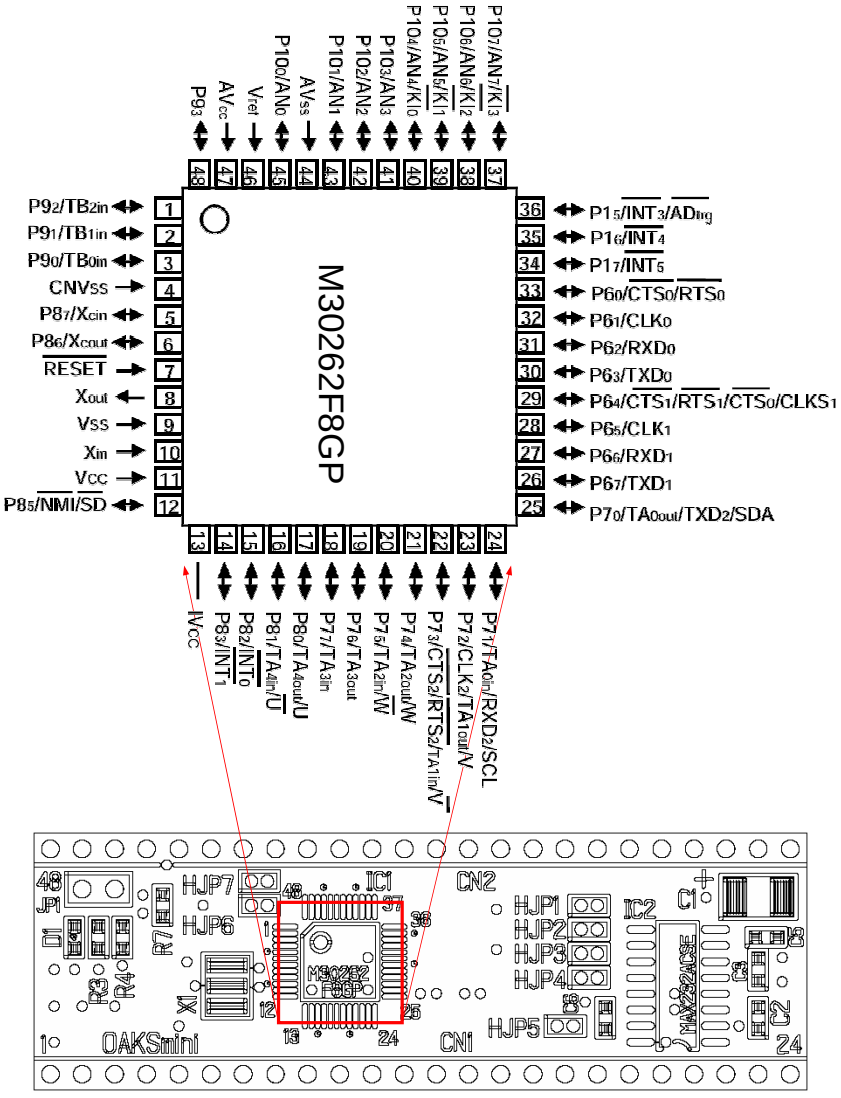
メカトロニクスラボ　C コース
– マイクロコントローラを動かしてみよう –

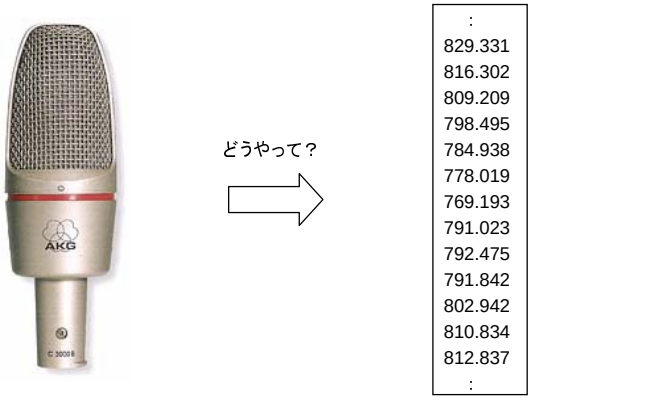
東京工業大学 工学部 制御システム工学科 4 学期
Rev. 1.1, November 2005
Rev. 2.1, October 2006
大学院理工学研究科 機械制御システム専攻
清水雅夫
mas@ok.ctrl.titech.ac.jp



目次

はじめに

コンピュータでプログラムを実行すれば、与えた情報を「処理」してくれます。このような処理は、既にC言語の実習を通して体験していると思います。それでは、音声信号を処理するためには、どのようにすればいいのでしょう。マイクロフォンを使えば、音声を電気信号に変換できるでしょう。では、その電気信号は、どうすればコンピュータで扱える「情報」になるでしょうか。

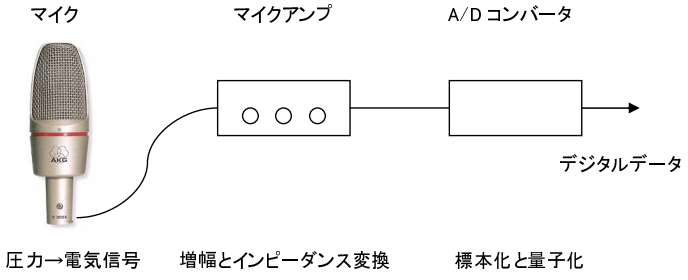


画像：http://www.sanden-shoji.co.jp/mic/akg/c3000b.html

音声信号を考える前に、身の回りにある家電製品を思い出してください。家電製品には、電源スイッチ以外にもスイッチがついています。家電製品の中には、もちろんコンピュータ(マイクロコントローラといいます)が使われていて、このマイクロコントローラがスイッチの状態を調べています。人間の意図によって変化するスイッチの状態は、もちろん「情報」です。マイクロコントローラは、入力された情報に応じてモータやランプを制御しているのです。

もっと具体的に、エアコンを考えてみましょう。エアコンには、動作スイッチや温度設定スイッチがあります。これらのスイッチは、人間が好みに応じて操作するものです。エアコンが作動すると、室温と温度設定スイッチに応じて温度制御のためのモータを駆動します。このとき、室温を測定する必要があります。どうやって測定しているのでしょうか。

マイクロコントローラが活躍する現実の世界には、スイッチの状態、温度、圧力、明るさ、などの物理的な量があります。このような物理量をマイクロコントローラで処理するためには、まずセンサで電気信号に変換して、さらにその電気信号をデジタル信号に変換する必要があります。メカトロニクスラボのマイクロコントローラ編では、電気信号をデジタル信号に変換してマイクロコントローラで処理する実習をします。

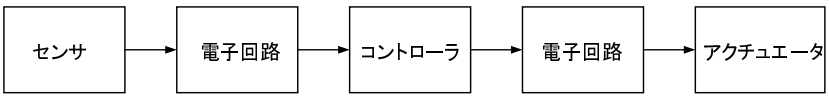


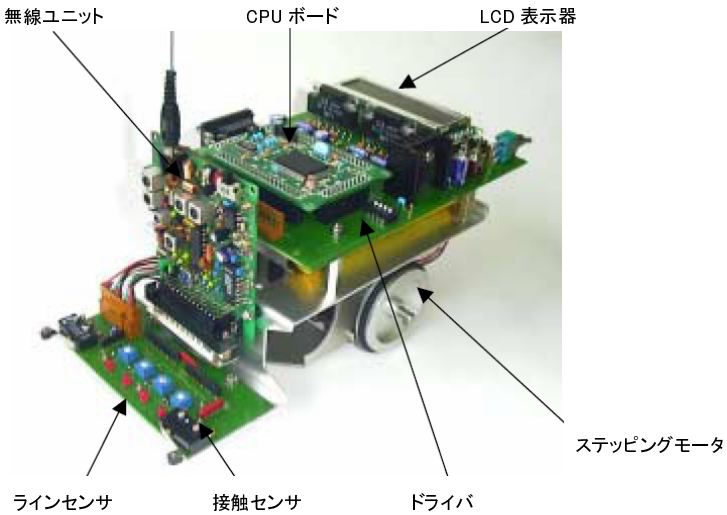
マイクロコントローラは、物理量を入力して処理するだけではありません。処理結果に応じて、モータなどを駆動するからこそ、「マイクロコントローラ」です。メカトロニクスラボのマイクロコントローラ編では、ラジコンサーボを制御する実習を通して、何かをコントロールして動かす楽しさを味わってもらいます。

制御、特にデジタル制御では、時間の概念がとても重要です。例えば、1秒ごとに温度を測定すれば正確な温度変化を知ることができます。しかし、同じ測定結果でも、どれほどの時間間隔で測定したものかがわからなければ、温度変化を知ることはできません。マイクロコントローラは、正確な周波数の「クロック」で動作しています。このクロックを利用して、一定時間ごとに何かをする、というプログラムの仕組みを作ることができます。

メカトロニクスラボのCコース（マイクロコントローラを動かしてみよう）では、このようにマイクロコントローラで何かを検出して何かを制御する、という実習を通して、マイクロコントローラのプログラムを学びます。同時進行の、メカニズム編や電子回路編で学ぶ、各種の機構、各種のセンサやアクチュエータ、電子回路を利用して、センサからの情報を処理してアクチュエータを駆動する、というマイクロコントローラのプログラムを作ります。

実習の各回では、最初にサンプルプログラムを各自のマイクロコントローラで動作させ、どのような特性があるか、何が問題か、などを考えます。次に問題の解決策を考え、各自で課題を解決します。解決策には、ただ一つの「正解」というものはありません。解決方法を考え、考えたとおりに実際にプログラムを書き、試して結果をよく見る、という実習を通して、センサ、電子回路、マイクロコントローラ、アクチュエータを実際に体験することが目的です。





画像：http://www.oaks-ele.com/oaks16/labo_manual/oaks-labo_catalogue.pdf

1　導入実習 – 実習セットの準備

この章の目次：

1.1　計算機室（南 5 号館 104 号室）の利用方法

1.1.1　磁気カードの登録

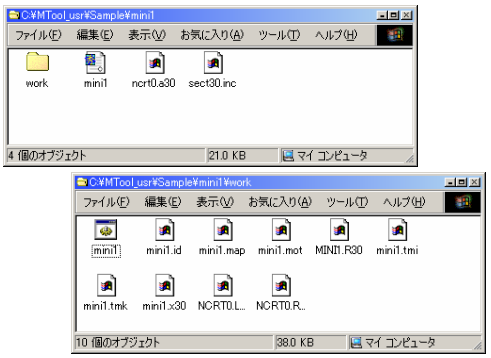
磁気情報が登録されているカードを、104 号室の入室カードとして利用します。**生協のカード**の利用をお勧めします。学生証は利用できません。磁気情報をドアの開閉器に読み取るので、重要な磁気カード（クレジットカード、銀行のカードなど）は利用しない方がよいでしょう。

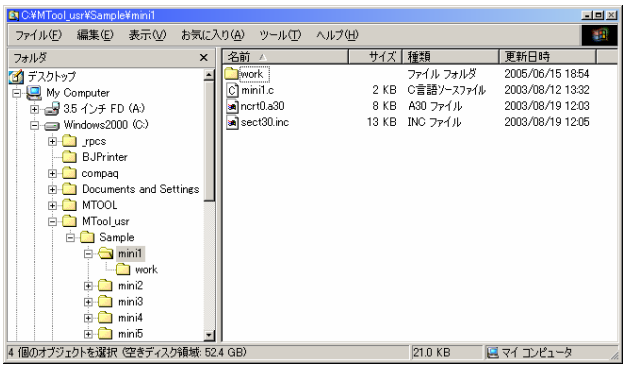
1.1.2　「南 5 号館 104 号室 利用の手引き」を参照

・利用可能時間。
・飲食禁止。
・傘の持ち込み禁止。
・ネットワーク構成と利用可能な資源。
・Windows2000 起動方法とパスワードの設定（[CapsLock] と [Ctrl] に注意）。
・プリンタの設定 – 「白黒、両面、2 ページを 1 枚に」を設定。
・各端末から見えるフォルダの構成。 – H:￥の下に各自のフォルダを作る。
・T:￥.emacs を H:￥の直下にコピー。
・インストール済みのアプリケーション。
・Office Suite のインストール – 「ワークステーションインストール」を選ぶこと。
・デスクトップの「インターネット接続ウィザード」アイコンをゴミ箱へ。
・「ゴミ箱」の注意 – **捨てたら元に戻らない。**

1.1.3 Windows のエクスプローラのプログラマ向け設定

Windows2000 のエクスプローラ（インターネットエクスプローラではなくて、ファイルエクスプローラの方）は、アプリケーションの起動、ファイルやフォルダのコピーや削除や移動などができる便利なファイルマネージャ（ファイル管理ソフトウェア）です。しかし、Windows2000 をインストールした直後の状態では、ウィンドウ内にアイコンとその名前しか表示されません（次の図の左）。表示されていない拡張子もあるし、階層構造内での現在のフォルダ位置もわかりません。これでは、あまりに不便です。

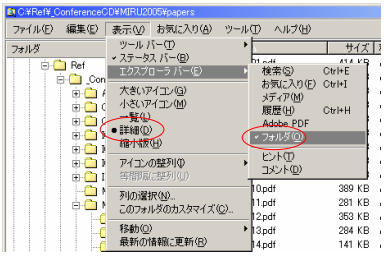


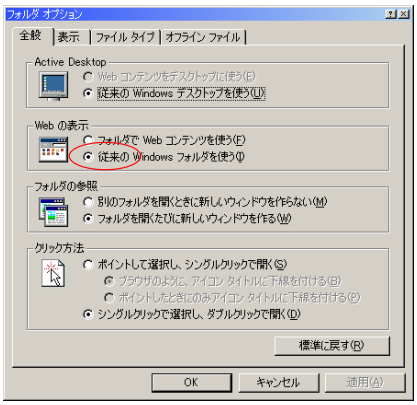


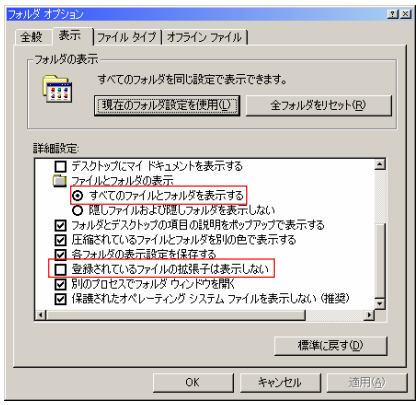
フォルダ構成全体の把握と、ファイル容量／拡張子／更新日時が確認できることは、プログラマにとってあまりにも当然のことです。エクスプローラの表示設定を変えるだけで、右側の図のようにフォルダ構成をツリー表示して、ファイルの属性を詳しく表示できるようになります。

このような表示にするためには、

0. デスクトップの「マイコンピュータ」の右クリックで出てくるメニューから「エクスプローラ」を選択。
1. [表示 (V)]-[詳細 (D)] を選択。
2. [表示 (V)]-[エクスプローラ バー (E)] ⇒ [フォルダ (O)] をチェック。







3. [ツール (T)]-[フォルダ オプション (O)...] で「フォルダ オプション」を表示。
4. 「全般」タブで、「従来の Windows フォルダを使う」を選択。
5. 「表示」タブで、「全てのファイルとフォルダを表示する」をチェック。
6. 同じく、「登録されているファイルの拡張子は表示しない」のチェックを外す（チェックしない）。
7. [適用] をクリック。
8. [現在のフォルダ設定を使用] をクリック。

1.2　実習で利用する拡張基板

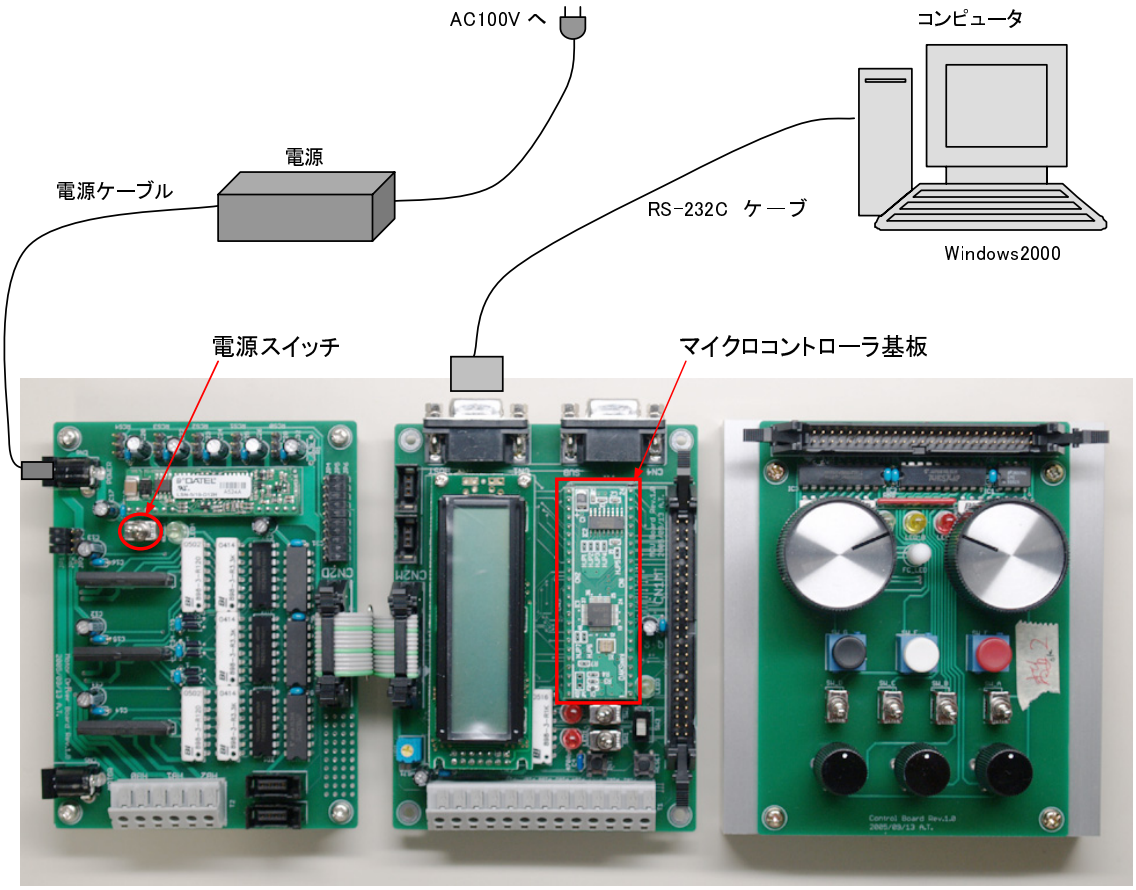
実習で利用する拡張基板は、メカトロニクスラボのために専用に製作されたものです。半年間期間限定で、各人に専用に貸し出します。拡張基板に付けたシリアル番号で、自分用の拡張基板であることを確認してから使うようにしてください。他の人の拡張基板は使わないでください。不具合があるときには、すぐに報告してください。

何か新しいことを学ぶ上で、試料や教材にダメージを与えてしまったり、場合によっては壊してしまうことは、ある程度はしかたないことと思います。しかし、そのときには、この実習の担当者の中の誰かが修理することになります。原則として、破損した部品の費用は壊した人が支払うことになります。注意して大切に扱ってください。

マイクロコントローラ基板を拡張基板に差し込む向きに注意してください。差し込むときには必ず、どんなに馴れても必ず、ピン番号を確認してください。逆さまに差し込むと、少なくともマイクロコントローラ基板が壊れます。壊れたら、修理は不可能ですから、マイクロコントローラ基板を各自で購入してください。

コンピュータのシリアルポートに接続した RS-232C ケーブルは、HOST コネクタに接続します。

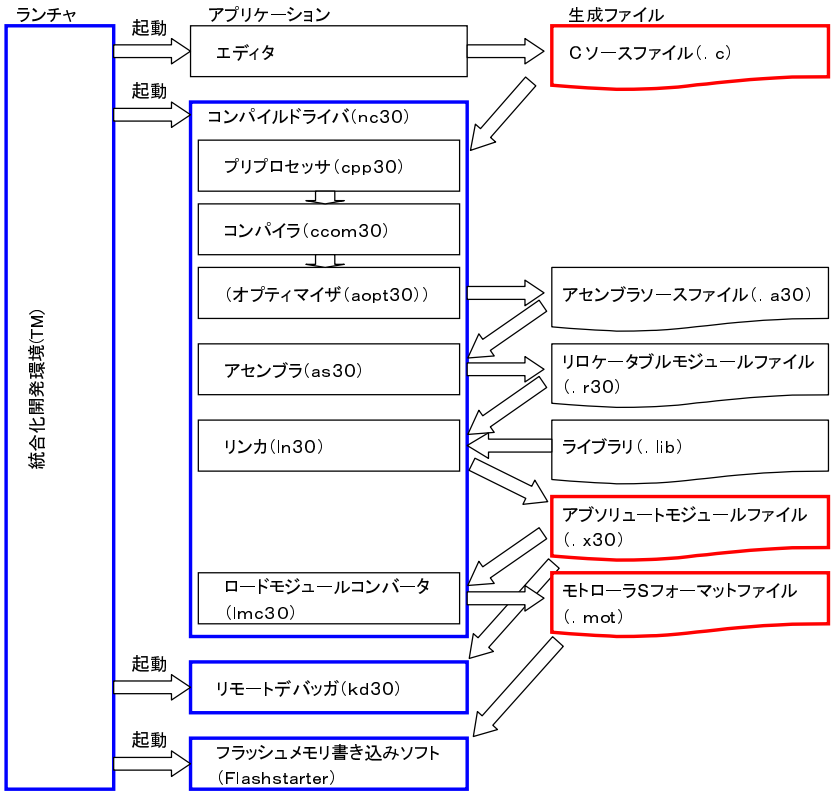
電源ケーブルは、拡張基板のコネクタ (POWER) に差し込みます。電源スイッチを入れると、LED4 が点灯します。



1.3 サンプルプログラムのコンパイルとダウンロード

1.3.1 利用するアプリケーションソフトウェアの関係

図の長方形は、アプリケーションを表します。青い長方形は、実習で利用するためにコンピュータにインストールしてあるアプリケーションを表します。各アプリケーションが生成するファイルの中で、赤で表したファイルは、他のアプリケーションで（意識して）読み込む必要があるファイルです。



統合化開発環境 (TM) は、アプリケーションを起動しやすくする、いわゆるアプリケーションランチャです。

エディタは、C ソースファイルを作るためのもので、統合化開発環境には付属していません。Windows に付属のものなら「メモ帳」(notepad.exe) が使えます。あらかじめインストールされているエディタとしては、「Meadow」があります。好みのエディタを TM に登録して利用します。

コンパイルドライバ (NC30) は、C ソースファイルを入力して最終的にアブソリュートモジュールファイルとモトローラ S フォーマットファイルを出力します。内部では、プリプロセッサ[1] 、コンパイラ[2] 、オプティマイザ[3] アセンブラ[4] 、リンカ[5] 、ロードモジュールコンバータ[6] が次々と自動的に起動されています。

[1] ソースファイルの一部を条件的にスキップしたり、ヘッダファイルを読み込んだり、マクロを展開するための前処理を行う。C ソースファイル中の「#」で始まる命令は、このとき展開される。

[2] 完全に展開された C ソースライルを、アセンブラソースファイルに翻訳する。アセンブラ命令は CPU ごとに異なるので、コンパイラも CPU ごとに異なる。

[3] C ソースの状態であらかじめ計算できる部分を計算したり、明らかに無意味なループを削除するなどの、各種最適化を行う。実習セットには付属しない。

[4] アセンブラソースファイルを CPU の命令コード（機械語などとも呼ばれる 16 進数表記のデータ）に置き換える。この段階では、置き換えた結果のデータを読み込むアドレスが確定していないので、出力はリロケータブル（再配置可能）モジュールになる。

[5] C ソースファイルからできた命令コードのデータや、あらかじめ用意された汎用の関数集（ライブラリ）をひとまとめにして、特定のアドレスに読み込んだときに実行できるようなデータ形式に変換する。

[6] リンカの出力を CPU にダウンロードするために、ファイル形式を変換する。変換結果は、テキストエディタでも表示できる。ダウンロードのときに転送エラーがあったら検出できるように、チェックサムが付加される。

リモートデバッガ (KD30) は、マイクロコントローラ内のモニタプログラムと通信して、リンカが出力したアブソリュートモジュールファイルをマイクロコントローラに転送して実行したり、指定箇所で実行を停止したり、変数の値をモニタすることができます。リモートデバッガで実行するプログラムは、確かにマイクロコントローラ内部で実行されますが、起動するためには RS-232C ケーブルで接続されたコンピュータ上で動作しているリモートデバッガが必要になります。

フラッシュメモリ[7] 書き込みソフトは、リンカが出力したアブソリュートモジュールファイルを変換したモトローラ S フォーマットファイルをマイクロコントローラ内部のフラッシュメモリに転送します。転送すると、マイクロコントローラ内部に単独で実行できるプログラムが保存されているので、プログラムの実行のためにコンピュータもリモートデバッガも必要ありません。

1.3.2 統合化開発環境の設定

統合開発環境 (TM) の起動と準備

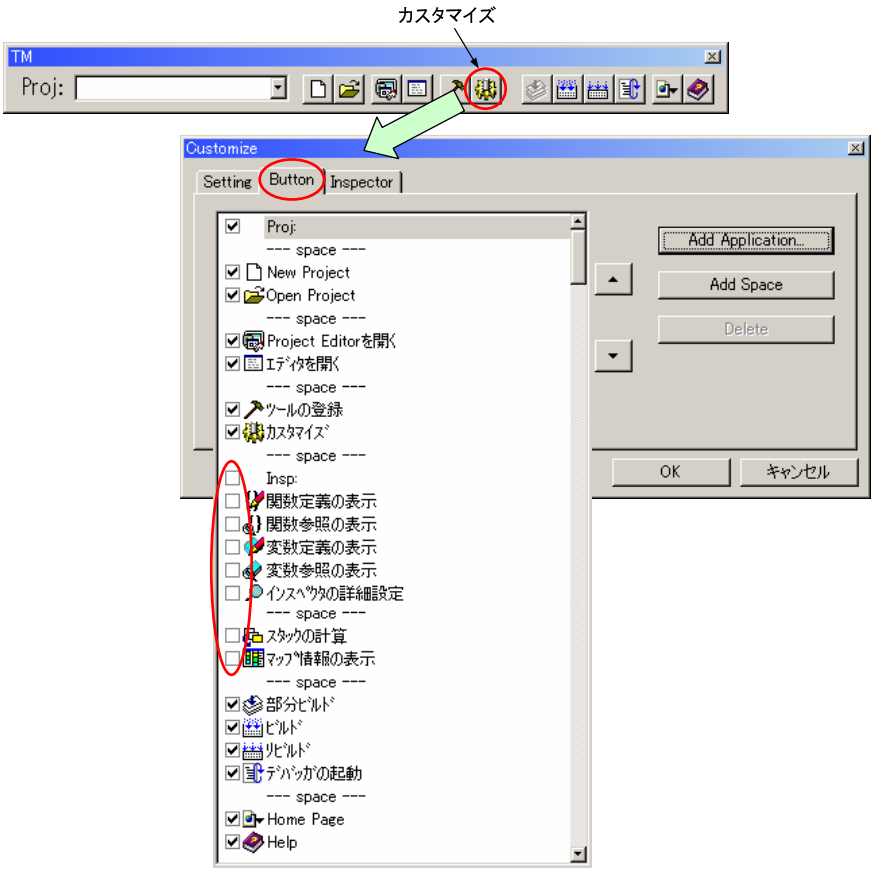
TM を起動すると、下図のようなツールバーが表示されます。

このツールバーには使用できないボタンも含まれているので、そのようなボタンを非表示にしてツールバーを見やすくしましょう。

(1) ツールバーの [カスタマイズ] ボタンをクリックします。

(2) [Button] タブを選んで、使わないボタンのチェックをはずします。

(3) [OK] をクリックします。



[7]CPU 内部のメモリで、CPU の電源を切っても保存される。

デバッガ (KD30) の登録

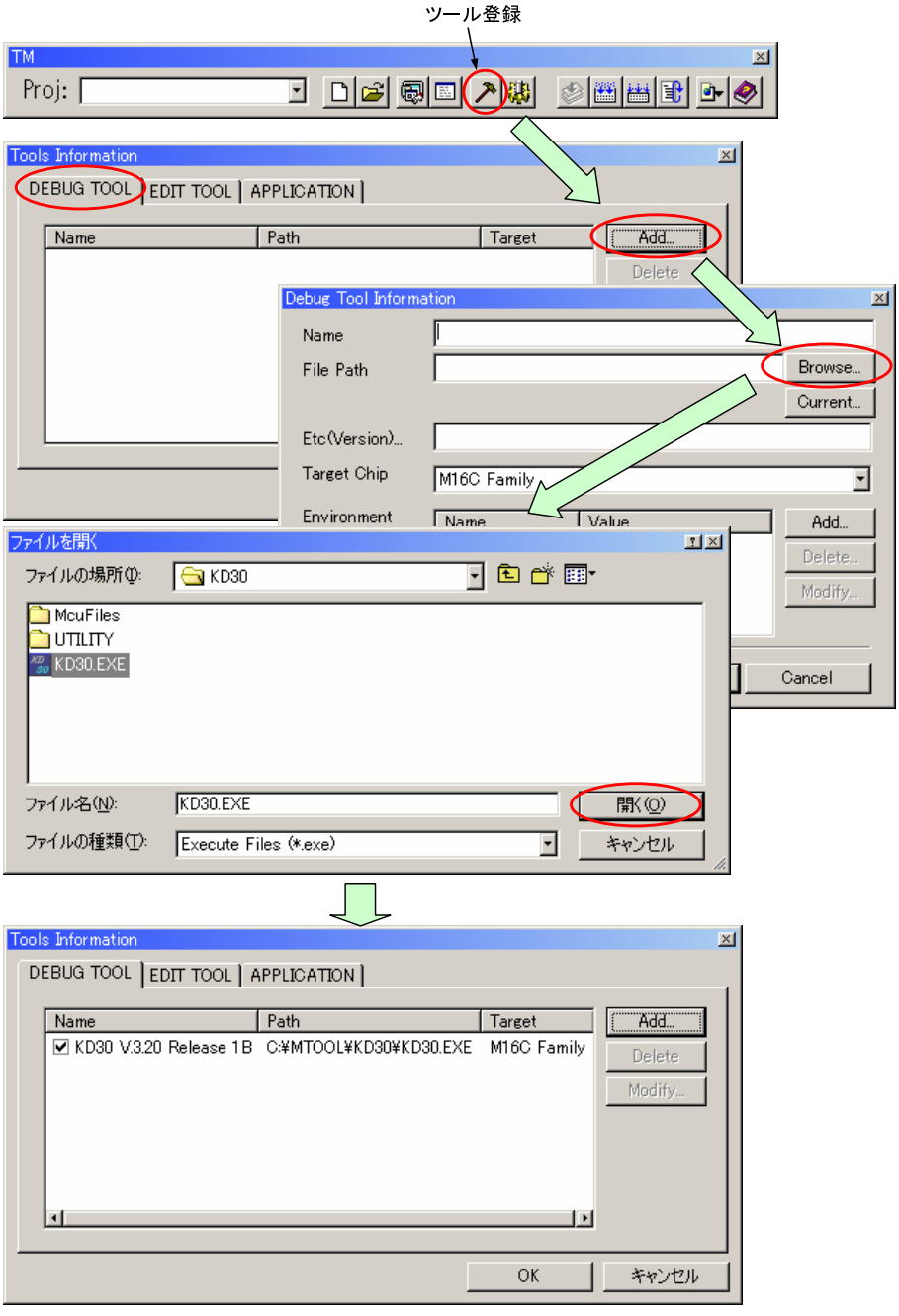
コンパイルしたプログラムをマイクロコントローラで実行するためには、デバッガが必要です。ツールバーから簡単に起動できるように、デバッガを登録します。

(1) ツールバーの [ツールの登録] ボタンをクリックします。

(2) ToolInformation ウィンドウの [DEBUG TOOL] タブで、[Add...] をクリックします。

(3) Debug Tool Information ウィンドウで [Browse...] をクリックして、C:￥MTOOL￥KD30￥KD30.exe を選びます。

(4) [OK] をクリックします。



フラッシュROM 書き込みソフトの登録

すぐに使うわけではありませんが、アプリケーションを登録して簡単に起動できるようにします。

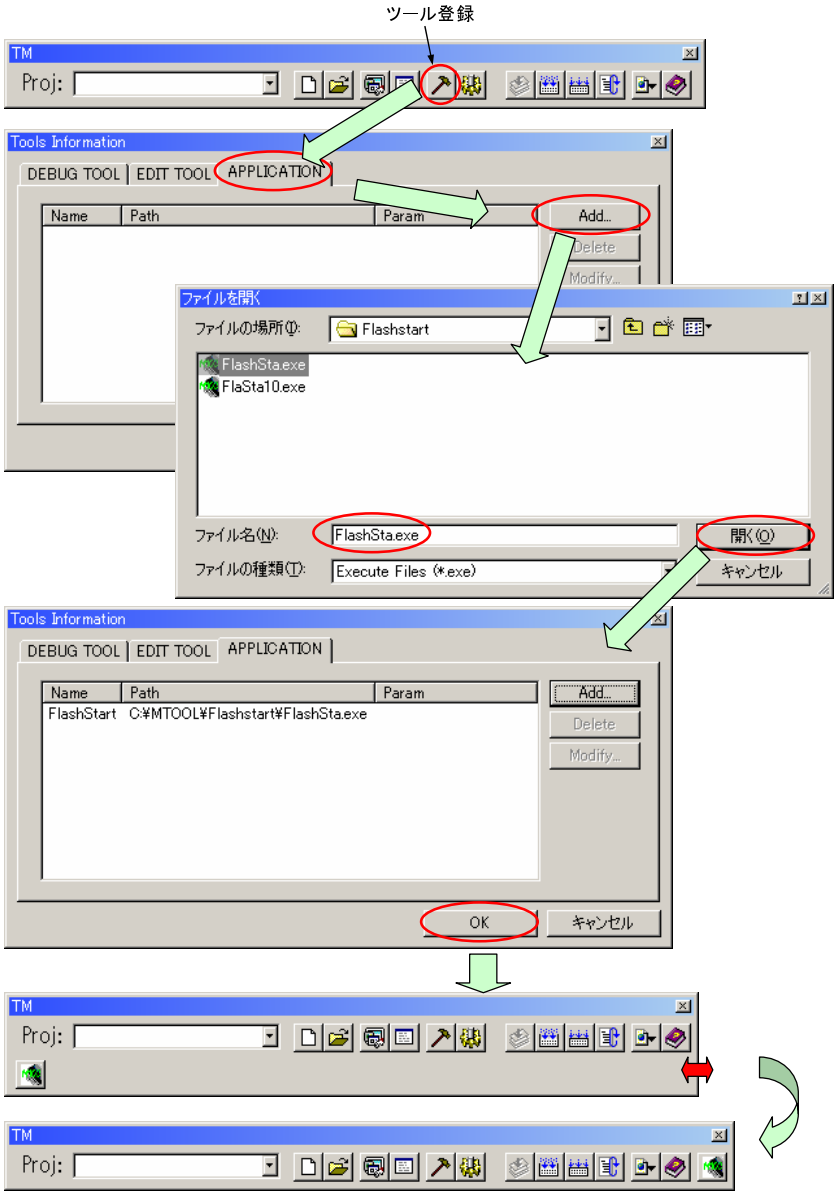
(1) ツールバーの [ツール登録] ボタンをクリックします。

(2) ToolInformation ウィンドウの [APPRLICATION] タブで、[Add...] をクリックします。

(3) Application Information ウィンドウで [Browse...] をクリックして、
`C:¥MTOOL¥Flashstart¥FlashSta.exe` を選びます。

(4) [OK] をクリックします。

(5) ツールバーが 2 段になってしまいますが、右側を引き延ばすようにすると 1 段に収まります。



エディタの登録

TM は、内部にエディタを持っていません。お好みのエディタ登録することができますが、ここでは、Meadow を登録します[8] 。

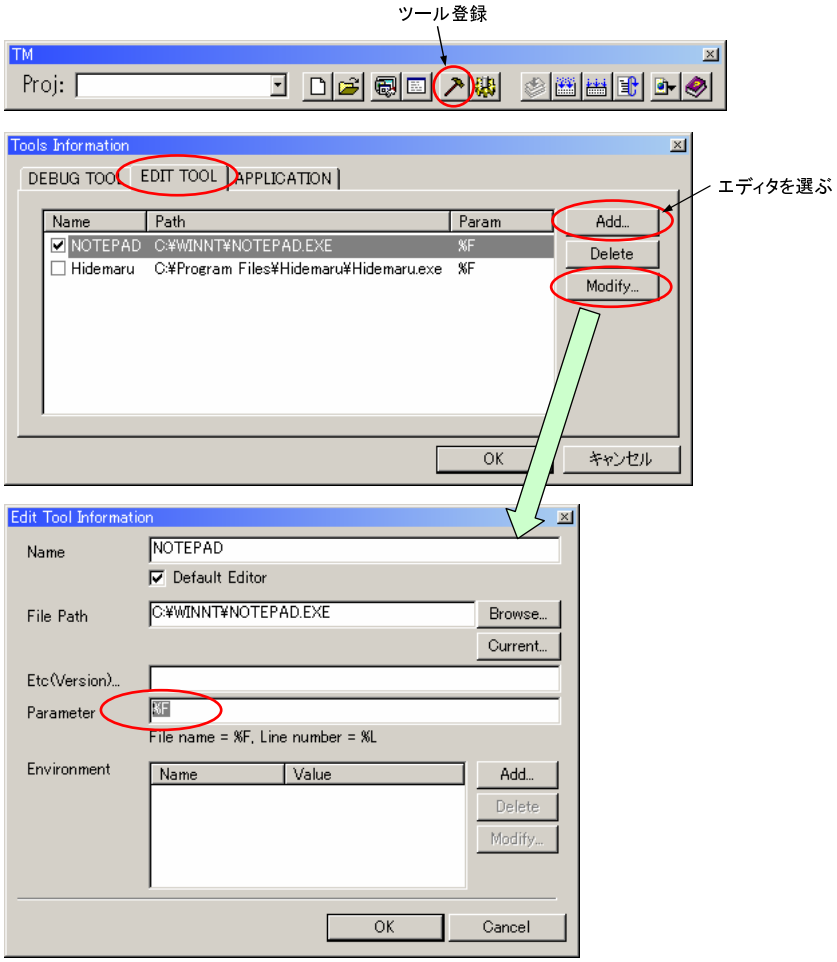
(1) ツールバーの [ツール登録] ボタンをクリックします。

(2) ToolInformation ウィンドウの [EDIT TOOL] タブで、[Add...] をクリックします。

(3) Edit Tool Information ウィンドウで [Browse...] をクリックして、
`C:￥Meadow￥2.00b2￥bin￥RunMW32.exe` を選びます。

(4) Edit Tool Information ウィンドウの Parameter に、「`+%L %F`」を指定します。

(5) [OK] をクリックします。



[8] 2 ページ後で使うことになる、リモートデバッガにはエディタが搭載されています。Meadow の代わりに Windows インタフェースを持つエディタを使いたい場合には、**6.6.4** を参照してください。

