情253「ディジタルシステム設計」

組み込みシステム・プログラミングII

Agenda

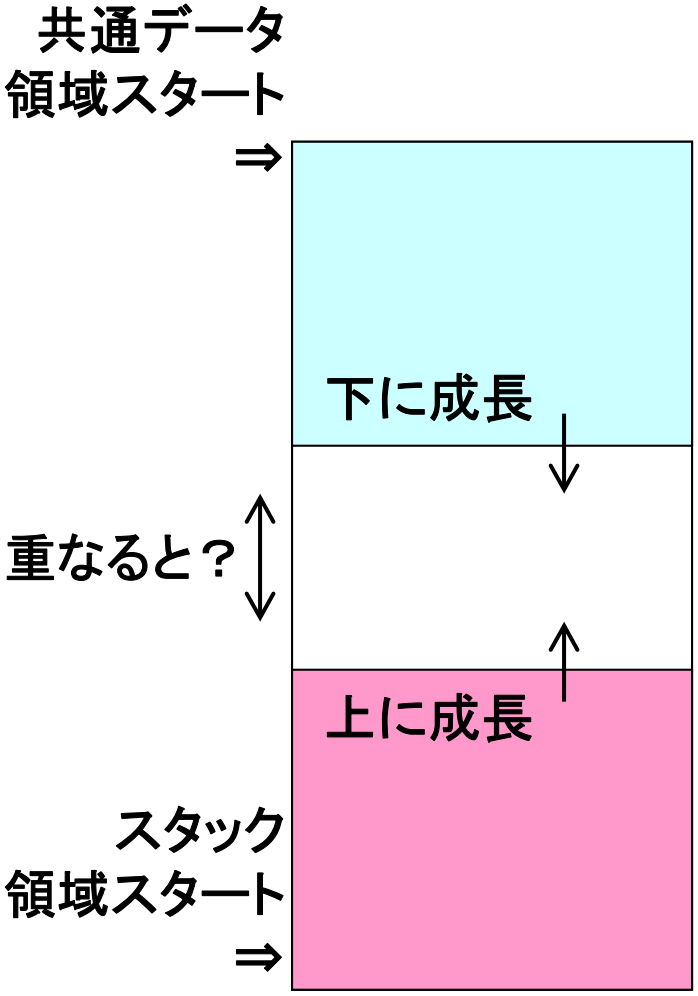
1. CPUとPeripheral HWとプログラム
2. 組み込みシステム・プログラミングの特徴
3. 開発環境
4. 簡単な応用例

Agenda

1. CPUとPeripheral HWとプログラム
2. **組み込みシステム・プログラミングの特徴**
3. 開発環境
4. 簡単な応用例

前回の復習

- プログラム実行中の動的なメモリ領域の確保
  - （**スタック領域**）関数呼び出しごとにその関数のローカル変数領域が取られる
  - （**ヒープ領域**）メモリ確保関数(C言語のmalloc, C++のnew等)による確保
- 重なると→暴走



ポーリングと割り込み (1/4)

- ポーリング
  - ユーザー・プログラム中で、事象の発生を定期的にチェックする。
    - 例えばPeripheral HWの動作状態をチェック。
  - プログラムの構造は簡単。
  - リアルタイム応答性を保つのが難しい。
    - 他の処理の影響を受けやすく、負荷が高くなるとチェックの定時性が乱れてリアルタイム応答性が失われる。
- 割り込み
  - ユーザー・プログラムを強制中断し、緊急性の高い処理を優先実行する。
  - プログラムの構造は難解になりがち。（慣れが必要）
  - リアルタイム応答性を確保しやすい。
    - ある事象の発生を、割り込み信号などの方法でCPUに通知する。
    - CPUはその信号を受け、所定の割り込み処理ルーチンを実行する。
    - 割り込み輻輳時の応答性能について注意深く設計しないと破綻する。

ポーリングと割り込み (2/4)

```
//ポーリングの例
do{

 h_sts = read_hw_status( );
 if(h_sts==TRUE){  //①
   <Do time critical job>
   <Do non-critical job>
 }

 f_sts = func1( );
 if(f_sts==TRUE){  //②
   func2( );
   func3( );
 }

}while(1)
```

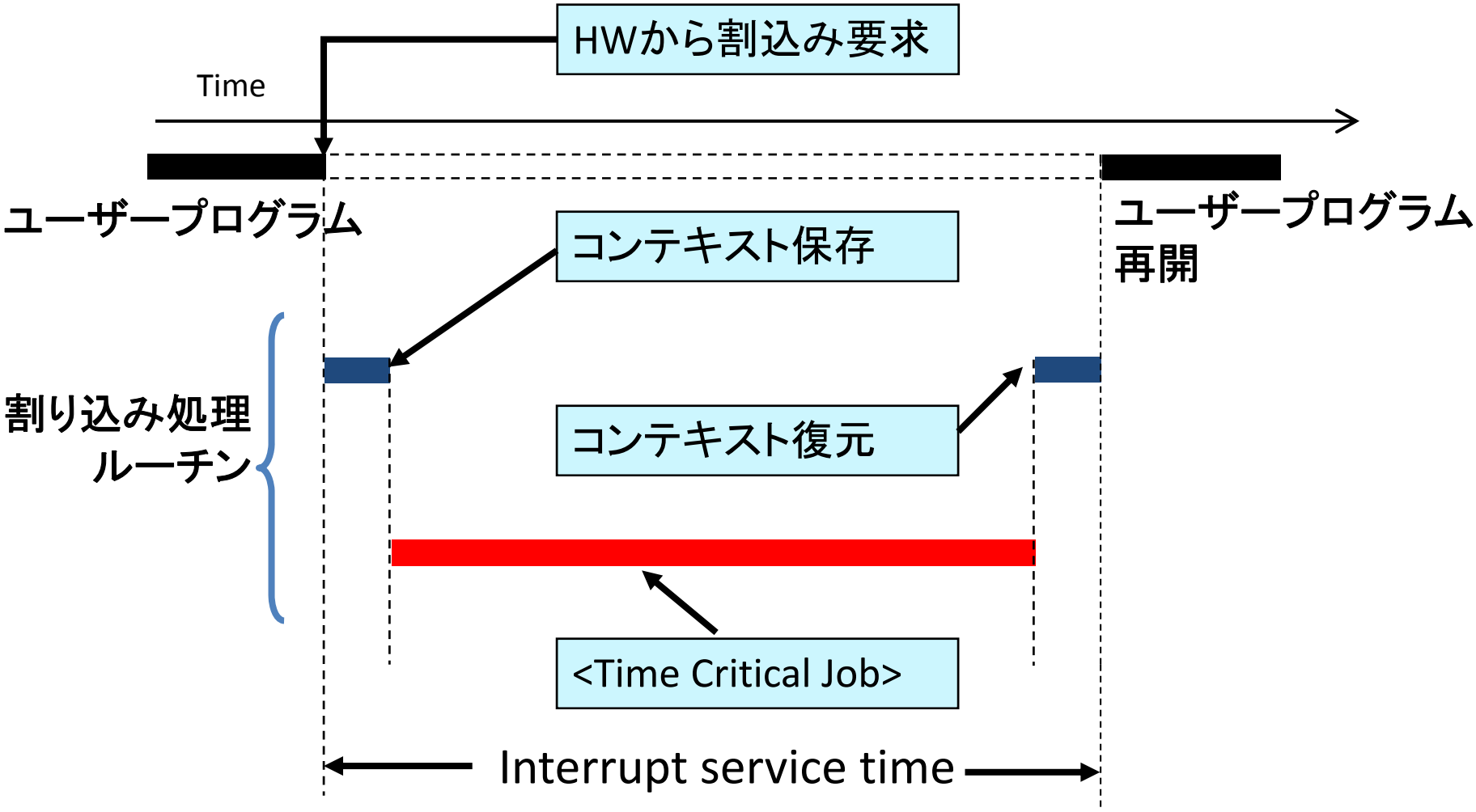
- ②のIF文が「FALSE」の間は、
  - ①の処理は定時性がある。
  - <Time critical job>は間に合う。
- ②のIF文が「TRUE」になると、
  - func2とfunc3が実行され、その処理が予想以上に重いと、
    - ①のリアルタイム性は崩れ、
    - <Time critical job>は破綻し、
    - システム・オーバーフローに至る可能性がある。

