Nios® II HAL API を使用したソフトウェア・サンプル集

「Modular Scatter-Gather DMA Core」

Ver.17.1

2018 年 8 月 Rev.1

Nios® II HAL API を使用したソフトウェア・サンプル集
「Modular Scatter-Gather DMA Core」

目次

1. はじめに

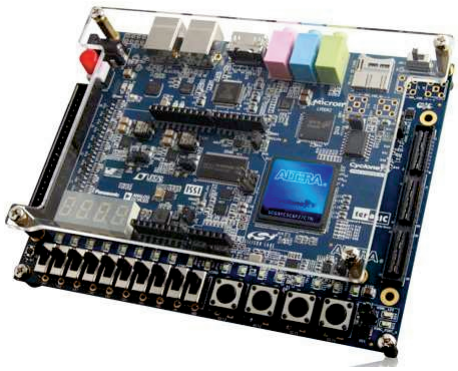
この資料は、Nios® II Hardware Abstraction Layer Application Program Interface（以降、HAL API ）を使用したソフトウェア・サンプル集シリーズの中から Modular Scatter-Gather DMA Core（以降、mSGDMA ）サンプルについて以下の内容を説明しています。

- 対応バージョン
- サンプルの機能
- サンプルのハードウェア・デザイン構成
- サンプルの使用方法
- サンプル・ソースコードの説明
- サンプル内で使用している HAL API の説明
  - インクルードするヘッダーファイル
  - 引数
  - 戻り値
  - 機能
  - 参考ドキュメントページ

2. 適用要件

このサンプルでは、下記のボード、ツールを使用しています。

- Cyclone® V GX スタータ開発キット
  https://www.intel.co.jp/content/www/jp/ja/programmable/products/boards_and_kits/dev-kits/altera/kit-terasic-cyclone-v-gx-starter.html



【図 2-1】 Cyclone V GX スタータ開発キット

- Intel® Quartus® Prime Version 17.1.0
  https://www.intel.co.jp/content/www/jp/ja/software/programmable/quartus-prime/overview.html

※ 17.1 以前のバージョンでも同様の方法で実装することは可能ですが、一部の機能や操作方法が異なる場合がありますのでご注意ください。

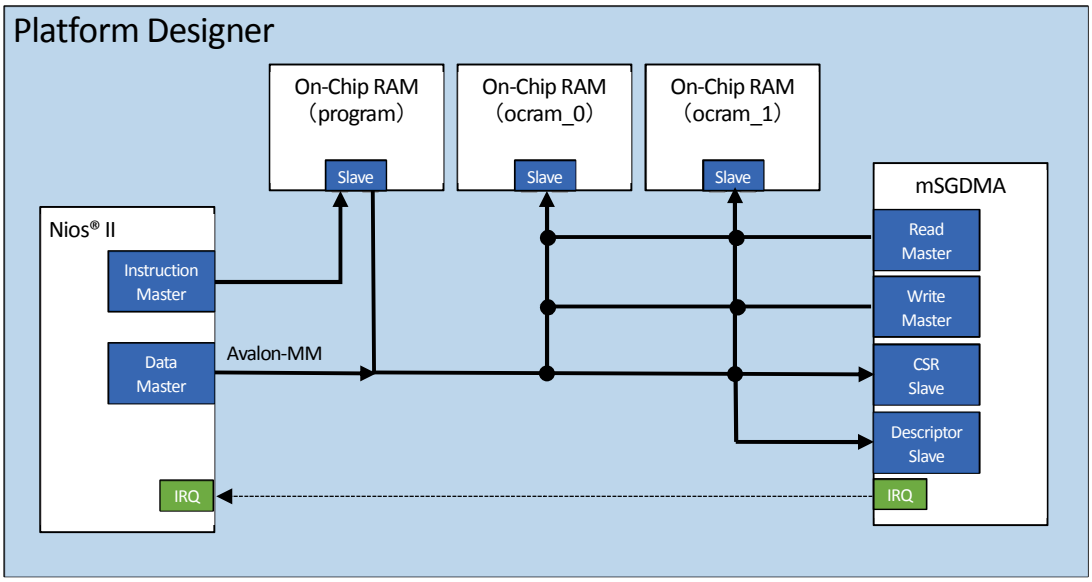
3. 仕様

3-1. 機能

Nios II を使用して mSGDMA を制御するためのサンプルデザインです。

このサンプルでは以下の内容を実行しています。

- On-Chip Memory (RAM and ROM) Core（以降、On-Chip RAM）のデータの初期化として、転送元 On-Chip RAM（ocram_0）に DMA 転送サイズ分のランダムデータを準備し、転送先 On-Chip RAM（ocram_1）を DMA 転送サイズ分ゼロクリアします。
- mSGDMA を使用して、Avalon Memory Mapped（以降、Avalon-MM ）インターフェースで接続されている 転送元 On-Chip RAM から 転送先 On-Chip RAM への DMA 転送（Avalon-MM to Avalon-MM の転送）を行います。
- alt_msgdma_standard_descriptor_async_transfer() HAL API 関数を使用して DMA 転送を行った場合は、mSGDMA 転送コールバック関数が呼ばれます。
- DMA 転送後の転送元 On-Chip RAM と 転送先 On-Chip RAM をデータベリファイします。
- 初期状態では、転送サイズは 1K バイトとしています。

