Embeded Linux
Embeded Linux
Embeded Linux

PARTNER for Linux

AM33/ARM/MIPS/SH シリーズ共通

京都マイクロコンピュータ株式会社

PARTNER for Linux (AM33/ARM/MIPS/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

本書は、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、追加機能についての PARTNER Ver.5.11 以降を対象とした AM33/ARM/MIPS/SH シリーズ共通マニュアルです。

Ver.3.5 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。また、Linux カーネル 2.6 系のデバッグ機能は Ver5.1 以降でのみサポートされています。旧バージョンのソフトを既にインストールしている場合や、本書の記述の全ての機能を使用したい場合は、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。

なお、本文中のスクリーンショットには ARM シリーズ用のソフトウェアを用いています。

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**の選択で行ってください。

製品構成と名称について

● **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。

● **PARTNER(高機能ソースレベルデバッガソフトウェア)**

Linux 対応 PARTNER 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。

本書内の表記について

本書では数種類の表記上の約束を採用しています。

引用・相互参照

本書内の別の箇所での記述を参照する場合や引用は二重鍵括弧(『』)を使用します。

PARTNER 設定ファイル名

PARTNER の設定ファイル名を「JETARM_1.JPX など」と表記します。「ARM」の部分はご使用の CPU によって「SH」や「MIPS」と読み替えてください。数字は『MULTI コマンド』(163 頁)によって開いた PARTNER ウインドウの番号に相当します。

機材別表記

複数の機材への操作を区別するために使用します。ユーザーによって入力するべき部分はアンダーライン付きで表記し、**「↓」記号で**コマンドの確定を示します。

WINPC>	: Windows PC のコマンドプロンプトへの入力
LINUX86>	: Linux PC のコマンド入力
TGT>	: ターゲット Linux のコンソール入力
PT>	: PARTNER のコマンドウインドウへの入力
PT#>	: 複数の PARTNER ウインドウが起動しているときの # 番目の PARTNER のコマンドウインドウへの入力

注意書き

操作方法や取り扱いに特に注意が必要な箇所や、間違えやすい箇所の補足説明に使用します。

その他のコメント

覚えておくと便利な操作方法や Tips テクニック、コラムなどを記述する際に使用します。

目次

第 1 章　チュートリアル

この章では、評価・開発用ボードで Linux を使用し、PARTNER でデバッグする過程について例示します。

1.1 チュートリアルの概要

これから Linux カーネルおよびアプリケーションを開発・デバッグする方にもっとも早くわかりやすい手引きは、習うよりも実際にやって慣れてしまうことです。この章では KZM-ARM11-01 ボードと PARTNER-Jet デバッガを使用して組み込み Linux 環境のデバッグを解説していきます。
PARTNER デバッガは多くの機能を備えており、様々な状況下でのデバッグのテクニックもあります。その全てを網羅しようとするのではなく、できるだけ快適な開発環境とするための設定を選んでいきます。
詳しい設定や機能内容については後述のページを参照してください。

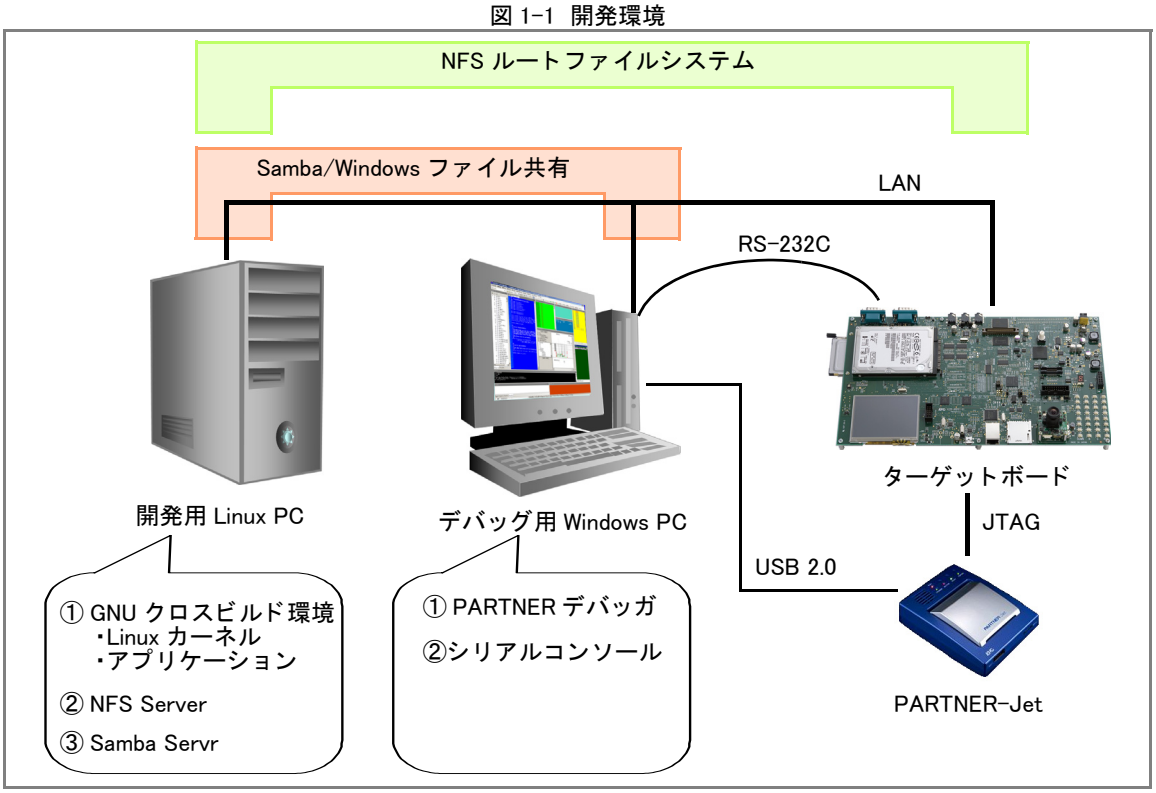
例として取り上げる KZM-ARM11-01 ボードは ARM11 コア搭載で各種のデバイスがサポートされた組み込み機器としてはハイエンドなスペックのボードです。ハードディスク、SD カード、USB ストレージやオンボード NAND Flash など様々なストレージデバイスから OS を起動することができますが、ここでは NFS を用いたネットワーク上のファイルシステムを使用します。デバッグのためにターゲットボード上に必要なインタフェースは JTAG と Ethernet コントローラだけですので、多くの組み込み Linux 開発環境で同様の構成を取ることが可能です。

この章では、組み込み Linux 環境のデバッグをチュートリアル方式で説明します。

1.2 開発環境について

KZM-ARM11-01 ボードには ARM 版の Linux 環境が CD-ROM で添付されているため、Linux カーネルのポーティング作業は行わなくても Linux 環境を立ち上げるために必要なソフトウェアは揃っています。
とはいえ、PC 上の Linux 環境とは違いターゲットボードとソフトウェア CD-ROM だけ用意してインストール＆実行というわけにはいきませんので、クロス開発環境を構築します。
構築する開発環境のハードウェア構成は図 1-1 のようになります。

図 1-1 開発環境



Linux カーネル及びアプリケーションの開発環境（コンパイル環境）として Linux PC を使用し、PARTNER デバッガの動作環境として Windows PC を使用します。Windows PC 上に VMWare 等仮想マシンを使用すれば物理的には開発用 Linux PC を省略することもできますが、構成としては PC2 台になります。
ターゲットボードと PC2 台を LAN で接続するわけですが、快適な開発環境にするポイントは NFS と Samba で同じディレクトリをネットワーク上に公開するということです。

1. ターゲットボードは NFS で Linux PC 上のディレクトリをルートファイルシステムとして使う
2. Windows PC は Samba でターゲットボードが見ているのと同じ場所を参照する

この 2 点によって PARTNER がデバッグ情報とソースコードを自動で対応付けできるようになりデバッグが快適になります。また、組み込みシステムのストレージ容量は通常あまり大きくないため、デバッグ情報付きのサイズの大きなプログラムをターゲットボードにはインストールしにくいと思いますが、NFS ルートファイルシステムを用いることで解決できます。

もちろんターゲット上で NFS ルートファイルシステムを使用することや、Samba によるファイル共有は必須ではありません。
・ターゲットはオンボードのストレージ上のルートファイルシステム（デバッグ情報なし）から起動
・Windows PC のハードディスク上にルートファイルシステムツリーのコピー（デバッグ情報付き）を配置
という構成であれば LAN 接続する必要もありません。
本チュートリアルで LAN 接続で NFS と Samba を使用するのは、[プログラムの修正]→[ビルド]→[ターゲットボードにデバッグ情報なしファイルを配置]→[デバッグ用 Windows PC へデバッグ情報付きファイルを配置]→[デバッグ] という繰り返し行う手順を楽にするためです。

KZM-ARM11-01 ボード付属の Linux OS は多くの組み込み向けディストリビューションがそうであるようにシリアルコンソールを採用しています。RS-232C は図 1-1 では Windows PC に接続していますが Linux PC に接続してもかまいません。

開発用 Linux PC のセットアップ方法の詳細は『付録 B 開発用 Linux PC の構築（225 頁）』で紹介していますので参考にしてください。
使用する開発環境のファイルパスを表 1-1 に示します。

表 1-1 Linux PC 上のファイル配置

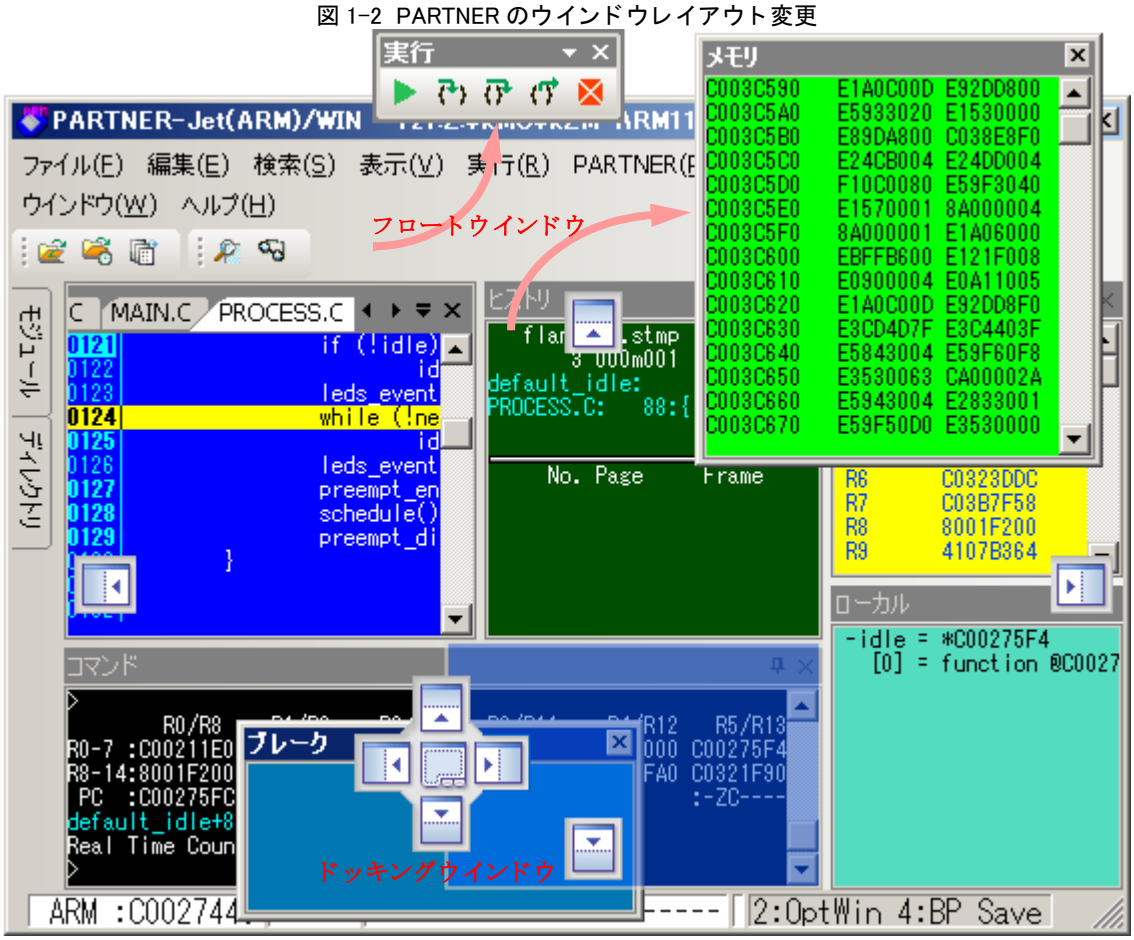
ソフトウェア	PATH
クロスツールチェイン	/opt/kmc/kzm-arm11/staging_dir/ コマンド：/opt/kmc/kzm-arm11/staging_dir/bin/ インクルード：/opt/kmc/kzm-arm11/staging_dir/include/ ライブラリ：/opt/kmc/kzm-arm11/staging_dir/lib/
ターゲット用ルートファイルシステム	/opt/kmc/kzm-arm11/root/
ターゲットの root ユーザのホーム	/opt/kmc/kzm-arm11/root/root/ ターゲット上のパス名：/root/
Linux カーネルソースツリー	/opt/kmc/kzm-arm11/build_src/linux/
Linux カーネル	/opt/kmc/kzm-arm11/build_src/linux/vmlinux /opt/kmc/kzm-arm11/build_src/linux/arch/arm/boot/zImage
デバッグサポートファイル (PRELOAD ライブラリ)	/opt/kmc/kzm-arm11/root/usr/lib/libkmcsup.so.2.0.0 ターゲット上のパス名：/usr/lib/libkmcsup.so.2.0.0

チュートリアルでは快適な開発環境とするために最も推奨される以下の設定を使用します。

- PARTNER の起動モードは「K2_LINUX_ADD_V26」を使用します（『 -OS オプション（138 頁）』参照）
- デバッグサポートファイルを PRELOAD ライブラリとして使用します（『2.3 アプリケーションデバッグサポートファイル（58 頁）』参照）

なお、以降のページでは説明のために PARTNER のスクリーンショットを多用しています。それぞれの画面形状がかなり異なりますが、それは図 1-2 のように PARTNER 内のウインドウレイアウトは自由に移動可能なためです。
できるだけトピックの内容に該当する箇所が見やすいように配置した画像を使用しています。

図 1-2 PARTNER のウインドウレイアウト変更

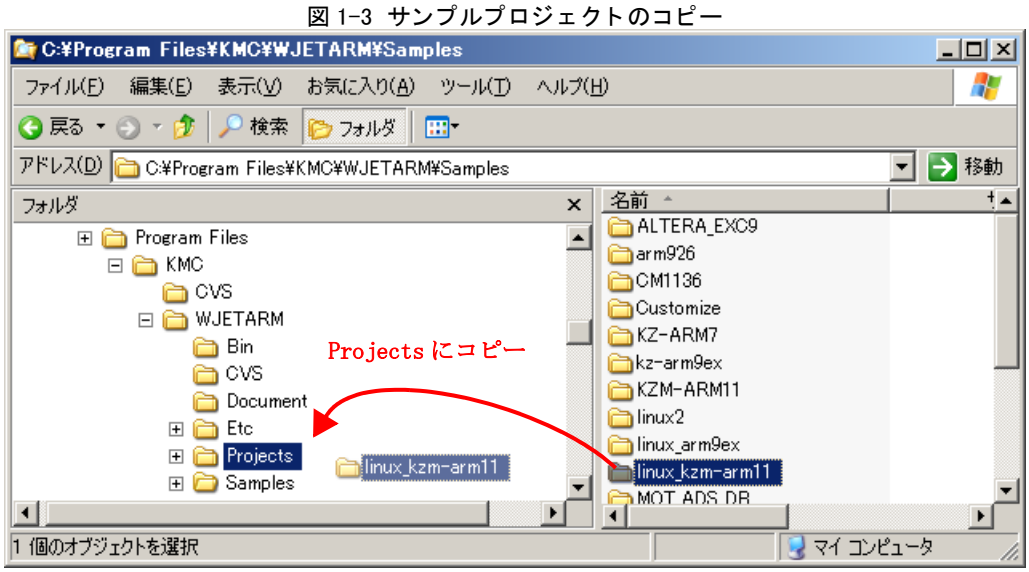


1.2.1 デバッグ用 Windows PC で PARTNER をセットアップする

PARTNER ソフトウェアと PARTNER-Jet ドライバのインストールはマニュアルを参照してください。ARM 版 PARTNER の接続とインストールは済んでいるものとして KZM-ARM11-01 用の設定を説明します。KZM-ARM11-01 ボード用の PARTNER の設定ファイルが PARTNER の Sample ディレクトリに用意されていますのでコピーします。

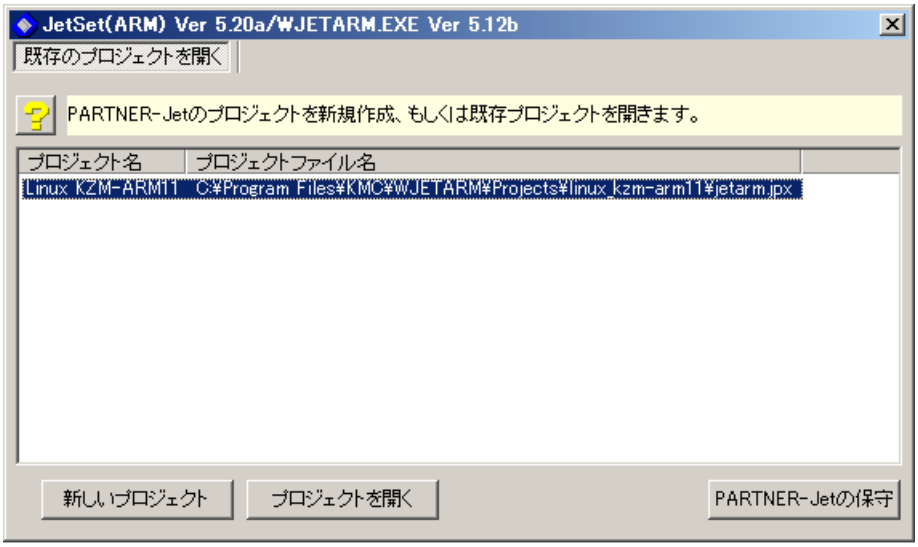
```
WINPC>c: ↓
WINPC>cd "Program Files¥KMC¥WJETARM¥Samples" ↓
WINPC>mkdir ..¥Projects¥linux_kzm-arm11 ↓
WINPC>copy linux_kzm-arm11 ..¥Projects¥linux_kzm-arm11 ↓
linux_kzm-arm11¥init.mcr
linux_kzm-arm11¥jetarm.cfg
linux_kzm-arm11¥jetarm.jpx
        3 個のファイルをコピーしました。
```

図 1-3 サンプルプロジェクトのコピー



コピーしたプロジェクトを PARTNER で開けば、かなり整ったボード用の設定が開きます。

図 1-4 PARTNER プロジェクトのオープン



CFG ファイルの設定はターゲットボードに依存する設定で、KZM-ARM11-01 ボードの場合はサンプルを変更する必要は特にありません。
ここでは構築する開発環境毎に設定が変わる、「PARTNER の起動オプション」を調整します。
(設定の詳細については『起動オプション（137 頁)』と『PARTNER の設定と起動（70 頁)』を参照してください)

● **デバッグ情報パス変換 (-XGX オプション)**

『Samba Server をセットアップする（234 頁)』の設定に合わせて -XGX/opt/kmc,z:¥kmc¥ と指定します。

● **アプリケーションデバッグ情報検索設定 (-RootDir オプション)**

ターゲットの NFS ルートを PARTNER が参照するパス -RootDir z:¥kmc¥kzm-arm11¥root を指定します。

● **OS デバッグモード指定 (-OS オプション)**

PARTNER のモードは、カーネルにもアプリケーションにも対応できて使いやすい「カーネル ADD モード」をお勧めします (カーネル (NON ADD) モードでできることは全てカーネル ADD モードでもできます)。ここでは 2.6 系 Linux カーネルを使用しますので「K2_LINUX_ADD_V26」を指定します。

● **カーネルオプション (-!! オプション)**

カーネル起動パラメータは init.mcr ファイルに記述することにして、default_command_line="" と空の引数を指定します。

ファイル共有設定と PARTNER の設定の関係をまとめると図 1-5 のようになります。

図 1-5 ファイル共有の相関

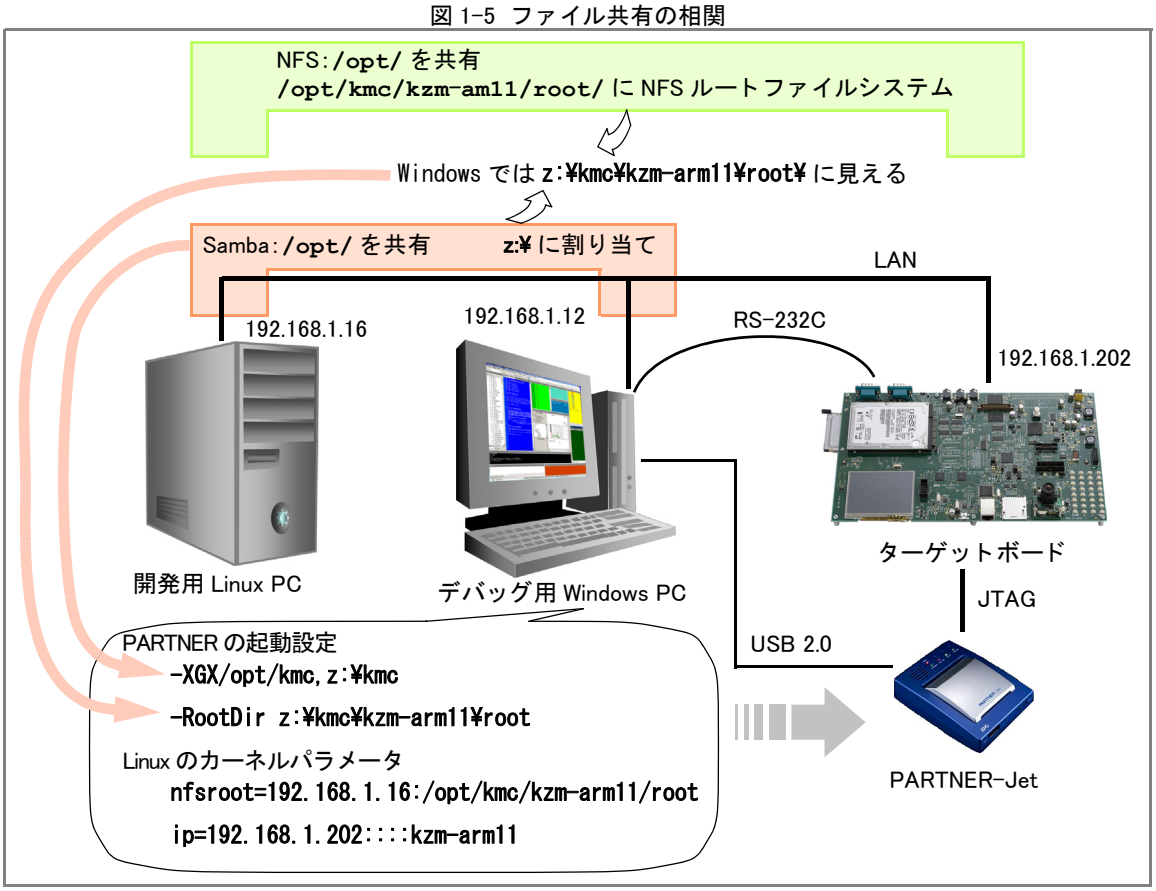
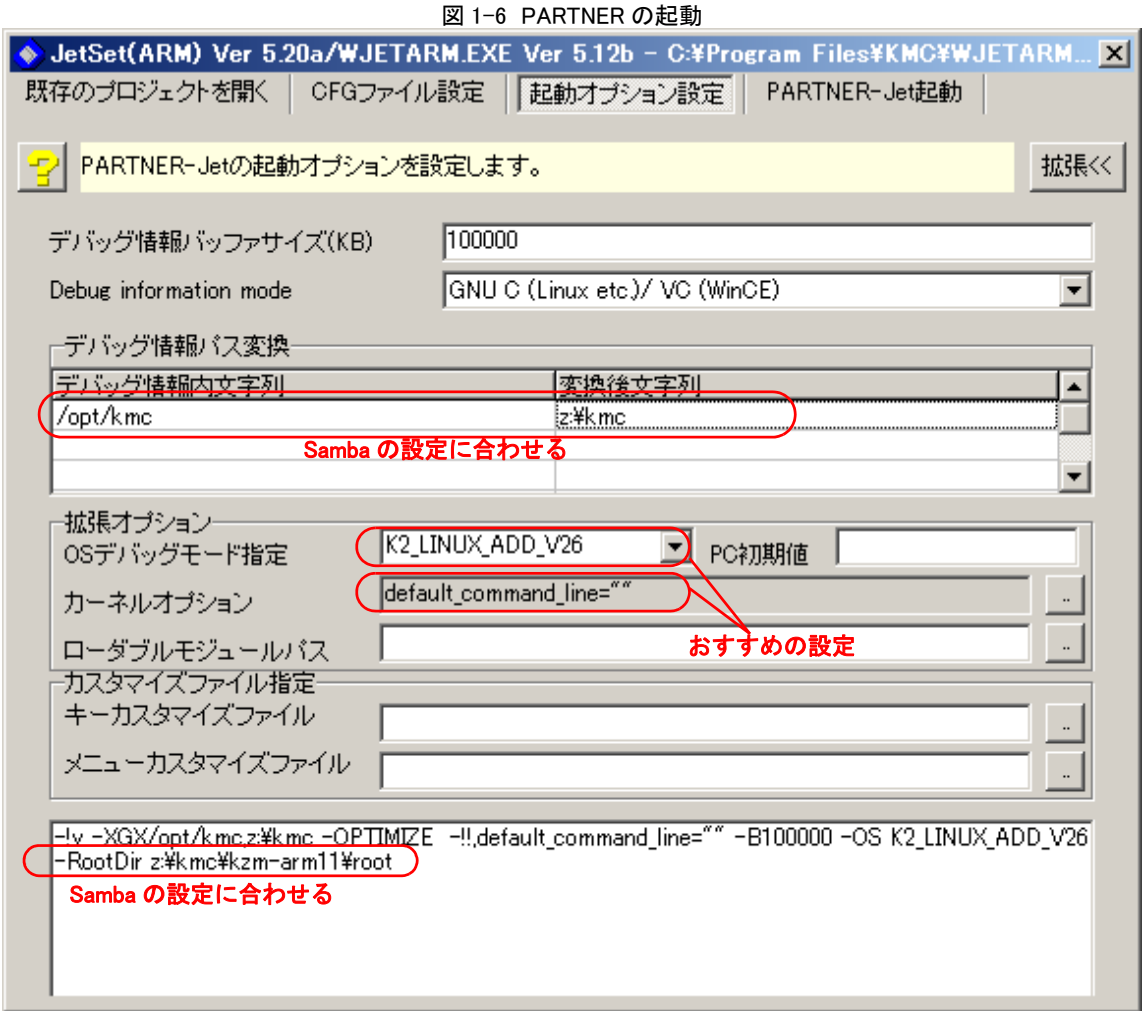


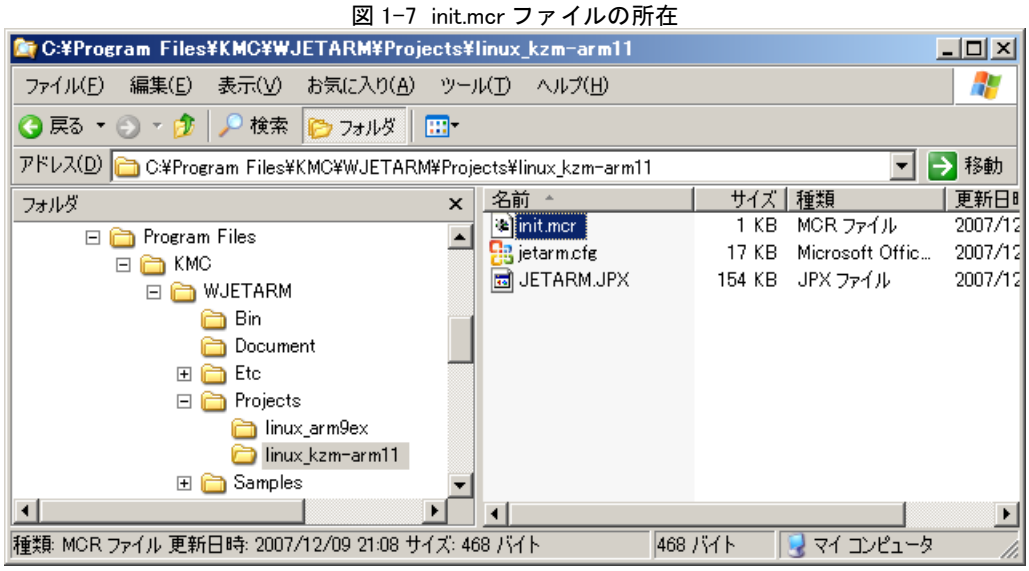
図 1-6 PARTNER の起動



次に、init.mcr ファイルを記述します。

init.mcr ファイルはプロジェクトフォルダに作成します。デフォルトは「init.mcr」で、PARTNER を MULTI コマンド（163 頁）で使用するときは init1.mcr, init2.mcr, init3.mcr... というファイル名が使われます。

図 1-7 init.mcr ファイルの所在



init.mcr の中には、Linux カーネルのロード設定をマクロにして入れておくと便利です。

図 1-8 init.mcr の記述

```
{load_linux26_nfs
init
cd "z:¥kmc¥kzm-arm11¥build_src¥linux"
l vmlinux noinitrd console=ttymxc0 root=/dev/nfs nfsroot=192.168.1.16:/opt/kmc/kzm-arm11/root
init=/linuxrc ip=192.168.1.202::::kzm-arm11 ./offs=0xc0000000
_pc=80008000
_r1=_956
_r0=0
linux set_attach_offset libkmcsup.so.2.0.0 0x00000624
br start_kernel.ex
}
{appls
ls %0
brc *
br main.ex
}
{appattach
attach %0
linux load_so
psid
}
```

NFS ルートのサーバとパス名

ターゲットボードの IP アドレス

VA → PA 変換オフセット（ARM 特有）

Linux カーネルのエントリーアドレスをプログラムカウンタにセット

ターゲット ID（ARM 特有）

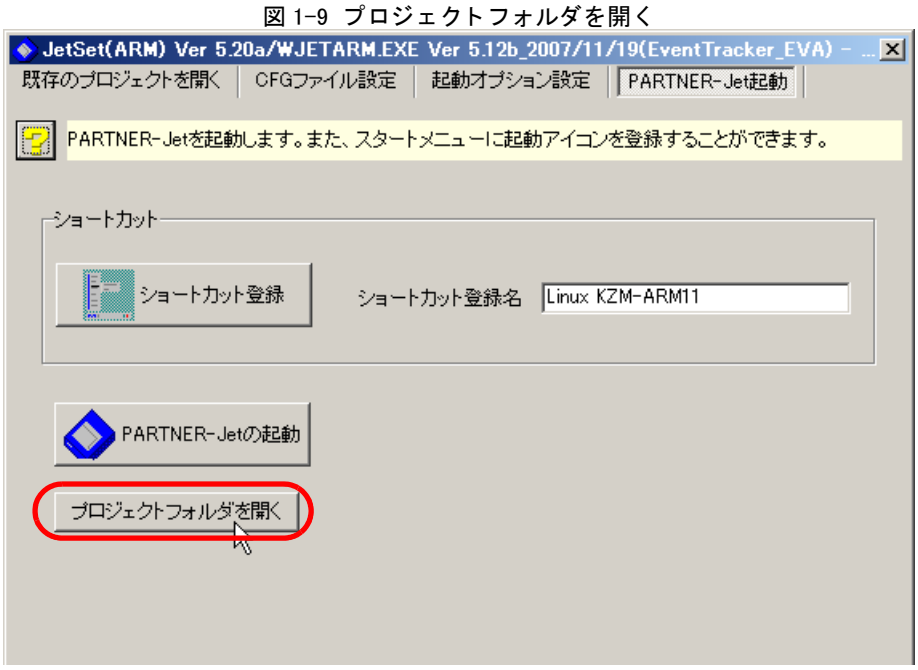
サポートファイル設定

VA でアクセス可能な場所に実行型ハードウェアブレークポイント

アプリケーションデバッグを楽にするマクロ（後述）

なお、プロジェクトフォルダを開くには JETSET の起動画面のボタンから開くと便利です。

図 1-9 プロジェクトフォルダを開く

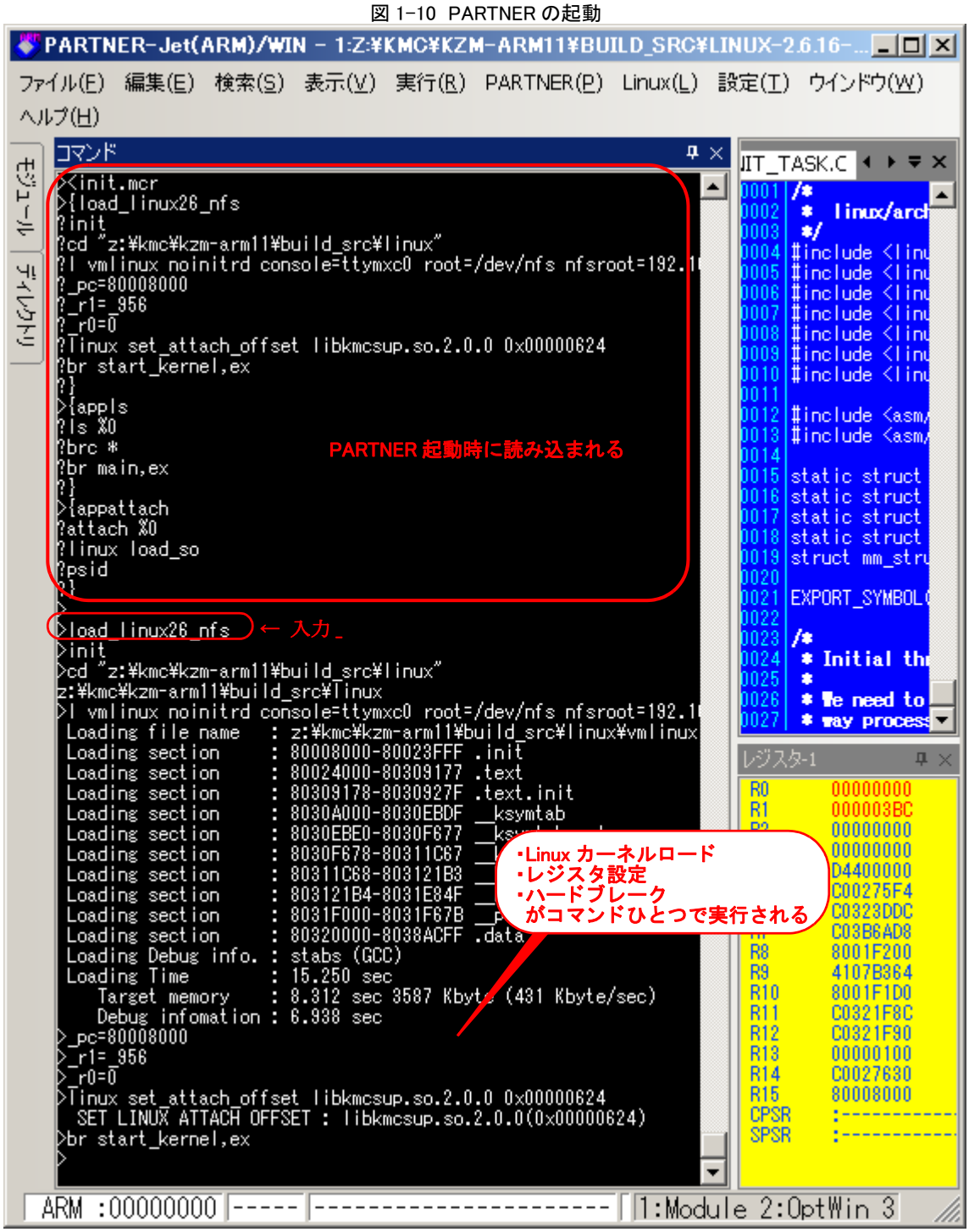


1.2.2 PARTNER から Linux をロード＆実行する

PARTNER のウインドウが開いた時点で init.mcr のマクロ設定が読み込まれています。登録したマクロ名をタイプするだけで Linux をロードして NFS ルートファイルシステムを使って起動します。

```
PT>load_linux26_nfs ↓
PT>g ↓
```

