

TOPPERS基礎実装セミナー

(M16C版:基本)
基礎編:2日目

TOPPERSプロジェクト
教育ワーキング・グループ

本教材の利用条件

NEXCESS基礎コース01 組込みソフトウェア開発技術の基礎

Copyright (C) 2006-2007 by 名古屋大学 組込みソフトウェア技術者人材養成プログラム
Copyright (C) 2006-2007 by 本田晋也

上記著作権者は、以下の(1)～(4)の条件を満たす場合に限り、本コンテンツ(本コンテンツを改変・翻訳したものを含む、以下同じ)を使用・複製・改変・翻訳・再配布(以下、利用と呼ぶ)することを無償で許諾する。

- (1) この枠内の著作権表記等が、そのままの形でコンテンツ中に含まれていること。
- (2) 本コンテンツを再配布する場合には、再配布の形態等を、以下のウェブサイトから報告すること。
<http://www.nces.is.nagoya-u.ac.jp/NEXCESS/REPORT/>
- (3) 本コンテンツを改変・翻訳する場合には、コンテンツを改変・翻訳した旨の記述を、コンテンツ中に含めること。また、改変・翻訳者の著作権表記等は、この枠内の著作権表記等とは別に行うこと。
- (4) 本コンテンツの利用により直接的または間接的に生じるいかなる損害からも、上記著作権者を免責すること。

※ 本コンテンツの一部は、文部科学省 科学技術振興調整費により、名古屋大学 組込みソフトウェア技術者人材養成プログラム(NEXCESS)の一環として作成しました。

※ 本コンテンツ中に記載されている商品名やサービス名などは、各社の商標または登録商標です。

本ドキュメントに関して

1. 著作権に関しての表記

＜TOPPERS基礎実装セミナー(M16C版:基本)2日目＞

Copyright (C) 2006-2007 by 名古屋大学 組込みソフトウェア技術者人材養成プログラム

Copyright (C) 2006-2007 by 本田晋也 名古屋大学

Copyright (C) 2007 by 竹内良輔 (株)リコー

上記著作権者は、以下の (1)～(3) の条件を満たす場合に限り、本ドキュメント (本ドキュメントを改変したものを含む。以下同じ) を使用・複製・改変・再配布 (以下、利用と呼ぶ) することを無償で許諾する。

(1) 本ドキュメントを利用する場合には、上記の著作権表示、この利用条件および以下の無保証規定が、そのままの形でドキュメント中に含まれていること。

(2) 本ドキュメントを改変する場合には、ドキュメントを改変した旨の記述を、改変後のドキュメント中に含めること。ただし、改変後のドキュメントがTOPPERSプロジェクト指定の開発成果物である場合には、この限りではない。

(3) 本ドキュメントの利用により直接的または間接的に生じるいかなる損害からも、上記著作権者およびTOPPERSプロジェクトを免責すること。また、本ドキュメントのユーザまたはエンドユーザからのいかなる理由に基づく請求からも、上記著作権者およびTOPPERSプロジェクトを免責すること。

本ドキュメントは、無保証で提供されているものである。上記著作権者およびTOPPERSプロジェクトは、本ドキュメントに関して、特定の使用目的に対する適合性も含めて、いかなる保障もしない。また、本ドキュメントの利用により直接的または間接的に生じたいかなる損害に関しても、その責任を負わない。

2. 本ドキュメントに関するご意見・ご提言・ご感想・ご質問等がありましたら、TOPPERSプロジェクト事務局までE-Mailにてご連絡ください。

3. 本ドキュメントの内容は、内容の改善や適正化の目的で予告無く改定することがあります。

本ドキュメントでは、Microsoft社のClip Art Galleryコンテンツを使用しています。

TRONは"The Real-time Operating system Nucleus"の略称です。ITRONは"Industrial TRON"の略称です。
μITRONは"Micro Industrial TRON"の略称です。TOPPERS/JSPはToyohashi Open Platform for Embedded Real-Time System/Just Standard Profile Kernelの略称です。」

本ドキュメント中の商品名及び商標名は、各社の商標または登録商標です。

スケジュール

■ 2日目

- | | |
|------------------------|-------|
| 1. 組込み開発のノウハウ | 0.5時間 |
| 2. メモリマップドレジスタの操作方法の確認 | 1.0時間 |
| 3. ポーリングプログラム | 2.0時間 |
| 4. 割込みプログラム | 2.0時間 |
| 5. まとめ | 0.5時間 |
| 6. おまけ: カップラーメンタイマーの作成 | |

概要

- マイコンボードを用いて
組込みプログラミングの基礎を学ぶ
- メモリマップドレジスタの操作方法の確認
 - デバッガを用いてデバイスレジスタを操作する
- ポーリングプログラミング
 - LED, スイッチ, タイマ, シリアルI/Oを操作する
プログラムの作成
- 割り込みプログラミング
 - スイッチ, タイマ, シリアルI/Oからの事象を割り込み
により扱うプログラムの作成
- カップラーメンタイマの作成
 - スイッチ, タイマによりカップラーメンタイマを実現

組込み開発のノウハウ

いろいろな開発環境

- CPU、半導体メーカーの開発ツールを使用する
実行環境に即した開発ができる
CPUが変わるごとに環境に熟練する必要がある
- 汎用の開発環境で開発する
スタートアッププログラムから自作する
ROMモニタを使用して開発する
- ICE等の専用ツールを使用する

2008/11/28

TOPPERSプロジェクト認定

7



組込み開発は、いろいろな開発環境があります。主な3つを紹介します。

1) CPU、半導体メーカーの開発ツールを使用する。

RTOSを使わない小規模な組込みシステムでは最適な開発環境です。ARMやMIPSやPowerPCのような高速なCPU、海外のCPUは開発環境は専用会社から供給されるケースが多いので2)のケースとなることが多い。

2) 汎用の開発環境で開発する

組込みサードベンダで供給する開発環境やGNUのように自分で開発環境を構築するケースがこれに当たる。RTOSを使うケースの中規模組込み開発はこの開発環境が多い。サードベンダに開発委託する場合、サードベンダも最後まで面倒を見る開発形態や、基本的な開発を行ってくれ、後は商品会社に引き継ぐ開発形態もある。長年にわたって機能アップやコストダウンする商品の場合は、後者の方が全体の開発経費が格段に安くなる。

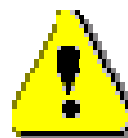
オープンソフト等を使用して自分で開発環境から構築する場合は、最初にROMモニタと呼ばれるプログラムのローディング実行を行うモニタをROMに書き込み少しずつ開発を行っていく。開発規模がある程度大きくなったらアップローダー(USBやネットワークを使ってシステムを更新するツール)を導入した方が開発効率が上がる。

3) ICE等の専用ツールを使用する

CPU、半導体メーカー、エレクトロニクス系のサードベンダはICE等の開発ツールを供給します。中規模以上の開発ではプログラム規模が大きくなり、不具合が発生した場合、どのモジュールが原因で発生したのかの特定が難しいケースが多々あります。このようなケースではICE等に機器を使用することで問題解決に役立ちます。ICEを使用する場合、商品用のボードにはない特殊なIFをもつボードが必要であったり、すべての開発者のICEを配ると開発費が高くなったりします。目的をもってICEを使用するようにしましょう。

デバッグ時の注意

- 組込みプログラムの注意として、プログラムミスを行うとプログラムの暴走に直結する
- WindowsやLinuxでは、アプリケーションが暴走状態になっても、そのアプリケーション以外には影響を与えないので、問題開発が容易い
- 小規模組込みシステムでは、暴走状態に陥ると履歴も取れず、ICEがないと現状状態の確認もできない
⇒注意深い、ソフトウェア開発が必要



2008/11/28

TOPPERSプロジェクト認定

8



デバッグ時の注意について説明を行います。組込みプログラムはパソコン上のプログラム開発と異なり、開発者が直接プロセッサ設定やハードウェアのプログラムを作成したり、アプリケーションレベルのプログラムでもメモリ保護機能がない場合が多いので、簡単なプログラムミスでプログラムが暴走したり、エクセプションが発生したりします。

小規模組込みシステムでは、管理機能も貧弱であるため暴走状態に陥ると何が原因かの特定が難しい場合が多々あります。このような場合、ICEを使用することで、このような場合でも、実行状態の確認ができるようになります。中規模組込みシステムでは、プログラムサイズが大きくなり、品質の低いソフトウェアが多いと、システムが不安定になったり、なかなか商品テストが終わらなかつたり等、納期遅れや市場対応等の大きな問題に発展しかねません。

どのような、ソフトウェア開発が必要なのでしょう？

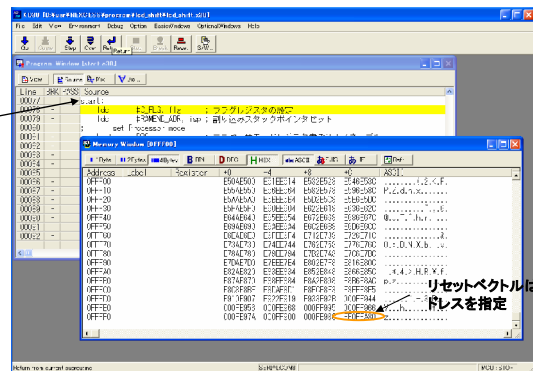
システムの基本部分のプログラムは、品質や機能拡張の対応のために何度も作り変えが発生します。注意深い開発や開発スキルの高い開発者あつめるような施策が必要となるでしょう。また、アプリケーション部でも、シミュレータ環境を構築し不具合を実機環境に持ち込まないような開発プロセスが必要です。モデル設計のような方法論をもちいるのも有効な開発手段です。

CPU、半導体メーカーのツールを使用する

- 一般的にリモートデバッガが多い
- M16Cの場合、モニタプログラムがリモートデバッガとユーザプログラムの間に介在し、リモートデバッグ環境を構築する
- Visual-Studioのようなデバッグ環境を提供する

ユーザプログラムのリセットアドレスは0xE00000

Goボタンを押すと、コマンドをモニタが受け取り、ユーザプログラムのリセットアドレスから実行させる



2008/11/28

TOPPERSプロジェクト認定

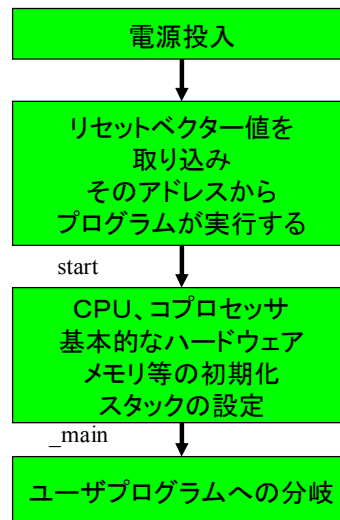
9



CPU、半導体メーカーのツールはプロセッサに特化した開発環境を提供します。パソコン上のVisual-Studioのような開発環境を提供してくれます。本セミナーではM16Cの開発環境を使って実習をおこないます。

スタートアップルーチンとは

- GNUなどの開発環境を使用する場合
- CPUはリセットベクトルから起動し、ハードウェアの初期化後、はじめてmain()が実行
 - 電源オンまたはリセットからmain()が実行するまでのプログラムをスタートアップルーチンと呼ぶ
- TMでは、リモートデバッガがスタートアップ処理を代行してくれている



2008/11/28

TOPPERSプロジェクト認定

10



スタートアップルーチンは、どのような組込みシステムでも必ず電源投入時に実行されるプログラムです。CPUや半導体メーカーの開発環境やサードベンダから供給される開発環境では、スタートアップルーチンがうまく隠蔽されていて、どのような機能をはたすのかわからないようになっています。

GNUなどのコンパイラを使用して組込みシステムを構築すると、スタートアップルーチンを自作しないと実際に作成したプログラムを実行されることができません。スタートアップルーチンがどのような機能をおこなうのか、簡単に説明します。スタートアップルーチンは電源投入時やリセット信号を受け付けたときに、実行するリセットベクタにより実行を開始します。スタートアップルーチンはプロセッサのレジスタを設定したり、C言語が実行可能なメモリ空間を構築するため、アセンブラ言語で記述されます。

- 1) プロセッサやハードウェア (特にメモリやポート) の初期設定を行う。スタックや割込みベクタの設定を行う。
- 2) C言語 (C++ 言語) のメモリ空間の設定を行う。メモリアロケーションやアボート時の設定を行う場合もある。
- 3) ユーザプログラム (メイン関数) に分岐を行う。

セクションデータの再配置

- C言語を正しく実行するためには、リンク後の各セクションデータが実行ROM、RAMに正しく配置しなければならない

セクション	Gcc系セクション	M16Cセクション
コンスタントデータ	.rodata	rom_FE
書換え可能データ	.data	data_NE～data_NO
ワーク領域	.bss	bss_NE～bss_NO
スタック領域		

- 書換え可能データは、最初はROM上にあるため対応のRAMにコピーする必要がある
- ワーク領域はゼロ初期化する必要がある
- スタック用に未使用のRAMを割り当てを行う

2008/11/28

TOPPERSプロジェクト認定

11



C言語やC++言語を正しく動作させるためにメモリ空間を設定する必要があります。セクションはコンパイラによって異なります。上記の例では、GCCの場合とM16C用のNC30コンパイラの例を挙げています。C言語のセクションは基本的に4つの領域の分けられています。

- 1) 実行の機械語のセクション
- 2) 読み取り専用のデータ領域(コンスタントデータ)
- 3) 初期値があり、プログラムの実行により書換えが行われるデータ領域
- 4) ワーク領域: 言語として初期値はゼロクリアされていないといけない

機械語の領域やコンスタンスデータの領域は一般的にROM領域に割り当てられるので、スタートアップルーチンでは、特別な処理の必要はありません。書換え可能データは初期値がROM上にありますが、領域自体はRAM上にありますので、C言語の実行前にコピーを行います。同様にワーク領域はRAM上の領域をゼロクリアします。

スタック領域はセクションで割り当てられていないRAM領域をスタックポインタにセットします。

C++言語の場合、セクション以外にスタート時静的インストラクタ、終了時静的デストラクタの実行を行わなければなりません。また、**new**や**delete**命令の実行用に**alloc**や**free**関数がきちんと動作するように初期設定をおこなわなければならない。

スタートアップルーチンの例

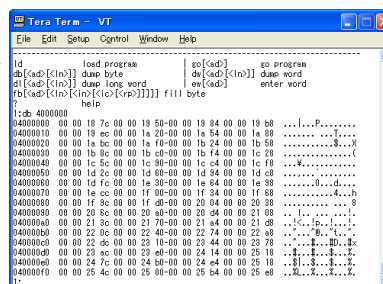
- start.a30がM16C用のスタートアップルーチン
- 割込みを使用する場合、割込み関数をベクタテーブルにセットする必要がある
- start.a30開いて、内容を確認してください
- 初期化終了した時点でjsr.a _mainでmain()の実行を始める

スタートアップルーチンの例が今回の実習プログラム中にあります。このプログラムはstart.a30というアセンブラ言語のプログラムです。このプログラムでは割込みベクタの設定も行っています。割込みプログラムの章で詳細の説明を行います。

ROMモニタから起動する

- ROMモニタとはエバレーション・ボードのROMに直接書き込まれ、プログラムのダウンロードや実行、データ参照を行うプログラム

- RAM領域が大きなシステムに多く、小規模な組み込みシステムには少ない
 - クロス環境で作成したプログラムをダウンロード実行してデバッグを行い
- 専用メーカーから購入したり、オープンソースを利用する場合もある



TOPPERSのWebからダウンロード可能なAKIH8_3069F用のROMモニタ

ROMモニタはエバレーションボードのROM領域に直接書き込まれ、プログラムのダウンロードや実行やデータの参照や設定を行うプログラムです。RAM領域が大きなシステムに多く、ワンチップCPUを用いる小規模な組み込みシステムではあまり使用しません。クロス環境で開発したプログラムをダウンロード実行をデバッグを行います。

ROMモニタは専用メーカーから製品として発売されているものや、オープンソースのものがあります。TOPPERSプロジェクトではH8のボード用のROMモニタのソースをオープン化しています。参考にして下さい。

ICEを使ってデバッグを行う

- デバッグ環境としては、リモートデバッガと同様な開発環境を提供する
- クロス環境にリモートデバッガがない環境でも、ICEと併用して、リモートデバッグ可能なものもある
- フルICEの場合、トレース実行などの実行状態の監視などの機能も追加される
 - プログラム暴走原因の解析等に有用である
- 高速なRISC系のCPUでは、サポートされないものも多い

組み込みの開発環境の中にはシミュレータは充実していますが、実機上のデバッガは機能が少ないものがあります。このような開発環境でもICEと組み合わせてリモートデバッグできるものもあります。

ICEにはいろいろな種類があり、フルICEと呼ばれるICEは、CPU実行状態をトレースできますので、プログラムが誤動作する前後のトレースやメモリアクセスを参照することができ、繊細なデバッグに役立ちます。一般的なICEはJ-TAG ICEと呼ばれ、プロセッサ内に仕掛けられたデバッグ機能をJ-TAGというインターフェイスを通して制御するICEです。しかし、実際にプログラム規模の大きい高速なCPUでは、実行クロックが高速すぎて、ノイズ等の影響を受けやすく、サポートされていないものも多いのも事実です。

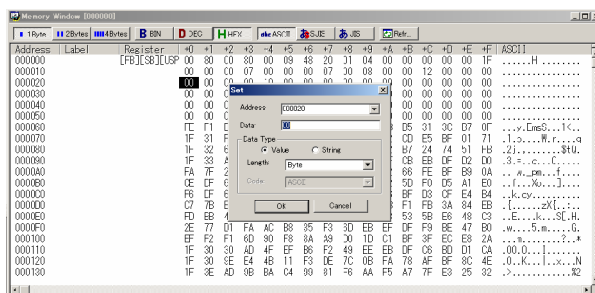
メモリマップドレジスタの操作方法 の確認

1. [デバッガの操作](#)
2. [LEDの接続](#)
3. [LEDの操作](#)

メモリマップドレジスタの操作方法の確認：目的

LEDが接続されているプログラマブル入出力ポートを例に
メモリマップドレジスタの操作方法を学ぶ

- デバッガ(KD30)を用いると任意のアドレスのメモリの読み書きが可能
- M16Cの周辺機能の制御レジスタはメモリ空間に配置されている(メモリマップドレジスタ)
 - デバッガからLEDが接続されているポートを操作可能
- KD30の Memory Window によるメモリの読み書き



2008/11/28

TOPPERSプロジェクト認定

16



メモリマップドレジスタとは、プロセッサのメモリアドレスに配置されたレジスタを意味します。M16Cのインターフェイスレジスタはメモリマップドレジスタですので、デバッガ(KD30)を用いて、参照や設定が行えます。デバッガの機能を用いて、LEDの操作を行います。TMからデバッガを起動するとプログラムのローディングが行われますので、直接、KD30を起動します。

デバッガの操作方法：デバッガ(KD30)の起動

- 開発環境の確認編の“ボードのセットアップ”に従ってマイコンボードとPCを接続する
- デバッガを起動する前は常に
 - BOOTスイッチがWRITE側ではないか?
 - ボードの電源が入っているか?
 を確認して、一度リセットボタンを押す



- デバッガの起動
 - スタート→プログラム→RENESAS-TOOLS→KD....から KD30を実行

2008/11/28

TOPPERSプロジェクト認定

17

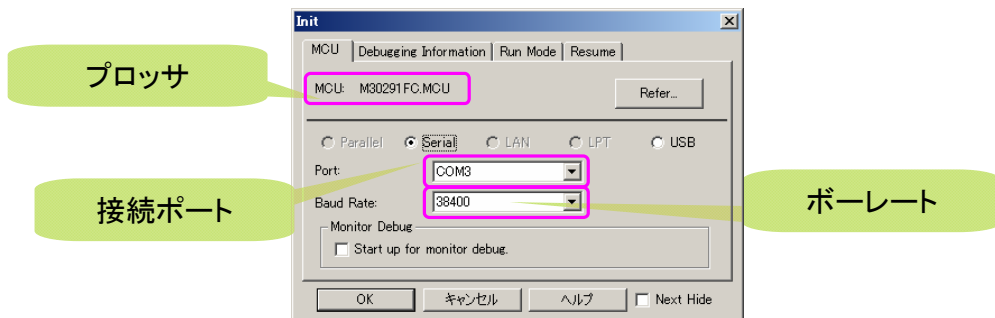


デバッガの起動の手順

- 1) 1日目の「ボードセットアップ」に従って、ボードとPCをUSBケーブルで接続します。
- 2) ボードのBOOTスイッチがWRITE側になっていないことを確認します。
- 3) ボードのPOWERランプが点灯していることを確認します。
- 4) リセットボタンを押します。
- 5) PCのスタートメニューからすべてのプログラム→RENESAS-TOOLS→KD30 4.10 Release 1→KD30を起動します。

デバッガの操作方法：デバッガ初期設定

- デバッガを起動すると表示されるデバッガの初期化画面で設定を確認
 - 設定は毎回確認すること
- デバッガが起動しない場合や、エラーとなる場合は、開発環境の確認編の“起動しない場合”を参照のこと



2008/11/28

TOPPERSプロジェクト認定

18



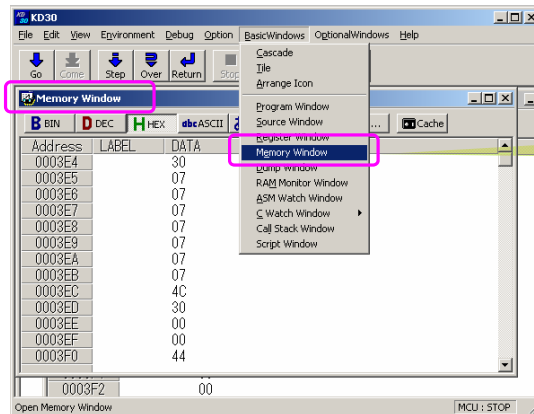
KD30の起動により、Initのダイアログボックスが表示されます。ダイアログボックスの以下の項目を確認してください。

- 1) MCU がM30291FU.MCU となっているか？
- 2) Port がUSBシリアルの設定と同じですか？
- 3) Baud Rate が38400(bps)になっていますか？

確認が終わったら、「OK」を押して、デバッガを起動してください。

デバッガの操作方法 : Memory Window

- Basic Windows → Memory Windowを選択してMemory Windowを開く

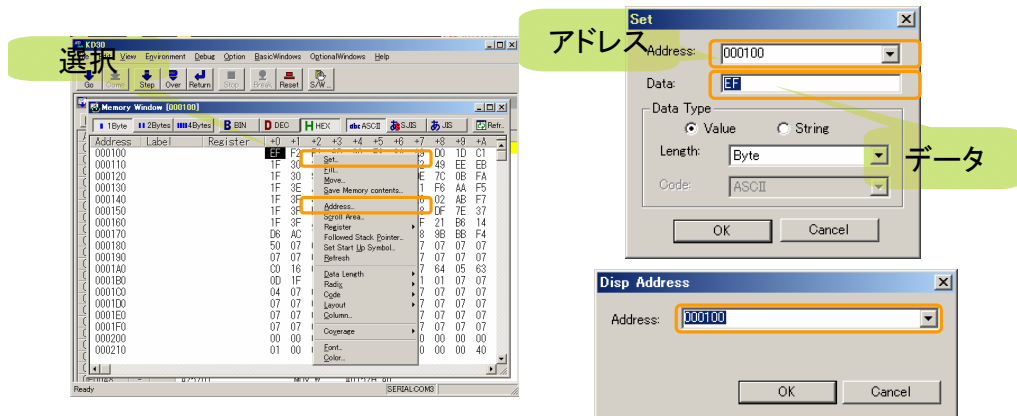


まず、Memory Windowを使ってメモリの読み書きを行います。

Basic Windows → Memory Windowsを選択して、Memory Windowを開きます。最初にMemory Windowsを開く場合は、表示アドレスを問い合わせるダイアログボックスが開かれますが、そのまま「OK」を押してください。

デバッガの操作方法：読み書き

- ・メモリウィンドウで右クリックすると表示されるメニューからSet..を選択
- ・選択すると表示されるウィンドウでAddress及びData(16進数)を設定してOKを押すと指定したアドレスに書き込まれる
- ・右クリックすると表示されるメニューからAddress..を選択すると表示されるウィンドウで表示したい(読みたい)アドレスを指定する



2008/11/28

TOPPERSプロジェクト認定

20



・メモリにデータをセットするには、Memory Windowsを右クリックし、Set...を選択すると、セット用のダイアログボックスが表示されます。AddressとDataに値をセットして「OK」を押すと、データがセットされます。

・メモリの表示アドレスを変更するには、Memory Windowsを右クリックし、Address...を選択すると、Dump Addressが表示され、Addressに値をセットして「OK」を押すと、表示アドレスが変わります。

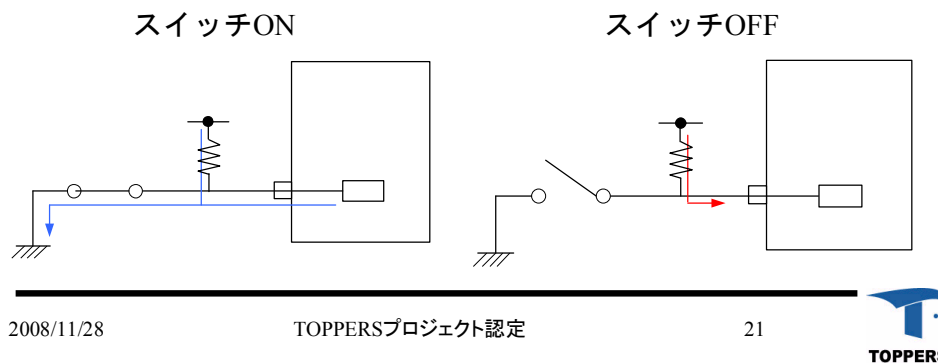
周辺デバイス：プルアップとプルダウン

- デジタル回路は電圧の高低(HiとLo)で'0'と'1'を判断する
例) 電源電圧が5vの場合, Hiが4~5v, Loは0~1v
- LSIへの入力はHiかLoの電圧になるようにしなければならない

