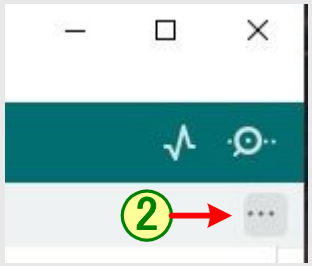
Arduino IDEにて　スケッチに ソースファイルを追加する方法

Arduino IDEにて　最初に新規スケッチを　作成する時は、他のアプリ同様に　左上の ファイルメニューを　クリックします。
① 参照　　これは一般的な操作なので　分かると思います。

次に　C++用で　追加で　新規ソースファイルを　作成する時は、右端に 点を横に3つ並べたアイコンが有ります。② 参照
これを、クリックします。

ポップアップメニューが　出てくるので　新しいタブ を　クリックします。　③ 参照

ファイル名入力の　ダイアログボックスが 表示されます。
新しいファイルの　ファイル名を入力します。　④ 参照
そして、OK　を　クリックすると　新しいファイルタブが出来ます。
⑤ 参照



既に、5本分の ファイルタブが出来てるスケッチに test_Lib.h を 追加したので
計 6本の ファイルタブが あります。 更に ファイルタブを 追加する場合は
② ～ ④ の操作を 繰り返して下さい。

他のマイコンの開発環境を 扱った事のある方であれば、想像が付くと 思いますが
Arduino IDE でいう スケッチというのは、他の開発環境でいう プロジェクト という事で
はないかと思います。 通常プロジェクトは 1本の実行ファイルを 作成するために 必要
な 複数のソースファイルを 管理する 器のようなものだと思います。

という事で、ソースファイルの 追加方法は理解できたと思います。

C++ 特有のコーディングについて

という事で、Arduino IDEにおいて 追加のソースファイルの作成方法は 理解出来たと、思いますが、これで テーブルの上に ごはん茶碗を 置いたところで、茶碗の中は 空なので この中にプログラムを 記述していく事になります。

で、C++にて ひとまとまりの I/O処理等のライブラリファイルを 作成するには、上記の例えでいうと 茶碗が 2つ必要です。

クラス等の型を宣言するヘッダーファイル（ 拡張子 .h ）と、クラスのメンバー関数を記述する 拡張子 .cppの ファイルの 2つです。

で、今回は 押しボタンスイッチのスキャニング処理の説明も 行う予定ですので、pb_sw.h と pb_sw.cpp を 使って C++特有の記述方法を、説明します。

尚、C++は Object指向言語ですが、Object指向の 説明は 抽象的で難しく かなり長くなるので、今回は しません。

C言語を知っている方を 前提に 話をして行きます。 クラスに関わる宣言は、さしあたり、一律このように宣言するものだ。 という事で 記述する形だけ憶えて下さい。

最近は、もう Object指向言語は 当たり前になってきて、Object指向の話も あまり聞かなくなっています。 じゃ、最近のプログラマは、全てObject指向の事を 理解しているのかというとそうではないと思います。

一部の方は しっかり理解されてる方もいると思いますが、大多数の方は あまり理解して無いと思います。 これは プログラミング言語の改良で 難しい事を理解しなくても やさしく使えるようになってきたからだと思います。

ヘッダーファイル .h に　関して

Object指向で　探したら C++言語は 型宣言を 厳密に行う言語と　書いてありました。
C++で　型宣言を行う場所は　ヘッダーファイル　.h 内です。 クラスの型宣言は　ヘッダーファイルにて行います。 その他、必要に応じて定数等の宣言も　行います。

で、ヘッダーファイルは　そのクラスのメンバー関数を　実装する .cppファイルの先頭で読み込まれます。
それと、そのクラスを使用するメインとなるソースファイル（ Arduino環境であれば　.ino ファイル ）の先頭でも #includeで　読み込まれます。 今回の例としては　先頭にて #include ″pb_sw.h″ という形で、宣言します。

それと、細かい事ですが　#include 右側のファイル名を　囲む　＜＞と　″ ″ ですが、少し意味が　異なります。

＜＞ は、言語標準の フォルダから　指定したファイルを　読み出します。 それに対し　″ ″ は スケッチのフォルダ内で　ファイルを　読み出します。 ファイルが存在しない場合は、コンパイル時　エラーが出ます。 よって、自分で　コーディングしたライブラリは　″ ″ を 使用します。
よって、今回の　押しボタンスキャンのライブラリは　#include ″pb_sw.h″ に　なります。

