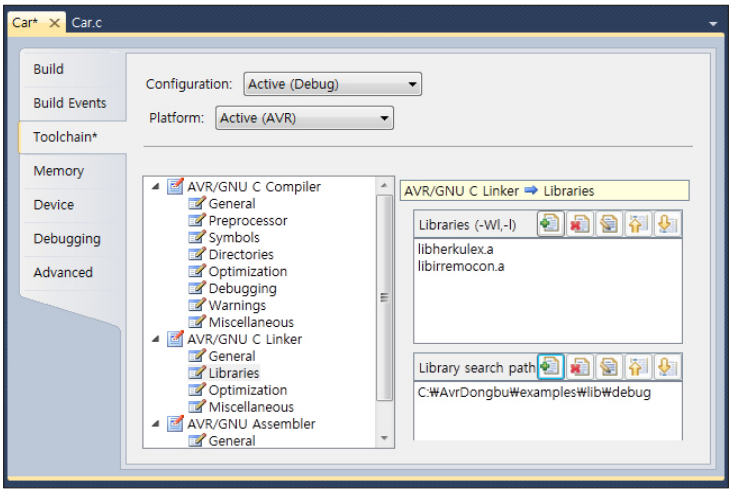
12 Debug

提供ライブラリフォルダにあるlibサブフォルダのdebugを選択します。



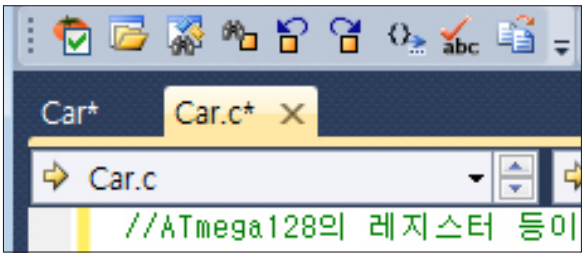
13 Path 完了

Relative Path(相対パス指定)チェックボックスを解除します。
Library search pathパスの指定が完了しました。
OKボタンをクリックします。



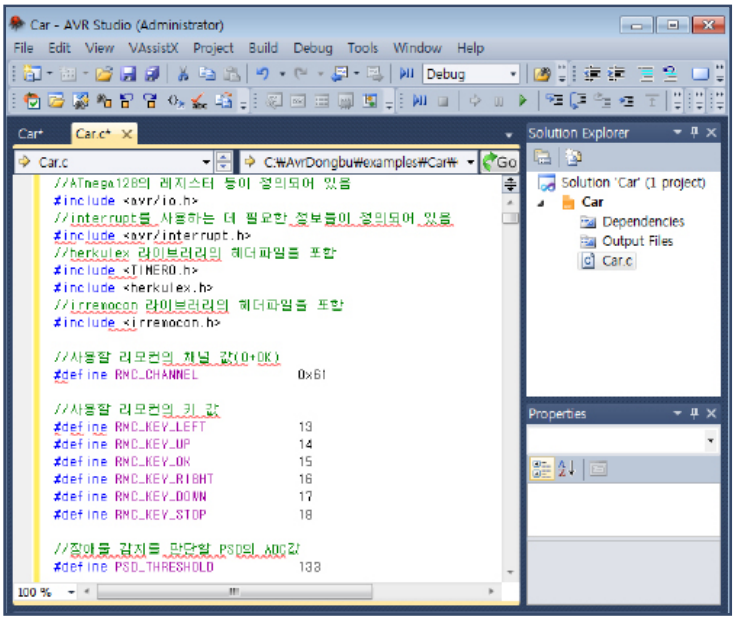
14 Link 完了

C Linker ライブラリファイルとパスの指定が完了しました。



15 保存

保存ボタンを押すと現在の設定値が保存されます。タブの名前が car* -> carlに変更されます。



16 コーディング

Car.c のタブを選択して、下の例を入力します。

例

リモコンでダンプトラックロボットを制御する例です。上部のキーを押すと前進、下のキーを押すと後進し、左右キーによって左折、右折します。OKをクリックするとダンプトラックの荷台を立てた後に元の位置に戻ります。動いている時に停止ボタンを押すと止まります。前方のPSDで障害物を感知して前進中に障害物に会うと所定の位置で右に回って再び前進します。

Car.c

```
//ATmega128のレジスタなどが定義されている
#include <avr/io.h>
//interruptを使用するために必要な情報が定義されている
#include <avr/interrupt.h>
//herkulex ライブラリのヘッダファイルを含む
#include <TIMER0.h>
#include <herkulex.h>
//irremocon ライブラリのヘッダファイルを含む
#include <irremocon.h>

//使用するリモコンのチャンネル値(0+OK)
#define RMC_CHANNEL                     0x61

//使用するリモコンのキー値
#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

//障害物の検出を判断するPSDのADC値
#define PSD_THRESHOLD           133

//I_JOG パケットで逆方向回転ビットを示すマスク
#define INVERSE_TURN_MASK               0x4000

/**
* Timer0のグローバル変数gucTimerTickを使用したディレイ関数である。
* gucTimerTickがunsigned charであるため、
*255以上の値を入れることができないので注意する。
*
* @author       Dongbu Robot
* @param        iDelay: 待ち時間(ms)
* @return       なし
* @version      1.0.0.0
*/
```

```
void delay_ms(int iDelay){
	//iDelayが0より大きい場合、繰り返す
	while(iDelay>0){
		//iDelayが500msより大きい場合
		if(iDelay>500){
			//gucTimerTickに250を入れてiDelayから500引く。
			gucTimerTick=250;
			iDelay -= 500;
			//Timer0によってgucTimerTickが0になるまで 待機
			while(gucTimerTick>0);
		}
		//iDelayが 500ms以下の場合
		else{
			//gucTimerTickにiDelayを2で割って入れてiDelayを0にする。
			gucTimerTick=iDelay/2;
			iDelay = 0;
			//Timer0によってgucTimerTickが0になるまで待機
			while(gucTimerTick>0);
		}
	}
}

/**
* ダンプトラックを停止させる関数である。
*車輪サーボ(0、2、4、5番)は速度制御に0を送って
*ステアリングサーボ(1、3番)は正面を眺めるようにして
*荷台サーボ(6番)は、元の位置に戻す
* I_JOGパケットを構成してサーボに送る。
*
* @author		Dongbu Robot
* @param		なし
* @return		なし
* @version		1.0.0.0
*/
void stop(void){
	//送信するパケットが保存されるパケットの構造体宣言
	DrsPacket stSendPacket;
	//for文で使用する変数の宣言
	unsigned char i;
	//I_JOG パケットのサイズ、ID、CMD設定
	stSendPacket.ucPacketSize = MIN_PACKET_SIZE + 7 * CMD_I_JOG_STRUCT_SIZE;
	stSendPacket.ucChipID = BROADCAST_ID;
	stSendPacket.ucCmd = CMD_I_JOG;
```

