SPOONsoftware
QR プロジェクト

ソース等説明書

Ver1.00-01

目次

1.はじめに

1.1.使用制限および免責

本QR出力プログラムはオープンソースとなっています。
ソースの改変を含む使用は自由に行なってくださってかまいません。本プログラムを使用する際は、プログラムの目的、有償・無償に関わらず弊社に通知なく使用していただいて構いません。使用するに際して、弊社モジュールを使用している旨をどこかに記載する必要もありません。
本プログラムを使用して生じた一切の不具合に対する責任を弊社は負いません。本プログラムを使用して作成されたQRコードが読み取り不可、あるいはコードの不良により読み取り結果が不正であり、それによって如何なる損害が発生した場合でも弊社はその責任を負いません。

1.2.バグのご報告のお願い

本プログラムにバグがある場合には、ご報告いただければバグ修正を行い、通知させていただきたいと考えております。
また、プログラム自体に関する要望等があれば、その点に関してもお知らせください。とはいえプログラムの改変は自由に行って構いませんので、ご自身で改変したければそうしてくださって結構です。
バグの報告および改変要望があればこのプログラムをダウンロードしたWebサイトを御覧ください。その際、既知のバグおよび要望に関してご一読ください。

1.3.その他の注意事項

本プログラムは二次元コードシンボルーQRコードー基本仕様（JIS X 0510:2004）およびその正誤表を元に作成されています。

2.ソース概要

2.1.前提条件

本プログラムは C++で書かれています。

本プログラムは Windows 環境および UNIX 環境で動作することを目指して書かれていますが、標準的な C++を使用して書いてありますので、どの環境でも動作すると思われます。

Windows コンパイラの警告メッセージに対応するため、strcpy, memcpy 関数は strcpy_s,mememcpy_s 関数を使用しています。プリプロセッサマクロに WIN32 がセットされている場合だけ、この関数を使用するようにしております。64bit 環境等で使用する場合には、WIN32 をセットすると、不都合かと思いますので、プログラム自体を改変してください。

また、Windows でコンパイルするとき、cpp ファイルの先頭に stdafx.h をインクルードしないとうまくコンパイル出来ない場合があります。ソースファイル上の cpp ファイルに stdafx.h をインクルードするか、コンパイラの設定で「プリコンパイル済みヘッダーを使用しない」にして、回避してください。

最後に重要な点ですが、本プログラムでは1バイト文字、2バイト文字を扱う際に、以下のものとして扱っております。

1 バイト文字：unsigned char
2 バイト文字：unsigned short

2 ファイル構成

以下のファイルからなっています。

ファイル名	*ファイル説明*
QRMatrix.h	QR マトリックスコードを作成するためのクラスのヘッダファイル。この本プログラムを使用するためには、このファイルをインクルードします。
QRMatrix.cpp	QR マトリックスコードを作成するためのクラス本体ファイル。
QRReadSolomon.h	QR のエラー訂正コードを生成するためのクラスのヘッダファイルおよび本体ファイル
QRReadSolomon.cpp	
QRDataElement.h	QR の出力データ情報を格納するためのデータクラスのヘッダーファイルおよび本体ファイル
QRDataElement.cpp	
QRInformation.h	QR データを出力するためのデータ群。
QRReadSolomonInformation.h	QR のエラー訂正コードを姿勢するためのデータ群。

2.2.CQRMatrix クラス

本プログラムを使用する際には、使用するソースコード内で、QRMatrix.h をインクルードします。
インクルードすると、CQRMatrix クラスが使用可能となります。
基本的な使用の流れは以下のとおりとなります。

CQRMatrix のインスタンスを作成
例）CQRMatrix qrMatrix;

QR コードにしたい文字列をセット
例）qrMatrix.AddQRCharacters(″★″ , QRCHARACTERTYPE_KANJI);

QR コードの作成
例）qrMatrix.MakeQRMatrix(COLLECTLEVEL_L , iErrorType);

QR の縦横サイズを取得（QR は正方形であるため、縦横のサイズは等しい）
例）int iNum = qrMatrix.GetModuleNumber();

QR の縦横サイズをループしてデータを1ドットずつ取得
例）

```
for( int iRow=0; iRow<iNum; iRow++ ){
    for( int iCol=0; iCol<iNuml; iCol++ ){
        if( qrMatrix.IsDarkPoint( iCol, iRow ) )
        {
            ＜塗りつぶされている場合の処理＞
        }
        {
            ＜塗りつぶされない場合の処理＞
        }
    }
}
```

詳しくはサンプルプログラムをご覧ください。
以下に CQRMatrix の仕様を記します。

2.2.1.public な関数

2.2.1.1.CQRMatrix()

コンストラクタ。各内部変数の初期化処理などを行う。

引数:なし

戻り値:なし

2.2.1.2.~CQRMatrix()

デストラクタ。内部変数の確保領域の開放などを行う。

引数:なし

戻り値:なし

*2.2.1.3.bool AddQRCharacters(const char * pcszString , int iCharacterType)*

QR コードに出力したい文字列を内部変数にセットするために使用する。

文字列は複数登録することも可能で、登録された文字列群は内部変数。

list<CQRDataElement> m_listQRData

に追加される（ただし、iCharacterType が最後に追加されたものと同一の場合には、m_listQRData への追加は行わず、最後の要素の文字列を追加する）

使用するユーザ側で文字タイプを指定する必要があるため、データの圧縮効率性などを念頭において使用することが必要となる。

引数:

pcszString (IN)

QR コードに出力する文字列へのポインタ。文字列は NULL 終端であること。

iCharacterType (IN)

文字列の型をあらわす int 型。QR コードでは4種類の文字列の出力が認められている。すなわち、数字文字列、アルファベット文字列、8ビット文字列、漢字文字列である。

iCharacterType は以下の定義されたマクロのいずれかを指定すること

QRCHARACTERTYPE_NUMERIC：数字文字列

QRCHARACTERTYPE_ALPHABET：アルファベット文字列

QRCHARACTERTYPE_ASCII：8ビット文字列

QRCHARACTERTYPE_KANJI：漢字文字列

※数字文字列は半角数字で指定すること。アルファベット文字列は大文字アルファベット、数字および数文字の記号のみの指定とすること。漢字文字列はシフト JIS コードで指定すること。

戻り値：現状すべて true を返す。

2.2.1.4.bool MakeQRMatrix(int iCollectLevel , int & iErrorType)

指定された条件を使ってQRコード情報を作成する。

この関数を呼び出す前にAddQRCharactersを使用して、コード化したい全ての文字列を登録しておく必要がある。

引数:

iCollectLevel　　(IN)

QRコードのエラー訂正レベルを指定する。QRコードのエラー訂正レベルは4種類のレベルがあり、レベルによってエラー訂正を行うことが出来るコード数が異なる。精度の高いQRコードを作るにはエラー訂正レベルを上げる必要があるが、エラー訂正レベルを上げると、QRコード自体の大きさが大きくなる。QRコードのエラー訂正レベルはL,M.Q.Hの4種類があるが、一般的にはMを使用する。

iCollectLevelには以下の定義されたマクロのいずれかを指定すること。

COLLECTLEVEL_L:エラー訂正レベルL

COLLECTLEVEL_M:エラー訂正レベルM

COLLECTLEVEL_Q:エラー訂正レベルQ

COLLECTLEVEL_H:エラー訂正レベルH

iErrorType　(OUT)

