「KMC/PARTNER-Jet JTAG-ICE」

Linux 対応デバッガ技術解説

京都マイクロコンピュータ株式会社

はじめに

ネットワークやマルチメディアなど組み込み機器に求められる機能の増加に伴い，豊富な機能と動作実績がある Linux が採用される機会が増えてきました．Linux を使用したソフトウェア開発と言っても,動作確認済みのハードウェアと OS 環境での開発が主体となるエンタープライズ系の開発と異なり，デバイスドライバの開発や OS のチューニングなど組み込み機器でのデバッグ作業には様々な要素が求められます．

本文書では，京都マイクロコンピュータ製の PARTNER-Jet JTAG デバッガの Linux 対応とデバッグ機能について解説します．製品の解説も含んでいますが，内容の多くは Linux 上の開発とデバッグのテクニックと JTAG-ICE に関する一般的なものです．

PARTNER-Jet JTAG デバッガの Linux 対応は，「JTAG-ICE が OS の動作を理解しながら動作する」というこれまでの ICE とは全く異なるアプローチで機能連携を行うものです．この仕組みを実現するために採った方法と実現できたこと自体が Linux カーネルを理解することにもつながると思います．実現には Linux カーネルがオープンソースであることも大いに役立っています．

今後の組み込み機器で Linux をはじめとした仮想記憶方式を使う高機能 OS が使用されていくことと，JTAG-ICE などのハードウェアに近いレベルでのデバッグを実現する仕組みが求められ続けることは，止まりようのない流れになっていると思います．本文書が Linux，組み込み機器およびソフトウェア開発の一助となれば幸いです．

京都マイクロコンピュータ株式会社

本社:	〒610－1104 京都市西京区大枝中山町 2-44
TEL:	075-(335)-1050
FAX:	075-(335)-1051
サポート TEL:	075-(335)-1060
URL:	http://www.kmckk.co.jp/
E-mail:	jp-info@kmckk.co.jp
東京オフィス:	〒105-0004 東京都港区新橋 2-14-4 R ビル 5F
TEL:	03-5157-4530
FAX:	03-5157-4531

2006 年 10 月

目次

デバッガと LINUX についての基礎知識 1
1. JTAG-ICE (PARTNER-JET) について 1
2. デバッガのしくみの基本知識 1
2.1. ソースコードデバッグ 2
2.2. ブレークポイント 2
2.3. トレース 2
3. デバッガに LINUX 対応が必要な理由 2
4. なぜ JTAG-ICE を使うのか 3
4.1. GDB（PTRACE） 3
4.2. ポスト PTRACE 4
4.3. KGDB 4
4.4. 求められているデバッグ機能 4
5. デバッグするための課題 5
5.1. ブートローダ 5
5.2. LINUX カーネル 5
5.3. ドライバモジュール 7
5.4. アプリケーション 7
5.5. 共有ライブラリ 8
5.6. プロセス間共有メモリ 9
LINUX 対応の方法（PARTNER-JET の場合） 10
6. 課題解決の方針 10
7. 課題解決の具体方法 11
7.1. デマンドページングの解決 11
7.2. 多重空間への対応の解決 11
7.3. リロケーションの解決 11
7.4. ユーザー空間とカーネル空間の認識 11
8. プログラムの修正 12
8.1. カーネルへのパッチ 12
8.2. サポートファイル 12
8.3. ICE フックファイル 12
9. まとめ 12
PARTNER-JET による LINUX デバッグ 15
10. ICE デバッグのための準備と方法 15
10.1. CFG ファイル設定 15
10.2. 起動オプション設定 15
10.3. LINUX 特有の起動オプション設定 15
10.4. カーネルの変更 16
11. カーネルのデバッグ 16
12. XIP カーネルのデバッグ 16
12.1. カーネルデバッグの方法（１） 16
12.2. カーネルデバッグの方法（２） 16
13. カーネルと実行トレース 16
14. ドライバモジュールのデバッグ 16
14.1. ドライバモジュールデバッグの方法 17
15. ユーザー空間のデバッグ 18
15.1. アプリケーションデバッグ（通常） 18
15.2. アプリケーションデバッグ（アタッチ） 18
15.3. アプリケーションとカーネルを同時にデバッグ 18
15.4. アプリケーションデバッグの方法 18

15.5. 共有ライブラリデバッグ 18
その他のデバッグに関するトピック 20
16. より使いやすく 20
16.1. 例外とハードウェアブレークポイント 20
16.2. 最適化コンパイルのデバッグを快適に 20
16.3. デバッグ情報の読み込みを自動化 20
16.4. PRELINK のデバッグも OK 20
16.5. オープンソースと LINUX 20
参考文献 22
著者 22

デバッガと Linux についての基礎知識

1. JTAG-ICE (PARTNER-Jet) について

世の中には様々な種類のデバッガがありますが，「ICE (In-Circuit Emulator)」は CPU で実行されるプログラムを直接デバッグするための装置です．ICE には Full-ICE，ROM-ICE といった種類がありますが，近年では CPU の周波数の増加もあり，高速な CPU では実現が困難になりました．現在では IEEE 1149.1 で標準化された CPU 内蔵のオンチップ・デバッグ機能（JTAG）を用いてデバッガを実現する JTAG-ICE が一般的です．

JTAG-ICE デバッガを使用するには，ターゲットの CPU から JTAG 用の信号線を出して，デバッグをするためのコンピュータに接続する，「クロス開発」のスタイルになります．

PARTNER-Jet を含め多くの場合デバッグ操作用ホストには Windows/PC を使用します．PC とターゲットボードは，PC の通信ポート（RS-232C，パラレルポート，USB など）を使用してターゲットには JTAG のケーブルを接続します．

最も安価な接続方法は，PC の通信ポートとターゲットのボードを単純な変換器を経由して繋ぐことです．接続に用いるポートの通信速度によってデバッガの反応も遅くならざるをえません．また，ケーブル上を JTAG 制御のためのデータが流れますので PC 上で動作するデバッガソフトウェアが全て制御する必要があり，PC 上のソフトウェアの動作負荷も高くなりがちです．

PARTNER-Jet の場合は通信ポートには USB 2.0（または Ethernet）を用います．USB 2.0 は帯域が広く，大量データの転送が可能ですが，少量のデータを送受信する場合のレイテンシは速くありません．PARTNER-Jet はそれ自体が CPU に SH4 を採用したコンピュータで，ターゲットボードの JTAG の制御やトレースの記録，デバッグに必要な様々な情報の解析を行うことが出来るため，PC と PARTNER-Jet 間の通信量を抑えることができ，快適な速度でデバッグを行うことができます．

図 2 JTAG ケーブルによる接続

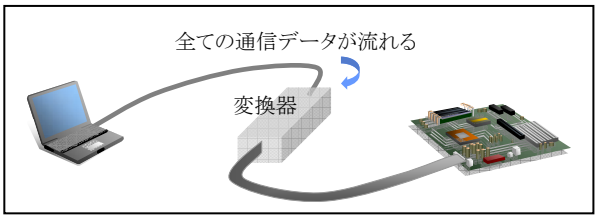
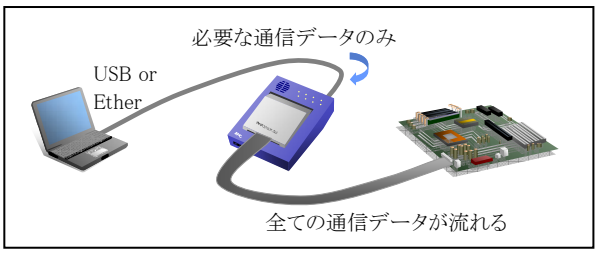


図 3 PARTNER-Jet の接続



2. デバッガのしくみの基本知識

デバッガはプログラムの不具合を発見しやすくするためのツールですので，製品ごとに様々な工夫が盛り込まれることが考えられますが，おおまかな機能はどのデバッガでも似ています．「デバッガを使ってデバッグ」をするときには以下のような操作をしていると思います．

- プログラムをデバッグ情報付きでコンパイルしておくことで，ソースコードと連動して実行する
- ブレークポイントを張って実行を止め，変数やメモリの内容を参照する
- 実行の履歴を取る

これらのデバッガの機能は，どのようにして実現されているのかを簡単に説明します．

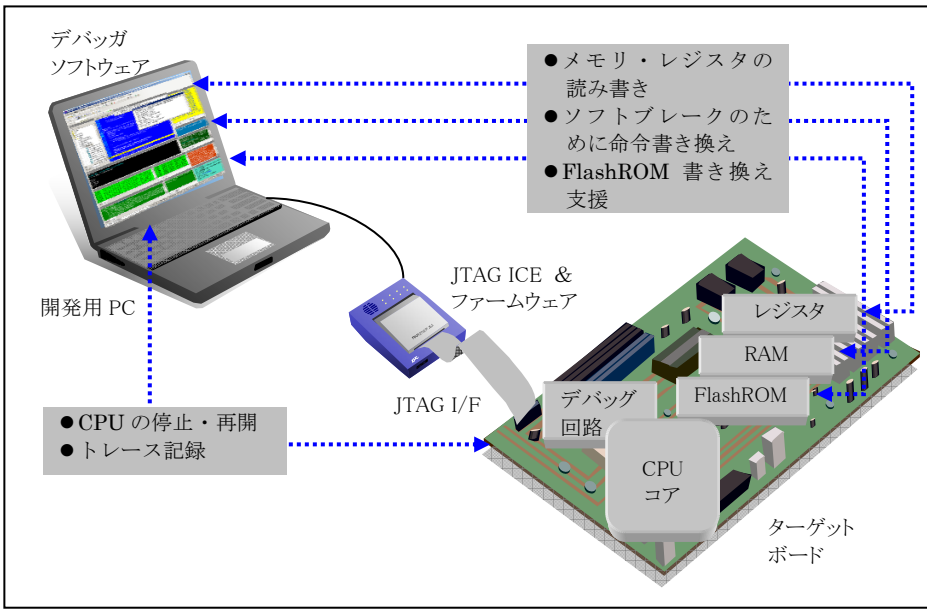


図 1 JTAG-ICE デバッグ

CPU 内蔵のデバッグ回路と接続して使う JTAG-ICE は，自由に CPU を停止・再開できる．実行履歴を解析して過去の状況を再現したり，OS に気づかれないように RAM を書き換えることも可能だ．
しかし，ソフトウェアブレークは「**RAM 上の実行命令を書き換える**」ことで実現されるため，止めたい命令の在りかがわからないと使いようがない．

2.1. ソースコードデバッグ

デバッガはデバッグ情報を読み込み，プログラムの実行箇所とソースコードの行を，デバッグ情報を元に対応付けを行い，ユーザーに表示します．

2.2. ブレークポイント

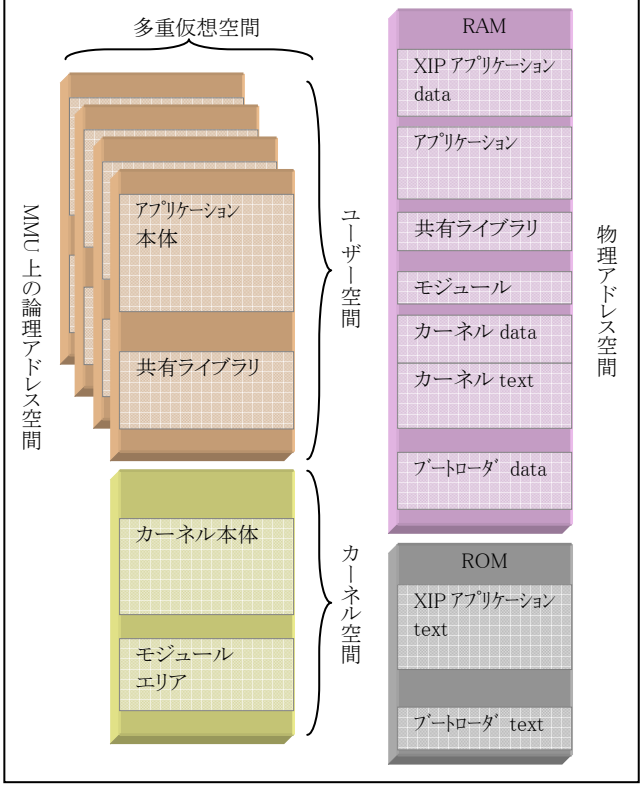
ブレークポイントには，ソフトウェアブレークポイントとハードウェアブレークポイントの 2 種類があります．

ソフトウェアブレークポイントは**メモリ上の実行命令をデバッガが書き換えることで実現します**．書き換えられたデバッガ専用の命令を実行すると CPU が停止します．デバッガは書き換える前の命令を退避して保持しているので，書き戻しを行うことで実行を続けることができます．プログラムが ROM 上にある場合は使用することができません．

