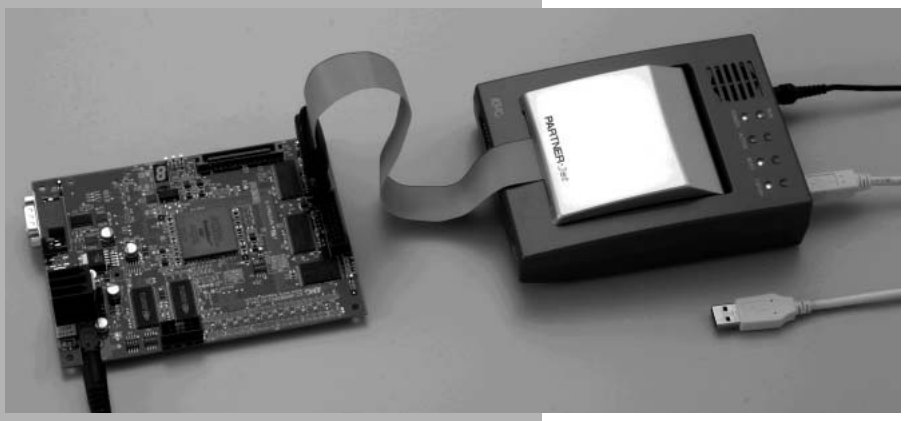


Linuxやマルチコア環境のデバッグを支える仮想化技術

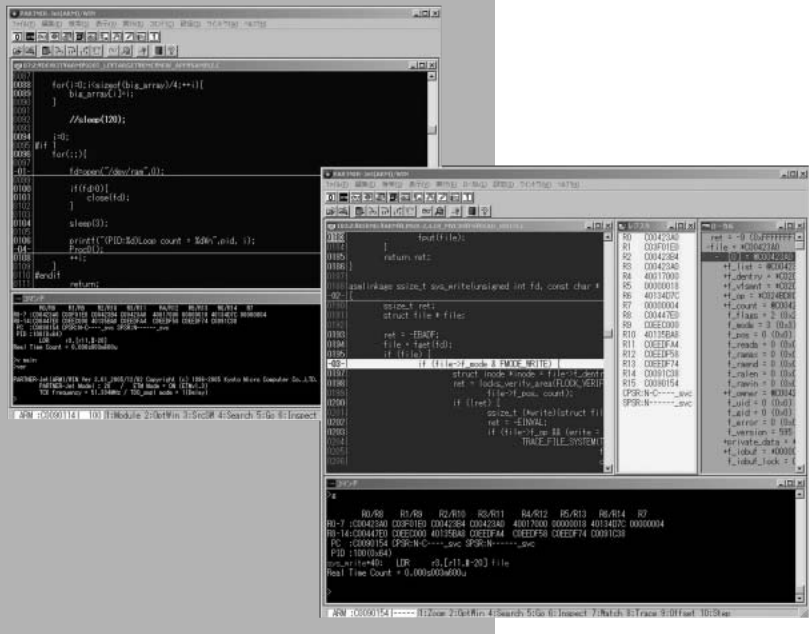
植田 省司
京都マイクロコンピュータ 戦略マーケティング部長

辻 邦彦
京都マイクロコンピュータ 東京オフィスゼネラルマネージャ

塚越 勲
京都マイクロコンピュータ 取締役開発部長

山本 彰一
京都マイクロコンピュータ 代表取締役社長

神田 良一
京都マイクロコンピュータ 開発部



プログラミングの効率やシステムの性能を高めることを狙ったソフトウエア実行環境の変化が，デバッグ環境をより複雑なものに変えようとしている。仮想記憶方式を採るOSやマルチコア構成のマイクロプロセサを組み込み機器に採用すると，従来のデバッグ環境では効率よく作業できないためである。こうした中，京都マイクロコンピュータは，Linuxとマルチコア型マイクロプロセサで実行する組み込みソフトウエアを想定したデバッガ・システムを開発した。OSが生成する複数のプロセスや，複数のCPUコアで並列に実行するOSおよびプロセスを1つのJTAG-ICEで同時にデバッグできるのが特徴である。このために「仮想化」の考え方を導入した。デバッガの仮想化技術や，それによってLinux環境やマルチコア環境のデバッグをどう実現しているのかについて解説してもらう。

（竹居 智久＝本誌）

† JTAG（Joint Test Action Group）＝LSIの検査方式であるバウンダリ・スキャン・テストの標準方式および，標準を定めた業界団体の名称。IEEE1149.1として標準化された。JTAGに対応するLSIはTAP（test access port）と呼ばれる5本の端子から成るインタフェースを備える。JTAGはLSIの検査だけでなく，ソフトウエアのデバッグにも利用されるようになった。

