14. エンベデッド・プロセッサを利用した ISP 用 Jam STAPL の使用

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。

MII51015-1.7

はじめに

プログラマブル・ロジック・デバイス（PLD）の進歩により、革新的なイン・システム・プログラマビリティ（ISP）機能が実現しました。Jam™ STAPL（Standard Test and Programming Language）、JEDEC 規格 JESD-71 は、JTAG（Joint Test Action Group）を介して ISP を提供する現在の PLD すべてと互換性があり、イン・システム・プログラミングおよびコンフィギュレーションに対するソフトウェア・レベルのベンダに依存しない規格となっています。設計者は、Jam STAPL を使用して ISP を実装することにより、最終製品の品質、柔軟性、および製品ライフサイクルを向上させることができます。プログラミングおよびコンフィギュレーションが必要な PLD の数に関係なく、Jam STAPL はイン・フィールド・アップグレードを簡略化し、PLD のプログラミングを変革させます。

この章では、エンベデッド・システムでの Jam STAPL を使用した MAX® II デバイスのプログラミング・サポートについて説明します。

この章は、以下の項で構成されています。

- 14–1 ページの「エンベデッド・システム」
- 14–5 ページの「ソフトウェア開発」
- 14–19 ページの「Jam を使用したデバイスのアップデート」

エンベデッド・システム

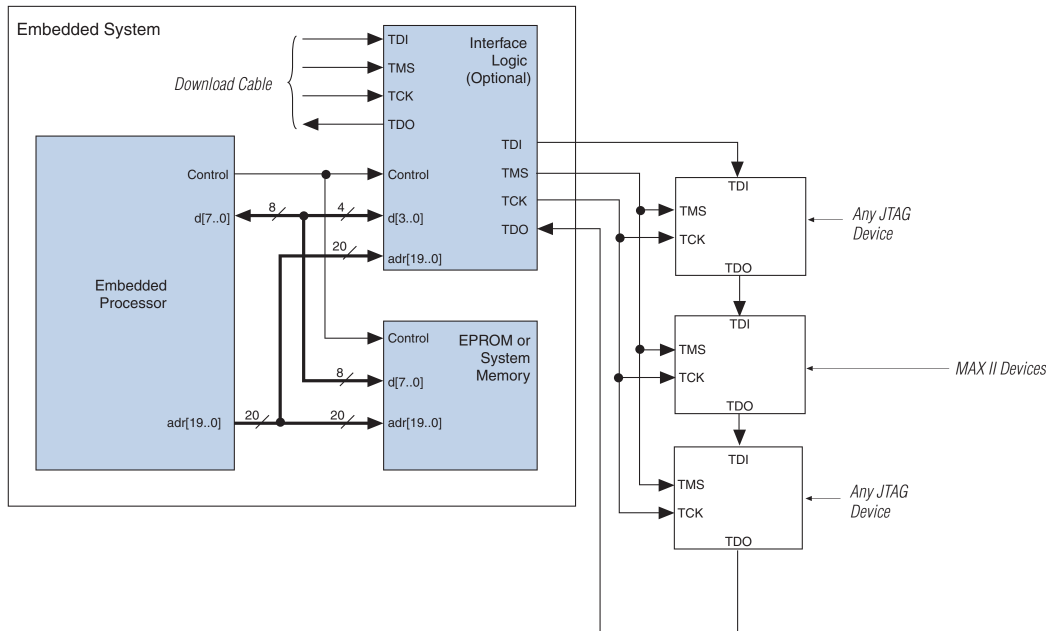
すべてのエンベデッド・システムは、ハードウェアおよびソフトウェア・コンポーネントの両方で構成されます。エンベデッド・システムを設計する場合、最初のステップはプリント基板（PCB）をレイアウトすることです。第 2 のステップは、ボードの機能を管理するファームウェアを開発することです。

エンベデッド・プロセッサへの JTAG チェインの接続

JTAG チェインをエンベデッド・プロセッサに接続するには、2 つの方法があります。最も簡単な方法は、エンベデッド・プロセッサを JTAG チェインに直接接続することです。この方法では、プロセッサのピンのうちの 4 本が JTAG インタフェース専用となるため、ボードのスペースが節約されますが、利用可能なエンベデッド・プロセッサのピン数は減ります。

図 14–1 は、第 2 の方法を示しており、インタフェース PLD を介して既存のバスに JTAG チェインを接続しています。この方法では、JTAG チェインは既存のバスでのアドレスになります。したがって、プロセッサは JTAG チェインを表すアドレスに読み出しや書き込みを行います。

図 14–1. エンベデッド・システムのブロック図



いずれの JTAG 接続方法でも、MasterBlaster™、ByteBlaster™ II、または USB-Blaster™ のヘッダ接続用にスペースを確保する必要があります。ヘッダを使用すると設計者は PLD のコンテンツを素早く検証したり修正できるため、プロトタイプ作成に役立ちます。生産時にはヘッダを取り除いてコストを削減できます。

インタフェース PLD デザイン例

図 14–2 は、インタフェース PLD の回路図の例を示します。様々なデザインが実装できますが、このデザイン例で示す重要な点は以下のとおりです。

- TMS、TCK、および TDI が同期出力
- マルチプレクサ・ロジックを搭載し、MasterBlaster、ByteBlaster II または USB-Blaster ダウンロード・ケーブルによるボード・アクセスが可能

このデザイン例は参考用です。data[3..0] を除く入力はすべてオプションであり、インタフェース PLD がエンベデッド・データ・バス上でアドレス・デコーダとして動作する方法を示すためだけに記載されています。

