

デジタルシステム設計 講義レジメ
担当：和田知久 (ファイヤー和田)

ちょっと考えましょう！

センサー検知し、エアバッグが動作するシステムを想定し、割り込みを用いた実装と割り込みを用いない実装例を説明し、それぞれのメリット・デメリットは？

基本的な周辺回路

1) リアルタイムクロック(RTC)

バックアップ電池、水晶発振、カウンター、レジスタ、バスインターフェース

OSが時間を知るため必要

割り込み機能を持つ場合あり

2) タイマー

内部クロックをカウントし、ある時刻ごとに割り込み信号を生成させる

このタイマーの方が RTC より高精度

○OS のタスク切り替え (コンテキストスイッチ) のタイミング

○デバイスドライバーのタイムアウトや異常検知のタイミングを計る手段

タイマーはマスク可能な割り込み手段

3) ウオッチドッグタイマー

プロセッサの以上を検出するタイマー

数十ミリ秒から数秒ごとに初期化されないとプロセッサが異常と判断し、割り込みもしくはリセットをかける。

○マスクできない割り込み

4) DMA

Direct Memory Access (DMA) とは CPU のデータ転送命令 (LOAD, STORE) を用いずにメモリをアクセスする機能

DMA コントローラが上記アクセスを実行する

○CPU をメモリアクセスの時間から解放し、計算等の仕事に集中させ無駄をはぶく

○組込みシステムでは、CPU の仕事を減らし低電力化の効果があり重要

○例：ストリーミング処理

ストリーミング処理はデータ転送が多い

CPU では映像・音楽処理を行いながら、DMA コントローラが DMA 転送を行う。

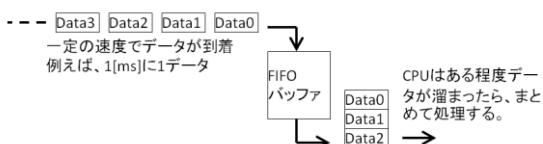
データがそろい次第、CPU は次の処理に入り、DMA コントローラは前の処理のデータの転送や次の処理のためのデータの準備を行う。

○複数の DMA 転送を行う場合が通常なので、DMA の制御レジスタを複数持ち、それぞれをチャネルと呼ぶ。

次ページに DMA を用いた、処理系の例を示す。

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。

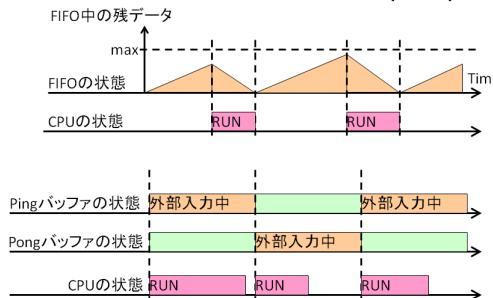


基本的なバッファ管理 (2/3)

- 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。



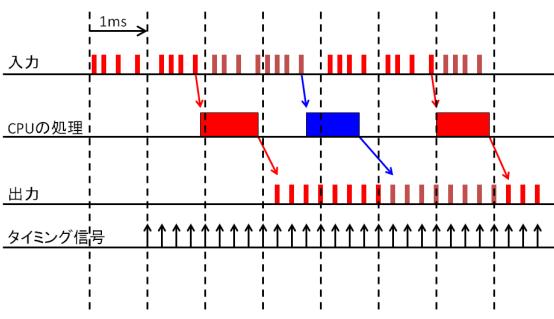
基本的なバッファ管理 (3/3)



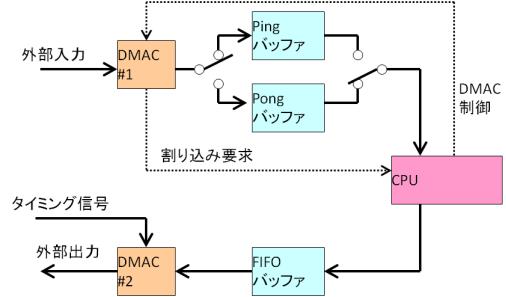
応用問題1

- 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



例題 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。

5) シリアル通信インターフェース

1 ビットずつシリアルに通信を行う。

方式の例 : UART (Universal Synchronous Asynchronous Receiver Transmitter) 図 6.16

RS232C、USB、JTAG など調べてみてください。

6) スタックに関する補足

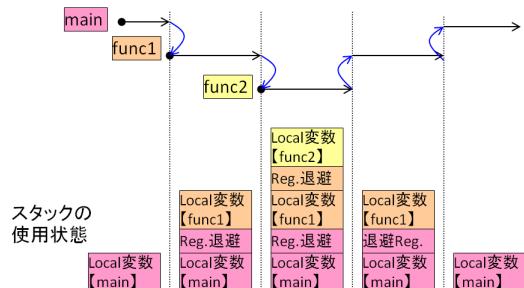
ファンクション・コールとスタック (1/3)

- ファンクションから別ファンクションをコールするのは日常茶飯事


```
int func1(<arg>,<arg>){
    //前処理
    val = func2(<arg>,<arg>); //ファンクション・コール
    //後処理
    return(val)
}
```

 - func1が「前処理」をした後、func1の実行を中断しfunc2が実行される。
 - func2の実行が終わると、func1の「後処理」を再開する。
- 中断したファンクションを、正しく再開するためには?
 - 途中経過を安全な場所に保存しておぐ場所が必要。
 - その保存場所が「スタック」と呼ばれるRAM上の領域。
- キーワード
 - CPUのレジスタ(プログラムカウンタ、CPUステータス、汎用レジスタ、etc)
 - 自動変数(ローカル変数)

ファンクション・コールとスタック (2/3)



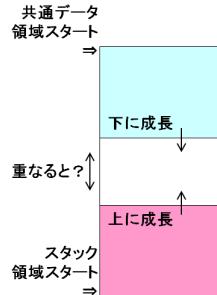
ファンクション・コールとスタック (3/3)

・スタック破壊の可能性

- これが起きると暴走
- デバッグが難しい
- 避ける手段は?

■PCやLinuxの場合だと

- OSが領域をプロテクト
- Core Dumpして終了



以上