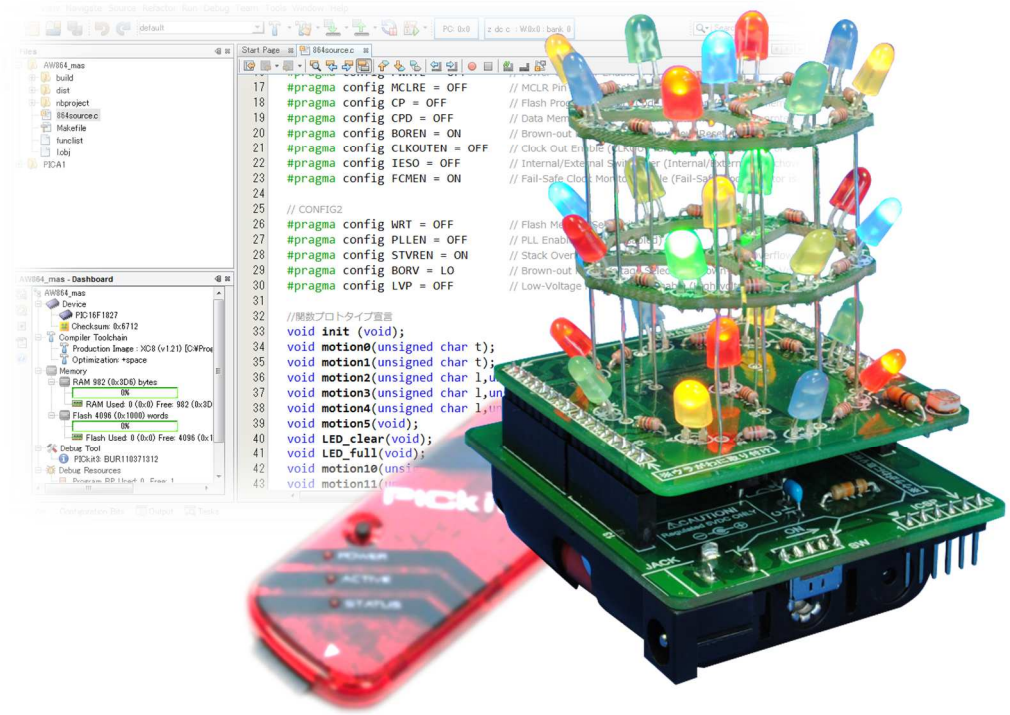
ピ　カ　・　タ　ワ　ー

PICΛTower

C 言語プログラミングマニュアル



株式会社イーケイジャパン

はじめに

このマニュアルは、エレキットの『PICA Tower(ピカ・タワー：AW-864)』を使用し、LED のオリジナル点灯パターンの作成やセンサーでの点灯制御を行うプログラム作成を通して、C 言語でのプログラミングの方法を学習することを目的としています。
PICA Tower には Microchip 社の『PIC16F1827』マイコンを使用しています。Microchip 社の PIC マイコンのプログラミングを行うための開発環境は同社から無償で提供されていますので、このマニュアルではその環境を使用した方法を解説しています。
(※マイコンへプログラムを書き込むためには、PICkit3(Microchip 社の純正プログラマー)、及びパソコン(Windows XP 以降、Mac OSX 以降、Linux)などが別途必要です。)

・PICA Tower (AW-864)について
PICA Tower は 4 色(赤・青・緑・黄)の LED(発光ダイオード)27 コを立体的に配置した 3D イルミネーションキットです。

PICA Tower は株式会社イーケイジャパンと福岡県立福岡工業高等学校の共同開発プロジェクト(2013 年)により商品化しました。

お断り

本書の内容は『PICA Tower (AW-864)』のマイコンのプログラミングに必要な最低限の解説となります。また、マイコンの機能もその一部についてのみ解説しており、全てを解説しているものではありません。
更に詳しく C 言語やマイコンについてお知りになりたい方は、専門書をお読みください。

また、プログラムに必要な開発環境などはその開発元である Microchip 社によるアップデート等により本書の内容と異なる場合があります。あらかじめご了承ください。

本書で解説している開発環境についてのサポートは当社ではお受けできません。
開発環境や PICkit3 についてのご質問等は、開発元である Microchip 社にお尋ねください。

もくじ

1. 開発環境の準備

プログラミングを始めるためにはプログラムを作成するためのソフトウエアや道具などを準備する必要があります。これらのソフトや道具のことを『開発環境』といいます。

用 意 す る も の

●パソコン

・Windows の場合 ――――― XP 以降

・Mac の場合 ――――――― OSX 以降

・Linux でも使用可能

●PICkit3

Microchip 社の純正プログラマーです。

パーツ店や通信販売で購入することができます。残念ながら当社では取り扱っておりません。

市販で 4,000～5,000 円ほどですが、PIC マイコンのプログラミングを行うなら、ぜひ揃えておきたい道具です。

●ソフトウエア

・MPLAB X IDE

プログラム作成やマイコンへの書き込みなどを行う統合ソフト。

Microchip 社のホームページから無償でダウンロードできます。

・XC8

C 言語でプログラムをするための C コンパイラ。

これも Microchip 社のホームページから無償でダウンロードできます。

コンパイラとはプログラム言語で書かれたものをコンピュータが理解できる機械語に翻訳するソフトのことで、C 言語を翻訳するソフトなので『C コンパイラ』といいます。

MPLAB X IDE のインストール

※用意するものの項で記載しましたが、プログラミングするために必要なパソコンは、Windows をはじめ Mac や Linux も使用することができます。
このマニュアルでは Windows7 で使用する場合を例に解説していきます。ご了承ください。

まず、マイクロチップ・テクノロジー・ジャパン株式会社のホームページにアクセスします。

http://www.microchip.co.jp/

上部に並んでいるメニュー部分の中の『製品情報』の文字の上にマウスカーソルを重ねると一覧が表示されますので、その中から『MPLAB X IDE』をクリックします。

表示されたページの左側にある『MPLAB X FREE DOWNLOAD』をクリックすると、ソフトのダウンロード画面が表示されますので、その中から使用するパソコンに対応した『MPLAB X IDE(v x.xx)をクリックしダウンロードします。ダウンロードするソフトの保存先は、インストール時にわかりやすいよう、デスクトップを保存先にしておきましょう。
(ダウンロードの方法はご使用のブラウザーのマニュアルをご覧ください。)
※(v x.xx)の部分は MPLAB X IDE(以降 MPLAB X と表記します)のバージョンを表しています。

タイトル	リリース日	サイズ	D/L
Windows (x86/x64)			
MPLAB® X IDE v1.80	2013/05/06	320 MB	
MPLAB® X IDEリリースノート/ユーザガイド v1.80 (インストーラに対する更新情報)	2013/05/06	150KB	
MPLAB® X IDE Chinese Translation Files v.1.80	08/08/2013	22Mb	
MPLAB® XC8コンパイラv1.12	2012/12/04	168 MB	
MPLAB® XC16コンパイラv1.11	2012/12/13	122 MB	
MPLAB® XC32コンパイラv1.21	2013/05/06	105 MB	
Linux 32-Bit and Linux 64-Bit (Requires 32-Bit Compatibility Libraries)			
MPLAB® X IDE v1.80	2013/05/06	270 MB	
MPLAB® X IDEリリースノート/ユーザガイド v1.80 (インストーラに対する更新情報)	2013/05/06	150KB	
MPLAB® X IDE Chinese Translation Files v.1.80	08/08/2013	22Mb	
MPLAB® XC8コンパイラv1.12	2012/12/04	172 MB	
MPLAB® XC16コンパイラv1.11	2012/12/13	120 MB	
MPLAB® XC32コンパイラv1.21	2013/05/06	104 MB	
Mac (10.X)			
MPLAB® X IDE v1.80	2013/05/06	240 MB	
MPLAB® X IDEリリースノート/ユーザガイド v1.80 (インストーラに対する更新情報)	2013/05/06	150KB	
MPLAB® X IDE Chinese Translation Files v.1.80	08/08/2013	22Mb	
MPLAB® XC8コンパイラv1.12	2012/12/04	169 MB	
MPLAB® XC16コンパイラv1.11	2012/12/13	121 MB	
MPLAB® XC32コンパイラv1.21	2013/05/06	107 MB	

もし、上記ページが英語になっていた場合、ページ上方のメニューから『Japanese』を選択し、日本語表示に変更することができます。

MPLAB X IDE の使い方を解説した『ユーザーズガイド』もダウンロードしておきましょう。
しかし、ソフトをダウンロードするページにある『MPLAB X IDE リリースノート/ユーザーガイド』は英語表記のものです。

ダウンロードタブの右の『関連文書』タブをクリックして現れたページの『MPLAB X IDE ユーザーズガイド』は日本語で書かれていますので、こちらをダウンロードしましょう。

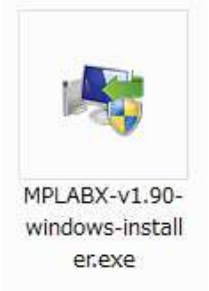
MPLAB X のダウンロードが完了すると、デスクトップに MPLAB X のインストーラーの圧縮ファイルが作成されています。このファイルは圧縮されていますので『展開』する必要があります。

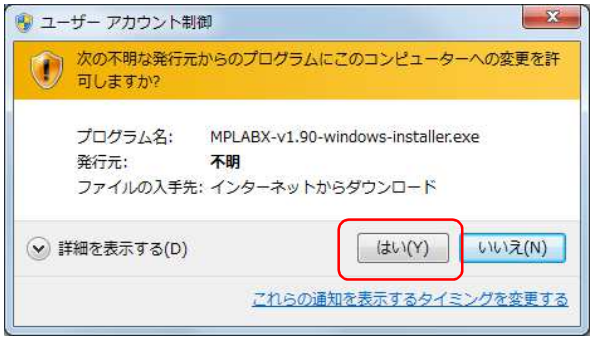
ファイルを右クリックしてメニューを表示し、『全て展開(T)…』を選びます。

下記ウインドウが表示されますので、『展開(E)』をクリックします。
『□完了時に展開されたファイルを表示する(H)』にチェックを入れておきましょう。

展開が完了すると自動的にフォルダの中身が表示され、その中にインストーラーがあります。
このインストーラーをダブルクリックして、インストールを開始します。

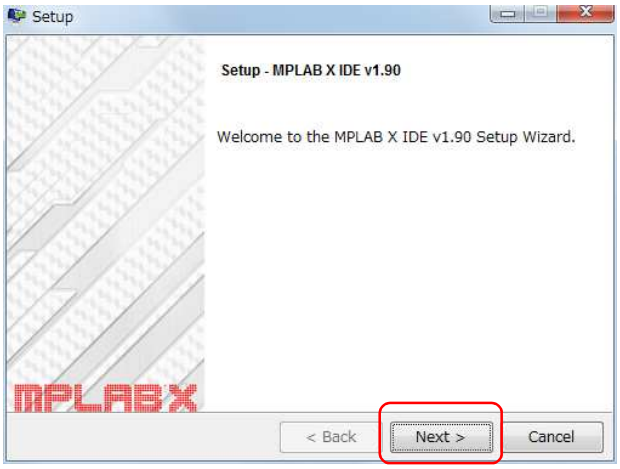
①





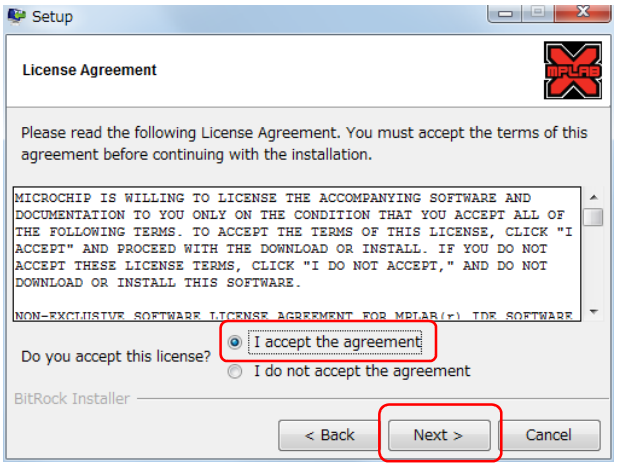
最初にこの画面が表示される場合がありますので、『はい(Y)』をクリックします。

②



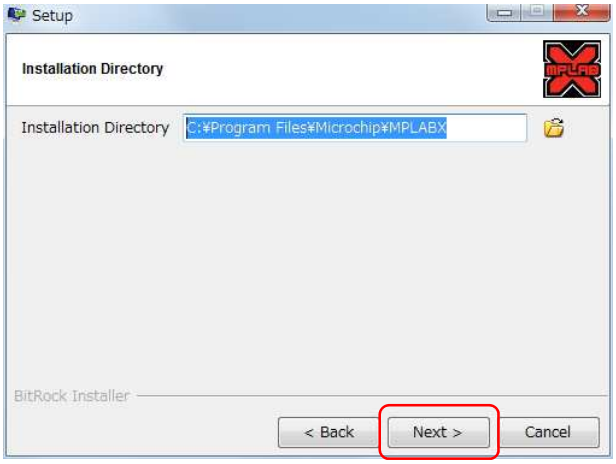
最初の画面が表示されたら『Next >』をクリックします。

③



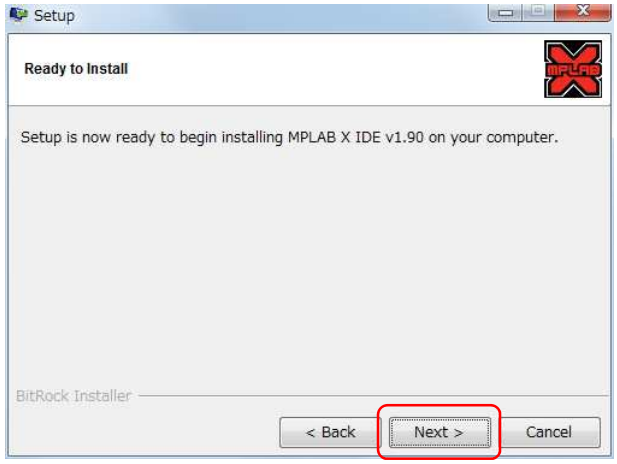
ライセンスに対する確認画面が表示されますので、『I accept the agreement』にチェックを入れ、『Next >』をクリックします。

④



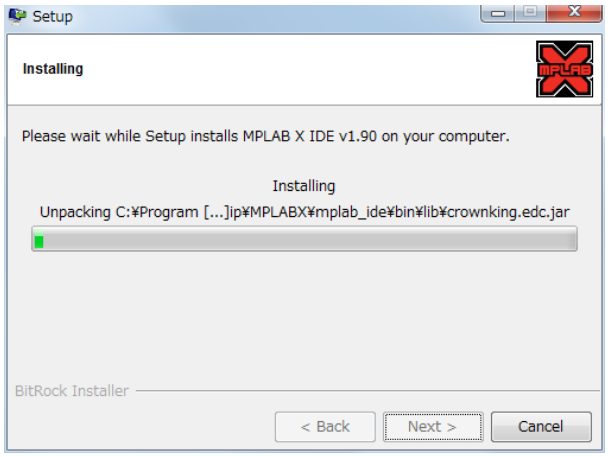
この画面はインストールする場所の設定ですから、特に変更せずに『Next >』をクリックします。

⑤



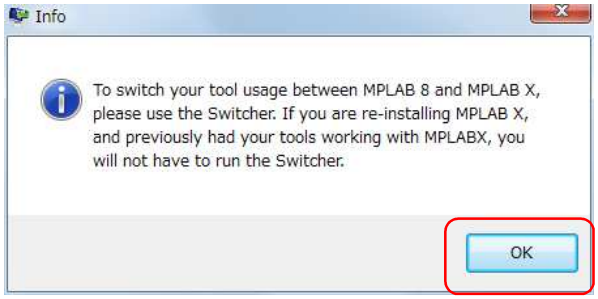
インストールする準備ができましたので、『Next >』をクリックするとインストールが始まります。

⑥



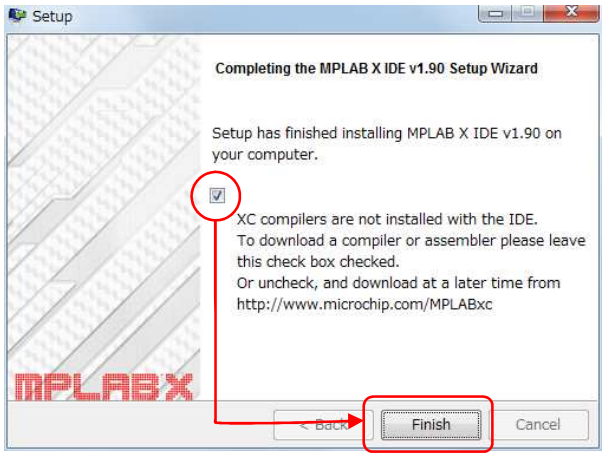
インストール中はバーグラフ表示になり、
進捗状況が表示されます。

⑦



インストールが完了するとこの画面が表示されますので、『OK』をクリックします。

⑧



全て完了すると左の画面が表示されますが、続けて『XC8 コンパイラ』をインストールする必要がありますので中ほどにある『□』にチェックを入れて『Finish』をクリックします。

⑨

自動的にブラウザーが起動し、XC コンパイラのダウンロードページが表示されます。

⑩

そのページの中ほどの左側にファイルをダウンロードするリンクがありますので、使用するパソコンに合った『XC8』をダウンロードします。
これもわかりやすいように、デスクトップに保存しておきましょう。

XC8 コンパイラのインストール

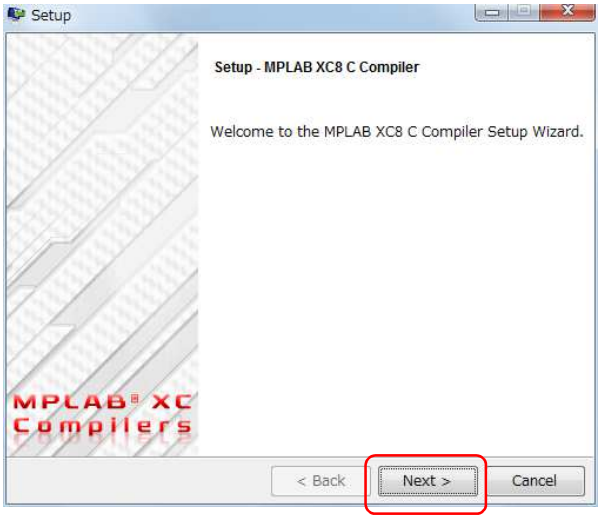
デスクトップに保存された XC8 コンパイラのファイルは圧縮されていませんので、MPLAB X の時のように展開する必要はありません。

ダブルクリックしてインストールを開始します。

①

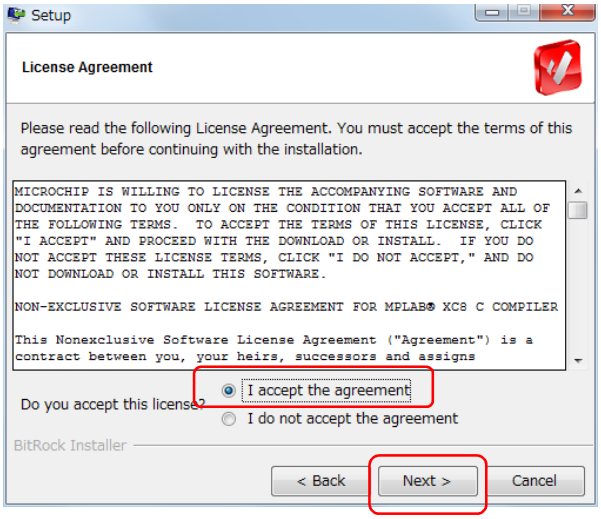
今回もこの画面が表示される場合がありますので、『はい(Y)』をクリックします。

②



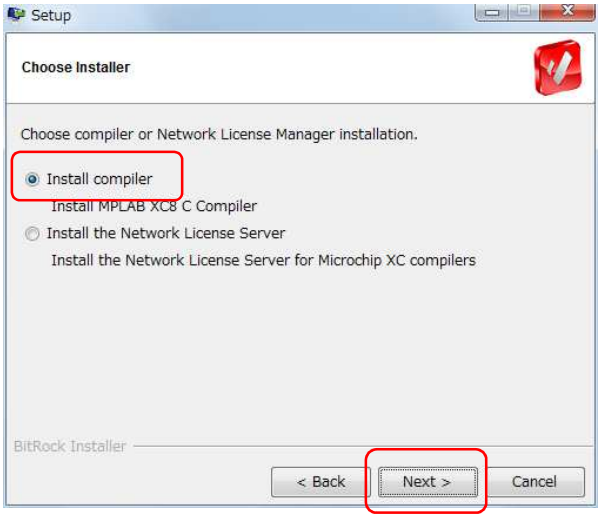
最初の画面が表示されたら『Next >』をクリックします。

③



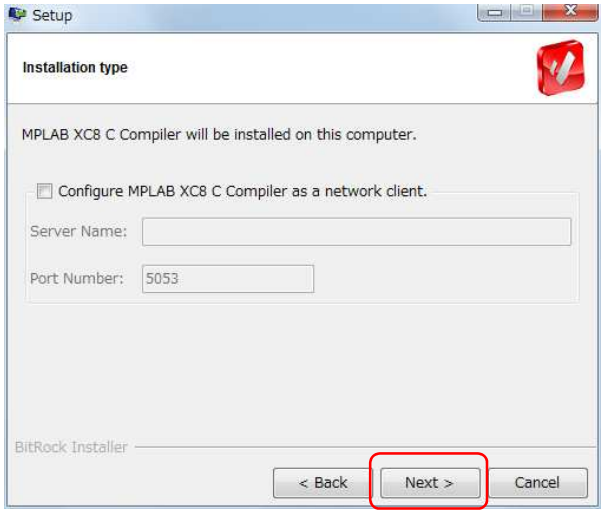
ライセンスに対する確認画面が表示されますので、『I accept the agreement』にチェックを入れ、『Next >』をクリックします。

④



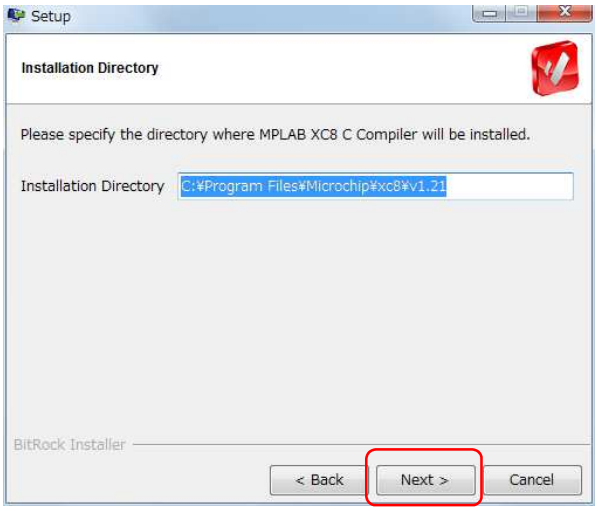
次に『Install compiler』にチェックを入れ、『Next >』をクリックします。

⑤



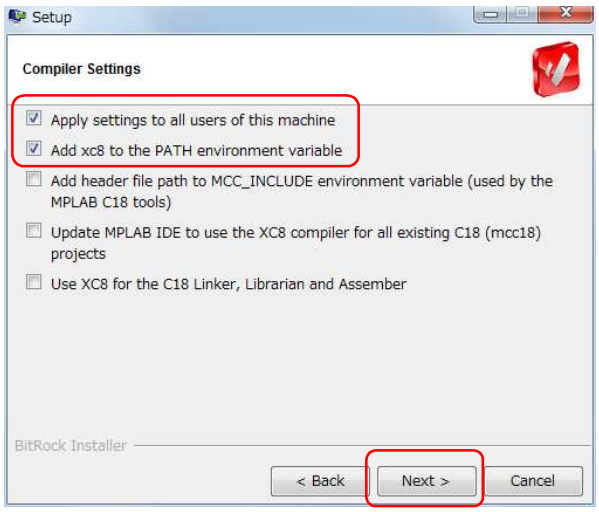
次の画面では何もチェックを入れずに、『Next >』をクリックします。

⑥



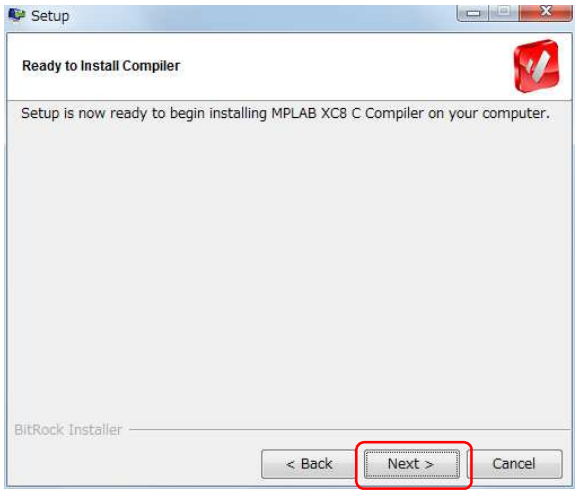
この画面はインストールする場所の設定ですから、特に変更せずに『Next >』をクリックします。

⑦



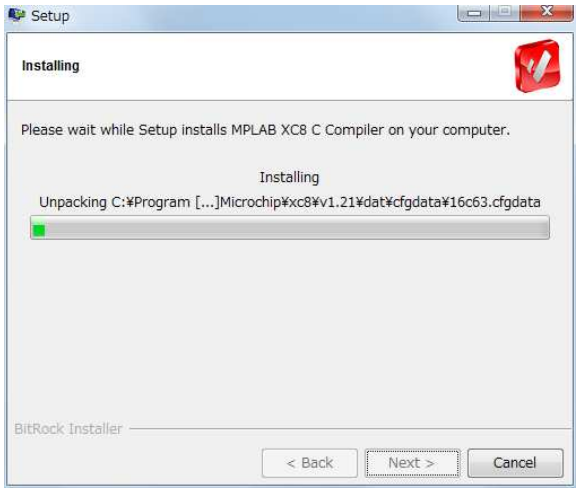
次の画面では『Apply settings to all users of this machine』と『Add xc8 to the PATH environment variable』にチェックを入れ、『Next >』をクリックします。

⑧



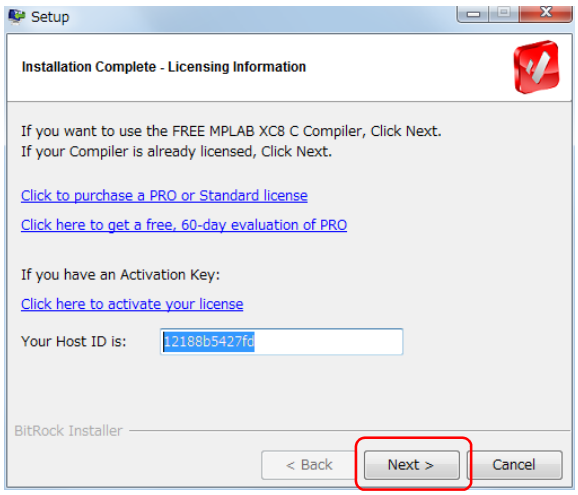
インストールの準備ができましたので『Next >』をクリックします。

⑨



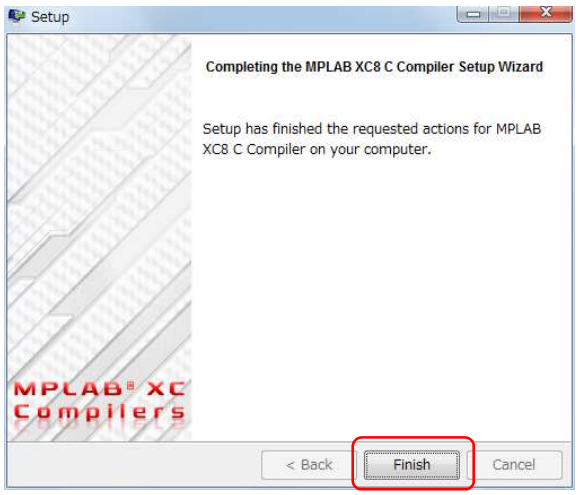
インストール中はバーグラフ表示になり、進捗状況が表示されます。

⑩



この画面が表示されたら『Next >』をクリックします。

⑪



これでインストールは完了です。
『Finish』をクリックし画面を閉じます。

XC コンパイラの種類

XC コンパイラは、プログラムをするマイコンの種類によって使用するものが異なります。
XC コンパイラのダウンロードページに対応表が記載されています。
『PICA Tower』では、使用するマイコンが『PIC16F1827』なので『XC8』をダウンロードしました。
PIC24 シリーズや dsPIC をプログラムするときは『XC16』を、PIC32 シリーズの場合は『XC32』を使用します。

マイクロチップ社製PICとコンパイラの 対応表

	PRO	Standard	Free** 左欄のリンクからダウンロード	C++
PIC 10/12/16/18 MCU	MPLAB® XC 8 ▪ ワークステーション ライセンス ▪ ネットワーク サーバ ライセンス	MPLAB® XC 8 ▪ ワークステーション ライセンス ▪ ネットワーク サーバ ライセンス	MPLAB® XC 8 ▪ ワークステーション ライセンス	MPLAB® XC 8 該当なし
PIC 24 MCU、dsPIC DSC	MPLAB® XC 16 ▪ ワークステーション ライセンス ▪ ネットワーク サーバ ライセンス	MPLAB® XC 16 ▪ ワークステーション ライセンス ▪ ネットワーク サーバ ライセンス	MPLAB® XC 16 ▪ ワークステーション ライセンス	MPLAB® XC 16 該当なし
PIC 32 MCU	MPLAB® XC 32 ▪ ワークステーション ライセンス ▪ ネットワーク サーバ ライセンス	MPLAB® XC 32 ▪ ワークステーション ライセンス ▪ ネットワーク サーバ ライセンス	MPLAB® XC 32 ▪ ワークステーション ライセンス	MPLAB® XC 32 ▪ ワークステーション ライセンス-Free ▪ ワークステーション ライセンス-PRO ▪ ネットワーク サーバ ライセンス-PRO

各コンパイラには『PRO』『Standard』『Free』(XC32 のみ『C++』)があります。
これらの違いはプログラムを作成し機械語に翻訳するときの最適化のレベルが異なります。
『PRO』が最も効率よく最適化されます。最適化率が高いほど、マイコンのプログラムを書き込む際に使用するメモリーの消費量を少なくすることができるので、より複雑なプログラムを書き込むことができるなどのメリットがあります。

それぞれの役割

開発環境を色々と準備してきましたが、それぞれはどのような役割をするのでしょう。

・MPLAB X IDE
MPLAB X IDE は『統合型開発環境』と呼ばれるもので、実際のプログラムファイル(ソースファイル)を作成する『エディタ』、ソースファイルを機械語に翻訳する『コンパイラ』、機械語に翻訳されたデータ(HEX データ)をマイコンに書き込む機能を持っています。その他にもプログラムの検証を行う『デバッガ』の機能も持っています。

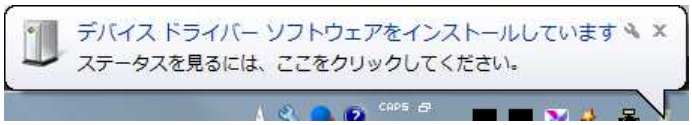
・XC8
MPLAB X IDE の機能の一部として、C 言語で書かれたソースファイルを機械語に翻訳する『コンパイラ』として働きます。

・PICkit3
パソコンとマイコンの橋渡しをし、マイコンに HEX データを書き込みます。

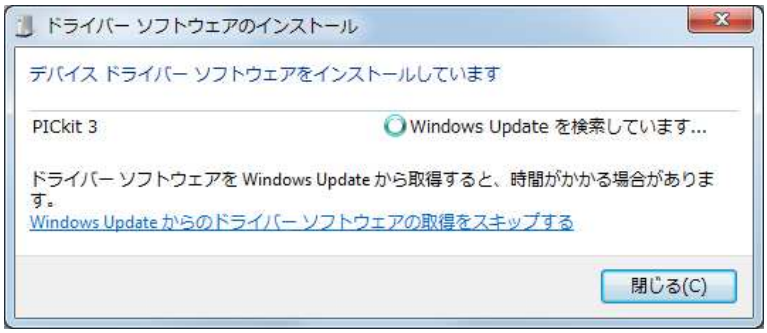
PICkit3 の接続

PICkit3 に付属の USB ケーブルを使用し、PICkit3 をパソコンに接続します。
PICkit3 を接続する USB ポートは、必ずパソコン本体の USB ポートに接続しましょう。
USB ハブなどを通して接続すると、うまく動作しない場合があります。

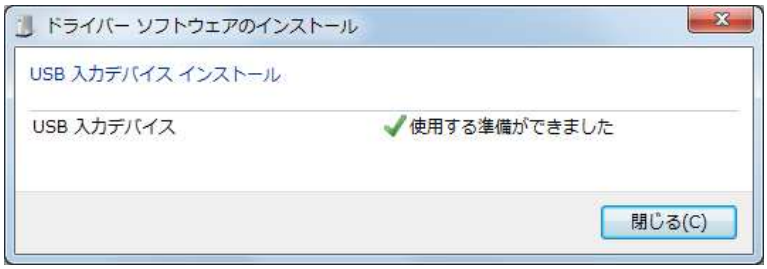
PICkit3 を接続すると、タスクバーに下記の表示が出て自動的にドライバーのインストールが始まります。