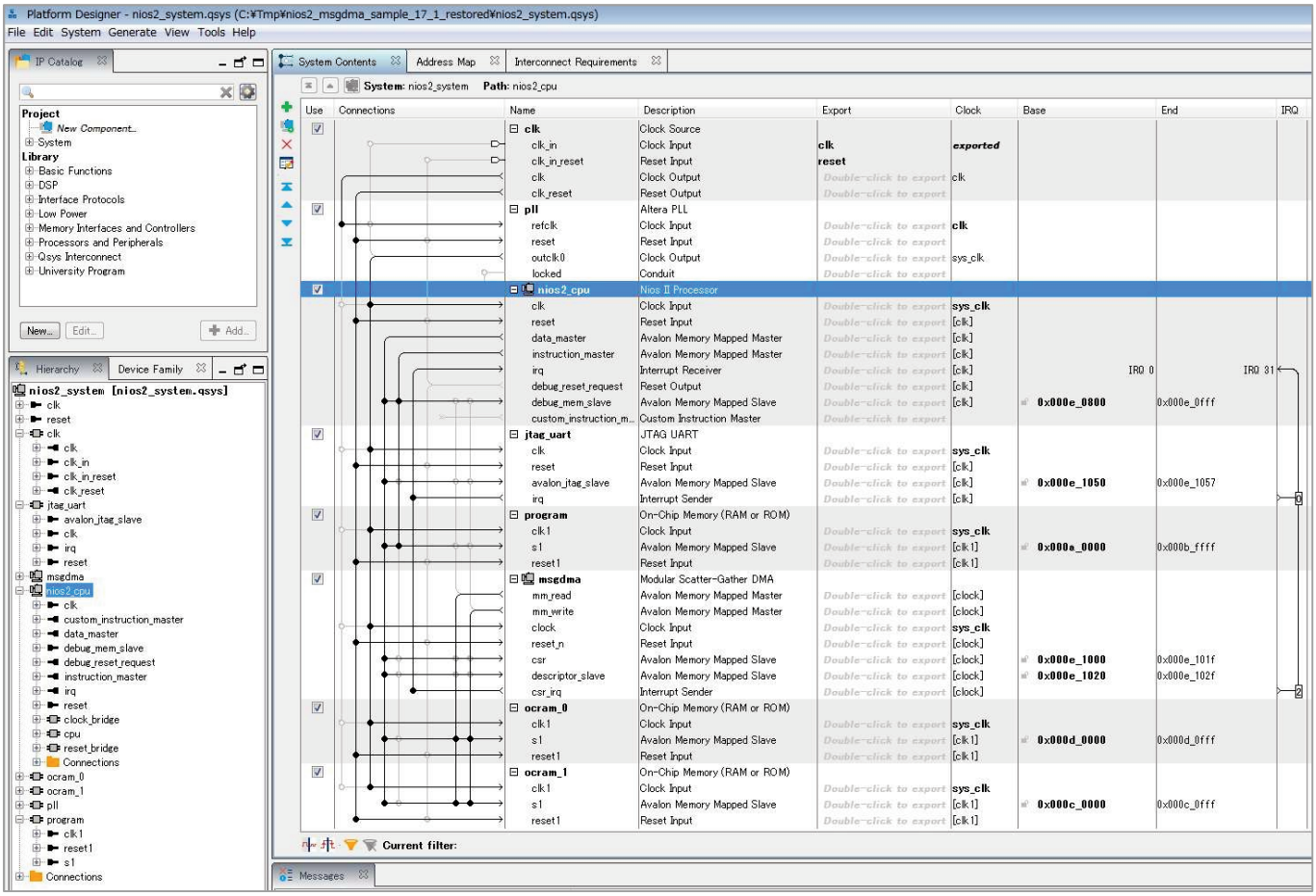


【図 3-1】 サンプルデザイン

3-2. サンプルのハードウェア・デザイン構成

このサンプルのハードウェア・デザイン nios2_msgdma_sample_17_1_restored/nios2_system.qsys には以下の IP コアが含まれており、Platform Designer（旧：Qsys システム統合ツール）上で Avalon-MM インターフェースで接続されています。