ハードウェアブレークポイントは JTAG 機能によって，CPU 実行ユニット内のデバッグユニットが監視をすることで実現されます．多くの場合は物理アドレスを指定してブレークポイントを設定します．ソフトウェアブレークポイントと異なり，メモリ上のプログラムには変更を加えないため，プログラムが ROM に置かれていても使用することが出来ます．CPU の内蔵機能ですので，ブレーク条件など設定できる内容は CPU 毎に多少異なります．

JTAG-ICE はソフトウェアブレーク用命令や JTAG のハードウェアブレーク機能を駆使して CPU を自在に停止して RAM やレジスタを読み書きできますので，JTAG-ICE が行う操作はターゲット CPU 上の誰にも気づかれずに行えます．ある意味でスーパーバイザーモードで動作する OS よりも強い権限で動作していると言えます．

図 4 アドレス空間イメージ



2.3. トレース

JTAG 機能によって CPU 実行ユニットの外へ実行情報を出力し，デバッガが情報をデバッガのメモリに記録していくことで実現されます．デバッガが情報を解析し，プログラムのデバッグ情報と合わせて再構成することでブレークした時点では既に消えてしまっている過去の状態を再現・検証できる「リアルタイムトレース」または「バックトレース」，「ヒストリ」と呼ばれる実行履歴機能を実現します．

デバッグをするために対象プログラムや実行環境に変更を加えてしまうと，変更によって動作が変わってしまうことがあり好ましくありません．デバッグ用のプリント文や Assert のようなプログラム自体に変更を埋め込む方法よりは，デバッガを使用したほうが，より実際の動作に近い動きでデバッグすることができます．しかしながら，ソフトウェアブレークポイントを使用するとメモリ上の命令の書換えを行いますので，コンパイル・実行したままのプログラムの命令で動作するわけではありません．ハードウェアブレークポイントの場合は，メモリ上の命令はそのままですが，CPU が停止するので命令パイプラインやキャッシュの状態は変化することになります．デバッガはプログラムの動作を人間にとってわかりやすいように表現してみせることによって不具合の発見を助けるツールですので，

- いかに人にわかりやすく実行状態を表現するか（操作性）
- どれだけ実際の実行に近い状態を保つか（実システムとの透過性：transparency）

ということがトレードオフで，非常に重要なポイントです．

3. デバッガに Linux 対応が必要な理由

Linux は，マルチタスク処理，仮想メモリ，共有ライブラリ，デマンドローディング，メモリ管理，および TCP/IP ネットワーク機能などを備えた UNIX クローンの OS です．標準でファイルシステムやネットワークシステムのコンポーネントを含んでいるため，様々な状況で利用されるようになってきました．

しかしながら，JTAG デバッガを使ったことがある方ならば，「デバッガが OS に対応する」ということ自体に違和感を覚えるかもしれません．実際，OS 無しの場合や，組み込み機器で多く用いられてきた μiTRON や Windows CE（Ver.5 以前）のように MMU を使わないか，もしくは物理メモリの領域分けのためにのみ MMU を使用し，シングル CPU でのみ動作する OS の場合は，OS に依存することなく JTAG デバッガを使用することができます．それは，物理アドレスそのまま，もしくは MMU を使用した仮想アドレスでも固定のエリアのアドレスで使用されるようにプログラムがリンク時にアドレス解決されているため，デバッガが OS に対応しなくてもデバッグ情報をそのまま利用できたためです．

ところが Linux の場合は MMU を仮想記憶方式ために使用するため，より複雑な状況になります．Linux 上で動作するユーザー空間のアプリケーション（プロセス）は図 4のプログラムの配置される空間のイメージのように，多重仮想空間に配置され，各プロセス毎に同一アドレス存在します．
リロケータブルなものや，デマンドページングが行われるものなどは，デバッガから見ると，「デバッグ情報に対応するアドレスが決まっていない」，または「デバッグ情報に対応するア

ドレスにプログラムが存在しない」ということになります．そのため，デバッガの基本機能であるソフトウェアブレークポイントを使用するための前提**「RAM 上の実行したいアドレスにある命令を書き換える」**ことが一筋縄では行かなくなってしまいます．

存在しない情報をデバッガで補うことは不可能なので，何らかの形で OS が所有している情報を実行時に引き出して補う必要あり，ここにデバッガが OS に対応することの必要性がでてきます．

つまり，実行時にしか決まらない情報や動的に変化する情報を持つプログラムは従来の方法では扱うことが出来ないため，可能な限り人間にわかりやすいように表現して不具合解決の補助とするにはどうすれば良いかを考える必要があります．

4. なぜ JTAG-ICE を使うのか

Linux でプログラミング・デバッグを行ったことがある方なら GDB を使ったことがあるのではないでしょうか．JTAG-ICE を Linux で使うためにわざわざ対応が必要になるのであれば，なぜ GDB ではだめなのか，もしくは JTAG-ICE と GDB の併用ではだめなのかという疑問が出るかもしれないと思います．そこで，Linux 上のデバッグ環境にどのような改善が求められているのか状況を説明します．

4.1. GDB（ptrace）

GDB は実行プログラムを自身の子プロセスとして実行し，子プロセスとカーネルとの間に介入しながらデバッグを行います．これを「ユーザーモードのデバッガ」と呼びます．GDB をはじめとするユーザーモードのデバッガは ptrace(2) システムコールを利用しています．ptrace では 2 種類の方法でユーザープロセスのデバッグを行うことができます．

(1) 子プロセスを生成し，生成したプロセスをデバッグ（PTRACE_TRACEME）
(2) 既存の実行中プロセスに対してアタッチしてデバッグ（PTRACE_ATTACH）

この方式のデバッガはソフトウェアのみで実現することもできるため手軽ですが，マルチプロセス・マルチスレッドデバッグ機能は不十分ですし， ptrace 機能そのものにも物足りない点があります．

ptrace の起源は古く，Unix 上のユーザープロセスをデバッグするための仕組みとして考案され，「スレッド」の仕組みが出来るよりも前に作られたものです．プロセスにシグナルを送信すると，シグナルを受信したプロセスは通常のプログラムの実行箇所の処理を中断してシグナルハンドラの処理を行います．この仕組みを利用してプロセスを停止（break）することでデバッガ機能を実現します．ptrace(2)を使用して組み込みシステムのプログラムをデバッグするのに不向きな理由は主に3つあります．

(1) デバッグ可能対象範囲が狭い

基本的にユーザー空間のプログラムのデバッグのために作られた仕組みなのでカーネル空間のデバッグには使えません．カーネルやデバイスドライバと関連するデバッグはプリント文(printk)頼りになってしまいます．

(2) シグナルのオーバーヘッド

ptrace(2)デバッグ対象のプロセスの実行を停止するためには必ずシグナルの送信を伴うということです．プロセスにシグナルを送信するということは，カーネル内ではプロセススケジューラを動作させて，プロセスがシグナルを受信できるように処理しなければなりませんので break したいタイミングの即時性があまりありません．

つまり，複数のプロセスをデバッグ中に，それぞれのプロセスを「同時」にブレークしたいとしても対象プロセスの全てにシグナルを送信しなければなりません．デバッグ対象のプロセスまたはスレッド数が増えるにつれてシグナルの配送のオーバーヘッドや伝達の遅延が無視できなくなります．多くても 10 のプロセスを同時デバッグするのが限界ともいわれています．

シグナルの受信はプロセス単位で行われるので，マルチスレッドのプログラムとは相性がよくありません．

また，シグナルハンドラを利用するプログラムの場合，特定のシグナルをデバッガがハンドリングしないように設定する,といった操作が必要になり不向きです．

(3) カーネル内の動作

ptrace(2) のためにはカーネルやデバイスドライバのコード

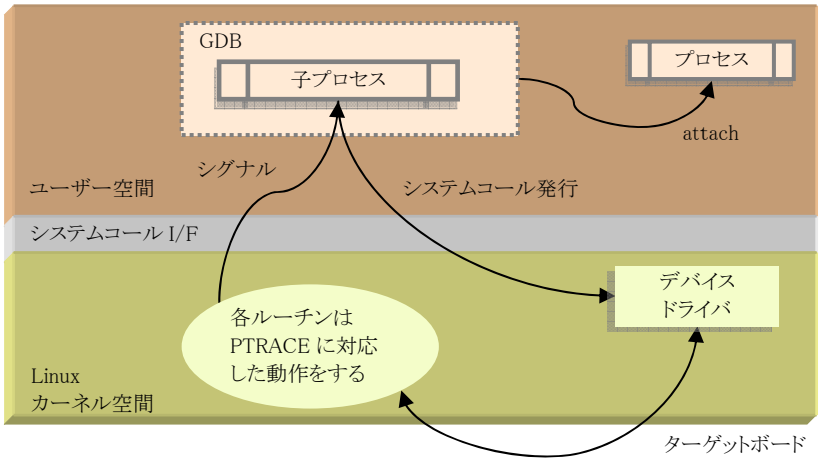


図 5 ユーザーモードデバッガ

ptrace システムコールを使用したデバッグでは特別なハードウェアを必要としない代わりに，OS の動作を停止できない．そのためデバッグ対象はユーザー空間のプログラムのみに限定される．
ブレーク(実行停止)のためにシグナルを使用することや，デバイスドライバ等カーネル内のルーチンは ptrace に対応するように適切に処理されている必要があるなど，デバイスドライバの開発を伴うことが多い組み込み分野では特に不便さがある．

中に適切なタスク状態チェックのコードを埋め込んでおかなければなりません．ルールに従っていないデバイスドライバがあった場合，「システムコール実行中の break」などの動作がおかしくなります．
デバッグ対象のユーザプログラムが開発途中のデバイスドライバを経由したデータを読み書きしている場合などはデバッガの動作自体が正しく行えないことがあります．
他にもユーザー空間のみのデバッグ手法としては，以下のようなツールもあります．いずれもデバッガではなく解析用ツールです．

- メモリ破壊，開放漏れなどのチェックツールを使う
- strace(システムコール発行履歴を取る)

4.2. ポスト ptrace

ptrace は Linux で登場した機能ではありませんが，アーキテクチャ固有の実装になるため，Linux においては各アーキテクチャ版毎に実装や動作の仕方が異なっているということも問題になっていて，Linus Torvalds 氏も「ptrace は wonderful とは言えない」と発言しています．
Linux カーネルに ptrace を代替・機能強化する仕組みが提案(utrace: 文献[3])もされていますが，現時点ではまだパッチはカーネルに採用されていません．

4.3. kgdb

kgdb というカーネル専用のデバッガもあります．しかし，普通の JTAG-ICE と同様，カーネル空間のデバッグしかできません．kgdb はカーネルのリモートデバッガで，ターゲット上でカーネル内のデバッグデーモンのように動作し，別のデバッグホスト PC の gdb と組み合わせてデバッグします．ホスト PC との通信中は割り込み禁止になるため，実動作と変わってしまう可能性がありますので，組み込みの開発で求められるデバッグとしては kgdb よりは通常の JTAG-ICE を使うほうが良いと考えられます．
他にも Linux カーネルのみのデバッグ手法としては，以下のようなものがありますので参考までにご紹介します．

- kdb

Linux カーネル専用のデバッガで kgdb と違い，ターゲットマシン単体上で動作します．モニターに近い機能で，残念ながらソースコードデバッグはできません．

- printk

カーネル空間内で使えるプリント文です．プログラムを停止させずに動作を見たい場合によく使用されます．

- Oops 分析

カーネルクラッシュ時にシステムコンソールに表示される Oops(パニック)メッセージを分析する方法です．ksymoops といった解析用ユーティリティを使用して，クラッシュの原因を調べます．

4.4. 求められているデバッグ機能

どのような状況・局面をデバッグしたいのかを挙げてみます．

- デバイスドライバとアプリケーションを一緒にデバッグしたい
- CPU の実行トレースを参照して問題解析したい
- 割り込みハンドラの最下層からデバッグしたい
- Linux の初期化(init プロセス)をデバッグしたい
- XIP カーネルをデバッグしたい
- カーネルやドライバが落ちたシステムフリーズ時のデバッグをしたい
- プロセス間でまたがるシーケンスをデバッグしたい
- マルチスレッドやマルチプロセスを完全にデバッグしたい
- 特定のプロセスでの特定のメモリライトを捕まえたい
- ターゲット資源(RAM など)に負荷をかけないでデバッグしたい

デバイスやドライバの開発中にアプリケーションの開発も平行して行うシーンが多く見られる組み込みシステム開発ならではの要望があります．
ユーザーモードデバッガでうまく扱えていない問題とも重なっていますが，本来 ICE が行っている CPU を停止したデバッグばかりでなく，カーネルやドライバは動作させたままプロセスのデバッグをするなど「システム全体を総合的にデバッグできる」仕組みが求められています．
ユーザーモードデバッガではカーネル空間のデバイスドライバとユーザープロセスを同時にデバッグすることは原理的に無理なので，JTAG-ICE に求められています．