図 1-10 PARTNER の起動

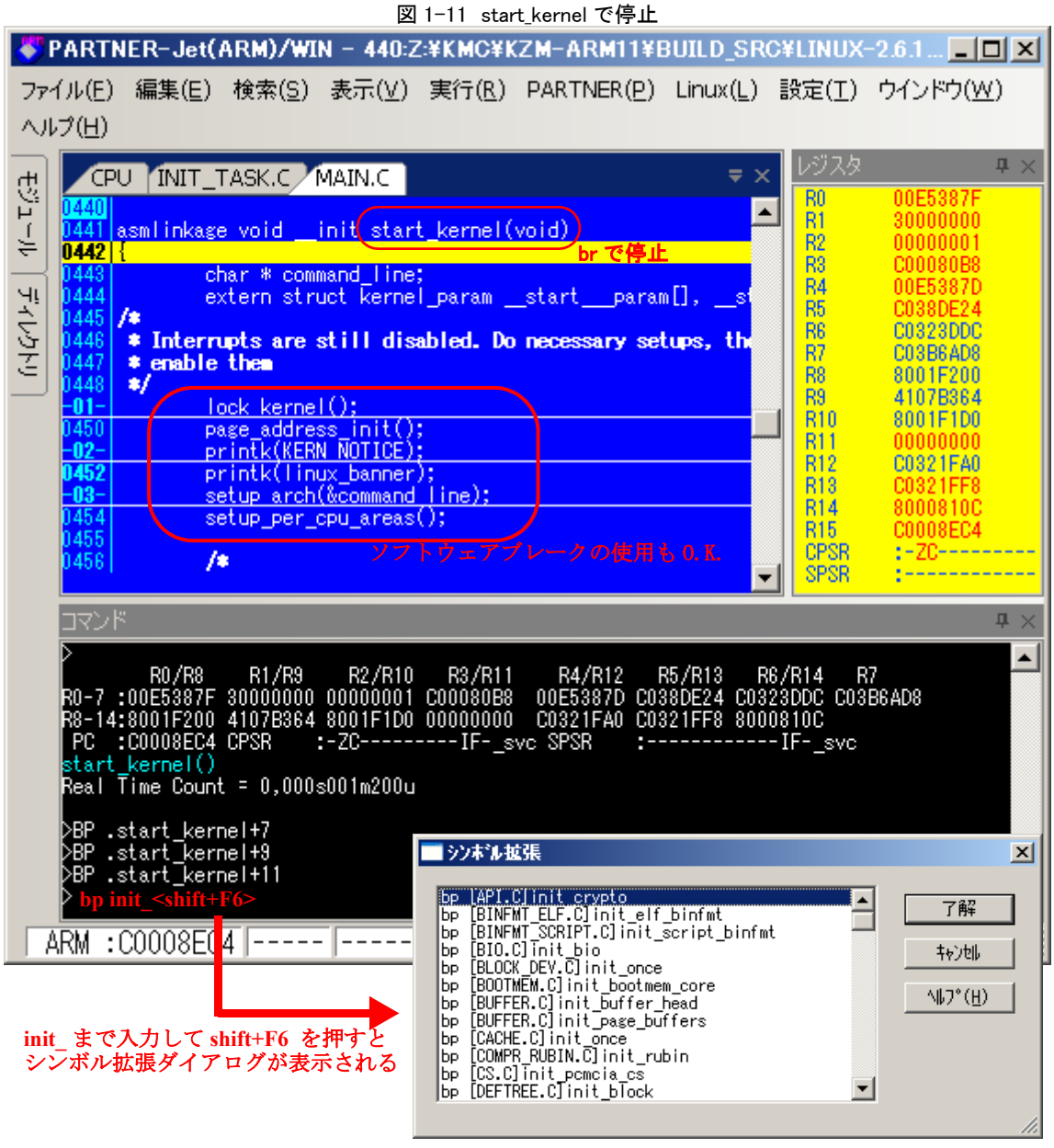


1.3 カーネル起動時の動きを見る

『PARTNER から Linux をロード＆実行する（10 頁）』の手順で Linux カーネルを実行すると start_kernel シンボルで停止します。
ARM CPU は MMU が有効になって仮想アドレスにアクセスできるようになるまではソフトウェアブレークポイントを使用することができませんが、start_kernel で停止以降は全てのデバッグ機能を使うことができます。

カーネル内のモジュールの初期化処理のシンボル名（関数名）は init で始まっていることが多いので、PARTNER のシンボル拡張機能を使用するとブレークポイントの設定が楽になります。

図 1-11 start_kernel で停止



1.4 簡単なアプリケーション

C 言語のサンプルプログラムの定番「HelloWorld」プログラムを作ってデバッグしてみます。

図 1-12 hello.c

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* const argv[])
{
        printf("Hello World\n");
        fflush(stdout);
        return EXIT_SUCCESS;
}
```

このソースコードには特に疑問は無いと思います。デバッグオプション付きでビルドします。

```
LINUX86>cd /opt/kmc/kzm-arm11/root/root ↓
LINUX86>vi hello.c ↓
LINUX86>arm-linux-gcc -o hello -g -O0 hello.c ↓
LINUX86>file hello ↓
hello: ELF 32-bit LSB executable, ARM, version 1 (SYSV), dynamically linked
(uses shared libs), not stripped
```

ターゲットに root でログインしていれば、/root/hello が見えるはずです。

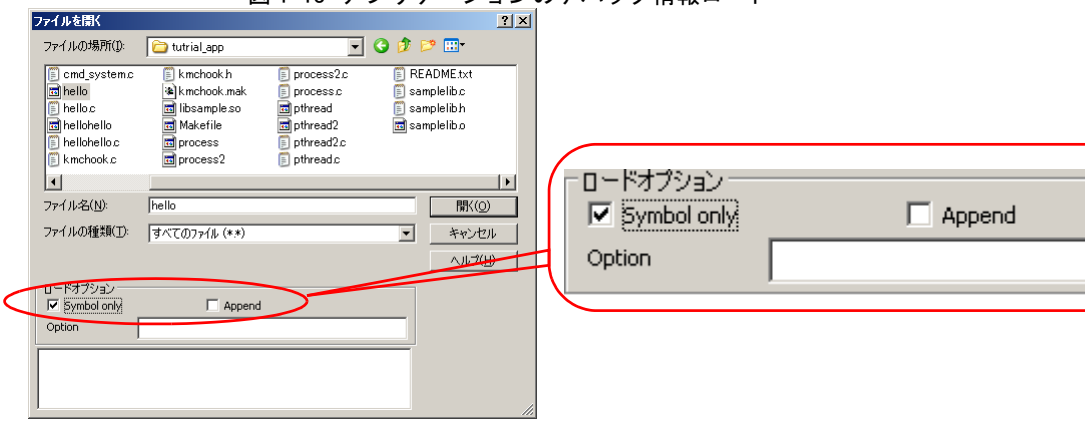
```
TGT> login: root ↓
TGT>pwd ↓
/root
TGT>ls ↓
hello  hello.c
```

PARTNER で [ESC] を押してターゲットを停止させ、デバッグ情報を読み込んでから、main 関数に実行型ブレークポイントを設定します。その際に MULTI コマンド（163 頁）で別のウインドウを開いておくと便利です。

```
PT>multi 2 ↓
PT2>cd z:¥kmc¥kzm-arm11¥root¥root ↓
PT2>ls hello ↓
PT2>br main.ex ↓
```

Linux のアプリケーションのプログラムコードはターゲット上で起動されてロードされる必要があります。GUI メニューからプログラムのデバッグ情報をロードするときには必ず [Symbol only] のチェックを行ってデバッグ情報のみをロードしてください。
また、まだ実行されてないアプリケーションの main シンボルにブレークポイントを設定するには、bp ではなく br を使用します。main シンボルはほぼ全てのプロセスが持っています。bp を使用してもエラーにはならず、「思うようにブレークがかからない」という現象になりますので注意してください。

図 1-13 アプリケーションのデバッグ情報ロード

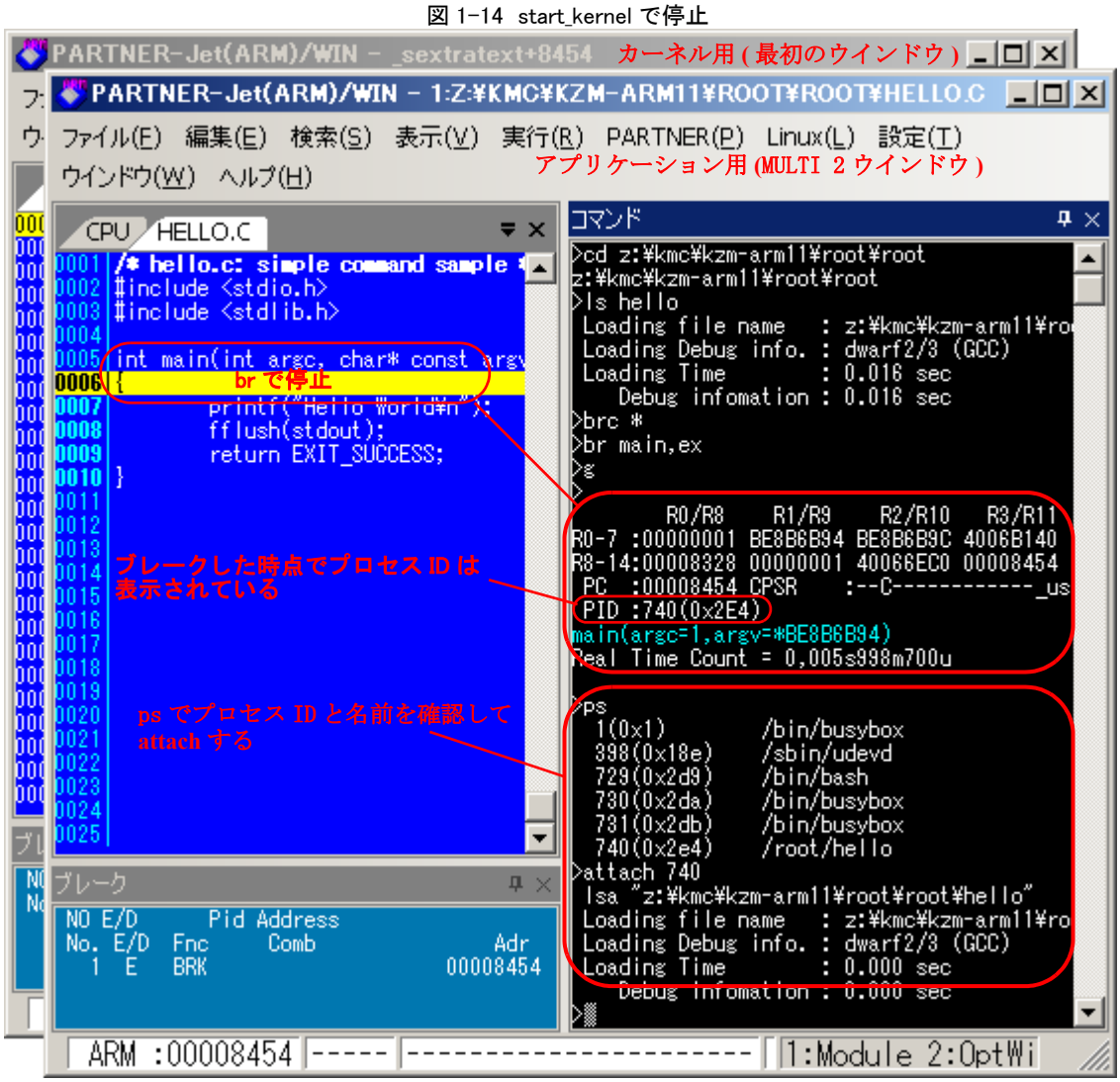


一度ロードされたファイルは、PARTNER がロードファイル名と場所をダイアログとコマンドヒストリに記憶するため、以降の操作を簡略化することができます。

ターゲット上でプログラムを実行すると main 関数で停止しますので、ATTACH コマンド（152 頁）を使用してプロセスにアタッチします。

```
PT2>ps ↓
（表示結果からプロセス ID が 740 だとわかります）
PT2>attach 740 ↓

PT2>attach hello ↓
（同一プログラムが 1 プロセスしか動作していないならば名前指定も可能です）
```

図 1-14 start_kernel で停止



この状態でプロセス内をトレースしたり、ソフトウェアブレークポイントを設定したりと、通常のデバッグ操作を行うことができます。

これだけでは普通のユーザーモードデバッガと変わらない動作ですが、PARTNER の特徴を活かしてもう一歩踏み込んだ操作をしてみたいと思います。

main 関数内の一行目の printf 文は標準出力への文字列「Hello World」の表示を行います。標準出力は Linux カーネル内で仮想端末 (tty) に結び付けられているために実際の動作はログインしたシリアルコンソールへの文字列の表示になります。printf 関数は tty デバイスファイルへの書き込みになっているはずです。

その様子を PARTNER ならば簡単に追いかけることができます。

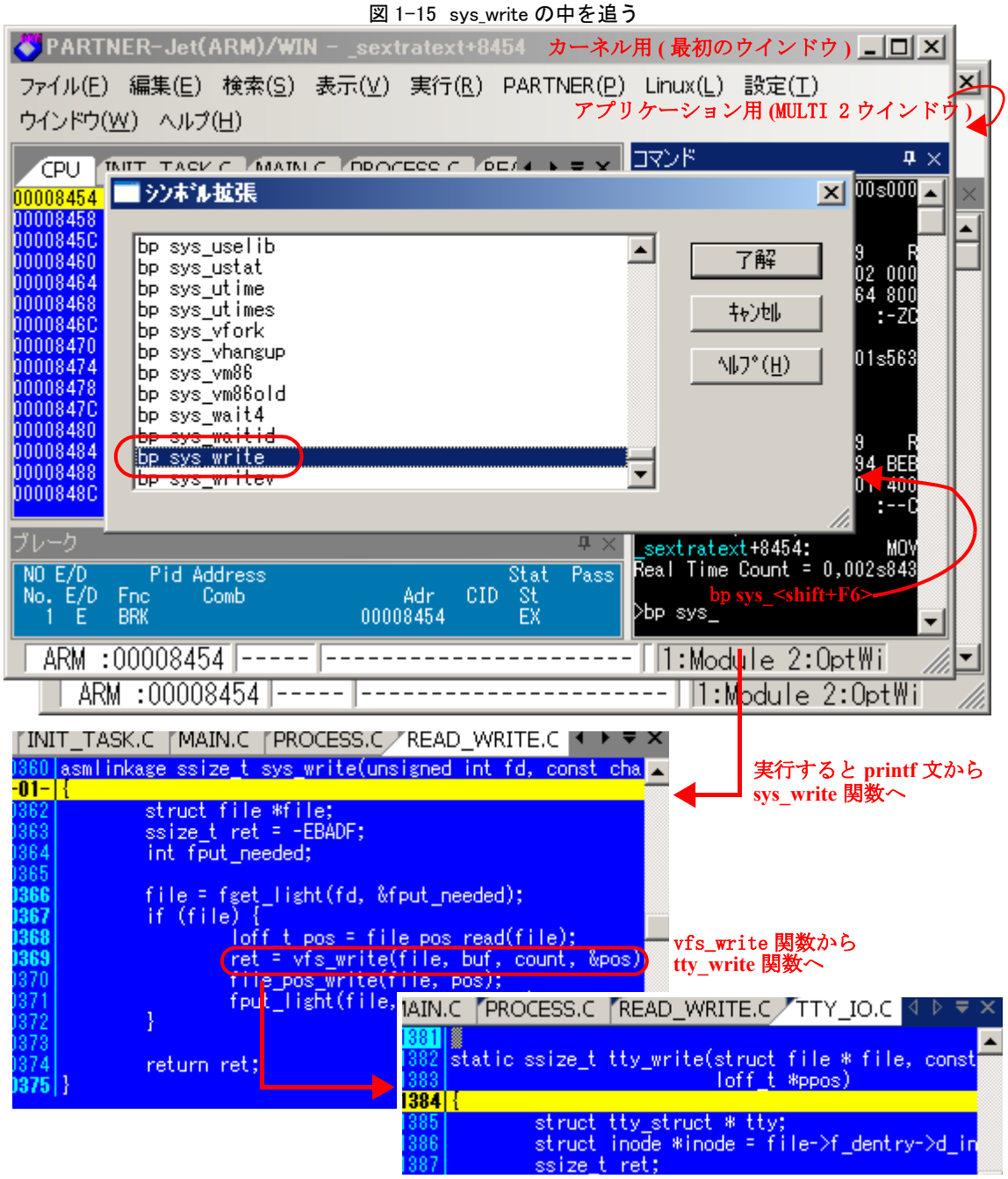
アプリケーション用の PARTNER ウインドウでいくつかブレークポイントを設定した後、カーネルのデバッグ情報が読み込まれている PARTNER ウインドウで ブレークポイントを設定します。

ライブラリ関数である printf() が最終的にファイル I/O を行うために用いるシステムコールは write() ですが、Linux のシステムコールのシンボル名は sys_ で始まっていますので bp sys_write と設定しておきます。

（アプリケーション用の PARTNER ウインドウ）
PT2>bp .main+1 ↓　（printf 文）
PT2>bp .main+2 ↓　（fflush 文）

（カーネル用の PARTNER ウインドウ）
PT>bp sys_write ↓

図 1-15　sys_write の中を追う



アプリケーション用ウインドウで printf 文が実行されたところで、次はカーネル用ウインドウがアクティブになり sys_write で停止します。カーネル内をステップしていくと tty_write 関数が呼ばれることがわかります。そのまま G コマンドを実行するとアプリケーション用ウインドウがアクティブになり fflush 関数の場所に停止します。つまり、アプリケーションのプロセスとカーネル内部とをシームレスに行き来してデバッグできます。

アタッチ後には LINUX コマンド（154 頁）の load_so を使用して共有ライブラリのデバッグ情報を読むようにすることを推奨しています。『init.mcr の記述（9 頁)』のようにマクロ登録しておくと、手順が以下のように簡略化できます。LINUX コマンドの書式 5 を使用して、ATTACH_AUTO_SO 設定を行うと ATTACH コマンド自体の動作を変更してアタッチ時に常に共有ライブラリのデバッグ情報を読み込むようにすることもできますが、多数の共有ライブラリを使用していてアタッチ処理が遅い場合など、共有ライブラリのデバッグ情報を読み込まない動作も行いたい場合にはマクロ登録が有効でしょう。

```
PT2>appls hello ↓
（デバッグ情報のロードと main へのブレークポイント設定）
PT2>g ↓
TGT>./hello ↓
PT2>ps ↓
（表示結果からプロセス ID が 740 のプログラムが hello だとわかる）
PT2>appattach 740 ↓
>attach 740
>linux load_so
 lsa "z:¥kmc¥kzm-arm11¥root¥lib¥ld-uClibc-0.9.29.so",/r .text=0x40000000
 Loading file name   : z:¥kmc¥kzm-arm11¥root¥lib¥ld-uClibc-0.9.29.so
 Loading Debug info. : stabs (GCC)
 Loading Time        : 0.032 sec
    Debug infomation : 0.032 sec

 lsa "z:¥kmc¥kzm-arm11¥root¥usr¥lib¥libkmcsup.so.2.0.0",/r .text=0x4000E000
 Loading file name   : z:¥kmc¥kzm-arm11¥root¥usr¥lib¥libkmcsup.so.2.0.0
 Loading Debug info. : stabs (GCC)
 Loading Time        : 0.000 sec
    Debug infomation : 0.000 sec

 lsa "z:¥kmc¥kzm-arm11¥root¥lib¥libuClibc-0.9.29.so",/r .text=0x40017000
 Loading file name   : z:¥kmc¥kzm-arm11¥root¥lib¥libuClibc-0.9.29.so
 Loading Debug info. : stabs (GCC)
 Loading Debug info. : dwarf2/3 (GCC)
 Loading Time        : 0.407 sec
    Debug infomation : 0.407 sec

 lsa "z:¥kmc¥kzm-arm11¥root¥lib¥libdl-0.9.29.so",/r .text=0x4006C000
 Loading file name   : z:¥kmc¥kzm-arm11¥root¥lib¥libdl-0.9.29.so
 Loading Debug info. : stabs (GCC)
 Loading Time        : 0.063 sec
    Debug infomation : 0.063 sec
>psid
PSID SET 740(0x2E4)  CURRENT 740(0x2E4) [ADD MODE]
APPLI. AREA : 00008000-00008FFF
APPLI. AREA : 00010000-00010FFF
APPLI. AREA : BED7E000-BED7EFFF
```

>
(attach,linux load_so が実行された後、psid で現在の状態を表示する)

本チュートリアルでは以後このマクロを使用します。

1.5 対話型アプリケーション

『簡単なアプリケーション（12 頁）』のプログラムは、すぐに処理を実行し、処理が終了したらプログラムも終了するようなタイプのアプリケーションでした（コマンド型、バッチ処理型などと呼ばれることもあります）。
プログラムの中にはもっとずっと長時間起動し続けるものもあります。ネットワーククライアントからの接続を受け付ける「サーバープログラム」や X-Window System などを使用した「GUI アプリケーション」などがこれに相当します。これらのプログラムの特徴は「対話型」であることです。対話相手がプログラムか人間かの差はありますが、常に 100% 処理し続けているわけではなく、何らかの事象や要求が起こるまで待っている時間が長いプログラムだともいえるでしょう。
このようなプログラムはシステム起動後すぐに実行され常駐プロセスにされることが多く、デバッグしたいタイミングも「アプリケーションを実行するとき」ではなく、「何らかの操作や事象を起こすとき」であることが多いでしょう。
PARTNER の実行中のプログラムへのアタッチデバッグ機能を使うことでこのようなプログラムのデバッグに対応することができます。

例として『hello.c（12 頁）』を少し改造して対話型にした hellohello.c を使用して説明します。

図 1-16 hellohello.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/select.h>

static int polldata(int fd)
{
        struct timeval tv;
        fd_set rfds;
        int ret;
        FD_ZERO(&rfds);
        FD_SET(fd, &rfds);
        tv.tv_sec = 0; tv.tv_usec = 0; /* polling */
        ret = select(fd+1, &rfds, NULL, NULL, &tv);
        if (ret==-1) perror("select");
        return ret;
}
static int echomsg()
{
        int count = 0;
        if (!polldata(STDIN_FILENO)) return 0;
        printf("Get message: "); fflush(stdout);
        for (;;) {
                int c;
                count += read(STDIN_FILENO, &c, 1);
                write(STDOUT_FILENO, &c, 1);
                if (!polldata(STDIN_FILENO)) break;
        }
        printf("count = %d\n", count); fflush(stdout);
        return count;
}
int main(int argc, char* const argv[])
{
#define NUM_MSGS 3
        const char* msgs[NUM_MSGS] = {
                "Hello...",
                "Are you there?",
                "Answer me back!",
        };
        unsigned long loop;
        printf("Hello World!\n");
        for (loop=0;; loop++) {
                printf("%s\n", msgs[loop%NUM_MSGS]);
                if (echomsg()) break;
                sleep(5);
        }
        return EXIT_SUCCESS;
}
```

このプログラムは 5 秒毎にターミナルへ pruntf 文でメッセージを表示し続けます。ターミナルにユーザからの入力があれば入力された文字列を表示して終了します。

デバッグオプション付きでコンパイルします。

```
LINUX86>cd /opt/kmc/kzm-arm11/root/root ↓
LINUX86>vi hellohello.c ↓
LINUX86>arm-linux-gcc -o hellohello -g -O0 hellohello.c ↓
LINUX86>file hellohello ↓
hellohello: ELF 32-bit LSB executable, ARM, version 1 (SYSV), dynamically linked
(uses shared libs), not stripped
```

ターゲット上で実行します。

図 1-17 対話型アプリケーションを実行

```
kzm-arm11 login: root
[root@kzm-arm11 ~]# ./hellohello
Hello World!
Hello...
Are you there?
Answer me back!
Hello...
```

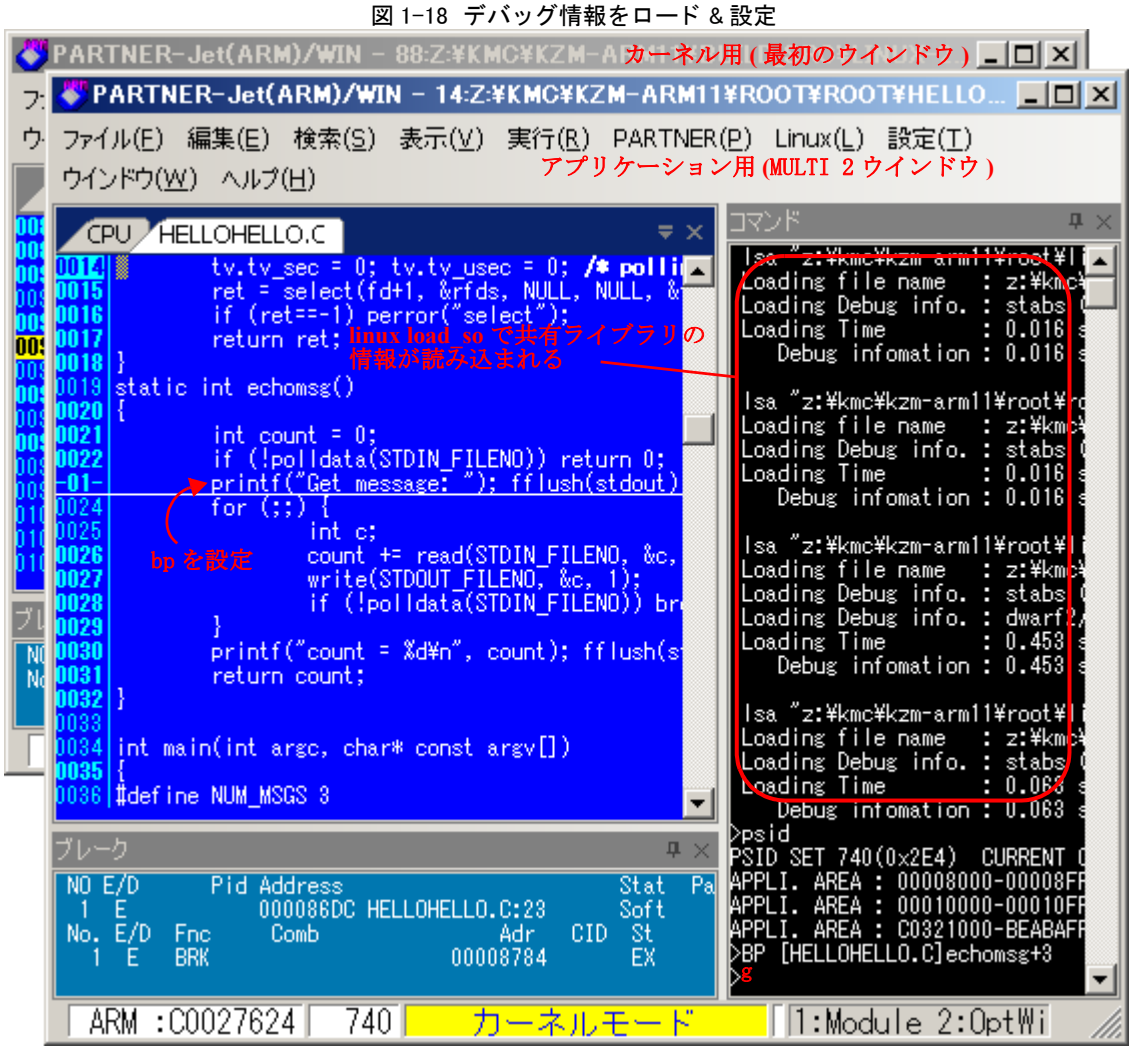
これで「プログラムが実行中」の状況を作ることができました。
PARTNER のウインドウで [ESC] キーを押してターゲットを停止し、デバッグ情報を読み込みます。

```
PT>multi 2 ↓
PT2>cd z:¥kmc¥kzm-arm11¥root¥root ↓
z:¥kmc¥kzm-arm11¥root¥root
PT2>appls hellohello ↓
```

main にブレークポイントが設定されますが、既に実行後なので止まりません。また、プロセスは実行中なのでこのまま ATTACH コマンド（152 頁）を使用してプロセスにアタッチします（つまり『簡単なアプリケーション（12 頁)』と手順は殆ど同じです)。

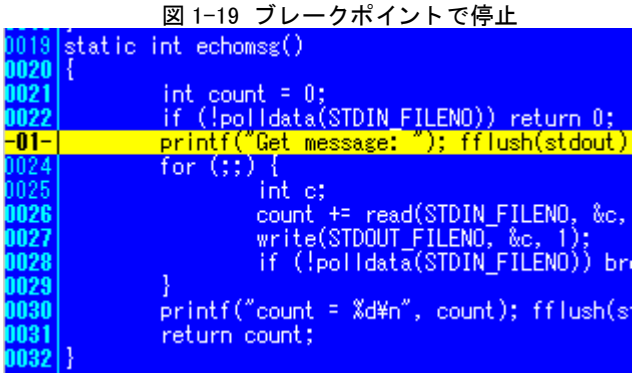
```
PT2>ps ↓
  1(0x1)        /bin/busybox
  398(0x18e)    /sbin/udevd
  601(0x259)    /bin/bash
  602(0x25a)    /bin/busybox
  603(0x25b)    /bin/busybox
  740(0x2e4)    /root/hellohello
（表示結果からプロセス ID が 740 だとわかります）
PT2>appattach 740 ↓
（マクロ内の linux load_so コマンドによって共有ライブラリの情報が読み込まれます）
PT2>BP [HELLOHELLO.C]echomsg+3 ↓
（ターミナルからの入力を受け付けたときに通る箇所にブレークポイントを設定）
PT2>g ↓
（継続実行）
```

図 1-18 デバッグ情報をロード & 設定



この状態でもプログラムは停止することなく定期的にコンソールへの出力を続けます。
ターミナルで何か文字を入力したとき初めてブレークポイントに停止します。

図 1-19 ブレークポイントで停止



1.6 pthreads ライブラリを使った並行処理

最近のプログラムへの要求仕様は複雑で高機能化していることや、組み込み機器でよく使われている RTOS のタスクモデルと動作イメージが近いことなどから、マルチスレッドによる平行プログラミングの手法はよく用いられていると思います。
しかしながら、これまで Linux のマルチスレッドプログラムを適切にデバッグできる環境というのはありませんでした。ターゲット上で動作するユーザーモードのデバッガ (GDB など) が利用している PTRACE システムコールのマルチスレッド・マルチプロセス機能の対応が不完全だということと、プロセス停止 (ブレーク) のためにシグナル配信機構を利用するためにスレッド数が増えると遅延が無視できなくなることが主な理由です。JTAG デバッガならばターゲットの CPU を停止させることができますし、デバッガはターゲット上のプロセスとして動作するわけではないため、そのような問題が起こりません。

ここでは pthreads ライブラリを用いたサンプルを使用して PARTNER で複数のスレッドコンテキストをデバッグしていきます。

図 1-20 pthread.c

```
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

static int printfflush(const char* fmt, ...)
{
    int r;
    va_list ap;
    va_start(ap, fmt); r=vprintf(fmt, ap); va_end(ap);
    fflush(stdout);
    return r;
}
static void func1(int* timesptr)
{
    int i, status = 1;
    for (i=0; i<15; i++) {
        printfflush("func1 doing [%d] times.\n", ++(*timesptr));
        sleep(1);
    }
    pthread_exit((void*)status);
}
static void func2(int* timesptr)
{
    int i, status = 2;
    for (i=0; i<10; i++) {
        printfflush("\t\t\tfunc2 doing [%d] times.\n", ++(*timesptr));
        sleep(2);
    }
    pthread_exit((void*)status);
}
static void show_total(int times1, int times2)
{
    int total = times1 + times2;
    printfflush("func1=%d, func2=%d for a total of %d\n", times1, times2, total);
}
int main(int argc, char* const argv[])
{
    pthread_t thread1, thread2;
    int r1, r2;
    int status;
    r1=0; r2=0;
    pthread_create(&thread1, NULL, (void*) func1, (void*) &r1);
    printfflush("thread1 = 0x%08X\n", thread1);
    pthread_create(&thread2, NULL, (void*) func2, (void*) &r2);
    printfflush("thread2 = 0x%08X\n", thread2);

    pthread_join(thread1, (void*)&status);
    printfflush("thread1 endstatus = %d\n", status);
    pthread_join(thread2, (void*)&status);
    printfflush("thread2 endstatus = %d\n", status);

    show_total(r1, r2);
    return EXIT_SUCCESS;
}
```

サンプルコード pthread.c は pthreads ライブラリのごく基本的な使い方の例です。
main 関数からスレッドを 2 つ生成し、それぞれに func1 関数、func2 関数の処理をさせます。つまりプログラム内で同時に実行されるコンテキストは 3 つになります。
func1 と func2 は両方とも一定の間隔でターミナルへの文字出力をします。両関数でやっていることはほとんど同じです。ひとつの関数を 2 つのスレッドで実行するようにもできますが、紙面上の見やすさから 2 つに分けています。
main 関数は 2 つのスレッドを生成した後はそれぞれが終了するのを待ちます。一般的な RTOS のタスクは他のタスクの終了を待つことはできないのが普通 (処理がより軽くなる) ですが、Linux(Unix) 上の pthreads ライブラリの実装ではプロセスと同様に他のスレッドの終了を待つことができる「結合可能スレッド」がデフォルトになっているのが普通なので、pthread_join 関数でスレッドの終了を待つことができます。

実行イメージは図 1-21 のようになります。

図 1-21 pthread アプリケーションを実行

```
[root@kzm-arm11 ~]# ./pthread
thread1 = 0x00000402
thread2 = 0x00000803
func1 doing [1] times.
                        func2 doing [1] times.
func1 doing [2] times.
                        func2 doing [2] times.
func1 doing [3] times.
func1 doing [4] times.
                        func2 doing [3] times.
func1 doing [5] times.
func1 doing [6] times.
                        func2 doing [4] times.
func1 doing [7] times.
func1 doing [8] times.
                        func2 doing [5] times.
func1 doing [9] times.
func1 doing [10] times.
                        func2 doing [6] times.
func1 doing [11] times.
func1 doing [12] times.
                        func2 doing [7] times.
func1 doing [13] times.
func1 doing [14] times.
                        func2 doing [8] times.
func1 doing [15] times.
thread1 endstatus = 1
                        func2 doing [9] times.
                        func2 doing [10] times.
thread2 endstatus = 2
func1=15, func2=10 for a total of 25
[root@kzm-arm11 ~]#
```

このプログラムは実行から終了までしばらくかかりますので、プログラムを実行した後からプロセスにアタッチしてもかまいませんが、ここでは main の先頭からデバッグしていくことにします。

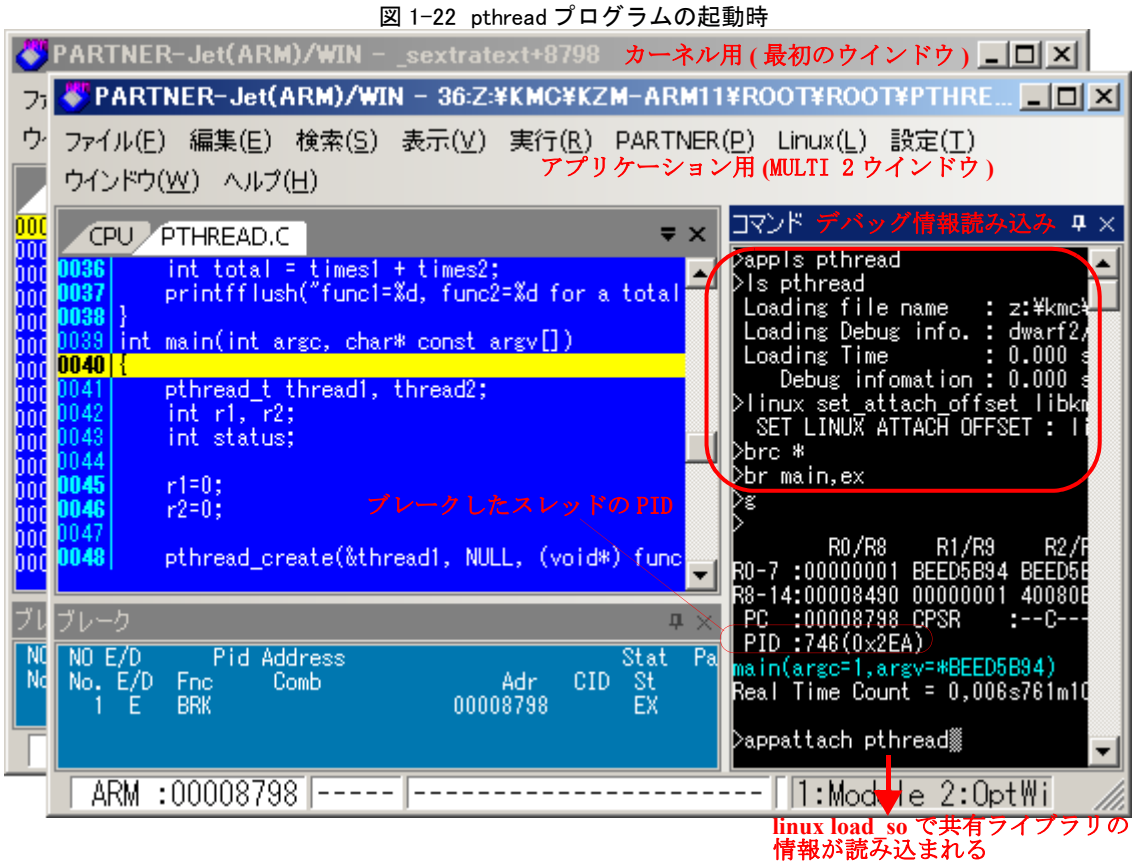
デバッグオプション付きでコンパイルします。

```
LINUX86>cd /opt/kmc/kzm-arm11/root/root ↓
LINUX86>vi pthread.c ↓
LINUX86>arm-linux-gcc -o pthread -g -O0 pthread.c -lpthread ↓
LINUX86>file pthread ↓
pthread: ELF 32-bit LSB executable, ARM, version 1 (SYSV), dynamically linked
(uses shared libs), not stripped
```

PARTNER にデバッグ情報を読み込みます。

PT>multi 2 ↓
PT2>cd z:¥kmc¥kzm-arm11¥root¥root ↓
z:¥kmc¥kzm-arm11¥root¥root
PT2>appls pthreads ↓
PT2>g ↓
TGT>./pthread ↓
（main でブレークする）
PT2>appattach pthread ↓
（プロセスにアタッチし、共有ライブラリのデバッグ情報を読み込む）

図 1-22 pthread プログラムの起動時



本チュートリアルでは PARTNER を ADD モードで起動していますので、このままアプリケーション用のウインドウの中で func1, func2 内にブレークポイントを設定してかまいません。
ブレークポイントを設定して実行すると図 1-23、図 1-24 のようにひとつのウインドウの中でコンテキストが切り替わりながらブレークされます。

