Embeded Linux
Embeded Linux
Embeded Linux

PARTNER for Linux

SH シリーズ

京都マイクロコンピュータ株式会社

PARTNER for Linux（SH シリーズ）マニュアル

この度は、Linux 対応ソースレベルデバッガ PARTNER をお買い上げいただき誠にありがとうございます。本製品は、快適なデバッグ環境を提供するために京都マイクロコンピュータ株式会社が開発、製造、販売している製品であり、大変有用なツールとして長く使用していただけるものと確信いたします。

はじめに

マニュアルについて

このマニュアルは、Linux カーネル、ローダブルモジュール、アプリケーションを弊社デバッガ PARTNER でデバッグするための手順、追加機能などが記述されています。

オンラインヘルプについて

Linux 対応 PARTNER には、画面上で機能や使い方を説明するオンラインヘルプが用意されています。
オンラインヘルプを表示するには、**[F1] キー** /**HELP コマンド**の入力、または [ヘルプ] メニュー→ [目次]/ ダイアログボックス上の [**ヘルプ**] **ボタン**の選択で行ったください。

目次

第 1 章　Linux アーキテクチャと PARTNER

Linux 対応 PARTNER は、従来の PARTNER デバッガに、組み込み Linux 用のデバッグ機能を拡張した高機能ソースレベルデバッガです。
この章では、Linux 対応 PARTNER の機能を応用することにより、組み込み Linux の開発において、あらたに実現できるデバッグ内容などを説明します。

1.1 Linux の開発環境と問題点

近年、組み込みシステムの複雑化 / 小型化 / 低価格化などにともない、次世代の組み込み機器用 OS として Linux に対する注目が高まってきています。
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、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ます。
さらに、提供されるソースコードのすべてがオープンソースであるということも加わり、Linux による組み込みシステムの開発は、リアルタイム OS(RTOS) などの従来の組み込みシステムに比べ、大幅な開発コストの削減につながります。
また、Linux は CPU の **MMU**(Memory Management Unit) を使用する OS でもあり、この MMU を利用することにより、仮想メモリ空間を生成し、各空間でのメモリ保護を行ったり、空間内でも割り当てをしていない領域にはメモリ保護を設定するなど、より堅牢なシステムを構築することができます。

しかしその一方で、現状の組み込み Linux の一般的な開発では、この MMU を使用することによる仮想メモリ空間 (論理多重空間) でのアドレス管理の複雑さなどにより、Linux カーネル / ローダブルモジュール (デバイスドライバ)/ アプリケーション (プロセス) ごとに対応するデバッガを使い分けることから生じる、一貫性に通じたデバッグが困難であるという欠点を持ち合わせています (表 1-1 参照)。

表 1-1 Linux 上で動作するソフトウェア

—	空間	実行アドレス	実行
(1) Linux カーネル	論理 / 物理一致の空間 (論理アドレスと物理アドレスが 1 対 1 でマッピングされる)	コンパイル / リンク時にアドレスが決定	ROM への書き込み、または ICE から転送を行い、指定のアドレスから直接起動
(2) ローダブルモジュール (デバイスドライバ)	論理 / 物理一致の空間 (論理アドレスと物理アドレスが 1 対 1 でマッピングされる)	insmod コマンドを用いた際に実行アドレスが決定	ファイルシステム上から Linux カーネルを介して起動
(3) アプリケーション (プロセス)	論理多重空間	コンパイル / リンク時にアドレスが決定	ファイルシステム上から Linux カーネルを介して起動

(1) Linux カーネル

コンパイル / リンクした際に配置アドレスが確定し、論理アドレスと物理アドレスが 1 対 1 でマッピングされ、指定したプログラムカウンタから実行を開始できるため、従来の組み込みシステム開発用の ICE とデバッガをそのまま使用して開発することが可能です。

(2) ローダブルモジュール (デバイスドライバ)

実行される空間は (1) と同様ですが、実際に配置されるアドレスは実行されるまで決定されない (リロケータブル) ということに特徴があります。
従来の組み込みシステム開発用の ICE とデバッガをそのまま使用した場合、実際には利用されていないアドレスを認識することとなり、正しいデバッグを行うことができない可能性があります。

(3) アプリケーション (プロセス)

一般的に、論理アドレスと物理アドレスが 1 対 1 ではなく 、また同じ論理アドレスが複数のプログラムで異なって利用されているため、これらを外から認識することが非常に困難とな り 、従来の組み込みシステム開発用の ICE とデバッガをそのまま使用することはできません。

また、既存のカーネルデバッガや ptrace() システムコールを利用したデバッグデーモンを使用した一般的な組み込み Linux のデバッグ環境 (図 1-1 参照) では、次のような問題点が挙げられます。

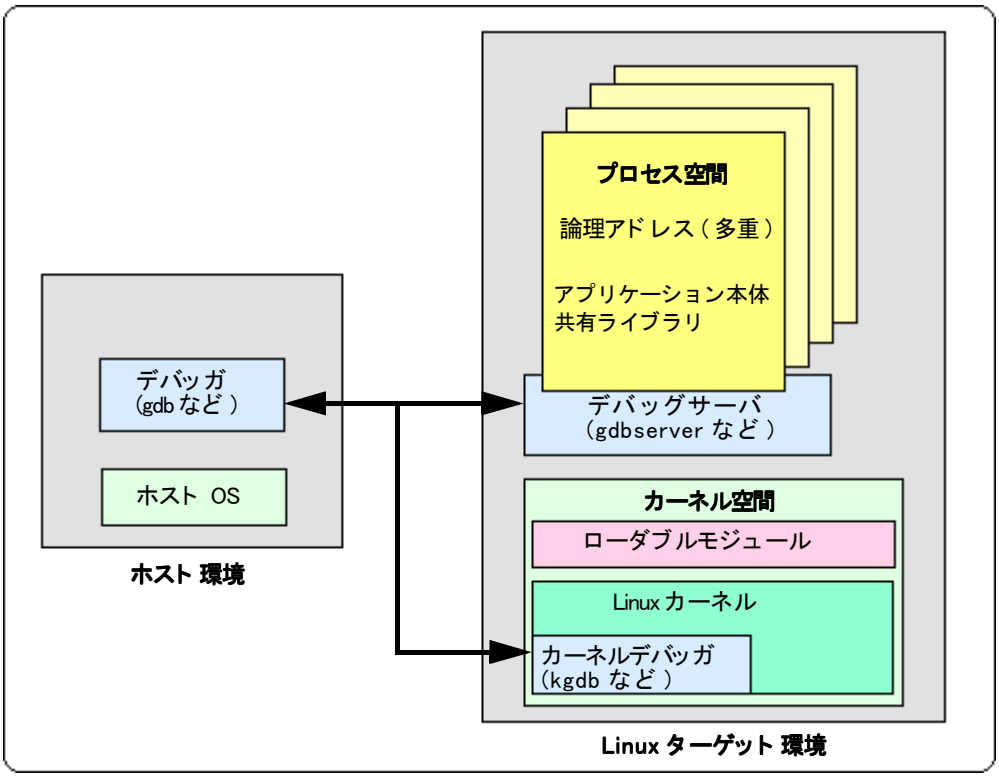


図 1-1 組み込み Linux の一般的なデバッグ環境例

● 一貫性を通したデバッグが困難

ターゲット上の Linux で動作するアプリケーションのデバッグは、ターゲット上で動作するデバッグサーバ (デバッグデーモン) とホスト PC で通信することにより行い、一方、カーネル / ローダブルモジュールのデバッグは、Linux カーネルに組み込んだカーネルデバッガとホスト PC で通信する手法が一般的です。したがって、カーネル / ローダブルモジュール / アプリケーションで利用するデバッガが異なるため、この手法のデバッグでは、アプリケーション実行中にカーネル / ローダブルモジュール内をステップイン実行することなどが不可能となります。

また、ターゲット上で動作するデバッグサーバは、ほとんどのデバッグ機能を Linux カーネルの ptrace() システムコールを利用することでを実現しています。つまり、デバッグサーバが機能するためには、Linux カーネルが動作している必要があります。

したがって、カーネルのデバッグ時にカーネルがブレークした場合、デバッグサーバが機能不能のためアプリケーションの状態 (変数など) を参照することができません。

● リアルタイムデバッグが困難

カーネルデバッガがホスト PC 上のデバッガと通信を行っている間は、割り込みを禁止した上でカーネルデバッガだけが動作します。したがって、割り込み処理のタイミングに関するデバッグは、実動作と変わる可能性があります。

また、アプリケーションをデバッグしているデバッグサーバにおいて、ブレークポイントによりプログラムが停止しても、カーネル / ローダブルモジュールの動作はリアルタイムに停止しません。

● 高機能なデバッグ支援機能が利用できない

ソフトウェアのみでデバッグ機能を実現するカーネルデバッガやデバッグサーバでは、次のような機能を使用することができません。

- 実時間での実行履歴（リアルタイムトレース機能など）
- ハードウェアブレークやトリガ条件などの設定
- プログラムの高速なダウンロード

1.2 PARTNER が実現する Linux 開発環境

この節では、Linux 対応 PARTNER が実現する Linux 開発環境として、次の項により説明します。

・PARTNER が可能にするデバッグ（5 頁）

・Linux 対応製品の概要（7 頁）

1.2.1 PARTNER が可能にするデバッグ

Linux 対応 PARTNER では、PARTNER-Jet とデバッガソフト PARTNER の機能拡張により、次の 2 つのデバッグモードを実現しています（それぞれのデバッグモードによる機能の詳細は「2.1 Linux と PARTNER のデバッグモード（12 頁）」を参照してください）。

【カーネルモードデバッグ】

PARTNER は、Linux カーネル / ローダブルモジュール / アプリケーションの異なるメモリ空間をすべて物理メモリで管理してます。したがって、アプリケーションからカーネルまでを完全に等価にデバッグすることができます。

【アプリケーションモードデバッグ】

アプリケーションの仮想メモリ空間だけを扱います。
PARTNER は論理アドレスだけを扱うため、マルチプロセス / マルチスレッドに完全に対応したデバッグを行うことができます。

これらの機能拡張により、PARTNER は次のデバッグを可能にします。

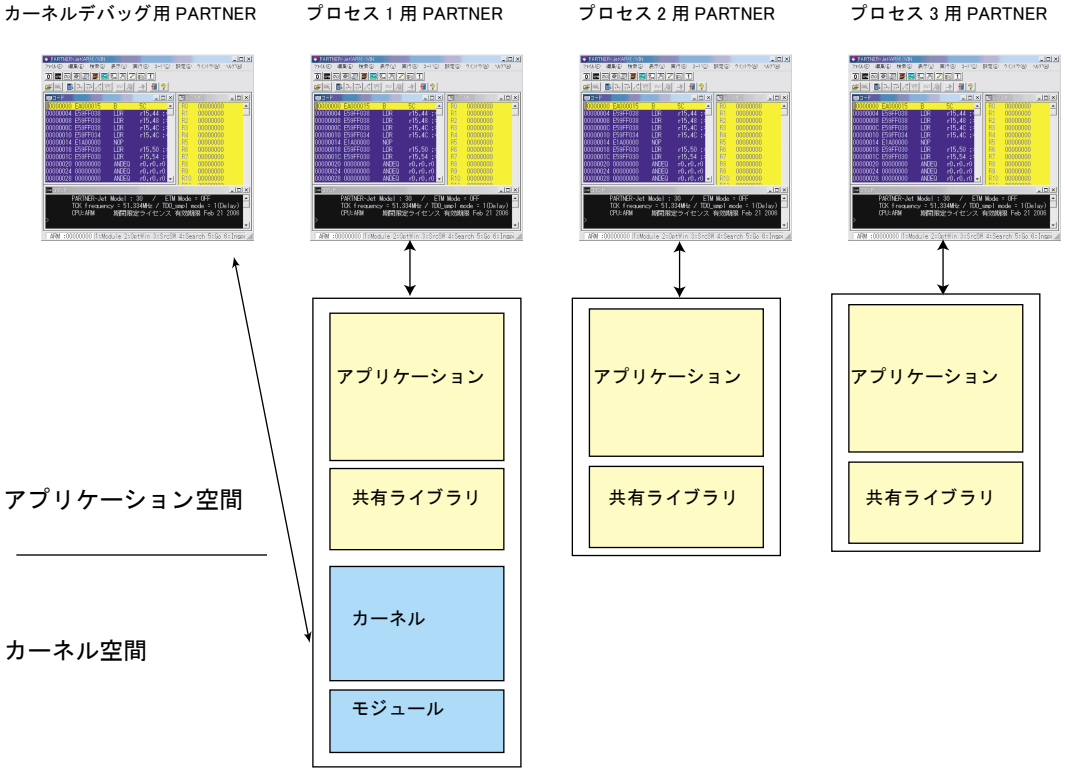
● 1 つのデバッガ (PARTNER) ですべてのデバッグが可能

Linux カーネル / ローダブルモジュール（デバイスドライバ）/ アプリケーションまでのすべてのメモリ空間を、PARTNER だけでデバッグすることが可能です。
したがって、アプリケーション部からカーネル / ローダブルモジュール内部の状態まで、一貫して問題を追いかけることができ、製品開発過程において、特に重要となるローダブルモジュール（デバイスドライバ）との連携を容易にデバッグすることができます。
また、カーネルやローダブルモジュールのブレーク時に、アプリケーションの特定箇所の状態（変数など）を参照することも可能となります。
このように、1 つのデバッガですべてのデバッグが可能となることにより、リアルタイム OS(RTOS) などの組み込みシステムで実現できていたデバッグ環境と同じように、Linux カーネル / ローダブルモジュール / アプリケーションまでを一貫性を持ってデバッグするができます。

● 複数のデバッガ (PARTNER) でのデバッグが可能

Linux カーネル / ローダブルモジュール、アプリケーションプロセス / スレッドをそれぞれ別の PARTNER ウインドウでデバッグすることが可能です。

図 1-2 マルチウインドウデバッグ



● リアルタイムデバッグの実現

アプリケーション部でブレークさせた場合、Linux カーネルやローダブルモジュール (デバイスドライバ) の動作も同時に停止します。(カーネルモードデバッグ時)
したがって、Linux カーネル / ローダブルモジュール / アプリケーション各部において、デバッグしたい状態の不一致がありません。

● マルチプロセス / マルチスレッド対応

仮想メモリ空間のすべてを物理メモリで制御することにより、アプリケーションのデバッグの際に、ptrace() システムコールの補助を必要としないため、マルチスレッドのデバッグが可能です。
また、アプリケーションデバッグモードでは仮想メモリ空間のみを扱うことができる機能を備えているため (「2.1.2 アプリケーションモードデバッグ (14 頁)」参照)、完全なマルチプロセス / マルチスレッドのデバッグが可能です。

● 高機能なデバッグ支援機能の適用

PARTNER ですべてのメモリ空間を管理することにより、ハードウェアブレークやリアルタイムトレースなどの高機能なデバッグが、Linux カーネル / ローダブルモジュール / アプリケーションのすべてに適用することができます。

● 高速転送の実現

Linux カーネルのダウンロードが数秒で完了します。

・2M ～ 4M Byte/Sec

【その他の特徴】

・デバッグ情報：stubs+ を始め、各種のデバッグ情報形式に直接対応

・USB2.0 対応

1.2.2 Linux 対応製品の概要

● PARTNER(高機能ソースレベルデバッガソフトウェア)

Linux 対応 PARTNER(Ver 3.5 以降) は、PARTNER-Jet 用のコントロールソフトに、Linux を組み込んだターゲットシステムをより快適にデバッグするための機能拡張を加えた高機能ソースレベルデバッガです。

● PARTNER-Jet(高速 JTAG ICE)

PARTNER-Jet は、Linux や T-Engine などの大規模組み込みシステムのデバッグ環境に最適の高速 JTAG ICE です。

PARTNER-Jetでは、MMUを用いた仮想メモリ空間および物理メモリ空間のすべてのソフトウェアを JTAG ICE で解決することによりデバッグを可能にしています。したがって、すべての空間において、リアルタイムトレースやハードウェアブレークポイントの利用が可能です。

また、アプリケーションの仮想メモリ空間のみを ICE で扱うデバッグモードが用意されており、マルチスレッド / マルチプロセスに完全に対応することが可能です。

1.3 動作環境

Linux 対応 PARTNER を使用して、Linux カーネル / ローダブルモジュール (デバイスドライバ)/ アプリケーションをデバッグするためには、次に示す環境を用意する必要があります。

1.3.1 ハードウェア環境

ハードウェア環境として、以下のハードウェアが必要です。

表 1-2 ハードウェア環境

ハードウェア	概　要
JTAG ICE	PARTNER-Jet
Windows PC	デバッガ (PARTNER) が動作するホストマシン
Linux PC	MIPS/ARM/SH などの Linux 環境をコンパイルするホストマシン
ターゲットボード	デバッグの対象となる Linux 環境のターゲットボード

1.3.2 ソフトウェア環境

ソフトウェア環境として、以下のソフトウェアが必要です。

表 1-3 ソフトウェア環境

ソフトウェア	概　要
PARTNER(Ver 3.5 以降)	デバッガソフトウェア 【注意】Ver3.5 未満の PARTNER は、Linux 対応のデバッグ環境をサポートしていません。Ver3.5 未満のソフトを既にインストールしている場合は、http://www.kmckk.co.jp のユーザサポートサイトより最新ソフトを入手してください。

1.3.3 システム構成

システム構成例を次に示します。

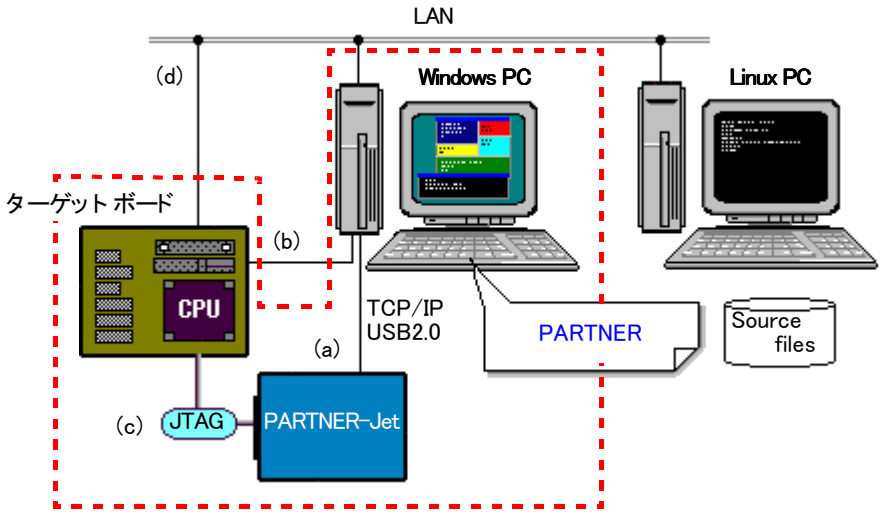


図 1-3 システム構成例

【図 1-3 の補足】

・赤い点線で囲まれた環境が、最も簡易な構成です。

・Linux PC の操作を Windows PC の telnet で接続し、ターゲットボードの操作を Windows PC のターミナルで接続することにより、すべての操作を Windows PC 上で行うことも可能です。

・Windows PC と PARTNER-Jet の接続は、USB もしくは LAN(Model30) をサポートしています。(a)。

・Windows PC とターゲットボードは、インターリンクケーブルなどで接続し、ターゲットのコンソールとしてターミナルを使用します (b)。

・ターゲットボードと PARTNER-Jet 本体は、JTAG コネクタにより JTAG 接続します (c)。

・ターゲットのルートファイルシステムを NFS などでマウントすることにより、より効率的なデバッグが可能となります (d)。

・ Linux PC のファイルシステムをを Samba で Windows PC にマウントすると効率的なデバッグが可能です。

・Linux PC 環境は、Windows PC 上に VMWare 等仮想 PC ソフトで構築することも可能です。

なお、これ以降、各環境へのコマンド入力の記述例は、次のとおりの意味とします。

WINPC>	: Windows PC の入力
LINUX86>	: Linux PC の入力
TGT>	: ターゲットのコンソール入力
PT>	: PARTNER のコマンドウィンドウへの入力

第 2 章　PARTNER の追加機能

この章では、Linux 対応 PARTNER に、新たに追加された機能とその設定方法について説明します。

2.1 Linux と PARTNER のデバッグモード

この節では、Linux 対応 PARTNER が持つ 2 つのデバッグモードについて、次の項により説明します。

・カーネルモードデバッグ（13 頁）
・アプリケーションモードデバッグ（14 頁）

2.1.1 カーネルモードデバッグ

カーネルモードデバッグは、Linux カーネル / ローダブルモジュール (デバイスドライバ) とアプリケーションの異なるメモリ空間を、PARTNER-Jet がすべて物理メモリで管理するデバッグモードです。
仮想メモリ空間を利用するプログラムであっても、プログラムを実行している際は、必ず物理メモリのどこかに存在しています。この論理アドレスと物理メモリの対応を PARTNER が自動的に行います。
したがって、PARTNER-Jet はすべてのプログラムを物理メモリベースでインターフェースすることとなり、Linux カーネルやローダブルモジュール (デバイスドライバ)、アプリケーション (プロセス) を分け隔てすることなく、同時に等価にデバッグすることが可能です。このモードでは、1 つのアプリケーションが停止 (ブレーク) すると、すべてのアプリケーションも停止します。
また、マルチプロセス / マルチスレッドに完全に対応して、ハードウェアブレークやリアルタイムトレースなどの高機能なデバッグを行うことができます。
なお、これらの機能は、特定の OS にまったく依存せずに PARTNER に実装されています。したがって、Linux カーネルにデバッグのための変更を加える必要はまったくありません。しかし、後述する『3.1 Linux カーネルソースの修正の必要性 (34 頁)』の変更を行うと、ローダブルモジュールやアプリケーションの PARTNER への自動アタッチなどが可能になり、開発効率が向上します。

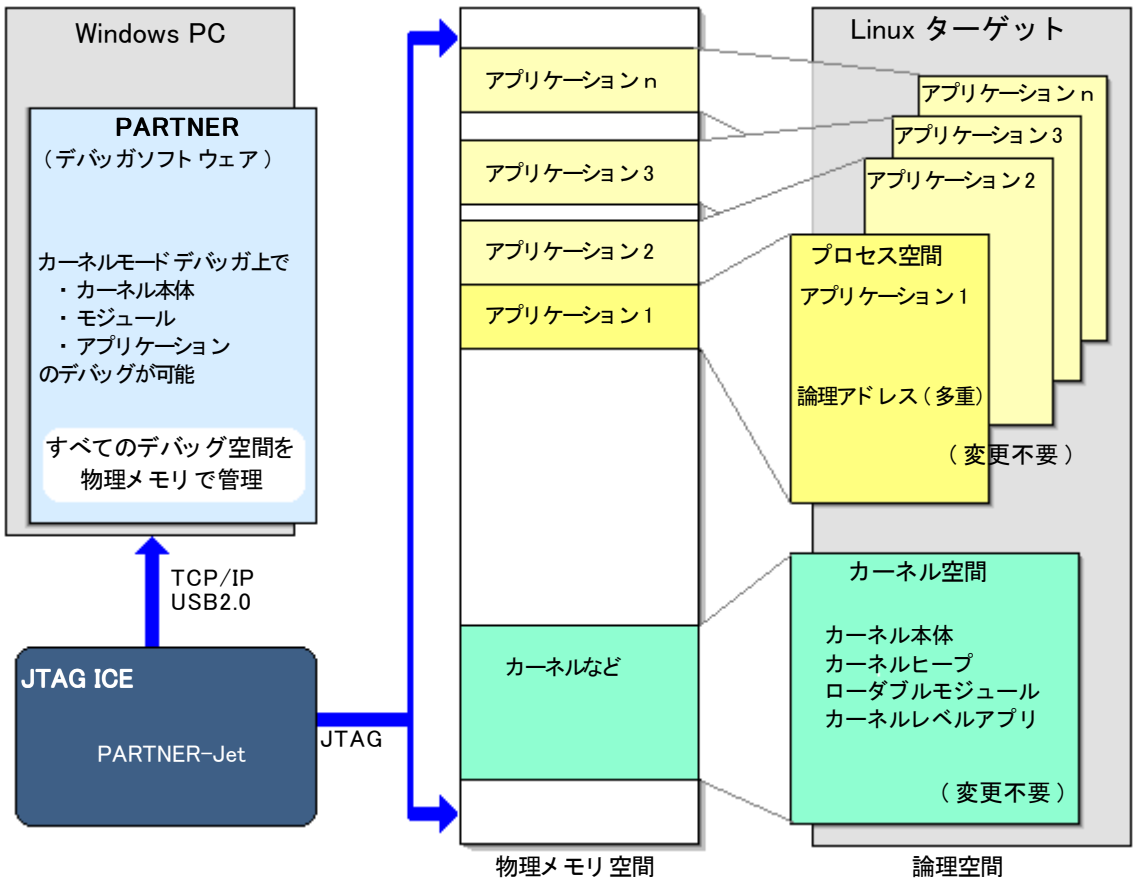


図 2-1 カーネルモードデバッグの概念

2.1.2 アプリケーションモードデバッグ

アプリケーションモードデバッグは、アプリケーション（プロセス）の仮想メモリ空間だけを扱うデバッグモードです。ただし、アプリケーションモードでも、Linux カーネル / ローダブルモジュール（デバイスドライバ）は、カーネルデバッグモードと同様に物理メモリで管理されデバッグを行います。
仮想メモリ空間を利用する OS では、一般的に、1 つのプロセスに 1 つの仮想メモリ空間が割り当てられます。この時、プロセスには配分された実行コンテキストも割り当てられるので、空間と合わせて 1 つの仮想マシンのように考えることができます。
アプリケーションモードデバッグでは、PARTNER-Jet とデバッガソフトが連携して、仮想メモリ空間ごとに仮想 ICE モジュールを生成します。この仮想 ICE モジュール上で、プロセスごとに起動させたデバッガを動作させることにより、そのデバッガは仮想メモリ空間だけがデバッグ対象となるデバッガになります。

この仮想 ICE モジュール方式の特徴は、仮想メモリ空間ごとに完全に独立したデバッグが可能になることです。
したがって、デバッグのためにプロセス 1 を停止（ブレーク）させて、変数を参照したりメモリを変更したりする際に、他のプロセスやカーネルを停止する必要はありません。この方式でデバッグを行うことにより、アプリケーション（プロセス）のデバッグ中に、通信がオーバフローすることなどを防ぐことができます。
また、マルチプロセス / マルチスレッドに完全に対応して、ハードウェアブレークやリアルタイムトレースなどの高機能なデバッグを行うことができます。

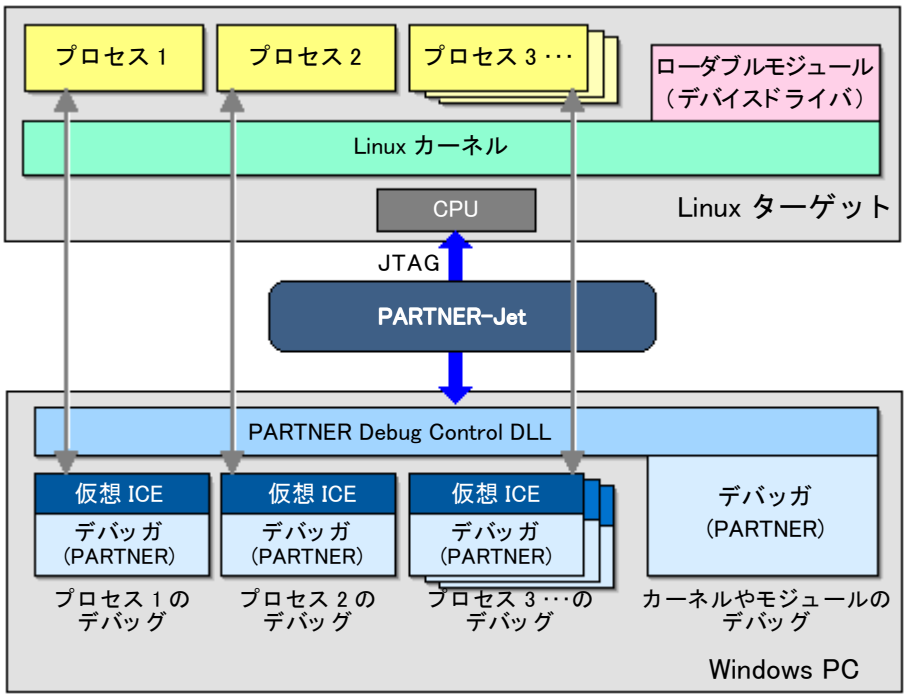


図 2-2 アプリケーションモードデバッグの概念

2.2 Linux デバッグのために追加された機能

Linux デバッグ用機能拡張のために、各種設定 / コマンドが追加、拡張されています。

2.2.1 CFG ファイルの拡張

MAP フィールド

Linux カーネルやローダブルモジュール、アプリケーションのデバッグを PARTNER で行うために、PARTNER からアクセスできるメモリ領域を指定する **MAP フィールド**が仕様変更になりました。
Linux のデバッグを行う場合には、論理アドレス領域のメモリマップと、物理アドレス領域のメモリマップを指定する必要があります。物理アドレス領域のメモリマップ指定は、00000000,FFFFFFFF を指定せずに存在する物理領域を指定してください。

書式 1 MAP 開始アドレス , 終了アドレス , 属性
書式 2 MAP 開始アドレス , 終了アドレス

書式 1 は、論理アドレス領域とその領域の属性を指定します。

表 2-1 属性の指定

属性	解説
APPLI	プロセスや共有ライブラリが配置される領域
KERNEL	Linux カーネルが配置される領域
MODULE	ローダブルモジュールが配置される領域

各属性は以下のように調べます。

● **APPLI 属性領域の確認**

ターゲットシステム上で、/proc/1/maps を参照して決定します。
MAP フィールドで論理アドレス領域が指定されていない場合でも、PARTNER でカーネルをロードして実行できる場合は、**MAPS コマンド**(28 ページ参照)で参照してアプリケーション領域を決定することが出来ます。

```
TGT>cat /proc/1/maps ↓
00400000-00408000 r-xp 00000000 00:07 6150       /sbin/init
00417000-00418000 rw-p 00007000 00:07 6150       /sbin/init
00418000-0041c000 rwxp 00000000 00:00 0
29556000-2956b000 r-xp 00000000 00:07 1605       /lib/ld-2.2.5.so
2957a000-2957b000 rw-p 00014000 00:07 1605       /lib/ld-2.2.5.so
2957b000-2969d000 r-xp 00000000 00:07 1612       /lib/libc-2.2.5.so
2969d000-296ab000 ---p 00122000 00:07 1612       /lib/libc-2.2.5.so
296ab000-296b1000 rw-p 00120000 00:07 1612       /lib/libc-2.2.5.so
296b1000-296b5000 rw-p 00000000 00:00 0
7bfff000-7c000000 rwxp 00000000 00:00 0
TGT>
```

● KERNEL 属性領域の確認

Linux カーネルをビルドしたときに出力される System.map ファイルやカーネルビルド用リンクスクリプトファイルなどを参照して決定します。

● MODULE 属性領域の確認

Linux カーネルソースで確認します。
include/asm/pgtable.hで宣言されているVM_ALLOC_STARTからVMALLOC_ENDまでの範囲で決定します。

【環境設定ファイル内の MAP 指定例】

```
MAP     00400000,7fffffff,APPLI
MAP     8c000000,8fffffff,KERNEL
MAP     c0000000,cfffffff,MODULE
MAP     00000000,003fffff
MAP     80000000,83ffffff
MAP     88000000,8bffffff
MAP     8c000000,8fffffff
MAP     90000000,93ffffff
MAP     94000000,97ffffff
MAP     a0000000,a3ffffff
MAP     a8000000,abffffff
MAP     ac000000,affffffff
MAP     b0000000,b3ffffff
MAP     b4000000,b7ffffff
MAP     f0000000,ffffffff
```

SH CPU は、0x80000000~0x9FFFFFFF の空間と 0xA0000000~BFFFFFFF の空間が同一のメモリ空間を参照しています。したがって、必ずその二つの空間の MAP 指定を正確に設定してください。

上記の MAP の指定を追加することにより、PARTNER は、MMU を通してアクセスする空間とダイレクトにアクセスする空間を区別します。

さらに、デバッガからの MMU 空間アクセスは、アタッチされたローダブルモジュール (ドライバ) やアプリケーションでアクセスエラーが起こらないことが確定している空間 (デバッガが自動判別) のみが許可されます。したがって、それ以外の空間をデバッガはアクセスすることができません。
このため、デバッガにローダブルモジュールやアプリケーションがアタッチされているか、アタッチされていないかで PARTNER の振る舞いは次のように異なります。

【アタッチされていない場合の、モジュール、アプリケーション空間の処理】

- ブレークポイントには、自動的にハードウェアブレークポイントが設定されます。
  したがって、ハードウェアブレーク設定可能数以内でのみブレークポイントが設定可能です。
- 参照したいメモリ空間が、MMU にヒットしていない場合は参照できません。
- アプリケーションは、同一の仮想メモリ空間で複数実行されます。このため、目的のブレークポイント以外でブレークしてしまう可能性があります (この場合は、[F5] キーにより目的のブレークポイントまでプログラムを続行させてください)。

【アタッチされている場合の、モジュール、アプリケーション空間の処理】

- ブレークポイントには、ソフトウェアブレークポイントが設定されます (ブレークポイントの設定数に制限がありません)。
- 参照したいメモリ空間が MMU にヒットしていない場合でも、MMU 自動ヒット機能で参照できます。
- ソフトウェアブレークを使用するため、特定のアプリケーションのみでブレークします。

2.2.2 起動オプション

Linux デバッグをサポートするため以下の起動オプションが追加 / 拡張されました。

-OS オプション

-OS〈デバッグモード〉

カーネルモード、アプリケーションモード、ADD モードを指定します。

表 2-2 デバッグモード

デバッグモード	解説
LINUX	カーネルモードを指定します。
LINUX_ADD	カーネル ADD モードを指定します。
LINUX_APP	アプリケーションモードを指定します。
LINUX_APP_ADD	アプリケーション ADD モードを指定します。

表 2-3 各デバッグモードの挙動

	マルチスレッドデバッグの方式	アプリケーションブレーク時の挙動
カーネルモード	一つのデバッガウインドウで、一つのスレッドをデバッグ	CPU が停止
カーネル ADD モード	一つのデバッガウインドウで、複数のスレッドをデバッグ	CPU が停止
アプリケーションモード	一つのデバッガウインドウで、一つのスレッドをデバッグ	デバッグ対象スレッドだけが停止（カーネルや他のスレッドは実行）
アプリケーション ADD モード	一つのデバッガウインドウで、複数のスレッドをデバッグ	デバッグ対象スレッドだけが停止（カーネルや他のスレッドは実行）

-XGX オプション

-XGX <変換前 PATH>,<変換後 PATH>

デバッグ情報モードを GNU C(stab, stab+, dwarf, dwarf-2) モードで PATH 情報付きにします。
また、ロードするファイルの PATH 情報を変換できる機構があり、Samba などでマウントされた PATH に変換することが出来ます。
このオプションは最大 10 個まで指定できます。

【使用例】

カーネルビルド時のPATHが/home/foo/work/linuxで、Z:ドライブに/home/foo/workをマウントした場合

-XGX/home/foo/work/linux/,Z:¥linux¥

カーネルビルド時の PATH が /home/foo/work/linux で、c:¥work¥kernel¥linux にコピーしている場合

-XGX/home/foo/work/linux/,C:¥work¥kernel¥linux¥

-!v オプション

PARTNER ウインドウを複数起動する場合に指定する必要があります。
-MULTI オプション (20 ページ参照) や **MULTI コマンド** (22 ページ参照) で PARTNER を起動する場合は、自動的に付加されます。

-!! オプション

-!! < 初期 PC 値 >,< カーネルオプションシンボル名 >=< カーネルオプション文字列 >

PC 初期値とカーネルオプションを指定できます。
< 初期 PC 値 > は省略できます。< 初期 PC 値 > を省略した場合は、ダウンロードしたファイルに指定してある PC 初期値が有効になります。
< カーネルオプション文字列 > は、カーネルオブジェクト (vmlinux) のロード時、< カーネルオプションシンボル名 > で指定したメモリに文字列を書き込みます。
< カーネルオプションシンボル名 > は、グローバル宣言された初期値付き char 配列で、カーネルソース内でカーネルオプションのデフォルト値として使用されている必要があります。
< カーネルオプションシンボル名 > を省略した場合は、empty_zero_page に 0x100 を加えたアドレスになります。

【使用例】

```
-!!,init_cmdline="mem=32M console=ttyS1,119200"
```

-SK オプション

-SK < ローダブルモジュール PATH>

ローダブルモジュールの PATH 情報を指定します。
カーネルソースツリー内のローダブルモジュールをデバッグする場合は不要です。

【使用例】

ローダブルモジュールビルドのディレクトリが /home/foo/work/modules で Z: ドライブに /home/foo/work をマウントした場合

```
-SKZ:¥modules
```

-MULTI オプション

-MULTI < 起動ウインドウ数 >

PARTNER ウインドウを指定個数起動します。指定可能なウインドウ数は 16 までです。

-OPTIMIZE オプション

オプティマイズ有り (-O2 等) でコンパイルされたオブジェクトをデバッグするときにユーザに判断しやすいように行番号情報を補正します。

-EUC オプション

コードウインドウに表示されるソースファイルの文字コードを SJIS から EUC に変更します。

2.2.3 追加コマンド

Linux デバッグをサポートするため、以下のコマンドが追加 / 拡張されました。

- MULTI(複数 PARTNER ウインドウの起動)（22 頁）
- Q,EXIT(終了)（23 頁）
- INSMOD(ローダブルモジュールアタッチ情報)（24 頁）
- PSID(プロセスアタッチ情報)（25 頁）
- THREAD(スレッドアタッチ情報)（26 頁）
- PS(プロセス情報)（27 頁）
- MAPS(メモリ情報)（28 頁）
- ATTACH(プロセスのアタッチ)（29 頁）
- G(実行)（30 頁）
- INS(インスペクト)（31 頁）
- SNAME(ソースパス表示)（32 頁）

MULTI(複数 PARTNER ウインドウの起動)

書式

MULTI < 起動 PARTNER 数 >

機能

PARTNER ウインドウの複数起動

解説

このコマンドは、指定された < **起動 PARTNER 数** > の PARTNER ウインドウを起動します。最大起動可能数は 16 個です。

起動オプション **-MULTI** で PARTNER 起動時に同様の動作をさせることが可能です。

【使用例】

```
>multi 3 ↓
>
```

【参考】

-MULTI オプション（20 頁）

Q,EXIT(終了)

書式 1

Q/A

書式 2

EXIT/A

機能

すべての PARTNER ウインドウの終了

解説

Q または EXIT コマンドに /A オプションをつけると、MULTI コマンドや -MULTI オプションで起動した複数の PARTNER ウインドウをすべて閉じることが出来ます。

【使用例】

>q/a ↓

【参考】

MULTI(複数 PARTNER ウインドウの起動)（22 頁）,-MULTI オプション（20 頁）

INSMOD(ローダブルモジュールアタッチ情報)

書式

INSMOD

機能

ローダブルモジュール情報の表示

解説

現在 PARTNER にアタッチされているローダブルモジュールのメモリ範囲を表示します。

【使用例】

```
>insmod ↓
INSMOD AREA : C1800000-C1801FFF
>
```

PSID(プロセスアタッチ情報)

書式 1

PSID

書式 2

PSID ADD

書式 3

PSID NON_ADD

機能

アプリケーションの情報 / モードの表示

解説

書式 1 はカレント PARTNER にアタッチされているアプリケーションのプロセス ID と使用メモリ範囲を表示します。ADD モードの場合はアタッチされているすべでのプロセス ID も表示します。

書式 2 はカレントの PARTNER を ADD モードにします。(起動オプションで -OS LINUX_ADD もしくは -OS LINUX_APP_ADD を付けて起動した時と同じ状態)

書式 3 は書式 2 の反対動作で、カレントの PARTNER の ADD モードを解除します。

【使用例】

```
>psid ↓
PSID SET 97(0x61)  CURRENT -1(0xFFFFFFFF) [ADD MODE]
APPLI. AREA : 00008000-00009FFF
APPLI. AREA : 00011000-00051FFF
APPLI. AREA : 00053000-00053FFF
APPLI. AREA : BFFFF000-BFFFFFFF
>
```

【参考】

-OS オプション（18 頁）

THREAD(スレッドアタッチ情報)

書式

THREAD

機能

スレッド状態を表示

解説

カレント PARTNER にアタッチされているすべてのスレッドの pid, task_struct, pc の情報を表示します。
このコマンドは ADD モードでマルチスレッドアプリケーションをデバッグしている場合に有効です。

【使用例】

```
>thread ↓
 pid:97(0x61) task_struct:C0F4C000 pc:400D59C4
 pid:99(0x63) task_struct:C0CA2000 pc:400D59C4
 pid:100(0x64) task_struct:C0CA0000 pc:400D59C4
>
```

PS(プロセス情報)

書式

PS [/R]

機能

プロセスの状態を表示

解説

現在実行中のアプリケーションのプロセスを表示します。**/R オプション**をつけるとプロセス ID の大きい順(逆順)に表示します。

【使用例】

```
>ps ↓
  1(0x1)        /sbin/init
  60(0x3c)      /sbin/portmap
  86(0x56)      /sbin/syslogd
  88(0x58)      /sbin/klogd
  93(0x5d)      /usr/sbin/inetd
  94(0x5e)      /bin/bash
  97(0x61)      /KMC/samples/sample
>ps /r ↓
  97(0x61)      /KMC/samples/sample
  94(0x5e)      /bin/bash
  93(0x5d)      /usr/sbin/inetd
  88(0x58)      /sbin/klogd
  86(0x56)      /sbin/syslogd
  60(0x3c)      /sbin/portmap
  1(0x1)        /sbin/init
>
```

MAPS(メモリ情報)

書式 1

MAPS [<PID>]

書式 2

MAPS [<PID>],< チェックアドレス >

機能

アプリケーションのメモリ情報

解説

書式 1 は <PID> で指定したプロセス ID のアプリケーションの maps(/proc/<PID>/maps と同じ情報) を表示します。カレント PARTNER にアプリケーションがアタッチされている場合に <PID> を省略すると、そのアタッチされているアプリケーションの maps 情報を表示します。

書式 2 は < **チェックアドレス** > で指定されたアドレスに対応する maps 情報のみ表示します。

【使用例】

```
>maps ↓
00008000-00009000 r-xp /KMC/samples/thread_sample
00010000-00011000 rw-p /KMC/samples/thread_sample
00011000-00013000 rwxp
40000000-40016000 r-xp /lib/ld-2.2.5.so
40016000-40018000 rw-p
4001d000-4001e000 rw-p /lib/ld-2.2.5.so
4001e000-4001f000 rwxp
4001f000-4002d000 r-xp /lib/libpthread-0.9.so
4002d000-4002f000 ---p /lib/libpthread-0.9.so
4002f000-4003c000 rw-p /lib/libpthread-0.9.so
4003c000-40148000 r-xp /lib/libc-2.2.5.so
40148000-4014c000 ---p /lib/libc-2.2.5.so
4014c000-40155000 rw-p /lib/libc-2.2.5.so
40155000-40159000 rw-p
bf400000-bf401000 ---p
bf401000-bf600000 rwxp
bf600000-bf601000 ---p
bf601000-bf800000 rwxp
bffff000-c0000000 rwxp
>maps 1,c000 ↓
00008000-0000f000 r-xp /sbin/init
>
```

ATTACH(プロセスのアタッチ)

書式

ATTACH <PID>

機能

アプリケーションのデバッガへのアタッチ操作

解説

このコマンドは､ターゲットシステムで実行されているアプリケーションを PARTNER にアタッチします。
アタッチするアプリケーションは <PID> で指定します。
このコマンドは、ターゲット環境で使用する glibc に『5.8 実行中のアプリケーションのアタッチ（143頁)』で行う修正がを行っている場合のみ使用可能です。
このコマンドは、CPU 実行中のみ使用できます。

【使用例】

```
>attach 100 ↓
>
```

G(実行)

書式

G/A

機能

プログラムの実行

解説

このコマンドオプションは、カーネルモードで手動マルチプロセス / マルチスレッドアプリケーションデバッグを行っているときに、/A **オプション**を付加するとすべてのプロセス / スレッド、カーネルを一斉に実行します。

INS(インスペクト)

書式 1

INS LINUX_TASK:<PID>

書式 2

INS LINUX_REG:<PID>

機能

プロセスのタスク情報 / レジスタ情報のインスペクト表示

解説

このコマンドは、**INS コマンド**の Linux 対応機能拡張コマンドで、Linux カーネルのデバッグ情報を読み込んだ PARTNER でのみ有効なコマンドです。

書式1は<PID>で指定されたプロセスのタスク情報(task_struct)をインスペクトウインドウで表示します。

書式 2 は <PID> で指定されたプロセスのレジスタ情報 (regs) をインスペクトウインドウで表示します。

SNAME(ソースパス表示)

書式

SNAME

機能

コードウインドウのソースパスの表示

解説

このコマンドは、現在コードウインドウに表示されているソースのソースパスを表示します。
表示されるソースパスは、コードウインドウのキャプションに表示されているものと同じです。

【使用例】

```
>sname ↓
CURRENT SRC PATH: J:¥HOME¥FOO¥LINUX¥ARCH¥ARM¥KERNEL¥PROCESS.C
>
```

第 3 章　Linux カーネルの変更

この章では、Linux 対応 PARTNER で Linux システムをより高度にデバッグするためのカーネルソースの修正について説明します。

3.1 Linux カーネルソースの修正の必要性

PARTNER は、Linux カーネルソースの一部を修正することにより、より高機能なデバッグを行うことが出来ます。

ローダブルモジュールデバッグの自動化

Linux カーネルを修正することによりローダブルモジュールがインストールされた時点で、PARTNER がブレークしデバッグ情報を自動的に読み込み、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

Linux カーネルを修正することによりアプリケーションモードでデバッグすることが可能になります。アプリケーションモードデバッグについては、『2.1.2 アプリケーションモードデバッグ（14 頁）』を参照してください。

リアルタイムトレースのプロセス別表示

Linux カーネルとアプリケーションを別の PARTNER ウインドウでデバッグしている場合、該当プロセスのリアルタイムトレースのみヒストリウインドウに表示することが出来ます。

この節では、高度なデバッグを行うための Linux カーネルソースの一部修正の方法を記述します。

3.2 カーネルソース修正

カーネルソースツリーに行う修正は、PARTNER の Linux サポートファイルの追加と既存ファイルの一部修正のみです。

3.2.1 追加ファイルリスト

以下のファイルを Linux カーネルソースツリーに追加してください。このファイルは付属 CD に入っています。
カーネルツリーのトップディレクトリで付属 CD からファイルを展開します。

```
LINUX86>tar xvzf kmc_kernel_modify.tgz ↓
```

- 追加 $(TOPDIR)/KMC/Makefile_kmc（35 頁）
- 追加 $(TOPDIR)/KMC/config_kmc.in（36 頁）
- 追加 $(TOPDIR)/KMC/Rules_kmc.make（36 頁）
- 追加 $(TOPDIR)/KMC/kmc.h（37 頁）
- 追加 $(TOPDIR)/KMC/kmc.c（41 頁）
- 追加 $(TOPDIR)/KMC/__brk_code.h（42 頁）

追加 $(TOPDIR)/KMC/Makefile_kmc

```
ifdef CONFIG_DEB_NO_OPTIMIZE
CFLAGS := $(filter-out -O2,$(CFLAGS))
CFLAGS := $(filter-out -Os,$(CFLAGS))
endif

ifndef CONFIG_DEBINFO_NONE
CFLAGS := $(filter-out -g,$(CFLAGS))
CFLAGS := $(filter-out -ggdb,$(CFLAGS))
ifdef CONFIG_DEBINFO_STAB
CFLAGS += -gstabs
else
ifdef CONFIG_DEBINFO_STABP
CFLAGS += -gstabs+
else
ifdef CONFIG_DEBINFO_DWARF
CFLAGS += -gdwarf
else
ifdef CONFIG_DEBINFO_DWARFP
CFLAGS += -gdwarf+
else
ifdef CONFIG_DEBINFO_DWARF2
CFLAGS += -gdwarf-2
else
CFLAGS += -g
endif
endif
endif
endif
endif
endif

CFLAGS  += -I$(TOPDIR)/KMC
```

追加 $(TOPDIR)/KMC/config_kmc.in

```
mainmenu_option next_comment

comment 'PARTNER Debugging'
   choice 'Debug information type'                                ¥
   "NONE                CONFIG_DEBINFO_NONE                       ¥
   STAB                 CONFIG_DEBINFO_STAB                       ¥
   extendedSTAB         CONFIG_DEBINFO_STABP                      ¥
   DWARF-1              CONFIG_DEBINFO_DWARF                      ¥
   extenedDWARF-1       CONFIG_DEBINFO_DWARFP                     ¥
   DWARF-2              CONFIG_DEBINFO_DWARF2" NONE

   bool 'Enable patch for PARTNER debug' CONFIG_KMC_PATCH

   if [ "$CONFIG_KMC_PATCH" = "y" ]; then

      bool '  Loadable module auto attach' CONFIG_KMC_MODULE_AUTO

      mainmenu_option next_comment
      comment 'PARTNER Extend menu'
         bool 'OFF:ICE AVAILABLE STATUS Support' CONFIG_KMC_PARTNER_AVAILABLE_OFF
          if [ "$CONFIG_KMC_MODULE_AUTO" = "y" -a "$CONFIG_KMC_PARTNER_AVAILABLE_OFF" = "n" ];
then
            bool 'ice check at module debug' CONFIG_KMC_ICE_CHK_MOD_DEB
         fi
         bool 'OFF:PARTNER Virtual ICE Support' CONFIG_KMC_PARTNER_VIRTUAL_ICE_OFF
         bool 'OFF:PARTNER THREAD COLLECTING' CONFIG_KMC_PARTNER_COLLECT_THREAD_OFF
         bool 'OFF:PROCESS TRACE ENHANCE' CONFIG_KMC_TRACE_EXT_OFF
      endmenu

   fi

endmenu
```

追加 $(TOPDIR)/KMC/Rules_kmc.make

```
ifeq ($(PATCHLEVEL),6)

__KMC_MODULE_NAME = -D__KMC_OBJ_NAME=¥"$(subst $(comma),_,/$(@D)/$(*F).ko)¥"

else

__KMC_MODULE_NAME = -D__KMC_OBJ_NAME=¥"$(subst $(TOPDIR),,$(shell pwd)/$(@F))¥"


endif

__KMC_TEMP_CFLAGS:= $(EXTRA_CFLAGS)
EXTRA_CFLAGS   =
EXTRA_CFLAGS   = $(__KMC_MODULE_NAME)
EXTRA_CFLAGS   += $(__KMC_TEMP_CFLAGS)

__KMC_TEMP_CFLAGS:= $(EXTRA_CFLAGS_nostdinc)
EXTRA_CFLAGS_nostdinc=
EXTRA_CFLAGS_nostdinc= $(__KMC_MODULE_NAME)
EXTRA_CFLAGS_nostdinc+= $(__KMC_TEMP_CFLAGS)



__KMC_TEMP_AFLAGS:= $(EXTRA_AFLAGS)
EXTRA_AFLAGS   =
EXTRA_AFLAGS   = $(__KMC_MODULE_NAME)
EXTRA_AFLAGS   += $(__KMC_TEMP_AFLAGS)
```

