XIAO ESP32 S3で　発生した不具合

前回、XIAO　S3で MicroPythonファームウェアを書き込んだ後、PCから　USBケーブルを抜いて　再度 PCに差し込むと、USBの接続が切れる音が、1～2秒周期で鳴りだしました。

再度、MicroPythonファームウェアを 書き込むと　鳴りやみました。 という事で何らかの理由で　ファームウェアが壊れたと思われます。

で、2回目は　ファームウェア書き込み後、連続して　Arduino IDEにて 通常のプログラムを書き込んでいたのです。 その状態で　USBケーブルを引き抜き、再度差し込むと ファームウェアは、壊れません。最後に書き込んだアプリが動いてます。

これは　もしかしてと思い、再度、ファームウェア書き込み後、PCからUSBケーブルを引き抜き　再度　PCに接続すると、USB接続が切れる音が、また 1～2秒周期で鳴り始めました。

これは、ファームウェア書き込み直後は、MicroPythonファームウェアだけが、Flashメモリに書き込まれていて、アプリに相当するプログラムが、無い状態にあります。

この状態で、USBケーブルの抜き差しにより、CPUリセットが かかり、アプリに相当するプログラムを　実行しようとして、無いので暴走して、Flushメモリを 壊したと、思われます。

よって、ファームウェア書き込み直後、連続して何でもいいですから、アプリに相当するプログラムを Arduino IDEにて 必ず書き込んで下さい。

これにより、プログラムの暴走を　防ぐ事が出来ます。　尚、XIAO C3においては、同様の事をしても、ファームウェアが壊れる事は、ありませんでした。 多分、CPUコアが　S3は　Xtensa LX7で、C3は Risc Vであるため、暴走の状況が異なるのでしょうね。　以上でした。

XIAO ESP32 C3の　10KΩ抵抗取り付け

　前回は、ブレッドボード上で　10KΩを接続しましたが、毎回ブレッドボードでやるのは、面倒なので、XIAO ESP32 C3のモジュール基板にハンダ付けしました。　モジュール基板裏側に略した信号名を付けてありますが、3V3と　D8の間に 10KΩの抵抗を　付けます。

右の写真では、抵抗は 1/6Wを使用してます。
　あと、青い積層セラコンを　VUSBと　GND間に 接続してますが、これは　無理に付ける必要は ありません。

esptool.exeに　関して

　前回は、私も esptoolに関して不十分な理解で　扱っていたので、本質をきちんと説明出来てなかったと、思います。
　MicroPythonを　ESP32に書き込む機能は間違ってないと思いますが、独立したユーティリティというだけでなく、Arduino IDEと協調してESP32に　アプリを書き込んでいます。

i さんの説明によると esptool.exe は Pythonのインタプリタと　利用するライブラリパッケージとソースを　圧縮してEXEにまとめる　Pybuilderというツールがあります。

このツールにより　esptool.exe は　作られている　との事です。

では、本題 割り込み処理に　入ります。

割り込み処理ですが、初心者の方は、割り込み処理のイメージが掴めるでしょうか。？

たとえば、事務所で　だれかが　パソコンに、データ入力の仕事を　していたとします。

その時、電話の呼び出し音が鳴ったら、データ入力の手を休め、受話器を取り　電話対応を行います。　電話対応が終わったら、またデータ入力の仕事の続きを行います。

この時の電話対応の処理こそが、割り込み処理という事です。　そして、その前後のデータ入力処理は、割り込み処理中は、止まっていますが、何事も無かったかのように、データ入力処理は、続けられます。　通常の割り込み処理は、メインの作業に　影響を及ぼさないように迅速に（ 超短時間に ）実行されます。

場合によっては、メインの流れに影響を及ぼす割り込みもあります。　それらは、例外割り込みといって、例えば割り算で、分母が 0 の割り算を行った場合、演算出来ないので 例外割り込みになりメイン処理が、即座に中止されます。

特権命令の例外割り込みもあります。　これはWindowsのような　OS上で動作するアプリで、OSカーネルしか実行できない特権レベル　0 でのみ実行できる命令を 特権命令といいます。

これを、特権レベル 3 の アプリが　特権命令を実行しようとすると、即座に　OSにトラップする事になります。　組み込みマイコンでは　当たり前に使用している IN命令、OUT命令は Windows上では、特権命令で使用出来ません。

Linuxでは、一部の IOポートを 解放している様で、条件付きで IN、OUT命令が　使えます。

割り込みの種類

大きく分けてハードウェア割り込みと、ソフトウェア割り込みが あります。

ハードウェア割り込みは、内蔵周辺回路または、CPUモジュール外部に接続したデバイスからの、割り込み信号を使用した割り込みです。

例としては、インターバルタイマー回路の割り込み、シリアル通信の受信処理に使用されます。 インターバルタイマーは 歯切れいい値として、1/1000秒周期で 割り込みを発生させる場合が 多いと思われます。