- Nios II CPU
- mSGDMA
- On-Chip RAM（program、 ocram_0、 ocram_1）



【図 3-2】 このサンプルの Platform Designer による接続

3-2-1. mSGDMA の設定

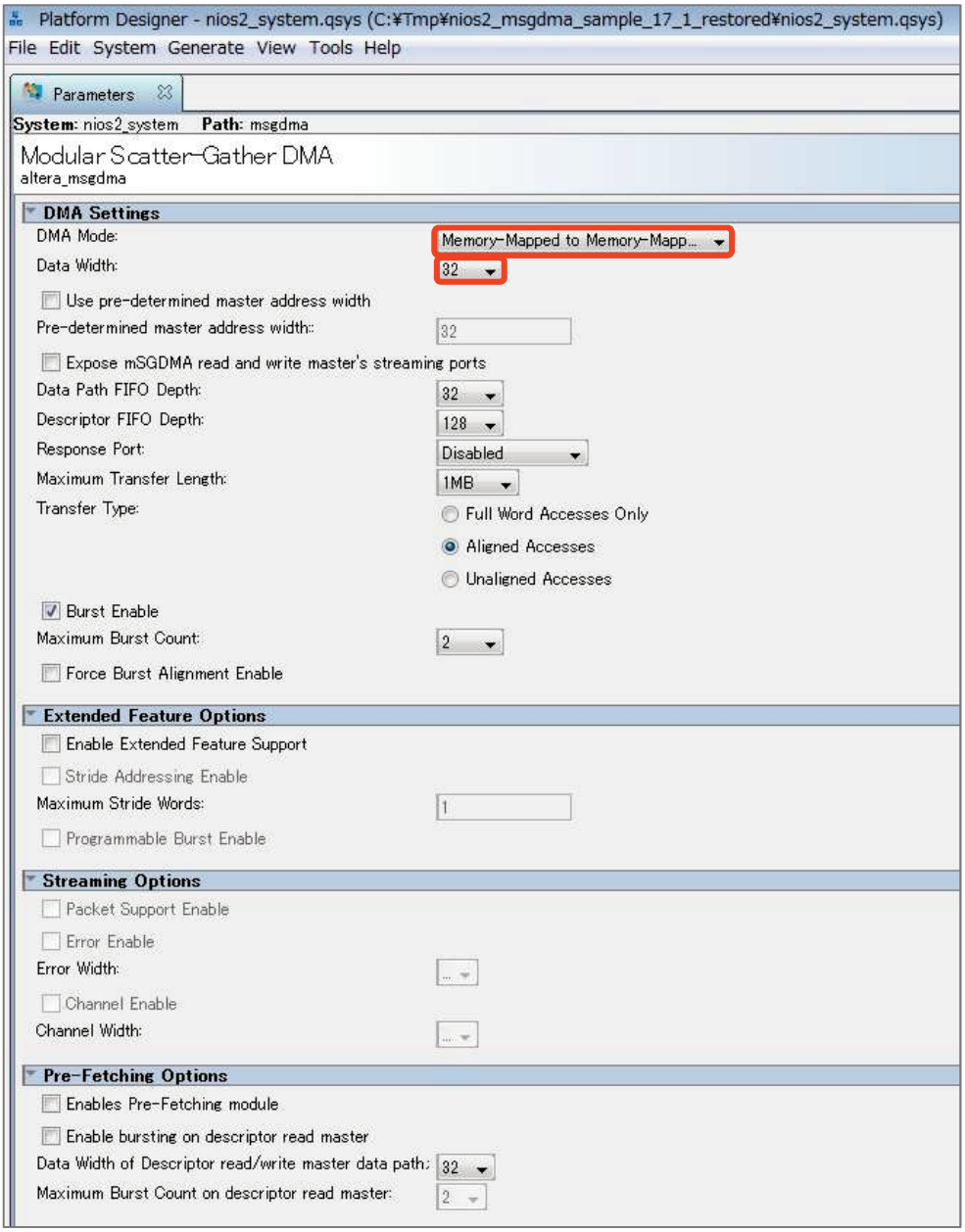
mSGDMA は下図のように設定しています。

【 mSGDMA 】

- DMA Mode ： Memory-Mapped to Memory-Mapped
- Data Width ： 32
- 上記以外は、接続するメモリの種類、送信データの種類や長さなどを考慮して設定してください。

DMA Mode は、転送元 On-Chip RAM から 転送先 On-Chip RAM への「Avalon-MM to Avalon-MM 転送」を行うので、“Memory-Mapped to Memory-Mapped” を設定しています。

Data Width は、On-Chip RAM の設定と合わせて “32” を設定しています。



【図 3-3】 mSGDMA の設定

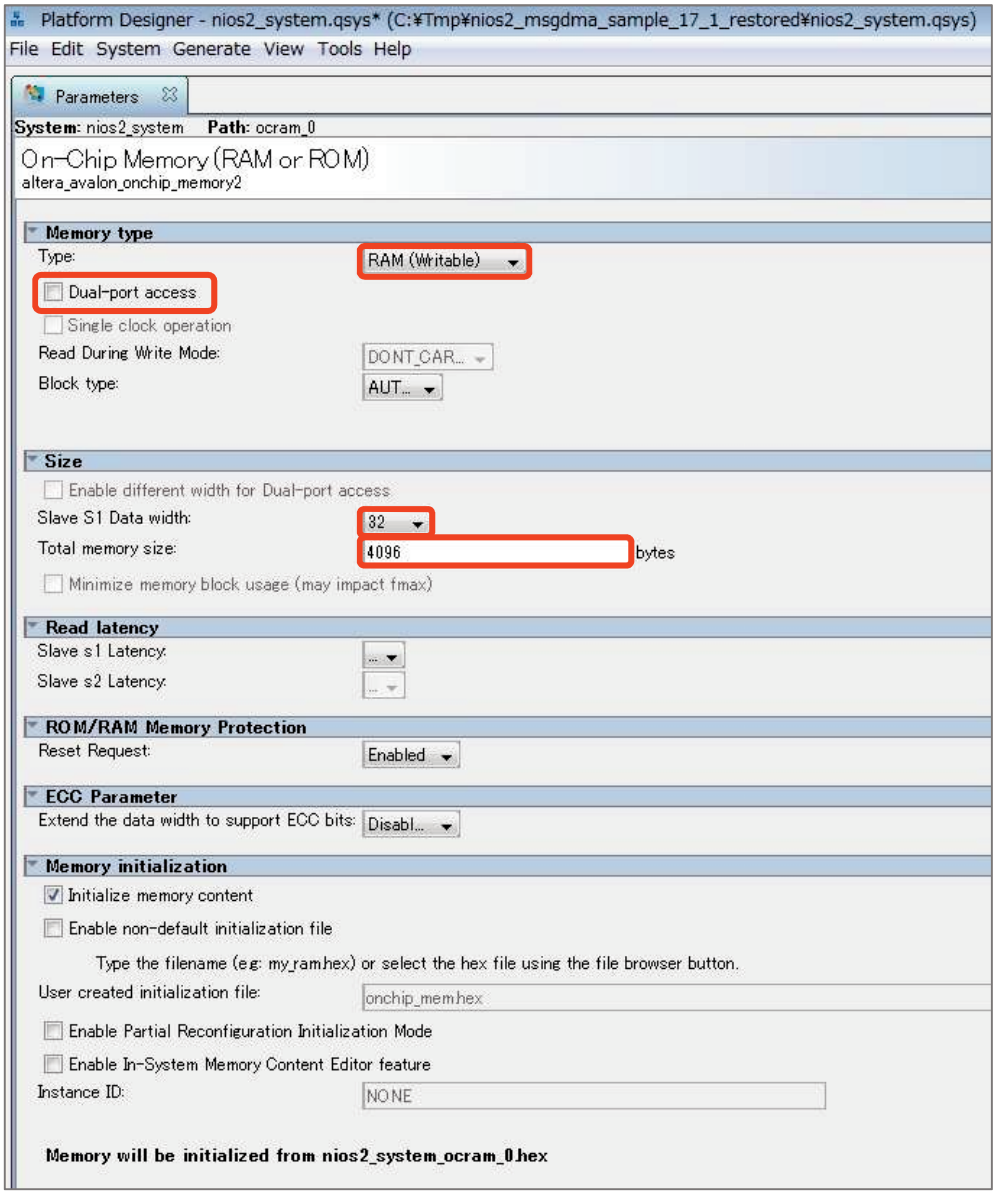
3-2-2. On-Chip RAM の設定

On-Chip RAM（ ocram_0、 ocram_1 ）は下図のように設定しています（下図は ocram_0 ）。

【 On-Chip RAM 】

- Type ： RAM(Writable)
- Dual-port access ： Off
- Slave S1 Data width ： 32
- Total memory size ： 4096

