TOPPERS基礎実装セミナー

（STM32F401/446 Nucleo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

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(STM32F401/446 Nucleo64版:基本)1日目>

---

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。

TOPPERS

スケジュール

- １日目

1.	はじめに	0.2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.	ROMモニタを使った実習	0.5時間
7.	まとめ	0.2時間

はじめに

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を用いてシステム的な制御を行う

  （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。

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STM32F401/446 Nuclo64（Cortex-M4）を用いて行う
- Cortex-M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
- Cortex-M4を使用したボードと開発に必要な機器の確認
- 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

組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

多様なプラットホーム（アーキテクチャ）

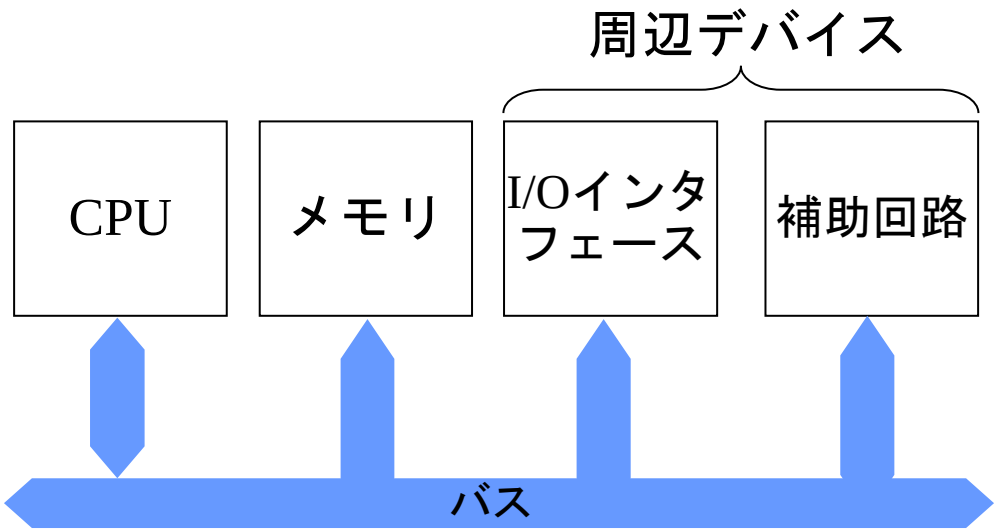
- ハードウェアが応用毎に最適化されて設計される
  - ➡ 質の高いプログラムを開発するためには，ハードウェアアーキテクチャの理解が必要

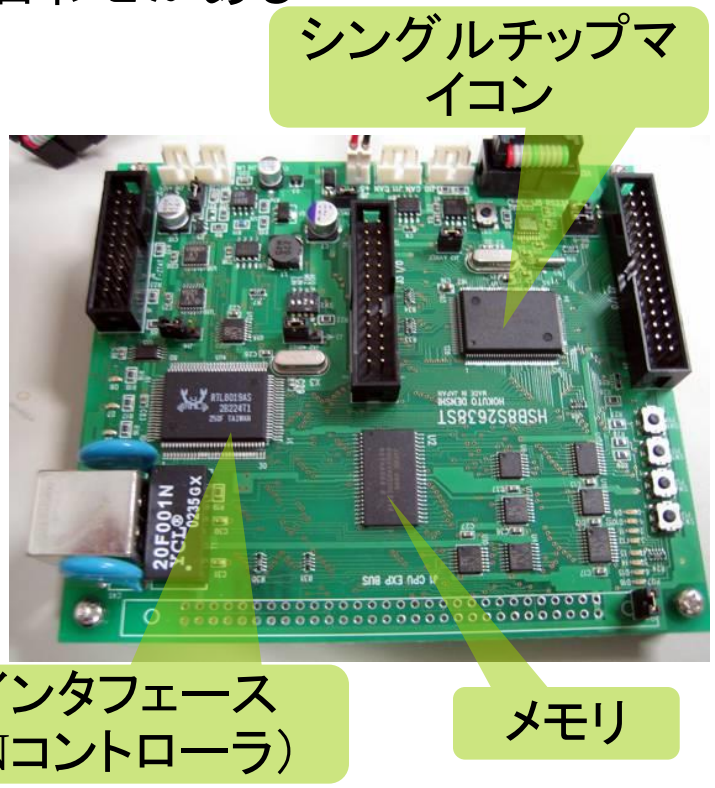
組込みソフトウェア特有の技術

- 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…

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





TOPPERS

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

組込みシステム用プロセッサの多様性

パソコンの現状

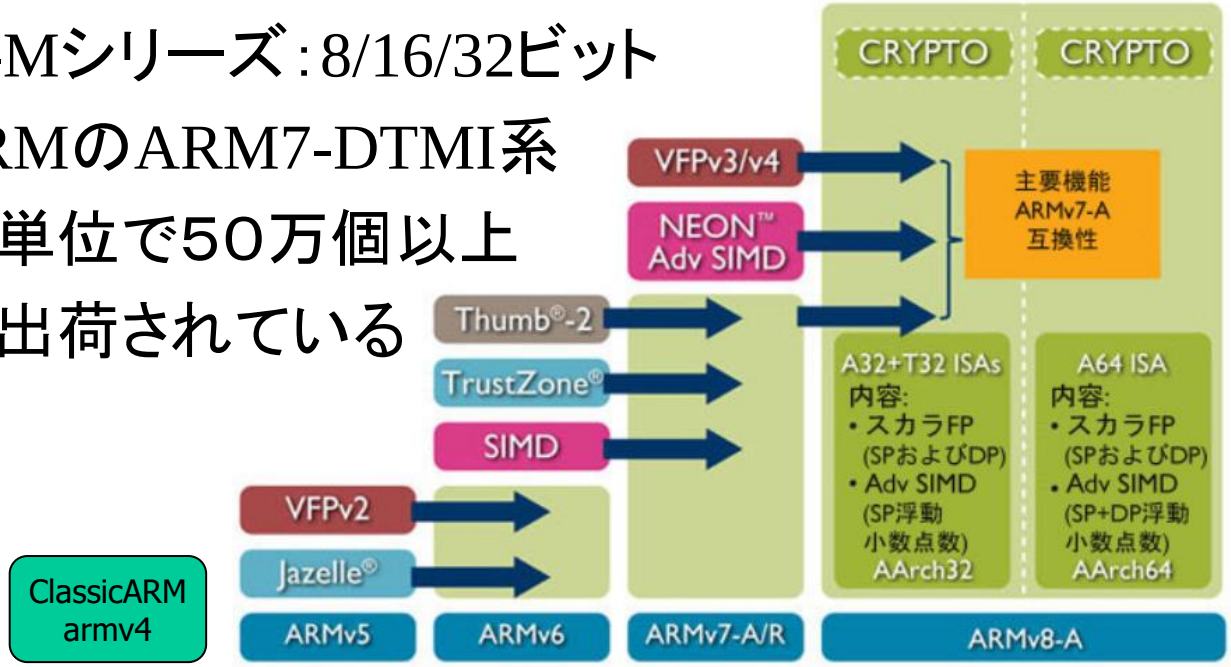
- IA-64(Core-Ix)が主に使われている
  - MACもPowerPCからIA-64へ

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，RISC-V系, MIPS系，PowerPC系，
  - RX，RH，RL系、H8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

ARMプロセッサアーキテクチャ

- ARMプロセッサは現在V8の世代に入り、組み込みからLINUXまで多く使用されるようになった
  - Cortex-Aシリーズ：32/64ビット、マルチプロセッサ
  - Cortex-Rシリーズ：32ビット、ARMv4互換
  - Cortex-Mシリーズ：8/16/32ビット
- Classic ARMのARM7-DTMI系は四半期単位で50万個以上（2015年）出荷されている



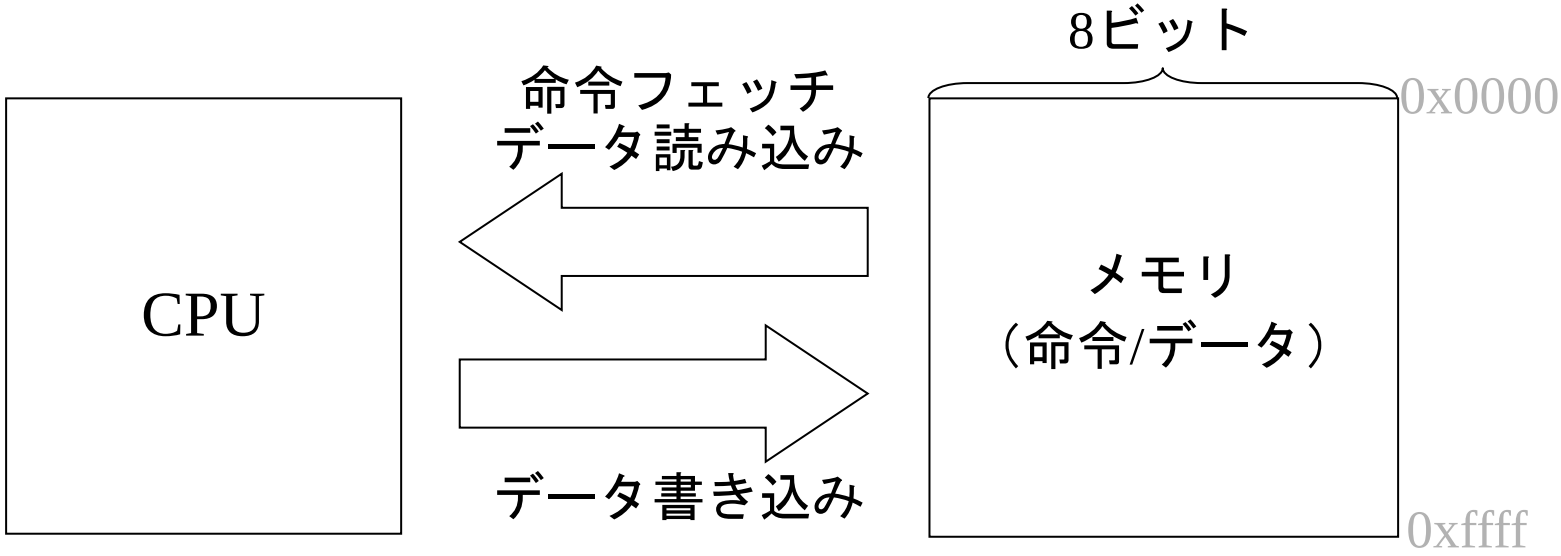
ARM The Architecture for the Digital Worldより参照

TOPPERS

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



プロセッサ：データ

データは命令と同様にメモリに置く

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

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

ポインタ
- 変数が格納されているメモリのアドレス

グローバル変数

C言語プログラム

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

コンパイル結果：機械語

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

アドレス

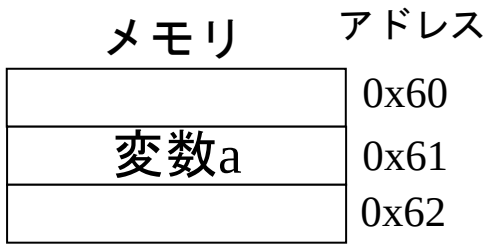
dの値は0x61

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

0x61への書き込み

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

メモリへの配置

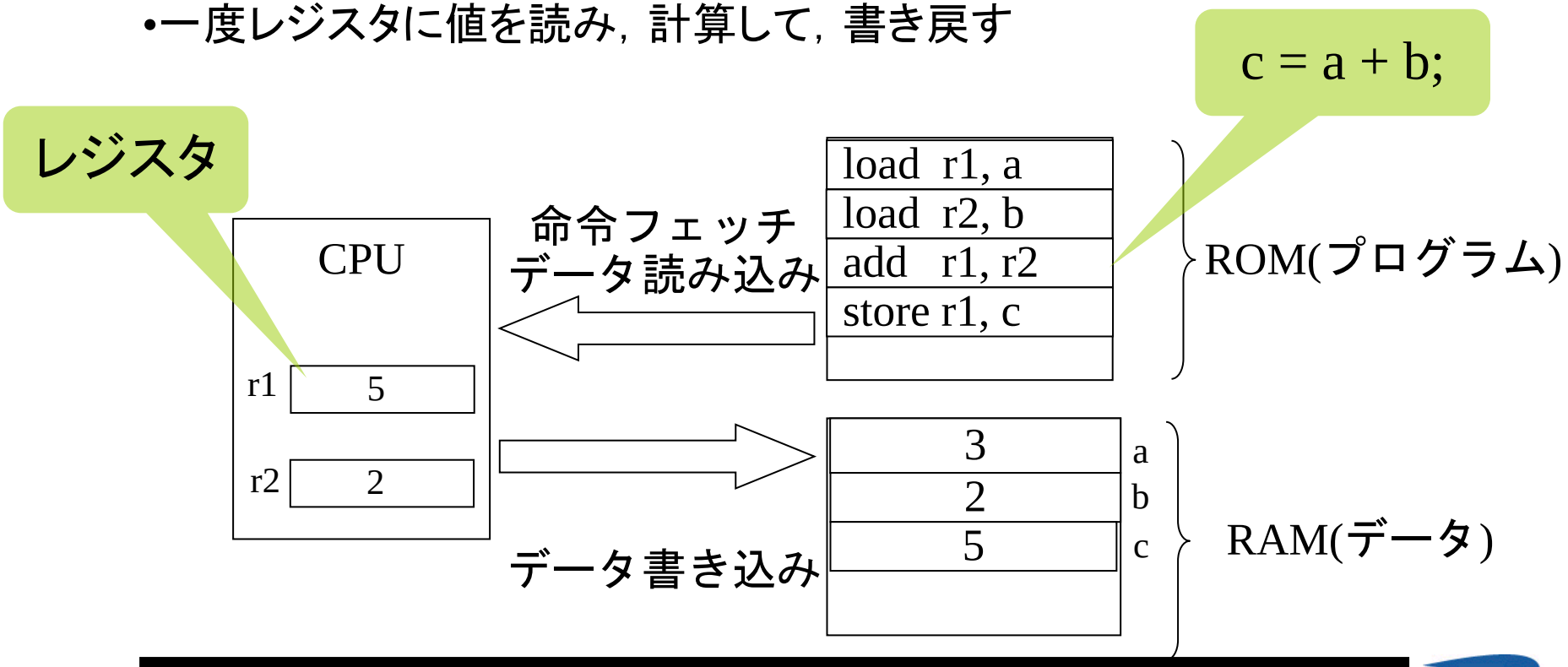


TOPPERS

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上の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



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：コントローラレジスタ）

汎用レジスタ

メモリから読み込んだデータや演算結果を保持する（アキュームレータ）

メモリの場所（アドレス）を保持する（アドレスレジスタ）

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（スタックポインタ）

専用のレジスタ

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

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

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

TOPPERS

Cortex-Mのレジスタ構成

- 実行モードはスレッドモード（特権モードと非特権モード）、ハンドラモード3つ
- CPSRレジスタは3つのレジスタに分解された
  - APSR,IPSR,EPSR
- FIQ割込みはなくなり、割込み発生時、スタック上に使用するレジスタをハードウェアでPUSHする
- 割込み復帰はBX命令のエクセプションモードで行われる

ユーザー	特権モード
R0	R0
R1	R1
R2	R2
R3	R3
R4	R4
R5	R5
R6	R6
R7	R7
R8	R8
R9	R9
R10	R10
R11	R11
R12	R12
R13	R13_svc
R14	R14
PC	PC

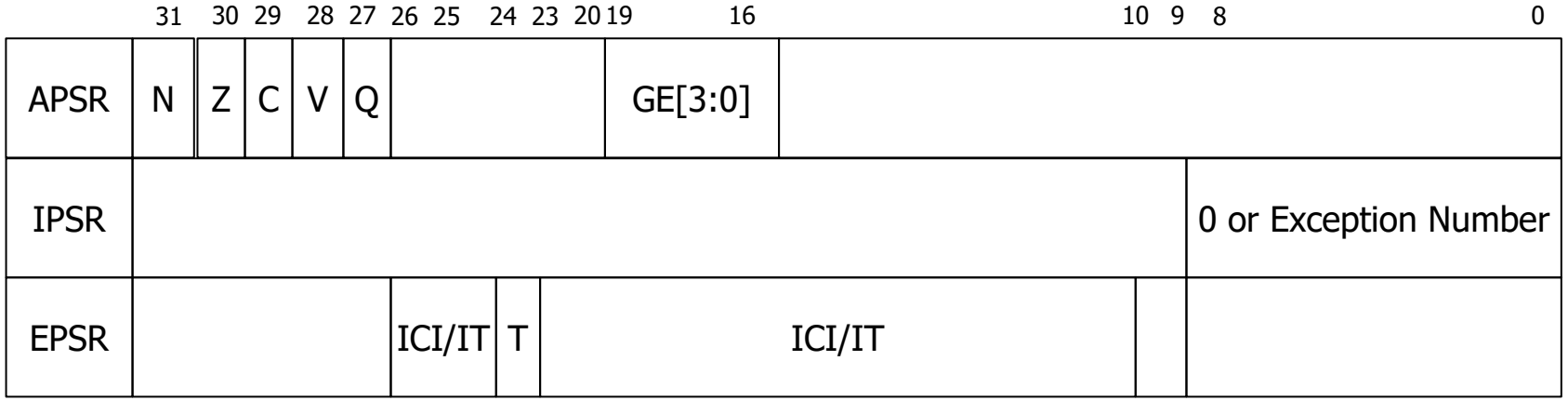
ARSP	APSR
IPSR	IPSR
EPSR	EPSR

特権モード移行時R13（SP）のみ変更となる

TOPPERS

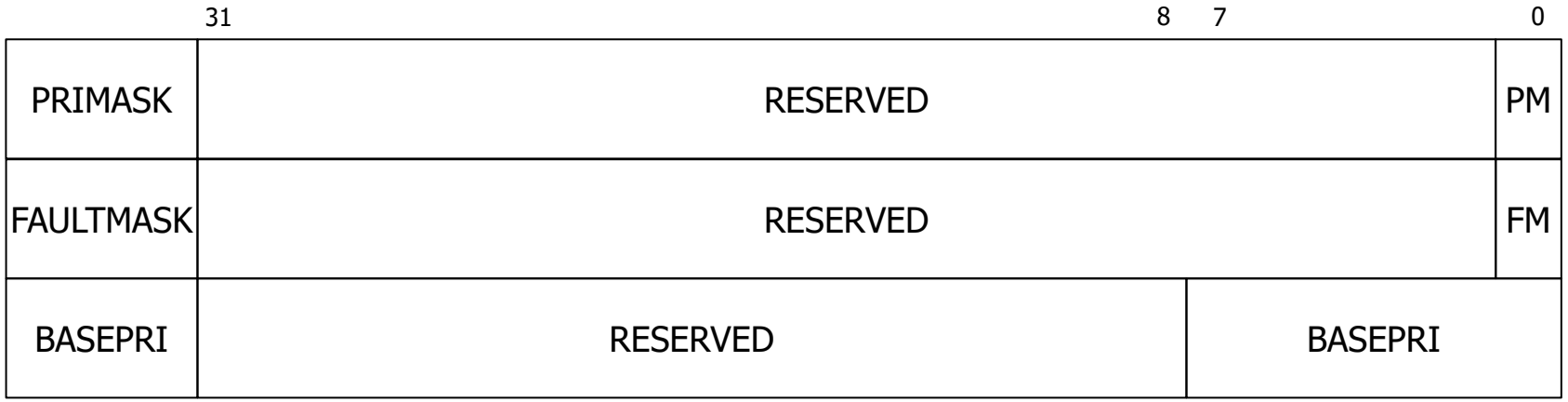
専用プログラムステータスレジスタ

- 専用プログラムステータスレジスタ、特権モードでMRS,MSR命令で設定
- APSR：アプリケーションPSR：演算結果のフラグレジスタ
- IPSR：割込みPSR：割込み番号を保持
- EPSR：実行PSR：Thumb-2の実行状態の保持



優先順位用特殊レジスタ

- 割込みマスクを行うレジスタ、特権モードでMRS,MSR命令で設定
- PRIMASK：PM=1実行優先権=0でNMIとハードフォールト以外の割込みをすべて禁止する
- FAULTMASK：FM=1実行優先権=-1でNMI以外のすべての割込みを禁止する
- BASEPRI：BASEPRI=N優先度以下の割込みをすべて禁止



CONTROLレジスタ

- nPRIV,bit[0]:0 特権モード、1 非特権モード
- SPSEL,bit[1]:0 SP_main（特権、ハンドラ）、1 SP_process
- FPCA,bit[2]:0 FPの実行なし、1 FPの実行あり
- スタックに関して
  - スレッドモード:特権モード(SP_main)、非特権モード(SP_process)
  - ハンドラモードでは、SP_mainが選択される

組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：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