追加 $(TOPDIR)/KMC/kmc.h

```
/*
 * Kyoto Microcomputer PARTNER Linux support
 */


#ifndef __KMC_H__ /* { */
#define __KMC_H__

#include "__brk_code.h"


#ifdef CONFIG_KMC_PATCH /* { */

#ifdef MODULE /* { */
#if defined(CONFIG_KMC_MODULE_AUTO) && defined(__KMC_MODULE_DEBUG) /* { */

#include <linux/version.h>

void __kmc_module_debug_start(void);
void __kmc_module_debug_end(void);

void __kmc_driver_start(void);
void __kmc_driver_end(void);
void __kmc_driver_init_end(void);

char __kmc_driver_tmp __attribute__ ((section (".bss"))) ;

#ifdef CONFIG_KMC_ICE_CHK_MOD_DEB /* { */
 #define __KMC_ICE_CHK()                 ¥
    {int ret;                                           ¥
    extern asmlinkage int sys_ptrace(long request, long pid, long addr, long data);¥
    if(0x434D4B00 != (0xffffff00&(ret=sys_ptrace(0x4B4D4300,0,0,0)))){¥
        return;                                                 ¥
    }                                                           ¥
    if(0 == (0x000000ff&ret)){          ¥
        return;                                                 ¥
    }                                                           ¥
    }
#else /* }{ !CONFIG_KMC_ICE_CHK_MOD_DEB */
 #define __KMC_ICE_CHK()
#endif /* } CONFIG_KMC_ICE_CHK_MOD_DEB */


#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,0) /* { */
static void __kmc_driver(void)
{
    asm("   .text");
    asm("   .long   0x4c434d83");
    asm("__kmc_driver_init_end:");
    __KMC_BRK_CODE();
    asm("   .long   0x4c434d82");
    asm("__kmc_driver_end:");
    __KMC_BRK_CODE();
    asm("   .long   0x4c434d81");
    asm("__kmc_driver_start:");
    __KMC_BRK_CODE();
    asm("   .long   0x4c434d80");
    asm("   .long   __kmc_driver_name");
    asm("   .long   __kmc_driver_tmp");
    asm("   .long   0x4c434d8f");
    asm("   .long   __kmc_init_text");
    asm("   .long   __kmc_exit_text");
    asm("   .long   __kmc_init_data");
    asm("   .long   __kmc_exit_data");
}

__exitdata int __kmc_exit_data;
__initdata int __kmc_init_data;

void __init __kmc_init_text(void)
{
    __KMC_ICE_CHK();
    __kmc_driver_start();
}

void __init __kmc_init_text_finish(void)
{
    __kmc_driver_init_end();
}

void __exit __kmc_exit_text(void)
{
    __KMC_ICE_CHK();
    __kmc_driver_end();
}
```

```
#define __kmc_module_debug_start__kmc_init_text
#define __kmc_module_debug_end__kmc_exit_text

#else /* }{ LINUX_VERSION_CODE < KERNEL_VERSION(2,6,0) */
static void __kmc_driver(void)
{
    asm("   .text");
    asm("   .long   0x4c434d82");
    asm("__kmc_driver_end:");
    __KMC_BRK_CODE();
    asm("   .long   0x4c434d81");
    asm("__kmc_driver_start:");
    __KMC_BRK_CODE();
    asm("   .long   0x4c434d80");
    asm("   .long   __kmc_driver_name");
    asm("   .long   __kmc_driver_tmp");
}

void
__kmc_module_debug_start(void)
{
    __KMC_ICE_CHK();
    __kmc_driver_start();
}

void
__kmc_module_debug_end(void)
{
    __KMC_ICE_CHK();
    __kmc_driver_end();
}
#endif /* } LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,0) */

#if defined(__KMC_MODULE_NAME) /* { */
static char __kmc_driver_name[] = __KMC_MODULE_NAME;
#else /* }{ */
static char __kmc_driver_name[] = __KMC_OBJ_NAME;
#endif /* } defined(__KMC_MODULE_NAME) */


// コンパイラによって警告が発生する
#undef  module_init
#undef  module_exit

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,0) /* { */
#define module_init(initfn)                 ¥
        int init_module(void)               ¥
        {int ret; __kmc_init_text(); ret = initfn();¥
        __kmc_init_text_finish(); return ret;}
#define module_exit(exitfn)                 ¥
        void cleanup_module(void)           ¥
        { exitfn(); __kmc_exit_text(); }

#else /* }{ LINUX_VERSION_CODE < KERNEL_VERSION(2,6,0) */
#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; }
#endif /* } LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,0) */

#endif /* } defined(CONFIG_KMC_MODULE_AUTO) && defined(__KMC_MODULE_DEBUG) */


#else /* }{ !MODULE */

/* ########## kernel/exit.c { ########## */
#ifdef CONFIG_KMC_PARTNER_COLLECT_THREAD_OFF /* { */
#define __KMC_MAX_PT_COUNT32

extern struct task_struct *__kmc_tss_list[__KMC_MAX_PT_COUNT + 1];

static inline void __kmc_do_exit__(struct task_struct *tsk)
{
    int i;
    extern void __kmc_do_exit(void);

    if(__kmc_tss_list[0]!=0){
        for(i=1;i<(__KMC_MAX_PT_COUNT + 1);++i){
            if(__kmc_tss_list[i]==tsk){
               (int)__kmc_tss_list[0]=i;
               __kmc_do_exit();
```

```
            }
        }
    }
}
#else /* }{ !CONFIG_KMC_PARTNER_COLLECT_THREAD_OFF */
#define __KMC_MAX_THREAD_COUNT32
#define __KMC_MAX_PT_COUNT16

extern   struct   task_struct   *__kmc_tss_list_array[__KMC_MAX_PT_COUNT   +
1][__KMC_MAX_THREAD_COUNT];

static inline void __kmc_do_exit__(struct task_struct *tsk)
{
    int i;
    extern void __kmc_do_exit(void);

    if(__kmc_tss_list_array[0][0] != 0){
        for(i = 1; i < (__KMC_MAX_PT_COUNT + 1); ++i){
            int j;
            struct task_struct **tpp;
            struct task_struct *tp;

            tpp = __kmc_tss_list_array[i];
            for(j = 0; j < __KMC_MAX_THREAD_COUNT; ++j){
               tp = *(tpp++);
               if(tp == 0) break;
               if(tp == tsk){
                   while(j < (__KMC_MAX_THREAD_COUNT - 1)){
                      if(*tpp == 0) break;
                      *(tpp -1 ) = *tpp;
                      *tpp = tsk;
                      ++tpp; ++j;
                   }
                   (int)__kmc_tss_list_array[0][0] = i;
                   (int)__kmc_tss_list_array[0][1] = j;
                   __kmc_do_exit();
                   i = (__KMC_MAX_PT_COUNT + 1);
                   break;
               }
            }
        }
    }
}
#endif /* } CONFIG_KMC_PARTNER_COLLECT_THREAD_OFF */
#define __KMC_DO_EXIT(t)__kmc_do_exit__(t)
/* ########## kernel/exit.c } ########## */

/* ########## kernel/timer.c { ########## */
#ifdef CONFIG_KMC_PARTNER_VIRTUAL_ICE_OFF /* { */
#define __KMC_SYS_GETPID()
#else /* }{ !CONFIG_KMC_PARTNER_VIRTUAL_ICE_OFF */
#define __KMC_SYS_GETPID()                          ¥
        {                                                       ¥
              extern int __kmc_sys_getpid_flag;¥
              extern void __kmc_sys_getpid(void);¥
              if(__kmc_sys_getpid_flag) __kmc_sys_getpid();¥
        }
#endif /* } CONFIG_KMC_PARTNER_VIRTUAL_ICE_OFF */
/* ########## kernel/timer.c } ########## */

/* ########## kernel/sched.c { ########## */
#ifdef CONFIG_KMC_TRACE_EXT_OFF /* { */
#define __KMC_SCHED_CALL(p,n)
#else /* }{ !CONFIG_KMC_TRACE_EXT_OFF */
#define KMC_MAX_SHEDULE_LIST    1024
#define __KMC_SCHED_CALL(p,n)                       ¥
        {                                                       ¥
              extern void (*__kmc_schedule_call)(struct task_struct *,struct task_struct *);¥
              __kmc_schedule_call(p,n);  ¥
        }
#endif /* CONFIG_KMC_TRACE_EXT_OFF } */
/* ########## kernel/sched.c } ########## */

/* ########## kernel/ksyms.c { ########## */
#ifdef CONFIG_KMC_ICE_CHK_MOD_DEB /* { */
#define __KMC_EXPORT_SYMBOL_PTRACE()     ¥
        extern asmlinkage int sys_ptrace(long request, long pid, long addr, long data);¥
        EXPORT_SYMBOL(sys_ptrace)
#else /* }{ !CONFIG_KMC_ICE_CHK_MOD_DEB */
#define __KMC_EXPORT_SYMBOL_PTRACE()
#endif /* } CONFIG_KMC_ICE_CHK_MOD_DEB */
/* ########## kernel/ksyms.c } ########## */

/* ########## arch/XXX/kernel/ptrace.c { ########## */
#ifdef CONFIG_KMC_PARTNER_AVAILABLE_OFF /* { */
#define __KMC_CHECK_PTRACE_REQUEST(req,r)
#else /* }{ !CONFIG_KMC_PARTNER_AVAILABLE_OFF */
#define __KMC_CHECK_PTRACE_REQUEST(req,r)¥
```

```
                if(req == 0x4B4D4300){¥
                        extern int __kmc_available_ice;¥
                        r = __kmc_available_ice;¥
                        goto out;               ¥
                }
#endif /* } CONFIG_KMC_PARTNER_AVAILABLE_OFF */
/* ########## arch/XXX/kernel/ptrace.c } ########## */


#endif /* } MODULE */
#else /* }{ !CONFIG_KMC_PATCH */

#define __KMC_SCHED_CALL(p,n)
#define __KMC_DO_EXIT(t)
#define __KMC_SYS_GETPID()
#define __KMC_EXPORT_SYMBOL_PTRACE()
#define __KMC_CHECK_PTRACE_REQUEST(req,r)


#endif /* } CONFIG_KMC_PATCH */


#endif /* } __KMC_H__ */
```

追加 $(TOPDIR)/KMC/kmc.c

```
/*
 * Kyoto Microcomputer PARTNER Linux support
 */

#include "kmc.h"

#ifdef CONFIG_KMC_PATCH /* { */

/* ########## kernel/exit.c { ########## */
void __kmc_do_exit(void)
{
        __asm__("nop");
}
#ifdef CONFIG_KMC_PARTNER_COLLECT_THREAD_OFF /* { */
struct task_struct *__kmc_tss_list[__KMC_MAX_PT_COUNT + 1];
#else /* }{ !CONFIG_KMC_PARTNER_COLLECT_THREAD_OFF */
struct task_struct *__kmc_tss_list_array[__KMC_MAX_PT_COUNT + 1][__KMC_MAX_THREAD_COUNT];
#endif /* } CONFIG_KMC_PARTNER_COLLECT_THREAD_OFF */
/* ########## kernel/exit.c } ########## */

/* ########## kernel/timer.c { ########## */
#ifndef CONFIG_KMC_PARTNER_VIRTUAL_ICE_OFF /* { */
int __kmc_sys_getpid_flag;
void __kmc_sys_getpid(void)
{
        __asm__("nop");
}
#endif /* } CONFIG_KMC_PARTNER_VIRTUAL_ICE_OFF */
/* ########## kernel/timer.c } ########## */

/* ########## kernel/sched.c { ########## */
#ifndef CONFIG_KMC_TRACE_EXT_OFF /* { */
int __kmc_schedules_list_pid[KMC_MAX_SHEDULE_LIST];
int __kmc_schedules_index;
int __kmc_schedules_index_max=KMC_MAX_SHEDULE_LIST;

void __kmc_schedule(struct task_struct *prev,struct task_struct *next)
{
    int index_next;

    index_next=__kmc_schedules_index & (KMC_MAX_SHEDULE_LIST-1);
    ++__kmc_schedules_index;
    __kmc_schedules_list_pid[index_next]=next->pid;
}

void (*__kmc_schedule_call)(struct task_struct *,struct task_struct *)=__kmc_schedule;
#endif /* CONFIG_KMC_TRACE_EXT_OFF } */
/* ########## kernel/sched.c } ########## */

/* ########## arch/XXX/kernel/ptrace.c { ########## */
#ifndef CONFIG_KMC_PARTNER_AVAILABLE_OFF /* { */
int __kmc_available_ice = 0;
#endif /* } CONFIG_KMC_PARTNER_AVAILABLE_OFF */
/* ########## arch/XXX/kernel/ptrace.c } ########## */

#endif /* CONFIG_KMC_PATCH } */
```

追加 $(TOPDIR)/KMC/_brk_code.h

```
/*
 * Kyoto Microcomputer PARTNER Linux support
 *
 *  各 CPU のブレークコード定義
 */

#ifndef __BRK_CODE_H__
#define __BRK_CODE_H__


#ifdef CONFIG_ARM
 #if defined(CONFIG_CPU_V6)
  #define__KMC_BRK_CODE()asm(".long0xe1200070")
 #else
  #define__KMC_BRK_CODE()asm(".long0xdeeedeee")
 #endif
#endif

#ifdef CONFIG_MIPS
 #if defined(CONFIG_CPU_TX49XX)
  #define__KMC_BRK_CODE()asm(".long0x0000000e")
 #elif defined(CONFIG_CPU_VR41XX)
  #define__KMC_BRK_CODE()asm(".long0x7000003f")
 #endif
#endif

#ifdef CONFIG_CPU_SH4
 #define__KMC_BRK_CODE()asm(".long0x003b003b")
#endif
#ifdef CONFIG_CPU_SH3
 #define__KMC_BRK_CODE()asm(".long0x00000000")
#endif

#ifdef CONFIG_AM33
 #define__KMC_BRK_CODE()asm(".long0xffffffff")
#endif

#ifndef __KMC_BRK_CODE
 #error !! __KMC_BRK_CODE is not defined !!
#endif

#endif
```

3.2.2 修正ファイルリスト

以下のファイルを修正してください。

- 修正 $(TOPDIR)/Rules.make (43 頁)
- 修正 $(TOPDIR)/arch/sh/Makefile (43 頁)
- 修正 $(TOPDIR)/arch/sh/config.in (43 頁)
- 修正 $(TOPDIR)/arch/sh/kernel/ptrace.c (44 頁)
- 修正 $(TOPDIR)/kernel/exit.c (44 頁)
- 修正 $(TOPDIR)/kernel/sched.c (44 頁)
- 修正 $(TOPDIR)/kernel/timer.c (46 頁)
- 修正 $(TOPDIR)/kernel/ksyms.c (46 頁)
- 修正 $(TOPDIR)/include/linux/init.h (46 頁)

修正 $(TOPDIR)/Rules.make

```
 MOD_SUB_DIRS:= $(sort $(subdir-m) $(both-m))
 ALL_SUB_DIRS:= $(sort $(subdir-y) $(subdir-m) $(subdir-n) $(subdir-))

+include $(TOPDIR)/KMC/Rules_kmc.make

 #
 # Common rules
 #

 %.s: %.c
```

修正 $(TOPDIR)/arch/sh/Makefile

```
 MODFLAGS+=

 #
 #

+include $(TOPDIR)/KMC/Makefile_kmc

 ifdef CONFIG_CPU_SH3
 CFLAGS        += -m3
 AFLAGS        += -m3
 endif
```

修正 $(TOPDIR)/arch/sh/config.in

```
       choice 'DataBits' ¥
          "7 CONFIG_KGDB_DEFBITS_7¥
           8 CONFIG_KGDB_DEFBITS_8"8
    endmenu
 fi

 endmenu

+source KMC/config_kmc.in

 source lib/Config.in
```

修正 $(TOPDIR)/arch/sh/kernel/ptrace.c

```
asmlinkage int sys_ptrace(long request, long pid, long addr, long data)
{
        struct task_struct *child;
        int ret;

        lock_kernel();
        ret = -EPERM;
        if (request == PTRACE_TRACEME) {
              /* are we already being traced? */
              if (current->ptrace & PT_PTRACED)
                    goto out;
              /* set the ptrace bit in the process flags. */
              current->ptrace |= PT_PTRACED;
              ret = 0;
              goto out;
        }
+       #include "kmc.h"
+       __KMC_CHECK_PTRACE_REQUEST(request,ret);
        ret = -ESRCH;
        read_lock(&tasklist_lock);
        child = find_task_by_pid(pid);
```

修正 $(TOPDIR)/kernel/exit.c

```
NORET_TYPE void do_exit(long code)
{
       struct task_struct *tsk = current;

+      #include "kmc.h"
+      __KMC_DO_EXIT(tsk);

       if (in_interrupt())
             panic("Aiee, killing interrupt handler!");
       if (!tsk->pid)
             panic("Attempted to kill the idle task!");

             :
             :
             :
             :
             :

#if !defined(__alpha__) && !defined(__ia64__) && !defined(__arm__)

/*
 * sys_waitpid() remains for compatibility. waitpid() should be
 * implemented by calling sys_wait4() from libc.a.
 */
asmlinkage long sys_waitpid(pid_t pid,unsigned int * stat_addr, int options)
{
       return sys_wait4(pid, stat_addr, options, NULL);
}

#endif

+#include "kmc.c"
```

修正 $(TOPDIR)/kernel/sched.c

2.4.19 以前のソースの場合

```
asmlinkage void schedule(void)
{
    struct schedule_data * sched_data;
    struct task_struct *prev, *next, *p;
              :
              :
              :
              :
    TRACE_SCHEDCHANGE(prev, next);
+   #include "kmc.h"
+   __KMC_SCHED_CALL(prev,next);

    /* This just switches the register state and the
     * stack.
     */
    switch_to(prev, next, prev);
    __schedule_tail(prev);

same_process:
```

2.4.20 以降のソースの場合

```
static inline task_t * context_switch(task_t *prev, task_t *next)
{
```

```
        struct mm_struct *mm = next->mm;
        struct mm_struct *oldmm = prev->active_mm;

        if (unlikely(!mm)) {
                next->active_mm = oldmm;
                atomic_inc(&oldmm->mm_count);
                enter_lazy_tlb(oldmm, next, smp_processor_id());
        } else
                switch_mm(oldmm, mm, next, smp_processor_id());

        if (unlikely(!prev->mm)) {
                prev->active_mm = NULL;
                mmdrop(oldmm);
        }

+       #include "kmc.h"
+       __KMC_SCHED_CALL(prev,next);

        /* Here we just switch the register state and the stack. */
        switch_to(prev, next, prev);

        return prev;
```

sched.c 内の修正個所は、switch_to(prev,next,prev); を実行する直前に挿入して ください。カーネルバージョンによって switch_to() 関数の記述個所が大きく 違います。注意して ください。

カーネルソース修正

修正 $(TOPDIR)/kernel/timer.c

```
 /* Thread ID - the internal kernel "pid" */
 asmlinkage long sys_gettid(void)
 {
+       #include "kmc.h"
+       __KMC_SYS_GETPID();
        return current->pid;
 }
```

修正 $(TOPDIR)/kernel/ksyms.c

```
 EXPORT_SYMBOL(init_task_union);

 EXPORT_SYMBOL(tasklist_lock);
 EXPORT_SYMBOL(pidhash);

+#include "kmc.h"
+__KMC_EXPORT_SYMBOL_PTRACE();
```

修正 $(TOPDIR)/include/linux/init.h

```
 #if defined(MODULE) || defined(CONFIG_HOTPLUG)
 #define __devexit_p(x) x
 #else
 #define __devexit_p(x) NULL
 #endif

+#ifndef __ASSEMBLY__
+#include "kmc.h"
+#endif /* __ASSEMBLY__ */

 #endif /* _LINUX_INIT_H */
```

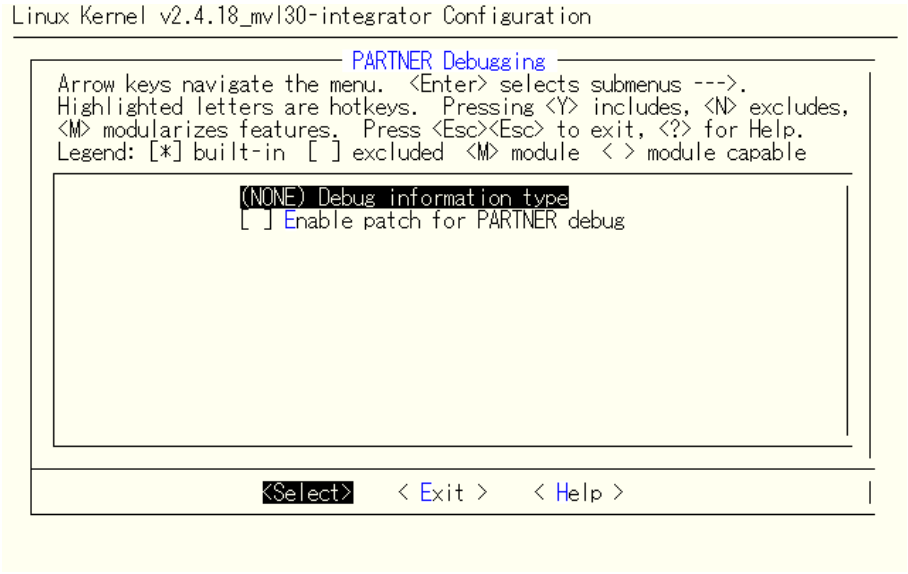
3.2.3 カーネルコンフィグレーション

Linux カーネルソースツリーに修正を施すと、カーネルコンフィグレーションメニューに [PARTNER Debugging] が追加されます。ここでは、PARTNER でのデバッグに関する項目が設定できるようになります。

```
LINUX86>make menuconfig ↓
```

[Kernel hacking]->[PARTNER Debugging]

図 3-1 カーネルコンフィグレーション



● Debug information type

Linux カーネルに付加するデバッグ情報のタイプを選択します。

NONE, STAB, extendedSTAB, DWARF-1, extendDWARF-1, DWARF2 の中から選択できます。

通常、extendedSTAB を選択します。デバッグ情報付きの Linux カーネルファイルを PARTNER で読み込んで、デバッグ情報がおかしなときは、他のフォーマット (DWARF2 など) を試してみてください。

● Enable patch for PARTNER debug

カーネルソース修正を有効にし、PARTNER での高機能なデバッグに対応するカーネル設定になります。

● Loadable module auto attach

ローダブルモジュールインストール時に PARTNER をブレークし、デバッグ情報を自動的にロードします。

詳しくは、『4.2 ローダブルモジュールのデバッグ (58 頁)』を参照してください。

[PARTNER Extend menu] は通常変更しないでください。

第 4 章 カーネル、ローダブルモジュールのデバッグの手順

この章では、Linux 対応 PARTNER でカーネル、ローダブルモジュールをデバッグする手順について説明します。

4.1 カーネルのデバッグ手順

この節では、Linux カーネルのデバッグ方法を次の流れで説明します。
この説明は『3.2 カーネルソース修正（35 頁）』で指示された修正がカーネルに対して行われていることを前提にされています。

(1) Linux カーネルのコンフィグレーション（51 頁）
(2) Linux カーネルのビルド（51 頁）
(3) カーネルモードでの PARTNER の起動（52 頁）
(4) Linux カーネルのロード（55 頁）
(5) Linux カーネルの実行（57 頁）

カーネルソースの修正が行えない環境では、手作業でデバッグ情報がカーネルオブジェクトに付加されるように Makefile を修正してください。

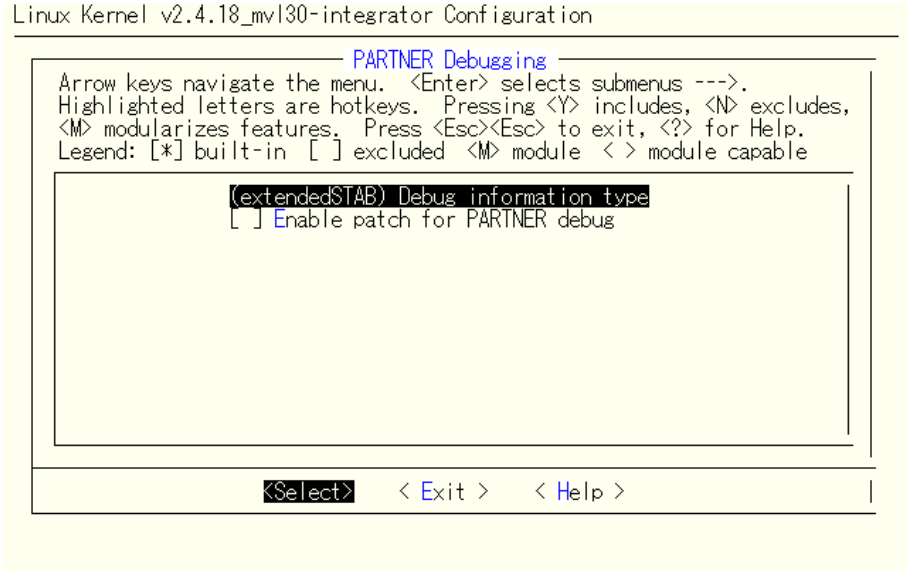
4.1.1 Linux カーネルのコンフィグレーション

Linux カーネルのコンフィグレーションでデバッグ情報のフォーマットを指定します。
推奨は extendedSTAB(-gstabs+) です。PARTNER で実際カーネルを読み込んでデバッグ情報がおかしなときは、DWARF2(-gdwarf-2) に変えて試みてください。
その他の設定項目はカーネルデバッグ時には関係ありません。

```
LINUX86>make menuconfig ↓
```

[Kernel hacking]->[PARTNER Debugging]->[Debug information type]

図 4-1 Linux カーネルのコンフィグレーション



4.1.2 Linux カーネルのビルド

コンフィグレーション設定後、Linux カーネル(vmlinux)を作成します。

```
LINUX86>make ↓
```

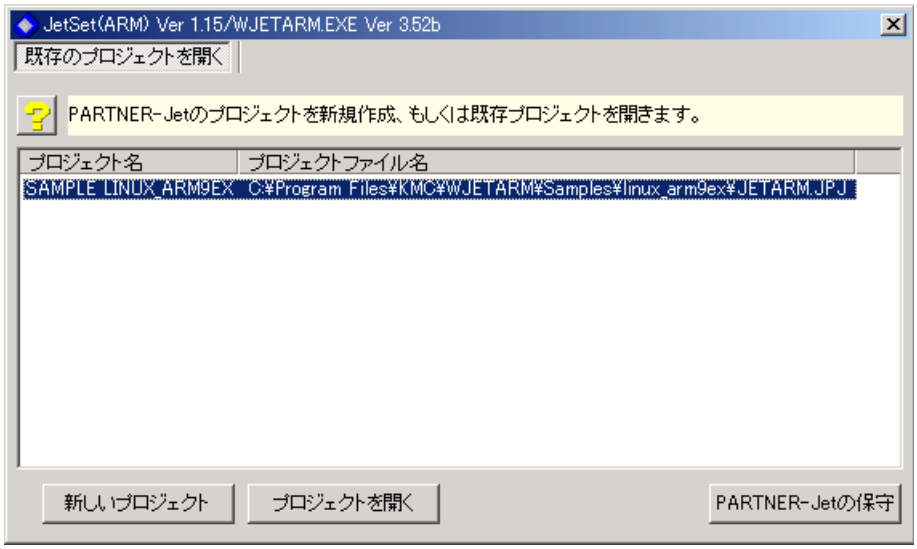
4.1.3 カーネルモードでの PARTNER の起動

カーネルモードで PARTNER を起動するには、次に示す手順で行います。

(1) Jetset でプロジェクトの新規作成もしくは既存のプロジェクトの選択

PARTNER の環境設定プログラム (JETSET(SH)) を起動し、プロジェクトを新規作成もしくは、オープンします。

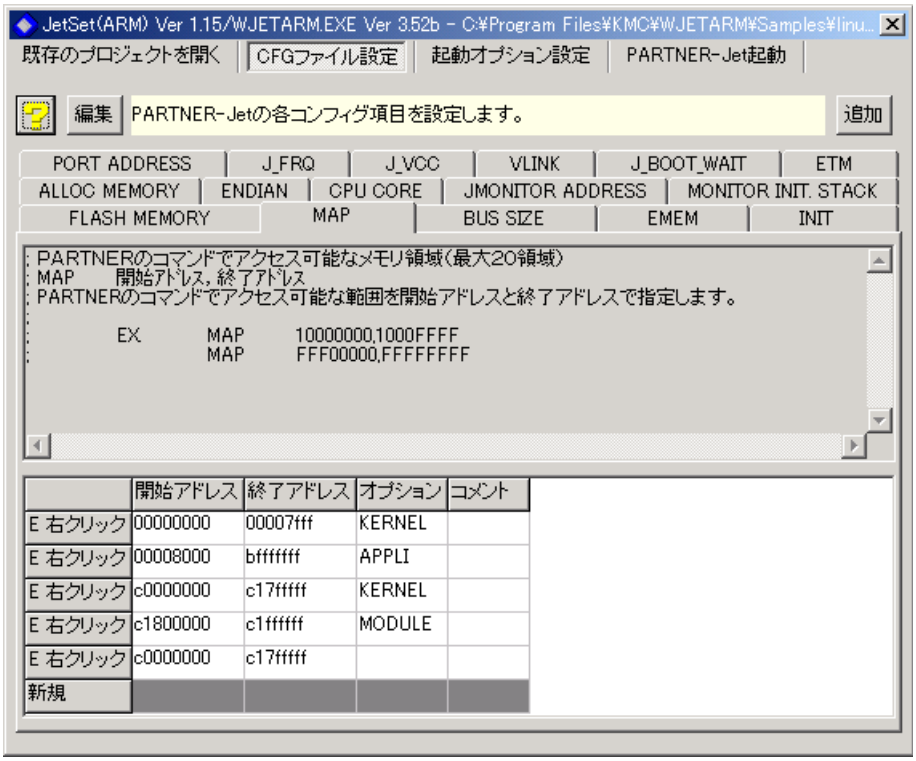
図 4-2 PARTNER プロジェクトのオープン



(2) CFG ファイル設定

MAP フィールドに Linux 用 MAP 情報を指定します。設定するアドレスや属性などについては『CFG ファイルの拡張 (15 頁)』を参照してください。

図 4-3 MAP フィールドの修正



(3) 起動オプション設定

[拡張 >>] ボタンを押し、Linux デバッグ用拡張オプションを指定できるようにします。
カーネルデバッグ時に必ず設定する必要のあるオプションは、以下のとおりです。

● **デバッグ情報バッファサイズ (-B オプション)**

サイズには 100000 程度を指定してください。
もし、カーネルファイルをロードしたときにエラーメッセージ『 デバッグ情報領域がいっぱいです (起動時の **-B オプション**参照)』が表示された場合は、バッファサイズを拡大してください。

● **デバッグ情報タイプ (-XGX オプション)**

『GNU C (Linux etc.)』を選択します。

● **デバッグ情報パス変換 (-XGX オプション)**

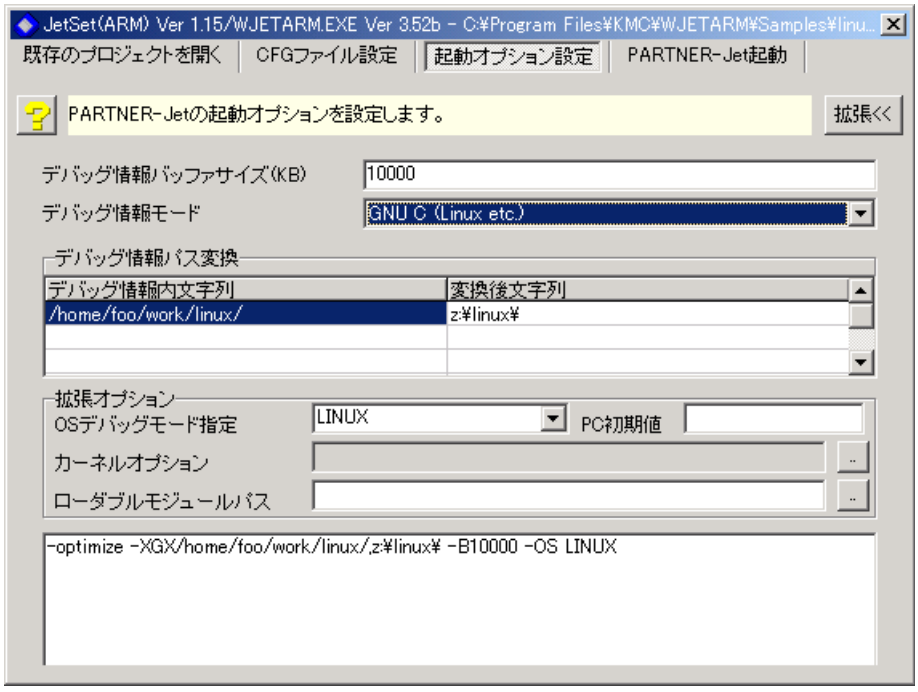
カーネル (vmlinux) をビルドした PATH と Samba でマウントしている PATH が違う場合に指定します。
たとえば、/home/foo/work/linux でカーネルをビルドして、ホスト PC で /home/foo/work を Z: ドライブにマウントした場合は、-XGX/home/foo/work/linux/,z:¥linux¥ と指定します。

● **OS デバッグモード指定 (-OS オプション)**

『LINUX』を選択します。

各オプションについての詳細は、『2.2.2 起動オプション (18 頁)』を参照してください。

図 4-4 起動オプションの設定



(4) PARTNER の起動

JETSET(SH) 上の [起動] ボタンをクリックすることにより、PARTNER が起動します。

図 4-5 PARTNER の起動

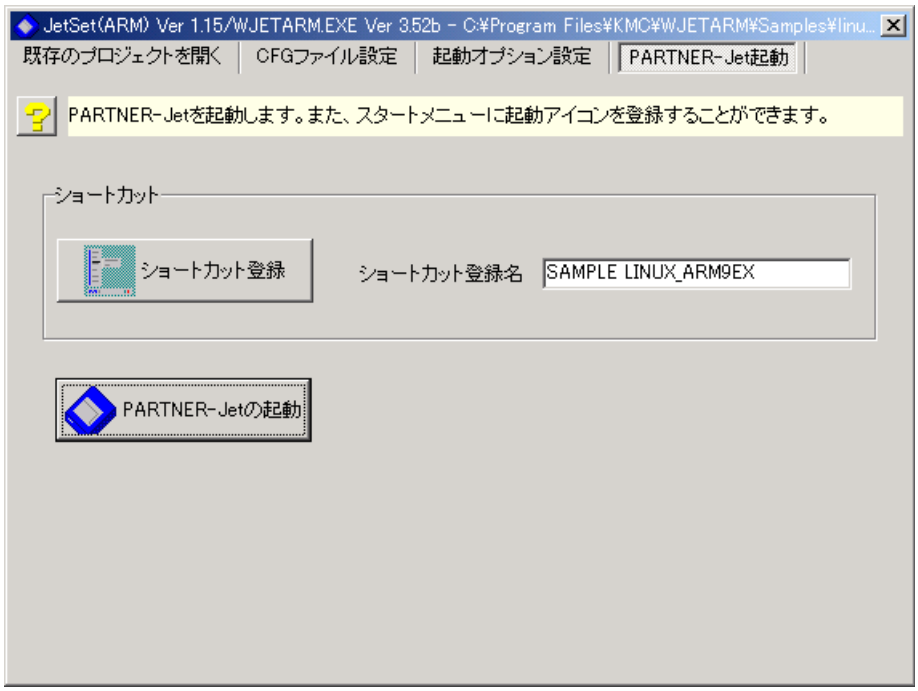
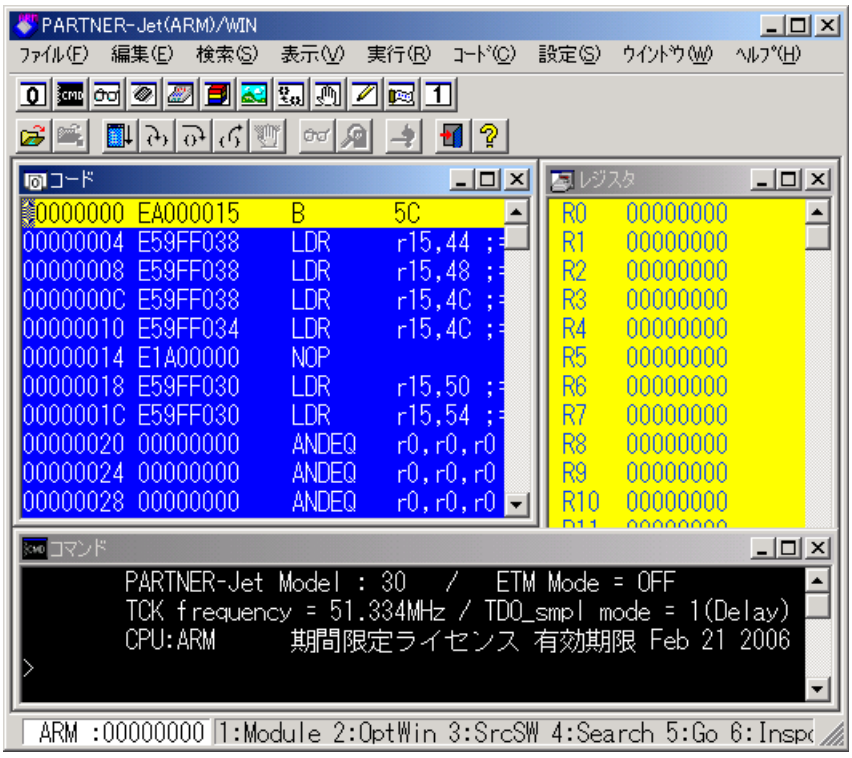


図 4-6 PARTNER の起動画面

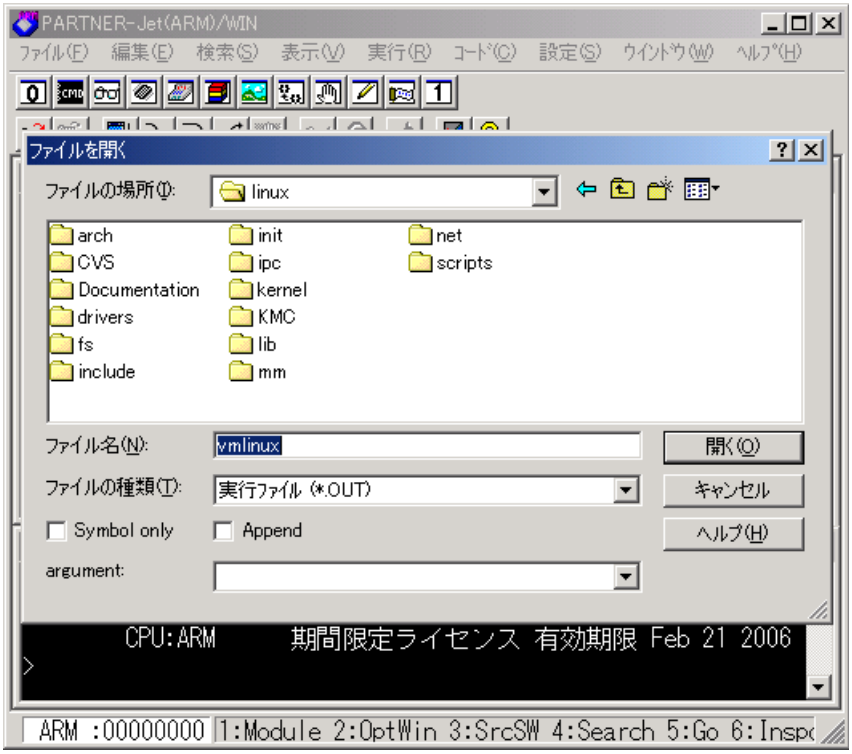


4.1.4 Linux カーネルのロード

『4.1.2 Linux カーネルのビルド (51 頁)』で作成したカーネル(vmlinux)をターゲットメモリにロードします。

```
PT>l vmlinux ↓
```

図 4-7 カーネルファイルのロード



vmlinux から作成された HEX ファイルのバイナリファイルをロードする場合には、以下の手順でカーネルを読み込みます。

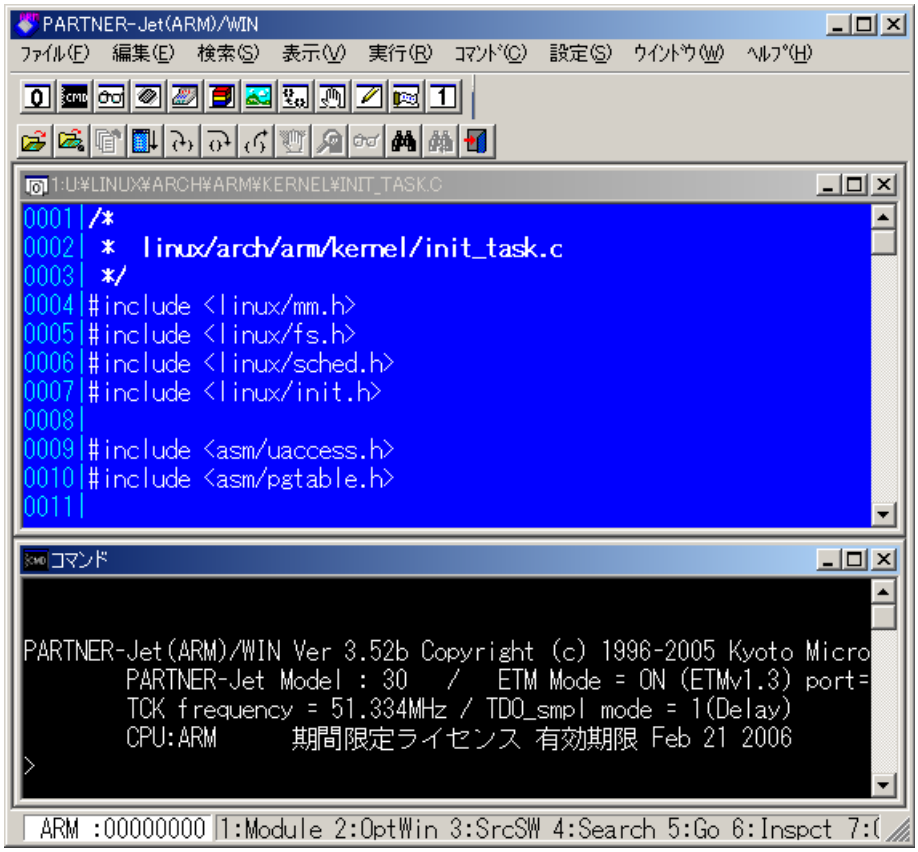
PARTNER のコマンドウィンドウにおいて、次のコマンドを入力します。

【例】rd コマンドで HEX ファイルを読み込み、ls コマンドでデバッグ情報を読み込みます。

```
PT>rd z:¥linux¥vmlinux.hex ↓
PT>ls z:¥linux¥vmlinux ↓
```

コンパイル時にデバッグ情報出力を指定している場合には、カーネルロード時にデバッグ情報も同時にロードされます。
また、一度ロードされたファイルは、PARTNER がロードファイル名と場所をダイアログとコマンドヒストリに記憶するため、以降の操作を簡略化することができます。

図 4-8 カーネルのロード完了



正常にロードされると、PARTNER のコードウィンドウに Linux のソースコードが表示されます。コードウィンドウ上に何も表示されない場合は、Linux PC のディレクトリ情報を Windows PC から参照可能なディレクトリに変換するパスの変換の設定が正しくない場合があります(『 -XGX オプション（19 頁)』参照)。

PARTNER のコードウィンドウに Linux のソースコードが正しく表示された時点で、PARTNER は Linux のカーネルに関して、完全なソースレベルデバッグが可能な状態となっています。

4.1.5 Linux カーネルの実行

カーネルのロードが正しくできた状態で、**G コマンド**、または**[実行] ボタン**でロードした vmlinux を実行させます。

```
PT>g ↓
```

Windows PC とターゲットボードが正しく接続されていて、ターミナルソフトが正常に起動していれば、ブートアップメッセージが表示されます。
ブートアップメッセージが正しく表示されない場合やハングアップする場合は、各設定を再確認してください。

図 4-9 Linux カーネルのブートアップメッセージ

```
NetWinder Floating Point Emulator V0.95 (c) 1998-1999 Rebel.com
Looking up port of RPC 100003/2 on 192.168.1.241
Looking up port of RPC 100005/1 on 192.168.1.241
VFS: Mounted root (nfs filesystem).
Freeing init memory: 212K
INIT: version 2.78 booting
Activating swap...
Checking all file systems...
Parallelizing fsck version 1.22 (22-Jun-2001)
mkdir: cannot create directory `/dev/pts': File exists
Mounting local filesystems...
nothing was mounted
Cleaning: /etc/network/ifstate.
Setting up IP spoofing protection: rp_filter.
Disable TCP/IP Explicit Congestion Notification: done.
Configuring network interfaces: done.
Starting portmap daemon: portmap.
Cleaning: /tmp /var/lock /var/run.
INIT: Entering runlevel: 3
Starting system log daemon: syslogd klogd.
Starting internet superserver: inetd.

MontaVista Linux 3.0, Professional Edition

kzp-arm login:
```

Linuxカーネルの再ロードを行う際は、必ずPARTNERのコマンドウィンドウよりINITコマンドを実行してください。
INITコマンドによって初期化を行わないと、再ロードしてもLinuxは起動できません。

Linuxカーネルが ROM から RAM へ転送されるシステムでは、実行する前にソフトウェアブレークポイントを設定することは出来ません。（カーネル転送後に設定可能）
転送前にブレークポイントを設定する場合は、ハードウェアブレークポイントを設定してください。

4.2 ローダブルモジュールのデバッグ

PARTNER から見た Linux のローダブルモジュール (デバイスドライバ) の特徴は、「Linux カーネル空間で動作するリロケータブルなオブジェクト」ということになります。
ローダブルモジュールは、リンク時 (作成時) にロードされるアドレスで決定されるわけではありません。配置されるアドレスは、Linux カーネルがロードした段階で、初めて決定されます。
したがって、ローダブルモジュールをデバッグするためには、Linux カーネルがインストールしたモジュールをどこに配置したか (実アドレスの情報) を PARTNER が知る必要があります。
PARTNER では、自動的に Linux カーネルがロードしたモジュールのアドレスを取得し、デバッグできます。

『 3.2 カーネルソース修正(35 頁)』を行っていない場合は、これから説明するローダブルモジュールの自動アタッチが出来ません。修正を行っていないカーネルを使用する場合は、『 付録 A 手動モジュールデバッグ(152 頁)』を参照してください。

この節では、RAM ディスク (rd.o) を使用した場合を例として、ローダブルモジュールのデバッグ方法を次の流れで説明します。

(1) Linux カーネルのコンフィグレーション（59 頁）
(2) Linux カーネルのビルド（59 頁）
(3) ローダブルモジュールソースの修正（60 頁）
(4) モジュールの作成（60 頁）
(5) カーネルモードでの PARTNER の起動（61 頁）
(6) Linux カーネルのロード（64 頁）
(7) Linux カーネルの実行（66 頁）
(8) ローダブルモジュールのインストール（67 頁）
(9) PARTNER のブレーク（68 頁）

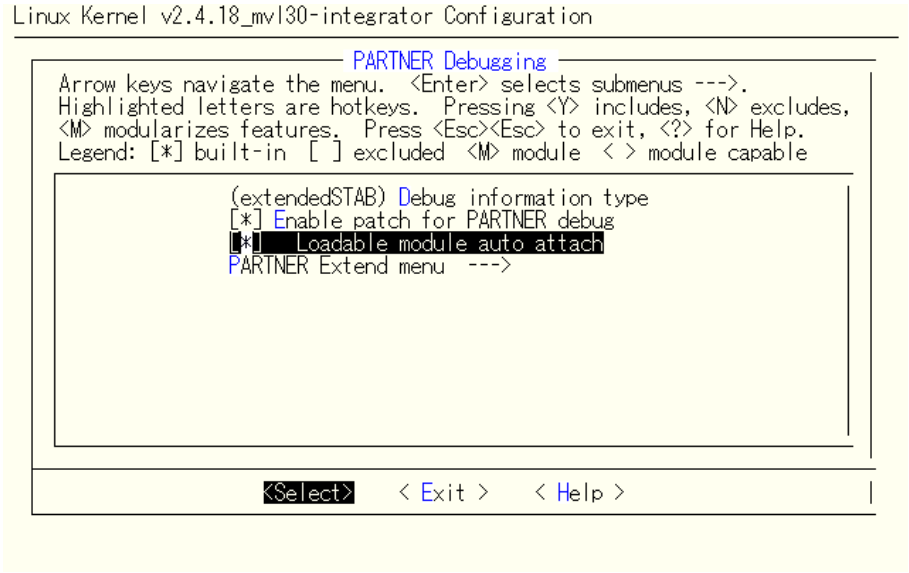
4.2.1 Linux カーネルのコンフィグレーション

ローダブルモジュールの自動デバッグの為、Linux カーネルのコンフィグレーションで [Enable patch for PARTNER debug] と [Loadable module auto attach] を有効にします。
また、デバッグ情報のフォーマットを指定します。推奨は、extendedSTAB(-gstabs+) です。

```
LINUX86>make menuconfig ↓
```

[Kernel hacking]->[PARTNER Debugging]->[Debug information type]
[Kernel hacking]->[PARTNER Debugging]->[Enable patch for PARTNER debug]
[Kernel hacking]->[PARTNER Debugging]->[Loadable module auto attach]

図 4-10 Linux カーネルのコンフィグレーション



4.2.2 Linux カーネルのビルド

上記のコンフィグレーションで、Linux カーネル(vmlinux)を作成します。

```
LINUX86>make ↓
```

4.2.3 ローダブルモジュールソースの修正

デバッグ対象ローダブルモジュールのソースで、module_init() 関数が定義されているソースの先頭に以下の一行を挿入します。

```
#define        __KMC_MODULE_DEBUG
```

【例】

```
+#define    __KMC_MODULE_DEBUG
/*
 * ramdisk.c - Multiple RAM disk driver - gzip-loading version - v. 0.8 beta.
 *
 * (C) Chad Page, Theodore Ts'o, et. al, 1995.
 *
```

module_exit() 関数がない場合は、ダミーの module_exit() 関数を定義してください。

また、独自にローダブルモジュールを作成した場合には、ローダブルモジュールファイルのフルパスを定義します。

```
#define __KMC_MODULE_NAME "モジュールのフルパス"
```

【例】

```
+#define    __KMC_MODULE_DEBUG
+#define    __KMC_MODULE_NAME       "/home/foo/new_module/rd.o"
/*
 * ramdisk.c - Multiple RAM disk driver - gzip-loading version - v. 0.8 beta.
 *
 * (C) Chad Page, Theodore Ts'o, et. al, 1995.
 *
```

Linux カーネルソースツリー内のローダブルモジュールでは、自動的にローダブルモジュールのフルパスを解決するようになっています。
失敗した場合は、**-SK オプション**(20 ページ参照)および、_KMC_MODULE_NAME でローダブルモジュールのフルパスを解決できるように定義してください。

4.2.4 モジュールの作成

デバッグするローダブルモジュールを作成します。
なおこの際に、カーネルと同じデバッグ情報の付加を行うことを忘れないでください(『4.2.1 Linux カーネルのコンフィグレーション (59 頁)』参照)。Linux カーネルソースツリー内のローダブルモジュールは、『4.2.1 Linux カーネルのコンフィグレーション (59 頁)』で指定したデバッグ情報が付加します。

```
LINUX86>make modules ↓
```

