初級者向けテキスト

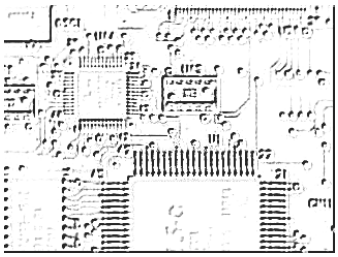
組込みソフトウェア管理者・技術者育成研究会
ーSESSAMEー
(http://www.sessame.jp/)

-*- 目　　　　　　次 -*-

-*-

プログラミング　組込み用語基礎知識

三浦　元



おしながき

1. プログラムはどのように動くか
   – ランタイム構造とスタック
2. 配列の実態
3. スタートアップ – main() 以前
4. 周辺デバイス
   – 周辺デバイスへのアクセス, ラッチとゲート
5. エッジ・トリガとレベル・センス
6. 組み込みプログラムの構造
   – ポーリングと割込み, 同期呼出しと非同期呼出し
7. volatile – 変数と最適化
8. ハードウェアとのお付き合い

概論 – 組込みソフトとビジネスソフトの違い？

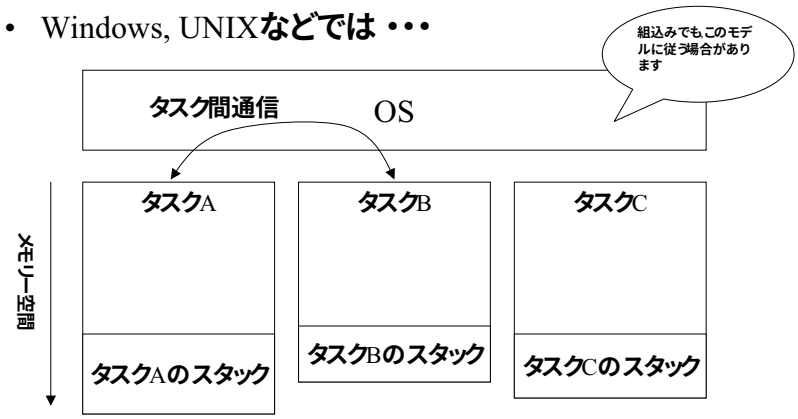
- ハードウェアと密接に関わる
  - ハードウェアの、またはハードウェアを介した事象・状態の読み取り・取得
  - ハードウェアの制御（レスポンス、アクション）
  - メモリーフットプリントや消費電力など、ハードウェアを意識したプログラミングが要求されることが多い
- リアルタイムプログラムが多い
- OSを使わないことが多い
  - OSを使う場合でも、プログラムの実行は特有の手続きが必要
- 稼働環境（物理環境）がいろいろ
  - 実行論理環境は比較的安定
- 品質・信頼性に、より高度なものが求められる
- 出荷したらめったにアップデートできない

1. プログラムはどのように動くか

- スタックとのおつきあい
- 関数呼び出しとスタック
  - スタックが保持する情報
  - 関数呼び出しの深さとスタック
  - スタックのオーバーフロー

プログラムのランタイム構造

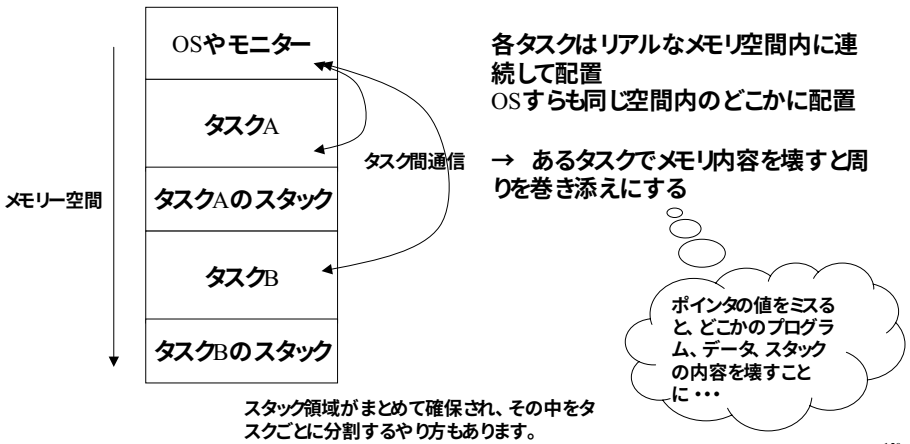
- Windows, UNIXなどでは・・・



各タスクは独立した仮想メモリ空間内で動作
→ 個別のタスクでメモリ内容を壊しても、そのタスクが死ぬだけ

プログラムのランタイム構造

- 多くのリアルタイムOSでは・・・



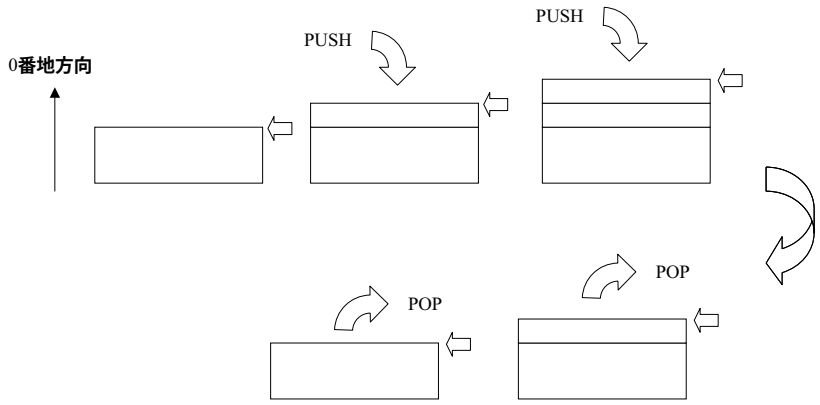
各タスクはリアルなメモリ空間内に連続して配置
OSすらも同じ空間内のどこかに配置

→ あるタスクでメモリ内容を壊すと周りを巻き添えにする

ポインタの値をミスると、どこかのプログラム、データ、スタックの内容を壊すことに・・・

スタック領域がまとめて確保され、その中をタスクごとに分割するやり方もあります。

スタックの使い方



スタックに積まれた途中のデータはPOPできない。
スタックの先頭位置（⇦）は、スタックポインタで常に管理されている。

151

スタックの使われ方 (1)

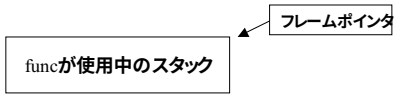
1.

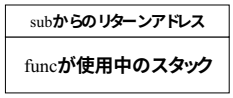
```
int func()
{
    int k;
    k = sub( 2 );
}
int sub( int n )
{
    int a=b=0;
    b = n * 2;
    return( b );
}
```

2.

```
int func()
{
    int k;
    k = sub( 2 );
}
int sub( int n )
{
    int a=b=0;
    b = n * 2;
    return( b );
}
```

引数をスタックに積む場合もあります



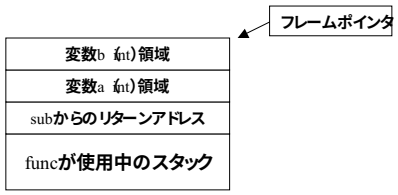


152

スタックの使われ方 (2)

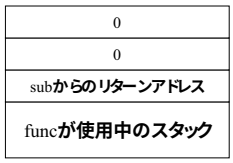
3.

```
int func()
{
    int k;
    k = sub( 2 );
}
⇨ int sub( int n )
{
    int a=b=0;
    b = n * 2;
    return( b );
}
```



4.