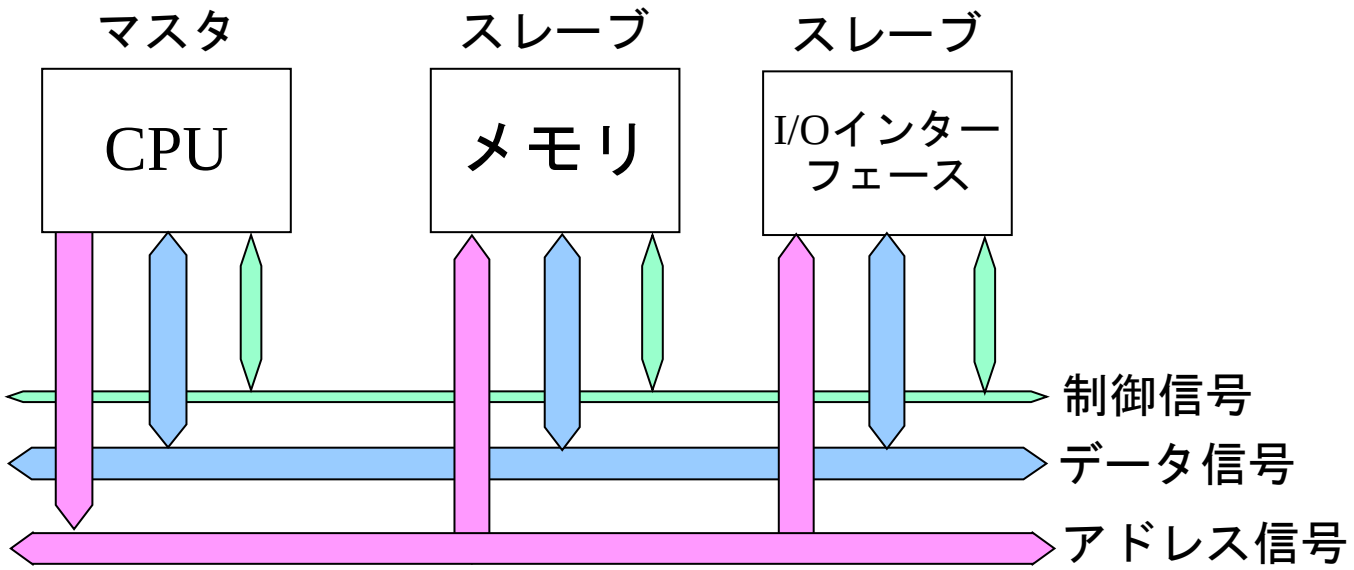
- バスから読み込み・書き込みの要求を受ける
- メモリ，I/Oインタフェース（GPIO,UART），…

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

制御信号
- 信号毎に異なる

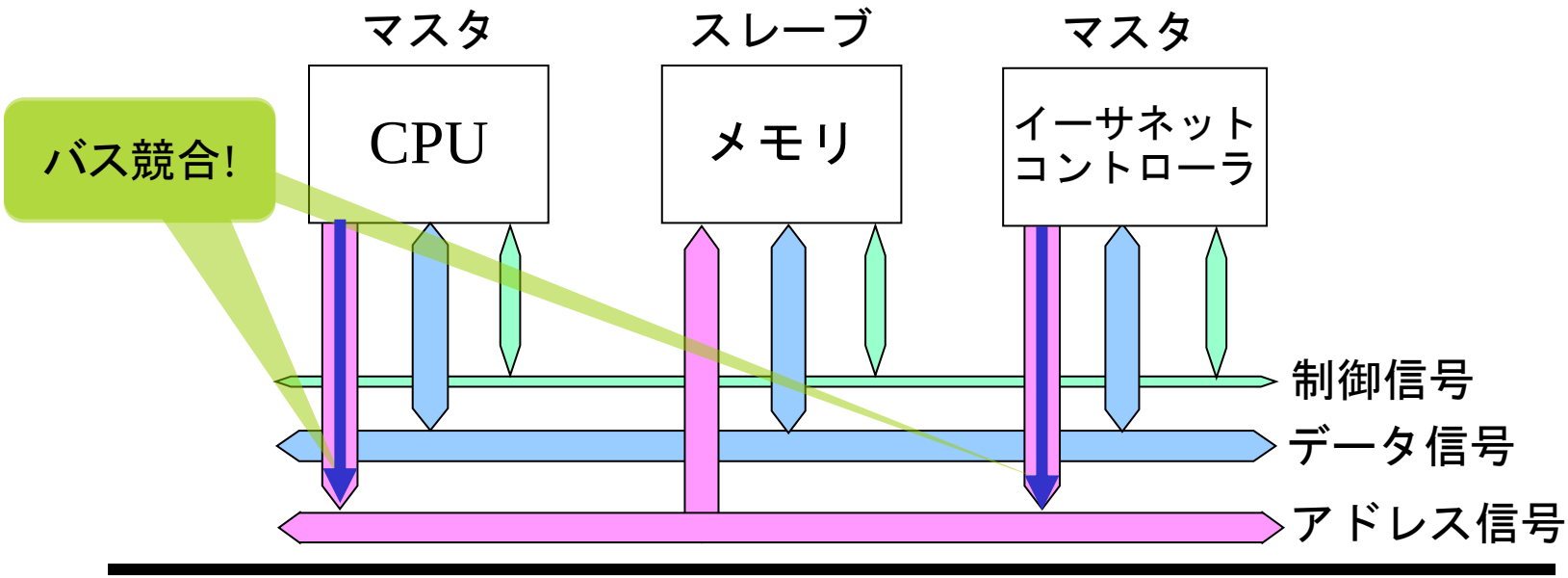


バス：バス競合

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…



TOPPERS

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（使用権）の取得を試みる
- バス権を取得したら，読込み・書込みを行う
- 読込み・書込みの終了後，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

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

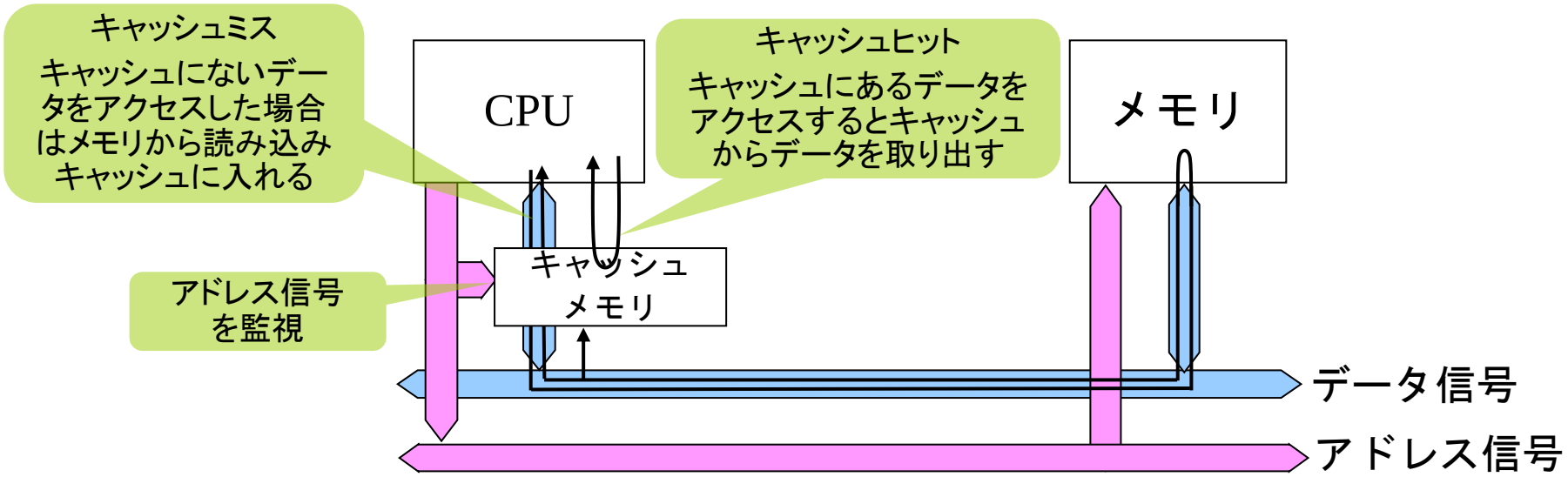
キャッシュ（1/2）

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



TOPPERS

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

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

エンディアン(2/2)

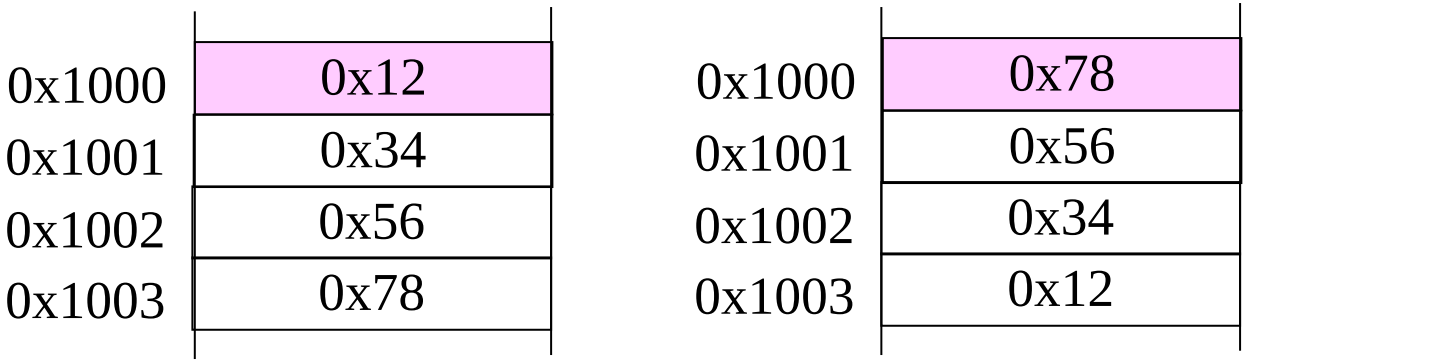
プロセッサのエンディアン

- 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



TOPPERS

組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

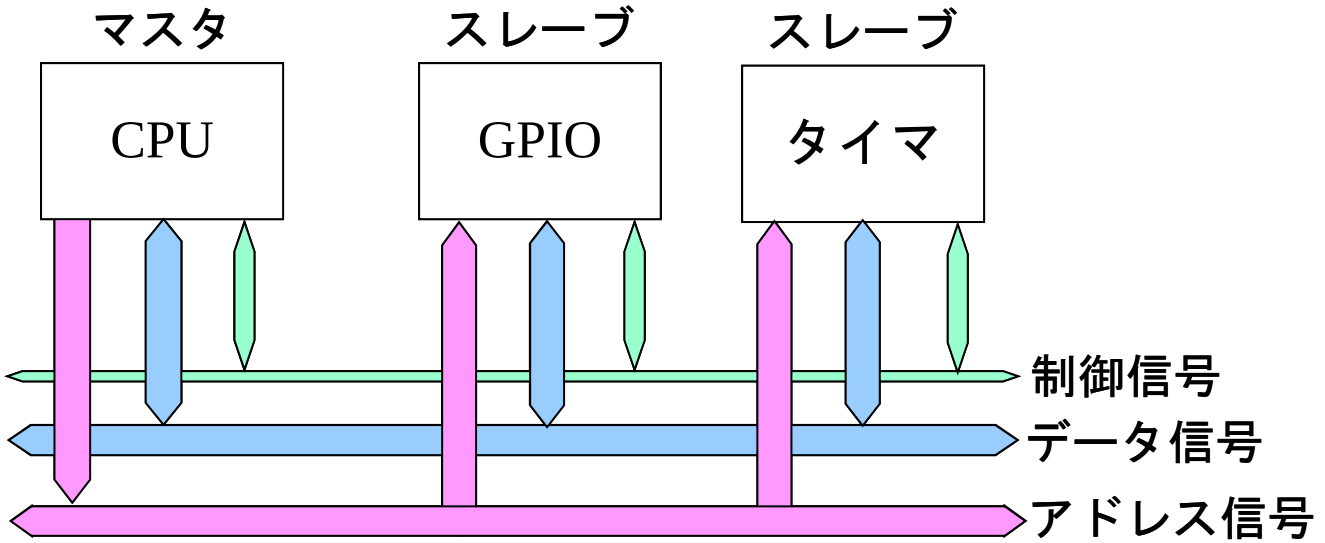
周辺デバイス

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



周辺デバイス：デバイスレジスタ

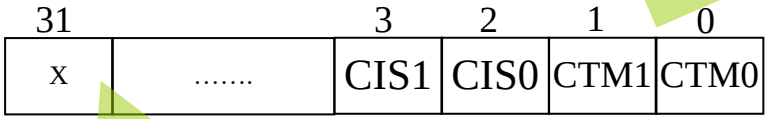
プロセッサと周辺デバイスとのインタフェース

- 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（LPC23XXタイマ）

名称	アドレス	R/W	アクセスサイズ
タイマ0コントロールレジスタ	0xE0004004	R/W	32 (4bit)
タイマ0マッチコントロールレジスタ	0xE0004014	R/W	32 (12bit)
タイマ0マッチレジスタ	0xE0004018	R/W	32

タイマ０コントロールレジスタの構成



動作モードを設定
00:タイマモード
01:カウンタモード(R)
10:カウンタモード(F)
11:カウンタモード(B)

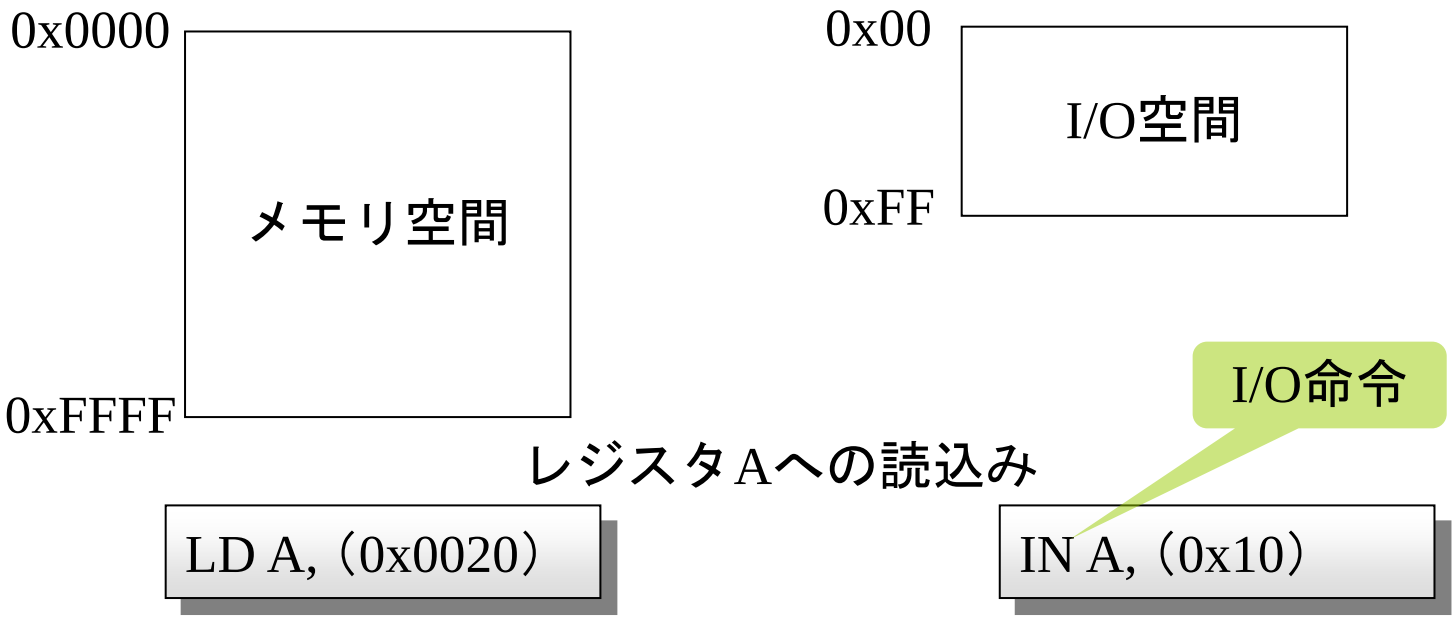
動作モ入力を指定
00:CAPn.0 for TIMER0
01:CAPn.1 for TIMER1
10:リザーブ
11:リザーブ

TOPPERS

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

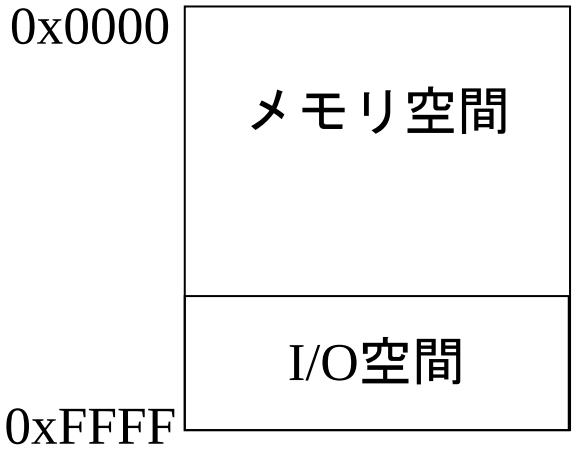
- 周辺デバイスのアクセス時にはキャッシュを無効にしなければならない.

