

JTAGオンチップ・デバッグによる エンベデッドLinuxデバイスの開発時間短縮

目次

- はじめに 1
- Linux OSの概要 1
- エンベデッド Linux デバイスのデバッグの複雑さ 2
- ブートローダ 2
 - ブートローダのデバッグに関する問題 2
 - ブートローダのデバッグの簡素化 3
- Linux カーネルとカーネル・モジュール 3
 - JTAG を使用したデバッグ 3
- Linux アプリケーションのデバッグ 4
- JTAG ベースの Linux デバッグのための
ウインドリバー・ソリューション 5
- Linux デバッグのための
先進的ウインドリバー・ソリューション 5
 - Linux 開発プロセス全体を対象とした
ウインドリバー・オンチップ・デバッグ 6
- おわりに 6

はじめに

従来、組み込みLinux製品のデバッグには、ハードウェア・ツールとソフトウェア・ツールの両方が必要でした。すなわち、ハードウェア立ち上げ用のJTAGツールと、ソフトウェア開発用のエージェントベース・ツールです。これらのJTAGツールとエージェントベース・ツールは、しばしば重要な問題を解決しましたが、組み込みLinuxの開発用にデザインされたものではありませんでした。

ウインドリバーは、従来のJTAGハードウェア・デバッグと、Linuxカーネル・コンフィギュレーション、パッチ管理、ユーザスペースのアプリケーションの開発／デバッグ／解析とを結合させ、Wind River Workbenchと呼ばれるEclipseベースの統合開発環境 (IDE) にまとめることで、開発者のLinuxデバッグ方法を変えました。これにより開発者は、従来のエージェ

ントベース・ソリューションを使うことが技術的に、あるいはコスト面で困難な場合でも、JTAG接続を利用可能とすることができます。

JTAG接続が使用可能な場合は、一般的に2つのシナリオが考えられます。エージェントベース・デバッグにEthernet接続を使用できない場合と、Linuxカーネルまたはユーザスペースで発生した問題を開発者が解決しなければならないなど、システムモードのデバッグが必要とされる場合です。ウインドリバーの次世代JTAGツールを使用すれば、開発者はハードウェア、ブートローダ、Linuxカーネル、ユーザスペースのデバッグを行い、システム・クラッシュにつながるイベントや、カーネル、ユーザスペース、ターゲットの間で発生するその他の問題を見つけ出すことができます。さらに、Wind River Linux OSを使用する開発者は、エージェント・デバッグとJTAGデバッグの間をシームレスに移動できます。オンチップ・デバッグにおけるこれらの革新は、エージェントベース・デバッグが使用できない場合、あるいはエージェントベース・デバッグでは費用のかかりすぎるプロジェクトに取り組む場合に、特にお奨めします。

Linux OSの概要

Linuxのトラブルシューティングの第一歩は、デバッグにより相互動作を解析することが必要な領域とLinux OSを理解することです。通常、Linuxは複数の相互動作コンポーネントで構成されており、複雑なメモリ・マッピングと管理方式を採用しています。Linuxベースの組み込みシステムのデバッグには、OSの動作とハードウェアとの相互動作を明らかにできる強力なツールが必要です。

Linux OSの主要コンポーネントは、Linuxカーネル、カーネル・モジュール、アプリケーション・ソフトウェアです。LinuxカーネルはこのOSのコアであり、絶対的な権限を持っています。Linuxカーネル・モジュールは動的にロード／アンロードされるOSの構成要素であり、しばしばデバイスドライバに使用されます。

