AN586: Jam STAPL および Jam STAPL Byte-Code Players のエンベデッド・システムへの移植

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。

2009 年 8 月 AN586-1.0

Jam™ STAPL（Standard Test and Programming Language）および Jam STAPL Byte-Code (JBC) Players は、プロセッサが、Jam ファイル (**.jam**) または Jam Byte-Code ファイル (**.jbc**) 内のアルゴリズムに基づいて CPLD あるいは FPGA デバイスをプログラムまたはコンフィギュレーションすることを可能にするソフトウェアです。

概要

このアプリケーション・ノートでは、Jam STAPL および Jam STAPL Byte-Code Players をエンベデッド・システムに移植する時に注意する必要のあるファンクションについて情報を提供しています。

☞ 移植に必要な変更はご使用のエンベデッド・システムおよび OS によって異なるため、このアプリケーション・ノートでは例を提供していません。

Jam STAPL または JBC Player の移植の概要

Jam STAPL または JBC Player は、**.jam** ファイル または **.jbc** ファイル内の Jam STAPL あるいは JBC 命令をそれぞれ解釈して実行します。メイン・プログラムは Jam または JBC Player の基本的な機能を実行します。ターゲットとされるエンベデッド・システムに応じて、Jam または JBC Player の I/O ファンクションを変更して、エンベデッド・プロセッサまたは OS（例えば Freescale™ V2 ColdFire Processor）に対してファンクションをカスタマイズする必要があります。

図 1 に、遅延ルーチン、OS 固有のファンクション、およびファイル I/O ピン用のルーチンを指定する、**jamstub.c** または **jbistub.c** に含まれるファンクションを示します。

図 1. Jam Player のソース・コード構造 *（注 1）*

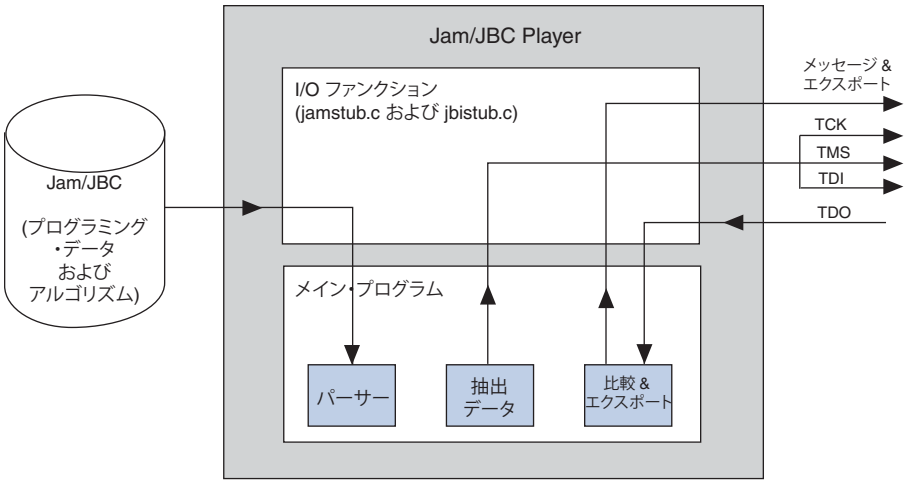

図 1 の注：

(1) TCK、TMS、TDI、および TDO は JTAGI/O ピンです。

jamstub.c または **jbistub.c** ファイルを編集することによって、任意のエンベデッド・システムに対して I/O ファンクションをカスタマイズしてソース・コードをコンパイルすることができます。Jam または JBC Player は C プログラミング言語で記述されています。コンパイル時に最大の互換性を確保するには、Jam または JBC PlayerC プログラミング言語対応のコンパイラをサポートするエンベデッド・システムへ移植することが推奨されています。

Jam および JBC Player のソース・コードは 32 ビットのプロセッサのみサポートします。他の種類のプロセッサをサポートするには、Jam または JBC Player のソース・コードを移植する必要があります。

CAUTION アルテラでは、**jamstub.c** および**jbistub.c**ファイル内のJamまたはJBC Playerのソース・コードのみを変更することが推奨されています。

jamstub.c および jbistub.c ファイル内のファンクション

この項では、移植時に注意する必要のあるファンクションが記載されています。表 1 に、ファンクションおよび **jamstub.c** と **jbistub.c** ファイル内の対応する名称を示します。表の後では、各ファンクションについて簡単に説明します。

