ALTIMA

Nios II SBT によるソフトウェア開発
セクション 1

ver.14

2020 年 6 月 Rev.3

ELSENA

ELSENA,Inc.

Nios II SBT によるソフトウェア開発 セクション 1

目次

1. はじめに 3
2. HAL を用いたプログラミング 3
2-1. HAL（Hardware Abstraction Layer） 3
2-2. Nios II SBT プロジェクトの構造 4
2-3. HAL API の使用例 5
2-3-1. システム・クロック 6
2-3-2. アラーム機能 7
2-3-3. タイムスタンプ・タイマ 8
2-4. HAL ファイル・システム 9
2-4-1. HAL ファイル・システム API 9
2-4-2. アプリケーション例 10
2-4-3. ファイル・アクセス関数 11
2-4-4. デバイスへのダイレクト・アクセス 11
3. ペリフェラルへのアクセス方法 12
改版履歴 14

1. はじめに

この資料では、Nios® II Software Build Tool(Nios II SBT)を用いたソフトウェア開発について紹介します。

❏ Nios II の HAL を用いたプログラミング

- HAL(Hardware Abstraction Layer)
- Nios II SBT プロジェクト構造
- HAL API の例(タイマとアラームの設定について)
- HAL ファイル・システム

❏ ペリフェラルへのアクセス方法

2. HAL を用いたプログラミング

この章では、Nios II プログラムで使用する HAL について説明します。

2-1. HAL(Hardware Abstraction Layer)

HAL システム・ライブラリはデバイス・ドライバ・インターフェースを提供します。プログラムはこのインターフェースを使用してハードウェアにアクセスすることができます。

HAL システム・ライブラリは以下のサービスを提供します。

- newlib ANSI C 標準ライブラリとの統合

HAL アプリケーション・プログラム・インターフェース(API)は、ANSI C 標準ライブラリと統合されているため、printf() や fopen() 等の一般的な C 標準関数を使用することができます。

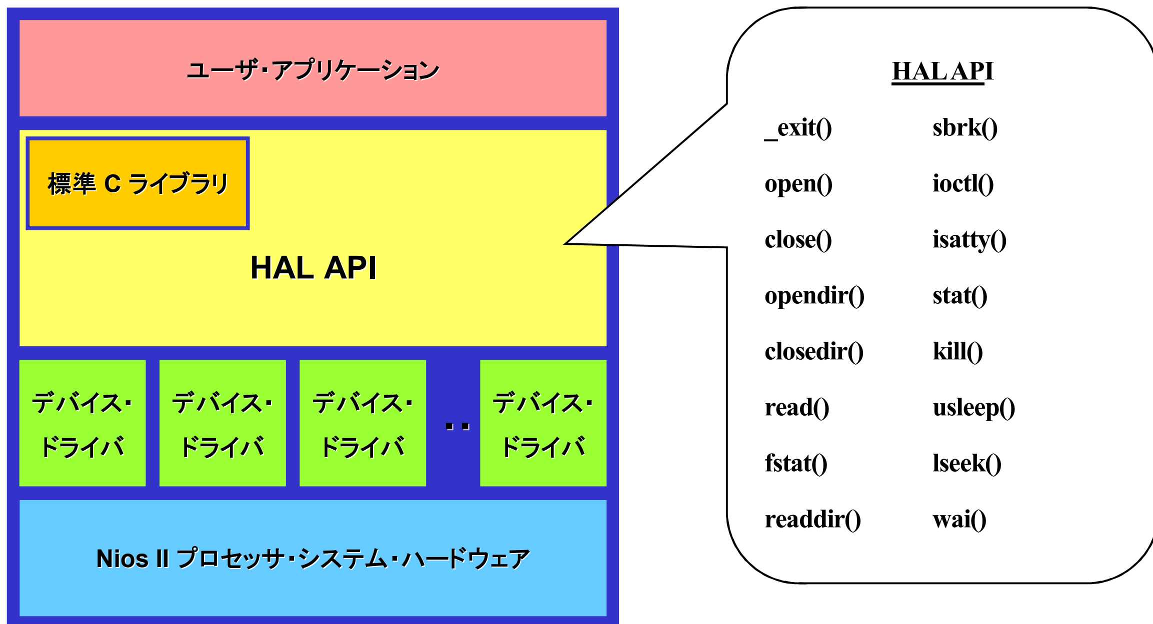
- デバイス・ドライバ

HAL システム・ライブラリは、特定の Qsys システム上に構築されます。Qsys システム内に組み込まれているペリフェラルのデバイス・ドライバ・ソフトウェアをリンクし構成されます。