プルアップとプルダウン

- GPIO入力ポートとしてスイッチ等を接続する場合, 入力を電氣的に安定させるための回路構成

プルアップの例



2008/11/28

TOPPERSプロジェクト認定

21



スイッチのハードウェアを確認します。二段になった下のボードBB STUDY I/Oの回路図を開いてください。Titleに「STUDY IO」と書かれた回路図です。

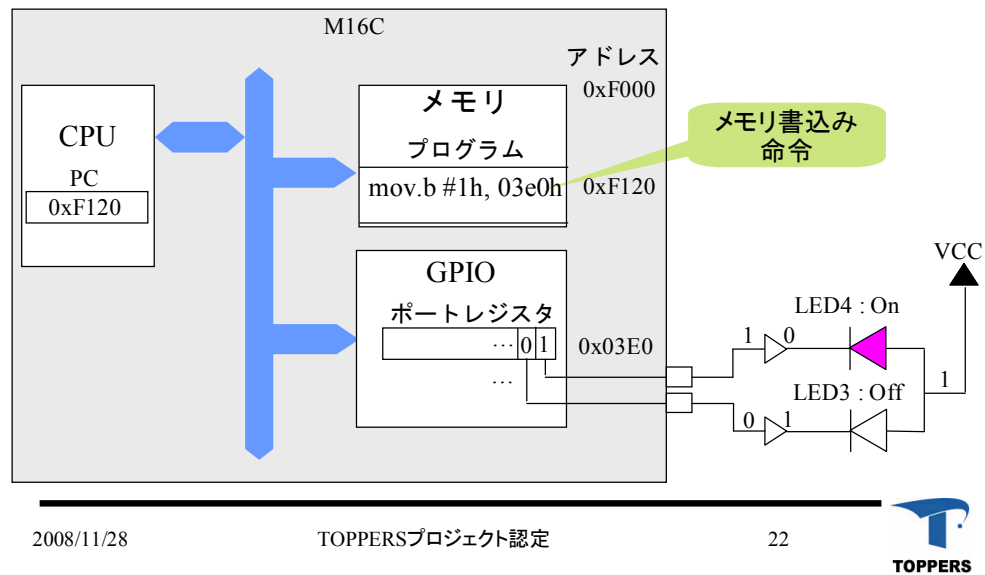
スイッチのONとOFFは電圧0Vか5Vで判断を行います。SW9とSW8は回路図の中央少し下に記載されています。SW9は→INT__SW1→INT3→J2コネクタ3ピンに結線しています。J2コネクタにより、上のボードに接続していることがわかります。上のボードはCPUボードで、今度は、titleに「HSB16C29S64 NE」と書かれた回路図を確認してください。右上のJ2コネクタの3ピンはCPUのP15に結線されていることがわかります。P15はポートP1レジスタの5ビット目を示します。同様に、SW9は→INT__SW2→INT4→J2コネクタの4ピン→P16に接続しています。これはCPUのポートP1レジスタの6ビット目をしめします。

上記の図のように、スイッチのオンオフはポートレジスタの15番と16番に接続しています。

周辺デバイス : プロセッサとLEDとの接続の関係

HSB16Cの例

–ポートレジスタの値が1でLEDが点灯する



2008/11/28

TOPPERSプロジェクト認定

22



LEDのハードウェアを確認します。LEDは下のボードに実装されています。title STUDY IOと書かれている回路図を参照してください。LEDは回路図の左中央下に記載されています。LEDからの結線は以下のようになっています。J2コネクタとCPUとの結線も確認してください。

LED1→J2コネクタの10ピン→P03 (ポートP0の3ビット目)

LED2→J2コネクタの9ピン→P02 (ポートP0の2ビット目)

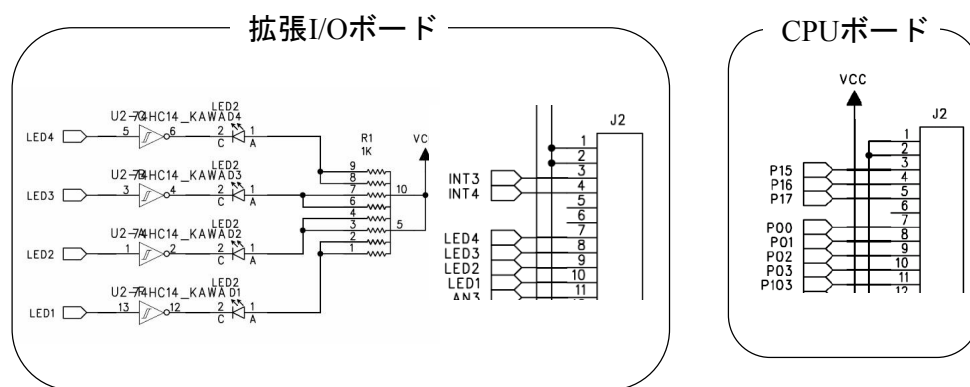
LED3→J2コネクタの8ピン→P01 (ポートP0の1ビット目)

LED4→J2コネクタの7ピン→P00 (ポートP0の0ビット目)

マニュアルの参照してポートP0が、どのアドレスに配置されているかを確認しましょう。マニュアルのプログラマブル入出力ポートをみるとP0～P3がアドレス0x03E0から0x03E3に配置されていることがわかります。

LEDの接続：マニュアルによる確認

- LEDはプルアップされている
- インバータを介してマイコンのプログラマブル入出力ポートに接続されている
 - マイコンから“1”を出力すると点灯する



2008/11/28

TOPPERSプロジェクト認定

23



今度はどのように点灯、消灯させるかを見てみましょう。LEDは電流が流れると点灯、流れないと消灯します。LEDのVccの反対の部分が0ボルトになると点灯することになります。Tile STUDY IOと書かれている回路図をまず参照しましょう。LED1～4よりCPU側の部分にインバータ(U2-A等)が入っています。インバータより右側が0ボルトになるには、左の側はプルアップされなければなりません。マイコン側が“1”のとき点灯し、“0”のとき消灯となります。

LEDの接続：レジスタ構成

- LEDはポート0に接続
 - LED1：ビット3：P03
 - LED2：ビット2：P02
 - LED3：ビット1：P01
 - LED4：ビット0：P00

- ポート0の構成レジスタ
 - ポート方向レジスタ：0x03E2
 - ポートレジスタ：0x03E0

ポートPi方向レジスタ(i=0~3, 6~8, 10)(注1)

シンボル	アドレス	リセット後の値
P00~P03	03E216, 03E316, 03E616, 03E716番地	0016
P06~P08	03E116, 03EF16, 03F216番地	0016
P010	03F616番地	0016

ビットシンボル	ビット名	機能	RW
PDL0	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)	RW
PDL1	ポートPi1方向ビット	1：出力モード (出力ポートとして機能)	RW
PDL2	ポートPi2方向ビット	(i=0~3, 6~8, 10)	RW
PDL3	ポートPi3方向ビット		RW
PDL4	ポートPi4方向ビット		RW
PDL5	ポートPi5方向ビット		RW
PDL6	ポートPi6方向ビット		RW
PDL7	ポートPi7方向ビット		RW

注1. PACRレジスタの設定をしてください。
80ピン版の場合、PACR2, PACR1, PACR0を“0112”にしてください。
64ピン版の場合、PACR2, PACR1, PACR0を“0102”にしてください。

ポートPiレジスタ(i=0~3, 6~8, 10)(注1)

シンボル	アドレス	リセット後の値
P0~P3	03E016, 03E116, 03E416, 03E516番地	不定
P6~P8	03EC16, 03ED16, 03F016番地	不定
P10	03F416番地	不定

ビットシンボル	ビット名	機能	RW
PL0	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讀むと、端子の レベルが読める。	RW
PL1	ポートPi1ビット	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	RW
PL2	ポートPi2ビット	0：“L”レベル	RW
PL3	ポートPi3ビット	1：“H”レベル(注1)	RW
PL4	ポートPi4ビット	(i=0~3, 6~8, 10)	RW
PL5	ポートPi5ビット		RW
PL6	ポートPi6ビット		RW
PL7	ポートPi7ビット		RW

注1. PACRレジスタの設定をしてください。
80ピン版の場合、PACR2, PACR1, PACR0を“0112”にしてください。
64ピン版の場合、PACR2, PACR1, PACR0を“0102”にしてください。

ルネサス M16C/29グループ
ハードウェアマニュアルP336,337より抜粋

2008/11/28

TOPPERSプロジェクト認定

24



2枚の回路図から読み取れることは以下の通りです。

【参照】

ハードウェアマニュアル (c:¥introductry)

ポートを利用するには、2つのレジスタを使用<ハードウェアマニュアル p329>
マニュアル iを数字で読み替えます。(PiはP0~3, P6~8, P10をあらわす)

ポートPi方向レジスタ・・・入出力ポートを入力に使用するか、出力に使用するか選択するためのレジスタ
ポートPiレジスタ・・・外部とのデータ入出力は、設定によりPiレジスタへの読み出しと書き出できます。

<ハードウェアマニュアル p336>

ポート方向レジスタ

①ポート方向レジスタ・・・該当ポートを入力として使用するか、出力として使用するかを指定します。

LEDに接続→出力とする マニュアルより、該当レジスタに “1”を書き込みます。

初期値が00

②ポート0方向レジスタのアドレス 0x03E2から始まる8ビットのレジスタ

③ポートレジスタ・・・入出力ポートに対応するビットに書くと、対応するビットを制御できます。

④ポート0レジスタのアドレス 0x3E0

ポートレジスタとは

ある値を書き込むと、内蔵されたさまざまな装置に何らかの動作を行わせたり、内蔵されたさまざまな装置に外部から与えられた情報を読み取ることができる仕掛けがなされた特殊なレジスタです。

出力ポート: データを書き込むと、書きこんだデータのビットに対応したピンの電圧が変化

入力ポート: 外部から与えられた電圧に応じて、レジスタのビット値が変化し、読み取ることができます。

LEDの接続：設定値

レジスタを操作して全てのLEDを点灯させる

- ポートを構成するレジスタとLEDとの接続を整理すると全てのLEDを点灯させるには以下の設定が必要
 - ポート方向レジスタ (0x03E2) の3,2,1,0ビットに‘1’を書き込み出力モードにする
 - ポートレジスタ (0x03E0) の3,2,1,0ビットに‘1’を書き込みポートの出力を‘1’にする
- レジスタへ書き込む値
 - 0x03E2 : 0x0F ("0000 1111")
 - 0x03E0 : 0x0F ("0000 1111")

2008/11/28

TOPPERSプロジェクト認定

25



LEDを操作するには、ポートP0方向レジスタに書き込み設定“1”を行い、ポートP0レジスタに“1”を書くと点灯、“0”書くと消灯となります。ポート方向レジスタは、初期値が00なので、最初は全部入力になっています。

全部点灯させるには

0x03E2番地に0x0F、0x03E0番地に0x0Fを書けば全部点灯となります。

LEDの操作 : 任意のLEDの点灯

- 次のパターンでLEDを点灯させるためのポートレジスタへ書き込む値を考え, 実際に確認せよ
 1. LED4:OFF LED3:OFF LED2:OFF LED1:OFF
 2. LED4:ON LED3:ON LED2:OFF LED1:OFF
 3. LED4:OFF LED3:ON LED2:ON LED1:OFF
 4. LED4:OFF LED3:OFF LED2:ON LED1:ON
 5. LED4:OFF LED3:ON LED2:OFF LED1:ON

2008/11/28

TOPPERSプロジェクト認定

26



デバッガを使って、上記の5つのパターンで点灯を行ってください。方向レジスタは一度設定すると変更の必要はありません。

ポートレジスタ (0x03e0)

8ビットなので、16進で表現すると2桁

1 0000 0x00

2 0011 0x03

3 0110 0x06

4 1100 0x0C

5 1010 0x0A

LEDの操作 : 任意のLEDの点灯

1. LED4:OFF LED3:OFF LED2:OFF LED1:OFF
 - 0x00 : "0000 0000"
2. LED4:ON LED3:ON LED2:OFF LED1:OFF
 - 0x03 : "0000 0011"
3. LED4:OFF LED3:ON LED2:ON LED1:OFF
 - 0x06 : "0000 0110"
4. LED4:OFF LED3:OFF LED2:ON LED1:ON
 - 0x0C : "0000 1100"
5. LED4:OFF LED3:ON LED2:OFF LED1:ON
 - 0x0A : "0000 1010"

ここからは、今まで行ったことでプログラムを作成します。

ポーリングプログラム

1. LEDプログラム

- ・プロジェクトの作成方法

2. スイッチプログラム

3. 宿題:タイマプログラム

4. 宿題:シリアルI/Oプログラム

ポーリングプログラミング：目的

周辺デバイスを操作するプログラムの書き方を学ぶ

- 初期設定, データの入出力, 事象の待ち方
 - スイッチのONやタイマのタイムアウトなどの外部事象はポーリングによって扱う
- プログラミング対象の周辺デバイス
 - LED(プロマブル入出力ポート出力)
 - スイッチ(プロマブル入出力ポート入力)
 - タイマ
 - シリアルI/O

ポーリングとは、該当するレジスタを繰り返しチェックする方式を指します。プログラムで作成する場合、**for**文などで実現します。ポーリングを用いて、周辺デバイスを制御するプログラムを作成してみましょう。

ポーリングプログラミング：プログラムファイル

- プログラムファイルの置き場所
 - 教材ディレクトリ¥program
- プログラム一覧
 - led_shift : LEDシフト点灯プログラム
 - led_count : LEDカウント点灯プログラム
 - switch_dip : DIPスイッチプログラム
 - switch_push : PUSHスイッチプログラム
 - timer : タイマプログラム
 - uart : シリアルI/Oプログラム

ここで学ぶプログラムはprogramディレクトリの下に用意してあります。

LEDプログラム : led_shift 概要

- 次のパターンでLEDを点灯させる
 - LED全てOFF → LED1 ON → LED2 ON → LED3 ON → LED4 ON → LED全てOFF



- 学習内容
 - プログラムの全体像の理解
 - ビット操作(セット, クリア)
 - デバイスレジスタ操作(様々な記法)
ポートからの出力
 - スタートアップルーチン, セクション
- 構成ファイル
 - led_shift.c
 - start.a30

2008/11/28

TOPPERSプロジェクト認定

31



led_shiftプログラムを参照して、LEDを制御するプログラムについて学習します。led_shiftはLEDを順番にシフトしながら点灯するプログラムです。ここではLEDの操作だけではなく、電源投入後のスタートアップルーチンやセクションについても学びます。

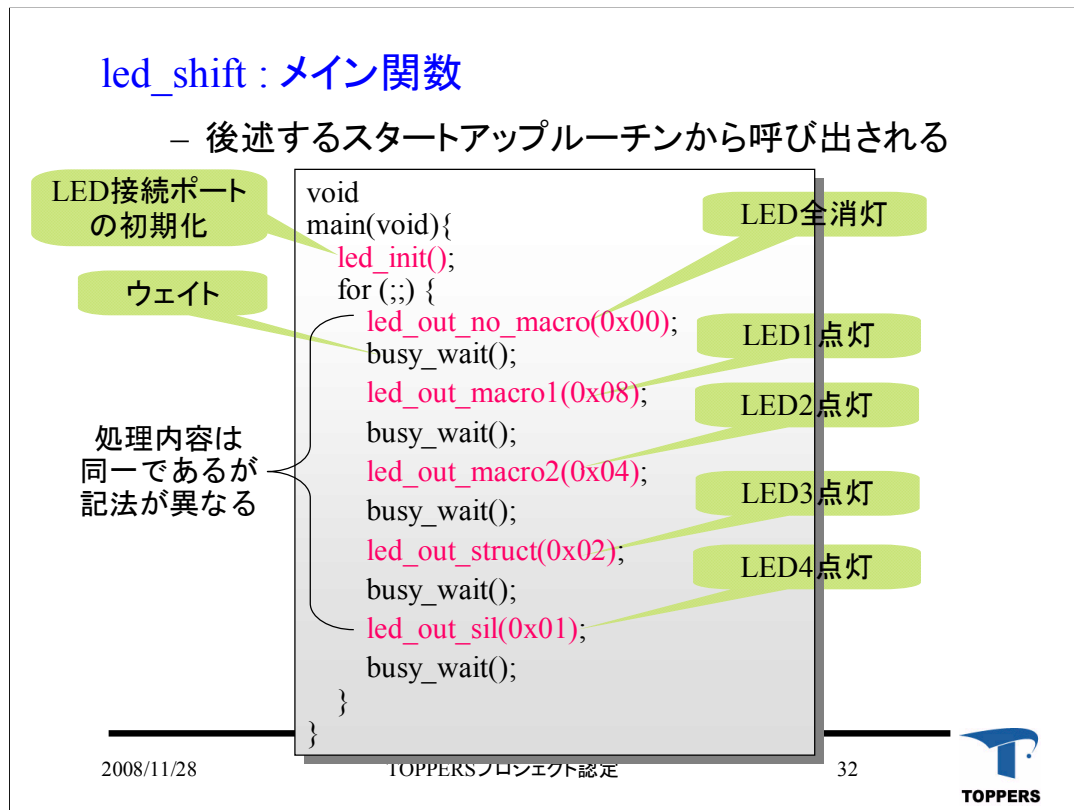
TMを起動してプロジェクトの追加から、program¥led_shift.tmkプロジェクトを開き、ビルドを行い、KD30デバッガでプログラムを実行してください。

led_shift¥led_shift.c

メインルーチン(C言語)

led_shift¥start.a30

スタートアップルーチン(アセンブラ言語)



エディタを使って、led_shift.cを表示します。

プロジェクトエディタ→all→led_shift.x30→led_shift.r30→led_shift.c

まず、ソースの下を行を参照します。そこにメイン関数 (main()) があります。使用している関数について説明します。

led_init() LEDの初期化を行う
 方向レジスタの初期化→出力に設定

busy_wait() 1秒間待つ。

LEDの点灯をいろいろな関数で行います。どれも、“1”のLEDが点灯、その他が消灯です。

led_out_no_macro()
 マクロを使用しないLED設定関数

led_out_macro1()
 マクロを使用したLED設定関数、volatile文を命令文に使用

led_out_macro2()
 マクロを使用したLED設定関数、volatile文をマクロに使用

led_out_struct()
 構造体を使用したLED設定関数

led_out_sil()
 SILを使用したLED設定関数
 SILとはトロンでガイドライン化されたデバイスアクセス手順

led_shift : ポートレジスタへの書き込み

- LED点灯・消灯制御
 - ポート0のポートレジスタ(0x3E0)への書き込み
 - 特定アドレスの読み書きが必要
- アセンブリコード

```
書き込み : MOV.B #00, 03E0H
読み込み : MOV.B 03E0H, R0L
```

- C言語
 - ポインタを使用
 - アドレスをポインタ変数にキャスト

```
書き込み : *((unsigned char *) (0x3E0)) = 00;
読み込み : tmp = *((unsigned char *) (0x3E0));
```

2008/11/28

TOPPERSプロジェクト認定

33



led_out_no_macro()関数を参照してください。この関数はマクロを使用せず、プログラムで直接ポート書き込みを行うプログラムです。

アセンブラ言語 (M16C用)を用いてポートP0への読み書きの記述について説明します。

MOV.size src,dest srcの値をdestに移します。

size B バイト(8ビット)W ワード(16ビット)

R0 16ビットのデータレジスタ 上位(R0H)と下位(R0L)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。この'H'は、16進数であることを示す、アセンブラ形式の書式です。

C言語で記述するとポートのアドレスをポインタに設定して、データ=文でデータの設定、読み取りができます。

```
*((unsigned char *) (0x03E0)) = 00;
```

0x03E0番地をunsigned char型のポインタとして、コンスタント値("0")を書き込みます。

```
tmp = *((unsigned char *) (0x03E0));
```

0x03E0番地をunsigned char型のポインタとして、取り出した値を変数tmpに格納します。

led_shift : volatile修飾子

- volatile
 - ポインタを用いてレジスタをアクセスする場合は、最適化を抑制するため、常にvolatile指定をつける必要がある

```
書き込み : *((volatile unsigned char *) (0x3E0)) = 00;
読み込み : tmp = *((volatile unsigned char *) (0x3E0));
```

- volatile修飾子の場所
 - 以下の記述でもコンパイルは通るが、最適化は抑制できないので注意が必要である(ポインタ変数がvolatile)
 - ポインタ変数が指し示す先をvolatileにしなければならない

```
tmp = *((unsigned char * volatile) (0x3E0));
```

2008/11/28

TOPPERSプロジェクト認定

34



ポインタを用いてポートにデータの読み出し書き込みを行う場合、**volatile**修飾子を付ける必要があります。**volatile**修飾子をデータが不定であることを表し、コンパイラで最適化(オブティマイズ)を行う場合、一度読み書きしたデータも、再読み書き時の値が不定ということで、最適化の対象となりません。

上記の説明のように**volatile**修飾子の位置に注意を行う必要があります。

led_shift : ポートレジスタ書き込み関数

- LED点灯・消灯制御
 - ポート0のポートレジスタ(0x3E0)への書き込み
 - アクセスサイズが1byteなので, unsigned char へのポインタへキャストして書き込む

```
void
led_out(unsigned char led_data){
    *((volatile unsigned char *) 0x3E0) = led_data;  /* 書き込み*/
}
```

今回はこれで十分だが、
再利用性を考えると不十分なコード

- 将来的にポート0のLEDが接続されているビット以外のビットに他の回路が接続された場合、このコードを実行する発生する不具合は？

2008/11/28

TOPPERSプロジェクト認定

35



マクロを使用しない関数で以下のような記述でも、問題なくLEDの操作が可能です。しかし、実際にLEDとして使用しているポートはビット3からビット0の4ビットのみで、他のビット7からビット4の4ビットが他の周辺デバイスの使用された場合、led_out関数の実行で、他のデバイスの設定を勝手に行ってしまうことになり、プログラム再利用性が悪くなります。

```
void led_out (unsigned char led_data)
{
    *((volatile unsigned char *)0x3E0) = led_data;  /* 書き込み */
}
```

led_shift : 変更値以外の保存

- 再利用性を考えると操作対象のビット以外は変更するべきではない
 - 操作対象のビット以外は変更前の値を書き込む
- 変更前の値をどこから読むか?
 - ポートレジスタから?

