

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。

2007年11月 ver.1.0

Application Note 458

はじめに

マイクロプロセッサを内蔵するどのスタンドアロン・エンベデッド・システムでも、システムがリセットした後、プロセッサはブート・コピアまたはブート・ローダと呼ばれる小さなコードを実行します。ブート・コピアは、不揮発メモリ内の適切なアプリケーション・ソフトウェアの場所を示し、それを RAM にコピーして、重要なシステム・コンポーネントを初期化し、アプリケーション・ソフトウェアのエントリ・ポイントに分岐します。不揮発メモリでのアプリケーション・ソフトウェアを含むデータ・ブロックは、一般にブート・イメージと呼ばれています。ブート・コピアは、バイト単位のコピー・ルーチンから、厳密なシステム・テストを実行し、複数のソフトウェア・アプリケーションの間で選択を行い、適切なアプリケーションをアンパックして、解凍し、該当するアプリケーション上でエラー検出を実行する包括的なアプリケーションまで、その規模はさまざまです。

本資料では、Nios®II プロセッサおよび Nios II 統合開発環境 (IDE) を使って、独自のカスタム・ブート・コピア・ソフトウェアを実装する方法について説明します。本資料ではさらに、Nios II のブート・プロセスを外部から制御するのに必要な基本情報も提供します。

本資料は、FPGA に内蔵されたコンフィギュレーション済みの Nios II プロセッサ用のカスタム・ブート・コピアの実装方法を説明しています。アルテラ FPGA のカスタム・コンフィギュレーション方法については、説明していません。



アルテラ FPGA のカスタム・コンフィギュレーション方法については、www.altera.co.jp/support/devices/configuration/cfg-index.html を参照してください。

読者の前提条件

本資料は、読者が Nios II の上級ユーザーであり、エンベデッド・ソフトウェアを問題なく読み書きできることを前提にしています。Nios II ハードウェアまたはソフトウェアの開発フローに精通していない場合には、まず Nios II マイクロプロセッサ・システムの構築を十分に理解することが必要です。



Nios II マイクロプロセッサ・システム例を構築するためのステップごとの手順については、「[Nios II ハードウェア開発チュートリアル](#)」を参照してください。

また、本資料は読者が IDE と Nios II Flash Programmer のコマンド・ライン操作の両方に精通していることも前提にしています。



Nios II Flash Programmer について詳しくは、「[Nios II フラッシュ・プログラマ・ユーザーガイド](#)」を参照してください。

カスタム・ブート・コピアの実装

カスタム・ブート・コピアを実装するには、通常の Nios II IDE ソフトウェア開発フローから逸脱する必要が生じます。ソース・ファイルを手で編集し、Nios II Command Shell からファイル変換ユーティリティを実行する必要があります。通常の Nios II IDE ソフトウェア・フローで作業したい場合は、デフォルト・ブート・コピアに制限されます。

本資料には、以下のタイプのカスタム・ブート・コピア用のソース・コードのサンプルが含まれています。

- 高度なブート・コピア—この例には、デュアル・ブート・イメージのサポートや CRC エラー・チェックなどの追加機能が含まれています。
- 小さなブート・コピア—この例は、メモリ・スペースをほとんど必要としない必要最小限のブート・コピアです。

デフォルトの Nios II ブート・コピア

この項では、デフォルトの Nios II ブート・コピアの動作を説明し、さらにコモン・フラッシュ・インタフェース (CFI) 対応フラッシュ・メモリおよびアルテラの EPCS (Erasable Programmable Configurable Serial) タイプの動作について説明します。デフォルト・ブート・コピアについて熟知していない読者は、カスタム・ブート・コピアを実装する前にこの項を読むことを推奨します。

デフォルトの Nios II ブート・コピアの概要

Nios II プロセッサに付属するデフォルトのブート・コピアは、大多数の Nios II アプリケーションにとって十分な機能を提供し、Nios II IDE で容易に実装できます。デフォルト・ブート・コピアは、アプリケーションがフラッシュ・メモリにプログラムされると、Nios II IDE によって自動的かつトランスペアレントにシステムに追加されます。



アルテラでは、異なる機能または追加機能を備えたカスタム・ブート・コピアが必要ない場合は、デフォルトの Nios II ブート・コピアを使用することを推奨します。カスタム・ブート・コピアを実装すると、ソフトウェア構築プロセスが複雑になり、アルテラからのテクニカル・サポートの提供に支障を来す可能性があります。

デフォルトの Nios II ブート・コピアは、以下の機能を備えています。

- CFI または EPCS フラッシュ・メモリをサポート
- ブート・イメージをアンパックして RAM にコピー
- RAM のアプリケーション・コードに自動的に分岐

デフォルトの CFI 対応フラッシュ・ブート・コピア

Nios II デフォルト・ブート・コピアは、**elf2flash** ユーティリティの実行中に、Nios II Flash Programmerによって自動的にインクルードされます。**elf2flash** ユーティリティは、プロセッサのリセット・アドレスに基づき、ブート・コピアが必要であるかどうかにかかわらず、アプリケーション・コードのエントリ・ポイントとフラッシュ・メモリのアドレス範囲を決定します。プロセッサのリセット・アドレスが CFI フラッシュ・メモリを指し、かつアプリケーションの .text セクションが CFI フラッシュ・メモリ以外の場所を指している場合は、必ず CFI ブート・コピアが必要です。ブート・コピアが必要な場合、**elf2flash** はアプリケーション・コードをブート・レコードにバックし、Motorola S レコード (.flash) ファイルを作成してブート・コピアとブート・レコードを格納します。Flash Programmer は、このブート・レコードを CFI フラッシュ・メモリにダウンロードします。

Nios II プロセッサがリセットを完了した直後に、ブート・コピアが実行され、6 ページの「ブート・イメージ」で説明するとおりブート・レコードを読み出して、アプリケーション・コードを RAM にコピーします。コピーの完了後、ブート・コピアは、ブート・レコードからアプリケーション・コードのエントリ・ポイントを読み出します。ブート・コピアがそのアドレスへのジャンプを実行すると、アプリケーション・ソフトウェアの実行が開始されます。

デフォルトの EPCS ブート・コピア

Nios II プロセッサのリセット・アドレスが SOPC Builder の EPCS Controller のベース・アドレスに設定されている場合、デフォルトの EPCS ブート・コピアが実装されます。EPCS Controller は、そのベース・アドレスにマップされたオンチップ・メモリの小さなブロックで Nios II プロセッサのブート・シーケンスをサポートします。EPCS ブート・コピアは、Quartus II のコンパイル中に、このオンチップ・メモリの最初のコンテンツとして指定されます。EPCS からブートする場合、**elf2flash** ユーティリティは、ブート・コピアを .flash ファイルにインクルードしません。その代わりに、ブート・レコードにパッケージされたアプリケーション・コードをインクルードします。Flash Programmer はデータをダウンロードし、それをオンチップ・メモリに配置された EPCS ブート・コピアを読み出します。

Nios II プロセッサがリセットを完了した直後に、EPCS Controller のオンチップ・メモリ・ブロックからブート・コピアが実行されます。EPCS ブート・コピアは、最初に、FPGA コンフィギュレーション・イメージ (プログラマ・オブジェクト・ファイル、すなわち .pof) が EPCS デバイスの先頭に配置されているかどうかをチェックします。このファイルを検出すると、EPCS ブート・コピアは、.pof ファイルのヘッダを読み出して、FPGA コンフィギュレーション・イメージのサイズを確定します。次にブート・コピアは、FPGA コンフィギュレーション・イメージの最終バイトの直後にある EPCS オフセットで、ソフトウェア・アプリケーション・ブート・レコードを探します。ブート・レコードが検出された場合、ブート・コピアはそれを読み出して、アプリケーション・コードを RAM にコピーします。コピーの完了後、ブート・コピアは、ブート・レコードからアプリケーション・コードのエントリ・ポイントを読み出します。ブート・コピアがそのアドレスへのジャンプを実行すると、アプリケーション・ソフトウェアの実行が開始されます。

デフォルト・ブート・コピアのソース・コードは、`<Nios II EDS install path>/components/altera_nios2/boot_loader_sources`ディレクトリに格納された、Nios II エンベデッド・デザイン・スイート (EDS) に付属し、また Nios II 資料ページ (www.altera.co.jp/literature/lit-nio2.jsp) に本資料と併せて掲載されています。

高度なブート・コピア例

この項では、高度なブート・コピア例を説明します。この例は、CFI フラッシュメモリまたはオンチップ・メモリのいずれかから実行するように、また CFI または EPCS フラッシュ・デバイスに格納されたブート・イメージをサポートするように構築できます。この例は C 言語で記述され、多くのコメントが含まれているため、容易にカスタマイズできます。この例には、デフォルト・ブート・コピアが提供する機能に加え、以下の機能を備えています。

- 2つの個別ブート・イメージをサポート
- JTAG UART を使用するステータス・メッセージをサポート
- ブート・イメージ・データでエラー・チェックを実行
- ワードアラインされていないブート・イメージをサポート



デザイン・ファイルへのハイパーリンクは、Nios II 資料ページで本資料の下に記載されています。 www.altera.co.jp/literature/lit-nio2.jsp をご覧ください。

ドライバの初期化

高度なブート・コピア例では、メモリ要件を低く抑えるために、それ自身の機能をサポートするのに最低限必要なドライバ初期化のみ実行します。デフォルトでは、以下のドライバが初期化されます。

- システム・クロック・タイマ
- JTAG UART
- プロセッサ割り込みハンドラ

ブート・コピアは、これらのドライバの初期化を完了した後、RAM 内の主要アプリケーション・コードに分岐し、このコードによりシステム・ドライバの完全な初期化が実行されます。

ユーザーは、ブート中にこれらのコンポーネントが不要であると判断した場合、各ドライバの初期化を個別にディセーブルにして、コード・サイズを低減できます。

JTAG UART への出力

この例のブート・コピアは、ブート・プロセス中に JTAG UART ペリフェラルへ情報を出力します。出力は、ブート・コピアのデバッグやシステムのブート状態の監視に役立ちます。デフォルトでは、スタートアップ・メッセージ、ブート・コピアがブート・イメージを検索しているフラッシュ・メモリ内のアドレス、ブート・コピアが最終的に選択してブートするイメージの指示などの基本情報が出力されます。独自の出力メッセージをコードに簡単に追加できます。

高度なブート・コピア例は、以下の理由により、`printf()` ライブラリ関数の使用を避けています。

- `printf()` ライブラリによって、JTAG UART からの出力を読み出すホストが存在しない場合に、ブート・コピアが停止することがあります。
- `printf()` ライブラリは、潜在的に大量のプログラム・メモリを消費する可能性があります。

JTAG UART による停止の防止

JTAG UART は従来の UART とは動作が異なります。従来の UART は、一般に外部ホストがリスンしているかどうかにかかわらず、シリアル・データを無分別に送信します。シリアル・データを読み出すホストがない場合、データが失われます。他方、JTAG UART は、送信データを出力バッファに書き込み、バッファからデータを読み出してバッファを空にする操作は外部ホストに依存します。デフォルトでは、出力バッファが満杯になると JTAG UART ドライバが停止します。ドライバは、外部ホストが出力バッファからデータを読み出すのを待ってから、追加の送信データを書き込みます。このプロセスにより、送信データの喪失が防止されます。

ただし、ブート時に JTAG UART にホストが接続されていない可能性はあります。この場合、JTAG UART 出力バッファから読み出される送信データはありません。出力バッファが充填されると、`printf()` 関数により、プログラム全体が停止します。ただし、ブート・コピアは、外部ホストが JTAG UART に接続済みであるかどうかにかかわらず、継続してシステムを立ち上げる必要があります。