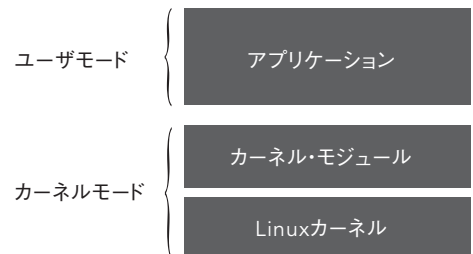


図1:Linux OSのコンポーネント

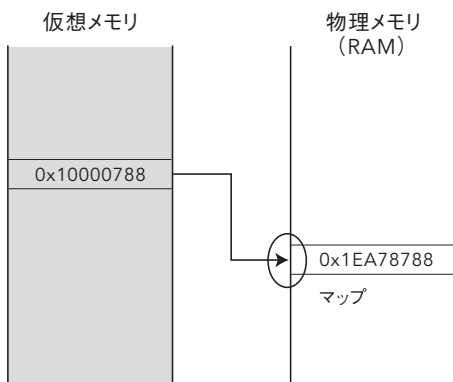


図2:Linuxのメモリ管理

ロードされたLinuxカーネル・モジュールは、Linuxカーネルと同じレベルの権限を持ちます。Linuxカーネルはすべてのソフトウェアを分け、ユーザーモードあるいはLinuxカーネルモードで実行します。Linuxシステム上では、ユーザーモードで実行されるアプリケーションの権限は制約されます。たとえば、これらのアプリケーションがLinuxカーネルのメモリやハードウェアに直接アクセスすることはできません。これは、アプリケーションが基盤システムに損害を与えないようにするためです。したがって、Linuxカーネルは、すべてのシステムレベルのアクセスを管理することになります。カーネルは、アプリケーション・ソフトウェアが周辺機器やメモリを含めたシステムに、一定の管理の下でアクセスできるようにします。

Linuxシステムでは、通常、アプリケーションはユーザーモードで実行され、デバイス・ドライバはカーネルモードで実行されます。この方式は、システム・クラッシュの危険性やシステムの可用性低下からコア（Linuxカーネル）を保護するという点では有効ですが、性能低下とのトレードオフを強いられます。通常は、ユーザスペース・アプリケーションよりもLinuxカーネル・モジュールの方が高性能なためです。

多くの組み込みデバイスで使用されている、より単純な「フラット」にあるいは「静的に」アロケートされた8ビット、16ビット、32ビットOSとは異なり、Linuxでは仮想アドレッシングという高度な技術が確立されています。仮想アドレッシングにより、Linuxのカーネル・モジュールとアプリケーションは、実際のソフトウェアの位置が物理メモリ上の複数のメモリ領域に断片化して置かれていても、連続的なメモリ空間に置かれたような形でシステムにロードして実行できます。Linuxカーネルは、Linuxのカーネル・モジュールとアプリケーションのために、メモリ・マッピングとメモリ・アロケーションを行います。LinuxカーネルとLinuxカーネル・モジュール用に使われる静的マッピングのメモリ管理では、定義された定数（たとえばオフセット）によって仮想アドレッシングから物理アドレッシングへのマッピングが行われます。ユーザーモード・アプリケーションに使われる仮想メモリから物理メモリへの動的マッピングでは、複雑な方法によって物理メモリへのマッピングが行われることがあります。

Linuxカーネルは、プロセスをメモリにアロケートして実行することもできます。プロセスには、Linuxカーネル・モジュールやユーザーモード・アプリケーションが含まれる場合があります。特定のタスクを完了させるためにメモリにアロケートし、不要になった時は削除できるので、これらのプロセスによればより望ましい形でシステム・メモリを管理できます。したがって、アプリケーションからLinuxカーネルおよびカーネル・モジュールへとメモリをトラバースする場合は、Linuxカーネルの静的および動的なアドレス変換能力、ならびにプロセスをメモリに動的にアロケートする能力を考慮するよう注意が必要です。

エンベデッドLinuxデバイスのデバッグの複雑さ

Linuxは、組み込みデバイスの分野で急速な成長を続けています。技術市場調査会社Venture Development Corporation (VDC) によれば、新規プロジェクトの23%がLinuxを使用する見込みです*1。Linuxエンベデッド・プロジェクトにおける開発は、ブートローダ、Linuxカーネル、カーネル・モジュール、アプリケーションなど多岐にわたるので、そのデバッグは極めて複雑なものになることが予想されます。Linux開発者が解決しなければならない問題には、ブートローダ用のターゲット・コンフィギュレーション・ファイルの確立、ユーザーモードからカーネルモードへのLinux仮想アドレッシングのトラバース、カーネル・シンボル情報のマッピング、ユーザスペースとカーネルスペースにまたがる難しい欠陥のトラブルシューティングが含まれます。これらの問題はいずれも、カーネルGNUデバッガ(KGDB)やGNUデバッガ(GDB)といったエージェントベースのデバッグ・ソリューションでは解決が困難です。

Linuxのデバッグにおける課題

- ハードウェアの初期化
- ブートローダのデバッグ
- 命令キャッシュとデータ・キャッシュへのアクセス
- Linuxカーネルの初期化コードへのアクセス
- アプリケーションとLinuxカーネル間の連携のデバッグ