この表示をクリックすると、下記のようなインストールの詳細が表示されます。



しばらくするとドライバーのインストールが完了し、下記の画面が表示されます。



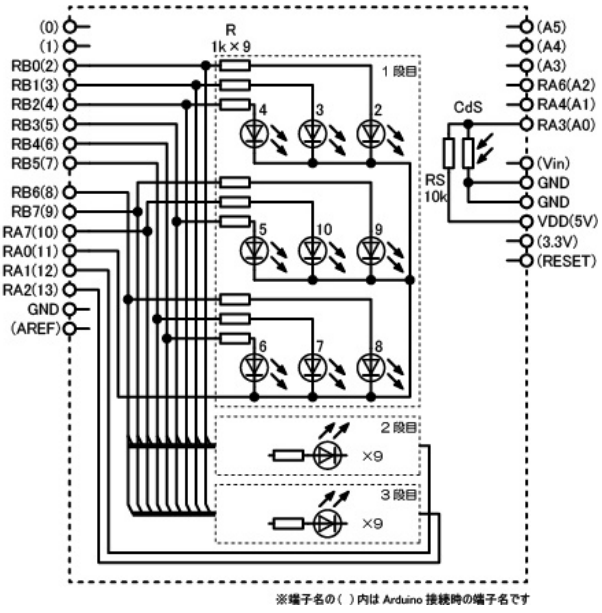
上記表示が出ない場合でも、タスクバーに下記表示が出ればドライバーのインストールは完了です。

2. PICA Tower の回路について

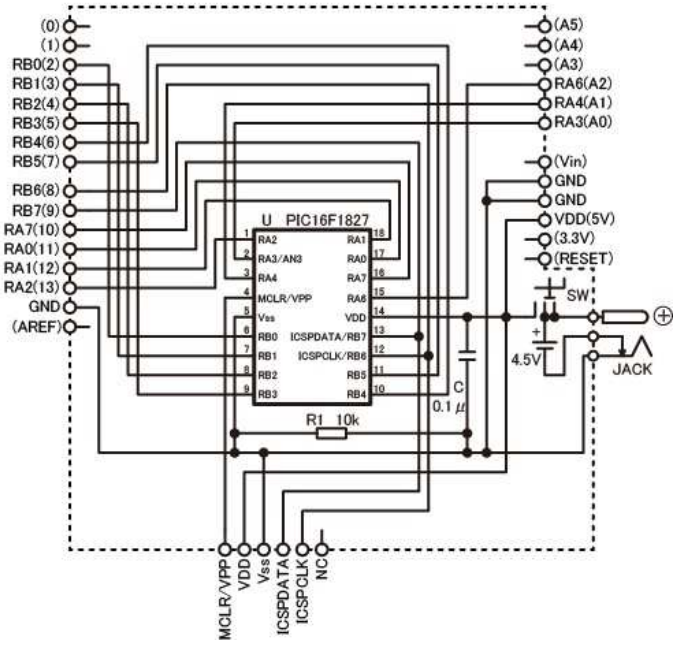
マイコンのプログラミングをするためには、そのマイコンと周辺部品がどのようにつながっているかを知っておく必要があります。
ここでは、PICA Tower の回路がどのような構成になっていて、LED を正しく光らせるにはどのようにすればよいかを説明します。

PICA Tower の回路

PICA Tower は、3 段に重なった LED 部分と、マイコンを搭載した部分が分離できるようになっていて、それぞれの回路は次のようになっています。



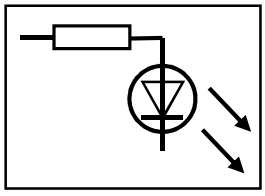
LED 部の回路図



マイコン部の回路図

マイコン部と LED 部の各端子で同じ名前の端子どうしがつながっています。
それでは、まず LED 部分に注目してみましょう。

LED が点灯するためには



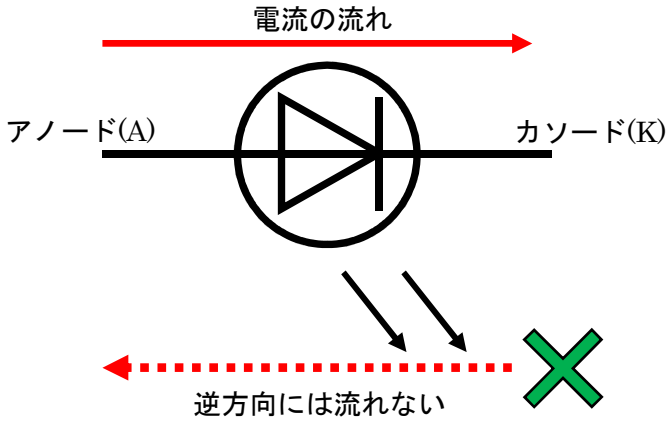
左の図は LED 部の回路から LED の回路を 1 つだけ抜き出したものです。

の記号が LED で、この図の上側の端子を『アノード(A と表します)』、下側の端子を『カソード(K と表します)』といいます。

①電流が流れる方向

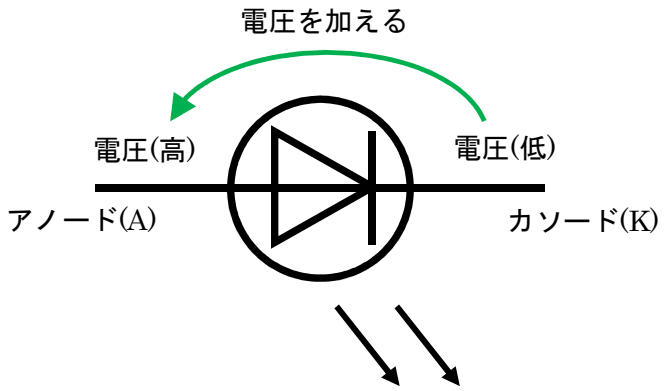
LED は電流を流すことで点灯するのですが、電流はアノードからカソードに向かって流れ、その逆には流

れません。ですから、LED を点灯させるためには、電流がアノードからカソードに流れるような回路にしなければなりません。

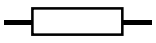


②LED に加える電圧

電子回路に電流を流すためには電圧を加えなければなりません。電子回路の電流の流れは、よく水の流れに例えられます。水は水圧の高い方から低い方へ流れます。これと同じように、電流も電圧の高い方から低い方に流れますので、LED のアノードからカソードに電流が流れるようにアノードとカソードの間にアノードの方の電圧が高くなるように電圧を加える必要があります。



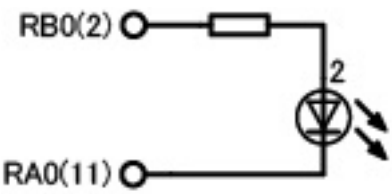
③順方向電流の調整

LED を光らせるためには電流を流さなくてはならないことは説明しましたが、LED には流すことのできる電流の上限が決められています。それ以上の電流を流すと、LED が壊れたり、寿命がとても短くなったりします。PICA Tower の回路を見ると、LED のアノード側  に がつながっています。この部品は『抵抗』で、電流が流れる量を調整する役割があります。

④実際の回路

では、PICA Tower の LED とマイコン部分の回路はどうなっているのでしょう。

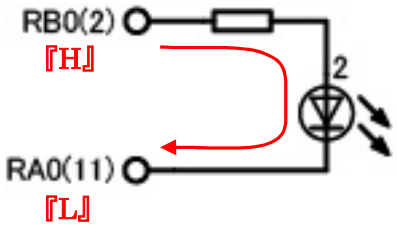
マイコンの RB0 端子だけのつながりを見てみると、抵抗→LED を通って RA0 端子へつながっています。



LED のアノードの電圧を高く、カソードを低くすると点灯しますので、この LED を点灯させるためには、マイコンの RB0 端子の電圧を高く、RA0 端子の電圧を低くなるようにプログラムを作ればよいわけです。

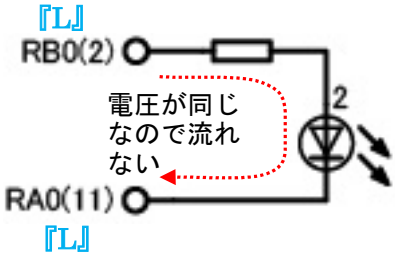
マイコンのポート

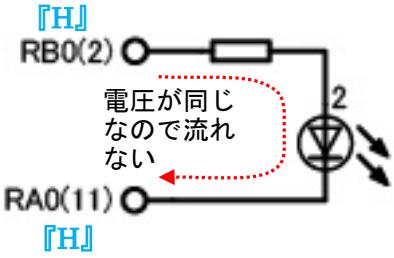
マイコンの端子の電圧を高くしたり低くしたりするとはどういうことでしょう。
マイコンはたくさんの端子を持っていて、それらは入力や出力に設定して使用します。
この端子のことを『ポート』と呼びます。
出力に設定したポートは、プログラムにより自分の好きなポートを『電圧がある状態』や『電圧がない状態』に設定することができます。
『電圧がある状態』とは、ポートから電源のプラスと同じ電圧を出力することで『H(ハイ)』、または『1』と表します。
『電圧がない状態』とは、ポートから電源のマイナスと同じ電圧を出力することで『L(ロー)』、または『0』と表します。
マイコンにつながった電源の電圧が 5V だった場合、『H』に設定したポートは『5V』に、『L』に設定したポートは『0V』になります。
先ほどの PICA Tower の回路の場合、RB0 ポートを『H』に、RA0 ポートを『L』にすると、LED のアノード側の電圧が高くなり、カソード側の電圧が低くなるため、LED に順方向電流が流れ点灯するのです。

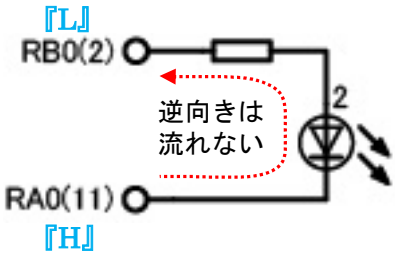


では、LED を消灯するためにはどうすればよいのでしょう。
LED のアノードとカソードの電圧を同じにするか、カソードの方をアノードより高い電圧にすればよいので、

- RB0、RA0 の両方を『L』
- RB0、RA0 の両方を『H』
- RB0 を『L』、RA0 を『H』のいずれかにすれば LED は消灯します。

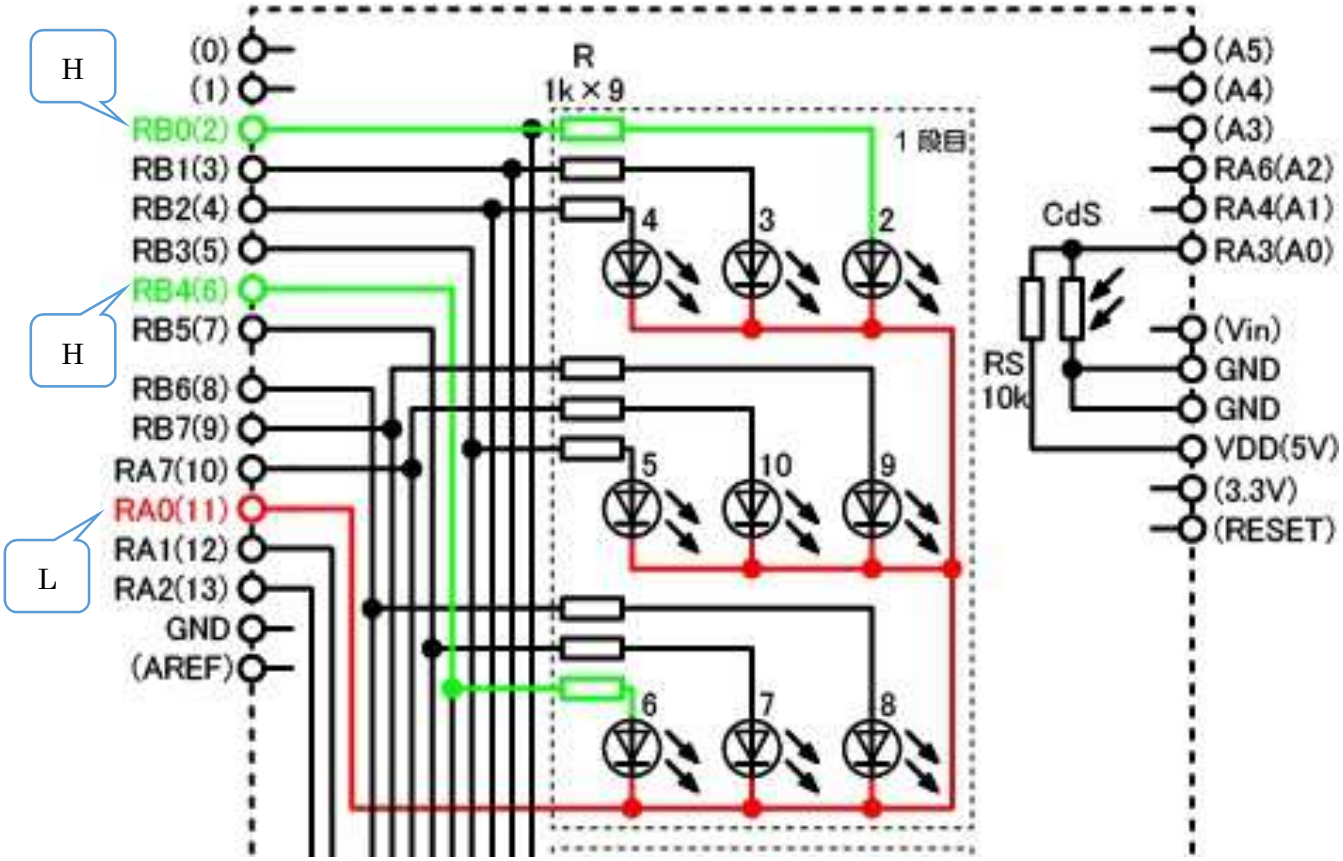




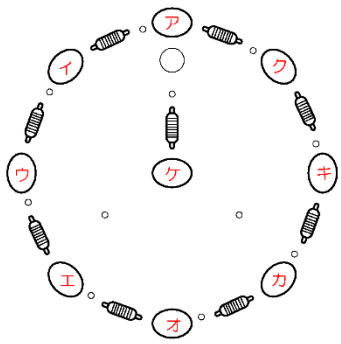


PICA Tower の LED は全て抵抗とつながっています。それぞれ接続先のポートが違うだけです。
縦の同じ位置にある LED のアノードはそれぞれ RB0～RB7、RA7 につながっています。
また、同じ段にある LED のカソードは 1 段目が RA0、2 段目のカソードが RA1、3 段目のカソードが RA2 につながっています。

例えば、2 段目の 5 番の LED を点灯させるためには、RB3 を『H』、RA1 を『L』にすれば OK です。
また、RB0～RB7、RA7 を全て『H』、RA0 を『L』にして RA1 と RA2 を『H』にすれば、1 段目の LED を全て点灯させることができますし、この状態で RA0 を『H』、RA1 を『L』、RA2 を『H』にすると 2 段目だけが点灯、さらに RA0 と RA1 を『H』、RA2 を『L』にすると 3 段目だけが光りますので、この RA0～RA2 のポートの状態の切り替えを順番に行うようにプログラムすると、LED が流れるように点灯させることができるのです。



上の図のように 1 段目だけを見てみましょう。
1 段目の LED のカソードは全てポート RA0 につながっています。
例えば、RB0 と RB4 を『H』に、RA0 を『L』にすると、1 段目の 2 番と 4 番の LED が点灯する。



各 LED のアノード側とカソード側の対応は下表のとおりです。
例えば『RB4』を『H』、『RA1』を『L』にすると、左図の『2 段目のオ』に対応するの LED が点灯します。

		アノード側								
		RB0	RB1	RB2	RB3	RB4	RB5	RB6	RB7	RA7
カソード側	RA0	1 段目 ア	1 段目 イ	1 段目 ウ	1 段目 エ	1 段目 オ	1 段目 カ	1 段目 キ	1 段目 ク	1 段目 ケ
	RA1	2 段目 ア	2 段目 イ	2 段目 ウ	2 段目 エ	2 段目 オ	2 段目 カ	2 段目 キ	2 段目 ク	2 段目 ケ
	RA2	3 段目 ア	3 段目 イ	3 段目 ウ	3 段目 エ	3 段目 オ	3 段目 カ	3 段目 キ	3 段目 ク	3 段目 ケ

PIC16F1827 について

マイコンのプログラミングをする時、そのマイコンの各端子の役割はどうなっているのか、マイコンがどのような機能を持っているのか、その機能を使用するためにはどのような設定が必要なのかなど、マイコンの仕様について知っておく必要があります。

Microchip 社のホームページから、PICA Tower で使用する『PIC16F1827』のデータシートをダウンロードしておきましょう。

マイコンのデータシートは英語で書かれていることがほとんどですが、慣れてくると書いてある意味がだんだん分かってくると思います。

データシートのはじめの方に下記の表が記載されています。

この表は、マイコンが持っている機能が表してあり、16F1827 では『I/O(入出力端子)』が最大 16 コ、『10-bit ADC(分解能 10 ビットのアナログ-デジタル・コンバータ)』が最大 12 チャンネル設定できるなどが分かります。

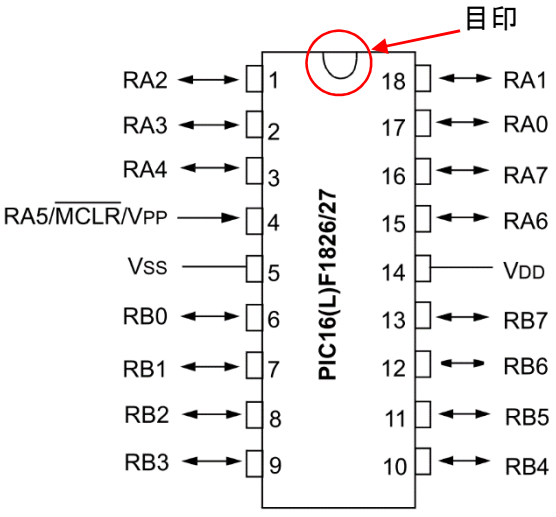
PIC16(L)F1826/27 Family Types

Device	Program Memory	Data Memory		I/O's(1)	10-bit ADC (ch)	CapSense (ch)	Comparators	Timers (8/16-bit)	EUSART	MSSP	ECCP (Full-Bridge)	ECCP (Half-Bridge)	CCP	SR Latch
	Words	SRAM (bytes)	Data EEPROM (bytes)											
PIC16LF1826	2K	256	256	16	12	12	2	2/1	1	1	1	—	—	Yes
PIC16F1826	2K	256	256	16	12	12	2	2/1	1	1	1	—	—	Yes
PIC16LF1827	4K	384	256	16	12	12	2	4/1	1	2	1	1	2	Yes
PIC16F1827	4K	384	256	16	12	12	2	4/1	1	2	1	1	2	Yes

Note 1: One pin is input only.

欄外に記載されていることが重要な情報である場合が多くあります。
ここでは、『I/O』16 コのうち、1 つだけは入力だけで出力には設定できないことが記載されています。

その下に記載してある図は、マイコンの端子位置を表した図です。



マイコンの端子は上から見て、目印の位置から反時計回りに番号が割り振られています。

例えば、この 16F1827 では 5 番ピンが Vss、VDD が 14 番ピンと端子番号と端子名が対応します。

左図では各端子に矢印が描かれています。この矢印は各端子が設定できる入出力を表していて、 ⟷ は入力、出力の両方に設定できることを表しています。

4 番ピンの端子は ⟶ と、1 方向の矢印で入力にしか設定できないことを表しています。これが上記の『Note 1』の内容です。

前ページの図は、各端子の役割が『I/O』についてのみ記載されています。各端子は『I/O』以外の役割も兼用しており、マイコンの機能を設定する『レジスタ』を書き換えることにより様々な機能を実現することができます。

TABLE 1: 18/20/28-PIN SUMMARY (PIC16(L)F1826/27)

I/O	18-Pin PDIP/SOIC	20-Pin SSOP	28-Pin QFN/UQFN	ANSEL	A/D	Reference	Cap Sense	Comparator	SR Latch	Timers	CCP	EUSART	MSSP	Interrupt	Modulator	Pull-up	Basic
RA0	17	19	23	Y	AN0	—	CPS0	C12IN0-	—	—	—	—	SDO2(2)	—	—	N	—
RA1	18	20	24	Y	AN1	—	CPS1	C12IN1-	—	—	—	—	$\overline{SS2}$(2)	—	—	N	—
RA2	1	1	26	Y	AN2	VREF-DACOUT	CPS2	C12IN2-C12IN+	—	—	—	—	—	—	—	N	—
RA3	2	2	27	Y	AN3	VREF+	CPS3	C12IN3-C1IN+C1OUT	SRQ	—	CCP3(2)	—	—	—	—	N	—
RA4	3	3	28	Y	AN4	—	CPS4	C2OUT	SRNQ	T0CKI	CCP4(2)	—	—	—	—	N	—
RA5	4	4	1	N	—	—	—	—	—	—	—	—	$\overline{SS1}$(1)	—	—	Y(3)	$\overline{MCLR}$, VPP
RA6	15	17	20	N	—	—	—	—	—	—	P1D(1) P2B(1,2)	—	SDO1(1)	—	—	N	OSC2 CLKOUT CLKR
RA7	16	18	21	N	—	—	—	—	—	—	P1C(1) CCP2(1,2) P2A(1,2)	—	—	—	—	N	OSC1 CLKIN
RB0	6	7	7	N	—	—	—	—	SRI	T1G	CCP1(1) P1A(1) FLT0	—	—	INT IOC	—	Y	—
RB1	7	8	8	Y	AN11	—	CPS11	—	—	—	—	RX(1,4) DT(1,4)	SDA1 SDI1	IOC	—	Y	—
RB2	8	9	9	Y	AN10	—	CPS10	—	—	—	—	RX(1) DT(1) TX(1,4) CK(1,4)	SDA2(2) SDI2(2) SDO1(1,4)	IOC	MDMIN	Y	—
RB3	9	10	10	Y	AN9	—	CPS9	—	—	—	CCP1(1,4) P1A(1,4)	—	—	IOC	MDOUT	Y	—
RB4	10	11	12	Y	AN8	—	CPS8	—	—	—	—	—	SCL1 SCK1	IOC	MDCIN2	Y	—
RB5	11	12	13	Y	AN7	—	CPS7	—	—	—	P1B	TX(1) CK(1)	SCL2(2) $\overline{SCK2}$(2) $\overline{SS1}$(1,4)	IOC	—	Y	—
RB6	12	13	15	Y	AN5	—	CPS5	—	—	T1CKI T1OSI	P1C(1,4) CCP2(1,2,4) P2A(1,2,4)	—	—	IOC	—	Y	ICSPCLK/ ICDCLK
RB7	13	14	16	Y	AN6	—	CPS6	—	—	T1OSO	P1D(1,4) P2B(1,2,4)	—	—	IOC	MDCIN1	Y	ICSPDAT/ ICDDAT
VDD	14	15,16	17,19	—	—	—	—	—	—	—	—	—	—	—	—	—	VDD
VSS	5	5,6	3,5	—	—	—	—	—	—	—	—	—	—	—	—	—	VSS

Note 1: Pin functions can be moved using the APFCON0 or APFCON1 register.
2: Functions are only available on the PIC16(L)F1827.
3: Weak pull-up always enabled when MCLR is enabled, otherwise the pull-up is under user control.
4: Default function location.

データシート 6 ページの表が各端子に割り付けられた機能一覧を表しています。

例えば、1 番ピンは『I/O』では入出力の『RA2 ポート』の役割ですが、AD コンバータの設定をすればアナログ入力端子の『AN2 ポート』の役割をする端子に変わります。

また、Cap Sense の設定を行えば、タッチセンサーの信号を入力する端子の『CPS2』にすることもできます。

このように、マイコンの機能を設定し、その機能がうまく働くようにデータのやり取りを行う命令を作ることが『プログラミング』なのです。

3. C 言語のプログラムの書き方

C 言語でプログラムを作成するためには、その書き方や命令など、幾つか知っておかなければならないことがあります。
ここではそれらの解説をしながら、C 言語でのプログラムの書き方について記載します。

文法の基本

C 言語のプログラムは、次の例のように書きます。

```
{
a = 4;          /*a に 4 を代入*/
b = c+d;        //c と d を足した結果を b に代入
e = g;          //e に g を代入
}
```

コメント

各行の後方に書かれている部分で　/*～*/　で囲まれた部分、または　//　以降の文は『コメント』です。
コメントはプログラムとは直接関係がなく、その内容を補足説明などする場合に記入する文章で、日本語で書くことも可能です。
コメント部分はプログラム実行時には空白として扱われますので、その内容は無視されます。
ちなみに、改行や TAB で挿入した部分も空白として扱われ、プログラム実行時は無視されます。ただし、全角の空白をコメント行以外に書いてしまうとエラーになるので注意してください。
/*～*/　で書いたコメントは　/*　と　*/　で囲まれた部分が全てコメントと認識されます。これは複数行にまたがって記入されていても OK です。
//　で書いたコメントは、　//　から行末までがコメントとして認識されます。

文

a = 4; のように書かれた部分は『文』で、途　；　中に(セミコロン)が付けられていますが、これは『その命文がここまでですよ』ということを表しています。
例えば

```
a = 4
b = c+d;
```

と、上の行末に　;　を付け忘れた場合、プログラムは

```
a = 4 b = c+d;
```

と認識され、エラーになってしまいます。
これもよくあるミスです。また　：　(ダブルコロン)と間違えやすいので注意しましょう。

ブロック

{　}　(波カッコ)で囲まれた部分を『ブロック』といいます。ブロックは文をまとめるために使用し、関数などの処理内容の手順を列記する時に使用します。プログラム中に『始まりの{　』があれば、それに対応する『終わりの }』が無くてはなりません。時々{　}の数が合わずにエラーになってしまうことがありますので注意しましょう。

定数、変数、データ型

定数と変数

C 言語であつかうデータは大きく『定数』と『変数』の 2 種類に分けられます。

例えば、　a = 5;　a = 10;　と書かれている場合、『a』は場合によって 5 や 10 になるので『変数』。

5 や 10 は変わらないので『定数』です。

定数は 10 進数のほか、2 進数や 16 進数で書かれることがあります。定数を書いた時、それが何進数かを表すために、次のような書き方をします。

```
10 進数・・・56　何も付けずそのまま表記
2 進数・・・0b00111000　最初に『0b』を付ける
16 進数・・・0x38　最初に『0x』を付ける

あまり使用しませんが、8 進数で書くこともできます
8 進数・・・070　最初に『0』を付ける
```

変数などに使用できる文字

変数や処理の手順を示す『関数』などに付ける『名前』のことを『識別子』と言います。

識別子には『半角のアルファベット』『半角の数字』『_(半角アンダーバー)』のみ使用可能ですが、1 文字目に数字を使うことはできません。また、アルファベットの大文字と小文字は区別されますので、ABC と abc は違うものとして認識されます。

ただし、あらかじめシステムが使用する単語がいくつか決められていて、その単語は識別子として使用することができません。その単語のことを『予約語』または『キーワード』と言います。

データ型

変数は、その変数が表す数値の範囲によって『データ型』を選ぶ必要があります。

変数の値の範囲を超えた計算を行うと正しい計算結果が得られなかったり、マイコンが使用するメモリーを無駄に消費するこがあります。

データ型と値の範囲は次のとおりです。

データ型	ビット数	値の範囲	データ型	ビット数	値の範囲
bit	1	0 ～ 1	short long	24	−8388608 ～ 8388607
signed char	8	−128 ～ 127	unsigned short long	24	0 ～ 16777215
unsigned char	8	0 ～ 255	long	32	−2147483648 ～ 2147483647
int	16	−32768 ～ 32767	unsigned long	32	0 ～ 4294967295
unsigned int	16	0 ～ 65535	float	24	-3.4×10^{38} ～ 3.4×10^{38}

変数を使うときは、あらかじめそのプログラムの中で、『これは変数ですよ！』と宣言します。

変数宣言の書き方は『変数のデータ型』と『変数』、その変数の『初期値』を書く場合などいろいろあり、下記はその一例です。

```
signed char a;              //a は signed char 型のデータ範囲を取り扱う変数だと宣言
int b,c;                    //『,』で区切って続けて書いても OK。b も c も int 型の変数。
int d=10;                   //初期値を書いても OK。d は int 型で、初期値に 10 が代入される。
```

※char と書いた場合、初期設定では unsigned char と認識されます。

演 算

定数、変数の足し算や引き算などの計算や、それぞれの比較などを行う時に『演算子』を使用します。

代入

代入演算子は『=』の右側の項の内容を左側の項に書き込みます

演算子	使用例	意味
=	a = 10;	a に 10 を代入するので、a は 10 になります
	b = 1+2;	b に 1+2 の結果を代入するので、b は 3 になります
	a = b;	a に b の値を代入します

『=』の左右が入れ替わると意味が変わってしまいますので、注意しましょう。

```
a = 10;                 //a は 10 になります
b = 5;                  //b は 5 になります
a = b;                  //a に b を代入するので、a は b と同じく 5 になります
```

```
a = 10;                 //a は 10 になります
b = 5;                  //b は 5 になります
b = a;                  //b に a を代入するので、b は a と同じく 10 になります
```

また、次のように書くとエラーになります。これは定数はその値しか取り得ないので、定数に他の値を代入することができないためです。

```
10 = a;                 //10 は定数なので a は代入できない。エラーになる
```

算術演算子

加減乗除などの計算をする演算子です。

演算子の種類と機能、使い方例は下記のとおりです。

演算子	使用例	意味
+	a = 1+2	足し算をします。1+2 の結果を a に代入するので、a は 3 になります。
-	b = 3-1	引き算をします。3-1 の結果を b に代入するので、b は 2 になります。
*	c = 2*2	掛け算をします。2×2 の結果を c に代入するので、c は 4 になります。
/	d = 8/4	割り算をします。8÷4 の結果を d に代入するので、d は 2 になります。
%	e = 7%2	モジュロ算という計算をします。モジュロ算は割り算をした時の『余り』を求める計算です。使用例の場合 7÷2 の余りを e に代入するので、e は 1 になります。

算術演算子は定数どうしだけでなく、定数と変数、変数どうしなどの計算もできます。

```
a = 1+2;                //a は 3 になります。
b = a*2;                //b は 3×2 で 6 になります。
c = b-a;                //b は 6、a は 3 なので、c は 6-3 で 3 になります。
```

計算結果とデータ型

int 型の変数 d に次の計算をすると、d の値はどうなるでしょう？

d = 10/4　　この場合、d は『2』になります。なぜなら int 型の値の範囲は『−32768 ～ 32767 の**整数**』だからです。

小数点以下を含んだデータを取り扱うには、データ型を float にしましょう。

インクリメント、デクリメント

インクリメントは『1 を足す』こと、デクリメントは『1 を引く』ことです。

演算子	使用例	意味
++	++a	a の値に 1 を足して、その値を a に書き込みます。
--	--b	b の値から 1 を引いて、その値を b に書き込みます。

例えば、

```
a = 10;          //a は 10 になります。
b = 5;           //b は 5 になります。
c = ++a;         //a は 1 を足した 11 になり、それを c に代入するので C も 11 になります。
d = --b;         // b は 1 を引いた 4 になり、それを d に代入するので d も 4 になります。
```

インクリメント、デクリメントは『a++』『b--』のように変数の後ろに書くこともできます。しかし、この場合、前に書いた時と計算をするタイミングが変わります。

```
a = 10;          //a は 10 になります。
b = 5;           //b は 5 になります。
c = a++;         //先に c に a を代入してから a をインクリメントする。
                 //c は 10 になり、a は 11 になる。
d = b--;         //先に d に b を代入してから b をデクリメントする。
                 //d は 5 になり、b は 4 になる。
```

比較演算子

2 つの定数や変数の関係を調べるための演算子です。

調べた結果、その関係が正しければ真、間違っていれば偽といいます。

真は『1』、偽は『0』で表すこともあります。

演算子	使用例	意味
>	a > b	a が b より大きければ真、小さければ偽になります。
>=	a >= b	a が b と同じか大きければ真、小さければ偽になります。
<	a < b	b が a より大きければ真、小さければ偽になります。
<=	a <= b	b が a と同じか大きければ真、小さければ偽になります。
==	a == b	a と b が等しければ真、異なれば偽になります。
!=	a != b	a と b が異なれば真、等しければ偽になります。
||	(a>b) || (c<d)	『||』の左右の関係式で、どちらか片方でも真ならば真。どちらも偽ならば偽になります。例では a>b と c<d のどちらか一方が真であれば真になります。
&&	(a>b) && (c<d)	『&&』の左右の関係式で、どちらも真ならば真。片方でも偽、または両方とも偽ならば偽になります。例では a>b と c<d の両方が真の時だけ真になります。
!	!(a>b)	『!』の後に書かれた関係式が真ならば偽、偽ならば真になります。つまり、関係式の真偽の逆になります。例では a>b が真ならば偽に、偽ならば真になります。

では、次の式の場合はどうなるでしょうか。

```
a = 10;                 //a は 10
b = 5;                  //b は 5
c = 3;                  //c は 3
d = 7;                  //d は 7

e = (a>b);              //a は 10、b は 5 なので a>b は真。真は『1』なので、e は 1 になる。
f = (b==c);             //b と c は等しくないので b==c は偽。偽は『0』なので、f は 0 になる。
g = ((a>b) && (c<d))    // a は 10、b は 5 なので a>b は真。c は 3、d は 7 なので c<d も真。
                        //2 つの関係式が両方ともに真なので(a>b) && (c<d)は真。g は『1』にな
る
```

