

TOPPERS基礎実装セミナー

(M16C版:基本)
基礎編:1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)1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

■ 1日目

- | | |
|--------------------|-------|
| 1. はじめに | 0.5時間 |
| 2. 組込みハードウェアの基礎知識 | 2.0時間 |
| 3. 組込みプログラム開発の基礎知識 | 2.0時間 |
| 4. マイコンボードの確認 | 0.5時間 |
| 5. 開発環境の確認 | 1.0時間 |
| 6. まとめ | 0.2時間 |

はじめに

パソコン型ソフトウェア開発

- 日本と米国のソフトウェア人口

	日本		米国
人口	1	:	2
ソフトウェア開発従事者	1	:	5

- なぜ、5倍のソフトウェア開発者が必要なのだろう

米国のソフトウェア開発を「パソコン型ソフトウェア開発」と呼ぶと以下のような構造となる。しかし、日本はアプリケーション開発以外は、基本設計・基本開発は行っていない。



2008/11/27

TOPPERSプロジェクト認定

6



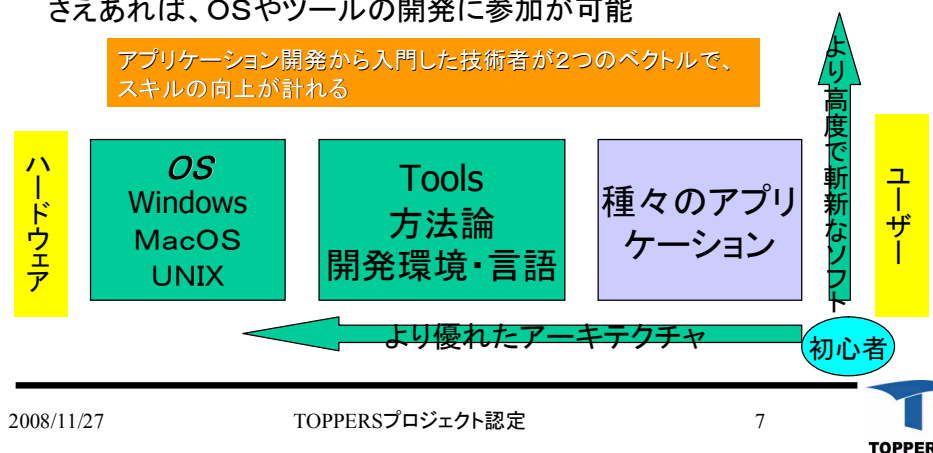
米国エバンス・データ調査によれば2007年における全世界でのソフトウェア開発者人口は1,450万人とされる。このうち米国の開発者シェアは23%で、約333万人です。日本を含むAPAC(アジア太平洋)地域は37%であり、536万人となっています。

日本のソフトウェア人口は、2004年に経済産業省が発表した「組込みソフトウェア産業実態調査」によれば、情報サービス産業で57万人、組込みソフトウェア産業で15万人とされています。合計は72万人で、調査年度は異なるが、日本と米国のソフトウェア開発人口は1対4.6となります。

実際、ソフトウェア開発者がどのような分野の開発を行っているかの資料はありませんが、日本のソフトウェア開発者のうち、OSの開発を行っているのは組込み開発者の一部、ツールの開発を行っているのも、情報サービス及び組込みソフトウェア開発者の極一部を推測される。それに比べ、主要な汎用OS、開発言語、オブジェクト指向などの方法論は米国主体で開発を行っている。

よくできたビジネスモデル・開発モデル

- パソコン型ソフトウェア開発は、セルフ開発環境であるため、パソコン環境でアプリケーションの開発が可能
ソフトウェア開発者は、より高度な機能、斬新なアプリケーションを実機上で開発でき、スキルを向上可能である
プラットフォームの改善や新しいToolの開発が必要な場合は、能力さえあれば、OSやツールの開発に参加が可能



パソコンのシステムの特徴はアプリケーションの実行環境であると共に、ソフトウェア開発の開発環境であるという点にあります。**UNIX**環境では**GNU**開発環境が同梱されており、**C**言語を知っていれば簡単にアプリケーションの開発を行えます。より高度なアプリケーションを簡単に開発するために開発ツールや開発環境を拡張していくことが可能です。

パソコン上の開発は、より高度で斬新なアプリケーションを開発していく方向と、新しい開発ツールや開発環境、画像やグラフィックや音楽やデータをより高度に簡単に扱えるようにプラットフォーム(**OS**)を改善していく方向があります。

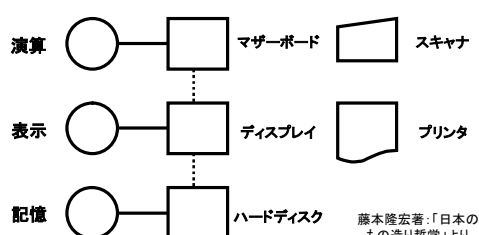
ここで問題となるのは、**CPU**上にきちんとしたシステム設計を行うには、**OS**等のプラットフォームの開発技術、開発環境に関する技術、アプリケーションの開発技術が揃ってはじめて可能となるという点です。組込みシステムという分野を考えるとバランスのとれた開発技術が必要となります。

パソコンは組み合わせ(モジュラー)型製品の代表

- 製品アーキテクチャは組み合わせ(モジュラー)型と摺り合わせ(インテグラル)型があると言われています
- パソコンはいろいろな部品を組み合わせることにより、製品として成り立つ組み合わせ型製品の代表と言われています
- しかし、パソコンの部品は摺り合わせ型、制御ソフトウェアは「パソコン型ソフトウェア」では開発できません

高機能化しなければ
ビジネスモデルが
成り立たない

➡ 部品自体がコストアップするため



パソコンのアプリは常に高速化・高機能化。より自由に開発できるように大容量のメモリ・ストレージを求めます

パソコンは常に高機能化しなければならない

2008/11/27

TOPPERSプロジェクト認定

8



それでは、全てがパソコン型のソフトウェア開発で開発できれば問題がないと思われそうですが、現実はその都合よく行きません。

製品のアーキテクチャは、組み合わせ(モジュラー)型と摺り合わせ(インテグラル)型があるといわれています。パソコンはいろんな部品を組み合わせることにより、製品として成り立つ組み合わせ型の製品の代表と言われています。しかし、パソコンの部品は、より安価で高性能のものを使うことにより、パソコン自体の商品価値は向上します。ハードディスクやプリンタにパソコンと同じインテルのCPUを使えば、コストアップとなり、商品として成り立ちません。そればかりか、より摺り合わせ型で開発することにより、性能を上げ、コストを下げるのが課題となってきます。組込みには組込み専用の開発技術が必要なのです。

組込み機器・組込みシステム

- 組込みシステムとは、各種の機器に組みこまれて、それを制御する組込みシステムのこと(一般のパソコン、コンピュータは、この範疇に含まれない)
 ➡ 明確な規定はない
- 組込みシステムは、機器の一部とみなされる

組込みシステムの範囲はさまざま

- パソコン、汎用コンピュータなどのコンピュータの形をしているもの以外は組込みシステム
- 狭い意味での組込みシステム(外見がコンピュータでないもの)をdeeply embedded systemと呼ぶ場合もある

ここで組込み機器、組込みシステムの一般的な定義について説明を行います。

組込みシステムとは、各種の機器に組み込まれて、それを制御するシステムのこと、一般のパソコンや、コンピュータはこの範疇に含まれていませんが、組込み機器の中にはパソコンやコンピュータ用の汎用OSを用いて制御を行っているものもあります。組込みシステムは、機器の一部をみなされます。

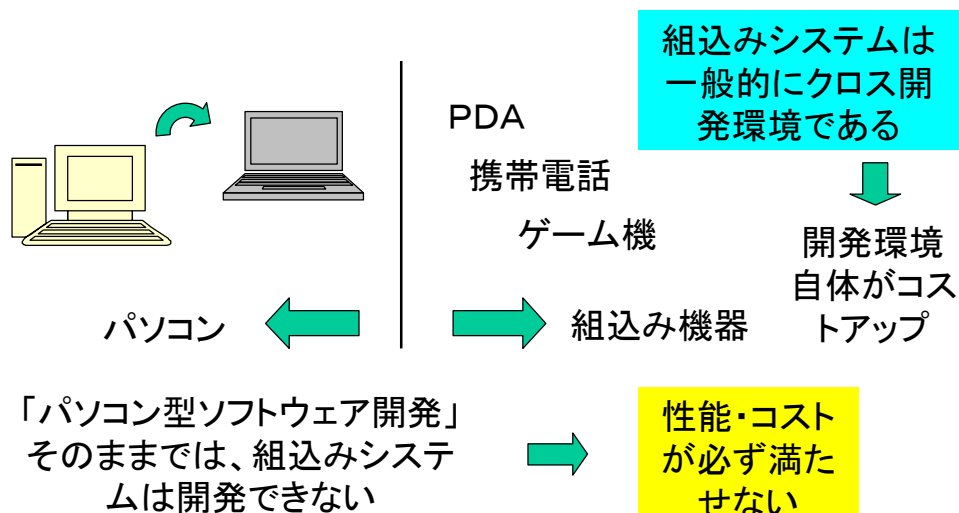
組込みシステムの範囲はさまざま、パソコンや汎用コンピュータなどのコンピュータの形をしているもの以外を組込みシステムと呼びます。このように狭い意味での組込みシステムをdeeply embedded systemと呼ぶ場合もあります。

組込みシステムの適応機器の例

- 家電機器(電子レンジ、炊飯器、乾燥機、エアコン)
- AV機器(テレビ、ビデオ、デジタルビデオ、オーディオ機器)
- 娯楽／教育機器(ゲーム機、電子楽器、カラオケ、パチンコ)
- 個人用情報機器(PDA、電子手帳、カーナビ)
- パソコン周辺機器(プリンタ、スキャナ、ディスク、DVDドライブ)
- OA機器(コピー、FAX)
- 通信機器 端末(電話機、留守番電話、携帯電話)
- ネットワーク設備(交換機、PBX、ネットワークルータ、HUB)
- 運輸機器(自動車、信号機、鉄道車両／制御、航空機、船舶)
- 工業制御／FA機器(プラント機器、工作機器、工業用ロボット)
- 設備機器(ビル用照明、ビル用空調、ビル用電力システム、エレベータ)
- 医療機器／福祉機器(血圧計、心電計、レントゲン、CTスキャン)
- 宇宙／軍事(ロケット、人工衛星、ミサイル)
- その他の事務用機器(事務用データ端末、POS端末、自動販売機)
- その他の計測機器(シンクロスコープ、ICテスタ、電子メータ)

パソコンと目的がかぶる組み込み機器

- パソコンのアーキテクチャのまま、情報機器化するとモバイルまで、コストがアップし、性能は落ちる



2008/11/27

TOPPERSプロジェクト認定

11



組み込み機器の中には、パソコンやコンピュータと目的がかぶるものが多くあります。例えば、PDA、携帯電話、ゲーム機などは、パソコンやコンピュータと同等の機能を持ちます。この場合の違いは何なのでしょうか？

多くの組み込み機器の特徴は、クロス開発環境にあります。組込システムは機器の一部となるため、開発環境もパワーやメモリや十分なストレージのあるパソコンやコンピュータに任せて、制御に必要なCPUパワーやメモリしか持たないケースが多いのです。この辺にパソコンやコンピュータと組み込みシステムを分ける境界があるといえます。

クロス開発を行う組み込みシステムでは、セルフ開発システムよりも、ハードウェアやリアルタイムOSやデバイスドライバの取り扱いに関して、多くの技術やノウハウやよりデリケートな開発が必要になります。

規模からみた組込みシステムの分類

- 組込み機器の規模から3つの組込みシステムに分類する

(1) 小規模組込みシステム

ワンチップCPUで制御を行う小規模な組込みシステム、ROMやRAMを拡張することもある。RTOSを用いないシステムが多い。

(2) 中規模組込みシステム

32ビットクラスのCPUを使うシステム、ソフトウェア規模も大きくRTOSを用いてシステムの制御を行う

(3) 大規模組込みシステム

パソコンに近い規模の組込みシステム、実際にパソコンやワークステーションを制御用に用いる場合もある。

2008/11/27

TOPPERSプロジェクト認定

12



組込みシステムの教育を行っていく上で、開発規模から組込みシステムを3つのシステムに分類します。

(1) 小規模組込みシステム

ワンチップCPUで制御を行う小規模な組込みシステム、プログラム規模は大きくても1万から2万行でくいで、リアルタイムOSを用いないシステムも多い、CPUメーカから供給されたツールやICE等の開発機器を用いて開発を行う。ソフトウェア開発者は1か2人くらい

(2) 中規模組込みシステム

32ビットクラスのCPUを使うシステム、ソフトウェア規模も大きく、ソフトウェアの開発は分業で行うためリアルタイムOSを用いてモジュール開発を行う。

(3) 大規模組込みシステム

パソコンやコンピュータの専用システムに近い組込みシステム、ソフトウェアステップ数も100万行を超えるもの、実際にembedded LINUXや汎用のOSをプラットフォームに使用し、メモリプロテクションでアプリケーションを保護するようなシステム。パソコンやワークステーションをそのまま制御機器として用いる場合もある。

小規模組込みシステムの技術

- クロス開発となるため、開発環境を用意しないと開発自体ができない・・・(1)
- 電源オンからアプリケーションプログラムが実行するまでのプログラムを自作する必要がある。割込み等も自分で実装(パソコンではOSが代行してくれる)
- 簡単なプログラムミスでハングアップ(機器が未応答になる状態)や暴走が発生する。これを解決するためにICE等の特別な開発ツールを用いてプログラムの実行を監視したり、地道にチェックポイントにLEDを点ける等のデバッグテクニックが必要。・・・(2)
- デバイス等がインテリジェント化し、内容を理解してデバイスドライバを作成しなければならない。開発にはハードウェアとソフトウェアの両方のスキルが必要。・・・(3)
家電機器、簡単なパソコン周辺機器、電子おもちゃ等

2008/11/27

TOPPERSプロジェクト認定

13



小規模組込みシステムはワンチップCPUの使用を対象とする小規模な組込みシステムを指します。プログラム規模は一万行程度でまで、ひとりまたは二人程度で開発が可能なプログラミング規模のシステムです。小規模組込みシステムには以下のような特徴があります。

(1) クロス開発となるため、開発に対応した開発環境のセットアップが必要となります。開発環境はCPUメーカーや半導体メーカーから供給される場合が多く、対応のCPUに特化した開発環境が用意されています。しかし、メーカーによって環境が異なるために、CPUが変われば、新しく開発環境の勉強が必要です。これは組込みシステム開発の大きな特徴①です。

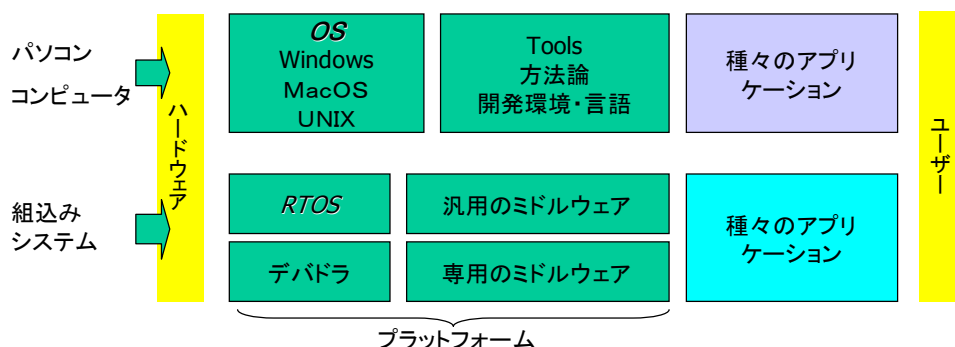
(2) 基本的に電源オンからアプリケーションプログラムが実行するまでのプログラムを自作する必要があります。実際は、もともとあるシステムの作り変えから入るケースや、デバイスドライバ等のサンプルプログラムの参照ができる場合も多く、すべてを自作するという意味ではありません。しかし、パソコン上のアプリケーション開発のように、書物やオンラインマニュアルがそろっているような開発環境ではなく、苦労して、試行錯誤で開発を行うケースも多いのが事実です。

(3) 簡単なプログラムミスで、ハングアップ状態に陥ってしまうケースも多々あります。特に、ICEなどのようなデバッグツールがない場合、どのような状態でハングアップしているのかの判定がでない場合が多い。たとえば、プログラムの実行途中でエクセプションが発生する障害の場合、ベクトルが正しく設定されていれば、リポートを繰り返すし、設定されていなければ、ROM領域以外を実行し永久にループに入ったり、種々の現象を引き起こす。パソコンでのプログラムの実行はアプリケーションの異常動作はOSが検知しプログラムを停止してくれるため、問題解決しやすいが、組込み開発では問題開発に多くの時間が必要となるケースが多い。これは組込み開発での開発上の問題であるため特徴②とします。

(4) USB等の通信デバイスはインテリジェント化し、LSI内のステートの変化等の管理を行ったり、EEPROM等も、ピン数を節約するため高速なシリアルデバイスに接続する形態となったたり、高速、安価にはなるが、複雑な制御が必要となっています。エレクトロニクス設計者はLSIの結線やノイズ等のスキルはあっても、複雑なLSI制御の知識に乏しいケースが多い。LSIを上手に制御するためには、エレクトロニクスのスキルとソフトウェアのスキルを兼ね備えた技術者が必要となってきます。このような技術者は育ちにくく、不足しがちです。パソコンでは、デバイスドライバの供給者は半導体メーカーであり、組込み開発のようにセットトップメーカーの組込み技術者が半導体メーカーのLSI制御プログラムを作成するケースは少ない。この問題も組込み開発の特徴③です。

中規模組込みシステムの技術(1)

- パソコンほどソフトウェア規模はないが、全てのソフトウェア開発、品質保証を行わなければならない
- コンピュータシステムのOS、Toolにあたる部分がプラットフォームと呼ばれ、設計の良し悪しが組込み機器の品質・性能・コストに大きく影響する。設計は企業秘密にあたり公開はされない・・・(4)



2008/11/27

TOPPERSプロジェクト認定

14



ある程度規模の大きなシステムを開発する場合、小規模組込み開発で開発した小さなボードをLAN等の通信を用いてネットワーク化し、多機能を実現する場合と、より高速なCPUを用いて多くの機能を同時に実行する場合があります。どちらのシステムが最適なのかは、その組込みシステムのもつ非機能要件に従います。小さなボードをネットワーク化するシステムは、小規模システムの延長上といえますので、ここでは言及しません。中規模組込み開発は、小規模組込みシステムに比べ、より高速なCPUを用いて多くの機能を同時に実行するようなシステムでプログラムサイズが2万行以上となり、分業化しプロジェクトとして組込み開発を行うことを特徴とします。分業化されたシステムを上手に開発するためにリアルタイムOSを用いて開発を行うことも大きな特徴です。

パソコン用のOSの場合、人間が通常行う業務やエンターテインメントを実現するためオペレーティングシステムです。組込みシステムの場合、機器の中に組み込まれるシステムであり、OSとして最適な形態は機器によって異なります。そのため、リアルタイムOSというタスク処理とシステム管理に限定したオペレーティングシステムをベースとしています。当然、機器の構成によってはパソコン用のOSのように、拡張した機能をオペレーティングシステム側で用意した方が、生産性や品質アップにつながります。組込み開発ではリアルタイムOSを拡張し、種々の機能を追加したシステムをプラットフォームと呼びます。プラットフォームは組込みシステムが多くするように、種々のものが多くあります。プラットフォームは組込み機器の中のベースとなるソフトウェアを指すものです。組込み機器は、商品であり、プラットフォーム自体も公開されるものではありません。プラットフォームの品質や設計が、商品の機能や品質に直結するのも事実です。

プラットフォームの開発には、高いスキルが必要です。一般的なソフトウェアの書物や技術文書は垂ぶろケーションの開発に関するものが多く、また、リアルタイムOSの書籍やセミナーはあっても、プラットフォームの開発手法や技術は明文化されておらず、「企業秘密」となっているケースが多いのは事実であり、このような事情から明文化は難しい状況です。これも組込みシステムの特徴④です。

中規模組込みシステムの技術(2)

- 専用ミドルウェアは機種独自の機能性能を達成するソフトウェアである。ハードウェアと協業して機能を達成する場合が多い。企業秘密にあたる部分で一般には公開されない・・・(5)
- 汎用ミドルウェアはプロトコルスタックやファイルシステムのような社外から導入するソフトウェアである。機能を取り込んだり、評価を行うために、やはり、ノウハウが必要。テストや最悪市場障害発生時等、社内の対応ができるようにする必要がある。

プリンタシステムの専用ミドルウェアは、画像処理やデータレンダリングや変換処理を行うプログラムが、これにあたる。

ゲーム機、プリンタやFAX、カーナビ、デジカメ等

2008/11/27

TOPPERSプロジェクト認定

15



中規模組込みシステムの、もうひとつの特徴はミドルウェアと呼ばれるソフトウェア部品を使用する点です。ミドルウェアには、市場に流通している汎用ミドルウェアと、その組込みシステムが独自にもつ、専用ミドルウェアがあります。ソフトウェア部品といっても、商品として品質を保つために、品質保証する必要があります。

また、中規模組込みシステムでは、市販のエバレーションボードがそのまま、商品として使用されることは、まずありません。汎用のミドルウェアでも、ハードウェアが変わったり、リアルタイムOSが変更となった場合は、プログラムの修正は発生しますので管理と運用が必要です。汎用のミドルウェアといっても、それを管理、運用する技術者を、社内、社外の協力会社に用意する必要があります。

専用のミドルウェアの方ですが、これもまた、厄介な問題があります。このミドルウェアは、その機器特有の機能を実現するためのミドルウェアといえます。例えば、ファクシミリは電話回線上にイメージデータの転送を行います。当然、データ量を少なくするためにデータ圧縮を行います。このデータ圧縮・伸張を行うモジュールがファクシミリの専用ミドルウェアのひとつです。初期のファクシミリでは、ハードウェアロジックのこの機能を行っていました。しかし、現在のファクシミリではCPUの高速化により、ソフトウェアで行った方が安価にこの機能を実現できます。また、用紙の四隅は印刷できないため、圧縮伸張に加え変倍機能が必要となったりします。専用ミドルウェアは非機能要件により、いろいろなバリエーションをアプリケーションに同一のインターフェイスで供給できるように設計しなければなりません。また、ロジックでもなく、通常のソフトウェアでもなく、DSP化される場合もあり、マルチCPU化される場合もあります。

このようなミドルウェアを上手に管理、制御する特徴⑤も、組込みシステムの特徴のひとつです。

大規模組込みシステムの技術

- パソコンやワークステーションのシステムをそのまま使用する組込みシステム
プラント制御、航空機、銀行ATM等
- 機器の規模が大きくなればなるほど、パソコン型ソフトウェア開発を、そのまま使用したほうが開発効率が良い、それでも、機器の非機能要件によっては、中規模組込みシステムの延長のアーキテクチャを採用するものも多い。
携帯電話等
- 組込みLinuxは中間型と言える
- PlayStation3は低価格のパソコンと同じ機能である。
ゲームプログラムを実行できるLinuxシステムにした方が、多機能化が簡単にできるのでは？

2008/11/27

TOPPERSプロジェクト認定

16



100万行を超えるようなソフトウェアシステムの場合、開発環境や安全性の問題で、通常のコンピュータシステムをベースに組込みシステムを構築した方が優位場合があります。大規模組込みシステムは通常のコンピュータをベースにした組込みシステムを指します。上記の記述のように、このような組込みシステムは多々ありますが、通常コンピュータの開発説明等は別途、技術化されていますので、ここでは教育の対象とはしません。

組込み開発形態からまとめスキルと問題点

- ・ パソコン型ソフトウェア開発の熟練したアプリ開発者でも、組込みシステムを円滑に開発するのは難しい
理由は(1)～(5)までにある

- (1)クロス開発であるために環境を整えないと開発体験ができない・・・本基礎セミナーの目標
- (2)パソコン型ソフトウェア開発に比べ、安全性の面でアプリケーションの開発が難しい
- (3)ソフトウェアとハードウェアの両方のスキルをもった技術者が必要
- (4)適切なプラットフォームの開発には熟練と経験が必要
- (5)企業秘密に属する技術は企業に入らないと教育ができない

2008/11/27

TOPPERSプロジェクト認定

17



組込みシステムの特徴を5点挙げます。

(1)クロス開発であるため、開発環境を整えないと開発ができません。パソコン上のソフトウェア開発のように、開発プログラムをインストールして、即開発というわけにはいきません。特に、実行確認を行うために開発ボードが必要となり、その設定にノウハウは必要です。

(2)パソコン上のソフトウェア開発は、プロセスというメモリ保護された実行環境しますので、おかしい挙動を行っても、OSがチェックしプログラムを安全に停止してくれます。開発環境には、問題点をトレースする機能があり安全に実行ができます。しかし、組込みシステムでは、制約されたボード上でプログラムを実行確認する必要あり、挙動がおかしいプログラムを実行すると、ハングアップ状態になる場合も多く、プログラムの開発にはストレスが伴います。

(3)デバイスは高速化、高機能化のために、複雑な機能を持つようになっています。この結果、汎用のデバイスドライバの作成にはハードウェアのスキルとソフトウェア開発のスキルが必要となっています。半導体技術は常に進化しており、それに対応した技術蓄積が必要です。

(4)中規模組込みシステムは、機器の機能を適切に実行するために管理システムを最適化する必要があります。この管理システムはプラットフォームと呼ばれ、機器に特化した汎用OSというべきものです。プラットフォーム開発の技術は企業秘密に属するもので、明文化されていません。そのため、現状ではプラットフォームの開発は、熟練と経験が必要となります。

(5)組込み機器は企業競争の産物であり、高商品価値や高品質を維持することが重要な目標となります。そのため、企業秘密に属する重要な技術は教育目的でも、一般に公開されることはなく、企業の中でOJT等の形で教育が行われることが多く、人材開発に大きな問題があります。

本セミナーでは、(1)、(2)の教育と問題解決を目標にしています。

組込みソフトウェア教育の問題点

- 前述の通り、パソコン型ソフトウェア開発の教育を、そのまま組込みソフトウェア教育に適用しようとしても、必ず、不足が発生する
- パソコンのプラットフォームにあたる部分の開発を日本では行っていないため技術補完がされない。また、学校等の長期のカリキュラムでの教育が必要である。
- 教育の形で解決できない技術取得があり、職場の先輩等のコミュニケーションがその代用を果たすケースが多い(新人技術者の一番重要なスキルかもしれない)。また、企業の組織自体も、この点を考慮しない組織を構築すると円滑に機能しない場合がある。(パソコン型ソフトウェア開発は考慮しなくても開発が行えるプロセスを持つ)

2008/11/27

TOPPERSプロジェクト認定

18



前述の通り、組込み技術はパソコンでのプログラム開発と異なった側面があるため、組込み技術者育成には種々の問題があります。

大きな問題のひとつは、現状のソリューション開発中心のソフトウェア教育が、組込みソフトウェア技術者の教育には適応しないという点です。これはビジネス規模からいって仕方がない問題かもしれませんが、組込みシステムの成り立ち自体コスト追及型であるため、難しい問題かもしれませんが、きつい、厳しい職種です。しかし、優秀な開発チームが、組込み技術に熟練しない多人数の開発チームより、商品性や品質に優れた組込み開発を行えるのは事実であり、筆者も、開発者により一ヶ月も問題解決ができなかった不具合が、開発者の変更で2, 3日で解決する等の現状を多々目撃しています。これはスキルの問題かもしれませんが、組込み技術の適切な教育が行われていれば、このような問題は自然に解消されるのではないのでしょうか。

もうひとつの大きな問題は、組込み技術は企業内技術であるという点です。企業での教育は商品開発の手段であり、実際の教育は開発設計行為のかなで行われたり、OJTの形で培われます。企業活動といいつつ、職人的な色彩が強いものです。重要な点は、マニュアル化やプロセス化だけの側面では解決が難しいという点です。逆にこの問題をうまく解決した企業が、組込み技術に卓越した企業といえるのかもしれませんが。

基礎1コースの概要

- 小規模組込みシステム開発の基礎知識を習得、クロス開発の体験をM16Cを用いて行う
- M16Cをベースした組込みハードウェアの基礎知識の習得
- 一般的な組込みソフトウェア開発の基礎知識の習得
- M16Cを使用したHSB16C29S64ボードと開発に必要な機器の確認
- M16Cの開発環境の確認
開発環境のインストールからボード上での実行確認まで
- 2日目は、実際のアプリケーションプログラムの開発を行います
- 3日目は、RTOS-ITRONの基礎知識の取得とTOPPERS/JSPの簡単な導入をおこないます

2008/11/27

TOPPERSプロジェクト認定

19



基礎1コースは小規模組込みシステムの開発技術に特化した技術教育を目的としています。しかし、中規模組込みシステムのシステム管理者にも必要な技術です。基礎1コースはルネサステクノロジ製のCPU:M16C29をコアとした(株)北斗電子のHSB16C29S64ボードを対象にセミナー教育を行います。M16Cの開発環境は非常に優れた環境であり、統合環境とリモートデバッガとの連携は、他の開発環境と比べて揃ったものです。M16C以外に実際に市場で多く使われているCPUはありますが、組込みシステムの特徴であるクロス開発を、組込み初心者簡単に体験して頂くには最適なのでこれを選択しました。他のCPUを使用して開発を行う場合でも、基本的には、この環境と同じです。

組み込みハードウェアの基礎知識

1. [コンピュータの構造](#)
2. バスとメモリ
3. 周辺デバイス
4. 外部事象の待ち方

組込みソフトウェア開発におけるハードウェア知識

組込みソフトウェア開発には、
ハードウェア(コンピュータシステム)の知識が必要

ハードウェアに密着したプログラミング

- 組込みシステムはシステムを制御することが目的
 ハードウェアを直接扱うプログラミングが必要

多様なプラットフォーム(アーキテクチャ)

- ハードウェアが応用毎に最適化されて設計される
 質の高いプログラムを開発するためには、ハードウェアアーキテクチャの理解が必要

組込みソフトウェア特有の技術

- 汎用コンピュータ向け技術とは異なる技術が多く使われる
 - ・ 低消費電力、デバッグ機能、高信頼性、etc...