ポーリングと割り込み (3/4)

```
//割り込みを使用時の例
do{

 //h_sts = read_hw_status( );
 if(h_sts==TRUE){  //①
   //<Do time critical job>
   <Do non-critical job>
 }

 f_sts = func1( );
 if(f_sts==TRUE){  //②
   func2( );
   func3( );
 }

}while(1)
```

- 割り込み処理ルーチン
  - <Time critical job>を担当
  - 完了時にh_stsをTRUEに設定
- ①のIF文では、
  - <Non-critical job>を処理する。
- ②のIF文が「TRUE」でも、
  - 処理を中断し、割り込み処理ルーチンを優先的に実行することで、
  - <Time critical job>のリアルタイム性が保たれる。

ポーリングと割り込み (4/4)

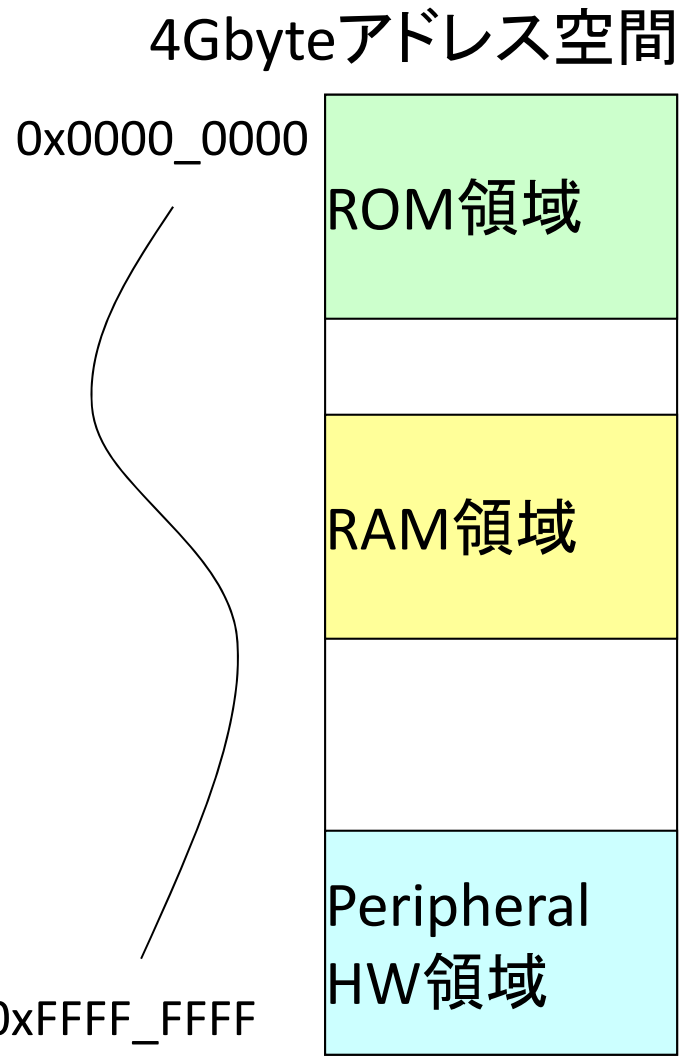


コンテキストスイッチ

- 複数のプロセスが1つのCPUを共有できるように、CPUの状態(コンテキスト)を保存したり復元したりする過程
- コンテキストとはそのプロセスが使用し得る全てのレジスタ(特にプログラムカウンタ)や、プロセスの実行に必要となるオペレーティングシステム固有の情報が含まれる。多くの場合、これらのデータは1つのデータ構造として保存。

メモリ・マップトI/O (1/2)

- CPUのアドレス空間が1枚（フラット）の時、
  - Peripheral HWは、その空間の一部に配置される。
  - メモリ・デバイス（ROM/RAM)も、その同じ空間の別アドレスに配置される。
- メモリと外部入出力（I/O）を担当するPeripheral HWが、アドレス空間的に区別が無い場合、
  - メモリ・マップトI/O方式と呼ぶ。



メモリ・マップトI/O (2/2)

- HWの制御レジスタにアクセスする場合、
  - 例えばDMACの制御レジスタが下記のようになっているのなら、
    - Source Address : 0xFFFFFFF0
    - Destination Address : 0xFFFFFFF4
    - Byte Length : 0xFFFFFFF8
    - Start : 0xFFFFFFFC
  - プログラムでも、そのアドレスをDefineし、

```
#define rDMA_SA          (*(volatile int *) 0xfffffff0)
#define rDMA_DA          (*(volatile int *) 0xfffffff4)
#define rDMA_BYTE        (*(volatile int *) 0xfffffff8)
#define rDMA_START       (*(volatile int *) 0xfffffffc)
```

  - それを使ってプログラムを書く。

```
rDMA_SA = &(rx_buf[0]);
rDMA_DA = &(tx_buf[0]);
rDMA_BYTE =  sizeof(rx_buf);
rDMA_START |= 1;
```

プログラムの移植性 (1/2)

- プログラムがシステム依存となる主な要因
  - ROM、RAM、Peripheral HWなどのアドレス配置
  - Peripheral HWの制御方法
  - CPUの割り込みアーキテクチャ
  - スタートアップ・ルーチン
  - 標準ライブラリ
  - ダミーの時間稼ぎループ
  - What else ?

