組み込みデバッグの最新動向

January 20, 2006

1. 背景

日本では古くから組み込み機器の開発が行われてきましたが、いわゆる徒弟制度に近い教育により、上司から部下へ組み込み機器上で動作するソフトウェアの開発手法が伝搬されてきました。しかしながら急激に増大するソフトウェア開発の量により、これまでのような手法では、要求される品質および納期を維持できなくなくなりつつあります。そこでこの記事では、組み込み機器の開発手法、特にデバッグ手法に焦点を当て、新しいトレンドについて解説します。

組み込み開発の最新動向

植田省司

1-1 従来の組み込みシステムのデバッグ

現在の開発スタイルの多くは、テキストエディタを用いてソースコードを書き、makefileで依存関係を記述し、コマンドラインコンパイラでコンパイル、リンクを行い、ICE（インサーキットエミュレータ）などを利用してターゲット上でデバッグするという形態です（図1）。デバッガが正常に動作しない状況では、勘だけを頼りとするデバッグなどがなされています。

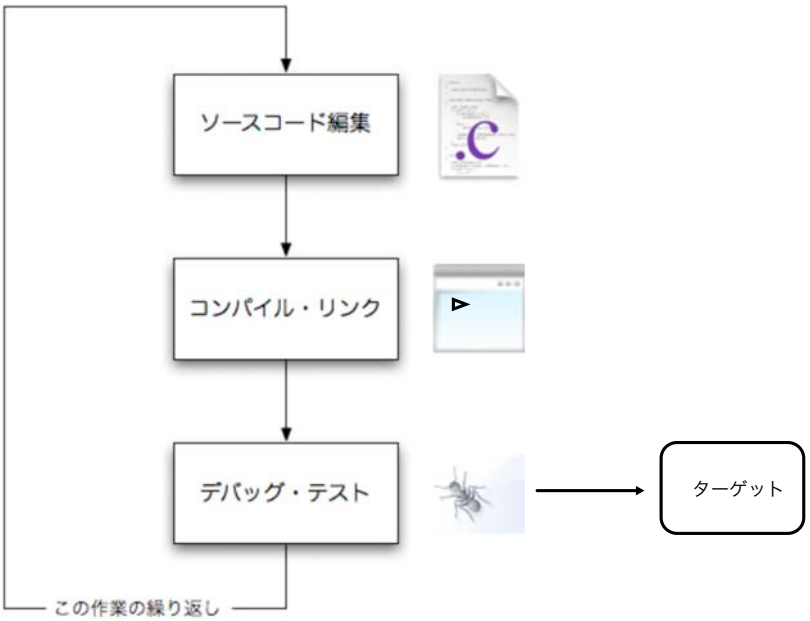


図１　典型的なデバッグサイクル

1-2 最近の組み込みシステムの状況

近年、ソフトウェアの規模の拡大などにより、ブラックボックス化されたアプリケーションをデバッグする必要が生じています。多人数によるチーム開発では、同一アプリケーションに開発者自身が理解していないコードが入り、実行されます。そのような状態では「快適に動作するデバッガ」なしでは、デバッグ作業がきわめて困難になります。

プログラムのパフォーマンスを十分に発揮するためには、ボトルネックルーチンを容易に発見し、それを最適化する必要もあります。この点でも、ブラックボックス化されたソフトウェア開発においては、「プロファイラ」があれば非常に有力なツールとなります。

これら組み込みソフトウェアの開発は、各社各様で、Windows 上でのVisual Studioの様なほぼ統一された環境が存在せず、情報も分散しがちです。したがって開発環境のノウハウなどはチーム内でしか蓄積されない状況が見られます。最近になって「Eclipseという統合開発プラットフォーム」で組み込みソフトウェア開発を行おうという機運もあり、今後状況が大きく変わる可能性があります。

そのような状況で現在、デジタル家電を中心に情報家電（Set Top Box、Digital TV、ゲームソフトウェア）、カーナビゲーション、携帯電話などの組み込み機器のソフトウェア開発がどんどんと巨大化してきました。組み込みシステムに使用されるCPUも「実CPUにデバッグ機能

（OCD: On Chip Debug）」を搭載したCPUが主流となり、さらに高度な「マルチコアCPU」も使用されつつあります。

また、ターゲットシステムのOSについても、従来はμITRONを中心としたリアルタイムカーネルを利用する場合が多くありましたが、近年は「Linux」が組み込み機器へ普及しつつあり、開発スタイルも大幅に変化してきました。

1-3 組み込みLinuxなどのOSを用いた新しいソフトウェア開発

組み込み機器はネットワーク接続の必要性などにより、より高度なOSを求めるようになってきました。Linuxはその最有力候補OSであり、その他に、WindowsCE、T-Engine、(携帯機器用の) Symbian OSなどがあります。これらのOSは、提供する機能が増えていることもありますが、複雑度が増しています。一概にLinuxといっても、さまざまなLinuxがあり、全てが同一とはいえません。

しかし、一度でもこれらのOSを利用するターゲットシステムの開発を経験すると、その経験は、OSの異なるバージョン、ディストリビューションにまたがっても生かすことができます。特にLinuxはインターネットと共に普及してきたこともあり、インターネット上にある情報量は圧倒的です（が反面情報が氾濫しすぎているともいえます）。

ここでは、これからの組み込み機器開発で重要と考えられるデバッグ環境について、On Chip Debug機能を利用した「JTAG-ICEデバッガ」（快適に動作するデバッガ）、「マルチコアCPUのデバッグ」を可能とする仕組み、「組み込みLinuxのデバッグ技術および考慮すべき点」、「パフォーマンス解析」、「JTAGデバッガの応用機能」、組み込みシステム開発ではまだ新参者の「Eclipseという統合開発環境」について解説します。

2. JTAG-ICEデバッガ

2-1 Full ICEの問題点

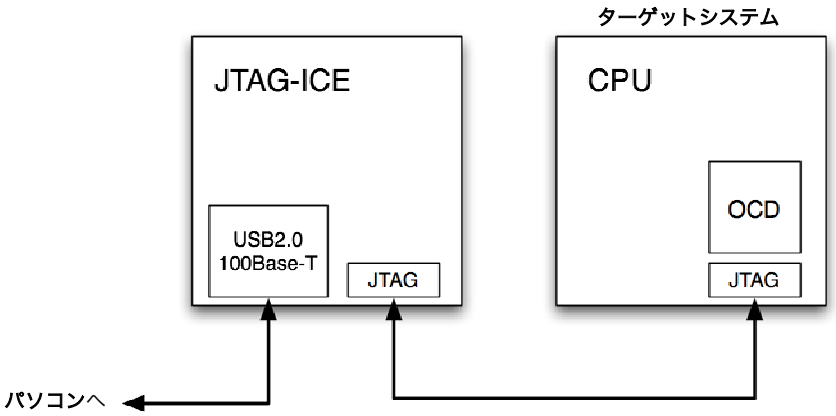
近年のCPUの性能向上に伴い、従来用いられてきたFull ICE（エミュレータとも呼びます）を利用した開発が困難になってきました。困難な理由として、以下のような理由があげられます。

- CPUクロックの上昇。
- キャッシュなどにより、CPUの内部で完結しCPUバスのモニタだけでは観測できない情報ある。
- パッケージの変化（微小化）により観測用端子を接続できない。
- エバCPUがコスト的に見合わなくなり提供できない。

2-2 Full ICEからJTAG-ICE への移行

Full ICEは、限りなく実際のCPUに似せたエバCPUを実際のCPU代わりに利用する手法です。近年は2-1の理由により特にハイエンドCPUでのFull ICEの採用事例が無くなりつつあります（4ビットや8ビットCPUではFull ICEによるデバッグは現在でも有効な手法です）。半導体はエバCPUを製造する代わりに、実CPUにデバッグ機能を搭載しました（OCD: On Chip Debug）。OCDの機能を利用したデバッガが1998年頃に登場したJTAG-ICEです（図2）。

図2 JTAG-ICEの基本形



JTAGはピン間接続をテストするために開発された手法ですが、ある時点からOCD機能を利用する汎用シリアル通信ポートとしても利用されることが多くなり、現在のJTAG-ICEの構成となりました。JTAG-ICEでは、CPUに搭載されたOCDとJTAGポートを利用して通信し、CPUに対するデバッグ機能はOCDが提供します。

OCDはCPU各社各様の仕様で実装されており、N-wire、 EJTAG、BDM（Background Debug Mode）などがあります。JTAG-ICEはこれらの異なるOCDの実装に合わせてデバッガを実装し、JTAGポートを経由してデバッグを行います。