MotionPlay.c

```
        //7個のサーボに共通のパラメータ設定+ ucId設定
        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;
        }

        //00番サーボ(左の前輪)を停止(速度制御)
        stSendPacket.unData.stIJogData.stIJog[0].stJog.uiValue = 0;
        stSendPacket.unData.stIJogData.stIJog[0].stSet.ucMode = 1;
        stSendPacket.unData.stIJogData.stIJog[0].ucPlayTime = 20;

        //1番サーボ(左のステアリング)を直進方向に(位置制御)
        stSendPacket.unData.stIJogData.stIJog[1].stJog.uiValue = 789;
        stSendPacket.unData.stIJogData.stIJog[1].stSet.ucMode = 0;
        stSendPacket.unData.stIJogData.stIJog[1].ucPlayTime = 150;

        //2番サーボ(右の前輪)を停止(速度制御)
        stSendPacket.unData.stIJogData.stIJog[2].stJog.uiValue = 0;
        stSendPacket.unData.stIJogData.stIJog[2].stSet.ucMode = 1;
        stSendPacket.unData.stIJogData.stIJog[2].ucPlayTime = 20;

        //3番サーボ(右のステアリング)を直進方向に(位置制御)
        stSendPacket.unData.stIJogData.stIJog[3].stJog.uiValue = 235;
        stSendPacket.unData.stIJogData.stIJog[3].stSet.ucMode = 0;
        stSendPacket.unData.stIJogData.stIJog[3].ucPlayTime = 150;

        //4番サーボ(左の後輪)を停止(速度制御)
        stSendPacket.unData.stIJogData.stIJog[4].stJog.uiValue = 0;
        stSendPacket.unData.stIJogData.stIJog[4].stSet.ucMode = 1;
        stSendPacket.unData.stIJogData.stIJog[4].ucPlayTime = 20;

         //5番サーボ(右の後輪)を停止(速度制御)
        stSendPacket.unData.stIJogData.stIJog[5].stJog.uiValue = 0;
        stSendPacket.unData.stIJogData.stIJog[5].stSet.ucMode = 1;
        stSendPacket.unData.stIJogData.stIJog[5].ucPlayTime = 20;

        //6番サーボ(荷台)を基本場所に(位置制御)
        stSendPacket.unData.stIJogData.stIJog[6].stJog.uiValue = 970;
        stSendPacket.unData.stIJogData.stIJog[6].stSet.ucMode = 0;
        stSendPacket.unData.stIJogData.stIJog[6].ucPlayTime = 100;

        //パケットを送信する
        hklx_SendPacket(stSendPacket);
}
```

```
/**
* 荷台サーボ(6番)を基本の位置に送る
* I_JOGパケットを構成してサーボに送る
*
* @author          Dongbu Robot
* @param           なし
* @return          なし
* @version         1.0.0.0
*/
void load_on(void){
        //送信するパケットが保存されるパケットの構造体宣言
        DrsPacket stSendPacket;

        //I_JOGパケットのサイズ、ID、CMD設定
        stSendPacket.ucPacketSize = MIN_PACKET_SIZE + CMD_I_JOG_STRUCT_SIZE;
        stSendPacket.ucChipID = BROADCAST_ID;
        stSendPacket.ucCmd = CMD_I_JOG;

        //6番サーボ(荷台)の位置制御時に基本位置にするために送信するパラメータ設定
        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;

        //パケットを送信する
        hklx_SendPacket(stSendPacket);
}

/**
* 荷台サーボ(6番)を荷物が降る場所に送る
* I_JOGパケットを構成してサーボに送る。
*
* @author          Dongbu Robot
* @param           なし
* @return          なし
* @version         1.0.0.0
*/
void load_off(void){
        //送信するパケットが保存されるパケットの構造体宣言
        DrsPacket stSendPacket;

        //I_JOG パケットのサイズ、ID、CMD設定
        stSendPacket.ucPacketSize = MIN_PACKET_SIZE + CMD_I_JOG_STRUCT_SIZE;
        stSendPacket.ucChipID = BROADCAST_ID;
        stSendPacket.ucCmd = CMD_I_JOG;
```

```
	//6番サーボ(荷台)を荷物が降りる場所に位置制御するために送信するパラメータ設定
	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;

	//パケットを送信する
	hklx_SendPacket(stSendPacket);
}

/**
* ステアリングサーボ(1、3番)の方向を変更して左折するように
* I_JOGパケットを構成してサーボに送る。
*
* @author		Dongbu Robot
* @param		なし
* @return		なし
* @version		1.0.0.0
*/
void left_turn(void){
	//送信するパケットが保存されるパケットの構造体宣言
	DrsPacket stSendPacket;

	//I_JOG パケットのサイズ、ID、CMD設定
	stSendPacket.ucPacketSize = MIN_PACKET_SIZE + 2 * CMD_I_JOG_STRUCT_SIZE;
	stSendPacket.ucChipID = BROADCAST_ID;
	stSendPacket.ucCmd = CMD_I_JOG;

	//1番サーボ(左のステアリング)を左折場所に位置制御するために送信するパラメータ設定
	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;

	//3番サーボ(右ステアリング)を左折場所に位置制御するために送信するパラメータ設定
	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;

	//パケットを送信する
	hklx_SendPacket(stSendPacket);
}

/**
* ステアリングサーボ（1、3番）の方向を変更して右折するように
* I_JOGパケットを構成してサーボに送る。
*
* @author		Dongbu Robot
* @param		なし
* @return		なし
* @version		1.0.0.0
*/
void right_turn(void){
	//送信するパケットが保存されるパケットの構造体宣言
	DrsPacket stSendPacket;

	//I_JOGパケットのサイズ、ID、CMD設定
	stSendPacket.ucPacketSize = MIN_PACKET_SIZE + 2 * CMD_I_JOG_STRUCT_SIZE;
	stSendPacket.ucChipID = BROADCAST_ID;
	stSendPacket.ucCmd = CMD_I_JOG;

	//1番サーボ（左のステアリング）を右折位置に位置制御するために送信するパラメータ設定
	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;

	//3番サーボ（右のステアリング）を右折位置に位置制御するために送信するパラメータ設定
	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;
```

```
		//パケットを送信する
		hklx_SendPacket(stSendPacket);
}

/**
* ダンプトラックを前進させる関数である。
*車輪サーボ（0、2、4、5番）は、前に転がって行くようにして
*ステアリングサーボ（1、3番）は正面を眺めるようにする
* I_JOGパケットを構成してサーボに送る。
*
* @author		Dongbu Robot
* @param		なし
* @return		なし
* @version		1.0.0.0
*/
void forward(void){
		//送信パケットと受信パケットが保存されるパケットの構造体宣言
		DrsPacket stSendPacket;
		//for文で使用する変数の宣言
		unsigned char i;

		//I_JOG パケットのサイズ、ID、CMD設定
		stSendPacket.ucPacketSize = MIN_PACKET_SIZE + 6 * CMD_I_JOG_STRUCT_SIZE;
		stSendPacket.ucChipID = BROADCAST_ID;
		stSendPacket.ucCmd = CMD_I_JOG;

		//6個のサーボに共通のパラメータ設定+ ucId設定
		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;
		}

		//0番サーボ（左の前輪）を前進方向に回転（速度制御）
		stSendPacket.unData.stIJogData.stIJog[0].stJog.uiValue = 800;
		stSendPacket.unData.stIJogData.stIJog[0].stSet.ucMode = 1;
		stSendPacket.unData.stIJogData.stIJog[0].ucPlayTime = 10;

		//1番サーボ（左のステアリング）を直進方向に（位置制御）
		stSendPacket.unData.stIJogData.stIJog[1].stJog.uiValue = 789;
		stSendPacket.unData.stIJogData.stIJog[1].stSet.ucMode = 0;
		stSendPacket.unData.stIJogData.stIJog[1].ucPlayTime = 150;

		//2番サーボ（右の前輪）を前進方向に回転（速度制御）
		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;

		//3番サーボ（右のステアリング）を直進方向に（位置制御）
		stSendPacket.unData.stIJogData.stIJog[3].stJog.uiValue = 235;