ブートローダ

Linuxは、ブートローダを使用してOSを起動します。ブートローダは、フラッシュメモリ、あるいはその他の不揮発性メモリに置かれるコードで、システムの電源投入またはリセット後に実行されます。CPU、メモリ、重要I/Oおよび周辺機器の低レベルの初期化は、ブートローダによって行われます。ブートローダの実行が開始されると、ブートローダは自身自身をRAMにコピーしてLinux OSを起動します。

ブートローダのデバッグに関する問題

ブートローダのデバッグは極めて複雑です。ブートローダはハードウェア中心のコードであり、開発者は、起動後にフラッシュメモリからRAMへブートローダを再配置させなければならないからです。今日のシステムオンチップ(SoC)プロセッサには数百個におよぶコンフィギュレーション・レジスタがあり、これらをすべて初期化する必要があります。このためには、特定の設定に関し、数千ページにおよぶドキュメンテーションを詳しく調べなければなりません。レジスタの設定を誤ると、Linuxカーネルやデバッグ・アプリケーションを起動した時に、ダウンストリーム側で問題が発生する恐れがあります。手動で編集を行ってレジスタをセットしていくのは、非常に煩雑です。

ブートローダ開発に伴うもう1つの一般的な課題は、ブートローダがRAMにLinuxをロードしてOSを起動する際に発生する問題のデバッグです。ブートローダがLinuxを物理メモリにロードする際、カーネルはカーネル用の仮想メモリマップを作成しますが、多くの場合、これがカーネルの使用アドレスを特定することを難しくします。したがって、RAMからカーネルが実行を開始すると、Linux OS用にシンボル・テーブルを再配置する方法を判定するのは非常に複雑になる可能性があります。残念ながら、エージェントベースのデバッグ・ソリューションではブートローダのデバッグを行うことはできません。このプロセス中は

OSが機能しないからです。したがって、開発者は、この重要プロセスのデバッグについてはJTAGツールに依存することになります。

ブートローダの開発とデバッグに多大の時間を費やすことになると、開発者は、デバイスのソフトウェアおよびアプリケーションの安定化とデバッグに集中できなくなります。

ブートローダのデバッグの簡素化

KGDBやGDBなどのエージェントベースのデバッグ・ソリューションは、ブートローダのデバッグには使用できません。しかし、JTAGデバッグ・ソリューションは、開発者によるブートローダの迅速かつ効果的なトラブルシューティングとテストを支援する、強力な機能を備えています。JTAGデバッグ・ソリューションを使用すれば、開発者は、レジスタ値のセットやレジスタ内容の精査を容易に行うことができます。また、ハードウェア・ブレイクポイントをセットできるので、フラッシュメモリ内でコードをステップ実行し、エラー箇所を迅速に特定することが可能です。ソースコードとアセンブリコードを並べて表示するためのミックス・モードと逆アセンブリとをサポートするIDEにより、デバッグ・プロセスが容易になります。強力なシンボル管理機能は、フラッシュメモリからRAMへのコードの再配置を考慮することによってデバッグを支援します。

一部のJTAGデバッグ・ソリューションには、もう1つ、ブートローダなしでLinuxカーネルをロードできるという利点があります。この機能はブートライン・パラメータを使用することによって実現されるもので、ブートローダが完成して使用できるようになる前にシステム開発を開始しようとするプロジェクト・マネージャにとっては特に有効です。ブートライン機能を持つJTAGベース・ソリューションは、ブートローダの並行開発とOSの安定化を可能にし、開発スケジュールの短縮を支援します。

Linuxカーネルとカーネル・モジュール

Linuxカーネルとカーネル・モジュールは、Linux OSのコア・コンポーネントです。Linuxカーネルは、ブートローダによってシステムが初期化された後、最初に起動する部分です。Linuxカーネル・モジュールは、その後必要に応じてロードされます。

OSの立ち上げ期間中、開発者は、Linux OSの最適化やカスタマイズ、ならびにカーネル・モジュールの開発に重点を置きます。ハードウェアとソフトウェアの連携は、このフェーズの間、極めて厳密に監視されます。Linuxカーネルのデバッグを行うには、開発者がレジスタ、キャッシュ、その他の低レベル・データを確実に把握する必要があります。Linuxカーネル・モジュールのデバッグでRAM内の動的にアロケートされたメモリを扱う際は、初期化コードの透明性も必要です。