図 14–2. インタフェース・ロジック・デザイン例

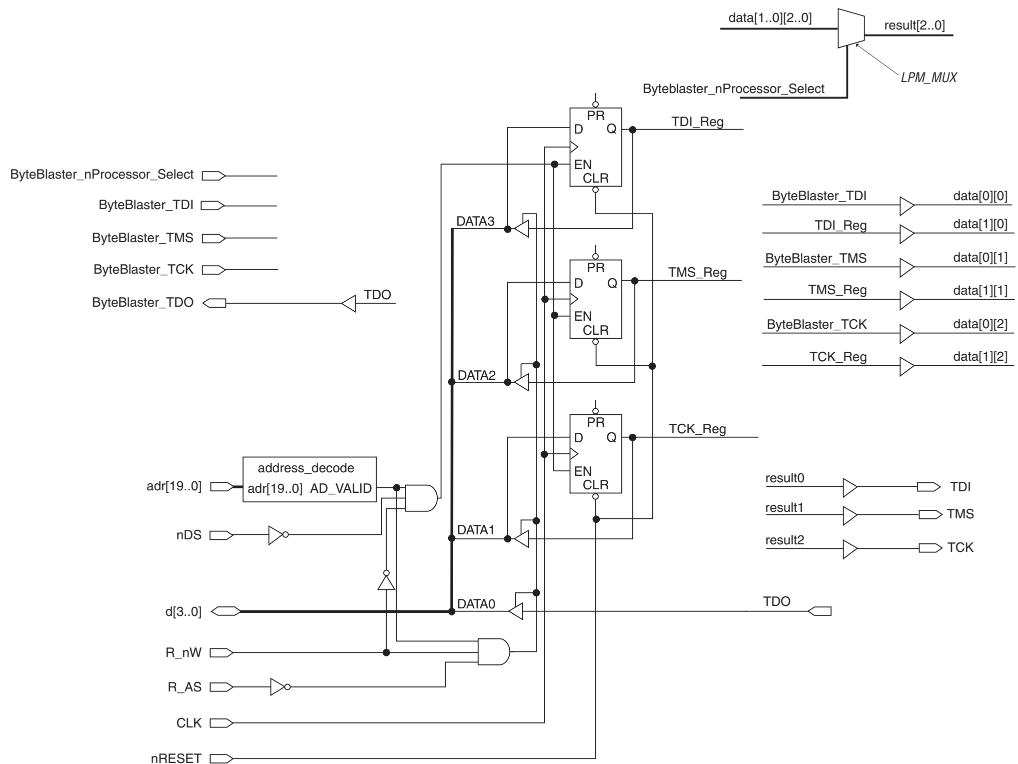


図 14–2 において、エンベデッド・プロセッサは JTAG チェインのアドレスをアサートし、R_nW 信号および R_AS 信号は、プロセッサがチェインにアクセスしようとしたときに、インタフェース PLD に通知するように設定できます。書き込みを行うには、システム・クロック（CLK）でクロックされる 3 つの D レジスタを介して、データ・パス data[3..0] を PLD の JTAG 出力に接続します。このクロックは、プロセッサが使用するクロックと同じにすることができます。同様に、読み出しを行うには、トライ・ステート・バッファをイネーブルし、TDO 信号をプロセッサに送り返す必要があります。また、このデザインでは、TDI、TMS、および TCK レジスタの値をリード・バックするためのハードウェア接続も提供します。このオプション機能を利用すると、インタフェース PLD 内のレジスタの有効なステートをソフトウェアでチェックでき、開発段階で役立ちます。さらに、マルチプレクサ・ロジックが搭載されているため、ダウンロード・ケーブルでデバイス・チェインをプログラムできます。この機能は、開発の試作段階で、プログラミングの検証が必要なときに役立ちます。

ボード・レイアウト

IEEE Std. 1149.1 JTAG チェインを介してプログラムするボードをレイアウトするときは、以下の事項が重要になります。

- TCK 信号配線パターンをクロック・ツリーとして扱うこと
- TCK にプルダウン抵抗を使用すること
- JTAG 信号配線パターンを可能な限り短くすること
- 出力が規定のロジック・レベルになるように外部抵抗を追加すること

TCK 信号配パターンの保護およびインテグリティ

TCK は、デバイスの JTAG チェイン全体に対するクロックです。これらのデバイスは TCK 信号でエッジ・トリガされるため、TCK を高周波ノイズから保護し、良好なシグナル・インテグリティを持つことが不可欠です。信号が該当するデバイス・ファミリ・データシートに記載された立ち上がり時間（t_R）および立ち下がり時間（t_F）パラメータに適合することを確認してください。また、オーバーシュート、アンダーシュート、またはリンギングを防止するために、信号に終端が必要な場合もあります。このステップは、この信号がソフトウェアで生成され、プロセッサの汎用 I/O ピンで発生するため、見落とされることがよくあります。

TCK のプルダウン抵抗

パワーアップ時に JTAG TAP（Test Access Port）を既知のステートに維持するために TCK はプルダウン抵抗を介して Low に保持することが必要です。プルダウン抵抗がないとデバイスがJTAG BSTステートでパワーアップし、それによってボード上で競合が発生する可能性があります。一般的な抵抗値は 1 kΩ です。

JTAG 信号の配線パターン

JTAG 信号の配線パターンを短くすると、ノイズやドライブ強度に関連した問題の解消に役立ちます。TCK ピンと TMS ピンには特別に注意が必要です。TCK と TMS は JTAG チェインのすべてのデバイスに接続されるため、これらのトレースは、TDI や TDO よりも負荷が大きくなります。JTAG チェインの長さと負荷によっては、プロセッサとの間で信号がインテグリティを維持しながら伝播できるように、いくつかの追加バッファリングが必要になることがあります。

外部抵抗

プログラミング中に出力を定義済みロジック・レベルにするには、出力ピンに外部抵抗を追加する必要があります。出力ピンは、プログラミング中にはトライ・ステートになります。また、MAX® II デバイスでは、ピンはウィーク・プルアップ抵抗によってプルアップされます。センシティブな入力ピンをドライブする出力は、外部抵抗を使用して適切なレベルに接続することを推奨します。

前述の各ボード・レイアウト事項は、特にシグナル・インテグリティなどのさらなる分析が必要になることがあります。場合によっては、ディスクリート・バッファを使用するかを判断するために、JTAG チェインの負荷とレイアウトを解析する必要があります。

詳細は、「MAX II デバイス・ハンドブック」の「MAX II デバイスのイン・システム・プログラマビリティ・ガイドライン」の章を参照してください。

ソフトウェア開発

アルテラのエンベデッド・プログラミングでは、標準化された Jam Player ソフトウェアとともに、Quartus® II ソフトウェア・ツールからの Jam ファイル出力を使用します。Jam ファイルには、MAX II デバイスをプログラムするためのすべてのデータが含まれているため、これらのツールの開発時に設計者が行う作業は最小ですみます。開発時間の大部分は、Jam Player をホスト・エンベデッド・プロセッサへ移植する作業に費やされます。