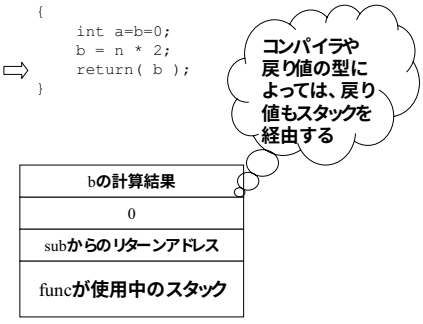
```
int func()
{
    int k;
    k = sub( 2 );
}
int sub( int n )
{
    int a=b=0;
⇨   b = n * 2;
    return( b );
}
```



スタックの使われ方 (3)

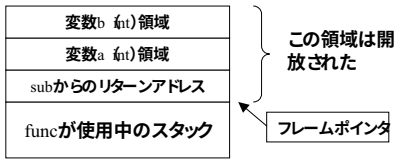
5.

```
int func()
{
    int k;
    k = sub( 2 );
}
int sub( int n )
{
    int a=b=0;
    b = n * 2;
⇨   return( b );
}
```



6.

```
int func()
{
    int k;
    k = sub( 2 );
⇨ }
int sub( int n )
{
    int a=b=0;
    b = n * 2;
    return( b );
}
```

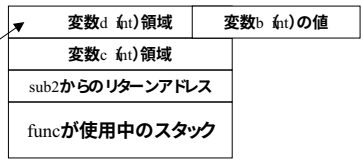


スタックが原因の トラブル (1)

```
int func()
{
    int i, j;
    i = sub( 2 );
    j = sub2( i );
}
int sub( int n )
{
    int a,b;
    b = n * 2;
    return( b );
}
int sub2( int m )
{
    int c, d;
    c = m + d;
    return( c );
}
```

計算結果がおかしい？？

sub() が使っていたスタック領域
続いてsub2()が使用



初期化していない

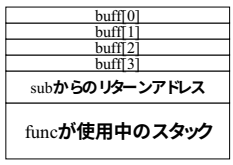
スタックが原因の トラブル (2)

1.

```
int func()
{
    sub( "hogehoge" );
}
int sub( char c[] )
{
    char buff[4];

    strcpy(buff, c);

    return 0;
}
```

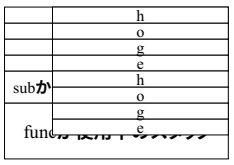


2.

```
int func()
{
    sub( "hogehoge" );
}
int sub( char c[] )
{
    char buff[4];

    strcpy(buff, c);

    return 0;
}
```



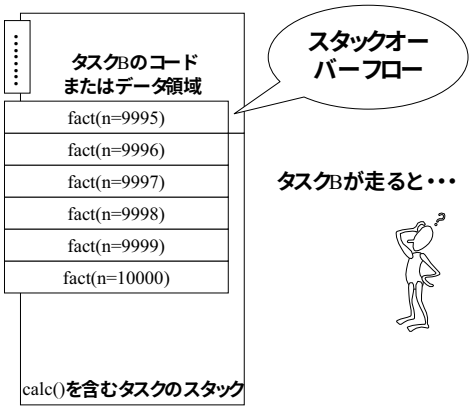
3.

次の瞬間の行き先は・・・

バッファオーバーフロー

スタックが原因のトラブル (3)

```
int calc()
{
    fact( 10000 );
}
int fact( int n )
{
    if ( n == 1 )
        return 1;
    return( fact( n-1 ) * n );
}
```



スタックに関する定石

- スタック変数をはじめとした変数の初期化を忘れない
- バッファへの書き込みは必ずサイズチェックを行い、バッファサイズを超えて書き込みを行わないようにプログラムで制御する
- タスクごとの最大スタック使用量を見積もり、資源の枠内で余裕を持ってスタックサイズを設定する

格言

挙動不審なプログラムの裏にスタックあり

159

2. 配列の実態

- **配列はメモリー上でどうなっている？**
- **intの配列の場合**

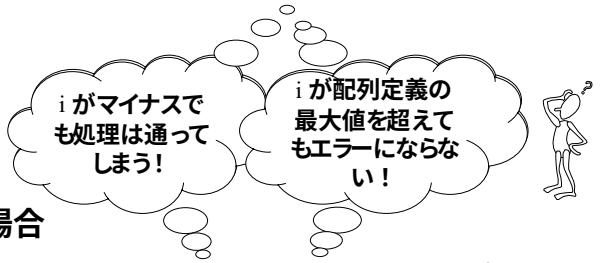
int iarray[4];

0
1
2
3

intの語長

iarray[i]へのアクセスは、Cコンパイラは次のように扱う・・・

(int)*(&iarray[0] + i * sizeof(int))



- **structの配列の場合**

```
typedef struct {
    int i1, i2;
    long lo;
} st_def;

st_def   sarray[10];
```

sarray[i]へのアクセスは、Cコンパイラは次のように扱う・・・

(st_def)*(&sarray[0] + i * sizeof(st_def))

160

つまり配列とは・・・

- 配列と同じ型情報をもった、配列要素個々の先頭のアドレスを指し示すポインタ の集まりである。

そして

- コンパイラは、ランタイムにポインタの指し示す先の正当性 ・妥当性の検証までは面倒を見てくれない。

つまり

- インデックスの正当性 ・妥当性の検証はプログラムロジックで記述しなければならない

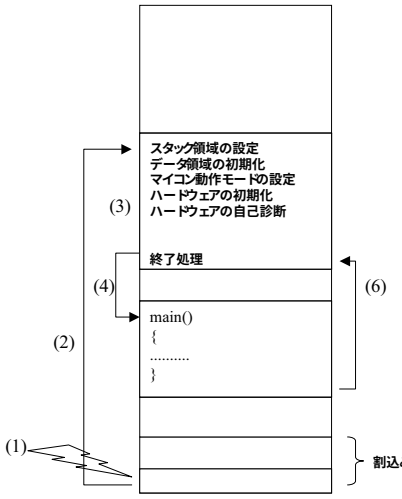
格言

配列のインデックスには悪魔が潜む

3. スタートアップ – main()以前

- 実行の前に：
  - 私の前に道は無い、私の後に道ができる
    - 誰が道を整備するのか？
    - 誰が処理をmain()の玄関口まで導くのか？
  - 環境を整えなければ動けない・・・・
    - ハードウェア初期化
      - 周辺デバイスの初期化、自己診断指示
      - メモリー初期化
    - データ領域の初期化
      - スタック領域の確保
    - マイコンの動作モード設定