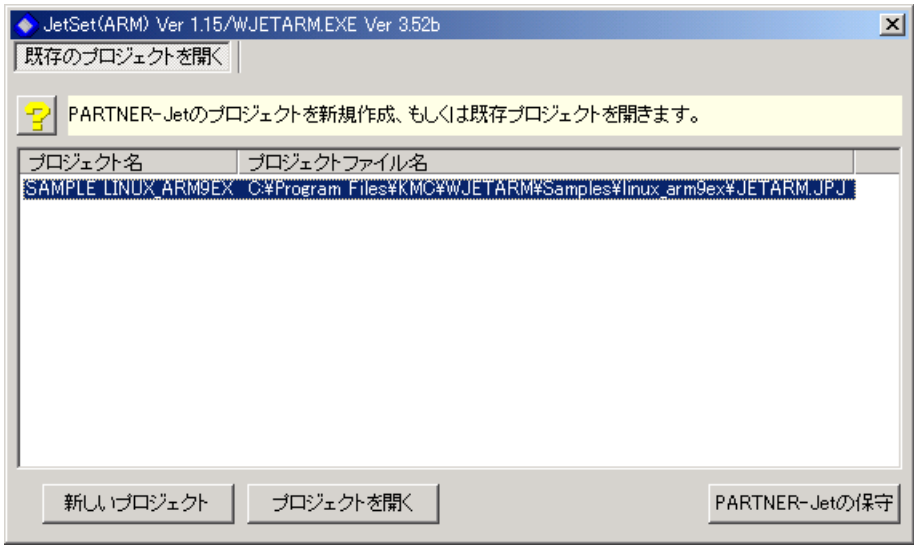
4.2.5 カーネルモードでの PARTNER の起動

カーネルモードで PARTNER を起動するには、次に示す手順で行います。

(1) Jetset でプロジェクトの新規作成もしくは既存のプロジェクトの選択

PARTNER の環境設定プログラム (JETSET(SH)) を起動し、プロジェクトを新規作成もしくは、オープンします。

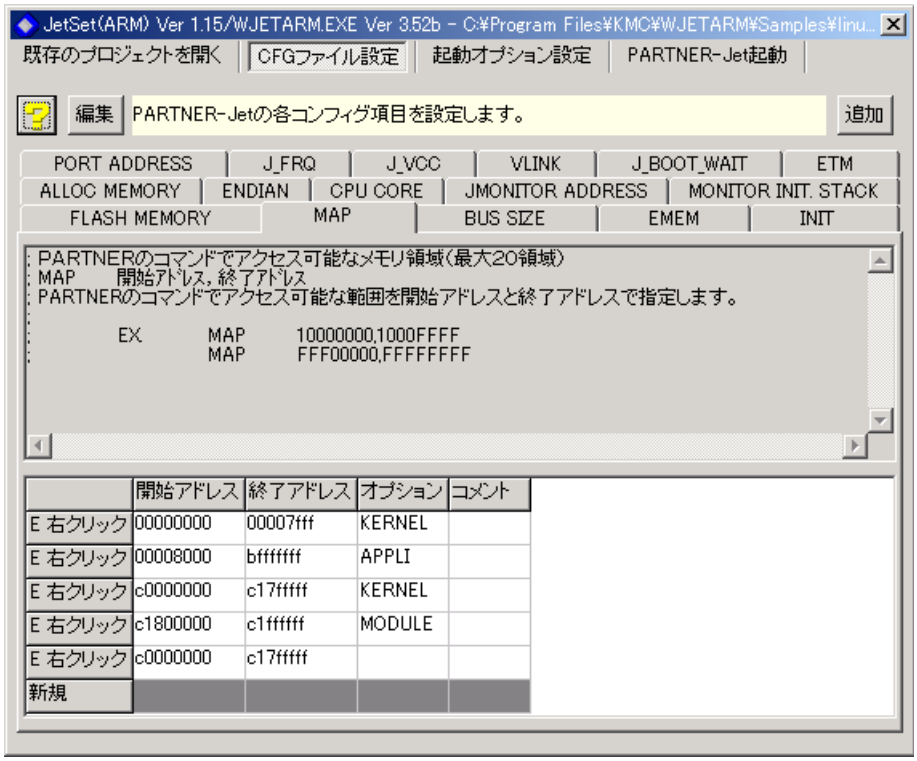
図 4-11 PARTNER プロジェクトのオープン



(2) CFG ファイル設定

MAP フィールドに Linux 用 MAP 情報を指定します。設定するアドレスや属性などについては『CFG ファイルの拡張 (15 頁)』を参照してください。

図 4-12 MAP フィールドの変更



(3) 起動オプション設定

[拡張 >>] ボタンを押し、Linux デバッグ用拡張オプションを指定できるようにします。
カーネルデバッグ時に必ず設定する必要のあるオプションは、以下のとおりです。

● **デバッグ情報バッファサイズ (-B オプション)**

サイズには 100000 程度を指定してください。
もし、カーネルファイルをロードしたときにエラーメッセージ『 デバッグ情報領域がいっぱいです (起動時の **-B オプション**参照)』が表示された場合は、バッファサイズを拡大してください。

● **デバッグ情報タイプ (-XGX オプション)**

『GNU C (Linux etc.)』を選択します。

デバッグ情報パス変換 (-XGX オプション)

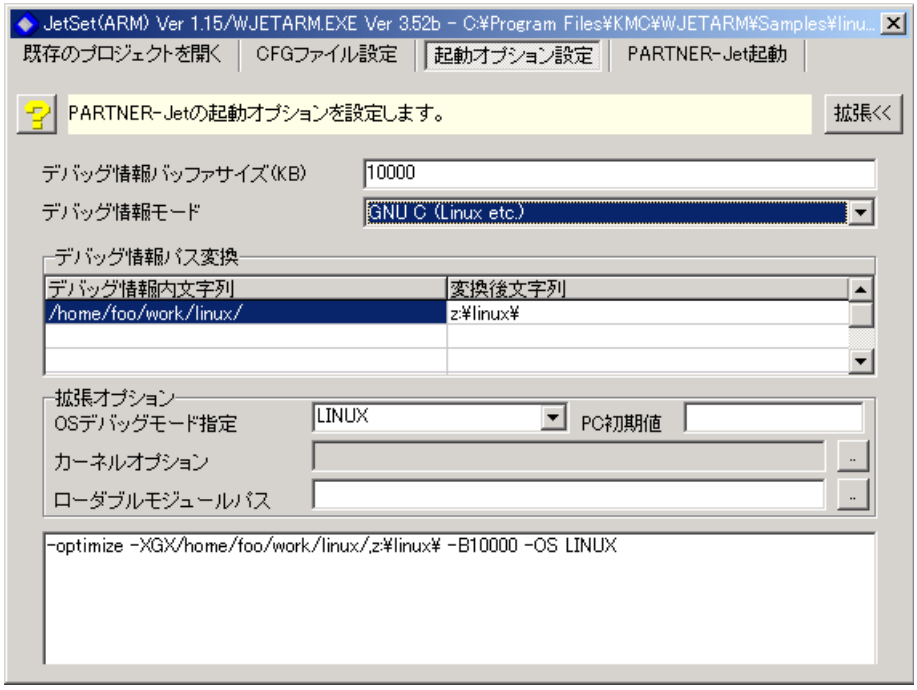
カーネル (vmlinux) をビルドした PATH と Samba でマウントしている PATH が違う場合に指定します。
たとえば、/home/foo/work/linux でカーネルをビルドして、ホスト PC で /home/foo/work を Z: ドライブにマウントした場合は、-XGX/home/foo/work/linux/,z:¥linux¥ と指定します。

● **OS デバッグモード指定 (-OS オプション)**

『LINUX』を選択します。

各オプションについての詳細は、『2.2.2 起動オプション (18 頁)』を参照してください。

図 4-13 起動オプションの設定



(4) PARTNER の起動

JETSET(SH) 上の [**起動**] **ボタン**をクリックすることにより、PARTNER が起動します。

図 4-14 PARTNER の起動

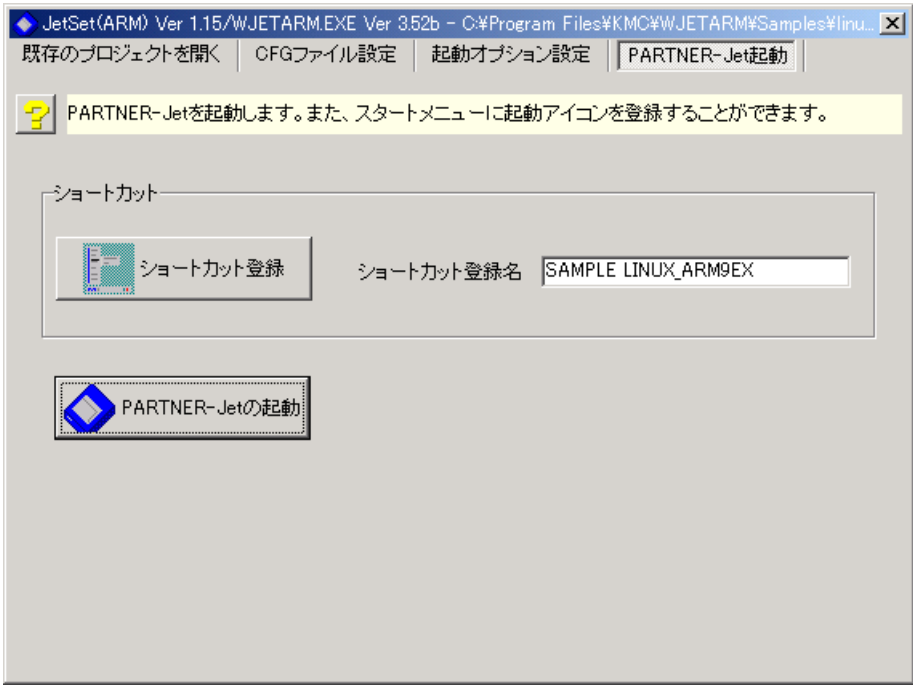
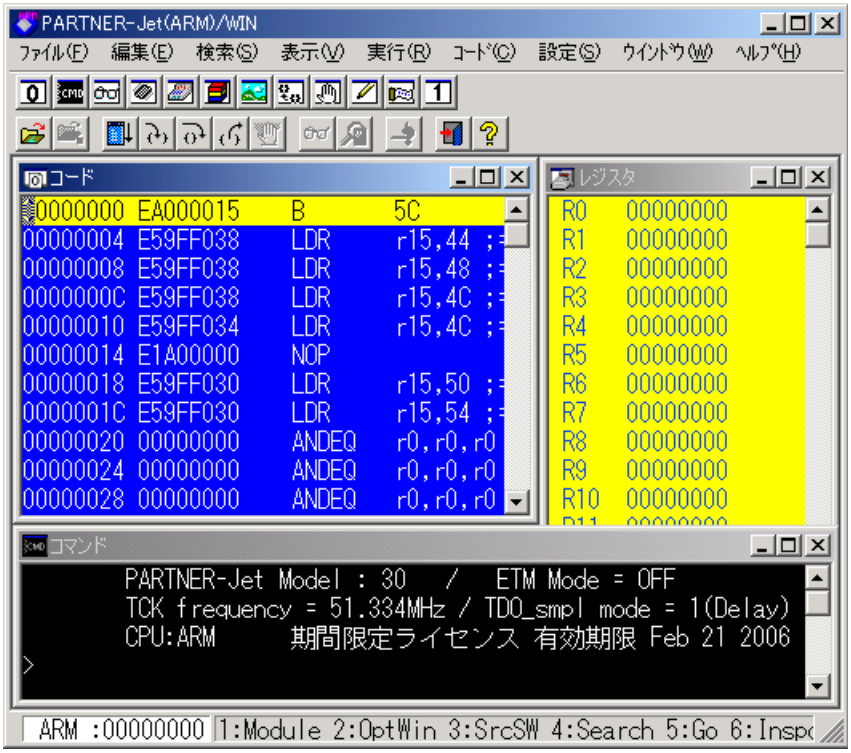


図 4-15 PARTNER の起動画面

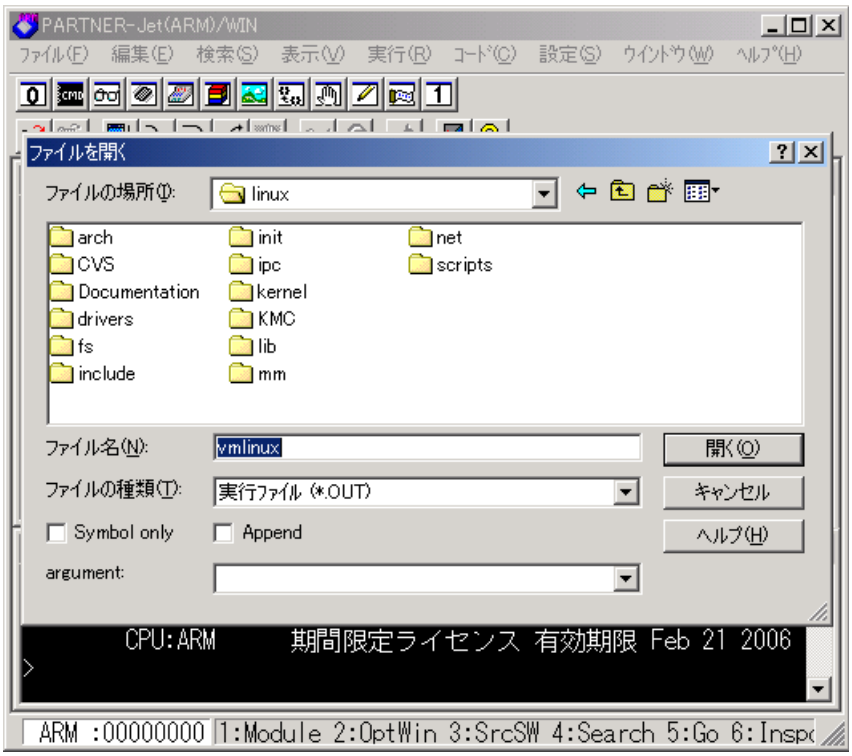


4.2.6 Linux カーネルのロード

『4.1.2 Linux カーネルのビルド (51 頁)』で作成したカーネル(vmlinux)をターゲットメモリにロードします。

```
PT>l vmlinux ↓
```

図 4-16 カーネルファイルのロード



vmlinux から作成された HEX ファイルのバイナリファイルをロードする場合には、以下の手順でカーネルを読み込みます。

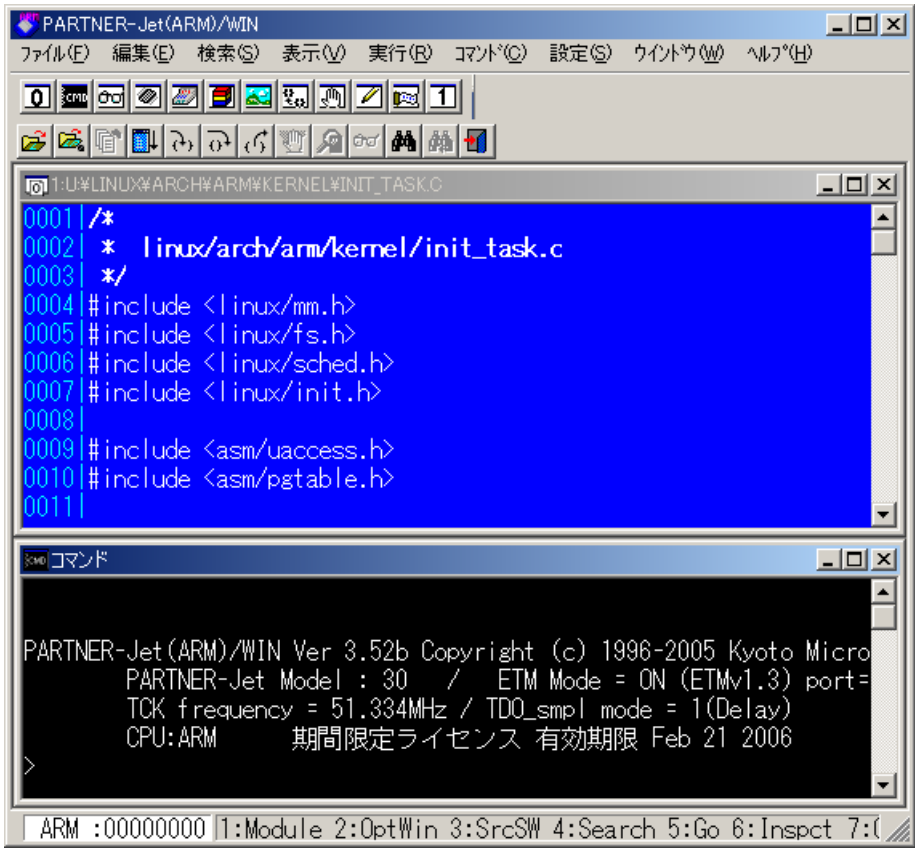
PARTNER のコマンドウィンドウにおいて、次のコマンドを入力します。

【例】rd コマンドで HEX ファイルを読み込み、ls コマンドでデバッグ情報を読み込みます。

```
PT>rd z:¥linux¥vmlinux.hex ↓
PT>ls z:¥linux¥vmlinux ↓
```

コンパイル時にデバッグ情報出力を指定している場合には、カーネルロード時にデバッグ情報も同時にロードされます。
また、一度ロードされたファイルは、PARTNER がロードファイル名と場所を記憶するため、以降の操作を簡略化することができます。

図 4-17 カーネルのロード完了



正常にロードされると、PARTNER のコードウィンドウに Linux のソースコードが表示されます。コードウィンドウ上に何も表示されない場合は、Linux PC のディレクトリ情報を Windows PC から参照可能なディレクトリに変換するパスの変換の設定が正しくない場合があります(『 -XGX オプション（19 頁)』参照)。

4.2.7 Linux カーネルの実行

カーネルのロードが正しくできた状態で、**G コマンド**、または**[実行] ボタン**でロードした vmlinux を実行させます。

```
PT>g ↓
```

Windows PC とターゲットボードが正しく接続されていて、ターミナルソフトが正常に起動していれば、ブートアップメッセージが表示されます。
ブートアップメッセージが正しく表示されない場合やハングアップする場合は、各設定を再確認してください。

図 4-18 カーネルブートアップメッセージ

```
NetWinder Floating Point Emulator V0.95 (c) 1998-1999 Rebel.com
Looking up port of RPC 100003/2 on 192.168.1.241
Looking up port of RPC 100005/1 on 192.168.1.241
VFS: Mounted root (nfs filesystem).
Freeing init memory: 212K
INIT: version 2.78 booting
Activating swap...
Checking all file systems...
Parallelizing fsck version 1.22 (22-Jun-2001)
mkdir: cannot create directory `/dev/pts': File exists
Mounting local filesystems...
nothing was mounted
Cleaning: /etc/network/ifstate.
Setting up IP spoofing protection: rp_filter.
Disable TCP/IP Explicit Congestion Notification: done.
Configuring network interfaces: done.
Starting portmap daemon: portmap.
Cleaning: /tmp /var/lock /var/run.
INIT: Entering runlevel: 3
Starting system log daemon: syslogd klogd.
Starting internet superserver: inetd.

MontaVista Linux 3.0, Professional Edition

kzp-arm login:
```

4.2.8 ローダブルモジュールのインストール

ターゲットシステムでデバッグ対象のローダブルモジュールをインストールします。

```
TGT>insmod rd.o ↓
```

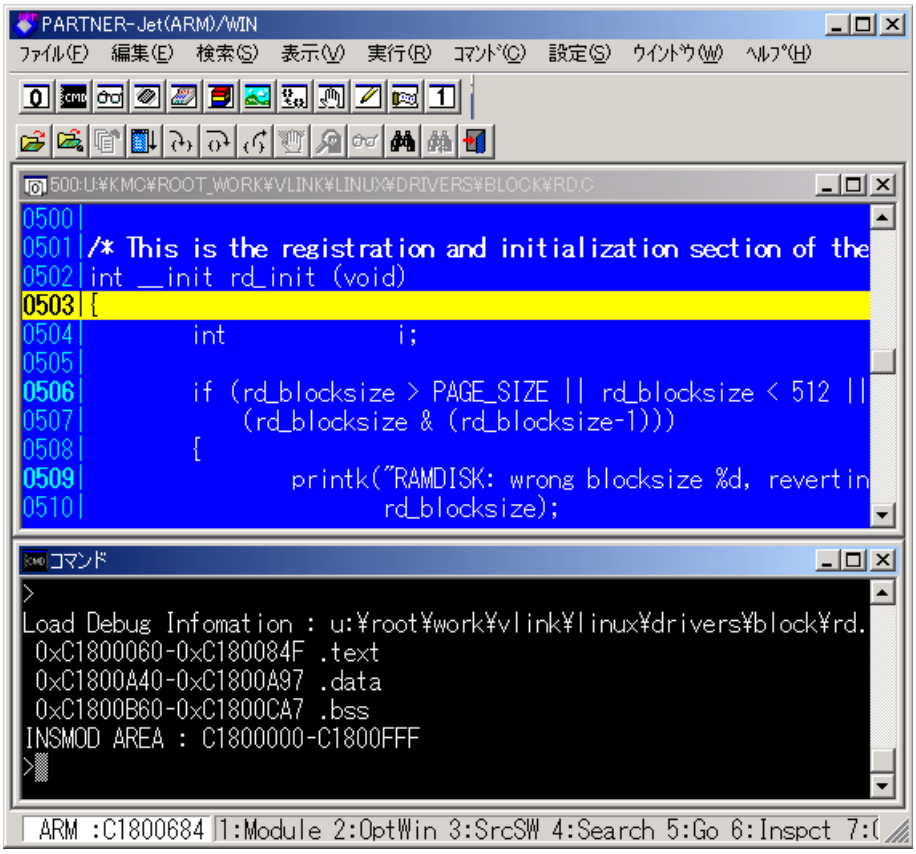
図 4-19 ローダブルモジュールのインストール

```
Parallelizing fsck version 1.22 (22-Jun-2001)
mkdir: cannot create directory `/dev/pts': File exists
Mounting local filesystems...
nothing was mounted
Cleaning: /etc/network/ifstate.
Setting up IP spoofing protection: rp_filter.
Disable TCP/IP Explicit Congestion Notification: done.
Configuring network interfaces: done.
Starting portmap daemon: portmap.
Cleaning: /tmp /var/lock /var/run.
INIT: Entering runlevel: 3
Starting system log daemon: syslogd klogd.
Starting internet superserver: inetd.

MontaVista Linux 3.0, Professional Edition

kzp-arm login: root
Last login: Thu Jan  1 00:00:27 1970 on console
Linux kzp-arm 2.4.18_mvl30-integrator #384 2005年 3月 29日 火曜日 20:06:36 JST a
rmv4l unknown

Welcome to MontaVista Linux 3.0, Professional Edition

root@kzp-arm:~# insmod rd.o
```

4.2.9 PARTNER のブレーク

ターゲットシステムでデバッグ対象のローダブルモジュールがインストールされると、PARTNER は自動的にデバッグ情報を読み込み、module_init() 関数のところでブレークします。このとき、ローダブルモジュールの配置情報も自動的に獲得し、以後ローダブルモジュールエリアのメモリ参照やブレークポイントの設定が可能になります。

図 4-20 PARTNER のブレーク画面



『指定ファイルがありません』のエラーメッセージが表示された場合は、インストールされたローダブルモジュールのファイル PATH の解決に失敗しているか、ローダブルモジュールファイルが存在していません。PATH の解決に失敗している場合は、**-SK オプション** (20 ページ参照) で問題を取り除いてください。

PARTNER のコードウインドウにローダブルモジュールのソースコードが表示されない場合は、デバッグ情報内のソース PATH と PARTNER からアクセスするソース PATH が異なっている可能性があります。**-XGX オプション** (19 ページ参照) でデバッグ情報の PATH 変換を行ってください。

デバッグしているローダブルモジュールを一旦アンロード (rmmod) して、再度、同じローダブルモジュールをインストール (insmod) すると、元々読み込んでいた Linux カーネルのデバッグ情報が失われます。Linux カーネルのデバッグ情報が必要な場合は、lsa コマンドで再度 Linux カーネルのデバッグ情報を読み込んでください。

4.3 カーネル / ローダブルモジュール作成についての補足

Linux カーネルやローダブルモジュールを PARTNER でソースレベルデバッグする場合、最適化コンパイルを行っていると実行可能行がソース記述と食い違ったり、変数スコープが変わったりして、分かりづらい問題があります。また、_init セクションに配置された関数は、ソースレベルデバッグが出来ない問題があります。
そこで、Linux カーネルやローダブルモジュールを最適化なしでコンパイルする時の注意点や _init セクションのデバッグ方法を次に説明します。

最適化を抑制するとカーネルの実行スピードが遅くなります。それによって問題の発生するターゲットシステムの場合は、最適化を抑制するファイルに注意してください。

4.3.1 最適化を抑制する

指定ファイルのみ抑制

対象ソースがあるディレクトリの Makefile に以下の行を追加して最適化オプションを CFLAGS から取り除きます。

```
CFLAGS_????.o := $(filter-out -O2,$(CFLAGS_????.o))
```

【例】

kernel/sched.c の最適化を抑制する場合、kernel/Makefile 内に次の 1 行を追加します。

```
CFLAGS_sched.o := $(filter-out -O2,$(CFLAGS_sched.o))
```

Kernel ツリー全体の抑制

カーネルツリー全体を最適化なしでコンパイルする場合は、$(TOPDIR)/Makefile 内に以下の行を追加して最適化オプションを CFLAGS から取り除きます。

```
CFLAGS := $(filter-out -O2,$(CFLAGS))
```

CFLAGS に指定されている最適化オプションが -O2 ではなく -Os が指定されている場合があります。
その場合は、-Os 文字列を取り除くように変更してください。

作成時の注意事項

カーネルソースには、C ソース内に asm 宣言で直接アセンブリ言語コードを記述している個所があります。最適化に合わせて関数のレジスタ引数を直接アセンブリ言語コードで使用している場合は、正しくカーネルが実行できません。アセンブリ言語コードを修正するか、その関数を C で記述してください。

最適化を抑制してコンパイルした場合、リンク時にシンボル未定義エラーが発生する場合があります。エラーを回避するには次の点をチェックしてください。

- inline 関数が extern 宣言されている場合、static 宣言に変更してください。

```
extern inline func(void)
            ↓
static inline func(void)
```

- プリプロセッサディレクティブ (#if,#ifdef 等) で、_OPTIMIZE_ で判断している個所の最適化なしの場合の記述を追加してください。

第 5 章 アプリケーションのデバッグ手順

この章では、Linux 対応 PARTNER でアプリケーションをデバッグする方法について説明します。

5.1 PARTNER のデバッグモード

PARTNER から見た Linux のアプリケーションの特徴は、「論理多重空間上の仮想アドレスで動作するオブジェクト」ということになります。
プロセス (アプリケーション) は、プロセス 1 つ 1 つに仮想的な論理空間がそれぞれ割り当てられます。一般的には、4G バイトの空間が存在し、その一部分のみが使用されます。
プロセスは、アプリケーション作成時 (リンク時) に決定される仮想アドレス上で動作します。SH を使用する場合では、0x00400000 付近からプロセスが始まり、これはほとんどのプロセスで同じになります (プロセスの先頭アドレスは固定的に決められているということです)。
当然、異なるプロセス同士が同時に同じ物理メモリを使用することはできません。物理的には異なりますが、同じアドレスになるように、Linux カーネルと CPU の MMU でこのような論理空間を生成しています。
PARTNER からは、物理的な CPU とメモリしか見えません。したがって、プロセス (アプリケーション) をデバッグするためには、この仮想アドレスを解決し、同じアドレスで動作するプログラムを識別する情報を PARTNER に伝えることが必要となります。
PARTNER はアプリケーションにデバッグサポートファイル (kmc-support.c) のリンクとソースファイルの一部修正をすることで、アプリケーションの仮想アドレス空間が自動的に解決されます。

PARTNER には、アプリケーションをデバッグするときにいろいろなモードが用意されており、ユーザシステムの環境により最適なデバッグ方法でデバッグすることが可能となっています。

PARTNER でのアプリケーションのデバッグには、カーネルモードとアプリケーションモードの 2 つのデバッグモードが存在します。また、それぞれのデバッグモードには、ADD モードと NON_ADD モードがあります。(『Linux と PARTNER のデバッグモード (12 頁)』参照)

『 3.2 カーネルソース修正(35 頁)』を行っていない場合は、これから説明するアプリケーションのデバッグが出来ません。修正を行っていないカーネルを使用する場合は、『 付録 B 手動アプリケーションデバッグ(162 頁)』を参照してください。

この章では、それぞれのモードでのアプリケーションのデバッグ手順を説明します。

- カーネルモードでのアプリケーションデバッグの手順（73 頁）
- カーネルモードでのマルチプロセス / マルチスレッドアプリケーションデバッグ手順（85 頁）
- アプリケーションモードでのアプリケーションデバッグ（104 頁）
- アプリケーションモードでのマルチプロセス / マルチスレッドアプリケーションのデバッグ（117 頁）

5.2 カーネルモードでのアプリケーションデバッグの手順

カーネルモードでシングルプロセスアプリケーションをデバッグする場合の手順をこの節で説明します。カーネルモードでデバッグする場合、デバッグ中のアプリケーションが停止すると、カーネル、他のアプリケーションプロセスすべてが停止し、停止した時点のカーネルリソースの参照などが可能になります。

アプリケーションモードでアプリケーションのデバッグ方法は、『5.4 アプリケーションモードでのアプリケーションデバッグ（104 頁）』を参照してください。

『 3.2 カーネルソース修正（ 35 頁）』を行っていない場合は、これから説明するアプリケーションのデバッグが出来ません。修正を行っていないカーネルを使用する場合は、『 付録 B 手動アプリケーションデバッグ（ 162 頁）』を参照してください。

この節では、サンプル（sample）を使用した場合を例として、カーネルモードでアプリケーション（プロセス）のデバッグ方法を次の流れで説明します。

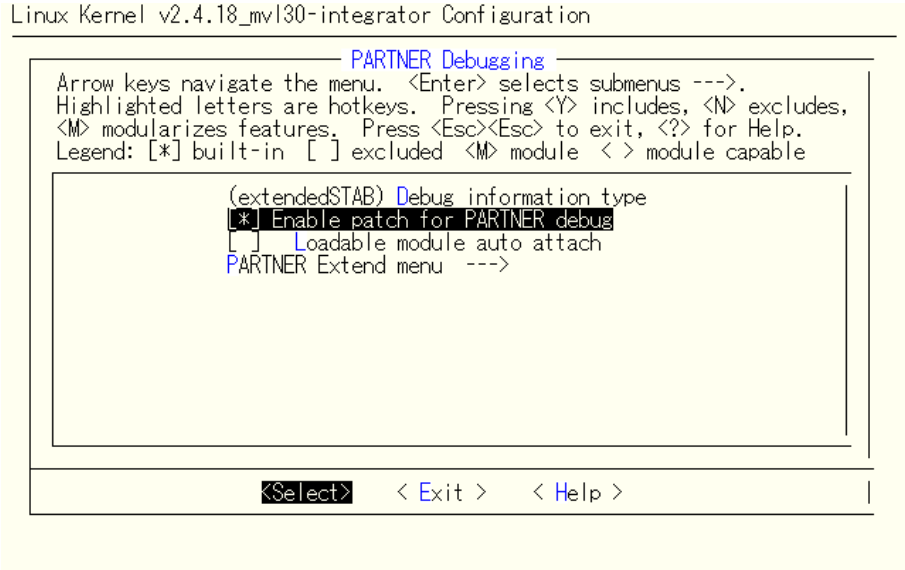
5.2.1 Linux カーネルのコンフィグレーション

カーネルモードでアプリケーションをデバッグする場合は、Linux カーネルのコンフィグレーションで [Enable patch for PARTNER debug] を有効にします。
また、カーネルのデバッグ情報のフォーマットを指定します。
推奨は extendedSTAB(-gstabs+) です。PARTNER で実際カーネルを読み込んでデバッグ情報がおかしなときは、DWARF2 などに変えて試みてください。

```
LINUX86>make menuconfig ↓
```

[Kernel hacking]->[PARTNER Debugging]->[Debug infomation type]
[Kernel hacking]->[PARTNER Debugging]->[Enable patch for PARTNER debug]

図 5-1 Linux カーネルのコンフィグレーション



5.2.2 Linux カーネルのビルド

上記のコンフィグレーションで、Linux カーネル(vmlinux)を作成します。

```
LINUX86>make ↓
```

5.2.3 アプリケーションの作成

デバッグするアプリケーションを作成する場合、以下の手順でアプリケーションを作成します。

(1) アプリケーションソースの修正

アプリケーションソースの main() 関数の先頭にデバッグスタブの呼び出しを挿入します。

```
__kmc_start_debuger(char *program_name);
```

デバッグスタブ関数の引数 char *program_name には、アプリケーションのファイル名が入るようにしてください。
ただし、シンボリックリンクでファイル名が変えられている場合などは、実際のファイル名を文字列で埋め込んでください。
PARTNER は、この引数の文字列とデバッガで読み込んだデバッグ情報内のファイル名を比較してデバッグ対象の場合はデバッガにアタッチします。

【例】

```
int main(int argc,char *argv[])
{
+    __kmc_start_debug(argv[0]);
                    :
                    :
                    :
                    :
```

(2) アプリケーションのリンク

アプリケーションのリンク時にサポートファイル(kmc-support.c)をリンクします。

kmc-support.c 内で "Select Target CPU type" のコンパイルエラーが発生した場合は、kmc-support.c 内の CPUTYPE シンボル宣言をターゲット CPU のみ有効にして再コンパイルしてください。

なおアプリケーションの作成時に、カーネルコンフィグレーションで指定したデバッグ情報を付加するようにしてください。

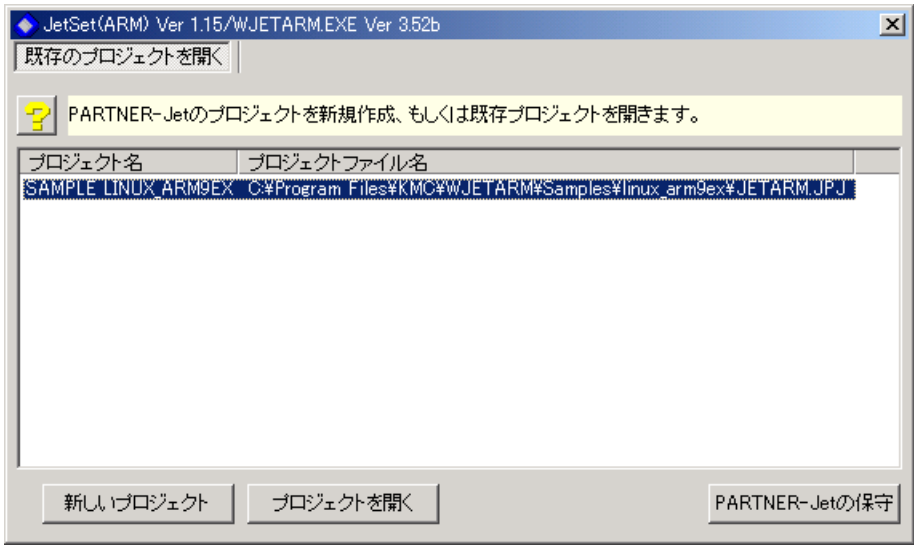
5.2.4 カーネルモードでの PARTNER の起動

カーネルモードで PARTNER を起動するには、次に示す手順で行います。

(1) Jetset でプロジェクトの新規作成もしくは既存のプロジェクトの選択

PARTNER の環境設定プログラム (JETSET(SH)) を起動し、プロジェクトを新規作成もしくは、オープンしてます。

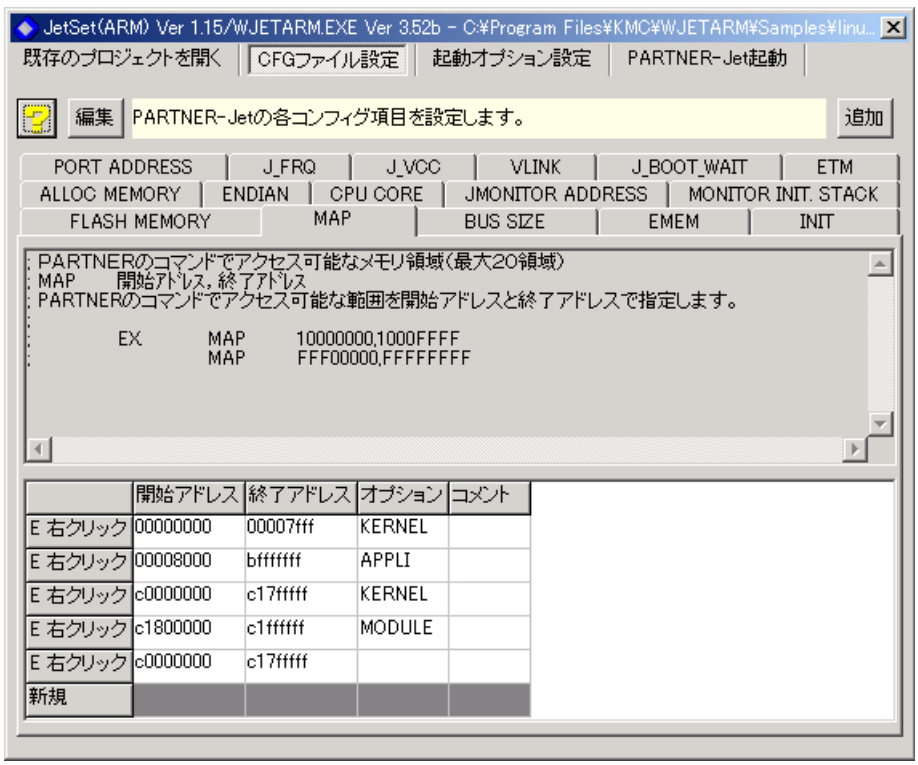
図 5-2 PARTNER プロジェクトのオープン



(2) CFG ファイル設定

MAP フィールドに Linux 用 MAP 情報を指定します。設定するアドレスや属性などについては『CFG ファイルの拡張 (15 頁)』を参照してください。

図 5-3 MAP フィールドの変更



(3) 起動オプション設定

[拡張 >>] ボタンを押し、Linux デバッグ用拡張オプションを指定できるようにします。
カーネルデバッグ時に必ず設定する必要のあるオプションは、以下のとおりです。

● **デバッグ情報バッファサイズ (-B オプション)**

サイズには 100000 程度を指定してください。
もし、カーネルファイルをロードしたときにエラーメッセージ『 デバッグ情報領域がいっぱいです (起動時の **-B オプション**参照)』が表示された場合は、バッファサイズを拡大してください。

● **デバッグ情報タイプ (-XGX オプション)**

『GNU C (Linux etc.)』を選択します。

デバッグ情報パス変換 (-XGX オプション)

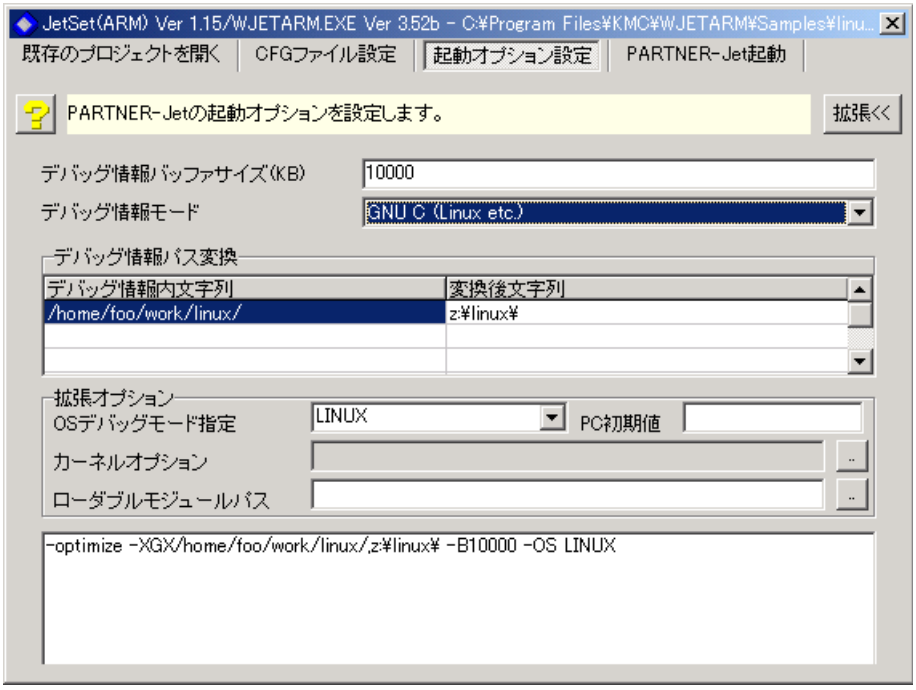
カーネル (vmlinux) をビルドした PATH と Samba でマウントしている PATH が違う場合に指定します。
たとえば、/home/foo/work/linux でカーネルをビルドして、ホスト PC で /home/foo/work を Z: ドライブにマウントした場合は、-XGX/home/foo/work/linux/,z:¥linux¥ と指定します。

● **OS デバッグモード指定 (-OS オプション)**

『LINUX』を選択します。

各オプションについての詳細は、『2.2.2 起動オプション (18 頁)』を参照してください。

図 5-4 起動オプションの設定



(4) PARTNER の起動

JETSET(SH) 上の [起動] ボタンをクリックすることにより、PARTNER が起動します。

図 5-5 PARTNER の起動

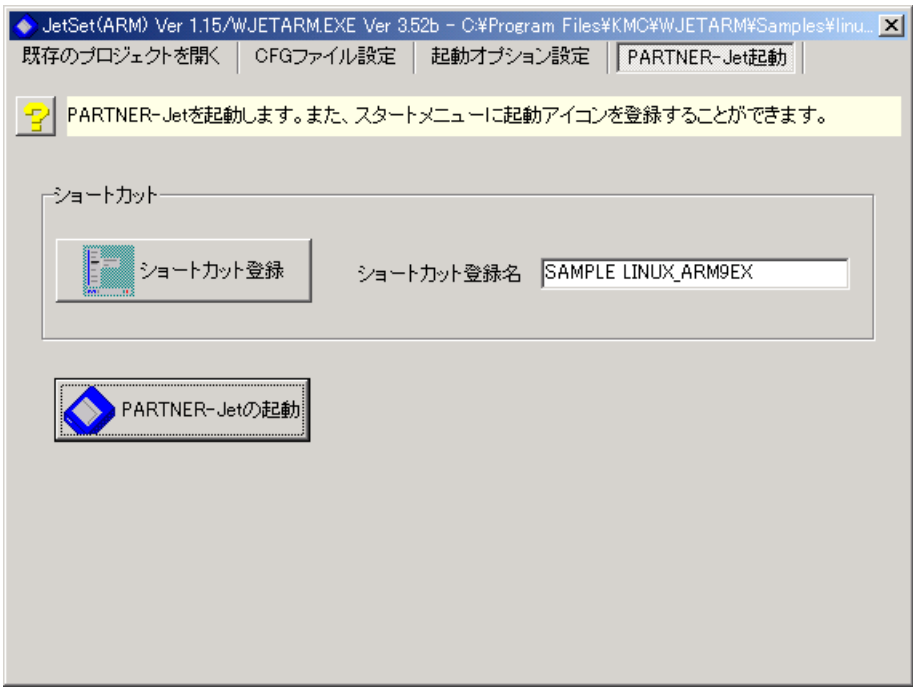
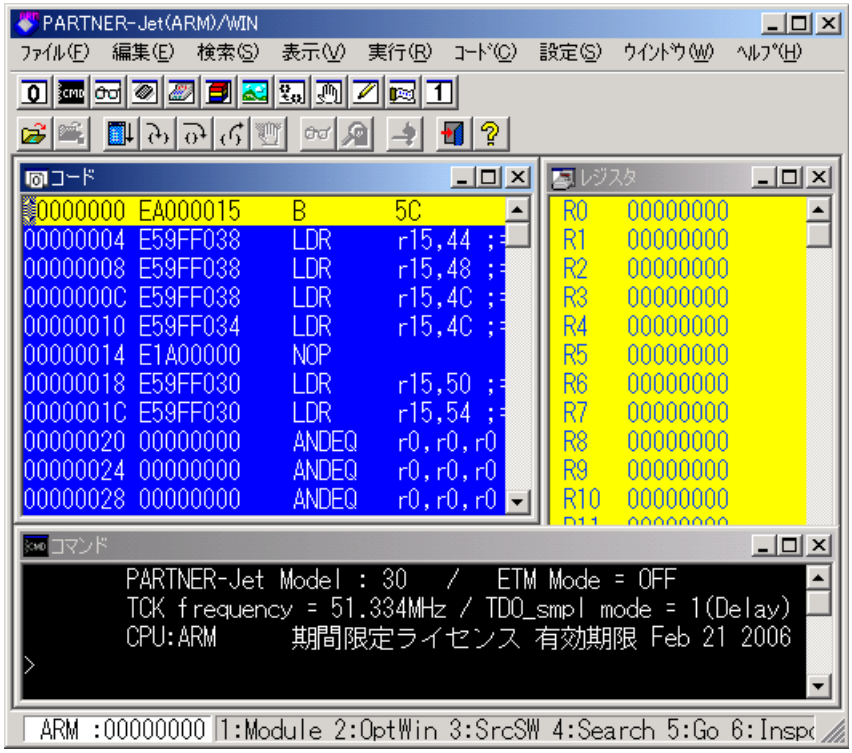


図 5-6 PARTNER の起動画面

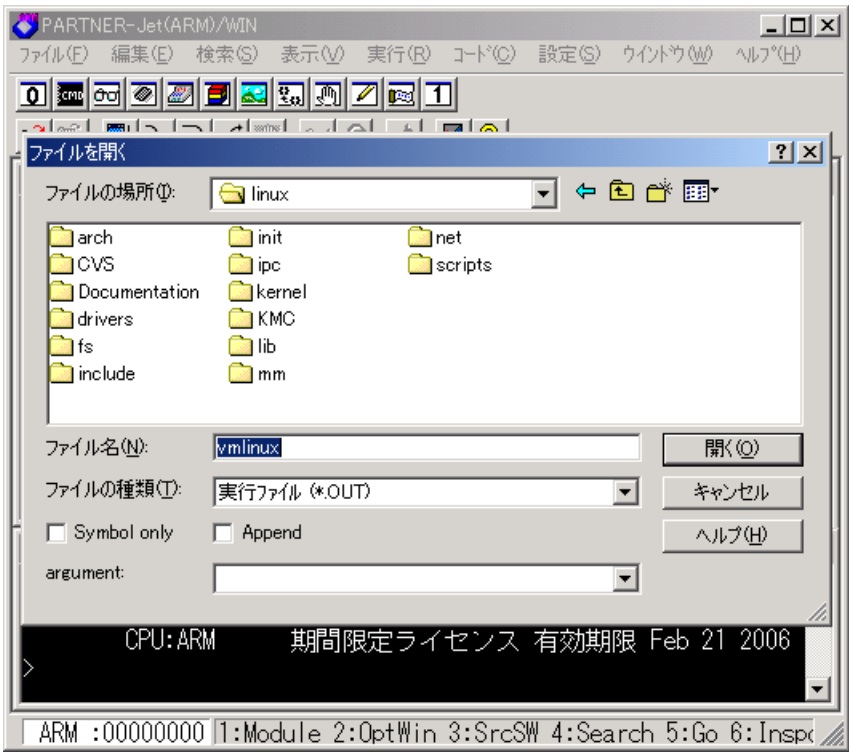


5.2.5 Linux カーネルのロード

『5.2.2 Linux カーネルのビルド (74 頁)』で作成したカーネル(vmlinux)をターゲットメモリにロードします。

```
PT>l vmlinux ↓
```

図 5-7 カーネルのロード



vmlinux から作成された HEX ファイルのバイナリファイルをロードする場合には、以下の手順でカーネルを読み込みます。

PARTNER のコマンドウィンドウにおいて、次のコマンドを入力します。

【例】rd コマンドで HEX ファイルを読み込み、ls コマンドでデバッグ情報を読み込みます。

```
PT>rd z:¥linux¥vmlinux.hex ↓
PT>ls z:¥linux¥vmlinux ↓
```

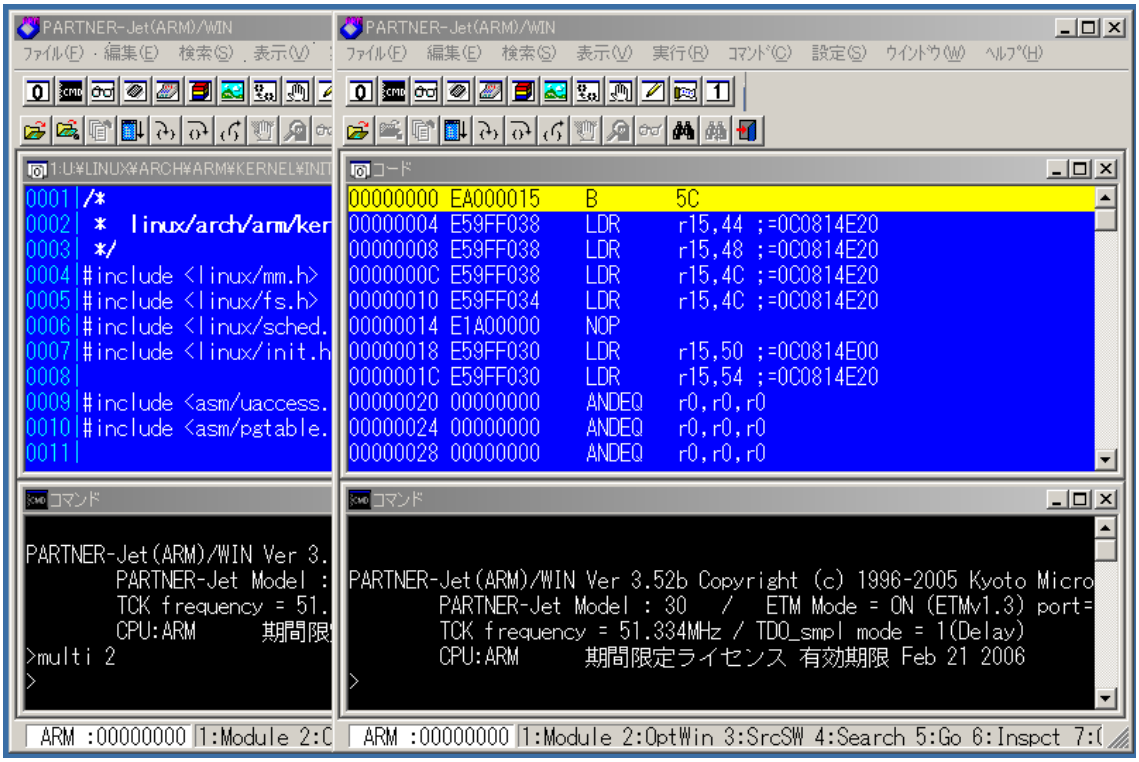
コンパイル時にデバッグ情報出力を指定している場合には、カーネルロード時にデバッグ情報も同時にロードされます。
また、一度ロードされたファイルは、PARTNER がロードファイル名と場所をダイアログとコマンドヒストリに記憶するため、以降の操作を簡略化することができます。

5.2.6 マルチウインドウデバッグ

Linux カーネルとアプリケーションを別の PARTNER ウインドウでデバッグする場合は、**MULTI コマンド** (22 ページ参照) で複数の PARTNER ウインドウを起動します。サンプルプログラム (sample) をデバッグする場合は、カーネル用 PARTNER ウインドウとアプリケーション用 PARTNER ウインドウの 2 つの PARTNER ウインドウを起動します。
カーネルをロードした PARTNER ウインドウでアプリケーションをデバッグする場合は、PARTNER ウインドウを起動する必要はありません。

```
PT>multi 2 ↓
```

図 5-8 複数 PARTNER ウインドウの起動



MULTI コマンド (22 ページ参照) や -MULTI オプション (20 ページ参照) で起動した PARTNER ウインドウのウインドウ情報 (ウインドウ配置等) やコマンドヒストリは、新規に生成されたプロジェクトファイル (jetsh_?.jpj) に保存され、次回起動時に利用されます。

5.2.7 Linux カーネルの実行

カーネルのロードが正しくできた状態で、**G コマンド**、または**[実行] ボタン**でロードした vmlinux を実行させます。

```
PT>g ↓
```