Jam Byte-Code Player の移植について詳しくは、14–11 ページの「Jam STAPL Byte-Code Player の移植」を参照してください。

Jam ファイル（.jam および .jbc）

アルテラは以下のタイプの Jam ファイルをサポートしています。

- ASCII テキスト・ファイル（**.jam**）
- Jam Byte-Code ファイル（**.jbc**）

ASCII テキスト・ファイル（.jam）

アルテラは以下の 2 つのタイプの Jam ファイルをサポートしています。

- JEDEC Jam STAPL フォーマット
- Jam バージョン 1.1（pre-JEDEC フォーマット）

JEDEC Jam STAPL フォーマットは、JEDEC Standard JESD-71A 規格で規定された構文を使用します。アルテラは、すべての新規プロジェクトで JEDEC Jam STAPL ファイルの使用を推奨しています。ほとんどの場合、Jam ファイルはテスタ環境で使用されます。

Jam Byte-Code ファイル（.jbc）

JBC ファイルは、Jam ファイルのコンパイルされたバージョンであるバイナリ・ファイルです。JBC ファイルは仮想プロセッサ・アーキテクチャにコンパイルされ、そこで ASCII Jam コマンドは仮想プロセッサ互換のバイト・コード命令にマップされます。JBC ファイルには、以下の 2 つのタイプがあります。

- Jam STAPL Byte-Code（JEDEC Jam STAPL ファイルのコンパイルされたバージョン）
- Jam Byte-Code（Jam バージョン 1.1 ファイルのコンパイルされたバージョン）

Jam STAPL Byte-Code ファイルは使用メモリが最小になるため、アルテラはエンベデッド・アプリケーションではこのファイルの使用を推奨します。

Jam ファイルの生成

Quartus II ソフトウェアは、Jam および JBC ファイル・タイプを生成できます。さらに Jam ファイルは、スタンドアロン Jam Byte-Code コンパイラによって、JBC ファイルにコンパイルできます。コンパイラは機能的に等価な JBC ファイルを作成します。

JBC ファイルは、直接 Quartus II ソフトウェアから簡単に生成できます。ソフトウェア・ツールは、1 つまたは複数の JBC ファイルから複数のデバイスのプログラミングとコンフィギュレーションをサポートしています。図 14–3 および 14–4 は、Quartus II ソフトウェアでデバイス・チェインおよび JBC ファイルの生成を指定するダイアログ・ボックスです。

図 14–3. Quartus II ソフトウェアの Programmer ウィンドウのマルチ･デバイス JTAG チェインの名前およびシーケンス

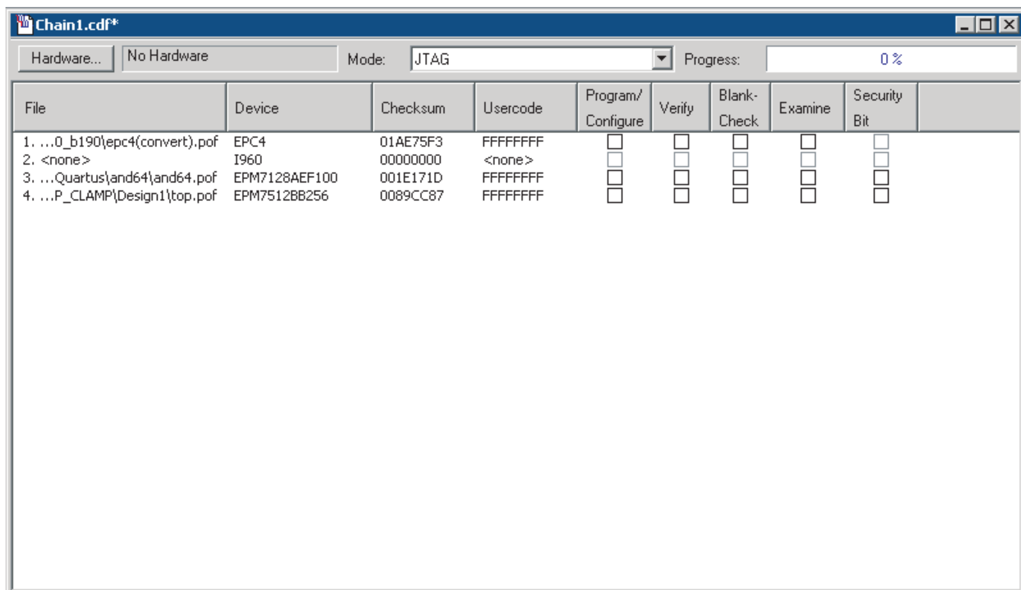
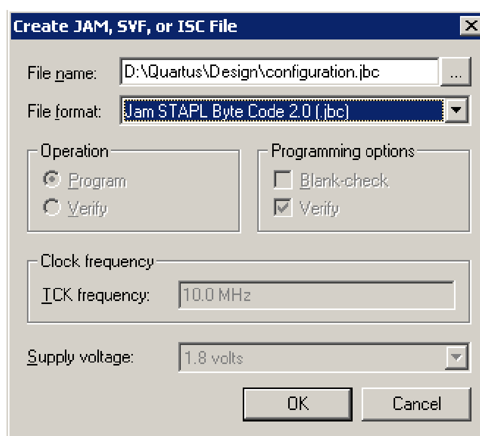


図 14–4. Quartus II ソフトウェアにおけるマルチ・デバイス JTAG チェイン用 JBC ファイルの生成



以下のステップで、Quartus II ソフトウェアを使用して JBC ファイルを生成する方法を説明します。

1. Tools メニューの **Programmer** をクリックします。
2. **Add File** をクリックし、デバイスごとにプログラミング・ファイルを選択します。
3. File メニューの **Create/Update** をポイントし、**Create JAM, SVF, or ISC File** をクリックします。図 14–4 を参照してください。
4. File format リストで Jam STAPL Byte-Code ファイルを指定します。
5. **OK** をクリックします。

JTAG チェインには、アルテラ・デバイスとアルテラ以外の JTAG 準拠デバイスの両方を追加できます。Programming File Names フィールドでプログラミング・ファイルを指定しなかった場合、JTAG チェインのデバイスはバイパスされます。

