プレイステーション2専用
ローリングスイッチ
技術資料

カプコン クラシックス コレクションの、フォゴットンワールド（ロストワールド）や、アルティメットエコロジーを、アーケードに近い感覚で遊ぶことができる、プレイステーション2専用ローリングスイッチの製作に関する技術資料です。
資料の前半は、主に通信仕様や信号処理に関する内容です。資料の後半は、主に部品の加工や基板の製作に関する内容です。すぐに製作に取り掛かりたい方は、「ローリングスイッチユニットの製作」以降をご覧ください。

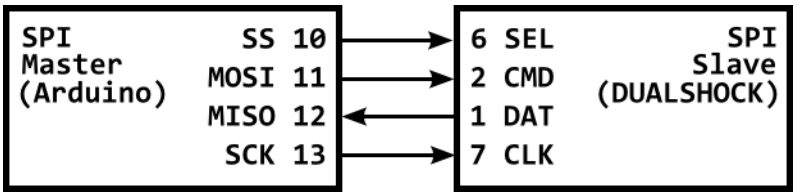
目次

2015年11月01日 初版

Kazumasa ISE
Twitter: @kaz_ise
http://magicpuppet.org
Mail: kzms.ise@gmail.com

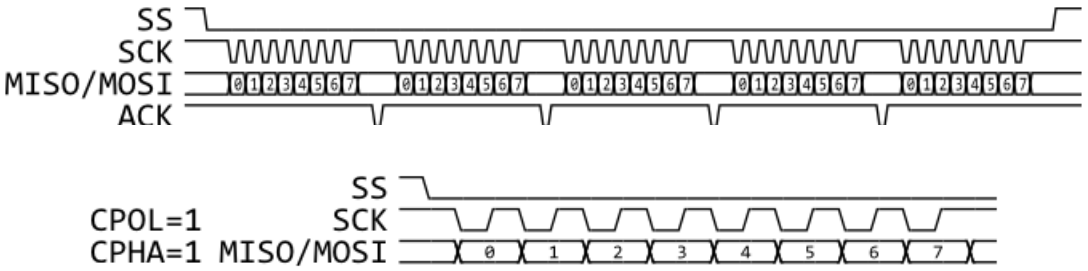
DUALSHOCK の通信仕様確認

DUALSHOCK の通信仕様は、プレイステーション・ＰＡＤ／メモリ・インターフェースの解析[1]に、非常に詳しく記載されています。
この文献によると、通信仕様は、殆ど Serial Peripheral Interface (SPI)[2]であることがわかります。違いは、ACK (Acknowledge)の有無だけです。
通信仕様の確認のために、DUALSHOCK の信号を Arduino で受信してみます。
DUALSHOCK を SPI スレーブ、Arduino を SPI マスターとします。



通信仕様

DUALSHOCK の信号は、以下のようになっています。

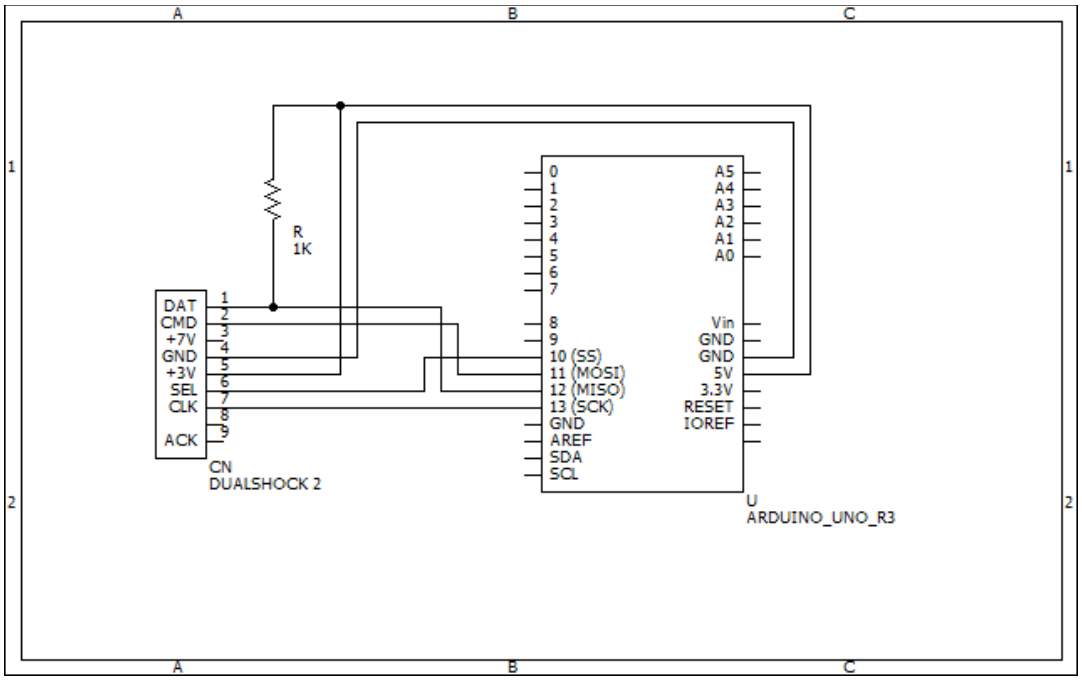


- ビットオーダーは LSBFIRST です。
- クロックはアイドル時 HIGH なのでクロック極性(CPOL)は 1 です。
- クロックの立ち上がりでデータを読み取るためクロック位相(CPHA)は 1 です。
- 動作速度は 250kHz です。

CPOL =1, CPHA =1 ならば、Arduino の動作モードは SPI_MODE3 となります。
Arduino の動作クロックは 16MHz なので、250kHz は 1/64（分周比 64）です。
データの送受信のインターバルは、アバウトでも良いようです。各バイトデータの送受信の間隔は、とりあえず 16μsec としました。一連のデータの通信は、16msec 周期で行われますが、これより長くても良いようです。
最も基本的なデジタルモードでの通信の場合、一回の通信では、以下のような 5bytes のデータを送受信します。

```
CMD: 0x01, 0x42, 0x00, 0x00, 0x00
DAT: 0xXX, 0x41, 0x5A, 0xXX, 0xXX
```

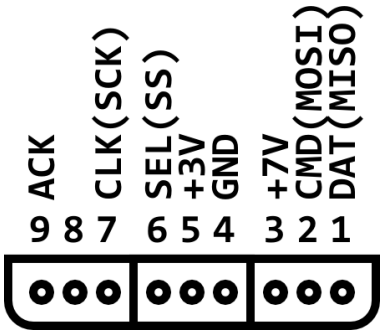
回路図

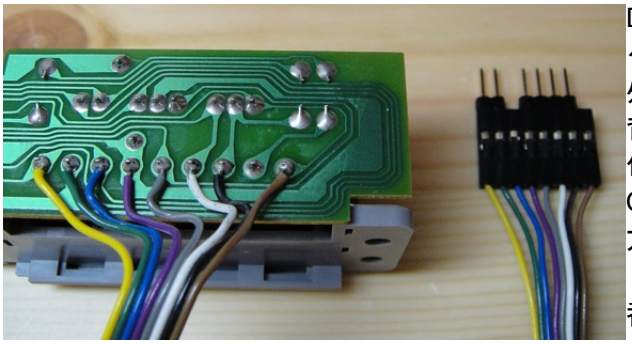


DUALSHOCK の電源電圧は、3V から 5V であれば良い[3]そうなので、ここでは Arduino の 5V を与えています。
本来、DUALSHOCK の電源電圧は 3.6V なので、Arduino の 3.3V を与えるべきなのかもしれませんが、そうなるとロジックレベル変換が厄介なので、ここでは全て 5V とします。
SPI 通信で使用する 4 本の信号線を接続します。ACK は、ここでは無視します。
DAT 信号は、Arduino の内臓プルアップ抵抗では受信できなかったので、文献[1]に従って、プルアップ抵抗 1kΩ を接続しました。

ピンアサイン

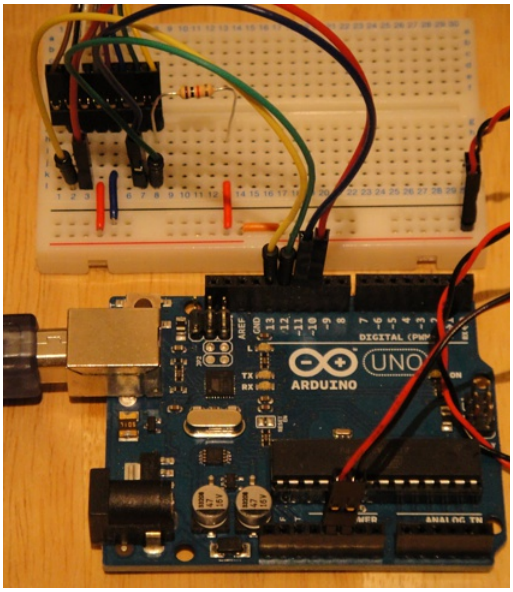
DUALSHOCK のピンアサインは以下のようになっています。

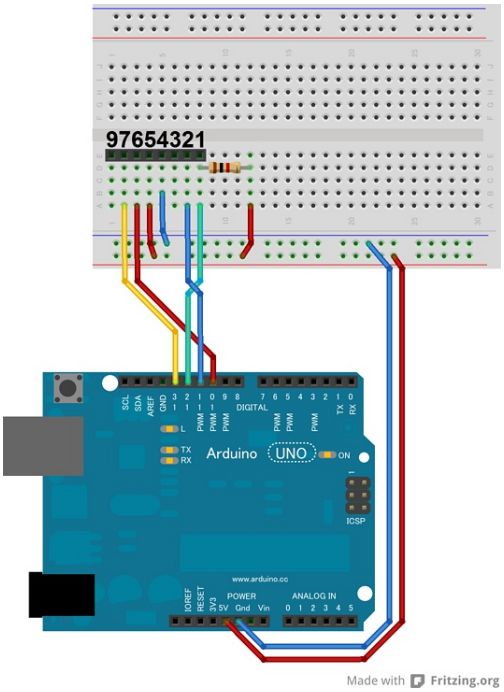




DUALSHOCK ソケットは、ジャンクのプレイステーションから取り外しました。実験しやすいよう、背面の端子部分にケーブルを半田付けし、反対側のケーブルの端をQI コネクタとします。さらに、両方長いピンヘッダを付け、ブレッドボードへ接続しやすくします。8番ピンは未接続です。

部品配置図





ソースコード

```
// SPI test Arduino: Master, DUALSHOCK: Slave
//
// Copyright (c) 2015 Kazumasa ISE
// Released under the MIT license
// http://opensource.org/licenses/mit-license.php

#include <SPI.h>

#define DEBUG
#define TRANSFER_WAIT 16
#define FRAME_WAIT 16
```

```
const byte CMD[] = {0x01, 0x42, 0x00, 0x00, 0x00};
const byte CMD_BYTES = sizeof CMD;
byte DAT[CMD_BYTES] = {0};

void setup() {
 SPI.setBitOrder(LSBFIRST);
 SPI.setClockDivider(SPI_CLOCK_DIV64);
 SPI.setDataMode(SPI_MODE3);
 SPI.begin();
 pinMode(MISO, INPUT);
#ifdef DEBUG
 Serial.begin(115200);
#endif
}

void loop() {
 digitalWrite(SS, LOW);
 delayMicroseconds(TRANSFER_WAIT);
 for (byte i=0; i<CMD_BYTES; i++) {
  DAT[i] = SPI.transfer(CMD[i]);
  delayMicroseconds(TRANSFER_WAIT);
 }
 digitalWrite(SS, HIGH);
 delay(FRAME_WAIT);
#ifdef DEBUG
 Serial.print("DAT: ");
 for (byte i=0; i<CMD_BYTES; i++) {
  Serial.print(DAT[i], HEX);
  Serial.print(" ");
 }
 Serial.println();
 delay(500);
#endif
}
```

Arduino の SPI ライブラリ[4]を使用します。

setup()

初期化を行います。
ビットオーダー、動作速度、動作モードを設定します。
デバッグのためにログ出力の準備をします。

loop()

送受信を行います。
SS を LOW にして、通信を開始します。
16μsec 待った後、CMD を送信し、同時に DAT を受信します。

各バイト毎に 16μsec 待ちます。
SS を HIGH にして、一回の通信を終了します。
16msec 待ち、次の通信に備えます。
デバッグのためにログを出力します。

結果

以下のような応答を確認しました。

```
DAT: FF 41 5A FF FF
DAT: FF 41 5A FF FF
DAT: FF 41 5A FF DF
```

DAT の 4byte 目と 5byte 目の各ビットは、レバーやボタンの入力に対応しています。
ON:0, OFF:1 です。
3 行目の応答は、〇ボタンを押したときのものです。
DF は、2 進数では 11011111 です。上位から 3 番目のビットは、〇ボタンに対応しているので、確かに〇ボタンが押された結果を受信しているのがわかります。
このように、通信仕様が確認できました。

部品

ハードオフ

DUALSHOCK (SCPH-1200)または DUALSHOCK 2 (SCPH-10010) ￥300
DUALSHOCK ソケット（ジャンク） ￥500

スイッチサイエンス

Arduino をはじめようキット￥4320

千石電商

【QI コネクタ】信号伝達コネクタ用ピン メス【F】(10 本) ￥74

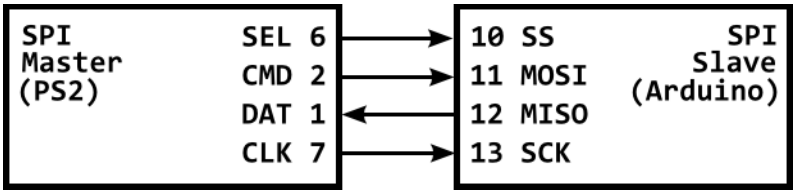
【QI コネクタ】信号伝達コネクタ（黒）1×8 2550-1×8 ￥21
ピンヘッダ 1 列×40P (ストレート・標準ピッチ) 2544-1×40 15.1(6.3/6.3) ￥105
協和ハーモネット RKV 10/0.12×10 列 L-1 10 芯リボンケーブル 1m ￥280