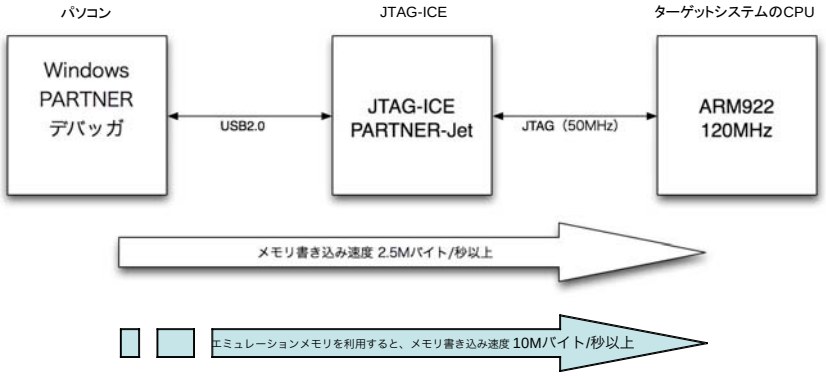
近年のCPUはクロックが数百MHzから1GHzへと上昇していますが、ICEのインターフェースとなるJTAGクロックは数KHｚから100 MHzの周波数で動作します。CPUクロックがさらに上昇してもJTAGクロック周波数は上昇しないので、今後これまで以上に高速なクロックを持つCPUが登場してもJTAG-ICEでデバッグ可能です。

さらに、Full ICEとJTAG-ICEの価格を比較すると、JTAG-ICEにはかなりの価格メリットがあります。上記のような理由と合わせて、Full ICEからJTAG-ICE にパラダイムシフトが起こりました。JTAG- ICEが利用しているJTAGのクロック数は十分に高速です。

弊社のJTAG-ICE「PARTNER-Jet」では、JTAG制御回路をFPGAを用いて独自開発し、JTAGポートの帯域を最大限使用可能な設計になっていますので十分に高速です。パソコンに「PARTNER-Jet」をコントロールし、ターゲットをデバッグする、コントロールソフト「PARTNERデバッガ」をインストールして、ハードウェア「PARTNER-Jet」とソフトウェア「PARTNERデバッガ」の組み合わせでJTAG-ICEとして使用します。例え

ば、SH4ターゲットとUSB2.0でパソコンと接続した場合、実機上のメモリへのダウンロードは約3Mバイト/秒もの高速が確保され、Full ICEよりも高速な転送速度を出しています(図3)。さらに、エミュレーションメモリ(オプション)を利用した場合、10Mバイト/秒以上のデータ転送速度を得ることができます。

図3 パソコンとターゲットの接続速度

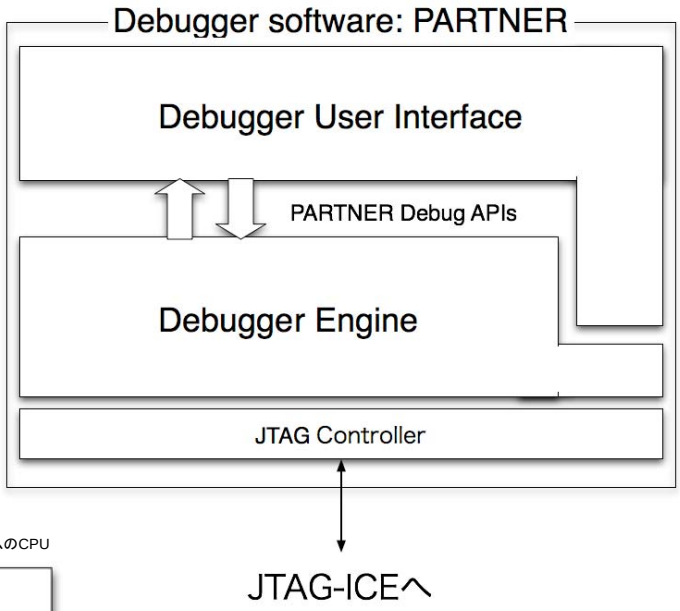


2-3 JTAG-ICEの実装例

JTAG-ICEの実装例として弊社のPARTNERデバッガについて説明します。図4に示すようにJTAG-ICEを構成する部分としては以下の構成要素があります。

- デバッガユーザインターフェース
- デバッガエンジン（Debug API)
- シンボル管理やソースコード管理、C++などの高級言語サポート機能
- 実行制御やメモリアクセスなどのターゲット制御

図4 JTAG-ICEの実装例：PARTNERデバッガの構造



Debug API

デバッガエンジンの各部分をDLL化することにより、PARTNERだけではなく、PARTNER以外の他のアプリケーションからもデバッガの機能を利用できるようになります。

PARNERデバッガ自身もこのDeug APIを使ってデバッグ環境を構築しています。このようにデバッガエンジンの各部分をDLL化することにより、後述する、マルチコアCPUのデバッグ、Linuxのデバッグを可能にしています。

API化されているため、PARTNER デバッガ以外のデバッガにも比較的簡単に接続でき、統合開発環境Eclipseから、このAPIを使用して接続させています。

また、API化によりデバッガ以外の目的にも簡単に応用可能で、実際

に製品テスト部門のテスト用プログラムとリンクして使用されていたりします（Virtual Link参照）。

3. マルチコアCPUのデバッグ

3-1　マルチコアCPUの登場

最近の半導体プロセスの向上により、より多くのロジックを１チップに実装できるようなってきました。また、より高いパフォーマンスと省電力化が重要になっています。そのような背景のもと、マルチコアCPUの開発が進められるようになりました。パーソナルコンピュータの技術を見てもマルチコアの利用が盛んになってきています。この流れは必ず組み込みソフトウェア開発にも発生すると思われます。事実、日本の半導体で既にマルチコアを利用したCPUを出荷しているところもあります。

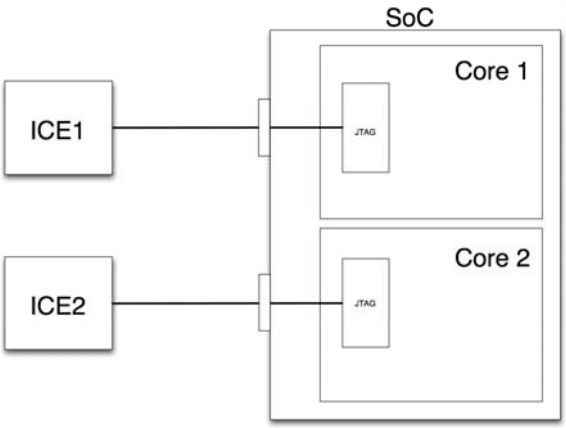
しかし、マルチコアをどうやってデバッグするのでしょうか？1コアに1台のICEをつなげるなどの非効率的なアプローチは現実的ではなく、先端を走るJTAG-ICEはマルチコア対応に取り組んでいます。弊社でもすでにマルチコア対応PARTNERの出荷を開始しています。ここではマルチコア対応をする際の問題点と解決方法について考えてみます。

3-2　マルチコアCPUデバッグにおけるJTAG-ICEの接続方法(ツール編)

マルチコアCPUの場合、JTAG-ICEの接続方法として考えられる構成は図5の3つのパターンです。

パターン1（図5a）は、単純に各コアのJTAGポートをコア毎に個々に持つ方法で、既存のツールとの接続性や互換性は十分です。しかし、この方法では多くのLSI端子が必要です。一般的なSoC（System On Chip）においてLSIの端子は貴重なので、（他の信号とマルチプレックスするとしても）これが欠点になります。また、必然的に各コアのJTAGポートにそれぞれのJTAG-ICEを接続することになります。パターン1は、デバッガソフトウェアの構築が容易です。各コアの独立性が高い場合は問題ありません。しかし、コア間の連携が必要になると、各コアを制御する各々のデバッガ間の通信が必要になり、ICE制御のリアルタイム性に問題が発生したり、機能上の制限が発生しやすくなります。

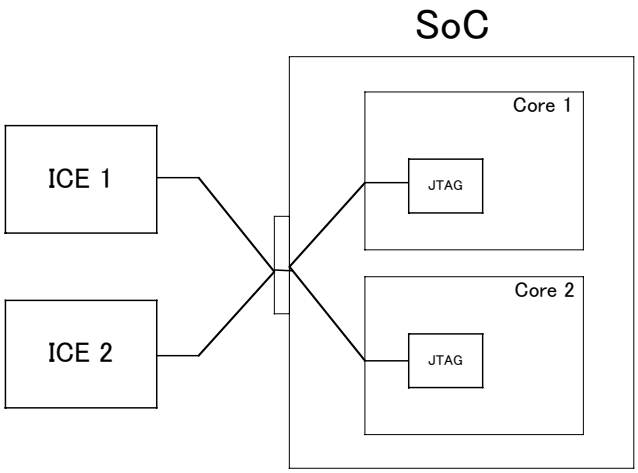
図5a　各コア毎にJTAGポートを持つ方法



パターン2(図5b)は、各コアがJTAGポートを共有し、１組のJTAG端子に複数のJTAG-ICEを接続します。LSI内部でのJTAGポートの共有方法（カスケード/パラレルスイッチ）については「3-3 JTAG接続の方法(ターゲット編)」で説明します。１つのJTAGポートには、各コアに対応する複数のJTAG-ICEをパラレルに接続します。各デバッガはそれぞれのコアへのアクセスを排他的に行うよう

