mbed ライブラリ

作成から公開まで

「トラ技_mbedライブラリの作成方法の勉強会」2014年11月7日向け資料

https://atnd.org/events/57766

version 1.1, 08-Nov.-2014

注意： この資料は，いかなる団体の公式な見解を述べたものではありません．
全くの個人的な意見を述べたに過ぎません (^^;

mbedのライブラリの作成と公開，そしてそれをコンポーネントとして登録するまでを解説します．

限られた時間なので，あまり細かくは説明できませんが，例を示しながら流れを，その後にいくつかの考慮したほうがよい点などを話します．

TEDD OKANO

自己紹介：簡単に

mbedのライブラリ

- 各ユーザが自由に公開
- mbed.org サイトで共有
- 簡単インポート→使用
- 部品箱の部品みたいなもの
- 『コピペで簡単』のカギ

mbedのライブラリは非常に便利

mbed.orgサイトでいろいろなユーザが公開

簡単にインポートして使用できる

手軽なソフトの部品箱

mbedの『コピペで簡単』と言われるキモの部分

```
#include "mbed.h"
#include "SB1602E.h"

SB1602E lcd( p28, p27 ); //  SDA, SCL

int main()
{
  lcd.printf( 0, "Hello world!\r" );
  lcd.printf( 1, "pi = %.6f\r", 3.14159265 );
}
```

たとえば

http://developer.mbed.org/components/SB1602E/

たとえばストロベリー・リナックスで販売されている液晶ライブラリの使用例
高いレベルでハードウェアが抽象化されていて，簡単に使える.
これはmbed-SDK(mbedライブラリとも呼ばれてました)が，ハードを高度に抽象化してるのと同じ
ライブラリにはサンプルコードも付いているので，すぐに試せる

ライブラリの公開ページにはハードの接続図の情報も.
高い抽象化のおかげでユーザは高次の実装部分に集中できる

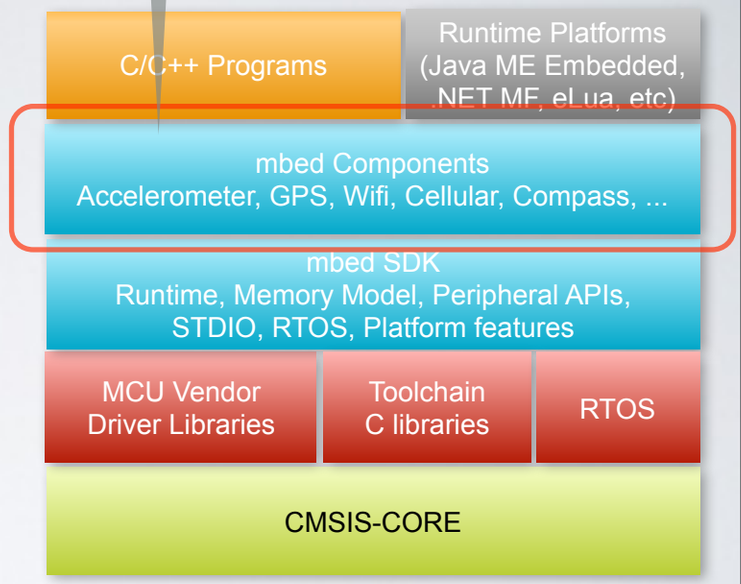
この例ではハードウェアを接続したピンを指定し,
最低限の関数コールで表示ができるようにしてある.
細かい実装部分はすべてクラスの中に隠蔽されている.

この「抽象化」のおかげで，ユーザは詳細を気にすることなく
高いレベルの実装に集中できる

ライブラリ

ハードウェアばかりではなく，
ネットワーク・プロトコルなども

- mbed-SDK (mbedライブラリ)
  ハードウェアの高度な抽象化
- コミュニティでサポートされる
  高機能ライブラリ群
- 必要があればレジスタにアクセス．最下層レベルでの最適化



http://developer.mbed.org/handbook/mbed-SDK

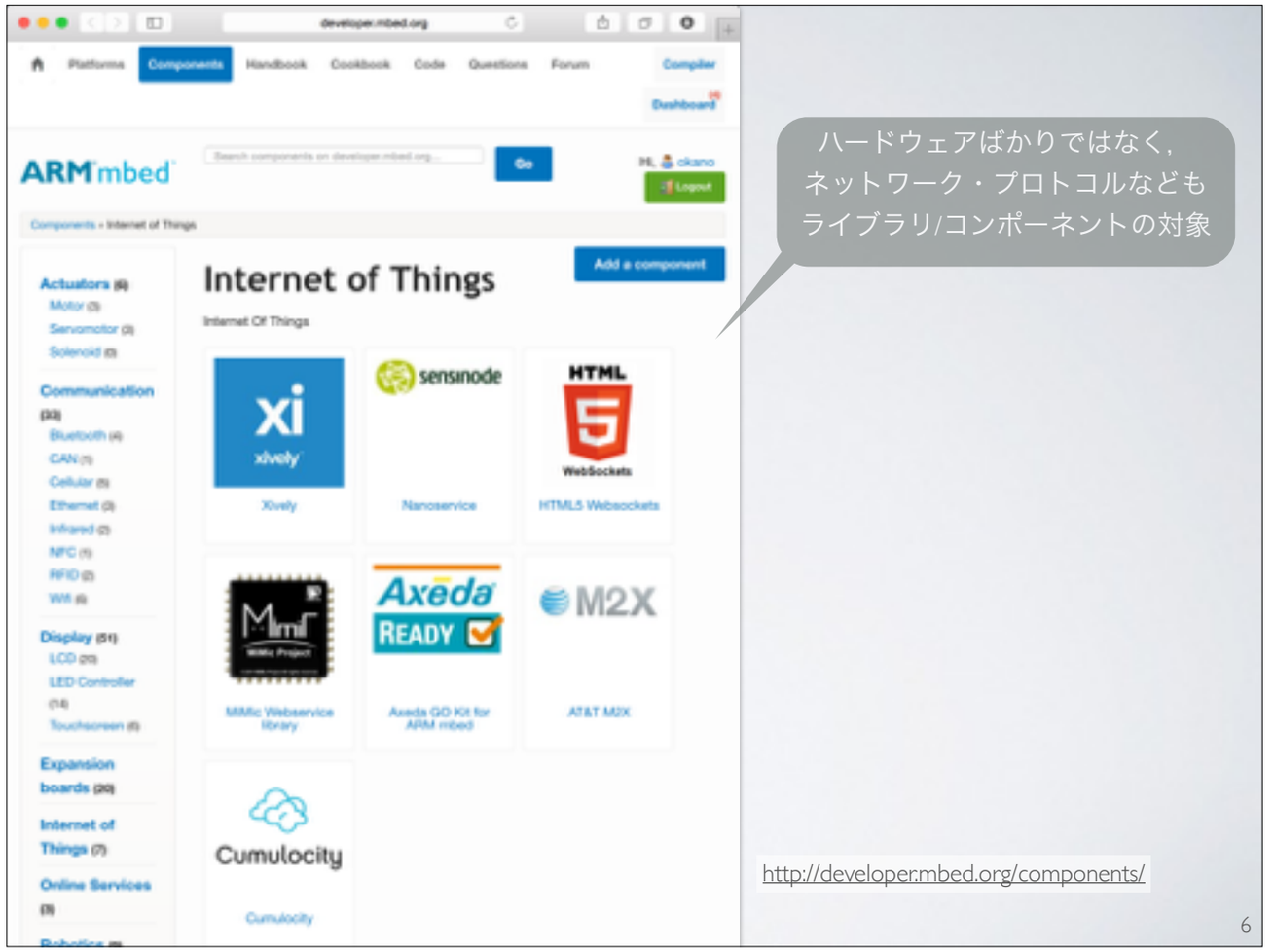
http://developer.mbed.org/components/

ライブラリは何層かに分けられたソフトウェアの一部

低いところはマイコンのペリフェラル(GPIOなど)や物理層に近いプロトコル(UARTやI2C..)などを実装

接続されたセンサ，表示機器，サーバへのアクセスは，上位のアプリケーションから使える部品(コンポーネント)として→「ライブラリ」

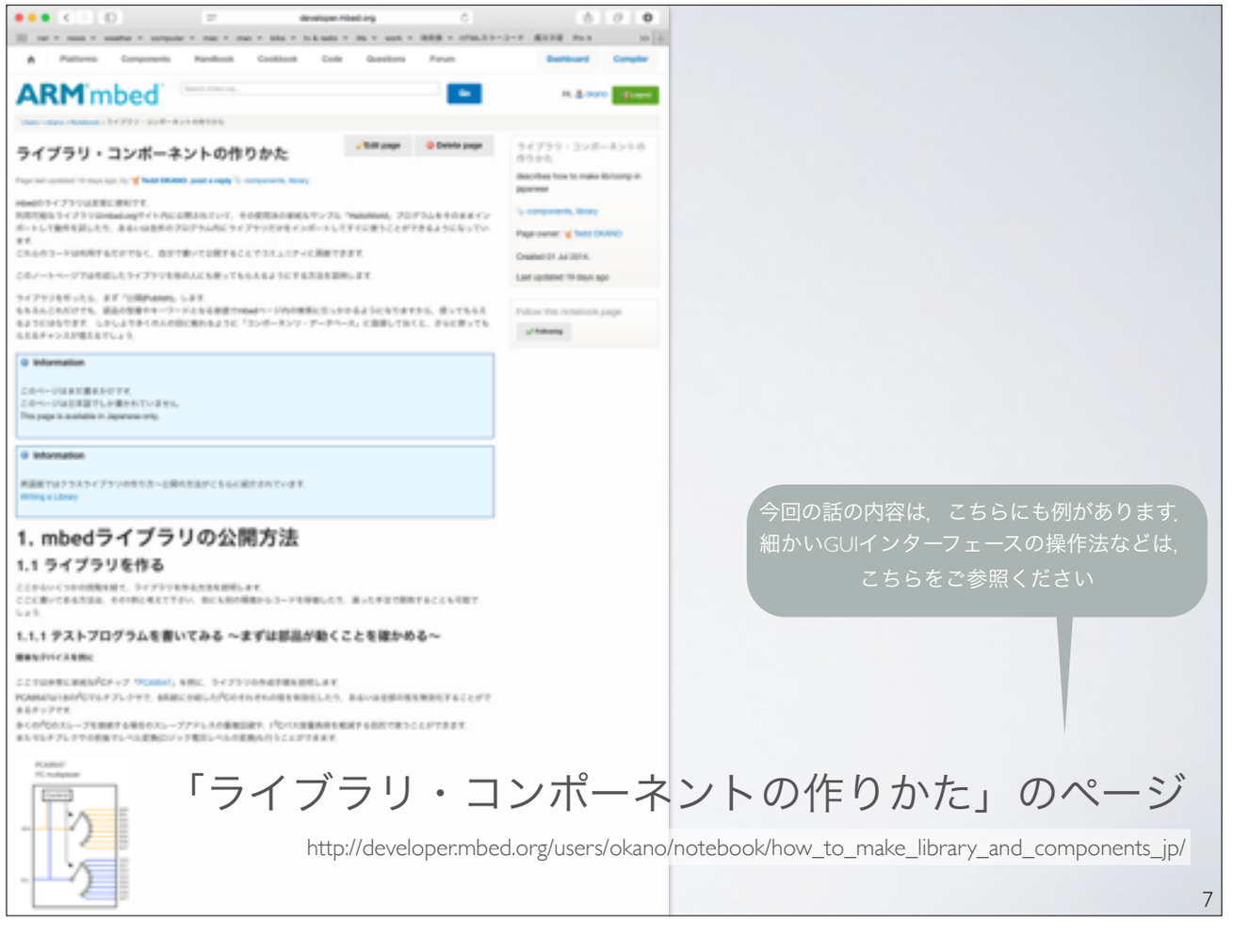
この部品であるライブラリを組み合わせてアプリケーションを作ることができる



ハードウェアばかりではなく，
ネットワーク・プロトコルなども
ライブラリ/コンポーネントの対象

http://developer.mbed.org/components/

コンポーネントは「ハードウェア部品のドライバ」という意味ではなく，「ソフトウェア部品」

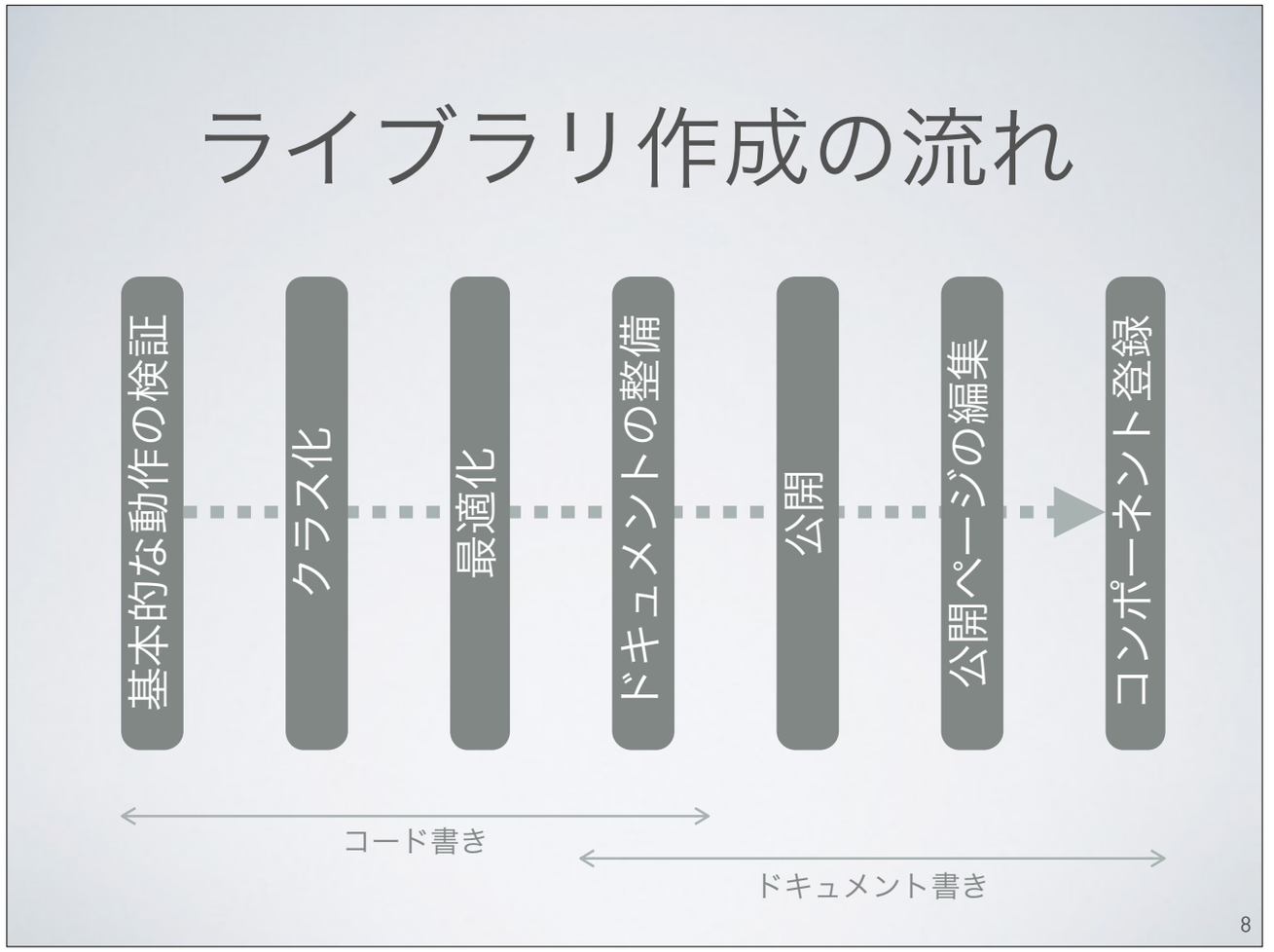


今回の話の内容は，こちらにも例があります．
細かいGUIインターフェースの操作法などは，
こちらをご参照ください

「ライブラリ・コンポーネントの作りかた」のページ

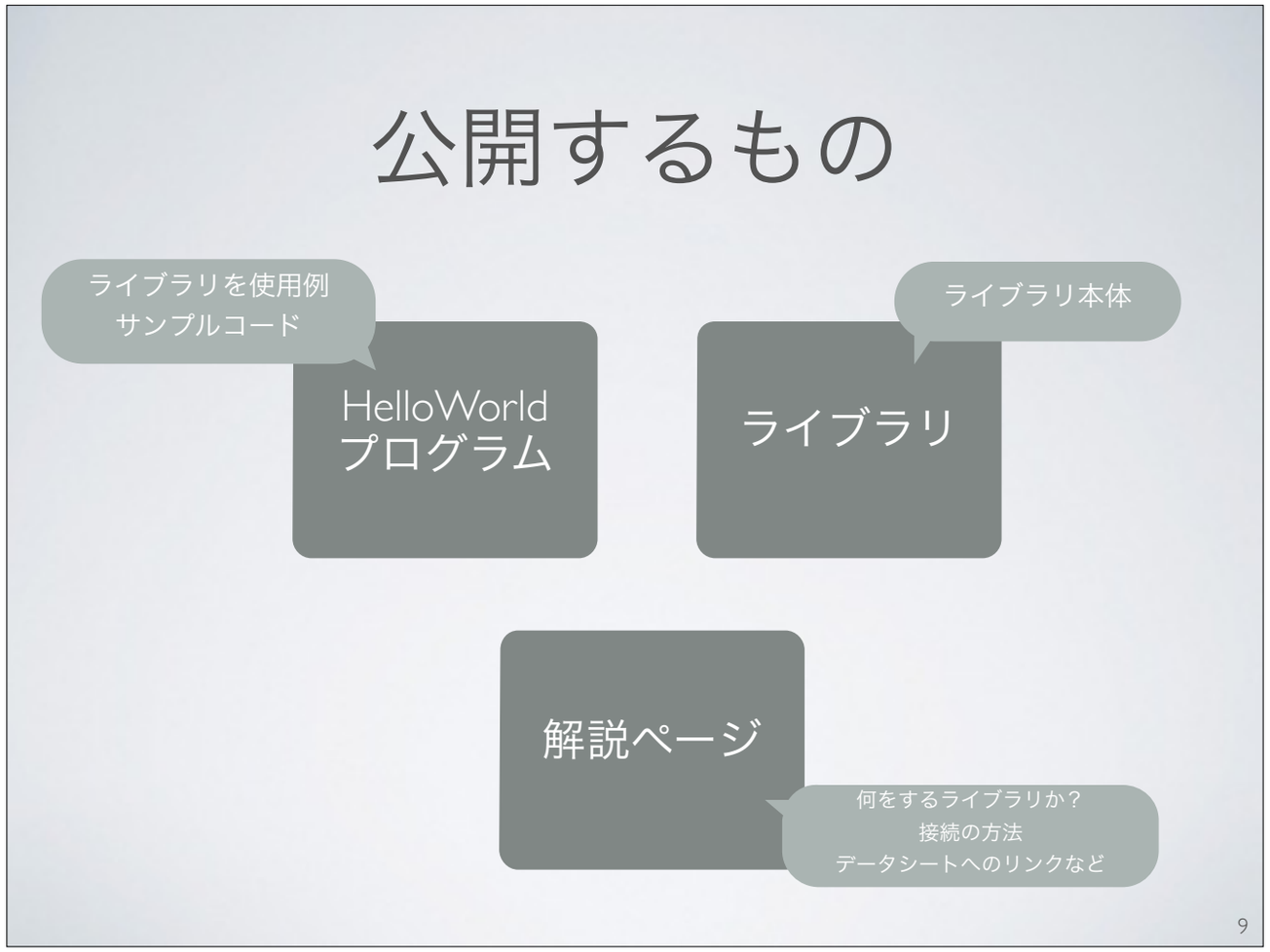
http://developer.mbed.org/users/okano/notebook/how_to_make_library_and_components_jp/

参考ページの紹介



だんだんカタい内容に入っていきます.

ライブラリを作って公開してコンポーネントとして登録するまで，敢えて私の例でその流れを書くとこうなります.
これは「こうしなければならない」という話ではなく，あくまで私の場合の例です



ライブラリを作ったら，「使ってもらいやすく」するために，いくつかのモノを用意しておくのが理想的です．
ライブラリ本体，ライブラリを直ぐに使ってみれるHelloWorldプログラム，解説のページなどです．

たいへんそう..？

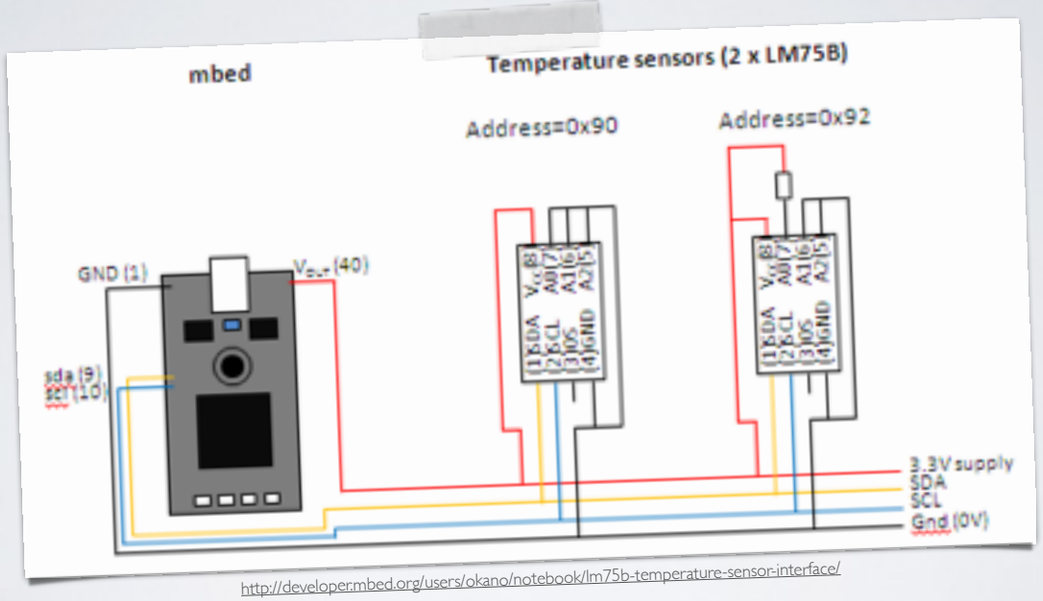
- 使ってもらいやすくするためには，いろいろ大変
- でも一度やってみれば，大したことは無い
- 必ず全部揃ってないと，いけないわけでもない
- できることまでやって，公開するのも方法
- そのあとでバージョンアップや，公開ページの記述追加も
- コードを共有して，他の人からの助けも

大変そうですが，一度やってみるとそうでも無いことがわかります.
最初から全て揃っていなくても大丈夫なので，まずは公開してみることも大事です (異論はあるとおもいます)

今回は簡単なハードウェアを例に説明します

今回は非常に単純なハードウェア用のライブラリを作る例で，具体的な流れを説明します

例で順を追って解説



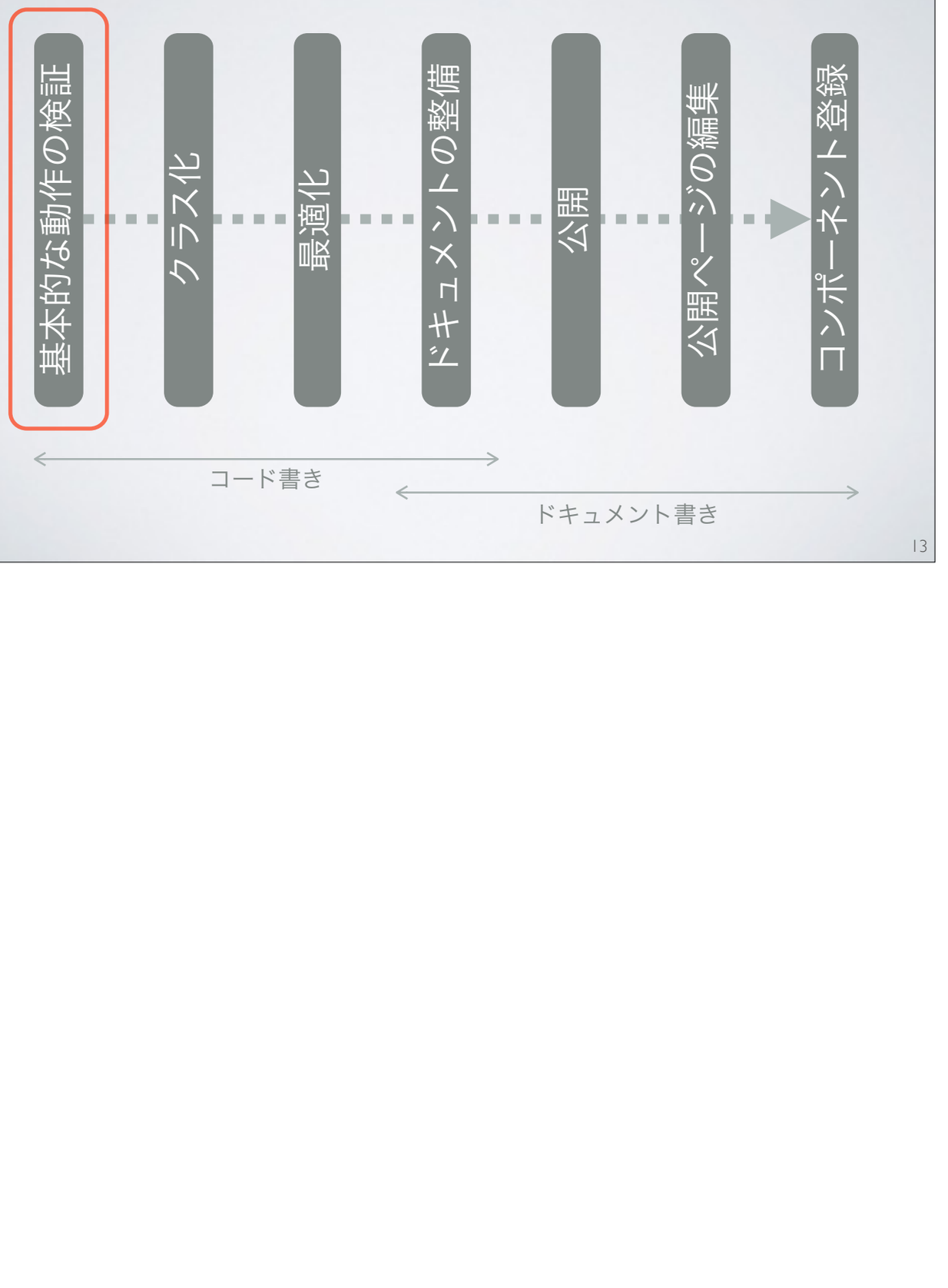
http://developer.mbed.org/users/okano/notebook/lm75b-temperature-sensor-interface/

I Cインターフェースの温度センサ「LM75B」を例に話をします.