参考文献

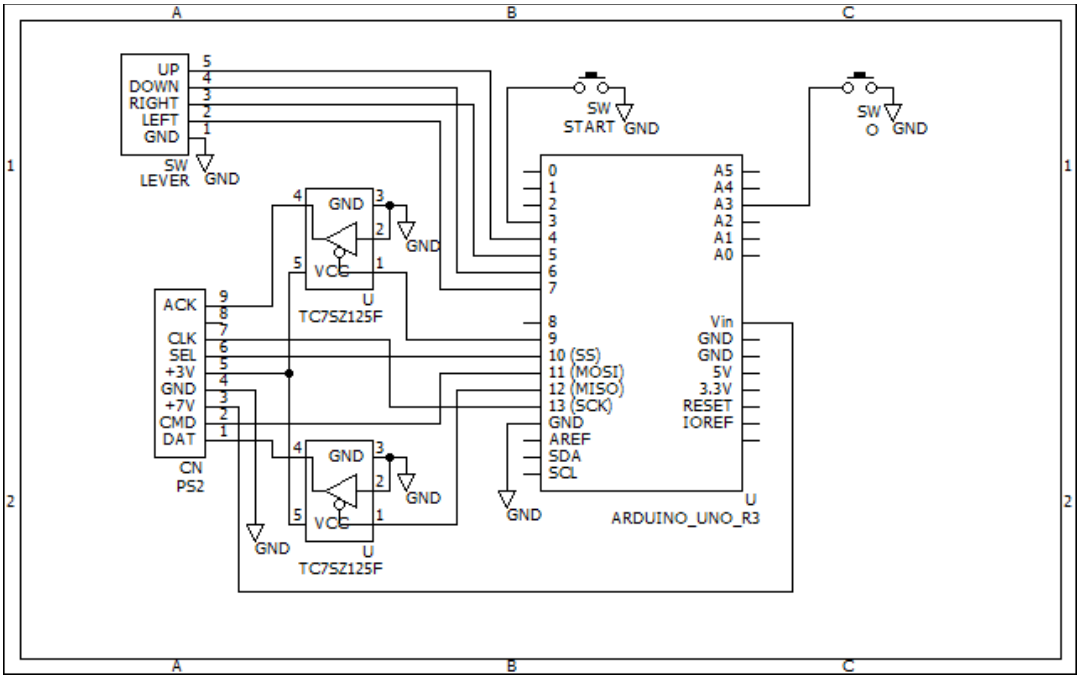
[1] プレイステーション・ＰＡＤ／メモリ・インターフェースの解析
http://kaele.com/~kashima/games/ps_jpn.txt
[2] Serial Peripheral Interface Bus - Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus
[3] PSX Controllers
http://www.gamesx.com/controldata/psxcont/psxcont.htm
[4] Arduino 日本語リファレンス - SPI の概要
http://www.musashinodenpa.com/arduino/ref/index.php?f=1&pos=531

Arduino による DUALSHOCK のエミュレート(1)

Arduino で DUALSHOCK をエミュレートし、PlayStation 2 のコントローラとして動作させてみます。ここでは、最も基本的なデジタルモードでの動作を試みます。
PlayStation 2 を SPI マスター、Arduino を SPI スレーブとします。
Arduino の SPI ライブラリは、SPI スレーブには対応していないため、非対応の部分については、ハードウェアの機能を直接利用することになります。



回路図



PlayStation 2 に対する電気的インタフェースは、文献[1]に従って HC125 相当を利用します。また、PlayStation 2 の信号レベルは 3.6V なので、Arduino の 5V からロジックレベル変換する必要があります。ここでは、トレラント機能のある TC7SZ125F を利用し、ロジックレベル変換を行います。
PlayStation 2 からの出力は、Arduino に直接与えます。本来ここでもロジックレベル変換を行うべきだと思いますが、特に問題なく動作するようなので、直接接続します。
Arduino の電源には、PlayStation 2 から供給される 7V を与えます。グランドは全て共通です。

デジタルモードでの通信の場合、一回の通信では、5bytes のデータを送受信します。DAT の 4byte 目と 5byte 目の各ビットは、以下のようにレバーやボタンの入力に対応しています。ON:0, OFF:1 です。

	CMD	DAT	b7	b6	b5	b4	b3	b2	b1	b0
1byte	0x01	--								
2byte	0x42	0x41								
3byte	0x00	0x5A								
4byte	0x00	0xXX	左	下	右	上	ST	1	1	SE
5byte	0x00	0xXX	□	×	○	△	R1	L1	R2	L2

レバーやボタンのスイッチのピンは、ポート操作[2]を行い、ON/OFF の状態を一括で取得します。SPI 通信で応答するバイト列にビットの並びをあわせることで、処理の手間が省けます。
ポートＣ（アナログピン 0~5）を○×□△ボタンに割り当てます。
ポートＣの値は、2 ビット左シフトし、下位 4 ビットをマスクし、5byte 目とします。

PINC:	A5	A4	A3	A2	A1	A0
	□	×	○	△	--	--

ポートＤ（デジタルピン 0~7）をレバーと START ボタンに割り当てます。
ポートＤの値は、下位 3 ビットをマスクし、4byte 目とします。

PIND:	D7	D6	D5	D4	D3	D2	D1	D0
	左	下	右	上	ST	--	--	--

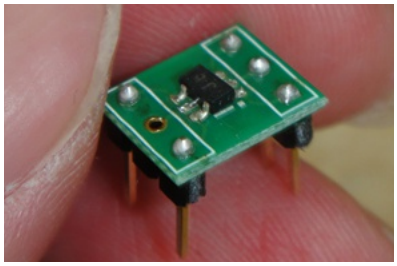
部品

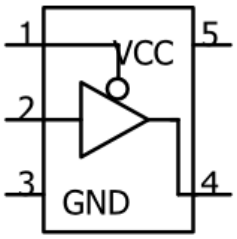
コントローラのケーブルは、ジャンクの DUALSHOCK から取り外しました。利用しやすいよう、ケーブルの端を QI コネクタで端末処理を行います。ブレッドボードへ接続する際は、両方長いピンヘッダを利用します。8 番ピンは未接続です。
ピン番号に対応するケーブルの色は、以下のようになっていました。

1	2	3	4	5	6	7	8	9
茶	橙	紫	黒	赤	黄	青	灰	緑

コントローラの種類によって、微妙に異なることがあるので、注意して下さい。

TC7SZ125F は表面実装部品なので、利用しやすいよう、変換基板に半田付けします。
かなり小さい部品なので、半田付けに慣れていないと苦労するかもしれません。
代替部品については、後ほど検討したいと思います。





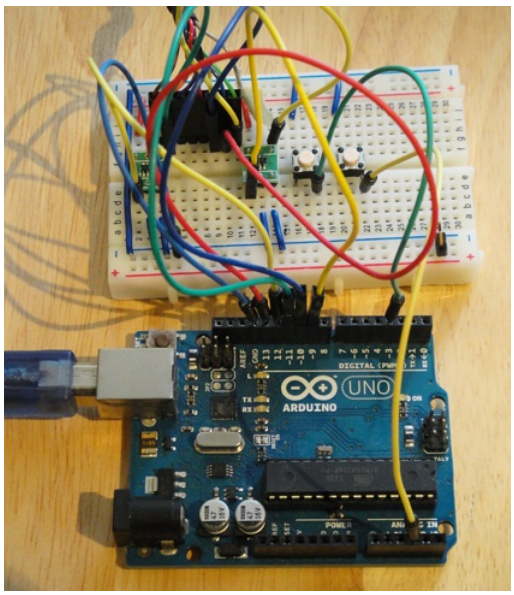
1 G
2 IN A
3 GND
4 OUT Y
5 VCC

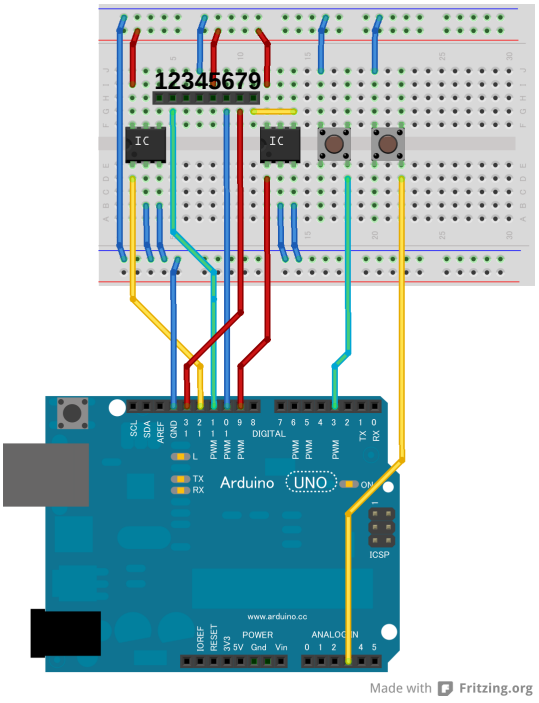
注意事項

実験を行う際は、接続に間違いがないか十分に確認して下さい。
接続を間違えると、PlayStation 2 が壊れてしまう可能性があります。
最初は、DUALSHOCK を USB に変換するコンバータを利用し、PC 上で確認することをおすすめします。

部品配置図

デバッグの都合上、Arduino は PC に接続したままとなります。Arduino には USB から電源が供給されるため、Vin は接続していません。
スイッチは、START ボタンと〇ボタンのみとし、レバーの入力は省略しています。





ソースコード

```
// SPI test PS2: Master, Arduino: Slave (DIGITAL MODE)
//
// Copyright (c) 2015 Kazumasa ISE
// Released under the MIT license
// http://opensource.org/licenses/mit-license.php

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

#define DEBUG
#define ACK_WAIT 0.5
#define ACK 9
#define SET_ACK_LOW  (PORTB &= ~B00000010)
#define SET_ACK_HIGH (PORTB |=  B00000010)
#define SW_LEFT      7
#define SW_DOWN      6
#define SW_RIGHT     5
#define SW_UP        4
#define SW_START     3
#define SW_SQUARE   19
#define SW_CROSS    18
#define SW_CIRCLE   17
#define SW_TRIANGLE 16
#define SW1 (PIND | B00000111)
#define SW2 ((PINC << 2) | B00001111)
#define CMD_BYTES 5

void setup() {
 // SPI setup
 SPI.setBitOrder(LSBFIRST);
 SPI.setDataMode(SPI_MODE3);
 SPCR &= ~(_BV(MSTR)); // Set as Slave
 SPCR |= _BV(SPE); // Enable SPI
 pinMode(SS, INPUT);
 pinMode(MOSI, INPUT);
 pinMode(MISO, OUTPUT);
 digitalWrite(MISO, HIGH);
 pinMode(SCK, INPUT);
 SPI.attachInterrupt();
 // ACK pin setup
 pinMode(ACK, OUTPUT);
 digitalWrite(ACK, HIGH);
 // Switches setup
 pinMode(SW_LEFT, INPUT_PULLUP);
 pinMode(SW_DOWN, INPUT_PULLUP);
 pinMode(SW_RIGHT, INPUT_PULLUP);
 pinMode(SW_UP, INPUT_PULLUP);
```

```
 pinMode(SW_START, INPUT_PULLUP);
 pinMode(SW_SQUARE, INPUT_PULLUP);
 pinMode(SW_CROSS, INPUT_PULLUP);
 pinMode(SW_CIRCLE, INPUT_PULLUP);
 pinMode(SW_TRIANGLE, INPUT_PULLUP);
#ifdef DEBUG
 Serial.begin(115200);
#endif
}

inline byte dat(byte i) {
 const byte DAT[] = {0xFF, 0x41, 0x5A, 0xFF, 0xFF};
 switch (i) {
 case 3: return SW1;
 case 4: return SW2;
 default: return DAT[i];
 }
}

inline void acknowledge() {
 SET_ACK_LOW;
 _delay_us(ACK_WAIT);
 SET_ACK_HIGH;
}

#ifdef DEBUG
#define LINE_FEED 0xAA
#define MAX_LOG_SIZE 300
volatile byte cmdLog[MAX_LOG_SIZE] = {0};
volatile byte datLog[MAX_LOG_SIZE] = {0};
volatile int logCount = 0;
#endif

ISR(SPI_STC_vect) {
 static byte CMD[CMD_BYTES] = {0};
 static byte cmdCount = 0;
 bool continueCom = false;
 CMD[cmdCount] = SPDR;
#ifdef DEBUG
 if (logCount < MAX_LOG_SIZE) {cmdLog[logCount++] = CMD[cmdCount];}
#endif
 // Check CMD
 if (cmdCount == 0) {
  if (CMD[cmdCount] == 0x01) {
   continueCom = true;
  }
 } else if (cmdCount == 1) {
  if (CMD[cmdCount] == 0x01) {
   cmdCount = 0; // Reset count
```

```
   continueCom = true;
  } else if (CMD[cmdCount] == 0x42) {
   continueCom = true;
  }
 } else if (cmdCount < CMD_BYTES-1) {
  continueCom = true;
 }
#ifdef DEBUG
 if (!continueCom && (logCount < MAX_LOG_SIZE)) {cmdLog[logCount++] =
LINE_FEED;}
#endif
 // Set next DAT
 cmdCount = continueCom ? cmdCount+1 : 0;
 SPDR = dat(cmdCount);
#ifdef DEBUG
 if (logCount < MAX_LOG_SIZE) {datLog[logCount] = dat(cmdCount);}
#endif
 if (continueCom) {acknowledge();}
}

void loop() {
#ifdef DEBUG
 if (logCount == MAX_LOG_SIZE) {
  logCount = 0;
  int lfPos = -1;
  for (int i=0; i<MAX_LOG_SIZE; i++) {
   if (cmdLog[i] == LINE_FEED) {
    Serial.print("DAT: ");
    for (int j=lfPos+1; j<i; j++) {
     Serial.print(datLog[j], HEX);
     Serial.print(" ");
    }
    lfPos=i;
    Serial.println();
    Serial.print("CMD: ");
   } else {
    Serial.print(cmdLog[i], HEX);
    Serial.print(" ");
   }
  }
 }
#endif
}
```

文献[3]を参考に、最も基本的なデジタルモードでの動作を試みます。

setup()
初期化を行います。
ビットオーダー、動作モードの設定などは、Arduino の SPI ライブラリが利用できます。スレーブモードの動作に関する設定は、SPI ライブラリが利用できないので、直接設定します。
SPI で使用するピンに加え、ACK ピンを設定します。ACK ピンは 9 番ピンとします。
レバーやボタンのスイッチのピンを設定します。スイッチはプルアップします。
デバッグのためにログ出力の準備をします。

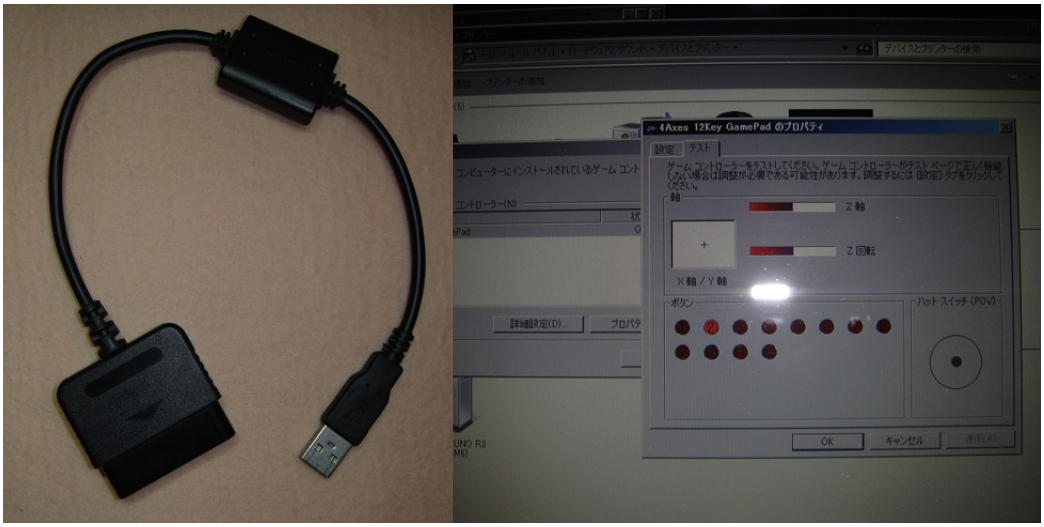
dat()
SPI 通信で応答するバイト列を生成します。
デジタルモードでの通信の場合、一回の通信では、5bytes のデータを送受信します。
応答するバイト列の 4byte 目と 5byte 目は、レバーやボタンの入力に対応しています。
レバーやボタンのスイッチのピンは、ポート操作を行い、ON/OFF の状態を一括で取得します。ポート操作で取得した値から、応答バイトを生成します。

acknowledge()
ACK の応答信号を生成します。
ACK を LOW にし、ACK_WAIT で指定した時間(μsec)待機した後、ACK を HIGH にして、処理を終了します。
割り込み中でも待機動作を行う_delay_us()を利用します。