周辺回路のレジスタはメモリマッピングされているが、メモリとは異なり書き込んだ値が読み込めるとは限らないため注意が必要

M16Cのポートレジスタの仕様(M16C/29グループハードウェアマニュアルP329から引用)

- ポートレジスタは、出力データを保持するポートラッチと端子の状態を読む回路で構成されています。入力モードに設定しているポートレジスタを読むと端子の入力レベルが読め、書くとポートラッチに書き込みます。出力モードに設定しているポートレジスタを読むとポートラッチを読み、書くとポートラッチに書き込みます。ポートラッチに書いた値は端子から出力されます。

M16Cのポートレジスタは現在の出力値を保持する仕様

- ポートレジスタが値を保持しない場合はグローバル変数に値を保持する

2008/11/28

TOPPERSプロジェクト認定

36



他の4ビットを設定を変えずに、LEDの4ビットのみを設定するにはどうすればよいのでしょうか？一度、ポートの内容を読み込み、設定する4ビットのみを変更して書き込めばled_outでLEDに関係しない値を書き換えてしまうことはありません。M16Cのポートの場合、ポートに書き込みを行った値は読み込みによって取り出せますが、ハードウェアによっては読み出した場合、書きこんだ値とは別の値が読み出される設計のものが 있습니다。(レジスタのロジックを減らせる)このようなポートの場合、書き込む値をグローバル変数にも値を書き込み、関数ごとにグローバル変数値も正しく書き換えるようなプログラムを作成すれば、問題回避できます。

led_shift : 特定ビットのみの変更

- 変更前の値はポートレジスタから取得可能
- 取得した値のうち、LEDが接続されているビット3,2,1,0にのみ引数 led_data の値を反映させる
- 例えば..
 - ポートレジスタの値(LED3:ON) : 0x22 ("0010 0010")
 - 引数led_data(LED4:ON) : 0x01 ("0000 0001")
 - 書き込みたい値は : 0x21 ("0010 0001")
- プログラムによる0x21の生成手順
 - ポートレジスタの値のビット3,2,1,0をクリア(0x20("0010 0000"))
 - led_dataのビット3,2,1,0のみを取り出す
 - 1.と2.で生成した値の論理和を生成
("0010 0000"|"0000 0001" = "0010 0001")
- 特定ビットのクリアや特定ビットの取り出しが必要
 ビット演算により実現

2008/11/28

TOPPERSプロジェクト認定

37



前のページで説明で行った。ポートP0の下4ビットのみの書換え方法を説明します。もともとのポートP0の値を0x22("0010 0010")とします。この状態でLED3のみ点灯です。これをLED4のみ点灯となるようにled_dataの値が0x01("0000 0001")が設定されたとします。もとの値の下4ビットのみ書換えをおこないたいので、書き込みたい値は0x21(0010 0001)です。

これをプログラムで行うには以下の手順で行います。

- 1) ポートP0から値0x22("0010 0010")を読み出す。
- 2) 読み出した値の4ビットをゼロクリアする。0x22("0010 0010") → 0x20("0010 0000")
- 3) led_dataの値0x01("0000 0001")を取り出し、不要な上4ビットをゼロクリアする。
0x01("0000 0001") → 0x01("0000 0001")
- 4) ふたつの値の論理和をつくる
0x20("0010 0000") | 0x01("0000 0001") → 0x21("0010 0001")
- 5) できた値をポートP0に書き込む

このような演算をビット演算と言います。

led_shift : led_outの変更

- ポートレジスタのビット3,2,1,0以外は元の値を保持

```
void
led_out_no_macro(unsigned char led_data){
    unsigned char reg;
    reg = *((volatile unsigned char *) 0x3E0);
    reg = reg & 0xf0;
    led_data = led_data & 0x0f;
    reg = reg | led_data;
    *((volatile unsigned char *) 0x3E0) = reg;
}
```

1. 元の値を読み込み

2. ビット3,2,1,0のみをクリア

3. ビット3,2,1,0以外は0に

4. 2.と3.のOR

5. 書き込み

2008/11/28

TOPPERSプロジェクト認定

38



これをC言語のプログラムで記述します。

- 1) ポートP0から値0x22 (“0010 0010”)を読み出す。

```
reg = *((volatile unsigned char *)0x3E0);
```

- 2) 読み出した値の4ビットをゼロクリアする。0x22 (“0010 0010”) → 0x20 (“0010 0000”)

```
reg = reg & 0xf0;
```

- 3) led_dataの値0x01 (“0000 0001”)を取り出し、不要な上4ビットをゼロクリアする。

```
0x01 (“0000 0001”) → 0x01 (“0000 0001”)
```

```
led_data = led_data & 0x0f;
```

- 4) ふたつの値の論理和をつくる

```
0x20 (“0010 0000”) | 0x01 (“0000 0001”) → 0x21 (“0010 0001”)
```

```
reg = reg | led_data;
```

- 5) できた値をポートP0に書き込む

```
*((volatile unsigned char *)0x3E0) = reg;
```

led_shift : マクロによる記述1

定数はマクロ(記号定数)で記述する

- レジスタのアドレスや接続ビットの値を直接プログラム中に記述すると可読性が悪くなる(マジックナンバー)
- レジスタ値や接続ビットの変更に対応が可能
- ポート0レジスタのマクロ定義

```
#define BADR_SFR_P0      0x3E0
```

- ポート0のLEDの接続ビット定義

```
#define LED1      0x08      /* LED1ビット定義 */
#define LED2      0x04      /* LED2ビット定義 */
#define LED3      0x02      /* LED3ビット定義 */
#define LED4      0x01      /* LED4ビット定義 */
#define LED_MASK  (LED1 | LED2 | LED3 | LED4) /* LEDのマスク */
```

2008/11/28

TOPPERSプロジェクト認定

39



led_out_macro1()関数を参照してください。このプログラムはled_out_no_macro()関数をマクロを使って記述したプログラムです。使用するマクロについて説明を行います。マジックナンバーとはプログラムの中に表れるリテラル値(具体的な数値)を示します。プログラムを数値で記述するとプログラムの可読性が低下します。例えば、0x3E0という値は、値のみでは何を指す値かわかりません。これをBADR_SFR_P0と書くと、SFRレジスタのP0ポートのアドレス(BADR)というように何をさすかが判るようになります。

led_out_no_macro()関数中のマジックナンバーをマクロで記述します。

```
#define BADR_SFR_P0      0x3E0      ポートP0レジスタのアドレス
```

```
#define LED1      0x08      LED1に対応したビット定義
```

```
#define LED2      0x04      LED2に対応したビット定義
```

```
#define LED3      0x02      LED3に対応したビット定義
```

```
#define LED4      0x01      LED4に対応したビット定義
```

プログラム中のLEDのマスク値は以下の定義となります。

```
#define LED_MASK      (LED1|LED2|LED3|LED4)
```

led_shift : マクロによるled_outの書き換え1

- 何をしているプログラムなのか分かりやすくなる

```
void
led_out_macro1(unsigned char led_data){
    unsigned char reg;

    reg = *((volatile unsigned char *)BADR_SFR_P0);
    reg = reg & ~LED_MASK;

    led_data = led_data & LED_MASK;

    reg = reg | led_data;
    *((volatile unsigned char *)BADR_SFR_P0) = reg;
}
```

led_out_no_macro()関数をマクロを使って書き換えます。気になる部分は、~LED_MASKです。上位4ビットを取り出すために、LED_MASKの否定を利用します。

reg = reg & ~LED_MASK 上位4ビットを取り出す

led_data = led_data & LED_MASK 下位4ビットを取り出す

led_shift : マクロによる記述2

デバイスレジスタでは, ポインタへのキャストの部分も含めてマクロ化する場合が多い

- 型の指定間違いを防ぐ
 - デバイスレジスタにはアクセスサイズがある
- 記述が簡素になる
- マクロ化の方法や命名規則はプログラムやプロジェクト単位で一定化しておく

- ポート0レジスタ

```
#define BREG_SFR_P0    ((volatile unsigned char *)BADR_SFR_P0)
```

- ポインタ部までマクロ化

```
#define BREG_SFR_P0    *((volatile unsigned char *)BADR_SFR_P0)
```

2008/11/28

TOPPERSプロジェクト認定

41



led_out_macro2()関数を参照してください。Led_out_macro1()関数のレジスタのマクロに型の指定を行います。このようにしておく、プログラム中のレジスタ型の記述ミスを軽減できます。

led_shift : マクロによるled_outの書き換え2

```
void
led_out_macro2(unsigned char led_data){
    unsigned char reg;

    reg = *BREG_SFR_P0;
    reg = reg & ~LED_MASK;

    led_data = led_data & LED_MASK;

    reg = reg | led_data;
    *BREG_SFR_P0 = reg;
}
```

- 1行での記述も可能

```
*BREG_SFR_P0 = (*BREG_SFR_P0 & ~LED_MASK)
                | (led_data & LED_MASK);
```

2008/11/28

TOPPERSプロジェクト認定

42



led_out_macro1()関数を、型付のマクロに書き換えれば、プログラムの記述も見やすくなります。

【補足】

&と|の優先順位は、&のほうが高いため、()はなくても構わない。

MISRA-Cでは、()を推奨

led_shift : 構造体による記述

構造体を用いてデバイスレジスタのアドレスを指定する

- ビット単位の読み書きが容易に記述可能
 - コンパイラ依存であり, 利用できないコンパイラもある
- 記述方法もコンパイラによって異なる

```
/* ビット構造体 */
struct bit_def {
  unsigned char b0:1;
  unsigned char b1:1;
  unsigned char b2:1;
  unsigned char b3:1;
  unsigned char b4:1;
  unsigned char b5:1;
  unsigned char b6:1;
  unsigned char b7:1;
};
```

ビット
フィールド

```
/* バイト構造体 */
union byte_def{
  struct bit_def bit;
  char byte;
};
```

```
/* アドレス指定(コンパイラ依存) */
#pragma ADDRESS p0_addr 03e0H

/* ポートレジスタ定義 */
union byte_def p0_addr;
```

2008/11/28

TOPPERSプロジェクト認定

43



今回は、led_out_struct()関数を参照してください。ポートレジスタを構造体を使って記述します。レジスタの各ビットを構造体のビットフィールドで指定します。

コンパイラマニュアル(D-5)

ビットフィールド型は、最下位のビットから配置されます(リトルエンディアンの場合逆の場合がある)。

#pragma ADDRESS 変数名 絶対アドレス

変数の絶対アドレスを指定

通常I/O変数に対して使用するため、volatile指定がなくても、volatile指定がされているものとして処理されます。(コンパイラ依存なので、すべてのコンパイラで利用できるわけではない)

#if 0 ~ #endifまでに、今までのアドレスをポインタにキャストした場合の記載もあります。

【補足】コンパイラマニュアルB-45

この宣言によって指定した絶対アドレスは文字列として、アセンブラファイルに展開される。数値の記述形式はアセンブラ数値表現で。

16進数は、最後にH又は、hをつけます。

#pragma ADDRESSにより指定された変数

- extern、static等の記憶クラスはすべて無効
- 関数外で定義された変数に対してのみ有効
- この宣言を行う前に宣言した変数に対しても有効。使用したら無効
- 通常I/O変数に対して使用するため、volatile指定がなくても、volatile指定がされているものとして処理されます。

→確認！

```
#define P0_ADDR (*(volatile union byte_def *)0x 03e0)
```

1ビットずつ指定できるが、ビットフィールドをサポートしていないコンパイラもあるので注意が必要です。ポータビリティが悪くなりますが、このコンパイラしか使わないとわかっていれば問題ありません。

led_shift : 構造体による書き換え

バイトアクセス
による記述

```
void
led_out_struct(unsigned char led_data){
    unsigned char reg;
    reg = p0_addr.byte;
    reg = reg & ~LED_MASK;
    led_data = led_data & LED_MASK;
    reg = reg | led_data;
    p0_addr.byte = reg;
}
```

ビットアクセス
による記述

```
void
led_out_struct(unsigned char led_data){
    p0_addr.bit.b0 = ((0x01 & led_data) == 0x01)? 1 : 0;
    p0_addr.bit.b1 = ((0x02 & led_data) == 0x02)? 1 : 0;
    p0_addr.bit.b2 = ((0x04 & led_data) == 0x04)? 1 : 0;
    p0_addr.bit.b3 = ((0x08 & led_data) == 0x08)? 1 : 0;
}
```

2008/11/28

TOPPERSプロジェクト認定

44



バイトアクセス

今までポインタのキャストを使ってレジスタの8ビットの値を読んでいた部分を、バイトアクセスで実現しています。

ビットアクセス

条件演算子 補助資料で説明しています。1ビットずつ操作するので、マスク処理等がいらなくなります。

led_shift : デバイスアクセス関数による記述

デバイスへのアクセスを専用の関数により実現

- プログラムのハードウェア依存性が弱まる
- シミュレーション環境への対応が容易

例) デバイスドライバ設計ガイドラインのアクセスインタフェース

– バイトデータの読み込みと書き込み

```
unsigned char
sil_reb_mem(unsigned int addr){
    return *((volatile unsigned char *)addr);
}
```

```
void
sil_wrb_mem(unsigned int addr, unsigned char bdata){
    *((volatile unsigned char *)addr) = bdata;
}
```

2008/11/28

TOPPERSプロジェクト認定

45



led_out_sil() 関数を参照してください。デバイスへアクセスするときは、デバイスアクセス関数を使って書くようにします。特定のメモリへのアクセスする場合は、読み込む関数書き込む関数を通してアクセスするようにします。ここで使用しているデバイスアクセス関数は、ITRONデバイスドライバ設計ガイドラインで定義しているSIL関数を使用しています。これにより、ハードウェア依存性が弱まり、コンパイラが変わった時は、関数のみを変えればよく、ドライバの移植性が向上します。また、シミュレーション環境への対応が容易になります。(実機で動かす場合はこのままで、シミュレーション環境のときには、この関数のみを環境用に変えればよい。)

led_shift : デバイスアクセス関数による書き換え

- アクセスサイズによって呼び出す関数が異なる

```
void  
led_out_sil(unsigned char led_data){  
    unsigned char reg;  
  
    reg = sil_reb_mem(BADR_SFR_P0);  
    reg = reg & ~LED_MASK;  
  
    led_data = led_data & LED_MASK;  
  
    reg = reg | led_data;  
    sil_wrb_mem(BADR_SFR_P0, reg);  
}
```

デバイスアクセス関数はデバイスのポートのサイズにより、呼び出す関数がことなります。ポートP0レジスタはバイトサイズなので、バイトリードアクセス関数とバイトライトアクセス関数を使用します。

led_shift : 初期化関数 led_init()

- ポートを出力方向に設定し, 全LEDを消灯
- LED接続ビット以外を変更しないようにする

```
void
led_init(void){
    unsigned char reg;
    /* 方向レジスタ出力設定 */
    reg = *((volatile unsigned char *)BADR_SFR_PD0);
    reg = reg | LED_MASK;
    *((volatile unsigned char *)BADR_SFR_PD0) = reg;
    /* 初期状態は消灯 */
    reg = *((volatile unsigned char *)BADR_SFR_P0);
    reg = reg & ~LED_MASK;
    *((volatile unsigned char *)BADR_SFR_P0) = reg;
}
```

3,2,1,0ビット
をセット

3,2,1,0ビット
をクリア

2008/11/28

TOPPERSプロジェクト認定

47



初期化関数led_init()を参照してください。処理内容は以下の通りです。

- 1) ポートP0方向レジスタのLED対応ポートを出力に設定します。
- 2) LEDを全消灯に設定します。

led_shift : ビジーウェイト関数 busy_wait()

- forループにより任意時間待ちを生成
- ループ回数はforループ1回の実行時間から求める
 - 実際には適当にためして決定する
- ループ変数はlong型
 - M16Cはintが16bitであるため、0xffffまでしか扱えず十分な時間を確保できないため

```
#define DELAY_LOOP 0x80000L

void
busy_wait(void){
    volatile long i;
    for(i = 0; i < DELAY_LOOP; i++);
}
```

Long(32bit)
指定

busy_wait()関数を参照してください。ソフトウェアの空実行により、一定時間の待ちをつくれます。およそ1秒くらいに設定しています。

led_shift : start.a30

アセンブリ言語で記述されたファイル

- 可変ベクタ以外は各プログラムで共通に使用できる
- セクションの配置
 - RAM領域とROM領域
- ベクタテーブル
 - 固定ベクタ
 - 可変ベクタ
- スタートアップルーチン
 - セクションの初期化
 - main関数の呼び出し

2008/11/28

TOPPERSプロジェクト認定

49



スタートアップルーチンについて説明を行います。エディタを使ってstart.a30を開きます。

プロジェクトエディタ→all→start.r30→start.a30

スタートアップルーチンはプロセッサレジスタやC言語に準拠しないデータをアクセスしますので、アセンブラ言語で記述されています。M16Cの場合、可変ベクタ以外はプログラムの関係なく使用できます。プログラム以外にはセクションの配置やベクタテーブルの定義も行います。

スタートアップルーチンは電源が入ったときや、リセットされた時点で実行するプログラムで、スタックの設定、セクションの初期化、main関数の呼び出しを行います。

led_shift : start.a30 セクションの配置

- start.a30で指定

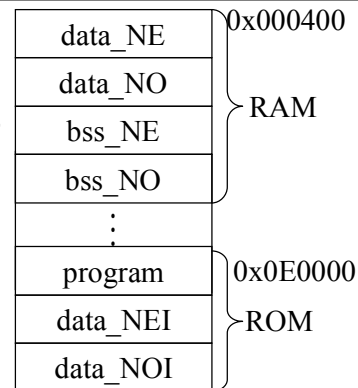
```
; ----- RAM 領域 -----
.SECTION data_NE,DATA
.ORG      00400h
.SECTION data_NO,DATA,ALIGN
.SECTION bss_NE,DATA,ALIGN
.SECTION bss_NO,DATA,ALIGN
```

```
; ----- ROM 領域 -----
.SECTION program,CODE
.ORG      E0000h
.SECTION data_NEI,ROMDATA,ALIGN
.SECTION data_NOI,ROMDATA,ALIGN
```

セクション名

- バス幅が16ビットの場合にデータアクセスを効率化するため、セクション名_NEには偶数データint (2byte)を置き、セクション名_NOには奇数データchar (1byte)を配置する

- data_NO/E : 初期値ありデータ
- bss_NO/E : 初期値なしデータ
- data_NO/EI : 初期値ありデータの初期値



2008/11/28

TOPPERSプロジェクト認定

50



セクションの設定について説明します。．SECTION以後の記述は設定したセクションの記述となります。セクション以後に．ORGの指定があると．ORG以後の記述は．ORGで設定されたアドレスから配置されます。

．SECTION セクション名 セクションタイプ ,ALIGN

セクションタイプには、CODE ROMDATA DATA のいずれかを記述できる

CODE: プログラムを記述する領域 ROMに配置

ROMDATA: プログラム以外の固定データ ROMに配置

DATA: 内容が変更可能なメモリを配置 RAMに配置

ALIGN指定がある場合、ln30がセクションの始まりを偶数番地に割り当てる

．ORG

セクションを絶対属性にする

led_shift : start.a30 固定ベクタ

- M16Cはリセット後, 0xFFFFDCから始まる固定ベクタの 0xFFFFFCに置かれたリセットベクタで指定した番地から実行を開始する
- .org命令によりセクションの絶対番地が指定可能
- リセット後, start(スタートアップルーチン)にジャンプ

4byteデータ	.SECTION fvector
	.ORG 0fffdch
	.LWORD _irregular_ext_handler_entry ; 00 0xdc
	.LWORD _irregular_ext_handler_entry ; 01 0xe0
	.LWORD _irregular_ext_handler_entry ; 02 0xe4
	.LWORD _irregular_ext_handler_entry ; 03 0xe8
関数startの アドレス	.LWORD _irregular_ext_handler_entry ; 04 0xec
	.LWORD _irregular_ext_handler_entry ; 05 0xf0
	.LWORD _irregular_ext_handler_entry ; 06 0xf4
	.LWORD _irregular_ext_handler_entry ; 07 0xf8
	.LWORD start ; RESET 0xfc

2008/11/28

TOPPERSプロジェクト認定

51



固定ベクタは0xFFFFDC番地から配置します。固定ベクタはfvectorセクションに設定しています。0xfffffc番地のベクタはRESETベクタで、電源投入時やリセット時の開始番地を指定します。ここでは、スタートアップルーチンの先頭アドレス(start)が設定されています。

led_shift : start.a30 スタートアップルーチン

- アセンブリ言語で記述
 - 設定しているレジスタをマニュアルでチェック
 - C言語と同様に定数はマクロ化

```
start:
ldc    #C_FLG, flg          ; フラグレジスタの設定
ldc    #RAMEND_ADR, isp     ; 割り込みスタックポインタセット
bset   PRC1                 ; プロセッサモードレジスタ書き込みイネーブル
mov.b  #C_PM0, PM0          ; シングルチップモード
mov.b  #C_PM1, PM1          ; 非拡張、ノーウェイト
bclr   PRC1                 ; プロセッサモードレジスタ書き込みディゼーブル
bset   PRC0                 ; クロックコントロールレジスタ書き込みイネーブル
mov.b  #C_CM2, CM2          ; システムクロックをメインクロックに
mov.b  #C_CM0, CM0          ; 発信
mov.b  #C_CM1, CM1          ; 分周なし
bclr   PRC0                 ; レジスタ書き込みディゼーブル
bset   PRC2                 ; 端子割り当て制御レジスタ書き込みイネーブル
mov.b  #C_PACR, PACR        ; 端子割り当て設定
bclr   PRC2                 ; レジスタ書き込みディゼーブル
ldintb #VECTOR_ADR          ; ベクタテーブルの設定
```

2008/11/28

TOPPERSプロジェクト認定

52



スタートアップルーチンの最初は、プロセッサレジスタの初期化、ハードウェア設定を行います。

レジスタ設定アセンブラ言語の簡単な説明:

ldc src,dest

dest(専用レジスタ) ← **src**

bset dest

dest ← 1

bclear dest

dest ← 0

ldintb src

srcをINTBに転送

led_shift : start.a30 セクションの初期化

```

;----- DATA領域の初期化 -----
    mov.w    #(topof bss_NE),R3
    mov.w    #(topof data_NE),A1
    sub.w    A1,R3
    jz       bss_clear
    mov.w    #(topof data_NEI & 0ffffh),A0
    mov.b    #(topof data_NEI >> 16),R1H
    smovf.b  ;
;----- BSS領域の初期化 -----
bss_clear:
    mov.w    #(topof bss_NE),A0
    mov.w    #0,R0
clear_loop:
    mov.w    R0,[A0]
    add.w    #2,A0
    cmp.w    #RAMEND_ADR,A0
    jne      clear_loop

```

data_NEからbss_NE
にdata_NEIをコピー

R1H,A0を転送元番地
A1を転送先番地
R3を転送回数

bss_NEから
RAMEND_ADRを'0'に

2008/11/28

TOPPERSプロジェクト認定

53



次にセクションの初期化を行います。データ領域 (**data_NE**、**data_NO**) は初期値のある書換え可能データです。そのため、初期値をROM領域 (**data_NEI**、**data_NOI**) から持ってきて、データ領域にコピーします。BSS領域 (**bss_NE**、**bss_NO**) はゼロクリアしたデータ領域です。この領域をゼロクリアします。グローバル変数は初期化しないと0になっている。

ここで使用するアセンブラ言語

smovf.b

20ビットで示される転送元番地から16ビットで示される転送先番地にアドレスの加算方向へ転送をおこなう

転送元番地の上位4ビットはR1H,転送元番地の下位16ビットはA0、転送先番地はA1、転送回数はR3に設定

mov src,dest : src をdestに転送

topof オペラントに指定したセクションの開始アドレスを値として扱う

sub src,dest : dest = dest-src

add src,dest : dest = dest + src

cmp A,B AとBを比較して

JNZ A != Bならばジャンプ

JZ A==Bならばジャンプ

led_shift : start.a30 main関数の呼び出し

- ハードウェアの初期化, data,bssの初期化が終了後, main関数を呼び出す

```

;          ***** メインルーチンへ *****
.glb      _main
jsr.a     _main          ; --> main()

```

- C言語関数はコンパイルされると"_"が前に付く
 - コンパイラやプロセッサにより異なる
- main()という名前で関数を作成するとCのランタイムの初期化処理を自動的に呼び出すルーチンを関数内に入れるコンパイラもあるので注意

最後にmain関数にジャンプします。

M16C用のコンパイラは、コンパイルすると関数名の前に_がつきます。また、mainという名前が特別な名前の関数処理として、勝手にコンパイラが初期化処理を追加するコンパイラもありますので注意が必要です。うまくいかない場合は、main関数の名前を変える必要があります。

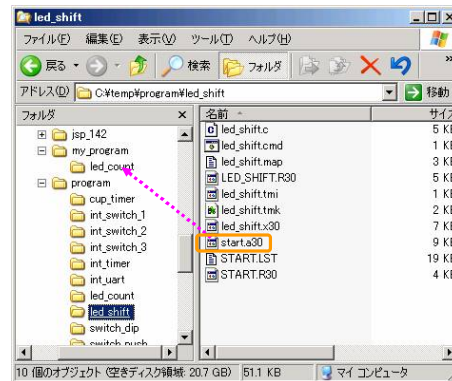
ポーリングプログラム

1. LEDプログラム
 [・プロジェクトの作成方法](#)
2. スイッチプログラム
3. 宿題:タイマプログラム
4. 宿題:シリアルI/Oプログラム

プロジェクトの作成方法

新規プロジェクトの作成方法を学習

- プロジェクトのためのディレクトリ(led_count)を作成
 - 作成場所に制約はない
- 教材ディレクトリ¥program¥led_shiftにあるstart.a30を作成したフォルダにコピーする



2008/11/28

TOPPERSプロジェクト認定

56



実習用のプログラムled_countを作成するためのプロジェクトを作成します。my_program¥led_countというディレクトリの下にled_countプロジェクトを作成します。まず、my_program¥led_countというディレクトリを作成します。この中にprogram¥led_shiftにあるstart.a30をコピーします。

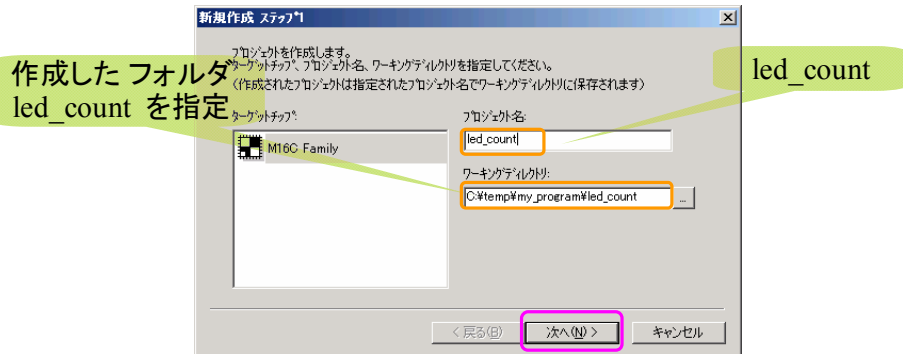
プロジェクトの作成方法：TMの起動

- TMを起動してプロジェクト作成ボタンをクリックします



ここをクリック!