シリアル通信の場合、受信処理には 割り込みが使用されます。 特に早いボーレイトの場合、割り込み処理を使用しないと、受信文字の欠落を発生させてしまいます。 送信の場合は他に忙しい処理がある場合、割り込みを使用した方がいいと思われます。 CPUによっては DMA機能を使用して、シリアル通信を行う事が出来る CPUが ありま す。

ソフトウェア割り込みは、古い話ですが、MS-DOSにおいて INT 21H の ソフトウェア割り込みを 使用して DOSのファンクションコールを 行っていました。 あと、デバッガの支援機能として、シングルステップ実行機能と、ブレークポイント割り込み命令が ありました。

割り込みには、マスク可能な割り込みと マスク不可能な割り込みもあります。 通常は マスク可能な割り込みを使用します。 マスク不可能な割り込みは NMI(NonMaskable Interrupt)と呼ばれ、システム全体に 致命的な障害をもたらす緊急事態に使用されます。マスク可能な複数の割り込み信号線を持っている CPUは、各信号線に 優先順位を設定できて、多重割り込みのレベル設定をサポートします。

あと実際に、多重割り込みの優先順位を決めるのは、やや難しい要素もありますが

仮にインターバルタイマーと、シリアル通信の受信処理の2つの割り込みであれば、シリアル通信のボーレイトが、100 kbpsを超える　極端に速い場合は、シリアル通信の受信処理の優先順位を早くした方が よいと思われます。

シリアル通信のボーレイトが、38400bps程度までなら、シリアル通信の受信処理と、インターバルタイマの 両方を　同じ優先順位で、使っても、私の場合　百円 R8Cマイコンで、問題ありませんでした。 たまたま動いていた　という事ではなくて、各　割り込み処理の入口で、LEDを点灯させて、割り込み処理の出口でLEDを消灯させます。 その2つのLEDの信号を、オシロスコープで2現象で観測して、時間的余裕度を、観測したという事です。

割り込み処理の　実際の処理時間を観測する用途で、オシロスコープは、よく用います。

割り込み処理による設計のポイント:

① 割り込み処理時間は出来るだけ、短時間に終わらせる。 割り込み周期に対して、割り込み処理時間が半分以下になるようにします。上記が無理な場合でも、割り込み処理時間は、割り込みの周期を超えてはいけません。
　処理が終わっていないのに次の割り込みタイミングになると、割り込み以外の処理が出来ず　システムが　破綻してしまいます。

② 割り込みの優先順位と、多重割り込み。
割り込み周期が短い場合や、割り込み処理が遅れると制御対象に影響がある場合は、割り込みの優先順位を上げて、多重割り込みを　検討します。 多重割り込みを　許可していない場合は、割り込みの処理途中に、

多重割り込みを　許可していない場合は、割り込みの処理途中に、異なる割り込みが、発生しても、先行する割り込みの処理が、終わるまで、次の割り込みは待たされてしまいます。

割り込まれる側での 注意事項

割り込む側の話を　中心にしてきましたが、割り込まれる側のプログラムにおいても、一部注意が必要となります。

よくある処理として、割り込み処理に同期してメインループのプログラムを走らせる場合があります。 その場合　割り込み処理から、メインループに渡される　フラグ変数が　あります。このフラグ変数は、変数宣言先頭に volatile を付けて下さい。オプティマイザに 改ざんされないようにするためです。 それと クリティカルパスの問題も あります。

クリティカルパスは、何らかの事例で 表した方が、分かりやすいと思うので、シリアル通信の受信処理と組みにして使用するリングバッファを 例に説明します。

リングバッファは　FIFOバッファ(先入れ先出しバッファ)で、書き込むタイミングと、読み出すタイミングが　ずれても、256byteほどのバッファに　受信文字列を貯め込んでおけるので、慌ててデータを 読み出す必要は　ありません。

リングバッファを構成する変数は

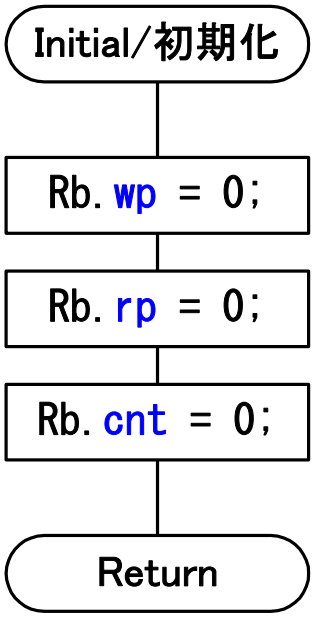
```
typedef  struct {
  unsigned char  buf[256]; // バッファ
  unsigned char  wp;     // 書込み位置
  unsigned char  rp;     // 読出し位置
  unsigned char  cnt;    // 格納byte数
} Ring_Buffer;
```