ISR(SPI_STC_vect)
SPI 通信完了時の割り込みハンドラです。
SS と SCK の信号に従って 1byte のデータを送受信した後呼ばれます。
SPDR から CMD を取得し、文献[1]の通信仕様に従って CMD を解釈し、SPDR へ DAT を設定します。設定した DAT は、次の通信でマスター側へ送信されます。
今何バイト目を通信しているかの情報は、関数内部で保持しており、それによって処理を分岐しています。
通信仕様では、CMD の 0x01 を検出して通信を開始し、続く CMD は 0x42 となっていますが、なぜか 0x01 の場合があり、これに応答する必要がありました。

loop()
デバッグのためにコマンドのログを出力します。それ以外は何もしません。

結果
サンワサプライ USB ゲームパッドコンバータ JY-PSUAD11 を利用し、実験を行いました。
PC 上でゲームコントローラとして認識され、〇ボタンに対応するタクトスイッチを押したところ、2 番のボタンが点灯しました。



このとき、以下のような通信を確認しました。

```
CMD: 1 42 0 0 0 DAT: FF 41 5A FF FF
CMD: 1 42 0 0 0 DAT: FF 41 5A FF FF
CMD: 1 42 0 0 0 DAT: FF 41 5A FF DF
```

3 行目の通信は、タクトスイッチを押したときのものです。
DF は、2 進数では 11011111 です。上位から 3 番目のビットは、〇ボタンに対応しているので、確かに〇ボタンが押された結果を通信しているのがわかります。
続いて、実際に PlayStation 2 に接続してみたところ、同じ結果となりました。
このように、Arduino による DUALSHOCK のエミュレートが実現できました。

部品

ハードオフ

DUALSHOCK コネクタケーブル（ジャンク） ￥300

アマゾン

サンワサプライ USB ゲームパッドコンバータ JY-PSUAD11 ￥ 2138

スイッチサイエンス

Arduino をはじめようキット￥4320

千石電商

【QI コネクタ】信号伝達コネクタ用ピン メス【F】(10 本) ￥74
【QI コネクタ】信号伝達コネクタ（黒）1×8 2550-1×8 ￥21
ピンヘッダ 1 列×40P (ストレート・標準ピッチ) 2544-1×40 15.1(6.3/6.3) ￥105

秋月電子通商

1 回路 3 ステートバッファ TC7SZ125F(TE85LF)（10 個入） ￥100
SOT23 変換基板 金フラッシュ （10 枚入） ￥150
細ピンヘッダ 1×40 （黒） ￥40

参考文献

[1] プレイステーション・ＰＡＤ／メモリ・インターフェースの解析
http://kaele.com/~kashima/games/ps_jpn.txt
[2] Arduino 日本語リファレンス - ポート操作
http://www.musashinodenpa.com/arduino/ref/index.php?f=0&pos=850
[3] AVR based PS2 digital controller
http://nfggames.com/forum2/index.php?topic=5001.0

Arduinoによる DUALSHOCKのエミュレート(2)

Arduinoで DUALSHOCKをエミュレートし、PlayStation 2のコントローラとして動作させてみます。ここでは、文献[1]を参考に、アナログモードでの動作を試みます。DUALSHOCKは、PlayStation 2のコマンドに従って、自動的にアナログモードへ遷移するため、PlayStation 2のコマンドに応答し、動作モードを変更する機能を追加します。コマンドへの応答は、 基本的にSCPH-1200をエミュレートするものとします。ハードウェア構成は、全く同じです。

コマンドの概要

コマンドは、以下のような5bytesまたは9bytesのデータとなります。

```
CMD: 0x01, 0x4X, 0x00, 0xMM, 0xNN(, 0xXX, 0xXX, 0xXX, 0xXX)
```

デジタルモードの場合5bytesのデータを、アナログモードかコンフィギュレーションモード中の場合9bytesのデータを送受信します。
感圧ボタンの機能を使用する場合は、さらに多くのデータを送受信するようです。
コマンドは、先頭の5bytesを確認すれば良いようです。コマンドの2byte目は、コマンドの種類となっています。コマンドの4byte目と5byte目は、コマンドの種類に応じて、DUALSHOCKの動作を設定する値となっています。

```
        CMD
1byte   0x01
2byte   0x4X   コマンドの種類
3byte   0x00
4byte   0xMM   コマンドの設定値1
5byte   0xNN   コマンドの設定値2
```

コマンドは、受信後直ちに指定された状態に遷移し、応答しなければならないようです。2byte目で受け取ったコマンドの種類に応じて、3byte目以降を応答する必要があります。

コマンドの種類とシーケンス

実験の結果、不明なコマンドも含めて正しく応答しなければ、アナログモードでは動作しないことがわかりました。応答しなければならないコマンドは、以下です。

```
READ_DATA             0x42 リードデータ
CONFIG_MODE           0x43 コンフィギュレーションモード
SET_MODE_AND_LOCK     0x44 セットモード＆ロック
QUERY_MODEL_AND_MODE  0x45 クエリーモデル＆モード
UNKNOWN_COMMAND_46    0x46 不明なコマンド46
UNKNOWN_COMMAND_47    0x47 不明なコマンド47
UNKNOWN_COMMAND_4C    0x4C 不明なコマンド4C
VIBRATION_ENABLE      0x4D バイブレーションイネーブル
```

リードデータコマンド以外は、コンフィギュレーションモードに関係するコマンドで

す。まず、リードデータコマンドで通信を行い、通信が確立された後、コンフィギュレーションモードに入り、各種設定を行います。その後、コンフィギュレーションモードを抜け、リードデータコマンドによる通信に復帰します。
実際のコマンドのシーケンスは、以下のようなものです。

```
01: CMD: 1 42  0  0  0              DAT: FF 41 5A FF FF              READ_DATA
02: CMD: 1 43  0  1  0              DAT: FF 41 5A FF FF              CONFIG_MODE/ENTER
03: CMD: 1 45  0  0  0  0  0  0  0 DAT: FF F3 5A  1  2  0  2  1  0 QUERY_MODEL_AND_MODE
04: CMD: 1 46  0  0  0  0  0  0  0 DAT: FF F3 5A  0  0  1  2  0  A UNKNOWN_COMMAND_46
05: CMD: 1 46  0  1  0  0  0  0  0 DAT: FF F3 5A  0  0  1  1  1 14 UNKNOWN_COMMAND_46
06: CMD: 1 47  0  0  0  0  0  0  0 DAT: FF F3 5A  0  0  2  0  1  0 UNKNOWN_COMMAND_47
07: CMD: 1 4C  0  0  0  0  0  0  0 DAT: FF F3 5A  0  0  0  4  0  0 UNKNOWN_COMMAND_4C
08: CMD: 1 4C  0  1  0  0  0  0  0 DAT: FF F3 5A  0  0  0  7  0  0 UNKNOWN_COMMAND_4C
09: CMD: 1 44  0  1  3  0  0  0  0 DAT: FF F3 5A  0  0  0  0  0  0 SET_MODE_AND_LOCK
10: CMD: 1 4D  0  0  1 FF FF FF FF DAT: FF F3 5A FF FF FF FF FF FF VIBRATION_ENABLE
11: CMD: 1 43  0  0  0  0  0  0  0 DAT: FF F3 5A FF FF 80 80 80 80 CONFIG_MODE/EXIT
12: CMD: 1 42  0  0  0  0  0  0  0 DAT: FF 73 5A FF FF 80 80 80 80 READ_DATA
```

1. リードデータコマンドとデジタルモードの応答
2. コンフィギュレーションモードに入る
3. クエリーモデル＆モードによる機種と状態の取得
4. 不明なコマンド 46
5. 不明なコマンド 46
6. 不明なコマンド 47
7. 不明なコマンド 4C
8. 不明なコマンド 4C
9. セットモード＆ロックによるアナログモードの設定
10. バイブレーションイネーブル
11. コンフィギュレーションモードを抜ける
12. リードデータコマンドとアナログモードの応答

これはタイトーメモリーズの例です。コマンドのシーケンスは、ソフトや PlayStation 2 のバージョンによって微妙に異なるようですが、概ねこれと同様だと思われます。

リードデータコマンド

コマンドは、以下のような 5bytes または 9bytes のデータとなります。

```
CMD: 0x01, 0x42, 0x00, 0xXX, 0xXX(, ...)
```

コマンドの設定値は振動機能で使用するようです。ここでは無視します。
応答するバイト列は、以下のような 5bytes または 9bytes のデータとなります。

```
DAT: 0xFF, 0xID, 0x5A, 0xXX, 0xXX(, 0xXX, 0xXX, 0xXX, 0xXX)
  ID=41: デジタルモード
  ID=73: アナログモード
  ID=F3: コンフィギュレーションモード中
```

デジタルモードの場合 5bytes のデータを、アナログモードかコンフィギュレーションモード中の場合 9bytes のデータを送受信します。DAT の 4byte 目と 5byte 目の各ビットは、以下のようにレバーとボタンの入力に対応しています。ON:0, OFF:1 です。これらはデジタルモード／アナログモードで共通です。アナログモードでは 6byte 目以降が存在し、アナログスティックの入力に対応しています。

```
        CMD   DAT  b7  b6  b5  b4  b3  b2  b1  b0
 1byte  0x01   --
 2byte  0x42  0xID  ID=41:デジタルモード ID=73:アナログモード
 3byte  0x00  0x5A
 4byte  0xXX  0xXX  左  下  右  上  ST   1   1  SE
 5byte  0xXX  0xXX  □  ×  ○  △  R1  L1  R2  L2
(6byte  0xXX  0xXX  右アナログスティック水平方向（左:0x00 中央:0x80 右:0xFF))
(7byte  0xXX  0xXX  右アナログスティック垂直方向（上:0x00 中央:0x80 下:0xFF))
(8byte  0xXX  0xXX  左アナログスティック水平方向（左:0x00 中央:0x80 右:0xFF))
(9byte  0xXX  0xXX  左アナログスティック垂直方向（上:0x00 中央:0x80 下:0xFF))
```

コンフィギュレーションモードコマンド

コマンドは、以下のような 5bytes または 9bytes のデータとなります。

```
CMD: 0x01, 0x43, 0x00, 0xMM, 0xXX(, ...)
  MM=00: コンフィギュレーションモードイグジット
  MM=01: コンフィギュレーションモードエンター
```

応答するバイト列は、リードデータコマンドと同じで良いようです。

セットモード＆ロックコマンド

コマンドは、以下のような 9bytes のデータとなります。

```
CMD: 0x01, 0x44, 0x00, 0xMM, 0xXX, ...
  MM=00: デジタルモード
  MM=01: アナログモード
```

応答するバイト列は、以下のような 9bytes のデータとなります。

```
DAT: 0xFF, 0xF3, 0x5A, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
```

クエリーモデル＆モードコマンド

コマンドは、以下のような 9bytes のデータとなります。

```
CMD: 0x01, 0x45, 0x00, 0xXX, 0xXX, ...
```

応答するバイト列は、以下のような 9bytes のデータとなります。

```
DAT: 0xFF, 0xF3, 0x5A, 0x01, 0x02, 0xMM, 0x02, 0x01, 0x00
  MM=00: デジタルモード
  MM=01: アナログモード
```

不明なコマンド 46
コマンドは、以下のような 9bytes のデータとなります。

```
CMD: 0x01, 0x46, 0x00, 0xMM, 0xXX, ...
  MM=00: 不明な設定 0
  MM=01: 不明な設定 1
```

DUALSHOCK(SCPH-1200)の応答を解析したところ、以下のようなものでした。
不明な設定 0 に応答するバイト列は、以下のような 9bytes のデータとなります。

```
DAT: 0xFF, 0xF3, 0x5A, 0x00, 0x00, 0x01, 0x02, 0x00, 0x0A
```

不明な設定 1 に応答するバイト列は、以下のような 9bytes のデータとなります。

```
DAT: 0xFF, 0xF3, 0x5A, 0x00, 0x00, 0x01, 0x01, 0x01, 0x14
```

不明なコマンド 47
コマンドは、以下のような 9bytes のデータとなります。

```
CMD: 0x01, 0x47, 0x00, 0xXX, 0xXX, ...
```

応答するバイト列は、以下のような 9bytes のデータとなります。

```
DAT: 0xFF, 0xF3, 0x5A, 0x00, 0x00, 0x02, 0x00, 0x01, 0x00
```

不明なコマンド 4C
コマンドは、以下のような 9bytes のデータとなります。

```
CMD: 0x01, 0x4C, 0x00, 0xMM, 0xXX, ...
  MM=00: 不明な設定 0
  MM=01: 不明な設定 1
```

応答するバイト列は、以下のような 9bytes のデータとなります。

```
DAT: 0xFF, 0xF3, 0x5A, 0x00, 0x00, 0x00, 0xNN, 0x00, 0x00
  NN=04: 不明な設定 0
  NN=07: 不明な設定 1
```

バイブレーションイネーブルコマンド
コマンドは、以下のような 9bytes のデータとなります。

```
CMD: 0x01, 0x4D, 0x00, 0xXX, 0xXX, ...
```

応答するバイト列は、以下のような 9bytes のデータとなります。

```
DAT: 0xFF, 0xF3, 0x5A, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF
```

