

この資料は、更新された最新の英語版が存在します。こちらの日本語版は参考用としてご利用下さい。設計の際は、必ず最新の英語版で内容をご確認下さい。

NI152004-1.2

はじめに

この章では、アルテラの HAL (Hardware Abstraction Layer) システム・ライブラリをベースとして、プログラムを開発する方法について説明します。

HAL ベース・システムの API は、Nios® II プロセッサを初めて使用するソフトウェア開発者の方でも、容易に利用できます。HAL をベースとしたプログラムは、ANSI C 標準ライブラリ関数とランタイム環境を使用し、HAL API の汎用デバイス・モデルを介してハードウェア・リソースにアクセスします。ANSI C 標準ライブラリは HAL システム・ライブラリから独立していますが、HAL API の大部分は、使い慣れた ANSI C 標準ライブラリ関数で定義されています。ANSI C 標準ライブラリと HAL は緊密に統合されているため、HAL システム・ライブラリ関数を直接呼び出さない有効なプログラムを開発することができます。例えば、`printf()`、`scanf()` などの ANSI C 標準ライブラリ I/O 関数を使用して、キャラクタ・モードのデバイスとファイルを操作できます。

この章では、HAL システム・ライブラリ API を使用するための基本的な参考情報を記載します。いくつかのトピックは、他の章で詳細に扱われています。この章で扱っていない以下の重要なトピックについては、目次を参照してください。

- デバイス・ドライバ、およびハードウェアと直接連携するコードの記述
- 例外処理および割り込みサービス・ルーチン
- キャッシュ・メモリを構成するためのプログラミング
- リアル・タイム・オペレーティング・システム (RTOS)
- イーサネット

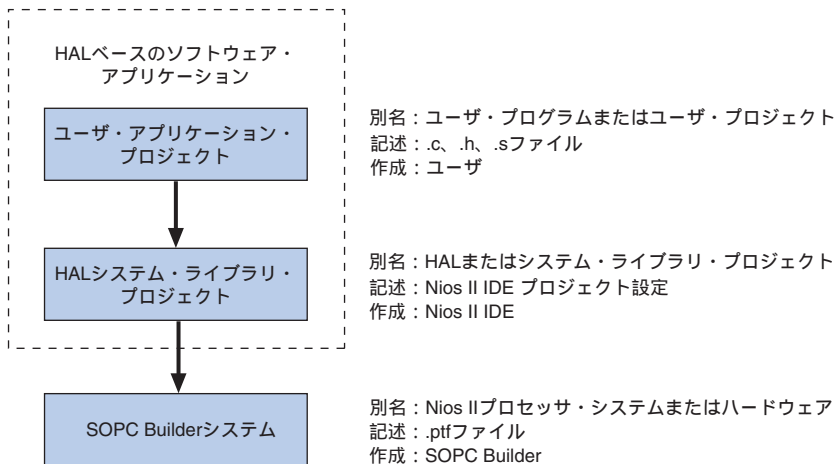
 本書では、ANSI C 標準ライブラリについては説明していません。

Nios II IDE プロジェクト 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 統合開発環境 (IDE) に緊密に統合されています。このセクションでは、HAL を理解するための基礎として、Nios II IDE プロジェクトについて説明します。

図 4-1 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 プログラムのブロックを示します。各ブロックのラベルはブロックの作成元または作成者を示し、矢印は各ブロック間の依存関係を示します。

図 4-1. Nios II IDE プロジェクトの構造



HAL ベースのシステムは、図 4-1 に示すように、2 つの Nios II IDE プロジェクトを使用して構築されます。ユーザのプログラムは、1 つのプロジェクト (ユーザー・アプリケーション・プロジェクト) に含まれ、別のシステム・ライブラリ・プロジェクト (HAL システム・ライブラリ・プロジェクト) に依存します。アプリケーション・プロジェクトには、ユーザが開発するすべてのコードが含まれています。プログラムの実行可能イメージは、このプロジェクトをビルドしたものを土台として作成されます。HAL システム・ライブラリ・プロジェクトには、プロセッサ・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ています。システム・ライブラリ・プロジェクトは Nios II プロセッサ・システムに依存し、SOPC Builder で生成された .ptf ファイルによって定義されています。

このプロジェクトの依存関係のため、SOPC Builder システムが変更された（つまり、.ptf ファイルが更新された）場合でも、Nios II IDE によって HAL システム・ライブラリが管理され、システム・ハードウェアを正確に反映するようにドライバ・コンフィギュレーションが更新されます。HAL システム・ライブラリによって、ユーザのプログラムは基本ハードウェアが変更されても影響を受けることはありません。そのため、ユーザは、自分のプログラムがターゲット・ハードウェアに適合するかどうかを気にすることなく、コードの開発とデバッグを実行できます。つまり、HAL システム・ライブラリをベースとするプログラムは、常にターゲット・ハードウェアと同期化されます。

system.h システム記述 ファイル

`system.h` ファイルは、HAL システム・ライブラリの基礎となります。`system.h` ファイルには、Nios II システム・ハードウェアのソフトウェア記述がすべて含まれています。このファイルは、ハードウェアおよびソフトウェアのデザイン・プロセス間における引き渡し点となります。`system.h` のすべての情報が、必ずしもプログラマに役立つとは限りませんが、C ソース・ファイルで明示的に指定する必要があるとも限りません。しかし、`system.h` には、「このシステムにはどのようなハードウェアが存在するか？」という基本的な疑問に対する解答があります。

`system.h` ファイルには、システム内の各ペリフェラルの記述と以下の詳細情報が入っています。

- ペリフェラルのハードウェア・コンフィギュレーション
- ベース・アドレス
- IRQ の優先順位（該当する場合）
- ペリフェラルの識別名

`system.h` ファイルは、絶対に編集しないでください。`system.h` ファイルは、HAL システム・ライブラリ・プロジェクト用に Nios II IDE によって自動的に生成されます。`system.h` の内容は、ユーザが Nios II IDE で設定するハードウェア・コンフィギュレーションおよび HAL システム・ライブラリ・プロパティの両方に依存します。



詳細については、Nios II IDE オンライン・ヘルプを参照してください。

system.h ファイル内の以下のコードは、このファイルが定義するハードウェア・コンフィギュレーションの一部を示します。

例：system.h ファイルの一部

```

/*
 * sys_clk_timer configuration
 *
 */

#define SYS_CLK_TIMER_NAME "/dev/sys_clk_timer"
#define SYS_CLK_TIMER_TYPE "altera_avalon_timer"
#define SYS_CLK_TIMER_BASE 0x00920800
#define SYS_CLK_TIMER_IRQ 0
#define SYS_CLK_TIMER_ALWAYS_RUN 0
#define SYS_CLK_TIMER_FIXED_PERIOD 0

/*
 * jtag_uart configuration
 *
 */

#define JTAG_UART_NAME "/dev/jtag_uart"
#define JTAG_UART_TYPE "altera_avalon_jtag_uart"
#define JTAG_UART_BASE 0x00920820
#define JTAG_UART_IRQ 1
    
```

データ幅と HAL 型の定義

Nios II プロセッサなどのエンベデッド・プロセッサでは、多くの場合、データの正確な幅と精度を知ることが重要になります。ANSI C のデータ型では、データ幅が明示的に定義されていないため、HAL は代わりに標準型定義のセットを使用します。ANSI C 型もサポートされていますが、これらのデータ幅はコンパイラの規約に依存します。

ヘッダ・ファイル alt_type.h では、HAL 型定義を定義しています。[表 4-1](#) に HAL 型定義を示します。

表 4-1. HAL 型定義	
型	意味
alt_8	符号付 8 ビット整数
alt_u8	符号なし 8 ビット整数
alt_16	符号付 16 ビット整数
alt_u16	符号なし 16 ビット整数
alt_32	符号付 32 ビット整数
alt_u32	符号なし 32 ビット整数

表 4-2 に、アルテラが提供する GNU Toolchain で使用するデータ幅を示します。

型	意味
char	8 ビット
short	16 ビット
long	32 ビット
int	32 ビット

UNIX 形式の インタフェース

HAL API は、多数の UNIX 形式の関数を提供します。UNIX 形式の関数によって、新たに Nios II を使用するプログラマにも親しみやすい開発環境が提供され、既存のコードを移植して HAL 環境で実行させるための作業が容易になります。HAL は主にこれらの関数を使用して、ANSI C 標準ライブラリ用のシステム・インタフェースを提供します。例えば、これらの関数は、`stdio.h` で定義された C ライブラリ関数が必要とするデバイス・アクセスを実行します。

