<table>
<tr><td colspan="3">Raspberry Pi Pico SPI スレーブ プログラミング(1)（公開資料）
(Raspberry Pi Pico SPI Slave Programming (1))</td><td rowspan="4">作成 2021/12/6
ニューラルソフト有限会社 市来 博記</td></tr>
<tr><td>初版</td><td>－</td><td>2021/12/6</td></tr>
<tr><td>B</td><td>spi_write_blocking への対応は不要であった為、削除した。</td><td>2021/12/15</td></tr>
<tr><td>C</td><td>PDF にメタ データを追加した。</td><td>2021/12/25</td></tr>
</table>

概要

本書は、Raspberry Pi Pico を SPI スレーブ デバイスとして動作させるプログラムの作成方法について調査した結果を記したものです。

調査内容

調査項目を示します。

1 SPI スレーブ デバイス開発用プロジェクトの作成方法
2 SPI スレーブ構成手順
  2.1 SPI マスタとスレーブ間の結線
  2.2 SPI 構成処理の実装方法
3 SPI 全二重通信処理の実装方法

情報源

Getting started withRaspberry Pi Pico
Raspberry Pi Pico C/C++ SDK
RP2040 Datasheet

関連情報

Raspberry Pi リモート開発環境構築
Raspberry Pi Pico 開発環境構築
TDK InvenSense ICM-20948 の使い方

前提条件

調査における前提条件を示します。

- 開発用ホスト マシンは、Windows10（64 ビット）がインストールされた PC（i7-3930K，32GB メモリ搭載の PC）を使用します。
- Raspberry Pi3B（Raspberry Pi OS 32BIT）を SPI マスタとして使用します。
- 1 台の Raspberry Pi Pico を SPI スレーブのターゲットとして使用し、もう一台を Probe 用として使用します。

- 開発環境は、図 1 の通りです。
  (Raspberry Pi 3B と Raspberry Pi Pico のリモート開発環境は構築済みとします。
  開発環境構築に関する資料はロボバイオ エクス サイトの一般ドキュメントのページにあります。)

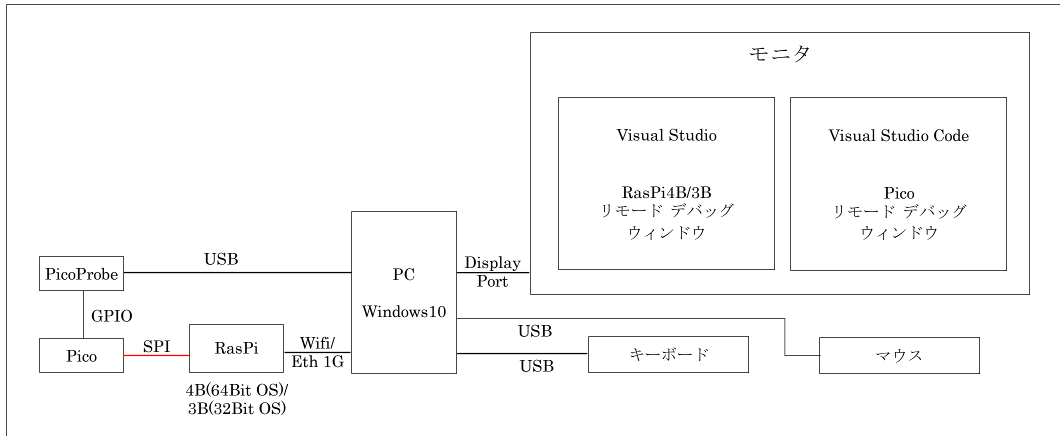

図 1 開発環境

- SPI 通信は 2Mbps の速度を想定し、約 4 ミリ秒周期で 64 バイト長のデータを全二重バースト転送するものとします。(結線は 20cm 程のジャンパー線を使用します。)
- 本書の記載内容は、C/C++言語での通信プログラム作成の経験がある方が対象です。

調査結果

〔SPI スレーブ デバイス開発用プロジェクトの作成〕

(以降、SPI スレーブ デバイス開発用プロジェクトを spi-slave と記します。)