表 1. jamstub.c および jbistub.c ファイル内のファンクション

ファンクション	jamstub.c ファイル内のファンクション	jbistub.c ファイル内のファンクション
メイン・ファンクション		
main	main()	main()
遅延ファンクション		
get_tick_count	get_tick_count()	get_tick_count()
calibrate_delay	calibrate_delay()	calibrate_delay()
delay	jam_delay()	jbi_delay()
追加ファンクション		
getc	jam_getc()	–
seek	jam_seek()	–
jtag_io	jam_jtag_io()	jbi_jtag_io()
message	jam_message()	jbi_message()
export_integer	jam_export_integer()	jbi_export_integer()
malloc	jam_malloc()	jbi_malloc()
initialize_jtag_hardware	initialize_jtag_hardware()	initialize_jtag_hardware()
close_jtag_hardware	close_jtag_hardware()	close_jtag_hardware()
read_byteblaster	read_byteblaster()	read_byteblaster()
write_byteblaster	write_byteblaster()	write_byteblaster()

メイン・ファンクション

main

このファンクションはあらゆる C プログラムの部分であり、すべての C プログラムの主要なビルディング・ブロックです。Jam または JBC Player のソース・コードでは、`main()` ファンクションには **.jam** または **.jbc** ファイルの位置、初期化リスト、および終了コードが含まれています。また、`jam_execute` または `jbi_execute` ファンクション（Jam または JBC Player への主要なエントリ・ポイント）は `main()` ファンクションによって呼び出されます。

デフォルトでは、初期化リスト、動作およびファイル位置が NULL に設定されています。コマンド・プロンプトで Jam または JBC Player に命令を与えるため、ファイル名および初期化リストはターミナル・プログラムによって入力ストリームから読み出されます。エンベデッド・プロセッサにユーザー・インタフェースがない場合は、このセクションをカスタマイズしなければなりません。初期化リストおよび動作の説明については、「初期化リストおよび動作」を参照してください。

初期化リストおよび動作

次の項では、`main()` ファンクションの初期化リストおよび動作について説明します。

初期化リスト

初期化リスト（`init_list`）はポインタの文字列のアドレスであり、それぞれ初期化文字列を含みます。初期化文字列はそれぞれ「文字列＝値」の形であります。初期化リストは、どちらの初期化文字列を実行するかについて、Jam または JBC Player に命令を提供します。表 2 に、Jam 仕様バージョン 1.1 で定義される文字列が記載されています。

表 2. Jam 仕様バージョン 1.1 で定義される文字列

初期化文字列	値	説明
DO_PROGRAM	0	デバイスをプログラムしない
	1 (デフォルト)	デバイスをプログラムする
DO_VERIFY	0	デバイスを検証しない
	1 (デフォルト)	デバイスを検証する
DO_BLANKCHECK	0	デバイスの消去ステートをチェックしない
	1 (デフォルト)	デバイスの消去ステートをチェックする
READ_USERCODE	0 (デフォルト)	JTAG USERCODE を読み出さない
	1	USERCODE を読み出し、エクスポートする
DO_SECURE	0 (デフォルト)	セキュリティ・ビットを設定しない
	1	セキュリティ・ビットを設定する

初期化リストを適切な方法で渡さなければなりません。無効な初期化リストが渡された場合、あるいは初期化リストが渡さない場合は、Jam または JCB Player は **.jam** または **.jbc** ファイルにのみ構文チェックを実行します。構文チェックがパスしたら、Jam または JBC Player は何のファンクションも実行せずに成功した終了コードを発行します。例 1 に、`init_list` をセットアップして Jam または JBC Player にプログラムの実行および動作の検証をさせるようにコードを定義する方法を示します。

例 1.

```
Char CONSTANT_AREA init_list[] [] ="DO_PROGRAM=1",
"DO_VERIFY=1"
```

Action

action は Jam または JBC Player が実行する動作を指定します。デフォルトでは、action は NULL に設定されます。ある初期化リストが必要とされない場合、NULL ポインタを使用して空の初期化リストを表すことができます。これは、動作がすでに Jam または JBC Player で定義されている場合にのみ適用されます。表 3 に、Jam および JBC Player で使用できる動作が記載されています。

表 3. Jam および JBC Players で使用できる動作

動作	説明
PROGRAM	デバイスをプログラムする
VERIFY	デバイスを検証する
BLANKCHECK	デバイスの消去ステートをチェックする
READ_USERCODE	USERCODE を読み出し、エクスポートする