Windows PC とターゲットボードが正しく接続されていて、ターミナルソフトが正常に起動していれば、ブートアップメッセージが表示されます。

図 5-9 カーネルブートアップメッセージ

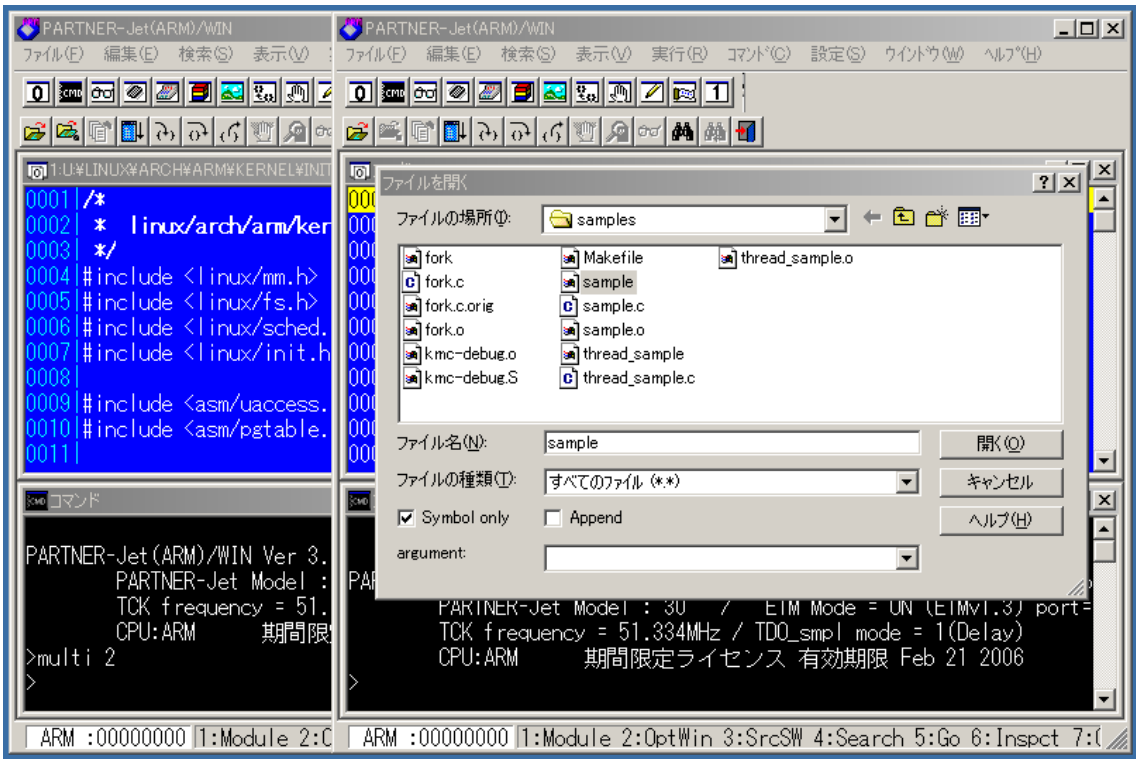
```
NetWinder Floating Point Emulator V0.95 (c) 1998-1999 Rebel.com
Looking up port of RPC 100003/2 on 192.168.1.241
Looking up port of RPC 100005/1 on 192.168.1.241
VFS: Mounted root (nfs filesystem).
Freeing init memory: 212K
INIT: version 2.78 booting
Activating swap...
Checking all file systems...
Parallelizing fsck version 1.22 (22-Jun-2001)
mkdir: cannot create directory `/dev/pts': File exists
Mounting local filesystems...
nothing was mounted
Cleaning: /etc/network/ifstate.
Setting up IP spoofing protection: rp_filter.
Disable TCP/IP Explicit Congestion Notification: done.
Configuring network interfaces: done.
Starting portmap daemon: portmap.
Cleaning: /tmp /var/lock /var/run.
INIT: Entering runlevel: 3
Starting system log daemon: syslogd klogd.
Starting internet superserver: inetd.

MontaVista Linux 3.0, Professional Edition

kzp-arm login:
```

5.2.8 アプリケーションデバッグ情報のロード

作成したアプリケーションのデバッグ情報を PARTNER にロードします。
ロード時には必ず [Symbol only] のチェックを行ってください。また、複数のデバッグ情報を一つの PARTNER ウインドウにロードする場合は、[Append] のチェックを行ってください。

```
PT>ls sample ↓
PT>lsa sample ↓
```

図 5-10 アプリケーションデバッグ情報のロード



一度ロードされたファイルは、PARTNER がロードファイル名と場所をダイアログとコマンドヒストリに記憶するため、以降の操作を簡略化することができます。

5.2.9 アプリケーションの実行

ターゲットシステムでデバッグ対象のアプリケーションを実行します。

```
TGT>./sample ↓
```

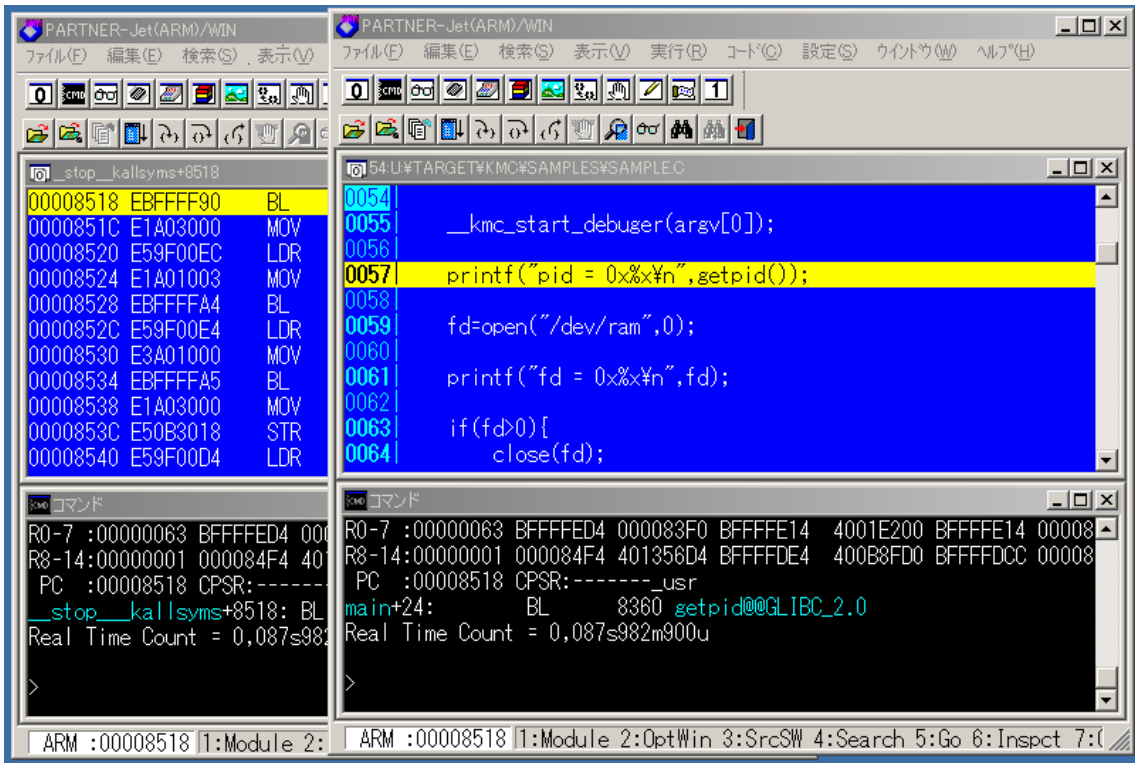
図 5-11 アプリケーションの実行

```
Parallelizing fsck version 1.22 (22-Jun-2001)
mkdir: cannot create directory `/dev/pts': File exists
Mounting local filesystems...
nothing was mounted
Cleaning: /etc/network/ifstate.
Setting up IP spoofing protection: rp_filter.
Disable TCP/IP Explicit Congestion Notification: done.
Configuring network interfaces: done.
Starting portmap daemon: portmap.
Cleaning: /tmp /var/lock /var/run.
INIT: Entering runlevel: 3
Starting system log daemon: syslogd klogd.
Starting internet superserver: inetd.

MontaVista Linux 3.0, Professional Edition

kzp-arm login: root
Last login: Thu Jan  1 00:00:19 1970 on console
Linux kzp-arm 2.4.18_mvl30-integrator #384 2005年 3月 29日 火曜日 20:06:36 JST a
rmv4l unknown

Welcome to MontaVista Linux 3.0, Professional Edition

root@kzp-arm:~# ./sample
```

5.2.10 PARTNER のブレーク

ターゲットシステムでデバッグ対象のアプリケーションが実行されると、PARTNER はデバッグスタブ関数(__kmc_start_debuger())の後ろでブレークします。このとき、アプリケーションの PID や配置情報を自動的に獲得し、以後アプリケーションエリアのメモリ参照やブレークポイントの設定が可能になります。

図 5-12 アプリケーションのアタッチ直後



PSID コマンド(25 ページ参照)でデバッガにアプリケーションがアタッチされているか確認します。

```
PT>psid ↓
PSID SET 103(0x67)  CURRENT 103(0x67)
APPLI. AREA : 00400000-00401FFF
APPLI. AREA : 00411000-00451FFF
APPLI. AREA : 00453000-00453FFF
APPLI. AREA : 7BFFF000-7BFFFFFF
```

PARTNERが正しくブレークしない場合､カーネルのコンフィグレーションが間違っている可能性があります『 5.2.1 Linux カーネルのコンフィグレーション(74 頁)』の設定と『 5.2.4 カーネルモードでの PARTNER の起動(76 頁)』で指定した OS デバッグモードがカーネルモードになっているか確認してください。
また、アプリケーションソースにデバッグスタブ関数(_kmc_start_debuger()) が挿入されているか、サポートファイル(kmc-support.c)がリンクされているか確認してください。

5.3 カーネルモードでのマルチプロセス / マルチスレッドアプリケーションデバッグ手順

カーネルモードでマルチプロセス / マルチスレッドアプリケーションをデバッグする場合の手順をこの節で説明します。
fork() や pthread を使用したアプリケーションのデバッグ手順は前述の『5.2 カーネルモードでのアプリケーションデバッグの手順（73 頁）』の手順とほぼ同じで、カーネルのコンフィグレーション、アプリケーションの作成と起動オプションの指定、マルチウインドウデバッグが異なるだけです。

マルチプロセス / マルチスレッドアプリケーションのデバッグには、各プロセス / スレッドを同じ PARTNER ウインドウでデバッグする ADD モードと、それぞれ別の PARTNER ウインドウでデバッグする NON_ADD モードがあります。
ADD モードと NON_ADD モードの切り替えは、**PSID コマンド**（25 ページ参照）または、**-OS オプション**（18 ページ参照）で指定します。

アプリケーションモードでのマルチプロセス / マルチスレッドアプリケーションのデバッグ方法は、『5.5 アプリケーションモードでのマルチプロセス / マルチスレッドアプリケーションのデバッグ（117 頁）』を参照してください。

『 3.2 カーネルソース修正（ 35 頁）』を行っていない場合は、これから説明するアプリケーションのデバッグが出来ません。修正を行っていないカーネルやアプリケーションを使用する場合は、『 付録 C 手動マルチプロセス / マルチスレッド 対応のデバッグ方法（ 169 頁）』を参照してください。

この節では、サンプル（fork/thread_sample）を使用した場合を例として、カーネルモードでのマルチプロセス / マルチスレッドアプリケーションのデバッグ方法を次の流れで説明します。

(1) Linux カーネルのコンフィグレーション（86 頁）
(2) Linux カーネルのビルド（86 頁）
(3) アプリケーションの作成（87 頁）
(4) カーネルモードでの PARTNER の起動（89 頁）
(5) Linux カーネルのロード（92 頁）
(6) マルチウインドウデバッグ（93 頁）
(7) Linux カーネルの実行（95 頁）
(8) アプリケーションのロード（96 頁）
(9) アプリケーションの実行（98 頁）
(10) PARTNER のブレーク（99 頁）

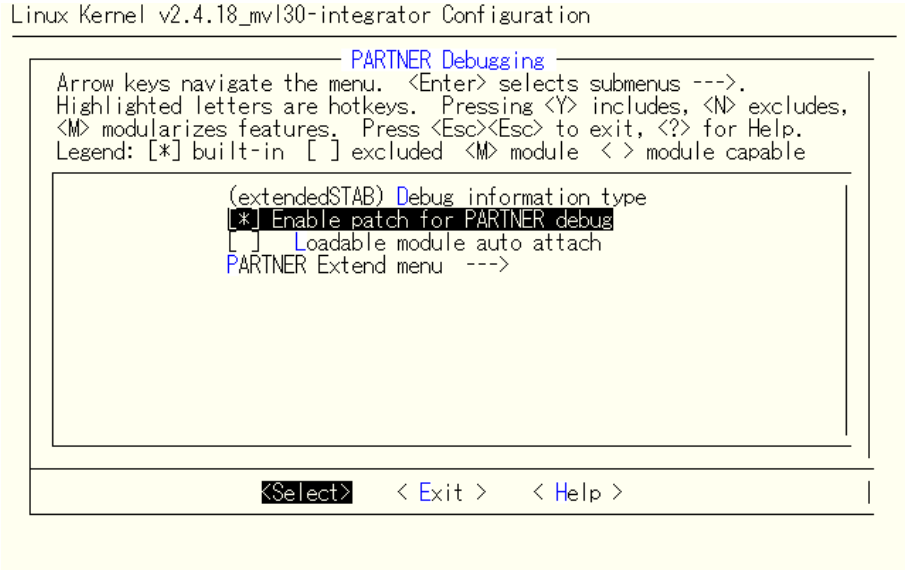
5.3.1 Linux カーネルのコンフィグレーション

カーネルモードでマルチプロセス / マルチスレッドアプリケーションをデバッグする場合は、Linux カーネルのコンフィグレーションで [Enable patch for PARTNER debug] を有効にします。
また、カーネルのデバッグ情報のフォーマットを指定します。
推奨は extendedSTAB(-gstabs+) です。PARTNER で実際カーネルを読み込んでデバッグ情報がおかしなときは、DWARF2 などに変えて試みてください。

```
LINUX86>make menuconfig ↓
```

[Kernel hacking]->[PARTNER Debugging]->[Debug infomation type]
[Kernel hacking]->[PARTNER Debugging]->[Enable patch for PARTNER debug]

図 5-13 Linux カーネルのコンフィグレーション



5.3.2 Linux カーネルのビルド

上記のコンフィグレーションで、Linux カーネル(vmlinux)を作成します。

```
LINUX86>make ↓
```

5.3.3 アプリケーションの作成

デバッグするアプリケーションを作成する場合、以下の手順でアプリケーションを作成します。

(1) アプリケーションソースの修正

【マルチスレッドアプリケーション(thread_sample)の場合】

アプリケーションソースの main() 関数の先頭とスレッドボディ関数の先頭でデバッグスタブの呼び出しを挿入します。

```
__kmc_start_debuger(char *program_name);
```

main() 関数直後に挿入するデバッグスタブ関数の引数 char *program_name には、アプリケーションのファイル名が入るようにしてください。
ただし、シンボリックリンクでファイル名が変えられている場合などは、実際のファイル名を文字列で埋め込んでください。
PARTNER は、この引数の文字列とデバッガで読み込んだデバッグ情報内のファイル名を比較してデバッグ対象と一致した場合はデバッガにアタッチします。
スレッドボディ関数の先頭に挿入するデバッグスタブ関数の引数 char *program_name には、0 を指定してください。

【例】

```
int main(int argc,char *argv[])
{
+    __kmc_start_debuger(argv[0]);
                          :
                          :
    pthread_create(&th,NULL,thread_func,NULL);
                          :
                          :
void *thread_func(void *)
{
+    __kmc_start_debuger(0);
                          :
                          :
```

【マルチプロセスアプリケーション(fork)の場合】

アプリケーションソースの main() 関数の先頭と子プロセスの先頭(fork() 関数の子プロセス側の戻り)でデバッグスタブの呼び出しを挿入します。

```
__kmc_start_debuger(char *program_name);
```

main() 関数直後に挿入するデバッグスタブ関数の引数 char *program_name には、アプリケーションのファイル名が入るようにしてください。
ただし、シンボリックリンクでファイル名が変えられている場合などは、実際のファイル名を文字列で埋め込んでください。
PARTNER は、この引数の文字列とデバッガで読み込んだデバッグ情報内のファイル名を比較してデバッグ対象と一致した場合はデバッガにアタッチします。
子プロセスの先頭に挿入するデバッグスタブ関数の引数 char *program_name には、0 を指定してください。

【例】

```
int main(int argc,char *argv[])
{
+    __kmc_start_debuger(argv[0]);
                        :
                        :
    if(fork()==0){
+       __kmc_start_debuger(0);
                        :
                        :
    }
                        :
                        :
```

(2) アプリケーションのリンク

アプリケーションのリンク時にサポートファイル(kmc-support.c)をリンクします。

kmc-support.c 内で "Select Target CPU type" のコンパイルエラーが発生した場合は、kmc-support.c 内の CPUTYPE シンボル宣言をターゲット CPU のみ有効にして再コンパイルしてください。

なおアプリケーションの作成時に、デバッグ情報を付加するようにしてください。

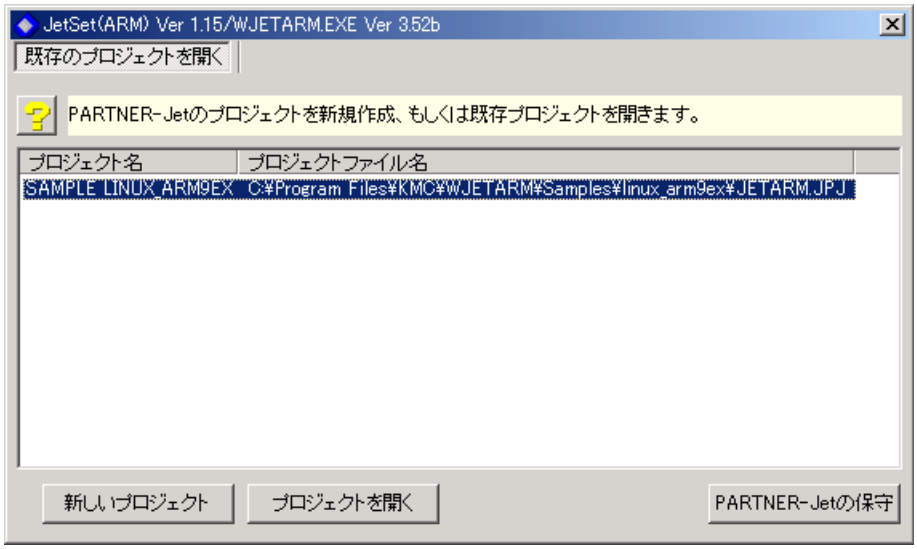
5.3.4 カーネルモードでの PARTNER の起動

カーネルモードで PARTNER を起動するには、次に示す手順で行います。

(1) Jetset でプロジェクトの新規作成もしくは既存のプロジェクトの選択

PARTNER の環境設定プログラム (JETSET(SH)) を起動し、プロジェクトを新規作成もしくは、オープンしてます。

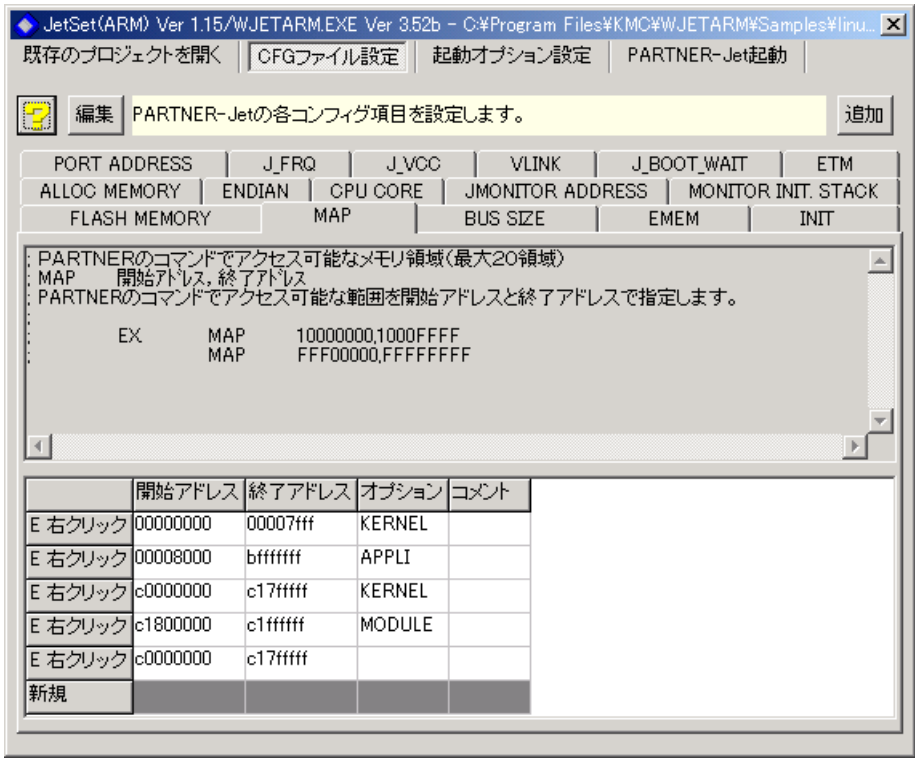
図 5-14 PARTNER プロジェクトのオープン



(2) CFG ファイル設定

MAP フィールドに Linux 用 MAP 情報を指定します。設定するアドレスや属性などについては『CFG ファイルの拡張 (15 頁)』を参照してください。

図 5-15 MAP フィールドの設定



(3) 起動オプション設定

[拡張 >>] ボタンを押し、Linux デバッグ用拡張オプションを指定できるようにします。
カーネルデバッグ時に必ず設定する必要のあるオプションは、以下のとおりです。

● **デバッグ情報バッファサイズ (-B オプション)**

サイズには 100000 程度を指定してください。
もし、カーネルファイルをロードしたときにエラーメッセージ『 デバッグ情報領域がいっぱいです (起動時の **-B オプション**参照)』が表示された場合は、バッファサイズを拡大してください。

● **デバッグ情報タイプ (-XGX オプション)**

『GNU C (Linux etc.)』を選択します。

デバッグ情報パス変換 (-XGX オプション)

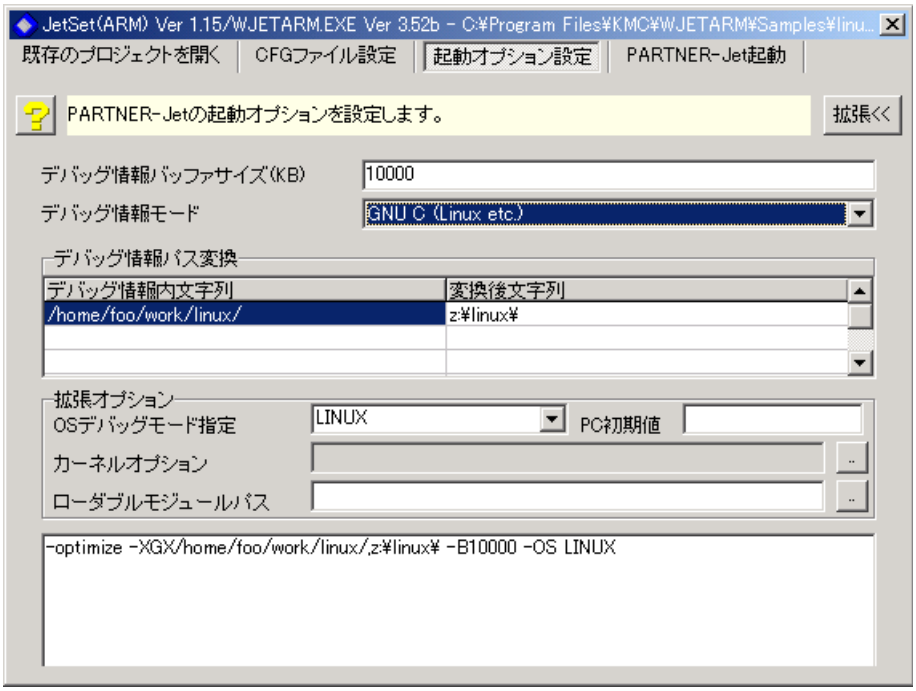
カーネル (vmlinux) をビルドした PATH と Samba でマウントしている PATH が違う場合に指定します。
たとえば、/home/foo/work/linux でカーネルをビルドして、ホスト PC で /home/foo/work を Z: ドライブにマウントした場合は、-XGX/home/foo/work/linux/,z:¥linux¥ と指定します。

● **OS デバッグモード指定 (-OS オプション)**

『LINUX』を選択します。
また、ADD モードでデバッグする場合は、『LINUX_ADD』オプションを選択してください。
ADD モードは PARTNER 起動後、**PSID コマンド** (25 ページ参照) でも切り替えることが可能です。

各オプションについての詳細は、『2.2.2 起動オプション (18 頁)』を参照してください。

図 5-16 起動オプションの設定



(4) PARTNER の起動

JETSET(SH) 上の [起動] ボタンをクリックすることにより、PARTNER が起動します。

図 5-17 PARTNER の起動

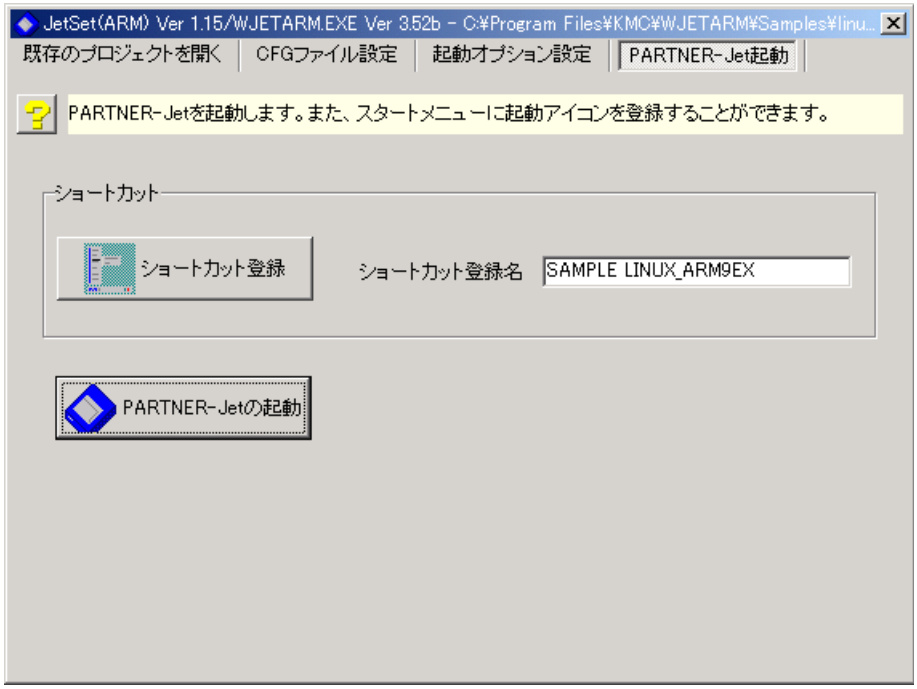
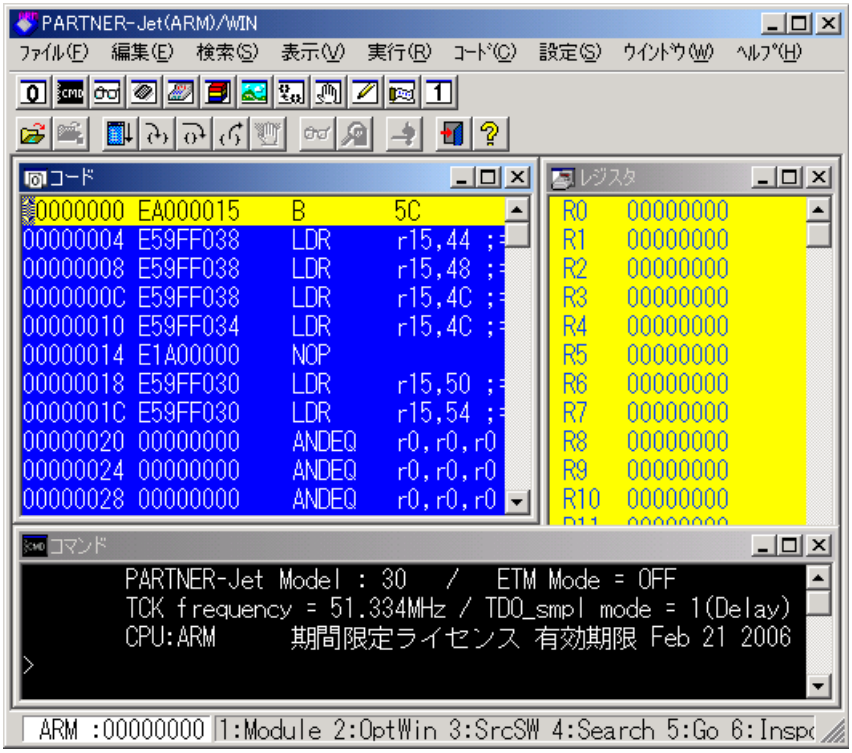


図 5-18 PARTNER の起動画面

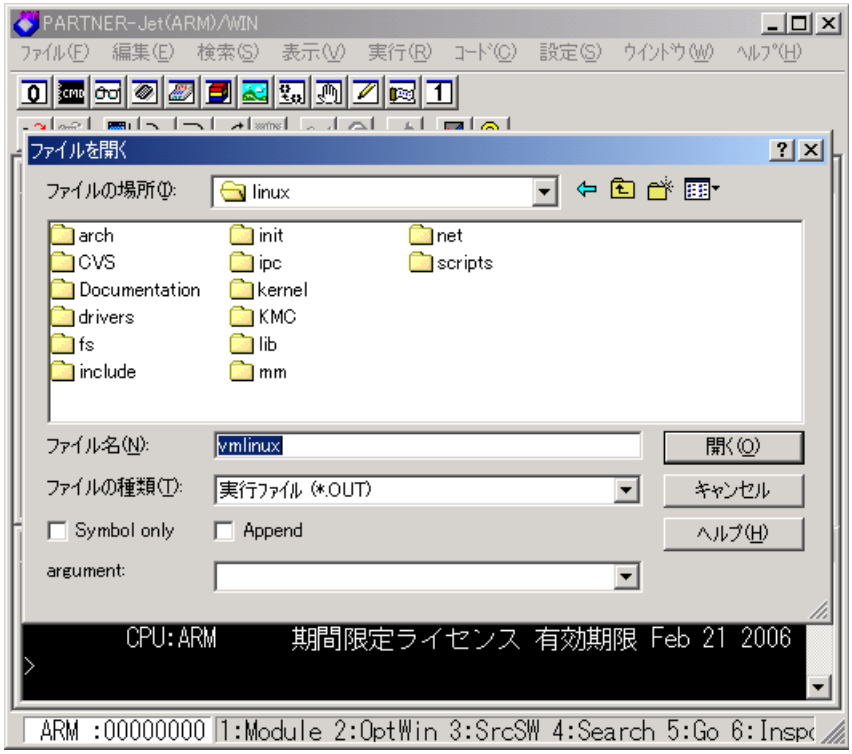


5.3.5 Linux カーネルのロード

『5.3.2 Linux カーネルのビルド (86 頁)』で作成したカーネル(vmlinux)をターゲットメモリにロードします。

```
PT>l vmlinux ↓
```

図 5-19 カーネルのロード



5.3.6 マルチウインドウデバッグ

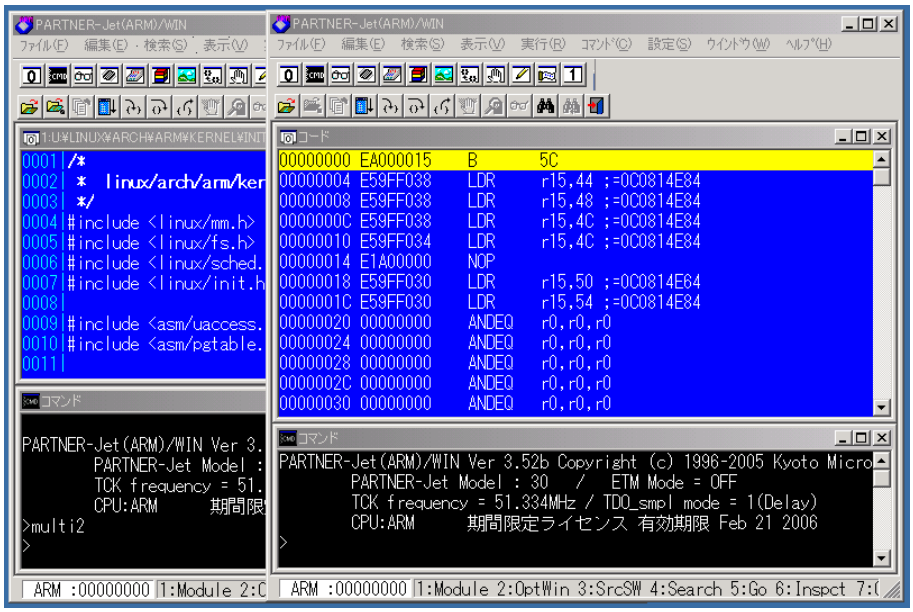
アプリケーション用 PARTNER ウインドウを起動します。

● ADD モードの場合

ADD モードの場合は、デバッグ対象アプリケーションから生成されるプロセス / スレッドはアプリケーション用 PARTNER ウインドウにアタッチされるため、アプリケーション用 PARTNER ウインドウを 1 つ追加起動します。

```
PT>multi 2 ↓
```

図 5-20 アプリケーション用 PARTNER ウインドウの起動 (ADD モード)



MULTI コマンド (22 ページ参照) や -MULTI オプション (20 ページ参照) で起動した PARTNER ウインドウのウインドウ情報 (ウインドウ配置等) やコマンドヒストリは、新規に生成されたプロジェクトファイル (jetsh_?.jpj) に保存され、次回起動時に利用されます。

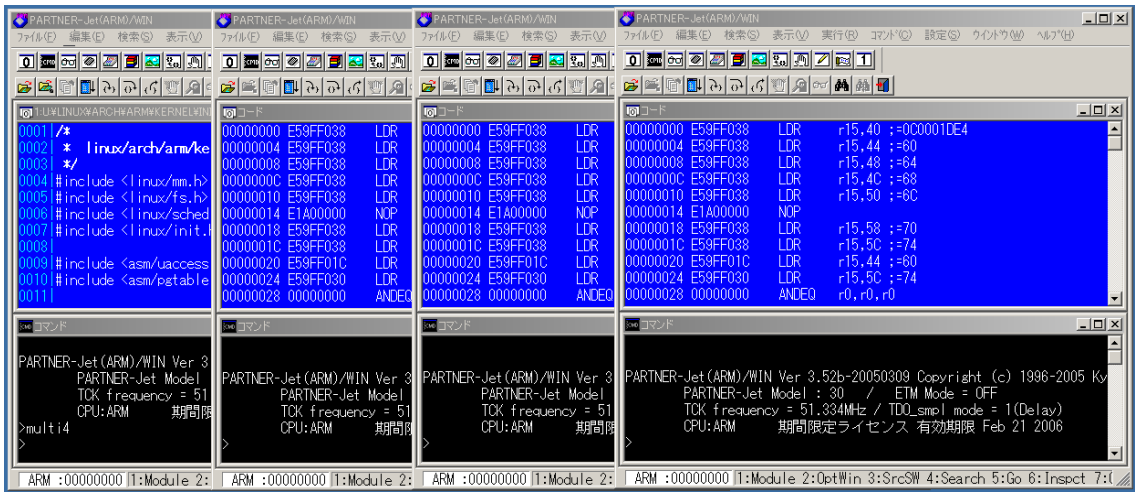
● NON_ADD モードの場合

ADD モードでない場合は、アプリケーション＋生成されるスレッド / プロセスの数の PARTNER ウインドウを起動します。

例えばサンプル thread_sample の場合、カーネル用、アプリケーション用、生成されるスレッド 2 つ用の 4 つのウインドウが必要となるため、4 つの PARTNER ウインドウを起動します。

PT>multi 4 ↓

図 5-21 アプリケーション / スレッド用 PARTNER ウインドウの起動 (NON_ADD モード)



MULTI コマンド (22 ページ参照) や -MULTI オプション (20 ページ参照) で起動した PARTNER ウインドウのウインドウ情報 (ウインドウ配置等) やコマンドヒストリは、新規に生成されたプロジェクトファイル (jetsh_?.jpj) に保存され、次回起動時に利用されます。

5.3.7 Linux カーネルの実行

カーネルのロードが正しくできた状態で、**G コマンド**、または **[実行] ボタン**でロードした vmlinux を実行させます。

```
PT>g ↓
```

Windows PC とターゲットボードが正しく接続されていて、ターミナルソフトが正常に起動していれば、ブートアップメッセージが表示されます。

図 5-22 Linux カーネルブートアップメッセージ

```
NetWinder Floating Point Emulator V0.95 (c) 1998-1999 Rebel.com
Looking up port of RPC 100003/2 on 192.168.1.241
Looking up port of RPC 100005/1 on 192.168.1.241
VFS: Mounted root (nfs filesystem).
Freeing init memory: 212K
INIT: version 2.78 booting
Activating swap...
Checking all file systems...
Parallelizing fsck version 1.22 (22-Jun-2001)
mkdir: cannot create directory `/dev/pts': File exists
Mounting local filesystems...
nothing was mounted
Cleaning: /etc/network/ifstate.
Setting up IP spoofing protection: rp_filter.
Disable TCP/IP Explicit Congestion Notification: done.
Configuring network interfaces: done.
Starting portmap daemon: portmap.
Cleaning: /tmp /var/lock /var/run.
INIT: Entering runlevel: 3
Starting system log daemon: syslogd klogd.
Starting internet superserver: inetd.

MontaVista Linux 3.0, Professional Edition

kzp-arm login:
```

5.3.8 アプリケーションのロード

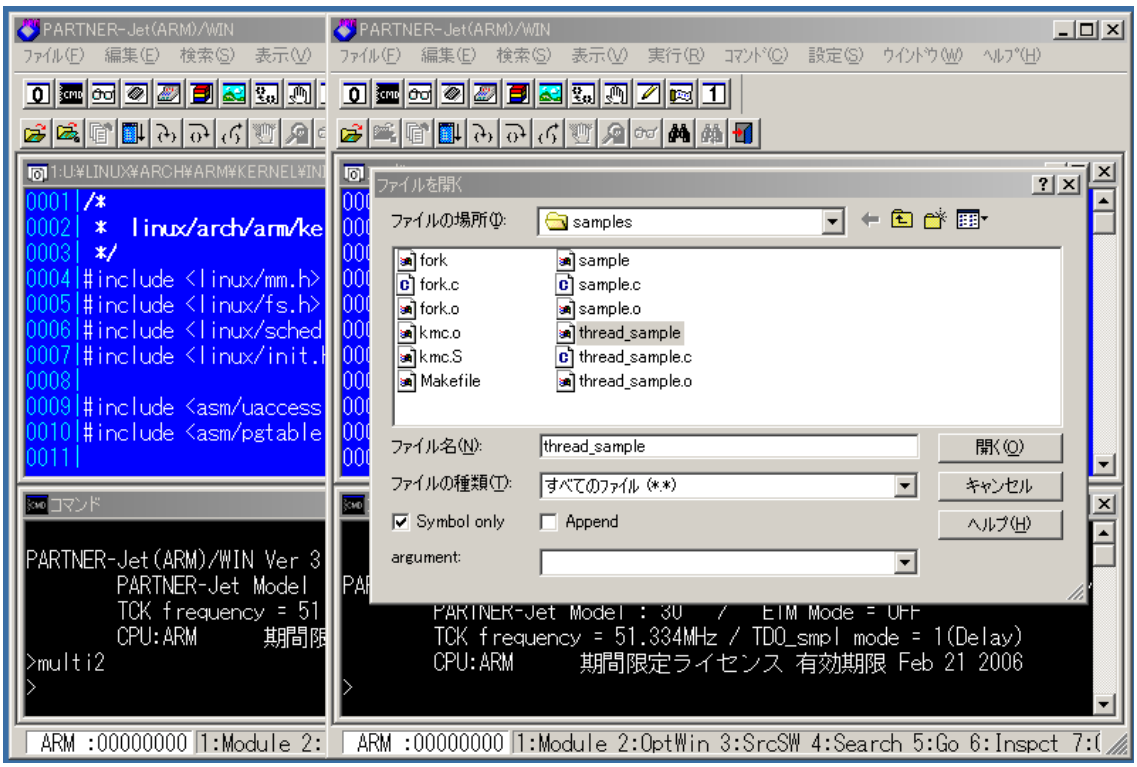
作成したアプリケーションのデバッグ情報を PARTNER にロードします。

● ADD モードの場合

アプリケーション用 PARTNER ウインドウでデバッグ対象アプリケーションのデバッグ情報のみをロードします。
ロード時には必ず [Symbol only] のチェックを行ってください。

```
PT>ls thread_sample ↓
```

図 5-23 アプリケーションのデバッグ情報ロード



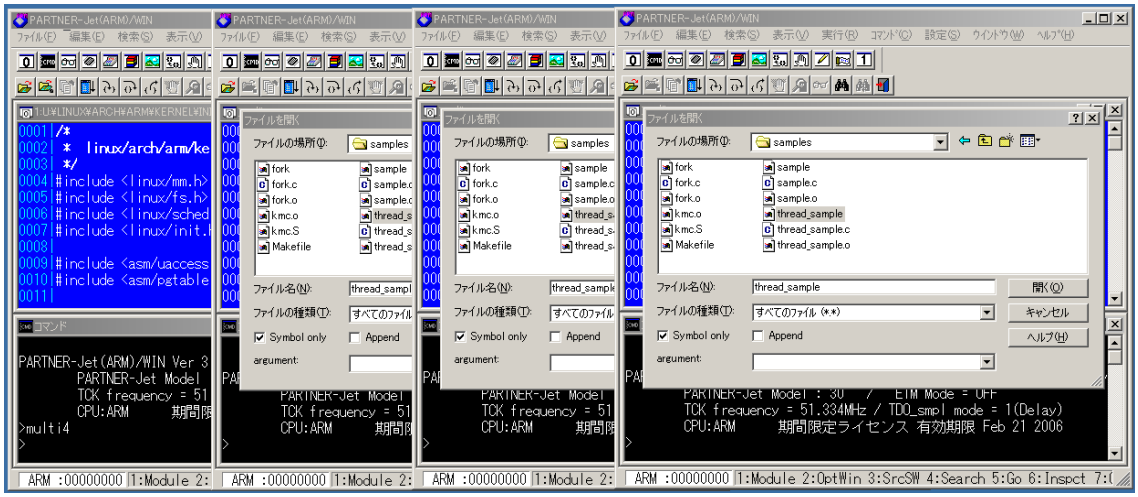
一度ロードされたファイルは、PARTNER がロードファイル名と場所をダイアログとコマンドヒストリに記憶するため、以降の操作を簡略化することができます。

● NON_ADD モードの場合

アプリケーション用 PARTNER ウインドウと子プロセス / スレッド用 PARTNER ウインドウにそれぞれ、デバッグ対象ファイルのデバッグ情報のみをロードします。
ロード時には必ず [Symbol only] のチェックを行ってください。

PT>ls thread_sample ↓

図 5-24 アプリケーションのデバッグ情報ロード



一度ロードされたファイルは、PARTNER がロードファイル名と場所をダイアログとコマンドヒストリに記憶するため、以降の操作を簡略化することができます。

5.3.9 アプリケーションの実行

ターゲットシステムでデバッグ対象のアプリケーションを実行します。

```
TGT>./thread_sample ↓
```

図 5-25 アプリケーションの実行

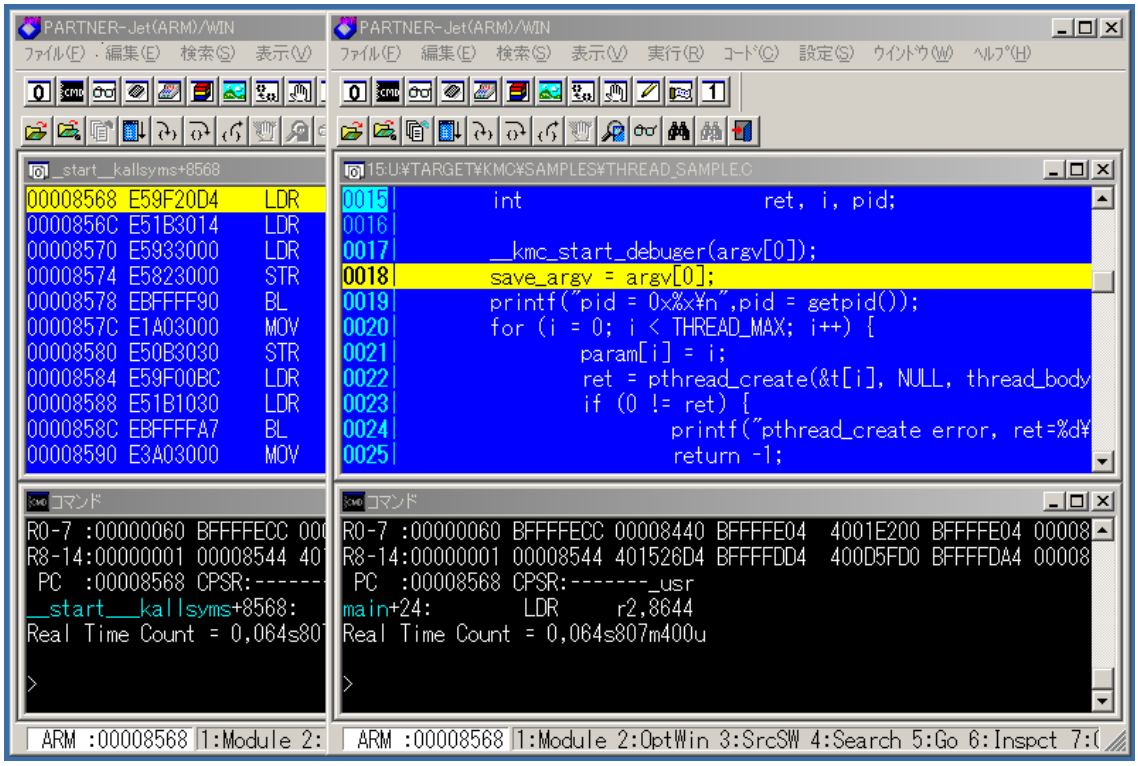
```
mkdir: cannot create directory `/dev/pts': File exists
Mounting local filesystems...
nothing was mounted
Cleaning: /etc/network/ifstate.
Setting up IP spoofing protection: rp_filter.
Disable TCP/IP Explicit Congestion Notification: done.
Configuring network interfaces: done.
Starting portmap daemon: portmap.
Cleaning: /tmp /var/lock /var/run.
INIT: Entering runlevel: 3
Starting system log daemon: syslogd klogd.
Starting internet superserver: inetd.

MontaVista Linux 3.0, Professional Edition

kzp-arm login: root
Last login: Thu Jan  1 00:01:25 1970 on console
Linux kzp-arm 2.4.18_mvl30-integrator #407 2005年 4月 11日 月曜日 16:15:57 JST a
rmv4l unknown

Welcome to MontaVista Linux 3.0, Professional Edition

root@kzp-arm:~# cd /KMC/samples/
root@kzp-arm:/KMC/samples# ./thread_sample
```

5.3.10 PARTNER のブレーク

● ADD モードの場合

ターゲットシステムでデバッグ対象のアプリケーションが実行されると、PARTNER は先ず、main() 関数後に挿入したデバッグスタブ関数 (__kmc_start_debuger ()) の後ろでブレークします。このとき、アプリケーションの PID や配置情報を自動的に獲得し、以後アプリケーションエリアのメモリ参照やブレークポイントの設定が可能になります。

図 5-26 アプリケーションのアタッチ (親プロセス)

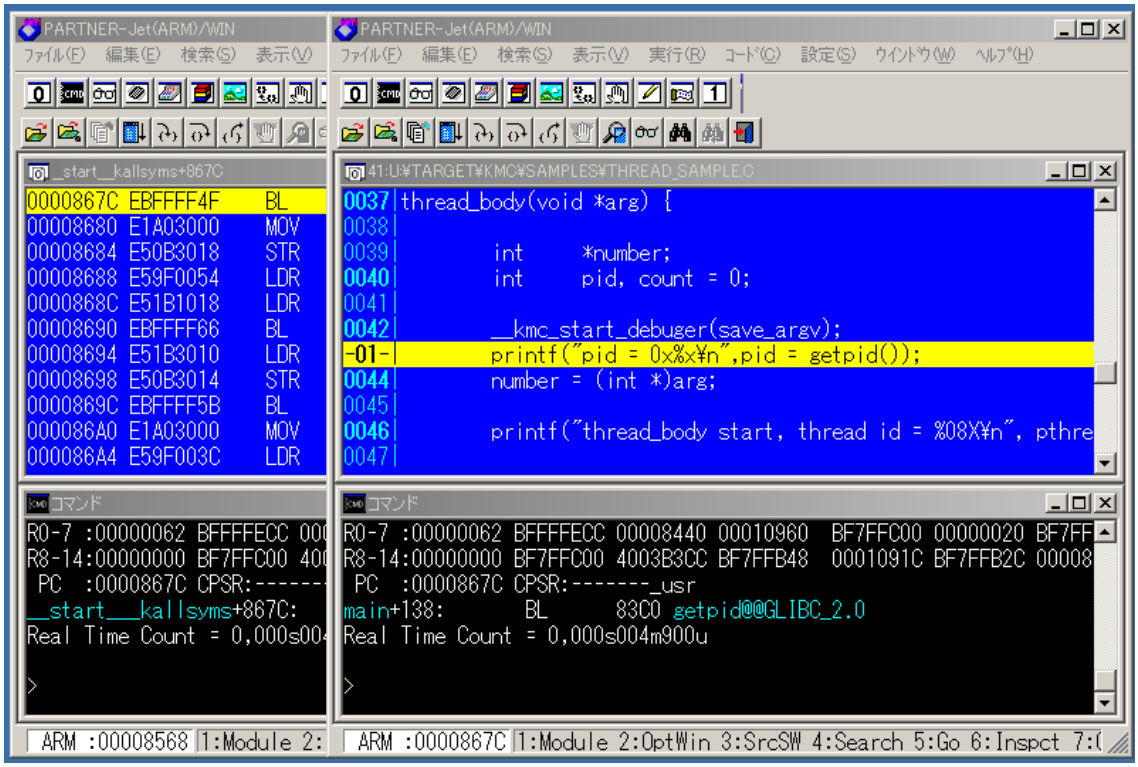


PSID コマンド (25 ページ参照) でデバッガにアプリケーションがアタッチされているか確認します。

```
PT>psid ↓
PSID SET 104(0x68)  CURRENT 104(0x68) [ADD MODE]
APPLI. AREA : 00400000-00400FFF
APPLI. AREA : 00410000-00410FFF
APPLI. AREA : 00410000-00410FFF
APPLI. AREA : 7BFFF000-7BFFFFFF
```

カーネルモード (ADD モード) でデバッグしている場合は、main() 関数後に挿入したデバッグスタブ関数 (__kmc_start_debuger ()) ではブレークしますが、スレッドボディ関数の先頭に挿入したデバッグスタブ関数ではブレークしません。
ここで thread_body () 関数内に挿入したデバッグスタブ関数の後 (スレッドの先頭のデバッグスタブの後ろ) にブレークポイントを設定し、アプリケーションを実行すると、新しく生成されたスレッドが自動的に PARTNER にアタッチされ設定したブレークポイントでブレークします。

図 5-27 アプリケーションのアタッチ (スレッド)



PSID コマンド (25 ページ参照) を実行するとアプリケーション用 PARTNER ウインドウで新しく生成されたスレッドがアタッチされていることが確認できます。

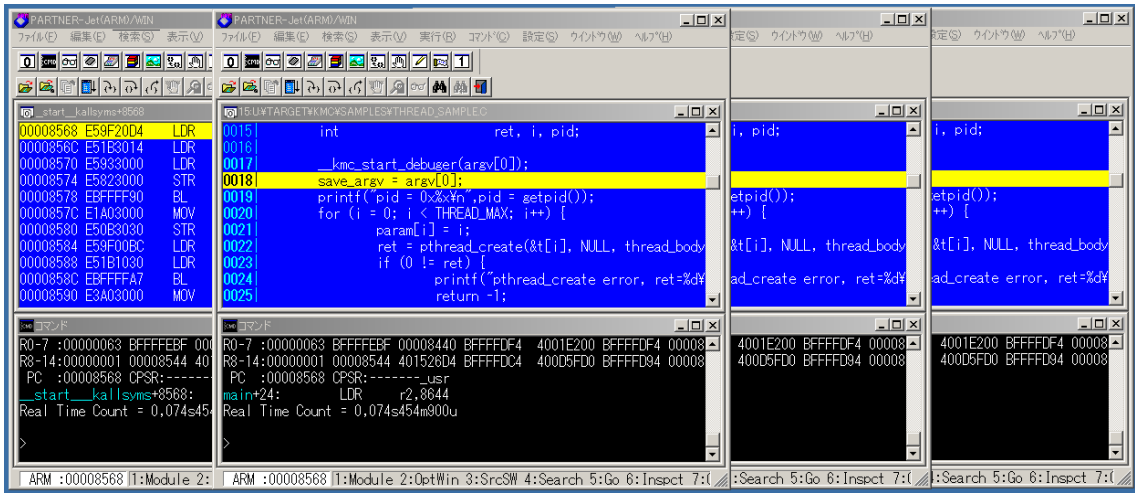
```
PT>psid ↓
PSID SET 106(0x6A)  CURRENT 106(0x6A) [ADD MODE 104]
APPLI. AREA : 00400000-00400FFF
APPLI. AREA : 00410000-00410FFF
APPLI. AREA : 00410000-00412FFF
APPLI. AREA : 7BFFF000-7BFFFFFF
```

