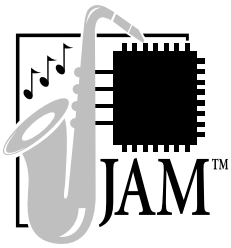
エンベデッド・プロセッサによるISPに Jam言語を使用する方法

1997年 7 月 ver.1 Application Note 88

イントロダクション



MAX® 9000、MAX 9000A、およびMAX 7000Sファミリの各デバイスでサポートされているエンベデッド・プロセッサを使用したイン・システム・プログラミングは、デザインの試作を容易にし、製造工程の簡略化やフィールドでの効率的なアップ・グレードを実現します。イン・システム・プログラマビリティ（ISP）をサポートしているデバイスにROM、FLASHカード、モデムまたは他のデータ・リンクを使用して新しいデータをダウン・ロードすることによって、フィールドで簡単にアップ・グレードを行うことができます。設計変更データをフィールドでエンベデッド・プロセッサを介してシステムにダウンロードさせることもできます。エンベデッド・プロセッサにプログラミング・データをメモリ・ソースからデバイスに転送させることによって、デザインのアップ・グレードが簡単に行えるようになります。

Jam™はISPのために開発された新しい標準ファイル・フォーマットのプログラミングおよびテスト用言語であり、IEEE 1149.1のJTAG（Joint Test Action Group）インタフェースを使用しているあらゆるISPデバイスのプログラミングをサポートしています。Jamはインタプリタ言語です。Jamのソース・コードはエンベデッド・プロセッサ上で動作するインタプリタ・プログラムによってダイレクトに実行され、これらのソース・コードをバイナリの実行コードにコンパイルする必要はありません。（詳細については４ページの「Jam言語を使用したエンベデッド・プログラミング」を参照して下さい。）Jamのソース・コード、またはJamのファイルには、１個または複数のデバイスをアップ・グレードするために必要なプログラミング・アルゴリズムとデータが含まれています。Jamはオープンな標準規格として、そのライセンスが無償で提供されています。

このアプリケーション・ノートはJam言語を使用して、エンベデッド・プロセッサによるISPの利点を実現する方法について解説したものです。このアプリケーション・ノートでは次の項目について解説します。

- エンベデッド・システムの構成と要求される機能
- Jam言語を使用したエンベデッド・プログラミング

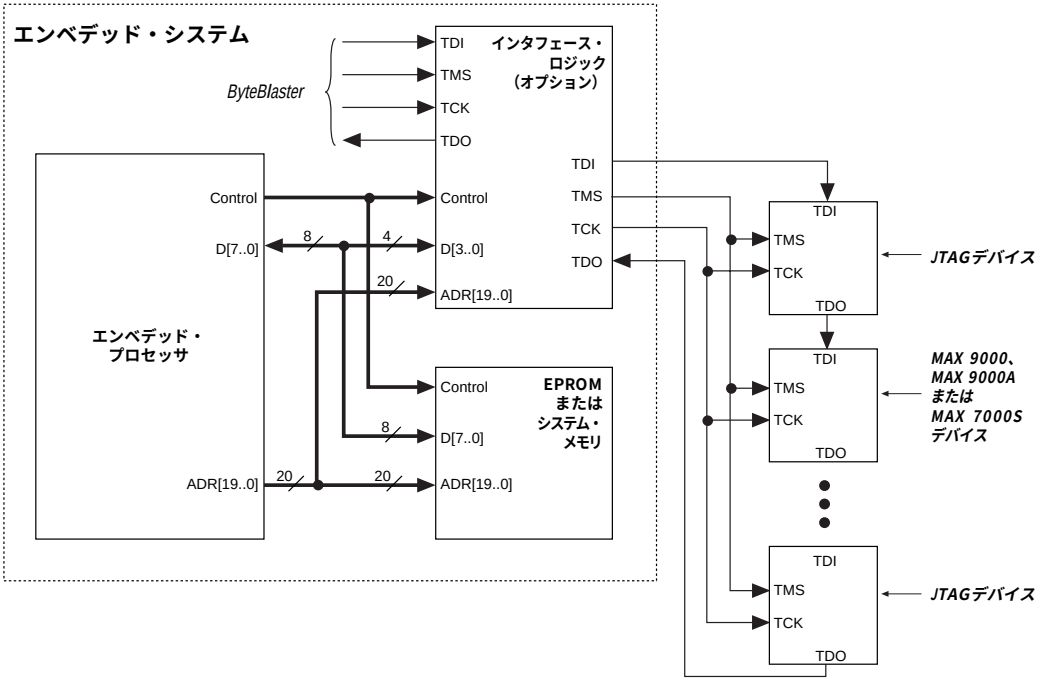
☞ このアプリケーション・ノートの使用にあたっては、「*Jam Programming & Test Language Specification*」を参照して下さい。

エンベデット・システムの構成と要求される機能

ISPが提供する利点を実現するためには、エンベデッド・システムがターゲット・デバイスを小容量のシステム・メモリを使用してプログラムできるようになっている必要があり、複数ベンダの製品が含まれているデバイス・セットの変更にも対応できるような柔軟性を持っていなければなりません。通常、エンベデッド・システムはエンベデッド・プロセッサ、EPROMまたはシステム・メモリ、およびいくつかのインタフェース・ロジックによって構成されます。プログラミング・データはシステム・メモリ（EPROMまた

はFLASHメモリ）にストアされます。イン・システム・プログラミングを行うときは、エンベデッド・プロセッサがプログラミング・データをシステム・メモリからISPデバイス（1個または複数）に転送します。図1はエンベデッド・システムの構成をブロック図で示したものです。

