

TOPPERS基礎実装セミナー

(Raspberry Pi Pico(W)/Pico2(W)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. 著作権に関する表記

＜TOPPERS基礎実装セミナー(Raspberry Pi Pico(W)/Pico2(W)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) 2007-2026 by 竹内良輔 教育WG

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	1.8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	0.7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(ベアメタル開発)、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。

小規模(ベアメタル開発の重要性)

- 大ヒット組込みCPU: ARM7以降ARMコアは3タイプとなった
 - Cortex-A,R,M (Application/Real-time/Microcontroller)
- ところが、小中組込み機器はCortex-Mが主流となってしまった
- Cortex-Mはアセンブラ用に割込みベクタとC言語の初期化を記載すると、その他は全てC言語で記載可能という特徴を持つ
 - PICマイコンのようにRTOSを使用しなくてもシステム構築可能
- RTOSを使う場合とベアメタルでは、どんな差があるか？
 - 下表はRTOS上で、パソコンでC言語を使う環境(組込みソフトウェアプラットフォーム)と、ベアメタル開発で同じものを作った場合のサイズ差
- 開発が難しくてもロットが多いと、利益率が全く違う

セクション	Asp版lcd_text4(バイト)	lcd_text6(バイト)	比率(%)
.text	63,888	38,440	60.17
.rodata	89,296	81,157	90.88
.data	4,208	3,084	73.29
.bss	27,064	5,280	19.51
ROM	157,392	122,681	77.95
RAM	31,272	8,368	26.76

基礎1.5(ベアメタル)、新基礎3
(組込みソフトウェア・プラットフォーム)で学習できます
・開発は新基礎3の方が楽
・PICマイコンのRAM容量の少なさは納得

メーカーの開発ツールを使わない開発環境

- 組み込み開発はターミナルとコンパイラと書き込みツールがあれば可能です、どの開発ツールもバックグラウンドで、これらを上手く呼び出しているだけです
- 代表的なターミナルでの開発環境
 - nRF5_SDK: BLEシェアNo.1のNordic Semiconductor ASAの開発環境、Makefile中のコンパイラ設定をターミナル環境に合わせるだけでターミナル環境で開発可能
 - STM32Cube: STMicoroelectronics社の開発環境、昔はGCCに対応していましたが、現在はSW4STM32に置き換わっていますが、Makefileさえ書ければソースを、そのまま使用してターミナル環境で開発可能
 - Pico-SDK: Raspberry Pi Picoの開発環境、Linuxターミナルの開発環境です、LinuxでCMake用の生成を行った後、Windowsのターミナル環境にコピーしてMakefileを作ればWindowsの開発環境で開発可能、自分でシステム構築したい場合はMakefileは自作した方が効率が良い
 - Arduino-IDE: 外部提供のボードの場合、必要なソースはパッケージで構築される、これらにC++言語のmain関数とMakefileを作成すれば、ターミナル環境は作れる

通常のソフトウェア開発とベアメタル開発の違い

- 通常のソフトウェア開発はソフトウェア・プラットフォーム(Linux, Windows, Android等)のアーキテクチャが確立した環境で、全体設計からコーディングレベルを構築していく
 - トップダウン設計(開発効率が良い)
- ベアメタル環境では、CPU、ハードウェア、開発ツール等で、動作させないとアーキテクチャを決定できない要素が多くある開発となる
- デバッガの使用も面倒(オプティマイズ状態でソースコードデバック不可)、一行の記載ミスでハングアップが発生
 - ボトムアップ設計(開発効率は悪い)
 - 最初はジャングルの中を注意深く、一步ずつ進むような開発作業
 - 手間を惜しまない、手間をかけて問題解決すれば、必ず、自分のスキルアップとなる
 - 問題が発生した時のヒントは(英文の)FAQにあり
 - 見栄えの良い技術公開は導入のヒントのみ

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Raspberry Pi Pico(W)/Pico2(W)を用いて行う
- Cortex-MやRISC-V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-M0+/Cortex-M33/RISC-Vを使用したボードと開発に必要な機器の確認(技術説明はARMを中心)
- 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(コーディング力向上)

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

組み込みソフトウェア開発におけるハードウェア知識

組み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

- 組み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)

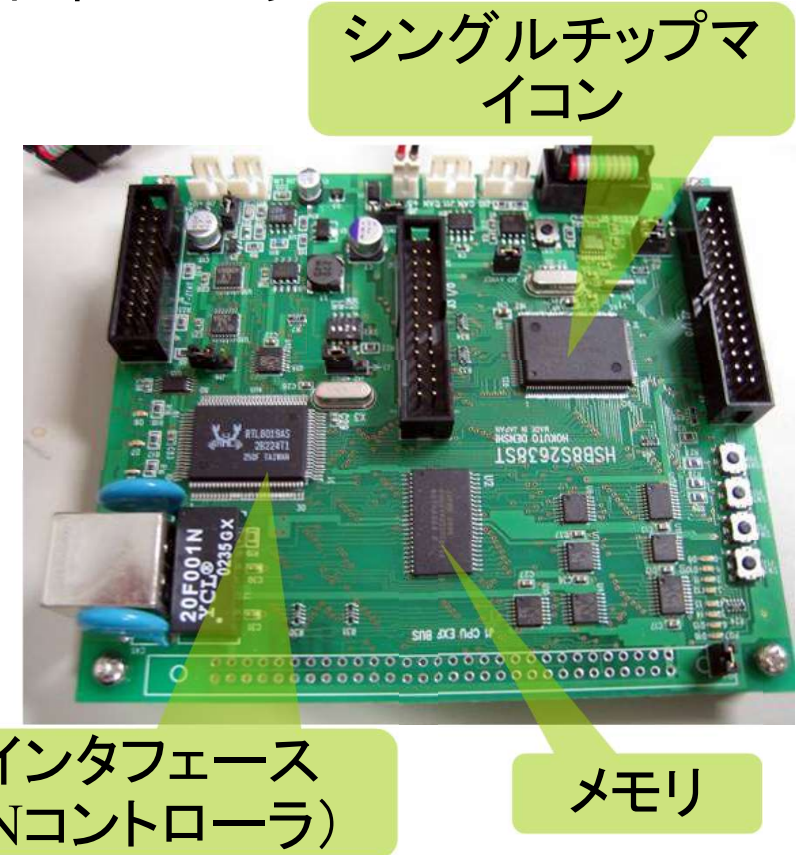
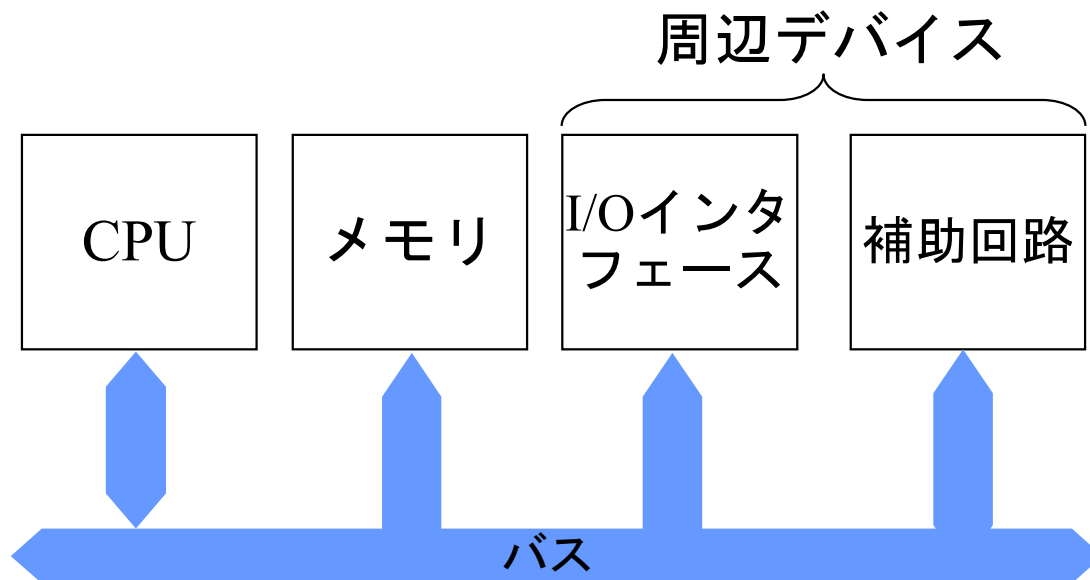
- ハードウェアが応用毎に最適化されて設計される
 ➡ 質の高いプログラムを開発するためには、ハードウェア
 アーキテクチャの理解が必要

組み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



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

組み込みシステム用プロセッサの多様性

パソコンの現状

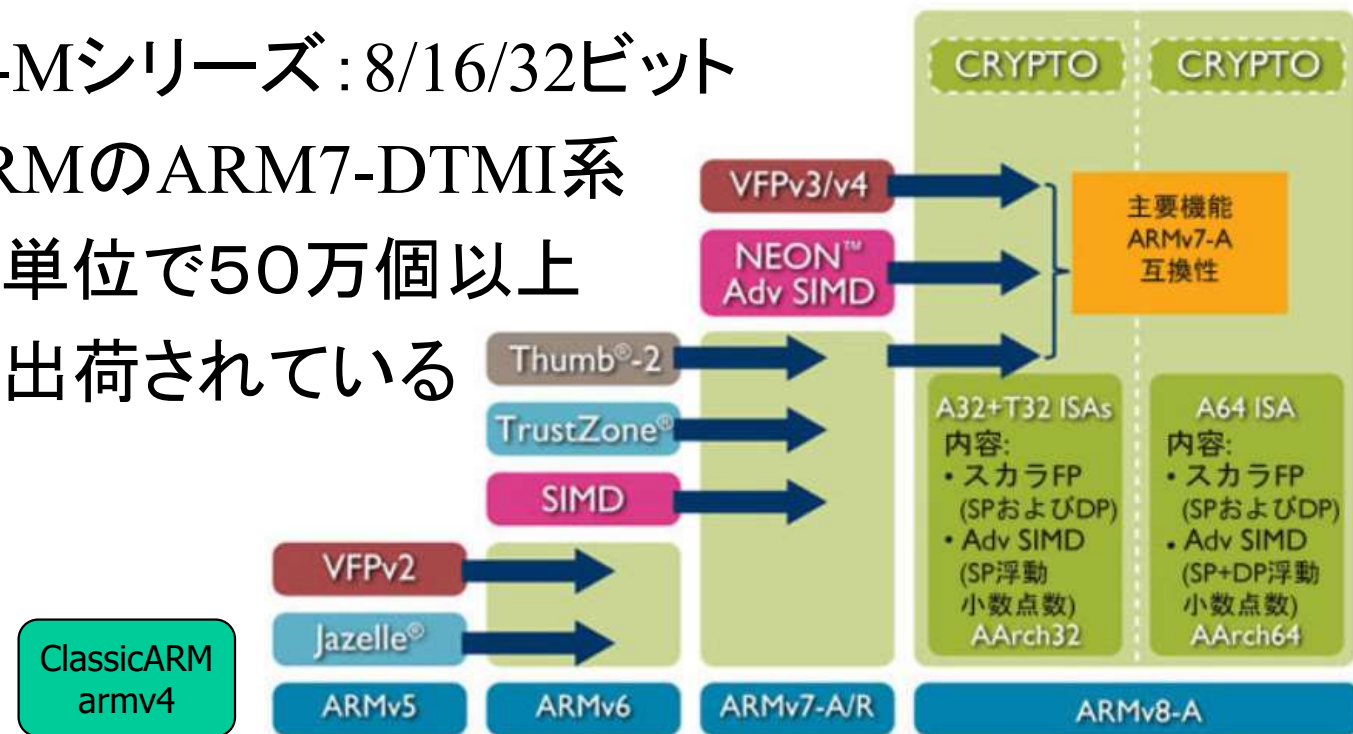
- IA-64(Core-Ix)が主に使われている
 - MACもPowerPCからIA-64へ、今は独自CPU

組み込みの現状

- 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, PIC系(やH8系, SH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