このサンプルでは、初期状態で DMA 転送データサイズを 1K バイトとしていますので、Total memory size に余裕を持って “4096”（4K バイト）を設定しています。



【図 3-4】 On-Chip RAM の設定

4. サンプルの使用方法

4-1. 使用手順

このサンプルには、使用手順を簡略化するためのスクリプトファイルが用意されています。

以下の手順によりサンプルを実行します。

① nios2_msgdma_sample_v17.1_r1.qar ファイルを任意のフォルダ（この例では C:¥Work）にコピーして Quartus Prime 開発ソフトウェアでオープンします。

② Cyclone V GX スタータ開発キットを用意して、必要な設定・接続を行います。

③ Nios II Command Shell を起動します。

④ Software フォルダに移動します。

```
$ cd "C:¥Work¥nios2_msgdma_sample_v17.1_r1_restored¥software"
```

⑤ 以下のスクリプトを実行し SOF ファイルの書き込みを行います。

```
$ ./write_sof.sh
```

⑥ ソフトウェアのビルドを実行します。

- BSP も含むフルビルドは以下のスクリプトを実行します。

```
$ ./build_all_sw.sh
```

- BSP は含まないアプリケーションのみのビルドは以下のスクリプトを実行します。

```
$ ./build_sw.sh
```

⑦ 以下のスクリプトを実行し Nios II ソフトウェアをダウンロードし、プログラムを実行します。

```
$ ./dl_nios.sh
```

4-2. プログラム実行例

このサンプルプログラムを実行すると、 Nios II Command Shell に以下のように表示されます。

```
$ ./dl_nios.sh
Using cable "USB-Blaster [USB-0]", device 1, instance 0x00
Pausing target processor: OK
Initializing CPU cache (if present)
OK
Downloaded 13KB in 0.2s (65.0KB/s)
Verified OK
Starting processor at address 0x000A018C
nios2-terminal: connected to hardware target using JTAG UART on cable
nios2-terminal: "USB-Blaster [USB-0]", device 1, instance 0
nios2-terminal: (Use the IDE stop button or Ctrl-C to terminate)

<< Start Program!! >>

<Debug> On-Chip RAM 0 Contents Before DMA!!

0000: 00 2D CF 46 29 04 B4 78 D8 68 A7 FF 3F 2B F1 FC
0010: D9 7A 96 09 2C A5 57 74 64 C4 AF 15 28 A4 E9 57
0020: DB 5E 20 FB 38 A8 4E A6 14 93 25 56 24 44 DF 59
  .
  . (途中省略)
  .
03D0: A7 8B 4B 4D FF 97 CC 84 30 AB DC 9E 84 AF 3B 11
03E0: 61 C9 8A ED EF 93 A4 CC 1D 2A BD E3 0A 59 1A 7B
03F0: B7 82 C7 05 FB 1A 90 4F 82 82 CA 99 32 6D B4 69

<Debug> On-Chip RAM 1 Contents Before DMA!!

0000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  .
  . (途中省略)
  .
03D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
03E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
03F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

<Debug> DMA Transfer Start!!
<Debug> DMA Callback Function called!!

<Debug> On-Chip RAM 0 Contents After DMA!!

0000: 00 2D CF 46 29 04 B4 78 D8 68 A7 FF 3F 2B F1 FC
0010: D9 7A 96 09 2C A5 57 74 64 C4 AF 15 28 A4 E9 57
0020: DB 5E 20 FB 38 A8 4E A6 14 93 25 56 24 44 DF 59
  .
  . (途中省略)
  .
03D0: A7 8B 4B 4D FF 97 CC 84 30 AB DC 9E 84 AF 3B 11
03E0: 61 C9 8A ED EF 93 A4 CC 1D 2A BD E3 0A 59 1A 7B
03F0: B7 82 C7 05 FB 1A 90 4F 82 82 CA 99 32 6D B4 69

<Debug> On-Chip RAM 1 Contents After DMA!!

0000: 00 2D CF 46 29 04 B4 78 D8 68 A7 FF 3F 2B F1 FC
0010: D9 7A 96 09 2C A5 57 74 64 C4 AF 15 28 A4 E9 57
0020: DB 5E 20 FB 38 A8 4E A6 14 93 25 56 24 44 DF 59
  .
  . (途中省略)
  .
03D0: A7 8B 4B 4D FF 97 CC 84 30 AB DC 9E 84 AF 3B 11
03E0: 61 C9 8A ED EF 93 A4 CC 1D 2A BD E3 0A 59 1A 7B
03F0: B7 82 C7 05 FB 1A 90 4F 82 82 CA 99 32 6D B4 69

<Debug> On-Chip RAM 0 and 1 Contents Verify!!
<Debug> Data Verify OK!!
```