図1　エンベデッド・システムのブロック・ダイヤグラム

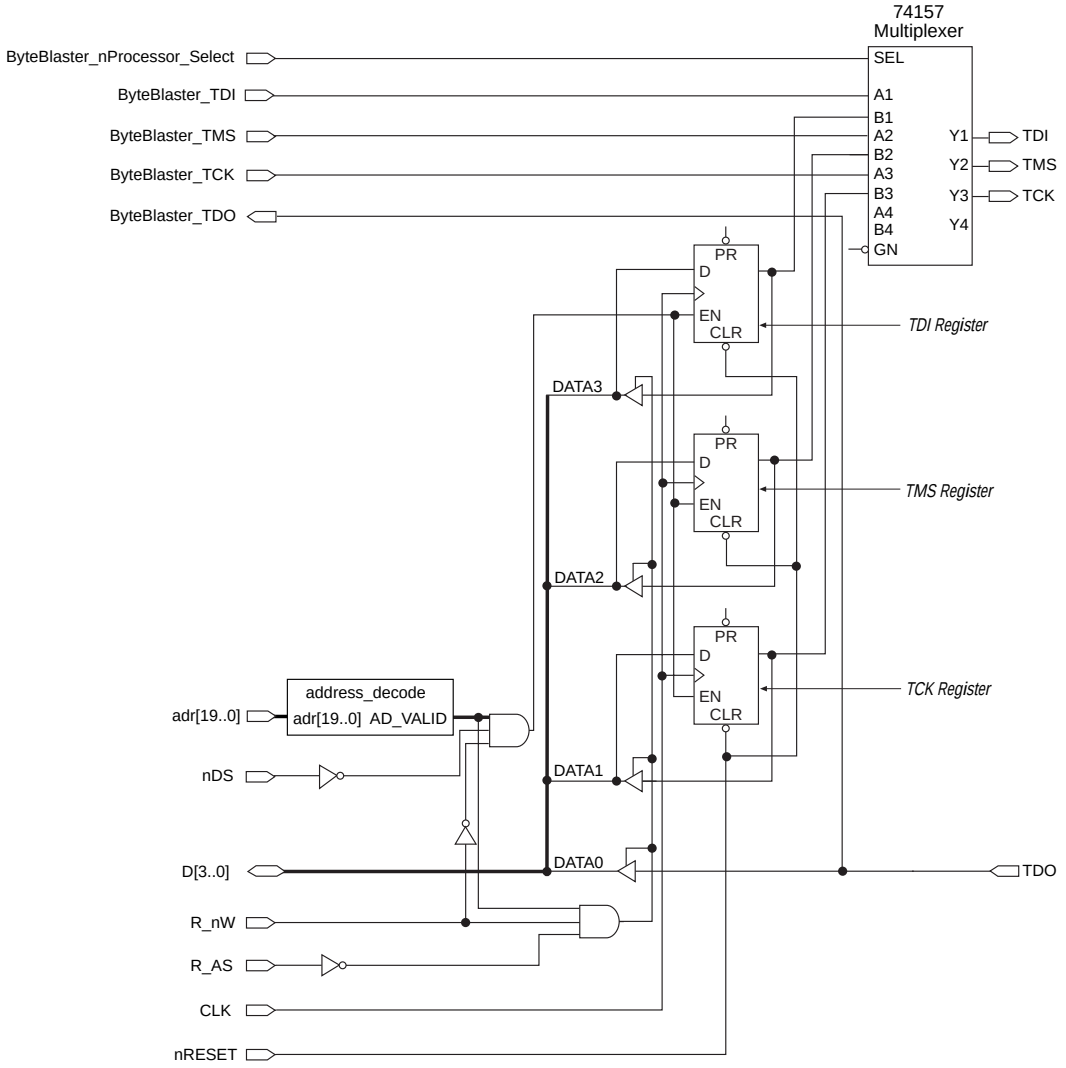


エンベデッド・プロセッサはEPROMまたはシステム・メモリ、およびオプションのインタフェース・ロジックを構成しているプログラマブル・ロジック・デバイス（PLD）と接続されます。JTAGチェインはエンベデッド・プロセッサの4本のデータ・ピンとダイレクトに接続することができますが、インタフェース・ロジックを追加することで、JTAGチェインを既存のバスのひとつのアドレス・ロケーションとして扱うことが可能になり、プロセッサの4本のポートを節約することができます。また、ボード上にByteBlaster™用に10ピンのヘッダを実装しておくことによって、MAX+PLUS® IIソフトウェアとByteBlasterパラレル・ポート・ダウンロード・ケーブルを使用したJTAGチェインのアクセスと検証が可能になります。

ByteBlasterパラレル・ポート・ダウンロード・ケーブルの詳細については、「*ByteBlaster Parallel Port Download Cable* のデータシート」を参照して下さい。

図2はエンベデッド・システムのインタフェース・ロジックを示したものです。

図2　インタフェース・ロジック



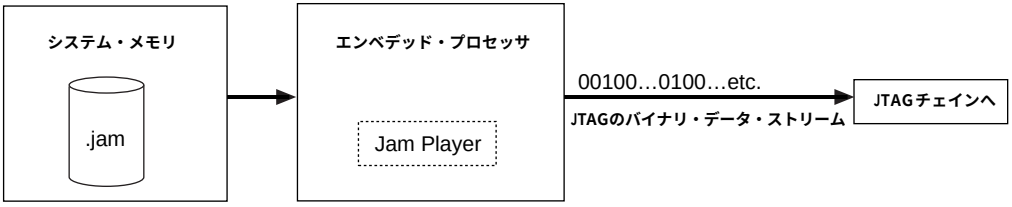
このインタフェース・ロジックはエンベデッド・プロセッサから適切なアドレスとコントロール信号を受信したときにアクティブとなります。このとき、レジスタはTDI、TCK、TMS信号のタイミング同期をとり、74157のマルチプレクサを経由して出力ピンをドライブします。このマルチプレクサは、ByteBlasterによるJTAGチェインのアクセスを可能にしています。

Jam言語を使用したエンベデッド・プログラミング

Jam言語はJam File（.jam）とJam Playerの２つの部分で実現されています。Jam FileはMAX+PLUS II開発ツールによって生成され、システム・メモリにストアされます。Jam Fileには１個または複数のISPデバイスをプログラムするために必要なすべての情報が含まれます。これに対して、Jam Playerはエンベデッド・プロセッサ上で動作し、Jam File内の情報を解読して、デバイスのプログラミングに必要なバイナリのデータ・ストリームを生成します。アップ・グレードの情報はJam Fileのみで供給されるため、Jam Playerはあらゆるベンダのデバイスをプログラムできるようになっている必要があります。

図３はJam言語を使用してイン・システム・プログラミングがどのように実現されるかをブロック図で示したものです。

図３　Jam FileとJam Playerを使用したISPのブロック図



Jam File（.jam）

Jam Fileはコンパクトなファイル・サイズとなっており、IEEE 1149.1で規定されているJTAG仕様に準拠したあらゆるISPサポート・デバイスに対して生成されます。