図 1-23 スレッドコンテキストでブレーク時

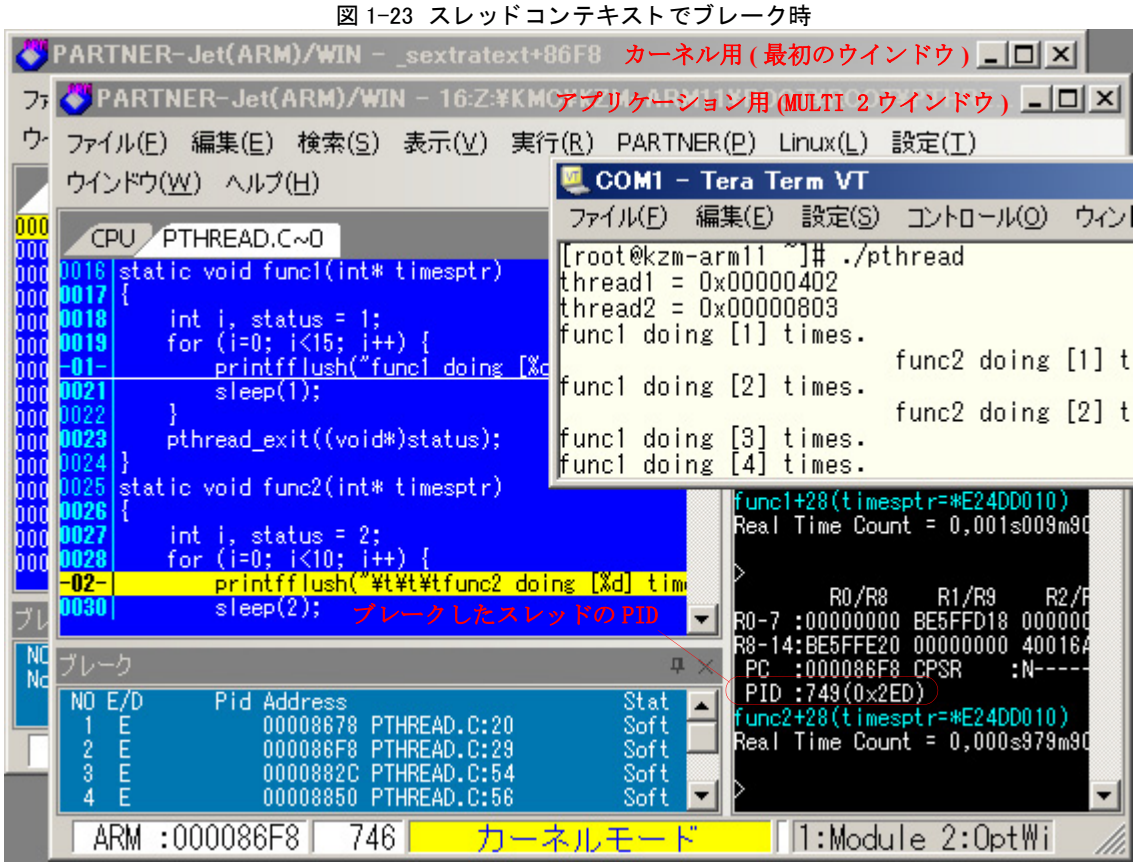
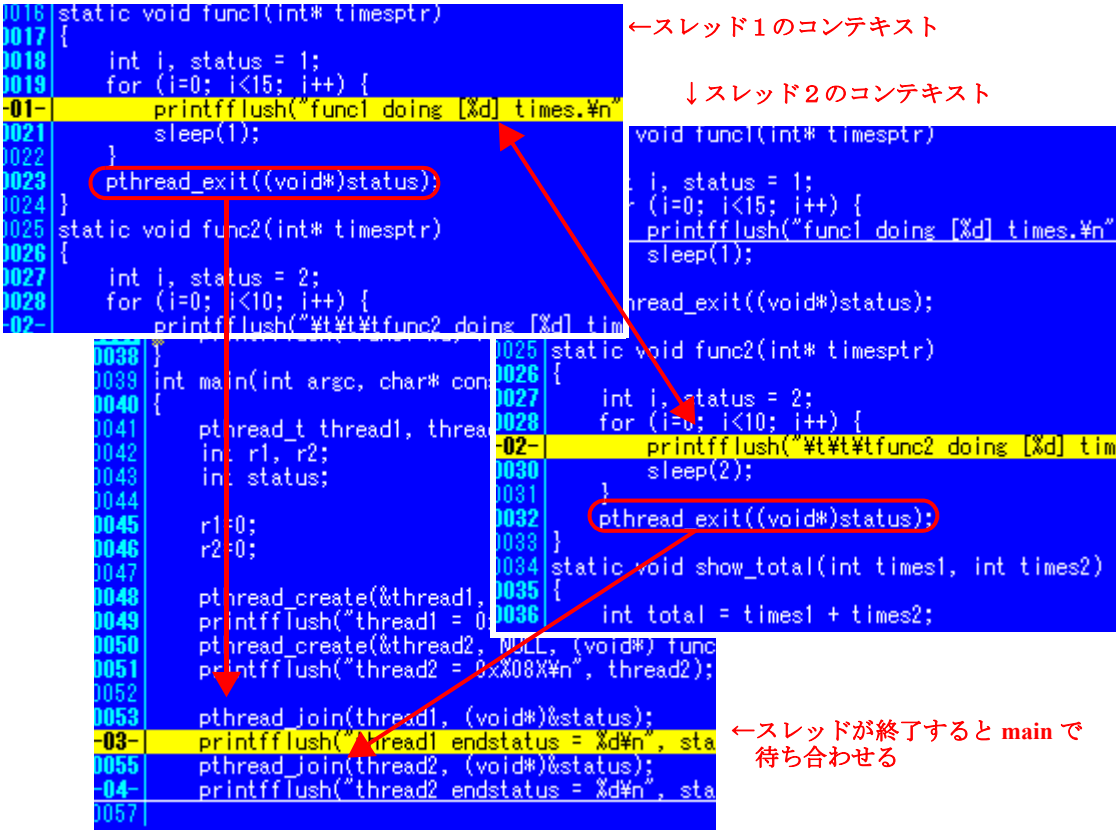
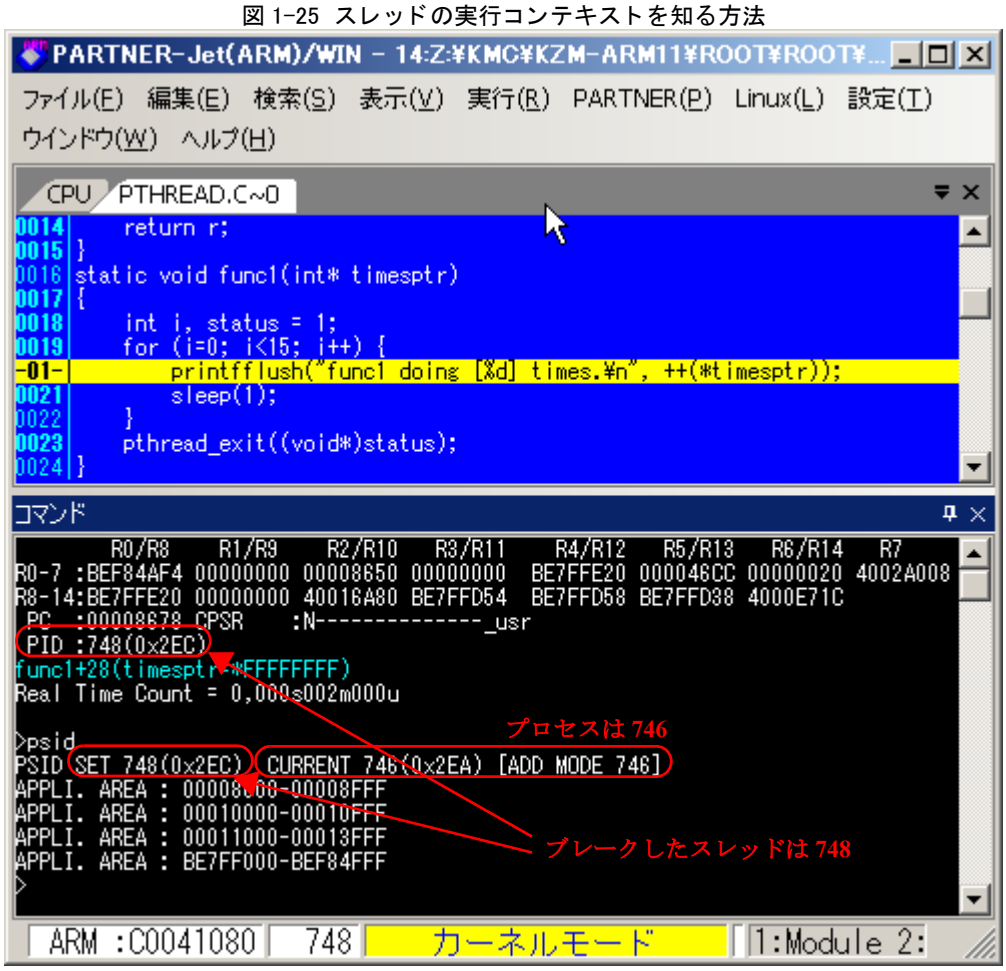


図 1-24 各スレッドコンテキストの動作イメージ



スレッドの場合は同じ関数が別のスレッドで実行されることもありますし、各スレッドのプログラム名は同じなのでブレーク中の実行コンテキストはコードウインドウの表示や PS コマンド（150 頁）で判断できるとは限りません。
どの実行コンテキストでブレークしたかはコマンドウインドウに表示される PID 表示や PSID コマンド（158 頁）でわかります。プロセス内でアタッチされているすべてのスレッド情報は THREAD コマンド（153 頁）で表示できます。

図 1-25 スレッドの実行コンテキストを知る方法



さらに、THREAD コマンド（153 頁）や K コマンド（164 頁）を使用するとスレッドの詳細な状態を確認できます。

```
PT2>thread ↓
 pid:746(0x2EA) task_struct:C6434CA0 pc:4003EE18
 pid:748(0x2EC) task_struct:C6435220 pc:4000E668
  （カーネル内の task_struct のアドレス、プログラムカウンタの表示）
PT2>
PT2>k 748 ↓
 KMC-SUPPORT.C: 468 : 4000E71C __kmc_pthread_entry+64(arg=*40001AAC)
 LIB1FUNCS.ASM: 195 : 4000E668 __kmc_sleep_thread+44()
  （スレッドのスタックの関数コールヒストリが表示されます）
PT2>
```

1.7 fork システムコールによる並行処理

Linux はマルチプロセスで動作する OS なので、平行プログラミングの手法としてプロセスを使用することもできます。

図 1-26 process.c は『図 1-20 pthread.c』とほぼ同じ動作を fork システムコールを用いたプロセスで実装したものです。各関数ともにほぼ全ての行が対になるように記述していますので比較してみてください。

図 1-26 process.c

```
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h>
#include <errno.h>

static int printfflush(const char* fmt, ...)
{
    int r;
    va_list ap;
    va_start(ap, fmt); r=vprintf(fmt, ap); va_end(ap);
    fflush(stdout);
    return r;
}
static void func1(int* timesptr)
{
    int i, status = 1;
    for (i=0; i<15; i++) {
        printfflush("func1 doing [%d] times.\n", ++(*timesptr));
        sleep(1);
    }
    exit(status);
}
static void func2(int* timesptr)
{
    int i, status = 2;
    for (i=0; i<10; i++) {
        printfflush("\t\t\tfunc2 doing [%d] times.\n", ++(*timesptr));
        sleep(2);
    }
    exit(status);
}
static void show_total(int times1, int times2)
{
    int total = times1 + times2;
    printfflush("func1=%d, func2=%d for a total of %d\n", times1, times2, total);
}
int main(int argc, char* const argv[])
{
    pid_t child1, child2;
    int shmid, *shmaddr;
    int *r1ptr, *r2ptr;
    int status;

    if ((shmid=shmget(IPC_PRIVATE, 2*sizeof(int), 0660)) == -1)
        perror("shmget"), exit(1);
    if ((shmaddr=(int*)shmat(shmid, (void*)0, 0)) == (void*)-1)
        perror("shmat"), exit(1);
    printfflush("shmid[%d] shmaddr[0x%08X]\n", shmid, (unsigned int)shmaddr);
    r1ptr = shmaddr; *r1ptr = 0;
    r2ptr = shmaddr+1; *r2ptr = 0;

    if ((child1 = fork())==0) func1(r1ptr);
    printfflush("child1 = %d\n", child1);
    if ((child2 = fork())==0) func2(r2ptr);
    printfflush("child2 = %d\n", child2);

    waitpid(child1, &status, 0);
    printfflush("child1 endstatus = %d\n", WEXITSTATUS(status));
    waitpid(child2, &status, 0);
    printfflush("child2 endstatus = %d\n", WEXITSTATUS(status));

    show_total(*r1ptr, *r2ptr);
    shmdt(shmaddr);
    return EXIT_SUCCESS;
}
```

デバッグオプション付きでコンパイルします。
(WEXITSTATUS などの値が Linux PC とターゲット用 Linux で異なっているので、参照先を明確にするために -I オプションを使ってインクルードディレクトリを指定しています)

```
LINUX86>cd /opt/kmc/kzm-arm11/root/root ↓
LINUX86>vi pthread.c ↓
LINUX86>arm-linux-gcc -o process -g -O0 -I../../staging_dir/include process.c ↓
LINUX86>file process ↓
process: ELF 32-bit LSB executable, ARM, version 1 (SYSV), dynamically linked
(uses shared libs), not stripped
```

実行イメージもほとんど同じになります。

図 1-27 process アプリケーションを実行

```
[root@kzm-arm11 ~]# ./process
shmid[196614] shmaddr[0x40006000]
func1 doing [1] times.
child1 = 763
                        func2 doing [1] times.
child2 = 764
func1 doing [2] times.
                        func2 doing [2] times.
func1 doing [3] times.
func1 doing [4] times.
                        func2 doing [3] times.
func1 doing [5] times.
func1 doing [6] times.
                        func2 doing [4] times.
func1 doing [7] times.
func1 doing [8] times.
                        func2 doing [5] times.
func1 doing [9] times.
func1 doing [10] times.
                        func2 doing [6] times.
func1 doing [11] times.
func1 doing [12] times.
                        func2 doing [7] times.
func1 doing [13] times.
func1 doing [14] times.
                        func2 doing [8] times.
func1 doing [15] times.
child1 endstatus = 1
                        func2 doing [9] times.
                        func2 doing [10] times.
child2 endstatus = 2
func1=15, func2=10 for a total of 25
[root@kzm-arm11 ~]#
```

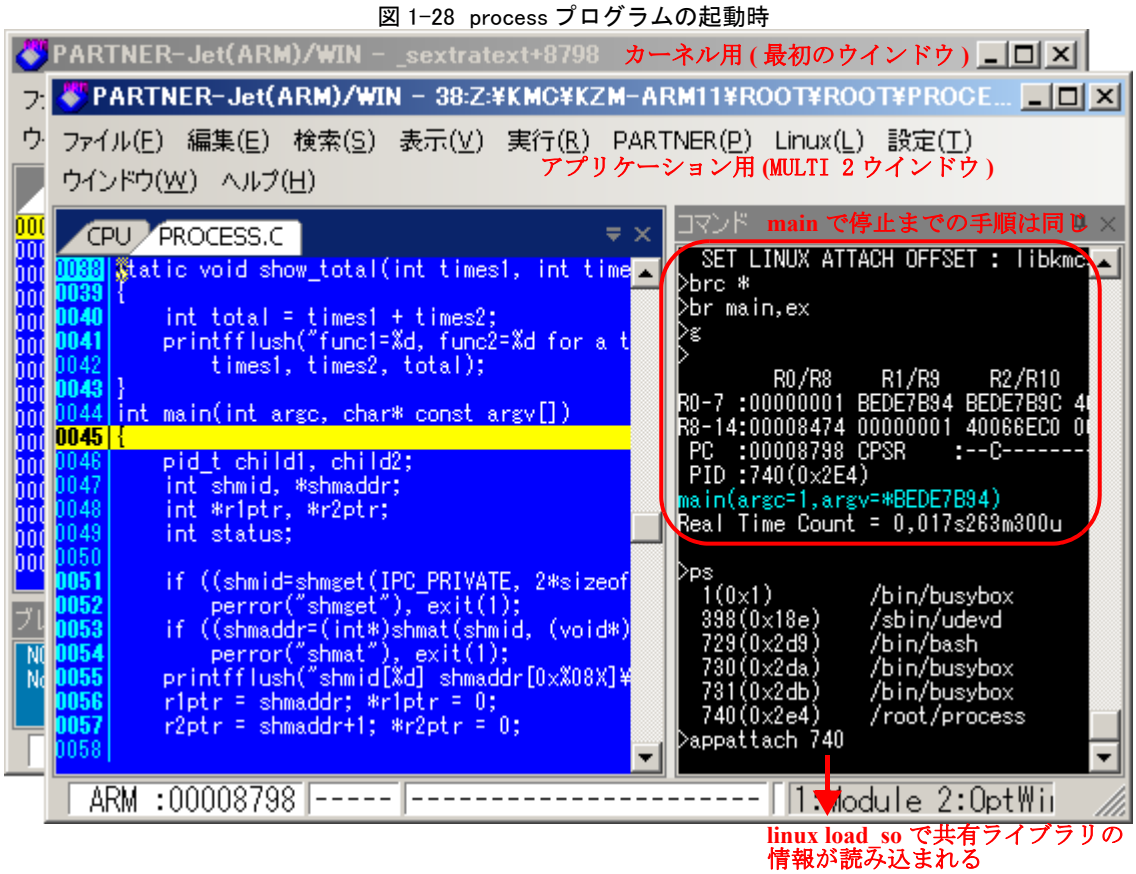
このプログラムを実際に PARTNER でデバッグしていきます。
デバッグ情報をロードしてプログラムを実行し、main 関数で止めるところまでは『1.6 pthreads ライブラリを使った並行処理』と全く同じです。

PARTNER にデバッグ情報を読み込みます。

```
PT>multi 2 ↓
PT2>cd z:¥kmc¥kzm-arm11¥root¥root ↓
z:¥kmc¥kzm-arm11¥root¥root
PT2>appls process ↓
PT2>g ↓
TGT>./process ↓ (main でブレークする)
PT2>ps ↓
  1(0x1)        /bin/busybox
  398(0x18e)    /sbin/udevd
  729(0x2d9)    /bin/bash
  730(0x2da)    /bin/busybox
  731(0x2db)    /bin/busybox
  740(0x2e4)    /root/process
PT2>appattach 740 ↓
```

(プロセスにアタッチし、共有ライブラリのデバッグ情報を読み込む)

図 1-28 process プログラムの起動時



ここから先、func1 と func2 にもブレークポイントを設置していきたいのですが、マルチスレッドのサンプルと異なりプロセスの場合は func1 と func2 は main と別プロセスになりますので、アドレス空間が別になります。マルチスレッドのサンプルと同様の手順を取ろうとすると問題が 2 つあります。

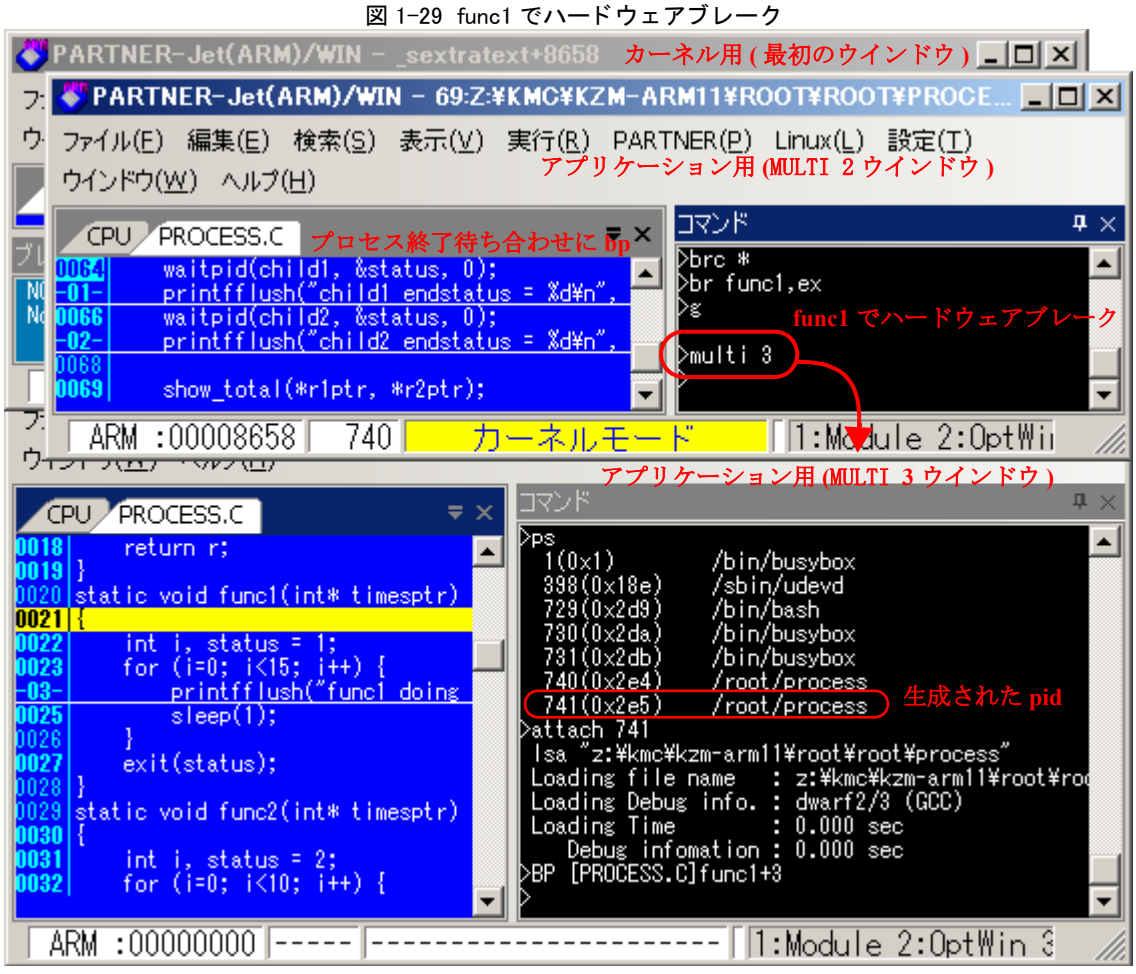
(1) この時点で func1,func2 にはソフトウェアブレークポイントは設定できない

(2) ひとつのウインドウで複数プロセスをデバッグすると使う人間の方が混乱してしまう

func1,func2 で停止するのはハードウェアブレークポイントを使用し、ADD モードであってもプロセス毎に PARTNER のウインドウを追加で 1 枚開くことにします。

```
(MULTI2 ウインドウ)
PT2>BP .main+20 ↓   (waitpid(child1,...) の後の printf 文)
PT2>BP .main+22 ↓   (waitpid(child2,...) の後の printf 文)
PT2>brc * ↓
PT2>br func1.ex ↓
PT2>g ↓
(func1 でハードウェアブレークがかかる)
PT2>multi 3 ↓

(MULTI3 ウインドウ)
PT3>appls process ↓
PT3>ps ↓
  1(0x1)        /bin/busybox
  398(0x18e)    /sbin/udevd
  729(0x2d9)    /bin/bash
  730(0x2da)    /bin/busybox
  731(0x2db)    /bin/busybox
  740(0x2e4)    /root/process
  741(0x2e5)    /root/process
PT3>attach 741 ↓
```

図 1-29 func1 でハードウェアブレーク

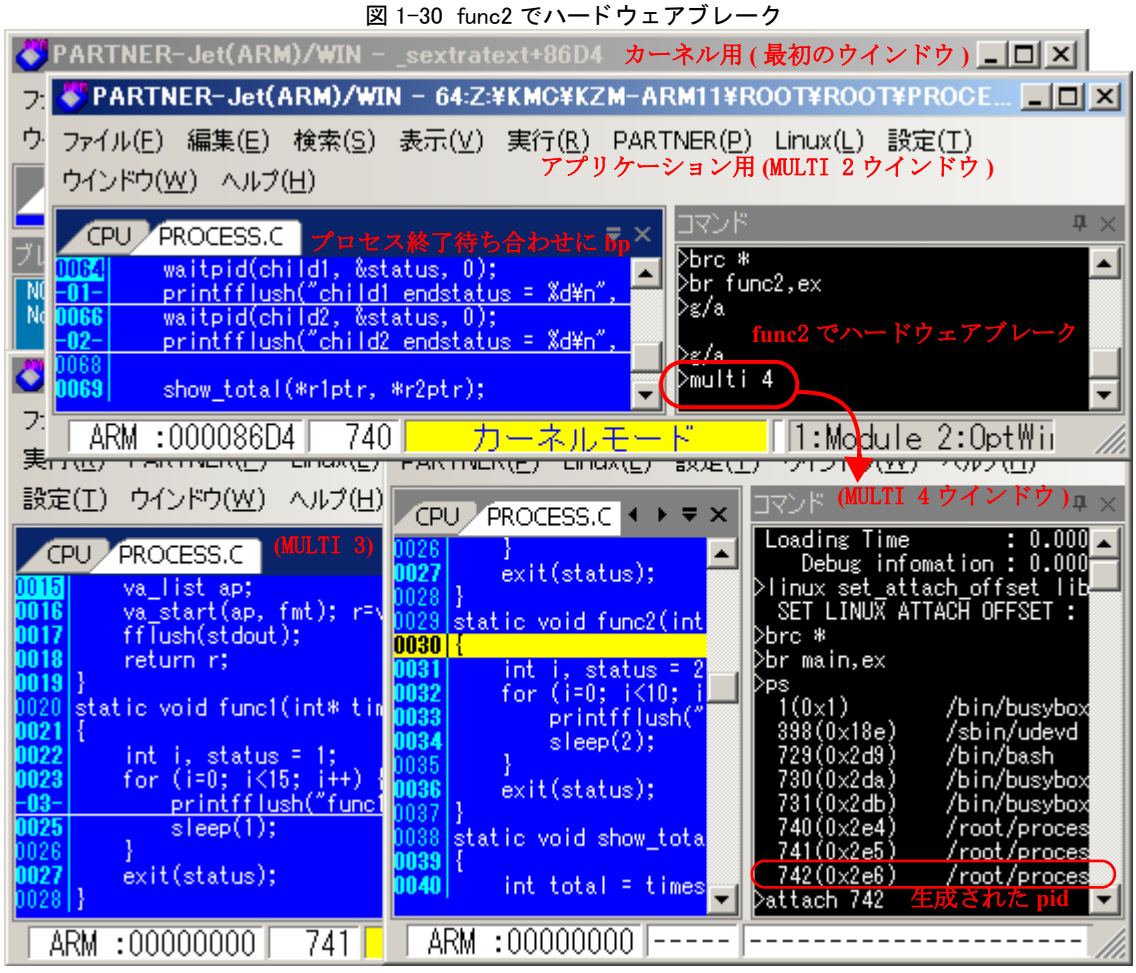


同様に func2 に対しても新しいウインドウを開いて対処します。

（MULTI2 ウインドウ）
PT2>brc * ↓
PT2>br func2.ex ↓
PT2>g/a ↓
（func2 でハードウェアブレークがかかるまで g/a する）
PT2>multi 4 ↓

（MULTI4 ウインドウ）
PT4>appls process ↓
PT4>ps ↓
1(0x1) /bin/busybox
398(0x18e) /sbin/udevd
729(0x2d9) /bin/bash
730(0x2da) /bin/busybox
731(0x2db) /bin/busybox
740(0x2e4) /root/process
741(0x2e5) /root/process
742(0x2e6) /root/process
PT4>attach 742 ↓

図 1-30 func2 でハードウェアブレーク



以後、G/A コマンドで実行するたびにアクティブなウインドウが切り替わりながらブレークポイントに停止します。

図 1-31 func1(pid 741) 内でソフトウェアブレーク

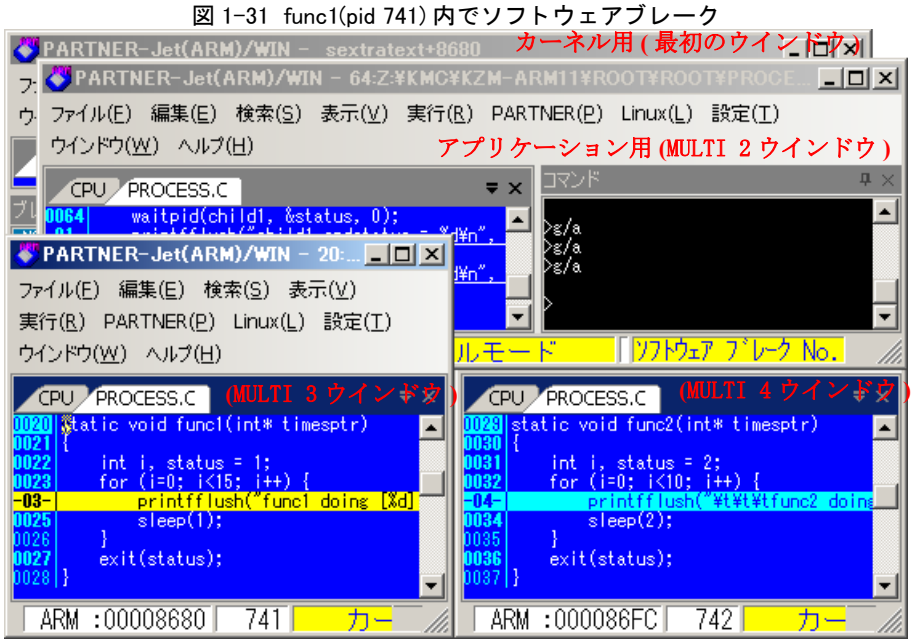


図 1-32 func2(pid 742) 内でソフトウェアブレーク

なお、G/A の代わりに G コマンドを使用すると実行されるのは、対象の PARTNER ウインドウで実行されるプロセスとカーネルだけになり、非対象の PARTNER ウインドウは表示が図 1-33 のように変化し実行されているプロセスは停止したままになります。

図 1-33 アプリケーション停止表示

1.8 共有ライブラリを作る

先のサンプル『pthread.c（21 頁)』と『process.c（26 頁)』は非常に似ているので多くの部分が共通にできます。共通部分はライブラリにしますが、同時に動く可能性のある複数のプログラムでリンクされるならば、スタティックライブラリよりも共有ライブラリの方がメモリ使用などのリソース効率が良くなりやすくなります。

以下に共有ライブラリを使うように変更したサンプルソースコードを示します。

図 1-34 samplelib.h

```
#ifndef ___SAMPLELIB_H___
#define ___SAMPLELIB_H___
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
typedef void* sample_t;
extern void sample_create(sample_t* sample_return, void (*fn)(int*), int* param);
extern void sample_exit(int status);
extern void sample_wait(sample_t* sample, int* status_return);
extern int sample_main(sample_t* sample1, int* r1ptr, sample_t* sample2, int* r2ptr);
#endif /* ___SAMPLELIB_H___ */
```

図 1-35 samplelib.c

```
#include "samplelib.h"

static int printfflush(const char* fmt, ...)
{
    int r;
    va_list ap;
    va_start(ap, fmt); r=vprintf(fmt, ap); va_end(ap);
    fflush(stdout);
    return r;
}
static void func1(int* timesptr)
{
    int i, status = 1;
    for (i=0; i<15; i++) {
        printfflush("func1 doing [%d] times.\n", ++(*timesptr));
        sleep(1);
    }
    sample_exit(status);
}
static void func2(int* timesptr)
{
    int i, status = 2;
    for (i=0; i<10; i++) {
        printfflush("\t\t\tfunc2 doing [%d] times.\n", ++(*timesptr));
        sleep(2);
    }
    sample_exit(status);
}
static void show_total(int times1, int times2)
{
    int total = times1 + times2;
    printfflush("func1=%d, func2=%d for a total of %d\n", times1, times2, total);
}
int sample_main(sample_t* sample1, int* r1ptr, sample_t* sample2, int* r2ptr)
{
    int status;

    *r1ptr=0; *r2ptr=0;

    sample_create(sample1, func1, r1ptr);
    printfflush("sample1 = 0x%08X\n", *sample1);
    sample_create(sample2, func2, r2ptr);
    printfflush("sample2 = 0x%08X\n", *sample2);

    sample_wait(sample1, &status);
    printfflush("sample1 endstatus = %d\n", status);
    sample_wait(sample2, &status);
    printfflush("sample2 endstatus = %d\n", status);

    show_total(*r1ptr, *r2ptr);
    return EXIT_SUCCESS;
}
```

共有ライブラリ samplelib を使う側は以下のようになります。

図 1-36 pthread2.c

```
#include "samplelib.h"
#include <pthread.h>

void sample_create(sample_t* sample_return, void (*fn)(int*), int* param)
{
    pthread_create((pthread_t*)sample_return, NULL, (void*)fn, (void*)param);
}
void sample_exit(int status) { pthread_exit((void*)status); }
void sample_wait(sample_t* sample, int* status_return)
{
    pthread_join((pthread_t)*sample, (void*)status_return);
}
int main(int argc, char* const argv[])
{
    pthread_t sample1, sample2;
    int rbuf[2];
    int *r1ptr, *r2ptr;
    int status;

    r1ptr = &rbuf[0];
    r2ptr = &rbuf[1];
    status = sample_main((sample_t*)&sample1, r1ptr, (sample_t*)&sample2, r2ptr);
    return status;
}
```

図 1-37 process2.c

```
#include "samplelib.h"
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h>

void sample_create(sample_t* sample_return, void (*fn)(int*), int* param)
{
    if ((*sample_return = (sample_t)fork())==0) fn(param);
}
void sample_exit(int status) { exit(status); }
void sample_wait(sample_t* sample, int* status_return)
{
    waitpid((pid_t)*sample, status_return, 0);
    *status_return = WEXITSTATUS(*status_return);
}
int main(int argc, char* const argv[])
{
    pid_t sample1, sample2;
    int shmid, *shmaddr;
    int *r1ptr, *r2ptr;
    int status;

    if ((shmid=shmget(IPC_PRIVATE, 2*sizeof(int), 0660)) == -1)
        perror("shmget"), exit(1);
    if ((shmaddr=(int*)shmat(shmid, (void*)0, 0)) == (void*)-1)
        perror("shmat"), exit(1);
    printf("shmid[%d] shmaddr[0x%08X]\n", shmid, (unsigned int)shmaddr);
    r1ptr = shmaddr;
    r2ptr = shmaddr+1;
    status = sample_main((sample_t*)&sample1, r1ptr, (sample_t*)&sample2, r2ptr);
    shmdt(shmaddr);
    return status;
}
```

共有ライブラリをデバッグオプション付きでコンパイルします。

```
LINUX86>cd /opt/kmc/kzm-arm11/root/root ↓
LINUX86>arm-linux-gcc -c -o samplelib.o -g -O0 -I../../staging_dir/include samplelib.c ↓
LINUX86>arm-linux-gcc -shared -Wl,-soname,libsample.so -o libsample.so samplelib.o -ldl ↓
LINUX86>file libsample.so ↓
libsample.so: ELF 32-bit LSB shared object, ARM, version 1 (SYSV), not stripped
```

共有ライブラリをインストールします。

```
LINUX86>$ su ↓
LINUX86># cp libsample.so /opt/kmc/kzm-arm11/root/usr/lib/ ↓
```

プログラムもデバッグオプション付きでコンパイルします。

```
LINUX86>arm-linux-gcc -g -O0 -I../../staging_dir/include -L. -lsample -lpthread pthread2.c -o pthread2 ↓
LINUX86>file pthread2 ↓
pthread2: ELF 32-bit LSB executable, ARM, version 1 (SYSV), dynamically linked (uses shared libs), not stripped
LINUX86>arm-linux-gcc -g -O0 -I../../staging_dir/include -L. -lsample process2.c -o process2 ↓
LINUX86>file process2 ↓
process2: ELF 32-bit LSB executable, ARM, version 1 (SYSV), dynamically linked (uses shared libs), not stripped
```

ターゲット上で ldd コマンドを使うとリンクしているライブラリを確認できます。

```
TGT>ldd pthread2 ↓
        libsample.so => /usr/lib/libsample.so (0x4000e000)
        libpthread.so.0 => /lib/libpthread.so.0 (0x40017000)
        libc.so.0 => /lib/libc.so.0 (0x40031000)
        libdl.so.0 => /lib/libdl.so.0 (0x40086000)
        ld-uClibc.so.0 => /lib/ld-uClibc.so.0 (0x40000000)
TGT>ldd process2 ↓
        libsample.so => /usr/lib/libsample.so (0x4000e000)
        libc.so.0 => /lib/libc.so.0 (0x40017000)
        libdl.so.0 => /lib/libdl.so.0 (0x4006c000)
        ld-uClibc.so.0 => /lib/ld-uClibc.so.0 (0x40000000)
```

pthread2 と process2 はどちらもほぼ同じ動作ですので、pthread2 プログラムを使って共有ライブラリにした libsample.so の中のデバッグを確認してみます。
main 関数でブレークするまでの手順はこれまでと同じです。

```
PT>load linux26 nfs ↓
PT>g ↓
PT>[ESC]
PT>multi 2 ↓
PT2>cd z:¥kmc¥kzm-arm11¥root¥root ↓
z:¥kmc¥kzm-arm11¥root¥root
PT2>appls pthreads2 ↓
PT2>g ↓
TGT>./pthread2 ↓ (main でブレークする)
```

この時点ではまだ共有ライブラリの情報が読み込まれていないため、[表示]→[モジュール]で表示されるファイル&関数には samplelib.c はありません。
プロセスへのアタッチ(『ATTACH コマンド (152 頁)』)と共有ライブラリの情報の解決(『LINUX コマンド (154 頁)』)によって PARTNER へ共有ライブラリのデバッグ情報が読み込まれてデバッグできるようになります。

図 1-38 pthread2 の main でブレーク時

PT2>appattach pthread ↓ (プロセスにアタッチし、共有ライブラリのデバッグ情報を読み込む)