DMA 転送前の転送元 On-Chip RAM のデータを
ダンプ表示して確認します（ランダムデータをフィル）

DMA 転送前の転送先 On-Chip RAM のデータを
ダンプ表示して確認します（ゼロデータをフィル）

DMA 転送を開始

DMA 転送後の転送元 On-Chip RAM のデータを
ダンプ表示して確認します（ランダムデータ）

DMA 転送後の転送先 On-Chip RAM のデータを
ダンプ表示して確認します
（ランダムデータが転送されている）

DMA 転送後に
転送元 と 転送先の On-Chip RAM のデータを
ベリファイします

【図 4-1】 サンプルプログラムの実行ログ

5. サンプル・ソースコードの説明

mSGDMA サンプルのソースコードについて以下に説明します。

```
//****************************************************************************
// file  sample.c
//
// attention
// Copyright (C) 2018 MACNICA,Inc. All Rights Reserved.¥n
//   This software is licensed "AS IS".
//   Please perform use of this software by a user's own responsibility and expense.
//   It cannot guarantee in the maker side about the damage which occurred by the ab-
//   ility not to use or use this software, and all damage that occurred secondarily.
//****************************************************************************

/****************************************************************************
 *  モジュラ・スキャッタギャザ DMA (mSGDMA) によるメモリ間転送サンプル
 ****************************************************************************/
/*****************************
 *  インクルード・ファイル
 *****************************/
#include "system.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/alt_cache.h>
#include <sys/alt_irq.h>

// mSGDMA 転送関連
#include <altera_msgdma_descriptor_regs.h>
#include <altera_msgdma_csr_regs.h>
#include <altera_msgdma.h>

/****************************************************************************
 *  re-definitions
 *  ※ このセクションは "system.h" で定義された定数を、
 *     本ソースファイル内で使用するために再定義したものです。
 *     ご使用の際は右側の定数を "system.h" を参照の上、変更してお使いください。
 ****************************************************************************/
#define MSGDMA_NAME (MSGDMA_CSR_NAME)    // mSGDMA のレジスタ名
#define OCRAM0_BASE (OCRAM_0_BASE)       // オンチップ RAM 0 のベースアドレス
#define OCRAM1_BASE (OCRAM_1_BASE)       // オンチップ RAM 1 のベースアドレス

/*****************************
 *  マクロ定義
 *****************************/
#define TRUE                -1
#define FALSE               0

#define USE_ASYNC_XFER  TRUE    // DMA 転送 :  TRUE = alt_msgdma_standard_descriptor_async_transfer() を使用
                                //             FALSE = alt_msgdma_standard_descriptor_sync_transfer() を使用
#define XFER_BYTES      1024    // DMA 転送のサイズ（バイト単位）: 1KByte

/*****************************
 *  グローバル変数
 *****************************/
volatile int xfer_cmp = FALSE;  // 転送完了フラグ
volatile int xfer_err = FALSE;  // 転送エラーフラグ
```

このサンプルのシステム・ヘッダーファイル（ハードウェア情報）

mSGDMA による転送で使用するインクルード・ファイル

"system.h" ファイルで定義された定数を本ソースファイル内で使用するラベルに再定義しています
ご使用の際は右側の定数をご使用の "system.h" ファイルを参照の上、変更してください

DMA 転送を Async か Sync のどちらで行うかを指定します

DMA 転送のサイズ（バイト単位）をここで指定します

次のページに続く

```
/*****************************
 *  プロトタイプ宣言
 *****************************/
void msgdma_callback( void* context );                                          // mSGDMA 転送コールバック関数
void init_ram( unsigned char *p_rd_adrs, unsigned char *p_wr_adrs, int bytes );  // RAM データ初期化関数
void dump_ram( unsigned char *p_ram_adrs, int bytes );                          // RAM データ・ダンプ関数
int verify_ram( unsigned char *p_rd_adrs, unsigned char *p_wr_adrs, int bytes );  // RAM データ・ベリファイ関数

/*****************************
 *  関数
 *****************************/
/*******************************************
 *  main 関数
 */
int main()
{
    alt_msgdma_dev *p_msgdma;                    // mSGDMA インスタンスへのポインタ
    alt_msgdma_standard_descriptor msgdma_desc; // スタンダード・ディスクリプタ構造体へのポインタ
    unsigned char *p_dma_rd;                     // DMA 転送元アドレスのポインタ
    unsigned char *p_dma_wr;                     // DMA 転送先アドレスのポインタ
    int status = 0;

    // DMA 転送元アドレスのポインタ（非キャッシュ領域でポインタ生成）
    p_dma_rd = (unsigned char*)( OCRAM0_BASE | 0x80000000 );
    // DMA 転送先アドレスのポインタ（非キャッシュ領域でポインタ生成）
    p_dma_wr = (unsigned char*)( OCRAM1_BASE | 0x80000000 );

    printf("<< Start Program!! >>¥n");

    //=========================================
    // オンチップ RAM のデータを初期化
    //=========================================

    // 転送元オンチップ RAM に DMA 転送サイズ分のランダム・データを準備
    // 転送先オンチップ RAM を DMA 転送サイズ分クリア
    init_ram( p_dma_rd, p_dma_wr, XFER_BYTES );

    //=========================================
    // DMA 転送前のオンチップ RAM データをダンプ表示
    //=========================================

    // 転送元オンチップ RAM データをダンプ表示
    printf( "¥n<Debug> On-Chip RAM 0 Contents Before DMA!!¥n" );
    dump_ram( p_dma_rd, XFER_BYTES );

    // 転送先オンチップ RAM データをダンプ表示
    printf( "¥n<Debug> On-Chip RAM 1 Contents Before DMA!!¥n" );
    dump_ram( p_dma_wr, XFER_BYTES );

    //=========================================
    // DMA 転送の実行
    //=========================================

    // mSGDMA インスタンスへのポインタを取得
    p_msgdma = alt_msgdma_open( MSGDMA_NAME );
    if ( p_msgdma == NULL )
    {
        printf( "Error: Could not open the mSGDMA¥n" );
        return -1;
    }
```

