Raspberry Pi Pico SPI スレーブ プログラミング(2)（公開資料） (Raspberry Pi Pico SPI Slave Programming (2))			作成 2021/12/11 ニューラルソフト有限会社 市来 博記
初版	－	2021/12/11	
B	PDF にメタ データを追加した。	2021/12/25	

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 DMA 転送を利用した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。(Raspberry Pi Pico SPI スレーブ プログラミング(1)の続きになるものです。)

調査内容

調査項目を示します。

1　SPI スレーブ構成における DMA の利用方法

情報源

Raspberry Pi Pico C/C++ SDK
RP2040 Datasheet
Raspberry Pi Pico SDK とサンプル コード(spi/spi_dma)

関連情報

Raspberry Pi Pico SPI スレーブ プログラミング(1)

前提条件

調査における前提条件を示します。

- 開発環境は Raspberry Pi Pico SPI スレーブ プログラミング(1)に記載した環境と同じとします。
- Raspberry Pi Pico SPI スレーブ プログラミング(1)に記載した DMA を使用しない SPI 通信プログラムは、作成済みとします。
- SPI 通信は 4Mbps 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

〔DMA 版 SPI スレーブ デバイス開発用プロジェクト(spi-slave2)の作成〕

1　Developer Command Prompt for VS 2019 を起動する。

2　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。

3　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

4　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。

4.1　プロジェクト名(Project Name) ← “spi-slave2”

4.2　プロジェクトの場所(Location)

4.3　ライブラリ オプション(Library Option)

「SPI」と「DMA support」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＞

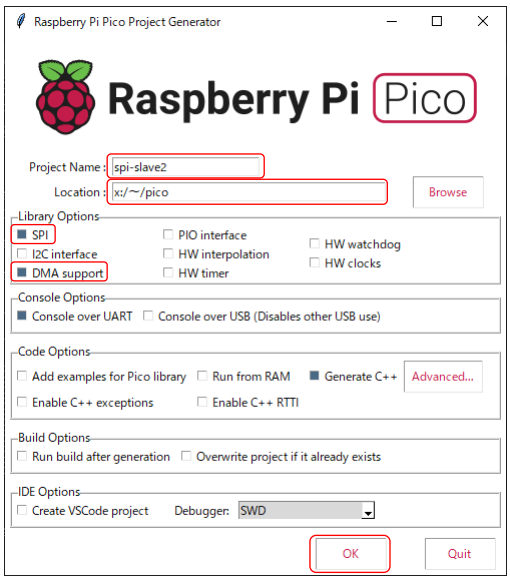

図 1 プロジェクト自動生成ツールの設定

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。

5　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-slave2 ディレクトリにコピーする。

6　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-slave2.elf",

7　Visual Studio Code で spi-slave2 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。）

8　雛形のコードをビルドする。

9　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、DMA 版 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 スレーブ構成における DMA の利用方法〕

DMA を利用した SPI 通信は、メモリから Tx FIFO、Rx FIFO からメモリに MPU で転送していた処理を DMA で行うようにするだけなので、非常に簡単に実現できます。一般的に DMA（Direct Memory Access）は、メモリから固定アドレスのアウトプット、固定アドレスのインプットからメモリ、メモリからメモリの転送が可能になっています。(RP2040 もそうなっています。) SPI の Tx FIFO へのライトは、メモリから固定アドレスのアウトプットへの転送機能を使用し、SPI の Rx FIFO からのリードは、固定アドレスのインプットからメモリへの転送機能を使用することになります。

DMA を利用した SPI 通信処理の概要は以下のようになります。

1　未使用 DMA チャンネルの確保
2　DMA チャンネル構成データの生成
3　メモリ上の送信データを確定
4　DMA チャンネルを構成
5　DMA 転送開始
6　DMA 転送完了待ち
　並行実行する処理がある場合：
　　並行実行する処理を実行した後、受信側 DMA 転送終了をポーリングで待つ。
　並行実行する処理がない場合：
　　スリープ状態に移行(省電力化)して受信側 DMA 転送完了の割り込みを待つ。
　どちらの場合でも、同期オブジェクトで待機するようにできます。
7　受信データに応じた処理
8　3 に戻る