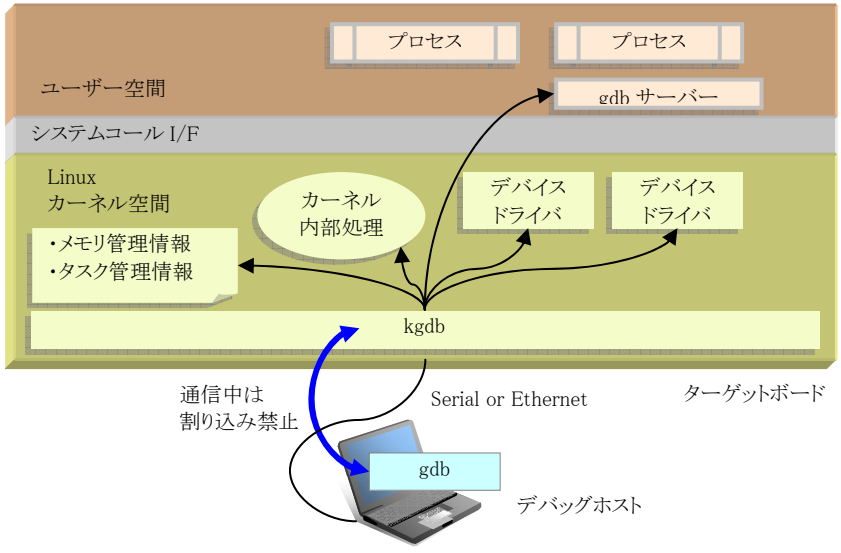


図 6 kgdb の方法

kgdb はターゲット上のカーネル空間で動作するサーバープログラムのようなもので，リモートデバッガの機能を提供する．リモートのホスト PC で動作する gdb と連動してデバッグを可能にする．
通信中は割り込み禁止になってしまうので組み込み機器の開発時に用いるツールとしては心もとない．

5. デバッグするための課題

Linux 上で動作するソフトウェアはメモリ空間や配置のされ方で表 1のように分類できます. それぞれのソフトウェアについて, デバッガから見たデバッグ方法の違いとデバッガとして対応しなければならない問題点について順に解説していきます.

5.1. ブートローダ

どのようなシステムであれ, ブートローダは必要になります. 通常ブートローダは ROM に置かれて電源投入後に最初に動作し, 役割はターゲットのハードウェアを初期化してからカーネルを RAM に転送します. (図 7参照)

電源投入時, MMU はオフになっていますので, ブートローダはコンパイル・リンク時に解決された固定番地で動作します. デバッグ情報のロードを行えば通常の ICE を使用したデバッグと同様です. プログラムが ROM に置かれている場合はソフトウェアブレークポイントが使えませんが, FlashROM やエミュレーションメモリを使用することで対応は可能になります.

カーネルの転送, またはカーネルへ制御を渡す部分には注意が必要な場合があります. カーネルを RAM に転送する前に RAM が読み書きできるように初期化しなければなりません. また, CPU にもよりますが TLB の初期化を行わないと論理アドレスにアクセスできません. 通常, カーネルは論理アドレスの固定番地でコンパイル・リンクされています. 転送し, ジャンプする先が論理アドレスになりますので, RAM の初期化が済んでいても, TLB の初期化前に論理アドレスにブレークポイントを設定することはできません. ブートローダを使用せずに JTAG-ICE を使用してカーネルをロードする場合は, RAM・TLB の初期化といったブートローダが行う処理をデバッガのターゲット初期化設定によって行うのが普通で, これは Linux に限ったことではありません.

5.2. Linux カーネル

通常のカーネルは, ブートローダによって RAM に転送された後に実行されます. 論理アドレスでコンパイル・リンクされているのが普通ですが, 固定番地でリンクされているためデバッグ情報をロードすれば通常の ICE を使用したデバッグとほぼ同様です. いくつか付随的なトピックがありますので以下に挙げます.

- 初期化セクション
  カーネルの作成上のテクニックとして, 起動時にのみ必要で用済みになったら捨て去られる特別なセクション (init.text, init.data)を設けることができます. 廃棄されたセクションにはプログラムが存在しなくなるので廃棄後にデバッグすることはできなくなります(してはいけなくなります). デバッガが廃棄されたことを認識して, カーネルのデバッグ情報を無効として扱う必要があります.
- XIP カーネル
  XIP(eXecuite-In-Place)カーネルとは, その場で実行するように作られたカーネルです. アーキテクチャによってはカーネルの再構築メニューに選択オプションがあります. 通常の Linux カーネルはストレージディスクから RAM に転送されてから実行されますが, MaskROM や NOR 型 FlashROM 上から直接実行するようにリンクされていれば転送が必要ありません. カーネルは ROM 上に置かれたセクション(プログラムコードや初期値)と RAM に置かれたセクション(変数)になります.
  プログラムコードが ROM にあるため, ソフトウェアブレークポイントを設定できなくなります. ソフトウェアブレークの実現のためには, エミュレーションメモリの使用(高速)や FlashROM の書き換え(低速:あまり実用的ではありません)による方法が考えられます. いずれにせよ ICE によるサポートが必要です.

#	*プログラム種別*	*空間*	*アドレス*	*ページング*	*デバッグに必要なこと*
5.1	ブートローダ	非 MMU 空間	固定番地	無し	通常の ICE デバッグと同じ
5.2	Linux カーネル	MMU 上の単一カーネル空間	固定番地	(ほぼ)無し	通常の ICE デバッグと(ほぼ)同じ ・初期化セクション
5.3	ドライバモジュール	MMU 上の単一カーネル空間	リロケータブル	デマンドページング	・リロケーション対応(vmalloc エリア) ・ページング対応
5.4	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	・論理多重空間対応 ・リロケーション対応 ・ページング対応
5.5	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	・論理多重空間対応 ・リロケーション対応(プロセス毎にマッピングされるアドレスが異なる) ・ページング対応. ・実行コンテキストの認識
5.6	プロセス間共有メモリ	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	・論理多重空間対応 ・リロケーション対応 ・ページング対応

表 1 Linux における プログラムの分類

Linux 上のアプリケーションには MMU 等のハードウェアの機能を利用して高機能な管理機構やセキュリティが提供される.
カーネル空間のソフトウェアとユーザー空間のソフトウェアを透過的にデバッグするためには, デバッガがカーネル内のメモリ管理・タスク管理情報を実行時に取得することが必須になる.

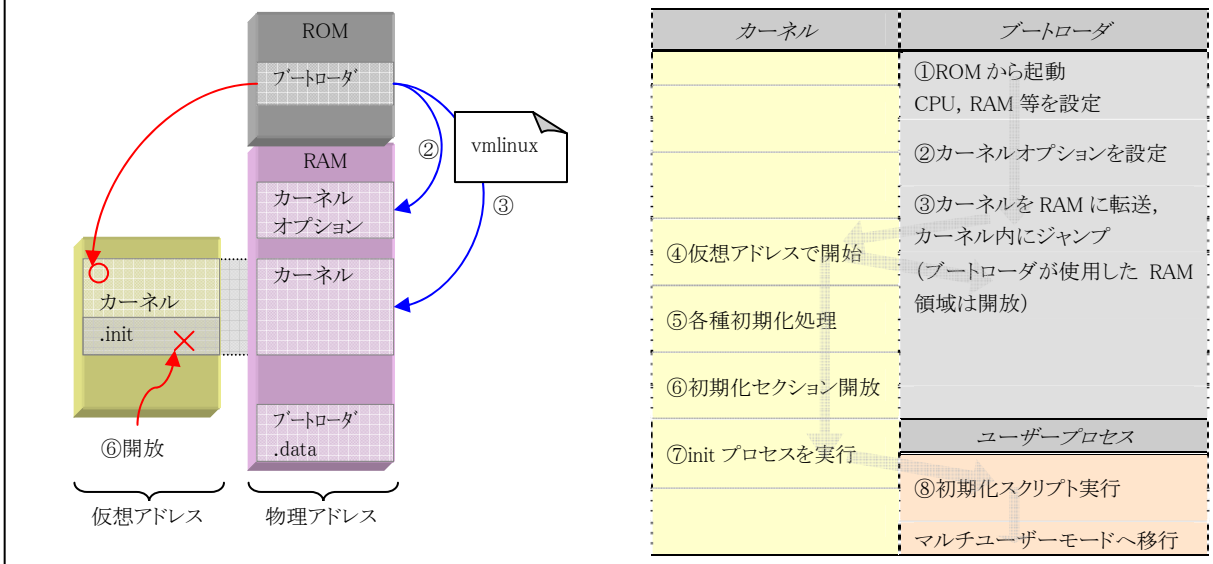


図 7　ブートローダからカーネルへ

ブートローダの処理中は物理アドレスで処理されている. 仮想アドレスにアクセスできるようになるのは TLB の初期化がすんでからになる. カーネルは仮想アドレスでリンクされている. これらの処理の過程でソフトウェアブレークが使用可能か, デバッガで RAM を読めるかといったデバッグ可能な条件が変化するが, 基本的には通常の組み込みのデバッグと同じだ.

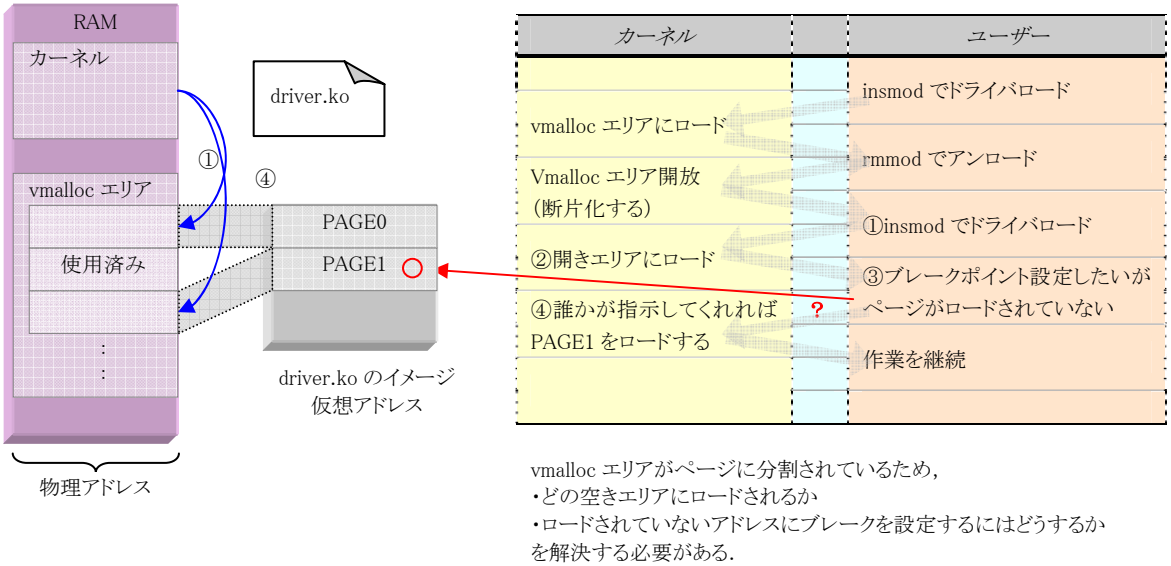


図 8　ドライバモジュール

Linux のドライバモジュールは insmod/rmmod コマンドによってシステム起動後にロードできる(ローダブル)なためアドレス未解決(リロケータブル)モジュールだ. ロードされた後はカーネルと同じ空間だが, ドライバがロードされる vmalloc エリアはページングが行われるため, ブレークポイント設定のためにはデバッガの対応が必要だ.

vmalloc エリアがページに分割されているため,
・どの空きエリアにロードされるか
・ロードされていないアドレスにブレークを設定するにはどうするか
を解決する必要がある.