QRコードの作成に失敗した場合、失敗した原因を示すエラーコードが格納される。

エラーコードはQRMatrix.h内に定義されているQRERR_で始まるマクロで定義されている。

戻り値:

QRデータが正常に作成出来ればtrueを、失敗した場合にはfalseを返す。

失敗した場合には、iErrorTypeにエラーコードが登録される。

2.2.1.5.int GetTypeNumber()

作成された QR コードの型番を返す関数。この関数は MakeQRMatrix が成功した場合のみ有効となる。QR コードの型番は 1 型~40 型と規定されており、それぞれの方でサイズが異なる。型は出力する文字列とその種類および、エラー訂正レベルによって決まってくる。

引数:なし

戻り値:作成された QR コードの型番号(1~40)

2.2.1.6.int GetModuleNumber()

作成された QR コードのモジュール数を返す関数。この関数は MakeQRMatrix が成功した場合のみ有効となる。モジュール数とは縦横のサイズである。1型の QR コードであれば、21×21 のモジュールがある。1 型の QR コードで本関数を呼び出した場合には 21 が返る。

戻り値:作成された QR コードのモジュール数

2.2.1.7.bool IsDarkPoint(int iColumnIndex , int iRowIndex)

作成された QR コードの指定位置が明か暗かを返す。この関数は MakeQRMatrix が成功した場合のみ有効となる。

引数:

iColumnIndex (IN)

チェックしたい列番号 0~GetModuleNumber()-1 の値を指定する。

iRowIndex (IN)

チェックしたい行番号 0~GetModuleNumber()-1 の値を指定する。

戻り値:

true の場合、指定位置のデータが暗データであることを示す。

false の場合、指定値のデータが名データであることを示す。

範囲外のデータが指定された場合には false が返る。

2.2.2.protected 関数

2.2.2.1.void ReleaseAll();

クラス内で確保した全てのクラス変数のメモリをリリースする。デストラクタ内で呼び出される。

引数:なし

戻り値:なし

2.2.2.2.void ReleaseQRMatrix();

クラス変数 m_ppucQRMatrix の確保領域を開放する

引数:なし

戻り値:なし

2.2.2.3.void ReleaseMaskMatrix();

クラス変数 m_ppucMaskMatrix の確保領域を開放する

引数:なし

戻り値:なし

2.2.2.4.void ReleaseData();

クラス変数 m_pucData の確保領域を開放する

引数:なし

戻り値:なし

2.2.2.5.void ReleaseError();

クラス変数 m_puszError の確保領域を開放する

引数:なし

戻り値:なし

2.2.2.6.bool GetDataBitNumber(int & iDataBit1_9 , int & iDataBit10_26 , int & iDataBit27_40);

QR コードに出力するデータのビット数をカウントする

この処理は内部変数 m_listQRData に出力する文字列が全てセットされた後（セットするには、2.2.1.3. bool AddQRCharacters(const char * pcszString , int iCharacterType)を使用する）2.2.2.9. bool SetTypeNumber(int iCollectLevel , int & iErrorType);から呼び出される。

QR コードは複数の文字モードをサポートしているが、データを配置するに際し、それぞれのモードの先頭部分で、該当モードで何文字分の文字列を出力するかを指定している（この部分を仕様書上「文字数指定子」と呼んでいる）。この文字数指定子が、文字モードごとあるいは QR コードの型ごとに異なる。仕様書上からの抜粋を以下に示す。

文字数指定子のビット数

型番	数字モード	英数字モード	8ビットバイトモード	漢字モード
1～9	10	9	8	8
10～26	12	11	16	10
27～40	14	13	16	12

このため、型が確定していない状態では文字数指定子が何ビットになるか分からないため、全体のデータが何ビットになるかも確定しない（逆に全体データが何ビットになるか分からないと、型番を決定できないというジレンマがある）。

そのため、本関数では、型番が 1～9 までの場合のデータビット数、型番が 10～26 間での場合のデータビット数、型番が 27～40 までのデータビット数を分けて計算している（本関数が型番決定よりも前に呼ばれる関数であるため）

引数：

iDataBit1_9　　(OUT)

型番が 1～9 となったときのデータ長（ビット数）出力用

iDataBit10_26　　(OUT)

型番が 10～26 となったときのデータ長（ビット数）出力用

iDataBit27_40　　(OUT)

型番が 27～40 となったときのデータ長（ビット数）出力用

戻り値：

==true：データ長の取得に成功

==false：データ長の取得に失敗（セットされているデータのモードが対象外の場合）

2.2.2.7.bool CreateDataBits(int & iErrorType);

内部変数 m_listQRData に登録されている各モードごとのデータから、データビット列を作成し、m_pucData を作成する。この関数が呼ばれる前に、2.2.2.9. bool SetTypeNumber(int iCollectLevel , int & iErrorType);を使用して QR コードの型番が決定している必要がある。

最終的に m_pucData でセットしたデータが QR データのデータ領域に満たない場合、仕様では埋め草コードを埋めることになっている。この埋め草コードで埋める動作も本関数内で行う。

m_pucData に最終的にどの様なデータが格納されるかはを 2.2.3.8. unsigned char * m_pucData; 参照のこと。

引数:

iErrorType　(OUT)

エラーが発生した場合のエラーコードの格納用変数

戻り値:

==true:作成成功

==false:作成失敗(iErrorType がセットされる)

2.2.2.8.unsigned char GetBitMask(int iBitNumber);

指定されたデータビット番号のビットマスクを取得する。

例えば、iBitNumber に 0 が指定されている場合には戻り値は 0x01、1 が指定されている場合には戻り値は 0x02、2 が指定されている場合には戻り値は 0x04 となる。

ビット番号

7	6	5	4	3	2	1	0

iBitNumber=4 の場合には上記の状態から「00010000」つまり「0x10(16 進数)」となる。

このマスクは特定の1バイト変数内の特定のビットが ON か OFF かを判定するために使用する。

引数:

iBitNumber　　(IN)

マスクを作成するビット番号

戻り値:

指定ビットのマスク数。iBitNumber が 0～7 でない場合の動作は不定

2.2.2.9.bool SetTypeNumber(int iCollectLevel , int & iErrorType);

QR コードで表示したいデータの容量とエラーレベル番号を使用して、QR コードの型番を決定する。QR コードは型番（＝サイズと考えても良い）が 1～40 型までの 40 個あり、それぞれで全体的に登録できるデータ数が異なり、その全体領域をデータ領域とエラー訂正領域に分けて使用する。