遅延ファンクション

jamstub.c および **jbistub.c** に 3 つの相互に関連する遅延ファンクション（delay()、calibrate_delay()、and get_tick_count()）があります。get_tick_count() ファンクションはシステムのチックカウント値を取得し、その値を calibrate_delay() ファンクションに返します。そして、calibrate_delay() ファンクションはシステムのチックカウントを用いて、1 ミリ秒の遅延に必要とされるループ数を測定します。次に、delay() ファンクションはこの情報を使用して、WAIT コマンドに必要とされる遅延を実行します。

get_tick_count

このファンクションは calibrate_delay() ファンクションによって呼び出され、ミリ秒でシステムのチックカウントを取得するために使用されます。デフォルトで、ソース・コードは下記の OS に合わせています。

- WINDOWS—GetTickCount() ファンクション
- UNIX—clock() システム・ファンクション

ご使用のエンベデッド・プロセッサが Windows または UNIX に対して上記のファンクションのいずれかも使用していない場合、このファンクションを適切にカスタマイズする必要があります。

calibrate_delay

このファンクションは、1 ミリ秒の遅延に必要とされるループ数を測定します。デフォルトで、Windows OS では、このソース・コードは計算に含まれています。

エンベデッド・プロセッサの OS が Windows でない場合、このファンクションをカスタマイズする必要があります。

delay

このファンクションは、PLD とメモリのプログラミングおよび SRAM ベースのデバイスのコンフィギュレーションに必要なプログラミング・パルス幅を実装します。これらの遅延は、ターゲットとされるプロセッサの速度に合わせて調整されたソフトウェア・ループによって実装されます。例えば、さまざまな幅のパルスは、アルテラの MAX® CPLD の内部 EEPROM セルをプログラムするのに使用されます。Jam または JBC Player は `delay()` ファンクションを使用してこれらのパルス幅を実装します。**.jam** または **.jbc** 内の `WAIT` コマンドは、必要な遅延を指定します。

このファンクションを、プロセッサの速度およびプロセッサが 1 つのループを実行するのに必要な時間を基づいてカスタマイズしなければなりません。Jam または JBC STAPL 文を実行する時間を最小限に抑えるためには、1 ミリ秒～ 1 秒の範囲で遅延を可能な限り正確に調整することが推奨されています。

追加ファンクション

getc

このファンクションは、**.jam** ファイル内のキャラクタを受信します。`getc()` ファンクションへの各呼び出しは、ファイル内のポインタの現在位置を進めます。ファンクションへの成功した呼び出しは、キャラクタの文字列を取得するのに必要です。成功した呼び出しがファイルの終端に到達すると、ファイル終端を示すインジケータは設定され、`getc()` ファンクションは EOF を返します。このファンクションは標準の `fgetc()` C ファンクションに似ています。このファンクションは読み出されたキャラクタ・コードを返し、また、キャラクタ・コードがない場合は (-1) を返します。

デフォルトで、Windows OS に対しては、ソース・コードは **.jam** 内のキャラクタの受信用のアルゴリズムを備えています。プロセッサの OS が別のアルゴリズムを使用する場合、このファンクションをカスタマイズする必要があります。

seek

このファンクションは、指定されたオフセットに基づいて、**.jam** 入力ストリーム内の現在のファイル位置ポインタを設定します。オフセットがファイル長以内であれば、このファンクションは 0 を返します。そうでない場合、0 でない数値が返されます。このファンクションは、標準の `fseek()` C ファンクションに似ています。

ソース・コードでは、**.jam** のストレージ・メカニズムはメモリ・バッファです。あるいは、ファイル・システムをストレージ・メカニズムとして使用することもできます。そうする場合は、C 言語の `fopen()` と `fclose()` ファンクションに等価するものを使用したり、ファイル・ポインタを格納したりするように、ファンクションをカスタマイズしなければなりません。

jtag_io

このファンクションは、IEEE 1149.1 JTAG 信号（TDI、TMS、TCK、および TDO）へのアクセスを提供します。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。