2008/11/27

TOPPERSプロジェクト認定

21

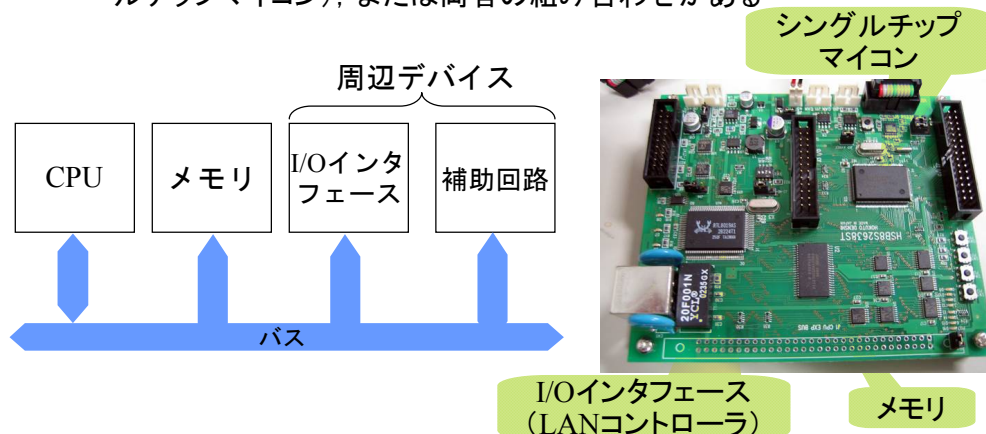


組込みシステムの開発では、制御ボード自体の開発も含まれます。ソフトウェアの開発者がハードウェアの設計者である場合は問題はないのですが、ハードウェアの設計者とファームウェアの設計者が別の場合は、当然、ファームウェアの設計者にハードウェア知識が必要となります。中規模以上の組込みシステムでは、一部のシステム設計者が有識者であれば、開発自体は可能ですが、ソフトウェアの開発規模が小さい小規模組込みシステムでは必須となります。たとえ、ハードウェアの設計制作やファームウェア部の設計の外部に委託したとしても、なにか問題が発生した場合、その解決にはハードウェアを含んだ知識が必要となります。以下のまとめると

- 1) 組込みシステムはハードウェア(エレキ、メカ)を直接制御することを目的としたシステムであり、ハードウェアを直接制御するプログラムが必要となります。(このようなハードウェアを直接制御するプログラムをファームウェア(firmware)といいます)
- 2) 組込みシステムは多様なシステムです。それを制御するハードウェアも機器ごとに最適化します。このような特異なハードウェアに最適化したプログラムを開発設計するには、ハードウェアアーキテクチャの理解が必要となります。これはハードウェアの知識さえあれば良いという意味ではありません。特にソフトウェア規模が大きい場合、ファームウェアはハードウェアにも、その他のソフトウェアにも最適に設計する必要があります。
- 3) パソコンのような汎用コンピュータ用のシステムでは、アプリケーションとハードウェアの間にオペレーティングシステムや種々のインターフェイスプログラムが介在し、手順さえわかれば(これが難しいという話もある)直接ハードウェアを制御する必要はありません。組込みシステムの開発はアプリケーションからこのようなハードウェア制御プログラムまで全てのソフトウェアを対象とした開発といえます。逆の発想として、パソコンと同じCPUやボードを使って、パソコンと同じOSを使用すれば、組込みソフトウェアは簡単になるという意見がありますが、パソコンの部品やパーツが組込み機器で構成されていることを考えれば、そのような組込みシステムには意味はないことは、すぐに理解できることだと思います。

コンピュータの構造

- プロセッサ(CPU)を中心に、メモリやI/Oインタフェースなどがバスにより接続されている
- それぞれの要素が独立したLSIで実現されており、基板上で組み合わされている場合や、1つのLSI上のみで構成されている場合(シングルチップマイコン)、または両者の組み合わせがある



2008/11/27

TOPPERSプロジェクト認定

22



現在の技術では、規模の大小はあれコンピュータの構成はどれも同じものです。コンピュータを構成する部品はどれも共通です。組込みに使用されるコンピュータはマイクロコンピュータ(マイコン)と呼ばれます。これは1960年代にLSI技術の進化によりコンピュータをLSI化が実現でき、これをマイクロコンピュータと呼んだことに起因しています。マイクロコンピュータは以下の部品により構成されています。部品はLSI化されていますので、基盤上で組み合わされて構成されている場合や、ひとつLSI上で、ほとんど全てを組み入れている場合もあります。最初のマイコンは電卓に使用されました。このとき、マイコンとメモリ等の部品は別々のLSIで構成されましたが、現在の電卓はほとんどすべてのコンピュータ部をひとつのLSIで実装されています。

- 1) プロセッサ (CPU)
- 2) メモリ (読み書きできるRAMや読み込み専用のROM等の種別がある)
- 3) バス
- 4) I/Oインターフェイス

プロセッサ

プログラム(ソフトウェア)を実行するハードウェア

基本動作

- プログラムはメモリ上に置かれる
- プロセッサはメモリから命令を読み込み(fetch), 解釈し(decode), 実行する(execute)

用語

- マイクロプロセッサ, プロセッサコア, CPU(Central Processing Unit), MPU(Micro Processing Unit), MCU(Micro Controller Unit), シングルチップマイコン

2008/11/27

TOPPERSプロジェクト認定

23



プロセッサ(CPU: 中央演算処理装置)はコンピュータ全体を制御する部品です。マイクロコンピュータ用のCPUは一般的に以下の機能を行います。

- 1) 時間を軸として、各単位時間(ステートと呼ぶ)ごとに行う作業を定めます。
- 2) メモリから命令を読み出し、解析を行います。
- 3) データを読み出したり、格納を行ったりします。
- 4) データの演算を行います。
- 5) I/O部に対して、コンピュータのどの部分から命令やデータの出し入れを行うかの指示を行います。
- 6) 外部から実行を制御したい場合、その指示に従い動作します。(割込みやCPU例外やホールド等)

組込みシステム用プロセッサの多様性

パソコンの現状

- IA-32 (x86, Pentium, Athlon) が主に使われている
 - MACもPowerPCからPentiumへ

組込みの現状

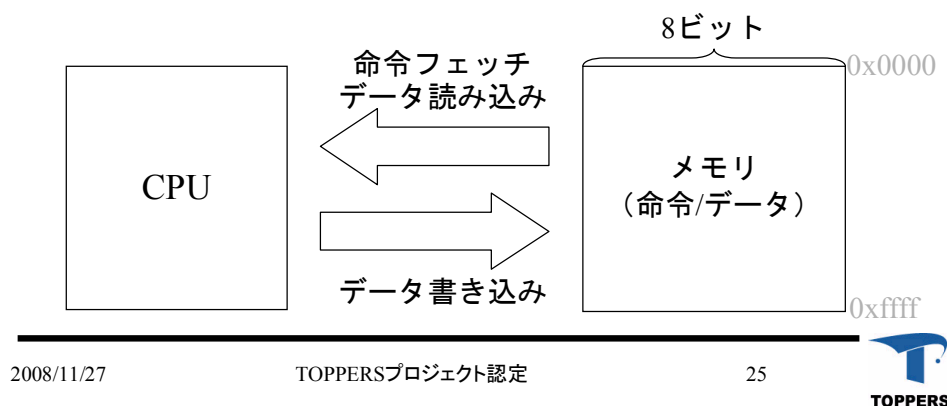
- 多種多様なアーキテクチャのプロセッサが使われてる
 - ARM系, MIPS系, PowerPC系,
 - SH系, V850系, H8系, M16C系, M32R/C系
 - DSP, コンフィギュラブルプロセッサ, ソフトコア
- アーキテクチャ毎に命令セットやレジスタの構成やサイズが異なる

パソコン用のプロセッサはIA-32 (Intel Architecture 32: Intel社の32ビットマイクロプロセッサで用いられるマイクロコンピュータアーキテクチャの総称) のプロセッサが主流です。組込み開発では多種多様なアーキテクチャのプロセッサが使用されます。これらのプロセッサはアーキテクチャ毎に命令セットやレジスタの構成やサイズ異なります。1973年に開発されたC言語が、このプロセッサに安全に使えるようになった1990年ごろまでは、個々の組込み機器はアセンブラ言語(機械語)で開発され、大変な開発負荷がありました。現在でも、コンピュータにはC言語では記述できないソフトウェア部分があります。

プロセッサ：メモリ & アドレス

メモリにはアドレス(番地)が割付けられている

- ・ プロセッサはアドレスを指定してメモリを読み書きする
- ・ プロセッサの命令とデータは、メモリ上に置かれている。
- ・ バイトアドレッシングが一般的
 - 1バイト(8ビット)単位でアドレスが割り当てられている



2008/11/27

TOPPERSプロジェクト認定

25



プロセッサの処理を正しく理解するためにはステートのついて説明を行う必要があります。プロセッサもデジタル回路であるため、クロックという周期単位の信号が必要となります。簡単なプロセッサではこのクロックの周期をステートと呼び、プロセッサが仕事を行う最小単位がステートとなります。しかし、メモリからデータを読み出す等の処理はステート単位では短すぎるので、複数のステートで一連の仕事を行う単位をつくり仕事を行います。この単位をマシンサイクルと言います。プロセッサはマシンサイクル毎に時分割に仕事が行えるように設計されています。

ここでプロセッサとメモリの関係について説明を行います。例えば、プロセッサがメモリから命令を読み込む場合、初期のマイクロプロセッサは1マシンサイクルで取り込みと解析を行いました。命令を取り込むためには、メモリに命令の格納されている番地をバスを通して指定し、その番地のデータをメモリがバスにのせ、プロセッサがバス上のデータを取り込むという一連の作業が必要となります。バス上の値はステート単位に切り替えることにより、いろいろな値がのせられます。

メモリ上のデータは一般的には8ビットをひとつのデータの塊として保管されており、このデータの塊をバイトと呼びます。すなわち、1バイトは8ビットのデータで構成されています。個々のバイトデータには番地が割り振られており、これをアドレスと呼びます。

プログラムはいくつかの命令で成り立っています。プロセッサはマシンサイクル毎に、命令を取り込み解析を行い、解析のコードに従って、データを取り込んだり、演算を行ったり等の仕事を行い、プログラムに従った仕事を行います。

プロセッサ：データ

データ命令と同様にもメモリに置く

機械語からのアクセス

- アドレスを指定して読み書きする

C言語の変数(グローバル変数)

- コンパイラがメモリ上に領域を確保する

ポインタ

- 変数が格納されているメモリのアドレス

dの値は0x61

```
char*d = &a;
```

0x61への書き込み

```
*((char*)0x61) = 1;
```

グローバル変数

C言語プログラム

```
char a;
void func(void){
    a = a - 1;
```

メモリへの配置

メモリ	アドレス
	0x60
変数a	0x61
	0x62

アドレス

コンパイル結果

```
sub.w #1, 0061h
```

2008/11/27

TOPPERSプロジェクト認定

26



プログラムで参照、更新を行うデータもメモリ上におきます。機械語からデータをアクセスする場合は、メモリ上に配置されたアドレスを指定してアクセスを行います。C言語の場合はグローバル変数と呼ばれ、コンパイラによりメモリ上に領域が割り当てられます。割り当てられたグローバル変数のアドレスはリンカによりアドレスが決定します。そのため、アセンブラ言語では直接アドレスではなく、グローバル変数を示すラベルでアドレスを指定することができます。

C言語ではグローバル変数aに対してマイナス1を行うにはM16Cの場合以下ようになります。

```
a = a - 1;
```

アセンブラ言語では、

```
mov.b _a, A0
```

```
dec.w A0
```

```
mov.b A0, _a0
```

機械語では

```
MOV.B 0x0061, A0
```

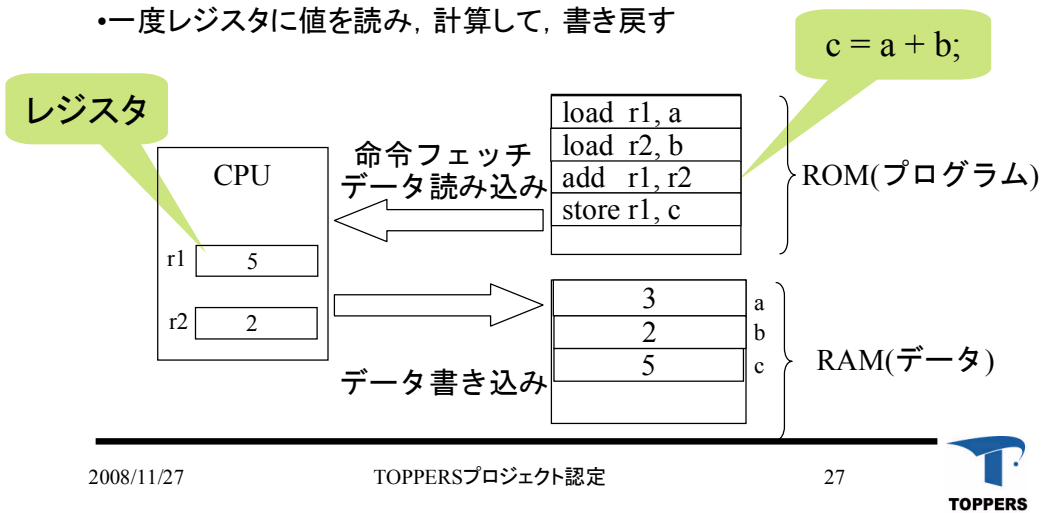
```
DEC.W A0
```

```
MOV.B A0, 0x0061
```

プロセッサ : レジスタ

プロセッサ内部でのデータの保持や、プロセッサの動作状態の保持・変更のための変数

- プログラムのデータ(変数)はメモリの中に置かれる
- プロセッサはメモリを値を直接計算できない
- 一度レジスタに値を読み、計算して、書き戻す



2008/11/27

TOPPERSプロジェクト認定

27



プロセッサは命令を円滑に行うために、メモリ上のデータやアドレスを蓄えます。また、プロセッサの状態を示す値を保持しなければなりません。このようなプロセッサ用のデータの格納場所をレジスタと呼びます。一般的なレジスタは汎用レジスタと呼ばれ、命令のアドレスやデータを保持するのに使用されます。なぜ、汎用レジスタが必要なのでしょう？もし、プロセッサがメモリ上のデータに対して、直接足し算や引き算のような演算を行うおうとすると、前節のような複数のマシンサイクルを使ってデータの取り出しや書き込みを行う必要が出てきます。レジスタとしてプロセッサの内部に持つことにより、少ないステートで演算を高速に行うことが可能になります。

プロセッサ：レジスタの種類

- アドレス、データ、演算結果を管理する汎用レジスタ
- プロセッサの役割管理を行う専用のレジスタ

(PC: プログラムカウンタ, SR: ステータスレジスタ, CR: コントロールレジスタ)

汎用レジスタ	専用のレジスタ
アキュムレータ メモリから読み込んだデータや演算結果を保持する	プログラムカウンタ プロセッサが実行する命令の場所(アドレス)を指し示す
アドレスレジスタ アキュムレータに読み込むメモリの場所(アドレス)を保持する	ステータスレジスタ プロセッサの動作状態を保持する比較演算結果, 割込み状態, 動作モード
スタックポインタ データの一時保存に使われるスタック領域の先頭を指し示す	コントロールレジスタ プロセッサの動作状態を変更するレジスタ, 動作モード, 割込み許可・禁止

2008/11/27

TOPPERSプロジェクト認定

28



レジスタは一般的には、汎用レジスタと専用のレジスタに分けられます。レジスタの構成はプロセッサのアーキテクチャにて決められますので、プロセッサの種類分のバリエーションがあります。従って、上記のレジスタの説明でも、当てはまりにくいレジスタ構成を持つものもありますので、要注意です。

代表的な汎用レジスタは以下のものがあります。

- アキュムレータ

メモリから読み込んだデータや、ALUによる演算結果を保持するためのレジスタ

- アドレスレジスタ

アキュムレータに読み込んだり、アキュムレータから書きだすためのアドレスを保持するレジスタ

- スタックポインタ

データの一時保存に使われるスタック領域の先頭を指し示すレジスタ、アドレスレジスタの一種である。

代表的な専用のレジスタは以下の通り

- プログラムカウンタ

プロセッサが実行する命令のアドレスを示すレジスタ。インストラクションレジスタ(IR)とも呼ばれる。

- ステータスレジスタ

プロセッサの動作状態を保持するレジスタ。ビット単位に意味が設定されている。

比較結果、演算結果、割込み状態、動作モードなどを示す。

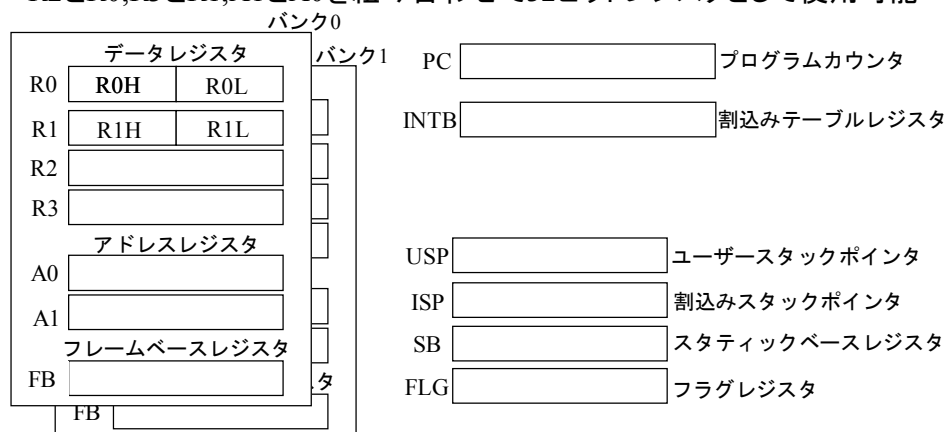
- コントロールレジスタ

プロセッサの動作状態を設定するためのレジスタ

割込み許可禁止、割り込みレベル設定、プロセッサやコプロセッサの動作モード等を行う。

プロセッサ : M16Cのレジスタセット

- 汎用レジスタは2バンク構成(フラグレジスタで切り替え可能)
- データレジスタは16ビットが基本だが, R0とR1は上位と下位を分けて, それぞれ8ビットのレジスタとして使用可能(R0L,R0H,R1L,R1H)
- R2とR0,R3とR1,A1とA0を組み合わせると32ビットレジスタとして使用可能



2008/11/27

TOPPERSプロジェクト認定

29



演習に使用するM16Cのレジスタセットについて説明を行います。

汎用レジスタは2バンク構成となっています。これは簡単な割込み処理でレジスタの値の退避時間を削減する設計が可能です。M16Cは16ビットCPUなので4つの16ビットレジスタが基本構成となっています。R0とR1は8ビットの処理ができるように、R0L, R0H, R1L, R1Hのように8ビットレジスタとして扱うことが可能です。アドレスレジスタはA0とA1の2つの16ビットアドレスレジスタが用意されています。M16Cは1MBのアドレス空間(番地)があります。16ビットアドレスレジスタでは64KBのアドレスしか設定できないため、A0とA1の2つのアドレスレジスタを組み合わせ、すべてのアドレスを設定できる機能を持っています。同様に、R2とR0, R3とR1の2つのレジスタを組み合わせると32ビットデータ保持と演算ができる機能も持っています。FBはフレームレジスタと呼ばれ、16ビット間接アドレス指定用のレジスタです。

専用のレジスタは、20ビットのプログラムカウンタ、INTBは20ビットで構成され割込みテーブルの先頭番地を指定します。スタックポインタは16ビットで、通常処理用のスタックポインタ:USPと割込み発生時のスタックポインタ:ISPの2つがあります。SB(スタックベースレジスタ)はSB相対アドレッシングに使用します。FLGは読み書きが可能なプロセッサの状態管理と設定が行えるレジスタです。データビット単位に決められています。

BIT15	予約領域
BIT14-BIT12	IPL : プロセッサの割込み優先レベル
BIT11-BIT8	予約領域
BIT7	U : スタックポインタ指定フラグ
BIT6	I : 割込み許可フラグ
BIT5	O : オーバフローフラグ
BIT4	B : レジスタバンク指定フラグ
BIT3	S : サインフラグ
BIT2	Z : ゼロフラグ
BIT1	D : デバッグフラグ
BIT0	C : キャリーフラグ

組み込みハードウェアの基礎知識

1. コンピュータの構造
2. [バスとメモリ](#)
3. 周辺デバイス
4. 外部事象の待ち方

バス : パラレルバスとシリアルバス

モジュール(LSIや機器)間でデータを
やり取りするための通信路

パラレルバス

- 複数の信号線を使ってデータをやり取りする
- SCSI, PCI, ISA, VMEBUS, AMBA...

シリアルバス

- 単一の信号線を使ってデータをやり取りする
- PCI Express, IEEE1394, CAN, USB, Ethernet...

複数の信号線のタイミングを合わせなければならないため、
経路が長くなると通信速度に限界がある。そのため、
経路が長い機器間の接続には、シリアルバスが使われる。

バス(bus)はコンピュータの内外でデータを交換するための通信路です。配線方式により、パラレルバスとシリアルバスに分類されます。

複数の信号線(基本的には8ビット、16ビット、32ビット)を使ってデータのやり取りを行うバスをパラレルバスといいます。パラレルバスはパソコンの創世記には標準的なバス方式でしたが、データ転送時、複数の信号線のタイミングを取るのが難しく(同期と干渉の問題)高速化できない問題があり、シリアルバス化が進んでいます。シリアルバスは1本の信号線で1ビット単位に順番にデータを転送するバスです。パソコンの創世記にはRS232Cというシリアルバスが標準化されており、これは(今でも)パラレルバスに比べてはるかに低速なバスでした。しかし、簡単な接続で機器間のデータ転送が行え、組込みシステムではデバッグ等の目的で今でも多くの機器で使われています。

バス：プロセッサバス

プロセッサとメモリ・周辺デバイスの間を接続するバスで、
パラレルバスが一般的

バス規格と種類

- AMBA, CoreConnect, その他プロセッサ毎に存在する
- 転送速度等により、複数の種類のバスを組み合わせる

マスタ

- バスに対して読み込み・書き込み要求を出す
- プロセッサ, DMA (Direct Memory Access), ...

スレーブ

- バスから読み込み・書き込みの要求を受ける
- メモリ, I/Oインタフェース (GPIO, UART), ...

プロセッサからメモリやIOインターフェイスにアクセスするバスをプロセッサバスと言います。このバスはパラレルバスを使用するのが一般的です。これは距離が短く、高速転送が要求されるからです。バスの規格はプロセッサのアーキテクチャによって定められていますので、プロセッサ毎にまちまちです。コンピュータの内部を考えると複数のバスで構成されている設計が多く、特にシステムバスは、プロセッサと混同されて使われます。システムバスはCPU周りのチップセット上のバスを示します。逆に、組み込み機器で多く使われるワンチップCPUの場合、CPU内にチップセットにあたるハードウェアを含んでおりプロセッサバス＝システムバスという構成が成り立ちます。

バス上でデータ転送を行う場合、読み出し側と受け取り側が同期を取って、データの受け渡しを行う必要があります。主従関係を持った通信機構で転送の指示を行う側をマスタ (Master)、指示を受ける側をスレーブ (Slave) と呼びます。

バス：接続

アドレス信号

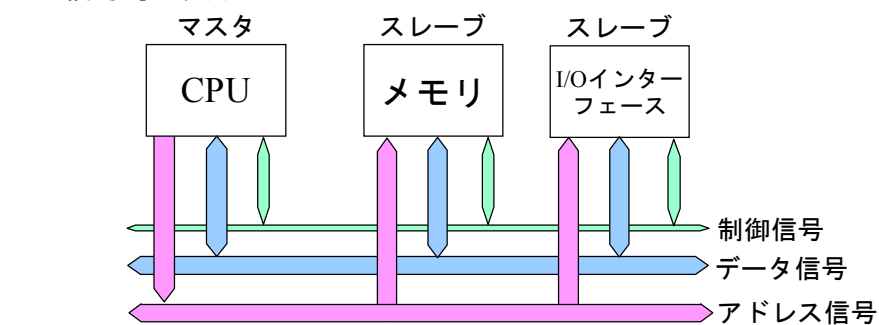
- マスタ側は出力, スレーブ側は入力

データ信号

- 読み込み時 : マスタ側が出力, スレーブ側が出力
- 書き込み時 : マスタ側が出力, スレーブ側が出力

制御信号

- 信号毎に異なる



2008/11/27

TOPPERSプロジェクト認定

33



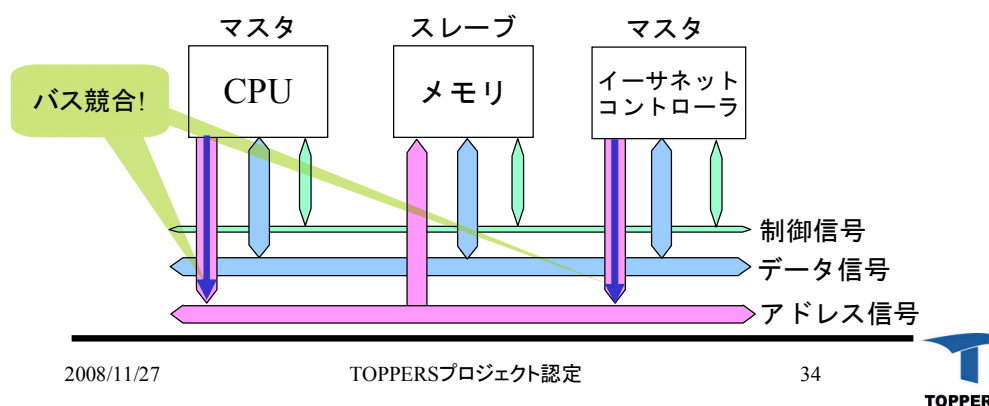
基本的なプロセッサバスはアドレス信号、データ信号、制御信号で構成される。アドレス信号はアドレスを転送するために使われる信号です。(アドレス信号はCPUから矢印の方向に送られます)メモリのアドレスやI/Oインターフェースのアドレスがやり取りされます。基本的にはマスタがプロセッサ、スレーブが周辺デバイスとなります。データ信号はプロセッサとメモリやI/Oインターフェースの間でデータを転送するための信号です。(データ信号は双方向に信号が送られます)制御信号はアドレス信号やデータ信号で入出力を行うタイミングや制御情報を取り交わす信号で、バスの構成によりさまざまな設定があります。

バス : バス競合

同じタイミングで複数のマスタがバスを使用すると、
バス競合が発生する

マスタとなるデバイス

– プロセッサ, DMA, イーサネットコントローラ...



マスタが複数になると、バスの競合が発生します。上記の例では、CPUとイーサネットコントローラがマスタとなりプロセッサバスを占有しようとした場合、バスの競合が発生します。イーサネットコントローラをマスタにしなければ、競合を考慮する必要はありませんが、イーサネットコントローラにデータの受信や送信を行う必要がある場合、データをプロセッサを用いて書き込んだり、読み込んだりする必要があり、プロセッサの負荷が増大します。イーサネットコントローラがバスマスタになれば、プロセッサは、データを読み込んだり、書き込んだりするメモリ上のアドレスをイーサネットコントローラに指定すれば、後はイーサネットコントローラがプロセッサを介さず、メモリから直接送信データを読み込んだり、受信したデータを直接メモリに書き込んだりすることができ、プロセッサの負荷を軽減できます。

バス : バス調停

バス使用するタイミングずらして衝突を回避する

バスアクセスの手順(バスサイクル)

- マスタは制御信号を使いバス権(使用权)の取得を試みる
- バス権を取得したら, 読み込み・書き込みを行う
- 読み込み・書き込みの終了後, バス権を解放する

バスアービタ

- バス権の調停(アービトレーション)を行う回路
- ラウンドロビンや優先度順にバス権をマスタに渡す

バス上に複数のマスタが存在する場合、どのマスタにバスの占有権を与えるかの調停を行う必要があります。バスの使用权の調停(アービトレーション)を行う回路をバスアービタといいます。バスアービタはプロセッサに付属した回路として、プロセッサの供給元が供給するのが普通ですが、組込み機器の外部へのデータ転送自体にハードリアルタイム性がある場合、ハードウェア設計者(ロジック設計者)が専用に設計を行うケースも多々あります。バスアービタは重要な回路で、設計ミスがあると、プロセッサのメモリアクセスが正しく動作せず(実際にはある特定のDMA処理で、データ転送が正しく行われなかったり、プロセッサのメモリに正しくデータが書かれなかったり等のハードウェア障害となるケースが多い)、ASICのつくり直しとなったり、ファームウェアで特別なプログラム制御が必要となったりします。

メモリ

システムは一般に複数種類のメモリにより構成されている

RAM：揮発性

- DRAM：大容量で比較的高速、リフレッシュが必要
- SRAM：容量は少ないがDRAMより高速で高価

ROM：不揮発性

- マスクROM, PROM, UV-EPROM, EEPROM

フラッシュメモリ：不揮発性

- ブロック単位で書き換え可能
- 書き換え回数には制限がある
- 書き込みは低速

バスステートコントローラ

- メモリ毎にアクセスタイミングが異なるため、メモリの仕様にあわせてバスの設定を変更するためのコントローラ

2008/11/27

TOPPERSプロジェクト認定

36



メモリについて説明を行います。メモリはプログラム自身やプログラムで扱うデータを保存するデバイスでさまざまな種類があります。

1) RAM(Random Access Memory)

読み書き可能な電気的な記憶媒体、電源を切るとデータが失われる。これを揮発性と呼ぶ。RAMにはDRAM(Dynamic Random Access Memory)とSRAM(Static Random Access Memory)がある。DRAMは記憶データを電荷によって行う、電荷は時間と共に失われるため定期的に再書き込みを行って(リフレッシュ)データを保持する必要がある。SRAMはフリップフロップ回路のような順序回路を用いてデータを記録するためリフレッシュを行う必要がなく、消費電力も小さい。しかし、記録単位のハードウェア単価が高いという欠点もある。

2) ROM(Read-Only Memory)

読み出し専用の半導体メモリで、電源を切ってもデータが消えない不揮発性のものを指す。特に近年組み込み開発で多く使われるROMがフラッシュメモリ(フラッシュEEPROM、フラッシュROM)です。EEPROMが1バイト単位で書き込みが行えるのに対して、フラッシュメモリはあらかじめブロック単位消去を行った後、ブロック単位でデータの書き込みを行う。一般的なRead-Only Memoryの語源は、あらかじめデータが書き込まれた状態で生産されるマスクROMやその組み込み機器では書換えができないPROMを指すが、コンピュータや組み込みの業界では、はじめマスクROMやPROMを使用していたものが、フラッシュメモリに置き換わったという経緯があり、フラッシュメモリもROMの範疇として分類している。

3) バスステートコントローラ(BSC)

メモリ仕様にあわせてバスの設定を変更するためのコントローラ。組み込みで使用するワンチップマイコンはマイコン内部にバスステートコントローラを持つものが多く、複数のDRAMに設定を変えるだけで対応が可能となっている。

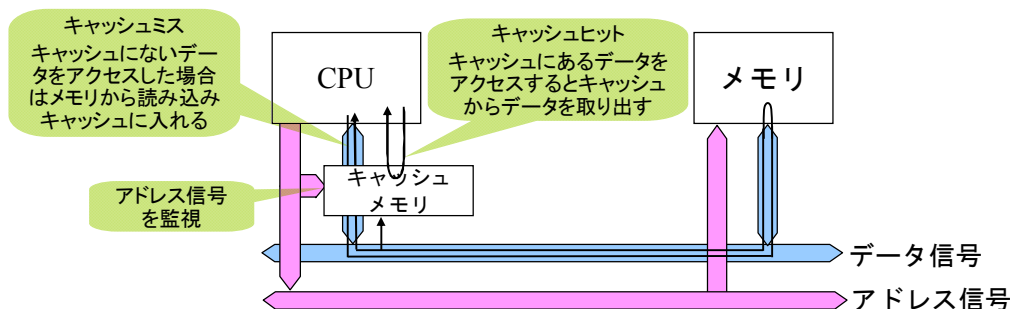
キャッシュ (1/2)

フォンノイマンボトルネック

- プロセッサは命令を実行する度にバスにアクセスしメモリを読み込むため、この間の通信速度が実行速度のボトルネックになる

キャッシュ

- プロセッサとバスの間に高速なメモリを置き、最近アクセスしたデータを格納して、次回はそちらを使用する



2008/11/27

TOPPERSプロジェクト認定

37



フォンノイマン・ボトルネック (Von Neumann bottleneck) とは、命令とデータを区別なく解釈実行するノイマン型コンピュータに存在する性能上のボトルネックを指すことばです。ノイマン型コンピュータは命令を実行するためにはアクセス速度の遅いメモリの参照が必要となり、これがコンピュータの性能を低下させる原因となっているというものです。これを解決するために、プロセッサとバスの間に高速なメモリを置き、よく使う命令やデータをこのメモリに保管することにより、性能改善が図られます。この機構をキャッシュ、キャッシュ用のメモリをキャッシュメモリと呼びます。

キャッシュ (2/2)

ライトキャッシュの方式

- ライトスルー(write through)キャッシュ
 - メモリとキャッシュの両方に書き込む
- ライトバック(write back) キャッシュ
 - キャッシュメモリのみ書き込む
 - 後で書き戻しが必要

最悪実行時間

- キャッシュを用いるとプログラムの実行速度が変動する
- すべてキャッシュミスした時を最悪とするのは悲観的過ぎ、キャッシュの必要性がなくなる
 - ➡ 実際にはそれより遅くなる場合もある

2008/11/27

TOPPERSプロジェクト認定

38



命令を扱うキャッシュメモリをインストラクションキャッシュ、データを扱うキャッシュをデータキャッシュと言います。インストラクションキャッシュは常に読み出しだけですが、データキャッシュの場合書き換えたデータをメモリに書き戻す必要があります。データを書き戻す機能をライトキャッシュと言います。

ライトキャッシュの方式には、ライトスルー (write through) キャッシュとライトバック (write back) キャッシュがあります。キャッシュメモリに書き戻すと同時にメモリにも書き戻す方式です。ライトバックキャッシュはキャッシュメモリに取り込んだデータが破棄される時点で書換えられたデータをメモリに書換えを行う。

最悪実行時間とはWCET (Worst Case Execution Time) の訳語です。組込みシステムではハードリアルタイム性を必要とするものが多くあり、これらの機器では要求時間内で特定のプログラムが確実に実行されなければなりません。最悪実行時間はプログラムが一番時間のかかる実行時間のことで、リアルタイム性を保証するために必要とされています。キャッシュ上で動作するプログラムは通常の実行時間と最悪実行時間に大きな差があるとされており、ハードリアルタイム性を保障するにはマイナス要因が多いとされています。

エンディアン(1/2)

1つのデータが複数のアドレスに跨る場合の置き方

ビックエンディアン

- 小さいアドレスに上位の桁を置く

リトルエンディアン

- 小さいアドレスに下位の桁を置く

例) 8ビット幅メモリの0x1000番地に
4バイトのデータ0x12345678を格納

ビックエンディアン

0x1000	0x12
0x1001	0x34
0x1002	0x56
0x1003	0x78

リトルエンディアン

0x1000	0x78
0x1001	0x56
0x1002	0x34
0x1003	0x12

プロセッサのデータ管理のアーキテクチャには、ビックエンディアン (**big endian**) とリトルエンディアン (**little endian**) があります。メモリ上の2バイト以上データを扱う場合1バイトごとに分割してメモリ上に格納する必要があります。どのバイト位置のデータをメモリ上のどの番地に格納するかを取り決めることをエンディアンと言います。上位の番地から昇順にデータを格納する方式をビックエンディアンといい、通信上のデータはビックエンディアンで扱われます。これに反して、データを下位の番地から降順に格納する方式をリトルエンディアンと言います。一般的にビックエンディアンの場合、データの先頭アドレスが正しいアラインの番地でない場合はアドレスエラーとなります。逆に、リトルエンディアンの場合、データが連続したイメージデータのような場合、4バイト等の大きな単位でデータ加工すると、正しく格納できない等の欠点があります。

エンディアン(2/2)

プロセッサのエンディアン

- ビックエンディアンのプロセッサとリトルエンディアンのプロセッサ
- エンディアンを静的に切り替え可能なプロセッサも存在(バイエンディアン)
 - プログラムは、コンパイル時のオプションにより、エンディアンを指定する

```
int x = 0x12345678;
char c = *((char *)&x);
```

Cの値は?