1.3.3　フォルダのコピー

共有フォルダ（T:ドライブ）にある sample フォルダを、フォルダごと H:ドライブのルートにコピーします。

注意：マイドキュメントやデスクトップにコピーしてはいけません。

1.3.4　マイクロコントローラで実行してみよう

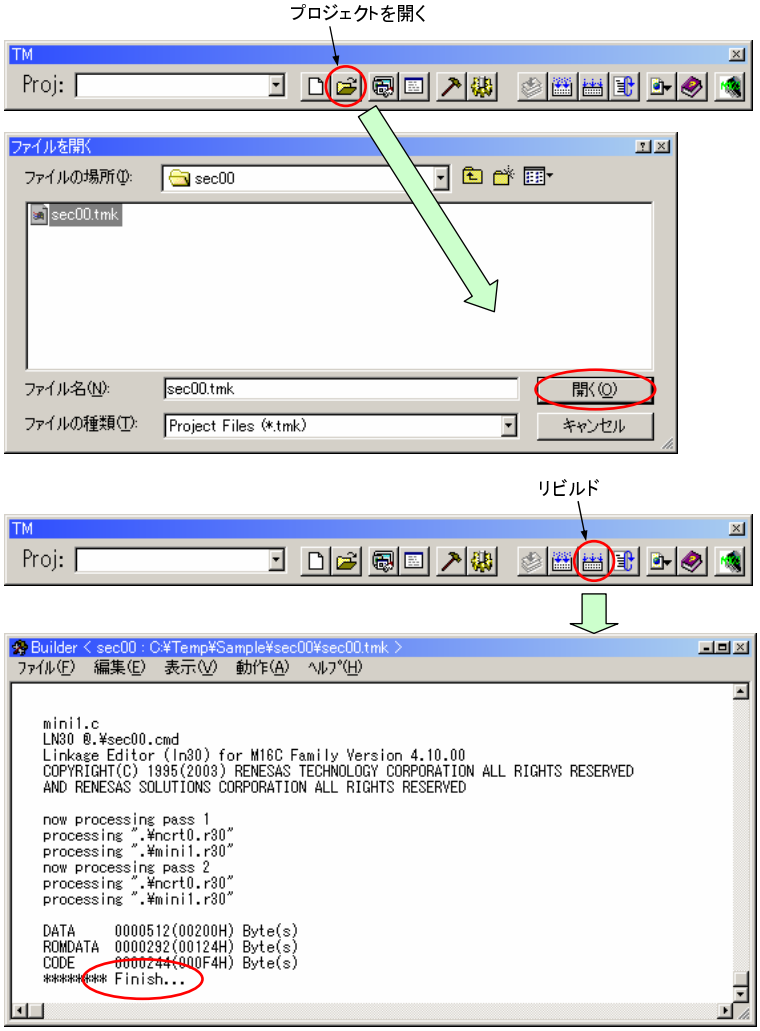
マイクロコントローラ基板をメカトロニクスラボ専用拡張基板に取り付けて、コンピュータとも接続して電源が入ったら、いよいよマイクロコントローラ基板でプログラムを動作させてみましょう。

サンプルプロジェクトのコンパイル

マイクロコントローラの動作を確認するために、プロジェクト[9] ファイルを開きます。

(1) [プロジェクトを開く] ボタンをクリックして、H:￥sample￥sec00￥sec00.tmk を選びます。

(2) [リビルド] ボタンをクリックします。



[9]「プロジェクト」は、最終的なプログラムを生成するために必要な一連のファイルのこと。実体は、「プロジェクト名.tmk」と「プロジェクト名.tmi」ファイル。「TM ユーザーズマニュアル」p.18 を参照。

参考 – ビルドとリビルドの違い

ビルドは、開いたプロジェクトに登録されているコマンドを実行します。このコマンドには、コンパイル、アセンブル、リンクなどの一連の作業が登録されていて、最終的にマイクロコントローラで実行できるファイルを作成します。要するに、make コマンドを起動するわけです。

TM では、3 種類のビルドができます。

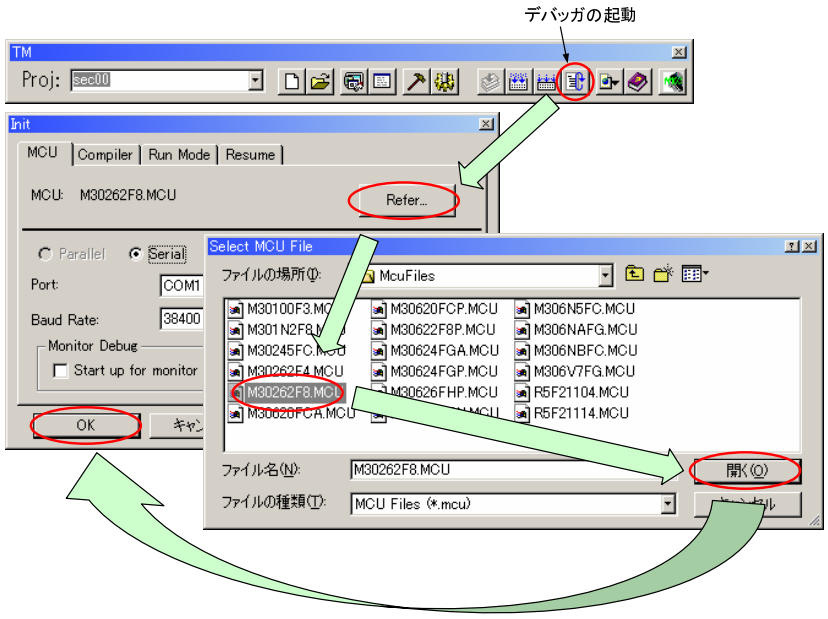
(1) 部分ビルド：　プロジェクトエディタで選択したソースファイルだけをビルドします。

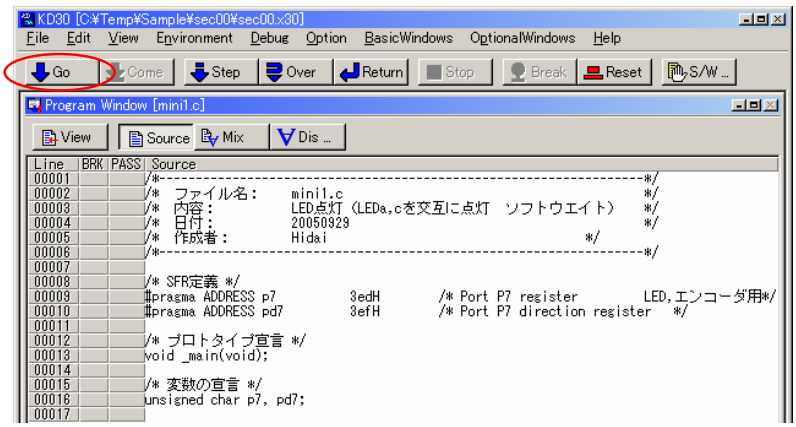
(2) ビルド：　make の実行と同様に、オブジェクトファイルが存在しないときや、オブジェクトファイルよりもソースファイルが新しいときに、必要に応じて一連のコマンドを実行します。ソースファイルが更新されていなければ、何もしません。

(3) リビルド：　clean に記述されているコマンドを実行してから、ビルドを行います。clean によって、中間生成ファイルとオブジェクトファイルが削除されるので、登録された全てのコマンドを必ず実行します。特に意図がないときには、常にこの**リビルド**をすれば良いでしょう。

リモートデバッガでプログラムを実行

(1) TM のツールバーの [デバッガの起動] ボタンをクリックして、デバッガを起動します。ただし、起動





しないこともあります[10] 。このときには、TM の [ツールの登録] ボタンでリモートデバッガ (KD30) をもう一度チェックします。デバッガが起動すると、Init ウィンドウが表示されます。一度だけ、デバッガの起動に必要な動作環境を設定します。

(2) Init ウィンドウで [Refer...] ボタンをクリックします。
(3) Select MCU File ウィンドウで、「M30262F8.MCU」を選択し [開く] をクリックします。
(4) Init ウィンドウで、[OK] をクリックします。
(5) マイクロコントローラと正常に通信できれば、デバッガが起動します。
(6) デバッガの [File]-[Download]-[Load Module...] で、`H:￥smaple￥sec00￥sec00.x30` を開きます。
(7) リモートデバッガの「↓ Go」ボタンをクリックします。
(8) プログラムが実行されると、リモコン基板上の LED が点滅します。
(9) プログラムを停止させるときは、デバッガの [STOP] ボタンをクリックします。

次回からは、(2) と (3) は不要です。[デバッガの起動] ボタンをクリックしたら、Init ウィンドウで [OK] をクリックすればデバッガが起動するはずです。

1.4 メカトロニクスラボ専用拡張基板と製品拡張基板との違い

メカトロニクスラボでは、オークス電子株式会社製のマイクロコントローラ基板を利用します。オークス電子株式会社は、このマイクロコントローラ基板用に拡張基板を販売しています。自宅でプログラムなどの開発を行うときには、オークス電子株式会社製の拡張基板を使うこともできます。ただし、全く同じように開発するというわけにはいきません。

この理由は、メカトロニクスラボ専用拡張基板と、オークス電子株式会社の製品である拡張基板キットでは、ポートの割付が異なるためです。マイクロコントローラ基板を、創造設計第二でも利用しやすいように変更しました。例えば、オークス電子株式会社の製品である拡張基板キットでは、アナログ入力も可能なポートを LCD に接続してしまっているので、アナログ入力を 1 本しか使うことができません。

ポートの割り当ては、次の表の通りです。左の列から、

・CPU のピン番号
・その信号名
・マイクロコントローラ基板のピン番号
・メカトロニクスラボ専用拡張基板でのポート割付
・メカトロニクスラボ専用拡張基板に接続するリモコン基板上のスイッチや LED など
・オークス電子株式会社の製品である拡張基板キットでのポート割付

を表しています。どちらも利用するときには、この違いをよく考えてプログラムを作るようにしてください。

[10] 理由は、プロジェクトファイルにリモートデバッガ (KD30) のインストールフォルダが記録されているため。サンプルプロジェクトを作ったコンピュータと実習用のコンピュータで、インストールフォルダが異なると、TM からデバッガを起動できない。

CPU	信号名	CN1	CN2	MC 拡張基板	MC リモコン基板	OAKS 拡張基板
	TXD0(MAX232)	24		Dsub-0 t2 (Sub)		J2-t2 (Option)
	RXD0(MAX232)	23		Dsub-0 t3 (Sub)		J2-t3 (Option)
	TXD1(MAX232)	22		Dsub-1 t2 (Host)		J2-t2 (Host)
	RXD1(MAX232)	21		Dsub-1 t3 (Host)		J2-t3 (Host)
7	RESET		48	プッシュSW to GND		SW2 to GND
4	CNVss		47	LED-3 / スライド SW to Vcc		JP1 to Vcc
6	P8-6/Xcout		46	LCD R/W		
5	P8-7/Xcin		45	LCD RS		
3	P9-0/TB0in		44	LCD DB4		
2	P9-1/TB1in		43	LCD DB5		
1	P9-2/TB2in		42	LCD DB6		
48	P9-3		41	LCD DB7		
47	AVcc		40			
46	Vref		39			
45	P10-0/AN0		38	ADC / I/O-0	VR-a	LCD DB4
43	P10-1/AN1		37	ADC / I/O-1	VR-b	LCD DB5
42	P10-2/AN2		36	ADC / I/O-2	VR-c	LCD DB6
41	P10-3/AN3		35	ADC / I/O-3		LCD DB7
40	P10-4/AN4/KI0*		34	ADC / I/O-4	トグル SW-a to GND	LCD E
39	P10-5/AN5/KI1*		33	ADC / I/O-5	トグル SW-b to GND	LCD R/W*
38	P10-6/AN6/KI2*		32	ADC / I/O-6	トグル SW-c to GND	LCD RS
37	P10-7/AN7/KI3*		31	ADC / I/O-7	トグル SW-d to GND	
36	P1-5/INT3*/ADtrg		30	ADtrg / I/O-8	プッシュSW-e to GND	
35	P1-6/INT4*		29	I/O-9	プッシュSW-f to GND	
34	P1-7/INT5*		28	I/O-a	プッシュSW-g to GND	
33	P6-0/CTS0*/RTS0*	27		LED-1		
32	P6-1/CLK0	20		LED-2		
31	P6-2/RXD0	19				Option
30	P6-3/TXD0	18				Option
29	P6-4/CTS1*/RTS1*/ CTS0*/CLKS1	17		トグル SW-1 to GND		
28	P6-5/CLK1	16				
27	P6-6/RXD1	15				Host
26	P6-7/TXD1	14				Host
25	P7-0/TA0out/ TXD2/SDA	13		JP to RCS-0 / Serial / H-br-0 PWM	フルカラー LED-Red	
24	P7-1/TA0in/ RXD2/SCL	12		JP to Serial / H-br-0 Dir	LED-a	
23	P7-2/TA1out/ V/CLK2	11		JP to RCS-1 / Serial / H-br-1 PWM	フルカラー LED-Green	
22	P7-3/TA1in/V*/ CTS2*/RTS2*	10		JP to Serial / H-br-1 Dir	LED-b	
21	P7-4/TA2out/W	9		JP to RCS-2 / H-br-2 PWM	フルカラー LED-Blue	LED3
20	P7-5/TA2in/W*	8		JP to H-br-2 Dir	LED-c	LED2
19	P7-6/TA3out	7		JP to RCS-3 / Enc-0 A (PhC)	エンコーダ a A	
18	P7-7/TA3in	6		JP to Enc-0 B (PhC)	エンコーダ a B	
17	P8-0/TA4out/U	5		JP to RCS-4 / Enc-1 A (PhC)	エンコーダ b A	SW5 to GND
16	P8-1/TA4in/U*	4		JP to Enc-1 B (PhC)	エンコーダ b B	SW4 to GND
15	P8-2/INT0*	3		プッシュSW to GND		SW3 to GND
14	P8-3/INT1*	2		トグル SW-2 to GND		
12	P8-5/NMI*/SD*	1		LCD E		
11	Vcc		26	5V (Vcc)	5V (Vcc)	5V (Vcc)
	GND		25	GND	GND	GND
9	Vss					
44	AVss					

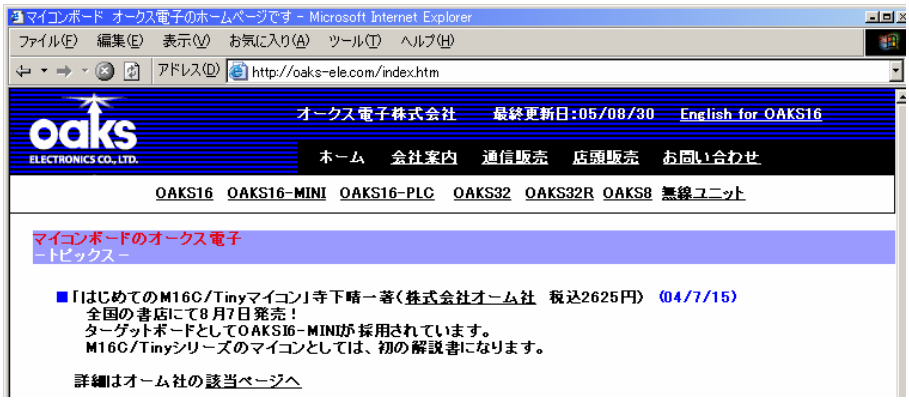
1.5　情報源など

製品の CD-ROM 内には、とりあえず必要なドキュメントが含まれています。必要に応じて、オークス電子株式会社や、CPU のメーカである株式会社ルネサステクノロジ[11] などのページをチェックするようにしてください。

ルネサスのマイクロコントローラは、各種の二足歩行ロボットによく使われています。ロボットの作者のホームページには、利用方法やテクニックなどの情報があることもあります。

これらのホームページを表示するためには、「http://www.google.com」で「オークス電子」「ルネサス」などを検索すれば OK です。

「ルネサステクノロジー」のホームページから、「製品」「マイクロコンピュータ – 統合開発環境」「ダウンロード」で、実習で使用する開発環境をダウンロードできます。ただし、ユーザ登録が必要です（たぶん無料）。ダウンロードする必要のあるファイルなどは、各自で調べてください。



[11] 日立製作所と三菱電機の、マイコン/ロジック IC/アナログ IC など、DRAM を除く半導体事業を引き継いだ会社。CPU の型番「M30262F8GP」の「M」は、実は三菱電機のマイコンに付けられた型番。

2　マイクロコントローラ入門

この章の目次：

2.1　目的

この章では、C 言語で記述したプログラムによって、マイクロコントローラが動作する仕組みを説明します。

まず、C 言語で記述したサンプルプログラムをマイクロコントローラにダウンロードして実行してみます。次に、このサンプルプログラムを例に、わかりやすく間違いが起きにくい C 言語の書き方を復習します。

そして、マイクロコントローラがプログラムを実行する仕組みを簡単に解説します。先ほどのサンプルプログラムを少し変更して、C 言語によってマイクロコントローラをプログラムできることを実感します。

最後に、統合化開発環境で新しいプロジェクトを作る方法を説明します。

2.2　サンプルプログラムのコンパイルとダウンロード

2.2.1　コンパイルとダウンロード

まず、共有フォルダ（T:ドライブ）にある sample フォルダを、フォルダごと各自の H:ドライブのルート直下にコピーします。（既に前回コピーしているはずです。）
注意：マイドキュメントやデスクトップにコピーしてはいけません。

(0) マイクロコントローラ基板を拡張基板にセットしてから、拡張基板に電源と RS-232C ケーブルを接続して、電源を入れます。
(1) TM を起動します。
(2) 「プロジェクトを開く」ボタンをクリックして H:￥smaple￥sec01￥sec01.tmk を開きます。
(3) 「リビルド」ボタンをクリックして、全てコンパイルします。時々、「製品版を購入しなさい」という勧告メッセージが表示されて動作が停止しますが、そのまま待ちます。コンパイルが終了すると、Builder ウィンドウに、「******** Finish... 」と出ます。
(4) 「デバッガの起動」ボタンをクリックして、リモートデバッガ (KD30) を起動します。
注意：起動しないこともあります。そのときは、「ツールの登録」ボタンをクリックして「DEBUG TOOL」の「KD30」をチェックしてから、「デバッガの起動」ボタンをクリックします。
(5) リモートデバッガの [File]-[Download]-[Load Module...] で、H:￥smaple￥sec01￥sec01.x30 を開きます。
(6) リモートデバッガの「↓ Go」ボタンをクリックします。

2.2.2　サンプルプログラムの動作

このプログラムは、一定の順序でプッシュスイッチを 3 回押すとフルカラー LED が赤く点灯し、別の一定の順序で 3 回押すと消灯する、というプログラムです[12] 。

2.2.3　参考：リモートデバッガでの実行とは

マイクロコントローラには、電源が入ると起動する「モニタプログラム」が書き込まれています。このモニタプログラムは、RS-232C 回線を通して送られてくるコマンドを待っています。コマンドは、例えば、
「これから RS-232C 回線を通してデータを送るから、それをアドレス xxxx からメモリに書き込め」
「アドレス xxxx からプログラムを実行せよ」
「実行しているプログラムを停止せよ」
「マイクロコントローラ内部レジスタを報告せよ」
といったものです。

つまり、リモートデバッガは、マイクロコントローラで操作しているモニタプログラムとの GUI(Graphical User Interface) なわけです。「アドレス xxxx から実行ってどういうこと？」とか、「プログラムを実行しているのにどうしてモニタプログラムは他のプログラムを停止できるの？」とか、疑問はつきないと思いますが、その疑問の一部はこの章の中で、一部は近いうちに解決されるでしょう。

リモートデバッガでの実行は、マイクロコントローラで操作しているモニタプログラムに対するコマンドで成り立っているので、デバッガが動作しているコンピュータがなければマイクロコントローラ上でのプログラムを起動することすらできません。チャレンジテーマでは、マイクロコントローラ単体での動作が求め

[12] 答え：黒→赤→白で点灯、赤→黒→白で消灯。

られるはずなので、このような実行方法とは別の方法が必要です。それが、フラッシュメモリに書き込んでの実行です。

フラッシュメモリに一度書き込んでしまえば、RS-232C ケーブルもデバッガもコンピュータも必要ありません。では、プログラムをデバッグするたびにフラッシュメモリに書き込んではどうでしょう。ところが、これには問題があります。フラッシュメモリは、書き込み回数に限界があるのです。デバッグでは、ちょっと変えて試してみて、またちょっと変えて試してみて、の繰り返しですが、そのたびにマイクロコントローラ内部回路の寿命を縮めていたのでは、安心して信頼性の高いプログラムを開発することもできません。

マニュアルを読んで、ちょっと試してみる分には大丈夫ですが、フラッシュメモリに書き込んで実行するのはもう少し待ってください。

2.3 C 言語の復習

ここでは、簡単に C 言語の復習をします。『プログラミングマニュアル＜ C 言語編＞』(6020jc.pdf) から、一部を抜粋して説明します。

システム開発（この実習では課題）を行う上で、最も時間がかかるのが、いわゆる「バグ取り」です。「バグ」には、大きく分けて 2 種類あると思います。

まず、アルゴリズムの検討が不十分なために起こる、プログラム動作の不完全さです。コンパイルも正常の終了して、プログラムも起動できるのに、実際に動作させてみると、思ったように動かない、という状況です。このときの対策は、どちらかというと簡単です。「技術者としてがんばっている」という、やりがいを感じるはずです。

もう 1 種類のバグは、非常にやっかいです。例えば、エラーが起きてコンパイルできないのに、ソースコードを何度見直しても何が悪いのかわからない、といった状況です。ほとんどの場合には、異なる変数型への代入やタイプミスが原因です。この種類のバグは、「発生させない」努力が大切です。見やすくわかりやすいソースコードを書くことで、このバグの発生を激減できます。

2.3.1 C 言語の特徴と規則

C 言語の特徴は、次に示すように、「関数」という単位でアルゴリズムを記述しやすく設計されていることです。

(1) 処理の流れを追いやすいプログラムが記述できる

構造化プログラミングの基本である「順次処理」、「分岐処理」、「繰り返し処理」を、全て制御文で記述できます。このため、処理の流れを追いやすいプログラムが記述できます。

(2) モジュール分割が容易にできる

C 言語で記述したプログラムは、「関数」と呼ばれる基本単位から構成されています。関数はパラメータの独立性が高いため、プログラムの部品化や再利用が容易にできます。また、アセンブリ言語で記述したモジュールをそのまま流用できます。

(3) 保守性のよいプログラムを記述できる

(1) と (2) の理由から、運用後のプログラムのメンテナンスが容易にできます。また、C 言語としての標準規格（ANSI 規格）が定められているため、ソースプログラムを少し変更するだけで他機種への移植も可能です。

また、一般的な規則は、次の通りです。

(1) プログラムは、原則的に英小文字で記述する。

(2) プログラムの実行文は、「;」で区切る。

(3) 関数や制御文の実行単位は、「{」と「}」で囲む。
(4) 関数や変数は、型宣言が必要である。
(5) 予約語は、識別子（関数名や変数名など）に使用できない。
(6) コメントは「/*　コメント　*/ 」で記述する[13] 。

2.3.2　特徴を生かすためのプログラミングスタイル

よく理解した上で適切に使えば大変強力でわかりやすい記述ができる C 言語ですが、予約語の中には悪名高い「goto」も含まれています。つまり、「関数」の形で独立性の高いモジュールを記述することも、あちこちに「goto」で飛んでいってわからない構造にすることも、プログラマ次第と言うことです。

わかりやすい記述は、直接、「バグ」を発生させにくいプログラムにつながります。そのためには、次のようなプログラミングスタイルを守ってください。

(1) 機能ごとに関数にする。
(2) １つの関数内の処理を大きくしない（画面をスクロールする必要がない 50 行前後が目安)。
(3) １つの行に複数の実行文は書かない。
(4) 処理ブロックごとに字下げを行う（通常は 4 タブを使用)。
(5) コメント文を効果的に記述してプログラムの流れを明確にする。
(6) プログラムを複数のソースファイルから作成する場合は、共通部分を別のファイルにして共有する。
(7) 変数名には、意味のある単語を使う。通常の変数と区別したいときには、意図して大文字を使う。
(8) 関数の内容の開始を表す「{」をどちらの行に付けるか、その前後にスペースを空けるか開けないか、if や for の中身が 1 行のときに「{}」を使うか使わないか、などは好みによる。統一した記述がよい。くどいようでも範囲が確実に限定されるように、「{}」を使う方がよい。

悪い例：

```
#pragma ADDRESS aaa    3e1H
#pragma ADDRESS aaaaa    3e3H
#pragma ADDRESS aa    3edH
#pragma ADDRESS aaaaaa    3efH
unsigned char aaa, aaaaa, aa, aaaaaa;
main(){aaaaa = 0; aaaaaa = 0xff; aa = 255;
while (1){ while ((aaa & 0xa0) == 0xa0) {}
if ((aaa& 0x20) !=0x20){while((aaa &0x80)== 0x80){}while ((aaa& 0x40)==0x40){}
aa = ∼0x01; }else {while((aaa &0x20)==0x20) {}while((aaa & 0x40) ==0x40  ) {}aa = ∼0x00;}}}
```

良い例：

```
/*=======================================================*/
/*    ファイル名：    0101.c                             */
/*    内容：          LED 点灯（フルカラー LED の赤を点灯）    */
/*    引数：                                             */
/*    戻り：                                             */
/*=======================================================*/

#pragma ADDRESS P1    3e1H        /* ポート P1 レジスタ */
#pragma ADDRESS PD1    3e3H        /* ポート P1 方向レジスタ */
                              /* 0：入力モード, 1：出力モード。bit0-4 は 0 書込 */
#pragma ADDRESS P7    3edH        /* ポート P7 レジスタ */
#pragma ADDRESS PD7    3efH        /* ポート P7 方向レジスタ*/
```

[13]「//」で始まる行も、その行の最後の改行まではコメントとして解釈されます。

```
                                /* 0：入力モード, 1：出力モード */
#pragma ADDRESS P10         3f4H    /* ポート P10 レジスタ */
#pragma ADDRESS PD10    3f6H    /* ポート P7 方向レジスタ */
                                /* 0：入力モード, 1：出力モード */

unsigned char P1, PD1, P7, PD7, P10, PD10;

main() {
    PD1 = 0x00;                     /* ポート 1 方向レジスタ　入力　*/
    PD10 = 0x00;                /* ポート 10 方向レジスタ　入力　*/
    PD7 = 0xff;                     /* ポート 7 方向レジスタ　出力　*/
    P7 = ∼0x00;                     /* ポート 7 出力 ∼L(LED 消灯)　*/

    while ( 1 ) {
        while ( (P1 & 0xa0) == 0xa0 ) {}    /* P1 の bit7or5 が 0 = 赤 or 黒押まで待つ */
        if ( (P1 & 0x20) == 0 ) {           /* P1 の bit5 が 0 = 赤押 */
            while ( (P1 & 0x80) != 0 ) {}    /* P1 の bit7 が 1 = 黒非押の間待つ */
            while ( (P1 & 0x40) != 0 ) {}    /* P1 の bit6 が 1 = 白非押の間待つ */
            P7 = ∼0x01;                       /* P7 の bit0 を 1 = フルカラー LED の赤色を点灯 */
        } else {                            /* P1 の bit7 が 1 = 黒押 */
            while ( (P1 & 0x20) != 0 ) {}    /* P1 の bit5 が 1 = 赤非押の間待つ */
            while ( (P1 & 0x40) != 0 ) {}    /* P1 の bit6 が 1 = 白非押の間待つ */
            P7 = ∼0x00;                       /* P7 の bit0 を 0 = フルカラー LED 消灯 */
        }
    }
}
```

2.4 マイクロコントローラの中身

ここでは、C 言語で記述したプログラムが、どのようなデータとしてマイクロコントローラにダウンロードされ、それをマイクロコントローラがどのような仕組みで実行するかを説明します。また、マイクロコントローラを使ってプログラムを作る上で必ず正確に理解する必要がある、「アドレス」と「データ」の違いを説明します。

2.4.1 C 言語に対応するアセンブリ言語と機械語

先ほど実行したサンプルプログラムをデバッガで「停止」すると、C 言語ソースプログラムに対応したアセンブリ言語と機械語を見ることができます。

プログラムを実行した状態では、赤または黒のプッシュスイッチが押させるまで待ち続けています。C 言語では、次の 1 行にあたります。

```
while ((P1 & 0xa0) == 0xa0) {} /* ポート P1 と 0xa0 との論理積が 0xa0 だったらループする */
```

この 1 行に対応するアセンブリ言語は、次のようになります。

```
MOV.B 03E1H,R0L ; R0 レジスタ下位バイトにアドレス 03E1H の 1 バイトをロードする。
MOV.B #0,R0H ; R0 レジスタ上位バイトに数値データ 0 をロードする。
AND.W #00A0H,R0 ; R0 レジスタ (2 バイト構成) と数値データ 00A0H との論理積を求める。
CMP.W #00A0H,R0 ; R0 レジスタ (2 バイト構成) と数値データ 00A0H とを比較する。
JEQ F0012H ; 等しかったらアドレス F0012H にジャンプする。
```

同じく、対応する機械語（アドレス F0012H から）を 16 進数表記すると、次のようになります。

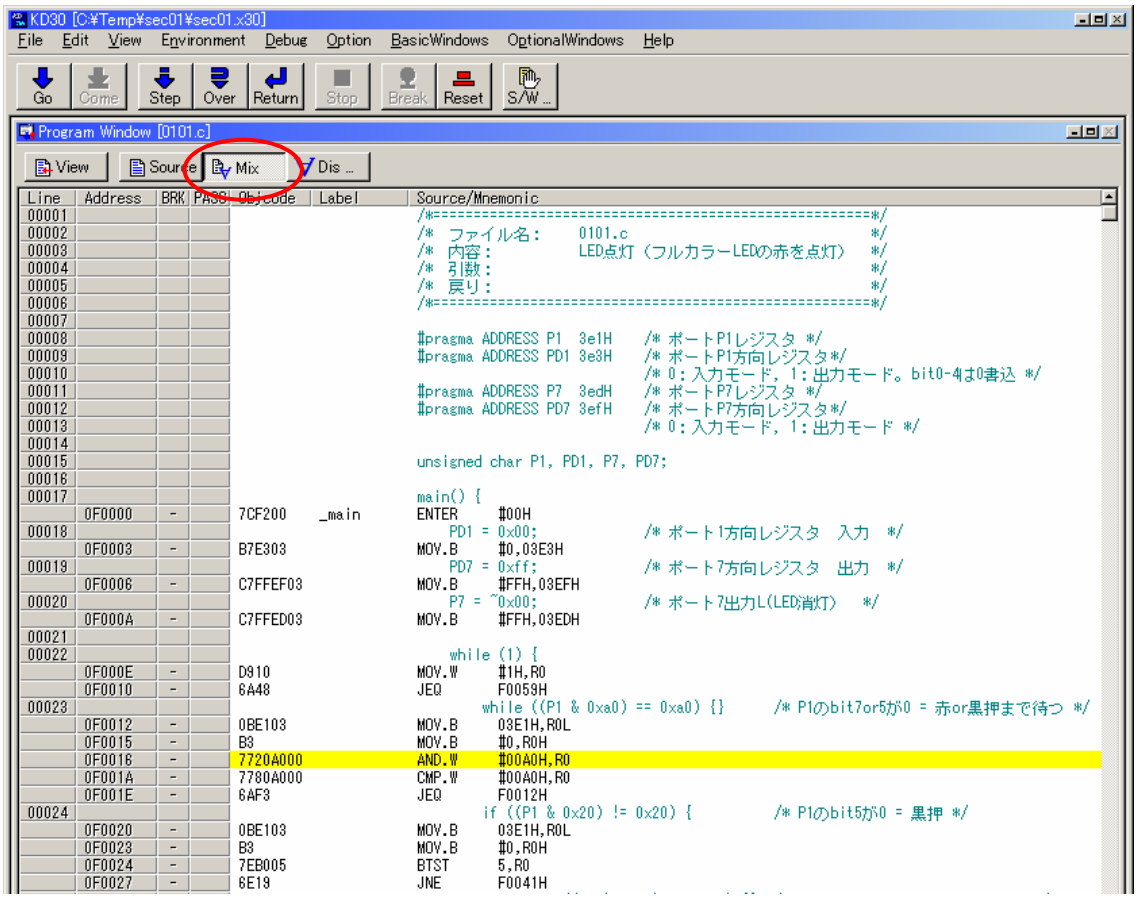


図. 2.1: 実行→停止した状態。表示モードを [Mix] にすること。

```
0BE103 B3 7720A000 7780A000 6AF3
```

マイクロコントローラは、この「機械語 (machine language)」を何も考えずに機械的に実行しています。マイクロコントローラ（CPU コア）の設計者は、この機械語を実行する仕様から設計をはじめます。マイクロコントローラが実行できる機械語は、マイクロコントローラごとに異なります。プログラム開発者の負担を考えなければ、機械語の仕様さえあれば、原理的にはどのようなプログラムでも開発できます。しかしそれではあんまりだ、ということで作ったのが、アセンブリ言語 (assembly language) です。

アセンブリ言語は、機械語と完全に 1:1 に対応します。C コンパイラが生成してくれない（かもしれない）CPU コアの高度で最新の演算命令を全て使うことができるので、OS の基盤となる部分やデバイスドライバなどは、今でもアセンブリ言語で記述されることがあります。

C 言語をアセンブリ言語に翻訳するのが、C コンパイラです。英語から日本語など、異なる言語間の翻訳と同様に、C コンパイラは設計者や最適化オプションによって、生成するアセンブリ言語が異なります。例えば前述の例では、ポート P1 は 1 バイト構成なので比較も 1 バイトだけで十分なのに、このコンパイラは 2 バイトでの比較を行うアセンブリ言語を出力しています。この部分を最適化するだけで、アセンブリ言語の 2 行目が不要で 3 行目と 4 行目の数値データが短くなるので、このループを表す 14 バイトが 3 バイト短くなります。ループ内の実行速度が、ほぼバイト数に比例すると考えると、21%も高速な機械語を生成できる可能性があります。

実習で利用するマイクロコントローラのアセンブリ言語と機械語に関しては、『アセンブリ言語マニュアル』(6020jasm.pdf) を参照してください。

2.4.2 レジスタ

アセンブリ言語では、「レジスタ」をよく利用します。レジスタは、マイクロコントローラ（の CPU コア）内にあって高速に読み書きできる特別なメモリです。プログラムでよく使う変数をレジスタに割り当てると、その変数に対する読み書きが極めて高速に実行できるので、プログラム自体の実行速度の向上します。アセンブリ言語で利用するときの、専用の機能が割り当てられたレジスタが何種類かあります。実習で利用するマイクロコントローラには、次のようなレジスタがあります（『ソフトウェアマニュアル』から抜粋）。詳細は、『ソフトウェアマニュアル』(6020jsm.pdf) を参照してください。

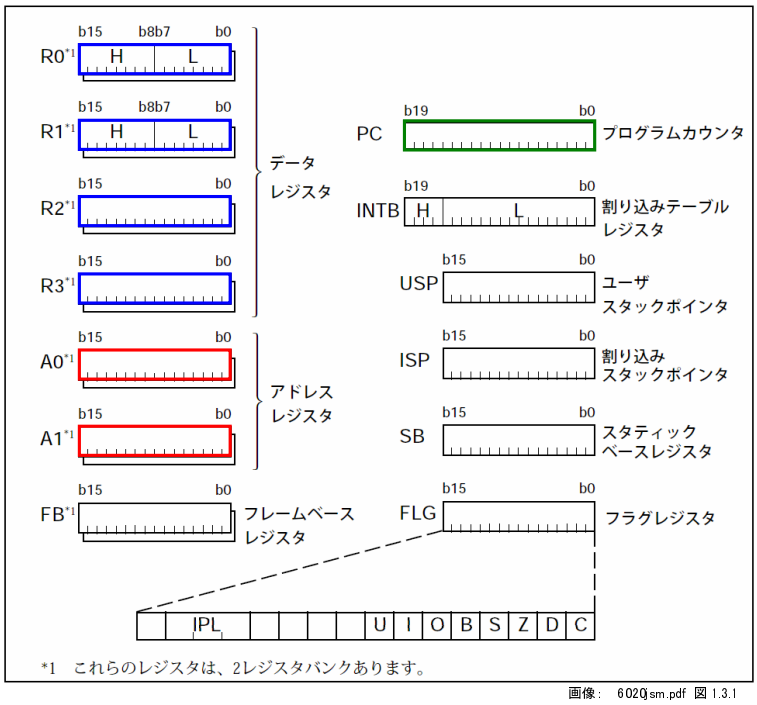


図. 2.2: レジスタ。

(1) データレジスタ： データレジスタ (R0/R1/R2/R3) は 16 ビットで構成されており、主に転送や算術、論理演算に使用します。R0/R1 は、上位 (R0H/R1H) と下位 (R0L/R1L) を別々に 8 ビットのデータレジスタとして使用することもできます。また、一部の命令では R2 と R0、R3 と R1 を組合せて 32 ビットのデータレジスタ (R2R0/R3R1) としても使用できます。

(2) アドレスレジスタ： アドレスレジスタ (A0/A1) は 16 ビットで構成されており、データレジスタと同等の機能を持ちます。また、アドレスレジスタ間接アドレッシングおよびアドレスレジスタ相対アドレッシングに使用します。一部の命令では A1 と A0 とを組合せて 32 ビットのアドレスレジスタ (A1A0) としても使用できます。

(3) フレームベースレジスタ： フレームベースレジスタ (FB) は 16 ビットで構成されており、FB 相対アドレッシングに使用します。

(4) プログラムカウンタ： プログラムカウンタ (PC) は 20 ビットで構成されており、次に実行する命令の番地を示します。

(5) 割り込みテーブルレジスタ： 割り込みテーブルレジスタ (INTB) は 20 ビットで構成されており、割り込みベクタテーブルの先頭番地を示します。

(6) ユーザスタックポインタ/割り込みスタックポインタ： スタックポインタは、ユーザスタックポインタ (USP) と割り込みスタックポインタ (ISP) の 2 種類があり、共に 16 ビットで構成されています。使用するスタックポインタ (USP/ISP) は、スタックポインタ指定フラグ (U フラグ) によって切り替えられます。U フラグは、フラグレジスタ (FLG) のビット 7 です。

(7) スタティックベースレジスタ： スタティックベースレジスタ (SB) は 16 ビットで構成されており、SB 相対アドレッシングに使用します。

(8) フラグレジスタ： フラグレジスタ (FLG) は 11 ビットで構成されており、1 ビット単位でフラグとして使用します。

2.4.3 プログラムカウンタ

アセンブリ言語でプログラムを書くときに主に使うレジスタは、データレジスタとアドレスレジスタです。これらのレジスタは、プログラムのアルゴリズムを実現するために利用するものです。

この他に、マイクロコントローラの仕組みを理解する上で重要なのが、「プログラムカウンタ」です。このレジスタ[14] は、これから処理しようとしているデータのある場所（アドレス）を指しています。

先ほどデバッガで確認したアセンブリ言語と機械語を、もう一度見てみましょう。

アセンブリ言語：

```
MOV.B 03E1H,R0L ; R0 レジスタ下位バイトにアドレス 03E1H の 1 バイトをロードする。
MOV.B #0,R0H ; R0 レジスタ上位バイトに数値データ 0 をロードする。
AND.W #00A0H,R0 ; R0 レジスタ (2 バイト構成) と数値データ 00A0H との論理積を求める。
CMP.W #00A0H,R0 ; R0 レジスタ (2 バイト構成) と数値データ 00A0H とを比較する。
JEQ F0012H ; 等しかったらアドレス F0012H にジャンプする。
```

機械語：

```
アドレス データ　(16 進数表示)
F0012 0BE103
F0015 B3
F0016 7720A000
F001A 7780A000
F001E 6AF3
```

プログラムカウンタ (PC) は、機械語の命令に応じて、どれだけ値をインクリメントするかをあらかじめ決められています。

例えば、PC がアドレス F0012H を指していたとします。CPU は、アドレス F0012H のデータを読み取り、0BH なので次の 2 バイトも読み取ります。この 3 バイトを一緒にして、機械語 0BE103H に対応する MOV.B 03E1H,R0L という動作を実行します。この機械語命令は、PC を自動的に 3 バイトインクリメントします。すると PC は、次の機械語 B3 が保存されているアドレス F0015H を指すことになります。この様子は、デバッガの [BasicWindows]-[Register Window] を表示してステップ実行すれば確認できます。

アセンブリ言語と対応する機械語については、『ソフトウェアマニュアル』(6020jsm.pdf) を参照してください。

参考： リセットしたときに、つまり最初に PC がどのような値を取るかは、あらかじめ決められています。このアドレスに実行できるプログラムを書き込んでおけば、そのプログラムが実行されます。

[14] アセンブリ言語プログラマが意図的に値を変えることは少ない。このため、「レジスタ」と呼ばれることはなく、必ず「プログラムカウンタ」と呼ばれる。

2.4.4　データの表現

データは単なる数値で、その数値のひとまとまりの長さ（バイト、ワードなど）や表現（2 進数表現、16 進数表現、10 進数表現、ASCII コードなど）は目的に応じて選ぶことができます。

ビット：　マイクロコントローラ内部では、最小のデータ単位を 1 ビットとしています。この単位は、0 と 1 のどちらかの値を取り、それ以外の値は存在しません。

バイト：　この最小単位（1 ビット）を 8 個まとめた単位が、1 バイトです。1 バイトは、2 進数で 00110110 などと表現できます。1 バイトで表現可能な組み合わせの数は $2^8 = 256$ 通り、0 から 10 進数の 255 までです。

ワード：　256 通りでは表現しきれない内容に対応するために、2 バイトや 4 バイトをまとめた単位が、ワードやロングワードです。多くのビットを効率的に表現するために、16 進数による表現が多く使われます。例えば、0011011011001110 = 36CEH [15] といった具合です。16 進数表記は、馴れると 10 進数表記とほとんど同じ感覚で読み取ることができるようになります。1 バイトのデータを 10 進数で表現するためには 3 桁が必要ですが、16 進数で表現すればちょうど 2 桁で表現できます。