この問題を回避するために、高度なブート・コピア例は、`my_jtag_write()` と呼ばれる独自のプリント・ルーチンを実装します。このルーチンにはユーザーが調整可能なタイムアウト機能が含まれており、JTAG UART は限られたタイムアウト期間にわたってプログラムを停止することができます。タイムアウト期間の経過後、プログラムは JTAG UART にそれ以上出力することなく継続して実行されます。このルーチンを `printf()` の代わりに使用すると、JTAG UART に接続されたホストがない場合にブート・コピアが停止するのを防止します。

出力へのメモリ使用の低減

高度なブート・コピア例では、JTAG UART の出力をすべてディセーブルにすることもできます。これによって、ブート・コピアのコード・サイズを大幅に削減できます。この例で JTAG UART 出力をディセーブルにするには、以下のステップに従います。

1. **advanced_boot_copier.c** ファイルで、次の行を見つけます。
`#define USING_JTAG_UART 1`
2. この行を次の行に置き換えます。
`#define USING_JTAG_UART 0`

このコード変更により、ブート時に JTAG UART がディセーブルにされ、ブート・コピアのメモリ要件が低減されます。

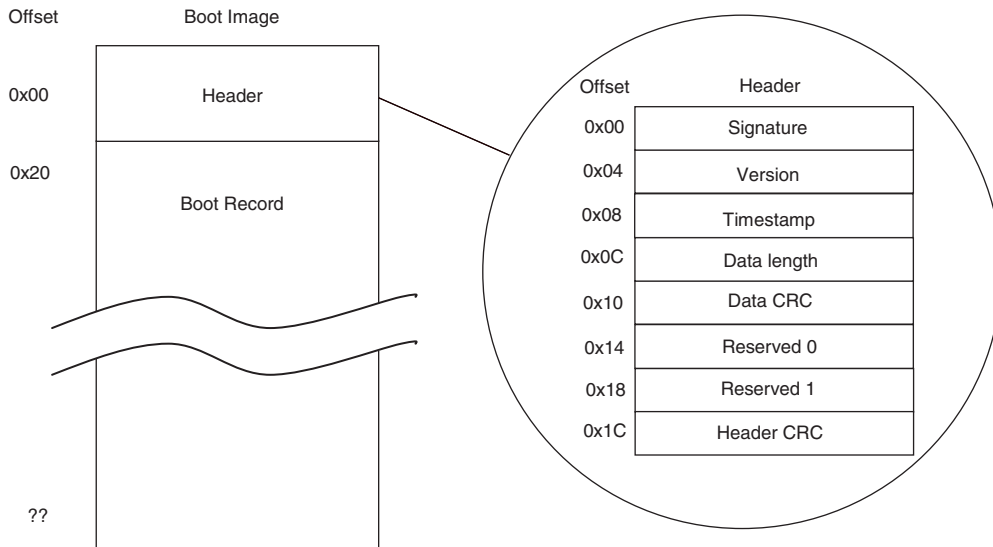
ブート・イメージ

高度なブート・コピア例は、特定のフォーマットに適合するブート・イメージを検出することを想定し、フラッシュ・メモリに格納されたブート・イメージを最大 2 つまでサポートします。この例は、両方のイメージがフラッシュの 32 ビット・データ境界から開始することを想定していません。

ブート・イメージのフォーマット

高度なブート・コピア例は、特定のフォーマットに適合するブート・イメージを検出することを想定しています。`make_flash_image_script.sh` スクリプトは、想定したフォーマットに適合するブート・イメージを作成します。`make_flash_image_script.sh` スクリプトは、`elf2flash` ユーティリティを実行して、`.elf` ファイルからアプリケーションのブート・レコードを作成し、そのブート・レコードにヘッダ情報を付加します。図 1 に、`make_flash_image_script.sh` スクリプトを使用して作成されたブート・イメージのメモリ・マップを示します。

図 1. ブート・イメージ例のメモリ・マップ



ブート・イメージのヘッダ・フォーマット

各ブート・イメージは、オフセット 0x0 にヘッダを含んでいます。ブート・コピア例は、各ブート・イメージに添付されたヘッダ情報を使用して、イメージに関する情報を抽出し、ブート・プロセス中にさまざまな判断を行います。**make_flash_image_script.sh** コマンド・シェル・スクリプトは、どのブート・イメージにも自動的にヘッダ情報を追加します。表 1 に、ブート・イメージ・ヘッダに含まれる情報を示します。

表 1. ブート・イメージのヘッダ・フォーマット例 (1 / 2)

フィールド	説明
署名	32 ビット フラッシュ・メモリ内のヘッダの位置を特定するために使用される署名。 make_flash_image_script.sh でこの値を変更します。 デフォルトのブート署名は、0xa5a5a5a5 です。
バージョン	32 ビット アプリケーションのバイナリ・エンコードされたバージョン識別子 make_flash_image_script.sh でこの値を変更します。
タイムスタンプ	32 ビット ヘッダが作成された時間 C 言語の標準時間整数値 (1970 年 1 月 1 日からの経過秒数) を使用します。 make_flash_image_script.sh により生成されます。
データ長	32 ビット ブート・イメージに含まれるアプリケーション・データの長さ (バイト数) make_flash_image_script.sh により生成されます。
データ CRC	32 ビット アプリケーション・データ全体に対する CRC32 値 make_flash_image_script.sh により生成されます。
Reserved 0	32 ビット 不特定用途 make_flash_image_script.sh でこの値を変更します。

表 1. ブート・イメージのヘッダ・フォーマット例 (2 / 2)	
フィールド	説明
Reserved 1	32 ビット 不特定用途 make_flash_image_script.sh でこの値を変更します。
ヘッダ CRC	32 ビット ヘッダ・データに対する CRC32 値 make_flash_image_script.sh により生成されます。

ブート・レコードのフォーマット

ブート・レコードは、ブート・イメージ・ヘッダの直後に続きます。ブート・レコードは、ブート・コピーによってロードされるアプリケーションを表現したものです。ブート・レコードには、アプリケーションの各コード・セクションの個別レコードが含まれています。コード・セクションは、メモリ内の固有の領域にリンクされる連続したコード部分です。ブート・コピーは、ブート・レコードを読み出して、アプリケーション・ソフトウェア・コードの各セクションのデスティネーション・アドレスを決定し、適切なコピー動作を実行します。

ソフトウェア・アプリケーションのコード・セクションはメモリ内の連続したロケーションにリンクされていない場合があるので、ブート・レコードが必要です。アプリケーションのコード・セクションは、メモリ・マップのいたるところに存在することがよくあります。アプリケーションをブートするには、フラッシュ・メモリにアプリケーション全体とアプリケーションの各部分をメモリのどの場所にコピーすべきかの情報が含まれている必要があります。ただし、フラッシュ・メモリはメモリよりも容量が小さいので、この表現をメモリ・ホールに含めることはできません。

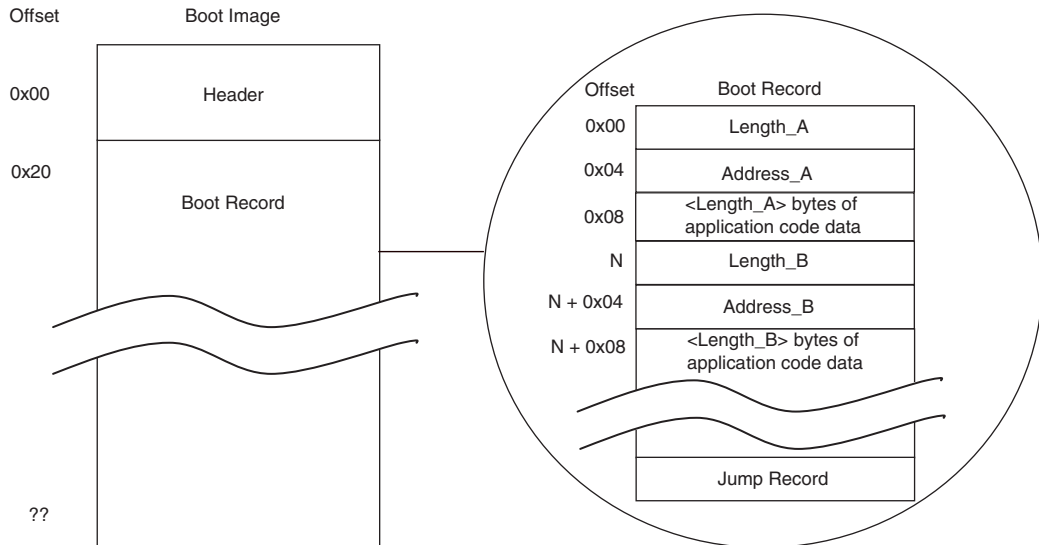
ブート・レコードには、RAM 内でのターゲットの位置にかかわらず、連続したデータ・ブロックにあるソフトウェア・アプリケーションのすべてのコード・セクションが含まれます。ブート・レコードは個別レコードのシーケンスであり、それぞれにコード・セクションのデータが含まれ、その前にデスティネーション・アドレスと長さが付加されています。ブート中に、ブート・コピーはブート・レコードからデスティネーション・アドレス (<destination address>) と長さ (<numbytes>) を読み出し、ブート・レコードの後続の <numbytes> バイトを <destination address> にコピーします。

ブート・レコードの最後の個別レコードは通常、特殊なジャンプ・レコードです。このレコードが読み出されると、アプリケーション・コードのコピーが完了し、後続の4バイトに格納された32ビット・アドレスへジャンプする必要があることがブート・コピアに通知されます。このアドレスはアプリケーションのエントリ・ポイントです。ジャンプ・レコードは、常に0x00000000としてエンコードされます。

3番目のタイプの個別レコードは停止レコードです。停止レコードはブート・コピアに実行停止を指示します。停止レコードは0xFFFFFFFFとしてエンコードされます。ブート・コピアは、フラッシュの消去済み領域を検出すると、それを停止レコードと解釈して、実行を停止します。

図2に、ブート・レコード例のメモリ・マップを示します。

図2. ブート・レコード例のメモリ・マップ



ブート・イメージの選択

高度なブート・コピア例は、フラッシュ・メモリに格納されたブート・イメージを2つまでサポートします。ブート・コピアは、フラッシュ・メモリ内の2つのロケーションを検査し、各ロケーションで有効なブート・イメージを検索し、それらのイメージの1つを選択してRAMにコピーし、実行します。2つのロケーションは、事前にロケーション番号1および2として指定されます。ブート・イメージを選択するために、ブート・コピアは以下の基準を記載順に使用します。

- イメージの有効性
 - 有効なブート・イメージが1つしか見つからない場合、ブート・コピアはそのイメージを使用してブートします。
 - 有効なブート・イメージが見つからない場合、ブート・コピアは5秒間待機した後、Nios II リセット・アドレスにジャンプして戻ります。
- リビジョン番号
 - 両方のブート・イメージが有効な場合、ブート・コピアは各イメージのリビジョン番号を調べます。
 - ブート・コピアは、バージョン番号が最も大きいブート・イメージを選択します。
- タイムスタンプ
 - 両方のブート・イメージのバージョン番号が同じ場合、ブート・コピアは各イメージのタイムスタンプを調べます。
 - ブート・コピアは、最新のタイムスタンプを持つブート・イメージを選択します。
- デフォルト
 - 両方のブート・イメージに同じタイムスタンプがある場合、ブート・コピアはロケーション番号2のイメージを選択します。