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



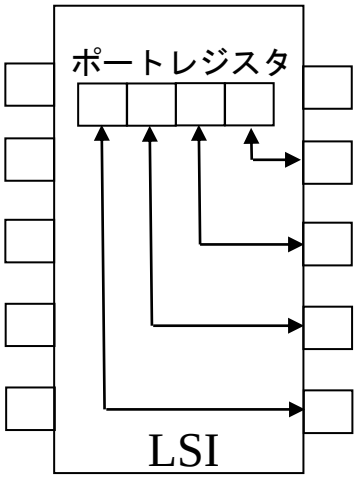
周辺デバイス：GPIO（汎用I/Oポート）

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

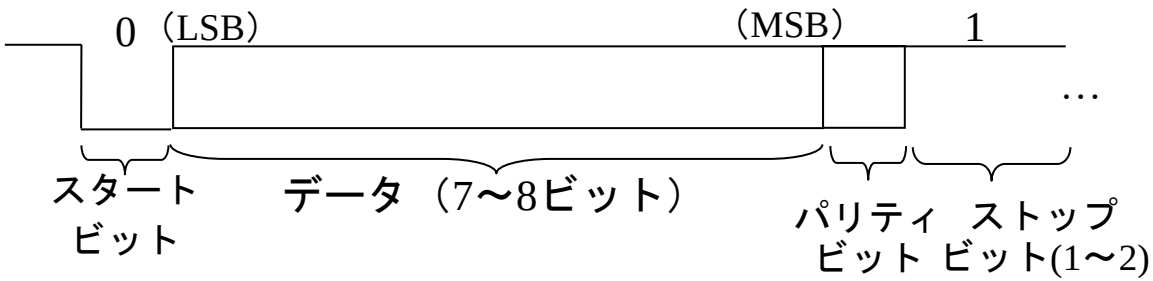
周辺デバイス：SIO（シリアルI/O）

単一のデータ線を用いて外部と通信を行う方式

– UART（RS-232C），I^2C，etc…

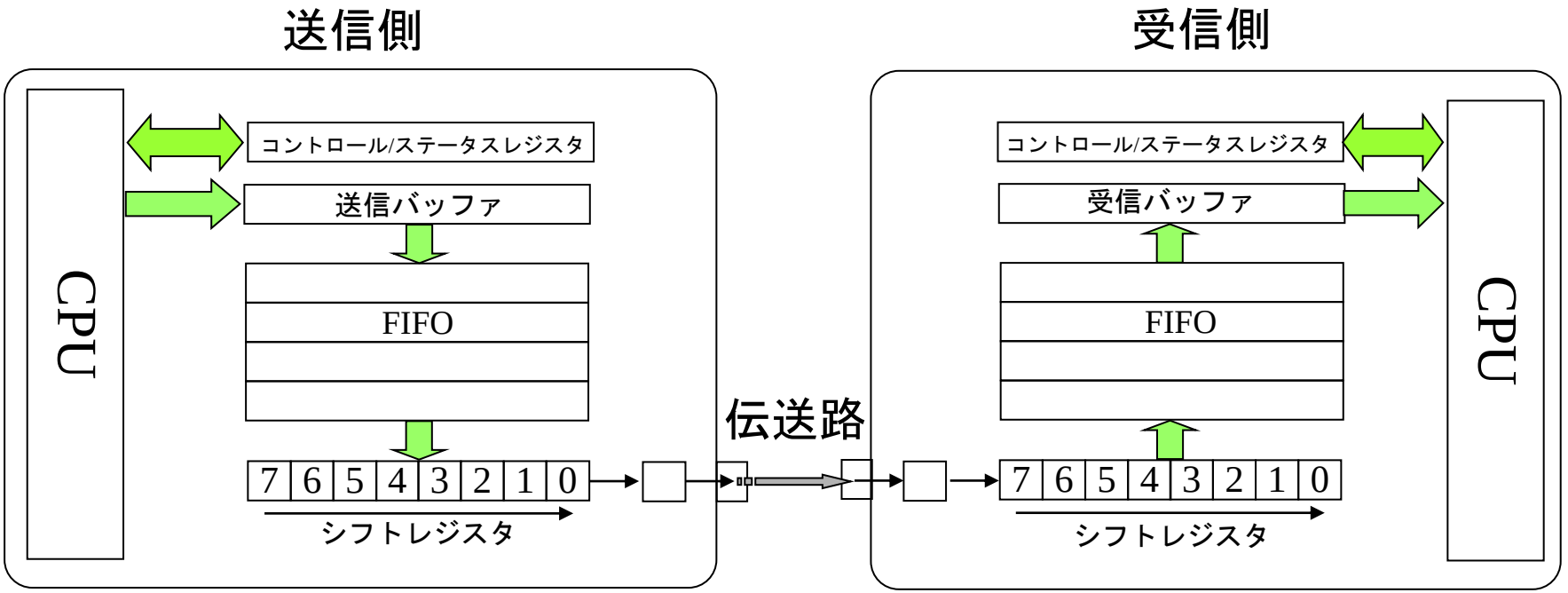
プロトコル

– データー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



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



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

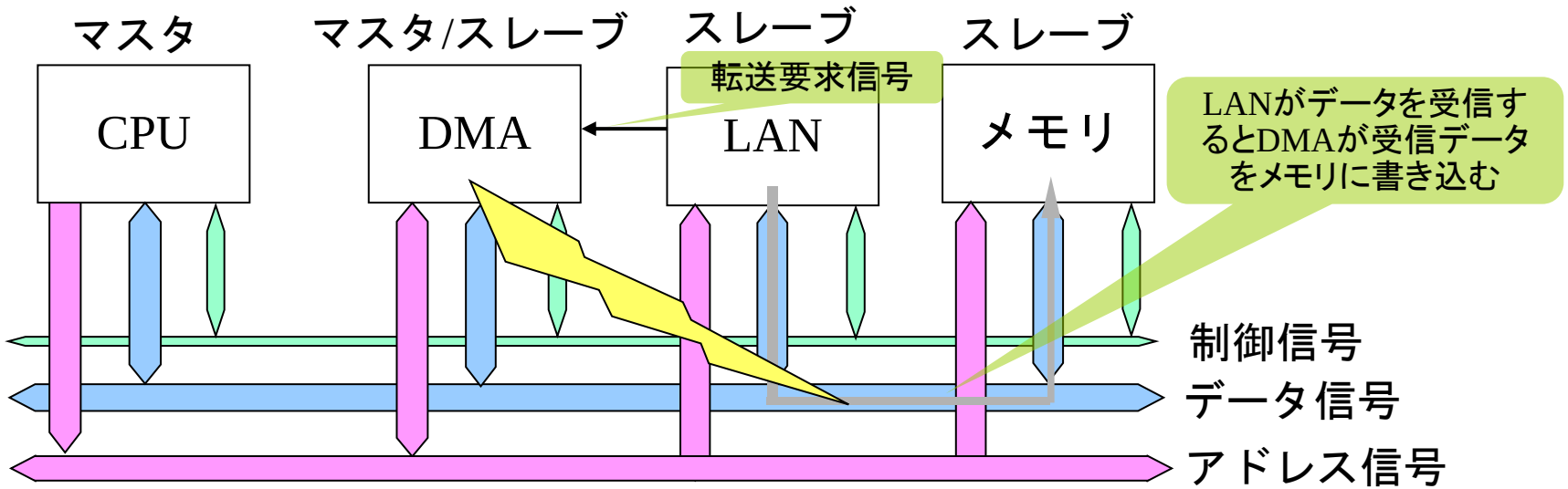
ハードウェアタイマ

- 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

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



組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

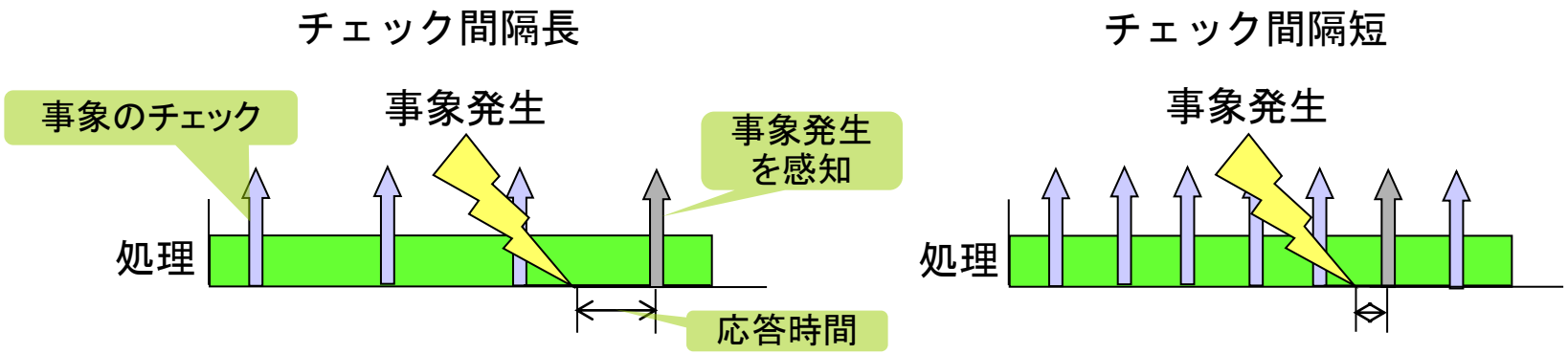
外部事象の待ち方：ポーリング

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



外部事象の待ち方：割込み

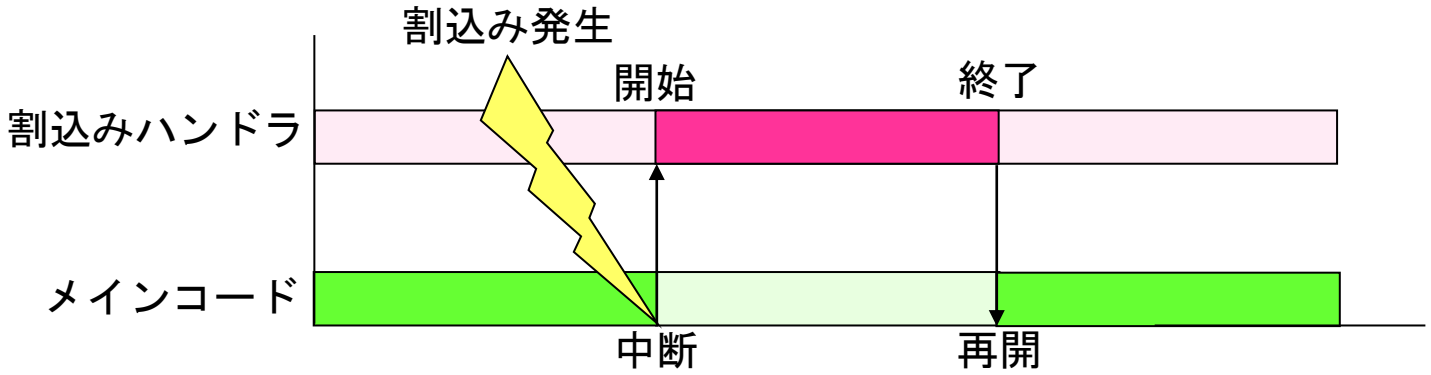
プロセッサが事象の発生を非同期に受け付ける方法

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



TOPPERS

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

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！

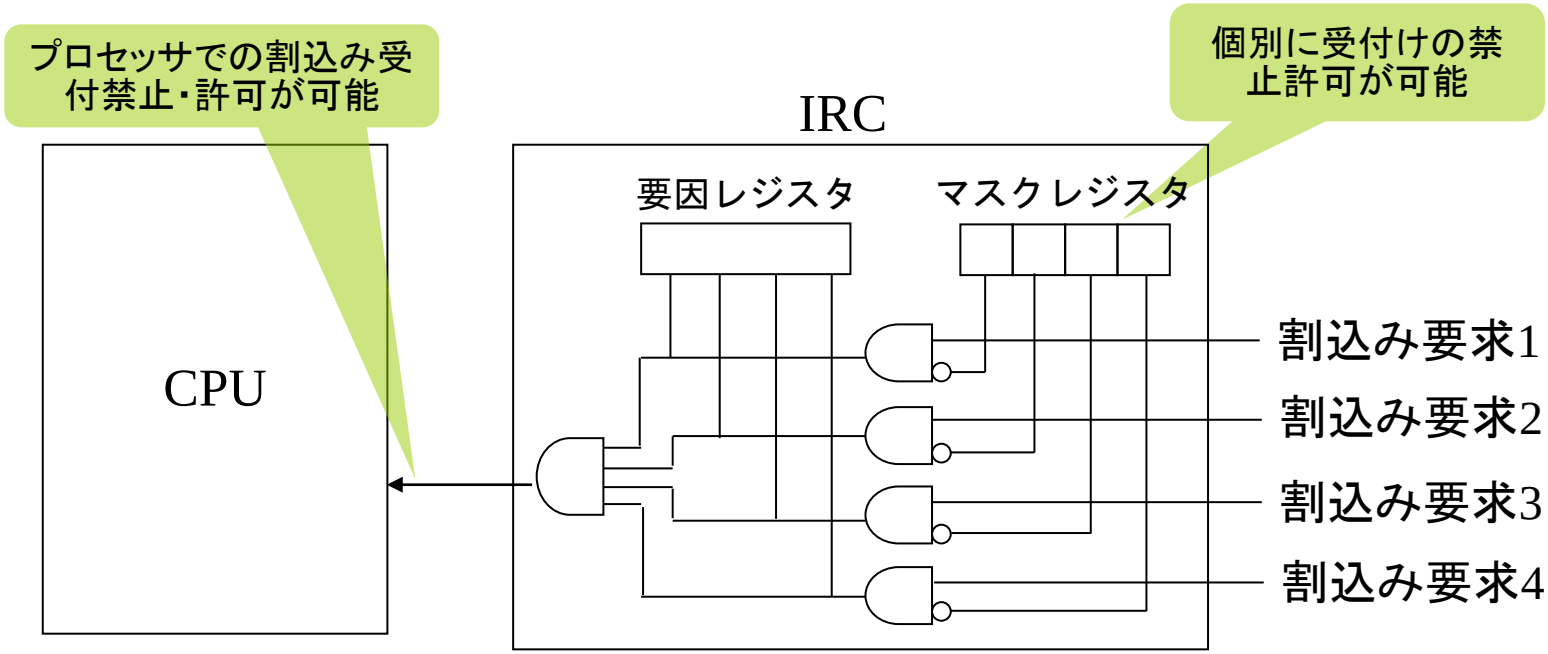
多様な割込みアーキテクチャ

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簡単化する傾向にある
  - 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



エッジトリガとレベルトリガ

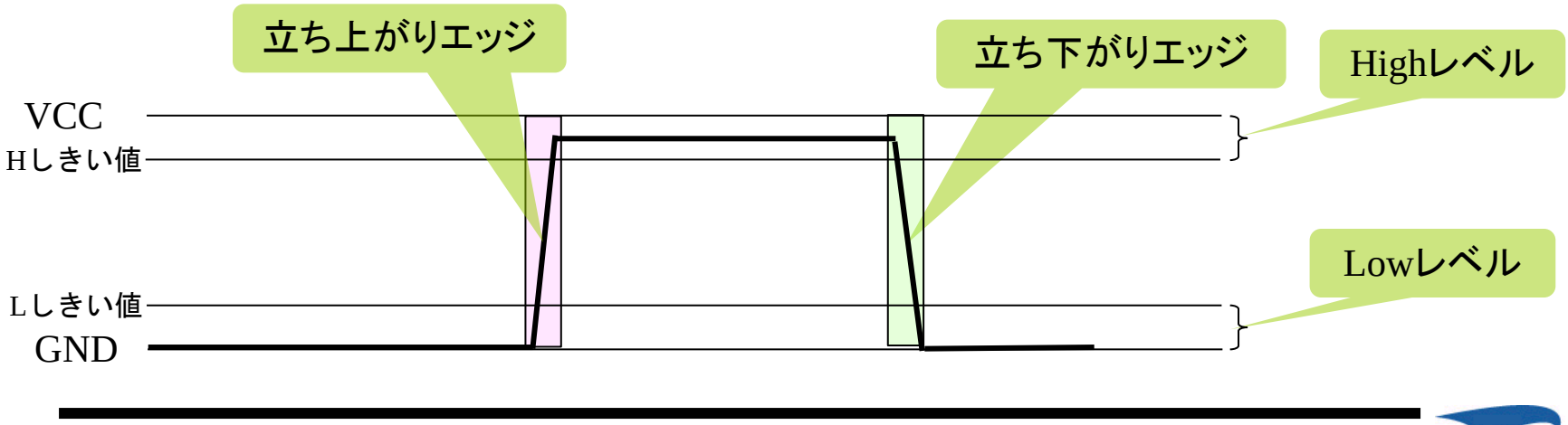
割込み要求の発生方法

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



TOPPERS

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

割込みベクタテーブル

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

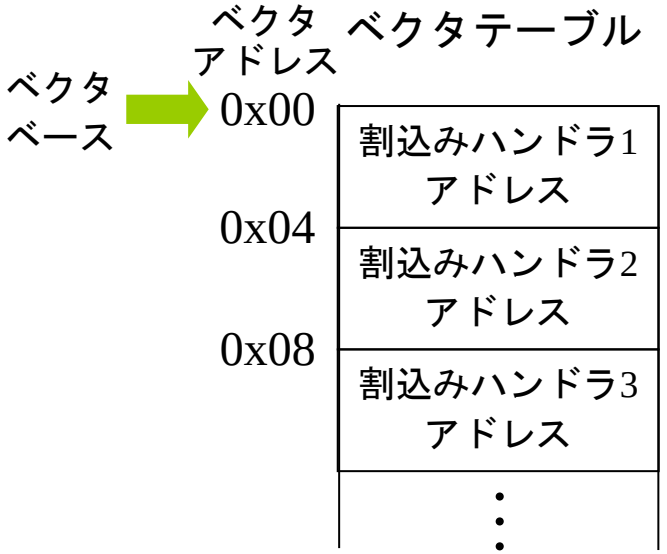
- プロセッサは割込みを受け付けると割込み要因を判定して，その要因に対応した割込みハンドラの先頭アドレスを読込みジャンプする
- CISC型のプロセッサで一般的な方式

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



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.

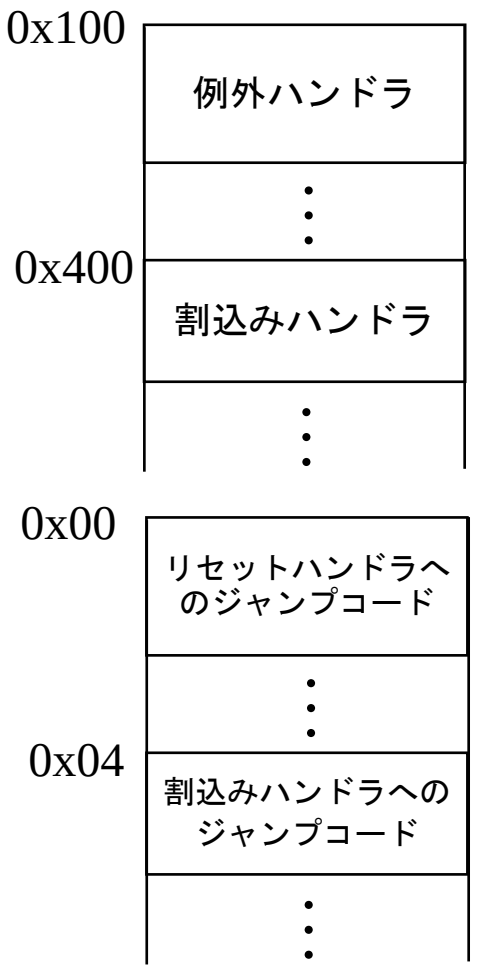
ベクタの間が4byteであるため，それぞれのアドレスにハンドラへのジャンプコードを置く

– 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



TOPPERS

プロセッサの割込み/例外処理の例

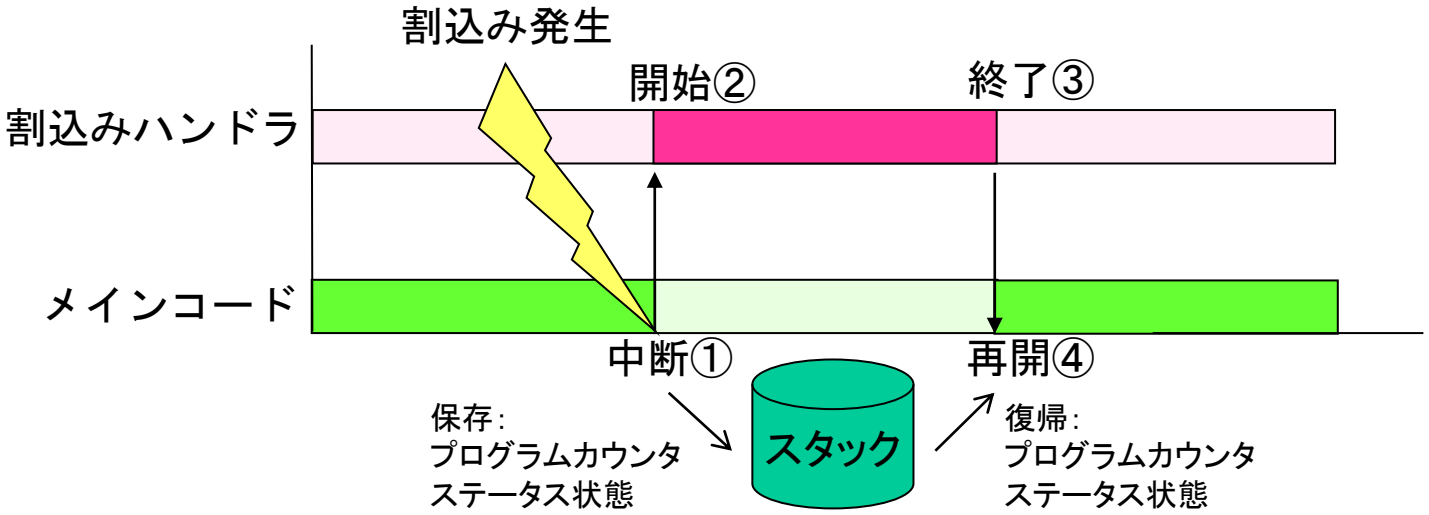
①メインコード実行中に割込みを受け付けると，割込み要因に対応したベクタテーブルに登録されているアドレスにジャンプする．

その際、メインコードで実行中のプログラムカウンタやステータス状態は、スタックに保存される．

②割込みベクタから割込みハンドラが実行

③割込みハンドラが終了すると、保存していたステータス状態に復帰

④保存していたメインコードのプログラムカウンタから再開する



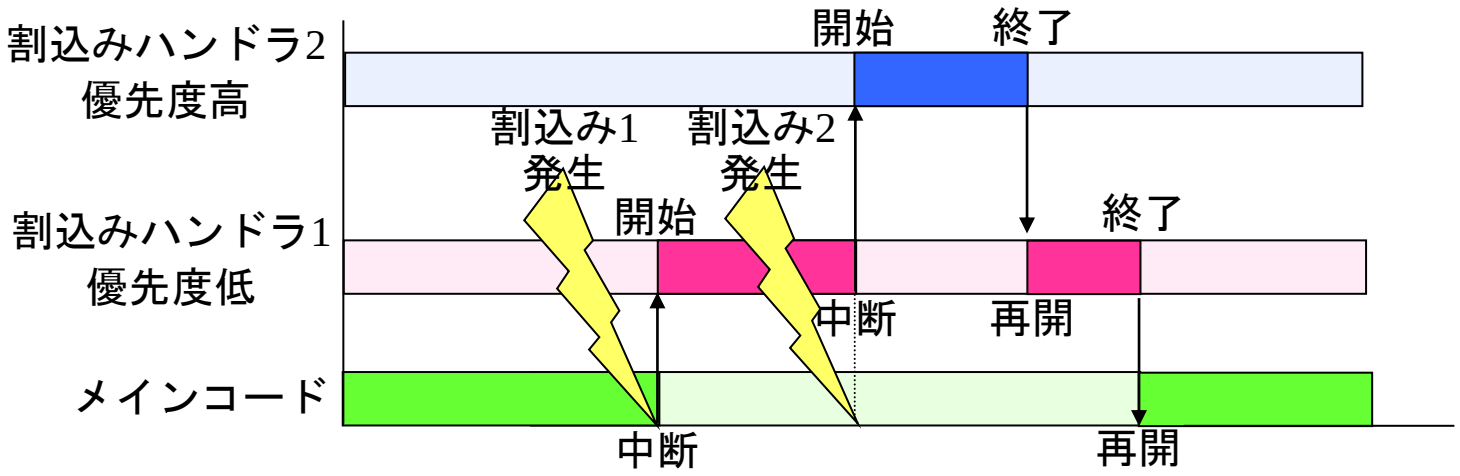
TOPPERS

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

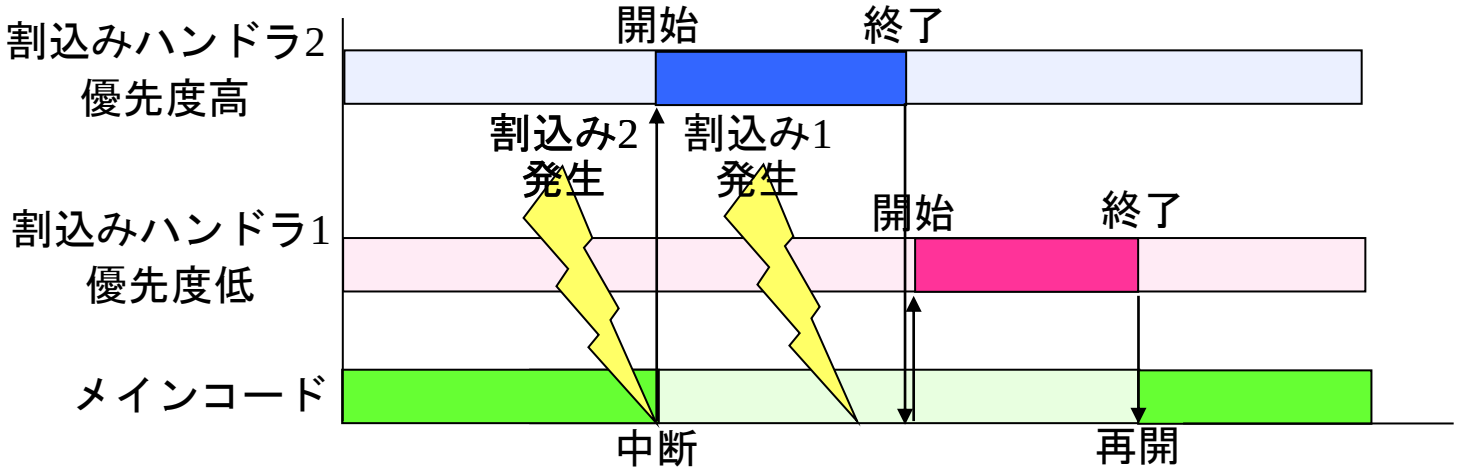
- 各割込み要因に優先度を持たせる．プロセッサは割込みレベルを持ち（可変），現在の割込みレベル以下の優先度を持つ割込みが発生しても，割込みを受け付けない
- 割込み毎に適切に優先度を設定する必要がある