		stSendPacket.unData.stIJogData.stIJog[3].stSet.ucMode = 0;
		stSendPacket.unData.stIJogData.stIJog[3].ucPlayTime = 150;
```

```
	//4番サーボ(左の後輪)を前進方向に回転(速度制御)
	stSendPacket.unData.stIJogData.stIJog[4].stJog.uiValue = 800;
	stSendPacket.unData.stIJogData.stIJog[4].stSet.ucMode = 1;
	stSendPacket.unData.stIJogData.stIJog[4].ucPlayTime = 10;

	//5番サーボ(右後輪)を前進方向に回転(速度制御)
	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;

	//パケットを送信する
	hklx_SendPacket(stSendPacket);
}

/**
* ダンプトラックを後退させる関数である。
*車輪サーボ(0、2、4、5番)は後ろに転がって行くようにして
*ステアリングサーボ(1、3番)は正面を眺めるようにする
* I_JOGパケットを構成してサーボに送る。
*
* @author		Dongbu Robot
* @param		なし
* @return		なし
* @version		1.0.0.0
*/
void backward(void){
	//送信するパケットが保存されるパケットの構造体宣言
	DrsPacket stSendPacket;
	//for文で使用する変数の宣言
	unsigned char i;

	//I_JOG パケットのサイズ、ID、CMD設定
	stSendPacket.ucPacketSize = MIN_PACKET_SIZE + 6 * CMD_I_JOG_STRUCT_SIZE;
	stSendPacket.ucChipID = BROADCAST_ID;
	stSendPacket.ucCmd = CMD_I_JOG;

	//6個のサーボに共通のパラメータ設定+ ucId設定
	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;
	}

	//0番サーボ(左の前輪)を後退方向に回転(速度制御)
	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;
```

```
	//1番サーボ(左のステアリング)を直進方向に(位置制御)
	stSendPacket.unData.stIJogData.stIJog[1].stJog.uiValue = 789;
	stSendPacket.unData.stIJogData.stIJog[1].stSet.ucMode = 0;
	stSendPacket.unData.stIJogData.stIJog[1].ucPlayTime = 150;

	//2番サーボ(右の前輪)を後退方向に回転(速度制御)
	stSendPacket.unData.stIJogData.stIJog[2].stJog.uiValue = 800;
	stSendPacket.unData.stIJogData.stIJog[2].stSet.ucMode = 1;
	stSendPacket.unData.stIJogData.stIJog[2].ucPlayTime = 10;

	//3番サーボ(右のステアリング)を直進方向に(位置制御)
	stSendPacket.unData.stIJogData.stIJog[3].stJog.uiValue = 235;
	stSendPacket.unData.stIJogData.stIJog[3].stSet.ucMode = 0;
	stSendPacket.unData.stIJogData.stIJog[3].ucPlayTime = 150;

	//4番サーボ(左の後輪)を後退方向に回転(速度制御)
	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;

	//5番サーボ(右の後輪)を後退方向に回転(速度制御)
	stSendPacket.unData.stIJogData.stIJog[5].stJog.uiValue = 800;
	stSendPacket.unData.stIJogData.stIJog[5].stSet.ucMode = 1;
	stSendPacket.unData.stIJogData.stIJog[5].ucPlayTime = 10;

	//パケットを送信する
	hklx_SendPacket(stSendPacket);
}

/**
* 障害物があった時に所定の位置で急速に回転するようにする関数である。
*車輪サーボ(0、2、4、5番)のうち左の車輪は前に、
*右の車輪は後方に転がるようにして右に回る
* I_JOGパケットを構成してサーボに送る。
*
* @author		Dongbu Robot
* @param		なし
* @return		なし
* @version		1.0.0.0
*/
void quick_turn(void){
	//送信するパケットが保存されるパケットの構造体宣言
	DrsPacket stSendPacket;
	//for文で使用する変数の宣言
	unsigned char i;

	//I_JOG パケットのサイズ、ID、CMD設定
	stSendPacket.ucPacketSize = MIN_PACKET_SIZE + 4 * CMD_I_JOG_STRUCT_SIZE;
	stSendPacket.ucChipID = BROADCAST_ID;
	stSendPacket.ucCmd = CMD_I_JOG;
```

```
		//4個のサーボに共通のパラメータ設定（0、3回前進方向、2、5回後進方向）
		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 設定
		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;

		//パケットを送信する
		hklx_SendPacket(stSendPacket);
}

int main(void)
{
		//送信するパケットが保存されるパケットの構造体宣言
		DrsPacket stSendPacket;
		//8ビット変数の宣言
		unsigned char ucData, ucDataOld, ucLength, ucForward;
		//16ビット変数の宣言
		int iADC;

		//ポートCの入出力方向の設定
		DDRC = 0b01111111;
		//ポートCの出力値の設定
		PORTC = 0b01111111;

		//VREF=AREF 入力電圧は、左側の配置をオフ、ADC6番単極性入力
		ADMUX=0b00000110;
		//ADC有効に、ADIFの初期化、クロック分周比128
		ADCSRA=0b10010111;

		//全体の割り込みを無効に
		cli();

		//HerkuleXを使用するために初期化
		hklx_Init(115200);

		//リモコン受信モジュールを使用するために初期化
		RMC_Init();

		//全体の割り込みを有効に
		sei();
```

```
//Torque Controlに0x60を使ってトルクをかけるパケットの構成
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;

//パケットを送信する
hklx_SendPacket(stSendPacket);

//ucDataOldを基本的な値で初期化
ucDataOld = RMC_ERROR_DATA;

//ucForwardを0に初期化
ucForward = 0;

//while内の内容を無限に繰り返す
while(1)
{
        //ADC 変換を開始
        ADCSRA |= (1<<ADSC);
        //ADC変換完了まで待つ
        while(!(ADCSRA & (1<<ADIF)));
        //ADCLとADCH値を読んで保存
        iADC = ADCL | (ADCH<<8);
        //ADIF 初期化
        ADCSRA |= (1<<ADIF);

        // 前進中に障害物があった場合、(0.65/5*1024 = 133.12)
        if(iADC > PSD_THRESHOLD && ucForward == 1){
                //しばらく停止し所定の位置で1秒間回転した後に前進
                stop();
                delay_ms(500);
                quick_turn();
                delay_ms(1000);
                forward();
                ucForward = 1;
        }

        //ucData, ucLengthにリモコン値の読み出し
        RMC_ReadRemocon(RMC_CHANNEL, &ucData, &ucLength);

        //リモコン値が前の値と異なっている場合
        if(ucDataOld != ucData){
                //UPキーが押された時に前進
                if(ucData == RMC_KEY_UP){
                        forward();
                        ucForward = 1;
                }
                //DOWNキーが押された時に後進
                else if(ucData == RMC_KEY_DOWN){
                        backward();
                        ucForward = 0;
```

```
                    }
                    //LEFT キーが押された時に左側に方向を変更する
                    else if(ucData == RMC_KEY_LEFT){
                            left_turn();
                    }
                    //RIGHT キーが押された時に右側に方向を変更する
                    else if(ucData == RMC_KEY_RIGHT){
                            right_turn();
                    }
                    //OK キーが押された時に荷台を立てて元に戻す
                    else if(ucData == RMC_KEY_OK){
                            load_off();
                            delay_ms(1000);
                            load_on();
                    }
                    //停止キーが押された時に停止
                    else if(ucData == RMC_KEY_STOP){
                            stop();
                            ucForward = 0;
                    }
            }

            //現在のデータ値を他の変数に保存
            ucDataOld = ucData;
    }

    //関数の形に合わせて定数値を返す
    return 1;
}
```