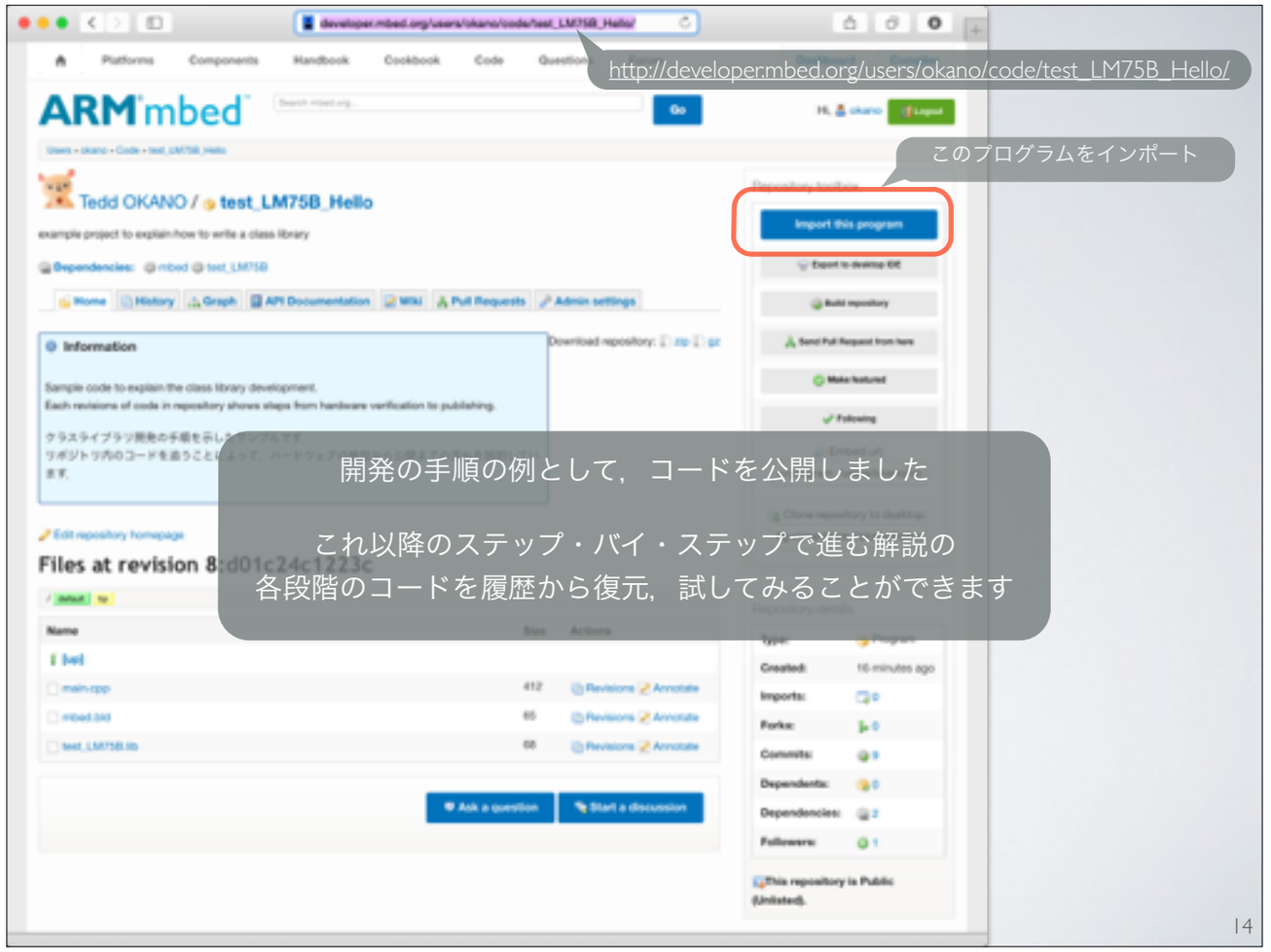
非常に単純なハードとソフトウェアで動かすことができるので，これを例に取り上げました.

LM75Bのフル機能のライブラリは，既にmbed.orgで公開されています.

ライブラリ作成の流れ

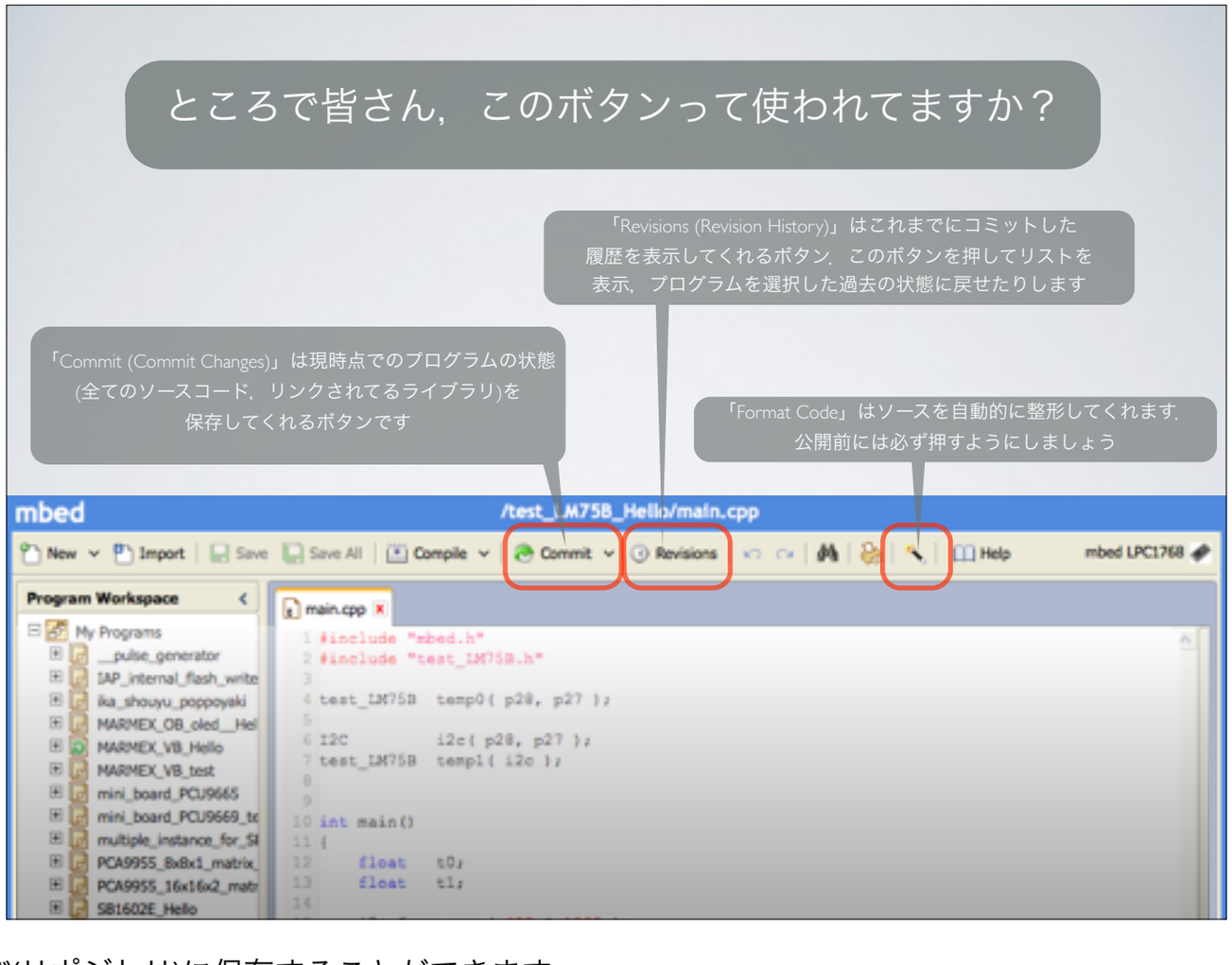


最初のステップ

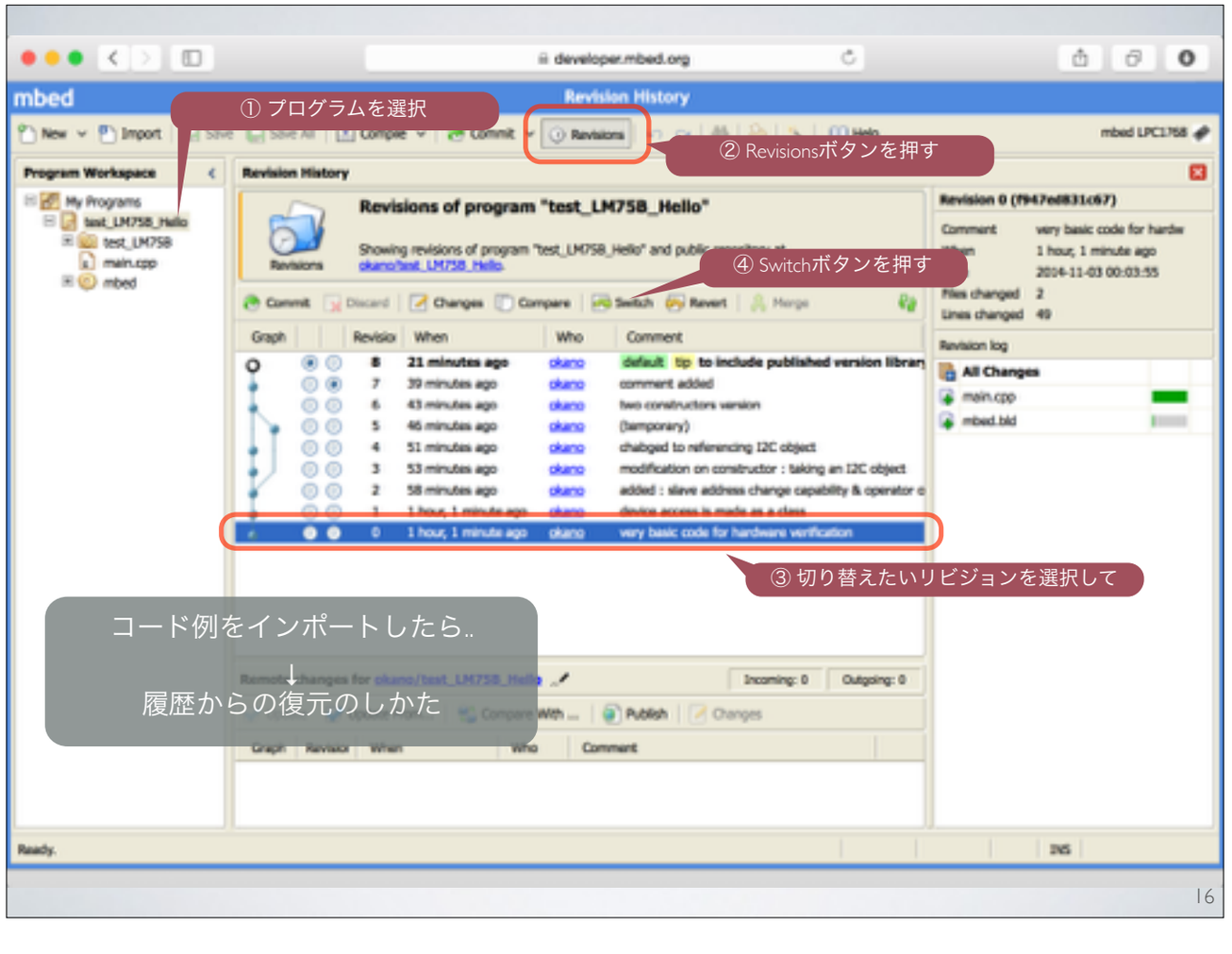


今回使用するサンプルコードはmbed.org内で公開されています.

是非，このコードをインポートして，履歴を辿ってみてください.



コードは履歴付きでmbed.org内のサーバ(リポジトリ)に保存することができます.



プログラムをインポートして履歴を辿るには「Rivision History」機能を使います.

(1) プログラムを選択して→ (2) 「Revisions」ボタンを押すと履歴の一覧が表示されます →(3) 切り替える先のリビジョンを選択肢ておいて → (4) 「Switch」ボタンを押すと見たいリビジョンのコードに置き換わります.

ハードウェアの動作を確認したコード

```
#include "mbed.h"

//  LM75B I2C slave address
#define     ADDRESS_LM75B   0x90

//  LM75B registers
#define     LM75B_Conf      0x01
#define     LM75B_Temp      0x00
#define     LM75B_Tos       0x03
#define     LM75B_Thyst     0x02

I2C     i2c( p28, p27 );

void    init( void );
float   read_temp( void );


int main()
{
    init();

    while(1) {
        printf( "temp = %7.3f\r\n", read_temp() );
        wait( 1 );
    }
}

void init( void )
{
    char    command[ 2 ];

    command[ 0 ]    = LM75B_Conf;
    command[ 1 ]    = 0x00;

    i2c.write( ADDRESS_LM75B, command, 2 );
}

float read_temp( void )
{
    char    command[ 2 ];

    command[ 0 ]    = LM75B_Temp;

    i2c.write( ADDRESS_LM75B, command, 1 );  // Send command string
    i2c.read(  ADDRESS_LM75B, command, 2 );  // read two bytes data

    return ( (float)( (command[ 0 ] << 8)| command[1] ) / 256.0 );
}
```

初期化

read_temp()を呼ぶと温度の値(摂氏)が返されるので，それを標準出力へ

スレーブアドレス0x90のデバイスに初期化レジスタ0x01に0x00を書く

スレーブアドレス0x90のデバイスのレジスタ0x00から2バイト読み出す

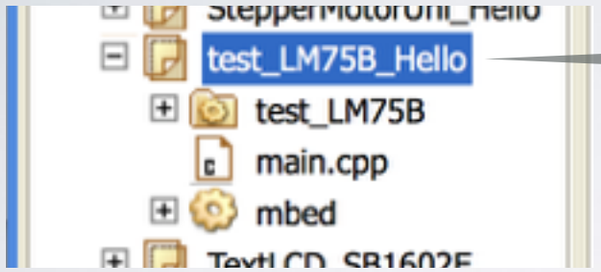
読み出したデータを摂氏に直して返す

一番古い履歴に残ってるコードは，まず最初に行ったハードが動くかどうかチェックするもの.

mbed-SDKを使った，ベタのCっぽい書き方のコードです．I^2Cのアクセスにはmbed-SDKのAPIが使われています.

プログラム名は「◯◯◯_Hello」(◯◯◯はライブラリ名．ライブラリはデバイスの型番など)のようにしておくのが，わかりやすいと思います

プログラム名は「○○○_Hello」(○○○はライブラリ名．ライブラリはデバイスの型番など)
のようにしておくのが，わかりやすいと思います

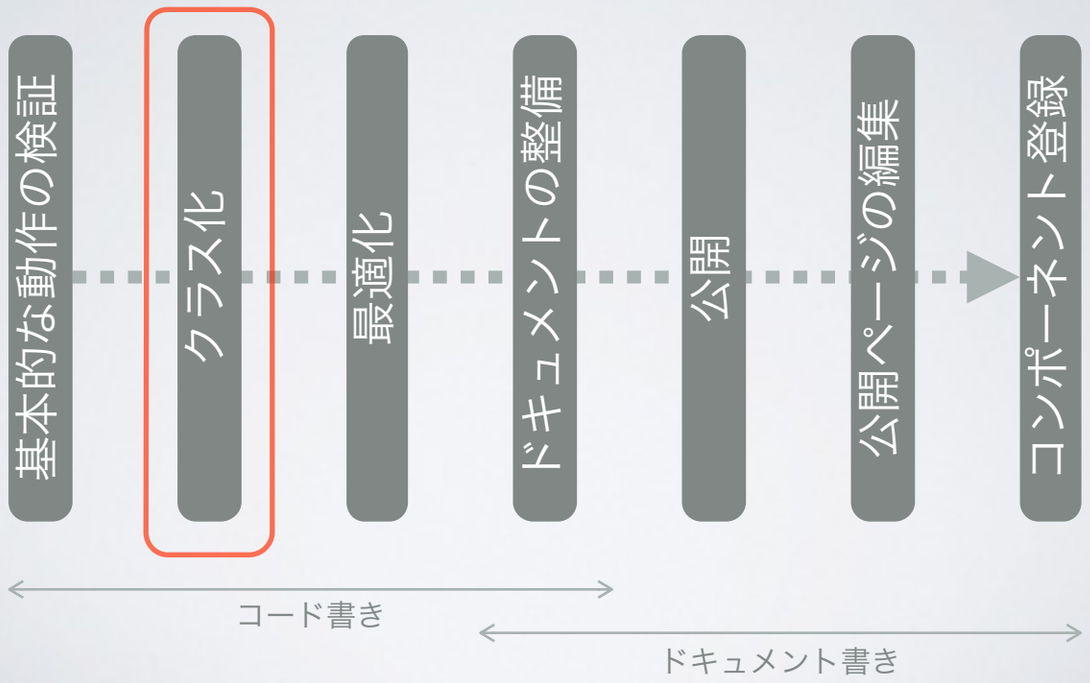


この例では，後で
「test_LM75B」という名のライブラリを
作るので「test_LM75B_Hello」という
プログラム名にしました

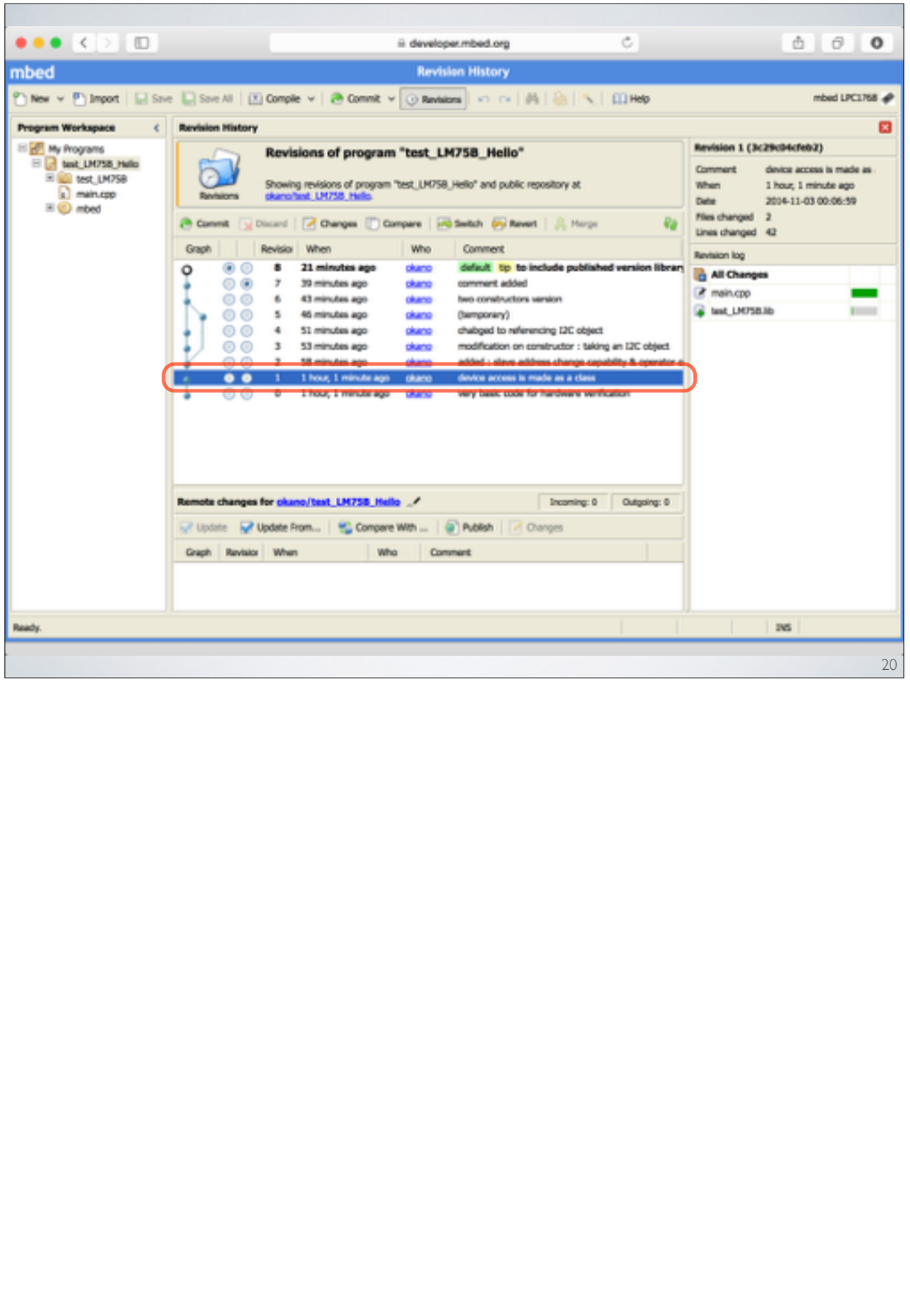
プログラムもライブラリもいつでも変更することは可能です．
公開名は公開時点で固定されるので，それまでに調整すれば問題ありません．

今回のライブラリは既に公開されているLM75Bのライブラリと重複しないように
「test_LM75B」としました．(公開者が違えば同じ名前で公開することは可能です)

ライブラリ作成の流れ



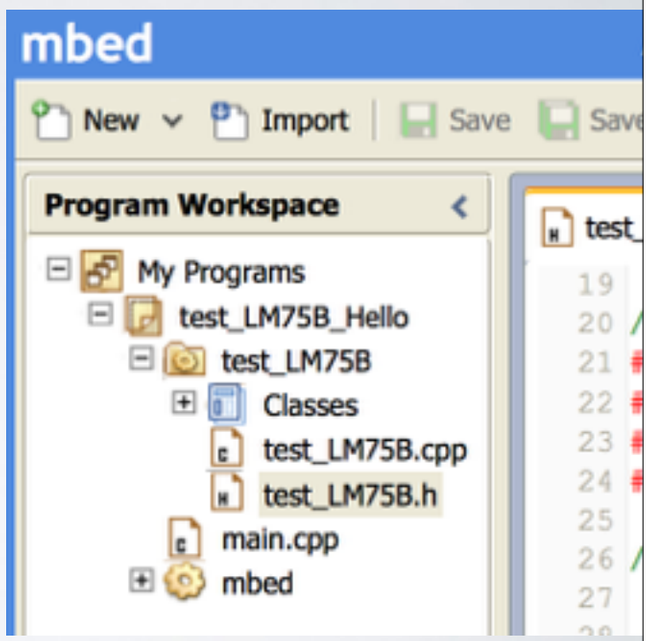
このコードのデバイスへのアクセス部分を，C++のコードとしてまとめ直します.



履歴でいうと，このリビジョン

クラス化

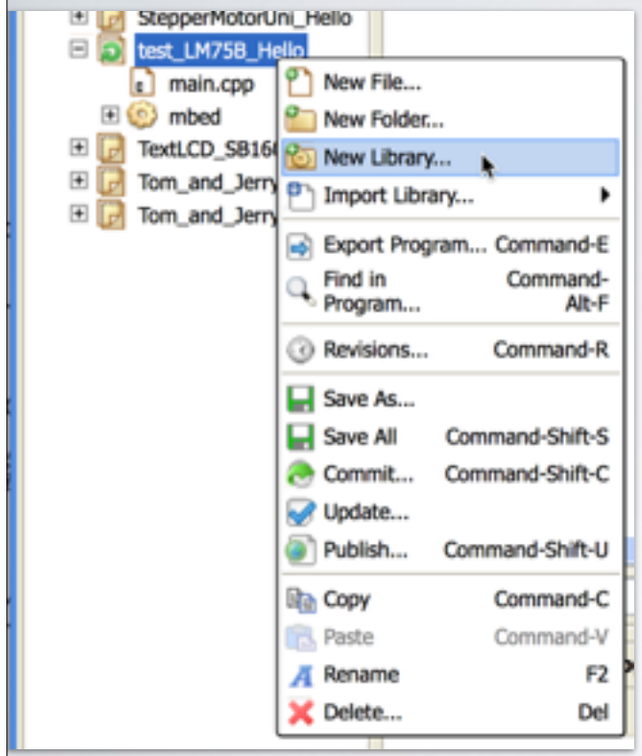
- ライブラリ・フォルダを作って，デバイス依存コードとアプリケーションを分離
- クラスは「.cpp」と「.h」のファイルに



デバイスのアクセス部分は，ライブラリフォルダ下の.cppと.hファイル内に.

.hはクラスの定義，.cppはメンバ関数の実装です.

main.cppはmain関数と，ライブラリを使う最低限のコードに (多機能なライブラリなら，別途テスト用コードを付くたりします)

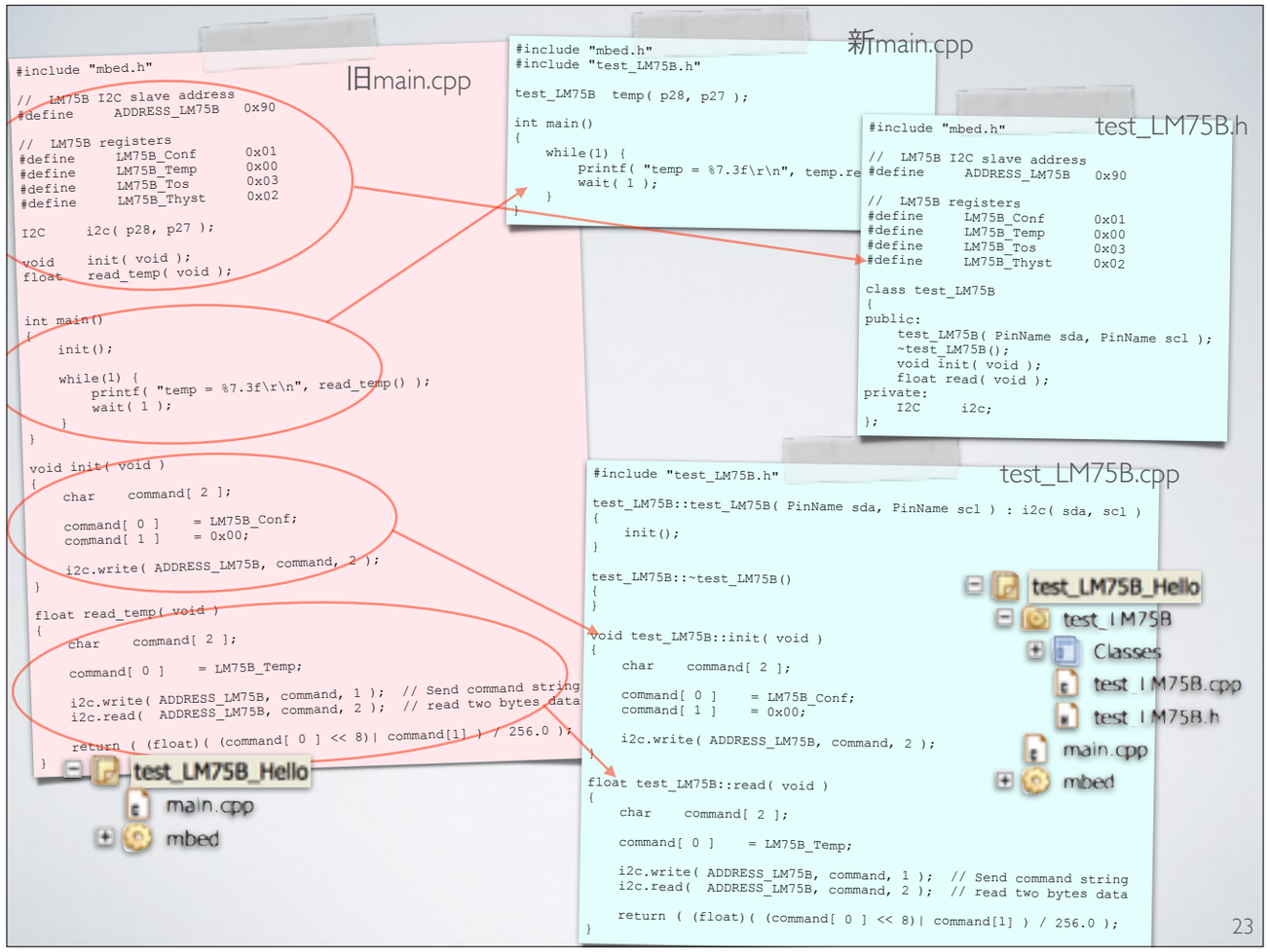


ちなみに..

- 各プログラムやフォルダ，ライブラリの操作は右クリックで
  - 新規追加 / 消去
  - Export(書き出し)
  - 公開
  - コピー / ペースト
  - 名称変更

  などができます

実際に独自のライブラリを作成する際には，プログラム・アイコンを右クリックして「New Library …」メニューを選択.
さらにそのフォルダ内で「New FIle …」を選択すると新しいファイルが作られます.
その他のプログラムに関する操作も右クリックで出てきます.