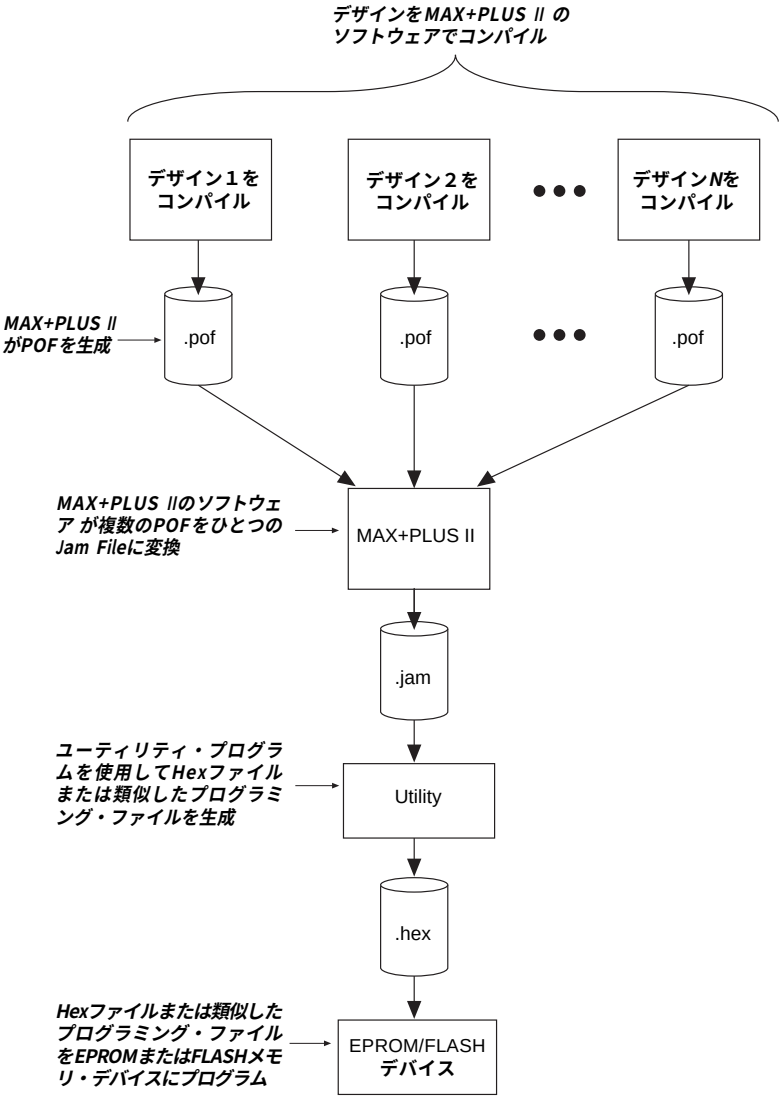
Jam Fileの生成方法

Jam言語を使用してアルテラのデバイスをプログラムする場合は、MAX+PLUS II開発用ソフトウェアを使用してJam Fileを生成する必要があります。MAX+PLUS IIのソフトウェアは、Create Jam/SVF Fileのコマンド（Fileメニュー）を使用してProgrammer Object File（.pof）からJam Fileを生成することができます。Jam FileはPOFから生成されるため、既存のデザインを再コンパイルする必要はありません。Jam FileをEPROMまたはFLASHメモリにストアする場合は、まずJam Fileをヘキサデシマル（インテル・フォーマット）のファイル（.hex）、または類似したフォーマットのプログラミング・ファイルに変換する必要があります。エンベデッド・プロセッサのソフトウェア・パッケージまたは他のユーティリティ・プログラムを使用して、Jam FileをEPROMまたはFLASHメモリのプログラミング・ファイルに自動的に変換することもできます。また、EPROMのプログラミング装置には、「raw binary」または「absolute binary」のフォーマットをサポートしているものがあり、こうしたプログラミング装置はJam Fileを変換なしで読み込むことができます。

Jam Fileを生成する方法の詳細については、MAX+PLUS IIのヘルプ機能を使用し、「Create Jam or SVF Files」の項目をサーチして確認して下さい。

図4はイン・システム・プログラミングに使用されるJam Fileを生成する方法を示したものです。

図4　Jam Fileの生成方法



イニシャライズの規定

MAX+PLUS IIのソフトウェアが生成するJam Fileには、Jamで規定されているイニシャライズに関する記述部分が含まれています。この部分にはJam言語によってサポートされているイニシャライズの仕様が記述され、これらの内容はJTAGチェイン内の各デバイスに応じて変更することができます。

DO_PROGRAM

DO_PROGRAMはデバイスをプログラムするかどうかを決定する変数です。DO_PROGRAMが1にセットされている場合は、Jam Playerが1個または複数のISPサポート・デバイスに対して、シリコンIDのチェック、デバイス全体の消去（バルク・イレーズ）、そしてプログラムを実行します。同一ファミリの複数のデバイスをプログラムする場合は、MAX+PLUS IIのソフトウェアはコンカレント・プログラミング・アルゴリズムを使用します（同じファミリのすべてのデバイスにプログラミング・データが同時にシフト・インされる）。

プログラミングが開始されると、ターゲット・デバイスのI/Oピンはトライ・ステートとなり、最後のデバイスのプログラミングが完了すると、トライ・ステートのモードが解除されます。これらの変化はJTAGチェイン内のすべてのターゲット・デバイスで同時に発生します。

DO_VERIFY

DO_VERIFYはJam Fileのデータとデバイスに書き込まれた内容が一致しているかを検証（ベリファイ）するための変数です。DO_VERIFYが1にセットされると、ターゲット・デバイスに対するベリファイ動作が実行されます。ベリファイの結果はJamプログラムの終了コードで表示されます。

DO_ERASE

DO_ERASEはJam Fileにバルク・イレーズを実行させるための変数です（デバイスが完全に消去される）。このプロセスを実行することで、各ビットに対する適切なプログロミングと、複数のプログラミング・サイクルを実行した場合でもデバイスに対する適切な信頼性が確保されます。

DO_BLANKCHECK

DO_BLANKCHECKはプログラミング前にデバイス全体が適切に消去されていることを確認するための変数です。DO_BLANKCHECKの変数の設定でデバイス内のすべてのデータが消去されていることが検証されます。