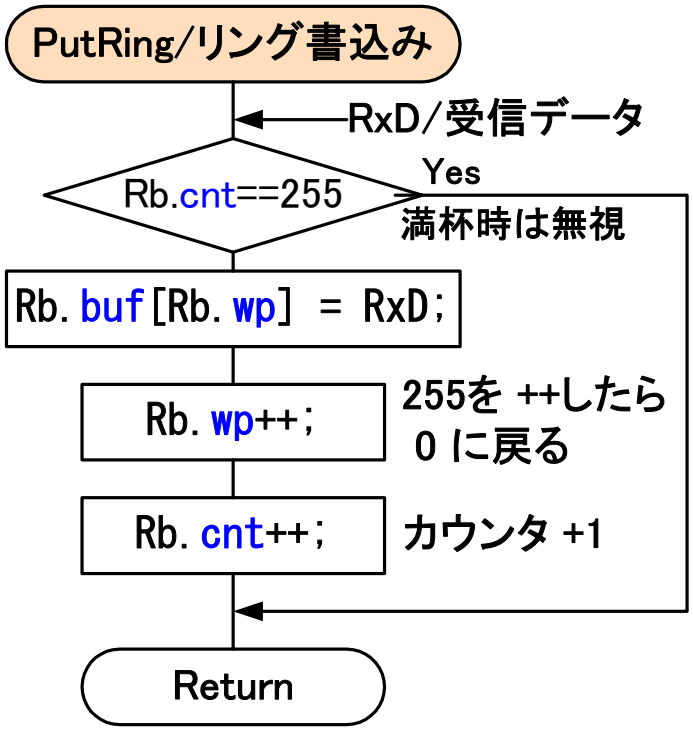
に なります。

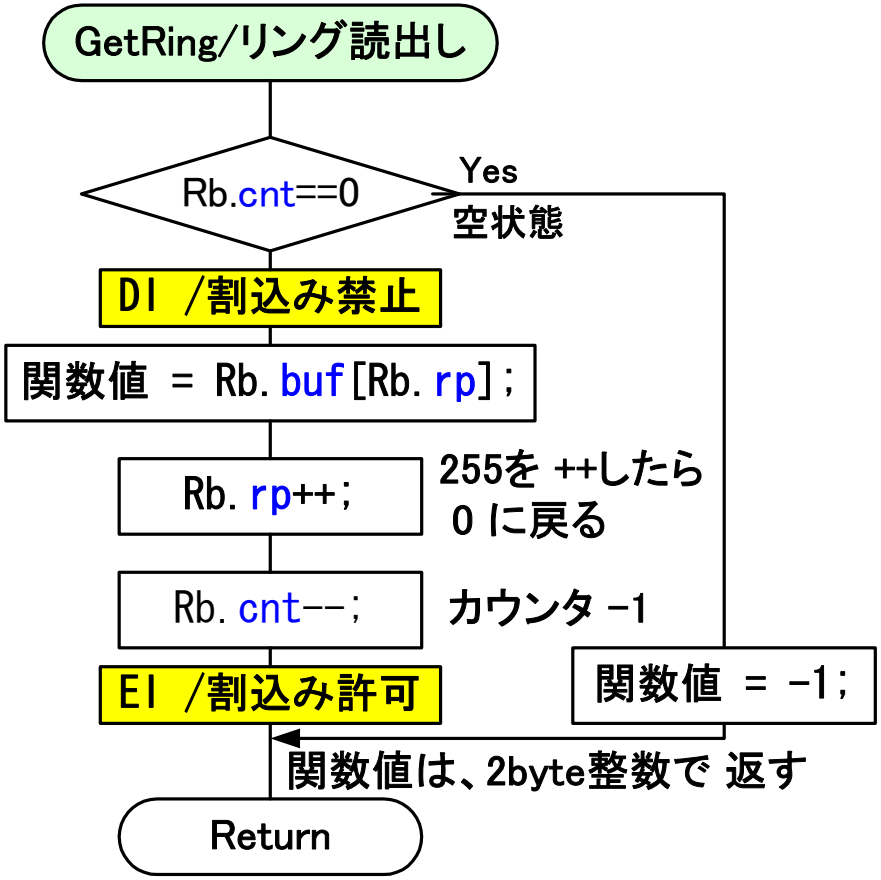
リングバッファ アクセス関数

ちょっと横道に逸れますが、リングバッファのアクセス関数について、フローをお見せします。
やっている内容は、簡単です。

```
Ring_Buffer  Rb;  // リングバッファ変数宣言
```