QR コードには仕様上、エラー訂正レベルが4段階あるため、それぞれの型番でデータ領域として使用できる領域も必然的に4段階に分けられる。

例えば1型の QR コードの場合

エラー訂正レベル L だと 19 バイト

エラー訂正レベル M だと 16 バイト

エラー訂正レベル Q だと 13 バイト

エラー訂正レベル H だと 9 バイト

のデータを記述できる。逆に言えば、データの必要領域とエラー訂正レベルを指定すれば逆引き的に型番を決定できるということになる。各型番で、どのくらいのデータが格納できるかは基本仕様を参照していただきたいが、ソースコードでは、2.5.1. STypeMatrix c_typeMatrix を参照いただきたい。本関数はこの逆引きを使用して、型番の決定を行っている。また、必要なデータ領域を計測するために、本関数から2.2.2.6. bool GetDataBitNumber(int & iDataBit1_9 , int & iDataBit10_26 , int & iDataBit27_40);を呼び出している。

引数:

iCollectLevel　　(IN)

要求するエラー訂正レベル。

以下の定義されたマクロを使用すること

COLLECTLEVEL_L:エラー訂正レベル L

COLLECTLEVEL_M:エラー訂正レベル M

COLLECTLEVEL_Q:エラー訂正レベル Q

COLLECTLEVEL_H:エラー訂正レベル H

iErrorType　(OUT)

エラーが発生した場合のエラーコードの格納用変数

戻り値:

==true:作成成功

==false:作成失敗（iErrorType がセットされる）

2.2.2.10.bool InitModuleArea(int & iErrorType);

型番にあわせて m_ppucQRMatrix と m_ppucMaskMatrix の領域を確保する。

QR コードの型番が決定すれば、モジュール数が確定されるが、そのモジュール数分のマトリックスを各メンバ変数に確保し、内部を QR_UNKNOWN(-1)で埋める。

各変数内の格納データの詳細は 2.2.3.2. char ** m_ppucQRMatrix;または 2.2.3.3. char ** m_ppucMaskMatrix;を参照のこと

引数:

iErrorType　(OUT)

エラーが発生した場合のエラーコードの格納用変数

戻り値:

==true:現在のところ true しか返らない。

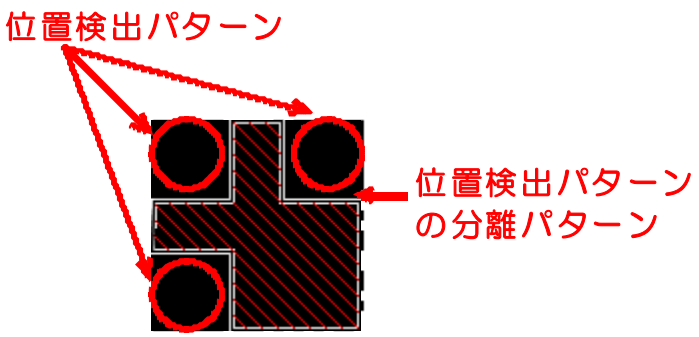
2.2.2.11.bool SetPositionDitectPattern(int & iErrorType);

m_ppucQRMatrix の該当位置に「位置検出パターン」および「位置検出パターンの分離パターン」情報を出力する。

具体的には、m_ppucQRMatrix の

「位置検出パターン」の暗部分を QR_POSITIONDITECT_DARK に、「位置検出パターン」の明部分および「位置検出パターンの分離パターン」を QR_POSITIONDITECT_LIGHT にする。

「位置検出パターン」、「位置検出パターンの分離パターン」とは下図に示す領域のことである。



引数:

iErrorType　(OUT)

エラーが発生した場合のエラーコードの格納用変数

戻り値:

==true:現在のところ true しか返らない。

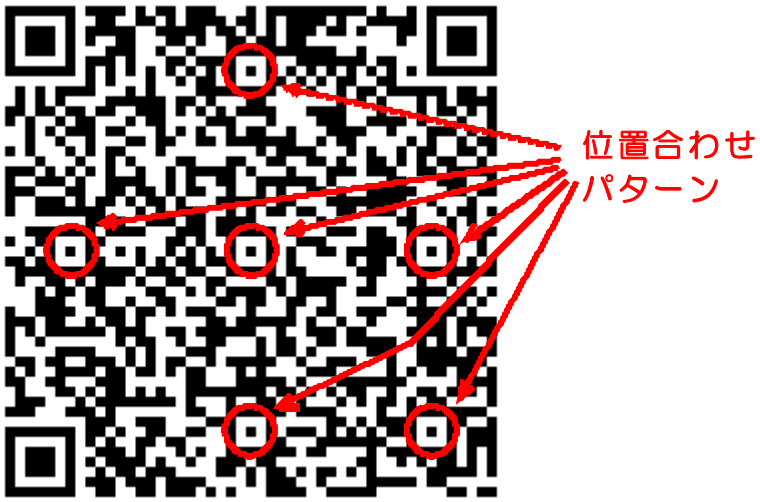
2.2.2.12.bool SetPositionCheckPattern(int & iErrorType);

m_ppucQRMatrix の該当位置に「位置合わせパターン」情報を出力する。

具体的には「位置合わせパターン」の暗部分を QR_POSITIONCHECK_DARK に、明部分を QR_POSITIONCHECK_LIGHT とする。

「位置合わせパターン」は2型以降の QR コードに出力される、3×3のモジュールで、各型番ごとに出力位置が決まっている。出力位置の具体的な場所に関しては基本仕様にあるとおりである。プログラム上の位置定義は 2.5.2. short c_sPositionChecks を参照のこと。

「位置合わせパターン」とは下図に示す部分のことである。



引数:

iErrorType　(OUT)

エラーが発生した場合のエラーコードの格納用変数

戻り値:

==true:現在のところ true しか返らない。

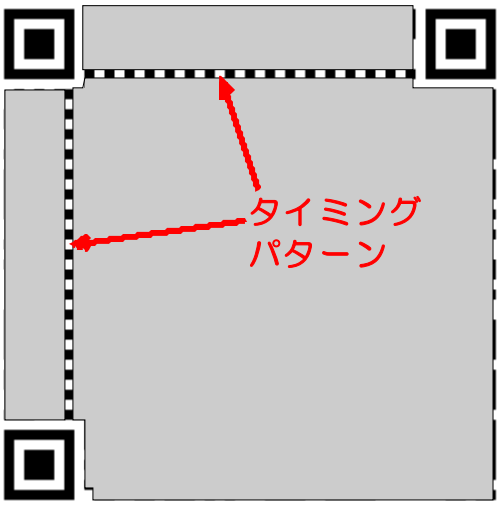
2.2.2.13.bool SetTimingPattern(int & iErrorType);

m_ppucQRMatrix の該当位置に「タイミングパターン」を出力する。

具体的には「タイミングパターン」の暗部分に QR_TIMING_DARK、明部分に QR_TIMING_LIGHT を出力する。

タイミングパターンは QR コードの 7 行目、7 列目のうち「位置検出パターン」「位置検出パターンの分離パターン」「位置合わせパターン」で内部分に暗・明を交互に配置する。

タイミングパターンの出力位置に関しては以下を参照



引数:

iErrorType　(OUT)

エラーが発生した場合のエラーコードの格納用変数

戻り値:

==true:現在のところ true しか返らない。

2.2.2.14.bool SetTypeInformation(bool bReserveOnly , int & iErrorType);