上の例のように、式の中に(　)がある場合は、(　)の中を先に計算します。

この比較演算子は、あとで説明する『制御文』の『繰り返し』や『条件分岐』でよく使用されます。

論理演算子

2 つの定数や変数を論理演算する時に使う演算子です。

演算子	使用例	意味
&	a & b	a と b を AND で論理演算した値になる。AND は『論理積』とも言う。
|	a | b	a と b を OR で論理演算した値になる。OR は『論理和』とも言う。
^	a ^ b	a と b を XOR で論理演算した値になる。XOR は『排他的論理和』とも言う。
~	~a	a の各ビットを反転(NOT)した値になる。

ビットごとの操作はその値を 2 進数に変換して考えるとわかりやすいです。

例えば a と b に入る値を 16 進数で書いてみました。

```
a = 0x56;              //a を 2 進数で表すと 0b01010110
b = 0x4A;              //b を 2 進数で表すと 0b01001010

c = (a&b);             //c は 0x42 になる
d = (a|b);             //d は 0x5E になる
e = (a^b);             //e は 0x1C になる。
f = ~a                 //f は 0xA9 になる。
```

これを 2 進数で考えてみましょう。

それぞれの変数の値は 2 進数で表すと

a = 0x56 = 0b01010110

b = 0x4A = 0b01001010

となります。

まず 2 つの値を上下に並べて書いてみます。

	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	操作内容
a	0	1	0	1	0	1	1	0	
b	0	1	0	0	1	0	1	0	
AND	0	1	0	0	0	0	1	0	各ビットを比較し、両方とも『1』のときだけ『1』。その他の場合は『0』になる。
OR	0	1	0	1	1	1	1	0	各ビットを比較し、少なくとも片方が『1』であれば『1』。両方とも『0』なら『0』になる。
XOR	0	0	0	1	1	1	0	0	各ビットを比較し、それぞれのビットの値が異なるときは『1』。 同じ値の場合は『0』になる。
NOT	1	0	1	0	1	0	0	1	各ビットの値が『1』なら『0』、『0』なら『1』になる。

2 進数で考えるととてもわかりやすくなりましたね。

AND、OR、XOR はどのようなときに使うのでしょう？

・AND
任意のビットを取り出すときに使います。
例)
a= 01101100
AND 11110000
01100000

取り出したいビットを『1』そうでないビットを『0』にすると『1』の部分だけそのまま取り出せる。

・OR
任意のビットを『1』にするときに使います。
例)
a= 01101101
OR 00111100
01111101

『1』にしたいビットを『1』そうでないビットを『0』にすると『1』の部分は『1』になる。『0』の部分はそのまま。

・XOR
任意のビットを反転にするときに使います。
例)
a= 01101101
XOR 00001111
01100010

反転したいビットを『1』そうでないビットを『0』にすると『1』の部分は反転する。『0』の部分はそのまま。

シフト演算子

定数や変数の値を指定されたビット数分、左または右にずらす処理をする演算子です。

演算子	使用例	意味
<<	a << b	a を左に b ビットシフトした値になる。
>>	a >> b	a を右に b ビットシフトした値になる。

次のような場合では

```
a = 0x56;              // 2 進数で表すと 0b01010110
b = 2;                 //ずらすビット数は 2
c = (a<<b);            //c は 0x58 になる
d = (a>>b);            //d は 0x15 になる
```

この場合も 2 進数で考えましょう。

まず、左シフトの場合は、

	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	操作内容
a	0	1	0	1	0	1	1	0	
左シフト	0	1	0	1	1	0	0	0	今回の例では 2 ビット左にずらす。 新たに右側に追加される値は『0』。

新たに追加

次に右シフトの場合は、

	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	操作内容
a	0	1	0	1	0	1	1	0	
右シフト	0	0	0	1	0	1	0	1	今回の例では 2 ビット右にずらす。 新たに左側に追加される値はその値のデータ型と最上位ビットの値により変わります。

新たに追加

※データ型がマイナスを扱える範囲の場合で、最上位ビットが『1』の時、新たに追加される値は『1』。

変数計算後に同じ変数に代入する演算子

例えば a という変数に 10 を足し、その結果を変数 a に代入する式は、

```
a = (a+10);
```

となります。

このような場合に便利な書き方があります。

上記の場合

```
a +=10;
```

と書くこともできます。

演算子	使用例	同じ処理の式	説明
+=	a+=b	a=(a+b)	a に b を足して、その結果を a に代入する。
-=	a-=b	a=(a-b)	a から b を引いて、その結果を a に代入する。
*=	a*=b	a=(a*b)	a と b をかけて、その結果を a に代入する。
/=	a/=b	a=(a/b)	a を b で割って、その結果を a に代入する。
%=	a%=b	a=(a%b)	a を b で割った余りを a に代入する。
&=	a&=b	a=(a&b)	a と b を AND し、その結果を a に代入する。
|=	a|=b	a=(a|b)	a と b を OR し、その結果を a に代入する。

^=	a^=b	a=(a^b)	a と b を XOR し、その結果を a に代入する。
<<=	a<<=b	a=(a<<b)	a を b ビット分左にシフトして、その結果を a に代入する。
>>=	a>>=b	a=(a>>b)	a を b ビット分右にシフトして、その結果を a に代入する。

演算の順番

算数の計算をする時、同じ式の中にいくつも加減乗除の計算がある場合、足し算や引き算よりも掛け算や割り算を先に、また(　)があればその中を先に計算するという決まりがありますね。

それと同じように、定数や変数の演算にも優先順位が決まっています。

優先順位	演算子
高い ↑	( )
	! ~ ++ --
	* / %
	+ -
	<< >>
	< <= > >=
	== !=
	&
	^
	|
	&&
	||
	= += -= *= /= %= &= ^= |= <<= >>=
低い ↓	,

制 御 文

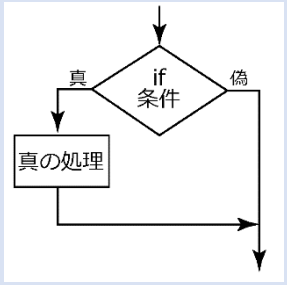
定数や変数の比較をし、その真偽によってプログラムで実行する処理の流れを変えたり、同じ処理を繰り返し行うなどの制御ができます。

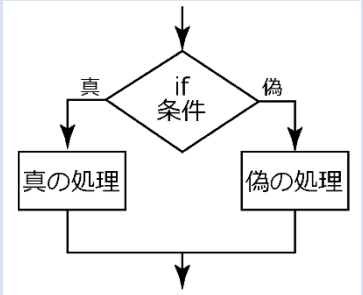
if 制御文

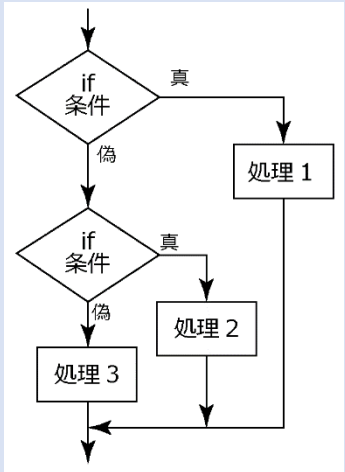
if を使った制御文は比較した結果により、処理の流れを変える時に使用します。

次の 3 通りの使い方があります。if 文は、その処理の中に if 文を記入する、多段 if 文を作ることもできます。

使い方	説明	使用例	フローチャート

if(条件)
{真の場合の処理}
	(条件)が真ならば、{真の場合の処理}を行う。	if(a>=5){
		b=10;
		C=b+2;
		}
			

if(条件)
{真の場合の処理}
else
{偽の場合の処理}
	(条件)が真ならば、{真の場合の処理}を行い、そうでなかったら{偽の場合の処理}を行う。	if(a>=5){
		b=10;
		C=b+2;
		}
		else{
		b=5;
		c=b-2;
		}
			

if(条件 1)
{処理 1}
else if(条件 2)
{処理 2}
else
{処理 3}
	(条件 1)が真ならば{処理 1}を行う。
	(条件 1)がそうではなかったら(条件 2)を調べ、真なら{処理 2}そうでなかったら{処理 3}を行う。
		if(a>=5){
		b=10;
		C=b+2;
		}
		else if(a<5){
		b=5;
		c=b-2;
		}
		else{
		b=1
		}
			

間違えやすいポイント

制御文を書く時に間違えやすいポイントがあります。

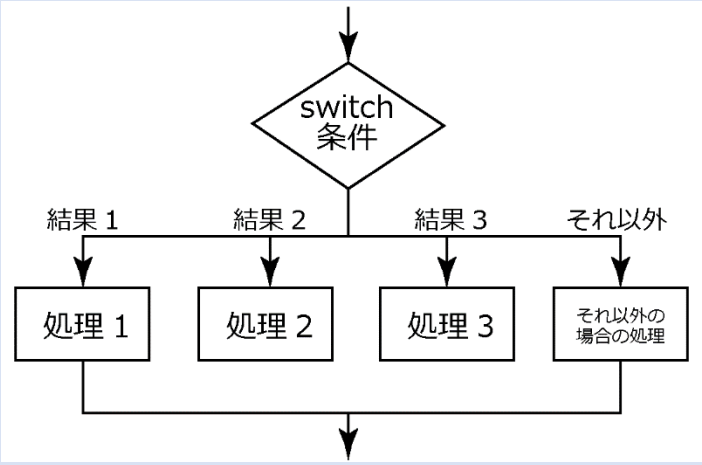
- **条件を書くときに使う演算子に注意！**
  a==0 と書かなくてはならないところを a=0 と書いてしまうと a に 0 を代入するという文になってしまい、期待した通りの処理をされなくなります。条件判断ではその値が 0 の時だけ『偽』、0 以外は全て『真』と判断します。
- **『;』を付けるとそこで終わり！**
  制御文の文字(if や else など)の後には『;』は必要ありません。{ }で囲った処理内の文の後には必ず必要ですが、if(a==1); {b=0;} と書くと、if 文の処理が {b=0;} の前で終わってしまい、{b=0;} は if の文として実行されません。if(a==1) {b=0;}; else{b=1;} と書くと、if(a==1) {b=0};で if 文の処理が終わってしまうことになり、その後の else に対応する if がなくなってしまうことになり、エラーになってしまいます。
  『;』は『そこでその文の処理が終わるマーク』とおぼえておくとよいでしょう。

switch 制御文

条件の内容により分岐先をいくつも変えるような場合に使用するのが switch 文です。

使い方	説明	使用例

switch(条件)
{
case 値 1:
値 1 の場合の処理;
break;
 	
case 値 2:
値 2 の場合の処理;
break;
 	
case 値 3:
値 3 の場合の処理;
break;
・
・
default:
条件結果がどれにも当てはまらない場合の処理;
 	
}
	(条件)の内容が値 1 に等しい場合は、値 1 の場合の処理を行う。
	
	(条件)の内容が値 2 に等しい場合は、値 2 の場合の処理を行う。
	
	(条件)の内容が値 3 に等しい場合は、値 3 の場合の処理を行う。
	・
	・
	・
	・
	・
	(条件)の内容が値のどれにも当てはまらない場合には『default』に書いた処理を行う。
		int a;
		int b;
		
		switch(a)
		{
		case 1:　　//a が 1 の時
		b=10;
		break;
		
		case 3:　　//a が 3 の時
		b=20;
		break;
		
		default:　　//a がそれ以外の時
		b=0;
		}

フローチャート		

間違えやすいポイント

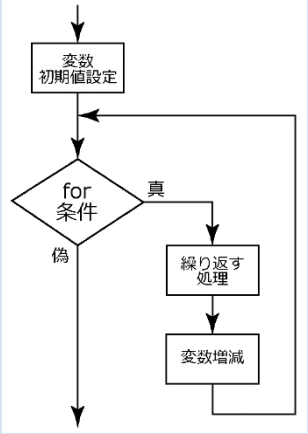
- case 値: 文の後ろは『;(セミコロン)』ではなく『:(ダブルコロン)』です。
- case 値: のあとの処理文の後には break; が必要！
  break; が無いと、条件判断に戻らず、次の行の処理を行うことになりますので、上の例で case 1: に対応する break; が無かったとすると、case 1 の処理後、続けて case 2 の処理をしてしまいます。
- case 値: の『値』には変数は使用できません！　定数、または定数の計算式のみ使用できます。

switch(条件)
{
case 値 1:
case 値 3:
値 1・3 の場合の処理;
break;
 	
case 値 2:
値 2 の場合の処理;
break;
}
	条件判断の結果の値が異なる場合でも同じ処理をさせたいときには、左記のように一つの処理に複数の『case 値:』を書くこともできます。

for 制御文

for を使うと、同じ処理を回数を数えながら繰り返すことができます。

使い方	説明	使用例	フローチャート

for(変数の初期値;条件;増減処理)
{
繰り返す処理;
}
	変数の初期値を設定し、その変数が条件の真偽を判断する。
	条件が真の場合は『繰り返す処理』を実行した後、変数の増減を行い再度条件判断を行う。
	変数が条件判断が真である限り『繰り返す処理』を行い、条件判断が偽になると繰り返しから抜ける。
		int a;
		int b=10;
		
		for(a=0;a<=5;++a)
		{
		--b;
		}
			

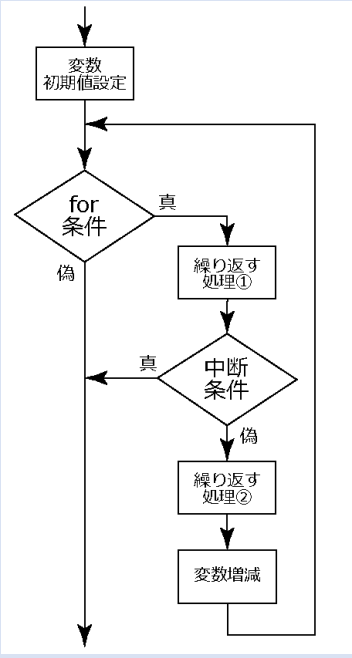
for 文は数を数えながら繰り返すだけではなく、条件が成立するならば繰り返すという使い方もできます。

使い方	説明	使用例

for(変数の初期値;条件;変数を変化させる処理)
{
繰り返す処理;
}
	前述の例の『変数の増減』が『変数の変化処理』に変わった書き方。
	処理は同じで『条件』が真である限り『繰り返す処理』を実行する。
	for の後の(　)内の『変数の初期化』と『変数を変化させる処理』は『,』で区切ることでいくつも書くことができる。
		int a,b,;
		int c=0x01;
		
		for(a=0,b=10;a<=b;++a,b-=2)
		{
		C<<=1;
		}

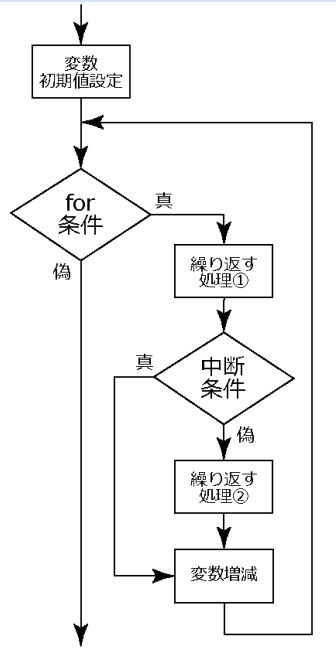
繰り返す処理の途中に if 文と break; を入れると、その if 条件を満たした場合、繰り返すループから抜けることができます。

使い方	説明	使用例	フローチャート

for(変数の初期値;条件;増減処理)
{
繰り返す処理①;
 	
if(中断条件) break;
 	
繰り返す処理②;
}
	for 文の条件が真の場合は『繰り返す処理①』を実行した後、『中断条件』判断を行い、真なら繰り返し処理を即座に中断します。
	『中断条件』判断が偽なら、繰り返す処理②を実行し、増減処理を行った後、for 文の条件判断を行います。
		int a;
		int b=c=10;
		
		for(a=0;a<=10;a++)
		{
		--b;
		if(a==5) break;
		C-=2;
		}
			

繰り返す処理の途中に if 文と continue; を入れると、その if 条件を満たした場合、continue 以降の処理を中断し、変数増減処理に飛ぶことができます。

使い方	説明	使用例	フローチャート

for(変数の初期値;条件;増減処理)
{
繰り返す処理①;
 	
if(中断条件)
continue;
 	
繰り返す処理②;
}
	for 文の条件が真の場合は『繰り返す処理①』を実行した後、『中断条件』判断を行い、真ならそれ以降の『繰り返す②』は実行せずに変数の増減処理に移行し、 for 文の条件判断を続行します。
	『中断条件』判断が偽なら、繰り返す処理②を実行し、増減処理を行った後、for 文の条件判断を行います。
		int a;
		int b=c=10;
		
		for(a=0;a<=10;++a)
		{
		--b;
		if(a==5) continue;
		C-=2;
		}
			

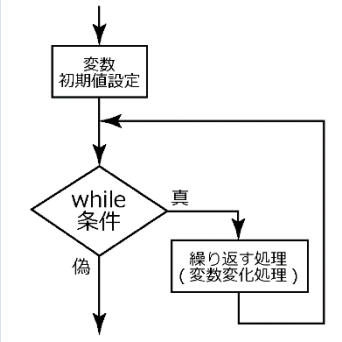
while 制御文

while 制御文は、条件が真である限りならば処理を繰り返し実行します。

前述の for 文で条件が成立する限り繰り返す書き方がありましたが動作はそれと同じです。

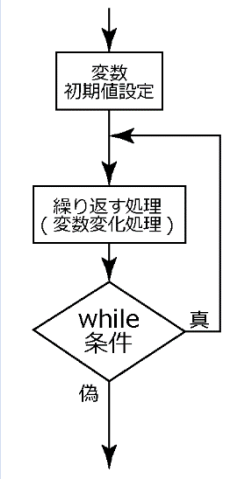
for 文ではその後の( )内に、条件の他に変数の初期設定と変化処理を書きましたが、while 文では while 文の前に変数の初期設定を、{ }でくくられた繰り返す処理の中に変数の変化処理を書きます。

使い方	説明	使用例	フローチャート

変数の初期設定;
 	
while(条件)
{
繰り返す処理(変数変化処理を含む);
}
	まず変数の初期設定。
	
	条件が真の場合は『繰り返す処理』を実行。偽なら繰り返し処理から抜ける。
	
	条件判断に使う変数の値の変化は『繰り返す処理』の中に含める。
		int a=0;
		int b=10;
		
		while(a<=10)
		{
		b-=2;
		++a;
		}
			

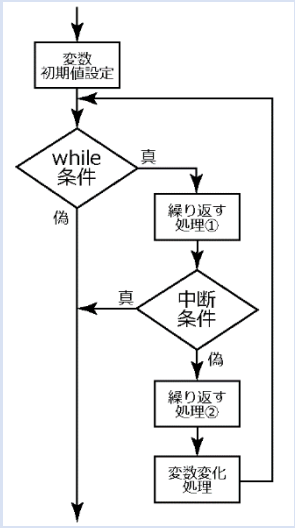
while 文では条件判断をした後に繰り返す処理を実行しますが、do-while 文を使用すると、最初に処理を実行した後、条件判断を行う事ができます。

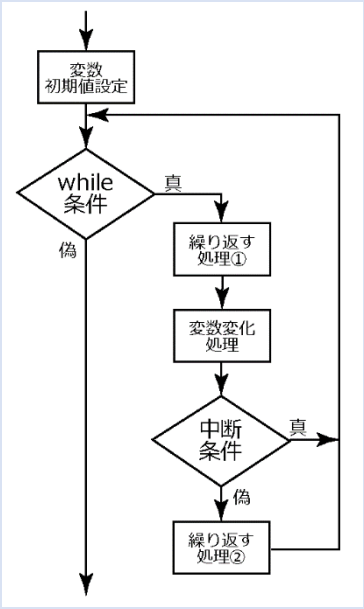
使い方	説明	使用例	フローチャート

変数の初期設定;
 	
do
{
繰り返す処理(変数変化処理を含む);
}
while(条件);
	まず変数の初期設定。
	
	最初に『繰り返す処理』を実行。
	
	その後に条件判断を行い、真なら『繰り返す処理』にもどり、偽ならこの繰り返しから抜ける。
		int a=0;
		int b=10;
		
		do
		{
		b-=2;
		++a;
		}
		while(a<=10);
			

while 文でも for 文の時と同じように break;や continue;を使用することができます。

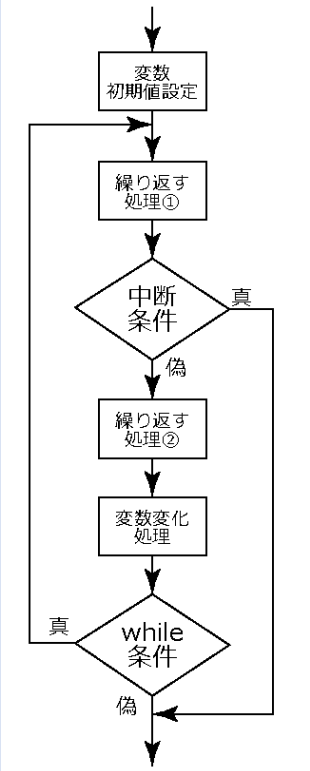
使い方	説明	使用例	フローチャート

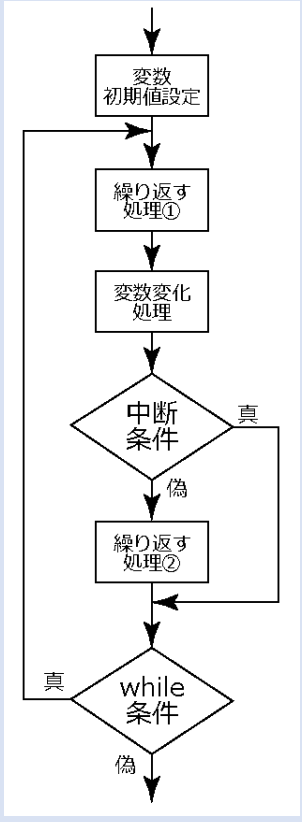
変数の初期設定;
 	
while(条件)
{
繰り返す処理①;
 	
if(中断条件) break;
 	
繰り返す処理②;
変数変化処理;
}
	while 文の条件が真の場合は『繰り返す処理①』を実行した後、『中断条件』判断を行い、真なら繰り返し処理を即座に中断します。
	『中断条件』判断が偽なら、繰り返す処理②を実行し、変数変化処理を行った後、while 文の条件判断を行います。
		int a=10;
		int b=c=0;
		
		while(a>=10)
		{
		++b;
		if(a==0) break;
		C+=2;
		a--;
		}
			

変数の初期設定;
 	
while(条件)
{
繰り返す処理①;
変数変化処理;
 	
if(中断条件)
continue;
 	
繰り返す処理②;
}
	while 文の条件が真の場合は『繰り返す処理①』を実行した後、『中断条件』判断を行い、真ならそれ以降の『繰り返す②』は実行せずに変数の変数変化処理に移行し、 while 文の条件判断を続行します。
	『中断条件』判断が偽なら、繰り返す処理②を実行し、変数変化処理を行った後、while 文の条件判断を行います。
		int a=10;
		int b=c=0;
		
		while(a>=10)
		{
		++b;
		a--;
		if(a==0) continue;
		C+=2;
		}
			

do-while 文でも break;や continue;を使用することができます。

使い方	説明	使用例	フローチャート

変数の初期設定;
 	
do
{
繰り返す処理①;
 	
if(中断条件) break;
 	
繰り返す処理②;
変数変化処理;
}
while(条件);
	まず変数の初期設定。
	
	最初に『繰り返す処理①』を実行した後、『中断条件』判断を行い、真なら繰り返し処理を即座に中断します。
	『中断条件』判断が偽なら、繰り返す処理②を実行し、変数変化処理を行った後、while 文の条件判断を行います。
	while 文の条件が真なら『繰り返す処理①』にもどり、偽ならこのループから抜けます。
		int a=10;
		int b=c=0;
		
		do
		{
		++b;
		if(a==0) break;
		C+=2;
		a--;
		}
		while(a>=10);
			

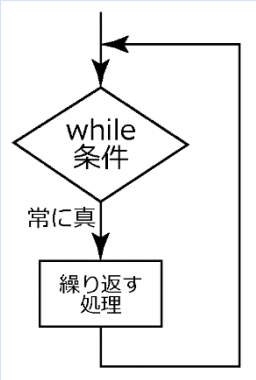
変数の初期設定;
 	
do
{
繰り返す処理①;
変数変化処理;
 	
if(中断条件)
continue;
 	
繰り返す処理②;
}
while(条件);
	まず変数の初期設定。
	
	最初に『繰り返す処理①』を実行した後、『中断条件』判断を行い、真なら『繰り返し処理②』を行わずに while 文の条件判断を行います。
	
	『中断条件』判断が偽なら、『繰り返す処理②』を実行し、変数変化処理を行った後、while 文の条件判断を行います。
	while 文の条件が真なら『繰り返す処理①』にもどり、偽ならこのループから抜けます。
		int a=10;
		int b=c=0;
		
		do
		{
		++b;
		a--;
		if(a==0) continue;
		C+=2;
		}
		while(a>=10);
			

間違えやすいポイント

- do-while 文で、do の後に(　)はありません。
- do-while 文で、while(条件)の後には『;』が必要です。
- continue を使った文では、変数変化処理を記述する場所によって『無限ループ』に陥ってしまいます。例えば、continue;の後に変数変化処理を書くと、中断条件により変数変化処理を行わずに while(条件)の判断になるため、条件変数が変化せず、いつまでも『繰り返す処理』を実行し続けるになります。

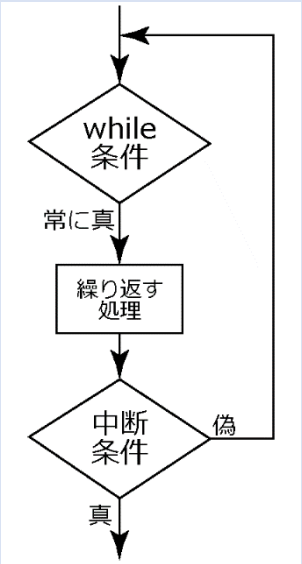
while 文を使用し、その処理をいつまでも繰り返す『無限ループ』を作ることもできます。

使い方	説明	使用例	フローチャート

while(1)
{
繰り返す処理;
}
	条件が『1』なので常に真となるため、『繰り返す処理』をいつまでも繰り返す。	int a=0;
		
		while(1)
		{
		++a;
		}
			

無限ループの繰り返し処理の中に break;文を書くことで、無限ループから抜ける事もできます。

使い方	説明	使用例	フローチャート

while(1)
{
繰り返す処理;
if(中断条件) break;
}
	条件が『1』なので常に真となるため、『繰り返す処理』実行。
	『中断条件』判断が偽なら、『繰り返す処理』を繰り返し、真なら無限ループから抜けます。
		int a=0;
		
		while(1)
		{
		++a;
		if(a==10) break;
		}
			

for 文を使用して無限ループを作ることもできます。もちろん break;も使用することができます。

使い方	説明	使用例

for(;;)
{
繰り返す処理;
}
	for 文の変数初期値や条件文、変数の増減処理がありませんが、【条件が『0』以外なので常に真】となるため、『繰り返す処理』をいつまでも繰り返す。	int a=0;
		
		for(;;)
		{
		++a;
		}

for(;;)
{
繰り返す処理;
if(中断条件) break;
}
	条件が『0』以外なので常に真となるため、『繰り返す処理』実行。
	『中断条件』判断が偽なら、『繰り返す処理』を繰り返し、真なら無限ループから抜けます。
		int a=0;
		
		for(;;)
		{
		++a;
		if(a==5) break;
		}

関 数

関数とは、ある処理に対しその手順を記したものです。

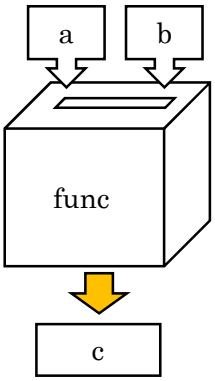
関数の書き方

関数は

```
戻り値の型　関数名(引数の型 仮引数名)
{
    処理記述;
    return 戻り値;
}
```

このように書きます。
戻り値や引数、仮引数など難しそうな名前が並んでいます。

もっとわかりやすく、関数を図にしてみました。



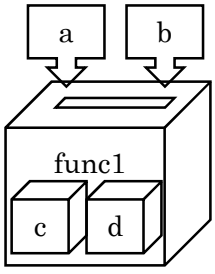
左のように、上から数字を入れると下から計算結果が出てくる箱があります。
この箱は『func』という名前がついています。
計算箱は中で『a+b という計算をしなさい！』という指令書が入っていました。
なので a+b の計算がされ、その答えが『c』とします。a と b は整数だとすると、答えの c も整数です。整数をあつかうデータの型には char や int などがありましたね。
ここでは int としましょう。

これを先ほどの関数の書き方に当てはめると

```
int func(int a, int b)
{
   int c;
   c=a+b;
   return c;
}
```

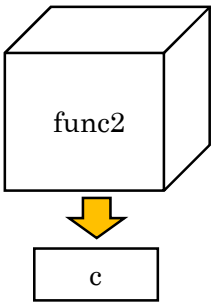
となります。戻り値はその関数の計算結果、仮引数とはその関数で使われる値の入れ物と考えるとわかりやすいでしょう。また、関数で計算をするときには仮引数に値を入れなくてはいけませんが、仮引数に入れる値のことを『実引数』といいます。
数学で　『$f(x)=x+y \quad x=5,y=2$』 のように書きますね。この時の『$f(x)$』が関数。数式中の『x,y』が仮引数。その値を決める『$x=5,y=2$』が実引数とイメージするとわかりやすいかもしれません。

関数の終わりに return を書くことで、計算結果を関数に『答えはこれですよ！』と伝えることができます。



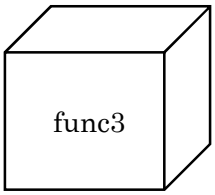
関数によっては『答え』を伝えない場合があります。
例えば、左の関数『func1』の中には『a を c に、b を d に入れなさい！』という指令書が入っていたとすると、上から a、b を入れて処理をした後、その値を関数に伝えなくて良い場合には return を書く必要はありません。
関数に戻ってくる値がない場合、『戻り値の型』には『void』と書きます。

```
int c,d;

void func1(int a, int b)
{
   c=a;
   d=b;
}
```



また、仮引数がない関数もあります。
例えば、左の関数『func2』の中には『1+6 をして c に入れなさい！』という指令書が入っていたとすると、上から何も入れなくても値『7』が入った『c』が出てくることになります。この c を関数に伝る場合には return を書いて関数に返します。
仮引数の値がないので(　)の中には『void』と書きます。

```
int func2(void)
{
   int c;
   c=1+6;
   return c;
}
```



戻り値も仮引数も無い関数もあります。
例えば、左の関数『func3』の中には『何もするな！』という指令書が入っていたとすると、上から何も入入ってこなくても『何もしないだけ』です。答えも出てきませんので関数に返す値もありません。
戻り値も仮引数の値もないので、戻り値の型にも(　)の中にも『void』と書きます。

```
void func3(void)
{
   NOP()
}
```

NOP()は何もしないという命令の時に使う関数です。
関数表記の『void』は省略することができます。

メイン関数

C 言語でのプログラムは 1 つ以上の『関数』の組み合わせで構成されますが、その中に『メイン関数』という関数があります。C 言語のプログラムでは必ず 1 つだけ『メイン関数』を書く決まりがあります。
プログラムをスタートさせた時、必ずこの『メイン関数』が最初に実行されます。

```
void main(void)
{
 .
}
```

メイン関数は左のように『void main(void)』で始まる関数でこの中にその他の関数などを入れて、プログラムを実行する関数の『かたまり』にします。

```
main()
{
 .
}
```

関数表記の『void』は省略することができるので左のように書くこともできます。

関数のプロトタイプ宣言

メイン関数に他の関数を書き加えてプログラムを書きますが、その他のプログラムをそのまま中に書くと、とても長くなったり、とてもわかりにくいプログラムになってしまいます。

```
void main(void)
{
int func0 (int a,int b)
  {
   return(a+b);
  }

void func1 (char c,char d)
 {
   d<<1;
   c>>1;
  }

int func0 (int a,int b)
  {
   return(a+b);
  }
}
```

左の例ではメイン関数の中に『func0』『func1』という関数が入っています。この例ではそれぞれの関数はそれほど長くないのですが、実際はもっと長くなることもあります。
また、　『func0』は 2 回書かれていますので、まとめてスッキリさせたいですね。

```
void main(void)
{
func0 (3,6);
func1 (0x11,0x80);
func0 (2,4);
}

int func0 (int a,int b)
  {
   return(a+b);
  }

void func1 (char c,char d)
 {
   d<<1;
   e>>1;
  }
```

メイン関数はこの部分だけ

その他の関数はここに書かれている。
メイン関数はそれに対応する関数を呼び出して使用する。

上のように書くと少しスッキリします。複数回使用する関数も 1 つだけ書けば良いので見やすくなります。

しかし！

この書き方はちょっと問題があります。それは、func0、func1 の関数の記述がメイン関数より下にあるので、メイン関数を実行した時、それぞれの関数がまだ読み込まれておらず、その時点では func0、func1 が何なのかわからないためエラーになってしまうのです。
メイン関数の上に func0、func1 の記述をすればよいのでしょうが、関数がとても多くなった場合にはとても大変なことになります。
そこで、メイン関数の前に『このプログラムでは次の関数を使いますよ！』と宣言をする記述をします。