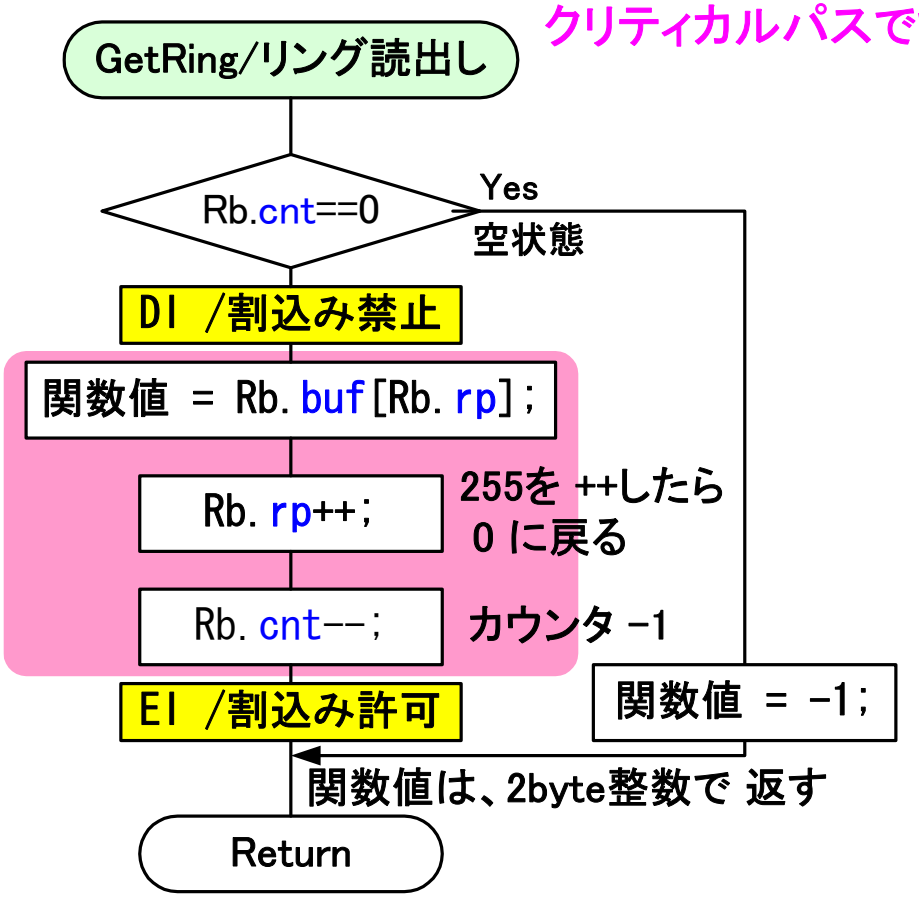


薄い赤色のリング書込みは、シリアル受信 割込み処理内にて使用されます。 薄緑のリング読み出しは、メインループ内で 呼び出されます。

リングバッファ アクセス時のクリティカルパス

先ほどの、リング読み出しのフローを、またお見せします。

ピンク色の部分は
クリティカルパスです。



リング読み出しルーチンの クリティカルパスの部分は、DI/割込み禁止から、EI/割込み許可の間です。 割込み禁止と 割込み許可の機能が無いと、リング読出し中に、クリティカルパス部分実行中に、シリアル受信割込みが、発生する場合があります。そして、リング書き込みが呼び出され、特に共通に使う cnt という変数のつじつまが、合わなくなります。 もう少し具体的に説明すると、左のフローでは cnt--を やってますが、アセンブラレベルで見ると、

① cntという変数をCPUのレジスタに読み出し。
② レジスタの値を デクリメントする。
③ デクリメントされたレジスタ値を cnt変数に、格納する。

という3段階を 実行します。

で、仮に cntが 最初 10 で、デクリメントして 9 に なった値を cnt変数に格納する訳ですが ③の cnt変数に値を書き込む直前に、受信割り込みが発生すると、どうなるかという事です。

リング読み出しルーチンの ③の段階で、9 を cnt変数に書き込もうとしていたタイミングで、シリアル受信割込みで、リング書込みルーチンが呼び出されると、RxDデータを リングバッファに書き込み、wp++ をして cntも ++ します。
具体的には、メインルーチンのリング読み出しルーチンで cntは ③の 9 を書き込む手前で、割込み処理に飛んだので cnt変数は、まだ 10のままです。 その cntを ++して、割り込み処理内で cnt=11 に変更されます。 その後 メインループの リング読み出しの ③を実行し 11 になっている、cntを 9 で 上書きしてしまいます。

結果として 1byte リングバッファ処理内で、受信データを消失してしまいます。このような障害が発生する恐れのある箇所を クリティカルパスと呼びます。

そのような障害の発生を 防止するため、DI/割込み禁止と、EI/割込み許可が 必要となります。

クリティカルパスの区間では、割込み処理が割り込まないようにしているという事です。
割込み処理を扱う時は、このような クリティカルパスの有無を、考慮する必要が あります。

通常、シリアル通信はライブラリで完備してあるでしょうから、特に考慮する必要はありません。但し、ライブラリに存在しない 特殊なデバイスを接続する場合は、自前で割り込み処理を作成する必要に迫られる場合が あります。