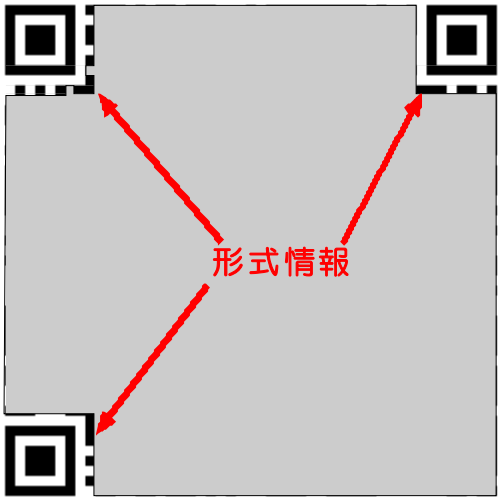
m_ppucQRMatrix の該当位置に「形式情報」を出力する。

具体的には「形式情報」の暗部分に QR_TYPE_DARK を明部分に QR_TYPE_LIGHT を出力する。

形式情報の出力は、データとエラー訂正符号を全て出力した後に8種類のマスクパターンを適用し、失点を計算後、マスクパターンを決定してから出ないと出力できない（形式情報にマスクパターンが含まれるため）。当然ながら、形式情報にはそのマスクパターンは適用されない。本プログラムではデータ領域とエラー訂正領域以外の全ての領域にマスクパターンを適用するための仕組みとして、データ領域とエラー訂正領域以外の全てのデータを出力してから、データがセットされていない領域を拾い出し、マスク適用領域としている（2.2.2.16. bool CreateMaskPatternArea(int & iErrorType);を参照）。

そのため、形式情報が確定する前に形式情報が出力されるべき領域にダミーデータを挿し込んでおく必要がある。第一引数はその用途の為に存在する。

形式情報の QR コード上の位置は以下を参照。



尚、形式情報にはエラー訂正レベル（2ビット）とマスクパターン（3ビット）のデータとそのエラー訂正符号（10ビット）の合計15ビットとなる。また、作成されたビットパターンが偏ったデータにならないように、それらのデータに特定のマスクパターンで処理を施している。

エラー訂正レベル2ビットは以下のような組み合わせとなる。

エラー訂正レベルL:　　01

エラー訂正レベルM:　　00

エラー訂正レベルQ:　　11

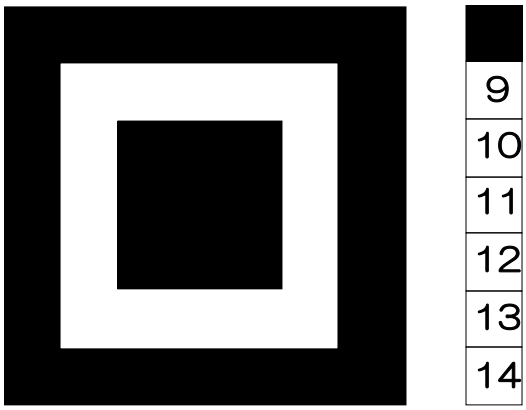
エラー訂正レベルH:　　10

またマスクパターンは 0 番～7番まであり、それぞれを2進数に変換する。

エラー訂正レベルはそれらのデータをある演算によって処理して作成するが、この演算方法は割愛する。

作成された15ビットのデータ列に 101010000010010 のマスクパターンを適用（XOR 演算すること）し、配置する。

どのように配置するかは基本仕様書を参照していただきたいが、QRコード左下のビット配置だけ説明すると以下のとおりとなる。



上記の数字はビット番号である。ビット番号はビットの最下位（右端）が 0 となる。今回の形式情報のビット数は15であるためビット番号14は最上位ビットと言うことになる。

前頁のQRコード例の形式情報を見てみると、上記の部分が「010101」となっていることが分かる。

これに、マスクパターンの上位 6 桁「101010」を再適用すると、「111111」となるので、

エラー訂正符号：Q

マスクパターン：7番

と言うことが読み取れる。説明が長くなったが、本関数の仕様については以下のとおりとなる。

引数：

bReserveOnly　　(IN)

true を指定すると、領域にダミーデータ（QR_TYPE_LIGHT）を埋め込む。false を指定すると、エラー訂正レベルとマスクパターンから形式情報を作成し、形式情報のエラー訂正符号を付加して該当場所に出力する。

iErrorType　　(OUT)

エラーが発生した場合のエラーコードの格納用変数

戻り値：

==true：出力に成功。

==false：出力に失敗

2.2.2.15.bool SetModelInformation(int & iErrorType);

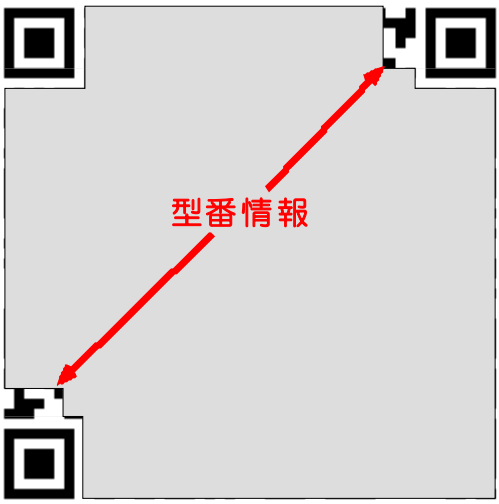
m_ppucQRMatrix の該当位置に「型番情報」を出力する。

7 型以上の QR コードにはその QR コードが何型かを示す型番情報が埋め込まれる。

型番情報は 7～40 であり、それを 6 ビットの 2 進数表記したものに、エラー訂正符号 12 ビットを付加した合計 18 ビットとなる。

型番情報は特定のマスクを適用することなく、そのまま記述される。

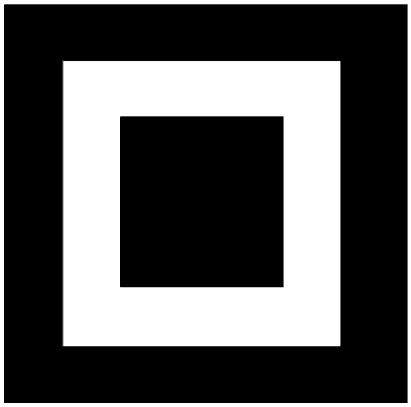
QR コード上で、型番情報が記述される位置は以下の通りとなる。



型番情報の各ビットが実際にはどう埋め込まれるかは基本仕様書を参照のこと。

左下の型番情報のビットの並びは以下の通りとなる。

0	3	6	9	12	15	■
1	4	7	10	13	16	□
2	5	8	11	14	17	■



上記の QR コードの上位6ビット（ビット番号 12～17）を見ると、001000 であるため、この QR コードが8型であることが確認できる。

本関数の仕様については以下のとおりとなる。

引数:

iErrorType　(OUT)

エラーが発生した場合のエラーコードの格納用変数

戻り値:

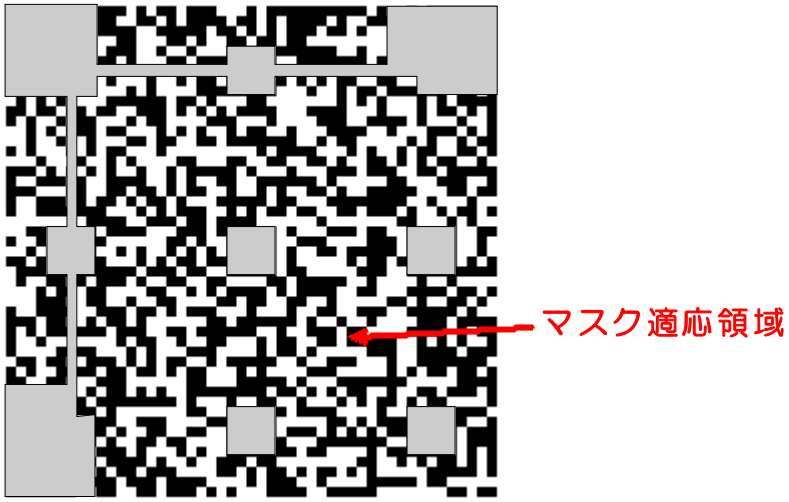
==true:現在のところ true しか返らない。

2.2.2.16.bool CreateMaskPatternArea(int & iErrorType);

現在の QR データのマトリックス m_ppucQRMatrix を全検索し、データが未セット（QR_UNKNOWN）の部分に対応するマスクマトリックス m_ppucMaskMatrix をマスクの明部分（QR_MASK_LIGHT）に変更する。

この関数は QR データ内に「位置検出パターン」、「位置検出パターンの分離パターン」、「位置合わせパターン」、「タイミングパターン」、「形式情報（ダミー）」、「型番情報」が出力された後に呼び出される。これにより、m_ppucMaskMatrix にはマスクを適用すべきモジュールに QR_MASK_LIGHT、マスクを適用すべきでないモジュールに QR_UNKNOWN がセットされた状態となり、後に実際にマスクをかけるときやマスクの失点を計算するときに、どの部分にマスク処理を施すかが分かるようになる。

マスク適用領域は概ね以下の通りとなる。



引数:

iErrorType　(OUT)

エラーが発生した場合のエラーコードの格納用変数

戻り値:

==true:現在のところ true しか返らない。

2.2.2.17.bool AddDataBits(unsigned char ucData , int iBitNumber , int & iErrorType);

クラス変数 m_pucData に対して、指定されたビットデータを追加する。

m_pucData は unsigned char 型の配列であり、データは1バイトごとで管理される。本関数は m_pucData に関数の引数である ucData の下位 iBitNumber ビット分のデータを追加する。追加するに際して、m_pucData の確保領域が足らない場合には自動拡張（128 バイトごと）し、何ビットのデータが登録されているかはクラス変数 m_iDataBitNumber にて管理する。

本関数をコールしたときの簡単な流れを以下に示す。

前提条件：m_iDataBitNumber が 0、m_pucData が空

関数コール：AddDataBits(0xff , 4 , iErrorType);

結果：

m_pucData に 128 バイトのデータが確保され、内部を 0x00 で初期化

m_pucData[0]の上位 4 ビットが 1111 となり、m_pucData[0]は 0xf0 となる。

m_iDataBitNumber が 4 となる。

↓ ↓ ↓ ↓

関数コール：AddDataBits(0xA5 , 6 , iErrorType);

結果：

0xA5 を 2 進数に直すと 10100101、下位 6 ビットをとると 100101 となる。

m_pucData[0]の下位 4 ビットが 1001 となり、m_pucData[0]は 0xf9 となる。

m_pucData[1]の上位 2 ビットが 01 となり、m_pucData[1]は 0x40 となる。

m_iDataBitNumber が 10 となる。

↓ ↓ ↓ ↓

関数コール：AddDataBits(0x9c , 8 , iErrorType);

0x9c を 2 新数に直すと 10011100

m_pucData[1]の下位 6 ビットが 100111 となり、m_pucData[1]は 0xA7 となる。

m_pucData[2]の上位 2 ビットが 00 となり、m_pucData[2]は 0x00 となる。

m_iDataBitNumber が 18 となる。

上記の要領で、現在セットされている m_pucData にビットごとのデータを追加しながら、最終的な m_pucData を作成する動作を本関数で行う。

引数:

ucData (IN)

m_pucData に追加したいデータが格納された1バイト変数。

最大で 8 ビットまでのデータを出力することができるが、有効とされるのは iBitNumber で示された下位ビットのみ

iBitNumber (IN)

ucData で示されたデータのうち、下位何ビットまでを m_pucData に追加するかを示す数。

1～8 を指定する。

iErrorType (OUT)

エラーが発生した場合のエラーコードの格納用変数

戻り値:

==true:データのセットに成功

==false:データのセットに失敗。(iBitNumber 不良)

2.2.2.18.bool AddDataBits(unsigned short usData , int & iErrorType);

クラス変数 m_pucData に対して指定された2バイトデータを出力する。

内部的には、上位8ビットと下位8ビットを 2.2.2.17. bool AddDataBits(unsigned char ucData , int iBitNumber , int & iErrorType);を使用して追加している。

引数:

usData (IN)

追加する2バイトデータを格納する。

iErrorType (OUT)

エラーが発生した場合のエラーコードの格納用変数

戻り値:

==true:データのセットに成功

==false:データのセットに失敗

*2.2.2.19.bool AddDataBits_Numeric(const char * pcszData , int iDataLength , int & iErrorType);*

指定された文字列を数字文字列として、m_pucData へ追加する。

数字文字列の出力は以下のような形式となる

項目	*ビット数*	*備考*
モード指示子	4	数字モードの場合、0001（2進数）となる
文字数指示子	10	QR コード型番が 1～9 型の場合
	12	QR コード型番が 10～26 型の場合
	14	QR コード型番が 27～40 型の場合
数字3文字分データ	10	数字を3文字ごとに区切り、10 ビットデータで指定する。
数字3文字分データ	10	
……		
最終データ	10	最終データが3文字の場合
	7	最終データが2文字の場合
	4	最終データが1文字の場合

数字データの出力は、該当となる数字3文字をひとまとめにして、10 ビットで記述する。

10 ビットで記述できる数字は 0～1023 となるため、10 ビットあれば、0～999 までの数字を記述することが可能である。

数字文字列が 3 の倍数でない場合には、最後の2文字を 7 ビットで表すか、最後の1文字を4ビットで表すかで調整する。

今、pcszData が「31415926536」、iDataLength が「11」の場合、本関数を使用して出力されるデータ列は以下のとおりとなる（QR コードは 1～9 型と仮定する）

数字文字列であるため、先頭 4 ビットは「0001」

文字数 11 文字をビットに直し、10 ビットとすると「0000001011」（前提より 1～9 型であるため）

最初の3文字 314 をビットに直し、10 ビットとすると「0100111010」

次の3文字 159 をビットに直し、10 ビットとすると「0010011111」

次の3文字 265 をビットに直し、10 ビットとすると「0100001001」

最後の2文字 36 をビットに直し、7 ビットとすると「0100100」

全てをつなぎあわせて、16 進数とすると、

「10 2D 3A 27 D0 94 80」このうち、末尾 5 ビットは残余ビットであるので、データビットには追加されない計算となる。

これら追加されたデータを関数 2.2.2.17. bool AddDataBits(unsigned char ucData , int iBitNumber , int & iErrorType);を使用して m_pucData に追加する。