〔送信と受信で使用する DMA チャンネルの確保〕

RP2040 は DMA のチャンネルを 12 持っています。その内の未使用の 2 チャンネルを SPI スレーブ モードの送受信用に確保します。

```
// 未使用の DMA チャネルを取得
const uint dmaTx = dma_claim_unused_channel(false);
const uint dmaRx = dma_claim_unused_channel(false);
if ((dmaTx < 0) || (dmaRx < 0))
{
    // DMA チャネル確保エラー
}
```

int dma_claim_unused_channel(bool required) が未使用の DMA チャンネルを確保する SDK 関数です。戻り値は確保した DMA のチャンネル番号です。required に true を指定すると、チャンネル確保ができない場合、「panic」として扱われるようです。

〔DMA チャンネル構成データの生成〕

DMA の構成要素を以下に示します。

- 転送中に転送元のアドレスをインクリメントするかどうかの選択
- 転送中に転送先のアドレスをインクリメントするかどうかの選択
- 転送要求信号の指定
  SPI 送信用の場合、Tx FIFO がフルでない時、転送要求がアクティブになる信号
  SPI 受信用の場合、Rx FIFO がエンプティでない時、転送要求がアクティブになる信号
- 転送完了時に次の転送を開始する DMA チャンネル番号の指定
- 転送ワードのビットサイズの指定
- 転送範囲を環状にするかどうかの選択
- バイト スワップを行うかどうかの選択
- 割り込み要求 QUIET モードの選択（QUIET モード : 転送完了時ではなく、NULL トリガ時に割り込み要求を発行するモード）
- 転送シーケンスの有効/無効の選択
- 転送データの監視をするかどうかの選択（FIFO を通過するデータから CRC などのチェック サムを生成できるようです。）

SPI 送信用と受信用の DMA 構成データの生成コードは以下のようになります。

```
    //  送信 DMA 構成データ設定
    dma_channel_config dmaTxConfig = dma_channel_get_default_config(dmaTx);
#if WD_BYTES == 1
    channel_config_set_transfer_data_size(&dmaTxConfig, DMA_SIZE_8);
#else
    channel_config_set_transfer_data_size(&dmaTxConfig, DMA_SIZE_16);
    channel_config_set_bswap(&dmaTxConfig, true);
#endif
    channel_config_set_dreq(&dmaTxConfig, spi_get_dreq(SPI_PORT, true));

    //  受信 DMA 構成データ設定
    dma_channel_config dmaRxConfig = dma_channel_get_default_config(dmaRx);
#if WD_BYTES == 1
    channel_config_set_transfer_data_size(&dmaRxConfig, DMA_SIZE_8);
#else
    channel_config_set_transfer_data_size(&dmaRxConfig, DMA_SIZE_16);
    channel_config_set_bswap(&dmaRxConfig, true);
#endif
    channel_config_set_dreq(&dmaRxConfig, spi_get_dreq(SPI_PORT, false));
    channel_config_set_read_increment(&dmaRxConfig, false);
    channel_config_set_write_increment(&dmaRxConfig, true);
```

見にくいコードになっていますが、dma_channel_config dma_channel_get_default_config 関数でデフォルトの構成データを取得した後、デフォルト設定と異なる項目を channel_config_set_～関数で上書きすると言う流れです。（SPI のワードのビットサイズを 16 にした場合は、上位バイトと下位バイトが反転するのでバイト スワップに「有効」を設定しています。）
デフォルトの構成データの内容は以下の通りです。

- 転送中に転送元のアドレスをインクリメントする
- 転送中に転送先のアドレスをインクリメントしない
- 転送要求信号は使用しない（強制転送）
- 転送完了時に次の転送を開始する DMA チャンネル＝自チャンネルの番号
- 転送ワードのビットサイズ＝32
- 転送範囲は非環状

- バイト スワップは行わない
- 転送完了時に割り込み要求を発行する
- 転送シーケンスは有効
- 転送データの監視はしない

〔DMA チャンネル構成〕

DMA チャンネルの構成は、チャンネル毎に SDK の 1 関数を呼び出すだけです。

```
    //  送信 DMA 構成
    dma_channel_configure(
        dmaTx, &dmaTxConfig,
        &spi_get_hw(SPI_PORT)->dr,  //  転送先アドレス
        txBuff,                     //  転送元アドレス
        sizeof(txBuff) / WD_BYTES,  //  転送エレメント数
        false);                     //  直後に転送を行わない

    //  受信 DMA 構成
    dma_channel_configure(
        dmaRx, &dmaRxConfig,
        rxBuff,                     //  転送先アドレス
        &spi_get_hw(SPI_PORT)->dr,  //  転送元アドレス
        sizeof(rxBuff) / WD_BYTES,  //  転送エレメント数
        false);                     //  直後に転送を行わない
```