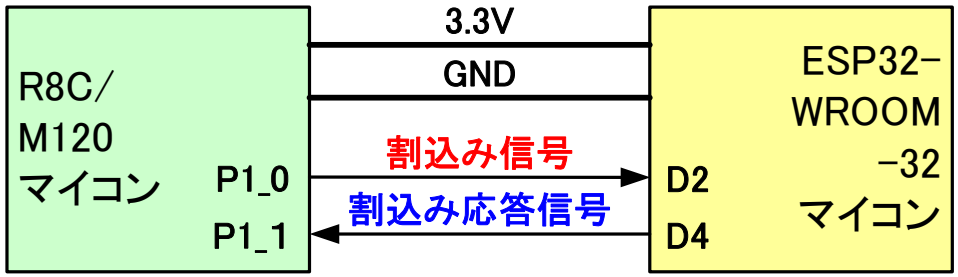
ややこしい話で すみませんでした。

今回の割り込み処理実験の概要

今回、ESP32を使って GPIO端子から、入ってくるデジタル信号を 割込み信号として受付けて割込み処理ルーチンを　呼び出す実験をします。 で、今回の場合、割込み信号となる信号を出す物を、別途用意しないといけません。

柔軟性を考慮して、別途 割込みテスト信号出力用のマイコンを　用意します。
殆どパルス発生器なので　百円 R8Cマイコンを使います。R8Cマイコンは 5Vでも　3.3Vでも使用できますので、今回は ESP32に合わせ、3.3Vで使用します。

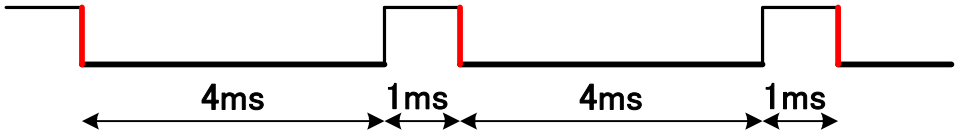
今回、パルスを　周期的に出し続ける仕様にします。 こうする事により、パルス波形を、オシロで観測しやすいというメリットも あります。



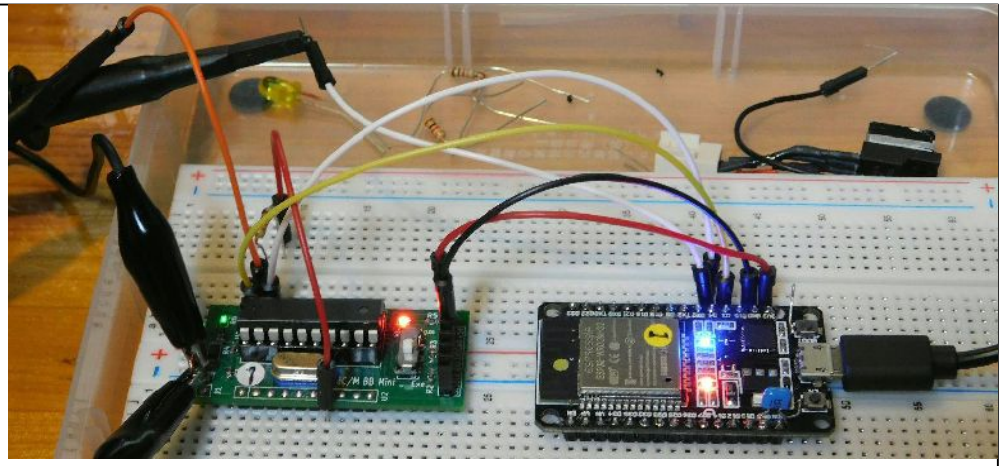
接続は、ブレッドボード上で、上記の配線を行っています。 電源は ESP32の USBケーブルで、5Vを ESP32基板に供給し、基板上の 3.3V三端子電源IC出力を、R8Cに分配してます。 R8Cは、ESP32に比べ　一桁遅いマイコンなので、その分 消費電力は小さいです。 割込み信号、割込み応答信号は、Low Activeとします。 且つ ESP32の D2端子での 割込み受付は FALLING (Highから Lowに変化した エッジ検出) です。
割込み応答信号て、何。？　という事になりますが、通常周辺回路には、CPUの割込み処理先頭にて　割込み信号を解除する機能が ありま す。 その模倣です。

割り込み信号の タイミング設計

まず、割り込みパルス発生器の方ですが、パルス出力は 5ms周期で 応答信号が戻って来ない場合、4ms経過したら 出力側で、パルス出力を 解除します。 よって Low Activeなので、Highから Lowに出力パルスが 落ちた時点で、割込みが発生した事を模倣しています。
パルスを Lowに落としたままでは、次の割込み信号が出せないので、4ms経過して、相手側（ ESP32 ）から、応答が無かった場合、パルス出力側にて、パルス出力を Highにして 次の割込み出力に備えます。
ESP32からの 応答が無かった場合の、タイムチャートを 以下に示します。