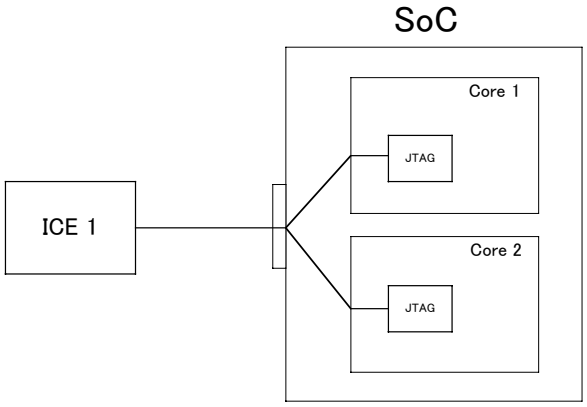
な仕組みを設けるだけで、パターン１と同様の動作や機能を実現できます。ただし、排他制御を行うレイヤがかなり上位層なので、各コア間での連携動作やデバッガ動作においてレスポンスが犠牲になることが予想されます。つまり、パターン２はパターン１に比べて、LSIの端子数の削減にはなりますが、それ以外の特徴はそのままパターン１と同じです。

図5b各コアが1つのJTAGポートを共有し複数のJTAG-ICEで排他制御



パターン3(図5c)は、各コアから1組のJTAG端子で出力されるJTAGポートに1つJTAGツールだけを接続します。この接続形態は、外見上は単一コアのCPUにおけるJTAGツールの接続と同じです。パターン3は、ハードウェアリソース（JTAG-ICEとJTAGポート）を完全に共有化し、ソフトウェアの構造化でマルチコアCPUに対応する方法です。このパターンを実現するには、うまく切り分けられたソフトウェア構造(DLL構造)とそれに対応できるハードウェアJTAG-ICE(PARTNER-Jetなど)が必要です。

図5c各コアが1つのJTAGポートを共有し1つのJTAG-ICEで制御



DLL構造

上記のパターン3において、パソコン上で動作する上位層のソフトウェアとJTAG-ICE内で動作する下位層ファームウェアを以下の動作環境で使用すると、CPUのコア間の連携動作において、より緻密な制御をハイレスポンスで行うことが可能になります。

- JTAG- ICE内で複数のコアに対応するJTAGコントローラを動作できる容量がある
- JTAG- ICE内のコントローラがハイパフォーマンスである（例えばPARTNER-Jetの場合、 SH4 240MHz/SDRAM 64MBを利用)
- プログラマブルなJTAG制御回路を搭載している(FPGA)

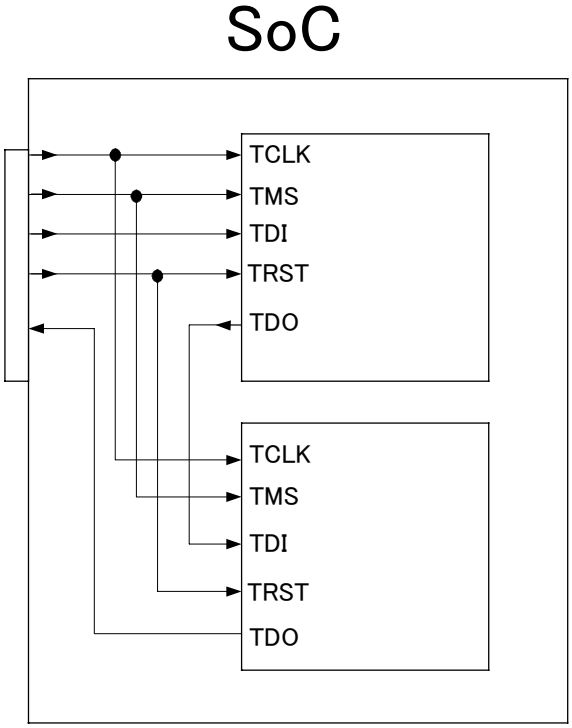
3-3 JTAG接続の方法(ターゲット編)

JTAGポートを共有するには、各コアでのJTAGポートを１つにまとめる必要があります。その方法には「カスケード接続」と「パラレル接続」の2通りの方法があります。

3-3-1 カスケード接続

図6は、PCB（Printed Circuit Board）上でのJTAGデバイスの接続に使用される一般的な接続方法です。この方法は論理的なデバイス数等の制限は無く、容易に複数のJTAGデバイスを接続することができます。SoCでのマルチコアデバッグでもこの接続方法を採用する場合は多くありますが、JTAGをデバッガポートとして使用する場合、この方法は最適とは言えません。カスケード接続は、すべてのデバイスを任意のタイミングで同時にアクセスすることができます。しかし、この方法では、接続されるコア数(デバイス)が多いほど、JTAGレジスタ(IR、DR)のアクセスに必要なJTAGクロック数(TCK)が増え、パフォーマンスの低下につながります。

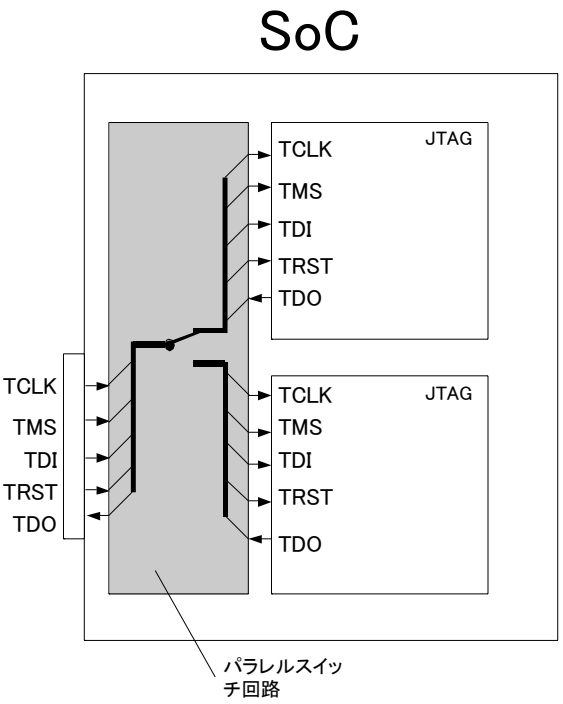
図6 マルチコアJTAG接続 - カスケード接続



3-3-2 パラレル接続

図7は、JTAGポートをパラレルに接続し、その接続先(コア)をスイッチ制御で切り替える接続方法です。これは、切り替えのための追加ロジック(ハードウェアリソース)を必要とします。この方法では、同一のコアにアクセスしている限り、JTAGレジスタ(IR、DR)のアクセスに追加のクロックは必要ありません。シングルコアと同じクロック数でのJTAG操作を行えます。デバッガが要求するJTAGポートへのアクセスは、ある時点で見ると特定の１つのコアに対する操作のみが要求されることがほとんどです。もちろんすべてのコアに対する同時アクセスにも対応する必要はありますが、一般的に、こちらの接続形態はカスケード接続に比べて効率的なオペレーションを実現できます。

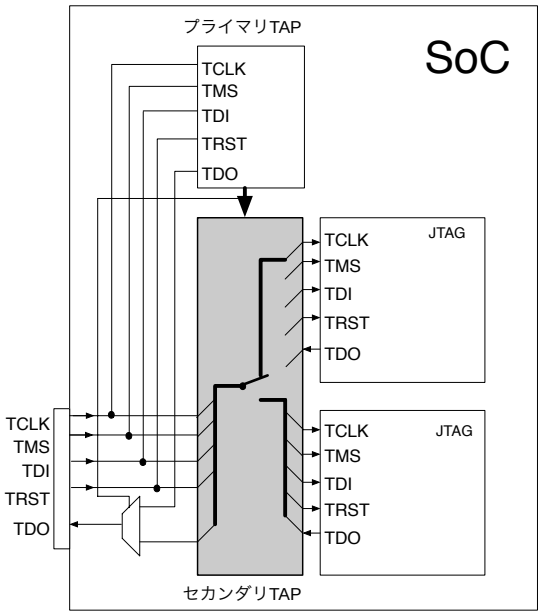
図7 マルチコアJTAG接続 - スイッチ接続



スイッチ制御の追加ロジックですが、図8に示すように各コア内のTAPコントローラ(セカンダリTAP)とは別に、チップ内にもう１つのTAPコントローラ(プライマリTAP)を実装します。プライマリTAPでは、スイッチ切り替えを指示するレジスタ(CORE_SEL)やマルチコアで必要となる追加のレジスタを実装し、マルチコアでの制御を行います。セカンダリTAPはCORE_SELレジスタで選択されたものだけが動作し、それ以外のTAPは動作させないようにします。たとえば、CORE_SELレジスタで指定されたコアのみTAPのステート遷移を発生させ、それ以外のセカンダリTAPは、Idleステートで停止させます。

