

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。

NII52005-1.1

はじめに

エンベデッド・システムは、一般に独自のデバイス・ドライバを必要とするアプリケーション固有のハードウェア機能を備えています。この章では、デバイス・ドライバを開発する方法と、それらのドライバを HAL (Hardware Abstraction Layer) システム・ライブラリに統合する方法について説明します。

ハードウェアとの直接的な連携は、デバイス・ドライバ・コードに限定する必要があります。一般に、ユーザ・プログラム・コードの大部分ではハードウェアへの低レベルなアクセスを行うことはありません。ハードウェアへのアクセスには、可能な限り、高レベル HAL API (Application Programming Interface) 関数を使用してください。それによって、コードの一貫性が向上し、ハードウェア・コンフィギュレーションが異なるその他の Nios® II システムへの移植性も向上します。

新しいドライバを作成する場合、ドライバは以下の 2 つレベルのいずれかで、HAL フレームワークに統合することができます。

- HAL API への統合
- ペリフェラル固有の API

HAL API への統合

HAL API への統合は、キャラクタ・モード・デバイスまたは DMA デバイスなど、HAL 汎用デバイス・モデル・クラスのいずれかに属するペリフェラルに適した方法です。HAL API への統合では、この章で説明するデバイス・アクセス関数を記述すると、ソフトウェアから標準 HAL API を介してデバイスにアクセスできるようになります。例えば、ASCII キャラクタを表示する新しい LCD 画面デバイスを使用する場合、キャラクタ・モード・デバイス・ドライバを記述します。このドライバを使用すると、プログラムから使い慣れた `printf()` 関数を呼び出して、LCD 画面にキャラクタを出力できます。

ペリフェラル固有の API


ペリフェラルが HAL 汎用デバイス・モデル・クラスのいずれにも属さない場合は、そのハードウェア実装専用のインタフェースを持つデバイス・ドライバを提供し、デバイスに対する API を HAL API から分離する必要があります。プログラムは、HAL API ではなくユーザが用意した関数を呼び出してハードウェアにアクセスします。

HAL API への統合では実装に労力を要しますが、デバイス进行操作する上で、HAL および C 標準ライブラリ API の利点が得られます。



HAL API への統合の詳細については、5-17 ページの「HAL へのデバイス・ドライバの統合」を参照してください。

この章の他のセクションは、HAL API にドライバを統合する方法、およびペリフェラル固有の API を使用してドライバを作成する方法に関するものです。

 C++ は HAL ベースのプログラムでサポートされていますが、HAL ドライバは C++ では記述しません。ドライバ・コードは C またはアセンブラのみとし、移植性を考慮して C を推奨します。

必要な知識

この章では、読者が HAL 用の C プログラミングに精通していることを前提としています。この章に入る前に、4-1 ページの「HAL を使用したプログラムの開発」の内容を確認してください。

HAL 用の新しいドライバを開発する手順は、デバイスの仕様によって大きく異なります。ただし、すべてのデバイス・クラスに以下の一般的な手順が適用されます。

1. レジスタについて記述するデバイス・ヘッダ・ファイルを作成します。このヘッダ・ファイルが、必要な唯一のインタフェースである場合もあります。
2. ドライバ機能を実装します。
3. `main ()` からテストします。
4. ドライバを HAL 環境に統合する最終段階へ進みます。
5. デバイス・ドライバを HAL フレームワークに統合します。

デバイス・ ドライバ作成 の開発フロー

SOPC Builder の概念

このセクションでは、ドライバ開発プロセスに関する理解を深めるために、アルテラの SOPC Builder ハードウェア・デザイン・ツールの概念について説明します。Nios II デバイス・ドライバを開発するのに、SOPC Builder を使用する必要はありません。

system.h と SOPC Builder の関係

system.h ヘッダ・ファイルは、Nios II システム・ハードウェアのすべてのソフトウェア記述を提供し、ドライバ開発の基本部分となるものです。ドライバは最下位レベルでハードウェアとやりとりを行うため、理解しやすいように **system.h** と Nios II プロセッサ・システム・ハードウェアを生成する SOPC Builder との関係を最初に説明しておきます。ハードウェア設計者は、SOPC Builder を使用して、Nios II プロセッサ・システムのアーキテクチャを指定し、必要なペリフェラルとメモリを統合します。したがって、各ペリフェラルの名前やコンフィギュレーションなど、**system.h** における定義は、SOPC Builder で選択したデザインを直接反映しています。



system.h ヘッダ・ファイルの詳細については、[4-1 ページの「HAL を使用したプログラムの開発」](#)を参照してください。

最適なハードウェア・コンフィギュレーションを目的とした SOPC Builder の使用

system.h 内に最適でない定義を見つけた場合は、SOPC Builder で基本ハードウェアを変更することによって、**system.h** の内容を修正できます。デバイス・ドライバを記述して、不完全なハードウェアに対処する前に、SOPC Builder で簡単にハードウェアを改良できないか検討することが重要です。

コンポーネント、デバイス、およびペリフェラル

SOPC Builder で使用する「コンポーネント」という用語は、システムに含まれるハードウェア・モジュールを表します。Nios II ソフトウェア開発において、SOPC Builder コンポーネントとは、ペリフェラルやメモリなどのデバイスのことです。以降のセクションで、SOPC Builder に密接に関連する内容を説明する場合、「コンポーネント」は「デバイス」や「ペリフェラル」と置き換え可能な意味で使用します。

ハードウェア へのアクセス

ソフトウェアでは、デバイスへのメモリマップド・インタフェースを抽象化するマクロを介して、ハードウェアにアクセスします。このセクションでは、各デバイスのハードウェア・インタフェースを定義するマクロについて説明します。