KGDBには、安定したLinuxカーネルと、エージェントを機能させるためのデバイス・ドライバなどのカスタム・ハードウェア・インタフェースが必要です。エージェントベースのデバッグではハードウェア内部の状態が分からず、ハードウェアとLinuxカーネルの連携を理解するために必要な診断機能も十分ではありません。エージェントはLinuxカーネルの計測を必要としますが、これは、組込みデバイスに副次的な悪影響を及ぼす恐れがあります。

エージェントを使用してLinuxカーネルとカーネル・モジュールをデバッグする際に考慮すべき他の点は、ブレイクポイントがヒットされた際に

システムが停止あるいはフリーズすることです。たとえば、KGDBは、開発者がシステムの現在状況を確認できるようにCPUを停止することはありません(特にマルチコアあるいはマルチプロセッシング環境)。エージェントを機能させるにはシステムが機能している必要がありますので、KGDBはクラッシュしたシステムをデバッグする助けにはなりません。さらにKGDBには、ホストシステムからターゲットへの通信を行うために、Ethernetなどの通信ポートが必要です。したがって、Linuxカーネルモードのデバッグにエージェントを使用するには、IPスタックが機能すること、Linuxカーネルが安定していること、デバイス・ドライバが機能することに加えて、通信チャンネルが確立されている必要があります。これは、このような通信機能を使用できないか、通信機能自体をデバッグする必要があるOSの安定を図る上で障害となります。

JTAGを使用したデバッグ

ターゲット・システムに対してLinuxカーネルが妥当であることを確認し、安定したものとするには、Linuxカーネルおよびカーネル・モジュールとともに動作する包括的なデバッグ・ソリューションが必要です。JTAGベースのデバッグ・ソリューションは、このワークフローにおいて重要な役割を果たします。これらのJTAGベース・デバッグ・ツールの特長には、ローカル／グローバル・シンボルおよびレジスタの表示機能や、データおよび命令キャッシュの内容表示機能が含まれます。一部の商品化されたJTAGソリューションでは仮想メモリから物理メモリへのシームレスなアドレス・マッピングが可能で、開発者は、正しいメモリのアドレスや内容を確認することができます。

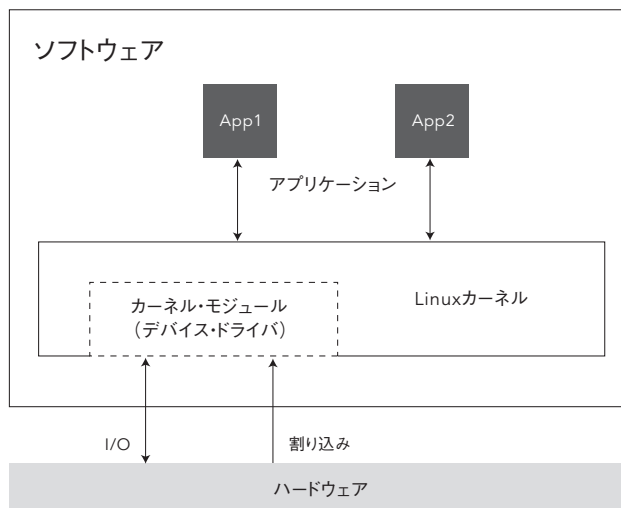


図3: Linuxにおけるソフトウェアとハードウェアの相互作用

これらのソリューションを使用すればLinuxカーネル・モジュールの初期化のデバッグが可能で、ターゲット・システムとの接続解除と再接続を行うことなく、カーネル・モジュールを複数回にわたってロード／アンロードすることができます。

システムを完全に停止してOSおよびアプリケーションの状態を確認できる点が、JTAGデバッグ・ソリューションの大きな特長です。システムモード・デバッグと呼ばれるこの機能は、LinuxカーネルおよびLinuxカーネル・モジュールのデバッグに非常に有効です。システムモード・デバッグを使用すれば、開発者は、プロセッサ、OS、すべてのスレッド、割り込み処理を含むシステム全体を停止させることができます。このようにしてシステムを停止させることにより、ハードウェアとソフトウェアの状態が詳細に分かります。また、システムの動作を再開させることも、その後コードをステップ実行することも可能です。

JTAGソリューションは、KGDBが機能できない場合、特にLinuxのカーネル・パニックやターゲット・クラッシュが発生した状態でも機能します。これらの機能は、デバイスドライバやOSの安定化を図る上で特に有効です。