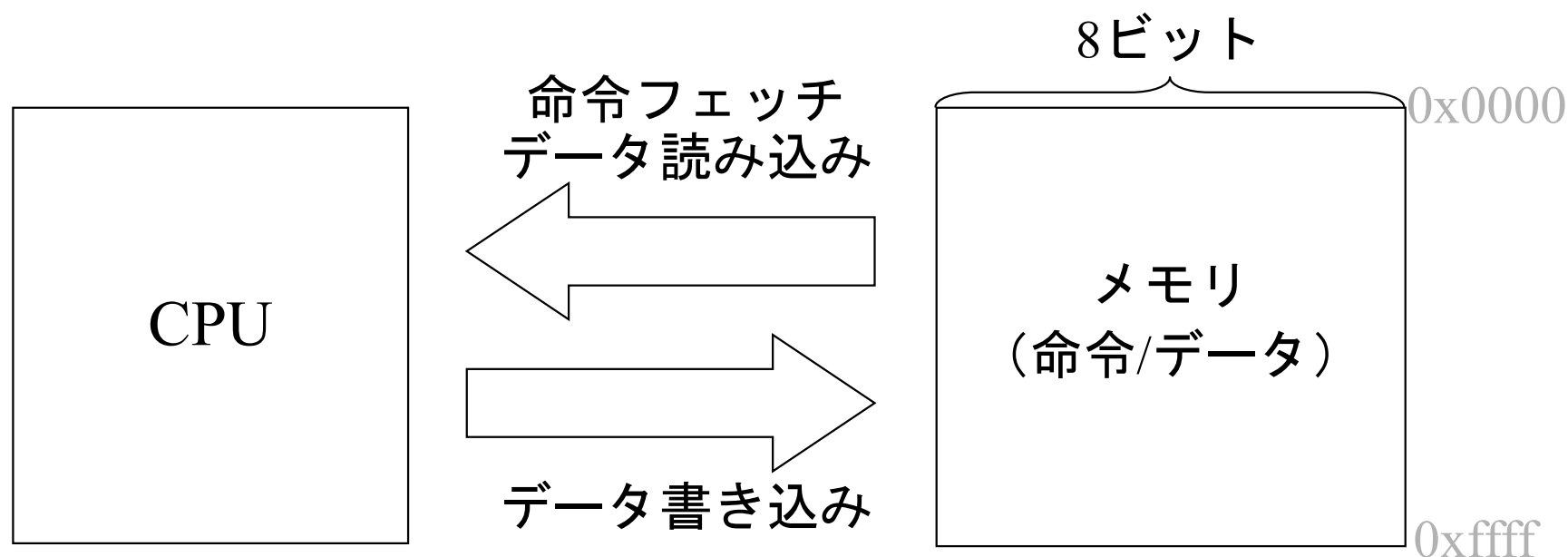


ARM The Architecture for the Digital Worldより参照

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

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;  
}
```

コンパイル結果：機械語

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

アドレス

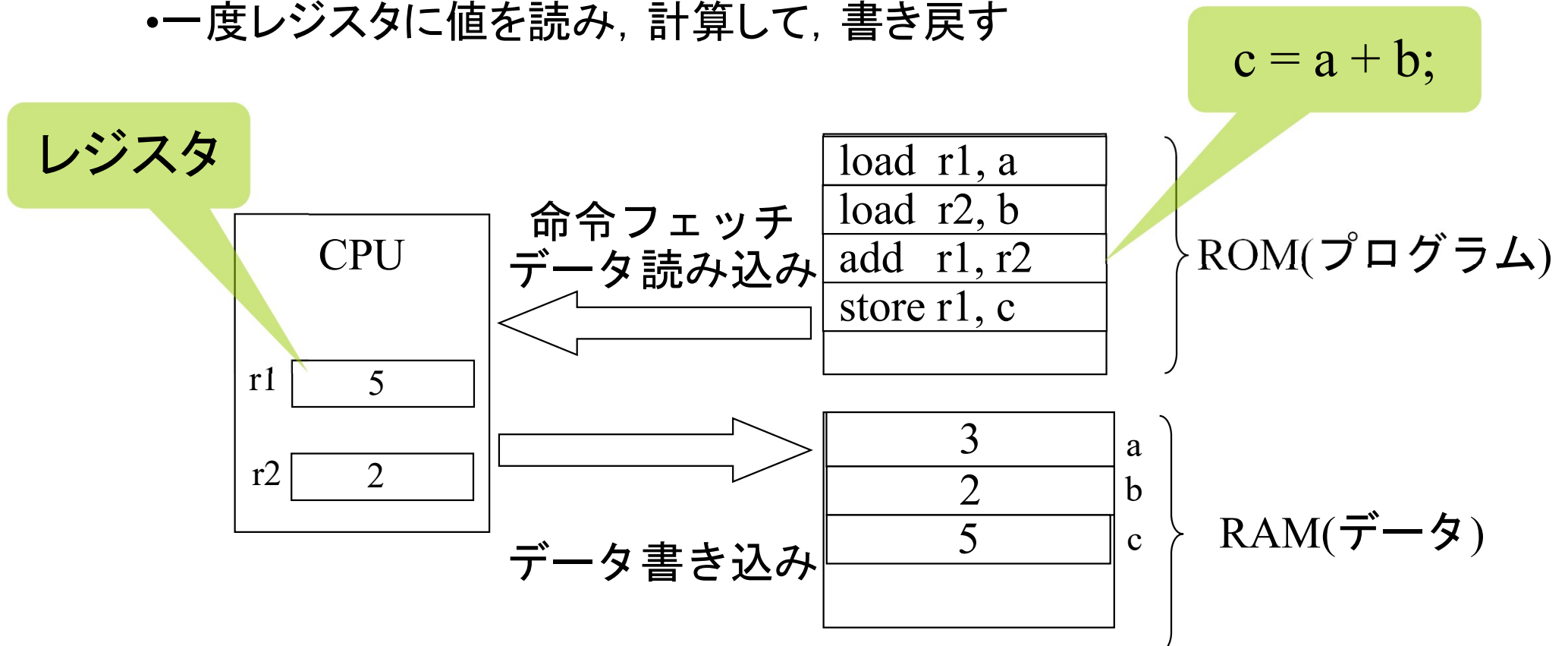
メモリへの配置

メモリ	アドレス
	0x60
変数a	0x61
	0x62

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制御レジスタ

(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)

制御レジスタ

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

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

特権モード移行時
R13(SP)
のみ変更となる

ARSP	APSR
IPSR	IPSR
EPSR	EPSR

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

		31	30	29	28	27	26	25	24	23	20	19	16		10			9	8	0	
APSR	N	Z	C	V	Q					GE[3:0]											
IPSR																0 or Exception Number					
EPSR						ICI/IT		T	ICI/IT												

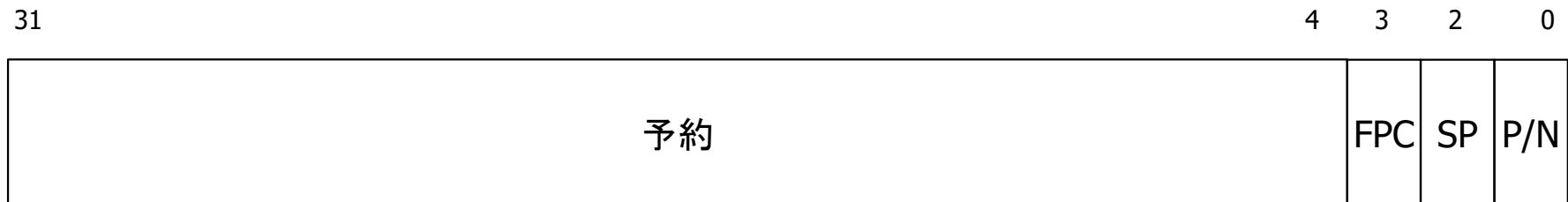
優先順位用特殊レジスタ

- 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

	31		8	7		0
PRIMASK	RESERVED					PM
FAULTMASK	RESERVED					FM
BASEPRI	RESERVED				BASEPRI	

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, SPI, I2C...

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

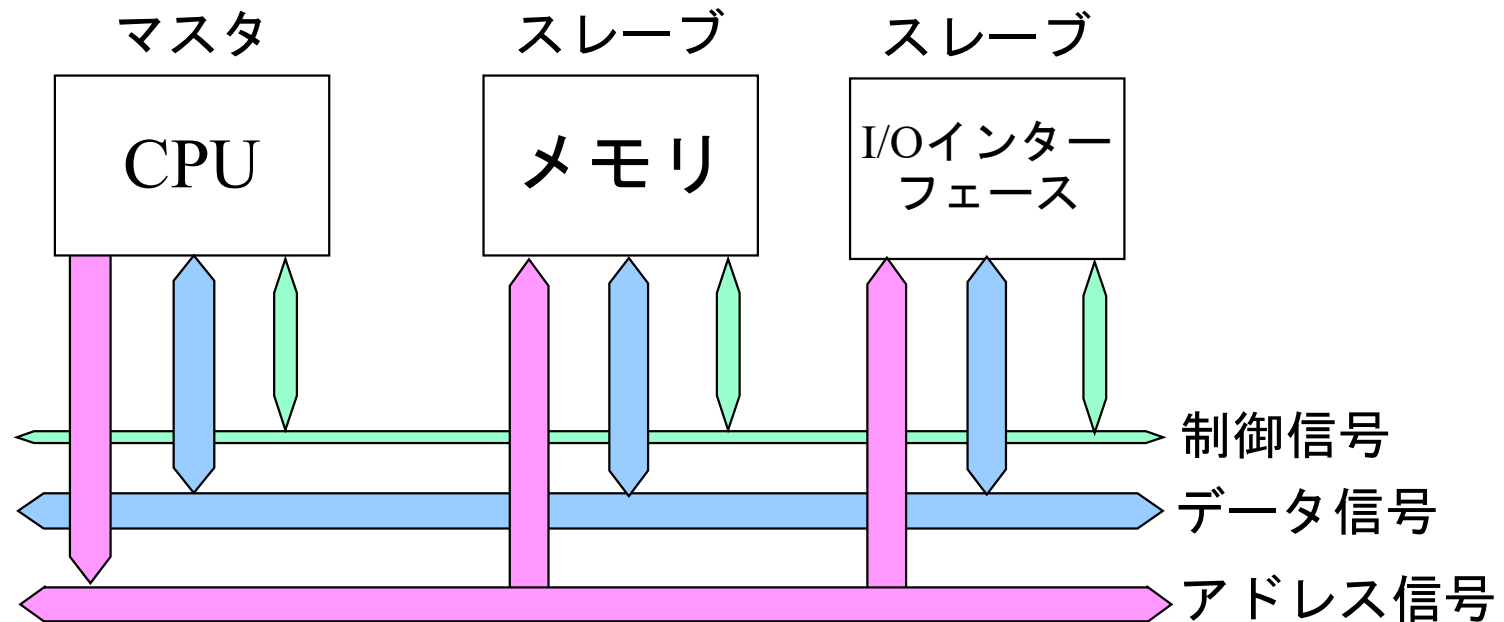
- マスタ側は出力, スレーブ側は入力

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

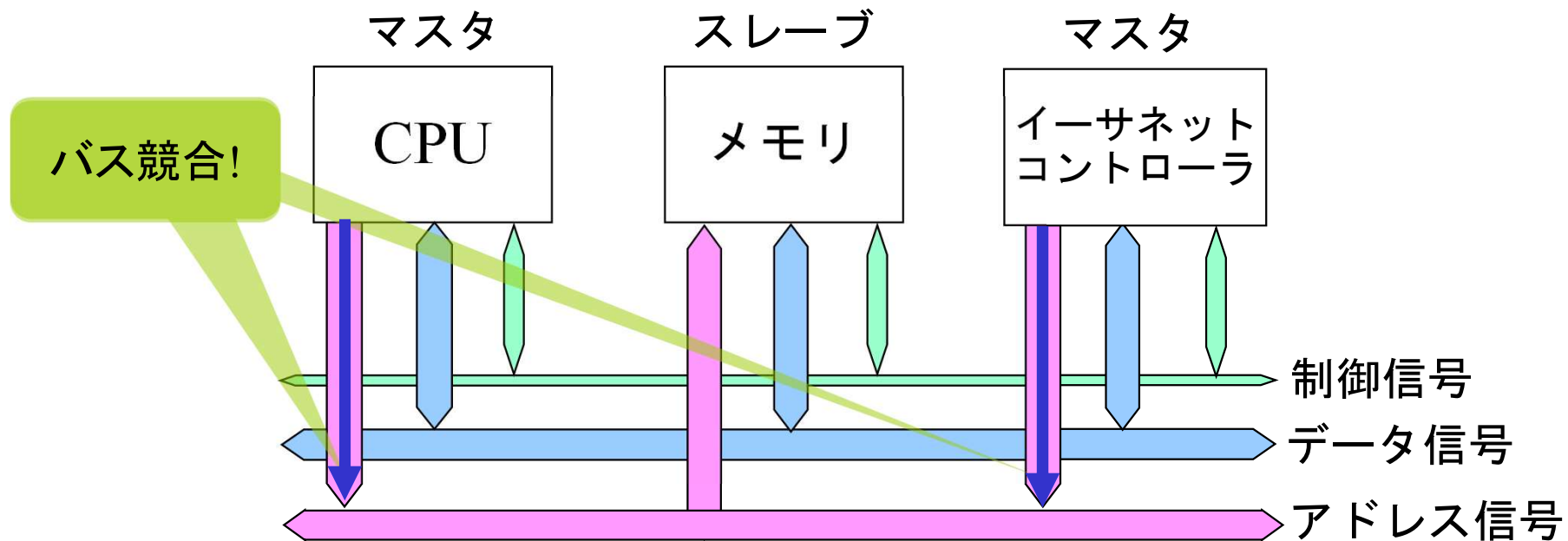


バス：バス競合

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...



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

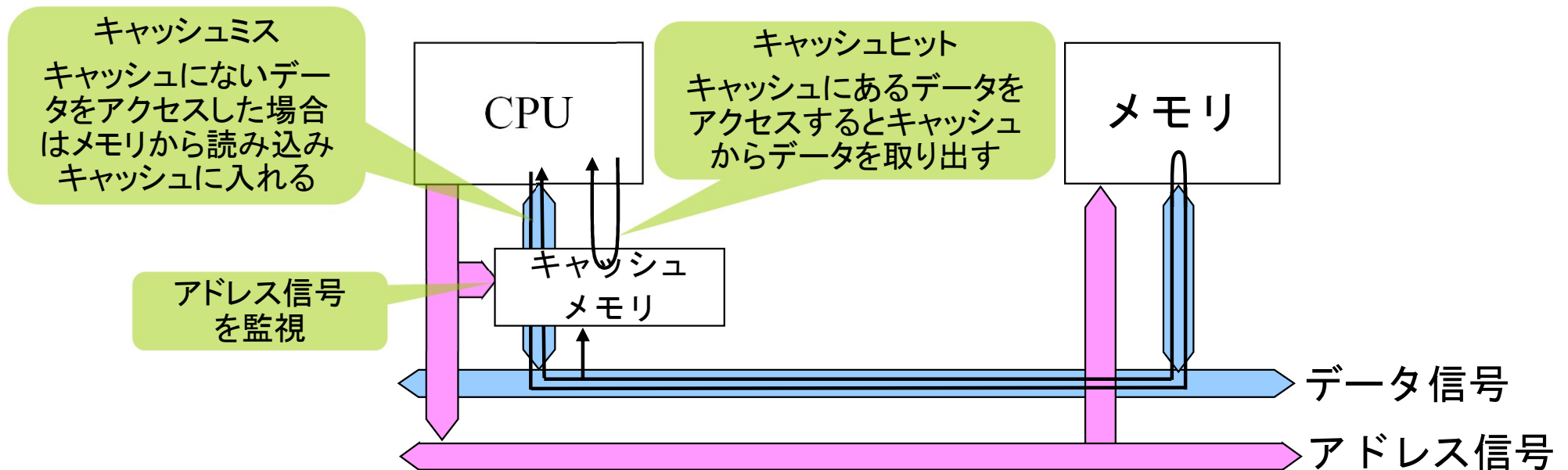
キャッシュ (1/2)

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、次回はそのちからを使用する



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
ビッグエンディアン

- 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

ビッグエンディアン

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

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

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

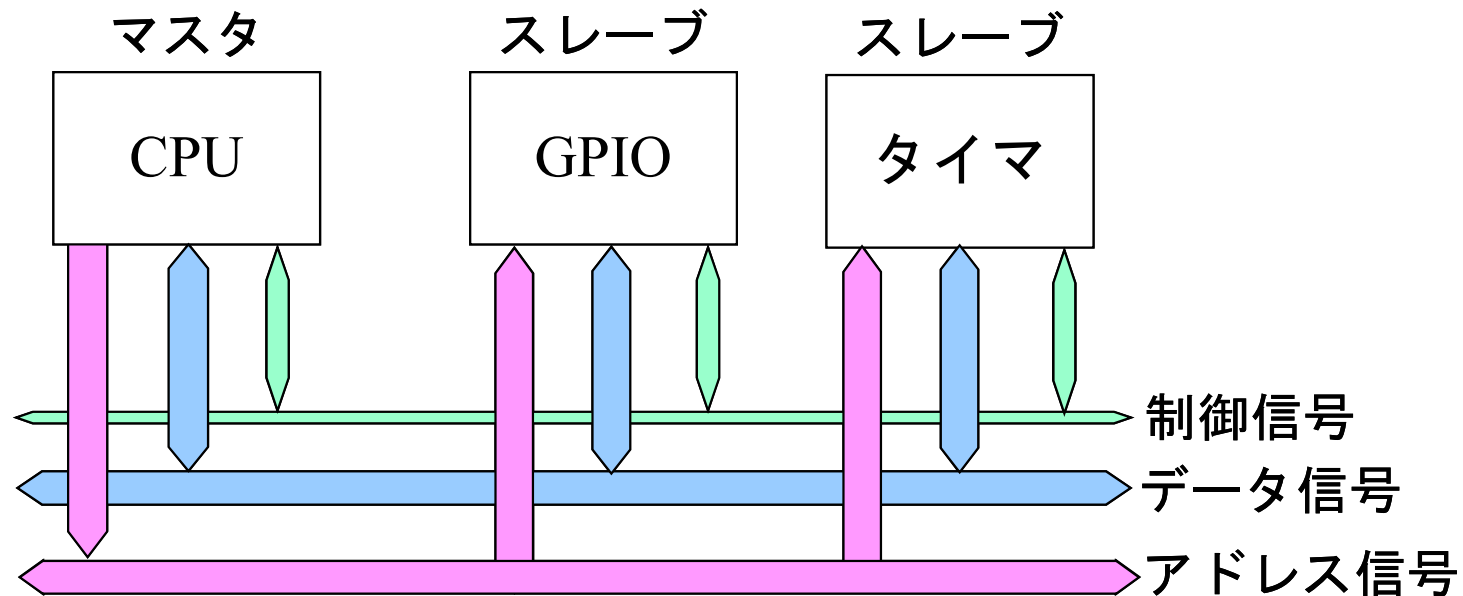
周辺デバイス

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（RP2040タイマ）

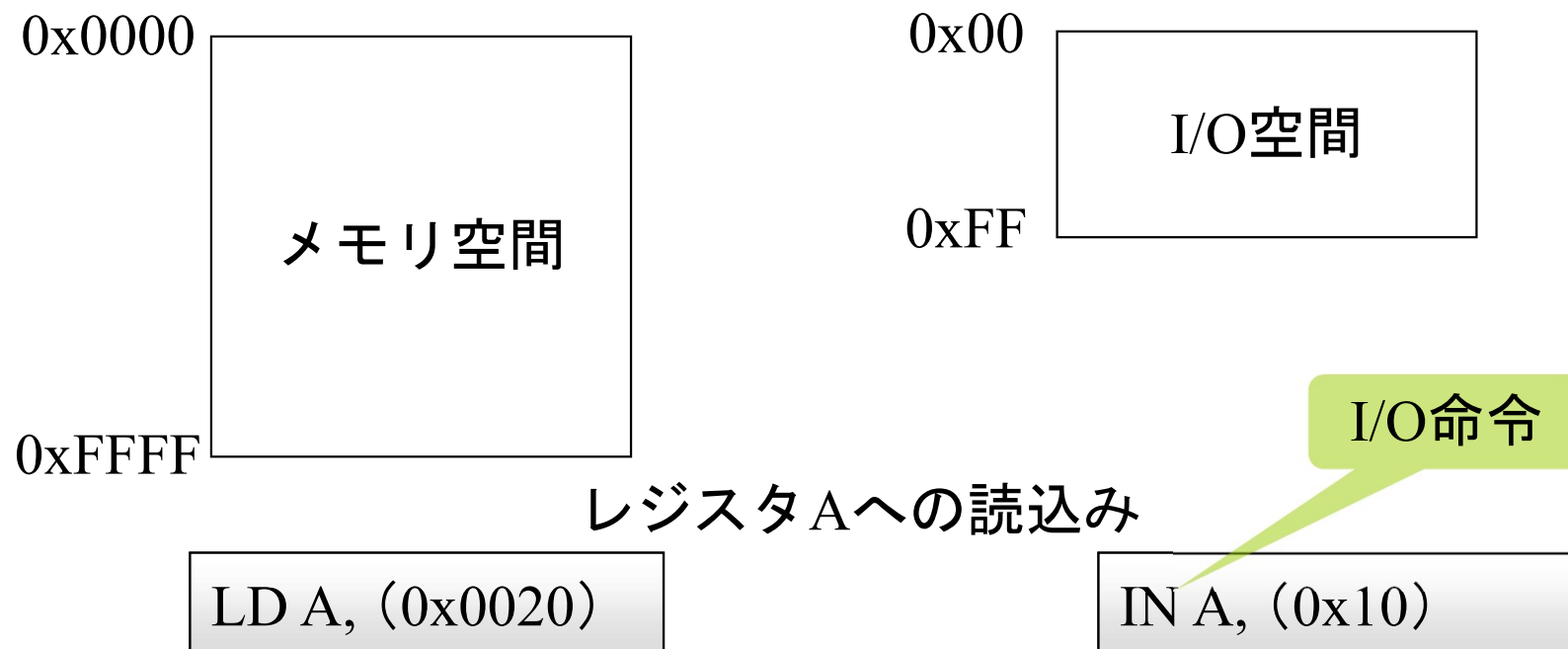
名称	アドレス	R/W	アクセス サイズ
タイマアラーム レジスタ	0x40054010～ 0x4005401C	R/W	32
タイマレジスタ	0x40054028	R	32
タイマイネーブル レジスタ	0x40054038	R/W	32 (4bits)

タイマレジスタがタイマアラームレジスタに設定した値に一致すると割込みが発生するアラームレジスタと割込みは4つ用意されており
イネーブルレジスタの対応ビットを1にすると割込み許可

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

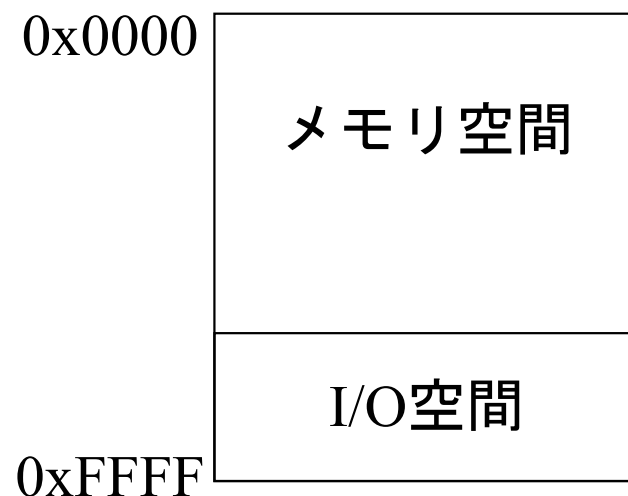
- 周辺デバイスのアクセス時にはキャッシュを無効にしなければならない。

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

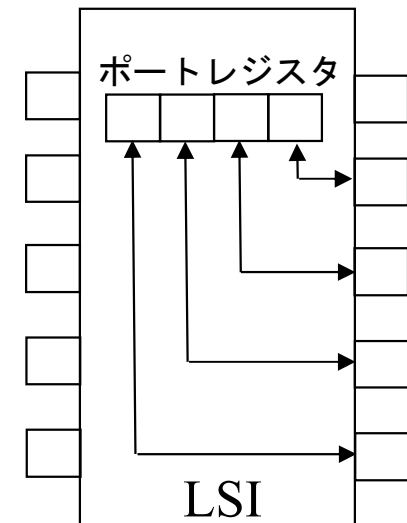
- ポートレジスタの設定内容が、ポートから出力される

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

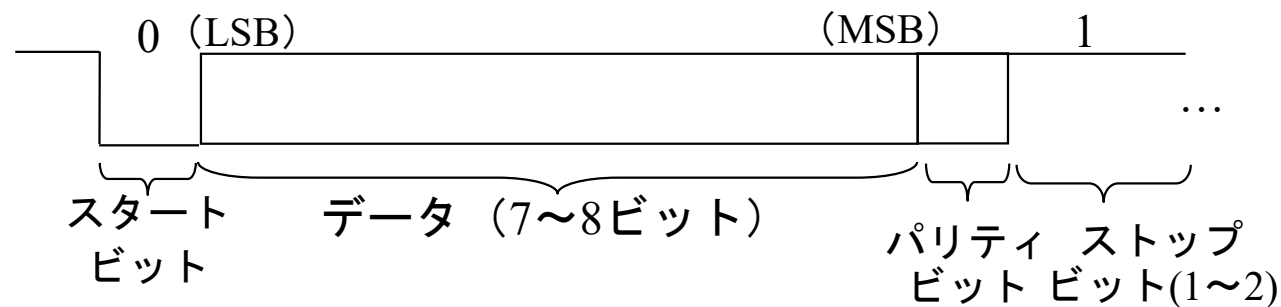
周辺デバイス：SIO(シリアルI/O)

単一のデータ線を用いて外部と通信を行う方式

- UART(RS-232C), I²C, etc...

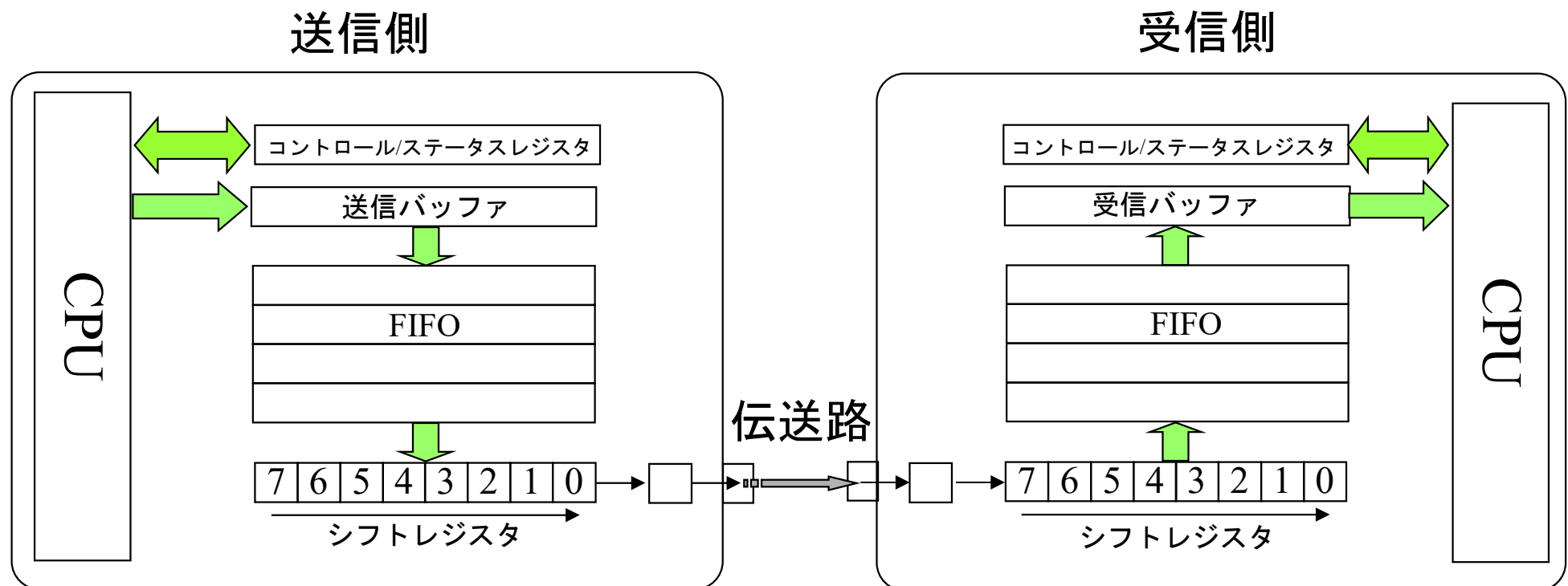
プロトコル

- データ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

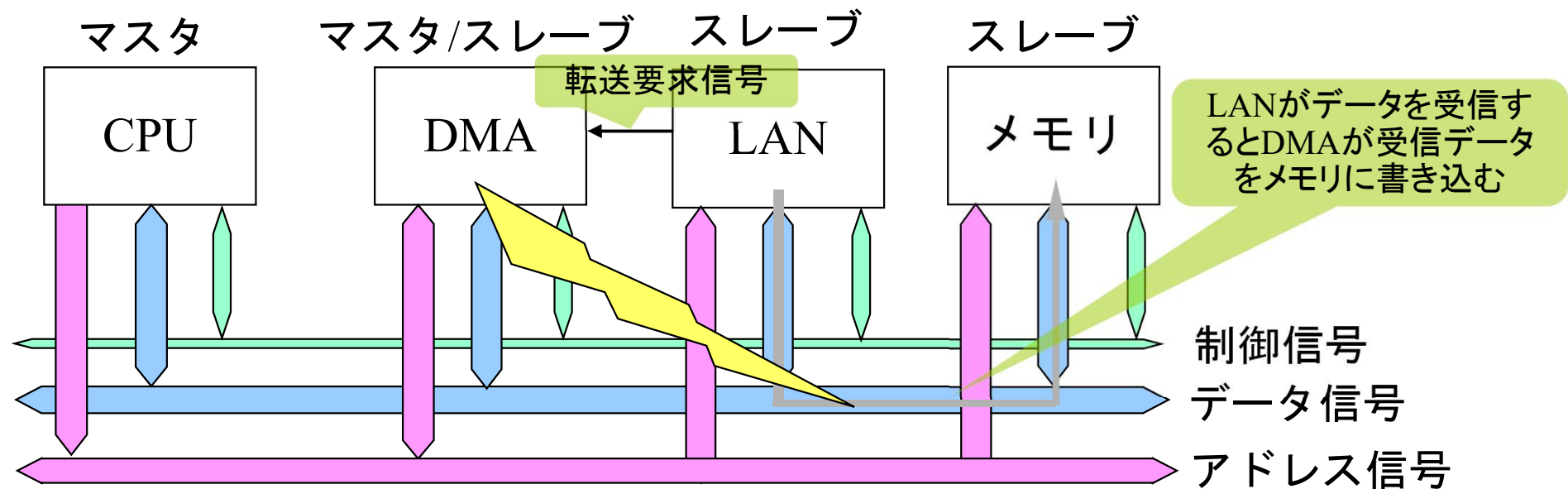
ハードウェアタイマ

- 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

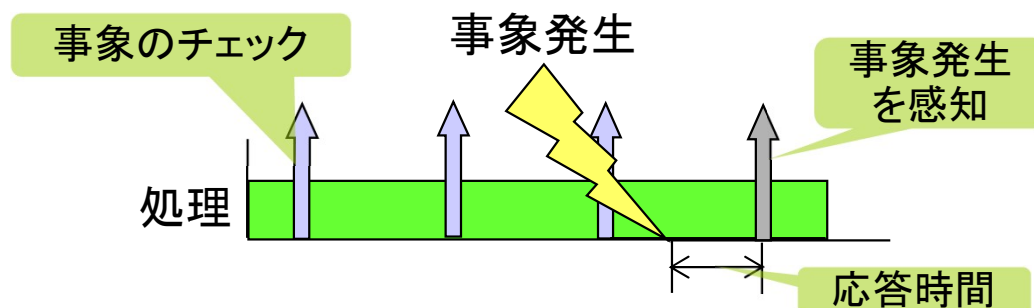
- 外部事象が発生すると, 対応したレジスタのビットがセットされる
- プロセッサ(プログラム)で該当ビットを繰り返しチェックする方法

チェックの間隔

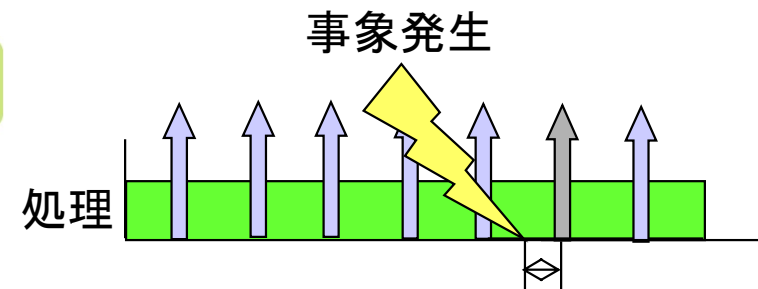
- チェックの間隔を短くすると, 処理に占めるチェックのためのオーバーヘッドが増大する
- チェックの間隔を広げると事象が発生してから, その発生を感知するまでの時間が長くなり, リアルタイム性が低くなる

➡ オーバーヘッドとリアルタイム性のトレードオフ

チェック間隔長



チェック間隔短



外部事象の待ち方：割込み

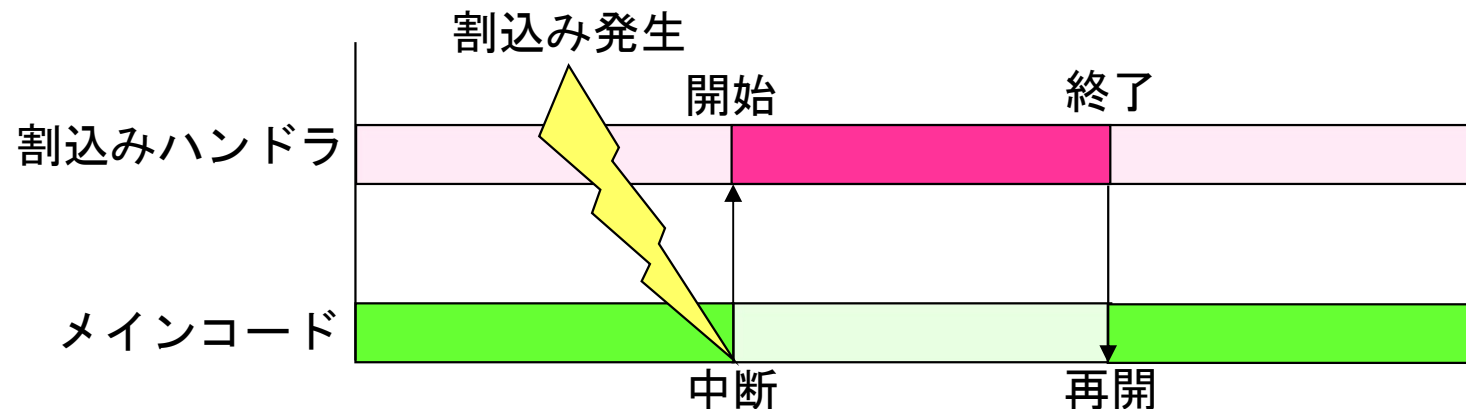
プロセッサが事象の発生を非同期に受け付ける方法

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



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(メーカー)によって異なるので注意！

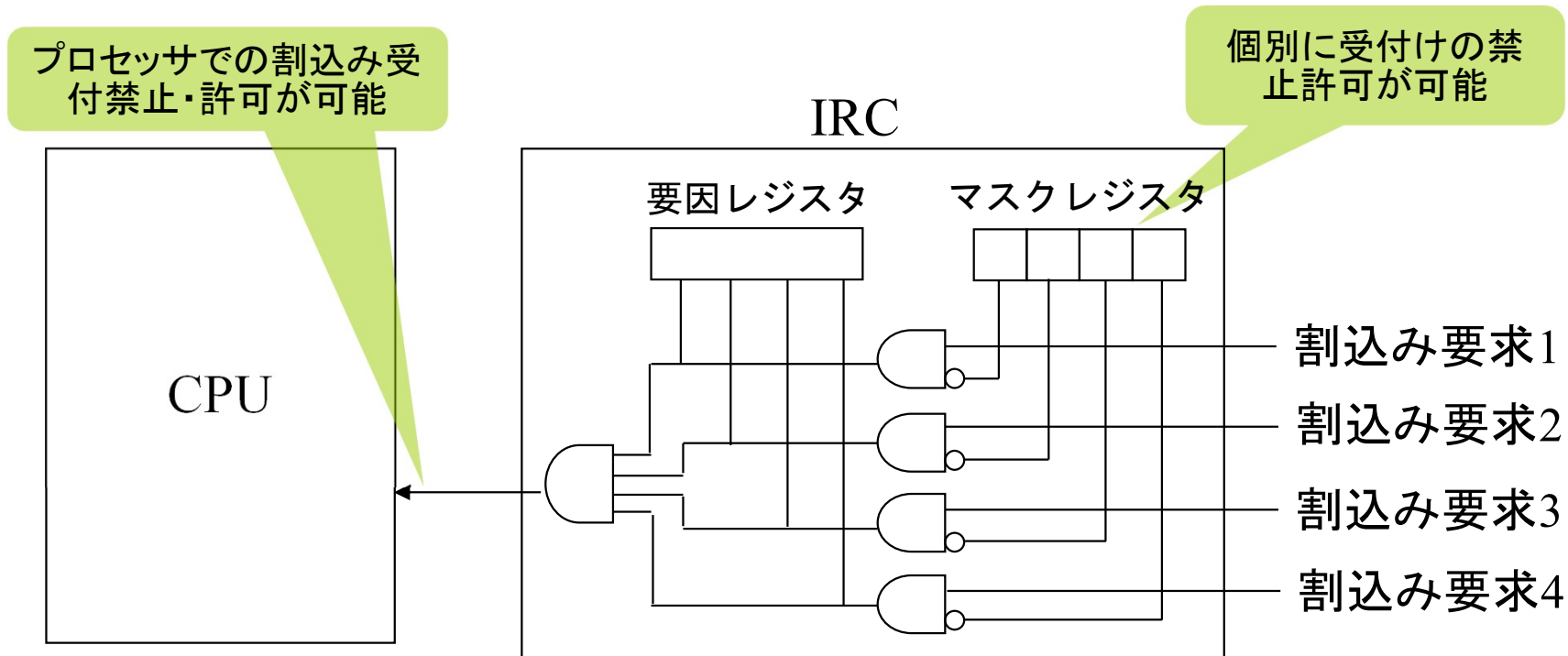
多様な割込みアーキテクチャ

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

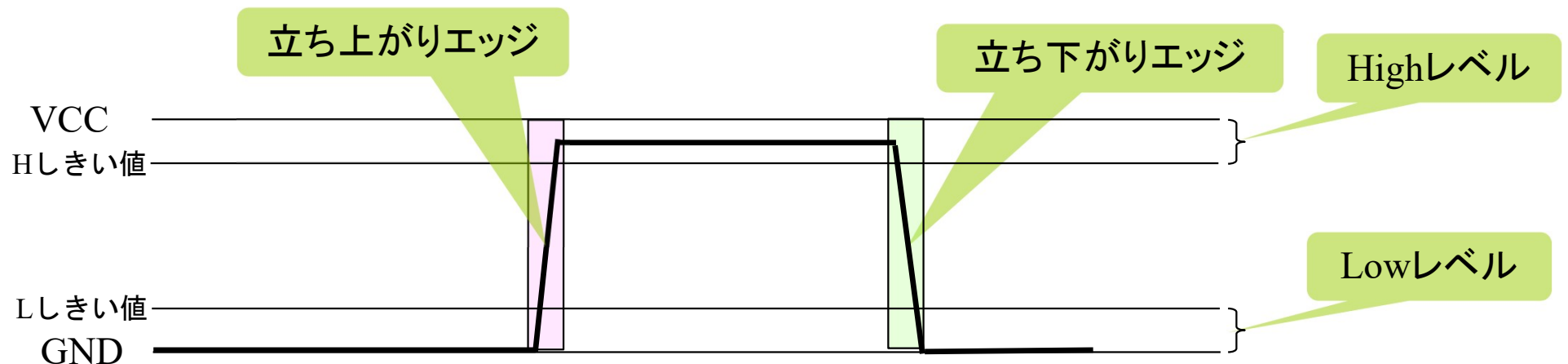
割込み要求の発生方法

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



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

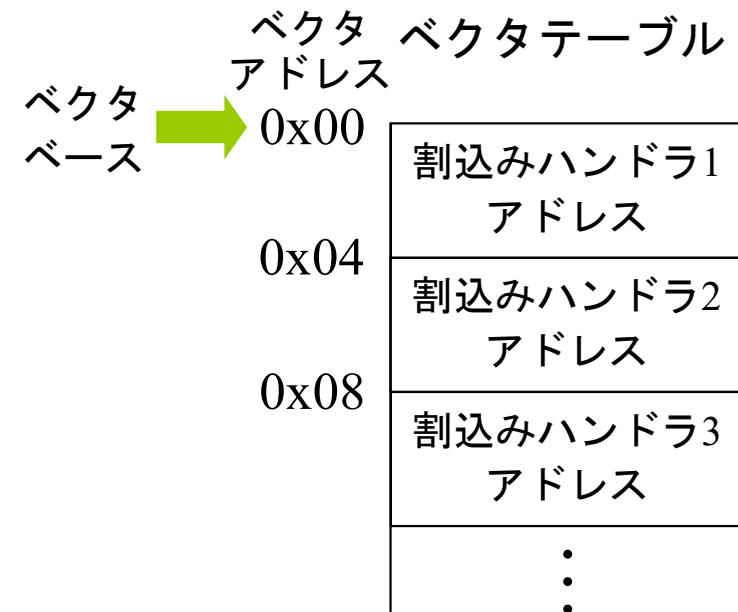
- プロセッサは割込みを受け付けると割込み要因を判定して、その要因に対応した割込みハンドラの先頭アドレスを読み込みジャンプする
- CISC型のプロセッサで一般的な方式

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



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.

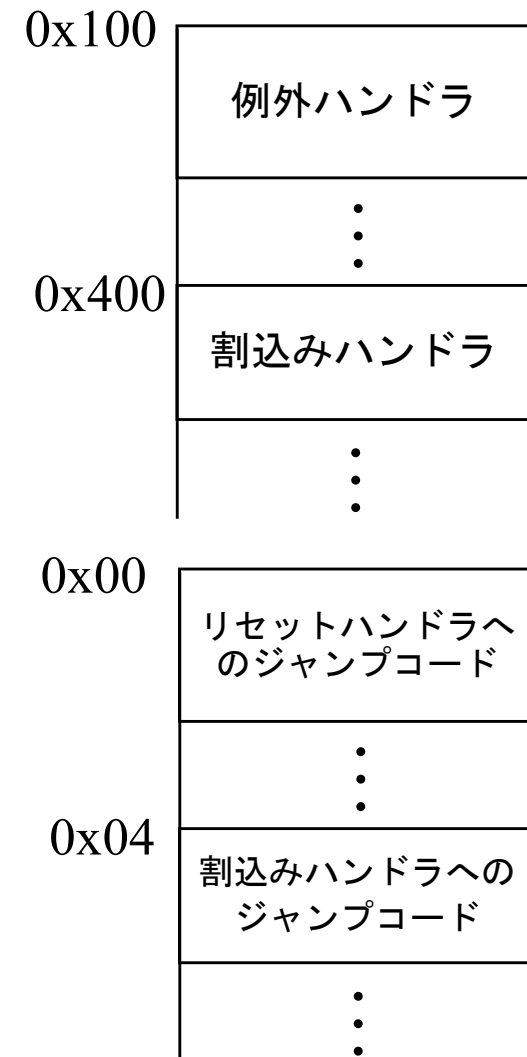
それぞれのアドレスにハンドラを置く

- 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



プロセッサの割込み/例外処理の例

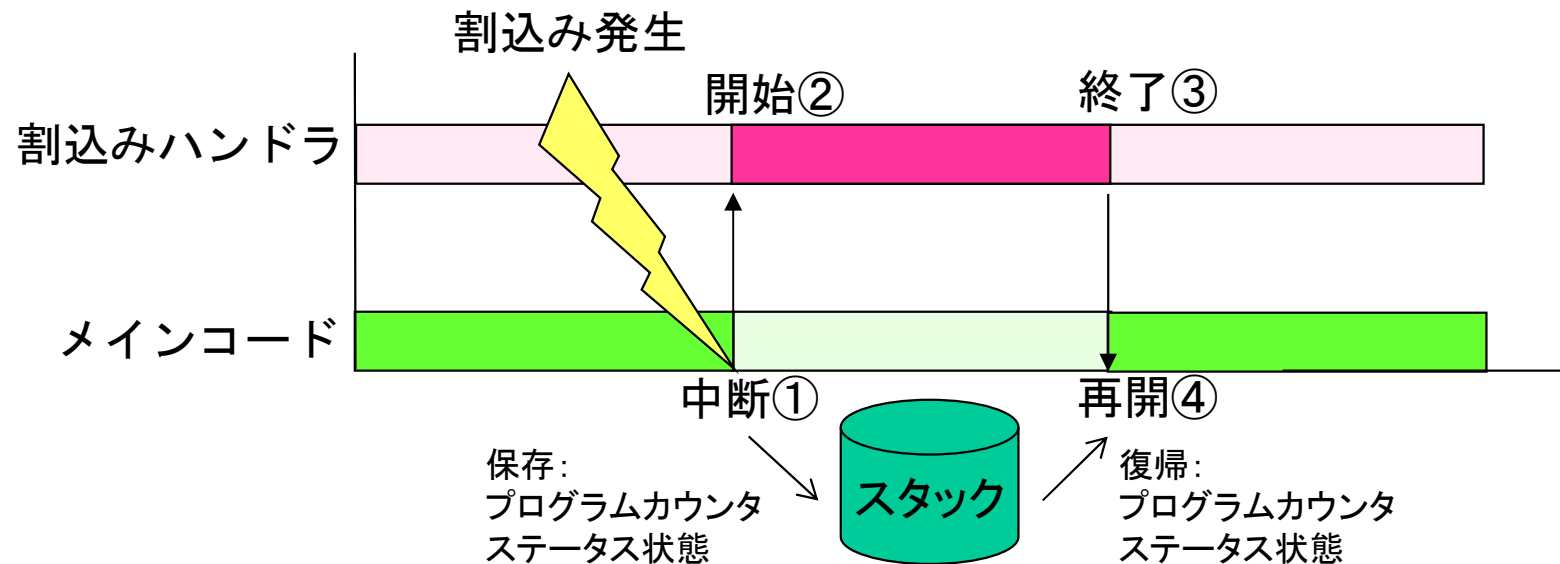
①メインコード実行中に割込みを受け付けると、割込み要因に対応したベクタテーブルに登録されているアドレスにジャンプする。

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

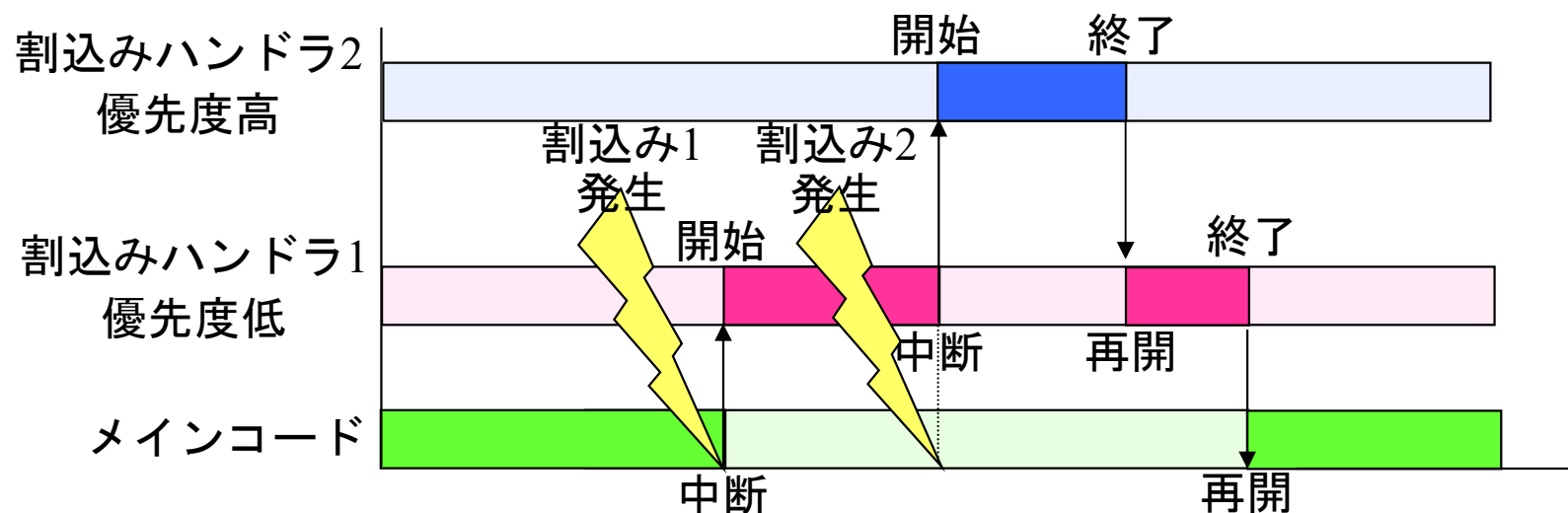


多重割込み：高優先度の割込みの発生

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

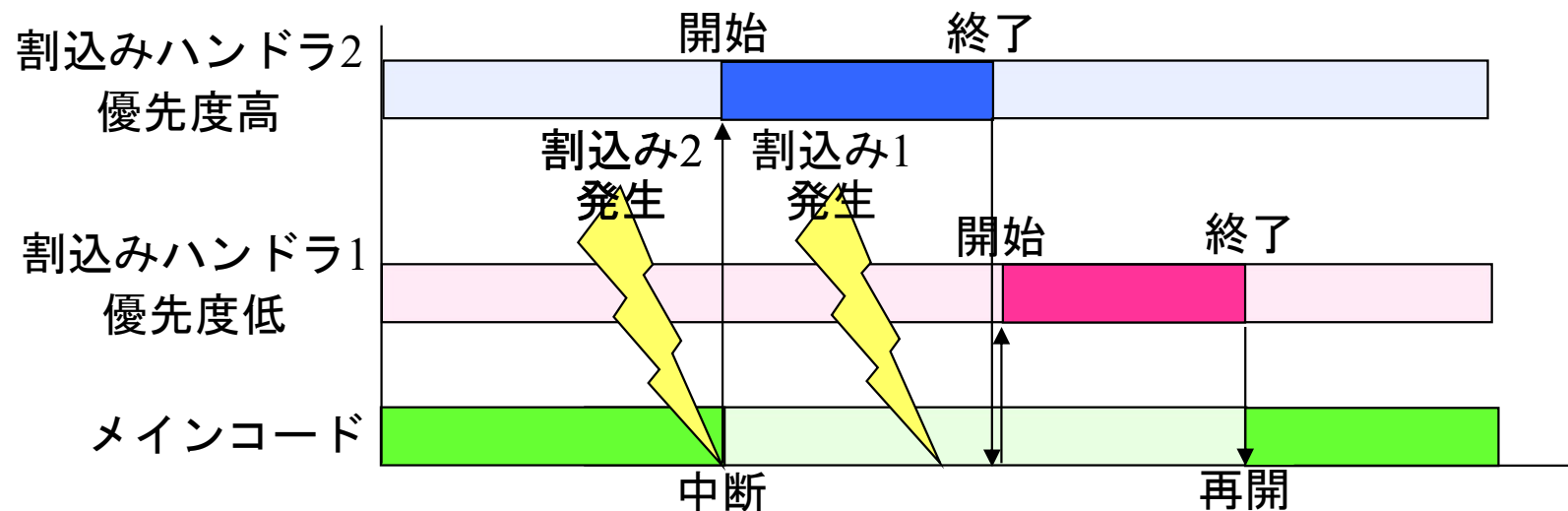
割込み優先レベル

- 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



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

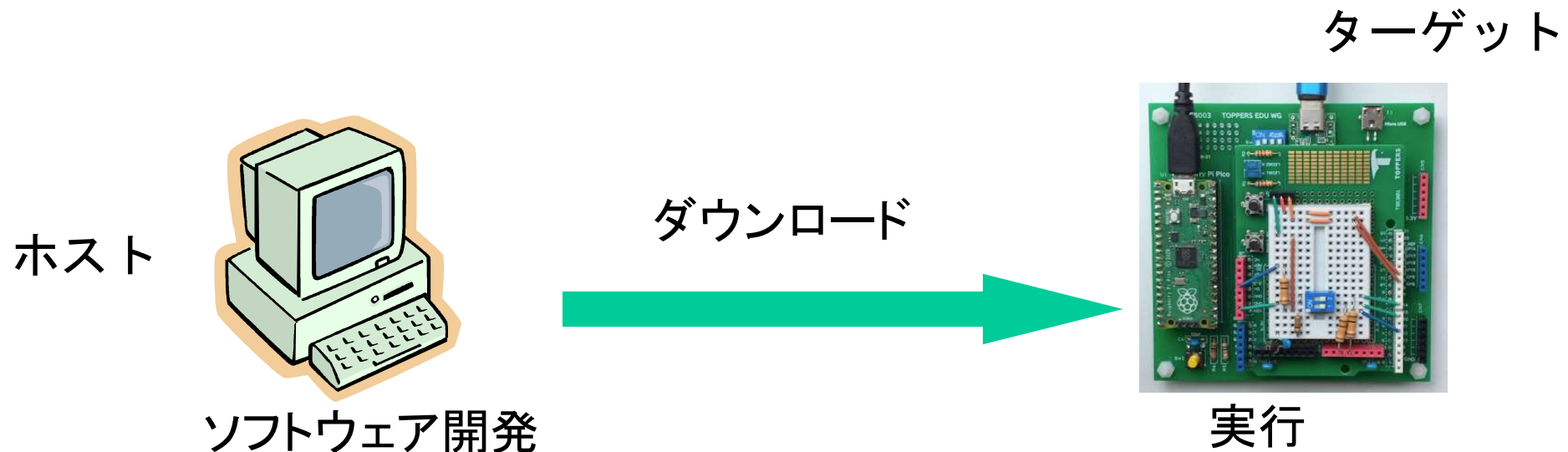
組み込みシステムはクロス開発により開発する

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



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
- 組み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ポ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，テストが終了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))`

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

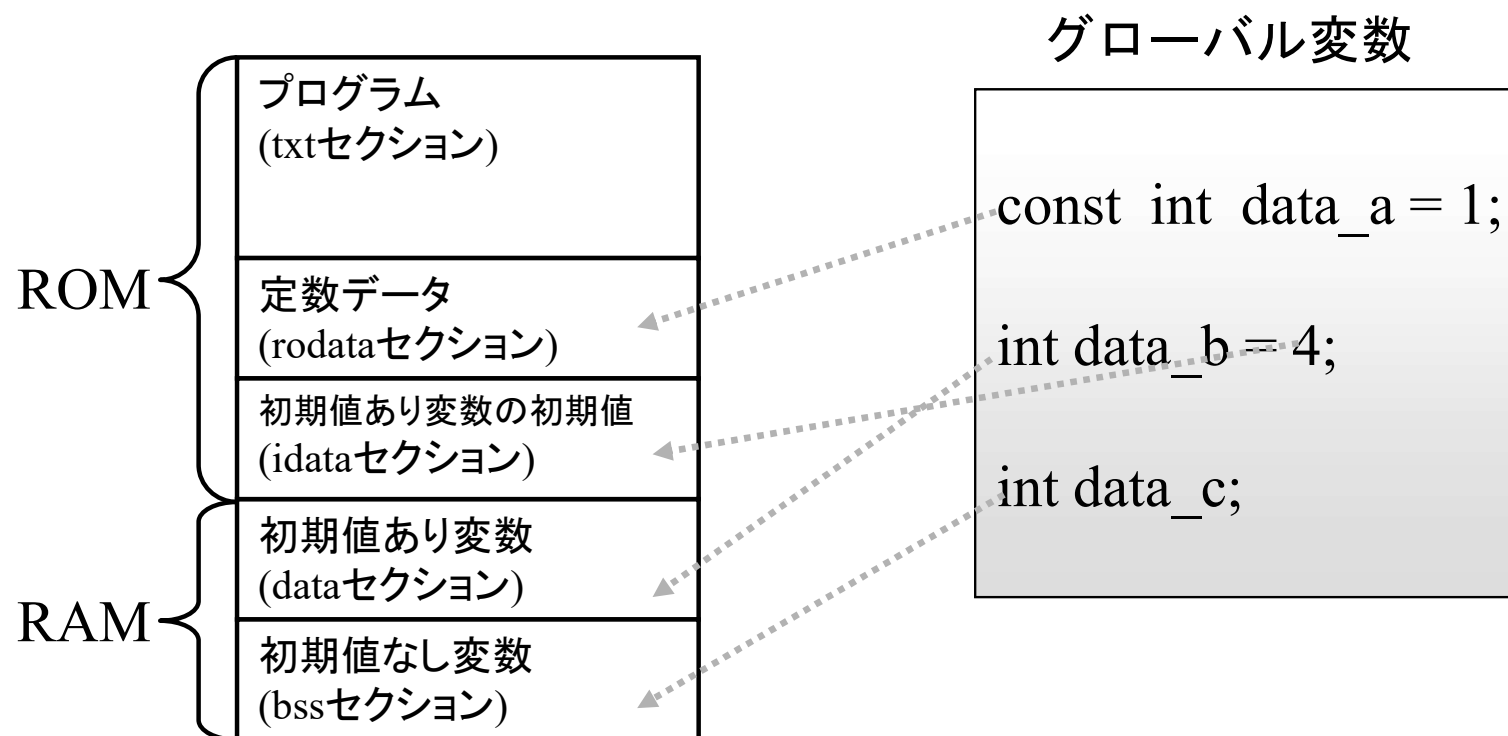
独自セクション

- 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)

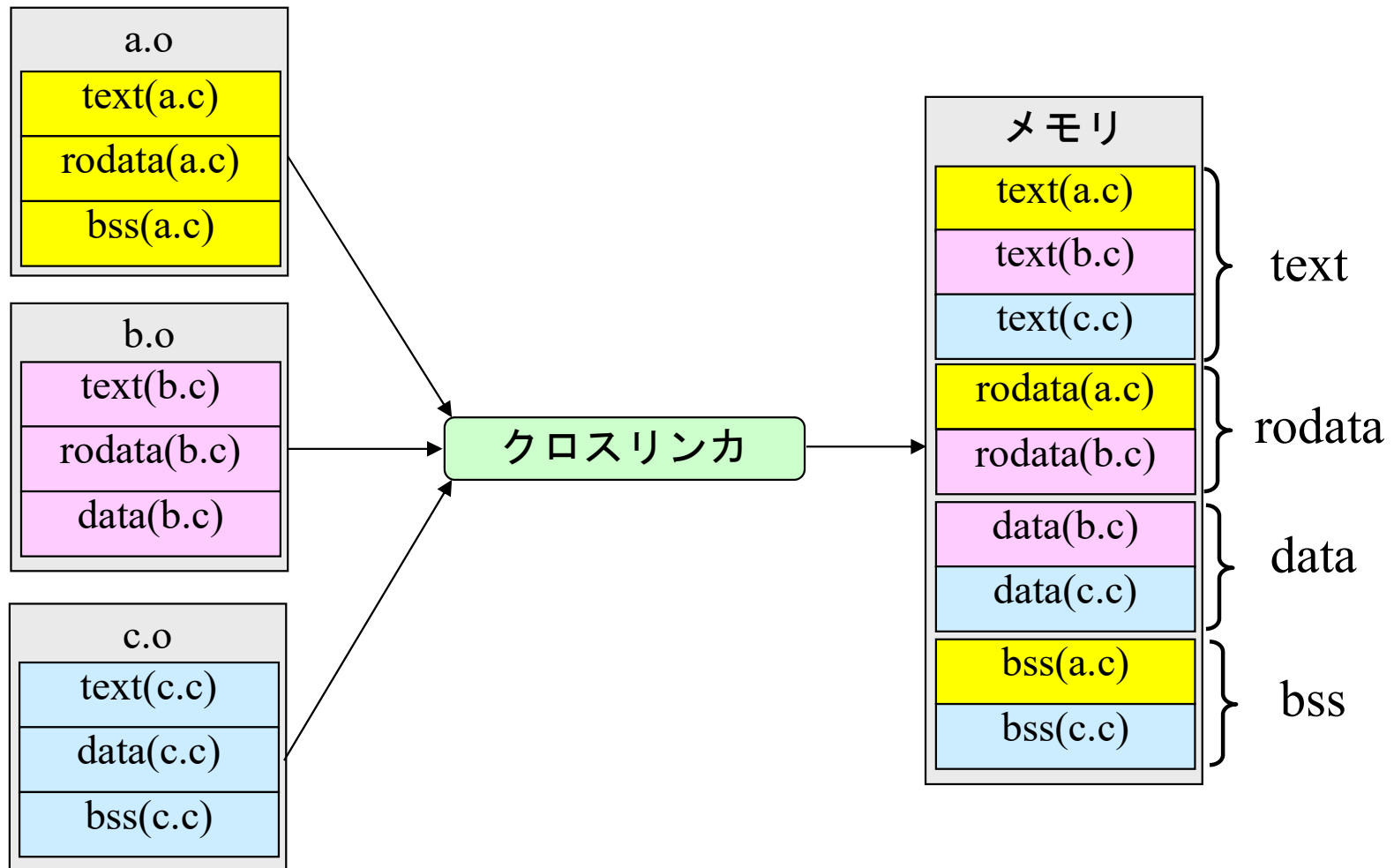
セクションの指定

- 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
  .....
}
```