READ_UESCODE

READ_UESCODEはデザインのリビジョンや、デバイスにプログラミングされた回数を記録しておくための便利な変数となっています。

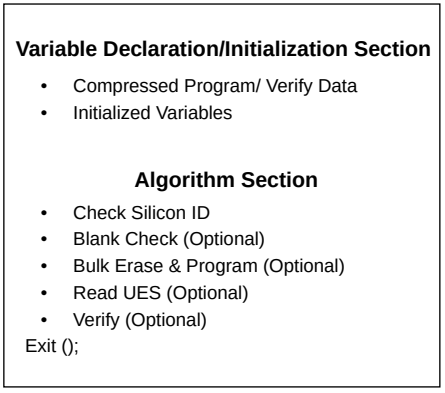
READ_USERCODEに１がセットされていると、UESのコードが読み出され、レポートされます。

イニシャライズの規定に関する詳細については、「*Jam Programming & Test Language Specification*」を参照して下さい。

Jam Fileの構造

エンベデッド・システムにおいて、Jam Fileはアップデート可能なシステム・メモリにストアされます。Jam Fileはコンパクトなメモリ・サイズにストアできるような構造となっており、Variable Declaration and Initialization Section（変数宣言／イニシャライズ・セクション）およびAlgorithm Section（アルゴリズム・セクション）を持っています。図５はJam Fileの構造を示したものです。

図５　Jam Fileの構造



Variable Declaration and Initialization Sectionには、Jam File内で使用される変数に関する宣言が含まれます。また、このセクションでは、変数を特定の値にイニシャライズすることが可能です。BOOLEANアレイの場合は、変数がRun-Length Compression（RLC）またはAdvanced Compression Algorithm（ACA）のいずれかのフォーマットに圧縮されたデータ・アレイにイニシャライズすることができます。他の変数もこのセクションで宣言および初期化することができます。なお、変数のイニシャライズはオプションです。

RLCおよびACAに関する詳細については、「*Jam Programming & Test Language Specification*」を参照して下さい。

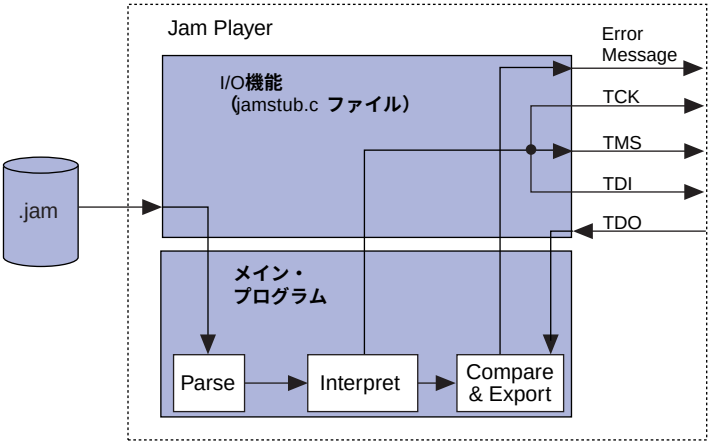
Algorithmのセクションはすべてテキストで記述され、実際のプログラミング・コマンドや他の必要な機能（ベリファイ結果に応じたブランチ、複数

のJTAGデータ・レジスタ・スキャンのループ、ターゲットとなっているJTAGチェインをトラックするための管理機能など）を実行するためのプログラミング・コードを含むサブセクションに分割することができます。Algorithmのセクションには1個または複数のデバイス上で実行される上位の変数（Blank Check、Verifyなど）が含まれます。

Jam Player

Jam PlayerはJam Fileの解析（parse）、各Jam命令の解読、JTAGチェインに対するデータのリード／ライト動作を行うためのCプログラムです。イニシャライゼーション・リストにしたがってJam Playerによって処理された各変数は実行時に有効となります。（詳細については12ページの「Jam Playerの実行」を参照）各アプリケーションにはそれぞれ個別の要求が存在するため、Jam Playerのソース・コードは簡単に変更できるようになっている必要があります。図6はJam Playerのソース・コード構造を示したものです。

図6　Jam Playerのソース・コード構造



メイン・プログラム部分はJam Playerのすべての基本的な機能が変更なしで実行できるようになっています。ここで変更が必要になるのは、jamstub.cのファイルに含まれるI/O機能の部分のみです。この部分には、I/Oピンに対するアドレス、遅延ルーチン、OSに特化する機能、およびファイルの入出力に関するルーチンが規定されます。

Jam Playerはシステム・メモリ内に常駐し、Jam File内のコマンドの解読、デバイス・プログラミングに使用するバイナリのデータ・ストリームの生成を行います。この構造の実現により、すべてのアップ・グレードはJam Fileで行えるようになり、Jam Playerをあらゆるシステム・アーキテ

クチャに対応させることができます。

Jam Playerの詳細については、「*Jam Programming & Test Language Specification*」を参照して下さい。

Jam Playerのカスタマイズ

Jam Playerは各アプリケーションやプラットフォームの要求に応じて簡単にカスタマイズできる構造となっています。すべてのファイルのI/Oやポートの構成はjamstub.cのファイルを書き換えることで簡単に変更することができます。まず、入力の機能として、jamstub.cファイルはJam Fileからのデータの検索、TDOから出力されたシフト・アウト・データの読み込みを行うことができます。また、出力機能として、jamstub.cファイルは3本のJTAGピン（TDI、TMS、TCK）に対する処理されたJTAGデータの送出、コールされたプログラムに対するフォーマット化されたエラーおよびインフォメーション・メッセージの返送、およびコール・プログラムに対するステータス・インフォメーションの返送などを行うことができます。

jamstub.cのファイルは何種類かのI/Oやプラットフォームに対応できる8つのルーチンを持っています。これらのルーチンは下記のようなコメント文で開始されます。

```
/****************************************************
*  Customized interface functions for Jam Player I/O:
*
*  jam_getc()
*  jam_seek()
*  jam_jtag_io()
*  jam_message()
*  jam_export()
*  jam_vector_map()
*  jam_vector_io()
*  jam_delay()
```

各プラットフォームに対応した8種類のルーチンをそれぞれカスタマイズすることによって、ソース・コードをあらゆるプラット・フォームで使用できるようにコンパイルすることが可能です。