プログラム実行の流れ



(1) リセットによる強制割り込みが入る

(2) 割り込みベクトル領域に書かれた番地にジャンプする

(3) スタートアップルーチンの処理を実行する

(4) main() を呼び出す

(5) アプリケーションプログラムを実行する

(6) アプリケーションプログラムを終了する

もう少し詳しく見てみよう

- メモリマップ
  - プログラムとデータをメモリ中でどのように配置するかを設計する
    - プログラム
      - プログラムはROMまたはFlash
        - » ROM/Flash上のコードを直接実行するか
        - » 一旦RAMに展開してから実行するか
      - ハードディスクにある場合はRAMに展開
    - データ
      - 定数はROMまたはFlash
        - » ROM/Flash上の値を直接参照するか
        - » 一旦RAMに展開してから参照するか
      - static, globalなどの変数はRAM
        - » ROM/Flash上の値をRAMに展開する必要がある
        - » 変数領域の初期化

変数はどこにある？

- 自動変数はスタック
- global,static宣言の変数はRAM
- register宣言の変数はCPUレジスタ
  - コンパイラの最適化機能により、特に宣言していなくてもregisterに割り付けられることも。
  - むやみにregister宣言をしても、資源は限られている。
    - registerが使えなければ、スタック変数に振り替えられる
      宣言したのにパフォーマンスがあがらないけれど・・・・・？）

プログラムの配置？

- どこに配置するかを設計時に明確にする
  - ROM領域　不揮発性だがアクセスが遅い
  - Flash領域　書き換え可能、不揮発性だが書き換え回数に制限がありアクセスが遅い
  - DRAM領域　大容量 書き換え可能 そこそこ高速だが揮発性
  - SRAM領域　高速だが揮発性、コスト高
    - バッテリーバックアップが付くと不揮発領域にできる
    - ここを使用する場合には、スタートアップの中でコードコピーを記述する必要がある
- インスタンス生成のコスト
  - インスタンスを生成する＝ROMからRAM領域にコードをコピーし初期化すること
    - CPU資源を使用する、時間がかかる、コンストラクトデストラクトを繰り返すとメモリーが虫食いになる・・・
  - スタートアップの中で、必要なインスタンスを生成しておくことも。

ちょっと休憩 - ROM　SESSAME組込み用語集 http://www.sessame.jp/ より

ROMは読出し専用のメモリで、書き込まれた内容は電源を切った状態でも保存されます。読出しは通常CPU等のバスマスタからリードサイクルで行うことができますが、書込みや消去はライトサイクルでは行うことができません 正確には全く出来ないもの、数ミリ秒の時間が必要なもの、特別な電圧が必要なもの、手順が複雑なもの等があります)。ROMにはEPROM、ワンタイムプログラマブルROM、EEPROM，フラッシュメモリ、マスクROM等の種類があり、それぞれの特徴に適したところに使用します。EPROM、EEPROM，フラッシュメモリは内容を消去して再書込みができますが、書込み回数には制限があり種類によってその保証回数は数百～数十万回とさまざまです。また、書き込まれた内容の保証期間も十数年～百年程度とメーカによりばらつきがありますので長期間使用する用途には注意が必要です。

EPROM

- EPROMは紫外線消去可能なROMで UV-EPROMと書かれることもあります。EPROMには石英ガラスの透明な窓がついていて、ここに紫外線を照射することでメモリの内容を消去することができるようになっています。書込み後はこの窓に遮光ラベルを貼って書いた内容を保護することが必要です。EPROMへの書込みにはROMライタ(ROM プログラマー)と呼ばれる専用ツールがよく使用されます。EPROMには書き込み回数の上限はありますが、実際には紫外線消去の手間が発生するので、その時間的な制約で書き込み制限回数を超えて使用されることはほとんど考えられません。もっとも、消去／再書き込みをするときには普通機械的にソケットから抜き差しするのでその頻度が多いとICのピン磨耗により破損する可能性が大きいと考えられます ソケットも通常多くても数十回程度の抜き差ししか保証されていませんので注意が必要です)。EPROM部品自体には石英ガラスが必要なのでその分コストアップになり、また実装には通常ソケットが使用されるのでそこにもコストがかかります。しかし バージョンアップ等のROM交換が現場でできることやEPROM自体が再利用できるメリットがあるので以前はよく使われていました。現在ではフラッシュメモリが代わりに多く使用され需要を伸ばしています。

ちょっと休憩- ROM (2)

ワンタイムプログラマブルROM(OTPROM)

- EPROMから石英ガラスの窓を無くした物で、1回だけ書込みが可能なROMです。それを除けばEPROMとほぼ同様です。石英ガラスの窓がない分EPROMよりコストが下がっています。ROMによっては同じ形状で窓付きのものと窓無し（ワンタイム）の両方発売されている場合と窓無し（ワンタイム）しか発売されてない場合があります。小規模のワンチップマイコンでコストや実装面積が優先されるような場合窓無しタイプしかない製品があります。窓無し（ワンタイム）しかないものは開発段階ではある程度使い捨てとなりますが、開発・生産・保守のライフサイクルを通じて書き換えの割合が少ない場合に使用されています。

EEPROM

- 電気的消去可能なROMで、実装された状態で読み書きが可能なメモリです。書込みはバイト単位（容量の大きなものではページと呼ばれる連続した領域）で行うことができて、CPUからのライトサイクルやコマンドで書き込みを行った後、書込みが完了するまでに数ミリ秒の時間を待つ必要があります（書き込み中は他の場所にデータを書くことはできません）。尚、消去は書込みの際に自動的に行われるので特に意識する必要はありません。EEPROMに書き込まれる内容は組み込まれる装置の動作に必要なパラメータ等、ユーザが電源を切っても保存しなければならないデータの格納に主として用いられます。書き込み回数に制限があるので（数十万回程度）、書き換えが多い用途には同じアドレスにメーカの指定する制限回数以上書込みをしないようなアルゴリズムが必要です。EEPROMにはシリアルEEPROMと呼ばれる種類があります。シリアルEEPROMにはアドレスバス・データバスといった信号は無く、数本の信号線（クロック、シリアルデータ等）を制御することで読み書きができるようになっています。シリアルEEPROMのメモリ配列をメモリ空間・I/O空間に配置することはできませんが、小型の装置で利用できる8ピン程度のDIPやSOP等のパッケージ製品もあります。

ちょっと休憩- ROM (3)

マスクROM

- マスクROMは部品の段階ですでにデータ（プログラム）が書かれているメモリで、消去／書込みは全く不可能です。実際にはICの回路上でデータ（プログラム）がパターンとして作りこまれていて変更することは出来ませんので他のメモリとは概念的に大きく異なると考えて差し支えないでしょう。しかし、部品としてのコスト（1ビット当たりの価格）はROMの中でも低い方です（最も低いのはNAND型フラッシュメモリと言われてますがストレージ以外の用途ではマスクROMはほぼ最低価格です）。ROMに書くデータ（プログラム）が確定し将来変更の可能性がない場合に使用されています。イニシャルコスト（数十万円以上）が必要ですから、部品単価・装置の生産台数を考慮して採用するか否かの判断をする必要があります。