5.3. ドライバモジュール

Linux ドライバモジュールはコンパイル・リンク時にアドレス解決されない，「リロケータブルモジュール」です．insmod(8) によってロードされた時にアドレス解決されます．アドレス解決された後はカーネルと同じ空間に配置されますが，配置される場所は固定ではないため rmmod(8) によってアンロードされた後に再度の insmod によって再ロードされても同じアドレスに配置されるとは限りません.デバッガは insmod(8) が実行される度にデバッグ情報とドライバの配置アドレスを対応付けしなおす必要があります．

ロードされた後はカーネル空間で動作しますが，vmalloc エリアと呼ばれるドライバモジュール用のメモリ領域に 4k バイト単位で配置され「ページング」が行われます．デバッグするためには「4k バイト以上離れた場所にブレークを張る」方法が必要になります．

また，ドライバモジュールにも初期化セクションがありますので初期化が済んだあとは vmalloc エリア内で開放・再利用されます．

5.4. アプリケーション

アプリケーションはユーザー空間で動作するように，MMU 上の多重仮想空間の固定番地でコンパイル・リンクされてファイルシステム上に置かれます．最も普通に作成・使用するプログラムです．ユーザー空間はプログラムに不具合があっても OS を安定動作させるための箱庭とも言え，ハードウェアから最も遠く抽象的なレイヤですので，デバッグするために解決するべき項目も多いです．

- 多重仮想空間への対応
  MMU を使用して多重仮想空間にプログラムを配置するということは，Linux ではユーザープロセス毎に別の仮想アドレスの割り当てを行うということを意味します.ユーザープロセスは図 4のように同一の仮想アドレスを使用していてもプロセス毎に別の空間として扱われます．デバッガは，プロセス ID と仮想アドレスの対応付けを行う必要があります．
- デマンドページングの対応
  Linux の仮想記憶方式は，メモリ保護を目的とした各プロセスの空間分離の他に，実 RAM サイズ以上のメモリを扱えるようにするページング（スワップ）機構を備えています．Intel x86 をはじめ多くの CPU では 4k バイト（Alpha AXP では 8k バイト）のページに分割されて RAM が使用され，ページそれぞれには，一意のページフレーム番号(Page Frame Number，PFN)が割り振られて管理されています．（RAM に空きがある状況でもページ単位で使用されます）
  ページングが行われる対象のプログラムは，カーネルによるページアウト操作によってプログラムの一部または全部が実メモリ上に存在していない可能性があります．デバッガがディスアセンブルを表示したり，メモリダンプを表示する際にはページアウトされているプログラム・データを読み込まなければ表示を行うことができません．
- マルチプロセス・スレッド対応
  子プロセスを生成するプログラムや複数のコマンドが協調動作するプログラムをデバッグするためにはデバッガが複数のプロセスを同時に扱えるようにならなければなりません.マルチスレッドの場合は，プログラムコードとグローバル変数が共有されますので，ブレークポイントを設定した場合に，同じプログラムコードを複数のスレッドで実行される可能性があります.プロセス単位（スレッド全部）が停止するのか，スレッド単位で停止するのか，またカーネルやドライバは停止する・しないといった扱い方の違いが考えられます．
  マルチプロセス・マルチスレッド共にひとつのコンテキストの停止中に他のコンテキストの実行を許す場合はデバッガで同時に複数の箇所をステップするということでもあります.機能の実現だけでなく，複数のコンテキストをどのように使用者に見せるか，というユーザーインターフェースも工夫する必要があります．
- XIP アプリケーション
  XIP アプリケーションは XIP カーネルと似た仕組みです．例えば，アプリケーションプログラムのプログラムコード(.text セクション)を RAM にロードせずに直接実行すれば起動が早くなり，RAM の消費も少なくなります．
  実装は，まず CE Linux Forum で配布されている Linear CRAMFS パッチ（文献[6]）が組み込みシステムでの実績があります．この手法は，ROM 上に置くためのファイルシステム CRAMFS に以下の修正を加えます．
  (1) 非圧縮にする（もとの CRAMFS は圧縮形式）
  (2) .text セクションを ROM に配置する
  この Lenear CRAMFS ファイルシステムを ROM または FlashROM に配置するようにすれば，プログラム起動時に RAM にプログラムコードをロードしなくても済みます．ROM デバイスに配置されるということはソフトウェアブレークポイントを設定できないことになりますので，デバッグするには工夫が必要になります．なお ptrace を使うデバッグの場合は，いったん RAM にページコピーされてからブレークされるようになっているため，実際の動きとは変わってしまいます．
  もうひとつ Carsten Otte 氏による XIP を実現する手法もあります．通常の Linux アプリケーションはファイルシステムに置かれた ELF 実行形式ファイルのイメージをページキャッシュに複製してから実行されます．Linux システムコール mmap(2) を使用してストレージディスク上のファイルをメモリ空間にマッピングしてから実行すればページキャッシュへの複製の手間と領域を節約できる，というものです．同じ XIP でもこちらの場合はブロックアクセスのストレージデバイス上のファイルシステムに置かれたプログラムファイルを使用する点でプログラムコード（.text）を ROM に配置する手法とは全く異なります．通常のアプリケーションとは起動までの動作が異なるため，同じ手順ではデバッグできません．この手法は，現時点では Linus Torvalds 氏管理のカーネルソースツリーには取り込まれていませんので一般的に使用されている機能ではありませんが，Andrew Morton 氏が管理しているツリーには取り込まれました（文献[7]）ので，将来採用される可能性はあります．

5.5. 共有ライブラリ

共有ライブラリは実行時にアプリケーションとリンクされてプロセスの一部として動作します.そのため，先に述べた多重仮想空間，デマンドページングの問題はプロセスと同様に解決する必要があります.

- リロケーション対応

  プログラムと実行時にリンクされるため，ライブラリのコンパイル時にはアドレス解決されていない「リロケータブルモジュール」になっています．LD_PRELOAD や LD_BIND_NOW 環境変数によってロードされるタイミングを意図的にずらされている場合を除いて，プログラムの実行開始直後にはアドレス解決されておらず，ld.so(8) によって使用時に解決されます.つまり，プログラムの開始直後でブレークした状態ではデバッグ情報をロード済みでも，共有ライブラリ内のコードにブレークポイントを設定することができないことになります.

  プログラムの実行に必要になるまでリンクされないためにブレークポイントを設定できず，動的リンクされた時には既にブレークしたかった箇所を通過してしまっていた，というのではデバッグできません．通常のデバッガユーザーの感覚ではデバッグ情報をロードしたら即ブレークポイントが設定できるべきです．動的リンクするタイミングをずらすことで実システムへの透過性を下げてでも操作性を優先した仕組みを作らなければなりません.

- マルチプロセスの対応

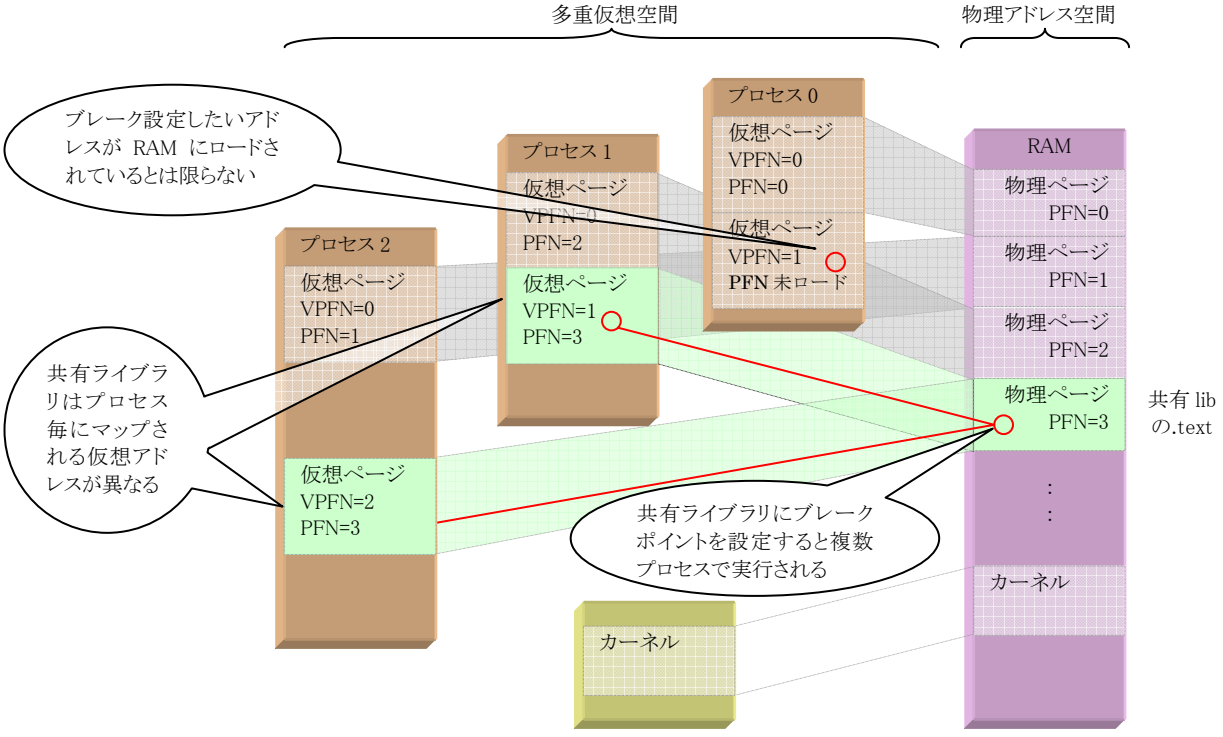
  文字通り各プロセスで共有されますが，可変データは各プロセスにコピーされます．例えば，C 標準ライブラリの errno 変数はプロセス毎に別の値になります．また，プログラム実行コード(.text)部分はプロセス毎にコピーされませんが，各プロセス毎に動的リンクされて配置される仮想アドレスは別になりますので，デバッガはプロセス毎にマッピングアドレスを認識してデバッグ情報との対応付けをしなければなりません.

  プログラムコード(.text)が共有されるということは，ブレークポイントを設定した箇所が複数のプロセスコンテキストで実行されるということです.どのコンテキストから実行された場合でも停止させるのか，デバッグ対象のプロセスからの実行の場合のみ停止させるのか，といった扱いや設定に対応するかどうかでデバッガの使い勝手がかなり変わります．デバッグ対象のプロセスではない場合にも停止するようにしてしまうと，恐らくデバッグ操作を続けることができませんし，ユーザーも混乱してしまいします.

  実行される仮想アドレスは異なりますが，RAM 上の実体としてはひとつです．ソフトウェアブレークポイントの場合は実行命令を書き換えますので，デバッグ対象ではないプロセスの動作を変えてしまわないように振舞わせるべきです．ハードウェアブレークポイントの設定はほとんどの CPU で物理アドレスを指定するようになっているのが普通です．デバッガが適切に設定を肩代わり

図 9 多重仮想空間・マルチプロセス・共有メモリ

もともとハードウェアのデバッグ用途 JTAG-ICE にとってはユーザー空間のプログラムをデバッグするためには解決しなければならない問題が多い．仮想アドレスと物理アドレスの対応させなければならないことはもとより，MMU を使用したメモリ保護が行われるため，それぞれのプロセスは別の仮想アドレス空間になっている．ページングや複数コンテキストによる実行も重要だ.



するような仕組みを用意しない限り，ユーザーが手計算でアドレスを算出して設定を行うのは非現実的な手間がかかります．また，ソフトウェアブレークポイントと同様に実行コンテキストの解決もデバッガが行わなければ，ユーザーはハードウェアブレークポイントで停止するたびにカレントプロセス ID を確認する手間がかかります．

5.6. プロセス間共有メモリ

Linux 環境上のソフトウェアの分類としてはもうひとつプロセス間共有メモリの存在も挙げておきます．共有ライブラリがプログラム実行コードをプロセス間で共有する仕組みなのに対し，プロセス間共有メモリは，複数のプロセス間で共通のメモリを介して情報を交換する仕組みです．単なるメモリ空間上のエリアではなくプロセス間通信（Inter Process Communication，IPC）のメカニズムです．Linux では Unix System V IPC Shared Memory 方式がサポートされています．

複数のプロセス間で同じ物理ページを共有するために，仮想記憶の機能が利用されています．プロセスから全てのメモリへのアクセスはページング機構を介して行われ，各プロセスはそれぞれ独立したページテーブルを持っています．ページテーブルのあるエントリ（仮想ページフレーム番号：VPFN）に対応する物理 PFN が同じ値になっているものが，複数のプロセスに存在すれば，それぞれのプロセスは同じ物理メモリページを共有していることになり，この状態を「共有仮想メモリ」と呼び，プロセス間共有メモリ機構に使われています．つまり，各プロセスから見えている仮想アドレスが異なっていても物理アドレスが同じ場所を指している状態が作られるということです．

さて，分類としては挙げましたが，共有メモリで扱うのは「データ」であってプログラムではないのでデバッガでどう扱うか，どう対応されているべきか考えるのは難しいものがあります．共有メモリ上にプログラムコードを置いて実行することはできないため，共有メモリ上にブレークポイントを設定することはありませんので，「対象外」とするのもひとつの考え方でしょう．強いてデバッグするとすれば，プロセス間のデータの流れを追いかけたい場合に共有メモリ内のデータにウォッチポイントを設定したいことがあると思います．しかしながら，共有メモリへのアクセスは読み・書き用の関数を使用するのが普通ですし，プロセス毎にアドレスが異なるためにデバッガでウォッチ機能を実現するのは困難です．Linux の場合はカーネルがオープンソースですので，カーネル自体をデバッグ用に改造するのが近道だと思われます．

Linux 対応の方法（PARTNER-Jet の場合）

Linux に対応するための課題を PARTNER-Jet ではどのようなアプローチで解決したのかを説明します．

6. 課題解決の方針

Linux に対応するためには，デバッガが必要な実行時の情報を補う必要があります．例えば，デバッガに読み込んだデバッグ情報に対応するプロセス ID や共有メモリのマッピングアドレスをユーザーに指定させる，というのもひとつの方法ですが，非常に煩雑な手順をデバッガの使用者に強いることになります．