Linuxなどの高機能なOSや，マルチコア型マイクロプロセサを組み込み機器に採用する例が増えている。こうしたシステムに向けたソフトウエア開発の効率化を狙い，我々はJTAG†インタフェースで接続するICE†（以下，JTAG-ICE）と，デバッガ・ソフトウエアから成る，新たな設計思想に基づくデバッガ・システムを開発した。1台のJTAG-ICEだけで，あたかも複数のJTAG-ICEを接続したかのように，複数のCPUコアやプロセスをデバッグできる。このために組み込みソフトウエアやマイクロプロセサの構造の違いをプログラマーから隠ぺいする「仮想化」の概念を持ち込んだ。

姿を変える組み込みソフトウエア

組み込み機器のソフトウエア実行環境は，ここにきて大きな変化を迎えている。その要因は2つある。1つが仮想記憶方式を採るOSを使うようになったこと。もう1つが，

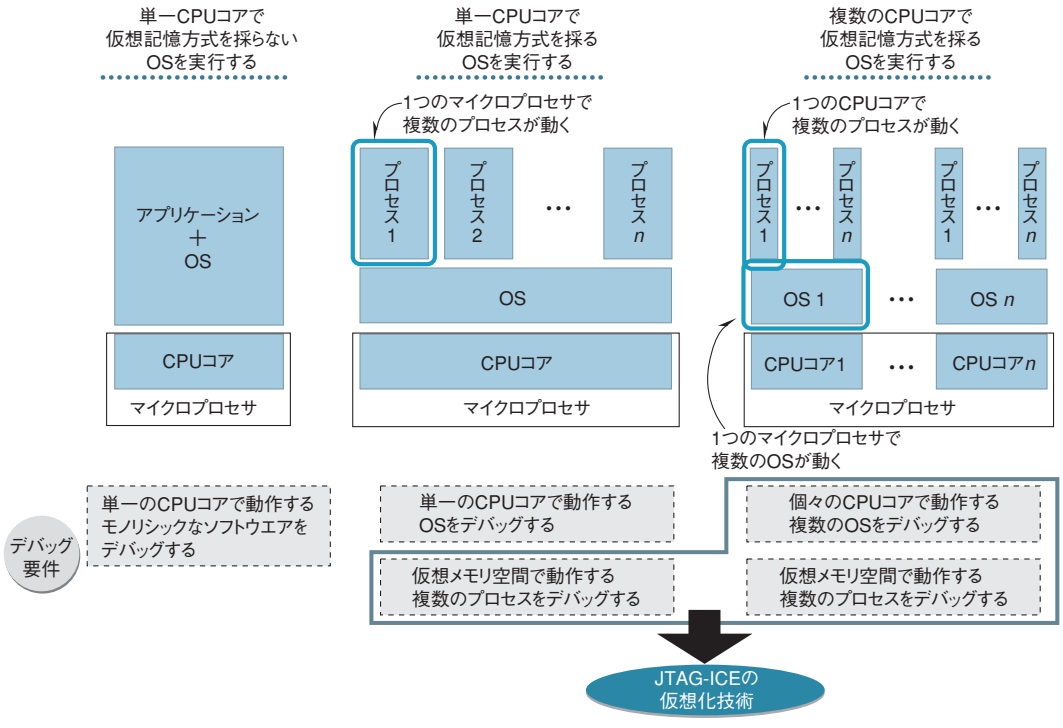


図1　複数のプロセスとCPUコアのデバッグが必要に
μITRON仕様のOSなどの仮想記憶方式を採らないOSの場合，メモリの物理アドレスをあらかじめ指定しておけるため，物理アドレスをたどればデバッグに必要な情報を取得できる。LinuxやT-Kernelなどの仮想記憶方式を採るOSでは，OSがプロセスごとに仮想的なメモリ空間を動的に定義するため，これに対応した論理アドレスを基にプロセスをデバッグする必要がある。複数のCPUコアを搭載するマルチコア型マイクロプロセサでは，それぞれのCPUコアで動作するOSや，その上で実行する複数のプロセスをデバッグできなければならない。

複数のCPUコアを集積したマイクロプロセサやSoCを搭載し始めたことである(**図1**)。

Linuxや「T-Kernel」「Windows CE」といった仮想記憶方式を採るOSでは，カーネルが複数のプロセスを立ち上げ，その後，各プロセスのスケジューリングを行う[注1)]。このため，独立した論理アドレスに基づく仮想的なメモリ空間で動作するプロセスのデバッグが必要になっている。これまで広く使われてきたμITRON仕様のOSを利用する場合は，OSのカーネルとアプリケーション・ソフトウエアを1つのソフトウエア・イメージにコンパイルする。物理アドレスに基づく単一のメモリ空間で動作するため，論理アドレスを考慮する必要がなかった。