ワード・アラインメント

ほとんどの場合、Nios II ブート・イメージはフラッシュ・メモリの 32 ビット・データ境界からプログラムされます。この配置により、ブート・コピアは、32 ビット・ワード転送を使用してアプリケーション・データをコピーすることができます。ただし、高度なブート・コピア例では、このアラインメントを想定していません。ブート・コピアは、フラッシュ・メモリ内で 32 ビットでワード・アラインメントされていない有効なブート・イメージを見つけた場合でも、アプリケーションを正確に RAM にコピーすることができます。ブート・コピアは、`memcpy()` ライブラリ関数を使用してコピーを実行します。`memcpy()` 関数はメモリを殆ど必要とせず、`memcpy()` の使用は、データをメモリ内のアラインメントに関係なく高速で信頼性の高いコピー方法です。

ブート方法

高度なブート・コピア例は、以下のブート方法をサポートします。

- CFI フラッシュからの直接ブート
- オンチップ・メモリから実行される CFI フラッシュからのブート
- オンチップ・メモリから実行される EPCS フラッシュからのブート

CFI フラッシュからの直接ブート

この方法では、Nios II リセット・アドレスは CFI フラッシュ・メモリのベース・アドレスに設定されます。次に、ブート・コピアがフラッシュ内のそのアドレスにプログラムされるため、Nios II プロセッサがリセットされるとブート・コピアの実行が開始されます。ブート・コピアは、CFI フラッシュから RAM にアプリケーションをコピーし、次にアプリケーションのエントリ・ポイントへ分岐します。

オンチップ・メモリから実行される CFI フラッシュからのブート

この方法では、Nios II リセット・アドレスは、FPGA オンチップ・メモリとして実装されたブート ROM のベース・アドレスに設定されます。ハードウェア・デザインが Quartus II ソフトウェアでコンパイルされた後、FPGA がコンフィギュレーションされると、ブート・コピアの実行ファイルがブート ROM にロードされます。Nios II プロセッサがリセットされると、ブート・コピアの実行が開始されます。ブート・コピアは、アプリケーション・コードを CFI フラッシュから RAM にコピーし、次にアプリケーションのエントリ・ポイントへ分岐します。

オンチップ・メモリから実行される EPCS フラッシュからのブート

この方法は、前述の方法と非常によく似ています。違いは、ブート・イメージが CFI フラッシュではなく、EPCS フラッシュに格納されていることです。この方法でも、Nios II リセット・アドレスは FPGA オンチップ・メモリとして実装されたブート ROM のベース・アドレスに設定されます。ハードウェア・デザインが Quartus II ソフトウェアでコンパイルされた後、FPGA がコンフィギュレーションされると、ブート・コピアの実行ファイルがブート ROM にロードされます。Nios II プロセッサがリセットされると、ブート・コピアの実行が開始されます。ブート・コピアは、アプリケーション・コードを EPCS フラッシュから RAM にコピーし、次にアプリケーションのエントリ・ポイントへ分岐します。

使用するブート方法の選択

高度なブート・コピア例は、前述の 3 つのブート方法をすべてサポートします。**advanced_boot_copier.c** ファイルの次の行は、実装された方法を制御します。

```
#define BOOT_METHOD <boot method>
```

<boot method> に使用できるオプションは、以下のとおりです。

- BOOT_FROM_CFI_FLASH
- BOOT_CFI_FROM_ONCHIP_ROM
- BOOT_EPCS_FROM_ONCHIP_ROM

フラッシュ内でのデータ・オーバーラップの防止

フラッシュ・メモリからブートするようにシステムを設定した場合は、そのフラッシュに格納されている他のデータにも配慮する必要があります。Nios 開発ボードは、CFI または EPCS のいずれかのタイプのフラッシュに、FPGA コンフィギュレーション・イメージとソフトウェア・ブート・イメージを一緒に格納できるように設計されています。フラッシュに複数のイメージを格納する場合は、イメージが互いにオーバーラップしないようにする必要があります。

CFI フラッシュでのデータのオーバーラップ

Nios 開発ボードは一般に、CFI フラッシュ内の上位オフセットを FPGA イメージ用に指定し、下位オフセットをソフトウェア・ブート・イメージおよびその他のデータに使用できるようにします。**nios2-elf-size** ユーティリティを使用して、各フラッシュ・イメージのサイズを計算し、それらのサイズ（または推定される将来のサイズ）に基づいて、イメージがオーバーラップしないように各イメージに対するフラッシュ内のオフセットを選択します。例えば、Nios 開発ボード Cyclone II Edition のコンフィギュレーション・コントローラは、CFI フラッシュのオフセット 0x00C00000 ~ 0x00CFFFFFF をユーザー FPGA コンフィギュレーション・イメージとして指定します。そのオフセット範囲にブート・イメージ・データが配置されると、1つのイメージが他のイメージを破壊します。

EPCS フラッシュでのデータのオーバーラップ

EPCS フラッシュでは、FPGA コンフィギュレーション・イメージが常にオフセット 0x0 から始まる必要があります。ブート・イメージが FPGA コンフィギュレーション・イメージの先頭にプログラムされないようにするために、FPGA コンフィギュレーション・イメージのエンド・オフセットを決定する必要があります。**sof2flash** ユーティリティを使用して、FPGA コンフィギュレーション・イメージ .sof ファイルを .flash イメージに変換し、そのフラッシュ・イメージ上で **nios2-elf-size** を実行します。その結果、EPCS フラッシュ内の FPGA コンフィギュレーション・イメージ最後のオフセットが求められます。EPCS フラッシュにプログラムされるどのソフトウェア・ブート・イメージも、FPGA コンフィギュレーション・イメージの最後を超えるオフセットから始まることを確認してください。

ブート・コピアのコード・サイズ

高度なブート・コピア例は、変更なしで約 6500 バイトのサイズの実行ファイルにコンパイルされます。JTAG UART とシステム・クロック・タイマ機能をすべてオフにした場合、実行ファイル例のサイズは約 2000 バイトに減ります。

ちなみに、デフォルトのブート・コピアのコード・サイズは、2 ページの「デフォルトの Nios II ブート・コピア」に示すとおり、CFI フラッシュからブートするようにコンパイルされた場合は約 200 バイト、EPCS フラッシュからブートするようにコンパイルされた場合は約 500 バイトです。

2000 バイト以下のカスタマイズ可能なブート・コピアが必要な場合は、24 ページの「小さなブート・コピアの例」を参照してください。小さなブート・コピアは Nios II アセンブリ言語で記述され、備えている機能はごくわずかです。このブート・コピアは、実行されると、CFI フラッシュ内のブート・レコードを RAM にコピーし、コピーされたアプリケーションへ分岐するだけです。コンパイルされたコード・サイズは約 200 バイトです。

高度なブート・コピア例の実装

この項では、Nios 開発ボード上で高度なブート・コピア例を構築および実行するのに必要なステップについて説明します。

ソフトウェア・ツールおよび開発ボードのセットアップ

高度なブート・コピア例を構築および実行するには、以下のステップに従います。

1. コンピュータに Nios II EDS バージョン 7.2 (またはそれ以降) および Quartus II バージョン 7.2 (またはそれ以降) がインストールされていることを確認します。
2. 電源および USB-Blaster を Nios II 開発ボードに接続します。

適切なハードウェア・デザインの作成

以下のステップでは、高度なブート・コピア例を実行できる Nios II システムを開いて、変更し、生成します。また、どのブート方法を実装するかを決める必要があります。以下のステップのいくつかでは、使用するブート方法に応じて、若干異なる処置を取る必要があります。

プロジェクト例を開くには、

1. Verilog HDL または VHDL で記述された、Nios 開発ボード用の Nios II 標準デザイン例を見つけます。標準デザイン例は、<Nios II EDS install path>/examples ディレクトリにあります。
2. 標準デザイン例を任意の作業ディレクトリにコピーします。オリジナルのデザイン例に影響を与えないでデザイン・ファイルを変更できるように、新しいロケーションを使用します。
3. Quartus II ソフトウェアの File メニューで、**Open Project** をクリックし、作成したディレクトリから <my_board>_standard.qpf プロジェクト・ファイルを開きます。
4. Tools メニューの **SOPC Builder** をクリックして、SOPC Builder を起動します。

CFI フラッシュから直接ブートする場合、標準デザイン例は追加メモリなしで動作するため、15 ページの「Nios II IDEでの高度なブート・コピアの構築」に進みます。

オンチップ・ブートROMをシステムに追加するには、以下のステップに従います。

1. SOPC Builderの**System Contents**タブで、**Memories and Memory Controllers, On-Chip**を展開して、**On-Chip Memory (RAM or ROM)**を選択します。
2. **Add**をクリックして、システムにコンポーネントを追加します。メモリを指定する際に、以下の設定を使用します。
 - メモリ・タイプ:ROM (読み出し専用)
 - シングル・ポート・アクセス (非デュアル・ポート)
 - メモリ幅:32 ビット
 - 合計メモリ・サイズ:8K バイト

指定されたペリフェラルのサイズは、ブート・コピア例の最大バージョンのコード・イメージ全体を確実に保持できます。このイメージには、以下のコードが含まれます。

- `.entry` セクション内のリセット・コード
- `crt0.s` スタートアップ・コード
- `alt_main` エントリ・ポイントを含む `.text` セクション
- 任意の初期化済み読み出し専用データを保持する `.rodata` セクション
- 任意の初期化済み読み出し/書き込みデータを保持する `.rwdata` セクション
- `.exception` セクション内の例外ハンドラ

これらのセクションのいくつかは、`crt0.s` スタートアップ・コードが実行されたときに例外のRAMにコピーされますが、これらのセクションはすべて最初に、このオンチップ・メモリに格納されます。

3. **System** メニューで、**Auto-Assign Base Addresses** をクリックします。
4. 新しい **On-Chip Memory** コンポーネントを右クリックし、**Rename** をクリックします。コンポーネントの名前を、`boot_rom` などの記述名に変更します。
5. オンチップ・メモリからのブート・コピアの実行をイネーブルにするには、システムの **cpu** コンポーネントを右クリックし、**Edit** をクリックします。
6. Nios II Processor 設定ウィンドウで、**Reset Vector Memory** を `boot_rom` に設定して、**Offset** を `0x00000000` にし、**Exception Vector Memory** を `ssram` に設定します。
7. **Finish** をクリックして、Nios II Processor 設定ウィンドウを終了します。
8. **Generate** をクリックして、SOPC Builder システムを生成します。

Nios II IDE での高度なブート・コピアの構築

Nios II の実行可能なコードでブート・コピア例を構築するには、以下のステップに従います。

1. SOPC Builder の **System Generation** タブで、**Nios II IDE** をクリックして、Nios II IDE を起動します。
2. Nios II IDE の File メニューで、**New** を選択し、次に **Project** を選択します。
3. **Nios II C/C++ Application** をダブルクリックして、新しいソフトウェア・プロジェクトを作成します。
4. **Blank Project** プロジェクト・テンプレートを選択し、例えば **advanced_boot_copier** のような、プロジェクトの記述名を指定します。本資料の以降の部分では、このプロジェクトを **advanced_boot_copier** と呼びます。
5. **Target Hardware** の選択が、作成した SOPC Builder システムに関連付けられた **.ptf** ファイルを指していることを確認します。
6. **Finish** をクリックします。
7. www.altera.com/literature/lit-nio2.jsp からデザイン・ファイルをダウンロードし、ブート・コピア例のソース・ファイル **advanced_boot_copier.c** および **advanced_boot_copier.h** を見つけます。これらのソース・ファイルは、**boot_copier_src/advanced_boot_copier** ディレクトリに含まれています。
8. Nios II IDE の Nios II C/C++ Projects ビューで、**advanced_boot_copier.c** ファイルおよび **advanced_boot_copier.h** ファイルを **advanced_boot_copier** フォルダにコピーします。
9. Nios II C/C++ Projects ビューで、新たにコピーされた **advanced_boot_copier.c** ファイルをダブルクリックしてエディタで開き、次の行を編集します。

```
#define BOOT_METHOD <boot_method>
```