10 進数表記	0	1	2	3	4	5	6	7
2 進数表記	0000	0001	0010	0011	0100	0101	0110	0111
16 進数表記	0	1	2	3	4	5	6	7

8	9	10	11	12	13	14	15
1000	1001	1010	1011	1100	1101	1110	1111
8	9	A	B	C	D	E	F

2.4.5　2 バイト以上のデータの並び順

メモリ上には、8 ビット単位（バイト単位）でデータを配置します[16] 。つまり、データのある「場所」をバイト単位で指定することができます。

それでは、表現したい数値が 255 よりも大きくなってしまったら、どうすればいいでしょう。2 バイトをひとまとめにして、ワードと呼ばれる 16 ビット構成のデータ単位にすれば、$256 \times 256 = 65536$ までの数値を表現できます。4 バイト構成にすれば（ロングワード）、$256 \times 256 \times 256 \times 256 = 4294967296$ までの数値を表現できます。しかし、ワードやロングワードでは、上位バイトと下位バイトを、どのように配置するかを決める必要があります。

この実習で利用するマイクロコントローラでは、下位バイトを小さいアドレスに配置します（リトルエンディアン[17] ）。

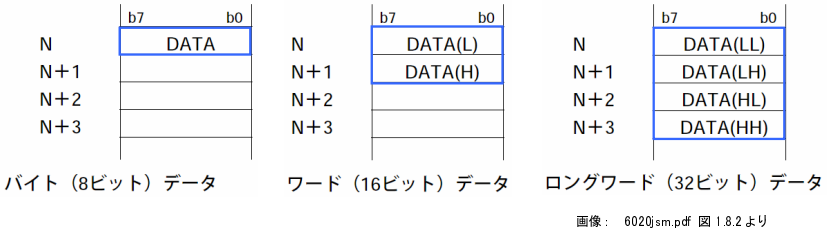


図. 2.3: 1 バイト、2 バイト、4 バイト構成のデータの配置。

[15] 16 進数表記であることを表すのに、「H」を付ける。C 言語では、0x36 と表記する。他に、36_{16} の表記も利用されている。
[16] この実習で利用するマイクロコントローラでは、バイト単位。他に、16 ビットや 32 ビット構成が可能な CPU もある。
[17] 上位バイトを小さいアドレスに配置する方式を、「ビッグエンディアン」という。Intel 系列はリトルエンディアン、Motorola 系列はビッグエンディアンである。

2.4.6　アドレス

アドレスは、「場所」を表しています。アドレスも、数値データと全く同様に、どのような表記をすることもできますが、ほとんどの場合 16 進数表記で表示します。四則演算などの演算をすることもできます。演算をした結果、指す「場所」が変化します。定数を加算や減算すれば、指す「場所」が加算した定数だけ移動します。

データは、図 2.4 のようなメモリに保存されます。図 2.4 のメモリは、「8 ビット構成」と呼ばれ、アドレスを指定すると 8 ビットのデータを同時に書き込みまたは読み出しすることができます。他に、1 ビット、4 ビット、16 ビット、32 ビット構成のメモリがあります。

プログラム上では、アドレスは 2 次元ではなく 1 次元（変数が 1 つだけ）です。このイメージをそのまま半導体上に実現すると、大変細長いメモリチップができあがります。これでは不便なので、正方形に近いチップ上にメモリを構成できるように、アドレスを 2 つに分けて、縦方向（行：row）と横方向（列：column）でメモリチップ上の「場所」を指定しています。

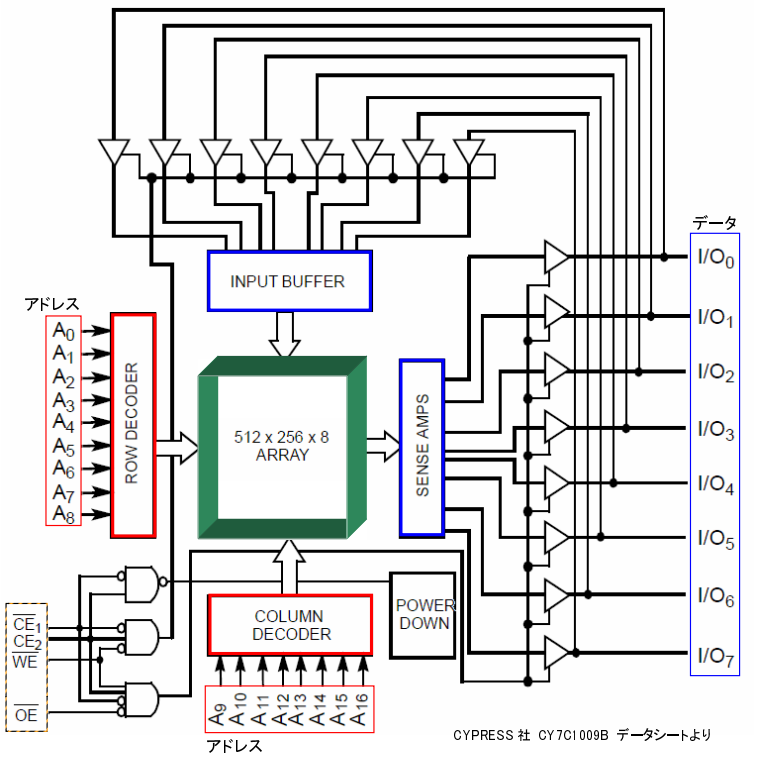


図. 2.4: 8 ビット構成メモリのブロック図。

2.4.7　アドレス空間

実習で利用するマイクロコントローラには、20 本のアドレス信号があります。20 本のアドレス信号で表現できる「場所」は、$2^{20} = 1048576 = 1\text{M}$ 通りです。1ヶ所の「場所」を指定すると、8 ビット（1 バイト）ひとまとめのデータを読み取ったり書き込んだりすることができます。このアドレスを、00000H から FFFFFH までで表現します。

00000H〜003FFH は、SFR（スペシャルファンクションレジスタ）領域です。マイクロコントローラに搭

載された各種周辺機能を制御するためのレジスタが、配置されています。

00400H 以降は、メモリ領域です。00400H からアドレスの大きい方向に RAM 領域を、FFFFFH からアドレスの小さい方向に ROM 領域を配置しています。

FFE00H〜FFFFFH は、固定ベクタ領域です。

詳細は、『ハードウエアマニュアル』(rjj09B0033_m16chm.pdf) を参照してください。

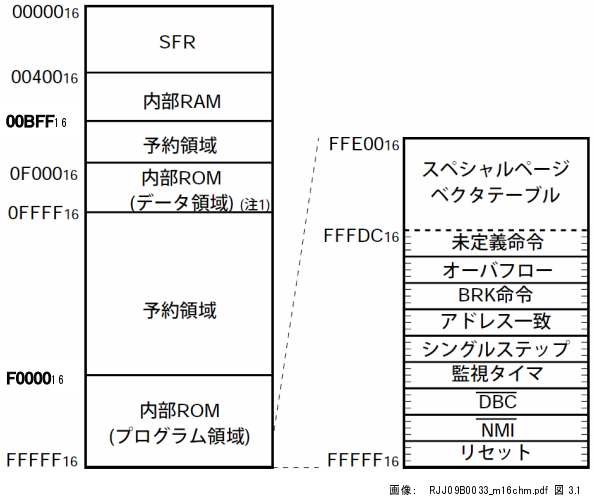


図. 2.5: アドレス空間。

2.4.8 周辺機能

実習で利用するマイクロコントローラは、CPU コアだけではなく、図 2.6 に示すように、I/O ポート、タイマ、監視タイマ、DMAC、AD 変換器、シリアル I/O、システムクロック発生回路などの周辺機能を搭載しています。これらの周辺回路の中の、I/O ポート、タイマ、AD 変換器に付いては、次回以後で順次説明します。ここでは、周辺機能を簡単に説明します。詳細は、『ハードウエアマニュアル』(rjj09B0033_m16chm.pdf) を参照してください。

I/O ポート： プログラマブル入出力ポートは、P15〜P17、P6、P7、P80〜P83、P85〜P87、P90〜P93、P10 で、合計 38 本あります。各ポートの入出力は、方向レジスタによって 1 本ごとに設定できます。また、4 本単位でプルアップ[18] するかしないかを選択できます。

タイマ： 16 ビットタイマが 8 本あり、機能によってタイマ A(5 本) とタイマ B(3 本) の 2 種類に分類されています。それぞれ独立して動作します。マイクロコントローラ内部のクロックをカウントするモード、外部クロックをカウントするモード、パルス幅変調モード、パルス周期やパルス幅を測定するモード、などに設定できます。

監視タイマ： 監視タイマ（ウォッチドッグタイマ）を使うと、プログラムの暴走を検出することができます。監視タイマは 15 ビットのカウンタで、内部クロックをダウンカウントします。このカウンタがアンダーフローしたときに、CPU の割り込みまたはリセットをかけることができます。プログラムが正常に動作している間は、定期的にこのタイマを初期化してアンダーフローを起こさないようにします。何らかの原因でプログラムが正常に動作しなくなると、定期的なタイマ初期化もしなくなりますが、このときには自動的にリセットしてプログラムを再実行してくれます。

[18] 抵抗でポートと電源を接続すること。

DMAC： DMAC（ダイレクト・メモリ・アクセス・コントローラ）は、CPU を使わずにデータを転送する機能です。2 チャネルあります。DMAC は、DMA 要求が発生すると転送元番地の 1 バイトまたは 2 バイトのデータを転送先番地に転送します。CPU と同じデータバスを使用しますが、DMAC のバス使用権は CPU よりも高く、サイクルスチール方式を採用しているために、DMA 要求が発生してからデータ転送を完了するまでの動作が高速です。

AD 変換器： 10 ビットの逐次比較変換方式 AD 変換器を、1 回路搭載しています。アナログ入力は、P100～P107 と端子を共用しています。

シリアル I/O： クロック同期形シリアル I/O、クロック非同期形シリアル I/O、I2C Bus などの各種方式でシリアル通信ができます。シリアル I/O は、UART0～UART2 の 3 チャネルで構成しています。リモートデバッガは、チャンネル 1 を使ってマイクロコントローラと通信しています。

クロック発生回路： メインクロック発振回路、サブクロック発振回路、リングオシレータの 3 つのクロック発生回路があります。メインクロックは周波数が最大 20MHz で、主に CPU のクロック源として利用します。サブクロックは 32.768KHz で、低消費電力モード時の CPU のクロックや、タイマのクロック源として利用します。リングオシレータの周波数は約 1MHz で、メインクロックまで停止した状態での CPU のクロック源になります。

2.5 新しいプロジェクトの作り方

ここでは、統合化開発環境 TM を使って、新しいプロジェクトを作る方法を説明します。新しいプロジェクトは、いくつかの方法で作ることができますが、ここでは、最も簡単な方法を説明します。詳細は、『TM V.3.20A ユーザーズマニュアル』(`TMUJ.pdf`) を参照してください。なお、一部の説明は、『TM V.3.20A ユーザーズマニュアル』からの抜粋です。

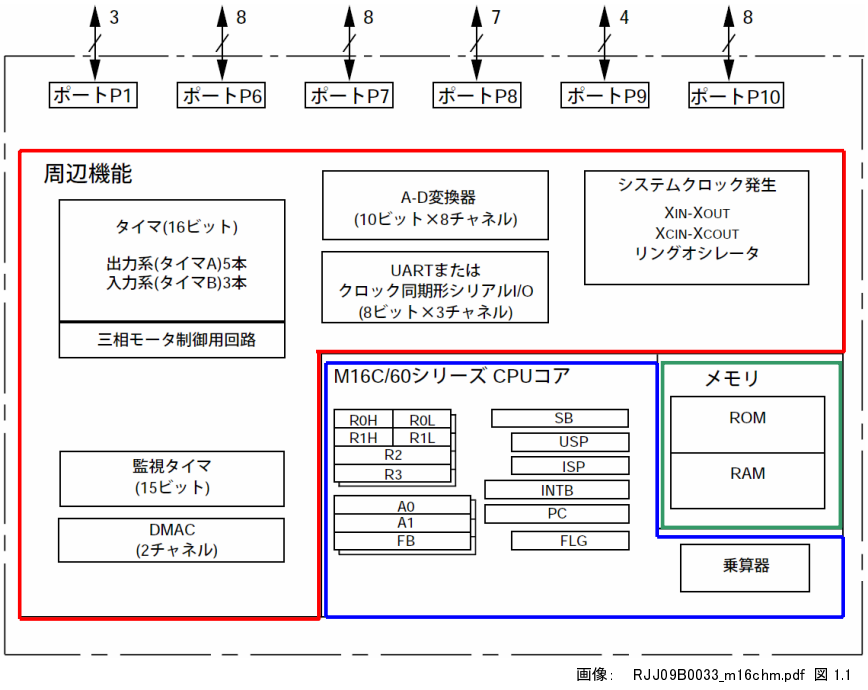


図. 2.6: 周辺機能。

2.5.1 プロジェクトとは

プロジェクトは、目的のソフトウエアをコンパイルするための情報です。プロジェクトによって、次のような情報を管理しています。
(1) 最終的な実行オブジェクト[19] を生成するための手順。
(2) 開発に関連するソースファイル。
(3) 使用する CPU ファミリに対応した生成依存関係。

プロジェクトを管理するために、TM では、次の 2 つのファイルを使います。
(1) 「プロジェクト名.tmk」
(2) 「プロジェクト名.tmi」

tmk ファイルは、UNIX 等で標準的に使われている「make」コマンドに対応したメイクファイル形式になっていて、最終オブジェクトを生成するための情報を保持しています。
tmi ファイルは、メイクファイル形式の tmk ファイルで管理できない情報を保持しています。

2.5.2 フォルダの準備

新しいプロジェクトを作るときには、まず、そのプロジェクト用のフォルダを用意します。このフォルダの中に、目的とするソフトウェアをコンパイルするのに必要なファイルを全て保存します[20] 。
フォルダを作るためには、これから作るフォルダを置くところを右クリックして「新規作成 (W)」→「フォルダ (F)」を選びます。
フォルダ名とパスには、次の制限があります。
(1) 漢字（2 バイト文字）を含むフォルダ名やファイル名は使用できません。
(2) ファイル名には、ピリオド「.」を 1 つだけ使用可能です。（使わなくてもよい）
(3) ネットワークパス名は使用できません。
(4) 「ショートカット」は使用できません。
(5) “My Documents” や “Program Files” のような、空白文字を含むディレクトリ名やファイル名は使用できません。
(6) “...” 表記を使った 2 つ以上のディレクトリを指定することはできません。
(7) パス指定を含めたファイル名の長さは、128 文字未満に制限されています。

フォルダを作ったら、`H:￥sample￥startup` フォルダ内の 5 つのファイルを作ったフォルダにコピーします。
注意： コンパイラのインストールフォルダ`￥MTOOL￥SRC30￥STARTUP`(例) にも同じファイル名のファイルがあります。このファイルは、**使わないで**ください。

2.5.3 ソースファイルの準備

次に、用意したフォルダに、ソースファイルを新規作成します。用意したフォルダ内で右クリックして、「新規作成 (W)」→「テキスト文書」を選びます。ファイル名には、フォルダ名と同じ制限があります。要するに、英数字で短いファイル名を付けてください。「新規テキスト文書.txt」では、**だめです**。

[19] マイクロコントローラにダウンロードして実行できるようにコンパイルした結果を、実行オブジェクトという。リンカがリンクする前の中間ファイルもオブジェクトと呼ばれることがあるので、「実行オブジェクト」のように「実行」を付けて区別した方がよい。
[20] 共通で利用するファイルを、別のフォルダに保存しておくこともできる。

図. 2.7: フォルダの準備。この例では、¥MecLab¥Kadai¥Sec01 フォルダを用意した。ファイルの日付を確認すること。「2003 年 4 月 2 日」のファイルは**使えない**。

図. 2.8: この例では、¥MecLab¥Kadai¥Sec01 フォルダに kadai.c ファイルを新規作成したところ。

2.5.4 最後に TM でプロジェクトを新規作成

ここまで準備ができたら、TM の「New Project（プロジェクトの新規作成）」ボタンをクリックします。

(1) 最初のダイアログで、「プロジェクト名」を入力します。よく考えてフォルダ名を付けたはずなので、フォルダ名と同じでいいでしょう。

(2) ワーキングディレクトリは、[...] ボタンをクリックして、先ほど用意したフォルダを選びます（以上が図 2.9）。

(3) 「次へ (N) >」でプロジェクトの種類を選びます。もちろん「C 言語プロジェクト」です。

(4) 次のダイアログで、スタートアッププログラムを選びます。「◎カスタム」を指定して [...] ボタンをクリックして、先ほど用意したフォルダの中のファイル (ncrt0.a30) を選びます（以上が図 2.10）。

(5) ウィザードの最後で、設定した内容を確認します（図 2.11）。プロジェクト名、ワーキングディレクトリ、スタートアッププログラムを確認します。

(6) 新しいプロジェクトができたら、ソースファイルをこのプロジェクトに登録します。「プロジェクトエディタ」で、目的とする実行オブジェクトのファイル名（この場合には、sec01.x30）を右クリックして、メニューから「アイテムの編集 (A)」→「ファイルの追加 (F)」を選択します。「ファイルを開く」ダイアログで、用意したフォルダの中のソースファイル kadai01.c を開きます（図 2.12）。

(7) ソースファイルがいくつかに分割されているときには、全てのソースファイルを同様に登録します。登録する場所は、kadai01.c と並ぶ sec01.x30 の下です。

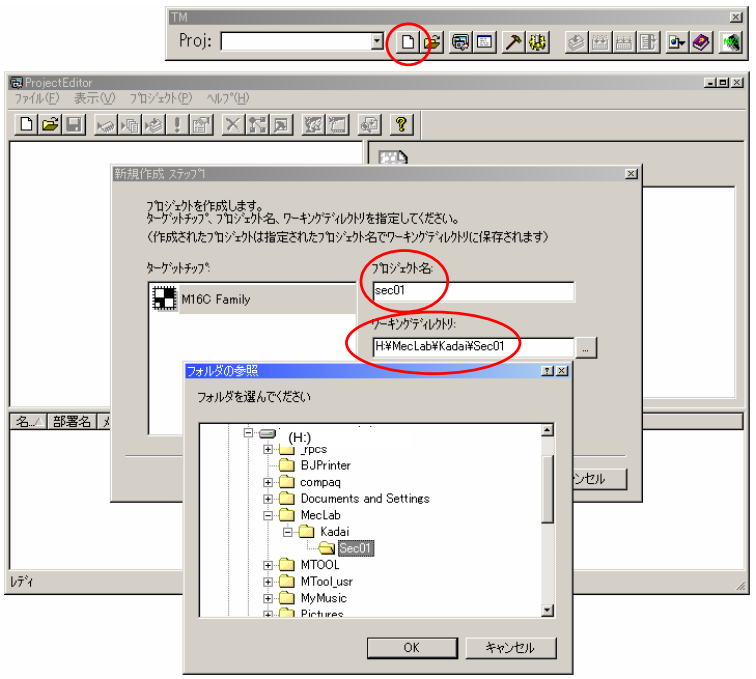


図. 2.9: プロジェクト名とワーキングディレクトリの指定。

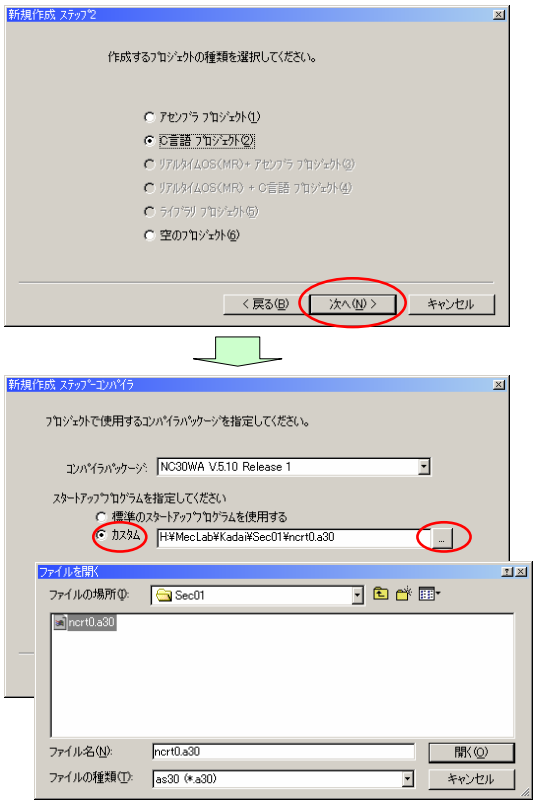


図. 2.10: プロジェクトの種類とスタートアッププログラムの指定。

図. 2.11: 設定した内容の確認。

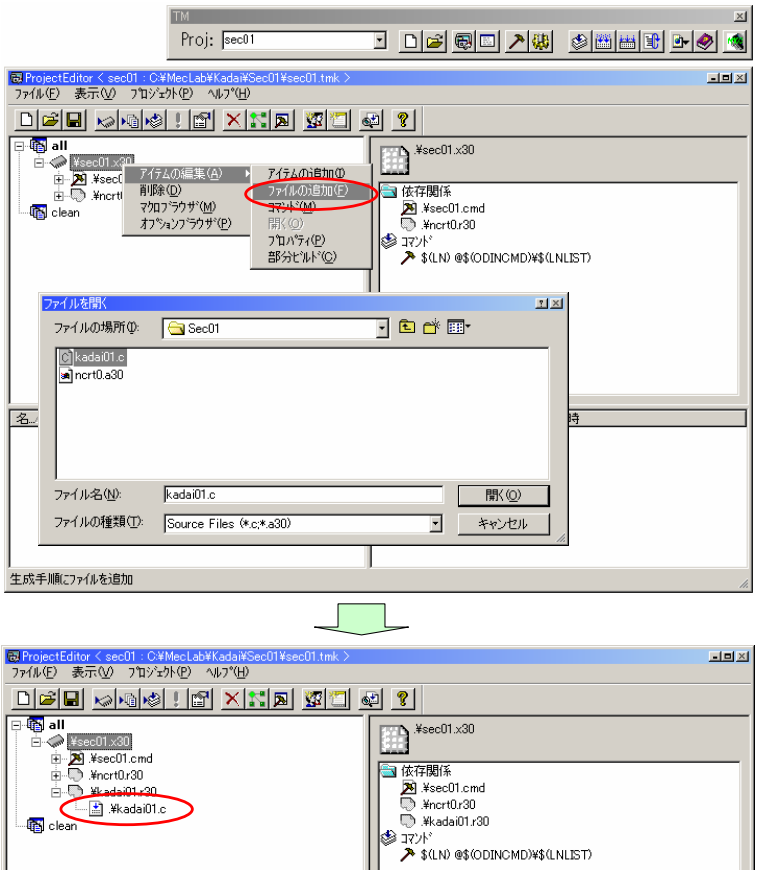


図. 2.12: ソースファイルの登録。

2.6 課題

サンプルプログラムを改造して、次のような動作をするプログラムを作ってください。プッシュスイッチを一定の順序（サンプルプログラムのまま）で押したときに、
トグルスイッチが「↑↓↓↓」ならフルカラー LED が「赤」、
トグルスイッチが「↓↑↓↓」ならフルカラー LED が「緑」、
トグルスイッチが「↓↓↑↓」ならフルカラー LED が「青」、
トグルスイッチが「↓↓↓↑」ならフルカラー LED が「白」、
に点灯する。プッシュスイッチを、別の一定の順序（サンプルプログラムのまま）で押すと消灯する。

ここでは、スイッチの入力とフルカラー LED の出力を試してみます。詳細な仕組みまで理解する必要はありません。今までの C 言語の知識で、スイッチ入力も LED の点灯制御もできることを確認します。

2.6.1 補足説明 – 演算子

関係演算子： 大小関係や等しいかどうかを判定して、真偽値を出力します。真偽値は、関係が正しければ 1（0 以外）、正しくなければ 0 になります。

演算子	意味	例
<	小さい	`if ( a < b )`
<=	小さいか等しい	`if ( a <= b )`
>	大きい	`if ( a > b )`
>=	大きいか等しい	`if ( a >= b )`
==	等しい	`if ( a == b )`
!=	等しくない	`if ( a != b )`

問題： 次の条件文を実行すると、b の値はどうなるでしょうか。

```
if ( a = 10 ) { b = 10; } else { b = 0; }
```

答え： _____

教訓： _____

論理演算子： 真偽値を否定したり、複数の真偽値を組み合わせます。優先順位や評価の順番が決まっていますが、くどいようでも () を指定して、絶対に間違えないようにします。

演算子	意味	例
!	否定	`if ( !a )`
&&	かつ（論理積）	`if ( ( a == b ) && ( c <= d ) )`
||	または（論理和）	`if ( ( a == b ) || ( c <= d ) )`

ビット演算子： ビット単位でデータ操作をします。操作には、論理処理、反転、シフトがあります。操作対象データは、整数だけです。浮動小数点型に対して、ビット演算を行うことはできません。

演算子	意味	例
&	ビットごとの AND	`a = b & 0x7fff`
|	ビットごとの OR	`a = b | 0x8000`
^	ビットごとの XOR	`a = b ^ 0x000f`
～	ビットの反転	`a = ～ a`
<<	左シフト	`a = a << 2`
>>	右シフト	`a = a >> 2`

& (AND) の使い方： &は、特定のビットをマスクする（0 にする）、または、特定のビットを残して残りを全て 0 にするために利用します。

例： 変数 n（1 バイト）のビット 0（最下位ビット）を 0 にする： `n = n & 0xfe`

例： 変数 n（1 バイト）のビット 3（下から 4 番目）だけを残して、後は 0 にする： `n = n & 0x08`

| (OR) の使い方： |は、特定のビットを 1 にするために利用します。

例： 変数 n（1 バイト）のビット 0（最下位ビット）を 1 にする： `n = n | 0x01`

^ (XOR) の使い方： ^は、特定のビットを反転させるために利用します。

例： 変数 n（1 バイト）のビット 0（最下位ビット）を反転する： `n = n ^ 0x01`

例： 変数 n（1 バイト）のビット 0 と 1（下から 2 ビット）を反転する： `n = n ^ 0x03`

<< >>（シフト）の使い方： 変数のビット並びを、左または右にシフトさせるために利用します。シフトする変数は、整数でなければなりません。シフトは一方向に行われて、あふれたビットは消滅します。負の数をシフトした結果は、処理系に依存します（どうなるかわからない）。

例： 変数 n（1 バイト）のビット 1（下から 2 番目）だけを取り出して、それを最下位ビットに移動する（右に 1 ビットシフトする）： `n = ( n & 0x02 ) >> 1`

2.6.2 補足説明 – ビットの指定に 2 進数をそのまま利用する方法

C 言語では、一般的に 10 進数と 16 進数を使います。ところが、スイッチの状態を調べたり、LED を点灯/消灯するときには、2 進数が使えると便利です。ここでは、2 進数をプログラム内で使う方法を説明します。

このために、関数 `strtol()` を使います。この関数は、文字列を long 値に変換するときに、基数指定ができる関数です。

注意： デメリットもあります。標準関数を実行する時間だけ、**プログラムが遅くなります**。16 進数で数値を表すことには、すぐに慣れるはずです。慣れたら、標準関数を使わずに、直接 16 進数を記述するようにします。

【書式】

```
#include <stdlib.h>
long strtol(const char *s, char **endptr, int base);
```

【引数】

`const char *s`：変換対象文字列。

`char **endptr`：変換不可能な文字列へのポインタの格納先（NULL のときには処理しない）。

`int base`：基数。`base=0`：C の定数表記に従う（0 が先頭なら 8 進数、0x または 0X が先頭なら 16 進数。それ以外なら 10 進数）。`base=2`〜`36`：指定する基数（5 なら 5 進数で変換）。

【戻り値】

成功時には、s の値を long で返す。失敗時には 0 を返す。

【使用例】

変数 n（1 バイト）のビット 0 を 0 にする： `n = n & strtol( "11111110", NULL, 2 )`

変数 n（1 バイト）のビット 4 を 1 にする： `n = n | strtol( "00010000", NULL, 2 )`

注意： 正しくは、`n = n | (char)strtol( "00010000", NULL, 2 )` です。この場合には、long から char への暗黙の型変換が行われています。

2.6.3 補足説明 – スイッチの状態を調べる方法

プッシュスイッチを押すと、対応するマイクロコントローラの端子電圧が 0V になり、ソフト側からは値「0」が読めます。押さない状態のときには、値「1」が読めます。

トグルスイッチも同様に、上側（プッシュスイッチ側）に倒すと、対応するマイクロコントローラの端子電圧が 0V になり、ソフト側からは値「0」が読めます。下側に倒すと、値「1」が読めます。

このときに「読み取る」場所が、次章で説明する I/O ポートです。具体的には、次のような if 文で、スイッチの状態を調べることができます。

プッシュスイッチ：

```
if ( ( P1 & strtol( "10000000", NULL, 2 ) ) == 0 ) // プッシュスイッチ黒（左）が押されている
if ( ( P1 & strtol( "10000000", NULL, 2 ) ) != 0 ) // プッシュスイッチ黒が押されていない
                    10000000 = 0x ____
if ( ( P1 & strtol( "01000000", NULL, 2 ) ) == 0 ) // プッシュスイッチ白（中）が押されている
if ( ( P1 & strtol( "01000000", NULL, 2 ) ) != 0 ) // プッシュスイッチ白が押されていない
                    01000000 = 0x ____
if ( ( P1 & strtol( "00100000", NULL, 2 ) ) == 0 ) // プッシュスイッチ赤（右）が押されている
if ( ( P1 & strtol( "00100000", NULL, 2 ) ) != 0 ) // プッシュスイッチ赤が押されていない
                    00100000 = 0x ____
```

トグルスイッチ：

```
if ( ( P10 & strtol( "10000000", NULL, 2 ) ) == 0 ) // トグルスイッチ D（最左）が上側
if ( ( P10 & strtol( "10000000", NULL, 2 ) ) != 0 ) // トグルスイッチ D が下側
if ( ( P10 & strtol( "01000000", NULL, 2 ) ) == 0 ) // トグルスイッチ C が上側
if ( ( P10 & strtol( "01000000", NULL, 2 ) ) != 0 ) // トグルスイッチ C が下側
if ( ( P10 & strtol( "00100000", NULL, 2 ) ) == 0 ) // トグルスイッチ B が上側
if ( ( P10 & strtol( "00100000", NULL, 2 ) ) != 0 ) // トグルスイッチ B が下側
if ( ( P10 & strtol( "00010000", NULL, 2 ) ) == 0 ) // トグルスイッチ A（最右）が上側
if ( ( P10 & strtol( "00010000", NULL, 2 ) ) != 0 ) // トグルスイッチ A が下側
```

2.6.4 補足説明 – フルカラー LED を思い通りの色で点灯させる方法

フルカラー LED のなかには R（赤）、G（緑）、B（青）の 3 種類の LED が入っています。この 3 種類の LED は、個別に点灯/消灯を制御できます。各 LED に接続したマイクロコントローラの端子に、5V を出力すると消灯、0V を出力すると点灯します。電圧を出力すると消灯するのは回路上の都合ですが、ソフトウェア上は値「1」を書いたときに点灯するようにすると、何かと都合がよいものです。そこで、サンプルプログラムでは、ビット反転演算子を使っています。

```
P7 = ～( strtol( "00000001", NULL, 2 ) ); // 赤
P7 = ～( strtol( "00000100", NULL, 2 ) ); //     緑
P7 = ～( strtol( "00010000", NULL, 2 ) ); //         青
P7 = ～( strtol( "00000101", NULL, 2 ) ); // 赤＋緑     ＝黄
P7 = ～( strtol( "00010100", NULL, 2 ) ); //     緑＋青＝水
P7 = ～( strtol( "00010001", NULL, 2 ) ); // 赤   ＋   青＝紫
P7 = ～( strtol( "00010101", NULL, 2 ) ); // 赤＋緑＋青＝白
```

3 I/O ポート

この章の目次：

3.1 目的

スイッチの状態をプログラムから調べることや、プログラムで LED を点灯/消灯させるために、多くの場合、I/O ポートを使います。I/O ポートは、マイクロコントローラが外部と接続するための代表的な手段です。今回は、プッシュスイッチを押した回数を正確に LCD に表示することを目標にします。

ここでは、まず I/O ポートの仕組みと設定方法を少し詳しく説明します。次に、メカトロニクスラボ専用の拡張基板に搭載されている、LCD モジュールとの接続方法とプログラミング方法を説明します。この LCD モジュールは大変便利です。課題でわずかなプログラムを作ることで、C 言語の標準的な書式付き表示関数である `sprintf()` が使えるようになります。

3.2 サンプルプログラムのコンパイルとダウンロード

まず、共有フォルダ（`T:`ドライブ）にある `sample` フォルダを、フォルダごと各自の `H:`ドライブのルート直下にコピーします。（既に前々回コピーしているはずです。）

注意：マイドキュメントやデスクトップにコピーしてはいけません。

(0) マイクロコントローラ基板を拡張基板にセットしてから、拡張基板に電源と RS-232C ケーブルを接続して、電源を入れます。

(1) TM を起動します。

3.2.1 サンプル 1

(2) 「プロジェクトを開く」ボタンをクリックして H:￥smaple￥sec02￥sample1.tmk を開きます。

(3) 「リビルド」ボタンをクリックして、全てコンパイルします。時々、「製品版を購入しなさい」という勧告メッセージが表示されて動作が停止しますが、そのまま待ちます。コンパイルが終了すると、Builder ウィンドウに、「******** Finish... 」と出ます。

(4) 「デバッガの起動」ボタンをクリックして、リモートデバッガ (KD30) を起動します。

注意：起動しないこともあります。そのときは、「ツールの登録」ボタンをクリックして「DEBUG TOOL」の「KD30」をチェックしてから、「デバッガの起動」ボタンをクリックします。

(5) リモートデバッガの [File]-[Download]-[Load Module...] で、H:￥smaple￥sec02￥sample1.x30 を開きます。

(6) リモートデバッガの「↓ Go」ボタンをクリックします。

最初のサンプルプログラムは、プッシュスイッチを押すと、押している間だけ対応する LED が点灯します。同時に複数のプッシュスイッチを押せば、対応する色の加法混色が見えます。

このサンプルプログラムの目的は、プッシュスイッチとフルカラー LED が正常に機能していることを確認することです。

```
/*====================================================================*/
/*    ファイル名：    0201.c                                          */
/*    内容：          LED 点灯（ボタンに応じてフルカラー LED を点灯）  */
/*====================================================================*/

#pragma ADDRESS P1    3e1H    /* ポート P1 レジスタ */
#pragma ADDRESS PD1   3e3H    /* ポート P1 方向レジスタ*/
                        /* 0：入力モード, 1：出力モード。bit0-4 は 0 書込 */
#pragma ADDRESS P7    3edH    /* ポート P7 レジスタ */
#pragma ADDRESS PD7   3efH    /* ポート P7 方向レジスタ*/
                        /* 0：入力モード, 1：出力モード */

unsigned char P1, PD1, P7, PD7;

main() {
    PD1 = 0x00;               /* ポート 1 方向レジスタ  入力  */
    PD7 = 0xff;               /* ポート 7 方向レジスタ  出力  */
    P7 = ~0x00;               /* ポート 7 出力 L(LED 消灯)  */

    while ( 1 ) {
        if ( (P1 & 0x20) == 0 ) {     /* P1 の bit5 が 0 のとき = 赤押の時 */
            P7 = ~(~P7 | 0x10);       /* P7 の bit4 を立てる = フルカラー LED の青色点灯 */
        } else {
            P7 = ~(~P7 & 0xef);       /* P7 の bit4 をマスク = フルカラー LED の青色消灯 */
        }

        if ( (P1 & 0x40) == 0 ) {     /* P1 の bit6 が 0 のとき = 白押の時 */
            P7 = ~(~P7 | 0x04);       /* P7 の bit2 を立てる = フルカラー LED の緑色点灯 */
        } else {
            P7 = ~(~P7 & 0xfb);       /* P7 の bit2 をマスク = フルカラー LED の緑色消灯 */
        }

        if ( (P1 & 0x80) == 0 ) {     /* P1 の bit7 が 0 のとき = 黒押の時 */
            P7 = ~(~P7 | 0x01);       /* P7 の bit0 を立てる = フルカラー LED の赤色点灯 */
        } else {
            P7 = ~(~P7 & 0xfe);       /* P7 の bit0 をマスク = フルカラー LED の赤色消灯 */
        }
```

```
    }
}
```

3.2.2 サンプル 2

(2) 「プロジェクトを開く」ボタンをクリックして H:￥smaple￥sec02￥sample2.tmk を開きます。

(3) 「リビルド」ボタンをクリックして、全てコンパイルします。時々、「製品版を購入しなさい」という勧告メッセージが表示されて動作が停止しますが、そのまま待ちます。コンパイルが終了すると、Builder ウィンドウに、「******** Finish... 」と出ます。

(4) 「デバッガの起動」ボタンをクリックして、リモートデバッガ (KD30) を起動します。

注意：起動しないこともあります。そのときは、「ツールの登録」ボタンをクリックして「DEBUG TOOL」の「KD30」をチェックしてから、「デバッガの起動」ボタンをクリックします。

(5) リモートデバッガの [File]-[Download]-[Load Module...] で、H:￥smaple￥sec02￥sample2.x30 を開きます。

(6) リモートデバッガの「↓ Go」ボタンをクリックします。

次のプログラムは、プッシュスイッチを押すごとに、フルカラー LED の対応する色が反転するように、と思って書いたものです。

ソースリストを見てみましょう。以前に読み取ったポートの内容をメモリに保存しておき (memP1)、プッシュスイッチに対応するビットが、前回H（押していない）で今回L（押している）のときに限って対応する LED の出力を反転します。ロジックには、全く変なところがありません。プッシュスイッチや LED は、サンプルプログラム 1 で正常に動作していることを確認してあります。何が悪いのでしょうか。

```
/*=========================================================*/
/*    ファイル名：    0202.c                               */
/*    内容：          フルカラー LED 点灯（トグル）        */
/*=========================================================*/

#pragma ADDRESS P1    3e1H    /* ポート P1 レジスタ */
#pragma ADDRESS PD1    3e3H    /* ポート P1 方向レジスタ*/
                          /* 0：入力モード, 1：出力モード。bit0-4 は 0 書込 */
#pragma ADDRESS P7    3edH    /* ポート P7 レジスタ */
#pragma ADDRESS PD7    3efH    /* ポート P7 方向レジスタ*/
                          /* 0：入力モード, 1：出力モード */

unsigned char P1, PD1, P7, PD7;

main() {
    int i;
    unsigned char memP1 = 0xff;

    PD1 = 0x00;                /* ポート 1 方向レジスタ  入力  */
    PD7 = 0xff;                /* ポート 7 方向レジスタ  出力  */
    P7 = ~0x00;                /* ポート 7 出力 L(LED 消灯)  */

    while ( 1 ) {
        if ( ((memP1 & 0x20) != 0) && ((P1 & 0x20) == 0) ) {
                          /* P1 の bit5 が前回 H、今回 L -> 赤押し */
            P7 ^= 0x10;        /* P7 の bit4 を反転 = フルカラー LED の青色反転 */
        }
```

```
        if ( ((memP1 & 0x40) != 0) && ((P1 & 0x40) == 0) ) {
                            /* P1 の bit6 が前回 H、今回 L -> 白押し */
            P7 ^= 0x04;         /* P7 の bit2 を反転 = フルカラー LED の緑色反転 */
        }

        if ( ((memP1 & 0x80) != 0) && ((P1 & 0x80) == 0) ) {
                            /* P1 の bit7 が前回 H、今回 L -> 黒押し */
            P7 ^= 0x01;         /* P7 の bit0 を反転 = フルカラー LED の赤色反転 */
        }
        memP1 = P1;
    }
}
```

3.3 スイッチ入力とチャタリング

3.3.1 スイッチの構造例

トグルスイッチ： レバーの操作によって電気接点が移動して、接触する端子を切り替えます。図 3.1（左）の例では、中央が共通端子で、レバーを右側に倒すと左の端子と、左側に倒すと右の端子と接触します。レバーがバネで押し戻されて、手を離すと必ずレバーが戻るスイッチもあります。

プッシュスイッチ： 押しボタンを押すと電気接点が移動して、接触する端子を切り替えます。図 3.1（右）の例では、左側が共通端子で、押しボタンを押すことで右側端子から中央端子に接触が切り替わります[21] 。一度押すと一方の端子と接触して、もう一度押すと他方の端子と接触するスイッチ[22] もあります。