左側のピンク色のコードを，右側の3つのファイルに分けました.

main.cppはアプリケーション部分

ライブラリフォルダ内の.hはクラスの定義，.cppはメンバ関数の実装部分です

test_LM75B.h

```
#include "mbed.h"

//  LM75B I2C slave address
#define     ADDRESS_LM75B   0x90

//  LM75B registers
#define     LM75B_Conf      0x01
#define     LM75B_Temp      0x00
#define     LM75B_Tos       0x03
#define     LM75B_Thyst     0x02

class test_LM75B
{
public:
    test_LM75B( PinName sda, PinName scl );
    ~test_LM75B();
    void init( void );
    float read( void );
private:
    I2C     i2c;
};
```

定数の宣言

関数のプロトタイプ

インスタンス変数

I²C

test_LM75B.cpp

```
#include "test_LM75B.h"

test_LM75B::test_LM75B( PinName sda, PinName scl ) : i2c( sda, scl )
{
    init();
}

test_LM75B::~test_LM75B()
{
}

void test_LM75B::init( void )
{
    char    command[ 2 ];

    command[ 0 ]    = LM75B_Conf;
    command[ 1 ]    = 0x00;

    i2c.write( ADDRESS_LM75B, command, 2 );
}

float test_LM75B::read( void )
{
    char    command[ 2 ];

    command[ 0 ]    = LM75B_Temp;

    i2c.write( ADDRESS_LM75B, command, 1 );  // Send command string
    i2c.read(  ADDRESS_LM75B, command, 2 );  // read two bytes data

    return ( (float)( (command[ 0 ] << 8)| command[1] ) / 256.0 );
}
```

コンストラクタ

クラス変数の初期化は「初期化リスト」で

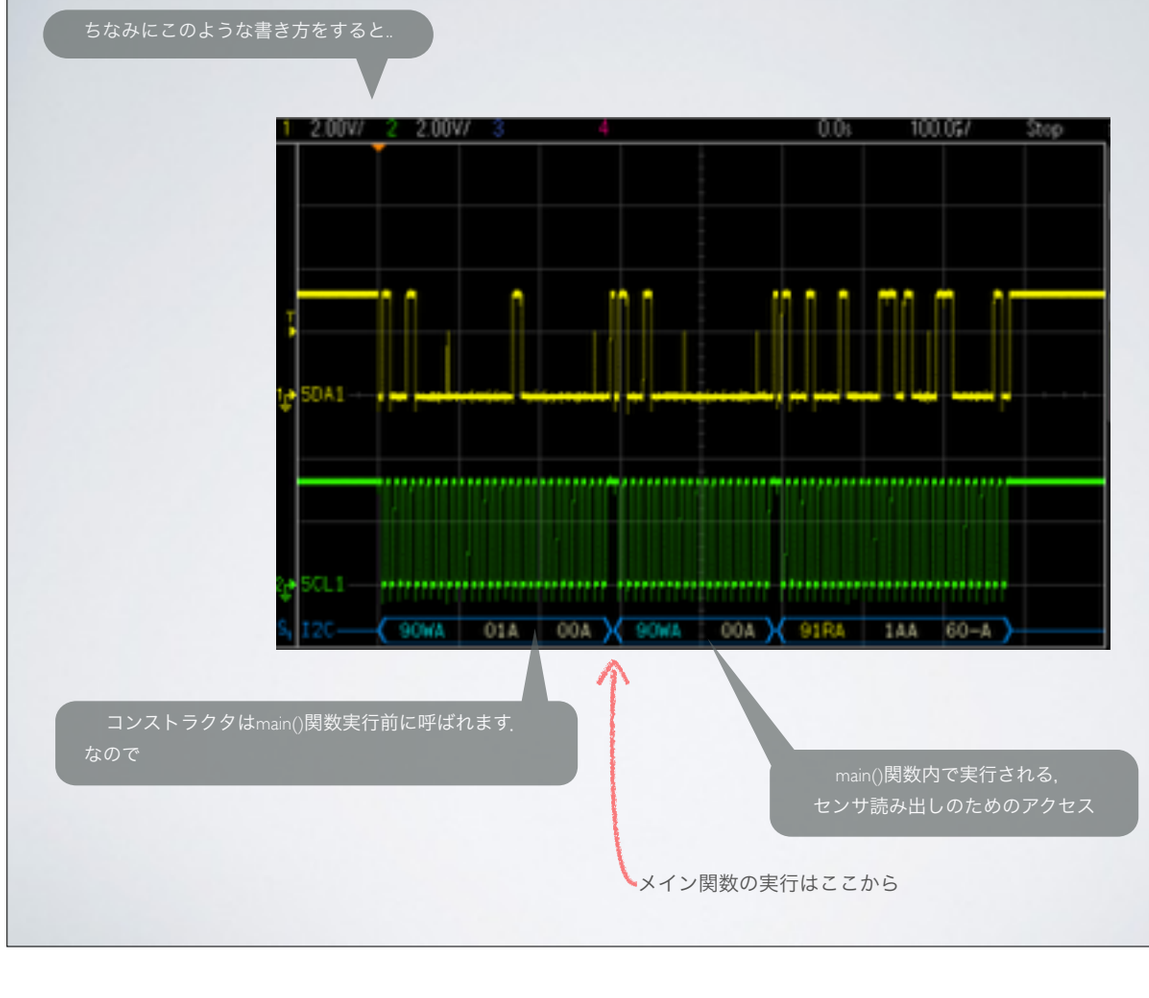
デストラクタ

メンバ関数：初期化

メンバ関数：デバイス読み出し

.hファイルには定義部分．クラス内では外部(クラス外から)公開(public)する部分と隠す(private)部分を分け，関数のプロトタイプとインスタンス変数がかかれています．

.cppファイルにはインスタンス作成時に呼ばれるコンストラクタ，破棄されるときに呼ばれるデストラクタ．あとは初期化部分をまとめたinit()関数と温度を読み出すためのread()関数を設けました．

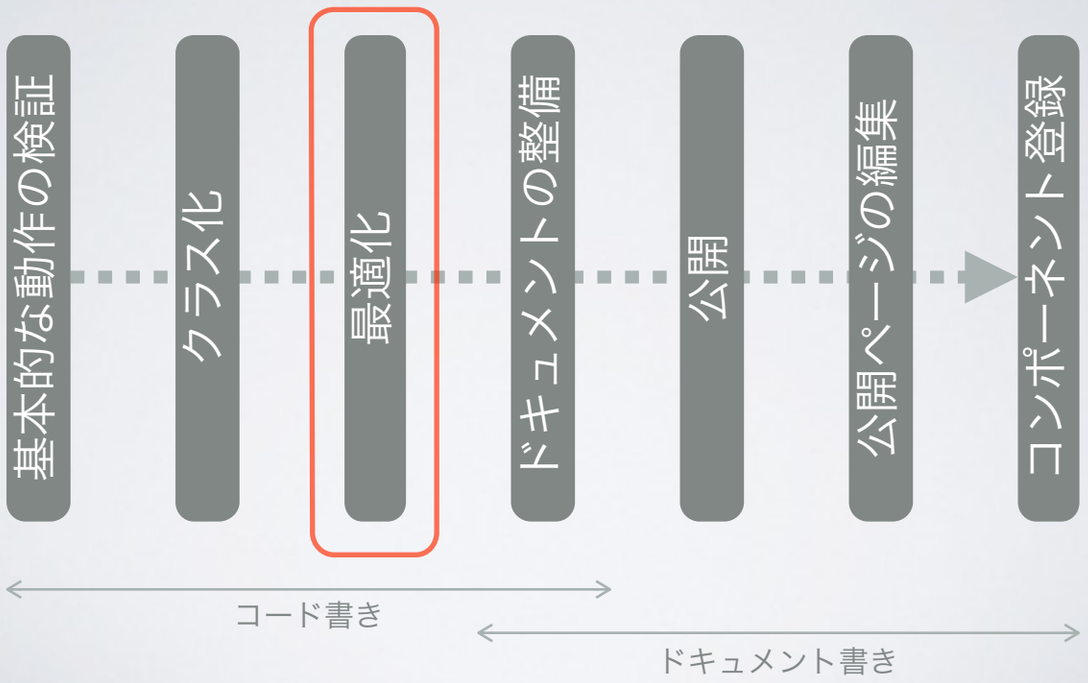


実行して確認.

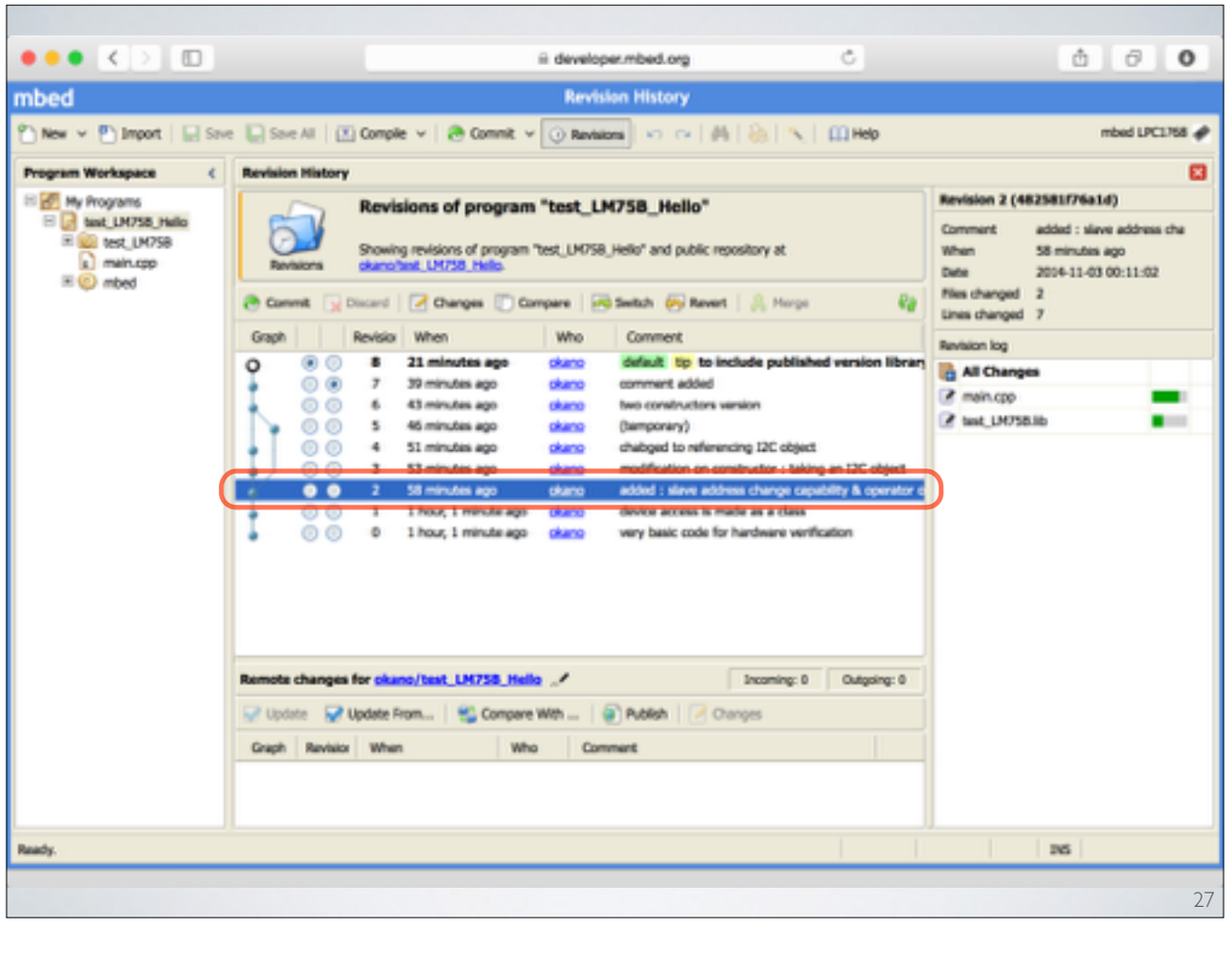
コンストラクタ内でのI^2Cがmain関数が呼ばれる前に実行されています (静的にインスタンスが作られているため)

温度の値の読み出しも，問題なく行われました

ライブラリ作成の流れ



次にC++的にもうすこし書き進めてみます



コードの履歴でいうと，このリビジョンです

test_LM75B.h

```
#include "mbed.h"

//  LM75B I2C slave address
#define     ADDRESS_LM75B   0x90

//  LM75B registers
#define     LM75B_Conf      0x01
#define     LM75B_Temp      0x00
#define     LM75B_Tos       0x03
#define     LM75B_Thyst     0x02

class test_LM75B
{
public:
    test_LM75B( PinName sda, PinName scl, char address = ADDRESS_LM75B );
    ~test_LM75B();
    void    init( void );
    float   read( void );
    operator float( void );
private:
    I2C     i2c;
    char    adr;
};
```

3番目の引数が与えられなかった場合は
デフォルトの値として0x90を使う

test_LM75B.cpp

```
.....
test_LM75B::test_LM75B( PinName sda, PinName scl, char address ) : i2c( sda, scl ), adr( address )
{
    init();
}
.....
```


温度センサのスレーブアドレスが違っていたり，あるいはセンサを複数使いたくなることがあるかもしれません.
これに対応するために，コンストラクタにスレーブアドレスを渡せるようにしてみました.
もしスレーブアドレスが指定されなければ，デフォルトのスレーブアドレスが使われるようにしています.
スレーブアドレスはインスタンス変数として保存されます．スレーブアドレスは初期化リストで代入されるようにしました.

main.cpp

```
#include "mbed.h"
#include "test_LM75B.h"

test_LM75B  temp0( p28, p27, 0x90 );
test_LM75B  temp1( p28, p27, 0x92 );

int main()
{
    while(1) {
        printf( "temp0 = %7.3f\r\n", temp0.read() );
        printf( "temp1 = %7.3f\r\n", temp1.read() );
        wait( 1 );
    }
}
```

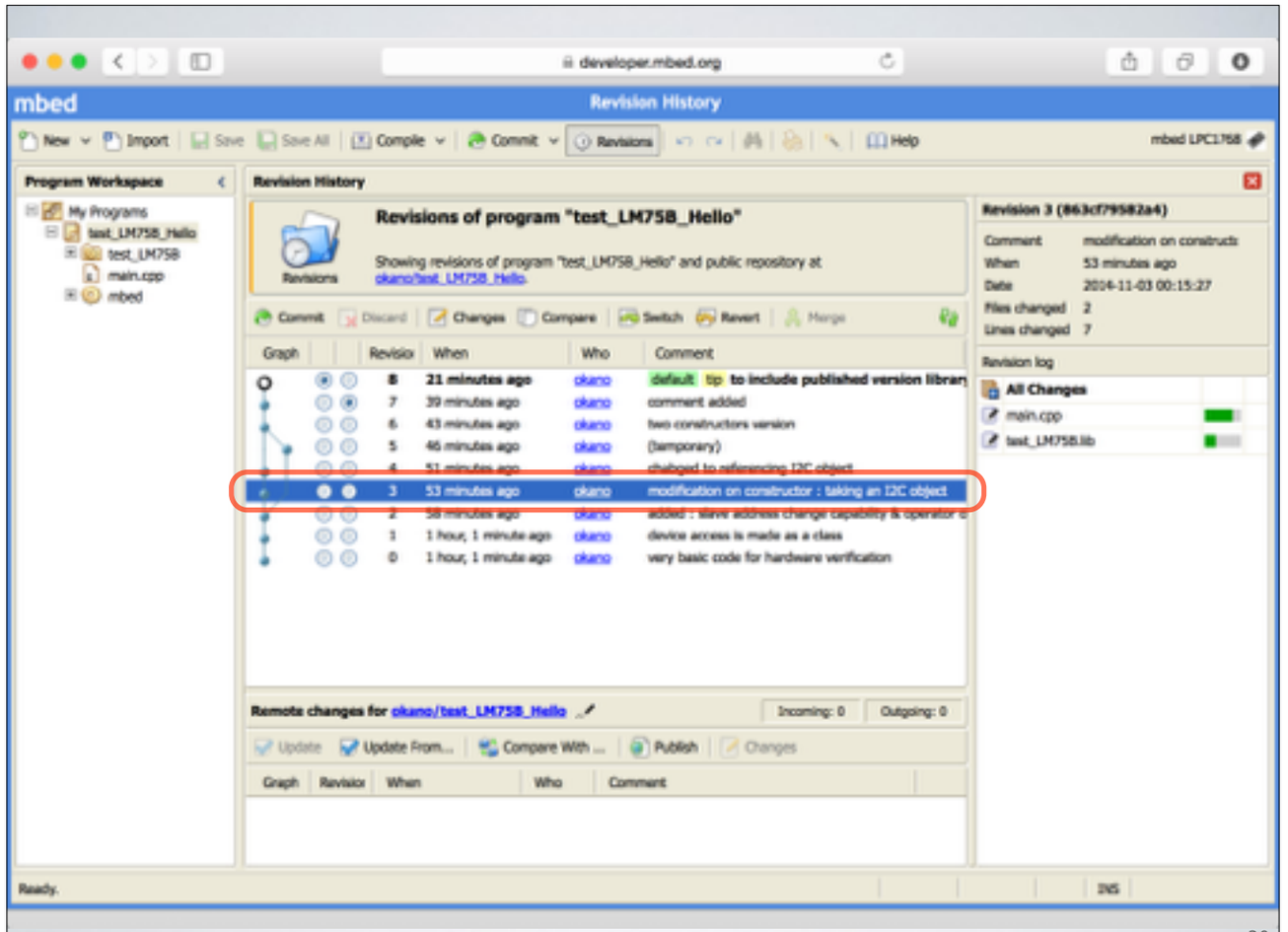
スレーブアドレス=0x90のセンサのインスタンス

スレーブアドレス=0x92のセンサのインスタンス

前ページのようなクラスにしておくと，
このような使い方が可能になります

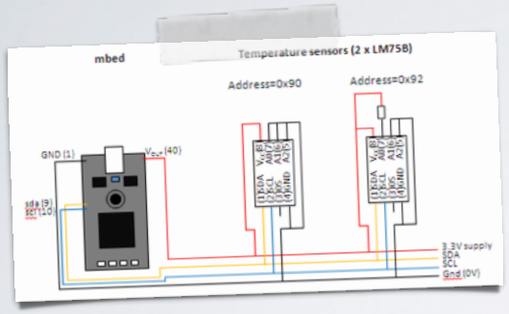
前ページのようなクラスにしておくと，アプリケーションからの使い勝手が良くなります.

この例では2つの温度センサを使おうとしており，センサにはそれぞれ0x90，0x92のスレーブアドレスが設定されています.



I2Cインスタンスを重複して作ってもいいのか？

素朴な疑問




```
#include "mbed.h"
#include "test_LM75B.h"

test_LM75B  temp0( p28, p27, 0x90 );
test_LM75B  temp1( p28, p27, 0x92 );

int main()
{
    while(1) {
        printf( "temp0 = %7.3f\r\n", temp0.read() );
        printf( "temp1 = %7.3f\r\n", temp1.read() );
        wait( 1 );
    }
}
```

Q

同時に2つのインスタンスを宣言すると
各温度センサのインスタンス内に
それぞれ別のI2Cクラスのインスタンスが
作られてしまう←これは問題ないか？

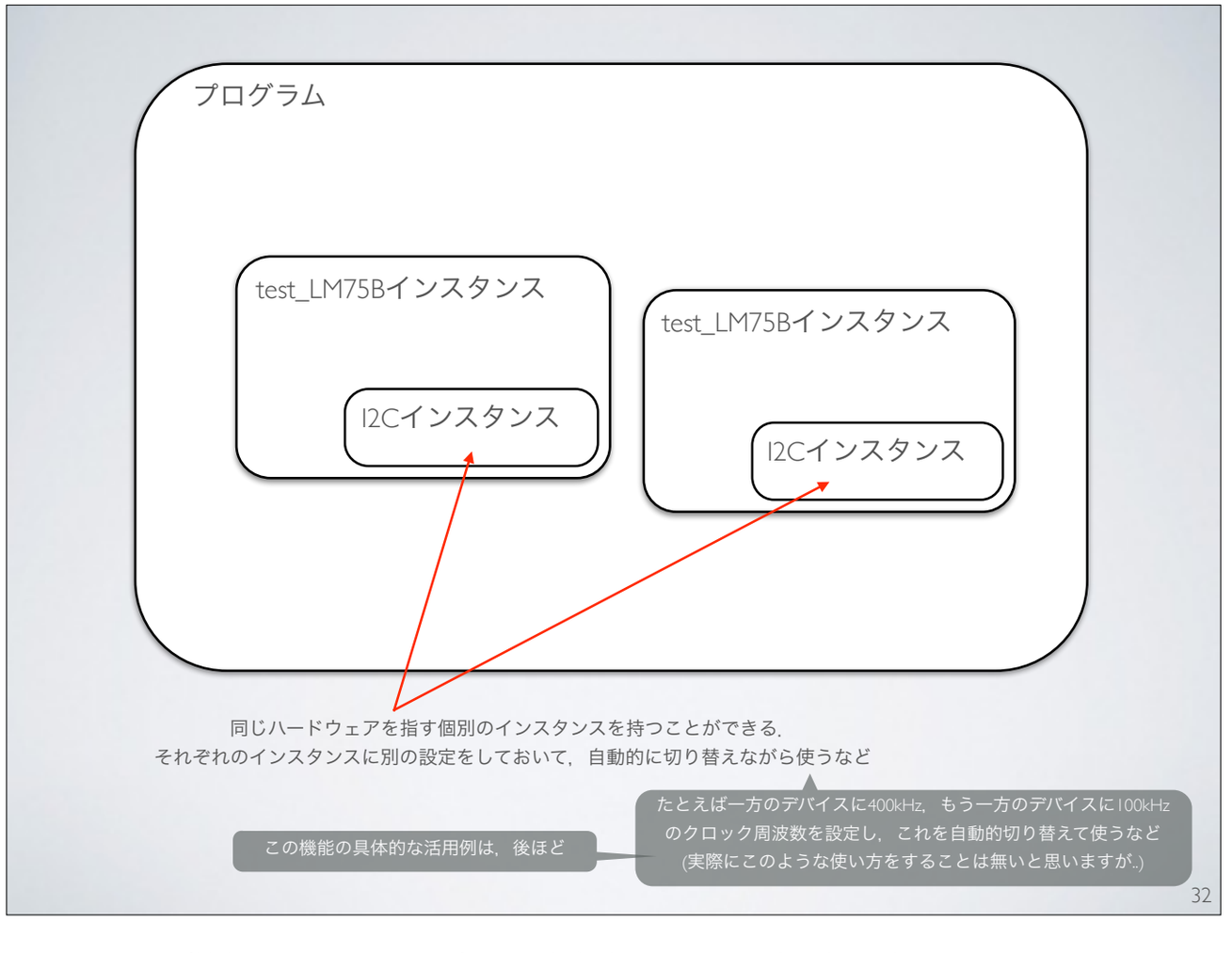
A

問題ありません.
mbed SDKは同一インターフェースに
複数のインスタンスを持たせることが
できるように作られています.

同じインターフェースのインスタンスを複数作っても問題ないのか？

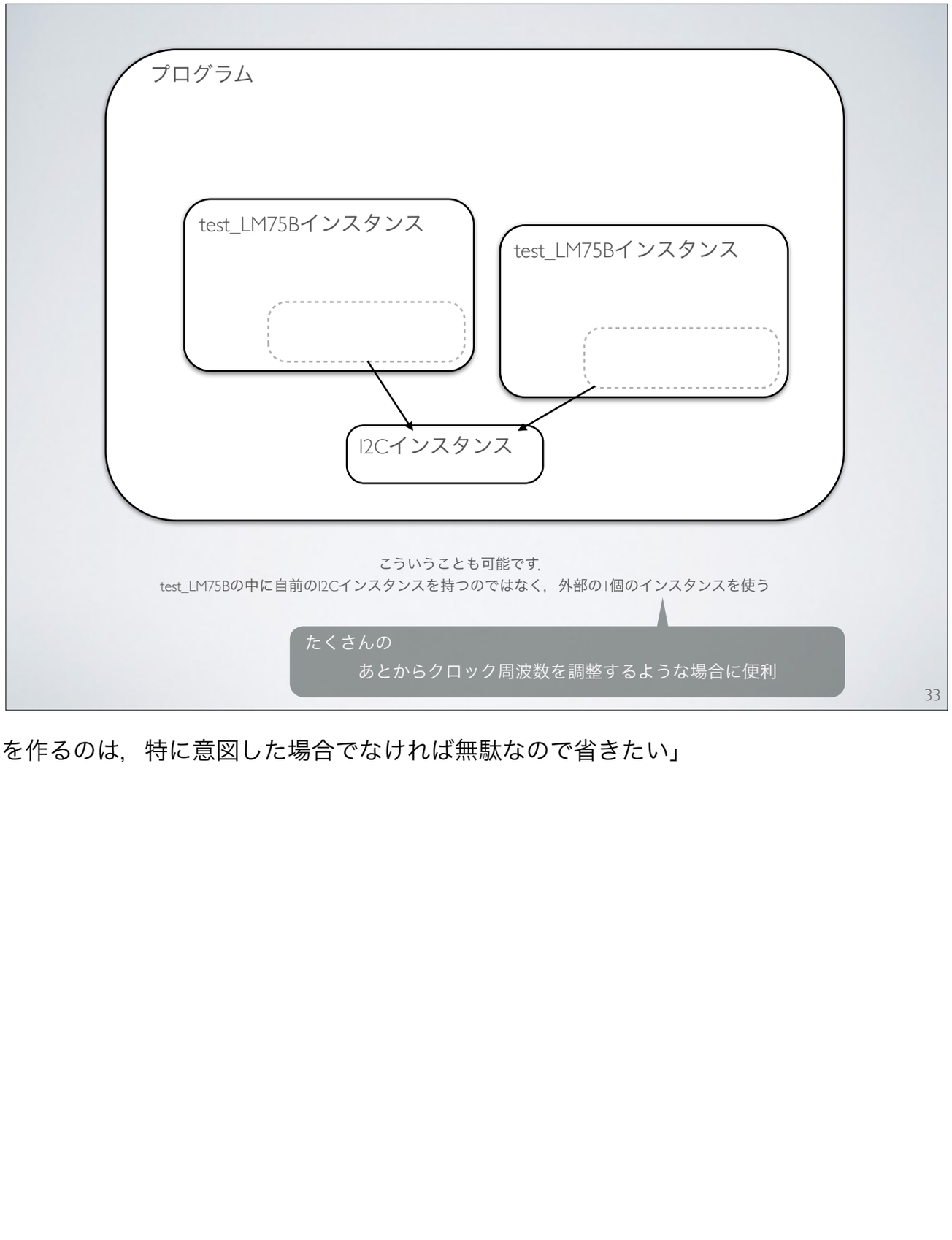
この例では各センサのインスタンス内に，同じインターフェース(I2Cのハードウェア)を使うインスタンスが作られてしまいます.

mbed-SDKは，ひとつのハードウェアを複数のインスタンスで操作しても問題ないように作られています.



各センサのインスタンス内に，インターフェース(この場合，I C)のインスタンスを保たせることができます．

このような実装にするとメモリ消費は増加しますが，センサのインスタンス毎に別の設定をもたせたりすることも可能になっています．



こういうことも可能です.
test_LM75Bの中に自前のI2Cインスタンスを持つのではなく，外部の1個のインスタンスを使う

たくさんの
あとからクロック周波数を調整するような場合に便利

こんな声も「この重複するインスタンスを作るのは，特に意図した場合でなければ無駄なので省きたい」

```
#include "mbed.h"
#include "test_LM75B.h"

test_LM75B  tmp[]    = {
    test_LM75B( p28, p27, 0x90 ),
    test_LM75B( p28, p27, 0x92 ),
    test_LM75B( p28, p27, 0x93 ),
    test_LM75B( p28, p27, 0x94 ),
    test_LM75B( p28, p27, 0x96 ),
    test_LM75B( p28, p27, 0x98 ),
    test_LM75B( p28, p27, 0x9A ),
    test_LM75B( p28, p27, 0x9C )
};

int main()
{
       for ( int i = 0; i < 4; i++ ) {
            printf( "temp = %7.3f\r\n",
             (float)(tmp[ i ]) );
        }
        wait( 1 );
    }
}
```

```
#include "mbed.h"
#include "test_LM75B.h"

I2C         i2c( p28, p27 );

test_LM75B  tmp[]    = {
    test_LM75B( i2c, 0x90 ),
    test_LM75B( i2c, 0x92 ),
    test_LM75B( i2c, 0x93 ),
    test_LM75B( i2c, 0x94 ),
    test_LM75B( i2c, 0x96 ),
    test_LM75B( i2c, 0x98 ),
    test_LM75B( i2c, 0x9A ),
    test_LM75B( i2c, 0x9C )
};

int main()
{
    i2c.frequency( 10 * 1000 );

    while(1) {
        for ( int i = 0; i < 4; i++ ) {
            printf( "temp = %7.3f\r\n”,
            (float)(tmp[ i ]) );
        }
        wait( 1 );
    }
}
```

たとえば..