- HAL API

デバイス・アクセス、割り込み処理、アラーム機能などの HAL サービスへの標準インターフェースを提供します。

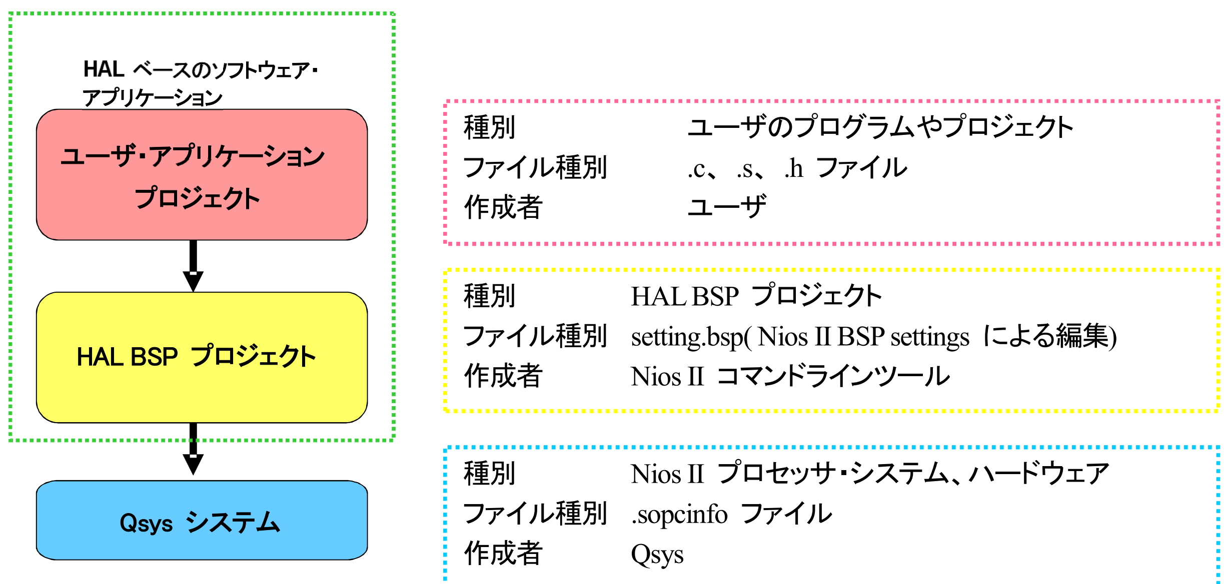
HAL ベースシステムのレイヤ



2-2. Nios II SBT プロジェクトの構造

HAL システム・ライブラリをベースとしたソフトウェア・プロジェクトの作成と管理は、Nios II Software Build Tools (Nios II SBT) に統合されています。

下図は HAL システム・ライブラリの組み込み方法に重点を置いた Nios II プログラム・ブロックを示します。HAL ベースのシステムは、図に示すように 2 つの Nios II SBT プロジェクトから成ります。ユーザが開発するすべてのプログラムはユーザ・アプリケーション・プロジェクトに含まれ、もう一方の BSP プロジェクト(HAL BSP プロジェクト)には、プロセッサやハードウェアとのインターフェースに関連するすべての情報が含まれます。

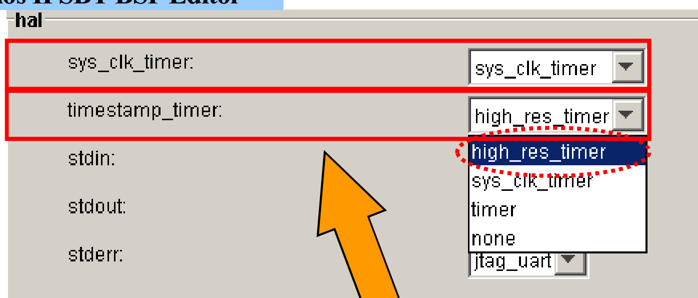


2-3. HAL API の使用例

タイマを例にして HAL API の使用方法を説明します。

HAL API は、2 種類のタイマ・デバイス・ドライバを提供します。システム内に複数のタイマ・ペリフェラルを組み込むことができます。1 つのハードウェアとしてのひとつのタイマ・ペリフェラルは、 sys_clk_timer または timestamp_timer のどちらかとして一方のみ動作可能で、両方同時に動作することはできません。 Nios II SBT の BSP Editor の Main タブで、使用するタイマ・ペリフェラルを選択します。このタイマ・ペリフェラルは、Qsys 内の Interval timer コアを使用します。