まずは、ヘッダーファイルのお約束事を 一つ紹介します。 ヘッダーファイルは　1回読み込んだ後に、また読み込むと　2回目は　エラーに　なります。2回読み込まないように注意すればいい。ともいえますが 大プロジェクトで、ソースファイルが　百本もあると、チェックが　結構手間という場合もあります。 よって、自動的に　2回目読み込まないようにする方法が　あります。次に　紹介します。

pb_sw.h

```
#ifndef pb_sw_h
#define pb_sw_h

ここに　本来のクラス宣言等を行う。

#endif
```

先頭に # のついた予約語は マクロと呼ばれます。 これは、C言語の時代から ありました。
先頭の #ifndef pb_sw_h は pb_sw_h という文字列が宣言されて無ければ 以下の行から #endif まで コンパイラにて読み込まれます。

#ifndef pb_sw_h の 次の行で #define pb_sw_h が、宣言されています。 #define は、文字列の宣言です。 この文字列は 一種の名前のようなものです。 文字列を スペースで区切って2つ文字列を設定する事も出来ます。

で、#define pb_sw_h を 宣言すると、その後 再度 pb_sw.h を 読み出しても pb_sw_h が 既にコンパイラ上で 宣言されているので、2回目の読み出しでは #ifndef pb_sw_h は 成立しないので、#ifndef 以下の コーディングは #endif までの間、コンパイラが 読み込みません。 この機能により、同じ名前の 多重宣言エラーを 回避しているという事です。
それと #define は スペースで区切って2つの文字列を設定できると書きましたが、これは、最初の文字列が ソース内に出てきたら 2個目の文字列に 置き換えられます。 例えば

#define PBSW_1 36 と 宣言すれば、PBSW_1の文字列を 36 の 文字列に置き換えてくれる。という事です。 この場合は I/Oポートの番号を PBSW_1 という意味の分かる文字列に しておく事が、出来るという事です。
では、pb_sw.h の ソースを 見てみましょう。

pb_sw.h　1/2

```
#ifndef pb_sw_h       // 多重宣言の回避
#define pb_sw_h

//   I/O Port 宣言
// ---------------------------------
#define PBSW_1  36     // 押しボタンSW 1
#define PBSW_2  39     // 押しボタンSW 2
#define PBSW_3  34     // 押しボタンSW 3
#define PBSW_4  35     // 押しボタンSW 4
#define PBSW_5  32     // 押しボタンSW 5
#define PB_CONT 375    // 長押し検出値
#define PB_CONT_IVL  25  // 繰り返しインターバル

//  押しボタンSW  処理クラス
// ------------------------------------
class Pb_Scan
{
  public:
```

先頭にある #ifndef pb_sw_h と次の行の #define pb_sw_h については前ページで説明したので省略します。

次に #defineで 押しボタン1 ～ 5 のI/Oポート番号を 意味の分かりやすい名前で 宣言しています。

PB_CONT 375 と PB_CONT_IVL 25 は単位時間 4msの 時間設定値です。

PB_CONT ボタンを 押し始めてから、1.5秒経過(0.004 * 375 = 1.5) で 長押し検出開始で 時刻設定のインクリメント、デクリメントを高速で ボタンを離すまで繰り返し行います。 PB_CONT_IVLは繰り返し周期です。 0.004 * 25 = 0.1秒です。 class Pb_Scan は クラスの 型宣言の 開始です。public: は これより下の宣言は、どこからでもアクセス可能な広域的宣言である事を 意味します。

pb_sw.h　2/2

```
    Pb_Scan();          // コンストラクタ
    ~Pb_Scan();         // デストラクタ

    void  init( void );           // ポート初期化
    char  pb_scan_proc( void );   // スキャニング メイン
    // １個の スイッチの状態を取り出しスイッチ状態の履歴を更新
    unsigned char  get_inp_sft( int ioa, unsigned char sft );
  private:
    unsigned char  sft_1;
    unsigned char  sft_2;
    unsigned char  sft_3;
    unsigned char  sft_4;
    unsigned char  sft_5;
    unsigned char  sws;
    unsigned char  sws2;
    unsigned char  dvc;
    short          swcn;
};

#endif
```

クラス名と同じ Pb_Scan();関数は コンストラクタ（ 初期化処理 ）です。頭に チルダを付けた ~Pb_Scan(); 関数はデストラクタ（ 廃棄処理 ）です。 このコンストラクタ、デストラクタは C++の 毎回のお約束と 思っていて下さい。

init()関数は 押しボタン関係の I/O初期化処理です。

pb_scan_proc 関数と、get_inp_sft 関数は ソース内の コメント通りの関数です。 細かい事は、また後で 説明します。

次に private: は このクラス内の関数しかアクセス出来ない変数を宣言しています。 変数だけでなく private: の 関数も 宣言出来ます。今回は 必要無かったので、private: 関数は 宣言して いません。

unsigned char の 変数を 8個と 2byte整数の変数を １個宣言しています。

pb_sw.cpp 1/4

```
#include <Arduino.h>
#include "pb_sw.h"

Pb_Scan::Pb_Scan()          // コンストラクタ
{
    init();         // I/O ポート初期化
}

Pb_Scan::~Pb_Scan()         // デストラクタ
{
}

void  Pb_Scan::init( void )     // ポート初期化
{
    pinMode( PBSW_1,  INPUT );
    pinMode( PBSW_2,  INPUT );
    pinMode( PBSW_3,  INPUT );
    pinMode( PBSW_4,  INPUT );
    pinMode( PBSW_5,  INPUT );
}
```