ソースコード

```
// SPI test PS2: Master, Arduino: Slave (ANALOG MODE)
//
// Copyright (c) 2015 Kazumasa ISE
// Released under the MIT license
// http://opensource.org/licenses/mit-license.php

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

#define DEBUG
#define ACK_WAIT 0.5
#define ACK 9
#define SET_ACK_LOW  (PORTB &= ~B00000010)
#define SET_ACK_HIGH (PORTB |=  B00000010)
#define SW_LEFT      7
#define SW_DOWN      6
#define SW_RIGHT     5
#define SW_UP        4
#define SW_START     3
#define SW_SQUARE   19
#define SW_CROSS    18
#define SW_CIRCLE   17
#define SW_TRIANGLE 16
#define SW1 (PIND | B00000111)
#define SW2 ((PINC << 2) | B00001111)
#define READ_DATA            0x42
#define CONFIG_MODE          0x43
#define SET_MODE_AND_LOCK    0x44
#define QUERY_MODEL_AND_MODE 0x45
#define UNKNOWN_COMMAND_46   0x46
#define UNKNOWN_COMMAND_47   0x47
#define UNKNOWN_COMMAND_4C   0x4C
#define VIBRATION_ENABLE     0x4D
#define CMD_BYTES 9

volatile bool isAnalogMode = false;
volatile bool isConfigMode = false;
volatile bool unknownFlag = false;
volatile byte RH = 0x80;
volatile byte RV = 0x80;
volatile byte LH = 0x80;
volatile byte LV = 0x80;

void setup() {
 // SPI setup
 SPI.setBitOrder(LSBFIRST);
 SPI.setDataMode(SPI_MODE3);
 SPCR &= ~(_BV(MSTR)); // Set as Slave
 SPCR |= _BV(SPE); // Enable SPI
 pinMode(SS, INPUT);
 pinMode(MOSI, INPUT);
```

```
 pinMode(MISO, OUTPUT);
 digitalWrite(MISO, HIGH);
 pinMode(SCK, INPUT);
 SPI.attachInterrupt();
 // ACK pin setup
 pinMode(ACK, OUTPUT);
 digitalWrite(ACK, HIGH);
 // Switches setup
 pinMode(SW_LEFT, INPUT_PULLUP);
 pinMode(SW_DOWN, INPUT_PULLUP);
 pinMode(SW_RIGHT, INPUT_PULLUP);
 pinMode(SW_UP, INPUT_PULLUP);
 pinMode(SW_START, INPUT_PULLUP);
 pinMode(SW_SQUARE, INPUT_PULLUP);
 pinMode(SW_CROSS, INPUT_PULLUP);
 pinMode(SW_CIRCLE, INPUT_PULLUP);
 pinMode(SW_TRIANGLE, INPUT_PULLUP);
#ifdef DEBUG
 Serial.begin(115200);
#endif
}

inline byte readDataResponse(byte i) {
 const byte DAT[] = {0xFF, 0x41, 0x5A, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF};
 switch (i) {
 case 1: return isConfigMode ? 0xF3 : (isAnalogMode ? 0x73 : 0x41);
 case 3: return SW1;
 case 4: return SW2;
 case 5: return RH;
 case 6: return RV;
 case 7: return LH;
 case 8: return LV;
 default: return DAT[i];
 }
}

inline byte setModeAndLockResponse(byte i) {
 const byte DAT[] = {0xFF, 0xF3, 0x5A, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00};
 return DAT[i];
}

inline byte queryModelAndModeResponse(byte i) {
 const byte DAT[] = {0xFF, 0xF3, 0x5A, 0x01, 0x02, 0x00, 0x02, 0x01,
0x00};
 switch (i) {
```

```
 case 5: return isAnalogMode ? 0x01 : 0x00;
 default: return DAT[i];
 }
}

inline byte unknownCommand46Response(byte i) {
 const byte DAT[] = {0xFF, 0xF3, 0x5A, 0x00, 0x00, 0x01, 0x02, 0x00,
0x0A};
 switch (i) {
 case 6: return unknownFlag ? 0x01 : 0x02;
 case 7: return unknownFlag ? 0x01 : 0x00;
 case 8: return unknownFlag ? 0x14 : 0x0A;
 default: return DAT[i];
 }
}

inline byte unknownCommand47Response(byte i) {
 const byte DAT[] = {0xFF, 0xF3, 0x5A, 0x00, 0x00, 0x02, 0x00, 0x01,
0x00};
 return DAT[i];
}

inline byte unknownCommand4CResponse(byte i) {
 const byte DAT[] = {0xFF, 0xF3, 0x5A, 0x00, 0x00, 0x00, 0x04, 0x00,
0x00};
 switch (i) {
 case 6: return unknownFlag ? 0x07 : 0x04;
 default: return DAT[i];
 }
}

inline byte vibrationEnableResponse(byte i) {
 const byte DAT[] = {0xFF, 0xF3, 0x5A, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF};
 return DAT[i];
}

inline byte dat(const byte CMD[], byte i) {
 switch (CMD[1]) {
 case READ_DATA:
  return readDataResponse(i);
 case CONFIG_MODE:
  isConfigMode = (CMD[3] == 0x01);
  return readDataResponse(i);
 case SET_MODE_AND_LOCK:
  isAnalogMode = (CMD[3] == 0x01);
  return setModeAndLockResponse(i);
 case QUERY_MODEL_AND_MODE:
  return queryModelAndModeResponse(i);
```

```
 case UNKNOWN_COMMAND_46:
  unknownFlag = (CMD[3] == 0x01);
  return unknownCommand46Response(i);
 case UNKNOWN_COMMAND_47:
  return unknownCommand47Response(i);
 case UNKNOWN_COMMAND_4C:
  unknownFlag = (CMD[3] == 0x01);
  return unknownCommand4CResponse(i);
 case VIBRATION_ENABLE:
  return vibrationEnableResponse(i);
 default:
  return readDataResponse(i);
 }
}

inline void acknowledge() {
 SET_ACK_LOW;
 _delay_us(ACK_WAIT);
 SET_ACK_HIGH;
}

#ifdef DEBUG
#define LINE_FEED 0xAA
#define MAX_LOG_SIZE 300
volatile byte cmdLog[MAX_LOG_SIZE] = {0};
volatile byte datLog[MAX_LOG_SIZE] = {0};
volatile int logCount = 0;
#endif

ISR(SPI_STC_vect) {
 static byte ID = 0x41;
 static byte CMD[CMD_BYTES] = {0};
 static byte cmdCount = 0;
 bool continueCom = false;
 CMD[cmdCount] = SPDR;
#ifdef DEBUG
 if (logCount < MAX_LOG_SIZE) {cmdLog[logCount++] = CMD[cmdCount];}
#endif
 const byte numOfCmd = 3+2*(ID & 0x0F);
 // Check CMD
 if (cmdCount == 0) {
  if (CMD[cmdCount] == 0x01) {
   continueCom = true;
  }
 } else if (cmdCount == 1) {
  if (CMD[cmdCount] == 0x01) {
   cmdCount = 0; // Reset count
   continueCom = true;
```

```
  } else if ((CMD[cmdCount] & 0x40) == 0x40) {
   continueCom = true;
  }
 } else if (cmdCount == 5) {
  if ((CMD[1] == READ_DATA) && (CMD[cmdCount] == 0x01)) {
   cmdCount = 0; // Reset count
  }
  continueCom = true;
 } else if (cmdCount < numOfCmd-1) {
  continueCom = true;
 }
#ifdef DEBUG
 if (!continueCom && (logCount < MAX_LOG_SIZE)) {cmdLog[logCount++] =
LINE_FEED;}
#endif
 // Set next DAT
 cmdCount = continueCom ? cmdCount+1 : 0;
 const byte DAT = dat(CMD, cmdCount);
 if (cmdCount == 1) {ID = DAT;}
 SPDR = DAT;
#ifdef DEBUG
 if (logCount < MAX_LOG_SIZE) {datLog[logCount] = DAT;}
#endif
 if (continueCom) {acknowledge();}
}

void loop() {
#ifdef DEBUG
 if (logCount == MAX_LOG_SIZE) {
  logCount = 0;
  int lfPos = -1;
  for (int i=0; i<MAX_LOG_SIZE; i++) {
   if (cmdLog[i] == LINE_FEED) {
    Serial.print("DAT: ");
    for (int j=lfPos+1; j<i; j++) {
     Serial.print(datLog[j], HEX);
     Serial.print(" ");
    }
    lfPos=i;
    Serial.println();
    Serial.print("CMD: ");
   } else {
    Serial.print(cmdLog[i], HEX);
    Serial.print(" ");
   }
  }
```

```
 }
#endif
}
```

文献[1]を参考に、アナログモードでの動作を試みます。

setup()
初期化を行います。
デジタルモードのプログラムと全く同じです。

readDataResponse()
リードデータコマンドと、コンフィギュレーションモードコマンドに応答するバイト列を生成します。これらの応答は同じで良いようです。

setModeAndLockResponse()
セットモード＆ロックコマンドに応答するバイト列を生成します。

queryModelAndModeResponse()
クエリーモデル＆モードコマンドに応答するバイト列を生成します。

unknownCommand46Response()
不明なコマンド 46 に応答するバイト列を生成します。

unknownCommand47Response()
不明なコマンド 47 に応答するバイト列を生成します。

unknownCommand4CResponse()
不明なコマンド 4C に応答するバイト列を生成します。

vibrationEnableResponse()
バイブレーションイネーブルコマンドに応答するバイト列を生成します。

dat()
SPI 通信で応答するバイト列を生成します。
コマンドの種類に応じて、応答するバイト列を生成する処理に分岐します。また、コマンドの設定値に応じて、デジタルモード／アナログモードなどの設定を行います。

acknowledge()
ACK の応答信号を生成します。
デジタルモードのプログラムと全く同じです。

ISR(SPI_STC_vect)
SPI 通信完了時の割り込みハンドラです。
デジタルモードのプログラムと殆ど同じですが、応答するバイト列の ID をモードに

従って変更し、送受信するデータ量を決定するよう、処理を変更しています。
通信仕様では、ID の下位 4 ビットで送受信するデータ量を決定しますが、リードデータコマンドの場合、ID に関係なく 5bytes を通信する場合があり、これに対応する必要がありました。

loop()

デバッグのためにコマンドのログを出力します。それ以外は何もしません。
デジタルモードのプログラムと全く同じです。

結果

PlayStation 2(SCPH-30000, SCPH-90000)に接続し、実験を行いました。
カプコンクラシックスコレクションの他、タイトーメモリーズなど、幾つかのソフトで問題なくアナログモードに遷移し、動作することを確認しました。

参考文献

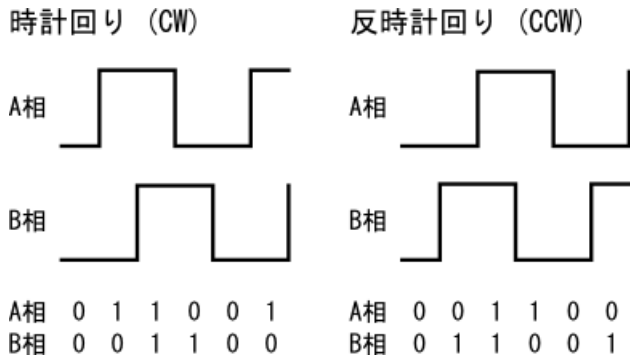
[1] デュアルショック(SCPH-1200)の解析
https://applause.elfmimi.jp/dualshock.txt

ロータリエンコーダの信号処理

ロータリエンコーダの信号を、DUALSHOCK のアナログスティックを想定した XY2 軸の信号に変換します。

ロータリエンコーダの回転角

ロータリエンコーダには、A 相と B 相の信号があり、B 相の信号は、A 相から 1/4 周期位相がずれています。



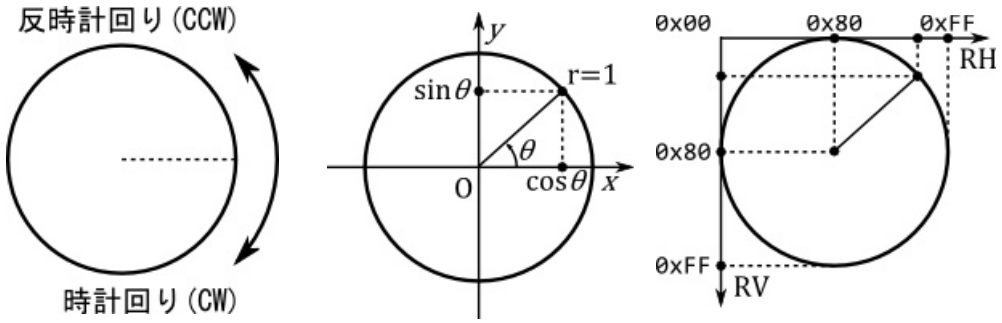
A 相と B 相の両方の信号の立ち上がりと立ち下がりで回転を検出できるため、1 回転あたりのパルス数は、見かけ上 4 倍となります。
今回使用するロータリエンコーダ RES20D-50-201-1 は、1 回転あたり 50 パルスなので、見かけ上のパルスは 50x4 = 200 パルス となり、1 パルスあたりの分解能は 360/200 = 1.8 度 となります。
A 相の信号と、信号が変化する直前の B 相の信号を比較すると、時計回り（CW）では常に異なり、反時計回り（CCW）では常に同じとなっています。
A 相を上位ビット、B 相を下位ビットとする 2 ビットの値を考えると、信号が変化する直前の下位ビットと、信号が変化した直後の上位ビットの排他的論理和（XOR）は、CW では常に 1、CCW では常に 0 です。

```
    AB AB AB AB AB AB AB AB ...
CW : 00 10 11 01 00 10 11 01 ...
CCW: 00 01 11 10 00 01 11 10 ...
```

これにより、回転の方向を検出することができます。
回転の方向と、見かけ上のパルス数から、回転角を算出できます。

回転角に対応するアナログスティックの傾き量

反時計回りを正とした回転角 θ に対応する x と y の値は、x = cosθ、y = sinθ です。
この x と y の値が、右アナログスティックの左右の傾き量 RH と、上下の傾き量 RV となります。
x と y の値-1〜1 を、アナログスティックの傾き量の値 0x00〜0xFF に置き換えます。

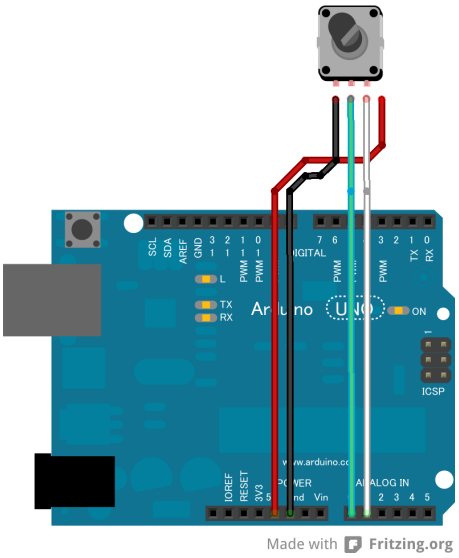


これにより、ロータリエンコーダの回転動作を、アナログスティックの回転動作とすることができます。

部品配置図

今回使用するロータリエンコーダ RES20D-50-201-1 は、光学式で、電源は 5V です。
信号線は、4.7kΩ でプルアップされています[1]。
A 相 B 相の信号は、ポート操作[2]を行い、HI/LOW の状態を一括で取得します。ビットの並びをあわせることで、処理の手間が省けます。
ポート C（アナログピン 0~1）を A 相 B 相の信号に割り当てます。

```
PINC: A5  A4  A3  A2  A1  A0
      --  --  --  --   A   B
```



