

この資料は英語版を翻訳したもので、内容に相違が生じる場合には原文を優先します。こちらの日本語版は参考用としてご利用ください。設計の際には、最新の英語版で内容をご確認ください。

NII52013-8.0.0

概要

NicheStack[®] TCP/IP スタック - Nios[®] II Edition は、実装面積の小さい TCP/IP (Transmission Control Protocol/Internet Protocol) スイートです。NicheStack TCP/IP スタック実装の目的は、豊富な機能を備えた TCP/IP スタックを提供しながらリソース使用量を低減することです。NicheStack TCP/IP スタックは、メモリ実装面積が小さいエンベデッド・システムで使用するために設計されており、Nios[®] II プロセッサ・システムに適しています。

アルテラは、Nios II 統合開発環境 (IDE) および Nios II ボード・サポート・パッケージ (BSP) ジェネレータを通じて使用可能なソフトウェア・コンポーネントとして、NicheStack TCP/IP スタックを提供しており、これはシステム・ライブラリまたは BSP に追加することができます。NicheStack TCP/IP スタックは、以下の機能を備えています。

- 複数のネットワーク・インタフェースによるパケット転送を含むインターネット・プロトコル (IP)
- ネットワークのメンテナンスおよびデバッグ用の ICMP (Internet Control Message Protocol)
- UDP (User Datagram Protocol)
- 輻輳制御、ラウンドトリップ時間 (RTT) 見積もり、および高速リカバリおよび再送機能を備えた TCP (Transmission Control Protocol)
- DHCP (Dynamic host configuration protocol)
- イーサネット用 ARP (アドレス解決プロトコル)
- 標準 API (Application Programming Interface)

この章では、NicheStack TCP/IP スタックを Nios II プロセッサでのみ使用する方法を詳しく説明します。この章は、以下の項で構成されています。

- 11-2 ページの「前提条件」
- 11-2 ページの「はじめに」
- 11-4 ページの「その他の TCP/IP スタック・プロバイダ」
- 11-4 ページの「NicheStack TCP/IP スタックの使用」
- 11-11 ページの「Nios II IDE での NicheStack TCP/IP スタックのコンフィギュレーション」
- 11-13 ページの「詳細情報について」
- 11-14 ページの「既知の制限」

前提条件

この章の情報を最大限に活用するには、以下のトピックについて基本的な知識が必要です。

- ソケット。ソケットによるプログラミングのトピックに関する書籍は多数あります。Richard Stevens 著の *Unix Network Programming* や Douglas Comer 著の *Internetworking with TCP/IP Volume 3* は優れた解説書です。
- Nios II エンベデッド・デザイン・スイート (EDS) Nios II EDS について詳しくは、「Nios II ソフトウェア開発ハンドブック」を参照してください。
- MicroC/OS-II リアル・タイム・オペレーティング・システム (RTOS) MicroC/OS-II については、「Using MicroC/OS-II RTOS with the Nios II Processor Tutorial」を参照してください。

はじめに

アルテラは、Nios II EDS における NicheStack TCP/IP スタックの Nios II 実装 (ソース・コードを含む) を提供しています。NicheStack TCP/IP スタックは、Nios II プロセッサのためのイーサネット接続用スタックへの即時アクセスを提供します。アルテラの NicheStack TCP/IP スタック実装には API ラッパーが含まれており、定評のある標準ソケット API を提供します。

NicheStack TCP/IP スタックは MicroC/OS-II RTOS マルチスレッド環境を使用します。したがって、NicheStack TCP/IP スタックを Nios II EDS で使用するには、C/C++ プロジェクトを MicroC/OS-II RTOS ベースにする必要があります。当然、Nios II プロセッサ・システムがイーサネット・インタフェース、すなわちメディア・アクセス・コントローラ (MAC) を備えていることも必要です。アルテラが提供する NicheStack TCP/IP スタックには、SMSC lan91c111 MAC/PHY デバイス、およびアルテラのトリプル・スピード・イーサネット MegaCore ファンクションのドライバ・サポートが含まれています。Nios II エンベデッド・デザイン・スイートには、両方の MAC 用のハードウェア、およびトリプル・スピード・イーサネット MegaCore の評価コピーがあります。NicheStack TCP/IP スタック・ドライバは割り込みベースであり、イーサネット・コンポーネント用の割り込みが接続されていなければなりません。