図 1-39 pthread プログラムの起動時

共有ライブラリ内のコードにもブレークポイントの設定ができるようになりました。

1.9 アプリケーションから外部コマンドを起動する

『1.7 fork システムコールによる並行処理』では複数のコンテキストで並行処理をするためにマルチプロセスを用いました。マルチプロセスの使用目的としてはもうひとつ、別の実行可能ファイル（外部コマンド）を実行することが挙げられます。
外部コマンドをデバッグするときには、いずれかのタイミングで起動されるプログラムのデバッグ情報を読み込む必要があります。ここでは外部コマンドを実行するための 3 種類の API を例として説明します。

システムコールの fork(2)+exec(3) を使う場合

低水準 API ですがプロセス動作をきめ細かく扱えるのでよく使われていると思います。exec(3) は Linux では execve(2) を呼び出す関数群として実装されています。『図 1-40 cmd_forkexec.c』では直接使う機会は少ないと思われる execve(2) を避けて execlp 関数を使用しています。

図 1-40 cmd_forkexec.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char* const argv[])
{
        const char* cmd = SMPLCMD;
        int status, childpid;

        printf("Execute command `%s'.\n", cmd);
        fflush(stdout);
        if ((childpid = fork())==0) {
                execlp(cmd, cmd, (char*)NULL);
        }
        printf("%d fork child %d\n", getpid(), childpid);
        fflush(stdout);
        waitpid(childpid, &status, 0);
        printf("Program `%s' end with code %d.\n", cmd, WEXITSTATUS(status));
        fflush(stdout);
        return EXIT_SUCCESS;
}
```

デバッグオプション付きでコンパイルします。

```
LINUX86>cd /opt/kmc/kzm-arm11/root/root ↓
LINUX86>vi cmd_forkexec.c ↓
LINUX86>arm-linux-gcc -o cmd_forkexec -g -O0 -I../../staging_dir/include ¥
-DSMPLCMD=¥"./hello¥" cmd_forkexec.c ↓
LINUX86>file cmd_forkexec ↓
cmd_forkexec: ELF 32-bit LSB executable, ARM, version 1 (SYSV), dynamically linked
(uses shared libs), not stripped
```

起動から main 関数で停止するまでの手順はこれまでと同様です。

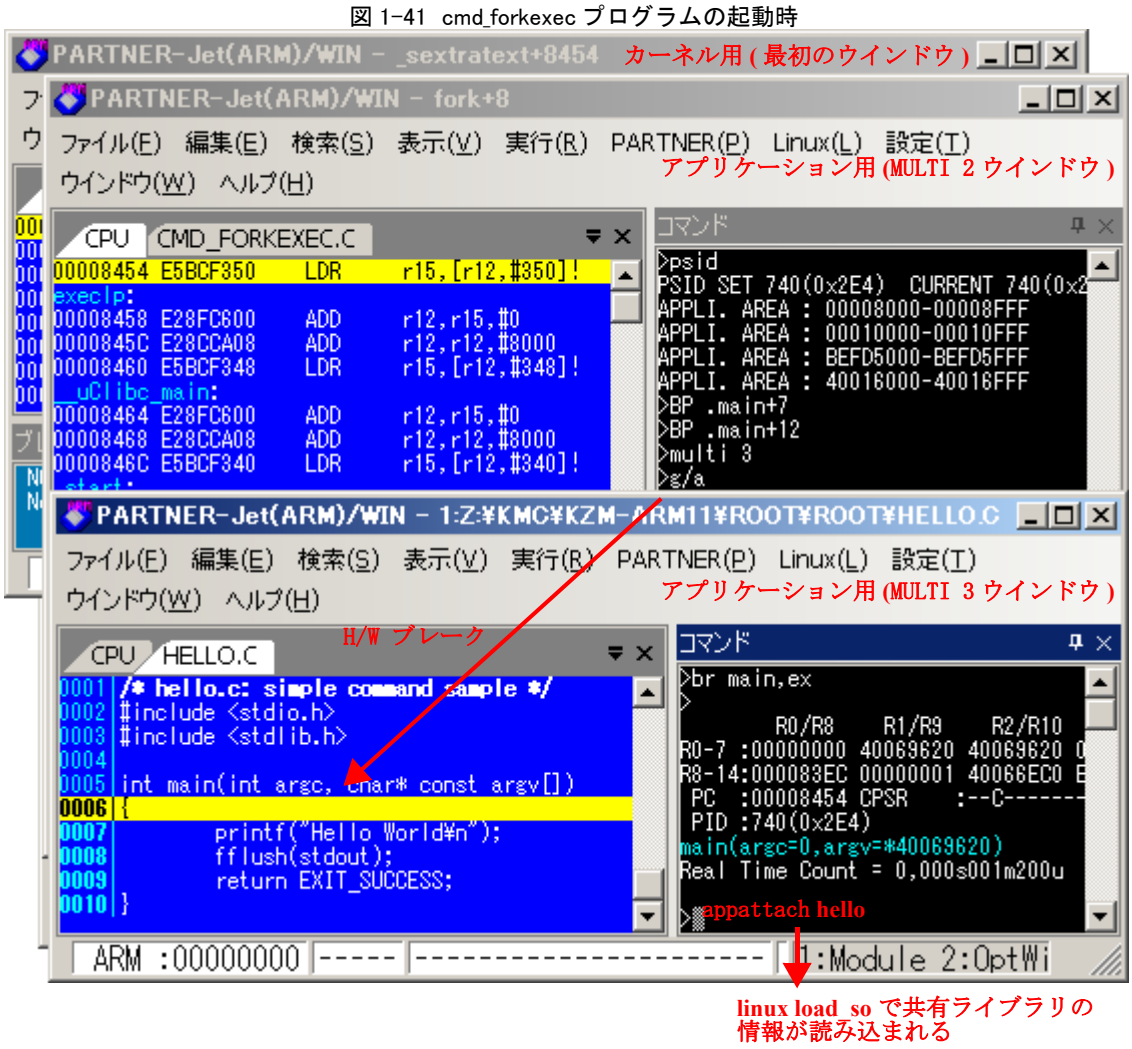
```
PT>multi 2 ↓
PT2>cd z:¥kmc¥kzm-arm11¥root¥root ↓
z:¥kmc¥kzm-arm11¥root¥root
PT2>appls cmd_forkexec ↓
PT2>g ↓
TGT>./cmd_forkexec ↓　（main でブレークする）
PT2>appattach cmd_forkexec ↓　（プロセスにアタッチ。プロセス ID で指定してもよい）
```

このサンプルプログラムよりも複雑なプログラムであっても、fork 関数を呼び出す直前では実行したい外部コマンド名が決まっていると思います。もう一枚 PARTNER のウインドウを開いて外部コマンド (./hello) のデバッグ情報を読み込みます。

(MULTI2 ウインドウ)
PT2>BP .main+7 ↓　(execlp)
PT2>BP .main+12 ↓　(waitpid の後の printf 文)
PT2>multi 3 ↓

(MULTI3 ウインドウ)
PT3>appls hello ↓
PT?>g/a ↓　(どの PARTNER ウインドウでもよい)
(hello の main でハードウエアブレーク)
PT3>appattach hello ↓　(プロセスにアタッチ。プロセス ID で指定してもよい)

図 1-41　cmd_forkexec プログラムの起動時



以降は『fork システムコールによる並行処理 (26 頁)』と同様に各ウインドウ毎にひとつのプロセスをデバッグすることができます。

POSIX の popen(3) 関数を使う場合

起動する外部コマンド（子プロセス）から標準出力への出力結果を起動元のプロセスで受け取って処理をする（フィルタ処理）ことがあります。『図 1-42 cmd_popen.c』は子プロセスからの出力をそのまま標準出力に出しています。シェルスクリプトで書くと以下の 1 行とほぼ同じ動作になります。

```
TGT>echo "Hello World" | cat -
```

なお、popen 関数で使える子プロセスとの I/O 機能では子プロセスの標準出力の読み出し又は、子プロセスの標準入力への書き込みのみの単方向なため、先の『図 1-16 hellohello.c』のように標準入力と標準出力を両方使うプログラムを起動するのには不向きです。

図 1-42 cmd_popen.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <limits.h>

#define BUFSZ        PIPE_BUF

int main(int argc, char* const argv[])
{
        FILE* fp;
        char buf[BUFSZ];
        const char* cmd = SMPLCMD;
        int status;

        printf("Execute command `%s'.\n", cmd);
        if ((fp = popen(cmd, "r")) == NULL)  {
                perror("popen");
                exit(EXIT_FAILURE);
        }
        while ((fgets(buf, BUFSZ, fp)) != NULL)
                printf("%s", buf);
        status = pclose(fp);
        printf("Program `%s' end with code %d.\n", cmd, WEXITSTATUS(status));
        fflush(stdout);
        exit(EXIT_SUCCESS);
}
```

デバッグオプション付きでコンパイルします。

```
LINUX86>cd /opt/kmc/kzm-arm11/root/root ↓
LINUX86>vi cmd_popen.c ↓
LINUX86>arm-linux-gcc -o cmd_popen -g -O0 -I../../staging_dir/include ¥
-DSMPLCMD=¥"./hello¥" cmd_popen.c ↓
LINUX86>file cmd_popen ↓
cmd_popen: ELF 32-bit LSB executable, ARM, version 1 (SYSV), dynamically linked
(uses shared libs), not stripped
```

起動から main 関数で停止するまでの手順はこれまでと同様です。

```
PT>multi 2 ↓
PT2>cd z:¥kmc¥kzm-arm11¥root¥root ↓
z:¥kmc¥kzm-arm11¥root¥root
PT2>appls cmd_popen ↓
PT2>g ↓
TGT>./cmd_popen ↓ (main でブレークする）
PT2>appattach cmd_popen ↓（プロセスにアタッチ、共有ライブラリの情報読み込み）
```

親プロセスと子プロセスはパイプの I/O を介して繋がっているので、子プロセスが大量の出力をするプログラムであれば親プロセスの処理の負荷によっては「子プロセス側が標準出力への書き込み待ちになるかもしれない」といったように、2 つのプロセスの挙動は『cmd_forkexec.c（36 頁)』と異なるはずです。
しかし、hello.c は出力がとても少ないプログラムなのでほとんど同じ動作になります。

デバッグ手順もほとんど変わりませんので、以降は手順のみ掲載します。

```
（MULTI2 ウインドウ）
PT2>BP .main+12 ↓  （fgets したときの printf 文）
PT2>BP .main+14 ↓  （pclose の後の printf 文）
PT2>multi 3 ↓

（MULTI3 ウインドウ）
PT3>appls hello ↓
PT?>g/a ↓  （どの PARTNER ウインドウでもよい）
（ハードウェアブレークで停止）
PT3>ps ↓
  1(0x1)        /bin/busybox
  398(0x18e)    /sbin/udevd
  518(0x206)    /bin/bash
  519(0x207)    /bin/busybox
  520(0x208)    /bin/busybox
  742(0x2e6)    /root/cmd_popen
  743(0x2e7)    /root/cmd_popen
（まだ hello が起動していなかった）
PT3>g ↓
（またハードウェアブレークで停止）
PT3>ps ↓
  1(0x1)        /bin/busybox
  398(0x18e)    /sbin/udevd
  518(0x206)    /bin/bash
  519(0x207)    /bin/busybox
  520(0x208)    /bin/busybox
  742(0x2e6)    /root/cmd_popen
  743(0x2e7)    /root/hello
（今度は hello が起動している）
PT3>appattach 743 ↓
PT3>BP .main+1 ↓  （適当にソフトウェアブレークポイントを設定）
PT3>BP .main+2 ↓
PT3>BP .main+3 ↓
PT?>g/a ↓  （どの PARTNER ウインドウでもよい）
```

以後 G/A コマンドを実行 するたびに MULTI2 ウインドウや MULTI3 ウインドウがアクティブになって設定しておいたソフトウェアブレークポイントに停止します。

C 標準ライブラリの system(3) 関数を使う場合

この関数はもっとも手軽なプロセス操作関数として知られていますが、実行されるのはシェルプログラムで、関数の引数はシェルに渡されるコマンド文字列だという点で先の 2 つの方式とは動作が異なります（普通は /bin/sh -c command の形式で実行される）。引数の文字列はシェルの子プロセスとして実行されることになるので（引数の文字列が exec コマンドの場合を除く）、引数の文字列をコマンドとして起動するために少なくとも 2 つのプロセスが起動されることになるからです。

図 1-43 cmd_system.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

int main(int argc, char* const argv[])
{
        const char* cmd = SMPLCMD;
        int status;

        printf("Execute command `%s'.\n", cmd);
        fflush(stdout);
        status = system(cmd);
        printf("Program `%s' end with code %d.\n", cmd, WEXITSTATUS(status));
        fflush(stdout);
        return EXIT_SUCCESS;
}
```

デバッグオプション付きでコンパイルします。

```
LINUX86>cd /opt/kmc/kzm-arm11/root/root ↓
LINUX86>vi cmd_system.c ↓
LINUX86>arm-linux-gcc -o cmd_system -g -O0 -I../../staging_dir/include ¥
-DSMPLCMD=¥"./hello¥" cmd_system.c ↓
```

デバッグ手順はこれまでと同様です。

```
PT>multi 2 ↓
PT2>cd z:¥kmc¥kzm-arm11¥root¥root ↓
z:¥kmc¥kzm-arm11¥root¥root
PT>2appls cmd_system ↓
PT2>g ↓
TGT>./cmd_system ↓ (main でブレークする）
PT2>appattach cmd_system ↓（プロセスにアタッチ、共有ライブラリの情報読み込み）
PT2>BP .main+7 ↓ （適当にソフトウェアブレークポイントを設定）
PT2>multi 3 ↓
PT3>appls hello ↓
PT?>g/a ↓ （どの PARTNER ウインドウでもよい）
（ハードウェアブレークで停止）
PT3>ps ↓ (hello のプロセス起動を確認）
  1(0x1)        /bin/busybox
  398(0x18e)    /sbin/udevd
  714(0x2ca)    /bin/bash
  715(0x2cb)    /bin/busybox
  716(0x2cc)    /bin/busybox
  740(0x2e4)    /root/cmd_system
  741(0x2e5)    /root/hello
PT3>appattach 741 ↓
```

以降は両プロセスをデバッグ可能です。

1.10 ローダブルモジュールを作る

Linux は多数のデバイスをサポートしていますが、組み込み機器の開発ではデバイスドライバを作成することがあると思います。
デバイスドライバは Linux のカーネル空間で動作するため、JTAG/ICE デバッガにとってはユーザー空間のアプリケーションよりもデバッグしやすい対象です。問題があるとすればローダブルモジュール形式でシステム起動後に動的にカーネルに組み込まれる場合です。
ローダブルモジュールのカーネルへの組み込みはターゲット上の insmod コマンドによって行いますが、組み込まれるまではアドレスが確定していません。デバッガにとってはモジュール初期化時に呼び出される module_init 関数からデバッグを開始するところに困難さがあります。
PARTNER では Linux カーネルをコンフィグレーションするときに有効にした [PARTNER Debugging] メニューの [Loadable module auto attach]、[Loadable module auto attach (patch is include in kernel)] の項目が効力を発揮し、とても簡単な手順でデバッグをすることができます。

Linux のローダブルモジュール形式は 2.4 系と 2.6 系で異なっています。本節の手法は 2.6 系カーネルでのみ有効です。

図 1-44 kzmled.c はローダブルモジュールの最もシンプルなインタフェース module_init と module_exit のみを持ったプログラムです。
ローダブルモジュールがカーネル空間で実行されて、ハードウェアに直接アクセスできることをわかりやすく示すために、モジュールがロードされるときとアンロードされるときに KZM-ARM11-01 ボードのオンボード LED を光らせます。

図 1-44 kzmled.c を KZM-ARM11-01 ボード以外の環境でコンパイルする場合は LED へのアクセス部分は無効です。printk によるメッセージ出力のみが行われます。

図 1-44 kzmled.c

```
#include <linux/config.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/ioport.h>
#include <linux/delay.h>

#ifdef CONFIG_MACH_MX3KZ
#include <asm-arm/arch-mxc/board-mx31kz.h>
#define KZM_LED7SEG_ADDR    (KZCTL_BASE_ADDRESS + KZCTL_7SEG_LED)
#define KZM_LED4BIT_ADDR    (KZCTL_BASE_ADDRESS + KZCTL_LED)
#endif /* End of CONFIG_MACH_MX3KZ */

#ifdef CONFIG_MACH_MX3KZ
MODULE_DESCRIPTION("KZM board 7 segments LED module");
#else
MODULE_DESCRIPTION("Simple kernel module");
#endif /* End of CONFIG_MACH_MX3KZ */
MODULE_AUTHOR("KMC");
MODULE_LICENSE("GPL");

#ifdef CONFIG_MACH_MX3KZ
static int kzmled7seg(int n, int dot)
{
    n += (dot) ? 0x10 : 0;
    outb(n, KZM_LED7SEG_ADDR);
    return 0;
}
static int kzmled4bit(int n)
{
    outb(n, KZM_LED4BIT_ADDR);
    return 0;
}
static void ledwait(void)
{
    volatile int i;
    for (i=0; i<10; i++) msleep(100);
}
#endif /* End of CONFIG_MACH_MX3KZ */

/* insmod */ static int kzmled7seg_init_module(void)
{
    int error = 0;

    printk("LED module is loaded.\n");
#ifdef CONFIG_MACH_MX3KZ
    printk("7 segments LED addr: 0x%X\n", KZM_LED7SEG_ADDR);
    printk("4 dots LED addr    : 0x%X\n", KZM_LED4BIT_ADDR);
    { int i; for (i=0; i<=0xF; i++) {
        if ((error = kzmled7seg(i, 1))) break;
        if ((error = kzmled4bit(i))) break;
        ledwait();
    } }
#endif /* End of CONFIG_MACH_MX3KZ */

    return (error) ? -ENODEV : 0;
}

/* rmmod */ static void kzmled7seg_cleanup_module(void)
{
#ifdef CONFIG_MACH_MX3KZ
    int error =0;
    { int i; for (i=0xF; i>=0; i--) {
        if ((error = kzmled7seg(i, 0))) break;
        if ((error = kzmled4bit(i))) break;
        ledwait();
    } }
#endif /* End of CONFIG_MACH_MX3KZ */
    printk("Simple LED module is unloaded.\n");
}

module_init(kzmled7seg_init_module);
module_exit(kzmled7seg_cleanup_module);
```

このソースファイルをコンパイルするのですが、Linux カーネル 2.6 系のローダブルモジュール形式 .ko を作るためには、Linux カーネルのソースツリーのコンパイル設定を利用するため、図 1-45 のような Makefile を作成します。

図 1-45 Makefile

```
KSRCDIR := /opt/kmc/kzm-arm11/build_src/linux
MAKEDIR := $(shell pwd)
MODNAME := kzmled
MODSRCS := kzmled.c

obj-m := $(MODNAME).o
ifneq ($(MODNAME),$(MODSRCS:.c=))
  $(MODNAME)-objs := $(MODSRCS:.c=.o)
endif

all:
        make -C $(KSRCDIR) SUBDIRS=$(MAKEDIR) KBUILD_VERBOSE=0 modules

clean:
        rm -f *.o *.ko *.mod.c *.map .*.cmd *~
        rm -rf .tmp_versions
```

ターゲット上の root ユーザのホームディレクトリ以下で作業するものとします。

```
LINUX86>cd /opt/kmc/kzm-arm11/root/root ↓
LINUX86>mkdir tutrial_kmod26 ↓
LINUX86>cd tutrial_kmod26 ↓
LINUX86>vi kzmled.c Makefile ↓
(kzmled.c と Makefile を作成)
```

コンパイルします。

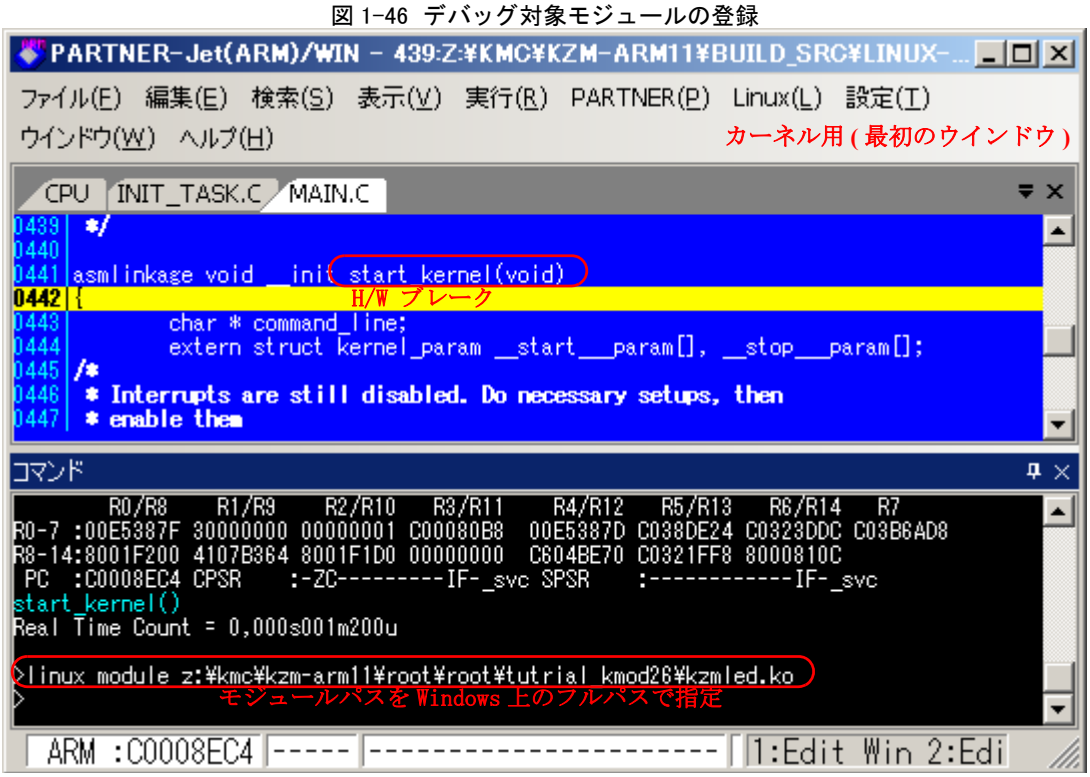
```
LINUX86>cd /opt/kmc/kzm-arm11/root/root/tutrial_kmod26 ↓
LINUX86>make ↓
make -C /opt/kmc/kzm-arm11/build_src/linux SUBDIRS=/opt/kmc/kzm-arm11/root/root/pt-debug-
linux/tutrial_kmod26 KBUILD_VERBOSE=0 modules
make[1]: ディレクトリ `/opt/kmc/kzm-arm11/build_src/linux' に入ります
  CC [M]  /opt/kmc/kzm-arm11/root/root/tutrial_kmod26/kzmled.o
  Building modules, stage 2.
  MODPOST
  CC      /opt/kmc/kzm-arm11/root/root/tutrial_kmod26/kzmled.mod.o
  LD [M]  /opt/kmc/kzm-arm11/root/root/tutrial_kmod26/kzmled.ko
make[1]: ディレクトリ `/opt/kmc/kzm-arm11/build_src/linux' から出ます
LINUX86> ls ↓
Makefile  kzmled.c  kzmled.ko  kzmled.mod.c  kzmled.mod.o  kzmled.o
(kzmled.ko ができています)
```

Linux カーネルソースツリーのコンパイル設定を利用するため、デバッグ情報をつけてビルドするかどうかはカーネルソースと同じになります。

では、実際にデバッグする手順を説明します。
ローダブルモジュールのデバッグのために LINUX コマンド（154 頁）を使用して、あらかじめ PARTNER にデバッグ対象のモジュールを登録しておく操作が必要なところが、他のプログラムのデバッグと異なります。

PT>load_linux26_nfs ↓
PT>g ↓
（start_kernel でブレークする）
PT>linux module z:¥kmc¥kzm-arm11¥root¥root¥tutrial_kmod26¥kzmled.ko ↓
（デバッグ対象のモジュールを登録）
PT>g ↓

図 1-46 デバッグ対象モジュールの登録



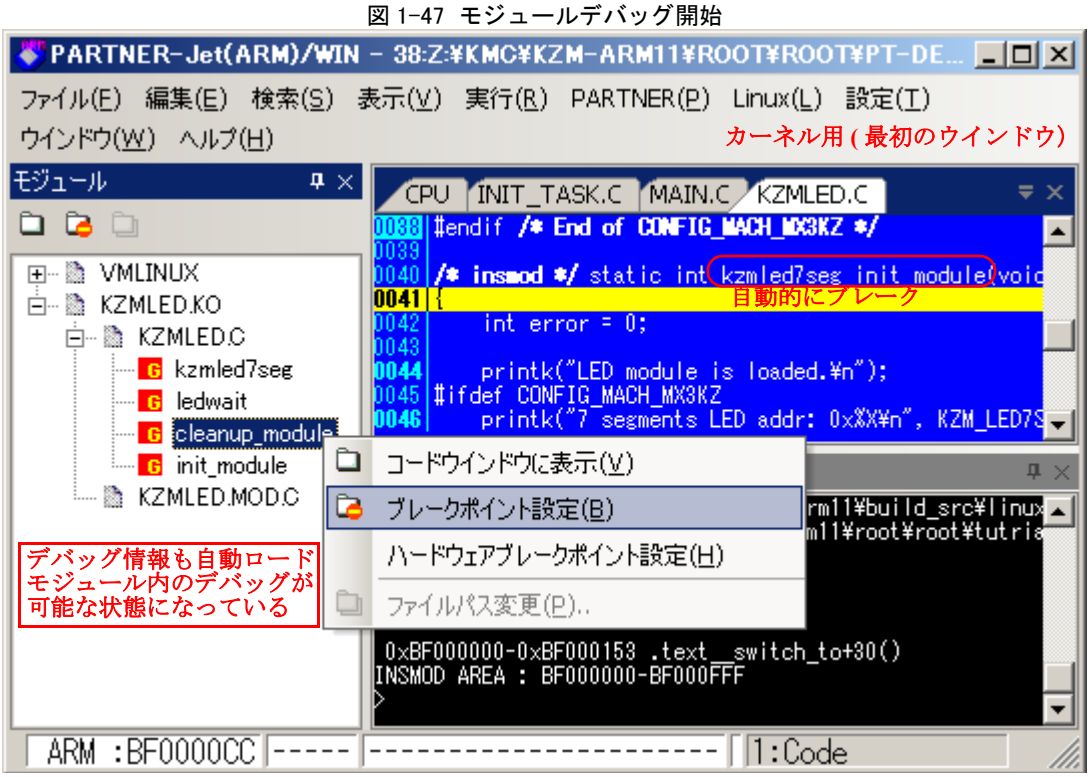
「linux module < パス名 >」の操作はローダブルモジュールが insmod される前ならばいつでもかまいませんが、Linux カーネルが有効になった後（start_kernel シンボル以降）である必要があります。
デバッグ対象モジュールの登録は「linux module」コマンドで確認したり、「linux module clr」で削除したりできます。
詳しくは、『LINUX コマンド（154 頁）』を参照してください。

ローダブルモジュールを作る

Linux が起動したら、root ユーザで insmod コマンドによってローダブルモジュールをカーネルに組み込みます。

```
TGT>login: root ↓
TGT># cd tutrial_kmod26 ↓
TGT># insmod kzmled.ko ↓
（自動的に PARTNER でブレークがかかります）
```

図 1-47 モジュールデバッグ開始



以後は自由にソフトウェアブレークポイントの設定が可能です。
例えば、cleanup_module にブレークポイントを設定しておくとターゲット上でモジュールのアンロードを行ったときにブレークポイントで停止します。

```
TGT># rmmod kzmled ↓
```

第 2 章　Linux デバッグと環境設定

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設定について説明します。

2.1 Linux デバッグのための設定概要

PARTNER は快適な Linux のデバッグを環境を提供するために、積極的に Linux カーネル内の情報を活用します。そのため、PARTNER は Linux カーネルのデバッグ情報を必要とするほか、いくつか設定が必要になります。
この節では、高度なデバッグを行うための設定の概要を説明します。

2.1.1 Linux デバッグのための設定種別

表 2-1 に Linux デバッグのための設定の種別一覧を掲載します。

表 2-1 Linux デバッグのための設定種別一覧

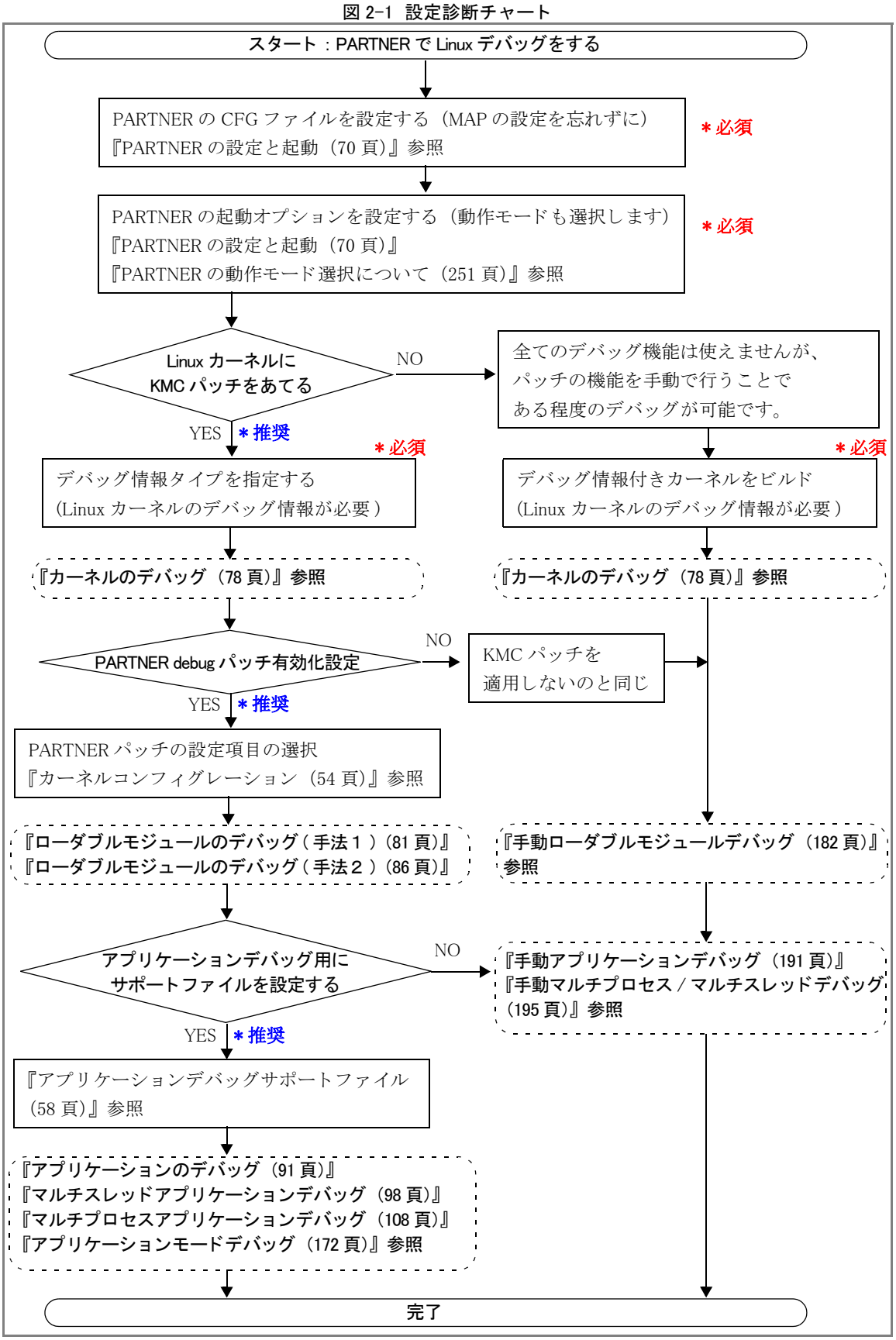
種別	内容
カーネル修正	デバッグを便利にする機能をカーネル内に組み込みます。 KMC よりパッチとして提供されて、各項目の使用・不使用を簡単に切り替えられます。 『Linux カーネルソースの修正と設定（50 頁）』参照。
アプリケーション修正	ユーザーモードのプログラム（アプリケーション）を便利にデバッグするための機能をアプリケーションに組み込みます。 KMC よりアプリケーションデバッグサポートファイルとして提供されます。アプリケーションへの組み込み方法は『アプリケーションデバッグサポートファイル（58 頁）』参照。
PARTNER 設定	PARTNER がターゲットボードを扱うための設定を記述します。Linux のデバッグのために書式が拡張されており、カーネルとユーザーモードプログラムの関係を PARTNER が認識します。 『CFG ファイルの拡張（132 頁）』参照。
PARTNER 起動オプション	ビルド環境とターゲット上でのパス名の違い（デバッグ情報内のパス名の変換）やターゲット上の Linux カーネルバージョンや PARTNER の動作モードなどを指定します。 『起動オプション（137 頁）』参照。

その他設定ではありませんが Linux でのデバッグに特化した機能として、デバッグ中の操作として使用できる PARTNER のコマンドが Linux 用に拡張されています（『追加コマンド（149 頁）』参照）。

2.1.2 設定診断

より快適なデバッグ環境とするにはどんな設定が必要なのか、あるいは設定やデバッグ用の変更を少なくした場合何ができるのか、選択肢をチャートで図 2-1 に示します。

図 2-1 設定診断チャート



2.2 Linux カーネルソースの修正と設定

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。

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。
修正内容は追加ディレクトリ（KMC/）とパッチファイルによって構成されます。
以下、Linux カーネルを展開したディレクトリを $(TOPDIR)/ と表記します。

2.2.1 Linux カーネルソース修正の必要性

PARTNER で Linux のデバッグを行うには、必ず Linux カーネルのデバッグ情報が必要です。

カーネル空間のデバッグが必要無く、ユーザーモードのアプリケーションだけをデバッグするときでも Linux カーネルのデバッグ情報が必要なことに注意してください。

通常は Linux カーネルのソースツリーにはデバッグ情報の付加設定がされていないのが一般的です。そのため、PARTNER 用にカーネルのコンフィグレーションメニューにデバッグ情報設定を追加するパッチを提供しています。
さらに、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るようになります。

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の詳細については、『5.1 アプリケーションモードデバッグ（172 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。

2.2.2 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ファイルを展開します。

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

・追加 $(TOPDIR)/KMC/Kconfig_kmc

・追加 $(TOPDIR)/KMC/Makefile_kmc

・追加 $(TOPDIR)/KMC/Rules_kmc.make

・追加 $(TOPDIR)/KMC/__brk_code.h

・追加 $(TOPDIR)/KMC/config_kmc.in

・追加 $(TOPDIR)/KMC/kmc.c

・追加 $(TOPDIR)/KMC/kmc.h

・追加 $(TOPDIR)/KMC/kmc_dt_am33.c

・追加 $(TOPDIR)/KMC/kmc_dt_arm.c

・追加 $(TOPDIR)/KMC/kmc_dt_mips.c

・追加 $(TOPDIR)/KMC/kmc_dt_sh.c

これらのファイルは PARTNER がソフトウェアブレークを使用するためのコードやデバッグ情報の種別の定義が入っています。

2.2.3 修正ファイルリスト

修正ファイルはパッチファイルで提供されます。
以下のようにしてパッチを Linux カーネルソースに適用します。

```
LINUX86>ls linux-* kmc_modify.patch ↓
linux-2.X.XX_kmc_modify.patch
LINUX86>cd $(TOPDIR) ↓
LINUX86>patch -p1 < ../linux-2.X.XX_kmc_modify.patch ↓
LINUX86>make menuconfig ↓
```

パッチファイルは Linux カーネルの代表的なバージョンに対して提供されます。(全てのバージョンではありません)

カーネルのバージョンによって修正されるファイルは異なります。たとえば 2.4.20 の場合は以下のファイルが修正されます。

Linux カーネル 2.4.20 の修正ファイルリスト例：

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

2.2.4 MIPS シリーズの注意事項

MIPS CPU をご使用の方は以下の箇所のパッチの適用にご注意ください。
これはシステムコール sys_gettid() を使用可能にするための修正です。
$(TOPDIR)/arch/mips/kernel/syscalls.h に SYS(sys_gettid,0) の記述があるかどうかをパッチ適用前にご確認ください。

修正 $(TOPDIR)/arch/<cpu>/kernel/syscalls.h

```
    SYS(sys_pivot_root, 2)
    SYS(sys_mincore, 3)
    SYS(sys_madvise, 3)
    SYS(sys_getdents64, 3)
    SYS(sys_fcntl64, 3)         /* 4220 */
   +SYS(sys_ni_syscall, 0)
   +SYS(sys_gettid, 0)
```

システムコール sys_gettid() は、デバッグサポートライブラリファイル (kmc-support.c) で使用されています。arch/mips/kernel/syscalls.h 内に、SYS(sys_gettid,0) の宣言がある場合は、この修正を行う必要はありません。
SYS(sys_gettid,0) の宣言がない場合は、include/asm-mips/unistd.h の _NR_xxxx 宣言の順番に合わせて sys_gettid のところまで記述してください。

2.2.5 手修正する場合の注意事項

パッチファイルが提供されていないバージョンの Linux カーネルをご使用の場合、Linux カーネルソースを手修正することになります。修正する内容は、もっとも近いバージョン用のパッチファイルを元にするのが良いと思われますが、注意が必要な箇所もあります。

修正 $(TOPDIR)/kernel/sched.c

_KMC_SCHED_CALL(prev,next) の埋め込みをします。必ず switch_to(prev,next,prev); を実行する直前に挿入してください（Linux 2.4 系、2.6 系どちらの場合でも必要です）。

sched.c 内の修正箇所は、カーネルバージョンによって switch_to() 関数の記述箇所が大きく違います。注意してください。

2.2.6 カーネルコンフィグレーション

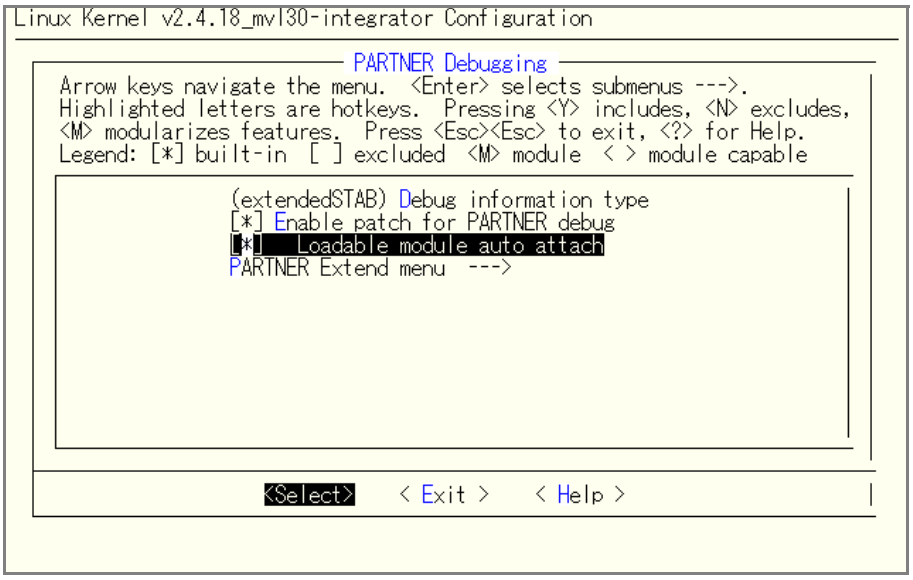
Linux カーネルソースツリーに修正を施すと、カーネルコンフィグレーションメニューに [PARTNER Debugging] が追加されます。ここでは、PARTNER でのデバッグに関する項目が設定できるようになります。