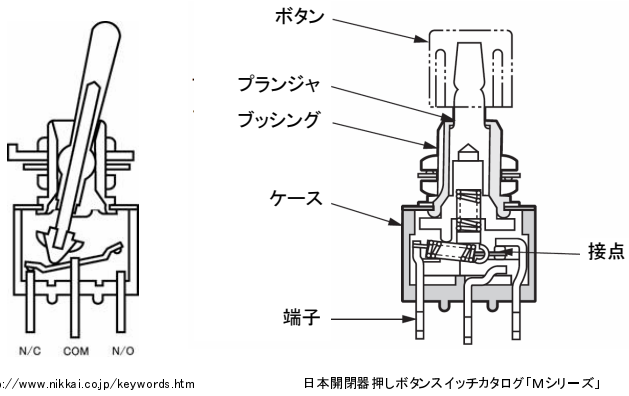


図. 3.1: トグルスイッチ（左）とプッシュスイッチ（右）の断面図の例。

3.3.2 スイッチ入力波形

スイッチは機械機構で構成されていて、接点の接触は単純な ON-OFF ではありません。例えばトグルスイッチでは、中央端子に接触した切片が接触状態を切り替えるときに振動して、何回も接触と非接触を繰り返します。

この不安定な接触状態をチャタリングといいます。チャタリング期間は、人間の操作方法、スイッチの構造、スイッチの個体差（製品特性のばらつき）、経年変化などのよって変化して一定ではありません。短いもので数ミリ秒、長いもので数十ミリ秒程度です。チャタリング期間中の断続周波数も一定ではありません。

[21] 押している間だけ接触が切り替わって、離すと元に戻る動作を、モーメンタリ動作という。
[22] オルタネート動作という。

マイクロコントローラの入力にスイッチを使うときや、電気回路の動作をスイッチで切り替えるときには、チャタリングがあることを考慮する必要があります。例えば、ベルトコンベアを通過する製品の個数を数えるときに、スイッチを直接カウンタに接続したら、まず間違いなく正確な個数を数えることはできません。単純にスイッチで電気回路を切り替えると、切り替えが断続することによる誤動作や破損の注意する必要があります。

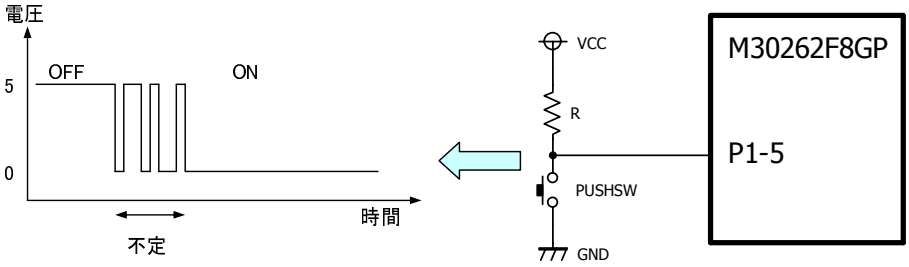


図. 3.2: オープン状態からクローズ状態にスイッチを変化させたときの入力信号の例。

3.3.3 チャタリング対策

チャタリングというものがあるということがわかっていれば、対策は簡単です。次に、代表的な対策を示します。マイクロコントローラなど、ソフトウェアでスイッチ入力を扱う場合には、**一定時間間隔でサンプリング方式**が多く利用されています。

付加回路方式： マイクロコントローラを使わない電気回路だけで、システムを構成するときに利用します。チャタリング波形を LPF(Low Pass Filter) でなまらせて、なまった波形をシュミットトリガ（ヒステリシス付きの入力バッファ）で成形する方法です。波形成形するときにヒステリシスがないと、結局「ヒゲ」が残ることがあるので注意します。

一定時間後に再度確認方式： スイッチ入力を監視して、変化したら一定時間後にもう一度そのスイッチ入力を確認します。例えば、スイッチ入力が「1」から「0」に変化したことを調べるには、「1 から 0 への変化」がないかどうか、常にチェックします。「1 から 0 への変化」を検出したら、一定の時間待ちます。再びスイッチ入力をチェックして、「0」だったら「これは本物だ」と思うわけです。

マイクロコントローラを使う場合に、ポート割り込み（入力ポートの信号が変化したことで割り込みをかける方式）が利用できるときには、この方法を使うこともあります。

一定時間間隔でサンプリング方式： チャタリング継続期間よりも十分に長く、しかも人間が操作する時間に対して十分に短い時間間隔で、スイッチの状態を調べる方法です。スイッチの状態をサンプリングする時間間隔は、スイッチの特性やシステムの目的に応じて決定します。例えば、1/100[秒] 単位のストップウォッチのスタート/ストップスイッチに対して、100[msec] のサンプリング周期（1/10[秒]）では、1/100 位の数値の意味がなくなります。一般的には 10[msec] 程度が多く使われるようですが、実際にテストして決定します。

3.4 マイコンで使える I/O ポート

プログラマブル入出力ポートは、P15〜P17（3 本）、P6（8 本）、P7（8 本）、P80〜P83、P85〜P87（7 本）、P90〜P93（4 本）、P10（8 本）で、合計 38 本あります。

各ポートの入出力は、方向レジスタによって 1 本ごとに設定できます。また、4 本単位でプルアップ[23] す

[23]抵抗でポートと電源を接続すること。

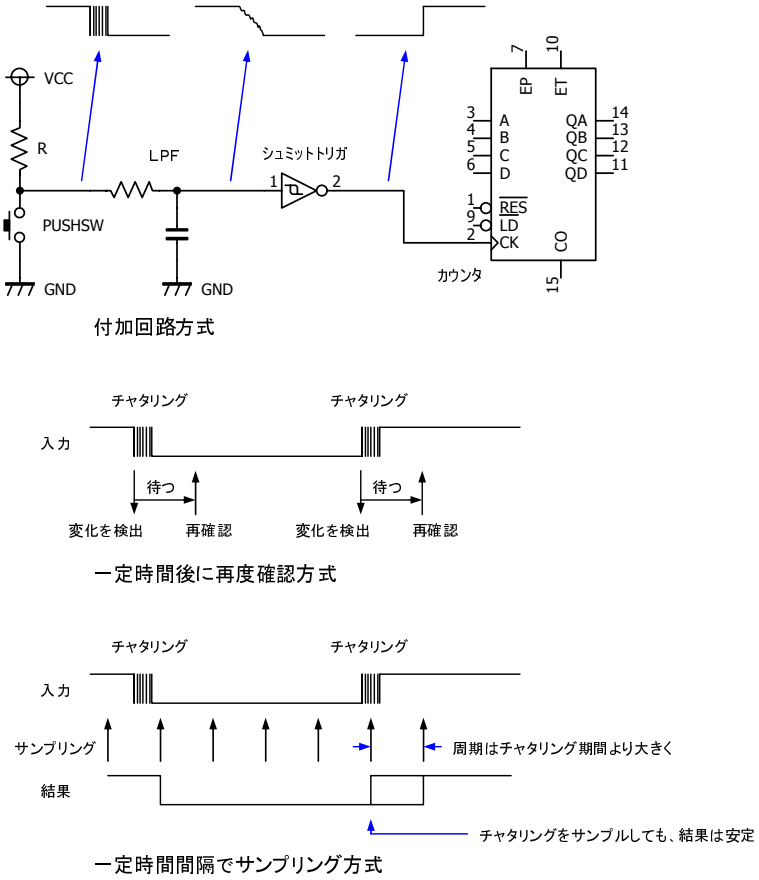


図. 3.3: チャタリング対策。

るかしないかを選択できます。プルアップをする目的は、スイッチ入力などの入力ポートとして使うときに、外付け部品数を減らすことです。

ポートによっては、他の機能と兼用になっていることがあるので注意してください。例えば、P10 はアナログ入力と兼用です。同じピンで両方の機能を使うことは、通常はできません。

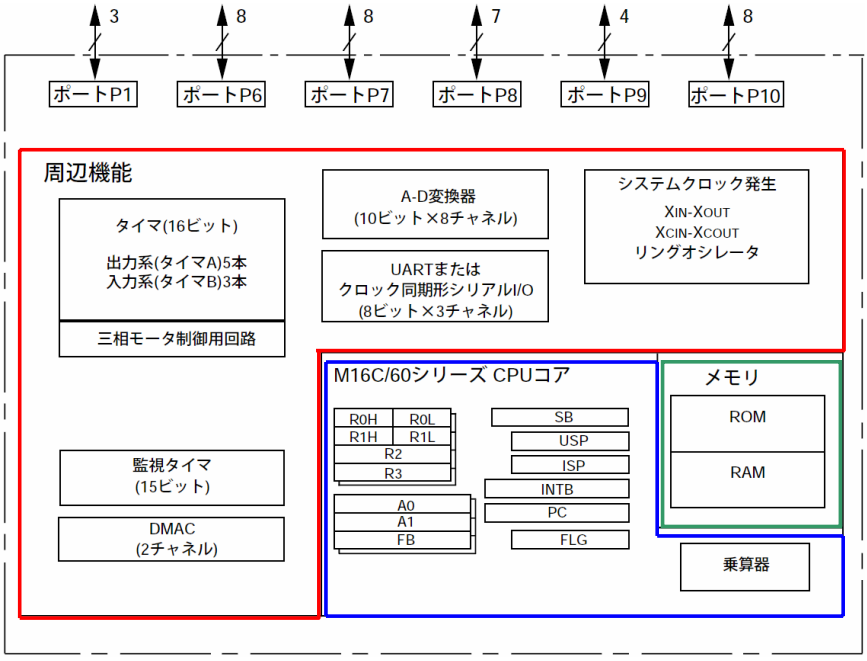
ポートアドレスの一覧は、『ハードウェアマニュアル』(`rjj09B0033_m16chm.pdf`) 10〜15 ページ参照。

詳細な内容は、『ハードウェアマニュアル』(`rjj09B0033_m16chm.pdf`) 162 ページ以後を参照。

3.4.1 I/O ポートの仕組み

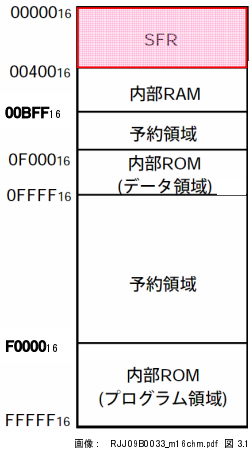
I/O ポートは、特定のアドレスにデータを書き込むことで、機能（入出力方向やプルアップをする/しない）や出力ピンの出力電圧を決定できます。また、特定のアドレスからデータを読み込むことで、入力ピンの電圧状態を調べることができます。この「特定のアドレス」は、「SFR(Special Function Register?)」としてアドレス一覧表があります（『ハードウェアマニュアル』(`rjj09B0033_m16chm.pdf`) 10〜15 ページ）。

まず、図 3.5 で、青の経路で示した入力ポートの動作を説明します。入力ポートとして機能が指定されると、赤で示した出力ポートとして利用する回路は動作していません。マイクロコントローラのピンに入力された電圧は、バッファを通してマイクロコントローラ内部のデータバスに接続されています。このバッファは、I/O ポートのデータを読むときに指定する「特別なアドレス」を指定したときだけ、出力がデータバスに接続されます。マイクロコントローラ（の中の CPU コア）は、メモリの値を読むときと同じように、データバス上の信号を読み取ることでピンに入力された電圧をデータとして読むことができます。



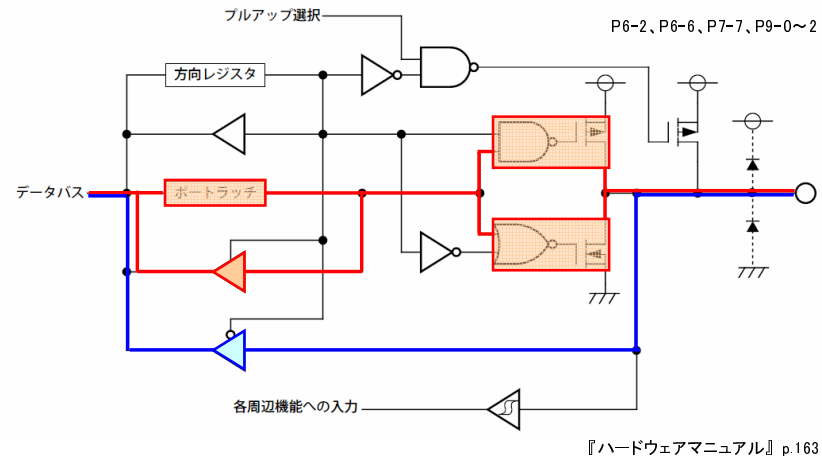
画像： RJJ09B0033_m16chm.pdf 図 1.1

次に、赤の経路で示した出力ポートの動作を説明します。出力ポートとして機能を設定したときには、I/O ポートのデータを読むことも書くこともできます。書き込むと、「ポートラッチ」にデータが書き込まれます。読み込むと、このポートラッチに書き込まれたデータを読み出すことができます。ポートラッチに書き込まれたデータは、出力バッファを通してマイクロコントローラのピンの電圧を決定します。



画像： RJJ09B0033_m16chm.pdf 図 3.1

図. 3.4: アドレス空間。

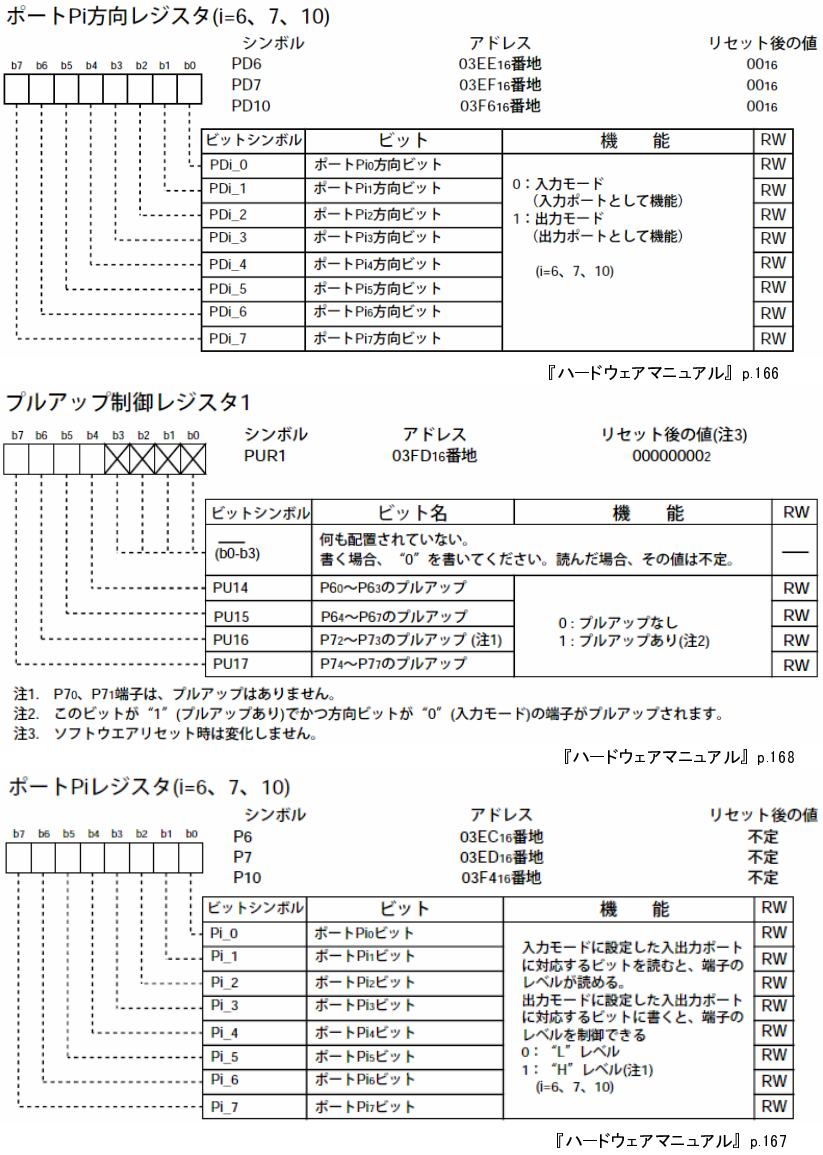


『ハードウェアマニュアル』p.163

図. 3.5: I/O ポート 1 ビット分の回路ブロック図。青で示した経路は入力、赤で示した経路は出力を表す。

3.4.2 I/O ポートの設定方法

I/O ポートの機能を設定するためには、まず設定したいポート番号に対応した SFR のアドレスを調べて、ソースプログラムに記述します。次に、入出力方向を指定します。必要に応じて、プルアップ抵抗を接続します。後は、ポートレジスタを普通のメモリと同じように読んだり書いたりします。



ポートPi方向レジスタ(i=6、7、10)

シンボル	アドレス	リセット後の値
PD6	03EE16番地	0016
PD7	03EF16番地	0016
PD10	03F616番地	0016

ビットシンボル	ビット	機　能	RW
PDi_0	ポートPi0方向ビット	0：入力モード（入力ポートとして機能） 1：出力モード（出力ポートとして機能） (i=6、7、10)	RW
PDi_1	ポートPi1方向ビット		RW
PDi_2	ポートPi2方向ビット		RW
PDi_3	ポートPi3方向ビット		RW
PDi_4	ポートPi4方向ビット		RW
PDi_5	ポートPi5方向ビット		RW
PDi_6	ポートPi6方向ビット		RW
PDi_7	ポートPi7方向ビット		RW

『ハードウェアマニュアル』p.166

プルアップ制御レジスタ1

シンボル	アドレス	リセット後の値(注3)
PUR1	03FD16番地	000000002

ビットシンボル	ビット名	機　能	RW
―― (b0-b3)	何も配置されていない。書く場合、“0”を書いてください。読んだ場合、その値は不定。		――
PU14	P60～P63のプルアップ	0：プルアップなし 1：プルアップあり(注2)	RW
PU15	P64～P67のプルアップ		RW
PU16	P72～P73のプルアップ (注1)		RW
PU17	P74～P77のプルアップ		RW

注1. P70、P71端子は、プルアップはありません。
注2. このビットが“1”(プルアップあり)でかつ方向ビットが“0”(入力モード)の端子がプルアップされます。
注3. ソフトウエアリセット時は変化しません。

『ハードウェアマニュアル』p.168

ポートPiレジスタ(i=6、7、10)

シンボル	アドレス	リセット後の値
P6	03EC16番地	不定
P7	03ED16番地	不定
P10	03F416番地	不定

ビットシンボル	ビット	機　能	RW
Pi_0	ポートPi0ビット	入力モードに設定した入出力ポートに対応するビットを読むと、端子のレベルが読める。 出力モードに設定した入出力ポートに対応するビットに書くと、端子のレベルを制御できる 0：“L”レベル 1：“H”レベル(注1) (i=6、7、10)	RW
Pi_1	ポートPi1ビット		RW
Pi_2	ポートPi2ビット		RW
Pi_3	ポートPi3ビット		RW
Pi_4	ポートPi4ビット		RW
Pi_5	ポートPi5ビット		RW
Pi_6	ポートPi6ビット		RW
Pi_7	ポートPi7ビット		RW

『ハードウェアマニュアル』p.167

図. 3.6: I/O ポートを設定するためのレジスタ。

設定例：

SFR アドレスとポートシンボルの定義：

```
#pragma ADDRESS P1  3e1H  /* ポート P1 レジスタ */
#pragma ADDRESS PD1 3e3H  /* ポート P1 方向レジスタ */
#pragma ADDRESS P7  3edH  /* ポート P7 レジスタ */
#pragma ADDRESS PD7 3efH  /* ポート P7 方向レジスタ */
unsigned char P1, PD1, P7, PD7;
```

入出力の設定と、実際の入力と出力：

```
my_function() {
    unsigned char sampled_P1;
    PD1 = 0x00;             /* ポート 1 方向レジスタ：入力 */
    PD7 = 0xff;             /* ポート 7 方向レジスタ：出力 */
    sampled_P1 = P1;        /* ポート 1 のデータを変数にコピー */
    P7 = ~0x00;             /* ポート 7 出力（LED 消灯）*/
}
```

3.4.3 P9 を出力ポートに設定するときの注意

ポート 9 はタイマ入力ピンを兼ねているため、プログラムの暴走などでポートの入出力方向が簡単に書き換えられることがないように、PD9（ポート 9 方向レジスタ）が保護されています。PD9 を設定するためには、プロテクトレジスタ PRCR の特定のビットをセット（「1」を書くこと）した直後の命令で PD9 を設定します。

詳細は、『ハードウェアマニュアル』(rjj09B0033_m16chm.pdf) 45 ページを参照してください。

3.4.4 デバッガで入力ポートの状態を調べる

プログラムが思うように動作しないときには、何が原因なのかをはっきりさせる必要があります。スイッチとの接続配線が断線していたら、プログラムでどんなにがんばってもスイッチ入力を読み取ることはできません。デバッガには、マイクロコントローラ内のメモリを表示する機能があります。I/O ポートの状態も、この機能を使って表示できます。

(1) サンプルプログラムを一度スタートしてから（[Go] をクリック）、停止します（[Stop] をクリック）。この状態で、プッシュスイッチが接続されている I/O ポートが入力に設定されました。

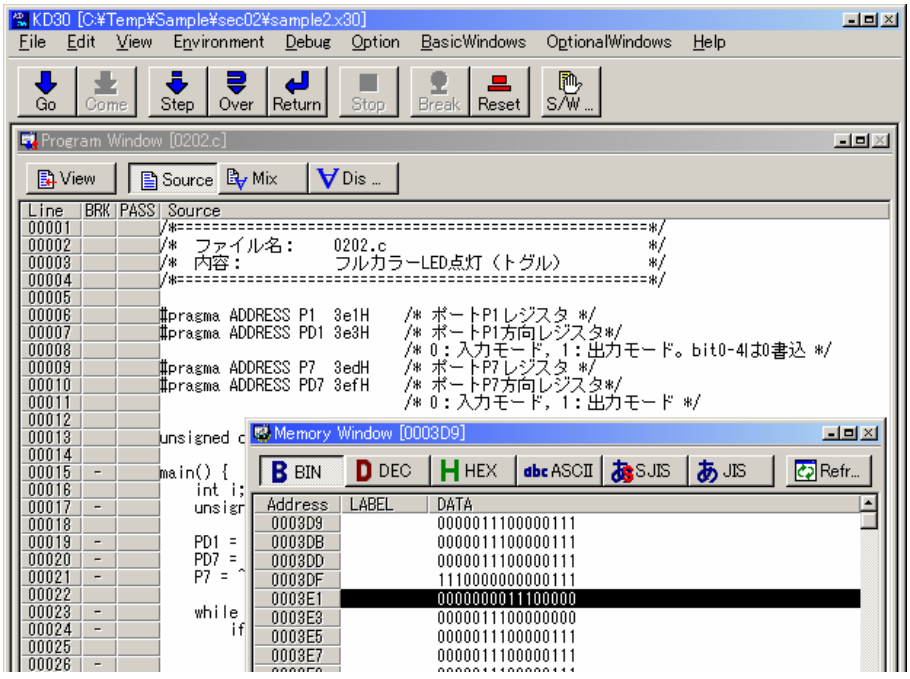


図. 3.7: デバッガの Memory Window で I/O ポートの状態を表示。

(2) メニューの [BasicWindows]-[Memory Window] で、メモリ内容を表示するためのウィンドウを表示します。2 進数表示モード（[BIN] をクリック）にすると、各ビットの変化がわかりやすくなります。

(3) プッシュスイッチは、P1-5（赤）、P1-6（白）、P1-7（黒）に接続しています。ポート 1 は、アドレス `0x3e1` のデータとして読めます。このアドレスが表示できるように、Memory Window のスクロールバーを調整します。（`0x3e0` と `0x3e1` が、1 行に表示されます。）

(4) Memory Window の [Referesh] をクリックすると、最新の I/O ポートの状態を表示します。プッシュスイッチを押しながら [Referesh] をクリックすると、bit5〜7 が押せば「0」、離せば「1」に変化します。

3.5 LCD を使おう

ここでは、マイクロコントローラや、マイクロコントローラに接続したセンサなどの出力値を表示するために、LCD（Lquid Crystal Display; 液晶ディスプレイ）を使う方法を説明します。最終的には、C 言語標準ライブラリに含まれる `sprintf()` 関数を使って、「書式付きで変数を表示」できるようにします。

3.5.1 LCD モジュール

LCD モジュールには、16 文字× 2 行の表示が可能な液晶パネルと、専用の IC [24] が搭載されています。

専用の IC のインタフェース端子をマイクロコントローラの I/O ポートに接続して、マイクロコントローラからデータを送ることで、LCD に指定したキャラクタを表示できます。インタフェース端子は、4 ビットまたは 8 ビット構成のデータバス、データバスの方向を指定する R/W 信号、専用 IC 内のレジスタを選択する RS 信号、データバス上の信号が有効であることを示す E 信号で構成されています。

他に、もちろん電源があります。メカトロニクスラボ専用の拡張基板上の LCD モジュール近くの VR（可変抵抗）は、液晶パネルに印可する電圧を調整するものです。この VR によって、表示コントラストが変化します。

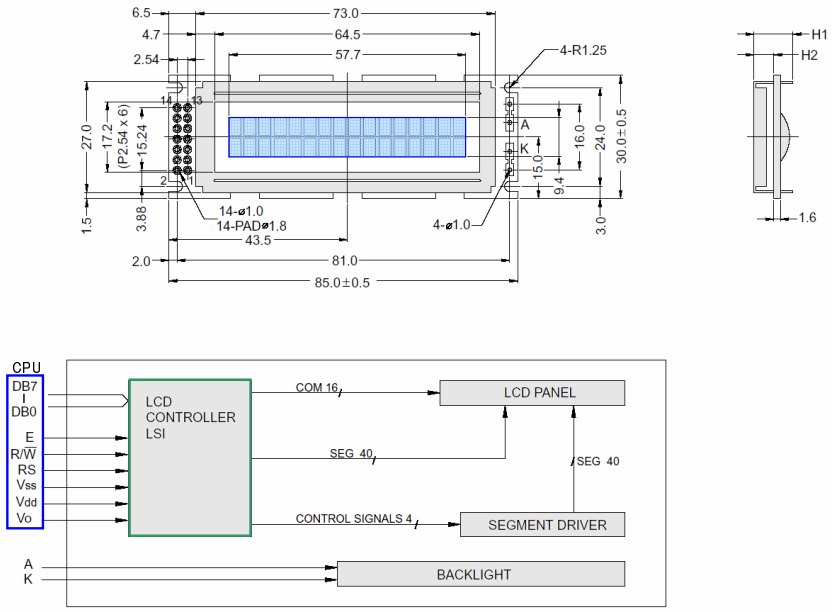


図. 3.8: 液晶モジュールとブロック図。

[24] 日立 HD44780U 相当品。IC のデータシートを探すときには、`http://www/alldatasheet.com` が便利。ただし英語。

3.5.2 接続したポート

専用の IC との接続は、4 ビットまたは 8 ビット構成のデータバスが必要です。データバスを 8 ビット構成にすると、LCD との接続だけで 11 本もポートを使ってしまいます。そこで、8 ビットのデータを 2 回に分けて送受信するモードを使い、データバスの 4 ビットをマイクロコントローラのポートに接続します。接続したポートは、表の通りです。

LCD モジュールにコマンドやデータを送出するときと、LCD モジュールの状態を読み出すときには、データバスの方向を変える必要があります。データバスはポート 9 に接続されているので、データバスの方向を変えるときには、プロテクトレジスタの特定ビットをセットして、直後の命令で PD9 を設定するようにします。

CPU	信号名	CN1	CN2	MC 拡張基板	MC リモコン基板
12	P8-5/NMI*/SD*	1		LCD E	
6	P8-6/Xcout		46	LCD R/W	
5	P8-7/Xcin		45	LCD RS	
3	P9-0/TB0in		44	LCD DB4	
2	P9-1/TB1in		43	LCD DB5	
1	P9-2/TB2in		42	LCD DB6	
48	P9-3		41	LCD DB7	

3.5.3 コマンドの例

以下は、CD-ROM の中にある LCD モジュールの説明書と、オークス電子株式会社のホームページ http://www.oaks-ele.com/oaks8/oaks8-manual.htm からダウンロードできる LCD モジュールの解説 LCD_16x2.pdf からの抜粋です。

初期化 LCD_init()：

・マイクロコントローラの LCD 接続ポートを出力に設定する。
・RS 信号を 0 にする。
・15[msec] 待つ。
・D3〜D0 に 0x03 を出力する。
・E 信号を 1 にして 0 にする。
・5[msec] 待つ。
・E 信号を 1 にして 0 にする。
・0.1[msec] 待つ。
・E 信号を 1 にして 0 にする。
・5[msec] 待つ。
・D3〜D0 に 0x02 を出力する。（4 ビットインターフェース設定）
・E 信号を 1 にして 0 にする。
・0.04[msec] 待つ。
・0x28 の上位 4 ビット (0b0010) を D3〜D0 に出力する。（0x28 = 00101000：_____）
・E 信号を 1 にして 0 にする。
・0x28 の下位 4 ビット (0b1000) を D3〜D0 に出力する。
・E 信号を 1 にして 0 にする。
・0.04[msec] 待つ（サンプルプログラムではステータスをチェック）。

4 ビットインターフェースモードでは、実際に必要な 8 ビットデータを 4 ビットデータ 2 回に分けて書き込みます。上位 4 ビットを最初に書き込みます。

データ書き込み `lcdwrite1data()`：

・RS 信号を 1 にする。
・書き込みデータの上位 4 ビットを D3～D0 に出力する。
・E 信号を 1 にして 0 にする。
・書き込みデータの下位 4 ビットを D3～D0 に出力する。
・E 信号を 1 にして 0 にする。
・0.04[msec] 待つ（サンプルプログラムではステータスをチェック）。

次に、インストラクションレジスタに書き込むデータとコマンドとの対応表を示します。
ある位置に文字を表示するときには、DDRAM アドレスに位置を指定します。LCD の左上が 0x00 ですが、左下は 0x40 です（図 3.10）。また、E 信号に対する他の信号の変化タイミングが規定されています。

Instruction	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		実行時間
ClearDisplay	0	0	0	0	0	0	0	0	0	1	画面消去し DDRAM アドレスカウンタを 0	1.52mS
ReturnHome	0	0	0	0	0	0	0	0	1	–	DDRAM アドレスカウンタを 0	1.52mS
EntryModeSet	0	0	0	0	0	0	0	1	I/D	S	カーソル移動モードとシフトモードを設定 **I/D=1:インクリメント** I/D=0:デクリメント S=1：ディスプレイシフト有効	37uS
DisplayOn/Off	0	0	0	0	0	0	1	D	C	B	ディスプレイ ON/OFF とカーソル・ブリンク設定 **D=1:ディスプレイ ON** D=0:ディスプレイ OFF C=1:カーソル ON C=0:カーソル OFF B=1:カーソル位置ブリンク	37uS
CursorShiftControl	0	0	0	0	0	1	S/C	R/L	–	–	ディスプレイシフトモード設定 S/C=1:ディスプレイシフト **S/C=0:カーソル移動** R/L=1:右シフト R/L=0:左シフト	37uS
FunctionSet	0	0	0	0	1	DL	N	F	–	–	インターフェースビット長と表示ライン・キャラクタサイズ設定 DL=1:8 ビット DL=0:4 ビット **N=1:2 ライン** N=0:1 ライン F=1:5x10dot **F=0:5x8dot**	37uS
SetCGRAMadrs	0	0	0	1	ACG						CGRAM アドレス設定	37uS
SetDDRAMadrs	0	0	1	ADD							DDRAM アドレス設定	37uS
ReadBusyFlag	0	1	BF	AC							Busy フラグとアドレスカウンタ読出し BF=1：ビジー状態	0uS
WriteCG/DDRAM	1	0	書き込みデータ								DDRAM/CGRAM データ書込み	37uS
ReadCG/DDRAM	1	1	読み出しデータ								DDRAM/CGRAM データ読出し	37uS

図. 3.9: コマンド一覧。

		桁															
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
行	1	00h	01h	02h	03h	04h	05h	06h	07h	08h	09h	0Ah	0Bh	0Ch	0Dh	0Eh	0Fh
	2	40h	41h	42h	43h	44h	45h	46h	47h	48h	49h	4Ah	4Bh	4Ch	4Dh	4Eh	4Fh

図. 3.10: 表示位置と DDRAM アドレス。

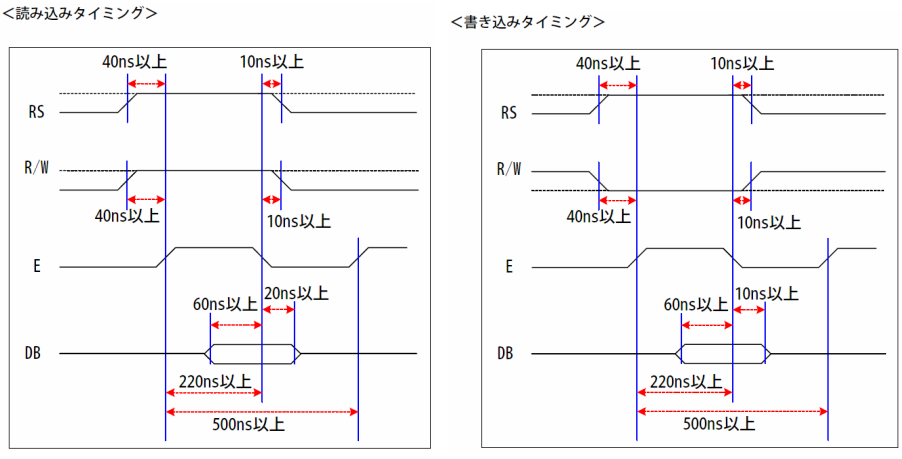


図. 3.11: 液晶モジュールとのインタフェースタイミング。

3.5.4 表示文字コードとパターン

基本的に、ASCII コードに対応する表示パターンが用意されています。CGRAM（キャラクタジェネレータ RAM）に、独自の表示パターンを定義することもできます。

図. 3.12: 表示文字コードと対応する表示パターン。

3.5.5 LCD 表示関数

それでは、実際に LCD に文字を表示してみましょう。

(2) 「プロジェクトを開く」ボタンをクリックして `H:￥smaple￥sec02￥LCD.tmk` を開きます。

(3) 「リビルド」ボタンをクリックして、全てコンパイルします。時々、「製品版を購入しなさい」という勧告メッセージが表示されて動作が停止しますが、そのまま待ちます。コンパイルが終了すると、Builder ウィンドウに、「******** Finish... 」と出ます。

(4) 「デバッガの起動」ボタンをクリックして、リモートデバッガ (KD30) を起動します。

注意：起動しないこともあります。そのときは、「ツールの登録」ボタンをクリックして「DEBUG TOOL」の「KD30」をチェックしてから、「デバッガの起動」ボタンをクリックします。

(5) リモートデバッガの [File]-[Download]-[Load Module...] で、`H:￥smaple￥sec02￥LCD.x30` を開きます。

(6) リモートデバッガの「↓ Go」ボタンをクリックします。

プログラムを起動したら、LCD の表示コントラストを調整します。

`0203.c` の中の `LCD_locate()` や `LCD_print()` を書き換えて、いろいろな文字を表示してみましょう。

注意して見てほしいところ：

・プロジェクトに、複数のソースファイルがある。（必ず確認すること）

・汎用に使える関数だけが記述してあるファイル (LCDfunc.c) がある。
このような記述方法やファイル分割ができるので、複数のプログラマによる共同開発が可能になります。ただし、最初に関数の仕様を明確に決めておく必要があります。

・ヘッダには、汎用に使える関数のプロトタイプが記述してある。
他の共同開発者には使わせたくない、内部使用だけにとどめたい、そのような関数はヘッダとして公開しなければよいわけです。ただし、C 言語でも C++言語でも、よりスマートな方法が用意されています。

確認：

`#include <stdio.h>// sprintf()` を使うため

と

`#include "LCDfunc.h" // LCD` 表示関数をインクルード

で<>と""の違いは？　答え：_____

LCD 表示関数の利用例。`main()` 関数内で `buff[64]` を確保するとスタック領域が不足するため、グローバル変数領域に確保しています。

```
//=========================================================
//     内容： LCD 表示テスト
// 0203.c
//=========================================================
#include <stdio.h>                    // sprintf() を使うため
#include "LCDfunc.h"                 // LCD 表示関数をインクルード
char buff[64];                        // グローバル変数にバッファを確保

void main(void) {
    int ad_result;

    LCD_init();                       // LCD 初期設定
    LCD_cls();                       // 全て消去

    LCD_locate( 10, 0, 1, 1 );       // カーソルを (10,1) に、カーソル ON、点滅
    LCD_print( 'A' );                // キャラクタを表示
    LCD_print( 'D' );
    LCD_print( '1' );
    LCD_print( '=' );
    LCD_print( '?' );
    LCD_print( '?' );

    ad_result = 127;
    sprintf( buff, "AD1=%d", ad_result );
    LCD_print_str( buff );

    while( 1 );                       // ここで停止
}
```

LCD 表示関数を利用するときにインクルードするヘッダファイル。

```
//=========================================================
// LCDfunc.h
// LCD 表示関数の実装
//=========================================================
```

```
void LCD_init( void );                    // LCD 初期設定
void LCD_cls( void );                     // 全て消去
void LCD_off( void );                     // 表示 OFF
void LCD_on( void );                      // 表示 ON
void LCD_locate( unsigned char x, unsigned char y,
                unsigned char cursor, unsigned char blink );
                                          // 表示位置を指定
                                          // cursor=1 のとき、カーソルを表示
                                          // blink=1 のとき、カーソル位置キャラクタを点滅
void LCD_print( unsigned char c );         // キャラクタを 1 文字出力
void LCD_print_str( unsigned char str[] );
                                          // (0,0) から文字列を出力
```