入力ストリームの変更

jam_getc()とjam_seek()のルーチンはANSI Cのstdioライブラリを使用しています。これらのルーチンはJam Fileから供給された情報の検索、Jam File内のポインタの位置の操作を行います。jam_getc()とjam_seek()の記述例を下記に示します。

```
int jam_getc(void)
{
```

```
    return (getc(fp));
}

int jam_seek(long offset)
{
    return (fseek(fp, offset, SEEK_SET));
}
```

I/Oポートの変更

I/OポートはTDI、TMS、TCK、およびTDOの４本のＪＴＡＧピンで構成されています。これらのピンの位置は、下記のようにjamstub.cファイル内の#defineの記述を変更することで、各ピンごとに個別のアドレスへ変更することができます。

```
#define TMS_BIT 0x01
#define TDI_BIT 0x02
#define READ_TDO 0x04
```

このピン・アドレスは、ＪＴＡＧポートに対するリードとライトを行う唯一のルーチンとなっているjam_jtag_io()のルーチンによって使用されます。この場合、TCKはjam_jtag_io()がコールされるごとに自動的に書き込まれるため、TCKのアドレスが規定されることはありません。jam_jtag_io()のルーチンは下記のようなソース・コードを含んでいます。

```
int jam_jtag_io(int tms_tdi)
{
    int data = ((tms_tdi & TDI_BIT) << 5) | ((tms_tdi
& TMS_BIT) << 1);
    int tdo = 0;
    if (!jtag_hardware_initialized)
    {
        initialize_jtag_hardware();
        jtag_hardware_initialized = TRUE;
    }
    write_ByteBlaster(0, data);
    if (tms_tdi & READ_TDO)
    {
        tdo = (read_ByteBlaster(1) & 0x80) ? 0 : 1;
    }
    write_ByteBlaster(0, data | 0x01);
    write_ByteBlaster(0, data);
    return (tdo);
}
```

WindowsまたはDOSのオペレーティング・システムの指定は、単純にjamstub.cファイルの先頭に#defineのステートメントを追加することで行えます。例えば、下記の記述でDOSのプラットフォームが指定できます。

```
#define PORT DOS
```

WindowsやDOSのオペレーティング・システムを使用していない場合は、PORTに他の値を指定する必要があります。このとき、プロセッサはjamstub.cのファイルからWindowsとDOSのコードをすべて除去します。

Jam Playerが適切なポートにライト動作を行えるようにするためには、jam_jtag_io()のルーチンを変更して、write_ByteBlaster()とread_ByteBlaster()の部分を対応するルーチンに置き換える必要があります。

jam_vector_map()とjam_vector_io()のルーチンは、VMAPおよびVECTORのコマンドが与えられたときにJam Playerがインフォメーション・メッセージを生成できるようにするためのものです。ISPによるプログラミングでは、これらのコマンドが実行されることはないため、これらのルーチンを書き換える必要はありません。ただし、将来のJam言語の仕様ではこれらのルーチンを使用して４本のJTAGインタフェース以外のピンをドライブする信号もサポートされる予定です。

VMAPとVECTORのコマンドに関する詳細については、「*Jam Programming & Test Language Specification*」を参照して下さい。

Delay機能の変更

jam_delay()のルーチンはプログラム実行時の遅延を実現します。このルーチンでは、適切な遅延時間の実現にcalibrate_delay()によって設定されるone_ms_delayのグローバル変数が使用されます。calibrate_delay()では、単独ループの実行で生じる遅延の計算に異なる絶対時間が使用されます。
絶対時間の差は、各プラットフォーム固有の機能を使用して絶対時間の検索を行うget_tick_count()によってレポートされます。

このルーチンのソース・コードは下記の通りです。

```
/************************************************
*
*   get_tick_count() -- Get system tick count in
*   milliseconds
*
*   for DOS, use BIOS function _bios_timeofday()
*   for WINDOWS use GetTickCount() function
*   for UNIX use clock() system function
*/
DWORD get_tick_count()
{
    DWORD tick_count = 0L;
#if PORT == WINDOWS
    tick_count = GetTickCount();
#elif PORT == DOS
    _bios_timeofday(_TIME_GETCLOCK, (long
    *)&tick_count);
#else
    /* assume clock() function returns microseconds */
    tick_count = (DWORD) (clock() / 1000L);
#endif
    return (tick_count);
}
```

Windows、DOSまたはUNIXのプラットフォームを使用しないアプリケーションでは#else文の後に使用されているclock()の表現をシステム・クロックの数を規定した表現に置き換える必要があります。（tick_count() もミリ秒の単位の値に戻す必要がある。）

Jam Playerの実行

Jam Playerはフラグを使用して実行するISPの機能が指定できる高い柔軟性を実現しており、これらのフラグは実行時にJam Playerに与えられるようになっています。Jam Playerの実行は下記のフォーマットで行えます。

Jam[-h] [-v] [-p<*パラレル・ポートの16進のアドレス*>] [-m<*バイト数で表されるメモリ・サイズ*>] -d<*イニシャライズ・リスト*> <*Jam File名*>

[] 内のフラグはオプションです。Jam Playerは１回で１個のJam Fileのみを処理することができます。表１は各フラグを解説したものです。

表 1　Jam Player のフラグ

フラグ	定義	機　　能
-h *(1)*	ヘルプ	Jam Player のバージョンをレポートさせる
-v *(1)*	レポート機能	詳細なリアルタイム情報とステータスおよびエラー・メッセージをレポートさせる
-d	イニシャライズ	Jam Playerに対して実行するイニシャライズ機能を指定
-p *(1)*	ポート	Jam Playerがデータを送出するパラレル・ポート・アドレスを指定
-m *(1)*	メモリ	Jam Player が使用可能なメモリ・サイズを規定

注
(1)　このフラグはオプションです。

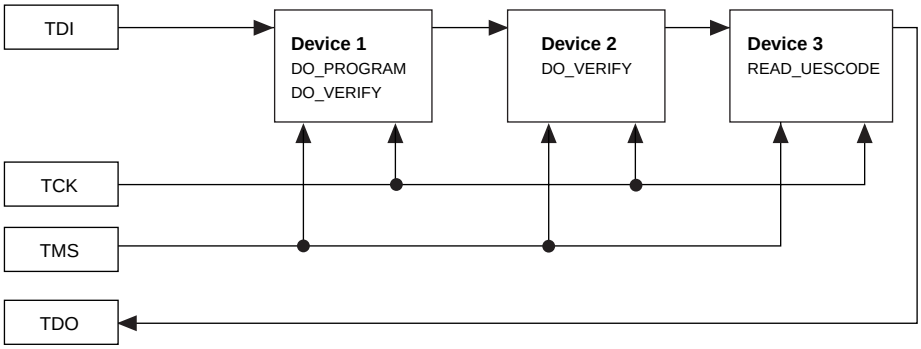
-dのフラグを使用する場合は、イニシャライゼーション・リストからJam Playerの初期化に必要な正しい変数が提供されなければなりません。-dフラグの後に使用される変数は表 2 の通りです。

