メモリの アライメント

メモリのアライメントと リトルエンディアン、ビッグエンディアン は 直接的に干渉する事はないですが、物理的なメモリ上の書き込み位置に違いが出ます。（ 右図参照 ）

特に C言語の場合は、構造体のパック、アンパックと、メモリのアライメントは、密接に関わってくるので、しっかり理解する必要があります。

今回は、ちょっと視点を変えて ハード的な メモリに接続されるアドレスバス、データバスの観点で 説明してみようと思います。 シンプルな 16bitの例で、説明します。 まずは、リトルエンディアン、ビッグエンディアンの メモリ配置イメージを図1に示します。アドレスバスを 8bitのイメージで表現してますが、最下位bit は メモリ素子に接続されません。その関係で アドレスバスの 最下位bit を * で表しています。

図1

リトルエンディアン

アドレス バス 8bit	データバス 16bit	
	b15 Odd b8	b7 Even b0
0000 000*b	Address = 01h	Address = 00h
0000 001*b	Address = 03h	Address = 02h

ビッグエンディアン

アドレス バス 8bit	データバス 16bit	
	b15 Even b8	b7 Odd b0
0000 000*b	Address = 00h	Address = 01h
0000 001*b	Address = 02h	Address = 03h

マイコンの場合、アドレスは Byte単位で 付けてあるので、16bit バスの場合、b0 ～ b7 を 偶数アドレスとして扱うと、リトルエンディアンに なります。b8 ～ b15 を 偶数アドレスとして扱うと ビッグエンディアンに なります。 薄緑色を付けている箇所は、偶数アドレス側（ アドレスバスの最下位bit = 0 の側 ）です。

今回は リトルエンディアンで説明します。
0番地に ワードデータ 1234h を格納すると、図2の 並びで データが 格納されます。 この場合、アドレスが 0000 000*b に 2バイトのデータが、行儀よく 横一列に入っているので、1回の書き込みで格納できます。
読み出しも同様に、1回で読み出せます。
このような状態を メモリアライメントが 正しくとれている という事になります。

次に、1番地に ワードデータ 1234h を格納すると 図3の並びになります。 この場合は、アドレスバスの 0000 000*b と 0000 001*b に跨って段違いになった状態で格納されます。 この状態を メモリ境界を跨いだ状態といいます。 メモリアドレスが 奇数番地でワードデータを、書き込もうとすると CPUによっては 例外割り込みが、発生したり、例外が発生しなくても 1番地に書いたつもりのデータが、0番地に ズッコケる場合があります。 パソコンの x86系CPUでは、メモリ境界を跨いだデータ書き込みでも 正常に書き込んでくれます。