LCD 表示関数。ただし、関数内部で使用している下位関数の一部は省略している（ページ数が多くなるため。サンプルソースコードには、ちゃんと入っている）。

```
//=========================================================
//     LCD 表示関数
// LCDfunc.c
//=========================================================
#include    "sfr26.h"                 // OAKS16 用定義ファイル
#include    "LCDfunc.h"                // LCD 表示関数

#define PORTIN        0x00              // ポート方向レジスタを入力に設定
#define    PORTOUT        0xff             // ポート方向レジスタを出力に設定
#define    LCDRS        p8_7              // RS 端子 */
#define    LCDRW        p8_6              // RW 端子 */
#define    LCDE        p8_5               // E 端子 */
#define    HIGH        1                  // 端子出力“ H ”
#define    LOW            0                 // 端子出力“ L ”
#define    LCD_COMMAND    0                 // RS-command 指定
#define    LCD_DATA    1                  // RS-data 指定
#define    BUSY        1                  // LCD 書き込み busy
#define    NOBUSY        0                // LCD 書き込み OK
#define    CHECKBUSY    1                 // LCD 書き込み busy をチェックする
#define    NOCHECKBUSY    0               // LCD 書き込み busy をチェックしない

//=========================================================
// 内部使用関数
//=========================================================
void lcdwriteinit( unsigned char command );
                                          // LCD 初期設定
void lcdwrite1command( unsigned char command, unsigned char flag );
                                          // LCD1 コマンド出力
void    lcdwrite1data( unsigned char data, unsigned char flag );
                                          // LCD1 データ出力
void wait1( void );                        // 0.1ms ウエイト
void wait2( void );                        // 4.1ms ウエイト
void wait3( void );                        // 15ms ウエイト
unsigned char lcdbusycheck( void );        // LCD ビジーチェック

//=========================================================
// グローバル変数
//=========================================================
unsigned char display_stat;            // LCD に送った表示モードを保持

//=========================================================
// 関数名     :LCD_init()
// 機能          :LCD 初期設定
//=========================================================
```

```
void LCD_init( void ) {
    p9 =  0x00;                          // ポート 9(LCD data ) 初期値設定
    prc2 = HIGH;                        // p9 プロテクトレジスタ解除
    pd9 = PORTOUT;                       // ポート 9 方向を出力に設定
    p8 =  0x00;                          // ポート 8(LCD data ) 初期値設定
    pd8 = PORTOUT;                       // ポート 8 方向を出力に設定

// 4 ビット× 2 回のデータ転送ができるように設定
    wait3();                            // 15ms ウエイト
    lcdwriteinit( 0x03 );                // LCD ファンクションセット
    wait2();                            // 4.1ms ウエイト
    lcdwriteinit( 0x03 );                // LCD ファンクションセット
    wait1();                            // 0.1ms ウエイト
    lcdwriteinit( 0x03 );                // LCD ファンクションセット
    wait1();
    lcdwriteinit( 0x02 );                // LCD データを 4 ビット長に設定
    wait1();
    lcdwrite1command(0x28, NOCHECKBUSY);    // 4bit2 行文 5 × 7 ドットに設定
    wait1();

// この後はビジーチェックをしてからデータ転送
    lcdwrite1command(0x08, CHECKBUSY);    // 表示オフ
    lcdwrite1command(0x01, CHECKBUSY);    // 表示クリア
    lcdwrite1command(0x06, CHECKBUSY);    // エントリーモードインクリメント
    lcdwrite1command(0x0f, CHECKBUSY);
    lcdwrite1command(0x14, CHECKBUSY);

    display_stat = 0x0f;                 // 現在の表示状態
}

//=============================================================
// 関数名     :LCD_cls()
// 機能：DDRAM の内容を全て消去して位置を (0,0) に設定
//=============================================================
void LCD_cls( void ) {
    lcdwrite1command( 0x01, CHECKBUSY );
}

//=============================================================
// 関数名     :LCD_off()
// 機能：表示 OFF
//=============================================================
void LCD_off( void ) {
    display_stat = display_stat & 0xfb;
    lcdwrite1command( display_stat, CHECKBUSY );
}

//=============================================================
// 関数名     :LCD_on()
// 機能：表示 ON
//=============================================================
void LCD_on( void ) {
    display_stat = display_stat | 0x04;
    lcdwrite1command( display_stat, CHECKBUSY );
}

//=============================================================
// 関数名     :LCD_locate()
// 機能：カーソル位置を (x,y) に指定。
//         cursor!=0 で、カーソルを表示。
//         blink!=0 で、カーソル位置のキャラクタを点滅
//=============================================================
```

```
void LCD_locate( unsigned char x, unsigned char y, unsigned char cursor, unsigned char blink ) {
    unsigned char DDRAMadrs;

    if ( x < 0 || x > 15 ) { return; }
    if ( y < 0 || y >  1 ) { return; }

    DDRAMadrs = x + 0x80;
    if ( y == 1 ) {
        DDRAMadrs = DDRAMadrs + 0x40;
    }
    lcdwrite1command( DDRAMadrs, CHECKBUSY );

    if ( cursor == 0 ) {
        display_stat = display_stat & 0xfd;
    } else {
        display_stat = display_stat | 0x02;
    }
    if ( blink == 0 ) {
        display_stat = display_stat & 0xfe;
    } else {
        display_stat = display_stat | 0x01;
    }
    lcdwrite1command( display_stat, CHECKBUSY );
}

//=============================================================
// 関数名：LCD_print()
// 機能：カーソル位置にキャラクタを 1 文字出力
//=============================================================
void LCD_print( unsigned char c ) {
    lcdwrite1data( c, CHECKBUSY );
}

//=============================================================
// 関数名：LCD_print_str()
// 機能：(0,0) から最大 32 文字を表示。
//=============================================================
void LCD_print_str( unsigned char str[] ) {


}

//=============================================================
// 関数名     :lcdwriteinit()
// 機能         :LCD 初期設定コマンドセット
//=============================================================
void lcdwriteinit( unsigned char command ) {
    prc2 = HIGH;            // p9 プロテクトレジスタ解除
    pd9 = PORTOUT;            // p9 出力に設定
    p8 =  0x00;               // ポート 8(LCD  data  ) 初期値設定

    command &= 0x0f;        // p8 コントロールデータセット (引数から）RW:0(W 指定) RS:0(コマンド指定）E:0
    p9 = command;            // p8 コントロールデータ出力
    LCDE = HIGH;            // E:1
#pragma ASM                   // アセンブラ表記
    NOP                       // 時間調整
    NOP
    NOP
    NOP
#pragma ENDASM                // アセンブラ表記終了
    LCDE = LOW;               // E:0
}
```

```
//=========================================================
// 関数名     :lcdwrite1command()
// 機能         :LCD コマンド出力
//=========================================================
void lcdwrite1command( unsigned char command, unsigned char flag ) {
    unsigned char outcommand;

    if ( flag == CHECKBUSY ) {
        while ( lcdbusycheck() == BUSY ) {}        //LCD ビジーチェック
    }

    prc2 = HIGH;            // p9 プロテクトレジスタ解除
    pd9 = PORTOUT;            // p9 出力に設定
    p8 = 0x00;                // p8 初期値セット

    outcommand = command>>4;// 上位 4 ビットを下位 4 ビットへ
    outcommand &= 0x0f;        // マスク処理
    p9 = outcommand;        // コマンドデータ上位 4 ビットを出力
    LCDE = HIGH;            // E:1
#pragma ASM                    // アセンブラ表記
    NOP                        // 時間調整
    NOP
    NOP
    NOP
#pragma ENDASM                // アセンブラ表記終了
    LCDE = LOW;                // E:0

    outcommand = command&0x0f;    // コマンドデータ下位 4 ビット抽出
    p9 = outcommand;            // コマンドデータ下位 4 ビットを出力
    LCDE = HIGH;            // E:1
#pragma ASM                    // アセンブラ表記
    NOP                        // 時間調整
    NOP
    NOP
    NOP
#pragma ENDASM                // アセンブラ表記終了
    LCDE = LOW;                // E:0
}

//=========================================================
// 関数名     :lcdwrite1data()
// 機能         :LCD データ出力
//=========================================================
void    lcdwrite1data( unsigned char data, unsigned char flag ) {
    unsigned char lcddata;

    if ( flag == CHECKBUSY ) {
        while ( lcdbusycheck() == BUSY ) {}        //LCD ビジーチェック
    }

    prc2 = HIGH;            // p9 プロテクトレジスタ解除
    pd9 = PORTOUT;            // p9 出力に設定
    p8 = 0x00;                // p8 初期値セット

    lcddata = data>>4;        // LCD データ上位 4 ビットを下位 4 ビットへ
    lcddata &= 0x0f;        // マスク処理
    p9 = lcddata;            // LCD データ上位 4 ビットを出力
    LCDRS = LCD_DATA;        // RS:1(データ指定)
    LCDE = HIGH;            // E:1
#pragma ASM                    // アセンブラ表記
    NOP                        // 時間調整
```

```
    NOP
    NOP
    NOP
#pragma ENDASM                  // アセンブラ表記終了
    LCDE = LOW;                  // E:0

    lcddata =data & 0x0f;     // LCD データの下位 4 ビット抽出 (上位 4 ビットマスク)
    p9 = lcddata;             // LCD データ下位 4 ビットを出力
    LCDRS = LCD_DATA;         // RS:1(データ指定)
    LCDE = HIGH;             // E:1
#pragma ASM                     // アセンブラ表記
    NOP                          // 時間調整
    NOP
    NOP
    NOP
#pragma ENDASM                  // アセンブラ表記終了
    LCDE = LOW;                  // E:0
}
```

3.6 課題

(1) LCDfunc.c の中の未完成の関数 LCD_print_str() を完成させてください。関数の仕様は次の通りです。

・引数（1 次元配列の先頭アドレス）で与えられる文字列を、¥0（終端記号）まで、または 32 文字まで、LCD に表示する。

・表示開始位置は、(0,0)（左上）とする。

・オプションで、現在位置から表示するように仕様を変更してもよい。

(2) LCD_print_str() が完成したら、sprintf() 関数を使って、SWa=234 のように書式付きで変数の値を文字列に出力できるようになります。この機能を使って、トグルスイッチをトグルした回数（「上げ-下げ」で 1 回とする）と、プッシュスイッチを押した回数を、正確に LCD に表示してください。チャタリング除去方法は、**一定時間後に再確認方式**を使えばよいと思います。サンプルプログラムを少し変更して、スイッチの変化（押された、上げた）を検出したら、何もしないループで時間をつぶします。スイッチの状態を再確認して、1 回の変化とします。

全部で 32 文字表示できるので、表示できる最大回数、スイッチの個数、表示デザインなどを工夫してみましょう。

sprintf() 関数の書式は『C コンパイラユーザーズマニュアル』(nc30uj.pdf) 付録 E-65 ページ、書式フォーマット指定方法は同じく付録 E-54 ページ以後を参照。

各スイッチが接続されているポートとビットは、次の表を参照。

CPU	信号名	CN1	CN2	MC 拡張基板	MC リモコン基板
40	P10-4/AN4/KI0*		34	ADC / I/O-4	トグル SW-a to GND
39	P10-5/AN5/KI1*		33	ADC / I/O-5	トグル SW-b to GND
38	P10-6/AN6/KI2*		32	ADC / I/O-6	トグル SW-c to GND
37	P10-7/AN7/KI3*		31	ADC / I/O-7	トグル SW-d to GND
36	P1-5/INT3*/ADtrg		30	ADtrg / I/O-8	プッシュSW-e to GND
35	P1-6/INT4*		29	I/O-9	プッシュSW-f to GND
34	P1-7/INT5*		28	I/O-a	プッシュSW-g to GND

4　割り込みとタイマ

この章の目次：

4.1　目的

まず、割り込みがどのような仕組みで実現されていて、割り込みを使うとどのような利点があるかを説明します。次に、制御プログラムなどで最もよく利用される、タイマ割り込みの使い方とプログラムの書き方を説明します。

4.2　サンプルプログラムのコンパイルとダウンロード

(0) マイクロコントローラ基板を拡張基板にセットしてから、拡張基板に電源と RS-232C ケーブルを接続して、電源を入れます。

(1) TM を起動します。

4.2.1　割り込みを使わないプログラム

(2) 「プロジェクトを開く」ボタンをクリックして `H:￥smaple￥sec03￥sample1.tmk` を開きます。

(3) 「リビルド」ボタンをクリックして、全てコンパイルします。時々、「製品版を購入しなさい」という勧告メッセージが表示されて動作が停止しますが、そのまま待ちます。コンパイルが終了すると、Builder ウィンドウに、「******** Finish... 」と出ます。

(4) 「デバッガの起動」ボタンをクリックして、リモートデバッガ (KD30) を起動します。

注意：起動しないこともあります。そのときは、「ツールの登録」ボタンをクリックして「DEBUG TOOL」の「KD30」をチェックしてから、「デバッガの起動」ボタンをクリックします。

(5) リモートデバッガの [File]-[Download]-[Load Module...] で、`H:￥smaple￥sec03￥sample1.x30` を開きます。

(6) リモートデバッガの「↓ Go」ボタンをクリックします。

このサンプルプログラムは、プッシュスイッチが押されたかどうかを調べて、押されたらLEDの点灯/消灯を反転するものです。ただし、このプログラムにはプッシュスイッチの状態を調べること以外にもたくさん仕事があります（`for ( i = 0; i < 1000000; i++ ){` の部分）。このため、仕事の合間にしかスイッチを調べることができません（`for ( i = 0; i < 100; i++ ){` の部分）。スイッチの状態を調べている期間は、確認のためにLEDa（赤）を点灯しています。

```
//=====================================================================
//     ファイル名：     0301.c
//     内容：              タイミングよく押せばフルカラー LED がトグル
//=====================================================================

#pragma ADDRESS P1    3e1H    // ポート P1 レジスタ
#pragma ADDRESS PD1    3e3H    // ポート P1 方向レジスタ
                              // 0：入力モード，1：出力モード。bit0-4 は 0 書込
#pragma ADDRESS P7    3edH    // ポート P7 レジスタ
#pragma ADDRESS PD7    3efH    // ポート P7 方向レジスタ
                              // 0：入力モード，1：出力モード

unsigned char P1, PD1, P7, PD7;

main() {
    unsigned long i;
    unsigned char memP1 = 0xff;

    PD1 = 0x00;                 // ポート 1 方向レジスタ　入力
    PD7 = 0xff;                 // ポート 7 方向レジスタ　出力
    P7 = ~0x00;                  // ポート 7 出力 L(LED 消灯）

    while ( 1 ) {
// 少しの間、スイッチの状態を確認
        for ( i = 0; i < 100; i++ ) {
            P7 = ~(~P7 | 0x02 );     // キー入力可能期間は LED を点灯
            if ( ((P1 & 0x20) != (memP1 & 0x20)) && ((P1 & 0x20) == 0) ) {
                                // P1 の bit5 が変化してかつ 0 = 赤が押された
                P7 ^= 0x01;     // P7 の bit0 を反転 = フルカラー LED の赤色反転
            }
            memP1 = P1;
            P7 = ~(~P7 & 0xfd );     // キー入力可能期間終了
        }
// 他にやるべきたくさんの仕事
        for ( i = 0; i < 1000000 ; i++ ) {
                                // スイッチのチェックどころではない、たいへんな処理内容
        }
    }
}
```

4.2.2 割り込みを使ったプログラム

(2) 「プロジェクトを開く」ボタンをクリックして `H:￥smaple￥sec03￥sample2.tmk` を開きます。

(3) 「リビルド」ボタンをクリックして、全てコンパイルします。時々、「製品版を購入しなさい」という勧告メッセージが表示されて動作が停止しますが、そのまま待ちます。コンパイルが終了すると、Builderウィンドウに、「******** Finish... 」と出ます。

(4) 「デバッガの起動」ボタンをクリックして、リモートデバッガ (KD30) を起動します。
注意：起動しないこともあります。そのときは、「ツールの登録」ボタンをクリックして「DEBUG TOOL」の「KD30」をチェックしてから、「デバッガの起動」ボタンをクリックします。

(5) リモートデバッガの [File]-[Download]-[Load Module...] で、H:￥smaple￥sec03￥sample2.x30 を開きます。

(6) リモートデバッガの「↓ Go」ボタンをクリックします。

このサンプルプログラムでは、main() 関数の while ループ内で、スイッチの状態を調べる、という処理自体を行っていません。ところが、プッシュスイッチの状態に応じて、LCD に「ON」と「OFF」が表示されて、スイッチを押せば必ず LED の点灯/消灯が反転します。

前回の課題で、LCDfunc.c の中の未完成の関数 LCD_print_str() が完成している人は、自分で作った方の LCDfunc.c を H:￥smaple￥sec03￥にコピーしてコンパイルしてください。50msec ごとにインクリメントする数字が表示されます。

このサンプルプログラムは、今までのものと少し違います。課題で作るプログラムは、このサンプルプログラムをもとに作ることになるので、違いを説明します。

まず、「プロジェクトの新規作成」で登録するスタートアッププログラムが違います。これは、後でプロジェクトの作り方を説明します。

プログラムでは、割り込み関数のプロトタイプ宣言と#pragma INTERRUPT 命令が必ず必要です。使わない関数の分も、全て必要です。割り込み関数の本体部分も、使わない関数でも必ず記述します。

```
//=====================================================================
//     ファイル名：     0302.c
//     内容：              一定時間ごとにスイッチ入力をチェック
//                       LED をトグル
//
// プロジェクトの新規作成では、「ncrt0_i.a30」をスタートアップとして登録すること。
// 割り込みを使うため。
//=====================================================================
#include <stdio.h>                      // sprintf() を使うため
#include "sfr26.h"                     // OAKSmini 用定義ファイル
#include "LCDfunc.h"                  // LCD 表示

// プロトタイプ宣言
void ta0int( void );                  // 割込み関数
void ta1int( void );                  // 割込み関数
void ta2int( void );                  // 割込み関数
void ta3int( void );                  // 割込み関数
void ta4int( void );                  // 割込み関数
void tb0int( void );                  // 割込み関数
void tb1int( void );                  // 割込み関数
void tb2int( void );                  // 割込み関数
void adint( void );                     // 割込み関数
#pragma INTERRUPT ta0int
#pragma INTERRUPT ta1int
#pragma INTERRUPT ta2int
#pragma INTERRUPT ta3int
#pragma INTERRUPT ta4int
#pragma INTERRUPT tb0int
#pragma INTERRUPT tb1int
#pragma INTERRUPT tb2int
#pragma INTERRUPT adint
```

```
//   マクロ定義
#define cnt_ta0 31250−1                 // タイマ A0 カウンタ値

// 変数の宣言
static unsigned char memp1;             // 前回のスイッチ状態のサンプリング
static unsigned char nowp1;             // 今回（最新）のスイッチ状態のサンプリング
static long timer_count;               // 一定時間ごとに自動的にインクリメントする変数
char buff[64];                          // グローバル変数にバッファを確保

//=====================================================================
// 関数名     :main()
// 機能         :     50msec 周期で割り込み関数を呼ぶ設定
//              :     タイマ A0 のカウントソースを f32 とする
//              :     1 カウントは 1/(20MHz/32)=1.6 μ sec
//              :     タイマ値の初期値 = 0.05sec/1.6 μ sec = 31250
//=====================================================================
void main( void ) {
// 変数の初期化
    memp1 = 0;                          // 前回のサンプリングを初期化
    nowp1 = 0;                          // 今回のサンプリングを初期化
    timer_count = 0;                   // 時間変数の初期化

// LCD 初期化
    LCD_init();                         // LCD 初期設定
    LCD_cls();                         // 全て消去

// I/O ポート初期化
    pd1 = 0x00;                         // ポート 1 方向レジスタ　入力
    pd7 = 0xff;                         // ポート 7 出力設定
    p7 = ∼0x00;                         // ポート 7 出力 L(LED 消灯）

// タイマ割り込み初期化
    udf = 0x00;                         // ダウンカウント設定
    ta0mr = 0x80;                      // タイマモード　　クロック：1/32
    ta0 = cnt_ta0;                     // タイマ値の初期化
    ta0ic = 0x06;                      // 割り込みレベルの設定
    tabsr = 0x01;                      // カウント開始
    _asm( "\tFSET I");                 // 割り込み許可

    while ( 1 ) {
// 時間のかかる処理をしていても、一定時間ごとにスイッチの状態を確認している
// スイッチの状態は、nowp1 でいつでも確認できる

// 時間変数を LCD に表示
        LCD_locate( 0, 0, 0, 0 );     // カーソルを (0,0) に、カーソル OFF
        sprintf( buff, "time=%d", timer_count );
        LCD_print_str( buff );

// スイッチの状態を LCD に表示
        LCD_locate( 0, 1, 0, 0 );     // カーソルを (0,1) に、カーソル OFF
        if ( (nowp1 & 0x20) == 0 ) {
            LCD_print( ' ' );
            LCD_print( 'O' );
            LCD_print( 'N' );
        } else {
            LCD_print( 'O' );
            LCD_print( 'F' );
            LCD_print( 'F' );
        }
    }
}
```

```
//=====================================================================
// 関数名     :ta0int()
// 機能          :ta0 割り込み関数
//          long  timer_count のインクリメント
//         赤プッシュスイッチ：押し  =>  フルカラー LED：赤を
//=====================================================================
void ta0int( void ) {
    timer_count = timer_count +1;    // 時間変数のインクリメント
    p7 ^= 0x02;                        // LEDa の点滅

    nowp1 = p1;                        // スイッチ状態のサンプリング
    if ( ( (memp1 & 0x20) != 0 ) && ( (nowp1 & 0x20) == 0 ) ) {
                                     // p1 の bit5 が、前回 1 で今回 0 = 赤押
        p7 ^= 0x01;                    // p7 の bit0 を反転 = フルカラー LED の赤色反転
    }
    memp1 = nowp1;                    // p1 を保存
}

//=====================================================================
// 以下は、使うときにきちんと記述する
//=====================================================================
void ta1int( void ) {}
void ta2int( void ) {}
void ta3int( void ) {}
void ta4int( void ) {}
void tb0int( void ) {}
void tb1int( void ) {}
void tb2int( void ) {}
void adint( void ) {}
```

4.3　割り込み

割り込みとは、実行中の処理を一時中断して、別の処理を行う機能です。別の処理が終了したら、中断していた処理を継続することができます。

詳細は、『ハードウェアマニュアル』(rjj09B0033_m16chm.pdf) 46〜62 ページを参照してください。

4.3.1　なぜ割り込みを使うのか

ある計算方法でデータ処理をして結果をファイルに出力する、といった C 言語プログラムでは、割り込み処理を使う必要はありません。では、なぜ割り込みを使うのでしょうか。

外部との入出力があるプログラムでは、いつ起こるかわからない入力の変化に応じて処理内容を変える必要があります。例えば、人間が行っている処理を見てみましょう。

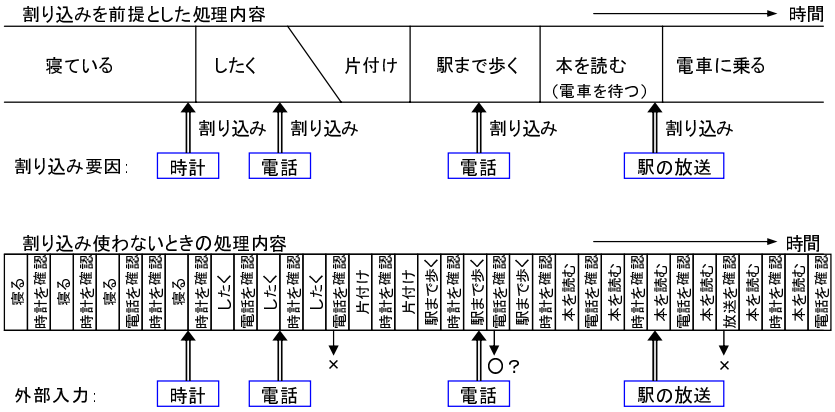


図. 4.1: 割り込みを使う場合と使わない場合の処理内容。

朝、予定の時間に目覚まし時計が鳴って目を覚まします。目覚まし時計が鳴るまでは、今何時なのかを気にする必要はなく、安心して寝てられます。電話がかかってくるとベルが鳴るので、もちろん電話がかかってきたことがわかります。電話がかかってきているかを頻繁にチェックする必要はありません。駅で電車を待っているときには、電車が来れば放送があります。電車が来るかどうか、常に気にしている必要はありません。このような、あたりまえの処理が、割り込み処理です。

それでは、もし割り込みを使わなかったらどうなるでしょうか。寝ている間も、定期的に時間を調べる必要があります。調べるのを忘れたら、予定の時間に起きられません。電話がかかってきているかどうかを頻繁にチェックして、予定の時間の電車に乗れるように時計もチェックしながらしたくします。歩いているときも電話をチェックします。チェックしていないときにかかってきた電話には、出ることができません（電話がないのと同じ）。駅の放送もチェックする必要がありますが、チェックする項目が多くなるとチェックの周期が長くなり、見逃す心配もあります（いつになっても電車が来ないのと同じ）。

割り込みを使う理由は、この例からも想像できるように、制御プログラムを作る上で、次のような重要な利点があるからです。

(1) 定期的に行う必要がある処理が確実にできる（一定時間ごとの値のサンプリングなど）。
(2) たまにしか起こらない外部入力（シリアル通信やスイッチ入力など）にも確実に応答できる。
(3) 外部入力に対する応答が早く確実になる。
(4) プログラム全体の処理効率が上がる（むだなチェックをする必要がない）。
(5) プログラムの構造がわかりやくすくなる。

次のような欠点もあります。ただし、対策はあるし、欠点以上に利点の方が大きいです。

(1) 割り込み制御レジスタの設定や、多重割り込みへの対策をする必要がある。

(2) デバッグがしにくい（複数のプログラムが同時に動作しているのと同じなので、どのプログラムによる影響なのかがわからなくなることがある）。

制御プログラムを作るときには、次のような点に心がけてダークサイドに飲み込まれないようにします。

(1) 割り込み制御レジスタの機能を理解する。

(2) 割り込み要因を整理して、むやみにたくさんの割り込みを扱わない。

(3) プログラム全体の構造を整理して、原因（割り込み要因とそれで変化する変数）と結果（変化した変数に対する応答）を把握する。

(4) 確実に動作するプログラムから構築を初める。

(5) 割り込み処理ルーチンでの処理時間を、実際に測定する。図 4.2 で、割り込み処理ルーチンの実行時間を測定して割り込み周期（T[sec]）よりもずっと短いことを確認する。

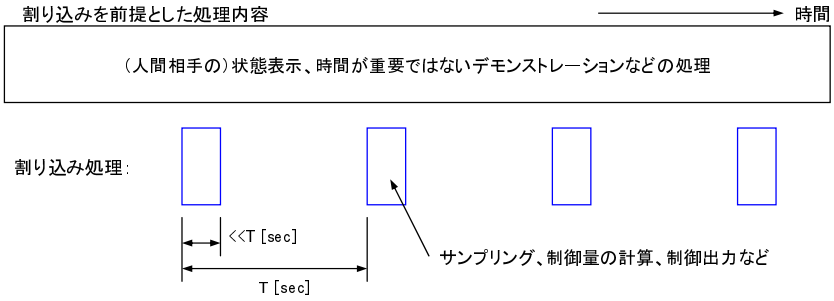


図. 4.2: 割り込み処理ルーチンの実行時間を必ず測定すること。

4.3.2 割り込みの分類

実習で利用するマイクロコントローラには、図 4.3 のような割り込み要因があります。

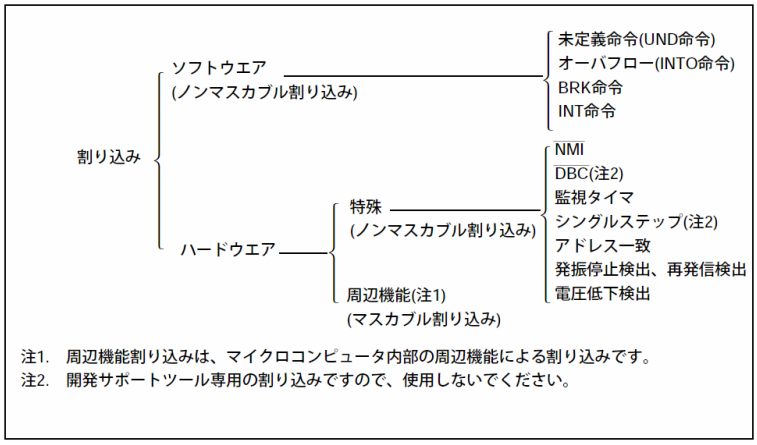


図. 4.3: 割り込み要因。

「マスカブル割り込み」というのは、割り込み許可フラグによる割り込みの許可（禁止）や、割り込み優先レベル指定による割り込み優先順位の変更が可能な割り込みのことです。制御プログラムで主に利用するタイマ割り込みは、マスカブル割り込みです。

「ノンマスカブル割り込み」というのは、割り込み許可フラグによる割り込みの許可（禁止）や、割り込み優先順位の変更が不可能な割り込みのことです。ソフトウェア上、またはハードウェア上、重大な障害が起こる可能性がある場合への対応を記述します。

4.3.3 割り込みが作動する仕組みと制御

マイクロコントローラは、割り込みを次のような手順で実行します。

(1) 割り込み要因（タイマなど）からの割り込み要求を検出。

(2) 割り込み優先レベルの判定（割り込みを受け付けるか無視するかを判定)。

(3) 割り込み処理。

(3-1) 割り込みシーケンス（マイクロコントローラのハードウェアが自動的に行う)。

(3-1-1) 割り込み番号（要因）と割り込み要求レベルを調べる。

(3-1-2) フラグレジスタを待避する。

(3-1-3) 割り込みを禁止する。

(3-1-4) レジスタの内容をスタックに待避する。

(3-1-5) プログラムカウンタを待避する。

(3-1-6) 受け付け中の割り込みの優先レベルを設定する。

(3-2) 割り込み関数を実行する。

(3-2-1) プログラムカウンタに割り込みベクタ（割り込み番号に対応した割り込み関数のアドレス）を設定する。

(3-3) レジスタとフラグを復元する。

C 言語で割り込みを使用するためには、次のような処理が必要です。

(1') タイマなどの割り込み要因の設定。

```
#define cnt_ta0 31250-1 // タイマ A0 カウンタ値
    udf = 0x00;         // ダウンカウント設定
    ta0mr = 0x80;       // タイマモード クロック：1/32
    ta0 = cnt_ta0;      // タイマ値の初期化
```

(2') 割り込み優先レベルの設定と割り込みの許可。

```
    ta0ic = 0x06;       // 割り込みレベルの設定
    tabsr = 0x01;       // カウント開始
    _asm( "\tFSET I");  // 割り込み許可
```

(3') 割り込み関数の記述。

```
void ta0int( void );    // 割込み関数のプロトタイプ
#pragma INTERRUPT ta0int

void ta0int( void ) {
関数の内容の記述
}
```

4.3.4　割り込み設定のためのレジスタ

前述の設定例では、タイマ A0 の割り込み優先レベルを「6」に設定しました。割り込みを使う上では、優先レベルを適切に設定する必要があります。

優先レベルは「7」が最も優先されて、他の割り込み関数を実行中でも、その処理に「割り込む」ことができます。優先レベル「7」の割り込み関数を実行している間は、一切のマスカブル割り込みを受け付けません。

優先レベル「1」は優先度が最も低く、より優先度の高い割り込みに割り込まれる可能性があります。

割り込み要因が 1 種類だけのときには特に気にする必要はありませんが、2 種類以上のときには、どちらを優先するかを決定する必要があります。また、優先されない方の割り込み関数は、関数を実行中に割り込まれることを覚悟の上で内容を設計します。

タイマ A0 の割り込み優先レベルを「6」に設定することの具体的な意味は次の通りです。

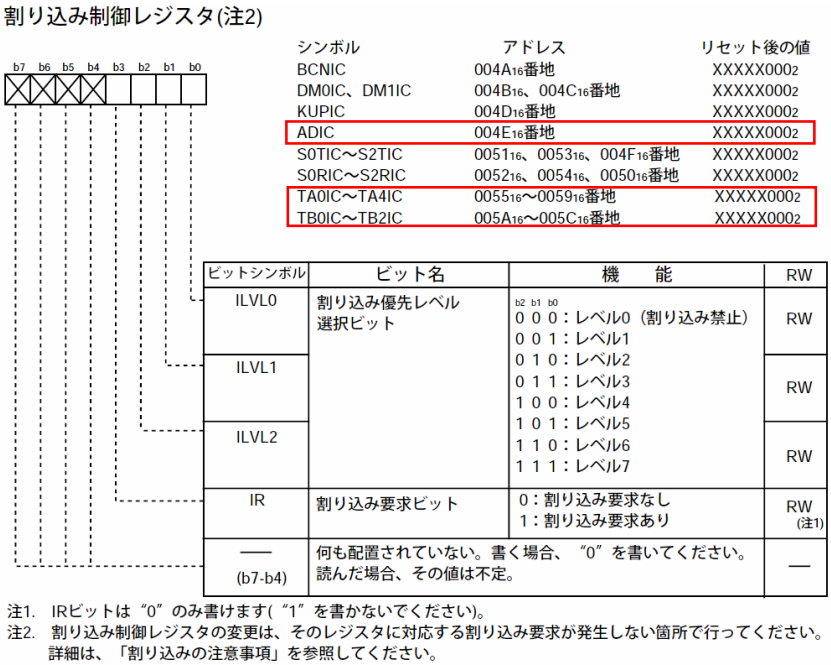
・タイマ A0 の優先度はかなり高い。

・他に割り込み関数があって、割り込み要因を検出して割り込み関数を実行中でも、その割り込みの優先レベルが「1〜5」だったら、タイマ A0 割り込みが実行される。

・他に割り込み関数があって、割り込み要因を検出して割り込み関数を実行中のとき、その割り込みの優先レベルが「6〜7」だったら、タイマ A0 割り込みは保留される。

・タイマ A0 割り込み関数を実行中に、割り込み優先レベル「7」の割り込み要因からの割り込みがあれば、それが実行される。

・タイマ A0 割り込み関数を実行中に、割り込み優先レベル「1〜6」の割り込み要因からの割り込みがあっても、それは保留される。



割り込み制御レジスタ(注2)

シンボル	アドレス	リセット後の値
BCNIC	004A16番地	XXXXX0002
DM0IC、DM1IC	004B16、004C16番地	XXXXX0002
KUPIC	004D16番地	XXXXX0002
ADIC	004E16番地	XXXXX0002
S0TIC〜S2TIC	005116、005316、004F16番地	XXXXX0002
S0RIC〜S2RIC	005216、005416、005016番地	XXXXX0002
TA0IC〜TA4IC	005516〜005916番地	XXXXX0002
TB0IC〜TB2IC	005A16〜005C16番地	XXXXX0002

ビットシンボル	ビット名	機　能	RW
ILVL0	割り込み優先レベル選択ビット	b2 b1 b0 0 0 0：レベル0（割り込み禁止） 0 0 1：レベル1 0 1 0：レベル2 0 1 1：レベル3 1 0 0：レベル4 1 0 1：レベル5 1 1 0：レベル6 1 1 1：レベル7	RW
ILVL1			RW
ILVL2			RW
IR	割り込み要求ビット	0：割り込み要求なし 1：割り込み要求あり	RW (注1)
—— (b7-b4)	何も配置されていない。書く場合、“0”を書いてください。読んだ場合、その値は不定。		—

注1. IRビットは“0”のみ書けます(“1”を書かないでください)。
注2. 割り込み制御レジスタの変更は、そのレジスタに対応する割り込み要求が発生しない箇所で行ってください。詳細は、「割り込みの注意事項」を参照してください。

図. 4.4: 割り込み制御レジスタ。

4.3.5　割り込みを使うプロジェクトの作り方

プロジェクト用のフォルダを作って、その中にこれから新しく作る空のソースファイルを作るところまでは、割り込みを使わないときと同じです。

各自の H:ドライブのルート直下にある H:￥sample￥startup フォルダの中身（5 個のファイル）を、全て用意したフォルダにコピーします。

TM の「New Project（プロジェクトの新規作成）」ボタンをクリックします。

(1) 最初のダイアログで、「プロジェクト名」を入力します。

(2) ワーキングディレクトリは、[...] ボタンをクリックして、先ほど用意したフォルダを選びます。

(3) 「次へ (N) >」でプロジェクトの種類を選びます。もちろん「C 言語プロジェクト」です。

(4') 次のダイアログで、スタートアッププログラムを選びます。「◎カスタム」を指定して [...] ボタンをクリックして、先ほど用意したフォルダの中の ncrt0_i.a30 を選びます。

(5) ウィザードの最後で、設定した内容を確認します。

(6) 新しいプロジェクトができたら、ソースファイルをこのプロジェクトに登録します。「プロジェクトエディタ」で、目的とする実行オブジェクトのファイル名を右クリックして、メニューから「アイテムの編集 (A)」→「ファイルの追加 (F)」を選択します。「ファイルを開く」ダイアログで、用意したフォルダの中のソースファイルを開きます。

(7) ソースファイルがいくつかに分割されているときには、全てのソースファイルを同様に登録します。

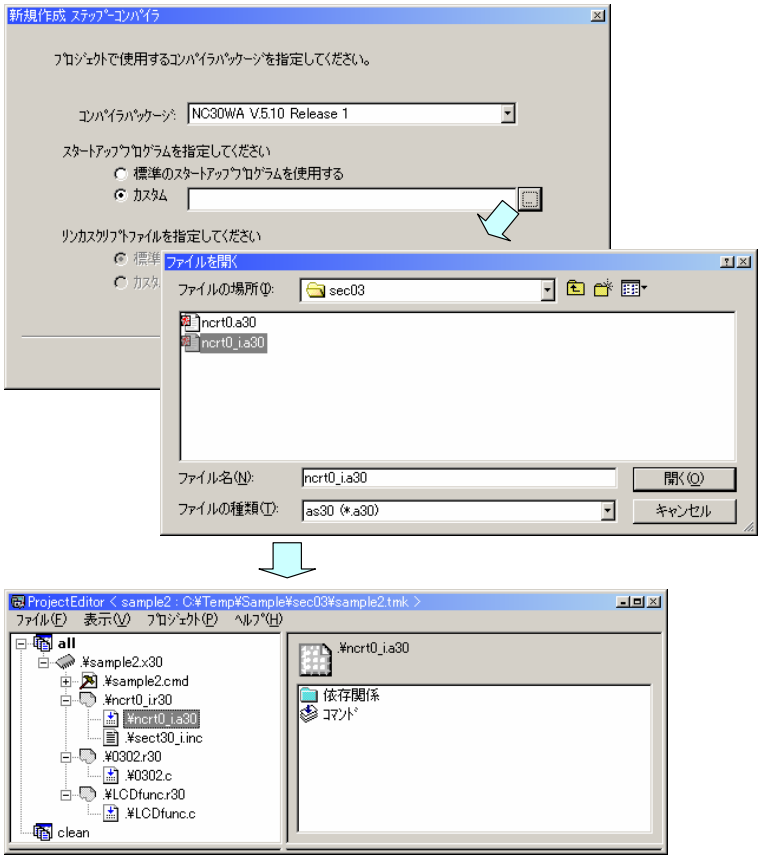


図. 4.5: 割り込みを使うときには、スタートアッププログラムとして ncrt0_i.a30 を指定すること。

4.4　タイマ

実習で使用するマイクロコントローラには、16 ビットタイマが 8 本あります。この 8 本のタイマは、持っている機能によってタイマ A（5 本）とタイマ B（3 本）の 2 種類に分類できます。すべてのタイマは、それぞれ独立して動作します。

詳細は、『ハードウェアマニュアル』(rjj09B0033_m16chm.pdf) 74〜102 ページを参照してください。

4.4.1　タイマの機能

ここでは、タイマ A の機能の概略を説明します。その後、サンプルプログラムで設定している機能を説明します。

タイマ A には、次の 4 種類のモードがあります。モードは、TAiMR レジスタ (i=0〜4) の TMOD1〜0 ビットで選択できます。

タイマモード：　内部カウントソース（CPU クロックやその分周）をカウントするモードで、必ずダウンカウントします。カウンタのアンダフローが起こると、割り込みを発生して、リロードレジスタの内容をカウンタに転送してカウントを継続します。TABSR レジスタの TAiS ビットを「1」にすると、カウントを開始します。TAi レジスタを読むと、カウント値が読めます。

イベントカウンタモード：　外部からのパルス、他のタイマのオーバフロー、またはアンダフローをカウントするモードです。

ワンショットタイマモード：　カウント値が「0」になるまでの間に、1 度だけパルスを出力するモードです。

パルス幅変調 (PWM) モード：　任意の幅のパルスを連続して出力するモードです。

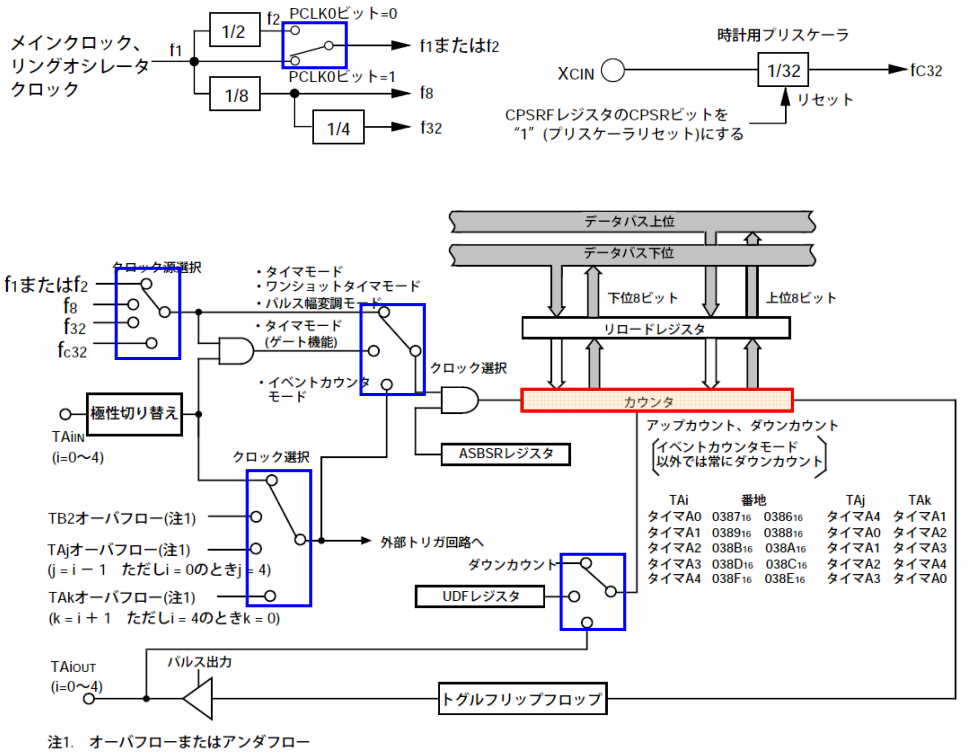


図. 4.6: タイマ A の構成。

4.4.2　タイマ割り込み

タイマからの割り込みを有効にするためには、まずタイマの周期を設定してから図 4.4 の割り込み制御レジスタに割り込み優先レベルを設定します。割り込み優先レベルを「0」以外に設定すれば、割り込みが有効になります。

4.4.3　設定例

ここでは、1/100 秒ごとにタイマ割り込みをかけるための設定方法を説明します。図 4.7 に、タイマカウンタの値の変化の概念図を示します。このように値が変化するように、タイマの各レジスタを設定します。