マルチコア構成のマイクロプロセサやSoCを使う場合も，デバッガに工夫が必要となる。多くの場合，それぞれのCPUコアでOSのカーネルが動作し，それらのカーネルが複数のプロセスを生成することになるからだ。デバッガは，これら複数のカーネルとプロセスを扱わなければならない。

多くの対象を制御する必要が

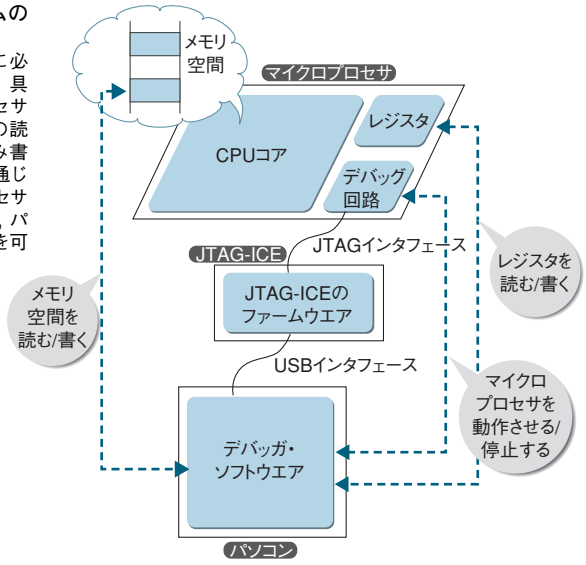
デバッガ・システムに必要な機能は3つに集約される(**図2**)。① CPUコアを動作させ，任意の時点で停止し，再び動作させるというCPUコアの制御，② マイクロプロセサが備えるレジスタに格納されたデータを読み出したり書き換えたりする，レジスタの制御，③ 主記憶に確保したメモリ空間に格納されたデータを読み出したり書き換えたりする，主記憶の制御，である。仮想記憶方式を採るOSとマルチコア型マイクロプロセサを扱うデバッガでは，これらの機能を複数の対象について実現する必要がある。

仮想記憶方式を採るOSでは，OSがプロセスごとの仮想メモリ空間を生成し，複数のプロセスにCPUコアの計算時間を割り当てる。つまり，OSが複数の仮想的なCPUコアとそれぞれに対応する主記憶を定義する。こうしたOSをマルチコア型マイクロプロセサで動作させると，CPUコアとソフトウエアの関係はさらに複雑になる。1つのシステムに，物理的なCPUコアとレジスタ・セットが複数存在するようになるからだ。

このため，仮想記憶方式のOSとマルチコア型マイクロプロセサを組み合わせたシステムの開発には，仮想的または物理的に存在する複数のCPUコアとメモリ空間を制御できるデバッガが求められる。

マルチコア型マイクロプロセサで実行するソフトウエアのデバッグには，JTAGインタフェースの問題もある。5端子から成るJTAGのTAP(test access port)をCPUコアごとに用意する方式では，CPUコアの数を増やしていくと早晩，パッケージのコストが容認できないレベルになってしまうからだ。多数のJTAG-ICEを購入しなければならないという課題もある。

図2　デバッガ・システムの基本機能
ソフトウエアのデバッグに必要な基本機能は3つある。具体的には，マイクロプロセサの動作と停止，レジスタの読み書き，メモリ空間の読み書きである。JTAG-ICEを通じてパソコンとマイクロプロセサのデバッグ回路を接続し，パソコン上からこうした制御を可能にする。



† ICE(in-circuit emulator)＝マイクロプロセサの動作を模擬するエミュレータのこと。もともとはマイクロプロセサの代わりにコネクタでICEを接続する，いわゆる「フルICE」を指す言葉だった。JTAG-ICEは，JTAGインタフェースでマイクロプロセサに接続し，パソコンのデバッガ・ソフトウエアと通信しながらマイクロプロセサのデバッグ回路を制御するハードウエアを指す。

注1) 複数のプロセスを立ち上げるOSを「マルチプロセス型」と呼ぶことが多い。プロセスは互いに独立したメモリ空間を持つ。このほかに，プロセスが確保したメモリ空間の中で複数のスレッドを立ち上げ，そのスレッドのスケジューリングも行う「マルチスレッド型」のOSもある。プロセス内のスレッドはメモリ空間を共有するため，データを共有しながら複数の処理を実行しやすい。

デバッガを分離して実装

こうした課題を解決するために我々は，仮想記憶方式を採るOS上で動作する複数のプロセスを監視するデバッガ・システムを着想した。マルチコア型マイクロプロセサにも1つのJTAGインタフェースで対応できると考えたからだ。