- システム・クロック・タイマ ： システム時刻とアラーム機能を実現します。
- タイムスタンプ・タイマ ： 高精度の時間測定を可能にします。



Qsys

		Avalon Memory Mapped Slave
☑	jtag_uart	JTAG UART
	avalon_jtag_slave	Avalon Memory Mapped Slave
☑	sys_clk_timer	Interval Timer
	s1	Avalon Memory Mapped Slave
☑	high_res_timer	Interval Timer
	s1	Avalon Memory Mapped Slave
☑	ext_flash	Flash Memory Interface (CFI)
	s1	Avalon Memory Mapped Tristate Slave
☑	lan91c111	LAN91C111 Interface

2-3-1. システム・クロック

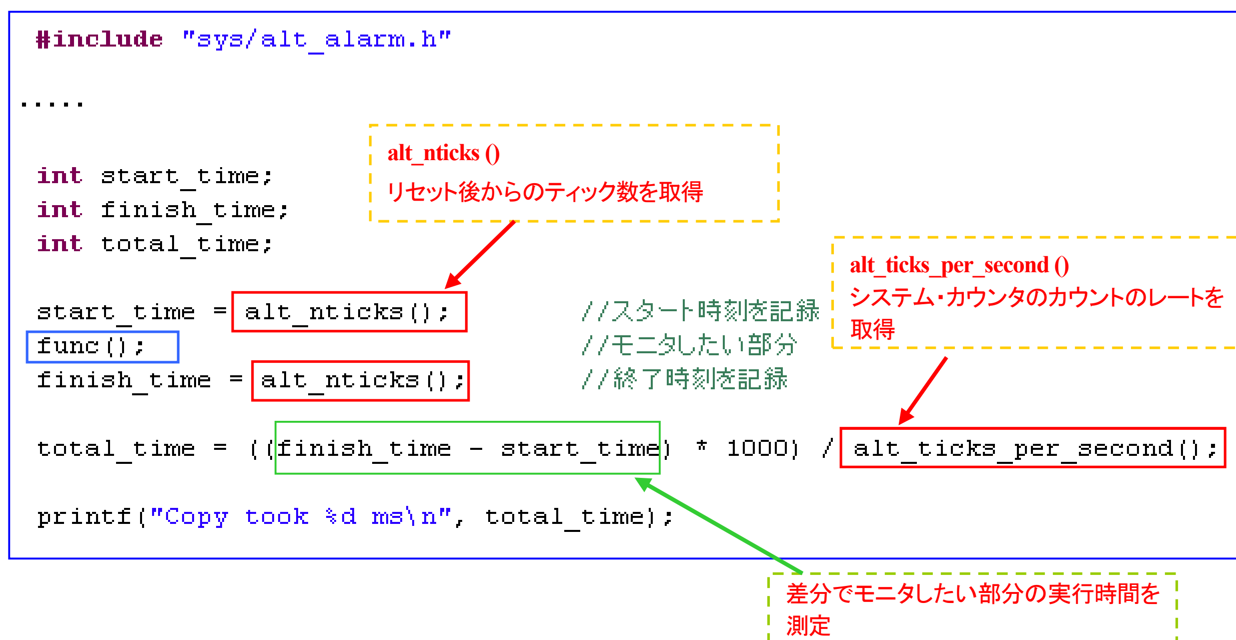
システム・クロック・タイマはシステム時刻の基本単位"ティック"を提供します。"ティック"とは、 Qsys 内のタイマ・ペリフェラルへ入力しているクロック周期の数で、例えば 100MHz を入力していたら 10ns が単位時間となります。ティックで指定した単位時間ごとにシステム・クロックをカウントアップし、指定した時間に関数の実行や時間の測定を行うことができます。

下記の関数 HAL API を使用して制御することができます。

- alt_nticks() ： リセット以降の経過時間をシステム・クロックのティック数で取得します。
- alt_ticks_per_second() ： システム・クロック・レート(ティック/秒)を取得します。

下記の例では、測定したい部分(func() 関数)の前後で alt_nticks() 関数を使用して、ティックのカウント数を取得し、ティック数の差分から実行時間を測定しています。

例） 簡易的な時間計測方法


```
#include "sys/alt_alarm.h"

.....

int start_time;
int finish_time;
int total_time;

start_time = alt_nticks();          //スタート時刻を記録
func();                             //モニタしたい部分
finish_time = alt_nticks();         //終了時刻を記録

total_time = ((finish_time - start_time) * 1000) / alt_ticks_per_second();

printf("Copy took %d ms\n", total_time);
```

2-3-2. アラーム機能

HAL アラーム機能を利用して、指定した時間に実行する関数を登録することができます。アラームは下記 HAL を使用して登録、キャンセルを行います。

- alt_alarm_start() ： アラームの登録

```
int alt_alarm_start ( alt_alarm*    alarm,                        // アラームを表す構造体へのポインタ
                      alt_u32       nticks,                       // callback を呼ぶまでの時間(ティック数で指定)
                      alt_u32       (*callback) (void* context),  // callback 関数へのポインタ
                      void*         context );                    // callback 関数への入力引数
```