以下に、利用可能な UNIX 形式の関数の全リストを示します。

- `_exit()`
- `close()`
- `fstat()`
- `getpid()`
- `gettimeofday()`
- `ioctl()`
- `isatty()`
- `kill()`
- `lseek()`
- `open()`
- `read()`
- `sbrk()`
- `settimeofday()`
- `stat()`
- `usleep()`
- `wait()`
- `write()`

最もよく使用される関数は、ファイル I/O に関するものです。
4-6 ページの「[ファイル・システム](#)」を参照してください。

ファイル・システム



これらの関数の使用法の詳細については、10-1 ページの「HAL API リファレンス」を参照してください。

HAL は、キャラクタ・モードのデバイスおよびデータ・ファイルを操作するのに使用可能なファイル・システムのコンセプトを提供します。newlib が提供する C 標準ライブラリのファイル I/O 関数 (`fopen()`、`fclose()`、`fread()` など) 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、ファイル・システム内のファイルにアクセスできます。

HAL では、ファイル操作に以下の UNIX 形式の関数を利用できます。

- `close()`
- `fstat()`
- `ioctl()`
- `isatty()`
- `lseek()`
- `open()`
- `read()`
- `stat()`
- `write()`



これらの関数の詳細については、10-1 ページの「HAL API リファレンス」を参照してください。

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 `/mount/zipfs()` としてマウントされている場合、`/mount/zipfs()/myfile` を対象とした `fopen()` のコールは、関連付けられた 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` ファイルでは、デバイス・ノード名は、プリフィックス `/dev/` に続いて、SOPC Builder 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、SOPC Builder での UART ペリフェラル `uart1` は、`system.h` では `/dev/uart1` となります。

カレント・ディレクトリという概念はありません。すべてのファイルは、絶対パスを使用してアクセスする必要があります。

以下に、HAL ファイル・システム内のノードとして登録されたリード・オンリ zip ファイル・サブシステム `rozipfs` から、キャラクタを読み取るコードを示します。

例：ファイル・サブシステムからのキャラクタの読み取り

```
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>

#define BUF_SIZE (10)

int main(void)
{
    FILE* fp;
    char buffer[BUF_SIZE];

    fp = fopen ("/mount/rozipfs/test", "r");
    if (fp == NULL)
    {
        printf ("Cannot open file.\n");
        exit (1);
    }

    fread (buffer, BUF_SIZE, 1, fp);

    fclose (fp);

    return 0;
}
```



これらの関数の使用法の詳細については、10-1 ページの「[HAL API リファレンス](#)」を参照してください。

キャラクタ・モード・デバイスの使用

キャラクタ・モード・デバイスとは、UART (Universal Asynchronous Receiver/Transmitter) のように、キャラクタをシリアルに送受信するハードウェア・ペリフェラルです。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されます。一般に、プログラムはファイル・ディスクリプタをデバイスの名前に関連付けた後に、`file.h` に定義された ANSI C ファイル操作を使用して、キャラクタをファイルから読み込んだり、ファイルに書き込んだりします。さらに、HAL では標準入力、標準出力、および標準エラーのコンセプトもサポートされているため、プログラムから `stdio.h` I/O 関数を呼び出すことができます。

標準入力、標準出力および標準エラー

シンプルなコンソール I/O を実装するには、標準入力 (stdin)、標準出力 (stdout)、および標準エラー (stderr) を使用するのが最も簡単な方法です。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および受信することが可能になります。例えば、システム・ライブラリは、printf() の出力は標準出力に、perror() は標準エラーに送ります。

各チャンネルは、Nios II IDE でシステム・ライブラリ・プロパティを設定することによって、特定のハードウェア・デバイスに関連付けます。



詳細については、Nios II IDE オンライン・ヘルプを参照してください。

以下のコードは、定番の Hello World プログラムを示します。このプログラムでは、Nios II IDE でコンパイルしたときに stdout に関連付けられる任意のデバイスへ、キャラクタを送信します。

例：Hello World

```
#include <stdio.h>
int main ()
{
    printf ("Hello world!");
    return 0;
}
```

UNIX 形式の API を使用する場合は、unistd.h で定義されたファイル・ディスクリプタ STDIN_FILENO、STDOUT_FILENO、および STDERR_FILENO を使用すれば、stdin、stdout、stderr にそれぞれアクセスできます。

キャラクタ・モード・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、または 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と同様に簡単です。以下に、uart1 という名前の UART にメッセージを書き込むコードを示します。

例：UART へのキャラクタの書き込み

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

int main (void)
{
    char* msg = "hello world";
    FILE* fp;
```

```

fp = fopen ("/dev/uart1", "w");
if (fp)
{
    fprintf(fp, "%s",msg);
    fclose (fp);
}
return 0;
}

```

C++ ストリーム

HAL ベースのシステムでは、C++ からのファイル操作に C++ ストリーム API が使用できます。

/dev/null

デバイス /dev/null は、すべてのシステムに含まれています。/dev/null に書き込んでも処理されず、データは破棄されます。/dev/null は、システム起動中に、安全な I/O リダイレクションを実現するために使用されます。また、このデバイスは、不適切なデータを出さないようにするアプリケーションにも役立ちます。

このデバイスは、完全にソフトウェアのみで構成されています。システム内の物理的なハードウェア・デバイスとは無関係です。

ファイル・サブシステムの使用

ファイル・サブシステム用の HAL 汎用デバイス・モデルを利用すれば、C 標準ライブラリのファイル I/O 関数を使用して、関連付けられるメディアに格納されたデータにアクセスできます。例えば、アルテラの zip リード・オンリ・ファイル・システムを利用すると、フラッシュ・メモリに格納されたファイル・システムにリード・オンリでアクセスできます。

ファイル・サブシステムの役割は、所定のマウント・ポイントにおけるすべてのファイル I/O アクセスを管理することです。例えば、ファイル・サブシステムがマウント・ポイント /mnt/rozipfs に登録されている場合、`fopen("/mnt/rozipfs/myfile", "r")` など、このディレクトリにおけるすべてのファイル・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。

キャラクタ・モード・デバイスと同様に、ファイル・サブシステム内のファイルは、`fopen()` や `fread()` など、`file.h` で定義された C のファイル I/O 関数を使用して操作できます。これらの関数の使用法の詳細については、[10-1 ページの「HAL API リファレンス」](#)を参照してください。

タイマ・デバイスの使用

タイマ・デバイスとは、クロックをカウントし、周期的な割り込み要求を生成できるハードウェア・ペリフェラルです。タイマ・デバイスを使用すると、HAL システム・クロック、アラーム、時刻、時間測定など、時間に関連する多数の機能を実現できます。タイマ機能を使用するには、Nios II プロセッサ・システムのハードウェアにタイマ・ペリフェラルが含まれていることが必要です。

HAL API は、2 種類のタイマ・デバイス・ドライバを提供します。1 つは、アラーム機能を可能にするシステム・クロック・ドライバ、もう 1 つは、高精度の時間測定を可能にするタイムスタンプ・ドライバです。特定のタイマ・ペリフェラルは、どちらか一方のみ動作できますが、両方同時には動作できません。

HAL では、標準 UNIX 関数の `gettimeofday()`、`settimeofday()`、および `times()` が実装されています。



タイマ・デバイスにアクセスするための HAL 特有の API 関数は、`sys/alt_alarm.h` および `sys/alt_timestamp.h` で定義されています。



これらの関数の使用法の詳細については、10-1 ページの「HAL API リファレンス」を参照してください。

HAL システム・クロック

HAL システム・クロック・ドライバは、周期的な「ハートビート」を供給して、各ビートごとにシステム・クロックを増加させます。システム・クロック機能を使用すると、指定した時間に関数を実行したり、タイミング情報を取得したりすることができます。特定のハードウェア・タイマ・ペリフェラルをシステム・クロック・デバイスとして関連付けるには、Nios II IDE でシステム・ライブラリのプロパティを設定します。



詳細については、Nios II IDE オンライン・ヘルプを参照してください。

システム・クロックは、「チック」の単位で時間を測定します。ハードウェアとソフトウェアの両方を扱うエンベデッド・エンジニアは、HAL システム・クロックと、Nios II プロセッサ・ハードウェアの同期化に使用されるクロック信号を混同しないように注意してください。HAL システム・クロック・チックの周期は、ハードウェア・システム・クロックよりもはるかに長くなります。

システム・クロックの現在の値は、`alt_nticks()` 関数を呼び出すと取得できます。この関数は、リセット以降の経過時間をシステム・クロック・チック単位で返します。システム・クロック・レート (チック / 秒) は、関数 `alt_ticks_per_second()` を使用すれば取得できます。HAL タイマ・ドライバは、システム・クロックのインスタンスを作成したときに、チック周波数を初期化します。

標準 UNIX 関数 `gettimeofday()` を利用すれば、現在の時間を取得できます。まず、`settimeofday()` を呼び出して、時刻をキャリブレートすることが必要です。さらに、`times()` 関数を使用して、経過したチック数に関する情報を取得することもできます。これらの関数は、`times.h` で定義されています。

アラーム

HAL アラーム機能を使用して、指定した時間に実行する関数が登録できます。アラームは、関数 `alt_alarm_start()` を呼び出すと、登録されます。