先頭で、#include <Arduino.h> を 行ってますが、これを入れないと 標準の入出力関数の pinMode 関数、digitalWrite 関数、digitalRead 関数が 使えません。I/O処理を行う時は 必須です。 #include "pb_sw.h" は、pb_sw.cpp用のヘッダファイルなので これも必須です。

コンストラクタですが、Pb_Scan::Pb_Scan() の左側の Pb_Scan:: は、クラス Pb_Scan 内のメンバー関数である事を 示していると思います。:: 右の Pb_Scan() は クラスと同じ名前で この同じ名前は 暗黙の了解で コンストラクタを意味します。 同様に ~Pb_Scan() は デストラクタを意味します。 コンストラクタは 主に初期化処理を行います。 ここでは、init()関数を 呼び出し 5つの I/Oポートを 入力に初期化しています。で、デストラクタですが、パソコンアプリであれば必要と思いますが、組み込み用途であれば、まず使わないと思います。

あと、クラスのコンストラクタは　いつ呼び出されるのだろうか。？　と　不思議に 思われるかもしれませんが、 クラスを 実行できるようにインスタンス (メモリ上に存在する実態)を生成していれば、システムのスタートアップ処理にて　自動的に呼び出されるようです。
で、インスタンスを生成するとは、
変数を　宣言するのと同じです。
seg7_pbsw.ino 先頭の方に

```
static  Seg_7 seg7;     // 7セグメントクラス
static  Pb_Scan pbs;   // 押しボタン スキャン処理
```

が、あります。 先頭の static は この場合、あっても無くても問題ないです。 static は、静的という意味もありますが、もう一つ、スコープ範囲が　生成したファイル内だけになります。別のソースファイルからアクセスする場合は、static 宣言を 取らなければなりません。

pb_sw.cpp　2/4

```
// スキャニング メイン
char  Pb_Scan::pb_scan_proc( void )
{
  unsigned char ssw, n, m;

    sft_1 = get_inp_sft( PBSW_1, sft_1 );
    sft_2 = get_inp_sft( PBSW_2, sft_2 );
    sft_3 = get_inp_sft( PBSW_3, sft_3 );
    sft_4 = get_inp_sft( PBSW_4, sft_4 );
    sft_5 = get_inp_sft( PBSW_5, sft_5 );
    sws = 0;  n = 0;
    if( sft_1 == 0x0F ) { sws |= 0x01;  n++; }
    if( sft_2 == 0x0F ) { sws |= 0x02;  n++; }
    if( sft_3 == 0x0F ) { sws |= 0x04;  n++; }
    if( sft_4 == 0x0F ) { sws |= 0x08;  n++; }
    if( sft_5 == 0x0F ) { sws |= 0x10;  n++; }
    if( sws == 0 )  { swcn = 0;  sws2 = 0;  dvc = 0; }
    if( sws2 == sws ) swcn++;
    else              swcn = 0;
    if( swcn > PB_CONT ) swcn = PB_CONT;
```

押しボタンスキャニングの メイン関数　pb_scan_proc() 関数です。
ちょっと、ややこしくて申し訳ないですが　左のソースを見てもらうと 行毎に 1　2　3　4　5 の番号の付いた変数がありますが、そのまま押しボタンの番号と対応しています。

get_inp_sft()関数が　1個の押しボタン信号を取り込むと同時に　最新の4サンプル分のデータを byteデータの下位4bit のビット並びで表現しています。ビットデータの更新は　ビットデータを　左シフトして、その後最下位 bit に 最新の押しボタンデータの ONを 1 として設定しています。上位 4bit は常時 ゼロにしています。
変数の sft_1 というのは シフトステータスの ボタン1という意味です。

pb_sw.cpp　2/4

```
sft_1 = get_inp_sft( PBSW_1, sft_1 );
      ～          ～          ～
sft_5 = get_inp_sft( PBSW_5, sft_5 );
sws = 0;  n = 0;
if( sft_1 == 0x0F ) { sws |= 0x01;  n++; }
if( sft_2 == 0x0F ) { sws |= 0x02;  n++; }
if( sft_3 == 0x0F ) { sws |= 0x04;  n++; }
if( sft_4 == 0x0F ) { sws |= 0x08;  n++; }
if( sft_5 == 0x0F ) { sws |= 0x10;  n++; }
if( sws == 0 )  { swcn = 0;  sws2 = 0;  dvc = 0; }
if( sws2 == sws ) swcn++;
else              swcn = 0;
if( swcn > PB_CONT ) swcn = PB_CONT; // Limit処理
```