I²Cを長く引き回すような場合，わざとクロック周波数を落として通信を行うことがある．
これを行うのにクラス内部に触ること無く，外部から変更を加える事はできないか？

たとえば..

また各クラスの外部からインターフェースのパラメータを変更もしたい．

(インターフェースのクラスを継承するのではなく)

インターフェースのインスタンスを別に宣言して，センサのコンストラクタに渡せば..

```
#include "mbed.h"
#include "test_LM75B.h"

I2C         i2c( p28, p27 );
test_LM75B  temp( i2c );

int main()
{
    float   t;

    i2c.frequency( 400 * 1000 );

    while(1) {
        t   = temp;
        printf( "temp = %7.3f\r\n", t );
        wait( 1 );
    }
}
```

main.cpp

こんなコードを書いてみました.

I2Cインスタンスをまず作っておき，それをセンサのインスタンスを作るときに渡す.

main関数内からクロック周波数変更を出来るように！

```
I2C          i2c( p28, p27 );
test_LM75B   temp( i2c );

int main()
{
    float   t;

    i2c.frequency( 400 * 1000 );

    while(1) {
        t   = temp;
        printf( "temp = %7.3f\r\n", t );
        wait( 1 );
    }
}
```

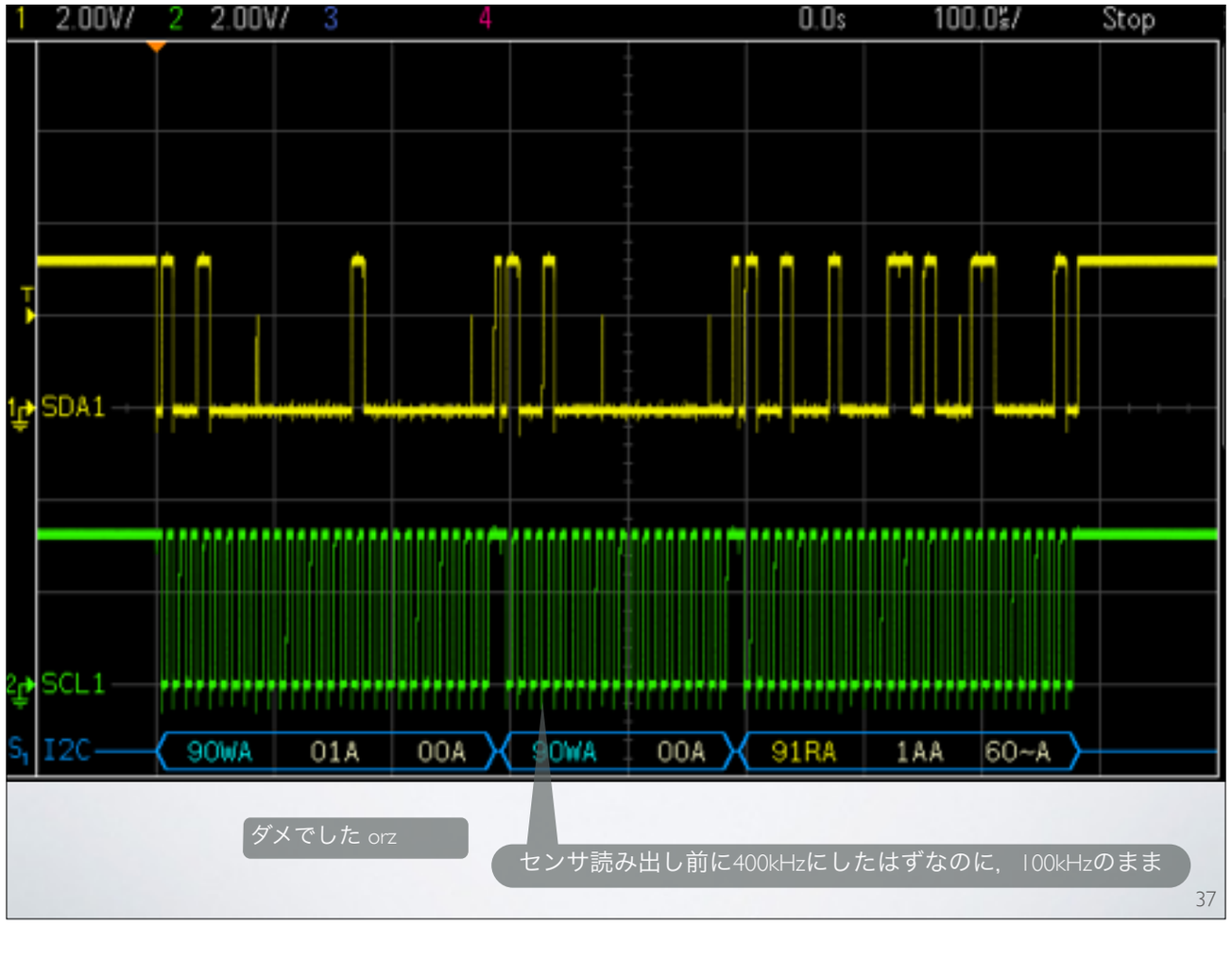
main.cpp

```
class test_LM75B
{
public:
    test_LM75B( I2C i2c_obj, char address = ADDRESS_LM75B );
    ~test_LM75B();
    void    init( void );
    float   read( void );
    operator float( void );
private:
    I2C     i2c;
    char    adr;
};
```

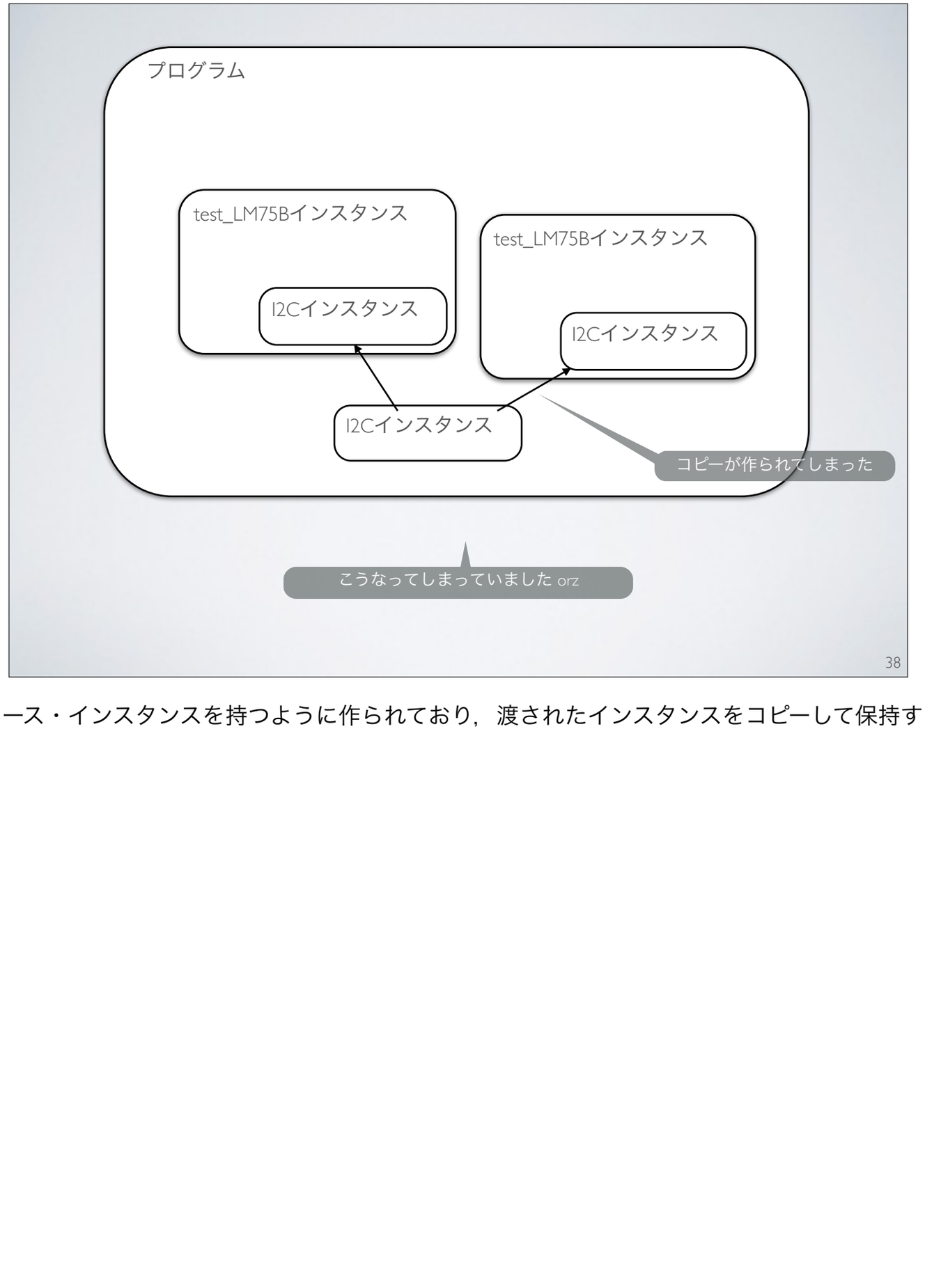
```
#include "test_LM75B.h"

test_LM75B::test_LM75B( I2C i2c_obj, char address )
                        : i2c( i2c_obj ), adr( address )
{
    init();
}
```

test_LM75B.cpp

サンプルコード全体はこんな感じ



I2Cのインスタンスを渡してもダメでした orz



センサのクラスでは内部にインターフェース・インスタンスを持つように作られており，渡されたインスタンスをコピーして保持するようになっていました.

```
I2C         i2c( p28, p27 );
test_LM75B  temp( i2c );

int main()
{
    float   t;

    i2c.frequency( 400 * 1000 );

    while(1) {
        t   = temp;
        printf( "temp = %7.3f\r\n", t );
        wait( 1 );
    }
}
```

main.cpp

```
class test_LM75B
{
public:
    test_LM75B( I2C i2c_obj, char address = ADDRESS_LM75B );
    ~test_LM75B();
    void    init( void );
    float   read( void );
    operator float( void );
private:
    I2C     i2c;
    char    adr;
};
```

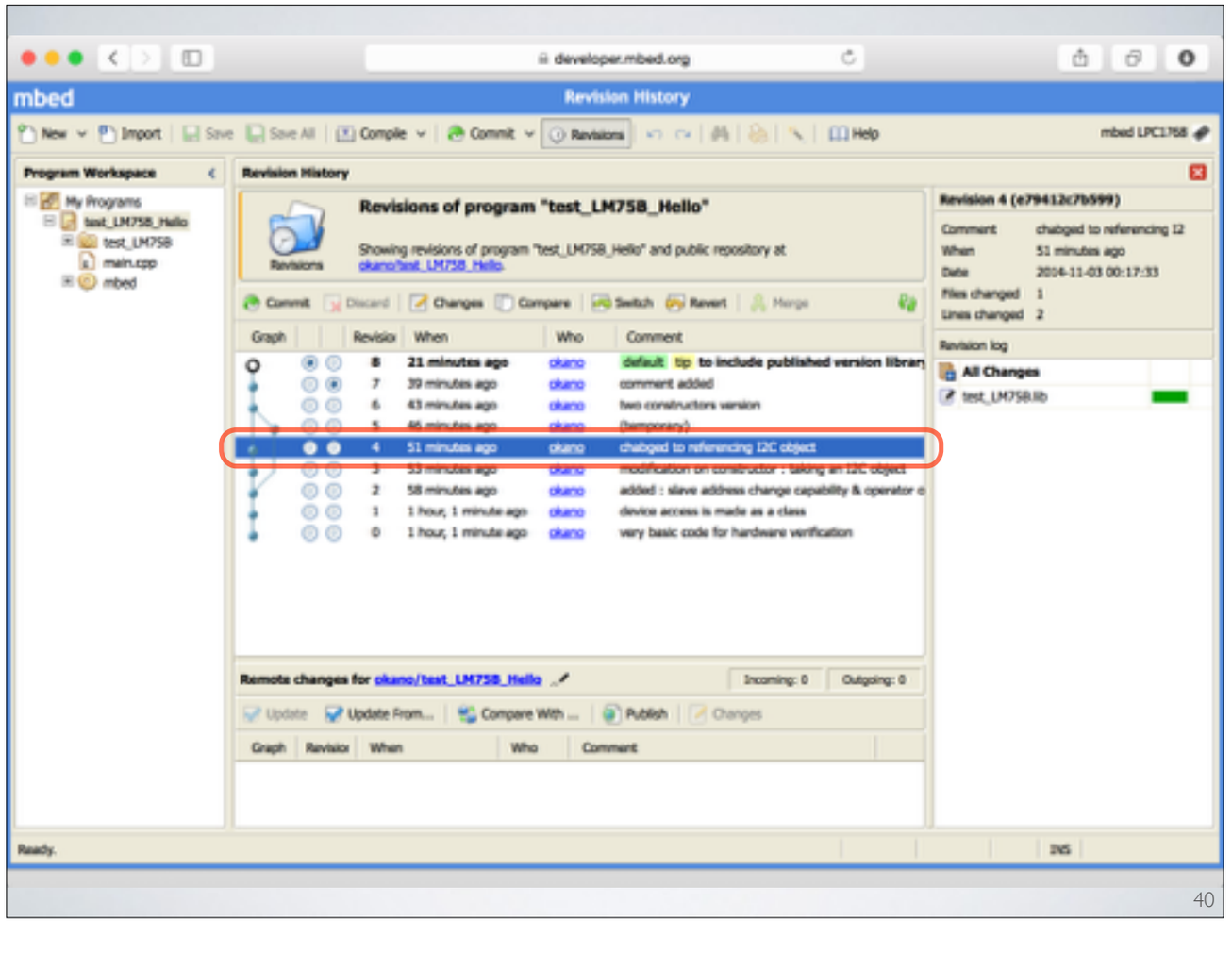
```
#include "test_LM75B.h"

test_LM75B::test_LM75B( I2C i2c_obj, char address )
                        : i2c( i2c_obj ), adr( address )
{
    init();
}
```

test_LM75B.cpp

残念ながらこれはうまく動きませんでした.
test_LM75Bインスタンス内にはI2Cインスタンスが
宣言されているため，オブジェクトが
コピーされてしまいます

これを大中さんに教えていただきました m(_ _)m

トラ技10月号特集第一章の大中さんに，問題点と解決法を教えていただきました.
ありがとうございます.


developer.mbed.org
mbed
Revision History
New
Import
Save
Save All
Compile
Commit
Revisions
Help
mbed LPC1768
Program Workspace
My Programs
test_LM75B_Hello
test_LM75B
main.cpp
mbed
Revision History
Revisions of program "test_LM75B_Hello"
Showing revisions of program "test_LM75B_Hello" and public repository at okano/test_LM75B_Hello.
Commit
Discard
Changes
Compare
Switch
Revert
Merge
Graph
Revisio
When
Who
Comment
8 21 minutes ago okano default tip to include published version librar
7 39 minutes ago okano comment added
6 43 minutes ago okano two constructors version
5 46 minutes ago okano (temporary)
4 51 minutes ago okano chabged to referencing I2C object
3 53 minutes ago okano modification on constructor : taking an I2C object
2 58 minutes ago okano added : slave address change capability & operator o
1 1 hour, 1 minute ago okano device access is made as a class
0 1 hour, 1 minute ago okano very basic code for hardware verification
Revision 4 (e79412c7b599)
Comment chabged to referencing I2
When 51 minutes ago
Date 2014-11-03 00:17:33
Files changed 1
Lines changed 2
Revision log
All Changes
test_LM75B.lib
Remote changes for okano/test_LM75B_Hello
Incoming: 0
Outgoing: 0
Update
Update From...
Compare With ...
Publish
Changes
Graph
Revisio
When
Who
Comment
Ready.
INS

対策したコード

```
I2C         i2c( p28, p27 );
test_LM75B  temp( i2c );

int main()
{
    float   t;

    i2c.frequency( 400 * 1000 );

    while(1) {
        t   = temp;
        printf( "temp = %7.3f\r\n", t );
        wait( 1 );
    }
}
```

main.cpp

```
class test_LM75B
{
public:
    test_LM75B( I2C &i2c_obj, char address = ADDRESS_LM75B );
    ~test_LM75B();
    void    init( void );
    float   read( void );
    operator float( void );
private:
    I2C     &i2c;
    char    adr;
};
```

```
#include "test_LM75B.h"

test_LM75B::test_LM75B( I2C &i2c_obj, char address )
                        : i2c( i2c_obj ), adr( address )
{
    init();
}
```

test_LM75B.cpp

元のI2Cインスタンスを参照できるようにしました.
ここではポインタではなく「参照型」を使っています.

オブジェクト自体をコピーするのではなく,
元のオブジェクトへの参照を渡しています.

参照型には代入はできません.
なので, **初期化リスト**での初期化が必要です.

main関数側は変更なし

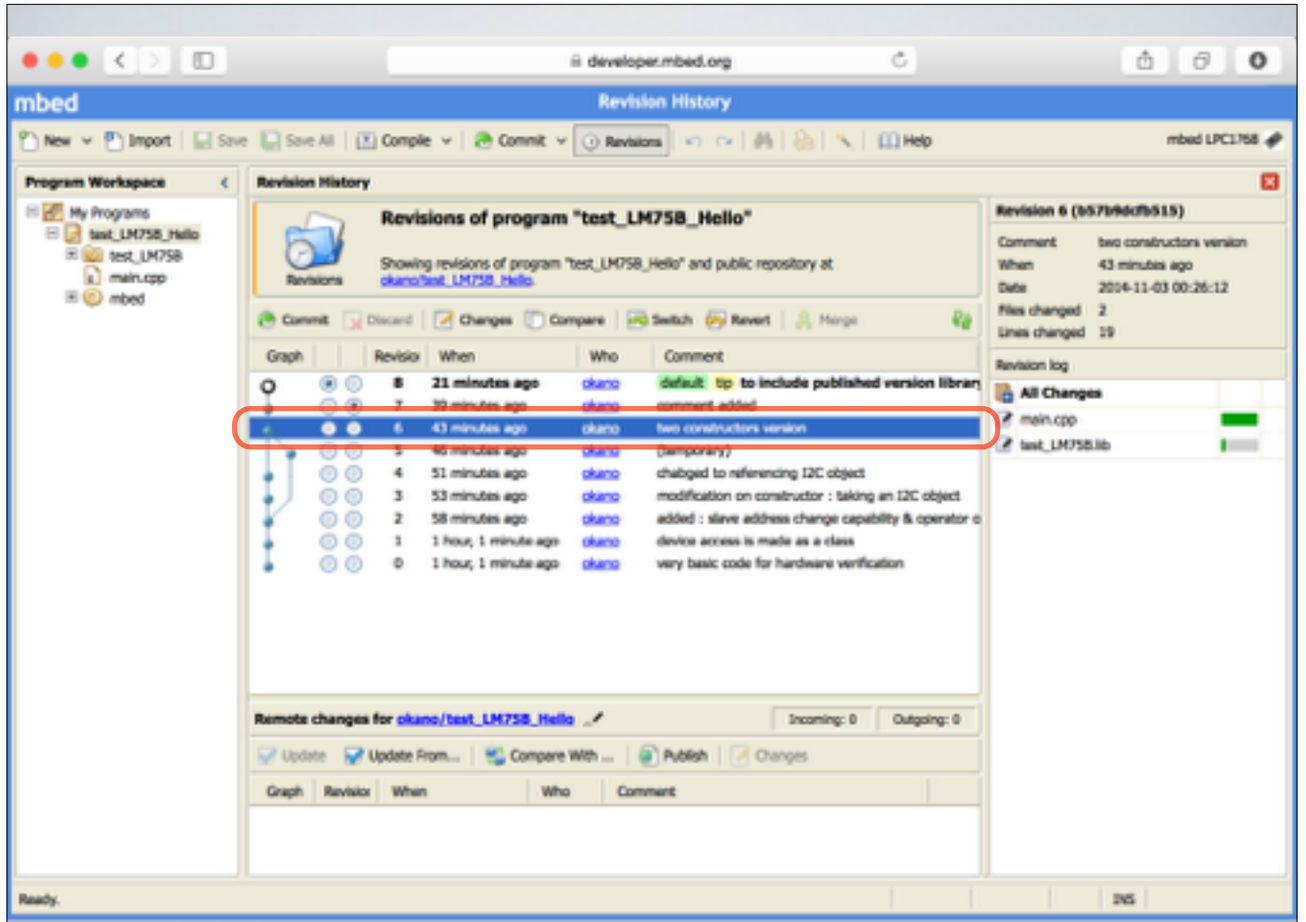
クラス側ではI2Cインスタンスそのものではなく，参照型で保持.

こうすることでクラス内部からは元のI2Cインスタンスを参照するようになる

注：「参照型へは代入ができない」ので初期化リストを使う必要がある


1 2.00V/ 2 2.00V/ 3 4 0.0s 500.0µs/ Stop
SDA1
SCL1
I2C
初期化．100kHz
できました．400kHz

できた！


developer.mbed.org
mbed
Revision History
New
Import
Save
Save All
Compile
Commit
Revisions
Help
mbed LPC1768
Program Workspace
My Programs
test_LM75B_Hello
test_LM75B
main.cpp
mbed
Revision History
Revisions of program "test_LM75B_Hello"
Showing revisions of program "test_LM75B_Hello" and public repository at okano/test_LM75B_Hello.
Commit
Discard
Changes
Compare
Switch
Revert
Merge
Graph
Revision
When
Who
Comment
8 21 minutes ago okano default tip to include published version library
6 43 minutes ago okano two constructors version
4 51 minutes ago okano chabged to referencing I2C object
3 53 minutes ago okano modification on constructor : taking an I2C object
2 58 minutes ago okano added : slave address change capability & operator
1 1 hour, 1 minute ago okano device access is made as a class
0 1 hour, 1 minute ago okano very basic code for hardware verification
Revision 6 (b57b9dcfb515)
Comment two constructors version
When 43 minutes ago
Date 2014-11-03 00:26:12
Files changed 2
Lines changed 19
Revision log
All Changes
main.cpp
test_LM75B.lib
Remote changes for okano/test_LM75B_Hello
Incoming: 0
Outgoing: 0
Update
Update From...
Compare With ...
Publish
Changes
Graph
Revision
When
Who
Comment
Ready.
INS

さらに

2つのバージョン

- インスタンスの作成方法
  - ピン名を渡すもの
  - I2Cインスタンスを渡すもの
- これらを分けて公開するのは効率が悪いので，統合してしまう
- C++は同じ関数名でも引数の種類が違うと，別物として扱ってくれる (^^)

ユーザとしての自分は，この「ピン名を渡す方法」も「オブジェクトを渡す方法」も両方使いたい．アプリケーションによって使い分けたい．
2種類のクラスを別に作って公開する？→統合したライブラリに．

こんな仕様にしてみた

- クラス内でI2Cにアクセスする際，インスタンス本体と参照を分けて処理するのは面倒．
  - なのでI2Cはコンストラクタ以外ではすべて参照を介してアクセス
- コンストラクタは
  - ピン名を渡されたら
    - そのピンを使うI2Cインスタンスを作る
    - そのインスタンスへの参照を保存して，以降はこれを使う
  - I2Cオブジェクトを渡されたら
    - そのインスタンスへの参照を作って保存．以降はこれを使う
- デストラクタが呼ばれたら
  - 自身がI2Cインスタンスを保持しているのかどうかを確認して，保持しているなら開放する

test_LM75B.h

```
....
class test_LM75B
{
public:
    test_LM75B( PinName sda, PinName scl, char address = ADDRESS_LM75B );
    test_LM75B( I2C &i2c_obj, char address = ADDRESS_LM75B );
    ~test_LM75B();
    void    init( void );
    float   read( void );
    operator float( void );
private:
    I2C     *i2c_p;
    I2C     &i2c;
    char    adr;
};
```

2種類のコンストラクタ

I2Cインスタンスへのポインタと
参照を保存できるようにしてある

test_LM75B.cpp

```
....
test_LM75B::test_LM75B( PinName sda, PinName scl, char address )
    : i2c_p( new I2C( sda, scl ) ), i2c( *i2c_p ), adr( address )
{
    init();
}

test_LM75B::test_LM75B( I2C &i2c_obj, char address )
    : i2c_p( NULL ), i2c( i2c_obj ), adr( address )
{
    init();
}

test_LM75B::~test_LM75B()
{
    if ( NULL != i2c_p )
        delete  i2c_p;
}
....
```

ピン名を渡されたら：I2Cインスタンスを作成.
さらにクラス内部ではI2Cインスタンスへの参照を
使うのでその参照を初期化

オブジェクトを渡されたら，I2Cインスタンスは作らずに，
そのオブジェクトへの参照を作る

コンストラクタでI2Cインスタンスが作られていたかどうかを確認し，
自身がインスタンスを持っていたなら，それを開放する

(模索しながら作った例なので，これがC++の一般的なコードになっているかどうかは分かりませんが..)