フラッシュメモリ

- フラッシュメモリはEEPROMから改良されたもので、1ビットの記憶セルを構成するトランジスタを2つから1つにすることでより大きな容量を得ることが可能となり、また、消去と書込みをブロック単位で行うようにしたことでEEPROMに比べて書き込みが速く、消費電力が小さくなっていることが特徴です。フラッシュメモリは従来のEPROMやOTPROMのようにプログラム等の固定的なデータメモリだけでなく、同時に一部をEEPROMのようにパラメータ保存領域として使うことも可能です。さらに、オンボードでプログラム書き換えができることやその書き換えが遠隔でも可能になること等メリットが大きく、最近のコストダウンと相まってその需要を大きく伸ばしています。
フラッシュメモリは大別してNOR型とNAND型があります（他の種類もありますがこの2つが現在の主流です）。
NOR型はランダムアクセス・高速アクセスが可能なので通常CPUのプログラムメモリによく使用されています。
NAND型はランダムアクセスが不可能で、またNOR型よりもアクセススピードが低速ですが、1ビット当たりの単価がNOR型よりも安いのが特徴です。欠点としてNAND型は読出し時の信頼性が低くビットエラーを起こすことがあります。そこでパリティのような補助的なビットをいくつか付加してエラー検出／エラー訂正を行うなどの工夫をして信頼性を向上させています。NAND型はストレージデバイスへの用途として（ハードディスクの代わりになるものとして）開発され、スマートメディア等の記憶素子に使われています。

ハードウェアの初期化

- ハードウェアリセットいっぱぁつ！ ・・・というわけにはいかない
  - 電気屋さんとのコミュニケーションが重要
    - CPUリセットと同時に、周辺デバイスはどのような振る舞いをするのか？ をよく知る必要がある。
  - その上で
    - 周辺デバイスに対してソフトウェアで何を設定（初期設定）すべきなのか、をシステム仕様から理解する。
    - 初期設定のプログラムをスタートアップ部に漏れなく記述する。
  - ハードウェアの自己診断結果を判断する
    - ハードウェアによっては、リセット投入後または初期化により自動的に自己診断を行うものもあり、その自己診断を意識してプログラムする必要もある。
    - 詳しくは電気屋さんの仕様書または周辺デバイスのデータシートをよく読むこと。

格言

スタートアップに始まりスタートアップに終わる。

リセットとは、責任を伴う行為である。

4. 周辺デバイス

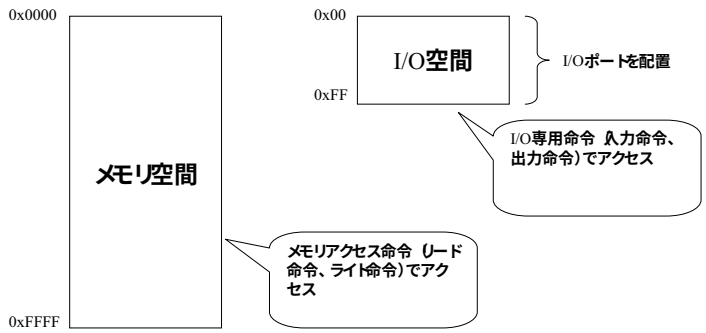
- 周辺デバイスはCPUから見て、入出力（I/O）の概念で扱われる。
- CPUと周辺デバイスは論理的にどうつながっているか？
  - 専用のI/O空間を持つ方式
  - メモリ・マップドI/O方式

どのような命令で入出力が実行されるか？

- ポート
  - 周辺デバイスとの間でデータの積み下ろしを行う「港/窓口」
- 通信
  - シリアルデータはデバイスでどのように扱われるか？

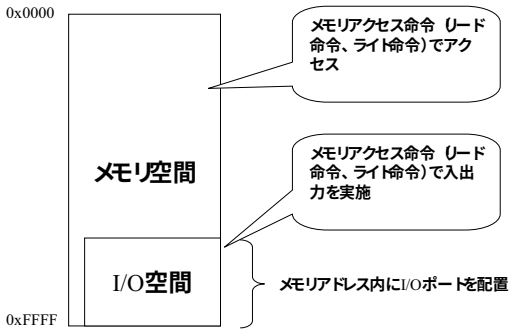
専用のI/O空間を持つ方式

- メモリ空間（アドレス）と、I/O空間（アドレス）を別に管理する方式



メモリ・マップドI/O方式

- メモリ空間内にI/Oを配置する方式



I/O（ポート）へのアクセス

- ポートにはアドレスがある
  - I/O空間のアドレス or メモリ空間のアドレス
- ポートにはデータ型がある
  - 8bit 幅アクセス、16bit幅アクセス・・・・
- つまり、ポートへのアクセスでは、型指定が重要

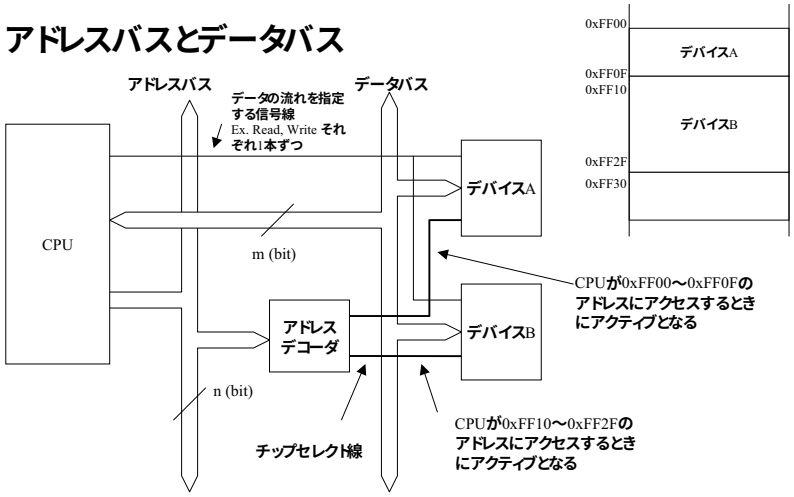
```
unsigned int *device_A_port1 = (unsigned int *)0x1200;

unsigned char *device_B_port3 = (unsigned char *)0x1201;
```

- 同じアドレスでも、READとWRITEでは違う機能にアクセスする（メモリーはアドレスが同じならR/Wは同じメモリセルが対象）

汝、如何にしてデバイスにアクセスするや？(1)

- アドレスバスとデータバス



177

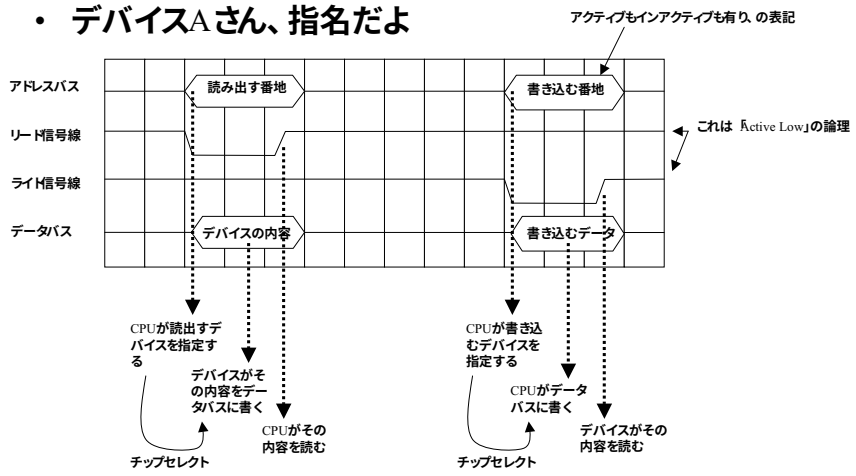
汝、如何にしてデバイスにアクセスするや？(2)