Linuxアプリケーションのデバッグ

Linuxアプリケーションは、Linuxカーネル管理下で実行されるユーザプログラムです。Linuxアプリケーションは、システムコールを行うことによってシステム・リソースにアクセスします。Linuxカーネルはこのシステムコールの処理を行い、ハードウェアおよびメモリへのアクセスを提供する方法を決定します。

ユーザモード・アプリケーションをデバッグするには、開発者がスレッドを開始/停止させて変数やスタックを表示する機能を使って、アプリケーション・スレッドに直接アクセスする必要があります。アプリケーションは多くのプロセスやスレッドで構成されるので、アプリケーションに影響を与える可能性のあるスレッドを含め、デバッグ対象のアプリケーション・スレッドに関するすべてのスレッドを停止させなければならないことがあります。また、さまざまなプロセスやCPUにまたがって周辺機器レジスタを監視する必要が生じることもあります。GDBはスレッド・レベルでのみ作動し、停止できるスレッドは1つだけです。GDBは、システム全体を停止させたり、複数スレッドを同時に停止させたりすることはできません。

ユーザモードでデバッグを行っている開発者は、ユーザモードからLinuxカーネルモードに入ってシステムコールを行い、再びユーザモードに戻る必要があります。

Linuxは仮想メモリ管理構造を取っているため、2つのモード間を移動する際のメモリ・アドレス追跡には困難が伴います。同じ物理アドレスに頼ることはできません。このプロセスでメモリ・マッピングとメモリ・アロケーションを正確に追跡することは、非常に重要です。結果として、エージェント・ソリューションでは、GDBとKGDBの両方を使用して、Linuxカーネルとカーネル・モジュールへのシステムコールを追跡しなければなりません。複数のエージェントベース・デバッグ・ツールを使用することは、デバッグ・プロセスを複雑にする恐れがあります。

また、Linuxアプリケーションを開発する際、組込みシステムのデバイス上にEthernetまたはシリアル通信ポートがない場合もあります。このような場合、エージェントベースのデバッグを行うことはできません。デバイスに通信ポートがあったとしても、エージェントを使用するにはインタフェースが使用可能な状

態になっている必要があります。通信ポートが使用可能な状態になかったりデバッグが必要だったりする場合、あるいはIPスタックまたはエージェント・ソフトウェア用にメモリを使用できない場合は、エージェントベースのデバッグを行うことはできません。

JTAGベースのデバッグなら、Linuxユーザモードで作動しているアプリケーションの状態を詳しく知ることができます。システムコールを行うアプリケーションの場合は、デュアルモードJTAGデバッグ・ソリューションを使用すれば、Linuxカーネルモードおよびユーザモードにおける状態を的確に把握できます。これらはいずれも、Linuxカーネルの計測なしで行われます。デュアルモード・ソリューションを使用すれば、すべてのアプリケーション・スレッド、そのコンテキスト、どのスレッドがLinuxカーネルに入っているかということや、使用パラメータおよびスレッド変数などの確認が可能です。通信ポートがないデバイス(たとえばモバイル・デバイス、医療用デバイス、自動車システムなど)については、JTAGベース・ソリューションが、エージェントベース・デバッグに代わって優れた機能を発揮します。

プロセッサを停止させてOSおよびすべてのスレッドの状態を確認できるシステムモード・デバッグを使用することで、マルチスレッド・アプリケーションのデバッグは簡略化できます。すでに述べたように、複数スレッドの相互動作の結果として多くの問題が発生します。エージェントベースのデバッグでは、すべてのスレッドを同時に停止させることができないので、問題を特定するのが困難な場合があります。これは、プロジェクトのデバッグに非常に時間がかかる恐れがあることを意味します。システムモード・デバッグを利用すれば、システムの現在の状態(例えば各スレッドの状態や変数など)を詳細に知ることができ、特定の時点でのシステムの状態を即座に余すところなく確認できるので、エージェントベースのデバッグよりも良好な結果を得ることができます。

JTAGベース・デバッグ・ソリューションのターゲット・ハードウェアに対する接続は、非侵入型です。JTAGは、すでに実行中のエラーの発生したシステムに接続できます。また、プロセッサ・レジスタの状態を変更することなく接続し、Linuxカーネルとアプリケーションのコンテキストに