これにより、使用するブート方法を示します。<boot_method> に使用可能なオプションは、以下のとおりです。

- BOOT_FROM_CFI_FLASH
- BOOT_CFI_FROM_ONCHIP_ROM
- BOOT_EPCS_FROM_ONCHIP_ROM

この #define は、コンパイラに使用しているブート方法に適したブート・コピアを構築するよう指示するものです。

10. ブート中にアプリケーションの JTAG UART へのメッセージ出力を制限するには、次の行を編集します。

```
#define USING_JTAG_UART 1
```

上の行を次のように書き替えます。

```
#define USING_JTAG_UART 0
```

この `#define` は、コンパイラにすべての JTAG UART コードを除外して、ブート・コピアを構築するよう指示するものです。

11. Nios II C/C++ Projects ビューで、ブート・コピア・プロジェクトを右クリックし、**System Library Properties** をクリックします。**System Library** ページを選択します。
12. **System Library** ページで、以下のメモリ・セクションを調整します。
 - CFI フラッシュから直接ブートする場合は、メモリ・セクション `.text` および `.rodata` を `ext_flash` に設定します。
 - オンチップ・メモリから実行される CFI または EPCS をブートする場合は、メモリ・セクション `.text` および `.rodata` を `boot_rom` に設定します。
 - メモリ・セクション `.rwddata`、`heap`、および `stack` を `sdram` に設定します。
13. コード・サイズを縮小するために、**System Library** セクションで以下の追加変更を行います。
 - **Max file descriptors** に 4 を入力します。
 - **Program never exits** をオンにします。
 - **Support C++** をオフにします。
 - **Clean exit** をオフにします。
 - **Reduced Device Drivers** をオンにします。
 - **Small C Library** をオンにします。
 - **System clock timer** が `sys_clk_timer` に設定されていることを確認します。
14. コード・サイズが小さくなるように、コンパイラ最適化を適切に設定します。
 - a. **System Library Properties** ダイアログ・ボックスで、**C/C++ Build** ページをクリックします。
 - b. **Tool Settings** タブで、**Nios II Compiler** を展開し、**General** をクリックします。
 - c. 最適化レベルとして **Optimize size (-Os)** を選択します。
15. **OK** をクリックして、**System Library Properties** ダイアログ・ボックスを閉じます。

16. コード・サイズが小さくなるように、ブート・コピア・プロジェクト自体にコンパイラ最適化を設定します。
 - a. Nios II C/C++ Projects ビューで、**advanced_boot_copier** プロジェクトを右クリックし、**Properties** をクリックします。
 - b. **C/C++ Build** ページをクリックします。
 - c. **Tool Settings** タブで、**Nios II Compiler** を展開し、**General** をクリックします。
 - d. 最適化レベルとして **Optimize size (-Os)** を選択します。
17. **OK** をクリックして、**Properties** ダイアログ・ボックスを閉じます。
18. **advanced_boot_copier_syslib** プロジェクトを右クリックし、次に **Build Project** をクリックします。



システム・ライブラリ・プロジェクトを構築すると、**alt_sys_init.c** ファイルが生成されます。メイン **advanced_boot_copier** プロジェクトを構築する前に、このファイルを変更します。

19. Nios II IDE の **Debug/system_description** の下の **advanced_boot_copier_syslib** プロジェクトで、**alt_sys_init.c** ファイルを見つけます。
20. Copy the **alt_sys_init.c** ファイルを、**advanced_boot_copier_syslib** プロジェクトからメイン **advanced_boot_copier** プロジェクトにコピーします。
21. メイン **advanced_boot_copier** プロジェクトで、新たにコピーされた **alt_sys_init.c** を右クリックし、次に **Rename** をクリックします。**alt_sys_init.c** の名前を **my_sys_init.c** に変更します。
22. **my_sys_init.c** をダブルクリックしてエディタで開きます。
23. **alt_sys_init** 関数の名前を **my_sys_init** に変更します。
24. **SYS_CLK_TIMER** および **JTAG_UART** を除くすべてのコンポーネントに対する、**_INSTANCE** および **_INIT** を含む行をコメントにします。
25. ブート中にアプリケーションが **JTAG_UART** にメッセージを出力しないようにして、ブート・コピアの実行中の遅延を排除したい場合は、**SYS_CLK_TIMER** 行および **JTAG_UART** 行をコメントにします。これにより、コード・サイズが約 1.5K バイトに縮小されます。



JTAG_UART ルーチンには、システム・クロック・タイマが必要です。**JTAG_UART** を使用する場合は、**SYS_CLK_TIMER** 初期化をディセーブルにしてはなりません。

26. 完成した **my_sys_init.c** システム初期化ルーチンを保存し、ファイルを閉じます。
27. **advanced_boot_copier** プロジェクトを右クリックし、**Build Project** をクリックして、メイン・ブート・コピア・プロジェクトを構築します。

これで実行可能なブート・コピアが完成し、Nios II プロセッサですぐに動作できる状態になりました。次に、新しいブート・コピアを使用してブートするアプリケーションを作成する必要があります。

ブートするテスト・アプリケーションの構築

以下のステップでは、Nios II IDE および Nios II Command Shell スクリプトを使用して、高度なブート・コピアを使ってブートするテスト・アプリケーションを構築します。

1. Nios II IDE の File メニューで、**New** を選択し、次に **Project** を選択します。
2. **Nios II C/C++ Application** をダブルクリックして、新しいソフトウェア・プロジェクトを作成します。
3. **Hello World** テンプレートを選択し、例えば **boot_test_app** のような、プロジェクトの記述名を指定します。本資料の以降の部分では、このプロジェクトを **boot_test_app** と呼びます。
4. **Finish** をクリックします。
5. Nios II C/C++ Projects ビューで、**boot_test_app** を右クリックし、**System Library Properties** をクリックします。**System Library** ページを選択します。
6. **System Library** ページで、**Target Hardware** の選択が前に作成した SOPC Builder システムに関連付けられた **.ptf** ファイルを指していることを確認します。
7. **OK** をクリックして、**System Library Properties** ダイアログ・ボックスを閉じます。Nios II C/C++ Projects ビューに戻ります。
8. **boot_test_app** プロジェクトを右クリックし、次に **Build Project** をクリックします。IDE により、**boot_test_app.elf** という名前のテスト・アプリケーション用実行ファイルが構築されます。

boot_test_app.elf をフラッシュ・メモリにプログラムする前に、それをブート・コピアが理解できるブート・レコードにパッケージする必要があります。これを行うために、Nios II Command Shell からスクリプトを実行します。簡単にするために、またスクリプトが後で容易に使用できるように、以下のステップに従ってスクリプトを Nios II 検索パス内のロケーションにコピーします。

9. www.altera.com/literature/lit-nio2.jsp からダウンロードしたデザイン・ファイルで、**flash_image_scripts** ディレクトリを見つけます。

- 以下のファイルを **flash_image_scripts** ディレクトリから `<Nios II EDS install path>/bin` ディレクトリにコピーして、Nios II Command Shell からスクリプトを使用できるようにします。

- **make_flash_image_script.sh**
- **make_header.pl**
- **read_flash_image.pl**

- Nios II Command Shell を開きます。(Windows で、スタート > すべてのプログラム > Altera > Nios II EDS > Nios II Command Shell をクリックします。)
- Nios II Command Shell で、**cd** コマンドを使用して、ディレクトリを `<project directory>/software/boot_test_app/Debug` に変更します。
- make_flash_image_script.sh** スクリプトを以下のパラメータで実行して、**.elf** ファイルをブート・レコードにパッケージします。

```
[SOPC Builder]$ make_flash_image_script.sh boot_test_app.elf
```



上記のスクリプトを実行すると、空のロードブル・セグメントに関するワーニングが発行され、中間ファイルの名前 **fake_flash_copier.srec** が表示されることがあります。これらのメッセージは無視することができます。

このスクリプトにより、**boot_test_app/Debug**ディレクトリに**boot_test_app.elf.flash.bin** ファイルおよび **boot_test_app.elf.flash.srec** ファイルが作成されます。これで、ブート・コピア例を用いてテスト・アプリケーションをブートするのに必要な、すべてのバイナリ・イメージが揃いました。次に、これらのイメージを適切なロケーションにプログラムします。

CFI フラッシュ・メモリからの直接ブート

この項では、Nios II Flash Programmer を使用して、ブート・コピアとテスト・アプリケーションを CFI フラッシュ・メモリにプログラムします。



オンチップ・メモリからブートする場合は、この項は適用されません。20 ページの「[オンチップ・メモリからの CFI または EPCS フラッシュのブート](#)」に進みます。

- Quartus II ソフトウェアの Tools メニューの **Programmer** をクリックします。
- File** カラムの `<none>` をクリックし、Quartus II プロジェクト・ディレクトリ内の **_standard.sof** ファイルを参照して選択します。
- Program/Configure** オプションがオンになっていることを確認します。
- Start** をクリックして、この **_standard.sof** ファイルで FPGA をコンフィギュレーションします。

5. Nios II IDE に戻り、**advanced_boot_copier** プロジェクトをクリックしてハイライトします。
6. Nios II IDE の Tools メニューの **Flash Programmer** をクリックします。
7. **Flash Programmer** ダイアログ・ボックスで、**Flash Programmer** をクリックし、次に **New** アイコンをクリックして、新しいフラッシュ・プログラマ・コンフィギュレーションを作成します。新しいコンフィギュレーションは、自動的に **advanced_boot_copier** プロジェクトの名前を使用します。
8. **Program software project into flash memory** をオンにします。このオプションにより、高度なブート・コピアの実行ファイルが、システムのリセット・アドレスである CFI フラッシュ・メモリのオフセット 0x0 にプログラムされます。
9. **Program FPGA configuration data into hardware-image region of flash memory** をオンにします。Quartus II プロジェクト・ディレクトリ内の **_standard.sof** ファイルを FPGA コンフィギュレーション・ファイルとして選択し、ハードウェア・イメージ・ロケーションに **user** を選択します。
10. **Program a file into flash memory** をオンにします。File ボックスで、前に作成した **boot_test_app.elf.flash.bin** ファイルに移動します。このファイルは **boot_test_app/Debug** ディレクトリにあります。オフセットを **0x00100000** または **0x00200000** に設定します。これは、CFI フラッシュ・モードからブートする場合、ブート・コピアはこれら 2 つのロケーションにそれぞれブート・イメージ 1 と 2 が存在すると想定しているためです。2 つのアドレスは同等に良好に機能します。
11. **Target Connection** タブで、JTAG ケーブルが識別されていることを確認します。
12. **Apply** をクリックし、次に **Program** をクリックして、フラッシュ・メモリをプログラムします。ブート・コピア、テスト・アプリケーション、および FPGA コンフィギュレーション・イメージのすべてが CFI フラッシュにプログラムされます。
13. 23 ページの「高度なブート・コピア例の実行」に進みます。

オンチップ・メモリからの CFI または EPCS フラッシュのブート

この項では、Quartus II ソフトウェアを使用してブート・コピアを FPGA の **boot_rom** メモリにプログラムし、次に Nios II Flash Programmer を使用してテスト・アプリケーションのブート・レコードを CFI または EPCS フラッシュ・メモリにプログラムします。



CFI フラッシュ・メモリから直接ブートする場合、この項は適用されません。CFI フラッシュ・メモリからの直接ブートは、19 ページの「CFI フラッシュ・メモリからの直接ブート」で説明されています。