```
int alt_alarm_start (alt_alarm* alarm,
                    alt_u32   nticks,
                    alt_u32   (*callback) (void* context),
                    void*     context);
```

関数 `callback` は、`nticks` が経過した後に呼び出されます。入力引数 `context` は、呼び出しが発生したときに、入力引数として `callback` に渡されます。入力引数 `alarm` が示す構造体は、`alt_alarm_start()` への呼び出しによって初期化されます。この構造体を初期化する必要はありません。

このコールバック関数はアラームをリセットできます。登録したコールバック関数の戻り値は、次の `callback` へのコールまでに経過するチック数です。戻り値がゼロであれば、アラームの停止が必要であることを意味します。`alt_alarm_stop()` を呼び出すと、アラームを手動でキャンセルできます。

アラーム・コールバック関数の記述には注意が必要です。これらの関数は、多くの場合、割り込み処理の中で実行され、機能に特定の制約が課されます (6-1 ページの「例外処理」を参照してください)。

以下に、1 秒ごとに周期的なコールバックを行うように、アラームを登録する方法を示すコードの一部分を示します。

例：周期的なアラーム・コールバック関数の使用

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

/*
 * コールバック関数
 */

alt_u32 my_alarm_callback (void* context)
{
    /* この関数は、1 秒ごとに呼び出されます。 */
    return alt_ticks_per_second();
}

...

/* alt_alarm は、アラームの期間存続する必要があります。 */
static alt_alarm alarm;

...

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

高精度時間測定

場合によっては、HAL システム・クロック・チックで得られるレベルを上回る精度で、時間間隔の測定が必要になることも考えられます。HAL は、タイムスタンプ・ドライバを使用する高精度タイミング関数を提供しています。タイムスタンプ・ドライバは、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、タイミング情報を取得できます。HAL はシステム内に 1 つのタイムスタンプ・ドライバのみサポートします。

タイムスタンプ・ドライバが存在すれば、関数 `alt_timestamp_start()` および `alt_timestamp()` が利用可能になります。アルテラが提供するタイムスタンプ・ドライバは、Nios II IDE のシステム・ライブラリ・プロパティ・ページで、ユーザが選択したタイマを使用します。

関数 `alt_timestamp_start()` を呼び出すと、カウンタが動作を開始します。続いて `alt_timestamp()` を呼び出すと、タイムスタンプ・カウンタの現在の値が返されます。再び `alt_timestamp_start()` を呼び出せば、カウンタはゼロにリセットされます。カウンタが $(2^{32}-1)$ に達したときのタイムスタンプ・ドライバの動作は定義されていません。

関数 `alt_timestamp_freq()` を呼び出すと、タイムスタンプ・カウンタが増加するレートを取得できます。一般にこのレートは、Nios II プロセッサ・システムが動作するハードウェア周波数（通常、毎秒数百万サイクル）です。タイムスタンプ・ドライバは、`alt_timestamp.h` ヘッダ・ファイルで定義されます。

以下のコードは、タイムスタンプ機能を使用して、コード実行時間を測定する方法の一部分です。

例：コード実行時間を測定するためのタイムスタンプの使用

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

int main (void)
{
    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(); /* 最初にモニタする関数 */
        time2 = alt_timestamp();
        func1(); /* 2 番目にモニタする関数 */
        time3 = alt_timestamp();

        printf ("time in func1 = %u ticks\n",
            (unsigned int) (time2 - time1));
        printf ("time in func2 = %u ticks\n",
            (unsigned int) (time3 - time2));
        printf ("Number of ticks per second = %u\n",
            (unsigned int)alt_timestamp_freq());
    }
    return 0;
}
```

フラッシュ・デバイスの使用

HAL は、不揮発性フラッシュ・メモリ・デバイス用の汎用デバイス・モデルを提供します。フラッシュ・メモリは、専用のプログラミング・プロトコルを使用して、データを格納します。HAL API は、データをフラッシュに書き込むための関数を提供します。例えば、これらの関数を使用して、フラッシュ・ベースのファイル・サブシステムを実装できます。

また、HAL API は、フラッシュを読み取るための関数を提供します。ただし、一般にこの機能は不要です。大部分のフラッシュ・デバイスでは、プログラムは読み取りの際にフラッシュ・メモリ空間をシンプルなメモリとして扱うことができ、専用の HAL API 関数を呼び出す必要はありません。フラッシュ・デバイスに、アルテラ EPCS シリアル・コンフィギュレーション・デバイスなど、データ読み取り専用プロトコルが用意されている場合、データの読み取りと書き込みのどちらも、HAL API を使用して実行する必要があります。

このセクションでは、フラッシュ・デバイス・モデル用の HAL API について説明します。以下の 2 つの API は、異なるレベルでフラッシュへのアクセスを可能にします。

- シンプル・フラッシュ・アクセス - バッファをフラッシュに書き込み、フラッシュからその内容を読み取るためのシンプルな API です。他のフラッシュ消去ブロックの以前の内容は保持されません。
- 高精度フラッシュ・アクセス - 個々のブロックへの書き込みまたはブロックの消去において、制御を必要とするプログラム用の高精度な関数です。一般に、この機能はファイル・サブシステムの管理に必要です。

フラッシュ・デバイスにアクセスするための API 関数は、`sys/alt_flash.h` で定義されています。



これらの関数の使用法の詳細については、[10-1 ページの「HAL API リファレンス」](#)を参照してください。

シンプル・フラッシュ・アクセス

このインタフェースは、`alt_flash_open_dev()`、`alt_write_flash()`、`alt_read_flash()`、および `alt_flash_close_dev()` で構成されています。

4-16 ページの「例：シンプル・フラッシュ API 関数の使用」のコードは、これらすべての関数の使い方を、1 つのコード例で示しています。`alt_flash_open_dev()` を呼び出してフラッシュ・デバイスをオープンすると、フラッシュ・デバイスへのファイル・ハンドルが返されます。この関数は、`system.h` で定義されているとおり、唯一の引数としてフラッシュ・デバイスの名前を受け取ります。

ハンドルを取得すれば、`alt_write_flash()` 関数を使用して、フラッシュ・デバイスにデータを書き込むことができます。プロトタイプは以下のとおりです。

```
int alt_write_flash(alt_flash_fd* fd,
                   int          offset,
                   const void*  src_addr,
                   int          length )
```

この関数を呼び出すと、ハンドル `fd` で識別されるフラッシュ・デバイスに、先頭から `offset` バイトの書き込みが実行されます。書き込むデータは、`src_addr` で指定されるアドレスから送られ、そのデータ量は `length` です。

また、フラッシュ・デバイスからのデータの読み取りには、`alt_read_flash()` 関数も利用できます。プロトタイプは以下のとおりです。

```
int alt_read_flash( alt_flash_fd* fd,
                   int          offset,
                   void*        dest_addr,
                   int          length )
```

この関数を呼び出すと、ハンドル `fd` を持つフラッシュ・デバイスの先頭から `offset` バイトが読み取られます。データは、`dest_addr` で示される位置に書き込まれ、データ量は `length` です。大部分のフラッシュ・デバイスでは、標準メモリとしてメモリ内容にアクセスできるため、`alt_read_flash()` を使用する必要はありません。

関数 `alt_flash_close_dev()` は、ファイル・ハンドルを受け取ってデバイスをクローズします。この関数のプロトタイプは以下のとおりです。

```
void alt_flash_close_dev(alt_flash_fd* fd )
```

以下のコードは、`system.h` で定義された `/dev/ext_flash` という名前のフラッシュ・デバイスに、シンプル・フラッシュ API 関数を使用してアクセスする方法を示します。

例：シンプル・フラッシュ API 関数の使用

```
#include <stdio.h>
#include <string.h>
#include "sys/alt_flash.h"
#define BUF_SIZE1024

int main ()
{
    alt_flash_fd* fd;
    int          ret_code;
    char         source[BUF_SIZE];
    char         dest[BUF_SIZE];

    /* ソース・バッファをすべて 0xAA に初期化 */
    memset(source, 0xa, BUF_SIZE);

    fd = alt_flash_open_dev("/dev/ext_flash");
    if (fd)
    {
        ret_code = alt_write_flash(fd, 0, source, BUF_SIZE);
        if (!ret_code)
        {
            ret_code = alt_read_flash(fd, 0, dest, BUF_SIZE);
            if (!ret_code)
            {
                /*
                 * 成功
                 * この時点で、フラッシュはすべて 0xa となり、
                 * フラッシュの内容がすべて dest に読み戻されているはず。
                 */
            }
        }
        alt_flash_close_dev(fd);
    }
    else
    {
        printf("Can't open flash device\n");
    }
    return 0;
}
```