MAX II ユーザー・フラッシュ・メモリ用 Jam ファイルの生成

Quartus II Programmer には、デバイス全体、ロジック・アレイ、またはユーザー・フラッシュ・メモリ（UFM）ブロックを個別にターゲットとするオプションがあります。UFM セクションはロジック・アレイから独立してプログラミングできるため、個別の Jam STAPLE および JBC オプションをコマンド・ラインで使用して、UFM およびコンフィギュレーション・フラッシュ・メモリ（CFM）ブロックを個別にプログラムできます。詳細は、14–21 ページの「MAX II Jam/JBC アクションおよびプロシージャ・コマンド」を参照してください。

Jam Player

Jam Player は、Jam ファイル内の記述的情報を読み出し、これらの情報をターゲット PLD をプログラムするデータに変換します。Jam Player は特定のデバイス･アーキテクチャやベンダをプログラムするのではなく、Jam ファイル仕様で定義された構文の読み出しを解釈のみを行います。フィールドでの変更は、Jam Player ではなく Jam ファイルに限定されます。その結果、イン・フィールド・アップグレードのたびに、Jam Player のソース・コードを修正する必要はありません。

2 種類の Jam ファイルに対応するために、ASCII Jam STAPL Player と Jam STAPL Byte-Code Player の 2 種類の Jam Player があります。この章に記載の全般的な概念は、どちらのタイプの Player にも当てはまりますが、以下の内容は Jam STAPL Byte-Code Player を対象としています。

Jam STAPL および JBC ファイルは、MAX II UFM ブロックの一方のみまたは両方のセクタをターゲットとして生成できるため、Jam Player を使用して、MAX II コンフィギュレーション・フラッシュ・メモリ・ブロックおよび UFM ブロックを別々にプログラムしたり、書き込むことができます。

Jam Player の互換性

エンベデッド Jam Player は、標準 JEDEC ファイル・フォーマットに準拠したJamファイルを読み込むことができます。エンベデッドJam Player は、バージョン 1.1 の構文を使用する従来の Jam ファイルと互換性があります。どちらの Player も下位互換性があり、バージョン 1.1 のファイルと Jam STAPL ファイルを実行できます。

バージョン 1.1 の構文に対するアルテラのサポートについて詳しくは、「AN 122: Using Jam STAPL for ISP & ICR via an Embedded Processor」を参照してください。

Jam STAPL Byte-Code Player

Jam STAPL Byte-Code Player は、16 ビットおよび 32 ビット・プロセッサ用に C プログラミング言語でコーディングされています。

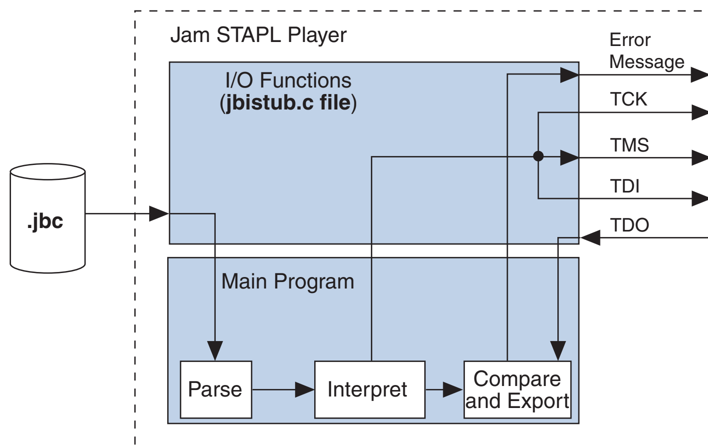
8 ビット・プロセッサに対するアルテラのサポートについて詳しくは、「AN 111: Embedded Programming Using the 8051 & Jam Byte-Code」を参照してください。

16 ビットおよび 32 ビットのソース・コードは、以下の 2 つのカテゴリに分類されます。

- I/O ファンクションを処理し、特定のハードウェアに適用されるプラットフォーム固有のコード（**jbistub.c**）
- Player の内部ファンクションを実行する汎用コード（他のすべての C ファイル）

図 14–5 に、ファンクションによるソース・コード・ファイルの構成を示します。プラットフォーム固有のコードが **jbistub.c** ファイル内に管理されているため、Jam STAPL Byte-Code Player を特定のプロセッサに移植するプロセスが簡略化されます。

図 14–5. Jam STAPL Byte-Code Player ソース・コードの構造



Jam STAPL Byte-Code Player の移植

jbistub.c ファイルのデフォルト・コンフィギュレーションには、DOS、32 ビット Windows、および UNIX 用のコードが含まれているため、ソース・コードを簡単にコンパイルして、これらの定義済みオペレーティング･システムの機能性評価とデバッグを行うことができます。エンベデッド環境の場合、このコードは単一のプリプロセッサ `#define` 文を使用して簡単に除去できます。さらに、コードを移植するには、**jbistub.c** ファイルのコードの特定部分にわずかな変更が必要です。

Jam Player を移植するには、表 14–1 に示す **jbistub.c** ファイルのいくつかのファンクションをカスタマイズする必要があります。

表 14–1. カスタマイズを必要とするファンクション

ファンクション	説明
`jbi_jtag_io()`	4 つの IEEE 1149.1 JTAG 信号、`TDI`、`TMS`、`TCK`、および `TDO` にインタフェースします。
`jbi_export()`	UES（User Electronic Signature）などの情報を呼び出し側のプログラムに渡します。
`jbi_delay()`	実行中に必要なプログラミング・パルスまたは遅延を実装します。
`jbi_vector_map()`	非 IEEE 1149.1 JTAG 信号に対して信号からピンへのマップを処理します。
`jbi_vector_io()`	VECTOR MAP で定義されるとおり、非 IEEE 1149.1 JTAG 信号をアサートします。

必要なコードをすべてカスタマイズしたことを確認するために。以下の 4 つのステップを実行します。

1. プリプロセッサのステートメントを設定して、無関係なコードを除外する。
2. JTAG 信号をハードウェア・ピンにマップする。
3. `jbi_export()` からのテキスト・メッセージを処理する。
4. 遅延較正をカスタマイズする。