すべての SOPC Builder コンポーネントは、デバイス・ハードウェアとソフトウェアを定義するディレクトリを提供します。例えば、Nios II 開発キットに含まれる各コンポーネントには、<Nios II インストール・パス>/components ディレクトリに配置された専用のディレクトリがあります。多くのコンポーネントは、ハードウェア・インタフェースを定義するヘッダ・ファイルを提供します。ヘッダ・ファイルは、<コンポーネント名>_regs.h の名前で、特定のコンポーネント用の inc サブディレクトリに格納されています。例えば、アルテラが提供する JTAG UART コンポーネントの場合、そのハードウェア・インタフェースは、ファイル <Nios II インストール・パス>/components/altera_avalon_jtag_uart/inc/altera_avalon_jtag_uart_regs.h 内で定義されています。

_regs.h ヘッダ・ファイルは、以下のアクセスを定義します。

- 動作をサポートするデバイス内で、各レジスタに対する読み取りマクロや書き込みマクロを提供するレジスタ・アクセス・マクロ。これらのマクロは、IORD_<コンポーネントの名前>_<レジスタの名前>、および IOWR_<コンポーネントの名前>_<レジスタの名前> です (7-1 ページの「キャッシュ・メモリ」を参照)。
- レジスタ内の個々のビット・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・マスクとオフセット。これらのマクロの名前は、以下のとおりです。
 - フィールドのビット・マスクの場合、<コンポーネントの名前>_<レジスタの名前>_<フィールドの名前>_MSK
 - フィールド先頭のビット・オフセットの場合、<コンポーネントの名前>_<レジスタの名前>_<フィールドの名前>_OFST
 - ステータス・レジスタの PE フィールドへのアクセスの場合、ALTERA_AVALON_UART_STATUS_PE_MSK および ALTERA_AVALON_UART_STATUS_PE_OFST

デバイスのレジスタにアクセスするには、_regs.h ファイルに定義されたマクロのみを使用してください。デバイスの読み取りまたは書き込みを行うときに、プロセッサがデータ・キャッシュを使用しないようにするために、レジスタ・アクセス関数を使用する必要があります。また、この処理では、ソフトウェアが基本ハードウェアの変更による影響を受けやすくなるため、ハード・コード化された定数は絶対に使用しないでください。

まったく新しいハードウェア・デバイス用のドライバを記述する場合、_regs.h ヘッダ・ファイルを作成する必要があります。



HAL デバイス・クラス用ドライバの作成

キャッシュ管理およびデバイス・アクセスに関する影響の詳細は、7-1 ページの「[キャッシュ・メモリ](#)」を参照してください。_regs.h ファイルの詳細な例については、アルテラ提供の SOPC Builder コンポーネントのコンポーネント・ディレクトリを参照してください。

HAL は、多数の汎用デバイス・モデル・クラスをサポートしています (3-1 ページの「[HAL システム・ライブラリの概要](#)」を参照)。このセクションの説明に従ってデバイス・ドライバを作成して、既知のデバイス・クラスのいずれかに分類される特定のデバイスのインスタンスを HAL に記述します。このセクションでは、HAL がドライバ関数に均等にアクセスできるように、ドライバ関数のための統一されたインタフェースを定義します。

以下のセクションでは、以下のデバイスのクラスに対する API を定義します。

- キャラクタ・モード・デバイス
- ファイル・サブシステム
- DMA デバイス
- システム・クロックとして使用されるタイマ・デバイス
- タイムスタンプ・クロックとして使用されるタイマ・デバイス
- フラッシュ・メモリ・デバイス
- イーサネット・デバイス

以下のセクションでは、各デバイス・クラスに対してデバイス・ドライバを実装する方法と、それらのドライバを HAL ベース・システムで使用するための登録方法について説明します。

キャラクタ・モード・デバイス・ドライバ

このセクションでは、デバイス・インスタンスの作成方法とキャラクタ・デバイスの登録方法について説明します。

デバイス・インスタンスの作成

デバイスをキャラクタ・モード・デバイスとして使用できるようにするには、そのデバイスに alt_dev 構造体のインスタンスを提供する必要があります。alt_dev 構造体は、以下のコードで定義されています。

```
typedef struct {
    alt_llist    llist;      /* 内部使用 */
    const char*  name;
    int (*open) (alt_fd* fd, const char* name, int flags, int mode);
    int (*close) (alt_fd* fd);
    int (*read)  (alt_fd* fd, char* ptr, int len);
    int (*write) (alt_fd* fd, const char* ptr, int len);
}
```

```

int (*lseek) (alt_fd* fd, int ptr, int dir);
int (*fstat) (alt_fd* fd, struct stat* buf);
int (*ioctl) (alt_fd* fd, int req, void* arg);
} alt_dev;

```

この構造体は、基本的には関数ポインタの集合です。これらの関数は、HAL ファイル・システムへのユーザ・アクセスにตอบสนองして呼び出されます。例えば、このデバイスに対応するファイル名を指定して関数 `open()` を呼び出すと、結果としてこの構造体に割り当てられた `open()` 関数が呼び出されます。



`open()`、`close()`、`read()`、`write()`、`lseek()`、`fstat()`、および `ioctl()` 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。

これらの関数はいずれも、グローバル・エラー・ステータスの `errno` を直接変更することはありません。ステータスを変更する代わりに、`errno.h` で定義された該当するエラー・コードの負の値を返します。

例えば、`ioctl()` 関数は、要求を処理できない場合には、`errno` を `ENOTTY` に直接設定するのではなく、戻り値として `-ENOTTY` を返します。`errno` は、これらの関数を呼び出す HAL システム・ルーチンが適切に設定します。