ブレークポイントを設定せずに実行すると、PARTNER にはデバッグスタブ関数 (_kmc_start_debuger()) を挿入したスレッドはアタッチされますが、自動ブレークは行いません。最初のアプリケーションがアタッチされた時点で、ブレークポイントを設定してください。

● NON_ADD モードの場合

ターゲットシステムでデバッグ対象のアプリケーションが実行されると、アプリケーション用 PARTNER ウインドウがデバッグスタブ関数の後ろでブレークします。このとき、アプリケーションの PID や配置情報を自動的に獲得し、以後アプリケーションエリアのメモリ参照やブレークポイントの設定が可能になります。

図 5-28 アプリケーションのアタッチ

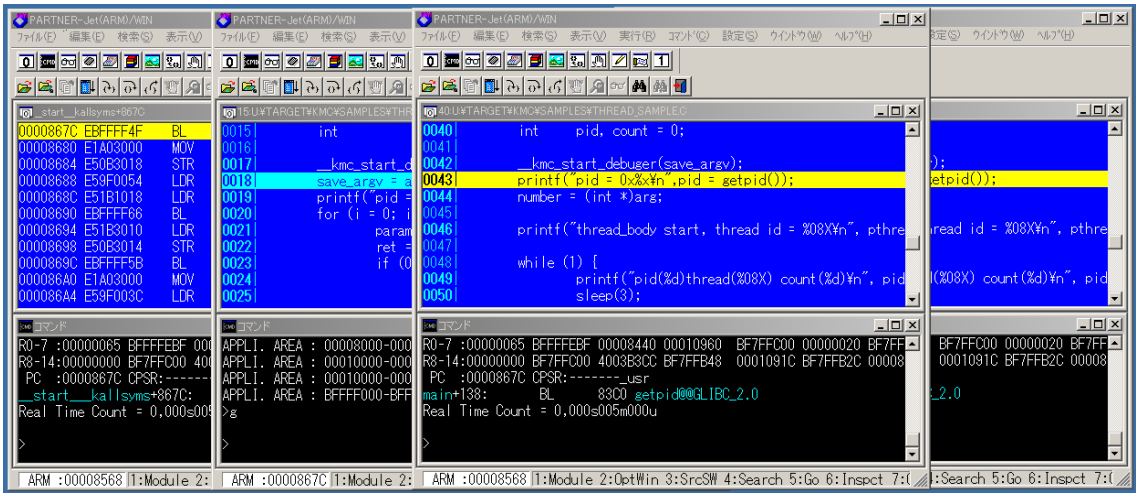


PSID コマンド (25 ページ参照) を実行するとアプリケーション用 PARTNER ウインドウでアプリケーションがアタッチされていることが確認できます。

```
PT>psid ↓
PSID SET 104(0x68)  CURRENT 104(0x68)
APPLI. AREA : 00400000-00400FFF
APPLI. AREA : 00410000-00410FFF
APPLI. AREA : 00410000-00410FFF
APPLI. AREA : 7BFFF000-7BFFFFFF
```

次に、アプリケーション用 PARTNER ウインドウでアプリケーションを再開し、create_thread() 関数が実行されスレッドが生成されると、スレッド用 PARTNER ウインドウがデバッグスタブ関数の後ろでブレークします。このとき、スレッドの PID や配置情報を自動的に獲得し、以後スレッドエリアのメモリ参照やブレークポイントの設定が可能になります。

図 5-29 スレッド 1 のアタッチ

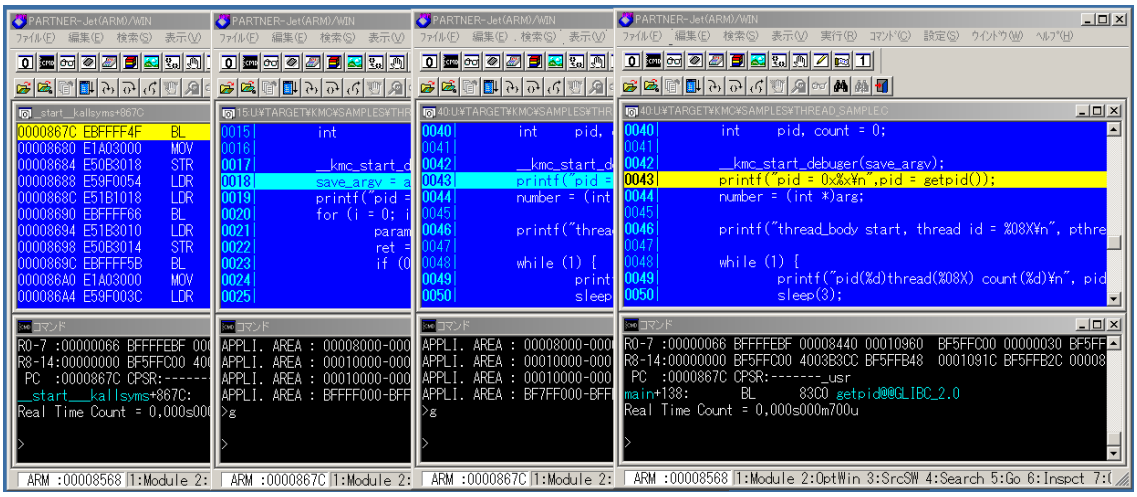


PSID コマンド (25 ページ参照) を実行するとスレッド 1 用 PARTNER ウインドウでスレッドがアタッチされていることが確認できます。

```
PT>psid ↓
PSID SET 106(0x6A)  CURRENT 106(0x6A)
APPLI. AREA : 00400000-00400FFF
APPLI. AREA : 00410000-00410FFF
APPLI. AREA : 00410000-00410FFF
APPLI. AREA : 7BFFF000-7BFFFFFF
```

再度、スレッド 1 用 PARTNER ウインドウでアプリケーションを再開し、create_thread() 関数が実行されスレッドが生成されると、もう一つのスレッド用 PARTNER ウインドウがデバッグスタブ関数の後ろでブレークします。このとき、スレッドの PID や配置情報を自動的に獲得し、以後スレッドエリアのメモリ参照やブレークポイントの設定が可能になります。

図 5-30 スレッド 2 のアタッチ



PSID コマンド(25 ページ参照)を実行するとスレッド 2 用 PARTNER ウインドウでスレッドがアタッチされていることが確認できます。

```
PT>psid ↓
PSID SET 107(0x6B)  CURRENT 107(0x6B)
APPLI. AREA : 00400000-00400FFF
APPLI. AREA : 00410000-00410FFF
APPLI. AREA : 00410000-00410FFF
APPLI. AREA : 7BFFF000-7BFFFFFF
```

PARTNERが正しくブレークしない場合､カーネルのコンフィグレーションが間違っている可能性があります『 5.3.1 Linux カーネルのコンフィグレーション(86 頁)』の設定と『 5.3.4 カーネルモードでの PARTNER の起動(89 頁)』で指定した OS デバッグモードがカーネルモードになっているか確認してください。
また、アプリケーションソースにデバッグスタブ関数(_kmc_start_debuger()) が挿入されているか、サポートファイル(kmc-support.c)がリンクされているか確認してください。

5.4 アプリケーションモードでのアプリケーションデバッグ

PARTNER でのアプリケーションのデバッグには、アプリケーションモードとカーネルモードの 2 つのデバッグモードが存在します(『Linux と PARTNER のデバッグモード (12 頁)』参照)。ここでは、アプリケーションモードのデバッグ方法を説明します。
カーネルモードでのアプリケーションのデバッグ方法は、『5.2 カーネルモードでのアプリケーションデバッグの手順 (73 頁)』を参照してください。

『 3.2 カーネルソース修正(35 頁)』を行っていない場合は、これから説明するアプリケーションのデバッグが出来ません。修正を行っていないカーネルを使用する場合は、『 付録 B 手動アプリケーションデバッグ(162 頁)』を参照してください。

この節では、サンプル (sample) を使用した場合を例として、アプリケーションモードでアプリケーション (プロセス) のデバッグ方法を次の流れで説明します。

(1) Linux カーネルのコンフィグレーション (105 頁)
(2) Linux カーネルのビルド (105 頁)
(3) アプリケーションの作成 (106 頁)
(4) アプリケーションモードでの PARTNER の起動 (107 頁)
(5) Linux カーネルのロード (110 頁)
(6) マルチウインドウデバッグ (111 頁)
(7) Linux カーネルの実行 (112 頁)
(8) アプリケーションデバッグ情報のロード (113 頁)
(9) アプリケーションの実行 (114 頁)
(10) PARTNER のブレーク (115 頁)

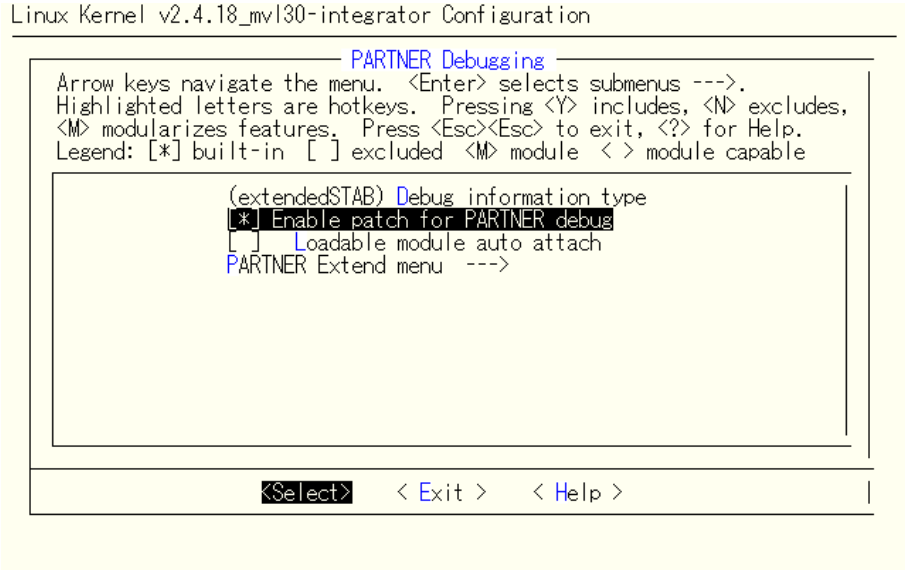
5.4.1 Linux カーネルのコンフィグレーション

アプリケーションモードでデバッグする場合は、Linux カーネルのコンフィグレーションで [Enable patch for PARTNER debug] を有効にします。
また、カーネルのデバッグ情報のフォーマットを指定します。
推奨は extendedSTAB(-gstabs+) です。PARTNER で実際カーネルを読み込んでデバッグ情報がおかしなときは、DWARF2 などに変えて試みてください。

```
LINUX86>make menuconfig ↓
```

[Kernel hacking]->[PARTNER Debugging]->[Debug infomation type]
[Kernel hacking]->[PARTNER Debugging]->[Enable patch for PARTNER debug]

図 5-31 Linux カーネルのコンフィグレーション



5.4.2 Linux カーネルのビルド

上記のコンフィグレーションで、Linux カーネル(vmlinux)を作成します。

```
LINUX86>make ↓
```

5.4.3 アプリケーションの作成

デバッグするアプリケーションを作成する場合、以下の手順でアプリケーションを作成します。

(1) アプリケーションソースの修正

アプリケーションソースの main() 関数の先頭にデバッグスタブの呼び出しを挿入します。

```
__kmc_start_debuger(char *program_name);
```

デバッグスタブ関数の引数 char *program_name には、アプリケーションのファイル名が入るようにしてください。
ただし、シンボリックリンクでファイル名が変えられている場合などは、実際のファイル名を文字列で埋め込んでください。
PARTNER は、この引数の文字列とデバッガで読み込んだデバッグ情報内のファイル名を比較してデバッグ対象の場合はデバッガにアタッチします。

【例】

```
int main(int argc,char *argv[])
{
+    __kmc_start_debug(argv[0]);
                    :
                    :
                    :
                    :
```

(2) アプリケーションのリンク

アプリケーションのリンク時にサポートファイル(kmc-support.c)をリンクします。

kmc-support.c 内で "Select Target CPU type" のコンパイルエラーが発生した場合は、kmc-support.c 内の CPUTYPE シンボル宣言をターゲット CPU のみ有効にして再コンパイルしてください。

なおアプリケーションの作成時に、カーネルコンフィグレーションで指定したデバッグ情報を付加するようにしてください。

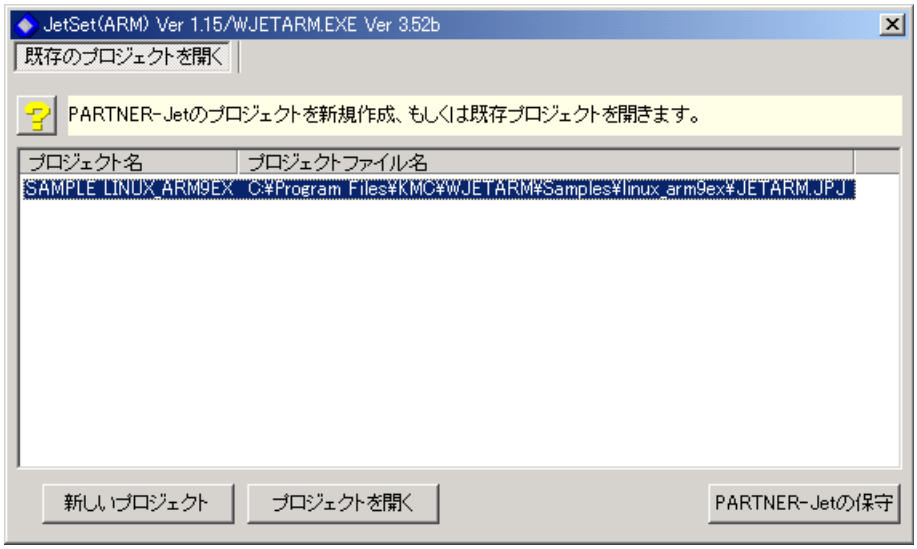
5.4.4 アプリケーションモードでの PARTNER の起動

アプリケーションモードで PARTNER を起動するには、次に示す手順で行います。

(1) Jetset でプロジェクトの新規作成もしくは既存のプロジェクトの選択

PARTNER の環境設定プログラム (JETSET(SH)) を起動し、プロジェクトを新規作成もしくは、オープンしてます。

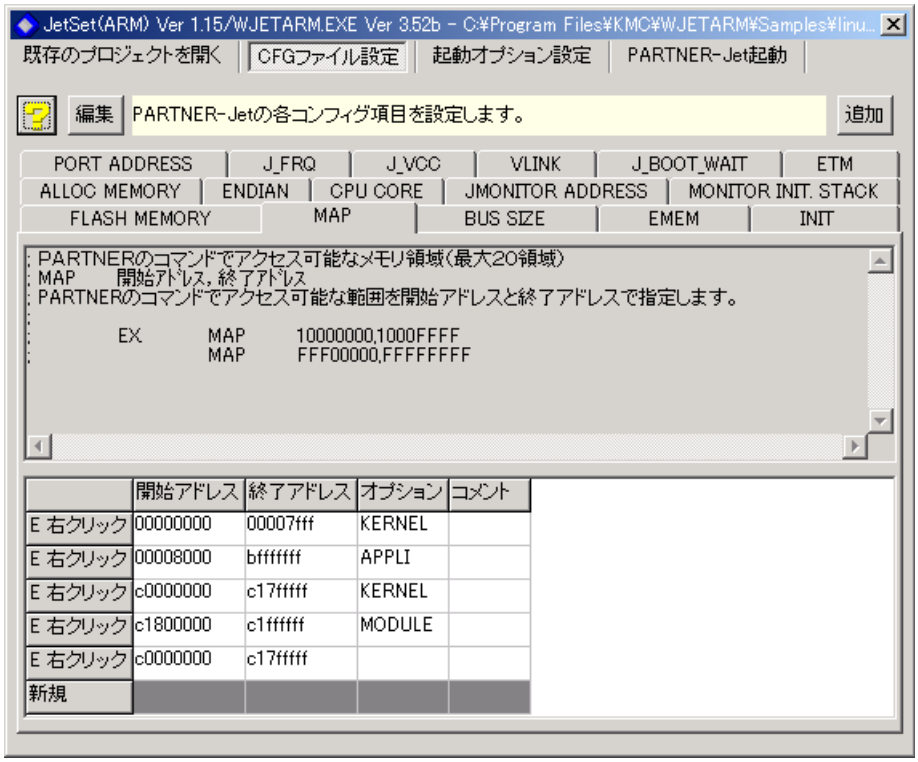
図 5-32 PARTNER プロジェクトのオープン



(2) CFG ファイル設定

MAP フィールドに Linux 用 MAP 情報を指定します。設定するアドレスや属性などについては『CFG ファイルの拡張 (15 頁)』を参照してください。

図 5-33 MAP フィールドの変更



(3) 起動オプション設定

[拡張 >>] ボタンを押し、Linux デバッグ用拡張オプションを指定できるようにします。
カーネルデバッグ時に必ず設定する必要のあるオプションは、以下のとおりです。

● **デバッグ情報バッファサイズ (-B オプション)**

サイズには 100000 程度を指定してください。
もし、カーネルファイルをロードしたときにエラーメッセージ『 デバッグ情報領域がいっぱいです (起動時の **-B オプション**参照)』が表示された場合は、バッファサイズを拡大してください。

● **デバッグ情報タイプ (-XGX オプション)**

『GNU C (Linux etc.)』を選択します。

デバッグ情報パス変換 (-XGX オプション)

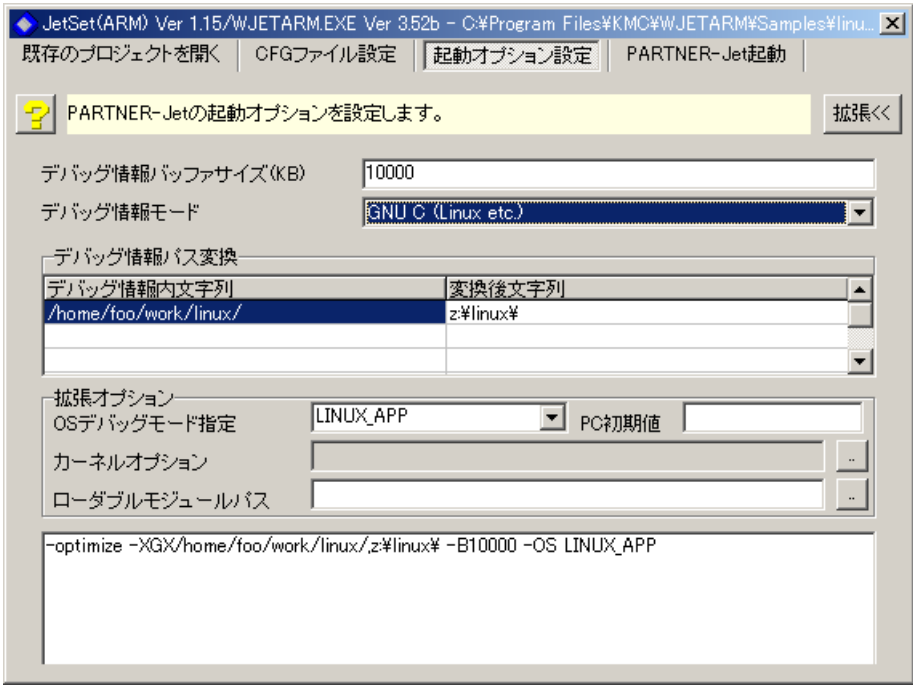
カーネル (vmlinux) をビルドした PATH と Samba でマウントしている PATH が違う場合に指定します。
たとえば、/home/foo/work/linux でカーネルをビルドして、ホスト PC で /home/foo/work を Z: ドライブにマウントした場合は、-XGX/home/foo/work/linux/,z:¥linux¥ と指定します。

● **OS デバッグモード指定 (-OS オプション)**

『LINUX_APP』を選択します。

各オプションについての詳細は、『2.2.2 起動オプション (18 頁)』を参照してください。

図 5-34 起動オプションの設定



(4) PARTNER の起動

JETSET(SH) 上の [起動] ボタンをクリックすることにより、PARTNER が起動します。

図 5-35 PARTNER の起動

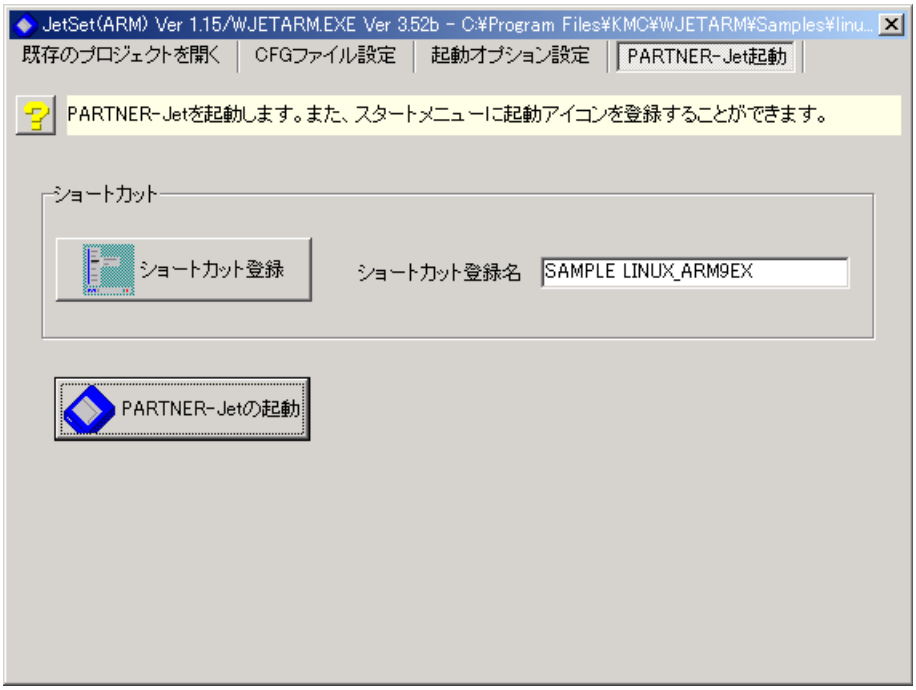
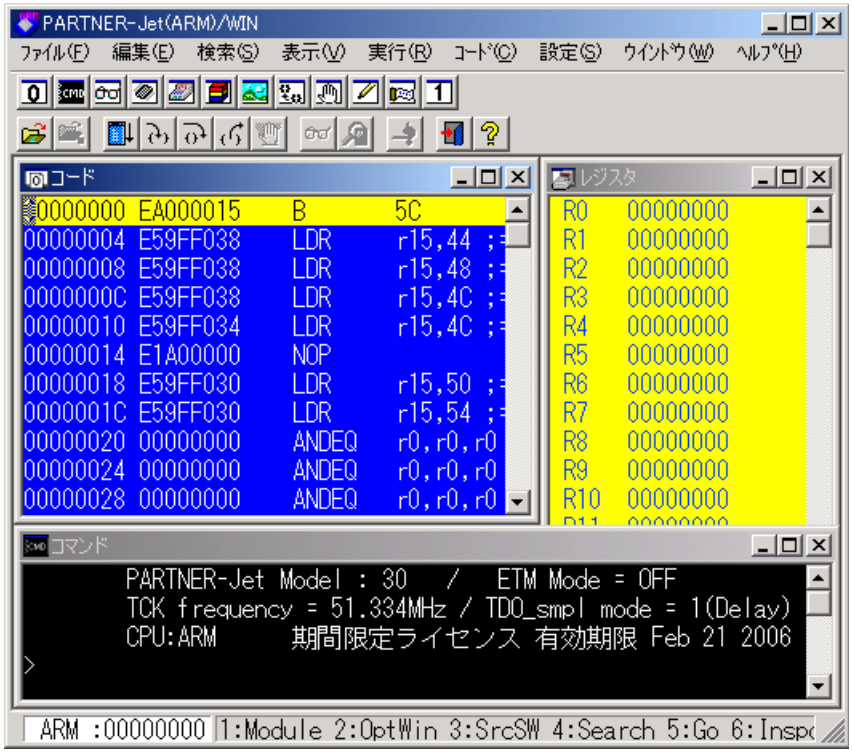


図 5-36 PARTNER の起動画面

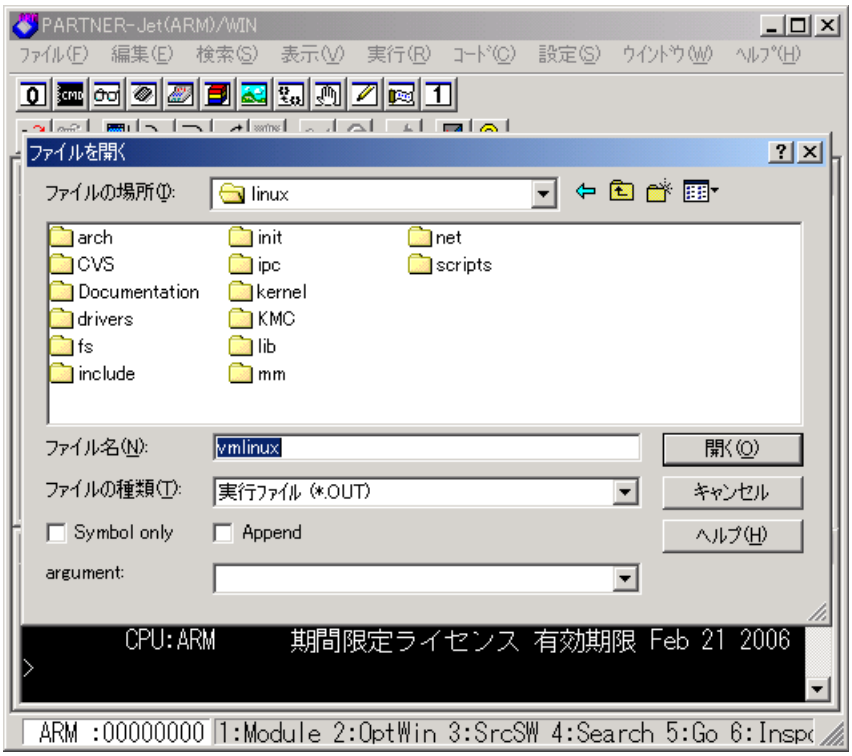


5.4.5 Linux カーネルのロード

『5.4.2 Linux カーネルのビルド (105 頁)』で作成したカーネル(vmlinux)をターゲットメモリにロードします。

PT>l vmlinux ↓

図 5-37 カーネルのロード



vmlinux から作成された HEX ファイルのバイナリファイルをロードする場合には、以下の手順でカーネルを読み込みます。

PARTNER のコマンドウィンドウにおいて、次のコマンドを入力します。

【例】rd コマンドで HEX ファイルを読み込み、ls コマンドでデバッグ情報を読み込みます。

PT>rd z:¥linux¥vmlinux.hex ↓
PT>ls z:¥linux¥vmlinux ↓

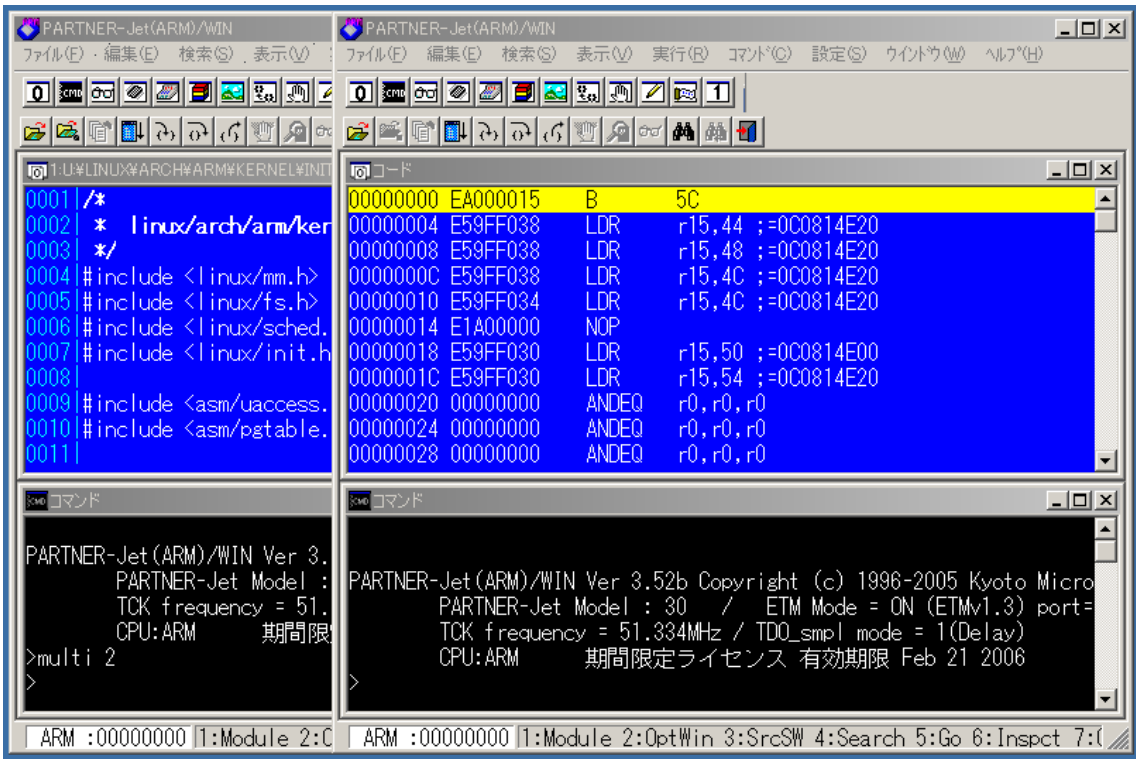
コンパイル時にデバッグ情報出力を指定している場合には、カーネルロード時にデバッグ情報も同時にロードされます。
また、一度ロードされたファイルは、PARTNER がロードファイル名と場所をダイアログとコマンドヒストリに記憶するため、以降の操作を簡略化することができます。

5.4.6 マルチウインドウデバッグ

Linux カーネルとアプリケーションを別の PARTNER ウインドウでデバッグする場合は、**MULTI コマンド** (22 ページ参照) で複数の PARTNER ウインドウを起動します。サンプルプログラム (sample) をデバッグする場合は、カーネル用 PARTNER ウインドウとアプリケーション用 PARTNER ウインドウの 2 つの PARTNER ウインドウを起動します。

```
PT>multi 2 ↓
```

図 5-38 複数 PARTNER ウインドウの起動



MULTI コマンド (22 ページ参照) や -MULTI オプション (20 ページ参照) で起動した PARTNER ウインドウのウインドウ情報 (ウインドウ配置等) やコマンドヒストリは、新規に生成されたプロジェクトファイル (jetsh_?.jpj) に保存され、次回起動時に利用されます。

5.4.7 Linux カーネルの実行

カーネルのロードが正しくできた状態で、**G コマンド**、または**[実行] ボタン**でロードした vmlinux を実行させます。

```
PT>g ↓
```

Windows PC とターゲットボードが正しく接続されていて、ターミナルソフトが正常に起動していれば、ブートアップメッセージが表示されます。

図 5-39 カーネルブートアップメッセージ

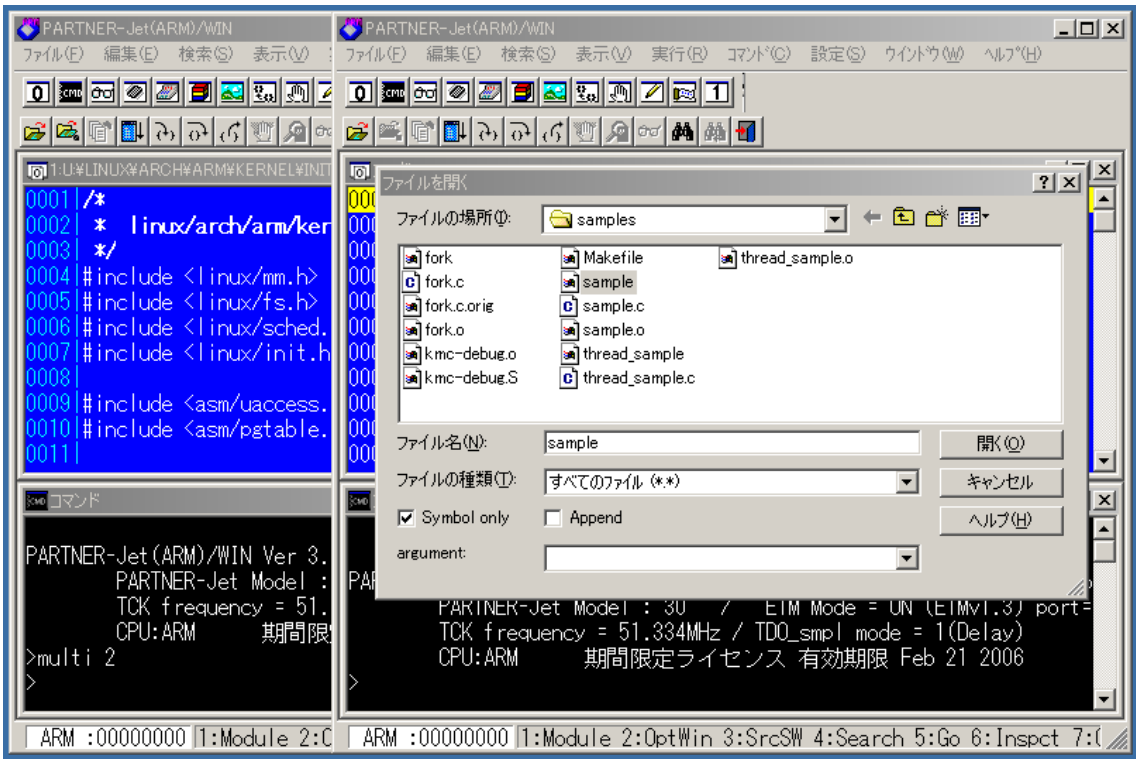
```
NetWinder Floating Point Emulator V0.95 (c) 1998-1999 Rebel.com
Looking up port of RPC 100003/2 on 192.168.1.241
Looking up port of RPC 100005/1 on 192.168.1.241
VFS: Mounted root (nfs filesystem).
Freeing init memory: 212K
INIT: version 2.78 booting
Activating swap...
Checking all file systems...
Parallelizing fsck version 1.22 (22-Jun-2001)
mkdir: cannot create directory `/dev/pts': File exists
Mounting local filesystems...
nothing was mounted
Cleaning: /etc/network/ifstate.
Setting up IP spoofing protection: rp_filter.
Disable TCP/IP Explicit Congestion Notification: done.
Configuring network interfaces: done.
Starting portmap daemon: portmap.
Cleaning: /tmp /var/lock /var/run.
INIT: Entering runlevel: 3
Starting system log daemon: syslogd klogd.
Starting internet superserver: inetd.

MontaVista Linux 3.0, Professional Edition

kzp-arm login:
```

5.4.8 アプリケーションデバッグ情報のロード

作成したアプリケーションのデバッグ情報を PARTNER にロードします。
ロード時には必ず [Symbol only] のチェックを行ってください。また、複数のデバッグ情報を一つの PARTNER ウインドウにロードする場合は、[Append] のチェックを行ってください。

```
PT>ls sample ↓
PT>lsa sample ↓
```

図 5-40 アプリケーションデバッグ情報のロード



一度ロードされたファイルは、PARTNER がロードファイル名と場所をダイアログとコマンドヒストリに記憶するため、以降の操作を簡略化することができます。

5.4.9 アプリケーションの実行

ターゲットシステムでデバッグ対象のアプリケーションを実行します。

```
TGT>./sample ↓
```

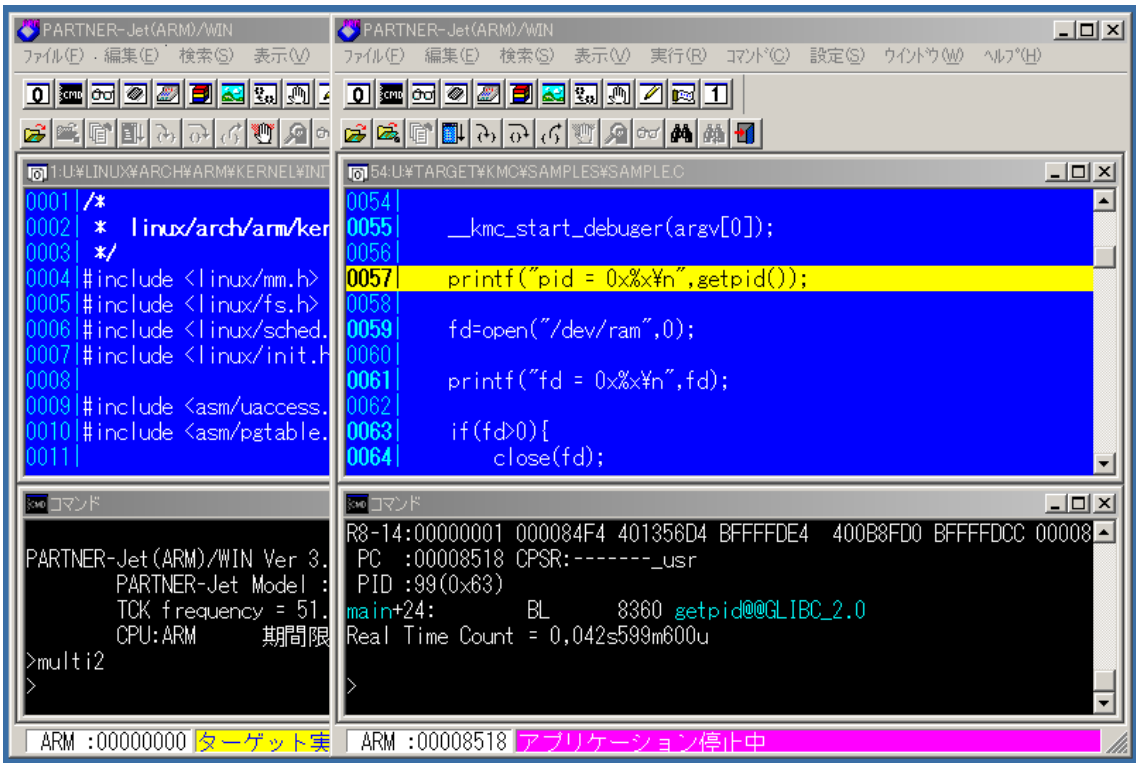
図 5-41 アプリケーションの実行

```
Parallelizing fsck version 1.22 (22-Jun-2001)
mkdir: cannot create directory `/dev/pts': File exists
Mounting local filesystems...
nothing was mounted
Cleaning: /etc/network/ifstate.
Setting up IP spoofing protection: rp_filter.
Disable TCP/IP Explicit Congestion Notification: done.
Configuring network interfaces: done.
Starting portmap daemon: portmap.
Cleaning: /tmp /var/lock /var/run.
INIT: Entering runlevel: 3
Starting system log daemon: syslogd klogd.
Starting internet superserver: inetd.

MontaVista Linux 3.0, Professional Edition

kzp-arm login: root
Last login: Thu Jan  1 00:00:19 1970 on console
Linux kzp-arm 2.4.18_mvl30-integrator #384 2005年 3月 29日 火曜日 20:06:36 JST a
rmv4l unknown

Welcome to MontaVista Linux 3.0, Professional Edition

root@kzp-arm:~# ./sample
```

5.4.10 PARTNER のブレーク

ターゲットシステムでデバッグ対象のアプリケーションが実行されると、PARTNER はデバッグスタブ関数（__kmc_start_debuger()）の後ろでブレークします。このとき、アプリケーションの PID や配置情報を自動的に獲得し、以後アプリケーションエリアのメモリ参照やブレークポイントの設定が可能になります。

図 5-42 アプリケーションのアタッチ直後



PSID コマンド（25 ページ参照）でデバッガにアプリケーションがアタッチされているか確認します。

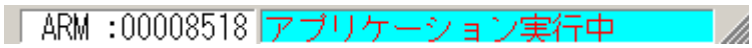
```
PSID SET 106(0x6A)  CURRENT 0(0x0)
APPLI. AREA : 00400000-00401FFF
APPLI. AREA : 00411000-00451FFF
APPLI. AREA : 00453000-00453FFF
APPLI. AREA : 7BFFF000-7BFFFFFF
```

PARTNERが正しくブレークしない場合､カーネルのコンフィグレーションが間違っている可能性があります『 5.4.1 Linux カーネルのコンフィグレーション（ 105 頁）』の設定と『 5.4.4 アプリケーションモード での PARTNER の起動（ 107 頁）』で指定した OS デバッグモード がアプリケーションモード になっているか確認してください。
また、アプリケーションソースにデバッグスタブ関数（_kmc_start_debuger()）が挿入されているか、サポートファイル（kmc-support.c）がリンクされているか確認してください。

5.4.11 アプリケーションモードでの PARTNER ステータスバー表示

アプリケーションモードで PARTNER を動作する場合、ターゲットの状態により、PARTNER のステータスバーは次のように変化します。

表 5-1 アプリケーションモードでのステータスバー表示

ターゲットの状態	ステータスバー
アプリケーション実行中	
アプリケーション停止	
Linux カーネル (CPU) 停止	

5.5 アプリケーションモードでのマルチプロセス / マルチスレッドアプリケーションのデバッグ

fork() や pthread を使用したアプリケーションのデバッグ手順は前述の『5.4 アプリケーションモードでのアプリケーションデバッグ（104 頁）』の手順とほぼ同じで、カーネルのコンフィグレーション、アプリケーションの作成と起動オプションの指定、マルチウインドウデバッグが異なるだけです。

マルチプロセス / マルチスレッドアプリケーションのデバッグには、各プロセス / スレッドを同じ PARTNER ウインドウでデバッグする ADD モードと、それぞれ別の PARTNER ウインドウでデバッグする NON_ADD モードがあります。
ADD モードと NON_ADD モードの切り替えは、**PSID コマンド**（25 ページ参照）または、**-OS オプション**（18 ページ参照）で指定します。
また、PARTNER でのアプリケーションのデバッグには、アプリケーションモードとカーネルモードの 2 つのデバッグモードが存在します（『2.1 Linux と PARTNER のデバッグモード（12 頁）』参照）。ここでは、アプリケーションモードのデバッグ方法を説明します。

カーネルモードでのアプリケーションのデバッグ方法は、『5.3 カーネルモードでのマルチプロセス / マルチスレッドアプリケーションデバッグ手順（85 頁）』を参照してください。

『 3.2 カーネルソース修正（ 35 頁）』を行っていない場合は、これから説明するアプリケーションのデバッグが出来ません。修正を行っていないカーネルを使用する場合は、『 付録 C 手動マルチプロセス / マルチスレッド対応のデバッグ方法（ 169 頁）』を参照してください。

この節では、サンプル（fork/thread_sample）を使用した場合を例として、アプリケーションモードでのアプリケーション（プロセス）のデバッグ方法を次の流れで説明します。

(1) Linux カーネルのコンフィグレーション（118 頁）
(2) Linux カーネルのビルド（118 頁）
(3) アプリケーションの作成（119 頁）
(4) アプリケーションモードでの PARTNER の起動（121 頁）
(5) Linux カーネルのロード（124 頁）
(6) マルチウインドウデバッグ（125 頁）
(7) Linux カーネルの実行（127 頁）
(8) アプリケーションのロード（128 頁）
(9) アプリケーションの実行（130 頁）
(10) PARTNER のブレーク（131 頁）

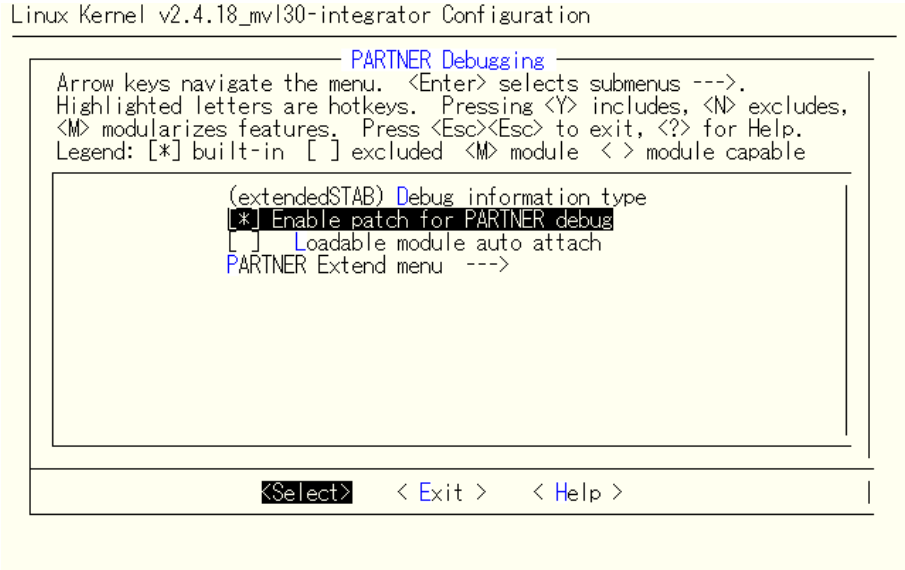
5.5.1 Linux カーネルのコンフィグレーション

アプリケーションモードでマルチプロセス / マルチスレッドアプリケーションをデバッグする場合は、Linux カーネルのコンフィグレーションで [Enable patch for PARTNER debug] を有効にします。
また、カーネルのデバッグ情報のフォーマットを指定します。
推奨は extendedSTAB(-gstabs+) です。PARTNER で実際カーネルを読み込んでデバッグ情報がおかしなときは、DWARF2 などに変えて試みてください。

```
LINUX86>make menuconfig ↓
```

[Kernel hacking]->[PARTNER Debugging]->[Debug infomation type]
[Kernel hacking]->[PARTNER Debugging]->[Enable patch for PARTNER debug]

図 5-43 Linux カーネルのコンフィグレーション



5.5.2 Linux カーネルのビルド

上記のコンフィグレーションで、Linux カーネル (vmlinux) を作成します。

```
LINUX86>make ↓
```

5.5.3 アプリケーションの作成

デバッグするアプリケーションを作成する場合、以下の手順でアプリケーションを作成します。

(1) アプリケーションソースの修正

【マルチスレッドアプリケーション(thread_sample)の場合】

アプリケーションソースの main() 関数の先頭とスレッドボディ関数の先頭でデバッグスタブの呼び出しを挿入します。

```
__kmc_start_debuger(char *program_name);
```

main() 関数直後に挿入するデバッグスタブ関数の引数 char *program_name には、アプリケーションのファイル名が入るようにしてください。
ただし、シンボリックリンクでファイル名が変えられている場合などは、実際のファイル名を文字列で埋め込んでください。
PARTNER は、この引数の文字列とデバッガで読み込んだデバッグ情報内のファイル名を比較してデバッグ対象と一致した場合はデバッガにアタッチします。
スレッドボディ関数の先頭に挿入するデバッグスタブ関数の引数 char *program_name には、0 を指定してください。

【例】

```
int main(int argc,char *argv[])
{
+    __kmc_start_debuger(argv[0]);
                          :
                          :
    pthread_create(&th,NULL,thread_func,NULL);
                          :
                          :
void *thread_func(void *)
{
+    __kmc_start_debuger(0);
                          :
                          :
```

【マルチプロセスアプリケーション(fork)の場合】

アプリケーションソースの main() 関数の先頭と子プロセスの先頭(fork() 関数の子プロセス側の戻り)でデバッグスタブの呼び出しを挿入します。

```
__kmc_start_debuger(char *program_name);
```

main() 関数直後に挿入するデバッグスタブ関数の引数 char *program_name には、アプリケーションのファイル名が入るようにしてください。
ただし、シンボリックリンクでファイル名が変えられている場合などは、実際のファイル名を文字列で埋め込んでください。
PARTNER は、この引数の文字列とデバッガで読み込んだデバッグ情報内のファイル名を比較してデバッグ対象と一致した場合はデバッガにアタッチします。
子プロセスの先頭に挿入するデバッグスタブ関数の引数 char *program_name には、0 を指定してください。

【例】

```
int main(int argc,char *argv[])
{
+    __kmc_start_debuger(argv[0]);
                        :
                        :
    if(fork()==0){
+       __kmc_start_debuger(0);
                        :
                        :
    }
                        :
                        :
```

(2) アプリケーションのリンク

アプリケーションのリンク時にサポートファイル(kmc-support.c)をリンクします。

kmc-support.c 内で "Select Target CPU type" のコンパイルエラーが発生した場合は、kmc-support.c 内の CPUTYPE シンボル宣言をターゲット CPU のみ有効にして再コンパイルしてください。

なおアプリケーションの作成時に、デバッグ情報を付加するようにしてください。

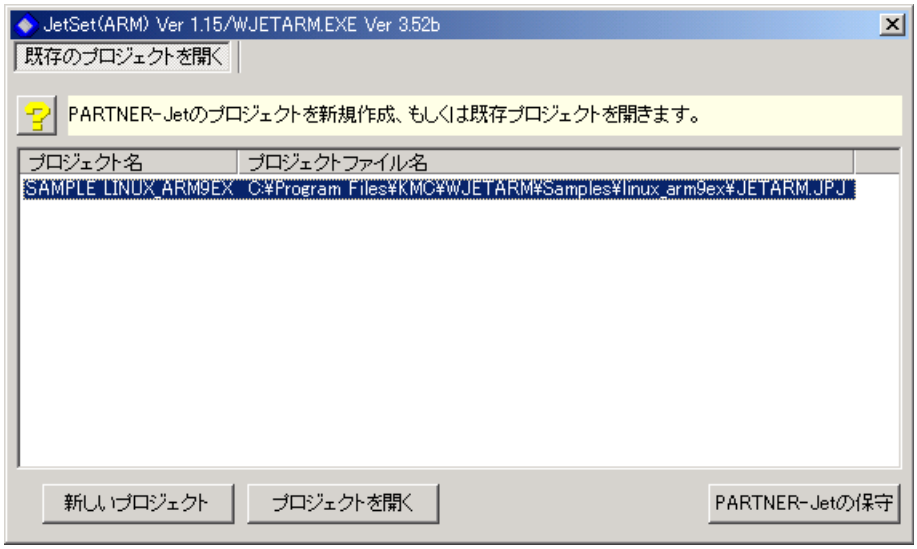
5.5.4 アプリケーションモードでの PARTNER の起動

アプリケーションモードで PARTNER を起動するには、次に示す手順で行います。

(1) Jetset でプロジェクトの新規作成もしくは既存のプロジェクトの選択

PARTNER の環境設定プログラム (JETSET(SH)) を起動し、プロジェクトを新規作成もしくは、オープンしてます。

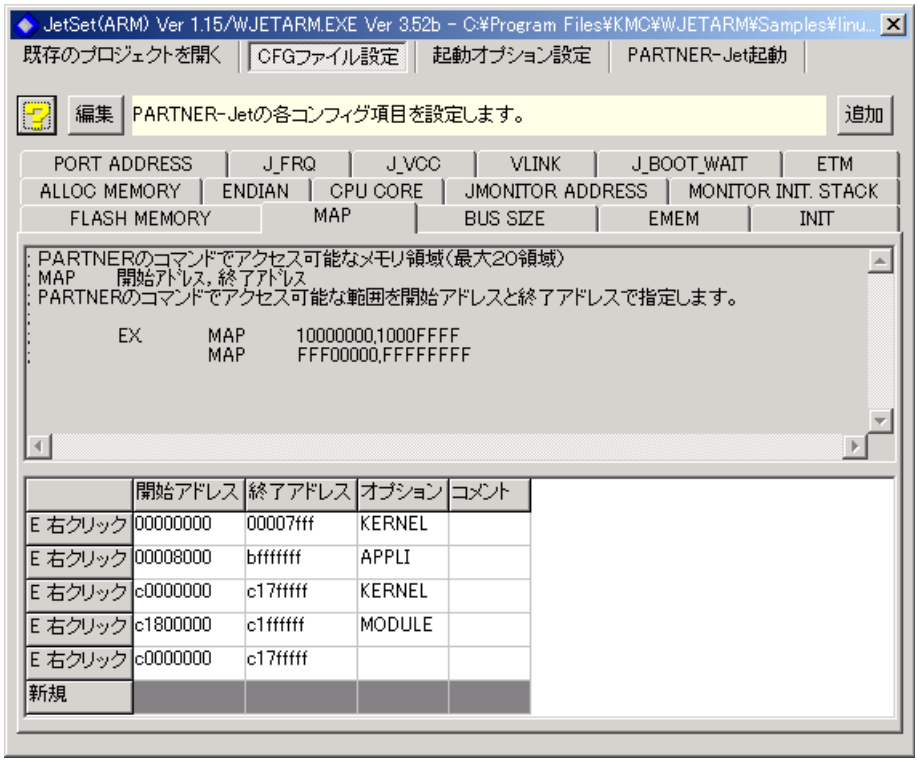
図 5-44 PARTNER プロジェクトのオープン



(2) CFG ファイル設定

MAP フィールドに Linux 用 MAP 情報を指定します。設定するアドレスや属性などについては『CFG ファイルの拡張（15 頁)』を参照してください。

図 5-45 MAP フィールドの設定



(3) 起動オプション設定

[拡張 >>] ボタンを押し、Linux デバッグ用拡張オプションを指定できるようにします。
カーネルデバッグ時に必ず設定する必要のあるオプションは、以下のとおりです。