ブート・コピアを FPGA の **boot_rom** メモリにプログラムするには、以下のステップに従います。

1. SOPC Builder がまだ開いている場合は、**Exit** をクリックして SOPC Builder を閉じます。
2. Quartus II ウィンドウで、Assignments メニューの **Settings** をクリックします。
3. **Category** リストで、**Compilation Process Settings** をクリックし、次に **Use Smart Compilation** をオンにします。このオプションは、オンチップ・メモリのコンテンツのアップデートのみ必要な場合に、デザイン全体のリコンパイルを防止します。ただし、最初の Quartus II コンパイルはフル・コンパイルでなければなりません。これは、システムにオンチップ・メモリが追加されると、デザインが変更されるからです。
4. Processing メニューの **Start Compilation** をクリックして、プロジェクトをリコンパイルします。
5. コンパイルが完了したら、Tools メニューの **Programmer** をクリックします。
6. **File** カラムの **<none>** をクリックし、Quartus II プロジェクト・ディレクトリ内の **_standard.sof** ファイルを参照して選択します。
7. **Program/Configure** オプションがオンになっていることを確認します。
8. **Start** をクリックして、この **_standard.sof** ファイルで FPGA をコンフィギュレーションします。

これで、FPGA の **boot_rom** メモリにブート・コピア例の実行可能イメージが含まれました。

テスト・アプリケーションをフラッシュ・メモリにプログラムするには、以下のステップに従います。

1. Nios II IDE に戻り、**boot_test_app** プロジェクトをクリックしてハイライトします。
2. Tools メニューの **Flash Programmer** をクリックします。
3. **New** アイコンをクリックして、新しいフラッシュ・プログラマ・コンフィギュレーションを作成します。新しいコンフィギュレーションは、自動的に **boot_test_app** プロジェクトの名前を使用します。
4. **Program software project into flash memory** をオフにします。ブート・コピアは、すでに FPGA のオンチップ **boot_rom** にプログラムされています。
5. **Program FPGA configuration data into hardware-image region of flash memory** をオンにします。
6. Quartus II プロジェクト・ディレクトリ内の **_standard.sof** ファイルを FPGA コンフィギュレーション・ファイルとして選択します。

7. フラッシュ・メモリ内の適切なロケーションを選択して、FPGA コンフィギュレーション・イメージをプログラムします。
 - CFI ブート・イメージをブートする場合は、ハードウェア・イメージ・ロケーションに **user** を選択します。これを選択すると、ブート・コピーを含む FPGA イメージがパワー・オン・リセット時に、FPGA で CFI フラッシュ・メモリからコンフィギュレーションされます。
 - EPCS ブート・イメージをブートする場合は、ハードウェア・イメージ・ロケーションに **epcs** を選択します。これを選択すると、ブート・コピーを含む FPGA イメージがパワー・オン・リセット時に、FPGA で EPCS フラッシュ・メモリからコンフィギュレーションされます。
 8. **Program a file into flash memory** をオンにします。File ボックスで、前に作成した **boot_test_app.elf.flash.bin** ファイルに移動します。このファイルは **boot_test_app/Debug** ディレクトリにあります。これはブート・コピーが読み出して、RAM にコピーするアプリケーション・ブート・イメージです。
 9. フラッシュ・メモリ内の適切なロケーションを選択して、アプリケーションのブート・イメージをプログラムします。
 - CFI フラッシュに格納されたイメージをブートする場合は、**Memory** を **ext_flash** に設定し、**Offset** を **0x00000000** または **0x00100000** に設定します。
 - EPCS フラッシュに格納されたイメージをブートする場合は、**Memory** を **epcs_controller** に設定し、**Offset** を **0x00060000** または **0x00080000** に設定します。
-  **advanced_boot_copier.c** でフラッシュ・イメージのオフセットを編集した場合は、**Offset** を、上記のオフセットではなく、**advanced_boot_copier.c** で定義したイメージ・オフセットの1つに設定します。
10. **Apply** をクリックし、次に **Program Flash** をクリックして、フラッシュ・メモリをプログラムします。テスト・アプリケーションと FPGA コンフィギュレーション・イメージがフラッシュ・メモリにプログラムされます。

高度なブート・コピア例の実行

開発ボードで高度なブート・コピア例を実行するには、以下のステップに従います。

- ✓ **Flash Programmer** の終了後、ボードのパワー・オン・リセット・スイッチを押します。

ブート・コピアは、直ちに LED0 を点灯させ、有効なブート・イメージを RAM にコピーした後、RAM にジャンプする直前に LED1 を点灯させます。有効なブート・イメージが見つからなかった場合、ブート・コピアは LED2 を点灯させ、5 秒間待ってからリセット・アドレスに戻り、ブート・プロセスを再度開始します。

JTAG UART がイネーブルされている場合、ブート・ローダおよびテスト・アプリケーションの両方が JTAG UART にステータス・メッセージを出力します。JTAG UART および SYS_CLK_TIMER が **my_sys_init.c** ファイルで初期化されていて、かつ **advanced_boot_copier.c** ファイルで USING_JTAG_UART の値が 1 のままである場合、これらのメッセージを表示できます。

メッセージを表示するには、以下のステップに従います。

1. Nios II Command Shell に戻り、次のコマンドを入力して **nios2-terminal** ユーティリティを実行します。

```
[SOPC Builder]$ nios2-terminal
```

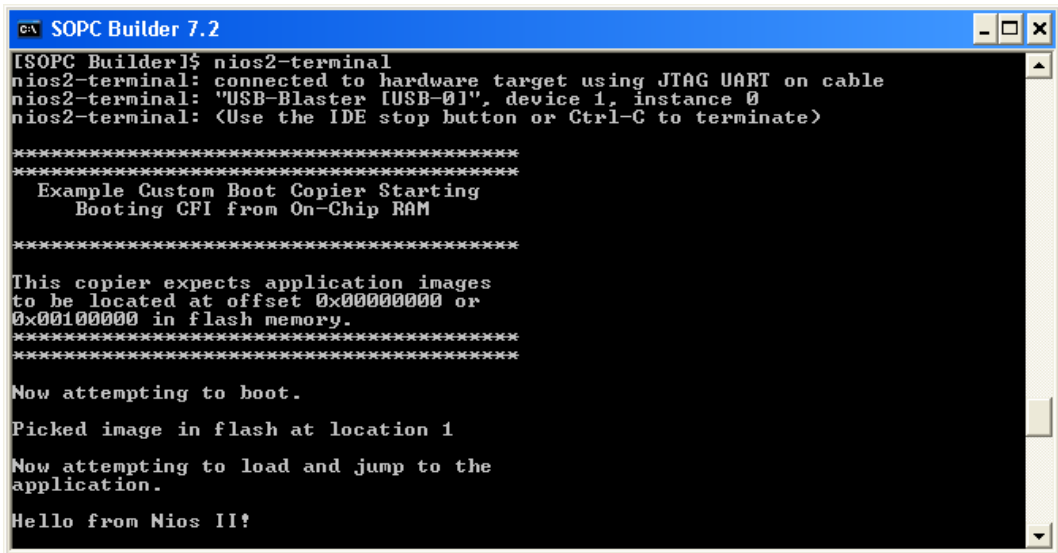


デフォルト設定で **nios2-terminal** を JTAG UART に接続できない場合は、**--help** オプションを指定してこのコマンドを実行し、必要と考えられるコマンドライン・スイッチを一覧表示します。

ブート・コピアが正常に実行された場合は、[図 3](#) に示すとおり、**nios2-terminal** からの出力が表示されます。外部メモリからブートした場合、または EPCS フラッシュからブートした場合、出力は若干異なります。

2. **nios2-terminal** に、ブート・コピアからの切り捨てられた出力が表示され、続いてブート・イメージが表示される場合は、開発ボードの CPU Reset ボタンを押して、ブート・プロセスを再度実行して、完全な出力を表示します。

図 3. 高度なブート・コピアの出力



```
[SOPC Builder]$ nios2-terminal
nios2-terminal: connected to hardware target using JTAG UART on cable
nios2-terminal: "USB-Blaster [USB-01]", device 1, instance 0
nios2-terminal: <Use the IDE stop button or Ctrl-C to terminate>

*****
Example Custom Boot Copier Starting
Booting CFI from On-Chip RAM
*****

This copier expects application images
to be located at offset 0x00000000 or
0x00100000 in flash memory.
*****

Now attempting to boot.

Picked image in flash at location 1

Now attempting to load and jump to the
application.

Hello from Nios II!
```

小さなブート・コピアの例

この項では、メモリ使用量を最小限に抑えたいユーザー向けに、コード・サイズが小さいブート・コピア例について説明します。



デザイン・ファイルへのハイパーリンクは、Nios II 資料ページの本資料の次に記載されています。www.altera.com/literature/lit-nio2.jsp をご覧ください。

小さなブート・コピアの特長

小さなブート・コピア例は、プログラム・メモリを殆ど使用しないように設計された最小のプログラムです。以下の動作のみ実行します。

1. フラッシュ・メモリからのアプリケーション・ブート・レコードの読み出し
2. RAM へのコピー
3. アプリケーションのエントリ・ポイントへのジャンプ

小さなブート・コピア例は、ブート・イメージを 1 つだけサポートし、エラー・チェックを実行せず、JTAG UART へのメッセージの出力をサポートしません。より高度なブート・コピアに関心がある場合は、4 ページの「[高度なブート・コピア例](#)」を参照してください。

Nios II アセンブリ言語での実装

コード・サイズをできる限り小さくするために、小さなブート・コピア例は Nios II アセンブリ言語で記述されています。ブート・コピアが使用する変数はすべて、Nios II プロセッサの RAM ではなく汎用レジスタに実装されています。したがって、ブート・コピア自体にはデータ・メモリの要件はありません。小さなブート・コピアには、`.rodata`、`.rwdata`、`stack`、`heap` の各セクションはありません。このブート・コピアは、データ・メモリを必要としないので、メモリ内の任意の場所に容易に再配置でき、データ・メモリ・セクションを設定しないで不揮発性フラッシュ・メモリから直接実行することもできます。

システム初期化

小さなブート・コピアは、必要最小限のシステム初期化のみ実行します。以下の初期化タスクが、ブート・コピアにより実行されます。

- プロセッサのステータス・レジスタをクリアして割り込みをディセーブル。
- 命令・キャッシュをフラッシュ。
- プロセッサのパイプラインをフラッシュ。

コード・サイズ

小さなブート・コピアはコンパイルすると、わずか 200 バイト長の実行ファイルになります。このブート・コピアは、Cyclone II FPGA におけるメモリの最小単位である M4K ブロックの 1 つに十分収まるサイズです。

小さなブート・コピア例の実装

この項では、Nios 開発ボードで小さなブート・コピア例を構築および実行するのに必要なステップについて説明します。このブート・コピアは、アセンブリ言語で記述されたコード・サイズが小さい必要最小限のものです。より豊富な機能を備えたブート・コピアを構築したい場合は、[13 ページの「高度なブート・コピア例の実装」](#)を参照してください。

小さなブート・コピア例は、**make** ユーティリティを使用して、Nios II Command Shell で構築されます。小さなブート・コピアを構築するには、Nios II IDE は使用しません。

ソフトウェア・ツールおよび開発ボードのセットアップ

1. コンピュータに Nios II EDS v7.2 (またはそれ以降) および Quartus II ソフトウェア v7.2 (またはそれ以降) がインストールされていることを確認します。
2. 電源および USB-Blaster を Nios II 開発ボードに接続します。