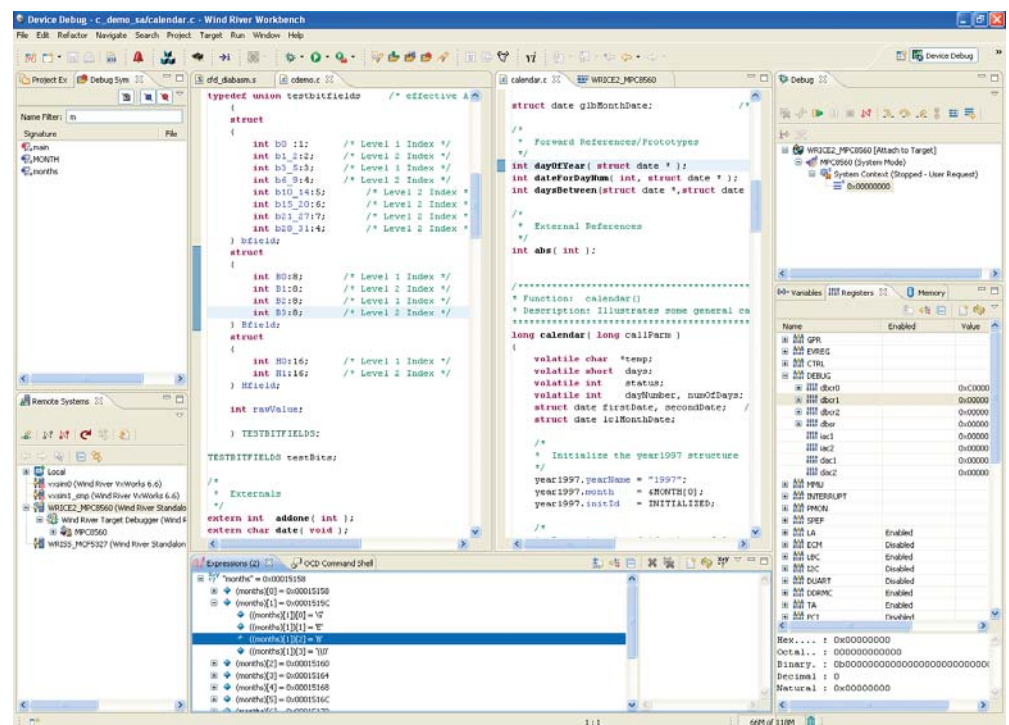


図4:Linuxアプリケーション・デバッグ用のEclipseベースIDE

同期してデバッグを行うことが可能です。その一例として、デッドロック状態にあるLinuxベースのシステム・スタックがあげられます。JTAGソリューションを使用すれば、開発者は、状態を変更することなくターゲットシステムに接続することができます。開発者はこの接続後に、Linuxカーネルのオブジェクト、スレッドを特定するためのアプリケーション・コンテキスト、システムコール、システムコールに使われたパラメータで障害状態の原因となったものを特定することが可能。エージェントベースの開発ソリューションを使用できない場合は、単一のエンドツーエンド・ソリューションが特に便利です。JTAGによるデバッグは、開発ツールの簡略化を通じてデバッグ時間を短縮します。

JTAGベースのLinuxデバッグのための ウインドリバー・ソリューション

ウインドリバーのJTAGベース・オンチップ・デバッグ・ソリューションは、一般的なエージェントベース・ソリューションに代わるものではなく、補完するものです。エージェントベースのKGDBやGDBなど、JTAGベースでないソリューションは、ブートローダのデバッグ、OS安定化、デバイス・ドライバのデバッグといった初期段階の開発には向いていません。このことは、エージェントベースのデバッグのみに依存する製品の開発に影響を及ぼすことがあります。

通常、ほとんどのデバイスは、デバッグおよびテスト用のJTAGポートを備えています。しかし多くのコンシューマ製品、特に価格に敏感なもの、スペースの制約があるもの、あるいはメモリに制約があるものなどには、エージェントベース・デバッグを行うために必要な通信ポートがありません。他にも、メモリスペースや物理的スペースの制約が厳しい場合、セキュリティ上の問題がある場合、あるいはIPスタックがない場合などは、エージェントベースのデバッグは使用できません。JTAGベースのソリューションは、このような環境に最適です。

また、JTAG は、エージェントを使用したデバッグでは得ることのできない、システム全体に関する重要な情報を提供します。デュアルユーザおよび Linux カーネルモードという Linux の構造からすると、アプリケーション、OS、基礎となるハードウェアに関するすべての情報を同時に得る必要がありますが、GDB および KGDB を使用するエージェントベースのデバッグでは、開発者が困難に直面することが考えられます。JTAG ソリューションはこれらの領域に関する重要な情報を同時に提供することができ、それによってデバッグをより容易かつ迅速に行うことが可能になります。