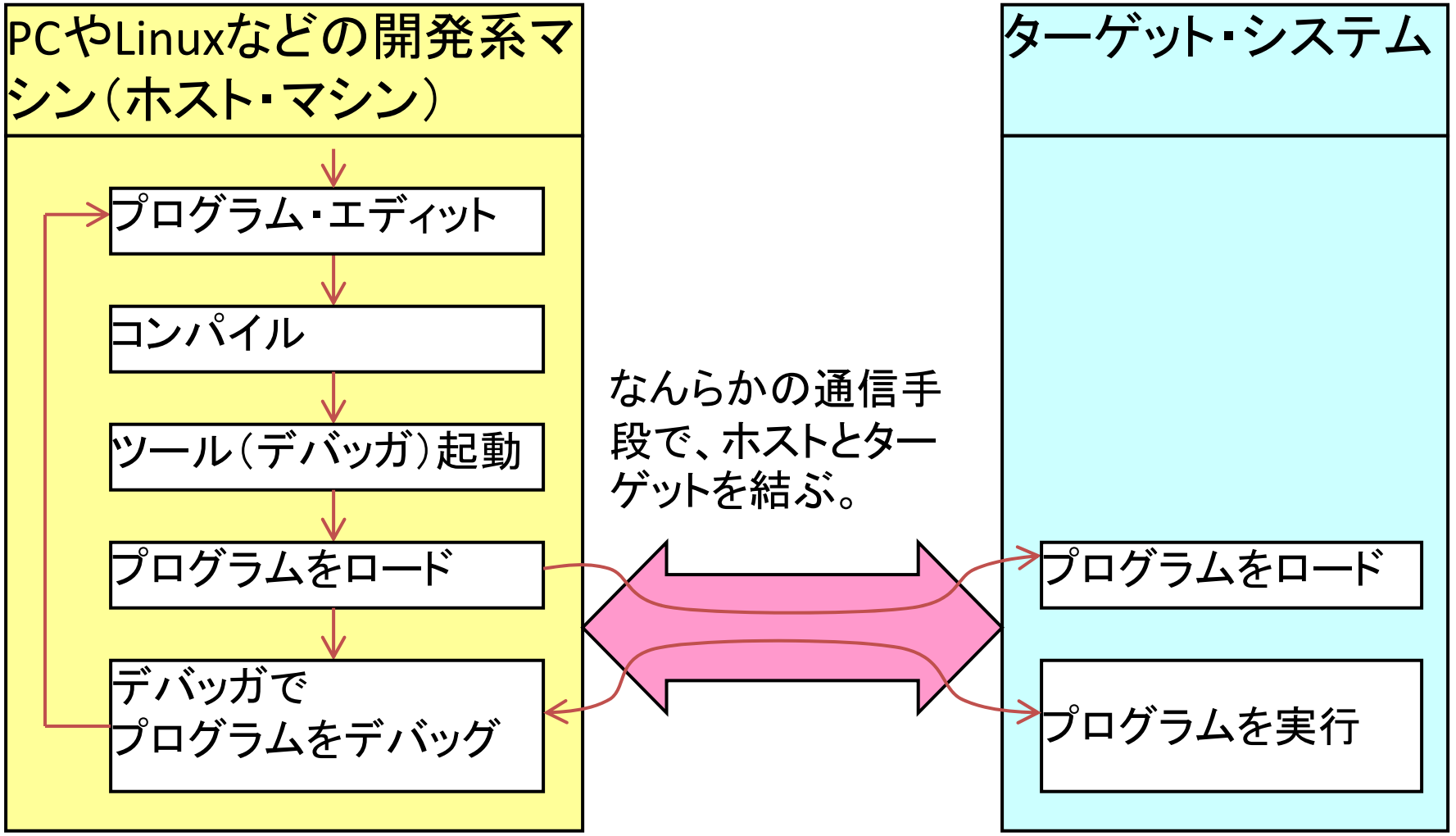
プログラムの移植性 (2/2)

- 機器依存となる要因を理解し局所化する。
  - 何が機器依存かを見極めるのは時に困難で、手間もかかる。
  - 将来機器間移植をするかどうかは不透明な場合が多い。
  - とは言っても、メンテナンスのことを考えるとやっていたほうがいい。
- 機器に組み込む前に、PCやLinuxマシン上でシミュレーションする時もある。（ある意味で移植を考えたプログラミング）
  - それを実現するコードを、スタブ・コードと呼ぶ。
  - スタブ・コード上に、システム依存部を押し込む。
  - 関数単体テストや、一部機能を切り出したテストなどでは使える。（全体となると難しい）