ブロックの消去または破壊

一般に、フラッシュ・メモリは複数のブロックに分割されます。ブロックにデータを書き込む前に `alt_write_flash()` で、ブロックの内容を消去しなければならないことがあります。この場合、ブロックの既存の内容は保存されません。ブロック境界をまたいで書き込みを行う場合も、この動作によって予期しないデータ破壊（消去）が発生することがあります。現在のフラッシュ・メモリの内容を保存する場合は、より高精度なフラッシュ関数を使用します。4-18 ページの「高精度フラッシュ・アクセス」を参照してください。

表 4-3 は、シンプルなフラッシュ・アクセス関数を使用した書き込みによって、予期しないデータ破壊がどのようにして発生するかを示します。表 4-3 は、2 つの 4K バイト・ブロックで構成される 8K バイトのフラッシュ・メモリの例を示します。最初に、すべて `0xAA` からなる 5 K バイトを、フラッシュ・メモリのアドレス `0x0000` に書き込み、次にすべて `0xBB` からなる 2 K バイトをアドレス `0x1400` に書き込みます。最初の書き込みが成功すると（時刻 $t(2)$ ）、フラッシュ・メモリには `0xAA` が 5 K バイト格納され、それ以外は空（つまり、`0xFF`）になります。次に、2 回目の書き込みが開始されますが、2 番目のブロックに書き込む前に、このブロックが消去されます。この時点（ $t(3)$ ）で、フラッシュ・メモリには、`0xAA` が 4 K バイト、`0xFF` が 4 K バイト格納されています。2 回目の書き込みが終了すると（ $t(4)$ ）、アドレス `0x1000` にある 2 K バイトの `0xFF` が予期せず破壊されます。

表 4-3. フラッシュへの書き込みと予期しないデータ破壊発生例

アドレス	ブロック	時刻 $t(0)$	時刻 $t(1)$	時刻 $t(2)$	時刻 $t(3)$	時刻 $t(4)$
		1 回目の書き込み前	1 回目の書き込み		2 回目の書き込み	
			ブロックの消去後	データ 1 の書き込み後	ブロックの消去後	データ 2 の書き込み後
<code>0x0000</code>	1	??	FF	AA	AA	AA
<code>0x0400</code>	1	??	FF	AA	AA	AA
<code>0x0800</code>	1	??	FF	AA	AA	AA
<code>0x0C00</code>	1	??	FF	AA	AA	AA
<code>0x1000</code>	2	??	FF	AA	FF	FF (1)
<code>0x1400</code>	2	??	FF	FF	FF	BB
<code>0x1800</code>	2	??	FF	FF	FF	BB
<code>0x1C00</code>	2	??	FF	FF	FF	FF

表 4-3 の注：

(1) 2 回目の書き込みのための消去中に、予期せずクリアされて FF に変化した。

高精度フラッシュ・アクセス

その他にも、フラッシュへの書き込みを最高レベルの精度で完全に制御する3つの関数

```
alt_get_flash_info()
alt_erase_flash_block()
alt_write_flash_block()
```

があります。

フラッシュ・メモリの性質上、あるブロック内の1アドレスだけを消去することはできません。一度にブロック全体を消去（つまり、すべて1に設定）する必要があります。フラッシュ・メモリへの書き込みでは、ビットが1から0に変化するだけで、どのビットでも0から1に変更するには、そのビットが含まれるブロック全体を消去する必要があります。したがって、ブロック内の特定の位置のみ変更し、周囲の内容が変化しないようにするには、ブロック全体の内容をバッファに読み出し、バッファ内で値を変更し、フラッシュ・ブロックを消去して、最後にブロック・サイズのバッファ全体をフラッシュ・メモリに書き戻すことが必要です。高精度フラッシュ・アクセス関数を利用すれば、このプロセスをフラッシュ・ブロック・レベルで実行できます。

`alt_get_flash_info()` は、消去領域の数、各領域内の消去ブロック数、および各消去ブロックのサイズを取得します。プロトタイプは以下のとおりです。

```
int alt_get_flash_info( alt_flash_fd* fd,
                       flash_region** info,
                       int*          number_of_regions)
```

呼び出しが成功した場合、関数が返された時点で、`number_of_regions` が示すアドレスには、フラッシュ・メモリ内の消去領域の数が格納されており、`info` は1番目の `flash_region` で記述されるアドレスを示しています。

`flash_region` 構造体は `sys/alt_flash_types.h` で定義され、その `typedef` は以下のとおりです。

```
typedef struct flash_region
{
    int  offset; /* フラッシュの開始位置からこの領域までのオフセット */
    int  region_size; /* この消去領域のサイズ */
    int  number_of_blocks; /* この領域のブロック数 */
    int  block_size; /* この消去領域内の各ブロックのサイズ */
} flash_region;
```

`alt_get_flash_info()` を呼び出して情報を取得すると、フラッシュのブロックを個別に消去またはプログラムできます。

`alt_erase_flash()` は、フラッシュ・メモリ内の単一のブロックを消去します。プロトタイプは以下のとおりです。


```
int alt_erase_flash_block( alt_flash_fd* fd,
                          int           offset,
                          int           length)
```

フラッシュ・メモリは、ハンドル `fd` で識別されます。ブロックは、フラッシュ・メモリの先頭からの `offset` バイトとして認識され、ブロック・サイズは `length` に渡されます。

`alt_write_flash_block()` は、フラッシュ・メモリ内の 1 ブロックに書き込みを実行します。プロトタイプは以下のとおりです。

```
int alt_write_flash_block( alt_flash_fd* fd,
                           int           block_offset,
                           int           data_offset,
                           const void   *data,
                           int           length)
```

この関数は、ハンドル `fd` で識別されるフラッシュ・メモリに書き込みを実行します。フラッシュの先頭から `block_offset` バイトの位置にあるブロックに書き込みます。この関数は、`data` で示された位置から `length` バイトのデータを、フラッシュ・デバイスの先頭から `data_offset` バイトの位置に書き込みます。

 これらのプログラムおよび消去関数では、アドレス・チェックは実行されず、また書き込み操作の範囲が隣のブロックに及ぶかどうかも検証されません。プログラムまたは消去するブロックについて、適切な情報を渡すことが必要です。

以下のコードは、高精度フラッシュ・アクセス関数の使用法を示します。

例：高精度フラッシュ・アクセス API 関数の使用

```
#include <string.h>
#include "sys/alt_flash.h"
#define BUF_SIZE 100

int main (void)
{
    flash_region* regions;
    alt_flash_fd* fd;
    int           number_of_regions;
    int           ret_code;
    char          write_data[BUF_SIZE];
```

```
/* write_data をすべて 0xa に設定 */
memset(write_data, 0xA, BUF_SIZE);

fd = alt_flash_open_dev(EXT_FLASH_NAME);

if (fd)
{
    ret_code = alt_get_flash_info(fd,
                                  &regions,
                                  &number_of_regions);

    if (number_of_regions && (regions->offset == 0))
    {
        /* 1 番目のブロックを消去 */
        ret_code = alt_erase_flash_block(fd,
                                          regions->offset,
                                          regions->block_size);

        if (ret_code)
        {
            /*
             * write_data から BUF_SIZE バイト、つまり 100 バイトを
             * フラッシュの 1 番目のブロックに書き込みます
             */
            ret_code = alt_write_flash_block( fd,
                                              regions->offset,
                                              regions->offset+0x100,
                                              write_data,
                                              BUF_SIZE);
        }
    }
}
return 0;
}
```