※ alarm はこのアラームを表す構造体へのポインタです。このポインタはユーザが作成する必要があり、作成した構造体はアラームが有効になっている間よりも長い間有効である必要があります。alarm が指し示す構造体の内容をユーザが初期化する必要はありません。

- alt_alarm_stop() ： 以前に登録されたアラームのキャンセル

```
void alt_alarm_stop (alt_alarm* alarm);    // 引数は呼び出しに使用したアラーム構造体へのポインタ
```

例） 周期的なアラーム・コールバック関数の使用

```
#include <stdio.h>
#include "system.h"
#include "sys/alt_alarm.h"
#include <stddef.h>

/* コールバック関数 */
alt_u32 my_alarm_callback( void* context )
{
    (*(alt_u8 *)context)++;

    printf("Callback¥n");
    return 3*alt_ticks_per_second();
}

int main(void)
{
    static alt_alarm alarm;

    printf("Hello from Nios II, start!!¥n");

    if(alt_alarm_start(&alarm,
                       (3*alt_ticks_per_second()),
                       my_alarm_callback,
                       NULL) < 0 )
    {
        printf("No system clock available¥n");
    }

    while(1)
    {
        asm("nop");
    }

    return 0;
}
```

コールバック関数

戻り値で次回 callback 呼び出しまでのティック数を指定。この場合は 3 秒間隔で my_alarm_callback 関数が呼び出される。

コールバック関数の登録

callback 関数を呼び出すまでの時間。この場合は 3 秒。

2-3-3. タイムスタンプ・タイマ

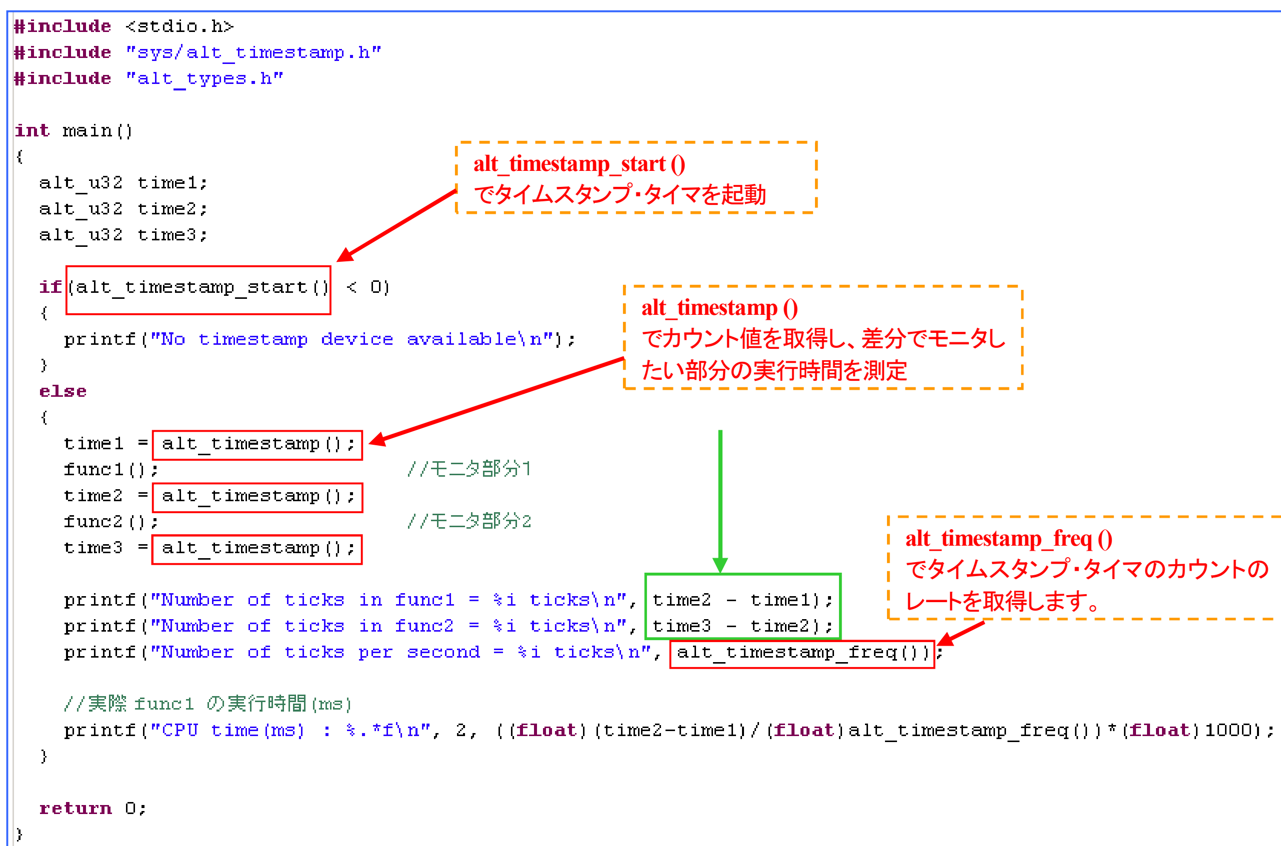
HAL システム・クロックのティックで得られるサンプリング周期以上の精度での時間間隔の測定が必要な場合には、タイムスタンプ・タイマを使用することができます。タイムスタンプ・タイマは単調に増加するカウンタを提供するため、このカウンタをサンプリングして時間計測に使用することができます。HAL はシステム内に 1 つのタイムスタンプ・タイマのみサポートします。タイムスタンプ・タイマは alt_timestamp.h ファイルで定義されます。