- プロジェクト作成ウィザードが開始されるのでプロジェクト名とワーキングディレクトリを設定して"次へ"をクリック



2008/11/28

TOPPERSプロジェクト認定

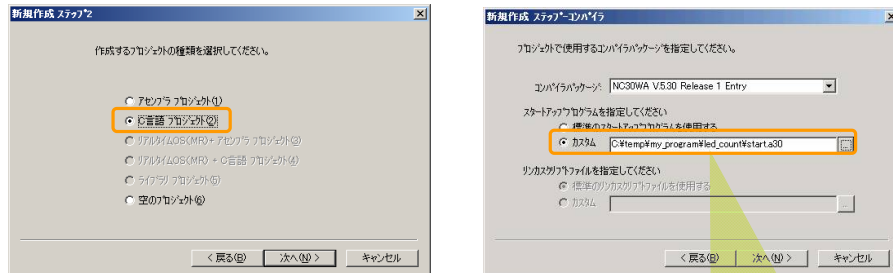
57



TMを起動します。プロジェクト作成ボタンをおします。新規作成ステップのプロジェクト名に「led_count」、ワークディレクトリにmy_program\led_countのディレクトリを設定します。「次へ」を押します。

プロジェクトの作成方法：スタートアップファイル

- C言語プロジェクトを選択して"次へ"
- 次のダイアログでは先ほどコピーしたstart.a30を選択



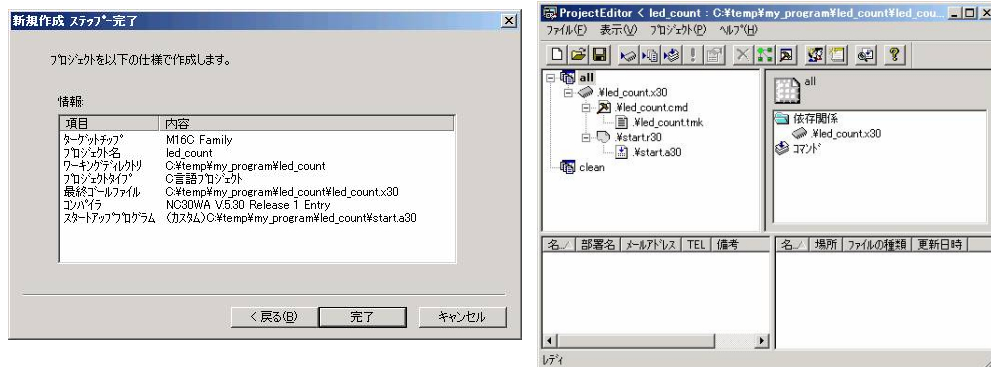
led_shiftからコピーした
start.a30 を指定

新規作成ステップ2ダイアログはそのまま「次へ」を押します。

新規作成ステップ3-コンパイラでは、スタートアッププログラムを指定してくださいでは、カスタムを指定して、**start.a30**を選択してください。その後「次へ」を押します

プロジェクトの作成方法：完了

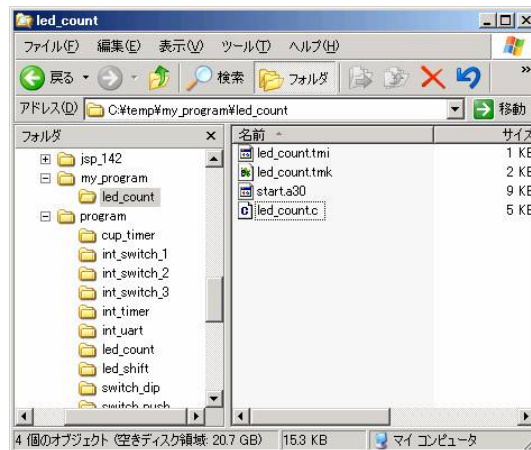
- 完了ダイアログが表示され“完了”を押すとプロジェクトが作成され、ProjectEditorが表示される



新規作成ステップー完了で「完了」を押すとProjectEditorが表示されます。

プロジェクトの作成方法：C言語プログラム

- コンパイルするC言語プログラムを用意する
- ¥led_shift¥led_shift.cの内容をコピーして、led_countのディレクトリに led_count.c として作成する



2008/11/28

TOPPERSプロジェクト認定

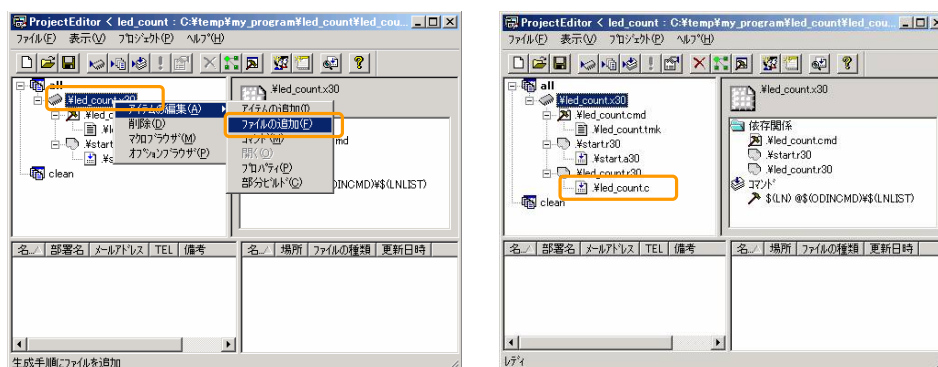
60



program¥led_shift¥led_shift.cをコピーし、led_count.cという名前に書き換えます。

プロジェクトの作成方法：ファイルの追加

- led_count.c をプロジェクトに追加する
- ProjectEditor の led_count.x30を右クリックして“アイテムの編集”→“ファイルの追加”を選択して led_count.cを追加する



2008/11/28

TOPPERSプロジェクト認定

61



コピーしたled_count.cをプロジェクトに追加します。ProjectEditor内のled_count.x30を右クリックして「アイテムの編集」→「ファイルの編集」→「ファイルの追加」を選択しled_count.cを選択します。

「リビルド」を押して、きちんとビルドできることを確認してください。

LEDプログラム : led_count 概要

- LEDを2進数の表示器とみなし、一定時間毎に0x0から0xfまでを表示する(カウントアップ)



- 学習内容
 - プロジェクトの作成方法
 - 出力値のエンコード方法
- 作成方法
 - led_shiftをベースに作成

led_countプログラムを作成します。プロジェクトの作成で作ったled_count.cを書き換えて作成を行います。led_countは、LEDを2進数の表示機にみたててカウントアップを行うプログラムです。ここではLEDの出力値のエンコード方法を学習します。

led_count : メイン関数

- 無限ループで一定時間毎に変数を引数にLEDの表示を更新する関数を呼び出し, 変数をインクリメントする
- led_out_bdigit()は引数の下位4bitをLEDに反映させる

```
void
main(void){
    unsigned char count = 0;
    led_init();
    for (;;) {
        led_out_bdigit(count++);
        busy_wait();
    }
}
```

ハードウェア
初期化

LED設定

メイン関数を修正します。led_out_bdigit()関数でcountを表示するようにします。

led_count : 作成

プログラム led_count を作成せよ

- 手順
 - led_out_bdigit()を作成
 - 引数の下位4bitをそのままポートに書き込むと正しく表示されないため(LED1はビット3に接続), 引数見て, 対応するビットをONにする必要がある
 - LED接続ポートへの書き込みは, led_out_xxx() 関数を用いる

2008/11/28

TOPPERSプロジェクト認定

64



今度は、led_out_bdigit()関数を作成します。LEDの表示はled_out_macro2を利用して作成します。LED表示は以下の手順となります。

count=1のとき

LED1点灯 3ビット目1を立てます

count=2のとき

LED2点灯 2ビット目1を立てます

count=3のとき

LED1 LED2点灯 3、2ビット目に1を立てます。

◎led_shiftとの違いをあえてあげると、一度に複数のLEDを点灯させる必要があります。

led_count : led_out_bdigit()

- 引数の各ビットをチェックし, 0ビット目が'1'ならLED1をON, 1ビット目が'1'ならLED2をONとなるようにled_dataを設定する
- led_dataの初期値は, 全てのLED OFF とする0x00にしておく

```
void
led_out_bdigit(unsigned char data){
    unsigned char led_data = 0x00;
    if ((data & 0x01) == 0x01) {
        led_data = led_data | 0x08;
    }
    if ((data & 0x02) == 0x02) {
        led_data = led_data | 0x04;
    }
    ...省略...
    led_out(led_data);
}
```

初期値

LED1設定

LED2設定

LED設定

2008/11/28

TOPPERSプロジェクト認定

65



一般的なプログラムでは、テーブルを作成して以下のように変換します。

```
static unsigned char led_table[16] = {
    0x00, 0x08, 0x04, 0x0c,
    0x02, 0x0a, 0x06, 0x0e,
    0x01, 0x09, 0x05, 0x0d,
    0x03, 0x0b, 0x07, 0x0f
};

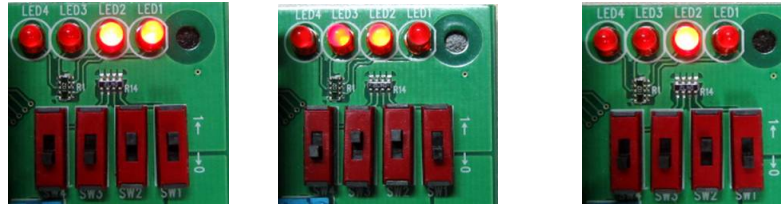
void
led_out_bdigit(unsigned char data){
    led_out(led_table[data & 0x0f]);
}
```

ポーリングプログラム

1. LEDプログラム
 - ・プロジェクトの作成方法
2. [スイッチプログラム](#)
3. 宿題:タイマプログラム
4. 宿題:シリアルI/Oプログラム

スイッチプログラム : switch_dip 概要

- ディップスイッチの状態をLEDに反映させる

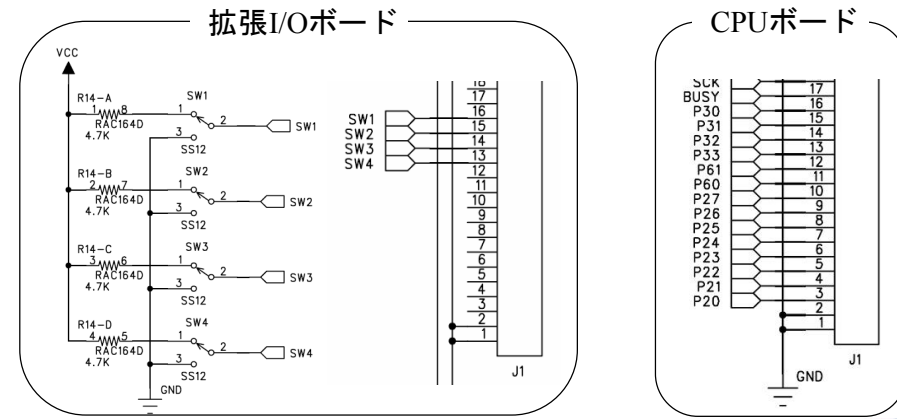


- 学習内容
 - ポートからの読み込み
- 作成方法
 - led_shiftをベースに作成

ディップスイッチの状態をLEDに表示するプログラムを作成します。スライドではプログラムの作成となっていますが、時間が足りないと思われるのでプログラムの説明を中心に行います。まず、**switch_dip**をビルドして動作の確認を行います。

switch_dip : ディップスイッチの接続

- マニュアルにより確認する
- マイコンのプログラマブル入出力ポートに接続されている
- ONでVcc, OFFでGNDに接続する
 - マイコンからはONで'1'が, OFFで'0'が読み込める



2008/11/28

TOPPERSプロジェクト認定

68



デップスイッチの回路図を確認します。title STUDY IOの左上にデップスイッチの回路図が記載されています。以下のようにポートP3を使用しています。

- SW1→J1コネクタの16ピン→P30 (ポートP3の0ビット目)
- SW2→J1コネクタの15ピン→P31 (ポートP3の1ビット目)
- SW3→J1コネクタの14ピン→P32 (ポートP3の2ビット目)
- SW4→J1コネクタの13ピン→P33 (ポートP3の3ビット目)

デップスイッチをオンにすると、プルアップされて“1”、オフにするとグラウンドに電流が流れ“0”となります。

switch_dip : ディップスイッチの接続ポート

- ポート3に接続
 - SW1 : ビット0 : P30
 - SW2 : ビット1 : P31
 - SW3 : ビット2 : P32
 - SW4 : ビット3 : P33
- ポート3の構成レジスタ
 - ポート方向レジスタ : 0x03E7
 - ポートレジスタ : 0x03E5
- 初期化
 - ポート3のビット3,2,1,0を入力モードへ
- スイッチ状態の取得
 - ポート3レジスタを読み込む

ポートPi方向レジスタ(i=0~3, 6~8, 10)(注1)

シンボル	アドレ	リセット後の値
P00~P03	03E216, 03E316, 03E616, 03E716番地	0016
P06~P08	03EE16, 03EF16, 03F216番地	0016
PD10	03F616番地	0016

ビットシンボル	ビット名	機能	RW
PDL0	ポートPi0方向ビット	0 : 入力モード	RW
PDL1	ポートPi1方向ビット	(入力ポートとして機能)	RW
PDL2	ポートPi2方向ビット	1 : 出力モード	RW
PDL3	ポートPi3方向ビット	(出力ポートとして機能)	RW
PDL4	ポートPi4方向ビット	(i=0~3, 6~8, 10)	RW
PDL5	ポートPi5方向ビット		RW
PDL6	ポートPi6方向ビット		RW
PDL7	ポートPi7方向ビット		RW

注1. PACRレジスタの設定をしてください。
80ピン版の場合、PACR2, PACR1, PACR0を“0112”にしてください。
64ピン版の場合、PACR2, PACR1, PACR0を“0102”にしてください。

ポートPiレジスタ(i=0~3, 6~8, 10)(注1)

シンボル	アドレ	リセット後の値
P0~P3	03E016, 03E116, 03E416, 03E516番地	不定
P6~P8	03EC16, 03ED16, 03FD16番地	不定
P10	03F416番地	不定

ビットシンボル	ビット名	機能	RW
PI0	ポートPi0ビット	入力モードに設定した入出力ポート	RW
PI1	ポートPi1ビット	に対応するビットを讀むと、端子の	RW
PI2	ポートPi2ビット	レベルが読める。	RW
PI3	ポートPi3ビット	出力モードに設定した入出力ポート	RW
PI4	ポートPi4ビット	に対応するビットに書くと、端子の	RW
PI5	ポートPi5ビット	レベルを制御できる	RW
PI6	ポートPi6ビット	0 : “L” レベル	RW
PI7	ポートPi7ビット	1 : “H” レベル(注1)	RW
		(i=0~3, 6~8, 10)	

注1. PACRレジスタの設定をしてください。
80ピン版の場合、PACR2, PACR1, PACR0を“0112”にしてください。
64ピン版の場合、PACR2, PACR1, PACR0を“0102”にしてください。

ルネサス M16C/29グループ
ハードウェアマニュアルP336,337より抜粋

M16Cのマニュアルを確認して、ポートP3の内容を確認します。デップスイッチはポートP3のビット3～ビット0の4ビットを使用します。ポートP3レジスタの下4ビットを確認すればデップスイッチの状態を読み取れます。このとき、ポートP3方向レジスタはスイッチとして利用するので、入力“0”を指定します。

switch_dip : メイン関数

- ディップスイッチの状態を読み込み, led_coutと同様に1bitずつチェックして, 対応するLEDを点灯させる

ハードウェア
初期化

```
void
main(void){
    unsigned char dip_data, led_data;
    led_init();
    switch_dip_init();
    for (;;) {
        dip_data = switch_dip_sense();
        led_data = 0;
        if ((dip_data & DSW1) == DSW1) {
            led_data = led_data | LED1;
        }
        if ((dip_data & DSW2) == DSW2){
            led_data = led_data | LED2;
        }
        ...省略...
        led_out(led_data);
    }
}
```

ディップスイッチ
状態読みこみ

LED1設定

LED2設定

2008/11/28

TOPPERSプロジェクト認定

70



switch_dipのメイン関数を確認します。メイン関数の最初にディップスイッチの初期化関数 (switch_dip_init) を追加します。ディップスイッチの内容を保存するdip_dataを変数として定義します。ディップスイッチの内容を読み取る関数 (switch_dip_sense) をdip_dataに取り込み、この内容からLEDの表示を行うように改造します。

switch_dip : 作成

プログラム switch_dip を作成

- 手順
 - マクロの定義
 - 初期化関数 switch_dip_init() の作成
 - ポートを入力モードへ
 - デップスイッチ状態取得関数 switch_dip_sense() の作成
 - ポートレジスタの読み込み

2008/11/28

TOPPERSプロジェクト認定

71



プログラム作成のポイントについて説明します。

1) マクロの定義

ポートP3のレジスタと方向レジスタ

デップスイッチがそれぞれのビットかをあらわすマクロ

2) 初期化関数 switch_dip_init の作成

ポートP3ポートの下4ビットを入力に設定する。

3) デップスイッチ状態取得関数 switch_dip_sense の作成

マクロを使用してポートP3の読み取りを行い、戻り値として返す。

switch_dip : マクロ定義

- ポインタ型へのキャストまで含めてマクロ化する(方針)
- ポート3関連のレジスタのマクロ定義

```
#define BREG_SFR_P3      ((volatile unsigned char *)0x03E5)
#define BREG_SFR_PD3    ((volatile unsigned char *)0x03E7)
```

- ポート3のディップスイッチの接続ビット定義

```
#define DSW1            0x01
#define DSW2            0x02
#define DSW3            0x04
#define DSW4            0x08
#define DSW_MASK        (DSW1|DSW2|DSW3|DSW4)
```

ポートP3のレジスタのマクロ定義を行います。ここではポイント型を取り込んだ定義とします。ポートP3用のレジスタはバイト型なので以下のようになります。

```
#define BREG_SFR_P3      ((volatile unsigned char *)0x03E5)
#define BREG_SFR_PD3    ((volatile unsigned char *)0x03E7)
```

ディップスイッチのビット定義を行います。

```
#define DSW1            0x01
#define DSW2            0x02
#define DSW3            0x04
#define DSW4            0x08
#define DSW_MASK        (DSW1|DSW2|DSW3|DSW4)
```


switch_dip :初期化関数 switch_dip_init()

- ポート3のビット3,2,1,0を入力モードへ
 - ポート3方向レジスタのビット3,2,1,0を'0'に設定する

```
void  
switch_dip_init(void){  
    unsigned char reg;  
  
    reg = *BREG_SFR_PD3;  
    reg = reg & ~DSW_MASK;  
    *BREG_SFR_PD3 = reg;  
}
```

ビット3,2,1,0の
み'0'に設定

switch_dip_init関数を作成します。ポートP3のビット7～ビット4は変更せず、ビット3～ビット0を入力方向“0”に設定します。~DSW_MASKは0xF0(“1111 0000”)になりますのでANDして使用します。

switch_dip : デップスイッチ状態取得関数

- switch_dip_sense()
- ポートレジスタ3のビット3,2,1,0の値を取得

```
unsigned char
switch_dip_sense(void){
    unsigned char reg;

    reg = *BREG_SFR_P3;
    reg = reg & DSW_MASK;
    return reg;
}
```

ビット3,2,1,0のみ
有効に

デップスイッチの状態を読み取る関数、**switch_dip_sense**を作成します。ポートP3の内容を読み込み **DSW_MASK** (0x0f (“0000 1111”)) でANDすればデップスイッチの状態を取り出せます。

スイッチプログラム : switch_push 概要

- プッシュスイッチSW9を押すとLEDの表示をカウントアップし, SW8を押すとLEDの表示をカウントダウンする
- プッシュスイッチがOFFからONになったタイミングで押されたと判定する
- 学習内容
 - マニュアルからの情報の取得
 - 状態の変化の捉え方
 - ポーリングによる事象待ち
- 作成方法
 - led_countをベースに作成

2008/11/28

TOPPERSプロジェクト認定

75



プッシュスイッチ (SW9とSW8) の押下で、LEDをカウントアップ、カウントダウンするプログラムを作成します。プロジェクトを作成します。my_program¥switch_pushを作成し、my_program¥led_countの下に2つプログラム(start.a30、led_count.c)をswitch_pushの下にコピーします。led_count.cのファイル名をswitch_push.cに変更します。TMを起動して、switch_pushプロジェクトを作成します。

1) プロジェクトの作成にてプロジェクト名switch_push、ディレクトリmy_program¥switch_pushを指定します。

2) switch_push.cをファイルとして追加します。

switch_push : 作成

プログラム switch_push を作成

- 手順
 - マニュアルによるプッシュスイッチの接続の確認
 - マクロの定義
 - 初期化関数 switch_push_init() の作成
 - ポートを入力モードへ
 - プッシュスイッチ状態取得関数 switch_push_sense() の作成
 - ポートの読み込み
 - メイン関数
 - スイッチの変化を捉える (OFFからONへ)

2008/11/28

TOPPERSプロジェクト認定

76

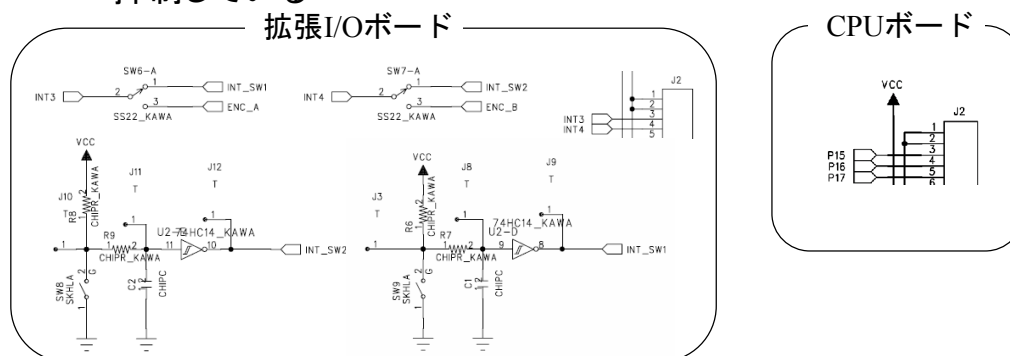


プログラムの修正手順について説明を行います。

- 1) 回路図とマニュアルでプッシュスイッチのポートの確認を行う。
- 2) プッシュスイッチで使用するマクロ定義を行います。
- 3) 初期化関数 `switch_push_init()` を作成します。
- 4) プッシュスイッチの状態を取り込む関数 `switch_push_sense()` を作成します。
- 5) メイン関数に `switch_push_init` と `switch_push_sense` を追加します。プッシュスイッチの内容でカウント方向を変えるように、プログラムを修正します。
- 6) ビルドして確認を行います。

switch push : プッシュスイッチの接続

- マイコンのプログラマブル入出力ポートにインバーターを介して接続されている
- ONでGND, OFFでVccに接続する
 - マイコンからはONで'1'が, OFFで'0'が読み込める
- チャタリング除去回路となっており, チャタリングの発生を抑制している



2008/11/28

TOPPERSプロジェクト認定

77



プッシュスイッチの回路は、**title STUDY IO**の回路図の中央下に記載されています。SW9とSW8はポートP1の5ビット目と6ビット目の接続しています。プッシュスイッチはデブススイッチの回路に加えて、抵抗やコンデンサやインバータが追加されています。これはスイッチを押したとき接続状態で細かいオンオフを繰り返す信号(チャタリングと呼ぶ)をキャンセルする回路です。インバータが入っていますが、オンオフのVccとグランドがデブススイッチとは逆になっていますので実質的にはデブススイッチと同様にオンで“1”、オフで“0”が入力されます。

SW9:INT SW1→INT3→J2コネクタの3ピン→P15 (ポートP1の5ビット目)

SW8:INT SW2→INT4→J2コネクタの4ピン→P16 (ポートP1の6ビット目)

switch_push : プッシュスイッチの接続ポート

- ポート1に接続
 - SW9 : ビット5 : P15
 - SW8 : ビット6 : P16
- ポート1の構成レジスタ
 - ポート方向レジスタ : 0x03E3
 - ポートレジスタ : 0x03E1
- 初期化
 - ポート1のビット6,5を入力モードへ
- スイッチ状態の取得
 - ポート1レジスタを読み込む

ポートPi方向レジスタ(i=0~3, 6~8, 10)(注1)

シンボル	アドレス	リセット後の値
P00~P03	03E216, 03E316, 03E616, 03E716番地	0016
P06~P08	03E116, 03EF16, 03F216番地	0016
PD10	03F616番地	0016

ビットシンボル

ビットシンボル	ビット名	機能	RW
PDL0	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)	RW
PDL1	ポートPi1方向ビット	1 : 出力モード (出力ポートとして機能)	RW
PDL2	ポートPi2方向ビット		RW
PDL3	ポートPi3方向ビット		RW
PDL4	ポートPi4方向ビット	(i=0~3, 6~8, 10)	RW
PDL5	ポートPi5方向ビット		RW
PDL6	ポートPi6方向ビット		RW
PDL7	ポートPi7方向ビット		RW

注1. PACRレジスタの設定をしてください。
80ピン版の場合、PACR2, PACR1, PACR0を"0112"にしてください。
64ピン版の場合、PACR2, PACR1, PACR0を"0102"にしてください。

ポートPiレジスタ(i=0~3, 6~8, 10)(注1)

シンボル	アドレス	リセット後の値
P0~P3	03E016, 03E116, 03E416, 03E516番地	不定
P6~P8	03EC16, 03ED16, 03FD16番地	不定
P10	03F416番地	不定

ビットシンボル

ビットシンボル	ビット名	機能	RW
PI0	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。	RW
PI1	ポートPi1ビット	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。端子の レベルを制御できる。	RW
PI2	ポートPi2ビット	0 : "L" レベル	RW
PI3	ポートPi3ビット	1 : "H" レベル(注1)	RW
PI4	ポートPi4ビット		RW
PI5	ポートPi5ビット		RW
PI6	ポートPi6ビット		RW
PI7	ポートPi7ビット		RW

注1. PACRレジスタの設定をしてください。
80ピン版の場合、PACR2, PACR1, PACR0を"0112"にしてください。
64ピン版の場合、PACR2, PACR1, PACR0を"0102"にしてください。

ルネサス M16C/29グループ
ハードウェアマニュアルP336.337より抜粋

2008/11/28

TOPPERSプロジェクト認定

78



マニュアルを参照して、ポートP1について調べます。SW9とSW8はポートP1のビット5とビット6に接続しています。ポートP1関連のレジスタの情報の確認を行います。

ポートP1方向レジスタ: アドレス0x03E3、ビット5と6は入力
ポートレジスタ : アドレス0x03E1、ビット5と6のみ参照

switch_push : マクロ定義

- ポインタ型へのキャストまで含めてマクロ化する(方針)
- ポート1関連のレジスタのマクロ定義

```
#define BREG_SFR_P1    ((volatile unsigned char *)0x03E1)
#define BREG_SFR_PD1  ((volatile unsigned char *)0x03E3)
```

- ポート1のデッドスイッチの接続ビット定義

```
#define PSW9          0x20
#define PSW8          0x40
#define PSW_MASK      (PSW9|PSW8)
```

ポートP1のレジスタのマクロ定義を行います。ここではポイント型を取り込んだ定義とします。ポートP1用のレジスタはバイト型なので、以下のようになります。

```
#define BREG_SFR_P1    ((volatile unsigned char *)0x03E1)
#define BREG_SFR_PD1  ((volatile unsigned char *)0x03E5)
```

デッドスイッチのビット定義を行います。

```
#define PSW9          0x20
#define PSW8          0x40
#define PSW_MASK      (PSW9|PSW8)
```

switch_push : 初期化関数

- ポート1のビット6,5を入力モードへ
 - ポート1方向レジスタのビット6,5を'0'に設定する

```
void
switch_push_init(void){
    unsigned char reg;

    reg = *BREG_SFR_PD1;
    reg = reg & ~PSW_MASK;
    *BREG_SFR_PD1 = reg;
}
```

ビット6,5のみ'0'
に設定

プッシュスイッチの初期化関数switch_push_initを作成します。処理内容は以下の通りです。

- 1) 現在のポート1方向レジスタの値を取り込む

```
reg = *BREG_SFR_PD1;
```

- 2) 今回使用する、ビット6,5のみ0、他ビットは1を立てたマスクとアンドを取る

```
reg = reg & ~PSW_MASK;
```

- 3) 作成した値を、ポート1方向レジスタへ格納

```
*BREG_SFR_PD1 = reg;
```


switch_push : プッシュスイッチ状態取得関数

- switch_push_sense()
- ポートレジスタ1のビット6,5の値を取得

```
unsigned char
switch_push_sense(void){
    unsigned char reg;

    reg = *BREG_SFR_P1;
    reg = reg & PSW_MASK;
    return reg;
}
```