それを『関数のプロトタイプ宣言』といいます。

```
int func0 (int a,int b);
void func1 (char c,char
d);

void main(void)
{
func0 (3,6);
func1 (0x11,0x80);
func0 (2,4);
}

int func0 (int a,int b)
  {
   return(a+b);
  }

void func1 (char c,char d)
 {
   d<<1;
   e>>1;
```

関数のプロトタイプ宣言

このプログラムで使う関数を、メイン関数の前に宣言します。

関数のプロトタイプ宣言は

戻り値の型　関数名(引数の型 仮引数名);

の形で書くことになっています。

グローバル変数とローカル変数

関数などで一時的に値を入れておく場所を変数と呼ぶことは最初にお話しました。

では、次のようなプログラムの場合、変数はどれになるでしょう？

```
void main(void)
{

int a,b,c;
a=b+c;

}
```

```
int a,b,c;

void main(void)
{

a=b+c;

}
```

どちらの書き方も『a、b、c は int 型の値を取る変数だ』と宣言していますので、a、b、c は変数ですね。

しかし、この 2 つは同じ変数の宣言をしているのですが、意味が異なります。

上の左側の場合のように、関数の中で変数を宣言をすると、その変数はその関数の中のみで使える変数になります。a、b、c はその関数のみで使える変数になり、このような変数を『ローカル変数』といいます。

上の右側の場合は、関数の外で変数の宣言をしています。この場合の a、b、c はメイン関数以外でも使用することができる変数になり、このような変数を『グローバル変数』といいます。

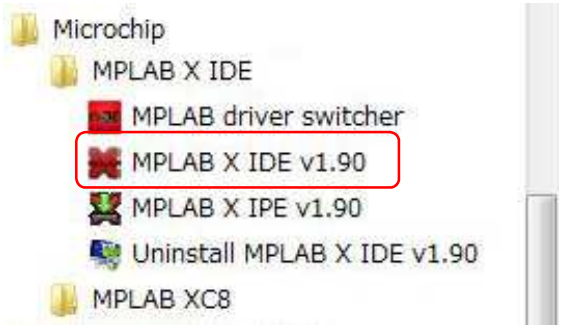
4. MPLAB X でプログラミング

開発環境が整い、LED が点灯、消灯するときのポートの条件がわかりました。また、C 言語でのプログラム方法もなんとなくわかったところで実際にプログラムを作成してみましょう。

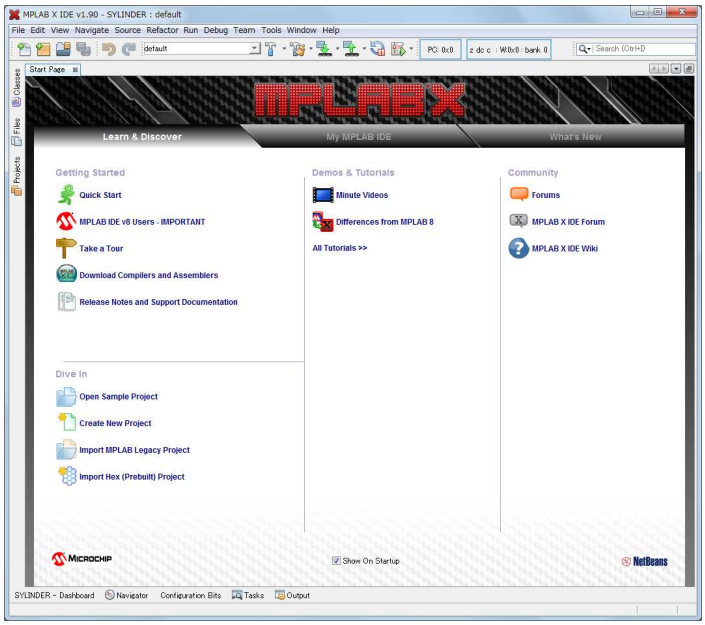
プロジェクトとソースファイルの作成

まず、MPLAB X を起動します。

デスクトップに作成されたアイコンをクリックするか、『スタートメニュー』→『すべてのプログラム』→『Microchip』→『MPLAB X IDE』→『MPLAB X IDE v x.xx』から起動します。(v x.xx はバージョンを表します。)



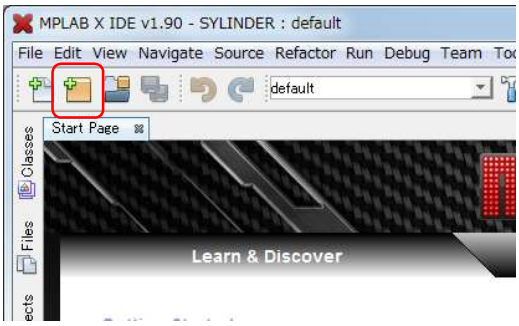
起動画面がしばらく表示された後、下の画面が表示されます。

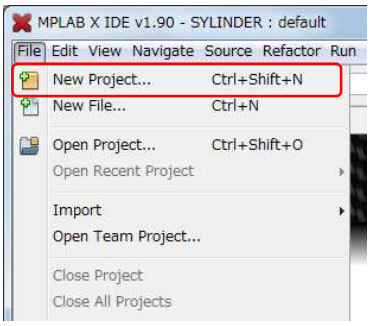


MPLAB X でのプログラミングは、そのプログラムに必要な色々なデータをひとまとめにした『プロジェクト』という単位で作成していきます。

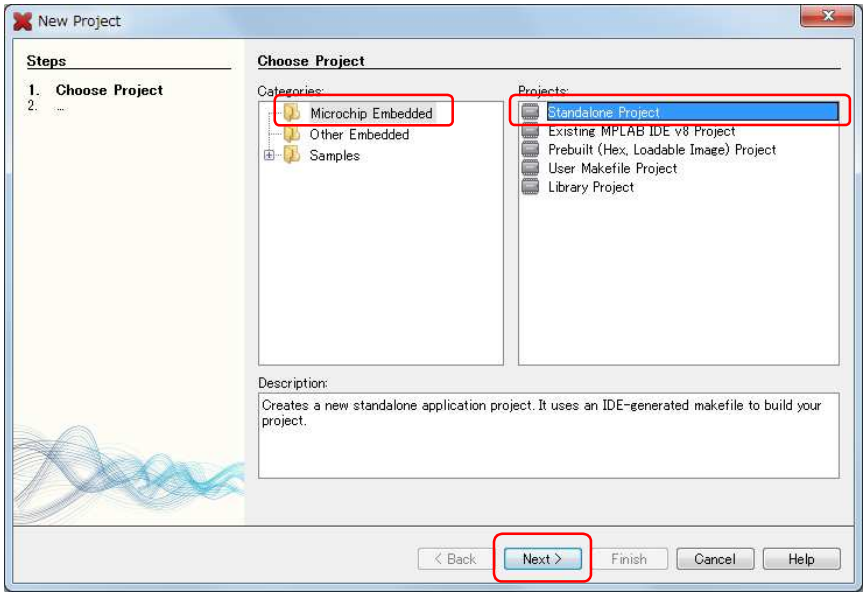
まず、そのプロジェクトを作成します。

ウインドウ左上の『New Project...』をクリック、または『File』→『New Project...』を選択します。

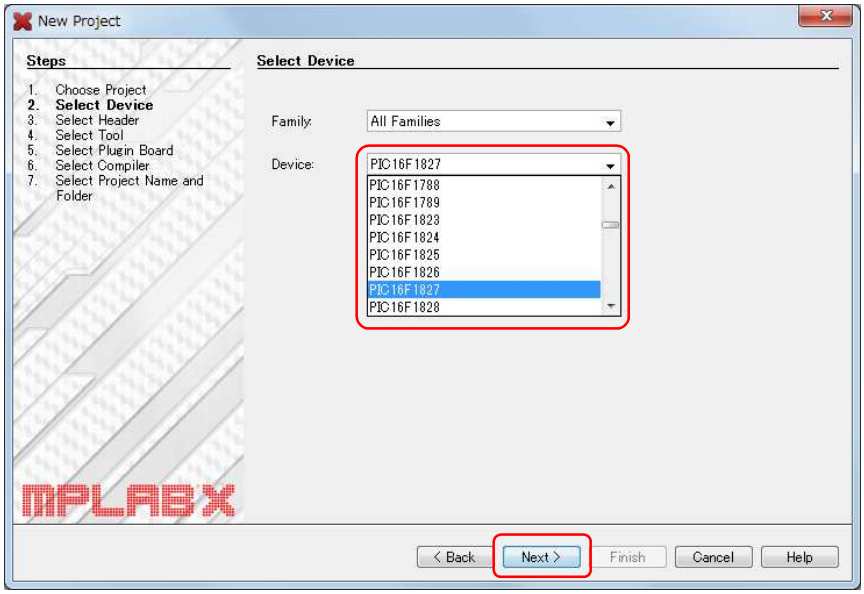




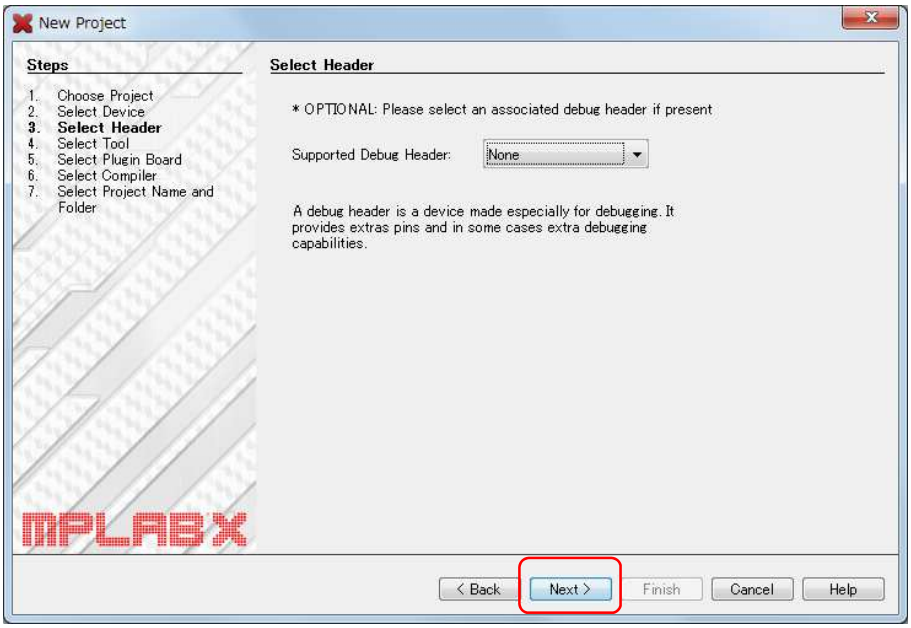
すると、プロジェクトを作成する最初のステップが表示されます。
最初の画面では、真ん中の『Categories:』で『Microchip Embedded』をクリックして選択、右側の『Projects:』で『Standalone Ploject』を選択して『Next >』をクリックします。



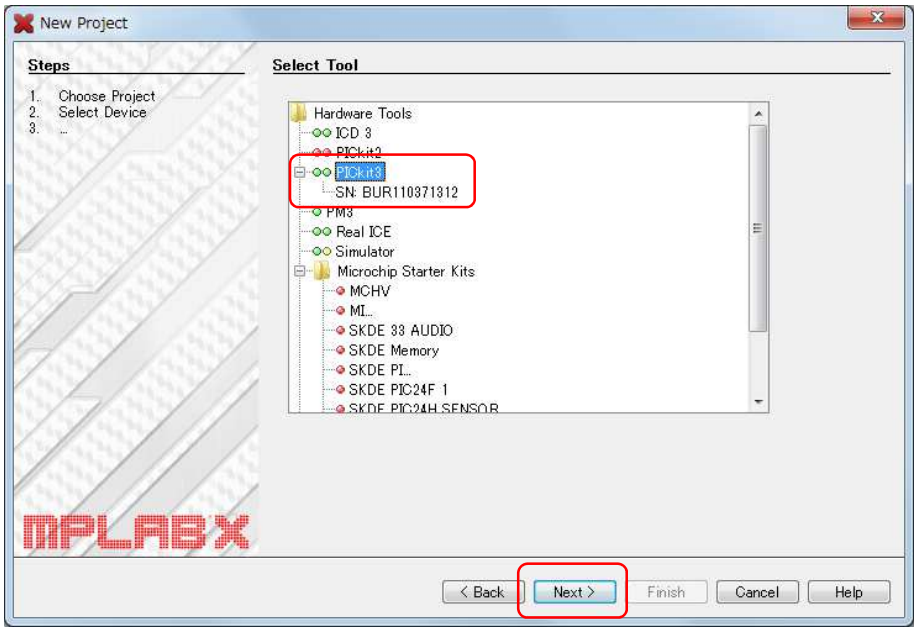
使用するマイコンを選択する画面が表示されますので、『Device:』の一覧から PICA Tower で使用している『PIC16F1827』を選択し『Next >』をクリックします。



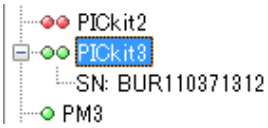
次の画面では特に何もしないで『None』のまま『Next >』をクリックします。



次の画面では Select Tool で『PICkit3』を選択します。PICkit3 を接続している場合、その PICkit3 のシリアルナンバーが表示されます。選択したら『Next >』をクリックします。

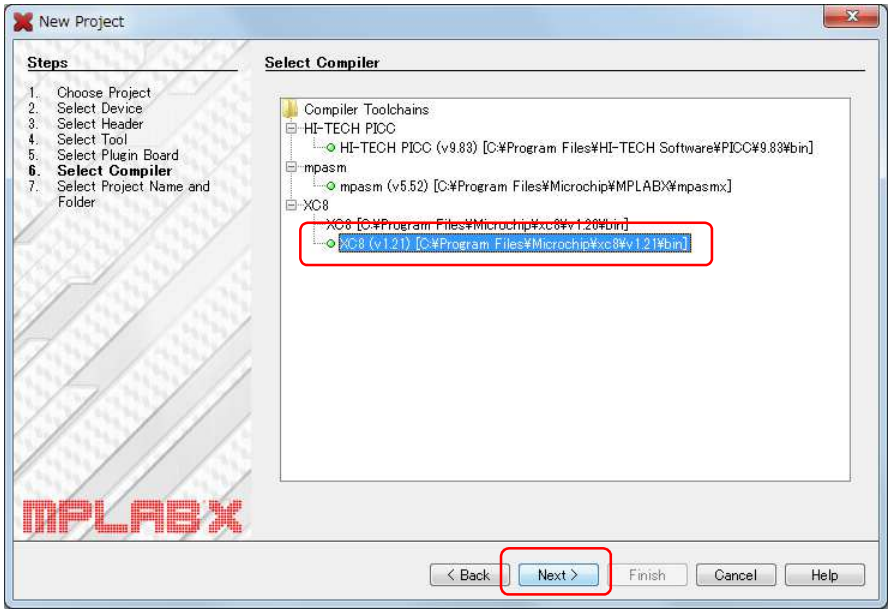


緑や赤い丸は何？

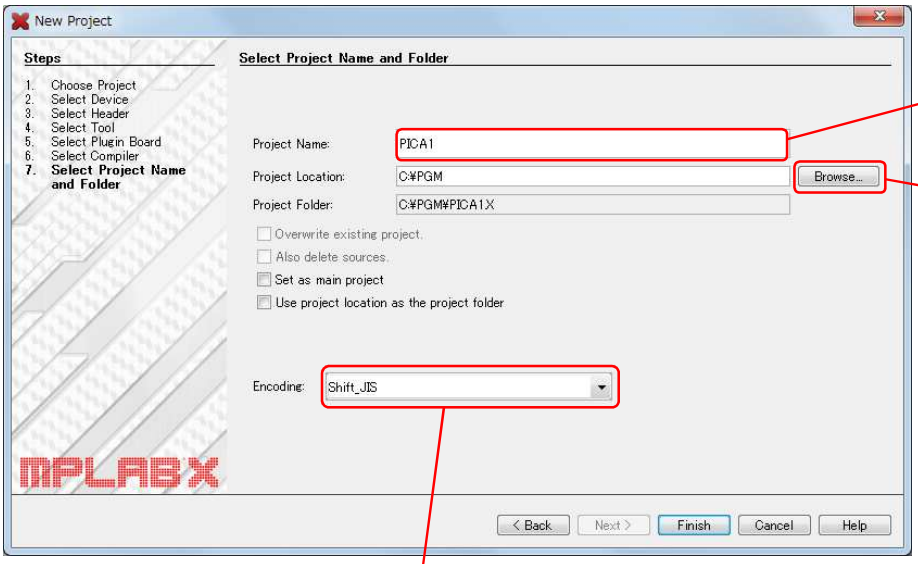


Select Tool には、PIC マイコンのプログラムの書き込みやテストをするときに使用するツールの一覧が表示されます。
この一覧表示の左に付いている ●● や ●● はそのツールが使用するマイコンに対応しているかどうかを表しています。
赤いマークは対応していないことを。緑のマークは対応されていることを表します。
たまに黄色いマークが表示される場合があります。その場合、機能としては搭載しているが検証が終わっていない「ベータ版対応」であることを表します。
詳しくは、5 ページでダウンロードした『MPLAB X IDE ユーザーズガイド』の 41 ページをご覧ください。

次の画面では、使用するコンパイラを選択します。今回は『XC8 C コンパイラ』を使用しますので、『XC8 (v x.xx)を選択し、『Next >』をクリックします。

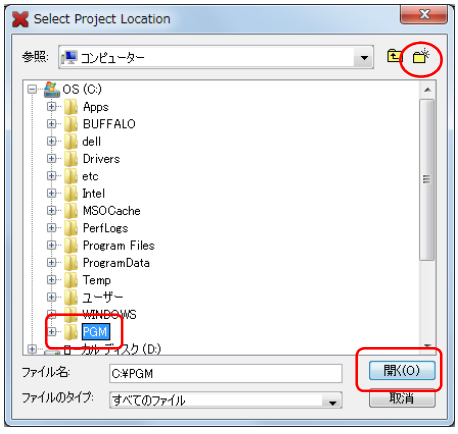


次にプロジェクト名と保存する場所を設定します。
今回はプロジェクト名を『PICA1』にしましょう。
保存する場所は C ドライブの直下に『PGM』フォルダを作成し、その中にプログラムを保存することにします。
ファイル名や保存する場所のフォルダ名には日本語などの『2 バイト文字』を使用しないようにしましょう。
2 バイト文字を使用すると MPLAB X が、ファイルの場所などを認識できない場合があります。



『PICA1』と入力

『Browse』をクリックすると下記画面が表示されるので、右上の をクリックし、作成されたフォルダに『PGM』と名前をつけ、その『PGM』フォルダを選択した状態で『開く』をクリックすると、『Project Location:』にフォルダのパスが表示されます。



プログラム中の日本語が文字化けをしないように『Shift_JIS』を選択しておきます。

入力が完了したら『Finish』をクリックします

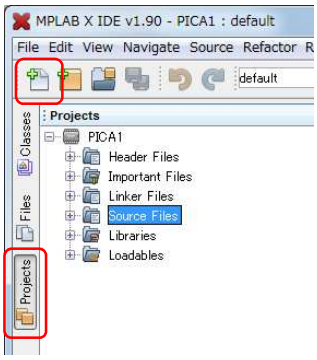
これでプロジェクトの設定は完了です。

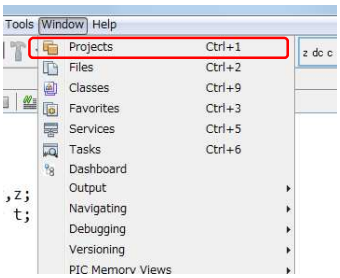
プロジェクトの設定が完了すると下記画面が表示され、設定したプロジェクトの『PICA1』が表示されます。

表示されない場合は、左側(または下)にある『Projects』をクリックして、表示させます。

このプロジェクトにプログラムを書き込む『ソースファイル』を作成します。

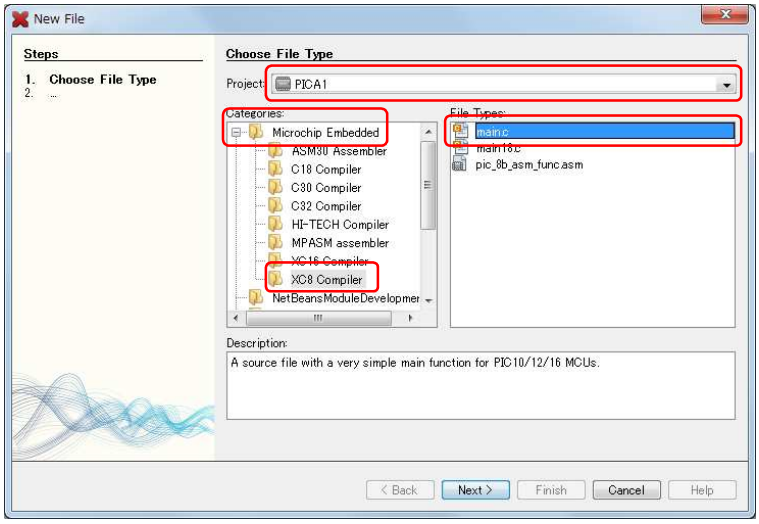
　表示されたプロジェクトの中の『Source Files』を選択した状態で、左上の『New File』アイコンをクリックします。



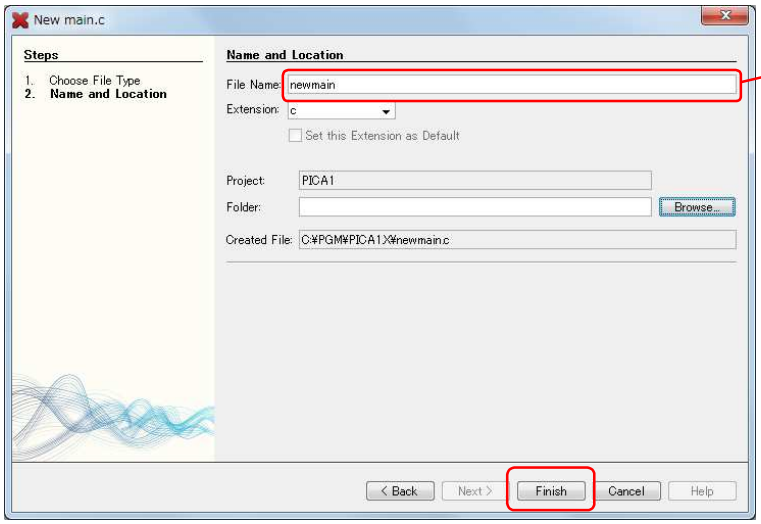


左側(または下)に『Projects』がない場合、『Projects』はメニューバーの『Window』→『Projects』で表示させることもできます。

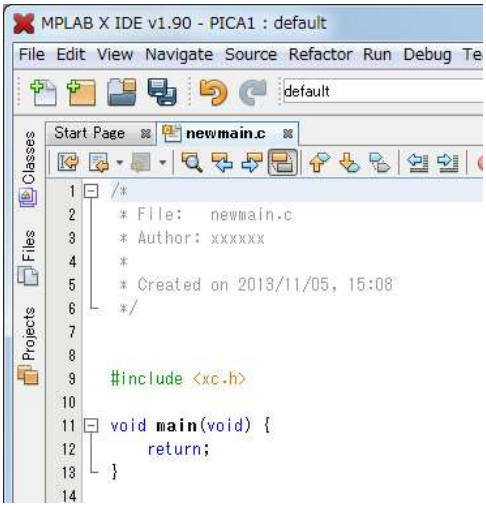
『Project:』に『PICA1』が表示されていることを確認し、『Categories:』の中かから『Microchip Embedded』→『XC8 Compiler』を選択し、右側の『File Types:』は『main.c』を選択し『Next >』をクリックします。



確認画面が表示されますので、『Finish』をクリックします。

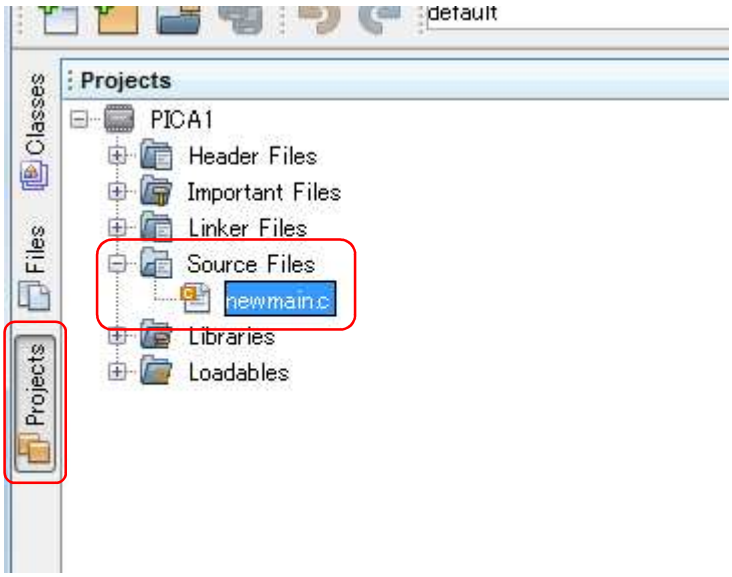


ここにわかりやすいファイルネームを
入力しても OK。
今回は『newmain』のままにしまし



左記の画面が表示されれば、プログラムの入力準備は完了です。

左側の『Projects』をクリックすると、『Source Files』の中に『newmain.c』が作成されていることがわかります。



もう一度、左側の『Projects』をクリックして newmain.c のプログラム入力画面を表示させます。

C 言語のプログラミング

ここからは実際のプログラムを例に解説を行っていきます。

まず、下記のプログラムを作成します。最初の行から順番に説明します。

```
/*
 * File:   newmain.c
 * Author: あなたの名前
 *
 * Created on ソースファイルを作成した日時(自動で入力されているはずです。)
 */


#include <xc.h>

// CONFIG1
#pragma config FOSC = INTOSC
#pragma config WDTE = OFF
#pragma config PWRTE = OFF
#pragma config MCLRE = OFF
#pragma config CP = OFF
#pragma config CPD = OFF
#pragma config BOREN = ON
#pragma config CLKOUTEN = OFF
#pragma config IESO = OFF
#pragma config FCMEN = OFF

// CONFIG2
#pragma config WRT = OFF
#pragma config PLLEN = OFF
#pragma config STVREN = OFF
#pragma config BORV = LO
#pragma config LVP = OFF

void main(void) {

    OSCCON = 0b00111000;            //システムクロックはデフォルトの 500kHz に
    OSCTUNE = 0;                    //クロックチューニングはなし
    PORTA = 0;                      //ポート A を全て『L』に
    LATA = 0;                       //ポート A のデータラッチレジスタをポート A と同じ値に
    ANSELA = 0;                     //ポート A をデジタル I/O に
    TRISA = 0;                      //ポート A は RA5 以外を出力に(RA5 は入力専用)
    nWPUEN = 0;                     //OPTION_REG の WPUEN を『L』にし、ウィークプルアップを有効に
    WPUA = 1;                       //ポート A にプルアップ設定(RA5 のみ)
    PORTB = 0;                      //ポート B を全て『L』に
    LATB = 0;                       //ポート B のデータラッチレジスタをポート A と同じ値に
    ANSELB = 0;                     //ポート B をデジタル I/O に
    TRISB = 0;                      //ポート B は全て出力に

    while(1){

        PORTB = 0xFF;               //ポート B を全て『H』に
        PORTA = 0x00;               //ポート A を全て『L』に

        }
}
```

※ここまでは自動で入力されているはずです。

・全体の構成

C 言語で作成するプログラムの全体構成は、主にこのようになっています。

```
宣言部

メイン関数

その他の関数
```

『宣言部』ではヘッダーファイル、コンフィグレーション、グローバル変数、関数のプロトタイプ宣言などが記載されます。
『メイン関数』はこのプログラムが実行する部分で、ここに処理の記述を行います。
『その他の関数』はメイン関数内で呼び出される関数を記述します。

・コメント

最初の/*　*/で囲まれた部分はコメントですね。
ここは newmain.c ファイルにあらかじめ記入されています。必要に応じて内容の変更や追加をして、このプログラムの内容がわかるようにしておくとよいでしょう。

・ヘッダーファイル

```
#include <xc.h>
```

この部分は『xc.h というファイルをこのプログラムに含めます！』と宣言する部分です。
#include <ファイル名>と書き、行末に『;』は付けません。
ファイル名のところの書き方が 2 通りあり、<ファイル名>と書く方法と”ファイル名”と書く方法があります。

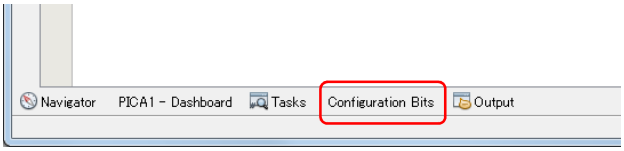
<ファイル名>と書いた場合は、XC8 がインストールされたフォルダ内でそのファイルを探しに行きます。
”ファイル名” と書いた場合は、プログラムのソースファイルがあるフォルダ内を探しに行きます。

『xc.h』はヘッダーファイルと言い、マイコンの機能を設定する『レジスタ』の名前の定義などがまとめて記述してあるファイルです。XC8 コンパイラでプログラムを作成する際、この記述をすることで、XC8 コンパイラでプログラムできるどのマイコンでも、わかりやすいレジスタの名前をプログラムに使用できるようになります。

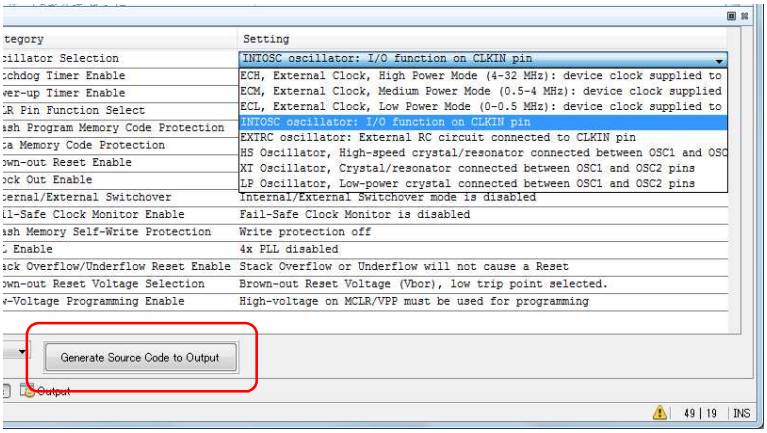
・コンフィグレーション・ビット

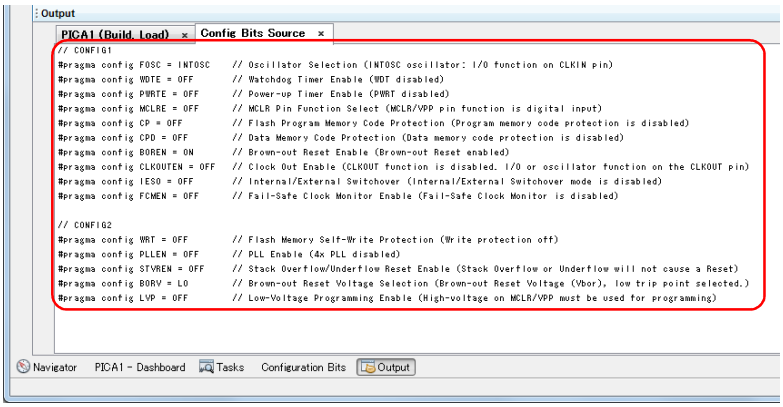
```
// CONFIG1
#pragma config FOSC = INTOSC
#pragma config WDTE = OFF
#pragma config PWRTE = OFF
#pragma config MCLRE = OFF
#pragma config CP = OFF
#pragma config CPD = OFF
#pragma config BOREN = ON
#pragma config CLKOUTEN = OFF
#pragma config IESO = OFF
#pragma config FCMEN = OFF

// CONFIG2
#pragma config WRT = OFF
#pragma config PLLEN = OFF
#pragma config STVREN = OFF
#pragma config BORV = LO
#pragma config LVP = OFF
```

この部分は、使用するマイコンの基本的な機能を設定するための記述をする部分です。
マイコンの基本的な機能は使用するマイコンにより多少異なりますので、あらかじめマイコンのデータシートでどのような機能があるのかを確認しておく必要があります。
本機で使用するマイコン『PIC16F1827』のデータシートでは、43 ページから記載されています。
データシートを見るとたくさんの機能があり、プログラムで記述することがとても大変に感じます。
しかし、MPLAB X IDE にはコンフィグレーション・ビットを記述するための便利な機能が備わっています。
MPLAB X IDE の画面下方にある『Configuration Bits』をクリックします。



下部に表示されていない場合は、メニューバーの『Window』→『PIC Memory Views』→『Configuration Bits』を選択しても表示することができます。





すると、設定する項目の一覧が表示されます。
右端の『Setting』の項目は選択メニューになっていますので、その中から設定したい内容を選択します。
すると『Option』の項目が設定する値に変わります。『Option』の方を選択して設定しても OK です。
設定が完了したら下方にある『Generate Source Code to Output』をクリックすると、『Config Bits Source』ウインドウが開き、設定された内容がプログラムとして記載された状態で表示されます。
このプログラムをコピーしてソースファイルに貼り付ければ、コンフィグレーション・ビットのプログラムができあがります。