ピン名を渡されたら

　　I2Cインスタンスを作成→i2c_pにポインタが保存される

　　そしてそのポインタの参照先を参照型として保存

オブジェクトが渡されたら

　　I2Cインスタンスは独自に作成しない→i2c_pをNULLに

　　渡されたオブジェクトを参照型で参照できるように

デストラクタが呼ばれたら

　　自身でI2Cインスタンスを持ってる場合，i2c_p != NULLなので，オブジェクトを捨てる

　　もしi2c_p == NULLならなにもしない

```
#include "mbed.h"
#include "test_LM75B.h"

test_LM75B  temp0( p28, p27 );

I2C         i2c( p28, p27 );
test_LM75B  temp1( i2c );


int main()
{
    float   t0;
    float   t1;

    i2c.frequency( 400 * 1000 );

    while(1) {
        t0   = temp0;
        t1   = temp1;
        printf( "temp = %7.3f, %7.3f\r\n", t0, t1 );
        wait( 1 );
    }
}
```

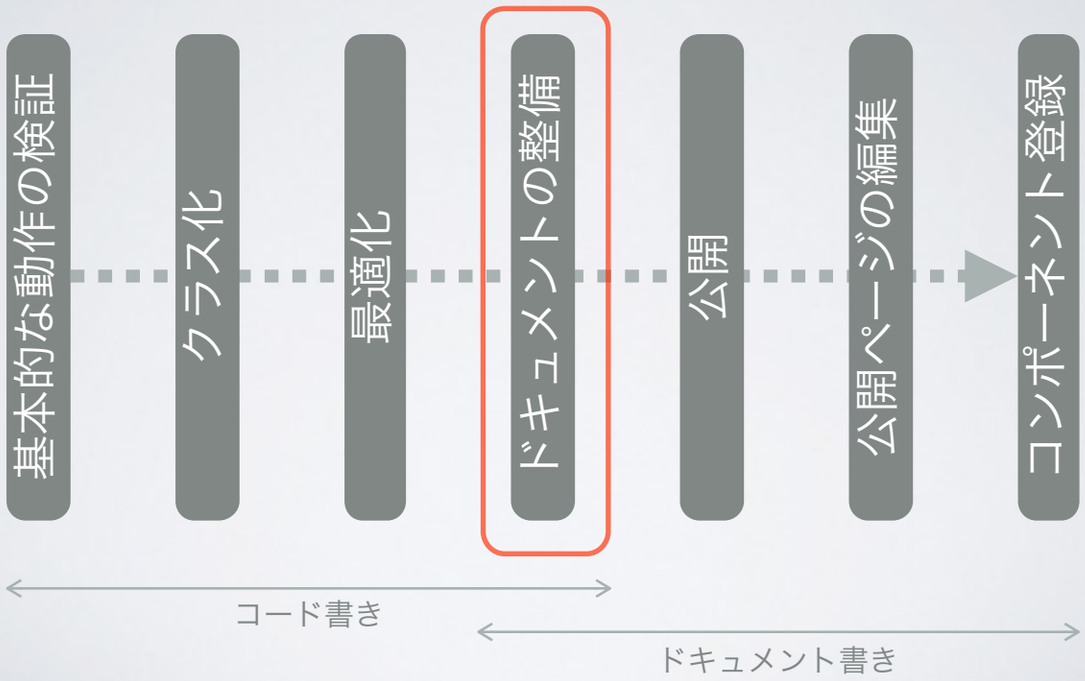
main.cpp

２つのセンサ・インスタンスを作って動作を確認.

temp0にアクセスした場合はI2Cクロック周波数のデフォルト，100kHzで

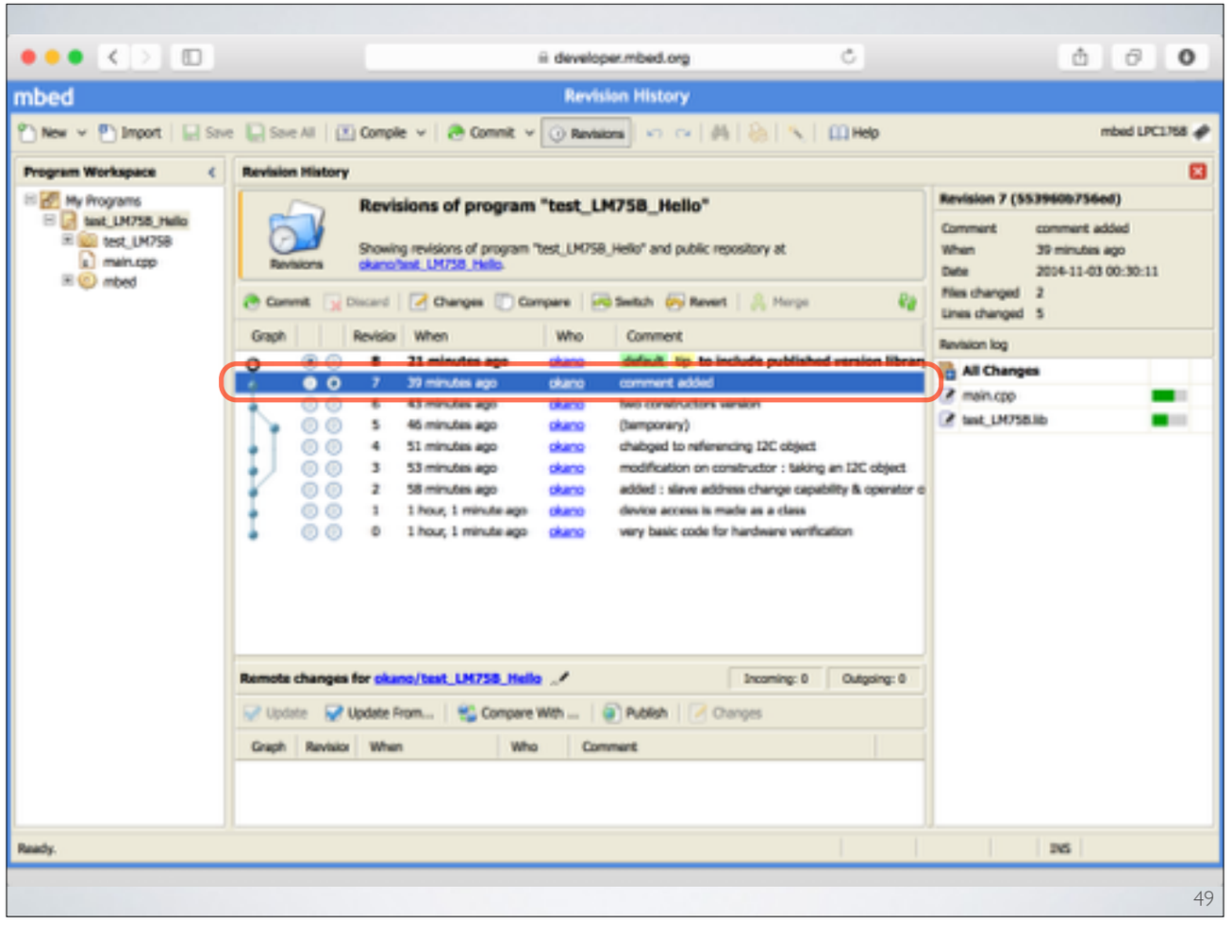
temp1にアクセスした場合は設定した通りの400kHzに

ライブラリ作成の流れ



ライブラリが仕様通りに動作したら..

公開の前にコメントを整備しましょう！

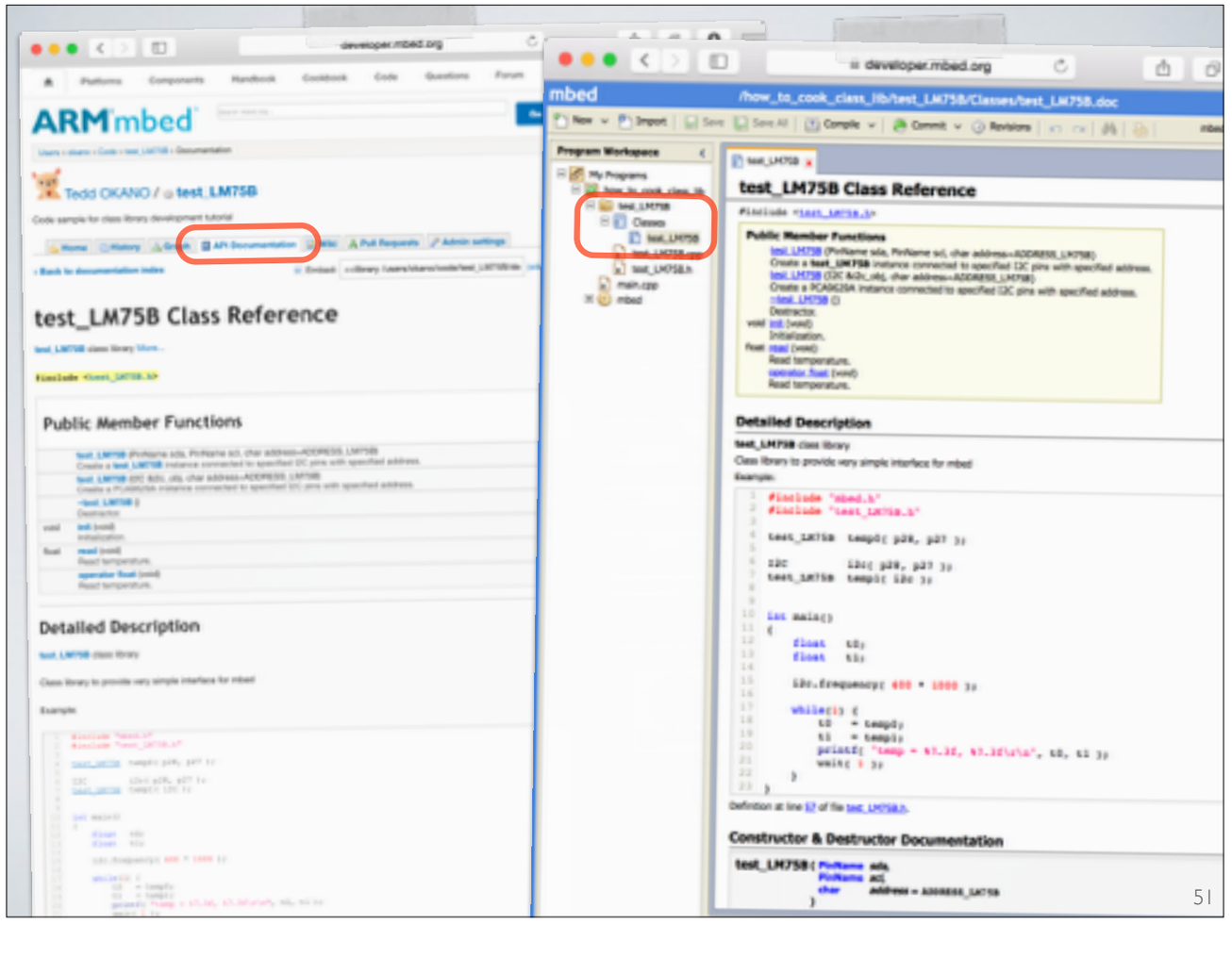

developer.mbed.org
mbed
Revision History
New
Import
Save
Save All
Compile
Commit
Revisions
Help
mbed LPC1768
Program Workspace
My Programs
test_LM75B_Hello
test_LM75B
main.cpp
mbed
Revision History
Revisions of program "test_LM75B_Hello"
Showing revisions of program "test_LM75B_Hello" and public repository at okano/test_LM75B_Hello.
Commit
Discard
Changes
Compare
Switch
Revert
Merge
Graph
Revision
When
Who
Comment
7 39 minutes ago okano comment added
5 46 minutes ago okano (temporary)
4 51 minutes ago okano chabged to referencing I2C object
3 53 minutes ago okano modification on constructor : taking an I2C object
2 58 minutes ago okano added : slave address change capability & operator
1 1 hour, 1 minute ago okano device access is made as a class
0 1 hour, 1 minute ago okano very basic code for hardware verification
Revision 7 (5539608756ed)
Comment comment added
When 39 minutes ago
Date 2014-11-03 00:30:11
Files changed 2
Lines changed 5
Revision log
All Changes
main.cpp
test_LM75B.lib
Remote changes for okano/test_LM75B_Hello
Incoming: 0
Outgoing: 0
Update
Update From...
Compare With ...
Publish
Changes
Graph
Revision
When
Who
Comment
Ready.
INS

オンラインドキュメント

- APIの解説を書いておく
- オンライン・ドキュメント
- クラスライブラリの「.h」ファイルに
- Doxygenフォーマットで
  - コメントとして書いておけば，公開後に自動的にオンラインドキュメント形式に変換してくれる

ライブラリにコメントを仕込んでおけば，mbed.orgが自動的にオンラインドキュメントを生成してくれます.

ライブラリの「.h」ファイルにDoxygenフォーマットで



Doxygenフォーマットのコメントは,

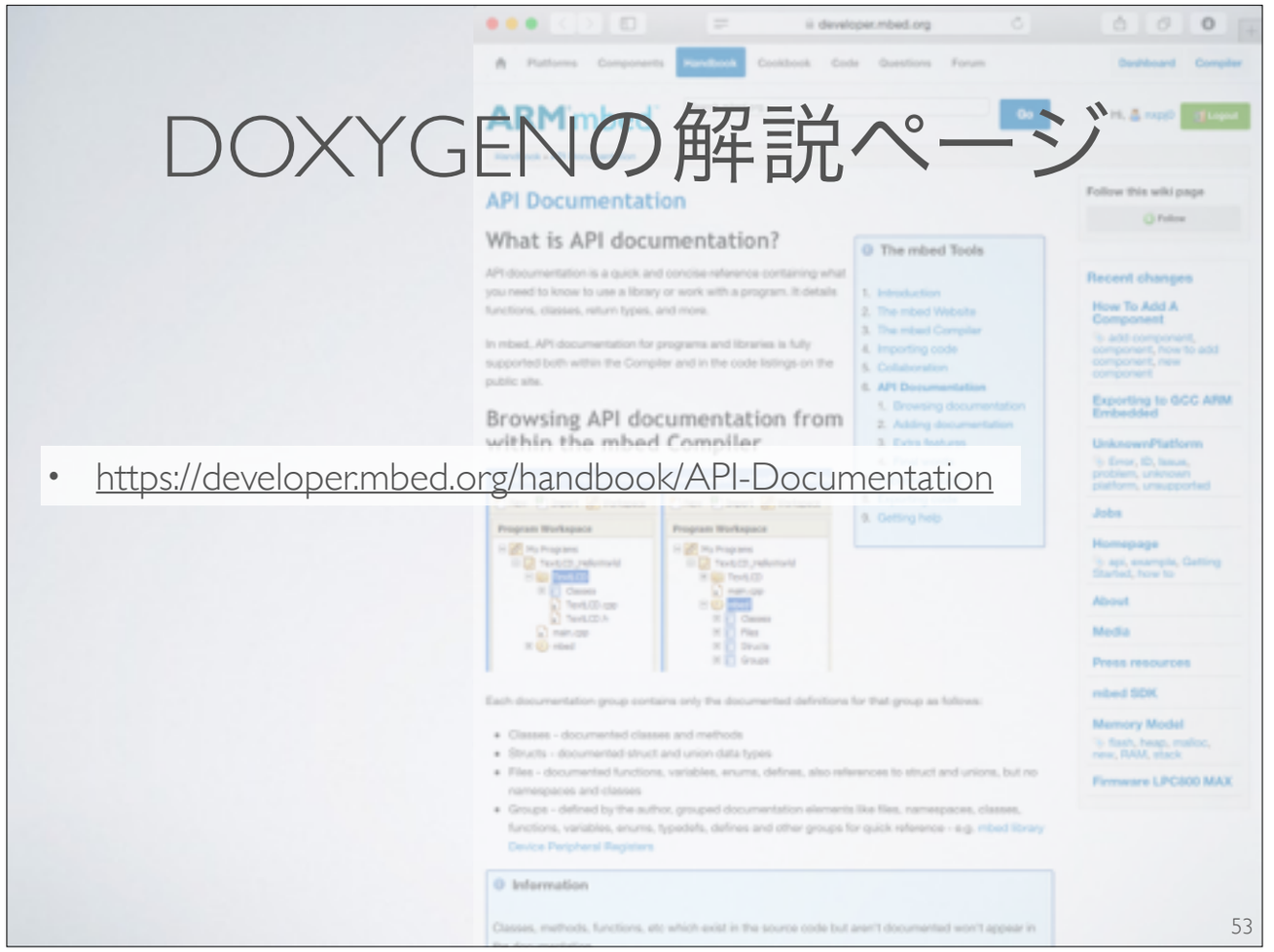
自動的に公開ページのAPI解説ページや, ライブラリのクラス説明になります

- 「/** 〜 */」などの形のコメントとして書く
- 「@」を付けて，各引数や返り値の解説を書く
- その他の情報も@を付けて
  - サンプル
  - 作者
  - バージョン
  - などなど

```
 *
 *      while(1) {
 *          t0   = temp0;
 *          t1   = temp1;
 *          printf( "temp = %7.3f, %7.3f\r\n", t0, t1 );
 *          wait( 1 );
 *      }
 *  }
 *  @endcode
 */
class test_LM75B
{
public:

    /** Create a test_LM75B instance connected to specified I2C pins with specified address
     *
     * @param sda I2C-bus SDA pin
     * @param scl I2C-bus SCL pin
     * @param address (option) I2C-bus slave address (default: 0x90)
     */
    test_LM75B( PinName sda, PinName scl, char address = ADDRESS_LM75B );

    /** Create a test_LM75B instance connected to specified I2C pins with specified address
     *
     * @param i2c_obj I2C object (instance)
     * @param address (option) I2C-bus slave address (default: 0x90)
     */
    test_LM75B( I2C &i2c_obj, char address = ADDRESS_LM75B );

    /** Destractor
     */
    ~test_LM75B();

    /** Initialization
     */
    void    init( void );

    /** Read temperature
     *
     *  @return value of degree Celsius (in float)
     */
    float   read( void );

    /** Read temperature
     *
     *  @return the object returns the read value
     */
    operator float( void );

private:
    I2C     *i2c_p;
    I2C     &i2c;
    char    adr;
};
```


たとえばこのように書きます

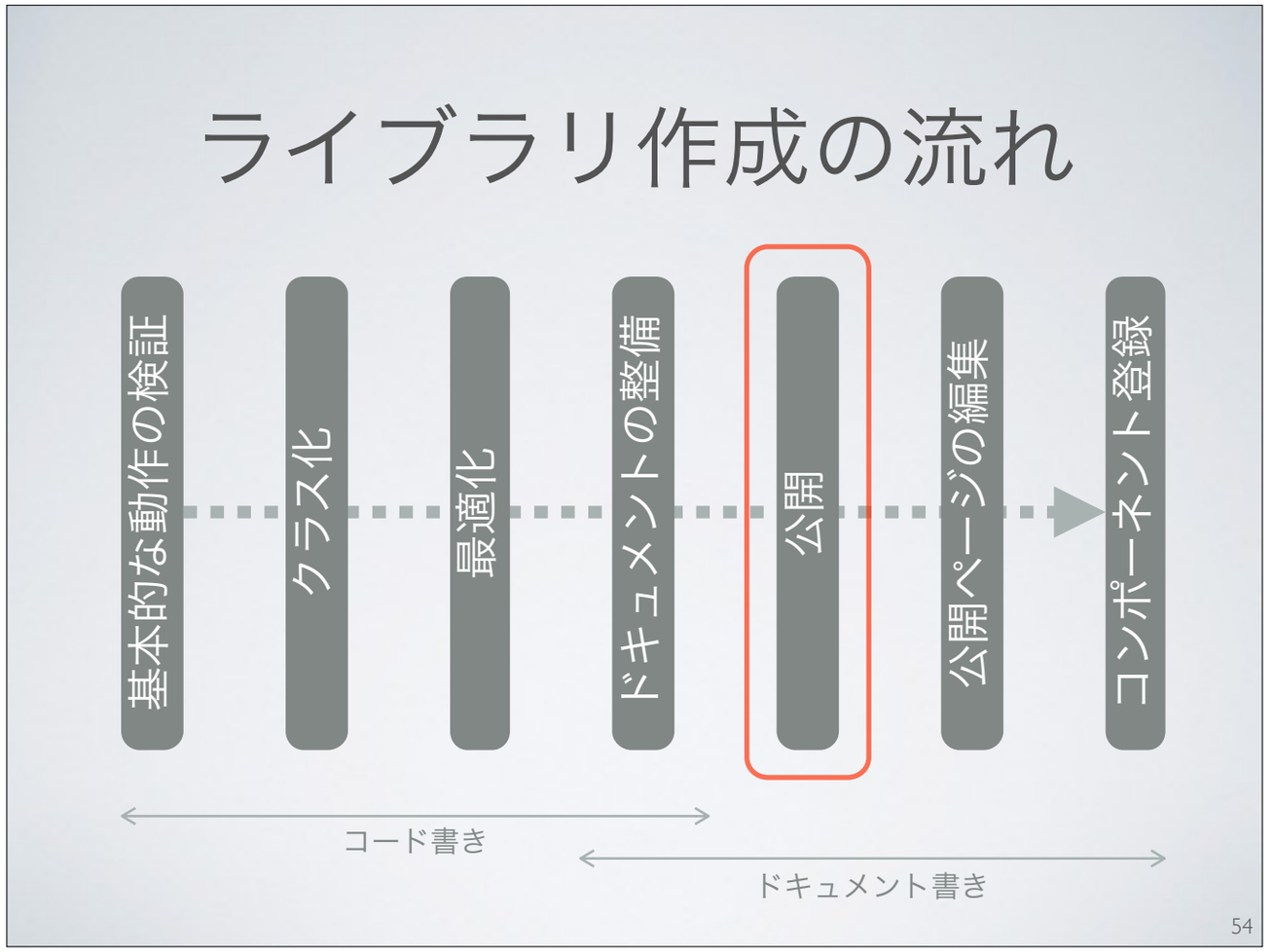
DOXYGENの解説ページ



- https://developer.mbed.org/handbook/API-Documentation

詳しくはこちらのURLをご覧ください

(このライブラリコードについては，まだ改良の余地がある(.hファイル内の#define文の定義をクラス内で扱うなど)のですが，ここではこれで切り上げます)

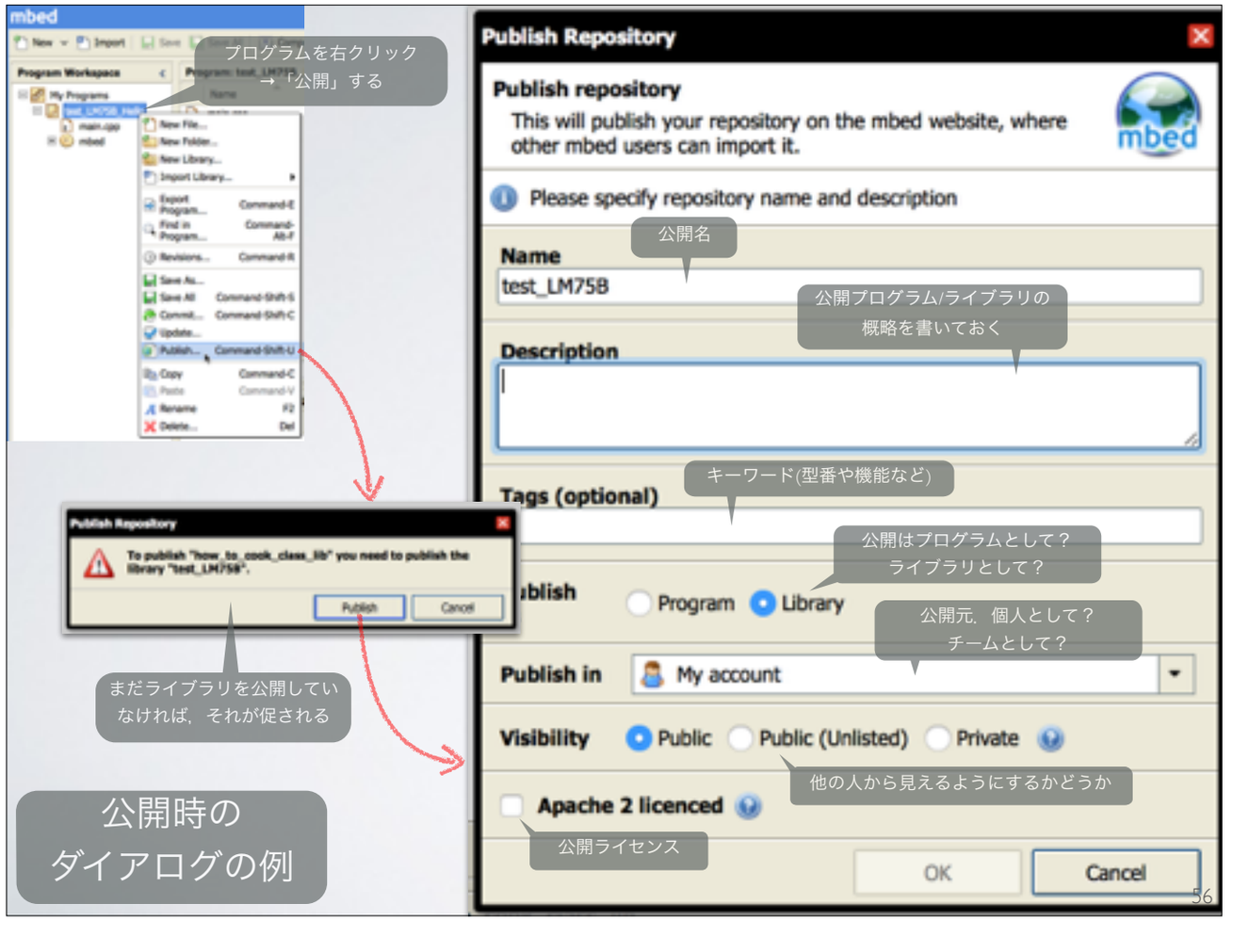

ライブラリ作成の流れ
基本的な動作の検証
クラス化
最適化
ドキュメントの整備
公開
公開ページの編集
コンポーネント登録
コード書き
ドキュメント書き
54

さていよいよ公開です

公開

- ライブラリとプログラムを公開
  - 未公開ライブラリを含むプログラムを公開しようとすると，先にライブラリ公開を促される
  - リポジトリにコミットされてないプログラム/ライブラリを公開しようとすると，先にコミットするよう促される
- 必要事項を入力してOKボタンを押すだけ！

公開はリポジトリにコミットしてあるコードが公開されます.
まだコミットしてないコードを公開しようとすると，コミットを促されます.
また，公開しようとしているプログラムの中に，未公開のライブラリが含まれる場合，先にそのライブラリを公開するようダイアログが開きます



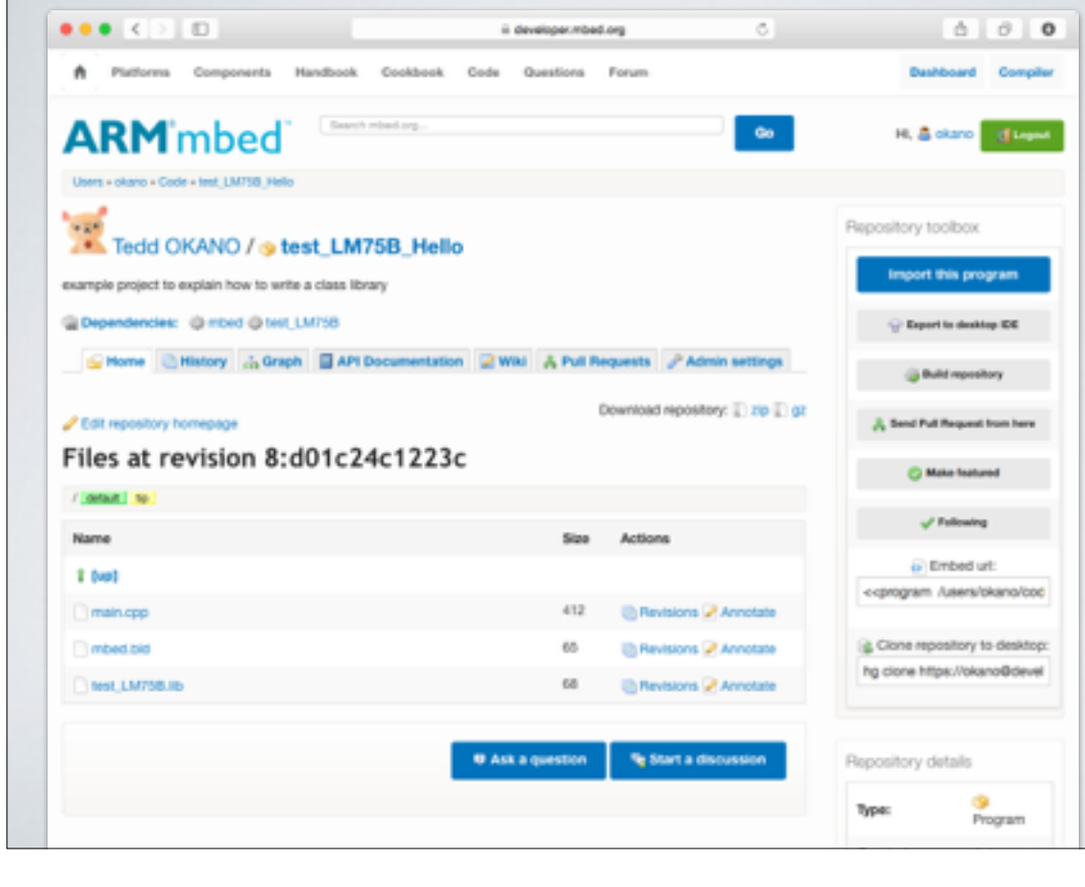
プログラム/ライブラリの公開はこの様なダイアログボックスに必要事項を設定してOKボタンを押すだけ

この中の「Public (Unlisted)」は自分の公開しているコード一覧の中には現れず，URLを知っている人のみがアクセスできるものとなります.

Apache2ライセンスで公開する場合，チェックボックスにチェックを入れるだけでそれを明示できます.

その他のライセンスで公開する場合は，コード内や公開ページにそれを明記しましょう.