ビット6,5のみ
有効に

プッシュスイッチの状態取得関数switch_push_sense関数を作成します。処理は以下の通りです。

- 1) 現在のプッシュスイッチの状態を読み取る

```
reg = *BREG_SFR_P1;
```

- 2) 今回使用する、ビット6,5のみ1のマスクとアンドを取る

```
reg = reg & PSW_MASK;
```

- 3) その値を戻り値で返す。

```
return reg;
```

switch_push : main()関数

- カウント用変数led_countの用意
- ハードウェアの初期化
- 無限ループ内でプッシュスイッチの状態をチェックしてSW9がOFFからONに変化した場合は, led_countをインクリメントし, SW8がOFFからONに変化した場合は, led_countをデクリメントする
- 状態の変化の捉え方
 - プッシュスイッチの以前の状態を保存して比較
 - ローカル変数を用意して前回の状態取得時の状態を保存する
- スwitchの状態(ON,OFF)はマクロ化する

```
#define SW_ON    1
#define SW_OFF   0
```

2008/11/28

TOPPERSプロジェクト認定

82



メイン関数を書換えます。SW9とSW8の状態と管理する定義と変数を作成します。

スイッチの状態:

#define SW_ON	1	オンの状態
#define SW_OFF	0	オフの状態

変数:

```
unsigned char sw9_state = SW_OFF;
unsigned char sw8_state = SW_OFF;
```

switch_push : main()関数

• SW9のみ抜粋

```
void main(void){
    unsigned char led_data;      SW9状態保存用変数
    unsigned char sw_data;
    unsigned char sw9_state = SW_OFF;
    led_init();
    switch_push_init();          プッシュスイッチ状態取得
    for (;;) {
        sw_data = switch_push_sense();
        if(((sw_data & PSW9) == PSW9) && (sw9_state == SW_OFF)){
            led_data++;          SW9がOFFからONになったか
            sw9_state = ((sw_data & PSW9) == PSW9)? SW_ON : SW_OFF;
            led_out_bdigit(led_data);
        }
    }
}
```

2008/11/28

TOPPERSプロジェクト認定

83

TOPPERS

LEDの表示前にプッシュスイッチの状態をsw9_stateとsw8_stateに更新し、その状態でLEDをカウントアップするか、カウントダウンするかの判定を行ったあとLEDの表示を行います。

無限ループで行うこと:

- 1) 現在のプッシュスイッチの状態を読み込む→sw_dataへ
- 2) 各SWがOFFからONに変わったかどうかの判定
- 3) 現在ONかOFFかを記憶→sw9_state sw8_state

SW9のみ説明すると、SW9がOFFからONになったかどうかの判定の行

```
if(((sw_data == PSW9) == PSW9) && (sw9_state == SW_OFF)){
```

左辺①

右辺②

左辺①SW9が現在ONかどうか→sw_dataで判定

右辺②前回OFFかどうか→sw9_stateで判定

ポーリングプログラム

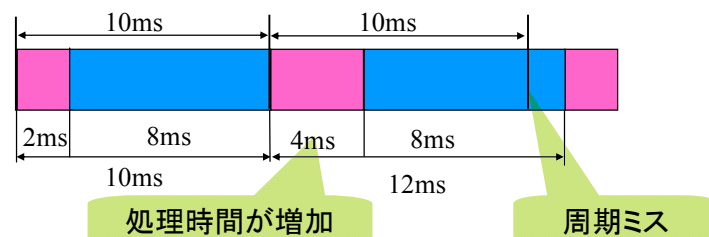
1. LEDプログラム
 - ・プロジェクトの作成方法
2. スイッチプログラム
3. 宿題:タイマプログラム
4. 宿題:シリアルI/Oプログラム

タイマプログラム : timer 概要

- ハードウェアタイマにより正確に1秒間隔でLEDの点灯をカウントアップする
- 学習内容
 - ハードウェアタイマの使い方
 - カウントソース, カウント値の決定方法
- 作成方法
 - led_count をベースに作成

timer : ハードウェアタイマによる時間生成

- led_countでは, 時間生成にbusy_wait()関数を使用
 - for()ループにより一定時間を生成
- ループでは正確な時間が生成できない
 - そもそもループでは正確な時間は作れない
 - 正確な周期処理を実現が困難
 - 周期 = 周期処理の実行時間 + ループの実行時間
 - 周期処理の実行時間が変わると, それに合わせてループの長さを変更する必要がある



timer : ハードウェアタイマの概要

- M16Cのハードウェアタイマ
 - 16ビットタイマを8個
 - タイマモード, カウント方法(ダウン/アップカウント)やクロックソースはモードレジスタで設定可能
- タイマA0を使用
 - タイマモードを使用
 - 内部カウントソースによりカウント
 - ダウンカウント
 - アンダーフロー時にオートリロード
 - メインクロック(f1)は20MHz
 - 周辺クロック選択レジスタで変更可能

2008/11/28

TOPPERSプロジェクト認定

87



timerのハードウェアマニュアルを参照します。目次から12. タイマ→12. 1タイマAを開きます。
タイマAには4種類のモードがあります。

•タイマモード

内部カウントソースをカウントするモード

•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ーフロー、または他のタイマのアンダーフローをカバーするモード

•ワンショットタイマモード

カウント値が"0000 (16)"なるまでの間、1度だけパルスを出力するモード

•パルス幅変調モード

任意の幅のパルスを連続して出力するモード

timer : レジスタ

- タイムアウトは割り込み制御レジスタのIRビットにより通知
 - タイマ側のレジスタのビットがセットされる場合が多い

名前	サイズ	アドレス	説明
TA0MR	B	0x396	タイマA0モードレジスタ (モード, カウントソースを指定)
TA0	H	0x386	タイマA0レジスタ(カウント値を設定)
TABSR	B	0x380	カウント開始フラグ(タイマの開始と停止を制御)
TA0IC	B	0x055	タイマA0割り込み制御レジスタ(タイムアウトを通知)

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
C01WKIC	0041h 番地	XXXXXXXX00
C0REIC	0042h 番地	XXXXXXXX00
C0TDMIC	0043h 番地	XXXXXXXX00
IC0C0IC	0045h 番地	XXXXXXXX00
IC0C1IC, IC1CIC(注3)	0046h 番地	XXXXXXXX00
BTIC, SCLDAIC(注3)	0047h 番地	XXXXXXXX00
BCNIC	0048h 番地	XXXXXXXX00
DM0IC, DM1IC	0049h, 004Ch 番地	XXXXXXXX00
C0TERRIC	004Dh 番地	XXXXXXXX00
ADIC, KUPIC(注3)	004Eh 番地	XXXXXXXX00
S0TIC ~ S2TIC	0051h, 0053h, 004Fh 番地	XXXXXXXX00
S0RIC ~ S2RIC	0052h, 0054h, 005Dh 番地	XXXXXXXX00
TA0IC ~ TA4IC	0055h ~ 0059h 番地	XXXXXXXX00
TB0IC ~ TB2IC	005Ah ~ 005Ch 番地	XXXXXXXX00

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	0 0 0 : レベル0 (割り込み禁止)	RW
		0 0 1 : レベル1	
ILVL1		0 1 0 : レベル2	RW
		0 1 1 : レベル3	
ILVL2		1 0 0 : レベル4	RW
		1 0 1 : レベル5	
		1 1 0 : レベル6	RW
		1 1 1 : レベル7	
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 (DR1)
(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”のみ書き込み可能(“1”を書かないでください)。
 注2. 割り込み制御レジスタの定義は、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、注意事項の「割り込みの注意事項」を参照してください。
 注3. IFSR2Aレジスタで選択してください。

ルネサス M16C/29グループ
ハードウェアマニュアルP89より抜粋

2008/11/28

TOPPERSプロジェクト認定

88



タイマAiのレジスタの確認を行います。

1) タイマAiモードレジスタ

タイマモードを設定

2) タイマAiレジスタ

2バイトでアクセス

ここで設定した値で、カウントソースをn+1分周する

3) カウント開始フラグ

使用するタイマのカウントを開始するかどうか

3) 割り込み制御レジスタ(これは9. 割り込みを参照)

タイムアウトをここで通知される

分周とは、

周波数fの入力信号を入れて、それに周期したf/nの出力信号を得ること。

→周波数fの入力信号を入れて、係数がnになる毎、パルスを出せばf/n周波数のパルスが出る。

timer : タイマ操作の流れ

- タイマA0モードを設定
 - タイマモード, カウントソースを選択
- タイマA0レジスタにカウント値を設定
 - カウントソースと計時したい時間から求める
- カウント開始フラグを設定
- タイマA0割込み制御レジスタをチェックして, タイムアウトの通知を待つ

timer : 作成

プログラム timer を作成

- 手順
 - カウントソースの決定
 - f1, f8, f32から選択
 - カウント値の決定
 - 1秒は直接計時不可能なため 100msecを10回
 - プログラミング
 - マクロの定義
 - 初期化関数
 - 計時関数
 - メイン関数

timer : カウントソースの選択

カウントソースを高速にすると精度は高くなるが測定可能な最大時間が小さくなり、カウントソースを低速にするとその逆となる

- カウント値とカウントソースの決定
 - カウント値を保持するタイマA0レジスタは16bit
 - 0から65535までを設定可能
 - カウントソース
 - f1, f8, f32から設定可能
 - メインクロック(f1)が20MHzなので,
 - f8 : $20\text{MHz}/8 = 2.5\text{MHz}$
 - f32 : $20\text{MHz}/32 = 0.625\text{MHz}$
 - 各ソースで測定可能な最大値
 - f1 : $1 / 20\text{MHz} * 65535 = 3236.75 * 10^{-6} = 3.2\text{msec}$
 - f8 : $1 / 2.5\text{MHz} * 65535 = 26214 * 10^{-6} = 26\text{msec}$
 - f32 : $1 / 0.625\text{MHz} * 65535 = 104856 * 10^{-6} = 104\text{msec}$

カウントソースとはカウントアップするクロックを指します。タイマAiはカウントソースの選択ができます。分周が多くなる(f32になるにつれて)間隔が長くなります。

timer : カウント値の決定

- f32でも1秒のカウントは不可能
 - 100msecを10回計測して1秒を生成する
 - カウント値を計算
 - $(100\text{msec} / (1/0.625\text{MHz})) - 1 = 62499$
- その他のレジスタの設定値
 - TA0BSRの0ビット目をセットするとカウントをスタート
 - タイムアウトするとTA0ICのビット3(0x08)が‘1’にセットされる
 - TA0ICのビット3から‘1’を読み込んだ後は‘0’にクリアする

*初期化時に設定

名前	サイズ	アドレス	設定値	説明
TA0MR	B	0x396	0x80	タイマモード,クロックソースはf32*
TA0	H	0x386	62499	100msec
TA0BSR	B	0x380	0x01	タイマスタート(0x00で停止)
TA0IC	B	0x055	0x00	タイムアウト通知・クリア

2008/11/28

TOPPERSプロジェクト認定

92



f32を(1/0.625MHz)しても、カウントアップは100msecの中に、62500個です。これは16ビットに収まりません。したがって、100msのカウントを10回計測して1秒を作ります。

TA0MR (タイマAiレジスタ):

タイマA0のモード、カウントリソース等を設定するレジスタ

TA0

タイマA0のカウント、現在のカウンタ値が読み書きできる

TA0BSR (カウント開始フラグ)

タイマA0のスタート、停止を設定するレジスタ

TA0IC (割り込み制御レジスタ):

タイマA0の終了条件や割り込み条件を設定するレジスタ

タイマレジスタに設定した値+1 カウントすると割り込み制御レジスタIRビットに1を立てる。

カウントする間隔をモードレジスタで設定。

timer : マクロの定義

- タイマ関連のレジスタ
 - アクセスサイズに注意

```
/* タイマA0モードレジスタ */
#define BREG_SFR_TA0MR ((volatile unsigned char *)0x0396)
/* タイマA0レジスタ */
#define HREG_SFR_TA0 ((volatile unsigned short *)0x0386)
/* カウント開始フラグ */
#define BREG_SFR_TABSR ((volatile unsigned char *)0x0380)
```

- タイマ関連の定義

```
#define TA0MR_F32_TMOD 0x80 /* f32を選択,タイマモード */
#define TIMER_COUNT 62499 /* f32で100msec */
#define TABSR_TA0S 0x01 /* タイマ0スタート */
/* タイマ0割込みレジスタ */
#define BREG_SFR_TA0IC ((volatile unsigned char *)0x0055)
#define TA0IC_IR 0x08 /* 割込み要求あり */
```

2008/11/28

TOPPERSプロジェクト認定

93



タイマ関連のマクロ定義を行います。

タイマA0のモードレジスタへの設定値:

```
#define TA0MR_F32_TMOD 0x80
```

タイマA0レジスタへの設定値:

```
#define TIMER_COUNT 624
```

タイマA0のカウントリソースf32の設定値

タイマA0カウント開始フラグへの設定値:

```
#define TABSR_TA0S 0x01
```

タイマA0のカウント開始フラグ

タイマA0割込みレジスタへの設定値:

```
#define TA0IC_IR 0x08
```

タイマA0のカウント終了確認フラグ

timer : 初期化関数

- タイマーのモードを設定する
 - 停止させる必要があるかはタイマによって異なる
 - M16Cはマニュアルに特に記載がない

```
void
timer_a0_init(void){
    *BREG_SFR_TABSR = *BREG_SFR_TABSR & ~TABSR_TA0S;

    *BREG_SFR_TA0MR = TA0MR_F32_TMOD;
}
```

タイマ停止

モード設定

タイマの初期化関数の作成を行います。処理は以下の通りです。

1) カウント開始フラグに タイマA0を停止を設定

```
*BREG_SFR_TABSR = *BREG_SFR_TABSR & ~TABSR_TA0S;
```

2) タイマモードレジスタに初期値を設定

```
*BREG_SFR_TM0MR = TA0MR_F32_TMOD;
```

timer : 計時関数

- 引数で指定された回数タイマで計時する
 - オートリロードのため、スタートさせると後は自動的に計時を繰り返す

```
void
timer_a0_wait(int cnt){
    int lp_cnt;
    *BREG_SFR_TABSR = *BREG_SFR_TABSR & ~TABSR_TA0S;
    *HREG_SFR_TA0 = TIMER_COUNT;
    *BREG_SFR_TA0IC = 0x00;

    *BREG_SFR_TABSR = *BREG_SFR_TABSR | TABSR_TA0S;
    for(lp_cnt = 0; lp_cnt < cnt; lp_cnt++){
        while((*BREG_SFR_TA0IC & TA0IC_IR) != TA0IC_IR);
        *BREG_SFR_TA0IC = 0x00;
    }
    *BREG_SFR_TABSR = *BREG_SFR_TABSR & ~TABSR_TA0S;
}
```

タイマ停止 (最初の *BREG_SFR_TABSR = *BREG_SFR_TABSR & ~TABSR_TA0S;)

カウント値設定 (*HREG_SFR_TA0 = TIMER_COUNT;)

タイムアウトフラグクリア (*BREG_SFR_TA0IC = 0x00;)

タイマスタート (*BREG_SFR_TABSR = *BREG_SFR_TABSR | TABSR_TA0S;)

タイムアウト待ち (while((*BREG_SFR_TA0IC & TA0IC_IR) != TA0IC_IR);)

フラグクリア (*BREG_SFR_TA0IC = 0x00;)

タイマ停止 (*BREG_SFR_TABSR = *BREG_SFR_TABSR & ~TABSR_TA0S;)

2008/11/28

TOPPERSプロジェクト認定

95



計時関数を作成します。引数で設定された回数分、タイマ時間(100ms)待ちします。以下に処理を記載します。

- 1) タイマを停止します。(初期化と同じ)

```
*BREG_SFR_TABSR = *BREG_SFR_TABSR & ~TABSR_TA0S;
```

- 2) タイマカウンタに100ms分のカウンタ値をセット

```
*HREG_SFR_TA0 = TIMER_COUNT;
```

- 3) 終了フラグをゼロにリセットします

```
*BREG_SFR_TA0IC = 0x00;
```

- 4) タイマをスタートします。

```
*BREG_SFR_TABSR = *BREG_SFR_TABSR | TABSR_TA0S;
```

- 5) 指定したカウント分、ループする。タイムアウトフラグが立つまで待ち、立ったらクリアする

```
while((*BREG_SFR_TA0IC & TA0IC_IR) != TA0IC_IR);
```

```
*BREG_SFR_TA0IC = 0x00;
```

- 6) ループを抜けたら、タイマを停止:1)と同じ

timer : メイン関数

- led_countのメイン関数を変更
 - busy_wait() を timer_a0_wait()に変更

```
void
main(void){
    int count = 0;
    led_init();
    timer_a0_init();
    for (;;) {
        led_out_bdigit(count++);
        timer_a0_wait(10);
    }
}
```

タイマの初期化

LEDの出力

LEDの初期化

タイマにより待つ

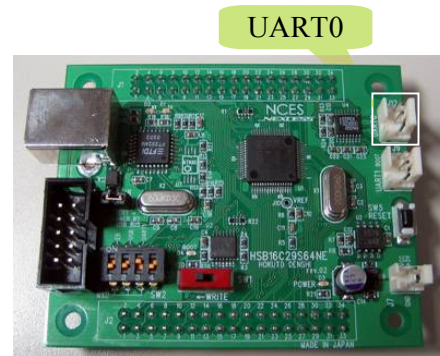
最後にメイン関数を作成します。これはled_countを修正します。タイマの初期化関数を追加して、ソフトウェア待ち関数busy_wait()をtimer_a0_wait(10)に置き換えれば終了です。

ポーリングプログラム

1. LEDプログラム
 - ・プロジェクトの作成方法
2. スイッチプログラム
3. 宿題:タイマプログラム
4. 宿題:シリアルI/Oプログラム

シリアルI/Oプログラム：uart 概要

- シリアルI/Oからデータを受信すると, LEDの点灯をカウントアップし, 受け取ったデータをシリアルI/Oに送信する (ポーリング)
- 学習内容
 - シリアルI/Oの使い方
- 作成方法
 - led_countをベースに作成
- ボードの設定
 - UART0とPCのシリアルポートを接続する

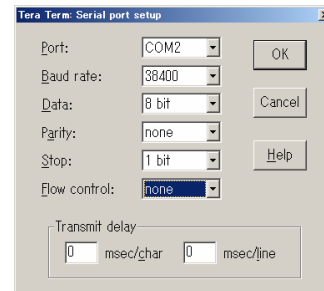
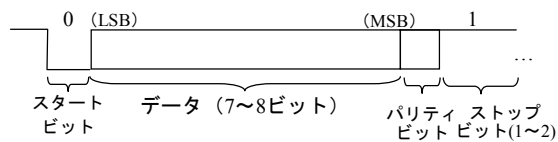


uart : シリアルI/Oのフォーマット

- 通信する双方で同じ設定にする必要がある
 - ボーレート(BPS) : 9600, 14400, 38400, 57600, 115200
 - データサイズ : 7bit, 8bit
 - パリティ : なし, even, odd
 - ストップビット : 1bit, 2bit
 - フローコントロール : なし, Xon/Xoff, hardware

• ターミナルソフトの設定例

• データフォーマット



uart : 送受信フォーマットとレジスタ

- フォーマット
 - ボーレート(BPS) : 38400, データサイズ : 8bit, パリティ: なし, ストップビット: 1bit, フローコントロール : なし
- 使用チャネル
 - UART0を使用(UART1はデバッグ用)
 - UARTモードで使用, カウントソースはf_i (20Mhz)

名前	サイズ	アドレス	説明
U0TB	B	0x03A2	送信バッファレジスタ
U0RB	B	0x03A6	受信バッファレジスタ
U0BRG	B	0x03A1	転送速度レジスタ
U0MR	B	0x03A0	送受信モードレジスタ
U0CO	B	0x03A4	送受信制御レジスタ0
U0CON	B	0x03B0	送受信制御レジスタ2
U0C1	B	0x03A5	送受信制御レジスタ1

uart : 作成

プログラム uart を作成

- 手順
 - 転送速度レジスタの設定値の計算
 - ボーレートが38400bpsになるよう設定
 - その他のレジスタの設定値の決定
 - プログラミング
 - マクロの定義
 - 初期化関数
 - 受信関数
 - 送信関数
 - メイン関数

uart : 転送速度レジスタの設定値の計算

- 転送速度は 38400bps
- カウントソースはf_i (20Mhz)を使用
- 転送速度レジスタの設定値の計算式(UARTモード)
 - $f_j / (16(n + 1))$
 - $20\text{MHz} / (16(n + 1)) = 38400$
- $n = 32$ にすると
 - $20\text{Mhz} / (16(32+1)) = 37878$
 - 誤差が数%以内なら動作

uart : その他のレジスタの設定値

フォーマットに従って設定値を決定する

- 送受信モードレジスタ : 0x05
 - UARTモード転送データ長8ビット, 内部クロック,
 - 1ストップビット, パリティ禁止
- 送受信制御レジスタ0 : 0x10
 - カウントソース:f1, CTS/RTS機能禁止
 - CMOS出力, 転送クロックの立下りでデータ出力, 立ち上がりで受信データ入力
 - LSBファースト
- 送受信制御レジスタ1 : 0x05
 - 送信許可
 - 受信許可

uart : マクロ定義1 : UART0関連のレジスタ

```
/* 送信バッファレジスタ */
#define BREG_SFR_U0TB ((volatile unsigned char *)0x03A2)
/* 受信バッファレジスタ */
#define BREG_SFR_U0RB ((volatile unsigned char *)0x03A6)
/* 転送速度レジスタ */
#define BREG_SFR_U0BRG ((volatile unsigned char *)0x03A1)
/* 送受信モードレジスタ */
#define BREG_SFR_U0MR ((volatile unsigned char *)0x03A0)
/* 送受信制御レジスタ0 */
#define BREG_SFR_U0CO ((volatile unsigned char *)0x03A4)
/* 送受信制御レジスタ2 */
#define BREG_SFR_UCON ((volatile unsigned char *)0x03B0)
/* 送受信制御レジスタ1 */
#define BREG_SFR_U0C1 ((volatile unsigned char *)0x03A5)
```


uart : マクロ定義2 : レジスタの設定値

- 定めた設定値でマクロを定義

```
#define U0MR_VAL 0x05 /* 送受信モードレジスタの設定値 */  
#define U0CO_VAL 0x10 /* 送受信制御レジスタ0の設定値 */  
#define U0BRG_VAL 32 /* 転送速度レジスタの設定値 */
```

- 送受信制御レジスタ1のビット定義

```
#define U0C1_TE 0x01 /* 送信許可 */  
#define U0C1_RE 0x04 /* 受信許可 */  
#define U0C1_TI 0x02 /* 送信バッファ空 */  
#define U0C1_RI 0x08 /* 受信バッファあり */
```

uart : 初期化関数 uart0_init

- 各制御レジスタを初期化し, 送受信を許可する

```
void
uart0_init(void){
    /* 送受信制御レジスタ2初期化 */
    *BREG_SFR_UCON = 0x00;
    /* 送受信モードレジスタの設定 */
    *BREG_SFR_U0MR = U0MR_VAL;
    /* 送受信制御レジスタ0の設定 */
    *BREG_SFR_U0CO = U0CO_VAL;
    /* 転送速度レジスタの設定 */
    *BREG_SFR_U0BRG = U0BRG_VAL;
    /* 送受信制御レジスタの設定（送受信許可） */
    *BREG_SFR_U0C1 = (U0C1_TE | U0C1_RE);
}
```

uart : 受信関数

- データを受信すると, 受信制御レジスタのRIビット(受信完了フラグ)がセットされる
- ポーリングでセットされるのを待つ

```
unsigned char
uart0_pol_getc(void){
    /* 受信バッファにデータが入るのを待つ */
    while((*BREG_SFR_U0C1 & U0C1_RI) != U0C1_RI);

    /* データ受信 */
    return *BREG_SFR_U0RB;
}
```

uart : 送信関数

- 送信データを書き込める状態であれば, 受信制御レジスタのTI(送信バッファ空フラグ)がセットされる
- ポーリングでセットされるのを待つ

```
void
uart0_pol_putc(unsigned char c){
    /* 送信バッファが空くのを待つ */
    while((*BREG_SFR_U0C1 & U0C1_TI) != U0C1_TI);

    /* データ送信 */
    *BREG_SFR_U0TB = c;
}
```

uart : メイン関数

- 受信関数でデータを受信後, LEDをカウントアップし, 送信関数で受信したデータを送信する

```
void
main(void){
    unsigned char count = 0;
    unsigned char c;

    led_init();
    uart0_init();

    for (;;) {
        c = uart0_pol_getc();
        led_out_bdigit(count++);
        uart0_pol_putc(c);
    }
}
```

データ受信

LEDカウントアップ

データ送信

割込みプログラミング

1. [M16Cの割込みアーキテクチャ](#)
2. スイッチ割込みプログラム
3. 宿題:タイマ割込みプログラム
4. 宿題:シリアルI/O割込みプログラム

目的

割込みを用いたプログラミングを学ぶ

- 割込みによる事象待ちのために必要な知識
 - 割込みハンドラ
 - 割込みベクタテーブル
 - 割込み関連のレジスタ
 - 割込み禁止・許可
 - 割込み優先度
- プログラム対象の周辺デバイス
 - プッシュスイッチ
 - タイマ
 - シリアルI/O

2008/11/28

TOPPERSプロジェクト認定

111



ポーリングプログラムではメインルーチンがハードウェアの状態を監視しハードウェアの状態が変化した場合、それに応じたプログラムを実行することにより、必要な処理を行いました。しかし、ポーリングプログラムでは常にハードウェアの状態を監視しなければならない。短時間にハードウェアの状態が変化する場合、メインプログラムの処理を阻害したり、取りこぼしが発生したり、全体の動作を阻害する場合があります。