0 0 0 0 0 0 0 0	0x00	押しボタンが　押されて無い状態
0 0 0 0 0 1 1 1	0x07	押しボタンが　押された瞬間を検出
0 0 0 0 1 1 1 1	0x0F	押しボタンが　押され続けている状態
0 0 0 0 1 0 0 0	0x08	押しボタンが　離された瞬間を検出

左にソースを表示して無いと、説明が し難いので　また表示しました。
押しボタンのシフトデータの 状態は 左下の図を　参照して下さい。

sft_1 ～ sft_5 の各変数の状態は、
0x00が ボタンが押されて無い状態。
0x07が ボタンが押された瞬間を検出。
0x0Fが ボタンが押されたままの状態。
0x08が ボタンが離された瞬間を検出。
左ソースの　sft_1 == 0x0F から　sft_5 == 0x0F の if 文は　sws に 押されている bit番号を 1にして n++ をしていますが、5個の if 文を 通り nが　1 なら 1つしかボタンが押されてない事を　意味します。　n が　2 以上なら押しボタンを多重押ししているという事です。
if 文の　sws2 == sws は　押されているボタンが、押した瞬間のボタンと同じである事を確認しています。

pb_sw.cpp　3/4

```
    ssw = 0;  m = 0;
    if( sft_1 == 0x07 ) { ssw |= 0x01;  m++; }
    if( sft_2 == 0x07 ) { ssw |= 0x02;  m++; }
    if( sft_3 == 0x07 ) { ssw |= 0x04;  m++; }
    if( sft_4 == 0x07 ) { ssw |= 0x08;  m++; }
    if( sft_5 == 0x07 ) { ssw |= 0x10;  m++; }
    if( ssw != 0 )  sws2 = ssw;
    if(( ssw != 0 ) and (n > 0) ) ssw = 0;
    if( m > 1 ) ssw = 0;
    if( (sws2 & 0x18) != 0 ) // Inc Dec ボタンの時のみ効くようにする
    {
      if( swcn == PB_CONT )  // ボタンを押して 1.5秒ほど経過
      {
        dvc++;
        if( dvc == PB_CONT_IVL )  // 連続 Inc or Decを出すインターバル
        {
          dvc = 0;
          ssw = sws2;
    } } }
    return  ssw;
}
```

上から5個の if 文は シフトステータスを 0x07 と、比較しているので 押しボタンが 押された瞬間を 検出しています。 ssw に どのボタンが押されたか bit 位置にて ssw変数に格納して、m変数を 利用して 一度に 複数 押しボタンが押されて無いか確認しています。

下半分の 処理は、上下押しボタンの Inc、Decの 長押しによる連続 Inc Dec 信号を 高速に出すための 判断処理です。
ちょっと、ややこしいので 興味のある方は ソースを ダウンロードして見て下さい。

pb_sw.cpp　4/4

```
// １個の スイッチの状態を取り出し スイッチ状態の履歴を更新する
unsigned char  Pb_Scan::get_inp_sft( int ioa, unsigned char sft )
{
  unsigned char sw;

  sft = sft << 1;            // 1bit 左シフトして 最下位bitを 開ける
  sw = digitalRead( ioa );  // 目的の PBスイッチの状態を取り出す
  if( sw == LOW )   sft = sft | 1;
  // PBスイッチ＝LOWであれば 最下位bitを 1 にする
  sft = sft & 0x0F;          // 下位 4bitのみ残す

  return  sft;
}
```

やっと終りに来ました。

get_inp_sft() 関数は　前のページで、多少説明したので、凡そやっている事は、分かると思いますが、引数で もらい受けた sft 変数を 1ビット左シフトして、最下位ビットを　開けます。 そして digitalRead()関数で、押しボタンスイッチの状態を　読み出します。

　押しボタンスイッチの状態が　LOW (接点が ONで 閉じている)で あれば、1 を　sft 変数の最下位ビットに　入れます。 因みに　左シフトした状態では、最下位ビットは　ゼロに なってます。
sft = sft & 0x0F;　// 下位 4bitのみ残す　は、上位4ビットを　ゼロにしています。
そして、 sft　の値を　関数値として　返します。

　視聴者の皆様が　I/O処理のプログラムを作成するとき、多少でも　役に立てば幸いです。