Agenda

1. CPUとPeripheral HWとプログラム
2. 組み込みシステム・プログラミングの特徴
3. **開発環境**
4. 簡単な応用例

クロス開発環境

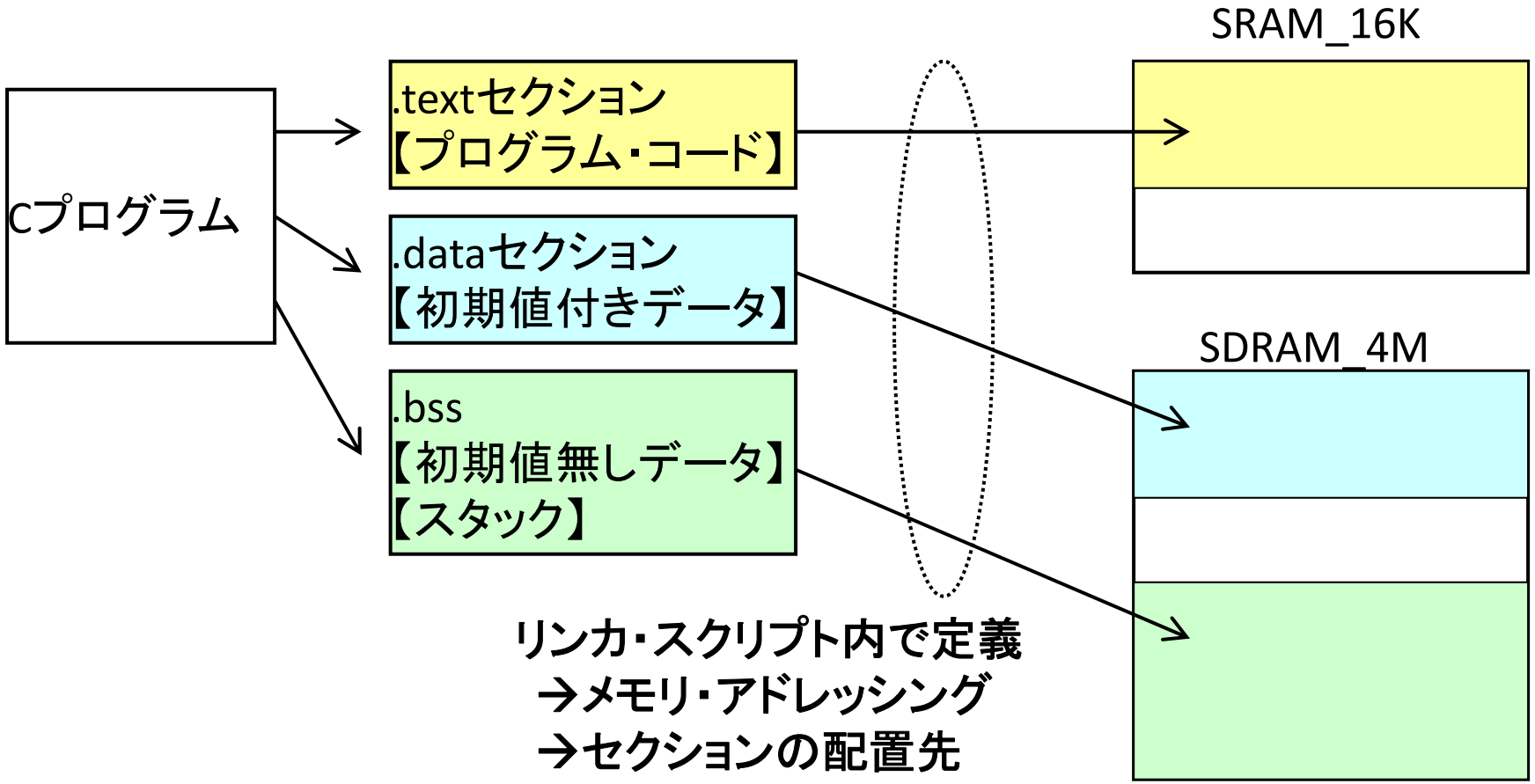

PCやLinuxなどの開発系マシン（ホスト・マシン）
プログラム・エディット
コンパイル
ツール（デバッガ）起動
プログラムをロード
デバッガで
プログラムをデバッグ
なんらかの通信手段で、ホストとターゲットを結ぶ。
ターゲット・システム
プログラムをロード
プログラムを実行

Cコンパイラ (1/3)

- ASCIIテキストで記述されたCプログラムを、CPU専用のマシン後に変換するツール。
- Cコンパイラと1言で呼ばれるが4つの段階がある。
  - **プリプロセス**
    - コンパイルの前処理 （#include、#define、#ifdef等を解決）
  - **コンパイル**
    - Cコード（Cプログラム）をアセンブラ・コードに変換
  - **アセンブル**
    - アセンブラ・コードをオブジェクト・コードに変換
    - オブジェクト・コードはマシン語に近いが、まだCPUが解釈できない中間的なコード。リンクに必要な情報が含まれている。
  - **リンク**
    - 全てのコードを結合
    - ターゲット・システムのメモリ・マッピングに合わせてコードを配置
    - マシン語の実行ファイルを生成

Cコンパイラ (2/3)

- リンクについてもう少し
  - ツール名としては、「リンカ」と呼ばれる。



Cコンパイラ (3/3)

- リンカ･スクリプトの非常に簡単な例

```
MEMORY
{
        SRAM_16K : ORIGIN = 0x20000000, LENGTH = 0x00001FFF
        SDRAM_4M : ORIGIN = 0x40000000, LENGTH = 0x003FFFFF
}
SECTIONS
{
        .text : { *(.text) } > SRAM_16K
        .data : { *(.data) } > SDRAM_4M
        .bss  : { *(.bss) *(COMMON) } > SDRAM_4M
}
```

デバッガ (1/3)