適切なハードウェア・デザインの作成

以下のステップでは、小さなブート・コピア例を実行できる Nios II システムを開いて、変更し、生成します。

プロジェクト例を開いて、システムにオンチップ ROM を追加するには、以下のステップに従います。

1. Verilog HDL または VHDL で記述された、Nios 開発ボード用の Nios II 標準デザイン例を見つけます。標準デザイン例は、<Nios II EDS install path>/**examples** ディレクトリにあります。
2. 標準デザイン例を任意の作業ディレクトリにコピーします。オリジナルのデザイン例に影響を与えないでデザイン・ファイルを変更できるように、新しいロケーションを使用します。
3. Quartus II ソフトウェアの File メニューで、**Open Project** をクリックし、作成したディレクトリから <my_board>_standard.qpf プロジェクト・ファイルを開きます。
4. Tools メニューの **SOPC Builder** をクリックして、SOPC Builder を起動します。
 1. SOPC Builder の **System Contents** タブで、**On-Chip** を展開して、**On-Chip Memory (RAM or ROM)** を選択します。
 2. **Add** をクリックして、システムにコンポーネントを追加します。メモリを指定する際に、以下の設定値を使用します。
 - メモリ・タイプ:ROM (読み出し専用)
 - シングル・ポート・アクセス (非デュアル・ポート)
 - メモリ幅:32 ビット
 - 合計メモリ・サイズ:512 バイト

指定されたオンチップ・メモリ・サイズは、メモリ・スペースが無駄にならないことを保証するものです。Cyclone II FPGA における使用可能な最小メモリ・ブロックは、512 バイト (1 つの M4K ブロック) です。小さなブート・コピア例は 200 バイトのメモリしか必要としませんが、M4K ブロックの残りの部分はイネーブルにしない限り使用できません。そのため、アルテラではブロック全体をイネーブルにして残りの部分を浪費しないよう推奨しています。

3. System メニューの **Auto-Assign Base Addresses** をクリックします。
4. 新しい **On-Chip Memory** コンポーネントを右クリックし、**Rename** をクリックします。boot_rom などの記述名を指定します。
5. オンチップ・メモリからのブート・コピアの実行をイネーブルにするには、システム内の **cpu** コンポーネントを右クリックし、**Edit** をクリックします。

6. Nios II Processor 設定ウィンドウで、**Reset Vector Memory** を **boot_rom** に設定し、オフセットを `0x00000000` にします。
7. **Finish** をクリックして、Nios II Processor 設定ウィンドウを終了します。
8. **Generate** をクリックして、SOPC Builder システムを生成します。

'make' を使用した小さなブート・コピアの構築

以下のステップでは、Nios II Command Shell を使用してブート・コピア例を構築します。

Nios II の実行可能なコードでブート・コピア例を構築するには、以下のステップに従います。

1. www.altera.com/literature/lit-nio2.jsp からデザイン・ファイルをダウンロードし、ブート・コピア例のソース・ファイル **small_boot_copier.s** と **small_boot_copier.h**、および Make ファイル **Makefile** を見つけます。これらのファイルは、**boot_copier_src/small_boot_copier** ディレクトリに含まれています。
2. これら 3 つのファイルをクリップボードにコピーします。
3. 作成した Quartus II プロジェクトのディレクトリに移動し、そのプロジェクト内に **software** という名前の新しいディレクトリを作成します。
4. ディレクトリを開いて、**small_boot_copier** という名前のサブディレクトリを作成します。
5. ソース・ファイルを **small_boot_copier** ディレクトリにペーストします。
6. Nios II Command Shell を開きます。(Windows で、スタート > すべてのプログラム > Altera > Nios II EDS > Nios II Command Shell をクリックします。)
7. Nios II Command Shell で、**cd** コマンドを使用して、ディレクトリを `<project directory>/software/small_boot_copier` に変更します。
8. SOPC Builder で、**ext_flash** コンポーネントのベース・アドレスを決定します (`<flash_base_address>`)。)
9. Nios II Command Shell で、次のコマンドを入力します。

```
[SOPC Builder]$ make all FLASH_BASE=<flash_base_address> \  
                    BOOT_IMAGE_OFFSET=0x00100000
```

このコマンドでは、小さなブート・コピアが構築され、フラッシュ・メモリ内のオフセット `0x00100000` にあるブート・イメージを検索するようにハードコーディングされます。



ブート・イメージのオフセット $0x00100000$ は、そこに他の重要なデータが配置されていないことを前提に選択されます。このオフセットはアプリケーションとより関連性の高い値に自由に変更できます。ただし、フラッシュ・メモリにブート・イメージをプログラムする (31 ページのステップ 9) ときは、必ず現在のステップで選択するオフセットと同じ値にプログラムしてください。

これで、**small_boot_copier.hex** という名前の実行可能なブート・コピアが完成し、Nios II プロセッサですぐに動作できる状態になりました。次に、新しいブート・コピアを使用してブートするテスト・アプリケーションを作成する必要があります。

ブートするテスト・アプリケーションの構築

以下のステップでは、Nios II IDE および Nios II Command Shell スクリプトを使用して、小さなブート・コピアを使ってブートするテスト・アプリケーションを構築します。

1. Nios II IDE で、**File > New > Project > Nios II C/C++ Application** をクリックして、新しいソフトウェア・プロジェクトを作成します。
2. **Hello World** テンプレートを選択し、例えば **boot_test_app** のような、プロジェクトの記述名を指定します。本資料の以降の部分では、このプロジェクトを **boot_test_app** と呼びます。
3. Nios II C/C++ Projects ビューで、**boot_test_app** を右クリックし、**System Library Properties** をクリックします。**System Library** ページを選択します。
4. **System Library** ページで、**Target Hardware** が前に作成した SOPC Builder システムに関連付けられた **.ptf** ファイルを指していることを確認します。
5. **OK** をクリックして、**System Library Properties** ダイアログ・ボックスを閉じます。Nios II C/C++ Projects ビューに戻ります。
6. **boot_test_app** プロジェクトを右クリックし、次に **Build Project** を選択します。Nios II IDE により、**boot_test_app.elf** という名前のテスト・アプリケーション用実行ファイルが構築されます。

boot_test_app.elf をフラッシュ・メモリにプログラムする前に、それをブート・コピアが理解できるブート・レコードにパッケージする必要があります。これを行うために、Nios II Command Shell からスクリプトを実行します。簡単にするために、またスクリプトが後で容易に使用できるように、以下のステップに従ってスクリプトを Nios II 検索パス内のロケーションにコピーします。

7. www.altera.com/literature/lit-nio2.jsp からダウンロードしたデザイン・ファイルで、**flash_image_scripts** ディレクトリを見つけます。

8. 以下のファイルを **flash_image_scripts** ディレクトリから `<Nios II EDS install path>/bin` ディレクトリにコピーして、Nios II Command Shell からスクリプトを使用できるようにします。
 - **make_flash_image_script.sh**
 - **make_header.pl**
 - **read_flash_image.pl**
9. Nios II Command Shell を開きます。(Windows で、**Start > All Programs > Altera > Nios II EDS > Nios II Command Shell** をクリックします。)
10. Nios II Command Shell で、**cd** コマンドを使用して、ディレクトリを `<project directory>/software/boot_test_app/Debug` に変更します。
11. **make_flash_image_script.sh** スクリプトを以下のパラメータで実行して、**.elf** ファイルをブート・レコードにパッケージします。

```
[SOPC Builder]$ make_flash_image_script.sh boot_test_app.elf
```



上記のスクリプトを実行すると、空のローダブル・セグメントに関するワーニングが発行され、中間ファイルの名前 **fake_flash_copier.srec** が表示されることがあります。これらのメッセージは無視することができます。

このスクリプトにより、**boot_test_app/Debug** ディレクトリに **boot_test_app.elf.flash.bin** ファイルおよび **boot_test_app.elf.flash.srec** ファイルが作成されます。これで、ブート・コピア例を用いてテスト・アプリケーションをブートするのに必要な、すべてのバイナリ・イメージが揃いました。次に、これらのイメージを適切なロケーションにプログラムします。

オンチップ・メモリからのブート

この項では、Quartus II ソフトウェアを使用してブート・コピアを FPGA の **boot_rom** メモリにプログラムし、次に Nios II Flash Programmer を使用してテスト・アプリケーションのブート・レコードを CFI フラッシュ・メモリにプログラムします。

ブート・コピアを FPGA の **boot_rom** メモリにプログラムするには、以下のステップに従います。

12. SOPC Builder がまだ開いている場合は、**Exit** をクリックして SOPC Builder を閉じます。
13. **software/small_boot_copier** ディレクトリで、**small_boot_copier.hex** ファイルを見つけます。
14. **small_boot_copier.hex** を前に作成した Quartus II プロジェクト・ディレクトリにコピーし、名前を **boot_rom.hex** に変更します。



そのディレクトリに同じ名前のファイルがすでに存在するというワーニングが表示されることがあります。古いファイルを置き換えるよう求められた場合は、**Yes** をクリックします。

次の Quartus II のコンパイルにより、ブート・コピアの実行ファイルが **boot_rom** のコンテンツとして実装されます。

15. Quartus II ウィンドウで、Assignments メニューの **Settings** をクリックします。
16. **Category** リストで、**Compilation Process Settings** をクリックし、次に **Use Smart Compilation** をオンにします。このオプションは、オンチップ・メモリのコンテンツのアップデートのみ必要な場合に、デザイン全体のリコンパイルを防止します。ただし、最初の Quartus II コンパイルはフル・コンパイルでなければなりません。これは、システムにオンチップ・メモリが追加されると、デザインが変更されるからです。
17. Processing メニューの **Start Compilation** をクリックして、Quartus II プロジェクトをリコンパイルします。
18. コンパイルが完了したら、Tools メニューの **Programmer** をクリックします。
19. **File** カラムの *<none>* をクリックし、Quartus II プロジェクト・ディレクトリ内の **_standard.sof** ファイルを参照して選択します。
20. **Program/Configure** オプションがオンになっていることを確認します。
21. **Start** をクリックして、この **_standard.sof** ファイルで FPGA をコンフィギュレーションします。

これで、FPGA の **boot_rom** メモリにブート・コピア例の実行可能イメージが含まれました。

テスト・アプリケーションをフラッシュ・メモリにプログラムするには、以下のステップに従います。

1. Nios II IDE に戻り、**boot_test_app** プロジェクトをクリックしてハイライトします。
2. Tools メニューの **Flash Programmer** をクリックします。
3. **New** アイコンをクリックして、新しいフラッシュ・プログラマ・コンフィギュレーションを作成します。新しいコンフィギュレーションは、自動的に **boot_test_app** プロジェクトの名前を使用します。
4. **Program software project into flash memory** をオフにします。ブート・コピアは、すでに FPGA のオンチップ **boot_rom** にプログラムされています。
5. **Program FPGA configuration data into hardware-image region of flash memory** をオンにします。

6. Quartus II プロジェクト・ディレクトリ内の `_standard.sof` ファイルを FPGA コンフィギュレーション・ファイルとして選択します。
7. ハードウェア・イメージ・ロケーションに **user** を選択します。これを選択すると、ブート・コピアを含む FPGA イメージがパワー・オン・リセット時に、FPGA で CFI フラッシュ・メモリからコンフィギュレーションされます。
8. **Program a file into flash memory** をオンにします。**File** ボックスで、前に作成した `boot_test_app.elf.flash.bin` ファイルに移動します。このファイルは `boot_test_app/Debug` ディレクトリにあります。これはブート・コピアが読み出して、RAM にコピーするアプリケーション・ブート・イメージです。
9. **Memory** を `ext_flash` に設定し、**Offset** を `0x00100000` に設定します。



小さなブート・コピア例を構築する際に (27 ページのステップ 9) 、`0x00100000` 以外のブート・イメージ・オフセットを選択する場合は、必ず選択したイメージ・オフセットに **Offset** を設定してください。