赤：5V
白：A 相
緑：B 相
黒：GND

ソースコード

```
// Rotary encoder to Analog stick test
//
// Copyright (c) 2015 Kazumasa ISE
// Released under the MIT license
// http://opensource.org/licenses/mit-license.php

#define DEBUG
#define RENC_A 15
#define RENC_B 14
#define RENC_AB (PINC & B00000011)
#define PULSE_NUM (50*4)

volatile byte RH = 0x80;
volatile byte RV = 0x80;

byte analogStickTableH[PULSE_NUM];
byte analogStickTableV[PULSE_NUM];

inline float degToRad(float deg) {
 return 2.0 * M_PI * deg / 360.0;
}

inline byte analogValue(float data) {
 const float temp = (data + 1.0) / 2.0; // -1~1 -> 0~1
 return (byte)(255 * temp + 0.5); // 0~1 -> 0~255
}

inline void initAnalogStickTable() {
 for (int i = 0; i < PULSE_NUM; i++) {
  const float deg = i * 360.0 / PULSE_NUM;
  const float rad = degToRad(deg);
  const float x = cos(rad);
  const float y = sin(rad);
  analogStickTableH[i] = analogValue( x);
  analogStickTableV[i] = analogValue(-y);
 }
}

inline int rotationSignum(byte curr) {
 static byte prev = 0;
 int signum = 0;
 if (prev != curr) {
  const bool cw = ((prev << 1) ^ curr) & B00000010;
  signum = cw ? -1 : 1;
  prev = curr;
 }
 return signum;
}

inline int rotationPulseCount(int signum) {
 static int count = 0;
```

```
 count += signum;
 count = (count + PULSE_NUM) % PULSE_NUM;
 return count;
}

void setup() {
 // Rotary encoder setup
 pinMode(RENC_A, INPUT);
 pinMode(RENC_B, INPUT);
 initAnalogStickTable();
#ifdef DEBUG
 Serial.begin(115200);
#endif
}

void loop() {
 const int signum = rotationSignum(RENC_AB);
 const int count = rotationPulseCount(signum);
 RH = analogStickTableH[count];
 RV = analogStickTableV[count];
#ifdef DEBUG
 if (signum != 0) {
  Serial.print("count: ");
  Serial.print(count);
  Serial.print(" RH: ");
  Serial.print(RH);
  Serial.print(" RV: ");
  Serial.print(RV);
  Serial.println();
 }
#endif
}
```

degToRad()
角度の値を、度からラジアンに変換します。

analogValue()
x と y の値-1〜1 を、アナログスティックの傾き量の値 0x00〜0xFF に変換します。

initAnalogStickTable()
アナログスティックの傾き量テーブルを初期化します。
傾き量テーブルは、ロータリエンコーダの、見かけ上のパルス数に対応する、アナログスティックの傾き量の値です。

rotationSignum()
ロータリエンコーダの、回転方向の符号を生成します。

直前の A 相 B 相の値を保持し、回転の方向を検出します。
CCW なら 1 を、CW なら-1 を、回転していないなら 0 を出力します。

rotationPulseCount()
ロータリエンコーダの、見かけ上のパルス数をカウントします。
CCW ならカウントをインクリメントし、CW ならカウントをデクリメントします。
カウント値は回転角に相当し、最大値を超えない範囲で循環するようにします。

```
CCW: 0 1 2 ... 198 199 0 1 2 ...
```

setup()
初期化を行います。
ロータリエンコーダの A 相 B 相の信号ピンを設定します。
アナログスティックの傾き量テーブルを初期化します。
デバッグのためにログ出力の準備をします。

loop()
ロータリエンコーダの、見かけ上のパルス数を、ポーリングによりカウントします。
ロータリエンコーダの信号処理は、通常は割り込みを使用しますが、割り込みは他で使用する予定があるため、ここではポーリングにより処理を行います。
デバッグのためにログを出力します。

結果

ロータリエンコーダを反時計回りに回転すると、以下のような結果となりました。

```
count: 0 RH: 255 RV: 128
count: 1 RH: 255 RV: 123
（中略）
count: 49 RH: 132 RV: 0
count: 50 RH: 127 RV: 0
count: 51 RH: 123 RV: 0
（中略）
count: 99 RH: 0 RV: 123
count: 100 RH: 0 RV: 128
count: 101 RH: 0 RV: 132
（中略）
count: 149 RH: 123 RV: 255
count: 150 RH: 128 RV: 255
count: 151 RH: 132 RV: 255
```

回転角 0 度(count = 0)では、アナログスティックは右(RH = 255, RV = 128)、
回転角 90 度(count = 50)では、アナログスティックは上(RH = 127, RV = 0)、
回転角 180 度(count = 100)では、アナログスティックは左(RH = 0, RV = 128)、
回転角 270 度(count = 150)では、アナログスティックは下(RH = 128, RV = 255)、
となっているのがわかります。このように、信号処理と値の変換が確認できました。

部品

ビットトレードワン

日本電産コパル電子 ロータリエンコーダ[RES20D-50-201-1] ￥1250

スイッチサイエンス

Arduino をはじめようキット￥4320

千石電商

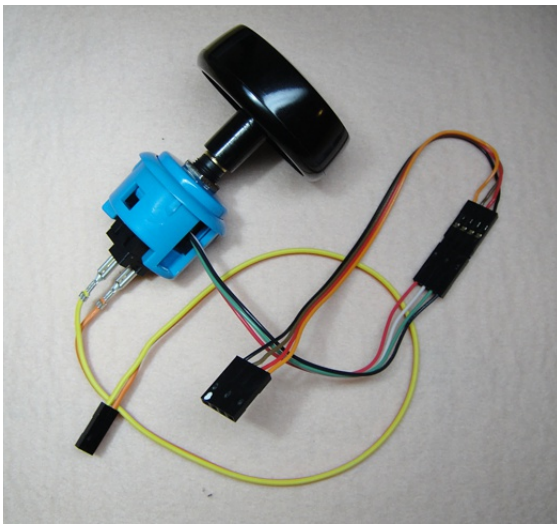
【QI コネクタ】信号伝達コネクタ用ピン メス【F】(10 本) ￥74×1
【QI コネクタ】信号伝達コネクタ（黒）1×4 2550-1×4 ￥21×1
ピンヘッダ 1 列×40P (ストレート・標準ピッチ) 2544-1×40 15.1(6.3/6.3) ￥105

参考文献

[1] 設定用光学式エンコーダ REC/RES カタログ PDF - Copal Electronics
http://www.nidec-copal-electronics.com/files/f942ed53125803ced27338faf050ac566fe48106.pdf
[2] Arduino 日本語リファレンス - ポート操作
http://www.musashinodenpa.com/arduino/ref/index.php?f=0&pos=850

ローリングスイッチユニットの製作

ローリングスイッチユニットの基本的な構造は、新方式ローリングスイッチ製作[1]を参考にさせていただきました。非常に画期的な構造で、操作感覚も良好なものでしたが、残念ながら正常に動作しなかったため、改良を加えました。



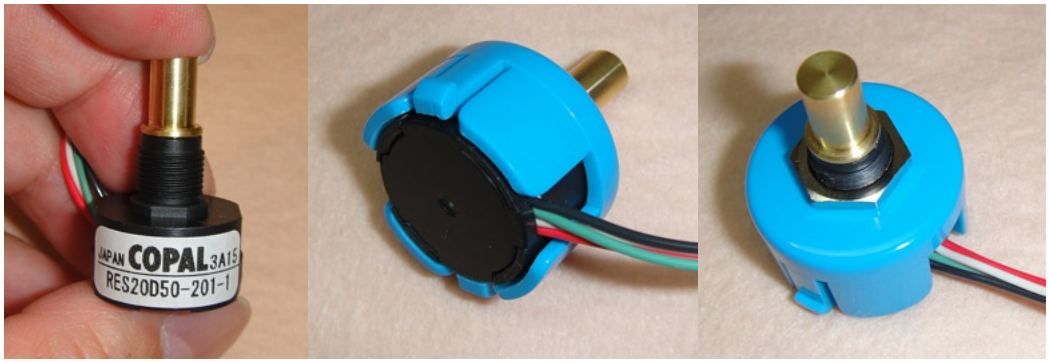
部品として指定されていた、秋月電子通商で入手可能なロータリーエンコーダ RE160F-40E3-20A-24P は、機械接点が ON/OFF する摺動（しゅうどう）式で、チャタリングやバウンスが避けられません。データシートを見ると、バウンスは 2ms、チャタリングは 3ms とあり、不感時間を設けて対策することになります。
実験の結果、ロータリーエンコーダを素早く回転した場合、信号を取りこぼしてしまい、回転を検出できなくなることがわかりました。
このため、ロータリーエンコーダは、チャタリングやバウンスの無い光学式のものを使用することにしました。

ボタン上部の加工

三和電子のボタン OBSF-30-B の上部を取り外し、加工します。内部の構造をニッパーで切り取り、ドリルで中心に 9mm の穴を開けます。

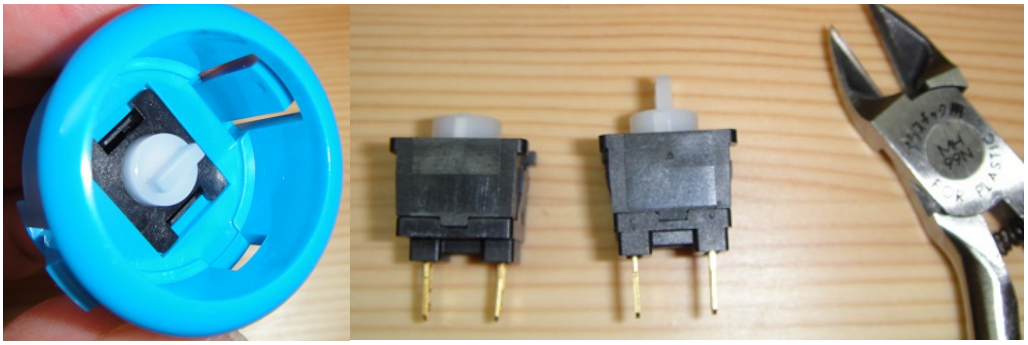
ロータリーエンコーダの取り付け

加工した部品に、日本電産コパル電子のロータリエンコーダ RES20D-50-201-1 を取り付けます。このように、ピッタリと収まります。



ボタン基部の加工

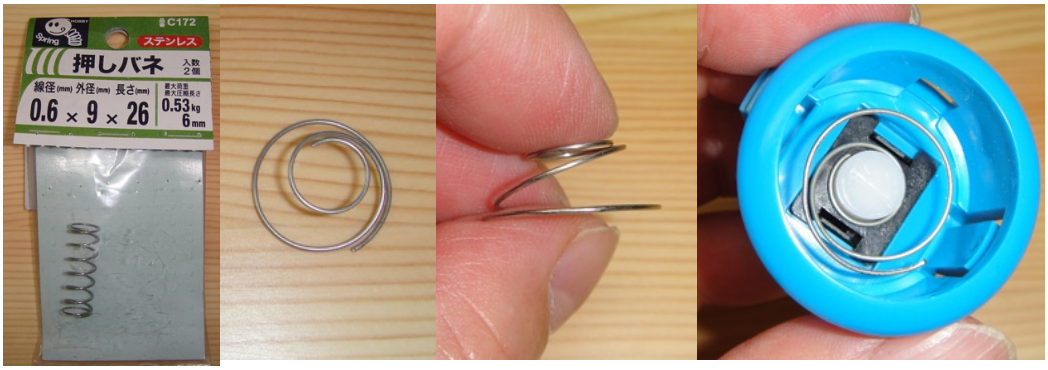
スイッチ部品を一旦取り外し、突起をニッパーで切り取り、元に戸します。



両脇にあるツメの一方を、ニッパーで切り取ります。ここから、ロータリーエンコーダのケーブルを引き出します。

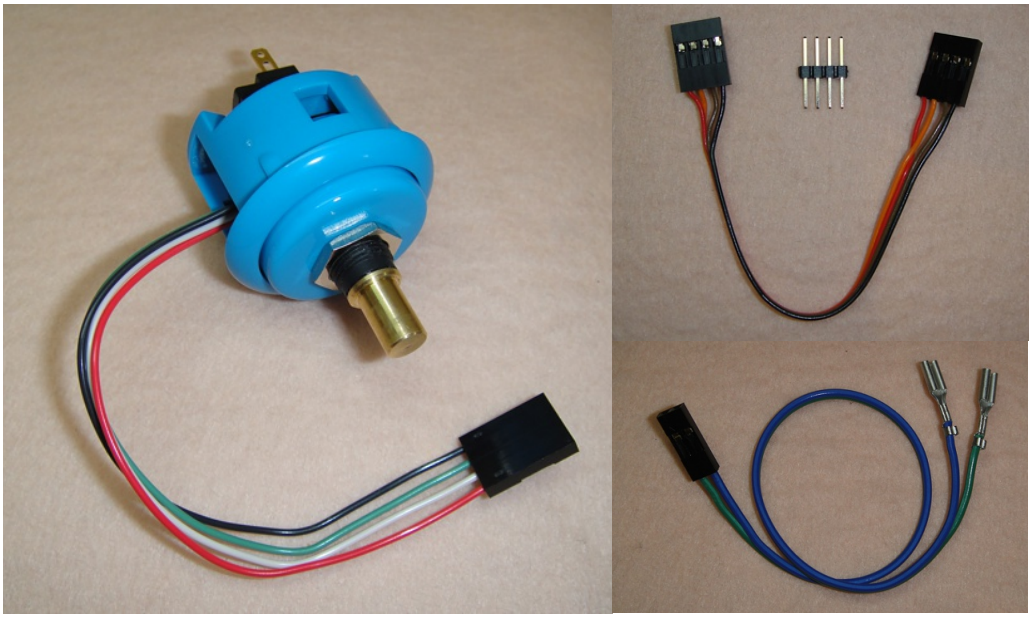
バネの加工と取り付け

バネを加工し、ボタン基部に取り付けます。加工するバネは、八幡ねじ 押しバネ 0.6×9×26 (線径×外径×長さ)が手ごろでした。これを 1/2 に切断し、ペンチでらせん状に曲げ、スイッチ部品に取り付けます。バネを切断する際は、鋼線を切断できるニッパーでなければ刃を傷めてしまうので、注意して下さい。

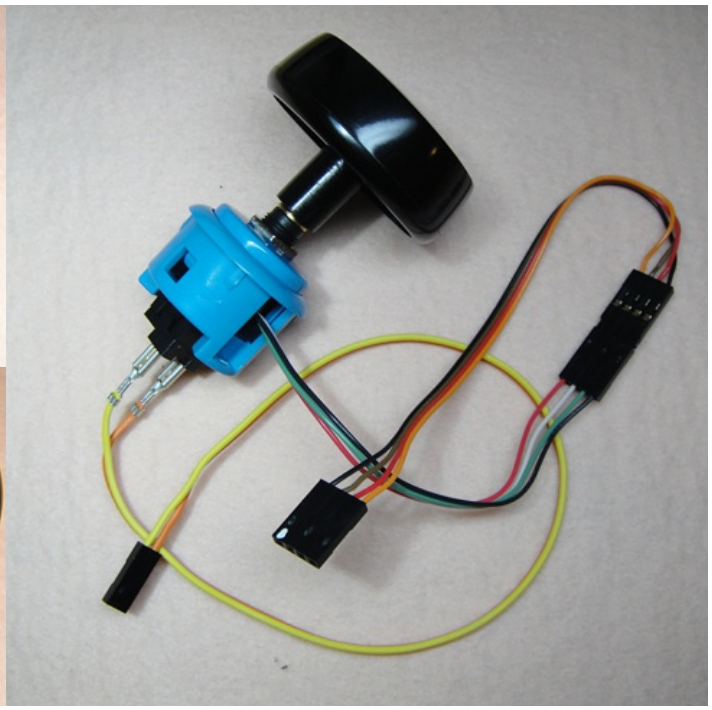


各部品の取り付け

ボタン基部にボタン上部を取り付けます。一旦取り付けてしまうと、取り外すのが困難なので、注意して下さい。ロータリーエンコーダのケーブルは、切り取ったツメの穴から引き出し、QI コネクタで端末処理を行います。ケーブルが短いので、延長用のハーネスを製作します。ケーブルは AWG26 の太さが手ごろでした。また、ファストン端子を圧着した、ボタンの端子用ハーネスも製作します。こちらは少し太めの AWG22 が手ごろでした。



ロータリーエンコーダの軸に、鍋ツマミを取り付けます。鍋ツマミは、デリッシュ鍋ツマミ LL が丁度良い大きさでした。中心に始めから開いているネジ穴に、ドリルで 6mm の穴を開け、取り付けます。かなりきついので、注意して下さい。
全ての部品を取り付けて完成です。