アルテラが実装する NicheStack TCP/IP スタックは、HAL (Hardware Abstraction Layer) 汎用イーサネット・デバイス・モデルをベースにしています。汎用デバイス・モデルに基づいて、新しいドライバを記述してターゲット・イーサネット MAC をサポートしたり、一貫性のある HAL およびソケット API を維持して、ハードウェアにアクセスすることができます。



イーサネット・デバイス・ドライバの記述について詳しくは、「Nios II ソフトウェア開発ハンドブック」の「HAL 用デバイス・ドライバの開発」の章を参照してください。

NicheStack TCP/IP スタック・ファイルおよびディレクトリ

Nios II IDE を使用して、C/C++ プログラムでスタックを使用するのに NicheStack TCP/IP スタック・ソース・コードを編集する必要はありません。ただし、アルテラは参考のためにソース・コードを提供しています。デフォルトでは、ファイルは Nios II EDS とともに、<Nios II EDS インストール・パス >/components/altera_iniche/UCOSII ディレクトリにインストールされます。簡潔にするために、この章ではこのディレクトリを <iniche path> と呼んでいます。

NicheStack TCP/IP スタックを簡単により新しいバージョンへアップグレードするために、スタックのディレクトリ・フォーマットは、<iniche path>/src/downloads ディレクトリ下で、可能な限りオリジナル・コードを維持するよう試みます。<iniche path>/src/downloads/packages ディレクトリには、オリジナルの NicheStack TCP/IP スタック・ソース・コードと資料が収められ、<iniche path>/src/downloads/30src ディレクトリには、MicroC/OS-II をサポートするソース・コードを含む、NicheStack TCP/IP スタックの Nios II 実装に固有のコードが収められています。



NicheStack TCP/IP スタックの参考文献は、www.altera.co.jp/literature/lit-nio2.jsp のその他の関連資料で入手できます。

アルテラの NicheStack TCP/IP スタックの実装は、プロトコル・スタックのバージョン 3.0 をベースにしており、コードを HAL システム・ライブラリに統合するために、コードの周囲にラッパーが配置されています。

ライセンス

NicheStack TCP/IP スタックは InterNiche Technologies, Inc. が考案した TCP/IP プロトコル・スタックです。NicheStack TCP/IP スタックのライセンスは、www.altera.co.jp/nichestack から取得できます。



その他のプロトコル・スタックは、InterNiche 社から直接取得できます。詳しくは、InterNiche 社のウェブサイト (www.interniche.com) を参照してください。

その他の TCP/IP スタック・ プロバイダ



その他のサードパーティ・ベンダーも、Nios II プロセッサ向けイーサネット・サポートを提供しています。特にサードパーティ RTOS ベンダーは、自社の特定の RTOS フレームワーク用のイーサネット・モジュールを提供していることがよくあります。

サードパーティ・プロバイダから入手可能な製品に関する最新情報は、アルテラのエンベデッド・ソフトウェア・パートナーのページ (www.altera.co.jp/products/software/partners/embedded/emb-partners.html) をご覧ください。

NicheStack TCP/IP スタックの使用

この項では、NicheStack TCP/IP スタックを Nios II プログラムに組み込む方法について説明します。

NicheStack TCP/IP スタックへの一次インタフェースは、標準ソケット・インタフェースです。さらに、以下の関数を呼び出して、スタックとドライバを初期化します。

- `alt_iniche_init()`
- `netmain()`

初期化プロセスでは、グローバル変数 `iniche_net_ready` も使用します。

以下の単純な関数を提供する必要があります。これらの関数は HAL システム・コードで MAC アドレスと IP アドレスを取得するために呼び出します。

- `get_mac_addr()`
- `get_ip_addr()`

Nios II システム要件

NicheStack TCP/IP スタックを使用するには、Nios II システムが以下の要件を満たす必要があります。

- SOPC Builder で生成されるシステム・ハードウェアは、割り込みが可能なイーサネット・インタフェースを備えていること
- システム・ライブラリが MicroC/OS-II をベースにしていること
- MicroC/OS-II RTOS は以下がイネーブルされるようにコンフィギュレーションされていること
 - TimeManagement / OSTimeTickHook がイネーブルされていること
 - 最大タスク数が 4 以上であること

- システム・クロック・タイマが適切なタイマ・デバイスを指すように設定されていること

NicheStack TCP/IP スタックのタスク

NicheStack TCP/IP スタックは、標準 Nios II コンフィギュレーションでは、2つの基本的なタスクで構成されています。これらの各タスクが MicroC/OS-II スレッドのスタック用メモリの一部に加え、MicroC/OS-II スレッド・リソースを消費します。プログラムで作成されるタスクに加え、これらのタスクが継続的に実行されます。