これらの関数の関数プロトタイプは、`int` 型ではなく `alt_fd*` 型の入力ファイル・ディスクリプタ引数を取るという点が、アプリケーション・レベルのプロトタイプとは異なります。

新しい `alt_fd` 構造体は、`open()` を呼び出したときに作成されます。この構造体のインスタンスは、関連するファイル・ディスクリプタに対して実行されたすべての関数呼び出しに、入力引数として渡されます。

`alt_fd` 構造体は、以下のコードで定義されています。

```

typedef struct
{
    alt_dev* dev;
    void* priv;
    int fd_flags;
} alt_fd;

```

ここで、

- `dev` は、使用中のデバイスのデバイス構造体へのポインタです。
- `fd_flags` は、`open()` に渡される `flags` の値です。
- `priv` は、HAL システム・コードで使用されない不定値です。

- `priv` は、ドライバが内部使用に必要なファイル・ディスクリプタごとの情報を格納するために利用できます。

ドライバで、`alt_dev` 構造体内のすべての関数を提供する必要はありません。関数ポインタが `NULL` に設定されている場合は、デフォルト動作が実行されます。表 5-1 に、利用可能な関数それぞれのデフォルト動作を示します。

| 関数 | デフォルト動作 |
|--------------------|--|
| <code>open</code> | デバイスが事前に <code>TIOCEXCL ioctl()</code> 要求によってロックされていない限り、このデバイスに対して <code>open()</code> を呼び出すと成功します。 |
| <code>close</code> | ファイル・ディスクリプタが有効な場合、このデバイスに対して <code>close()</code> を呼び出すと、必ず成功します。 |
| <code>read</code> | このデバイスに対して <code>read()</code> を呼び出すと、必ず失敗します。 |
| <code>write</code> | このデバイスに対して <code>write()</code> を呼び出すと、必ず失敗します。 |
| <code>lseek</code> | このデバイスに対して <code>lseek()</code> を呼び出すと、必ず失敗します。 |
| <code>fstat</code> | デバイスは、自身をキャラクタ・モード・デバイスとして識別します。 |
| <code>ioctl</code> | デバイスを参照しなければ処理できない <code>ioctl()</code> は失敗します。 |

関数ポインタに加えて、`alt_dev` 構造体にはさらに、`l1ist` と `name` の 2 つのフィールドが含まれています。`l1ist` は内部で使用するためのもので、常に `ALT_LLIST_ENTRY` の値に設定する必要があります。`name` は、HAL ファイル・システム内のデバイスの位置で、`system.h` で定義されたデバイスの名前です。

キャラクタ・デバイスの登録

`alt_dev` 構造体のインスタンスを作成したら、これを HAL に登録し、以下の関数を呼び出すことによって、システムでデバイスを利用できるようにする必要があります。

```
int alt_dev_reg (alt_dev* dev)
```

この関数は登録するデバイス構造体を唯一の入力引数として受け取ります。戻り値のゼロは正常に終了したことを示します。負の戻り値は、デバイスが登録できないことを示します。


デバイスがシステムに登録されると、HAL API および ANSI C 標準ライブラリを使用してデバイスにアクセスできます (4-1 ページの「HAL を使用したプログラムの開発」を参照)。デバイスのノード名は、`alt_dev` 構造体で指定されます。

ファイル・サブシステム・ドライバ

ファイル・サブシステム・デバイス・ドライバは、グローバル HAL ファイル・システム内の指定されたマウント・ポイントでのファイル・アクセスを処理します。

デバイス・インスタンスの作成

ファイル・システムの作成と登録は、キャラクタ・モード・デバイスの作成と登録とよく似ています。ファイル・システムを利用可能にするには、`alt_dev` 構造体のインスタンスを作成します (5-5 ページの「キャラクタ・モード・デバイス・ドライバ」を参照)。唯一の相違点は、デバイスの `name` フィールドがファイル・サブシステムのマウント・ポイントを表すことです。もちろん、キャラクタ・モード・デバイスの場合と同様に、`read()` や `write()` など、ファイル・サブシステムにアクセスするのに必要な関数も、ユーザが用意する必要があります。

 `fstat()` を実装していない場合、デフォルトの動作によって、キャラクタ・モード・デバイスの値が返され、これはファイル・サブシステムに対して不正な動作となります。

ファイル・サブシステム・デバイスの登録

ファイル・サブシステムは、以下の関数を使用して登録できます。

```
int alt_fs_reg (alt_dev* dev)
```

この関数は登録するデバイス構造体を唯一の入力引数として受け取ります。負の戻り値は、ファイル・システムが登録できないことを示します。

ファイル・サブシステムが HAL ファイル・システムに登録されると、HAL API および ANSI C 標準ライブラリを使用して、このサブシステムにアクセスできます (4-1 ページの「HAL を使用したプログラムの開発」を参照)。ファイル・サブシステムのマウント・ポイントは、`alt_dev` 構造体で指定された `name` です。

タイマ・デバイス・ドライバ

このセクションでは、システム・クロックとタイムスタンプ・ドライバについて説明します。

システム・クロック・ドライバ

システム・クロック・デバイス・モデルを使用するには、周期的な「チック」を生成するドライバが必要です（[4-1 ページの「HAL を使用したプログラムの開発」](#)を参照）。1 つのシステムには 1 つのシステム・クロック・ドライバしか存在できません。そのため、システム・クロック・ドライバは、周期的な割り込みを生成するタイマ・ペリフェラルに対する割り込みサービス・ルーチン（ISR）として実装します。このドライバは、以下の関数を周期的に呼び出す必要があります。

```
void alt_tick (void)
```