これまでのところ ICE には実システムとの透過性を強く求められてきましたが，この問題の解決のアプローチとしては，「操作性←→実システムへの透過性」のトレードオフとして，実行の状態を多少操作してでも人にわかりやすいようにデバッガ側で OS との調整を行うべきだと考え，

- JTAG-ICE(PARTNER-Jet) が積極的にカーネルの内部情報を解析することで解決する
- デバッグ操作を簡単にするためにはカーネルなどにパッチを充てることもやむをえない

という方針を立てました．

先にも述べましたが，JTAG-ICE は CPU のスーパーバイザーモードで動作するカーネルさえも止めてメモリの読み書きを行うことができます．カーネルのデバッグ情報を持って「動作中のカーネルの気持ち」を知れば，カーネル上で動作しているプログラムの全てをデバッガで扱うことができるようになるという発想です．

ただし，この方針によって以下のような解決できること，便利な点，不便な点があります．

メリット・デメリット

×	デバッグするためにカーネル（vmlinux）のデバッグ情報が必須になる
○	JTAG-ICE がユーザー要求のバックグラウンドで補助処理を自動的に行うため，ユーザーがデバッグのために補助的な情報を指定しながら操作しなくてよい．
○	ターゲット上でデーモンプログラムを動かしたりしないため，システムの透過性が比較的高く保てる． また，ICE 本来の CPU を停止させるブレーク＆デバッグの手法を阻害しない．
△	バックグラウンド処理のために JTAG アクセスを行うため，JTAG アクセス速度の遅い CPU では快適な速度で実現するのがつらい． （CPU 毎の JTAG アクセスの違いがどういう点かについてはコラム1を参照してください．）

コラム 1　CPU と JTAG アクセスのスピード

JTAG は元々バウンダリ・スキャン（BST：Boundary Scan Test：境界走査試験）と呼ばれるチップ回路テスト用に開発された手法で，IEEE 1149.1 で標準化されていますが，デバッグ機能を実現するための機能拡張は CPU ベンダ各社で独自に行われており，仕様が異なっています．
Renesas SH シリーズの H-UDI や NEC VR シリーズ(MIPS)の N-Wire，MIPS Technorogies の EJTAG などがあります．EJTAG を使っている CPU では JTAG アクセス速度が遅いものが多いので，本稿の PARTNER-Jet の Linux 対応方法のように JTAG アクセスを増やす使い方とは相性が悪くなります．

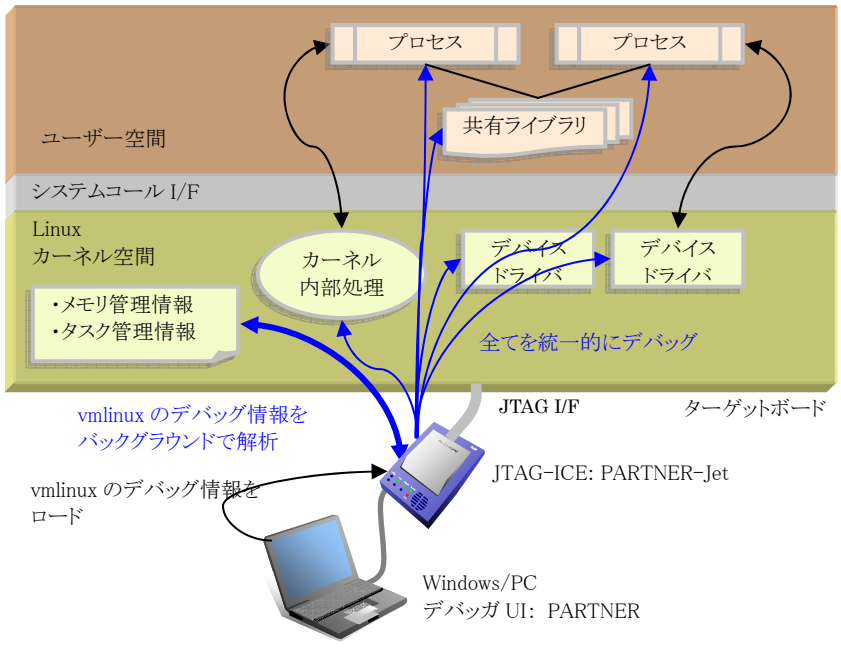


図 10　PARTNER-Jet の方法

PARTNER-Jet ではデバッグ中は常に vmlinux のデバッグ情報を使用して ICE ファームウェアがバックグラウンドでカーネルの動作を解析する．
その効果はユーザー空間のプログラムのデバッグ時に最も端的に発揮され，仮想多重空間のアドレス解決，対象プロセスの認識等が自動的に追従する．さらにトレースデータをカーネルと各プロセスで分離保存されて適切なバックトレースが実現する．

7. 課題解決の具体方法

7.1. デマンドページングの解決

ページアウトされているエリアにアクセスするための方法としては,力技でスワップファイルの中身を解析したり, 未ロードの命令をファイルから探す方法も考えられないことはないですが,強制的にページインをさせて確実メモリ上に配置された状態にするほうがシンプルです.「強制的にページイン」を起こすといっても,参照したいアドレスを read すれば,Linux カーネルが自動的にページイン処理を行ってくれます.

- ページインを発生させるために JTAG-ICE から仮読み(read)を行う
- 安全に参照するために vmlinux の内部情報を JTAG-ICE が解析する
- 参照に必要な最低限のページのみをページインさせればよい

JTAG-ICE によってカーネルが持つメモリ管理情報を解析し,ページインが必要だとわかると, JTAG-ICE は仮読みを行うためにメモリ上のプログラムの書き換えを行います. その状態で一瞬 CPU を動作させるとカーネルによってページイン処理が行われます. この間, ソフトウェアブレークポイントの操作と同様バックグラウンドで CPU やカーネルが動作したかどうかユーザーにはわかりません.

この手法は Linux カーネルのページング機構が正しく動作していることが前提になりますので, Linux カーネルのメモリ管理機能自体をデバッグすることができないですが,それ以外の場合ではとても快適に機能します.(図 11参照)

7.2. 多重空間への対応の解決

ICE が vmlinux の内部情報と CPU の TLB を解析して, 仮想アドレスと物理アドレスを変換して読み書き
トレースデータとプロセスを結びつける事が可能なトレースパケットの出力.

7.3. リロケーションの解決

ローダブルモジュールの初期化処理にソフトウェアブレークポイントを埋め込む事で, ロードアドレスを認識
vmlinux の内部情報を解析して, 共有ライブラリのマップ位置を特定する.

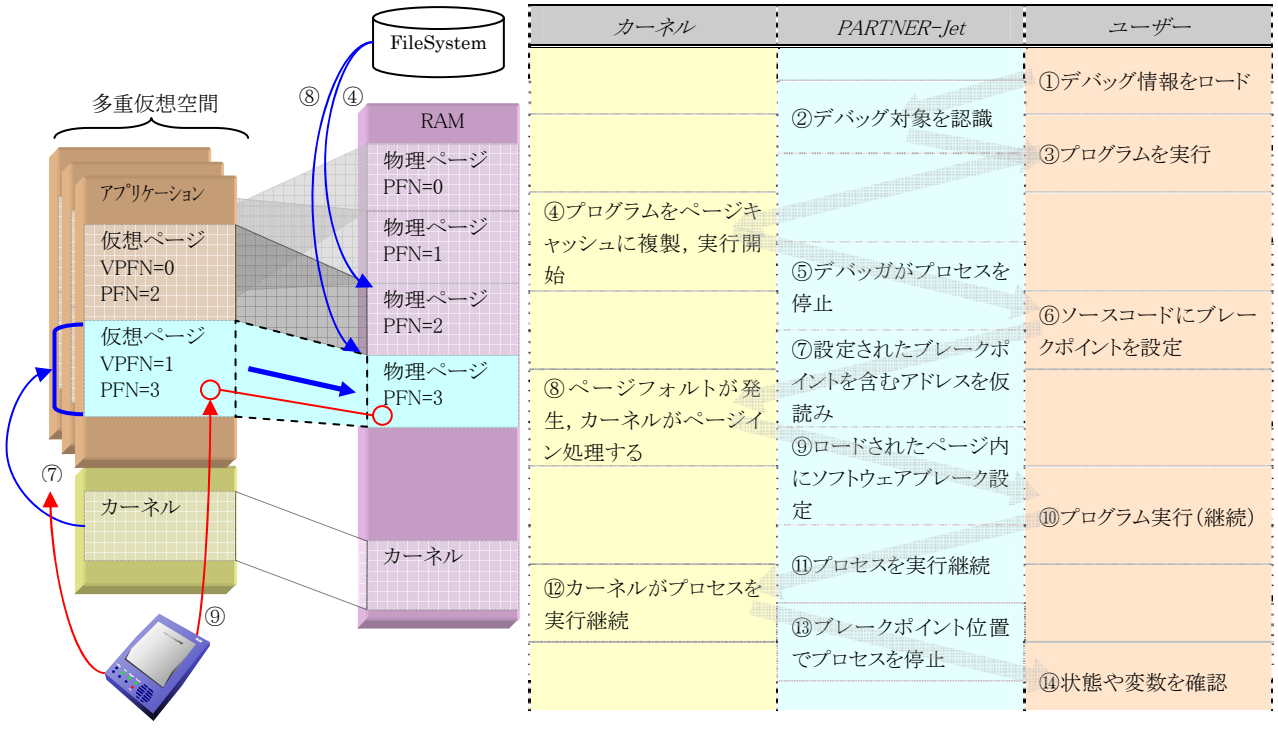
7.4. ユーザー空間とカーネル空間の認識

JTAG リアルタイムトレースは物理アドレスで実行番地と内容出力します. 履歴機能として利用するためには,アドレス空間の解決も必要ですが, カーネル空間とユーザー空間の切り替わりをデバッガが認識する必要があります.

PARTNER-Jet では Linux カーネルへパッチを当てることでコンテキストスイッチするたびに JTAG-ICE に通知する仕組みを Linux カーネルに持たせることにしました.

図 11 デマンドページング ～PARTNER-Jet の方法～

プログラムは常に RAM 上にあるとは限らない. PARTNER-Jet が介入することで自動的にページイン処理が行われる.
物理アドレスで設定しなければならないハードウェアブレークポイントの設定もユーザーにとっては簡単な作業になる.



8. プログラムの修正

8.1. カーネルへのパッチ

Linux カーネルに修正を加えなくても vmlinux のデバッグ情報を利用したデバッグ機能を使用することは可能ですが，カーネルソースの一部を修正することにより，より高機能（快適）なデバッグを行うことができるようになります．

Linux カーネルへのパッチを使用することで可能になることを以下にまとめます．

- ドライバモジュールデバッグの自動化
  ドライバモジュールのデバッグ情報は vmlinux ファイルに含まれないため，通常はデバッグ情報を手作業でデバッガに読み込ませる必要があります．
  カーネルに修正を加えると，ドライバモジュールがロードされた時点（insmod）で，自動的にブレークしてデバッグ情報を読み込み，すぐにデバッグが開始できるようになります．
- アプリケーションモードの対応
  JTAG-ICE を使用したデバッグでは，通常はブレーク時に CPU が停止します．それに対し，GDB 等のユーザーモードデバッガでは，カーネルや他のプロセスが動作したままデバッグ対象のプロセスのみを停止させます．
  カーネルに修正を加えることによって，JTAG-ICE を使用しながら，ユーザーモードデバッガのようにプロセス毎に停止を行うことができるようになります．このことをアプリケーションモードと呼んでいます．
- リアルタイムトレースのプロセス別表示
  カーネルに修正を加えることによって，リアルタイムトレースによる履歴をデバッガがプロセス毎に認識できるようになります．デバッガのユーザーインターフェースで，カーネルと，デバッグ対象プロセスの履歴を分けて表示することができるようになります．

Linux カーネルにパッチを適用すると，カーネル再構築時のメニューに機能追加項目が表示されるようになります．カーネルに対して加えている変更内容自体は，JTAG-ICE と Linux カーネルの連動のためには有用なもので，特に PARTNER-Jet のハードウェアに依存するようなものではありません．

しかし，ボードサポート用のパッチなどと異なり，アーキテクチャ共通部分のコードにも修正を加えていることや，パッチを適用したカーネルを JTAG-ICE を接続しない状態で起動できなくなるような修正が含まれていることから，今後も Linux カーネルのオリジナルソースツリーに取り込まれることは無いと思われます．

8.2. サポートファイル

ソフトウェアブレークポイントは**メモリ上の実行命令をデバッガが書き換えることで実現**しますが，複数のコンテキストで実行されるプログラムコードの書き換えを安全に行うためには，プログラムカウンタ退避用の領域が必要になります．この領域はプログラム中から参照されることはありませんが，デバッガでソフトウェアブレークをするときに使用されます．