- ビックエンディアンでは、C == 0x12
- リトルエンディアンでは、C == 0x78

ビックエンディアン

0x1000	0x12
0x1001	0x34
0x1002	0x56
0x1003	0x78

リトルエンディアン

0x1000	0x78
0x1001	0x56
0x1002	0x34
0x1003	0x12

2008/11/27

TOPPERSプロジェクト認定

40



プロセッサにはビックエンディアンのプロセッサとリトルエンディアンのプロセッサがあります。また、エンディアンを静的(プロセッサの設定)で切り替え可能なプロセッサもあります。しかし、このようなバイエンディアンのプロセッサでも、実際の使用はどちらかのエンディアンで使用されます。これはコンパイラ等の開発環境が使われていないエンディアンの保障をしきれないためです。例えば、MIPS系のCPUはビックエンディアン、ARM系のCPUはリトルエンディアンで使用されます。

プログラムを作成する場合、通常データ処理ではエンディアンの違いは大きな違いはありませんが、I/Oインターフェイスにデータの読み書きを行う場合や、イメージや音楽のような不定長のデータを扱う場合、エンディアンの違いでプログラムの書換えが発生する場合がありますので、注意してください。

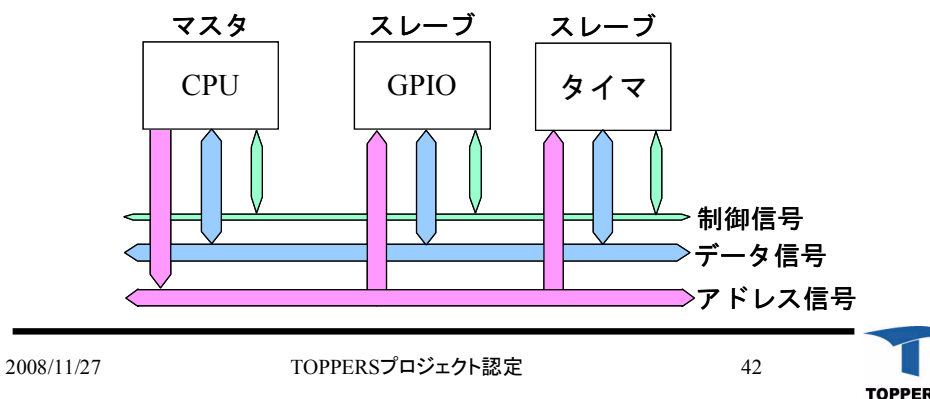
組み込みハードウェアの基礎知識

1. コンピュータの構造
2. バスとメモリ
3. 周辺デバイス
4. 外部事象の待ち方

周辺デバイス

システムに各種機能を提供する回路

- プロセッサバスに接続されている
- 一般的な周辺デバイス
 - プロセッサと外界とのやり取りを実現
 - GPIO, パラレルI/O, シリアルI/O
 - プロセッサの処理を補助
 - タイマ, DMA, アクセラレーション回路



バスに接続する電子デバイス、半導体デバイスを周辺デバイス(業界用語?)と呼びます。周辺デバイスはバスを通してプロセッサにより制御できます。周辺デバイスには外界の機器やデバイスとやりとりを行うものと、プロセッサの処理を補助するものがあります。

• プロセッサと外界とのやりとりを実現する周辺デバイス

①GPIO—General Purpose I/O、さまざまな電子デバイスを接続できるように設計された汎用パラレルI/O

②パラレルI/O—複数のデータ信号を用いて外部機器と通信を行うインターフェイスを指します。最近をシリアル化されています。SCSIやプリンタ等の通信に用いられたセントロニクスI/F (IEEE1284) 等が有名です。

③シリアルI/O—周辺機器とシリアル信号で通信を行うインターフェイスを指します。RS232CやUSBなどのインターフェイスが有名です。

• プロセッサの補助をする周辺デバイス

①タイマクロック入力を用いて、時間のカウントを行うデバイス

②DMA—I/Oやメモリ間で、プロセッサを介さずにデータを転送するデバイス

③アクセラレーション回路—プロセッサで処理を行うより高速に処理を行うための専用電子デバイス

周辺デバイス：デバイスレジスタ

プロセッサと周辺デバイスとのインタフェース

- それぞれのレジスタにはアドレスと機能が割り付けられている
 - アクセスサイズに注意
- 周辺デバイスのプログラミングの第一歩は、プログラミング対象のデバイスのデバイスレジスタを理解することである

レジスタ構成例(M16Cタイマ)

名称	アドレス	R/W	アクセスサイズ
タイマA0モードレジスタ	0x0396	R/W	8
タイマA0レジスタ	0x0387	R	16
カウント開始フラグ	0x0380	R	8

タイマA0モード
レジスタの構成

7	6	5	4	3	2	1	0
TMOD0	TMOD1	MR0	MR1	MR2	MR3	TCK0	TCK1

カウントソース
選択ビット

動作モードを指定

00:タイマモード
01:イベントカウンタモード
10:ワンショットモード
11:PWMモード

モードにより
機能が異なる

2008/11/27

TOPPERSプロジェクト認定

43



周辺デバイスの例として、M16Cに内蔵されているタイマ回路について説明を行います。M16Cの場合、メモリマップI/Oとなっており、アドレス空間にタイマ用の制御レジスタが割り付けられています。M16CにはAタイマ5本とBタイマ3本が用意されています。上記の説明ではタイマAのひとつタイマA0のデバイスレジスタについて解説を行っています。

0x0396番地に8ビットポートでタイマA0のモードレジスタが用意されています。モードレジスタはタイマの実行モードを設定するレジスタです。Bit7とBit6により動作モードを指定します。

- | | |
|-----------------|---|
| 00:タイマモード | 内部カウンタソースをカウントするモード |
| 01: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、またはアンダーフローをカウントするモード |
| 10:ワンショットタイマモード | カウント値が0x0000になるまでの間、1度だけパルスを出力するモード |
| 11: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 |

Bit5～Bit2は動作モードによって詳細のモードを設定する。Bit1とBit0はカウントソースを選択するビットで、動作モードがタイマモードの場合は、入力クロックの分周設定となります。

0x0387番地の16ビット(パルス幅変調モードの8ビットPMWをのぞく)のレジスタで、現在のカウント値の読み出し、書き込みができます。

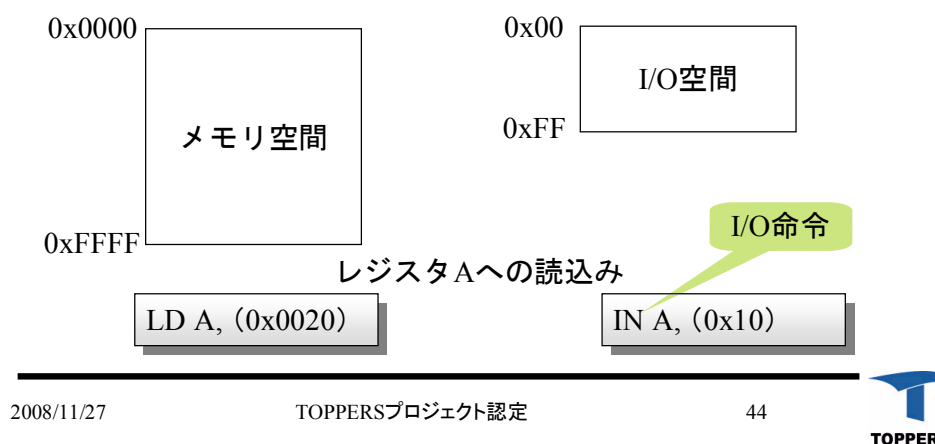
0x0380番地がカウント開始フラグで、タイマA0がBit0が対応しており、0でストップ、1でスタートです。

このように周辺デバイスの制御方法を理解するには制御用レジスタの仕様をよく理解することが、一番の近道です。

デバイスレジスタのアクセス方法：I/O空間

メモリのアドレスと周辺デバイスのアドレスを別のアドレス空間で管理

- Intel系のプロセッサで用いられている (Pentium, Z80等)
- 専用のI/O命令でアクセスする



デバイスレジスタをアクセスする手段はプロセッサにより最適な手段が異なります。メモリ空間を利用する使用するものと、プロセッサで専用に管理しているI/O空間を使用するものがあります。Intel系のPentiumやZ80等のプロセッサはI/O部をアクセスする専用のI/O空間があり、I/O命令を用いて、この空間に配置されればデバイスレジスタにアクセスします。I/O空間を持たないプロセッサでは、メモリ空間にデバイスレジスタを配置します。この場合、デバイスレジスタへのアクセスはメモリへのロード、ストア命令を用いてアクセスを行います。

デバイスレジスタのアクセス方法：メモリマップドI/O

メモリと周辺デバイスのアドレス空間を区別しない方式

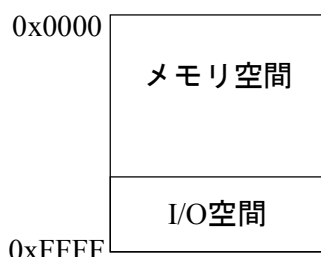
- 周辺デバイスのアクセス時にはキャッシュを無効にしなければならない。

アドレス空間

- アドレス空間を幾つかの領域に分けて無効にする領域を設定し、そこにI/Oを割り当てる(設定方法はプロセッサによって異なる)。
- 命令はメモリアクセスと同じ命令を用いる

専用命令

- キャッシュを無効にする専用の命令を持つ



2008/11/27

TOPPERSプロジェクト認定

45



I/O空間をもたないプロセッサでは、メモリ空間にデバイスレジスタを配置します。メモリ上のI/O空間はキャッシュメモリを介さないメモリ空間でなければなりません。キャッシュをもつプロセッサの場合、キャッシュを通してメモリをアクセスするアドレスとキャッシュを通さずメモリをアクセスするアドレスは別アドレスとなっている場合が多く、また、プロセッサによっては非メモリデバイス用のアクセス設定のできるアドレス空間を持つものもあります。I/O空間は非キャッシュメモリ空間や非メモリデバイス用のアドレス空間に配置されます。従って、一般のプロセッサではIOアクセスのためのキャッシュの無効化は必要ありませんが、N IOSのような特殊なプロセッサでは、このような設定が必要なようです。

キャッシュと重要なかわりをもつIOにDMAがあります。DMAはメモリ上のデータを転送します。プロセッサでデータをメモリに書き込んでも、キャッシュ上にありメモリに書き込みが行われていない状態では正しくデータが転送されません。このような場合、キャッシュ命令を用いてキャッシュ上のデータをメモリ上にフラッシュする操作が必要となります。

周辺デバイス：GPIO（汎用I/Oポート）

LSIの端子（ポート）を介してデジタル信号の入出力を行うデバイス

出力ポート

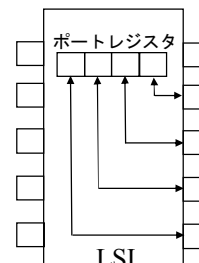
- ポートレジスタの設定内容が、ポートから出力される

入力ポート

- ポートからの入力値が、ポートレジスタに反映される

コントロールレジスタ

- 各ポートを出力ポートとして使うか、
入力ポートとして使うかなどを設定する
- ビット単位もしくはあるグループ毎に設定可能



ピンファンクションコントローラ

- LSIの端子に割り当てる機能を選択する回路
- 多くの周辺デバイスを持つシングルチップマイコンでは、LSIの端子を複数の機能で共有している場合が多い

2008/11/27

TOPPERSプロジェクト認定

46



GPIOは周辺のデバイスと簡単なデータのやり取りを行う場合、よく使われる汎用のI/Oポートです。GPIOではポートレジスタを用いて周辺デバイスと通信を行います。通信の手法では出力と入力があり、出力ではポートレジスタの内容が電位差として外部デバイスに渡されます。逆に入力では、外部の電位差がポートレジスタを通して読み込みが行えます。

ポートを入力用に使用するか、出力用に使用するかを指定するレジスタがコントロールレジスタです。コントロールレジスタの設定によりポートをビット単位またはグループ単位で入力または出力の設定を行うことができます。

LSIは端子を増やせば、増やすほどLSI単価が上がります。そのため、ひとつの端子を複数の入出力信号に割り当てLSIを使用する時点で、どの信号に割り当てるかを決定します。特に汎用のワンチップマイコンの場合、使用される目的に応じて割り当てを設定するケースものが多いようです。ピンの割り当てを設定する回路をピンファンクションコントローラ(PFC)といいます。また、設定のレジスタをピンファンクションコントローラレジスタと言います。このレジスタはリセット後、一度しか設定を許可しないものもありますので注意が必要です。

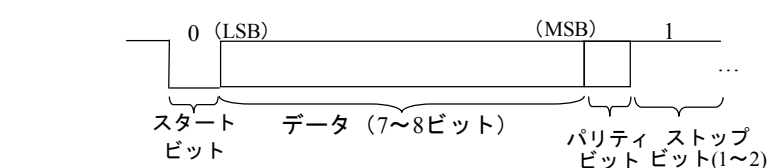
周辺デバイス：SIO(シリアルI/O)

単一のデータ線を用いて外部と通信を行う方式

- UART(RS-232C), I²C, etc...

プロトコル

- データをやり取りするための手順
- 例：RS-232Cのプロトコル
 - 幾つかの可変項目があり、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
 - データフォーマット



2008/11/27

TOPPERSプロジェクト認定

47

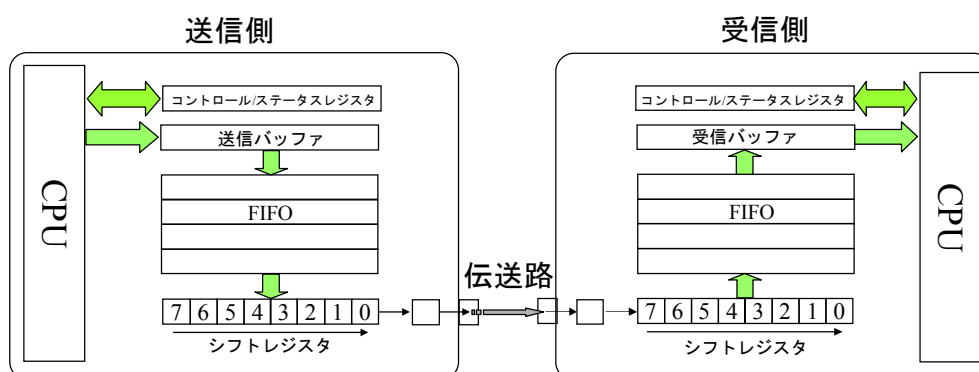


シリアルI/Oはシリアル転送方式を用いてデータのやり取りを行う入出力インターフェイスです。シリアル転送とは1本のデータ線を使って1ビットずつデータを転送する方式です。シリアル通信には、UART(RS232C)やI2CやUSBやPCIexpress等、種々の方式があります。

コンピュータの世界でのプロトコルは複数の機器間の通信のための取り決めを指します。上記の例ではRS-232C通信の調歩同期方式によるシリアル通信方式について説明を行っています。この方式の特徴はスタートビット(論理0)とエンドビット(論理1)の間で1キャラクタ毎に送信側と受信側で同期を取ってキャラクタを送る方式です。

周辺デバイス：シリアルI/O：回路構成

- ・ コントロールレジスタで送受信フォーマットを設定
- ・ 送信バッファに書き込み受信バッファから受け取る
 - それぞれFIFOがある場合が多い
- ・ 送信・受信完了は、ステータスレジスタをチェックする



2008/11/27

TOPPERSプロジェクト認定

48



上記の例はRS-232C方式でのコンピュータ間のデータ通信の例について記載しています。RS-232C方式では2つのコンピュータ間で通信設定を合わせる必要があります。通信設定は以下のような設定があります。

- ・通信形態の設定(半2重、全2重通信):通常は全2重
- ・同期方式と非同期(調歩動機)方式:組込み機器では調歩同期方式が多い
- ・パリティビットの採用の有無:通常はパリティ1ビット
- ・スタートビットとストップビット:通常はストップ1ビット
- ・データ長:通常は8ビット
- ・ボーレート:組込み機器では9800bpsや38400bpsが多い
- ・フロー制御

これらの設定はシリアルデバイスのコントローラステータスレジスタに設定を行います。送受信バッファは8ビット単位にデバイスから読み書きを行います。シリアルデータの変換を行う前にはFIFO部がありデータの蓄積を行います。

周辺デバイス：ハードウェアタイマ

時間計測の必要性

- 時間計測
- 周期的な処理の実行
- 任意の時間に処理を実行

ソフトウェア(プロセッサ)による計時

- ループの実行回数により計時
- 精度が低い(分解能が低い, キャッシュやバスによるばらつき).
- 他の処理を行えない

ハードウェアタイマ

- 時間を計測するためのハードウェア
- 入力クロックに従いカウントを行うため, 精度が高い
- プロセッサと並列に動作

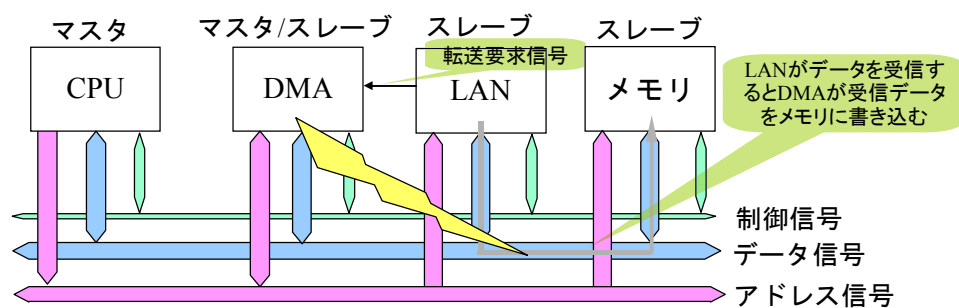
ハードウェアタイマは時間の計測を行うために使用します。カレンダーを管理するのはRTCという専用のデバイスがありますので、タイマはクロックソースによる経過時間の測定が主な仕事です。時間の計測はソフトウェアの実行時間を計算して計測することも可能です。このタイマをソフトウェアタイマと呼びます。ソフトウェアタイマでは、タイマの実行中は別の処理ができなかったり、別の処理をしようとすると、測定精度が低下する等の問題が発生します。

ハードウェアタイマは時間を測定するための専用ハードウェアで、入力クロックをカウントすることで計測を行うため、高い精度の時間測定が可能となります。

周辺デバイス : DMA (Direct Memory Access)

バスを駆動して、メモリ・メモリ間、
周辺デバイス・メモリ間のデータ転送を行う回路

- プロセッサの負荷を下げる
 - タイミング転送(事象, 時間)
- プロセッサと比較して高速に転送可能
- キャッシュに注意(OFFもしくは転送前にパージ)
- バスを駆動してプロセッサの動作を阻害するのでリアルタイム性に注意



2008/11/27

TOPPERSプロジェクト認定

50



DMAはプロセッサを介さずにデータの転送を行う周辺デバイスです。メモリからメモリにデータを転送を行ったり、メモリからデバイスへ、デバイスからメモリへ等のデータの転送が可能です。プロセッサを介さないため、プロセッサとの並列処理が可能となります。データキャッシュを使用するプロセッサの場合、DMAを開始する前に、フラッシュやクリア等の設定を行う必要があります。DMAはバスを駆動してデータ転送を行うため、プロセッサの動作や他のデータ転送を阻害しないように十分にバス設計を行う必要があります。

組み込みハードウェアの基礎知識

1. コンピュータの構造
2. バスとメモリ
3. 周辺デバイス
4. 外部事象の待ち方

外部事象の待ち方：ポーリング

外部事象の例

- スイッチが押された, データを受信した, データを送信可能, etc...

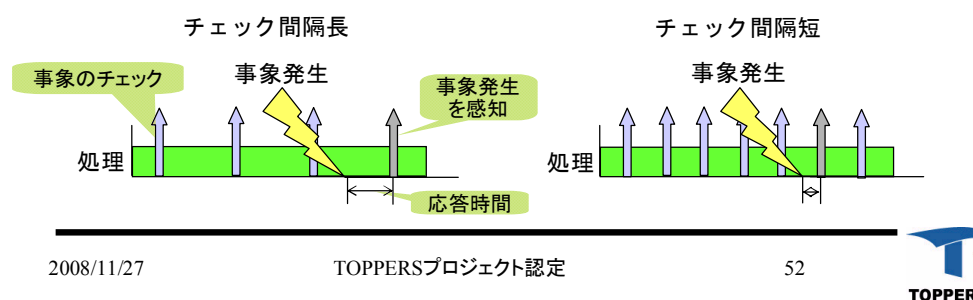
ポーリング

- 外部事象が発生すると, 対応したレジスタのビットがセットされる
- プロセッサ(プログラム)で該当ビットを繰り返しチェックする方法

チェックの間隔

- チェックの間隔を短くすると, 処理に占めるチェックのためのオーバーヘッドが増大する
- チェックの間隔を広げると事象が発生してから, その発生を感知するまでの時間が長くなり, リアルタイム性が低くなる

➡ オーバーヘッドとリアルタイム性のトレードオフ



2008/11/27

TOPPERSプロジェクト認定

52



組み込みソフトウェアはハードウェアの状態の変化を検知し、ハードウェアに適切な指示を行うことが重要な仕事となります。ハードウェアの状態変化とは、スイッチが押された, データを受信した, ...など, 外部から受ける変化のうち, その変化がトリガーとなり何か処理をするべき変化のことを言います。

一般的なハードウェアの事象待ちはポーリングを行います。外部事象が発生すると対応するレジスタのビットがセットされます。プログラムで該当ビットを繰り返しチェックすることをポーリングといいます。プログラムの的には, `for`ループの中でチェックする処理を行うことで, 繰り返しチェック処理を行うことになります。

そのチェックの間隔ですが, 間隔を短くすると, 外部処理が発生してから感知するまでの時間が短くなり, より敏感に反応することができます。しかし, プロセッサ処理に占めるチェックのためのオーバーヘッドが増大するというデメリットがあります。反対にチェックの間隔を広げると, 事象が発生してからその発生を感知するまでの時間がながくなり, リアルタイム性が低くなる可能性があります。これも, オーバーヘッドとリアルタイム性のトレードオフになります。システムによって何が要求されているかにより, 設計方針が異なります。

外部事象の待ち方：割込み

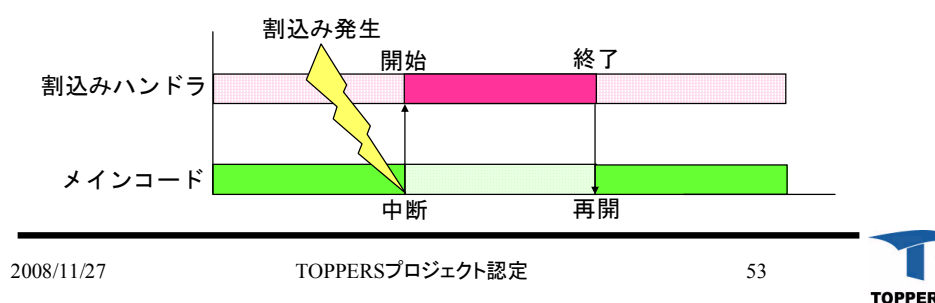
プロセッサが事象の発生を非同期に受け付ける方法

ソフトウェア的な視点

- 事象が発生すると、プロセッサが現在の処理を中断して、決められた関数(割込みハンドラ)を自動的に実行する機構
- 割込みハンドラ内に、事象に対する処理を記述する

ハードウェア的な視点

- プロセッサの割込み端子に信号が入力されると、その時点のレジスタやPCの値を保存して、PCの値を割込みハンドラのアドレスにセットする



もうひとつの事象待ち方法として、割込みがあります。割込みとは周辺機器からプロセッサへの非同期の電気的な信号のことです。割込み信号が発生すると、プロセッサは一時的に実行中の作業を中断し、割込みハンドラと呼ばれる関数を実行します。この割込みハンドラの処理が終了すると、プロセッサは中断していた作業を再開します。

このようなことが行われるためには、ハードにも準備がないと実現できません。周辺機器からの割込み信号が発生すると、プロセッサのプログラムカウンタ(PC)に定められた値をセットすることで、逐次的に実行していたプログラムを中断し、その信号に対応するプログラムを強制的に実行し、その後もとのプログラムの実行へ戻ることができる仕組みです。この仕組みにより、周辺機器の処理要求が発生すると同時に、即時対応することができます。

割込み関連の用語の整理

割込み

- 外部からプロセッサに非同期に発行される要求
- プロセッサは要求を受け付けると要求に応じた処理を行う

割込み要因

- 割込みの発生要因
- プロセッサは複数の割込みを受け付けることができる

割込みマスク/割込み禁止・許可

- 使わない割り込みは無視したい
- 割込みが入ってもらいたくないときがある

割込みハンドラ(ISR)

- 割込みが発生すると実行される処理(関数)

割込み応答時間

- 割込みを受け付けてからハンドラが実行されるまでの時間
- 実時間性が求められる組込みシステムでは重要

2008/11/27

TOPPERSプロジェクト認定

54



割込み関連の用語をまとめてみます。

1) 割込み

外部からプロセッサに非同期に発行される要求であり、プロセッサがこの要求を受け付けると、要求に応じた処理をメインのプログラムと同期せずに処理を行う。

2) 割込み要因

割込みの発生要因、電気的には信号により伝達される。要因の受付にはエッジトリガとレベルトリガがある。

3) 割込みマスク／割込み禁止許可

メインのプログラムにて割込みの禁止・許可やどの割込みを受け付けるかの選択が可能となる。割込み等の処理で阻害されたくない区間をクリティカルセクションという。

4) 割込みハンドラ(ISR)

割込みが発生すると実行される処理(関数)

5) 割込み応答時間

割込みを受け付けてから割込みハンドラが実行されるまでの時間。ハードリアルタイム性が求められる組込みシステムでは重要な要因である。

割込みと例外

割込み(外部割込み)

- 実行中のプログラムとは無関係に発生
- 緊急な処理を要求する場合に使用

例外(内部割込み)

- 実行中のプログラムに依存して発生
- エラー処理を要求する場合に使用

例外要因の例

- ゼロ除算
- ページアウト, アクセス違反, TLBミス
- バスエラー, アドレスエラー
- ソフトウェア割込み(TRAPA命令, INT命令)

！用語の使い方は人(メーカ)によって異なるので注意！

割込みは外部からの事象の変化を受け付けるための機構です。これに対して、例外は内部の事象の変化を受け付けるための機能です。处理的には似ていますが、例外は重要な問題の発生時に発行され、一般にマスクがかけられません。例外要因の例は上記を参照してください。

多様な割込みアーキテクチャ

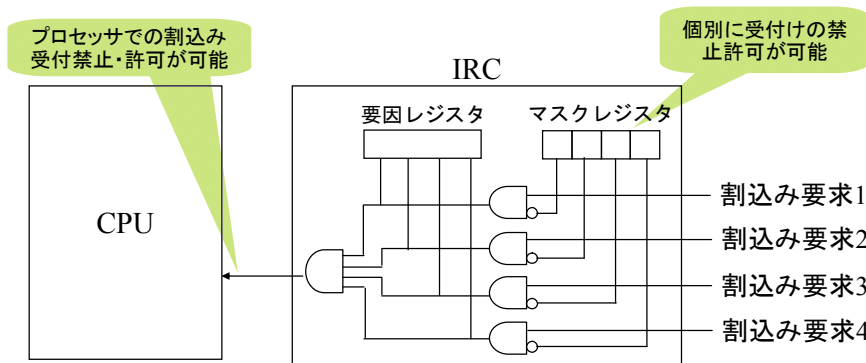
プロセッサ, システム毎に割込みアーキテクチャは異なる

- 近年のRISCプロセッサは機能が単純化する傾向にある
 - 割込みベクタ
 - テーブル型 or 常に一箇所にジャンプ
 - 割込みマスク
 - レベル型 or 個別マスク型
 - スタック
 - 割込み専用スタックの有無
 - 割込みレベル
 - 数字が大きいほど優先度が高いのかそれとも逆か
- さらに用語もプロセッサ(メーカー毎?)に異なる
 - 例外は割込みの一種か
 - 割込みは例外の一種か

割込みのアーキテクチャはプロセッサやシステム(周辺設計)により異なり、多種多用です。近年のRISC系のプロセッサは割込み機能が単純化する傾向にあります。CISCプロセッサでは割込みベクタ主流でしたが、固定番地から割込み起動するものもあります。割込みマスクもCISCではレベル型主流でしたが、ビットマスクで割込みをマスクするものも増えてきました。スタックに関しては、割込み専用スタックを持つものが主流です。

割込みコントローラ(IRC)

- 複数の割込み要求を受け付け、プロセッサへ割込み要求を出す回路
- 割込み要求毎に、要求受け付けの禁止・許可を設定可能
- どの割込み要求から要求が出ているかを判定する
割込み要因レジスタを持つ
- IRCが多段に接続されている場合もある



2008/11/27

TOPPERSプロジェクト認定

57



複数の割込みが発生した場合の調整と、割込み処理終了後に中断していたもとの処理へ戻れるようにするための仕組みを、ハードウェアとソフトウェアで用意する必要があります。なぜなら割り込み処理といえども通常のプログラムと同じであり、レジスタやメモリを資源として使用するからです。そのため、割込み処理から元の処理へ復帰させるときにはあたかも何もなかったかのように、このレジスタやメモリを元に戻すための仕掛けが必要となります。

一般的に、プロセッサには1つ以上の割り込み要因を受け付けるための入力端子が用意されています。割り込み入力端子に割り込み要求を加えます。あらかじめ用意しておいた割り込み処理プログラムを実行するように、処理の流れが変更されます。複数の割り込み要因に対応するためには、複数の割り込み入力端子を用意し、それぞれに対応した割り込み処理プログラムを記憶できるようにします。または、プロセッサに用意された一つの割り込み入力端子を利用して複数の割り込みに対応できるようにするための専用IC(割り込みコントローラ)が用意されています。

エッジトリガとレベルトリガ

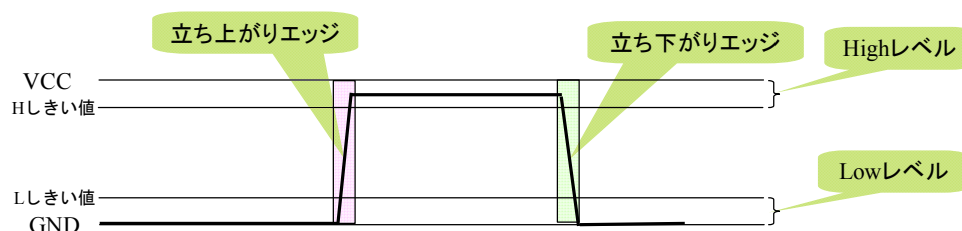
割込み要求の発生方法

エッジトリガ

- 信号の立ち上がりもしくは立下りにより、割込みの発生とみなす
 - 例) 自動販売機のボタンと, ATM/新幹線のタッチパネル
- 割込みを受け付けた後、割込みコントローラのクリアが必要

レベルトリガ

- 信号のレベルにより、割込みの発生とみなす
- 割込みを受け付けた後、割込みの発生元(ソース)を操作して、割込み要求を下げる必要がある



2008/11/27

TOPPERSプロジェクト認定

58



割込みの電氣的な要因受付には、エッジトリガとレベルトリガがあります。

エッジトリガとは、割込み要因の信号が、ローレベルからハイレベルに変化した場合(立ち上がり)、または、ハイレベルからローレベルに変換した場合(立下り)に、割込み要因が発生したと判定するものをいいます。レベルトリガとは、信号のレベルにより、割込みの発生とみなすものを言います。レベルトリガの場合、割込みを発生させたデバイスに対して、受け付けたことを通知するため、信号のレベルを戻す操作が必要となります。

割込み禁止と割込み優先度

割込みを受け付けない, また受け付けた後に
割込みがかかり続けられないためのしくみ

全ての割込みの禁止

- 全ての割込みの受け付けを禁止する
- ある一連の処理を妨げられたくない場合に使用
- プロセッサにより実現

個別の割込み禁止

- 割込み要因ごとに割込み受付の許可・禁止を設定する
 - ・ 受け付けた割込みを禁止する,
 - ・ 使用しない割込みを受け付けない
- IRCにより実現

ノンマスカブル割込み (NMI)

- 受付を禁止できない割込み,
デバッグ, システムの致命的なエラーの通知に使用される

いつでも、割込みを受け付けてしまうと、正しく機器を制御できないケースが多々あります。割込みには禁止・許可や優先度を設定する機能があります。

すべての割込みを禁止指定が可能です。プロセッサの専用レジスタによって、この設定ができます。割込み禁止を行えない割込みをノンマスカブル割込み (NMI) といいます。プロセッサにより、ノンマスカブル割込みと例外は同様な意味で用いられますので注意が必要です。

個別の割込みを禁止・許可する設定も可能です。この設定はIRCに設定を行う場合やデバイス事態に割込みを禁止させる機能があり、使い分けが必要です。逆に、割込みを許可する場合、すべての割込み許可を行わないと割込み許可にはなりませんので、これにも注意が必要です。

割込みベクタテーブル

割込み受け付け時に実行する
関数(割込みハンドラ)の先頭アドレスを置くテーブル

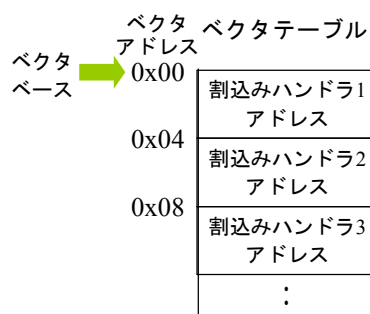
- プロセッサは割込みを受け付けると割込み要因を判定して、その要因に対応した割込みハンドラの実頭アドレスを読み込みジャンプする
- CISC型のプロセッサで一般的な方式

ベクタベース

- ベクタテーブルを置くメモリのアドレス
- 固定 or 変更可能

ベクタアドレス

- 割込み要因毎のベクタテーブル中の先頭アドレスの置き場所
- ベクタベースからのオフセット



割込みベクタテーブルとは、割込み受付時に実行する関数の先頭アドレスを置くテーブルです。割込み要因とその要因によって動くべき割込みハンドラ(関数)を対応づけるものと考えてください。この対応づけがベクタテーブルです。この割込みベクタテーブルは、割込みハンドラを実現する関数ポインタの配列で、決められたメモリアドレスに配置されます。割込みの種類によって決められている割込み番号をその配列へのインデックスとして使用します。

ベクタテーブルを置くメモリのアドレスをベクタベースといいます。これはプロセッサによって固定のものと可変のものがあります。ベクタアドレスとは、割込み要因ごとのベクタテーブルの中の先頭アドレスの置き場所のことです。通常は、ベクタベースからのオフセットで表します。

RISC型割込みベクタ

割込み発生時に特定のアドレスから実行を開始する

- RISCプロセッサで一般的な方法

例1) SH3 : ベクタベースは可変.