`alt_tick()` は割り込み処理中に呼び出されます。

システム・クロック・ドライバを登録するには、以下の関数を呼び出します。

```
int alt_sysclk_init (alt_u32 nticks)
```

入力引数 `nticks` は、1 秒あたりのシステム・クロック・チック数です。この値はシステム・クロック・ドライバによって決まります。この関数の戻り値は、成功時にはゼロ、それ以外はゼロではない値です。



割り込みサービス・ルーチンの記述に関する詳細は、[6-1 ページの「例外処理」](#)を参照してください。

タイムスタンプ・ドライバ

タイムスタンプ・ドライバは、`alt_timestamp_start()`、`alt_timestamp()`、および `alt_timestamp_freq()` の 3 つのタイムスタンプ関数の実装を提供します。システムには、1 つのタイムスタンプ・ドライバしか存在できません。



これらの関数の使用方法に関する詳細は、[4-1 ページの「HAL を使用したプログラムの開発」](#) および [10-1 ページの「HAL API リファレンス」](#)の章を参照してください。

フラッシュ・デバイス・ドライバ

このセクションでは、フラッシュ・ドライバの作成方法とフラッシュ・デバイスの登録方法について説明します。

フラッシュ・ドライバの作成

フラッシュ・デバイス・ドライバは、`sys/alt_flash_dev.h` で定義された `alt_flash_dev` 構造体のインスタンスを持つ必要があります。この構造体は以下のコードで表されます。

```
struct alt_flash_dev
{
    alt_llist                llist; // 内部使用のみ
    const char*             name;
    alt_flash_open          open;
    alt_flash_close         close;
    alt_flash_write         write;
    alt_flash_read          read;
    alt_flash_get_flash_info get_info;
    alt_flash_erase_block   erase_block;
    alt_flash_write_block   write_block;
    void*                   base_addr;
    int                     length;
    int                     number_of_regions;
    flash_region            region_info[ALT_MAX_NUMBER_OF_FLASH_REGIONS];
};
```

最初のパラメータ `llist` は内部で使用するためのもので、常に `ALT_LLIST_ENTRY` の値に設定する必要があります。`name` は、HAL ファイル・システム内のデバイスの位置で、`system.h` で定義されたデバイスの名前です。

`open` から `write_block` までの 8 つのフィールドは、以下の関数へのユーザ API 呼び出しの支援機能を実装する関数ポインタです。

- `alt_flash_open_dev()`
- `alt_flash_close_dev()`
- `alt_flash_write()`
- `alt_write_flash()`
- `alt_read_flash()`
- `alt_get_flash_info()`
- `alt_erase_flash_block()`
- `alt_write_flash_block()`

ここで、

- `base_addr` パラメータは、フラッシュ・メモリのベース・アドレスです。
- `length` は、フラッシュのバイト・サイズです。
- `number_of_regions` は、フラッシュ内の消去領域の数です。
- `region_info` には、フラッシュ・デバイス内のブロックの位置とサイズに関する情報が含まれています。



`flash_region` 構造体の形式に関する詳細は、4-14 ページの「[フラッシュ・デバイスの使用](#)」を参照してください。

共通フラッシュ・インタフェース (CFI) 準拠のデバイスなど、フラッシュ・デバイスには、領域数とそれらのコンフィギュレーションを実行時に読み出せるものもあります。実行時に読み出せない場合、これらの 2 つのフィールドはコンパイル時に定義しなければなりません。

フラッシュ・デバイスの登録

`alt_flash_dev` 構造体のインスタンスを作成したら、以下の関数を呼び出してデバイスを HAL システムで利用できるようにする必要があります。

```
int alt_flash_device_register( alt_flash_fd* fd)
```

この関数は、登録するデバイス構造体を唯一の入力引数として受け取ります。戻り値のゼロは正常に終了したことを示します。負の戻り値は、デバイスが登録できなかったことを示します。

DMA デバイス・ドライバ

HAL は、DMA 転送を受信チャネルと送信チャネルの 2 つのエンドポイント・デバイスでの制御対象としてモデル化します。このセクションでは、DMA チャネルの各タイプに対するドライバについて個々に説明します。

HAL DMA デバイス・モデルの詳細な説明については、4-20 ページの「[DMA デバイスの使用](#)」を参照してください。

DMA デバイス・ドライバ・インタフェースは、`sys/alt_dma_dev.h` で定義されます。

DMA 送信チャンネル

DMA 送信チャンネルは、以下の `alt_dma_txchan` 構造体のインスタンスを作成することによって構築されます。

```
typedef struct alt_dma_txchan_dev_s alt_dma_txchan_dev;
struct alt_dma_txchan_dev_s
{
    alt_llist    llist;
    const char* name;
    int          (*space) (alt_dma_txchan dma);
    int          (*send) (alt_dma_txchan dma,
                        const void* from,
                        alt_u32 len,
                        alt_txchan_done* done,
                        void* handle);
    int          (*ioctl) (alt_dma_txchan dma, int req, void* arg);
};
```