こうした目的に向け，パソコン上で実行するデバッガ・ソフトウエアのGUI機能と，マイクロプロセサのデバッグ回路を制御してデバッグ情報を送受信する機能（デバッガ・エンジン）を分離する構成を採った（**図3**）。

注2）CPUコアのアーキテクチャによってレジスタ構成が異なるため，完全なアーキテクチャ非依存にはできないが，依存部を極力減らすよう注意している。

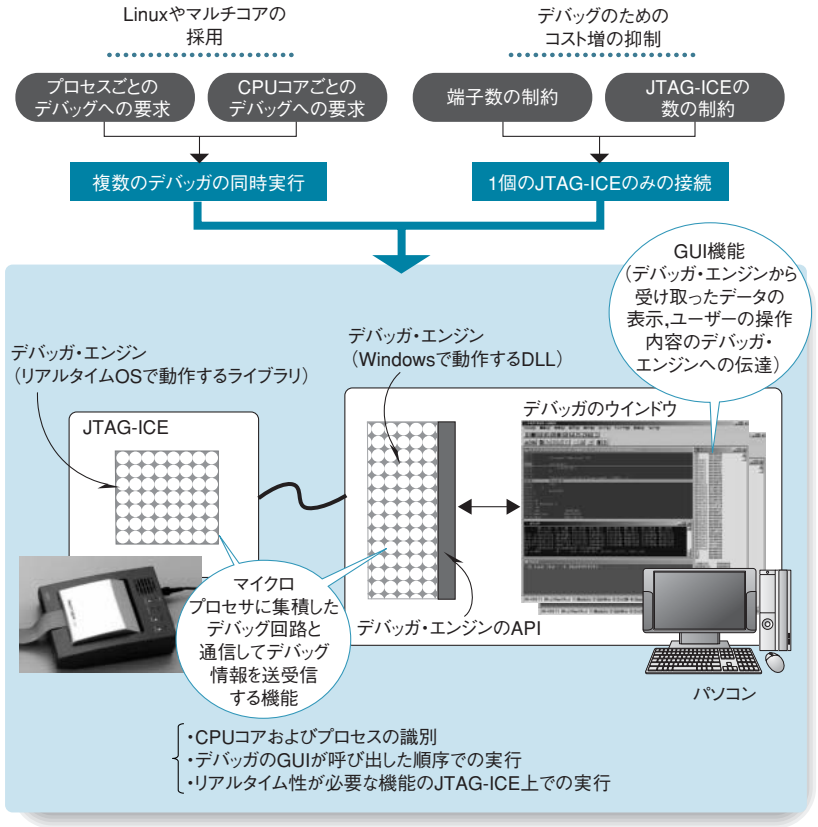


図3　デバッガのGUIとエンジンを分離
複数のプロセスやCPUコアごとのデバッグを，1個のJTAG-ICEを通じて実行できれば，Linuxやマルチコアに対応しやすくなる。そこで，デバッガのGUI機能と，マイクロプロセサのデバッグ回路を制御する機能（デバッガ・エンジン）を分離する手法を採った。複数のデバッガ・ウインドウが必要な機能をAPIで呼び出すことになる。デバッガ・エンジンはCPUコアやプロセスを識別する機能を備えており，デバッガ・ウインドウから呼び出された順番で実行する。パソコンとJTAG-ICEを接続するUSBインタフェースがリアルタイム性を損なわないために，デバッガ・エンジンの一部をJTAG-ICE上で実行する構成にした。

デバッガ・エンジンは，パソコンまたはJTAG-ICEで動作するライブラリ・ソフトウエアとして実装し，これらのライブラリを呼び出すAPIを定義した（**表1**）。デバッガのGUIソフトウエアがレジスタや主記憶の書き込みおよび読み出しといった機能を呼び出すと，ライブラリが実行される。

ライブラリを呼び出す順序には制約を設けず，呼び出された順番通りにライブラリを単純に実行するようにした。このため，特定のプロセスをデバッグするためのGUIと，OSのカーネルをデバッグするためのGUIなどのように複数のデバッグ用GUIソフトウエアを立ち上げても問題が発生しない。1つのJTAG-ICEを通じて，個々のプロセスをデバッグするための制御や，カーネルをデバッグするための制御ができる。

複数のデバッガ・ウインドウからの要求を効率よく処理するためには，JTAGインタフェースの通信帯域を最大限に活用する必要がある。このため，半分以上のライブラリ・ソフトウエアをJTAG-ICE上で実行するようにした。JTAG-ICEには240MHzで動作する「SH-4」コアを備えたマイクロコントローラと，64Mバイトの主記憶を搭載した。デバッグ回路を制御するライブラリ・ソフトウエアは，ターゲットのCPUコアのアーキテクチャにできるだけ依存しないように開発した[注2)]。異なるアーキテクチャのCPUコアを混載するヘテロジニアスなマルチコア型マイクロプロセサに対応するためである。