例の説明

Car.cで#include文を使用して含ませたヘッダファイル中libherkulex.aに関連するTIMER0.hとherkulex.hの説明はChapter8. HerkuleXを制御するを参照してください。また、libirremocon.aに関連するirremocon.hの説明はChapter12. 赤外線リモコン信号の読み取りを参照してください。

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

avr/ io.hヘッダファイルには、DDRX、PORTXレジスタなどAVR制御のために必要なレジスタなどが定義されていて、avr/interrupt.hヘッダファイルには、割り込みを使用するための関数が定義されています。

```
#include <herkulex.h>
#include <TIMER0.h>
#include <irremocon.h>
```

herkulex.hヘッダファイルには、パケットを構成して送信するための宣言と構造体、関数が定義されているので、ぜひ含める必要があります。そしてgucTimerTickを使ってdelayを実行するために、TIMER0.hも含めます。 irremocon.hには赤外線リモコン受信モジュールを初期化し、リモコンデータを読み込むための関数が定義されているので追加します。

```
#define RMC_CHANNEL                    0x61
```

リモコンを設定するチャネルの値は0x61から0x6Aまでです。今回の例では、リモコンの0番ボタン+ OKを押したときに設定されている基本的な値である0x61をチャンネル値として使用します。

```
#define RMC_KEY_LEFT                   13
                    …
#define RMC_KEY_STOP                   18
```

使用するリモコンのキーとそれに対応するキーの値を宣言します。上下左右の方向キーとOK、そして赤の停止キーを使用しています。キーとデータの値を宣言しておけば、キーの値を毎回調べる必要がなく便利です。

```
#define PSD_THRESHOLD                  133
```

PSDの値を読んで障害物が前にあると判断するときの基準値です。Chapter6のH/ Wの原則に出てきた距離とPSD出力値のグラフ上で、距離20センチでPSDの電圧は約0.65Vになります。0.65VはADC値に変換すると、0.65/5*1024=133.12なので133を基準にしています。これよりも距離が近いと0.65Vよりも高い電圧が出力されるので、133以上の値がADCから読み込まれ、20センチ以内に障害物があります。

```
#define INVERSE_TURN_MASK              0x4000
```

I_JOGとS_JOGを使用してサーボを速度制御する際には正方向回転と逆方向回転ができます。逆方向回転時には、stJog構造体のuiValueに速度値に加えて14番のビットを1に設定します。そこで上記のようマスクを宣言して速度値とマスクを一緒にBitwise OR演算してuiValueに代入します。

```
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);
                }
        }
}
```