- アドレスバスとデータバス
  - アドレスバスが16bit幅であるCPUは、2^{16}=0xFFFF だけのアドレスを指定可能
  - 実際にすべてのアドレス空間にメモリやデバイスが隙間無く配置されることは少ない・・・ハード仕様書をよく読むことが重要
  - データバスが16bit幅であるCPUは、1回のREAD/WRITE命令（またはIN/OUT）で2バイト幅のデータ(short)を転送できる。

178

汝、如何にしてデバイスにアクセスするや？(3)

- デバイスAさん、指名だよ



179

汝、如何にしてデバイスにアクセスするや？(4)

- ご指名でないデバイスは玄関のドアを閉じる
  - トライ・ステート（Tri-State）
    - High H デジタル論理の「高」状態
    - Low L デジタル論理の「低」状態
    - High Impedance Z 接続していないのと同様の状態
- つまり
  - チップ・セレクトがアクティブでなければ、デバイスはデータバスをHigh Impedance状態にして「しらんぷり」を決め込む。

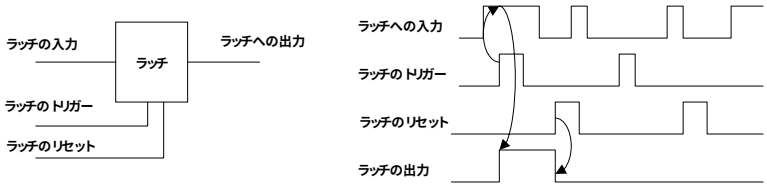
「混信」を防ぐだけでなく、ファンアウト性能のからみもある。

聴衆が少なければ小さな声で良いが、聴衆が増えると大きな声を出さないと通らない

180

ラッチとゲート(入出力)

- プログラムは、見に行った時点の「状態」しか知りえない・・・
- ある時点のスナップショットを「保持」するラッチ
  - ソフトウェアトリガまたは定期的なクロックでスナップショットを取得する
  - 保持した値をリセットするには明示的に「リセット」を指示する



- 状態をリアルタイムにスルーするゲート
  - 読みに行った「瞬間」の状態を取得する

I/Oマップ– I/Oの仕様書

入力の記述例

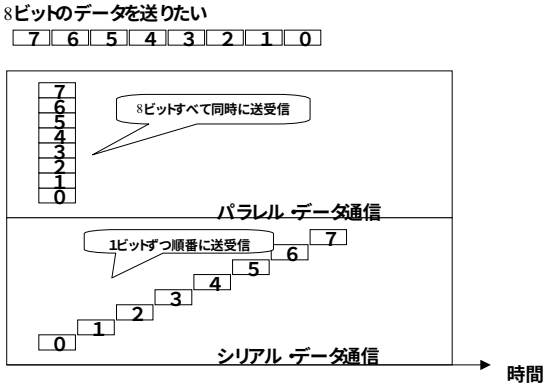
番地	bit	周辺装置	デバイス	注釈
0xFF00	0(LSB)	スイッチA	ゲート	0: 押されている 1: 押されていない
	1	スイッチB	ゲート	0: 押されている 1: 押されていない
	••••			
	7(MSB)	スイッチH	ゲート	0: 押されている 1: 押されていない
0xFF01	0 - 7	パラレル	ラッチ	
••••				

出力の記述例

番地	bit	周辺装置	デバイス	注釈
••••				
0xFF01	0	パラレル	ラッチ	ラッチトリガー
	1	パラレル	ラッチ	ラッチのリセット
••••				

通信　：シリアルとパラレル

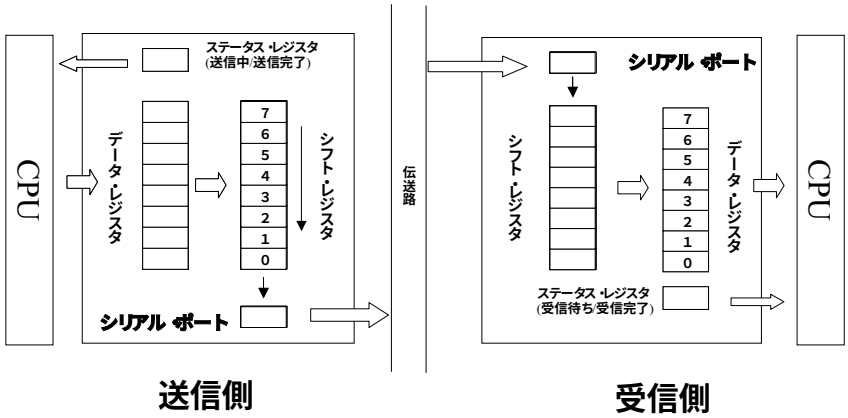
- **シリアル通信 :**RS232C, Ethernet, USB, IEEE1394 etc.
- **パラレル通信 :内部バス**, IDE, **セントロニクス**, SCSI etc.



183

SESSAME CONTENTS 2004

シリアル通信の仕組み



184

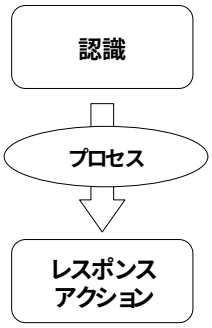
SESSAME CONTENTS 2004

格言

隣人をよく理解することはトラブル防止への第一歩

5. エッジ・トリガとレベル・センス

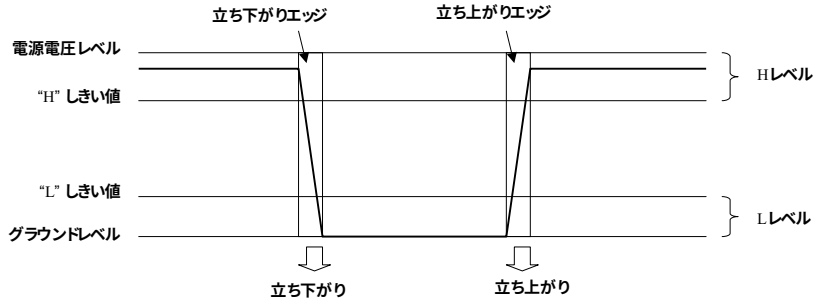
- 現象を、デバイスを通してどのように受け取るか？
  - 入力
    - スイッチ入力
    - LSIの状態入力
    - 通信データ入力
  - 入力には、次の3種類がある。
    - 変化したことを読み取る
    - 読取りにいった瞬間の状態を読む
    - ある時点の状態を保持し、それを後から読み取る