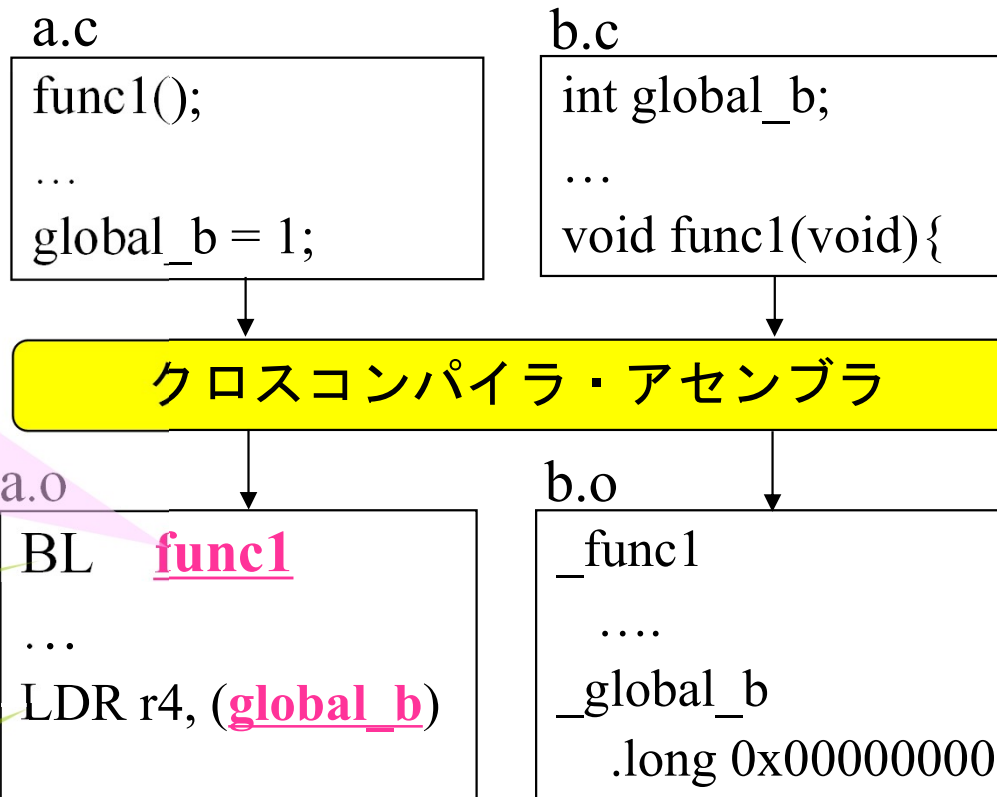
クロスリンカ：アドレス(シンボル)の解決

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

実際のアドレスは、
クロスリンカによる
ローケーション決
めの後にしか分
からない(クロス
コンパイラは分か
らない)