表 2　-d フラグに使用される変数名と機能

変数名	値	機　　能
DO_ERASE	0	バルク・イレーズを実行しない
	1	バルク・イレーズを実行する
DO_BLANKCHECK	0	デバイスがイレーズされている状態をチェックしない
	1	デバイスがイレーズされている状態をチェックする
DO_PROGRAM	0	デバイスをプログラムしない
	1	デバイスをプログラムする
DO_VERIFY	0	デバイスをベリファイしない
	1	デバイスをベリファイする
READ_UESCODE	0	JTAG UES コードをリードしない
	1	UES コードをリードし、リポートする
DO_SECURE	0	セキュリティ・ビットをセットしない
	1	セキュリティ・ビットをセットする

-dのフラグの後でイニシャライズされない変数は 0 にセットされます。イニシャライゼーション・リスト内の変数の順番は重要ではありません。-dのフラグ後に指定されている各変数はJam File内で指定されるデバイスに適用されます。図 7 は、JTAGチェイン内の 3 個のデバイスに対してJam Fileによってそれぞれ異なる機能が指定されている例を示したものです。

図７　３個のデバイスで構成されるJTAGチェイン



ここで、Jam Playerの実行時にDO_PROGRAMとDO_VERIFYの変数のみが１にイニシャライズされた場合は、デバイス１に対してはプログラムとベリファイ、デバイス２にはベイファイが実行され、デバイス３には何も実行されないようになります。

例えば、コマンド・ラインから以下のようなテキストをタイプすると、Jam Playerはエラーおよびインフォメーション・メッセージをレポートし、デバイス（１個または複数）のプログラムおよびベリファイを実行し、さらにメモリを１Mバイトに制限して、Chiptrip.jamという名前のJam Fileに対するデータのリード／ライトを0×378のベース・アドレスのパラレル・ポートを通じて行います。

```
Jam -v -p378 -m1000000 -dDO_PROGRAM=1 -dDO_VERIFY=1
Chiptrip.jam
```

Jam Playerの下位レベル動作

Jam PlayerはJTAG 1149.1で規定されているTAP（Test Access Port）コントローラのステート・マシンを操作するための下位レベル・インタフェースを提供しています。このTAPコントローラはTCKの立ち上がりエッジで遷移する16ステートのステート・マシンとなっており、TMSピンがデバイスのJTAG動作のコントロールに使用されます。図８はJTAG TAPコントローラのステート・マシンを示したものです。