『PIC16F1827』で設定できるコンフィグレーション・ビットは下記の通りで、各ビットの説明中に赤字で記載している値が今回のプログラムで設定した値です。

ビット名	機能内容	設定	設定値の内容
FOSC	マイコンを動作させるために必要な発振回路の構成を設定。 詳しくはデータシートの 53 ページ～を参照。	ECH	外部発振回路から CLKIN に 4～20MHz のクロックを入力する時に設定。
		ECM	外部発振回路から CLKIN に 0.5～4MHz のクロックを入力する時に設定。
		ECL	外部発振回路から CLKIN に 0～0.5MHz のクロックを入力する時に設定。
		INTOSC	マイコン内部の発振回路を使用する時に設定。CLKIN 端子は I/O として利用可。
		EXTRC	CLKIN に抵抗、コンデンサを接続し、CR 発振回路を構成する時に設定。
		HS	周波数が 4～20MHz 以上の発振子を OSC1、OSC2 間に接続して構成する時に設定。
		XT	周波数が 4MHz までの発振子を OSC1、OSC2 間に接続して構成する時に設定。
		LP	周波数が 32.768kHz の時計用発振子等を OSC1、OSC2 間に接続して構成する時に設定。

WDTE	マイコンが暴走した時にリセットする『ウォッチドッグタイマー』の設定。	ON	ウォッチドッグタイマーを有効にする。
		NSLEEP	マイコンが稼働中はウォッチドッグタイマーを有効に、スリープ中は無効にする。
		SWDTEN	ウォッチドッグタイマーの ON・OFF を『WDTCON』レジスタの『SWDTEN』で制御する。
		OFF	ウォッチドッグタイマーを無効にする。
PWRTE	マイコンの電源 ON 後に、一定時間リセットをする『パワーアップタイマー』の設定。	OFF	パワーアップタイマーを無効にする時に設定。
		ON	パワーアップタイマーを有効にする時に設定。
MCLRE	MCLR 端子に外部リセット回路を接続するどうかを設定。	ON	MCLR 端子に外部リセット回路を接続し、リセット入力端子として使用する時に設定。
		OFF	MCLR 端子に外部リセット回路は接続せず、デジタル入力端子として使用する時に設定。
CP	マイコンのプログラムを書き込むメモリー領域を外部から読み出せないようにするプロテクトを設定する。	OFF	プロテクトを無効にする。
		ON	プロテクトを有効にする。
CDP	マイコンのデータなどを書き込む EEPROM 領域を外部から読み出せないようにするプロテクトを設定する。	OFF	プロテクトを無効にする。
		ON	プロテクトを有効にする。
BOREN	マイコンの電源電圧が設定値より低くなった時にリセットする『ブラウンアウト・リセット』を設定する。	ON	ブラウンアウト・リセットを有効にする。
		NSLEEP	マイコンが稼働中はウブラウンアウト・リセットを有効に、スリープ中は無効にする。
		SBODEN	ブラウンアウト・リセットの ON・OFF を『BORCON』レジスタの『SBOREN』で制御する。
		OFF	ブラウンアウト・リセットを無効にする。
CLKOUTEN	CLKOUT 端子の動作モードを設定する。	OFF	CLKOUT 端子からクロックを出力する機能を無効にし、入出力端子に設定する。FOSC を HS、XT、LP に設定した時もこの設定にしておく。
		ON	CLKOUT 端子からクロックを出力するように設定する。
IESO	外部からクロックを供給する場合、そのクロックが安定するまでの間、内部発振回路を使用する『内外クロック切り替え機能』の設定。	ON	内外クロック切り替え機能を有効にする。
		OFF	内外クロック切り替え機能を無効にする。
FCMEN	外部からクロックを供給する場合、外部クロックが停止してしまった時に内部発振回路に切り替える『フェールセーフクロックモニター』の設定。	ON	フェールセーフクロックモニター機能を有効にする。
		OFF	フェールセーフクロックモニター機能を無効にする。
WRT	プログラムを書き込む領域に、そのプログラム自体が『EECON』レジスタを使用してデータを書くことを許可するかどうかを設定。	OFF	書き込み保護を行わない。
		BOOT	000h～1FFh 番地まで保護し、200h～FFFh 番地を書き換え可能に設定する。
		HALF	000h～7FFh 番地まで保護し、800h～FFFh 番地を書き換え可能に設定する。
		ALL	000h～FFFh 番地まで、すべての領域を保護する。
PLLEN	クロックの周波数を 4 倍で使用する『4x PLL 機能』の設定。	ON	4x PLL 機能を使用する。
		OFF	4x PLL 機能を使用しない。
STVREN	データを一時的に退避するスタック領域がオーバーフローしたり、退避したデータが無いのに読み出そうとした時(アンダーフロー)にリセットするか否かを設定。	ON	オーバーフロー、アンダーフローが発生した時にリセットを行う。
		OFF	オーバーフロー、アンダーフローが発生してもリセットを行わない。

BORV	前ページの『BOREN』で OFF 以外に設定した時、ブラウンアウト・リセットが起動する電圧を設定。	LO	電源電圧が約 1.9V より低くなった時、ブラウンアウト・リセットが起動する。
		HI	電源電圧が約 2.5V より低くなった時、ブラウンアウト・リセットが起動する。
LVP	マイコンにプログラムを書き込むときに低電圧モードで書き込むか否かを設定。	ON	低電圧書き込みモードを ON に設定。
		OFF	低電圧書き込みモードを OFF に設定。

・メイン関数

C 言語のプログラムの書き方のところでも書いたように、マイコンがプログラムをスタートさせた時、最初に実行される関数が、この『メイン関数』です。

```
void main(void) {
・
・
・
・
}
```

コンフィグレーション・ビットの次の行の『void main(void)』で始まる部分が『メイン関数』で、『void』は省略することも可能でしたね。ですから『main( )』と書くこともできます。

void main(void)の後の{　}で囲まれた部分がメイン関数の本文で、この中に書かれた命令を順番に実行していくことになります。

```
void main(void) {

    OSCCON = 0b00111000;
    OSCTUNE = 0;
    PORTA = 0;
    LATA = 0;
    ANSELA = 0;
    TRISA = 0;
    nWPUEN = 0;
    WPUA = 1;
    PORTB = 0;
    LATB = 0;
    ANSELB = 0;
    TRISB = 0;

    while(1){

        PORTB = 0xFF;
        PORTA = 0x00;

        }
}
```

左の色をつけた記述の部分がメイン関数の本文で、その中の下の方にも{　}がありますが、これは『制御文』の項で説明した無限ループ『while(1)』の命令部分です。

・マイコンの初期設定

メイン関数の中に書かれている下記の行は、マイコンの機能や初期状態を設定する部分です。

```
OSCCON = 0b00111000;   //システムクロックはデフォルトの 500kHz に
OSCTUNE = 0;           //クロックチューニングはなし
PORTA = 0;             //ポート A を全て『L』に
LATA = 0;              //ポート A のデータラッチレジスタをポート A と同じ値に
ANSELA = 0;            //ポート A をデジタル I/O に
TRISA = 0;             //ポート A は RA5 以外を出力に(RA5 は入力専用)
nWPUEN = 0;            //OPTION_REG の WPUEN を『L』にし、ウィークプルアップを有効に
WPUA = 1;              //ポート A にプルアップ設定(RA5 のみ)
PORTB = 0;             //ポート B を全て『L』に
LATB = 0;              //ポート B のデータラッチレジスタをポート A と同じ値に
ANSELB = 0;            //ポート B をデジタル I/O に
TRISB = 0;             //ポート B は全て出力に
```

機能や初期状態の設定は、マイコンの『レジスタ』の値を書き換えることで決定します。

使用するマイコンによって『レジスタ』の種類や内容が異なり、PICA Tower で使用する PIC16F1827 の

『レジスタ』は、データシートの 27～35 ページに一覧が記載されています。
では最初の行から見てみましょう。

OSCCON レジスタ

データシート 65 ページに記載されている図に緑色で描いた線と文字を追加し少し加工してみました。

REGISTER 5-1: OSCCON: OSCILLATOR CONTROL REGISTER

R/W-0/0	R/W-0/0	R/W-1/1	R/W-1/1	R/W-1/1	U-0	R/W-0/0	R/W-0/0
SPLLEN		IRCF<3:0>			—	SCS<1:0>	
bit 7	bit6	bit5	bit4	bit3	bit2	bit1	bit 0

レジスタはデータが入っている『箱』の集まりで、OSCCON には bit0～bit7 まで 8 コの箱を持っています。
そしてそれぞれの箱には『1』か『0』が入ります。
その次に書かれている下記の表は、それぞれの箱の性質を説明しています。

Legend:		
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'
u = Bit is unchanged	x = Bit is unknown	-n/n = Value at POR and BOR/Value at all other Resets
'1' = Bit is set	'0' = Bit is cleared	

R=読み出し可能	W=書き込み可能	U=機能の割り当てがなく常に 0 と読み出される
u=変化しないビット	x=内容不明なビット	-n/n= POR、BOR 時の値/その他のリセット時の値
1=そのビットの値は 1	0=そのビットの値は 0	(※)POR:パワーオン・リセット BOR：ブラウンアウト・リセット

OSCCON レジスタの表の上部に『R/W-0/0』のように記載された項目があります。
これが箱の性質を表す項目で『R/W-0/0』の場合は『読み・書き可能で POR、BOR された時の値は 0、その他のリセットを実行された時の値も 0』という意味になります。
その下に記載されている下記の内容は、OSCCON レジスタの役割を設定するための説明で、OSCCON レジスタの表の中段と照らしあわせて設定します。

bit 7	**SPLLEN:** Software PLL Enable bit If PLLEN in Configuration Word 1 = 1: SPLLEN bit is ignored. 4x PLL is always enabled (subject to oscillator requirements) If PLLEN in Configuration Word 1 = 0: 1 = 4x PLL Is enabled 0 = 4x PLL is disabled
bit 6-3	**IRCF<3:0>:** Internal Oscillator Frequency Select bits 000x =31 kHz LF 0010 =31.25 kHz MF 0011 =31.25 kHz HF(1) 0100 =62.5 kHz MF 0101 =125 kHz MF 0110 =250 kHz MF 0111 =500 kHz MF (default upon Reset) 1000 =125 kHz HF(1) 1001 =250 kHz HF(1) 1010 =500 kHz HF(1) 1011 =1 MHz HF 1100 =2 MHz HF 1101 =4 MHz HF 1110 =8 MHz or 32 MHz HF(see Section 5.2.2.1 "HFINTOSC") 1111 =16 MHz HF
bit 2	**Unimplemented:** Read as '0'
bit 1-0	**SCS<1:0>:** System Clock Select bits 1x = Internal oscillator block 01 = Timer1 oscillator 00 = Clock determined by FOSC<2:0> in Configuration Word 1.
Note 1:	Duplicate frequency derived from HFINTOSC.

bit7 には『SPLLEN』の機能が、bit6～bit3 には『IRCF』の機能が、bit1～bit0 には『SCS』の機能が割り当てられていて、それぞれのビットに『1』または『0』を書き込むことで機能設定します。

『SPLLEN』はプログラムで 4xPLL 機能を使用するか否かを設定。今回は、4xPLL 機能は使用しないので『4xPLL is disabled』、つまり『0』に設定します。
『IRCF』はマイコン内部の発振回路の周波数を設定。今回はリセット時のデフォルトの 500kHz を使用しますので、『IRCF』

は『0111』に設定します。

『SCS』はマイコンのシステムクロックをどこから供給するかを設定。今回はコンフィグレーション・ビットの FOSC で設定した内容を適用(内部発振)しますので、『SCS』は『00』に設定します。

以上をまとめると、『OSCCON』レジスタは、

R/W-0/0	R/W-0/0	R/W-1/1	R/W-1/1	R/W-1/1	U-0	R/W-0/0	R/W-0/0
SPLLEN	IRCF<3:0>				—	SCS<1:0>	
bit 7							bit 0
0	0	1	1	1	0	0	0

と設定することになるため、プログラムで

OSCCON = 0b00111000;　と書きます。

OSCTUENE レジスタ

更にデータシートを読み進めると、『OSCTUNE』レジスタで、内部発振回路の発振周波数の微調整ができるようです。(データシート 57 ページ)

しかし、デフォルトの 500kHz で使用する場合、工場で調整されているようです。本機の場合、多少周波数がずれても問題ありませんが、どういうものか知っておくためレジスタの設定をしみましょう。

REGISTER 5-3:　OSCTUNE: OSCILLATOR TUNING REGISTER

U-0	U-0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0
—	—	TUN<5:0>					
bit 7							bit 0

bit 7-6	**Unimplemented:** Read as '0'
bit 5-0	**TUN<5:0>:** Frequency Tuning bits 011111 = Maximum frequency 011110 = • • • 000001 = 000000 = Oscillator module is running at the factory-calibrated frequency. 111111 = • • • 100000 = Minimum frequency

『OSCTUNE』レジスタは bit5～bit0 の 5 ビットで設定し、『000000』にすると工場で調整した周波数で動作するようです。

OSCTUNE = 0;

(16 進数なら 0x00、2 進数なら 0b00000000)と書いておきましょう。

・ポートの設定

PIC16F1827 には最大で 16 コの信号を入出力するポートがあります。そのポートを入力にするのか、出力にするのか、デジタル信号をあつかうのか、アナログ信号をあつかうのかなど、ポートの役割を設定しないといけません。

今回のプログラムでは LED の点灯を行うことが目的ですから、全てのポートで『デジタル信号をあつかう』ように設定します。また、いろいろな点灯パターンを作り出すためには、ポートの出力状態を変化させて点灯させる LED をコントロールしますので、入力専用ポート以外の全ポートを『出力』に設定します。

16 コ全てのポートをデジタル出力にする場合、『RA0』～『RA7』の『ポート A』が 8 コと、『RB0』～『RB7』の『ポート B』が 8 コあります。
データシートで入出力ポートの説明は 117 ページに記載してあります。
そこにはポートの設定をするためには『TRISx』『PORTx』『LATx』の各レジスタの設定と、場合によっては『ANSELx』『WPUx』の各レジスタの設定が必要と記載されています。(『○○○x』の x は A、または B で、A ポートを設定するレジスタは『TRISA』のように、ポート B を設定するレジスタは『TRISB』のように表すことを意味しています。)
実際のプログラムでポートの設定を行うにはどのように記述したら良いのでしょう。
データシート 117 ページの右下にポート A を初期化する場合のサンプルプログラムが記載されていますので、これを流用することにしましょう。
　しかし、ここに書かれているプログラムは C 言語ではなく『アセンブリ言語』(アセンブラとも言います)で書かれています。それぞれの命令は下記の赤い文字で記載している内容を行っています。

『アセンブリ言語』は C 言語よりもマイコンが直接理解できる『マシン語』(マシン語は 1 と 0 だけで記述された言語)に近い言語です。
使用するマイコンごとにプログラムの書き方を変える必要がある場合もあります。

EXAMPLE 12-1:　INITIALIZING PORTA

```
; This code example illustrates
; initializing the PORTA register. The
; other ports are initialized in the same
; manner.

BANKSEL  PORTA        ;
CLRF     PORTA        ;Init PORTA
BANKSEL  LATA         ;Data Latch
CLRF     LATA         ;
BANKSEL  ANSELA       ;
CLRF     ANSELA       ;digital I/O
BANKSEL  TRISA        ;
MOVLW    B'00111000'  ;Set RA<5:3> as inputs
MOVWF    TRISA        ;and set RA<2:0> as
                      ;outputs
```

『PORTA』レジスタに『0』を書き込む
『LATA』レジスタに『0』を書き込
『ANSELA』レジスタに『0』を書き込
『TRISA』レジスタに『00111000』を書き込む

このサンプルプログラムを参考に C 言語で書いたものが下記のプログラムです。

```
PORTA = 0;
LATA = 0;
ANSELA = 0;
TRISA = 0;
```

『PORTA』を『0』、『LATA』を『0』、『ANSELA』を『0』に設定までサンプルプログラムと同じで、最後の『TRISA』のみサンプルと異なる『0』にしています。
上記のサンプルプログラムでは『RA3』～『RA5』の 3 コは入力、それ以外は出力に設定していますが、今回作成するプログラムでは全て出力なので『0』としています。

とりあえずサンプルプログラムを参考に C 言語のプログラムを作成しましたが、それぞれのレジスタの役割は次のとおりです。

PORTA レジスタ

REGISTER 12-3:　PORTA: PORTA REGISTER

R/W-x/x	R/W-x/x	R-x/x	R/W-x/x	R/W-x/x	R/W-x/x	R/W-x/x	R/W-x/x
RA7	RA6	RA5	RA4	RA3	RA2	RA1	RA0
bit 7							bit 0

『PORTA』レジスタは、ポート A の状態(『H』なのか『L』なのか)を読み書きするレジスタです。
このレジスタは各ポートの状態により内容が変化しますので、『読み込む』動作をした時、各端子のその時

の状態をチェックすることができます。

『書き込む』ときは、各端子を『H』や『L』に設定することができます。

bit 7-0	**RA<7:0>**: PORTA I/O Value bits(1) 1 = Port pin is > VIH 0 = Port pin is < VIL

ポート A の『RA5』はコンフィグレーション・ビットでデジタル入出力端子に設定しました(MCLRE = OFF)ので、この端子は『MCLRE』機能ではなく、デジタル入力の『RA5』になります。

LATA レジスタ

REGISTER 12-5: LATA: PORTA DATA LATCH REGISTER

R/W-x/u	R/W-x/u	U-0	R/W-x/u	R/W-x/u	R/W-x/u	R/W-x/u	R/W-x/u
LATA7	LATA6	—	LATA4	LATA3	LATA2	LATA1	LATA0
bit 7							bit 0

bit 7-6	**LATA<7:6>**: RA<7:6> Output Latch Value bits(1)
bit 5	**Unimplemented:** Read as '0
bit 4-0	**LATA<4:0>**: RA<4:0> Output Latch Value bits(1)

『LATA』レジスタはポート A の出力をコントロールするレジスタです。『PORTA』レジスタはポートの状態により変化するのに対し、『LATA』はポートの状態の影響を受けないレジスタです。

ANSELA レジスタ

REGISTER 12-7: ANSELA: PORTA ANALOG SELECT REGISTER

U-0	U-0	U-0	R/W-1/1	R/W-1/1	R/W-1/1	R/W-1/1	R/W-1/1
—	—	—	ANSA4	ANSA3	ANSA2	ANSA1	ANSA0
bit 7							bit 0

bit 7-5	**Unimplemented:** Read as '0'
bit 4-0	**ANSA<4:0>**: Analog Select between Analog or Digital Function on pins RA<4:0>, respectively 0 = Digital I/O. Pin is assigned to port or digital special function. 1 = Analog input. Pin is assigned as analog input(1). Digital input buffer disabled.

『ANSELA』レジスタは、ポート A の端子を『デジタル入出力』ポートに設定するか、『アナログ入力』ポートに設定するかを決めるレジスタです。

ポート A はのうち、『RA0』～『RA4』の 5 コは『アナログ入出力端子』としての機能も持っていますので、その端子の設定を行います。

『RA0』～『RA4』の各ポートが『ANSELA』レジスタの『ANSA0』～『ANSA4』の各ビットに対応しており、『ANSELA』の各ビットを『0』に設定するとそれに対応したポートが『デジタル入出力端子』に、『1』に設定すると『アナログ入力端子』になります。

今回は全てデジタル出力に設定するので、『ANSELA』レジスタは『0』に設定します。

『ANSELx』レジスタは各リセット起動後には全ビットが『1』になり、全ポートが『アナログ入力端子』にセットされます。ポートをデジタル入出力端子として使用するとき、マイコンの初期設定で『ANSELx』レジスタの内容を設定することを忘れないように注意しましょう！

TRISA レジスタ

REGISTER 12-4: TRISA: PORTA TRI-STATE REGISTER

R/W-1/1	R/W-1/1	R-1/1	R/W-1/1	R/W-1/1	R/W-1/1	R/W-1/1	R/W-1/1
TRISA7	TRISA6	TRISA5	TRISA4	TRISA3	TRISA2	TRISA1	TRISA0
bit 7							bit 0

bit 7-6	**TRISA<7:6>:** PORTA Tri-State Control bit 1 = PORTA pin configured as an input (tri-stated) 0 = PORTA pin configured as an output
bit 5	**TRISA5:** RA5 Port Tri-State Control bit This bit is always '1' as RA5 is an input only
bit 4-0	**TRISA<4:0>:** PORTA Tri-State Control bit 1 = PORTA pin configured as an input (tri-stated) 0 = PORTA pin configured as an output

『TRISA』レジスタは、各ポートを入力にするか、出力にするか設定するポートです。

『RA0』～『RA7』の各ポートが『TRISA』レジスタの『TRISA0』～『TRISA7』の各ビットに対応しており、『TRISA』の各ビットを『0』に設定するとそれに対応したポートが『出力』に、『1』に設定すると『入力』になります。

『TRISA5』ビットは『R-1/1』となっておりデータを書き込むことができません。そのビットの値は常に『1』で、『入力専用』のポートです。

bit5 が入力なら『TRISA』に書き込むデータは　TRISA = 0;　ではなく　TRISA = 0b00100000;　では？と思った人がいるかもしれません。

『TRISA』のの bit5 は書き込みができないビットになっていますので、そのビットに書き込もうとするデータが『1』でも『0』でも無視されます。したがって、TRISA = 0;と簡単に書くことができます。

もちろん TRISA = 0b00100000;と書いても OK です。

『PORTA』『LATA』レジスタの bit5 や『ANSELA』レジスタの bit5～bit7 は、読み出しのみ、または機能が割り当てられていませんので、どんなデータを書き込むプログラムでもそのビットの命令は無視されます。

WPUA レジスタ

```
nWPUEN = 0;
WPUA = 1;
```

次の 2 行はポート A の入力端子『RA5』に『内部プルアップ』を付けるかどうかの設定です。

ポート A の『RA5』は前述のとおり、コンフィグレーション・ビットでデジタル入出力端子に設定しました(MCLRE = OFF)ので、この端子に内部プルアップを設定します。

REGISTER 12-6: WPUA: WEAK PULL-UP PORTA REGISTER

U-0	U-0	R/W-1/1	U-0	U-0	U-0	U-0	U-0
—	—	WPUA5	—	—	—	—	—
bit 7							bit 0

bit 7-6	**Unimplemented:** Read as '0'
bit 5	**WPUA5:** Weak Pull-up RA5 Control bit If $\overline{\text{MCLRE}}$ in Configuration Word 1 = 0, $\overline{\text{MCLR}}$ is disabled): 1 = Weak Pull-up enabled(1) 0 = Weak Pull-up disabled If $\overline{\text{MCLRE}}$ in Configuration Word 1 = 1, $\overline{\text{MCLR}}$ is enabled): Weak Pull-up is always enabled.
bit 4-0	**Unimplemented:** Read as '0'

Note 1: Global $\overline{\text{WPUEN}}$ bit of the OPTION register must be cleared for individual pull-ups to be enabled.
2: The weak pull-up device is automatically disabled if the pin is in configured as an output.

ポート A でプルアップを設定できる端子は『RA5』だけで、『WPUA』レジスタの『WPUA5』ビットを『1』に設定するとプルアップが有効になります。

しかしこれだけではプルアップは有効になりません。『Note 1 : 』に『それぞれのプルアップを有効にするためには、『オプションレジスタ』の『$\overline{\text{WPUEN}}$』ビットを『0』にする必要があります。』と記載されています。
データシートの 176 ページに『OPTION_REG』レジスタについて記載があります。

REGISTER 20-1: OPTION_REG: OPTION REGISTER

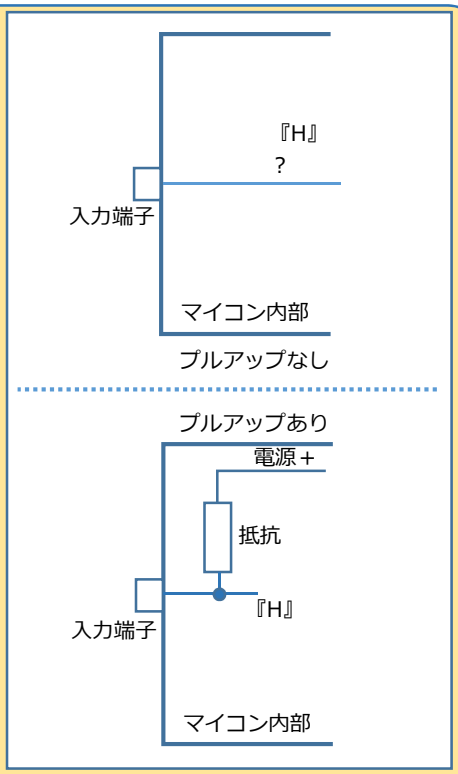
R/W-1/1	R/W-1/1	R/W-1/1	R/W-1/1	R/W-1/1	R/W-1/1	R/W-1/1	R/W-1/1
$\overline{\text{WPUEN}}$	INTEDG	TMR0CS	TMR0SE	PSA	PS<2:0>		
bit 7							bit 0

bit 7 **$\overline{\text{WPUEN}}$:** Weak Pull-up Enable bit
1 = All weak pull-ups are disabled (except $\overline{\text{MCLR}}$, if it is enabled)
0 = Weak pull-ups are enabled by individual WPUx latch values

『$\overline{\text{WPUEN}}$』ビットは bit7 に割り当てられていて、このビットを『0』にしなくてはなりません。
今回は『OPTIO_REG』の他のビットは設定しませんので『$\overline{\text{WPUEN}}$』ビットだけを表す nWPUEN を使用して nWPUEN = 0; と記述し、『WPUA』レジスタ設定の前に書いています。

●プルアップについて

『RA5』ポートに設定した『プルアップ』とは何でしょう？
出力に設定した端子は、その端子の状態をプログラムで『H』、『L』に設定することができます。
しかし、入力に設定した端子は、『H』、『L』などの状態がその端子に入力される信号により決まります。もしその入力端子がどこにもつながっていなかったらどうなるでしょう？
そのような場合、その入力端子は状態が不定となります。
入力が不定になるとマイコンが誤動作したり、最悪の場合はマイコンが壊れたりすることもあります。
この状態を防止するためには、その端子を安定な状態にする必要があります。そのため、マイコン内部で『数十 kΩ』の抵抗を介して、電源＋と接続します。このことを『プルアップ』と言い、こうすることで入力端子を『H』の状態にすることができます。
マイコンの使用しない端子は『出力』に設定するか、入力に設定する場合には『プルアップ』を付けるようにしましょう。



※PICA Tower の回路は、マイコンの外部の 10kΩ 抵抗で RA5 端子がプルアップされていますので、本来ならこのプルアップの設定は必要ありませんが、プルアップの設定方法の説明のために設定しています。

ポート A の初期設定はこれで完了です。

次の 4 行はポート B の設定です。

```
PORTB = 0;
LATB = 0;
ANSELB = 0;
TRISB = 0;
```

ポート A の時と同様に設定します。それぞれのレジスタはデータシートの 127 ページから記載があります。

ポート B がポート A と異なる点は、全ての端子が入力、出力のどちらにも設定できることと、入力に設定した場合の内部プルアップが、全てのポートに設定できる点です。

今回の場合、ポート B は全て出力に設定しましたので、内部プルアップは設定しません。

これで発振回路の周波数や各ポートの設定は完了です。

・点灯パターンの記述

while(1)で始まる無限ループの部分が点灯パターンの内容を決めるプログラムです。

```
while(1){

PORTB = 0xFF;              //ポート B を全て『H』に
PORTA = 0x00;              //ポート A を全て『L』に

}
```

無限ループですから、{　}で囲まれた部分を電源を OFF にするまで繰り返すことになります。

PICA Tower の回路では、各段の LED のアノード側は『RB0』～『RB7』と『RA7』に、カソード側は『RA0』～『RA2』につながっていましたね。（本書 12 ページ：PICA Tower の回路図を参照。）

また、LED が点灯するためにはアノード側が『H』、カソード側が『L』の場合のみでしたね。

点灯パターンのプログラムでは、最初に

```
PORTB = 0xFF;              //ポート B を全て『H』に
```

と書かれています。『0xFF』は 16 進数なので、これを 2 進数にすると『0b11111111』です。

PORTB レジスタはそれぞれのビット(bit0～bit7)がポート B の『RB0』～『RB7』に対応していて、『1』を書き込むと『H』に、『0』を書き込むと『L』になるのでしたね。

PORTB = 0xFF;　は、『RB0』～『RB7』を全て『H』にする命令というわけです。

では

```
PORTA = 0x00;              //ポート A を全て『L』に
```

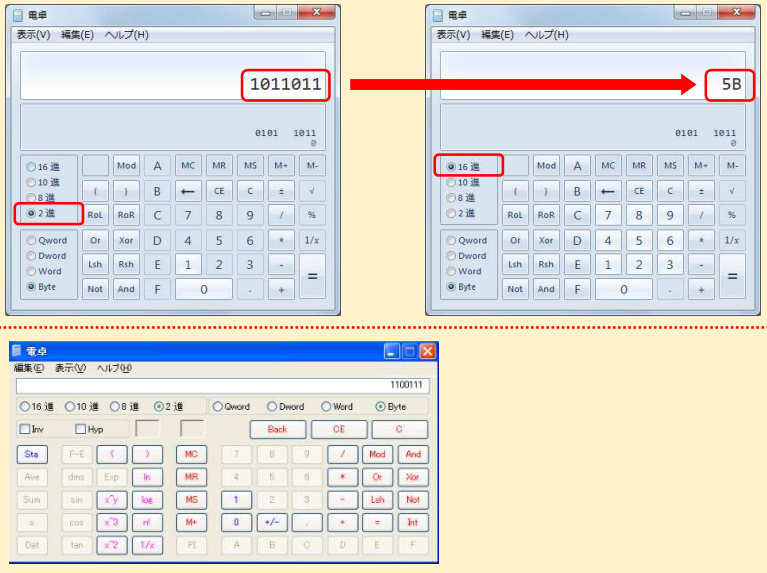
この命令は PORTA レジスタを全て『L』にする命令ですね。

この命令を実行すると LED はどのように点灯するか考えてみてください。

Windows の電卓は便利！

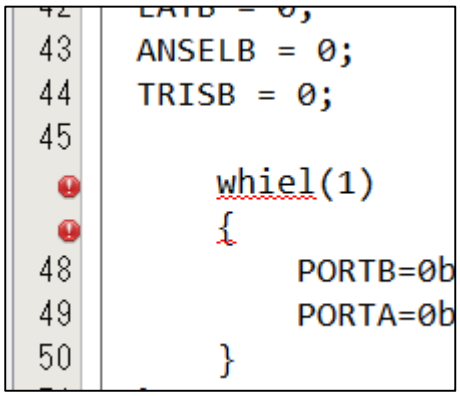
Windows パソコンに標準でインストールされている『電卓』は 2 進数や 16 進数の計算に便利です。
2 進数や 16 進数は慣れるまでわかりにくいかもしれませんが、2 進数を 16 進数に変換したりする時にとても重宝します。
右にある画面の上 2 つは、Windows7 の電卓の画面です。メニューの『表示』から『プログラマー』を選ぶとプログラミングに便利な電卓モードになります。
電卓画面の左に 2 進数～16 進数のボタンがあります。入力する進数を選び数値を入力したら、変換したい進数にボタンを押すとその進数に変換してくれます。
また、AND や OR、XOR などの計算もできますので、計算結果を確かめるときなどにも便利です。
下の画面は WindowsXP の電卓です。こちらは『表示』から『関数電卓』を選びます。

※データ型の設定などにより電卓の計算結果とプログラムの内容が異なる場合があります。



5. プログラムをマイコンへ書き込む方法

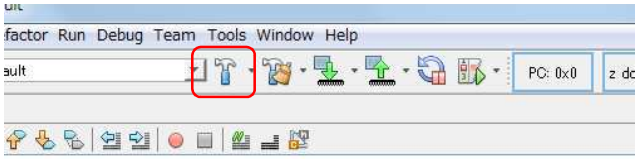
プログラムの入力が終わったら、そのデータをマイコンに書き込む準備を行います。
MPLAB X は、プログラムを入力している途中にその記述に間違いがあった場合、その左側の行番号のところに『 』が表示されます。



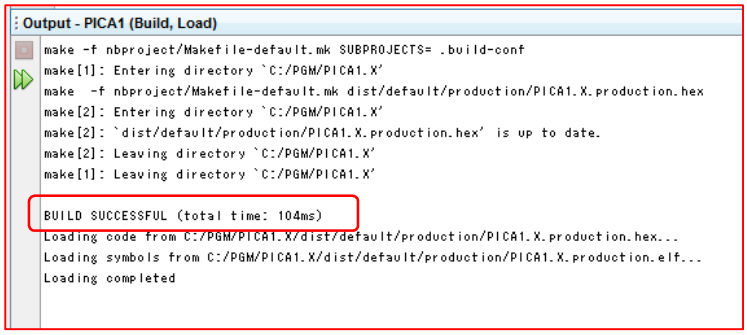
しかし、プログラムのすべての場合の間違いに表示されるわけではなく、プログラムの書き方でつじつまが合うような間違いをしていると表示されませんので注意してください。
左の例では『while』と書くべきところを『whiel』と間違ったスペルで書いてしまったため、『この whiel を識別できません！』とエラーになっています。