関数呼び出し

グローバル
変数の参照

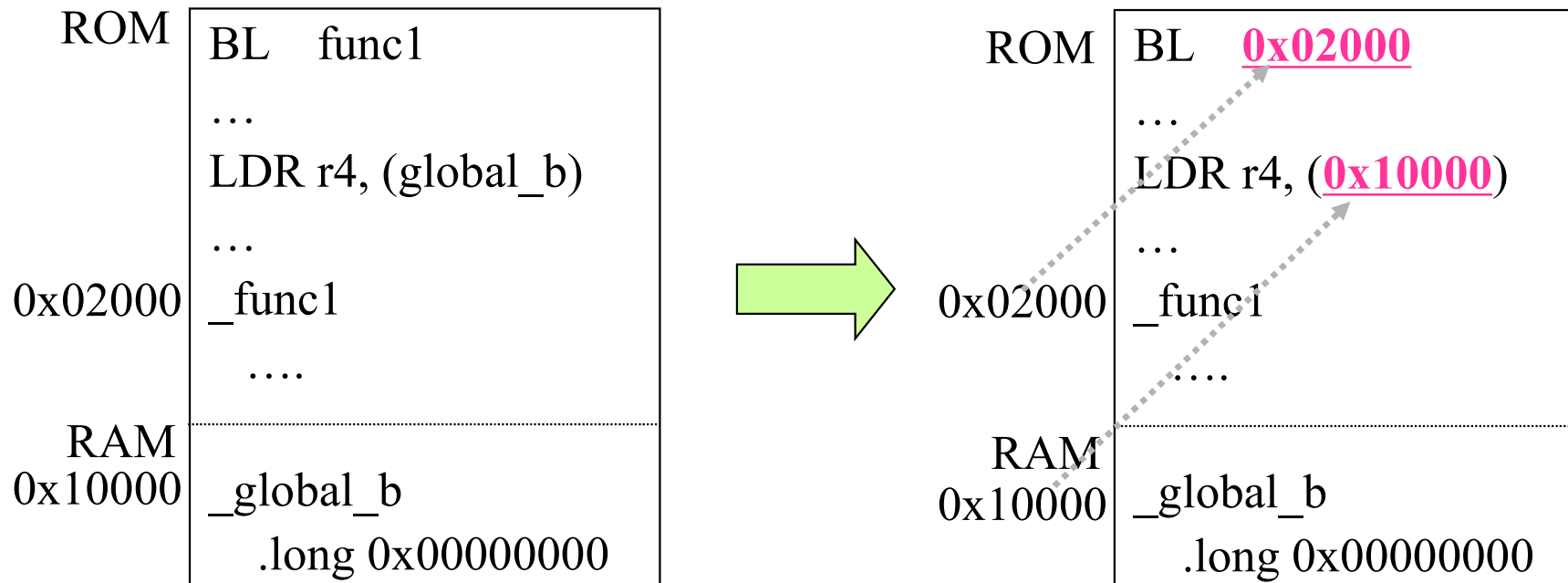
クロスリンカ：アドレス(シンボル)の解決

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

ロケーション決め

アドレス解決



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)

ソースコード

```
extern int  global_b;  
extern void func2(void);  
int global_a;  
  
void func1(void){  
    global_b = 1;  
    func2();  
}
```

シンボル情報

シンボル名

属性	シンボル名
U	_func2
U	_global_b
b	.bss
d	.data
t	.text
T	_func1
C	_global_a

シンボル値

00000000
00000000
00000000
00000000
00000010

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：ROM化可能コードの生成

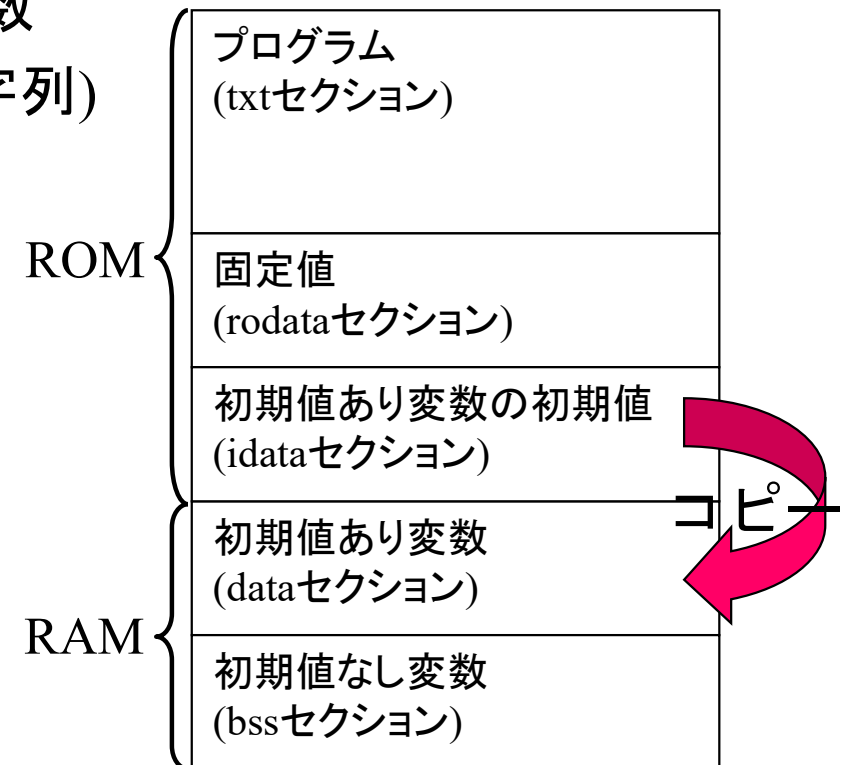
プログラムコードをROMに置いて動作させる方法

- 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



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

組み込みソフトウェアのデバッグ環境

組み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
- 実習では、ROMモニタを使用する

リモートデバッガ(対デバッガ)

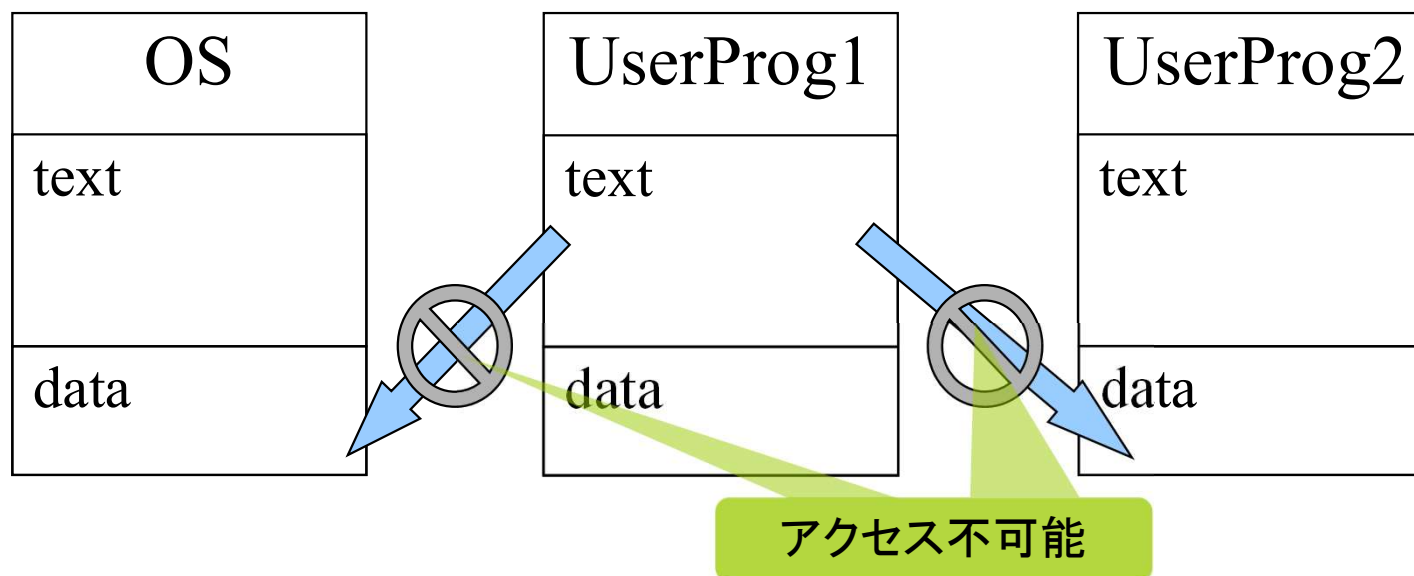
- 高度なデバッガ機能をホストシステムのデバッガで実現
- ターゲットシステムにはターゲットシステムの状態を参照・設定するための最低限のソフトウェアを実行
 - レジスタ/メモリの読出し/書込み
 - 実行の開始/再開, 1ステップ実行

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

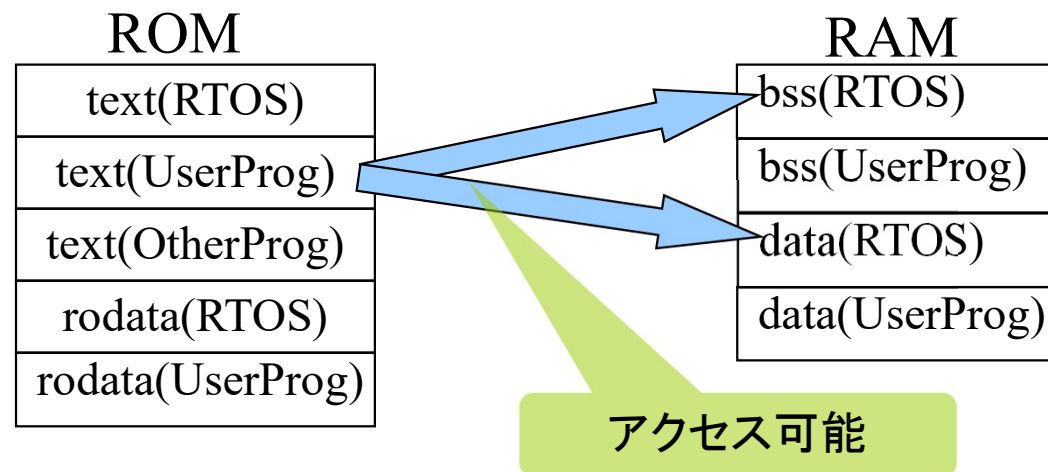
- 各プログラムは、独立したメモリ空間(論理空間)で動作
 - メモリマネージメントユニット(MMU)によるマッピングとメモリ保護
 - 他のプログラムやOSのメモリ空間へのアクセスは不可能
 - プログラムに不具合があり、正しくないメモリ空間にアクセスした場合は、アクセス違反で終了されるだけで、システム全体に影響は及ぼさない



組み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

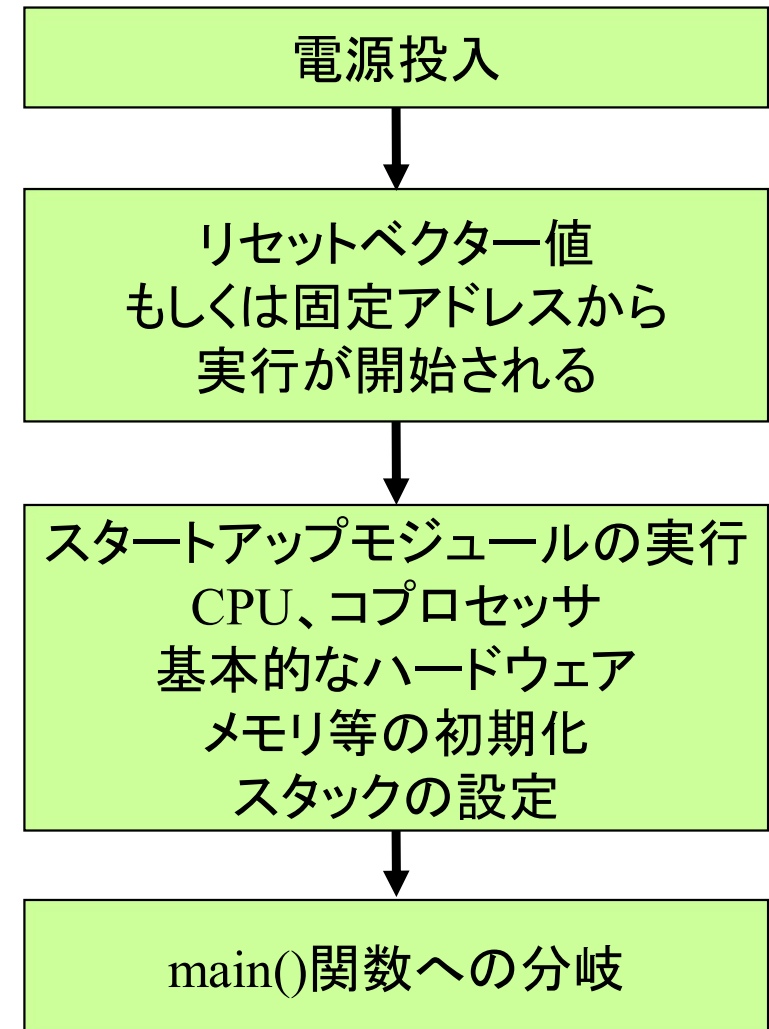
- ユーザープログラムがリアルタイムOSを含む他のプログラムやデータとミックスされて配置され, それぞれは保護されていない
 - ➡ MMUを用いたメモリ保護は実行オーバーヘッドが大きく, リアルタイム性の弊害となる
- ユーザープログラムは容易にリアルタイムOSのデータを破壊可能
 - ➡ システム内のソフトウェアは信頼できるという前提
- メモリアクセス関連のデバッグが困難



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)コンストラクタとデストラクタを実行する
 - 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

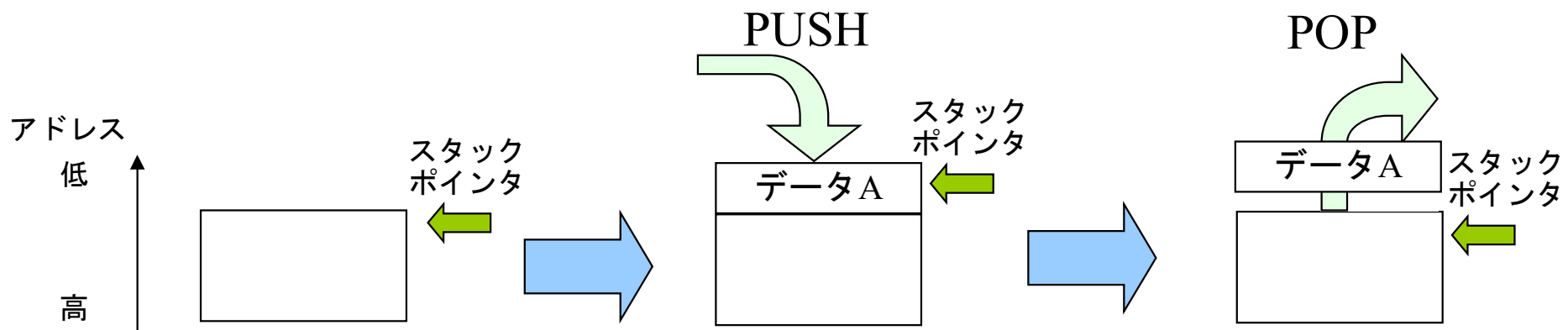
- 後入れ先出し(LIFO : Last In First Out)でデータを扱う
- Push操作, Pop操作, 成長方向は環境依存

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

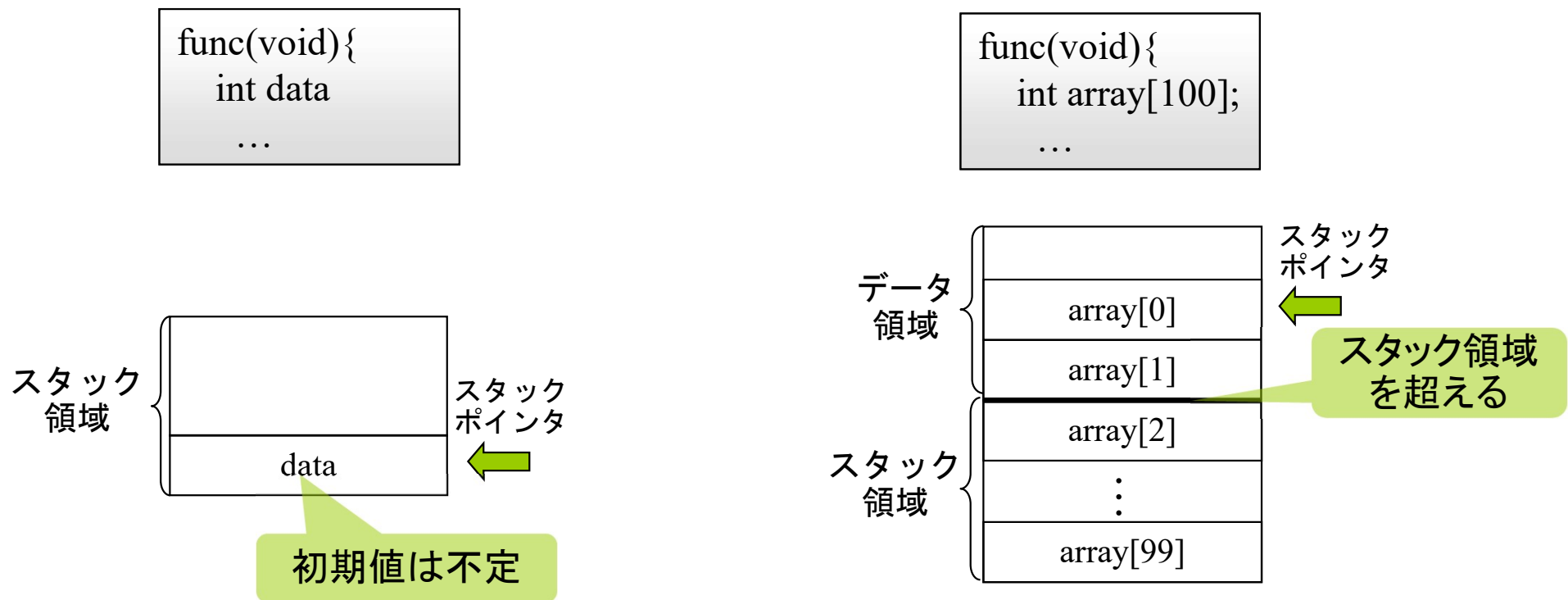


スタック：ローカル変数

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



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

使用
済み

引数2 : 3
引数1 : 2
戻り値 : 5
func1の使用領域

スタック
ポインタ



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;  
}
```

呼び出し側の想定

オフセット	
0x00	引数1(31-24) : 0
0x01	引数1(23-16) : 0
0x02	引数1(15-8) : 0
0x03	引数1(7-0) : 2
0x04	戻り番地(31-24)
0x05	戻り番地(23-16)
0x06	戻り番地(15-8)
0x07	戻り番地(7-0)

スタックポインタ ←

呼び出される側の想定

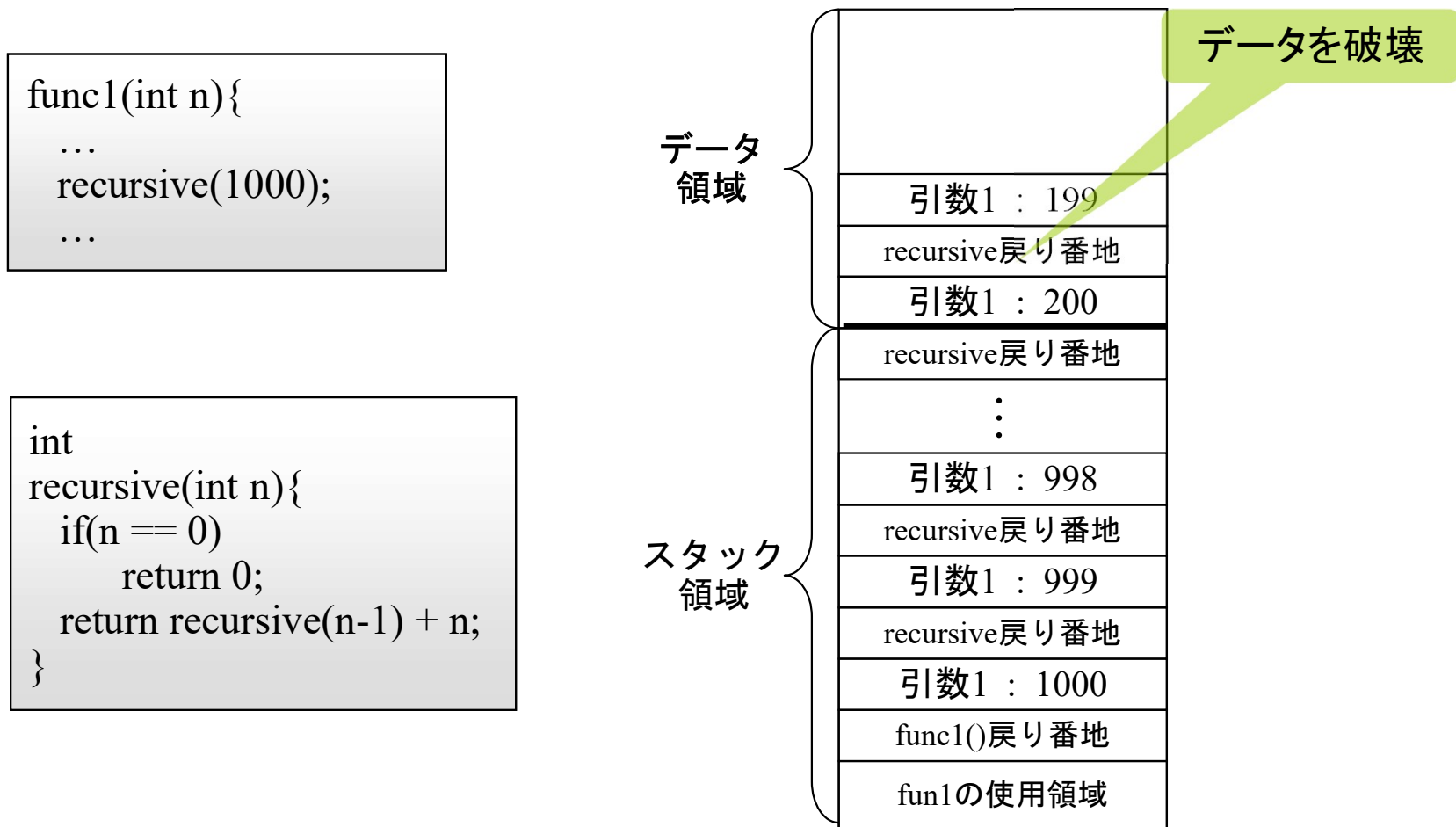
オフセット	
0x00	引数1(15-8)
0x01	引数1(7-0)
0x02	戻り番地(31-24)
0x03	戻り番地(23-16)
0x04	戻り番地(15-8)
0x05	戻り番地(7-0)
0x06	戻り番地(15-8)
0x07	戻り番地(7-0)

スタックポインタ ←

何処に戻るか？

スタック：再帰呼び出し

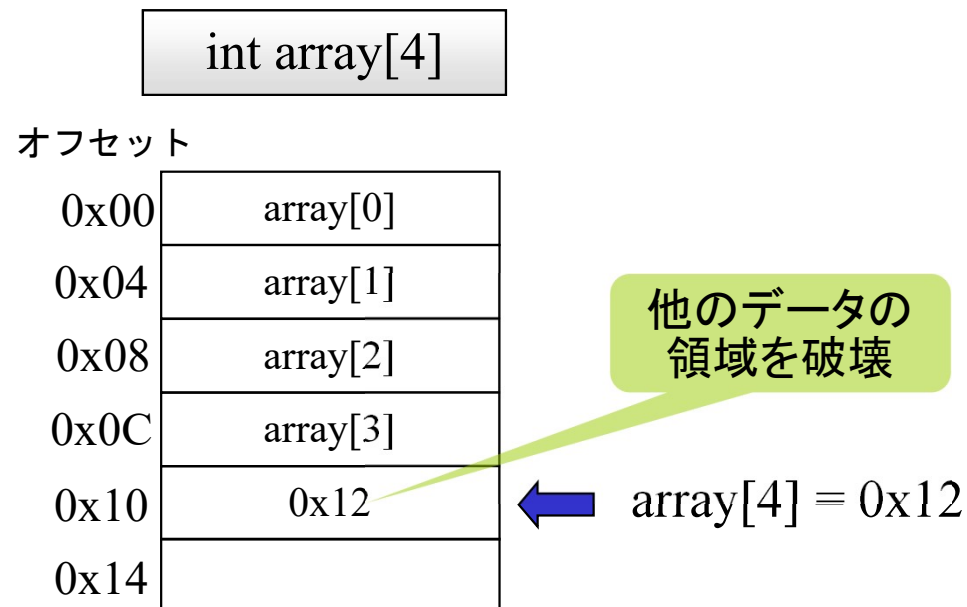
関数を呼び出すたびにスタックを消費するため、
再帰の深さとスタックサイズに注意が必要



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;
```

オフセット	st1_inst
0x00	a(31-24)
0x01	a(23-16)
0x02	a(15-8)
0x03	a(7-0)
0x04	b(31-24)
0x05	b(23-16)
0x06	b(15-8)
0x07	b(7-0)

2バイト
境界



4バイト
境界



	st2_inst
0x00	a(7-0)
0x01	
0x02	b(15-8)
0x03	b(7-0)
0x04	c(31-24)
0x05	c(23-16)
0x06	c(15-8)
0x07	c(7-0)

charは1byte
shortは2byte
intは4byte
とする.