図8　JTAG TAPコントローラのステート・マシン

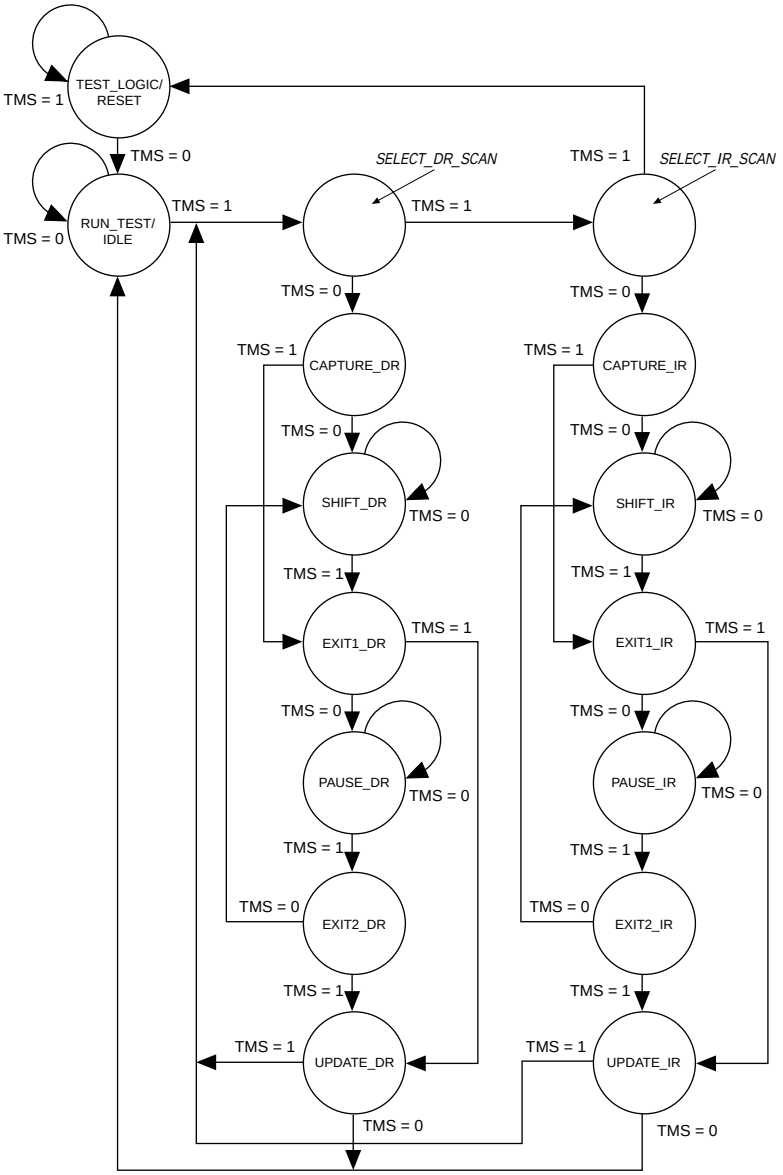


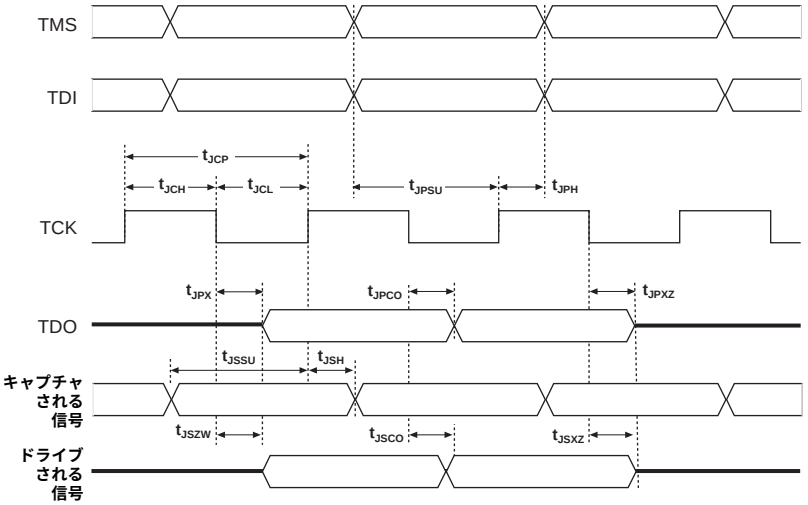
表３はＴＡＰコントローラのタイミング規格を示したものです。これらのタイミング・パラメータはIEEE 1149.1の仕様で規定されているものと同一です。

表３　JTAG のタイミング・パラメータ

シンボル	パラメータ	MAX 9000		MAX 7000S		単位
		最小	最大	最小	最大	
t_{JCP}	TCK のクロック期間	100		100		ns
t_{JCH}	TCK の High 時間	50		50		ns
t_{JCL}	TCK の Low 時間	50		50		ns
t_{JPSU}	JTAG ポートのセットアップ・タイム	20		20		ns
t_{JPH}	JTAG ポートのホールド・タイム	45		45		ns
t_{JPCO}	JTAG ポートの Clock-to-Output 遅延		25		25	ns
t_{JPZX}	JTAG ポートのハイ・インピーダンスから出力確定までの遅延		25		25	ns
t_{JPXZ}	JTAG ポートの確定出力からハイ・インピーダンスまでの遅延		25		25	ns
t_{JSSU}	キャプチャ・レジスタのセットアップ・タイム	20		20		ns
t_{JSH}	キャプチャ・レジスタのホールド・タイム	45		45		ns
t_{JSCO}	アップデート・レジスタの Clock-to-Output 遅延		25		25	ns
t_{JSZX}	アップデート・レジスタのハイ・インピーダンスから出力確定までの遅延		25		25	ns
t_{JSXZ}	アップデート・レジスタの確定出力からハイ・インピーダンスまでの遅延		25		25	ns

図９は各タイミング・パラメータを波形で示したものです。システムの設計にあたっては、これらのタイミング・パラメータを基準にして、あらゆるシステム上でJam Playerが適切に動作するようにしなければなりません。

図9　JTAGの波形とタイミング



Jam PlayerがTAPコントローラを操作する下位レベルのドライバを提供するのに対して、Jam Fileは与えらえたデバイスのプログラムに必要な上位レベルの機能を提供します。デバイスに対するJTAGデータの操作を行うすべてのJam命令はステート・マシンのデータ・レジスタの1部あるいはインストラクション・レジスタの1部を通じてTAPコントローラに与えられ、実行されます。例えば、JTAG命令のロードはTAPコントローラをSHIFT_IRのステートに遷移させ、TDIピンから命令をインストラクション・レジスタにシフト・インさせます。次に、TAPコントローラはRUN_TEST/IDLEのステートに遷移し、ここでラッチされる命令の実行に必要な遅延が実現されます。このプロセスはステート・マシンのデータ・レジスタの一部が変化することを除いてデータ・レジスタのスキャン動作と同じです。

ハイ・レベルなJAM命令としては、JTAGデータ・レジスタをスキャンするためのDRSCAN命令、インストラクション・レジスタをスキャンするためのIRSCAN、ステート・マシンを規定された時間まで特定のステートでウェイトさせるWAITコマンドがあります。TAPコントローラの各レジスタはJAMファイル内の命令に従って、ターゲット・デバイスがプログラムされるまで繰り返しスキャンされます。

Jam命令の詳細については、「*Jam Programming & Test Language Specification*」を参照して下さい。

図10はJam PlayerがJam Fileの解析（parse）を行うときの動作フローを示したものです。DRSCAN、IRSCANまたはWAITの命令が与えられると、Jam PlayerはTCK、TMS、TDIの各ピンに対する適切なデータを生成して、命令を実行させます。この動作フロー図には、DRSCAN、IRSCAN、WAITの各命令に対応したブランチが示されています。Jam Playerは他の命令もサポートしていますが、簡略化するためにこのフローでは示されていません。

図10　Jam Playerの下位レベル動作を示すフロー・ダイヤグラム(1/2)