〔受信完了割り込み要求の設定（SPI 転送完了を受信完了割り込みで検知する場合)〕

以下の設定を実施します。

- DMA コントローラに対して受信チャンネルの IRQ0 を有効化する。
- 割り込みコントローラに対して IRQ0 の割り込みハンドラを登録する。
- 割り込みコントローラに対して IRQ0 の割り込みを有効化する。

```
    //  受信完了割り込み設定
    dma_channel_set_irq0_enabled(dmaRx, true);
    //  DMA_IRQ_0 有効化
    irq_set_exclusive_handler(DMA_IRQ_0, dma_handler);
    irq_set_enabled(DMA_IRQ_0, true);
```

irq_set_～関数を使用するにはソース ファイルへのヘッダ ファイルのインクルードと、CMakeLists.txt ファイルへのライブラリ指定の追加が必要です。

ヘッダ ファイルのインクルード

```
#include "hardware/irq.h"
```

CMakeLists.txt ファイルへのライブラリ指定

```
target_link_libraries(spi-slave2
        hardware_spi
        hardware_dma
        hardware_irq  ← これを追加します。
        )
```

〔DMA 転送の開始〕

DMA 転送を開始することで SPI 全二重バースト転送の準備が完了となり、マスタから選択させたタイミングで送受信が開始されます。

```
dma_start_channel_mask((1u << dmaTx) | (1u << dmaRx));
```

〔DMA 転送完了待ち〕

DMA 転送完了をポーリングで検知する場合：

dma_channel_wait_for_finish_blocking 関数を呼び出して受信側 DMA 転送完了を待ちます。

```
dma_channel_wait_for_finish_blocking(dmaRx);
```