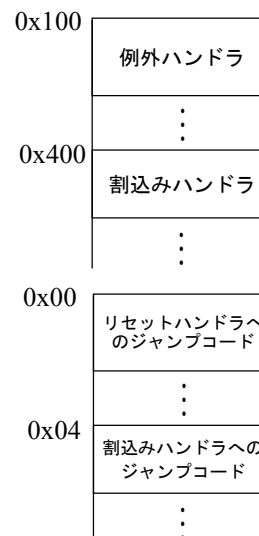
それぞれのアドレスにハンドラを置く

- リセット : 0x000
- 例外 : ベクタベース + 0x100
- 割込み : ベクタベース + 0x400

例2) ARM : ベクタベースは固定.

ベクタの間が4byteであるため, それぞれのアドレスにハンドラへのジャンプコードを置く

- リセット : 0x00
- 未定義命令実行 : 0x04
- データアボート : 0x10
- 割込み : 0x18



2008/11/27

TOPPERSプロジェクト認定

61



RISC系のプロセッサでは、割込み発生時に特定のアドレスからプログラムの実行を開始するものが多いです。

上記の例では、SH3の場合、リセットの発生時、0x000番地からリセットプログラムが実行します。同様に、例外はベクタベースから0x100番地から、割込みはベクタベースから0x400番地から割込みプログラムが実行します。指定の番地の例外や割込みプログラムが収まりきらない場合は、空き番地にジャンプしてプログラムを記述する必要があります。

ARMでは、プログラムを記載する場所がない番地から例外や割込みが実行されます。ジャンプ命令を書きこんでプログラムを分岐する必要があります。

プロセッサの割込み/例外処理の例 (CISCプロセッサ:M16C)

- 要求された割込みの優先レベルが、プロセッサのフラグレジスタ (FLG)中の割込み優先マスク(IPL)より大きければ、割込みを受け付ける
- 割込みを受け付けると、割込み要因に対応したベクタテーブルに登録されているアドレスにジャンプする。
その際、以下の処理をハードウェアが行う
 - スタックを割込みスタック (ISP) に変更
 - 割込み要因に対応した割込み制御レジスタのIRビットをクリア
 - FLGの割込み許可フラグを‘0’にして割込み禁止
 - FLGとPCをスタックに保存
 - 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

2008/11/27

TOPPERSプロジェクト認定

62



M16Cの割込みと例外について記載します。M16Cの場合、割込み禁止の設定はFLGレジスタ中の‘I’ビットが1で割込み許可、0で割込み禁止となります。また、割込みの許可レベルはFLGレジスタ中のIPL (4ビット)の設定で行います。IPLの設定以下の割込みレベルの割込みは禁止されます。IPLに15 (0xF)を設定しても全割込み禁止となります。

M16Cでの割込み受付手順は以下の通りです。

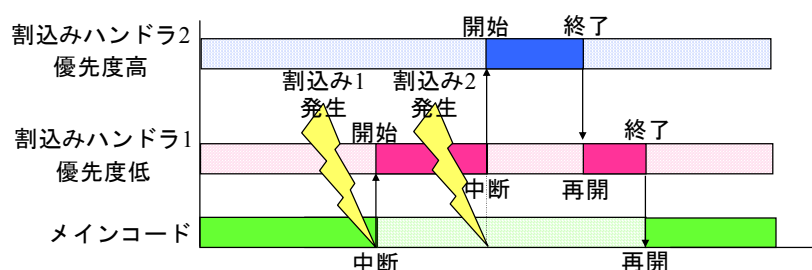
- 1) 割込みを受け付けると、スタックを割込みスタック (ISP) に変更する。
- 2) 割込み要因に対応した割込み制御レジスタのIRビットをクリアする。
- 3) FLGの割込み許可フラグ‘I’を0にクリアして割込み禁止にする。
- 4) FLGの内容とPCをスタックに保存します。
- 5) プロセッサ割込み優先レベル (IPL) に受け付けた割込みレベルを設定する。

多重割込み：高優先度の割込みの発生

割込み処理中にさらに高優先度の割込みの要求を受け付ける

割込み優先レベル

- 各割込み要因に優先度を持たせる。プロセッサは割込みレベルを持ち(可変)、現在の割込みレベル以下の優先度を持つ割込みが発生しても、割込みを受け付けない
- 割込み毎に適切に優先度を設定する必要がある



2008/11/27

TOPPERSプロジェクト認定

63

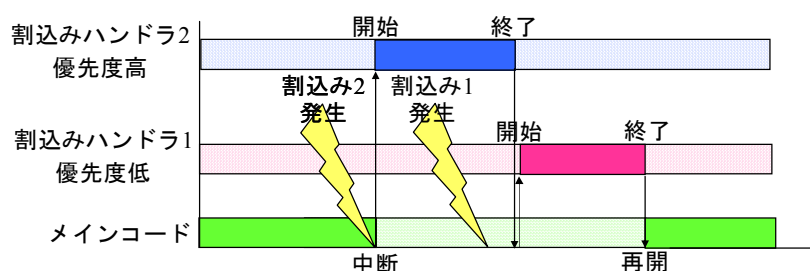


割込みは外部イベントであり、プログラムやカーネルの処理中にも発生します。つまり割込みハンドラの優先度は、実行中のプログラムやカーネルよりも高く設定されています。割込み処理中にも割込みが発生することがあります。この割込みを受け付けられるようにした方式を多重割込みといいます。多重割込みに関しては、割込みハンドラごとの実行優先度を考えなければなりません。ある割込み処理中により優先度の高い割込みが発生すると、あとに発生した割込みが優先度の低い割込みに割り込んで処理されます。つまり先に実行していた優先度の低い割込みハンドラは一時中断します。優先度の高い割込み処理終了後、中断された処理の続きを行います。

後に発生する割込みハンドラを実行させたくない場合は処理中の割込みハンドラが発生してくる割込みを禁止しなければいけません。

多重割込み：低優先度の割込みの発生

- 高優先度の割込みを受け付けている際に、受付中の割込みより低優先度の割込みが発生すると、高優先度の割込み処理が終了してから、低優先度の割込みが受け付けられる



2008/11/27

TOPPERSプロジェクト認定

64



高い優先度の割込みハンドラが実行中に優先度の低い割込みが発生した場合、優先度の低い割込みは優先度の高い割込みハンドラの処理が終了してから受けつけられます。それまで待たされることになります。

このようなことがありますので、各割込みハンドラの優先度は考慮して設定する必要があります。また、高い優先度の割込みハンドラは、他の処理を待たせる可能性がありますので、高い優先度の割込みハンドラほど、短い時間で終了するような配慮が必要です。

組込みプログラム開発の基礎知識

1. [開発環境](#)
2. デバッグ環境
3. 実行環境
4. コーディングルール

クロス開発環境：ターゲットシステムとホストシステム

組込みシステムはクロス開発により開発する

クロス開発

- 開発環境（ホスト）と実行環境（ターゲット）が異なる開発形式
- 機械語も異なる

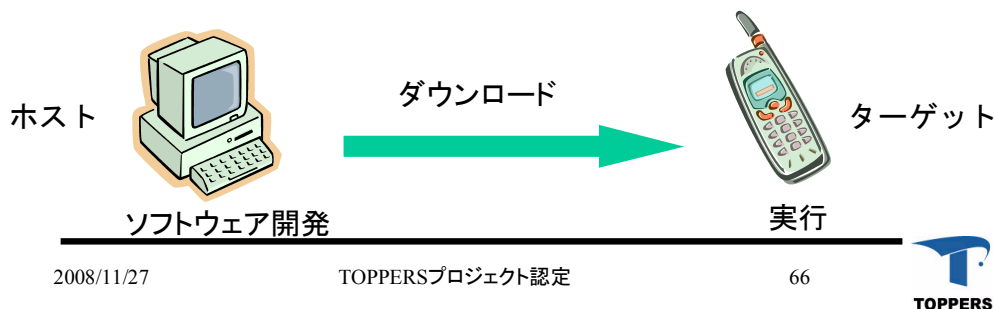
↔ セルフ開発

ホスト

- ソフトウェアを開発する（開発環境を実行する）計算機

ターゲット（ターゲットプロセッサ）

- 開発対象の計算機（システム）



組込み開発の大きな特徴がクロス開発環境です。クロス開発環境とは、組込みプログラムの開発を行うシステム（ホストシステム）と実際にソフトウェアが動作するターゲットシステムが異なるソフトウェア開発環境を指します。現在のコンピュータの開発環境がセルフ開発環境であることを考えれば、クロス開発環境では種々の開発負荷が発生します。大きな負荷は作成したソフトウェアを実行するためにターゲットシステムにソフトウェアを記憶させる必要があります。これを実現するための一般的な方式がダウンロードという方式で、ホストシステムとターゲットシステムの間にソフトウェアを通信するプログラムとターゲットシステム上でそれを記憶実行するプログラムを用意して、ホストシステムで開発したソフトウェアをターゲットシステムで実行します。プロセッサメーカーの開発環境やJTAG等のICEが設定できれば、開発側でこれらのプログラムを開発する必要はありませんが、通常、組込みに使用しないようなプロセッサを使用する場合は、これらのプログラムの開発も組込み開発作業に含まれる可能性もあります。

ホストシステム上にはクロス開発環境用ツールをインストールします。クロス開発環境ツールには以下のツールが含まれます。

- クロスコンパイラ: ターゲットシステム用プロセッサのCやC++言語用コンパイラ
- クロスアセンブラ: ターゲットシステム用プロセッサのアセンブラ
- クロスリンカ: クロスコンパイラやアセンブラで作ったオブジェクトを実行形式に結合変換するツール
- クロスデバッガ: ターゲットシステムをデバッグするツール（供給されない場合もある）
- シミュレータ: ホストシステム上でターゲットシステム用のプログラムを検証するツール（供給されない場合もある）

組込みソフトウェア開発に使われるプログラミング言語

C言語

- ハードウェアを直接操作するプログラミングが可能であるため、組込みソフトウェア開発では、最も使われている

アセンブリ言語

- DSPなどの特殊なプロセッサで使われる場面が多い
- コンパイラが扱えない特殊命令を直接記述して、性能を出す

C++言語

- 利用は広がっているが、まだ限定的
- オーバーヘッドが大きい
- どのような実行コードになるか見えにくい

2008/11/27

TOPPERSプロジェクト認定

67



一般的に組込みで使用されるプログラミング言語について記載します。

•C言語

1970年代にベル研究所のデニス・リッチーがUNIX用に開発したプログラミング言語です。ハードウェアを直接操作するプログラミングが可能のため、組込みソフトウェア開発では、もっとも多く使われている言語です。移植性の良さの普及の要因となっています。

•アセンブラ言語

プロセッサの動作を記述する機械語を人間にわかりやすいように表記したプログラミング言語、C言語はコンパイラによりアセンブラ言語の変換されます。C言語では記述できない特別な操作(レジスタの設定、読み出しやスタックの制御)を記述するのに使用します。また、DSPなどの高級プログラミング言語のない特殊なプロセッサに多く使われます。

•C++言語

C言語に対して、オブジェクト指向的な拡張を行ったプログラミング言語、商用では多く使用されています。組込み用に利用は広がっていますが、また、限定的です。組込み用途に広まりを見せない原因は、以下のことが指摘されています。

- 1) プログラムサイズやデータサイズや実行オーバーヘッドがC言語に比べて大きい
- 2) 関数名やデータ名が隠蔽化されるため、組込み用のデバッグがしにくい等

C言語のツールチェーン

C言語コードを実行コードに変換するためのツール郡

コンパイルドライバ

- 実行コードを生成するまでの一連の処理を実行
 - プリプロセッサ, コンパイラ, アセンブラ, リンカを呼び出す

プリプロセッサ

- #includeやマクロを展開

コンパイラ

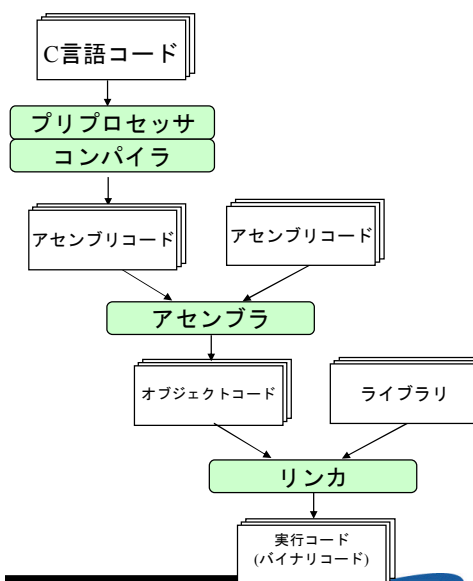
- プリプロセスされたC言語コードをアセンブリコードへ変換

アセンブラ

- アセンブリコードをオブジェクトコード(機械語プログラム)に変換

リンカ

- 複数のオブジェクトコードとライブラリをリンクし実行コードを生成する



2008/11/27

TOPPERSプロジェクト認定

68



クロス開発環境でのプログラミング言語とクロス開発ツールの関係について記載します。

C言語やC++言語で記載したソースコードはクロスコンパイラを用いて、アセンブリコードに変換されます。通常、クロスコンパイラはクロスアセンブラを自動実行しますので、オブジェクトコードに変換すると言った方が正しい記載かもしません。アセンブリコードもクロスアセンブラを用いてオブジェクトコードに変換します。複数のオブジェクトコードとライブラリを入力として結合し、ターゲットシステムで実行可能な実行コードに変換するプログラムをクロスリンカと呼びます。

また、ビルドに使用するツール郡をツールチェーンと呼びます。(広義のツールチェーンでは開発環境を含む場合もあります)

M16C用のツールチェーン

プリプロセッサ : cpp30

- #includeやマクロを展開する

コンパイラ : nc30, ccom30

- オブジェクトコードを生成するまでの一連の処理(プリプロセッサ, コンパイラ, アセンブラ, リンカの呼び出し)を行う場合は, コンパイラドライバとも呼ばれる

アセンブラ : as30

- アセンブリ言語記述を機械語プログラムに変換する.

リンカ : ln30

- 複数の機械語プログラムとライブラリをリンクし最終イメージを生成する

2008/11/27

TOPPERSプロジェクト認定

69



M16C用のツールチェーンについて説明を行います。

•M16C用クロスコンパイラ

M16C用のC言語用クロスコンパイラはnc30です。コンパイラは複数のフェーズでソースソースコードをオブジェクトコードに変換します。nc30はコンパイルのための一連の処理手順を行うプログラムであり、このようなプログラムをコンパイラドライバと呼びます。nc30から呼び出される処理は以下の通りです。

- 1) cpp30: プリプロセッサ - マクロ記述 (#include, #define 等) た条件コンパイル (#if ~ #else ~ #endif) の解決処理を行います。
- 2) ccom30: コンパイラ本体 - C言語ソースをas30で処理可能なアセンブリ言語ソースコードに変換します。
- 3) aopt30: アセンブラ用オプティマイザ
- 4) 設定によりas30やln30も実行します。

•M16C用クロスアセンブラ

M16C用のアセンブリ言語コードを機械語プログラムのリロケータブルモジュールファイル(IEEE-695準拠)に変換します。

•M16C用クロスリンカ

リロケータブルモジュールファイルやライブラリファイルを分枝情報ファイルやコマンド指定に従って、アプソリュートモジュールファイルに変換します。

クロスコンパイラ

C言語等の高級言語記述からホストコンピュータと異なる
プロセッサのアセンブリ言語記述を生成するプログラム

- 一般に、コンパイルの前にプリプロセッサを呼び出し、コンパイル後にアセンブラ・リンカを呼び出し、実行コードを生成する
 - コンパイラドライバ
- 豊富なコンパイルオプション
 - エンディアン, フローティングの扱い, etc...
 - ターゲット非依存とターゲット依存
 - ターゲット依存のコンパイルオプションの例 (Nios2用gcc)
 - -meb : Use big-endian byte order
 - -mel : Use little-endian byte order
 - -mno-hw-mul : Disable MUL instructions
 - -mno-hw-div : Disable DIV
 - -msmallc : Link with a limited version of the Clibrary

2008/11/27

TOPPERSプロジェクト認定

70



クロスコンパイラとは高級言語で作成したソースコードから、開発に使用している機器とは異なる機器で使用可能な実行可能形式のプログラムを生成するソフトウェアを指します。これに反して開発機器で実行するプログラムを生成するコンパイラをセルフコンパイラと呼びます。また、機械語プログラム(機械が直接理解できる言語)をネイティブコードと呼び、ネイティブコードを生成するコンパイラをネイティブコンパイラ呼びます。例えばJAVEで生成される実行形式は基本的にネイティブコードではありません。

現在のコンパイラは種々のコンパイル機能を順番の呼び出し制御するコンパイラドライバを指します。前述の通り、コンパイルは種々のサブ機能を組み合わせて実行する巨大のプログラムです。そのため、コンパイラの内に不具合が含まれる場合もあります。このような場合、以下の対処方法があります。

- 1) アセンブリ言語を生成して、問題の部分のアセンブリプログラムを確認してみる。
- 2) 最適化を行っている場合、誤ったアセンブリプログラムを生成することがあります。これも、アセンブリプログラムを確認すれば確認ができます。このような場合、最適化を行わないようにすれば回避できる場合がおおくあります。

クロスコンパイラ：プリプロセッサ

コンパイルの前処理, プリプロセッサにより実行される

プリプロセッサディレクティブ

- プリプロセッサへの指令
- includeディレクティブ
- defineディレクティブ(マクロの展開)
- 条件コンパイル

マクロの展開

```
#define N_DATA 10
...
int n_data = N_DATA
```

条件コンパイル

```
#ifdef DEBUG
...
#else
...
#endif /* DEBUG */
```

多重インクルードの防止

```
#ifndef _TEST_H_
#define _TEST_H_
本文
...
#endif /* _TEST_H_ */
```

2008/11/27

TOPPERSプロジェクト認定

71



クロスコンパイラのプリプロセッサ機能について説明を行います。プリプロセッサはコンパイルの前処理に当たります。この処理はプリプロセッサディレクティブを用いて以下の定義の解決を行い、必要な場合エラーの通知を行います。

- 1) #include文の解決
- 2) #defineや#undef等のマクロの展開
- 3) #if、#else、#elif、#endif等の条件付コンパイルの解決
- 4) #warning、#error、#line等のエラーや警告の報告
- 5) #regionや#endregion等のソースコードの領域分け
- 6) #pragmaのプリAGMA

クロスコンパイラ：最適化技法

効率のよいアセンブリコードを生成する技法

- 基本はプログラムの実行速度の向上
- 数多くの技法が存在し、コンパイラオプションで細かく制御（有効・無効）可能なコンパイラが多い
- 組込みシステム特有の最適化
 - コードサイズ縮小
 - コンパイラの最適化技法の中には、高速化とコードサイズの縮小が同時に達成できるものと、両者が両立しないものがある
 - 同時達成：共通部分式除去
 - 両立不可能：ループ展開
 - 消費電力削減のための最適化
 - 研究テーマ段階

コンパイラの最適化技法について説明を行います。コンパイラには実行時間の高速化やメモリ使用量の最小化をおこなうために最適化の機能があります。これらを実現するために数多くの技法が存在し、コンパイラオプションを細かく制御可能となっています。組込み特有の最適化として消費電力を最小化する最適化も研究中です。

クロスコンパイラ：最適化の弊害

メモリアクセス

- 最適化によりメモリアクセスパターンが変わる可能性
 volatile宣言(標準のC言語の機能, 後述)

デバッグ

- 最適化レベルを上げるとデバッグが難しくなる
 - 最適化により命令の順序が入れ替わるため, ソースコードとの対応をつけることが難しくなる
 - 関数の途中で, 引数の値が失われている場合も (デバッガが扱いに困る)
- ソフトウェアテスト・デバッグ中は最適化レベルを下げておき, テストが終ると最適化レベルを上げる方法は, 受け入れられない場合が多い
 - 最適化レベルで時間的な振舞いが変わる
 - コンパイラの最適化にバグがある可能性

2008/11/27

TOPPERSプロジェクト認定

73



最適化も、全て良いことばかりではありません。メモリアクセスに関してはプログラムで記載通りのメモリアクセスでは、論理的に不要なメモリアクセスが発生する場合、単純なアクセスに置き換えるような最適化が行われます。**volatile**宣言を用いることにより、これを明示的に防止することができます。

また、デバッグにおいても最適化を行うと、C言語コードとは異なった実行順序の機械語コードが生成され、アセンブリコードを生成しても、C言語との対比が難しかったり、関数の途中で引数の値がなくなったり、自動変数の割り当てが不規則になったり等、デバッガでトレースすることが困難になります。この問題として、ソフトウェアのデバッグ中は最適化を解除しておき、デバッグが終わると最適化を行ってプログラムを作成する方法が考えられますが、以下の理由により、実際のプロダクトでは受け入れられない場合が多くあります。

- 1) 最適化の設定により時間的な振る舞いが変わってしまう。
- 2) コンパイラの最適化にバグがあったり、不具合が発生する可能性がある。

クロスコンパイラ：インラインアセンブラ

C言語記述中に直接アセンブリコードを記述するための機能

- C言語では記述できないプロセッサの特殊命令や、特殊レジスタの操作を記述する
 - コントロールレジスタの操作
 - 割込み禁止許可命令
- 記述方法は、コンパイラによって異なる

NC30の例

1行だけアセンブリ言語で
記述する場合

```
asm("    fset I")
```

複数行アセンブリ言語で
記述する場合

```
#pragma ASM
fset I
fclr I
#pragma ENDASM
```

インラインアセンブラはC言語やC++言語のソース記述中に直接アセンブリコードを記述するための機能です。言語記述にないコントロールレジスタの操作や割込みの設定を直接ソースに記載できます。記述の方法はコンパイラによって異なります。但し、インラインアセンブラを多用しすぎるとプロセッサ毎の移植性が低下しますので注意してください。

クロスコンパイラ：プラグマとアトリビュート

コンパイラへ特定の情報を渡すためのコンパイラ指定

- プログラム中に記述する
- C言語標準の指定(浮動小数点演算関連)もあるが、多くはコンパイラ毎に定義されている

プラグマの例

- warning除去 : `#pragma warning(disable : xxx)`
- 割込みハンドラ関数の指定 : `#pragma interrupt`
- アセンブラ関数 : `#pragma inline_asm`
- アライメント : `#pragma align 4`

アトリビュートの例

- セクション指定 : `int a __attribute__((section(".shared"),nocommon));`
- 関数のエイリアス : `void foo() __attribute__((alias("bar")));`
- 常にインライン : `void foo() __attribute__((flatten))`

2008/11/27

TOPPERSプロジェクト認定

75



プラグマはコンパイラに特定の情報を渡すために使用するコンパイラ指定です。プラグマを使用することにより、より詳細なコンパイル内容の制御が可能になります。

プラグマは**#pragma**で始まる書式でソースコード内に設定できます。プラグマの指定はコンパイラ毎の定義されており、多用すると移植性が低下しますので注意してください。

クロスアセンブラ

アセンブリ言語で記述されたプログラムを入力とし、
ターゲットプロセッサ用の機械語のプログラム
(オブジェクトコード)を生成するプログラム

アセンブラプリプロセッサ

- 処理の前にプリプロセッサを呼び出すものもある
 - C言語と同等のマクロが使用可能

アセンブリ言語記述の互換性

- 同じプロセッサのアセンブラであっても、入力となるアセンブリ言語記述の互換性があるとは限らない
- マクロ定義、定数やデータの記述方法が異なる

クロスアセンブラはアセンブリコードを解析し、開発に使用している機器とは別の機器で実行可能な機械語のプログラム(オブジェクトコード)を生成するソフトウェアです。GNUのアセンブラの場合、アセンブリプリプロセッサがあり、C言語と同じマクロの指定が可能です。全てのアセンブラでC言語と同じマクロを指定できるわけではなく、プリプロセッサ機能がないものもあります。逆に、プロセッサ用のアセンブラであっても、開発元によりアセンブリ言語やマクロ定義や定数やデータ記述が異なる場合がありますので注意が必要です。

クロスリンカ

複数のオブジェクトコードをまとめてターゲットプロセッサで
実行可能な実行コードを作成するプログラム

- ロケーション決めとアドレス(シンボル)解決

セクション

- セグメントとも呼ばれる
- コンパイラ, アセンブラが出力するプログラムまたはデータの固まりをセクションと呼ぶ
- セクションにはそれぞれプログラム属性を持っている
- 標準的なコンパイラは, 次のセクションを生成する(呼び方はコンパイラによって異なる)
 - text : プログラムコード
 - rodata : 定数データ(文字列)
 - data : 初期値を持つデータ
 - bss : 初期値を持たないデータ(0に初期化される)

2008/11/27

TOPPERSプロジェクト認定

77



クロスリンカは複数のオブジェクトコードやライブラリを結合して、開発に使用している機器とは別の機器で実行可能な実行コードを生成するソフトウェアです。結果として、関数やデータのロケーションが決定しアドレスが確定します。確定したアドレスはオプションでマップファイルやシンボルフファイルの生成を行えば参照が可能です。オブジェクトのリンク時プログラムは個々の属性によりセクション別に再配列されます。この分別はセグメントとも呼ばれます。以下にGNUリンカのシンボルフファイルを載せます。

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.text	00012d00	00200400	00200400	00000180	2**4
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
1	.vector	00000030	00200000	00200000	00000100	2**2
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
2	.varrom	00005fc8	00213100	00213100	00012e80	2**2
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
3	.data	0000040c	002190c8	002190c8	00018e48	2**2
	CONTENTS, ALLOC, LOAD, DATA					
4	.bss	0000385c	002194d4	002194d4	00019254	2**2
	ALLOC					
5	.var	00000dc0	ffffd000	ffffd000	00019280	2**2
	CONTENTS, ALLOC, LOAD, DATA					
6	.comment	00001008	00000000	00000000	0001a040	2**0
	CONTENTS, READONLY					
7	.debug	000df3fc	00000000	00000000	0001b048	2**2
	CONTENTS, READONLY, DEBUGGING					
8	.line	000111c2	00000000	00000000	000fa444	2**0
	CONTENTS, READONLY, DEBUGGING					
9	.debug_srcinfo	00002e9c	00000000	00000000	0010b606	2**0
	CONTENTS, READONLY, DEBUGGING					
10	.debug_sfnames	00002bf0	00000000	00000000	0010e4a2	2**0
	CONTENTS, READONLY, DEBUGGING					
11	.debug_aranges	0000130c	00000000	00000000	00111092	2**0
	CONTENTS, READONLY, DEBUGGING					
12	.debug_pubnames	00001b84	00000000	00000000	0011239e	2**0
	CONTENTS, READONLY, DEBUGGING					

クロスリンカ : セクション

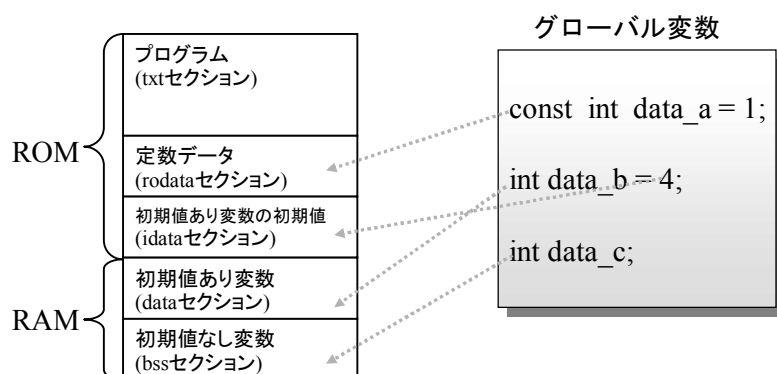
独自セクション

- ユーザー独自のセクションを作成可能.
- 例) ベクタテーブル

→ メモリロケーションの指定(後述)

セクションの指定

- プラグマ, 属性指定 (gccではattribute), アセンブラへの指令



2008/11/27

TOPPERSプロジェクト認定

78



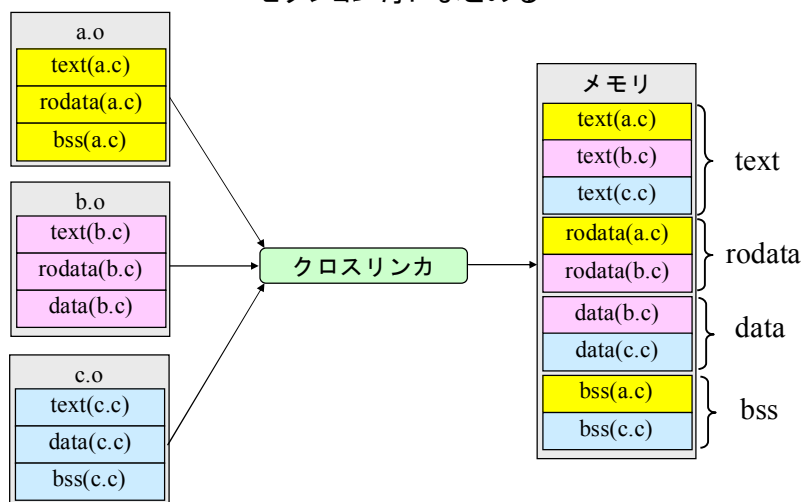
独自のセクションを設定することが可能です。例えばベクタテーブルのセクションを設定できます。(前頁の.vector) GNUの場合リンクスクリプトやプラグマで設定を行います。

上記の例はグローバル変数が、どのセクションに設定されるか示されています。

- 1) `const ....`は、変更のない固定データなので**rodata**セクションに設定されます。
- 2) `int data_4 = 4;`は初期値ありのデータなので**data**セクションにアドレスされます。初期値は**idata**セクションの置かれ、ブート時スタートアップモジュールにより**data**セクションにコピーされます。
- 3) `int data_c;`は初期値なしデータなので**bss**セクションにアドレスされます。一般的に**bss**セクションはブート時にスタートアップモジュールによりゼロ値に初期化されます。

クロスリンカ : 複数のオブジェクトコードのまとめ

各オブジェクトコードに含まれてるセクションを
セクション毎にまとめる



クロスリンカの機能について説明を行います。クロスリンカの入力データであるオブジェクトコードは上の説明のa.o、b.o、c.oのようにセクション毎に各データは配置されて保存されています。クロスリンカは入力オブジェクトコードのデータをセクション毎に再配置します。その後アドレスデータに従って、セクションをアドレッシングします。結果として、個々のオブジェクトコード中のセグメントが、実行形式のまとめられます。

クロスリンカ：メモリロケーションの指定 (リロケーション)

各セクションをどの場所(アドレス)に配置するかを指定する

ROMとRAMの使い分け

- 固定値のプログラムとデータはROMに、可変値のデータはRAMに

メモリの性質による使い分け

- 組込みシステムは性質の異なるメモリで構成されている
 - 高速だが小容量のSRAM
 - 低速だが大容量のDRAM
 - 不揮発のフラッシュメモリ
- データやプログラムの性質に応じて配置する場所を適切に指定する

特定アドレスへの配置

- ベクターテーブル等はプロセッサによって定められた場所(アドレス)に配置する必要がある。
- リセット後の実行開始アドレスにプログラムの先頭を配置する(スタートアップモジュール)

2008/11/27

TOPPERSプロジェクト認定

80



組込みシステムの場合、プログラムの各アドレスを細かく設定する必要があります。リンカは種々の設定とセクションの組み合わせでいろいろなアドレッシングが可能です。パソコン上の実行形式ファイルでは順番でセクションを並べれば実行可能ですが、組込みでは、なぜ、細かなアドレス設定が可能なのでしょう

1) ROMとRAMの使い分け

組込みシステムではプログラムと固定のデータはROMに、可変値のデータはRAMに配置されます。また、ROMとRAMのアドレスはプロセッサやボードによってさまざまに配置されますので、それに合わせてアドレス設定する必要があります。

2) メモリの性質により使い分け

RAMの構成も異なる場合があります。高速アクセスなデータはSRAMに、大容量のデータはDRAMに、不揮発性のデータはEEPROM上に等、同じデータでも、いろいろ分割してアドレスを設定する必要があります。

3) 特定アドレスへの配置

ベクターテーブル等はプロセッサによって定められた特定のアドレスに配置する必要があります。電源投入後やリセット後、特定のアドレスからプログラムを実行する場合は特定のプログラムを特定のアドレスに配置する必要があります。