引数:

pcszData (IN)

追加する文字列(半角数字のみを指定)

iDataLength (IN)

追加する文字列の文字数

iErrorType (OUT)

エラーが発生した場合のエラーコードの格納用変数

戻り値:

==true:データのセットに成功

==false:データのセットに失敗。文字列内に半角数字以外が含まれている場合など

*2.2.2.20.bool AddDataBits_Alphabet(const char * pcszData , int iDataLength , int & iErrorType);*

指定された文字列をアルファベット文字列として、m_pucData へ追加する。
アルファベット文字列の出力は以下のような形式となる

項目	*ビット数*	*備考*
モード指示子	4	アルファベットモードの場合、0010（2進数）となる
文字数指示子	9	QR コード型番が 1～9 型の場合
	11	QR コード型番が 10～26 型の場合
	13	QR コード型番が 27～40 型の場合
2文字分データ	11	アルファベット文字を2文字ずつに区切り、それぞれを 11 ビットデータで指定する。
2文字分データ	11	
……		
最終データ	11	最終データが2文字の場合
	6	最終データが1文字の場合

アルファベット1文字は以下の様にコード化する

数字の 0～9　：　値 0～9
アルファベットの A～Z　：　値 10～35
「 」スペース　：　値 36
「$」（ダラー）　：　値 37
「%」（パーセント）　：　値 38
「*」（アスタリスク）　：　値 39
「+」（プラス）　：　値 40
「-」（マイナス）　：　値 41
「.」（ピリオド）　：　値 42
「/」（スラッシュ）　：　値 43
「:」（コロン）　：　値 44

アルファベット2文字を扱う際には1文字目のアルファベットを上記の様に数値化し、それに 45 を乗じて2文字目のアルファベットを数値化したものを足す。アルファベット1文字を扱う際にはその文字を数値化したものを使用する。

今、「pcszData」が「**SPOON2011**」、iDataLength が「13」のとき、本関数を使用して出力されるデータ列は以下のとおりとなる（QR コードは 1～9 型と仮定する）

アルファベット文字列であるため、先頭 4 ビットは「0010」

文字数 13 文字を 9 ビットの2進数に変換すると「000001101」（前提より 1～9 型であるため）

最初の2文字**を上述の方法で変換すると、39×45+39=1794

11ビットの2進数に直して「11100000010」

次の2文字 SP も同様に 28×45+25=1285→「10100000101」

次の2文字 OO も同様に 24×45+24=1104→「10001010000」

次の2文字 N2 も同様に 23×45+2=1037→「10000001101」

次の2文字 01 も同様に 0×45+1=1→「00000000001」

次の2文字 1*も同様に 1×45+33=「00001001110」

最後の1文字*を 6 ビットの2進数に変換 33→「100001」

全てをつなぎ合わせて 16 新数にすると

「40 DE 05 41 62 84 0D 00 21 3A 10」

となる。（最終の 4 ビットは残余ビットとなり、実際には追加されない）

これら追加されたデータを関数 2.2.2.17. bool AddDataBits(unsigned char ucData , int iBitNumber , int & iErrorType);を使用して m_pucData に追加する。

引数:

pcszData　(IN)

追加する文字列（半角英数字（大文字）と適切な記号のみ）

iDataLength (IN)

追加する文字列の文字数

iErrorType　(OUT)

エラーが発生した場合のエラーコードの格納用変数

戻り値:

==true：データのセットに成功

==false：データのセットに失敗。文字列内にアルファベットで指定できる文字以外が含まれている場合など

*2.2.2.21.bool AddDataBits_Ascii(const char * pcszData , int iDataLength , int & iErrorType);*

指定された文字列を ASCII 文字列として、m_pucData へ追加する。
ASCII 文字列の出力は以下のような形式となる

項目	*ビット数*	*備考*
モード指示子	4	ASCII モードの場合、0100（2進数）となる
文字数指示子	8	QR コード型番が 1～9 型の場合
	16	QR コード型番が 10～26 型の場合
	16	QR コード型番が 27～40 型の場合
1文字分データ	8	1文字分のデータを 8 ビットで指定する
1文字分データ	8	
……		
最終文字データ	8	

各文字列は ASCII 文字列として扱うため、それぞれの文字をそのまま8ビットで表現する。

今、「pcszData」が「@QRcode」、iDataLength が「13」のとき、本関数を使用して出力されるデータ列は以下のとおりとなる（QR コードは 1～9 型と仮定する）

ASCII 文字列であるため、先頭 4 ビットは「0100」

文字数 13 文字を 8 ビットの2進数に変換すると「00001101」（前提より 1～9 型であるため）

@のキャラクターコード　：0x40

Q のキャラクタコード　：0x51

R のキャラクタコード　：0x52

c のキャラクタコード　：0x63

o のキャラクタコード　：0x6f

d のキャラクタコード　：0x64

e のキャラクタコード　：0x65

全てをつなぎ合わせて 16 進数にすると

「40 D4 05 15 25 36 F6 46 50」

となる。（最終の 4 ビットは残余ビットとなり、実際には追加されない）

これら追加されたデータを関数 2.2.2.17. bool AddDataBits(unsigned char ucData , int iBitNumber , int & iErrorType);を使用して m_pucData に追加する。

引数:

pcszData (IN)

追加する文字列

iDataLength (IN)

追加する文字列の文字数

iErrorType (OUT)

エラーが発生した場合のエラーコードの格納用変数

戻り値:

==true:データのセットに成功

==false:データのセットに失敗。

*2.2.2.22.bool AddDataBits_Kanji(const char * pcszData , int iDataLength , int & iErrorType);*

指定された文字列を漢字文字列として、m_pucData へ追加する。漢字文字列はシフト JIS で指定すること。

漢字文字列の出力は以下のような形式となる

項目	*ビット数*	*備考*
モード指示子	4	漢字モードの場合、1000（2進数）となる
文字数指示子	8	QR コード型番が 1～9 型の場合
	10	QR コード型番が 10～26 型の場合
	12	QR コード型番が 27～40 型の場合
漢字1文字分データ	13	pcszData 2バイト分を1文字として、下に述べる方法を用いてコード化し、13 ビットにまとめる。
漢字1文字分データ	13	
……		
最終文字データ	13	

漢字1文字分データは以下の方法を用いて 11 ビットに変換する

漢字1文字分のコードが 0x8140～0x9FFC の場合、1文字分のコードから 0x8140 を引く（引いたコードを(A)とする）

コードが 0xE040～0xEBBF の場合、1文字分のコードから 0xC140 を引く（引いたコードを(A)とする）

それ以外のコードの場合はエラーとなる。

(A)の上位バイトを(AH)、下位バイトを(AL)とする。

(AH)に 0xC0 を乗じ、(AL)を足し、その値を 13 ビットの 2 進数に変換する

今、「pcszData」が「★鵜」、iDataLength が「2」のとき、本関数を使用して出力されるデータ列は以下のとおりとなる（QR コードは 1～9 型と仮定する）