1 Git Bash を起動して、Raspberry Pi Pico SDK のディレクトリがあるディレクトリ（SDK ディレクトリの親）に移動する。
2 次のコマンドでプロジェクト自動生成ツール（python プログラム）を取得する。
  git clone https://github.com/raspberrypi/pico-project-generator.git
  “pico-project-generator”ディレクトリが生成される。
3 Developer Command Prompt for VS 2019 を起動する。
4 プロジェクト自動生成ツールのディレクトリに移動する。
5 次のコマンドでプロジェクト自動生成ツールを起動する。
  pico_project.py --gui
6 自動生成ツールの GUI で次の設定を行ってから、「OK」ボタンを押下する。
  6.1 プロジェクト名(Project Name) ← “spi-slave”
  6.2 プロジェクトの場所(Location)
  6.3 ライブラリ オプション(Library Option)
  「SPI」にチェックを入れる。
  (その他の設定は必要に応じて設定します。)

＜設定例＞

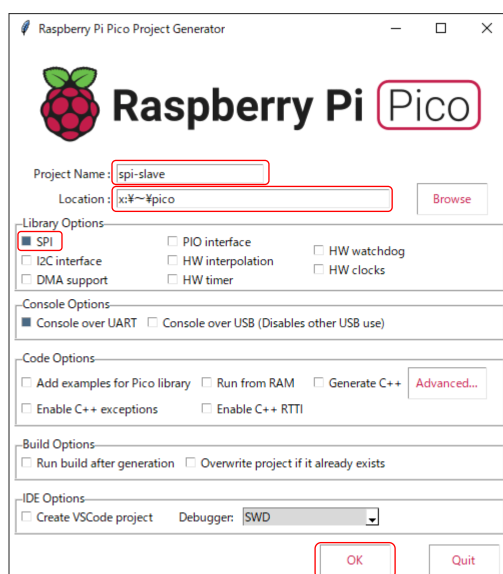

図 2 プロジェクト自動生成ツールの設定

プロジェクト ディレクトリが生成され、その中に雛形のコード ファイルと CMakeLists.txt などが生成される。

7 Raspberry Pi Pico SDK 導入時に、「PICO_SDK_PATH」を" ..¥..¥pico-sdk"に設定した場合は、フルパス指定に変更する。

7.1 Windows 10 の設定/システム/詳細情報/システム詳細設定/環境変数の設定で、「PICO_SDK_PATH」をフルパスに変更する。

7.2 Developer Command Prompt for VS 2019 から Visual Studio Code を起動して、Cmake: Configure Environment の「PICO_SDK_PATH」をフルパスに変更する。

7.3 既に作成済みのプロジェクトのディレクトリにある.vscode/launch.json の"svdFile"設定行を PICO_SDK_PATH フルパスに対応する。

8 既に作成済みのプロジェクトのディレクトリにある.vscode ディレクトリを spi-slave ディレクトリにコピーする。

9 spi-slave/.vscode/ディレクトリの launch.json の"executable"設定行のモジュール指定を変更する。
＜変更例＞ "executable": "${workspaceRoot}/build/spi-slave.elf",

10 Visual Studio Code で spi-slave ディレクトリを開いて、使用するコンパイラに"GCC for arm-none-eabi"を指定する。(自動的に入力フィールドが表示されます。)

11 雛形のコードをビルドする。

12 Raspberry Pi Pico を PC に接続して、openOCD を起動後、F5 でデバッグを開始できるか確認する。

これで、SPI スレーブ デバイス開発用プロジェクトの作成は完了です。

〔SPI マスタ(Pi3B)と SPI スレーブ(Pico)間の結線〕

表 1 と図 3 に、SPI マスタ(Pi3B)と SPI スレーブ(Pico)間の結線のピン対応と結線の例を示します。

表 1 SPI マスタ(Pi3B)と SPI スレーブ(Pico)間の結線

SPI マスタ(Pi3B)		結線	SPI スレーブ(Pico)	
	21	—	25	
	23	—	24	
	25	—	23	
	24	—	22	
	19	—	21	

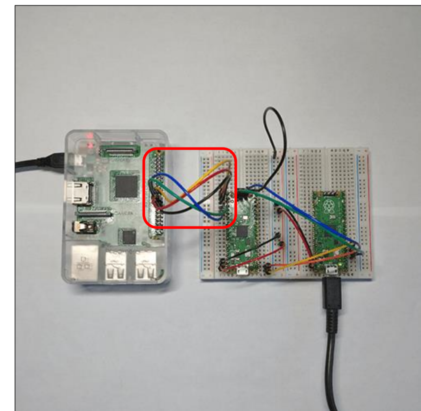

図 3 結線の例

〔SPI 構成処理の実装方法〕

RP2040 Datasheet の 4.4.3.2 章に、「SPI の構成は回路が無効になっている状態で行わなければならない」と記述があるので、Raspberry Pi Pico SDK の spi_init を元にスレーブ用の初期化関数を作成します。