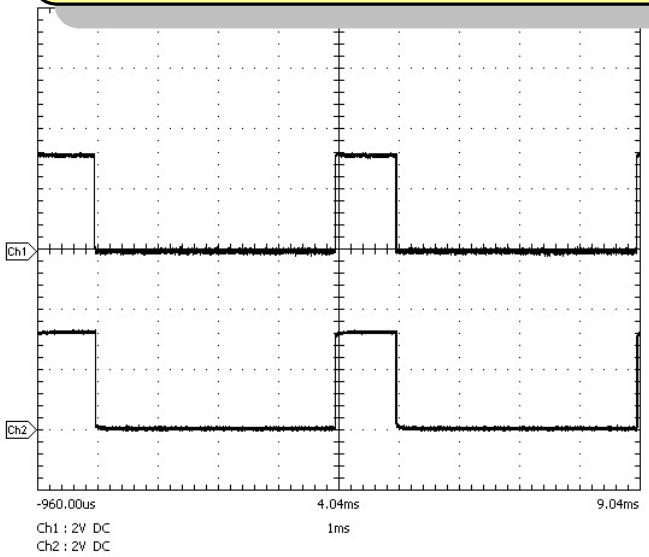


赤線が 割り込みが発生した タイミングとなります。



割り込み信号を受け付けるESP32の方ですが D2端子が、Lowになっているか確認して、Lowであれば割り込み処理を始めます。 まず、割込み応答信号を D4端子から Lowを出力します。 次に、メインループに渡すフラグに Trueを設定します。 次に D2端子が、Highになったか確認し続けます。 Highになったら ループから抜けます。
D4 割り込み応答信号を Highにします。 本来であれば、この下に、何らかの割り込み処理が、あるはずですが、今回は 特にやる事は無いので 割込み処理から、リターンします。