TOPPERS

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



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

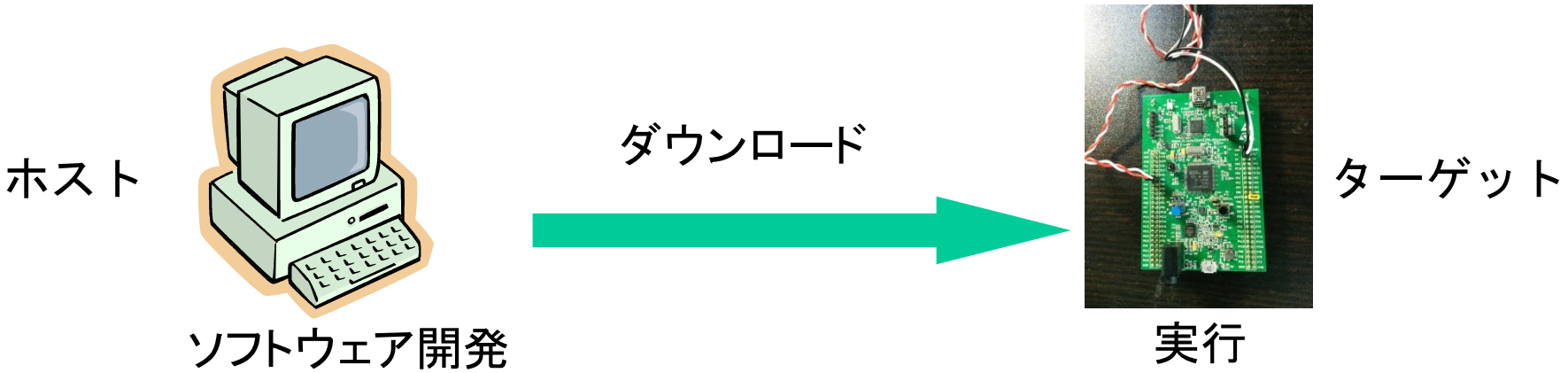
組込みシステムはクロス開発により開発する

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

クロス開発環境の説明は、開発環境編「ソフトウェア環境の構築」の章を参照



TOPPERS

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
    - -EB : Use big-endian byte order
    - -EL : Use little-endian byte order
    - -mthumb : USE THEMB CODE
    - -mcpu=xxxx : CPU type
    - -mfpu=yyyy : FPU type

クロスコンパイラ：プリプロセス

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_ */
```

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

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

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

armv4の例

CPSRレジスタ取得

```
inline uint32_t current_sr(void ){
  uint32_t sr;
  asm(“mrs %0,CPSR”:”=r” (sr));
  return sr;
}
```

CPSRレジスタ書き込み

```
inline void set_sr(uint32_t sr){
  asm(“mrs CPSR,%0”::”r” (sr):”cc”);
}
```

割込み許可

```
inline void locl_cpu(void){
   set_sr(current_sr() | 0x80);
}
```

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（浮動少数点演算関連）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))

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

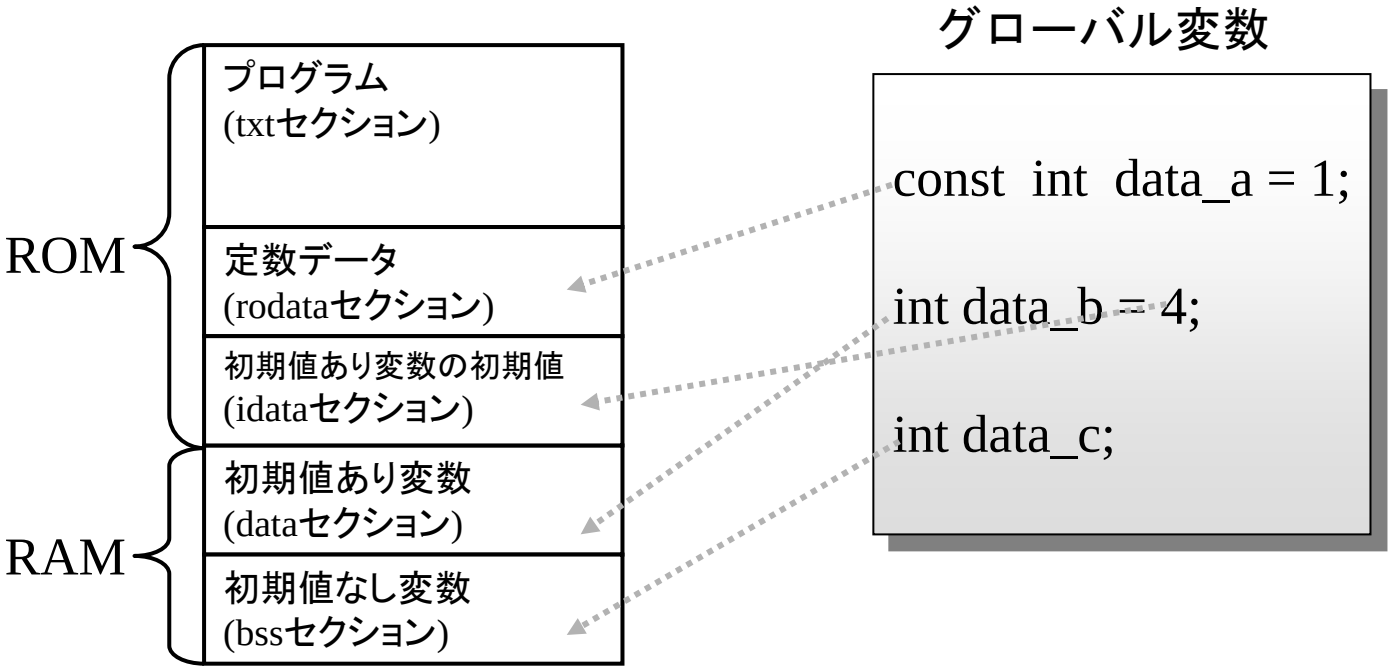
- セグメントとも呼ばれる
- コンパイラ，アセンブラが出力するプログラムまたはデータの固まりをセクションと呼ぶ
- セクションにはそれぞれプログラム属性を持っている
- 標準的なコンパイラは，次のセクションを生成する（呼び方はコンパイラによって異なる）
  - text　　：プログラムコード
  - rodata　：定数データ（文字列）
  - data　：初期値を持つデータ
  - bss　：初期値を持たないデータ（0に初期化される）

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

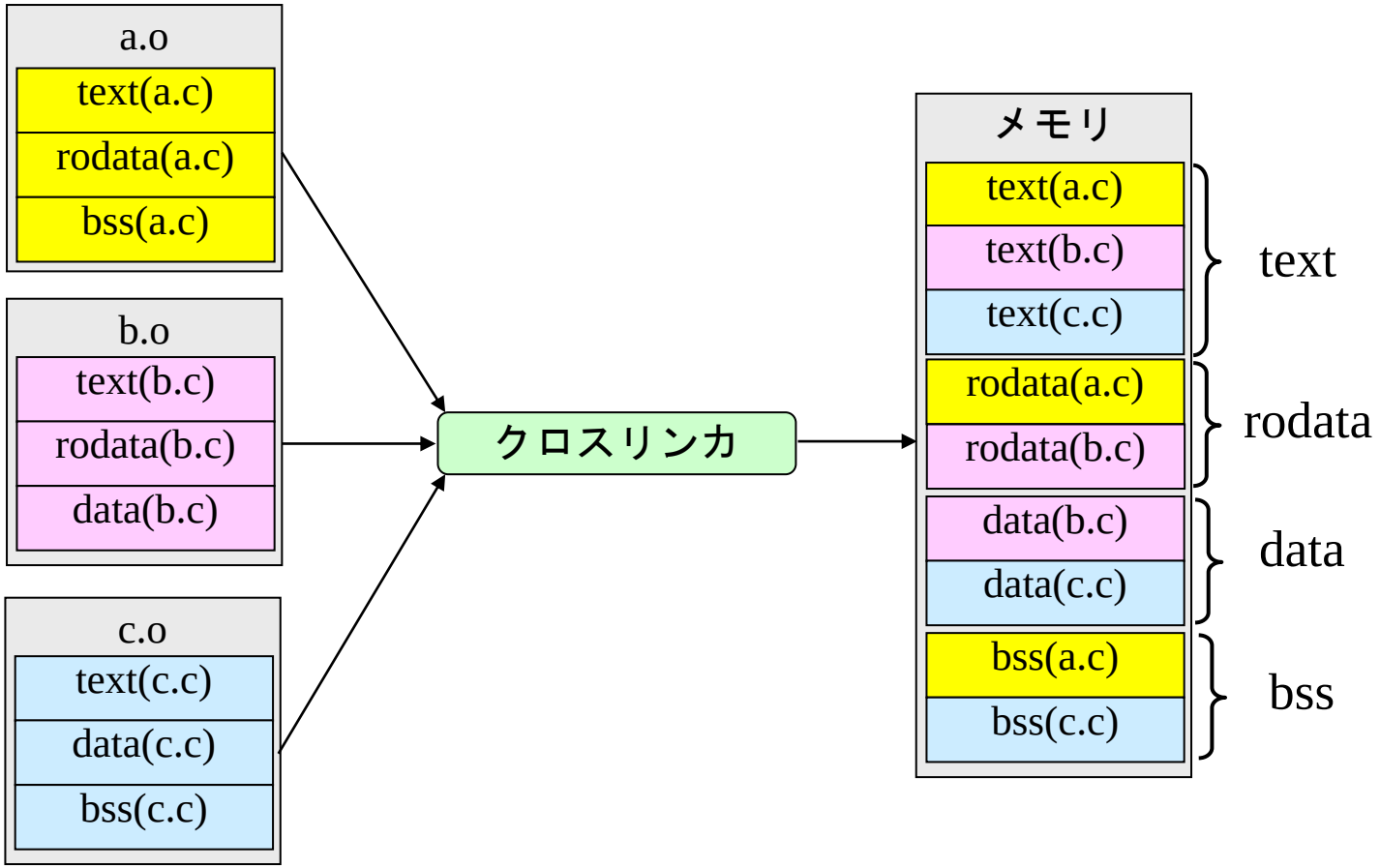
➡ メモリロケーションの指定（後述）

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



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：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）

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
…..
}
```

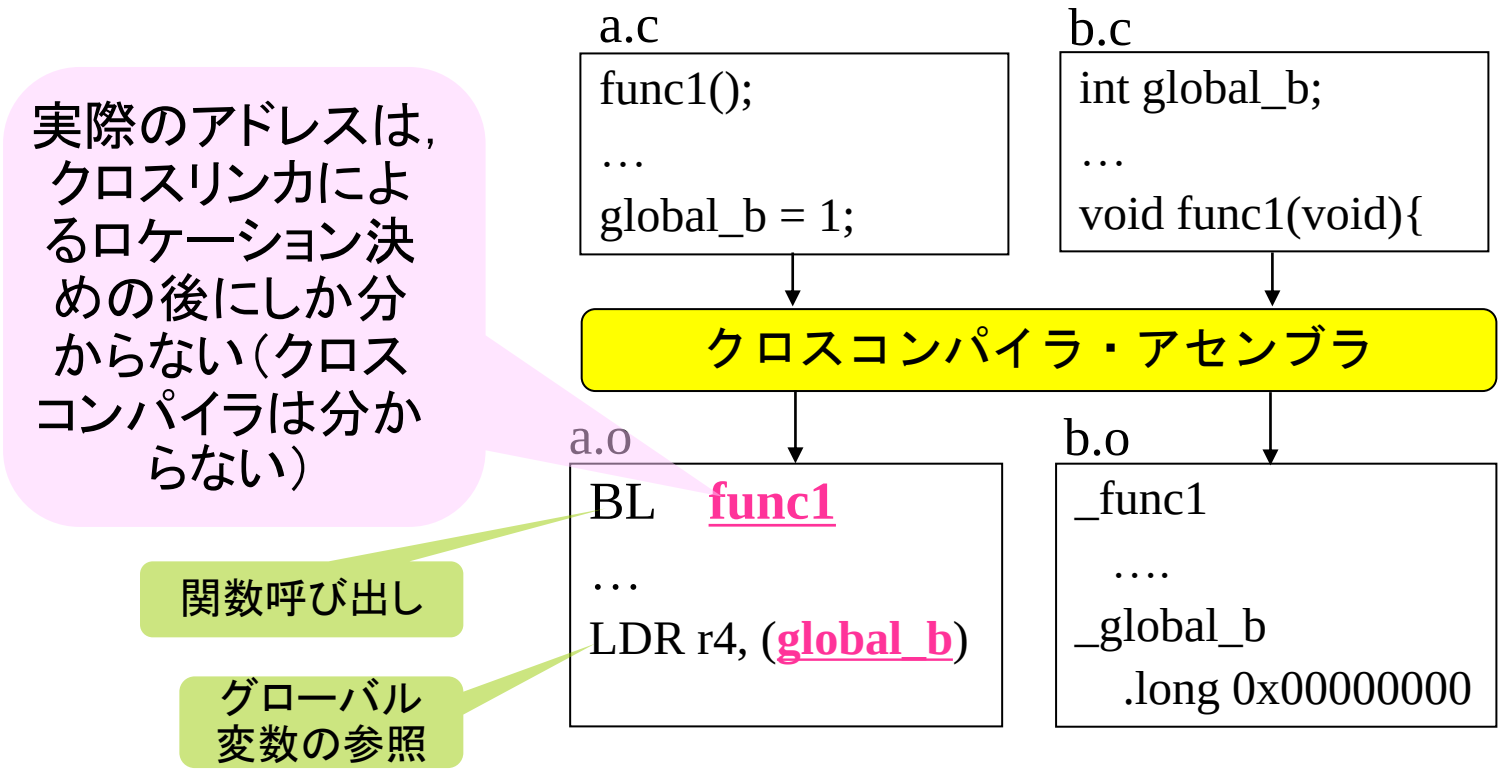

TOPPERS

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

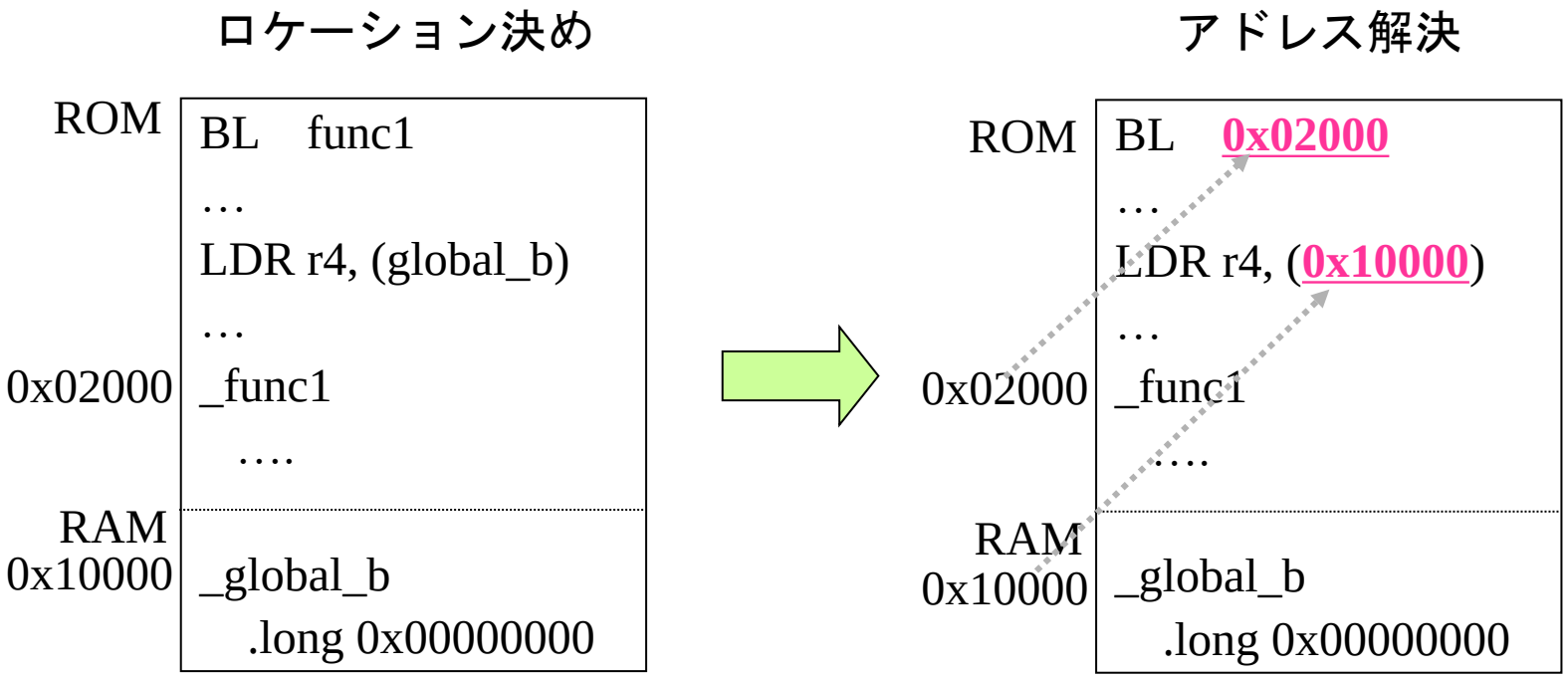


TOPPERS

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

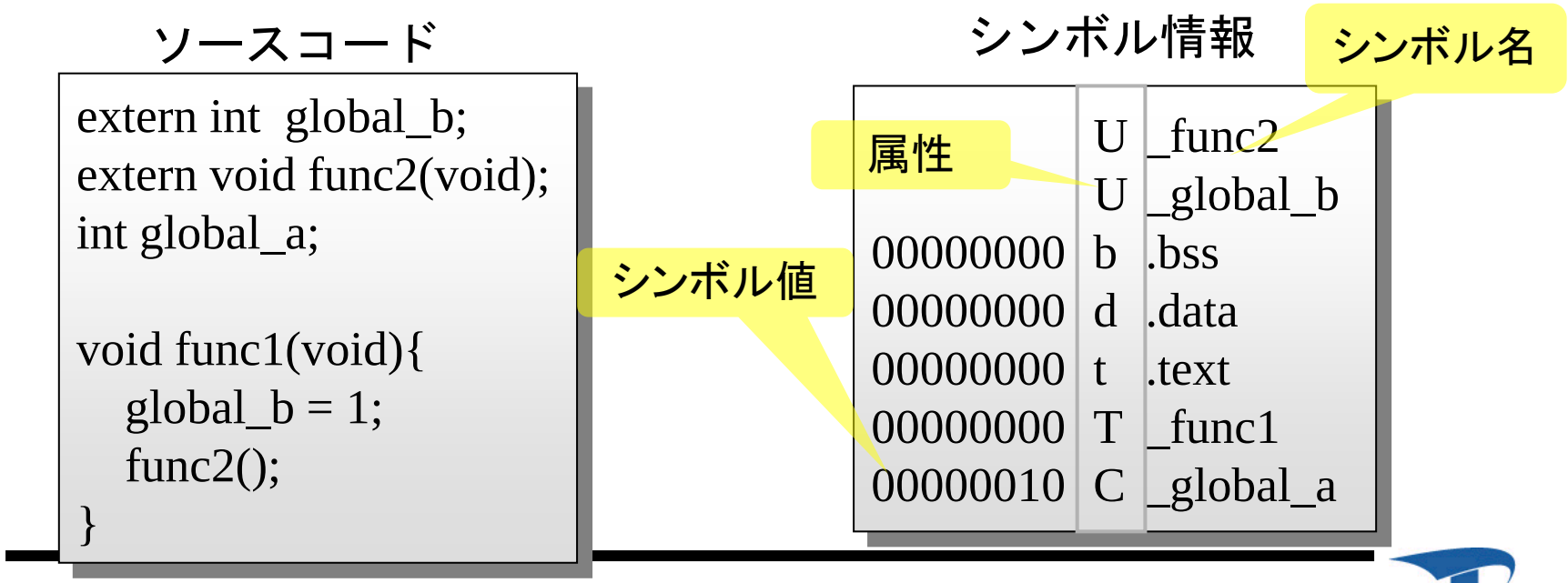


クロスリンカ：シンボル解決の実現方法

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）



TOPPERS

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ければ，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）

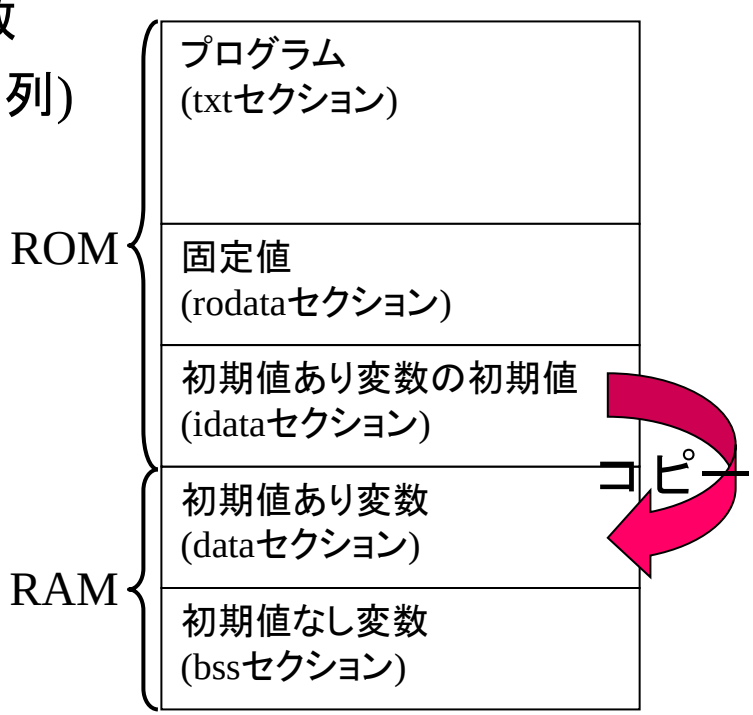
```
000081e0 T func1
000081c4 T func2
0000a6e8 B global_a
0000a6e4 B global_b
```


TOPPERS

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，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