表 5-2 に、利用可能なフィールドとそれらの機能を示します。

| フィールド | 機能 |
|-------|--|
| llist | このフィールドは内部で使用するためのもので、常に ALT_LLIST_ENTRY の値に設定する必要があります。 |
| name | <code>alt_dma_txchan_open()</code> の呼び出しでこのチャンネルを示す名前。name は、 <code>system.h</code> で定義されるデバイスの名前です。 |
| space | デバイスのキューに格納できる追加送信要求数を返す関数へのポインタ。入力引数は、 <code>alt_dma_txchan_dev</code> 構造体へのポインタです。 |
| send | ユーザ API 関数 <code>alt_dma_txchan_send()</code> を呼び出した結果として呼び出される関数へのポインタ。この関数は、DMA デバイスに送信要求を送信します。 <code>alt_txchan_send()</code> に渡されるパラメータは、 <code>send()</code> に直接渡されます。パラメータおよび戻り値の説明は、10-20 ページの「 <code>alt_dma_txchan_send()</code> 」を参照してください。 |
| ioctl | この関数はデバイス固有の I/O 制御を実行します。デバイスがサポートできる一般的なオプションのリストは、 <code>sys/alt_dma_dev.h</code> を参照してください。 |

`space` 関数および `send` 関数の両方を定義する必要があります。 `ioctl` フィールドが `null` に設定されている場合、このデバイスに対して `alt_dma_txchan_ioctl()` を呼び出すと、`-ENOTTY` が返されます。

`alt_dma_txchan` 構造体のインスタンスを作成したら、以下の関数を呼び出して、デバイスを HAL システムに登録して利用可能にする必要があります。

```
int alt_dma_txchan_reg (alt_dma_txchan_dev* dev)
```

入力引数 `dev` は、登録するデバイスです。戻り値は成功時にはゼロになり、デバイスが登録できない場合は負の値になります。

DMA 受信チャンネル

DMA 受信チャンネルは、以下の `alt_dma_rxchan` 構造体のインスタンスを作成すれば構築されます。

```
typedef alt_dma_rxchan_dev_s alt_dma_rxchan;
struct alt_dma_rxchan_dev_s
{
    alt_llist    list;
    const char* name;
    alt_u32     depth;
    int         (*prepare) (alt_dma_rxchan  dma,
                           void*          data,
                           alt_u32        len,
                           alt_rxchan_done* done,
                           void*          handle);
    int         (*ioctl) (alt_dma_rxchan dma, int req, void* arg);
};
```

表 5-3 に、利用可能なフィールドとそれらの機能を示します。

| フィールド | 機能 |
|---------|---|
| l1ist | この機能は内部で使用するためのもので、常に <code>ALT_LLIST_ENTRY</code> の値に設定する必要があります。 |
| name | <code>alt_dma_rxchan_open()</code> の呼び出しで、このチャンネルを示す名前。name は、 <code>system.h</code> で定義されるデバイスの名前です。 |
| depth | どの時点でも未処理状態にすることができる受信要求の総数。 |
| prepare | ユーザ API 関数 <code>alt_dma_rxchan_prepare()</code> を呼び出した結果として呼び出される関数へのポインタ。この関数は DMA デバイスに受信要求を送信します。 <code>alt_dma_rxchan_prepare()</code> に渡されたパラメータは、 <code>prepare()</code> に直接渡されます。パラメータおよび戻り値の説明は、10-14 ページの「 alt_dma_rxchan_prepare() 」を参照してください。 |
| ioctl | この関数は、デバイス固有の I/O 制御を実行します。デバイスがサポートする一般的なオプションのリストは、 <code>sys/alt_dma_dev.h</code> を参照してください。 |

`prepare()` 関数を定義する必要があります。

`ioctl` フィールドが `null` に設定されている場合、このデバイスに対して `alt_dma_rxchan_ioctl()` を呼び出すと、`-ENOTTY` が返されます。

`alt_dma_rxchan` 構造体のインスタンスを作成したら、以下の関数を呼び出してデバイスを HAL システムに登録して利用可能にする必要があります。

```
int alt_dma_rxchan_reg (alt_dma_rxchan_dev* dev)
```

入力引数 `dev` は、登録するデバイスです。戻り値は成功時にはゼロになり、デバイスが登録できない場合は負の値になります。

イーサネット・デバイス・ドライバ

イーサネット・デバイス対応の HAL 汎用デバイス・モデルを利用すると、MicroC/OS-II オペレーティング・システム上で動作する軽量 IP (lwIP) TCP/IP スタックへのアクセスが可能になります。このセクションで定義するドライバ関数を供給することによって、新しいイーサネット・デバイスをサポートできます。

新しいイーサネット・デバイス用のデバイス・ドライバの作成を検討する前に、lwIP のアルテラ・ポートとその使用法の基本を理解しておく必要があります (9-1 ページの「イーサネットと Lightweight IP」を参照)。

新しいイーサネット・デバイス・ドライバを作成する最も簡単な方法は、SMSC lan91c111 デバイスに対するアルテラの実装から始めて、これをユーザのイーサネット・メディア・アクセス・コントローラ (MAC) に適合するよう修正することです。このセクションでは、ユーザがこの方法を利用するものと仮定しています。すなわち、lwIP スタック実装の詳細を学習するのではなく、動作確認済みの例を修正するだけでドライバを作成できます。したがって、このセクションでは、lwIP スタックのアルテラのポートを内部実装する上での最小限の内容について説明します。



lwIP 実装の詳細については、www.sics.se/~adam/lwip/doc/lwip.pdf を参照してください。

lan91c111 ドライバのソース・コードは、`src` ディレクトリおよび `inc` ディレクトリ内の
`<Nios II インストール・パス>/components/altera_avalon_lan91c111/UCOSII/`
 に、Nios II 開発キットとともに提供されています。イーサネット・デバイス・ドライバ・インタフェースは、
`<lwIP コンポーネント・パス>/UCOSII/inc/alt_lwip_dev.h`
 で定義されています。