弊社では、図8の制御方法を推奨し、すでに実チップで使用されています。

図8 JTAGパラレルスイッチ接続



3-4 マルチコアのデバッグに付随する必要な機能

マルチコアCPUのデバッグでは、JTAG-ICEとターゲットの効率的な接続方法におけるターゲット自身の対応（図8）の他にも、JTAG-ICE側およびマルチコアCPU側でデバッグに必要な仕組みを各種用意する必要があります。そうすることが最終的にはデバッグの効率を上げることにつながります。

同期実行/ブレーク

ユーザプログラムの実行およびブレークを各コアで独立に行うことや、同期して行うことができる必要があります。コア間の同期実行および同期ブレークの実現手法として特別なハードウェア機構の追加なしで、ツール側のソフトウェア制御で行う方法と、SoC内に同期機構をサポートするハードウェアロジックを追加して行う方法があります。

ソフトウェア同期では、コア間の

時間差はツールのソフトウェアやファームウェアに依存するため厳密な同期処理はできません。

ハードウェア同期は、SoC内に同期機構を組み込む方法でコアのクロックレベルでの同期が可能となります。この機能は基本的に各コアの状態(DBGACK)信号と各コアへのブレーク要求(DBGREQ)信号を相互に接続することで実現できますが、この部分の仕様をCPUおよびツールで策定する必要があり、異種類コアでの接続性の問題などを考慮する必要があります。ARMに関してはマルチコアデバッグのための仕様が策定されています。

弊社では、MIPS系CPUでハードウェア同期に対応したコアクロックレベルでの同期実行の実績があります。

バス状態のトレース

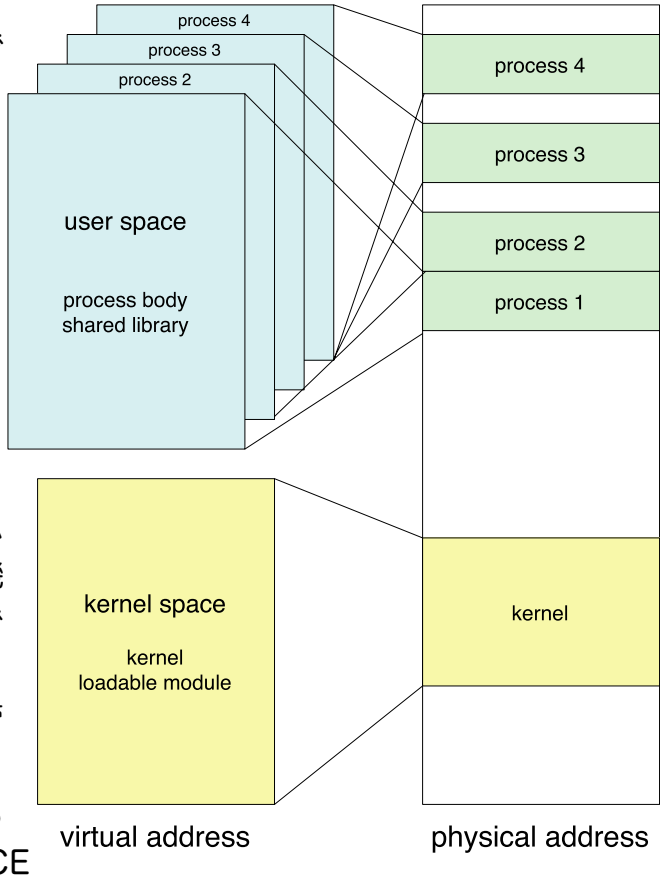
複数コアで同一のメモリ空間を共用する場合、各コアのリアルタイムトレース以外に、メモリバスのリアルタイムトレース機能やプロファイル機能がJTAG-ICEにあると便利です。
マルチコアにおけるリソース管理

メモリバスが共有されている場合、バス上の資源もJTAG-ICEから操作する場合に排他制御する必要があります。たとえば、Flashメモリやエミュレーションメモリは、すべてのコアからアクセスされる可能性を常に考慮した制御機構を必要とします。

4. 組み込みLinuxのデバッグ

組み込みLinuxのデバッグは困難です。従来の手法が通用しないのが主な理由です。例えば、ダイナミックローディングなどの実行時にアドレス解決される実行モジュールも存在し、従来採用されていたμITRONのモデルである、単一プロセス、仮想空間なしのルールが通用しません。図9にLinuxにおけるメモリ構造を示します。仮想化され多重化されているのが分かって頂けるでしょう。

図9 Linuxにおけるメモリ構造



4-1 組み込みLinuxのデバッグの難しさ - メッセージ出力関数を使うデバッグでは不十分

もっとも単純なLinuxデバッグ手法（おそらく実際に経験された方も多

いでしょう）は、printf()、syslog()、printk()などのメッセージ出力関数を利用するデバッグです。printf()は通常のアプリケーションデバッグで利用でき、またprintk()では割り込みがマスクされるためクリティカルセクションや割り込みハンドラの内部でも利用できます。これらのメッセージ出力関数を用いてステータスをコンソールに出力します。この方法はμITRONなどのデバッグでも用いられてきた手法です。

しかし、print関数系を実行コードに手動で埋め込むのは簡単ですが、状況を確認するだけであり、デバッガを利用した場合にくらべて効率的とはいえません。またコードサイズ、データサイズも増え、最終的なメモリサイズに収まらないなどの弊害もあります。なによりマルチプロセス、マルチスレッドでメッセージ出力を行うと、さまざまな箇所からメッセージが出力され、訳が分からなくなるという問題もあります。量子力学の世界ではありませんが、観測を行うと障害の現象が再現できないなどの問題も多いものです。

ソフトウェアデバッガgdbとkgdbの限界

Linuxのクロス開発環境では、上記print系に加えてgdbを利用してデバッグを行うこともありますが、gdbのコマンドは複雑で、最近のデバッグ環境と比較するとそのレベルのギャップは大きいものです。最近は“ddd”や“insight”と呼ばれるgdbのGUIラッパが用意され、大いにユーザインターフェースに改善が施されていますが、いずれの方法でも多数のスレッドが生成されるようなアプリケーションのデバッグは困難です。また、gdbはアプリケーションのデバッグしかサポートしておらず、カーネルをデバッグできるkgdbというものもありますが、カーネル、デバイスドライバ、アプリケーションを一貫してデバッグできる仕組みは残念ながら現在のLinuxには実装されていません。組み込みソフトウェア開発においてはアプリケーションを開発しながら、デバイスドライバのデバッグを行うなどは必須要項のためgdbによるデバッグでは不十分です。

たとえgdb、kgdbで問題箇所がデバッグ可能であっても、従来から用いられてきたICEの機能が提供されるわけではありません。例えば、ヒストリ（リアルタイムトレース）表示はICEでは有効な機能として提供されていますが、ソフトウェアデバッガであるgdbにはそのような機能がありません。

LinuxのデバッグAPIの限界

gdbの実装にはptraceといわれるLinuxのローレベルのデバッグAPIが利用されています。ptraceでは実行しているアプリケーションにgdbなどのデバッガを取り付け（attach）し、メモリのリードライト、レジスタの参照などを行うことを可能にします。しかしデバッグ対象となるのはあくまでもアプリケーションプロセスであり、カーネルやドライバをデバッグすることができません。（**コラム「ptrace()によるデバッグとの比較」参照**）

従来のハードウェアデバッガの限界

従来の組み込みソフトウェアのデバッグに利用されていたICE、ROMエミュレーションデバッガ、JTAGデバッガが利用できない理由に、アドレスの動的解決、仮想メモリ空間管理、カーネルとアプリケーションメモリ空間の分離があります。μITRONのようにアプリケーションとカーネルが静的にリンクされている場合は、メモリにカーネル＋アプリケーションをロードし、それに対応するデバッグ情報を基にデバッグすることができますが、仮想メモリ空間で動的に実行モジュールのアドレスが変わるようになると、デバッグ情報と実際の動作アドレスとの対応が取れなくなりつじつまが合わなくなります。これがLinuxで、今までのツールが利用できない大きな理由です。