合計7byteのはず



しかし8byte必要

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

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;
```

volatile : 必要性

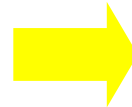
コンパイラの最適化を抑制するC言語の型修飾子

- 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

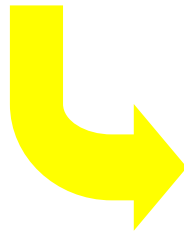
```
short *preg = 0x10000;  
...  
for(;;){  
    if(*preg == 0x0001)  
        break;  
}
```

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

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
 $0x11("00\underline{01} 0001") \& 0x30("00\underline{11} 0000") = 0x10("00\underline{01} 0000")$

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
 $\sim 0x30("00\underline{11} 0000") = 0xCF("11\underline{00} 1111"),$
 $0x21("00\underline{10} 0001") \& 0xCF("11\underline{00} 1111") = 0x01(00\underline{00} 0001)$

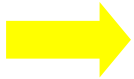
2010年頃からビット演算不要のレジスタ設定(SET/CLR)が主流となり
ビット演算を使用する頻度は下がっている

割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(){  
    ....  
}
```

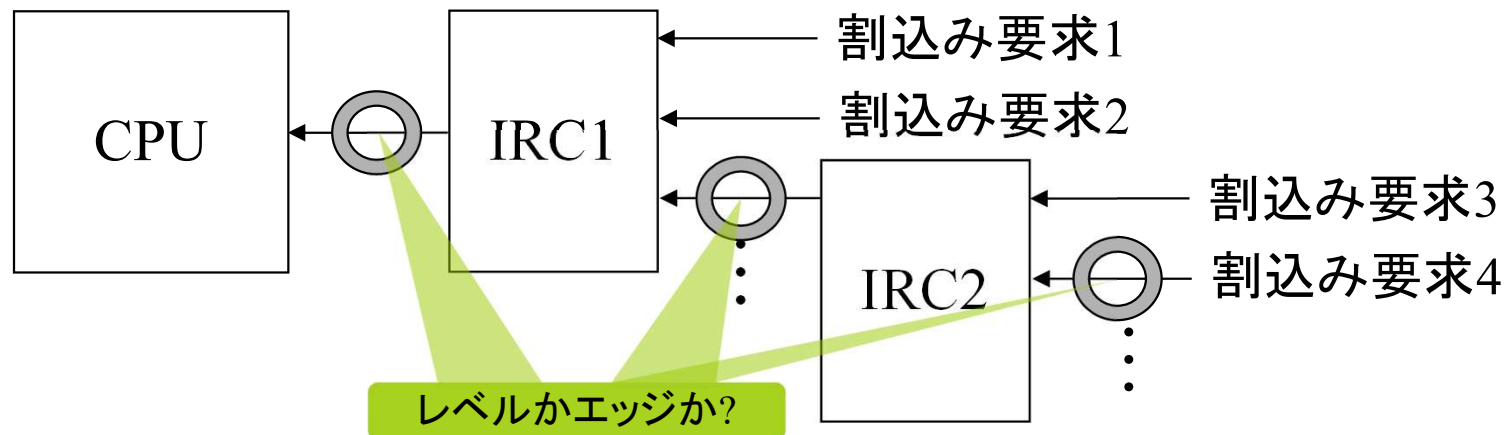
割込み要因のクリア

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



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 */
    :
```

多重割込み

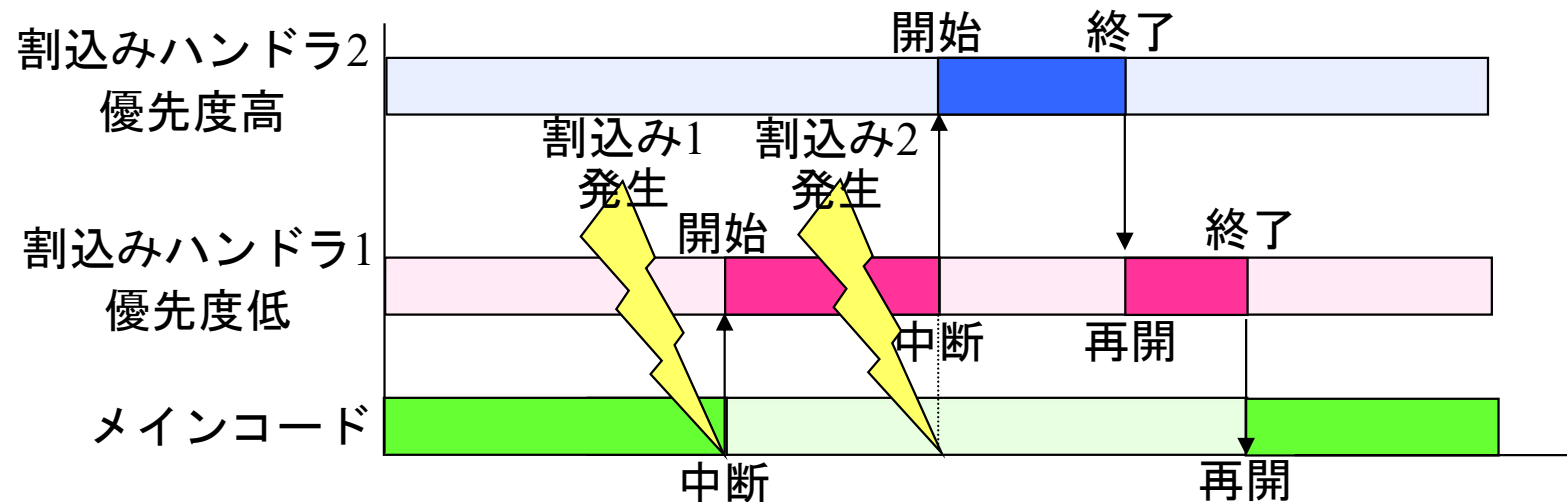
割込みを受け付けた直後

- 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



割込み禁止区間

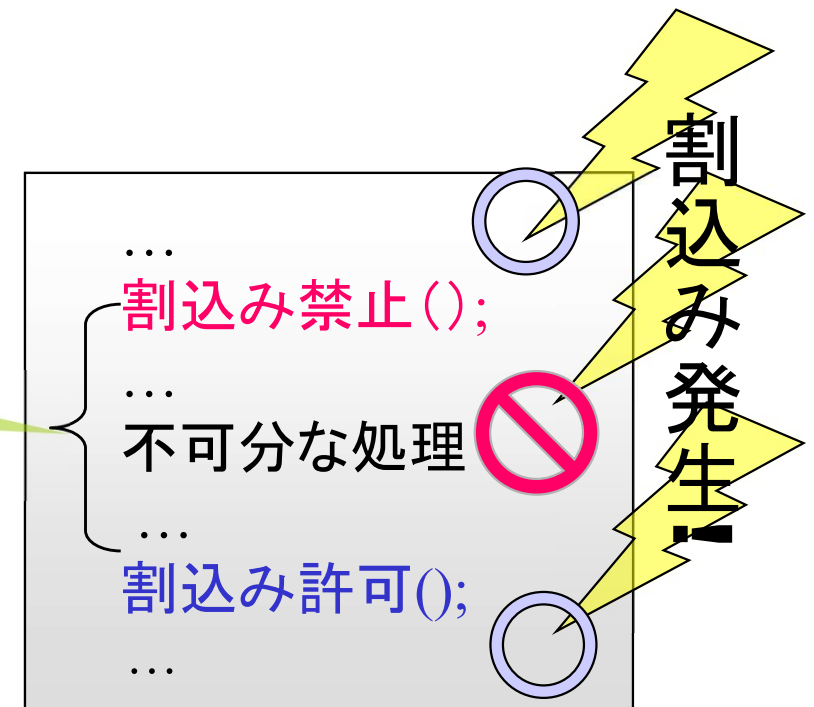
割込み禁止の必要性

- 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

割込み禁止
区間



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

特権モード移行時
R13(SP)
のみ変更となる

ARSP	APSR
IPSR	IPSR
EPSR	EPSR

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

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

マイコンボードの確認

1. [マイコンボード\(Pico/Pico2+TEB003\)](#)
2. プロセッサ (RP2040:Cortex-M0+)
3. JTAGエミュレータ

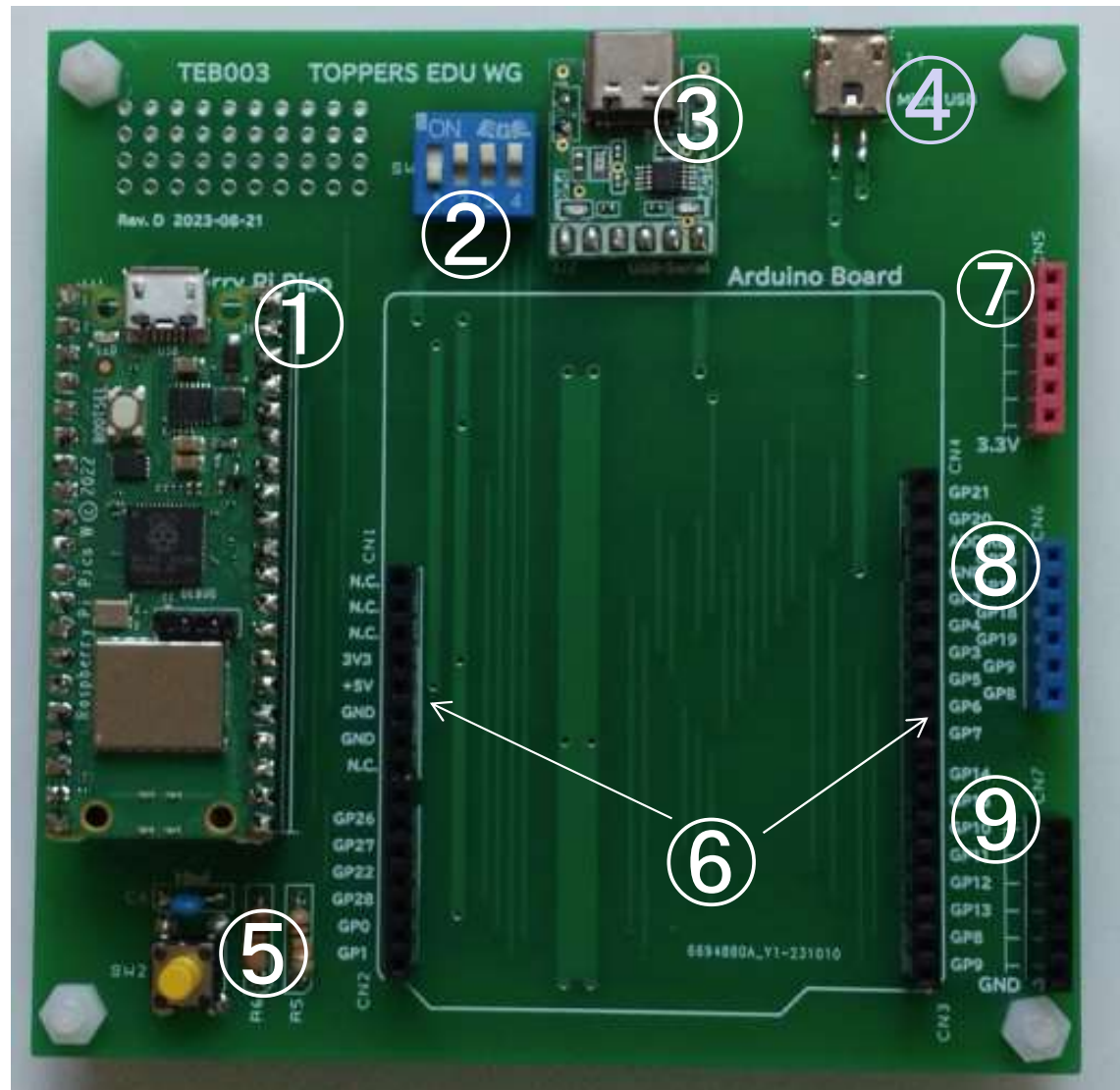
マイコンボードの概要

マイコンボードと搭載プロセッサについて解説する

- マイコンボード : Raspberry Pi Pico(W)/Pico2(W)
 - Raspberry Pi Ltd製
- 検証ボード : TEB003
- プロセッサ
 - Pico:Cortex-M0+ 125MHz
 - Pico2:Cortex-M33またはRISC-V 150MHz
- 実装機能
 - TEB003ボードで、Arduino Unoコネクタが使用可能
 - USBシリアル、USER SW、外部給電

マイコンボード : Pico/Pico2 + TEB003ボード

1. Picoコネクタ
2. DIP SW
3. USB シリアル
4. 外部給電コネクタ
5. USERスイッチ
6. Arduinoコネクタ
7. 拡張GPIO
8. 3.3Vコネクタ
9. GNDコネクタ



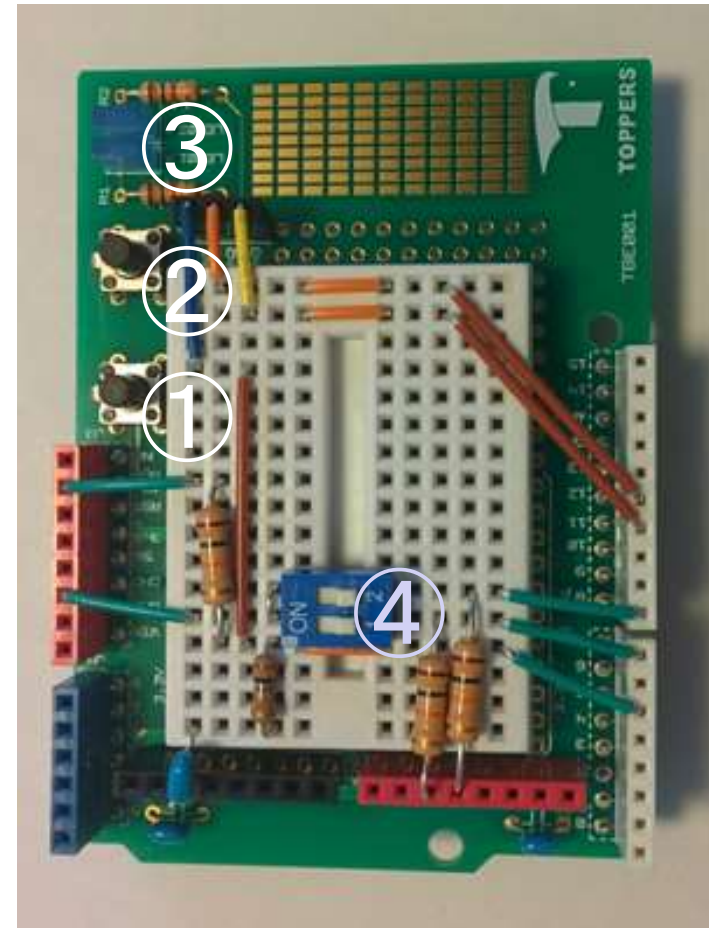
他のPicoボードの使用

- 本講座ではPicoコネクタに、以下のPicoボードを装着して学習ができます
 - Raspberry Pi Pico
 - Raspberry Pi Pico W
 - Raspberry Pi Pico2
 - Raspberry Pi Pico2 W
- Pico2ではCortex-M33、RISC-Vの両方の学習が可能、以下のmakeコマンドのビルド引数で対象を変更可能
 - BOARDNO 0:Pico, 1:Pico2:Cortex-M33, 2:Pico2:RISC-V
 - BOARDTYPE PICOとPICO_W

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
本実習では使用できない
2. USERスイッチ
3. LED1,2
4. DIPSW 1,2



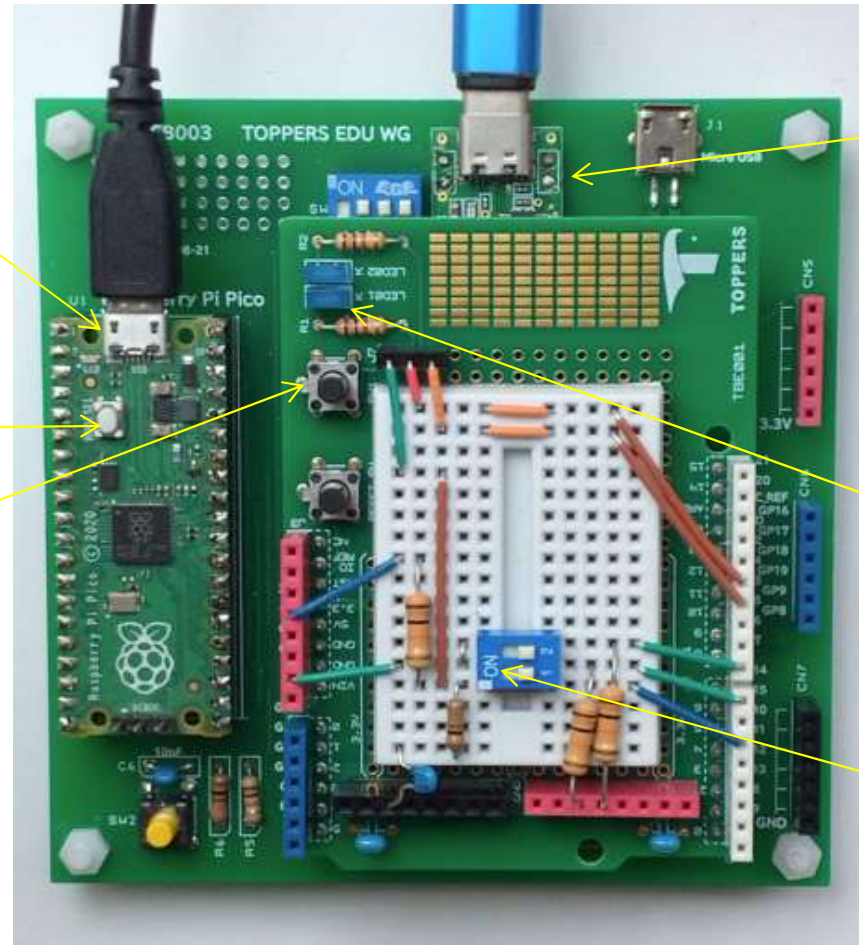
実習で使うハードウェア

- プロトタイピング・シールドを接続した形で実習を行う

①Pico USB
フラッシュROM
の書き込み
電源供給

②BOOT SW

③USERスイッチ
ポーリングと割込み
でスイッチの検知



④USBシリアル
パソコンと接続
してROMモニタ
表示

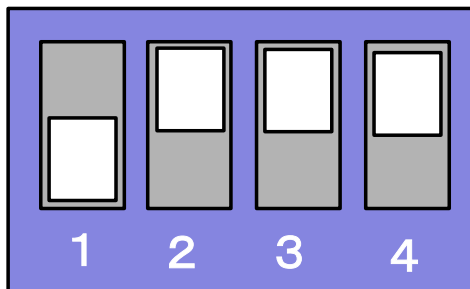
⑤LED
点灯、点滅、消灯
の実習

⑥DIPSW
タスク間通信

DIP-SWの設定

- TEB003のDIP-SWは以下の設定です
- DIP-SWの1は通常OFFですが、USBホストを実行する場合、Pico/Pico2のUSBのVBUSは給電となるため、DIP-SWの1をONにして、外部給電に切り替えます

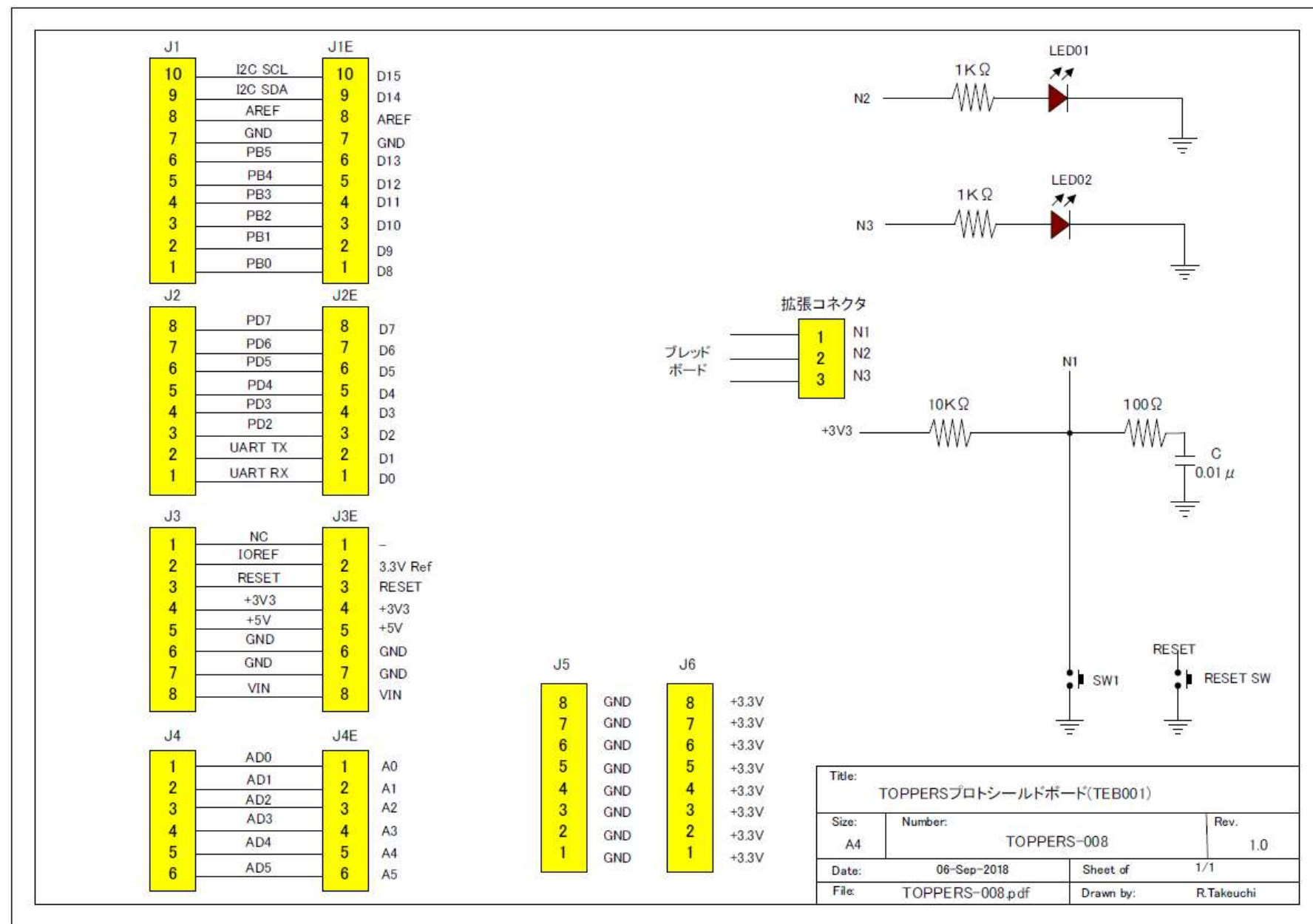
番号	ON/OFFの設定	有効機能
1	ONで外部電源をVBUSに接続	外部電源より給電
2	ONでUSBシリアルのRXをGP0に接続	USBシリアル有効化
3	ONでUSBシリアルをTXをGP1に接続	USBシリアル有効化
4	ONでプッシュスイッチをGP22に接続	プッシュスイッチ有効化



ON

OFF

プロトタイピング・シールド回路図



マイコンボードの確認

1. マイコンボード (Pico/Pico2+TEB003)
2. プロセッサ (RP2040:Coretex-M0+)
3. JTAGエミュレータ

プロセッサ (RP2040) : 概要

- RP2040はARM Cortex-M0+2コアとしたSoCで、ラズベリーパイ財団が独自に開発しました
- 製品型番: RP2040/SC0914
- パッケージ: 56pin QFNパッケージ
- CPU: デュアルコア ARM Cortex-M0+ @133MHz
- 内部RAM: 264KB SRAM
- Flashメモリ: なし、Quad SPIによる外付け
- I/O電圧: 3.3V
- 電源電圧: 3.3V

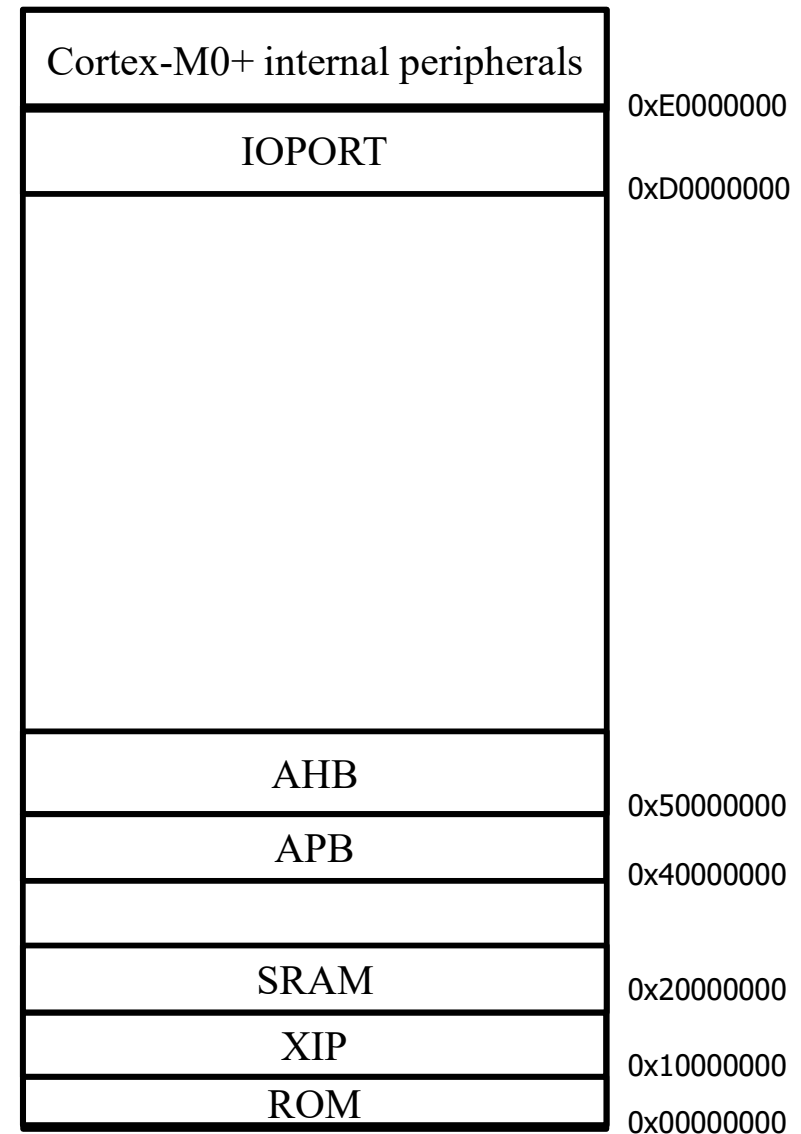


RP2040概要

- インターフェースとして以下を持ちます
 - 1 × USB 1.1 (デバイス・ホストモードに対応)
 - 30 × GPIO (内4本がアナログ対応)
 - 2 × UART
 - 2 × I2C
 - 2 × SPI
 - 16 × PWM (8 × A/Bチャネル、Bは入力対応)
 - 8 × PIO
 - 5 × ADC 12bit 500kサンプル/秒 (ひとつは内部温度センサーに接続)
 - 1 × タイマー (4 × アラート機能付き)
 - 1 × RTC
 - 1 × ウォッチドッグタイマー
 - 1 × 内部温度センサー
 - 1 × 3ピンARMシリアルワイヤデバッグ(SWD)ポート

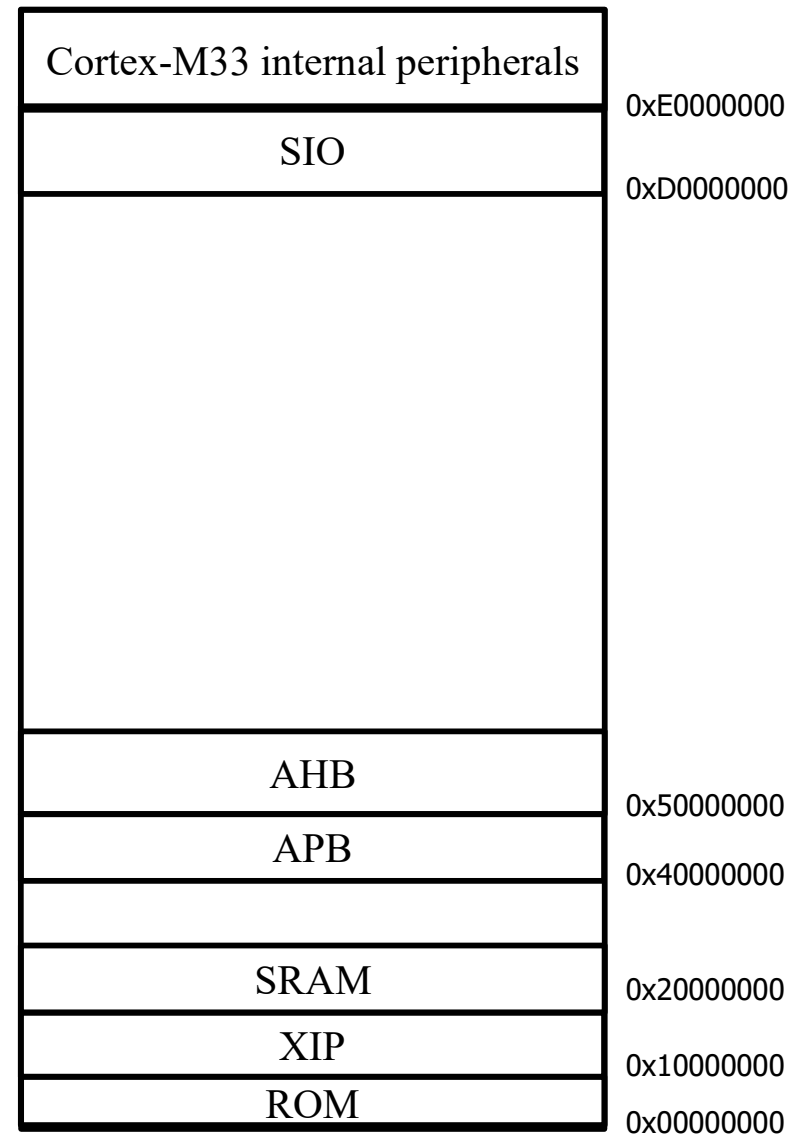
Pico: メモリマップ

- 2MBフラッシュROM領域にプログラムを置き、256KB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
- フラッシュROM領域にROMモニタプログラムを置き、SRAM中の0x2003F000～0x2003FFFFの4KBをROMモニタのデータ領域とし
- 0x20000000～0x2003EFFFをダウンロード領域とする



Pico2: メモリマップ

- 4MBフラッシュROM領域にプログラムを置き、512KB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
- フラッシュROM領域にROMモニタプログラムを置き、SRAM中の0x2007F000～0x2007FFFFの4KBをROMモニタのデータ領域とし
- 0x20000000～0x2007EFFFをダウンロード領域とする

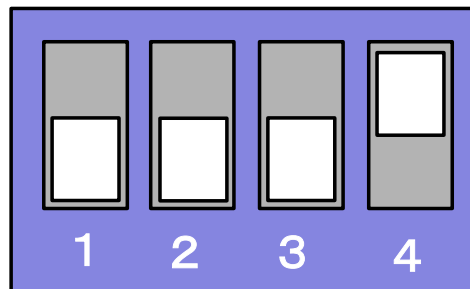


マイコンボードの確認

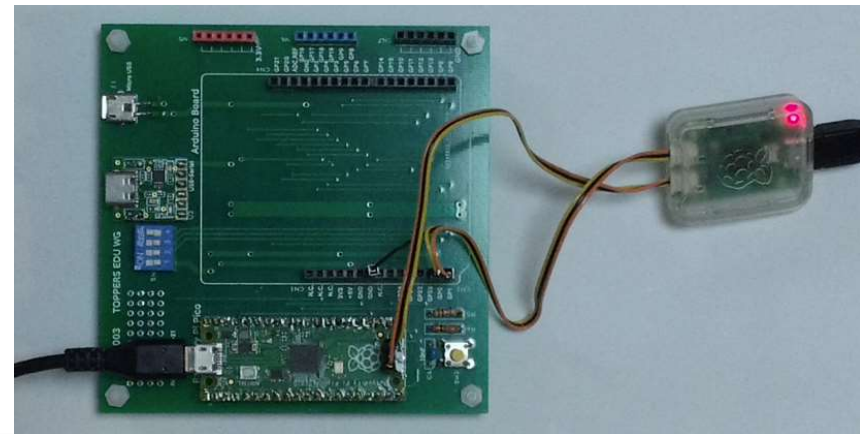
1. マイコンボード (Raspberry Pi Pico/Pico2)
2. プロセッサ (RP2040:Cortex-M0+)
3. JTAGエミュレータ

JTAGエミュレータ

- CPUコアと通信しながら、ボードやプログラムのデバッグを行うツールです
 - 通信にはJTAG(IEEE1149.1)規格を用います
 - CPU上にデバッグ機能を埋め込んだOCD(オンチップデバッグ)が主流でARMコアにもOCDが埋め込まれています
- Pico/Pico2用は「Raspberry Pi Debug Probe」があります
 - VS Codeで拡張機能: Raspberry Pi Picoを取り込むと簡単にボードデバッグができます
 - ツールのバックグラウンドではOpenOCDが動作
 - GDBなのでオプティマイズしたコードのデバッグは難しい



DIP-SWの2,3をOFF
A4-GP0,A5-GP1となる
ので、Debug
ProbeのUARTが使用
できる



PowerShellを使ってOpenOCDを動かす

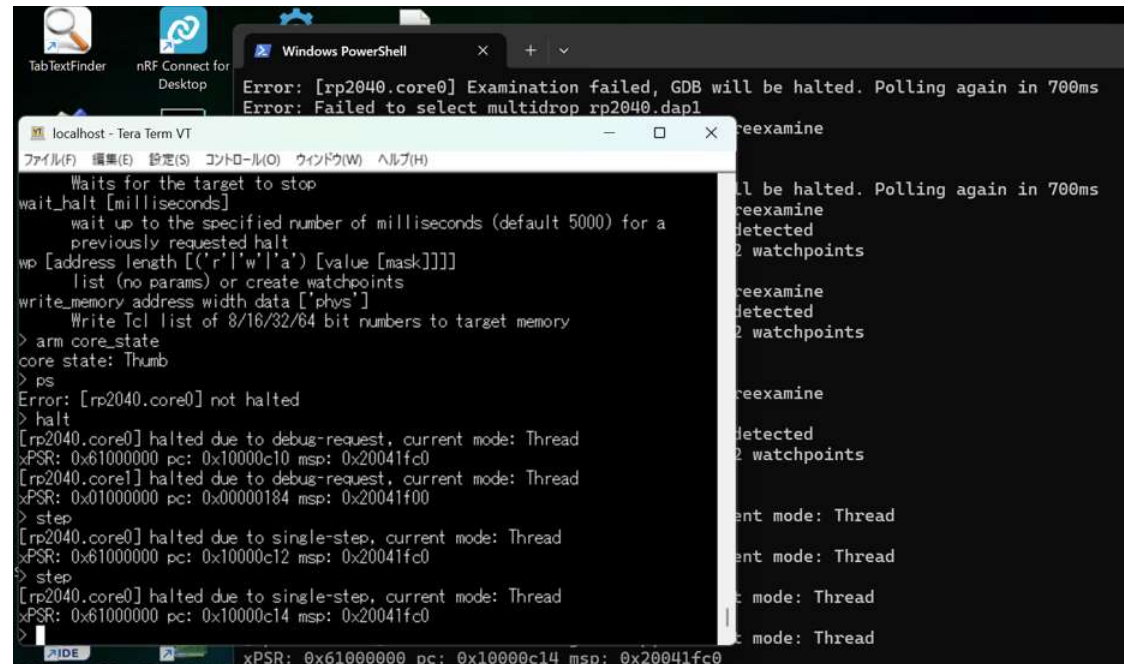
- WindowsのPowerShell上でOpenOCDを使って、elfデータを直接Picoのフラッシュメモリに書き込む
 - `openocd -f .\interface\cmsis-dap.cfg -f .\target\rp2040.cfg -c "adapter speed 5000" -c "program blink.elf verify reset exit"`(reteun)