```
LINUX86>make menuconfig ↓
```

【Linux2.4 系カーネルの場合】

```
[Kernel hacking]
└→ [PARTNER Debugging]
      ├──→ [Debug information type]
      └┬→ [Enable patch for PARTNER debug]
       ├──→ [Loadable module debug by PARTNER-Jet]
       |        - (debug hook in module side)
       ├──→ [enable PARNTER-Jet Event Tracker]
       └──→ [PARTNER Extend menu]
```

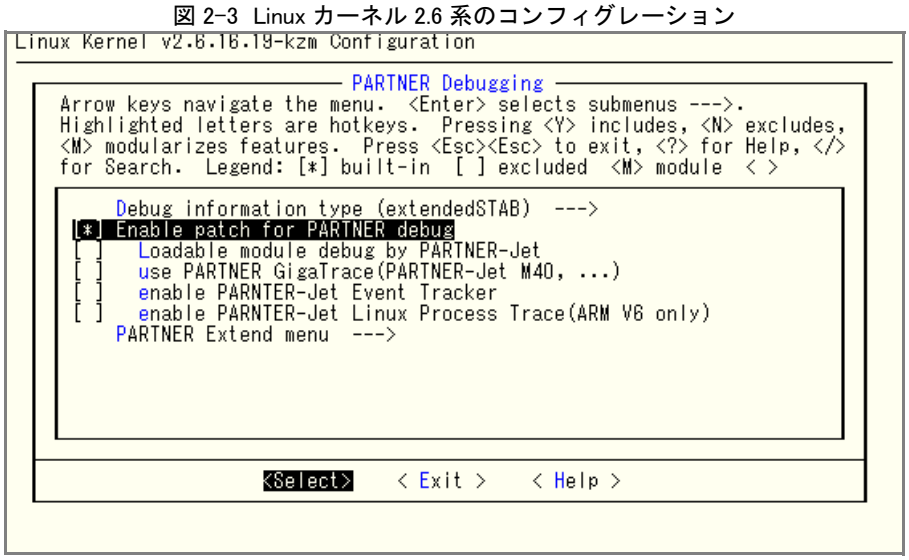
図 2-2 Linux カーネル 2.4 系のコンフィグレーション



【Linux2.6 系カーネルの場合】

[PARTNER Debugging]
├──→ [Debug information type]
└┬──→ [Enable patch for PARTNER debug]
├──→ [Loadable module debug by PARTNER-Jet]
│ select module debug type
│ - (debug hook in kernel side)
│ - (debug hook in module side)
├──→ [use PARTNER GigaTrace(PARTNER-Jet M40, ...)]
├──→ [enable PARNTER-Jet Event Tracker]
├──→ [enable PARNTER-Jet Linux Process Trace(ARM V6 only)]
└──→ [PARTNER Extend menu]

図 2-3 Linux カーネル 2.6 系の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。
推奨は extendedSTAB(-gstabs+) です。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。
本項目を無効にするとカーネルにパッチを当てていない状態と同じになります。

● Loadable module debug by PARTNER-Jet

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。
2.4 系カーネルの場合は (debug hook in module side) しか選べませんが、2.6 系カーネルの場合は (debug hook in module side) と (debug hook in kernel side) で選択可能で、(debug hook in kernel side) が推奨設定です。
詳しくは、『3.2 ローダブルモジュールのデバッグ (手法１) (81 頁)』『3.3 ローダブルモジュールのデバッグ (手法２) (86 頁)』を参照してください。

● use PARTNER GigaTrace(PARTNER-Jet M40, ...)

ギガトレース機能を使うための設定です。PARTNER-Jet Model 40 等、ギガトレース機能対応 ICE を必要とします。詳しくはギガトレース機能対応機器で別途提供されるマニュアルを参照してください。

● enable PARNTER-Jet Event Tracker

PARTNER のイベントトラッカー機能を使ってプロセスやスレッドの実行履歴を可視化するために必要な設定です。詳しくは『第 6 章 イベントトラッカー（203 頁)』を参照してください。

● enable PARNTER-Jet Linux Process Trace(ARM V6 only)

この項目は V6 命令をサポートする ARM CPU の場合しか表示されません。通常はそのままの設定をお使いください。

● PARTNER Extend menu

特殊な項目ですので、通常は変更しないでください。

これらの設定項目の適用範囲をまとめると表 2-2 のようになります。
◎は有効設定が必須で、空欄の設定はどちらでもかまいません。表内の項目は常に全て有効に設定しておいても特に問題はありません。

表 2-2 デバッグ用コンフィグレーションの適用範囲

設定項目 / デバッグ対象	Debug infomation type	Enable patch for PARTNER debug	Loadable module debug (hook in kernel side)	Loadable module debug (hook in module side)	use PARTNER GigaTrace (PARTNER-Jet M40, ...)	enable PARNTER-Jet Event Tracker
カーネルデバッグ	◎					
モジュールデバッグ（手法１）	◎	◎	◎	—		
モジュールデバッグ（手法２）	◎	◎	—	◎		
アプリケーションデバッグ	◎	◎				

2.2.7 カーネル修正による動作への影響について

Linux カーネルソースツリーへの修正内容は、全てカーネルコンフィグレーションメニューの [PARTNER Debugging] によって一括で ON/OFF の切り替えができます。Linux カーネルソースツリーから適用した修正内容を明示的に削除しなくてもコンフィグレーションをし直してカーネルをリビルドするだけでリリース版を作成することができます。

カーネルコンフィグレーションメニューの [PARTNER Debugging] で PARTNER を使用したデバッグ用設定を有効にしたカーネルで、JTAG デバッガを未接続なまま使用しても動作に影響はありません。
ただし、コンテキストスイッチをするときとプロセスが終了するときにごくわずかのオーバーヘッドは発生します。

2.3 アプリケーションデバッグサポートファイル

この節では、高度なデバッグを行うためのアプリケーションデバッグサポートファイルについて説明します。

2.3.1 デバッグサポートファイルの必要性

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。プロセスが動作開始する仮想アドレスはほとんどのプロセスで同じになります。

AM33 CPU を使用する場合では、**0x08000000 付近**
ARM CPU を使用する場合では、**0x00008000 付近**
MIPS/SH CPU を使用する場合では、**0x00400000 付近**

つまりプロセスの先頭アドレスは CPU アーキテクチャ毎に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。

アプリケーションデバッグ機能を完全に使うためには、プロセスの仮想アドレスを解決するためにデバッグサポートファイル（kmc-support.c）によるスタブ関数埋め込みが必要になります。ターゲット環境にアプリケーションサポート用のスタブがあることで、PARTNER はアプリケーションの仮想アドレス空間を自動的に解決します。
デバッグサポートファイルをどのように使用するか方法がいくつかあります。

(1) デバッグサポートファイル（kmc-support.c）をアプリケーションに直接リンクした上で、アプリケーションソースコード内にデバッグスタブ関数（_kmc_start()）を埋め込む（旧方式）。またはサポートファイルを共有ライブラリにした上でアプリケーションからリンクする。

(2) 殆どのプログラムがリンクしているライブラリ（libc など）にデバッグサポートファイル（kmc-support.c）をリンクしておき、アタッチする

(3) デバッグサポートファイル（kmc-support.c）を Preload ライブラリ（libkmcsup.so）として使用し、アタッチする（新方式）

それぞれ一長一短がありますが、比較すると表 2-3 のようになります。

表 2-3 デバッグサポートファイルリンク方式比較

リンク方法	形式	利点・難点	デバッグ開始
(1) アプリケーションに直接リンク	.o	○修正の影響範囲はアプリケーション単体で閉じている × main() やスレッドの先頭にスタブ関数を埋め込む必要がある ×アプリケーションのリンク設定を変更する必要がある ×スタブ関数を埋め込んだプログラムを JTAG デバッガ非接続状態で起動すると例外が起こる	スタブ関数でブレーク
(1)' 共有ライブラリをアプリケーションに直接リンク	.so		スタブ関数でブレーク
(2) libc にリンク	.o	○個別のアプリケーションを修正する必要がない ○スレッドの開始も自動フック × libc として使われるライブラリ（glibc や uclibc）は複雑で、パッチを入れることが難しい △システム全体に影響するライブラリにデバッグスタブを入れることに抵抗を感じる人もいる	プロセスにアタッチ
(3) Preload ライブラリ	.so	○個別のアプリケーションを修正する必要がない ○リリース版ファイルシステム上にライブラリが入っていても問題がない ○ LD_PRELOAD 環境変数の設定だけで OFF にできる ○スレッドの開始も自動フック ×プログラム開始時のブレーク（br main,ex）で別のプロセスに止まることがある △ Preload 設定を忘れるとデバッグできない	プロセスにアタッチ

新方式の (3)Preload ライブラリにも問題はありますが、(1) アプリケーションに直接リンクする方式と比べると、スレッドの生成箇所にスタブ関数（_kmc_start()）を埋め込む必要が無いので、特に共有ライブラリ内で生成されるスレッドのデバッグが楽になる点で明白なアドバンテージがあります。
デバッグ時の快適さでは libc にリンクする方式 (2) が有利ですが、(3)Preload ライブラリ方式はデバッグする時としない時では、環境変数 LD_PRELOAD の設定を変えるだけなので導入・削除の容易さで利点があります。そこで、(3)Preload ライブラリ方式を推奨方式とします。

なお、スタブ関数は (2) や (3) の場合でも併用が可能です。スタブ関数を「実行コンテキストで 1 度だけ自動的にブレークがかかる関数」と考えれば開発時にはいつでも有用だと考えられます。スタブ関数の使い方から使用方法を整理すると表 2-4 のようになります。

表 2-4 デバッグサポートファイル使用方法

	スタブ関数	リンク方法
①	なし	(2) libc にリンク (3) libkmcsup.so を Preload
②	main 等の任意の箇所に挿入して良いか	(1) kmc-support.c をリンク (1)' libkmcsup.so をリンク (2) libc にリンク (3) libkmcsup.so を Preload
③	スレッド、子プロセスに挿入が必要か	(1) kmc-support.c をリンク

ここでは、推奨の (3)Preload ライブラリ方式とアプリケーション単体で修正範囲が閉じている (1) アプリケーション直接リンク方式について使用方法を説明します。

2.3.2 Preload ライブラリ方式

Preload ライブラリ方式の場合はデバッグサポートファイルを共有ライブラリ形式でビルドし、ターゲットにインストールしたうえで、Preload ライブラリとしての設定を行います。

ライブラリの作成

アプリケーションデバッグのサポート用ライブラリはソースコードで提供され、二つのファイルで構成されます。

Makefile
kmc-support.c

以下の手順でライブラリを作成します。

(1) コンパイル

クロスコンパイラ（ARCH 変数）と CPU 種別（CPU 変数）を指定して make コマンドを実行するように Makefile が記述されています。もし起動されるコンパイラを変えたい場合は、Makefile を編集し、ターゲット 用の gcc コンパイラを CC に指定します。
コンパイルに成功すると libkmcsup.so と libkmcsup.so.2.0.0 の 2 つのファイルができます（libkmcsup.so は libkmcsup.so.2.0.0 へのシンボリックリンクです）。

【ARM シリーズ ARM11 CPU の例】

```
LINUX86> tar zxvf libkmcsup.tgz ↓
LINUX86> cd libkmcsup ↓
LINUX86> make ↓
クロスコンパイラを指定してください。ARCH={arm,mips,sh}
make: *** [envchk] エラー 1
LINUX86> make ARCH=arm ↓
CPU を指定してください。CPU={arm7,arm9,arm11}
make: *** [envchk] エラー 1
LINUX86> make ARCH=arm CPU=arm11 ↓
arm-linux-gcc -c -gstabs+ -O0 -DCPUTYPE_ARM11 -D__KMC_DYNAMIC_SUPPORT
-fPIC -o kmc-support.od kmc-support.c
arm-linux-gcc -shared -Wl,-soname,libkmcsup.so -o libkmcsup.so.2.0.0
kmc-support.od -ldl
rm -f libkmcsup.so
ln -fs libkmcsup.so.2.0.0 libkmcsup.so
---- PARTNER COMMANDLINE ------------------------------
linux set_attach_offset libkmcsup.so.2.0.0 0x00000624
-------------------------------------------------------
LINUX86>
```

(2) アドレス情報の控え

表示される PARTNER COMMANDLINE の次の 1 行は、サポートライブラリに含まれるシンボル _kmc_sleep_thread のオフセットアドレス情報です。
控えておいて、Linux カーネルが起動した後で PARTNER のコマンドとして入力してください。

デバッグサポートライブラリのインストールと設定

以下の手順でデバッグサポートライブラリをターゲットで使用できるように設定します。

(1) ターゲットへのインストール

ターゲット Linux のルートファイルシステムに、生成された二つのファイルをコピーします。ディレクトリはどこでも構いません。

(2) ターゲットの設定

このライブラリは、ld.so の Preload 機能を使ってプロセスにマッピングします。Preload 機能により、アプリケーション実行時に最初にロードされることでデバッガとの連携機能が働きます。したがって、ライブラリの Preload の指定を行う必要があります。

設定は、下記のいずれかの方法で行います。詳細は ld.so のマニュアルページでご確認ください。

1. /etc/ld.so.preload ファイルに設定
   /etc に ld.so.preload というファイルを作成し、そこに Preload させたい共有ライブラリ（libkmcsup.so.2.0.0）のフルパスを記述します。
   必要であれば ldconfig コマンドを使用してライブラリのキャッシュを更新します。

2. 環境変数 LD_PRELOAD に設定
   スペース区切りでライブラリのパスを指定します。環境変数が設定された個別のプロセスのみが対象になります。システム上のプログラム全体に適用するには /etc/profile 等を書き換えて、プログラムを起動するシェルの環境変数に登録します。

Linux 環境で動作に違いがありますので、『Linuix 互換性についての注意（67 頁）』も参照してください。

コラム 2-4 デバッグサポートファイル内の機能の活用

デバッグサポートファイル内で定義されているデバッグスタブ関数（詳細は『アプリケーション直接リンク方式（62 頁）』を参照）はデバッグサポートファイルをライブラリとしてビルドする場合でも利用可能です。
デバッグスタブ関数をアプリケーションに挿入すると、ハードウェアブレークポイントを使用しなくてもデバッグスタブ関数の直後で自動的にブレークします。プロセス / スレッド内で一度だけブレークするポイントとして利用することができます。
また、デバッグサポートファイル内で定義されている _KMC_BRK_CODE_() マクロを利用するとプログラム内にブレークポイントを埋め込むことができます（ブレーク後にプログラムの実行を継続するにはプログラムカウンタの値を進める必要があります）。
どちらもアプリケーションのソースコードをデバッグ用に修正することになりますが、開発時のテクニックとしてご活用ください。

2.3.3 アプリケーション直接リンク方式

デバッグサポートファイルをアプリケーションに直接リンクする場合は、以下の手順でデバッグサポートファイルをデバッグ対象アプリケーションに組み込みます。

アプリケーションの修正とビルド

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。

図 2-5 デバッグスタブ関数

```
__kmc_start(char *program_name);
または
__kmc_start_debuger(char *program_name);
```

デバッグスタブ関数の引数 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。

デバッグスタブ関数をアプリケーションに挿入すると、ハードウェアブレークポイントを使用しなくても _kmc_start() の直後で自動的にブレークします。
デバッグスタブ関数名 _kmc_start() は以前のバージョンでは _kmc_start_debuger() という名前でした。下方互換性を保つために旧関数名も残されています。なお、debuger の g は 1 文字です。

【例】 main 関数の直後へ埋め込み

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

マルチスレッドのプログラム (pthreads ライブラリ使用) の各スレッドをデバッグするときは、各スレッドのエントリー関数の先頭にもデバッグスタブ関数を挿入します。スレッドのエントリー関数の先頭に挿入するデバッグスタブ関数の引数 program_name には、0 を指定してください。

【例】 マルチスレッドプログラムへ埋め込み

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

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()) ではブレークしますが、スレッドのエントリー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。main() 関数で停止中に 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。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。

fork() システムコールを使うマルチプロセスのプログラムの各子プロセスをデバッグするときは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。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 program_name には、0 を指定してください。

【例】 マルチプロセスプログラムへ埋め込み

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

(2) アプリケーションへのリンク

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する場合は、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。

コラム 2-1　ソースファイルを変更せずにアプリケーションを修正する Tips

デバッグサポートファイルをアプリケーションに直接リンクする方式ではデバッグスタブ関数を埋め込まなくてはなりませんが、C 言語のマクロを使用すれば必ずしもアプリケーションのソースファイルを修正しなくても済む場合があります。
ここでは、マクロによって main 関数などを置き換えることでスタブ関数の埋め込む方法の一例をご紹介します。

C 言語のマクロをソースファイルに適用するには、通常はソースコード内で #include ディレクティヴを記述しますが、コンパイラの機能を使うことでも行うことができます。
コンパイラの機能を使う場合はソースファイルを変更することなくシンボル名の置換が行えます。スタブ関数を埋め込む手法としては以下のような方法が考えられます。

1. コンパイラのプレインクルード機能（gcc の -include オプション）を使ってヘッダーファイルをインクルードする
2. コンパイラのマクロ定義機能（gcc の -D オプション）を使って置換したいシンボルを置き換える
3. 上記 1, 2 によるシンボル名の置換と併用して置換後の関数を記述したソースファイルをアプリケーションにリンクするなどの合わせ技

ここでは上記 1. の方法での実現例をご紹介します。
この方法では、スタブ関数を埋め込み済みの処理を記述したマクロをヘッダーファイルで用意します。
デバッグ用に必要な変更としては主に下記の 2 点になります。

・ヘッダーファイルをプレインクルードすること（Makefile を修正）

・デバッグサポートファイルをリンクすること（Makefile を修正）

ソースファイルへの直接の変更は無いですし、Makefile 内の記述でデバッグ用とリリース用の内容を変えることは通常の開発プロジェクトでは普通に行われることだと思います。
もちろん、デバッグ用のビルドでしか起こらない逆ハイゼンバグには注意する必要がありますので、適用される置換の内容をよく理解してから使用する必要があります。

(1) マクロファイルの用意

図 2-2 にプレインクルード用のデバッグスタブ関数を埋め込み済みのヘッダーファイルを示します。

図 2-2 kmchook.h

```
#ifndef ___KMCHOOK_H___
#define ___KMCHOOK_H___
#ifdef __KMCHOOK_WITH_PTHREAD
#include <stdlib.h>
#include <errno.h>
#include <pthread.h>
#define __kmchook_pthread_support \
typedef struct {\
    void *(*start_routine)(void *);\
    void *__restrict arg;\
} __kmc_pthread_create;\
static void* __kmc_pthread_entry(void* arg) {\
    extern void __kmc_start(char*);\
    __kmc_pthread_create* pthread_func = (__kmc_pthread_create *)arg;\
    void* ret;\
    __kmc_start((void*)0);\
    ret = pthread_func->start_routine(pthread_func->arg);\
    free(pthread_func);\
    return ret;\
}\
int __kmchook_pthread_create(pthread_t *__restrict __threadp,\
                __const pthread_attr_t *__restrict __attr,\
                void *(*__start_routine) (void *),\
                void *__restrict __arg) {\
    __kmc_pthread_create* pthread_func;\
    pthread_func = (__kmc_pthread_create *)malloc(sizeof(__kmc_pthread_create));\
    if (NULL == pthread_func) return ENOMEM;\
    pthread_func->start_routine = __start_routine;\
    pthread_func->arg = __arg;\
    return __kmc_pthread_create_org(__threadp, __attr, \
__kmc_pthread_entry, pthread_func);\
}
#undef pthread_create
static int (*__kmc_pthread_create_org)(pthread_t*, const pthread_attr_t*,
                void *(*)(void *), void *) = pthread_create;
#define pthread_create __kmchook_pthread_create
#else  /* !__KMCHOOK_WITH_PTHREAD */
#define __kmchook_pthread_support
#endif /* End of __KMCHOOK_WITH_PTHREAD */
#define __kmchook_fork_support \
int __kmchook_fork(void) {\
    extern void __kmc_start(char*);\
    int ret;\
    if ((ret=__kmc_fork_org())==0) __kmc_start((char*)0);\
    return ret;\
}
#undef fork
extern int fork(void);
static int (*__kmc_fork_org)(void) = fork;
#define fork __kmchook_fork
#define main \
main(int argc, char* const argv[]) {\
    extern void __kmc_start(char*);\
    extern int __main(int argc, char* const argv[]);\
    __kmc_start(__KMC_APPNAME);\
    return __main(argc, argv);\
}\
__kmchook_pthread_support \
__kmchook_fork_support \
int __main
#endif /* ___KMCHOOK_H___ */
```

(2) スタブ関数埋め込みの仕組み

kmchook.h は以下の置換を行います。

表 2-1 デバッグモード

名前	置換
main	デバッグスタブ関数埋め込み済み main 関数
	デバッグスタブ関数埋め込み済み _kmchook_pthread_create() 関数
	デバッグスタブ関数埋め込み済み _kmchook_fork() 関数
	_main
fork	_kmchook_fork
pthread_create	_kmchook_pthread_create

マクロですので当然ここで挙げた名前を含め副作用があります。
また、main の置換はトリッキーです。通常の C の main 関数の定義、

```
int main(int argc, char* const argv[]) { ... }
```

のうち、「int」と「(」の間の main を多くのコードで置き換えることに注意してください。main 関数のプロトタイプ宣言をしている場合など、期待通りの展開結果が得られないことがあります。
その他 static な変数はファイル単位でオブジェクトサイズの増加の可能性があることなど、効果と影響を十分に認識した上で使用する必要があります。

(3) アプリケーションのコンパイル

シングルスレッドのアプリケーションの場合は、kmchook.h を -include オプションを使用してプレインクルードします。

```
LINUX86>arm-linux-gcc -c hello.c -o hello.o -I/opt/kmc/kzm-arm11/staging_dir/include ¥
-D__KMC_APPNAME=¥"hello¥" -g -O0 -include kmchook.h ↓
LINUX86>arm-linux-gcc -o hello hello.o kmc-support.o ↓
```

pthread ライブラリを使用するアプリケーションの場合はさらに -D_KMCHOOK_WITH_PTHREAD を定義します。

```
LINUX86>arm-linux-gcc -c pthread.c -o pthread.o -I/opt/kmc/kzm-arm11/staging_dir/
include ¥
-D__KMC_APPNAME=¥"pthread¥" -D__KMCHOOK_WITH_PTHREAD -g -O0 -include kmchook.h ↓
LINUX86>arm-linux-gcc -o pthread pthread.o kmc-support.o ¥
-L/opt/kmc/kzm-arm11/staging_dir/lib -lpthread ↓
```

このように、コンパイルオプションで kmchook.h を指定するだけですので Makefile の書き換えのみで済み、アプリケーションのソースコードの管理の手間は楽になります。
なお、マクロ展開後の main 関数とスタブ関数の位置は元の main 関数（_main）よりも手前になります。本来の main 関数の前に数ステップ実行する必要があるため若干デバッガの操作感は煩雑になります。

2.3.4 Linuix 互換性についての注意

この節ではアプリケーションデバッグサポートファイルを扱うにあたって、各種 Linux ディストリビューションでの互換性の問題によって障害となりうる点を説明します。

(1) ld.so の互換性（Preload 設定が効かない）

標準 C ライブラリに uClibc を使用したディストリビューションなど glibc を使用しない環境では /etc/ld.so.preload ファイルが効かないことがあります。その場合は環境変数 LD_PRELOAD を使用してください。共有ライブラリ機構をサポートしていない環境には対応しておりません。

(2) libdl の互換性（マルチコンテキストデバッグができない）

アプリケーションデバッグサポートファイルを共有ライブラリ形式（libkmcsup.so）でビルドする場合には libdl ライブラリを必要とします。
共有ライブラリ形式（libkmcsup.so）の場合は、デバッグサポートファイルの中にデバッグフック機能つきの pthread_create() 関数と fork() を作成し、それらの関数の中から libpthread ライブラリや libc ライブラリの中にある「本来の」pthread_create() 関数や fork() を呼び出す処理を行っています。このときに「本来の関数」のアドレスを検索するために libdl ライブラリの dlsym インタフェースを利用しています。
いくつかの Linux ディストリビューションでは libdl で提供される関数の動作の互換性に問題があります（glibc に付属の ld.so とは異なる ld.so を使用している Linux で該当することが多いようです）。
不具合が発生する場合の状況と対策方法を具体的に説明します。
まず、デバッグサポートファイル内で「本来の関数を探す処理」は以下のようになっています。

【例】「本来の」fork() を探す

```
__kmc_fork_org = dlsym(RTLD_NEXT, "fork");
```

このとき、互換性に問題の無い Linux ディストリビューションでは dlsym() 関数の戻り値は「本来の」fork() 関数のアドレスになり、そのアドレスは libc の中にあります。互換性に問題のある Linux ディストリビューションでは正しい値を返しません。例えばデバッグサポートファイル内に定義した fork() 関数のアドレスが返っている場合は、無限再起呼び出しになってスタックオーバーフローを引き起こします。全く無関係なアドレスが返っている場合は動作は予期できません。
現状判っている範囲ではこの非互換性は dlsym() 関数の RTLD_NEXT 指定の動作にあります。
回避方法としては RTLD_NEXT 指定を使用せずに「本来の fork() 関数」が入っているライブラリを dlopen() 関数で指定するようにデバッグサポートファイルのコードを変更します。
つまり、予めどのライブラリに検索したい関数が入っているのかを調べておく必要があります。
回避対策後のコードは以下のようになります（libYourLibC 部分は適宜読み替えてください）。

【例】不具合を回避した fork() 検索処理