Linux環境の全モジュールを対象に

OSにLinuxを用いるシステムに向けたデバッガ・システムは，次の2つの要件を満たすように開発した。①Linuxカーネル，ローダブル・モジュール，共有ライブラリ，アプリ

ケーション・ソフトウエアのすべてをデバッグできること，②プロセスやシステム・コールの発生に追従するリアルタイム・トレースが可能なこと，の2点である（**図4**）。

Linuxには，標準的なデバッガ・ソフトウエアが存在する。アプリケーション・ソフトウエアをデバッグできる「gdb」や，カーネルをデバッグできる「kgdb」である。gdbはLinuxが備えるデバッグAPI「ptrace()」を使って実装したもので，主記憶の読み書きやレジスタの参照が可能である。しかし，複数のプロセスが連携して処理を行うときのデバッグが困難という問題がある。例えば，プロセスAを実行中にptrace()でブレークした場合，その処理結果を待っていたプロセスBでタイムアウトが発生してしまうことが多い。ptrace()を発行するgdbserverも別のプロセスとして動作しているため，2つのプロセスを同時に停止させることは至難の業なのである。またgdbやkgdbだけでは，デバイス・ドライバ・ソフトウエアのデバッグができないという問題もある。

Linuxを利用した組み込みソフトウエア全体をデバッグできるようにするために，3つの機能を実現する必要があった。すなわち，①メモリ管理ユニット上に複数生成されたアプリケーション空間の読み書き，②リンク・アドレスを動的に解決する仕組みへの対応，③データや実行コードを参照した時点で初めて主記憶に読み込む「デマンド・ページング」への対応，である。

動的なリンクに対応するために，ソフトウエア・モジュールがロードされた直後に呼び出すカーネルの関数「module_init()」に手を入れた。module_init()が呼び出されると，JTAG-ICEに何のモジュールを起動したのかを書き出すようにすることで，ローダブル・モジュールや共有ライブラリが使用するリンク・アドレスを把握できるようにした。

デマンド・ページングへの対応は，ローダブル・モジュールと共有ライブラリ，アプリケ

表1　デバッガ・エンジンのAPI一覧

分類	関数名	機能
初期化	PT_open()	デバッグDLLのオープン
レジスタ	PT_set_reg ()	ターゲット・レジスタの設定
	PT_get_reg ()	ターゲット・レジスタの取得
メモリ	PT_peek_8 ()	8ビット・データの読み出し
	PT_peek_16 ()	16ビット・データの読み出し
	PT_peek_32 ()	32ビット・データの読み出し
	PT_peek_64 ()	64ビット・データの読み出し
	PT_poke_8 ()	8ビット・データの書き込み
	PT_poke_16 ()	16ビット・データの書き込み
	PT_poke_32 ()	32ビット・データの書き込み
	PT_poke_64 ()	64ビット・データの書き込み
メモリ・ブロック	PT_put_blk ()	メモリ・ブロックの書き込み
	PT_get_blk ()	メモリ・ブロックの読み出し
	PT_mov_blk ()	メモリ・ブロックの移動
	PT_fill_blk ()	メモリ・ブロックのフィル
実行	PT_go ()	ターゲット実行
	PT_check_break ()	ターゲット実行チェック
	PT_nmi ()	ターゲット強制ブレーク
ブレークポイント	PT_set_break ()	ブレークポイントの設定
	PT_control_break ()	ブレークポイントの制御
終了	PT_close ()	デバッグDLLの終了

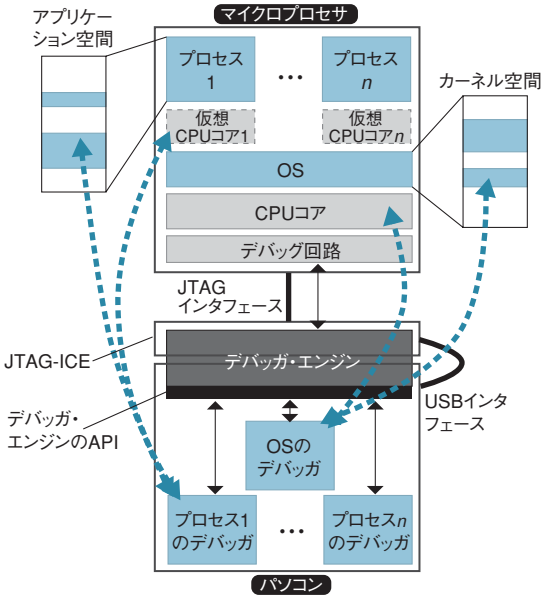


図4　プロセスのメモリ空間を動的に生成するOSに対応
デバッガ・エンジンのAPIを利用するデバッガを複数同時に立ち上げて，OSのカーネル空間と，複数のアプリケーション空間を同時に監視できるようにした。プロセスのデバッガは，OSが時分割などで提供する仮想的なCPUコアの動作/停止，仮想的なCPUコアのレジスタの読み書き，主記憶のアプリケーション空間の読み書きができる。