漢字文字列であるため、先頭 4 ビットは「1000」

文字数 2 文字を 8 ビットの2進数に変換すると「00000010」（前提より 1～9 型であるため）

★のキャラクタコード ：0x819A

0x8140～0x9FFC なので、0x8140 を引く →0x0054（上位バイト 0x00、下位バイト 0x54）

上位バイトに 0xC0 を乗じ、下位バイトを足す。

0x00×0xC0+0x54=0x54　　→13 ビット 2 進数に変換「0000001010100」

鵜のキャラクタコード：0xEA40

0xE040～0xEBBF なので、0xC140 を引く　　→0x2900（上位バイト 0x29、下位バイト 0x00）

上位バイトに 0xc0 を乗じ、下位バイトを足す

0x29×0xC0+0x00=0x1EC0　　→13 ビット 2 進数に変換「1111011000000」

全てをつなぎ合わせて 16 進数にすると
「20 08 0A 9E C0」

となる。

これら追加されたデータを関数 2.2.2.17. bool AddDataBits(unsigned char ucData , int iBitNumber , int & iErrorType);を使用して m_pucData に追加する。

引数：

pcszData　(IN)

追加する文字列（漢字文字列のみ）

iDataLength (IN)

追加する文字列の文字数

iErrorType　(OUT)

エラーが発生した場合のエラーコードの格納用変数

戻り値：

==true：データのセットに成功

==false：データのセットに失敗（漢字文字列以外が指定された場合など）

2.2.2.23.bool OutputDataBits(int & iErrorType);

データビット列（m_pucData）とエラー訂正ビット列（m_pucError）を QR コードのデータ領域とエラー訂正領域に出力する。これにより m_ppucQRMatrix にデータとエラー訂正が埋め込まれる。

本関数を出力する前に「位置検出パターン」「位置検出パターンの分離パターン」「位置合わせパターン」「タイミングパターン」「型番情報」「形式情報（ダミー）」が出力されている必要がある。

また、m_pucData（ 2.2.2.7. bool CreateDataBits(int & iErrorType);）と m_pucError（ 2.2.2.31. bool CreateErrorCollectData(int & iErrorType);）が作成されている必要がある。

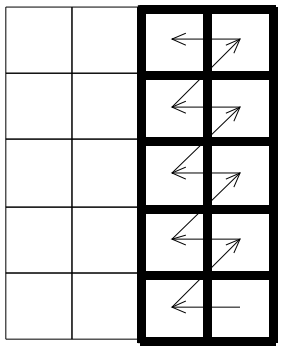
データの出力はまずデータビット列を QR コードの右下から出力していく、その後続きの部分にエラー訂正ビット列を出力する。データビット列・エラー訂正ビット列は一定の法則に従って出力する。一定の法則とは以下のとおり

最初のデータの出力方向は上方向とする。

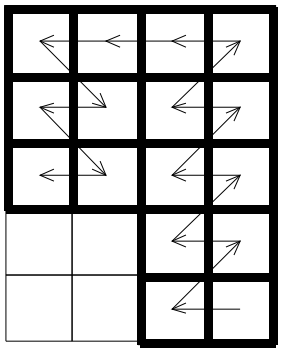
QR コードの右端から2モジュールずつを出力ブロックとする（左側のタイミングパターンの部分は縦側にひとつも出力領域がないため、出力ブロックには設定しない）

データの1ビット目を QR コードの一番右下に出力し、その後2ビット目を1モジュール左側に出力する。3ビット目は2ビット目の右上（1ビット目の上）に出力し、4ビット目はその左側に出力する。

上記のように、2モジュールずつの出力ブロックの右側・左側を埋め順次上に向かって出力する。QR コードの上端に達したら、その後、同じ法則に従って、下方向への出力を行う。

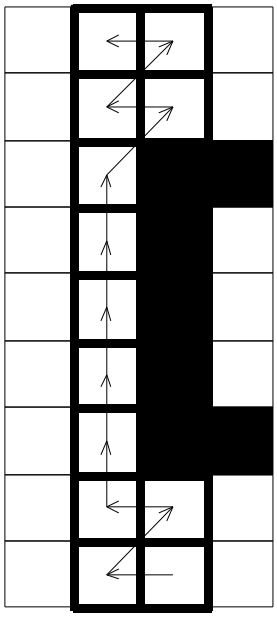


ＱＲコード出力方向



ＱＲコード上端

既に位置合わせパターンなどが出力されている部分に関しては、それらを避けてデータの出力を行う。



各ビットの出力パターンに関しては、基本仕様書に詳しく述べられているため、これを参照すること。また、本関数を見るのも出力方法を調べるのに役立つと思われる。

さて、データビットの出力、エラー訂正ビットの出力は、データの先頭から単純に出力を行うわけではない。QR コードの型によってデータビット及びエラー訂正ビットは複数の RS ブロックに分けられる（RS ブロック＝リードソロモンブロック）。分けられた RS ブロック毎の1バイト目のデータを出力し、次に2バイト目のデータを出力すると行った具合に、データの出力を行う。説明がうまくないので、具体的な例を下記に示す。

5型の QR コードで、エラー訂正レベルが H のものを考えてみる。

基本仕様書（あるいは STypeMatrix でも良いが）を見ると、本 QR コードの全領域が134バイト（1072ビット）で、そのうちデータとして使用できる領域が46バイト（368ビット）であることが分かる。また、RSブロック数は4となる。全領域からデータ領域を引いた88バイト（704ビット）がエラー訂正領域で有ることも分かる。

データ領域をRSブロック数で割ると、1ブロック当たりのバイト数は幾つになるだろうか？

46÷4＝11バイトあまり2バイトとなる。このことから、RSブロック2つが11バイト、残りのRSブロック2つが12バイトとなる。

エラー訂正領域をRSブロック数で割ると幾つになるだろうか？

88÷4＝22バイト（エラー訂正領域は必ずRSブロック数で割り切れる様に設計されている）

5型のエラー訂正レベルHのデータ領域は46バイトあることは前述のとおりであるが、これらを先頭バイトから便宜上D1～D46と呼ぶことにする。

上述のとおり、データ領域を4つのRSブロックに分けたとき、前の2つのブロックが11バイト、後の2つのブロックが12バイトとなる。これを各データに割り振ると下記のようになる。

RSブロック1 ： D1～D11

RSブロック2 ： D12～D22

RSブロック3 ： D23～D34

RSブロック4 ： D35～D46

同様にエラー訂正領域をE1～E88とすると、

RSブロック1 ： E1～E22

RSブロック2 ： E23～E44

RSブロック3 ： E45～E66

RSブロック4 ： E67～E88

となる。

データの出力、エラー訂正の出力は各RSブロックのバイトごとに行われるため、

[D1][D12][D23][D35][D2][D13]…[D11][D22][D33][D44][D34][D36][E1][E23][E45][E67][E2]…[E22][E44][E66][E88]

という順序で行われることになる。

★リードソロモンを使用したエラー訂正符号化の際にも同様のブロックのくくりを用いて行う。リードソロモンを使用したエラー訂正符号化に関しては 2.2.2.31. bool CreateErrorCollectData(int & iErrorType);を参照のこと。