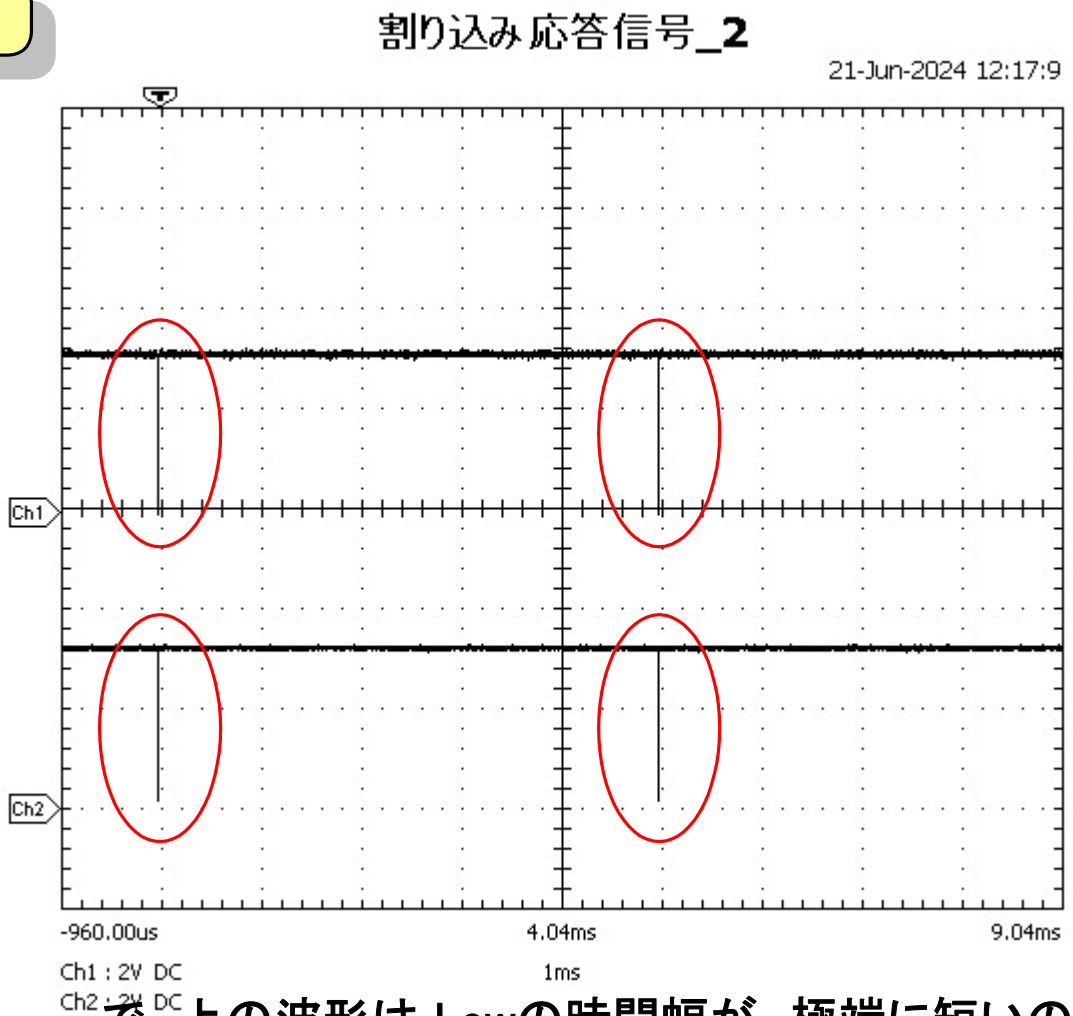
割込み信号、割込み応答信号の波形



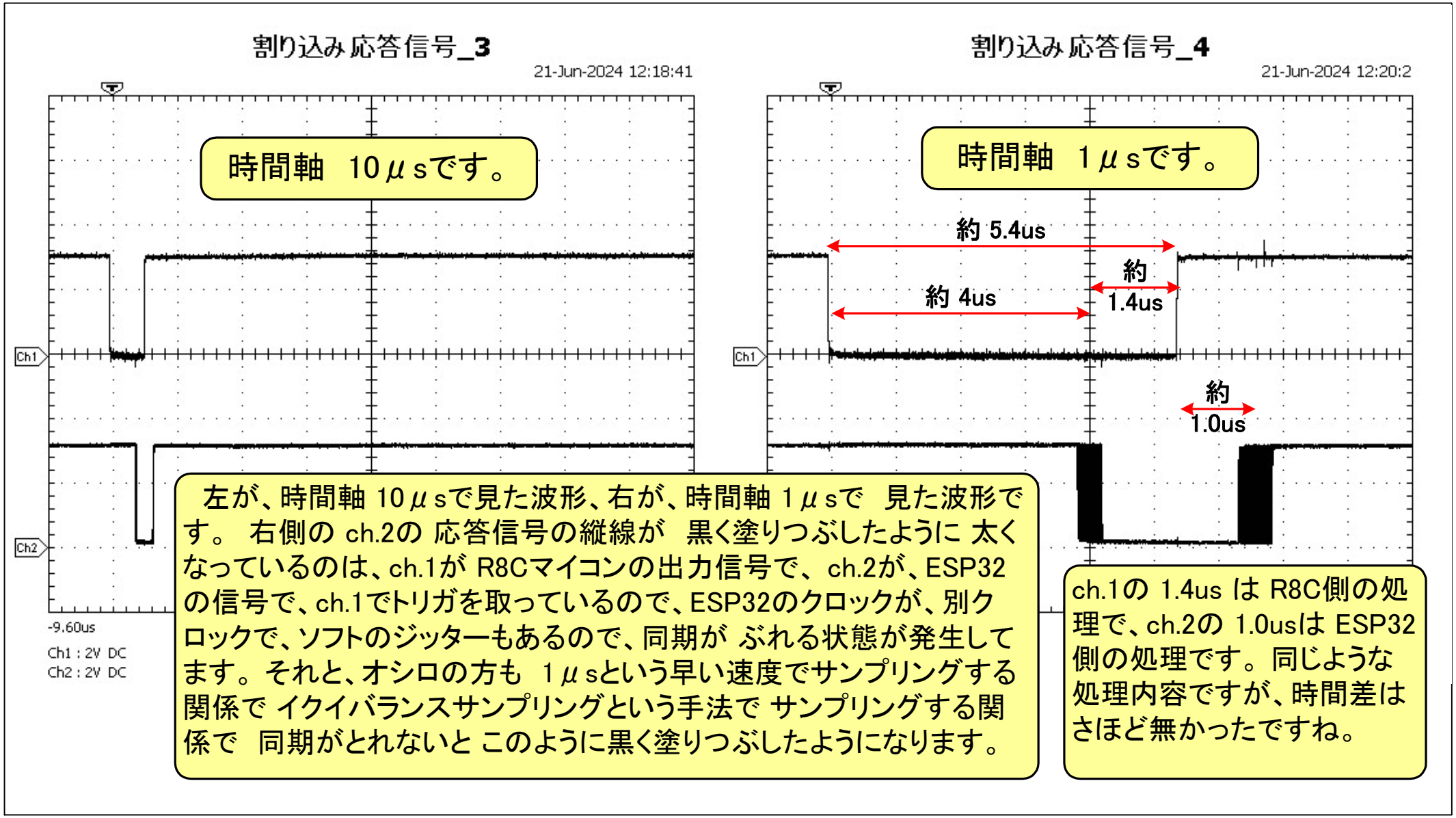
上が、ESP32のD2に入っている割込み信号で

下が、D4から出力される割込み応答信号です。

上の波形は、横軸(時間軸) 点線のひとマスが、1msです。 右の波形も 同じ時間軸 幅です。で、上の波形は、ESP32から、割込み応答信号が、出ていますが、ブレッドボード上で、ジャンパ線を引き抜き 応答信号を R8Cに 返してません。 その関係で 上下同じような 4ms 幅の波形が出ています。 で 右の波形は R8Cに 応答信号を返した波形です。