タイムスタンプ・ドライバを使用する場合には必ず Nios II SBT の BSP Editor にて Timestamp Timer が設定されている必要があります。デフォルトでは設定されていません。

タイムスタンプ・タイマが存在すれば、下記の HAL API を使用することができます。

- alt_timestamp_start () : カウンタが動作を開始します。この関数を呼び出すたびにタイムスタンプ・タイマの値はゼロにリセットされます。タイマ・コアは 32 ビット、もしくは 64 ビット(コアのオプション設定にて変更可能)がサポートされ、カウンタの値は 2^{32} -1、 もしくは 2^{64} -1 までカウントすることができ、それ以降は 0 で停止します。
- alt_timestamp() : タイムスタンプ・タイマの現在の値を返します。
- alt_timestamp_freq() : タイムスタンプ・カウンタの増加レートを取得します。このレートは Timestamp timer に指定した Interval Timer の動作周波数です。

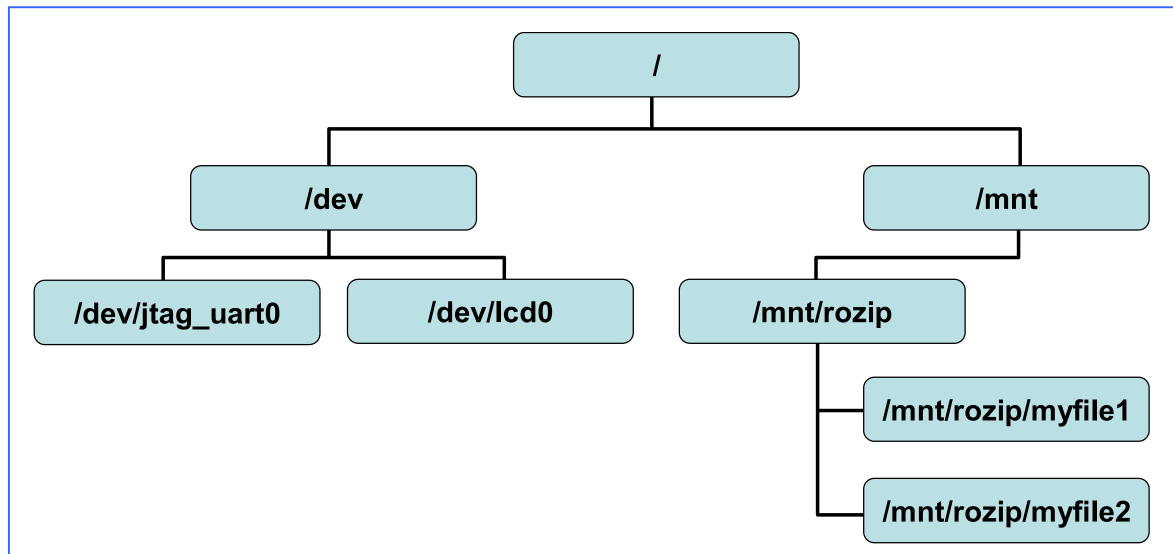
例) コード実行時間を測定するためのタイムスタンプ・タイマの使用




```
#include <stdio.h>
#include "sys/alt_timestamp.h"
#include "alt_types.h"

int main()
{
  alt_u32 time1;
  alt_u32 time2;
  alt_u32 time3;

  if(alt_timestamp_start() < 0)
  {
    printf("No timestamp device available\n");
  }
  else
  {
    time1 = alt_timestamp();
    func1();                    //モニタ部分1
    time2 = alt_timestamp();
    func2();                    //モニタ部分2
    time3 = alt_timestamp();

    printf("Number of ticks in func1 = %i ticks\n", time2 - time1);
    printf("Number of ticks in func2 = %i ticks\n", time3 - time2);
    printf("Number of ticks per second = %i ticks\n", alt_timestamp_freq());

    //実際 func1 の実行時間(ms)
    printf("CPU time(ms) : %.*f\n", 2, ((float)(time2-time1)/(float)alt_timestamp_freq())*(float)1000);
  }

  return 0;
}
```