```
shutdown error
PS C:\work\OpenOCD-20251211-0.12.0\share\openocd\scripts> ..\..\bin\openocd.exe -f .\interface\cmsis-dap.cfg -f .\target\rp2040.cfg -c "adapter speed 5000" -c "program blink.elf verify reset exit"
Open On-Chip Debugger 0.12.0 (2025-12-11) [https://github.com/sysprogs/openocd]
Licensed under GNU GPL v2
libusb1: d52e355daa09f17ce64819122cb067b8a2ee0d4b
For bug reports, read
  http://openocd.org/doc/doxygen/bugs.html
Info : [rp2040.core0] Hardware thread awareness created
Info : [rp2040.core1] Hardware thread awareness created
adapter speed: 5000 kHz
Info : Using CMSIS-DAPv2 interface with VID:PID=0x2e8a:0x000c, serial=E6647C740376762E
Info : CMSIS-DAP: SWD supported
Info : CMSIS-DAP: Atomic commands supported
Info : CMSIS-DAP: Test domain timer supported
Info : CMSIS-DAP: FW Version = 2.0.0
Info : CMSIS-DAP: Interface Initialised (SWD)
Info : SWCLK/TCK = 0 SWDIO/TMS = 0 TDI = 0 TDO = 0 nTRST = 0 nRESET = 0
Info : CMSIS-DAP: Interface ready
Info : clock speed 5000 kHz
Info : SWD DPIDR 0x0bc12477, DLPIDR 0x00000001
Info : SWD DPIDR 0x0bc12477, DLPIDR 0x10000001
Info : [rp2040.core0] Cortex-M0+ r0p1 processor detected
Info : [rp2040.core0] target has 4 breakpoints, 2 watchpoints
Info : [rp2040.core0] Examination succeed
Info : [rp2040.core1] Cortex-M0+ r0p1 processor detected
Info : [rp2040.core1] target has 4 breakpoints, 2 watchpoints
Info : [rp2040.core1] Examination succeed
Info : [rp2040.core0] starting gdb server on 3333
Info : Listening on port 3333 for gdb connections
[rp2040.core0] halted due to debug-request, current mode: Thread
xPSR: 0xf1000000 pc: 0x000000ea msp: 0x20041f00
```

PowerShellを使ってOpenOCDを動かす

- WindowsのPowerShell上でOpenOCDを起動
 - `openocd -f .\interface\cmsis-dap.cfg -f .\target\rp2040.cfg -c “adapter speed 5000”(reteun) で起動`
- TeraTermでtelnet localhost port:4444で立ち上げ、接続後、コマンドデバックする
 - `halt`で停止
 - `step`でステップ実行

これらは、VS Codeのデバッグのバックグラウンドで行われる通信と同様です



The screenshot shows two overlapping windows. The top window is a Windows PowerShell terminal with the following text:

```
Error: [rp2040.core0] Examination failed, GDB will be halted. Polling again in 700ms
Error: Failed to select multidrop rp2040.dap1
```

The bottom window is a TeraTerm VT terminal showing a telnet session with localhost:4444. The output is as follows:

```
localhost - Tera Term VT
ファイル(F) 編集(E) 設定(S) コントロール(O) ウィンドウ(W) ヘルプ(H)

Waits for the target to stop
wait_halt [milliseconds]
wait up to the specified number of milliseconds (default 5000) for a
previously requested halt
wp [address length [(('r'|'w'|'a')] [value [mask]])]
list (no params) or create watchpoints
write_memory address width data ['phys']
Write Tcl list of 8/16/32/64 bit numbers to target memory
> arm core_state
core state: Thumb
> ps
Error: [rp2040.core0] not halted
> halt
[rp2040.core0] halted due to debug-request, current mode: Thread
xPSR: 0x61000000 pc: 0x10000c10 msp: 0x20041fc0
[rp2040.core1] halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x00000184 msp: 0x20041f00
> step
[rp2040.core0] halted due to single-step, current mode: Thread
xPSR: 0x61000000 pc: 0x10000c12 msp: 0x20041fc0
> step
[rp2040.core0] halted due to single-step, current mode: Thread
xPSR: 0x61000000 pc: 0x10000c14 msp: 0x20041fc0
>
xPSR: 0x61000000 pc: 0x10000c14 msp: 0x20041fc0
```

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モニタを書き込む

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, Xyzzzy 等)
 - ターミナルソフトウェア (TeraTermPro 等)
 - PDFリーダー (Acrobat Reader 等)

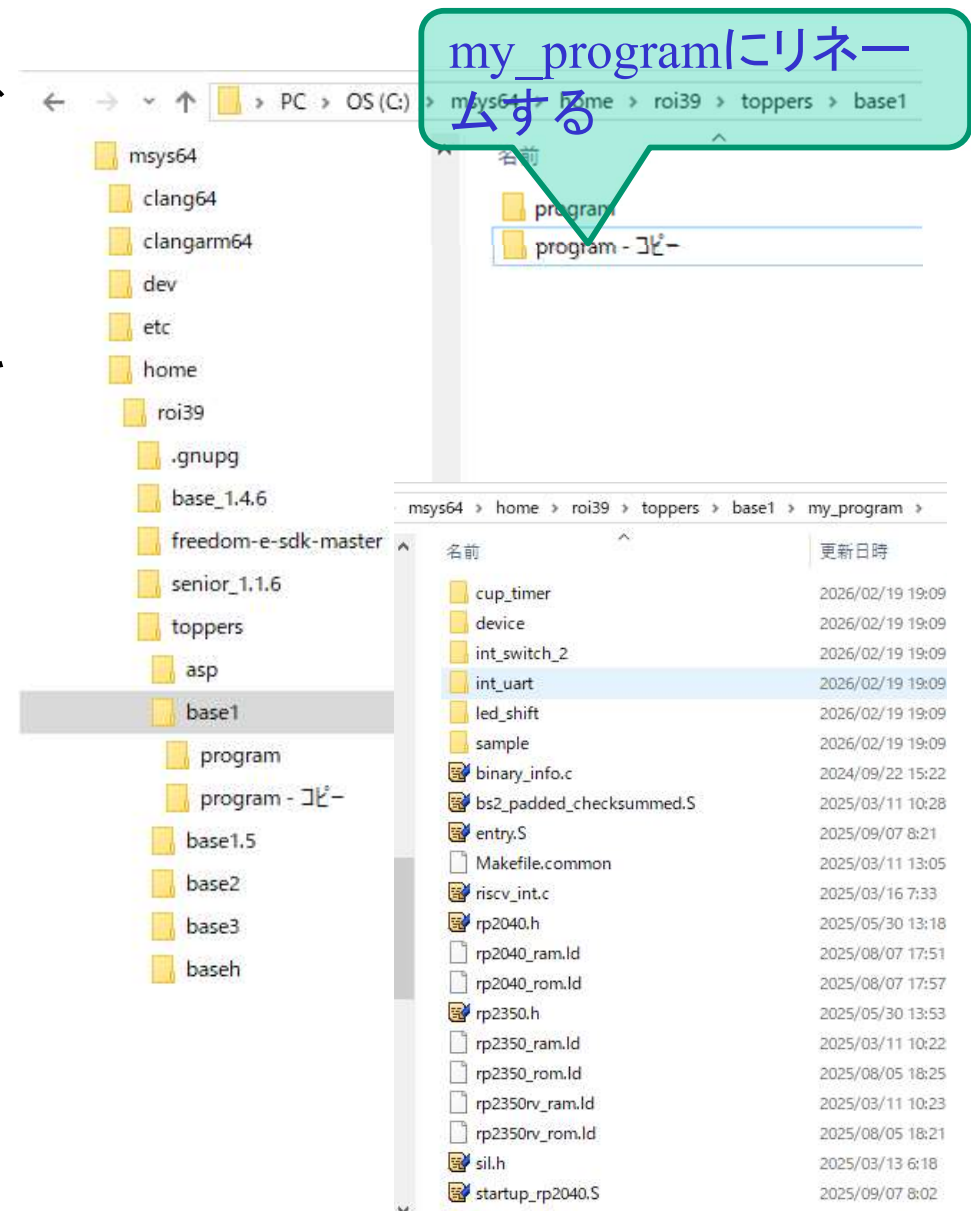
開発環境の確認

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モニタを書き込む

ワークディレクトリを作成する

- toppers/base1/programフォルダをbase1上にコピーして名前をmy_programに変更します
- 作成したmy_programが作業を行うディレクトリになります
- 2日目の実習ではこのディレクトリで作業するため、エクスプローラーのクイックアクセスに追加して、直ぐに移動できるようにします

セミナ受講の場合、このディレクトリは作成済です



補助プログラムの説明

- programディレクトリに、ビルドを補助するプログラムが置かれている
- Cortex-M0+/Cortex-M33/RISC-Vの対応のためプログラム数が多い
- 以下の4ファイルは、Pico/Pico2共通ファイル
 - sys_def.hとSysConfig.cは、BOARDNOでコアを判定

make BOARDNO=0	BOARDNOは省略可能、Picoをビルド
make BOARDNO=1	Pico2:Cortex-M33をビルド
make BOARDNO=2	Pico2:RISC-Vをビルド

ファイル	ファイルの内容
Makefile.common	ビルド環境設定用のMakefile用のインクルードファイル
sys_defs.h	システム定義のインクルードファイル
SysConfig.c	C言語記述のスタートアップルーチン
syslog.c	syslog表示プログラム
t_syslog.h	syslog表示用のインクルードファイル
sil.h	SIL記述用のインクルードファイル

補助プログラムの説明

- Pico:Cortex-M0+に依存するファイルは以下の通り
- make時のオプション設定が面倒な場合、Makefile.commonにBOARDNO設定を追加することで変更が可能

ファイル	ファイルの内容
startup_rp2040.S	Pico2:Cortex-M0+用アセンブラ記述のスタートアップルーチン
bs2_padding_check summed.S	Picoのパディング・データのアセンブラ・ソース
rp2040.h	PicoのマップドIO定義ファイル
rp2040_ram.ld	Pico:Cortex-M0+用RAMリンク用リンクスクリプト・ファイル
rp2040_rom.ld	Pico:Cortex-M0+用ROMリンク用リンクスクリプト・ファイル

```
5 #  
6 # ボード番号の設定  
7 #  
8 ifeq ($(BOARDNO),)  
9 BOARDNO = 1  
10 endif
```

この行を追加
Pico2:Cortex-M33に設定

補助プログラムの説明

- Pico2に依存するファイルは以下の通り
- Cortex-M33とRISC-Vでリンクスクリプトが異なる
- entry.Sとriscv_int.cはRISC-V専用のファイル
 - Cortex-M33がハードウェアの部分も、RISC-Vではプログラムが必要となる

ファイル	ファイルの内容
startup_rp2350.S	Pico2:Cortex-M33用アセンブラ記述のスタートアップルーチン
startup_rp2350rv.S	Pico2:RISC-V用アセンブラ記述のスタートアップルーチン
rp2350.h	Pico2のマップドIO定義ファイル
rp2350_ram.ld	Pico2:Cortex-M33用RAMリンク用リンクスクリプト・ファイル
rp2350_rom.ld	Pico2:Cortex-M33用ROMリンク用リンクスクリプト・ファイル
entry.S	Pico2:RISC-V専用、アセンブラ記述の割込み出入り口処理
riscv_int.c	Pico2:RISC-V専用、割込み機能用の関数
rp2350rv_ram.ld	Pico2:RISC-V用RAMリンク用リンクスクリプト・ファイル
rp2350rv_rom.ld	Pico2:RISC-V用ROMリンク用リンクスクリプト・ファイル

最低限覚えておきたい、シェルコマンド

- ターミナルで使用するシェルコマンドで、常時使用するものは、以下の通です

コマンド	機能
ls	ディレクトリ中のファイルを表示する
cd	ディレクトリを変更する。引数はパス名、元のディレクトリに戻るには、パス名”..”を指定 引数なしで、ホームディレクトリに戻る
diff	ファイルまたはディレクトリ内のファイルをテキスト形式で比較し、差異を表示する
history	今まで使用したコマンドの一覧を表示する
↑ ↓	↑ひとつ前に発行したコマンドを表示 ↓次のコマンドを表示
help	使用可能なデフォルトコマンドを表示する
mkdir	ディレクトリを作成する
make	Makefileを実行する。ビルドを行う

bin2uf2コマンドの作成

- ビルドした実行イメージをuf2ファイルに変換することにより、Pico/Pico2のフラッシュROMに書き込みが可能になる
- 変換はELF(exe)をbinファイルを経由してuf2ファイルに変換
 - rommon.exe→rommon.bin
 - rommon.bin→rommon.uf2
- uf2ファイルは、PICO専用フォーマットなので、専用の変換コマンド(bin2uf2)が必要となる
- bin2uf2コマンドのソースはasp/tools/bin2uf2に用意している
- bin2uf2に移動して、makeコマンドで、bin2uf2.exeをビルドする

세미나受講の場合、この作業は終了しています

```
~/toppers/fmp_1.4.0/tools/bin2uf2
-rw-r--r-- 1 roi39 roi39 5228 Mar 15 11:55 sample1.h
roi39@DESKTOP-JMDVGP MINGW64 ~/toppers/fmp_1.4.0/OBJ/RASPBERRYPI_PICO_GCC/SAMPL
E1
$ cd ../../..