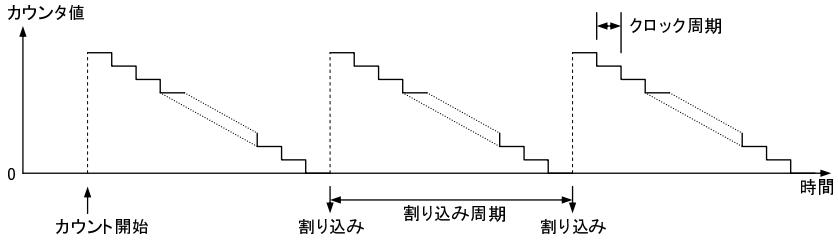


図. 4.7: カウンタ値の変化。

タイマAiモードレジスタ(i=0～4)

b7	b6	b5	b4	b3	b2	b1	b0	シンボル	アドレス	リセット後の値
		0				0	0	TA0MR～TA4MR	039616～039A16番地	0016

ビットシンボル	ビット名	機　能	RW
TMOD0	動作モード選択ビット	b1 b0 0 0 : タイマモード 0 1 : イベントカウンタモード 1 0 : ワンショットタイマモード 1 1 : パルス幅変調(PWM)モード	RW
TMOD1			RW
MR0	パルス出力機能選択ビット	0 : パルス出力なし (TAiOUT端子は入出力ポート) 1 : パルス出力あり(注1) (TAiOUT端子はパルス出力端子)	RW
MR1	ゲート機能選択ビット	b4 b3 0 0:, 0 1: ゲート機能なし (TAiIN端子は入出力ポート) 1 0: TAiIN端子に“L”が入力されている期間カウントする(注2) 1 1: TAiIN端子に“H”が入力されている期間カウントする(注2)	RW
MR2			RW
MR3	タイマモードでは“0”にしてください		RW
TCK0	カウントソース選択ビット	b7 b6 0 0 : f1またはf2 0 1 : f8 1 0 : f32 1 1 : fc32	RW
TCK1			RW

注1. TA0OUT端子はNチャネルオープンドレイン出力。
注2. TAiIN端子に対応するポート方向ビットは“0”(入力モード)にしてください。

図. 4.8: モードレジスタ。

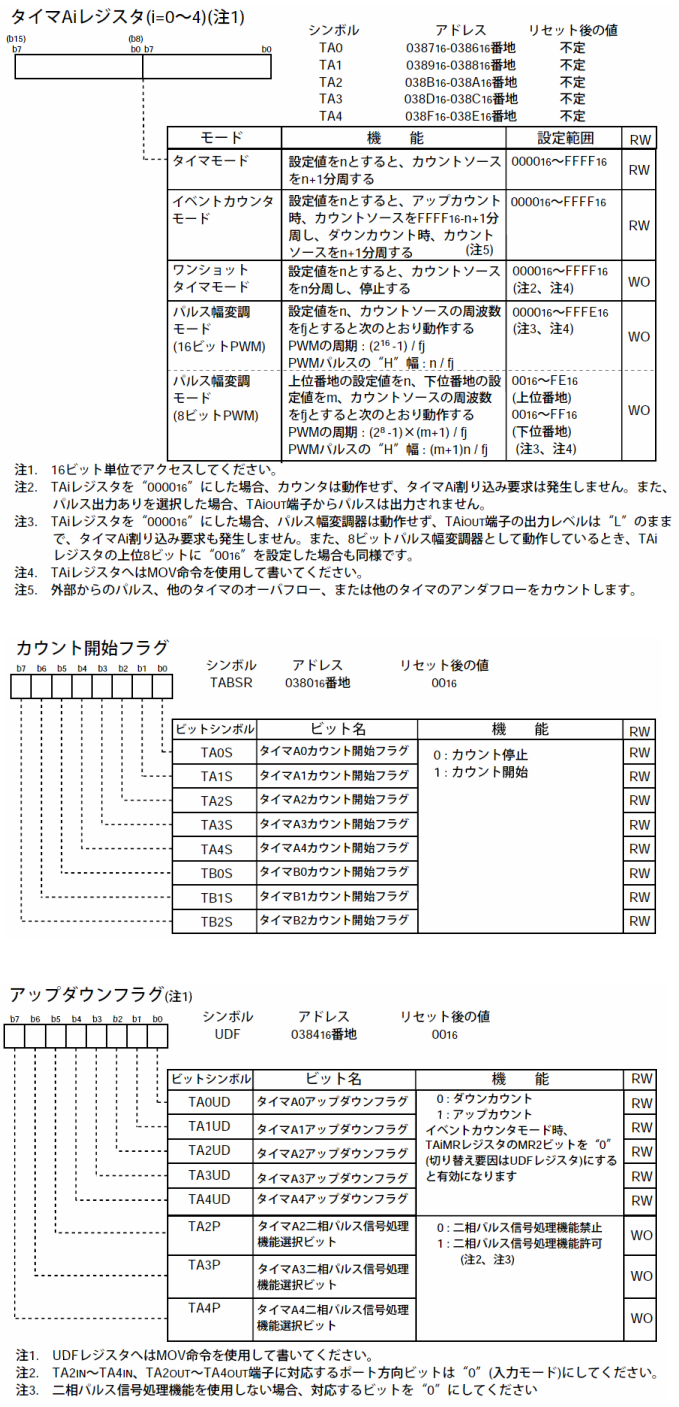
まず、タイマモードレジスタで、カウントソースを選び、タイマがカウントするクロックの周期を決めます。

```
ta0mr = 0x80;          // タイマモード クロック：1/32
```

CPU クロックの 32 分周は、20[MHz]/32 = 625[kHz] = 1.6[μ sec] です。
次に、カウントアップまたはカウントダウンを指定します。

```
udf = 0x00;            // ダウンカウント設定
```

さらに、1.6[μ sec] 周期のクロックをいくつカウントしたら、希望する割り込み周期になるかを決めます。ここでは、1/100[秒] = 10[msec] ごとに割り込みをかけたいので、10[msec]/1.6[μ sec] = 6250 をカウンタの初期値に設定します。ただし、レジスタへの設定値を n とすると、$n+1$ 分周した周期で割り込みがかかるので、実際に設定する値は、$6250-1$ になります。



タイマAiレジスタ(i=0〜4)(注1)

(b15) b7 (b8) b0 b7 b0

シンボル	アドレス	リセット後の値
TA0	038716-038616番地	不定
TA1	038916-038816番地	不定
TA2	038B16-038A16番地	不定
TA3	038D16-038C16番地	不定
TA4	038F16-038E16番地	不定

モード	機　能	設定範囲	RW
タイマモード	設定値をnとすると、カウントソースをn+1分周する	000016〜FFFF16	RW
イベントカウンタモード	設定値をnとすると、アップカウント時、カウントソースをFFFF16-n+1分周し、ダウンカウント時、カウントソースをn+1分周する (注5)	000016〜FFFF16	RW
ワンショットタイマモード	設定値をnとすると、カウントソースをn分周し、停止する	000016〜FFFF16 (注2、注4)	WO
パルス幅変調モード (16ビットPWM)	設定値をn、カウントソースの周波数をfjとすると次のとおり動作する PWMの周期：(216-1) / fj PWMパルスの“H”幅：n / fj	000016〜FFFE16 (注3、注4)	WO
パルス幅変調モード (8ビットPWM)	上位番地の設定値をn、下位番地の設定値をm、カウントソースの周波数をfjとすると次のとおり動作する PWMの周期：(28-1)×(m+1) / fj PWMパルスの“H”幅：(m+1)n / fj	0016〜FE16 (上位番地) 0016〜FF16 (下位番地) (注3、注4)	WO

注1. 16ビット単位でアクセスしてください。
注2. TAiレジスタを“000016”にした場合、カウンタは動作せず、タイマAi割り込み要求は発生しません。また、パルス出力ありを選択した場合、TAiOUT端子からパルスは出力されません。
注3. TAiレジスタを“000016”にした場合、パルス幅変調器は動作せず、TAiOUT端子の出力レベルは“L”のままで、タイマAi割り込み要求も発生しません。また、8ビットパルス幅変調器として動作しているとき、TAiレジスタの上位8ビットに“0016”を設定した場合も同様です。
注4. TAiレジスタへはMOV命令を使用して書いてください。
注5. 外部からのパルス、他のタイマのオーバフロー、または他のタイマのアンダフローをカウントします。

カウント開始フラグ

b7 b6 b5 b4 b3 b2 b1 b0

シンボル	アドレス	リセット後の値
TABSR	038016番地	0016

ビットシンボル	ビット名	機　能	RW
TA0S	タイマA0カウント開始フラグ	0：カウント停止 1：カウント開始	RW
TA1S	タイマA1カウント開始フラグ		RW
TA2S	タイマA2カウント開始フラグ		RW
TA3S	タイマA3カウント開始フラグ		RW
TA4S	タイマA4カウント開始フラグ		RW
TB0S	タイマB0カウント開始フラグ		RW
TB1S	タイマB1カウント開始フラグ		RW
TB2S	タイマB2カウント開始フラグ		RW

アップダウンフラグ(注1)

b7 b6 b5 b4 b3 b2 b1 b0

シンボル	アドレス	リセット後の値
UDF	038416番地	0016

ビットシンボル	ビット名	機　能	RW
TA0UD	タイマA0アップダウンフラグ	0：ダウンカウント 1：アップカウント イベントカウンタモード時、TAiMRレジスタのMR2ビットを“0”(切り替え要因はUDFレジスタ)にすると有効になります	RW
TA1UD	タイマA1アップダウンフラグ		RW
TA2UD	タイマA2アップダウンフラグ		RW
TA3UD	タイマA3アップダウンフラグ		RW
TA4UD	タイマA4アップダウンフラグ		RW
TA2P	タイマA2二相パルス信号処理機能選択ビット	0：二相パルス信号処理機能禁止 1：二相パルス信号処理機能許可 (注2、注3)	WO
TA3P	タイマA3二相パルス信号処理機能選択ビット		WO
TA4P	タイマA4二相パルス信号処理機能選択ビット		WO

注1. UDFレジスタへはMOV命令を使用して書いてください。
注2. TA2IN〜TA4IN、TA2OUT〜TA4OUT端子に対応するポート方向ビットは“0”(入力モード)にしてください。
注3. 二相パルス信号処理機能を使用しない場合、対応するビットを“0”にしてください

図. 4.9: タイマレジスタ、カウント開始フラグ、アップダウンフラグ。

```
#define cnt_ta0 6250-1  // タイマ A0 カウンタ値
    ta0 = cnt_ta0;      // タイマ値の初期化
```

後は、割り込み優先レベルの設定と割り込みの許可をするだけです。

```
    ta0ic = 0x06;       // 割り込みレベルの設定
    tabsr = 0x01;       // カウント開始
    _asm( "\tFSET I");  // 割り込み許可
```

4.5 課題

LCD ディスプレイを使って、1/100[秒] 単位で時間を計測できる「高性能」ストップウォッチを作ってみましょう。機能、利用するスイッチの種類と個数、内部動作状態の表示、LCD に表示する内容などは、各自で設計してください。

参考までに、ある携帯電話に実装されているストップウォッチの操作と動作を示します。

(1) リセットで、[開始] ボタンだけを受け付ける状態になる。「経過時間」も「スプリット表示」も「00' 00" 00」にリセットされる。

(2) [開始] を押すと作動中になり、「経過時間」に 1/100 秒単位で経過時間が表示される（実際には、1/100 秒ごとに必ず表示しているかどうかは人間にはわからない）。

(3) 作動中に [リセット] を押すと、リセット状態に戻る。

(4) 作動中に [停止] を押すと、停止状態になり経過時間が停止する。

(5) 作動中に [スプリット] を押すと、作動中のままでそれまでの経過時間がスプリットとして表示される。

(6) 停止状態のときに [リセット] を押すと、リセット状態に戻る。

(7) 停止状態のときに [再開] を押すと、再び作動中になる。

リセット後の状態　　作動中　　停止状態

図. 4.10: 携帯電話に実装されたストップウォッチの例。

プログラム上は、1/100[秒] ごとの割り込みでスイッチの状態を調べ、動作を決定します。割り込み関数で経過時間とスプリットを管理して、それを main() 関数で表示します。

5 AD 変換と PWM

この章の目次：

5.1 目的

マイクロコントローラに搭載されている AD 変換器を使って、アナログ電圧を入力する方法を説明します。リモコン基板の 3 個のボリュームの回転角度によってアナログ電圧が変化するように構成されているので、ボリュームの回転角度を入力として利用できるようになります。

また、タイマの機能を使って、パルス幅変調 (PWM) 出力をする方法を説明します。この章では、PWM 出力によってフルカラー LED の明るさを変える方法を説明します。

チャレンジテーマの製作では、ラジコン用サーボを使うことになります。ボリュームの回転角度に応じて PWM 波形を変化させることで、ラジコン用サーボの回転角度を指定できるようになります。次章（来週）では、実際にラジコン用サーボを動かすための PWM 波形出力を説明します。

5.2 サンプルプログラムのコンパイルとダウンロード

(0) マイクロコントローラ基板を拡張基板にセットしてから、拡張基板に電源と RS-232C ケーブルを接続して、電源を入れます。

(1) TM を起動します。

5.2.1 AD 変換結果の表示

(2) 「プロジェクトを開く」ボタンをクリックして `H:￥smaple￥sec04￥sample1.tmk` を開きます。

(3) 「リビルド」ボタンをクリックして、全てコンパイルします。時々、「製品版を購入しなさい」という勧告メッセージが表示されて動作が停止しますが、そのまま待ちます。コンパイルが終了すると、Builder ウィンドウに、「******** Finish... 」と出ます。

(4) 「デバッガの起動」ボタンをクリックして、リモートデバッガ (KD30) を起動します。

注意：起動しないこともあります。そのときは、「ツールの登録」ボタンをクリックして「DEBUG TOOL」の「KD30」をチェックしてから、「デバッガの起動」ボタンをクリックします。

(5) リモートデバッガの [File]-[Download]-[Load Module...] で、`H:￥smaple￥sec04￥sample1.x30` を開きます。

(6) リモートデバッガの「↓ Go」ボタンをクリックします。

このサンプルプログラムは、右側のボリュームの回転角度に応じて約 0〜5[V] に変化する電圧の AD 変換結果を、LCD にバーグラフで表示します。この電圧は、マイクロコントローラのアナログ入力チャンネル 0 に接続されています。2 章の課題で、`LCDfunc.c` の中の未完成の関数 `LCD_print_str()` が完成している人は、自分で作った方の `LCDfunc.c` を `H:￥smaple￥sec04￥` にコピーしてコンパイルしてください。AD 変換の結果を、0 から 1023 までの値[25] で確認できます。

プログラムでは、LCD と AD 変換の初期設定を行った後、$n(=10)$ 回の AD 変換の結果を平均して表示します。バーグラフでは、コード `0xff` のキャラクタを使って 16 段階で電圧を表示します。

```
//=========================================================
//     内容： AD 変換結果を LCD に表示
//              右側の VR の回転角度がわかる
// 0401.c
//=========================================================
#include <stdio.h>                      // sprintf() を使うため
#include "sfr26.h"                      // OAKSmini 用定義ファイル
#include "LCDfunc.h"                    // LCD 表示関数をインクルード
char buff[64];                          // グローバル変数にバッファを確保

void main(void) {
    unsigned int ad0_result;            // AD0 変換結果
    int i, n;

// LCD の初期化
    LCD_init();                         // LCD 初期設定
    LCD_cls();                          // 全て消去

// AD 変換の初期化
                                        // 10bit 分解能/SH あり
                                        // φ AD = 10MHz
                                        // 単発モード/ソフトウェアトリガ
                                        // 変換時間 = 100nsec × 33 = 3.3usec
    adst = 0;          // AD 変換停止
    adic = 0x00;       // AD 割り込みレベル = 0(割り込みを使わない)

    adcon0 = 0x80;      // bit2,1,0:     000: アナログ入力端子 AN0 を選択
                       // bit4,3:         00: アナログ入力モード＝単発モード
                       // bit5:            0: ソフトウエアトリガ選択
                       // bit6:            0: AD 変換停止
                       // bit7:            1: AD 変換動作周波数 fAD/2 を選択
    adcon1 = 0x28;      // bit1,0:         00: AD 掃引端子 AN0,AN1 を選択 (単発モード：無効)
                       // bit2:            0: AD 動作モード選択繰り返し掃引モード 1 以外
                       // bit3:            1: 分解能選択 10 ビット選択
                       // bit4:            0: AD 変換動作周波数 fAD/2 または fAD/4 を選択
```

[25] 様々な都合で、ボリュームをどのように回しても、0 から 1023 までの全ての値を表示することはできないかもしれません。

```
                    // bit5:            1: Vref 接続：接続
                    // bit7,6:          00: ANEX0,ANEX1 は使用しない
    adcon2 = 0x01;    // bit0:          1: サンプル＆ホールド有り

    while ( 1 ) {
        n = 10;                         // n 回の結果を平均する
        ad0_result = 0;                 // AD 変換結果保存
        for ( i = 0; i < n; i++ ) {
            adst = 1;                 // AD 変換開始
            while ( ir_adic == 0 ) {}     // AD 変換終了を待つ (割り込み要求ビットをチェック)
            ir_adic = 0;              // 割り込み要求ビットをクリア
            ad0_result += ad0;         // AD 変換結果保存
        }
        ad0_result = ad0_result / n;
        sprintf( buff, "AD=%4d", ad0_result );

        LCD_locate( 0, 0, 0, 0 );    // カーソル OFF、点滅しない
        LCD_print_str( buff );         // 数値で表示

        LCD_locate( 0, 1, 0, 0 );    // カーソル OFF、点滅しない
        for ( i = 0; i <= ad0_result/64; i++ ) {
            LCD_print( 0xff );         // バーグラフで表示
        }
        for ( i = 1 + ad0_result/64; i < 16; i++ ) {
            LCD_print( ' ' );
        }
    }
}
```

5.2.2 AD 変換結果で LED の明るさを変える

(2) 「プロジェクトを開く」ボタンをクリックして H:￥smaple￥sec04￥sample2.tmk を開きます。

(3) 「リビルド」ボタンをクリックして、全てコンパイルします。時々、「製品版を購入しなさい」という勧告メッセージが表示されて動作が停止しますが、そのまま待ちます。コンパイルが終了すると、Builder ウィンドウに、「******** Finish... 」と出ます。

(4) 「デバッガの起動」ボタンをクリックして、リモートデバッガ (KD30) を起動します。

注意：起動しないこともあります。そのときは、「ツールの登録」ボタンをクリックして「DEBUG TOOL」の「KD30」をチェックしてから、「デバッガの起動」ボタンをクリックします。

(5) リモートデバッガの [File]-[Download]-[Load Module...] で、H:￥smaple￥sec04￥sample2.x30 を開きます。

(6) リモートデバッガの「↓ Go」ボタンをクリックします。

このサンプルプログラムは、右側のボリュームの回転角度に応じて約 0〜5[V] に変化する電圧の AD 変換結果を、LCD にバーグラフで表示します。同時に、フルカラー LED（赤）の明るさが連続的（約 1000 段階）に変化します。ボリュームのつまみを右に回すほど、LED は明るく点灯します。

フルカラー LED の明るさは、パルス幅変調 (PWM) 出力によって点灯している時間を変化させることによって変えています。

```
//=====================================================
//     内容： AD 変換結果を LCD に表示
//               右側の VR の回転角度がわかる
//                フルカラー LED 赤の明るさを PWM で変更する
```

```
// 0402.c
//=========================================================
#include <stdio.h>                      // sprintf() を使うため
#include "sfr26.h"                      // OAKSmini 用定義ファイル
#include "LCDfunc.h"                    // LCD 表示関数をインクルード
char buff[64];                          // グローバル変数にバッファを確保

void main(void) {
    unsigned int ad0_result;            // AD0 変換結果
    int i, n;
    unsigned int temp_ta0;

// フルカラー LED 接続ポートの初期化
    pd7 = 0xff;                         // ポート 7 方向レジスタ=出力
    p7 = ~0x00;                         // LED 消灯

// LCD の初期化
    LCD_init();                         // LCD 初期設定
    LCD_cls();                          // 全て消去

// AD 変換の初期化
                                        // 10bit 分解能/SH あり
                                        // φ AD = 10MHz
                                        // 単発モード/ソフトウェアトリガ
                                        // 変換時間 = 100nsec × 33 = 3.3usec
    adst = 0;           // AD 変換停止
    adic = 0x00;        // AD 割り込みレベル = 0(割り込みを使わない)

    adcon0 = 0x80;      // bit2,1,0:    000: アナログ入力端子 AN0 を選択
                        // bit4,3:      00: アナログ入力モード=単発モード
                        // bit5:        0: ソフトウエアトリガ選択
                        // bit6:        0: AD 変換停止
                        // bit7:        1: AD 変換動作周波数 fAD/2 を選択
    adcon1 = 0x28;      // bit1,0:      00: AD 掃引端子 AN0,AN1 を選択 (単発モード：無効)
                        // bit2:        0: AD 動作モード選択繰り返し掃引モード 1 以外
                        // bit3:        1: 分解能選択 10 ビット選択
                        // bit4:        0: AD 変換動作周波数 fAD/2 または fAD/4 を選択
                        // bit5:        1: Vref 接続：接続
                        // bit7,6:      00: ANEX0,ANEX1 は使用しない
    adcon2 = 0x01;      // bit0:        1: サンプル&ホールド有り

// タイマ A0 の初期化
    udf = 0x00;         // ダウンカウントに設定
    trgsr = 0xff;       // オーバフローまたはアンダーフローを選択
    ta0ic = 0x00;       // TA0 割り込みを使用しない

    ta0mr = 0x0f;       // bit1,0:      11: PWM モード
                        // bit2:        1: PWM では 1
                        // bit4,3:      01: ゲート機能無し (TAiIN 端子は通常のポート端子)
                        // bit5:        0: 16bit モード
                        // bit7,6:      00: カウントソース = f1
                                        // 周期=3.277msec
    ta0 = 0x00ff;                       // タイマ値の初期化
    tabsr = 0x01;                       // TA0 カウント開始

    while ( 1 ) {
        n = 10;                         // n 回の結果を平均する
        ad0_result = 0;                 // AD 変換結果保存
        for ( i = 0; i < n; i++ ) {
            adst = 1;                   // AD 変換開始
            while ( ir_adic == 0 ) {}   // AD 変換終了を待つ (割り込み要求ビットをチェック)
            ir_adic = 0;                // 割り込み要求ビットをクリア
```

```
            ad0_result += ad0;          // AD 変換結果保存
        }
        ad0_result = ad0_result / n;
        sprintf( buff, "AD=%4d", ad0_result );

        LCD_locate( 0, 0, 0, 0 );     // カーソル OFF、点滅しない
        LCD_print_str( buff );          // 数値で表示

        LCD_locate( 0, 1, 0, 0 );     // カーソル OFF、点滅しない
        for ( i = 0; i <= ad0_result/64; i++ ) {
            LCD_print( 0xff );          // バーグラフで表示
        }
        for ( i = 1 + ad0_result/64; i < 16; i++ ) {
            LCD_print( ' ' );
        }

        temp_ta0 =  ~(ad0_result << 6) ;     // AD 変換結果の下位 10bit → 16bit にシフト
                                    // 右回転で明るくなるように、その結果を反転
        if ( temp_ta0 >= 0xffff ) {
            temp_ta0 = 0xfffe;          // TA0 設定可能範囲に補正
        }
        ta0 = temp_ta0;
    }
}
```

5.3　AD 変換

ここでは、まず AD 変換の仕組みを説明します。次に、実習で使うマイクロコントローラに搭載されている AD 変換器の制御方法を説明します。

5.3.1　AD 変換器の仕組みと動作

AD 変換は、アナログ量をデジタルデータに量子化する変換です。アナログ量は、いくらでも細かな変化を表現することができますが、デジタルデータでは表現できる最小変化量が決まっています。

例えば、1 ビットで変数を表現すると、0 と 1 の値を表すことができます。0 から 5[V] の電圧値を 1 ビットのデジタルデータで表現すると、0 から 2.5[V] までは 0、2.5 から 5[V] までは 1 で表すことができます（論理回路のようなもの）。このような 1 ビット AD 変換は、次のような電気回路で実現できます。

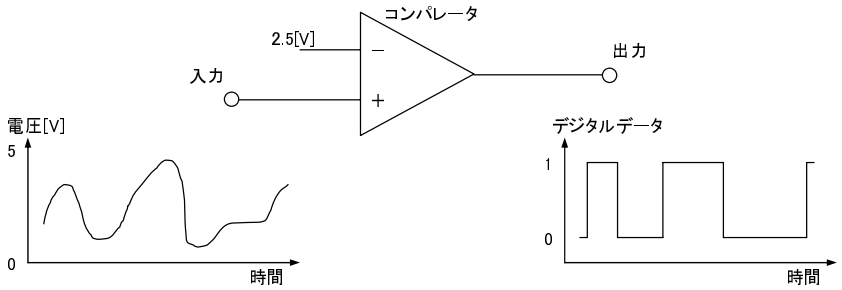


図. 5.1: 1 ビット AD 変換器。

この方式をそのまま大規模化して、8 ビットや 10 ビットの AD 変換器を構成する方式が、**フラッシュ型（並列比較型）**AD 変換器です。フラッシュ型 AD 変換器では、量子化分解能に対応するそれぞれの電圧と入力を比較するので、量子化ビット数 n に対して、$2^n - 1$ 個のコンパレータが必要になります。つまり、10 ビット AD 変換器では、1023 個のコンパレータが必要になりますが、入力のアナログ信号をたいへん高速にデジタルデータに変換できるので、ビデオ信号などの広帯域アナログ信号用の AD 変換器として広く利用されています。

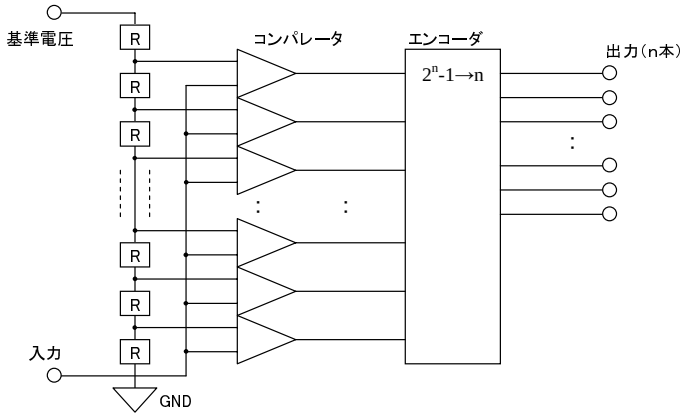


図. 5.2: フラッシュ型 AD 変換器。

コンパレータをたくさん使うフラッシュ型に対して、1 個のコンパレータと 1 個の DA 変換器を使って、上位ビットから逐次デジタルデータを決定する方式の AD 変換器が**逐次比較型** AD 変換器です。実習で使用す

るマイクロコントローラには、10 ビットの逐次比較型 AD 変換器が 1 個搭載されています。

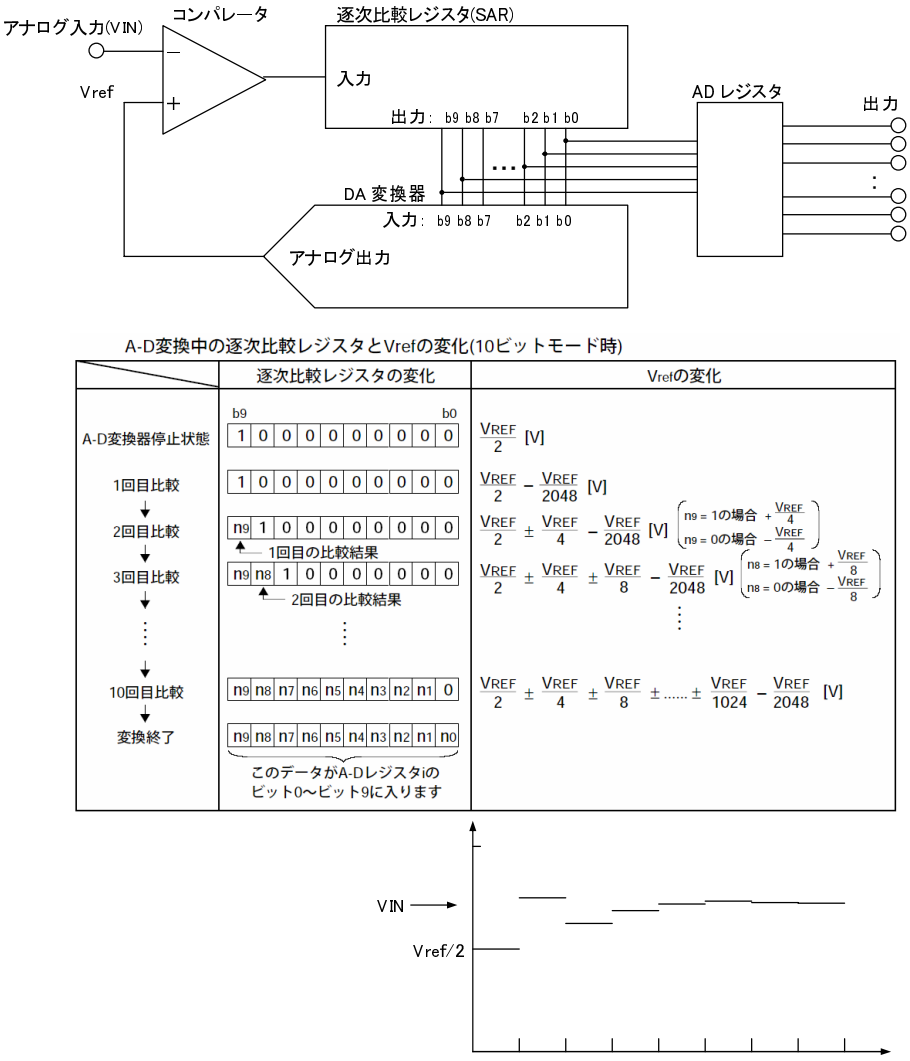


図. 5.3: 逐次変換型 AD 変換器。

逐次比較型 AD 変換器は、逐次比較レジスタ (SAR；Succesive Approximation Register) に保持されている値に応じて内部の DA 変換器で生成される比較電圧 (Vref) と、アナログ入力端子の入力電圧 (VIN) を比較し、その結果を逐次比較レジスタに反映することによって、VIN をデジタルデータに変換します。

AD 変換開始コマンドを検出すると、次のような一連の処理を行います（10 ビット構成の場合)。

(1) ビット 9（最上位ビット）の決定

まず、逐次比較レジスタに [1000000000] をセットして、Vref と VIN を比較します。コンパレータでの比較結果によって、Vref ＜ VIN なら「1」、Vref ＞ VIN なら「0」に決定します。

(2) ビット 8 の決定

逐次比較レジスタに [$b_9$100000000]（b_9 は (1) で決定した最上位ビット）をセットして、Vref と VIN を比較します。コンパレータでの比較結果によって、Vref ＜ VIN なら「1」、Vref ＞ VIN なら「0」に決定します。

(3) 以下、ビット 0 まで同様

逐次変換型 AD 変換器は、回路規模が小さく高精度ですが、一連の処理の間に入力電圧が変化してしまうと変換結果に大きな誤差が入ります。このため、入力にはサンプルホールド回路を使って、変換期間中に入力電圧が変化しないようにします。

5.3.2 標本化/量子化と誤差

アナログ入力端子に入力した電圧は 1 次元信号と考えることができます。この 1 次元信号（電圧）を、一定の時間間隔でデジタルデータとして得ることが、標本化/量子化です。実習で使うマイクロコントローラの場合には、約 3.3[μ sec] 周期で 10 ビットに量子化をすることができます。

問題： 標本化周波数の半分の周波数まで記録/再生が可能とすると、その周波数は？
答え： _____

また、回路上の特性や電源などに含まれるノイズの影響で、AD 変換結果には誤差が含まれます。実習で使うマイクロコントローラのデータシートでは、絶対精度と微分非直線性誤差が規定されています。以下は、アプリケーションノート『AD 変換の方法と精度』(rjj05b0309_m16cap.pdf) からの転載です。

絶対精度： 理論的 AD 変換特性における出力コードと、実際の AD 変換結果の差が絶対精度です。絶対精度測定時は、理論的 AD 変換特性において同じ出力コードを期待できるアナログ入力電圧の幅（1LSB 幅）の中点の電圧を、アナログ入力電圧として使用します。例えば分解能 10 ビット、基準電圧 (VREF)=5.12[V] の場合、1LSB 幅は 5[mV] で、アナログ入力電圧には 0、5、10、15、20[mV]…を使用します。絶対精度＝±3LSB とは、アナログ入力電圧が 25[mV] の場合、理論的 AD 変換特性では出力コード 005h が期待できますが、実際の AD 変換結果は 002h～008h になることを意味します。絶対精度は、ゼロ誤差とフルスケール誤差

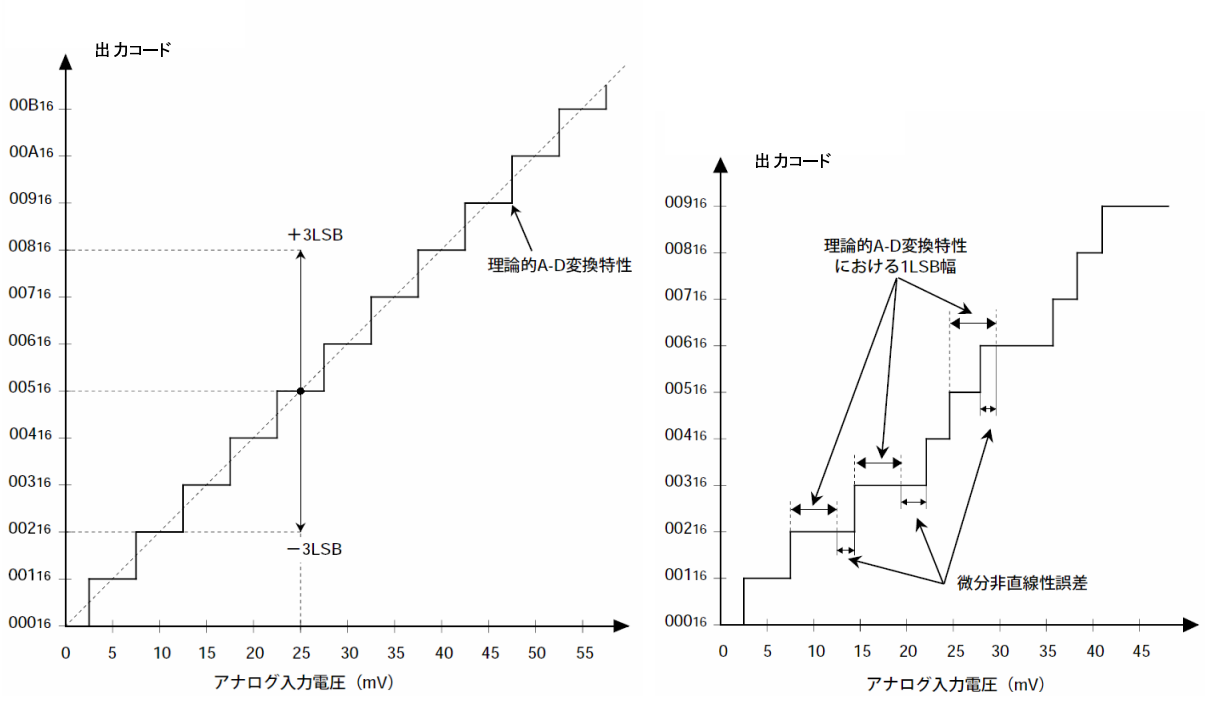


図. 5.4: 絶対精度（左）と微分非直線性誤差（右）。

微分非直線性誤差： 微分非直線性誤差は、理論的 AD 変換特性における 1LSB 幅（同じ出力コードを期待できるアナログ入力電圧の幅）と、実測定される 1LSB 幅（同じコードを出力するアナログ入力電圧の幅）の差を示すものです。分解能 10 ビット、基準電圧 (VREF)=5.12[V] の場合、微分非直線性誤差=± 1LSB ならば、理論的 AD 変換特性における 1LSB 幅は 5[mV] ですが、実測定される 1LSB 幅は 0〜10[mV] になることを意味します。

データシートでは積分非直線性を規定しています。積分非直線性は、入力電圧のフルスケールに対する出力コードのフルスケールを結ぶ直線に対する、実際の変換結果の最大の誤差です。

5.3.3 アナログ入力への接続

実習で使うリモコン基板には、ボリュームが 3 個、パルス入力ダイアルが 2 個搭載されています。この 3 個のボリュームは、約 0〜5[V] の電圧を、マイクロコントローラのアナログ入力 0〜2 に入力しています。下の表で、VR-a/VR-b/VR-c は、それぞれ右側/中央/左側のボリュームです。

CPU	信号名	CN1	CN2	MC 拡張基板	MC リモコン基板
45	P10-0/AN0		38	ADC / I/O-0	VR-a
43	P10-1/AN1		37	ADC / I/O-1	VR-b
42	P10-2/AN2		36	ADC / I/O-2	VR-c
41	P10-3/AN3		35	ADC / I/O-3	
40	P10-4/AN4/KI0*		34	ADC / I/O-4	トグル SW-a to GND
39	P10-5/AN5/KI1*		33	ADC / I/O-5	トグル SW-b to GND
38	P10-6/AN6/KI2*		32	ADC / I/O-6	トグル SW-c to GND
37	P10-7/AN7/KI3*		31	ADC / I/O-7	トグル SW-d to GND

5.3.4 マイクロコントローラに搭載された AD 変換器の構成

実習で使用するマイクロコントローラには、10 ビット分解能の AD 変換器が 1 個搭載されています。この AD 変換器へのアナログ入力は、8 個の接続端子から選べるようにアナログマルチプレクサを内蔵しています。8 個の信号源に直接接続することができますが、いずれかの入力を選択してから AD 変換をします[26] 。

AD 変換器は逐次比較型なので、クロックが必要になります。このクロックは、要求される AD 変換速度と消費電流に合わせて選択できるようになっています。実習では消費電流を低く抑える必要はないので、後で示す設定例では変換速度が最大になるように 10[MHz] のクロックを選択しています。

分解能を 8 ビットに設定すれば、10 ビットに設定した場合と比較して、変換時間は短くなります（33 クロックから 28 クロックに）。しかし 8 ビット分解能のときの誤差 (± 2LSB) を考えると、8 ビットモードの使用は避けた方がいいでしょう。

また、AD 変換器を使わないときにむだな電流を消費しないために、ラダー抵抗（DA 変換のための）に流す電流を切断することができます。

5.3.5 設定するレジスタ

電源の消費電流を気にしない用途では、10 ビット分解能、10[MHz] クロック、サンプル/ホールドあり、ラダー抵抗への電流は流しっぱなし、という設定をします。

```
adst = 0;        // AD 変換停止
adic = 0x00;     // AD 割り込みレベル = 0(割り込みを使わない)
adcon0 = 0x80;   // bit2,1,0:    000: アナログ入力端子 AN0 を選択
                 // bit4,3:       00: アナログ入力モード＝単発モード
                 // bit5:          0: ソフトウエアトリガ選択
                 // bit6:          0: AD 変換停止
```

[26] アナログ入力を自動的に切り替えて、複数のアナログ入力に対する AD 変換結果を得るモード（掃引モード）もある。

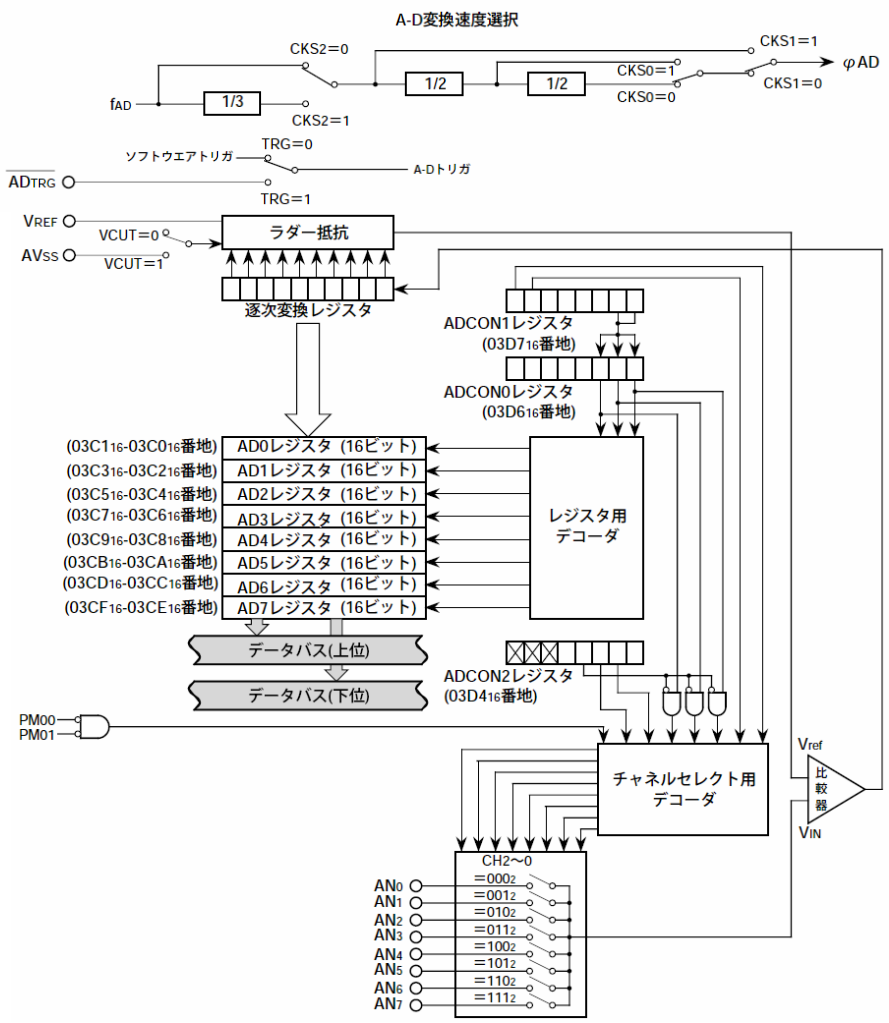


図. 5.5: AD 変換器のブロック図。

```
                // bit7:          1: AD 変換動作周波数 fAD/2 を選択
adcon1 = 0x28;  // bit1,0:       00: AD 掃引端子 AN0,AN1 を選択 (単発モード：無効)
                // bit2:          0: AD 動作モード選択繰り返し掃引モード 1 以外
                // bit3:          1: 分解能選択 10 ビット選択
                // bit4:          0: AD 変換動作周波数 fAD/2 または fAD/4 を選択
                // bit5:          1: Vref 接続：接続
                // bit7,6:       00: ANEX0,ANEX1 は使用しない
adcon2 = 0x01;  // bit0:          1: サンプル＆ホールド有り
```