ステップ **1:** プリプロセッサのステートメントを設定して、無関係なコードを除外する

jbistub.c の先頭で、デフォルトの PORT パラメータを EMBEDDED に変更して、すべての DOS、Windows、UNIX のソース・コード、およびインクルードされたライブラリを除外します。

```
#define PORT EMBEDDED
```

ステップ **2: JTAG** 信号をハードウェア・ピンにマップする

jbi_jtag_io() ファンクションには、バイナリ・プログラミング・データを送受信するコードが含まれています。4 つの JTAG 信号はそれぞれ、エンベデッド・プロセッサのピンに再マップする必要があります。デフォルトでは、ソース・コードは PC のパラレル・ポートに書き込みます。jbi_jtag_io() 信号は、図 14–6 に示す PC パラレル・ポート・レジスタに JTAG ピンをマップします。

図 14–6. デフォルトの PC パラレル・ポート信号マップ 注 (1)

7	6	5	4	3	2	1	0	I/O Port
0	TDI	0	0	0	0	TMS	TCK	OUTPUT DATA - Base Address
TDO	X	X	X	X	---	---	---	INPUT DATA - Base Address + 1

図 14–6 の注 :

(1) PC パラレル・ポート・ハードウェアは、最上位ビットの TDO を反転させせます。

以下の jbi_jtag_io() ソース・コードでは、マップ処理を強調して示します。

```
int jbi_jtag_io(int tms, int tdi, int read_tdo)
{
    int data=0;
    int tdo=0;

    if (!jtag_hardware_initialized)
    {
        initialize_jtag_hardware();
        jtag_hardware_initialized=TRUE;
    }
    data = ((tdi?0x40:0)|(tms?0x2:0));
/*TDI,TMS*/
    write_byteblaster(0,data);
    if (read_tdo)
    {
        tdo=(read_byteblaster(1)&0x80)?0:1;   /*TDO*/
    }
    write_blaster(0,data|0x01);              /*TCK*/
    write_blaster(0,data);
    return (tdo);
}
```

前のコードで、PC パラレル・ポートは TDO の実際の値を反転させせます。jbi_jtag_io() ソース・コードは、この値を再度反転させて元のデータを回復します。TDO の値を反転させる行は、以下のとおりです。

```
tdo=(read_byteblaster(1)&0x80)?0:1;
```

ターゲット・プロセッサが TDO を反転させない場合、コードは以下のように記述します。

```
tdo=(read_byteblaster(1)&0x80)?1:0;
```

信号を正しいアドレスにマップするには、左シフト（<<）または右シフト（>>）演算子を使用します。例えば、TMS と TDI がそれぞれポート 2 とポート 3 の場合、コードは以下のようになります。

```
data=(((tdi?0x40:0)>>3)|((tms?0x02:0)<<1));
```

TCK および TDO にも同じ手法を適用します。

read_byteblaster および write_byteblaster 信号はそれぞれ、**conio.h** ライブラリの inp() および outp() ファンクションを使用して、ポートの読み出しと書き込みを行います。これらのファンクションが利用できない場合は、同等のファンクションで代用する必要があります。

ステップ **3: jbi_export( )** からのテキスト・メッセージを処理する

jbi_export() ファンクションは、printf() ファンクションを使用して、テキスト・メッセージを stdio に送信します。Jam STAPL Byte-Code Player は jbi_export() 信号を使用して、オペレーティング・システムまたは Player を呼び出すソフトウェアに情報（デバイスの UES または USERCODE など）を渡します。ファンクションはテキスト（文字列形式）と数値（10 進整数形式）を渡します。

これらの用語の定義については、「AN 39: IEEE 1149.1 (JTAG) Boundary-Scan Testing in Altera Devices」を参照してください。

stdout が利用できるデバイスが存在しない場合、情報はファイルまたはストレージ・デバイスにリダイレクトされるか、あるいは Player を呼び出すプログラムに変数として渡されます。

ステップ **4:** 遅延較正をカスタマイズする

calibrate_delay() ファンクションは、ホスト・プロセッサが 1 ミリ秒間に実行するループ数を決定します。プログラミングとコンフィギュレーションで正確な遅延が使用されるため、この較正は重要です。デフォルトでは、この数値は 1 ミリ秒あたり 1,000 ループとしてハードコード化され、以下のように表されます。

```
one_ms_delay = 1000
```

このパラメータが既知の場合、それに従って変更します。既知でない場合は、Windows および DOS プラットフォームに同様のコードを使用できます。1 つの while ループの実行に必要な時間分のクロック・サイクル数をカウントするプラットフォームのためのコードが含まれています。このコードは、遅延の基準となる正確な結果を得るために、複数回のテストを通じてサンプリングされ、平均化されます。この手法の利点は、ホスト・プロセッサの速度に基づいて較正を変更できることです。

Jam STAPL Byte-Code Player が移植され動作した後、ターゲット・デバイスでの JTAG ポートのタイミングとスピードを検証してください。MAX II デバイスのタイミング・パラメータは、「MAX II デバイス・ハンドブック」の「DC およびスイッチング特性」の章に記載された値に準拠する必要があります。

Jam STAPL Byte-Code Player がタイミング仕様で動作しない場合は、適切な遅延でコードを最適化する必要があります。タイミング違反は、プロセッサが非常に高性能で、18 MHz を超える高速レートで TCK を生成できる場合にのみ発生します。

jbistub.c file を除く他のファイルのソース・コードはデフォルト状態から変更しないことを強く推奨します。これらのファイルのソース・コードを変更すると、Jam Player の動作は予測不能になります。

Jam STAPL Byte-Code Player のメモリ使用量

Jam STAPL Byte-Code Player は、予測可能な方法でメモリを使用します。この項では、ROM および RAM メモリの使用量を見積もる方法を示します。

ROM 使用量の見積もり

Jam Player および JBC ファイルの格納に必要な ROM の最大容量を見積もるには、以下の式を使用します。

ROM $_{サイズ}$ = JBC ファイル・サイズ + Jam Player サイズ

JBC ファイル・サイズは、プログラミング・データの格納に必要なメモリ容量とプログラミング・アルゴリズムに必要なスペースの 2 つのカテゴリに分割できます。JBC ファイル・サイズの見積もりには、以下の式を使用します。

$$\text{JBC ファイル・サイズ} = Alg + \sum_{k=1}^{N} Data$$

ここで、

Alg = アルゴリズムで使用されるスペース
Data = 圧縮されたプログラミング・データで使用されるスペース
k = ターゲットとなるデバイスを表すインデックス
N = チェイン内のターゲット・デバイスの数