roi39@DESKTOP-JMDVGP MINGW64 ~/toppers/fmp_1.4.0
$ cd tools/bin2uf2/

roi39@DESKTOP-JMDVGP MINGW64 ~/toppers/fmp_1.4.0/tools/bin2uf2
$ make
gcc -O3 -Wall bin2uf2.c -I. -o bin2uf2

roi39@DESKTOP-JMDVGP MINGW64 ~/toppers/fmp_1.4.0/tools/bin2uf2
$ ls -la
total 269
drwxr-xr-x 1 roi39 roi39 0 Apr 19 11:03 .
drwxr-xr-x 1 roi39 roi39 0 Feb 13 11:21 ..
-rw-r--r-- 1 roi39 roi39 323 Sep 11 2022 Makefile
-rw-r--r-- 1 roi39 roi39 5160 Sep 19 2022 bin2uf2.c
-rwxr-xr-x 1 roi39 roi39 262483 Apr 19 11:03 bin2uf2.exe

roi39@DESKTOP-JMDVGP MINGW64 ~/toppers/fmp_1.4.0/tools/bin2uf2
$ |
```

bin2uf2コマンドの作成

- ビルドしたbin2uf2コマンドを、B-shellから実行するために、ユーザーの実行コマンドを設定するディレクトリに移動する
 - /msys64/usr/local/bin
- binディレクトリは作られていないので、エクスプローラでlocalの下にbinディレクトリを作成し、ビルドしたbin2uf2.exeをコピーする

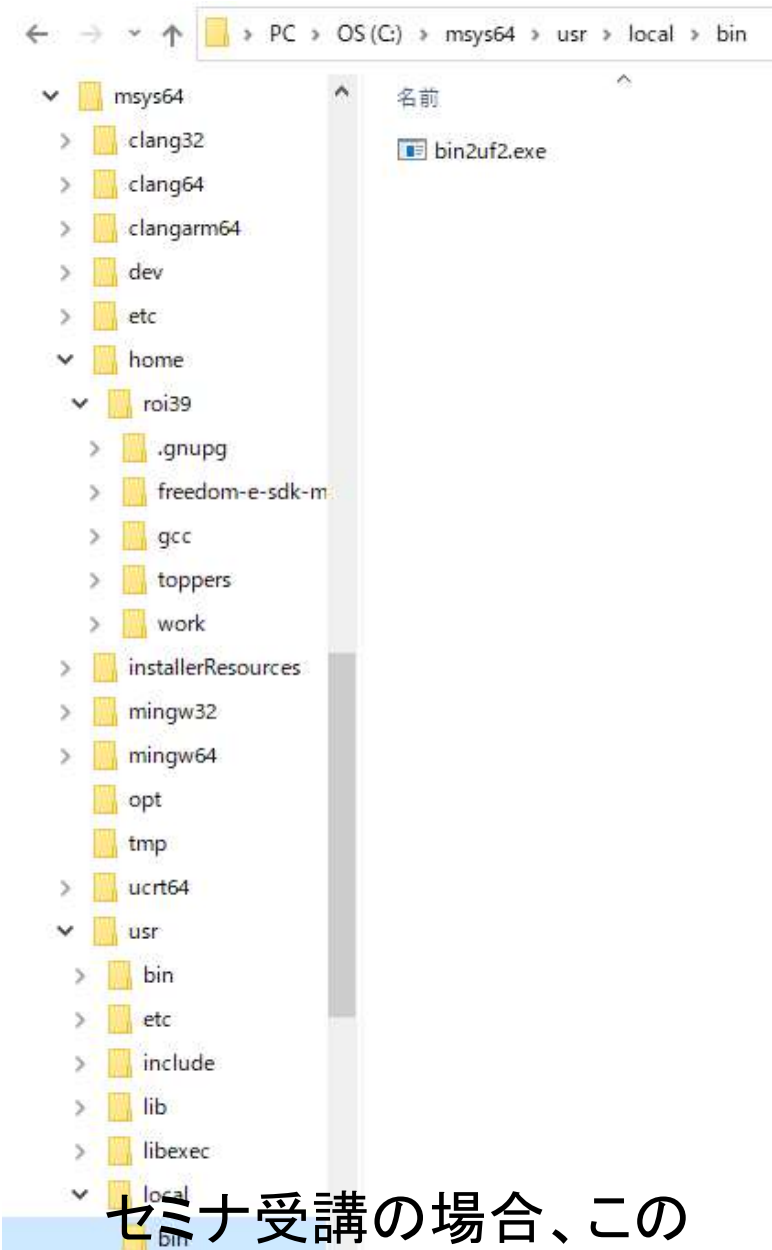
```
~/toppers/fmp_1.4.0/tools
roi39@DESKTOP-JMDVGP MINGW64 ~/toppers/fmp_1.4.0/tools/bin2uf2
$ make
gcc -O3 -Wall bin2uf2.c -I. -o bin2uf2

roi39@DESKTOP-JMDVGP MINGW64 ~/toppers/fmp_1.4.0/tools/bin2uf2
$ ls -la
total 269
drwxr-xr-x 1 roi39 roi39  0 Apr 19 11:03 .
drwxr-xr-x 1 roi39 roi39  0 Feb 13 11:21 ..
-rw-r--r-- 1 roi39 roi39 323 Sep 11 2022 Makefile
-rw-r--r-- 1 roi39 roi39 5160 Sep 19 2022 bin2uf2.c
-rwxr-xr-x 1 roi39 roi39 262483 Apr 19 11:03 bin2uf2.exe

roi39@DESKTOP-JMDVGP MINGW64 ~/toppers/fmp_1.4.0/tools/bin2uf2
$ cd ..

roi39@DESKTOP-JMDVGP MINGW64 ~/toppers/fmp_1.4.0/tools
$ bin2uf2
usage: C:\msys64\usr\local\bin\bin2uf2.exe inputfile [-options]
        <-addr>                write address

roi39@DESKTOP-JMDVGP MINGW64 ~/toppers/fmp_1.4.0/tools
$ |
```



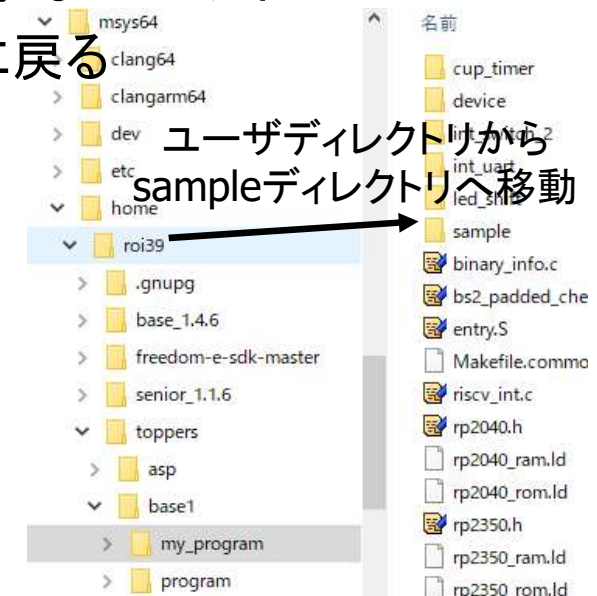
세미나受講の場合、この作業は終了しています

MSYS2でのワークディレクトリへの移動

- MSYS2を起動すると、ユーザのベース・ディレクトリ上に設定されます
- ビルド作業を行うために、ワークディレクトリに移動する必要があります
- ターミナル(MSYS2)では、ディレクトリの移動やファイル名表示などの処理はコマンドを使って行います
 - ディレクトリの移動はcdコマンド、ファイルの表示はlsコマンド
 - cd コマンドで..を指定すると、元のディレクトリに戻る

```
$ cd toppers/base1/my_program/led_shift(enter)
$ ls(enter)
  Makefile led_shift
$ cd ../sample(enter)
$ ls(enter)
  Makefile sample
```

roi39→toppers→my_program→led_shift
→my_program→sample



RAMモードのビルド

- MSYS2を起動して、sampleディレクトリに移動する
- make DBGENV=ROM(enter) でROM化可能な sample.uf2ファイルをビルドできる

binファイルをuf2
ファイルに変換する
ためにbin2uf2コマ
ンドが必要

```
~/toppers/base1/program/sample
EXEC=1 -DDEBUG -I.. -I. ../startup_rp2040.S
arm-none-eabi-gcc -c -mcpu=cortex-m0plus -Wa,--gstabs -mthumb -mthumb-interwork
-mlittle-endian -nostdlib -O2 -g -Wall -DBOARDNO=0 -DRP2040_CORE1_RESET -DROM_
EXEC=1 -DDEBUG -I.. -I. ../bs2_padded_checksummed.S
arm-none-eabi-gcc -mcpu=cortex-m0plus -Wa,--gstabs -mthumb -mthumb-interw
ork -mlittle-endian -nostdlib -O2 -g -Wall -DBOARDNO=0 -DRP2040_CORE1_RESET -DR
OM_EXEC=1 -DDEBUG -I.. -I. -nostdlib -T ../rp2040_rom.ld -o sample.exe \
startup_rp2040.o bs2_padded_checksummed.o sample.o SysConfig.o
syslog.o -lc -lgcc
arm-none-eabi-nm sample.exe > sample.syms
arm-none-eabi-objcopy -O srec -S sample.exe sample.srec
arm-none-eabi-objcopy -O binary -S sample.exe sample.bin
bin2uf2 sample.bin -family 0
arm-none-eabi-objdump -t -h sample.exe > sample.map
arm-none-eabi-objdump -D sample.exe > sample.lst

roi39@DESKTOP-JMDVGVP MINGW64 ~/toppers/base1/program/sample
$ ls
Makefile          sample.bin  sample.lst  sample.srec  startup_rp2040.o
SysConfig.o       sample.c   sample.map  sample.syms  syslog.o
bs2_padded_checksummed.o sample.exe  sample.o    sample.uf2

roi39@DESKTOP-JMDVGVP MINGW64 ~/toppers/base1/program/sample
$
```

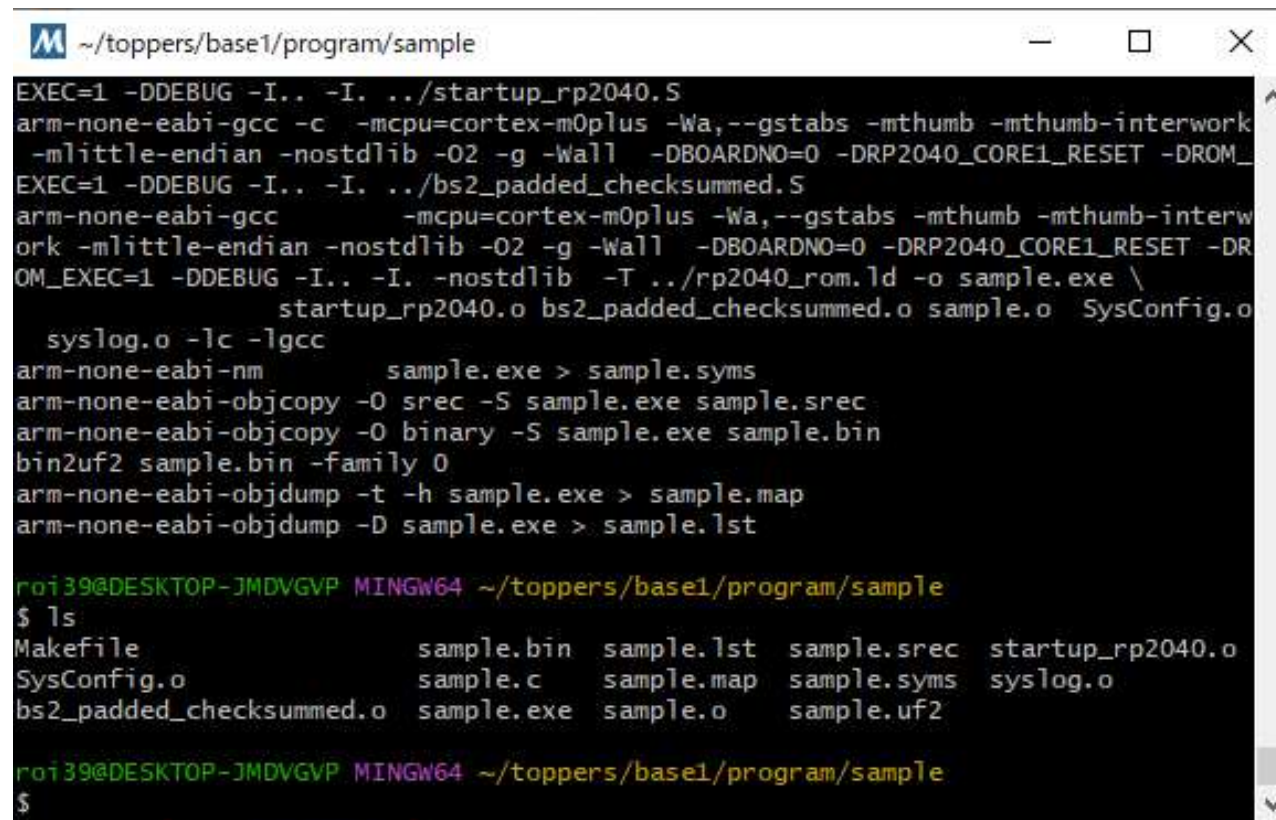
ビルド:ファイルの確認

- Picoでビルド後、12ファイルが生成される
- ダウンロード用はSRECファイル、ROM書き込み用にはUF2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_rp2040.o	アセンブラのスタートアップルーチンから生成された中間コードファイル
SysConfig.o	C言語スタートアップルーチンから生成された中間コード
syslog.o	sys_printf文、syslog文を実行するプログラム
bs2_padding_checksummed.o	PicoのフラッシュROMヘッダー部につけるパディング・データの間コード
sample.bin	sampleプログラムバイナリ・イメージファイル
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
sample.uf2	sampleプログラムフラッシュメモリ書き込み用

ROMモードのビルド

- make clean(enter)
- make DEBUG=1 DBGENV=ROM(enter) でROM化可能なsample.uf2ファイルをビルドできる



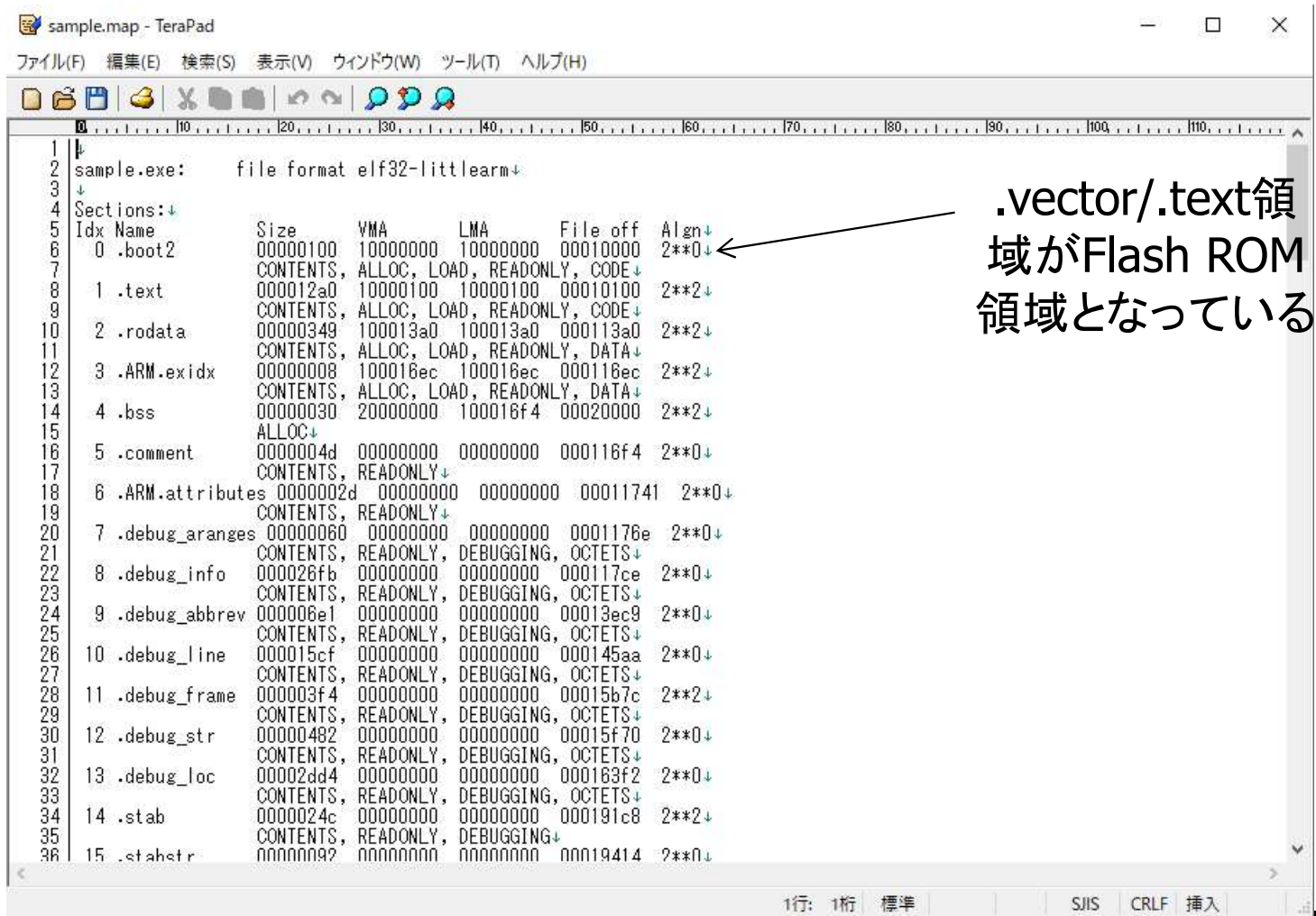
```
~/toppers/base1/program/sample
EXEC=1 -DDEBUG -I.. -I. ../startup_rp2040.S
arm-none-eabi-gcc -c -mcpu=cortex-m0plus -Wa,--gstabs -mthumb -mthumb-interwork
-mlittle-endian -nostdlib -O2 -g -Wall -DBOARDNO=0 -DRP2040_CORE1_RESET -DROM_
EXEC=1 -DDEBUG -I.. -I. ../bs2_padded_checksummed.S
arm-none-eabi-gcc -mcpu=cortex-m0plus -Wa,--gstabs -mthumb -mthumb-interw
ork -mlittle-endian -nostdlib -O2 -g -Wall -DBOARDNO=0 -DRP2040_CORE1_RESET -DR
OM_EXEC=1 -DDEBUG -I.. -I. -nostdlib -T ../rp2040_rom.ld -o sample.exe \
startup_rp2040.o bs2_padded_checksummed.o sample.o SysConfig.o
syslog.o -lc -lgcc
arm-none-eabi-nm sample.exe > sample.syms
arm-none-eabi-objcopy -O srec -S sample.exe sample.srec
arm-none-eabi-objcopy -O binary -S sample.exe sample.bin
bin2uf2 sample.bin -family 0
arm-none-eabi-objdump -t -h sample.exe > sample.map
arm-none-eabi-objdump -D sample.exe > sample.lst

roi39@DESKTOP-JMDVGVP MINGW64 ~/toppers/base1/program/sample
$ ls
Makefile          sample.bin  sample.lst  sample.srec  startup_rp2040.o
SysConfig.o       sample.c   sample.map  sample.syms  syslog.o
bs2_padded_checksummed.o sample.exe  sample.o    sample.uf2

roi39@DESKTOP-JMDVGVP MINGW64 ~/toppers/base1/program/sample
$
```

ROM化の確認

- sample.mapをエディタで開いて、プログラム(.text)がROM領域(0x10000000)となっていることを確認する



sample.map - TeraPad

ファイル(F) 編集(E) 検索(S) 表示(V) ウィンドウ(W) ツール(T) ヘルプ(H)

```
1 | sample.exe: file format elf32-littlearm
2 |
3 |
4 | Sections:
5 | Idx Name      Size      VMA      LMA      File off  Algn
6 | 0 .boot2      00000100 10000000 10000000 00010000 2**0
7 |   CONTENTS, ALLOC, LOAD, READONLY, CODE
8 | 1 .text       000012a0 10000100 10000100 00010100 2**2
9 |   CONTENTS, ALLOC, LOAD, READONLY, CODE
10 | 2 .rodata     00000349 100013a0 100013a0 000113a0 2**2
11 |   CONTENTS, ALLOC, LOAD, READONLY, DATA
12 | 3 .ARM.exidx   00000008 100018ec 100018ec 000118ec 2**2
13 |   CONTENTS, ALLOC, LOAD, READONLY, DATA
14 | 4 .bss        00000030 20000000 100016f4 00020000 2**2
15 |   ALLOC
16 | 5 .comment     0000004d 00000000 00000000 000116f4 2**0
17 |   CONTENTS, READONLY
18 | 6 .ARM.attributes 0000002d 00000000 00000000 00011741 2**0
19 |   CONTENTS, READONLY
20 | 7 .debug_aranges 00000080 00000000 00000000 0001176e 2**0
21 |   CONTENTS, READONLY, DEBUGGING, OCTETS
22 | 8 .debug_info  000026fb 00000000 00000000 000117ce 2**0
23 |   CONTENTS, READONLY, DEBUGGING, OCTETS
24 | 9 .debug_abbrev 000008e1 00000000 00000000 00013ec9 2**0
25 |   CONTENTS, READONLY, DEBUGGING, OCTETS
26 | 10 .debug_line  000015cf 00000000 00000000 000145aa 2**0
27 |   CONTENTS, READONLY, DEBUGGING, OCTETS
28 | 11 .debug_frame 000003f4 00000000 00000000 00015b7c 2**2
29 |   CONTENTS, READONLY, DEBUGGING, OCTETS
30 | 12 .debug_str    00000482 00000000 00000000 00015f70 2**0
31 |   CONTENTS, READONLY, DEBUGGING, OCTETS
32 | 13 .debug_loc    00002dd4 00000000 00000000 000163f2 2**0
33 |   CONTENTS, READONLY, DEBUGGING, OCTETS
34 | 14 .stab         0000024c 00000000 00000000 000191c8 2**2
35 |   CONTENTS, READONLY, DEBUGGING
36 | 15 .stabstr      00000000 00000000 00000000 00019414 2**0
```

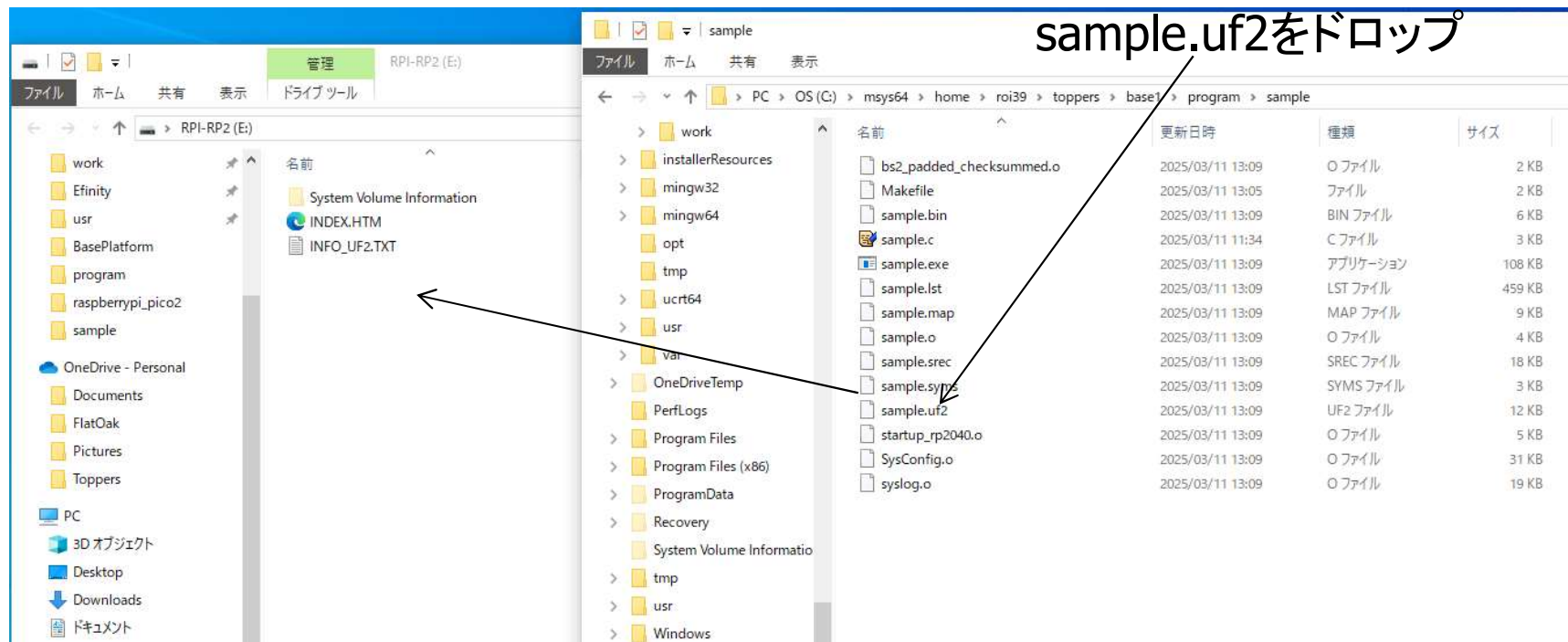
.vector/.text領域がFlash ROM領域となっている

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モニタを書き込む

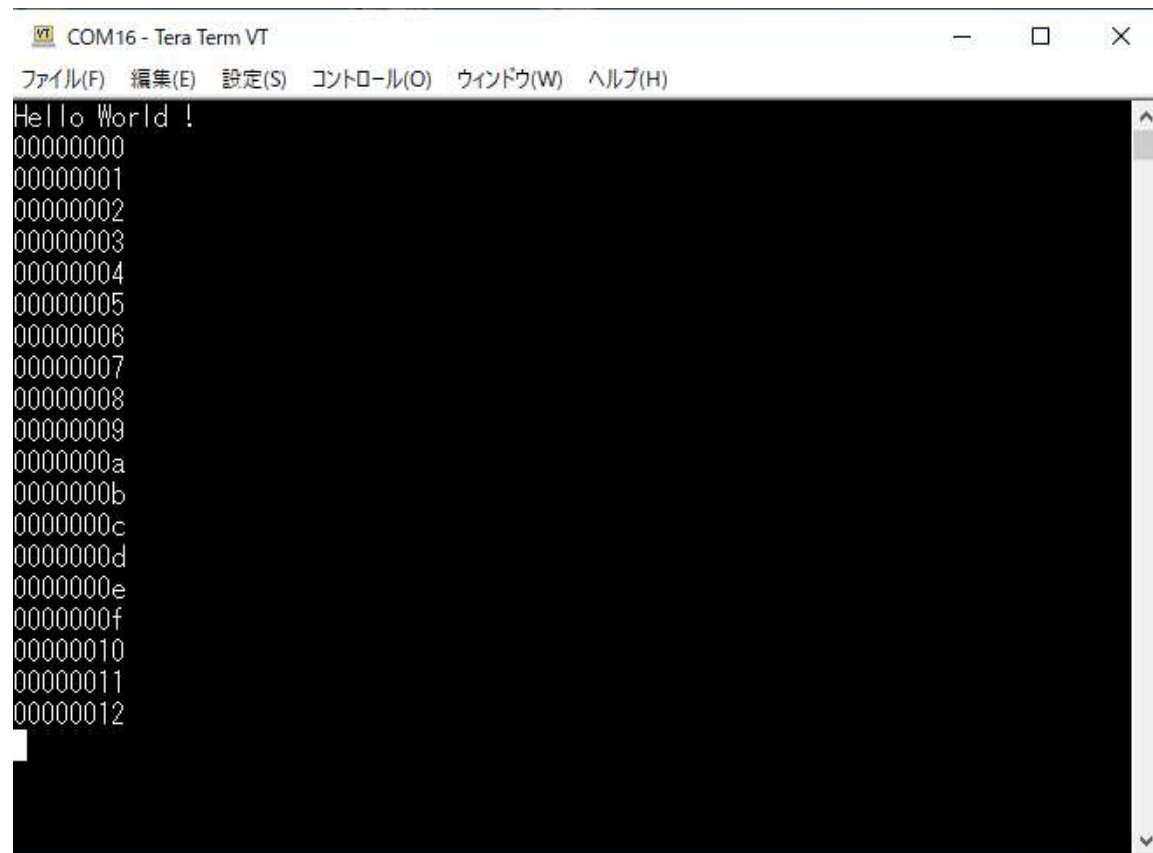
sampleをフラッシュROMに書き込む

- BOOT-SWをオンしながら、Picoをパソコンに接続
- フォルダが表示されるので、sample.uf2をフォルダにドロップすると、フラッシュROMに書込まれる



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、
USBケーブルを
指し直す
- カウンタ表示と
LEDが点滅