TIMER0を利用したディレイ関数です。パラメータとしては、iDelayTickがあり、どれだけの時間待機するかどうかをミリ秒（ms）単位で表したものです。gucTimerTickはTIMER0.hで宣言されたグローバル変数で、普段は0に維持されて0以外の値を代入すると、libherkulex.aに定義されたTIMER0オーバーフロー割り込みサービスルーチンによって2ミリ秒ごとに1ずつ減少します。例えば、10ミリ秒を待ちたい場合はgucTimerTickに5という値を入れてgucTimerTickが0になるまで待ちます。これを利用してdelay_ms()関数を実行します。

gucTimerTickはunsigned　char、つまり符号なし1バイトの変数です。そのため0～255までしか値を持つことができません。gucTimerTickの255は510ミリ秒に相当する時間です。gucTimerTickに一度代入しただけでは約0.5秒しかディレイを実行することができません。そのためこの関数では、ディレイ値iDelayを500ミリ秒ずつに切ってgucTimerTickに代入します。

iDelayが500より大きい場合は、gucTimerTickには250を代入してiDelayを500引いた値に再代入します。そしてwhile(gucTimerTick>0);を使ってgucTimerTickが0より大きい間、無限に繰り返しながら待機します。gucTimerTickが0になるとwhile文から出ます。iDelayが500より小さいか等しい場合はgucTimerTickはiDelayを2で割って代入してiDelayを0にします。そして同じようにwhile文を使ってgucTimerTickが0になるまで待ちます。この過程をwhile(iDelay>0)に入れることでiDelayが0より大きい間繰り返し、iDelayが0になった時にwhile文が終了しdelay_ms()関数が終了します。

```
void stop(void){
…
}
```

次の関数は、stop()関数です。ダンプトラックを停止するI_JOGパケットをHerkuleXに送信する関数です。

```
DrsPacket stSendPacket;
unsigned char i;
```

変数を宣言します。stSendPacketは送信するI_JOGパケットが保存される領域です。iはfor文を制御するモーターの数だけ繰り返すために宣言された変数です。

```
stSendPacket.ucPacketSize = MIN_PACKET_SIZE + 7 * CMD_I_JOG_STRUCT_SIZE;
stSendPacket.ucChipID = BROADCAST_ID;
stSendPacket.ucCmd = CMD_I_JOG;
```

パケットのサイズとID、CMDを設定する部分です。パケットサイズはI_JOGのパケット構造に応じて、基本7バイト（MIN_PACKET_SIZE）にモーターあたり5バイト*モーターが加算されます。この関数では、ID0～6の7個のモーターを制御するため、パケットサイズは42になります。I_JOGパケットなので、IDはBROADCAST_ID（0xFE）、CMDはCMD_I_JOG（0x05）です。

```
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命令は、様々な設定ができるパラメータが多いので、モーターに一つずつパラメータの設定をすると、コードが長くなります。そのため値が共通するパラメータにはfor文を通って一度に設定します。そして最後にucIdもiに設定して、stIJog配列のインデックスとモーターIDが同じになるようにします。

```
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;
```

0番モーターから6番モーターまでそれぞれ制御モードと速度/位置、値、実行時間を設定します。車輪に対応する0、2、4、5番モーターは速度制御モードで速度の値は0の時に停止させ、ステアリング軸に対応する1、3番サーボは位置制御モードでそれぞれに789、235を送って並んで直進する方向を向くようにします。荷台の6番サーボは位置制御モードで970を送って基本の位置にあるようにします。

```
hklx_SendPacket(stSendPacket);
```

hklx_SendPacket()関数を使用して上記のように作成し、パケットをHerkuleXに送信します。

```
void load_on(void){
…
}
```

次の関数は、load_on()関数です。ダンプトラックの荷台サーボ(6番)を基本の位置に送ります。

```
DrsPacket stSendPacket;


stSendPacket.ucPacketSize = MIN_PACKET_SIZE + CMD_I_JOG_STRUCT_SIZE;
stSendPacket.ucChipID = BROADCAST_ID;
stSendPacket.ucCmd = CMD_I_JOG;
```

送信するパケットを保存するstSendPacket構造体を宣言して、パケットのサイズ、ID、CMD設定します。6番サーボのみを制御するので、このときのパケットのサイズは、基本的に7バイト(MIN_PACKET_SIZE)に5バイトが加わって12になります。

```
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);
```

stJogのuiValueは基本の位置である970、stSetのucModeは0(位置制御)、ucIdは6、実行時間は100(約1.12秒)に設定した後、パケットを送信します。荷台サーボが基本の位置に移動します。

```
void load_off(void){
…
}
```

次の関数は、load_off関数です。ダンプトラックの荷台サーボ(6番)を荷物を下ろす場所に送信します。この関数の内容はload_on関数とほぼ同じです。

```
stSendPacket.unData.stIJogData.stIJog[0].stJog.uiValue = 700;
```

load_off関数がload_on関数と違う部分です。 6番サーボに送信する場所だけ違っており、他はすべて同じです。700を荷台サーボに送信するとダンプトラックが荷物を下ろすように荷台をななめにします。

```
void left_turn(void){
…
}
```

ステアリングサーボ(1、3番)を動かして左折する方向で前輪を変える関数です。

```
DrsPacket stSendPacket;


stSendPacket.ucPacketSize = MIN_PACKET_SIZE + 2 * CMD_I_JOG_STRUCT_SIZE;
stSendPacket.ucChipID = BROADCAST_ID;
stSendPacket.ucCmd = CMD_I_JOG;
```

送信するパケットを保存するstSendPacket構造体を宣言して、パケットサイズ、ID、CMD設定します。1、3番サーボを制御するので、このときのパケットサイズは、基本的に7バイト(MIN_PACKET_SIZE)に5*2バイトが加わって17になります。

```
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);
```

1番モーターと3番モーターを位置制御して左折方向に前輪を変更するようにパラメータを設定します。stJogのuiValueは基本の位置である789、235より100小さく、それぞれ689、135、stSetのucModeは0(位置制御)、ucIdはそれぞれ1、3、実行時間は50(約0.56秒)に設定した後、パケットを送信します。

```
void right_turn(void){
…
}
```

ステアリングサーボ(1、3番)を動かして右折する方向に前輪を変える関数です。この関数の内容はleft_turn()関数とほぼ同じです。

```
SendPacket.unData.stIJogData.stIJog[0].stJog.uiValue = 889;
                                        …
stSendPacket.unData.stIJogData.stIJog[1].stJog.uiValue = 335;
```

load_off関数がload_on関数と違う部分です。1、3番サーボに送信する場所だけ違っており、他はすべて同じです。今回は基本の位置である789、235より100大きく、それぞれ889、335に設定します。

```
void forward(void){
…
}
```

この関数は、トラックを前進させる関数です。車輪サーボ(0、2、4、5番)は前に転がる方向に速度制御して、ステアリングサーボ(1、3番)は正面を眺めるようにI_JOGパケットを送信します。