クロスリンカ : リンカスクリプト

リンカに対して各セクションのリンク方法や
その先頭番地を指定するスクリプト(名前は環境によって異なる)

GCCの例

```
MEMORY{
  VECT (rx):  ORIGIN = 0x000000, LENGTH = 0x0001c0
  ROM  (rx):  ORIGIN = 0x0001c0, LENGTH = 0x01fe40
  RAM  (rwx): ORIGIN = 0x21f000, LENGTH = 0x001000  /* 外部RAM */
  STACK(rw):  ORIGIN = 0xffe600, LENGTH = 0x000600  /* 内蔵RAM */
}
SECTIONS{
  .vectors : {
    *(.vectors)
  } > VECT
  .text : {
    entry.o(.text)
    *(.text)
    *(.rodata)
  } > ROM
  .data : {
    *(.data)
  } > RAM
  .....
}
```

2008/11/27

TOPPERSプロジェクト認定

81



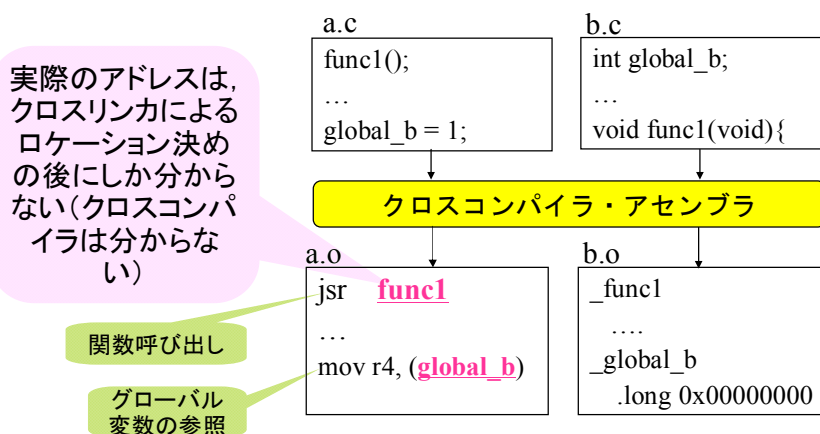
GCCの場合、リンカスクリプトを用いてセクションやアドレスの設定を行います。

クロスリンカ : アドレス(シンボル)の解決

コンパイル時には定まっていない参照アドレスを決定する

参照アドレスの使用箇所

- 関数呼び出し(関数の先頭番地)
- グローバル変数の参照



2008/11/27

TOPPERSプロジェクト認定

82

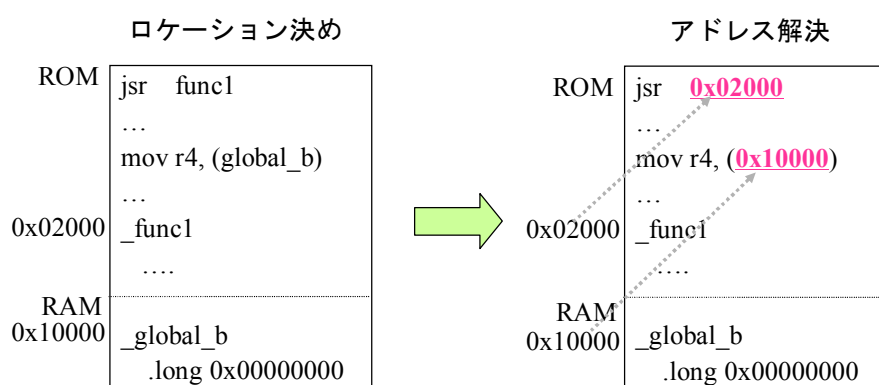


各セクションの先頭が決定することにより、関数の先頭番地やグローバル変数のアドレスが解決します。上記の例ではb.cで記述されたfunc1がコンパイルによりC関数にアセンブラ表記では前にアンダーラインが付けられ_func1となり、これがROMの先頭にa.oのtextセクション、その後b.oのtextセクションが配置され、_func1のアドレスが0x02000番地に決定します。また、b.cの先頭のグローバル変数global_bがコンパイルによりC関数にアセンブラ表記では前にアンダーラインが付けられ_global_bとなり、これがRAM領域の先頭にb.oのdataセクションとして配置され、_global_bのアドレスが0x10000番地に決定します。

クロスリンカ : アドレス(シンボル)の解決

ロケーション決めの後, 参照アドレスを用いている
箇所のコードを変更する

- ロケーション決めによって, 関数とグローバル変数のアドレスが決定される

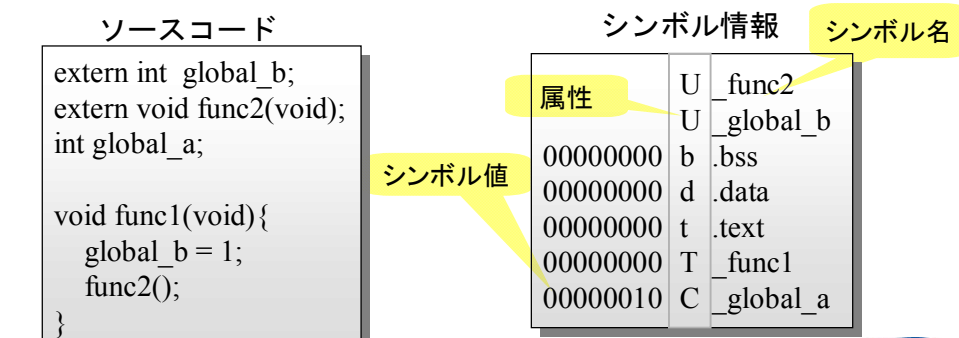


クロスリンカ : シンボル解決の実現方法

コンパイル・アセンブル後のオブジェクトコードには、
機械語の命令に加えて、"シンボル情報"が含まれている

シンボル

- ソースコードで定義されている関数やグローバル変数の名前の情報
- ソースコードで参照している他のソースコードで定義された関数や変数への参照情報(未解決シンボル)



2008/11/27

TOPPERSプロジェクト認定

84



コンパイル後のオブジェクトコードには、実際にメモリ上に配置される機械語やデータのほかにシンボル情報が含まれています。オブジェクトコード中のシンボルは確定されたものではありません。最終的にシンボルのアドレスはクロスリンカにより、ローケーションが決められた後に解決されます。オブジェクトコード中のシンボル情報はクロスリンカがローケーションを解決するための情報です。

クロスリンカ : シンボル解決の実現方法

- 各オブジェクトコードの未解決シンボルの情報をチェックして、他のオブジェクトコードでそのシンボルが定義されていないか、ライブラリで定義されていないかチェックし、定義されていればライブラリをリンク対象にする
- リンカはロケーション決めを行い、各関数とグローバル変数のアドレスを決定する
 - シンボルにアドレスが割り当てられる
- 各オブジェクトコードの未解決シンボルの情報をチェックして、未解決シンボルがあれば、参照箇所を実際のアドレスに設定する
- 実行ファイルのシンボル情報の例)

```
000081e0 T func1
000081c4 T func2
0000a6e8 B global_a
0000a6e4 B global_b
```

クロスリンカのシンボル解決手順について説明します。複数のオブジェクトコードを結合した時点で、解決されないシンボルがある場合、**UNDEFINED SYMBOL**エラーとなりリンクが完了しません。この場合、不足のオブジェクトコードを追加したりライブラリを追加して未解決シンボルを解決しなければなりません。全ての未定義シンボルが解決されるとクロスリンカは、リンク情報に従ってロケーションを決定します。ベースとなるロケーションが決定するとオブジェクトコード中のシンボル情報を使って、各シンボルのアドレスを決定します。

シンボル情報はツールによりリスト化することが可能です。GNUの場合、**objcopy**や**objdump**等のツールによってリスト化が可能です。ARMの場合は以下のコマンドとなります。

```
arm-elf-objcopy -O binary -S jsp jsp.srec
```

クロスリンカ : ROM化可能コードの生成

プログラムコードをROMに置いて動作させる方法

- プログラムコードと定数データはROMに置く
 - 定数データ: C言語でconst宣言された変数
: 文字列定数(“”で囲んだ文字列)

- 初期化されるデータは, 値をROMに置いて, ROM
プログラム実行前にRAMにコピーして使う
 - データを配置すべきアドレスと, 実行時に
参照すべきアドレスが異なる

➡ スタートアップモジュールの役割



2008/11/27

TOPPERSプロジェクト認定

86



クロスリンカによって配置された実行形式のROMに対応する部分をターゲットシステムのROMに書き込むことにより、電源の再投入により正しく実行させるには「スタートアップモジュール」を正しく設定しなければなりません。電源投入時の実行開始ではRAMの領域のデータは不定データです。そのため、最初にスタートアップモジュールが動作するように設定しておきます。スタートアップモジュールは、最初にスタックの設定を行い、次にidataセクションのデータをdataセクションにコピーします。その後、bssセクションをゼロクリアして、メインプログラムに実行権を渡します。メインプログラムの実行時はRAM上のデータは正しく設定されていますので、プログラムは正しく動作します。

組込みプログラム開発の基礎知識

1. 開発環境
2. [デバッグ環境](#)
3. 実行環境
4. コーディングルール

組込みソフトウェアのデバッグ環境

組込みソフトウェア開発には様々なデバッグ環境が存在

基礎知識：デバッガ

- バグ (bug) の発見を支援するツール

主な機能

- プログラムのロード
- プログラムの実行と停止
- プログラムの読出し(対応するソースコードの表示)
- データの読出し, 書換え(変数, メモリ, レジスタ)
- 関数(またはサブルーチン)の呼出し
- ブレークポイント(実行ブレーク, アクセスブレーク)
- ステップ動作(ソースコードレベル, マシン語レベル)
- スタック状態の読出し, バックトレース

2008/11/27

TOPPERSプロジェクト認定

88



プログラミングを行い、ビルドし実行形式に変換すれば、問題なく組み込みシステムが動くわけではありません。作成したプログラムが正しく動作するように不具合(バグ)の解析と修正を行わなければなりません。そのために、組込みソフトウェア開発には様々なデバッグ環境が存在します。バグを発見し正しく動作するかの確認を行うツールをデバッガといいます。

デバッガの主な機能は以下の通りです。

- 1) プログラムのロード
- 2) プログラムの実行と停止
- 3) プログラムの表示、機械語ではなくC言語レベルの表示が可能なものが多い
- 4) データの読み出しと書換え、データとはメモリ上のシンボルデータ、C言語中の変数、レジスタ等を指す
- 5) 関数(またはサブルーチン)の呼び出し
- 6) ブレークポイント
 - 実行ブレークとは機械語や関数の先頭や実行ステートを指定して、そこに到達すると停止させる機能
 - アクセスブレークとはデータを読み書きした時点で、実行を停止する機能
- 7) ステップ実行
- 8) スタック情報の読み出しや関数の実行履歴表示(バックトレース)

デバッグモニタ

ターゲットシステムでプログラムとして動作させ、
シリアルインタフェース等で人間もしくはホストシステムの
デバッガと通信することでデバッグを行う

ROMモニタ(対人間)

- ソースコードデバッグ機能は持たない

リモートデバッガ(対デバッガ)

- 高度なデバッグ機能をホストシステムのデバッガで実現
- ターゲットシステムにはターゲットシステムの状態を参照・設定するための最低限のソフトウェアを実行
 - レジスタ/メモリの読出し/書込み
 - 実行の開始/再開, 1ステップ実行

ROMモニタは、ターゲットボード中に書き込まれ、ボードのメモリ情報の表示や書換え、プログラムのダウンロードや実行を行うプログラムです。

ROMモニタはボードに書き込まれたプログラムのみでデバッグを行いますが、リモートデバッガはボードに書き込まれたプログラムとパソコン上のデバッグプログラムが共同でデバッグ機能を実現するデバッガです。パソコン上にはソースコードやシンボル情報がありますので、ソースコードレベルデバッグやレジスタの内容、そして実行プログラムに関連するその他の情報を表示するための機能を持っています。

ICE (In-Circuit Emulator)

- ターゲットボード上のプロセッサを外し、代わりにプロセッサをエミュレーションするICEを接続する
- プロセッサの動きをリアルタイムに監視し、必要なタイミングで動作を止める/変えることができる
- “フルICE”とも呼ばれる
 - ➡ システムを停止させないデバッグ作業を支援
 - ユーザーインタフェース部分はホスト上で動作させる



ICEはプログラムの動作状況やCPUのステータスなどをCPUに代わって監視・制御するために作られました。ICEはCPUの装着されていたソケットに、CPUの代わりのプローブを差し込んでターゲットボードと接続します。ICEはプローブを通してターゲットボードを実行させるとともに、実行情報を監視、制御、履歴の保存ができます。また、プログラムの実行を一時的に止めて、その時のメモリの内容を読み書きしたりCPUのステータスを参照したりもできます。

ICEの問題点

- プロセッサ毎に開発しなければならないため高価
 ➡ ROMエミュレータ
- ワンチップ化やキャッシュにより、チップの外部からバスが観測できない
 ➡ エバチップ (Evaluation Chip)
 ➡ オンチップデバッグ機能
- プロセッサの高速化により、プローブを指すと動作しなくなる/動作が不安定になる
 ➡ オンチップデバッグ機能

ICEはデバッグを行うには有用なツールですが、問題点もあります。

1) プロセッサを置き換えることにより、デバッグ機能を実現するためプロセッサごとにICEを開発しなければなりません。そのために高価となります。ROMエミュレータはプロセッサではなく、ROMをプローブするデバッガです。

2) ワンチップCPU等はROMやRAMがLSI内に実装されているため、ROMやRAMのインターフェイスでプローブができません。また、高速なプロセッサではキャッシュ機能があり、実際に読み出したアドレスとプロセッサの実行アドレスが一致しません。このようなプロセッサではICEの対応ができません。これを解決するデバッガとして以下の物があります。必要なアクセス情報を外に取り出すためのインターフェイスをもつデバッグ専用プロセッサをエバチップといいます。また、デバッグ専用の回路(オンチップデバッグ機能)をプロセッサ内にもつプロセッサ等があります。

3) また、プロセッサの高速化によりICEのプローブ用にデータ線の引き出しにより、干渉やノイズが発生しICEの動作が不安定となり、実用に耐えなくなる等の問題もあります。

ROMエミュレータ/エバチップ

ROMエミュレータ

- プロセッサではなく、ROMソケットにプルーブを差すことでプロセッサの動作を監視
- プロセッサの動作を止める/変えるためには、NMIを使う



エバチップ (Evaluation Chip)

- プロセッサ内部のバスを外部に引き出した、デバッグ時専用のプロセッサ

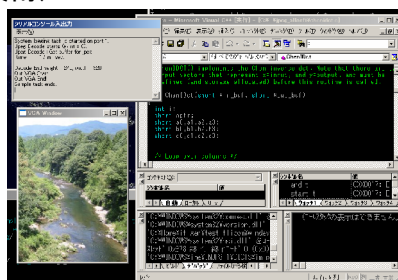
オンチップデバッグ機能

- デバッグ支援回路をチップ内に実装する
- 外部のインターフェースユニットを経由してホストパソコンから制御
- ICE(のサブセット)がチップ内に入っているイメージ



シミュレータ

- ハードウェアが完成する前にソフトウェアをテスト・デバッグしたい場合に広く用いられる
- シミュレーション速度と時間精度のトレードオフに
 - Cycle Accurateなシミュレータ(CAS, 低速)
 - 命令セットレベルのシミュレータ(ISS, 中間)
 - C言語レベルのシミュレータ(高速, 時間精度は無視)
- コシミュレーション(ハードウェアとソフトウェアを同時にシミュレーションするための技術)

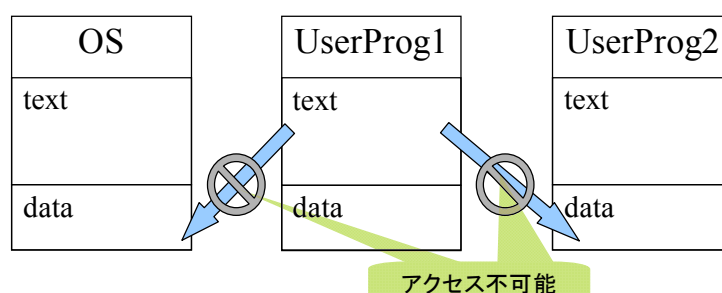


組み込みプログラム開発の基礎知識

1. 開発環境
2. デバッグ環境
3. [実行環境](#)
4. コーディングルール

汎用PC用ソフトウェアのランタイム構造

- 各プログラムは、独立したメモリ空間(論理空間)で動作
 - メモリマネジメントユニット(MMU)によるマッピングとメモリ保護
 - 他のプログラムやOSのメモリ空間へのアクセスは不可能
 - プログラムに不具合があり、正しくないメモリ空間にアクセスした場合は、アクセス違反で終了されるだけで、システム全体に影響は及ぼさない



2008/11/27

TOPPERSプロジェクト認定

96



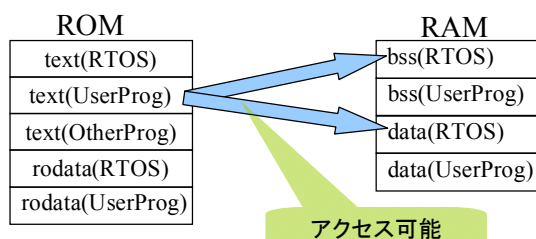
汎用PCのアプリケーションは、保護されたメモリ空間で動作します。この実行単位はプロセスと呼ばれ、他のプロセス(アプリケーション)やOSの実行領域をアクセスできません。もし、プロセス上のプログラムに不具合があり、他のプロセスやOSの領域に対して不当なアクセスを行っても、OSはプログラムのアクセス違反を検知しプロセスを停止させます。システム全体の安全性は保たれます。

しかし、OS自体のプログラムに不具合があれば簡単にシステムの実行を阻害することが可能です。OS上のプログラムは高い信頼性が求められます。

組込みソフトウェアのランタイム構造(一般的な)

1リンクモデル

- ユーザープログラムがリアルタイムOSを含む他のプログラムやデータとミックスされて配置され、それぞれは保護されていない
 - ➡ MMUを用いたメモリ保護は実行オーバーヘッドが大きく、リアルタイム性の弊害となる
- ユーザープログラムは容易にリアルタイムOSのデータを破壊可能
 - ➡ システム内のソフトウェアは信頼できるという前提
- メモリアクセス関連のデバッグが困難



2008/11/27

TOPPERSプロジェクト認定

97



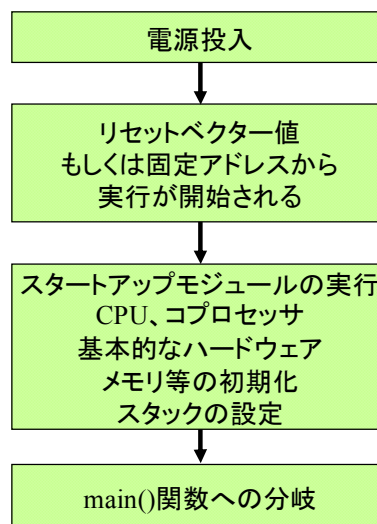
一般的な組込みソフトウェアの構造について説明を行います。一般的な組込みソフトウェアは、1リンクモデルを採用しています。システム全体(アプリケーション+システムサービス+リアルタイムOS)を1つのロードモジュールにリンクして実行形式にしています。これを1リンクモデルといいます。システムサービスとはミドルウェアつまり、デバイスドライバなどを含みます(ここではシステムサービス+リアルタイムOSをプラットフォームと呼びます:プラットフォームがPC等のOSにあたります)。この1つになっている実行形式を、ROM(またはRAM)において実行します。

この方法ですと、アプリケーションからリアルタイムOS等のサービスコールを関数呼び出しの形で簡単に呼び出すことができます。その反面、アプリケーションから簡単にプラットフォームのデータ領域を破壊できてしまいます。そのため、搭載するプログラムを作成するときはこの点に注意して作成しなくてはなりません。

スタートアップモジュール

C言語のmain関数が実行される前に実行されるプログラム

- 標準的なコンパイラは、何も指定しないと標準のスタートアップモジュールをリンクする
- スタートアップモジュールの主な処理内容
 - ハードウェア依存の初期化
 - スタックポインタの初期化
 - dataセクションの初期化 (ROMからRAMにコピー)
 - bssセクションの初期化 (0に初期化する)
 - (C++の場合)コンストラクタとイニシャライザを実行する
 - main関数を呼ぶ



2008/11/27

TOPPERSプロジェクト認定

98



次にスタートアップモジュールについて説明を行います。組込み機器に電源を投入すると、main関数から始まるように思えますが、実際にはmain関数の前にスタートアップルーチンという処理がmain関数が動作できる環境を設定します。

そのスタートアップモジュールはハードウェア依存の初期化、スタックポインタの初期化、dataセクションの初期化、bssセクションの初期化等を行った後で、最後にmain関数の呼び出しを行います。このプログラムはハードウェア依存ですので、ターゲットによって設定の仕様が異なります。

今回の演習で使用するM16Cでのスタートアップルーチンでどのようなことをしているかを解説します。

- 1) フラグレジスタの設定
- 2) 割り込みスタックポインタのセット
- 3) プロセッサモードの設定
- 4) システムクロックの設定
- 5) 端子割り当ての設定

今回のボードは、64ピンか80ピンの2通りあります。このソフトウェアが何ピン用かを指定する必要がありますので、ここで指定しています。

- 5) データ領域の初期化

初期値ありの変数は、ROM領域に置いた初期値をコピーします。初期値なしのデータはここでゼロに初期化します。

その後、main関数へ飛ぶような指示が書かれています。

スタック

プログラム中の一時的な変数の保存に使われるメモリ領域
LIFO方式

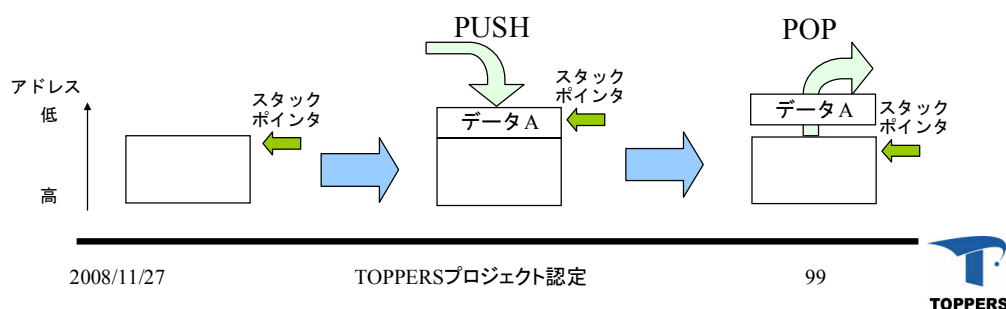
- 後入れ先出し(LIFO : Last In First Out)でデータを扱う
- Push操作, Pop操作, 成長方向は環境依存

スタックポインタ

- スタックを管理するためのレジスタ, スタックの先頭番地を示す
- スタックの先頭番地にデータが入っているかは環境依存

スタック領域

- プログラムがスタックとして使用可能な領域
- セクションとして確保したり, グローバル配列としてdata領域に確保する



スタックは、プログラム中の一時的な変数の保存に使われるメモリ領域です。

標準的なスタックの機構は、LIFO (Last In First Out) 方式と呼ばれ、最後に保存したデータを先に取り出す方式です。例を挙げて説明すれば、机の上に書類をどんどん積んでいって、一番上にあるものから処理をしていくといったイメージです。スタック領域に、データを入れる操作をPush操作, データを取り出す操作をPop操作といいます。Pushした時にメモリ上にデータがどちら方向に伸びていくかは、環境依存になります。

スタックポインタ

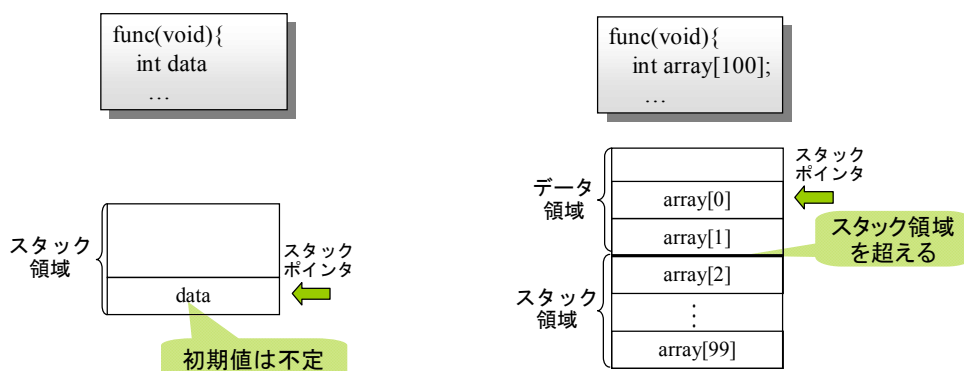
スタックポインタは、スタックを管理するためのレジスタです。スタックの先頭番地を示しています。スタックポインタがさしているところにデータを入れるのか、それとも、スタックポインタが指しているところからデータを取り出かは環境に依存します。C言語で記述する場合、関数が呼び出されるとスタックポインタが更新されて関数用のデータ領域がスタック上に確保されます。関数中の変数は、この領域が割り当てられます。関数の終了時、スタックポインタが、もとに戻されてデータ領域が開放されます。

スタック：ローカル変数

ローカル変数はスタックに確保される

スタックオーバーフロー

- スタック領域のサイズを超えるローカル変数を確保すると、隣接する領域を破壊してしまう



2008/11/27

TOPPERSプロジェクト認定

100



C言語等の関数中にローカル変数を宣言すると、スタックにデータ領域が確保されます。汎用OSのスタック領域は仮想記憶上に関数が要求する分だけ領域をOSが割り当ててくれます。組込み環境のスタック領域はRTOS上に初期設定して確保されます。そのため、関数中の自動変数を多く作りすぎるとスタック領域をオーバーしてデータ領域をつくってしまい、別のデータ領域を破壊する可能性がありますので、十分な注意が必要です。スタック領域が不足して発生する問題をスタックオーバーフローといいます。

スタック：関数呼び出し

関数呼び出しの際、引数、リターンアドレスや
戻り値をスタックに入れるプロセッサがある

- スタックを使うかどうか、使う場合の使い方は、プロセッサやコンパイラによって異なる

➡ Application Binary Interfaceによって規定

```
func1(void){
...
i = add(2, 3);
...
}
```

```
int
add(int arg1, int arg2){
return arg1 + arg2;
}
```

add()呼び出し時 スタック
 ポインタ

引数2 : 3
引数1 : 2
addからの戻り番地
func1の使用領域

add()リターン時

引数2 : 3
引数1 : 2
戻り値 : 5
func1の使用領域

使用
済み

スタック
ポインタ

2008/11/27

TOPPERSプロジェクト認定

101



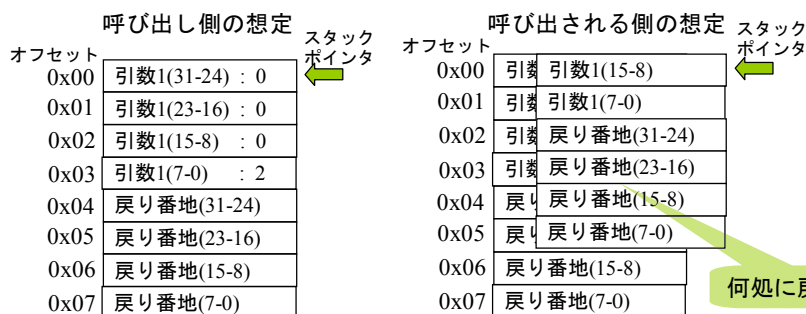
C言語等で関数呼び出しの際に、引数やリターンアドレスをスタックを用いて行う場合があります。これらをレジスタを用いて行う場合もあります。どのように引渡しを行うかは、プロセッサやコンパイラによって定められています。基本的にレジスタ渡しの方が処理性能がアップしますが、引数が多すぎて割り当てのレジスタの数が足りない場合、足りない部分をスタック渡しにするのが一般的です。関数呼び出し時のインターフェイスはApplication Binary Interfaceによって規定されています。

スタック：関数のプロトタイプ宣言

- 関数のプロトタイプ宣言を怠ると....
 - コンパイラによる引数や戻り値のチェックが行えない
 - 暗黙の型変換が行われず、予期しない挙動になる

```
func1(void){
  ...
  i = square(2);
  ...
}
```

```
int
square(short arg1){
  return arg1 * arg1;
}
```



2008/11/27

TOPPERSプロジェクト認定

102



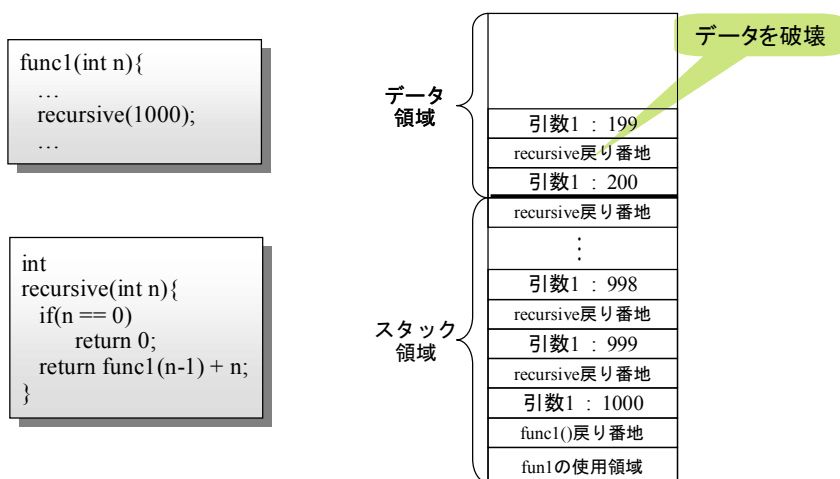
関数のプロトタイプ宣言の重要性について説明を行います。上記の例ですと、**func1**関数の中で、**square**関数が呼び出されています。**func1**関数は**square**関数のプロトタイプ宣言がないと、戻り値も引数も**int**型であると判断します(**int**型は4バイトとします)。**func1**関数は、**square**関数を呼び出すときに、戻り番地をスタックに積み、その上に引数を**int**型4バイトと仮定して、実引数を積みます。

しかし、呼び出された**square**関数は、自分としては引数は**short**型(2バイト)のつもりですから、スタックポインタのあるところから2バイト分を仮引数として処理を行います。ここで、正しく仮引数が受け取れない可能性があるだけでなく、処理終了後どこに戻るかを示す戻り番地が格納されている場所も、誤って認識してしまっ、プログラムが暴走する可能性があります。

実際には、GNU等の標準的なC言語を用いていれば**char**型や**short**型は**int**型に補正されて渡されますので、こんなことはおきませんが、作成したプログラムをどのようなコンパイラでも問題なく、動作させたいならプロトタイプ宣言をきちんと行っておくことで、問題の発生を防止できます。

スタック：再帰呼び出し

関数を呼び出すたびにスタックを消費するため、
再帰の深さとスタックサイズに注意が必要



2008/11/27

TOPPERSプロジェクト認定

103



関数処理中に、さらに自分自身を呼び出す処理を再帰呼び出しといいます。関数を呼び出すたびにスタックを消費するため、再帰のレベルが深くなればなるほどスタックを消費します。一般的に再帰呼び出しをする関数は、再帰のレベルがどんどん深くなりがちなためスタック領域を消費し、意図せずデータ領域を破壊してしまう可能性があります。そのため、実行時の利用スタックサイズが予測できない再帰呼び出しは、組込みシステムでは使用しない方が良くとされています。(MISRA-C)

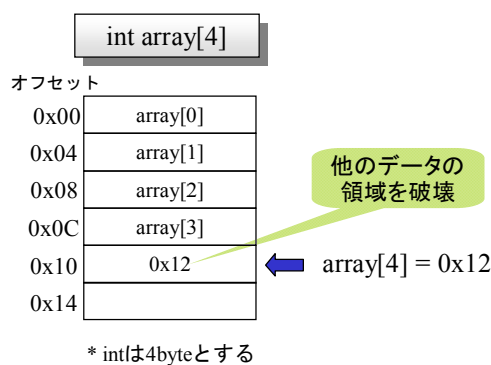
前述の通り、汎用OS上のプロセスでは、スタックサイズはOSでアロケートしてくれますので、このようなことを気にせずプログラミングが可能です。

配列

メモリ上に一次元に確保される

- Cコンパイラはインデックスの正当性・妥当性の検証は行わない
- 範囲外のインデックスでアクセスを行うと、他のデータを破壊してしまう

→ バッファオーバーフロー/ラン



2008/11/27

TOPPERSプロジェクト認定

104



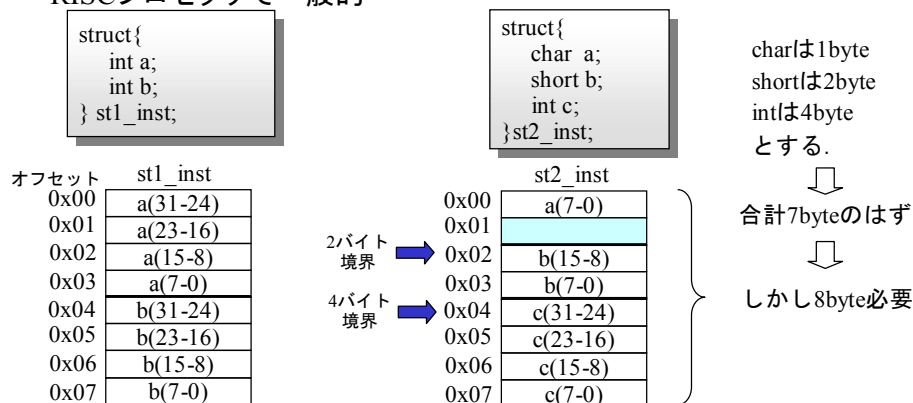
C言語コンパイラは、配列のインデックスの正当性・妥当性の検証は行いません。つまり、配列の先頭アドレスしか管理していません。上記の例では、配列`array`は要素数は4つ分確保されます。その領域にアクセスするためのインデックスは、0, 1, 2, 3です。プログラム中でインデックスを4にしてアクセスしてしまってもコンパイルエラーを発生しません。このように、インデックスの設定の間違いで確保した領域以外の部分を破壊してしまう可能性があります。