このサンプルの main 関数です

転送元 On-Chip RAM のアドレスを設定

転送先 On-Chip RAM のアドレスを設定

転送元 On-Chip RAM にランダムデータを準備し、
転送先 On-Chip RAM をゼロクリアしておきます

DMA 転送前に
転送元 および 転送先 On-Chip RAM のデータを
ダンプ表示して確認します

alt_msgdma_open() HAL API 関数を呼び出して
mSGDMA インスタンスへのポインタを取得します

次のページに続く

```
    // alt_msgdma_irq() からコールバックされる mSGDMA 転送コールバック関数を登録
    alt_msgdma_register_callback( p_msgdma,
                                  msgdma_callback,
                                  ALTERA_MSGDMA_CSR_GLOBAL_INTERRUPT_MASK |
                                  ALTERA_MSGDMA_CSR_STOP_ON_ERROR_MASK |
                                  ALTERA_MSGDMA_CSR_STOP_ON_EARLY_TERMINATION_MASK,
                                  p_msgdma );

    // キャッシュ・フラッシュ
    alt_dcache_flush_all();

    // DMA ディスクリプタを構成
    status = alt_msgdma_construct_standard_mm_to_mm_descriptor( p_msgdma,
                                                                &msgdma_desc,
                                                                (alt_u32*)p_dma_rd,
                                                                (alt_u32*)p_dma_wr,
                                                                XFER_BYTES,
                                                                ALTERA_MSGDMA_DESCRIPTOR_CONTROL_TRANSFER_COMPLETE_IRQ_MASK );
    if( status != 0 )
    {
        printf( "Error: mSGDMA descriptor Fail[%d]¥n", status );
        return status;
    }

    // DMA 転送を開始
    printf( "¥n<Debug> mSGDMA Transfer Start!!¥n" );
#if USE_ASYNC_XFER
    status = alt_msgdma_standard_descriptor_async_transfer( p_msgdma, &msgdma_desc );
    if ( 0 != status )
    {
        printf( "Error: mSGDMA async_transfer Fail[%d]¥n", status );
        return status;
    }
    // 転送完了（コールバック）待ちループ
    while (1)
    {
        // 転送完了であればループを抜ける
        if ( xfer_cmp == TRUE )
            break;
        // mSGDMA 転送エラーが検出されていればループを抜ける
        if ( xfer_err )
            break;
    }
#else
    status = alt_msgdma_standard_descriptor_sync_transfer( p_msgdma, &msgdma_desc );
    if ( 0 != status )
    {
        printf( "Error: mSGDMA sync_transfer Fail[%d]¥n", status );
        return status;
    }
#endif

    //=======================================
    // DMA 転送後のオンチップ RAM データをダンプ表示
    //=======================================

    // 転送元オンチップ RAM データをダンプ表示
    printf( "¥n<Debug> On-Chip RAM 0 Contents After DMA!!¥n" );
    dump_ram( p_dma_rd, XFER_BYTES );

    // 転送先オンチップ RAM データをダンプ表示
    printf( "¥n<Debug> On-Chip RAM 1 Contents After DMA!!¥n" );
    dump_ram( p_dma_wr, XFER_BYTES );
```

alt_msgdma_register_callback() HAL API 関数を呼び出して、msgdma_callback() 関数を登録します

alt_msgdma_construct_standard_mm_to_mm_descriptor() HAL API 関数を呼び出して DMA ディスクリプタを構成します

alt_msgdma_standard_descriptor_async_transfer() HAL API 関数
または、
alt_msgdma_standard_descriptor_sync_transfer() HAL API 関数
を呼び出して DMA 転送を開始します

alt_msgdma_standard_descriptor_async_transfer() HAL API 関数を使用して DMA 転送を行う場合は、転送完了時に割り込みハンドラーから msgdma_callback() コールバック関数が呼ばれ、xfer_cmp フラグに TRUE がセットされるはずなので、それをここで待ちます

DMA 転送後に
転送元 および 転送先 On-Chip RAM のデータを
ダンプ表示して確認します

次のページに続く

```
    //=======================================
    // DMA 転送後のオンチップ RAM データをベリファイ
    //=======================================

    printf( "¥n<Debug> On-Chip RAM 0 and 1 Contents Verify!!¥n" );
    status = verify_ram( p_dma_rd, p_dma_wr, XFER_BYTES );
    if ( 0 != status )
        printf( "<Debug> Data Verify Error!!¥n" );
    else
        printf( "<Debug> Data Verify OK!!¥n" );

    return status;
}

/*******************************************
 * mSGDMA 転送コールバック関数
 *
 *   引数 :   ユーザ定義コンテキストへのポインタ
 *   戻り値 : 無し
 */
void msgdma_callback( void* context )
{
    struct alt_msgdma_dev *p_msgdma = (alt_msgdma_dev*)context;

    printf( "<Debug> mSGDMA Callback Function called!!¥n" );

    // mSGDMA の コントロール・レジスタ と ステータス・レジスタ の値を取得
    int nCtrl = IORD_ALTERA_MSGDMA_CSR_CONTROL( p_msgdma->csr_base );
    int nStat = IORD_ALTERA_MSGDMA_CSR_STATUS( p_msgdma->csr_base );

    // 指定バイト数分転送完了したか?
    if ( nStat & ALTERA_MSGDMA_CSR_DESCRIPTOR_BUFFER_EMPTY_MASK )
    {
        // Yes: 転送完了フラグを立てる
        xfer_cmp = TRUE;
    }
    else
    {
        // それ以外の割り込みはエラーとする
        xfer_err = TRUE;
        printf( " mSGDMA_IRQ: stat_reg=%X. ctrl_reg=%X¥n", nStat, nCtrl );
    }
}

/*******************************************
 * RAM データ初期化関数
 *
 *   引数 : *p_rd_adrs - 転送元 RAM のアドレス・ポインタ
 *          *p_wr_adrs - 転送先 RAM のアドレス・ポインタ
 *               bytes - 転送バイト数
 *   戻り値 : 無し
 */
void init_ram( unsigned char *p_rd_adrs, unsigned char *p_wr_adrs, int bytes )
{
    int i;

    for ( i = 0; i < bytes; i++ )
    {
        // 転送元 RAM に DMA 転送サイズ分のランダム・データを準備
        p_rd_adrs[i] = rand();
    }
    // 転送先 RAM を DMA 転送サイズ分クリア
    memset( p_wr_adrs, 0x00, bytes );
}
```