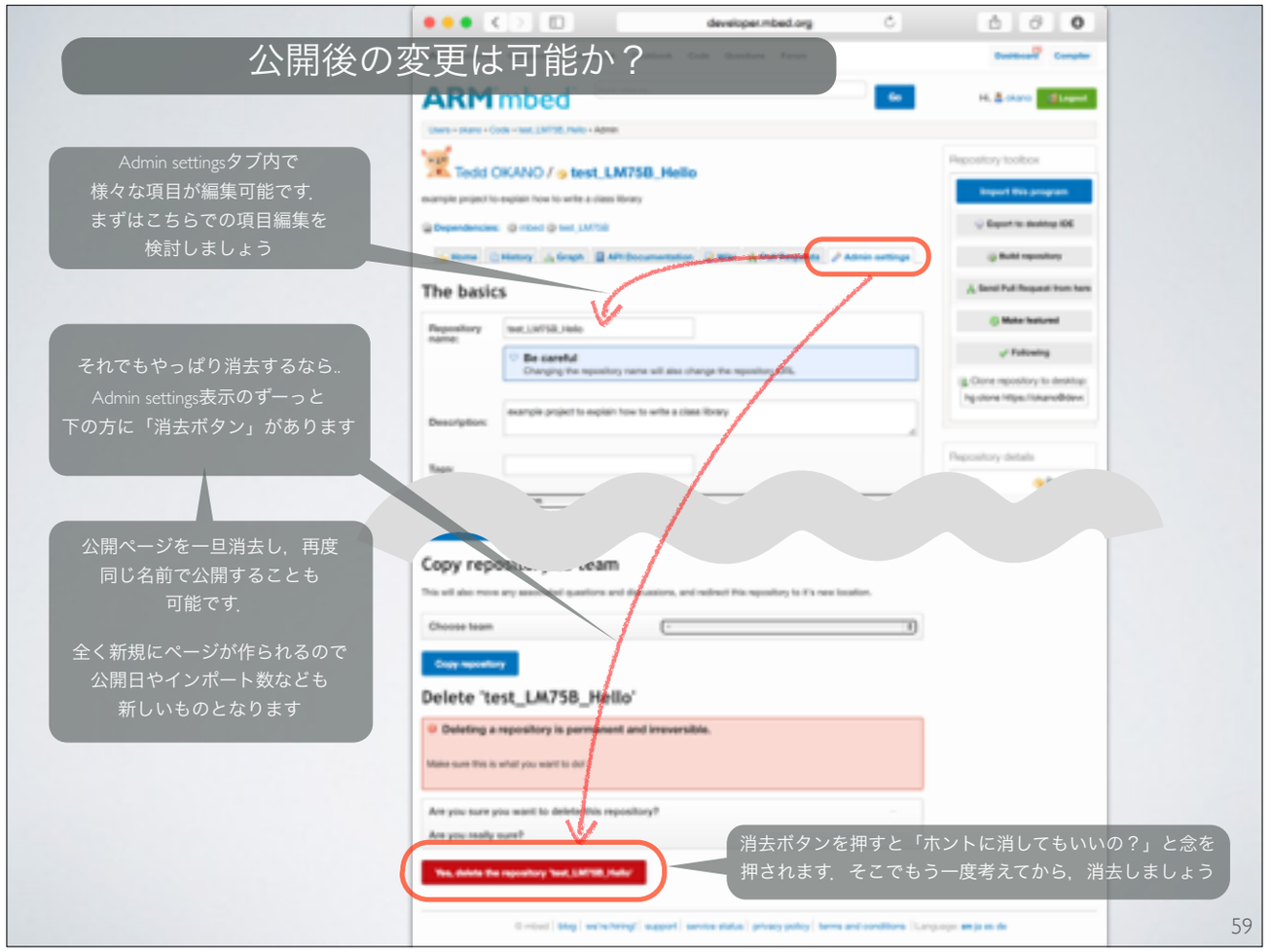
公開するとこんな感じのページができます



ちょっと殺風景

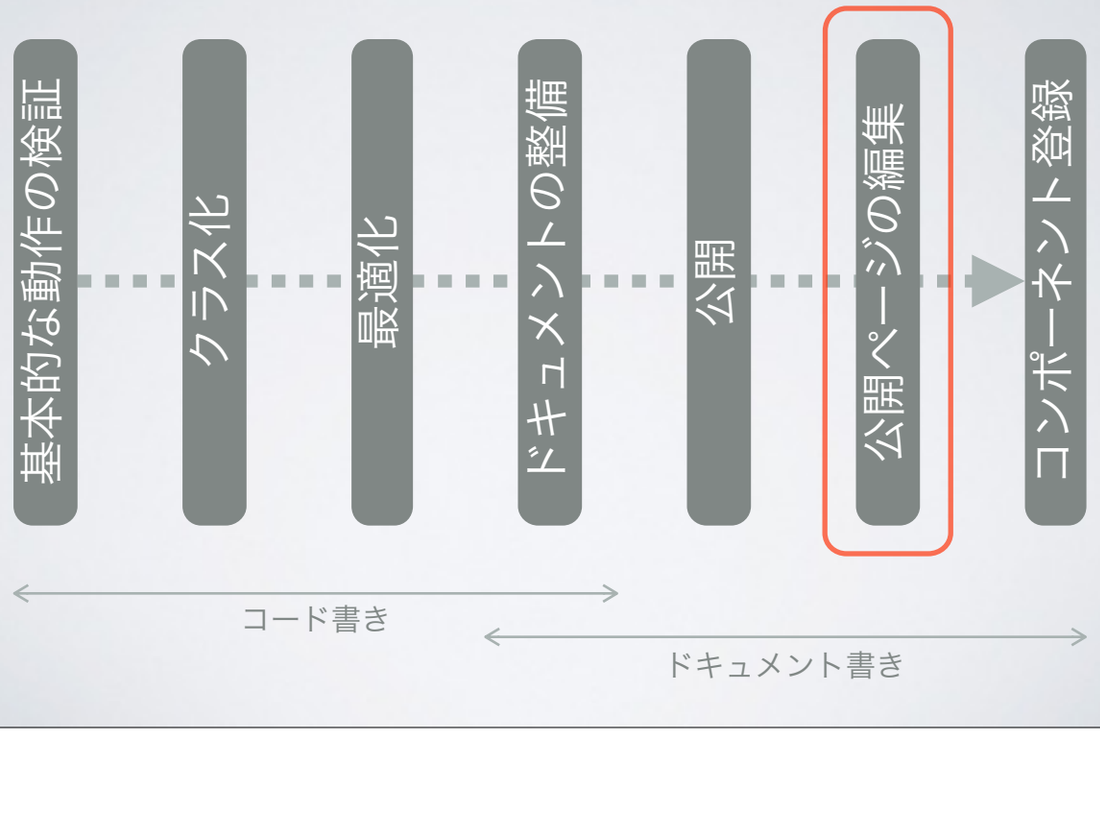
公開ページの例

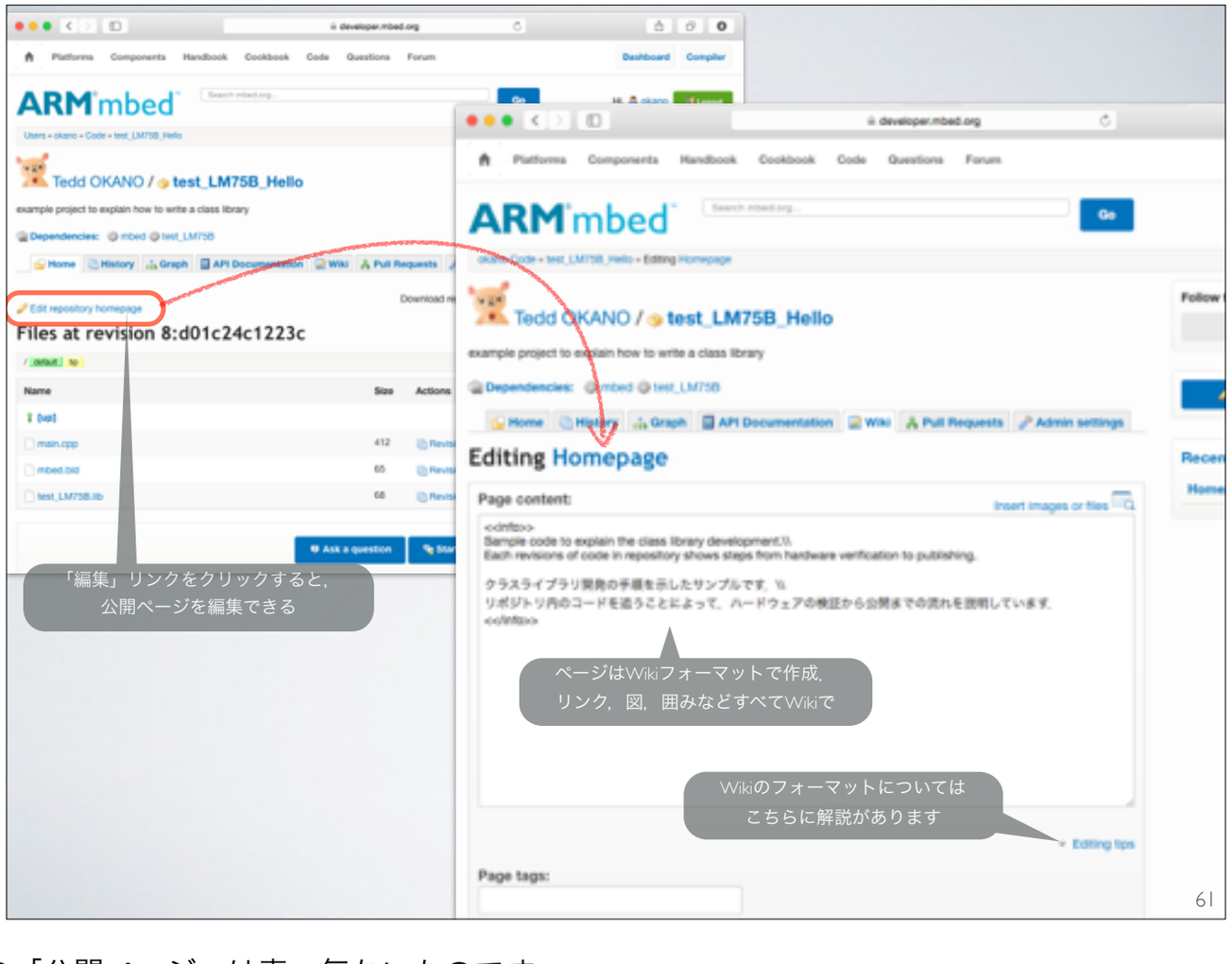
自分がチームに参加していれば，チーム名義でコードを公開することも可能です (誰がコードを公開したかは，たとえば履歴(Historyタブ)を見ればわかります).
公開ページの内容は自由に編集することができます.



自分がチームに参加していれば，チーム名義でコードを公開することも可能です (誰がコードを公開したかは，たとえば履歴(Historyタブ)を見ればわかります).
公開ページの内容は自由に編集することができます.

ライブラリ作成の流れ


基本的な動作の検証
クラス化
最適化
ドキュメントの整備
公開
公開ページの編集
コンポーネント登録
コード書き
ドキュメント書き

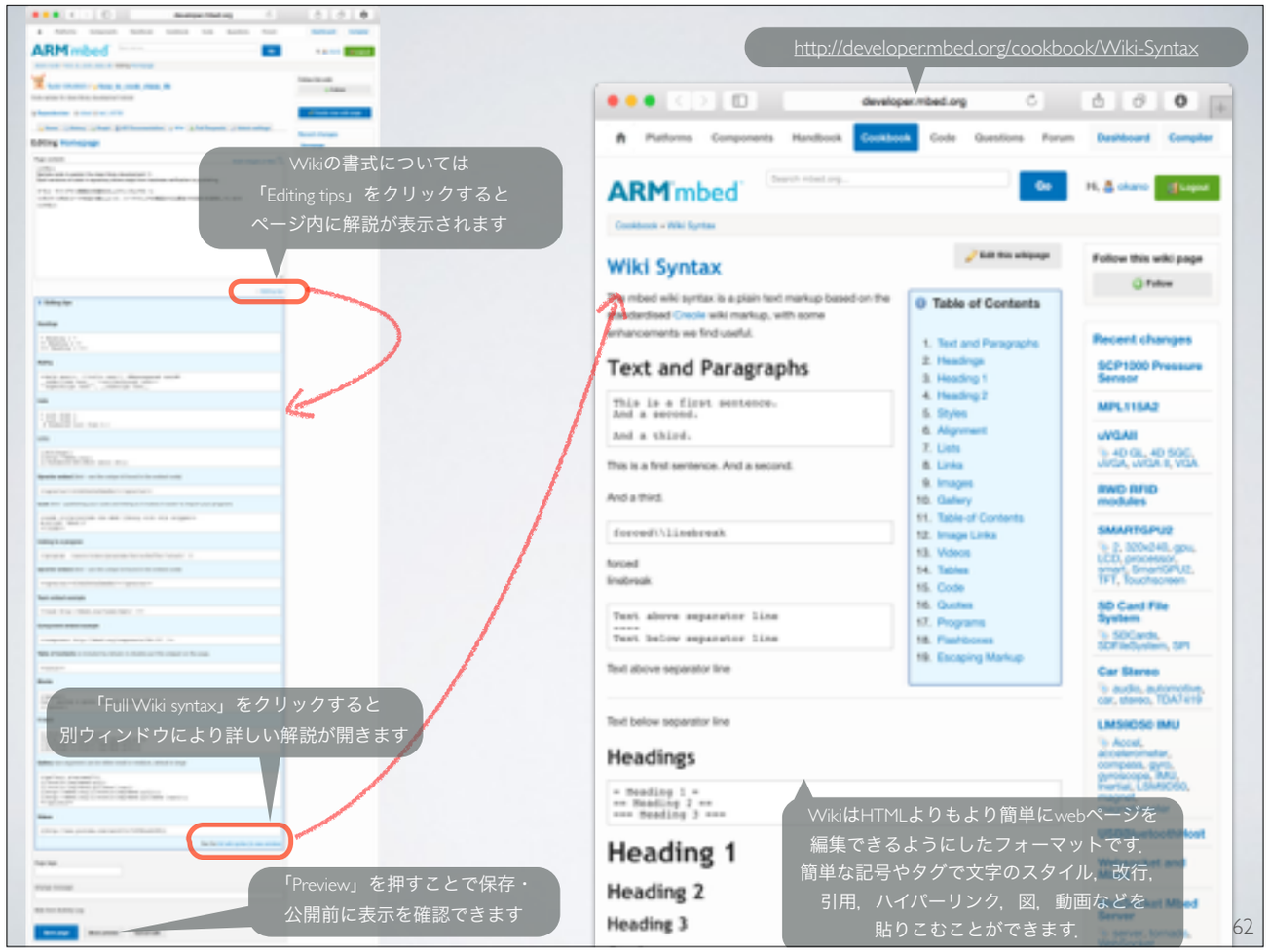


プログラム/ライブラリを公開した直後の「公開ページ」は素っ気ないものです．

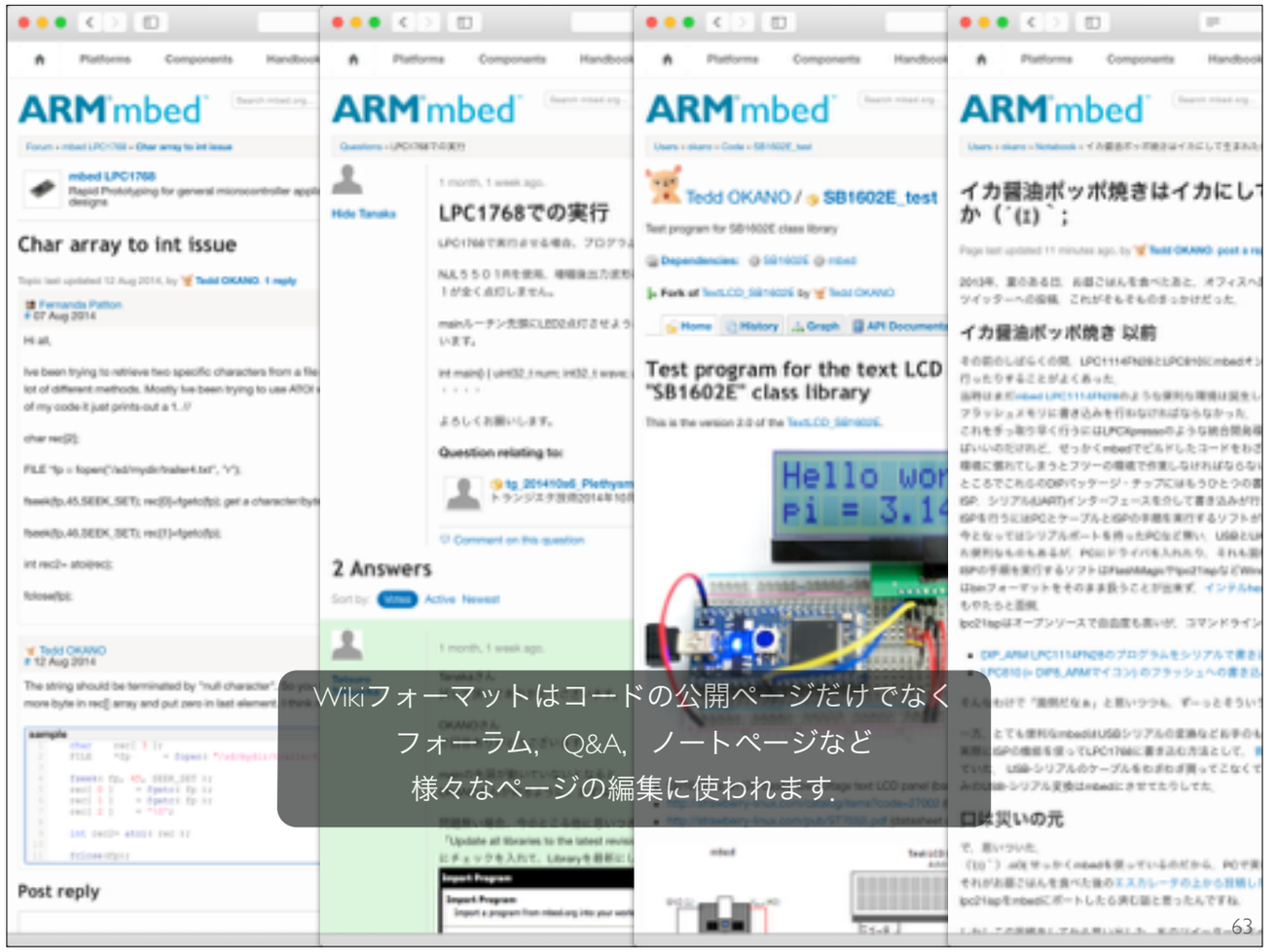
ここに説明などを書いておくと親切です．

公開ページの「Edit repository homepage」リンクをクリックすれば，編集用のページが開きます．

このページにはWikiフォーマットのテキストを入力します．



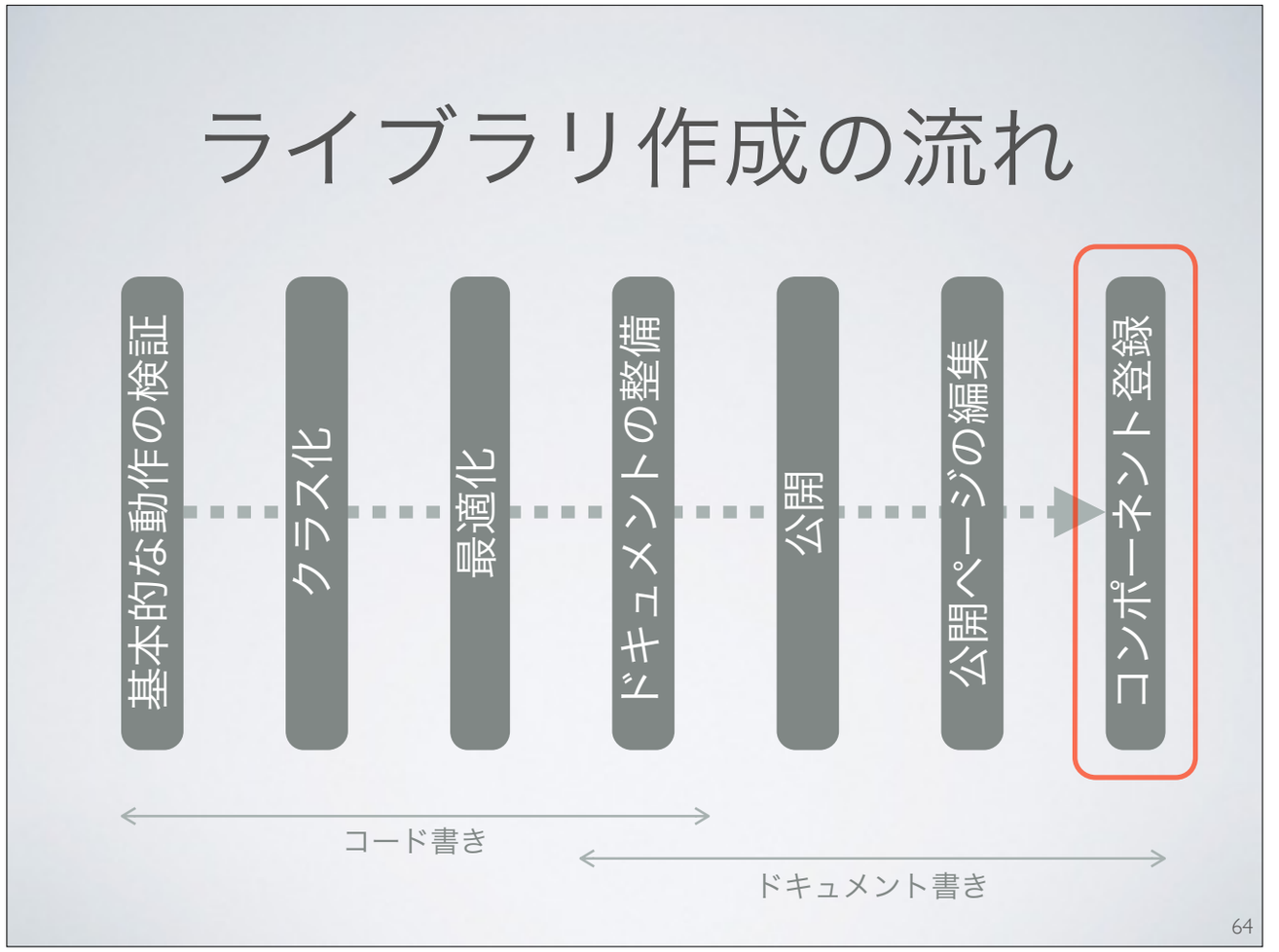
Editing tipsリンクをクリックすると，そのページ内にWikiの書法の簡単な解説が表示されます.
さらにその一番下の「Full Wiki Syntax」をクリックすると，別ウィンドウに全ての書式解説が現れます.
文字の大きさ，改行，リンク，図の張り込み，コードの引用などをどのように書くかを確認できます.



このWikiの書式はmbed.org内で共通です．

フォーラム，Q&A，コード公開ページ，ノートブック.. その他で使えます．

たとえば質問があるときにコードを表示したい場合，Wiki書式を使って楽にその表示を行うことができます．

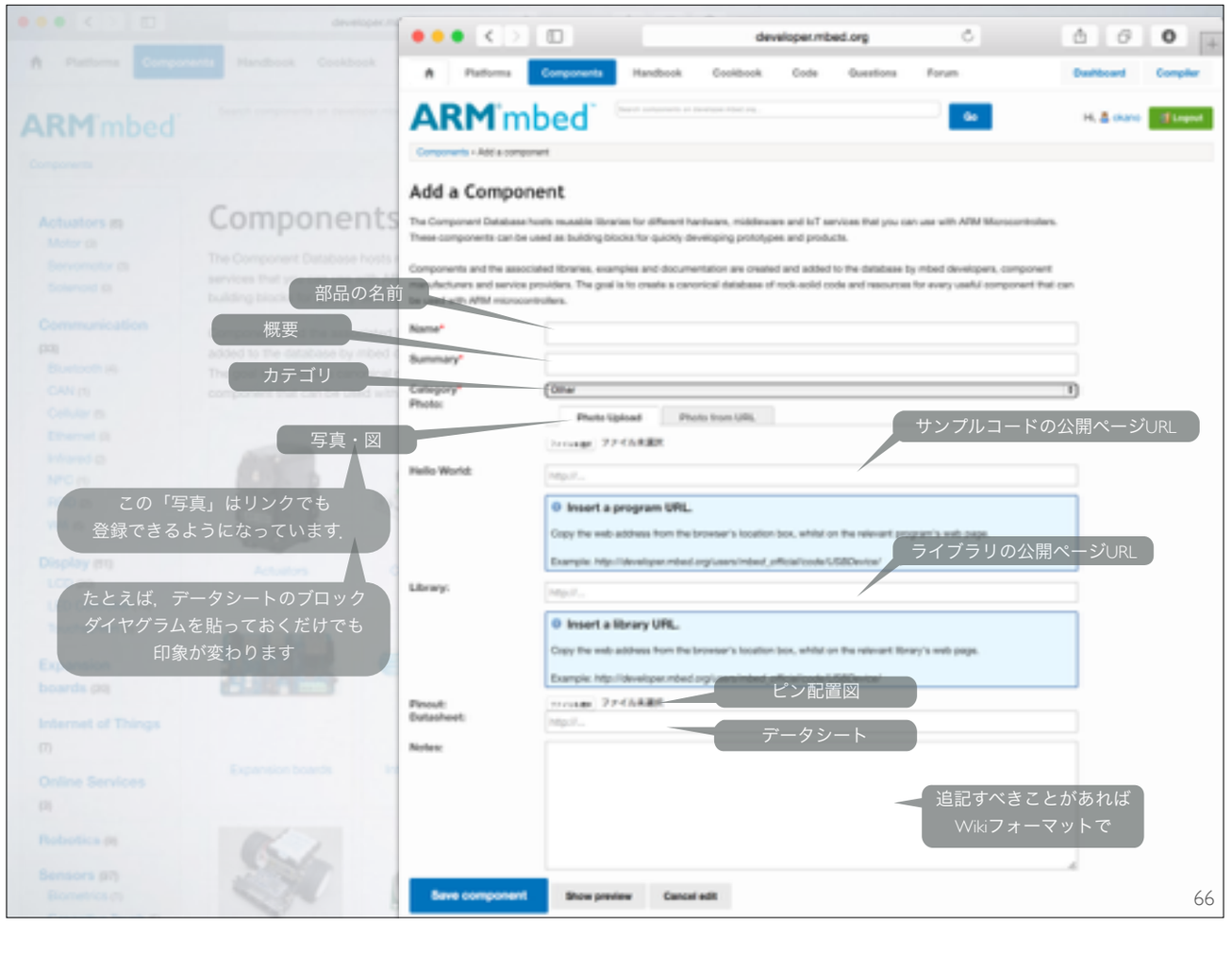


最後のステップ，コンポーネント登録です.

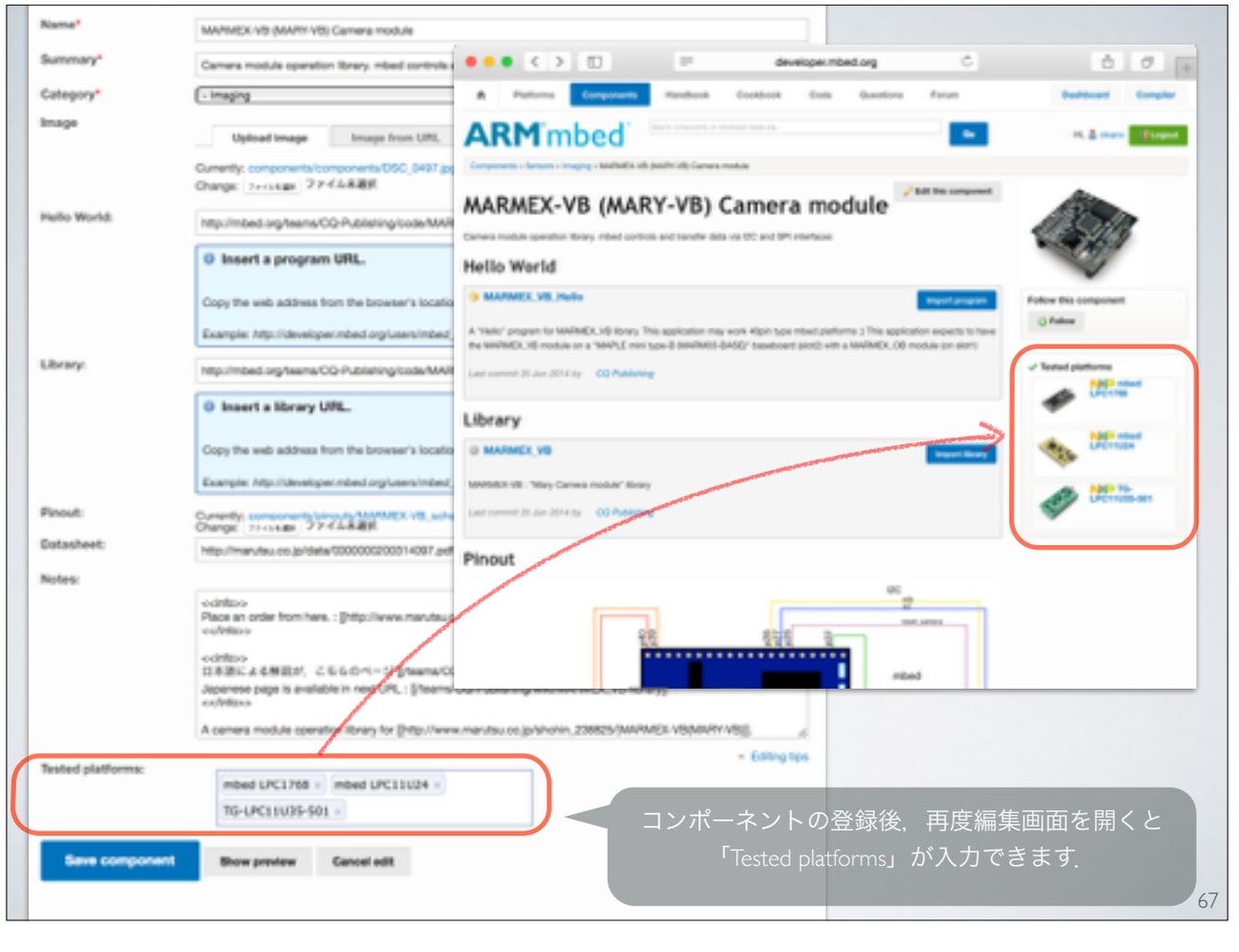
mbedのコンポーネント・ページは比較的新しいページで，以前はCookbookにあったサンプルをより体系的に見れるようにしたページになっています.

mbedサイトでは，気になる部品の型番や通信プロトコルの名前などで検索を行えば，必要なライブラリを見つけることはできますが，ここに登録しておくことでより見つけてもらいやすくできます

カテゴリ別に登録されているので，ジャンル別に見つけやすくなっています.
登録されているライブラリにはコンポーネンツ・ページが与えられ，そこにライブラリ本体，ライブラリを試してみるサンプルプログラム(HelloWorldプログラム)，データシートやピン配置などの情報がまとめられます.
コンポーネンツ・ページで「Add a component」ボタンを押すと，登録ページが現れます



登録ページには必要な項目の入力欄が現れるので，必要事項を書いていけば，それでそのまま登録できます.