構造体

プロセッサのアライメントに従い配置される

アライメント

- 4バイトデータは4バイト境界に、2バイトデータは2バイト境界に置いてアクセスする必要があるプロセッサがある
- RISCプロセッサで一般的



2008/11/27

TOPPERSプロジェクト認定

105



データはプロセッサのアライメントに従い配置されます。アライメントとは、データをどのように配置するかを規定するもので、プロセッサに依存します。たとえば、4バイトデータは4バイト境界に、2バイトデータは2バイト境界に配置するといった規則です。

上記の構造体の例で解説します。st2_inst構造体は、メンバーが3つあります、char型のa、short型のb、int型のcです。では、この構造体はメモリ上にどれだけの領域を必要とするでしょうか？単純に計算すると(int型を4バイト、short型を2バイトとします)char型(2バイト) + short型(2バイト) + int型(4バイト)で7バイトとなります。しかし、上記アライメントを適用しますと、short型のメンバーは2バイト境界に配置されますので、char型のメンバとの間に1バイトあけて配置されます。従って、この構造体が必要とする構造体の領域は8バイトとなります。

char型2つとshort型1つの合計3つのメンバを持つ構造体を定義するときには、メンバの並べ方によって必要となる領域の大きさが変わる可能性があります。

例1) 以下の場合6バイト

```
char
short
char
```

例2) 以下の場合4バイト

```
char
char
short
```

自分が使用するプロセッサのアライメントは確認する必要があります。

C言語から周辺デバイスへのアクセス(1/2)

ポインタを用いてアクセスする
(メモリマップドI/Oの場合)

ポインタ

- アドレスを指し示し, 単項演算子*を使ってアクセスすることにより特定のアドレスへのアクセスが可能

特定アドレスへのアクセス

- デバイスのレジスタは特定のアドレスを持っている
- ポインタ変数にアドレスを代入し, そのポインタ変数を使うことでアクセス可能
- アドレス値をポインタ型へキャストする

aのポインタ値の取得

```
int a;
int *p_a = &a;
```

アドレス 変数名

0x2000	a
0x2004	b
0x2008	c
0x200C	d

2004番地へのアクセス

```
int *p_b = 0x2004;
*p_b = 1;

*(int*)(0x2004) = 1;
```

キャスト

2008/11/27

TOPPERSプロジェクト認定

106



ポインタとは、アドレスを格納する変数です。そして、間接演算子(*)を用いて、自分が格納しているアドレスが指している先にアクセスすることができます。

ポインタ変数は宣言をしたあと、初期化をしないとどこを指示しているかは不定です。上記の例では、ポインタp_aは、変数aのアドレスで初期化されています。&変数名で、その変数が割り当てられているアドレスを意味します。これにより、int *p_a = &a;で、ポインタ変数p_aを宣言して変数aを指しています。これによって、ポインタ変数p_aを用いて、変数aの領域にアクセスできるようになりました。

同じように、int *p_b = 0x2004; これで、ポインタ変数p_bは0x2004番地を指しています。そこで、*p_b=1とするとp_bが指している先、つまり0x2004番地に1を代入することになります。

このポインタを使用して特定のアドレスへのアクセスが可能になります。組込みシステムではその特性上、デバイスレジスタへの値の書き込みや読み込みを行います。デバイスレジスタというのは特定のアドレスを持っていますので、ポインタ変数にデバイスレジスタのアドレスを格納しそのポインタを使って、デバイスレジスタにアクセスを行います。

C言語から周辺デバイスへのアクセス(2/2)

型とアクセスサイズ

- デバイスのレジスタにはアクセスサイズが定まっている
- int : 4byte, short : 2byte, char : 1byte とすると, intポインタは4byteアクセス, shortポインタは2byteアクセス, charポインタは1byteアクセスとなる

例) 0x1000 にあるアクセスサイズが2byteのデバイスレジスタの読み込みと書き込み

読み込み

```
data = *((short *)(0x10000));
```

```
short *p_reg = 0x10000;
data = *p_reg;
```

書き込み

```
*((short *)(0x10000)) = data;
```

```
short *p_reg = 0x10000;
*p_reg = data;
```

2008/11/27

TOPPERSプロジェクト認定

107



ポインタは一般的にアドレスと認識されますが、厳密には違います。ここで、int型:4バイト、short型:2バイト、char型:1バイトとします。

たとえばint型のポインタが1000番地を指していたとします。そのポインタを使用して、1000番地から4バイトの領域にアクセスできるということです。short型のポインタが1000番地を指していたとすると、そのポインタを使用して1000番地から2バイトの領域にアクセスできるということです。このようにポインタには型があり、その型によってポインタを使用してアクセスできる領域の大きさが異なります。

上記の例では、0x1000番地から始まる16ビットのレジスタへの読み込みと書き込みの仕方を示しています。

以下の2つの記述は同じ内容です。

- 1) `data = *((short *)(0x10000));`
- 2) `short *p_reg = 0x10000;`
`data = *p_reg;`

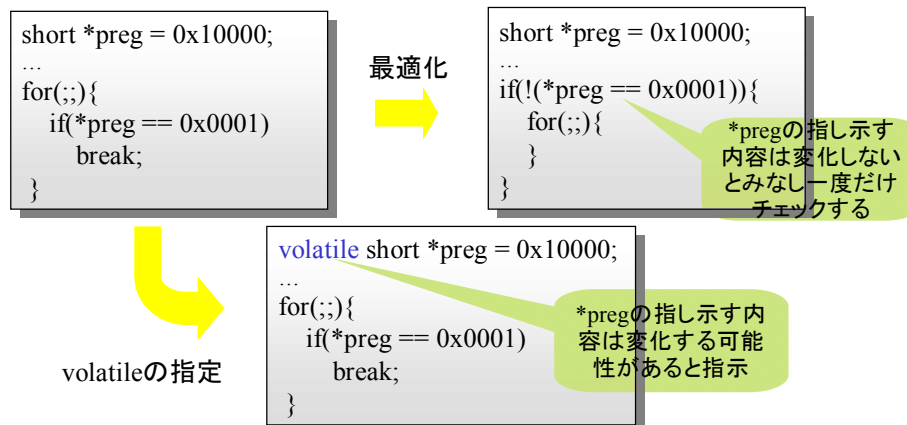
つまり、0x10000というアドレスをp_regというshort型のポインタに格納し、ポインタp_regを使えば、0x1000番地から2バイト分のレジスタにアクセスすることができます。

volatile : 必要性

コンパイラの最適化を抑制するC言語の型修飾子

- メモリアクセスのパターンを保存する

例) 0x10000にあるレジスタの内容が0x0001になるまで待つ



2008/11/27

TOPPERSプロジェクト認定

108



組み込みプログラミングをする際には、コンパイラの最適化を抑制する`volatile`修飾子をつける必要がある場合があります。コンパイラの最適化とは、プログラムの実行効率を上げるために実行ファイルを効率化し、実行時間やメモリ使用量を最小化するように調整する機能のことです。つまり、冗長なコードだとコンパイラが判断すると処理上不要なプログラムを変更してしまいます。内容が不定なレジスタを行う場合、問題となることがあります。上記例を見てみましょう。

```
short *preg = 0x10000;
for(;;){
    if(*preg == 0x0001)
        break;
}
```

このプログラムは、`for`ループの中でポインタ`preg`の指している先が、`0x0001`かどうか毎回チェックします。しかし、ポインタ`preg`の指している先が変更されるソースコードはどこにもありません。このように明示的に、ポインタ`preg`の指している先が変更されないと判定すると、最適化により、ループ処理で毎回、確認する必要はないと判断し、以下のようにソースコードを変更します。

```
if(!(*preg == 0x0001)){
    for(;;){
        // Empty loop body
    }
}
```

つまり、`for`ループに入る前に一回ポインタ`preg`の指している先を確認すれば、その後変更されるはずはないので十分と判断するわけです。これが組み込みプログラムで行われると不都合な場合があります。たとえば、`preg`がレジスタを指していたとします。プログラム上で明示的に変更されなくても、システム実行中にレジスタが変更されることが十分考えられますので、ループの中で毎回中身を確認する必要があります。このように最適化されては困るような場合には、`volatile`修飾子を指定して変数を宣言すると、最適化が抑制されます。

volatile : 共有変数

マルチタスク/スレッド, 割込みを使用する場合は,
グローバル変数にもvolatile指定が必要なことがある

- コンパイラはコンパイル対象以外の関数が共有変数を書き換えることを想定していない
 - 割込みハンドラ, 他のタスク・スレッドでの書き換えが発生する可能性がある
 - ループ中で複数回アクセスする場合に最適化の対象となる

例) グローバル変数pregの内容が1になるまで待つ

```
volatile short preg; /* グローバル変数 */
...
for(;;){
    if(preg == 1)
        break;
}
```

グローバル変数にも**volatile**指定が必要な場合があります。必要な理由は、前ページの場合と同じで、他のタスクが変数を書き換える可能性があるからです。コンパイラはコンパイル対象以外の関数が共有変数を書き換えることを想定してません。したがって、マルチタスクや割込みを使用してコンパイル対象以外から共有変数を書き換えられる可能性がある場合は、**volatile**指定をする必要があります。明示的にプログラムで変更しなくても、変更される可能性があるので最適化しないでほしいということを、コンパイラに伝える必要があります。

volatile : 記述方法

- ポインタ変数

```
volatile *char preg;
*preg = 0x01;
```

- キャスト

```
*((volatile char *)0x1000) = 0x01;
```

- ポインタの volatile 宣言時の注意

- device_registerの指す先がvolatile

```
volatile char *device_register
```

- device_register自身がvolatile

```
char *volatile device_register
```

2008/11/27

TOPPERSプロジェクト認定

110



volatile指定子が必要な理由は、前ページまでで解説してきました。ではどのように宣言すればいいのでしょうか？

volatileの指定は場所により、不定の指定が変わります。

1) ポインタが際してるデータが不定の場合

```
volatile char *device_register;
```

ポインタdevice_registerの指している先が最適化抑制対象です。

2) ポインタの値自体が不定の場合

```
char *volatile device_register
```

この記述でも、コンパイルエラーにはなりませんが、意味が異なってきます。こちらの記述ですとdevice_register自身がvolatile指定となります。レジスタの中身でなくて、レジスタのアドレスが格納されているポインタ変数の中身(アドレス)が、暗黙的に変わる可能性があると言っているわけです。そのため、意図した最適化が行われません。

ビット演算

ビット関連の演算(&, |, ~, ^, <<, >>)は,
デバイスレジスタの操作で多用する

特定ビットのみの取り出し

- 取り出したいビットパターンとの論理積(AND)により生成
- 0x11のビット5,4(0x30)のみ残したい
 $0x11("00\textcolor{blue}{01} 0001") \& 0x30("00\textcolor{blue}{11} 0000") = 0x10("00\textcolor{blue}{01} 0000")$

特定ビットのクリア

- クリアしたいビットパターンの否定との論理積(AND)により生成
- 0x21のビット5,4(0x30)をクリアしたい
 $\sim 0x30("00\textcolor{blue}{11} 0000") = 0xCF("11\textcolor{blue}{00} 1111"),$
 $0x21("00\textcolor{blue}{10} 0001") \& 0xCF("11\textcolor{blue}{00} 1111") = 0x01(00\textcolor{blue}{00} 0001)$

2008/11/27

TOPPERSプロジェクト認定

111



デバイスレジスタは、ビットごとに意味を持ちます。そのため、組込みプログラミングでは、ビット演算を多用します。

たとえば、特定ビットのみの取り出し方法ですが、取り出したいビットパターンとの論理積(AND)により生成することができます。8ビットレジスタのうち、今アクセスしたいビットが、4ビット目5ビット目だとします。この場合取り出したいビットパターン0x30(0011 0000)との論理積をとります。

具体例で説明します。対象レジスタの値が現在0x11(0001 0001)だったとします。今回必要なのは、ビット5、4です。したがって、現在の値0x11(0001 0001)と、取り出したいビットパターン0x30(0011 0000)の論理積をとると、ビットパターンで1が立っているビットの情報だけ取り出すことができます。つまり、AND演算結果は、0x10(0001 0000)となります。

次に、特定のビットのみクリアしたい場合があります。その場合は、クリアしたいビットパターンの否定との論理積(AND)により生成します。現在のレジスタの状態が0x21(0010 0001)で、ビット5、4をクリアするとします。

まず、クリアしたいビット5、4の否定は、 $\sim 0x30(0011 0000) = 0xCF(1100 1111)$ となります。これと現在のレジスタの状態との論理積(AND)をとります。

$0x21(0010 0001) \& 0xCF(1100 1111) = 0x01(0000 0001)$
となります。

割り込みプログラミング

- 割り込みのプログラムを作成するには
 - 割り込みの概念の理解
 - プロセッサアーキテクチャの理解
 - 割り込みコントローラのアーキテクチャの理解(結線)
 - コンパイラの理解
- が必要

組み合わせの多様性

- 組み込みシステムでは、多くの種類のプロセッサとコンパイラを扱うことになるため、組み合わせ毎にプログラミングは異なる
- ➡ 基本的な考え方はどの環境でも同じ

割り込みのプログラムを作成するには、いろいろな知識が必要となります。まず、割り込み概念の理解、使用するプロセッサアーキテクチャの理解、割り込みコントローラのアーキテクチャの理解、コンパイラの理解などです。

組み込みシステムでは、多くの種類のプロセッサとコンパイラがあります。その組み合わせごとにプログラムは異なります。環境ごとに設定すべき項目、場所が異なりますが、基本的な考え方は同じです。

プロセッサの割込みアーキテクチャの例：M16C

- 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くれ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、あとはソフトウェア(プログラム)で処理しなければならない

2008/11/27

TOPPERSプロジェクト認定

113



割込みを受けつけた後に、プロセッサが何をするかをここで説明します。M16Cの場合はプロセッサがいろいろな割込み処理手順を代行してくれます。以下の1)～9)はM16Cが処理を行います。

- 1) どの割込みが入ってきたか判断する。
- 2) 該当する割込みの割込み制御レジスタIRビットには、割込みが入ったことにより1が立っているためクリアします。
- 3) 現在のフラグレジスタを退避します。
- 4) フラグレジスタの割込み許可フラグ(I)を0にクリアして、割込みを禁止します。割込みを受けつけた後に再び受けつけないようにしています。
- 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’にして割込み用のスタックを有効にします。これにより割込みの処理はタスク処理とは違うスタックを使用します。
- 6) CPU内部の一時レジスタをスタックに退避します。
- 7) プログラムカウンタをスタックに退避します。現在のプログラムカウンタを退避することで、割込み処理が終了したらどこに戻るかが保存されます。
- 8) プロセッサ割込み優先レベルに受けつけた割込み優先レベルを設定します。
- 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

簡単にまとめると、割込みを禁止し、プロセッサ割込み優先レベルを設定し、割込みベクタテーブルに設定した関数へジャンプします。

割込みハンドラ

割込み受付時はレジスタの保存(RISC型は割込み要因の判定),
ハンドラ終了時は復帰命令の呼び出しが必要

アセンブリ言語による記述

- 入出力をアセンブラで記述し, ハンドラ本体はC言語で記述する場
合が多い
- 必要なレジスタの保存をした後, C言語の関数を呼び出し, C言語
の関数が終了したら, 復帰命令を呼び出す
- コーリングコンベンションに従って呼び出す必要がある

C言語による記述

- `pragma`により指定することにより, レジスタの保存, 復帰命令の呼
び出しをコンパイラが関数の前後に入れる
- 最低限の処理しか行わないので, 利用できない場合もある

```
#pragma interrupt
void
int_handler(){
....
}
```

割込み受付時はレジスタの保存、ハンドラ終了時は復帰命令の呼び出しが必要など、通常の関数の呼び出し、復帰とは違う処理が必要です。

この入出力の処理は、スタックの保存やレジスタの保存など、プロセッサに密着した処理になりますので、アセンブリ言語による記述が必要になります。このレジスタの保存をした後、C言語の関数を呼び出し、C言語の関数が終了したら復帰命令を呼び出します。

C言語による記述

割込みハンドラとして使用する関数を、`pragma`指定することにより、レジスタの保存、復帰命令の呼び出しをコンパイラが関数の前後に入れてくれます。

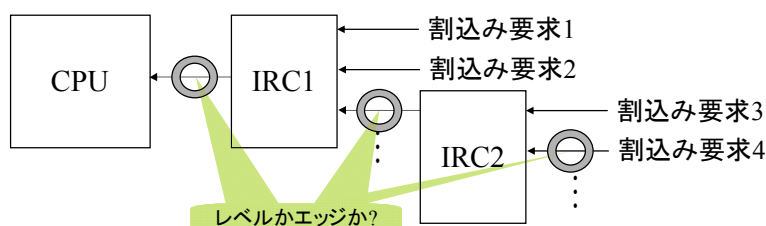
割込み要因のクリア

割込みの要求方法

- エッジトリガ
 - 割込みコントローラーの要求レジスタをクリア
- レベルトリガ
 - 割込みのソースとなるデバイスの割込み要求をクリア
 - クリアの条件は様々. ビットのクリア, データの読み込み, 書き込み

割込み周りのアーキテクチャ

- IRCのアーキテクチャ, IRCがネストしている場合も



2008/11/27

TOPPERSプロジェクト認定

115



割込みは処理後、要因クリアする必要があるものがあります。これを行わないと、同一の内容の割込みが割込み処理後、連続して発生します。

割込みの要求には、エッジトリガとレベルトリガがあります。エッジトリガは割込み信号の変化で割込み判定を行います。この場合、割込みコントローラーの要求レジスタをクリアすることにより、割込み要因のクリアが行えます(エッジトリガの場合、割込みクリアを行わなくても良い回路もある)。レベルトリガは、割込み信号がハイレベルの場合に、割込みと判定して割込みを発生します。したがって、ソースとなるデバイスの割込み要因をクリアして、信号をローレベルに変化させる必要があります。信号変化に時間がかかり、割込み処理が終了しても、ローレベルに変化していない場合、再度割込みが発生する場合がありますので注意が必要です。

割込み周りのアーキテクチャで、複数の割込みをまとめて割込みを発生させるものがあります。この場合、割込み後のプログラムで割込みの要因判定を行う必要があります。

ベクタテーブルの実現

割込みベクタテーブル

- ベクタテーブル用のセクションの作成
- 割込み要因毎の割込みハンドラの先頭番地を登録したテーブルを作成

RISC型割込み

- 特定の関数(通常, アセンブラ関数)をプロセッサで定められたアドレスに配置するようにする

ベクターテーブルの例

```
.section vector
.global vector_table
vector_table:
    .long handler1 /* オフセット0 */
    .long handler2 /* オフセット1 */
    .long handler3 /* オフセット2 */
    :
```

割込みベクタテーブルを作成する必要があります。その作成方法は、プロセッサにより違います。概要としては、ベクタテーブル用のセクションを作成します。そのセクションに割り込み要因ごとの割込みハンドラの先頭番地を登録したテーブルを作成します。

RISC系のプロセッサでは、ベクタテーブル型ではなく、割込み時、特定のアドレスから実行を始めるものが多く、特定の関数をプロセッサで定められたアドレスに配置する必要があります。

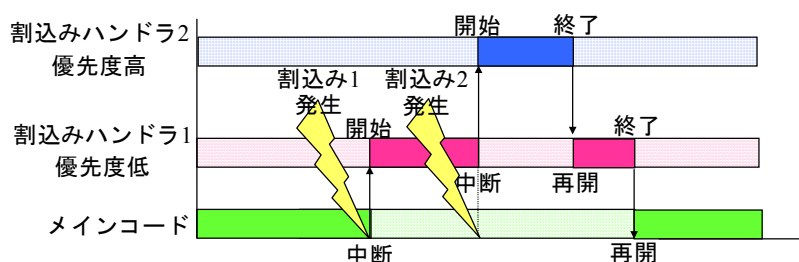
多重割込み

割込みを受け付けた直後

- 一回目の割込みと二回目以降の割込みで処理が異なる場合があるので、どちらか判定する
例) スタックポインタをソフトウェアで変更しなければならない場合

割込みを許可する前

- 割込みマスク、割込み優先度を適切に設定する
- 割込みを受け付けるとプロセッサで上書きされるレジスタ(退避レジスタ等)を保存する



2008/11/27

TOPPERSプロジェクト認定

117



割込みを実行中に、その割込みよりレベルの高い割込みが発生し、ネストして割込み処理が発生する割込みを多重割込みといいます。

多重割込みを受け付けた直後の処理で気をつけないといけないことがあります。1回目の割込みと2回目以降の割込みで行わなくてはいけない処理が異なる場合があります。たとえば、通常の処理で使用しているスタックと、割込み時に使用するスタックが異なるプロセッサがあります。その場合は、1回目の割込みでは通常使用しているスタックから、割込み処理で使用するスタックに切り替える必要があります。2回目の時はすでに割込み用のスタックを使用しているので切り替える必要がありません。このようにスタックポインタの切り替えをソフトウェアで行うような場合には注意が必要です。

また、割込みを受け付けるとプロセッサによって、プログラムカウンタ(PC)やスタックポインタ(SP)を別のレジスタに退避させるものもあります。2回目の割込み時には、そこにすでに1回目の割込みで退避されたPCとSPが収まっていますので、上書きしないように保存しておく必要があります。

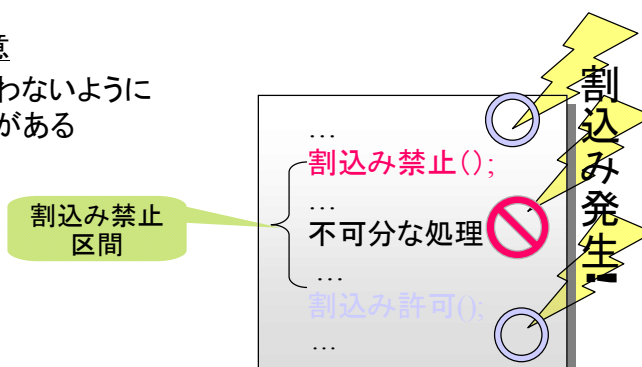
割込み禁止区間

割込み禁止の必要性

- デバイスレジスタの操作
 - 一定時間内に複数のレジスタを操作しなければならない場合
- 割込みハンドラと共有する変数の操作
 - 特に複数変数の整合性を保たなければならない場合

割込み禁止に関する注意

- 割込み応答性を損なわないように適切に設定する必要がある



2008/11/27

TOPPERSプロジェクト認定

118



割込み処理中に割込み禁止の必要性がある区間があります。

1) デバイスレジスタの操作時

一定時間内に複数のレジスタを操作しなければならない場合。また、一連のレジスタ操作が中断されずに行われたい場合などに、その処理中は割り込み禁止とします。割込み禁止区間は、共有資源を操作する時などに必要です。

2) 割込みハンドラと共有する変数の操作

特に複数変数の整合性を保たなければならない場合等、同じ共有変数を処理する割込みハンドラが起動しないように割り込み禁止区間を設定する必要があります。

割込み禁止区間をあまり長く設定すると、その分割込み応答性が悪くなります。そのため、割込み禁止区間には、どうしても割り込み禁止で行わなければならない処理のみ記述し、できるだけ早く割込み禁止を解除するプログラムを作成する必要があります。

組込みプログラム開発の基礎知識

1. 開発環境
2. デバッグ環境
3. 実行環境
4. コーディングルール

コーディングルール(プログラミングガイドライン)

ソフトウェア品質を保つために
プログラム開発(記述)で守るべきルール

ソフトウェア品質

- 理解しやすく管理しやすい(可読性が高い)
 - 複数の人間により開発・保守される
- 信頼性が高い(バグがない)

コーディングルールの例

- 命名規則
- コメントの記述方法やインデント
- 問題が発生しやすい記述の使用を避ける
 - C言語のgoto文

2008/11/27

TOPPERSプロジェクト認定

120



コーディングルールは、ソフトウェア品質を上げるために必要なものです。ソフトウェア品質とは何でしょうか？理解しやすく、可読性が高く、保守、テスト、デバッグ、改造、移植がしやすい。バグがなく信頼性が高いことなどが上げられます。

コーディングルールをプログラムに適応させることにより、命名規則を定めることにより可読性の高いソフトウェアを書くことができます。また、データ型も表すような命名法にしておけば、データ型の不一致なども変数名、関数名から容易に見えます。

問題の発生しやすい記述の使用をやめるという方針もあります。たとえば、goto文などはソースコードの解析が難しくなり、デバッグが難しくなります。また、第三者が見た場合に、ソースコード解析が難しいだけでなく、バグを含んでいても修正しづらいといったことが考えられます。このような規則を作ることで、バグの少ない、誤用されにくいプログラムを作成することができます。

C言語の危険性

C言語は高い能力をもち自由度が高い
プログラム言語だが、危険性も高い

ポインタ

- ポインタ値の妥当性はチェックしない
 - バッファオーバーラン

配列

- インデックスの妥当性をチェックしない

言語仕様中の未規定・未定義事項

- コンパイラやターゲットによって挙動が異なる

例) int型のビット幅

構造体のフィールドの配置

char型が符号ありかなしか

2008/11/27

TOPPERSプロジェクト認定

121



組み込み開発において、C言語は広く普及した言語といえます。C言語は自由度が高く、どんな記述もできて便利な反面、危険性も高いといえます。コンパイル時や実行時のチェック機能が弱く、機能安全においては推奨できないといった評価がされています。

たとえば、ポインタが指している先の領域の妥当性まではチェックしていません。ポインタで、あるアドレスを指定して、そこから連続した領域にアクセスするような記述は使用してはいけない領域までアクセスしてしまう可能性があります。

同じような理由で、配列も気をつける必要があります。インデックスの妥当性をチェックしていないので、配列の領域以外をアクセスしてしまうプログラムを書いてしまう可能性もあります。重要なデータを破壊してしまう可能性もあります。

また、C言語の仕様として未規定・未定義の事項があります。これらは環境によって挙動が異なりますので、環境に依存するようなプログラミングをする必要があります。

たとえば、int型のビット幅、構造体のメンバーの配置(アライメント)、単なるchar型が符号ありかなしか。といったことが挙げられます。

このような、記述するとC言語を使わない方が良いといった内容になってしまいますが、C言語の自由度はいろいろなメリットがあります。特に、プラットフォームにあたる部分のプログラムを記述する場合、種々の恩恵を受けられます。このような部分のプログラムの開発には繊細なプログラム設計が必要となります。

メトリクス

ソースコードの特性を定量的に表現したもの

- ソースコードの品質管理・評価に用いられる

- メトリクスの例

- 経路複雑度

- 1つの関数の中にある, 条件分岐の数に1を加えたもの

- ネストの深さ

- 制御構造の入れ子の数

- コード・サイズ

- 関数の行数

メトリクスとは、ソースコードの特性を定量的に現したものです。静的なメトリクスと動的なメトリクスがあります。特に、静的なメトリクスはツールを使って簡単に計測できますので、ソフトウェアの品質評価や管理に使用されています。

コーディングルールの例 : MISRA-C

高信頼性を目的とした,
C言語プログラミング開発のガイドライン

- 自由度の高いC言語を使い品質の高いプログラムを開発するためのルール集
 - 危険性の高い記述を回避する
- 欧州の自動車業界が策定
 - MISRA(Motor Industry Software Reliability Association)
- 127個のルールが定められている
- 車載用ソフトウェアをターゲット
 - 品質と信頼性の向上が目的
 - 一般の組込みソフトウェア開発にも有用

MISRA-Cはもともとは車載用ソフトウェアを対象としたプログラミングガイドラインです。MISRA-Cは、自動車業界のみならず広く一般の組込みソフトウェア技術者にとっても有用であるとされています。

IPA／SEC:

組込みソフトウェア開発向けコーディング作法ガイド

- IPA／SEC(情報処理推進機構 ソフトウェア・エンジニアリング・センター)は組込み分野での開発後半のソースコードレベルでの確実な品質作りこみの実現のために、作法ガイドを作成
 - 組込みソフトウェア向けコーディング作法ガイド(V1. 0)
- C言語のコーディング規約を策定している方を対象とした「作法ガイド」で、目的にあったコーディング規約を策定することを目標にしたガイドです

このガイドでは、ソフトウェアの品質と同様に、コードの品質も「信頼性」「保守性」「移植性」などの品質特性で分類できると考え、コーディングの作法とルールを『JIS X 0129-1 ソフトウェア製品の品質』を基に体系化しています。このガイドにおける作法とは、ソースコードの品質を保つための慣習や実装の考え方で、個々のルールの基本的な概念を示します。

マイコンボードの確認

- [1. マイコンボード\(HSB16C29S64\)](#)
2. プロセッサ(M16C)
3. 開発環境の説明

マイコンボードの概要

マイコンボードと搭載プロセッサ及び 拡張ボードについて解説する

- マイコンボード : HSB16C29S64
 - 北斗電子製
 - USB-Serial変換チップ
- プロセッサ
 - ルネサス製 M16C/29
- 拡張ボード
 - 拡張I/Oボード (STUDY I/O)
 - 倒立2輪車 (PUPPY)

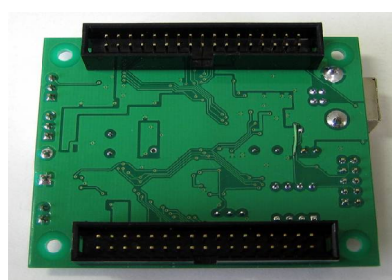
今回、使用するマイコンボードは、北斗電子製HSB16C29S64です。

USB－Serial変換チップを持っていて、シリアルポートを持っていないようなノートパソコンでも使用することができます。このマイコンボードにのっているプロセッサが、ルネサス製のM16C/29です。

拡張ボードは、今回は北斗電子製のSTUDY/IOを使用します。その他には、倒立2輪車PUPPYを取り付けて使用することもできます。

マイコンボード : HSB16C29S64

- USB経由で電源を供給可能
- USB-Serial変換チップを搭載
 - シリアルポートがないPCでも演習が可能
- UARTを2ポート
 - ユーザープログラムでUARTを使用可能(デバッガで1ポートを使用)
- 拡張コネクタ
 - 既存のボードと互換



2008/11/27

TOPPERSプロジェクト認定

127

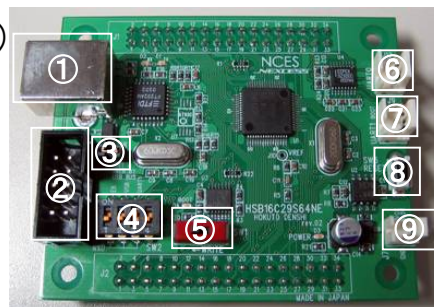


マイコンボードの解説を行います。今回使用するマイコンボードは、USB経由で給電が可能です。UARTを2ポート持っています。このうち1ポートは、RTOS付きのプログラムで使用します。

UARTとは、パソコンの標準的なシリアルポートと通信が可能なインターフェイスです。マイコンボードには、拡張ポートと接続するためのコネクタがあります。

マイコンボード : HSB16C29S64の詳細

1. USBコネクタ(USB_B)
2. E8接続コネクタ(E8)
3. USB給電切り替えジャンパ(J8)
4. UART1切り替えスイッチ(SW2)
5. BOOTスイッチ(SW1)
6. UART0コネクタ(J12)
7. UART1コネクタ(J9)
8. リセットスイッチ(SW5)
9. 電源コネクタ(J7)



2008/11/27

TOPPERSプロジェクト認定

128



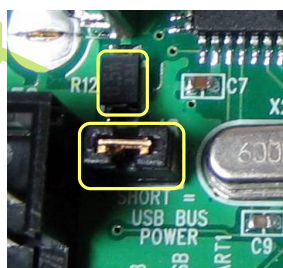
マイコンボードの実装について説明を行います。

- ①USBコネクタです。
- ②E8接続コネクタです。デバッグ用のコネクタです。
- ③USB給電切り替えジャンパです。クローズでUSBからの給電、オープンで拡張ボードまたは⑨からの給電となります。
- ④UART1切り替えスイッチです。E8、USB、⑦のUART1から選択できます。
- ⑤BOOTスイッチです。通常ブートかFLASHROMの書き込みを選択できます。
- ⑥UART0コネクタです。
- ⑦UART1コネクタです。
- ⑧リセットスイッチです。
- ⑨電源コネクタです。

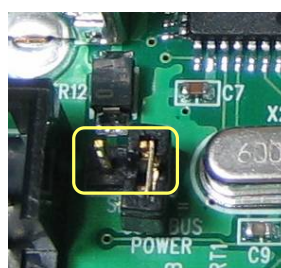
マイコンボード：USBによる給電

- J8 をショートさせるとUSB経由で電源を供給する
- J7 から直接 5V を入力する場合や、拡張ボードから電源を供給する場合は、オープンに(外して)すること
- J8 をショートさせた状態で、USBケーブルでPCと接続してもPOWER LED (D3)が点灯しない場合は、R12のヒューズが切れている可能性が高いので取り替えること