● **デバッグ情報バッファサイズ (-B オプション)**

サイズには 100000 程度を指定してください。
もし、カーネルファイルをロードしたときにエラーメッセージ『 デバッグ情報領域がいっぱいです (起動時の **-B オプション**参照)』が表示された場合は、バッファサイズを拡大してください。

● **デバッグ情報タイプ (-XGX オプション)**

『GNU C (Linux etc.)』を選択します。

デバッグ情報パス変換 (-XGX オプション)

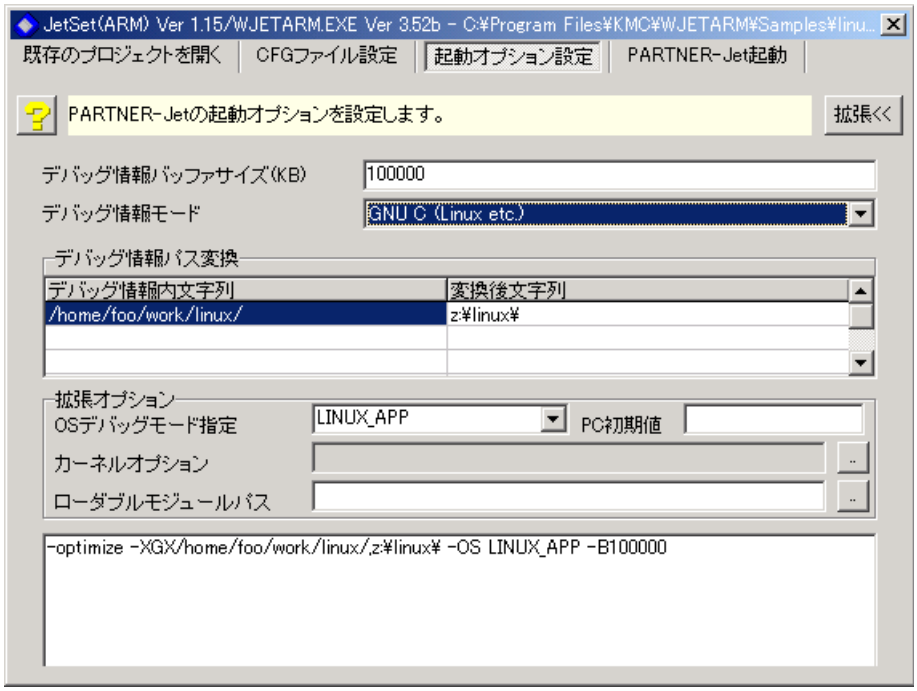
カーネル (vmlinux) をビルドした PATH と Samba でマウントしている PATH が違う場合に指定します。
たとえば、/home/foo/work/linux でカーネルをビルドして、ホスト PC で /home/foo/work を Z: ドライブにマウントした場合は、-XGX/home/foo/work/linux/,z:¥linux¥ と指定します。

● **OS デバッグモード指定 (-OS オプション)**

『LINUX_APP』を選択します。
また、ADD モードでデバッグする場合は、『LINUX_APP_ADD』を選択してください。
ADD モードは PARTNER 起動後、**PSID コマンド** (25 ページ参照) でも切り替えることが可能です。

各オプションについての詳細は、『2.2.2 起動オプション (18 頁)』を参照してください。

図 5-46 起動オプションの設定



(4) PARTNER の起動

JETSET(SH) 上の [起動] ボタンをクリックすることにより、PARTNER が起動します。

図 5-47 PARTNER の起動

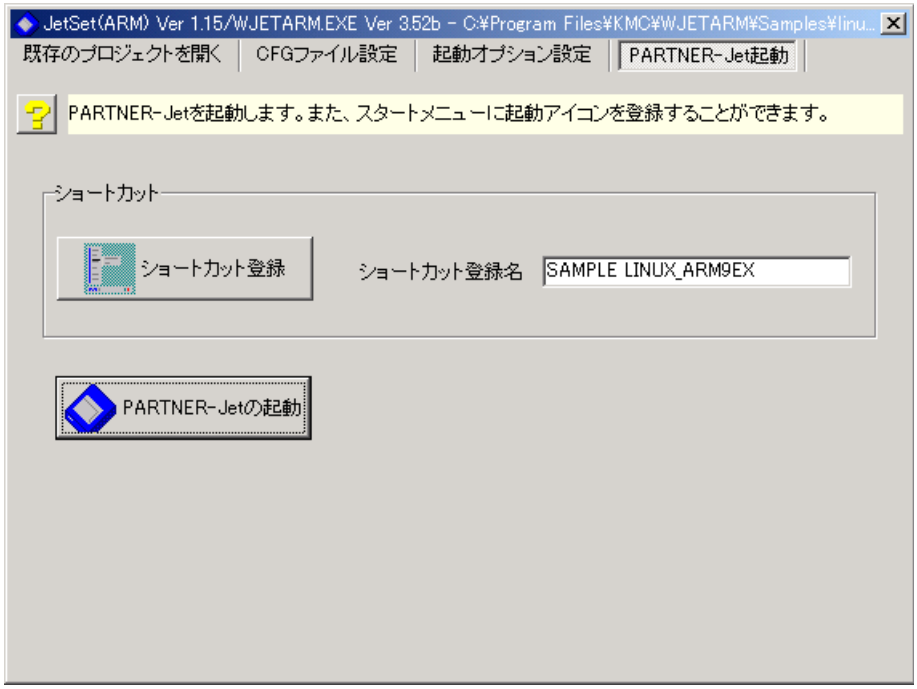
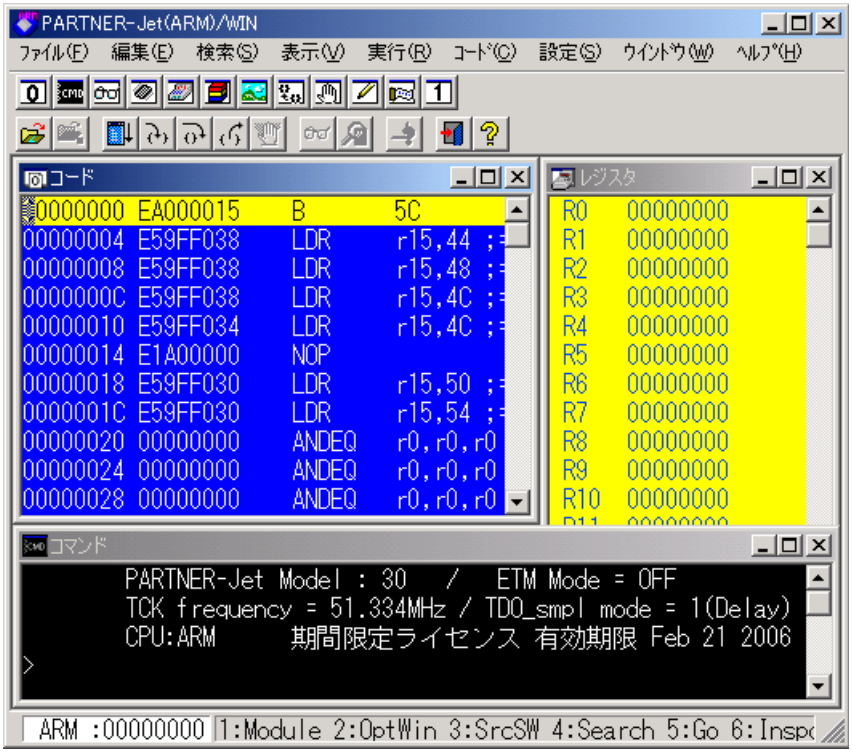


図 5-48 PARTNER の起動画面

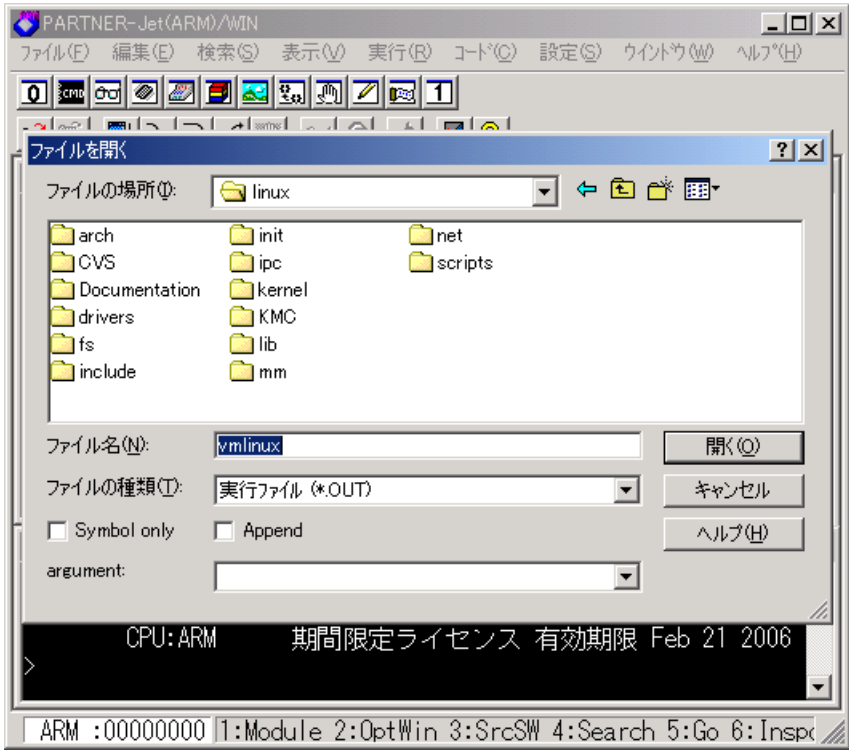


5.5.5 Linux カーネルのロード

『5.5.2 Linux カーネルのビルド (118 頁)』で作成したカーネル(vmlinux)をターゲットメモリにロードします。

```
PT>l vmlinux ↓
```

図 5-49 カーネルのロード



5.5.6 マルチウインドウデバッグ

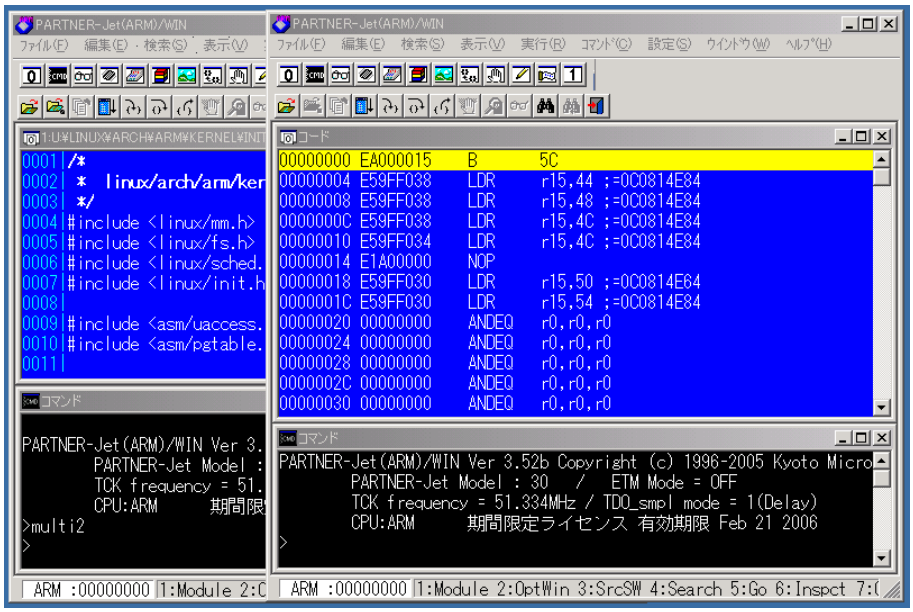
アプリケーション用 PARTNER ウインドウを起動します。

● ADD モードの場合

ADD モードの場合は、デバッグ対象アプリケーションから生成されるスレッドはアプリケーション用 PARTNER ウインドウにアタッチされるため、アプリケーション用 PARTNER ウインドウを 1 つ追加起動します。

```
PT>multi 2 ↓
```

図 5-50 アプリケーション用 PARTNER ウインドウの起動 (ADD モード)



MULTI コマンド (22 ページ参照) や -MULTI オプション (20 ページ参照) で起動した PARTNER ウインドウのウインドウ情報 (ウインドウ配置等) やコマンドヒストリは、新規に生成されたプロジェクトファイル (jetsh_?.jpj) に保存され、次回起動時に利用されます。

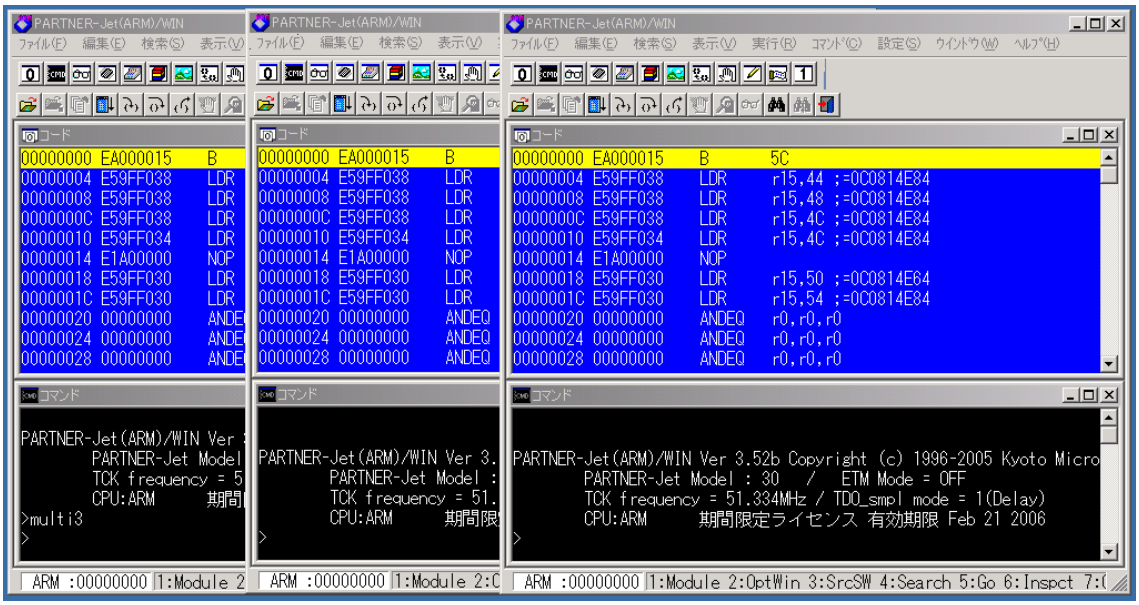
● NON_ADD モードの場合

ADD モードでない場合は、アプリケーション＋生成されるスレッド / プロセスの数の PARTNER ウインドウを起動します。

例えばサンプル fork の場合、カーネル用、アプリケーション用、生成される子プロセス用の 3 つのウインドウが必要となるため、3 つの PARTNER ウインドウを起動します。

```
PT>multi 3 ↓
```

図 5-51 アプリケーション用 PARTNER ウインドウの起動 (NON_ADD モード)



MULTI コマンド (22 ページ参照) や -MULTI オプション (20 ページ参照) で起動した PARTNER ウインドウのウインドウ情報 (ウインドウ配置等) やコマンドヒストリは、新規に生成されたプロジェクトファイル (jetsh_?.jpj) に保存され、次回起動時に利用されます。

5.5.7 Linux カーネルの実行

カーネルのロードが正しくできた状態で、**G コマンド**、または**[実行] ボタン**でロードした vmlinux を実行させます。

```
PT>g ↓
```

Windows PC とターゲットボードが正しく接続されていて、ターミナルソフトが正常に起動していれば、ブートアップメッセージが表示されます。

図 5-52 Linux カーネルブートアップメッセージ

```
NetWinder Floating Point Emulator V0.95 (c) 1998-1999 Rebel.com
Looking up port of RPC 100003/2 on 192.168.1.241
Looking up port of RPC 100005/1 on 192.168.1.241
VFS: Mounted root (nfs filesystem).
Freeing init memory: 212K
INIT: version 2.78 booting
Activating swap...
Checking all file systems...
Parallelizing fsck version 1.22 (22-Jun-2001)
mkdir: cannot create directory `/dev/pts': File exists
Mounting local filesystems...
nothing was mounted
Cleaning: /etc/network/ifstate.
Setting up IP spoofing protection: rp_filter.
Disable TCP/IP Explicit Congestion Notification: done.
Configuring network interfaces: done.
Starting portmap daemon: portmap.
Cleaning: /tmp /var/lock /var/run.
INIT: Entering runlevel: 3
Starting system log daemon: syslogd klogd.
Starting internet superserver: inetd.

MontaVista Linux 3.0, Professional Edition

kzp-arm login:
```

5.5.8 アプリケーションのロード

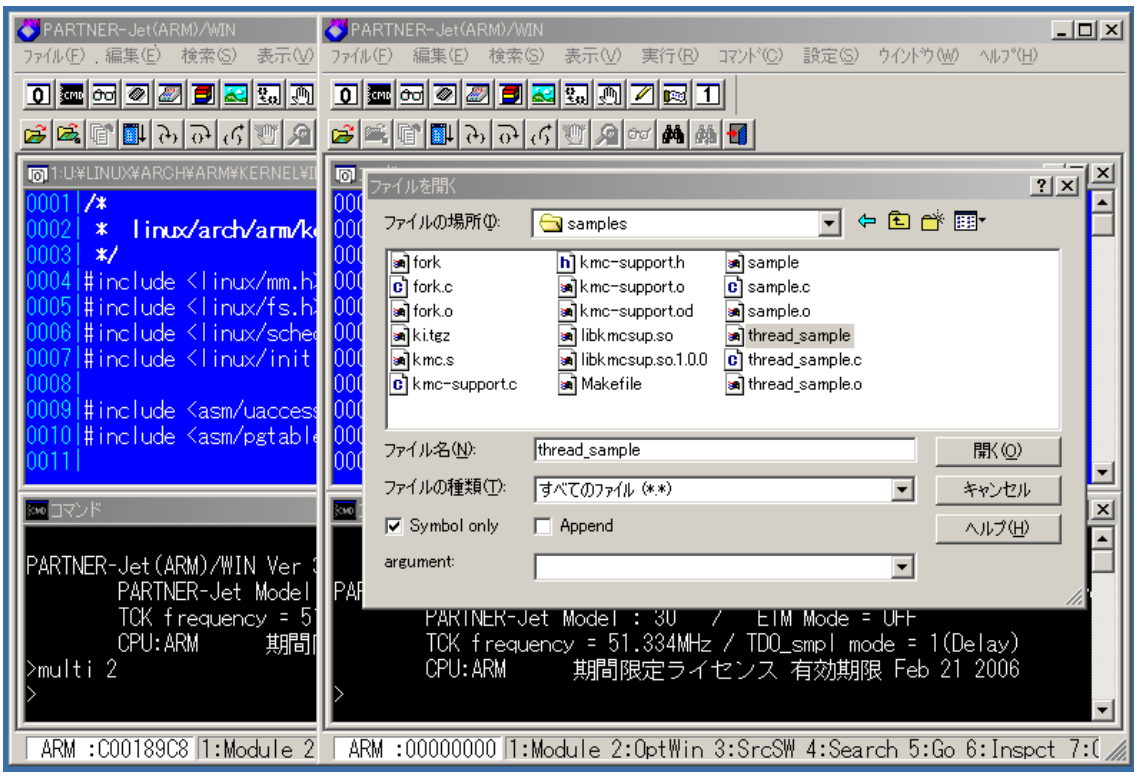
作成したアプリケーションのデバッグ情報を PARTNER にロードします。

● ADD モードの場合

アプリケーション用 PARTNER ウインドウでデバッグ対象アプリケーションのデバッグ情報のみをロードします。
ロード時には必ず [Symbol only] のチェックを行ってください。

```
PT>ls thread_sample ↓
```

図 5-53 アプリケーションのデバッグ情報ロード



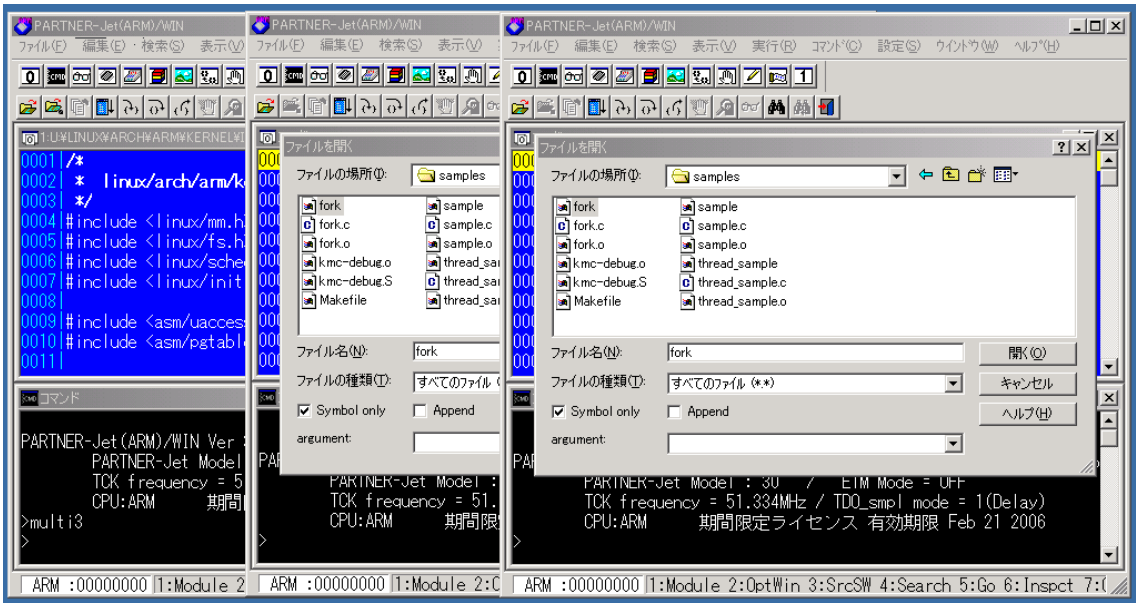
一度ロードされたファイルは、PARTNER がロードファイル名と場所をダイアログとコマンドヒストリに記憶するため、以降の操作を簡略化することができます。

● NON_ADD モードの場合

アプリケーション用 PARTNER ウインドウと子プロセス / スレッド用 PARTNER ウインドウにそれぞれ、デバッグ対象ファイルのデバッグ情報のみをロードします。
ロード時には必ず [Symbol only] のチェックを行ってください。

```
PT>ls fork ↓
```

図 5-54 アプリケーションのデバッグ情報ロード



一度ロードされたファイルは、PARTNER がロードファイル名と場所をダイアログとコマンドヒストリに記憶するため、以降の操作を簡略化することができます。

5.5.9 アプリケーションの実行

ターゲットシステムでデバッグ対象のアプリケーションを実行します。

```
TGT>./fork ↓
```

図 5-55 アプリケーションの実行

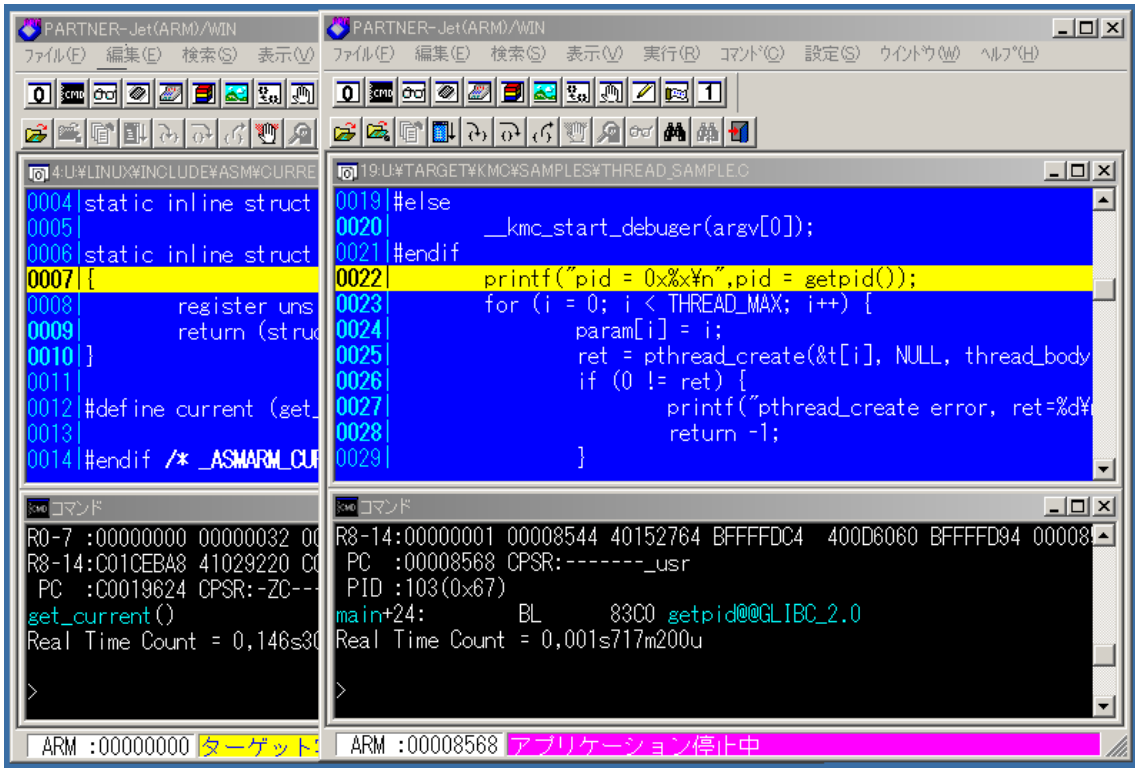
```
expected (0x802/0x203ba), got (0x802/0x203b9)
chmod: changing permissions of `/dev/ttyp1': Input/output error
nfs_refresh_inode: inode number mismatch
expected (0x802/0x203bc), got (0x802/0x203bb)
chmod: changing permissions of `/dev/ttyp3': Input/output error
nfs_refresh_inode: inode number mismatch
expected (0x802/0x63f60), got (0x802/0x205a6)
INIT: Entering runlevel: 3
Starting system log daemon: syslogd klogd.
Starting internet superserver: inetd.
nfs_refresh_inode: inode number mismatch
expected (0x802/0x205a6), got (0x802/0x645a3)

MontaVista Linux 3.0, Professional Edition

kzp-arm login: root
Last login: Thu Jan  1 00:00:20 1970 on console
Linux kzp-arm 2.4.18_mvl30-integrator #390 2005年 3月 31日 木曜日 10:41:07 JST a
rmv4l unknown

Welcome to MontaVista Linux 3.0, Professional Edition

root@kzp-arm:~# cd /KMC/samples/
root@kzp-arm:/KMC/samples# ./fork
```

5.5.10 PARTNER のブレーク

● ADD モードの場合

ターゲットシステムでデバッグ対象のアプリケーションが実行されると、PARTNER は先ず、main() 関数後に挿入したデバッグスタブ関数 (__kmc_start_debuger()) の後ろでブレークします。このとき、アプリケーションの PID や配置情報を自動的に獲得し、以後アプリケーションエリアのメモリ参照やブレークポイントの設定が可能になります。

図 5-56 アプリケーションのアタッチ

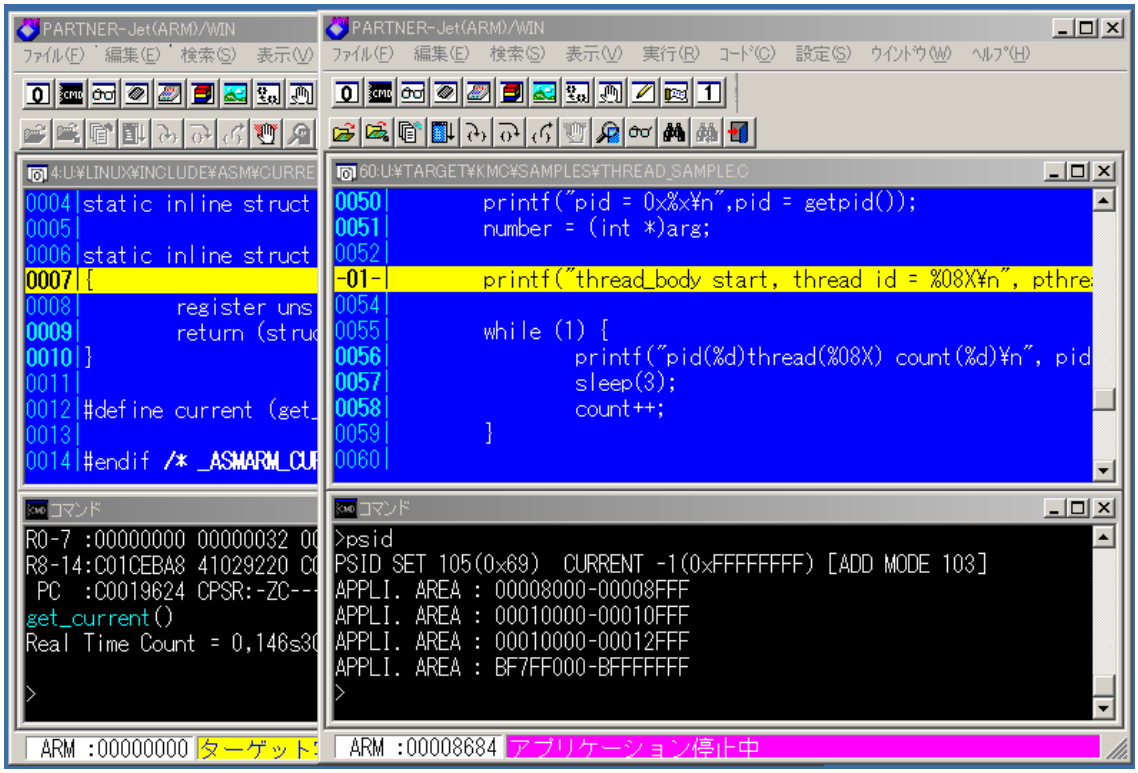


PSID コマンド (25 ページ参照) でデバッガにアプリケーションがアタッチされているか確認します。

```
PSID SET 109(0x6D)  CURRENT 0(0x0) [ADD MODE]
APPLI. AREA : 00400000-00400FFF
APPLI. AREA : 00410000-00410FFF
APPLI. AREA : 00410000-00410FFF
APPLI. AREA : 7BFFF000-7BFFFFFF
```

ここで thread_body() 関数の先頭に挿入したデバッグスタブ関数の後にブレークポイントを設定し、アプリケーションを実行すると、生成スレッドが自動的にアタッチされブレークします。

図 5-57 アプリケーションのアタッチ (生成スレッド)



PSID コマンド (25 ページ参照) を実行するとアプリケーション用 PARTNER ウインドウで生成スレッドがアタッチされていることが確認できます。

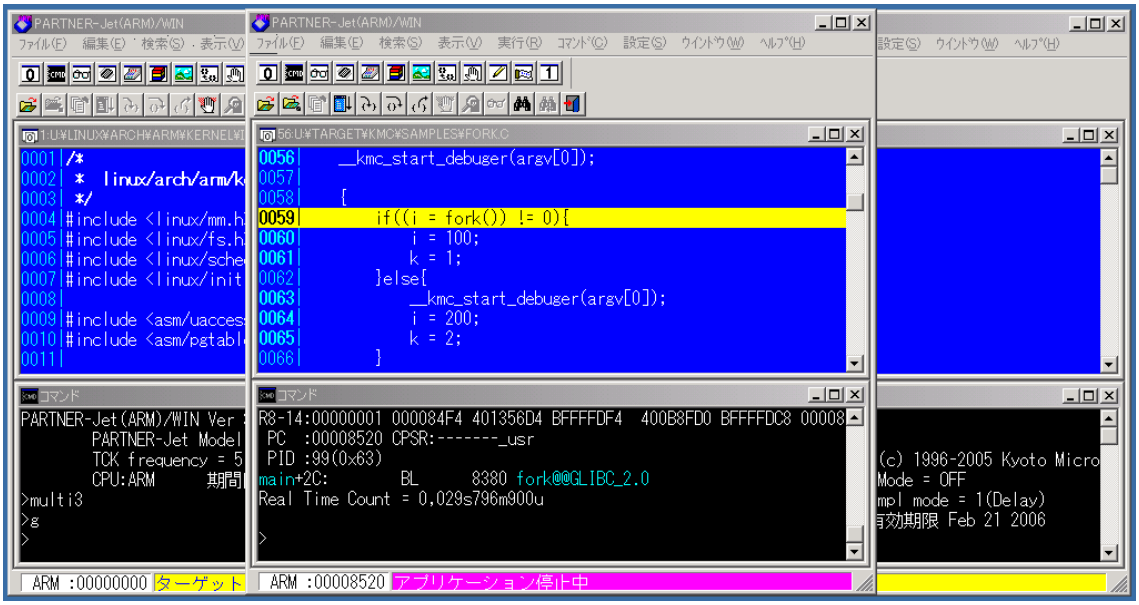
```
PT>psid ↓
PSID SET 111(0x6F)  CURRENT -1(0xFFFFFFFF) [ADD MODE 109,112]
APPLI. AREA : 00400000-00400FFF
APPLI. AREA : 00410000-00410FFF
APPLI. AREA : 00410000-00412FFF
APPLI. AREA : 7B7FF000-7BFFFFFF
>
```

以後、生成されるスレッドは、アプリケーション用 PARTNER ウインドウにアタッチされ、スレッドのメモリ参照 / 変更やブレークポイントの設定など通常のデバッグ操作がアタッチしたスレッドに対して可能になります。

● NON_ADD モードの場合

ターゲットシステムでデバッグ対象のアプリケーションが実行されると、アプリケーション用 PARTNER ウインドウがデバッグスタブ関数の後ろでブレークします。このとき、アプリケーションの PID や配置情報を自動的に獲得し、以後アプリケーションエリアのメモリ参照やブレークポイントの設定が可能になります。

図 5-58 アプリケーションのアタッチ（親プロセス）

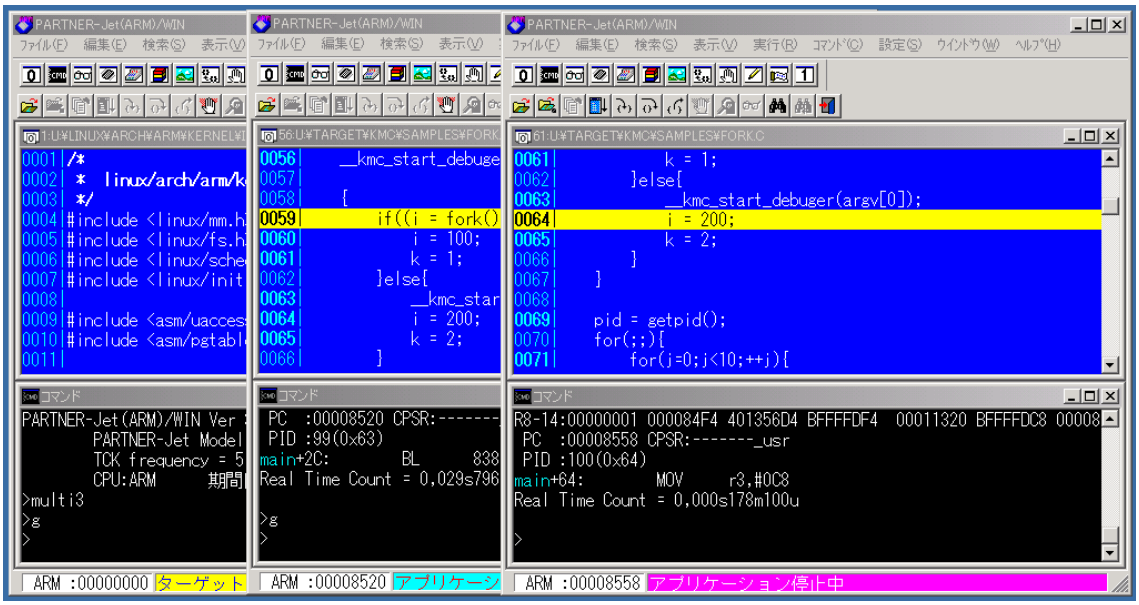


PSID コマンド（25 ページ参照）を実行するとアプリケーション用 PARTNER ウインドウで親プロセスがアタッチされていることが確認できます。

```
PT>psid ↓
PSID SET 112(0x70)  CURRENT 112(0x70)
APPLI. AREA : 00400000-00400FFF
APPLI. AREA : 00410000-00410FFF
APPLI. AREA : 00410000-00412FFF
APPLI. AREA : 7B7FF000-7BFFFFFF
```

次に、アプリケーション用 PARTNER ウインドウで fork() 関数が実行され子プロセスが生成されると、子プロセス用 PARTNER ウインドウがデバッグスタブ関数の後ろでブレークします。このとき、子プロセスの PID や配置情報を自動的に獲得し、以後子プロセスエリアのメモリ参照やブレークポイントの設定が可能になります。

図 5-59 アプリケーションのアタッチ (子プロセス)



PSID コマンド (25 ページ参照) を実行するとアプリケーション用 PARTNER ウインドウで子プロセスがアタッチされていることが確認できます。

```
PT>psid ↓
PSID SET 113(0x71)  CURRENT 113(0x71)
APPLI. AREA : 00400000-00400FFF
APPLI. AREA : 00410000-00410FFF
APPLI. AREA : 00410000-00412FFF
APPLI. AREA : 7B7FF000-7BFFFFFF
```

PARTNERが正しくブレークしない場合､カーネルのコンフィグレーションが間違っている可能性があります『 5.5.1 Linux カーネルのコンフィグレーション (118 頁)』の設定と『 5.5.4 アプリケーションモードでの PARTNER の起動 (121 頁)』で指定した OS デバッグモードがアプリケーションモードになっているか確認してください。
また、アプリケーションソースにデバッグスタブ関数 (_kmc_start_debuger()) が挿入されているか、サポートファイル (kmc-support.c) がリンクされているか確認してください。

5.6 共有ライブラリのデバッグ

共有ライブラリはアプリケーションの一部です。したがって、所属するアプリケーションでデバッグ可能な状態にしておき、そのアプリケーションの一部としてデバッグします。そのため、共有ライブラリのデバッグを行う場合は、アプリケーションをアタッチしておく必要があります。

この節では、glibc をデバッグする場合を例として、共有ライブラリデバッグ方法を次の流れで説明します。

(1) **共有ライブラリにデバッグ情報（135 頁）**
(2) **アプリケーションのアタッチ（135 頁）**
(3) **共有ライブラリのデバッグ情報読み込み（136 頁）**

5.6.1 共有ライブラリにデバッグ情報

デバッグ対象となる共有ライブラリにデバッグ情報を付加します。
デバッグ情報はカーネルやアプリケーションと同じフォーマットを選択してください。PARTNER で正しくデバッグ情報を読み込めない場合は、他のフォーマットを試してみてください。

5.6.2 アプリケーションのアタッチ

『5.2 カーネルモードでのアプリケーションデバッグの手順（73 頁）』、『5.3 カーネルモードでのマルチプロセス / マルチスレッドアプリケーションデバッグ手順（85 頁）』、『5.4 アプリケーションモードでのアプリケーションデバッグ（104 頁）』、『5.5 アプリケーションモードでのマルチプロセス / マルチスレッドアプリケーションのデバッグ（117 頁）』を参照して、アプリケーション用 PARTNER ウインドウにアプリケーションがアタッチされている状態にします。

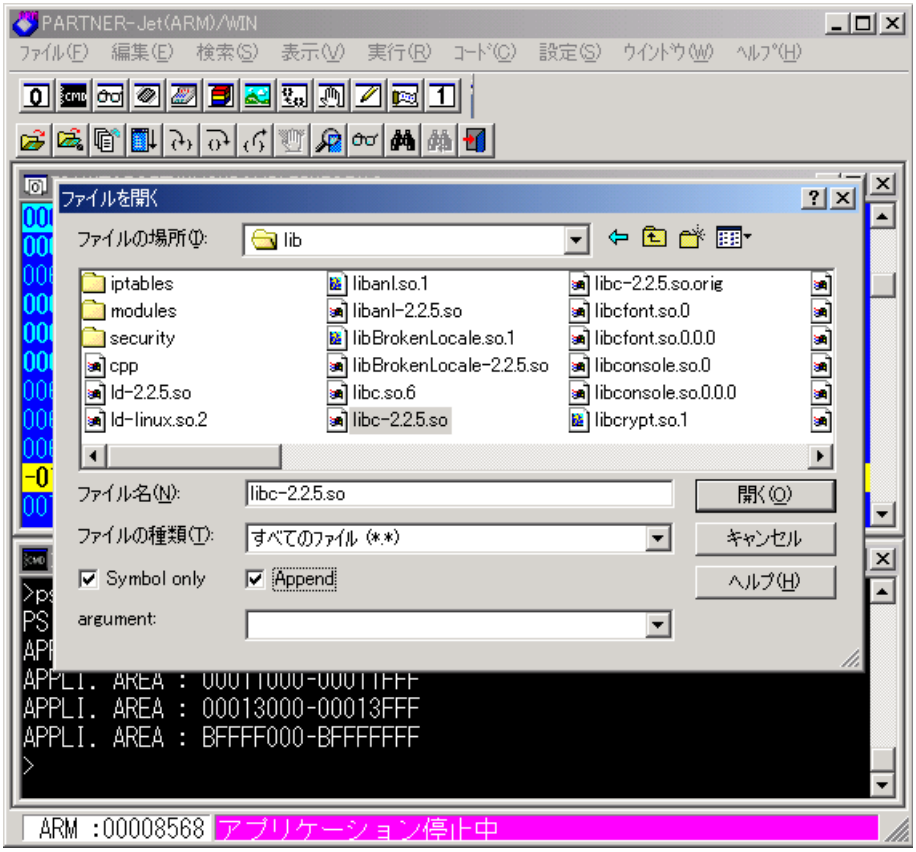
5.6.3 共有ライブラリのデバッグ情報読み込み

『5.6.1 共有ライブラリにデバッグ情報（135 頁）』で作成した共有ライブラリのデバッグ情報を読み込みます。

ロード時には必ず [Symbol only] のチェックと [Append] のチェックを行ってください。

```
PT>lsa libc-2.2.5.so ↓
```

図 5-60 共有ライブラリのデバッグ情報読み込み



デバッグ情報の読み込みが完了すると、共有ライブラリのソースレベルデバッグが出来るようになります。

共有ライブラリ内にブレークポイントを設定して、アプリケーションを実行すると、設定したブレークポイントでブレークします。

その時点で **PSID コマンド**（25 ページ参照）を実行すると、共有ライブラリ空間が新たに PARTNER ウインドウにアタッチされたことが判断できます。

```
PT>psid ↓
PSID SET 113(0x71)  CURRENT 113(0x71)
APPLI. AREA : 00400000-00400FFF
APPLI. AREA : 00410000-00410FFF
APPLI. AREA : 00410000-00412FFF
APPLI. AREA : 7B7FF000-7BFFFFFF
APPLI. AREA : 2959F000-296C0FFF
```

5.7 Linux OS 対応ヒストリ表示

PARTNER のリアルタイムトレース機能には Linux OS に対応したヒストリ表示が可能です。
マルチウインドウで、カーネル、アプリケーションを別々の PARTNER ウインドウでデバッグしている場合、それぞれアタッチしている (デバッグ情報を読み込んでいる) プロセスのヒストリのみ表示させることが出来ます。

Linux OS 対応ヒストリ表示は、カーネルモードでデバッグしてください。

この節では、サンプル (sample) を使用した場合を例として、カーネルモードでのアプリケーション (プロセス) のヒストリ表示を次の流れで説明します。

(1) Linux カーネルのコンフィグレーション（138 頁）
(2) Linux カーネルのビルド（138 頁）
(3) アプリケーションのアタッチ（138 頁）
(4) Linux OS 対応ヒストリ表示（139 頁）

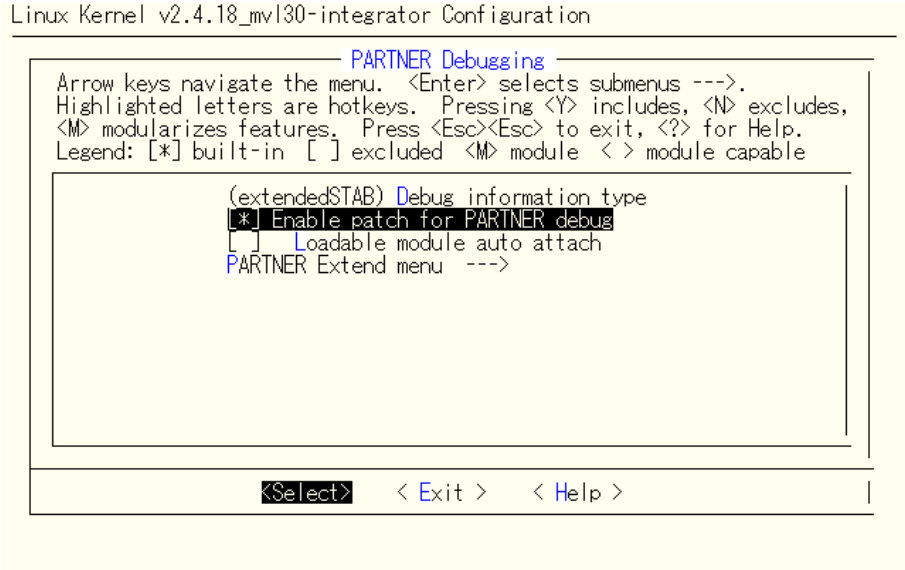
5.7.1 Linux カーネルのコンフィグレーション

カーネルモードで Linux OS 対応ヒストリ表示を行う場合は、Linux カーネルのコンフィグレーションで [Enable patch for PARTNER debug] を有効にします。
また、カーネルのデバッグ情報のフォーマットを指定します。
推奨は extendedSTAB(-gstabs+) です。PARTNER で実際カーネルを読み込んでデバッグ情報がおかしなときは、DWARF2 などに変えて試みてください。

```
LINUX86>make menuconfig ↓
```

[Kernel hacking]->[PARTNER Debugging]->[Debug infomation type]
[Kernel hacking]->[PARTNER Debugging]->[Enable patch for PARTNER debug]

図 5-61 Linux カーネルのコンフィグレーション



5.7.2 Linux カーネルのビルド

上記のコンフィグレーションで、Linux カーネル(vmlinux)を作成します。

```
LINUX86>make ↓
```

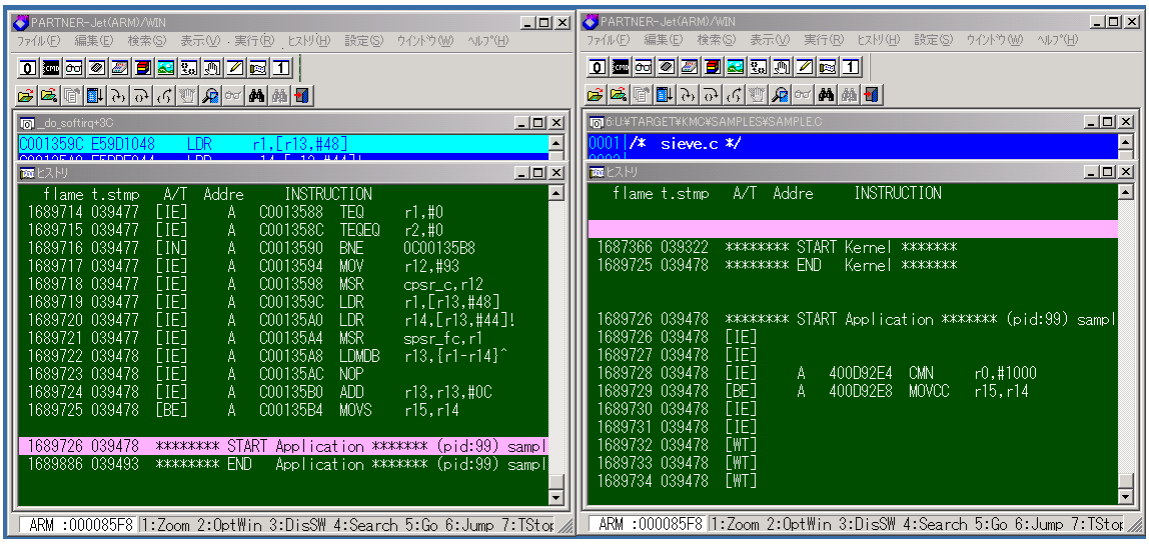
5.7.3 アプリケーションのアタッチ

『5.2 カーネルモードでのアプリケーションデバッグの手順 (73 頁)』を参考にしてデバッグ環境を整えます。

5.7.4 Linux OS 対応ヒストリ表示

CPU をブレークし、ヒストリを表示します。

図 5-62 LinuxOS 対応ヒストリ表示



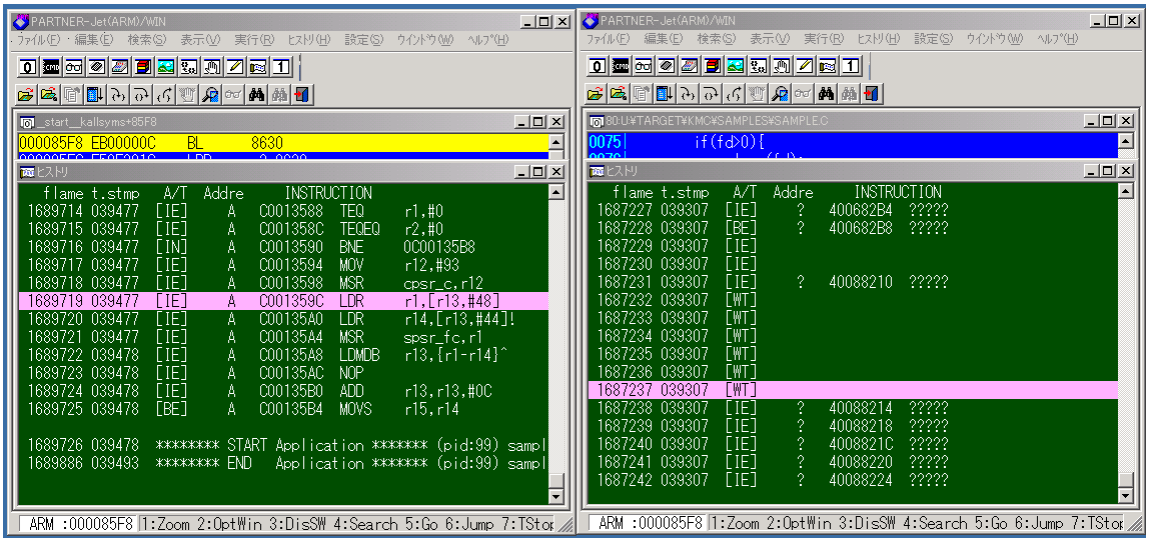
ヒストリ表示には、Linux OS 対応にあたって以下の行が表示されるようになっています。

```
**** START Kernel ****
**** END   Kernel ****
**** START Application **** (pid:xx) app_name
**** END   Application **** (pid:xx) app_name
```

上記の行をダブルクリックすると、該当カーネル / アプリケーションをデバッグしている PARTNER ウインドウがある場合、その PARTNER ウインドウにフォーカスを移し、同一フレーム番号を表示します。