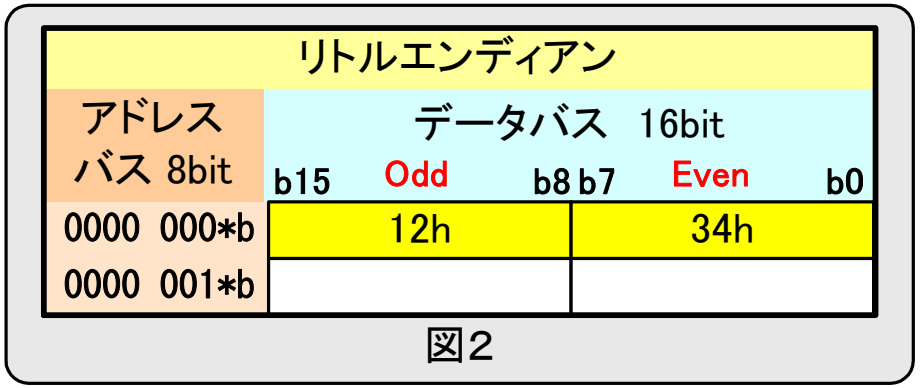


図2

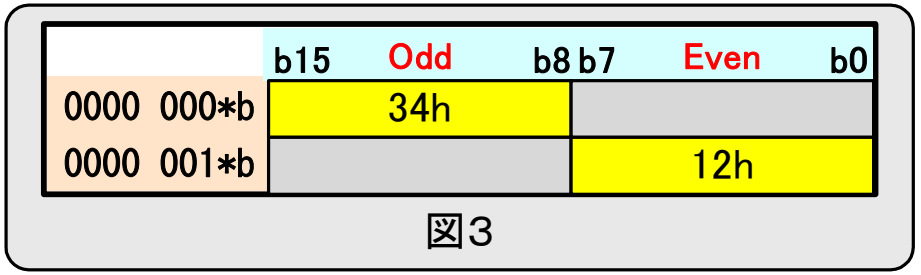


図3

但し、メモリ境界を跨いでいると、2回メモリをアクセスすることになり、アクセス速度が低下します。

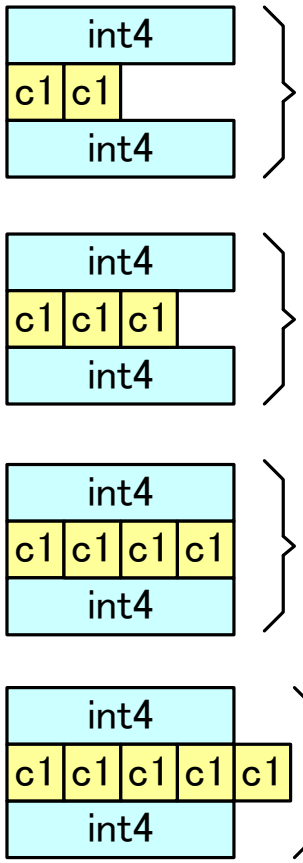
32bit CPUの場合は、メモリ境界が 4Byte 単位で発生します。32bit CPUで、16bitデータを書き込む場合 奇数番地でも、メモリ境界を跨がなければ、1回で書き込めます。例) 1番地は 1回、3番地は、2回です。

ESP32の　メモリアライメント

　メモリのアライメントの話も　ややこしい話でしたが、凡そ理解出来ましたでしょうか。？

　では、これからは　32bitの ESP32の話をします。 実は、C　C++コンパイラは、賢くて　先ほどのメモリ境界を跨がないように、変数のアドレスを調整してくれます。 もう少し具体的に言うと　変数の先頭アドレスが、4の倍数に なるように調整します。 で、今回の実験は　Cコンパイラでよく問題になる構造体データのパック、アンパックも兼ねて説明します。

　今回の実験は　構造体内で宣言した総バイト数が、1つずつ異なる構造体データを　4つ用意します。右の図の構造体データです。
4byte整数を 2つ置き　その間に char 配列を 2byte、3byte、4byte、5byte と　順に用意します。




```
int  i1;
char tx[2];
int  i2;
```
Block_2

```
int  i1;
char tx[3];
int  i2;
```
Block_3

```
int  i1;
char tx[4];
int  i2;
```
Block_4

```
int  i1;
char tx[5];
int  i2;
```
Block_5

　上記構造体データの、構造体のサイズを sizeof 演算子を使い　byte単位の サイズを確認します。

	変数を足し合わせたByte数
Test ー Start.	
Size of Block_2 ＝ 12	4 + 2 + 4 = 10
Size of Block_3 ＝ 12	4 + 3 + 4 = 11
Size of Block_4 ＝ 12	4 + 4 + 4 = 12
Size of Block_5 ＝ 16	4 + 5 + 4 = 13
Test ー End.	

一番上の Block_2 の 上記の構造体データの tx配列が 2byteなので そのまま 後ろの整数 i2 を 連結すると、4byteのメモリアライメントに i2が 納まらず、2byte前に ずれた形になるので、tx[2]の 後ろに 2byteの Padを コンパイラが入れ込む事で アライメントを調整しているという事です。 アドレスの前方向には、領域を確保された変数群が並んでいるので、アライメントを跨ぐ変数は、メモリの後方に移動させられます。 一番下の Block_5は tx[5]が

Block_2
```
int  i1;
char tx[2];
int  i2;
```

Block_4
```
int  i1;
char tx[4];
int  i2;
```

Block_3
```
int  i1;
char tx[3];
int  i2;
```

Block_5
```
int  i1;
char tx[5];
int  i2;
```

メモリ境界位置より、1byteはみ出しているので 3byteの Padを 挿入して 後ろの int i2 が アライメントの境界内に納まる様にした。という事です。よって、変数を足し合わせたbyte数が 10、11、12、13 に 対して 構造体のサイズが 12、12、12、16 と 4の 倍数に なっています。

今回の例でいくと、構造体は アンパックという事に なります。 右側の 10、11、12、13 に構造体サイズが、なる場合は 構造体は、パックになります。

タイマー割込みに関して

前々回にも、割り込み処理を取り上げましたが、今回は定周期の時間で、割り込みが、かかるタイマー割り込みです。 このタイマー割り込みは、家電製品を含めた組み込み分野や計測制御分野では よく使われます。 一定の周期で、外部事象データを取り込み そのデータを収録、または 取り込んだデータを 元に演算判定処理を行い制御を行う場合が あります。 あるいは、広域的に必要なデータであれば、ローカルにデータを収録すると同時に、別の場所にデータ転送する場合もあります。

まずは、定周期というか一定のサンプリングレイトで、データを 取り込みます。 昔は、コンピュータでサンプリングするデータというと、アナログ信号のデータが多かったですが、最近はセンシングデバイスが I2Cインタフェースで接続できる物が 増えてきているので、便利になりました。