ーション・ソフトウエアのデバッグに必要となる。デバッグ時には，通常，ソース・コードにブレークポイントを設定する。しかし，そのソース・コードに対応する実行コードを読み込んでいない段階では，実行環境の主記憶上にブレークポイントを定義できない。いざ実行コードを読み込んだらすぐに実行し終えてしまうため，意図した通りにブレークできないことになる。そこで，ブレークポイントを設定したソース・コードに対応する実行コードを，前もって読み込ませるようにした（**図5**）。

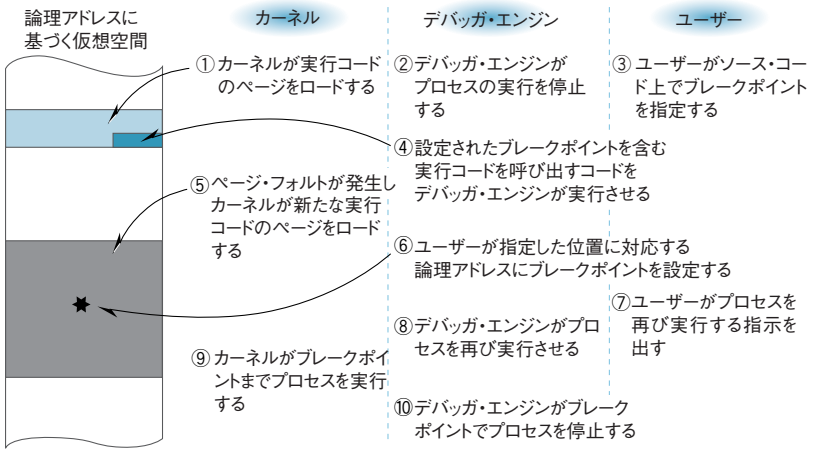


図5　ページ・フォルトを意図的に発生させる
Linuxカーネルは，一度にすべての実行コードを主記憶に読み込むわけではない。実行することになったコードが主記憶に読み込まれていないことを検知する，いわゆる「ページ・フォルト」が発生して初めて当該の実行コードを読み込む。カーネルがまだ読み込んでいない部分にもブレークポイントを設定可能にするために，意図的にページ・フォルトを発生させるようにした。ユーザーがソース・コード上で指定したブレークポイントに対応する実行コードを前もって呼び出す。

分岐先がプロセスかカーネルかを判別

組み込みソフトウエアのデバッグでは，どのような順序でそれぞれの処理を実行しているかという情報も必要である。不要なプロセスを実行していたり，1つのプロセスに予想以上に多くの時間をかけていたり，といった問題を抽出するために，一般に「リアルタイム・トレース」と呼ぶ機能を使う。Linuxを搭載したシステムでこうしたリアルタイム・トレースを実現するために，デバッグ回路が分岐先の論理アドレスを出力した際に，スイッチ先がカーネルかプロセスか，プロセスの場合はどのプロセスなのかを判定できるようにした（**図6**）。このために，コンテクスト・スイッチを実行するカーネルの関数に独自のパッチを当てた。