部品

ビットトレードワン

日本電産コパル電子 ロータリエンコーダ[RES20D-50-201-1] ￥1250

千石電商

三和電子 φ30 ゲーム SW（はめ込み） 青 OBSF-30-B（青） ￥173
日本圧着端子製造(JST) ファストン端子#110（メス） LTO-41T-110N (10 個) ￥110
【QI コネクタ】信号伝達コネクタ用ピン メス【F】(10 本) ￥74×2
【QI コネクタ】信号伝達コネクタ（黒）1×4 2550-1×4 ￥21×3
【QI コネクタ】信号伝達コネクタ（黒）1×2 2550-1×2 ￥21
ピンヘッダ 1 列×40P (ストレート・標準ピッチ) 2544-1×40 15.1(6.3/6.3) ￥105
協和ハーモネット RKV 0.3×4 列 L-1 4 芯リボンケーブル 1m ￥250
協和ハーモネット RKV 10/0.12×10 列 L-1 10 芯リボンケーブル 1m ￥280

ジョイフル本田

八幡ねじ 押しバネ 0.6×9×26 (線径×外径×長さ) [入数 2] ￥162
デリッシュ 鍋ツマミ LL ￥204

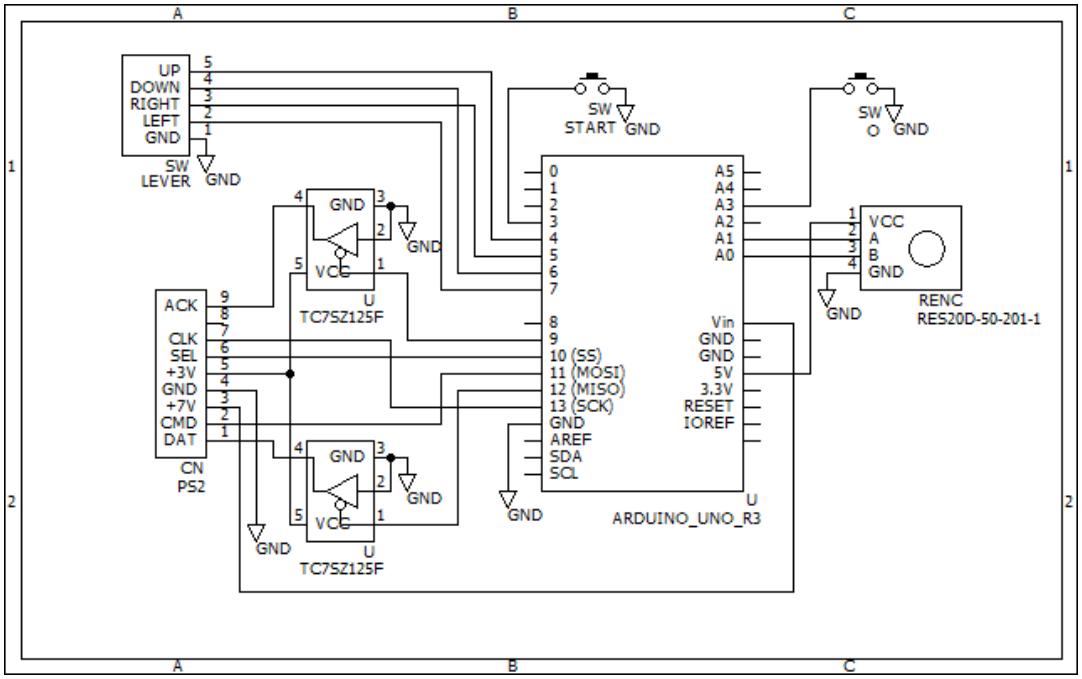
参考文献

[1] 新方式ローリングスイッチ製作
http://extendead.web.fc2.com/iroiro/R_SW/R_SW.html

ローリングスイッチシールドの製作

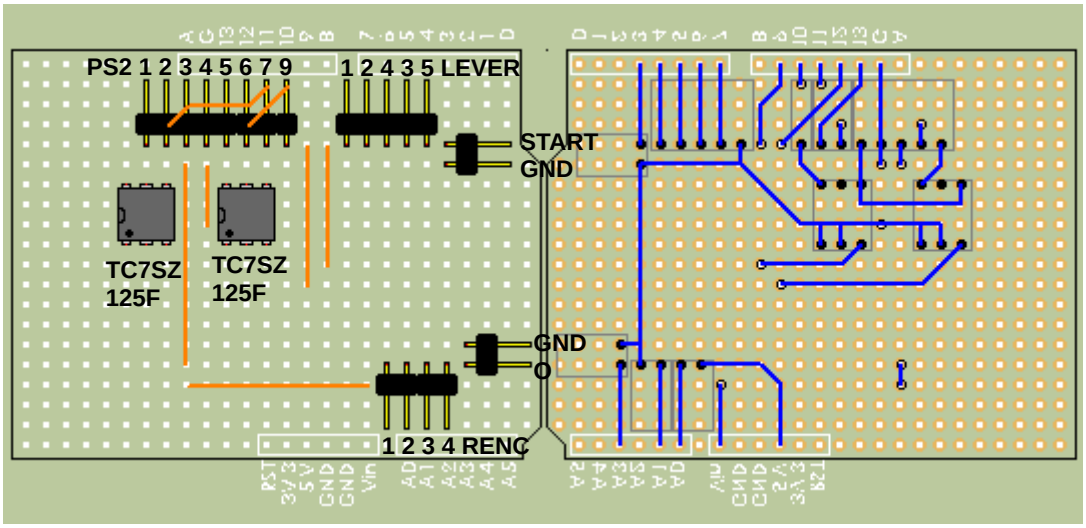
Arduino のローリングスイッチシールドを製作します。ブレッドボードで実験した回路を、Arduino 用ユニバーサル プロトシールド基板上に実装します。
レバーとボタンとローリングスイッチユニットを接続するコネクタを配線します。

回路図 1



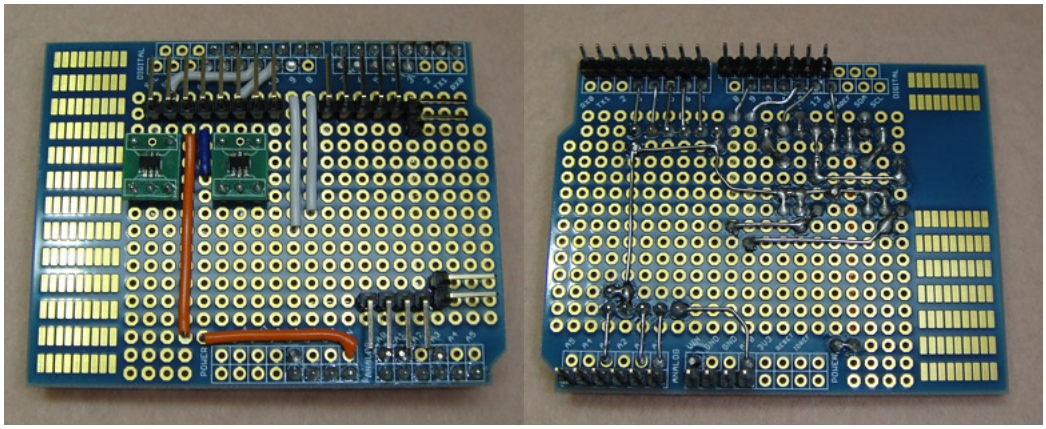
部品配置図 1

基板表面　　基板裏面



完成写真 1

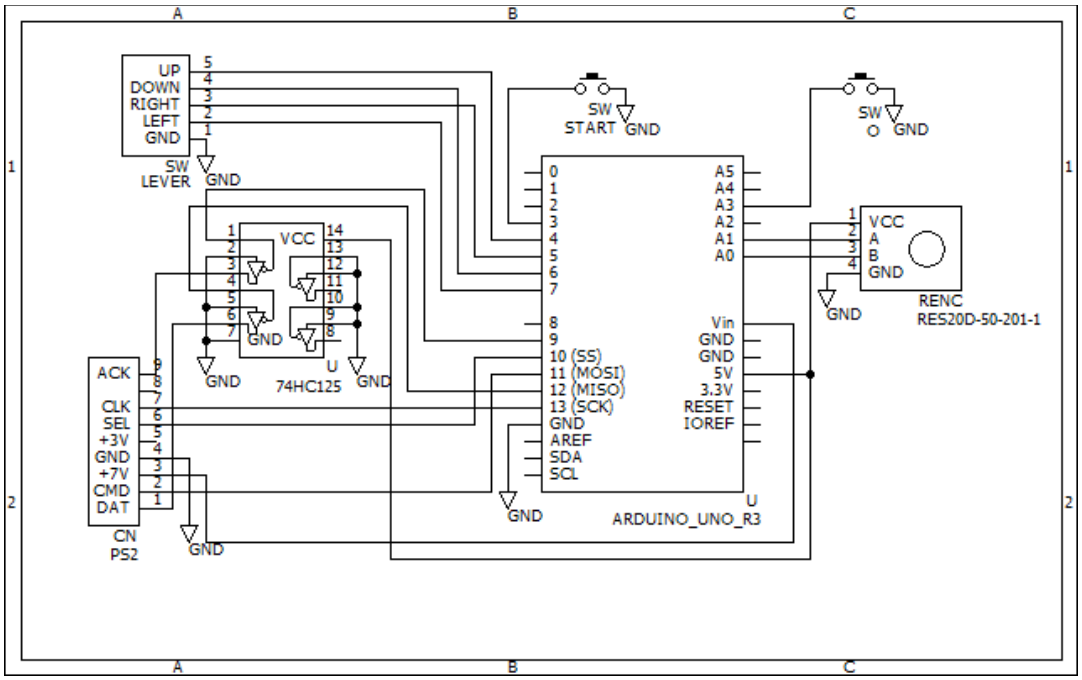
基板表面　　基板裏面



ロータリエンコーダのピン配置が逆転してしまっているので注意して下さい。このまま製作すると、サテライトが逆向きに回転してしまいます。ハーネスのコネクタを入れ替えるなどして対応して下さい。

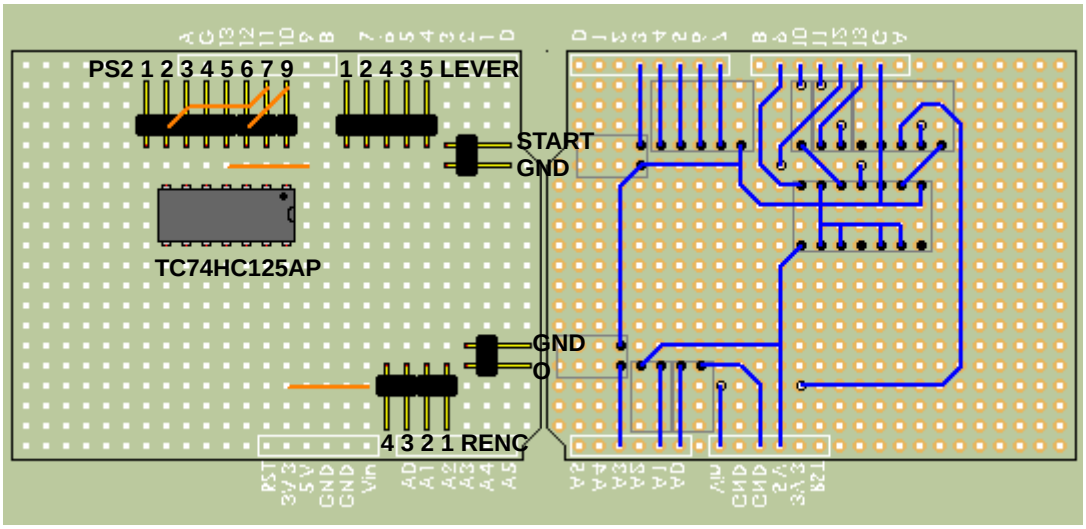
回路図 2

代替部品による回路です。TC7SZ125F の代わりに TC74HC125AP を使用しています。オープンドレインの出力を 3V でプルアップするのであれば、5V で動作させても問題ないようです。こちらの方が簡単に製作できると思います。



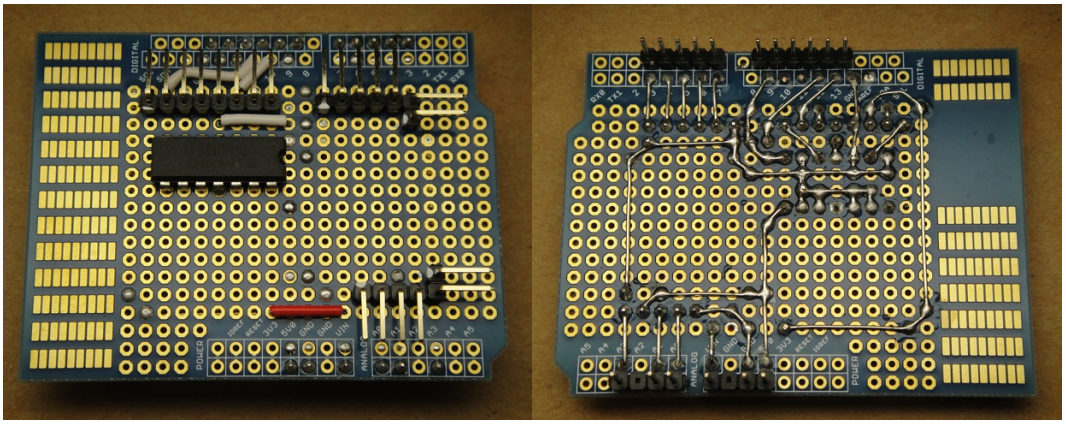
部品配置図 2

基板表面　　基板裏面



完成写真 2

基板表面　　基板裏面



ロータリエンコーダのピン配置が逆転してしまっていたのを修正しました。

その他

コネクタは QI コネクタと標準的な 2.54mm ピッチのピンヘッダで実現していますが、使用した L 型のピンヘッダが、通常のものとは異なる形のものでした。千石電商で入手可能ですが、ジョイスティック本体内に収まるのであれば、ストレートのピンヘッダでも良いと思います。ジャンパ線は、耐熱通信機器用ビニル電線を利用しましたが、もっと安価な部品でも良いと思います。 部品配置図に BANANAWANI MICOM. CULB さんの Arduino プロトシールドの PasS 用データ[1]を使用させていただきました。

部品 1

秋月電子通商

1 回路 3 ステートバッファ TC7SZ125F(TE85LF)（10 個入） ￥100
SOT23 変換基板 金フラッシュ （10 枚入） ￥150
細ピンヘッダ 1×40 （黒） ￥40

部品 2

千石電商

東芝 CMOS ロジック IC TC74HC125AP ￥84

共通部品

千石電商

2545-1×40 ピンヘッダ 1 列×40P (L 型・標準ピッチ) ￥84

秋月電子通商

Arduino 用ユニバーサル プロトシールド基板 ￥200
ピンヘッダ 1×40 (40P) ￥40
耐熱通信機器用ビニル電線 2m×10 色 外径 0.65mm ￥620
スズメッキ線(0.6mm 10m) ￥210
鉛フリーハンダ 0.8mm ￥280