```
DrsPacket stSendPacket;
unsigned char i;
```

変数を宣言します。stSendPacketは送信するI_JOGパケットが保存される領域です。 iはfor文を制御するモーターの数だけ繰り返すために宣言された変数です。

```
stSendPacket.ucPacketSize = MIN_PACKET_SIZE + 6 * CMD_I_JOG_STRUCT_SIZE;
stSendPacket.ucChipID = BROADCAST_ID;
stSendPacket.ucCmd = CMD_I_JOG;
```

パケットのサイズとID、CMDを設定する部分です。この関数では、ID0～5の6個のモーターを制御するため、パケットのサイズはI_JOGのパケット構造に応じて、基本的に7バイト(MIN_PACKET_SIZE)に5*6バイトが加わって37になります。

```
for(i=0;i<6;i++){
…
}
```

値が共通するパラメータをfor文を介して一度に設定します。そして最後にucIdもiに設定して、stIJog配列のインデックスとモーターIDが同じになるようにします。

```
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;
```

0番モーターから5番モーターまでそれぞれ制御モードと速度/位置、値、実行時間を設定します。車輪に対応する0、2、4、5番モーターは速度制御モードで速度の値は左の0、4番は正方向800、右側の2、5番は逆方向800に設定して前進方向にすべての車輪が変更されます。2、5番にstJog.uiValueを設定する時に800とINVERSE_TURN_MASKをBitwise　OR演算した値を入れ、逆方向に回転させます。ステアリング軸に対応する1、3番サーボは位置制御モードでそれぞれ789、235を送って並んで直進する方向を向くようにします。

```
hklx_SendPacket(stSendPacket);
```

hklx_SendPacket()関数を使用して上で構成されたパケットをHerkuleXに送信します。

```
void backward(void){
…
}
```

この関数は、トラックを後退させる関数です。車輪サーボ(0、2、4、5番)は後ろに転がっていく方向に速度制御してステアリングサーボ(1、3番)は正面を眺めるようにI_JOGパケットを送信します。この関数はforward()関数と内容がほぼ同じです。

```
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()関数がforward()関数と違う部分です。正方向/逆方向を直進と逆で後退方向にすべての車輪モーターが動くように設定します。

```
void guide turn(void){
…
}
```

quick_turn()関数は、障害物があった時に使用します。障害物を避けるために左車輪は前進、右車輪は後退して所定の位置から右に回るパケットを送信します。

```
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;
```

送信するパケットを保存するstSendPacket構造体を宣言し、for文で使用する変数iを宣言して、パケットのサイズ、ID、CMDを設定します。0、2、4、5番サーボを制御するので、この時のパケットのサイズは基本7バイト(MIN_PACKET_SIZE)に5*4バイトが加わって27になります。

```
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;
}
```

値が共通するパラメータをfor文を介して一度に設定します。この関数では、すべてのサーボの制御モード、速度の値は、実行時間が同じなので、for文で一度に設定します。ただし今回は、サーボのIDがばらばらなのでucIdをiに設定する部分は抜かしています。

```
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);
```

ucIdに対象サーボのIDを入れてパケットを送信します。このパケットを送信するとダンプトラックは、所定の位置から右に回転します。

ここからはmain関数の説明です。

```
DrsPacket stSendPacket;
unsigned char ucData, ucDataOld, ucLength, ucForward;
int iADC;
```

メイン関数で使用する変数を宣言します。DrsPacket構造体タイプでパケットを送信する時に使う変数を宣言します。ucDataは現在、whileループで読んだリモコンの値であり、ucDataOldは直前のwhileループで読んだリモコンの値です。毎回whileループの最後でucDataOldはucDataの値に更新されます。この二つの変数は、リモコンを押したり離したりしてリモコン値が変わった瞬間にタイマー/カウンタ2の値を変更するために使用します。ucLengthはリモコンが押された長さを読み取る変数です。ucForwardは現在、ロボットが前進していることを示す変数です。1の場合、前進しているという意味です。iADCは読んだADC値が保存される16ビットの定数変数です。

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

ポートCの0～6番ピンを出力に設定し、出力値を1にします。これにより、すべてのLEDが消灯します。

```
ADMUX=0b00000110;
ADCSRA=0b10010111;
```

ADCを単極性モードでADC6ポートからアナログ入力を読み込むためのレジスタを設定します。ADMUXから見ると、DRCの回路上での基準電圧は、ATmega128の動作電圧である5Vと同じようにAREFピンに接続されているのでREFS1:0ビットは0b00に設定し、左に整列を使用していないため、ADLARも0に設定します。そしてADC6ポートを単極性モードで読み取るため、MUX4:0ビットは0b00110に設定します。

ADCSRAでは、まずADCを有効にするために、ADENビットを1に設定します。まだ変換を開始するものではなくフリーランニングも使用しないのでADSC、ADFRは0に設定します。ADIFの場合は意図せずに1に設定されていることがありますので0に初期化するためには1を使用して、割り込みを使用しないためにADIEは0に設定します。最後に、ATmega128の16MHzクロック周波数をADCに適した50～200kHzに分けて与えるためにADPS2:0ビットを0b111に設定して128で割った125kHzをADCクロックに供給するように設定します。

ADCレジスタの詳細については、Chapter6を参照してください。

```
cli();
hklx_Init(115200);
RMC_Init();
sei();
```

割り込みの設定を変える時に割り込みが発生して予期せぬ誤動作が発生することを防止するために、まずcli()命令を通して全体の割り込みを無効にします。その後hklx_Init()関数を使用してHerkuleXを制御するためのタイマー/カウンタ0とUSART0を初期化します。使用ボーレートが115200であるため、パラメータとしてその値を使用します。そしてRMC_Init()関数を使ってリモコン信号を受信するための外部割り込み4番ピンとタイマー/カウンタ1を初期化します。初期化が終わった後は、sei()命令を通して全体の割り込みを有効にします。

```
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);
```

モーターが自ら動くように制御するためにはモーターにトルクを認可する必要があります。上記のコードがトルクを認可させるためのRAM_WRITEパケットを構成して送信する部分です。パケットのSIZEは基本MIN_PACKET_SIZE(7バイト)に加え、アドレス+長さ+データ(1バイト)で総MIN_PACKET_SIZE+3(10バイト)であり、接続されたすべてのモーターに送信するためのIDはBROADCAST_ID、つまり254です。CMDはCMD_RAM_WRITE(0x03)であり、unData共用体のメンバーの中でstRWData構造体のucAddress、ucLen、ucData[0]の値をそれぞれ52、1、0x60にします。その後はhklx_SendPacket()関数を呼び出してパケットを送信します。

```
ucDataOld = RMC_ERROR_DATA;
ucForward = 0;
```