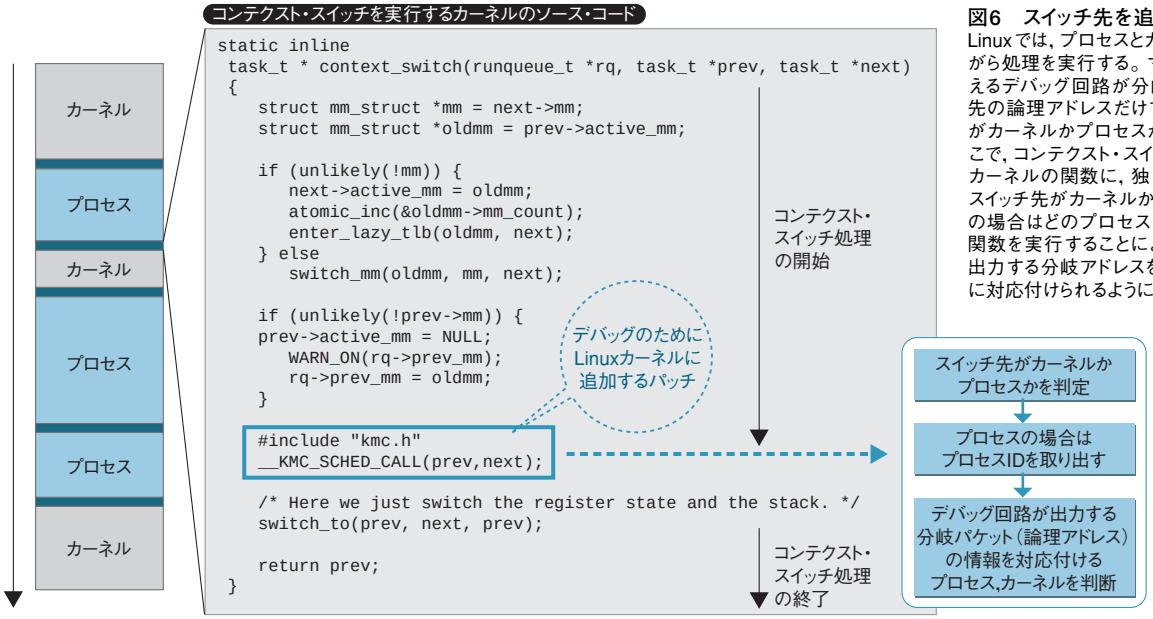
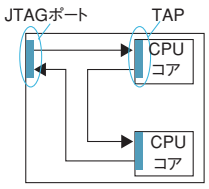
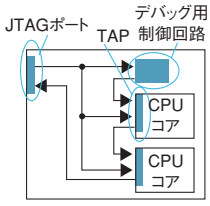
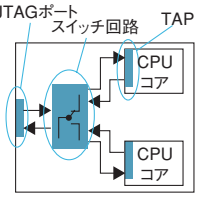
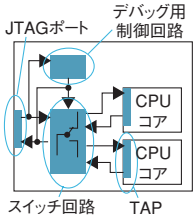


図6　スイッチ先を追いかける
Linuxでは，プロセスとカーネルをスイッチしながら処理を実行する。マイクロプロセサが備えるデバッグ回路が分岐時に出力する分岐先の論理アドレスだけでは，実行中のコードがカーネルかプロセスかを判断できない。そこで，コンテクスト・スイッチを実行するLinuxカーネルの関数に，独自のパッチを当てた。スイッチ先がカーネルかプロセスか，プロセスの場合はどのプロセスIDなのかを判定する関数を実行することにより，デバッグ回路が出力する分岐アドレスをカーネルやプロセスに対応付けられるようにした。

表2　JTAGインタフェースのCPUコア間接続手法

接続方法	カスケード接続		パラレル・スイッチ接続	
	制御回路なし	制御回路あり	制御回路なし	制御回路あり
回路構成				
CPUコアごとの電源オフへの対応	×	×	○	○
ハードウエアによるCPUコア間の同期	×	○	×	○
JTAG通信の最適化	×	×	○	○
追加が必要な回路規模	追加なし	中	小	大

TAP：test access port

マイクロプロセサが備えるデバッグ回路は，分岐処理が発生した時点で分岐先の論理アドレスを出力する機能を備える。物理アドレスが論理アドレスと等しい場合は，このアドレスだけでソース・コードと照らし合わせたリアルタイム・トレースが可能である。しかしLinuxは，物理アドレス上にカーネルが確保した仮想メモリ空間の論理アドレスを使う。さらに，アプリケーション空間を多重化しているため，どのプロセスの論理アドレスかを判別する必要があった。

我々がこうしたデバッガ・システムを構築できたのは，Linuxカーネルがオープンソースだからという側面もある。カーネルが受け持つ機能をソース・コード・レベルで分析し改変した。これにより，カーネルが持つ情報を効率よくJTAG-ICEに吸い上げられるようになった。

デバッガに必要なマルチコア接続方法

次に，今回構築したマルチコア型マイクロプロセサ向けのデバッグ環境について解説する。マルチコア型マイクロプロセサの場合，デバッグ性能を高めるためにはJTAG-ICEだけでなく，LSIにどのような回路を実装するかも重要な問題となる。

JTAGインタフェースは，制御対象ごとに備えるTAPをカスケード接続し，特定のTAPだけに通信できる仕様となっている。JTAGインタフェースを使ったデバッグを想定した場合，JTAGポートと制御対象のCPUコアの接続方法には，4種類が考えられる。カスケード接続にするか，スイッチ回路をLSIに集積したパラレル・スイッチ接続にするかという選択肢と，CPUコア間の同期を取る機能などを備えるデバッグ用のCPUコア制御回路を集積するかしないかという選択肢がある(**表2**)。

我々が推奨するのは，パラレル・スイッチ接続で，デバッグ用制御回路を集積する方式である。CPUコアごとの電源オフへの対応，CPUコア間のハードウエアによる同期などが可能になるためだ。また，通信経路が物理的に長くなり，動作周波数が低いCPUコアが通信速度のボトルネックとなるカスケード接続と比べ，対象のCPUコアとの通信を最適化できるという利点もある。ただし，他の方式に比べて，追加する回路規模が大きくなる。それでも，2個～4個のCPUコアを搭載する場合で1000ゲート前