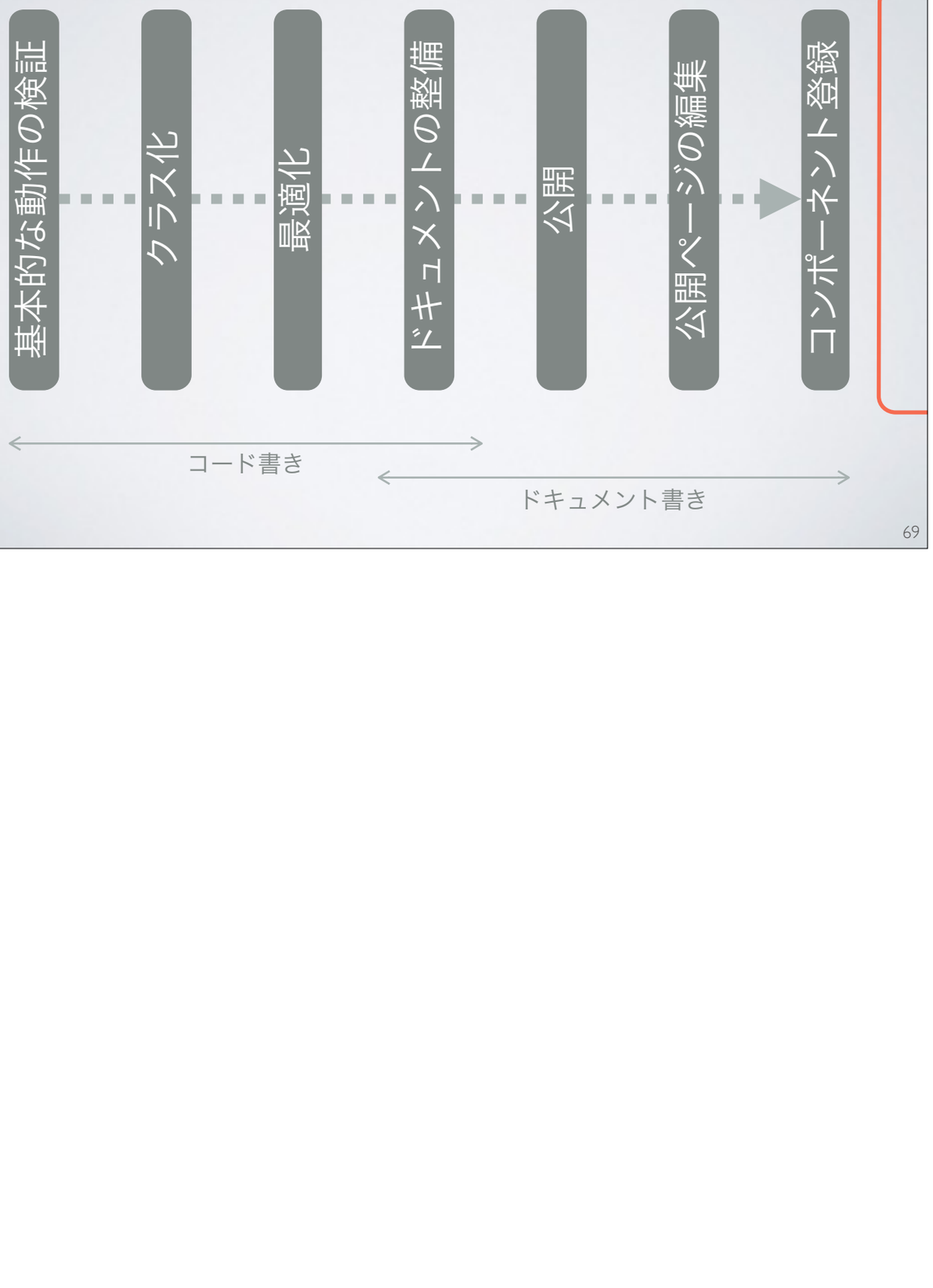


コンポーネントの登録後，再度編集画面を開くと「Tested platforms」が入力できます.

実際に動作を確認したmbedプラットフォームを書いておくことができます.

コンポーネント・ページは関連情報への「ハブ」ということができるでしょう

ライブラリ作成の流れ



ライブラリを公開・登録したら

公開後

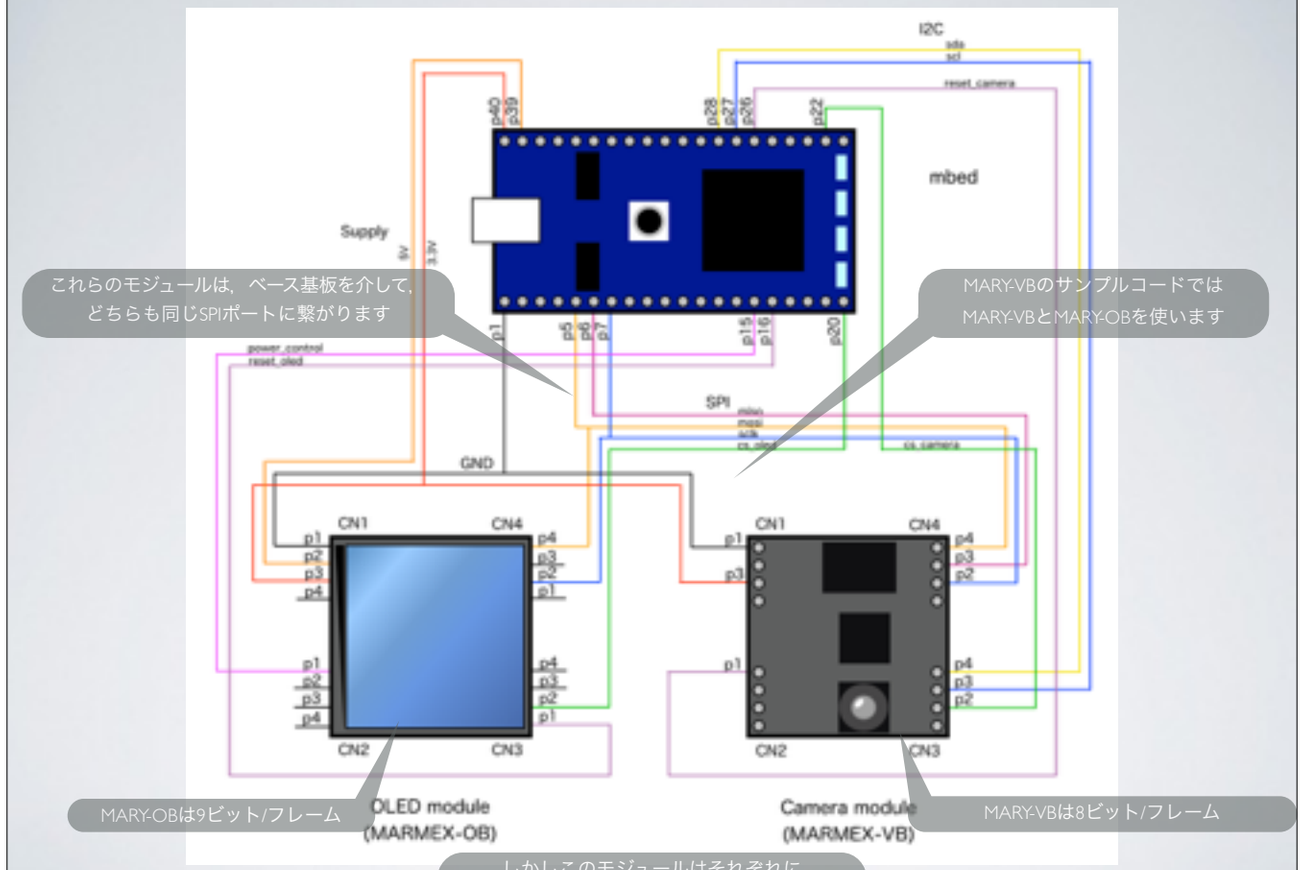
- 特にすることはありませんが..
  - 必要があればアップデート
    - バグフィクス
    - 機能追加
    - 対応プラットフォーム追加
    - ユーザからのプルリクエスト
    - 同じ部品用のライブラリが，フォークして公開されている場合にはPullリクエストを出してもらいましょう
- (できれば)インポート数が上がるように，公開ページの改良・宣伝活動など
  - 使ってもらえればそれだけフィードバックも多くなります
  - 多くの改善のアイデアで，よりよいライブラリに

この後は特にすることはありません．強いて言うとすると..

参考 (I)

- クラス内でのインターフェース・インスタンスの保持について
- MARY-VBを例に

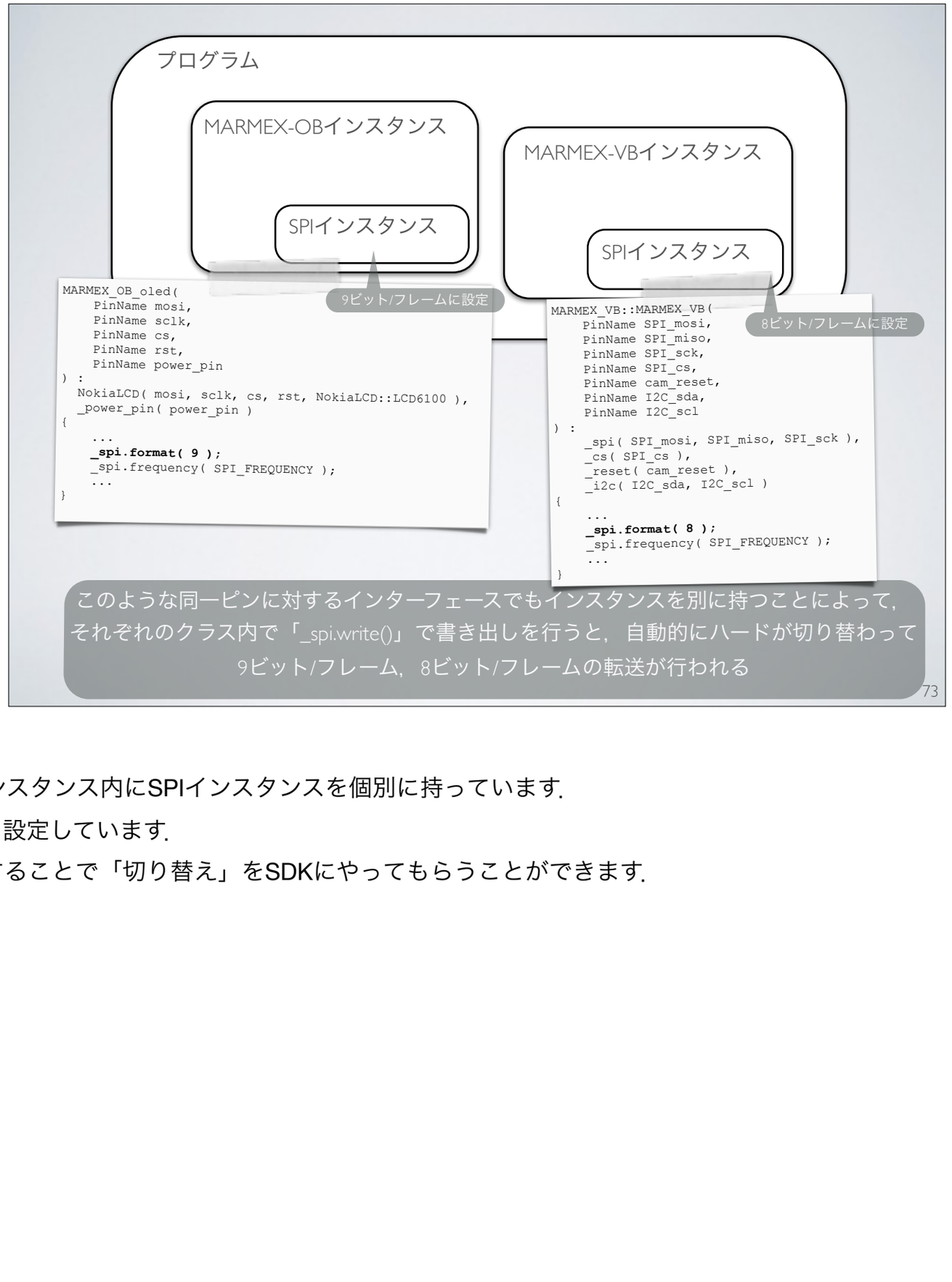
インターフェースのインスタンスを，ライブラリ内に持たせることについて



MARMEX-VBのサンプルコードではカメラとOLEDの両方のモジュールが，同一SPIバスに接続されています.

データ転送はカメラでは8ビット単位，OLEDでは9ビット単位で行われます.

両方と通信を行う場合は，この切替を行う仕組みが必要になります.

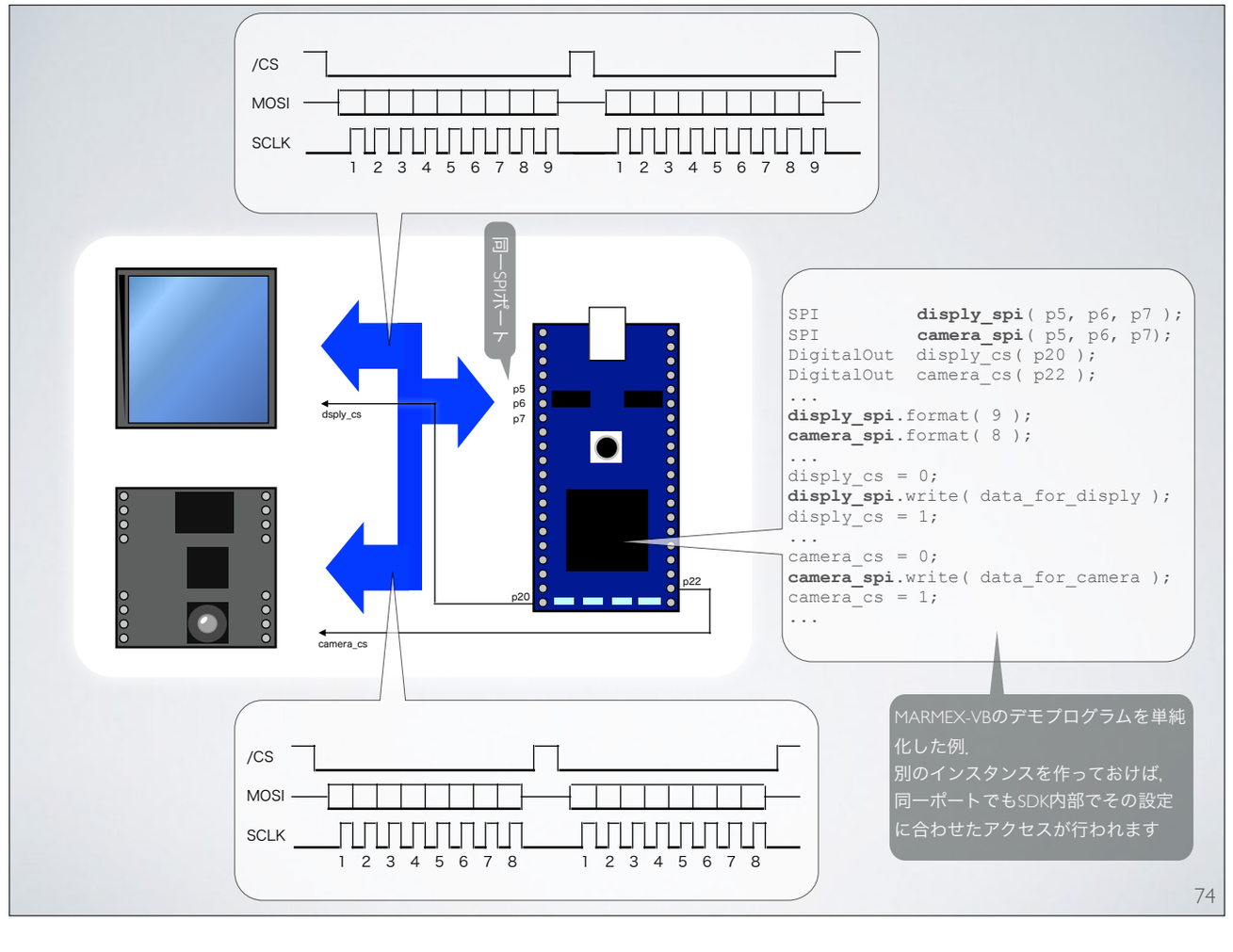


実際の例を見てみましょう.

カメラとOLEDには各(モジュールの)インスタンス内にSPIインスタンスを個別に持っています.

そしてそれぞれに転送単位のビット数を設定しています.

このようなインスタンスの持たせ方をすることで「切り替え」をSDKにやってもらうことができます.



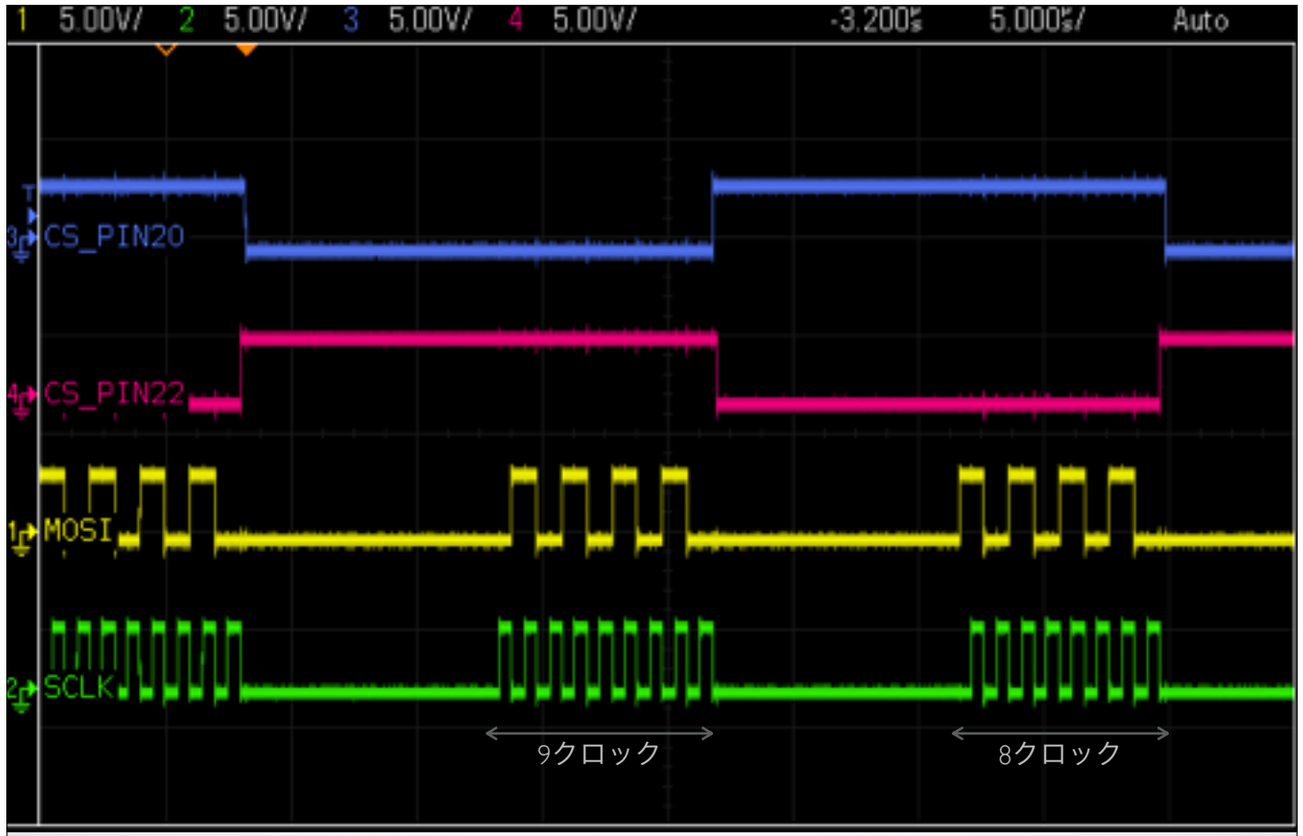

```
SPI          disply_spi( p5, p6, p7 );
SPI          camera_spi( p5, p6, p7);
DigitalOut   disply_cs( p20 );
DigitalOut   camera_cs( p22 );
...
disply_spi.format( 9 );
camera_spi.format( 8 );
...
disply_cs = 0;
disply_spi.write( data_for_disply );
disply_cs = 1;
...
camera_cs = 0;
camera_spi.write( data_for_camera );
camera_cs = 1;
...
```

MARMEX-VBのデモプログラムを単純化した例.
別のインスタンスを作っておけば,
同一ポートでもSDK内部でその設定に合わせたアクセスが行われます

先のコードを単純化した例です.

同一SPIを2つのインスタンスで定義しておいて, それぞれに別の設定.

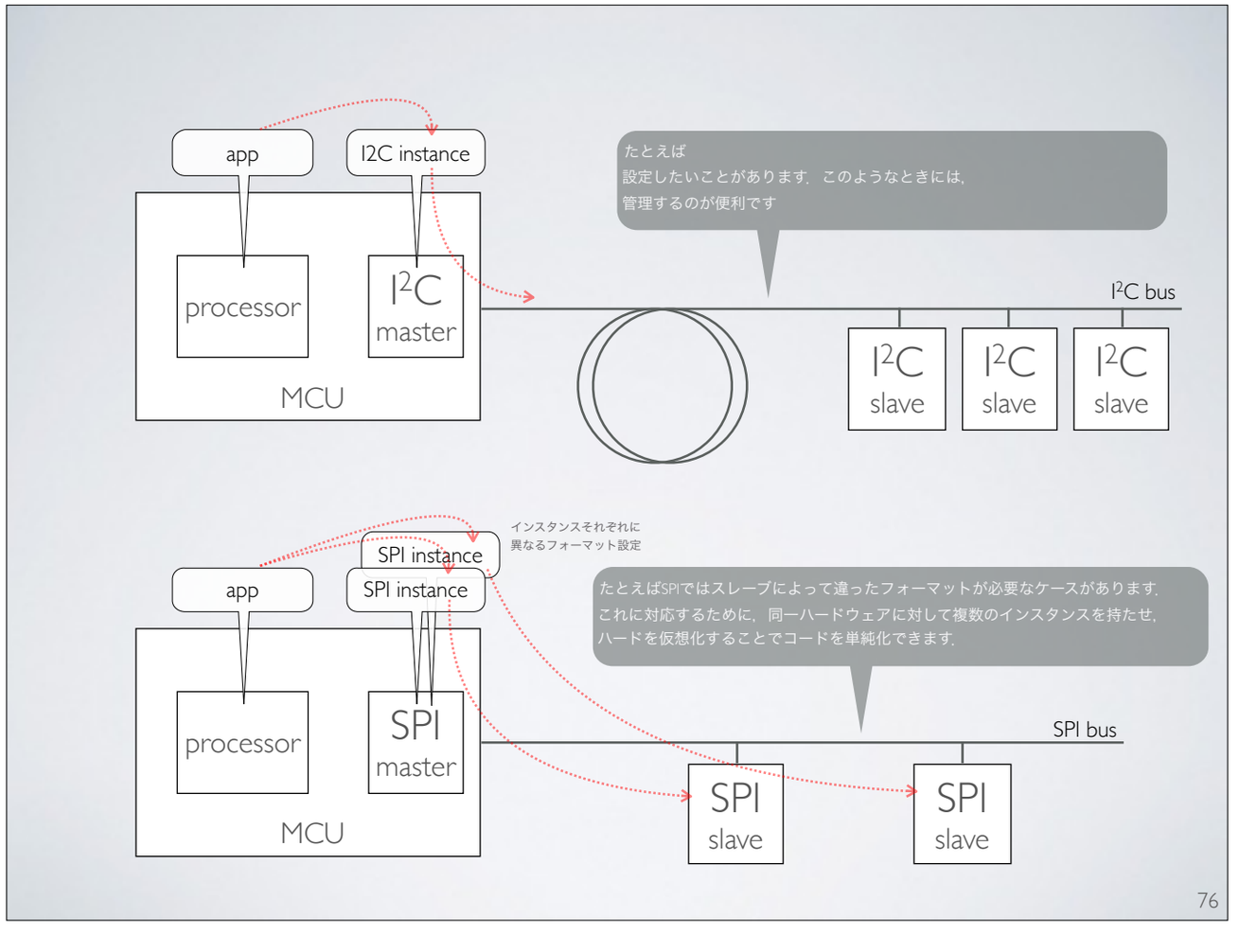
そのインスタンスのwrite関数を呼ぶと, 設定に応じた転送が行われます.