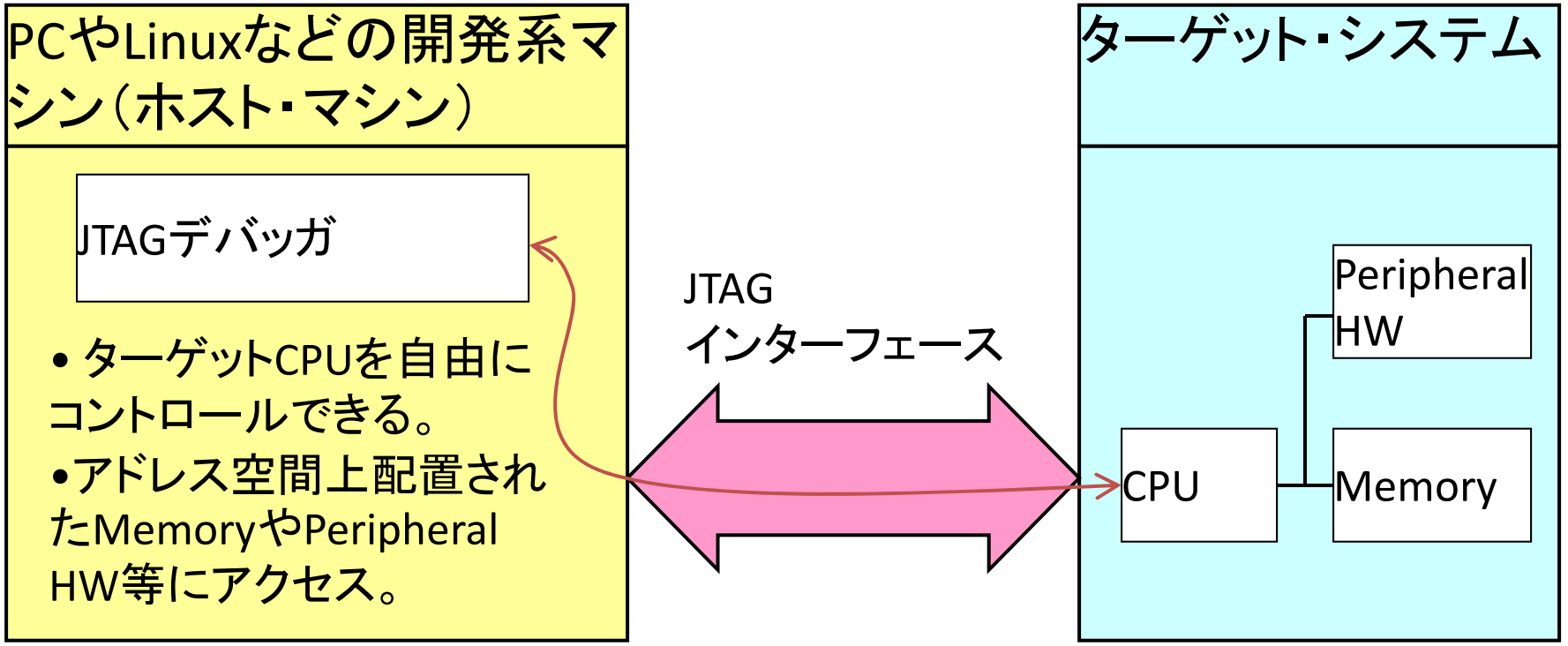
- プログラムをデバッグする為のツール。
- 基本的な機能
  - ブレイク・ポイント
    - プログラムを所望の場所で止める。
  - ステップ実行
    - プログラムを1行づつ実行する。
  - アドレス空間参照
    - データ、CPUのレジスタ、Peripheral HWのレジスタ等のメモリマッピングされた場所の値を参照できる。
  - 他にも非常に多くの機能がある。
- プログラムを一旦止めてデバッグするのが基本。
  - CPUとは独立して動作している部分がある場合には要注意。
    - 例えば、プログラムは止まってもPeripheral HWは止まらない。

インサーキット・エミュレータ (In-circuit emulator)

- CPUの機能をエミュレート（再現）するハードウェアで、実際のマイクロプロセッサと同じ機能を実装し、さらにブレーク（プログラムの実行を一時停止する）などのデバッグ機能を持つ特別な装置。主に組み込みシステムやOS、BIOSなど、ソフトウェアデバッガを使用できない環境でのプログラム開発に使用する。通常、略してICE (アイス)と呼ばれる。
- デバッガとしては最強と言えるが、かなりの高額であるうえ、特定のCPUにしか使用できない。
- 近年のCPUの高速化に伴いICEの開発が難しくなっており、CPUの進化についていけていない。 組み込みシステムではASICを採用する例が多くなってきており、ICEを使用することができない。
- といった事情があるため、近年では使われることが少なくなってきており、JTAGエミュレータやROMエミュレータなどのオンチップ・エミュレータを使用することが多くなってきている。

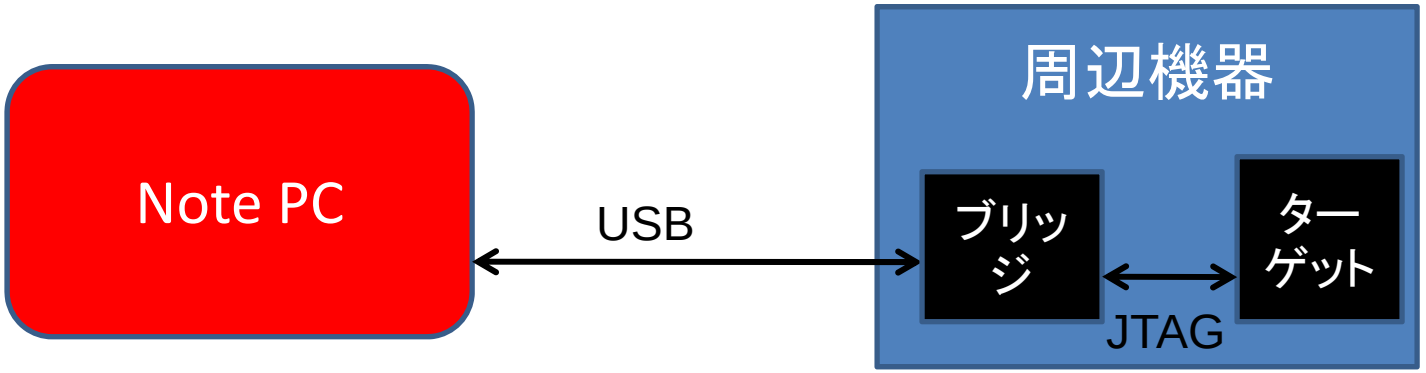
デバッガ (2/3)

- 組み込みシステム向けはJTAGデバッガが主流。
  - JTAGインターフェースで、ホストとターゲットを接続する。
  - よく、JTAG-ICEと呼ばれる。（ICE = In-Circuit Emulator）



JTAG　& USB & RS-232C

- JTAG（ジェイタグ, Joint Test Action Group）、IEEE 1149.1 標準で定められたシリアルの**集積回路や基板の検査、デバッグをする信号通信規格**
- USB（Universal Serial Bus：ユニバーサル・シリアル・バス）は、コンピュータに周辺機器を接続するためのシリアルバス規格、**Mbps以上に対応**。
- RS-232 (Recommended Standard 232) は、パソコンと音響カプラ、モデムなどを接続するシリアル通信方式のインターフェースの一つである。インターフェースはポートとも呼ばれるため、シリアルポートと一般に呼ばれることもある。ケーブルの最大長は約15mで、最高通信速度は115.2kbps。



デバッガ (3/3)

- ブレイク・ポイントについてもう少し
  - インストラクション・ブレイク
    - 通常、ブレイク・ポイント設定といったらこれ。
    - 本来の命令コードを「ブレイク命令」に書き換える。
    - デバッガが勝手にやるのでユーザーからは見えない。
    - ROMの場合どうする？
  - データ・ブレイク
    - 命令コードでは無くデータなのでどうやって実現する？
    - あるアドレス空間へのデータ・リード／ライトを、どうやって検出する？
- ハードウェア・ブレイク・ポイント
  - CPUが専用のレジスタを具備している場合がある。
    - この特殊用途レジスタは、CPUのHWの一部と考えてこのように呼ぶ。
  - ブレイク・アドレスをそのレジスタに指定する方式。
    - 命令コード書き換えでは無いので、ROM上のプログラムにも適用可能。
    - データ・リード／ライトにも適用可能。
  - レジスタ数には限りがあるので、設定する数が制限され自由度は劣る。