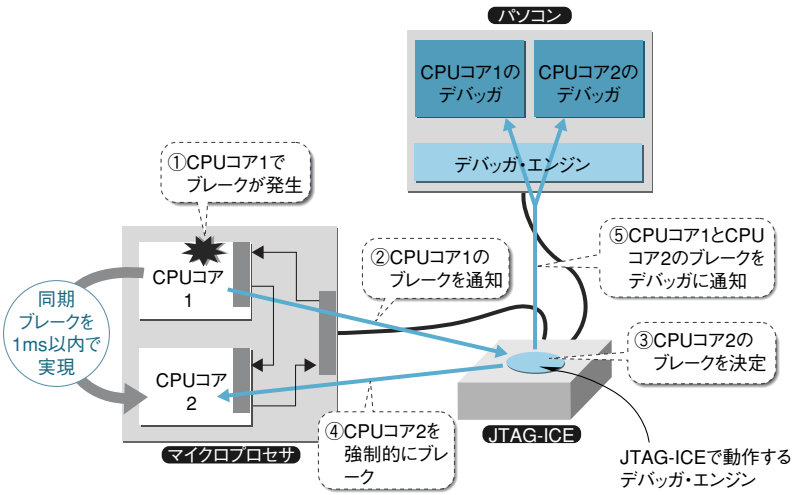


図7　ソフトウエアによる同期ブレークで1ms以内を実現
あるCPUコアで発生したブレークに同期して別のCPUコアを停止する処理を，JTAG-ICEに搭載したソフトウエアで制御する仕組みを構築した。1つのCPUコアで発生したブレークの情報を受け取ると，まず別のCPUコアを強制的に停止する。その後，パソコン上のデバッガに，ブレークしたことを通知する。

後である。現在のマイクロプロセサの回路規模から考えれば，それほど大きなオーバーヘッドではないと考える[注3)]。

制御対象となるCPUコアごとに，JTAG-ICEなどで動作するライブラリ・ソフトウエアがJTAGインタフェースを通じてデバッグ情報を通信できるようにした。複数のJTAGインタフェースを備えているように見せる，JTAGインタフェースの仮想化を実現している。

ソフトウエア・ブレークで1msを達成

同期ブレークや同期実行を実現する回路を持たないLSIも多い。そこで我々は，デバッガ・ソフトウエアとJTAG-ICEだけで複数のCPUコアの同期ブレークと同期実行を実現した。厳密な同期ではないが，1個のCPUコアでブレークが発生してから1ms以内で別のCPUコアをブレークできる。同期実行も1ms以内を達成している。

パソコンとJTAG-ICEの接続に使うUSBインタフェースはベスト・エフォート型の通信インタフェースであり，レイテンシが長くなる。そこで，同期ブレークのためのデバッガ・エンジンはパソコンではなく，JTAG-ICE上で実行するようにした。1つのCPUコアでのブレークを検知したJTAG-ICEが，別のCPUコアを強制的にブレークし，その後でパソコン上のデバッガ・ソフトウエアにブレークしたことを通知する（**図7**）。

我々は，JTAG-ICEの仮想化技術を確立したことにより，既存のJTAG-ICEよりも広い用途に使えるであろうと考えている。その1つが，性能解析である。100μsに1回，JTAG-ICEがタイマ割り込みを発生させ，その割り込み処理の終了後に戻るアドレスをJTAG-ICEの主記憶に書き出す機能を実装した。JTAG-ICEにアドレス・データを格納するだけならば15分間，パソコンにアドレス・データを書き出すようにすれば，より長時間の実行履歴が分かる。100μsの間にCPUコアは多数の命令を実行するが，どの関数を何回実行したのか，どこが処理のボトルネックになっているのか，などを把握しやすくなる。本来のソフトウエア実行に与える性能上のオーバーヘッドは，CPUコアに「ARM9」を使った場合で1％未満である。

このほか，ターゲット上のシリアル通信ポートやEthernetポートの代わりに，汎用通信ポートとして仮想化したJTAGポートを使えるようにした。ターゲットとのデータ・ブロックの送受信や，1バイトのデータの送受信などを実行できるライブラリを用意し，そのAPIを公開している。周辺ハードウエアを持たない開発ボードでも，JTAGインタフェースに接続できれば各種のインタフェースを備えた機器のソフトウエアを開発できるようになった。今後は，JTAG-ICEを仮想的なファイル・システムとして利用したり，品質検査に利用したりできると考えている。NE

注3）既にこうした接続方式を採ったLSIを開発した半導体メーカーもある。