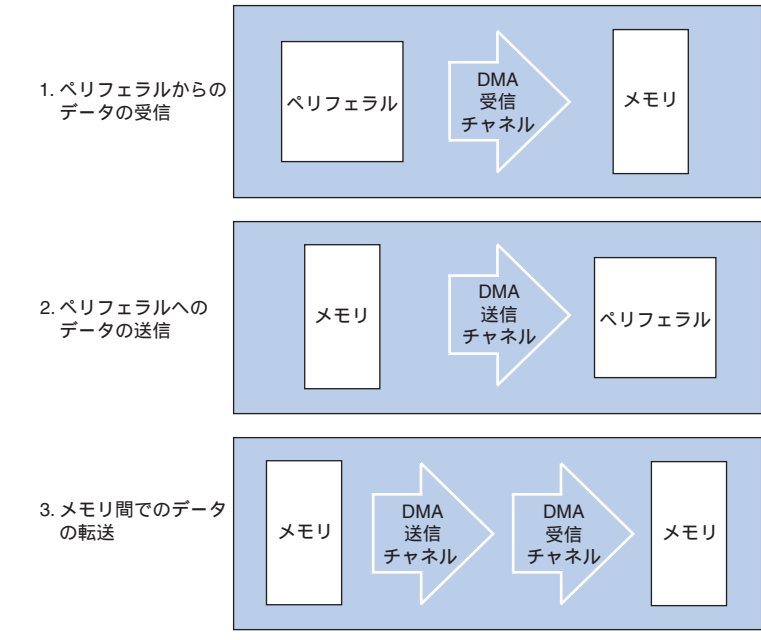
DMA デバイスの使用

HAL は、DMA (Direct Memory Access) デバイスに対応するデバイス抽象化モデルを提供します。これらのモデルは、データ・ソースからディスティネーションへのバルク・データ転送を実行するペリフェラルです。イーサネット接続など、メモリやその他のデバイスをソースおよびディスティネーションにすることができます。

HAL DMA デバイス・モデルにおいて、DMA 転送は、2 つのカテゴリ、つまり送信と受信のいずれかに分類されます。したがって、HAL は送信チャンネルと受信チャンネルを実装するために、2 つのデバイス・ドライバを提供しています。送信チャンネルは、ソース・バッファにデータを受け取り、それをディスティネーション・デバイスに送信します。受信チャンネルは、デバイスからデータを受信し、ディスティネーション・バッファに格納します。基本ハードウェアの実装状態に応じて、ソフトウェアでは、これら 2 つのエンドポイントの一方のみにアクセスすることもできます。

図 4-2 に、DMA 転送の 3 つの基本形式を示します。メモリ間でデータをコピーする場合、受信 DMA チャンネルと送信 DMA チャンネルを同時に使用します。

図 4-2. DMA 転送の 3 つの基本形式



DMA デバイスにアクセスするための API 関数は、`sys/alt_dma.h` で定義されています。



これらの関数の使用法の詳細については、10-1 ページの「HAL API リファレンス」を参照してください。

DMA デバイスは、物理メモリの内容を操作するため、データの読み取りおよび書き込みを行う際には、キャッシュの相互作用を考慮する必要があります。7-1 ページの「キャッシュ・メモリ」を参照してください。

DMA 送信チャンネル

DMA 送信要求は、DMA 送信デバイスのハンドルを使用して、キューに格納されます。ハンドルは関数 `alt_dma_txchan_open()` を使用して取得します。この関数は、`system.h` で定義されているように、1 つの引数（使用するデバイスの名前）のみ受け取ります。

以下のコードは、DMA 送信デバイス `dma_0` のハンドルを取得する方法を示します。

例：DMA デバイスのファイル・ハンドルの取得

```
#include <stddef.h>
#include "sys/alt_dma.h"

int main (void)
{
    alt_dma_txchan tx;

    tx = alt_dma_txchan_open ("/dev/dma_0");
    if (tx == NULL)
    {
        /* エラー */
    }
    else
    {
        /* 成功 */
    }
    return 0;
}
```

このハンドルを使用すると、`alt_dma_txchan_send()` によって送信要求を送信できます。プロトタイプは以下のとおりです。

```
typedef void (alt_txchan_done)(void* handle);

int alt_dma_txchan_send (alt_dma_txchan dma,
                        const void* from,
                        alt_u32 length,
                        alt_txchan_done* done,
                        void* handle);
```

`alt_dma_txchan_send()` を呼び出すと、チャンネル `dma` に送信要求が送信され、`length` バイトのデータがアドレス `from` から送信されます。この関数は、すべての DMA 転送が完了する前に呼び出し元に復帰します。戻り値は、要求が正常にキューに格納されたかどうかを示します。負の戻り値は、要求が失敗したことを示します。転送が完了すると、通知を行うために引数 `handle` を渡して、ユーザ提供の関数 `done` が呼び出されます。

DMA 送信チャンネルを操作するために、さらに 2 つの関数

`alt_dma_txchan_space()`、`alt_dma_txchan_ioctl()` が提供されています。`alt_dma_txchan_space()` 関数は、デバイスのキューに格納できる追加送信要求数を返します。`alt_dma_txchan_ioctl()` 関数は、送信デバイスのデバイス固有操作を実行します。

DMA 受信チャンネル

DMA 受信チャンネルは、DMA 送信チャンネルと同様の方式で動作します。DMA 受信チャンネルのハンドルは、`alt_dma_rxchan_open()` 関数を使用して取得できます。ハンドルを取得すると、`alt_dma_rxchan_prepare()` 関数を使用して、受信要求を送信できます。`alt_dma_rxchan_prepare()` のプロトタイプは以下のとおりです。

```
typedef void (alt_rxchan_done)(void* handle, void* data);

int alt_dma_rxchan_prepare (alt_dma_rxchan  dma,
                           void*          data,
                           alt_u32         length,
                           alt_rxchan_done* done,
                           void*          handle);
```

この関数を呼び出すと、受信要求がチャンネル `dma` に送信され、最大で `length` バイトのデータがアドレス `data` に置かれます。この関数は、DMA 転送が完了する前に呼び出し元に復帰します。戻り値は、要求が正常にキューに格納されたかどうかを示します。負の戻り値は、要求が失敗したことを示します。転送が完了すると、通知を行うための引数 `handle`、およびデータを受け取るためのポインタを渡して、ユーザが提供した関数 `done` が呼び出されます。

DMA 受信チャンネルを操作するために、さらに 2 つの関数

`alt_dma_rxchan_depth()`、`alt_dma_rxchan_ioctl()` が用意されています。

`alt_dma_rxchan_depth()` 関数は、デバイスのキューに格納できる最大受信要求数を返します。`alt_dma_rxchan_ioctl()` 関数は、受信デバイスのデバイス固有操作を実行します。

以下のコードは、DMA 受信要求を送信し、転送が完了するまで `main()` で待機するアプリケーション例を示します。

例：受信チャンネルでの DMA 転送

```
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include "sys/alt_dma.h"
#include "alt_types.h"

/* 転送の完了を示すために使用するフラグ */
volatile int dma_complete = 0;

/* 転送が完了したときに呼び出される関数 */
void dma_done (void* handle, void* data)
```

```
{
    dma_complete = 1;
}

int main (void)
{
    alt_u8 buffer[1024];
    alt_dma_rxchan rx;

    /* デバイスのハンドルを取得 */
    if ((rx = alt_dma_rxchan_open ("/dev/dma_0")) == NULL)
    {
        printf ("Error:failed to open device\n");
        exit (1);
    }
    else
    {
        /* 受信要求を送信 */
        if (alt_dma_rxchan_prepare (rx, buffer, 1024, dma_done, NULL)
            < 0)
        {
            printf ("Error:failed to post receive request\n");
            exit (1);
        }

        /* 転送が完了するまで待機 */
        while (!dma_complete);
        printf ("Transaction complete\n");
        alt_dma_rxchan_close (rx);
    }
    return 0;
}
```

メモリ間 DMA 転送

メモリ・バッファの間でデータをコピーするには、受信 DMA ドライバと送信 DMA ドライバの両方が必要です。以下のコードは、受信要求に続いて送信要求をキューに格納して、メモリ間 DMA 転送を実現するプロセスを示します。

例：メモリ間でのデータのコピー

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

#include "sys/alt_dma.h"
#include "system.h"

static volatile int rx_done = 0;

/*
 * データ受信の完了を示す通知を取得する
 * コールバック関数
 */
```

```
static void done (void* handle, void* data)
{
    rx_done++;
}

/*
 *
 */

int main (int argc, char* argv[], char* envp[])
{
    int rc;

    alt_dma_txchan txchan;
    alt_dma_rxchan rxchan;

    void* tx_data = (void*) 0x901000; /* 送信するデータを示すポイン
    タ */
    void* rx_buffer = (void*) 0x902000; /* rx バッファを示すポインタ */

    /* 転送チャンネルを作成 */

    if ((txchan = alt_dma_txchan_open("/dev/dma_0")) == NULL)
    {
        printf ("Failed to open transmit channel\n");
        exit (1);
    }

    /* 受信チャンネルを作成 */

    if ((rxchan = alt_dma_rxchan_open("/dev/dma_0")) == NULL)
    {
        printf ("Failed to open receive channel\n");
        exit (1);
    }

    /* 送信要求を送信 */

    if ((rc = alt_dma_txchan_send (txchan,
                                   tx_data,
                                   128,
                                   NULL,
                                   NULL)) < 0)
    {
        printf ("Failed to post transmit request, reason = %i\n", rc);
        exit (1);
    }