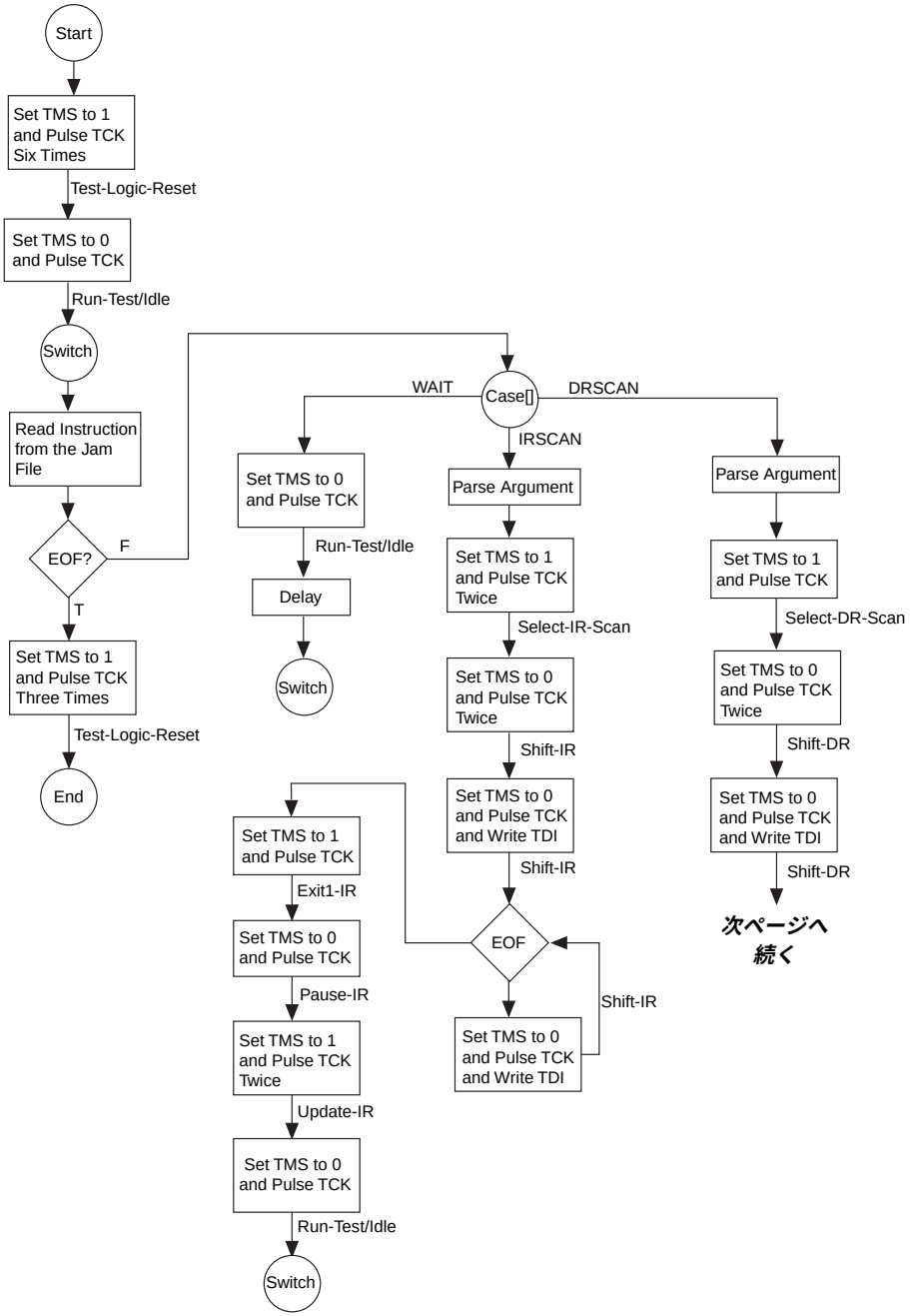
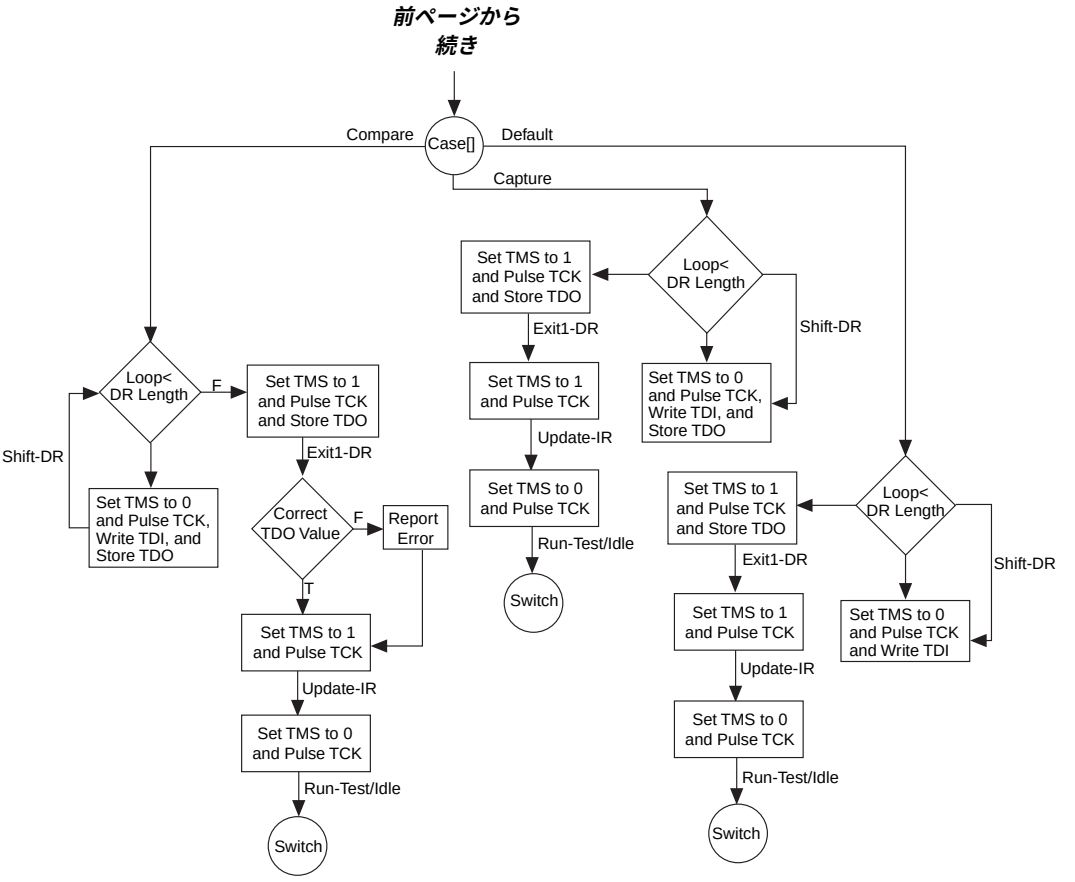


図10　Jam Playerの下位レベル動作を示すフロー・ダイヤグラム(2/2)



結論

エンベデッド・プロセッサによるイン・システム・プログラミングから得られる利点の実現には、小さなファイル・サイズ、使いやすさ、プラットフォームに対する非依存性など、システム側に要求される各項目を満たしているJamプログラミングおよびテスト用言語の使用が最適です。エンベデッド・プロセッサによるイン・システム・プログラミングにJam言語を使用することによって、フィールドでのアップ・グレードがサポートされ、デザインの試作が容易になると共に製造工程が短縮されます。これらの利点は、最終製品のライフ・サイクルを延長し、品質と柔軟性を強化します。また、プログラムされたデバイスの在庫や管理が不要になるため、デバイスの在庫レベルを低減させることができます。

日本アルテラ株式会社

〒163-04
東京都新宿区西新宿2-1-1
新宿三井ビル私書箱261号
TEL. 03-3340-9480 FAX. 03-3340-9487

本社 Altera Corporation

101 Innovation Drive, San Jose,
CA 95134
TEL: (408) 544-7000
FAX: (408) 944-0952
http://www.altera.com