参考文献

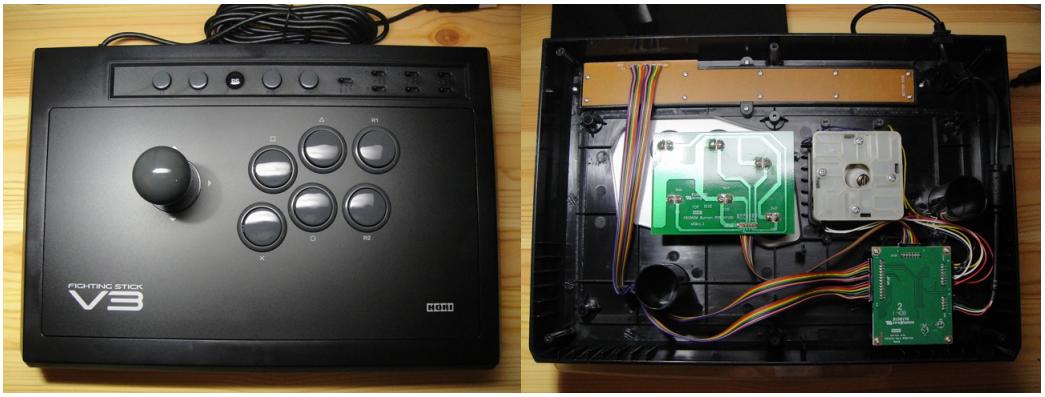
[1] BANANAWANI MICOM. CULB: Arduino プロトシールドの PasS 用データ
http://bananawani-mc.blogspot.jp/2010/10/arduinopass.html

ローリングスイッチの製作

製作したローリングスイッチユニットと、Arduino＋ローリングスイッチシールドを、ジョイスティック本体に組み込みます。ジョイスティック本体は、三和電子のボタン OBSF-30-B が取り付けられるものであれば、好きなものを利用して良いのですが、中古を除くと、選択肢は少ないと思います。HORI リアルアーケード Pro が確実なのですが、ここでは、HORI ファイティングスティック V3 を利用しました。

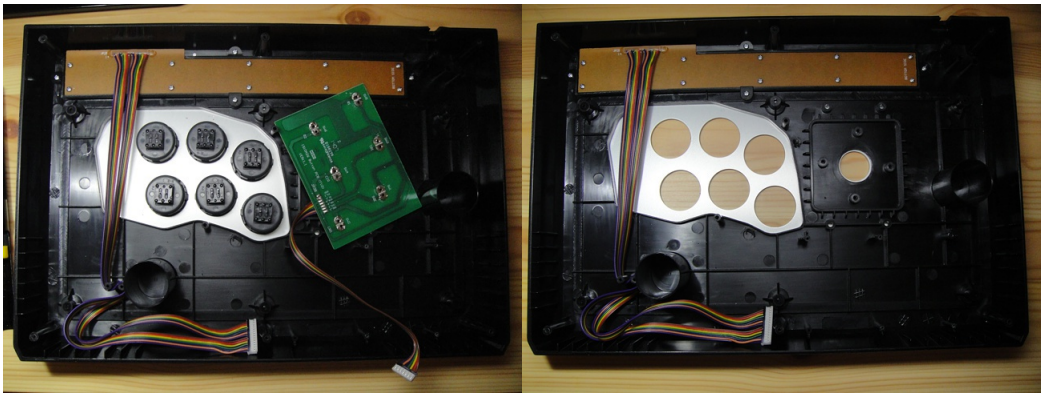
部品の取り外し

折角なので、文献[1]を参考に、レバーを三和電子のレバー JLF-TP-8Y に交換することにします。裏面のネジを外し、裏蓋を取り外します。レバーとボタンとコントロール基板があるのが分かります。これらを全て取り外します。
レバーとコントロール基板は、ネジで固定されているだけなので、ネジを外すと取り外せます。レバーの握り玉は、シャフトがネジになっていて、固定されています。
シャフトの溝にマイナスドライバーを差し込んで回すと、握り玉が外せます。



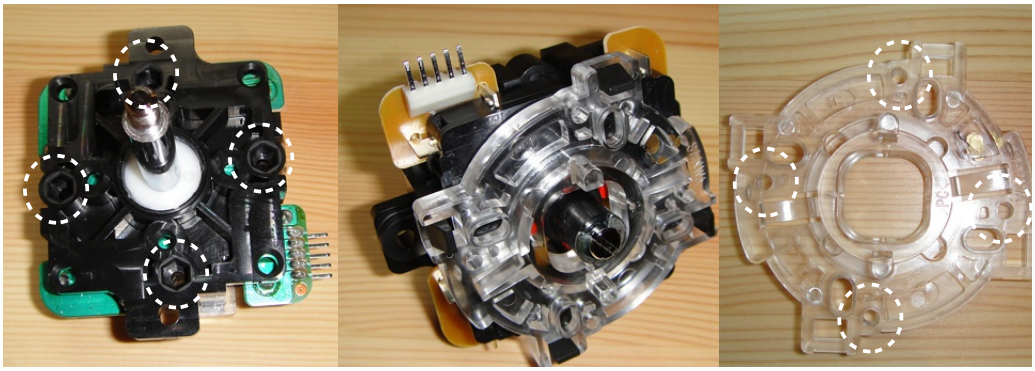
ボタンは基板に半田付けされているので、半田吸い取り器を使って基板を外し、一個

ずつ取り外します。上部にある、スタートボタンや連射スイッチ等は、とりあえずそのままにしておきます。



レバーの加工

三和電子のレバー JLF-TP-8Y のレバーガイドを取り外し、加工します。レバーガイドの上下左右の位置にある窪みを貫通するように、ドリルで 3mm の穴を開けます。ただしこれだと少しきついので、できれば 3.2mm のほうが良いと思います。窪みの反対側にある突起は、予めニッパーで切断しておきます。



穴の位置は、レバーを上から見たとき、上下左右の位置に予め空いている穴と一致するのが、正しい位置です。

レバーハーネスの加工

ハーネスのピン番号に対応するケーブルの色は、以下のようになっていました。ハーネスによっては、色が異なるかもしれません。

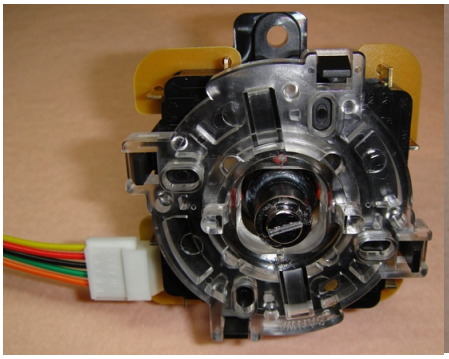
1	2	3	4	5
橙	緑	黒	赤	黄

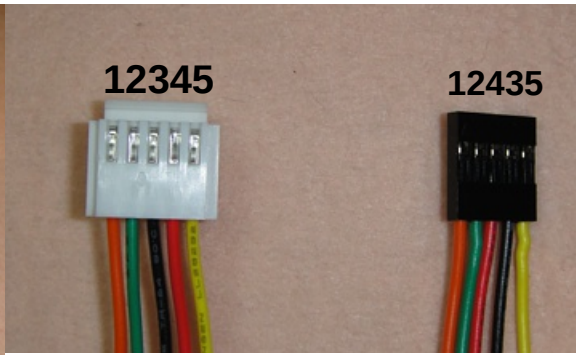
レバーを下から見たとき、コネクタが左側に位置するようレバーを取り付けるものと

すると、レバーの方向は、以下のようになっていました。

1	2	3	4	5
橙	緑	黒	赤	黄
G	左	右	下	上

ローリングスイッチシールドのピン配置に合わせて、3 番と 4 番のケーブルを入れ替え、QI コネクタで端末処理を行います。



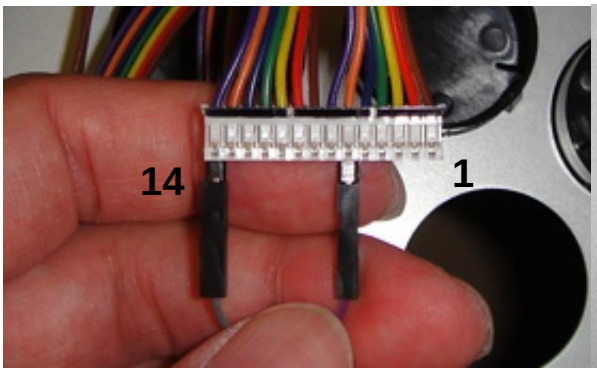


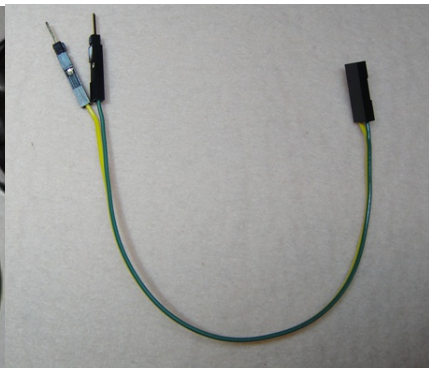
スタートボタン用ハーネスの製作

スタートボタンは、ジョイスティック本体に始めから付いているボタンを利用することにします。ジョイスティック本体の上部にある基板から伸びているケーブルのピン番号と色は、以下のようになっていました。スタートボタンの信号線は 6 (橙)でした。

1	2	3	4	5	6	7	8	9	10	11	12	13	14(GND)
茶	赤	黄	緑	青	橙	紫	茶	赤	黄	緑	青	橙	紫

6 ピンと 14 ピンを、ローリングスイッチシールドに接続するためのハーネスを製作します。片側を 2 ピン×1、反対側を 1 ピン×2 とし、QI コネクタで端末処理を行います。ケーブルは AWG26 の太さが手ごろでした。両方長いピンヘッダを利用し、ケーブルのコネクタに接続します。



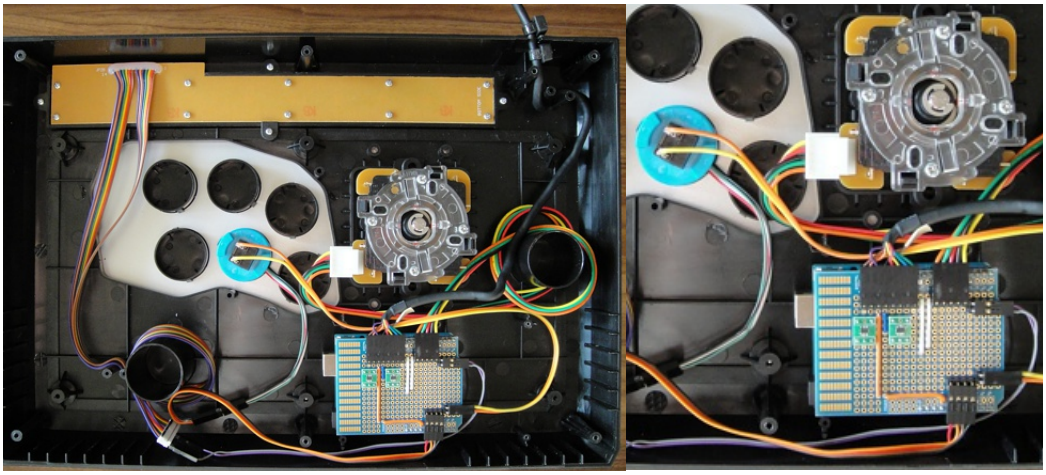


部品の組み込み

ここまでに製作した部品を、ジョイスティック本体に組み込みます。

- DUALSHOCK のケーブル
- ローリングスイッチユニット
- Arduino＋ローリングスイッチシールド
- 三和電子のレバー JLF-TP-8Y
- レバーハーネス
- スタートボタン用ハーネス

レバーの固定に利用するネジは、文献[1]では、なべタッピングネジ 3×30(太さ×長さ)となっていますが、ねじ山が少し足りないように思えたので、3×35 の先端 5mm を電エペンチで切断し、利用しました。余ったボタンの穴は目隠しキャップで塞ぎます。



PlayStation 2 に接続する際は、結線に間違いがないか十分に確認して下さい。結線を間違えると、PlayStation 2 が壊れてしまう可能性があります。
最初は、DUALSHOCK を USB に変換するコンバータを利用し、PC 上で確認することをおすすめします。