    /* 受信要求を送信 */

    if ((rc = alt_dma_rxchan_prepare (rxchan,
                                       rx_buffer,
                                       128,
                                       done,
                                       NULL)) < 0)
    {
```

```
        printf ("Failed to post read request, reason = %i\n", rc);
        exit (1);
    }

    /* 転送が完了するまで待機 */

    while (!rx_done);

    printf ("Transfer successful!\n");

    return 0;
}
```

コード・フットプリントの削減

コードを格納するメモリ・デバイスにはコストがかかるため、コード・サイズは常にシステム開発者の悩みです。コード・サイズのコントロールと削減は、このコストを管理する上で重要です。

HAL 環境は、一般にユーザが要求する機能のみが全体的なコード・フットプリントに関与するように設計されています。ユーザの Nios II ハードウェア・システムに、プログラムで使用するだけのペリフェラルしかない場合、HAL はそのハードウェアを制御するのに必要なドライバしか持つ必要はありません。

以下のセクションでは、コード・サイズを絶対最小サイズに削減する必要がある場合に検討すべきオプションについて説明します。

コンパイラ最適化の有効化

`nios2-elf-gcc` コンパイラに `-O3 compiler` 最適化レベルを使用します。すると、コードはサイズと速度の両方が最大限に最適化されてコンパイルされます。これは、システム・ライブラリとアプリケーション・プロジェクトの両方に対して行う必要があります。

スモール・フットプリント・デバイス・ドライバの使用

いくつかのデバイスでは、フル機能の「高速」型、および軽量の「スモール」型の 2 つの改良型ドライバが用意されています。これら 2 つの改良型によって、どの機能が利用できるかは、デバイスによって異なります。デフォルトで、HAL システム・ライブラリは常に高速型ドライバを使用します。Nios II IDE で HAL システム・ライブラリの `Use Small Footprint Drivers` オプションをオンにすると、スモール・フットプリント・ドライバを選択できます。HAL システム・ライブラリを構築するときには、プリプロセッサ・オプション `-DAL_T_USE_SMALL_DRIVERS` が利用できません。

表 4-4 に、スモール・フットプリント・ドライバを提供するアルテラ製 Nios II ペリフェラルを示します。また、その他のペリフェラルもスモール・フットプリント・オプションに影響されることがあります。スモール・フットプリント・ドライバの動作に関する詳細は、各ペリフェラルのデータシートを参照してください。

ペリフェラル	スモール・フットプリント動作
UART	IRQ 駆動ではなく、ポーリングによって動作します。
JTAG UART	IRQ 駆動ではなく、ポーリングによって動作します。
共通フラッシュ・インタフェース・コントローラ	ドライバは、スモール・フットプリント・モードでは除外されます。
LCD モジュール・コントローラ	ドライバは、スモール・フットプリント・モードでは除外されます。

ファイル・ディスクリプタ・プールの削減

キャラクタ・モード・デバイスおよびファイルにアクセスするファイル・ディスクリプタは、利用可能なファイル・ディスクリプタのプールから割り当てられます。このプールのサイズは、コンパイル時の定数 `ALT_MAX_FD` によって定義され、Nios II IDE のシステム・ライブラリ・プロパティとして管理できます。デフォルトは 32 です。例えば、プログラムで 10 のみ必要であれば、`ALT_MAX_FD` の値を小さくすることによって、メモリ・フットプリントを削減できます。

/dev/null の使用

ブート時において、標準入力、標準出力、および標準エラーは、すべて `null` デバイス、つまり `/dev/null` に送られます。これによって、ドライバの初期化中に `printf()` を呼び出しても何も実行されず、動作に影響を与えません。すべてのドライバがインストールされると、これらのストリームは、HAL で構成されたチャンネルにリダイレクトされます。このリダイレクトを実行するコードのフットプリントは小さなものですが、`stdin`、`stdout`、および `stderr` に対して `null` を選択すれば、リダイレクトを完全に回避できます。このように選択するには、標準出力または標準エラーに対して送信されたすべてのデータを破棄し、プログラムで `stdin` を介した入力を受信しないことが前提となります。`stdin`、`stdout`、および `stderr` チャンネルは、Nios II IDE ではシステム・ライブラリ・プロパティとして制御できます。

ANSI C ではなく UNIX ファイル I/O の使用

UNIX 形式のファイル I/O 関数を使用してデバイスやファイルにアクセスすると、アクセスごとにそれに付随するパフォーマンス・オーバーヘッドが発生します。パフォーマンスを改善するために、ANSI C ファイル I/O を使用することでバッファ使用のアクセスが可能になり、実行されるハードウェア I/O アクセスの総数が少なくなります。また、ANSI C API は柔軟性が高く、使いやすくなります。しかし、これらの利点は、コード・フットプリントを犠牲にして実現されています。UNIX 形式の I/O API を直接使用することで、コード・フットプリントを削減することが可能です。4-5 ページの「UNIX 形式のインタフェース」を参照してください。

縮小版 Newlib C ライブラリの使用

多くの場合、エンベデッド・システムには完全な ANSI C 標準ライブラリは不要です。HAL は、一般にエンベデッド・システムでは不要な newlib の機能を取り除くために、newlib ANSI C 標準ライブラリの縮小版を提供します。縮小版の newlib 実装には、小さなコード・フットプリントが必要です。この newlib 実装は、Nios II IDE ではシステム・ライブラリ・プロパティとして制御できます。また、このオプションは、nios2-elf-gcc の `-msmallc` コマンドライン・オプションでも制御できます。



縮小版の newlib C ライブラリがサポートする関数の詳細については、Nios II 開発キットと共にインストールされた newlib の資料を参照してください。(Windows のスタート・メニューから) **プログラム** > **Altera** > **Nios II Development Kit** > **Nios II Documentation** の順にクリックします。

表 4-5 に、縮小版 newlib C ライブラリ実装の制限事項を要約します。

制限事項	影響する関数
<p><code>printf()</code> ルーチン・ファミリーに対しては浮動小数点はサポートされません。右記の関数は実装されていますが、<code>%f</code> オプションと <code>%g</code> オプションはサポートされていません。</p>	<p><code>asprintf()</code> <code>fiprintf()</code> <code>fprintf()</code> <code>iprintf()</code> <code>printf()</code> <code>siprintf()</code> <code>snprintf()</code> <code>sprintf()</code> <code>vasprintf()</code> <code>vfiprintf()</code> <code>vfprintf()</code> <code>vprintf()</code> <code>vsnprintf()</code> <code>vsprintf()</code></p>
<p><code>scanf()</code> ルーチン・ファミリーはサポートされません。右記の関数は、サポートされていません。</p>	<p><code>fscanf()</code> <code>scanf()</code> <code>sscanf()</code> <code>vfscanf()</code> <code>vscanf()</code> <code>vsscanf()</code></p>
<p>シークはサポートされていません。右記の関数は、サポートされていません。</p>	<p><code>fseek()</code> <code>ftell()</code></p>
<p><code>FILE *</code> のオープン/クローズはサポートされていません。既にオープンされた <code>stdout</code>、<code>stderr</code>、および <code>stdin</code> のみ利用できます。右記の関数は、サポートされていません。</p>	<p><code>fopen()</code> <code>fclose()</code> <code>fdopen()</code> <code>fcloseall()</code> <code>fileno()</code></p>
<p><code>FILE *</code> ルーチン(つまり、すべての <code>stdio.h</code> ルーチン)のバッファ機能はありません。</p>	<p>stdio.h で定義されたすべてのルーチン。</p> <p>これらの関数は、サポートはされていますが、バッファ機能はありません。</p> <p><code>setbuf()</code> および <code>setvbuf()</code> はサポートされていません。</p>
<p>ロケールはサポートされていません。</p>	<p><code>setlocale()</code> <code>localeconv()</code></p>
<p>上記の関数がサポートされていないため、C++ のサポートはありません。</p>	

未使用デバイス・ドライバの除去

ハードウェア・デバイスがシステムに存在する場合、Nios II IDE は、そのデバイスがドライバを必要するものと想定し、それに応じて HAL システム・ライブラリを構成します。適切なドライバを見つけることができれば、HAL はこのドライバのインスタンスを作成します。ユーザ・プログラムで実際にデバイスにアクセスしない場合、デバイス・ドライバの初期化にリソースが不必要に消費されることになります。

デバイスがハードウェアに含まれているが、ユーザ・プログラムを使用しない場合は、デバイスを完全に削除するオプションを検討する必要があります。これによって、コード・フットプリントと FPGA リソースがともに削減されます。ただし、デバイスが存在してもソフトウェアがドライバを必要としない、回避不能なケースもあります。

最も一般的な例は、フラッシュ・メモリです。この場合、ユーザ・プログラムはフラッシュ・メモリへの書き込みアクセスが必要ないことが多いため、フラッシュ・ドライバも不要です。このような場合、オプション `-DALT_NO_CFI_FLASH` をプリプロセッサに指定すれば、HAL がフラッシュ・ドライバをシステム・ライブラリに組み込まないようにすることができます。

独立した環境を使用して、デバイス・ドライバの初期化プロセスをより詳細に制御できます。4-31 ページの「[ブート・シーケンスおよびエントリ・ポイント](#)」を参照してください。

非完全終了に対する `_exit()` の使用

HAL は、システム・シャットダウン時に `exit()` 関数を呼び出して、プログラムからの終了を実現します。`exit()` は C ライブラリの内部 I/O バッファをすべて消去し、`atexit()` に登録された関数を呼び出します。特に、`exit()` は `main()` から戻るときに呼び出されます。

一般に、エンベデッド・システムは終了することがないため、このコードは冗長となります。クリーンな終了の実現に伴うオーバーヘッドを回避するために、ユーザ・プログラムでは、`exit()` 関数の代わりに `_exit()` 関数が使用できます。この関数ではソース・コードの変更は必要ありません。終了動作は、Nios II IDE でのシステム・ライブラリ・プロパティとして、またはプリプロセッサ・オプション `-Dexit=_exit` を指定することによって制御できます。

命令エミュレーションの無効化

HAL ソフトウェア例外ハンドラは、プロセッサが乗算命令および除算命令をサポートしていない場合は、これらの命令をエミュレートできます。この機能は、システム・ライブラリ・プロジェクトの C プリプロセッサ・マクロ `ALT_NO_INSTRUCTION_EMULATION` を定義すれば無効にできます。

使用中のコアがハードウェア乗算 / 除算をサポートしていれば、プロセッサがハードウェア乗算 / 除算をサポートしていない場合でも、ほとんどのケースでこの機能を無効にすることができます。ハードウェア乗算 / 除算命令をサポートしていないシステムに対して構築された、システム・ライブラリ・プロジェクトおよびアプリケーション・プロジェクトは、`-mno-hw-mul` オプションを指定してコンパイルおよびリンクされます。したがって、これらのプロジェクトの一部としてコンパイルされたコードに対しては、乗算命令のエミュレーションは不要です。除算命令エミュレーションは、`-mhw-div` オプションを指定してコードを明示的にコンパイルする場合のみ必要です。

ブート・シーケンスおよびエントリ・ポイント

これまでの説明では、プログラムのエントリ・ポイントを関数 `main()` と仮定していました。それに代わって利用できるエントリ・ポイントとして `alt_main()` があり、これを使用するとブート・シーケンスをより高度に制御できます。`main()` または `alt_main()` からのエントリの概念は、ホスト型アプリケーションと独立型アプリケーションの違いです。

ホスト型アプリケーションと独立型アプリケーション

ANSI C 規格では、ホスト型アプリケーションは、`main()` を呼び出して実行を開始するアプリケーションとして定義されています。`main()` の開始時に、ホスト型アプリケーションは、ランタイム環境およびすべてのシステム・サービスが初期化され、使用できる準備が整っていると仮定しています。HAL システム・ライブラリでも同様に仮定されます。事実、ホスト型環境では、システム内にどのデバイスが存在するか把握したり、各デバイスの初期化方法を考慮する必要がなく、HAL がシステム全体を自動的に初期化するため、Nios II を初めて使用するプログラマにとって、ホスト型環境は HAL の最大の利点の 1 つです。

ANSI C 規格は、自動初期化を回避する代替エントリ・ポイントも用意しており、Nios II プログラマが使用されているハードウェアを手動で初期化することを想定しています。alt_main() 関数は独立型環境を提供し、ユーザはシステムの初期化を完全に制御できます。独立型環境では、プログラマはプログラムで使用されるシステム機能を手動で初期化する必要があります。例えば、独立型環境では、alt_main() が最初にキャラクタ・モード・デバイス・ドライバをインスタンス化し、stdout をデバイスにリダイレクトしない限り、printf() を呼び出しても正しく機能しません。



独立型環境を使用すると、ユーザは HAL の利点を利用できず、システムの初期化をすべて手動で実行しなければならないため、Nios II プログラムの記述は複雑になります。独立型環境を検討する主な目的が、コード・フットプリントの削減の場合は、[4-26 ページの「コード・フットプリントの削減」](#)で説明した推奨事項を考慮してください。HAL システム・ライブラリのフットプリントを削減するには、独立型モードを使用するよりも、Nios II IDE で利用できるオプションを使用する方が簡単です。

Nios II 開発キットには、独立型プログラムとホスト型プログラムの例が用意されています。



詳細については、Nios II IDE オンライン・ヘルプを参照してください。

HAL ベース・プログラムのブート・シーケンス

HAL は、以下のブート・シーケンスを実行するシステム初期化コードを提供します。

- 命令キャッシュおよびデータ・キャッシュを消去する。
- スタック・ポインタをコンフィギュレーションする。
- グローバル・ポインタをコンフィギュレーションする。
- リンカが供給するシンボル __bss_start および __bss_end を使用して、BSS 領域をゼロで初期化する。これらは、BSS 領域の先頭および末尾を示すポインタです。
- システム内にブート・ローダが存在しない場合に、.rwdata、.rodata、および/または例外セクションを RAM にコピーする([4-37 ページの「ブート・モード」](#)を参照)。
- alt_main() を呼び出す。

`alt_main()` 関数を用意していない場合は、デフォルト実装が以下のステップを実行します。

- `ALT_OS_INIT()` を呼び出して、オペレーティング・システム固有の必要な初期化を実行する。OS スケジューラがないシステムの場合、このマクロは無効です。
- HAL がオペレーティング・システムで使用されている場合、HAL ファイル・システムへのアクセスを制御する `alt_fd_list_lock` セマフォ (semaphore) を初期化する。
- 割り込みコントローラを初期化し、割り込みを可能にする。
- `alt_sys_init()` 関数を呼び出して、システム内のすべてのデバイス・ドライバとソフトウェア・コンポーネントを初期化する。Nios II IDE は、各 HAL システム・ライブラリに対応するファイル `alt_sys_init.c` を自動的に作成および管理します。
- 適切なデバイスを使用するために、C 標準 I/O チャネル (`stdin`、`stdout`、および `stderr`) をリダイレクトする。
- `_do_ctors()` 関数を使用して、C++ コンストラクタを呼び出す。
- システム・シャットダウン時に呼び出される C++ デストラクタを登録する。
- `main()` を呼び出す。
- `exit()` を呼び出し、`main()` の戻りコードを `exit()` の入力引数として渡す。

このデフォルト実装は、Nios II 開発キットのインストール・ディレクトリ内のファイル `alt_main.c` に記述されています。

ブート・シーケンスのカスタマイズ

Nios II IDE プロジェクトで `alt_main()` を定義するだけで、スタートアップ・シーケンスの独自の実装を提供できます。これにより、ブート・シーケンスを完全に制御し、HAL サービスを選択することができます。ユーザのアプリケーションで `alt_main()` エントリ・ポイントが必要な場合、デフォルト実装を開始ポイントとしてコピーし、必要に応じてカスタマイズできます。

この関数は呼び出し元に復帰しません。`alt_main()` のプロトタイプは以下のとおりです。