で、上の波形は Lowの時間幅が 極端に短いので、細い縦線のように見えます。


割り込み応答信号_3
21-Jun-2024 12:18:41
時間軸　10μsです。
Ch1
Ch2
-9.60us
Ch1 : 2V DC
Ch2 : 2V DC
割り込み応答信号_4
21-Jun-2024 12:20:2
時間軸　1μsです。
約 5.4us
約 4us
約 1.4us
約 1.0us
Ch1
左が、時間軸 10μsで見た波形、右が、時間軸 1μsで　見た波形です。 右側の ch.2の 応答信号の縦線が　黒く塗りつぶしたように 太くなっているのは、ch.1が R8Cマイコンの出力信号で、ch.2が、ESP32の信号で、ch.1でトリガを取っているので、ESP32のクロックが、別クロックで、ソフトのジッターもあるので、同期が ぶれる状態が発生してます。それと、オシロの方も　1μsという早い速度でサンプリングする関係で イクイバランスサンプリングという手法で サンプリングする関係で　同期がとれないと このように黒く塗りつぶしたようになります。
ch.1の 1.4us は R8C側の処理で、ch.2の 1.0usは ESP32側の処理です。同じような処理内容ですが、時間差はさほど無かったですね。

ESP32側 今回のソース

```
// 使用 IOピン
#define IRQ_sig 2         // 割込みポート D2 使用
#define IRQ_Active  LOW   // 割込みポート状態有効
#define IRQ_Idle    HIGH  // 割込みポート状態無効
#define LED_Red 4         // LED赤 Port

// 外部信号による割り込み 受付 Flag
volatile boolean ExIRQ_flag = false;

// 割込み処理プログラム
void IRAM_ATTR on_port_irq( void )
{
  char flg;

  ExIRQ_flag = false;      // 仮初期化
  flg = digitalRead( IRQ_sig );
  if( flg == LOW )
  {
    digitalWrite( LED_Red, LOW );   // 赤LED 点灯 ( 割込み応答信号 )
    ExIRQ_flag = true;    // メインループに割り込みが 発生した事を 伝えるフラグ
```

volatile は、割込み処理により、変更を加えられる可能性のある変数である事を宣言してます。 これにより、オプティマイザの最適化を、この変数に対して行わないように指示します。

IRAM_ATTR は、この on_port_irq関数は 割込み処理なので、メモリスワップは行わず 常時 RAM上に 固定的に 配置する宣言です。

```
    while( 1 ) // 外部との間で 簡易なハンドシェークを行う
    {
      flg = digitalRead( IRQ_sig );
      if( flg == HIGH )
      {
        digitalWrite( LED_Red, HIGH );   // 赤LED 消灯
        break;
      }
    }
  }
}

void setup()
 {
  // put your setup code here, to run once:
  pinMode( IRQ_sig, INPUT_PULLUP );
  pinMode( LED_Red, OUTPUT );
  digitalWrite( LED_Red, HIGH );   // 赤LED 消灯 ( 割込み無し )
  attachInterrupt( IRQ_sig, on_port_irq, FALLING );
}
```

attachInterrupt(IRQ_sig, on_port_irq, FALLING); は
割込み処理関数を システムに登録する 割込み処理 登録関数です。
第1引数 IRQ_sig は GPIOの番号です。 今回は2番です。
第2引数 on_port_irq は 割込み処理関数の関数名です。
第3引数 FALLING は、信号の立ち下がりで、割り込みを発生させます。

```
void loop()
{
  // put your main code here, to run repeatedly:
  boolean sw;

  sw = false;
  noInterrupts();    // 割り込み禁止
  if( ExIRQ_flag == true )
  {
    sw = true;
    ExIRQ_flag = false;    // フラグ無効化
  }
  interrupts();    // 割り込み許可

  if( sw == true )
  {
      // やや時間のかかる処理をやらせる
  }
}
```

メインループ内の処理です。
noInterrupts 関数が 全ての割込み処理を禁止する関数です。
interrupts 関数が 禁止した割込みを 許可する関数です。
この2つの関数で 挟んでいる箇所は クリティカルパス という事に なります。
この２つの関数は、Arduino UNOと 同じ関数名です。互換性を 持たせてあるようです。

あと、ExIRQ_flag == true は、メインループに割込みがあった事を 通知するフラグです。
割込み処理は、極力短時間で処理を終了する必要があるので、時間の ややかかる処理は、メインループ側で やらせるという事です。