10. **Apply** をクリックし、次に **Program Flash** をクリックして、フラッシュ・メモリをプログラムします。テスト・アプリケーションと FPGA コンフィギュレーション・イメージが CFI フラッシュ・メモリにプログラムされます。

小さなブート・コピア例の実行

開発ボードで小さなブート・コピア例を実行するには、以下のステップに従います。

1. **Flash Programmer** の終了後、ボードのパワー・オン・リセット・スイッチを押します。この時点で、ブート・コピアがテスト・アプリケーションをブートするはずですが。
2. テスト・アプリケーションが実際にロードおよび実行されるかテストするには、Nios II Command Shell ウィンドウに戻って、次のコマンドを入力して `nios2-terminal` ユーティリティを実行します。

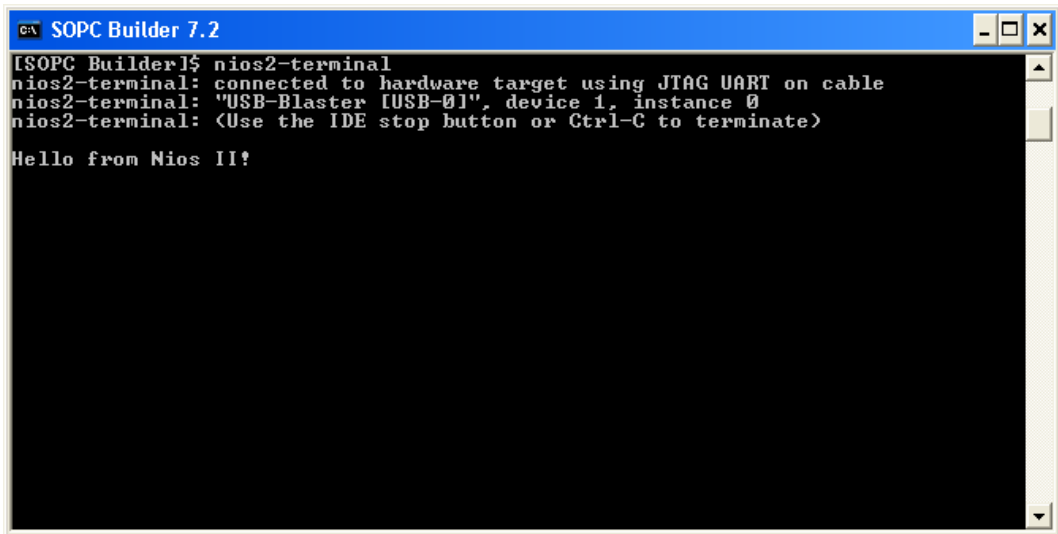
```
[SOPC Builder]$ nios2-terminal
```

ブート・コピアが正常に実行された場合は、`nios2-terminal` からの出力が [図 4](#) に示すとおり表示されます。



デフォルトの設定値で `nios2-terminal` を **JTAG UART** に接続できない場合は、`--help` オプションを指定してこのコマンドを実行し、必要と考えられるコマンドライン・スイッチを一覧表示します。

図 4. 小さなブート・コピアの出力

A screenshot of a terminal window titled "SOPC Builder 7.2". The terminal shows the following text:

```
[SOPC Builder]# nios2-terminal  
nios2-terminal: connected to hardware target using JTAG UART on cable  
nios2-terminal: "USB-Blaster [USB-01]", device 1, instance 0  
nios2-terminal: <Use the IDE stop button or Ctrl-C to terminate>  
  
Hello from Nios II!
```

ブート・コピア のデバッグ

Nios II IDE デバッガをブート・コピア・コードを実行中のプロセッサに接続する場合は、いくつかの特別な配慮が必要です。以下の項では、ブート・コピアのデバッグの要件について説明します。

ハードウェアおよびソフトウェアのブレークポイント

ブート・コピアは、多くの場合は不揮発性メモリから実行され、これはコードで設定可能なブレークポイントの種類に影響します。Nios II デバッガが使用するブレークポイントは、ソフトウェア・ブレークポイントとハードウェア・ブレークポイントの 2 種類です。ソフトウェア・ブレークポイントは、ブレークポイント・ロケーションにあるプロセッサ命令を、実行を停止する別の命令に置き換えます。この置換方法では、ブレークポイント命令を書き込めるように、プログラム・メモリが書き込み可能であることが必要です。ブート・コピアは多くの場合フラッシュ・メモリなどの不揮発性メモリから実行されるので、ブート・コピア内にソフトウェア・ブレークポイントを設定することはできません。

ハードウェア・ブレークポイントは、命令アドレス・バス上のブレークポイントのアドレス値を検出し、ハードウェアを使用してプロセッサを停止します。したがって、ハードウェア・ブレークポイントは、揮発性メモリまたは不揮発性メモリに設定できます。フラッシュ・メモリから実行されるブート・コピアに設定できるのは、ハードウェア・ブレークポイントだけです。

ハードウェア・ブレークポイントのイネーブル

Nios II プロセッサのハードウェア・ブレークポイントをイネーブルするには、以下のステップに従います。

1. **SOPC Builder** で、システムの Nios II プロセッサをダブルクリックして Nios II ウィザードを開きます。
2. Nios II ウィザードで、**JTAG Debug Module** ページをクリックします。
3. **debugging level 2** 以上を選択します。デバッグ・レベル 2 では、Nios II デバッガが自動的に使用する 2 つの同時ハードウェア・ブレークポイントが許可されます。

main() より前での中断

ブート・コピアをデバッグする際に、main() 関数に達するまで待機しないで、リセットの直後にデバッグを開始したいことがあります。ブート・コピアには、main() 関数が全くないものもあります。これらの場合は、デバッガにプログラムのエン트리・ポイントにブレークポイントを設定するよう指示します。

デバッガの設定

ブート・コピアをデバッグするように Nios II デバッガをコンフィギュレーションするには、以下のステップに従います。

1. Nios II IDE で、デバッグするブート・コピア・プロジェクトの名前をハイライトし、**Run** メニューの **Debug** をクリックします。
2. **Debug** コンフィギュレーション・ダイアログ・ボックスで、**New** アイコンをクリックして、新しいデバッグ・コンフィギュレーションを作成します。
3. **Debugger** タブをクリックします。
4. **Download and Reset** ボックスで、以下のことを行います。
 - a. ブート・コピアがリセット時にフラッシュ・メモリから実行される場合は、**Reset target and execute from reset vector (no download)** を選択します。
 - b. ブート・コピアがオンチップ ROM から実行される場合は、**Download program to RAM** を選択します。
5. **Breakpoints at Start-up** ボックスで、**Break at program entry point** をオンにし、**Break at main()** をオフにします。**Break at main()** をオフにすると、2 つの使用可能なハードウェア・ブレークポイントのうちの 1 つが後で使用できるよう保存されます。
6. **Apply** をクリックします。

7. **Debug** ボタンをクリックして、デバグを起動します。デバグは、接続されるとブート・コピアのエントリ・ポイントで中断します。

Nios II ブート・プロセスの外部からの制御

Nios II プロセッサをブートするもう 1 つの方法は、もう 1 つのプロセッサなどの別のコンポーネントに外部からブート・プロセスを制御させることです。この状況では、外部プロセッサが何らかのソースから Nios II アプリケーションを読み出し、それを Nios II プログラム・メモリにロードします。外部プロセッサは、さまざまなソースから Nios II アプリケーション・コードを取り出すことができます。例えば、ハード・ディスクなどの不揮発性ストレージ・メディアからコードを読み出したり、イーサネット接続によりコードをダウンロードする場合があります。

外部プロセッサが Nios II アプリケーション・コードを取り出す方法は、本資料では説明しません。この項では、アプリケーション・コードを Nios II プログラム・メモリに安全にロードし、Nios II プロセッサにアプリケーションを正しく実行するよう指示するプロセスに重点を置いて説明します。

概要

外部からブート制御される Nios II システムの実装には、2 つの異なる方法を使用できます。

- 外部プロセッサが Nios II ブート・イメージをアンパックし、実行可能なアプリケーション・コードを Nios II プログラム・メモリに書き込みます。
- 外部プロセッサはブート・イメージを RAM にコピーするだけです。Nios II プロセッサが RAM からブート・イメージを受け継ぎ、ブート・イメージ自体をアンパックします。

Nios II プロセッサにブート・イメージをアンパックさせ、ブート・イメージからアプリケーションをロードさせる後者の方法は、Nios II プロセッサ上で通常のブート・コピアを実行するプロセスに似ています。唯一の相違点は、Flash Programmer がブート・イメージをフラッシュ・メモリに配置するのではなく、外部プロセッサがブート・イメージを RAM にコピーすることです。外部プロセッサが Nios II プロセッサをリセットから解放した後は、すべての動作が Nios II プロセッサがフラッシュ・メモリからブートしているかのように行われます。

この項では、外部プロセッサが Nios II ブート・イメージをアンパックし、アプリケーション・コードを Nios II プログラム・メモリにコピーし、ついで Nios II プロセッサをアプリケーションのエントリ・ポイントに導く最初の方法を絞って説明します。

外部ブート方法に関係なく共通する1つの要件は、コピーおよびアンパック・プロセス中に外部プロセッサが書き込んでいるメモリ・スペース内のいかなるコードも Nios II プロセッサが実行しないようにしなければならないことです。そうしなければ、競合状態やデータ破壊の問題が発生する可能性があります。この項で説明するプロセスは、アプリケーション・コードが Nios II プログラム・メモリにコピーされている間、Nios II プロセッサをリセット状態に保持することによって、Nios II プロセッサがコードを実行しないようにします。アプリケーション・コードがコピーされた後、Nios II はリセットから解放されてアプリケーションを実行します。

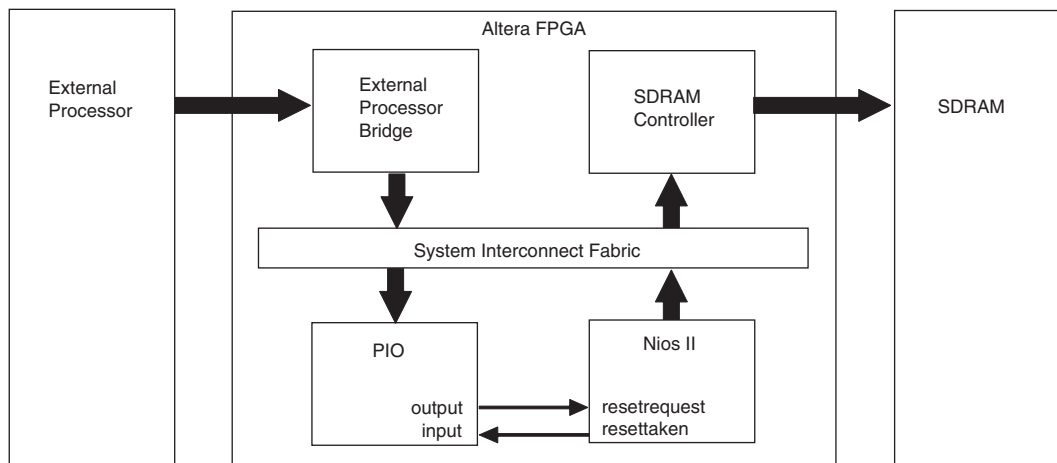
適切な SOPC Builder システムの構築

外部から制御される Nios II ブートを正常に実装するには、事前に SOPC Builder システムに必要なハードウェアが含まれていることを確認しておく必要があります。外部プロセッサは、適切なシステム・ペリフェラルにアクセスし、Nios II プロセッサのリセット状態を制御できなければなりません。以下のリストは、外部から制御される Nios II ブートをサポートするために必要な最低限のハードウェア・エレメントを示しています。