サポートファイルは kmc-support.c ファイルにまとまられた極少量のコードです．アプリケーションプログラムにリンクすれば，マルチスレッド実行中のソフトウェアブレークポイントについて安全になります．同じファイルを共有ライブラリ中にいれておけば，複数プロセスから参照される共有ライブラリのプログラムコードにソフトウェアブレークポイントを安全に張ることができます．

8.3. ICE フックファイル

また，ICE フックファイルにはデバッグ開始用スタブが入っています．main()関数の先頭で呼び出すようにしておけば，デバッガがプロセスを認識することができ，アタッチ可能になります．

9. まとめ

PARTNER-Jet の Linux 対応を，その他のデバッガの対応状況と合わせて表 3にまとめてあります．

「PARTNER-Jet なら Linux システムを総合的にデバッグできる」ということがお分かりいただけると思います。

表 2 デバッグ用パッチ

vmlinux のデバッグ情報さえあれば，とりあえずの ICE としてのカーネルデバッグは可能．だが，快適に Linux のデバッグ作業を進めるためにはソフトウェアに修正を加える必要がある．カーネル用のパッチとアプリケーション用のサポートファイル（kmc-support.c）がある．

種別・名称	*対象*	*目的*	*備考*
カーネルパッチ	カーネル ソースコード	・コンテキストスイッチをデバッガが認識するため ・ユーザー空間のデバッグ	パッチを適用後，カーネル再構築メニューで有効化してビルドする．
サポートファイル （kmc-support.c）	glibc ライブラリ等	・実行中のアプリケーションへのアタッチを実現する ・共有ライブラリのデバッグを実現する ・ソフトウェアブレークポイントのための命令コードの書き換えを安全に行うため PC の退避などを行う	・サポートファイルを glibc に追加する ・glibc は殆どのプログラムにリンクされるので推奨
ICE フック ＆ デバッグスタブ （kmc.s）	アプリケーション ソースコード	・プロセス起動をデバッガが認識するため ・ソフトウェアブレークポイントのための命令コードの書き換えを安全に行うため	・アプリケーションソースの main()関数先頭でデバッガ用関数を呼び出す ・サポートファイルをアプリケーションにリンクする

表 3　デバッガ機能比較

デバッガにはそれぞれ特徴があり，得手不得手がある．PARTNER-Jet は Linux の機能をほぼ網羅しており，総合的なデバッグが可能だ．

		PARTNER-Jet	*一般的な ICE*	*gdb (ptrace)*	*kgdb*
デバッグ対象	カーネル	○	○	×	○
	モジュール	○	△※3	×	○
	アプリケーション	○	×	○	×
	特別なアプリ(init=プロセス ID1 番)	○	×	×	×
	XIP アプリケーション※1	○	×	△※2	×
	マルチプロセス	○	×	○	×
	マルチスレッド	○	×	○	×
デバッグ機能 (カーネル空間)	ソフトウェアブレーク	○	○	―	○
	ハードウェアブレーク(命令実行)	○	○	―	△※6
	ハードウェアブレーク(データアクセス)	○	○	―	△※6
	実行トレース(分岐トレース)	○	△※4	―	×
	ソースレベルデバッグ	○	○	―	○
	ローダブルモジュールの自動リロケーション	○	△※5	―	未調査
	デマンドページングの解決	○	×	―	○
	カーネルブレーク中のプロセスデバッグ	○	×	―	×
デバッグ機能 (ユーザー空間)	ソフトウェアブレーク	○	―	○	―
	ハードウェアブレーク(命令実行)	○	―	△※6	―
	ハードウェアブレーク(データアクセス)	○	―	△※6	―
	実行トレース(分岐トレース)	○	―	×	―
	ソースレベルデバッグ	○	―	○	―
	デマンドページングの解決	○	―	○	―
	共有ライブラリの自動リロケーション	○	―	○	―
	実行中プロセスへのアタッチ	○	―	○	―
	プロセスブレーク中のカーネルデバッグ	○	―	×	―
	プロセスブレーク中のカーネルや他のプロセスの実行	○※7	―	○	―

※1 XIP アプリケーションとは，アプリケーションの.text セクションがフラッシュメモリなどの ROM デバイス上に配置されるアプリケーション．
※2 カーネルに XIP アプリケーションデバッグの機能を実装すれば gdb でもデバッグ可能です．
※3 ICE にリロケーション機能の実装が必要です．またページアウトしている部分はデバッグ出来ません．
※4 ユーザー空間の実行トレースが存在すると，正しく解析できない可能性があります．
※5 デバッグ情報をリロケーションする機能が必要です．
※6 カーネルのデバッグ機能に,CPU 固有のハードウェアブレークポイント機能の実装が必要です．
※7 仮想 ICE 機能で実現します．

コラム 2　対応する Linux カーネルのバージョン

PARTNER-Jet では Linux カーネルの 2.4 系と 2.6 系に対応しています.

対応するカーネルのバージョンを記述するのには訳があります. PARTNER_-Jet は Linux カーネル(vmlinux)のデバッグ情報を利用してメモリ管理とタスク管理の情報を取得します. Linux 2.4 系では struct task_struct 型の構造体のリストの先頭を init_task シンボル名で参照してましたが, 2.6 系では child_reaper ポインタに統一(※1)されました. PARTNER-Jet では JTAG-ICE ファームウェアが Linux カーネル内のシンボル名を探し出してきて解析をするため, シンボル名や構造体の形式が変わるような変更があれば, デバッガも対応する必要があります. たとえば, task_struct のメンバ mm の位置やアライメントが変わっても問題ありませんが, メンバ名が変更になった場合は対応が必要になります.

このような重要なシンボル名の変更は滅多に行われるものではないと思いますが, メジャーなバージョンアップ時には変わるかもしれません. (※2)

※1

child_reaper ポインタ自体は 2.4 系にも存在していますが, マルチコア対応では init_tasks[cpu]配列の 0 番目に init_task のアドレスが格納されているなど, シンボル名としては init_task が使われていました.

※2

現在のカーネルの開発状況では Linux カーネルの 2.7 系は作られずに, 予定されていた変更が 2.6.12 以降に加えられていっているようです. バージョン番号が今までは a.b.c の形式でしたが, もし今後変わっていくのであれば, 2.6 系の中でも対応が必要になることがあるかもしれません.

PARTNER-Jet による Linux デバッグ

10. ICE デバッグのための準備と方法

ここでは，PARTNER-Jet を使用してデバッグを行う手順を示します．詳細はマニュアルに書かれていますので，概要を説明します．
Linux のデバッグをするためには，設定と手順が重要になります．どのような設定をするのか，実際にデバッグをするときに手順のイメージをつかむための参考としてください．

全体的な流れは，以下のようになります．

- ICE の設定（CFG ファイルの変更）
- カーネルに ICE 用フックエントリのパッチをあてる
- カーネルにデバッグオプションをつけてコンパイルする
- アプリケーションまたはライブラリにサポートファイルをリンクする

なお，デバッグの環境の構築例として図 12を参考にしてください．組み込みシステムの場合は,HDD・NAND-Flash・SD 等のカード・光ディスクドライブ,といったストレージの機器またはデバイスドライバの開発の段階でアプリケーションの開発も平行して行われることが少なくないと思います．図 12の例ではストレージ機器が存在していない場合の構築例ですが,カーネルやアプリケーションの変更後にターゲットへのインストール作業をせずに動作確認が行える点で,一般的に開発効率の高い構成だと思われます．

10.1. CFG ファイル設定

ハードウェアの設定をします．一度設定すれば基本的に設定変更せずに使い回しが可能です．
Linux 特有の注意事項としては「MAP セクションが Linux 対応になっている必要がある」という点です．

10.2. 起動オプション設定

デバッガソフトの設定（色やフォントなど），各個人毎に変更が必要な設定です．

- デバッグ情報バッファサイズ

使用している PC の搭載 RAM 容量によって限界はありますが，もしメモリ不足でデバッグ情報を読み込めない場合は増やしてみてください．

10.3. Linux 特有の起動オプション設定

- デバッグ情報モード

Linux の場合は ″GNU C （Linux etc）″ を指定します．

- OS モード

″-OS LINUX_ADD″が標準です．

- ソースパスの変換の設定

パス名の変換は特に重要です．デバッガ UI（PARTNER）は Windows 上で動作しますので，ソースコードデバッグを行う場合は Windows/PC で見える場所にソースファイルが置いてある必要があります．ところが，Linux カーネルや Linux 用アプリケーションは Linux システム上でクロスコンパイルされていますので，デバッグ情報には Linux システムでのパス名が書き込まれています．そのため，Linux で見たパス名と Windows から見たパス名の変換指定が必要になります．
実際の作業時は， Linux から Windows へソースファイルツリーをコピーするのは手間ですので，図 12のように Linux のファイルシステムを Samba で共有して Windows からみえるようにするのがよいと思われます．

- カーネルオプション

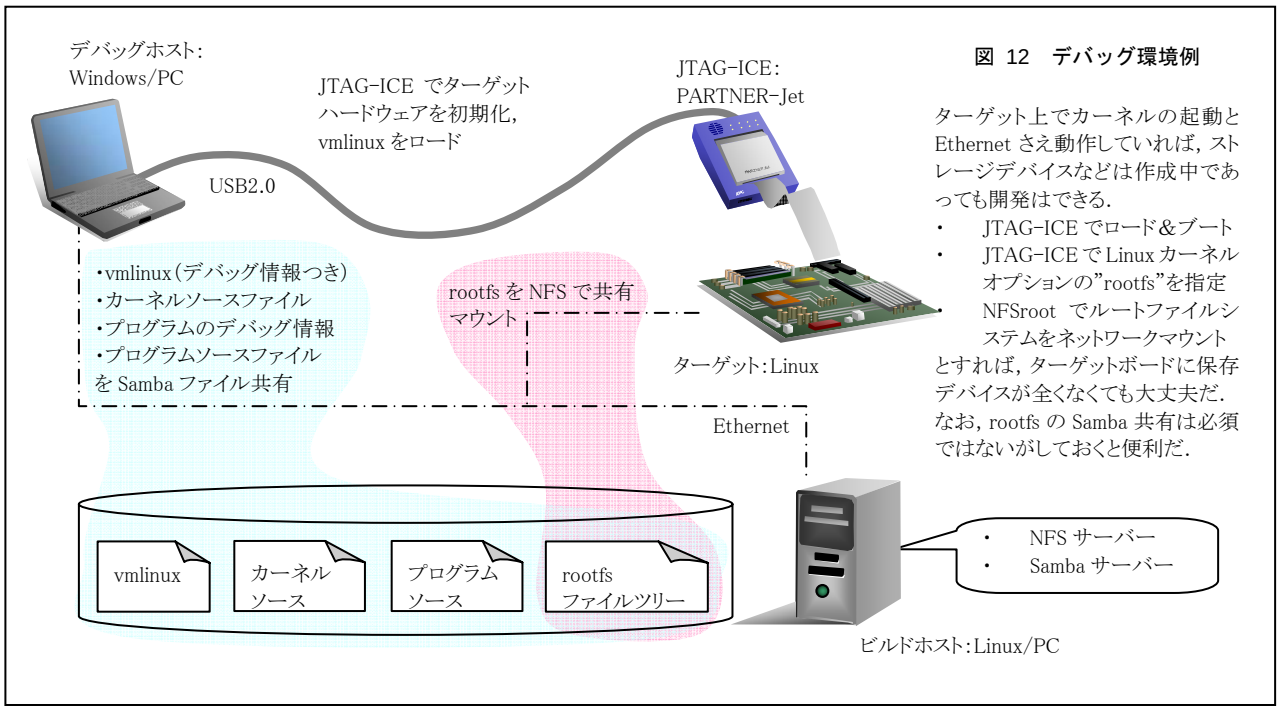


図 12 デバッグ環境例

ターゲット上でカーネルの起動と Ethernet さえ動作していれば，ストレージデバイスなどは作成中であっても開発はできる．

- JTAG-ICE でロード＆ブート
- JTAG-ICE で Linux カーネルオプションの”rootfs”を指定 NFSroot でルートファイルシステムをネットワークマウント

とすれば，ターゲットボードに保存デバイスが全くなくても大丈夫だ．
なお，rootfs の Samba 共有は必須ではないがしておくと便利だ．

Linux カーネルの起動パラメータなどを含む文字列を設定します. 通常はブートローダが設定するものを PARTNER-Jet が肩代わりします.

- ルートファイルシステムパス

Windows から見えるターゲットのルートファイルシステムを設定します.

- ローダブルモジュールパス

Windows から見えるローダブルモジュールのパスを設定します.

10.4. カーネルの変更

「ICE デバッグ用フックのパッチ」をあてます. 多人数での開発では誰かが一元管理してパッチを当てて置くのが良いでしょう.

全てのパッチは, カーネルコンフィグレーションで設定しないと有効になりません. (カーネルコンフィグレーションで簡単に切り離し可能です)

ソースを一度変更すれば, 特に変更する必要は無いので状況(リリース)に応じて, ICE フックの有効/無効を設定してビルドすればよいでしょう.