前ページのコードを実行してみた様子

http://developer.mbed.org/users/okano/code/multiple_instance_for_SPI/

前ページのコードを実行してみた様子



ハードウェアとインスタンスの関係は1対1でなくても構いません．

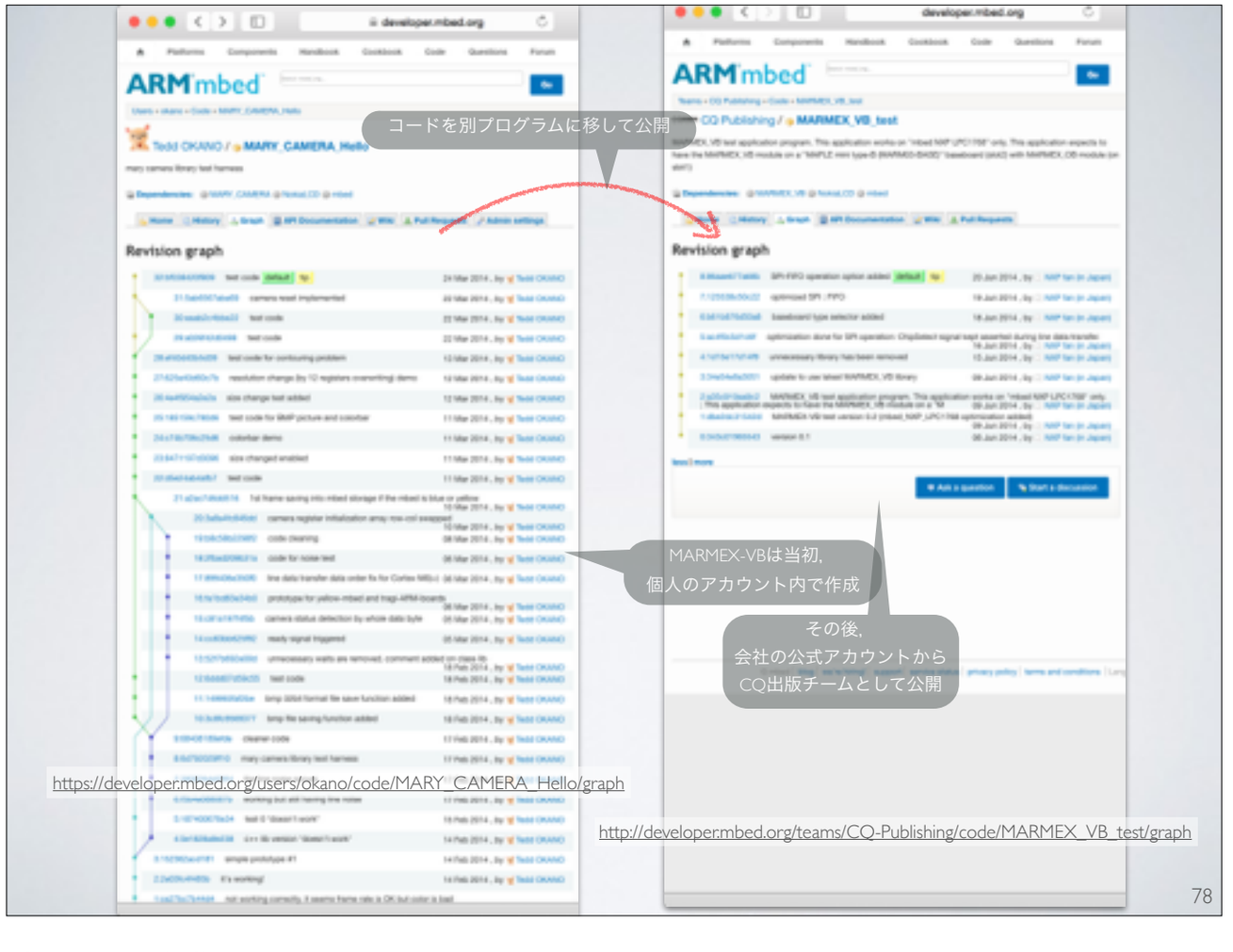
ひとつのハードウェア資源を集中的に管理する場合は1個のインスタンスで操作するほうが便利でしょうし，

多義的に使う場合には複数のインスタンスを用いることでコードを単純化できます．

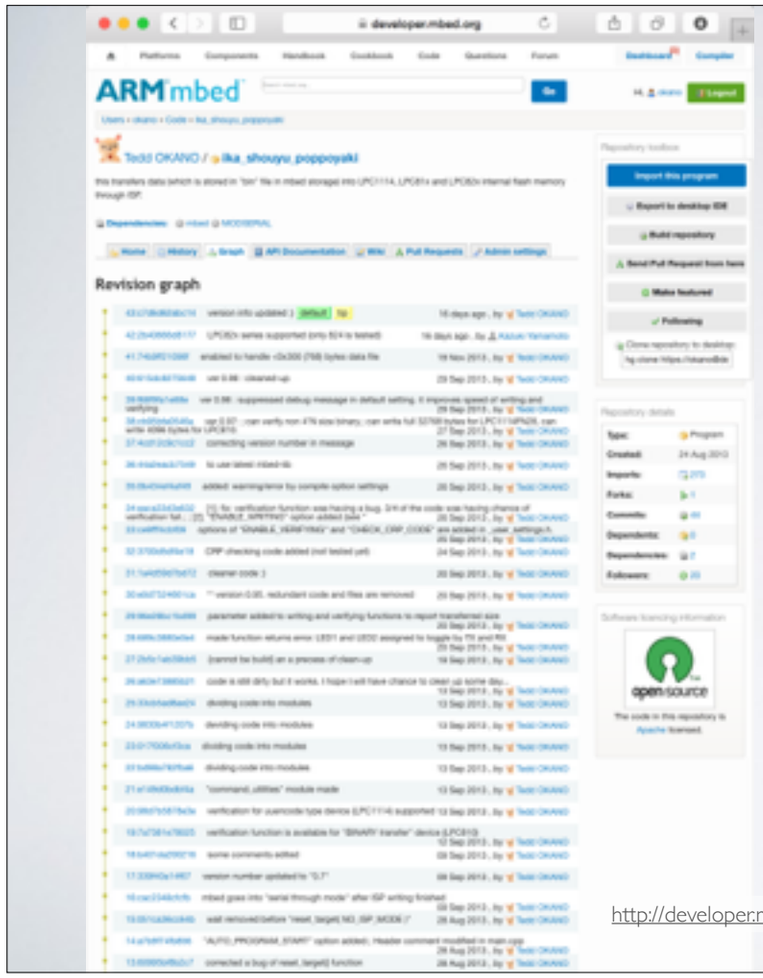
参考 (2)

- 公開用リポジトリ
  - 開発用と分けたほうがいいかもしれません
- 公開すると，全ての履歴(コミット)が公開されます
  - 後から公開バージョンを追いにくい
  - 公開バージョンはこのうちのいくつかだけなのに「バグ付きのまま公開して更新ばかりしている」なんて揶揄されます
- 逆に開発ステップを含めて公開したい場合には分ける必要は無いでしょう

プログラム/ライブラリを公開すると，公開時点までのリポジトリが丸ごと公開されます.
細かいコミットを行っていると，その履歴が全部見えてしまうことになります.



なので開発用のプログラムと，公開用のプログラムを分けたほうが良いかもしれません.



- こちらは開発用リポジトリをそのまま公開してしまった例
- 全てのコード変更→コミットのステップが見えてしまっており各公開バージョンがどれだったかが判らなくなっています
- 細かいコミットも表示されてしまっており，後からこれを見た人に「まともに検証すら行ってないコードを公開・訂正を繰り返すとはナニゴトか！」と怒られたことも(^^;
- その一方で，ISPプログラムを独自に開発されている方から「シリアル上のプロトコルを確認するのに，履歴の初めから追うことによってよく理解出来ました」との声も

http://developer.mbed.org/users/okano/code/ika_shouyu_poppoyaki/graph

こちらはプログラムを分けずに公開した例

参考 (3)

- どのプラットフォームでテストしておくべきか
  - mbed LPC1768 (mbedのリファレンス)
  - あとは好みで (^^;
  - トラ技ARMライタも！

どの「プラットフォームに対応しておくべきか」は悩みどころです.
基本的には「青mbed (mbed LPC1768)」には対応しておくべきと考えます.
その他のプラットフォームについては，適宜その用途に合わせた選択をするのが良いでしょう.

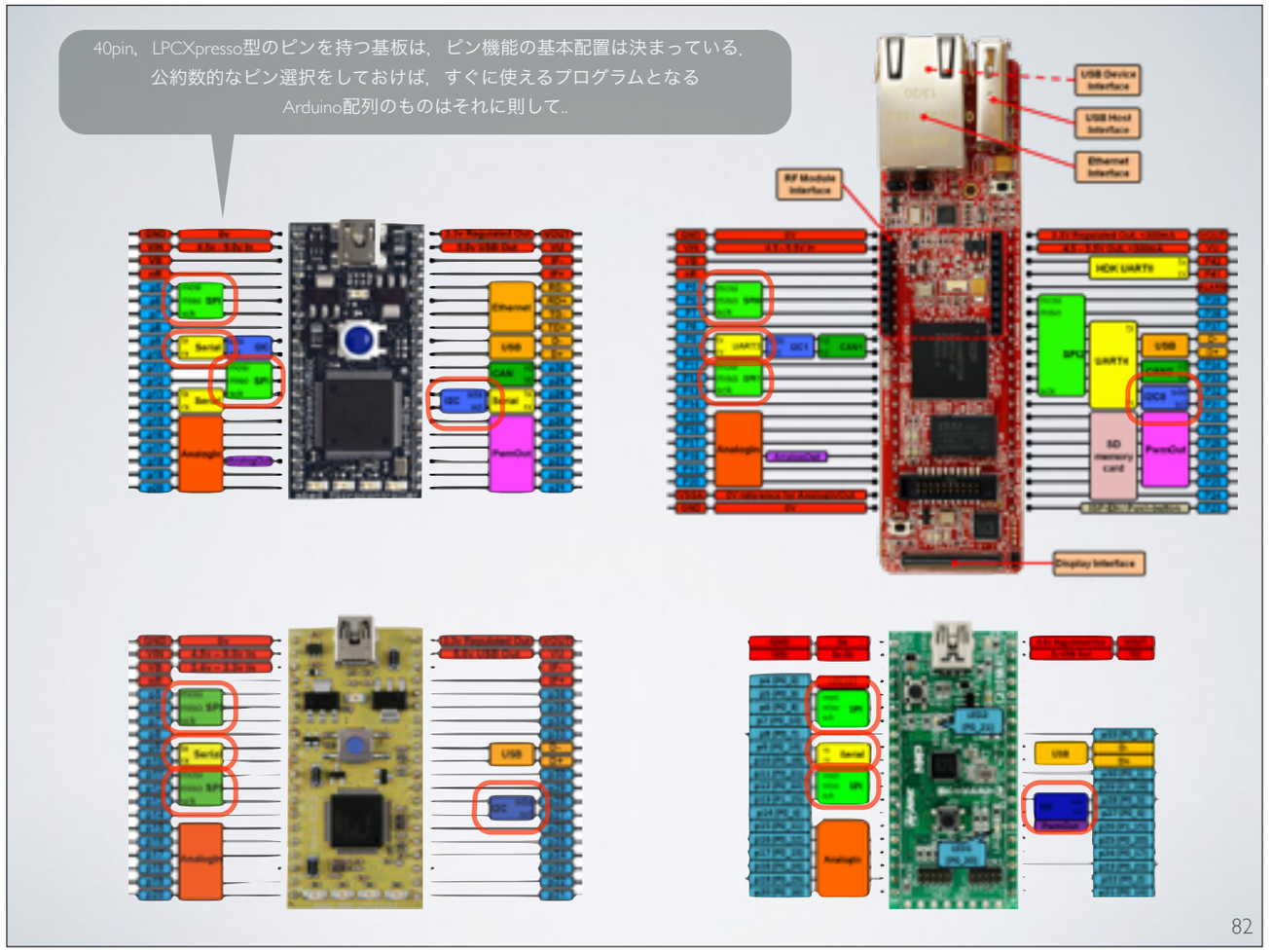
参考 (4)

- サンプルコードではどのピンを使うか？

どの「プラットフォームに対応しておくべきか」は悩みどころです.

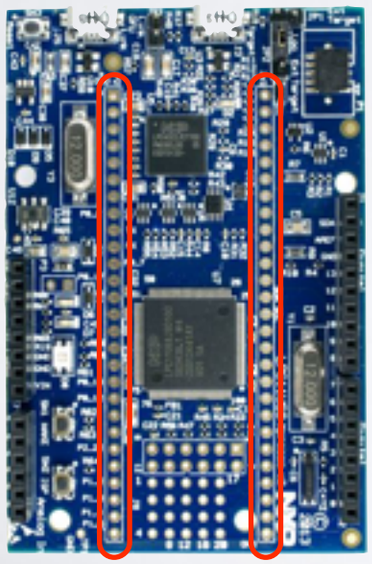
基本的には「青mbed (mbed LPC1768)」には対応しておくべきと考えます.

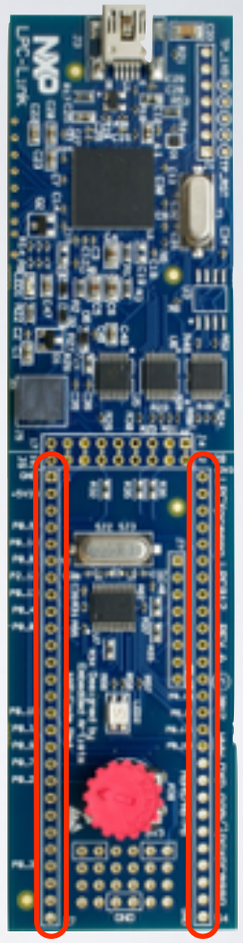
その他のプラットフォームについては，適宜その用途に合わせた選択をするのが良いでしょう.



40pin，LPCXpresso型のピンを持つ基板は，ピン機能の基本配置は決まっている．
公約数的なピン選択をしておけば，すぐに使えるプログラムとなる
Arduino配列のものはそれに則して..

ちなみに..
LPCXpressoシリーズのピンヘッダ用端子も最大限,
青mbedに合わせて配置されています





各種LPCXpressoのピン配置も，青mbedをベースにしています.

参考 (5)

- (MCUの)レジスタレベルの最適化
- サンプルコードに使うピンを，ターゲットによって切り替える

コードを書いていると，どうしても「最適化の誘惑」から逃れられません.
部品が本来持っている性能を引き出したくなります.
mbedではMCUのレジスタレベルのアクセスも可能です．でもMCUを抽象化してくれているmbed-SDKをバイパスしてしまうため，他のMCUでの互換性が犠牲になります.

```
#if ( LINE_READ_OPT == USING_SSP_FIFO )

    #define FIFO_DEPTH  4

    #if defined( SSP_AUTO_SELECTION )
        #if defined( TARGET_MBED_LPC1768 )
            #define SPI_PORT_SELECTOR   LPC_SSP1
        #elif defined( TARGET_LPC11U35_501 ) || defined( TARGET_LPC11U24_401 )
            #define SPI_PORT_SELECTOR   LPC_SSP0
        #endif
    #elif defined( SSP_USE_SSP0 )
        #define SPI_PORT_SELECTOR   LPC_SSP0
    #elif defined( SSP_USE_SSP1 )
        #define SPI_PORT_SELECTOR   LPC_SSP1
    #else
        #error when using FIFO option for the optimization, choose one of definition SSP_USE_SSP0 ..
   #endif   //  #if defined( SSP_AUTO_SELECTION )

    char    reg = COMMAND_READ | CAMERA_DATA_REGISTER | COMMAND_ADDR_INCREMENT;
    int     n;

    if ( _read_order_change ) {

        _cs = 0;

        for(n = FIFO_DEPTH; n > 0; n--) {
            SPI_PORT_SELECTOR->DR = reg;
        }

        do {
            while (!(SPI_PORT_SELECTOR->SR & 0x4));
            *p  = (SPI_PORT_SELECTOR->DR & 0xFF);

            if (n++ < (n_of_pixels << 1) - FIFO_DEPTH)
                SPI_PORT_SELECTOR->DR = reg;

            while (!(SPI_PORT_SELECTOR->SR & 0x4));
            *p++    |= (SPI_PORT_SELECTOR->DR << 8);

            if (n++ < (n_of_pixels << 1) - FIFO_DEPTH)
                SPI_PORT_SELECTOR->DR = reg;

        } while(n < (n_of_pixels << 1));

        _cs = 1;
```

ターゲットによってコードを切り替える

CMSIS定義のレジスタにアクセス
(アドレス直叩きも可能ですが，お行儀としては
こっちのほうがマシでしょう)

http://developer.mbed.org/teams/CQ-Publishing/code/MARMEX_VB/file/84e6c89a9a6d/MARMEX_VB.cpp

互換性を諦めずにこれを行うには..

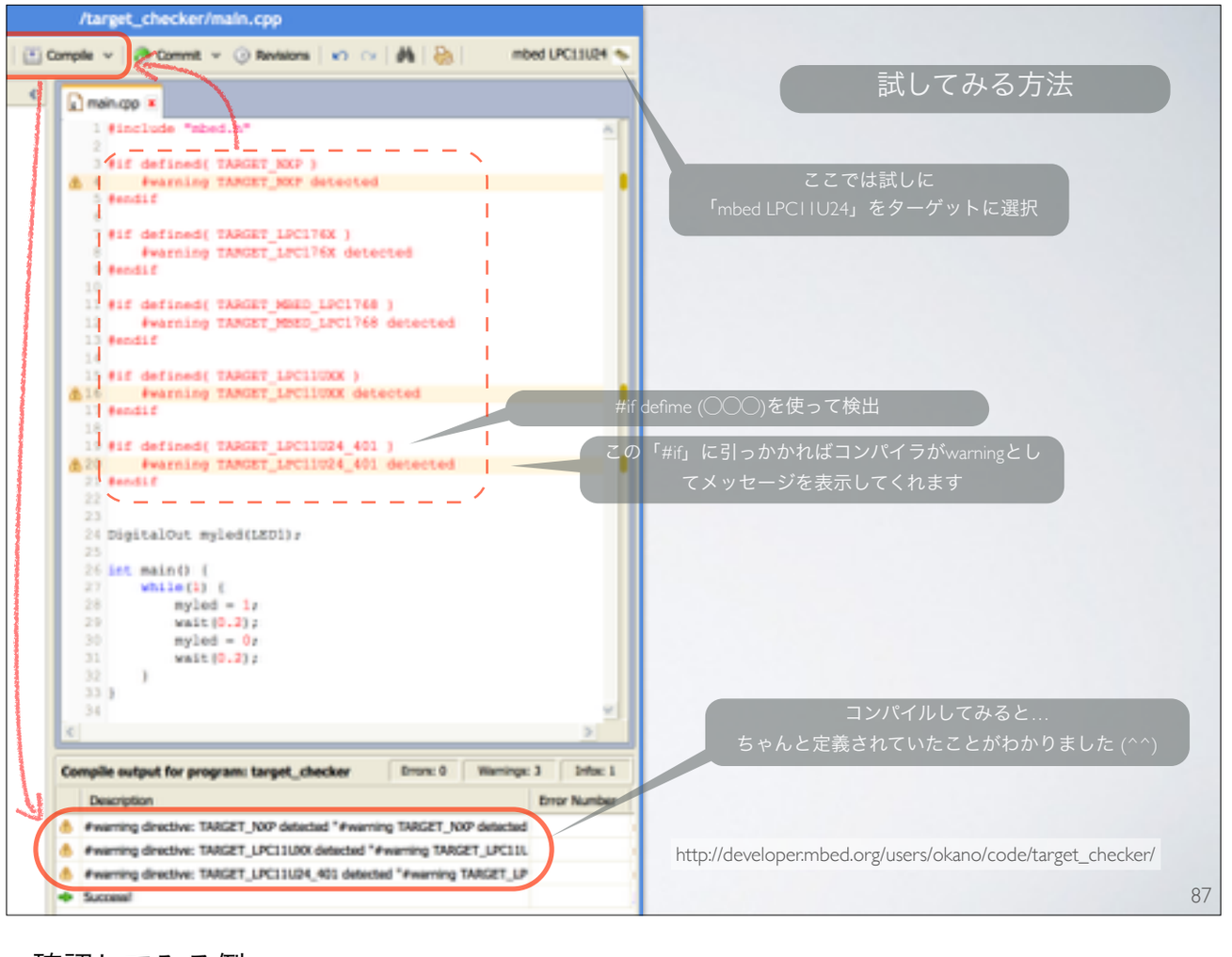
mbedのターゲットで条件コンパイルする方法があります.

http://developer.mbed.org/users/mbed_official/code/mbed-src/file/17d2d502961d/targets/hal

ターゲットを識別する定義済みのシンボル名は
mbed-srcのコード中のフォルダ名「TARGET_○○」
が使えます

たとえば
NXPのMCUであるかどうかを識別したいなら
「TARGET_NXP」
LPC11Uxxシリーズであることを識別するなら
「TARGET_LPC11UXX」
LPC11U24/401であることを認識するなら
「TARGET_LPC11U24_401」
のようなシンボル名を使えます

条件コンパイルのターゲット名はmbed-SDKのソースを見ることで確認できます.



シンボル名が定義されているかどうかを，確認してみる例.

#if define() と #warning を組み合わせることで，コンパイルしてみるだけで結果が出ます.

参考 (6)

- 協業しましょう
  - プルリクエストを出しましょう
  - 他人様のコードを改良したら→プルリクエスト
  - オリジナルのリポジトリにマージしてもらう

他の人が公開しているコードを改良したら..

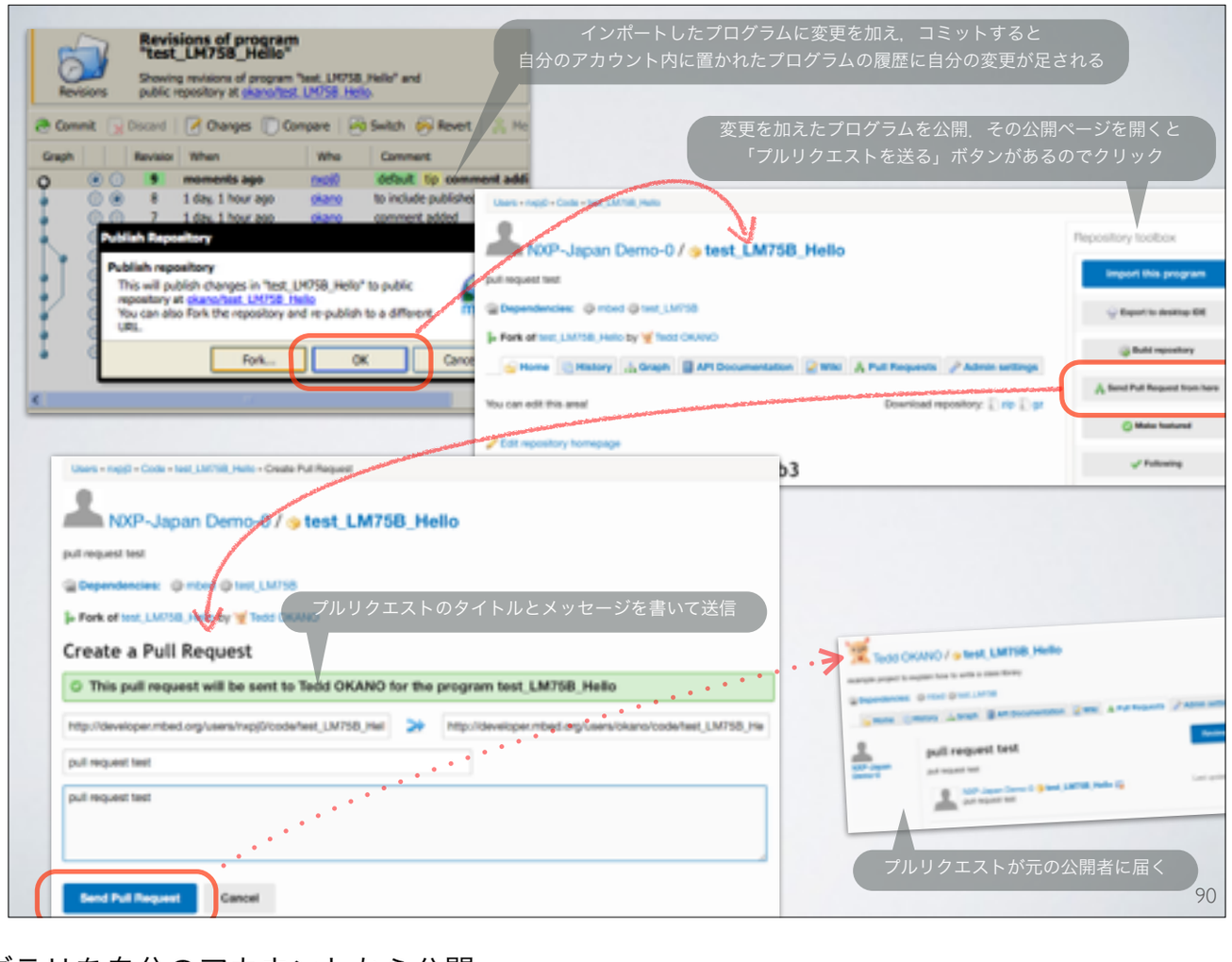
そのコードを知らせてあげましょう！

せっかくの共同作業が出来る環境．使わないともったいないです.

http://developer.mbed.org/users/okano/code/ika_shouyu_poppoyaki/graph

Yamamotoさん，ありがとうございました！ \(^^)/

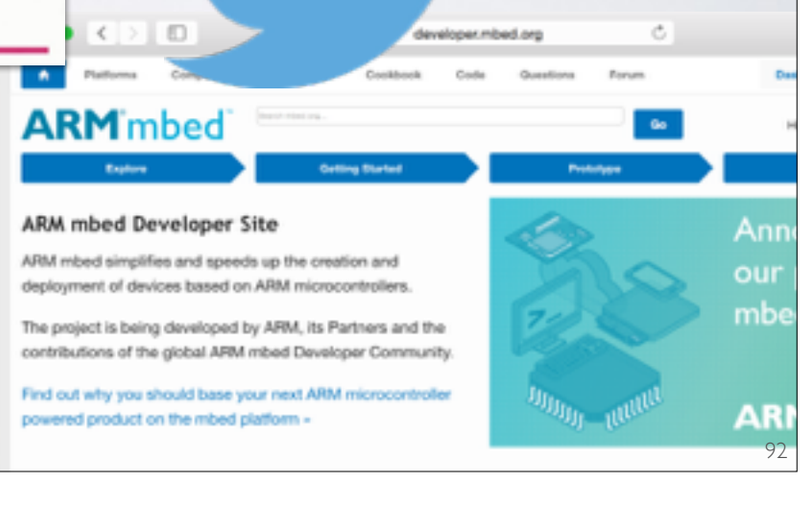
プルリクエストをいただくと，履歴に作者の名前が残ります.



インポート→改良したプログラム/ライブラリを自分のアカウントから公開.
公開ページから「プルリクエスト」を送信.

プルリクエストを受け取ると，コードの内容を確認してテスト.
問題なければコミットして公開します.

参考 (7)



- 情報源

勉強会参加者に教えて頂いたんですが，
この本だったのに，ひょっとすると
絶版かもしれません．残念 (>_<;

参考になる書籍やmbed.orgサイトを参照しましょう.

どうしても行き詰まったら，mbed.org内のフォーラムやTwitterが役に立つかも？

参考 (8)

- どのようにしていいか，わからないとき
- →優れた例を探してみる

https://developer.mbed.org/users/shintamainjp/

mbed.orgはサンプルの宝庫です.

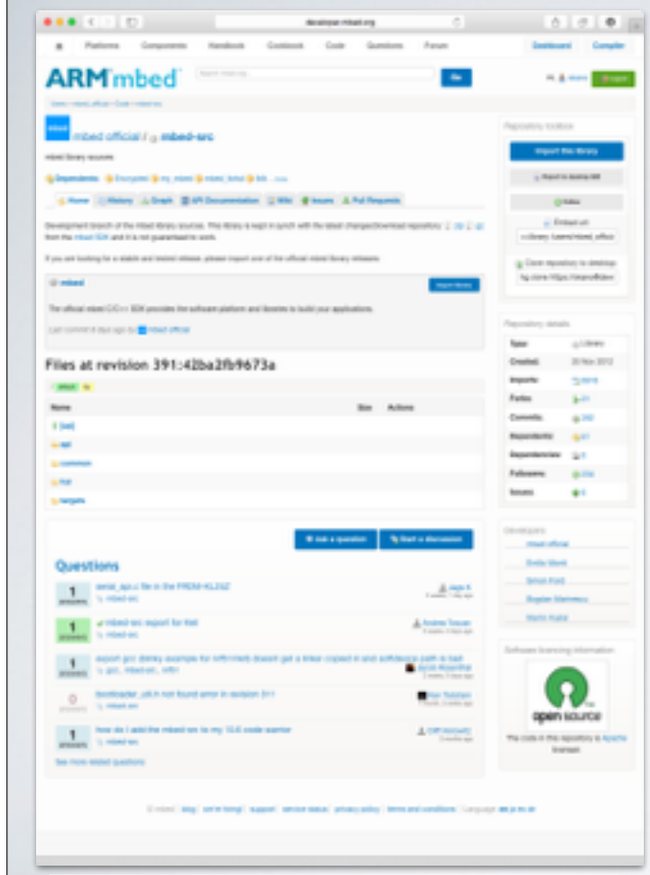
C++について，ライブラリの書き方について，かっこいい書き方，その他いろいろな情報がたくさんあります.

行き詰まった時には，いろいろ掘り返してみるのも手です (^^)

参考 (9)

- mbed-SDKの中身を知りたいときは？

mbed-SDKの内部を知りたいときは



http://developer.mbed.org/users/mbed_official/code/mbed-src/

https://github.com/mbedmicro/mbed

mbed.org内にmbed-SDKのソースが公開されています.

また，最新の開発中のソースはGithub内で見ることができます.

変更履歴

- **Version 1.0 (07-Nov-2014)**
  - 正式公開版
- **Version 1.1 (08-Nov-2014)**
  - 59ページを追加：「公開後の変更は可能か？」ページ
  - 65ページを変更：コンポーネント・ページは作成者以外も編集可能であることを追記
  - 86, 87ページを追加：「ターゲット設定を識別するための定義済みの語」
  - 92ページを変更：書籍の入手性情報を追記
  - 96ページを追加：変更履歴ページ