ビルド

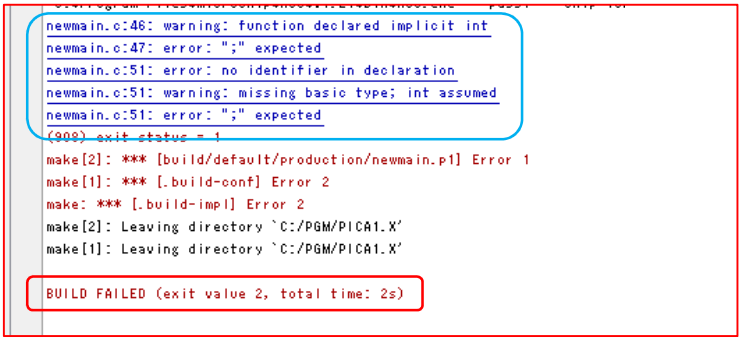
プログラム入力でエラー表示がなくなったら、入力したソースファイルを PIC マイコンを動作させるためのファイルを生成します。この作業を『ビルド』といいます。
MPLAB X の上部にあるツールバーのハンマーの形をしたマークをクリックします。
メニューバーの『Run』→『Build Project(プロジェクト名)』でも OK です。



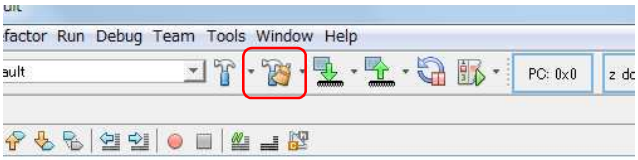
すると、画面下部に『Output-(プロジェクト名)』ウインドウが開き、ビルドの結果を表示します。



この画面の下の方に『BUILD SUCCESSFUL』と表示されていればビルドは完了です。



しかし、プログラムに間違いがあると『BUILD FAILED』と表示され、エラーになります。
エラーになってしまった場合、プログラムの間違い箇所が青字で表示されますので、その文字をクリックすると、ソースファイルの間違い箇所を表示してくれます。



ツールバーのハンマーのマークの横に、ハンマーとほうきがくっついたマークのボタンがあります。
このボタンは『クリーンアンドビルド』ボタンで、メニューの『Run』→『Clean and Build Project(プロジェクト名)』でも同じです。

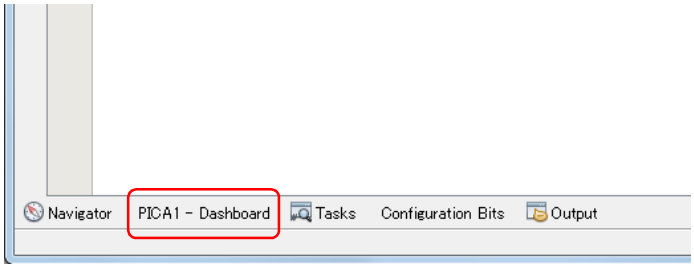
クリーンアンドビルドは、以前ビルドしたプロジェクトを再ビルドするとき、前回のビルドで作成されたファイルを一旦消してからビルドする時に使います。

プログラムの書き込み

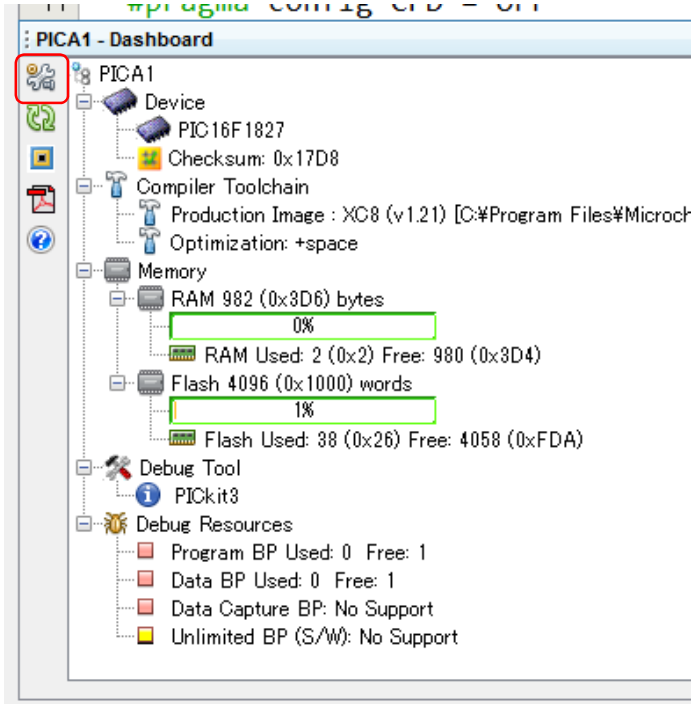
ビルドが完了したらマイコンにプログラムを書き込みましょう。

・PICkit3 の接続とセッティング

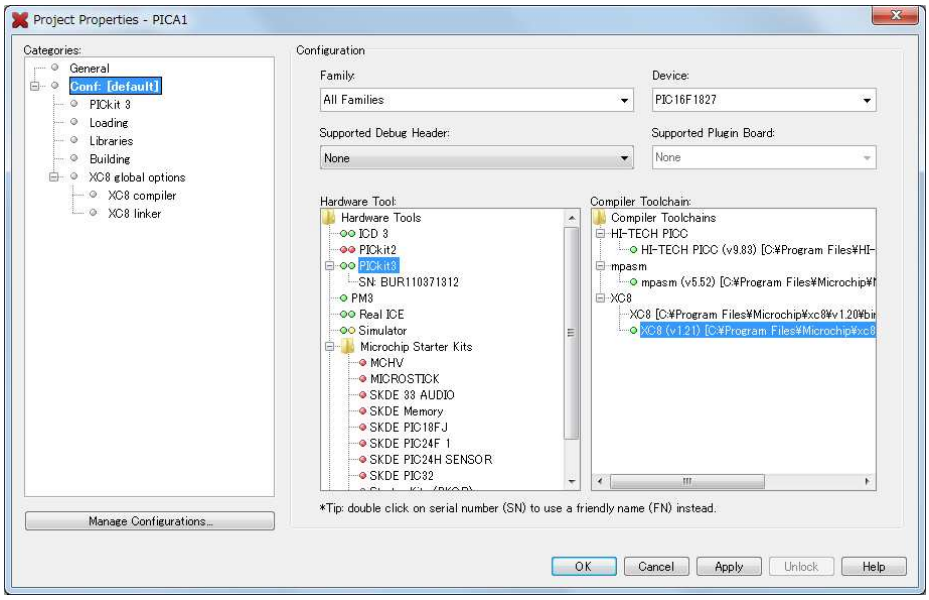
パソコンの USB ポートに PICkit3 を接続します。



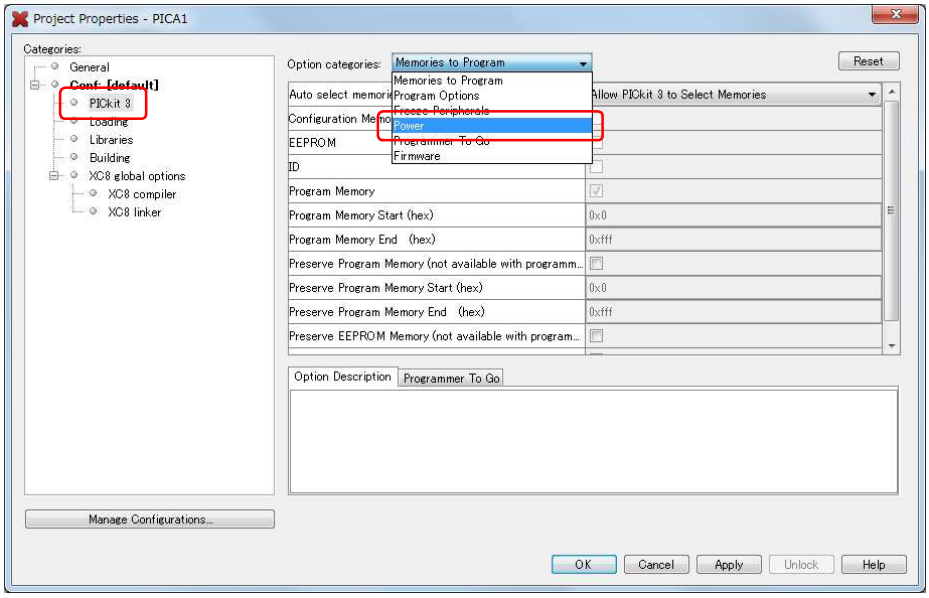
次に、画面下部にある『(プロジェクト名)ー Dashboard』をクリックします。
メニューバーの『Window』→『Dashboard』で表示することもできます。



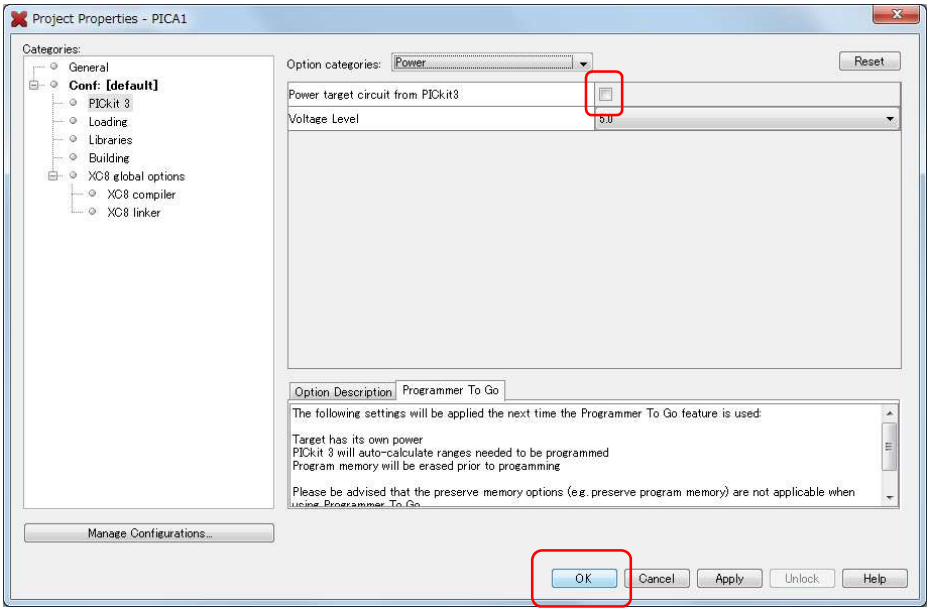
すると、左のような画面が表示されます。
この画面は、プロジェクトの設定内容(使用するマイコンの型番や書き込みに使用する機器、マイコン内のメモリーの消費量など)が表示される『ダッシュボード画面』です。
この画面の左上にあるマークをクリックします。



左のような画面が表示されます。各項目が、最初のプロジェクトの作成時に設定した内容の通りに設定されていますが、その内容を変更したいとき(使用するマイコンやコンパイラの種類、使用する書き込み器など)に、ここで変更することができます。



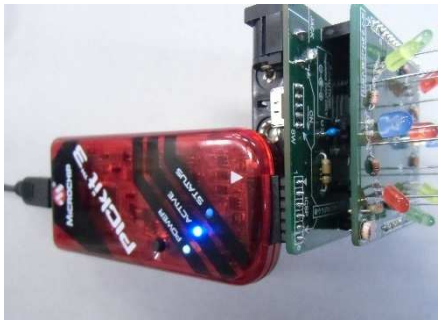
今回は左の『Categories』から『PICkit3』を選択し、右側最上部の『Option categories』の『Power』を選択します。



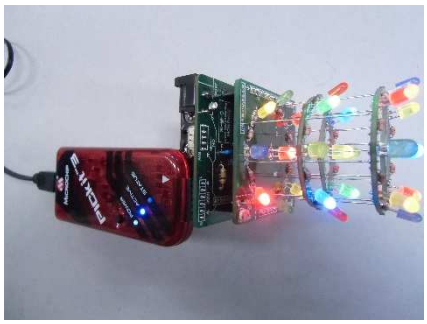
この画面が表示されたら、『Power target circuit from PICkit3』のチェックをはずしておきます。
これは、マイコンにプログラムを書き込むとき、マイコンが搭載された回路の電源を PICkit3 から供給するかどうかを設定するものです。
今回の回路では、その電源は乾電池から供給しますのでチェックをはずし、PICkit3 からは供給しない設定にします。
この設定が終わったら下方にある『OK』をクリックします。

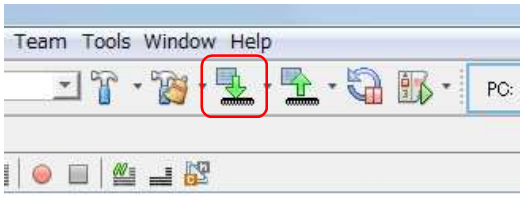
・マイコンへの書き込み

PICA Tower の電源を OFF にして、スイッチの横にある『ICSP』端子に、PICkit3 を接続します。
この時、ICSP 端子の『▼』マークのピンに PICkit3 の『▲』マークの端子が来るように差し込みます。

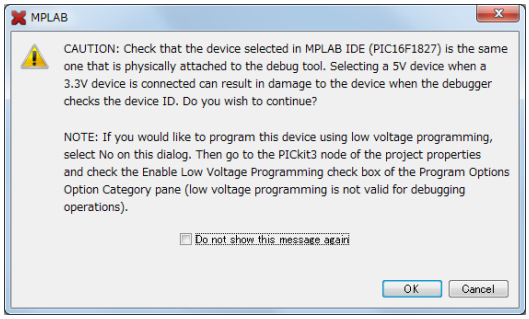


差し込んだら、PICA Tower の電源をONにします。
ONにするるとあらかじめ書き込まれている点灯パターンで LED が点灯します。

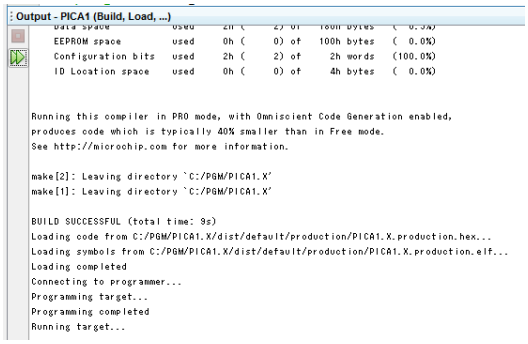




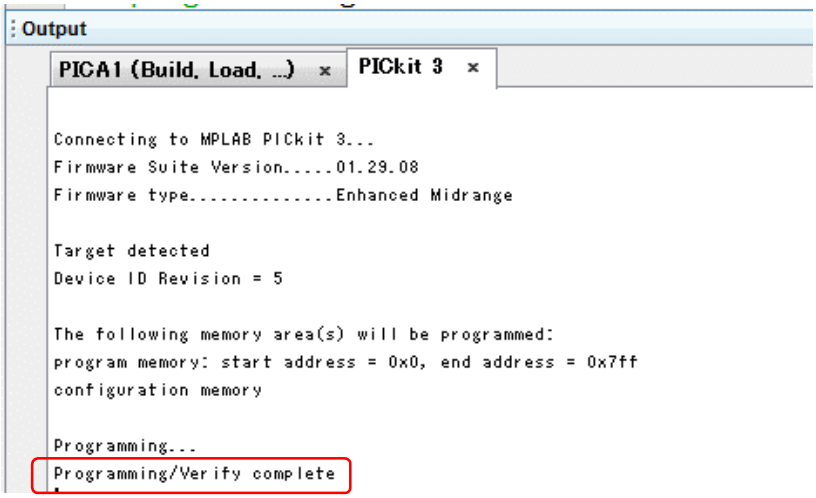
MPLAB X のツールバーの『Make and Program Device』ボタンをクリックします。



左のメッセージが表示されます。これは電源についての注意を促す注意文ですが、問題ないので『OK』をクリックします。

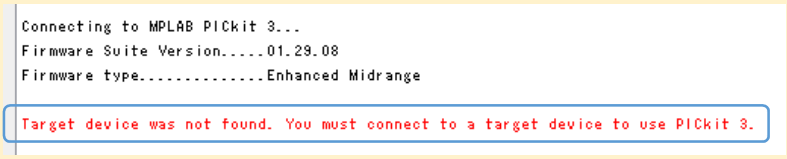


すると、プロジェクトのビルドが始まり、『BUILD SUCCESSFUL』になると『PICkit3』タブが表示されます。



接続している PICkit3 の情報などが表示された後、マイコンへの書き込みが始まり、『Programming / Verify complete』と表示されれば完了です。
PICA Tower は新しい点灯パターンで点灯しているはずですので、一旦 PICA Tower のスイッチを OFF にして PICkit3 をはずします。

PICkit3 は、使用するマイコンに合わせて、PICkit3 のファームウエアを書き換えるようになっていますので、書き込みが始まる前にファームウエアのダウンロード、書き込みを行う場合があります。



書き込み中に左記のようなメッセージが表示された場合、書き込みをしようとしているマイコンがつながっていない状態です。
PICA Tower の電源スイッチが ON になっているか、PICkit3 を取り付ける向きは合っているかなど、再度チェックしてください。

今回作成した点灯パターンは、真ん中の縦 3 つ以外がずっと点灯したままになるプログラムです。
プログラムを書き込む際に思っていた点灯パターン通りでしょうか？

問題

今回作ったプログラムを、真ん中の縦 3 つの LED も点灯するようなプログラムに変更してください。

ヒント：真ん中の LED のアノード側はどのポートに、またカソード側はどのポートにつながっているかな？　また LED が点灯する条件は？

```
while(1)
    {
        PORTB=0xFF;
        PORTA=0x00;
    }
```

→

```
while(1)
    {
      ?
      ?
    }
```

答え

真ん中の LED のアノード側はポート A の bit7(A7)、カソード側他の LED と同じく、1 段目(一番上)がポート A の bit0(A0)、2 段目(真ん中の段)がポート A の bit1(A1)、3 段目(一番下)がポート A の bit2(A2)につながっています。LED が点灯するためには、アノード側が『H』、カソード側が『L』です。

LED が全部点灯するためには、ポート B の全てと A7 が『H』、A0、A1、A2 が『L』になればよいわけです。

これを各レジスタの設定に当てはめ、わかりやすいように 2 進数と 16 進数で書いてみると下記の表の通りですから、答えはこのようになります。

2 進数	16 進数
PORTB=0b11111111;	PORTB=0xFF;
PORTA=0b10000000;	PORTA=0x80;

なので →

```
while(1)
    {
        PORTB=0xFF;
        PORTA=0x80;
    }
```

実際にプログラムを入力し、マイコンに書き込んで確かめましょう。

6. オリジナル点灯パターンをつくろう！

ここからは、自分の好きな点灯パターンを作る方法や、明るさセンサーを使って、周囲の明るさによって点灯をコントロールする方法を学習しましょう。

LED の点滅

最初に作ったプログラムは、LED が点灯したままのものでした。

では、ピカピカと点滅させるにはどうしたらよいでしょう。そですね。点灯と消灯を繰り返せばよいはずです。

では、点灯パターン部分のプログラムを考えてみましょう。

```
while(1)
    {
      PORTB=0xFF;
      PORTA=0x00;
      PORTB=0x00;
    }
```

左のプログラムでどうでしょう？

{　}内の最初の 2 行は、円形部分の LED が点灯するプログラムですね。

3 行目は LED が点灯しない条件のアノード側を『L』にする命令ですから、LED は消灯することになります。この時ポート A は『L』のままで OK ですから、ポート A の操作はしていません。

では先に作ったプログラムを変更して、マイコンに書き込んでみましょう。

どうなりましたか？

ピカピカと点滅・・・しませんね！　なぜでしょう？

これは、プログラムの実行速度がとても早いため、点灯と消灯を繰り返す時間が非常に短く、人間の目には常に点灯しているように見えてしまうからです。

では、人間の目にもちゃんとピカピカと点滅して見えるようにするためにはどうすればよいでしょう？

点灯のあと、消灯のあとに、しばらくそのまま待機するようにして、人間の目にも点滅が見えるようにすればよいのです。

```
#include <xc.h>

#define _XTAL_FREQ 500000            //システムクロックは 500kHz=500000Hz

// CONFIG1

        .
    OSCCON = 0b00111000;             //システムクロックはデフォルトの 500kHz に
        .
        .
        .
        .
        .

    while(1){

        PORTB = 0xFF;                //ポート B を全て『H』に
        __delay_ms(500);             //500ms=0.5 秒待つ
        PORTA = 0x00;                //ポート A を全て『L』に

        PORTB = 0x00;                //ポート B を全て『L』に
        __delay_ms(500);             //500ms=0.5 秒待つ

        }
}
```

上記プログラムの赤字で書いた部分が待ち時間を作るためのプログラムです。

```
#define _XTAL_FREQ 500000
```

まず最初に、このマイコンが動いている発振周波数を宣言します。
周波数は Hz(ヘルツ)の単位で書きます。

今回は初期設定で 500kHz に設定しましたね。500kHz=500000Hz なので、このような書き方になります。

『#define』を使った行の最後には『;』は付けません。

```
__delay_ms(500);
```

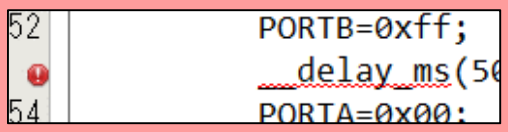
下のこの部分が待ち時間を作る命令です。

書き方	意味	ヘッダーファイル
__delay_us(待ち時間);	1μs(マイクロ秒)単位で待ち時間を設定する	xc.h
__delay_ms(待ち時間);	1ms(ミリ秒)単位で待ち時間を設定する	
_delay (待ち時間);	1 命令サイクル単位で待ち時間を設定する	

『__delay_us(待ち時間);』『__delay_ms(待ち時間);』では、最初の『_(アンダーバー)』は『2 つ』です。

『ヘッダーファイル』は、この命令を使用する時に読み込んまなければならない外部ファイルで、『#include』で読み込んでおきます。この待ち時間の命令は『xc.h』ヘッダーファイルに定義などが書いてあるので、これを読み込んまないと待ち時間命令は使用できません。

MPLAB X と CX8 コンパイラで__delay_us( )または__delay_ms( )を使うと、本書の 54 ページで書いた、間違いがあった場合につくエラーが表示されてしまいます。



しかし、この状態でビルドしてもエラーになりませんし、マイコンに書き込んでも正常に動作します。どうやら MPLAB X のバグ(?)のようです。このエラーは気にしないでおきましょう。

また、__delay_ms( )または us( )の『( )』の中は変数は使用することができず、実数しか扱うことができません。

```
int t=100;
 if(t>0)
  {
   __delay_ms(t);
   --t;
  }
```

例えば、
左のようなプログラムを書くことができませんので注意しましょう。
待ち時間を変数にするテクニックもありますので、後ほど解説します。

『#define』

周波数の宣言で#define を使用しました。#define は周波数の宣言だけでなく、識別子や定数を『ほかの識別子』で表すことができます。
例えば、ポート B の『bit0』は LED1 のアノード側、ポート A の『bit2』は 3 段目のカソード側だとわかりやすいように

```
#define LED1_A RB0              //RB0 は LED1_A という名前にする
#define COLUM3_K RA2            //RA2 は COLUM3_K という名前にする
```

と、別名を付けてわかりやすくすることができます。

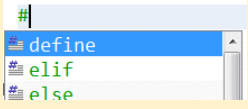
```
#define _XTAL_FREQ 500000
```

周波数の宣言で書いたこの文はマイコンの周波数である 500000 を『_XTAL_FREQ』という識別子で表しているのです。

```
#define AAA BBB+3
```

#define は、識別子や定数だけでなく、式をほかの識別子で表すこともできます。

#define で置き換えた識別子は『変数』ではありませんので、その内容が変わることはありません。



MPLAB X では、プログラムの入力時に『#』を入力すると、#を使う文の一覧が表示されますので、使用する文をダブルクリックして入力します。

では、プログラムに待ち時間の命令を書き加え、マイコンに書き込んでみてください。
今度は 0.5 秒おきにピカピカと点滅するはずです。

LED のシフト

では次の問題です。

問題

1 段目の LED 全て→2 段目の LED 全て→3 段目の LED 全て→1 段目の LED・・・・と、0.3 秒おきに点灯するようなプログラムを組んでみましょう。

答え

1 段目の LED 全てを点灯させるためにはポート B 全てを『H』＋ポート A の bit7 を『H』で、ポート A の bit0 が『L』の場合。
2 段目の LED 全てを点灯させるためにはポート B 全てを『H』＋ポート A の bit7 を『H』で、ポート A の bit1 が『L』の場合。
3 段目の LED 全てを点灯させるためにはポート B 全てを『H』＋ポート A の bit7 を『H』で、ポート A の bit2 が『L』の場合。
ということは、ポート B 全てとポート A の bit7 は常に『H』で、ポート A の bit0、bit1、bit2 を 0.3 秒毎に『L』にすればよいわけですね。

```
while(1){

PORTB = 0b11111111;     //16 進数だと 0xff

PORTA = 0b10000110;     //16 進数だと 0x86
__delay_ms(300);        //0.3 秒=300ms 待つ

PORTA = 0b10000101;     //16 進数だと 0x85
__delay_ms(300);        //0.3 秒=300ms 待つ

PORTA = 0b10000011;     //16 進数だと 0x83
__delay_ms(300);        //0.3 秒=300ms 待つ

}
```

このように、『点滅』と『点灯させる場所』を変えていくことで LED が流れるように点灯するパターンを作成することができます。

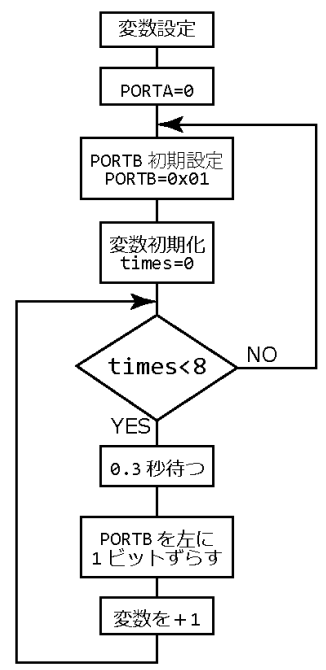
では、ちょっと応用です。縦一列をクルクル回るように点灯させる場合はどうなりますか？
真ん中の縦一列は消灯したままとしましょう。

```
while(1)                        //無限ループ
{
        PORTA = 0;              //PORTA は全て常に『L』

        PORTB = 0b00000001;     //PORTB の bit0 だけ『H』
        __delay_ms(300);        //0.3 秒=300ms 待つ

        PORTB = 0b00000010;     //PORTB の bit1 だけ『H』
        __delay_ms(300);        //0.3 秒=300ms 待つ

        PORTB = 0b00000100;     //PORTB の bit2 だけ『H』
        __delay_ms(300);        //0.3 秒=300ms 待つ

        PORTB = 0b00001000;     //PORTB の bit3 だけ『H』
        __delay_ms(300);        //0.3 秒=300ms 待つ

        PORTB = 0b00010000;     //PORTB の bit4 だけ『H』
        __delay_ms(300);        //0.3 秒=300ms 待つ

        PORTB = 0b00100000;     //PORTB の bit5 だけ『H』
        __delay_ms(300);        //0.3 秒=300ms 待つ

        PORTB = 0b01000000;     //PORTB の bit6 だけ『H』
        __delay_ms(300);        //0.3 秒=300ms 待つ

        PORTB = 0b10000000;     //PORTB の bit7 だけ『H』
        __delay_ms(300);        //0.3 秒=300ms 待つ

}
```

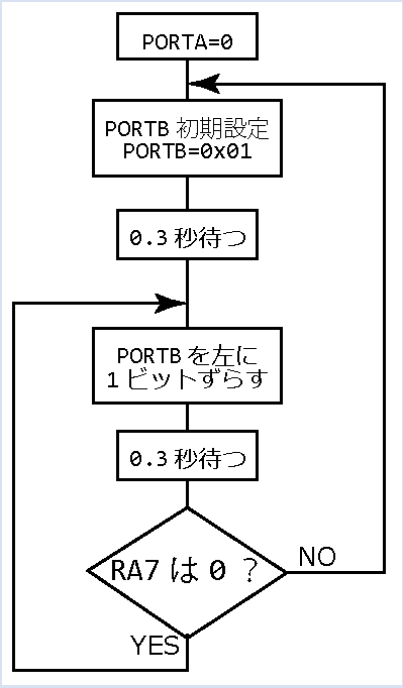
ポート A の bit0～bit2 は常に『L』にしておき、ポート B を 1 ビットずつ横にずらしていけば OK ですね。

しかし、このプログラムの書き方ではすごく長くなり、とても見づらくなりますね。
『演算子』に、指定したビット分ずらす『シフト演算子』というモノがありましたね。
また、回数を数えながら繰り返す『for 文』も使えそうです。
では、シフト演算子と for 文を使用したプログラムを考えてみましょう。

フローチャート	プログラム

	`int times;` //回数を数える変数を設定
	`PORTA = 0;` //PORTA は全て常に『L』
	
	`while(1)`
	`{`
	`PORTB = 0b00000001;` //PORTB の bit0 は最初『H』
	
	`for(times=0;times<8;times++)` //1 周するまで数える
	`{`
	`__delay_ms(300);` //0.3 秒=300ms 待つ
	`PORTB <<= 1;` //PORTB を 1 ビット左にシフト
	`}`
	`}`

この他にも do-while 文を使った例も考えて見ました。

フローチャート	プログラム
	


```
PORTA = 0;                  //PORTA は全て常に『L』


while(1)
    {
        PORTB = 0b00000001;     //PORTB の bit0 は最初『H』
        __delay_ms(300);        //0.3 秒=300ms 待つ

        do{
       PORTB <<= 1;             //16 進数だと 0x86
        __delay_ms(300);        //0.3 秒=300ms 待つ
        }

        while(RB7==0);          //PORTB の bit7 までシフトして 1 に
                                //なればこのループを抜ける

    }
```

このように、同じ点灯パターンでも、プログラムの書き方はいくつもあります。
慣れるまでは自分のわかりやすい書き方をして、その他にどんな書き方があるか考えてみるのもよいでしょう。

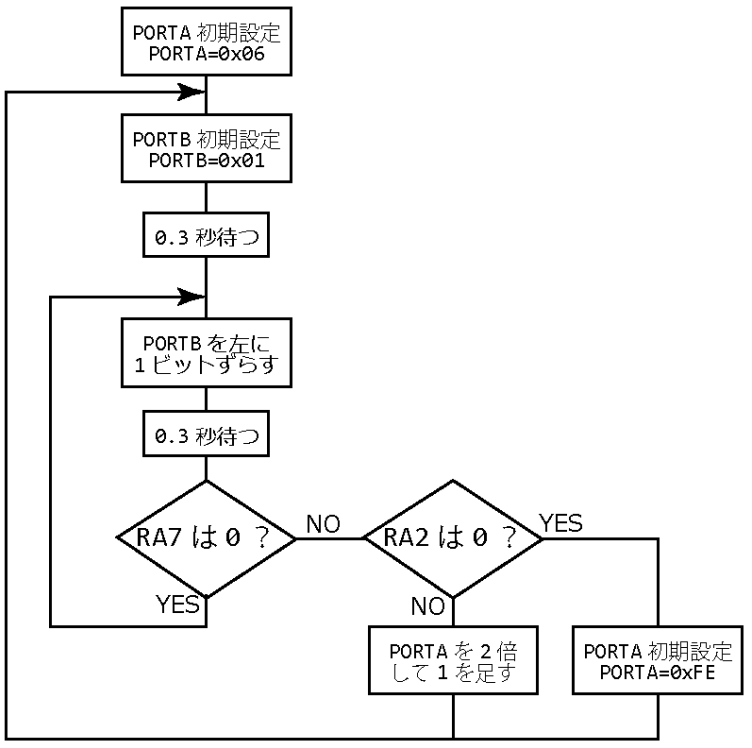
問題

1 段目の LED だけを 1 つずつシフトしながら 1 周したあと、2 段目 LED を 1 周、その後 3 段目の LED を 1 周させる。これを繰り返すプログラムを組んでみましょう。
LED がシフトする間隔は 0.1 秒で設定してください。

答え

『1 段目の LED だけを 1 つずつシフトしながら・・』は、PORTB の bit0 だけ『H』に設定し、それを 1 ビットずつシフトさせ、この時の PORTA は bit0 だけを『L』にすれば OK です。
『2 段目の・・・』は PORTB は 1 段目の時と同じですが、PORTA を bit1 だけ『L』に切り替えます。『3 段目・・・』の時も同じで、PORTA の bit2 だけ『L』にすれば OK ですね。
ちょうど 62 ページの問題と 63 ページの例題をミックスしたような点灯パターンですね。
シフト演算子や制御文の組み合わせで実　現できそうです。

フローチャート	プログラム

	`PORTA = 0x06;  //PORTA 初期化`
	`//下位 3 ビットを 110 に`
	
	`while(1){`
	
	`PORTB = 0x01;  //PORTB は bit0 だけ『H』`
	`__delay_ms(100);  //0.1 秒待つ`
	
	`do{`
	`PORTB <<= 1; //PORTB を 1 ビットシフト`
	`__delay_ms(100);  //0.1 秒待つ`
	`}`
	`while(RB7==0);  //RB7 までシフト？`
	
	`if(RA2==0){  //RA2 までシフト？`
	`PORTA = 0x06;  //PORTA 初期化`
	`}`
	
	`else{`
	`PORTA = LATA*2+1;`
	`//PORTA の『L』ポートを`
	`//1 ビット分シフト`
	`}`
	`}`

複雑なプログラムを考えるとき、フローチャートを書いてプログラムの流れを考えるとよりわかりやすくなります。

このプログラムのポイントのひとつは、最初のポート A の初期値を無限ループの外に置いていることです。無限ループの内側に置くと、ポート A を 1 ビットずらして 2 段目を点灯させようとした時、ポート A も初期化されてしまい、ず～っと 1 段目だけしか点灯しないことになります。

2 つ目ののポイントはポート A を左にシフトするプログラムです。

`PORTA = LATA*2+1;` がポート A をシフトする操作です。

ここで『LATA レジスタ』を使用していることもポイントなのですが、これは次のような理由があります。

ポート A、ポート B の出力状態を操作するとき『PORTA = 0x5f』のように PORTx レジスタを使用するのが一般的ですが、PORTx レジスタは『実際のポートの状態(値)を読む』入力にも使用されるレジスタでもあります。

ただし、レジスタの値を書き込む命令の中には

『一旦、レジストの内容を全て読む』→『値を書き込むビットの内容を設定』→『レジスタに書き込む』

のようなプロセスで実行されるものがあります。このような命令を『リード・モディファイ・ライト命令』といいます。

今回のプログラムのように PORTA レジスタを使用してシフトする命令を実行する時、一旦 PORTA の状態を全て読み込みます。PICA Tower の回路はプログラム書き込みに使用する Vpp 端子が RA5 を兼ねており、その端子は抵抗でプルアップされているため、RA5 の値は『H』です。この状態で PORTA を 2 ビットシフトさせると RA7 が『H』に設定され、中心の LED の A 側が『H』になるため点灯してしまいます。

そのような場合には、PORTx レジスタのかわりに LATx レジスタを使います。
LATx レジスタは入力には使用されないため、ポートの状態に関係なくレジスタの内容を設定することが可能です。
PIC16F1827 データシートの LATx レジスタの説明部分(122、127 ページ)には、次の記載があります。

Note 1: Writes to PORTA are actually written to corresponding LATA register.