ちょっと 余談ですが、I2Cのデバイスを 使用する際（ SPI も そうですが ）元々は 同じ基板上で使用する事を想定したデバイスなので、インタフェースの信号を 長々と伸ばして使用すると、誤動作する危険性が高いです。

どのくらい配線を伸ばせるのかというのは、ケースバイケースで 何とも言えませんが 凡そ30cmぐらいが限度と考えた方がいいと思います。 あと、SCLや SDAの信号線の末端に 終端抵抗を付けますが、初期の頃は 2.2KΩとか本に書かれてました。 終端抵抗は、抵抗値が低くなる方が、波形の立ち上がりが、早くなりますが、あまり抵抗値を 低くすると マイコンやデバイスの負担になるので 低くしても 1KΩまでにして下さい。 余談でした。

タイマー割込みの ライブラリ関数 概要

事前の準備:

タイマー割り込みに関わる関数は、ライブラリ化されていますが、何種類かあるようです。
訳あって、2種類試してみましたが、最初に 取り扱ったライブラリが、最終的に 扱いが簡単で
安定していた uTimerLib を、今回 使う事に しました。ツールの ライブラリを管理をクリックして
ライブラリマネージャから インストール して下さい。uTimerLibのバージョンは 1.7.2 でした。

① タイマー割り込みプログラムの登録と サンプリングレイトの設定を 行う関数:

まず、最初に呼び出す関数です。

void TimerLib.setInterval_us(割り込み処理関数名、マイクロ秒単位のインターバル値);

但し、インターバル値は ミリ秒単位の設定で、ミリ秒以下の 値は切り捨てられるようです。

② タイマー割り込みを 止める関数。

タイマー割り込みを 止める時に使用する関数です。

void TimerLib.clearTimer(void);

途中で止める必要があれば、使用して下さい。

この、2本のメンバー関数だけです。 シンプルですね。

あとは、今回のテストプログラムを お見せします。

追記:
要は、ミリ秒(1/1000秒)分解能のタイマから、分周して呼び出されるようです。
5678 と設定しても 678は 切り捨てられて 5000 に なります。

Timer_IRQ.ino

```
#include <uTimerLib.h>

#define led_pno2  2      //  赤LEDポート (2)
#define led_pno4  4      //  緑LEDポート (4)

static  volatile char   led;
```

プログラム Timer_IRQ.ino の 先頭部分です。
まず、インクルードファイル uTimerLib.h を 呼び出しています。

次に、GPIO2に 赤のLED、GPIO4に 緑の LEDを 接続しています。 #defineで led_port2 と led_port4 を 宣言しています。

char ledという名前の変数を宣言しています。
予約語 volatile で この変数に対するオプティマイザの 最適化を 停止させています。

```
//**************************
//**  タイマー割込み処理 **
//**************************
void  IRAM_ATTR  blink( void )
{
  led++;
  if( (led & 1) == 0 )     // LED変数 最下位bitは 0 か？
  {
    digitalWrite( led_pno2, LOW );     //  赤：消灯
    digitalWrite( led_pno4, HIGH );    //  緑：点灯
  }
  else
  {
    digitalWrite( led_pno2, HIGH );    //  赤：点灯
    digitalWrite( led_pno4, LOW );     //  緑：消灯
  }
}
```

割り込み処理の中では、GPIOのアクセスは　問題ないと思いますが、シリアル通信は、割り込みを使用しているため　呼び出してはいけません。

タイマー割り込み処理関数 blink(void) です。 割り込み処理関数は、メイン処理関数とは、全く非同期に　呼び出されますので、引数も渡せませんし、関数値を戻す事も出来ません。 よって引数も、関数値も　void です。
IRAM_ATTRは、RAMメモリのキャッシュ領域ではなく、スタティクな　RAM領域に配置されます。

この関数内でやっている事は、変数 ledを インクリメントして、最下位 bit が　ゼロか　否かにより赤と緑のLEDを　交互に点滅させています。

```
void setup() {
  // put your setup code here, to run once:
  led = 0;                    // LED点滅カウンタ初期化
  pinMode( led_pno2, OUTPUT );    // 赤LEDポート出力に設定
  digitalWrite( led_pno2, LOW );  // 赤LEDポートに LOW出力
  pinMode( led_pno4, OUTPUT );    // 緑LEDポート出力に設定
  digitalWrite( led_pno4, LOW );  // 緑LEDポートに LOW出力

  TimerLib.setInterval_us( blink, 500000 );
}

void loop() {
  // put your main code here, to run repeatedly:
  // TimerLib.clearTimer();
}
```

setup関数は、初期化処理だけです。

２つの LEDポートの 出力指定と、出力を LOWに設定しています。

そして割り込み処理登録がTimerLib.setInterval_us()関数で割り込み処理関数名　blink と、割り込み周期 500000(0.5秒)を設定しています。

loop関数は、何もしていません。

次は、動画をお見せします。