- デバイスを通してどのようなデータ出力を行うか？
  - データバス上の「出力状態」は瞬間
  - 瞬間の出力で良いか、瞬間の状態を保持する必要があるか？

変化したことを読み取る - エッジ センス

- おや、立とうとしているね・・・
  - 信号の 変化」を捕らえる
- 立ち上がりエッジと立下りエッジ



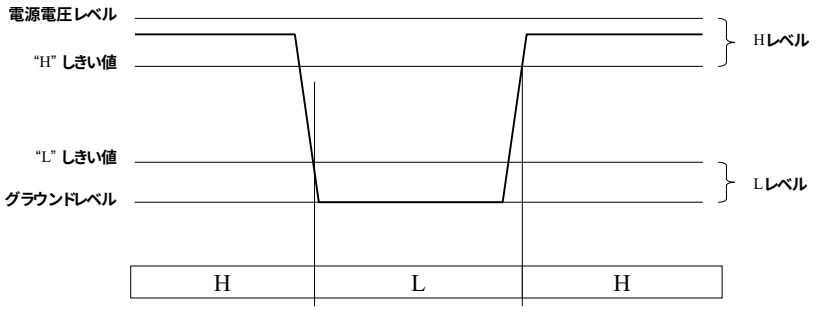
クロックはエッジトリガ方式として使用される信号の代表的なものです。各種信号の同期を取るのに用いられます。
割り込みに関して、レベルセンス／エッジトリガをソフトウェアで選択設定できるマイコンがあります。いずれに設定するかは、割込みの性質によって変わってきます。
センサやスイッチ等の変化を割込みに使用する場合はエッジトリガ方式が一般的です。

187

SESSAME CONTENTS 2004

読取りにいった瞬間の状態を読む - レベル センス

- うん、君は今座っているね
  - 信号の 状態」を捕らえる



回路では、チップセレクト R/W信号等の制御信号でレベルセンス方式がしばしば用いられます。

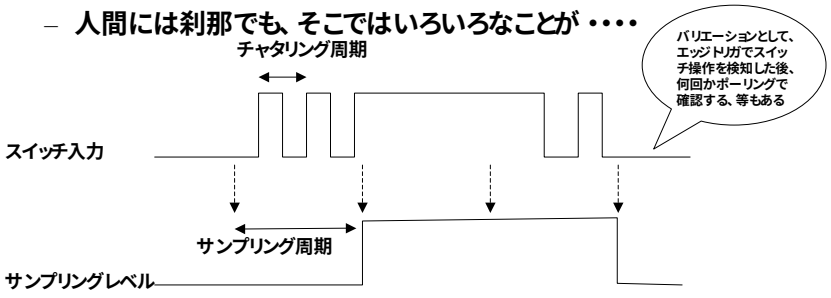
PCIバスで採用されているように、複数デバイスで割り込みが共有されているような場合にはレベルセンス方式が使用されることがあります。

188

SESSAME CONTENTS 2004

君、玄関のベルを2回押した？

- チャタリング
  - スイッチやリレーの接点が、自身の持つバネ性によりバウンスしてしまい、信号が短い周期でH/Lを行き来すること。
  - 人間には刹那でも、そこではいろいろなことが ・・・・



チャタリング周期よりもサンプリング周期を長く設定することでチャタリングの影響を回避できる。
ただし、サンプリング周期を長くしすぎると、操作に対する反応の遅れを生み、操作感を損なってしまうので注意が必要。

ある時点の状態を保持し、それを後から読む – ラッチ

- ラッチの実施方法
  - 特定の信号線の立ち上がりエッジ／立下りエッジでデータをラッチする
  - 定期的にデータをラッチする
    - クロック信号のエッジでラッチ
    - ソフトウェアでラッチストローブを出力
    - など

変化する状態や信号を処理しなければならない場合に、ラッチはとても便利です。

ただし、定期的なラッチも、プログラムの処理状況によって1サイクル分読み漏らしをしてしまったりする危険性が無いとはいえません。
そのような場合はラッチではなくDMA転送でゲートから読み込んだ値をメモリの特定領域に連続して格納することもあります。読み込むデータの周期、読み漏らしリスクの程度などにより、最適なデータ入力の回路・手法を選定するようにしましょう。

格言

変化点か状態か、それが問題だ

入出力の定石

- ハードウェア仕様をよく理解する
  - 周辺デバイスの初期化、アクセスに際して必要な制御、それらにハードウェアとして要する時間　などをよく理解する。
- 入出力のデータ幅、データ型に注意する
- 入出力がラッチなのか、ゲートなのかをよく理解し、
  - ゲートの場合は事象の発生として扱う最低（and/or 最高）の時間幅を明確にし、
  - 入出力の原因となる事象とあわせて、ソフトウェアが得た信号の扱いを適切にプログラミングする。

6. 組込みプログラムの構造

- プログラムではあれこれしながら、何やかや ・・・
  - タイムリな処理
  - ハードウェアからの状態取得
  - ハードウェアへの出力
  - ・・・・・

いくつもの皿回しをしながらお客さんと会話をし・・あぁ、サインをねだられちゃった・・おっと皿の回転が弱まったからエネルギー供給、っと・・あぁそうそう、サインが途中だった・・・えっ、今度は皿じゃなくてやかんだって！？

- 周辺デバイスからの状況をどのような仕掛けで入手するか
  - ポーリングを使用する
  - 割り込みを使用する

- プログラムの実行構造をどのようにするか
  - 同期呼び出し
  - 非同期呼び出し

ポーリング – 御用聞きモデル

- 各家庭を訪問して 何かご入用のものはありますか」と注文を聞き、それを届ける 御用聞き」
  - 訪問の順序は決まっている
  - 訪問の途中の各家庭での注文の有無、急ぎの配達のために一旦店に戻ったりという事象発生の有無などにより、訪問の時刻が保障されない
  - 各家庭からの注文に即時対応できない

```
main()
{
    初期化;
    while(1) {
        事象Aを調べる;
        事象Aが条件を満たしていたら処理する;
        事象Bを調べる;
        事象Bが条件を満たしていたら処理する;
        ・・・・・・・
    }
}
```

ポーリング周期の要件を明確にする。
全事象への処理時間が周期要件よりも長くならないようにする。
ことが重要！

割込み – 電話注文モデル

- 電話で注文を受け、焼きたてピザを届けるピザ屋
  - 電話が入るまでは下ごしらえなどの別処理をしている
  - 電話が入ると、それを待ち行列につなぎ、あるいは優先度の組換えをしながら約束のタイミング（配達納期）に間に合うように注文に従った処理を実施
  - 下ごしらえ中に電話が入ると、「どこまで下ごしらえをした」メモを書き、受話器を取る

```
main()   // 割り込まれる側
{
    初期化;
    while(1) {
        割込み禁止にする;
        割り込まれてはならない処理;
        割り込みを許可する;
        割り込まれても良い処理;
    }
}
```

```
interrupt()  // 割り込み処理
{
    割込み処理で使う資源を退避する;
    割り込みに応じた処理;
    退避していた資源の復帰;
}
```