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

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）

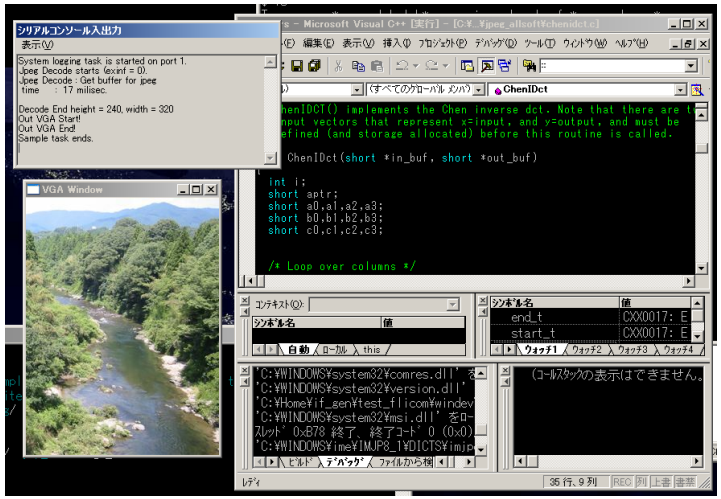
- ソースコードデバッグ機能は持たない

リモートデバッガ（対デバッガ）

- 高度なデバッガ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

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）



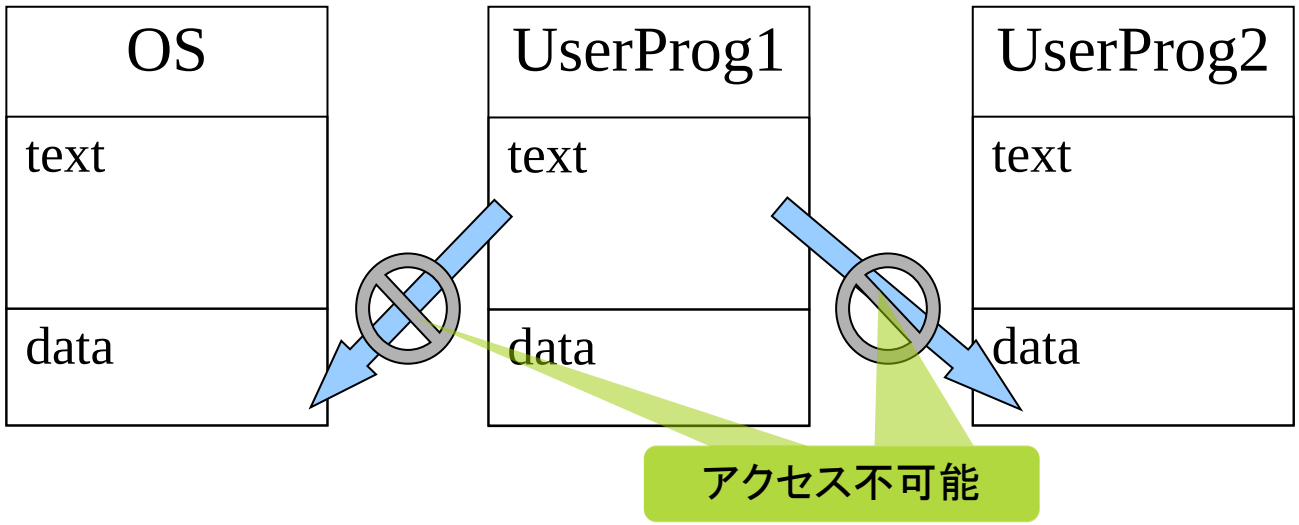
TOPPERS

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

汎用PC用ソフトウェアのランタイム構造

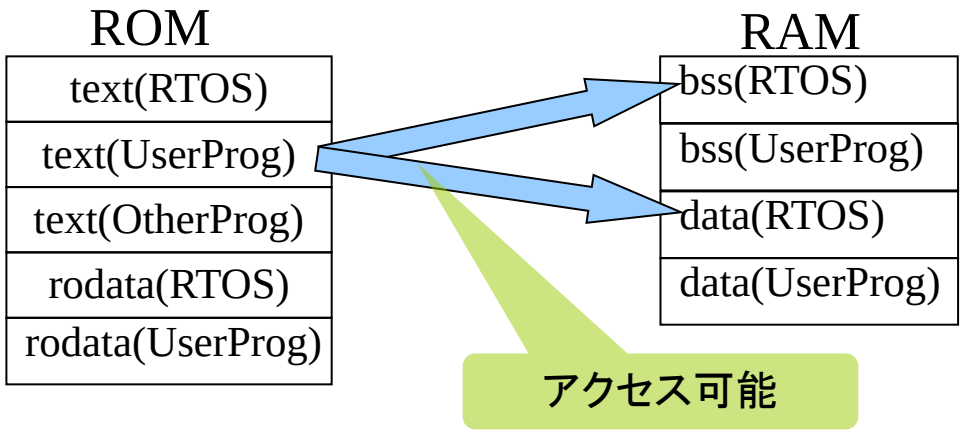
- 各プログラムは，独立したメモリ空間（論理空間）で動作
  - メモリマネージメントユニット（MMU）によるマッピングとメモリ保護
  - 他のプログラムやOSのメモリ空間へのアクセスは不可能
  - プログラムに不具合があり，正しくないメモリ空間にアクセスした場合は，アクセス違反で終了されるだけで，システム全体に影響は及ぼさない



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

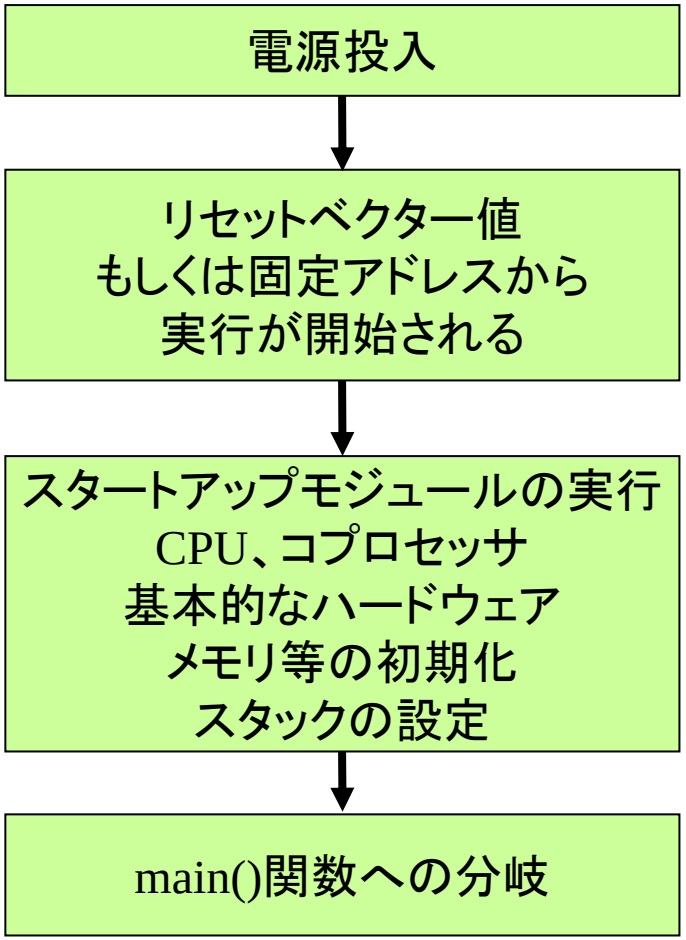
- ユーザープログラムがリアルタイムOSを含む他のプログラムやデータとミックスされて配置され，それぞれは保護されていない
  - ➡ MMUを用いたメモリ保護は実行オーバーヘッドが大きく，リアルタイム性の弊害となる
- ユーザープログラムは容易にリアルタイムOSのデータを破壊可能
  - ➡ システム内のソフトウェアは信頼できるという前提
- メモリアクセス関連のデバッグが困難



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



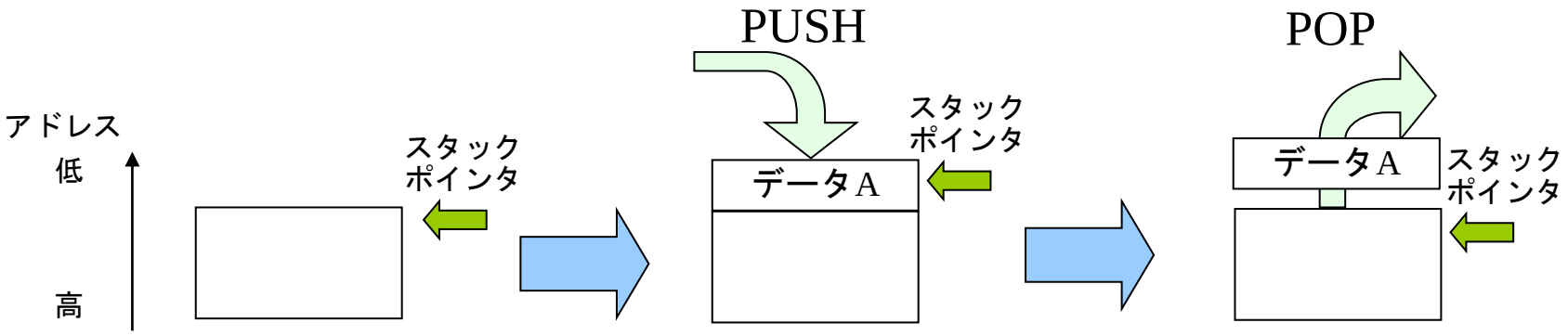
スタック

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



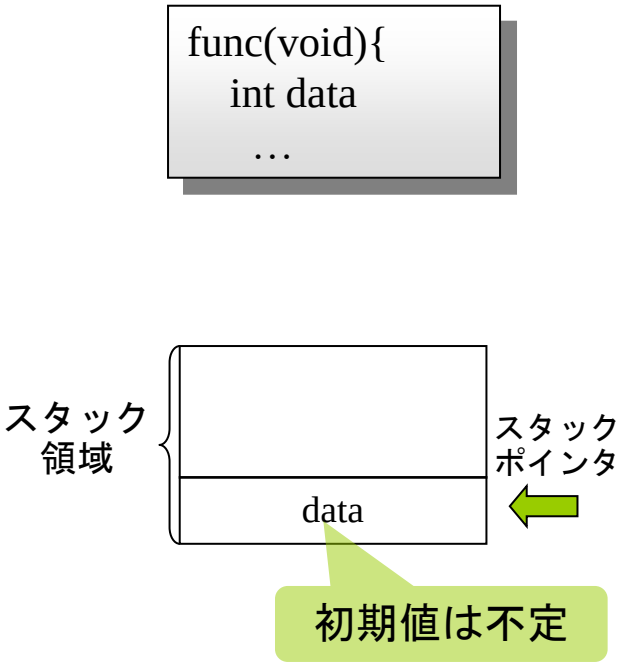
TOPPERS

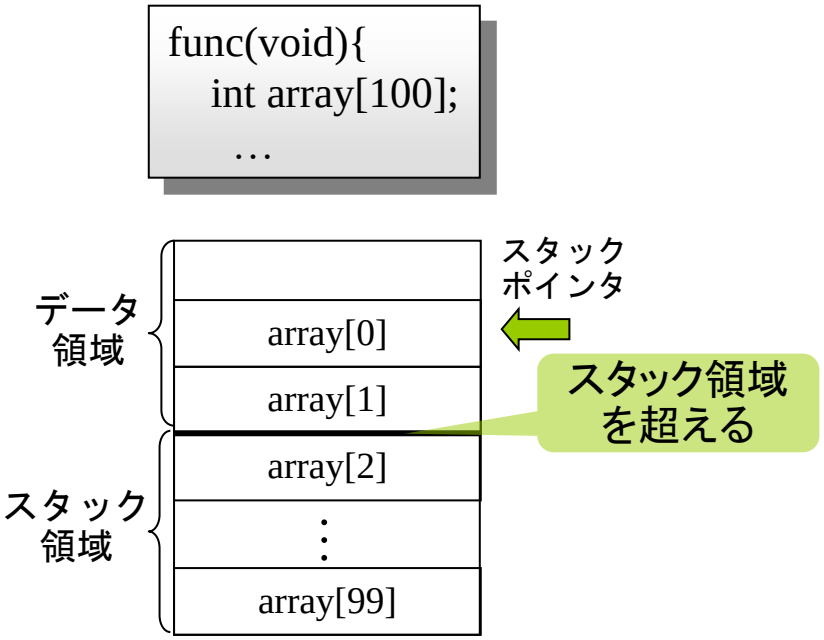
スタック：ローカル変数

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





TOPPERS

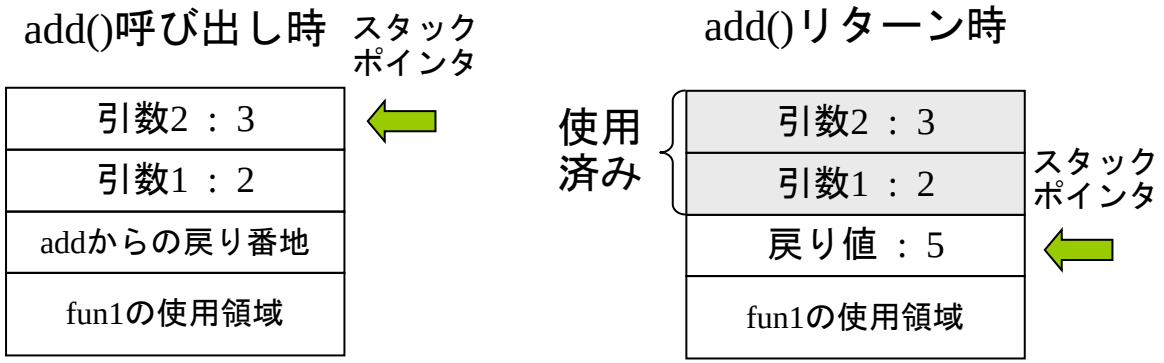
スタック：関数呼び出し

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;
}
```



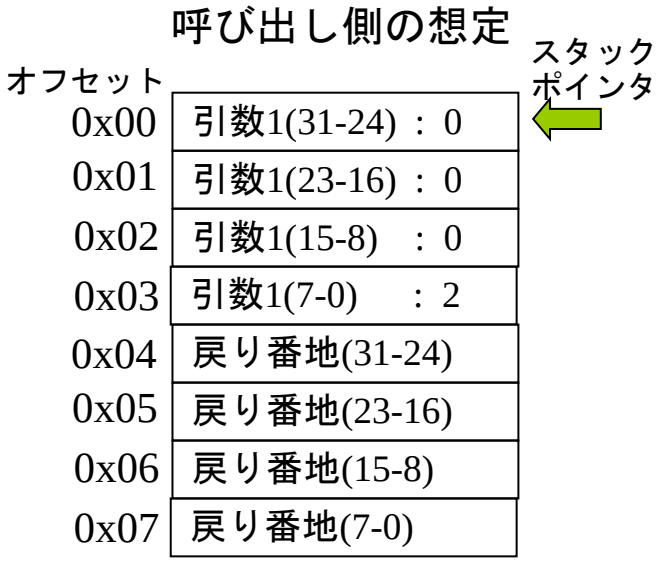
TOPPERS

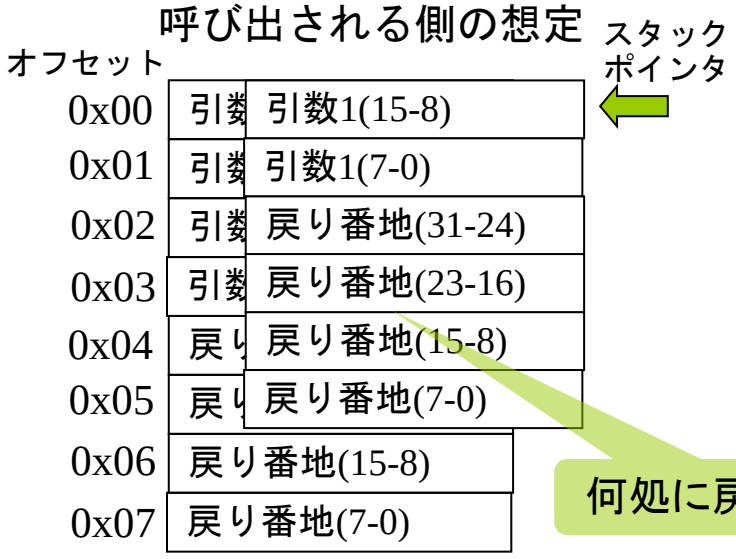
スタック：関数のプロトタイプ宣言

- 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;
}
```





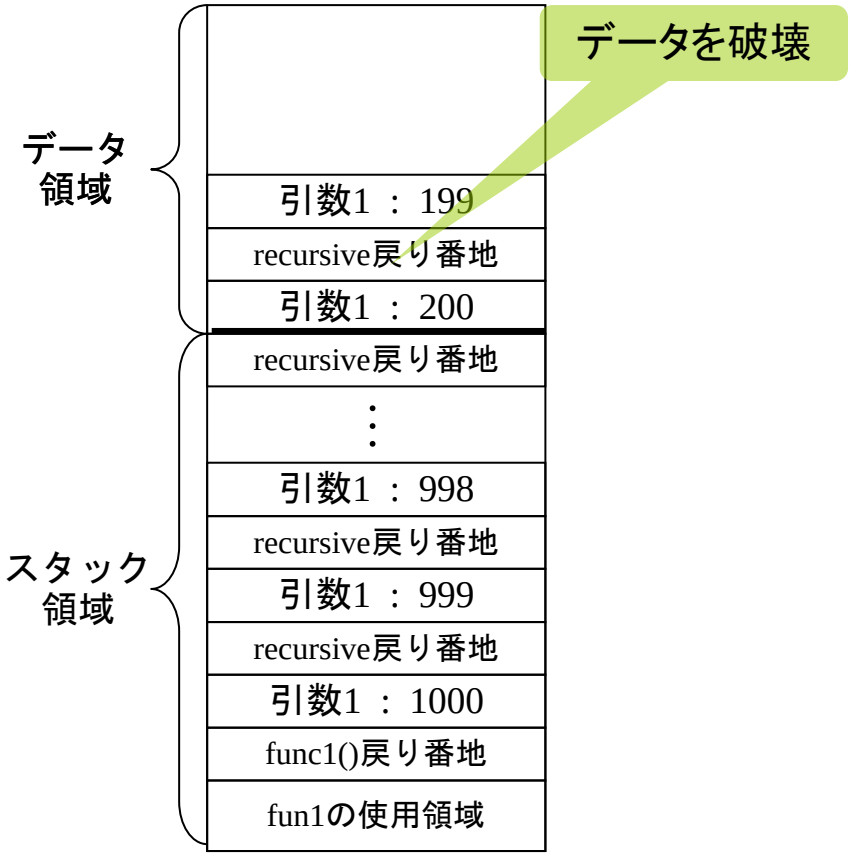
TOPPERS

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

```
func1(int n){
  …
  recursive(1000);
  …
```

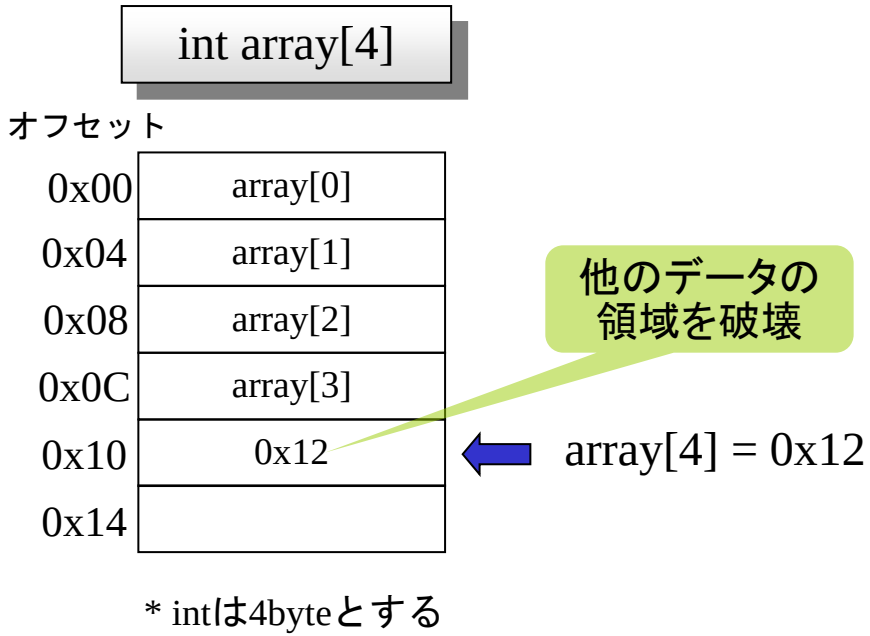
```
int
recursive(int n){
  if(n == 0)
      return 0;
  return recursive(n-1) + n;
}
```



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



* intは4byteとする

構造体

プロセッサのアライメントに従い配置される

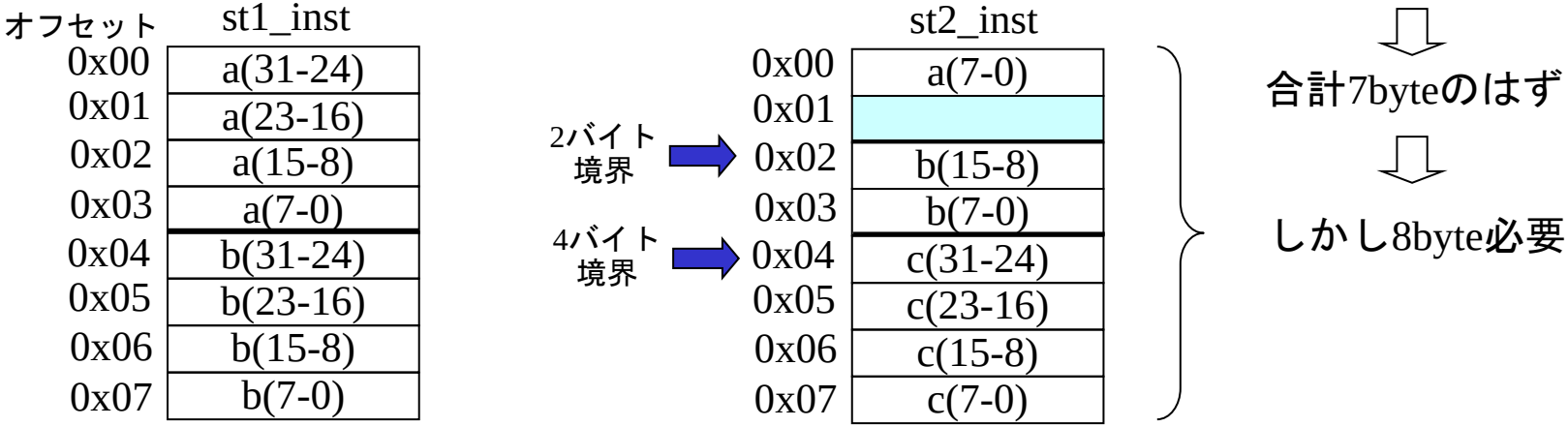
アライメント

- 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

```
struct{
    int a;
    int b;
} st1_inst;
```

```
struct{
    char  a;
    short b;
    int c;
}st2_inst;
```

charは1byte
shortは2byte
intは4byte
とする.