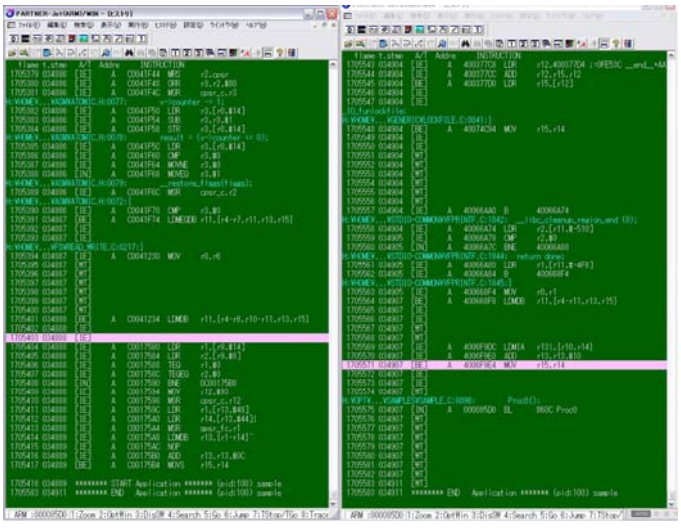
4-2 Linuxデバッグを可能にするJTAG-ICE

弊社では、PARTNERデバッガのLinux対応にあたり、従来JTAG-ICEが提供できているフルの機能をLinux対応JTAG-ICEで提供いたしました。カーネルにパッチを当てることなくデバッグをすることも可能ですが、デバッグ作業を容易にするため、10カ所程度に小さなカーネルパッチを行うことを弊社では勧めています。パッチを施す場所は例えばプロセスが終了する際に必ず通過する部分などの、プロセスの生成・プロセススイッチ・プロセスの終了部分などです。当てるパッチはどれもソース上で1行もしくは2行くらいの小さな変更にとどまっています。またメニュー操作により簡単にパッチを有効にしたり、無効にしたりすることが可能です。

4-2-1 Linuxにおけるリアルタイムトレース

Linuxではプロセススイッチングを行いながらソフトウェアを実行し、割り込みも入ります。リアルタイムトレースでは論理アドレス情報がパケットデータとして出力されますので、多重空間になっているどのアプリケーションのトレース情報なのかを判定するには特殊な技法が必要となります。これらのことを考慮せずに、生のリアルタイムトレース表示を行うと解読不可能な状態のままです。PARTNERデバッガではトレースデータを解析する際に、それがカーネルコードなのか、どのアプリケーションコードなのかを解析し、それぞれを分離して表示することが可能です（図10）。

図10 カーネルトレースとアプリケーショントレースの分離



4-2-2 Linuxの実行モデル

Linuxに関するソフトウェアは、以

下の実行モデルに分類できます。それぞれの特徴を図11に示します。

図11 Linuxにおける実行モデル

プログラム種別	空間	アドレス	ページング	デバッグ
ブートローダ	非MMU空間	固定番地	無し	通常の組み込みデバッグに同じ
Linuxカーネル	MMU上のカーネル空間	固定番地	無し	通常の組み込みデバッグに同じ
ローダブルモジュール	MMU上のカーネル空間	リロケータブル	デマンドページング	リロケーションとページングへの対応が必要
共有ライブラリ	MMU上の多重仮想空間	リロケータブル	デマンドページング	論理多重空間、リロケーション、ページングへの対応が必要
アプリケーション	MMU上の多重仮想空間	固定番地	デマンドページング	論理多重空間、ページングへの対応が必要

従来のICEで容易にデバッグできるのは、ブートローダとLinux カーネルです。これらの実行モデルはコンパイル・リンク時に物理=論理アドレスが固定されているためにICEおよびデバッガでの対応が容易です。ブートローダ以外はMMU（Memory Management Unit）がONの状態で動作しますが、MMU以外にアドレス方式（静的アドレスか動的アドレスか）とメモリページング（デマンドページング）がICE対応に大きく関係します。

デマンドページングはUnixライクなOSの特徴でもあるメモリ空間の効率的な利用方法です。デマンドページングでは物理メモリにデータが乗るのは、そのデータを実際に参照した時点です。これは、データだけではなく、実行コード（テキストセクション）でも同様です。プログラムが走り始めた瞬間に必要な最小単位のプログラムコードが物理メモリ（RAM）上に展開されます。

デバッガでデマンドページング対応が難しい点は、実行していないプログラム、つまりまだメモリ上に展開されてない可能性があるプログラムに対してブレークポイントを設定できる仕組みを用意する必要があるためです。それを実現するために、PARTNERデバッガでは独自のページフィックステクノロジを用いて、まだロードされていないコードを強制的にロードする機能があります。ページフィックステクノロジは、JTAG-ICEが小さなプログラムをアプリケーション空間に送り込み、そのプログラムを実行させ、そのプログラムがメモリを参照し、ページフォルトを発生させ、Linux カーネルが必要とされているプログラムをメモリに載せるという仕組みです。

4-2-3 ローダブルモジュールでのアドレス解決とページングへの対応

ローダブルモジュールは、“insmod”コマンドでカーネルに後付でロードされリンクされるモジュール構造です。PARTNERデバッガはこのデバッグにも対応しています。ローダブルモジュールはカーネル空間にダイナミックにロードされるモジュールのため、実際にカーネル空間にロードされるまで実行アドレスが不明です。PARTNERデバッガでは、ローダブルモジュールの先頭に“break”コマンドとマジックコードを挿入することで、ファイルシステムからローダブルモジュールがloadされる瞬間をとらえ、デバッグ情報のオフセットを行い、デバッグ可能とします。この仕組みの準備はとても簡単で、ローダブルモジュールがロードされて最初に呼び出される関数、module_init()関数が存在するソースの先頭に

```
#define _KMC_MODULE_DEBUG
```

を挿入するだけです。

このようにPARTNERデバッガは様々なLinuxの仕組みを理解した上で動作していますが、それを可能にするのがターゲットとデバッガ間でのレイテンシの低さです。ダイナミックリンクなどを理解しながら動作するには様々なOS情報を取得する必要があります。そのためにはJTAG-ICEには広い帯域と高速な応答性が求められます。

4-2-4 カーネルモードとアプリケーションモードの違い

PARTNERデバッガでLinuxをデバッグする場合、モードが2つ有ります。１つはカーネルモードでもう一つはアプリケーションモードです。この２つの最も大きな違いはbreakに関する点です。カーネルモードでbreakすると、システム全体が停止します（つまりCPUが停止してしまいます）。一方のアップリケーションモードでは特定のアプリケーションでbreakしても他のプロセス、カーネルは停止しません（図12）。

図12 カーネルモードとアプリケーションモードの違い

	マルチスレッドデバッグの方式	アプリケーションでブレーク発生時の挙動
カーネルモード	一つのデバッガウィンドウで、一つのスレッドをデバッグ	CPUが停止
	一つのデバッガウィンドウで、複数のスレッドをデバッグ	CPUが停止
アプリモード	一つのデバッガウィンドウで、一つのスレッドをデバッグ	対象スレッドだけが停止（カーネルや他のアプリは実行）
	一つのデバッガウィンドウで、複数のスレッドをデバッグ	対象スレッドだけが停止（カーネルや他のアプリは実行）

これは例えば1つのプロセスでデータストリーミングを行いながら、グラフィカルユーザインターフェース部分をデバッグする際に非常に有用です。これによりバックグランド実行しているメディア情報の受信（データストリーミング）を停止することなくユーザインターフェース部分のデバッグすることが実現可能です。

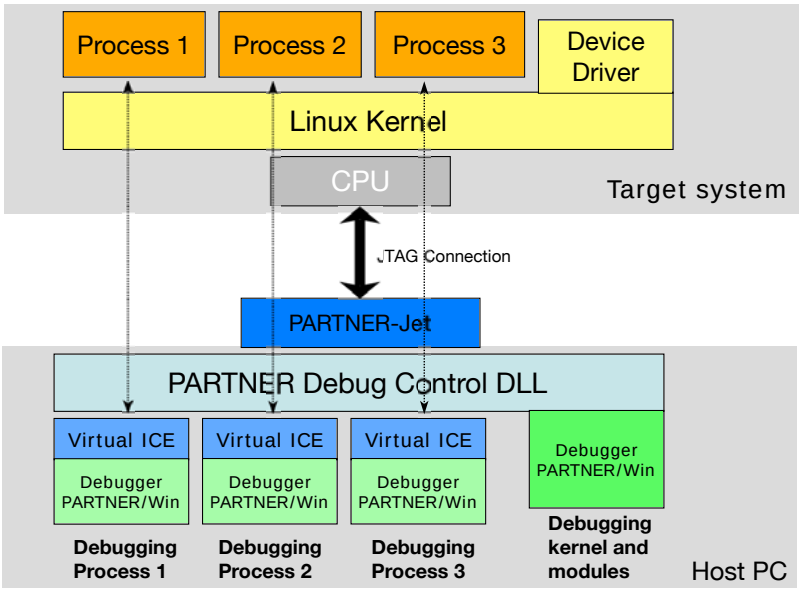
アイデアは簡単そうですが実装は意外と難しく、止めたいアプリケーションにプロセスが切り替わるとJTAG-ICEがいったん制御を奪い、直ぐにプロセススイッチを発生させ、停止していないプロセス、カーネルが一切止まっていないように見せかけている訳です。