その他のデバッグ補助ツール

- print関数
  - 古典的だがやっぱり有効な手段。
  - プログラム（CPU）を止めなくてもいい。
  - 意外とコードサイズが大きいので注意。
  - 組み込みシステムの場合の出力先は？
- モニタ・プログラム
  - 本来のアプリケーションと一緒に実装してしまうデバッグ用プログラム。
  - HyperTerminal等を利用しRS232C等でホストと通信して、ターゲット・システム内部の状態を参照／設定する。
  - プログラム（CPU）を止めなくてもいいが、多少CPUの負荷が増える。
- ロジック・アナライザ
  - 信号線の1/0の状態を観測する測定器。
  - プログラムの状態をチップの外部に出力できれば使える。
  - 外部ポートへの出力なのでCPUの負担も小さい。

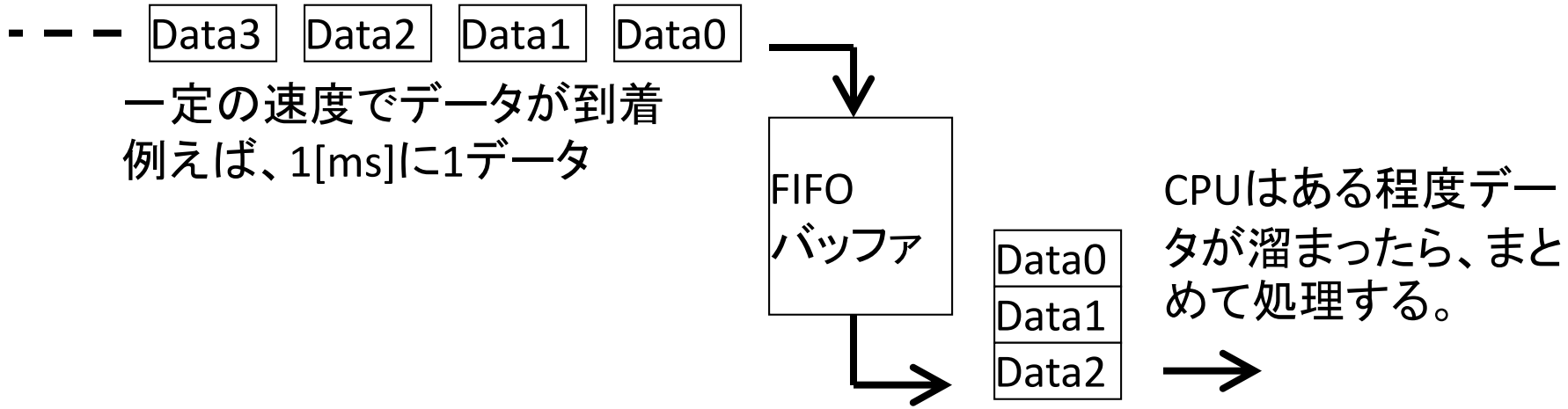
Tektronix ロジックアナライザー

Agenda

1. CPUとPeripheral HWとプログラム
2. 組み込みシステム・プログラミングの特徴
3. 開発環境
4. **簡単な応用例**

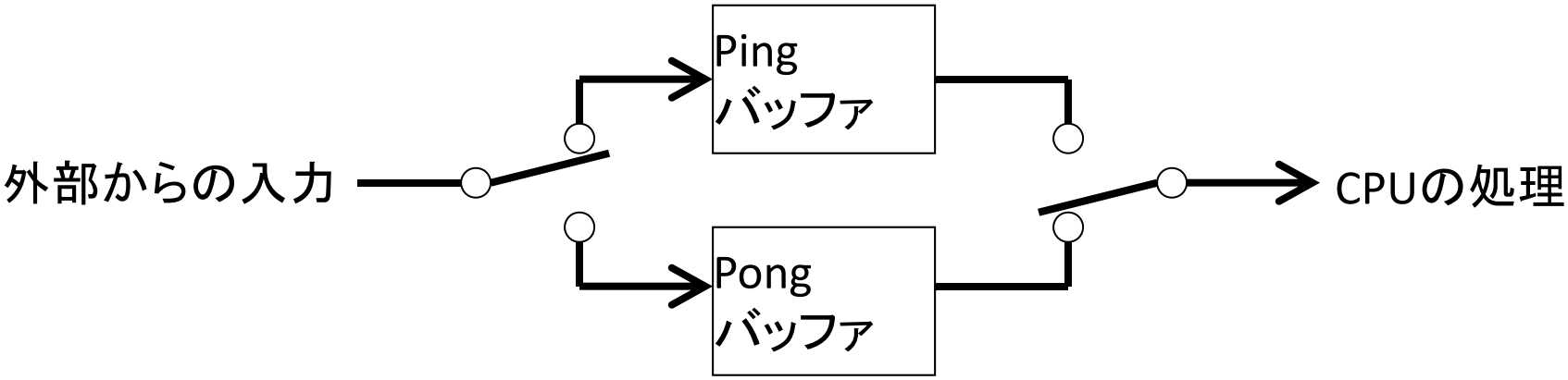
基本的なバッファ管理 (1/3)

- FIFOバッファ
  - FIFO = First In First Out
  - マイコンの外部からのデータ入力のスピードと、CPUの処理スピードのミスマッチを解消する時に有効。
    - 入力レートは一定だが、CPUの処理速度はバラつきがある場合。
  - 他にも使い道はいろいろあるが、各種のミスマッチ解消用のバッファ。