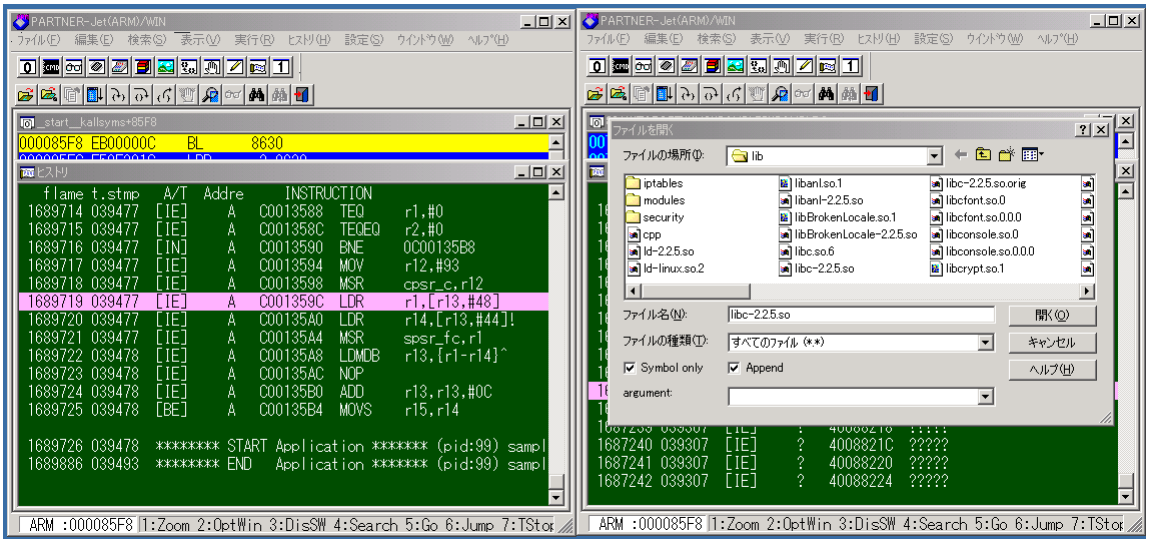
アプリケーション用 PARTNER ウインドウのヒストリ表示内で、????? 表示されている個所があります。
これは、共有ライブラリなどデバッグ中のプロセスでデバッグ情報が読み込まれていない領域になります。

図 5-63 不明なヒストリ表示



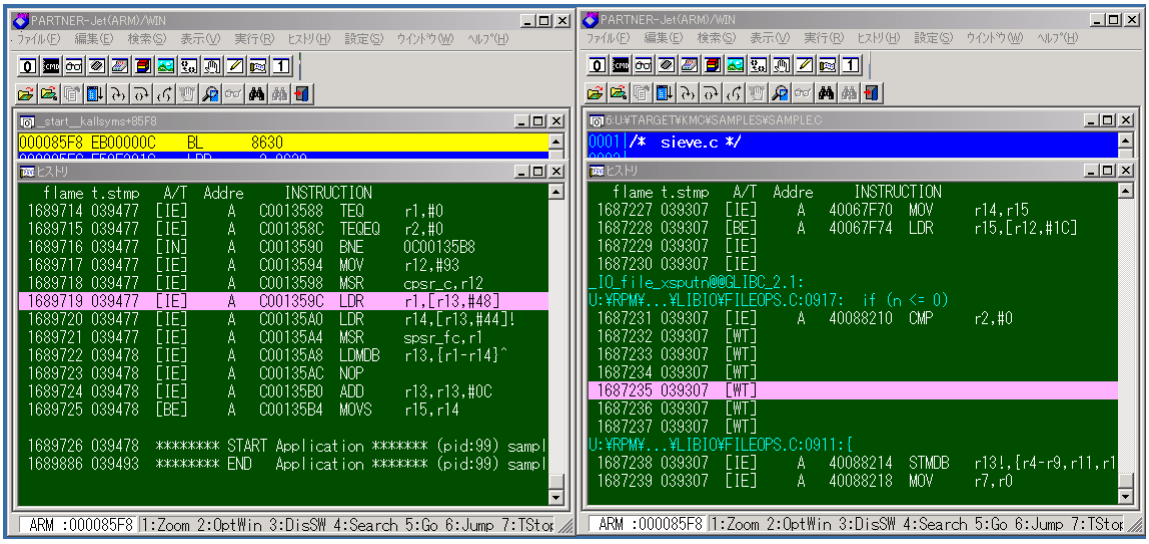
????? 表示を解消するには、該当アドレスの空間のデバッグ情報を読み込みます。(**MAPS コマンド** (28 ページ参照) で確認できます)

図 5-64 デバッグ情報の読み込み



デバッグ情報の読み込みが完了すると、????? 表示だった個所が解消され、通常表示になります。

図 5-65 ????? が解消された表示



現在のヒストリ内容でPARTNERがアタッチしていないプロセスのヒストリ表示を行うには、次の手順で行います。

1.PARTNERから現在のトレースデータをバイナリデータでセーブ

PT>tdsbin ↓

2.MULTIコマンドで新しくPARTNERウインドウを起動

3.新しく起動したPARTNERウインドウで目的のプロセスのデバッグ情報をロード

PT>ls＜ファイル名＞↓

4.新しく起動したPARTNERウインドウに目的のPIDを設定

PT>psid test <PID> ↓

5.新しく起動したPARNTERウインドウに1.でセーブしたトレースデータをロード

PT>tdlbin ↓

6.ヒストリウインドウに目的のリアルタイムトレース表示がされます。

なお、この方法はヒストリウインドウに目的プロセスのヒストリを参照できるだけで、デバッグは出来ません。

5.8 実行中のアプリケーションのアタッチ

PARTNER はターゲットシステム上で既に実行されているアプリケーションをアタッチし、デバッグ可能状態にすることが出来ます。
ただし、アプリケーションのアタッチには、glibc の修正が必要です。

この節では、実行中のアプリケーションをアタッチし、デバッグする方法を次の流れで説明します。

5.8.1 ターゲットシステムの glibc の修正

実行中のアプリケーションをアタッチするためには、glibc にサポートファイル(kmc-support.c)をリンクする必要があります。
次の手順で、PARTNER アタッチ用 glibc を作成します。

1. サポートファイル(kmc-support.c)のコピー
   glibc ビルド環境の、glibc-x.x.x/sysdep/unix/sysv/linux/sh にサポートファイル(kmc-support.c)をコピーします。

2. Makefile の修正
   glibc-x.x.x/csu/Makefile 内に、routines += kmc-support の 1 行を追加します。

【例】

```
 routines = init-first libc-start $(libc-init) sysdep version check_fds
+routines += kmc-support
 csu-dummies = $(filter-out $(start-installed-name),crt1.o Mcrt1.o)
```

3. glibc の作成
   以下のコマンドで、glibc の作成を行ってください。

```
LINUX86>make clean ↓
LINUX86>make build ↓
```

glibc-x.x.x/csu/Makefile に挿入した 1 行をコメントアウトすれば、サポートファイル(kmc-support.c)を削除しなくても、元通りのビルドが可能です。

kmc-support.c 内で "Select Target CPU type" のコンパイルエラーが発生した場合は、kmc-support.c 内の CPUTYPE シンボル宣言をターゲット CPU のみ有効にして再コンパイルしてください。

4. glibc のチェック
   作成した glibc がサポートファイル(kmc-support.c)を正しくリンクしたか確認します。

```
LINUX86>nm libc.so.x | grep  kmc_sleep_thread ↓
```

   シンボルが表示されれば OK です。

5. glibc のインストール
   作成した glibc をターゲットシステム上にインストールしてください。

5.8.2 PARTNER への glibc の登録

LINUX コマンドで、『5.8.1 ターゲットシステムの glibc の修正 (144 頁)』で作成した glibc のファイル名とサポートファイル(kmc-support.c)内シンボルアドレス(__kmc_sleep_thread)を指定します。

PT>LINUX set attach offset <ライブラリ名> <__kmc_sleep_thread のアドレス> ↓

__kmc_sleep_thread のアドレスは、『glibc のチェック (144 頁)』で確認したアドレスを指定してください。

【例】

```
LINUX86>nm libc-2.2.5.so | grep  __kmc_sleep_thread ↓
0010ba34 t __kmc_sleep_thread

PT>linux set attach offset libc-2.2.5.so 0x10ba34 ↓
    SET LINUX ATTACH OFFSET : libc-2.2.5.so(0x0010BA34)
```

サポート ファイル(kmc-support.c)をリンクしたライブラリ (glibc)が変更された場合以外は、この LINUX コマンドを省略することが出来ます。

5.8.3 カーネル / アプリケーションの実行

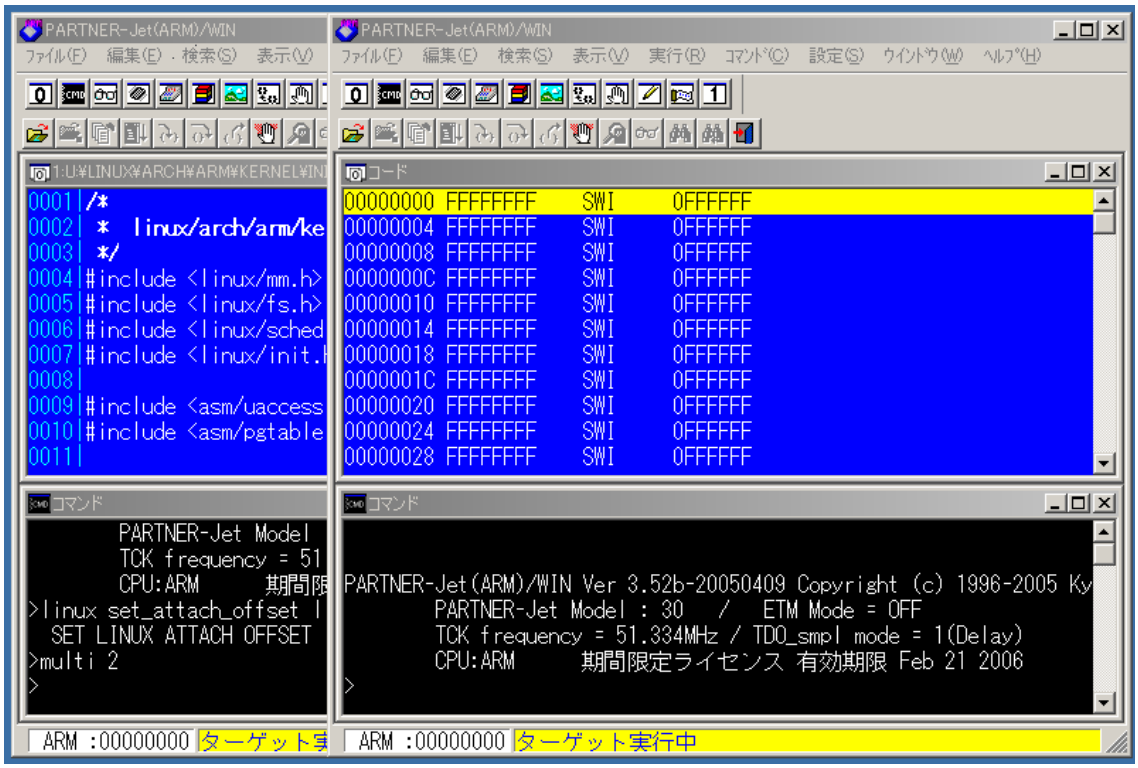
『5.2 カーネルモードでのアプリケーションデバッグの手順 (73 頁)』もしくは『5.4 アプリケーションモードでのアプリケーションデバッグ (104 頁)』を参照し、PARTNER からカーネルを実行し、デバッグ対象アプリケーションも実行しておきます。

5.8.4 アプリケーション用 PARTNER ウインドウの起動

MULTI コマンド (22 ページ参照) で、アプリケーション用 PARTNER ウインドウを起動します。

```
PT>MULTI 2 ↓
```

図 0-1 アプリケーション用ウインドウの起動



MULTIコマンド (22 ページ参照) や -MULTIオプション (20 ページ参照) で起動した PARTNER ウインドウのウインドウ情報 (ウインドウ配置等) やコマンドヒストリは、新規に生成されたプロジェクトファイル (jetsh_?.jpj) に保存され、次回起動時に利用されます。

5.8.5 実行しているアプリケーションの PID の確認

PS コマンド (27 ページ参照) で、デバッグしたいアプリケーションの PID を確認します。

```
PT>ps ↓
  1(0x1)      /sbin/init
  60(0x3c)    /sbin/portmap
  88(0x58)    /sbin/syslogd
  90(0x5a)    /sbin/klogd
  95(0x5f)    /usr/sbin/inetd
  96(0x60)    /bin/bash
  99(0x63)    /KMC/samples/sample
```

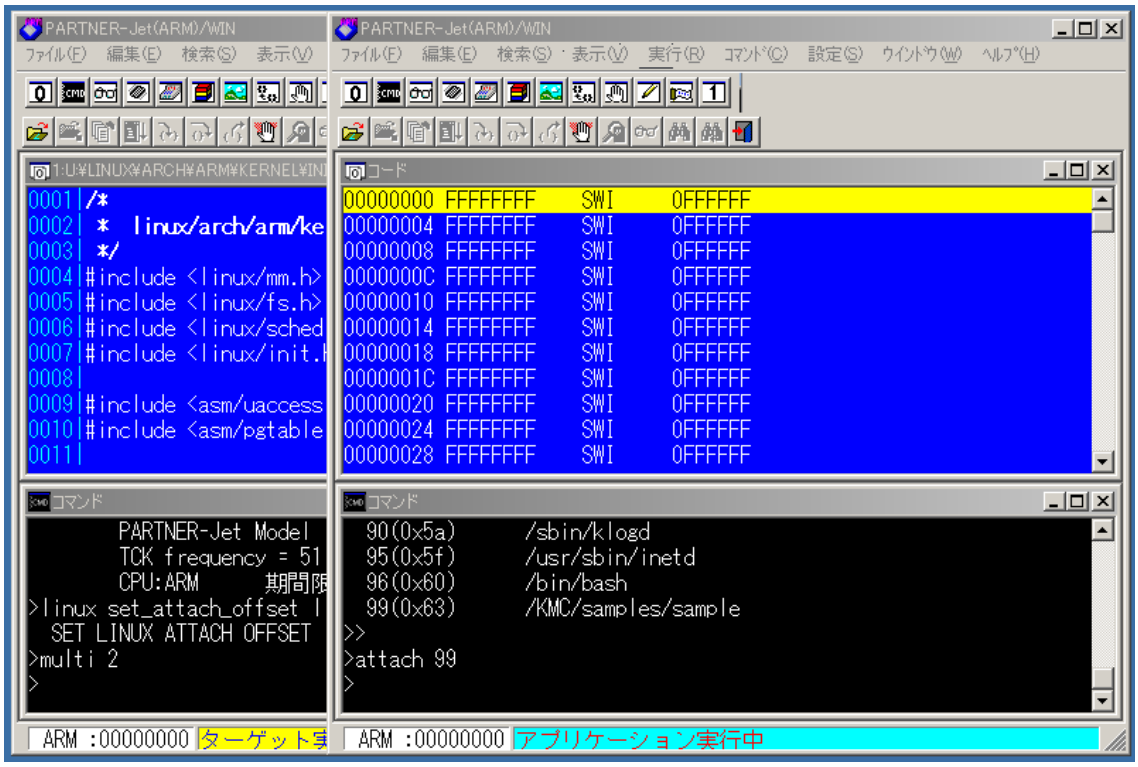
5.8.6 実行しているアプリケーションのアタッチ

アプリケーション用 PARTNER ウインドウで **ATTACH コマンド** (29 ページ参照) を実行し、デバッグしたいアプリケーションをアタッチします。

```
PT>attach 99 ↓
```

アプリケーションモードで起動している場合、アタッチが完了するとステータスバーが**【ターゲット実行中】**から**【アプリケーション実行中】**になります。カーネルモードで起動している場合は、変化しません。

図 5-66 アプリケーションモードでのアタッチ



PSID コマンド (25 ページ参照) でアタッチが完了していることが確認できます。

```
PT>psid ↓
PSID SET 99(0x63)  CURRENT -1(0xFFFFFFFF)
APPLI. AREA : 00400000-00401FFF
APPLI. AREA : 00411000-00451FFF
APPLI. AREA : 00453000-00453FFF
APPLI. AREA : 7BFFF000-7BFFFFFF
```

アタッチが正常に出来ない場合、デバッグ対象アプリケーションが使用している glibc が『 ターゲットシステムの glibc の修正(144 頁)』の修正を施していないか、『 PARTNER への glibc の登録(145 頁)』で登録した

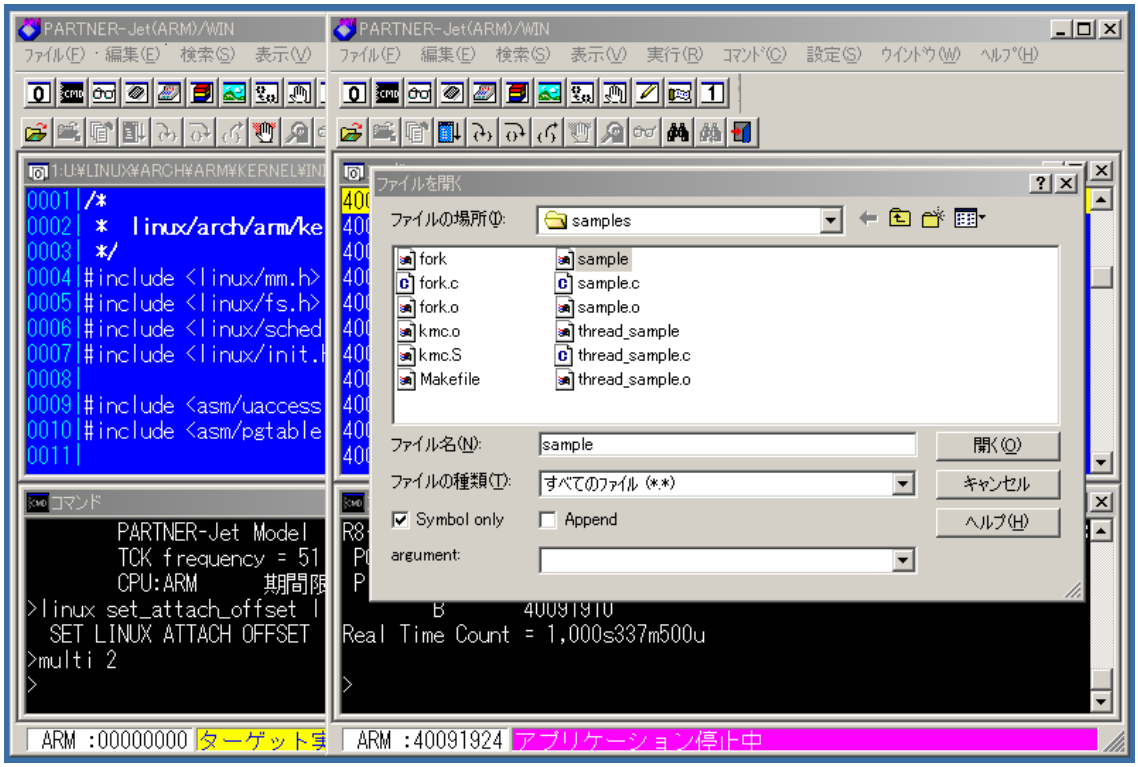
以降、glibc が変更されている可能性があります。glibc が変更されている場合は、再度登録を行ってください。

5.8.7 アプリケーションのデバッグ情報のロード

アプリケーション用 PARTNER ウインドウでアタッチしたアプリケーションのデバッグ情報のみをロードします。
ロード時には必ず [Symbol only] のチェックを行ってください。

```
PT>ls sample ↓
```

図 5-67 アプリケーションのデバッグ情報ロード



デバッグ情報のロードが完了した時点で、実行中のアプリケーションのソースレベルデバッグが可能になります。

一度ロードされたファイルは、PARTNER がロードファイル名と場所をダイアログとコマンドヒストリに記憶するため、以降の操作を簡略化することができます。

付録

ここでは、Linux カーネルの修正が出来ない場合のローダブルモジュール、アプリケーションのデバッグ方法やその他デバッグ手法を説明します。

付録 A　手動モジュールデバッグ

カーネルの修正が出来ない場合や、ローダブルモジュールの修正が出来ない場合ののローダブルモジュールのデバッグ手順を説明します。

この節では、RAM ディスク (rd.o) を使用した場合を例として、ローダブルモジュールのデバッグ方法を次の流れで説明します。

A-1　ローダブルモジュールの作成

デバッグ対象のローダブルモジュールをデバッグ情報付きで作成します。

A-2　ターゲット上でローダブルモジュールの MAP ファイル作成

『4.1 カーネルのデバッグ手順（50 頁）』の手順でカーネルを実行し、作成したローダブルモジュールをターゲットボードにインストールします。次の手順により行います。

(1) Linux カーネルの起動

『4.1 カーネルのデバッグ手順（50 頁）』の手順でデバッグ環境を構築し、Linux カーネルを起動します。

(2) モジュールのマップファイルの作成

ターゲットの Linux 上で、**insmod コマンド**を使用してローダブルモジュール（rd.o）のインストールを行い、**insmod コマンド**の -m オプションによってマップファイル（rd.map）を作成します。
作成する MAP ファイル名は、インストールするローダブルモジュールファイルと同じ名前で、拡張子は .map に指定してください。また、作成するディレクトリは、ローダブルモジュールと同じディレクトリにしてください。

```
TGT>insmod -m rd.o >rd.map ↓
```

図 A-1 ローダブルモジュールの MAP ファイル作成

```
nothing was mounted
Cleaning: /etc/network/ifstate.
Setting up IP spoofing protection: rp_filter.
Disable TCP/IP Explicit Congestion Notification: done.
Configuring network interfaces: done.
Starting portmap daemon: portmap.
Cleaning: /tmp /var/lock /var/run.
nfs_refresh_inode: inode number mismatch
expected (0x802/0x203ab), got (0x802/0x205a6)
INIT: Entering runlevel: 3
Starting system log daemon: syslogd klogd.
Starting internet superserver: inetd.

MontaVista Linux 3.0, Professional Edition

kzp-arm login: root
Last login: Thu Jan  1 00:00:28 1970 on console
Linux kzp-arm 2.4.18_mvl30-integrator #398 2005年 3月 31日 木曜日 15:55:38 JST a
rmv4l unknown

Welcome to MontaVista Linux 3.0, Professional Edition

root@kzp-arm:~# insmod -m rd.o >rd.map
RAMDISK driver initialized: 16 RAM disks of 4096K size 1024 blocksize
root@kzp-arm:~#
```

マップファイルを作成した後、rmmod コマンドを使用して、ローダブルモジュール(rd.o)をアンロードしてください。

```
TGT>rmmod rd ↓
```

図 A-2 ローダブルモジュールのアンロード

```
Cleaning: /etc/network/ifstate.
Setting up IP spoofing protection: rp_filter.
Disable TCP/IP Explicit Congestion Notification: done.
Configuring network interfaces: done.
Starting portmap daemon: portmap.
Cleaning: /tmp /var/lock /var/run.
nfs_refresh_inode: inode number mismatch
expected (0x802/0x203ab), got (0x802/0x205a6)
INIT: Entering runlevel: 3
Starting system log daemon: syslogd klogd.
Starting internet superserver: inetd.

MontaVista Linux 3.0, Professional Edition

kzp-arm login: root
Last login: Thu Jan  1 00:00:28 1970 on console
Linux kzp-arm 2.4.18_mvl30-integrator #398 2005年 3月 31日 木曜日 15:55:38 JST a
rmv4l unknown

Welcome to MontaVista Linux 3.0, Professional Edition

root@kzp-arm:~# insmod -m rd.o >rd.map
RAMDISK driver initialized: 16 RAM disks of 4096K size 1024 blocksize
root@kzp-arm:~# rmmod rd
root@kzp-arm:~#
```

PARTNER では、マップファイルから読み込んだモジュールの先頭アドレスとサイズから、モジュールのリロケーションを自動的に計算します。
したがって、モジュールの内容を変更した場合でも、モジュールの先頭アドレスに変更がなければ、次の場合を除いて、マップファイルを作成しなおす必要はありません。
マップファイルを再生成する必要がある場合は次のとおりです。

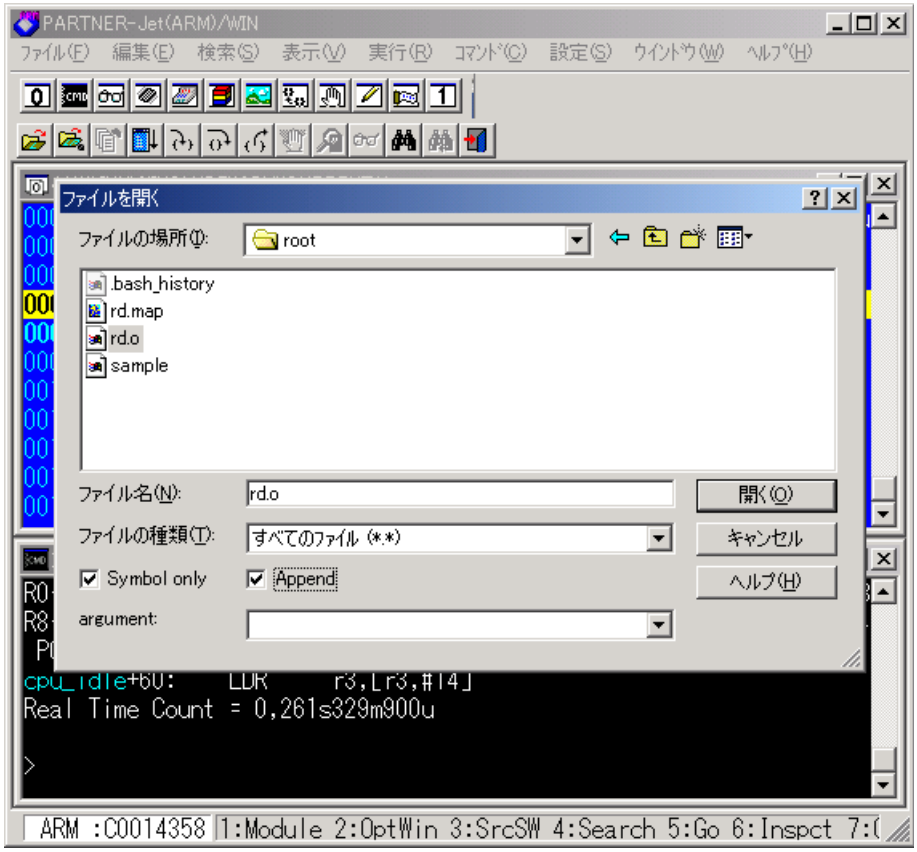
- モジュールをインストール(insmod)する順番を変更した場合
- モジュールのサイズが 4K バイト 以上変化した場合
- カーネルのサイズが 4K バイト 以上変化した場合

A-3　ローダブルモジュールのデバッグ情報の読み込み

動作しているカーネルを [ESC] キーで停止し、ローダブルモジュールのデバッグ情報を PARTNER にロードします。
ロード時には必ず [Symbol only] のチェックと [Append] のチェックを行ってください。

PT>lsa rd.o ↓

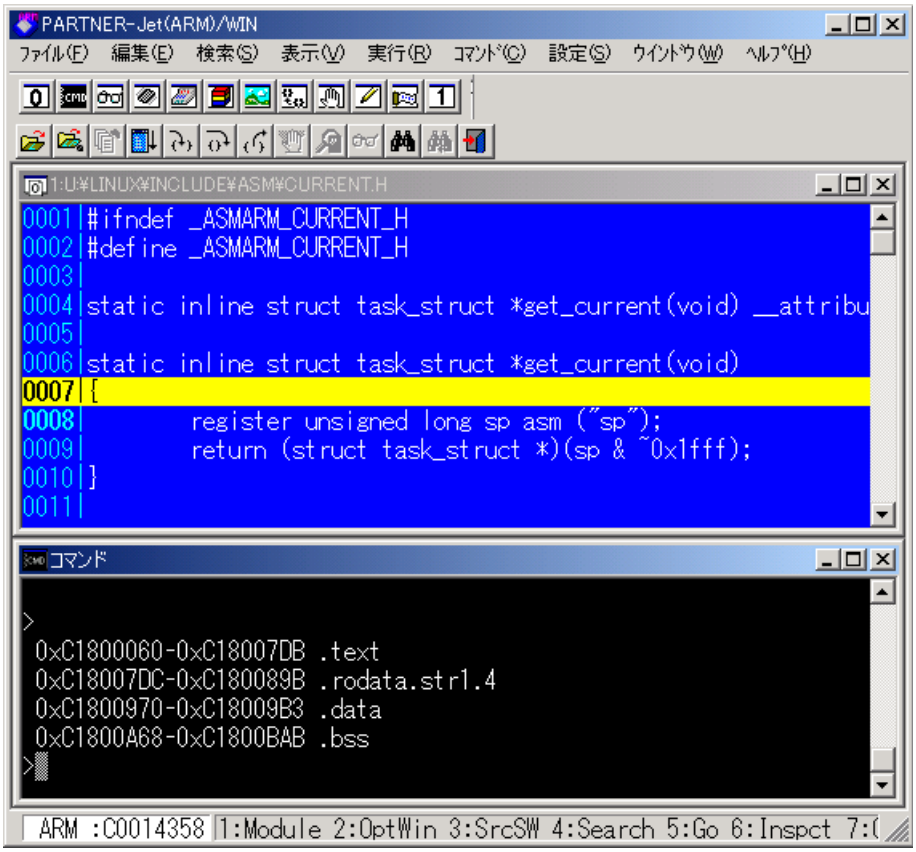
図 A-3 ローダブルモジュールデバッグ情報のロード



一度ロードされたファイルは、PARTNER がロードファイル名と場所をダイアログとコマンドヒストリに記憶するため、以降の操作を簡略化することができます。

正常にデバッグ情報がロードされると、ローダブルモジュール(rd.o)の各セクションのアドレスが表示されます。まだ、この時点では、ローダブルモジュールのメモリ参照は出来ません。

図 A-4 セクション情報の表示



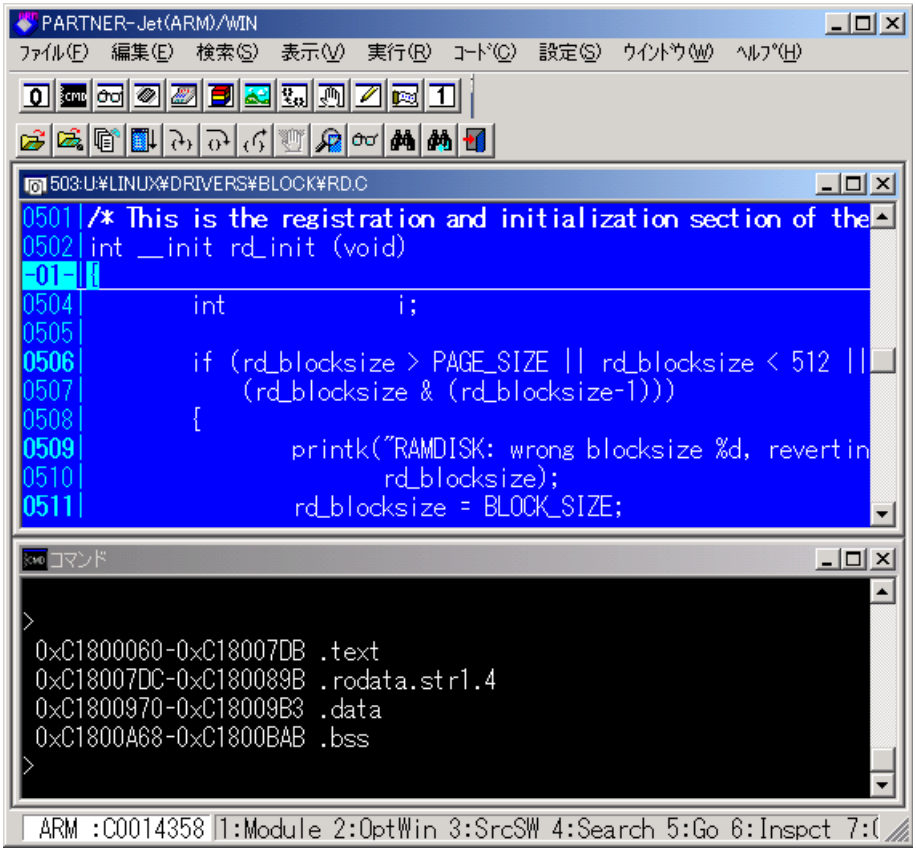
デバッグ情報ロード時に、エラーメッセージ『指定ファイルがありません』が表示された場合は、ロードしたローダブルモジュールと同じディレクトリに MAP ファイルが存在しないしません。再度、『A-2 ターゲット上でローダブルモジュールの MAP ファイル作成(153 頁)』の MAP ファイル作成を行ってください。

A-4　ブレークポイントの設定

ローダブルモジュール(rd.o)の初期化部分(rd_init 関数)にブレークポイントを設定します。
ブレークポイントの設定は、1 点だけにしてください。

```
PT>bp rd_init ↓
```

図 A-5 初期化関数へのブレークポイント設定



ブレークポイント 設定時に『 Verify エラー』メッセージが表示された場合は、既にブレークポイント が設定されています。設定できるブレークポイント は 1 点のみです。ブレークポイント を削除して、ローダブルモジュールの初期化部分にブレークポイント を設定して ください。

A-5 ローダブルモジュールのインストール

カーネルを実行し、ターゲットシステムでデバッグ対象のローダブルモジュールをインストールします。

```
TGT>insmod rd.o ↓
```

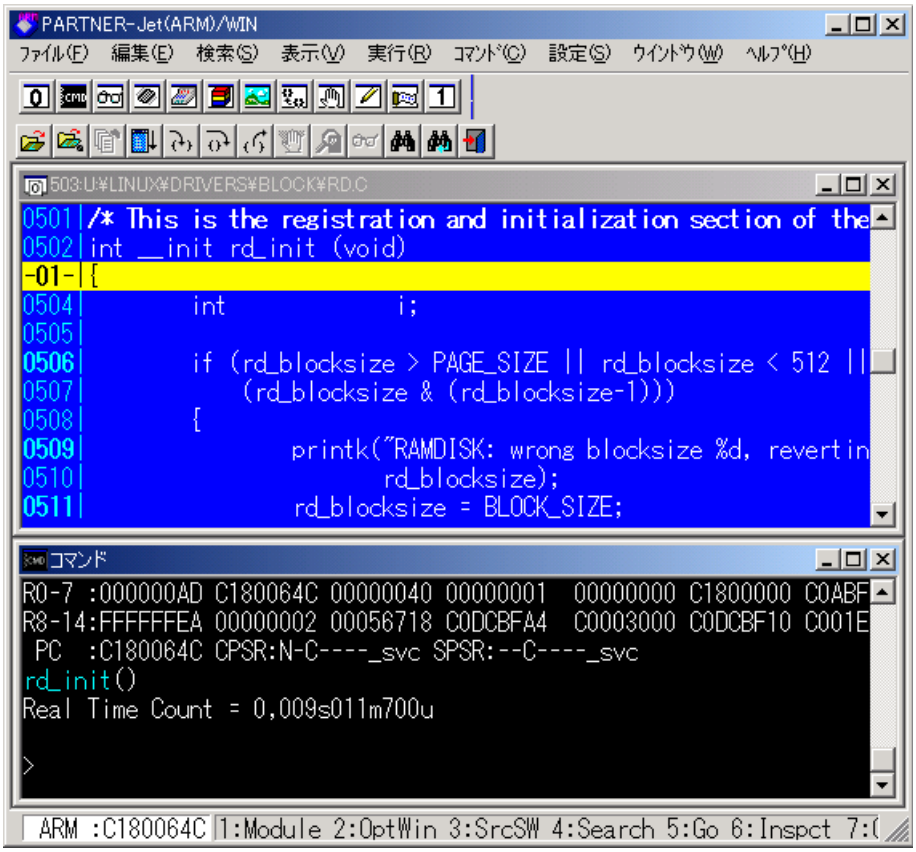
図 A-6 ローダブルモジュールのインストール

```
Parallelizing fsck version 1.22 (22-Jun-2001)
mkdir: cannot create directory `/dev/pts': File exists
Mounting local filesystems...
nothing was mounted
Cleaning: /etc/network/ifstate.
Setting up IP spoofing protection: rp_filter.
Disable TCP/IP Explicit Congestion Notification: done.
Configuring network interfaces: done.
Starting portmap daemon: portmap.
Cleaning: /tmp /var/lock /var/run.
INIT: Entering runlevel: 3
Starting system log daemon: syslogd klogd.
Starting internet superserver: inetd.

MontaVista Linux 3.0, Professional Edition

kzp-arm login: root
Last login: Thu Jan  1 00:00:27 1970 on console
Linux kzp-arm 2.4.18_mvl30-integrator #384 2005年 3月 29日 火曜日 20:06:36 JST a
rmv4l unknown

Welcome to MontaVista Linux 3.0, Professional Edition

root@kzp-arm:~# insmod rd.o
```

A-6 PARTNER のブレーク

ターゲットシステムでデバッグ対象のローダブルモジュールがインストールされると、PARTNER は設定したブレークポイントの位置でブレークします。

図 A-7 ローダブルモジュールでブレーク



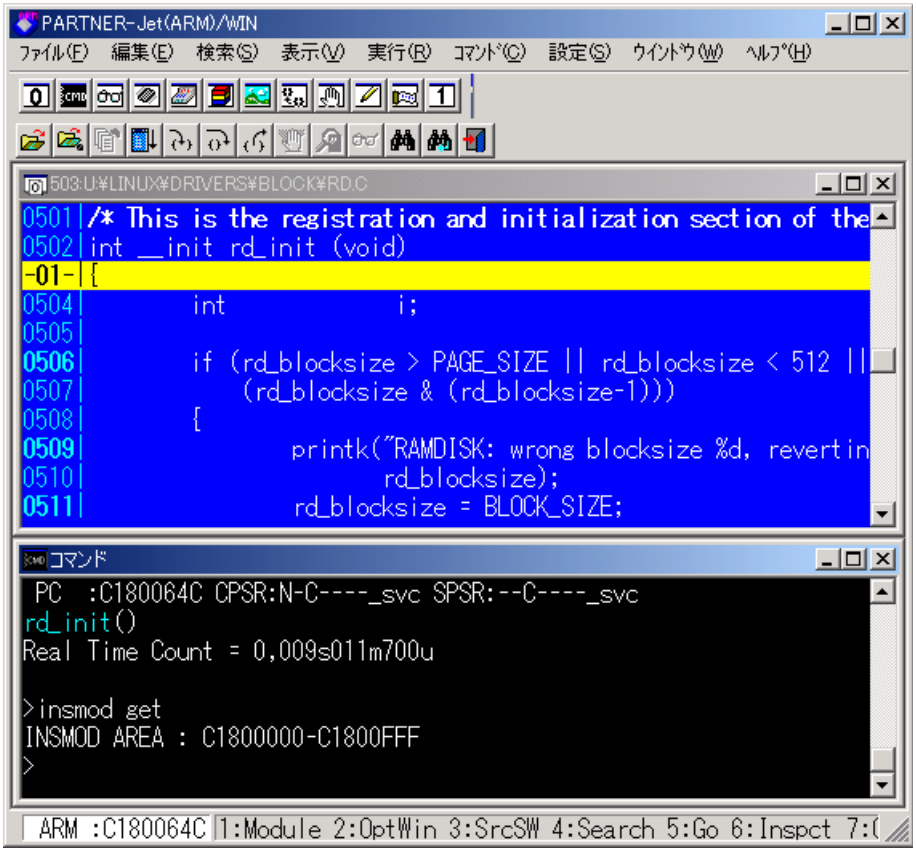
A-7 モジュールのアタッチ

INSMOD コマンド (188 ページ参照) でローダブルモジュール (rd.o) をアタッチします。

```
PT>INSMOD GET ↓
```

コマンドが正常に完了し、ローダブルモジュール (rd.o) の空間が表示されれば、ソースコード上からメモリ参照やソフトウェアブレークの利用ができるようになります。

図 A-8 ローダブルモジュールのアタッチ完了



A-8 モジュールのデタッチ

ターゲットでローダブルモジュールを rmmod **コマンド**でアンロードする前に、PARTNER でアタッチしているローダブルモジュールをデタッチする必要があります。

INSMOD **コマンド** (188 ページ参照) でローダブルモジュールをデタッチします。

```
PT>INSMOD CLRALL ↓
```

コマンドが正常に完了すれば、ターゲットシステムで rmmod **コマンド**を実行できます。

```
TGT>rmmod rd ↓
```

図 A-9 ローダブルモジュールのアンロード

```
Setting up IP spoofing protection: rp_filter.
Disable TCP/IP Explicit Congestion Notification: done.
Configuring network interfaces: done.
Starting portmap daemon: portmap.
Cleaning: /tmp /var/lock /var/run.
INIT: Entering runlevel: 3
Starting system log daemon: syslogd klogd.
Starting internet superserver: inetd.

MontaVista Linux 3.0, Professional Edition

kzp-arm login: root
Last login: Thu Jan  1 00:00:59 1970 on console
Linux kzp-arm 2.4.18_mvl30-integrator #402 2005年 4月 7日 木曜日 13:21:45 JST ar
mv4l unknown

Welcome to MontaVista Linux 3.0, Professional Edition

root@kzp-arm:~# insmod rd.o
RAMDISK driver initialized: 16 RAM disks of 4096K size 1024 blocksize
root@kzp-arm:~# lsmod
Module                  Size  Used by    Not tainted
rd                      2988   0  (unused)
root@kzp-arm:~# rmmod rd
root@kzp-arm:~#
```

rmmod コマンド でアンロード する前に、PARTNER からデタッチしなかった場合には、カーネルパニックが発生します。

付録 B　手動アプリケーションデバッグ

カーネルの修正が出来ない場合や、アプリケーションの修正が出来ない場合ののアプリケーションのデバッグ手順を説明します。

手動アプリケーションデバッグの時は、アプリケーションモードでのデバッグは出来ません。

この節では、サンプル(sample)を使用した場合を例として、アプリケーションのデバッグ方法を次の流れで説明します。

B-1　アプリケーションの作成

デバッグ対象のアプリケーションをデバッグ情報付きで作成します。

B-2　カーネルの実行

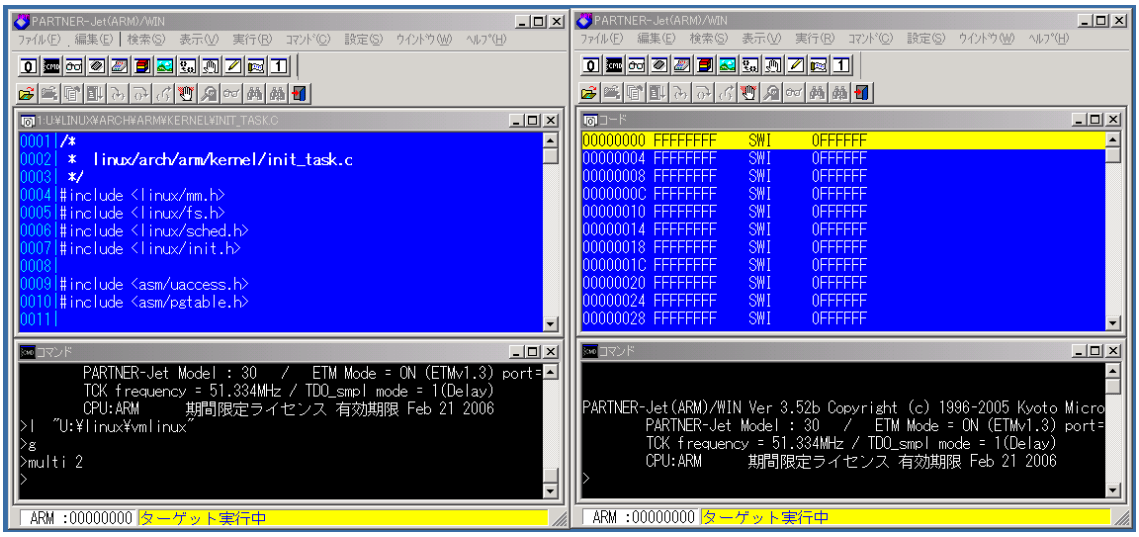
『4.1 カーネルのデバッグ手順（50 頁）』を参照し、カーネルモードで PARTNER からカーネルを実行します。

B-3　アプリケーション用 PARTNER ウインドウの起動

MULTI コマンド（22 ページ参照）で、アプリケーション用 PARTNER ウインドウを起動します。

```
PT>MULTI 2 ↓
```

図 B-1　アプリケーション用ウインドウの起動



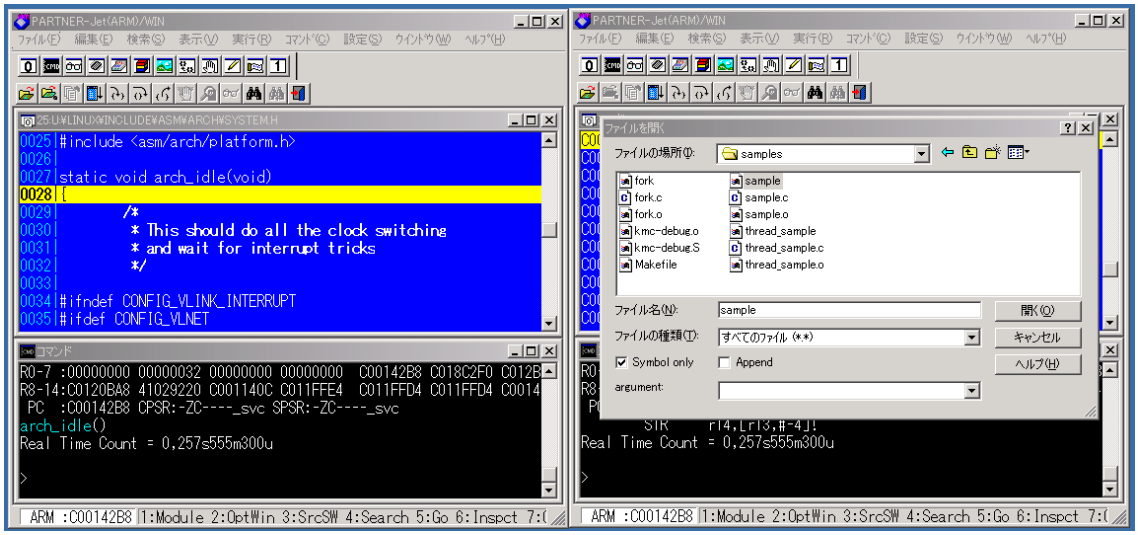
MULTI コマンド（22 ページ参照）や -MULTI オプション（20 ページ参照）で起動した PARTNER ウインドウのウインドウ情報（ウインドウ配置等）やコマンドヒストリは、新規に生成されたプロジェクトファイル（jetsh_?.jpj）に保存され、次回起動時に利用されます。

B-4 アプリケーションのデバッグ情報の読み込み

動作しているカーネルを [ESC] キーで停止し、アプリケーション用 PARTNER ウインドウでアプリケーションのデバッグ情報を PARTNER にロードします。
ロード時には必ず [Symbol only] のチェックを行ってください。

```
PT>ls sample ↓
```

図 B-2 デバッグ情報の読み込み



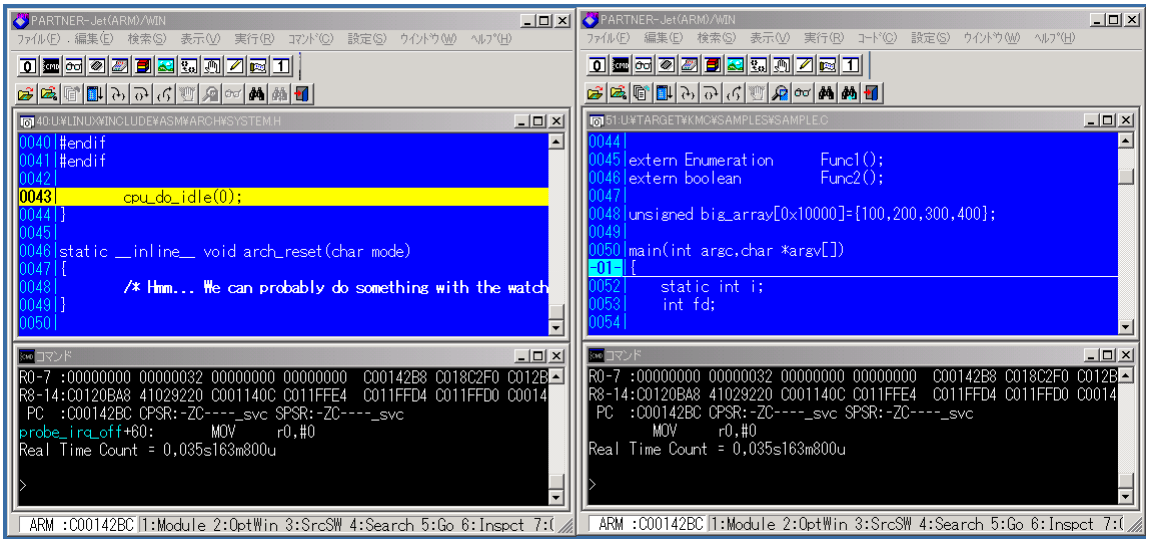
一度ロードされたファイルは、PARTNER がロードファイル名と場所をダイアログとコマンドヒストリに記憶するため、以降の操作を簡略化することができます。

B-5　ブレークポイントの設定

アプリケーションの main() 関数ににブレークポイントを設定します。
ブレークポイントの設定は、1 点だけにしてください。

```
PT>bp main ↓
```

図 B-3 ブレークポイントの設定



ブレークポイント 設定時に『 Verify エラー』メッセージが表示された場合は、既にブレークポイント が設定されています。設定できるブレークポイント は 1 点のみです。ブレークポイント を削除して、アプリケーションのエントリにブレークポイント を設定してください。

B-6 アプリケーションの実行

カーネルを実行し、ターゲットシステムでデバッグ対象のアプリケーションを実行します。

```
TGT>./sample ↓
```

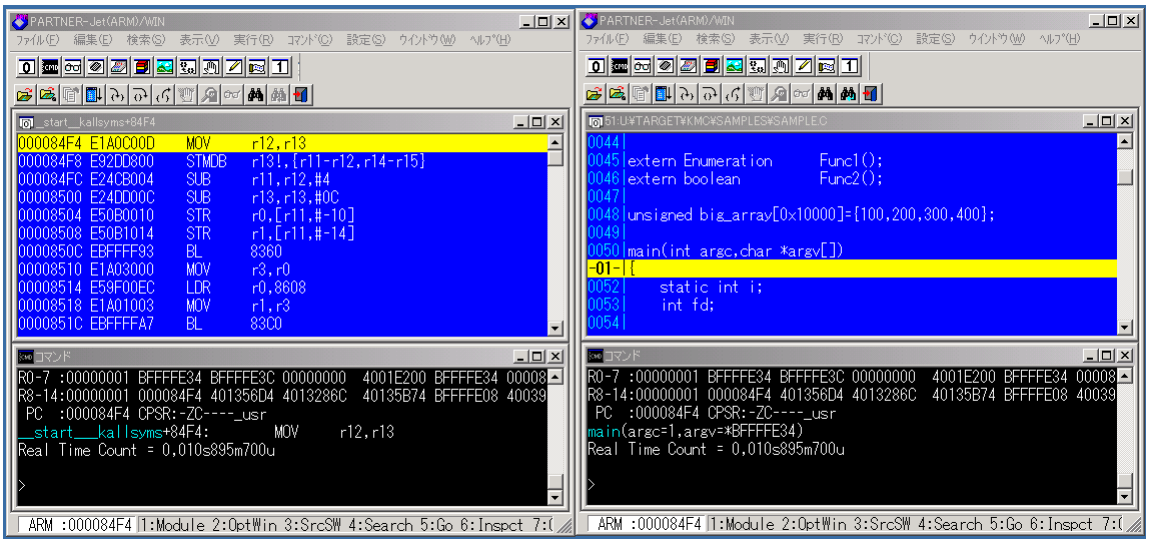
図 B-4 アプリケーションの実行

```
Parallelizing fsck version 1.22 (22-Jun-2001)
mkdir: cannot create directory `/dev/pts': File exists
Mounting local filesystems...
nothing was mounted
Cleaning: /etc/network/ifstate.
Setting up IP spoofing protection: rp_filter.
Disable TCP/IP Explicit Congestion Notification: done.
Configuring network interfaces: done.
Starting portmap daemon: portmap.
Cleaning: /tmp /var/lock /var/run.
INIT: Entering runlevel: 3
Starting system log daemon: syslogd klogd.
Starting internet superserver: inetd.

MontaVista Linux 3.0, Professional Edition

kzp-arm login: root
Last login: Thu Jan  1 00:00:37 1970 on console
Linux kzp-arm 2.4.18_mvl30-integrator #398 2005年 3月 31日 木曜日 15:55:38 JST a
rmv4l unknown

Welcome to MontaVista Linux 3.0, Professional Edition

root@kzp-arm:~# ./sample
```