4-2-5 仮想ICE技術

3で述べたように、PARNERはデバッグエンジンをDLL化し、Debug APIを用意しています。デバッグDLLは複数のクライアントをサポートする仕組みを用意しているため、一つのJTAGポートを利用して、Linux用に

複数プロセスごとにデバッガを起動したり、またマルチコアの際に一台のJTAG-ICEで複数のコアのデバッグを可能としています。これを弊社ではICEの仮想化と呼んでいます（図13）。

図13 ICEの仮想化技術（複数のデバッガを起動可能）



4-2-6 LinuxとJTAG-ICEの共存

Linuxを用いた組み込みシステムを効率よくデバッグするには、Linuxに対応したデバッガツール（JTAG-ICE）を使用するだけではなく、OSとCPUの特性をある程度理解する必要もあります。

ハードウェアによるウォッチポイント

全てのCPUに適応されるかどうか分かりませんが、Linuxでプロセスにハードウェアによるウォッチポイントを設定する際には少し注意が必要です（プロセスにハードウェアブレークを設定できるJTAG-ICEも少ないです）。

過去の組み込みシステムの開発では、開発者は0番地にメモリライトでハードウェアブレークポイントを設定し、0番地書き込みなどを検出するデバッグを行ってきました。それと同じ事をLinuxのプロセスでも行おうとして、0番地にメモリライトでハートウェアブレークポイントを設定し、そのようなシーンが発生するプログラムを動作させても、全くハードウェアブレークポイントにヒットしません。

なぜなら、Linuxのプロセスは、CPUのMMU（Memory Management Unit）上の仮想空間で動作し、0番地付近は最初からページフォルトが発生するように設定されています。一般的にCPUはその番地へのメモリライトのハードウェアブレークポイントが発生する前に、ページフォルトが発生するようになっています。そのため、ICEで設定したハードウェアブレークポイントではヒットせず、カーネルの例外ハンドラで処理されます。

この特性を知らずに、Linux採用以前と同じ感覚でデバッグしていると、貴重な時間を多く失ってしまう事になります。

このような場合には、ICEでカーネルの例外ハンドラ（ARMならdo_user_fault()　とdo_kernel_fault()）に常にブレークポイントを設定してデバッグすると良いでしょう。そうする事で貴重な資源であるウォッチポイントはより有効な変数に設定する事ができ、かつ0番地など不正なエリアへのアクセス

も発生した時に例外ハンドラでブレークして調べる事ができます。例外ハンドラはブレークを設定しなくてもレジスタダンプなどが出るので最低限の情報は後からでも分かります。しかし、不正なアクセスが発生した時にCPUを停止できると、その時の状態が完全に調査できるので、より効率よく問題を解決できるでしょう。

MMUのアクセス違反を監視し、かつ必要に応じてウォッチポイントが設定できるので、JTAG-ICEを用いた組み込みLinuxのデバッグは、そういう面ではLinux採用以前より強力なデバッグ環境を構築しているといえます。

IPCによる通信のデバッグ

ＩＰＣ（Ｉｎｔｅｒ Ｐｒｏｃｅｓｓ Communication）による共有メモリを用いた場合のデバッグにおいて、「メモリに誰が何を書いたか」という事をLinux以前のデバッグ方法で調査するのは困難です。IPC共有メモリの場合、同じ物理メモリであっても、使っているプロセスによって論理アドレスが異なる場合が多いのです（異なる事が普通です）。誤解されやすい事ですが、JTAG ICEでCPUのデバッグユニットを用いてハードウェアブレークポイントを設定している場合、そのアドレスは多くの場合は物理アドレスでなく、論理アドレスです。したがって、二つのプロセスが利用するIPC共有メモリへの書き込みタイミングをハードウェアブレークポイントだけで検出する事は、その共有メモリの物理論理アドレスが二つのプロセスで異なるために非常に困難です。

このような状況をデバッグするには、カーネルのIPCの実装に少し手を入れて、デバッグ用の監視機能を作成し、そこをICEで捕まえるのが良いと思われます。工夫の仕方によっては、これも今までよりも強力なデバッグ環境が構築できる可能性があります。
Linuxがオープンソースであることよる恩恵

ICEとカーネルの連携でデバッグ環境が強力になると説明しましたが、これはLinuxカーネルがオープンソースであり、利用者は常にカーネルを変更する権利を持っているからこそ、このような機能が実現できるのです。デバッグのためにカーネルを改造することに対して否定的な方もいらっしゃるかもしれません。その改造を製品化するカーネルに入れるのではなく、デバッグのためだけに少しの改造を入れて、そして元に戻すという事でも十分に効果を発揮する（問題解決の時間を短縮する）可能性もありますので、是非試してみる事をお勧めします。

パソコンやサーバのLinuxではそのような開発は難しいかもしれませんが、組み込みＬｉｎｕｘでは弊社のPARTNER-Jetなどを用いればカーネルをデバッグする事は非常に容易で（カーネルのデバッグは、今までの組み込みシステムのデバッグとほぼ同じといって差し支えない）、カーネルを

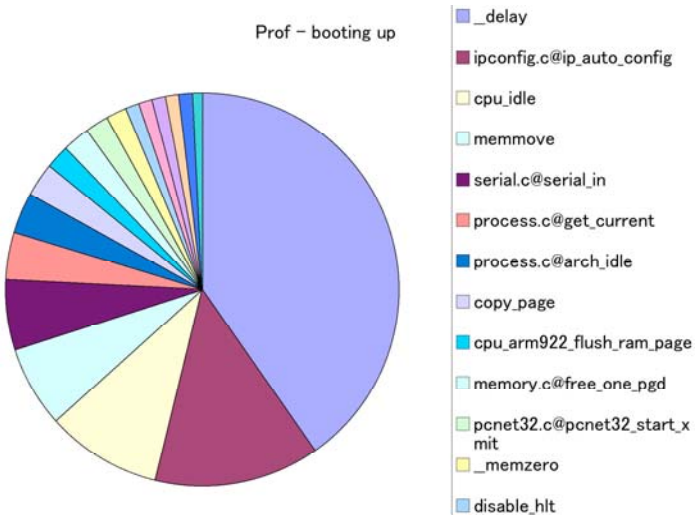
変更する時も非常に扱いやすいです。このような環境があるからこそ、ICEとカーネルを組み合わせて利用すると、複雑な問題を今まで以上に効率よく解決する事が可能になります。

弊社のPARTNER-JetのLinux対応機能も、Linuxがオープンソースであるがゆえに多くの機能を実装できました。他のプロセスのメモリを見るために、プロセス管理の構造体やファイル管理の構造体、またメモリ管理の構造体など、実行中のLinuxカーネルの動作について非常に細かく把握してデバッグを行いました。私たちも開発のためにLinuxに少し手を入れて調査し、そして機能を作り上げて開発を行いました。このような開発がサードパーティでできる事はLinuxがオープンソースで無かったら不可能だったでしょう。

5. パフォーマンス解析技術

現在、弊社で開発中の技術に１００μ秒ごとのＰＣ（Program Counter）サンプリングするパフォーマンス解析技術があります。通常のパフォーマンス解析の場合、ターゲットのリソースをかなり消費します。例えばバッファ用のメモリの消費や実行速度の低下などが起こるわけです。しかし弊社が開発しているプロファイルテクノロジーでは100μ秒ごとにPC値をサンプリングしますが、その速度低下は1%未満（ARM9の場合）であり、バッファにはJTAG-ICEのハードウェアに搭載されているメモリを使用するので、ターゲットの実行への影響がほとんどありません。100μ秒ではかなりの命令が実行されますが、マクロ的に見ると、ボトルネックがどの関数でどれくらい発生しているか十分に分かります。これも既に述べたリアルタイムトレースと同様にカーネルコードか、どのアプリケーションかを判別可能な形で情報を表示するため、開発者は容易に時間を消費している部分を特定することが可能です。例としてARM9のLinuxをNFSでbootする際のサンプルプロファイルを図14に表示します。このグラフからCPUが待ちに入っている状態（I/O処理によるwaitなど:_delayとcpu_idle）がbootプロセスの約50%を消費していることが簡単に分かります。