1. NicheStack のメイン・タスク `tk_netmain()` — このタスクは、初期化後に新しいパケットが処理できるようになるまでスリープします。パケットは割り込みサービス・ルーチン (ISR) が受け取ります。ISR はパケットを受け取ると、それを受信キューに入れてメイン・タスクをウェイク・アップさせます。
2. NicheStack ティック・タスク `tk_nettick()` — このタスクは周期的にウェイク・アップして、タイムアウト条件をモニタします。

これらのタスクは、初期化プロセスで `netmain()` 関数が成功すると起動されます。この関数は [11-6 ページの「netmain\(\)」](#) で説明しています。



コンフィギュレーション・ファイル `ipport.h` の `#define` 文を使用して、タスクの優先順位とスタックのサイズを変更することができます。`ipport.h` を編集して NicheStack TCP/IP スタックのその他のオプションをイネーブルすると、追加のシステム・タスクが作成される場合があります。

スタックの初期化

スタックを初期化する前に、`main()` から `OSStart()` を呼び出して、MicroC/OS-II スケジューラを起動します。優先順位の高いタスクで、スタックの初期化を実行します。これは、RTOS が動作し、I/O ドライバが使用可能になるまで、コードがそれ以上初期化を試みないようにするためです。

スタックを初期化するには、関数 `alt_niche_init()` および `netmain()` を呼び出します。スタックの初期化が完了すると、グローバル変数 `iniche_net_ready` が `true` に設定されます。



`iniche_net_ready`が`true`に設定されるまで、コードがソケット・インタフェースを使用しないようにしてください。例えば、11-7 ページの例 11-1 に示すように、優先順位の最も高いタスクから `alt_iniche_init()` と `netmain()` を呼び出し、`iniche_net_ready` を待ってから、他のタスクの実行を許可します。

`alt_iniche_init()`

`alt_iniche_init()` は、MicroC/OS-II オペレーティング・システムで使用するスタックを初期化します。`alt_iniche_init()` のプロトタイプ:

```
void alt_iniche_init(void)
```

`alt_iniche_init()` は、戻り値を返さず、パラメータもありません。

`netmain()`

`netmain()` は、NicheStack タスクの初期化と起動を行います。`netmain()` のプロトタイプ:

```
void netmain(void)
```

`netmain()` は、戻り値を返さず、パラメータもありません。

`iniche_net_ready`

NicheStack スタックは初期化を完了すると、グローバル変数 `iniche_net_ready` を 0 以外の値に設定します。



`iniche_net_ready` が `true` になるまで、NicheStack API 関数（初期化用を除く）を呼び出さないでください。

例 11-1 に、ネットワーク・スタックが初期化を完了するまで待機するための、`iniche_net_ready` の使用を示します。

例 11-1. NicheStack TCP/IP スタックのインスタンス化

```
void SSSInitialTask(void *task_data)
{
    INT8U error_code;

    alt_iniche_init();
    netmain();

    while (!iniche_net_ready)
        TK_SLEEP(1);

    /* Now that the stack is running, perform the application
       initialization steps */

    .
    .
    .
}
```

マクロ `TK_SLEEP()` は、NicheStack TCP/IP スタックの OS ポーティング・レイヤの一部です。

`get_mac_addr()` および `get_ip_addr()`

NicheStack TCP/IP スタックのシステム・コードは、デバイス初期化プロセスで、`get_mac_addr()` と `get_ip_addr()` を呼び出します。これらの関数は、システム・コードでネットワーク・インタフェース用の MAC アドレスと IP アドレスを設定するのに必要です。これは、**Software Components** ダイアログ・ボックスの **NicheStack TCP/IP Stack** タブにある **MAC interface** で選択されます。これらの関数はユーザーが自分で記述するため、システムは MAC アドレスと IP アドレスをデバイス・ドライバにハードコードされた固定の場所ではなく、任意の場所に格納するための柔軟性を備えています。例えば、フラッシュ・メモリに MAC アドレスを格納するシステムもあれば、それをオンチップ・エンベデッド・メモリに持っているシステムもあります。

両方の関数とも、NicheStack TCP/IP スタックにより内部で使用されるデバイス構造をパラメータとして取り込みます。ただし、構造の詳細に関する知識は必要ありません。必要なのは、MAC アドレスと IP アドレスを書き込むことだけです。