先ほど説明したように、ucDataOld変数には、while文で直前に読み取ったリモコンの値が保存されます。しかし、まだwhile文に進入する前であり、リモコン値を読む前なので、基本値RMC_ERROR_DATAをucDataOldに保存します。また、今はダンプトラックが止まっているので、前進しているかどうかを示すucForward変数を0に初期化します。

```
ADCSRA |= (1<<ADSC);
while(!(ADCSRA & (1<<ADIF)));
iADC = ADCL | (ADCH<<8);
ADCSRA |= (1<<ADIF);
```

ここからはwhile文の説明です。 while文の最初の部分では、ADC変換を繰り返します。まずADCSRAのADSCビットを1にしています。これにより、ADC変換が開始します。while(!(ADCSRA&(1<< ADIF)));命令に従いADCSRAのADIFが0である間は待機します。ADC変換が終わってADIFビットが1になる瞬間にwhile文を出て次に進行します。

ADC変換が完了してADCLとADCHが更新されると、ADCLとADCHの値を読み取ってiADCに保存します。ADCHは、ADC値の上位1バイトに対応するため8ビット(1バイト)ほど左に<<(shift left)演算をしてADCLと|(bitwise OR)演算をすることで10ビットのデータをiADCに保存することができます。

私たちは、ADC変換完了割り込みを使用しないので、ADIFビットが自動的に0に設定されていません。そのためADC値を読んだ後には、割り込みフラグを再度0に初期化する必要があります。ADCSRAのADIFビットに1を使用してこのビットを0に初期化します。.

```
if(iADC > PSD_THRESHOLD && ucForward == 1){
        stop();
        delay_ms(500);
        quick_turn();
        delay_ms(1000);
        forward();
        ucForward = 1;
}
```

iADC値がPSD_THRESHOLDより大きく20センチ以内に障害物があると判断され、現在前進している場合、if文の中に進入します。まず、stop()関数を呼び出して停止した後0.5秒の間、待機します。そしてquick_turn()関数を呼び出して所定の位置で右折するパケットを送信し、1秒間右折を待ちます。その後はforward()関数を呼び出して、前進するようにします。そしてucForwardを1にして前進していることを示します。

```
RMC_ReadRemocon(RMC_CHANNEL, &ucData, &ucLength);
```

RMC_ReadRemocon()関数を使用して、リモコンの値を読み込みます。最初のパラメータには、希望のリモコンのチャンネルであるRMC_CHANNELを入れて、2番目、3番目のパラメータには、ucDataとucLengthのアドレスを入れます。そして、RMC_CHANNELチャンネルで入ってきたデータと押された長さがucDataとucLengthに保存されます。

```
if(ucDataOld != ucData){
…
}
```

直前のwhileループで読み取ったリモコン値（ucDataOld）とたった今読んリモコン値（ucData）を比較して、異なる場合にif文の中の内容を実行します。リモコンキーが押された瞬間にのみ命令を実行する部分です。

```
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;
}
```

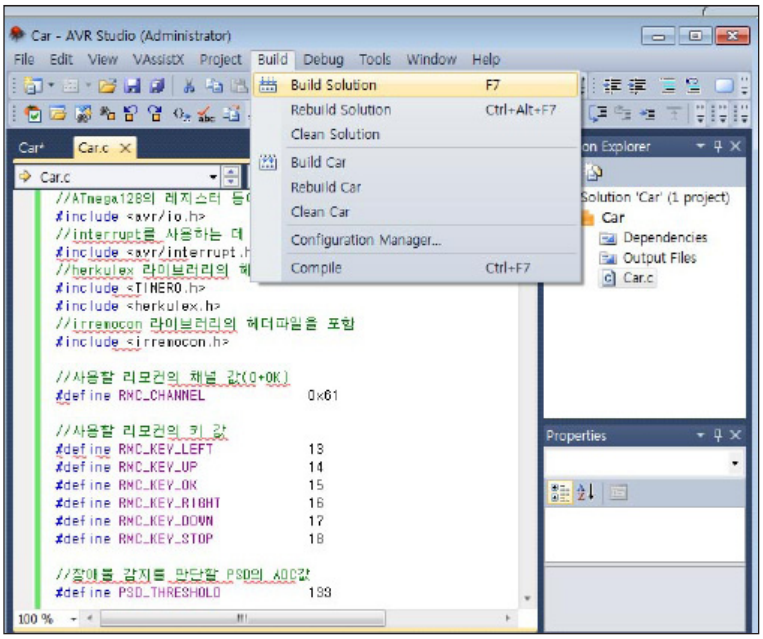
ucDataの値に応じて対応する動作を実行する部分です。ucDataがRMC_KEY_UPと同じなら、上の方向キーが押されているためforward関数を呼び出してトラックが前進しucForwardを1に変更します。続くelse　if文もリモコンキーとそれに対応する動作を示します。下の方向にキーが押されたらbackward関数を呼び出して、左の方向にキーが押されてたらleft_turn、右の方向にキーが押されたらright_turnを呼び出します。OKキーが押されたらload_off関数を呼び出して荷物を下ろす動作をして、1秒後に再びload_on関数を呼び出して荷台を元の位置にします。停止キーが押されたらstop関数を呼び出します。後退する時と停止する時にはucForwardを0にして、後退している時、または停止状態では、PSDが認識した障害物に対しては反応しないようにします。

```
ucDataOld = ucData;
```

whileループが終了するたびに、現在のループで読んだucDataの値をucDataOldに代入して、次のループで比較するところに使用します。

実行

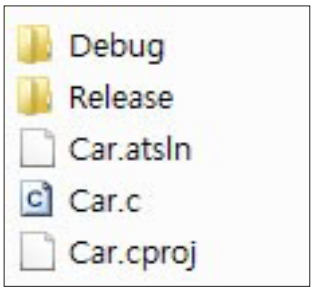
ダンプトラックロボットに接続し、ファームウェアをダウンロードして実行すると、リモコンの入力に応じてダンプトラックロボットを制御することができます。