これは、『PORTx レジスタに値を書き込むことは、それに対応する LATx レジスタに書き込むことと同じ結果になります。』と書いてります。ですから、LATx レジスタに値を書き込んでも、出力ポートの値を設定することが可能です。
出力ポート操作のを行う場合、その時のポートの状態に左右されないよう、PORTx レジスタではなく LATx レジスタを使用したり、PORTx レジスタの全てを書き換えるような工夫が必要な場合があることを覚えておきましょう。

ポイントの 3 つ目は PORTA の bit0～bit2 の『L』の値を左にシフトしていく処理の書き方です。

```
PORTA = LATA*2+1;
```

このプログラムでは左のように、LATA レジスタを 2 倍して 1 を足したものを PORTA い書き込んでいます。

```
LATA <<= 1;
```

このように、単純に左にシフトする演算子を使っても良いのではないかと疑問がわきませんか？

シフト演算子を使用する後者では、シフトした後の bit0 に『0』が入ってきますので、演算した結果、ポート A は『00001100』となってしまい、2 段目だけでなく 1 段目も点灯してしまいます。なので 1 を足して、最下位ビットを『1』にしています。
しかし、なぜ『*2』なのでしょう？　実はここは『LATA*2』ではなく『LATA<< 1』でもよいのです。『ポートを左に 1 ビットシフトする』ということは『2 倍する』と同じことなのです。
54 ページに書いた Windows の電卓を使って確かめてみてください。
まず 2 進数で『110』と入力し 16 進数になおします。すると『6』(0x06)と表示されますね。
では 16 進数のまま、6 に 2 をかけてください。すると『C』と表示されます。それを 2 進数に直してみましょう。すると『1100』と表示されましたね。
これを 8 ビット表記で書くと、最初の値は『00000110』。これに 2 をかけると『00001100』となり、左に 1 ビットシフトして最下位ビットに『0』が入りました。これはシフト演算子『<< 1』と同じ結果になります。
2 進数では『左に 1 ビットシフトする』ことは『1 桁上がる』ことを意味し、2 進数で 1 桁上がることは『2 倍になる』ことを意味するのです。

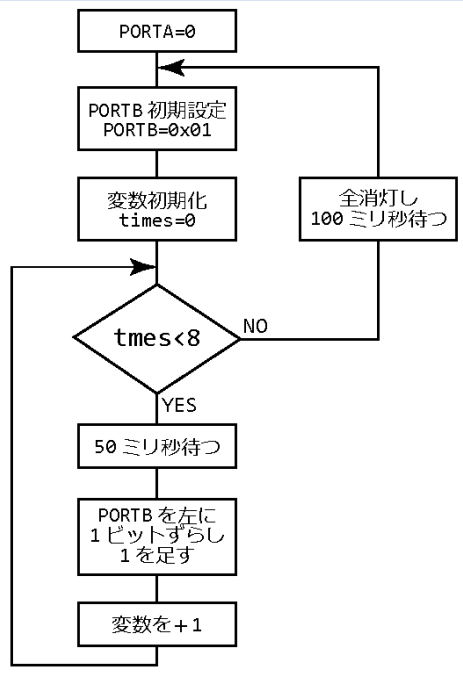
では、LED のシフト点灯の問題をもう一つ。

問題

外周の LED の縦 1 列を回りながら全て点灯させるプログラムを組んでみましょう。
LED がシフトする間隔は 50 ミリ秒で設定してください。
62 ページの応用問題と似ていますが、この問題では点灯する位置がシフトするのではなく、だんだん増えていくように点灯させましょう。

答え

縦 1 列の LED が点灯するので、PORTA の bit0～bit2 は全て『L』ですね。
PORTB は最初 bit0 だけ『H』に設定してそれを 1 ビットずつシフトさせ、シフトした後の最下位ビットに入ってくる『0』を『1』に変えてやれば OK ですね。
そして 1 周するまでカウントし、その後初期状態に戻してやれば OK です。

フローチャート	プログラム
	


```
int times;              //回数を数える変数を設定
PORTA = 0;              //PORTA は全て常に『L』

while(1)
{
       PORTB = 0x01;    //PORTB の bit0 は最初『H』

        for(times=0;times<8;times++)    //1 周するまで数える
        {
          __delay_ms(50);           //50ms 待つ
       PORTB <<= 1;                 //PORTB を 1 ビット左にシフト
       PORTB += 1;                  //PORTB を 1 ビット左にシフト

        }

        PORTB = 0;                  //POTRB を全て『L』(全消灯)
        __delay_ms(100);            //100ms 待つ
}
```

関数を使った書き方

これまでは 1 種類の点灯パターンを繰り返すプログラムを考えてきました。

では次に、複数の点灯パターンを順番に繰り返すプログラムを考えてみましょう。

例題として、本書の 54 ページ『点灯パターンの記述』前ページまでの点灯バターンのプログラムを繰り返すプログラムを作ってみます。

54 ページの全点灯は 1 秒、その後のプログラムは 3 回ずつ行うプログラムにしましょう。

```
/*メイン関数より前は省略しています。(実際は要記述)*/

void main(void) {
//初期設定
    OSCCON = 0b00111000;              //システムクロックはデフォルトの 500kHz に
    OSCTUNE = 0;                      //クロックチューニングはなし
    PORTA = 0;                        //ポート A を全て『L』に
    LATA = 0;                         //ポート A のデータラッチレジスタをポート A と同じ値に
    ANSELA = 0;                       //ポート A をデジタル I/O に
    TRISA = 0;                        //ポート A は RA5 以外を出力に(RA5 は入力専用)
    nWPUEN = 0;                       //OPTION_REG の WPUEN を『L』にし、ウィークプルアップを有効に
    WPUA = 1;                         //ポート A にプルアップ設定(RA5 のみ)
    PORTB = 0;                        //ポート B を全て『L』に
    LATB = 0;                         //ポート B のデータラッチレジスタをポート A と同じ値に
    ANSELB = 0;                       //ポート B をデジタル I/O に
    TRISB = 0;                        //ポート B は全て出力に

    while(1){

         PORTB = 0xFF;                //ポート B を全て『H』に
         PORTA = 0x00;                //ポート A を全て『L』に
         __delay_ms(1000);

         int a;                       //繰り返し用の変数設定

        for(a=0;a<3;a++){
             PORTB = 0xFF;            //ポート B を全て『H』に
             __delay_ms(500);         //500ms=0.5 秒待つ
             PORTA = 0x00;            //ポート A を全て『L』に
             PORTB = 0x00;            //ポート B を全て『L』に
             __delay_ms(500);         //500ms=0.5 秒待つ
                 }

         for(a=0;a<3;a++){
             PORTB = 0b11111111;      //16 進数だと 0xff
             PORTA = 0b10000110;      //16 進数だと 0x86
             __delay_ms(300);         //0.3 秒=300ms 待つ
             PORTA = 0b10000101;      //16 進数だと 0x85
             __delay_ms(300);         //0.3 秒=300ms 待つ
             PORTA = 0b10000011;      //16 進数だと 0x83
             __delay_ms(300);         //0.3 秒=300ms 待つ
                 }

             PORTA = 0;               //PORTA は全て常に『L』

         for(a=0;a<3;a++){
             PORTB = 0b00000001;      //PORTB の bit0 は最初『H』
             __delay_ms(300);         //0.3 秒=300ms 待つ
             do{
            PORTB <<= 1;              //16 進数だと 0x86
             __delay_ms(300);         //0.3 秒=300ms 待つ
             }
                    while(RB7==0);    //PORTB の bit7 までシフトして 1 になればこのループを抜ける
                 }

           PORTA = 0x06;       //PORTA 初期化。下位 3 ビットを 110 に
           for(a=0;a<3;a++){   //3 回繰り返し
                  PORTB = 0x01;       //PORTB は bit0 だけ『H』
              __delay_ms(100);        //0.1 秒待つ
```

```
                do{
               PORTB <<= 1;          //PORTB を 1 ビットシフト
                __delay_ms(100);     //0.1 秒待つ
                   }
                while(RB7==0);       //RB7 までシフト？
            if(RA2==0){              //RA2 までシフト？
            PORTA = 0x06;        //PORTA 初期化
                }
            else{
                PORTA = LATA*2+1;    //PORTA の『L』ポートを 1 ビット分シフト
                }
             }
          int times;                 //回数を数える変数を設定
          PORTA = 0;                 //PORTA は全て常に『L』

            for(a=0;a<3;a++){        //3 回繰り返し
                 PORTB = 0x01;       //PORTB の bit0 は最初『H』
                  for(times=0;times<8;times++)      //1 周するまで数える
                  {
                   __delay_ms(50);  //50ms 待つ
                 PORTB <<= 1;                //PORTB を 1 ビット左にシフト
                 PORTB += 1;                 //PORTB を 1 ビット左にシフト
                  }
                  PORTB = 0;                 //POTRB を全て『L』(全消灯)
                  __delay_ms(100); //100ms 待つ
          }
      }
}
```

無限ループの中に、上から順番にプログラムを書き込んだものがこのプログラムです。
ただし、このプログラムでは 1 段ずつクルクル点灯させるプログラム部分は、無限ループを 3 回繰り返す命令に変えただけでは、変数初期化のタイミングの違いで 1 回しか繰り返しませんので、ちょっと変更が必要です。

なんだかどこからどこまでが 1 つの点灯パターンのプログラムなのか非常に解読しにくいものになってしまいました。そこで、1 つの点灯パターンを『関数』とし、メイン関数内でその点灯パターン関数を呼び出すようにしてみましょう。
途中にコメントを記入し、その記述や関数の役割などをメモしておくと、他の人が見てもわかりやすいものになります。

```
/*コンフィグレーションビット設定までは省略しています。(実際は要記述)*/

//関数のプロトタイプ宣言
void initial(void);                              //初期設定関数
void ptn_0(unsigned int t);                      //全消灯の関数　t は消灯待機時間の仮引数
void ptn_1(unsigned int t);                      //全点灯の関数　t は点灯時間の仮引数
void ptn_2(unsigned char a,unsigned int t);  //全点滅関数　a は繰り返し回数、t は点灯時間の仮引数
void ptn_3(unsigned char a,unsigned int t);  //1 段ずつ点灯関数　　〃
void ptn_4(unsigned char a,unsigned int t);  //縦 1 列ずつ点灯関数　　〃
void ptn_5(unsigned char a,unsigned int t);  //1 段 1 コずつ点灯関数　　〃
void ptn_6(unsigned char a,unsigned int t);  //縦 1 列増加点灯関数　　〃
void time_ms(unsigned int t);                    //delay 関数に変数使用可にする関数。t は ms 時間の仮引数
//-----------------------------------------------

//ここからメイン関数-------------------------------
void main(void) {
```

```
    initial();                  //初期設定関数呼び出し

  while(1){                     //無限ループ

    ptn_1(1000);                //全点灯関数を点灯時間 1000ms で呼び出し
    ptn_0(500);                 //全消灯関数を消灯時間 500ms で呼び出し
    ptn_2(3,500);               //全点滅の数を 3 回繰り返し、点灯間隔 100ms で呼び出し
    ptn_0(200);                 //全消灯関数を消灯時間 100ms で呼び出し
    ptn_3(5,100);               //1 段ずつ点灯関数を 5 回繰り返し、点灯間隔 100ms で呼び出し
    ptn_0(200);                 //全消灯関数を消灯時間 100ms で呼び出し
    ptn_4(4,80);                //縦 1 列ずつ点灯関数を 4 回繰り返し、点灯間隔 200ms で呼び出し
    ptn_0(200);                 //全消灯関数を消灯時間 100ms で呼び出し
    ptn_5(3,50);                //1 段 1 コずつ点灯関数を 3 回繰り返し、点灯間隔 50ms で呼び出し
    ptn_0(200);                 //全消灯関数を消灯時間 100ms で呼び出し
    ptn_6(4,100);               //縦 1 列増加点灯関数を 8 回繰り返し、点灯間隔 100ms で呼び出し
    ptn_0(200);                 //全消灯関数を消灯時間 100ms で呼び出し

    }
  }
//メイン関数はここまで-----------------------------

//初期設定関数-------------------------------------
void initial(void){
    OSCCON = 0b00111000;        //システムクロックはデフォルトの 500kHz に
    OSCTUNE = 0;                //クロックチューニングはなし
    PORTA = 0;                  //ポート A を全て『L』に
    LATA = 0;                   //ポート A のデータラッチレジスタをポート A と同じ値に
    ANSELA = 0;                 //ポート A をデジタル I/O に
    TRISA = 0;                  //ポート A は RA5 以外を出力に(RA5 は入力専用)
    nWPUEN = 0;                 //OPTION_REG の WPUEN を『L』にし、ウィークプルアップを有効に
    WPUA = 1;                   //ポート A にプルアップ設定(RA5 のみ)
    PORTB = 0;                  //ポート B を全て『L』に
    LATB = 0;                   //ポート B のデータラッチレジスタをポート A と同じ値に
    ANSELB = 0;                 //ポート B をデジタル I/O に
    TRISB = 0;                  //ポート B は全て出力に
  }
//-------------------------------------------------

//全消灯の関数-------------------------------------
void ptn_0(unsigned int t){
    PORTB = 0;                  //ポート B を全て『L』に
    time_ms(t);                 //変数化 delay 関数呼び出し
  }
//-------------------------------------------------

//全点灯の関数-------------------------------------
void ptn_1(unsigned int t){
    PORTB = 0xFF;               //ポート B を全て『H』に
    PORTA = 0x00;               //ポート A を全て『L』に
    time_ms(t);                 //変数化 delay 関数呼び出し
  }
//-------------------------------------------------

//全点滅の関数-------------------------------------
void ptn_2(unsigned char a,unsigned int t){
    char b;                     //ローカル変数 b を設定
    for(b=0;b<a;b++){           //b が繰り返し回数 a より小さい間繰り返す
    PORTB = 0xFF;               //ポート B を全て『H』に
    time_ms(t);                 //変数化 delay 関数呼び出し
    PORTA = 0x00;               //ポート A を全て『L』に
    PORTB = 0x00;               //ポート B を全て『L』に
    time_ms(t);                 //変数化 delay 関数呼び出し
    }
  }
//-------------------------------------------------

//1 段ずつ点灯関数---------------------------------
void ptn_3(unsigned char a,unsigned int t){
    char b;                     //ローカル変数 b を設定
      for(b=0;b<a;b++){         //b が繰り返し回数 a より小さい間繰り返す
      PORTB = 0b11111111;       //16 進数だと 0xff
      PORTA = 0b10000110;       //16 進数だと 0x86
```

```
      time_ms(t);              //変数化 delay 関数呼び出し
      PORTA = 0b10000101;      //16 進数だと 0x85
      time_ms(t);              //変数化 delay 関数呼び出し
      PORTA = 0b10000011;      //16 進数だと 0x83
      time_ms(t);              //変数化 delay 関数呼び出し
      }
  }
//-------------------------------------------------

//縦 1 列ずつ点灯関数---------------------------------
void ptn_4(unsigned char a,unsigned int t){
    char b;                    //ローカル変数 b を設定
    PORTA = 0;                 //PORTA は全て常に『L』
      for(b=0;b<a;b++){        //b が繰り返し回数 a より小さい間繰り返す
      PORTB = 0b00000001;      //PORTB の bit0 は最初『H』
      time_ms(t);              //変数化 delay 関数呼び出し
        do{
        PORTB <<= 1;           //16 進数だと 0x86
        time_ms(t);            //変数化 delay 関数呼び出し
        }
        while(RB7==0);         //PORTB の bit7 までシフトして 1 になればこのループを抜ける
      }
  }
//-------------------------------------------------

//1 段 1 コずつ点灯関数-------------------------------
void ptn_5(unsigned char a,unsigned int t){
  char b;                      //ローカル変数 b を設定
  char c;                      //ローカル変数 c を設定
    PORTA = 0x06;              //PORTA 初期化。下位 3 ビットを 110 に
      for(b=0;b<a;b++){        //b が繰り返し回数 a より小さい間繰り返す
      c=0;                     //c を初期化

        while(c<3){            //3 回繰り返し
        PORTB = 0x01;          //PORTB は bit0 だけ『H』
        time_ms(t);            //変数化 delay 関数呼び出し
          do{
          PORTB <<= 1;         //1 周するまで PORTB を 1 ビットシフト
          time_ms(t);          //変数化 delay 関数呼び出し
          }

          while(RB7==0);       //RB7 までシフトならば、PORT シフト処理へ

          if(RA2==0){          //RA2 までシフトならば
          PORTA = 0x06;        //PORTA 初期化
          }
          else{
          PORTA = LATA*2+1;    //3 段目点灯まで、PORTA の『L』ポートを 1 ビット分シフト
          }
        c++;                   //c をインクリメント
        }
      }
  }
//-------------------------------------------------

//縦 1 列増加点灯関数---------------------------------
void ptn_6(unsigned char a,unsigned int t){
  char b;                      //ローカル変数 b を設定
  char c;                      //回数を数える変数を設定
  PORTA = 0;                   //PORTA は全て常に『L』
    for(b=0;b<a;b++){          //b が繰り返し回数 a より小さい間繰り返す
    PORTB = 0x01;              //PORTB の bit0 は最初『H』
      for(c=0;c<8;c++)         //1 周するまで数える
      {
      time_ms(t);              //変数化 delay 関数呼び出し
      PORTB <<= 1;             //PORTB を 1 ビット左にシフト
      PORTB += 1;              //PORTB 最下位ビットに 1 を足す
      }
    PORTB = 0;                 //POTRB を全て『L』(全消灯)
    time_ms(t);                //変数化 delay 関数呼び出し
    }
  }
```

```
//-----------------------------------------------

//変数化 delay 関数---------------------------------
void time_ms(unsigned int t){
  unsigned int d;              //ローカル変数 d を設定
    for(d=0;d<t;d++){          //d が待ち時間 h 変数 t より小さい間
    __delay_ms(1);             //1ms の待ち時間を繰り返す
    }
  }
//-----------------------------------------------
```

```
//関数のプロトタイプ宣言
void initial(void);
void ptn_0(unsigned int t);
void ptn_1(unsigned int t);
void ptn_2(unsigned char a,unsigned int t);
void ptn_3(unsigned char a,unsigned int t);
void ptn_4(unsigned char a,unsigned int t);
void ptn_5(unsigned char a,unsigned int t);
void ptn_6(unsigned char a,unsigned int t);
void time_ms(unsigned int t);
```

まず『宣言部』で、『ヘッダーファイルの#include』や『コンフィグレーションビットの設定』、このプログラム全体で使う『グローバル変数』、『関数のプロトタイプ宣言』を行います。
関数のプロトタイプ宣言の書き方は

```
戻り値の型　関数名(引数の型 仮引数名);
```

でしたね。

```
//ここからメイン関数----------------------
void main(void) {
    initial();

  while(1){

    ptn_1(1000);
    ptn_0(500);
    ptn_2(3,500);
    ptn_0(200);
    ptn_3(5,100);
    ptn_0(200);
    ptn_4(4,80);
    ptn_0(200);
    ptn_5(3,50);
    ptn_0(200);
    ptn_6(4,100);
    ptn_0(200);

    }
  }
//メイン関数はここまで----------------------
```

メイン関数はこのプログラムが実行する部分でしたね。この中の記述を順次実行していきます。
まず最初に初期化のための『initial()』関数を呼び出し、実行します。
次に while(1)の無限ループで、点灯パターンを順番に呼び出します。
関数の『void』は省略できるので付けていません。
また、各関数の(　)中の数字は、関数の仮引数に入れる実引数です。
例えば、全点滅を実行する関数を呼び出す際、繰り返し回数を決める仮引数『a』に『3』を、点滅間隔時間を決める仮引数『t』に『500』を入れ、『ptn_2』関数を実行することになります。

```
ptn_2(3,500);
```

```
//全点滅の関数-----------------------------
void ptn_2(unsigned char a,unsigned int t){
    char b;
    for(b=0;b<a;b++){
    PORTB = 0xFF;
    time_ms(t);
    PORTA = 0x00;
    PORTB = 0x00;
    time_ms(t);
    }
  }
//----------------------------------------
```

3
500

今回のメイン関数では、各点灯パターンの間に全ての LED を消灯する関数を入れています。
そうすることで各点灯パターン間の区切りがはっきりし、全てを順番に実行するとき、点灯にメリハリができます。

```
//変数化 delay 関数---------------------------
void time_ms(unsigned int t){
  unsigned int d;
    for(d=0;d<t;d++){
    __delay_ms(1);
    }
  }
//----------------------------------------
```

60 ページで『__delay_us()』や『__delay_ms()』には変数を使用することができないと説明しましたが、左のように『関数』にすることで変数も使用できるようになります。

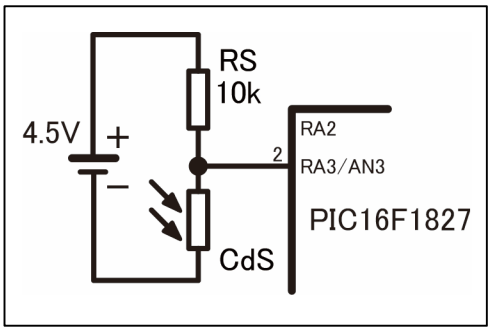
この関数では、『t』に入れる実引数の回数 1ms を繰り返すことで『t』と同等の時間を作り出します。この『t』を変数にすることで、状況によって時間を変えるというようなプログラムも可能になります。

```
for(t=100,t>0,t--)
 {
  time_ms(t);
 }
```

左記のプログラムは、だんだんと時間が短くなっていくプログラムです。

明るさセンサーを使う

PICA Tower には周囲の明るさをキャッチする明るさセンサーが標準で搭載されています。
この明るさセンサーは『CdS(シーディーエス)』という電子部品で、CdS に当たる光が明るくなると CdS の抵抗値が低くなり、暗くなると高くなるという性質があり、これを利用して周囲の明るさをキャッチします。



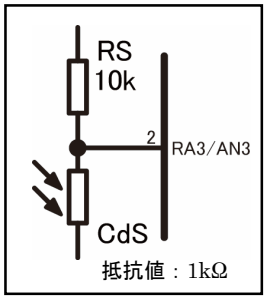
左は PICA Twer の CdS 回路部分だけを抜き出したものです。
抵抗と CdS が直列につながり、その両端が電源に、中間がマイコンの RA3 につながっています。
RA3 につながっている端子の電圧は CdS の抵抗値により変化します。これは、電源電圧の 4.5V が固定抵抗 RS と CdS の抵抗値の比により分割されるからです。このことを『分圧』といいます。

これはオームの法則で求めることができます。
例えば、CdS の抵抗値を XkΩとすると、全体の合成抵抗は(10＋X)kΩ ですね。計算しやすいように kΩ ははずして考えましょう。

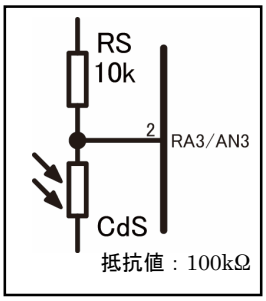
この合成抵抗に流れる電流は、電流＝電圧÷抵抗ですから、$4.5/10+X$ です。

厳密にはマイコンにも電流が流れるのですが、その電流はこの抵抗に流れる電流に比べとても小さいため、無視して計算することができます。

この時 CdS に加わる電圧は、電圧＝電流×抵抗ですから、$\left(4.5/10+X\right)\times X = 4.5(X/10+X)$となり、電源電圧を固定抵抗と CdS の合成抵抗で割ったものに CdS の抵抗値をかけたもの、つまり CdS と合成抵抗の比になります。



例えば CdS に光があたっている時(周囲が明るい時)の CdS の抵抗値が 1kΩ だったとすると・・・
CdS に加わる電圧＝RA3 に加わる電圧ですから
RA3 の電圧$= 4.5(1/10+1) \fallingdotseq 0.4$
で、約 0.4V です。



また CdS に光があたっていない時(周囲が暗い時)の CdS の抵抗値が 100kΩ だったとすると・・・

CdS に加わる電圧＝RA3 に加わる電圧ですから

RA3 の電圧= $4.5\left({}^{100}/_{10+100}\right) \fallingdotseq 4.1$

で、約 4.1V になります。

このように、CdS の抵抗の変化を電圧の変化に変換することで、その電圧の値をマイコンで読み取ることができます。これを『A/D コンバータ』といい、これを利用すると読み取った電圧の値に応じて LED の点灯パターンを変えるというプログラムを作ることができるのです。

・A/D コンバータの仕組み

PIC16F1827 は A/D コンバータに使用できるポートを 12 コ持っていて、それぞれの分解能は 10 ビットです。

『分解能が 10 ビット』とは、基準になる電圧を『$2^{10} = 1024$』分割した精度で読み取ることができるという意味です。

例えば、基準電圧が 5V の場合、$5 \div 1024 = 0.00488\cdots$で、約 4.9mV 刻みで読み取ることができます。

読み取った値が 0V だった時、A/D コンバータの値は 16 進数で 0x000、5V の時が 0x3FF となります。

読み取った電圧値を 16 進数の値に変換する方法は次のとおりです。

$1023 \times \left({}^{\textit{読み取った電圧値}}/_{\textit{基準電圧値}}\right)$の答えで、小数点以下を切り捨て、その値を 16 進数に変換することで求めることができます。

例えば基準電圧が 5V で読み取った値が 3.2V の場合、654.72 なので 654 を 16 進数に変換すると 0x28E となります。

・A/D コンバータを使うためには

マイコンの A/D コンバータ機能を使用するためには、マイコンのポートの初期設定をしましょう。

```
OSCCON = 0x38;          //システムクロックはデフォルトの 500kHz に
PORTA = 0;              //ポート A を全て『L』に
LATA = 0;               //ポート A のデータラッチレジスタをポート A と同じ値に
ANSELA = 0x08;          //RA3 を AN3(アナログポート)に、他をデジタル I/O に
TRISA = 0x08;           //ポート A は AN3 入力、それ以外を出力に(RA5 は入力専用)
WPUA = 1;               //ポート A にプルアップ設定(RA5 のみ)
nWPUEN = 0;             //OPTION_REG の WPUEN を『L』にし、ウィークプルアップを有効に
PORTB = 0;              //ポート B を全て『L』に
LATB = 0;               //ポート B のデータラッチレジスタをポート A と同じ値に
ANSELB = 0;             //ポート B をデジタル I/O に
TRISB = 0;              //ポート B は全て出力に
```

これまでのプログラムは、全てのポートを『デジタル I/O』として使用していましたが、今回は RA3 をアナログ入力に設定します。ちなみに RA3 のように『Rxn』(x はポート、n はビット数)と書いた場合、そのポートはデジタルポート、AN3 のように『Axn』と書いた場合はアナログポートであることを表します。

マイコンの端子は同じ端子でも、その役割によって名前を変えて呼びます。

上の初期設定では、『ANSELA』レジスタでポート A の RA3 をアナログポートに、『TRISA』レジスタで AN3 を入力に設定しています。

ポート B はデジタル I/O のままなので変更していません。

ANSELA レジスタと TRISA レジスタ

REGISTER 12-7: ANSELA: PORTA ANALOG SELECT REGISTER

U-0	U-0	U-0	R/W-1/1	R/W-1/1	R/W-1/1	R/W-1/1	R/W-1/1
—	—	—	ANSA4	ANSA3	ANSA2	ANSA1	ANSA0
bit 7							bit 0

bit 7-5	**Unimplemented:** Read as '0'
bit 4-0	**ANSA<4:0>**: Analog Select between Analog or Digital Function on pins RA<4:0>, respectively 0 = Digital I/O. Pin is assigned to port or digital special function. 1 = Analog input. Pin is assigned as analog input(1). Digital input buffer disabled.