現在のソース・コードでは、PC パラレル・ポートは TDO の実際の値を反転させます。jbi_jtag_io() ソース・コードは、TDO の値を再度反転させて元のデータを回復します。

```
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、TDI と TMS 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。

PC パラレル・ポートを使用していない場合、このファンクションを、適切なハードウェア・ポートに書き込むようにカスタマイズしなければなりません。

message

Jam または JBI Player が **.jam** または **.jbc** で PRINT コマンドを検出すると、Player はテキスト・メッセージを処理し、結果を message() ファンクションに渡します。標準の出力デバイスがない場合、message() は何もしません。

Player はテキスト・メッセージの最後に改行文字を追加しません。ご使用のエンベデッド・システムに改行を追加する必要がある場合、このファンクションの情報およびエラー・メッセージを標準出力に出力するように message() ファンクションを変更する必要があります。このファンクションを使用していない場合、このルーチンを削除し、あるいは message() ファンクションにある puts() ファンクションへの呼び出しをコメント・アウトすることができます。

export_integer

export_integer() および export_boolean_array() ファンクションは、Jam または JBC Player からの情報を呼び出しプログラムに返します。このルーチンの最も一般的な用途は、ユーザー電子署名（UES）命令コードをデバイスから Jam または JBC Player を呼び出すプログラムに返送します。これらのファンクションは、printf() ファンクションを用いてテキスト・メッセージを stdio に送信します。Jam または JBC Player は export_integer() および export_boolean_array() ファンクションを使用して、Jam または JBC Player を呼び出す OS あるいはソフトウェアに情報（例えば、UES 命令コードまたはデバイスの USERCODE）を渡します。

デフォルトで、Jam または JBC Player は printf コマンドを用いて値を出力します。printf コマンドが使用できない場合、あるいは stdout に対して使用可能なデバイスがない場合は、これらのファンクションを変更する必要があります。情報をファイルまたはストレージ・デバイスにリダイレクトするか、あるいは情報を変数として Jam または JBC Player を呼び出すプログラムに渡します。

malloc

このファンクションは呼び出されるたびに、必要とされるメモリを割り当てます。プログラム実行時に、Jam または JBC Player タスクを実行するために、Jam または JBC Player はメモリを割り当てる必要があります。Jam または JBC Player がメモリを割り当てるとき、malloc() ファンクションは呼び出されます。例えば、プログラムが Jam または JBC ファイルをメモリに書き込む場合、Jam または JBC Player はこのファンクションを使用して、**.jam** または **.jbc** の格納に必要なメモリを割り当てます。Jam または JBC ファイルのサイズは、ターゲットとされるデバイスおよびその量によって異なります。それぞれのデザインを評価し、適切なメモリ・リソースを選択しなければなりません。

場合によっては、malloc() ファンクションはエンベデッド・システムにサポートされないことがあります。このような場合は、このファンクションを等価なファンクションで置き換える必要があります。

ROM および RAM メモリの使用量を推定する方法について詳しくは、*「AN 425: Using Command-Line Jam STAPL Solution for Device Programming」*の「Jam STAPL Byte-Code Player Memory Usage」の項を参照してください。

initialize_jtag_hardware

このファンクションはハードウェア I/O を初期化し、プレーヤの JTAG ポートへの書き込みを可能にします。デフォルトで、Jam または JBC Player ソース・コードには、ハードウェア I/O を Windows OS に対して初期化するルーチンが含まれています。

OS およびハードウェア要件を基づいてハードウェア I/O を初期化するには、このファンクションをカスタマイズする必要があります。

close_jtag_hardware

このファンクションはハードウェア I/O を閉じて（あるいは非アクティブにして）、プレーヤを JTAG ポートの書き込まないようにします。デフォルトで、Jam または JBC Player のソース・コードには、Windows OS に対して通信ポートを閉じるルーチンが含まれています。

Windows 以外の OS に対してハードウェア I/O を閉じるには、このファンクションをカスタマイズする必要があります。

read_byteblaster

このファンクションは、ByteBlaster™ II ダウンロード・ケーブルを介してデータを読み出します。read_byteblaster ファンクションは、conio.h ライブラリからの inp() ファンクションを使用して、パラレル・ポートから読み出します。このファンクションは Windows システムに対してのみカスタマイズされています。

このファンクションを、ByteBlaster II ダウンロード・ケーブルを介してリード動作を実行する、エンベデッド・プロセッサ内の等価なファンクションでカスタマイズする必要があります。

write_byteblaster

このファンクションは、ByteBlaster™ II ダウンロード・ケーブルを介してデータを書き込みます。write_byteblaster ファンクションは、conio.h ライブラリからの outp() ファンクションを使用して、パラレル・ポートに書き込みます。このファンクションは Windows システムに対してのみカスタマイズされています。

このファンクションを、ByteBlaster II ダウンロード・ケーブルを介してライト動作を実行する、エンベデッド・プロセッサ内の等価なファンクションでカスタマイズする必要があります。

改訂履歴

表 4 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 4. 改訂履歴

日付および リビジョン	変更内容	概要
2009 年 8 月	初版	—

101 Innovation Drive
San Jose, CA 95134
www.altera.com
Technical Support
www.altera.com/support