図14 Linux(ARM9）boot up時のパフォーマンス解析



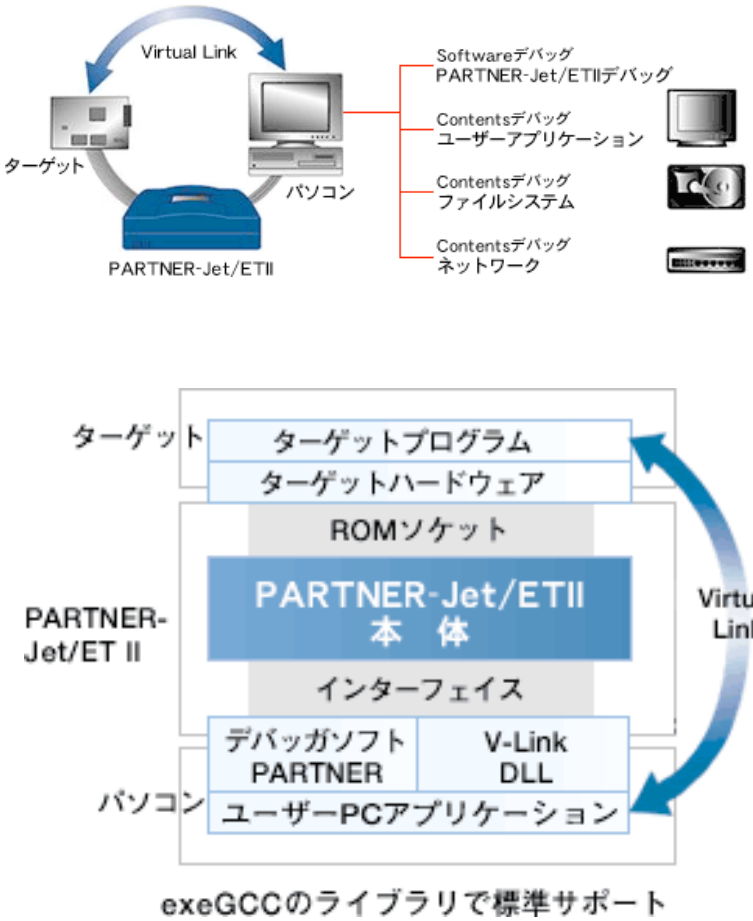
6. JTAGデバッガの応用機能 - Virtual Link

JTAGは、本来のピン間接続テストでの利用を越えて、オンチップデバッグに利用されるようになりました。それをさらに推し進めて汎用通信ポートにしたのが、弊社独自のVirtual Linkテクノロジです。

Virtual Linkはターゲット上のシリアルポートやEthernetを仮想化する技術で、ターゲットとデバッガが動作しているパソコンとの間に仮想的な通信ポートをJTAG上に構築します。

弊社のexeGCCにはVirtual Linkをサポートしたライブラリが付属されています。これは通常のC言語ライブラリのファイルアクセス関数（fopen、fread, fwriteなど）がVirtual Link機能を利用して、デバッガが動作しているパソコンのディスクにアクセスできます（図15）。

図15 Virtual Link技術



弊社ではVirtual Linkを利用してLinux用に仮想シリアルドライバと仮想Ethernetドライバを開発し、提供しています。これにより、ターゲットボードとPARTNER-Jet間をJTAGケーブルで接続し、PARTNER-Jetとパソコンを USBケーブルで接続した状態で、ターゲット上で適切に設定されたLinuxカーネルを立ち上げると、LinuxのttyはPARTNERデバッガが起動しているパソコンのCOMポートに接続され、またEthernetは同様にパソコンに接続された形となります。このような仮想化を行うことで、ターゲットにJTAG-ICEだけを接続して開発できます。

このような仮想化技術を用いることで、JTAG-ICEをデバッガとしてのみ利用するのではなく、仮想的な

ファイルシステムとして利用したり、品質検査に利用することも可能になり、JTAG-ICEの応用範囲が広がります。

7. Eclipseと組み込みソフトウェア開発

EclipseとはEclipse.org（http://www.eclipse.org）によりオープンソースベースで開発が進んでいる統合開発プラットフォームです。従来の統合開発環境（IDE）に相当しますが、Eclipseはplug-inによって拡張できる範囲が広く、様々な用途で利用されているため、最近は統合開発プラットフォームと表現しています。

EclipseはIBMがJavaの開発環境として自社で開発していたものを、オープンソース化したもので、サポート言語の最有力候補はJavaです。しかしCDT（C Development Toolkit）による拡張により、C/C++ 言語での開発も可能です。

7-1 Eclipseが組み込み開発と同じ方向と別の方向

Eclipseは統合開発環境であり、これが組み込みで利用できることには非常に意味があります。組み込みソフトウェア開発では標準といえる開発環境が無く、プロジェクトごとで環境が異なるのは極当たり前とされてきました。Eclipseはそのような状態に変化をもたらすものであり、今後の組み込みソフトウェア開発での活躍が期待されています。一方で、JTAG-ICEのサポートはもちろんのこと、組み込みソフトウェアではごく当たり前とされている機能が不足しているのが正直なところです。例えば、ICEでは必ずサポートされているトレース結果の表示のためのプラグインなどは、現在のEclipseには用意されていません。しかしながら、これまで組み込みソフトウェア開発でこれと言った統合開発環境がなかったところに、Eclipseが出てきたことには非常に大きな意味があります。

7-2 KMCのEclipseへの取り組み

弊社では組み込み分野でも次世代標準統合開発環境はEclipseであると判断して早い段階から開発を行ってきました。

京都マイクロコンピュータは2006年1月にEclipse FoundationのAdd-in Provider memberとして参加して日本の組み込み開発の手法を広く世界に知らしめるための活動を開始しました。

第一段階（リリース済み）では、図16aの様にJTAG-ICE をサポートするためのEclipse plug-inを開発して、exeGCCと合わせて提供することで、デバッグを含む開発環境はEclipse、コンパイラはexeGCC、JTAG-ICEのハードウェアはPARTNER-Jetシリーズという構成をサポートしました。ただこのバージョンでは前述したヒストリの表示などがEclipseでできないため、図16bのようにPARTNERデバッガも並行して立ち上げておくことを推奨しています。この構成であればEclipseでソースコードを書き、exeGCCでコンパイルし、Eclipse plug-inにてデバッグを行うことが可能です。また、機能的に不足なところがあれば、従来のPARTNERデバッガに切り替え、並行してデバッグ作業を

進めることができます。

2006年春に開発終了を目標にしている次のバージョンでは、従来のPARTNERデバッガが持っていた機能のほぼ全てを、Eclipseで利用できるようにする予定です。

従来の組み込みソフトウェア開発になれている人はPARTNERデバッガを利用し、また統合開発環境に慣れているユーザはEclipseを使うなどの使い分けが起こると思っています。このバージョンではPARTNERデバッガでは既にサポートしている各種OS(LinuxやT-Engineなど)の開発もサポートする予定になっています。

7-3 組み込み業界のEclipse への取り組み

Eclipseにはサブプロジェクトとして様々な機能拡張が行われていますが、2005年3月8日に“Device Software Development Platform”プロジェクト発足の発表が行われました。組み込みソフトウェアの開発にEclipseを使おうとする大きな流れであり、今後の成果が期待されています。

8. まとめ

以上、デバッグという観点から最近の組み込みソフトウェアの開発環境を考えてみました。複雑化する組み込みシステムへのマルチコアCPUの採用やLinuxが組み込みシステムのOSとして採用されることにより、デバッグ手法にも変革が必要とされています。より多くのソフトウェア開発が要求される中で、開発の効率化、特にデバッグの効率化は重要な話題となっています。また今後は、組み込みアプリケーションの規模が拡大し、パフォーマンス解析、仮想化されたJTAGポートを利用しての品質管理システムの構築など、さらにJTAG-ICEの応用範囲が広がるであろうと思われます。

コラム ptrace()によるデバッグとJTAG-ICEの比較

By 辻邦彦

ドライバの実装がデバッグに影響を与える場合

組み込みLinuxの場合、そのハードウェアに応じてデバイスドライバを新規に実装する事が多々あります。Linuxにはドライバの実装に関して”暗黙のルール”は存在しますが、完全な教科書や解説書は世の中に存在せず、従来のデバイスドライバを参考にしがちです。しかし、参考にしたドライバやその解釈によって、実は重要な機能であるのに、その機能の移植を見落とすという事もあるでしょう。特に、それが主にデバッグ時にしか使用されないような機能であると、ソースコードの実装からは読み取れない事が多いと思われます。

ドライバシステムコールの中で待ちに入ってスケジューラが呼ばれる事は多くあり、上記はその時の処理の一例です。ところが、ドライバによっては、上記の#1だけ行い、#2のチェックで#3の戻り値をERESTARTSYSに設定せずに、「シグナルを受信した」の意味であるEINTRを返す実装があります（#3がret = -EINTR; のパターン）。