```
void *handle = dlopen("/lib/libYourLibC.so", RTLD_NOW);
__kmc_fork_org = dlsym(handle, "fork");
/* */ dlclose(handle);
```

なお、dlopen() 関数は対になる dlclose() 関数で閉じるのがセオリーですが、dlclose() 関数を使用すると不具合が発生するケースがあるようです。その場合は dlclose() 関数呼び出しをコメントアウトしてみてください。

2.4 デバッグ環境の起動

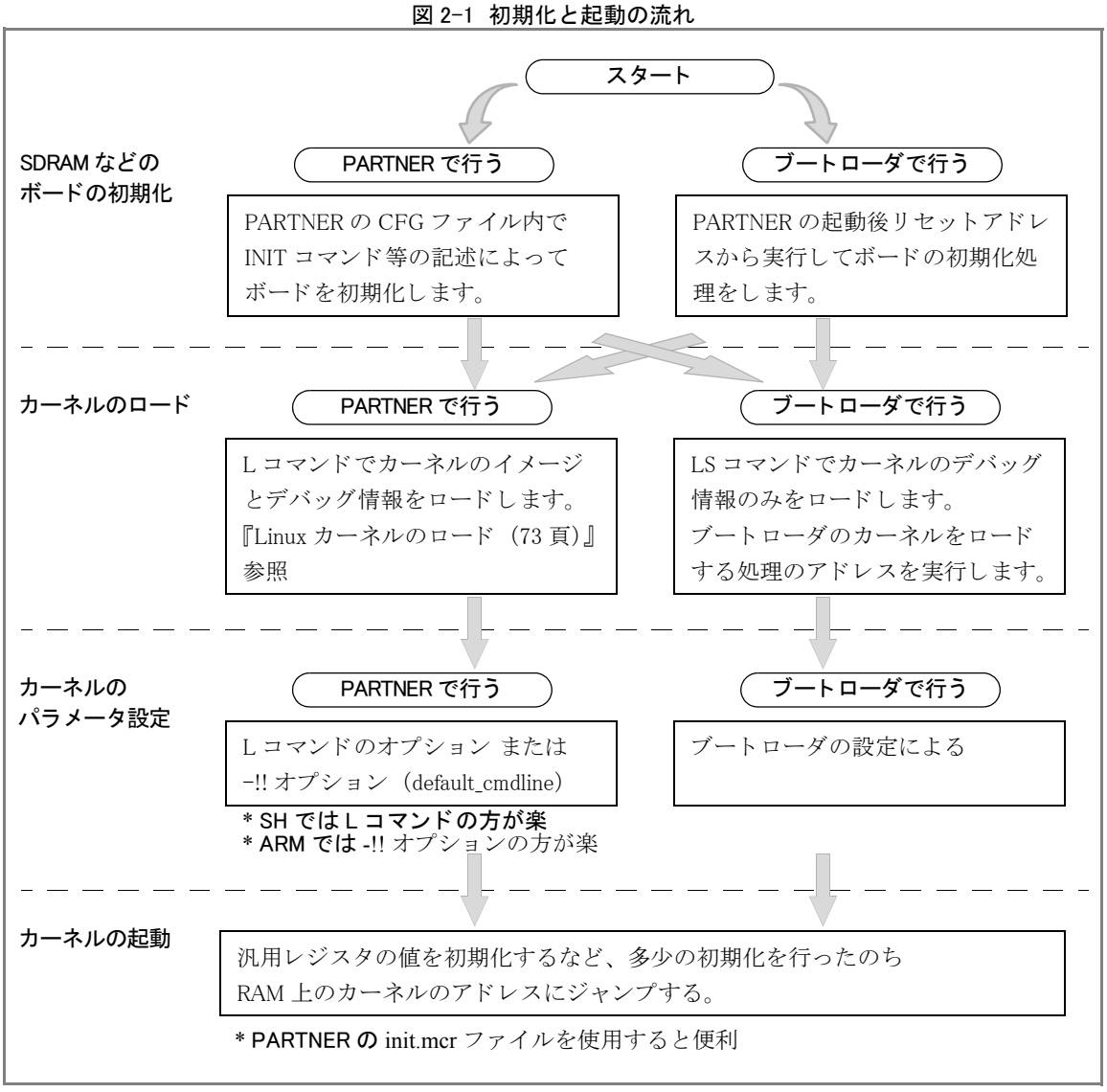
この節では、カーネルモードのデバッグまでの手順を説明します。この説明は『2.2 Linux カーネルソースの修正と設定（50 頁）』で指示された修正がカーネルに対して行われていることが前提です。Linux カーネルの設定メニュー（『カーネルコンフィグレーション（54 頁）』参照）の PARTNER に必要な設定を行い、コンフィグレーション設定後、Linux カーネル（vmlinux）を作成してください。

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。なお、Linux カーネルのコンフィグレーション設定項目で必要な箇所はデバッグしたい対象範囲に依存するので、PARTNER の起動モードとは関係ありません。

2.4.1 ターゲットの初期化と起動方法の種別

組み込み Linux の開発環境では、ターゲットの初期化や起動を JTAG デバッガ（PARTNER）を使って行う方法とターゲットボード上のブートローダを使って行う方法が考えられます。最終的な製品では JTAG デバッガは使いませんので、開発の過程で変更することもあると思います。
この節では PARTNER を使用する方法を解説しますので、ブートローダを使用する場合は手順や設定項目がどのように変わるのか図 2-1 で確認してください。

図 2-1 初期化と起動の流れ



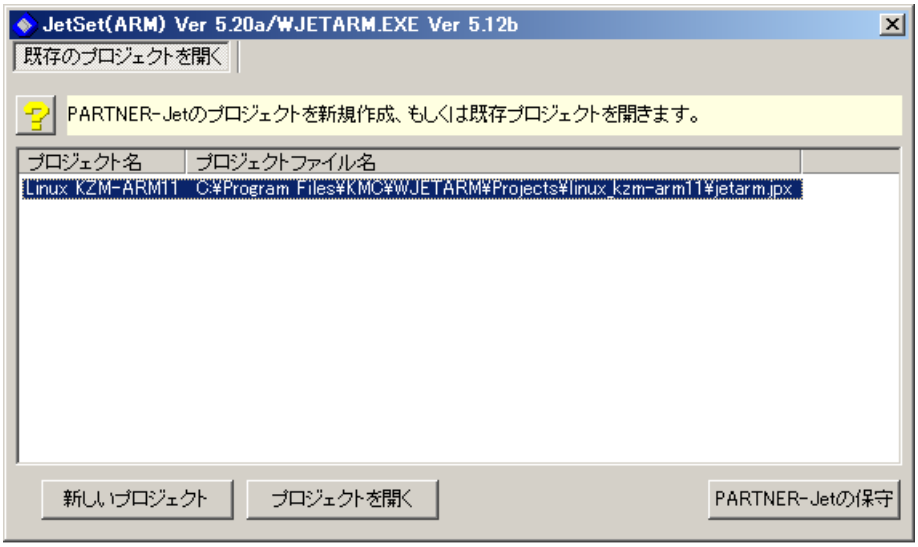
2.4.2 PARTNER の設定と起動

カーネルモードで PARTNER を起動するには、以下に示す手順で行います。

(1) 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) 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。

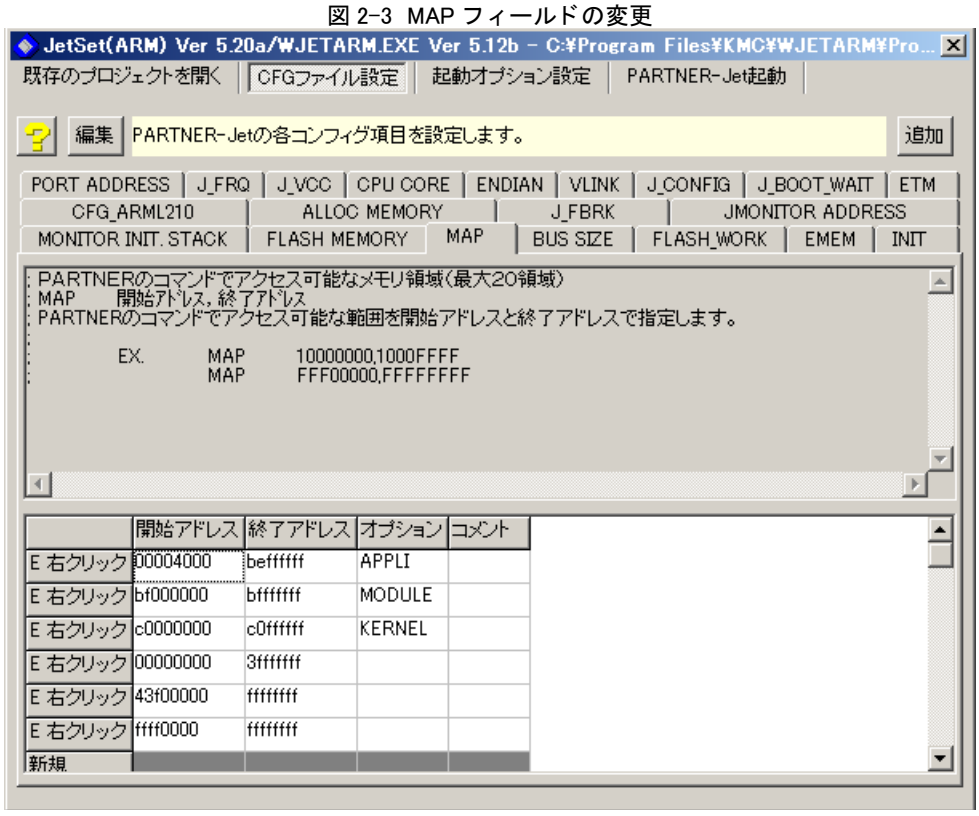
図 2-2 PARTNER プロジェクトのオープン



(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『MAP フィールド (133 頁)』を参照してください。

図 2-3 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、以下の通りです。

● 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。

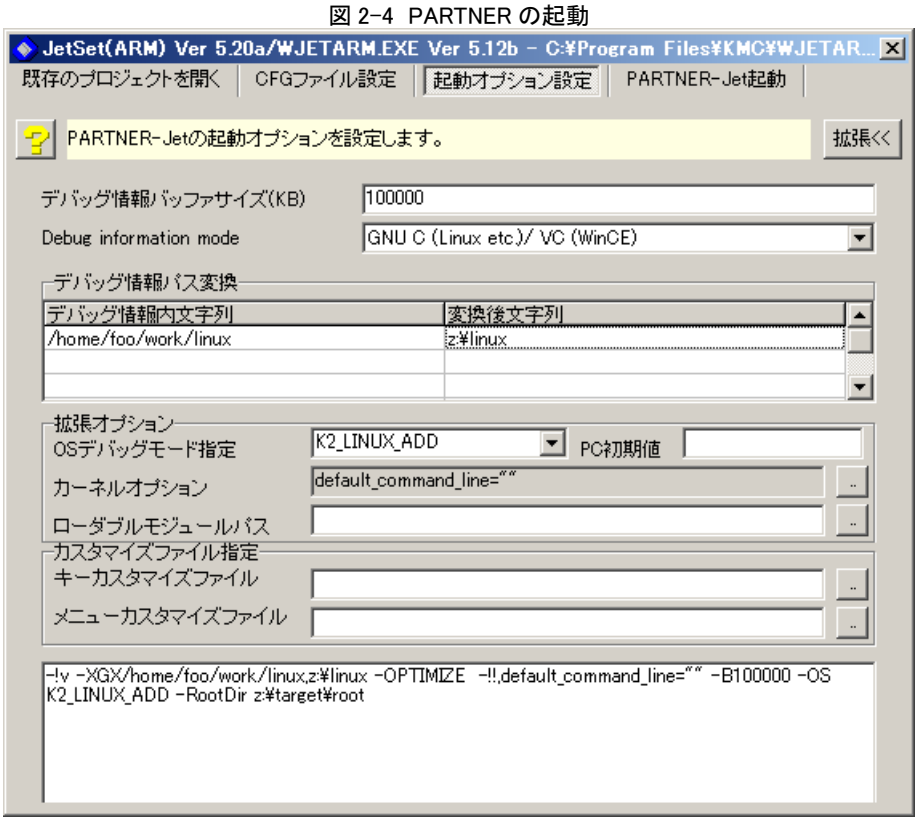
● デバッグ情報パス変換 (-XGX オプション)

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)

『カーネル (NON_ADD) モード』『カーネル ADD モード』『アプリケーション (NON_ADD) モード』『アプリケーション ADD モード』のオプションを選択します。推奨は「K2_LINUX_ADD」または「K2_LINUX_ADD_V26」です。
各オプションについての詳細は、『 -OS オプション (138 頁)』及び『付録 F PARTNER の動作モード選択について (251 頁)』を参照してください。

図 2-4 PARTNER の起動



(4) PARTNER ウインドウの起動

JETSET 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 2-5 PARTNER の起動

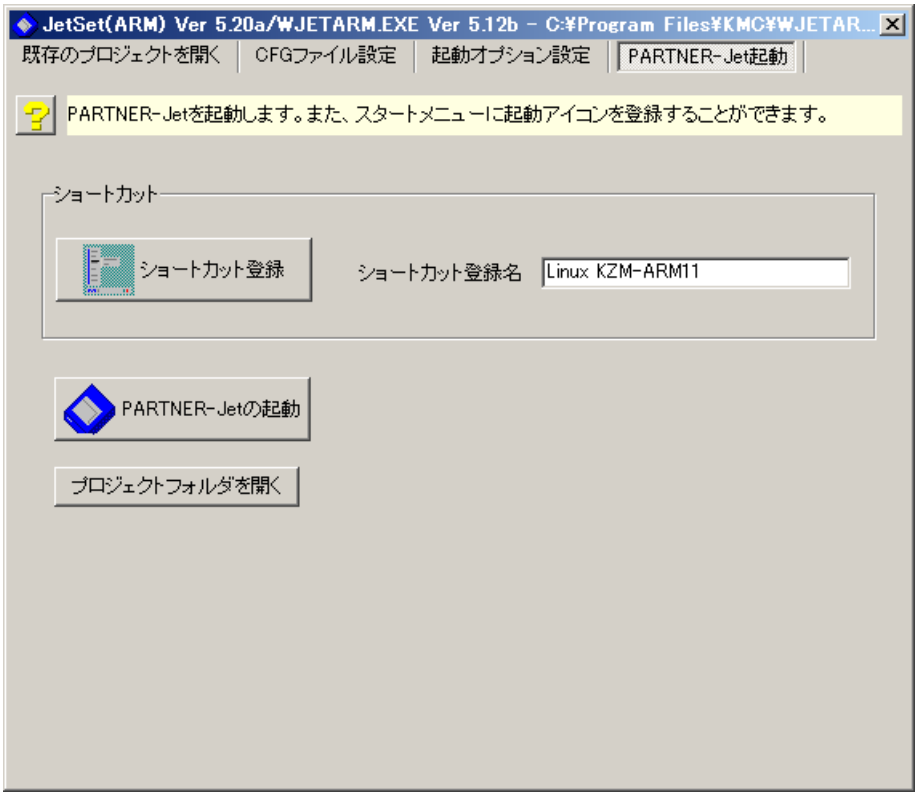
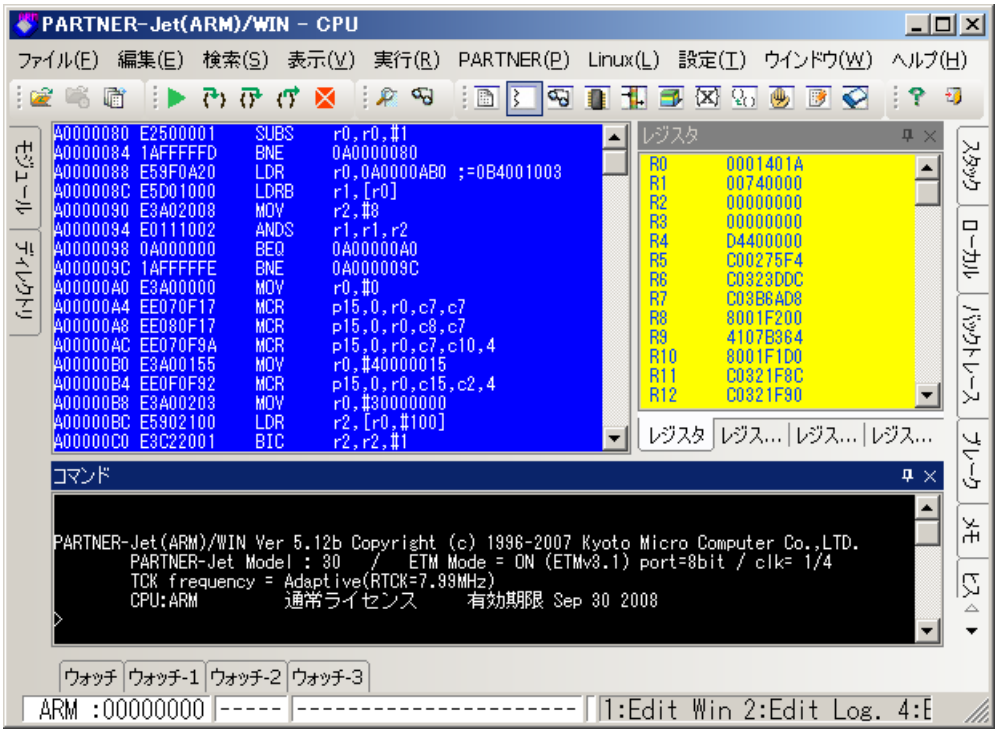


図 2-6 PARTNER の起動画面

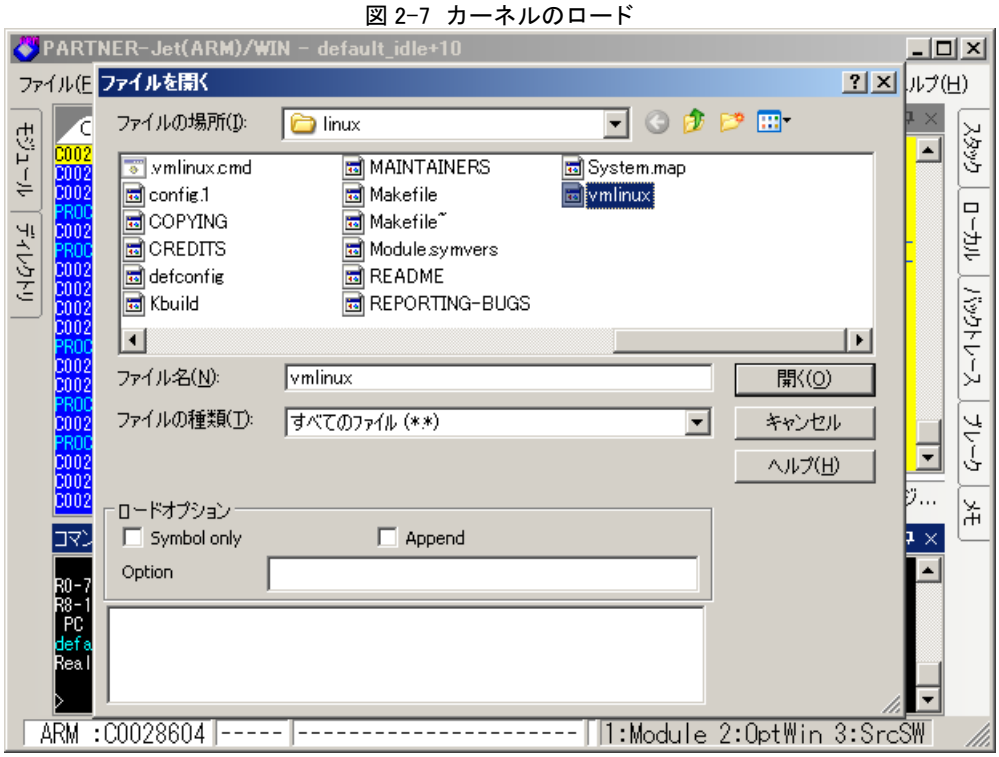


2.4.3 Linux カーネルのロード

『2.4.2 PARTNER の設定と起動（70 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 2-7 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。

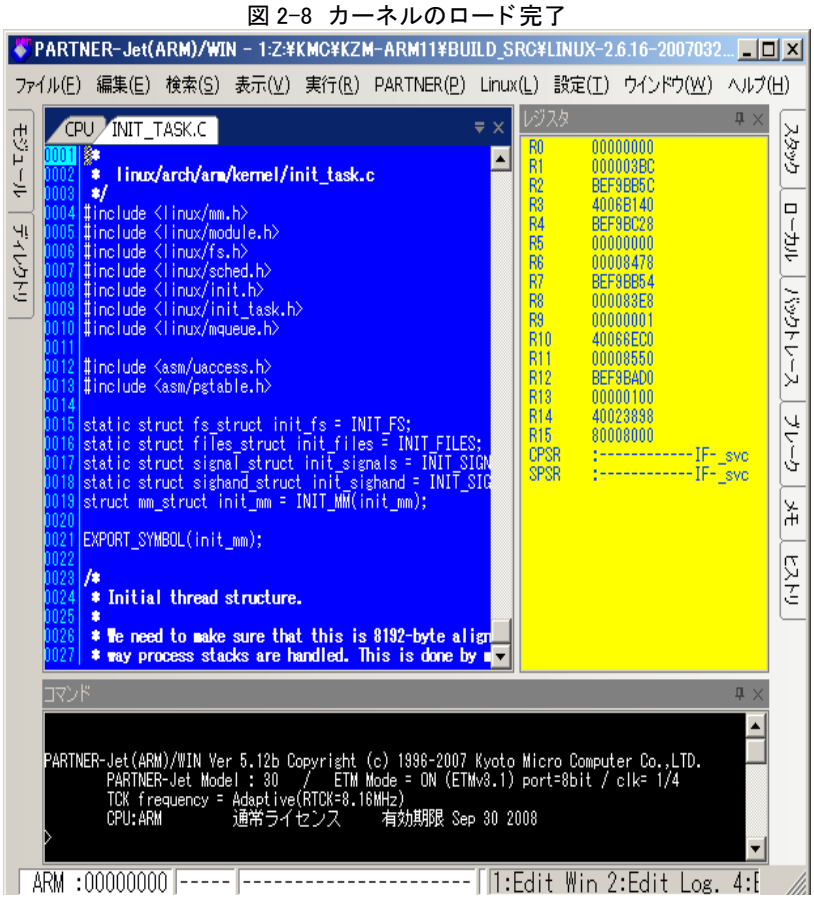
PARTNER のコマンドウインドウにおいて、次のコマンドを入力します。

【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。
Linux カーネルが ROM から RAM へ転送されるシステムでは、L コマンドの代わりに LS コマンドを使用してデバッグ情報のみを PARTNER にロードしてください。

図 2-8 カーネルのロード完了



正常にロードされると、PARTNER のコードウインドウに Linux のソースコードが表示されます。コードウイ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（140 頁)』参照)。

PARTNER のコードウインドウに 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。

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。

コラム 2-9 PARTNER のマクロ機能の利用

PARTNER にはマクロコマンド機能が搭載されており、複数のコマンドを組み合わせたりマクロ制御コマンドを使用して、新たなコマンドを作成することができます。マクロ制御コマンドには以下のようなものがあります。詳しくは PARTNER のヘルプを参照してください。

- {(マクロコマンドの定義)
- DO{..}WHILE(DO-WHILE マクロ)
- FOR{..}(FOR マクロ)
- WHILE{..}(WHILE マクロ)
- REPEAT{..}(REPEAT マクロ)
- BREAK(マクロ抜け出し)
- LALL(マクロ表示出力指定)
- SALL(マクロ表示抑止指定)
- MLIST(マクロ表示)
- KILL(マクロの削除)
- IF{..}(IF マクロ)
- KEYIN(キー入力指定)

また、プロジェクトフォルダに「init.mcr」という名前のテキストファイルを作成して中にマクロを記述しておくと PARTNER ウインドウの起動時にマクロが読み込まれます。

この機能を使用すると、カーネルロードの手順を簡略化したり簡単に切り替えたりできます。

例えば、図 2-10 のように記述すると PARTNER のコマンドで「PT>load_nor」を実行すると、カーネルは Windows PC から PARTNER を使用してロードし、ルートファイルシステムは NOR フラッシュを使用します。「PT>load_ide」ならば HDD 上のルートファイルシステムを使用します。

図 2-10 マクロ作成例

```
{load_nor        マクロ名
l z:¥kzm¥linux¥vmlinux noinitrd console=ttymxc0 root=/dev/mtdblock2
rootfstype=jffs2 init=linuxrc ip=none ,/offs=0xc0000000
                                  L コマンドのオプションでカーネルパラメータを設定
_r0=0
_r1=_956          レジスタ値 / プログラムカウンタを設定
_r15=0x80008000
br start_kernel,ex     ブレークポイント設定
}

{load_ide
l z:¥kzm¥linux¥vmlinux noinitrd console=ttymxc0 root=/dev/hda1 rootfs=/dev/ide/
host0/bus0/target0/lun0/part1 init=linuxrc ,/offs=0xc0000000
_r0=0
_r1=_956
_r15=0x80008000
br start_kernel,ex
}
```

2.4.4 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 2-11 Linux カーネルのブートアップメッセージ

```
TCP established hash table entries: 4096 (order: 2, 16384 bytes)
TCP bind hash table entries: 4096 (order: 2, 16384 bytes)
TCP: Hash tables configured (established 4096 bind 4096)
TCP reno registered
TCP bic registered
NET: Registered protocol family 1
VFP support v0.3: implementor 41 architecture 1 part 20 variant b rev 2
IP-Config: Guessing netmask 255.255.255.0
IP-Config: Complete:
      device=eth0, addr=192.168.1.202, mask=255.255.255.0, gw=255.255.255.255,
     host=kzm-arm11, domain=, nis-domain=(none),
     bootserver=255.255.255.255, rootserver=192.168.1.16, rootpath=
Looking up port of RPC 100003/2 on 192.168.1.16
Looking up port of RPC 100005/1 on 192.168.1.16
VFS: Mounted root (nfs filesystem).
Freeing init memory: 112K
Starting the hotplug events dispatcher udevd
Synthesizing initial hotplug events
Initializing random number generator... done.


Welcome to the Erik's uClibc development environment.

kzm-arm11 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。

第 3 章　デバッグ手順リファレンス

この章では、Linux 対応 PARTNER を使用したデバッグ手順について説明します。

3.1 カーネルのデバッグ

この節では、Linux カーネルのデバッグ方法を説明します。

3.1.1 デバッグに必要な設定条件

表 3-1 カーネルデバッグの条件

* 設定条件 / デバッグ対象	カーネルメニュー						-OS オプション			
	Debug infomation type	Enable patch for PARTNER debug	Loadable module debug by PARTNER-Jet (hook in kernel side)	Loadable module debug by PARTNER-Jet (hook in module side)	use PARTNER GigaTrace (PARTNER-Jet M40, ...)	enable PARNTER-Jet Event Tracker	カーネルモード	カーネル ADD モード	アプリケーションモード	アプリケーション ADD モード
カーネル	◎						—— †			

* ◎：必須、○：推奨、△：非推奨、×：不可、空欄：どちらでもよい

† カーネルデバッグに -OS オプション指定は必須ではありませんが、いずれかを指定することを推奨します。

カーネルデバッグに必要なことはカーネルがデバッグ情報付きでコンパイルされていることだけです。カーネルソースの修正が行えない環境でも、カーネルのデバッグ情報を生成する必要があります。

コラム 3-1 カーネルのデバッグ情報の生成方法

KMC パッチの「Debug information type」メニューは、デバッグ情報の付加と種別設定を手軽に行うためのものです。このメニューを使わずに行うには以下のような方法があります。
いずれの方法でもデバッグ情報つきのカーネルをビルドすることができます。

(1) カーネルコンフィグの「Kernel Hacking」メニューで指定する

(2) Makefile を書き換えて「CFLAGS+=-g」のように行を追記する

(3) make するときのコマンド引数で CFLAGS 変数を指定する

3.1.2 Linux カーネルの初期化時のデバッグ

PARTNER で Linux 起動時の処理をデバッグする際にはシステムの構成に注意を払う必要があります。

(1) PARTNER でカーネルを RAM にロードして実行する場合

カーネルをロードした時点でデバッグ可能です (手順は『Linux カーネルのロード (73 頁)』参照)。
ただし、システム構成によっては注意するべきことがあります。

● 自己解凍型の場合

おもにカーネルイメージを保存するストレージ容量を節約する目的でカーネルが圧縮されているシステムがあります。その中でも RAM 上に転送された時点ではまだ解凍されていないようなシステムの場合は、PARTNER でカーネルをロードした時点ではカーネルの処理にブレークポイントを設定することができません。ひとつのファイルであっても実体は自己解凍ルーチン＋ Linux カーネルの 2 つのプログラムになっていて Linux カーネルのデバッグ情報はついていないはずです。
この場合は自己解凍ルーチンの展開処理が終わった時点で、LS コマンドで Linux カーネルのデバッグ情報を読み込んでください。

● 物理アドレス (PA) と仮想アドレス (VA) が異なる場合

Linux カーネルが仮想アドレスでリンクされているシステムがあります (おもに ARM CPU 用の Linux カーネルが該当します)。
このようなシステムでは RAM 上で実行されるごく初期の段階の処理で TLB の初期化が行われ MMU が有効になります。この初期化処理が済むまでは仮想アドレス (VA) にはアクセスすることができないため、VA でソフトウェアブレークポイントの設定、変数を参照、ディスアセンブル等をしたりすることが出来ません。
VA が使用可能になって後の箇所 (例えば start_kernel シンボル) に実行型ハードウェアブレークポイントを設定しておくと、ハードウェアブレークポイントで停止して以降はソフトウェアブレークポイントを使用することが可能です。

【例】

```
PT>l vmlinux ↓
PT>br start_kernel,ex ↓
PT>g ↓
```

(2) ブートローダがカーネルをロードして実行する場合

ブートローダはおもに ROM から RAM へカーネルイメージを転送するために用いられます。この場合、カーネルのデバッグ情報を PARTNER にロード済みであっても、カーネルの転送前にはソフトウェアブレークポイントを使用することができません。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。

(3) ROM 上で実行される場合

カーネルプログラムの一部または全部が ROM 上に配置されているシステム (XIP カーネルとも呼ばれます) の場合、ROM 上に配置されているプログラムコードにはソフトウェアブレークポイントを設定することができません。ハードウェアブレークポイントを設定してください。

ソフトウェアブレークポイントに関する注意事項
ソフトウェアブレークポイントは RAM 上のプログラムコードをデバッガが書き換えを行うことでプログラム停止を実現する技術です。
「プログラム転送前には使用できない」「ROM 上のプログラムには使用できない」といった制限はその仕組みに由来しています。

3.1.3 システム起動後のカーネルのデバッグ

カーネルを実行 (手順は『Linux カーネルの実行 (76 頁)』参照) 後のカーネルデバッグについて特に制限はありません。
動作中に PARTNER で [ESC] キーを押して CPU 停止状態にしてカーネル内の任意の箇所にブレークポイントを設定することができます。

3.2 ローダブルモジュールのデバッグ（手法１）

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。

本節の手法は Linux カーネルのバージョンが 2.4 系でも 2.6 系でも使用できます。ただし 2.6 系カーネルの場合は、ローダブルモジュールのソースコード修正が不要な『ローダブルモジュールのデバッグ（手法２）(86 頁)』をお勧めします。

3.2.1 デバッグに必要な設定条件

表 3-2 ローダブルモジュールデバッグ（手法１）の条件

* 設定条件 / デバッグ対象	カーネルメニュー						-OS オプション			
	Debug infomation type	Enable patch for PARTNER debug	Loadable module debug by PARTNER-Jet (hook in kernel side)	Loadable module debug by PARTNER-Jet (hook in module side)	use PARTNER GigaTrace (PARTNER-Jet M40, ...)	enable PARNTER-Jet Event Tracker	カーネルモード	カーネル ADD モード	アプリケーションモード	アプリケーション ADD モード
ローダブルモジュール	◎	◎	—	◎			○	○	△ †	

* ◎：必須、○：推奨、△：非推奨、×：不可、空欄：どちらでもよい
† できないわけではないですがお勧めしません。ローダブルモジュール内をデバッグ・ブレークするときにはカーネル・CPU が停止します。

『2.2 Linux カーネルソースの修正と設定（50 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、『5.2 手動ローダブルモジュールデバッグ（182 頁）』を参照してください。

3.2.2 デバッグの手順

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) ローダブルモジュールソースの修正（83 頁）

(2) モジュールのビルド（83 頁）

(3) デバッグの準備（83 頁）

(4) ローダブルモジュールの組み込み（84 頁）

(5) PARTNER のブレーク（84 頁）

ローダブルモジュールのデバッグ（手法１）

(1) 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】デバッグ用宣言の埋め込み

```
+#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】ローダブルモジュールパスの埋め込み

```
+#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（143 頁）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。

(2) モジュールのビルド

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。Linux カーネルソースツリー内のローダブルモジュールは、『(1) ローダブルモジュールソースの修正（83 頁）』で指定したデバッグ情報が付加されます。

```
LINUX86>make modules ↓
```

(3) デバッグの準備

PARTNER と Linux を立ち上げて使用可能な状態にします（『デバッグ環境の起動（68 頁）』参照）。

(4) ローダブルモジュールの組み込み

正常に起動したら、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カーネルに組み込みます。ローダブルモジュールのインストールはスーパーユーザ（root）で insmod コマンドを使用します。

```
TGT>insmod rd.o ↓
```

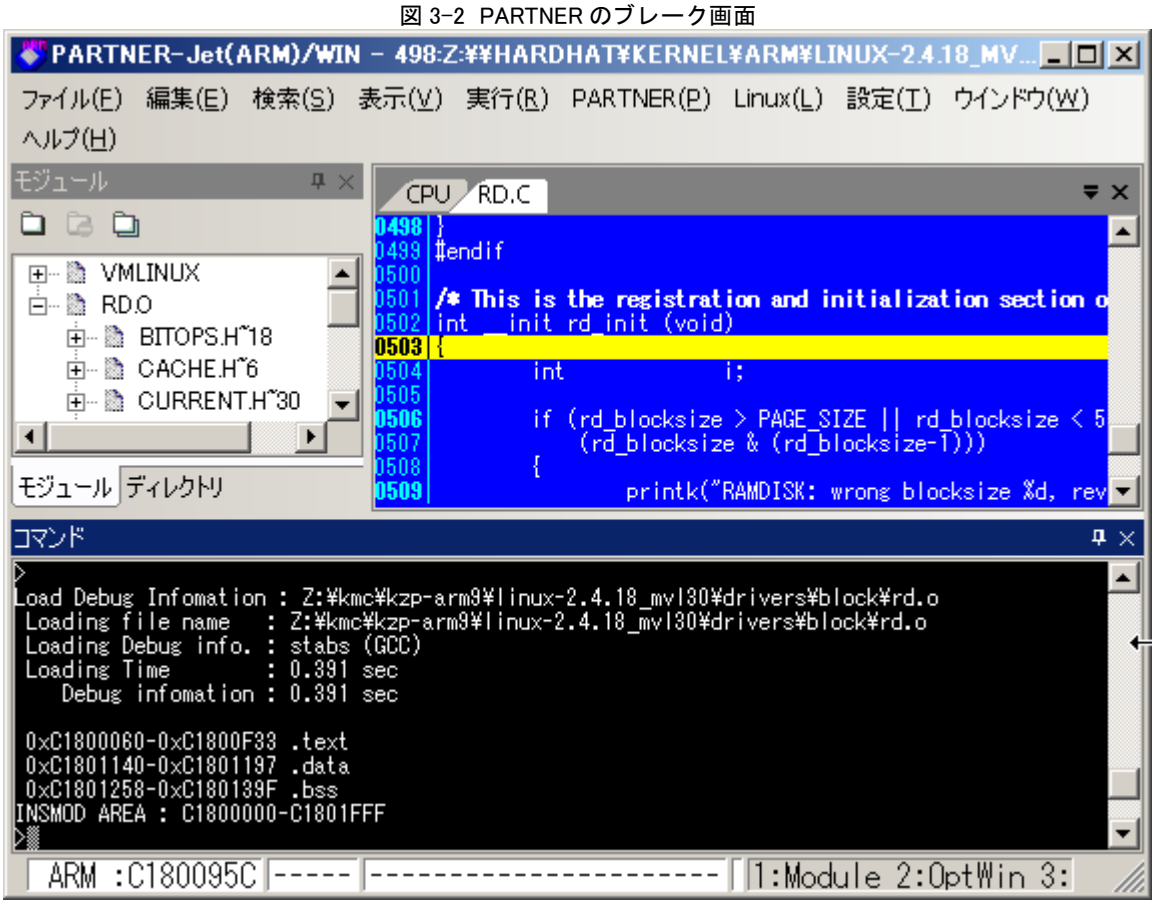
図 3-1 ローダブルモジュールのインストール

```
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
```

(5) 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 3-2 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（143 頁）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（140 頁）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) した後、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。

3.3 ローダブルモジュールのデバッグ（手法２）

2.6 系 Linux カーネル用のローダブルモジュール形式は 2.4 系カーネルのものとは異なり、.ko 形式になっています。PARTNER は 2.6 系カーネルの場合にはローダブルモジュールのソースコードの修正なしに insmod 時点からのデバッグ開始が可能です。

本節の手法は Linux カーネルのバージョンが 2.6 系の場合のみ可能です。2.4 系カーネルでは使用できません。
2.6 系カーネルの場合は『ローダブルモジュールのデバッグ（手法１）(81 頁)』の方法も使用できますが、本節の手法を推奨します。

3.3.1 デバッグに必要な設定条件

表 3-1 ローダブルモジュールデバッグ（手法２）の条件

* 設定条件 / デバッグ対象	カーネルメニュー						-OS オプション			
	Debug infomation type	Enable patch for PARTNER debug	Loadable module debug by PARTNER-Jet (hook in kernel side)	Loadable module debug by PARTNER-Jet (hook in module side)	use PARTNER GigaTrace (PARTNER-Jet M40, ...)	enable PARNTER-Jet Event Tracker	カーネルモード	カーネル ADD モード	アプリケーションモード	アプリケーション ADD モード
ローダブルモジュール	◎	◎	—	◎			○	○	△ †	

* ◎：必須、○：推奨、△：非推奨、×：不可、空欄：どちらでもよい
† できないわけではないですがお勧めしません。ローダブルモジュール内をデバッグ・ブレークするときにはカーネル・CPU が停止します。

『2.2 Linux カーネルソースの修正と設定（50 頁）』を行っていない場合は、ここで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、『5.2 手動ローダブルモジュールデバッグ（182 頁）』を参照してください。

3.3.2 デバッグの手順

この節では、RAM ディスク（rd.ko）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) モジュールのビルド（88 頁）

(2) デバッグの準備（88 頁）

(3) ローダブルモジュールの組み込み（89 頁）

(4) PARTNER のブレーク（89 頁）

(1) モジュールのビルド

2.6 系カーネルのローダブルモジュールはソースツリーの Makefile によってコンパイルされます。以下の 2 点を確認してください。

1. カーネルのコンパイル設定でデバッグオプションが付いているか
   カーネルに PARTNER のパッチが入っているならば、コンフィグレーションメニューの [PARTNER Debugging]->[Debug infomation type] から設定できます。Makefile を直接編集して CFLAGS にコンパイラへのフラグを追記する方法でも可能です。
2. モジュールとしてコンパイルされるようになっているか
   デバッグ対象のデバイスドライバがカーネルツリーの中に含まれているものの場合、ローダブルモジュールとしてコンパイルされる設定になっているかどうか確認してください。
   RAM ディスク (rd.ko) の場合は [Device Drivers]->[Block devices]->[RAM disk support] の項目が <M> に設定されているかどうか確認します。

デバッグするローダブルモジュールを作成します。

```
LINUX86>make modules ↓
```

ターゲットのファイルシステムにインストールします。例えば、ターゲットのファイルシステムのディレクトリツリーが ${TARGET_ROOT} にあるならば、以下のように入力します。

```
LINUX86>make INSTALL_MOD_PATH=${TARGET_ROOT}/ modules_install ↓
```
（ターゲットの /lib/modules/2.6.16.XXX/ 以下にインストールされます）

(2) デバッグの準備

PARTNER と Linux を立ち上げて使用可能な状態にします（『デバッグ環境の起動（68 頁）』参照）。PARTNER の LINUX コマンド（154 頁）でデバッグ対象のモジュールを登録します。登録可能なモジュール数は 8 個です（Linux カーネルの KMC パッチ内で定義されているため変更は可能です）。

```
PT>linux module z:¥kmc¥kzm-arm11¥root¥lib¥modules¥2.6.16.19-kzm¥kernel¥drivers¥block¥rd.ko ↓
```
（デバッグ対象のモジュールを登録）

「linux module <パス名>」の操作はローダブルモジュールが insmod される前ならばいつでもかまいませんが、Linux カーネルが有効になった後（start_kernel シンボル以降）である必要があります。
デバッグ対象モジュールの登録は「linux module」コマンドで確認したり、「linux module clr」で削除したりできます。詳しくは、『LINUX コマンド（154 頁）』を参照してください。

登録されたかどうか確認します。

```
PT>linux module ↓
 00000000 : z:¥kmc¥kzm-arm11¥root¥lib¥modules¥2.6.16.19-kzm¥kernel¥drivers¥block¥rd.ko
PT>
```

(3) ローダブルモジュールの組み込み

正常に起動したら、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カーネルに組み込みます。ローダブルモジュールのインストールはスーパーユーザ（root）で insmod コマンドを使用します。

```
TGT># insmod /lib/modules/2.6.16.19-kzm/kernel/drivers/block/rd.ko ↓
```

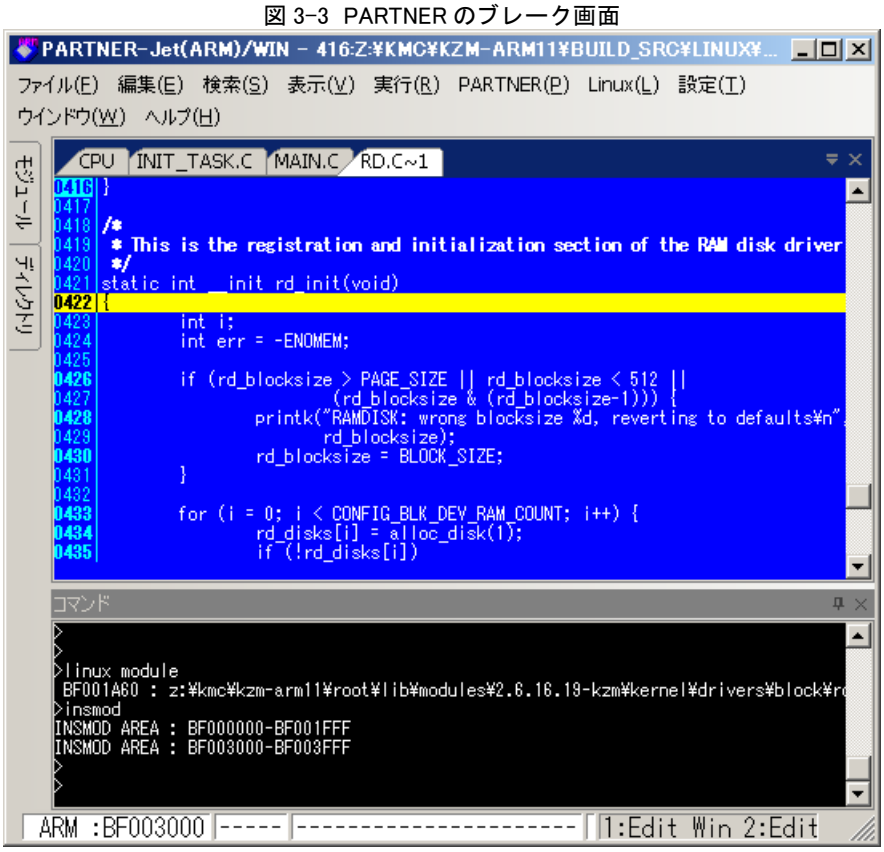
(4) PARTNER のブレーク

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。

なお、カーネルに組み込まれたモジュールの配置は INSMOD コマンド（156 頁）によって確認できます。

```
PT>insmod ↓
INSMOD AREA : BF000000-BF001FFF
INSMOD AREA : BF003000-BF003FFF
```

図 3-3 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 (143 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 (140 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。

3.4 アプリケーションのデバッグ

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(『-OS オプション（138 頁)』参照)。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。

3.4.1 デバッグに必要な設定条件

表 3-2 アプリケーションデバッグの条件

* 設定条件 / デバッグ対象	カーネルメニュー						-OS オプション			
	Debug infomation type	Enable patch for PARTNER debug	Loadable module debug by PARTNER-Jet (hook in kernel side)	Loadable module debug by PARTNER-Jet (hook in module side)	use PARTNER GigaTrace (PARTNER-Jet M40, ...)	enable PARNTER-Jet Event Tracker	カーネルモード	カーネル ADD モード	アプリケーションモード	アプリケーション ADD モード
アプリケーション	◎	◎					○	○	—†	

* ◎：必須、○：推奨、△：非推奨、×：不可、空欄：どちらでもよい
† アプリケーションモードのデバッグ方法は『5.1 アプリケーションモードデバッグ（172 頁)』を参照してください。

『2.2 Linux カーネルソースの修正と設定（50 頁)』を行っていない場合は、ここで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、『5.3 手動アプリケーションデバッグ（191 頁)』を参照してください。

3.4.2 デバッグの手順

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)のデバッグ手順を説明します。

(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。
アプリケーションのソースコードは全く修正する必要がありません。

【例】

```
LINUX86> linux-arm-gcc -o sample.out -g sample.c ↓
```

デバッグサポート用ファイルを『Preload ライブラリ方式（60 頁）』で使用する場合、デバッグサポートライブラリ（libkmcsup.so）は<u>アプリケーションにリンクしないでください</u>。（ターゲット上で ldd コマンドを使用すればリンクされている共有ライブラリを確認できます。）

(2) デバッグの準備

『デバッグ環境の起動（68 頁）』を参照し、カーネルモードの PARTNER で Linux のデバッグができる状態にします。

(3) アプリケーション用 PARTNER ウインドウを開く

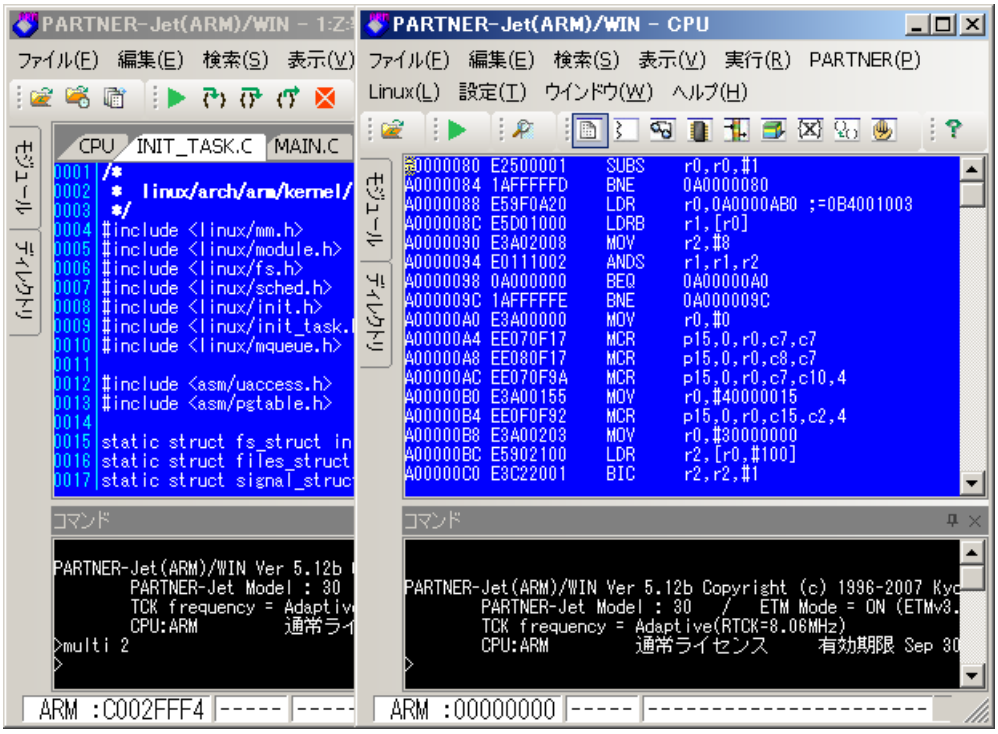
Linux カーネルとアプリケーションを別の PARTNER ウインドウでデバッグする場合は、MULTI コマンド（163 頁）で複数の PARTNER ウインドウを起動します。サンプルプログラム (sample) をデバッグする場合は、カーネル用 PARTNER ウインドウとアプリケーション用 PARTNER ウインドウの 2 つの PARTNER ウインドウを起動します。