AD 変換を行うときには、アナログ入力チャンネルを選択してから、AD 変換をスタートさせます。AD 変換をスタートさせるためには、用意されているヘッダファイル sfr26.h の中で定義されている adst に「1」を書き込みます（通常の変数のように代入するだけ）。AD 変換が終了したかどうかは、ir_adic が「1」になるのを待ちます。この ir_adic は、AD 変換が終了したことを示す「フラグ」で、AD 変換終了で割り込みをかける設定にしているときにはこのフラグの変化によって割り込みがかかります。AD 変換の終了を検出したら、ir_adic をクリアします。

```
adcon0 = 0x80 + ad_chan;    // ad_chan: AD 入力チャンネル (0～7)
adst = 1;                   // AD 変換開始
while ( ir_adic == 0 ) {}   // AD 変換終了を待つ (割り込み要求ビットをチェック)
```

```
ir_adic = 0;                    // 割り込み要求ビットをクリア
```

AD 変換する入力チャンネルが 1 つだけのときには、adcon0 で設定する AD 動作モードを、「繰り返しモード」に設定することで、変換結果をいつでも読み出すことができます。また、「繰り返し掃引モード」に設定すると、入力チャンネル 0〜3 までを自動的に切り替えながら繰り返し AD 変換を行い、いつでも最新の AD 変換結果を読み出すことができます。

AD 変換結果は、入力チャンネルに対応した AD レジスタを読み出します。

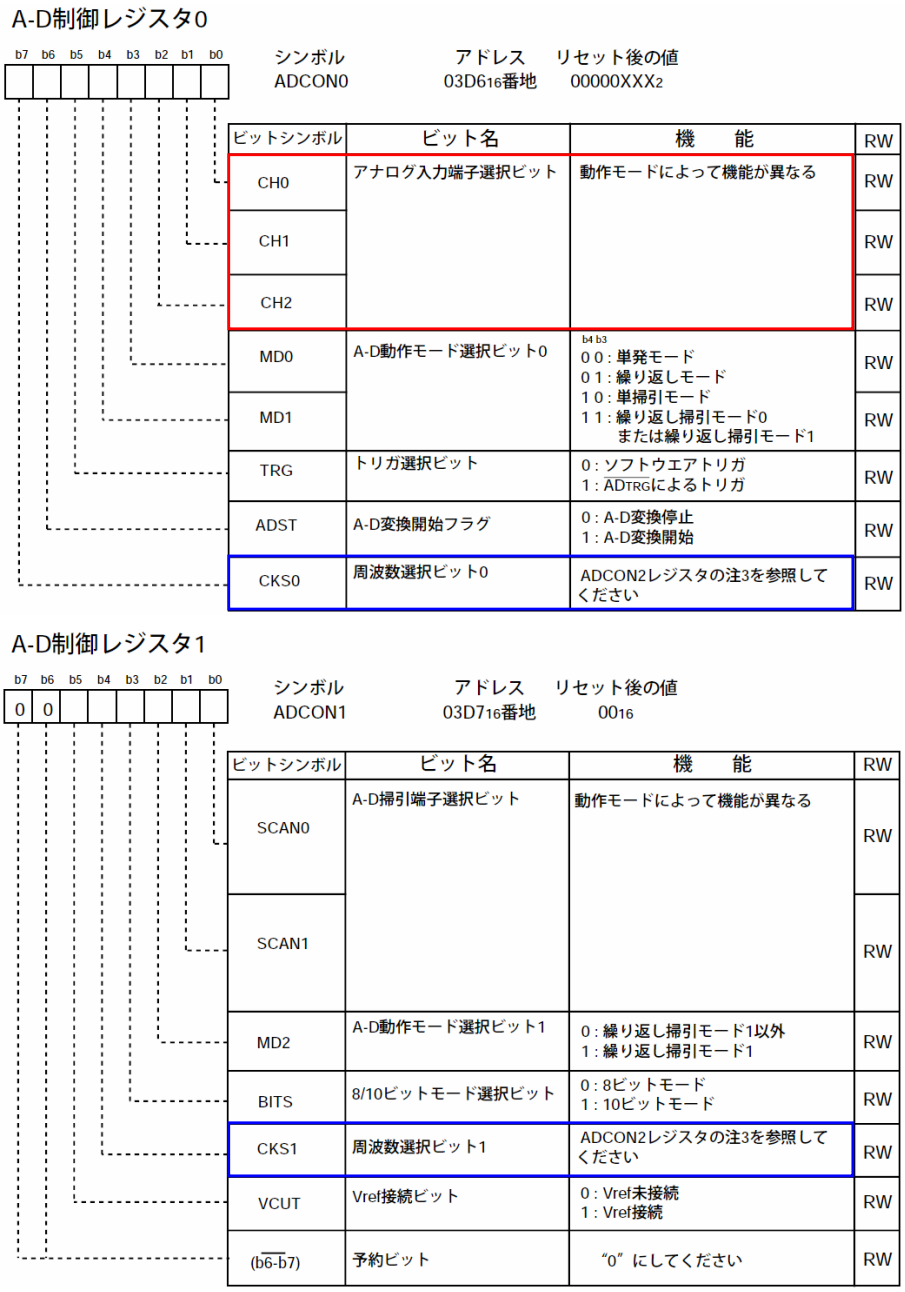
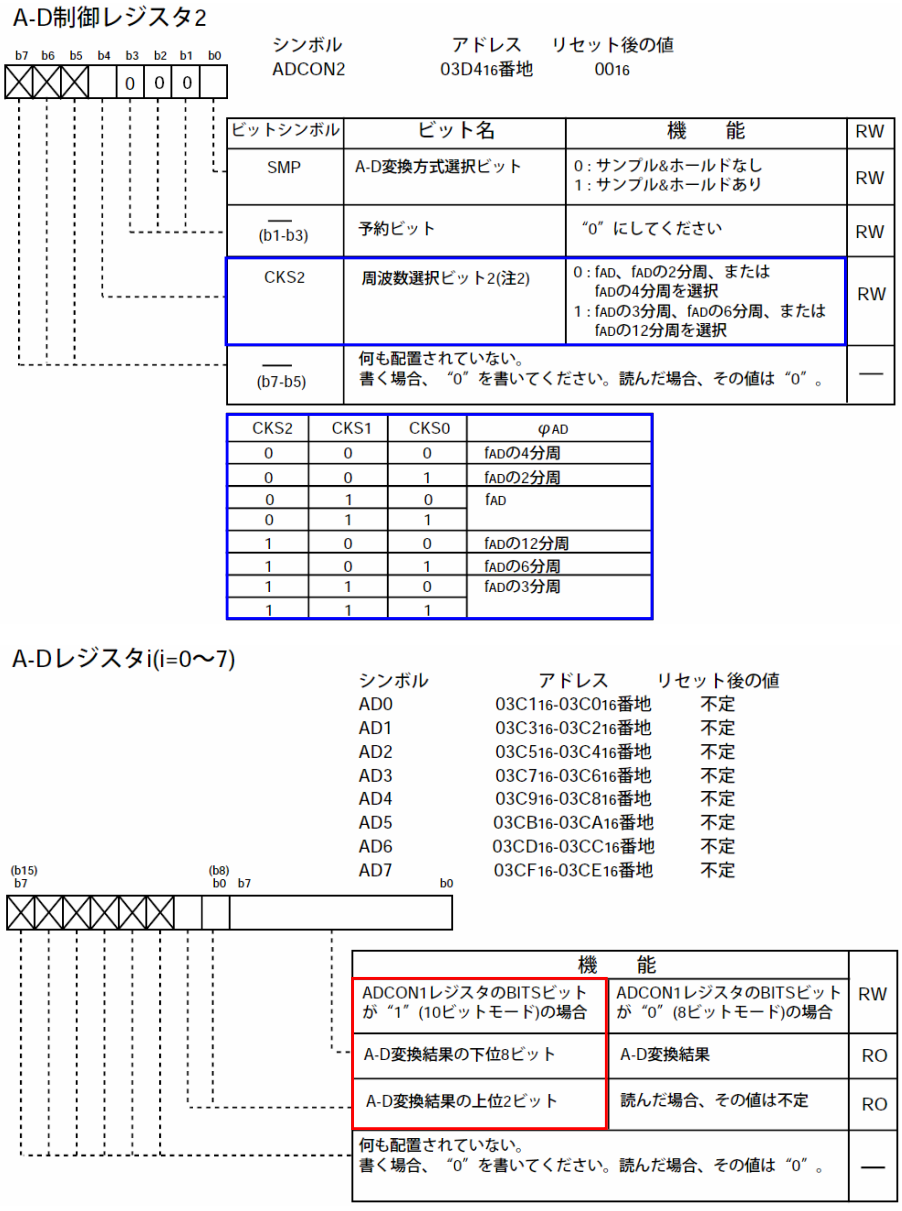


図. 5.6: AD 変換器の制御レジスタ（その 1）。



A-D制御レジスタ2

シンボル	アドレス	リセット後の値
ADCON2	03D4₁₆番地	00₁₆

b7 b6 b5 b4 b3 b2 b1 b0 : 0 0 0

ビットシンボル	ビット名	機　能	RW
SMP	A-D変換方式選択ビット	0：サンプル&ホールドなし 1：サンプル&ホールドあり	RW
— (b1-b3)	予約ビット	“0”にしてください	RW
CKS2	周波数選択ビット2(注2)	0：fAD、fADの2分周、または fADの4分周を選択 1：fADの3分周、fADの6分周、または fADの12分周を選択	RW
— (b7-b5)	何も配置されていない。 書く場合、“0”を書いてください。読んだ場合、その値は“0”。		—

CKS2	CKS1	CKS0	φAD
0	0	0	fADの4分周
0	0	1	fADの2分周
0	1	0	fAD
0	1	1	
1	0	0	fADの12分周
1	0	1	fADの6分周
1	1	0	fADの3分周
1	1	1	

A-Dレジスタi(i=0～7)

シンボル	アドレス	リセット後の値
AD0	03C1₁₆-03C0₁₆番地	不定
AD1	03C3₁₆-03C2₁₆番地	不定
AD2	03C5₁₆-03C4₁₆番地	不定
AD3	03C7₁₆-03C6₁₆番地	不定
AD4	03C9₁₆-03C8₁₆番地	不定
AD5	03CB₁₆-03CA₁₆番地	不定
AD6	03CD₁₆-03CC₁₆番地	不定
AD7	03CF₁₆-03CE₁₆番地	不定

(b15) b7 (b8) b0 b7 b0

機　能		RW
ADCON1レジスタのBITSビットが“1”(10ビットモード)の場合	ADCON1レジスタのBITSビットが“0”(8ビットモード)の場合	RW
A-D変換結果の下位8ビット	A-D変換結果	RO
A-D変換結果の上位2ビット	読んだ場合、その値は不定	RO
何も配置されていない。 書く場合、“0”を書いてください。読んだ場合、その値は“0”。		—

図. 5.7: AD 変換器の制御レジスタ（その 2）。

5.4 PWM 出力

ここでは、デジタルデータをアナログ的に出力する方法の 1 つである、PWM について説明します。次に、マイクロコントローラのタイマ機能を使った PWM 出力を行う方法を説明します。

5.4.1 デジタルデータの出力方法

マイクロコントローラで処理した結果を、アナログ量として出力したい場合があります。例えば、LED の明るさを変化させたり、モータの回転速度を変化させたりしたいことがあります。このようなときに、マイクロコントローラでは、次のような方法を多く利用します。

DA 変換： デジタルデータをアナログの電圧に変換します。基本的には、加算回路で実現することができます。図 5.8（左）に示すように、オペアンプを使った加算回路の帰還抵抗 R に対して、デジタルデータの最上位ビットに対応して入力抵抗 R を、次のビットに対して 2R を、以下順次に最下位ビットに対して 2nR をスイッチで接続します。スイッチは、電源電圧か GND に接続します。しかしこの方法では、相対的な抵抗精度が厳しく、現実的ではありません。一般的には、図 5.8（右）に示すように、R-2R ラダーと呼ばれる構成の抵抗回路を使います。

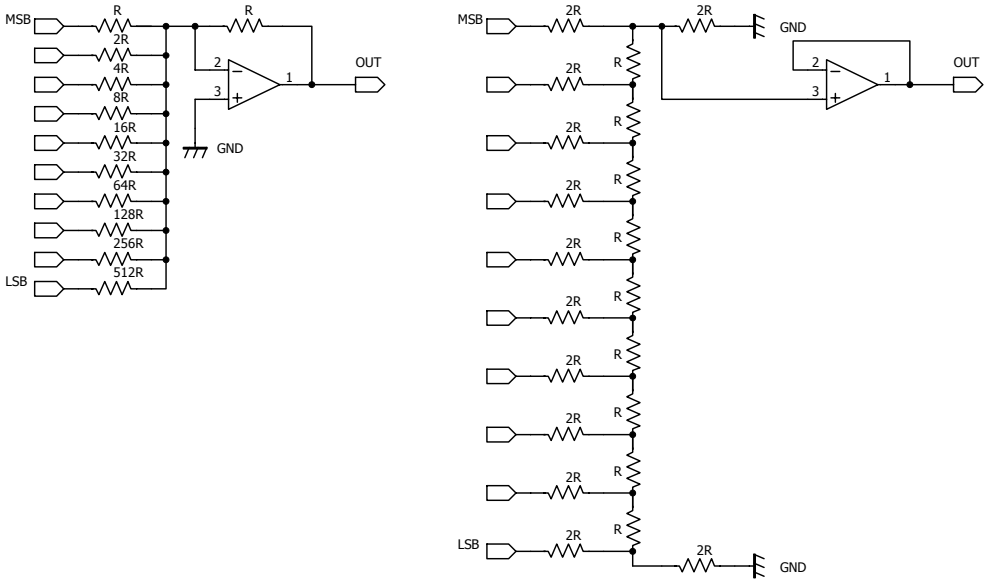


図. 5.8: DA 変換器の原理。

DA 変換器はアナログ電圧を出力するために、最終的な目的である LED の明るさやモータの回転速度を変えるための回路も、アナログ回路で構成する必要があります。アナログ回路で電力を制御すると、回路構成が複雑になり、また発熱などの問題もあります。そこで最近では、次のパルス幅変調を使って外部機器を制御する方法が一般的になっています。

パルス幅変調 (PWM; Pulse Width Modulation)： PWM では、通常の論理回路出力をそのまま利用してアナログ量を出力することができます。原理は単純で、論理「1」を出力している期間と、論理「0」を出力している期間の割合を変えるだけです。例えば、1[msec] の間「1」を出力して 9[msec] の間「0」を出

力するような波形（図 5.9）は、電圧を平均すれば「0.1」という値を出力しているのと同じです。この「0.1」は、もちろんデジタルデータではありません。

PWM では、波形の周期は一定にすることがほとんどで、パルスの幅（デューティという）だけを変えます。この波形をモータの回転速度制御に使うということは、例えば「1」の期間だけモータに通電して、「0」の期間は通電しない、ということに相当します。「1」の期間の割合が大きいほどモータは高速で回転することは、直感的に理解できます。

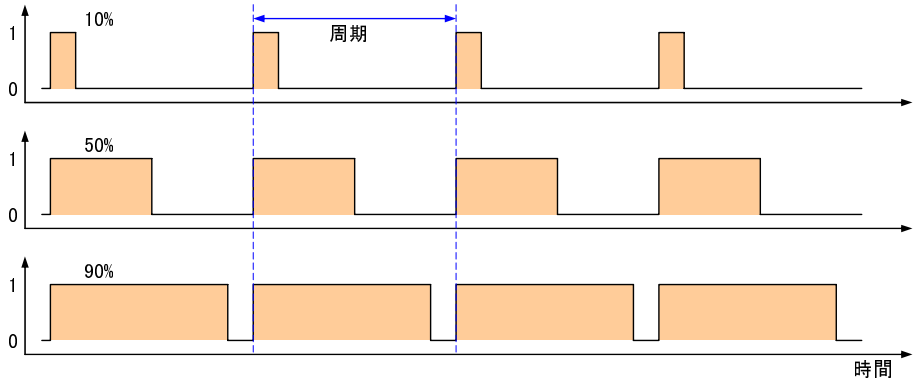


図. 5.9: パルス幅変調の例。

5.4.2 タイマの PWM モード

実習で使用するマイクロコントローラに搭載されているタイマ A には、PWM 出力モードがあります。このモードでは、16 ビット PWM または 8 ビット PWM としてカウンタを利用することができます。図 5.10 に、PWM モードのときの `TAiMR` レジスタ (i=0〜4) と、16 ビット PWM の動作例示します。

5.4.3 PWM 出力への接続

タイマ A0/A1/A2 が、それぞれフルカラー LED の赤/緑/青に接続しています。同時に、ラジコン用サーボ 0/1/2 の制御端子にも出力されています。ラジコン用サーボの制御方法は、次章で説明します。

CPU	信号名	CN1	CN2	MC 拡張基板	MC リモコン基板
25	P7-0/TA0out/ TXD2/SDA	13		JP to RCS-0 / Serial / H-br-0 PWM	フルカラー LED-Red
23	P7-2/TA1out/ V/CLK2	11		JP to RCS-1 / Serial / H-br-1 PWM	フルカラー LED-Green
21	P7-4/TA2out/W	9		JP to RCS-2 / H-br-2 PWM	フルカラー LED-Blue

5.4.4 設定例

次に、16 ビットモード PWM の設定例を示します。「16 ビットモード」は、カウンタを 16 ビットカウンタとして利用するという意味です。PWM 波形の周期は、(クロック周期) × $(2^{16} - 1)$ になります。この場合には、20[MHz] クロックをカウントソースに設定しているので、PWM 波形の周期は 3.2768[msec] になります。

タイマレジスタに設定する 16 ビットの数値の期間だけ、PWM 波形が「1」になります。設定できる 16 ビットの数値は、`0x0000`〜`0xfffe`(65535) までです。`0xffff` を設定すると PWM が停止してしまうため、サン

プルプログラムでは一時変数に値を用意してから範囲のチェックをしています。もしも 0xffff を設定して PWM が停止してしまったら、PWM を再起動するためには電源を再投入するか CPU リセットを行う必要があります。

```
udf = 0x00;     // ダウンカウントに設定
trgsr = 0xff;   // オーバフローまたはアンダーフローを選択
ta0ic = 0x00;   // TA0 割り込みを使用しない
ta0mr = 0x0f;   // bit1,0:      11: PWM モード
                // bit2:         1: PWM では 1
                // bit4,3:      01: ゲート機能無し (TAiIN 端子は通常のポート端子)
                // bit5:         0: 16bit モード
                // bit7,6:      00: カウントソース = f1
ta0 = 0x00ff;   // タイマ値の初期化
tabsr = 0x01;   // TA0 カウント開始
```

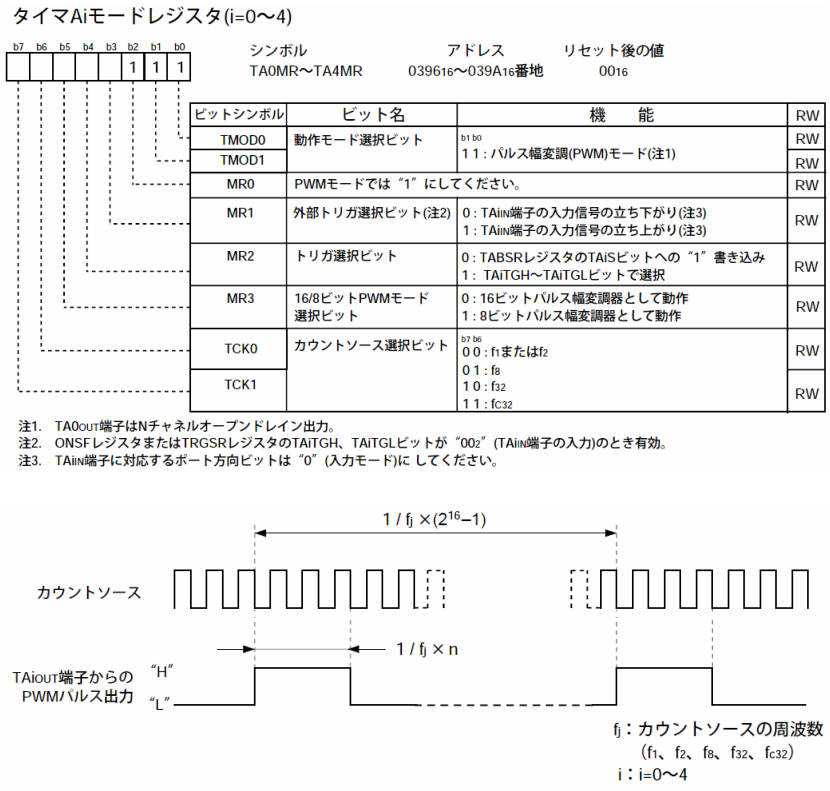


図. 5.10: タイマモードレジスタと PWM 出力。

5.5 課題

リモコン基板の 3 個のボリュームで、フルカラー LED の「色相 (H)」「彩度 (S)」「明るさ (I)」を変えてみよう。

色相 (H) は、どのような色なのかを表します。例えば、赤、青、緑などです。再度 (S) は、色の濃さを表します。同じ赤でも、鮮やかな赤のときには彩度が高く、白に近い赤のときには彩度が低いといいます。明るさ (I) は、文字通り明るさを表します。

ボリュームの回転角度で、マイクロコントローラのアナログ入力端子の電圧が変化します。この電圧は、AN0〜AN2 の電圧を AD 変換することで得ることができます。10 ビットモードで AD 変換した結果は、0〜1023 です。この AD 変換結果を、次の **HSI 色空間の円柱モデル**の変換式 (5.2) のスケールに変換します。もちろん、浮動小数点型を使います。

式 (5.2) で変換した RGB の値を 16 ビット PWM 出力の設定値範囲 1〜$2^{16}-1$ のスケールに変換して、フルカラー LED の赤 (R)、緑 (G)、青 (B) の明るさを変化させます。タイマレジスタに `0xffff` を代入すると、PWM 出力が停止してしまうので注意してください。

プログラムで三角関数を利用するときには、`math.h` をインクルードしてください。また、どのような入力が、どのような出力に変換されているのかを確認できるように、LCD に HSI と RGB の値を表示してください。

```
        1--------0----5
HSI->   3.14 0.40 2.22
RGB->   0.29 0.50 0.21
        1--------0----5
```

5.5.1 HSI 色空間の円柱モデル

図 5.11 に、**HSI 色空間の円柱モデル**の概念図を示します。このモデルは、RGB 座標系を黒の位置 $(0,0,0)$ と白の位置 $(1,1,1)$ を結ぶ色線を軸とする円柱座標系に変換するものです。

このモデルの HSI 変換は、つぎのように表すことができます。

$$\begin{aligned} H &= \tan^{-1}\frac{G-B}{2R-G-B} \\ S &= \frac{(B-R)^2+(R-G)^2+(G-B)^2}{3} \\ I &= R+G+B \end{aligned} \tag{5.1}$$

逆変換は 、つぎのように表すことができます。

$$\begin{aligned} R &= I/3+2S\cos H \\ G &= I/3-S\cos H+S\sin H \\ B &= I/3-S\cos H-S\sin H \end{aligned} \tag{5.2}$$

ただし、$0\le R,G,B\le 1$、$0\le I\le 3$、$0\le H\le 2\pi$、$0\le S\le 2/3$ の値を考えます。

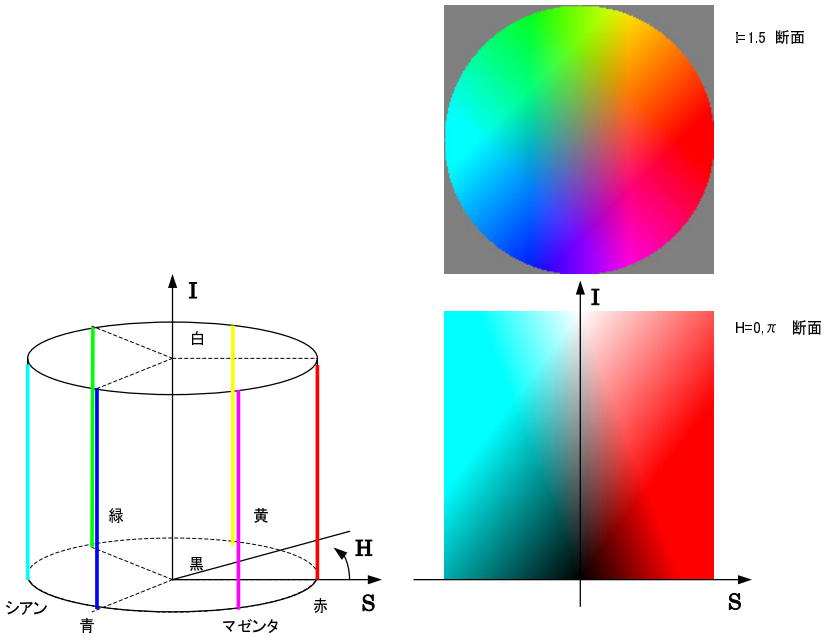


図. 5.11: 円柱モデルの概念図（左）と HSI 逆変換による HSI 空間の色彩表現（右）。

6　チャレンジテーマ用リモコン

この章の目次：

6.1　目的

この章では、チャレンジテーマに取り組むために必要な、いくつかの重要な項目を説明します。まず、チャレンジテーマにも応用できる、ロータリーエンコーダの信号を入力するためのタイマの設定方法を説明します。次に、ラジコン用サーボの駆動方法やデバッガなしにマイコン基板を単独で動作させる方法などを説明します。また、実際に独自のプログラムを開発する上で役に立つ、デバッガの利用方法を説明します。

6.2　ロータリーエンコーダ

6.2.1　エンコーダの原理

回転角度と回転方向を検出する機構を、ロータリーエンコーダといいます。図 6.1 に示すように、コンピュータの GUI のためのボール式マウスに使われていて、誰でも日常的に利用している仕組みです。より高精度なロータリーエンコーダが、産業用ロボットの関節角度の検出などに利用されています。

ポテンショメータ（電気抵抗が回転角度に応じて変化するボリューム）を使っても回転角度を検出できます。しかし、マウスのように、どれだけの距離を移動するか予測できず、入力角度範囲が決められない装置では、ロータリーエンコーダを使う必要があります。また、図 6.2 に示すように、ロータリーエンコーダは、角度検出機構に機械的な接触部品がないため、信頼性が高く長寿命な入力機構です。最初からデジタル信号で回転角度を表している点も、大きな利点です。

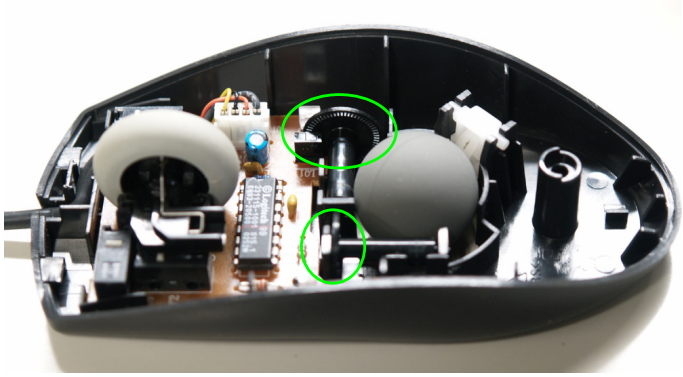
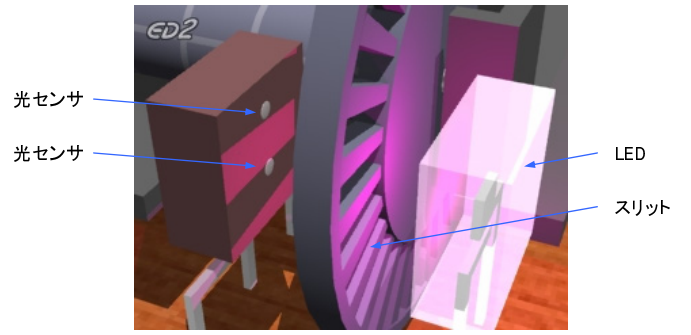


図. 6.1: マウスにはロータリーエンコーダが使われている。



http://kyoiku-gakka.u-sacred-heart.ac.jp/jyouhou-kiki/sozai/1302/index.html 「情報機器と情報社 会のしくみ」

図. 6.2: ロータリーエンコーダの構成例。

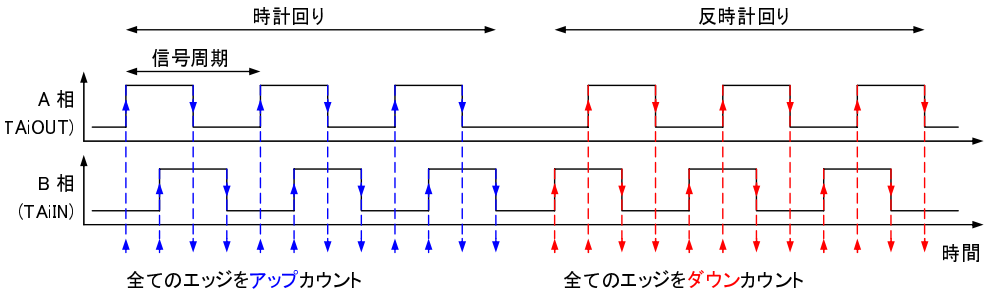


図. 6.3: A 相信号と B 相信号。

ロータリーエンコーダでは、2 個のセンサを利用します。2 個のセンサ出力を、A 相/B 相といいます。図 6.2 の例では、回転スリットの周期よりも 2 個のセンサの間隔が狭くなるように配置されていて、回転スリットの回転に伴って A 相/B 相信号の位相が異なります。基本的には、この位相で回転方向を、センサが出力するパルス数で回転角度を検出します。回転角度は相対的な角度しか検出できないので、絶対角度が必要な

ときにはリミットスイッチなどと組み合わせます。

回転角度の検出精度を向上するために、図 6.3 に示すような 4 逓倍カウントを利用することがあります。4 逓倍カウントでは、センサのパルス数ではなくパルスの変化を全てカウントします。位相の異なる 2 つのパルスの変化を使えば 4 倍のカウントになるので、1 つのセンサが出力するパルス数を数える方式に対して、1/4 の回転角度変化を検出できるようになります。

実習で利用するリモコン基板には、1 回転で 24 パルスを出力する接触式のロータリーエンコーダを 2 個搭載しています。4 逓倍で回転角度を検出すれば、96[カウント]/1[回転]、つまり 3.75[度]/1[パルス] の角度検出ができます[27] 。

6.2.2 接続しているポート

リモコン基板右側のエンコーダ（エンコーダ a）はタイマ A3 に、左側のエンコーダ（エンコーダ b）はタイマ A4 に接続しています。

タイマ A2、A3、A4 は、A 相/B 相の外部信号をカウントできます。さらにタイマ A3 と A4 は、4 逓倍のカウントが可能です。このときの A 相/B 相の入力は、次の表に示すように、TAk-out を A 相に、TAk-in を B 相に (k=2, 3) 接続します。TAk-out 端子が「H」の期間に TAk-in 端子に立ち上がりパルスが入力する位相関係のときにはアップカウント、TAk-out 端子が「H」の期間に TAk-in 端子に立ち下がりパルスが入力する位相関係のときにダウンカウントします。

CPU	信号名	CN1	CN2	MC 拡張基板	MC リモコン基板
19	P7-6/TA3out	7		JP to RCS-3 / Enc-0 A (PhC)	エンコーダ a A
18	P7-7/TA3in	6		JP to Enc-0 B (PhC)	エンコーダ a B
17	P8-0/TA4out/U	5		JP to RCS-4 / Enc-1 A (PhC)	エンコーダ b A
16	P8-1/TA4in/U*	4		JP to Enc-1 B (PhC)	エンコーダ b B

6.2.3 タイマの設定

外部からの入力信号をカウントする、イベントカウンタモードに設定します。A 相/B 相のパルス信号をカウントするときには、次の例のように `udf` レジスタと `trgst` レジスタを設定します。また、ポートの入出力方向を「入力」に設定する必要があります。ポート 8 の一部は、LCD の制御信号として必ず出力に設定する必要があるので、その信号までも不用意に入力に設定しないように注意します。

```
// タイマ A3 の初期化
    ta3ic = 0x00;                   // TA0 割り込みを使用しない
    ta3mr = 0xd1;   // bit1,0:      01: イベントカウンタモード
                    // bit2:         0:
                    // bit4,3:      10:
                    // bit5:         0:
                    // bit6:         1: フリーラン
                    // bit7:         1: 4 逓倍
// タイマ A4 の初期化
    ta4ic = 0x00;                   // TA0 割り込みを使用しない
    ta4mr = 0xd1;   // bit1,0:      01: イベントカウンタモード
                    // bit2:         0:
                    // bit4,3:      10:
                    // bit5:         0:
                    // bit6:         1: フリーラン
                    // bit7:         1: 4 逓倍
```

[27] 接触式なので、パルスのエッジにはノイズがあります。このため、多少のカウント誤差があります。

```
// イベントカウンタモードとしての設定
    udf = 0xc0;                     // 2 相パルス入力を許可
    trgsr = 0x00;                   // TA1-TA4: 全て 2 相パルス入力
    pd7 = 0x3f;                     // P7-6,7: 入力
    pd8 = 0xfc;                     // P8-0,1: 入力。LCD でも使っているので注意！
    tabsr = 0x18;                   // TA3 と TA4 カウント開始
```

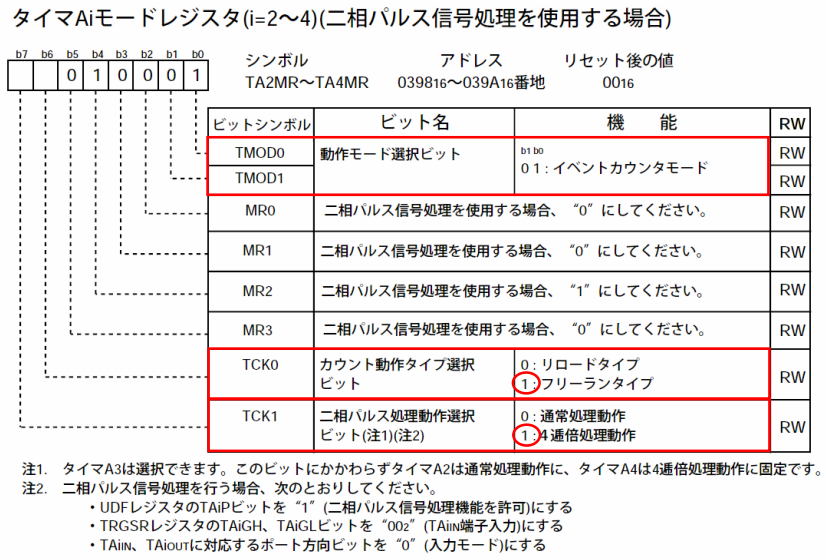


図. 6.4: 2 相パルスをカウントするときのモードレジスタ。

6.2.4 サンプルプログラム

このサンプルプログラムでは、左右のエンコーダを 4 逓倍でカウントして、それぞれ LCD に表示します。タイマの設定の後は、それぞれのタイマレジスタの内容を読み出すだけです。

```
//=========================================================
// 左右のエンコーダのカウント値を LCD に表示
// 0501.c
//=========================================================
#include <stdio.h>                  // sprintf() を使うため
#include "sfr26.h"                  // OAKSmini 用定義ファイル
#include "LCDfunc.h"                // LCD 表示関数をインクルード
char buff[64];                      // グローバル変数にバッファを確保

void main(void) {
    unsigned int count3, count4;    // カウンタ

// LCD の初期化
    LCD_init();                     // LCD 初期設定
    LCD_cls();                      // 全て消去
    LCD_locate( 0, 1, 0, 0 );       // カーソル OFF、点滅しない

// タイマ A3 の初期化
    ta3ic = 0x00;                   // TA0 割り込みを使用しない
    ta3mr = 0xd1;     // bit1,0:     01: イベントカウンタモード
                      // bit2:       0:
                      // bit4,3:     10:
```

```
                    // bit5:          0:
                    // bit6:          1: フリーラン
                    // bit7:          1: 4 逓倍
    ta3 = 0;

// タイマ A4 の初期化
    ta4ic = 0x00;                    // TA0 割り込みを使用しない
    ta4mr = 0xd1;    // bit1,0:       01: イベントカウンタモード
                    // bit2:          0:
                    // bit4,3:        10:
                    // bit5:          0:
                    // bit6:          1: フリーラン
                    // bit7:          1: 4 逓倍
    ta4 = 0;

// イベントカウンタモードとしての設定
    udf = 0xc0;                      // 2 相パルス入力を許可
    trgsr = 0x00;                    // TA1-TA4: 全て 2 相パルス入力
    pd7 = 0x3f;                      // P7-6,7: 入力
    pd8 = 0xfc;                      // P8-0,1: 入力。LCD でも使っているので注意！
    tabsr = 0x18;                    // TA3 と TA4 カウント開始
    p7 = ∼0x00;                      // LED 消灯

    while ( 1 ) {
        count3 = ta3;
        count4 = ta4;
        sprintf( buff, "%6d %6d", count4, count3 );
        LCD_print_str( buff );       // 数値で表示
    }
}
```

6.3 ラジコン用サーボ

6.3.1 仕組み

ラジコン用サーボは、制御用の信号で回転角度を指定できる小型アクチュエータです。もともとは、模型に搭載する小型エンジンのスロットルや、飛行機や自動車の操舵を行うためのアクチュエータでしたが、最近の製品は出力トルクが大きく強度も高いため、歩行ロボットの関節などにも利用されています。

チャレンジテーマでは、ラジコン用サーボをアクチュエータとして利用して、リンク機構で構成したロボットを実際に制作します。

ラジコン用サーボは、図 6.5 に例を示すように、DC モータ、ギア、ポテンショメータ、制御回路などで構成されています。DC モータは普通のモータで、電池を接続すれば単純に回転します。この回転数を落として出力軸に接続し、出力トルクを大きくするのがギアです。しかしこのままでは、角度指定ができないので、出力軸にポテンショメータを接続して回転角度を検出して、指定された回転角度と一致したらモータの駆動を停止するような制御回路を搭載しています。

6.3.2 制御信号

ラジコン用サーボの制御信号は、図 6.6 に示すような PWM 信号です。この PWM 信号の「H」パルス幅で、出力軸の回転角度を指定します。1.5[msec] で中立、このパルス幅に対して± 0.6[msec] でそれぞれの角度方向に出力軸は約 60[度] 回転します（出力軸に取り付ける「つの」をホーンと呼びます）。ただし、回転角度誤差が比較的大きいので、ロボットを設計するときには余裕を持った設計にします。

図. 6.5: ラジコン用サーボの解体例。

パルス周期は、16〜23[msec] です。パルス周期ごとに回転角度を制御するので、周期が長すぎるとむだ時間が大きくなります。しかし、短すぎるとラジコン用サーボ内蔵の制御回路が応答しません。前回説明した PWM 出力方法では、16 ビット PWM でカウントソースを f8 に設定したときに約 26[msec] 周期になります。この方法でもラジコン用サーボは駆動できますが、後で示すサンプルプログラムでは、パルス周期を 15[msec] に設定しています（定数を変えれば、任意の周期に設定可能）。