これは「どちらの戻り値が正しいか？」という事ではなく、システムコールの呼び出し側と合わせて実装しないと正しく動作しないという問題です。すなわち、EINTRで戻る場合、呼び出し側はシグナルを受信して戻った事を意識して適切な処理をしなければなりません。ところが、組み込みデバイス向けに新規に作られるドライバとアプリケーションは用途限定（その機器向けに開発したので当たり前ですが）であるために、あまりこのあたりの考慮がされていません。つまり、ドライバはEINTRを戻しますが、システムコールを発行したプロセスはEINTRで返却されるとデバイスが異常だと判断して障害処理を

```
#1 interruptible_sleep_on(&info->delta_msr_wait);
            /* see if a signal did it */
#2          if (signal_pending(current)) {
#3                  ret = -ERESTARTSYS;
            }
```

#1 実行中のプロセスをwait状態にする。ただしシグナルを受信したらwait解除

#2 wait解除要因がシグナル受信か？

#3 シグナル受信でwait解除であれば、ERESTARTSYSを戻り値にしてセッション終了を戻り値にしてセッション終了

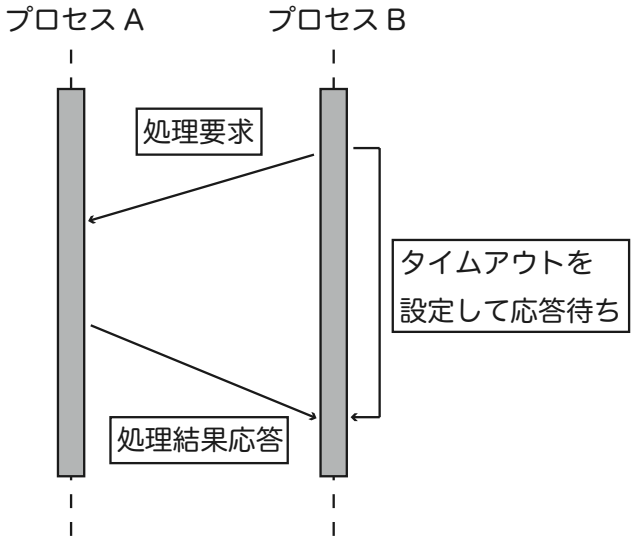
行うような実装もよくあります。通常時であればシグナルが発生しない（ように作っていればですが）かもしれず、それでも問題が無いかもしれません。しかし、ptrace()デバッグはシグナルを利用しており、例えば上記のドライバの待ち状態の時にデバッガ（gdbなど）からattachを行うと、その対象のプロセスにシグナルを送信し、上記のドライバの待ち状態がシグナルを受信したために解除されてしまいます。そのために、デバッガでattachした結果、デバイスドライバからEINTRが戻り、デバッグしようとしたプロセスにアタッチはできたが、意図しない障害処理が実行されてしまい、結果的に正しくデバッグができなくなるという事に陥ります。

一方、ドライバがwait中にシグナルを受信した時にERESTARTSYSを戻り値に設定した場合、Linuxはシステムコール処理からユーザ処理へ戻る時に再度自動的にシステムコールを再実行させます。デバッガからアタッチを行うと、システムコールを発行した命令の所（ARMならSWI命令）でプロセスにアタッチが行われ、そしてデバッガから実行するとシステムコールを再実行します。これであれば不必要な障害処理も行われず、比較的意図した通りにデバッグしやすいです。しかし、ドライバの中の実装が、そのような実装に耐え得るかについては検討の必要があります（ERESTARTSYSはptrace()デバッグのためだけに存在するわけではありません）。

ここで問題なのは、体系的な教科書が無いLinuxにおいて、上記のような事を意識して開発をしなければならないのか？という事です。ptrace()システムコールによるデバッグは、デバイスドライバの実装方法によってデバッグが大きく影響され、結果的に正しくデバッグできないという限界があります。UNIXの上でptrace()を使ってデバッグしている時（gdbやdbxなど）に、同時にカーネルやデバイスドライバを開発してデバッグしているケースは稀なことではないでしょう。組み込みLinuxの開発における特殊性はまさにそこにあり、特にデバイスドライバと同時に開発を行う事が多いために、このような問題が発生しやすいのです。

ここで説明した問題は、PARTNER-Jetのプロセスのデバッグ機能を使えば、全く問題は発生しません。PARTNER-Jetにも実行中のプロセスにアタッチの機能はありますが、シグナル機能は全く利用していないので、アタッチをしてもデバイスドライバ内での動作が通常時と異なる事もなく、また呼び出したプロセスの動作が変わる事もありません。上記の例でいえば、EINTRが返却されてプロセスの障害処理も実行されないし、またシステムコールの再実行も行われません。PARTNER-Jetからwait状態のプロセスにアタッチを行うと、そのプロセスがwait解除された時にデバッグが可能になるだけです。

このような局面がある事を考えると、組み込みLinuxのデバッグでは、ptrace()だけでは十分とはいえず、JTAG-ICEが多くの場面で有効に活用できると思われます。

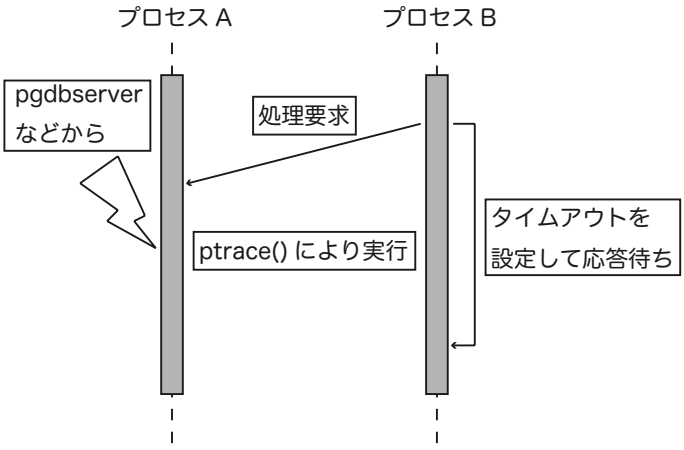


マルチプロセスのデバッグ

複数のプロセスが連携して処理を行う時にptrace()方式ではデバッグが難しい時があります。

上記は非常に一般的なパターンですが、プロセスAが要求された処理を実行している最中にptrace()でブレークさせたりした場合、プロセスBでタイムアウトが発生して、その後の処理が想定通りにデバッグできない事があります。もちろん、可能な限り同時にプロセスBもptrace()からブレークさせれば良いのですが、ptrace()を発行するgdbserverなどがプロセスとして動作しており、カーネルのスケジューラにより実行タイミングが決定されるため、完全にプロセスA/Bを同時に停止させる事は難しいです。

Linuxのプロセスという概念が採用される以前の組み込みシステムでも、上記のような事はタスク間通信などで行われてきました。しかし、デバッグについてはICEを用いる事が多く、タスク処理のどこでブレークが発生してもCPUが止まるので、他のタスクが実行されない事が保証されていました。これを前提として、別のタスクの処理完了待ちに新たにブレークポイントを設定したり、処理完了待ちのタスクの変数を参照したりするデバッグが行われていました。したがって、組み込み開発を以前からしていた開発者にとっては、ptrace()だけでは今まで行ってきたデバッグの方式が通用せず、開発に苦労する事になってしまいます。



PARTNER-Jetでは、ICEでプロセスAにおいてブレークすればCPUが停止し、そしてプロセスBにブレークを設定したりメモリを参照したりするデバッグも可能です。したがって、今までの組み込みシステムとほぼ同じような感覚でデバッグが可能であり、カーネル・ドライバ・アプリケーションのどこでブレークが発生しても

CPUが停止して、任意の箇所にブレークを設定してデバッグできます。

　またこれとは逆に、システム全体を見ると、アプリケーションでブレークが発生した時にCPUが停止すると困る場合があります。ptrace()でデバッグすると、これがICEとは異なる面でのメリットでしょう。PARTNER-Jetではそのような用途にも応えられるようVirtual ICEテクノロジを用いた「アプリケーションモード」を実装しています。これにより、ICEでデバッグしていても、デバッグ中のアプリケーション以外はデバッグに影響されないような感覚（そのプロセスでブレークしても、他のプロセスは動作している状態）でデバッグを行えます。

※ 仮想ICEテクノロジは厳密にはCPUは停止します。停止する時間もCPUにより異なり、CPUの仕様によっては仮想ICEテクノロジの利用が現実的で無い場合もあります。

KYOTO MICROCOMPUTER

戦略マーケティング部長　植田省司

2005年1月から現職、京都マイクロコンピュータの戦略マーケティング部長として国内のマーケティング及び海外の事業開発を担当する。

京都マイクロコンピュータに所属する前はTexas州Austinに本社があるMetrowerksの日本法人の代表取締役社長を務め、開発ツールCodeWarriorの日本での普及に尽力しCodeWarrior for PlayStaion2などの成功に導く。

京都マイクロコンピュータ株式会社

〒610-1104

京都市西京区大枝中山2-44

http://www.kmckk.co.jp/
jp-info@kmckk.co.jp
075-335-1050