2-4. HAL ファイル・システム

HAL はキャラクタ・モードのデバイスおよびデータ・ファイル・システムを扱うためにファイル・システムの概念を持ちます。newlib が提供する C 標準ライブラリのファイル I/O 関数（fopen()、fread() 等）、または HAL システム・ライブラリが提供する UNIX 形式のファイル I/O を使用して、キャラクタ・デバイスにアクセスできます。



ファイル・システムは、自身をグローバル HAL ファイル・システム内のマウント・ポイントとして登録します。マウント・ポイントの下にあるファイルにアクセスを試みると、アクセスはそのファイル・サブシステムに対して実行されます。例えば、zip ファイル・サブシステムが /mnt/rozip として登録されている場合、/mnt/rozip/myfile1 を対象とした fopen() のコールは、関連付けされた zipfs ファイル・サブシステムによって処理されます。

同様にキャラクタ・モード・デバイスは HAL ファイル・システム内のノードとして登録します。慣例的に、system.h ファイル内では、デバイス・ノード名は、プリフィックス /dev に続いて Qsys でハードウェア・コンポーネントに割り当てられた名前を付加して定義されます。例えば、Qsys での JTAG UART ペリフェラルが jtag_uart0 という名前で割り当てられていると system.h ファイルでは /dev/jtag_uart0 と定義されます。

HAL ファイル・システムではカレント・ディレクトリという概念はなく、すべてのファイルは絶対パスを通してアクセスする必要があります。

2-4-1. HAL ファイル・システム API

- ❏ Newlib 標準 C ライブラリ、I/O 関数を用いたファイル・システム・デバイス（ファイル）へのアクセス

  fopen()、fclose()、fread()　…etc

- ❏ HAL UNIX スタイルのファイル I/O

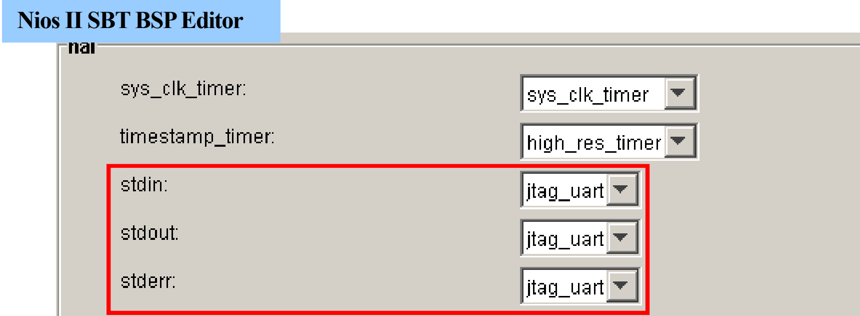
  close()、open()、fstat()、read()、ioctl()、stat()、isatty()、write()、lseek()

2-4-2. アプリケーション例

❏ 標準入力、標準出力、標準エラー

シンプルなコンソール I/O を実装するには、標準入出力(stdin、stdout、stderr)を使用する方法がもっとも簡単な方法です。HAL システム・ライブラリは背後で stdin、stdout、stderr を管理するため、ファイル・ディスクリプタを明示的に管理することなく、これらのチャンネルを介してキャラクタを送信、受信することができるようになります。

Nios II SBT の BSP Editor にて、特定のハードウェア・デバイスへの関連付けを行います。



例) Hello World

```
#include <stdio.h>

int main()
{
  printf("Hello from Nios II!\n");

  return 0;
}
```

定番の Hello World プログラム
Nios II SBT でビルドした際に stdout に関連付けられるデバイスにキャラクタが送信されます。

❏ キャラクタ・デバイスへの汎用アクセス

キャラクタ・モード・デバイス(stdin、stdout、stderr を除く)へのアクセスは、ファイルを開いたり、ファイルに書き込んだりするのと同様です。

例) JTAG UART へのキャラクタの書き込み

```
#include <stdio.h>
#include <string.h>

int main()
{
  char* msg = "Hello World";
  FILE* fp;

  fp = fopen ("/dev/jtag_uart", "w");
  if (fp)
  {
    fprintf(fp, "%s", msg);
    fclose (fp);
  }
  return 0;
```