`get_mac_addr()` のプロトタイプ:

```
int get_mac_addr(NET net, unsigned char mac_addr[6]);
```

この関数で、`mac_addr` に MAC アドレスを書き込む必要があります。

`get_mac_addr()` のプロトタイプは、ヘッダ・ファイル `<iniche path>/inc/alt_iniche_dev.h` にあります。NET 構造は、`<iniche path>/src/downloads/30src/h/net.h` ファイルに定義されています。

例 11-2 に、`get_mac_addr()` の実装を示します。この例ではデモ用として、MAC アドレスはアドレス `CUSTOM_MAC_ADDR` に格納されます。この例には、エラー・チェックはありません。実際のアプリケーションでは、エラーがない場合、`get_mac_addr()` は -1 を返します。

例 11-2. `get_mac_addr()` の実装

```
#include <alt_iniche_dev.h>
#include "includes.h"
#include "ipport.h"
#include "tcpport.h"
#include <io.h>
int get_mac_addr(NET net, unsigned char mac_addr[6])
{
    int ret_code = -1;

    /* Read the 6-byte MAC address from wherever it is stored */
    mac_addr[0] = IORD_8DIRECT(CUSTOM_MAC_ADDR, 4);
    mac_addr[1] = IORD_8DIRECT(CUSTOM_MAC_ADDR, 5);
    mac_addr[2] = IORD_8DIRECT(CUSTOM_MAC_ADDR, 6);
    mac_addr[3] = IORD_8DIRECT(CUSTOM_MAC_ADDR, 7);
    mac_addr[4] = IORD_8DIRECT(CUSTOM_MAC_ADDR, 8);
    mac_addr[5] = IORD_8DIRECT(CUSTOM_MAC_ADDR, 9);
    ret_code = ERR_OK;

    return ret_code;
}
```

関数 `get_ip_addr()` を記述して、プロトコル・スタックの IP アドレスを割り当てる必要があります。プログラムは、スタティック・アドレスを割り当てるもの、または DHCP を要求して IP アドレスを見つけるもののいずれでもかまいません。`get_ip_addr()` の関数プロトタイプ:

```
int get_ip_addr(alt_iniche_dev* p_dev,
                ip_addr*         ipaddr,
                ip_addr*         netmask,
                ip_addr*         gw,
                int*              use_dhcp);
```

`get_ip_addr()` は、リターン・パラメータを次のように設定します。

```
IP4_ADDR(ipaddr, IPADDR0, IPADDR1, IPADDR2, IPADDR3);
IP4_ADDR(gw, GWADDR0, GWADDR1, GWADDR2, GWADDR3);
IP4_ADDR(netmask, MSKADDR0, MSKADDR1, MSKADDR2, MSKADDR3);
```

ダミー変数 `IP_ADDR0-3` には、IP アドレスのバイト 0 ~ 3 の式を代入します。 `GWADDR0-3` には、ゲートウェイ・アドレスのバイトを代入します。 `MSKADDR0-3` には、ネットワーク・マスクのバイトを代入します。例えば、次のステートメントは、`ip_addr` を IP アドレス 137.57.136.2 に設定します。

```
IP4_ADDR ( ip_addr, 137, 57, 136, 2 );
```

DHCP をイネーブルするには、以下の行をインクルードします。

```
*use_dhcp = 1;
```

NicheStack TCP/IP スタックは、サーバから IP アドレスの取得を試みます。サーバが 30 秒以内に IP アドレスを供給しない場合、スタックはタイム・アウトして、`IP4_ADDR()` 関数呼び出しに指定されたデフォルト設定を使用します。