01 Build

Buildメニューを選択しBuild Solutionをクリックしてビルドします。

1 succeeded0 failedが出るとエラーなしでコンパイルが完了したということです。



02 Debug フォルダ

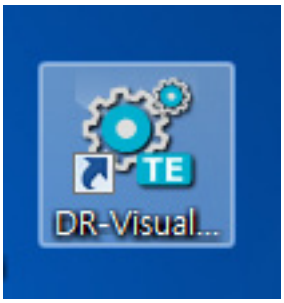
Build後にソースがあるフォルダに入るとDebugフォルダが作成されています。クリックしてフォルダの中身を見ます。



03 hex 確認

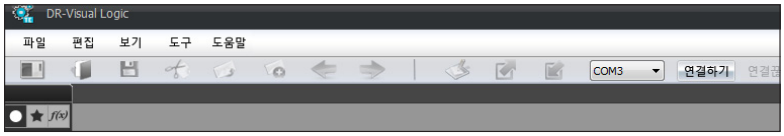
Debugフォルダの中のhexファイルを確認します。Hexファイルは制御機に移動するファイルです。

DR-Visual Logicでファームウェアの修復を行います。



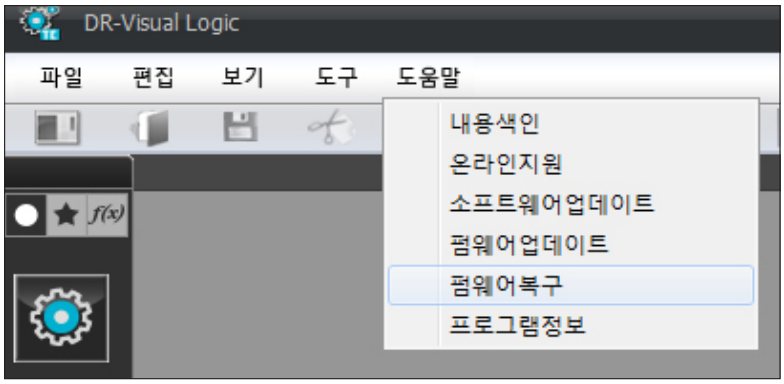
04 Visual Logic

DR-Visual Logicを実行します。



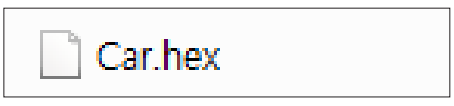
05 接続する

COM番号を設定して接続するをクリックします。



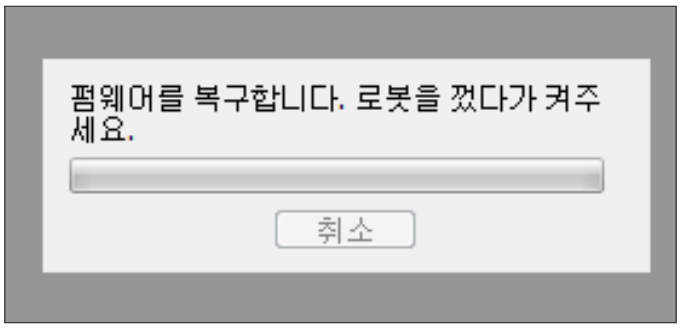
06 ファームウェアの修復

[ヘルプ]メニューからファームウェア修復をクリックします。



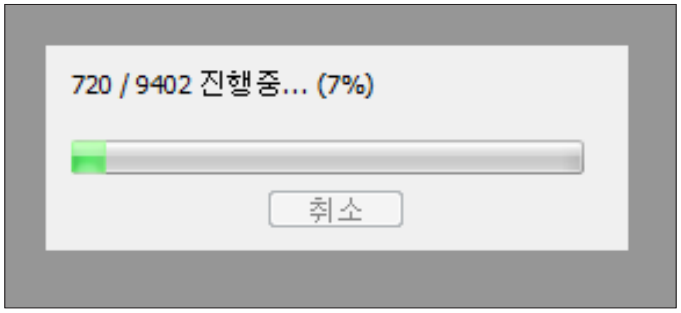
07 ロード

DebugフォルダにHexファイルをロードします。



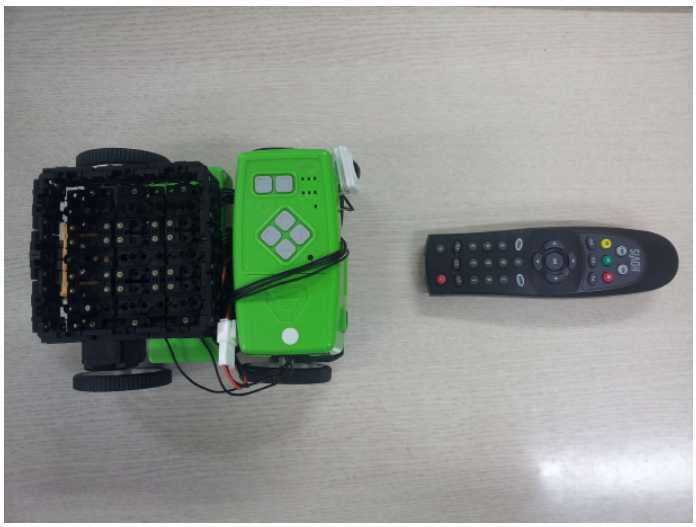
08 ファームウェアの修復

“ロボットの電源を切って入れ直してください”というメッセージが出ます。制御機の電源ボタンを一度切ってから再度入れ直してください。



09 修復の開始

ファームウェアの修復が開始されます。



10 実行

DRCをダンプトラックロボットに接続し、ファームウェアをダウンロードして実行すると、リモコンの入力に応じてダンプトラックロボットの制御ができます。

ロボットで学ぶ AVR C Programming



ATmega128マイクロプロセッサを適用した ロボット制御機用AVR Cプログラミングの本物

PART 01. HOVIS DRCとATmega128の習得

PART 02. Visual Logicと関するC言語文法の習得

PART 03. AVRロボット制御機のファームウェアの習得

PART 04. ロボットに応用する

Dongbu Robot

株)Dongbu Robot www.dongburobot.com

本 社：11th Floor, Bucheon Techno Park 401, Yakdae-dong, Wonmi-gu, Bucheon-city, Gyunggi-do 420-734, Korea
TEL：+82-32-329-5551(ext.311), FAX：+82-32-329-5569, E-MAIL：robotsales@dongbu.com
工 場：27, 6 Gil, 4 Sandan, Jiksan-eup, Seobuk-gu, Cheonan-city, Chungcheongnam-do 331-814, Korea
TEL：+82-41-590-1700.FAX：+82-41-590-1701

93550
9 788998 873141
ISBN 978-89-98873-14-1