Hello World の文字を JTAG UART に送信します。
デバイスは Qsys でのコンポーネント名で定義されます。
system.h ファイル内で
#define JTAG_UART_NAME "/dev/jtag_uart"
のように定義されています。

2-4-3. ファイル・アクセス関数

使い慣れたファイル・アクセス関数を HAL/Newlib が提供します。

- ANSI C

```
fd = fopen ( " / dev / lcd0 " , " w " );
fprintf ( fp , " %s " , msg );
```

- UNIX スタイル

```
fd = open ( " / dev / / lcd0 " , 0_WRONLY );
write ( fd , msg , strlen ( msg ) );
```

- Newlib は C++ ストリームもサポート

```
ofstream ofp ( " / dev / lcd0 / " , ios : : out );
ofp << msg ;
```

※ その他の UNIX スタイル関数

- int **usleep** (int us);
- int **settimeofday** (const struct timeval *t , const struct timezone *tz);
- int **gettimeofday** (struct timeval *ptimeval . struct timezone *ptimezone);

2-4-4. デバイスへのダイレクト・アクセス

特定のデバイスには、汎用 API では補足できない使用条件を持つハードウェア固有の機能があります。HAL システム・ライブラリは、UNIX 形式の ioctl () 関数を提供することによって、ハードウェア固有の機能に対応しています。ハードウェアの機能はペリフェラルごとに異なるため、各ペリフェラルがサポートする ioctl 機能の詳細は、ペリフェラルのドキュメントを確認する必要があります。

3. ペリフェラルへのアクセス方法

この章では、Nios II から接続されている各ペリフェラルへのアクセスについて説明します。

データ・キャッシュのバイパス

Nios II プロセッサ・コアはインストラクション・キャッシュとデータ・キャッシュを搭載することができます。インストラクション・キャッシュのみを持つのか、インストラクション・キャッシュとデータ・キャッシュ両方持つのかはそれぞれのコアによって異なります。Nios II/Fast コアはインストラクション・キャッシュとデータ・キャッシュ、Nios II/Standard コアはインストラクション・キャッシュ、Nios II/Economy コアはキャッシュを搭載することはできません。

Nios II のアドレス空間は 2GByte のミラーリングになっています。下位 2GByte がキャッシュ領域、上位 2GByte は非キャッシュ領域で、最上位の Bit 31 が制御ビットとなっています。 Fast コアを使用した場合、すべての変数データはデフォルトでキャッシュされます。データ・キャッシュはダイレクト・マップ、ライト・バック方式です。ハードウェアへアクセスする場合には、ソフトウェアでキャッシュをバイパスする操作が必要です。

データ・キャッシュをバイパスするためのマクロが用意されています。IOWR、IORD を使用してハードウェアに直接アクセスする記述をします。使用方法は以下のようになります。IOWR は stwio に IORD は ldwio 命令のアセンブリ命令に展開されます。

マクロ	用途
IORD (BASE, REGNUM)	ベース・アドレス BASE のデバイス内部のオフセット REGNUM のレジスタの値を読み込みます。レジスタはバスのアドレス幅だけオフセットされていると仮定されます。
IOWR (BASE, REGNUM, DATA)	ベース・アドレス BASE のデバイス内部のオフセット REGNUM のレジスタに DATA の値を書き込みます。レジスタはバスのアドレス幅だけオフセットされていると仮定されます。
IORD_32DIRECT (BASE, OFFSET)	アドレス BASE+OFSET の位置で 32 ビットの読み込みアクセスを行います。
IOWR_32DIRECT (BASE, OFFSET, DATA)	アドレス BASE+OFSET の位置で 32 ビットの書き込みアクセスを行います。
IORD_16DIRECT (BASE, OFFSET)	アドレス BASE+OFSET の位置で 16 ビットの読み込みアクセスを行います。
IOWR_16DIRECT (BASE, OFFSET, DATA)	アドレス BASE+OFSET の位置で 16 ビットの書き込みアクセスを行います。
IORD_8DIRECT (BASE, OFFSET)	アドレス BASE+OFSET の位置で 8 ビットの読み込みアクセスを行います。
IOWR_8DIRECT (BASE, OFFSET, DATA)	アドレス BASE+OFSET の位置で 8 ビットの書き込みアクセスを行います。

※ IOWR、IORD マクロと volatile 宣言

ポインタを volatile として宣言し、この volatile ポインタを使用してアクセスしてもデータ・キャッシュをアクセスできないことに注意してください。volatile 宣言はコンパイラに対してアクセスを最適化しないようにするための指示です。IORD / IOWR マクロは、データ・キャッシュのバイパスと volatile 宣言を両方行ったものと同様の効果です。