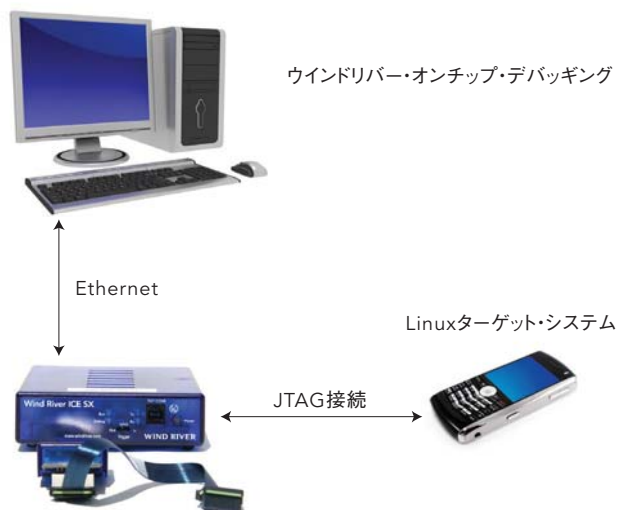


図5:ウインドリバーのJTAGベース・オンチップ・デバッグ・ソリューション

Linuxデバッグのための 先進的ウインドリバー・ソリューション

業界標準のEclipseをベースとしたプラットフォームであるWind River Workbench, On-Chip Debugging Editionは、組込みLinuxシステム用にJTAGベースの先進的なデバッグ機能を提供します。Workbench On-Chip Debuggingは、Wind River Linuxをサポートしています。自社作成 (roll-your-own) Linuxまたは半導体Linuxディストリビューションについて、ウインドリバーはオンチップ・デバッグ機能を拡張して、kernel.org LinuxのJTAGベースのカーネルモードとユーザモードのデバッグを行えるようにしました。ウインドリバーのデバッグ・ソリューションは、最も先進的なマルチコア・プロセッサを含め、さまざまなターゲット・ハードウェアをサポートしています。

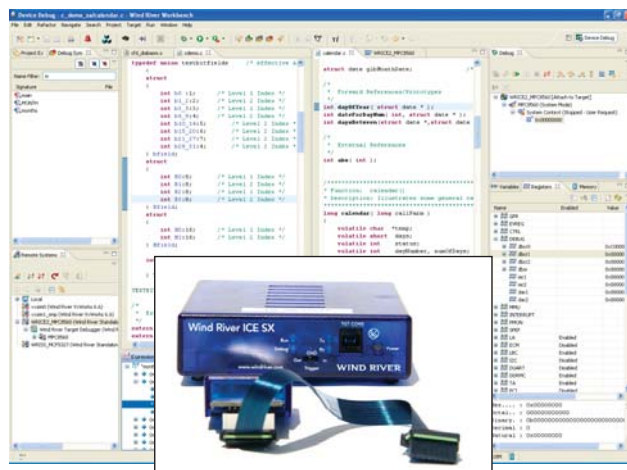


図6:JTAGベース・デバッグのためのWind River Workbench, On-Chip Debugging EditionとWind River ICE

Workbench On-Chip Debugging

- 新規ハードウェア上でのLinux立ち上げ時の問題を解決するためのブートローダ開発
- OSおよびデバイス・ドライバに関する問題を解決するためのカーネルモード・デバッグ
- カーネル計測、通信チャンネル、IPスタックなしでのユーザモード・デバッグ
- Linuxカーネルへの入り方とどのスレッド(1つまたは複数)を介したか、使用されたパラメータ、スレッド変数を含めたシステムおよびアプリケーションのコンテキストを開発者が確認できるユーザモード・デバッグの先進的機能
- 包括的なシステムモード・デバッグ:プロセッサ全体およびすべてのスレッドを停止させるブレークポイント

Linux開発プロセス全体を対象とした ウインドリバー・オンチップ・デバッグ

Wind River Workbenchを使用すれば、Linux開発者はJTAG接続を使用して、ハードウェア立ち上げ、Linuxカーネル／ミドルウェア／ユーザモード・アプリケーションのデバッグ、必要に応じたWind River Linux用エージェントベース・デバッグへのシームレスな移行などを、すべて同じIDE内で行える柔軟性を手に入れることができます。これらの機能により、複数の開発チーム間の協力が促進され、問題解決に必要な時間が短縮されます。