ヒューズ



ショート



オープン

2008/11/27

TOPPERSプロジェクト認定

129

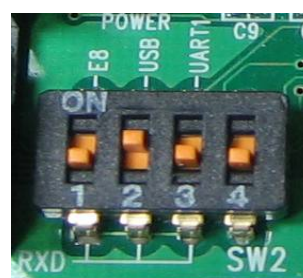


J8給電先を設定します。拡張ボードから電源を供給するような場合、オープンにします。J7(電源コネクタ)から供給する場合もオープンです。USBで給電するばあいはショートにしてください。拡張ボードから電源を供給したり、電源コネクタから給電する場合にJ8をショートにしておくと、ヒューズが飛んでしまいます。

マイコンボード：シリアルポート

- プロセッサのUART1(RXD1,TXD1)は, E8の接続コネクタ(J4), USB-シリアル変換チップ(U6), UART1コネクタ(J9)のいずれかに接続することが可能
- 接続はSW2で切り替える
 - スイッチを上にとするとON
 - 同時に複数のスイッチをONにしないこと

	SW2-1	SW2-2	SW2-3
E8	ON	OFF	OFF
USB	OFF	ON	OFF
UART1	OFF	OFF	ON



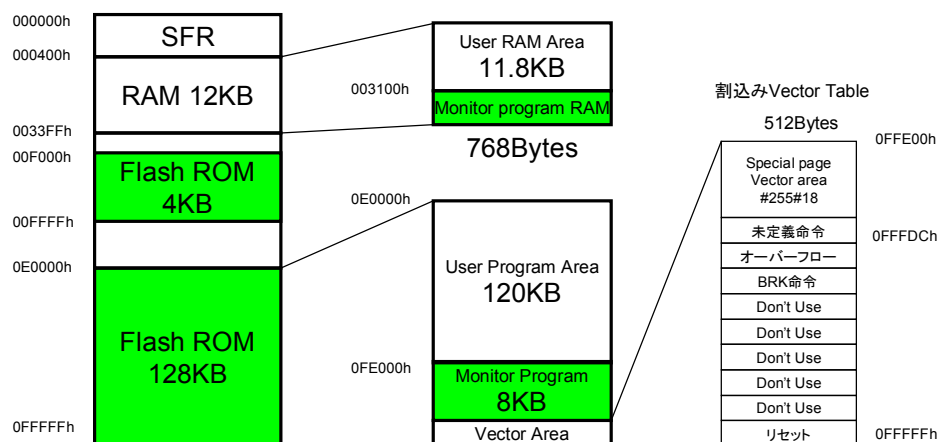
マイコンボードの確認

1. マイコンボード (HSB16C29S64)
- [2. プロセッサ \(M16C\)](#)
3. 開発環境の説明

プロセッサ (M16C) : 概要

- ルネサス製16ビットシングルマイクロコンピュータ
- M16C/29(MC30291FC)
 - ROM : 128KB + 4KB RAM : 12KB
- 周辺回路 : メモリマップドI/O方式
 - 入出力ポート : 55本
 - タイマ : タイマA:16ビット×5チャンネル
タイマB:16ビット×3チャンネル etc...
 - シリアルI/O : 5チャンネル
 - A/D : 16チャンネル
 - CAN : 1チャンネル
 - DMAC : 2チャンネル

プロセッサ (M16C) : メモリマップ



- SFR : 周辺回路の制御レジスタ
- モニタプログラムが使用するメモリ領域はユーザプログラムで使用しないこと

2008/11/27

TOPPERSプロジェクト認定

133



M16Cのメモリマップについて説明を行います。

000000h～0003FFh

SFR:プロセッサ内蔵の周辺回路の制御レジスタが配置

000400h～0033FFh

RAM領域です。デバッガを使用する場合、
003100h以上の領域は使用できません

0E0000h～0FFFFFFh

FLASH ROM領域

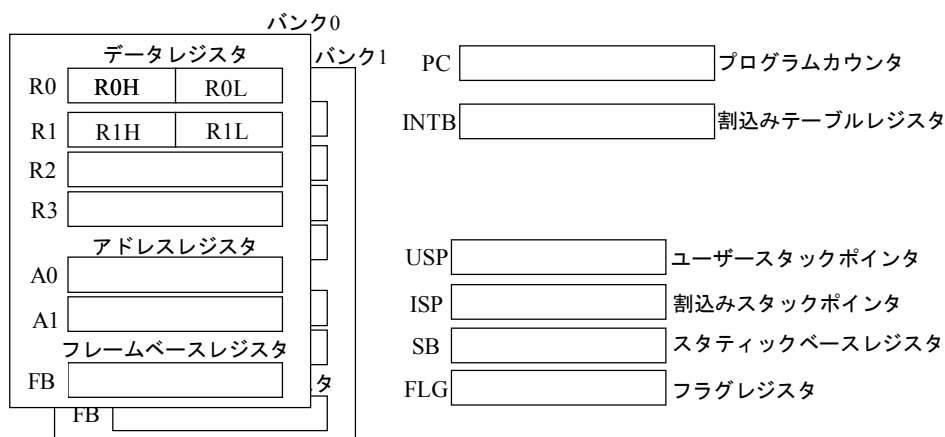
0FE000hから0FFDFFhまでの領域はデバッガ用のモニタ
プログラムが書き込まれています。デバッガを使用する場合
使用できません。

0FFE00h～0FFFFFFh

割り込みベクタ領域

プロセッサ(M16C) : レジスタセット

- 汎用レジスタは2バンク構成(フラグレジスタで切り替え可能)
- データレジスタは16ビットが基本だが, R0とR1は上位と下位を分けて, それぞれ8ビットのレジスタとして使用可能(R0L,R0H,R1L,R1H)
- R2とR0,R3とR1,A1とA0を組み合わせて32ビットレジスタとして使用可能



2008/11/27

TOPPERSプロジェクト認定

134



M16Cのレジスタセットについて説明を行います。汎用レジスタは2バンク構成となっており、割込みの高速化等に使用できます。

R0～R3:データレジスタ、データの演算等に使用します。R0とR1は8ビットレジスタとして使用できます。R0:R1、R2:R3は組み合わせて32ビットレジスタとして使用できます。

A0～A1:アドレスレジスタ、ポインタとしてデータを間接指定する場合に使用します。A0:A1は組み合わせて32ビットレジスタとして使用できます。

FB:フレームベースレジスタ、16ビットオフセット間接アドレス指定専用のレジスタ

PC:プログラムカウンタ、実行プログラムアドレスを指定するレジスタ

INTB:割込みテーブルレジスタ、割込みテーブルの先頭を指定するレジスタ

USP:ユーザースタックポインタ、ユーザープログラム実行時のスタックポインタ

ISP:割込みスタックレジスタ、割込み時のスタックポインタ

SB:スタティックベースレジスタ、16ビットオフセット間接アドレス指定専用のレジスタ

FLG:フラグレジスタ、プロセッサの状態を示すレジスタ

プロセッサ (M16C) : リセットと割込みベクタ

- M16Cではリセットは割込みの一種
 - プロセッサによって定義は異なるので注意
- 固定ベクタテーブル
 - 0xFFFFDC～0xFFFFFにノンマスカブル割込み(ソフトウェア例外, NMI, シングルステップ)の割込みベクタが並んでいる
 - リセットのベクタは 0xFFFFC～0xFFFFFに配置
 - プログラムの先頭のアドレスを書いておく
- 可変ベクタテーブル
 - マスカブル割込みの割込みベクタが並んでいる
 - 周辺回路の割込みベクタ
 - 任意の番地に配置が可能

2008/11/27

TOPPERSプロジェクト認定

135



M16Cでは、リセットは割込みの一種として扱います。

固定ベクタテーブルには、ノンマスカブル割込みの割込みベクタが並んでいます。このテーブルは、0xFFFFDC～0xFFFFF番地に置かれることが決まっています。それで固定ベクタテーブルと呼ばれています。それぞれのノンマスカブル割込みの要因ごとにどの番地にベクタを配置するか以下のように決まっています。

未定義命令	FFFDC ～ FFFDF
オーバフロー	FFFE0 ～
BRK命令	FFFE4 ～ FFFE7
アドレス一致	FFFE8 ～ FFEFB
シングルステップ	FFFE0 ～ FFEF
ウォッチドッグタイマ、発振停止、再発振検出、電圧低下検出	FFFF0 ～ FFFF3
DBC	FFFF4 ～ FFFF7
NMI	FFFF8 ～ FFFFB
リセット	FFFFC ～ FFFFF

例えば、リセットベクタは0xFFFFC～に配置されているので、ここにプログラムの先頭アドレス(たとえばスタートアップルーチン)を書いておけば、電源が入った時にリセット割込みが発生して、そこで指定された関数が実行されます。

可変ベクタテーブルには、マスカブル割込みの割込みベクタが並んでいます。要因ごとに対応するベクタを並べてテーブルを作成し、このテーブルは任意の番地に配置が可能です。そのため、可変ベクタテーブルと呼ばれます。どの番地に配置するかは、スタートアップルーチンなどで、INTBレジスタに設定します。

プロセッサ(M16C)：割込み/例外処理の流れ

- 要求された割込みの優先レベルが、プロセッサのフラグレジスタ(FLG)中の割込み優先マスク(IPL)より大きければ、割込みを受け付ける
- 割込みを受け付けると、割込み要因に対応したベクタテーブルに登録されているアドレスにジャンプする
その際、以下の処理をハードウェアが行う
 - スタックを割込みスタック(IPS)に変更
 - 割込み要因に対応した割込み制御レジスタのIRビットをクリア
 - FLGの割込み許可フラグを‘0’にして割込み禁止
 - FLGとPCをスタックに保存
 - プロセッサの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
- 割込みからの復帰
 - REIT命令：スタックからFLGとPCを復帰

2008/11/27

TOPPERSプロジェクト認定

136



M16Cでは、現在のプロセッサの割込み優先度マスクは、フラグレジスタ(FLG)中の割込み優先度マスク(IPL)を見ればわかります。要求された割込みの優先レベルが、プロセッサの割込み優先度マスクより大きければ割込みを受け付けます。

割込みを受け付けると、要因に対応したベクタテーブルに登録されているアドレスにジャンプします。その際に以下の処理をハードウェアが行ってくれます。

- 1) M16Cは通常使用しているスタックと割込み処理時に使用するスタックが異なります。よってまず、スタックを割込み用のスタック(IPS)に変更します。
- 2) 割込み要因に対応した割り込み制御レジスタのIRビットをクリアします。(要求があると、1が立ちます)
- 3) フラグレジスタの割り込み許可フラグを0にして、割込み禁止にします。
- 4) フラグレジスタとプログラムカウンタをスタックに保存します。
- 5) プロセッサ割り込み優先度レベルに、受け付けた割込み優先度レベルを設定します。

マイコンボードの確認

1. マイコンボード (HSB16C29S64)
2. プロセッサ (M16C)
3. [開発環境の説明](#)

拡張ボード：拡張I/Oボード(BB STUDY I/O)

- プログラマブル入出力ポートを用いた基本的なプログラミング実習のための拡張ボード

- ハードウェア

1. LED : 4
2. ディップスイッチ : 4
3. プッシュスイッチ : 2
 - ・ 割込み入力と兼用
4. ブザー : 1
5. CAN コネクタ : 2
6. CAN終端抵抗切り替え : 1



2008/11/27

TOPPERSプロジェクト認定

138



拡張ボードは、プロセッサの入出力ポートを用いていろいろなハードウェアを検証するためのボードです。
 拡張ボード:BB STUDY I/O上のハードウェアを検証しましょう。

- ①LEDです。4つ付いています。
- ②デップスイッチです。4つ付いています。
- ③プッシュスイッチです。SW9とSW8の2つがあります。
- ④ブザー
- ⑤CANコネクタです。2チャンネルあります。
- ⑥CAN終端抵抗の切り替えスイッチです。

拡張ボード：倒立2輪車(PUPPY)

- 倒立制御学習用キット
 - ロータリーエンコーダー, ジャイロの入力によりモーター(マブチ)を駆動して倒立する
- ハードウェア
 - モータ
 - ジャイロ(角速度)
 - ロータリーエンコーダ
 - LED
 - ディップスイッチ
 - 電源(単3乾電池)
 - 上2個：プロセッサ用
 - 下2個：モータ用



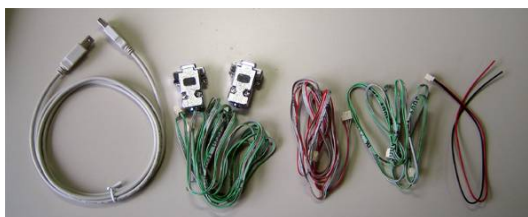
倒立二輪車:PUPPY、ジャイロとモータつき2輪を用いて、倒立歩行が可能な拡張ボードです。

開発環境の確認

1. [ハードウェア・ソフトウェア環境の構築](#)
2. クロス環境
3. サンプルプログラムのビルドと実行
4. フラッシュメモリへの書き込み

ハードウェア環境の構築 : マイコンボードの梱包物

- CPUボード : HSB16C29S64
- 拡張ボード : BB STUDY I/O
- USBケーブル x 1
- シリアルケーブル x 2
- 4線CANケーブル(5V供給可能) x 1
- 3線CANケーブル x 1
- 電源ケーブル(片側圧着)
- CD-ROM



2008/11/27

TOPPERSプロジェクト認定

141



まず、実習ハードウェアの確認を行います。ボード箱の中には以下の部品があります。内容を確認ください。

- 1) CPUボード: HSB16C29S64
- 2) 拡張ボード: BB STUDY I/O CPUボードと接続しています。
- 3) USBケーブル: 上記の写真の左端
- 4) シリアルケーブル: 2本あります。上記の写真の左から2番目
- 5) 4線CANケーブル(5v供給可能)x1。上記の写真の真ん中
- 6) 3線CANケーブルx1。上記の写真の右から2番目
- 7) 電源ケーブル(片側圧着)。写真の右端
- 8) CD-ROM

ハードウェア環境の構築 : マイコンボードのセットアップ

- UART1をPCと接続(3種類の接続形態)
 - シリアルケーブルによりPCのCOMポートと接続
 - SW2-3をON
 - USBケーブルにより接続
 - SW2-2をON
 - E8のコネクタ経由で接続(E8 or USB writer)
 - SW2-1をON
- UART0をPCと接続
 - PCにCOMポートがある場合には, UART0をシリアルケーブルで接続する
- 電源の接続
 - USB経由で給電しない場合は, 電源ケーブルを用いて J7 から 5V を供給すること

2008/11/27

TOPPERSプロジェクト認定

142



マイコンボードのセットアップについて説明します。パソコンとの通信には以下の2つの接続ができます。本実習ではUART1をUSBケーブルで接続し、電源の供給も行います。

1) UART1

- シリアルケーブルをPCのCOMポートに接続: SW2-3をON
- USBケーブルをPCのUSBコネクタに接続: SW2-2をON
- E8のコネクタを用いてパソコンと接続: SW2-1をON

2) UART0

- PCのCOMポートにシリアルケーブルで接続
- USB経由で給電しない場合は、電源ケーブルを用いてJ7から5vを供給する。

ソフトウェア環境の構築

- M16C開発環境(付属CD-ROMからインストール)
 - NC30WA : コンパイラ, アセンブラ, リンカ
 - TM : 統合開発環境
 - KD30 : リモートデバッガ
 - FlashSta : フラッシュメモリライタ

TMのインストールディレクトリにコピー
- その他のソフトウェア
 - エディタ(TeraPad, サクラエディタ, Xyzzy等)
 - ターミナルソフトウェア(TeraTermPro等)
 - PDFリーダー(Acrobat Reader等)

2008/11/27

TOPPERSプロジェクト認定

143



ソフトウェアの環境構築について説明します。M16C専用の開発環境は付属のCD-ROMからインストールができます。開発環境は以下の4つがあります。

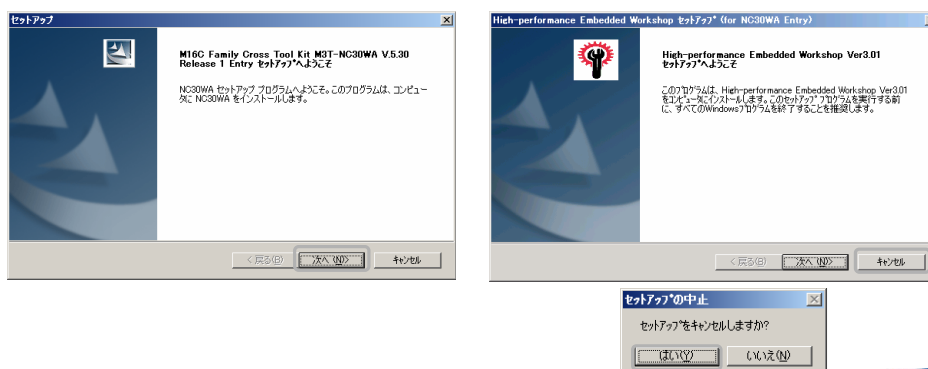
- 1) NC30WA M16C用のコンパイラ、アセンブラ、リンカ
- 2) TM 統合開発環境
- 3) KD30 M16C用リモートデバッガ
- 4) FlashSta フラッシュメモリ書き込みプログラム

M16C専用の開発環境以外に、以下のソフトウェアのインストールが必要です。

- 1) エディタ(TeraPad、サクラエディタ、Xyzzy等)
- 2) ターミナルソフトウェア(TeraTermPro等) 1, 2日のセミナーでは使いません。
- 3) PDFリーダー(Acrobat Reader等)

NC30WAのインストール

- インストーラを起動
 - “CD-ROM”¥NC30WA¥nc30wa¥w95j¥setup.exe
- HEWのインストールのキャンセル
 - “次へ”を押すとHEWのインストーラーが起動するが、今回はインストールしないため、“キャンセル”を押す



2008/11/27

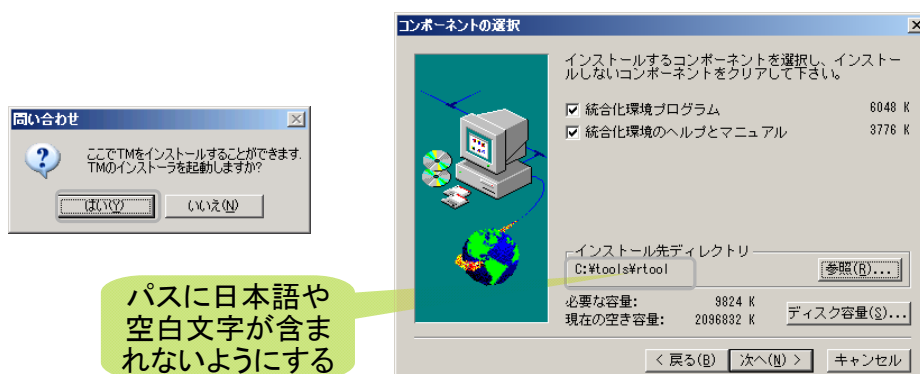
TOPPERSプロジェクト認定

144



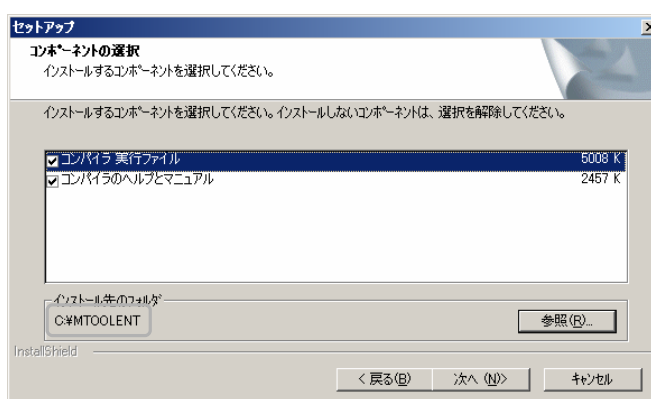
NC30WAのインストール

- HEWのインストールのキャンセル後、TMをインストールするか尋ねられるので“はい”を選択する
- 数ステップ後にインストール先のディレクトリの指定画面になるので、インストールしたいディレクトリを指定する



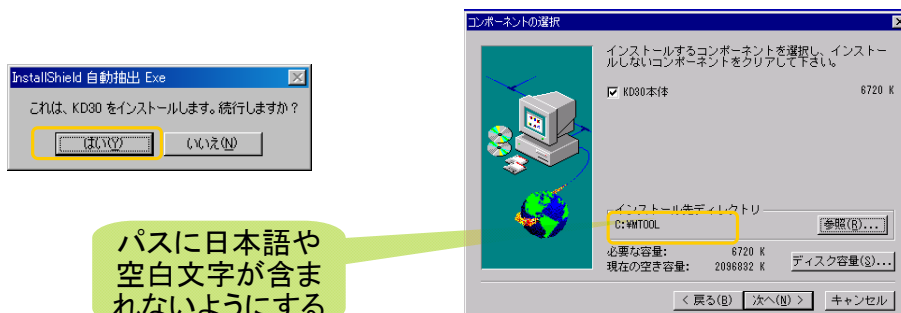
NC30WAのインストール

- TMのインストール終了後, NC30WAのインストール先のディレクトリを指定する
 - 利便性のためTMと合わせる(必須ではない)
 - パスには日本語, 空白が含まれないようにする



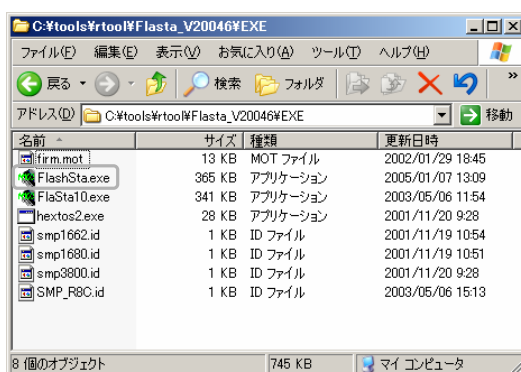
KD30(デバッガ)のインストール

- インストーラを起動
 - “CD-ROM”¥KD30_4.10¥KD30V410R1_J_20041203.EXE
- 自動抽出 EXE
 - “はい”を押してインストーラを展開してインストールを行う
- フォルダの指定
 - パスには日本語, 空白が含まれないようにする



FlashStaのインストール

- インストーラーがないので、CD-ROMから以下のフォルダごと、ハードディスクへコピーする(NC30WAと同じフォルダを推奨)
 - “CD-ROM”¥Flasta_V20046¥EXE
- FlashSta.exe を使用するので、ショートカットをデスクトップや、スタートメニューに登録する



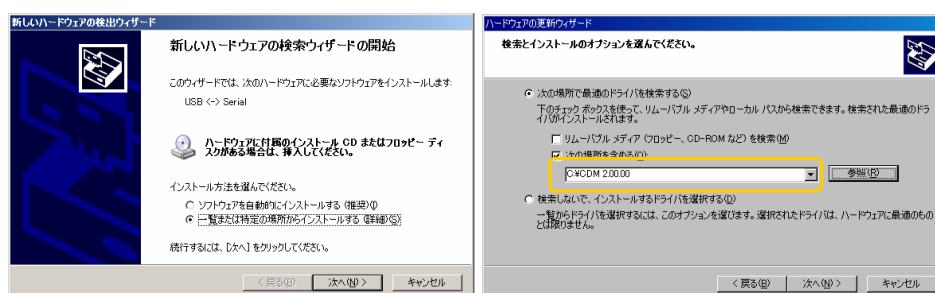
USB-シリアル変換ドライバのインストール

- ドライバのコピー
 - ドライバファイルをCD-ROMからハードディスクにコピーする
 - WindowsXP/2000の場合
 - “CD-ROM”¥Driver¥XP_2003_2k¥CDM 2.00.00
 - 本資料ではC:ドライブの直下にコピーしたとして説明する



USB-シリアル変換ドライバのインストール

- ホストPCとの接続
 - ボードとホストPCをUSBケーブルで接続する
 - ドライバインストール画面が表示されるので、“一覧または特定の場所からインストールする”を選択
 - 次の画面で“次の場所を含める”で、先ほど展開したドライバのフォルダを指定する



2008/11/27

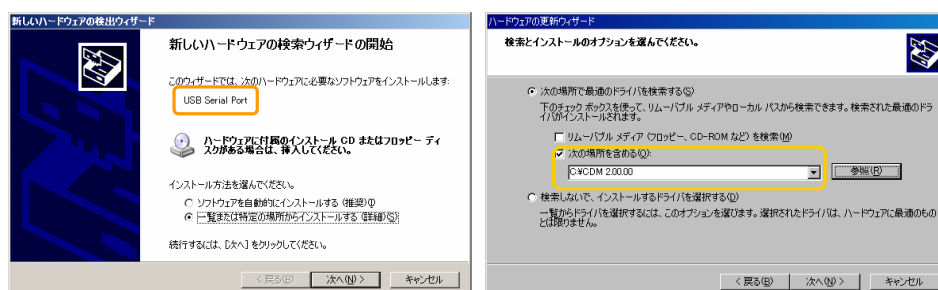
TOPPERSプロジェクト認定

150



USB-シリアル変換ドライバのインストール

- ・ インストールが終了すると、もう一度ドライバのインストール画面が表示されるので、同様に“一覧または特定の場所からインストールする”を選択する
- ・ 次の画面で“次の場所を含める”で、先ほど展開したドライバのフォルダを指定する



2008/11/27

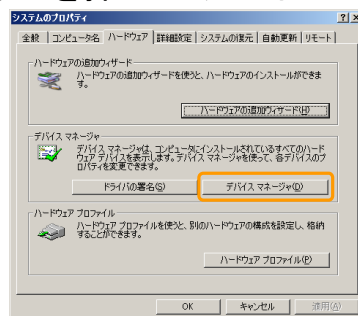
TOPPERSプロジェクト認定

151



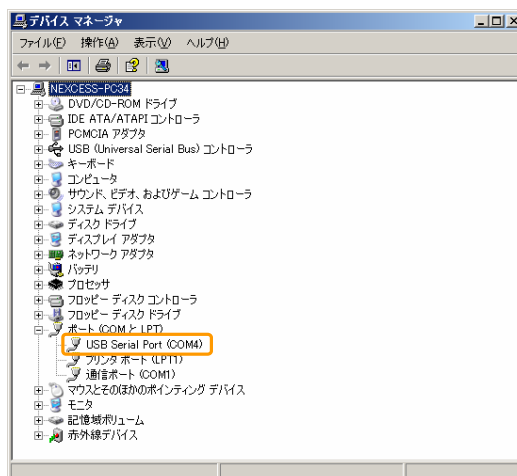
USB-シリアル変換ドライバのインストール

- COMポートの確認
 - USB-シリアル変換に割り当てられたCOMポートを確認する
 - “マイコンピュータ”で右クリックして“プロパティ”を選択して“システムのプロパティ”を開く
 - “ハードウェア”のタブを選択し, “デバイスマネージャ”のボタンを押して”デバイスマネージャ”を起動する”



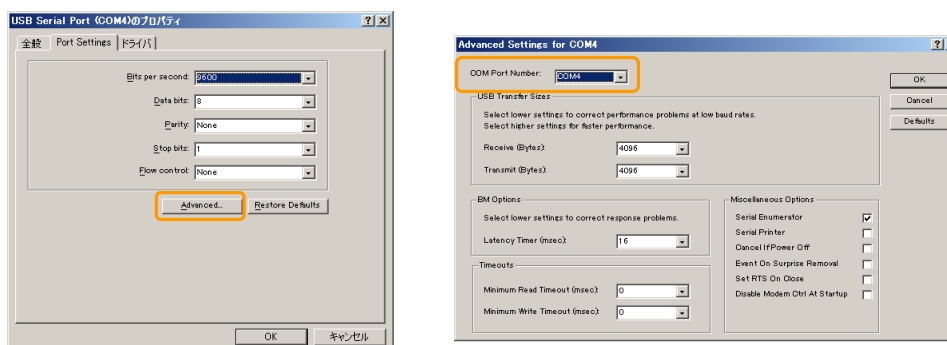
USB-シリアル変換ドライバのインストール

- “ポート(COMとLPT)”のタブを開く
 - USB Serial Port の項目に割り当てられたCOMポートが表示される



USB-シリアル変換ドライバのインストール

- 割当てCOMポートの変更
 - USB Serial Port の項目で右クリックし, ”プロパティ”を選択する
 - Port Settings のタブを選択し, ”Advanced”をクリックする
 - “COM Port Number:”で, 割り当てるCOMポートを選択する



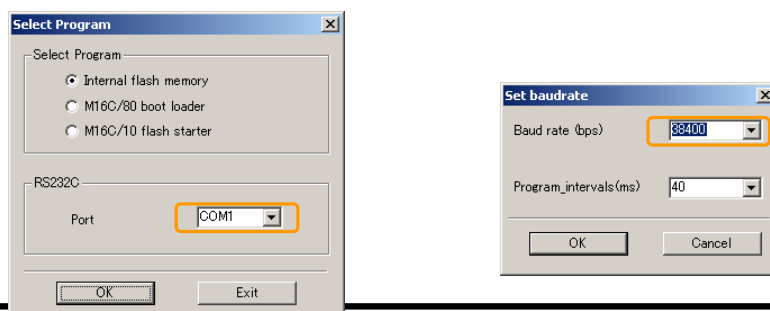
モニタプログラムの書込み

- KD30を用いてデバッグを行うためには、ボードにモニタプログラムを書き込んでおく必要がある
- モニタプログラムの書込みは、FlashStaにより行う
- ボード設定
 - ボードのBOOTスイッチをON(左側)にしてリセット(SW5)を押す
 - ボードとPCをUSBケーブルで接続する



モニタプログラムの書込み : FlashStaによる書込み

- FlashStaを起動
- “Select Program”ウィンドが表示されたらUSB-シリアル変換のポート番号を指定して, "OK"を押す
- “Setbaudrate”ウィンドが表示された場合は, Baud rate を 115200に設定
 - 後エラーとなった場合は 57600 → 38400 → 19200 → 9600 と通信速度を落としてリトライする



2008/11/27

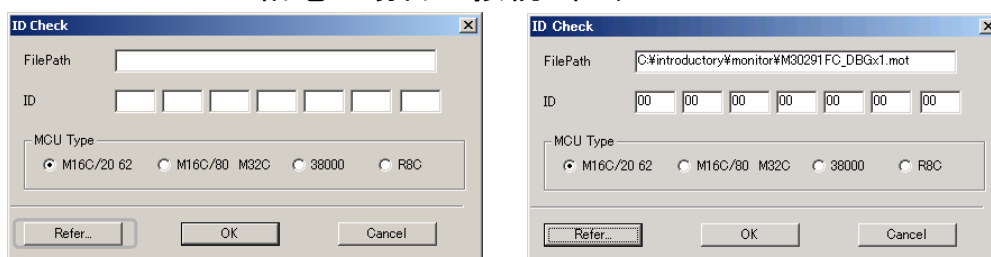
TOPPERSプロジェクト認定

156



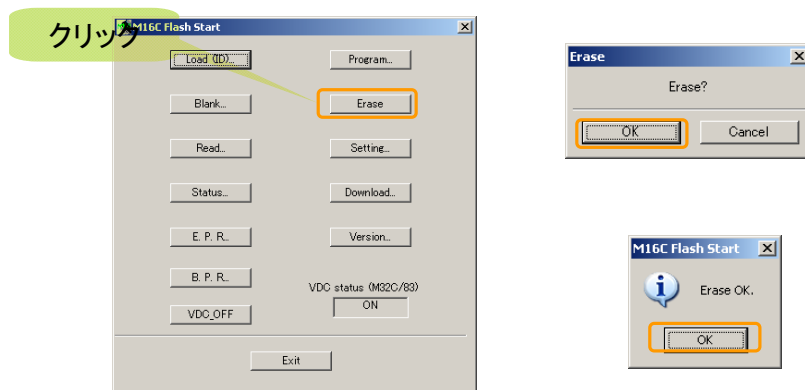
モニタプログラムの書込み : ファイル選択

- “ID Check”ウィンドウが表示されたら“Refer...”ボタンを押してCD-ROM内のモニタプログラム (M30291FC_DBGx1.mot) を指定する
- ID番号は書き込まれているプログラムのIDを指定する
 - ボード出荷時に書き込まれているデモプログラムのIDは, ”ff ff ff ff ff ff ff”
- エラーとなる場合は電源を入れなおすこと
 - USBの給電の場合は接続し直す



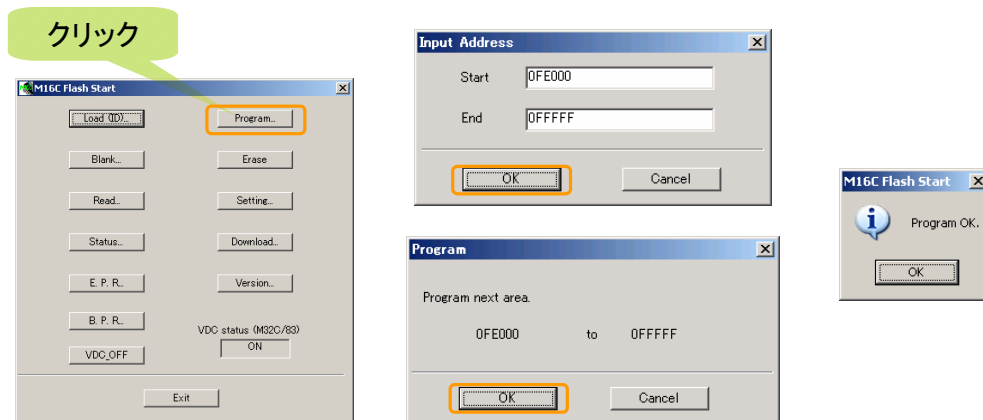
モニタプログラムの書込み：消去

- メインウィンドウが表示されたら、ファイルを書き込む前に現在のフラッシュの内容を消去する
- メニューからEraseを選択すると“Erase”ウィンドウが表示されるので“OK”を押し消去を開始する
- 消去が終わると“Erase OK.”のウィンドウが表示される