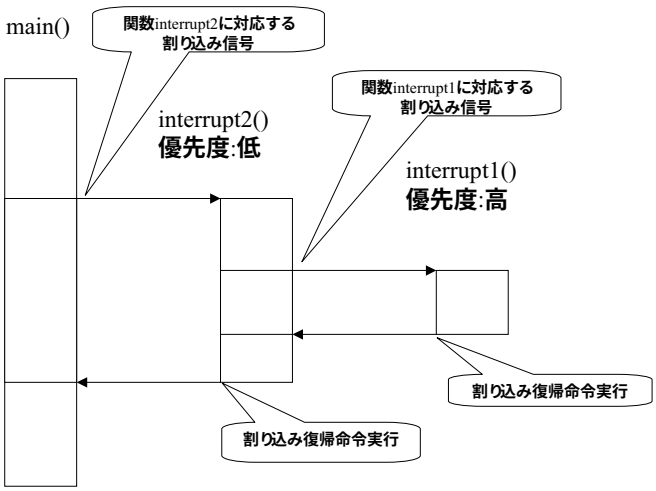

割込みも交通整理が必要

- 割込みのおきて
  - 割込みが発生した際に、処理中の作業のメモ（プログラムカウンタ、スタックポインタ等：コンテキスト）を保存する
  - 割込み処理を実行する
  - コンテキストを復元する（i.e. もとの作業に戻る）
- 割り込みの交通整理
  - 割込み優先度を考慮する
    - 割込みには優先度が付与できる
      - 割込みが重複した場合、優先度のより高い割込みが先に処理される
  - マスク可能割込み/マスク不能割込みを使い分ける
    - 割込みの受付を禁止／解除できる割込み　：マスク可能割込み
    - どのような状態下であっても受けなければならない割込み　：マスク不能割込み

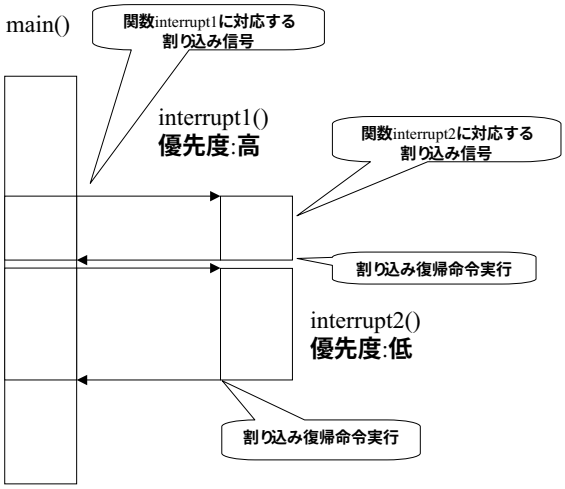
注文での長電話はご法度

- ピザの注文の電話で長電話をしてはいけない
  - 電話（割込み処理）は短く簡潔に
  - 急ぎの用件は割込み処理の中で、受け付けてから実処理まで余裕のあるものは別処理で
    - 割込みの発生したこと、詳細情報などをフラグ等にセットし、通常処理に受け渡す
  - 割込み応答の時間、割込みが発生してからそれに応じた処理が開始されるまでの最遅時間（ワーストケース）をきちんと抑えること
- お得意様や遠方のお客様の処理は早く
  - 割込みには優先度を設定する
  - キャッチホン（多重割込み）で優先度の高い割込みを先に扱う

キャッチホンをうまく使う - 多重割込み



多重割込み – お得意様は神様です



199

割り込み処理の定石

- 割込みで対応すべき入力／イベントを洗い出す
- それらについて性格を分析し、明確に定義する
  - 発生頻度、重要度 緊急度はどれくらいか
  - 発生したら必ず受けなければならないものか
    - マスク可能か、不可能か
  - 発生した場合の対応処理にはどのくらいの時間が必要か
    その時間のばらつきはどれくらいか オーバーしてしまう可能性はどれくらいあるか）
  - 割込み発生から処理開始までの遅延はどの程度許されるのか、そのマージンはどの程度か
  - 受け付けた後、メインの処理の性格が変わるようなものか
    - 受け付けて処理した後、元の仕事に戻るかどうか
    - 元の仕事に戻らない場合、元の仕事に与える影響はどういったものか
  - 割込み処理の中で使用するスタック量はどれくらいか
- 割込みルーチン内で処理すべきもの、メインの処理内で扱うべきものを明確にする

200

処理の実装

- 例えばエアコンのボタン操作を受け付ける処理は
  - ポーリングを繰り返してボタン押下を検知する
  - ボタン押下をハードウェアからの割り込みとして検知する
  - OSレベルに任せる

など、さまざまな実装方法がありえる。

Windowsでのボタンクリック処理など

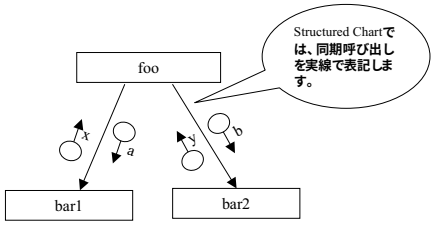
- 処理それぞれで必要とされる要件を明確にする
- 同期呼び出し、非同期呼び出しをうまく使い分ける

ことが重要

同期呼び出し

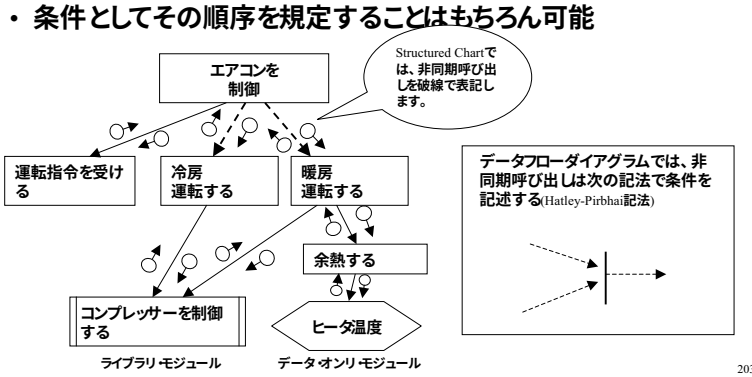
- シングルタスク処理のイメージ
- JISフローチャートのように、手順どおりに：
  - 呼び出しは、直ちにそれを実行開始することを意味し
  - 「呼び出す／結果を受け取る」を必ずペアで扱う　つまり
  - 先の結果を受け取るまでは、次の処理を呼び出すことができない

```
foo()
{
    ....
    x = bar1( a );
    y = bar2( b );
    z = x/y;
}
```



非同期呼び出し

- 処理の起動の直後に処理が必ずしも実施されない
  - 必要な条件がそろうまで待ち、条件が揃った時点で実行
  - 個々の処理は、その実行順序が規定されない
    - 条件としてその順序を規定することはもちろん可能



格言

締め切りは待ってくれない

へたな鉄砲は数を撃っても当たらない

7. volatile – 変数と最適化

- コンパイラの最適化によるおせっかい迷惑を防ぐための宣言
  - 例えば、周辺デバイスのレジスタの中身を確認する時、現在最新の状態を読み直す動きをさせるようにコンパイラに指示する機能。
  - 例えば、左のソースコードはコンパイラの最適化機能により右のように扱われる

```
int data;
.....
while(1) {
      if ((data&4) != 0) {
            処理1;
      }
}
.....
```

結果のあらかじめわかっている判定をループの度にいちいち実施するのは処理を遅くするだけ！

=

```
int data;
.....
if ((data&4) != 0) {
      while(1) {
            処理1;
      }
}
.....
```


最適化による予期せぬ振る舞い

- 周辺デバイスのレジスタをポーリングで読み込むような場合にvolatile宣言を付けないでいると、コンパイラによっては最適化オプションが災いして、再度レジスタを読むという作業を省いて機械語に変換してしまう。

```
const unsigned char *port;
port = 0xfd0c;
.....
while(1) {
      /* レジスタのbit2をチェック */
      if (((*port)&4) != 0) {
            処理1;
      }
}
.....
```