(ICE デバッグ用フックが入っているまま ICE 未接続で起動しても特に問題はありません)

11. カーネルのデバッグ

vmlinux にデバッグ情報がついていれば通常の組み込みデバッグと同様, 以下のようなことができます.

- カーネルの完全なソースデバッグ
- 実行トレース(AUD や ETM など)
- 割り込み処理のソースデバッグ
- 最適化コンパイルされたコードのソースデバッグ
- ブートローダ代替機能
- カーネルコマンドラインをデバッガから指定
- デバッガからカーネルをロードするだけで, Linux カーネルの起動が可能

ただし, XIP カーネルは方法が異なります.

12. XIP カーネルのデバッグ

XIP カーネルをデバッグする場合, ROM に置かれている場合は ROM のアドレスに対してはソフトウェアブレークポイントが使えません. エミュレーションメモリを利用するか, FlashROM を使用して PARTNER-Jet の FlashROM 書き換え機能を併用すればソフトウェアブレークポイントを使用できるようになります.

12.1. カーネルデバッグの方法(1)

カーネルをブートローダが起動する場合です. 実システムでの起動と同じ手順になります. ブートローダがカーネルを RAM に転送し, 制御をカーネルに移します.

1. あらかじめデバッグ情報だけを読み込んでおきます

```
> ls vmlinux
```

2. start_kernel にハードウェアブレークを設定します

```
> br start_kernel,ex
```

ブートローダが処理している間はカーネルのアドレスのソフトウェアブレークポイントは転送処理によって上書きされてしまうため, ハードウェアブレークを使用します. シンボル start_kernel に停止後は, 自由にデバッグ可能です.

12.2. カーネルデバッグの方法(2)

カーネルを ICE から転送する場合です.

1. ブートローダのプロンプトを表示
2. ターゲットを強制停止
3. カーネルファイルをロード

```
> l vmlinux
```

この時にデバッガがカーネルオプションを指定します. カーネルオプションは通常はブートローダで設定される文字列で, 特定の RAM 領域にコピーされた状態でカーネルの処理がかいしされます. PARTNER はデバッガオプション -!! で指定されるカーネルコマンドラインを, ブートローダと同じように RAM 領域に書き込みます. (デフォルトは各 CPU によって異なり、例えば SH4 CPU では empty_sero_pages シンボル).

start_kernel に停止後は, ソフトウェアブレークポイントを任意の場所に設定して, 自由にデバッグ可能です.

13. カーネルと実行トレース

実行トレースを使うと以下のことができます.

- カーネル内の実行を, 命令単位で解析可能
- 実行トレースとソースコードの連携
- 実行トレースを保存して, 後から解析する事も可能

なお, 各プロセス毎にトレースを分離するためには, PARTNER をアプリケーションモードで起動して, 少なくとも 2 つ以上の Window を開いて操作する必要があります. トレースをカーネル空間とユーザー空間で分離することは PARTNER をカーネルモードで起動していてもできます.

14. ドライバモジュールのデバッグ

本当はカーネルに静的リンクしてデバッグするのが一番簡単です. (カーネルのデバッグと全く同じになります)

モジュールとしてデバッグしたいときは PARTNER-Jet の Linux 対応機能を使用します.

- モジュール単位でデバッグする事も可能
- モジュールを ICE デバッグビルドする事で, 簡単にソースデバッグする事が可能
- 実行トレースの利用も可能

モジュールのソースデバッグを可能にするには以下の2つの手順が必要です.

- デバッグしたいモジュールでデバッグフックを有効にする
- ターゲット上で insmod を実行

これだけの手続きで, リロケーションアドレスはデバッガが自動解析してくれ, 自由にソフトウェアブレークポイントを設定してソースデバッグする事が可能になります.

モジュールがロードされたあとは, カーネルとモジュールの間をステップ実行が可能です.

rmmod で自動的にモジュールデバッグが終了します.

14.1. ドライバモジュールデバッグの方法

1. デバッグフックの埋め込み
ドライバの module_init()が存在するソースファイルの中で, ファイルの先頭(init.h の前)に `_KMC_MODULE_DEBUG' を define します.
デバッグビルドしたモジュールを, ターゲットで insmod すれば自動的にデバッガで停止してデバッグ可能になります.
ICE フック ON で ICE 未接続で起動した場合は, その時点でカーネルエラーになるので要注意です.

module_init()があるファイル(抜粋)

```
#define__KMC_MODULE_DEBUG

#include <linux/config.h>
#include <linux/string.h>
#include <linux/slab.h>
#include <linux/module.h>
#include <linux/init.h>

～中略～

static void __exit rd_cleanup (void)
{
～中略～
}

static int __init rd_init (void)
{
～中略～
}

module_init(rd_init);
module exit(rd cleanup);
```

図 13 モジュールデバッグ用のソース修正

デバイスドライバをデバッグする場合は静的にカーネルにリンクしてしまうのが最も手っ取り早い. しかし, モジュールにしておけば, 動的にロード&アンロードでき, 静的リンクには無い使い方ができるので, どうしてもモジュールの形式にしたいこともあるだろう.
モジュール形式のドライバをデバッグする場合はソースコードを修正してデバッグ用のフックを埋め込む必要があるという制限がつくが, フックの埋め込みをしてしまえば insmod 時に自動的にデバッガで停止してデバッグ可能になり, 作業に支障はなくなる.

linux/init.h(抜粋)

```
#if defined(CONFIG_KMC_MODULE_AUTO) && defined(__KMC_MODULE_DEBUG)
#include       <asm/kmc-ice.h>
__KMC_ICE_HOOK_MODULE();
#if defined(__KMC_MODULE_NAME)
static char __kmc_driver_name[] = __KMC_MODULE_NAME;
#else
static char __kmc_driver_name[] = __KMC_OBJ_NAME;
#endif
#define module_init(x) ¥
       int init_module(void) ¥
       { __kmc_module_debug_start(); return x(); } ¥
       static inline __init_module_func_t __init_module_inline(void) ¥
       { return x; }
#define module_exit(x) ¥
       void cleanup_module(void) ¥
       { x(); __kmc_module_debug_end(); } ¥
       static inline __cleanup_module_func_t __cleanup_module_inline(void) ¥
       { return x; }
#else
#define module_init(x) ¥
int init_module(void) __attribute__((alias(#x))); ¥
static inline __init_module_func_t __init_module_inline(void) ¥
{ return x; }
#define module_exit(x) ¥
void cleanup_module(void) __attribute__((alias(#x))); ¥
static inline __cleanup_module_func_t __cleanup_module_inline(void) ¥
{ return x; }
#endif
```

ICE フック ON 時のモジュール初期化、終了マクロ

ICE フック OFF 時のモジュール初期化、終了マクロ

15. ユーザー空間のデバッグ

PARTNER-Jet では，デバッグフックを入れる事で，簡単にソースデバッグが可能になります．シングルプロセス，マルチプロセス，マルチスレッドのデバッグが可能で，

- 共有ライブラリのデバッグも簡単
- デバッグ情報を読むだけで，リロケーションも自動解決
- 実行トレースをプロセス別に振り分け可能
- プロセス単位でハードウェアブレークポイントの利用
- mmap エリアのメモリ参照も可能

です．

デバッガでは，デバッグ情報だけを読むことがポイントです．

ITRON などの場合であれば，ターゲットオブジェクトも ICE からロードできましたが，Linux ではアプリケーションはファイルシステムを介して実行する必要があるので，ICE からターゲットにダウンロードは出来ません．

15.1. アプリケーションデバッグ（通常）

以下の手順で使用します．

- アプリケーション用デバッグウィンドウを立ち上げて，アプリケーションのデバッグ情報を読み込む
- ターゲットでアプリケーションを起動

自動的にアプリケーションが停止するので，後は自由にデバッグが可能です．

15.2. アプリケーションデバッグ（アタッチ）

gdb のようなユーザーモードデバッガと同様に，実行中のプロセスにデバッガからアタッチして途中からデバッグする事が可能です．

以下の手順で使用します．

- デバッグフックを glibc に入れる
  デバッグフックはアタッチ以外では全く実行されないので，常に入れておいても問題ありません．
- アタッチ時に指定スレッドのみにアタッチ（non_add モード）
- アタッチ時に存在する複数のスレッドも同時にアタッチ（add モード）
- デバッグ情報の自動読み込み
  （起動オプションの-RootDir 指定が有効な時）

アタッチ後は通常にデバッグ可能です．

15.3. アプリケーションとカーネルを同時にデバッグ

PARTNER の動作設定はカーネルモードを利用します．

OS デバッグモード指定は，’-OS LINUX’ または ’-OS LINUX_ADD’ を使用します．

これらの設定により，カーネルに停止している時でも，アプリケーションにブレークポイントを設定する事が可能になります．これによって，高度なリアルタイムトレース解析が可能になります．

- カーネルとプロセスをまたぐ解析
  リアルタイムトレースをカーネルとプロセス側をまたがってソースレベルで解析することができます．
  システムコール実行時，例外発生時などに重宝します．
- カーネル内からプロセスのバックトレースを見る
  システムコール実行中（プロセスはシステムコール終了待ち状態）や例外発生時など，カーネル内の処理を実行中にブレークしたときに，指定プロセスのトレースを参照できます．

15.4. アプリケーションデバッグの方法

1. ICE フック呼び出しの埋め込み

プロセスやスレッドなどの，’空間’の開始位置に ICE フックを入れます．

- main()の直後
- fork()の子プロセス側の戻り
- thread の本体（pthread_create()の第三引数）

2. ICE フックをリンクする

ICE フック本体の kmc.s をリンクします．あらかじめ，デバッグしたいアプリケーションのデバッグ情報を読み込んでおけば，アプリケーションが起動したときに，自動的にデバッガで停止します．

Linux のコマンドシェルを使わなくても PARTNER の psid コマンドでプロセスの実行状況は確認可能です．

ただし，フック ON のまま ICE 未接続でアプリケーション起動すると，プロセスが例外終了するので要注意です．

```
void __kmc_start_debuger(char *argv0)

*argv0：そのプロセスの名称（通常は argv[0]）
```

なお，main()関数より前の処理をデバッグしたい場合は，ld.so を使用した特殊な方法があります．

15.5. 共有ライブラリデバッグ

基本的にアプリケーションデバッグの準備がしてあれば大丈夫で，デバッグ情報は 2 種類の読み込み方法があります．

1. デバッグ情報ファイルを指定して追加読み込み

PARTNER の lsa コマンドを使用します．

```
> lsa <共有ライブラリパス名>
```

2. 共有ライブラリ名から自動検索

起動オプションに-RootDir 指定がある時に有効です．

```
> linux load_so
    プロセスの全ての共有ライブラリのデバッグ情報をロード
> linux load_so  <名称>
    指定した共有ライブラリのデバッグ情報をロード
    linux load_so libc-2.3.2.so
    linux load_so libc*
```

同一の共有ライブラリでも，プロセスが異なればアドレスが変化する可能性があるので注意が必要です．

共有ライブラリ内に設定するブレークポイントは，他のプロセスで参照される場合にも影響します．

main()と fork()があるファイル例(抜粋)

```
main(int argc, char *argv[])
{
    int i=1,j,k;
    char c;
    int pid;

    __kmc_start_debuger(argv[0]);  ← 親プロセス空間開始の先頭

if((i = fork()) != 0){
    i = 100;
    k = 1;
}else{
    __kmc_start_debuger(0);  ← 子プロセス空間開始の先頭(引数は 0)
    i = 200;
    k = 2;
}
```

図 14　マルチプロセスデバッグサンプル

マルチプロセスデバッグのサンプルとして，ここでは fork によって子プロセスを生成するプログラムを例にしている．多くの場合は，fork()の成功後には exec を呼び出して子プロセスで別のコマンドを実行するだろう．

ここでのポイントは ICE フックの関数を親プロセスの先頭と子プロセスの先頭の両方に入れておくことだ．このことによって，カーネルによってプロセスが生成され，制御が子プロセスに移ったところでデバッガによってブレークすることができる．

ただし，daemon() ライブラリ関数は特殊な動作になっているため，使用している場合は要注意だ．

一見ただの関数呼び出しに見えるが，このライブラリ関数では親プロセスが終了して，子プロセスとして関数を戻ってくる(内部で fork() されている)．したがって、daemon() の後にも ICE フックを入れる必要がある．

図 15　マルチスレッドデバッグサンプル

マルチスレッドのプログラミングは Linux 独自システムコールの clone を直接呼び出したりはせずに，POSIX 標準で規定された pthread ライブラリインタフェースを使用するのが一般的だ．(ソースコードの移植性も高くなる)