```
void alt_main (void)
```

HAL 構築環境の特徴の 1 つは、すべてのソース・ファイルおよびインクルード・ファイルがサーチ・パスを使用して検索されることです。常にユーザのプロジェクトが最初にチェックされるため、デフォルトのデバイス・ドライバおよびシステム・コードをユーザ自身の実装に置き換えることができます。例えば、`alt_sys_init.c` の代わりにユーザ独自のファイルを提供する場合、そのファイルをユーザのシステム・プロジェクト・ディレクトリに置けばそれが可能です。ユーザが提供したファイルは、自動生成ファイルに優先して使用されます。



`alt_sys_init()` の詳細については、5-1 ページの「HAL 用デバイス・ドライバの開発」を参照してください。

メモリの使用

このセクションでは、HAL がメモリを使用する方法と、HAL がコード、データ、スタックなどをメモリ内で配置する方法について説明します。

メモリ・セクション

デフォルトでは、HAL ベースシステムは、Nios II IDE で作成および管理される自動生成されたリンカ・スクリプトを使用してリンクされます。このリンカ・スクリプトは、利用可能なメモリ・セクション内で、コードおよびデータのマッピングを制御します。自動生成されたリンカ・スクリプトは、システム内の各物理メモリ・デバイスに対するセクションを作成します。例えば、`system.h` ファイルで定義された `on_chip_memory` という名前のメモリ・コンポーネントが存在する場合は、`.on_chip_memory` という名前のメモリ・セクションが存在します。

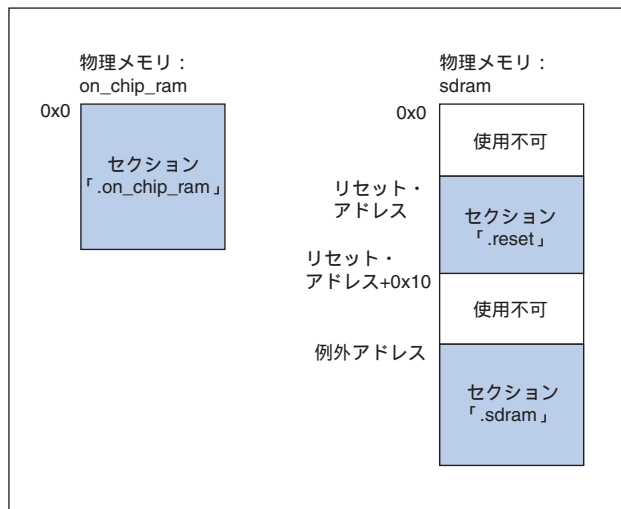
メモリ・デバイスに Nios II プロセッサのリセット・アドレスや例外アドレスが含まれるのは、特殊な場合です。メモリ・デバイスにこれらのアドレスのいずれかが含まれる場合、そのアドレス以下のすべてのメモリは、そのメモリ・デバイスに関連付けられたセクションから除外されます。32 バイトのリセット・セクションはリセット・アドレスを開始位置として構築され、リセット・ハンドラ専用として使用するために予約されます。



このメモリ方式で作成できる使用不可領域は、マルチプロセッサ・システムの他のプロセスが使用できます。

図 4-3 に、物理メモリをメモリ・セクションに分割する方法の一例を示します。あくまでも説明を目的として、この例では、リセット・アドレスおよび例外アドレスの配置によって使用不可となるメモリ領域を人為的に作成しています。デフォルトでは、アルテラのツールは、リセット・アドレスと例外アドレスをメモリにマップするため、アクセス不可のメモリは存在しません。デフォルトのメモリ・マップを使用するシステムでは、リセット・アドレスはデバイス・メモリまたはフラッシュ・メモリのオフセット $0x0$ にあり、例外アドレスはシステム生成時に SOPC Builder に指定されたメモリ内のオフセット $0x20$ にあります。

図 4-3. HAL のメモリ・パーティション



メモリ・パーティションへのコードとデータの割り当て

このセクションでは、特定のメモリ・セクションでプログラム・コードとデータの配置を制御する方法について説明します。通常、Nios II 開発ツールは、合理的なデフォルトのパーティション作成方法を自動的に選択します。例えば、性能を向上させるための一般的な技法は、性能重視のコードとデータをアクセス・タイムが高速なデバイス RAM に配置します。また、デバッグ段階では RAM 内の位置からプロセッサをリセット（つまりブート）し、システムのリリース・バージョンではフラッシュ・メモリからブートすることも一般的です。このような場合は、どのコードがどのセクションに属するかを手動で指定する必要があります。

シンプルな配置オプション

リセット・ハンドラ・コードは常に、`.reset` パーティションに配置されます。例外ハンドラ・コードは常に、例外アドレスを含むセクション内の最初のコードになります。デフォルトでは、その他のコードおよびデータは、以下の3つの出力セクションに分割されます。

- `.text` - その他のすべてのコード
- `.rodata` - 読み取り専用データ
- `.rwdata` - 読み取りおよび書き込みデータ(ゼロに初期化されたデータを含む)

`.text`、`.rodata`、および `.rwdata` の配置は、Nios II IDE でシステム・ライブラリ・プロパティとして制御できます。



詳細については、Nios II IDE オンライン・ヘルプを参照してください。

高度な配置オプション

ユーザのプログラム・ソース・コード内で、特定のコードに対してターゲット・メモリ・セクションを指定できます。C または C++ でこの指定を行う場合は、`section` 属性を使用できます。以下のコードは、`.on_chip_memory` という名前のメモリ内の変数 `foo`、および `.sdram` という名前のメモリ内の関数 `bar()` を配置する方法を示します。

例：特定のメモリ・セクションへのCコードの手動による割り当て