ターゲットが実行中の場合はまず [ESC] キーで CPU の実行を停止させてから以下のコマンドを入力します。

```
PT>multi 2 ↓
```

カーネルをロードしたPARTNERウインドウを使用してプロセス数がひとつだけのアプリケーションをデバッグする場合は、2 つ目の PARTNER ウインドウを起動する必要はありません。

図 3-4 複数 PARTNER ウインドウの起動



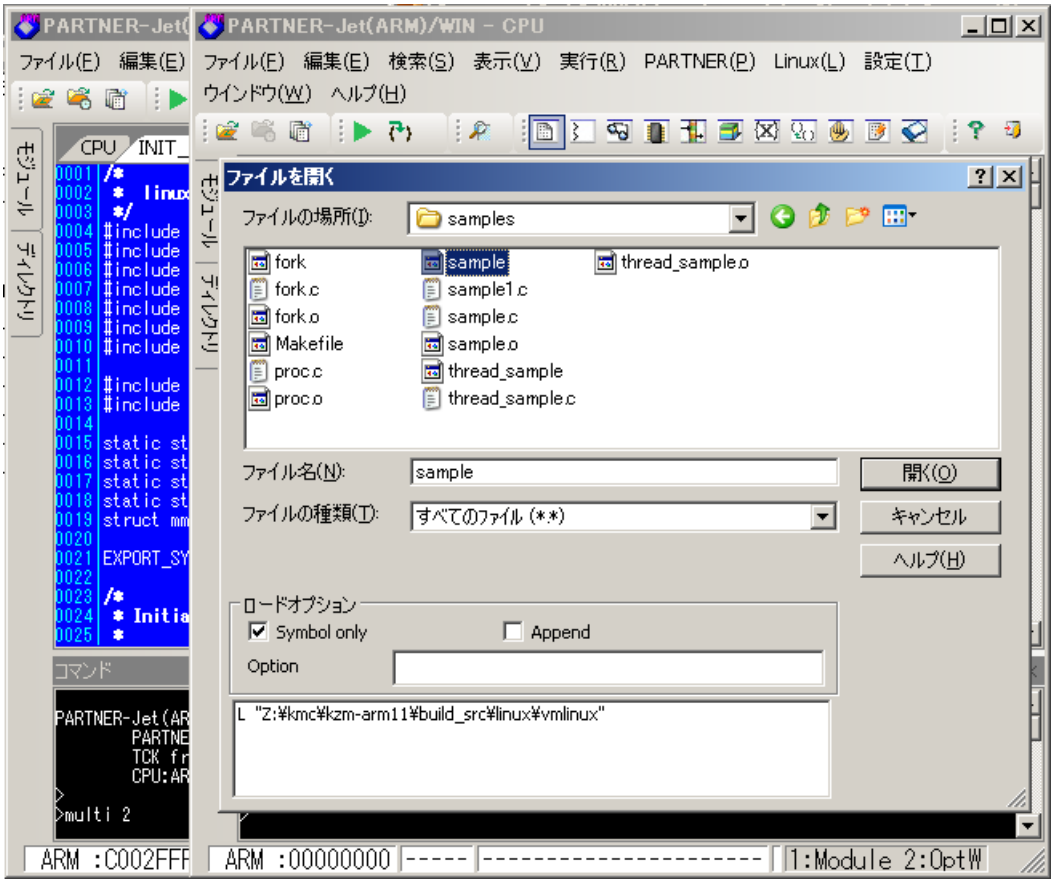
MULTI コマンド（163 頁）や -MULTI オプション（145 頁）で起動した PARTNER ウインドウのウインドウ情報（ウインドウ配置等）やコマンドヒストリは、新規に生成されたプロジェクトファイル（JETARM_1.JPX など）に保存され、次回起動時に利用されます。

(4) 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 3-5 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) ブレークポイントの設定

アプリケーションの main() 関数に実行型ハードウェアブレークポイントを設定します。

```
PT>br main.ex ↓
```

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、既に設定可能な数の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数は CPU により異なります）。ブレークポイントを削除してから、アプリケーションのエントリにブレークポイントを再度設定してください。

(6) 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 ↓

(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実行されると、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 3-6 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。

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。『2.2.6 カーネルコンフィグレーション（54 頁）』の設定と『2.4.2 PARTNER の設定と起動（70 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、サポートライブラリ (libkmcsup.so.2.0.0) が Preload されているか確認してください。

(8) アプリケーションのアタッチと確認

ATTACH コマンド（152 頁）を使用してアプリケーションにアタッチします。PARTNER はアプリケー

ションの PID や配置情報を獲得し、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。

```
PT>ps ↓
  1(0x1)        /bin/busybox
  398(0x18e)    /sbin/udevd
  612(0x264)    /bin/bash
  613(0x265)    /bin/busybox
  614(0x266)    /bin/busybox
  740(0x2e4)    /root/sample
PT>attach 740 ↓
または
PT>attach sample ↓
```

PSID コマンド（158 頁）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できます。
アプリケーションがアタッチされていないときはアスタリスク表示になります。

```
PT>psid ↓
PSID ****
```

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。
アプリケーションがアタッチされている場合の PSID コマンド（158 頁）の結果は以下のようになります。

```
PT>psid ↓
PSID SET 740(0x2E4)  CURRENT 740(0x2E4) [ADD MODE]
APPLI. AREA : 00008000-00009FFF
APPLI. AREA : 00011000-00015FFF
APPLI. AREA : BEAD4000-BEAD4FFF
（表示される内容は CPU 種別や MAP フィールドの設定によって異なります）
```

共有ライブラリのデバッグもできるように、LINUX コマンド（154 頁）も入力することをお勧めします。

```
PT>linux load so ↓
```

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 ↓
```

3.5 マルチスレッドアプリケーションデバッグ

カーネルモードで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。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『3.4 アプリケーションのデバッグ（91 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。

マルチスレッドアプリケーションのデバッグには、各スレッド実行コンテキストを同じ 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（158 頁）または、-OS オプション（138 頁）で指定します。

3.5.1 デバッグに必要な設定条件

表 3-3 マルチスレッドアプリケーションデバッグの条件

* 設定条件 / デバッグ対象	カーネルメニュー						-OS オプション			
	Debug infomation type	Enable patch for PARTNER debug	Loadable module debug by PARTNER-Jet (hook in kernel side)	Loadable module debug by PARTNER-Jet (hook in module side)	use PARTNER GigaTrace (PARTNER-Jet M40, ...)	enable PARNTER-Jet Event Tracker	カーネルモード	カーネル ADD モード	アプリケーションモード	アプリケーション ADD モード
アプリケーション	◎	◎					○†		—‡	

* ◎：必須、○：推奨、△：非推奨、×：不可、空欄：どちらでもよい
† ADD モード、NON_ADD モードにより動作が異なります
‡ アプリケーションモードのデバッグ方法は『5.1 アプリケーションモードデバッグ（172 頁）』を参照してください。

『2.2 Linux カーネルソースの修正と設定（50 頁）』を行っていない場合は、ここで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、『5.3 手動アプリケーションデバッグ（191 頁）』を参照してください。

3.5.2 デバッグの手順

この節では、サンプル(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スレッドアプリケーションのデバッグ手順を説明します。

(1) アプリケーションの作成（100 頁）
(2) デバッグの準備（100 頁）
(3) アプリケーション用 PARTNER ウインドウを開く（100 頁）
(4) アプリケーションのデバッグ情報のロード（101 頁）
(5) ブレークポイントの設定（102 頁）
(6) アプリケーションの実行（102 頁）
(7) PARTNER のブレーク（102 頁）

(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。
アプリケーションのソースコードは全く修正する必要がありません。

【例】

```
LINUX86> linux-arm-gcc -o thread_sample -g thread_sample.c -lpthread ↓
```

デバッグサポート用ファイルを『Preload ライブラリ方式（60 頁）』で使用する場合、デバッグサポートライブラリ（libkmcsup.so）は<u>アプリケーションにリンクしないでください</u>。（ターゲット上で ldd コマンドを使用すればリンクされている共有ライブラリを確認できます。）

(2) デバッグの準備

『デバッグ環境の起動（68 頁）』を参照し、カーネルモードの PARTNER で Linux のデバッグができる状態にします。

(3) アプリケーション用 PARTNER ウインドウを開く

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 ↓
```

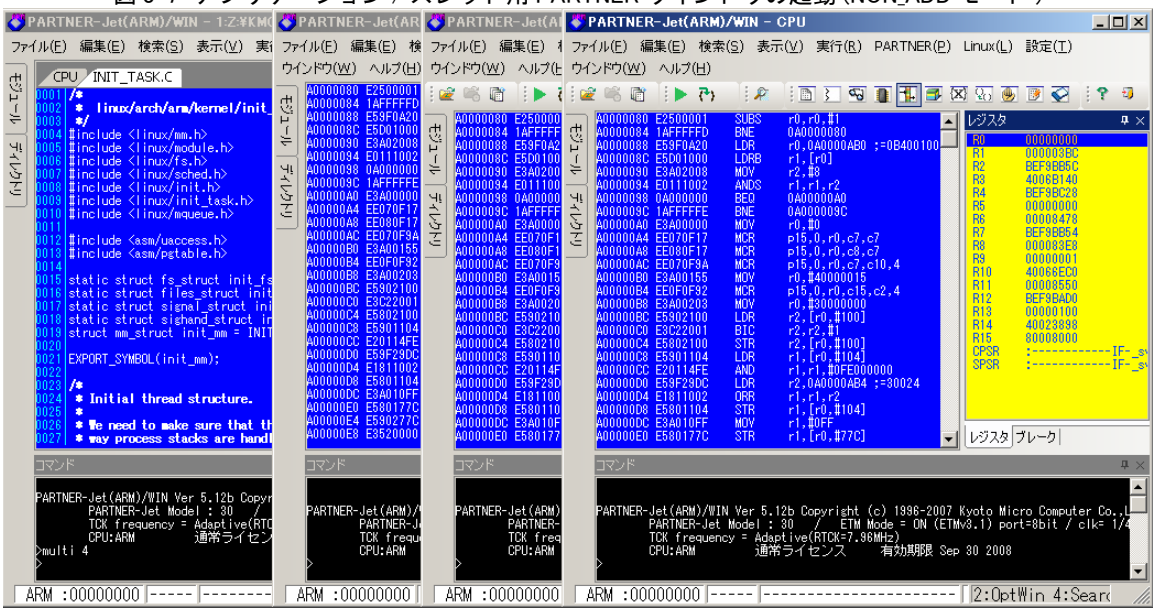
● NON_ADD モードの場合

ADD モードでない場合は、「カーネル＋アプリケーション＋生成されるプロセスの数」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 3-7 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（163 頁）や -MULTI オプション（145 頁）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 (JETARM_1.JPX など) に保存され、次回起動時に利用されます。

(4) アプリケーションの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。

● 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。

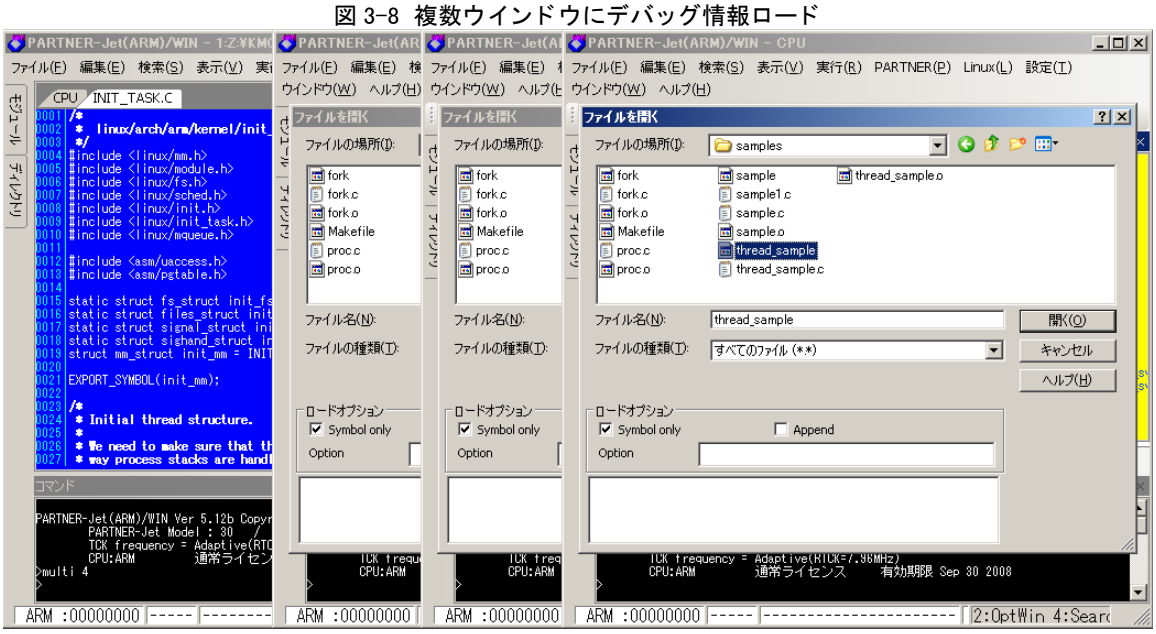
```
PT2>ls thread sample ↓
```

● 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。

```
PT2>ls thread sample ↓
PT3>ls thread sample ↓
PT4>ls thread sample ↓
```

図 3-8 複数ウインドウにデバッグ情報ロード



(5) ブレークポイントの設定

アプリケーションの main() 関数に実行型ハードウェアブレークポイントを設定します。

```
PT2>br main.ex ↓
```

(6) 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 ↓
```

(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実行されると、PARTNER は設定したハードウェアブレークポイントの位置 (main 関数) でブレークします。

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。

---

(8) アプリケーションのアタッチと確認

ATTACH コマンド （152 頁）を使用してアプリケーションにアタッチします。PARTNER はアプリケーションの PID や配置情報を獲得し、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。

```
PT2>ps ↓
  1(0x1)        /bin/busybox
  398(0x18e)    /sbin/udevd
  557(0x22d)    /bin/bash
  558(0x22e)    /bin/busybox
  559(0x22f)    /bin/busybox
  740(0x2e4)    /root/thread_sample
PT2>attach 740 ↓
```

または

```
PT2>attach thread sample ↓
```

PSID コマンド（158 頁）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できます。

```
PT2>psid
PSID SET 740(0x2E4)  CURRENT 740(0x2E4)
APPLI. AREA : 00008000-00009FFF
APPLI. AREA : 00012000-00012FFF
APPLI. AREA : 00015000-00015FFF
APPLI. AREA : BE913000-BE913FFF
APPLI. AREA : 40016000-40016FFF
```

（表示される内容は CPU 種別や MAP フィールドの設定によって異なります）

このときに、共有ライブラリのデバッグもできるように、LINUX コマンド（154 頁）も入力することをお勧めします。

```
PT2>linux load so ↓
```

● ADD モードの場合

main() 関数内や thread_body() 関数内にソフトウェアブレークポイントを設定します。実行し、ソフトウェアブレークポイントの箇所を通過するときにブレークがかかります。複数のスレッドコンテキストで実行されるアドレスならば、各スレッドでブレークします。
PSID コマンド（158 頁）でブレークした実行コンテキストを確認できます。

```
PT2>psid ↓
PSID SET 741(0x2E5)  CURRENT 741(0x2E5) [ADD MODE]
APPLI. AREA : 00008000-00009FFF
APPLI. AREA : 00012000-00012FFF
APPLI. AREA : 00015000-00015FFF
APPLI. AREA : BEA0D000-BEA0DFFF
APPLI. AREA : 40016000-40016FFF
PT2>BP 「THREAD_SAMPLE.C]thread_body+11 ↓
PT2>g ↓
        R0/R8    R1/R9    R2/R10   R3/R11    R4/R12   R5/R13   R6/R14   R7
R0-7 :0000000C 40083620 00000000 00000000  BE3FFE20 000046CC 00000020 4002A008
R8-14:BE3FFE20 00000000 40016A80 BE3FFD54  40029F10 BE3FFD18 000091E8
 PC  :000091E8 CPSR    :-ZC------------_usr
 PID :743(0x2E7)
thread_body+34(arg=*BEA0DAC0)
Real Time Count = 0,000s002m200u

PT2>psid ↓
PSID SET 743(0x2E7)  CURRENT 743(0x2E7) [ADD MODE 741]
APPLI. AREA : 00008000-00009FFF
APPLI. AREA : 00012000-00012FFF
APPLI. AREA : 00015000-00017FFF
APPLI. AREA : BE3FF000-BEA0DFFF
APPLI. AREA : 40016000-40016FFF
PT2>ps ↓
  1(0x1)        /bin/busybox
  398(0x18e)    /sbin/udevd
  571(0x23b)    /bin/bash
  572(0x23c)    /bin/busybox
  573(0x23d)    /bin/busybox
  741(0x2e5)    /root/thread_sample
  742(0x2e6)    /root/thread_sample
  743(0x2e7)    /root/thread_sample
  744(0x2e8)    /root/thread_sample
  745(0x2e9)    /root/thread_sample
  746(0x2ea)    /root/thread_sample
PT2>g ↓
        R0/R8    R1/R9    R2/R10   R3/R11    R4/R12   R5/R13   R6/R14   R7
R0-7 :0000000C 40083620 00000000 00000000  BE1FFE20 000046CC 00000030 4002A008
R8-14:BE1FFE20 00000000 40016A80 BE1FFD54  40029F10 BE1FFD18 000091E8
 PC  :000091E8 CPSR    :-ZC------------_usr
 PID :744(0x2E8)
thread_body+34(arg=*BEA0DAC4)
```

```
Real Time Count = 0,000s000m400u

PT2>psid ↓
PSID SET 744(0x2E8)  CURRENT 744(0x2E8) [ADD MODE 741,743]
APPLI. AREA : 00008000-00009FFF
APPLI. AREA : 00012000-00012FFF
APPLI. AREA : 00015000-00017FFF
APPLI. AREA : BE1FF000-BEA0DFFF
APPLI. AREA : 40016000-40016FFF
PT>
```

● NON_ADD モードの場合

メイン用のウインドウで main() 関数内にソフトウェアブレークポイントを設定し、スレッド用のウインドウで thread_body() 関数内にソフトウェアブレークポイントを設定します。

PSID コマンド（158 頁）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 ↓     （カーネル用のウインドウ）
PSID ****
PT2>psid ↓  （メイン用のウインドウ）
PSID SET 740(0x2E4)  CURRENT 740(0x2E4)
APPLI. AREA : 00008000-00009FFF
APPLI. AREA : 00012000-00012FFF
APPLI. AREA : 00015000-00015FFF
APPLI. AREA : BE913000-BE913FFF
APPLI. AREA : 40016000-40016FFF
PT3>psid ↓  （スレッド 1 用のウインドウ）
PSID ****
PT4>psid ↓  （スレッド 2 用のウインドウ）
PSID ****
（表示される内容は CPU 種別や MAP フィールドの設定によって異なります）
```

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 ウインドウがスレッドのエントリーでブレークします。このとき、スレッドの PID や配置情報が自動的に収集されており、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な状態になっています（生成されたスレッドに対するアタッチ操作不要）。

PSID コマンド（158 頁）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。

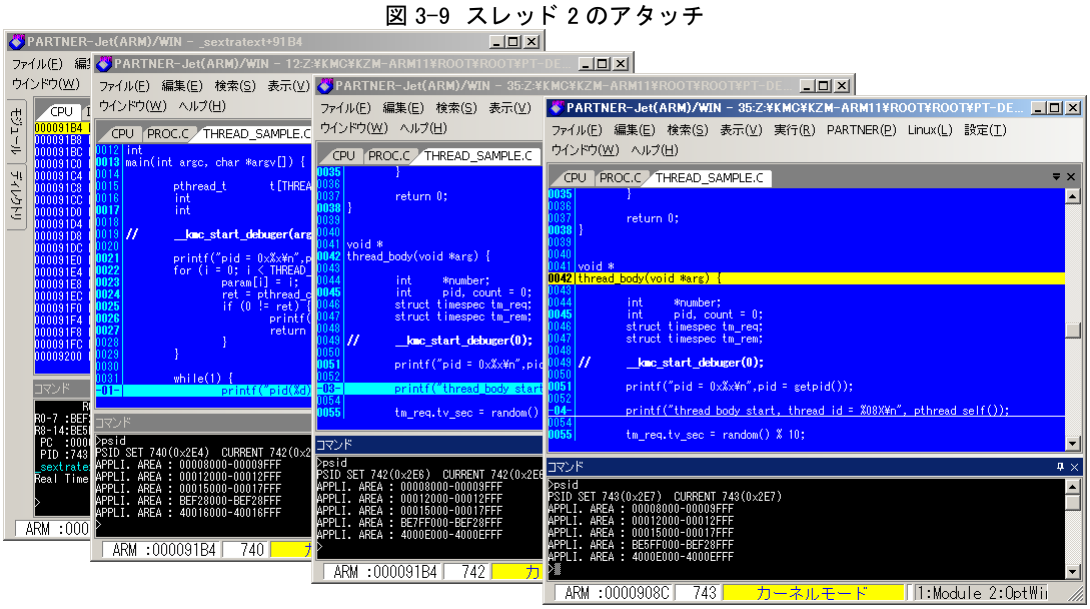
```
PT3>psid ↓
PSID SET 742(0x2E6)  CURRENT 742(0x2E6)
APPLI. AREA : 00008000-00009FFF
APPLI. AREA : 00012000-00012FFF
APPLI. AREA : 00015000-00017FFF
APPLI. AREA : BE1FF000-BE913FFF
APPLI. AREA : 4000E000-4000EFFF
PT3>g
```

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 ウインドウがスレッドのエントリーでブレークします。このとき、スレッドの PID や配置情報が自動的に収集されており、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な状態になっています（生成されたスレッドに対するアタッチ操作不要）。
PSID コマンド（158 頁）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。

```
PT4>psid ↓
PSID SET 743(0x2E7)  CURRENT 743(0x2E7)
APPLI. AREA : 00008000-00009FFF
APPLI. AREA : 00012000-00012FFF
APPLI. AREA : 00015000-00017FFF
APPLI. AREA : BDFFF000-BE913FFF
APPLI. AREA : 4000E000-4000EFFF
```

（表示される内容は CPU 種別や MAP フィールドの設定によって異なります）

図 3-9 スレッド 2 のアタッチ



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。『2.2.6 カーネルコンフィグレーション（54 頁）』の設定と『2.4.2 PARTNER の設定と起動（70 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、サポートライブラリ (libkmcsup.so.2.0.0) が Preload されているか確認してください。

3.6 マルチプロセスアプリケーションデバッグ

カーネルモードでマルチ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。fork() を使用したアプリケーションのデバッグ手順は前述の『3.4 アプリケーションのデバッグ（91 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。

マルチスレッドもマルチプロセスも平行動作する複数の実行コンテキストを扱いますが、マルチスレッドアプリケーションと異なりマルチプロセスアプリケーションのデバッグには ADD モードと NON_ADD モードの違いは関係ありません。
PARTNER の ADD モードはスレッドのデバッグにしか効きません。異なるプロセスをデバッグするときには、それぞれのプロセス毎に PARTNER ウインドウを開く必要があります。

3.6.1 デバッグに必要な設定条件

表 3-4 マルチプロセスアプリケーションデバッグの条件

* 設定条件 / デバッグ対象	カーネルメニュー						-OS オプション			
	Debug infomation type	Enable patch for PARTNER debug	Loadable module debug by PARTNER-Jet (hook in kernel side)	Loadable module debug by PARTNER-Jet (hook in module side)	use PARTNER GigaTrace (PARTNER-Jet M40, ...)	enable PARNTER-Jet Event Tracker	カーネルモード	カーネル ADD モード	アプリケーションモード	アプリケーション ADD モード
アプリケーション	◎	◎					○†		—‡	

* ◎：必須、○：推奨、△：非推奨、×：不可、空欄：どちらでもよい
† ADD モード、NON_ADD モードどちらでもデバッグできますが、動作に違いはありません。
‡ アプリケーションモードのデバッグ方法は『5.1 アプリケーションモードデバッグ（172 頁）』を参照してください。

『2.2 Linux カーネルソースの修正と設定（50 頁）』を行っていない場合は、ここで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、『5.3 手動アプリケーションデバッグ（191 頁）』を参照してください。

3.6.2 デバッグの手順

この節では、サンプル (fork) 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アプリケーションのデバッグ手順を説明します。

(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。
アプリケーションのソースコードは全く修正する必要がありません。

【例】

```
LINUX86> linux-arm-gcc -o fork -g fork.c ↓
```

デバッグサポート用ファイルを『Preload ライブラリ方式（60 頁）』で使用する場合、デバッグサポートライブラリ（libkmcsup.so）は**アプリケーションにリンクしないでください。**（ターゲット上で ldd コマンドを使用すればリンクされている共有ライブラリを確認できます。）

(2) デバッグの準備

『デバッグ環境の起動（68 頁）』を参照し、カーネルモードの PARTNER で Linux のデバッグができる状態にします。

(3) アプリケーション用 PARTNER ウインドウを開く

アプリケーション用 PARTNER ウインドウを起動します。
「カーネル＋アプリケーション＋生成されるプロセスの数」の PARTNER ウインドウが必要です。
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 ↓
```

ADD モードか NON_ADD モードかにかかわらずプロセス数の PARTNER ウインドウが必要です。
ただし、同時にデバッグしたいプロセスがひとつだけの場合は、カーネルと同じウインドウでデバッグしてもかまいません。

(4) アプリケーションの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。
アプリケーション用 PARTNER ウインドウと子プロセス用 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。

```
PT2>ls fork ↓
PT3>ls fork ↓
```

図 3-10 複数ウインドウにデバッグ情報ロード

(5) ブレークポイントの設定

アプリケーションの main() 関数に実行型ハードウェアブレークポイントを設定します。

```
PT2>br main.ex ↓
```

(6) 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 ↓
```

(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実行されると、PARTNER は設定したハードウェアブレークポイントの位置 (main 関数) でブレークします。

**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。**

---

(8) アプリケーションのアタッチと確認

ATTACH コマンド （152 頁）を使用してアプリケーションにアタッチします。PARTNER はアプリケーションの PID や配置情報を獲得し、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。

```
PT>ps ↓
  1(0x1)        /bin/busybox
  398(0x18e)    /sbin/udevd
  557(0x22d)    /bin/bash
  558(0x22e)    /bin/busybox
  559(0x22f)    /bin/busybox
  740(0x2e4)    /root/fork
PT>attach 740 ↓
または
PT>attach fork ↓
```

PSID コマンド（158 頁）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できます。

```
PT>psid
PSID SET 740(0x2E4)  CURRENT 740(0x2E4)
APPLI. AREA : 00008000-00009FFF
APPLI. AREA : 00012000-00012FFF
APPLI. AREA : 00015000-00015FFF
APPLI. AREA : BE913000-BE913FFF
APPLI. AREA : 40016000-40016FFF
（表示される内容は CPU 種別や MAP フィールドの設定によって異なります）
```

このときに、共有ライブラリのデバッグもできるように、LINUX コマンド（154 頁）も入力することをお勧めします。

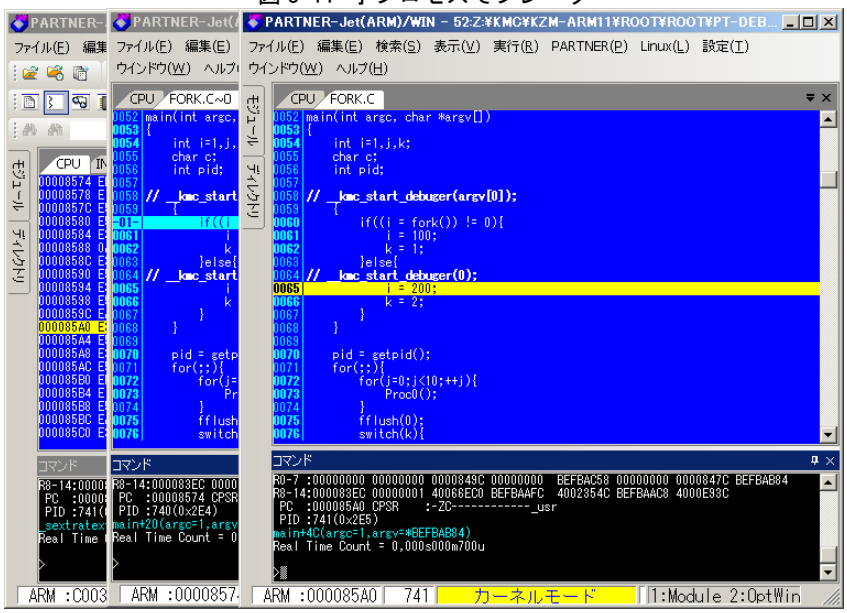
```
PT>linux load so ↓
```

PSID コマンド（158 頁）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 ↓    （カーネル用のウインドウ）
PSID ****
PT2>psid ↓ （メイン用のウインドウ）
PSID SET 740(0x2E4)  CURRENT 741(0x2E5) [ADD MODE]
APPLI. AREA : 00008000-00009FFF
APPLI. AREA : 00011000-00011FFF
APPLI. AREA : BEFBA000-BEFBAFFF
PT3>psid ↓ （子プロセス用のウインドウ）
PSID ****
（表示される内容は CPU 種別や MAP フィールドの設定によって異なります）
```

次に、アプリケーション用 PARTNER ウインドウで G または G/A コマンドを使用してアプリケーションを再開し、fork() 関数が実行されプロセスが生成されると、子プロセス用の PARTNER ウインドウで fork() の子プロセス側の処理でブレークします。このとき、スレッドの PID や配置情報が自動的に収集されており、以後子プロセスの空間のメモリ参照やブレークポイントの設定が可能な状態になっています（生成されたプロセスに対するアタッチ操作不要）。

図 3-11 子プロセスでブレーク



PSID コマンド（158 頁）を実行すると子プロセス用 PARTNER ウインドウでプロセスがアタッチされていることが確認できます。

```
PT3>psid ↓
PSID SET 741(0x2E5)  CURRENT 741(0x2E5) [ADD MODE]
APPLI. AREA : 00008000-00009FFF
APPLI. AREA : 00011000-00011FFF
APPLI. AREA : BEFBA000-BEFBAFFF
PT3>g
```

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。『2.2.6 カーネルコンフィグレーション（54 頁）』の設定と『2.4.2 PARTNER の設定と起動（70 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、サポートライブラリ (libkmcsup.so.2.0.0) が Preload されているか確認してください。

3.7 共有ライブラリのデバッグ

共有ライブラリは複数のプロセスが使用するプログラムモジュールですが、実行中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。

3.7.1 デバッグに必要な設定条件

デバッグの対象としてはアプリケーションのプロセスになります。必要な設定条件はアプリケーションのデバッグと同じです。
共有ライブラリのデバッグを行う前に、アプリケーションをアタッチしておく必要があります。

3.7.2 デバッグの手順

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手順を説明します。

(1) 共有ライブラリにデバッグ情報をつける（115 頁）
(2) アプリケーションのアタッチ（115 頁）
(3) 共有ライブラリのデバッグ情報読み込み（115 頁）

(1) 共有ライブラリにデバッグ情報をつける

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。

(2) アプリケーションのアタッチ

『3.4 アプリケーションのデバッグ (91 頁)』、『3.5 マルチスレッドアプリケーションデバッグ (98 頁)』、『3.6 マルチプロセスアプリケーションデバッグ (108 頁)』、『5.1 アプリケーションモードデバッグ (172 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。

(3) 共有ライブラリのデバッグ情報読み込み

アプリケーションにアタッチしている状態で LINUX コマンド (154 頁) を使用します。

```
PT>linux load so ↓
```

明示的にデバッグ情報をロードする場合は LSA コマンドを使用します。GUI メニューからロード時には必ず [Symbol only] と [Append] のチェックを付けてください。

```
PT>lsa libc-2.2.5.so ↓
```

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 (158 頁) 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 99(0x63)  CURRENT -1(0xFFFFFFFF)
APPLI. AREA : 00008000-00009FFF
APPLI. AREA : 00011000-00011FFF
APPLI. AREA : 00013000-00013FFF
APPLI. AREA : BFFFF000-BFFFFFFF
APPLI. AREA : 4001F000-4012AFFF
APPLI. AREA : 4012F000-40137FFF
APPLI. AREA : 40000000-40015FFF
```

(表示される内容は CPU 種別や MAP フィールドの設定によって異なります)

3.7.3 共有ライブラリデバッグの注意

プログラムの実行の観点から見ると共有ライブラリはアプリケーションの実行時にリンク・アドレス解決されるライブラリととらえることができます。共有ライブラリ内のコードはアプリケーションのプロセスで実行されるためアプリケーションのプロセスにアタッチすることでデバッグできるようになります。
しかし、アプリケーションの本体やスタティックリンクのライブラリとは動作に違いがありますのでデバッグの際の注意点を記載します。

共有ライブラリ内で生成されるスレッド・プロセス

PARTNER のアプリケーションデバッグサポートファイル（kmc-support.c）を共有ライブラリ（libkmcsup.so）としてビルドする場合では、pthread_create() 関数や fork() 関数にデバッガが実行コンテキストの生成を認識するためのコードを埋め込むことで実現されています。
つまり共有ライブラリ内で pthread_create() 関数や fork() 関数を使用している場合、libkmcsup.so ライブラリ内の pthread_create() 関数や fork() 関数が実行されなくてはなりません。デバッグサポートファイルを Preload ライブラリとして使用している場合には pthread ライブラリに入っている pthread_create() 関数や libc ライブラリに入っている fork() 関数よりも先に libkmcsup.so ライブラリ内の pthread_create() 関数や fork() 関数が実行されるはずです。
もし、共有ライブラリ内で生成されるスレッドやプロセスのデバッグがうまくいかない場合は libkmcsup.so ライブラリ内の関数が呼ばれているかどうか確認してください。
一方、サポートファイル（kmc-support.c）をアプリケーションに直接リンクしている場合は pthread_create() 関数や fork() 関数を使用する箇所にスタブ関数（_kmc_start()）を埋め込む必要がありますが、共有ライブラリはアプリケーションとは別にビルドされているため、アプリケーションのソースコード内にスタブ関数を埋め込むだけでは不十分な場合があります。

共有ライブラリ内のブレークポイント

共有ライブラリ内のコードは複数のプロセスで実行される可能性があります。共有ライブラリ内のコードにブレークポイントを設定した場合、デバッグ中のプロセスとは別のプロセスでブレークする可能性があります。PARTNER はデバッグ中のプロセス ID と異なるプロセスでブレークした場合は自動的に再実行しますので操作上の見た目は停止しているようには見えません。実際には CPU 停止と再実行が行われていますので、多くのプロセスから参照されている箇所の場合、動作が遅くなることがあります。

3.8 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。

3.8.1 必要な設定条件

表 3-5 Linux OS 対応ヒストリ表示の条件

* 設定条件 / デバッグ対象	カーネルメニュー						-OS オプション			
	Debug infomation type	Enable patch for PARTNER debug	Loadable module debug by PARTNER-Jet (hook in kernel side)	Loadable module debug by PARTNER-Jet (hook in module side)	use PARTNER GigaTrace (PARTNER-Jet M40, ...)	enable PARNTER-Jet Event Tracker	カーネルモード	カーネル ADD モード	アプリケーションモード	アプリケーション ADD モード
アプリケーション	◎	◎					○	○	× †	

* ◎：必須、○：推奨、△：非推奨、×：不可、空欄：どちらでもよい
† Linux OS 対応ヒストリ表示は、「カーネルモード」「カーネル ADD モード」で使用してください。

3.8.2 デバッグの手順

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) LinuxOS 対応ヒストリ表示に必要なカーネルのコンフィグレーション（118 頁）
(2) アプリケーションの準備（118 頁）
(3) Linux OS 対応ヒストリ表示（118 頁）

(1) LinuxOS 対応ヒストリ表示に必要な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 [Debug infomation type] と [Enable patch for PARTNER debug] を有効にします。

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 ↓
```

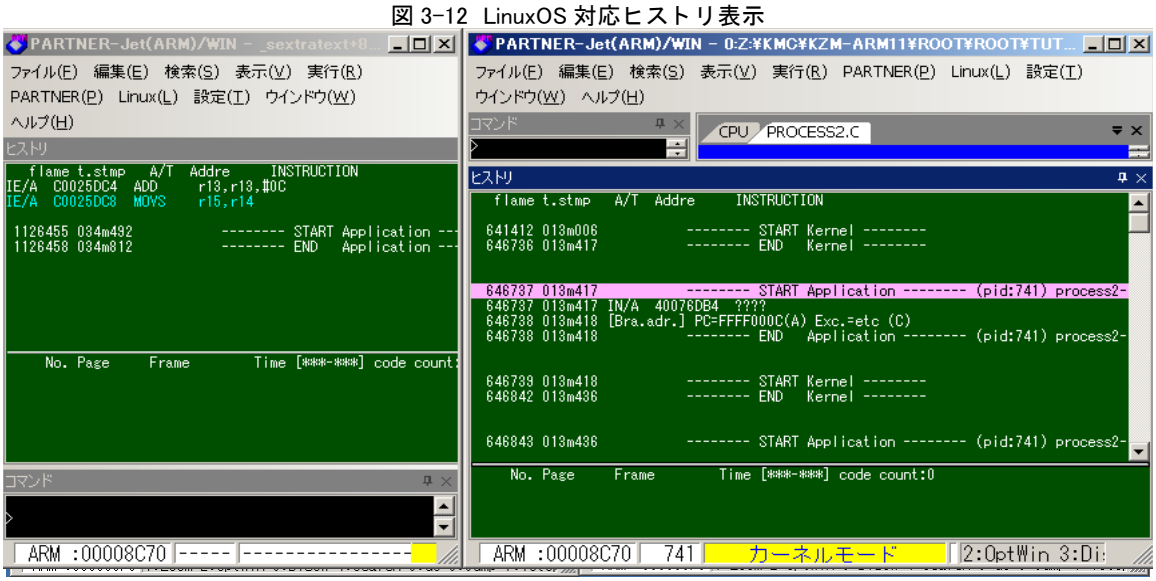
(2) アプリケーションの準備

『3.4 アプリケーションのデバッグ (91 頁)』を参考にしてデバッグ環境を整えます。

(3) 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 3-12 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
```