B-7 PARTNER のブレーク

ターゲットシステムでデバッグ対象のアプリケーションがインストールされると、PARTNER は設定したブレークポイントの位置でブレークします。

図 B-5 アプリケーションのブレーク



PARTNER は、デバッグ情報（シンボル情報）から取得したアドレスでハードウェアブレークを設定します。
アプリケーションの場合、このアドレスが他のプログラムでも使用されている可能性があるため、プログラム実行時に別のプログラムによってブレークが発生する可能性があります。
この場合は、目的のアプリケーションでブレークするまで、プログラムをそのまま続行（[F5] キー）させてください。

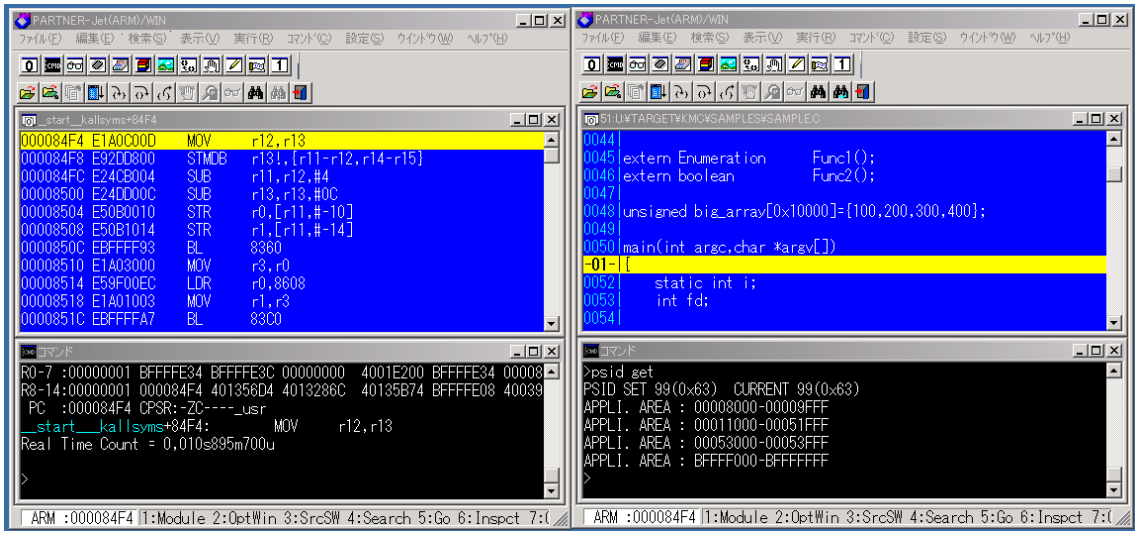
B-8 アプリケーションのアタッチ

PSID コマンド (190 ページ参照) でアプリケーション (sample) をアタッチします。

```
PT>PSID GET ↓
```

コマンドが正常に完了し、アプリケーション (sample) の空間が表示されれば、ソースコード上からメモリ参照やソフトウェアブレークの利用ができるようになります。

図 B-6 アプリケーションのアタッチ



なお、アプリケーション (sample) が終了した場合には、PARTNER 側で、登録した psid 空間が削除されたことを宣言してください。

```
PT>PSID CLRALL ↓
```

付録 C　手動マルチプロセス / マルチスレッド対応のデバッグ方法

カーネルの修正が出来ない場合や、アプリケーションの修正が出来ない場合ののアプリケーションのデバッグ手順を説明します。

手動マルチプロセス / マルチスレッド アプリケーション デバッグの時は、アプリケーション モード でのデバッグは出来ません。
また、ブレークポイント を設定した命令番地を、2 つ以上のスレッド やプロセスが実行したときに正しく デバッガが正しく 扱えないことがあります。

1) アプリケーション 停止中に、停止しているブレークポイント の個所を他のスレッド やプロセスが実行したとき
2) デバッガにアタッチしていないプロセスやスレッド が、ブレークポイント を設定したアドレスを実行したとき

この節では、サンプル (fork) を使用した場合を例として、アプリケーションのデバッグ方法を次の流れで説明します。

C-1 アプリケーションの作成

デバッグ対象のアプリケーションをデバッグ情報付きで作成します。

C-2 カーネルの実行

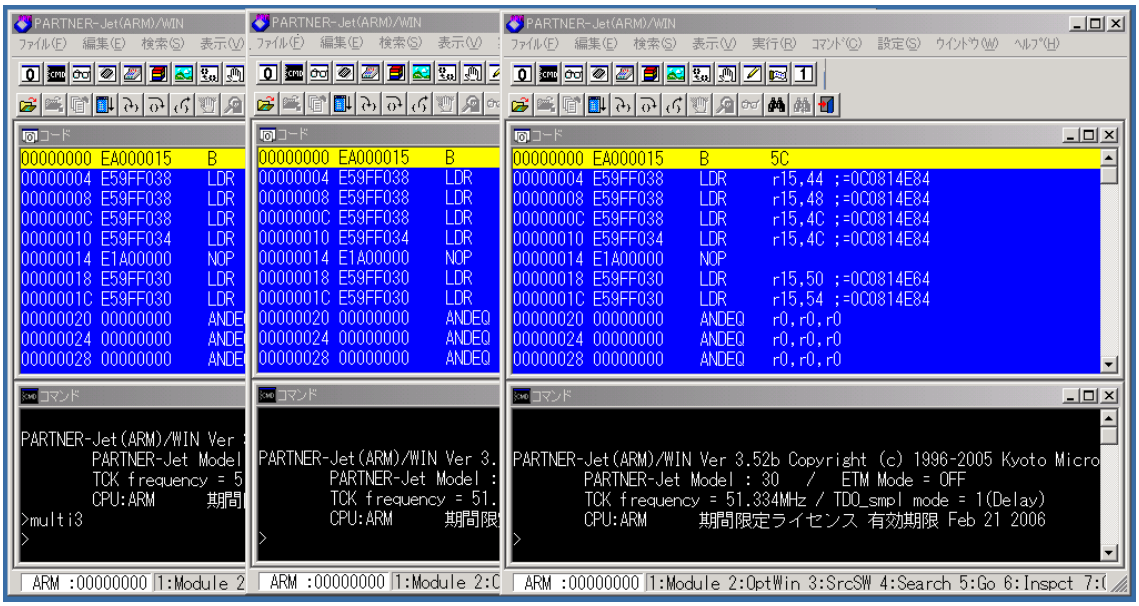
『4.1 カーネルのデバッグ手順（50 頁）』を参照し、カーネルモードで PARTNER からカーネルを実行します。

C-3 アプリケーション用 PARTNER ウインドウの起動

MULTI コマンド（22 ページ参照）で、アプリケーション用 PARTNER ウインドウをカーネル＋プロセス数の数だけ起動します。
サンプル（fork）の場合、3 つの PARTNER ウインドウを起動します。

```
PT>MULTI 3 ↓
```

図 C-1 PARTNER ウインドウの複数起動



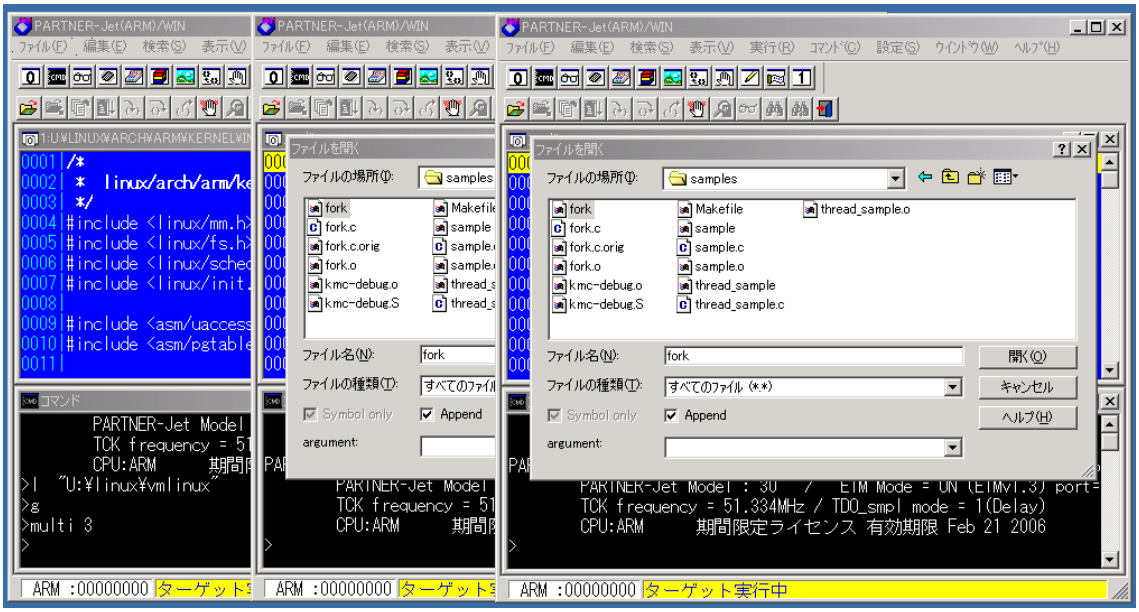
MULTI コマンド（22 ページ参照）や -MULTI オプション（20 ページ参照）で起動した PARTNER ウインドウのウインドウ情報（ウインドウ配置等）やコマンドヒストリは、新規に生成されたプロジェクトファイル（jetsh_?.jpj）に保存され、次回起動時に利用されます。

C-4　アプリケーションのデバッグ情報の読み込み

動作しているカーネルを [ESC] キーで停止し、親プロセス用 PARTNER ウインドウと子プロセス用 PARTNER ウインドウでアプリケーションのデバッグ情報を各 PARTNER ウインドウにロードします。ロード時には必ず [Symbol only] のチェックを行ってください。

PT>ls fork ↓

図 C-2　アプリケーションデバッグ情報のロード



一度ロードされたファイルは、PARTNER がロードファイル名と場所をダイアログとコマンドヒストリに記憶するため、以降の操作を簡略化することができます。

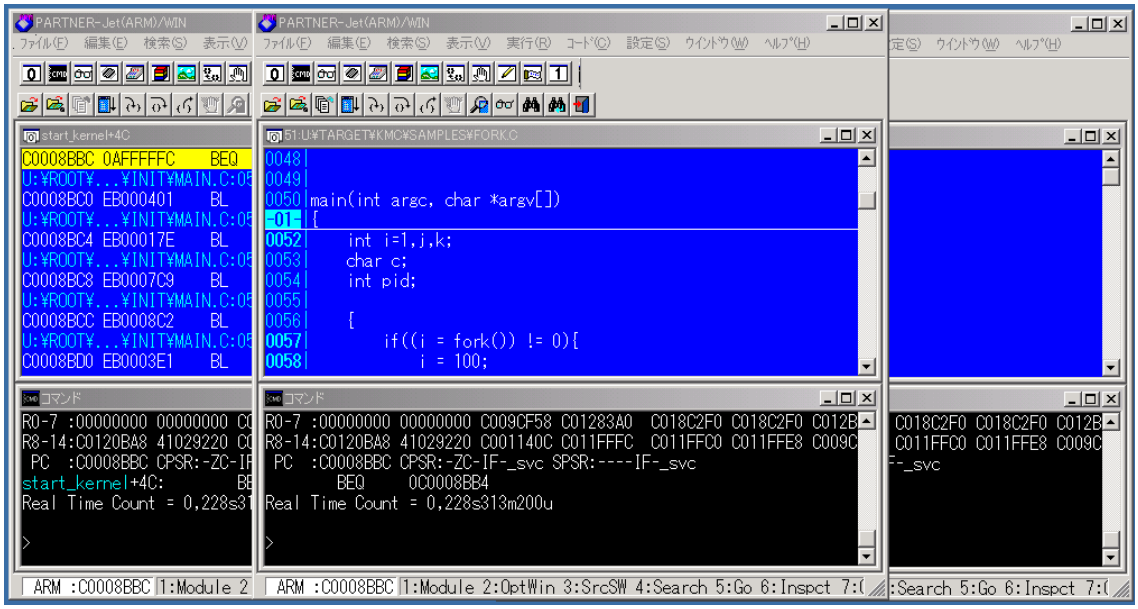
共有ライブラリから生成されるスレッドをアタッチする場合は、共有ライブラリのデバッグ情報もロードする必要があります。共有ライブラリのデバッグ情報のロード方法は、『5.6 共有ライブラリのデバッグ（ 135 頁）』を参照してください。

C-5 親プロセスへのブレークポイントの設定

親プロセスの main() 関数ににブレークポイントを設定します。
ブレークポイントの設定は、1 点だけにしてください。

```
PT>bp main ↓
```

図 C-3 ブレークポイントの設定



ブレークポイント 設定時に『 Verify エラー』メッセージが表示された場合は、既にブレークポイント が設定されています。設定できるブレークポイント は 1 点のみです。ブレークポイント を削除して、アプリケーションのエントリにブレークポイント を設定してください。

C-6　アプリケーションの実行

カーネルを再実行し、ターゲットシステムでデバッグ対象のアプリケーションを実行します。

PT>./fork ↓

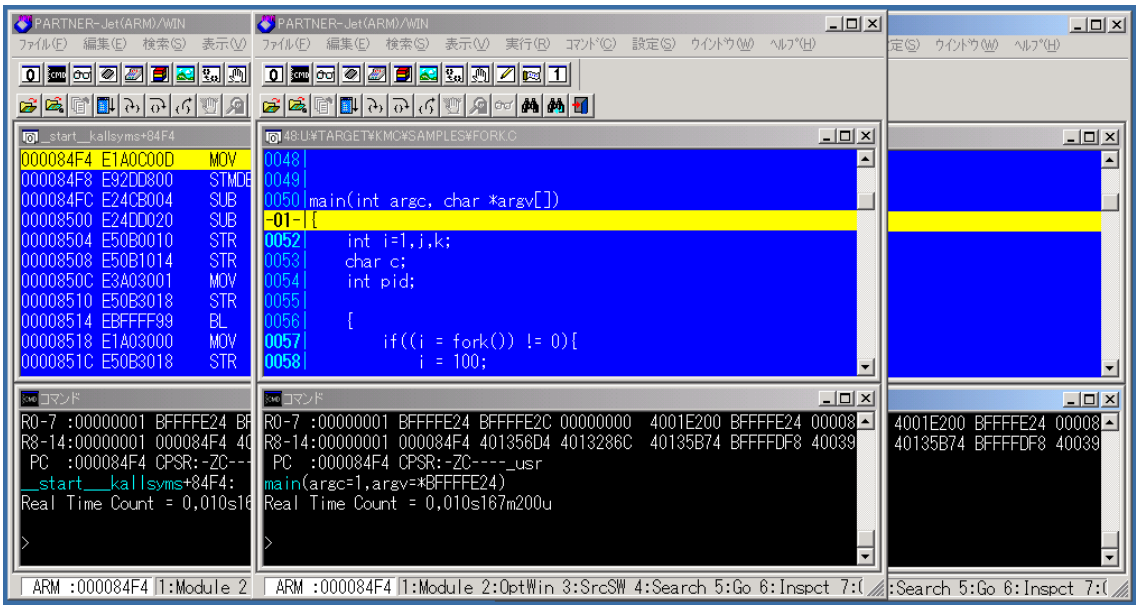
図 C-4 アプリケーションの実行

```
expected (0x802/0x203ba), got (0x802/0x203b9)
chmod: changing permissions of `/dev/ttyp1': Input/output error
nfs_refresh_inode: inode number mismatch
expected (0x802/0x203bc), got (0x802/0x203bb)
chmod: changing permissions of `/dev/ttyp3': Input/output error
nfs_refresh_inode: inode number mismatch
expected (0x802/0x63f60), got (0x802/0x205a6)
INIT: Entering runlevel: 3
Starting system log daemon: syslogd klogd.
Starting internet superserver: inetd.
nfs_refresh_inode: inode number mismatch
expected (0x802/0x205a6), got (0x802/0x645a3)

MontaVista Linux 3.0, Professional Edition

kzp-arm login: root
Last login: Thu Jan  1 00:00:20 1970 on console
Linux kzp-arm 2.4.18_mvl30-integrator #390 2005年 3月 31日 木曜日 10:41:07 JST a
rmv4l unknown

Welcome to MontaVista Linux 3.0, Professional Edition

root@kzp-arm:~# cd /KMC/samples/
root@kzp-arm:/KMC/samples# ./fork
```

C-7 PARTNER のブレーク

ターゲットシステムでデバッグ対象のアプリケーションがインストールされると、PARTNER は設定したブレークポイントの位置でブレークします。

図 C-5 アプリケーションのブレーク



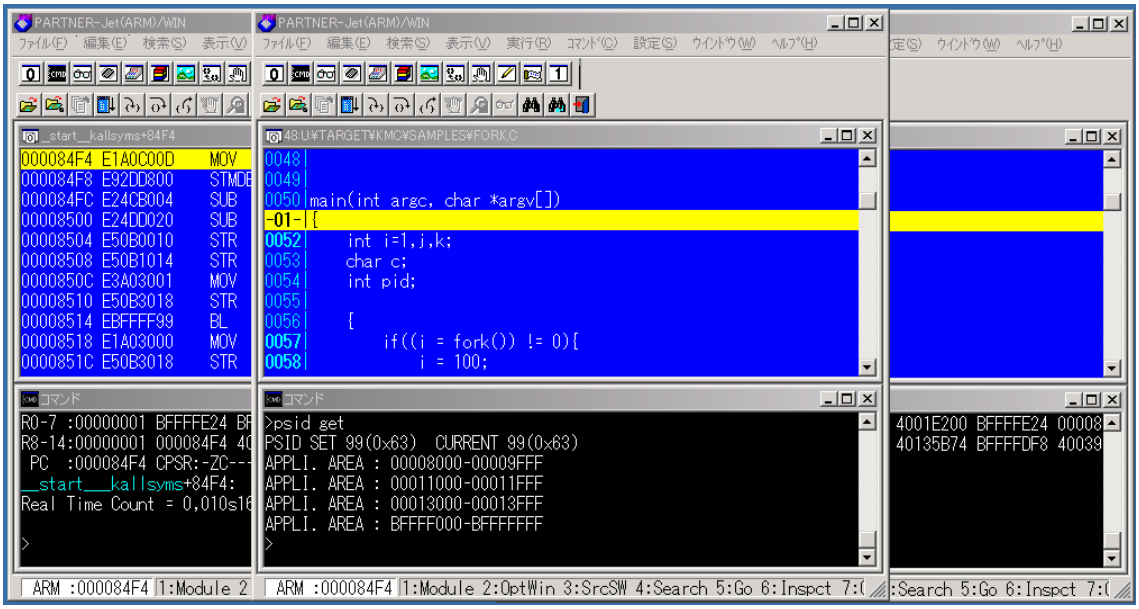
PARTNER は、デバッグ情報（シンボル情報）から取得したアドレスでハードウェアブレークを設定します。
アプリケーションの場合、このアドレスが他のプログラムでも使用されている可能性があるため、プログラム実行時に別のプログラムによってブレークが発生する可能性があります。
この場合は、目的のアプリケーションでブレークするまで、プログラムをそのまま続行（[F5] キー）させてください。

C-8　親プロセスのアタッチ

親プロセス用 PARTNER ウインドウで **PSID コマンド** (190 ページ参照) を実行し、親プロセスをアタッチします。

```
PT>PSID GET ↓
```

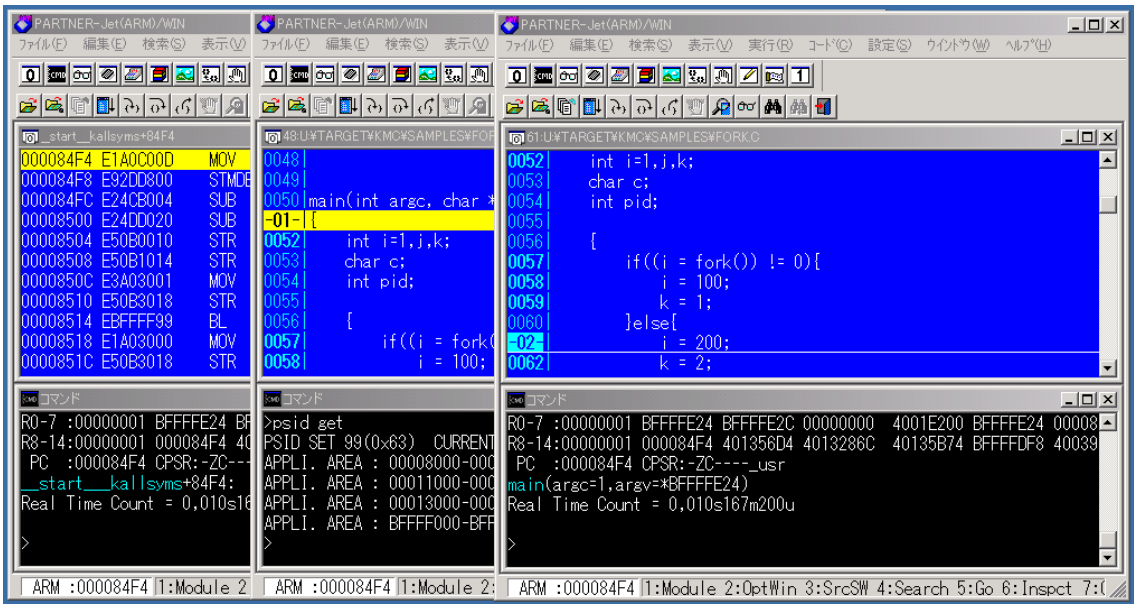
図 C-6 親プロセスのアタッチ



C-9 子プロセスへのブレークポイントの設定

子プロセス用 PARTNER ウインドウで、fork() システムコール呼び出しの子プロセス側の戻り番地にブレークポイントを設定します。マルチスレッドの場合は、スレッドの先頭にブレークポイントを設定します。ブレークポイントの設定は、1 点だけにしてください。

図 C-7 子プロセスへのブレークポイント



ブレークポイント 設定時に『 Verify エラー』メッセージが表示された場合は、既にブレークポイント が設定されています (PSID GET して PARTNER にアタッチしている空間に設定されているブレークポイント は含みません)。設定できるブレークポイント は 1 点のみです。ブレークポイント を削除して、アプリケーション のエントリにブレークポイント を設定してく ださい。

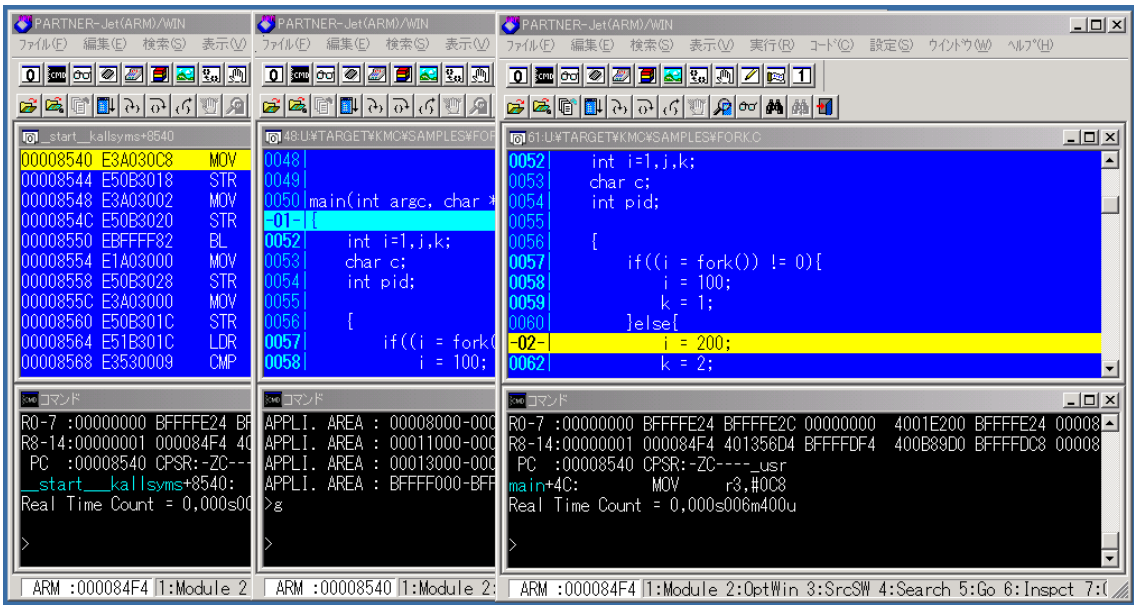
C-10 アプリケーションの再実行

親プロセス用 PARTNER ウインドウでアプリケーションを再開します。

C-11　PARTNER のブレーク

子プロセスが生成されると、PARTNER は設定したブレークポイントの位置でブレークします。

図 C-8　子プロセスのブレーク



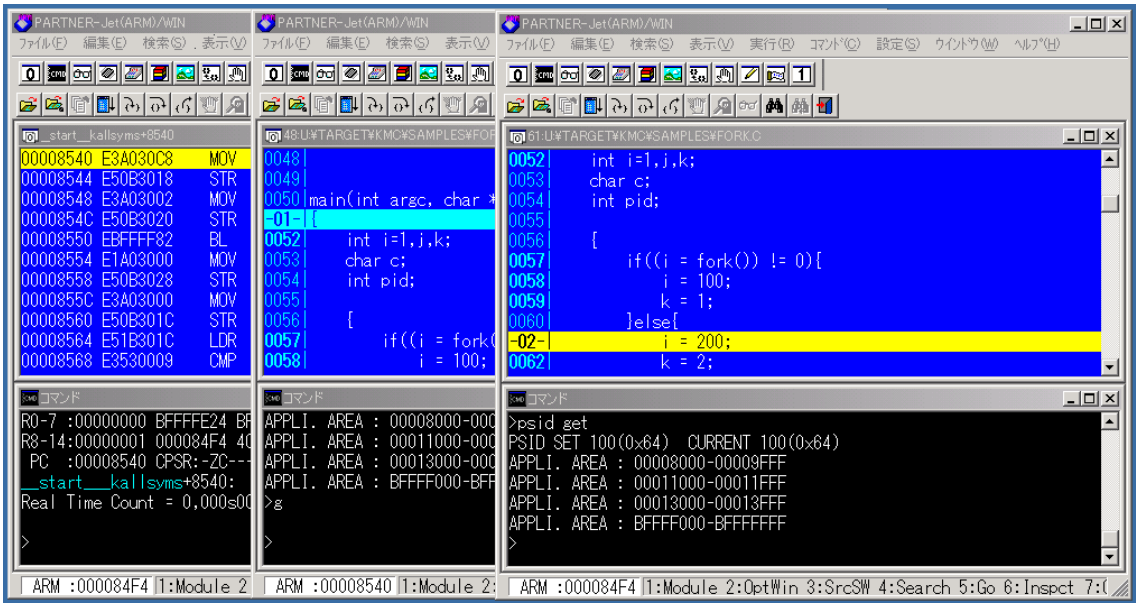
PARTNER は、デバッグ情報（シンボル情報）から取得したアドレスでハードウェアブレークを設定します。
アプリケーションの場合、このアドレスが他のプログラムでも使用されている可能性があるため、プログラム実行時に別のプログラムによってブレークが発生する可能性があります。
この場合は、目的のアプリケーションでブレークするまで、プログラムをそのまま続行（[F5] キー）させてください。

C-12 子プロセスのアタッチ

子プロセス用 PARTNER ウインドウで **PSID コマンド** (190 ページ参照) を実行し、子プロセスをアタッチします。

```
PT>PSID GET ↓
```

図 C-9 子プロセスのアタッチ



これで、fork() システムコールを使用した親プロセスと子プロセスを個々に制御してデバッグを行うことが可能です。子プロセスが複数になった場合には、繰り返しを行うことでデバッグが可能です。
マルチプロセス対応を実行した (別個の PARTNER で、それぞれ **psid get コマンド**を発行した状態) のちは、次のような動作となります。

1. プロセス A にブレークポイントを設定し、プロセス A で停止する。
2. ここで、プロセス B のウィンドウで **g コマンド**によりプログラムを実行する。
3. プロセス A は“アプリケーション停止中”となり、擬似的に実行が抑制される。

つまり、停止したウィンドウとは別のウィンドウで **g コマンド**を発行することにより、そのプロセスの実行を抑制することができます。また、別のウィンドウで、**g コマンド**の変わりに **g/a コマンド**を発行することにより、そのプロセスを抑制しないですべてのプロセスを実行することができます。

---

fork() システムコールによるマルチプロセスやマルチスレッド のデバッグには、次の注意点があります。

- 複数のプロセスで同じ 番地にブレークポイント を設定しない。
  (あるタイミングで正常な処理ができない可能性があります。)
- 関数の戻り 番地にブレークポイント を設定しない。
  (停止中のプロセスと 別のプロセスが関数から 戻ってきた場合、処理が継続できない可能性があります。)

付録 D　動的なリンカローダ (ld.so) のデバッグ手順

動的なリンカローダ (ld.so) のデバッグは、『5.6 共有ライブラリのデバッグ (135 頁)』で説明した共有ライブラリのデバッグと違って PARTNER でのアドレスの自動解決などが行えません。したがって、手動で PARTNER にアタッチする必要があります。

この節では、ld.so/sample をデバッグする場合を例として、リンカローダのデバッグ方法を次の流れで説明します。

D-1　リンカローダ (ld.so) にデバッグ情報

デバッグ対象となるリンカローダにデバッグ情報を付加します。
デバッグ情報はカーネルやアプリケーションと同じフォーマットを選択してください。PARTNER で正しくデバッグ情報を読み込めない場合は、他のフォーマットを試してみてください。

D-2　アプリケーションの作成

デバッグ対象となるアプリケーションを作成します。リンカローダと一緒にアプリケーションをデバッグする場合には、アプリケーション内にデバッグスタブ (__kmc_start_debuger()) を記述しないでください。

D-3 リンカローダ内のシンボル _dl_start のアドレス確認

『D-1 リンカローダ (ld.so) にデバッグ情報 (179 頁)』で作成したリンカローダ内のシンボル _dl_start のアドレスを確認します。

```
LINUX86>nm ld-2.2.5.so | grep _dl_start ↓
```

図 D-1 シンボル _dl_start のアドレス確認

```
[root@kmvl3 lib]# nm ld-2.2.5.so | grep _dl_start
00001eac t _dl_start
00002500 t _dl_start_final
0000daf0 T _dl_start_profile
00001d18 t _dl_start_user
0001e460 B _dl_starting_up
[root@kmvl3 lib]#
```

D-4 カーネルの実行

『5.2 カーネルモードでのアプリケーションデバッグの手順 (73 頁)』の『5.2.7 Linux カーネルの実行 (81 頁)』までを参照して、アプリケーション用 PARTNER ウインドウが起動していて、カーネルが実行中になっている状態にします。

D-5 リンカローダの .text 開始位置の確認

PARTNER でアプリケーションのメモリマップを確認し、リンカローダの .text 開始位置を確認します。

```
PT>maps 1 ↓
00400000-00408000 r-xp /sbin/init
00417000-00418000 rw-p /sbin/init
00418000-0041c000 rwxp
29556000-2956b000 r-xp /lib/ld-2.2.5.so
2957a000-2957b000 rw-p /lib/ld-2.2.5.so
2957b000-2969d000 r-xp /lib/libc-2.2.5.so
2969d000-296ab000 ---p /lib/libc-2.2.5.so
296ab000-296b1000 rw-p /lib/libc-2.2.5.so
296b1000-296b5000 rw-p
7bfff000-7c000000 rwxp
```

D-6　カーネルの強制ブレーク

PARTNER ウインドウで ESC キーを押し、カーネルを強制ブレークさせます。

D-7　ハードウェアブレークの設定

ハードウェアブレークポイントをリンカローダの _dl_start に設定します。_dl_start のアドレスは、『D-3 リンカローダ内のシンボル _dl_start のアドレス確認（180 頁）』で確認したオフセット値と『D-5 リンカローダの .text 開始位置の確認（180 頁）』の .text 開始位置を足した値です。

```
PT>br 0x1eac + 0x29556000.ex ↓
```

ハードウェアブレークを _dl_start に設定すると、すべてのプロセスでブレークが発生するので注意が必要です。

D-8　デバッグしたいアプリケーションの実行

カーネルを再実行し、ターゲットシステムでデバッグ対象のアプリケーションを実行します。

```
PT>./sample ↓
```

D-9　リンカローダのデバッグ情報読み込み

PARTNER がブレークしたところで、アプリケーション用 PARTNER ウインドウのコマンドウインドウでデバッグ対象のリンカローダのデバッグ情報をロードします。

```
PT>ls ld-2.2.5.so./r .text=0x29556000 ↓
```

リンカローダのデバッグ情報をロードするときには、リロケーションアドレスを必ず指定する必要があります。

D-10　ハードウェアブレークの解除

『D-7 ハードウェアブレークの設定（181 頁）』で設定したハードウェアブレークポイントを解除します。

```
PT>brc * ↓
```

D-11 プロセスのアタッチ

PSID コマンド(190 ページ)で現在のアプリケーションプロセスをアタッチします。

```
PT>psid get ↓
PSID SET 118(0x76)  CURRENT 118(0x76)
APPLI. AREA : 00400000-00401FFF
APPLI. AREA : 00411000-00451FFF
APPLI. AREA : 00453000-00453FFF
APPLI. AREA : 7BFFF000-7BFFFFFF
```

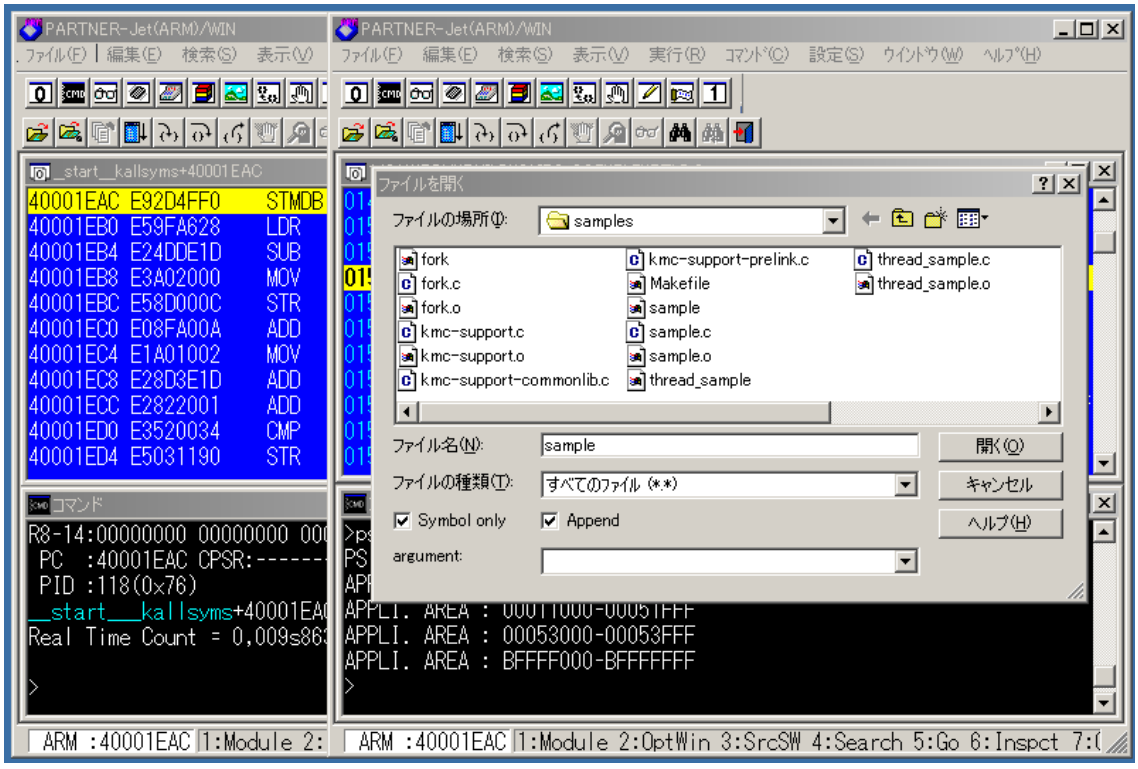
D-12 アプリケーション / 共有ライブラリのデバッグ情報追加読み込み

アプリケーション用 PARTNER ウインドウでデバッグ対象アプリケーションや共有ライブラリのデバッグ情報を追加読み込みを行います。これで、リンカローダ、アプリケーション、共有ライブラリが一緒にデバッグ可能になります。

ロード時には必ず [Symbol only] と [Append] のチェックを行ってください。

```
PT>lsa sample ↓
```

図 D-2 デバッグ情報の追加読み込み



付録 E　共有ライブラリ内で生成されるスレッドのデバッグ方法

『5.3 カーネルモードでのマルチプロセス / マルチスレッドアプリケーションデバッグ手順（85 頁）』とほぼ同じ手順でデバッグを行いますが、『5.3.3 アプリケーションの作成（87 頁）』で記述しているアプリケーションの作成方法が変わります。

E-1　サポートファイルの共有ライブラリ化

『5.3.3 アプリケーションの作成（87 頁）』では、アプリケーションに直接リンクしていたサポートファイル (kmc-support.c) をサポート共有ライブラリとして作成します。
共有ライブラリの作成は、サンプルに入っている Makefile で行えます。

```
LINUX86>make libkmcsup.so.1.0.0 ↓
```

サポート ファイル (kmc-supoort.c) をスレッド を生成する共有ライブラリにリンクすることでも対応可能です。

kmc-support.c 内で "Select Target CPU type" のコンパイルエラーが発生した場合は、kmc-support.c 内の CPUTYPE シンボル宣言をターゲット CPU のみ有効にして再コンパイルしてください。

E-2　アプリケーションの作成

アプリケーションソースの main() 関数の先頭にデバッグスタブの呼び出しを挿入します。また、デバッグスタブを挿入したソースにデバッグスタブのプロトタイプ宣言 (kmc-support.h) を挿入します。

```
__kmc_start(char *program_name);
```

main() 関数直後に挿入するデバッグスタブ関数の引数 char *program_name には、アプリケーションのファイル名が入るようにしてください。
ただし、シンボリックリンクでファイル名が変えられている場合などは、実際のファイル名を文字列で埋め込んでください。

【例】

```
+#include "kmc-support.h"
                    :
                    :
```

```
 int main(int argc,char *argv[])
 {
+    __kmc_start(argv[0]);
                          :
                          :
```

E-3 スレッドを生成する共有ライブラリの作成

共有ライブラリでは、スレッドボディ関数の先頭ににデバッグスタブを挿入します。また、デバッグスタブを挿入したソースにデバッグスタブのプロトタイプ宣言(kmc-support.h)を挿入します。

```
__kmc_start(char *program_name);
```

スレッドボディ関数直後に挿入するデバッグスタブ関数の引数 char *program_name には、0 を指定してください。

【例】

```
+#include "kmc-support.h"
                          :
                          :
 void thread_func(void *)
 {
+    __kmc_start(0);
                          :
                          :
```

共有ライブラリリンク時に『E-1 サポートファイルの共有ライブラリ化 (183 頁)』で作成したサポート共有ライブラリを -l オプションで指定します。

```
LINUX86>$(CC) -shared ..............-lkmcsup ↓
```

サポート ファイル(kmc-supoort.c)をスレッドを生成する共有ライブラリにリンクしている場合は、-l オプションで指定する必要はありません。

E-4 共有ライブラリのデバッグ情報の読み込み

『5.3 カーネルモードでのマルチプロセス / マルチスレッドアプリケーションデバッグ手順 (85 頁)』の手順どおり作業を行って、main() 関数に挿入したデバッグスタブでブレークしたところで、共有ライブラリのデバッグ情報を読み込みます。

```
PT>lsa lib-shared.so ↓
```

『5.3.10 PARTNER のブレーク（99 頁）』を参照してください。

付録 F　手動デバッグ時コマンド

PARTNER で手動デバッグを行うために、コマンドが拡張されています。
手動デバッグ用コマンドは、『付録 A　手動モジュールデバッグ（152 頁）』、『付録 B　手動アプリケーションデバッグ（162 頁）』または『付録 C　手動マルチプロセス / マルチスレッド対応のデバッグ方法（169 頁）』の時に使用してください。

INSMOD(ローダブルモジュールのアタッチ制御)

書式 1

INSMOD

書式 2

INSMOD GET

書式 3

INSMOD CLR

書式 4

INSMOD CLRBP

書式 5

INSMOD CLRALL

機能

ローダブルモジュールのアタッチ制御

解説

Linux 上の insmod **コマンド**でインストールしたローダブルモジュールのアタッチ制御を行います。
このコマンドは、デバッグ対象のローダブルモジュールのデバッグ情報を読み込んでおく必要があります。
書式 1 は、現在 PARTNER にアタッチされているローダブルモジュールのメモリ範囲を表示します。
書式 2 は､Linux 上の insmod **コマンド**でインストールしたローダブルモジュールをデバッガにアタッチし、ローダブルモジュール空間内のメモリのアクセス、ブレークポイントの設定が可能になります。
書式 3 は、書式 2 でアタッチしたローダブルモジュールをデタッチします。
ただし、書式 2 でアタッチしたローダブルモジュール空間内のブレークポイントは保持されます。
書式 4 は、書式 2 でアタッチしたローダブルモジュール空間内のブレークポイントを削除します。
書式 5 は、完全なクリアです。書式 2 でアタッチしたローダブルモジュールをデタッチし、ブレークポイントを削除します。

【使用例】

```
>ls u:¥linux¥drivers¥block¥rd.o ↓
 0xC1800060-0xC18007DB .text
 0xC18007DC-0xC180089B .rodata.str1.4
 0xC1800970-0xC18009B3 .data
 0xC1800A68-0xC1800BAB .bss
>bp rd_init ↓
>g ↓
/* Linux 上でローダブルモジュールをインストール #insmod rd.o */
/* PARTNER がローダブルモジュールの初期化エントリ（rd_init 関数）でブレーク */
        R0/R8    R1/R9    R2/R10   R3/R11    R4/R12   R5/R13   R6/R14   R7
R0-7 :000000AB C180064C 00000040 00000001  00000000 C1800000 C0ABB000 00000060
R8-14:FFFFFFEA 00000002 00056718 C0BA7FA4  C0003000 C0BA7F10 C001E118
 PC  :C180064C CPSR:N-C----_svc SPSR:--C----_svc
rd_init()
Real Time Count = 0,010s446m100u
>insmod get ↓
INSMOD AREA : C1800000-C1800FFF
>
```

PSID(プロセスのアタッチ制御)

書式 1

PSID

書式 2

PSID GET

書式 3

PSID CLR

書式 4

PSID CLRBP

書式 5

PSID CLRALL

機能

アプリケーションのアタッチ制御

解説

書式 1 は、カレント PARTNER にアタッチされているアプリケーションのプロセス ID と使用メモリ範囲を表示します。

書式 2 は、カレント PARTNER が読み込んでいるデバッグ情報のプロセスを PARTNER にアタッチし、プロセス空間内のメモリのアクセス、ブレークポイントの設定が可能になります。

書式 2 を実行するタイミングとしては、アプリケーションのスタート (main() 関数直後)、プロセス / スレッドの生成直後のエントリにブレークポイントを設定した後、デバッグ対象アプリケーションを実行し、設定したブレークポイントでブレークした際が一般的です。

書式 3 は、書式 2 でアタッチしたプロセスをデタッチします。

ただし、書式 2 でアタッチしたプロセス空間内のブレークポイントは保持されます。

書式 4 は、書式 2 でアタッチしたプロセス空間内のブレークポイントを削除します。

書式 5 は、完全なクリアです。書式 2 でアタッチしたプロセスをデタッチし、ブレークポイントを削除します。

【使用例】

```
>ls u:¥appli¥sample ↓
>bp main ↓
>g ↓
/* Linux 上でアプリケーション (sample) を実行 */
/* PARTNER がアプリケーションの main でブレーク */
        R0/R8    R1/R9    R2/R10   R3/R11    R4/R12   R5/R13   R6/R14   R7
R0-7 :00000001 BFFFFE24 BFFFFE2C 00000000  4001E200 BFFFFE24 0000833C 4000C728
R8-14:00000001 000084F4 401356D4 4013286C  40135B74 BFFFFDF8 40039328
 PC  :000084F4 CPSR:-ZC----_usr
main(argc=1,argv=*BFFFFE24)
>
>psid get ↓
PSID SET 99(0x63)  CURRENT 99(0x63)
APPLI. AREA : 00008000-00009FFF
APPLI. AREA : 00011000-00011FFF
APPLI. AREA : 00013000-00013FFF
APPLI. AREA : BFFFF000-BFFFFFFF
>
```

付録 G　トラブルシューティング

G-1　カーネルデバッグ

ソースコードが表示されない

カーネルコンパイル時のコンパイルオプションにデバッグ情報付加のオプションがついていることを確認してください。『4.1.1 Linux カーネルのコンフィグレーション（51 頁）』参照
起動オプション -XGX 指定で PATH 変換が正しく行われているか確認してください。コードウインドウのキャプションバーに表示されているソースファイル PATH が PARTNER がオープンしようとしているファイル名です。『 SNAME(ソースパス表示) (32 頁)』,『 -XGX オプション（19 頁）』参照
LINUX と Windows のファイル共有設定 (Samba 設定など) を確認してください。

ブレークできない

カーネルがフラッシュメモリから RAM へ転送されるシステムの場合、start_kernel() 関数まで実行されていない場合はソフトウェアブレークを設定してもでブレークできません。ハードウェアブレークを使用してください。

```
PT>br start_kernel.ex ↓
PT>brc* ↓
PT>bp sys_write ↓
```

G-2　ローダブルモジュールデバッグ

PARTNER に自動アタッチできない

モジュールソースの修正が正しく出来ているか確認してください。『4.2.3 ローダブルモジュールソースの修正（60 頁）』参照
起動オプションで -OS LINUX が設定されているか確認してください。『4.2.5 カーネルモードでの PARTNER の起動（61 頁）』,『 -OS オプション（18 頁）』参照

デバッグ情報がロードされない

ローダブルモジュールを作成したディレクトリにローダブルモジュールオブジェクトが存在するか確認してください。また、__KMC_MODULE_NAME でローダブルモジュールの PATH を指定した場合、PATH にローダブルモジュールオブジェクトが存在するか確認してください。『4.2.3 ローダブルモジュールソースの修正（60 頁）』参照

ソース表示できない

ローダブルモジュールのコンパイル時にデバッグ情報付加のオプションがついていることを確認してください。
起動オプション -XGX,-SK 指定で PATH 変換が正しく行われているか確認してください。コードウインドウのキャプションバーに表示されているソースファイル PATH が PARTNER がオープンしようとしているファイル名です。『 SNAME(ソースパス表示) (32 頁)』,『 -XGX オプション (19 頁)』,『 -SK オプション (20 頁)』参照
LINUX と Windows のファイル共有設定 (Samba 設定など) を確認してください。

G-3 アプリケーションデバッグ

アプリケーションが自動アタッチできない

アプリケーションを実行すると PARTNER がハングアップする場合、CFG ファイルの **MAP フィールド** の指定が間違っている可能性があります。**MAP フィールド** を確認してください。『 MAP フィールド (15 頁)』参照
PARTNER にアプリケーションを実行している Linux システムのカーネルのデバッグ情報を少なくと一度はロードしましたか。していない場合は、現在デバッグしているカーネルのデバッグ情報をロードしてください。
アプリケーションのソースにデバッグスタブ関数 (`__kmc_start_debuger()` もしくは `__kmc_start()`) が挿入されているか、デバッグスタブ関数の引数に正しいアプリケーション名が入っているか確認してください。『5.2.3 アプリケーションの作成 (75 頁)』,『5.4.3 アプリケーションの作成 (106 頁)』参照
起動オプションで **-OS LINUX** が設定されているか確認してください。『 -OS オプション (18 頁)』参照

マルチスレッドアプリケーション (pthread) のデバッグができない

アプリケーションのソースの `main()` 関数の先頭 / スレッドボディの先頭にデバッグスタブ関数 (`__kmc_start_debuger()` もしくは `__kmc_start()`) が挿入されているか、デバッグスタブ関数の引数に正しいアプリケーション名が入っているか確認してください。『5.2.3 アプリケーションの作成 (75 頁)』,『5.4.3 アプリケーションの作成 (106 頁)』参照
起動オプションで **-OS LINUX** が設定されているか確認してください。『 -OS オプション (18 頁)』参照
MULTI コマンドでアプリケーション用の PARTNER ウインドウを立ち上げて、そのアプリケーション用の PARTNERウインドウでアプリケーションのデバッグ情報をロードしていますか。『 MULTI(複数PARTNER ウインドウの起動) (22 頁)』参照
1 つのアプリケーション用 PARTNER ウインドウでデバッグする場合、ADD モードになっているか確認してください。『 PSID(プロセスアタッチ情報) (25 頁)』参照
NON_ADD モードの場合は、各スレッド毎に PARTNER ウインドウを立ち上げて、デバッグ情報をロードしてください。

マルチプロセスアプリケーション(fork)のデバッグができない

アプリケーションのソースの main() 関数の先頭 / 子プロセス側の先頭にデバッグスタブ関数(__kmc_start_debuger() もしくは __kmc_start()) が挿入されているか、デバッグスタブ関数の引数に正しいアプリケーション名が入っているか確認してください。『5.2.3 アプリケーションの作成 (75 頁)』, 『5.4.3 アプリケーションの作成 (106 頁)』参照

起動オプションで -OS LINUX が設定されているか確認してください。『 -OS オプション (18 頁)』参照

MULTI コマンドでアプリケーション用の PARTNER ウインドウを立ち上げて、そのアプリケーション用の PARTNERウインドウでアプリケーションのデバッグ情報をロードしていますか。『 MULTI(複数PARTNER ウインドウの起動) (22 頁)』参照

実行中のアプリケーションのアタッチができない

アプリケーションが使用している glibc にサポートファイル (kmc-support.c) がリンクされているか確認してください。『5.8.1 ターゲットシステムの glibc の修正 (144 頁)』参照

アプリケーションが使用しているglibcが変更された場合は、PARTNERに登録する必要があります。『5.8.2 PARTNER への glibc の登録 (145 頁)』参照

PARTNERにアタッチしたいアプリケーションが実行中の場合にのみATTACHコマンドは有効になります。PS コマンドで確認してください。『 ATTACH(プロセスのアタッチ) (29 頁)』, 『 PS(プロセス情報) (27 頁)』参照

G-4 リアルタイムトレース

トレース表示できない

使用している PARTNER-Jet がトレースサポートモデルか確認してください。

ターゲットと接続している JTAG プローブがトレースサポートプローブか確認してください。

ターゲット CPU がトレースをサポートしている CPU か確認してください。CPU によっては、トレース Enable をデバッガでセットしないと出来ないものがあります。注意してください。

アプリケーションとカーネルのトレース分離ができない

カーネル用 PARTNER ウインドウ、アプリケーション用 PARTNER ウインドウそれぞれにカーネル、アプリケーションのデバッグ情報を読み込んでいるか確認してください。

起動オプションで -OS LINUX が設定されているか確認してください。『 -OS オプション (18 頁)』参照

Linux カーネルの変更が正しく行われているか確認してください。『3.2 カーネルソース修正 (35 頁)』参照

アプリケーションのトレース表示が正しく表示できない

アプリケーションのデバッグ情報を読み込んでいるか確認してください。

共有ライブラリ実行時のトレース表示は、共有ライブラリのデバッグ情報を読み込んでおく必要があります。『5.7 Linux OS 対応ヒストリ表示 (137 頁)』参照

索引

か

き

さ

し

す

そ

て

と

は

ひ

ふ

ま

ろ