割込み(interrupt)は、ハードウェアの状態が変化した場合、割込みサービスルーチンという特別なプログラムをメインルーチンを中断して実行する機能です。従って、短時間にハードウェアの状態が変化しても割込みサービスルーチンを使って確実に必要な処理を行うことができます。

割込みを用いたプログラミングを行うためには、割込みルーチンを起動するためのいくつかの手続きが必要です。

1) 割込みハンドラ

割込みを制御するためのプログラム、割込みサービスルーチンと同様な意味です

2) 割込みベクターテーブル

割込み発生時、プロセッサが割込みハンドラを認識するための実行アドレスを割込みベクターと呼びます。割込みを要求するハードウェアごとに割込みベクターを並べたテーブルを割込みベクターテーブルと呼びます。

3) 割込み関連のレジスタ

割込みの禁止・許可、割込み発生状態、割込みの優先度等を設定するハードウェアレジスタ

4) 割込み禁止・許可

割込みを禁止・許可には、いろいろな設定があります。割込みの中には禁止のできない割込み(ノンマスクابل割込み)があります。また、禁止の方法はプロセッサ側で優先度に従って禁止・許可を行う方法と、デバイス側で割込みの禁止・許可を行う方法があり、両方が許可されないと割込み処理は実行されません。

5) 割込み優先度

割込みには優先度が設定でき、割込みサービスルーチンが実行中でも、さらに優先度が高い割込み要求があると、その割込みを実行することができます。これを多重割込みといいます。

割込みプログラミング：プログラムファイル

- プログラムファイルの置き場所
 - 教材ディレクトリ¥program
- プログラム一覧
 - int_switch_1 : PUSHスイッチ割込みプログラム1
(単一割込み)
 - int_switch_2 : PUSHスイッチ割込みプログラム2
(割込み禁止/割込み応答時間)
 - int_switch_3 : PUSHスイッチ割込みプログラム3
(複数割込み/優先度の設定)
 - int_timer : タイマ割込みプログラム
 - int_uart : シリアルI/O割込みプログラム

2008/11/28

TOPPERSプロジェクト認定

112



割込みプログラムは、教材ディレクトリのprogram以下にあります。本セミナーではint_switch_1からint_switch_3までのプログラムを実習で実行させます。int_timerとint_uartは宿題となっています。

1) int_switch_1ではSW9の単一割込みを学習します。LEDを2進数カウンタとして、スイッチを押すごとにLEDをカウントアップします。

2) int_switch_2ではSW9の単一割込みの割込み禁止と割込み応答時間について学習します。LEDを2進数の自動インクリメントカウンタとして起動させ、スイッチを押すごとにカウンタリセットを行います。

3) int_switch_3ではSW9とSW8の複数割込みについて学習します。LEDを2進数カウンタとして、SW9を押すごとにLEDをカウントアップします。SW8を押すごとにLEDをカウントダウンしてください。

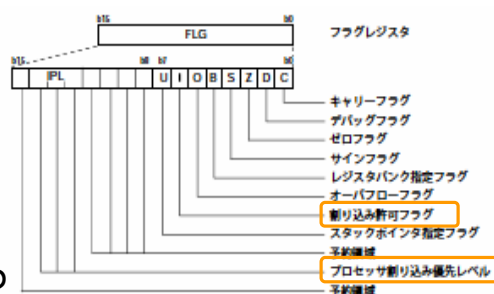
4) int_timerでは、タイマを割込みで起動させ、1秒ごとにLEDカウンタをインクリメントするプログラムを割り込みを用いて作成します。通常タイマは割込みで制御します。

5) int_uartでは、ポーリングプログラムで作成したLEDの表示カウントをシリアル通信するプログラムを割り込みを用いて作成します。通常UARTは割込みで制御します。

M16Cの割込みアーキテクチャ

フラグレジスタでの割込みの禁止・許可や優先度を制御

- 割込み許可フラグ
 - Iフラグ
 - '1'で許可
- プロセッサ割込み優先レベル
 - IPL
 - 設定値より大きい値を持つ割込みのみ受け付ける



ルネサス M16C/29グループ
ハードウェアマニュアルP36より抜粋

M16Cのプロセッサの割込み制御について説明を行います。M16Cの割込み制御はFLAGレジスタを用いて行います。割込みを制御するフラグは以下の2つです。

1) Iフラグ

割込みの禁止・許可フラグ、ノンマスカブル割込み以外の割込みの禁止・許可の設定を行います。0の設定で禁止、1の設定で許可となります。

2) IPLフラグ

プロセッサ割込み優先レベルを指定します。IPLで設定した優先レベルより低い優先度の割込みは禁止されます。7で全てのマスカブル割込みが禁止されます。設定より高い優先度の割込みでも、Iフラグが0の場合は割込みは発生しません。

M16Cの割込みアーキテクチャ：可変ベクタテーブル

周辺機能の割り込み用割込みハンドラのアドレスを登録

- ベクタ番地は周辺機能毎に定まっている
 - ←M16C/29グループマニュアル：P87
- 先頭番地をINTBレジスタに設定する
 - スタートアップで設定
- マスカブル割込み
 - 抑止できる割込み
 - 割込みはそれぞれレベルをもち、プロセッサの割込み優先レベル以下のレベルの割込みは発生しても受け付けられない
 - 割込み優先レベルがないプロセッサもあります(禁止or許可のみ)
- ノンマスカブル割込み(固定ベクタテーブル)
 - 発生したら必ず受け付ける割込み

2008/11/28

TOPPERSプロジェクト認定

114



M16Cの割込みアーキテクチャについて説明を行います。割込みを実行するには割込みハンドラを指定する割込みベクタを設定する必要があります。M16Cは可変ベクタテーブルと固定ベクタテーブルの2つのベクタテーブル構成となっています。固定ベクタテーブルはハードウェアに関連したノンマスカブル割込み用のベクタを設定するベクタテーブルで、0xFFFFC番地から0xFFFFF番地に固定で設定されています。可変ベクタテーブルは先頭番地をINTBレジスタによって設定できるベクタテーブルでソフトウェア要因のノンマスカブル割込みや外部のハードウェア割込みの割込みハンドラを指定するベクタテーブルです。INTBレジスタの設定はアセンブラにより記述されます。

`ldintb #VECTOR_ADR ;ベクタテーブルの設定`

可変ベクタテーブル上ベクタ番号は割込み要求に対応しており(ハードウェアにより決定している)、個々のベクタ番号の切り替えはできません。

割込みには、禁止指定が可能なマスカブル割込みと禁止できないノンマスカブル割込みがあります。ノンマスカブル割込みはプロセッサによってはエクセプション(exception)に分類されるものもありますので、ここではM16Cのマニュアルに記載されているノンマスカブル割込みを例にとり解説を行います。ノンマスカブル割込みは想定外の事象が発生した場合の割込みです。

1) M16Cで定義されているソフトウェアを対象としたノンマスカブル割込み

未定義命令(UND命令)、オーバフロー(INTO命令)、BRK命令、INT命令

2) M16Cで定義されているハードウェアを対象としたノンマスカブル割込み

NMI、DBC、ウォッチドックタイマ、発振停止、再発信検出、電圧低下検出

シングルステップ、アドレス一致

マスカブル割込みは、抑制できる割込みです。主に外部のハードウェアからの割込みです。プロセッサによって異なりますが、多くのプロセッサは割込み優先度の設定ができ、優先度の低い割込みを禁止する機能を持ちます。

M16Cの割り込みアーキテクチャ：割り込み制御レジスタ

- ・ 割り込み要因毎に割り込み制御レジスタを持つ
 - － 割り込み優先レベルを設定 ←M16C/29グループマニュアル：P89
 - － 割り込み要求ビット
- ・ プロセッサに割り込みが受付られる条件は
 - － 割り込み許可フラグ(Iフラグ) = 1 & 割り込み要求ビット = 1
& 割り込み優先レベル > プロセッサ割り込み優先レベル(IPL)

ビットシンボル	ビット名	機能	RW
INT3IC	割り込み優先レベル選択ビット	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
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)
POIL	極性切り替えビット	0 : 立ち下がりエッジを選択 (注3、注4) 1 : 立ち上がりエッジを選択	RW
(b5)	予約ビット	"0" にしてください	RW
(b7-b6)		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。	—

ルネサス M16C/29グループ
ハードウェアマニュアルP89より抜粋

2008/11/28

TOPPERSプロジェクト認定

115



M16Cの割り込み制御レジスタについて説明を行います。M16Cの割り込みアーキテクチャでは、割り込み要因の設定を独立して設定できるものと、共有設定となっているものがあります。いくつかの割り込みが共有して設定する仕組みとなっています。上記のS4ICとINT5IC、S3ICとINT4Cは共有設定です。割り込み制御レジスタは割り込み要因側の割り込み設定を行うレジスタです。ILVL2～ILVL0は割り込み優先レベル選択ビットは割り込みデバイスの優先度レベルの設定を行います。IRは割り込み要求ビットは割り込みデバイスの割り込み状態を示します。割り込みが発生してプロセッサが受け付けていない状態で1にセットされ、割り込みが受け付けられると0にリセットされます。割り込みを中断するために0を設定できますが、1は設定できません。

M16Cに接続した割り込みデバイスに対して、割り込みを受けられる設定を以下に示します。

1) 割り込みデバイスに対応した割り込み制御レジスタに設定を行います。

ILVL2～ILVL0にレベル設定、例えばレベル3を設定します。

IRを1に設定します。

2) プロセッサのFLGレジスタに割り込み許可を設定します。

IPLに2以下の値を設定します。

この条件で、割り込みデバイスからプロセッサに割り込み信号が設定されれば、対応のベクタ番号にセットされた割り込みハンドラのアドレスから割り込み実行します。

M16Cの割込みアーキテクチャ：割込みシーケンス

- プロセッサによる割込み処理シーケンス
 1. 割込みを受け付ける
 2. 割込みの要因に対応した割込み制御レジスタのIRビットをクリア
 3. フラグレジスタをCPU内部の一時レジスタに退避
 4. フラグレジスタの割込み許可フラグ(I)を‘0’にして割込み禁止
 5. フラグレジスタのスタックポインタ指定フラグ(U)を‘0’にして割込みスタックを有効に
 6. CPU内部の一時レジスタをスタックに退避
 7. PCをスタックに退避
 8. プロセッサ割込み優先レベルに受け付けた割込み優先レベルを設定
 9. 割込みベクタテーブルに設定されたアドレスにジャンプ

ハードウェアが何をしてきて何をしてくれないかを明確にする

- RISCプロセッサの多くは4までの処理しか行わず、残りはプログラム（ソフトウェア）で処理しなければならない

2008/11/28

TOPPERSプロジェクト認定

116



最後にM16Cの割込みが実行される手順について説明を行います。割込みを発生させるにはデバイスがプロセッサ外にある場合は割込み信号(INT0～INT5)を通してプロセッサに割込み要求が発行されます。この要求を受けたプロセッサは以下の手順を自動で行います・

1. デバイスから割込みを受け付ける。このとき割込みベクタ番号を同時に取り込む
2. 割込み要因に対応した割込み制御レジスタのIRビットを0にクリアする
3. フラグレジスタの内容をプロセッサ内の一時レジスタに退避する
4. フラグレジスタのIフラグを0にクリアして割込み禁止状態にする
5. フラグレジスタのUフラグを0にクリアして割込みスタックポインタを有効にする。

(M16Cはユーザー使用のスタックポインタと割込み使用のスタックポインタを設定できる。どちら設定を指定するかは、フラグレジスタのUビットの指定に従う)

6. フラグレジスタを退避した一時レジスタの内容を割込みスタックに退避する。
7. 現在のPCを割込みスタックに退避する。割込みハンドラから、現在実行中のプログラムに戻る場合、割込みスタック上のフラグの値とPCの値を利用する。
8. フラグレジスタのIPLビットに割込み要因の優先度レベルをセットする。これにより、受け付けた割込みと同一以下の割込みは禁止される。
9. 割込みベクタ番号に設定されたアドレスにジャンプする。もちろん、このアドレスは割込みハンドラの先頭番地であるため、割込みハンドラが実行される。

割込みプログラミング

1. M16Cの割込みアーキテクチャ
2. [スイッチ割込みプログラム](#)
3. 宿題:タイマ割込みプログラム
4. 宿題:シリアルI/O割込みプログラム

スイッチ割込みプログラム1 : int_switch_1 概要

- SW9を押すと発生する割込み(INT3)により, LEDをカウントアップする
- 学習内容
 - 割り込みプログラムの全体像の理解
 - 割込みベクタテーブル
 - 割込み制御レジスタ
 - 割込みハンドラ
- 動作確認
 - 作成済みのプログラムを実行し動作を確認する

2008/11/28

TOPPERSプロジェクト認定

118



int_switch_1はSW9を押すとLEDをカウントアップするプログラムです。このとき、SW9の変化は割込みによって処理します。この実習ではint_switch_1プログラムを理解することにより、割込みプログラミングについて理解を行うことを目的とします。学習の内容は以下の順序でプログラムの説明を行っていきます。

1. 割込みプログラムの全体像の理解
2. 割込みベクタテーブル
3. 割込み制御レジスタ
4. 割込みハンドラ

最後に、プログラムをビルド、実行して動作を確認します。

int_switch_1 : 割込みプログラムの全体像

- 割込みをプログラムの視点から見ると
 - メイン関数を実行中に割込みが入ると、一旦メイン関数中の処理が中断され、割込みに対応した関数(割込みハンドラ)が実行される
 - 割込みハンドラの実行が終了すると、メイン関数の処理が再開される
- プログラムで必要となる処理
 - 割込み要因ハードウェアの初期化
 - 割込みハンドラの用意
 - 割込みベクタテーブルの用意
 - 割込み関連の初期化

2008/11/28

TOPPERSプロジェクト認定

119



まず、ソフトウェアの観点から、割込みがどのように実行されるかを確認しましょう。Int_switch_1においてもポーリングプログラムと同じように、通常はメイン関数(main)が実行しています。Int_switch_1では、LEDの制御は割込みハンドラで実行するためにメイン関数は空ループとなっていますが、通常の割込みを使った機器用のメイン関数は多くの処理を実行します。

メイン関数実行中に、SW9が押されると割込み要求がプロセッサに伝えられます。これによりメイン関数の処理が中断され、割込みハンドラ(int3_int_handler)が実行されます。割込みハンドラが終了するとメイン関数の処理が中断された場所から再開されます。

割込みプログラムを作成するために以下の処理が必要となります。

- 1) 割込み要因ハードウェアの初期化
- 2) 割込みハンドラの用意
- 3) 割込みベクタテーブルの用意
- 4) 割込み関連の初期化

int_switch_1 : 割込み要因ハードウェア

プッシュスイッチ(SW9)を割込み要因として用いる

- 接続
 - プログラマブル入出力ポート1のビット5(P15)に接続
 - P15は外部割込み3(INT3)とピンを共有
 - ボタンを押すと‘1’が入力される
- 割込み優先度 ←M16C/29グループマニュアル : P89
 - 割込み制御レジスタ(INT3IC)により設定
- 割込みベクタ ←M16C/29グループマニュアル : P87
 - 割込み番号4, 可変ベクタテーブルの0x10に割込みハンドラを登録する

2008/11/28

TOPPERSプロジェクト認定

120



BB STUDY I/Oボードはプッシュスイッチ(SW9)を割込み要因として用いるために必要なハードウェア設定が行われています。SW9の配線はJ2コネクタのP3ピンを通してM16Cの47番ピン(P15/INT3/ADTRG/IDV)に接続されています。このピンは入出力ポートP15やINT3等と共有したピンです。ポートを入力設定にするとINT3の入力として使用することができます。SW9を押した状態で‘1’、押さない状態で‘0’の入力となります。INT3はエッジ割込みで、極性に従って割込みを発生させます。極性は立ち上がりエッジ(電流が0から1の変化)と、立ち下りエッジ(電流が1から0に変化)を選択できます。押した状態で割込みを発生させるには、立ち上がりエッジを選択します。

INT3用の割込み制御レジスタに割込みレベルと極性を設定します。INT3用の割込みベクタに割込みハンドラのアドレスをセットします。

1) INT3用の割込み制御レジスタは、INT3IC(0x0044番地)

2) INT3用の割込みベクタは、割込み番号4番

M16C/29グループマニュアルを確認してください。

int_switch_1 : 割込み要因ハードウェアの初期化

ポート関連のレジスタと
割込み制御関連のレジスタの初期化が必要

- ポート関連レジスタの初期化
 - ポーリングプログラミング switch_push の初期化関数を流用
- 割込み制御関連のレジスタ
 - 割込み制御レジスタ
 - 割込みレベルや割込みエッジの設定
 - 割込み要因選択レジスタ ←M16C/29グループマニュアル : P90
 - 割込み極性切り替え(両エッジか片エッジか)

2008/11/28

TOPPERSプロジェクト認定

121



int_switch_1.cを参照してください。196行目からmain関数の記述があります。ハードウェアの初期化は以下の3つの関数で行います。

- | | |
|---------------------|----------------------------|
| 1) led_init | LEDの初期化を行う(ポーリングプログラムを参照) |
| 2) switch_push_init | スイッチポートの初期化(ポーリングプログラムを参照) |
| 3) int3_init | 割込みの初期化 |

割込みの初期化では割込み制御レジスタと割込み要因選択レジスタを用いて、割込み優先度レベルと極性の設定を行います。

割込み要因選択レジスタについてはM16C/29のマニュアルで確認しましょう。

int_switch_1 : 割込み制御関連のレジスタの定義

- 割込み制御レジスタ

```
#define BREG_SFR_INT3IC ((volatile unsigned char *)0x0044)
```

- 割込み制御レジスタビット定義

```
#define IC_ILVL_MASK 0x07 /* 割込み優先レベルのマスク */
#define IC_IR        0x08 /* 割込み要求ビット */
#define IC_POL       0x10 /* 立ち上がりエッジ */
```

- 割込み要因選択レジスタ

```
#define BREG_SFR_IFSR ((volatile unsigned char *)0x035F)
```

- 割込み要因選択レジスタビット定義

```
#define IFSR_INT3_EDGE 0x08 /* INT3両エッジ */
```

2008/11/28

TOPPERSプロジェクト認定

122



割込み制御レジスタの定義はBREG_SFR_INT3ICの定義しています。割込み制御レジスタ内のビット定義は、以下の定義で行っています。

- | | |
|-----------------|--|
| 1) IC_ILVL_MASK | 割込み優先度レベルのマスク設定 |
| 2) IC_IR | 割込み要因ビット(0のみ設定なので、この定義は使用しない) |
| 3) IC_POL | 立ち上がりエッジ指定(スイッチを押した場合のみ割込み処理を行うので、立ち上がりエッジを指定) |

割込み要因選択レジスタは極性を片エッジ(立ち上がり、立ち上がりのどちらか)か両エッジ(立ち上がりと立ち上がりの両方)の選択をします。割込み要因選択レジスタの定義はBREG_SFR_IFSRの定義です。割込み要因選択レジスタはINT0からINT7までの極性をビット単位に指定します。INT3の極性は以下の定義です。

- | | |
|-------------------|-------------|
| 1) IFSR_INT3_EDGE | INT3の両エッジ指定 |
|-------------------|-------------|

int_switch_1 : 割込み制御関連のレジスタの初期化

- 優先度を引数とし、割込み制御レジスタと割込み要因選択レジスタを初期化する
- プッシュボタンを押した際に割込みが入るよう立ち上がりエッジかつ片エッジを選択

```
void
int3_init(unsigned char ilvl){
    /* 設定可能ビット以外をクリア */
    ilvl = (ilvl & IC_ILVL_MASK);
    /* 割込み制御レジスタを初期化 */
    *BREG_SFR_INT3IC = (IC_POL | ilvl);
    /* 割込み要因選択レジスタ設定(片エッジ) */
    *BREG_SFR_IFSR = (*BREG_SFR_IFSR & ~IFSR_INT3_EDGE);
}
```

2008/11/28

TOPPERSプロジェクト認定

123



割込みの初期化関数int3_initについて説明を行います。引数ilvlは割込み優先度レベルの指定です。

```
ilvl = (ilvl & IC_ILVL_MASK);
```

割込み優先度レベルをマスクした値をilvlにセットします。

```
*BREG_SFR_INT3IC = (IC_POL | ilvl);
```

指定された割込み優先度と立ち上がりエッジ指定をINT3用の割込み制御レジスタに設定します。

```
*BREG_SFR_IFSR = (*BREG_SFR_IFSR & ~IFSR_INT3_EDGE);
```

割込み要因選択レジスタの内容を読み出し、INT3に対応したビットを0にリセットし、割込み要因選択レジスタに書き込みます。これにより、INT3を片エッジに指定します。

int_switch_1 : 割込みハンドラ

- グローバル変数をカウントアップしてLEDに出力

```
void
int3_int_handler(void){
    /* LEDをカウントアップ */
    led_out_bdigit(++led_count);
}
```

- C言語関数を割込みハンドラとする場合の問題点
 - 単に関数を割込みハンドラとすると正しく動作しない
 - 割込みハンドラの出入り口では、元の処理の保存と復帰を行う必要がある

割込みハンドラ(int3_int_handler)について説明を行います。割込みハンドラはled_out_bdigit関数を呼び出し、引数の値を2進でLEDに表示します。

割込みハンドラは、普通のC言語の関数とは異なる処理が必要となります。割込みハンドラの入り口では前の実行状態(レジスタ等)を保存し、出口では前の実行状態に戻す必要があります。また、出口の復帰時、FLGレジスタ(とPC)の内容が切り替わっているため、この内容を元の状態に戻す復帰命令REITを実行して復帰しなければなりません。C言語の復帰命令は呼び出した次のアドレスにPCに復帰し、割込み用の復帰は起こりません。

これを解決するために#pragumaを用いて、コンパイラに割込みハンドラの指定を行います。

int_switch_1 : 割込みハンドラ #pragma指定

- 割込みハンドラの出入り口処理
 - 元の処理に戻るための必要な情報(コンテキスト)は残す必要がある(スタックに)
 - 割込みハンドラから戻る場合は関数からのリターンとは異なる命令で戻す必要がある(M16Cではreit)
- コンパイラによる出入り口処理の付加
 - 関数が割込みハンドラであることを通知すると, 割込みハンドラ用の出入り口処理を付加したアセンブリコードを生成可能なコンパイラが存在する(NC30も)
 - コンパイラへは一般的に#pragmaで通知する
 - #pragma の書式はコンパイラによって異なる

```
#pragma INTERRUPT int3_int_handler
```

割込みハンドラ用のプリAGMA指定は関数本体前に記述します。**#pragma INTERRUPT**を指定した関数は、コンパイルにて**REIT**命令や、レジスタの保存を行うアセンブラ言語を生成します。この宣言を行わない場合、割込みが発生しハンドラが実行する暴走します。割込みハンドラのプリAGMA指定はコンパイラによって書き方が異なりますので注意が必要です。

int_switch_1 : 割込みベクタテーブル

- 可変ベクタテーブル
 - start.a30に記述

```

絶対アドレス指定      .SECTION vector
                        .ORG      VECTOR_ADR
                        ;
                        .LWORD    _irregular_int_handler_entry    ; 00
                        .LWORD    _irregular_int_handler_entry    ; 01
                        .LWORD    _irregular_int_handler_entry    ; 02
                        ....
                        ソフトウェア
                        割込み番号

```

- INTBの初期化
 - スタートアップルーチンで初期化

```

start:
    ....
    ldintb    #VECTOR_ADR    ; ベクタテーブルの設定

```

2008/11/28

TOPPERSプロジェクト認定

126



割込みベクタの登録をおこないます。M16Cの開発環境では標準の割込みベクタテーブルはstart.a30にアセンブラで記載されています。ここではstart.a30を直接書き換えて割込みベクタを登録します。INTBの設定もstart.a30中で行っています。start.a30はスタートアッププログラムと呼ばれ、メイン関数を実行するためのハードウェアやC言語の初期化処理を行うプログラムです。スタートアップルーチンはC言語の初期化されていない状態で実行するためアセンブラ言語で記述を行います。

start.30の内容理解のためにM16Cのアセンブラ言語を簡単に説明します。

- SECTION セクションの定義を行う擬似命令
- ORG 以下に指定するプログラムやデータの先頭アドレスを指定する擬似命令
- GLB 以下のラベルが外部宣言されていることを示す擬似命令
- LWORD 4バイトでデータをおく擬似命令 (ベクタアドレスは4バイトで指定します)

以下のハンドラは未登録ハンドラ用の割込みハンドラです。レジスタを使用せず、reit命令により、元のプログラムの復帰のみを行うプログラムです。

```

;-----
;      未登録割込み用のハンドラ
;-----
.SECTION program
_unregist_int_handler_entry:
_unregist_ext_handler_entry:
    reit

```

int_switch_1 : 割込みハンドラの登録

- 可変ベクタへ登録
 - 割込みハンドラのアドレスをINT3の割込みベクタ番地である 0x10 番地 (0x10 : 16/4 byte = 4番目) に配置
 - C言語の関数名を書くと開始アドレスに置き換わる
 - コンパイル後の関数名には_が付く
 - .GLB宣言を忘れないこと (extern 宣言)

外部宣言..

```
.LWORD  _irregular_int_handler_entry  ; 03
.GLB    _int3_int_handler
.LWORD  _int3_int_handler             ; 04
.LWORD  _irregular_int_handler_entry  ; 05
....
```

2008/11/28

TOPPERSプロジェクト認定

127



start.a30への可変ベクタテーブルへの登録方法について説明を行います。

- 1) 割込みハンドラ. GLB宣言で、**extern**宣言します。NC20でコンパイルしたラベル名は、関数の前に__が付加されます。アセンブラ言語では__付けた割込みハンドラ名を指定する必要があります。
- 2) 該当する割り込みベクタテーブルのダミー割り込み関数名から使用する割り込み処理関数名に置きかえます。INT3は割り込み番号が4のハンドラ名を書き換えます。

int_switch_1 : メインルーチン

- 初期化後, 割込みを許可して無限ループで割込みを待つ
- 割込みの許可はC言語では直接記述できない(プロセッサ毎に命令が異なる)ため, インラインアセンブラによりアセンブル言語による記述する
 - 記述方法はコンパイラによって異なるので注意が必要

```
void
main(void){
    led_init();
    switch_push_init();
    int3_int_init(INT3_INT_LVL);
    led_count = 0;
    _asm( "fset 1");
    for (;;) {
    }
}
```