TOPPERS

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;
```

2004番地へのアクセス

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

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

キャスト

TOPPERS

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）0x10000 にあるアクセスサイズが2byteのデバイスレジスタの読込みと書き込み

読込み

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

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

書き込み

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

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

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

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

最適化

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

*pregの指し示す内容は変化しないとみなし一度だけチェックする

volatileの指定

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

*pregの指し示す内容は変化する可能性があると指示

TOPPERS

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;
 }
```

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
```

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("0001 0001") & 0x30("0011 0000") = 0x10("0001 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

  ~0x30("0011 0000") = 0xCF("1100 1111"),

  0x21("0010 0001") & 0xCF("1100 1111") = 0x01(0000 0001)

割り込みプログラミング

- 割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

- 出入り口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(){
   ….
```

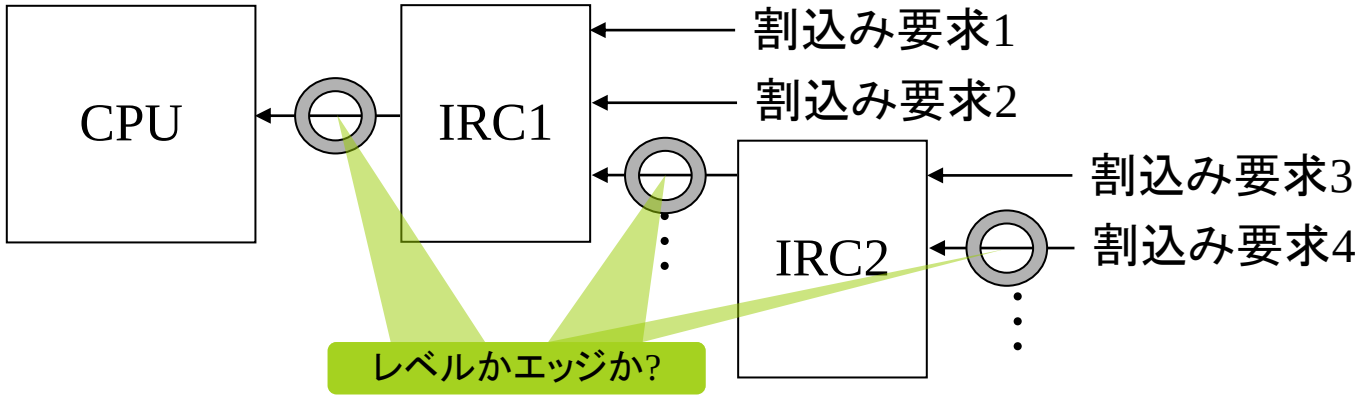
割込み要因のクリア

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



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 */
                  :
```


TOPPERS

多重割込み

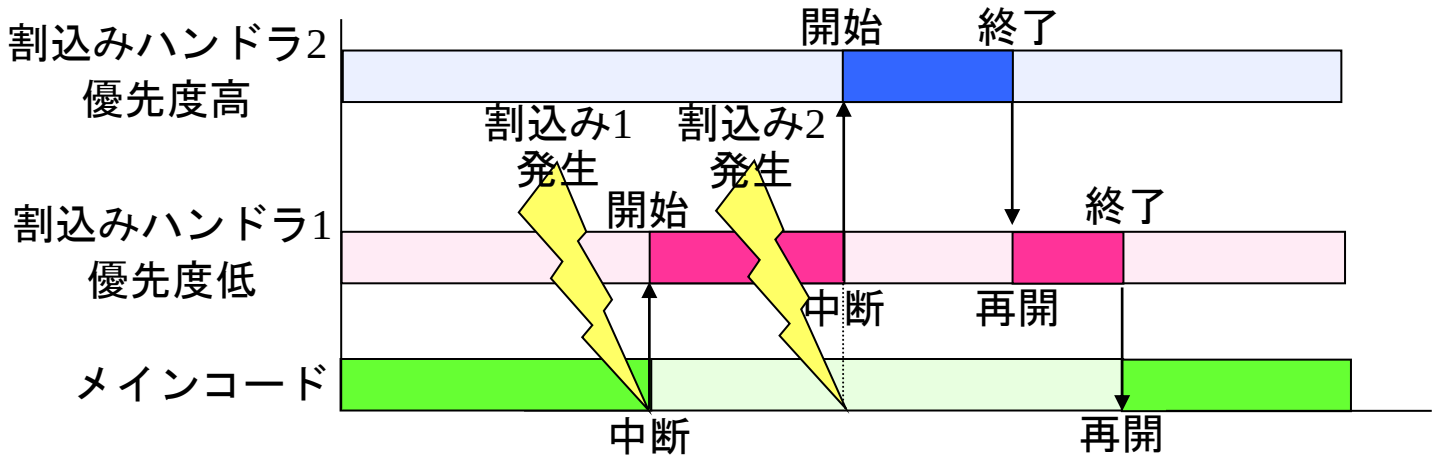
割込みを受け付けた直後

- 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



TOPPERS

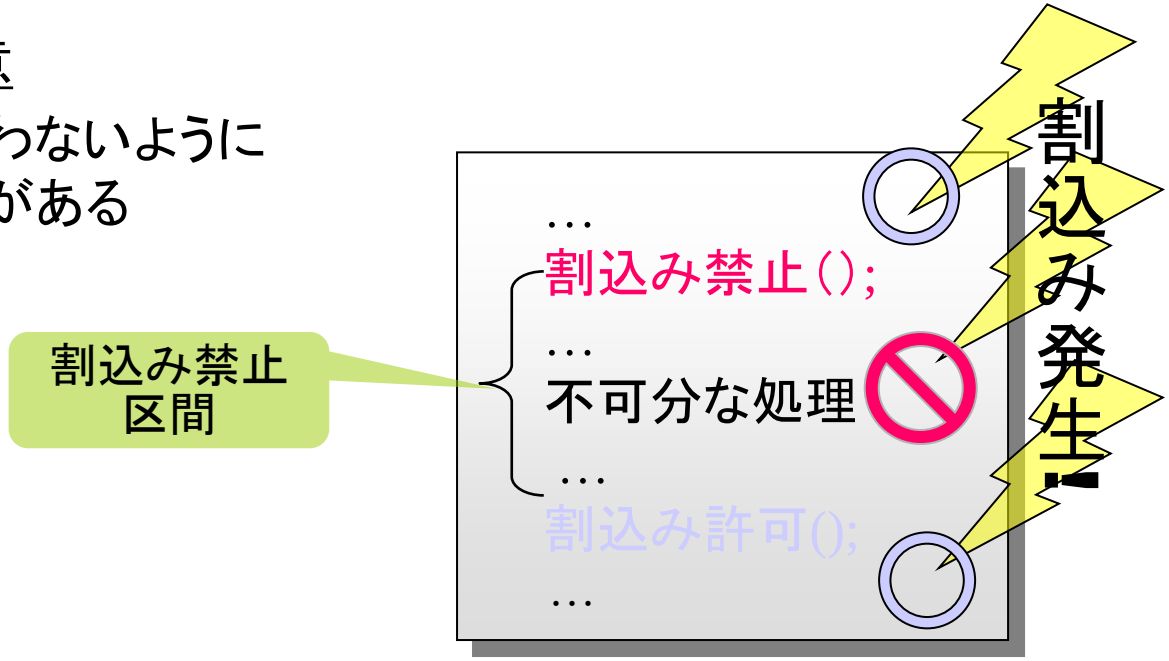
割込み禁止区間

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



プロセッサの割込みアーキテクチャの例：Cortex-M

- 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. ハンドラーモードに遷移する
  3. スタックをSP_main(MSP)に切り替える
  4. 汎用レジスタをスタックする
  5. スタックしたレジスタの内容を特殊レジスタの内容に置き換える
  6. ベクターの割込みハンドラにジャンプする

- 以上の処理はハードウェアで実行するため、プログラムを作成する必要はない

Cortex-M系の割込み

- Cortex-M系はNVICという割込みコントローラに変更され、CISCプロセッサと同等のベクタ型割込みとなりました
- Cortex-AやRは旧来のARMと同等の割込み方式です

例外タイプ	位置	優先度	説明
-	0	-	リセット時SPの値
リセット	1	-3(最高)	電源投入、ウォームリセット
マスク不能割込み	2	-2	割込み禁止ができない割込み
ハードフィールド	3	-1	
メモリ管理	4	構成可能	メモリ管理ユニットの例外
バスフォールト	5	構成可能	フェッチやメモリアクセスエラー
用法フォールト	6	構成可能	未定義命令実行等の例外
-	7-10	-	予約
SVCall	11	構成可能	ソフトウェア割込み
デバッグモニタ	12	構成可能	ホールド中でないときのデバッグモニタ
-	13	-	予約
PendSV	14	構成可能	
SysTick	15	構成可能	システムタイマの割込み
外部割込み	16以上	構成可能	NVIC経由の外部割込み

Cortex-Mのレジスタ構成

- 実行モードは特権モードと非特権（ユーザ）モードの2つだけ
- CPSRレジスタは3つのレジスタに分解された
  - APSR,IPSR,EPSR
- FIQ割込みはなくなり、割込み発生時、スタック上に使用するレジスタをハードウェアでPUSHする
- 割込み復帰はBX命令のエクセプションモードで行われる

ユーザー	特権モード
R0	R0
R1	R1
R2	R2
R3	R3
R4	R4
R5	R5
R6	R6
R7	R7
R8	R8
R9	R9
R10	R10
R11	R11
R12	R12
R13	R13_svc
R14	R14
PC	PC

ARSP	APSR
IPSR	IPSR
EPSR	EPSR

特権モード移行時R13（SP）のみ変更となる

TOPPERS

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

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

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

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

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

マイコンボードの確認

1. マイコンボード（STM32F401 Nocleo）
2. プロセッサ（STM32F401RET6）
3. 開発成果

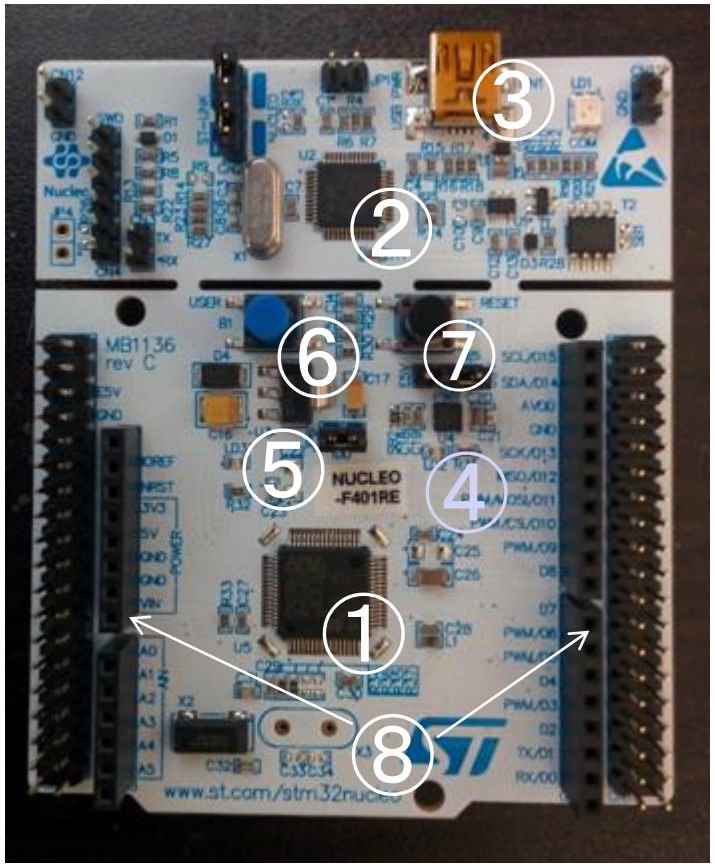
マイコンボードの概要

マイコンボードと搭載プロセッサついて解説する

- マイコンボード : STM32F401 Nucleo64
  - STMicroelectronics製
- プロセッサ
  - STMicroelectronics製 CPU:Cortex-M4F
  - 周波数84MHz
- 実装機能
  - ST-LINK用USB（仮想COM対応）、ST-LINK部は別ボード用のデバッガとして使用可能
  - Arduino UnoとST morphoコネクタ
  - ３つのLED、RESETとUSERスイッチ

マイコンボード : STM32F401 Nucleo64

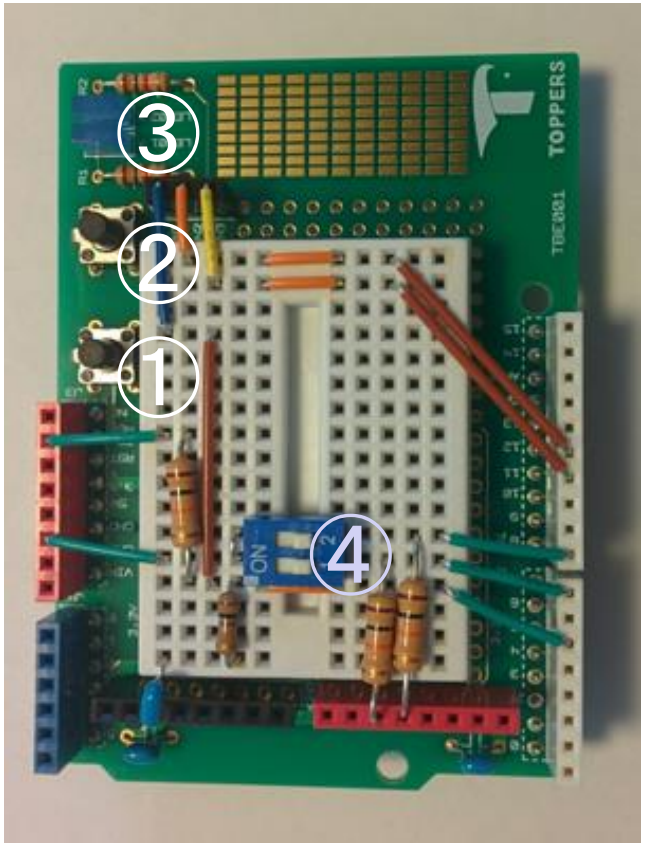
1. STM32F401
2. STM32F103
3. ST-LINK USB
4. USER LED
5. POWER LED
6. USERスイッチ
7. リセットスイッチ
8. Arduinoコネクタ



プロトタイピング・シールド

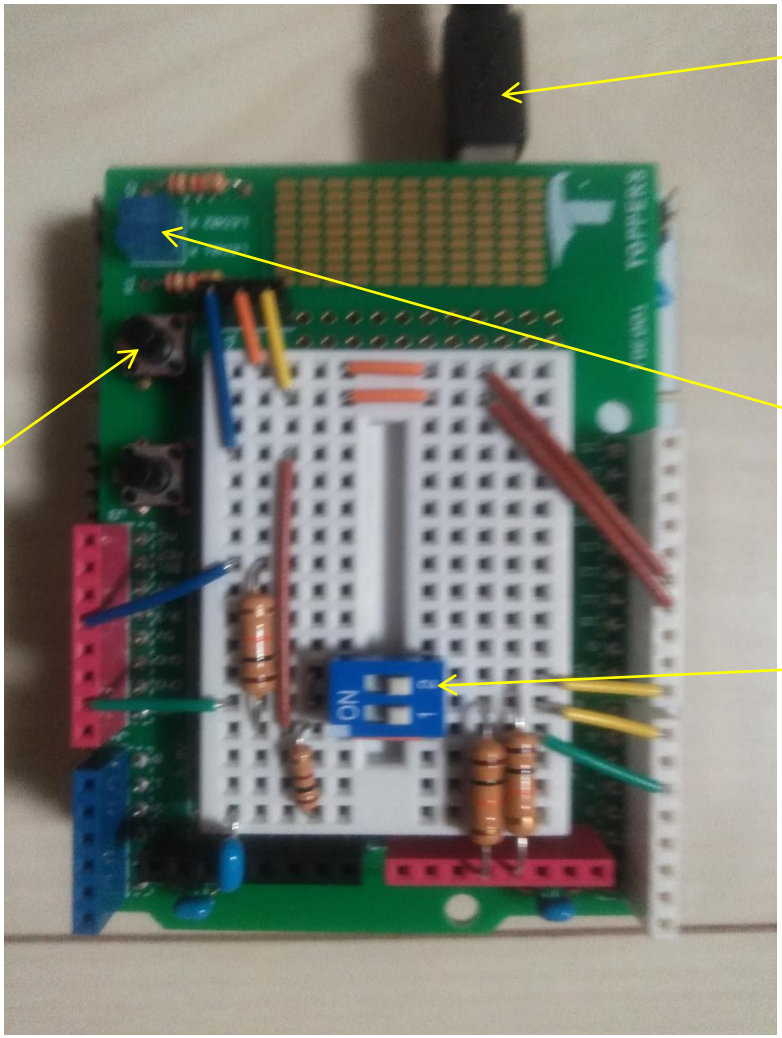
- Arduinoコネクタに対応したプロトタイピング・シールドを使用する

1. リセットスイッチ
2. USERスイッチ
3. LED1,2
4. DIPSW 1,2



実習で使うハードウェア

- プロトタイピング・シールドを接続した形で実習を行う



③ST-LINK USB

ROMモニタ表示
フラッシュROM
の書き込み
電源供給

④LED

点灯、点滅、消灯
の実習

⑤DIPSW

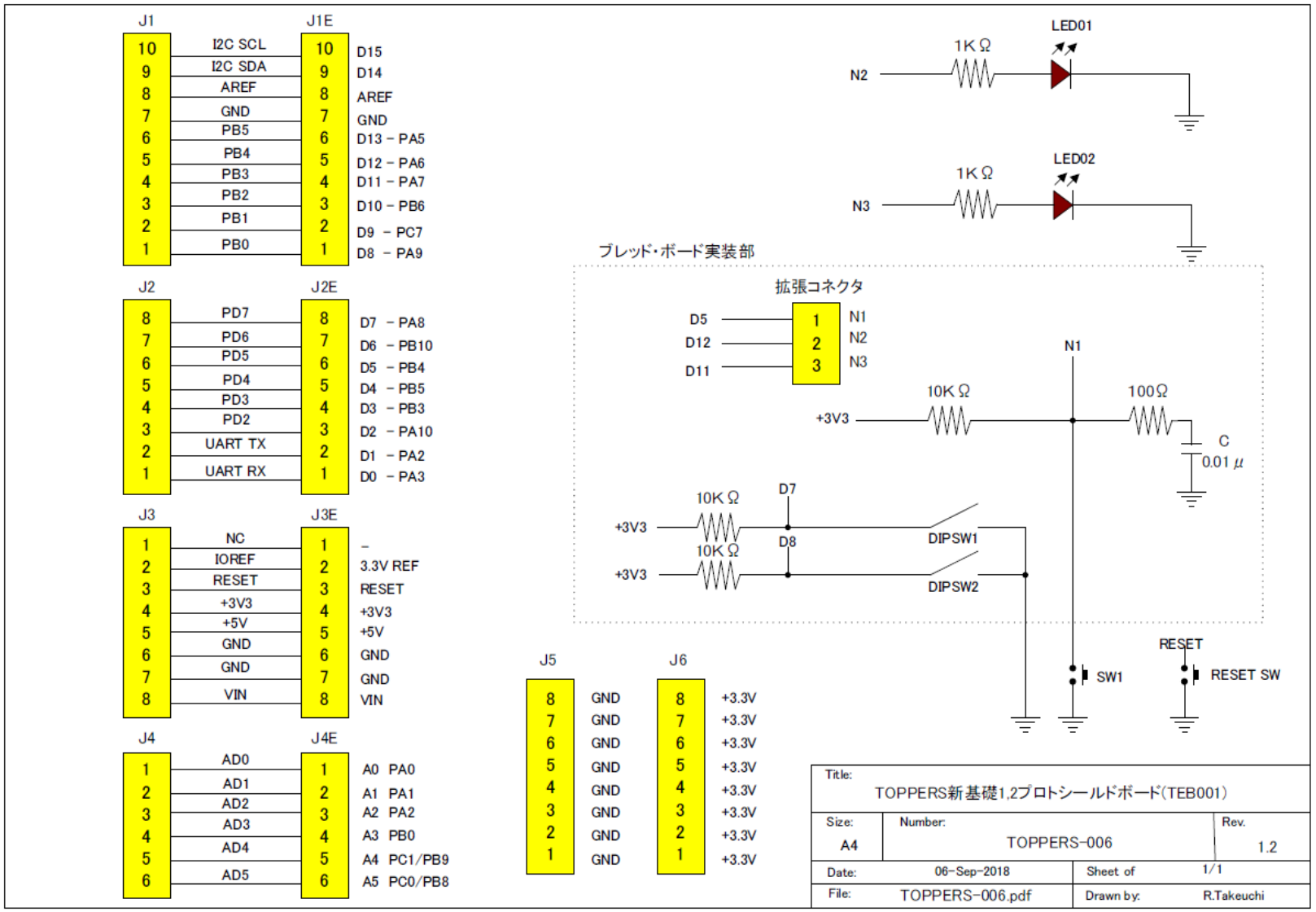
タスク間通信

②USERスイッチ

ポーリングと割込み
でスイッチの検知

TOPPERS

プロトタイピング・シールド回路図



TOPPERS

マイコンボードの確認

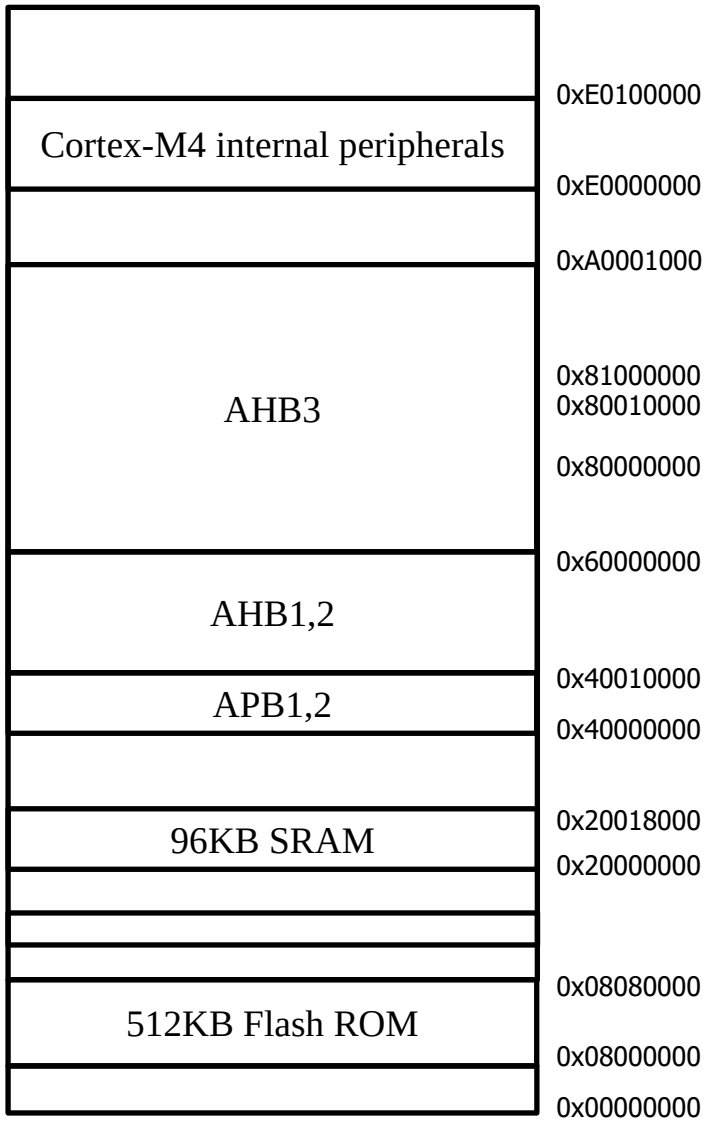
1. マイコンボード（STM32F401 Nucleo）
2. プロセッサ（ STM32F401RET6 ）
3. 開発成果

プロセッサ（STM32F401RET6）：概要

- STM製Cortex-M4Fをコアとしたシングルマイコンプロセッサ
  - F:ハードウェア浮動小数点
- ROM:512KB RAM:96KB
- 周辺回路：メモリマップドI/O方式
  - 入出力ポート　：50本
  - タイマ　：7 general purpose timer
  　：1 Advance control timer
  - シリアルI/O　：1usb OTG FS/3uart
  　SPI:3 SDIO
  - A/D　：16channel × 12bit ADC
  - その他　：2系統全二重I2S

STM32F401：メモリマップ

- 512KBのFlash ROM領域にプログラムを置き、96KBのSRAM領域をデータとしてプログラム開発を行う
- 今回、演習用にROMモニタを使用する
- Flash ROM領域にROMモニタプログラムを置き、SRAM中の0x20017000～0x20017FFFの4KBをROMモニタのデータ領域とし
- 0x2000000～0x20016FFFをダウンロード領域とする



STM32F401 Nucleoのリセットと実行モード

- 本実習の開発環境では、ユーザープログラムはスレッド特権モードで実行し、割込みハンドラは各ベクターから実行する
- リセットはベクターベースの先頭ベクターをMSPに設定し、次のベクターからリセット処理を実行する
- startup_stm32f4xx.S中にデフォルトの割込みハンドラ関数が.weak宣言で用意してあり、Default_Handlerにジャンプする
- 割込みハンドラ関数を、グローバル関数で別定義するとグローバル関数が正式なハンドラとしてリンクされる