この式によって JBC ファイルが見積もられ、この値はデバイスの利用率によって ±10% 変動することがあります。デバイス利用率が低い場合、ファイル・サイズを最小化する圧縮アルゴリズムは、繰り返しデータを検出する可能性が高いため、JBC ファイル・サイズが小さくなる傾向があります。

この式は、アルゴリズム・サイズが 1 つのデバイス・ファミリに対しては一定となるが、プログラミング・データ・サイズは、ターゲットとするデバイスが増えるほど増大することも示しています。デバイス・ファミリでは、JBC ファイル・サイズ（データ・コンポーネントによる）の増加は線形となります。

表 14–2 に、1 個の MAX II デバイスをターゲットとする場合のアルゴリズム・ファイル・サイズ定数を示します。

表 14–2. 1 つのアルテラ・デバイス・ファミリをターゲットとするアルゴリズム・ファイル・サイズ定数

デバイス	標準的な JBC ファイル・アルゴリズム・サイズ（KB）
MAX II	24.3

表 14–3 に、ISP 用の Jam 言語をサポートする MAX II デバイスのデータ・サイズ定数を示します。

表 14–3. データ定数

デバイス	標準的な Jam STAPL Byte-Code のデータ・サイズ（KB）	
	圧縮	非圧縮 (1)
EPM240	12.4 (2)	12.4 (2)
EPM570	11.4	19.6
EPM1270	16.9	31.9
EPM2210	24.7	49.3

表 14–3 の注 :
(1) 非圧縮プログラミング・データを使用した JBC ファイルの生成方法について詳しくは、アルテラ・アプリケーションにお問い合わせください。
(2) JBC コンパイラでは、圧縮アレイに 64 K ビットの最小サイズ制限があります。64 K ビット（8 K バイト）より小さいプログラミング・データ・アレイは圧縮されません。EPM240 のプログラミング・データ・アレイはこの制限値より小さいため、JBC ファイルは常に圧縮されません。この制限の理由は、圧縮解除にメモリ・バッファが必要であり、小さなエンベデッド・システムの場合はアレイを圧縮解除するよりも、圧縮されていない小さなアレイを直接使用するほうが効率的であるからです。

JBC ファイルを見積もった後、表 14–4 に示す情報を使用して Jam Player サイズを見積もります。

表 14–4. Jam STAPL Byte-Code Player のバイナリ・サイズ

構築	説明	サイズ（KB）
16 ビット	MasterBlaster または ByteBlasterMV ダウンロード・ケーブルを使用する Pentium/486	80
32 ビット	MasterBlaster または ByteBlasterMV ダウンロード・ケーブルを使用する Pentium/486	85

ダイナミック・メモリ使用量の見積もり

Jam Player が必要とする DRAM の最大容量を見積もるには、以下の式を使用します。

$$\text{RAMサイズ} = \text{JBCファイル・サイズ} + \sum_{k=1}^{N} Data\ (\text{非圧縮データ・サイズ})_k$$

JBC ファイル・サイズは、シングル・デバイスまたはマルチ・デバイスの式で求められます（14–15 ページの「ROM 使用量の見積もり」を参照）。

Jam Playerga 使用する RAM の容量は、JBC ファイルのサイズにターゲットとする各デバイスに必要なデータの合計を加算したものです。JBC ファイルが圧縮データを使用して生成される場合、データを解答して一時的に格納するために、Player によって一部の RAM が使用されます。表 14–3 に、非圧縮データ・サイズを示します。非圧縮 JBC ファイルが使用される場合は、以下の式を使用します。

RAM サイズ = JBC ファイル・サイズ

スタックおよび蓄積のためのメモリ要件は、Jam STAPL Byte-Code Player が使用する全メモリ容量に関しては無視できます。スタックの最大の深さは、**jbimain.c** ファイル内の `JBI_STACK_SIZE` パラメータによって設定されます。

メモリ見積もり例

以下の例では、16 ビット Motorola 68000 プロセッサを使用して、IEEE Std. 1149.1 JTAG チェイン内の EPM7128AE および EPM7064AE デバイスを圧縮するデータを使用する JBC ファイルでプログラムします。メモリ使用量を算出するには、必要な ROM の容量を求めてから RAM の使用量を見積もります。Jam Byte-Code Player が必要とする DRAM の容量を見積もるには、以下のステップを実行します。

1. JBC ファイル・サイズを算出します。以下のマルチ・デバイスの式を使用して、JBC ファイル・サイズを見積もります。JBC ファイルは圧縮データを使用するため、表 14–3 に示す圧縮データのファイル・サイズ情報を使用して *Data* サイズを算出します。

   $$\text{JBC ファイル・サイズ} = Alg + \sum_{k=1}^{N} Data$$

   ここで、

   Alg = 21 Kbytes
   $Data$ = EPM7064AE $Data$ + EPM7128AE $Data$ = 8 + 4 = 12 Kbytes

   したがって、JBC ファイル・サイズは 33 K バイトになります。

2. JBC Player のサイズを見積もります。この 68000 は 16 ビット・プロセッサなので、この例では 62 K バイトの JBC Player サイズを使用します。以下の式を使用して、必要な ROM 容量を算出します。

   ROM サイズ = JBC ファイル・サイズ + Jam Player サイズ

   ROM サイズ = 95 Kbytes

3. 以下の式で、RAM 使用量を見積もります。

   $$\text{RAM サイズ} = 33\ \text{Kbytes} + \sum_{k=1}^{N} Data\ (\text{非圧縮データ・サイズ})_{k}$$

   JBC ファイルは圧縮データを使用するため、使用される全 RAM 容量を算出するには、各デバイスの非圧縮データのサイズを合計する必要があります。非圧縮データ・サイズの定数は以下のとおりです。

- EPM7064AE = 8 Kbytes
- EPM7128AE = 12 Kbytes
- DRAM の全使用量を以下のとおり計算します。
- RAM サイズ = 33 Kbytes + (8 Kbytes + 12 Kbytes) = 53 Kbytes