REGISTER 12-4: TRISA: PORTA TRI-STATE REGISTER

R/W-1/1	R/W-1/1	R-1/1	R/W-1/1	R/W-1/1	R/W-1/1	R/W-1/1	R/W-1/1
TRISA7	TRISA6	TRISA5	TRISA4	TRISA3	TRISA2	TRISA1	TRISA0
bit 7							bit 0

bit 7-6	**TRISA<7:6>:** PORTA Tri-State Control bit 1 = PORTA pin configured as an input (tri-stated) 0 = PORTA pin configured as an output
bit 5	**TRISA5:** RA5 Port Tri-State Control bit This bit is always '1' as RA5 is an input only
bit 4-0	**TRISA<4:0>:** PORTA Tri-State Control bit 1 = PORTA pin configured as an input (tri-stated) 0 = PORTA pin configured as an output

『ANSELA』レジスタと『TRISA』レジスタのおさらいです。

『ANSELA』デジスタは各ビットを『1』に設定するとアナログ入力に、『0』に設定するとデジタル I/O に、

『TRISA』レジスタは各ビットを『1』に設定すると入力に、『0』に設定すると出力になります。

ADCON1 レジスタ

A/D コンバータを使用するためのポートの設定が終わったら、次はその A/D コンバータの処理方法などを設定します。

```
ADCON1 = 0x30;          //データ格納左寄せ、変換クロック FRC、基準電圧 Vdd
ADCON0 = 0x38;          //AN3 を選択、ADC を有効
```

AD コンバータの処理方法は、この 2 つのレジスタで設定します。

REGISTER 16-2: ADCON1: A/D CONTROL REGISTER 1

R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	U-0	R/W-0/0	R/W-0/0	R/W-0/0
ADFM	ADCS<2:0>			—	ADNREF	ADPREF<1:0>	
bit 7							bit 0

bit 7	**ADFM:** A/D Result Format Select bit 1 = Right justified. Six Most Significant bits of ADRESH are set to '0' when the conversion result is loaded. 0 = Left justified. Six Least Significant bits of ADRESL are set to '0' when the conversion result is loaded.
bit 6-4	**ADCS<2:0>:** A/D Conversion Clock Select bits 000 = Fosc/2 001 = Fosc/8 010 = Fosc/32 011 = FRC (clock supplied from a dedicated RC oscillator) 100 = Fosc/4 101 = Fosc/16 110 = Fosc/64 111 = FRC (clock supplied from a dedicated RC oscillator)
bit 3	**Unimplemented:** Read as '0'
bit 2	ADNREF: A/D Negative Voltage Reference Configuration bit 0 = VREF- is connected to VSS 1 = VREF- is connected to external VREF- pin(1)
bit 1-0	**ADPREF<1:0>:** A/D Positive Voltage Reference Configuration bits 00 = VREF+ is connected to VDD 01 = Reserved 10 = VREF+ is connected to external VREF+ pin(1) 11 = VREF+ is connected to internal Fixed Voltage Reference (FVR) module(1)

『ADCON1』レジスタでは、A/D 変換後のデータの『格納方法』(、A/D 変換を行う際に使用される『A/D 変換クロック』、使用する『基準電圧』の各設定を行います。

ADRESH レジスタと ADRESL レジスタ

A/D 変換後のデータは『ADRESH』『ADRESL』の各レジスタに格納されます。

『ADRESH』『ADRESL』レジスタはそれぞれ 8 ビットです。A/D コンバータで得られる値は 10 ビットなので、この 2 つのレジスタに分けて格納されます。

そのとき、『ADFM』ビットが『0』の時と『1』の時では格納のされ方が異なります。

ADFM=1 の場合 右に寄せて格納																
ADRESH レジスタ								ADRESL レジスタ								
0	0	0	0	0	0	ADRESH		レジスタの下位 2 ビットと				ADRESL		レジスタに		格納
bit7							bit0	bit7								bit0

ADFM=0 の場合 左に寄せて格納																
ADRESH レジスタ								ADRESL レジスタ								
ADRESH		レジスタと		ADRESL		レジスタの上位 2		ビットに		格納	0	0	0	0	0	0
bit7							bit0	bit7								bit0

A/D 変換後の値を 10 ビット分全て使用する場合には右寄せがよく使われ、分解能が 8 ビットでよい場合には左寄せにして『ADRESH』レジスタの値だけ使用します。分解能が 8 ビットということは基準電圧を 256 分割した精度になり、約 19.5mV 刻みで読み取ることになります。

その場合、変換結果を表す値は、$255 \times \left({}^{\textit{読み取った電圧値}} / _{\textit{基準電圧値}} \right)$となります。16 進数にするには、この答えの小数点以下を切り捨て、16 進数に変換して求めます。

例えば基準電圧が 5V で読み取った値が 3.2V の場合、163.2 なので 163 を 16 進数に変換すると 0xA3 となります。

『ADCS』ビットは『A/D 変換クロック』に何を使用するか決めるビットです。『A/D 変換クロック』は A/D 変換する時に使用されるクロックで、A/D コンバータには A/D 変換専用の発振器が内蔵されていま

す。前ページで設定した FRC は、この内蔵発振器を使用する設定です。
内蔵発振器の他に、マイコンを動作させているクロックを使用することも可能です。変換クロックの周期を TAD と表します。A/D 変換を行うためには 11.5 TAD の時間が必要になります。設定する A/D 変換クロックにより TAD の時間が異なり、速く変換したいような用途では TAD が短い A/D 変換クロックに設定する必要があります。

TABLE 16-1: ADC CLOCK PERIOD (TAD) Vs. DEVICE OPERATING FREQUENCIES

ADC Clock Period (TAD)		Device Frequency (FOSC)					
ADC Clock Source	ADCS<2:0>	32 MHz	20 MHz	16 MHz	8 MHz	4 MHz	1 MHz
FOSC/2	000	62.5ns(2)	100 ns(2)	125 ns(2)	250 ns(2)	500 ns(2)	2.0 µs
FOSC/4	100	125 ns(2)	200 ns(2)	250 ns(2)	500 ns(2)	1.0 µs	4.0 µs
FOSC/8	001	0.5 µs(2)	400 ns(2)	0.5 µs(2)	1.0 µs	2.0 µs	8.0 µs(3)
FOSC/16	101	800 ns	800 ns	1.0 µs	2.0 µs	4.0 µs	16.0 µs(3)
FOSC/32	010	1.0 µs	1.6 µs	2.0 µs	4.0 µs	8.0 µs(3)	32.0 µs(3)
FOSC/64	110	2.0 µs	3.2 µs	4.0 µs	8.0 µs(3)	16.0 µs(3)	64.0 µs(3)
FRC	x11	1.0-6.0 µs(1,4)	1.0-6.0 µs(1,4)	1.0-6.0 µs(1,4)	1.0-6.0 µs(1,4)	1.0-6.0 µs(1,4)	1.0-6.0 µs(1,4)

使用する A/D 変換クロックによる TAD の違い

『ADNREF』と『ADPREF』は基準電圧の設定を行います。
『基準電圧』は A/D コンバータの変換結果が 0x000、0x3FF になる場合の電圧の設定です。
74 ページの設定では、0x000 となる電圧を Vss=0V に、0x3FF になる電圧を VDD=電源電圧(PICA Tower を乾電池で使用した場合は約 4.5V)にしています。
マイコンの VREF−端子、または VREF+端子に外部から電圧を加え、それを基準にすることもできます。

ADCON0 レジスタ

REGISTER 16-1: ADCON0: A/D CONTROL REGISTER 0

U-0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0
—	CHS<4:0>					GO/DONE	ADON
bit 7							bit 0

bit 7 **Unimplemented:** Read as '0'
bit 6-2 **CHS<4:0>:** Analog Channel Select bits
00000 = AN0
00001 = AN1
00010 = AN2
00011 = AN3
00100 = AN4
00101 = AN5
00110 = AN6
00111 = AN7
01000 = AN8
01001 = AN9
01010 = AN10
01011 = AN11
01100 = Reserved. No channel connected.
•
•
•
11101 = Temperature Indicator(3)
11110 = DAC output(1)
11111 = FVR (Fixed Voltage Reference) Buffer 1 Output(2)
bit 1 **GO/DONE:** A/D Conversion Status bit
1 = A/D conversion cycle in progress. Setting this bit starts an A/D conversion cycle.
This bit is automatically cleared by hardware when the A/D conversion has completed.
0 = A/D conversion completed/not in progress
bit 0 **ADON:** ADC Enable bit
1 = ADC is enabled
0 = ADC is disabled and consumes no operating current

『ADCON0』レジスタは動作させる A/D コンバータの選択、マイコン中の A/D コンバータを行う『ADC Module』を有効にするかどうか、A/D 変換の開始と完了のモニターを設定します。

『CHS』ビットで動作させる A/D コンバータを設定します。今回は AN3 を動作させますので『00011』に

なります。
最下位ビットの『ADON』は、マイコンの中にある A/D 変換を行う『ADC Module』を有効にするかどうかを設定するビットで、A/D 変換をするためにはここを『1』にしなければなりません。
そして『GO/$\overline{\text{DONE}}$』ビットですが、ここを『1』にすると A/D 変換が開始されます。変換中は『1』のままで、変換が完了すると自動的に『0』になります。ですから、このビットを『1』に設定し変換スタートさせた後、『0』になるまで待って、『0』になったら A/D 変換のデータが『ADRESH』レジスタに格納されているのです。

では、A/D コンバータを動作させるプログラムを書いてみましょう。

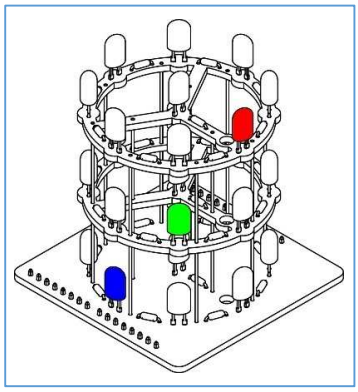
```
unsigned char ADVALUE;          //A/D 変換値用変数

void main (void){


    OSCCON = 0x38;              //システムクロックはデフォルトの 500kHz に
    PORTA = 0;                  //ポート A を全て『L』に
    LATA = 0;                   //ポート A のデータラッチレジスタをポート A と同じ値に
    ANSELA = 0x08;              //RA3 を AN3(アナログポート)に、他をデジタル I/O に
    TRISA = 0x08;               //ポート A は AN3 入力、それ以外を出力に(RA5 は入力専用)
    WPUA = 1;                   //ポート A にプルアップ設定(RA5 のみ)
    nWPUEN = 0;                 //OPTION_REG の WPUEN を『L』にし、ウィークプルアップを有効に
    PORTB = 0;                  //ポート B を全て『L』に
    LATB = 0;                   //ポート B のデータラッチレジスタをポート A と同じ値に
    ANSELB = 0;                 //ポート B をデジタル I/O に
    TRISB = 0;                  //ポート B は全て出力に


  while(1){                     //無限ループ

    ADCON1 = 0x30;              //データ格納左寄せ、変換クロック FRC、基準電圧 Vdd
    ADCON0 = 0x0D;              //AN3 を選択、ADC を有効

    __delay_us(10);             //ホールドキャパシタなど、内部回路のセッティング待ち時間

    GO_nDONE = 1;               //A/D 変換スタート

    while(GO_nDONE){}           //GO/DONE が『0』になるまで待つ

    ADVALUE = ADRESH;           //AERESH レジスタの値を ADVALE に代入


    if(ADVALUE>0xB2){           //A/D 値が 0x7F(電圧値 1/2Vdd)より大きい(周囲が暗い)なら

        PORTB = 0xFF;           //LED 全点灯
        RA7 = 1;
        RA0 = RA = RA2 = 0;
      }

    else{                       //明るいなら

        PORTB = 0x00;           //LED 全消灯
        PORTA = 0;
        }
  }
}
```

このプログラムは、明るいところでは LED は消灯していますが、周囲が暗くなると全ての LED が点灯します。if 文の条件で 0x7F の値を変更すると、明るさセンサーの感度を変更することができます。
A/D 変換を行うときの注意として、ADC を有効にした後、ちょっとの時間を待ってから変換をスタートさせなければならないことです。A/D コンバータにはコンデンサやスイッチ回路などが内蔵されており、これらがセッティングされる時間を待ってからスタートしなければなりません。この時間のことを『アクイ

ジション・タイム』といい、PIC16F1827 では約 4.5 マイクロ秒です。(データシート 149 ページ参照)
この時間は周囲の温度などにより少々変化しますので、余裕を見て 10 マイクロ秒としました。
また、A/D 変換後の値を『ADVALUE』という変数に代入した後にその値を条件値として if 文を作成しています。直接『ADRESH』レジスタの値を読みに行ってもよいのですが、A/D 変換を複数個使用するような場合には、それぞれの変換データは全て『ADRESH』『ADRESL』レジスタに格納されますので、どの A/D 変換の値が読み込まれるのかわかりづらくなります。そのため、それぞれの A/D 変換用に変数を作成し、それぞれの A/D 変換の値を入れ、それを使用するようにしましょう。

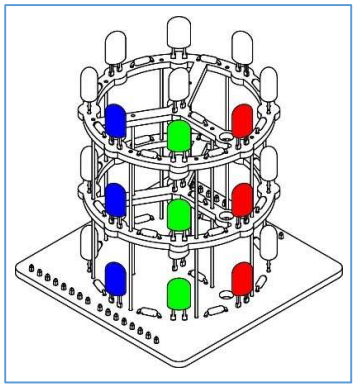
スタティック点灯とダイナミック点灯

いきなり問題です。下の図の場所の 3 つの LED を点灯させてください。



これまでどおり、ポート B の各ビットを『H』にし、ポート A は 3 段とも『L』にするとよさそうですが・・・

```
PORTB = 0b11000001;
PORTA = 0b00000000;
```



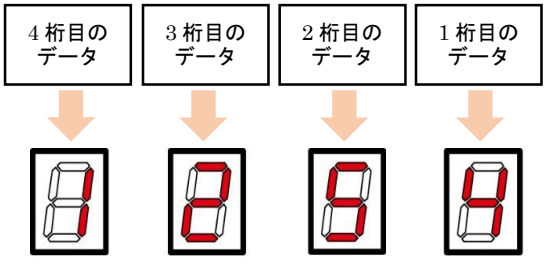
しかし、その方法では左のように点灯してしまいます。
縦 1 列のアノード側が共通、各段のカソード側が共通になっているため、このようになるのは当然ですね。

では、上の図のように光らせることはできないのでしょうか？
実はある方法を使えば可能なのです。

・スタティック点灯

スタティック点灯とは、LED を点灯させるとき、その LED に電流を流し続けて点灯させる方法です。
上の点灯の場合、RB0、RB7、RB8 をずっと『H』に、RA0、RA1、RA2 をずっと『L』にしていますので、LED にはずっと電流が流れ続けます。

ちょっと例を変えて、時計のような 4 桁の数字を表示する装置を考えてみましょう。



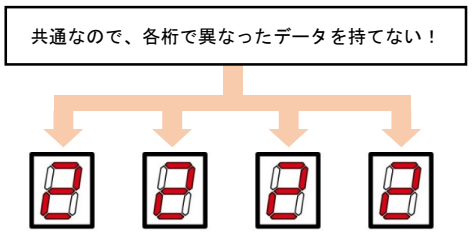
このように、スタティック点灯方式はそれぞれの桁にそれぞれの点灯データを持ち、そのデータによって各桁が独立して点灯するような方式です。

・ダイナミック点灯

スタティック点灯は、それぞれの桁ごとの点灯データで LED を光らせますので、桁数が増えるとその桁数分の回路を追加しなければなりません。

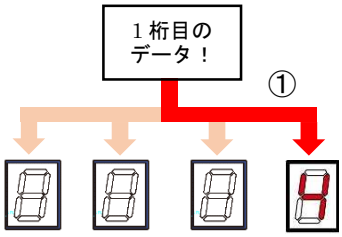
また、PICA Tower のように、1～3 段目のアノード側が共通である場合、それぞれの段ごとに異なったデータを持つことができない場合もあります。

PICA Tower のように、各桁が共通の場合の 4 桁の数字表示は下記のようになります。



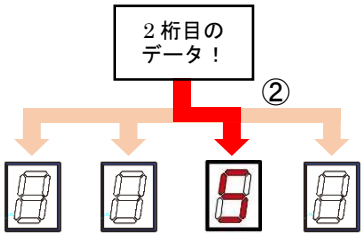
では、このような場合、どうすればそれぞれの桁で異なった表示ができるようになるのでしょう？

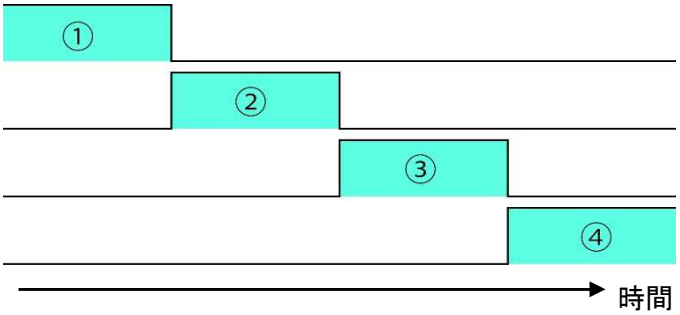
答えは、LED に流す電流をずっと流し続けないで、一定時間ごとに切り替えて電流を流すのです。

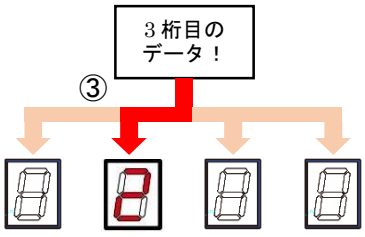


1 桁目、2 桁目・・と一定時間ごとに送るデータを切り替え、それをずっと繰り返すことで、それぞれの桁で違う表示をさせることができます。

下のイラストは、それぞれの桁を表示させるタイミングを表したチャートです。



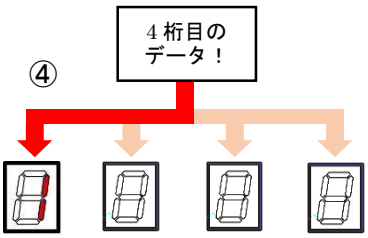




60 ページの LED の点滅で、点滅周期が早いと、人間の目では点灯したままに見えることをやってみましたね。ちょうどそれと同じように、速いスピードで切り替えることで、すべての桁が点灯しているように見えるのです。

このように点灯タイミングを一定時間ごとに切り替えてデータを送り表示させる方法を『ダイナミック点灯』といいます。

ダイナミック点灯はすべての桁に対し、データを送る回路が一つで OK なので、回路を比較的簡素にすることができます。



では、最初の問題のプログラムを作成してみましょう。

```
void main(void){

OSCCON = 0b00111000;          //システムクロックはデフォルトの 500kHz に
OSCTUNE = 0;                  //クロックチューニングはなし
PORTA = 0;                    //ポート A を全て『L』に
LATA = 0;                     //ポート A のデータラッチレジスタをポート A と同じに
ANSELA = 0;                   //ポート A をデジタル I/O に
TRISEA = 0;                   //ポート A は RA5 以外を出力に(RA5 は入力専用)
nWPUEN = 0;                   //OPTION_REG の WPUEN を『L』にし、ウィークプルアップを有効に
WPUA = 1;                     //ポート A にプルアップを設定(RA5 のみ)
PORTB = 0;                    //ポート B を全て『L』に
LATB = 0;                     //ポート B のデータラッチレジスタをポート A と同じに
ANSELB = 0;                   //ポート B をデジタル I/O に
TRISB = 0;                    //ポート B は全て出力に

while(1){

//1 段め
       PORTB = 0x01;          //ポート B の bit0 のみ『H』に
       PORTA = 0x06;          //ポート A の bit0～bit2 のうち、bit0 だけを『L』に
       __delay_ms(3);         //点灯周期を 100Hz(1 周期約 10ms)なので、1 段あたり約 3ms
       PORTA = 0x07;          //他ポート干渉防止のため、一旦消灯

//2 段め
       PORTB = 0x80;          //ポート B の bit7 のみ『H』に
       PORTA = 0x05;          //ポート A の bit0～bit2 のうち、bit1 だけを『L』に
       __delay_ms(3);         //点灯周期を 100Hz(1 周期約 10ms)なので、1 段あたり約 3ms
       PORTA = 0x07;          //他ポート干渉防止のため、一旦消灯

//3 段め
       PORTB = 0x40;          //ポート B の bit6 のみ『H』に
       PORTA = 0x03;          //ポート A の bit0～bit2 のうち、bit2 だけを『L』に
       __delay_ms(3);         //点灯周期を 100Hz(1 周期約 10ms)なので、1 段あたり約 3ms
       PORTA = 0x07;          //他ポート干渉防止のため、一旦消灯
       }
}
```

ダイナミック点灯で、各段を切り替える周期(1 周する時間)は、20ms(周波数で 50Hz)より短くするとよいでしょう。これより長くなると、切り替えタイミングが目で見えるようになり、LED がちらついて見えてしまいます。今回は 1 周を約 10ms(100Hz)で設定しました。3 段を切り替えるので、1 段あたり約 3.3ms です。少数を切り捨て 3ms の待ち時間としています。各段の表示を切り替える前にポート A の bit0～bit2 を『H』にし全 LED を消灯しています。この消灯をしない場合、他の LED がうっすらと点灯してしまう場合があります。

・内蔵タイマーと割り込みを使う

PIC マイコンにはいくつかのタイマー機能が内蔵されており、それを使用することで一定の時間を発生させたり、回数を数えたりすることができます。また、このタイマーを利用して一定時間や設定した回数をカウントしたらイベントを発生させるような機能も備わっています。この機能を使用して、段の切り替えをしてみましょう。

PIC16F1827 には『タイマー0』という 8 ビットのタイマーを持っており、一定間隔で 1 ずつインクリメントされます。そのタイマーを使うと、タイマーの値がいっぱいになった『0xFF』から＋1 されて『オーバーフロー』した時に『割り込み』を発生させることができます。

『割り込み』とは、あるプログラムを実行中に他のプログラムの実行を割りこませることができる機能です。

例えばプログラム A を実行中に何かをきっかけにしてプログラム B を実行させるということができます。

今回は、メイン関数のプログラムを実行中に、タイマー0 がオーバーフローすることをきっかけに、割り込みプログラム(段の切り替え)を行おうというわけです。

『割り込み』を発生させる『きっかけ』は色々設定することができます。

例えば、

・割り込み専用端子の状態が変化した時

・内蔵タイマーの動作をきっかけに

・A/D 変換終了時　　などなど・・・

色々な割り込みが発生可能です。詳しくは PIC16F1827 データシートの 81 ページをご覧ください。

割り込みが発生した時に実行するプログラムはメイン関数とは別に『interrupt』という文字を付けた関数を作ります。割り込みが発生した時は、必ずこの関数を実行します。

```
//宣言部
#include  ヘッダーファイル
#progma  コンフィグレーションビットなど

//メイン関数
void main(void){
初期設定
実行処理
}

//割り込み処理
void interrupt func(void){
割り込み時の処理
}
```

では、実際のプログラムを書いてみましょう。

```
/*コンフィグレーションビット設定までは省略しています。(実際は要記述)*/

//関数プロトタイプ宣言
void init (void);                                   //初期設定関数
void time_ms(unsigned int t);                       //delay 関数変数化
void motion0(unsigned char t, unsigned(char  x );   //点灯パターン発生関数

//グローバル変数宣言
char LED_around[3];                                 //LED データ格納レジスタ
char LED_center[3];                                 //中心 LED データ格納レジスタ
char LED_flag = 0;

//メイン関数-----------------------------------------
main(){
        init();                             //初期設定
        while(1){
                unsigned char y;

                for(y=0;y<8;y++){           //8 回繰り返しで 1 周期
                        motion0(75,y);      //スピ～ド，繰り返し回数とビットシフト数の同期
                }
            }
        }
//-------------------------------------------------
```

```
//点灯パターン発生関数------------------
 void motion0(unsigned char t,unsigned char x)
        {
          LED_around[0] = (0x80<<x) | (0x01<<(x-1)); //縦１列斜めで回転するパターン
          LED_around[1] = (0x01<<x);
          LED_around[2] = (0x02<<x) | (0x02>>(8-x));
          LED_center[0] = x%2;                          //真ん中は交互に点滅
          LEDJ:enter[1] = x%2+1;
          LEDJ:enter[2] = x%2;

          time_ms(t);
        }
//---------------------------------

//初期設定関数-----------------------
 void init (void)
          {
          05CCON = 0b00111000;          //内部クロック 500kHz
          OSCTUNE = 0;                  //クロック調整なし
          LATA = 0;                     //A ポート初期化
          ANSELA = 0;
          TRISA = 0b00100000;
          PORTB = 0x00;                 //B ポート初期化
          LATB = 0;
          ANSELB = 0;
          TRISB = 0x00;

//内部クロック(500kHz)で Timer0 を使用  プリスケーラー1/2  1 カウント 16μs
       OPTIEON_REG = 0b00000000;
       TMR0 = 0;                        //タイマー0 の初期化
       TMR0IF = 0;                      //タイマー0 割込プラグ(T0IF)を 0 にする
       TMR0IE = 1;                      //タイマー0 割込み(T0IE)を許可する
       GIE = 1;                         //全割込み処理を許可する
              }
//---------------------------------

//割り込み時関数---------------------
   void interrupt InterTimer(void)     //ダイナミック点灯用(約 250Hz)
   {
       Static char LED_Flag = 0;

          LED_Flag++;                   //何段目？用のフラグ
          if(LED_flag == 4){
                 LED_Flag = 0;
          }

      IF(TMR0IF == 1){                  //タイマー0 の割込み発生か？
                 TMR0 = 0;              //タイマー0 の初期化
                 TMR0IF = 0;            //タイマー0 割込フラグをリセット

                 PORTB = 0x00;          //一旦全消灯
                 RA = 0;
                 PORTA |= 0x07;         //RA0～RA2 に『1』をセット

                 PORTB = LED_around[LED_Flag];       //周囲の LED 点灯パターン
                 RA7 = LED_center[LED_Flag];         //真ん中の LED 点灯パターン
                 PORTA &= ~(0x01<<LED_flag);         //RA0~RA2 の対応するピンを 0 に

          }
   }
   // -----------------------------
```

```
//delay 関数変数化--------------
void time_ms(unsigned int t){

	unsigned int z;
	for(z=0;z<t;z++){
		__delay_ms(1);
	}
}
//-----------------------------
```

では、内容を解説します。まず、ヘッダーファイルやコンフィグレーション、関数のプロトタイプ、グローバル変数など、各種宣言をします。

・配列

変数の書き方が今までと違った書き方がされています。

```
char LED_around[3];
char LED_center[3];
```

これは変数を『配列』という書き方で書いたものです。

配列で書くと、同じ名前、データ型の変数に番号をつけて作ることができます。

今回の例では、『char 型』の『LED_around』という、3 つの値を格納できる変数を作成しています。

	番号		
変数名	[0]	[1]	[2]
LED_around[3]	1 個目の値を格納	2 個目の値を格納	3 個目の値を格納

この表からもわかるように、『char LED_around[3];』は『3 個の値を格納できる変数』で、それぞれは『LED_around [0]』、『LED_around [1]』、『LED_around [2]』に格納されています。

配列には初期値を書くこともできます。

```
char LED_around[3] = {10,20,30};
```

と書いた時には LED_around[0]は 10、LED_around[1]は 20、LED_around[2]は 30 になります。

配列はデータを格納する場所を指定する時は、『0』から数えることに注意しましょう。

上記の配列は格納する場所が 1 列なので『1 次元配列』ですが、『2 次元配列』や『3 次元配列』などを作成することもできます。

```
char LED_around[2][3] = {{10,20,30},
                         {40,50,60}};
```

	番号		
変数名	[0][0]	[0][1]	[0][2]
LED_around[2][3]	1 個目の値を格納	2 個目の値を格納	3 個目の値を格納
	[1][0]	[1][1]	[1][2]
	4 個目の値を格納	5 個目の値を格納	6 個目の値を格納

データを格納する場所を指定する時は、最初に『段の番号』『次に列の番号』で指定します。

この例では LED_around[0][2]は 30、LED_around[1][0]は 40、LED_around[1][2]は 60 です。

```
x = LED_around[1][1];
```

上記の場合、変数 x は 5 番目の値が代入されますので、50 になります。

初期設定の前半部分は、これまでと同じです。

後半部分に、タイマー0 のオーバーフロー割り込みの設定があります。

```
//内部クロック(500kHz)で Timer0 を使用　プリスケーラー1/2　1 カウント 16μs
    OPTION_REG = 0b00000000 ;
    TMR0 = 0;          // タイマー0 の初期化
    TMR0IF = 0;        // タイマー0 割込フラグ(T0IF)を 0 にする
    TMR0IE = 1;        // タイマー0 割込み(T0IE)を許可する
    GIE = 1;           // 全割込み処理を許可する
          }
```

OPTION_REG レジスタ

REGISTER 20-1:　OPTION_REG: OPTION REGISTER

R/W-1/1	R/W-1/1	R/W-1/1	R/W-1/1	R/W-1/1	R/W-1/1	R/W-1/1	R/W-1/1
$\overline{\text{WPUEN}}$	INTEDG	TMR0CS	TMR0SE	PSA	PS<2:0>		
bit 7							bit 0

bit 7　**$\overline{\text{WPUEN}}$:** Weak Pull-up Enable bit
1 = All weak pull-ups are disabled (except $\overline{\text{MCLR}}$, if it is enabled)
0 = Weak pull-ups are enabled by individual WPUx latch values

bit 6　**INTEDG:** Interrupt Edge Select bit
1 = Interrupt on rising edge of INT pin
0 = Interrupt on falling edge of INT pin

bit 5　**TMR0CS:** Timer0 Clock Source Select bit
1 = Transition on T0CKI pin
0 = Internal instruction cycle clock (FOSC/4)

bit 4　**TMR0SE:** Timer0 Source Edge Select bit
1 = Increment on high-to-low transition on T0CKI pin
0 = Increment on low-to-high transition on T0CKI pin

bit 3　**PSA:** Prescaler Assignment bit
1 = Prescaler is not assigned to the Timer0 module
0 = Prescaler is assigned to the Timer0 module

bit 2-0　**PS<2:0>:** Prescaler Rate Select bits

Bit Value	Timer0 Rate
000	1 : 2
001	1 : 4
010	1 : 8
011	1 : 16
100	1 : 32
101	1 : 64
110	1 : 128
111	1 : 256

タイマー0 は『OPTION_REG』レジスタで行います。

bit7 は前にも出てきた『ウイークプルアップ』の設定です。

『INTEDG』はタイマー0 には関係ないところですので『0』にしておきましょう。

『TMR0CS』はタイマー0 カウント用のクロックに何を使うかです。クロックは外部から供給することもできますが、今回はシステムクロックの周波数を 1/4 にしたものを使用しますので『0』です。

『PSA』はタイマー0 に『プリスケーラ』を使用するかどうかを決めるビットです。

プリスケーラとは使用するクロックの周波数を一定の値で割って使用する事ができる機能です。タイマー0 は 8 ビットなので、0～255 までカウントすることができます。このとき、1 カウント分の時間が長ければ、それだけ長い時間のタイマーが作れることになります。

今回は 100Hz 付近のタイマーを作りたいので、256 回カウントした時に 3ms くらいになれば OK です。

マイコンのシステムクロックが 500kHz で、タイマー0 のクロックが 500kHz÷4=125kHz(1 カウント 8 マイクロ秒)で 256 回カウントすると 8 マイクロ秒×256=約 2ms となり、約 500Hz となります。これでも良いのですが、1:2 のプリスケーラを使用し、約 4ms(250Hz)でダイナミック点灯の桁を切り替えることにします。

以上で、『OPTION_REG』は全ビット『0』に設定します。

次に、タイマー0 のカウント値が入る『TMR0』レジスタを『0』に設定し、初期化します。

INTCON レジスタ

次は、割り込み機能の設定です。

REGISTER 8-1: INTCON: INTERRUPT CONTROL REGISTER

R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	R-0/0
GIE	PEIE	TMR0IE	INTE	IOCIE	TMR0IF	INTF	IOCIF(1)
bit 7							bit 0

bit 7 **GIE:** Global Interrupt Enable bit
1 = Enables all active interrupts
0 = Disables all interrupts

bit 6 **PEIE:** Peripheral Interrupt Enable bit
1 = Enables all active peripheral interrupts
0 = Disables all peripheral interrupts

bit 5 **TMR0IE:** Timer0 Overflow Interrupt Enable bit
1 = Enables the Timer0 interrupt
0 = Disables the Timer0 interrupt

bit 4 **INTE:** INT External Interrupt Enable bit
1 = Enables the INT external interrupt
0 = Disables the INT external interrupt

bit 3 **IOCIE:** Interrupt-on-Change Enable bit
1 = Enables the interrupt-on-change
0 = Disables the interrupt-on-change

bit 2 **TMR0IF:** Timer0 Overflow Interrupt Flag bit
1 = TMR0 register has overflowed
0 = TMR0 register did not overflow

bit 1 **INTF:** INT External Interrupt Flag bit
1 = The INT external interrupt occurred
0 = The INT external interrupt did not occur

bit 0 **IOCIF:** Interrupt-on-Change Interrupt Flag bit(1)
1 = When at least one of the interrupt-on-change pins changed state
0 = None of the interrupt-on-change pins have changed state

『GIE』ビットは割り込み機能を使うとき、必ず『1』にしておく必要があります。

bit6～bit3 はどんなことをきっかけに割り込みを発生させるかの設定です。

今回はタイマー0 のオーバーフローをきっかけに発生する割り込みを使うので『TMR0IE』を『1』に、その他は『0』です。

bit2～bit0 は割り込み発生時に『1』になるフラグビットです。タイマー0 オーバーフロー割り込みでは『TMR0IF』が割り込みが発生した時に自動的に『1』に変化します。このビットを監視することで、割込みが発生したことを知ることができます。このビットは割り込み処理終了後に自動的に『0』になりませんので、プログラムで『0』にもどす処理が必要になります。リセット時はこのフラグは『0』ですが、念のため『0』を書き込んで初期化しておきます。

次は点灯パターンを発生させるプログラムです。

```
//メイン関数----------------------------------------
main() {
       init(); while(1){
               unsigned char y;

               for(y=0;y<8;y++){
                              motion0(75,y);
                }
            }
        }
//-----------------------------------------------

//点灯パターン発生関数----------------------------
void motion0(unsigned char t,unsigned char x)
      {
       LED_around[0] = (0x80<<x) | (0x01<<(x-1));
       LED_around[1] = (0x01<<x);
       LED_around[2] = (0x02<<x) | (0x02>>(8-x));
       LED_center[0] = x%2;
       LED_center[1] = x%2+1;
       LED_center[2] = x%2;

       time_ms(t);
      }
//-----------------------------------------------
```

ここはプログラムをじっくり見ると、その内容がわかると思います。

motion0 関数で、周囲の LED と真ん中の LED の点灯パターンを作っています。

周りの LED は、1 つずつシフトさせて流れるように LED を回転させていますが、メイン関数の for 文を 8 回繰り返すことで周囲の LED が 1 周することを作り出しています。for 文のカウント 1 回と LED が 1 コ隣へシフトするタイミングを合わせている点と、シフトがオーバーフローした時に、LED が正しく点灯するようにしています。

LED_around[0]が 1 段目の LED のパターン LED_around[1]が 2 段目、LED_around[2]が 3 段目のパターンです。

真ん中の LED は for 文の回数を 2 でモジュロ算をして、交互に点灯するようにしています。

次に割り込み関数の説明をします。この部分が割り込み関数です。『interrupt』がついていますね。

```
//割り込み時関数---------------------------------
void interrupt InterTimer( void )
{
        LED_flag++;
        if(LED_flag == 3){
                LED_flag = 0;
        }

        if (TMR0IF == 1) {
                TMR0 =0;
                TMR0IF = 0;

                PORTB = 0x00;
                RA7 = 0;
                PORTA |= 0x07;

                PORTB = LED_around[LED_flag];
                RA7 = LED_center[LED_flag];
                PORTA &= ~(0x01<<LED_flag);

        }
}
//-----------------------------------------------
```

まず、変数『LED_flag』を＋1 します、これは何度割り込みがあったか=何段目のデータを表示するかを設定します。
割り込みが 4 回めの場合、『LED_flag』を 0 に戻します。
『INTCON』レジスタの『TMR0IF』をチェックし、タイマー0 オーバーフロー割り込みが発生したことをチェックします。
『TMR0』レジスタをクリアし、タイマー0 をリセットした後、『TMR0IF』を初期化します。
この『TMR0IF』の初期化を忘れてしまうとずっと割り込みがかかった状態になってしまいますので注意しましょう。
一旦、LED を全部消灯させた後、LED_frag の内容に対応した周囲の LED の点灯パターンデータを PORTB と RA7 に、点灯する段数の PORTA を『0』にして、LED を点灯させます。
これを、タイマー0 オーバーフロー割り込みが発生するたびに各段の点灯パターンを表示させます。

```
//delay 関数変数化--------------
void time_ms(unsigned int t){

        unsigned int z;
        for(z=0;z<t;z++){
                __delay_ms(1);
        }
}
//-----------------------------
```

最後は以前も出てきた、delay 関数に変数を使えるようにした関数です。

以上で、C 言語による PICA Tower のプログラムの解説はおしまいです。
しかし、マイコンにはこの他にも色々な機能があり、LED の点灯だけでなく、色々と便利な用途に使用することができます。

2014 年 1 月 15 日　初版発行