実際にQRコードの型によってどのように分けられるかに関しては STypeMatrix 構造体のメンバ m_iRSBlockNumber に記述されている。各 RS ブロックごとのデータ数、エラー訂正数に関しては 2.5.1. STypeMatrix c_typeMatrix を参照のこと。

本関数の仕様に関しては以下のとおりとなる。

引数:

iErrorType (OUT)

エラーが発生した場合のエラーコードの格納用変数

戻り値:

==true:データのセットに成功

==false:データのセットに失敗

2.2.2.24.unsigned char GetDataBit(int iBitIndex , bool bDataRequest);

2.2.2.25.bool GetNextDataPutPoint(int & iColumnIndex , int & iRowIndex, bool & bFirstColumn, bool & bDirectionUp);

2.2.2.26.bool DecideMaskPattern(int & iErrorType);

2.2.2.27.int GetLostPoint_SeqModule(int iSeqBlock);

2.2.2.28.int GetLostPoint_ModuleBlock(int iColumnIndex , int iRowIndex);

2.2.2.29.int GetLostPoint_PositionDitect(int iColumnIndex , int iRowIndex);

2.2.2.30.int GetLostPoint_BlackWhiteRatio(int iBlackPoint);

2.2.2.31.bool CreateErrorCollectData(int & iErrorType);

2.2.2.32.bool SeparateTotalNumber(int iSeparateCount , int iTotalNumber, int & iFirstCount , int & iFirstNumber , int & iSecondCount , int & iSecondNumber);

2.2.2.33.inline bool SetModulePoint(int iColumnIndex , int iRowIndex , int iQRPoint)

2.2.2.34.inline int GetModulePoint(int iColumnIndex , int iRowIndex)

2.2.2.35.inline int GetMaskPoint(int iColumnIndex , int iRowIndex , int iQRMaskPattern = -1)

2.2.2.36.bool GetExpressionMod(unsigned char *puszBaseData , int iBaseDataLength, unsigned char *puszCoefficients , unsigned char *puszError , int iErrorLength);

2.2.3.protected 変数

2.2.3.1.list<CQRDataElement> m_listQRData;

2.2.3.2.char ** m_ppucQRMatrix;

2.2.3.3.char ** m_ppucMaskMatrix;

2.2.3.4.int m_iModuleNumber;

2.2.3.5.int m_iModelTypeIndex;

2.2.3.6.int m_iDataBitNumber;

2.2.3.7.int m_iDataSizeNumber;

2.2.3.8.unsigned char * m_pucData;

2.2.3.9.int m_iErrorSizeNumber;

2.2.3.10.unsigned char * m_pucError;

2.2.3.11.int m_iQRMaskPattern;

2.3.CQRReadSolomon クラス

リードソロモン法による QR のエラー訂正コードを取得するためのクラスです。QRReadSolomon.h をインクルードすることによって使用可能となります。

本クラスは CQRMatrix 内で使用されます。

以下に CQRReadSolomon クラスの仕様を記します。

2.3.1.public **関数**

現在製作中

2.3.2.protected **関数**

現在製作中

2.3.3.protected **変数**

現在製作中

2.4.CQRDataElement クラス

QR コードに出力する文字列を管理するためのクラスです。QRDataElement.h をインクルードすることによって使用可能となります。
本クラスは CQRMatrix 内で使用されます。

以下に CQRDataElement クラスの仕様を記します。

2.4.1.public 関数

現在製作中

2.4.2.protected 関数

現在製作中

2.4.3.protected 変数

現在製作中

2.5.QRInformation.h

2.5.1.STypeMatrix c_typeMatrix

2.5.2.short c_sPositionChecks

3.サンプルプログラムに関して

サンプルプログラムは Sample フォルダ内にある以下の3つのプログラムです。

CommandSample.cpp

WindowsSample.cpp

GDSample.cpp

3.1.CommandSample.cpp

コマンドプロンプトベースのサンプルプログラムです。

Windows でも UNIX でも動作します。

実行すると、暗モジュールが「■」で、明モジュールが「　」で表示 s れます。このデータは「■」に余白があるため、QR コードとしての読み取りは出来ないと思われますが、動作のサンプルプログラムとしては分かりやすいため、本プログラムを使用するにはこのサンプルを使うのが良いと思われます。

Windows の場合、コマンドプロンプトで空のプロジェクトを作成し、プロジェクトに QR コード出力用の各プログラム
（QRMatrix.h、QRMatrix.cpp、QRReadSolomon.h、QRReadSolomon.cpp、QRDataElement.h、QRDataElement.cpp、QRMatrixInformation.h、QRReadSolomonInformation.h）と、サンプルプログラム CommandSample.cpp をプロジェクトに追加後、コンパイルして実行します。全てのプログラムは同一のフォルダ内に入れてください。

UNIX の場合、gc コンパイラでコンパイルするには、全てのプログラムを同一のディレクトリに入れ、

cc -c QRMatrix.cpp

cc -c QRReadSolomon.cpp

cc -c QRDataElement.cpp

cc -c CommandSample.cpp

としてコンパイルを実行し、

cc -o CommandSample CommandSample.o QRMatrix.o QRReadSolomon.o QRDataElement.o -lstdc++

として実行ファイルを作成します。

3.2.WindowsSample.cpp

Windows のウィンドウを表示し、そこに QR コードを表示するサンプルです。

コードの大半はウィンドウの表示と制御絡みとなっており、サンプルとしてはシンプルではありませんが、ここで表示された QR コードは携帯電話などを含む各読み取り装置で読み取ることが出来ます。

実行するには、WIN32 プロジェクトで空のプロジェクトを作成し、QR コード出力用の各プログラムおよび、WindowsSample.cpp をプロジェクトに追加します。各プログラムは同一のフォルダに置いてください。また、コンパイル設定はマルチバイト文字セットで行ってください。（サンプルはユニコードに対応していません）

その後、ビルドを行い、実行します。

3.3.GDSample.cpp

UNIX で GD を使用して JPEG ファイルを作成するサンプルです。実行すると、カレントディレクトリに「qrimage.jpg」ファイルが作られます。

コンパイルするには、QR コード出力用の各プログラム及び GDSample.cpp を同一のディレクトリに入れ、

cc -c QRMatrix.cpp

cc -c QRReadSolomon.cpp

cc -c QRDataElement.cpp

cc -c GDSample.cpp

として、コンパイルを行い、

cc -o GDSample GDSample.o QRMatrix.o QRReadSolomon.o QRDataElement.o -lstdc++ -lm -ljpeg -lm

として、実行モジュールを作成します。

GD がインストールされていることが前提条件となります。もし、gd.h がインクルード出来ない場合は、「-I/usr/local/include」（ディレクトリは GD のインクルードディレクトリにより異なります）を GDSample.cpp のコンパイル時に指定してください。また、リンクするときに GD 絡みでリンクエラーが発生する場合はリンク時に「-L/usr/local/lib」をリンク時に指定してください。

4.最後に

本ドキュメントの著作権は SPOONsoftware に帰属します。

本ドキュメントを改変しないでください。