モニタプログラムの書込み : 書き込み

- “Program” を選択すると “Input Address” ウィンドウが表示されるので, “OK” を押す
- “Program” ウィンドウで “OK” を押すと書き込みが始まり終了すると通知ウィンドウが表示される



モニタプログラムの書込み : 実行

- 書き込み終了後BOOTスイッチをOFFにする(右側)
- リセットボタン(SW5)を押すと書き込んだモニタプログラムが実行される

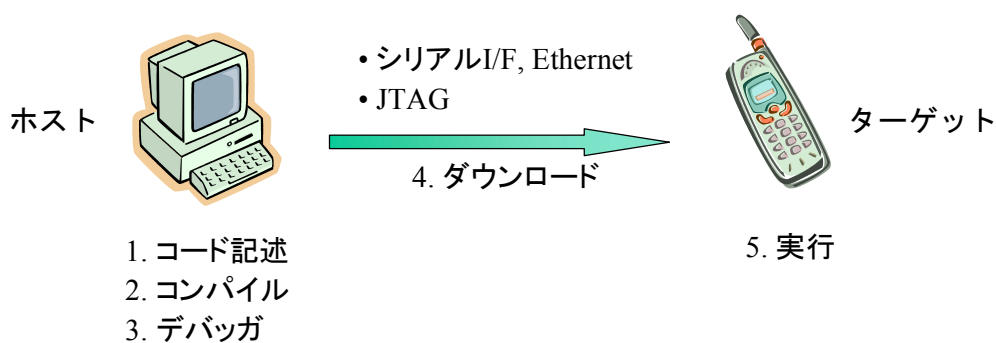


開発環境の確認

1. ハードウェア・ソフトウェア環境の構築
2. [クロス環境](#)
3. サンプルプログラムのビルドと実行
4. フラッシュメモリへの書き込み

クロス開発 : ホストとターゲット

- 組込みシステム開発特有の開発形式
 セルフ開発
- ホスト(開発環境)とターゲット(実行環境)が異なる



2008/11/27

TOPPERSプロジェクト認定

162



パソコン等のソフトウェア開発はセルフ開発と呼ばれ、パソコン上で開発したプログラムはパソコン上で実行が可能です。組込みシステム開発ではプログラムの開発はパソコン上で行いますが、プログラムの実行はターゲット上で行います。これをクロス開発といい、組込みシステム開発特有の開発形態です。

クロス開発では開発側のホスト、実行側をターゲットと呼びます。ホストとターゲットはシリアルI/Fまたは、Ethernetまたは、JTAG等のインターフェイスで接続します。まず、ホスト上で、コードの記述、コンパイル、ビルドを行います。接続インターフェイスを通して、プログラムをターゲットにダウンロードし、ターゲット上でプログラムの実行やデバッグを行います。

クロス開発 : クロスコンパイラとリモートデバッグ

- クロスコンパイラ
 - ホスト(開発環境)のコンピュータと異なる機械語を生成するコンパイラ
 ◀ セルフコンパイラ
 - ホストでは直接実行不可能
 - ISS (Instruction Set Simulator) によりシミュレーションする
 - GCCは多くのプロセッサに対応
 - ARM, MIPS, SH, Sparc, H8, M68K, V850
 - 本講義ではルネサステクノロジのNC30WAを使用
- リモートデバッグ
 - ホストからターゲットをデバッグするデバッグ
 - デバッグモニタを使ったデバッグ
 - ターゲットにデバッグと通信するプログラムが動作
 - ICEを用いた方法
 - 本講義ではデバッグモニタとルネサステクノロジのKDを使用

2008/11/27

TOPPERSプロジェクト認定

163



M16Cの開発で使用するクロスコンパイラとリモートデバッグについて説明します。

クロスコンパイラはホストで、ターゲット用の機械語プログラムを生成するプログラムです。機械語はホスト用のものと異なるため、ホスト上で実行はできませんが、ISS (Instruction Set Simulator) を使用することにより、ホスト上で擬似実行が可能です。クロスコンパイラとしてはGCCが有名で、多くのプロセッサに対応を行っていますが、オープンソースという点もあり、確実に生成データを保障できないという問題もあります。本セミナーではクロスコンパイラとしてNC30WAを使用します。

リモートデバッグとは、ターゲットボード上に書き込んだデバッグモニタと通信しながら、ホスト上でターゲットのデバッグを行うプログラムです。デバッグの実体が開発ホスト上にあることから、ソースレベルの高度なデバッグ機能を実現します。本セミナーでは、ルネサステクノロジ製のデバッグモニタとKD30を使用します。

C言語プログラムのコンパイル

- C言語で記述したソースコードをプログラムとしてプロセッサ上で動作させるためには、プリプロセッサ、コンパイラ、アセンブラ、リンカを用いてオブジェクトコードに変換する必要がある。
- プリプロセッサ : cpp30
 - #includeやマクロを展開する
- コンパイラ : nc30, ccom30
 - オブジェクトコードを生成するまでの一連の処理(プリプロセッサ, コンパイラ, アセンブラ, リンカの呼び出し)を行う場合は, コンパイラドライバとも呼ばれる
- アセンブラ : as30
 - アセンブリ言語記述を機械語プログラムに変換する.
- リンカ : ln30
 - 複数の機械語プログラムとライブラリをリンクし最終イメージを生成する

NC30WAの内容について説明を行います。NC30WAはコンパイラドライバで、種々のコンパイルプログラムを管理実行することにより、コンパイルやリンクの機能を実現します。

- 1) nc30:コンパイラドライバ、以下のプログラムの実行を指定する
- 2) cpp30:プリプロセッサ、インクルード文やマクロの展開を行います。
- 3) ccom30:コンパイラ本体、C言語をアセンブラ言語に変換する。
- 4) as30:アセンブラプログラム、アセンブラ言語を機械語のオブジェクトコードに変換する。
- 5) ln30:リンカ、複数のオブジェクトコードとライブラリを統合して、実行形式のファイルを生成する。

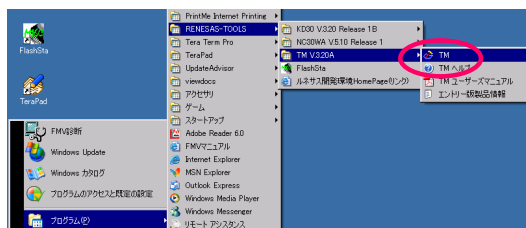
開発環境の確認

1. ハードウェア・ソフトウェア環境の構築
2. クロス環境
3. [サンプルプログラムのビルドと実行](#)
4. フラッシュメモリへの書き込み

ビルド：開発環境の起動

開発環境の動作確認のため、
サンプルプログラムをコンパイルする

- 統合開発環境(TM)の実行
 - スタート→プログラム→RENESAS-TOOLS→TM V... から
TMを実行する

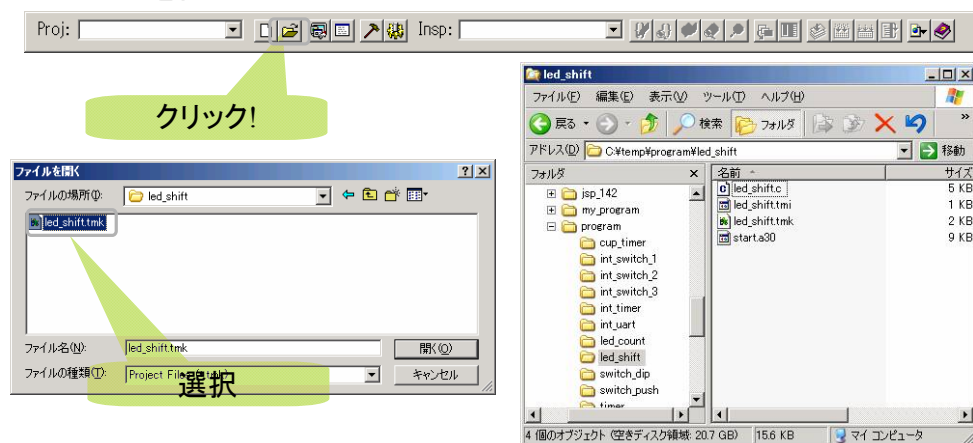


- 起動後、画面上部にバーが表示される



ビルド : サンプルプロジェクトを開く

- バーにある Open Project のボタンを押して表示されるダイアログから
¥教材ディレクトリ¥program¥led_shift¥led_shift.tmk
を開く



2008/11/27

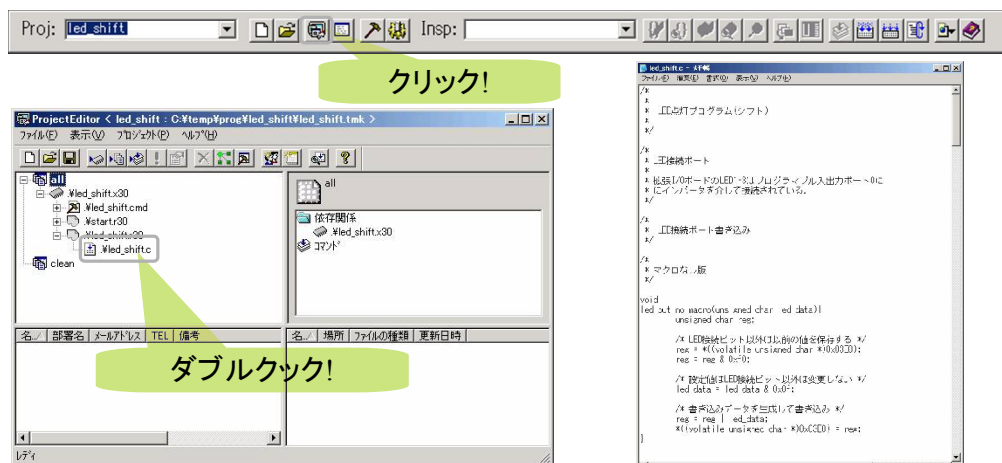
TOPPERSプロジェクト認定

167



ビルド : ファイルの確認

- ツールバーのボタンからProjectEditorが起動する
- 左上のウィンドウからled_shift.cをダブルクリックするとソースコードが表示される
- led_shiftは led_shift.c と start.a30 で構成されている



2008/11/27

TOPPERSプロジェクト認定

168



ビルド : ビルド開始

- バーのビルドボタンを押してビルドする



ここをクリック!

- ビルドウィンドウが表示されてビルドが開始される

```

Builder < led_shift : C:\temp\proj\led_shift\led_shift.link >
ファイル(F) 編集(E) 表示(V) 動作(A) ヘルプ(H)
led_shift.c
LN80 @.Vled_shift.cmd
Linkage Editor (Ln80) for R8C/Tiny/M16C Series Version 5.10.01
COPYRIGHT(C) 1995(2004) RENESAS TECHNOLOGY CORPORATION
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

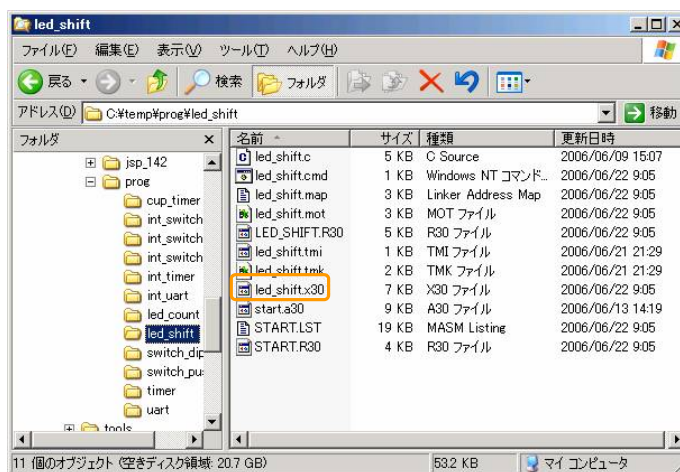
now processing pass 1
processing ".Vstart.r30"
processing ".Vled_shift.r30"
processing "Libraries"
now processing pass 2
processing ".Vstart.r30"
processing ".Vled_shift.r30"

DATA 00000000(000000H) Byte(s)
ROMDATA 00000000(000000H) Byte(s)
CODE 0000847(0034FH) Byte(s)
LMC30 -L .Vled_shift.x30
Load Module Converter (Lmc30) for R8C/Tiny/M16C/60 Series Version 4.00.01
COPYRIGHT(C) 1995(2004) RENESAS TECHNOLOGY CORPORATION
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

***** Finish...
  
```

ビルド : ビルド結果

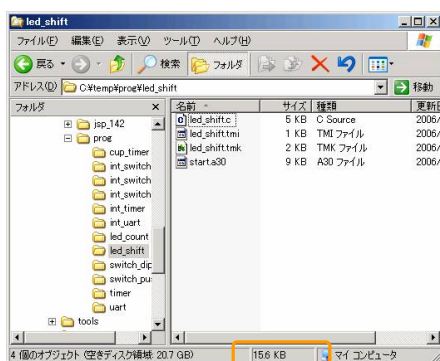
- led_shift.x30という名前のファイルが作成されていればビルドは成功
- 実行する前にボードをPCに接続しておくこと



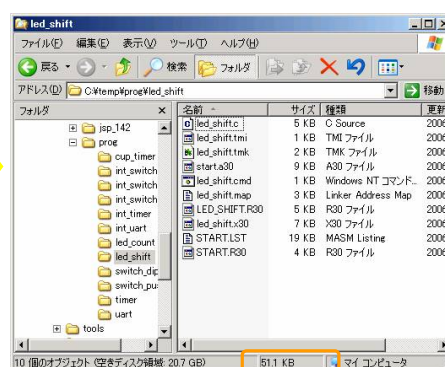
ビルド : ファイルのクリーン

- ビルドを行うと多くの中間ファイルが生成されます
- プログラムを外部メディアに保存する場合は, 不必要な中間ファイルを削除する必要があります

• ビルド前



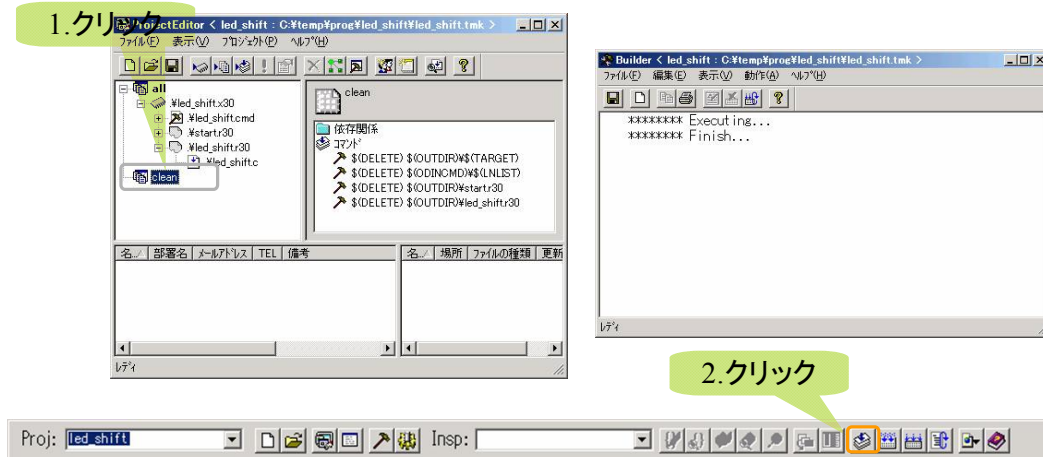
• ビルド後



ビルド : クリーン方法

- ・ クリーンしたいプロジェクトをTMで開き, "ProjectEditor"を起動します.
- ・ “ProjectEditor”のアイテムウィンドウにあるcleanをクリックし, TMのツールバーの部分ビルドをクリックするとクリーンが実行される

1.クリック



実行：デバグの起動

- デバグを起動する前に



- TeraTerm等がデバグ用のCOMポートを使用していないことを確認
- 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?

以上を確認して一度リセットボタンを押す



- デバグの起動

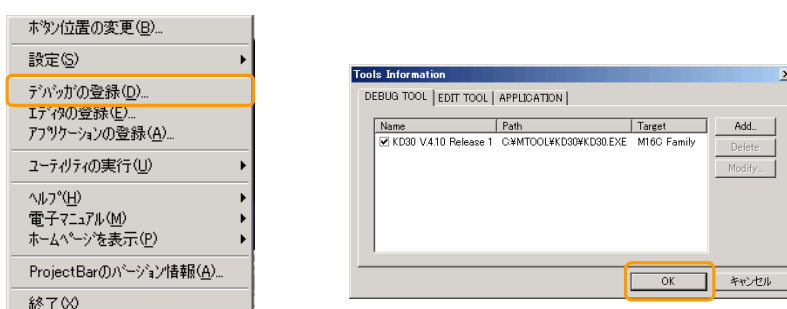
- led_shiftのプロジェクトを開いた状態で, TMのツールバーからデバグを起動する



ここをクリック!

実行 : TMからデバッガが起動しない場合

- KDが異なるパスにインストールされた環境で作成されたプロジェクトを流用した場合は, KDが起動しない場合がある
- この場合は, TMのツールバーのボタンがない箇所で右クリックして, "デバッガの登録"を選択する
- デバッガの選択画面が呼び出され, デバッガのパスが更新されるので, "OK"を押す
- 再びTMのツールバーからデバッガを起動する



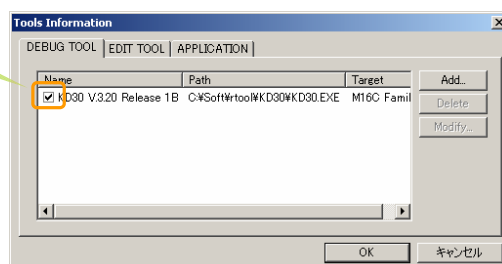
実行：デバッガの選択

- デバッガを起動すると、場合によっては下のようなダイアログが表示され“OK”を押すとデバッガ選択画面になる



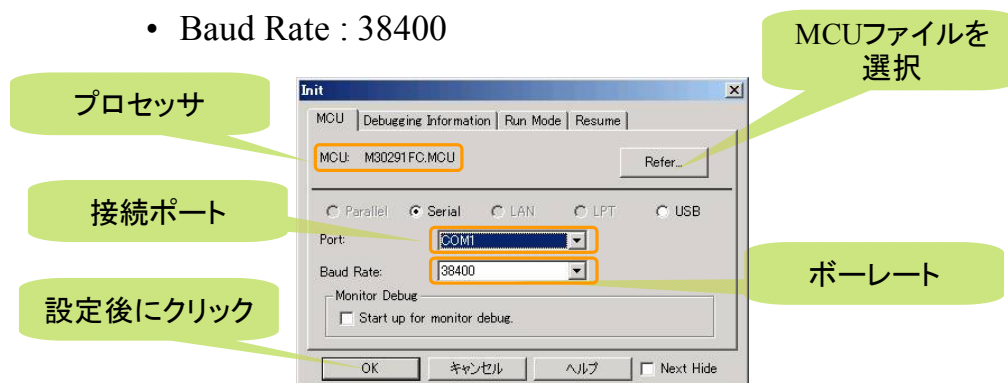
- デバッガ選択画面でKD30にチェックを入れて"OK"

チェック



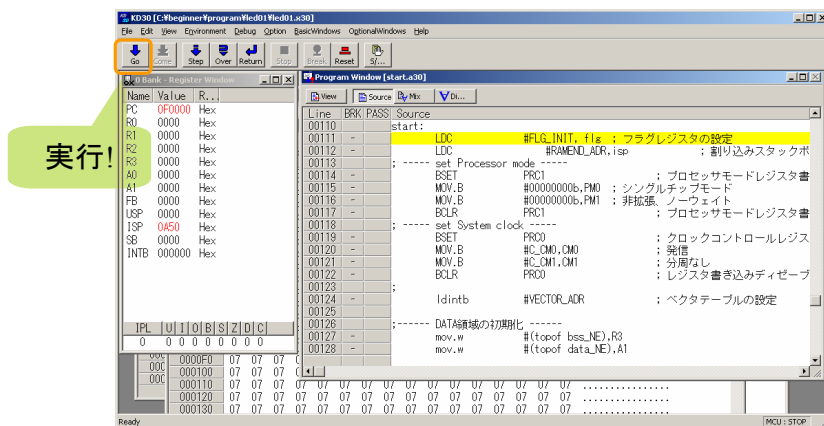
実行：デバッガの設定

- デバッガ初期化画面で以下を設定
 - MCUファイル：Refer でダイアログを開き M30291FC.MCUを選択
 - Port：デバッガ用COMポート番号
 - Baud Rate：38400



実行 : プログラムの実行

- デバッガが起動されると自動的にロードされる
- ロード終了後GOボタンでプログラムの実行を開始
- LEDが LED1:ON→LED2:ON→LED3:ON→LED4:ON と点灯すれば成功



2008/11/27

TOPPERSプロジェクト認定

177

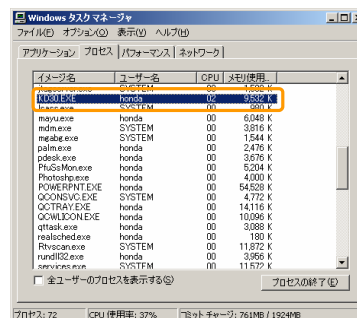


実行 : 起動しない場合

- デバッガが起動しない場合は、以下の事項をチェックし、KDを再実行する
 - マイコンボードの電源
 - BOOTスイッチの設定（右側か）
 - リセットが押されていない
 - デバッガの初期化画面での設定情報
- 上記をチェックしても動作しない場合は、ターゲットにモニタが書き込まれていない可能性がある
 - ➡ 本資料の ”モニタプログラム書込み” に従ってモニタを復帰すること

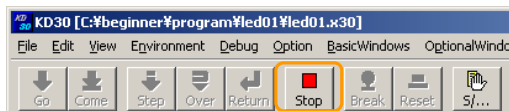
実行 : 起動しない場合

- エラーが出力されない場合は、デバッガ (KD30.EXE) がフリーズしている可能性がある
 - Ctrl+Alt+Delでタスクマネージャーを呼び出し、KD30.EXEを「プロセスの終了」で終了し、再実行する
- TMからKDを起動できない場合は、インストールに失敗している可能性があるので、TM,KD共にアンインストールして再インストールすること

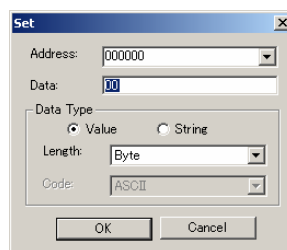
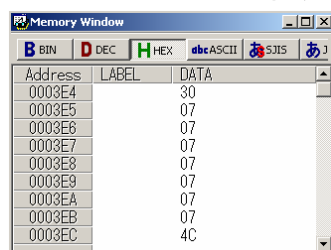


実行 : デバッガの使い方

- プログラムの停止
 - Stopボタンを押す



- メモリ/レジスタの読み書き
 - BasicWindows → Memory Window
 - 右クリック“Set”で特定のアドレスに書き込み可能



開発環境の確認

1. ハードウェア・ソフトウェア環境の構築
2. クロス環境
3. サンプルプログラムのビルドと実行
4. [フラッシュメモリへの書き込み](#)

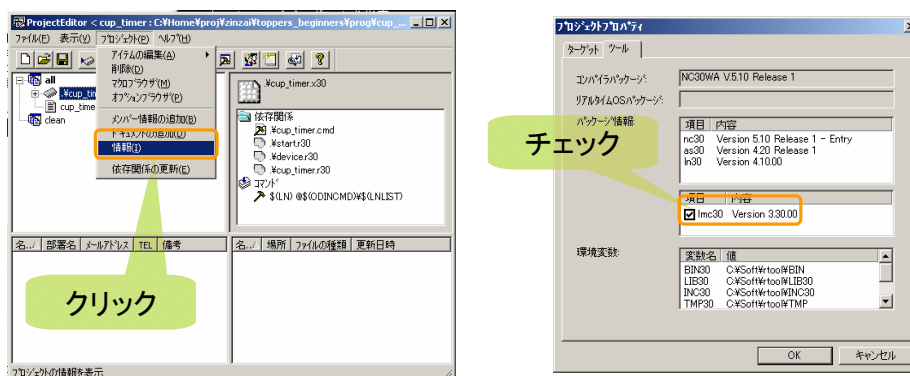
フラッシュメモリへの書き込み

プログラムのフラッシュメモリへの書き込み方法を解説

- マイコンボードには、通常モニタプログラムが書き込まれており、電源を入れるとこのモニタプログラムが動作する
- モニタプログラムはデバッガと通信を行い、ユーザープログラムのダウンロードと実行を実現する
 - ユーザープログラムはフラッシュメモリに書き込まれるが、電源ON時には動作しない
- 電源ONと同時にユーザープログラムを動作させるには、フラッシュメモリ書き込みソフトによりフラッシュメモリに書き込む必要がある

フラッシュメモリへの書き込み : TMの設定

- TMを起動してROM化したいプロジェクトを開き, ProjectEditorで"プロジェクト"→"情報"を選択
- 表示されるプロジェクトプロパティで"ツール"のタブを選択してlmc30の項目にチェックする
- チェックした後, プロジェクトをリビルドするとプロジェクト名.motというファイルが生成される



2008/11/27

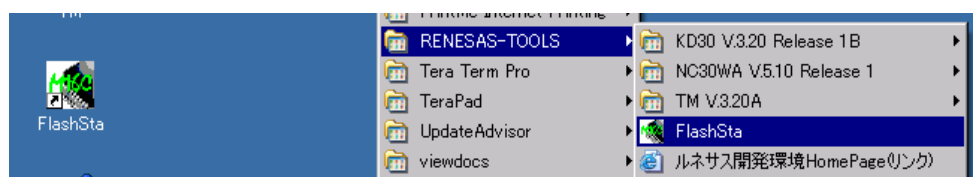
TOPPERSプロジェクト認定

183



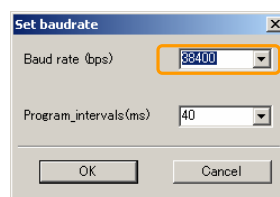
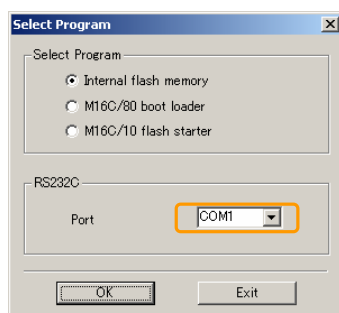
フラッシュメモリへの書き込み：ボードの設定

- ボード設定
 - ボードのBOOTスイッチをON(左側)にする
 - 電源をオン
- 書き込みソフトの起動
 - コピーしたFlashStaを起動



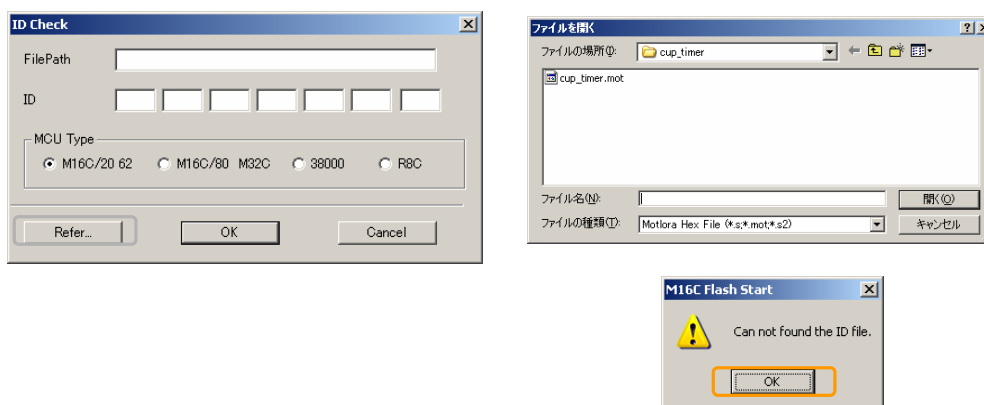
フラッシュメモリへの書き込み：接続設定

- "Select Program"ウィンドが表示されたら"OK"を押す
- "Setbaudrate"ウィンドウでBaud rate を115200に設定する
 - もしこの後エラーとなった場合は57600 → 38400
→19200→9600 と通信速度を落としてリトライする



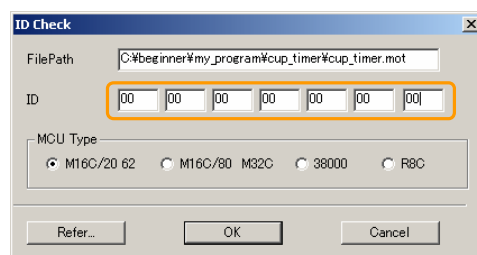
フラッシュメモリへの書き込み：ファイル選択

- “ID Check”ウィンドウが表示されたら“Refer...”ボタンを押して書き込むファイル(拡張子 mot)を選択する
- ファイルを選択して“開く”を押すとエラーが表示されるので“OK”を押す



フラッシュメモリへの書き込み : IDの設定

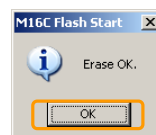
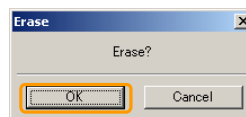
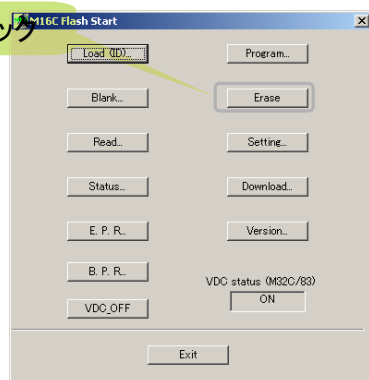
- IDには, 書き込んであるプログラムのIDを指定する



フラッシュメモリへの書き込み：消去

- メインウィンドウが表示されたら、ファイルを書き込む前に現在のフラッシュの内容を消去する
- メニューからEraseを選択すると“Erase”ウィンドウが表示されるので“OK”を押し消去を開始する
- 消去が終わると“Erase OK.”のウィンドウが表示される

クリック



2008/11/27

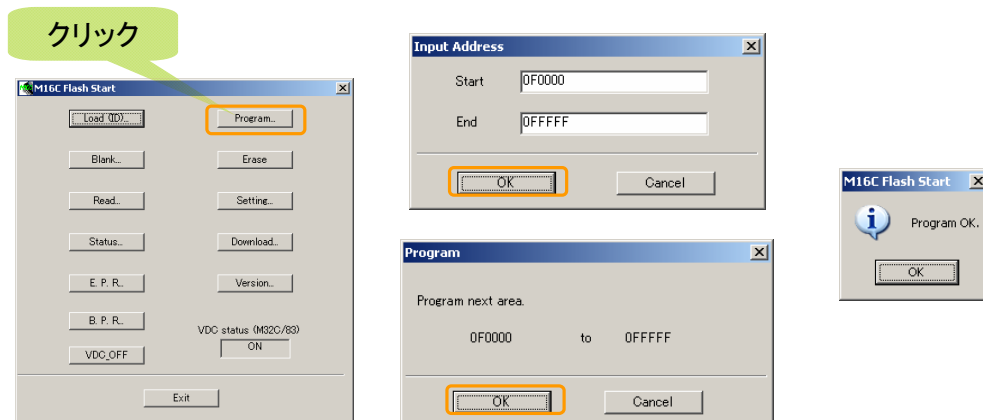
TOPPERSプロジェクト認定

188



フラッシュメモリへの書き込み：書き込み

- “Program” を選択すると “Input Address” ウィンドウが表示されるので, “OK” を押す
- “Program” ウィンドウで “OK” を押すと書き込みが始まり終了すると通知ウィンドウが表示される



フラッシュメモリへの書き込み：実行

- 書き込み終了後、ボードの電源をオフにし、BOOTスイッチをOFFにする(右側)
- 電源を入れると書き込んだプログラムが実行される



- モニタの復帰方法
 - 再びデバッガを使いたい場合は、ボード付属のCD-ROMにあるデバッグモニタを再度フラッシュメモリに書き込む必要がある

まとめ

参考文献

- SESSAME初級セミナーテキスト, プログラミング 組込み用語基礎知識 : 三浦元
- 組込みシステム開発のためのエンベデッド技術 : 社団法人日本システムハウス協会 エンベデッド技術者育成委員会 編・著, 電波新聞社
- 組込み開発者におくる MISRA-C 組込みプログラミングの高信頼性ガイド : MISRA-C研究会 編, 日本規格協会
- An Embedded Software Primer : David E. Simon, ADDISON-WESLEY

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

- はじめに
 - 竹内 良輔((株)リコー)
- 組込みハードウェアの基礎知識
 - 本田 晋也(名古屋大学)
- 組込みプログラム開発の基礎知識
 - 本田 晋也(名古屋大学)
- マイコンボードの確認
 - 本田 晋也(名古屋大学)
- 開発環境の確認
 - 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.