DMA 転送後に
転送元 と 転送先の On-Chip RAM のデータを
ベリファイします

データベリファイの結果を表示

alt_msgdma_irq() 割り込みハンドラーからコールバックされる
mSGDMA 転送コールバック関数です

転送が完了していれば xfer_cmp フラグに TRUE をセット

転送元 On-Chip RAM にランダムデータを準備し、
転送先 On-Chip RAM をゼロクリアする関数です

次のページに続く

```
/*****************************************
 * RAM データ・ダンプ関数
 *
 *   引数： *p_ram_base - RAM のアドレス・ポインタ
 *              bytes - 転送バイト数（ダンプバイト数）
 *   戻り値： 無し
 */
void dump_ram( unsigned char *p_ram_adrs, int bytes )
{
    int i;
    unsigned char ucData;

    // キャッシュ・フラッシュ
    alt_dcache_flush_all();

    printf( "¥n0000: " );
    for ( i = 0; i < bytes; i++ )
    {
        ucData = p_ram_adrs[i];
        if(( i % 16 == 15 ) && ( i < bytes - 1 ))
            printf( "%02X ¥n%04X: ", ucData, i + 1 );
        else
            printf( "%02X ", ucData );
    }
    printf( "¥n" );
}

/*****************************************
 * RAM データ・ベリファイ関数
 *
 *   引数： *p_rd_adrs - 転送元 RAM のアドレス・ポインタ
 *         *p_wr_adrs - 転送先 RAM のアドレス・ポインタ
 *              bytes - 転送バイト数
 *   戻り値： 0 = エラー無し、 -1 = ベリファイ・エラー
 */
int verify_ram( unsigned char *p_rd_adrs, unsigned char *p_wr_adrs, int bytes )
{
    int i;
    int status = 0;

    for ( i = 0; i < bytes; i++ )
    {
        if ( p_rd_adrs[i] != p_wr_adrs[i] )
        {
            printf( "<Debug> Data Verify Error!! Address = %05d¥n", (unsigned int)i );
            status = -1;
        }
    }
    return status;
}
```

RAM のデータをダンプ表示する関数です

転送元 と 転送先の RAM のデータをベリファイする関数です

データが異なる場合は、そのメモリ・アドレスを表示します

終わり

【リスト 5-1】 mSGDMA サンプルのソースコード sample.c

6. サンプル内で使用している HAL API の説明

このサンプル内で使用している mSGDMA に関連する HAL API を以下に説明します。

6-1. alt_msgdma_open

【表 6-1】 alt_msgdma_open() 関数

関数名	`alt_msgdma_dev* alt_msgdma_open ( const char* name )`
機能	mSGDMA インスタンスへのポインタを取得します。

ヘッダーファイル	`#include <altera_msgdma_descriptor_regs.h>`
	`#include <altera_msgdma_csr_regs.h>`
	`#include <altera_msgdma.h>`

引数	* name - HAL に登録された mSGDMA ペリフェラルの名前へのポインタ。 たとえば、Platform Designer の mSGDMA は、「`MSGDMA_CSR_NAME`」を与えることによってオープンします。
戻り値	msgdma デバイス・インスタンス構造体へのポインタを返します。 デバイスが開けなかった場合は null を返します。
ソースコード	<Quartus インストール・フォルダ>¥ip¥altera¥altera_msgdma¥top¥HAL¥src¥altera_msgdma.c <Quartus インストール・フォルダ>¥ip¥altera¥altera_msgdma¥top¥HAL¥inc¥altera_msgdma.h
参考ドキュメント	Embedded Peripherals IP User Guide / 26.8.16. alt_msgdma_open（p.319） https://www.intel.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_embedded_ip.pdf#page=319

6-2. alt_msgdma_register_callback

【表 6-2】 alt_msgdma_register_callback () 関数

関数名	`void alt_msgdma_register_callback ( alt_msgdma_dev *dev,`
	`alt_msgdma_callback callback,`
	`alt_u32 control,`
	`void *context );`

機能	ユーザ固有のルーチンを mSGDMA 割り込みハンドラーに関連付けます。 コールバックが登録されている場合、すべての ノンブロッキング mSGDMA 転送はコールバックを実行させる割り込みを有効にします。 コールバックは割り込みサービスルーチンの一部として実行されるため、「Nios II ソフトウェア開発ハンドブック」に記載されている許容される割り込みサービスルーチン動作のガイドラインに従うことに十分注意しなければなりません。 ただし、この関数を呼び出すことで、ブロッキング転送の CSR 制御設定の一部を変更することができます。

ヘッダーファイル	`#include <altera_msgdma_descriptor_regs.h>`
	`#include <altera_msgdma_csr_regs.h>`
	`#include <altera_msgdma.h>`

引数	*dev - msgdma インスタンスへのポインタ。 callback - 割り込みレベルで実行するコールバック・ルーチンへのポインタ。 control - ノンブロッキングおよびブロッキング転送関数においてコントロール・レジスタの他の制御ビットに OR して設定する値。 *context - ユーザ定義コンテキストへのポインタ。
戻り値	無し
ソースコード	<Quartus インストール・フォルダ>¥ip¥altera¥altera_msgdma¥top¥HAL¥src¥altera_msgdma.c <Quartus インストール・フォルダ>¥ip¥altera¥altera_msgdma¥top¥HAL¥inc¥altera_msgdma.h
参考ドキュメント	Embedded Peripherals IP User Guide / 26.8.15. alt_msgdma_register_callback（p.318） https://www.intel.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_embedded_ip.pdf#page=318