- 外部プロセッサ・ブリッジ
- 以下の機能を備えた Nios II プロセッサ：
 - cpu_resetrequest 信号
 - RAM を指すリセット・アドレス
 - 1 ビット・パラレル IO (PIO) ペリフェラル・デバイス

図 5 に、Nios II プロセッサのブートを外部から制御できるシステムのブロック図を示します。

図 5. 外部から制御される Nios II ブート・システムのブロック図



外部プロセッサ・ブリッジ

外部プロセッサが SOPC Builder システムのペリフェラルにアクセスできるようにするには、システムが Avalon ファブリックと外部プロセッサ・バス間のブリッジを備えている必要があります。

外部プロセッサへのブリッジは、IP (Intellectual Property) として入手可能ですが、内部で開発することもできます。多くの設計者は、一般に Avalon ファブリック・アーキテクチャを他のバス・プロトコルにブリッジする方が比較的簡単であるという理由から、独自の SOPC Builder 用外部プロセッサ・ブリッジ・コンポーネントを開発しています。SOPC Builder で使用可能な Component Editor ツールは、外部プロセッサ・ブリッジなどの IP を作成するのに役立ちます。



アルテラが提供しているブリッジIPのリストについては、アルテラのIP (Intellectual Property) ウェブサイト (www.altera.co.jp/products/ip/ipm-index.html) の Interface & Peripherals セクションを参照してください。

cpu_resetrequest 信号

Nios II プロセッサ v6.0 以降では、プロセッサのリセット状態を制御するために、オプションの `cpu_resetrequest` 信号を使用できます。この `cpu_resetrequest` 信号は、Nios II プロセッサしかりセットしない点で、通常の SOPC Builder システム・ワイド・リセット信号 `reset_n` とは異なります。SOPC Builder システムのリセットは動作可能な状態に保持されます。この信号は、コードが Nios II プログラム・メモリに移動される間、Nios II プロセッサをリセット状態に保持します。

`cpu_resetrequest` 信号によって、Nios II プロセッサが直ちにリセット状態に入ることはありません。`cpu_resetrequest` が High に保持されているときには、Nios II プロセッサはその時点でパイプラインに存在する命令の実行を完了してからリセットに入ります。このプロセスに要するクロック・サイクル数は不確定な場合があるので、ステータス信号 `cpu_resettaken` は、Nios II プロセッサがリセット状態に達すると High にドライブします。プロセッサは、この信号を 1 サイクルの間 High に保持します。`cpu_resettaken` 信号は、`cpu_resetrequest` 信号が High に保持されている間、継続して周期的にアサートされます。

`cpu_resetrequest` 信号をイネーブルするには、SOPC Builder で Nios II プロセッサが含まれるプロジェクトを開きます。Nios II コンポーネントをダブルクリックして Nios II ウィザードを開き、次に **Advanced Features** ページをクリックします。**Include `cpu_resetrequest` and `cpu_resettaken` signals** をオンにして、信号をイネーブルにします。これらの信号は、システム再生成後に、トップレベル SOPC Builder システムのポートになります。

Nios II リセット・アドレス

Nios II リセット・アドレスは、プロセッサがリセットから解放された後、最初に実行する命令のアドレスです。したがって、外部から制御されるブートの機能を有する Nios II システムでは、Nios II リセット・アドレスが書き込み可能なメモリ (RAM) を指している必要があります。このクラスのリセット・アドレスは一般に、従来のブート・シナリオでは不要なものですが、この項で説明する外部からのブート制御の状況では、Nios II リセット・アドレスが RAM を指していることが重要です。

Nios II プロセッサを RAM にコピーされたアプリケーション・コードに導くために、外部プロセッサは Nios II プロセッサがリセット時に実行する最初の命令 (1 つまたは複数) を書き込むことができなければなりません。そのため、Nios II リセット・アドレスが RAM を指している必要があります。一般に、リセット・アドレスに書き込まれる命令は、アプリケーションのエントリ・ポイントへの無条件分岐 (br) です。

Nios II プロセッサのリセット・アドレスとして、RAM 内の任意の未使用 32 ビット・ロケーションを選択できますが、通常は Nios II プログラム・メモリのベース・アドレス (オフセット 0x0) を選択するのが適当です。デフォルトでは、Nios II 例外テーブルがプログラム・メモリのオフセット 0x20 に配置され、アプリケーション・コードの残りの部分が連続したメモリ内の例外テーブルの後に配置されます。この配置では、オフセット 0x0 ~ 0x1C が使用できるよう確保されます。オフセット 0x0 のリセット・アドレスは、リセット・アドレスとアプリケーションのエントリ・ポイントとの差が、このプロセスが動作するために要求される 64K バイトを超えないことを保証します。差が 64K バイトを超えてはならない理由については、[40 ページ](#)の命令ステップ 4 の説明を参照してください。

1 ビット PIO ペリフェラル

1 ビット PIO ペリフェラルは、外部プロセッサから Nios II `cpu_resetrequest` 信号を制御するのに必要です。外部プロセッサは、外部プロセッサ・ブリッジを介して Avalon マップト PIO ペリフェラルにアクセスします。外部プロセッサは、PIO に値 1 を書き込んで `cpu_resetrequest` ピンをアサートするか、値 0 を書き込んでディアサートします。

外部プロセッサは、同じ PIO ペリフェラルを使用して、`cpu_resettaken` 信号の状態を読み出すこともできます。ただし、Nios II プロセッサは、一度に 1 クロック・サイクルの間しか `cpu_resettaken` 信号をアサートしません。したがって、ソフトウェアからこの信号をサンプリングして、リセットがいつ行われたかを確認しようとしても無駄です。信号は、Nios II プロセッサによる `cpu_resettaken` の有効なアサーションが外部プロセッサによってキャプチャされないように、サンプル間で容易に再アサートおよび再ディアサートできます。

SOPC Builder に付属する PIO コンポーネントには、この状況で使用するためのエッジ・キャプチャ機能が含まれています。エッジ・キャプチャ機能は、PIO の入力ポートのビットに定義済みタイプのエッジが現れるたびに、PIO のエッジ・キャプチャ・レジスタ内の対応ビットを設定します。外部プロセッサは、cpu_resetrequest をアサートした後、いつでもエッジ・キャプチャ・レジスタを読み出すことができます。cpu_resetrequest のアサーション後の任意の時点で cpu_resettaken 信号がアサートされた場合は、PIO のエッジ・キャプチャ・レジスタ内の関連ビットが設定されます。

エッジ・キャプチャ機能を使用して cpu_resettaken のアサーションを検出するようにコンフィギュレーションされた PIO コンポーネントをシステムに追加するには、以下のステップを実行します。

1. SOPC Builder でシステムを開きます。
2. **System Contents** タブで、**Peripherals** の下の **Microcontroller Peripherals** の下にある **PIO (Parallel I/O)** コンポーネントをクリックします。
3. **Add** をクリックします。
4. **PIO** ウィザードで、幅を 1 ビットに設定し、**Both input and output ports** を選択します。
5. **Input Options** タブを選択し、**Synchronously Capture** ボックスをチェックして、**Rising Edge** を選択します。
6. **Finish** をクリックして、PIO コンポーネントをシステムに追加します。

これで、Nios II cpu_resetrequest 信号をアサートし、cpu_resettaken 信号の立ち上がりエッジを検出する機能を持つ PIO コンポーネントがシステムに含まれます。



SOPC Builder は、PIO コンポーネントの入力および出力ポートを Nios II cpu_resettaken および cpu_resetrequest 信号に自動的に接続しません。SOPC Builder の生成後、Quartus II プロジェクトのトップレベルでこれらの接続を行う必要があります。

ブート・プロセス

外部で制御される Nios II ブート・プロセスの重要なハードウェア側面についての説明は以上です。この項では、外部プロセッサ上で動作中のソフトウェアの観点から、ブート・プロセス全体について説明します。

ブート・イメージ

ここで説明する手順は、読者が 6 ページの「ブート・イメージ」で説明したフォーマットの Nios II ブート・イメージを持っているものと仮定しています。

C コード例

`boot_copier_src/external_boot` ディレクトリに、Nios II プロセッサのブートを制御するために外部プロセッサで実行できる C ソース・コードのサンプルがあります。このコードには多くのコメントが含まれているため、比較的簡単に変更およびカスタマイズすることができます。このコード例では、偶然 CFI フラッシュのオフセット 0x0 からブート・イメージを取り出していますが、実際のシステムではどこからでもブート・イメージが来る可能性があります。プロセスのこの部分については、ユーザーの判断に委ねられます。

外部ブート・フロー

以下の項では、前項で述べた C コード例に実装されたブート・フローについて説明します。これらのステップは、Nios II ブート・プロセスを制御する役割を持つ外部プロセッサ上で動作するソフトウェアの観点から記述されています。

1. Nios II ブート・イメージを取り出します。

ソフトウェアは、さまざまな方法で Nios II ブート・イメージを取り出すことができます。一般的な方法としては、ハード・ディスクやフラッシュ・メモリなどの不揮発性ストレージからのブート・イメージの読み出し、イーサネット接続でのダウンロード、または RAM 内のブート・イメージのロケーションへのポインタの受け渡しなどがあります。最も重要なことは、イメージをアンパックして Nios II プログラム・メモリにコピーしようとする前に、イメージ全体にローカルでアクセスできることです。

2. 1 ビット PIO を使用して、Nios II プロセッサをリセット状態に保持します。

- PIO コンポーネントのオフセット 0x3 に任意の 32 ビット値を書き込んで、エッジ・キャプチャ・レジスタをクリアします。エッジ・キャプチャ・レジスタを使用して `cpu_resettaken` 信号がいつ High になったかを検出するには、最初にエッジ・キャプチャ・レジスタをクリアして、レジスタ値が過去に発生したエッジ・イベントを反映しないようにする必要があります。
- PIO コンポーネントのオフセット 0x0 に値 1 を書き込んで、Nios II `cpu_resetrequest` 信号をアサートします。
- 値1になるまで、PIOコンポーネントのオフセット0x3を連続してポーリングします。この値は Nios II `cpu_resettaken` 信号が High に遷移したことを示します。Nios II プロセッサが現在リセット状態にあり、ユーザーはプログラム・メモリへのアプリケーション・コードのコピーを安全に開始することができます。

3. アプリケーションをメモリ・スペースのデスティネーション・アドレスにコピーします。

6. Nios II プロセッサをリセットから解放します。

PIOペリフェラルのオフセット0x0に0を書き込んで、Nios II `cpu_resetrequest` 信号をディアサートします。Nios II プロセッサは、リセットから抜け出して、分岐命令を実行し、アプリケーションのエントリ・ポイントへ分岐し、アプリケーションの実行を開始します。

これでブートは完了です。Nios II プロセッサはオフになって動作し、これにより外部プロセッサは他のシステム・タスクを実行できます。

参考資料

このアプリケーション・ノートでは、以下の資料に関連する情報を記載または参照しています。

- 「Nios II フラッシュ・プログラマ・ユーザーガイド」
- 「Nios II ハードウェア開発チュートリアル」
- 「Nios II プロセッサ・リファレンス・ハンドブック」
- Nios II ソフトウェア開発ハンドブックの「Hardware Abstraction Layer」の章
- 「Quartus II ハンドブック Volume 4: SOPC Builder」
- 「Quartus II ハンドブック Volume 5: エンベデッド・ペリフェラル」

改訂履歴

表 3 に、このアプリケーション・ノートの改訂履歴を示します。

表 3. 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
2007 年 11 月 v1.0	初版	—



101 Innovation Drive
San Jose, CA 95134
www.altera.com
Technical Support:
www.altera.com/support/
Literature Services:
literature@altera.com

Copyright © 2007 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