ウインドリバーのソリューションは、ますます複雑化するシステムのニーズに対応できるようデザインされています。ターゲット・システムへのリモート・アクセスも可能で、世界中に分散した開発チームの生産性向上を実現することができます。接続およびコンフィギュレーション管理は複数ターゲットへの接続を簡素化し、1つのユーザインタフェースを介してシステム全体のデバッグを支援します。これらの接続は、コア、

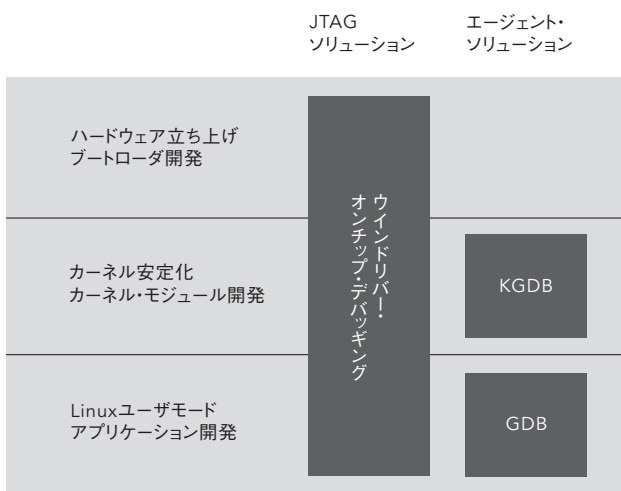


図7:ウインドリバーのオンチップ・デバッグは
広範なLinux開発をカバー

プロセッサ、あるいはプロセスの場合もあります。ウインドリバーのオンチップ・デバッグは、JTAGツールおよびエージェントを含め複数のデバッグ接続タイプをサポートしており、ブートローダ立ち上げからLinuxカーネルおよびLinuxカーネル・モジュールの安定化、アプリケーション開発まで、Linuxシステムのデバッグに最大限の柔軟性を提供します。

おわりに

組込みシステムがより複雑になるにつれて、デバッグは開発者にとってこれまで以上に困難な課題となってきました。ユーザモードおよびLinuxカーネルモードのデバッグに従来使われてきたエージェントベースのLinuxデバッグ・ソリューションは、今日の複雑な環境下ではその効果と効率が低下しつつあります。JTAGデバッグはLinux開発において重要な役割を果たし、Linux環境用に適切にデザインされたEclipseなどの標準ベースの開発環境と統合すれば、編集-コンパイル-デバッグのプロセスを改善することができます。

現在、ウインドリバーが提供するようなソリューションは、エージェントベース・デバッグが使用できない場合や、コスト的に見合わない場合の強力な代替手段となるオンチップ・デバッグの使用を拡張する革新的な解決策です。オンチップ・デバッグにおけるこれらのユニークな機能は、エージェントベース・デバッグと効果的に融合し、極めて複雑な環境下においてもデバッグ性能を向上させます。市場投入までの時間短縮がますます重要になる中、企業はウインドリバーのオンチップ・デバッグ・ソリューションを使用することによって、開発時間を短縮しコストを削減することができます。

注記

※1 VDC, "Linux in the Embedded Systems Market," The Embedded Software Market Intelligence Program, November 2007. (「組込みシステム市場におけるLinux」、組込ソフトウェア市場情報分析プログラム、2007年11月)

ウインドリバーはスマートデバイス搭載ソフトウェアの最適化(DSO)をワールドワイドに提供するリーディングカンパニーです。企業がスマートデバイスに搭載するソフトウェアを、品質および信頼性のさらなる向上を実現しつつ、リーズナブルなコストで開発することを可能にし、早期にマーケットへ投入することを支援します。

WIND RIVER ウインドリバー株式会社

東京本社 〒150-0012 東京都渋谷区広尾1-1-39 恵比寿プライムスクエアタワー TEL.03-5778-6001(代表) FAX.03-5778-6002
大阪営業所 〒532-0011 大阪市淀川区西中島7-5-25 新大阪ドビル TEL.06-6100-5760(代表) FAX.06-6100-5761
E-mail:info-jp@windriver.com http://www.windriver.co.jp

登録商標: Wind River, Wind Riverロゴ, Tornado, VxWorksは、Wind River Systems, Inc.の登録商標または商標です。記載されているすべての名称は、各社の登録商標、商標またはサービスマークです。