Qsys に提供されているコンポーネントにはハードウェア・インターフェースを定義するヘッダ・ファイルが提供されています。各ヘッダ・ファイルは<コンポーネント名>_regs.h の名前で各コンポーネントの inc ディレクトリに保存されています。これらのヘッダ・ファイルを必要に応じてインクルードすることで、HAL を使用可能になります。例えば PIO のコンポーネントの場合には、以下のファイルが保存されています。

<インストールパス>¥ip¥altera¥sopc_builder_ip¥altera_avalon_pio¥inc に altera_avalon_pio_regs.h

<コンポーネント名>_regs.h ファイル内では、以下の内容が定義されています。

- 各レジスタにアクセスするための読み取りマクロや書き込みマクロ
  - ***IOWR_<コンポーネント名>_<レジスタ名>***
  - ***IORD_<コンポーネント名>_<レジスタ名>***
- レジスタ内の個々ビット・フィールドへのアクセスを可能にする、ビット・フィールド・マクロとオフセット
  - フィールド・ビット・マスク

    <コンポーネント名>_<レジスタ名>_<フィールド名>_MSK
  - フィールド・先頭のビット・オフセット

    <コンポーネント名>_<レジスタ名>_<フィールド名>_OFST

ペリフェラル用ヘッダ・ファイルの例　altera_avalon_dma.h

```
#define IOADDR_ALTERA_AVALON_DMA_RADDRESS(base)      __IO_CALC_ADDRESS_NATIVE(base, 1)
#define IORD_ALTERA_AVALON_DMA_RADDRESS(base)        IORD(base, 1)
#define IOWR_ALTERA_AVALON_DMA_RADDRESS(base, data)  IOWR(base, 1, data)

#define IOADDR_ALTERA_AVALON_DMA_WADDRESS(base)      __IO_CALC_ADDRESS_NATIVE(base, 2)
#define IORD_ALTERA_AVALON_DMA_WADDRESS(base)        IORD(base, 2)
#define IOWR_ALTERA_AVALON_DMA_WADDRESS(base, data)  IOWR(base, 2, data)

#define IOADDR_ALTERA_AVALON_DMA_LENGTH(base)        __IO_CALC_ADDRESS_NATIVE(base, 3)
#define IORD_ALTERA_AVALON_DMA_LENGTH(base)          IORD(base, 3)
#define IOWR_ALTERA_AVALON_DMA_LENGTH(base, data)    IOWR(base, 3, data)

#define IOADDR_ALTERA_AVALON_DMA_CONTROL(base)       __IO_CALC_ADDRESS_NATIVE(base, 6)
#define IORD_ALTERA_AVALON_DMA_CONTROL(base)         IORD(base, 6)
#define IOWR_ALTERA_AVALON_DMA_CONTROL(base, data)   IOWR(base, 6, data)

#define ALTERA_AVALON_DMA_CONTROL_BYTE_MSK             (0x1)
#define ALTERA_AVALON_DMA_CONTROL_BYTE_OFST            (0)
#define ALTERA_AVALON_DMA_CONTROL_HW_MSK               (0x2)
#define ALTERA_AVALON_DMA_CONTROL_HE_OFST              (1)
#define ALTERA_AVALON_DMA_CONTROL_WORD_MSK             (0x4)
#define ALTERA_AVALON_DMA_CONTROL_WORD_OFST            (2)
```

改版履歴

Revision	年月	概要
1	2015 年 5 月	初版
2	2016 年 8 月	誤記訂正 ・ 「3. ペリフェラルへのアクセス方法」の説明（表中） ーIOWR_32DIRECT (BASE, OFFSET, DATA) 誤）読み込み 正）書き込み ーIORD_8DIRECT (BASE, OFFSET) 誤）書き込み 正）読み込み
3	2020 年 6 月	・"最終ページ"：「免責およびご利用上の注意」内のリンクを修正

免責およびご利用上の注意

弊社より資料を入手されましたお客様におかれましては、下記の使用上の注意を一読いただいた上でご使用ください。

1. 本資料は非売品です。許可無く転売することや無断複製することを禁じます。
2. 本資料は予告なく変更することがあります。
3. 本資料の作成には万全を期していますが、万一ご不明な点や誤り、記載漏れなどお気づきの点がありましたら、本資料を入手されました下記代理店までご一報いただければ幸いです。

   株式会社マクニカ 半導体事業 お問い合わせフォーム
4. 本資料で取り扱っている回路、技術、プログラムに関して運用した結果の影響については、責任を負いかねますのであらかじめご了承ください。
5. 本資料は製品を利用する際の補助的な資料です。製品をご使用になる際は、各メーカ発行の英語版の資料もあわせてご利用ください。