DMA 転送完了を割り込みで検知する場合：

WFI 命令を実行してスリープ状態に移行します。

```
// スリープ状態に移行
__wfi();
```

割り込み発生時、ハンドラ内で割り込みをクリアする必要があります。

```
dma_hw->ints0 = 1u << 受信側 DMA チャンネル番号;
```

〔DMA の利用した SPI スレーブ プログラムの全体像〕

乱暴な部分もあり、見にくいコードですが、次の頁に DMA 版 SPI スレーブ プログラムの例を示します。（マスタ側のプログラムは Raspberry Pi Pico SPI スレーブ プログラミング(1) のプログラムと同じです。）

```
#include <stdio.h>
#include <string.h>
#include "pico/stdlib.h"
//#include "pico/binary_info.h"
#include "hardware/spi.h"
#include "hardware/dma.h"
#include "hardware/irq.h"
#include "hardware/sync.h"

// SPI Defines
// We are going to use SPI 0, and allocate it to the following GPIO pins
// Pins can be changed, see the GPIO function select table in the datasheet for information on GPIO assignments
#define SPI_PORT spi0
#define PIN_MISO 16
#define PIN_CS   17
#define PIN_SCK  18
#define PIN_MOSI 19

#define WD_BYTES 2
#define INTERRUPT

#if defined(INTERRUPT)
static uint s_dmaRxCh = -1;
void __not_in_flash_func(dma_handler)()
{
    if (s_dmaRxCh > 0)
    {
        dma_hw->ints0 = 1u << s_dmaRxCh;
        s_dmaRxCh = -1;
    }
}
#endif

int main()
{
    stdio_init_all();

    puts("SPI Slave Test Start.");

    //  未使用の DMA チャネルを取得
    const uint dmaTx = dma_claim_unused_channel(false);
    const uint dmaRx = dma_claim_unused_channel(false);
    if ((dmaTx < 0) || (dmaRx < 0))
    {
        puts("No Free DMA Chanel.");
        return -1;
    }

    //  SPI スレーブ構成
    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);

#if WD_BYTES == 1
    spi_slave_init(SPI_PORT, 8, SPI_CPOL_0, SPI_CPHA_1, SPI_MSB_FIRST);
#else
    spi_slave_init(SPI_PORT, 16, SPI_CPOL_0, SPI_CPHA_1, SPI_MSB_FIRST);
#endif

    constexpr int tranferSize = 64;
    uint8_t txBuff[tranferSize];
    uint8_t rxBuff[tranferSize];
    memset(&txBuff[0], 0, sizeof(txBuff));
    memset(&rxBuff[0], 0, sizeof(rxBuff));

    //  送信 DMA 構成データ設定
    dma_channel_config dmaTxConfig = dma_channel_get_default_config(dmaTx);
#if WD_BYTES == 1
    channel_config_set_transfer_data_size(&dmaTxConfig, DMA_SIZE_8);
#else
    channel_config_set_transfer_data_size(&dmaTxConfig, DMA_SIZE_16);
    channel_config_set_bswap(&dmaTxConfig, true);
#endif
    channel_config_set_dreq(&dmaTxConfig, spi_get_dreq(SPI_PORT, true));
    //  受信 DMA 構成データ設定
    dma_channel_config dmaRxConfig = dma_channel_get_default_config(dmaRx);
#if WD_BYTES == 1
    channel_config_set_transfer_data_size(&dmaRxConfig, DMA_SIZE_8);
#else
    channel_config_set_transfer_data_size(&dmaRxConfig, DMA_SIZE_16);
    channel_config_set_bswap(&dmaRxConfig, true);
#endif
    channel_config_set_dreq(&dmaRxConfig, spi_get_dreq(SPI_PORT, false));
    channel_config_set_read_increment(&dmaRxConfig, false);
    channel_config_set_write_increment(&dmaRxConfig, true);

    int errors = 0;
    for (int i = 0; i < 10000; ++i)
    {
        for (int j = 0; j < tranferSize; ++j)
        {
          txBuff[j] = (uint8_t) (100 + j);
        }
        memset(&rxBuff[0], 0, tranferSize);

        //  送信 DMA 構成
        dma_channel_configure(
            dmaTx, &dmaTxConfig,
            &spi_get_hw(SPI_PORT)->dr,  //  転送先アドレス
            txBuff,                     //  転送元アドレス
            sizeof(txBuff) / WD_BYTES,  //  転送エレメント数
            false);                     //  直後に転送を行わない

        //  受信 DMA 構成
        dma_channel_configure(
            dmaRx, &dmaRxConfig,
            rxBuff,                     //  転送先アドレス
            &spi_get_hw(SPI_PORT)->dr,  //  転送元アドレス
            sizeof(rxBuff) / WD_BYTES,  //  転送エレメント数
            false);                     //  直後に転送を行わない

#if defined(INTERRUPT)
        //  受信完了割り込み設定
        dma_channel_set_irq0_enabled(dmaRx, true);
        //  DMA_IRQ_0 有効化
        irq_set_exclusive_handler(DMA_IRQ_0, dma_handler);
        irq_set_enabled(DMA_IRQ_0, true);
        s_dmaRxCh = dmaRx;
#endif

        //  送受信データ転送開始
        dma_start_channel_mask((1u << dmaTx) | (1u << dmaRx));

#if !defined(INTERRUPT)
        dma_channel_wait_for_finish_blocking(dmaRx);
#else
        __wfi();
#endif

        if (!dma_channel_is_busy(dmaTx))
        {
            for (int j = 0; j < tranferSize; ++j)
            {
                if (rxBuff[j] != (uint8_t) (10 + j))
                {   //  受信データ異常
                    ++errors;
                    break;
                }
            }
        }
        else
        {   //  プロトコル異常
            ++errors;
        }
    }

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

緑の網掛部が spi_write_read_blocking/spi_write16_read16_blocking 関数に相当する部分です。

上記のプログラムを下記の条件で数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。

- WD_BYTE=1 ＆ ポーリングによる転送完了待ち
- WD_BYTE=1 ＆ 受信完了割り込みによる転送完了待ち
- WD_BYTE=2 ＆ ポーリングによる転送完了待ち
- WD_BYTE=2 ＆ 受信完了割り込みによる転送完了待ち

通信の品質評価は、下記の環境で実施する予定です。

- 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

DMA 関連のサンプル プログラムが一通り用意されている為､容易に SPI スレーブ通信プログラムに DMA 関連の処理を組み込むことができました。pico-examples 以外に、pico-extras と pico-playground のサンプル集があり、他の分野の調査でも大きな助けになると思われます。
Raspberry Pi 3/4 と Raspberry Pi Pico 間の SPI 通信は、単体であれば、4MHz のクロックで問題なくコミュニケーションできると思われます。RealNoids プロジェクトで使用する場合、配線長とモータのノイズがどれくらい影響するかが問題になります。（消費電力も心配です。）