一般に、Jam ファイルは ROM より RAM を多く使用します。RAM の方が安価であり、多数のデバイスがプログラムされるほど簡単なアップグレードを実現するために要求される全体的なコストが低下されるため、これは好ましい傾向です。ほとんどのアプリケーションでは、メモリ・コストよりもアップグレードの容易さの方が重要です。

Jam を使用したデバイスのアップデート

フィールドでのデバイスのアップデートとは、多くの場合は「プログラム」動作によって新しい JBC ファイルをダウンロードし、Jam STAPL Byte-Code Player を実行することを意味します。

Player 実行のためのメイン・エントリ・ポイントは、`jbi_execute()` です。このルーチンは特定の情報を Player に渡します。Player が終了すると、終了コードが返され、併せてランタイム・エラーがあればそれにかんする詳細なエラー情報が返されます。インタフェースは、ルーチンのプロトタイプ定義で定義されます。

```
JBI_RETURN_TYPE jbi_execute
(
    PROGRAM_PTR program
    long program_size,
    char *workspace,
    long workspace_size,
    *action,
    char **init_list,
    long *error_line,
    init *exit_code
)
```

`jbi_execute()` に渡される変数は、**jbistub.c** にある `main()` 内のコードで決定されます。ほとんどの場合、このコードはエンベデッド環境には適用できません。したがって、このコードを削除し、エンベデッド環境用に `jbi_execute()` ルーチンを設定することができます。表 14–5 に、各パラメータを説明します。

表 14–5. パラメータ 注 (1)

パラメータ	ステータス	説明
program	必須	JBC ファイルへのポインタ。大部分のエンベデッド・システムでは、このパラメータは jbi_execute() を呼び出す前にポインタにアドレスを割り当てるのと同じように容易に設定できます。
program_size	必須	JBC ファイルが占有するメモリ容量（バイト単位)。
workspace	オプション	JBC Player が必要なファンクションの実行に使用できるダイナミック・メモリへのポインタ。このパラメータの目的は、Player のメモリ使用を定義済みのメモリ空間に限定することです。このメモリは、jbi_execute() を呼び出す前に割り当てる必要があります。ダイナミック・メモリの最大使用量が問題でない場合、このパラメータは null に設定します。それによって、Player は必要なメモリを動的に割り当てて、特定のアクションを実行することができます。
workspace_size	オプション	workspace が指すメモリ容量を（バイト単位）を表すスケーラ。
action	必須	文字列（Player に指示するテキスト）へのポインタ。action の例に PROGRAM や VERIFY があります。ほとんどの場合、このパラメータは文字列 PROGRAM に設定されます。Player では大文字と小文字が区別されないため、テキストは大文字と小文字のどちらでも構いません。Player は、「Jam Standard Test and Programming Language Specification」で定義されるすべてのアクションをサポートしています。表 15–6 を参照してください。文字列は null で終了する必要があることに注意してください。
init_list	オプション	文字列へのポインタの配列。このパラメータは、Jam バージョン 1.1 ファイルを適用する場合に使用します。(2)
error_line	—	長い整数へのポインタ。実行中にエラーが発生した場合、Player はエラーが発生した JBC ファイルの行を記録します。
exit_code	—	長い整数へのポインタ。JBC ファイルの構文または構造に関するエラーが発生した場合、コードを返します。このようなエラーが発生した場合は、サポートしているベンダに問い合わせて、終了コードが発生した状況を詳しく説明する必要があります。

表 14–5 の注：
(1) Player を実行するために、必須パラメータを渡す必要があります。
(2) 詳細は、「AN 122: Using Jam STAPL for ISP & ICR via an Embedded Processor」を参照してください。

MAX II Jam/JBC アクションおよびプロシージャ・コマンド

Jam/JBC でサポートされている MAX II デバイス用動作コマンドを、定義も含めて表 14–6 に示します。各動作で実行可能なオプションの手順を定義と併せて、表 14–7 に示します。

表 14–6. MAX II Jam/JBC アクション

Jam/JBC アクション	説明	オプションのプロシージャ（デフォルトではオフ）
PROGRAM	デバイスをプログラムします。CFM および UFM をオプションで個別にプログラムできます。	DO_BYPASS_CFM DO_BYPASS_UFM DO_SECURE DO_REAL_TIME_ISP DO_READ_USERCODE
BLANKCHECK	デバイス全体をブランク・チェックします。CFM および UFM をオプションで個別にブランク・チェックできます。	DO_BYPASS_CFM DO_BYPASS_UFM DO_REAL_TIME_ISP
VERIFY	Jam ファイルのプログラミング・データに対してデバイス全体を検証します。CFM および UFM をオプションで個別に検証できます。	DO_BYPASS_CFM DO_BYPASS_UFM DO_REAL_TIME_ISP DO_READ_USERCODE
ERASE	デバイスのプログラミング・コンテンツを消去します。CFM および UFM をオプションで個別に消去できます。	DO_BYPASS_CFM DO_BYPASS_UFM DO_REAL_TIME_ISP
READ_USERCODE	Quartus II ソフトウェアの Assignments メニュー -> Device -> Device and Pin options -> General タブで USERCODE データを入力できるので、プログラミング・ファイル内の READ_USERCODE の返り値には特定の値を設定できます。	—

表 14–7. MAX II Jam/JBC オプションのプロシージャの定義

プロシージャ	説明
DO_BYPASS_CFM	set =1 のとき、DO_BYPASS_CFM は CFM をバイパスし、指定されたアクションを UFM だけで実行します。set =0 のとき、このオプションは無視されます（デフォルト）。
DO_BYPASS_UFM	set =1 のとき、DO_BYPASS_UFM は UFM をバイパスし、指定されたアクションを CFM だけで実行します。set =0 のとき、このオプションは無視されます（デフォルト）。
DO_BLANKCHECK	set =1 のとき、デバイス、CFM、または UFM は、ブランク・チェックされます。set =0 のとき、このオプションは無視されます（デフォルト）。
DO_SECURE	set =1 のとき、デバイスのセキュリティ・ビットは設定されます。セキュリティ・ビットは、CFM データにのみ影響します。UFM は保護することができません。set =0 のとき、このオプションは無視されます（デフォルト）。
DO_REAL_TIME_ISP	set =1 のとき、リアルタイム ISP 機能は、ISP アクションが実行されるためにイネーブルされます。set =0 のとき、デバイスはあらゆる動作のためのノーマル ISP モードを使用します。
DO_READ_USERCODE	set =1 のとき、デバイスからの JTAG USERCODE レジスタ情報を返します。