〔スレーブのクロック レシオ〕

RP2040 Datasheet の 4.4.3.4 章に以下の記述があります。

> In slave mode, the same maximum SSPCLK frequency of 133 MHz can achieve a peak bit rate of 133 / 12 = ~11.083 Mbps. The SSPCPSR register can be programmed with a value of 12, and the SCR[7:0] field in the SSPCR0 register can be programmed with a value of 0.

Raspberry Pi Pico SDK のデフォルト動作周波数は 125MHz なので、上の文中の 133MHz を 125MHz に読み替えると、

> スレーブ モードにおいて、SSPCPSR レジスタが 12、SSPCR0 レジスタの SCR[7:0]フィールドが 0 で設定された場合、125MHz の SSPCLK で 125/12=~10.416Mbps を達成できる。

になると思います。

更にその下に以下の記載があります。

> $F_{SSPCLK}(\text{min}) \geq 12 \times F_{SSPCLKIN}(\text{max})$, for slave mode.
> $F_{SSPCLK}(\text{max}) \leq 254 \times 256 \times F_{SSPCLKIN}(\text{min})$, for slave mode.

スレーブ モードの場合、「SPI 信号線の SCK の周波数は、SPI 制御回路のクロック周波数の $\frac{1}{12}$ 以下、且つ $\frac{1}{254 \times 256}$ 以上の範囲」と言うことだと思います。(SSPCPSR=12、SPCR0:SCR＋1=1 の時、最高周波数となり、SSPCPSR=254、SPCR0:SCR＋1=256 の時、最低周波数となる。)

上記の認識より、クロック レシオ設定は、

SSPCPSR レジスタ ＝12

SSPCR0 レジスタの SCR[7:0]フィールド ＝0

とします。

〔SPI 構成処理の順序〕

RP2040 Datasheet の 4.4.3 章にある記載通り、以下の順序とします。

- SPI 内部回路リセット
- Motorola SPI の選択（リセットで Motorola SPI が選択された状態になるので省略）
- スレーブ モードの選択
- クロック レシオの設定
- フレーム フォーマットの設定
  フレーム フォーマットの詳細は、インターネット上に分かり易い日本語の説明がありますので、そちらを参照して下さい。（"SPI 通信"で検索すると見つかります。）
- SPI 内部回路を有効化

〔SPI 構成処理の実装〕

```
void spi_slave_init(spi_inst_t *spi, uint data_bits, spi_cpol_t cpol, spi_cpha_t cpha, __unused spi_order_t order) {
    spi_reset(spi);
    spi_unreset(spi);

    spi_set_slave(spi, true);
    spi_get_hw(spi)->cpsr = 12;
    hw_write_masked(&spi_get_hw(spi)->cr0, 0, SPI_SSPCR0_SCR_BITS);

    spi_set_format(spi, data_bits, cpol, cpha, order);

    // Always enable DREQ signals -- harmless if DMA is not listening
    hw_set_bits(&spi_get_hw(spi)->dmacr, SPI_SSPDMACR_TXDMAE_BITS | SPI_SSPDMACR_RXDMAE_BITS);

    // Finally enable the SPI
    hw_set_bits(&spi_get_hw(spi)->cr1, SPI_SSPCR1_SSE_BITS);
}
```

緑の網掛け部以外は spi_slave_init 関数と同じ処理です。

〔SPI 全二重通信処理の実装方法〕

〔実現が困難なプロトコル〕

マスタから指定されたアドレスを先頭とするブロックの値を返すような単一トランザクションを spi_read_blocking と spi_write_blocking 関数を組み合わせて実現した場合、マスタ側の受信データの 3 バイト目に先頭アドレスに対応する値が格納されます。（先頭アドレスを Rx FIFO からリードしてから Tx FIFO に送信データをライトするまでの時間が長くなると、更に後ろになります。）2 バイト目に格納されるようにするには、信号レベルの 1 バイト目の受信が完了する前に、SPI 制御回路が 2 バイト目の送信データを Tx FIFO からリードできるようにしておく必要があると思われます。