以下のセクションでは、新しいイーサネット・デバイス用のドライバを提供する方法について説明します。

alt_lwip_dev_list のインスタンスの提供

以下のコードは、各デバイス・ドライバで提供する必要がある alt_lwip_dev_list 構造体のインスタンスを示します。

```
typedef struct
{
    alt_llist    llist;      /* 内部使用 */
    alt_lwip_dev dev;
} alt_lwip_dev_list;

struct alt_lwip_dev
{
    /* netif ポインタは構造体の最初の要素であることが必要 */
    struct netif* netif;
    const char* name;
    err_t (*init_routine)(struct netif*);
    void (*rx_routine)();
};
```

name パラメータは、system.h で定義されるデバイスの名前です。

lwIP システム・コードは netif 構造体を内部で使用して、デバイス・ドライバへのインタフェースを定義します。netif 構造体は、<lwIP コンポーネントのパス >/UCOSII/src/downloads/lwip-0.7.2/src/include/lwip の netif.h で定義されています。特に、netif 構造体には、以下の内容が含まれます。

- インタフェースの MAC アドレス用のフィールド
- インタフェースの IP アドレス用のフィールド
- MAC デバイスを初期化する低レベル関数への関数ポインタ
- パケットを送信する低レベル関数へのポインタ
- パケットを受信する低レベル関数への関数ポインタ

init_routine() の提供

alt_lwip_dev 構造体の init_routine は、netif 構造体の設定とハードウェアの初期化を行う関数へのポインタです。この関数は、ターゲットのイーサネット・デバイス用に提供する必要があります。この関数のプロトタイプは以下のとおりです。

```
err_t init_routine(struct netif* netif)
```

`init_routine()` は、ルーチン `get_mac_addr()` および `get_ip_addr()` を呼び出すことによって、MAC アドレスおよび IP アドレスに対する `netif` フィールドに入力します。これらの関数は、9-1 ページの「イーサネットと Lightweight IP」で定義されています。さらに、`init_routine()` では、必要な低レベル・レジスタ・アクセスを実行して、ハードウェアをコンフィギュレーションする必要があります。

`output()` と `linkoutput()` の提供

また、`init_routine()` 関数では、`output()` および `link_output()` の2つの送信関数へのポインタに対する `netif` フィールドにも入力する必要があります。

`link_output()` は、イーサネット・ハードウェア上でパケットを送信します。`link_output()` 関数のプロトタイプは以下のとおりです。

```
link_output(struct netif *netif, struct pbuf *p)
```

`link_output()` は、イーサネット・インタフェース上で IP パケットを送信します。IP アドレスに関連付けられる MAC アドレスに対して ARP 要求が発行され、次に `link_output()` を呼び出してパケットを送信します。`link_output()` 関数のプロトタイプは、以下のとおりです。

```
output(struct netif *netif,  
       struct pbuf *p,  
       struct ip_addr *ipaddr)
```

`rx_routine()` の提供

`alt_lwip_dev` 構造体の `rx_routine` は、TCP/IP スタックへの着信パケットを受信するために呼び出されるルーチンへの関数ポインタです。

新しいパケットが到着すると、割り込み要求 (IRQ) が生成されます。関連付けられた割り込みサービス・ルーチン (ISR) がこの割り込みをクリアし、`rx_mbox` という名前のメッセージ・キューにメッセージを送信します。このメッセージ・ボックスは、ファイル `<lwIP コンポーネントのパス>/UCOSII/src/alt_lwip_dev.c` で定義されています。

`rx_thread` は、`rx_mbox` 内で新しいメッセージを検出すると、`rx_routine()` を呼び出します。`rx_routine()` によって、ハードウェアからパケットが受信され、TCP/IP スタックに渡されます。

HAL への デバイス・ ドライバの 統合

この関数のプロトタイプは以下のとおりです。

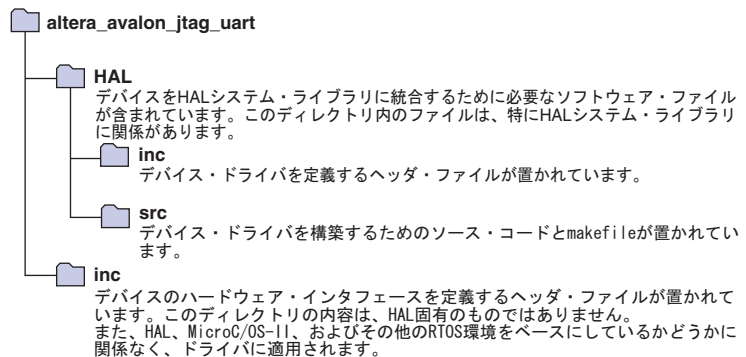
```
void rx_func()
```

このセクションでは、HAL の機能を活用して、システム初期化中にデバイス・ドライバを自動的にインスタンス化および登録する方法について説明します。いずれかの HAL 汎用デバイス・モデルに対してデバイス・ドライバを作成した場合でも、ペリフェラル専用のデバイス・ドライバを作成した場合でも、このサービスを活用できます。HAL が提供する自動化機能の主な利点は、HAL ディレクトリ構造の適切な場所にファイルを配置するプロセスです。

HAL デバイスのディレクトリ構造

各ペリフェラルは、特定の SOPC Builder コンポーネント・ディレクトリに提供されたファイルで定義されています(5-3 ページの「ハードウェアへのアクセス」を参照)。このセクションでは、アルテラの JTAG UART コンポーネントの例を使用して、ファイルの位置を示します。図 5-1 に、JTAG UART コンポーネント・ディレクトリのディレクトリ構造を示します。このディレクトリは、<Nios II インストール・パス>/components ディレクトリ内に配置されています。