スレッド生成用 pthread_create 関数の仕様は fork 関数とは異なり，その場で実行コンテキストが変わるのではなく，関数の第3引数で指定された関数をスレッド開始エントリポイントとして制御が移されるようになっている．

pthread でスレッド生成する例(抜粋)

```
ret = pthread_create(&t[i], NULL, thread_body, (void *)&param[i]);

void * thread_body(void *arg) {  ← スレッド本体(上記第三引数)

__kmc_start_debuger(0);  ← スレッド開始の箇所に埋め込み

～略～
```

デバッガとスレッドの関係は 2 種類
add モード　一つのデバッガで複数のスレッドをデバッグ
non_add モード　一つのデバッガで一つのスレッドをデバッグ

そこでスレッド本体の入り口に ICE フック関数を埋め込む．

さて，スレッドのプログラミングモデルは主に3種類に分類される．

- ボス/ワーカーモデル
  同じ処理を行う多数のワーカースレッドをボススレッドが起動する．
- ピアモデル
  最初にスレッドは複数生成され，それぞれは独立した処理を行う．
- パイプラインモデル
  複雑なフィルタなど，長い処理を並行実行可能な単位に区切って流れ処理する．それぞれのスレッドは独立した処理を行う．

このうち，ボス/ワーカーモデルのワーカースレッドの場合は，スレッドのエントリポイントは同じ関数になるため，デバッグする際には，どのコンテキストで実行されているのか注意が必要だ．普通はボススレッドはひとつにすると思うが，複数のボススレッドがワーカースレッドを起動する場合は，スレッドエントリの ICE フックでブレークした時点でスレッドが起動された出自を理解していないとデバッグしにくいかもしれない．

もちろん，リエントラントな関数を複数のスレッドから呼び出していて，その関数内にブレークポイントを設定する場合も実行コンテキストに注意を払う必要がある．

スレッドのコンテキストを知るには pthread のハンドルで調べる他，PARTNER でブレークしたときに表示されるプロセス ID を見ることでも調べられる．なお，Linux のスレッドの実装は 2.4 系カーネルでは Linux Thread 版なのでプロセス ID と 1 対 1 に対応するが，2.6 系カーネルでは Linux Thread 版の場合と NPTL 版の場合があり．NPTL 版の環境では，プロセス ID ひとつに複数のスレッドが含まれるので，使用している環境によって見方を変える必要がある．

その他のデバッグに関するトピック

16. より使いやすく

デバッグという作業は人間が行うものですので，デバッガの機能にも「完成形」というものはなく，より使いやすく，様々な好みの人に対応できるように，いつまでも進化し続けるものだと思います．

ここでは，デバッグ上のテクニックや，最新の PARTNER デバッガで対応した機能をご紹介します．

16.1. 例外とハードウェアブレークポイント

TLB ミスで例外になるアドレスは，ハードウェアブレークポイントに引っかかりません．例えばプロセスの 0 番地アクセス(Null pointer exception)はハードウェアブレークポイントでは止まりません．ICE を使ってデバッグをするときに，例外ハンドラのアドレスに常にブレークポイントを設定しておくことで，プログラム内で例外が発生したときにすばやく要因を判断する，といった使い方をされている方も多いのではないかと思います．

同様のことを Linux のユーザープロセスに対して実現するには，カーネルのユーザー空間のフォルト処理にブレークポイントを常に設定しておき，ハードウェアブレークと例外でのブレークポイントの両方を使います．

16.2. 最適化コンパイルのデバッグを快適に

コンパイラの最適化を行うと，コンパイラによって処理の順序変更をされたり不要なコードが削除されたりするため，デバッガでステップ実行したときに C 言語のソースコード通りの順序に実行したように見えません(実際そうなので)．

最適化によって変更や削除されるのは，元の記述の論理を壊さない範囲にとどめられるため，C 言語で記述した論理が意図したとおり正しく動作しているかをステップ実行で確認するためには，最適化オプションを外してコンパイルするのが一番です．

しかし，Linux カーネルの最適化を外そうと思うとかなり大変です．リンクエラーの回避が必要だったりインラインアセンブリの変更など Linux カーネルのコーディングテクニック上の問題だったり，割り込みハンドラなど最適化をかけないと動作しないコードがあったりするためです．

そこで，テクニックを 2 つご紹介します．

- 対象ファイルのフラグ変更

GNU make のテクニックを使って，Makefile を修正することでデバッグしたいファイルだけを最適化なし(-O0)にする．

```
CFLAGS_xxxx.o := -O0
```

または

```
xxxx.o: CFLAGS := -O0
```

とすることで特定ファイルのみ CFLAGS を変更できます．

```
yyyy.o: CFLAGS :+= -DMY_DEFINES
```

というように，特定ファイルのみにコンパイルオプションの追加もできます．

ただし，対象のファイルがオプション変更できない時は使えません．

- PARTNER の最適化コンパイルデバッグ機能を使う

最適化によって削除された行はプログラムコードには現われません(アセンブラで見ると存在しないのがわかります)が，デバッグ情報の生成では，プログラムコードとソースコードの行との対応付けが行われます．削除された行は，対応する命令が出力されていないため，通常は関数の入り口などブロックの区切り位置を指し示しています．

そのため，デバッガでステップ実行すると，関数の入り口付近と現在の実行行を何度も行き来しているように表示されてしまい，デバッグ上みづらいものになります．これは，実際にプログラムカウンタが前後しているのではなく，デバッガがデバッグ情報に対応する行への移動を表現しようとして表示上前後しているようにみえるだけです．

PARTNER では，オプション '-optimize'を付加する事で，最適化によって削除された行（＝デバッグでステップ表示する必要がない）を認識することで，ある程度気持ちよくソースデバッグが可能です．

16.3. デバッグ情報の読み込みを自動化

デバッグ環境を図 12の例のように構築している場合，実運用時とは異なりルートファイルシステムをビルドホストに置くため，ストレージの容量をあまり気にせずに済みます．そこでアプリケーションやライブラリにデバッグ情報を付けておき，さらに Samba で共有してデバッグホストからも参照できるようにすれば，プログラム実行時に PARTNER が自動的に対象プロセスや共有ライブラリのデバッグ情報をロードしてくれるようになり，デバッグの作業効率が上がります．

16.4. prelink のデバッグも OK

組み込みシステムではアプリケーション起動速度を速めるために共有ライブラリのアドレス解決を事前に(リンク時に)行っておく prelink 技術を用いることがあります．prelink されている場合でも PARTNER で問題なくデバッグできます．

16.5. オープンソースと Linux

Linux カーネルがオープンソースのため，ブラックボックスとして扱わざるを得ない部分が無いという点で，デバッグのためにはとても大きな利点があります．

PARTNER-Jet が ここまで多岐にわたる機能に対応できたのもカーネルにパッチを当てることが出来たからです．

カーネル空間のデバイスドライバを作成することが多い組み込み分野ではこの利点は大きな意味を持つと思います．Linux でシステム開発される方もこの利点を生かして，ぜひ様々なアイデアを試してみていただきたいと思います．

コラム 3　Linux 対応機能開発の経緯よもやま話

PARTNER-Jet が Linux 対応をし始めたのは 2002～2003 年頃のことです.

それまでも京都マイクロコンピュータ社内では実験的に Linux 対応が行われていましたが, 本格的な市場からの要求として表れたのは携帯電話で Linux 採用端末の開発が決まったことでした. 当時は, JTAG-ICE で Linux システムをどこまで詳細にデバッグできるのかという問いに応えられたメーカはあまりありませんでした. カーネルとドライバモジュールのデバッグは, gcc のデバッグ情報に対応していればある程度可能ではありましたが, ユーザー空間へは踏み込めていなかったと思います.

Linux カーネルのデバッグ情報を利用して JTAG-ICE がバックグラウンドでカーネルの持つメモリ管理情報を解析するという,

- 既存の JTAG 以外に特別なハードを必要としない
- ターゲットデーモンが必要無い

という点で, より実システムとの透過性が高くて軽量な方式を PARTNER-Jet は独自に開拓し, ユーザー空間のデバッグへもいち早く対応してきたという自負があります.

当時, Emblix の開発環境ワーキンググループによる開発環境標準化仕様では, Linux ターゲットシステム上で動作する デバッグ・デーモンが I/F メモリ(/dev/ice デバイスドライバ)を経由して Linux カーネルが持つメモリ管理情報をデバッガに伝える方法が提案されていました. この仕様は,

- ターゲットハードウェアに I/F メモリが必要
- ICE は元々ターゲットの RAM を読み出せるにもかかわらず, デバイスドライバを経由してやりとりを行う
- I/F メモリのアクセスには通信パケットフォーマットが定められている
- GDB との連動も考慮されている

と, Linux の多重仮想空間の解決のためにリモートデバッギング機構を用意するというかなり重厚な構成になっていて, 現在でも業界標準とまで一般的ではありません.

なお, 現在 ptrace の後継として提案されている utrace は Plan9 のデバッグインタフェースの動作と似ているそうで, 将来のカーネルではクラッシュしたプロセスのデバッグ, リモートデバッグ, /proc 等を利用したデバッガへの情報伝達, といった Emblix で検討されていた内容の一部と似た機能が搭載されるようになるかもしれません(文献[4][5]).

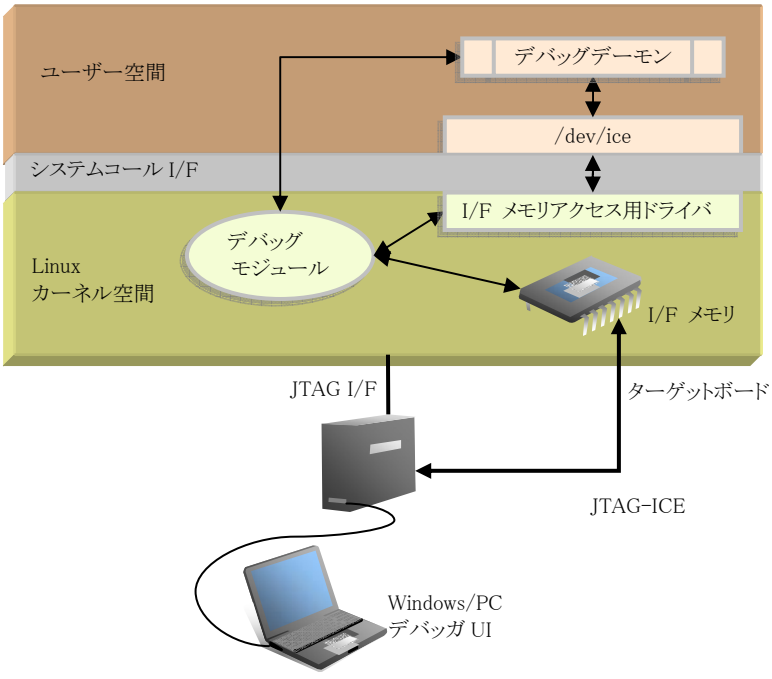


図 16　Emblix で提案された手法

ICE の Linux 対応を推進するために提案された手法で, ユーザー空間で動作するデバッグデーモンが/dev/ice デバイススペシャルファイル経由でカーネル内に用意するデバッグモジュールとやり取りをし, I/F メモリを介して ICE ファームウェアと通信をすることでカーネルの管理情報をデバッガと連携する.

ソフトウェア用デバッガ(ptrace)とハードウェア用デバッガ(ICE)の, それぞれのデバッガ実現手法を混在させたようなアプローチで, gdb との連携やリモートデバッギングまで見据えたような設計だが, ターゲットシステム上でデバッグのために使用するソフトウェア, ハードウェアが多いということは, 実システムへの透過性はかなり低くなってしまう.

参考文献

[1] KMC のサイト, JTAG デバッガ及びマニュアル等
http://www.kmckk.co.jp/

[2] ptrace(2) のマニュアル
http://www.linux.or.jp/JM/html/LDP_man-pages/man2/ptrace.2.html

[3] Roland McGrath 氏の Linux utrace
http://people.redhat.com/roland/utrace/

[4] David S. Miller 氏の utrace へのコメント
http://vger.kernel.org/~davem/cgi-bin/blog.cgi/2006/07/27

[5] LKML 過去記事「utrace vs. ptrace」
http://lkml.org/lkml/2006/7/13/42

[6] CE Linux Forum developers Wiki
技術文書, パッチがある
http://tree.celinuxforum.org/CelfPubWiki
Kernel XIP
http://tree.celinuxforum.org/CelfPubWiki/KernelXIP
Application XIP
http://tree.celinuxforum.org/CelfPubWiki/ApplicationXIP

[7] 2.6.12-rc5-mm1
Carsten Otte 氏の XIP アプリケーションパッチが取り込まれた
ftp://ftp.kernel.org/pub/linux/kernel/people/akpm/patches/2.6/2.6.12-rc5/2.6.12-rc5-mm1/announce.txt

[8] 日本エンベデッド・リナックス・コンソーシアム(Emblix)
技術ドキュメント
http://www.emblix.org/doc.html

著者

京都マイクロコンピュータ
辻邦彦

深瀬ソフトウェア研究所
深瀬茂寛