```
/* セクション属性を使用する場合は、データの初期化が必要 */
int foo __attribute__((section(".on_chip_memory"))) = 0;

void bar __attribute__((section(".sdram")))(void)
{
    foo++;
}
```

アセンブリでは、この処理は `.section` 指示文を使用して実行します。例えば、次に示す行以降のすべてのコードは、`on_chip_memory` という名前のメモリ・デバイスに配置されます。

```
.section .on_chip_memory
```



これらの機能の使用法の詳細については、GNU コンパイラおよびアセンブラの資料を参照してください。

ヒープおよびスタックの配置

ヒープとスタックは両方とも常に、`.rwdata` セクションと同じメモリ・パーティションに存在するように配置されます。スタックは、セクションの末尾から下方（下位アドレスへ）に拡張されます。ヒープは、`.rwdata` セクション内で最後に使用されたメモリから上方に拡張されます。

HAL は、実行時にヒープおよびスタックに十分なスペースがあるかどうかはチェックしません。ユーザは自分のプログラムが、ヒープとスタックに利用できるメモリの制限内で動作することを保証する必要があります。

ブート・モード

リセット・ベクタを持つメモリ・デバイスがプロセッサのブート・デバイスになります。外部フラッシュまたはアルテラの EPCS シリアル・コンフィギュレーション・デバイス、あるいはオンチップ RAM をブート・デバイスにすることができます。ブート・デバイスの性質にかかわらず、すべての HAL ベースのシステムは、すべてのコードおよびデータ・セクションが最初の段階でブート・デバイス内に格納されるように構築されます。これらのセクションは、ブート時にシステム・ライブラリ・プロパティ・ページ上で指定された実行位置にコピーされます。

`.text` セクションがブート・デバイスに配置されていない場合、Nios II IDE の Altera Flash Programmer は、`_start` を呼び出す前にすべてのコードおよびデータ・セクションをロードするブート・ローダを自動的にリセット・アドレスに配置します。EPCS デバイスからブートする場合、このローダの役割はハードウェアで提供されます。

ただし、`.text` セクションがブート・デバイス内に配置されている場合、システム内に個別のローダは存在しません。ローダを使用する代わりに、HAL 実行コマンド内の `_reset` エントリ・ポイントが直接呼び出されます。関数 `_reset` は、命令キャッシュを初期化した後に `_start` を呼び出します。そのため、フラッシュ・メモリから直接ブートおよび実行するアプリケーションの開発が可能です。

このモードで実行する場合、RAM へのロードが必要なセクションはすべて、HAL 実行コマンドで行う必要があります。セクション `.rwdata`、`.rodata`、および例外セクションは、必要に応じて、`alt_main()` を呼び出す前に自動的にロードされます。この自動ロードは、関数 `alt_load()` によって実行されます。

HAL システム・ライブラリ・ファイルへのパス

通常は、HAL システム・ライブラリ・ファイルは絶対に編集しないでください。ただし、参考のために、ヘッダ・ファイルなどを参照することは可能です。

HAL ファイルの検出

Nios II システム独自の性質により、HAL システム・ライブラリ・ファイルは、いくつかの個別ディレクトリに存在します。Nios II システムごとに異なるペリフェラルが含まれることがあるため、各システムの HAL システム・ライブラリも異なります。HAL 関連ファイルは、以下の場所のいずれかにあります。

- ほとんどの HAL システム・ライブラリ・ファイルは、<Nios II インストール・パス >/components ディレクトリに格納されています。
- HAL の汎用デバイス・モデルを定義するヘッダ・ファイルは、<Nios II インストール・パス >/components/altera_hal/HAL/inc/sys に格納されています。#include ディレクティブの場合、これらのファイルは、<Nios II インストール・パス >/components/altera_hal/HAL/inc/ に対して参照されます。例えば、DMA ドライバを取り込むには、#include sys/alt_dma/h を使用します。
- system.h ファイルは、特定の HAL システム・ライブラリ・プロジェクト用の Nios II IDE プロジェクト・ディレクトリに格納されています。
- Newlib ANSI C ライブラリ・ヘッダ・ファイルは、<Nios II インストール・パス >/bin に格納されています。

HAL 関数の置き換え

ユーザが関数を独自に実装するには、Nios II IDE アプリケーション・プロジェクトにファイルをインクルードします。実行ファイルを構築するときに、Nios II IDE はユーザの関数を最初に検出し、HAL バージョンの関数の代わりに使用します。