ソースコード

```
// Rolling switch for PS2
//
// Copyright (c) 2015 Kazumasa ISE
// Released under the MIT license
// http://opensource.org/licenses/mit-license.php

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

//#define DEBUG
//#define ROTATION_TEST
```

```
#define ACK_WAIT 0.5
#define ACK 9
#define SET_ACK_LOW  (PORTB &= ~B00000010)
#define SET_ACK_HIGH (PORTB |=  B00000010)
#define SW_LEFT      7
#define SW_DOWN      6
#define SW_RIGHT     5
#define SW_UP        4
#define SW_START     3
#define SW_SQUARE   19
#define SW_CROSS    18
#define SW_CIRCLE   17
#define SW_TRIANGLE 16
#define SW1 (PIND | B00000111)
#define SW2 ((PINC << 2) | B00001111)
#define RENC_A 15
#define RENC_B 14
#define RENC_AB (PINC & B00000011)
#define READ_DATA            0x42
#define CONFIG_MODE          0x43
#define SET_MODE_AND_LOCK    0x44
#define QUERY_MODEL_AND_MODE 0x45
#define UNKNOWN_COMMAND_46   0x46
#define UNKNOWN_COMMAND_47   0x47
#define UNKNOWN_COMMAND_4C   0x4C
#define VIBRATION_ENABLE     0x4D
#define CMD_BYTES 9
#define PULSE_NUM (50*4)

volatile bool isAnalogMode = false;
volatile bool isConfigMode = false;
volatile bool unknownFlag = false;
volatile byte RH = 0x80;
volatile byte RV = 0x80;
volatile byte LH = 0x80;
volatile byte LV = 0x80;

byte analogStickTableH[PULSE_NUM];
byte analogStickTableV[PULSE_NUM];

inline float degToRad(float deg) {
 return 2.0 * M_PI * deg / 360.0;
}

inline float sign(float x) {
 return (x < 0.0) ? -1.0 : ((x > 0.0) ? 1.0 : 0.0);
}

inline float lerp(float x, float y, float s) {
 return x + s * (y - x);
}

inline byte analogValue(float data) {
 const float temp = (data + 1.0) / 2.0; // -1~1 -> 0~1
 return (byte)(255 * temp + 0.5); // 0~1 -> 0~255
}
```

```
inline void initAnalogStickTable() {
 for (int i = 0; i < PULSE_NUM; i++) {
  const float deg = i * 360.0 / PULSE_NUM;
  const float rad = degToRad(deg);
  const float x = cos(rad);
  const float y = sin(rad);
  const float x2 = sign(x);
  const float y2 = sign(y);
  const float param = 0.3; // 0~
  analogStickTableH[i] = analogValue( lerp(x, x2, param));
  analogStickTableV[i] = analogValue(-lerp(y, y2, param));
 }
}

inline int rotationSignum(byte curr) {
 static byte prev = 0;
 int signum = 0;
 if (prev != curr) {
  const bool cw = ((prev << 1) ^ curr) & B00000010;
  signum = cw ? -1 : 1;
  prev = curr;
 }
 return signum;
}

inline int rotationPulseCount(int signum) {
 static int count = 0;
 count += signum;
 count = (count + PULSE_NUM) % PULSE_NUM;
 return count;
}

void setup() {
 // SPI setup
 SPI.setBitOrder(LSBFIRST);
 SPI.setDataMode(SPI_MODE3);
 SPCR &= ~(_BV(MSTR)); // Set as Slave
 SPCR |= _BV(SPE); // Enable SPI
 pinMode(SS, INPUT);
 pinMode(MOSI, INPUT);
 pinMode(MISO, OUTPUT);
 digitalWrite(MISO, HIGH);
 pinMode(SCK, INPUT);
 SPI.attachInterrupt();
 // ACK pin setup
 pinMode(ACK, OUTPUT);
```

```
 digitalWrite(ACK, HIGH);
 // Switches setup
 pinMode(SW_LEFT, INPUT_PULLUP);
 pinMode(SW_DOWN, INPUT_PULLUP);
 pinMode(SW_RIGHT, INPUT_PULLUP);
 pinMode(SW_UP, INPUT_PULLUP);
 pinMode(SW_START, INPUT_PULLUP);
 pinMode(SW_SQUARE, INPUT_PULLUP);
 pinMode(SW_CROSS, INPUT_PULLUP);
 pinMode(SW_CIRCLE, INPUT_PULLUP);
 pinMode(SW_TRIANGLE, INPUT_PULLUP);
 // Rotary encoder setup
 pinMode(RENC_A, INPUT);
 pinMode(RENC_B, INPUT);
 initAnalogStickTable();
#ifdef DEBUG
 Serial.begin(115200);
#endif
}

inline byte readDataResponse(byte i) {
 const byte DAT[] = {0xFF, 0x41, 0x5A, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF};
 switch (i) {
 case 1: return isConfigMode ? 0xF3 : (isAnalogMode ? 0x73 : 0x41);
 case 3: return SW1;
 case 4: return SW2;
 case 5: return RH;
 case 6: return RV;
 case 7: return LH;
 case 8: return LV;
 default: return DAT[i];
 }
}

inline byte setModeAndLockResponse(byte i) {
 const byte DAT[] = {0xFF, 0xF3, 0x5A, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00};
 return DAT[i];
}

inline byte queryModelAndModeResponse(byte i) {
 const byte DAT[] = {0xFF, 0xF3, 0x5A, 0x01, 0x02, 0x00, 0x02, 0x01,
0x00};
 switch (i) {
 case 5: return isAnalogMode ? 0x01 : 0x00;
 default: return DAT[i];
```

```
 }
}

inline byte unknownCommand46Response(byte i) {
 const byte DAT[] = {0xFF, 0xF3, 0x5A, 0x00, 0x00, 0x01, 0x02, 0x00,
0x0A};
 switch (i) {
 case 6: return unknownFlag ? 0x01 : 0x02;
 case 7: return unknownFlag ? 0x01 : 0x00;
 case 8: return unknownFlag ? 0x14 : 0x0A;
 default: return DAT[i];
 }
}

inline byte unknownCommand47Response(byte i) {
 const byte DAT[] = {0xFF, 0xF3, 0x5A, 0x00, 0x00, 0x02, 0x00, 0x01,
0x00};
 return DAT[i];
}

inline byte unknownCommand4CResponse(byte i) {
 const byte DAT[] = {0xFF, 0xF3, 0x5A, 0x00, 0x00, 0x00, 0x04, 0x00,
0x00};
 switch (i) {
 case 6: return unknownFlag ? 0x07 : 0x04;
 default: return DAT[i];
 }
}

inline byte vibrationEnableResponse(byte i) {
 const byte DAT[] = {0xFF, 0xF3, 0x5A, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF};
 return DAT[i];
}

inline byte dat(const byte CMD[], byte i) {
 switch (CMD[1]) {
 case READ_DATA:
  return readDataResponse(i);
 case CONFIG_MODE:
  isConfigMode = (CMD[3] == 0x01);
  return readDataResponse(i);
 case SET_MODE_AND_LOCK:
  isAnalogMode = (CMD[3] == 0x01);
  return setModeAndLockResponse(i);
 case QUERY_MODEL_AND_MODE:
  return queryModelAndModeResponse(i);
 case UNKNOWN_COMMAND_46:
  unknownFlag = (CMD[3] == 0x01);
```

```
  return unknownCommand46Response(i);
 case UNKNOWN_COMMAND_47:
  return unknownCommand47Response(i);
 case UNKNOWN_COMMAND_4C:
  unknownFlag = (CMD[3] == 0x01);
  return unknownCommand4CResponse(i);
 case VIBRATION_ENABLE:
  return vibrationEnableResponse(i);
 default:
  return readDataResponse(i);
 }
}

inline void acknowledge() {
 SET_ACK_LOW;
 _delay_us(ACK_WAIT);
 SET_ACK_HIGH;
}

#ifdef DEBUG
#define LINE_FEED 0xAA
#define MAX_LOG_SIZE 300
volatile byte cmdLog[MAX_LOG_SIZE] = {0};
volatile byte datLog[MAX_LOG_SIZE] = {0};
volatile int logCount = 0;
#endif

ISR(SPI_STC_vect) {
 static byte ID = 0x41;
 static byte CMD[CMD_BYTES] = {0};
 static byte cmdCount = 0;
 bool continueCom = false;
 CMD[cmdCount] = SPDR;
#ifdef DEBUG
 if (logCount < MAX_LOG_SIZE) {cmdLog[logCount++] = CMD[cmdCount];}
#endif
 const byte numOfCmd = 3+2*(ID & 0x0F);
 // Check CMD
 if (cmdCount == 0) {
  if (CMD[cmdCount] == 0x01) {
   continueCom = true;
  }
 } else if (cmdCount == 1) {
  if (CMD[cmdCount] == 0x01) {
   cmdCount = 0; // Reset count
   continueCom = true;
  } else if ((CMD[cmdCount] & 0x40) == 0x40) {
   continueCom = true;
```

```
  }
 } else if (cmdCount == 5) {
  if ((CMD[1] == READ_DATA) && (CMD[cmdCount] == 0x01)) {
   cmdCount = 0; // Reset count
  }
  continueCom = true;
 } else if (cmdCount < numOfCmd-1) {
  continueCom = true;
 }
#ifdef DEBUG
 if (!continueCom && (logCount < MAX_LOG_SIZE)) {cmdLog[logCount++] =
LINE_FEED;}
#endif
 // Set next DAT
 cmdCount = continueCom ? cmdCount+1 : 0;
 const byte DAT = dat(CMD, cmdCount);
 if (cmdCount == 1) {ID = DAT;}
 SPDR = DAT;
#ifdef DEBUG
 if (logCount < MAX_LOG_SIZE) {datLog[logCount] = DAT;}
#endif
 if (continueCom) {acknowledge();}
}

void loop() {
 int signum = rotationSignum(RENC_AB);
#ifdef ROTATION_TEST
 signum = 1;
 delay(20);
#endif
 const int count = rotationPulseCount(signum);
 RH = analogStickTableH[count];
 RV = analogStickTableV[count];
#ifdef DEBUG
 if (logCount == MAX_LOG_SIZE) {
  logCount = 0;
  int lfPos = -1;
  for (int i=0; i<MAX_LOG_SIZE; i++) {
   if (cmdLog[i] == LINE_FEED) {
    Serial.print("DAT: ");
    for (int j=lfPos+1; j<i; j++) {
     Serial.print(datLog[j], HEX);
     Serial.print(" ");
    }
    lfPos=i;
    Serial.println();
    Serial.print("CMD: ");
```

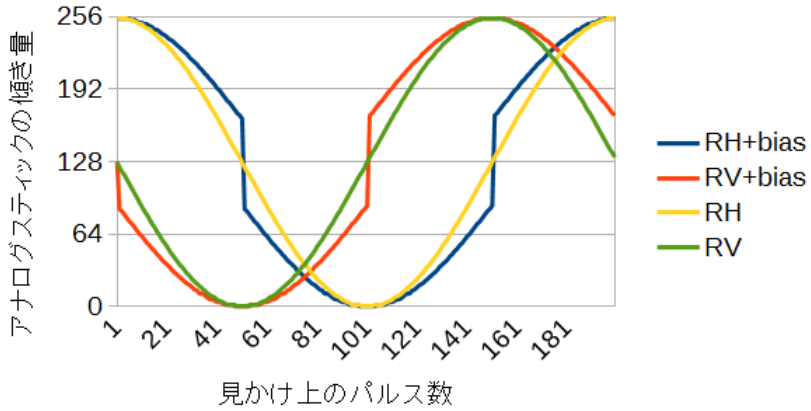
```
    } else {
      Serial.print(cmdLog[i], HEX);
      Serial.print(" ");
    }
   }
  }
#endif
}
```

アナログモードに対応したDUALSHOCKのエミュレートプログラムに、ロータリエンコーダの信号処理プログラムを組み合わせます。
変更を加えた部分について、説明します。

sign()
符号関数です。

lerp()
線形補間を行います。

initAnalogStickTable()
アナログスティックの傾き量テーブルを初期化します。
アナログスティックのｘ軸とｙ軸の中央付近では、傾きを認識しない範囲があり、サテライトが滑らかに回転せず、引っかかったような動きとなってしまいました。
この範囲の値を使用しないよう、以下のグラフに示すように、符号関数によるバイアスを加算します。



コメントアウトされている以下の行を有効にすると、サテライトが自動で回転し、滑らかに回転するかどうか、テストすることができます。

```
//#define ROTATION_TEST
```

加算パラメータの値は0.3が適当でした。

部品

HORI ファイティングスティック V3 ￥5658

千石電商

三和電子 JLF-TP-8Y ジョイスティック ￥1838
三和電子 TP 専用ハーネス JLF-H ￥467
三和電子 φ30 ゲーム SW（はめ込み） 青 OBSF-30-B（青） ￥173
三和電子 ボタン用目隠しキャップ φ30 OBSMφ30 ￥74×5
セイミツ クリアレバーボール 35φ ブルー LB-39 ブルー ￥210
日本圧着端子製造(JST) ファストン端子#110（メス） LTO-41T-110N (10 個) ￥110
【QI コネクタ】信号伝達コネクタ用ピン メス【F】(10 本) ￥74×4
【QI コネクタ】信号伝達コネクタ（黒）1×8 2550-1×8 ￥21
【QI コネクタ】信号伝達コネクタ（黒）1×5 2550-1×5 ￥21
【QI コネクタ】信号伝達コネクタ（黒）1×4 2550-1×4 ￥21×3
【QI コネクタ】信号伝達コネクタ（黒）1×2 2550-1×2 ￥21×3
【QI コネクタ】信号伝達コネクタ（黒）1×1 2550-1×1 ￥21×2
ピンヘッダ 1 列×40P (ストレート・標準ピッチ) 2544-1×40 15.1(6.3/6.3) ￥105
2545-1×40 ピンヘッダ 1 列×40P (L 型・標準ピッチ) ￥84
協和ハーモネット RKV 0.3×4 列 L-1 4 芯リボンケーブル 1m ￥250
協和ハーモネット RKV 10/0.12×10 列 L-1 10 芯リボンケーブル 1m ￥280
東芝 CMOS ロジック IC TC74HC125AP ￥84 ※2

秋月電子通商

Arduino Uno Rev3 ￥2940
Arduino 用ユニバーサル プロトシールド基板 ￥200
ピンヘッダ 1×40 (40P) ￥40
1 回路 3 ステートバッファ TC7SZ125F(TE85LF)（10 個入） ￥100 ※1
SOT23 変換基板 金フラッシュ （10 枚入） ￥150 ※1
細ピンヘッダ 1×40 （黒） ￥40 ※1
耐熱通信機器用ビニル電線 2m×10 色 外径 0.65mm ￥620
スズメッキ線(0.6mm 10m) ￥210
鉛フリーハンダ 0.8mm ￥280

ビットトレードワン

日本電産コパル電子 ロータリエンコーダ[RES20D-50-201-1] ￥1250

ジョイフル本田

八幡ねじ 押しバネ 0.6×9×26 (線径×外径×長さ) [入数 2] ￥185
八幡ねじ なべタッピング 3×35 (太さ×長さ) (12 本) ￥103
デリッシュ 鍋ツマミ LL ￥204

ハードオフ

DUALSHOCK コネクタケーブル（ジャンク） ￥300
※2 は※1 の代替部品。

参考文献

[1] ファイティングスティック V3(FSV3)を改造する（基本編） 雑記ビルディング（仮）
http://kazyangs.blog114.fc2.com/blog-entry-1.html

ソフトウェア及び工具

ソフトウェア

Arduino IDE: Arduino - Software
http://arduino.cc/en/Main/Software
文書作成: ホーム | LibreOffice - オフィススイートのルネサンス
https://ja.libreoffice.org/
図形描画: Draw Freely. | Inkscape
http://www.inkscape.org/ja/
ブレッドボード配線図: Welcome - Fritzing
http://fritzing.org/
回路図: 水魚堂の回路図エディタ
http://www.suigyodo.com/online/schsoft.htm
部品配置図: Ｐａｓｓ
http://www.geocities.jp/uaubn/pass/

工具

太陽電機産業 goot ニクロムはんだこて KS-30R(30W)
太陽電機産業 goot こて先クリーナー ST-30
白光(HAKKO) 簡易はんだ吸取器 ハッコースッポン 18G
エンジニア 精密圧着ペンチ PA-21
VESSEL ワイヤーストリッパー No.3500E-2
トップ工業 ラジオペンチ RA3-150
太陽電機産業 goot 精密ニッパー フラットカット YN-10
タミヤ 精密ニッパー
RYOBI 充電式ドライバドリル BD-72KT
KUROTO 鉄工ドリルセット 13 本組 DRR003
NACHI 鉄工用ドリル 7.2mm
NACHI 鉄工用ドリル 8.0mm
NACHI 鉄工用ドリル 9.0mm
アネックス(ANEX) フォーラインドライバー No.8400 + 2×100
アネックス(ANEX) フォーラインドライバー No.8400 + 1×75
アネックス(ANEX) フォーラインドライバー No.8300 + 0×75
アネックス(ANEX) フォーラインドライバー No.8400 - 5.5×75
アネックス(ANEX) フォーラインドライバー No.8200 - 2.5×75
ロブテックス 電装圧着工具（電工ペンチ） FK1
内外 クラフト保護メガネ （SG2610）
百均ニッパー
軍手

謝辞

今回のローリングスイッチは、多くの先駆者の方々の努力の成果を元に製作させていただいております。私個人の力だけでは、完成に至ることはできなかったと思います。大変感謝しております。
私に Arduino の存在を教えてくれた@overhilowsee さん、ありがとうございます。
船田巧さん(@sentoki)の資料「Arduino 日本語リファレンス」
http://www.musashinodenpa.com/arduino/ref/
には、大変お世話になりました。ありがとうございます。
藤田さんの資料「プレイステーション・ＰＡＤ／メモリ・インターフェースの解析」
http://kaele.com/~kashima/games/ps_jpn.txt
と、寺川愛印さん(@elfmimi)の資料「デュアルショック(SCPH-1200)の解析」
https://applause.elfmimi.jp/dualshock.txt
は、今回のローリングスイッチを製作する上で非常に重要な資料でした。この資料がなければ、製作は不可能だったと思います。ありがとうございます。
ヤマザキさん(@mahaman_101)と白楽さん(@hak__rak)の資料
「新方式ローリングスイッチ製作法」
http://extendead.web.fc2.com/iroiro/R_SW/R_SW.html
は非常に画期的な方式で、結局これを超える方法は思いつくことが出来ませんでした。この資料がなければ、製作は不可能だったと思います。ありがとうございます。
他にも沢山の方々が作成した Web 上の資料を参考にさせていただきました。ありがとうございます。