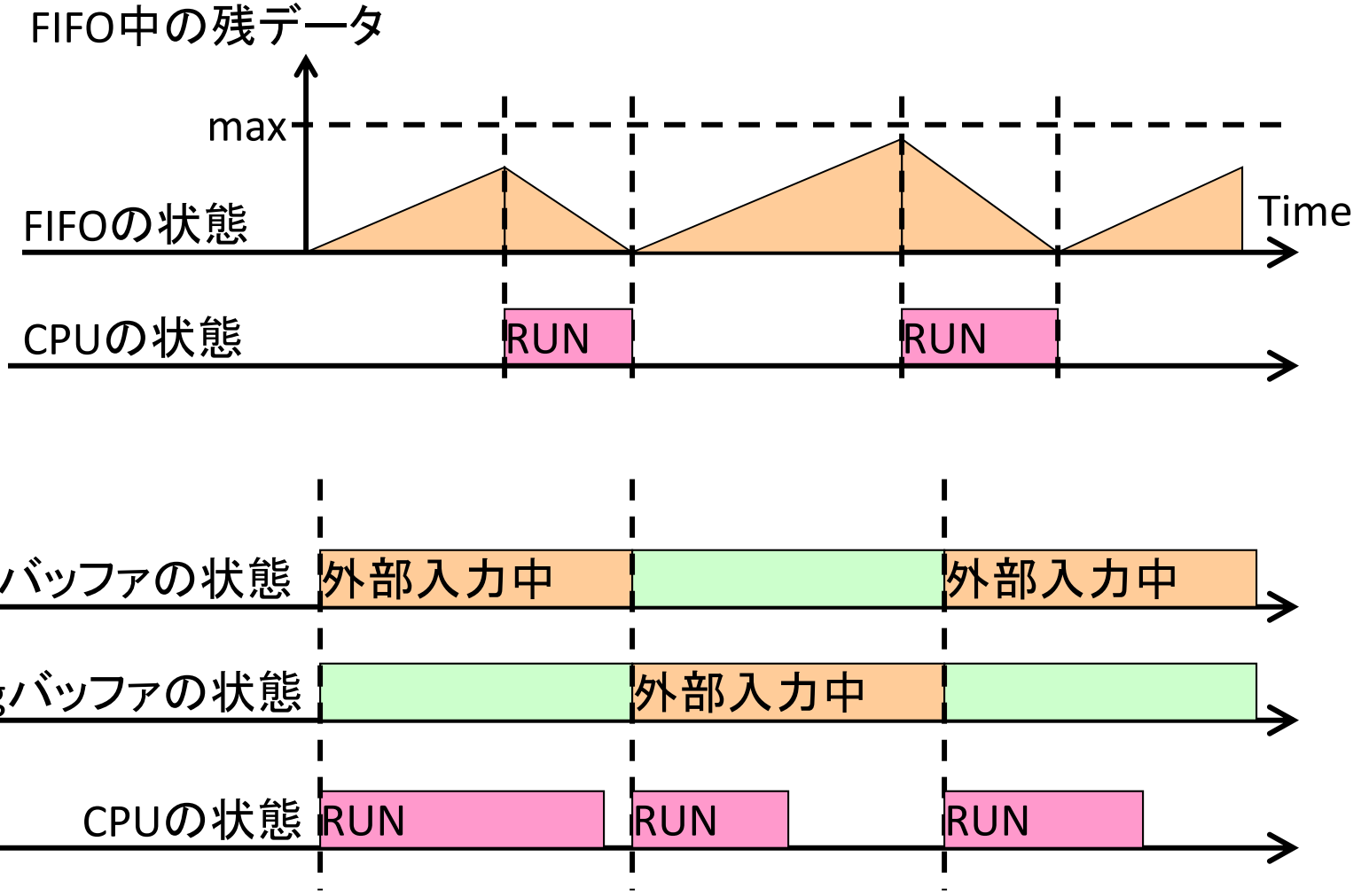


基本的なバッファ管理 (2/3)

- Ping-Pongバッファ
  - 2つのバッファを交互に使う。
  - データ入力と、データ処理をオーバーラップさせるときに有効。
  - FIFOと似たような使い方。
    - 入力と処理のミスマッチ解消。
  - メモリを2倍持つことになる。



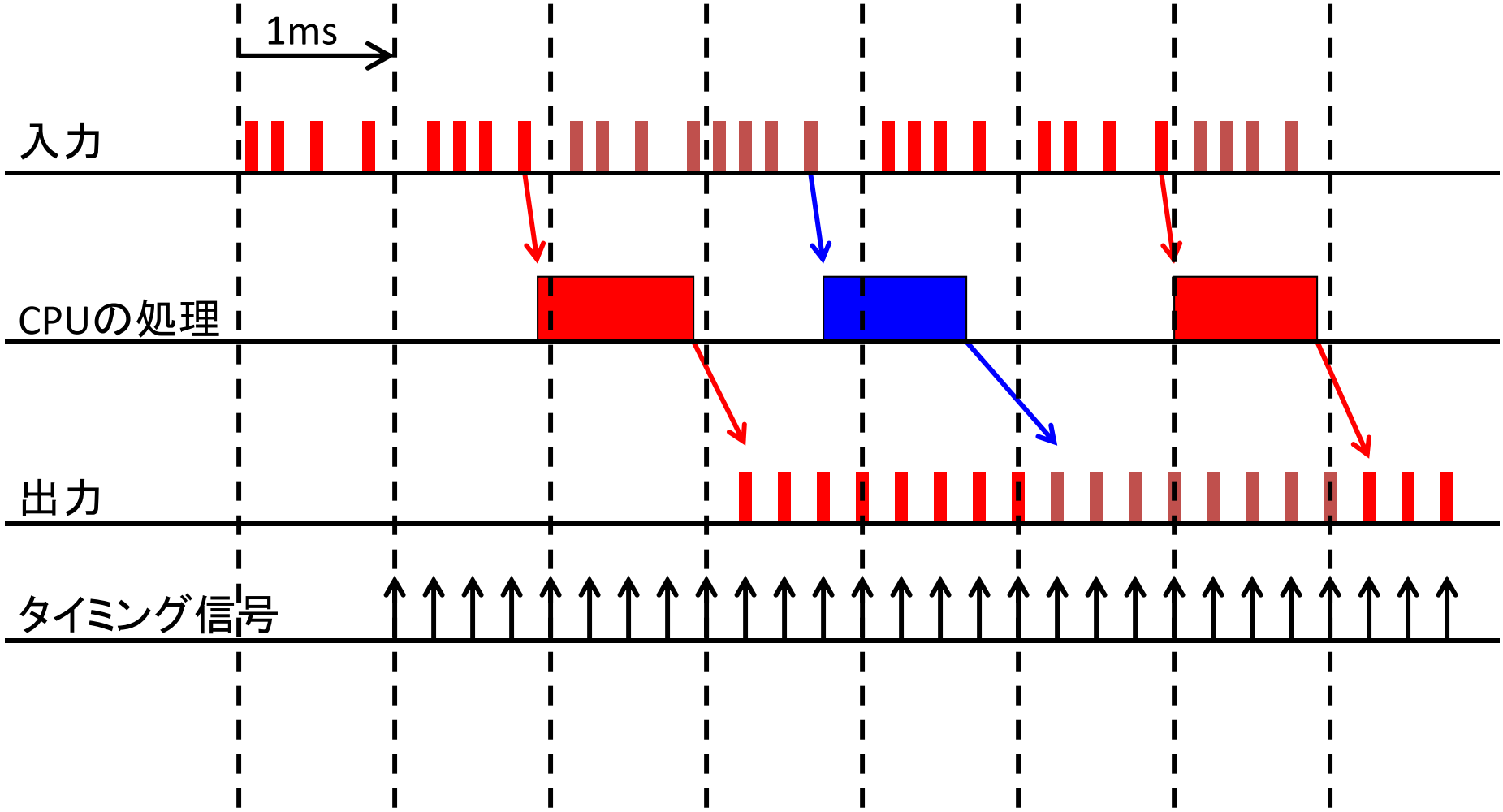
基本的なバッファ管理 (3/3)