スタティック IP アドレスを割り当てるには、次の行をインクルードします。

```
*use_dhcp = 0;
```

`get_ip_addr()` のプロトタイプは、ヘッダ・ファイル `<iniche path>/incl alt_iniche_dev.h` にあります。

例 11-3 に、`get_ip_addr()` の実装と、必要なインクルード・ファイルのリストを示します。

この例には、エラー・チェックはありません。実際のアプリケーションでは、場合によってエラー時に -1 を返す必要があります。

例 11-3. get_ip_addr() の実装

```
#include <alt_iniche_dev.h>
#include "includes.h"
#include "ipport.h"
#include "tcpport.h"
int get_ip_addr(alt_iniche_dev *p_dev,
               ip_addr* ipaddr,
               ip_addr* netmask,
               ip_addr* gw,
               int*      use_dhcp)
{
    int ret_code = -1;
    /*
     * The name here is the device name defined in system.h
     */
    if (!strcmp(p_dev->name, "/dev/" INICHE_DEFAULT_IF))
    {
        /* The following is the default IP address if DHCP
         fails, or the static IP address if DHCP_CLIENT is
         undefined. */
        IP4_ADDR(&ipaddr, 10, 1, 1, 3);
        /* Assign the Default Gateway Address */
        IP4_ADDR(&gw, 10, 1, 1, 254);
        /* Assign the Netmask */
        IP4_ADDR(&netmask, 255, 255, 255, 0);

#ifdef DHCP_CLIENT
        *use_dhcp = 1;
#else
        *use_dhcp = 0;
#endif /* DHCP_CLIENT */

        ret_code = ERR_OK;
    }
    return ret_code;
}
```

`system.h` に定義されている `INICHE_DEFAULT_IF` は、ユーザーが SOPC Builder で定義したネットワーク・インタフェースを特定します。Nios II IDE では、**Software Components** ダイアログ・ボックスの **NicheStack TCP/IP Stack** タブの **MAC interface** コントロールで、`INICHE_DEFAULT_IF` を設定できます。Nios II BSP ジェネレータでは、`iniche_default_if` BSP 設定を使用します。

同様に `system.h` に定義されている `DHCP_CLIENT` は、DHCP クライアント・アプリケーションを使用して IP アドレスを取得するか否かを指定します。この設定は、Nios II IDE で (**Use DHCP to automatically assign IP address** チェック・ボックスを使用)、または Nios II BSP ジェネレータを介して (`dhcp_client` 設定による)、設定またはクリアすることができます。

ソケット・インタフェースの呼び出し

イーサネット・デバイスの初期化後、プログラムの残りの部分でソケット API を使用して、IP スタックにアクセスします。

ソケット API を使用して IP スタックと通信する新しいタスクを作成するには、関数 `TK_NEWTASK()` を使用する必要があります。`TK_NEWTASK()` 関数は、NicheStack TCP/IP スタックの OS ポーティング・レイヤの一部です。`TK_NEWTASK()` は、`MicroC/OS-II OSTaskCreate()` 関数を呼び出してスレッドを作成し、NicheStack TCP/IP スタックに固有のその他の処理をいくつか実行します。

`TK_NEWTASK()` のプロトタイプ:

```
int TK_NEWTASK(struct inet_task_info* nettask);
```

これは `<niche path>/src/downloads/30src/nios2/osport.h` にあります。このヘッダ・ファイルは、次のようにインクルードすることができます。

```
#include "osport.h"
```

OS ポーティング・レイヤのその他の詳細は、NicheStack TCP/IP Stack コンポーネント・ディレクトリ `<niche path>/src/downloads/30src/nios2/` の `osport.c` ファイルにあります。



アプリケーションで `TK_NEWTASK()` を使用方法について詳しくは、「Using the NicheStack® TCP/IP Stack - Nios II Edition Tutorial」を参照してください。

Nios II IDE での NicheStack TCP/IP スタックの コンフィギュ レーション

NicheStack TCP/IP スタックは多数のオプションを備えており、これらは `ippport.h` ファイルの `#define` ディレクティブを使用してコンフィギュレーションできます。Nios II 統合開発環境 (IDE) により、ソース・コードを編集しないで特定のオプションをコンフィギュレーションする (すなわち、`system.h` の `#defines` を変更する) ことができます。最もよくアクセスされるオプションは、**Software Components** ダイアログ・ボックスの **NicheStack TCP/IP Stack** タブで使用できます。

IDE からアクセスできない、使用頻度の低いオプションもいくつかあります。これらのオプションを変更する必要がある場合は、Nios II BSP ジェネレータを使用するか、または `ippport.h` ファイルを手動で編集する必要があります。



Nios II BSP ジェネレータについて詳しくは、「Nios II ソフトウェア開発ハンドブック」の「Nios II Software Build Tools」の章を参照してください。

`ippport.h` は、システム・ライブラリ・プロジェクトの `debug/system_description` ディレクトリで見つけることができます。



`ippport.h` ファイルを直接変更する場合は、Nios II IDE の **Clean Project** ビルド・オプションを選択しないように注意してください。**Clean Project** を選択すると、変更済みの `ippport.h` ファイルがこのファイルの起動テンプレート・バージョンに置き換えられます。

以下の項では、Nios II IDE を介してコンフィギュレーション可能な機能について説明します。IDE は各機能にデフォルト値を与えます。一般に、これらの値は適切な開始点であり、システムのニーズに合わせて後で微調整することができます。