インラインアセンブラ

フラグレジスタのIビットをセットし割込みを許可

何も行わない無限ループ

2008/11/28

TOPPERSプロジェクト認定

128



最後にメイン関数について説明を行います。初期化後、カウンタをゼロに初期化して、割込みを許可しforループによる無限ループ中で割り込みを待ちます。

```
_asm("fset 1");
```

上記のインラインアセンブラは、割込みを許可する命令でFLGレジスタのIビットを1にセットします。C言語で直接書けませんので、インラインアセンブラを使用しています。電源投入後、FLGのIビットは0に設定されていますので、この設定が行われるまでは割込みは許可されません。

int_switch_1をビルドし、デバッガを用いて実行してみてください。

スイッチ割込みプログラム2 : int_switch_2 概要

- メイン関数でLEDを一定周期でカウントアップし, SW9 (INT3)が入力されるとカウント値をクリアする
- 学習内容
 - メインルーチンと割込みハンドラ間でのデータの共有
 - 割込み禁止の必要性(クリティカルセクション)
- 動作確認
 - 作成済みのプログラムを実行し動作を確認する

int_switch_2ではメイン関数でLEDをカウントアップし、SW9でクリアプログラムです。このプログラムではLEDへの表示をメイン関数と割込みハンドラの両方で行います。実はメイン関数でLEDをアクセス中に割込みが発生しLEDに対して上書きを行うと表示がおかしくなります。LEDは簡単なポート出力で順序性はあまりありませんが、LCDのようなコマンドを設定して制御してする表示デバイスでは、完全に表示がおかしくなってしまいます。

次に、配布のint_switch_2について解説を行い、どのように修正すれば問題が回避できるか考えて見ましょう。

int_switch_2 : データ共有と割込みハンドラ

メインルーチンと割込みハンドラ間でのデータの共有は、
グローバル変数(共有変数)で実現する

- LEDのカウントデータはグローバル変数とする

```
unsigned char led_data;
```

- 割込みハンドラではカウントデータをクリアする

```
void
int3_int_handler(void){
    led_data = 0;
    led_out_bdigit(led_data);
}
```

データのクリア

LED用のカウントデータ`led_data`はグローバル変数として定義してあります。これによりメイン関数でも、割込みハンドラでもLEDカウンタの変更ができます。割込みハンドラでのLEDの表示前に`led_data`にゼロをセットすることにより、LEDのクリアを行います。

int_switch_2 : メイン関数

- led_dataのカウンタアップは外部関数で行う
- 外部関数の戻り値(インクリメントした値)を led_data に入れる

```
void
main(void){
    led_init();
    switch_push_init();
    int3_init(INT3_INT_LVL);
    led_data = 0;
    _asm("fset    I");
    for (;;) {
        led_data = inc_data(led_data);
        led_out_bdigit(led_data);
        busy_wait();
    }
}
```

LEDのカウンタ
アップ

2008/11/28

TOPPERSプロジェクト認定

131



今度はメイン関数を解説します。led_dataは割込み許可の前にゼロにリセットします。永久ループ中にLEDのカウンタをアップし表示を行い、次の表示まで待つプログラムが記述されています。

led_data = inc_data(led_data);

led_dataをカウンタアップする。

led_out_bdigit(led_data);

LEDにカウンタを表示する。

busy_wait();

次のカウンタアップと表示まで待ちを入れる。

inc_data();関数は引数に+1して戻り値を返す関数です。busy_wait();関数はダミーループを行い時間待ちを行う関数です。

int_switch_2 : カウントアップ関数

- 引数の値をインクリメントして返す
- busy_waitにより故意に時間を消費している
 - 関数の実行によりある一定の処理時間が必要であるという想定を模している
 - 外部関数がライブラリで提供されていると処理時間は分からない場合がある
 - busy_wait()は以前のプログラムの半分の時間待つ

```

unsigned char
inc_data(unsigned char data){
    busy_wait();
    return (data + 1);
}

```

ダミーウェイト

2008/11/28

TOPPERSプロジェクト認定

132



inc_data関数について説明を行います。Inc_data関数は単純に引数をカウントアップするプログラムではなく、カウントアップ前にbusy_wait関数を入れて待ちを行うようにプログラムしています。これはLEDのカウントアップをアップするのにbusy_waitの設定時間分実行時間が必要となると想定した設計とするためです。(カウンタアップはデータをインクリメントするだけなので時間はかからない、しかし、実際の組込み機器では種々の処理でカウントアップに時間がかかるものと想定している:これにより、メイン関数と割込みハンドラ間で排他制御しないと表示がおかしくなる環境を作り出している)

メイン関数のループ部は、以下のような処理を行います。

```

↓ -----+
1) led_dataを取り込み      |
2) busy_wait分待ちを行う   |
3) カウンタをアップ        |
4) led_dataに書き込む      |
5) LEDにカウンタを表示    |
6) busy_wait分待ちを行う   |
-----+

```

int_switch_2 : プログラムの問題点

前述のプログラムには問題があり正しく動作しない

- SW9を押して割込みハンドラでLEDをクリアしても, 次のカウントのタイミングで'1'ではなく, 前のカウント値をインクリメントした値が表示される

実際に動作させて問題点を確認する

- メイン関数ではグローバル変数 `led_data` を引数に `inc_data()` を呼び出し, 戻り値を `led_data` に戻している
 ➡ グローバル変数の値を読み込んでから書き戻すまでに時間が必要

2008/11/28

TOPPERSプロジェクト認定

133



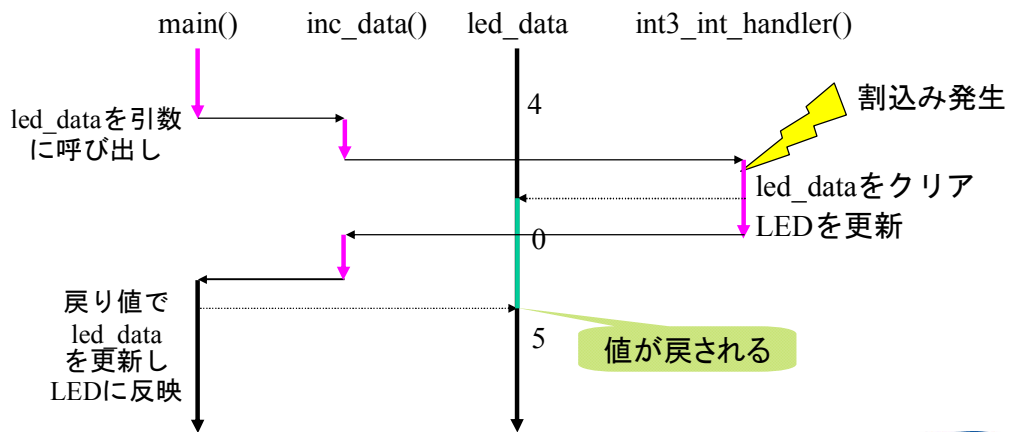
それでは、`int_switch_2`をビルドして実行しプログラムの確認を行ってください。SW9を押すと割込みハンドラで`led_data`をクリアしているにもかかわらず、たまにLEDのクリア後'1'ではなく、クリア前のカウンタ+1の値が表示されることがあります。

このような現象はなぜ発生するのでしょうか？

問題は`inc_data`関数にあります。`inc_data`関数前後の`led_data`値を変化を次のスライドで説明します。

int_switch_2 : led_dataの更新

- main()関数でled_dataを読み込んでinc_data()を実行している間に、割り込みが発生し、割り込みハンドラでled_dataをクリアすると、その後main()関数によって上書きされてしまう



2008/11/28

TOPPERSプロジェクト認定

134



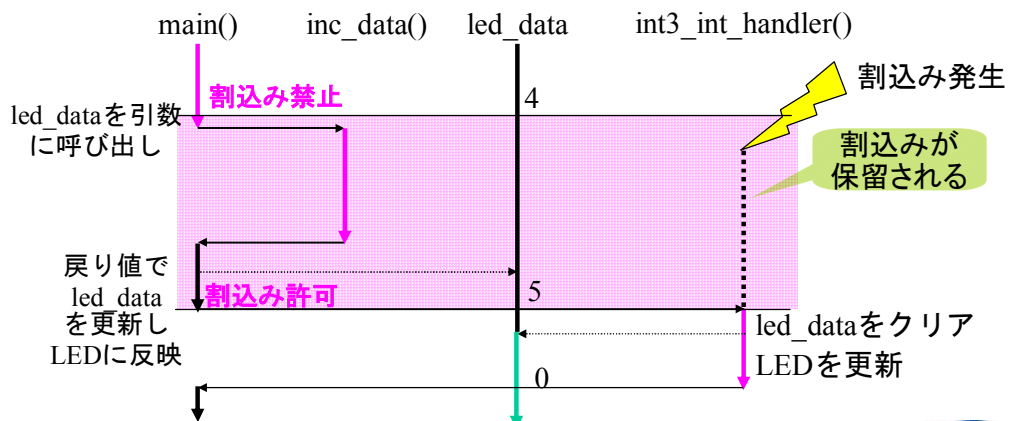
メイン関数での`led_data`の値を、上の図で説明します。まず、`led_data`に‘4’がセットされているとします。`inc_data`関数を実行すると、`led_data`の値‘4’が`inc_data`関数に渡されます。`inc_data`関数の終了前に割り込みが発生すると、割り込みハンドラ内で`led_data`がクリアされて‘0’がセットされます。メイン関数に戻って`inc_data`関数の終了時、`inc_data`関数は引数として渡された‘4’に1を加えて‘5’を戻り値として返します。戻り値は`led_data`に書き込まれ、`led_data`は‘0’から‘5’に書き換わります。

これにより、`led_data`は割り込みハンドラでは正しくクリアされません。

int_switch_2 : 割込みの禁止

- 共有データ(led_data)を排他的に扱えば問題は発生しない
- main()関数でled_dataを更新する間はSW9の割込みがプロセッサによって受け付けられないようにすればよい

➡ 割込み禁止により実現



2008/11/28

TOPPERSプロジェクト認定

135



どのようにプログラムを書き換えれば、led_dataは正しくクリアできるのでしょうか？

上記の図のように、inc_data関数の実行中は割込みが発生しないようにすれば、このような現象は発生しません。割込みの禁止・許可を行うにはFLGのビットを操作すれば、led_dataの実行前に割込み禁止状態に、実行後に割込み許可状態に設定ができます。

int_switch_2 : 割込みの禁止 プログラムでの実現

- 割込み禁止・許可命令で実現する
 - プロセッサ毎に実現方法は異なる
 - 割込みマスクで実現する方法も(後述)

割込み禁止

```
void
main(void){
  ...省略...
  for (;;) {
    _asm( "fclr I");
    led_data = inc_data(led_data);
    led_out_bdigit(led_data);
    _asm( "fset I");
    busy_wait();
  }
}
```

割込みが発生しても受け付けられず、割込みハンドラと排他的に実行される

2008/11/28

TOPPERSプロジェクト認定

136



int_switch_2.cのメイン関数を改造して割込み禁止・許可を追加してください。

_asm("fclr I"); FLGのビットにゼロを設定:割込み禁止
_asm("fset I"); FLGのビットに1を設定:割込み許可

プログラムを改造し、実行して確認をしてください。

int_switch_2 : クリティカルセクション

排他的に動作する(しなければならない)区間

- 実現方法
 - メイン関数と割込みハンドラ間や割込みハンドラ間は、割込み禁止・許可命令を組み合わせで実現
- クリティカルセクションの実行時間
 - 長くなると、割込みが発生してから応答するまでの時間(割り込み応答時間)が長くなり、リアルタイムシステムでは問題となる場合がある
 - 修正したプログラムでは、スイッチを押してからクリアされるまでのタイムラグが発生する

2008/11/28

TOPPERSプロジェクト認定

137



クリティカルセクションとは、単一の処理やデータに対して、複数の操作が同時に発生すると破綻をきたす区間をさします。Int_switch_2ではinc_data関数の実行中がクリティカルセクションとなります。この区間は排他制御を行わなければなりません。通常処理と割込み関数の間の排他制御は、割込み禁止・許可を用いて排他制御を行います。

割込み禁止時間を長くすると、スイッチの応答性が低下します。実際のプログラムではinc_data内で細かくled_dataを排他制御するのが正しい対応となります。

実際の組込みシステムでは割込み禁止時間を長くするとハードリアルタイム性を保障できないケースが発生します。例えば、I2CやUARTのような割込みを使ったシリアルデバイスでは割込み禁止時間が長すぎると受信データの取りこぼしが発生しますので注意が必要です。

スイッチ割込みプログラム3 : int_switch_3 概要

- SW9を押すと発生する割込み(INT3)により, LEDをカウントアップし, SW8を押すと発生する割込み(INT4)によりLEDをカウントダウンする
- 学習内容
 - 複数の割込みの扱い
 - 割込み関連のマニュアル読み
 - 割込み優先度
- 作成方法
 - int_switch_1 をベースに開発

2008/11/28

TOPPERSプロジェクト認定

138



int_switch_3では、int_switch_1を改造します。SW9に加えて、SW8を押すと割込みが発生するように修正しLEDをカウントダウンするようにしてください。

この自習では、以下のプログラミングについて学習します。

- 1) 複数の割込みの扱い方
- 2) 割込み関連のマニュアルの読み方
- 3) 割込み優先度の設定

int_switch_3 : 作成

プログラム int_switch_3 を作成

- ポートの初期化, SW9の割込み関連の初期化は流用する
- 手順
 - 割込み要因ハードウェア(SW8)の整理
 - 割込み関連のレジスタの整理
 - SW8の割込み(INT4)は, 他の割込みと共有しているため, SW9 (INT3)と比較して初期化項目が多い
 - プログラミング
 - 割込み関連のレジスタの定義
 - 割込み関連の初期化
 - 割込みハンドラ
 - 割込みベクタテーブルへの登録

2008/11/28

TOPPERSプロジェクト認定

139



SW8を割込み処理する手順に関して説明を行います。SW8はINT4に対応しています。基本的にはINT3用のプログラムを改造すれば、SW8への対応が可能です。以下の手順でプログラムの改造を行います。

- 1) SW8用の割込み要因ハードウェアの整理を行う。
- 2) SW8の割込みINT4の割込み制御レジスタ、割込み要因選択レジスタ、割込みベクタをマニュアルを用いて確認します。割込み要因選択レジスタ他の割込みレジスタと共有しています。
- 3) プログラミングを行います。プログラミングは以下の手順で行います。
 - 3-1) 割込み関連のレジスタの定義を行います。
 - 3-2) INT3の割込み関連の初期化を追加します。
 - 3-3) INT3用の割込みハンドラを追加します。
 - 3-4) INT3用の割込みベクタテーブルの改造を行います。

int_switch_3 : 割込み関連のハードウェア整理

- 割込み要因ハードウェア (SW8) の整理
 - プログラマブル入出力ポート1のビット6 (P16) に接続
 - P16は外部割込み4 (INT4) とピンを共有
 - ボタンを押すと'1'が入力される
- 割込み関連のレジスタの整理
 - 割込み優先度 ←M16C/29グループマニュアル : P89
 - 割込み制御レジスタ (INT4IC) により設定
 - 割込み要因選択レジスタ ←M16C/29グループマニュアル : P90
 - INT4とSIO3の割込みは共有しているためINT4を有効に
 - 割込みベクタ
 - 割込み番号9, 可変ベクタテーブルの0x24に割込みハンドラを登録する ←M16C/29グループマニュアル : P87

2008/11/28

TOPPERSプロジェクト認定

140



BB STUDY I/Oボードはプッシュスイッチ (SW8) を割込み要因として用いるために必要なハードウェア設定が行われています。SW8の配線はJ2コネクタのP4ピンを通してM16Cの46番ピン (P16/INT4/IDW) に接続されています。このピンは入出力ポートP16やINT4等と共有したピンです。ポートを入力設定にするとINT4の入力として使用することができます。SW8を押した状態で'1'、押さない状態で'0'の入力となります。INT4はエッジ割込みで、極性に従って割込みを発生させます。極性は立ち上がりエッジ (電流が0から1の変化) と、立ち下りエッジ (電流が1から0に変化) を選択できます。押した状態で割込みを発生させるには、立ち上がりエッジを選択します。

入出力ポート1のP6を入力設定にします。P16のポート設定はP15と同様のSFDのP1 (0x03e1) とPD1 (0x03e3) を使用し、ビット6に対応を行います。このSW8に対しての設定はint_switch_1.cに設定済みなのでプログラムを変更する必要はありません。

INT4の割込み制御レジスタはINT4IC (0x0049) を使用し、割込み要因選択レジスタはINT3と共用のIFSR (0x035F) を使用します。INT4の割込みベクタは9番を使用します。このベクタはシリアルポートと共有設定です。マニュアルで、これらのレジスタを確認してください。

int_switch_3 : 割込み関連のレジスタの定義

- 割込み制御レジスタ

```
#define BREG_SFR_INT4IC ((volatile unsigned char *)0x0049)
```

- 割込み制御レジスタビット定義はSW9と共有

```
#define IC_ILVL_MASK 0x07 /* 割込み優先レベルのマスク */
#define IC_IR        0x08 /* 割込み要求ビット */
#define IC_POL       0x10 /* 立ち上がりエッジ */
```

- 割込み要因選択レジスタ

```
#define BREG_SFR_IFSR ((volatile unsigned char *)0x035F)
```

- 割込み要因選択レジスタビット定義

```
#define IFSR_INT4_EDGE 0x10 /* INT4両エッジ */
#define IFSR_INT4_EN   0x40 /* INT4有効 */
```

2008/11/28

TOPPERSプロジェクト認定

141



INT4の割込み制御レジスタの定義をソースに追加します。

- 1) BREG__SFR__INT4C INT4用の割込み制御レジスタの定義

割込み制御レジスタのビット定義はINT3と共用で使用できます。そのため、定義を追加する必要はありません。

- 1) IC__ILVL__MASK INT3とINT4用の割込み優先度マスクの定義
- 2) IC__IR INT3とINT4割込み要求ビット(使用しない)
- 3) IC__POL INT3とINT4立ち上がりエッジ設定定義

割込み要因選択レジスタは、INT3とINT4は共用なので、定義を追加する必要はありません。

- 1) BREG__SFR__IFSR INT3とINT4用の割込み要因選択レジスタ

割込み要因選択レジスタ中のINT4用のビット定義を追加します。INT4はSIO3と共用の割込みなので、割込み指定をINT4に設定する必要があります。この設定は割込み要因選択レジスタのビット6を‘1’に設定することで指定ができます。以下に割込み要因選択レジスタに設定が必要なビット定義を追加します。

- 1) IFSR__INT4__EDGE INT4の両エッジの定義
- 2) IFSR__INT4__EN INT4の割込み有効定義

int_switch_3 : 割込み制御関連のレジスタの初期化

- 優先度を引数とし、割込み制御レジスタと割込み要因選択レジスタを初期化する
- INT4を有効にするように割込み要因レジスタをセットする

```
void
int4_init(unsigned char ilvl){
    /* 割込み要因選択レジスタ設定(INT4有効) */
    *BREG_SFR_IFSR = (*BREG_SFR_IFSR | IFSR_INT4_EN);
    /* 設定可能ビット以外をクリア */
    ilvl = (ilvl & IC_ILVL_MASK);
    /* 割込み制御レジスタを初期化 */
    *BREG_SFR_INT4IC = (IC_POL | ilvl);
    /* 割込み要因選択レジスタ設定(片エッジ) */
    *BREG_SFR_IFSR = (*BREG_SFR_IFSR & ~IFSR_INT4_EDGE);
}
```

2008/11/28

TOPPERSプロジェクト認定

142



INT4用の割込み初期化関数を追加します。この関数をint4_init関数とします。この関数はint3_init関数と同等に、引数として割込み優先レベルをしてするものとします。上記のようにこの関数を追加してください。各行の機能の説明に以下に記載します。

```
*BREG_SFR_IFSR = (*BREG_SFR_IFSR | IFSR_INT4_EN);
```

割込み要因選択レジスタの内容を読み込みビット6を‘1’にセットし、再び、割込み要因選択レジスタに書き込みます。これにより、割込みベクタ9の割込み要因をINT4に設定します。(‘0’の場合、SIO3)

```
ilvl = (ilvl & IC_ILVL_MASK);
```

引数で指定されたINT4の割込み優先レベルをINT4用割込み制御レジスタのビットレイアウトにあうようにマスクを行います。

```
*BREG_SFR_INT4IC = (IC_POL | ilvl);
```

INT4用の割込み制御レジスタに割込み優先レベルと立ち上がりエッジを設定します。

```
*BREG_SFR_IFSR = (*BREG_SFR_IFSR & ~IFSR_INT4_EDGE);
```

割込み要因選択レジスタの内容を読み込みビット4を‘0’にリセットし、再び、割込み要因選択レジスタに書き込みます。これにより、INT4の割込み極性を片エッジに指定します。

int_switch_3 : 割込みハンドラとベクターテーブルへの登録

- グローバル変数(led_count)をデクリメントしてLEDに出力
- SW9の割込みハンドラとの排他制御
 - #pragma INTERRUPT で生成した割込みハンドラは、割込みを禁止した状態で呼び出される

```
void
int4_int_handler(void){
    /* LEDをカウントダウン */
    led_out_bdigit(--led_data);
}
```

- 割込みハンドラは、可変ベクターテーブルの0x24番地に配置 (0x24 : 36/4 byte = 9番目)

```
.GLB    _int4_int_handler
.LWORD  _int4_int_handler      ; 09
```

2008/11/28

TOPPERSプロジェクト認定

143



次にINT4用の割込みハンドラを追加します。ハンドラ名をint4_int_handlerとします。Int3_int_handlerを参考に作成してください。led_out_bdigitはそのまま使用できます。led_out_bdigit関数にカウントを渡す場合、カウントダウンするので、--led_dataに書換えカウントダウンするようにプログラムを変更してください。Int4_int_handlerは割込みハンドラなので以下のようにpragma INTERRUPT指定を追加してください。

```
#pragma      INTERRUPT    int4_int_handler
```

start.a30を書き換えて、INT4用割込みベクタ記述を書き換えてください。

```
.LWORD  _unregist_int_handler_entry      ; 09
```

↓

```
.GLB    _int4_int_handler
```

```
.LWORD  _int4_int_handler      ; 09
```

int_switch_3 : 割込み優先度

割込み優先度を学ぶ

- 割込みを抑止する方法として割込み優先度を用いる
 - 割込み要因毎に優先度を設定し、プロセッサの優先度より高い割込みのみを受付ける
 - 割込み優先度の概念がなく、割込みを個別に禁止・許可する機能しかもたないプロセッサも存在
- 優先度を適切に設定すると特定の割込みを禁止できる
 - 重要な割り込みのみを受付ける
 - 多重割込みとの組み合わせ
- 割込み優先度の設定
 - プロセッサの特殊レジスタ
 - 割込みコントローラで設定

2008/11/28

TOPPERSプロジェクト認定

144



int_switch_2では割込み許可・禁止設定を行うのにFLGレジスタのIビットを用いて制御を行いました。Iビットで割込み禁止を設定すると、割込み優先度に関係なく、マスカブル割込みは禁止となります。FLGレジスタのIPLを設定することにより、優先度の低い割込みのみを禁止することができます。

割込みを抑止する方法として割込み優先度を用いる方法があります。これにより、優先度の高い割込みのみを受け付けるしくみを構築できます。プロセッサによって割込み優先度の設定がなく、割込みを個別に禁止・許可する機能しかもたないプロセッサもありますので注意が必要です。

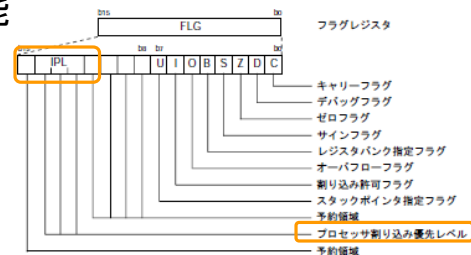
割込み優先度レベルを適切に設定することにより、重要な割込みを優先的に受け付けることができます。(M16Cの場合、割込みのハードウェア実行処理でFLGのIPLが実行するIPLに設定されるので、それ以下の割込みは禁止となる。ireit命令の実行でIPLの値は元にもどされるので、低い割込みも許可される。)例えば、多重に割込みを受け付けた場合、優先度レベルの高い割込みハンドラの実行中は、それより高い割込みレベルの割込みのみ受け付けることができます。低いレベルの割込みは高いレベルの割込みハンドラの実行後、割込みハンドラが起動します。

int_switch_3 : M16Cの割り込み優先度

- 割り込み要因の優先度
 - 各割り込み要因は、割り込み制御レジスタにより割り込み優先度を0～7の範囲で設定可能
 - 割り込み関連初期化関数の引数で設定
- プロセッサの割り込み優先度
 - フラグレジスタのIPLフィールドで設定可能
 - ldip命令により設定可能

設定値

```
_asm("ldipl #001h");
```



ルネサス M16C/29グループ
ハードウェアマニュアルP36より抜粋

M16CのIPLの設定はldipl命令(アセンブラ言語)で設定します。
優先度レベルを‘1’に設定するには以下のように記述します。

```
_asm("ldipl #001");
```

int_switch_3 : 割込み優先度 メイン関数

- 割込みを許可する前に割込み優先度を設定する

```
void
main(void){
    led_init();
    switch_push_init();
    int3_init(4);          /* INT3割込み優先度設定 */
    int4_init(2);          /* INT4割込み優先度設定 */
    led_data = 0;

    _asm("ldipl #001h"); /* 割込み優先度設定 */
    _asm("fset I");      /* 割り込み許可 */

    for (;;) {
    }
}
```