図 5-1. HAL ペリフェラルのディレクトリ構造



HAL 用デバイス・ドライバ・ファイル

このセクションでは、必要なファイルを提供して、デバイス・ドライバを HAL に統合する方法について説明します。

デバイスの HAL ヘッダ・ファイルと alt_sys_init.c

HALの中心となるのは、自動生成されたソース・ファイルの alt_sys_init.c です。alt_sys_init.c には、システム内のすべてのサポート対象デバイスのデバイス・ドライバを初期化するために、HAL が使用するソース・コードが含まれています。特に、このファイルでは alt_sys_init() 関数が定義されています。この関数は、main() の前に呼び出されて、すべてのデバイスを初期化し、プログラムからそれらのデバイスを利用できるようにします。

以下のコードは、alt_sys_init.cファイルから一部分を抜粋したものです。

例：ドライバの初期化を実行する alt_sys_init.c ファイルの一部


```
#include "system.h"
#include "sys/alt_sys_init.h"

/*
 * デバイス・ヘッダ
 */
#include "altera_avalon_timer.h"
#include "altera_avalon_uart.h"

/*
 * デバイス・ストレージの割り当て
 */
ALTERA_AVALON_UART_INSTANCE( UART1, uart1 );
ALTERA_AVALON_TIMER_INSTANCE( SYSCLK, sysclk );

/*
 * デバイスの初期化
 */
void alt_sys_init( void )
{
    ALTERA_AVALON_UART_INIT( UART1, uart1 );
    ALTERA_AVALON_TIMER_INIT( SYSCLK, sysclk );
}
```

新しいソフトウェア・プロジェクトを作成すると、Nios II 統合開発環境 (IDE) は、SOPC Builder システムの固有のハードウェア内容に適合するように、alt_sys_init.c の内容を自動的に生成します。Nios II IDE は、ジェネレータ・ユーティリティ gtf-generate を呼び出して、alt_sys_init.c を作成します。

 gtf-generate は、明示的に呼び出す必要はありません。HAL の低レベル動作での gtf-generate について述べているため、ここでは参考として説明しています。

プロセッサから認識可能な各デバイスに対して、ジェネレータ・ユーティリティは、デバイスの HAL/inc ディレクトリ内で関連するヘッダ・ファイルを検索します。ヘッダ・ファイルの名前は、SOPC Builder コンポーネント名によって異なります。

例えば、アルテラの JTAG UART コンポーネントの場合、ジェネレータはファイル altera_avalon_jtag_uart/HAL/inc/altera_avalon_jtag_uart.h を検出します。ジェネレータ・ユーティリティは、該当するヘッダ・ファイルを検出すると、alt_sys_init.c にコードを挿入して、以下の動作を実行します。

- デバイスのヘッダ・ファイルをインクルードします。
5-18 ページの「例：ドライバの初期化を実行する alt_sys_init.c ファイルの一部」の /* デバイス・ヘッダ */ を参照してください。
- マクロ <デバイスの名前>_INSTANCE を呼び出して、デバイスにストレージを割り当てます。
5-18 ページの「例：ドライバの初期化を実行する alt_sys_init.c ファイルの一部」の /* デバイス・ストレージの割り当て */ のセクションを参照してください。
- alt_sys_init() 関数内でマクロ <デバイスの名前>_INIT を呼び出して、デバイスを初期化します。
5-18 ページの「例：ドライバの初期化を実行する alt_sys_init.c ファイルの一部」の /* デバイスの初期化 */ のセクションを参照してください。

これらの *_INSTANCE マクロおよび *_INIT マクロは、関連付けられたデバイス・ヘッダ・ファイルで定義する必要があります。

例えば、altera_avalon_jtag_uart.h では、マクロ

```
ALTERA_AVALON_JTAG_UART_INSTANCE と
```

```
ALTERA_AVALON_JTAG_UART_INIT を定義する必要があります。
```

*_INSTANCE マクロは、ドライバが必要とする静的メモリ割り当てをデバイスごとに実行します。*_INIT マクロは、デバイスのランタイム初期化を実行します。どちらのマクロも 2 つの入力引数を受け取ります。最初の引数は大文字で表したデバイス・インスタンスの名前、2 番目の引数は小文字で表したデバイス名です。これは、システム生成時に SOPC Builder のコンポーネントに与えられた名前です。これらの入力パラメータを使用して、system.h ファイルからデバイス固有のコンフィギュレーション情報を抽出できます。

詳細な例は、アルテラ提供のデバイス・ドライバを参照してください。

- 👉 プロジェクトの再構築時間を短縮するには、ペリフェラル・ヘッダ・ファイルに system.h を直接指定しないようにします (alt_sys_init.c で既に指定されています)。

SOPC Builder コンポーネント用のデバイス・ドライバを公開するには、コンポーネントのディレクトリ内にファイル `HAL/inc/<component_name>.h` を提供します。このファイルは、上記のようにマクロ `<COMPONENT_NAME>_INSTANCE` および `<COMPONENT_NAME>_INIT` を定義するために必要です。このようにデバイスに対応したインフラストラクチャを準備すれば、HAL システム・ライブラリは、`main()` を呼び出す前に、デバイス・ドライバを自動的にインスタンス化して登録します。

デバイス・ドライバのソース・コード

一般に、デバイス・ドライバは、ヘッダでは完全に定義できません(5-18 ページの「[デバイスの HAL ヘッダ・ファイルと alt_sys_init.c](#)」を参照)。ほとんどの場合、コンポーネントは、システム・ライブラリに組み込まれるその他のソース・コードも提供する必要があります。