結果のあらかじめわかっている判定をループの度にいちいち実施するのは処理を遅くするだけ！

と、コンパイラは判断してしまう・・・

=

```
const unsigned char *port;
port = 0xfd0c;
.....
if (((*port)&4) != 0) {
      while(1) {
            処理1;
      }
}
.....
```

期待した動作をしない・・・

volatileの使用

```
volatile unsigned char *port;
port = 0xfd0c;
.....
while(1) {
      /* レジスタのbit2をチェック */
      if (((*port)&4) != 0) {
          処理1;
      }
}
.....
```

コンパイラは、volatile宣言された変数は最適化の対象外であると判断します。

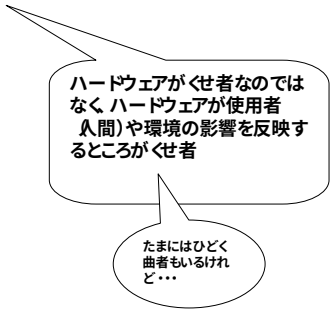
- ポートから読み込む値だけでなく割込み処理や他のタスクから書き換えられる可能性のある変数をチェックするような場合、volatile宣言の考慮が必要！

格言

シルクハットの中のハンカチは
突然ぞうきんに変わる・・かも

8. ハードウェアとのお付き合い

- ハードウェアは曲者だらけ
  - タイミング
  - 順序
  - 前準備、後処理
  - 本当にその値？？
  - ノイズとのお付き合い
  - 発熱があると・・・・

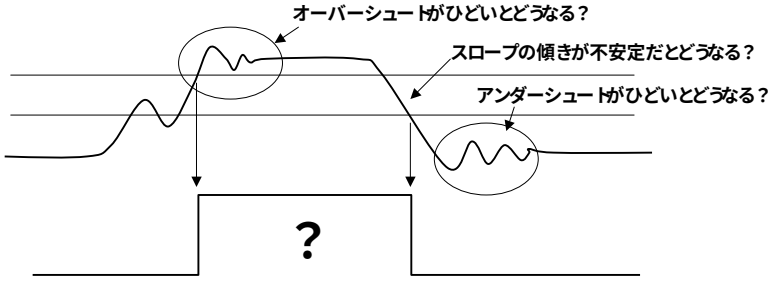


デジタルはアナログより簡単？

- デジタルの波形
  - 矩形波形は、無限の周波数（高調波）成分を含む
- デジタル信号の伝送路は、抵抗がゼロではなく、隣の配線とコンデンサ要素を形成したり、自分自身がコイル的な要素を持ったりする。
  - 交流的抵抗成分「インピーダンス」を持つ
- インピーダンスを持つことにより、信号線は高周波ほど伝送特性が劣化する、
- また、信号線は隣接する信号線上の信号に影響を与える（クロストーク）、つまり
- 信号線を通るデジタル波形は、理想的な形ではなく、崩れている。

デジタルはアナログより簡単？

- デジタルの「グレーゾーン」
  - デジタル回路も、内部はとてもアナログ的
  - デジタルはノイズに強いって！？
  - デジタルは時間に正確か？



また、ディジタル回路で用いられるパーツには、Hiレベルと Lowレベル 2つのしきい値があります。
各パーツのデータシートには、Hiレベル／Lowレベルの入出力電圧に関する記載があります。
中間レベルでは、Hi/Low不定です。

211

ハードウェアは確実？

- ソフトウェアが状態を出力したものが出力デバイスに直ちに反映される・・・・・とは限らない。
- 周辺デバイスなどのハードウェアの状態変化を直ちに 間違いなくソフトウェアで読み取ることができる・・・・・とは限らない。
- ハードウェアの状態は動作環境によって不変で安定している・・・・・とは限らない。

信号やハードの振る舞い、タイミングに関する信頼性がどの程度実現されるべきか、は、製品によって異なります。
ハードウェアの「癖」に対応するための定石をいくつも身につけておき、さまざまな信頼性要求に応えられるようになりましょう。

212

なめるなよ、ノイズ

- 内部で発生するノイズ
- 外部から侵入してくるノイズ
  - 電源から
  - アースを取ったラインから
  - 電磁波として
  - 人間を経由して
    - 誘導
    - 静電気
- ノイズの影響を受けやすい回路配置がありうるが、製品企画上ハード屋さんがそれ以外に選択できない場合がある
  - そのような場合は、ノイズによる誤動作を防ぐ判断ロジックをソフトの側で組む必要がある

温度によって、湿度によって、操作者によって、気分によって（らそ）、つど変わる

しかし、システムへの論理的な不安定要因であるのは間違いない

百推は一見にしかず

- 動作が不安定な場合、問題領域のソースコードや実際のスタックの状態を見直すのは当然ですが、ソースコードにミスが見つからない場合は、信号の状態をオシロスコープやロジックアナライザで実際に見てみましょう。
- 百の推測すべてに思いを致して悩むより、推測を絞り、その推測に基づいて「システム」を実際に覗いて見ましょう。

ハードウェア任せ、ではなく、より良い・信頼性の高い製品にするために、ソフトとして何ができるかを考えることは重要なことです。

格言

知彼知己　百戦不殆
彼を知り、己を知れば、百戦して危うからず）

トラブルは境界に潜む

自分と他メンバー
ハードとソフト
製品の内側と外側
チーム内と外
発注元と開発者 etc.

「不安定」には論理的な原因がある。
その原因をハードで対処できない場合は、ソフトで対処しなければならない

システム屋への道

- 組込みシステムは、メカ屋さん、エレキ屋さん、ソフト屋さんの努力の結果世に出ます。
- それぞれの担当領域でのエキスパートとなるのは当然として、しかし他の担当領域に興味を示さなかったり、「あとは君らの仕事、あっしの預かり知らぬことで」というのでは良い品質の製品を世に出すことは難しいでしょう。
- エキスパートにならないまでも、他の領域のシゴトが理解できるようになりましょう。
- 「ソフト屋育ち、メカ・エレキもそれなりに分かる」システム屋さんは、製品の要求分析やシステム設計の時に核となれる存在です。こういったメンバーが上流工程できっちり仕事をすると、その後の仕事が楽になると思います。
- ぜひ、システム屋を目指してください。

OpenSESSAME Seminor

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

2002 **年** 10 **月** 15 **日 初版 第** 1 **刷発行**
2003 **年** 10 **月** 29 **日 初版 第** 2 **刷発行**
2004 **年** 3 **月** 19 **日 第** 2 **版 第** 1 **刷発行**
2004 **年** 4 **月** 30 **日 第** 3 **版 第** 1 **刷発行**
2004 **年** 6 **月** 17 **日 第** 4 **版 第** 1 **刷発行**

著 者 上原慶子、大野晋、坂本直史、鈴木圭一、須田泉、西康晴、
二上貴夫、三浦元、三宅貴章、森孝夫、山田大介、山崎辰雄

編集・発行 組込みソフトウェア管理者・技術者育成研究会
（SESSAME）
http://www.sessame.jp