```
   .section .text.Default_Handler,”ax”,%progbits
Default_Handler:
Infinite_Loop:
   b  Infinite_Loop
```


TOPPERS

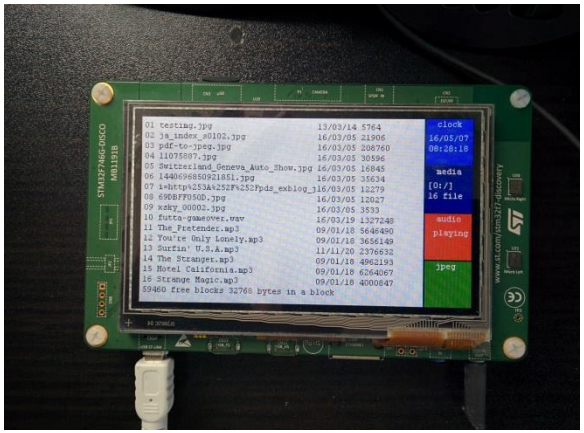
マイコンボードの確認

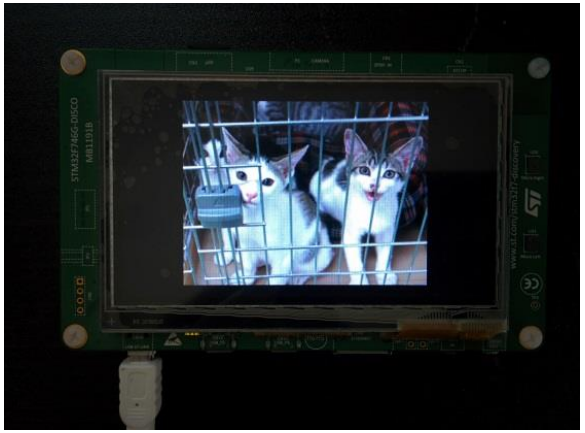
1. マイコンボード（STM32F401 Nucleo）
2. プロセッサ（ STM32F401RET6 ）
3. 開発成果

リファレンス・システム

- TOPPERS BASE PLATFORMを使ったアプリの構築例
  - SD-CARD/MSCプレイヤー

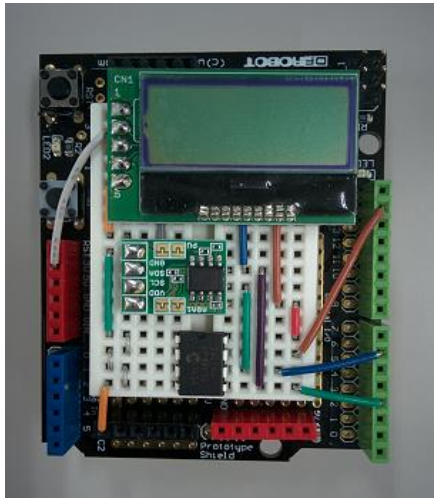
- STM32F746 Discoveryボードを使って、SDカードやUSBメモリのJPEGファイル表示、MP3ファイルの音楽演奏

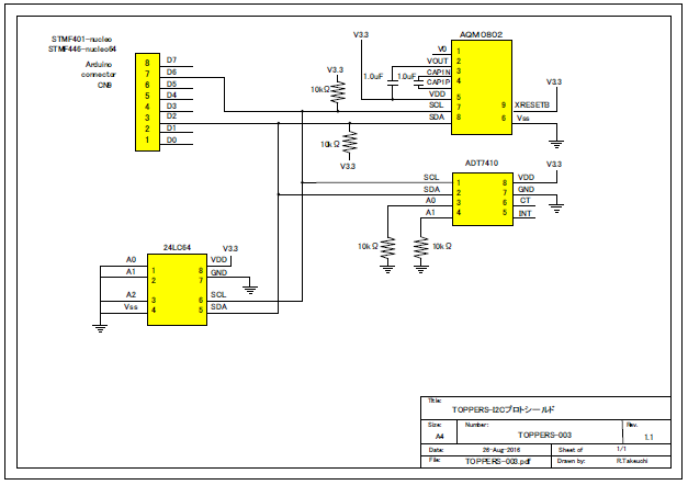




I2Cの実験ボード

- プロトタイピング・シールド上にI2Cに対応したデバイスを実装し、センサーやLCDの表示実験
  - ET2016のハードウェア・トラックで紹介
- サンプルプログラムでは、温度センサーから読み取った温度を1秒単位にLCDに表示を行う。EEPROMはreadプログラムを用意
- Prototyping ShieldがD14,D15ピンに対応していないため、拡張用にI2Cのインターフェイスをもつ、401nucleo-64,446 nucleo-64が実行対象であるが、D14,15ピンに対応しているシールドを使用すればArduinoコネクタをもつすべてのボードで対応可能





対応ボード
STM32F401 nucleo-64
STM32F446 nucleo-64

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. サンプルプログラムのビルドと実行
3. フラッシュメモリへの書き込み
4. ROMモニタの書き込み

ハードウェア環境の確認

- 使用するハードウェアを確認します
- 詳細は、開発環境編
- 開発環境の確認:「実習ハードウェアの構成」

ソフトウェア環境の構築

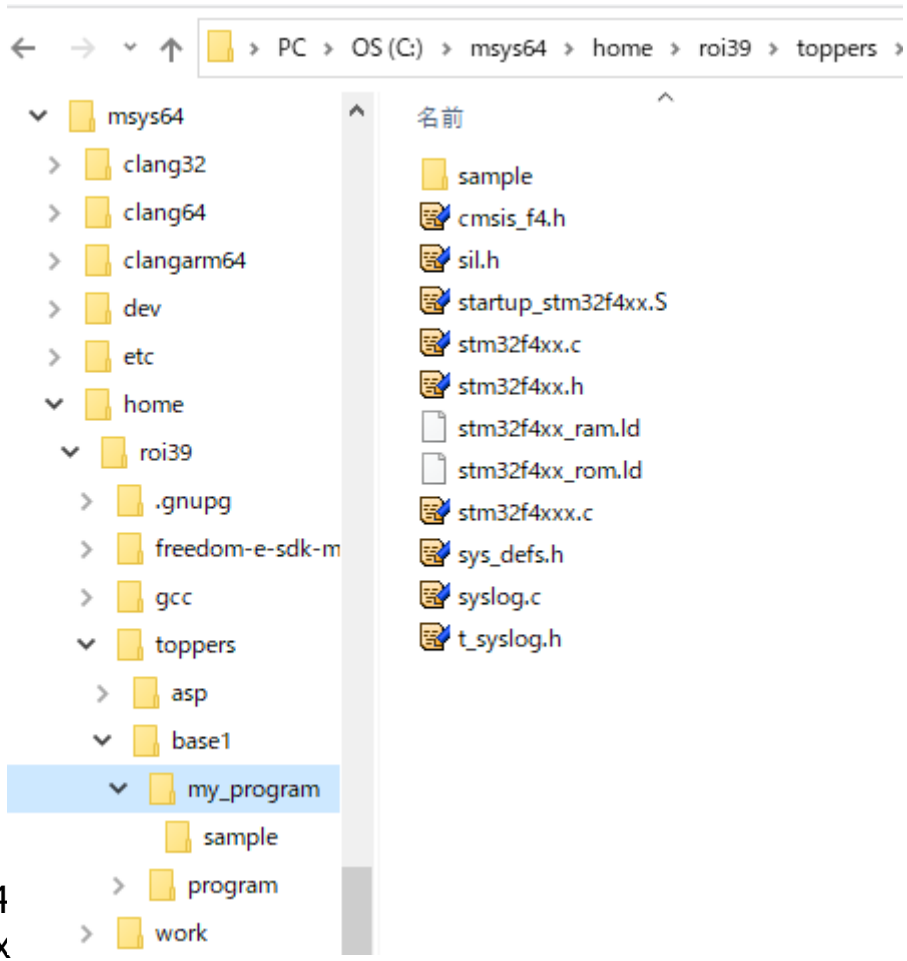
- パソコン上に開発環境がセットアップされていない場合は開発環境編、開発環境の確認「ソフトウェア環境の構築」を参照し、セットアップを行ってください
- その他のソフトウェア
  - エディタ（TeraPad, サクラエディタ，Xyzzy等）
  - ターミナルソフトウェア（TeraTermPro等）
  - PDFリーダー（Acrobat Reader等）

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. サンプルプログラムのビルドと実行
3. フラッシュメモリへの書き込み
4. ROMモニタの書き込み

SAMPLEプログラムのコピー

- msys2またはCygwin上のディレクトリにbase1/programをコピーする
- base1上にmy_programディレクトリを作成する
- program/sampleをmy_program上にコピーする：ここをビルド用のディレクトリにする
- program以下の以下のプログラムもmy_programにコピー



cmsis_f4.h/sil.h/startup_stm32f4xx.S/stm32f4xx.c/stm32f4xx.h/stm32f4xx_ram.ld/stm32f4xx_rom.ld/sys_defs.h/syslog.c/t_syslog.h

ビルド:ファイルの確認

- ビルド後、10ファイルが生成される
- ダウンロード、ROM書込み用にはSRECファイルを使用する

ファイル	ファイルの内容
sample.c	sampleプログラムC言語ソース
Makfile	GCC用MAKEファイル
sample.o	sample.cから生成された中間コードファイル
startup_stm32f4xx.o	スタートアップルーチンstartup_stm32f4xx.Sから生成された中間コードファイル
stm32f4xx.o	UART表示プログラムstm32f4xx.cから生成された中間コードファイル
syslog.o	sys_printf文、syslog文を実行するプログラム
sample.exe	sampleプログラムの実行形式、ELF形式
sample.srec	sampleプログラムの実行形式、SRECフォーマットダウンロード用
sample.syms	sampleプログラムシンボルファイル
sample.map	sampleプログラムMAPファイル
sample.lst	sampleプログラム逆アセンブラファイル

ROM実行設定に切り替え

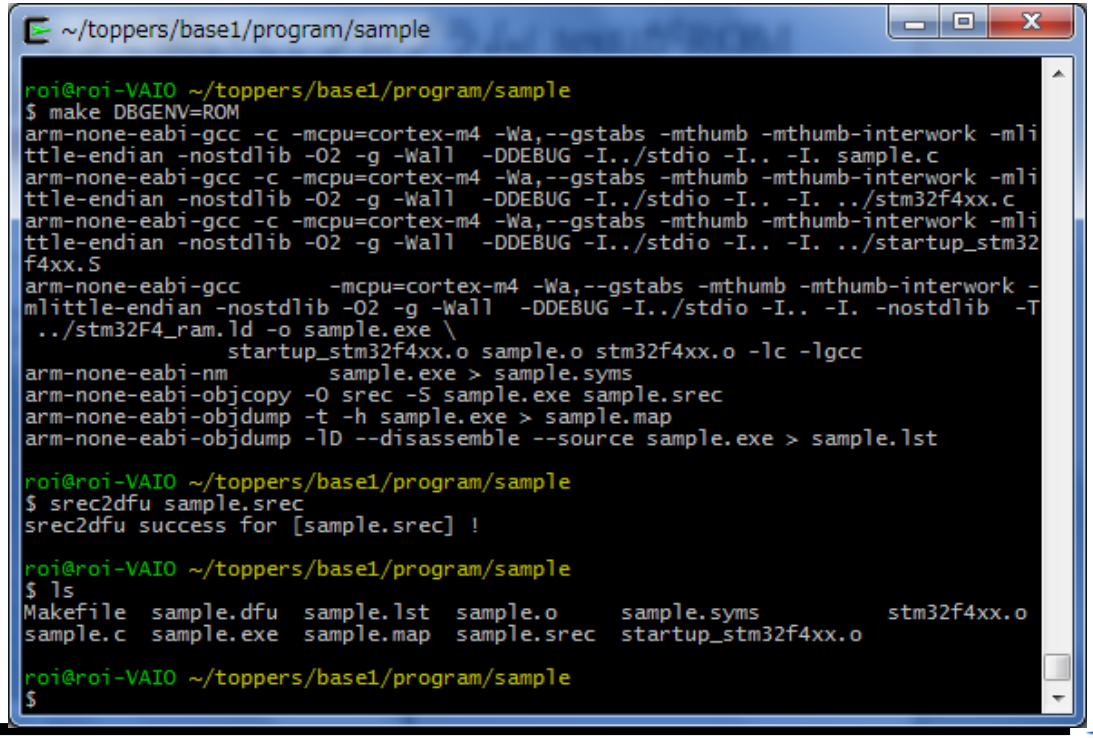
- ダウンロード実行は、RAM領域でプログラムを実行するため、電源を切れば毎回ダウンロードする必要がある
- プログラム領域をFLASH ROMに切り替え、ST-LINK Utiltityで書き込みを行えば、電源オンでプログラムを実行することができる
- ROM実行モードにするためには、Makefileの設定を変えてビルドする必要がある
  - Makefile3行目のDBGENV = ROMを追加
- make DBGENV=ROMでも、ROM化可能