コマンド・プロンプトから Jam ファイルを実行するには、以下の例に示すように、-a オプションを使用して動作を指定する必要があります。

```
jam -aPROGRAM <filename>
```

このコマンドは filename に指定されている Jam ファイルで、MAX II デバイス全体をプログラムします。

以下の例に示すように、-a オプションを使用して、関連動作によりオプションの手順を実行できます。

```
jam -aPROGRAM -dDO_BYPASS_UFM=1
        -dDO_REAL_TIME_ISP=1 <filename>
```

このコマンドは、リアルタイム ISP がイネーブルされている場合にのみ（すなわち、プロセス全体でデバイスがユーザー・モードになったまま）、MAX II CFM ブロックをプログラムします。

JBC プレイヤは実行コマンド各を除いて、同じ形式を使用します。

Player は、JBI_RETURN_TYPE または整数型のステータス・コードを返します。この値は、アクションが成功した（「0」を返す）かどうかを示します。「Jam Standard Test and Programming Language Specification」で定義されるとおり、jbi_execute() は、以下の表表 14–8 に示す終了コードのいずれか 1 つを返すことができます。

表 14–8. 終了コード

終了コード	説明
0	成功
1	チェインのチェックの失敗
2	IDCODE の読み出しの失敗
3	USERCODE の読み出しの失敗
4	UESCODE の読み出しの失敗
5	ISP への移行の失敗
6	認識されないデバイス ID
7	デバイスのバージョンがサポートされていない
8	消去の失敗
9	ブランク・チェックの失敗
10	プログラミングの失敗
11	検証の失敗
12	読み出しの失敗
13	チェックサム計算の失敗
14	セキュリティ・ビット設定の失敗
15	セキュリティ・ビット照会の失敗
16	ISP 終了の失敗
17	システム・テスト実行の失敗

Jam STAPL Byte-Code Player の実行

Jam STAPL Byte-Code Player の呼び出しは、その他のサブルーチンの呼び出しと類似しています。この場合、サブルーチンはアクションとファイル名が指定され、その関数を実行します。

イン・フィールド・アップグレードは、現在のデバイス・デザインが最新かどうかによって実行できる場合があります。多くの場合、JTAG USER CODE は、PLD デザインのリビジョンを示す電子「スタンプ」として使用されます。USERCODE が古い値に設定されると、エンベデッ

ド・ファームウェアはデバイスをアップデートします。以下の擬似コードは、Jam Byte-Code Player を複数回呼び出して、ターゲット PLD をアップデートする方法を示しています。

```
result = jbi_execute(jbc_file_pointer, jbc_file_size,
0, 0, “READ_USERCODE”, 0, error_line, exit_code);
```

ここで Jam STAPL Byte-Code Player は JTAG USERCODE を読み出し、`jbi_export()` ルーチンを使用してこれをエクスポートします。次に、コードはその結果に基づいて分岐できます。

以下に、Jam Player を使用するコードの例を示します。

```
switch (USERCODE)
{
    case "0001":   /*Rev 1 is old - update to new Rev*/
       result = jbi_execute (rev3_file, file_size_3,
        0, 0, “PROGRAM”, 0, error_line, exit_code);
    case "0002":   /*Rev 2 is old - update to new Rev*/
       result = jbi_excecute(rev3_file, file_size_3,
        0, 0, "PROGRAM", 0, error_line, exit_code);
    case "0003":
       ;           /*Do nothing - this is the current
        Rev*/
    default:      /*Issue warning and update to current
Rev*/
       Warning - unexpected design revision;
        /*Program device with newest rev anyway*/
       result = jbi_execute(rev3_file, file_size_3, 0,
        0, "PROGRAM", 0, error_line, exit_code);
}
```

switch 文を使用すると、どのデバイスにアップデートが必要で、どのデザイン・リビジョンを使用するかを決定できます。Jam STAPL Byte-Code ソフトウェア・サポートによって、PLD アップデートはコードに数行追加するのと同じくらい簡単なものになります。

まとめ

Jam STAPL を使用すると、ISP の利点を簡単に活用できます。Jam は、小さなファイル・サイズ、使いやすさ、プラットフォームからの独立性など、必要なエンベデッド・システムの要件のすべてを満たします。アップデートを Jam STAPL Byte-Code ファイルに限定することによって、イン・フィールド・アップグレードが簡単になります。Jam Player の実行は、使用されるリソースの計算と同様に簡単です。最新のアップデートおよび情報については、www.altera.co.jp/jamisp の Jam のウェブサイトにアクセスしてください。

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。

- 「AN 39: IEEE 1149.1 (JTAG) Boudary-Scan Testing in Altera Devices」
- 「AN 111: Embedded Programming Using the 8051 & Jam Byte-Code」
- 「AN 122: Using Jam STAPL for ISP & ICR via an Embedded Processor」
- 「MAX II デバイス・ハンドブック」の「DC およびスイッチング特性」の章
- 「MAX II デバイス・ハンドブック」の「MAX II デバイスのイン・システム・プログラマビリティ・ガイドライン」の章

改訂履歴

表 14–9 に、本資料の改訂履歴を示します。

表 14–9. 改訂履歴

日付 & ドキュメント・バージョン	変更内容	概要
2007 年 12 月 v1.7	「参考資料」の項を追加。	—
2006 年 12 月 v1.6	改訂履歴を追加。	—
2006 年 8 月 v1.5	「エンベデッド・システム」の項を更新。	—
2005 年 8 月 v1.4	表 14-2 および 14-3 を更新。	—
2005 年 6 月 v1.3	v 1.2 の表 14-6 を削除。 新しい項、「MAX II Jam/JBC アクションおよびプロシージャ・コマンド」を追加。	—
2005 年 1 月 v1.2	15 章から変更。内容の変更はなし。	—
2004 年 12 月 v1.1	参考資料を AN88 から AN 122 へ変更。	—