〔全二重通信の実装〕

SPI スレーブ モードの構成後に、spi_write_read_blocking 又は spi_write16_read16_blocking 関数を呼び出すことで全二重のバースト転送を実現できます。極めてシンプルですが、受信オーバー ランや送信アンダー ランが起きないようにプロトコルを制定しなければなりません。又、受信データの信頼性は SPI 通信を使用する側が確認する必要があります。

約 4 ミリ秒間隔で 64 バイト長のデータを全二重転送するプログラムの例を以下に示します。

SPI マスタ(Raspberry Pi 3B)側のソース コード

```
int main()
{
    printf("SPI マスタ － スレーブ通信テスト開始¥n");

    rbxSys::SPI spi(0, 0);
    Bool sts = spi.SetupSPI(rbxSys::SPI::CLK_MODE1, 8, false, 2 * 1000 * 1000);
    if (sts == true)
    {
        printf("SPI マスタ初期化完了¥n");
        constexpr Int32 transferLen = 64;
        UInt8 wBuff[transferLen];
        UInt8 rBuff[transferLen];

        Int32 errors = 0;
        for (Int32 i = 0; i < 10000; ++i)
        {
            for (Int32 j = 0; j < transferLen; ++j)
            {
                wBuff[j] = UInt8(10 + j);
            }
            memset(rBuff, 0, transferLen);

            spi.WriteReadBlocking(wBuff, rBuff, sizeof(rBuff));
            for (Int32 j = 0; j < transferLen; ++j)
            {
                if (rBuff[j] != UInt8(100 + j))
                {
                    ++errors;
                    break;
                }
            }

            usleep(3400);
        }
        printf("テスト結果 Errors = %d¥n", errors);
    }

    printf("SPI マスタ － スレーブ通信テスト終了¥n");
    return 0;
}
```

Raspberry Pi 3/4 で SPI マスタとして通信する方法に関しての情報は、TDK InvenSense ICM-20948 の使い方の 5 章 準備の「SPI 通信の準備」の 6～7 項にあります。

SPI スレーブ(Raspberry Pi Pico)側のソース コード

```
int main()
{
    stdio_init_all();

    gpio_set_function(PIN_MISO, GPIO_FUNC_SPI);
    gpio_set_function(PIN_CS,   GPIO_FUNC_SPI);
    gpio_set_function(PIN_SCK,  GPIO_FUNC_SPI);
    gpio_set_function(PIN_MOSI, GPIO_FUNC_SPI);

    puts("SPI Slave Test Start.");

    spi_slave_init(SPI_PORT, 8, SPI_CPOL_0, SPI_CPHA_1, SPI_MSB_FIRST);

    const int tranferLen = 64;
    uint8_t wBuff[64];
    uint8_t rBuff[64];
    memset(&wBuff[0], 0, sizeof(wBuff));
    memset(&rBuff[0], 0, sizeof(rBuff));

    int errors = 0;
    for (int i = 0; i < 10000; ++i) {
        for (int j = 0; j < tranferLen; ++j) {
            wBuff[j] = (uint8_t) (100 + j);
        }
        memset(&rBuff[0], 0, tranferLen);

        spi_write_read_blocking(SPI_PORT, &wBuff[0], &rBuff[0], tranferLen);

        for (int j = 0; j < tranferLen; ++j) {
            if (rBuff[j] != (uint8_t) (10 + j)) {
                ++errors;
                break;
            }
        }
    }

    printf("SPI Slave Test End. (Erros = %d)", errors);
    return 0;
}
```

上記のプログラムを数十回実行した限りでは、受信データのエラーは発生しませんでした。
通信の品質評価は、DMA 版を下記の環境で動作させて実施する予定です。

- 通信周波数：4MHz
- ポート 0：スレーブ／ポート 1：マスタ
- マスタ配下のスレーブ数：2

所感

Raspberry Pi Pico を SPI スレーブ デバイスとして動作させる方法についての日本語の情報がインターネット上で見つからなかった為、手探り作業になりましたが、比較的スムーズに調査を進めることができました。Raspberry Pi の公式資料と Pico SDK＋ツールが整備されていることがありがたかったです。（情報源が英語なので、解釈に誤りや不十分な点があるかも知れません。リアルノイド プロジェクトの進行過程で、情報の誤りや不足が発覚した場合は、本書を改定します。）