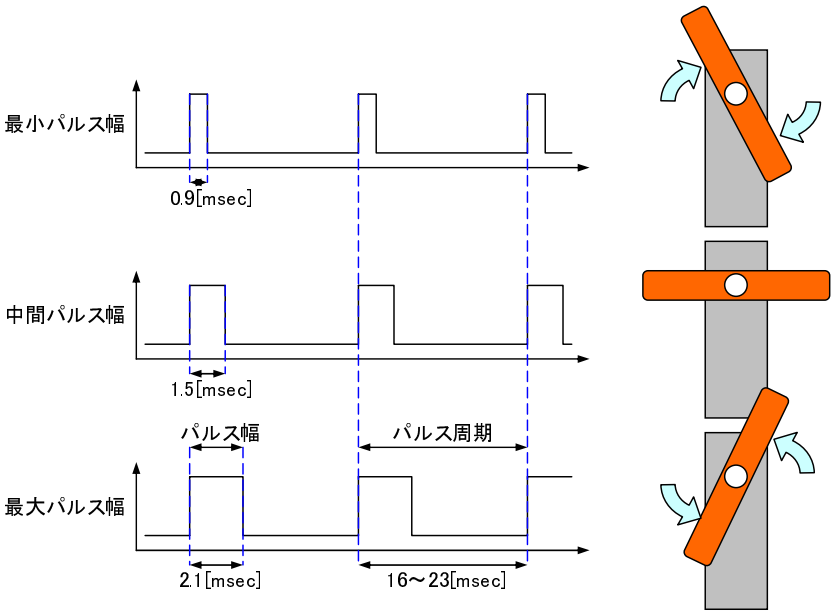


図. 6.6: ラジコン用サーボの制御信号とホーンの回転角度。

GWS 社製 Micro2BBMG の主な仕様：

駆動電圧	4.8〜7.5 [V]
回転速度	0.17[sec]/60[度]
トルク	5.4 [kgcm]
動作温度	-20〜+60 [℃]

6.3.3 接続方法

ラジコン用サーボからは、3 本の電線が出ていて、3 ピンのコネクタに接続しています。この 3 本の電線は、電源 (VCC)、接地 (GND)、制御信号です。

拡張基板の RC0〜RC2 コネクタに、図 6.7 に示すように接続してください。

RC0/RC1/RC2 の制御信号は、それぞれタイマ A0/A1/A2 の出力に接続しています。これらの制御信号は、同時に、フルカラー LED にも接続していますが、制御信号のパルス幅の変化は PWM 周期と比較して小さいので、フルカラー LED の色の変化は残念ながら識別できません。

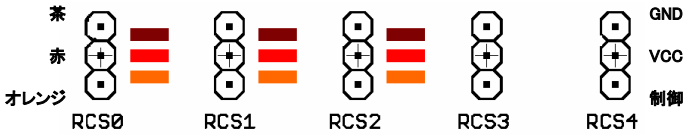


図. 6.7: ラジコン用サーボの接続コネクタ。

CPU	信号名	CN1	CN2	MC 拡張基板	MC リモコン基板
25	P7-0/TA0out/ TXD2/SDA	13		JP to RCS-0 / Serial / H-br-0 PWM	フルカラー LED-Red
23	P7-2/TA1out/ V/CLK2	11		JP to RCS-1 / Serial / H-br-1 PWM	フルカラー LED-Green
21	P7-4/TA2out/W	9		JP to RCS-2 / H-br-2 PWM	フルカラー LED-Blue

6.3.4 周期とデューティを設定できる PWM 出力

先週の「PWM 出力」で説明した 16 ビット PWM モードでは、PWM 波形の周期はカウントクロックで決まってしまいます。このため、20[MHz] クロックをカウントクロックに設定すると周期は、3.2768[msec]、次に周波数の高い f8（20[MHz] クロックの 8 分周）をカウントクロックに設定すると、26.214[msec] になります。

8 ビット PWM モードでは PWM 波形の周期もある程度設定できますが、制御信号のパルス幅の分解能が低く、ホーンの角度を何種類かに設定できる程度になってしまいます。

そこで、分解能 16 ビットで任意の周期とデューティを設定できる PWM を構成するために、タイマ B2 を使います。タイマ A0〜A4 をワンショットタイマモードに設定して、ワンショットパルスのトリガをタイマ B2 のアンダーフローに設定することで、このような PWM 波形を出力することができます。ただし、周期はタイマ A0〜A4 で全て同じです。ワンショットタイマというのは、トリガ（きっかけ）が入力されたら 1 つだけ指定したパルス幅のパルスを出力するモードです。このトリガを、タイマ B2 のアンダーフローに設定することで、結果的に繰り返しパルスを出力することになります。

次の例では、タイマ B2 のカウントソースを f8 にして周期 15[msec] のカウントダウンタイマを構成します。タイマ A0 は、カウントソース f1(20[MHz]) のワンショットタイマに設定します。後は、tabsr レジスタで使用するタイマのカウントを開始、onsf レジスタでトリガを設定します。

```
// タイマ A0 の初期化
    ta0ic = 0x00;    // TA0 割り込みを使用しない
    ta0mr = 0x16;    // bit1,0:      10: ワンショット
                     // bit2:         1: 出力あり
                     // bit4,3:      10:
                     // bit5:         0:
                     // bit7,6:      00: カウントソース = f1
    ta0 = 0x00ff;                     // タイマ値の初期化
// タイマ B2 の初期化
```

```
    tb2ic = 0x00;    // TB2 割り込みを使用しない
    tb2mr = 0x80;    // bit1,0:       00: タイマモード
                     // bit2:          0: どちらでも OK
                     // bit4,3:       00: このように設定
                     // bit5:          0: このように設定
                     // bit7,6:       10: カウントソース = f32
    tb2 = 9375-1;                     // タイマ値の初期化
                                      // 周期=15msec
// ワンショット開始とトリガ選択
    udf = 0x00;      // 全てダウンカウントに設定
    tabsr = 0x81;    // TB2 と TA0 カウント開始
    trgsr = 0x55;    // TA1-TA4: 全て TB2 のアンダーフローでトリガ
    onsf = 0x41;     // TA0: TB2 のアンダーフローでトリガ、TA0 ワンショット開始 (01000001)
```

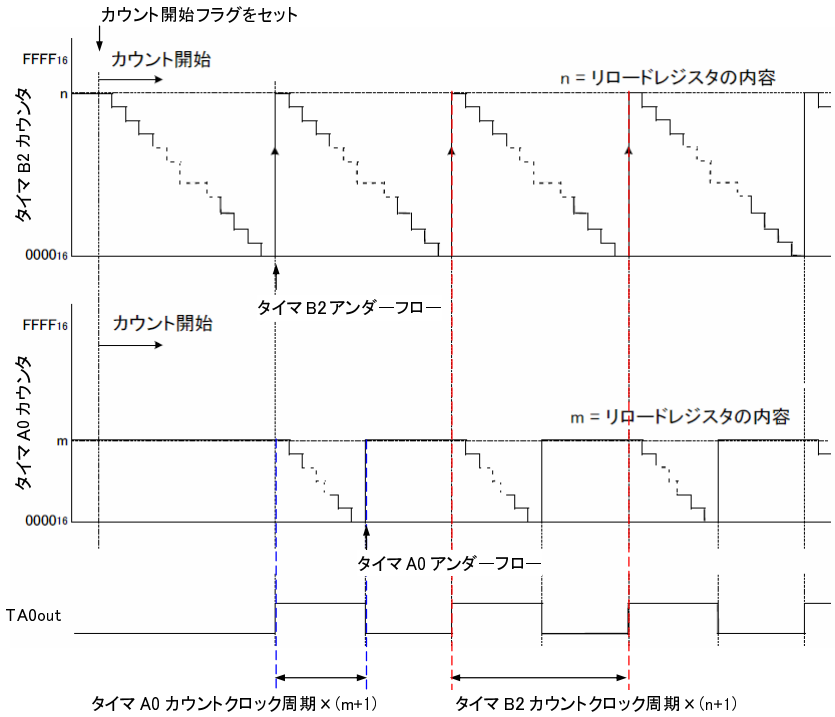


図. 6.8: タイマモードとワンショットタイマモードの組み合わせ。

6.3.5 サンプルプログラム

以上の設定で、TA0 から出力する PWM 波形のパルス幅は、ta0 レジスタに 18000（10 進数）を設定したときに 0.9[msec]、42000（10 進数）を設定したときに 2.1[msec] になります。ボリュームの回転角度を AD 変換して入力として、この 10 ビットデータをこの範囲の値に割り当てます。ta0 の値を計算する行で、どこまでの計算を浮動小数点で行っているかに注意してください。

```
//=========================================================
// 右側の VR で RC サーボ 0 の制御出力。基板外側がオレンジの配線。
// 0502.c
//=========================================================
#include <stdio.h>                    // sprintf() を使うため
#include "sfr26.h"                    // OAKSmini 用定義ファイル
#include "LCDfunc.h"                  // LCD 表示関数をインクルード
```

```
char buff[64];                          // グローバル変数にバッファを確保

void main(void) {
    unsigned int ad0_result;          // AD0 変換結果
    int i, n;
    unsigned int temp_ta0;

// RC サーボ接続ポートの初期化
    pd7 = 0xff;                         // ポート 7 方向レジスタ=出力
    p7 = ~0x00;                          // LED 消灯

// LCD の初期化
    LCD_init();                         // LCD 初期設定
    LCD_cls();                         // 全て消去

// AD 変換の初期化
                                       // 10bit 分解能/SH あり
                                       // φ AD = 10MHz
                                       // 単発モード/ソフトウェアトリガ
                                       // 変換時間 = 100nsec × 33 = 3.3usec
    adst = 0;          // AD 変換停止
    adic = 0x00;     // AD 割り込みレベル = 0(割り込みを使わない)

    adcon0 = 0x80;     // bit2,1,0:     000: アナログ入力端子 AN0 を選択
                      // bit4,3:        00: アナログ入力モード=単発モード
                      // bit5:          0: ソフトウエアトリガ選択
                      // bit6:          0: AD 変換停止
                      // bit7:          1: AD 変換動作周波数 fAD/2 を選択
    adcon1 = 0x28;     // bit1,0:        00: AD 掃引端子 AN0,AN1 を選択 (単発モード：無効)
                      // bit2:          0: AD 動作モード選択繰り返し掃引モード 1 以外
                      // bit3:          1: 分解能選択 10 ビット選択
                      // bit4:          0: AD 変換動作周波数 fAD/2 または fAD/4 を選択
                      // bit5:          1: Vref 接続：接続
                      // bit7,6:        00: ANEX0,ANEX1 は使用しない
    adcon2 = 0x01;     // bit0:          1: サンプル&ホールド有り

// タイマ A0 の初期化
    ta0ic = 0x00;     // TA0 割り込みを使用しない
    ta0mr = 0x16;     // bit1,0:          10: ワンショット
                      // bit2:          1: 出力あり
                      // bit4,3:        10:
                      // bit5:          0:
                      // bit7,6:        00: カウントソース = f1
    ta0 = 0x00ff;                       // タイマ値の初期化

// タイマ B2 の初期化
    tb2ic = 0x00;     // TB2 割り込みを使用しない
    tb2mr = 0x80;     // bit1,0:          00: タイマモード
                      // bit2:          0: どちらでも OK
                      // bit4,3:        00: このように設定
                      // bit5:          0: このように設定
                      // bit7,6:        10: カウントソース = f32
    tb2 = 9375-1;                       // タイマ値の初期化
                                       // 周期=15msec

// ワンショット開始とトリガ選択
    udf = 0x00;          // 全てダウンカウントに設定
    tabsr = 0x81;     // TB2 と TA0 カウント開始
    trgsr = 0x55;     // TA1-TA4: 全て TB2 のアンダーフローでトリガ
    onsf = 0x41;     // TA0: TB2 のアンダーフローでトリガ、TA0 ワンショット開始 (01000001)

    while ( 1 ) {
```

```
        n = 10;                          // n 回の結果を平均する
        ad0_result = 0;                  // AD 変換結果保存
        for ( i = 0; i < n; i++ ) {
            adst = 1;                  // AD 変換開始
            while ( ir_adic == 0 ) {}     // AD 変換終了を待つ (割り込み要求ビットをチェック)
            ir_adic = 0;              // 割り込み要求ビットをクリア
            ad0_result += ad0;         // AD 変換結果保存
        }
        ad0_result = ad0_result / n;
        sprintf( buff, "AD=%4d", ad0_result );

        LCD_locate( 0, 0, 0, 0 );     // カーソル OFF、点滅しない
        LCD_print_str( buff );          // 数値で表示

        LCD_locate( 0, 1, 0, 0 );     // カーソル OFF、点滅しない
        for ( i = 0; i <= ad0_result/64; i++ ) {
            LCD_print( 0xff );          // バーグラフで表示
        }
        for ( i = 1 + ad0_result/64; i < 16; i++ ) {
            LCD_print( ' ' );
        }
                                    // 0.9[msec] のパルス幅：ta0 = 18000
                                    // 2.1[msec] のパルス幅：ta0 = 42000
        temp_ta0 = (int)( ( (42000.0 −18000.0) * (float)ad0_result ) / 1024.0 ) + 18000;
        if ( temp_ta0 >= 0xffff ) {
            temp_ta0 = 0xfffe;         // TA0 設定可能範囲に補正
        }
        ta0 = temp_ta0;
    }
}
```

6.3.6 チャレンジテーマ用リモコンの考慮事項

ロボットを、初期位置に戻すための「メカ初期化」関数を用意して、電源を入れてプログラムを起動したときと、「初期化」ボタンを押したときに、この関数を呼ぶようにします。

初期位置としては、多くの場合、ラジコン用サーボの中立位置を選びます。

こうすることで、ロボットを組み立てるときも中立位置、マイクロコントローラを接続して電源を入れてもそのまま、調整が終わって電源を切るときにはまた中立位置に戻すことができます。電源スイッチを入れたとたんに、いきなりロボットが動くことがありません。

6.4 端子台

拡張基板の端子台からは、チャレンジテーマで利用するセンサなどを接続するための信号の入出力ができます。

ただし、プッシュスイッチやトグルスイッチとも接続されているので注意してください。例えば、プッシュスイッチにも同時に接続されているセンサからの信号を正しく受け取るためには、プッシュスイッチを押していない状態にしておく必要があります。トグルスイッチの場合には、手前に倒しておく必要があります。

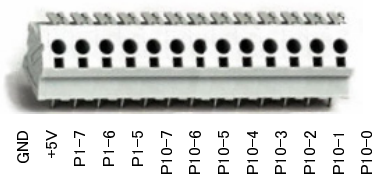


図. 6.9: 端子台で接続できる I/O ポートと電源。

しかし、このようにセンサの代わりにスイッチを接続しておくことで、センサを実際に接続しなくてもロボットの動作を確認やデバッグができます。

CPU	信号名	CN1	CN2	MC 拡張基板	MC リモコン基板
45	P10-0/AN0		38	ADC / I/O-0	VR-a
43	P10-1/AN1		37	ADC / I/O-1	VR-b
42	P10-2/AN2		36	ADC / I/O-2	VR-c
41	P10-3/AN3		35	ADC / I/O-3	
40	P10-4/AN4/KI0*		34	ADC / I/O-4	トグル SW-a to GND
39	P10-5/AN5/KI1*		33	ADC / I/O-5	トグル SW-b to GND
38	P10-6/AN6/KI2*		32	ADC / I/O-6	トグル SW-c to GND
37	P10-7/AN7/KI3*		31	ADC / I/O-7	トグル SW-d to GND
36	P1-5/INT3*/ADtrg		30	ADtrg / I/O-8	プッシュSW-e to GND
35	P1-6/INT4*		29	I/O-9	プッシュSW-f to GND
34	P1-7/INT5*		28	I/O-a	プッシュSW-g to GND

6.5 マイクロコントローラ基板を単独で動作させる方法

ここでは、マイクロコントローラ基板を単独で、すなわちコンピュータとの接続なしに動作させる方法を説明します。チャレンジテーマに出場するためには、必ず必要になります。

6.5.1 マイクロコントローラのメモリ構成の復習

マイクロコントローラには、RAM（揮発性メモリ[28]）と ROM（不揮発性メモリ[29]）が搭載されています。RAM に書き込んだ内容は電源を切れば消えてしまいますが、ROM に書いておけば電源を切っても消えません。RAM と ROM は、別なアドレスに配置されています。

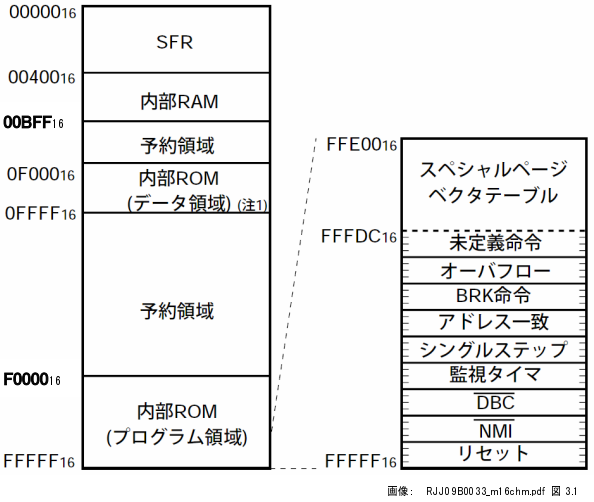


図. 6.10: アドレス空間。

6.5.2 デバッガでプログラムを実行する仕組みの復習

マイクロコントローラには、電源が入ると起動する「モニタプログラム」が書き込まれています。このモニタプログラムは、RS-232C 回線を通して送られてくるコマンドを待っています。コマンドは、例えば、

「これから RS-232C 回線を通してデータを送るから、それをアドレス xxxx からメモリに書き込め」
「アドレス xxxx からプログラムを実行せよ」
「実行しているプログラムを停止せよ」
「マイクロコントローラ内部レジスタを報告せよ」

といったものです。

[28] RAM; Random Access Memory（任意アドレスへの書き込みと読み出しができるメモリ）。用語本来の意味では、揮発性か不揮発性かの区別をしているわけではないが、通常は揮発性メモリのことを言う。

[29] ROM; Read Only Memory（読み出しだけできるメモリ）。半導体を製造する段階で一度だけ書き込めるもの（マスク ROM）、一定の手順で一度だけ書き込めるもの（ヒューズ ROM）、紫外線を当てれば初期化されて何回か書き直しができるもの（UV-EPROM; Ultra-Violet Erasable Programmable Read Only Memory）、一定の手順で何回か電気的に消去や書き込みができるもの（EEPROM; Electronically Erasable and Programmable Read Only Memory）などがある。最近では、EEPROM を搭載しているものが一般的。フラッシュメモリとも呼ぶ。何回か書き込みができても、ROM と呼ぶ。

つまり、リモートデバッガは、マイクロコントローラで操作しているモニタプログラムとの GUI(Graphical User Interface) なわけです。

リモートデバッガでの実行は、マイクロコントローラで操作しているモニタプログラムに対するコマンドで成り立っているので、デバッガが動作しているコンピュータがなければマイクロコントローラ上でのプログラムを起動することすらできません。チャレンジテーマでは、マイクロコントローラ単体での動作が求められるはずなので、このような実行方法とは別の方法が必要です。それが、フラッシュメモリに書き込んでの実行です。

フラッシュメモリに一度書き込んでしまえば、RS-232C ケーブルもデバッガもコンピュータも必要ありません。では、プログラムをデバッグするたびにフラッシュメモリに書き込んではどうでしょう。ところが、これには問題があります。フラッシュメモリは、書き込み回数に限界があるのです。デバッグでは、ちょっと変えて試してみて、またちょっと変えて試してみて、の繰り返しですが、そのたびにマイクロコントローラ内部回路の寿命を縮めていたのでは、安心して信頼性の高いプログラムを開発することもできません。

6.5.3 フラッシュメモリに書き込む方法

(1) プロジェクトの作成とデバッグ

今までと同じ手順でプロジェクト用のフォルダを作り、必要なファイルをそのフォルダにコピーします。TM の「プロジェクトの新規作成」でプロジェクトファイルを用意します。コンパイルできることを確認します。また、デバッガを使って十分に動作を確認します。

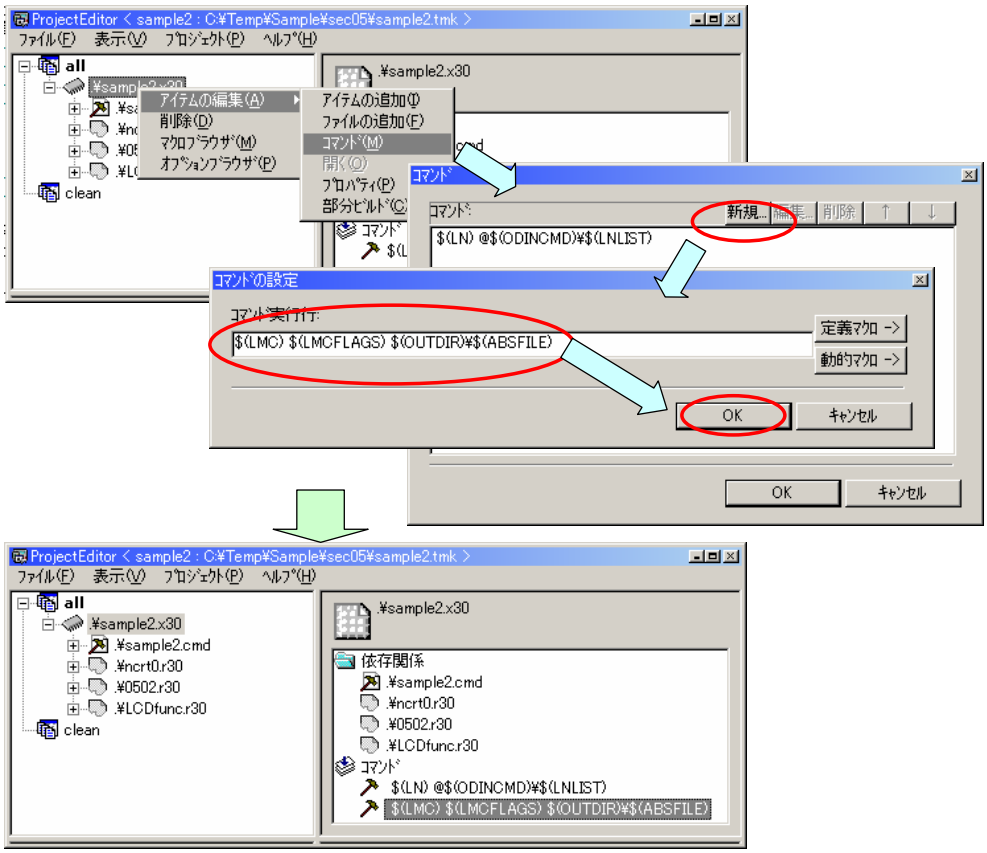


図. 6.11: プロジェクトファイルへのコマンドの追加。

(2) MOT ファイル作成の設定を追加

Flashstarter（フラッシュROM に書き込むためのソフト）で使用するモトローラ s フォーマットファイルを作成するためのコマンドを追加します。

[アイテムの編集]-[コマンド] を選択すると、図 6.11 のウィンドウが表示されます。これには、ビルドの最後に実行されるコマンドが記述されています。

[新規] ボタンをクリックして、

```
$(LMC) $(LMCFLAGS) $(OUTDIR)￥$(ABSFILE)
```

を追加します。

(3) ビルド

ビルドすると、拡張子が mot のファイルができています。これを「Flash Starter」で、マイクロコントローラの ROM に書き込みます。

(4) 準備

デバッガ KD30 が起動していたら終了します。拡張基板の電源を切ります。シリアルケーブルを接続します。マイクロコントローラ基板の近くにあるスライドスイッチを緑色の LED 側に切り替えて、電源を入れます（図 6.12）。

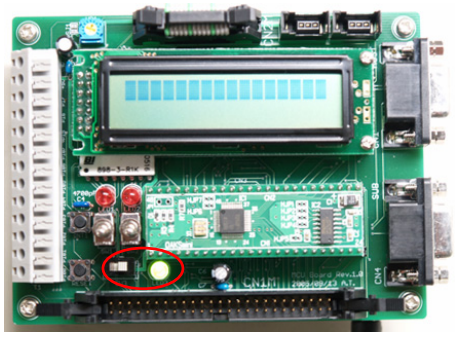


図. 6.12: フラッシュメモリに書き込むときのスライドスイッチ。この図は、「書き込み」側。

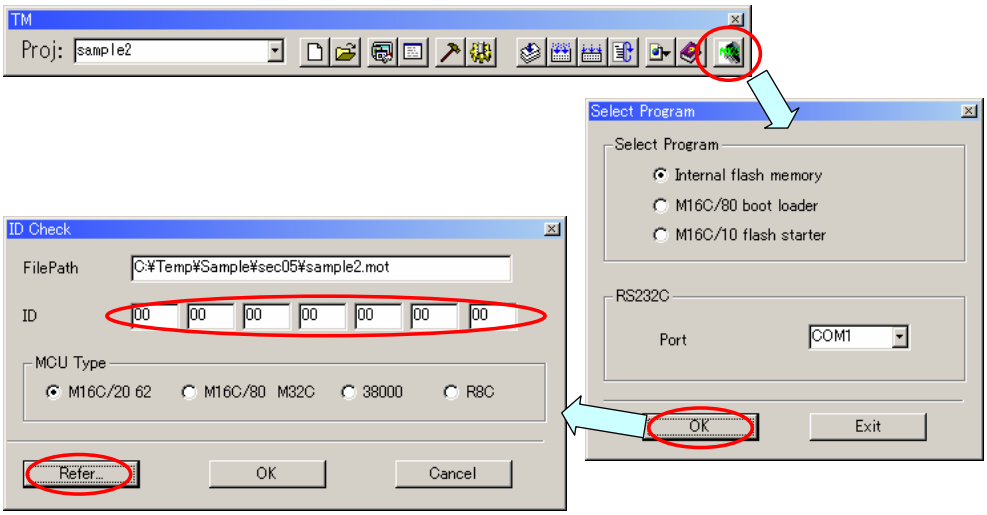


図. 6.13: 書き込むファイルの選択。

(5) Flash Starter での書き込み – 書き込むファイルの選択

TM の「Flash Starter」（登録してあれば）をクリックします。「Select Program」ダイアログではそのまま [OK] をクリックします。次の「ID Check」ダイアログで、[Refer...] ボタンをクリックして書き込むファイル（作成したばかりの mot ファイル）を選びます。「ID ファイルが存在しない」という警告が出ますが、[OK] をクリックします。「ID Check」ダイアログで、ID には全て「00」を入力します。

(6) Flash Starter での書き込み – ROM の消去

「ID Check」ダイアログで [OK] をクリックすると、いよいよ「Flash Starter」ダイアログから ROM の内容を変更します。まず、ROM を消去します。

[Erase] ボタンをクリックして、確認ダイアログで [OK] をクリックします。しばらくすると、「正常に消去された」という内容を表示します。[OK] をクリックします（図 6.14）。

(7) Flash Starter での書き込み – プログラムの書き込み

「Flash Starter」ダイアログで、[Program...] をクリックします。「Input Address」で [OK] をクリック、「Program」でも [OK] をクリックします。書き込み中は、何事もなく終了するように、できるだけさわったりいじったりしないようにします。書き込みには、時間がかかります。「Program OK」が表示されたら [OK] をクリックします。

「Flash Starter」ダイアログで、[Exit] をクリックします（図 6.15）。

(8) 書き込んだプログラムの実行

拡張基板の電源を切ります。シリアルケーブルを外します。マイクロコントローラ基板の近くにあるスライドスイッチを、緑色の LED の反対側に切り替えて、電源を入れます。

6.5.4 デバッガで実行するためのモニタプログラムに戻す方法

ROM への書き込みには時間がかかり、手順も複雑です。このため、プログラムをデバッグするときには、コンピュータとシリアルケーブルで接続して、デバッガでの実行が便利です。しかし先ほどの書き込みで、デバッガとの通信をするモニタプログラムを消してしまいました。そこで、もう一度モニタプログラムを書き込みます。モニタプログラムは、各自コピーした H:￥Sample￥Monitor フォルダにあります。

(1) 準備

デバッガ KD30 が起動していたら終了します。拡張基板の電源を切ります。シリアルケーブルを接続します。マイクロコントローラ基板の近くにあるスライドスイッチを緑色の LED 側に切り替えて、電源を入れます。

(2) Flash Starter での書き込み – 書き込むファイルの選択

TM の「Flash Starter」（登録してあれば）をクリックします。「Select Program」ダイアログではそのまま [OK] をクリックします。次の「ID Check」ダイアログで、[Refer...] ボタンをクリックして書き込むファイル（26mo_p.mot）を選びます。ID ファイルが存在するので、自動的に ID には全て「00」が入ります。

(3) Flash Starter での書き込み – ROM の消去

「ID Check」ダイアログで [OK] をクリックすると、いよいよ「Flash Starter」ダイアログから ROM の内容を変更します。まず、ROM を消去します。

[Erase] ボタンをクリックして、確認ダイアログで [OK] をクリックします。しばらくすると、「正常に消去された」という内容を表示します。[OK] をクリックします。

(4) Flash Starter での書き込み – プログラムの書き込み

「Flash Starter」ダイアログで、[Program...] をクリックします。「Input Address」で [OK] をクリック、「Program」でも [OK] をクリックします。書き込み中は、何事もなく終了するように、できるだけさわったりいじったりしないようにします。「Program OK」が表示されたら [OK] をクリックします。

「Flash Starter」ダイアログで、[Exit] をクリックします。

(5) デバッガでの実行の準備

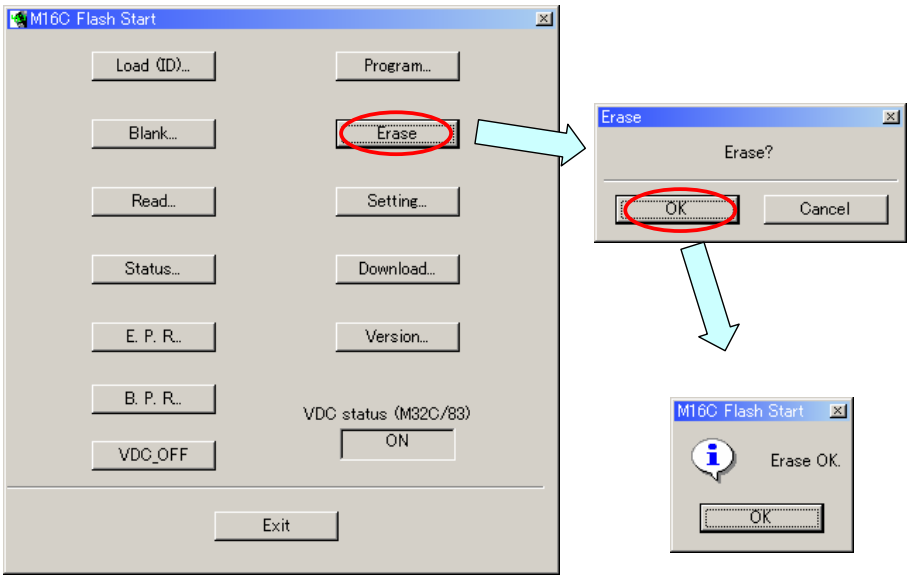


図. 6.14: ROM の消去。

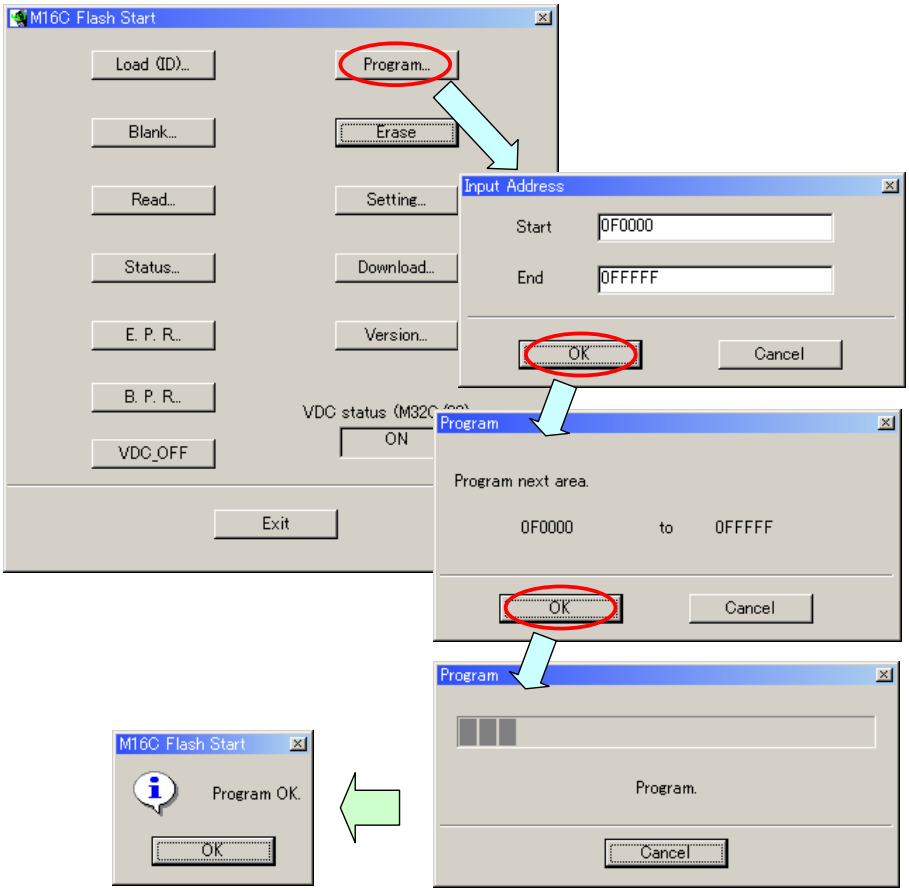


図. 6.15: プログラムの書き込み。

拡張基板の電源を切ります。マイクロコントローラ基板の近くにあるスライドスイッチを、緑色の LED の反対側に切り替えて、電源を入れます。

6.6 リモートデバッガ KD30 の利用方法

実習で使う開発環境では、たいへん高機能で使いやすいデバッガを利用しています。全ての機能を説明することはできませんが、プログラムをデバッグする上で便利な機能をいくつか紹介します。紹介した内容の詳細や他の機能については、ヘルプを参照してください。以下は、ヘルプからの抜粋です。

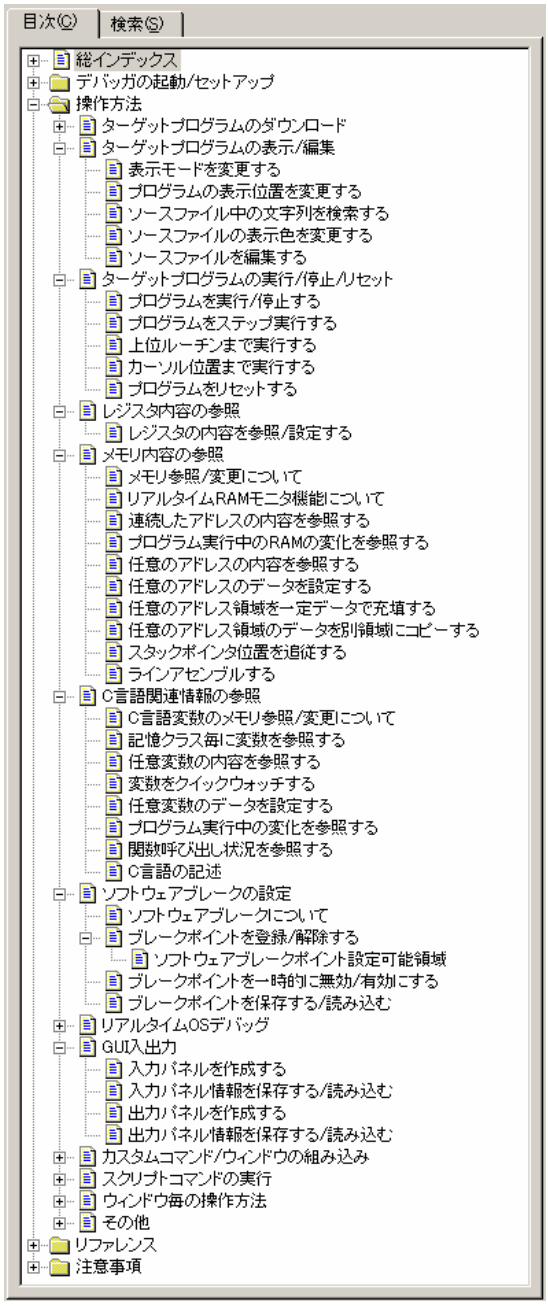


図. 6.16: デバッガ KD30 のヘルプ目次。

6.6.1 変数の内容を参照する

任意の C 言語変数の内容は、C ウォッチウィンドウで参照します。

C ウォッチウィンドウをオープンするためには、メニュー [Basic Windows] → [C Watch Window] → [C Watch Window] を選択して下さい。登録した C 言語変数を、C ウォッチポイントと呼びます。登録には、「通常登録」と「ポインタとして登録」の 2 種類があります。登録する C 言語変数に合わせて指定して下さい。

C 言語変数の値は、プログラム（ソース）ウィンドウで参照することもできます（クイックウォッチ）。

プログラム (ソース) ウィンドウで、マウスカーソルを一定時間（約 0.5sec）参照する C 言語変数上に静止させて下さい。C 言語変数の内容をポップアップ表示します。

6.6.2 ブレークポイントを設定する

ソフトウェアブレークポイントは、プログラム（ソース）ウィンドウで設定/解除できます。

まず、[View] ボタンをクリックして、さらに [Souece...] ボタンをクリックして、デバッグしたいソースファイルを表示します。

プログラム（ソース）ウィンドウで、ブレークポイント設定する行のブレークポイント表示領域をダブルクリックして下さい (”-” の表示が”B” に変わります)。もう一度ダブルクリックすると、ブレークポイントの設定解除となります (”B” の表示が”-” に変わります)。プログラム (ソース) ウィンドウのブレークポイント表示領域が空欄の場合 (データ定義行、コメント行、空行 等) は、ソフトウェアブレークポイントの設定ができません。

6.6.3 カーソル位置まで実行する

カム実行（カーソル位置までの実行）は、現プログラムカウンタ位置からクリックしたカーソル位置までターゲットプログラムを実行する機能です。

プログラム（ソース）ウィンドウで、停止させたい行をクリックしてください。データ定義行、コメント行、空行は、指定できません。

メニュー [Debug] → [Come] で、現プログラムカウンタ位置からクリックした行まで実行します。

6.6.4 ソースファイルを編集する

プログラム（ソース）ウィンドウ上で、ターゲットプログラムのソースファイルを編集することができます (エディット機能)。

モードの切り替え： デフォルトの状態では、デバッグモード（編集不可）です。編集モードに切り替えるには、プログラム（ソース）ウィンドウのプログラム表示領域上を右クリックし、ポップアップメニューから [Edit] → [On] を選択してください。再度、ポップアップメニューから [Edit] → [On] を選択することにより、デバッグモード（編集不可）に切り替わります。

ソースファイルの編集： 編集するためのショートカットキーを割り付けています（変更することも可能です)。

Ctrl+C： 文字列のコピー
Ctrl+X： 文字列の消去
Ctrl+V： 文字列のペースト
Ctrl+Z： 編集のやり直し（1 回のみ）

ソースファイルの内容を変更すると、タイトルバーに「*」が表示されます。この「*」の表示は、ソースファイルの更新内容を保存した時に消えます。

ソースファイルの保存： 編集したソースファイルを保存するには、プログラム（ソース）ウィンドウ上で右クリックし、ポップアップメニュー [Edit] → [Save] を選択してください。別名でファイルを保存する場合は、ポップアップメニュー [Edit] → [Save As...] を選択してください。ポップアップメニュー [Edit] → [Save All] を選択することにより、編集モードの全プログラム（ソース）ウィンドウの編集内容を保存することができます（現在のファイルに上書きします)。

ソースファイルの内容が変更されている状態（タイトルバーに「*」が表示されている状態）で、プログラム（ソース）ウィンドウをデバッグモードに切り替えると（ウィンドウを閉じた場合も同様)、警告ダイアログを表示します。警告ダイアログでは、変更内容をファイルに保存しデバッグモードに切り替える場合は「はい」、変更内容を破棄しデバッグモードに切り替える場合は「いいえ」、編集モードの状態を維持する場合は「キャンセル」を選択してください。

編集モードのプログラム（ソース）ウィンドウが存在する場合、ターゲットプログラムの実行/リセット/ダウンロード操作はできません。その場合は、デバッグモードに切り替えてください。