必要なソース・コードは、`HAL/src` ディレクトリに配置しなければなりません。さらに、`makefile` の一部分の `component.mk` を取り込む必要があります。`component.mk` ファイルは、システム・ライブラリに含まれるソース・ファイルをリストします。ファイル名をスペースで区切ると、複数のファイルをリストできます。以下のコードは、アルテラの JTAG UART デバイスに対する `makefile` の例を示します。

例：component.mk makefile の例

```
C_LIB_SRCS += altera_avalon_uart.c
ASM_LIB_SRCS +=
INCLUDE_PATH +=
```

Nios II IDE は、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・プロジェクトをコンパイルするときに、`component.mk` ファイルをトップ・レベルのmakefileに自動的に取り込みます。`component.mk` は、利用できる `make` 変数を変更できますが、これは `C_LIB_SRCS`、`ASM_LIB_SRCS` および `INCLUDE_PATH` に限定されます。表 5-4 にこれらの変数を示します。

| make 変数 | 意味 |
|---------------------------|--|
| <code>C_LIB_SRCS</code> | システム・ライブラリに組み込まれる C ソース・ファイルのリスト。 |
| <code>ASM_LIB_SRCS</code> | システム・ライブラリに組み込まれるアセンブラ・ソース・ファイルのリスト（これらは C プリプロセッサで前処理されます）。 |
| <code>INCLUDE_PATH</code> | インクルード検索パスに追加するディレクトリのリスト。ディレクトリ <code><component>/HAL/inc</code> は自動的に追加されるため、コンポーネントで明示的に定義する必要はありません。 |

`component.mk` によって、その他の `make` 規則とマクロを必要に応じて追加できますが、マクロ名は相互運用性を確保するために、ネーム空間規則に準拠する必要があります（5-22 ページの「ネーム空間の割り当て」を参照）。

要約

要約すると、HAL フレームワークにデバイス・ドライバを統合するには、以下の処理を実行することが必要です。

- `*_INSTANCE` マクロおよび `*_INIT` マクロを定義するインクルード・ファイルを作成し、デバイスの `HAL/inc` ディレクトリに配置します。
- デバイスを操作するソース・コード・ファイルを作成し、そのファイルをデバイスの `HAL/src` ディレクトリに配置します。
- makefile の一部分、`component.mk` を記述し、`HAL/src` ディレクトリに配置します。

ドライバ・フットプリントの削減

HAL では、`ALT_USE_SMALL_DRIVERS` という名前の C プリプロセッサ・マクロが定義されています。このマクロをドライバ・ソース・コード内で使用すると、最小のコード・フットプリントを必要とするシステムに対応した、代替動作が実現できます。Nios II IDE のオプションを利用すれば、機能が限定されたデバイス・ドライバを有効にすることができます。`ALT_USE_SMALL_DRIVERS` が定義されていない場合、ドライバ・ソース・コードはドライバのフル機能バージョンを実装します。マクロが定義されている場合、ソース・コードは機能を限定したドライバを提供します。例えば、あるドライバはデフォルトでは割り込み駆動式の動作を実装できますが、`ALT_USE_SMALL_DRIVERS` が定義されていれば、ポーリング式（より軽量と予想される）動作を実装できます。

デバイス・ドライバを作成する際に、`ALT_USE_SMALL_DRIVERS` の値を無視するように選択すれば、マクロの定義にかかわらず、同じバージョンのドライバが使用されます。

ネーム空間の割り当て

SOPC Builder システムでデバイスによって定義されるシンボルの名前が重複するのを回避するために、すべてのグローバル・シンボルに定義済みプリフィックスを付加する必要があります。グローバル・シンボルには、グローバル変数と関数名があります。デバイス・ドライバの場合、プリフィックスは SOPC Builder コンポーネントの名前の後に下線を付加したものです。このネーミングでは文字列が長くなるため、これに代わる短縮形式も利用できます。この短縮形式ではベンダ名が基本となり、例えば `alt_` は、アルテラが供給するコンポーネントに対するプリフィックスです。すべてのコンポーネントの相互運用性は、コンポーネントを供給するベンダがテストすることが求められます。

例えば、`altera_avalon_jtag_uart` コンポーネントの場合、以下の関数名は有効です。

- `altera_avalon_jtag_uart_init()`
- `alt_jtag_uart_init()`

以下の名前は無効です。

- `avalon_jtag_uart_init()`
- `jtag_uart_init()`

ソース・ファイルは検索パスを使用して検索されるため、これらのネーム空間制限事項はデバイス・ドライバのソース・ファイルとヘッダ・ファイルにも適用されます。

デフォルト・ デバイス・ ドライバの 置き換え

すべての SOPC Builder コンポーネントは、HAL デバイス・ドライバを提供するを選択できます (5-17 ページの「HAL へのデバイス・ドライバの統合」を参照)。ただし、コンポーネントに用意されたドライバがアプリケーションに適さない場合は、Nios II IDE のシステム・ライブラリ・プロジェクト・ディレクトリに別のドライバを供給して、デフォルトのドライバを置き換えることができます。

Nios II IDE は検索パスを使用して、すべてのインクルード・ファイルとソース・ファイルを検索します。この場合、システム・ライブラリ・プロジェクト・ディレクトリが常に最初に検索されます。例えば、コンポーネントがヘッダ・ファイル `alt_my_component.h` を提供し、システム・ライブラリ・プロジェクト・ディレクトリにもファイル `alt_my_component.h` が含まれている場合、コンパイル時にシステム・ライブラリ・プロジェクト・ディレクトリに提供されるバージョンが使用されます。これと同じメカニズムによって、C ソース・ファイルとアセンブラ・ソース・ファイルを置き換えることが可能です。