6-3. alt_msgdma_construct_standard_mm_to_mm_descriptor

【表 6-3】 alt_msgdma_construct_standard_mm_to_mm_descriptor () 関数

関数名	int alt_msgdma_construct_standard_mm_to_mm_descriptor (alt_msgdma_dev *dev, alt_msgdma_standard_descriptor *descriptor, alt_u32 *read_address, alt_u32 *write_address, alt_u32 length, alt_u32 control);
機能	DMA ディスクリプタを構築します。 この関数は、mm_to_mm スタンダード・ディスクリプタを構築するためのヘルパー関数で、 "alt_msgdma_construct_standard_descriptor" を呼び出します。 不必要な引数は 0 に設定され、ハードウェアによって無視されます。
ヘッダーファイル	#include <altera_msgdma_descriptor_regs.h> #include <altera_msgdma_csr_regs.h> #include <altera_msgdma.h>
引数	*dev - msgdma インスタンスへのポインタ。 *descriptor - スタンダード・ディスクリプタ構造体へのポインタ。 *read_address - 転送元メモリのベースアドレスへのポインタ。 *write_address - 転送先メモリのベースアドレスへのポインタ。 length - ディスクリプタごとに転送するバイト数を指定するために使用されます。 入力できる最大値は "0xffffffff" です。 control - コントロール・フィールド。
戻り値	成功の場合は "0" を返します。 無効な引数の場合は -EINVAL を返し、ハードウェア設定値より大きな値を持つ引数が原因である可能性があります。
ソースコード	<Quartus インストール・フォルダ>¥ip¥altera¥altera_msgdma¥top¥HAL¥src¥altera_msgdma.c <Quartus インストール・フォルダ>¥ip¥altera¥altera_msgdma¥top¥HAL¥inc¥altera_msgdma.h
参考ドキュメント	Embedded Peripherals IP User Guide / 26.8.9. alt_msgdma_construct_standard_mm_to_mm_descriptor（p.312） https://www.intel.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_embedded_ip.pdf#page=312

6-4. alt_msgdma_standard_descriptor_async_transfer

【表 6-4】 alt_msgdma_standard_descriptor_async_transfer () 関数

関数名	`int alt_msgdma_standard_descriptor_async_transfer (`
	`  alt_msgdma_dev *dev,`
	`  alt_msgdma_standard_descriptor *desc );`

機能	DMA 転送を開始します。 この関数を呼び出すときには、ディスクリプタを構築した *desc へパラメータ・ポインタとして渡す必要があります。 この関数はヘルパー関数 "alt_msgdma_descriptor_async_transfer" を呼び出して、一度に 1 つのスタンダード・ディスクリプタのノンブロッキング転送を開始します。 この呼び出し時に読み込み/書き込みの FIFO バッファがいっぱいになると、ルーチンはすぐに -ENOSPC を返し、アプリケーションはブロックされずにどのように進めるかを決定することができます。 ディスクリプタをディスパッチャに書き込むための時間が 5 ミリ秒を超えると -ETIME が返されます。 詳細については、ヘルパー関数を参照することをお勧めします。 この特定の mSGDMA コントローラにコールバック・ルーチンが事前に登録されている場合、転送は割り込み生成を有効にするように設定されます。

ヘッダーファイル	`#include <altera_msgdma_descriptor_regs.h>`
	`#include <altera_msgdma_csr_regs.h>`
	`#include <altera_msgdma.h>`

引数	*dev - msgdma インスタンスへのポインタ。 *desc - スタンダード・ディスクリプタ構造体へのポインタ。
戻り値	成功の場合は "0" を返します。 -ENOSPC : FIFO バッファが一杯であることを示します。 -EPERM : ディスクリプタ・タイプの競合により許可されない動作を示します。 -ETIME : タイムアウトを示し、5 ミリ秒後にループをスキップします。
ソースコード	<Quartus インストール・フォルダ>¥ip¥altera¥altera_msgdma¥top¥HAL¥src¥altera_msgdma.c <Quartus インストール・フォルダ>¥ip¥altera¥altera_msgdma¥top¥HAL¥inc¥altera_msgdma.h
参考ドキュメント	Embedded Peripherals IP User Guide / 26.8.1. alt_msgdma_standard_descriptor_async_transfer（p.304） https://www.intel.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_embedded_ip.pdf#page=304

6-5. alt_msgdma_standard_descriptor_sync_transfer

【表 6-5】 alt_msgdma_standard_descriptor_sync_transfer () 関数

関数名	`int alt_msgdma_standard_descriptor_sync_transfer (`
	`  alt_msgdma_dev *dev,`
	`  alt_msgdma_standard_descriptor *desc );`

機能	この関数を呼び出すときには、スタンダード・ディスクリプタを構築し、*desc へのパラメータ・ポインタとして渡す必要があります。 この関数はヘルパー関数 "alt_msgdma_descriptor_sync_transfer" を呼び出して、一度に 1 つのスタンダード・ディスクリプタのブロック転送を開始します。 この呼び出しの時点で、リードまたはライト用 FIFO バッファがいっぱいになると、ルーチンは処理を続行するために空き FIFO バッファが使用可能になるか、または 5 ミリ秒のタイムアウトになるまで待機します。 この関数はディスパッチャがリードおよびライト・コマンドバッファが空になる前にリードマスタおよびライトマスタの両方にコマンドを発行させるエラーまたは条件が発生した場合、エラーを返します。 エラーと完了ステータスを確認するのはアプリケーション開発者の責任です。

ヘッダーファイル	`#include <altera_msgdma_descriptor_regs.h>`
	`#include <altera_msgdma_csr_regs.h>`
	`#include <altera_msgdma.h>`

引数	*dev - msgdma インスタンスへのポインタ。 *desc - スタンダード・ディスクリプタ構造体へのポインタ。
戻り値	成功の場合は "0" を返します。 エラーの場合は、 mSGDMA がマスタにコマンドを発行することを停止させるエラーまたは条件を示し、CSR ステータス・レジスタでエラーに設定されたビットをチェックすることを推奨します。 -EPERM: ディスクリプタ・タイプの競合により許可されていない操作を示します。 -ETIME: タイムアウトを示し、5 ミリ秒後にループをスキップします。
ソースコード	<Quartus インストール・フォルダ>¥ip¥altera¥altera_msgdma¥top¥HAL¥src¥altera_msgdma.c <Quartus インストール・フォルダ>¥ip¥altera¥altera_msgdma¥top¥HAL¥inc¥altera_msgdma.h
参考ドキュメント	Embedded Peripherals IP User Guide / 26.8.4. alt_msgdma_standard_descriptor_sync_transfer（p.307） https://www.intel.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_embedded_ip.pdf#page=307

※記載しているドキュメントのページ番号は資料バージョンによって変わることがあります

その他の mSGDMA に関連する情報は、Embedded Peripherals IP User Guide をご覧ください。

https://www.intel.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_embedded_ip.pdf

改版履歴

Revision	年月	概要
1	2018 年 8 月	初版