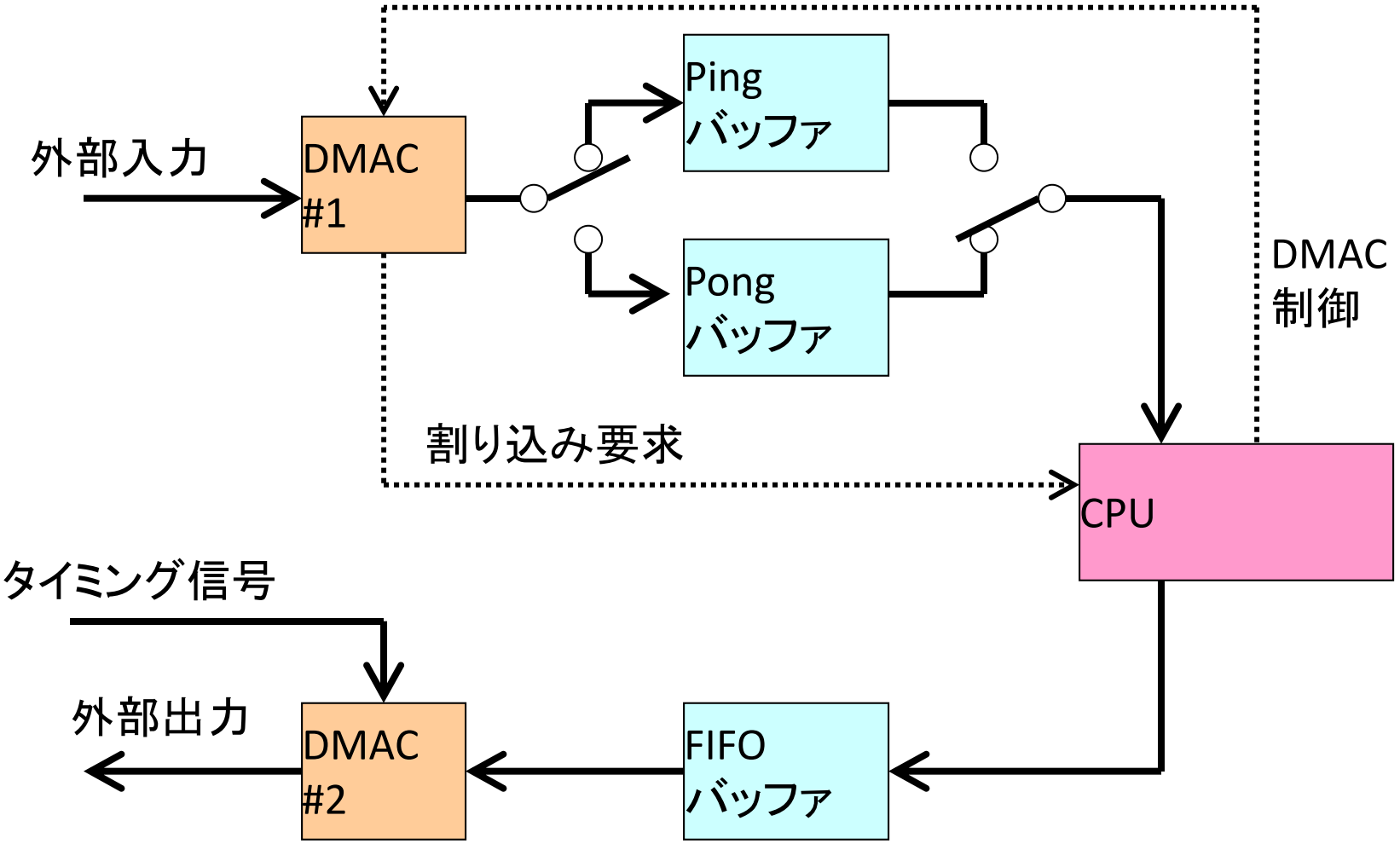
応用問題1

- 入力データ
  - 1[ms]の間に1byte単位で4byteのデータが到着する。
  - 1byte毎の間隔は一定ではない。
    - 4byte/1[ms]は確定、1byte/0.25[ms]は不確定。
- CPUの処理
  - １回の処理には8byteのデータが必要。
  - 8byteのデータをランダムにアクセスして計算する。
  - 出力は8byte。
- 出力データ
  - タイミング信号に同期して一定の間隔で出力する。
  - 出力単位は1byte。

応用問題1



応用問題1


外部入力
DMAC
#1
Ping
バッファ
Pong
バッファ
DMAC
制御
割り込み要求
CPU
タイミング信号
外部出力
DMAC
#2
FIFO
バッファ

HW2

- 以下の課題に対して回答レポートを作成せよ
- 採点基準（各問に対してA4 1ページ程度のレポートを作成せよ）
  1. ロジックアナライザーとオシロスコープを調査し、その類似点・相違点を述べよ
  2. 以下の事項についてさらに詳細に調査報告せよ
     JTAG、USB、RS-232C、UART
  3. 以下の応用問題2を実現する方式を提案せよ
     - 入力データ
       - 1[ms]の間に4*64byteのデータが到着する。
       - 1[ms]の最初に2*64byte到着し、最後に2*64byte到着する
     - CPUの処理
       - １回の処理には4*64byteのデータが必要、すべてのデータがそろわないと計算を開始できない。
       - CPU計算に0.7ms程度必要で、結果出力は4byte。
     - 出力データ
       - 1[ms]に1回のタイミング信号に同期して4byteを出力する。