2008/11/28

TOPPERSプロジェクト認定

146



最後にメイン関数を書き換えます。INT4の割込み初期化と割込み優先度を設定します。割込み優先度は2を設定します。初期化部に以下の処理を追加します。

```
int4_init(2);
```

メイン関数の割込み優先度設定(IPL)を1に設定します。

```
_asm("ldipl #001");
```

改造が終わった時点で、プログラムをビルド実行して、SW8が機能することを確認してください。

int_switch_3 : 割込み優先度 プログラムによる確認

割込み優先度の効果をプログラムにより確認

- 次の条件になるよう int_switch_3 を書き換えてそれぞれ実行する
 - 条件1
 - SW9(INT3) : 優先度 4
 - SW8(INT4) : 優先度 2
 - プロセッサ : 優先度 1
 - 条件2
 - SW9(INT3) : 優先度 4
 - SW8(INT4) : 優先度 2
 - プロセッサ : 優先度 2
 - 条件3
 - SW9(INT3) : 優先度 4
 - SW8(INT4) : 優先度 2
 - プロセッサ : 優先度 4
 - 条件4
 - SW9(INT3) : 優先度 2
 - SW8(INT4) : 優先度 4
 - プロセッサ : 優先度 3

2008/11/28

TOPPERSプロジェクト認定

147



最後にメイン関数を書き換え、4つの条件で割込み優先度を変更して割込み禁止・許可の確認を行ってください。

条件1:	SW9(INT3)	優先度4	→両方の割り込みが入る
(最初の条件)	SW8(INT4)	優先度2	
	プロセッサ	優先度1	

条件2:	SW9(INT3)	優先度4	→SW8は入らない
	SW8(INT4)	優先度2	
	プロセッサ	優先度2	

条件3:	SW9(INT3)	優先度4	→両方入らない
	SW8(INT4)	優先度2	
	プロセッサ	優先度4	

条件4:	SW9(INT3)	優先度2	→SW8しか入らない
	SW8(INT4)	優先度4	
	プロセッサ	優先度3	

割込みプログラミング

1. M16Cの割込みアーキテクチャ
2. スイッチ割込みプログラム
3. [宿題:タイマ割込みプログラム](#)
4. 宿題:シリアルI/O割込みプログラム

タイマ割込みプログラム : int_timer 概要

- タイマA0の割込みを用いて1秒間隔でLEDをカウントアップする
- 学習内容
 - タイマ割込みの扱い方
- 作成方法
 - ポーリングプログラム timer をベースに作成

int_timer : 作成

プログラム int_timer を作成する

- 手順
 - 割込み要因ハードウェアの初期化(タイマA0)
 - カウントソース, カウント値を適切に設定してスタートさせる
 - 割込み制御レジスタの初期化
 - 割込み優先度の設定
 - 割込みハンドラ
 - 10回呼び出されるとLEDをカウントアップする

int_timer : 割込み要因ハードウェアの初期化

- オートリロードなので, 一度スタートさせれば, 設定周期で割込みが発生する

```
void
timer_a0_init(void){
    /* タイマA0停止 */
    *BREG_SFR_TABSR = *BREG_SFR_TABSR & ~TABSR_TA0S;
    /* タイマA0モードの設定 */
    *BREG_SFR_TA0MR = TA0MR_F32_TMOD;
    /* カウント値の設定 */
    *HREG_SFR_TA0 = TIMER_COUNT;
    /* スタート */
    *BREG_SFR_TABSR = *BREG_SFR_TABSR | TABSR_TA0S;
}
```

int_timer : 割込み制御レジスタの初期化

- 割込み制御レジスタ関連のマクロ定義
 - タイマA0割込み制御レジスタ

```
#define BREG_SFR_TA0IC ((volatile unsigned char *)0x0055)
```

- タイマA0割込み制御レジスタの初期化

```
void  
timer_a0_int_init(unsigned char ilvl){  
    /* 設定可能ビット以外をクリアして初期化 */  
    *BREG_SFR_TA0IC = (ilvl & IC_ILVL_MASK);  
}
```


int_timer : 割込みハンドラ

- 10回呼び出されると, LEDをカウントアップ

```
void
timer_a0_int_handler(void){
    static unsigned char timer_count = 0;
    timer_count++;
    /* 10回目の呼び出し */
    if(timer_count == 10){
        led_out_bdigit(++led_data);
        timer_count = 0;
    }
}
```

- 可変ベクタテーブルの0x54番地に配置 (0x54: 84/4 byte = 21 番目)

```
.GLB      _timer_a0_int_handler
.LWORD    _timer_a0_int_handler      ; 21
```

割込みプログラミング

1. M16Cの割込みアーキテクチャ
2. スイッチ割込みプログラム
3. 宿題:タイマ割込みプログラム
4. 宿題:シリアルI/O割込みプログラム

シリアルI/O割込みプログラム : int_uart

- シリアル受信割込みを用いて, ポーリングプログラム uart と同様の機能を実現する
- 学習内容
 - シリアルI/O割込みの扱い方
- 作成方法
 - ポーリングプログラム uart をベースに作成

int_uart : 作成

プログラム int_uart を作成する

- 手順
 - 割込み要因ハードウェアの初期化 (UART0)
 - ポーリングプログラム uart を流用
 - 割込み制御レジスタの初期化
 - 割込み優先度の設定
 - 受信割込みと送信割込みは独立
 - 割込みハンドラ
 - データをシリアルI/Oから受信して, LEDをカウントアップし, 受信したデータをシリアルI/Oに送信する

int_uart : 割り込み制御レジスタの初期化

- 割り込み制御レジスタ関連のマクロ定義
 - UART0受信割り込み制御レジスタ

```
#define BREG_SFR_S0RIC ((volatile unsigned char *)0x0052)
```

- UART0受信割り込み制御レジスタの初期化

```
void  
uart0_rx_int_init(unsigned char ilvl){  
    /* 設定可能ビット以外をクリアして初期化 */  
    *BREG_SFR_S0RIC = (ilvl & IC_ILVL_MASK);  
}
```

int_uart: 割込みハンドラ

- LEDをカウントアップし、データを受信して、データを送信

```
void
uart0_rx_int_handler(void){
    unsigned char c;

    led_out_bdigit(++led_data);
    c = uart0_pol_getc();
    uart0_pol_putc(c);
}
```

LEDカウントアップ

データ受信

データ送信

- 可変ベクタテーブルの0x48番地に配置 (0x48:72/4byte = 18番目)

```
.GLB    _timer_a0_int_handler
.LWORD  _timer_a0_int_handler    ; 18
```

まとめ

小規模組込み開発のまとめ

- 小規模な組込み開発では少数の開発者でファームウェア、ソフトウェアの開発を行うため、ハードウェアの知識や開発環境を構築する知識も必要となる
- 開発言語はC言語を用いることが多いが、C言語で記述できない部分は、アセンブラ言語で開発する必要がある
- 開発環境はプロセッサメーカーの開発環境を用いることが多く、開発前にノウハウの蓄積が必要
- ハードウェアの制御は、ポーリングを用いるのもの、割り込みを用いて制御を行うものがあり、適切な方式を選択する必要がある
- 割り込みに制御はプロセッサ毎に異なる。特別な設定を行う必要があり、処理を熟知して開発を行う必要がある

小規模組込み開発のまとめ

- アプリケーション部分の開発については、オブジェクト指向やモデル設計を用いれば、可視的なソフトウェア開発ができる。
- このような開発は実行ROM、RAMサイズの増加を伴うことが多いので注意が必要
- NPO法人 組込みソフトウェア管理者・技術者育成研究会では、いろいろは組込みソフトウェア開発手法の紹介を行ってますので、組込み開発プロセスの参考にしてください。<http://www.sesame.jp/>

おまけ:カップラーメンタイマの作成

1. [仕様](#)

2. プログラムの作成

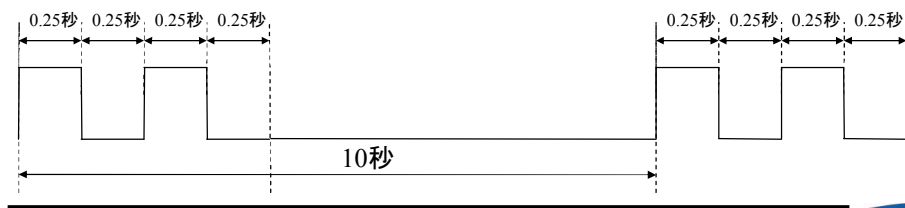
- 1) Step1
- 2) Step2
- 3) Step3
- 4) Step4
- 5) Step5
- 6) Step6

カップラーメンタイマの仕様

- 概要
 - タイマを起動してから、設定した時間が経過したことを知らせる
 - 設定時間は30秒刻みで設定できる
- スイッチとLEDの使い方
 - LED1 : 電源がONであること
(プログラムが動いていること)を表示
 - LED2 : SW8が押されたことを表示
 - LED4 : タイマが動いていることを表示
 - SW1 : タイマを起動・停止
 - SW8 : 設定時間を30秒延長

カップラーメンタイマの仕様

- 外部仕様
 - 電源をONの間(プログラムが動いている間)、LED1の点灯・消灯を1秒間隔で繰り返す
 - SW1がONになると設定時間を30秒にして、タイマを起動する
 - SW1がOFFになると、タイマを停止する
 - タイマが動いている間にSW8がONになると、ONの間LED2を点灯させ、設定時間を30秒延長する
 - タイマが動いている間は、10秒間隔で、LED4を2回点滅させる。LED4の2回点滅は、点灯・消灯を0.25秒間隔で2回繰り返すことで行う
 - 設定時間が経過すると、LED4を15秒間点滅させた後、消灯する。LED4の点滅は、点灯・消灯を0.25秒間隔で繰り返すことで行う



2008/11/28

TOPPERSプロジェクト認定

164



おまけ: カップラーメンタイマの作成

1. 仕様
2. プログラムの作成
 - 1) Step1
 - 2) Step2
 - 3) Step3
 - 4) Step4
 - 5) Step5
 - 6) Step6

カップラーメンタイマプログラム `cup_timer`

- カップラーメンタイマーの仕様を実現するプログラムを作成する
- プログラムの構造や処理の流れについては、構造化詳細設計演習の結果を用いる
- ハードウェアの操作に関してはこれまでに学習した内容を用いる
 - LEDの点灯
 - スイッチ(ディップ, プッシュスイッチ)のスキャン
 - タイマによる時間生成
 - 割込みによる時間通知

作成の手順

5つのステップに分けて順に進める

- Step1 : cup_timer_step1
 - デバイスドライバの作成
- Step2 : cup_timer_step2
 - タイマによる基本時間(0.25秒)の作成
- Step3 : cup_timer_step3
 - メイン関数でのLED1の1秒間隔の点灯の実現
- Step4 : cup_timer_step4
 - スイッチプロセスの作成
- Step5 : cup_timer_step5
 - タイマプロセスの状態遷移判定と各状態の大枠作成
 - LED4の点灯の実現
- Step6 : cup_timer
 - タイマプロセスの状態遷移判定と各状態の詳細

おまけ: カップラーメンタイマの作成

1. 仕様
2. プログラムの作成
 - 1) [Step1](#)
 - 2) Step2
 - 3) Step3
 - 4) Step4
 - 5) Step5
 - 6) Step6

Step1 : デバイスドライバの作成

LED, スイッチ(ディップ, プッシュ), タイマの操作関数を作成

- LED, ディップスイッチ, プッシュスイッチ
 - ポーリングプログラムの初期化関数と操作関数を流用
 - LED1,2,4を個別にON,OFFする関数を追加

```
void set_led1_state(unsigned char state)
```

- SW1,SW8の状態を個別に読み込む関数を追加

```
unsigned char get_sw1_state(void)
```

- ON,OFF状態を示すマクロを定義

```
#define ON      1  /* LEDやスイッチON状態 */
#define OFF     0  /* LEDやスイッチOFF状態 */
```

Step1 : デバイスドライバの作成

- タイマ
 - タイマ割込みプログラムの初期化関数を流用
 - 初期設定後, スタートさせ周期的に割込みハンドラを実行する
 - 1msecでタイムアウトするようにカウント値を変更する
 - タイマ割込みプログラムでは100msecで設定

Step1 : プログラム例 LEDの個別ON,OFF関数

- LEDの状態保存関数

```
unsigned char LedState;
```

- LEDの個別ON,OFF関数

```
void  
set_led1_state(unsigned char state){  
    if (state == ON) {  
        LedState = LedState | LED1;  
    } else {  
        LedState = LedState & ~LED1;  
    }  
}
```

Step1 : プログラム例 スwitchの個別読み込み関数

- スwitch毎にON/OFFの状態を読み込む

```
unsigned char  
get_sw1_state(void){  
    if ((switch_dip_sense() & DSW1) == DSW1) {  
        return ON;  
    } else {  
        return OFF;  
    }  
}
```

- タイマのカウント値
 - f32で1msec周期になるように設定

```
#define TIMER_COUNT    624
```

おまけ: カップラーメンタイマの作成

1. 仕様
2. プログラムの作成
 - 1) Step1
 - 2) [Step2](#)
 - 3) Step3
 - 4) Step4
 - 5) Step5
 - 6) Step6

Step2 :タイマによる基本時間(0.25秒)の作成

- グローバル変数
 - 1msec毎にインクリメントする変数(BaseTime)を用意
- タイマ割込みハンドラ
 - 1msec周期で実行され, BaseTimeをインクリメント
- メイン関数
 - BaseTimeを用いて250msec周期の処理を実現
 - 各ハードウェアの初期化と割込みの許可
 - 無限ループ内でBaseTimeをチェックする

Step2 : プログラム例 タイマ割込みハンドラ

- グローバル変数 BaseTime
 - unsigned long (32bit)とする

```
unsigned long BaseTime;
```

- タイマ割込みハンドラ

```
#pragma INTERRUPT base_time_handler  
  
void  
base_time_handler(void){  
    BaseTime++;  
}
```

Step2 : プログラム例 メイン関数

- 各ハードウェアの初期化
 - メイン関数の先頭で実行

```
led_init();           /* LED初期化          */
switch_dip_init();    /* DIPスイッチ初期化  */
switch_push_init();   /* PUSHスイッチ初期化 */
timer_a0_init();       /* タイマA0初期化      */
timer_a0_int_init(TIMER_A0_INT_LVL); /* タイマA0割込み初期化 */
```


Step2 : プログラム例 メイン関数

- 250msec周期の処理

```

#define TIMER_GRANULE 250  /* 時間単位 */

void
main(void){
    unsigned long timer250 = TIMER_GRANULE;
    .....
    BaseTime = 0;
    _asm( "fset i");
    for (;;) {
        if (timer250 <= BaseTime) { /* 250m周期で実行 */
            /* 現在時刻の250msec後を設定 */
            timer250 += TIMER_GRANULE;
        }
    }
}

```

割込み許可

タイマ割込みハンドラで1ms周期でインクリメントされる

おまけ: カップラーメンタイマの作成

1. 仕様
2. プログラムの作成
 - 1) Step1
 - 2) Step2
 - 3) [Step3](#)
 - 4) Step4
 - 5) Step5
 - 6) Step6

Step3 :メイン関数でのLED1の1秒間隔の点灯の実現

- 250msec周期を用いて1秒毎の処理を実現
 - 4回の実行で1秒
- 1秒毎の処理
 - LED1の状態を反転させる
- メイン関数内のローカル変数
 - LED1の点灯状態を保持する変数(led1)
 - 1秒を生成するための変数(sec)

Step3 : プログラム例 メイン関数

- 4回実行されるとLED1を反転させる

```

#define TO_SEC      4    /* 1秒間の呼び出し回数 */
void main(void){
    unsigned char sec = 0;
    unsigned char led1 = OFF;
    .....
    if (timer250 <= BaseTime) {
        timer250 += TIMER_GRANULE;
        if (++sec == TO_SEC) {
            if (led1 == ON) {
                led1 = OFF;
            } else {
                led1 = ON;
            }
            sec = 0;
            set_led1_state(led1);
        }
    }
}

```

4回実行

LED1反転

LED1設定

おまけ: カップラーメンタイマの作成

1. 仕様
2. プログラムの作成
 - 1) Step1
 - 2) Step2
 - 3) Step3
 - 4) [Step4](#)
 - 5) Step5
 - 6) Step6

Step4 :スイッチプロセスの作成

- イベントの定義とビット割り当て
 - 開始 : EVT_TIMER_START 0x01
 - 停止 : EVT_TIMER_STOP 0x02
 - 時間アップ : EVT_TIMER_COUNT 0x04
- グローバル変数
 - スwitchの状態を保持するための変数(Sw1,Sw8)
 - イベント通知用変数(Event)
- スwitchプロセス
 - Sw1,Sw8の変化をチェックし, イベント通知用変数をセットする
 - Sw8の状態に応じて, LED2を点灯・消灯させる

Step4 : プログラム例 イベント定義とグローバル変数

- イベント定義

```
#define EVT_TIMER_START    0x01
#define EVT_TIMER_STOP     0x02
#define EVT_TIMER_COUNT    0x04
```

- グローバル変数

- メイン関数で初期化

```
unsigned char Event;
unsigned char Sw1;
unsigned char Sw8;
```

```
Event = 0;
Sw1 = get_sw1_state();    /* 現在のSW1の取り込み */
Sw8 = get_sw8_state();    /* 現在のSW8の取り込み */
```

Step4 : プログラム例 SW1の状態取得

- SW1の状態をチェックし, 状態が変化していれば, イベント変数に対応するイベントをセットする

```
void  
switch_process(void){  
    unsigned char sw;  
    sw = get_sw1_state();  
    if (Sw1 != sw) {  
        if (sw == ON) {  
            Event |= EVT_TIMER_START;  
        } else {  
            Event |= EVT_TIMER_STOP;  
        }  
        Sw1 = sw;  
    }  
}
```

開始イベント

終了イベント

Step4 : プログラム例 SW8の状態取得

- SW8の状態をチェックし, 状態が変化していれば, イベント変数に時間アップのイベントをセットする
- SW8の状態に合わせてLED2を点灯・消灯する

```
sw = get_sw8_state();  
if (Sw8 != sw) {  
    if (sw == ON) {  
        Event |= EVT_TIMER_COUNT;  
        set_led2_state(ON);  
    } else {  
        set_led2_state(OFF);  
    }  
    Sw8 = sw;  
}
```

時間アップ
イベント

LED2:ON

LED2:OFF

おまけ: カップラーメンタイマの作成

1. 仕様
2. プログラムの作成
 - 1) Step1
 - 2) Step2
 - 3) Step3
 - 4) Step4
 - 5) [Step5](#)
 - 6) Step6

Step5 : タイマプロセスの大枠とLED4点灯の実現

タイマプロセスの状態遷移判定と各状態の大枠作成
LED4の点灯の実現

- タイマプロセスの状態を示す列挙型(TIMER_MODE)
 - 計時終了 : STOP_TIMER_MODE
 - 計時中 : ACT_TIMER_MODE
 - タイムアウト : TIMEOUT_MODE
- タイマプロセスのローカル変数(static変数)
 - タイムアウト時間を保持する変数(max_time)
 - 現在の計時時間を保持する変数(current_time)
 - カップラーメンタイマのモードを保持する変数(timer_mode)
 - LED4のタイミング生成用変数(alarm)
 - LED4の点灯状態を保持する変数(led4)

Step5 : 列挙型とローカル変数

- タイマプロセスの状態を示す列挙型 (TIMER_MODE)

```
enum TIMER_MODE {  
    STOP_TIMER_MODE, /* 計時終了モード */  
    ACT_TIMER_MODE,   /* 計時中モード */  
    TIMEOUT_MODE      /* タイムアウトモード */  
};
```

- タイマプロセスのローカル変数 (static変数)
 - タイマプロセスは250msec周期で呼び出され、終了するため変数の値を保存するために static変数とする

```
static unsigned long max_time;  
static unsigned long current_time;  
static unsigned char timer_mode = STOP_TIMER_MODE;  
static unsigned char alarm = 0;  
static unsigned char led4 = OFF;
```

Step5 : 定数マクロの定義

- 各定数をマクロで定義する
 - タイムアウト時間の初期値
 - 時間アップ(延長)単位
 - タイムアウト時のLED4の点滅時間
 - カップラーメンタイマ動作通知のためのLED4点滅のインターバル
 - カップラーメンタイマ動作通知のためのLED4点滅の点滅時間

```
#define INIT_TIME      30  /* スタート時のタイムアウト時間 */
#define TIME_UNIT      30  /* 時間アップ単位 */
#define TIMEOUT_BLINK  15  /* タイムアウト時の点滅時間 */
#define ACT_INTERVAL   10  /* タイマ動作通知LEDのインターバル */
#define ACT_ON_TIME     1   /* タイマ動作通知LEDの点灯時間 */
```

Step5 : プログラム例 タイマプロセス状態遷移判定

- イベント通知用変数を参照してカップラーメンタイマのモード変数(状態)を設定する

```
/* switch_processより通知 */
if (Event != 0) {
    if (Event & EVT_TIMER_START) { /* 開始 */
        timer_mode = ACT_TIMER_MODE;
    }
    if (Event & EVT_TIMER_STOP) { /* 停止 */
        timer_mode = STOP_TIMER_MODE;
    }
    if (Event & EVT_TIMER_COUNT) { /* 時間アップ */
        if (timer_mode == ACT_TIMER_MODE) {

        }
    }
    Event = 0;
}
```

Step5 : プログラム例 各状態毎の処理

- timer_modeによりモードを判定して実行

```
switch (timer_mode) {  
    case ACT_TIMER_MODE: /* 計時中モード */  
        if(current_time >= max_time){ /* タイムアウト */  
        }else if((current_time % (ACT_INTERVAL*1000)) == 0){  
        }  
        break;  
    case TIMEOUT_MODE: /* タイムアウトモード */  
        /* 15秒間点滅したらタイマー停止状態に */  
        if (current_time >= (TIMEOUT_BLINK * TO_SEC)){  
        }  
        break;  
    case STOP_TIMER_MODE: /* 計時終了モード */  
        break;  
    default:  
        break;  
}
```

Step5 : プログラム例 LED4の点灯

- LED4のタイミング生成用変数(alarm)が0以上なら点灯させる

```
if (alarm > 0) {          /* LED4点灯要求 : 250ms経過 */
    if(led4 == ON) {
        led4 = OFF;
    } else {
        led4 = ON;
    }
    alarm--;
} else {                  /* 点灯要求がなくLED4がオンなら消灯 */
    led4 = OFF;
}
set_led4_state(led4); /* LED4の設定 */
```


おまけ: カップラーメンタイマの作成

1. 仕様
2. プログラムの作成
 - 1) Step1
 - 2) Step2
 - 3) Step3
 - 4) Step4
 - 5) Step5
 - 6) [Step6](#)

Step6 : タイマプロセスの状態遷移判定と各状態の詳細

- 状態遷移判定での処理
 - 開始
 - current_time を0で初期化
 - max_time を設定し30秒でタイムアウトするよう設定
 - 停止
 - alarm を0にしてLED4の点灯を停止
 - 時間アップ
 - 計時中モードであれば max_time を30秒延長

Step6 : タイマプロセスの状態遷移判定と各状態の詳細

- 各状態(モード)での処理
 - 計時中モード
 - タイムアウトしていればモードをタイムアウトモードに移行させる
 - 10秒間隔のLED4の点灯が必要かチェックし, 必要ならLED4を点灯させるため alarm を設定する
 - タイムアウトモード
 - 15秒経過したらLED4の点灯を停止し, 計時終了モードに移行する
 - 計時終了モード
 - 特になし

Step6 : プログラム例 状態遷移判定の詳細

- 状態遷移と共に変数を設定する

```
if (Event & EVT_TIMER_START) {    /* 開始 */
    timer_mode = ACT_TIMER_MODE;
    current_time = 0;
    max_time = INIT_TIME*1000;
}
if (Event & EVT_TIMER_STOP) {     /* 停止 */
    timer_mode = STOP_TIMER_MODE;
    alarm = 0;
}
if (Event & EVT_TIMER_COUNT) {    /* 時間アップ */
    if (timer_mode == ACT_TIMER_MODE) {
        max_time += TIME_UNIT*1000;
    }
}
```

Step6 : プログラム例 各状態の詳細

- 計時中モード

```
case ACT_TIMER_MODE:
    if (current_time >= max_time) {
        /* タイムアウト */
        alarm = TIMEOUT_BLINK*TO_SEC;
        timer_mode = TIMEOUT_MODE;
        current_time = 0;
    } else if ((current_time % (ACT_INTERVAL*1000)) == 0) {
        /* 動作通知 */
        alarm = ACT_ON_TIME*TO_SEC;
    }
    current_time += TIMER_GRANULE; /* タイムアップ */
    break;
```

Step6 : プログラム例 各状態の詳細

- タイムアウトモード

```
case TIMEOUT_MODE:
    /* 15秒間点減したら計時終了モードに */
    if (current_time >= ((TIMEOUT_BLINK * TO_SEC)
                        * TIMER_GRANULE)) {
        timer_mode = STOP_TIMER_MODE;
    }
    current_time += TIMER_GRANULE; /* タイムアップ */
    break;
```

謝辞

本教材の開発において、NPO法人組込みソフトウェア管理者・技術者育成研究会およびNPO法人TOPPERSプロジェクトの作成した以下の教材を参考としました。

- OpenSESSAME Seminar組込みソフトウェア技術者・管理者向けセミナー初級者向けテキスト第5版
- TOPPERS初級実装セミナー教材

両団体および上記教材の作者の皆様へ感謝します。

本教材の開発において、NEXCESS初級コースおよび指導者養成コースの受講者の皆様のコメントを参考としました。
両コースの受講者の皆様へ感謝します。

著者リスト

- メモリマップドレジスタの操作方法の確認
 - 本田 晋也(名古屋大学)
- ポーリングプログラミング
 - 本田 晋也(名古屋大学)
- 割込みプログラミング
 - 本田 晋也(名古屋大学)
- おまけ:カップラーメンタイマの作成
 - 本田 晋也(名古屋大学)

教材に関するご意見は,
nexcess-contents@nces.is.nagoya-u.ac.jp
までお寄せください.