NicheStack TCP/IP スタックの一般的な設定

ARP、UDP、および IP プロトコルは常にイネーブルされています。表 11-1 にプロトコル・オプションを示します。

オプション	説明
TCP	TCP (Transmission Control Protocol) をイネーブルまたはディセーブルします。

表 11-2 に、TCP/IP スタックの全体的な動作に影響を与えるグローバル・オプションを示します。

オプション	説明
Use DHCP to automatically assign IP address	オンのとき、コンポーネントは DHCP を使用して IP アドレスを取得します。オフのとき、スタティック IP アドレスを割り当てる必要があります。
Enable statistics	このオプションをオンにすると、スタックは受信パケット、エラーなどのカウンタを保持します。カウンタは、 <code><niche path>/src/downloads/30src/h</code> ディレクトリの各種ヘッダ・ファイルに定義されている mib 構造内に定義されています。mib 構造について詳しくは、NicheStack の資料を参照してください。
MAC interface	IP スタックが複数のネットワーク・インタフェースを備えている場合、このパラメータはどのインタフェースを使用するかを示します。11-14 ページの「 既知の制限 」を参照してください。

IP オプション

表 11-4 に IP オプションを示します。

オプション	説明
Forward IP packets	複数のネットワーク・インタフェースがあるときに、このオプションがオンになっている場合、あるインタフェース用の IP スタックが他のスタックに宛てられたパケットを受信すると、その IP スタックは他のインタフェースからのパケットを転送します。11-14 ページの「既知の制限」を参照してください。
Reassemble IP packet fragments	このオプションがオンになった場合、NicheStack TCP/IP スタックは IP パケットのフラグメントを完全な IP パケットに再アセンブルします。オプションがオフの場合は、IP パケットのフラグメントを破棄します。これについては、Richard Stevens 著の <i>Unix Network Programming</i> で説明されています。

TCP オプション

表 11-4 に、TCP オプションがオンの場合にのみ使用可能な TCP オプションを示します。

オプション	説明
Use TCP zero copy	このオプションは、NicheStack ゼロ・コピー TCP API をイネーブルします。このオプションにより、NicheStack TCP/IP スタックを使用するときに、バッファ間コピーが不要になります。詳しくは、NicheStack リファレンス・マニュアルを参照してください。アプリケーション・コードを変更して、ゼロ・コピー API を活用する必要があります。

詳細情報 について

アルテラの NicheStack の実装について詳しくは、「Using the NicheStack® TCP/IP Stack - Nios II Edition Tutorial」を参照してください。このチュートリアルでは、NicheStack TCP/IP スタックに関する詳細情報を提供し、ネットワーキング・アプリケーションでの使用方法を説明しています。

NicheStack について詳しくは、NicheStack TCP/IP スタックのリファレンス・マニュアルを参照してください。この資料は www.altera.co.jp/literature/lit-nio2.jsp のその他の関連資料から入手できます。

既知の制限

NicheStack コードには複数のネットワーク・インタフェースをサポートするための機能が含まれていますが、これらの機能はテストされていません。複数のネットワーク・インタフェースのサポートについては、NicheStack TCP/IP スタックのリファレンス・マニュアルおよびソース・コードを参照してください。

参考資料

この章では以下のドキュメントを参照しています。

- 「HAL 用デバイス・ドライバの開発」の章（Nios II ソフトウェア開発ハンドブック）
- 「NicheStack TCP/IP Stack documentation」（オンライン資料：Nios II プロセッサ、その他の関連資料）
- 「Using the NicheStack TCP/IP Stack - Nios II Edition Tutorial」
- 「Nios II Software Build Tools」の章（Nios II ソフトウェア開発ハンドブック）

改訂履歴

表 11-5 に、本資料の改訂履歴を示します。

表 11-5. 改訂履歴		
日付およびドキュメント・バージョン	変更内容	概要
2008 年 5 月 v8.0.0	前バージョンからの内容の変更はありません。	
2007 年 10 月 v7.2.0	前バージョンからの内容の変更はありません。	
2007 年 5 月 v7.1.0	<ul style="list-style-type: none"> ● 9 章から 10 章に変更。 ● マイナーな内容の明確化を内容に追加。 ● 「概要」の項に目次を追加。 ● 「参考資料」の項を追加。 	
2007 年 3 月 v7.0.0	前バージョンからの内容の変更はありません。	
2006 年 11 月 v6.1.0	初版	