```
2 ROOT_PATH = ..
3 DBGENV = ROM
4
```


TOPPERS

ROMモードのビルド

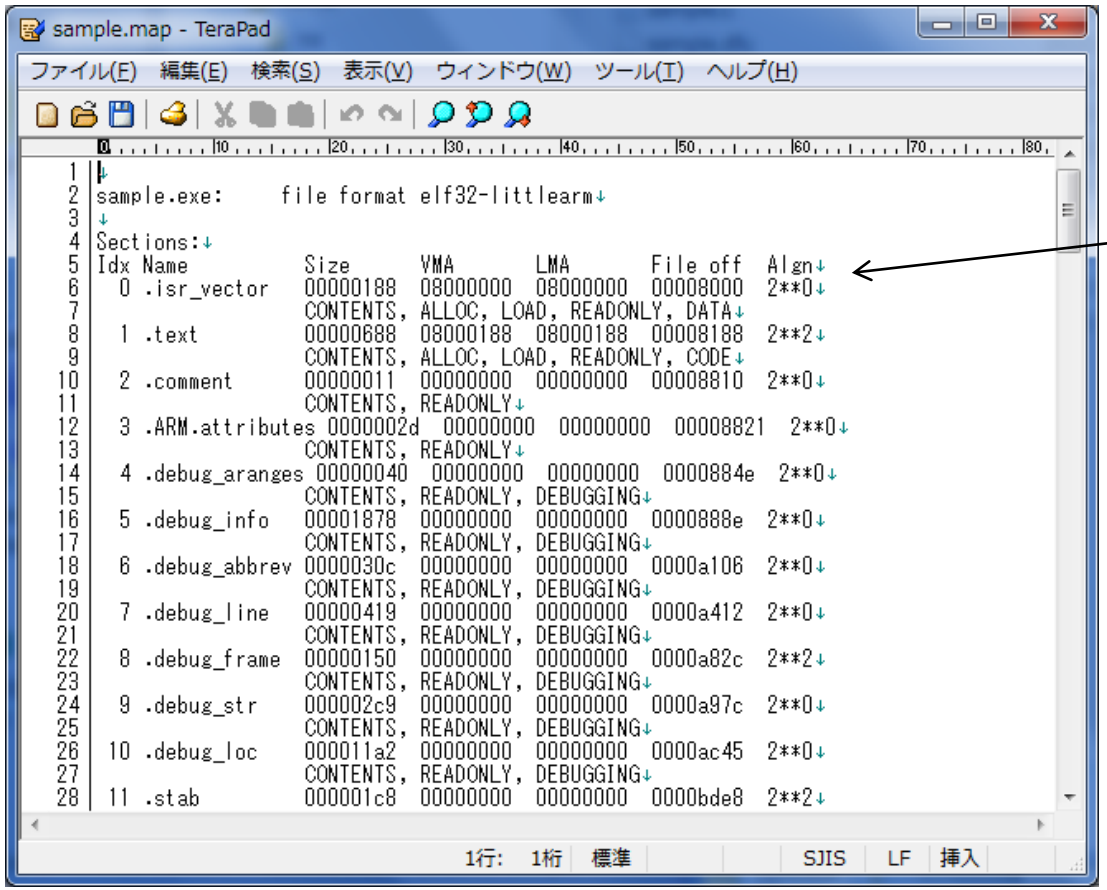
- make clean(enter)
- make DBGENV=ROM(enter) でROM化可能なSRECファイルをビルドできる



TOPPERS

ROM化の確認

- sample.mapをエデッタで開いて、プログラム(.text)がROM領域(0x800000)となっていることを確認する



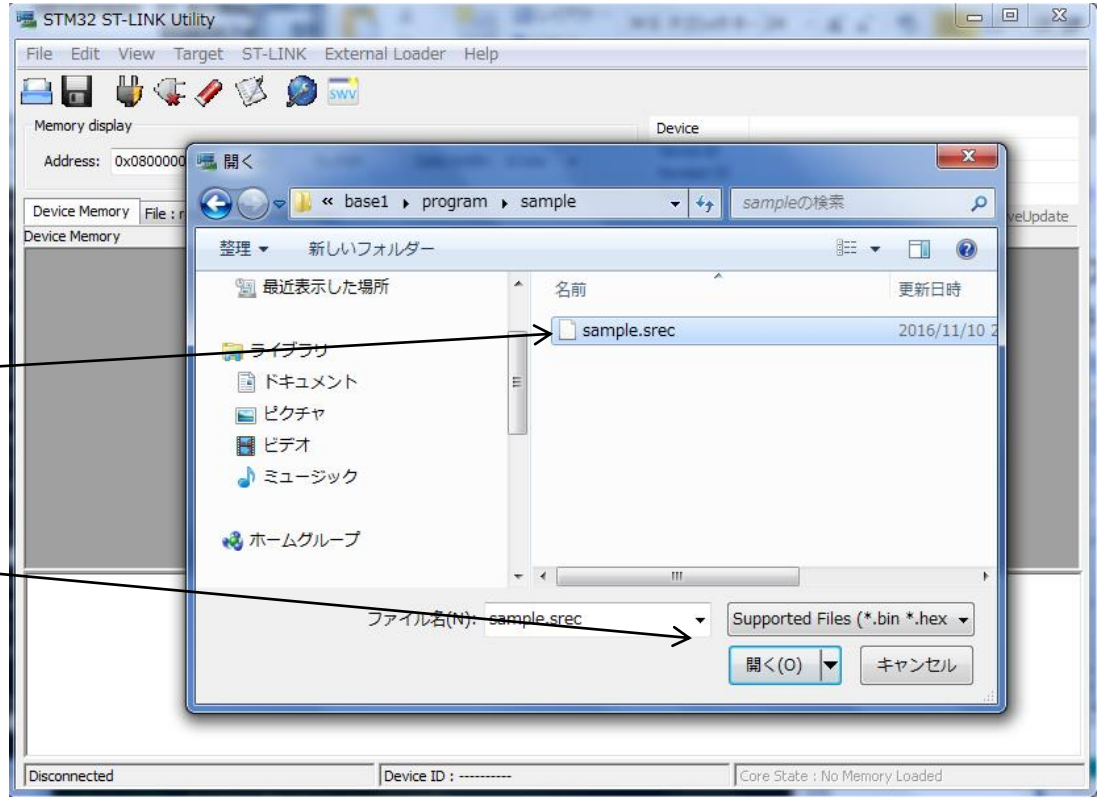
.vector/.text領域がFlash ROM領域となっている

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
3. サンプルプログラムのビルドと実行
3. フラッシュメモリへの書き込み
4. ROMモニタの書き込み

sampleをFlash ROMに書き込む

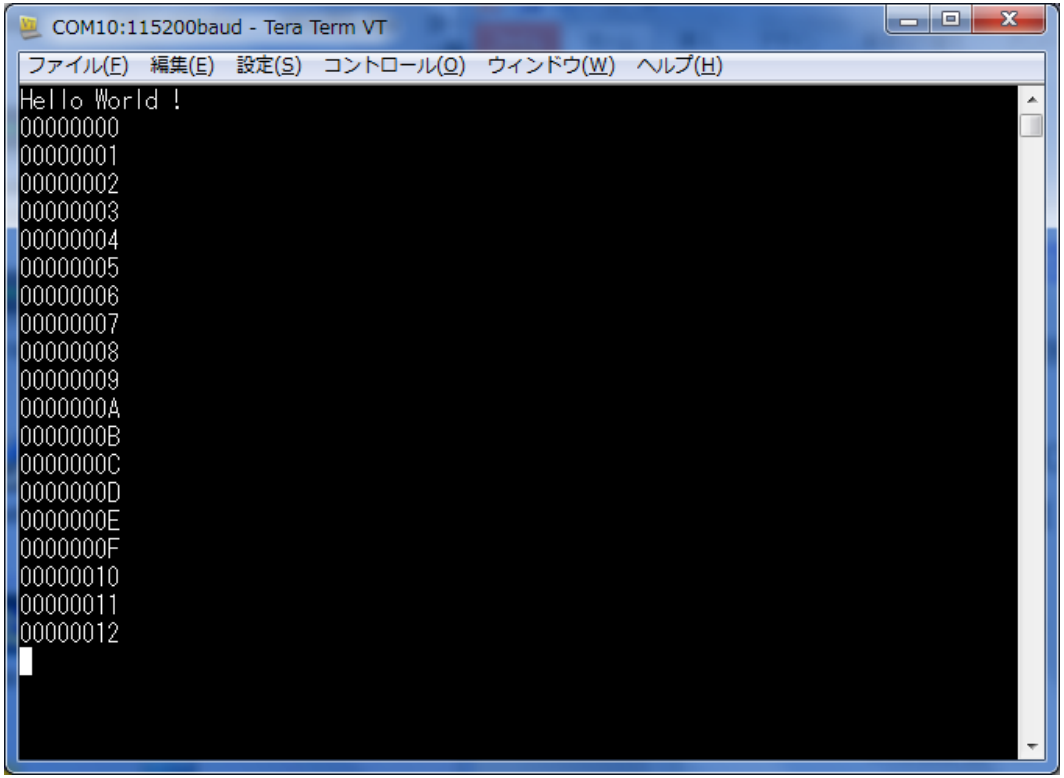
- ST-LINK Utilityのファイルで書込みファイルを選択
- program/sample/sample.srecを選択する
- Target→Program…で書き込む



TOPPERS

sampleをROM実行する

- TeraTermをCOMポートに対応して立ち上げる
  - 115200baud
  - 8bit
  - Non parity
  - Stop bit1
  - None flow
- 書込み後、
  リブート



開発環境の確認

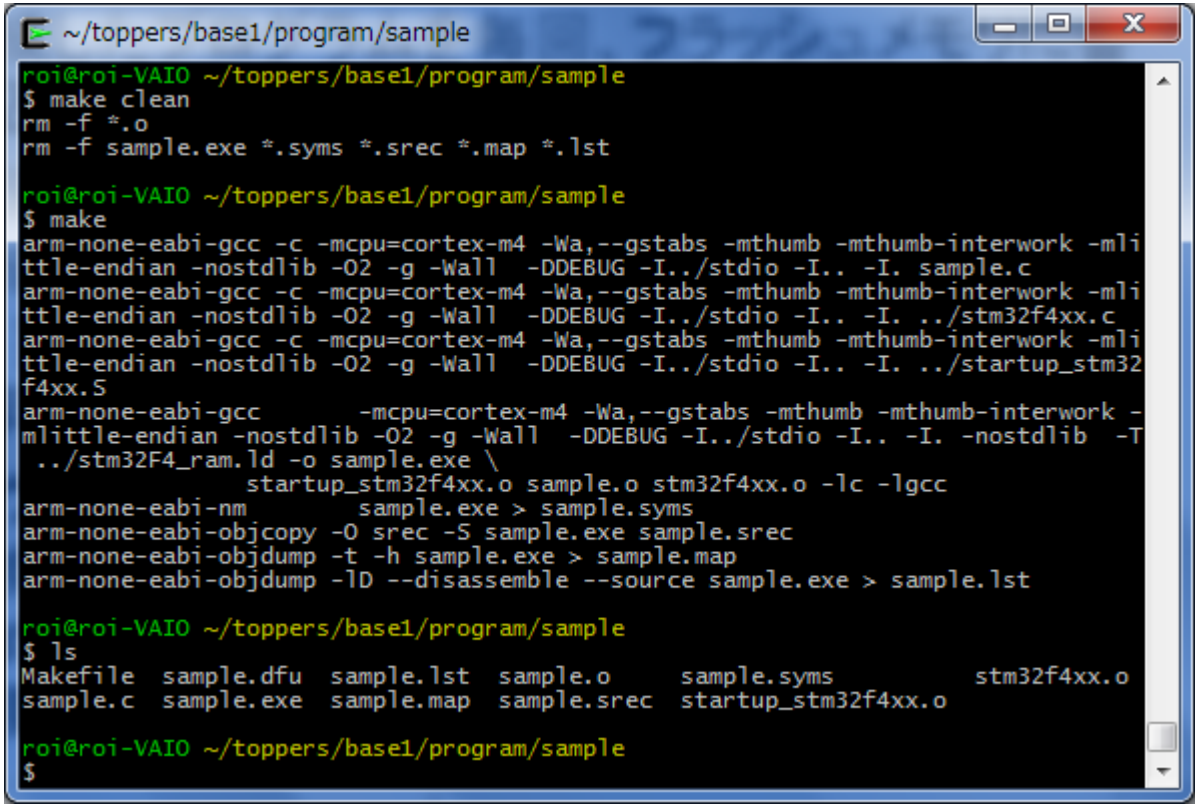
1. ハードウェア・ソフトウェア環境の構築
2. サンプルプログラムのビルドと実行
3. フラッシュメモリへの書き込み
4. ROMモニタの書き込み

実習ではROMモニタを使用

- 作成したプログラムを毎回、フラッシュメモリに書き込んで実行確認を行うのは大変です
- 実習では、ROMモニタをボード上で実行させ、作成したプログラムをシリアル通信でダウンロード、実行すればプログラムのデバッグを簡略化できます
- ダウンロードするプログラムはRAM上で実行します、RAM上で実行するプログラムを作成するために、RAM指定でプログラムを再ビルドする必要があります

SAMPLEのビルド

- make clean(enter)でROM実行プログラムをクリアする
- make(enter)　にてプログラムをビルドする
- ビルドの手順はMakefileに記載、実行の記述はsample.cに記載している



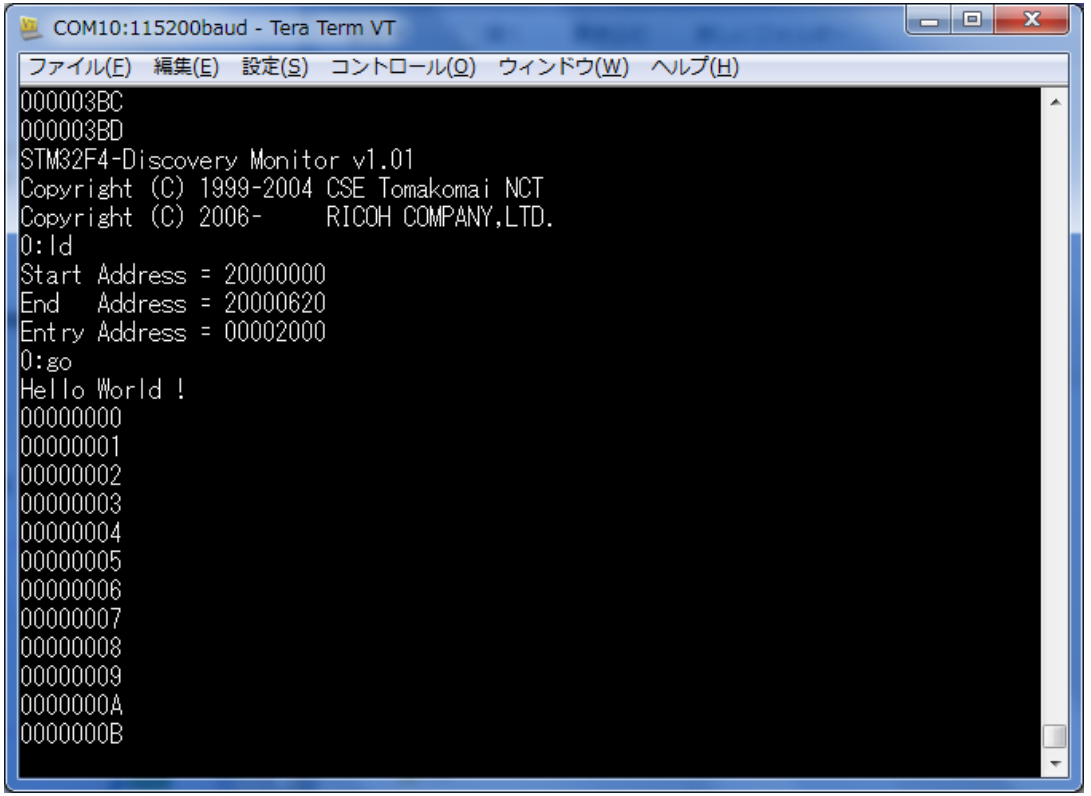
ROMモニタの書き込みと実行

- フラッシュメモリにROMモニタを書き込む
- 書き込みの手順は開発環境編、ROMモニタ「ROMモニタを書き込む」を参照
- ROMモニタを起動後、ビルドしたプログラムのダウンロードと実行を行う
- ダウンロード実行の手順は開発環境編、ROMモニタ「ダウンロードと実行」を参照

実行:ダウンロード開始、実行開始

- ダイアログのファイル転送をクリックするとダウンロードを開始する
- ダウンロード終了後、モニタのgo(enter) でプログラムを実行する

Hello World！
と0.5秒ごとにカウントを表示する



TOPPERS

デバッグ機能の確認

- sample.cを確認する
- _main関数には以下の2つの関数でUART表示を行う
  - sys_printf　　引数の文字列と引数を表示する
  - sys_puthex　　引数を16進表示する:マクロ

```
void
_main(void){
        int count = 0;

        sys_printf("Hello World !¥n");
        for (;;) {
                sys_printf(“%08x¥n“, count);
                count++;
                busy_wait();
        }
}
```

デバッグ機能の確認

- Makefileを確認する
- $(CORE)はstm32f4xxを指す
- 59行目、62行目のCOBJSにsyslog.oを追加することにより、sys_printfとsys_puthex関数を使用できるようになる
- 後のプログラム実習で、この機能を使えばデータやIOマップの表示確認を行える

```
55 AOBJS = startup_$(CORE)
56
57 ($(DBGENV),ROM
58 CDEFS := $(CDEFS) –DROM_EXEC=1
59 COBJS := $(COBJS) $(CORE).o syslog.o
60 else
61 ifeq ($(DEBUG),1)
62 COBJS := $(COBJS) $(CORE).o syslog.o
63 endif
64 endif
```


TOPPERS

オプション指定によるUART表示

- Makefileの61行目の設定のように、DEBUGスイッチが1の場合、syslog.oを取り込むように設定してある
- make DEBUG=1 (enter)またはROM実行でsyslog.oがリンクされDEBUGコンパイルスイッチ内の表示が有効となる
- make (enter) ではsyslog.oはリンクされず、表示も行わない
- ログ表示を行わない場合は、メモリの節約になる

```
66 ifeq ($(DEBUG),1)
67 CDEFS *= $(CDEFS) -DDEBUG
68 endif
```

オプション指定によるUART表示

- 共通のインクルードファイルsys_defs.hに、uart出力のプロトタイプ宣言ある
- DEBUGが0の場合、uart関数は関数ごと未定義となる

```
#if DEBUG == 0
#define sys_puthex(val)
#define sys_printf(format, …)
#else
#define sys_puthex(val) sys_printf(“%08x”, (val))
extern void sys_printf(const char *format, …);
#endif
```

- ２日目以降の演習で以下のように、変数の表示が可能となる

```
led_out_bdigit(++led_count);
sys_printf(“%d¥n” led_count);
```


TOPPERS

ROMモニタを使った実習

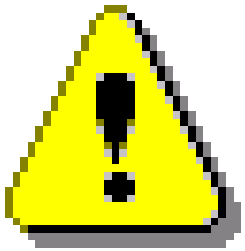
いろいろな開発環境

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

デバッグ時の注意

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

  ⇒注意深い、ソフトウェア開発が必要



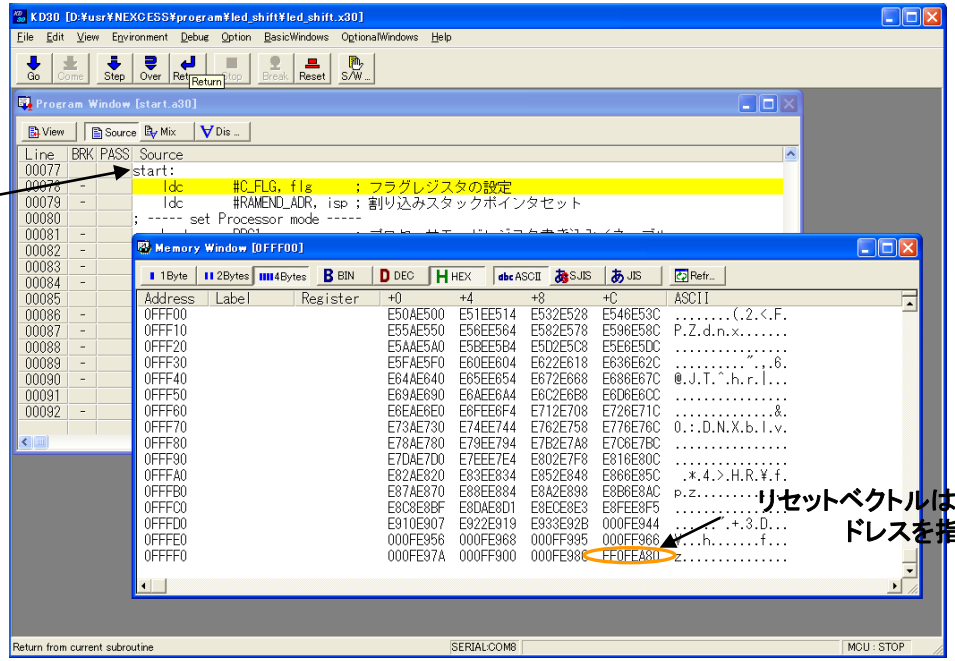
TOPPERS

CPU、半導体メーカのツールを使用する

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

ユーザプログラムのリセットアドレスは0xE00000

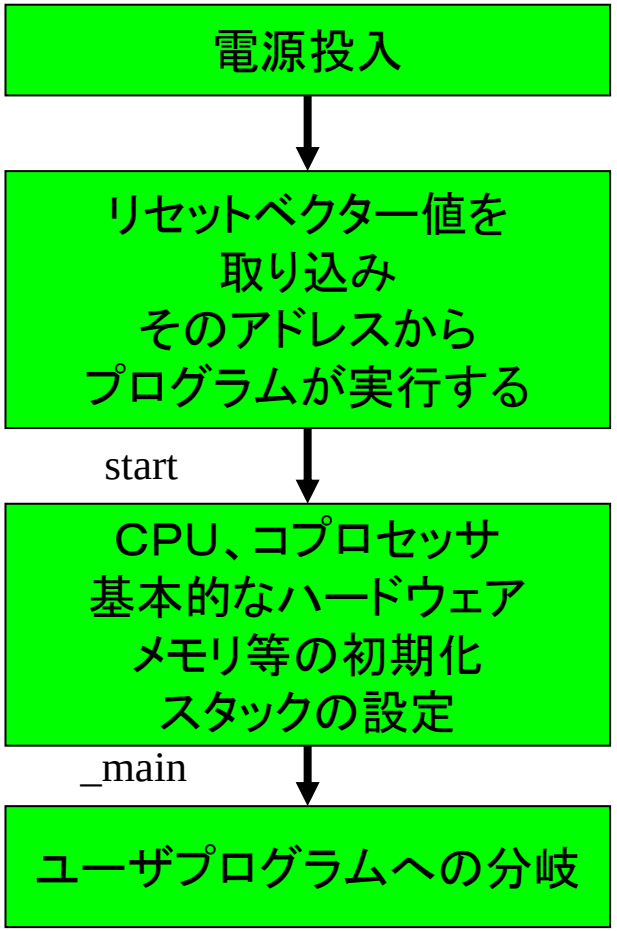
Goボタンを押すと、コマンドをモニタが受け取り、ユーザプログラムのリセットアドレスから実行させる



TOPPERS

スタートアップルーチンとは

- CPUはリセットベクトルから起動し、ハードウェアの初期化後、はじめてmain()が実行
  - 電源オンまたはリセットからmain()が実行するまでのプログラムをスタートアップルーチンと呼ぶ
- GCCなどの開発環境を使用する場合、自作する必要がある



セクションデータの再配置

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

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

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

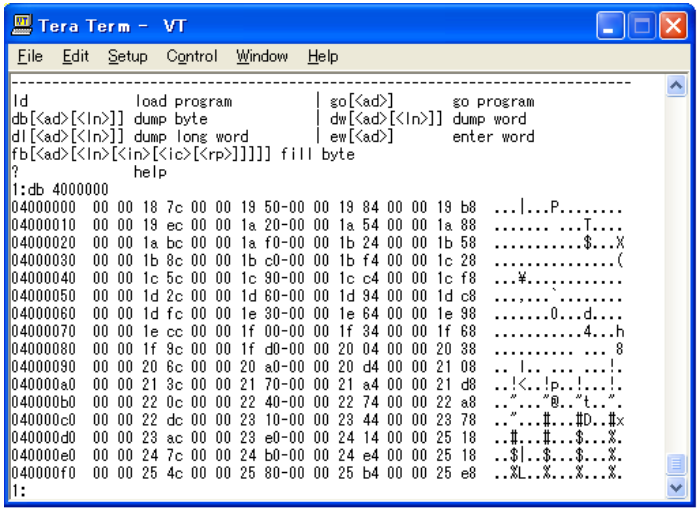
スタートアップルーチンの例

- startup_stm32f4xx.SがSTM32F401 Nucleo用のスタートアップルーチン
- 割込みを使用する場合、割込み関数をベクタテーブルにセットする必要がある
- startup_stm32f4xx.S開いて、内容を確認してください
- 初期化が終了した時点での_main関数の実行を始める
  - 以下のアセンブラ命令

```
start_0:
        bl      _main
        bx      lr
```

ROMモニタから起動する

- ROMモニタとはエバレーション・ボードのROMに直接書き込まれ、プログラムのダウンロードや実行、データ参照を行うプログラム
  - RAM領域が大きなシステムに多く、小規模な組込みシステムには少ない
  - クロス環境で作成したプログラムをダウンロード実行してデバッグを行い
- 専用メーカから購入したり、オープンソースを利用する場合もある



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

TOPPERS

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）
- Cortex-M4ボード用の改変
  - 竹内良輔（(株)リコー）