```
COM16 - Tera Term VT
ファイル(F) 編集(E) 設定(S) コントロール(O) ウィンドウ(W) ヘルプ(H)
Hello World !
00000000
00000001
00000002
00000003
00000004
00000005
00000006
00000007
00000008
00000009
0000000a
0000000b
0000000c
0000000d
0000000e
0000000f
00000010
00000011
00000012
```

開発環境の確認

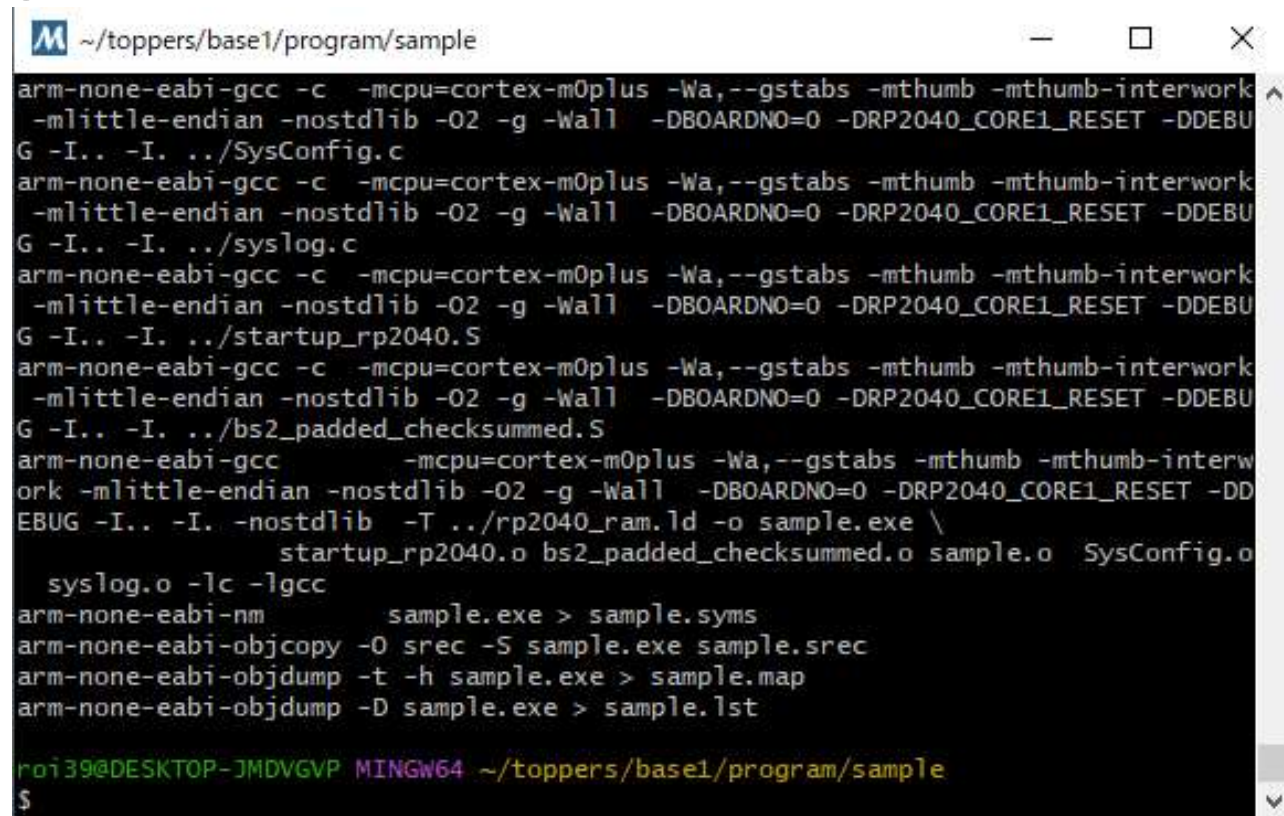
1. ハードウェア・ソフトウェア環境の構築
2. サンプルプログラムのビルドと実行
3. フラッシュメモリへの書き込み
4. [ROMモニタを書き込む](#)

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



```
~/toppers/base1/program/sample
arm-none-eabi-gcc -c -mcpu=cortex-m0plus -Wa,--gstabs -mthumb -mthumb-interwork
-mlittle-endian -nostdlib -O2 -g -Wall -DBOARDNO=0 -DRP2040_CORE1_RESET -DDEBU
G -I.. -I. ../SysConfig.c
arm-none-eabi-gcc -c -mcpu=cortex-m0plus -Wa,--gstabs -mthumb -mthumb-interwork
-mlittle-endian -nostdlib -O2 -g -Wall -DBOARDNO=0 -DRP2040_CORE1_RESET -DDEBU
G -I.. -I. ../syslog.c
arm-none-eabi-gcc -c -mcpu=cortex-m0plus -Wa,--gstabs -mthumb -mthumb-interwork
-mlittle-endian -nostdlib -O2 -g -Wall -DBOARDNO=0 -DRP2040_CORE1_RESET -DDEBU
G -I.. -I. ../startup_rp2040.S
arm-none-eabi-gcc -c -mcpu=cortex-m0plus -Wa,--gstabs -mthumb -mthumb-interwork
-mlittle-endian -nostdlib -O2 -g -Wall -DBOARDNO=0 -DRP2040_CORE1_RESET -DDEBU
G -I.. -I. ../bs2_padded_checksummed.S
arm-none-eabi-gcc -mcpu=cortex-m0plus -Wa,--gstabs -mthumb -mthumb-interw
ork -mlittle-endian -nostdlib -O2 -g -Wall -DBOARDNO=0 -DRP2040_CORE1_RESET -DD
EBUG -I.. -I. -nostdlib -T ../rp2040_ram.ld -o sample.exe \
startup_rp2040.o bs2_padded_checksummed.o sample.o SysConfig.o
syslog.o -lc -lgcc
arm-none-eabi-nm sample.exe > sample.syms
arm-none-eabi-objcopy -O srec -S sample.exe sample.srec
arm-none-eabi-objdump -t -h sample.exe > sample.map
arm-none-eabi-objdump -D sample.exe > sample.lst

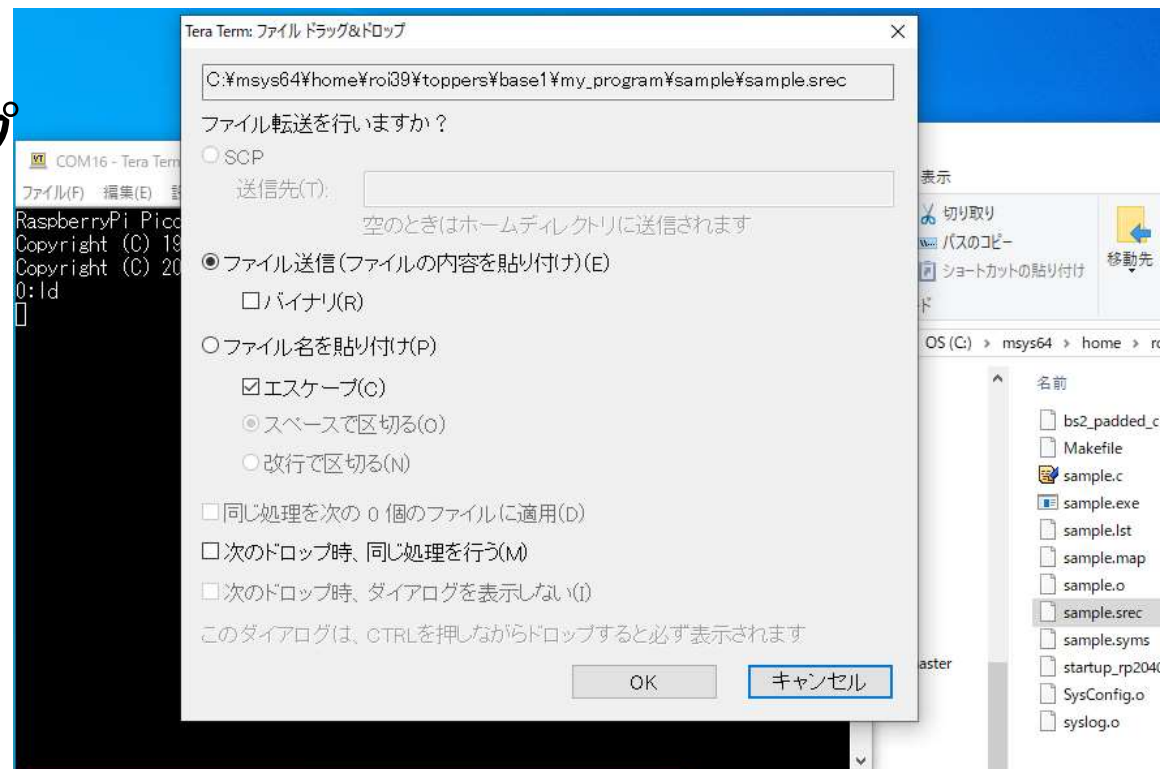
roi39@DESKTOP-JMDVGV MINGW64 ~/toppers/base1/program/sample
$
```

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、「Pico/Pico2に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

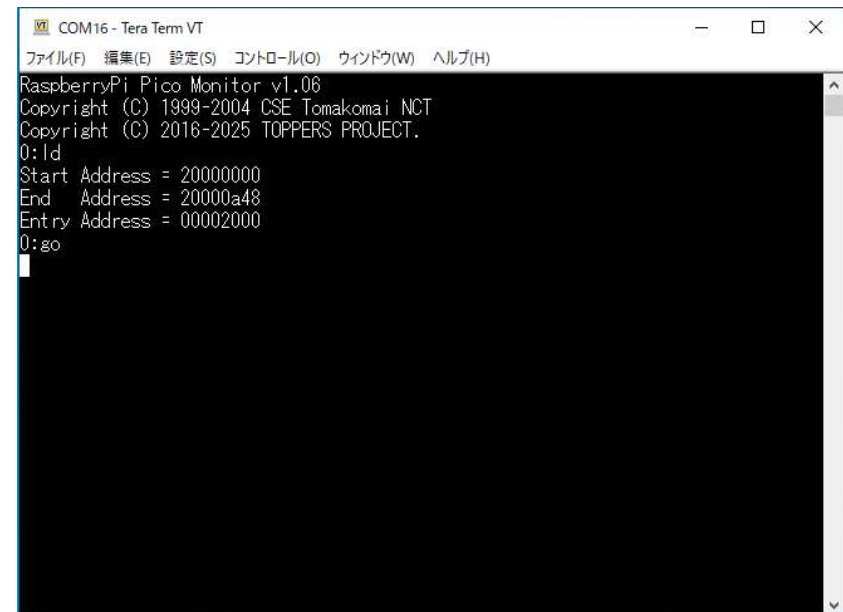
実行:ダウンロード開始、実行開始

- ROMモニタを書き込むと、TeraTermのROMモニタのプロントが表示される
- TeraTermのROMモニタをダウンロードモードに設定
 - ld(return)
- sample.srecを
TeraTermにドロップ
ファイル転送で
OKをクリック
- ダウンロード開始



実行:ダウンロード開始、実行開始

- ダウンロードが終了すると、ROMモニタにプロンプトが出るので、goコマンドで実行する
 - go(return)
- LEDは点滅するがカウンタを表示しない
- カウント表示したい場合はビルドでデバッグ指定 (DEBUG=1)が必要
- デバッグ指定でビルドし直す
- make clean(enter)
- make DEBUG=1(enter)

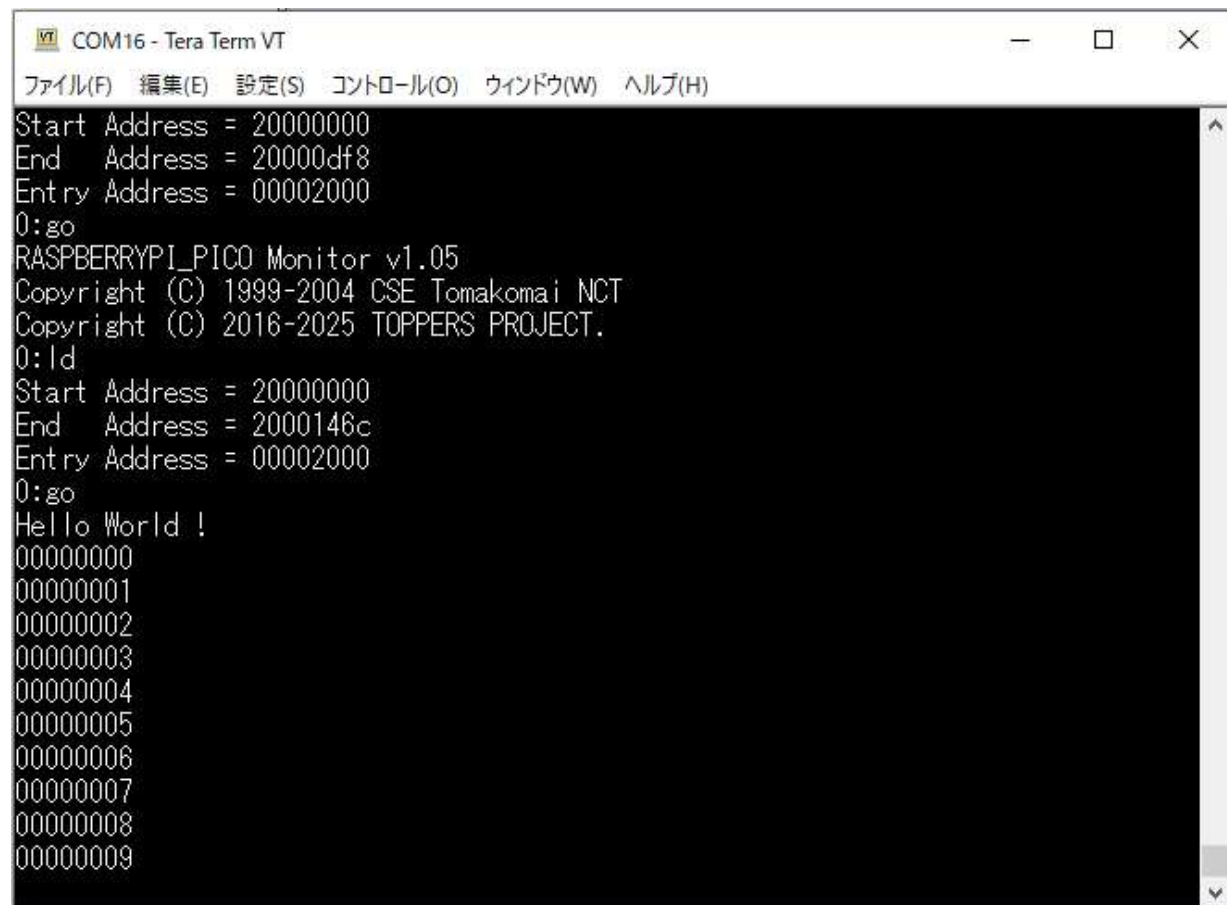


```
COM16 - Tera Term VT
ファイル(F) 編集(E) 設定(S) コントロール(O) ウィンドウ(W) ヘルプ(H)
RaspberryPi Pico Monitor v1.06
Copyright (C) 1999-2004 CSE Tomakomai NCT
Copyright (C) 2016-2025 TOPPERS PROJECT.
0:ld
Start Address = 20000000
End Address = 20000a48
Entry Address = 00002000
0:go
█
```

実行:ダウンロード開始、実行開始

- 再度、sample.srecをダウンロードする
- 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



```
COM16 - Tera Term VT
ファイル(F) 編集(E) 設定(S) コントロール(O) ウィンドウ(W) ヘルプ(H)
Start Address = 20000000
End Address = 20000df8
Entry Address = 00002000
0:go
RASPBERRYPI_PICO Monitor v1.05
Copyright (C) 1999-2004 CSE Tomakomai NCT
Copyright (C) 2016-2025 TOPPERS PROJECT.
0:ld
Start Address = 20000000
End Address = 2000146c
Entry Address = 00002000
0:go
Hello World !
00000000
00000001
00000002
00000003
00000004
00000005
00000006
00000007
00000008
00000009
```

デバッグ機能の確認

- sample.cを確認する
- main関数にて、全ての機能を記載する
 - 初期化とLOOPでの形 (Arduinoのsetupとloop)
 - 複雑な状態は、状態遷移図、状態遷移表が有効

```
int
main(){
    int count = 0;

    sys_printf("Hello World !¥n");
    for (;;) {
        sys_printf("%08x¥n", count);
        led_out_macro2(count << PINPOSITION3);
        count++;
        busy_wait();
    }
    return 0;
}
```

引数の文字列と引数を表示

LEDの点灯、消灯を操作

デバッグ機能の確認

- Makefileを確認する
- 必要なソースはMakefile.commonで設定する
- 45行目、48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

```
41 AOBJS = $(LOCAL_AOBJS)
42
43 $(DBGENV),ROM
44 CDEFS := $(CDEFS) -DROM_EXEC=1
45 COBJS := $(COBJS) syslog.o
46 else
47 ifeq ($(DEBUG),1)
48 COBJS := $(COBJS) syslog.o
49 endif
50 endif
```

オプション指定によるUART表示

- Makefileの53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

```
52 ifeq ($(DEBUG),1)
53 CDEFS *= $(CDEFS) -DDEBUG
54 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);
```

ROMモニタを使った実習

いろいろな開発環境

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

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



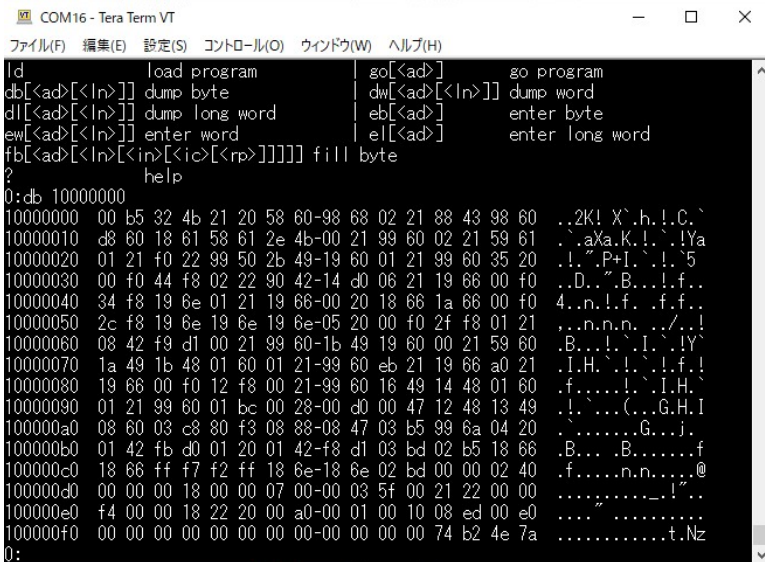
スタートアップルーチンの例

- startup_rp2040.SがPico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_rp2040.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

```
/* Call the clock system initialization function.*/  
bl SystemInit  
/* Call the application's entry point.*/  
bl main  
  
LoopForever:  
b LoopForever
```

ROMモニタから起動する

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



```
COM16 - Tera Term VT
ファイル(F) 編集(E) 設定(S) コントロール(O) ウィンドウ(W) ヘルプ(H)

ld[<ad>] load program      go[<ad>] go program
db[<ad>] dump byte        dw[<ad>] dump word
dl[<ad>] dump long word   eb[<ad>] enter byte
ew[<ad>] enter word       el[<ad>] enter long word
fb[<ad>] fill byte
? help

0:db 10000000
10000000 00 b5 32 4b 21 20 58 60-98 68 02 21 88 43 98 60 ..2K! X`.h.!C.`
10000010 d8 60 18 61 58 61 2e 4b-00 21 99 60 02 21 59 61 ..aXa.K.!..!Ya
10000020 01 21 f0 22 99 50 2b 49-19 60 01 21 99 60 35 20 !.P+I.!..!5
10000030 00 f0 44 f8 02 22 90 42-14 d0 06 21 19 66 00 f0 ..D..B...!f..
10000040 34 f8 19 6e 01 21 19 66-00 20 18 66 1a 66 00 f0 4..n.!f..f.f..
10000050 2c f8 19 6e 19 6e 19 6e-05 20 00 f0 2f f8 01 21 ...n.n.n.../..!
10000060 08 42 f9 d1 00 21 99 60-1b 49 19 60 00 21 59 60 .B...!..!Y
10000070 1a 49 1b 48 01 60 01 21-99 60 eb 21 19 66 a0 21 .I.H...!..f.!
10000080 19 66 00 f0 12 f8 00 21-99 60 18 49 14 48 01 60 .f.....!..I.H.
10000090 01 21 99 60 01 bc 00 28-00 d0 00 47 12 48 13 49 .!.....G.H.I
100000a0 08 60 03 c8 80 f3 08 88-08 47 03 b5 99 6a 04 20 .....G...j.
100000b0 01 42 fb d0 01 20 01 42-f8 d1 03 bd 02 b5 18 66 .B...B.....f
100000c0 18 66 ff f7 f2 ff 18 6e-18 6e 02 bd 00 00 02 40 .f.....n.n...@
100000d0 00 00 00 18 00 00 07 00-00 03 5f 00 21 22 00 00 .....!..
100000e0 f4 00 00 18 22 20 00 a0-00 01 00 10 08 ed 00 e0 .....
100000f0 00 00 00 00 00 00 00 00-00 00 00 00 74 b2 4e 7a .....t.Nz
```

TOPPERSのWebからダウンロード
可能なPico用のROMモニタ

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
- 組み込みハードウェアの基礎知識
 - 本田 晋也(名古屋大学)
- 組み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
 - 竹内良輔
- 開発環境の確認
 - 竹内良輔
- Pico/Pico2+TEB003ボード用の改変
 - 竹内良輔