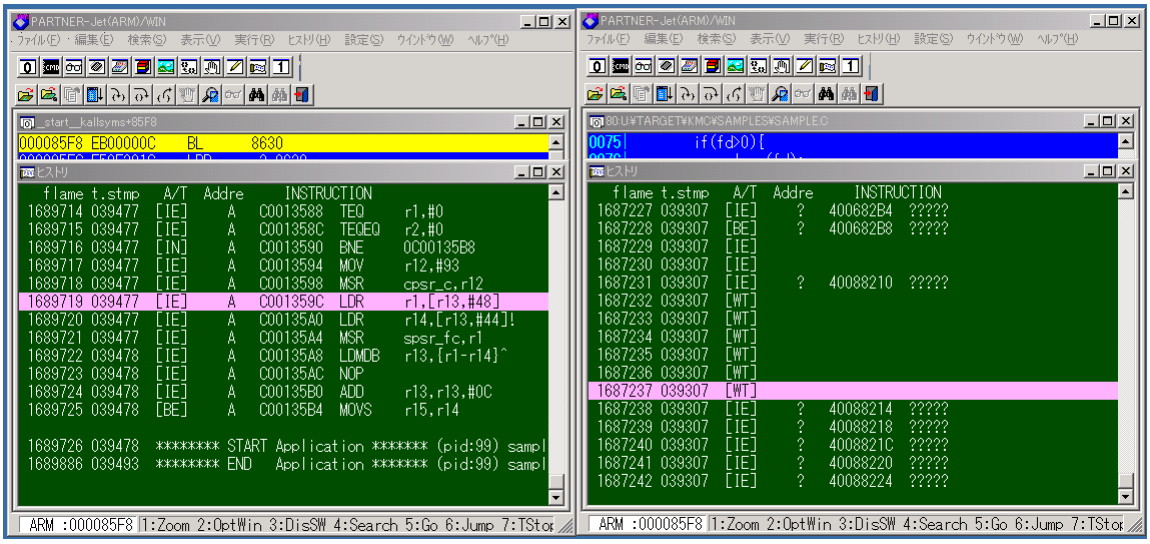
PARTNER のバージョンによって、「** START Kernel ****」「**** END Kernel ****」のように多少表示が異なることがあります。**

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、????? 表示されている箇所があります。これは、共有ライブラリでメモリにアクセスできない箇所の場合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。

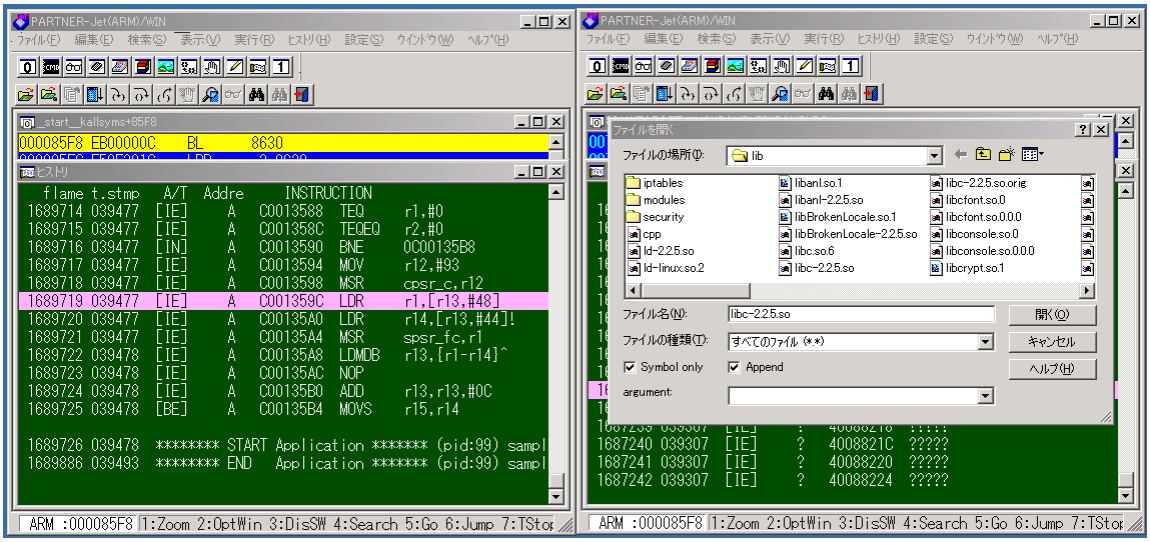
PARTNER の起動設定の –OS オプション（138 頁）で K2 プレフィクスを使用している場合には自動解決されて ?????? にならないこともあります）。

図 3-13 不明なヒストリ表示



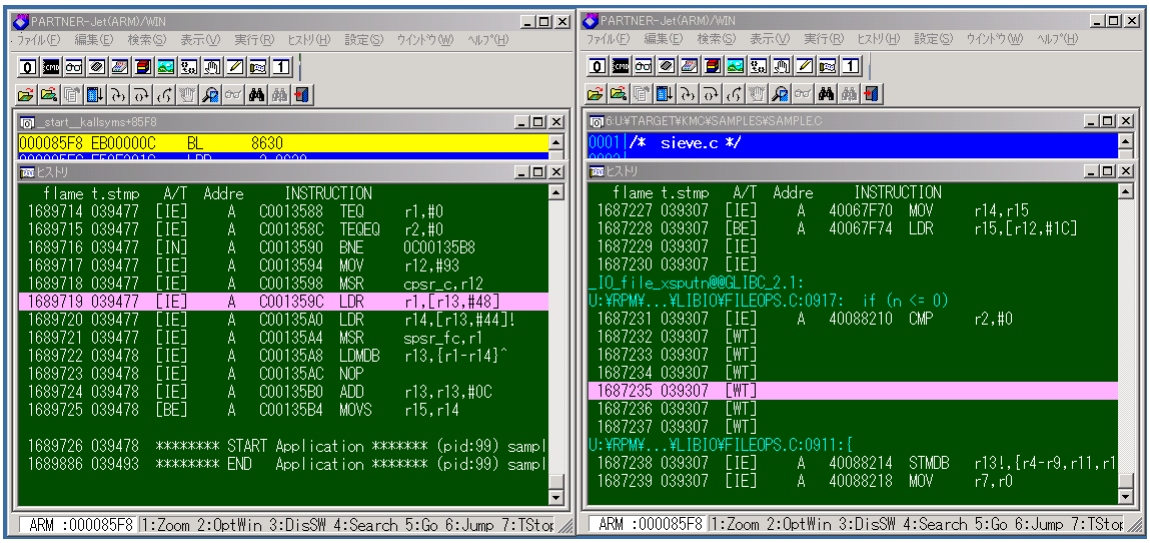
????? 表示を解消するには、該当アドレスの空間のデバッグ情報を読み込みます。MAPS コマンド（151 頁）で確認できます）

図 3-14 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、????? 表示だった箇所が解消され、通常表示になります。

図 3-15 ????? 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（163 頁）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。

(4) 実行例

共有ライブラリを作る（32 頁）で使用したサンプルプログラム pthread2.c（33 頁）を使用して実際に実行した際の動作を Linux が起動した状態から説明します。

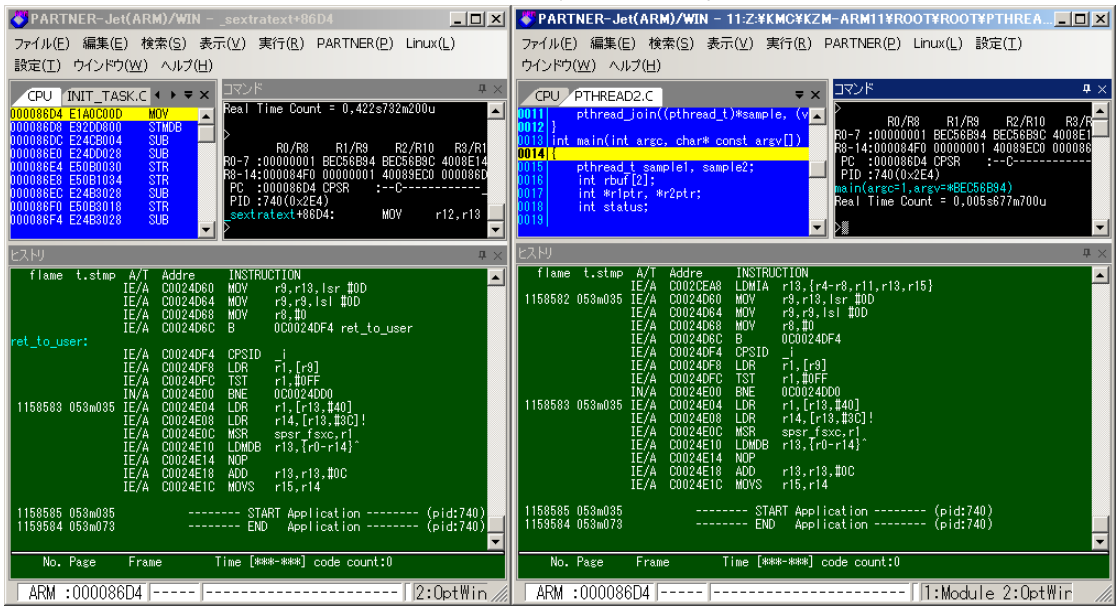
アプリケーションの先頭からデバッグを開始します。

```
PT>multi 2 ↓
PT2>ls pthread2 ↓
PT2>brc * ↓
PT2>br main.ex ↓
PT2>g ↓
TGT>./pthread2 ↓
```

（main でハードウェアブレークポイントにより停止します）

ヒストリ欄をクリックするとトレース結果が表示されますが、まだプロセスにアタッチしていないためアプリケーションプロセスのシンボル情報は表示されません。

図 3-16 ヒストリ表示（main で停止）



アプリケーションのプロセスにアタッチします。

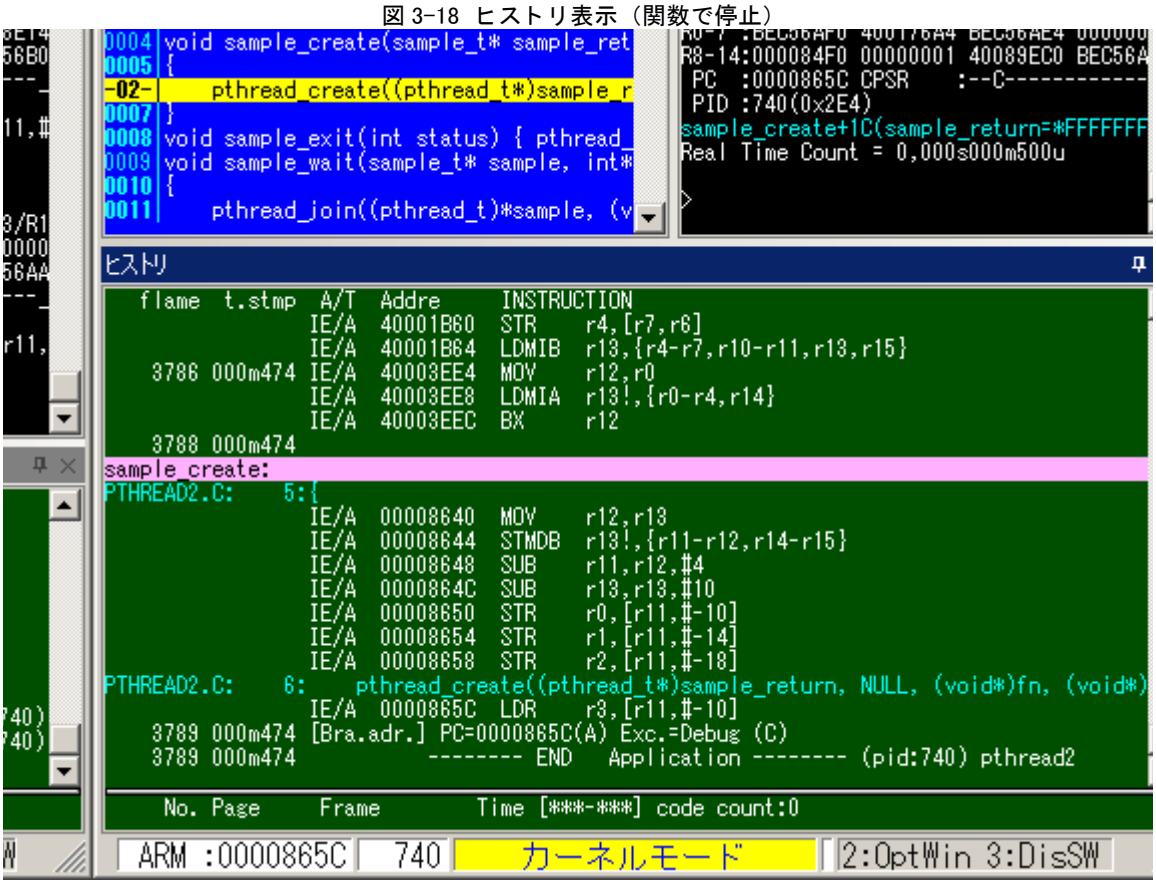
```
PT2>ps ↓
PT2>attach 740 ↓
```

アタッチした状態でトレース結果を表示するとアプリケーションのシンボル情報が表示されます。

図 3-17 ヒストリ表示（プロセスにアタッチ済）

PT2>BP .sample_create+1 ↓
PT2>g ↓

図 3-18 ヒストリ表示（関数で停止）

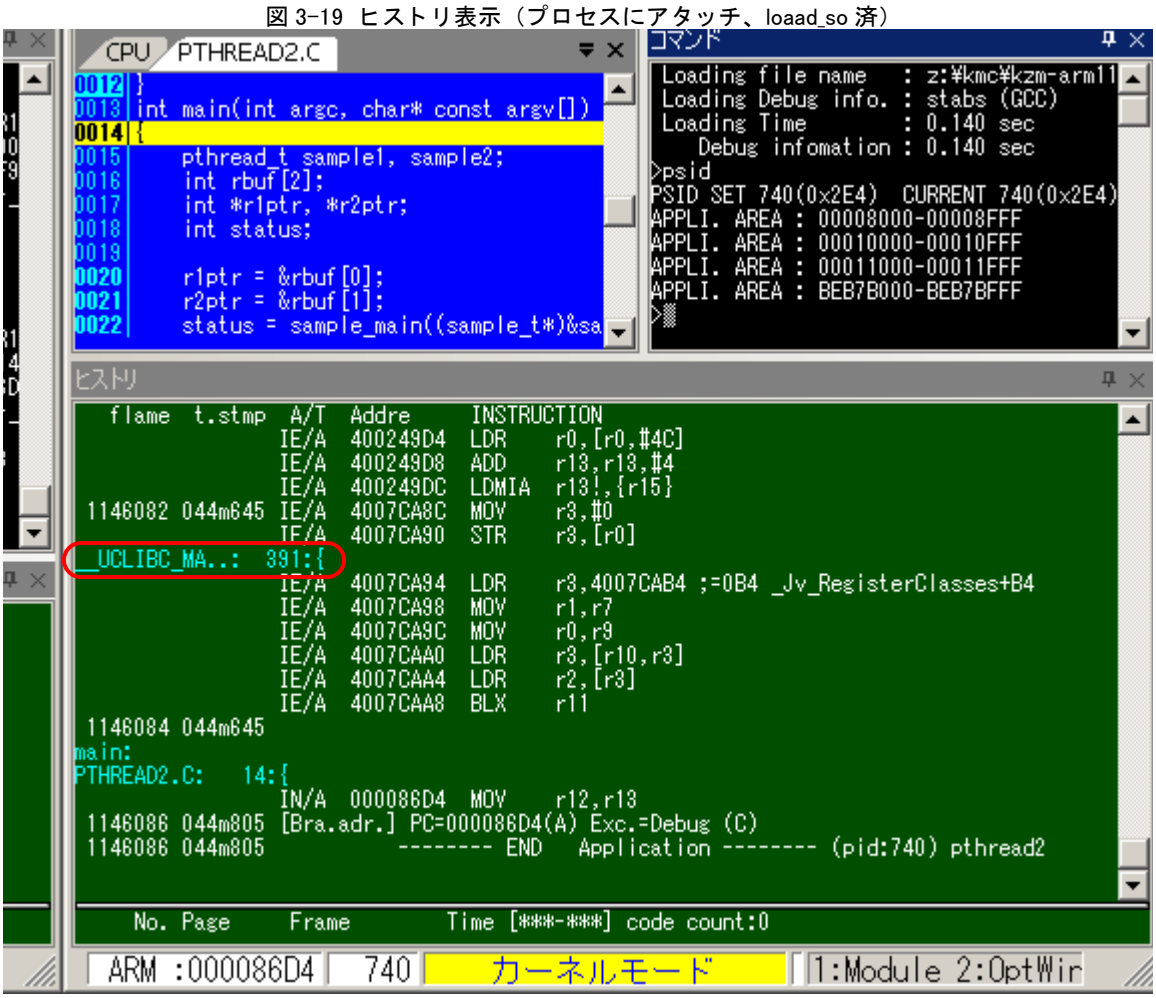


このプログラムでは sample_create() 関数は共有ライブラリ libsample.so から呼ばれますが、共有ライブラリ内のシンボルは図 3-18 では表示されていません。プロセスにアタッチするとともに LINUX コマンド（154 頁）で load_so 指定をしておくと共有ライブラリの情報も表示されます。

再度アプリケーションの先頭からアプリケーションの先頭からデバッグを開始する手順を示します。

PT>multi 2 ↓
PT2>ls pthread2 ↓
PT2>brc * ↓
PT2>br main.ex ↓
PT2>g ↓
TGT>./pthread2 ↓
（main でハードウェアブレークポイントにより停止します）
PT2>ps ↓
PT2>attach 740 ↓
（アプリケーションのプロセスにアタッチします）
PT2>linux load so ↓
（共有ライブラリの情報をロードします）

図 3-19 ヒストリ表示（プロセスにアタッチ、loaad_so 済）



この時点で libc など使用している共有ライブラリでシンボル情報を読み出し可能なものは表示されます。

さらに、sample_create() 関数にブレークポイントを設定して実行します。

PT2>BP .sample_create+1 ↓
PT2>g ↓

図 3-20 ヒストリ表示（共有ライブラリから呼ばれる関数で停止）

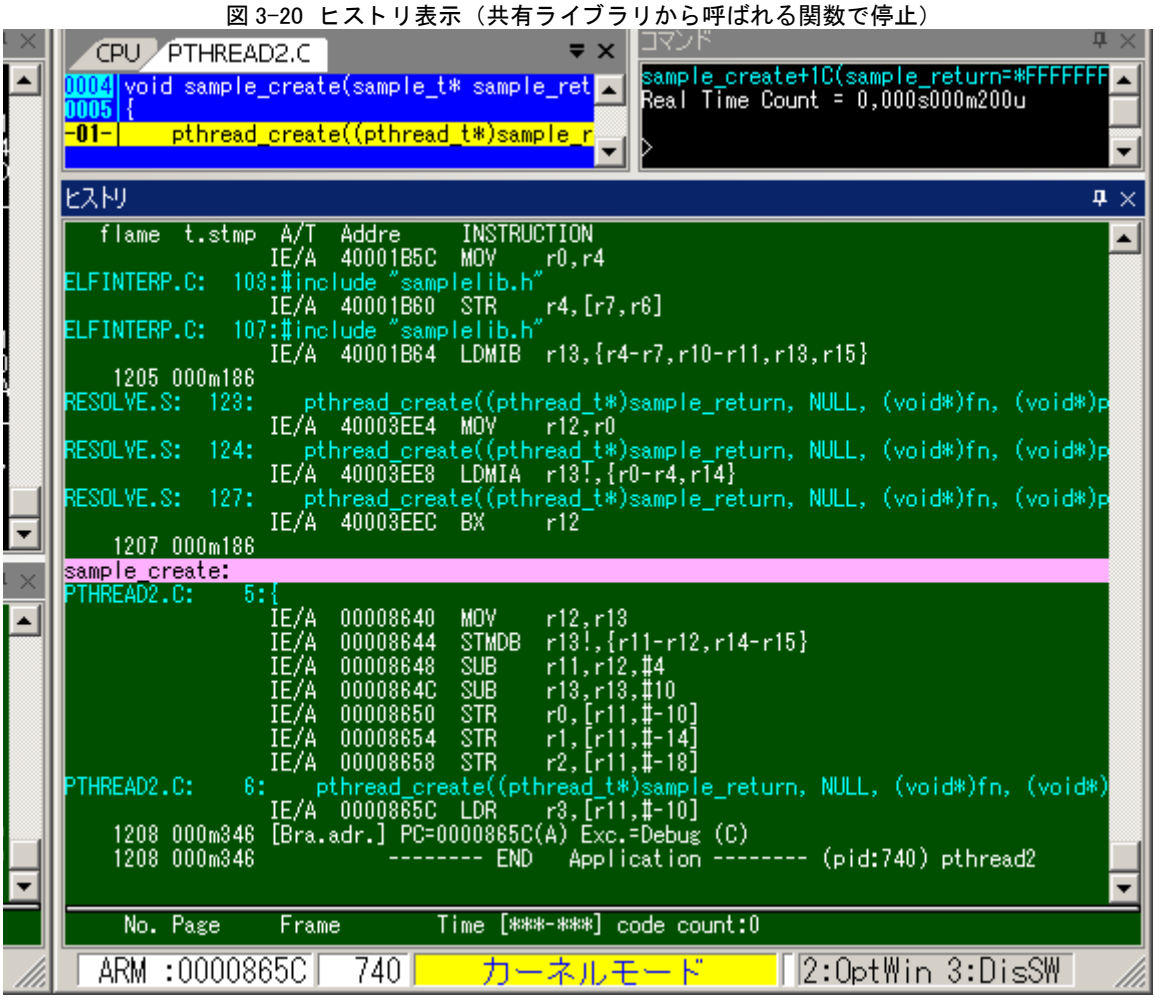


図 3-18 と図 3-20 を比べてみてください。sample_create() 関数で停止した状態でのヒストリ表示で、図 3-20 では共有ライブラリ libsample.so に含まれるシンボル情報が表示されていることがわかります。

3.9 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。
このことは通常のアプリケーションのデバッグ（『アプリケーションのデバッグ（91 頁）』、『アプリケーションモードデバッグ（172 頁）』参照）でプログラムの先頭からデバッグする場合と、技術的には特別な違いがあるわけではありません。
アプリケーションの実行前にあらかじめ main などのアプリケーション内のシンボルにブレークポイントを設定しておくことは必須ではないというだけのことです。

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

(1) デバッグの準備

実行中のアプリケーションをアタッチするためには、デバッグサポートファイル kmc-support.c の機能が共有ライブラリになっている必要があります。
共有ライブラリ形式のデバッグサポートファイルは _kmc_sleep_thread シンボルのアドレスを LINUX コマンド（154 頁）で PARTNER へ登録する必要があります。

【例】

```
LINUX86>nm <ライブラリ名> | grep  kmc_sleep_thread ↓
000f2b54 t __kmc_sleep_thread
PT>LINUX set attach offset <ライブラリ名> <  kmc_sleep_thread のアドレス> ↓
```

詳細は『Preload ライブラリ方式（60 頁）』参照）または『サポートファイルを glibc に入れる方法（236 頁）』）を参照してください。

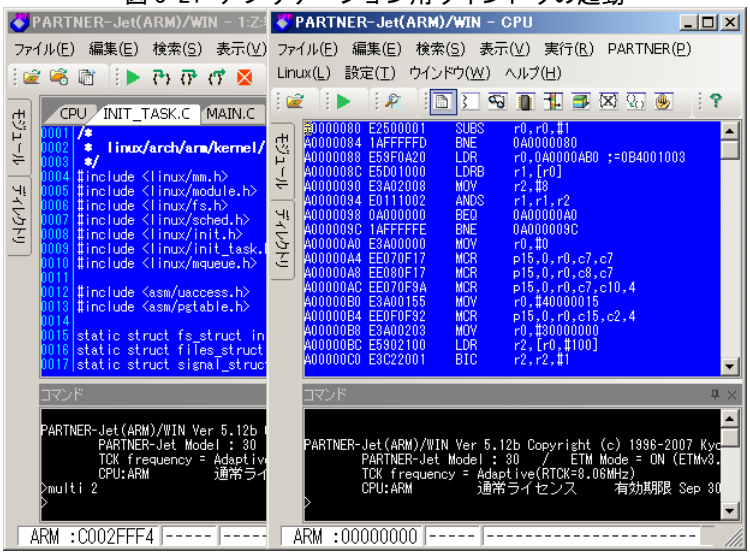
『3.4 アプリケーションのデバッグ（91 頁）』もしくは『5.1 アプリケーションモードデバッグ（172 頁）』を参照し、PARTNER からカーネルを実行し、デバッグ対象アプリケーションも実行しておきます。

(2) アプリケーション用 PARTNER ウインドウの起動

MULTI コマンド（163 頁）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 3-21 アプリケーション用ウインドウの起動



MULTI コマンド（163 頁）や -MULTI オプション（145 頁）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（JETARM_1.JPX など）に保存され、次回起動時に利用されます。

(3) 実行しているアプリケーションの PID の確認

PS コマンド（150 頁）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)        /bin/busybox
  398(0x18e)    /sbin/udevd
  512(0x200)    /bin/bash
  513(0x201)    /bin/busybox
  514(0x202)    /bin/busybox
  740(0x2e4)    /root/sample
```

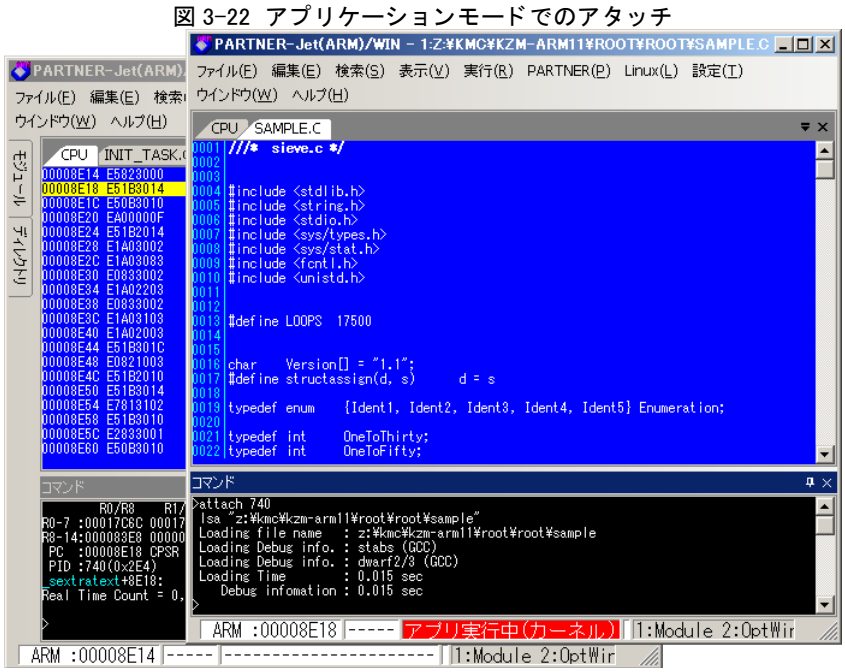
(4) 実行しているアプリケーションのアタッチ

アプリケーション用 PARTNER ウインドウで ATTACH コマンド（152 頁）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 740 ↓
```

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 3-22 アプリケーションモードでのアタッチ



アプリケーションモードでの動作については本書内でまだ説明していませんので、後述の『5.1 アプリケーションモードデバッグ（172 頁）』を参照してください。

PSID コマンド（158 頁）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 740(0x2E4)  CURRENT 740(0x2E4)
APPLI. AREA : 00008000-00009FFF
APPLI. AREA : 00011000-00015FFF
APPLI. AREA : BEBF6000-BEBF6FFF
（表示される内容は CPU 種別や MAP フィールドの設定によって異なります）
```

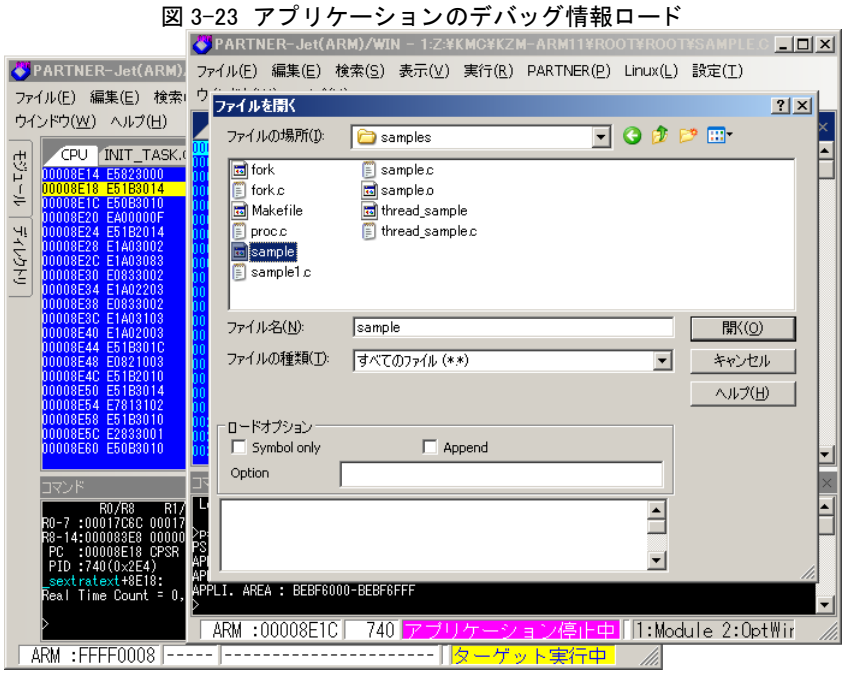
PARTNERが正しくブレークしない場合､カーネルのコンフィグレーションが間違っている可能性があります｡『2.2.6 カーネルコンフィグレーション（54 頁）』の設定と『2.4.2 PARTNER の設定と起動（70 頁）』で指定した OS デバッグモードがアプリケーションモードになっているか確認してください。
また、サポートライブラリ (libkmcsup.so.2.0.0) が Preload されているか確認してください。

(5) 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] のチェックを付けてください。

```
PT>ls sample ↓
```

図 3-23 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。

第 4 章　Linux 対応機能リファレンス

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。

4.1 CFG ファイルの拡張

Linux デバッグをサポートするため、以下の設定が追加 / 拡張されました。

・MAP フィールド（133 頁）

MAP フィールド

書式

MAP 開始アドレス , 終了アドレス [, 属性]

機能

PARTNER でアクセスするメモリ領域指定

解説

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。

書式にしたがい、論理アドレス領域とその領域の属性を指定します。

表 4-1 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（151 頁）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 ↓
00008000-0000f000 r-xp 00000000 00:07 137095     /sbin/init
00016000-00017000 rw-p 00006000 00:07 137095     /sbin/init
00017000-0001b000 rwxp 00000000 00:00 0
40000000-40016000 r-xp 00000000 00:07 133213     /lib/ld-2.2.5.so
4001d000-4001e000 rw-p 00015000 00:07 133213     /lib/ld-2.2.5.so
4001e000-4001f000 rwxp 00000000 00:00 0
4001f000-4012b000 r-xp 00000000 00:07 138215     /lib/libc-2.2.5.so
4012b000-4012f000 ---p 0010c000 00:07 138215     /lib/libc-2.2.5.so
4012f000-40138000 rw-p 00108000 00:07 138215     /lib/libc-2.2.5.so
40138000-4013c000 rw-p 00000000 00:00 0
bffff000-c0000000 rwxp 00000000 00:00 0
TGT>
```

CPU 種別や Linux カーネルバージョンにより表示は異なります。第 1 カラムが物理アドレスの範囲になっていますので 00000000,FFFFFFFF 以外の表示を設定します。

● 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。

ARM CPU の場合：
include/asm/arch/vmalloc.h で宣言されている VMALLOC_START から VMALLOC_END までの範囲を計算して決定します。
PARTNER でデバッグ情報付きの Linux カーネルをロード / 実行し、変数 high_memory の値が初期化された後ブレークし、変数 high_memory を PARTNER の VAL コマンドで確認します。

```
PT>val high memory ↓
(void *) *C1000000
```

MIPS/SH/AM33 CPU の場合：
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】

ARM CPU の場合：

```
MAP     00008000,bfffffff,APPLI                 ; -- Linux アプリケーション空間
MAP     c0000000,c17fffff,KERNEL                ; -- Linux カーネル空間
MAP     c1800000,c1ffffff,MODULE                ; -- Linux ローダブルモジュール空間
MAP     ffff0000,ffffffff,KERNEL                ; -- Linux カーネル空間

MAP     00000000,000fffff                       ; -- MMU OFF 時のアクセス可能空間
MAP     c0000000,c1ffffff                       ; -- MMU OFF 時のアクセス可能空間
```

MMU が ON の場合と OFF の場合の MAP 指定を正しく設定してください。まず、始めに MMU ON 状態（Linux の OS が動作している状態）での MAP 指定をしてください。その設定の後ろに、MMU OFF 状態（電源投入時またはたリセット直後の状態）での MAP 指定をしてください。ARM の Liunx 環境では、APPLI,KERNEL,MODULE の各メモリ空間は、ビルド環境によって異なる場合が多く、カーネルビルド時に作成される System.map や arc/arm/vmlinux.lds ファイルなどを参照して確認してください。

MIPS CPU の場合：

```
MAP     00400000,7FFFFFFF,APPLI
MAP     80000000,80FFFFFF,KERNEL
MAP     C0000000,CFFFFFFF,MODULE
MAP     9FC00000,9FFFFFFF
MAP     A0000000,A0FFFFFF
MAP     AA000000,ABFFFFFF
MAP     BF800000,BFFFFFFF
```

MIPS CPU は、0x80000000~0x9FFFFFFF の空間と 0xA0000000~0xBFFFFFFF 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。

SH CPU の場合：

```
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,afffffff
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。

AM33 CPU の場合：

```
MAP     00000000,07ffffff,KERNEL
MAP     08000000,6fffffff,APPLI
MAP     70000000,7fffffff,DRIVER
MAP     40000000,4fffffff
MAP     8e000000,dfffffff
```

MMU が ON の場合と OFF の場合の MAP 指定を正しく設定してください。まず、はじめに MMU ON 状態（Linux の OS が動作している状態）での MAP 指定をしてください。その設定の後ろに、MMU OFF 状態（電源投入時またはリセット直後の状態）での MAP 指定をしてください。

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。

4.2 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

書式　　-OS <デバッグモード>[,,4]

機能

デバッグモード指定

解説

本オプション指定は Linux ローダブルモジュールとアプリケーションデバッグのために必須です。

カーネルのみをデバッグする場合は必要ではありませんが、常に指定することを推奨します。

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。

推奨は「K2_LINUX_ADD」または「K2_LINUX_ADD_V26」です。

表 4-2 デバッグモード

デバッグモード	解説	備考
LINUX	カーネルモードを指定します。	Linux カーネル 2.4 系の場合
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。	

各デバッグモードには以下のプレフィクス、サフィクスを指定できます。

表 4-3 デバッグモード追加文字列

追加設定	解説	備考
K2_ プレフィクス	拡張仮想アドレス解決機能を使用します。	別の（カレントではない）プロセス内の変数をインスペクトできるようになります。
_V26 サフィクス	Linux カーネル 2.6 系対応を指定します。	デフォルトは Ver2.4 対応となります。

K2_ プレフィクスは通常は付けることを推奨します。拡張仮想アドレス解決機能の実現は PARTNER が Linux カーネル内の仮想アドレス変換テーブルを参照することで行われており、カーネルへの依存性が高い機能なので、使用することで挙動がおかしくなる場合は外してください。

Linux カーネルのバージョンは PARTNER によって自動判定されています。_V26 サフィクスを指定せずに 2.6 系のカーネルをデバッグしても通常は正しく動作します。まれに自動判定が効かないことがありますので、デバッグ対象のカーネルバージョンが判っている場合は指定してください。

【使用例】

-OS LINUX
-OS LINUX_ADD
-OS LINUX_V26
-OS LINUX_ADD_V26
-OS LINUX_APP_ADD_V26
-OS K2_LINUX_ADD_V26
-OS K2_LINUX_ADD_V26,,4（多くの SH CPU 用 Linux カーネルの場合）

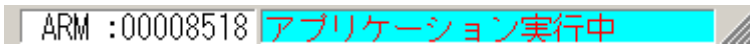
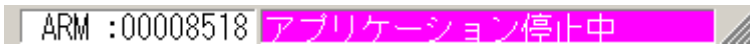
各デバッグモードの動作をまとめると表 4-4 のようになります。

表 4-4 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 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 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）

アプリケーションモードで PARTNER を使用する場合、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 4-5 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	

-XGX オプション

書式　　-XGX〈変換前 PATH〉,〈変換後 PATH〉

機能

デバッグ情報に含まれるパス名の解決

解説

本オプション指定は Linux デバッグのために必須です。

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

書式　　**-!v**

機能

PARTNER の複数ウインドウ起動設定

解説

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（145 頁）や MULTI コマンド（163 頁）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

書式　　-!!〈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〉

機能

PC 初期値設定
カーネルオプション文字列指定

解説

PC（プログラムカウンタ）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内で指定されている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、以下のデフォルトの値になります。

表 4-6 カーネルオプションシンボル名

CPU	デフォルト値	推奨値
ARM	kzp01_arm_command_line	--!!,default_cmdline="" 指定してください。
MIPS	arcs_cmdline	ボードによってかなり異なります。
SH	empty_zero_page に 0x100 を加えたアドレス	デフォルト値で問題ありません。
AM33	cmdline	

【使用例】

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

-SK オプション

書式　　-SK <ローダブルモジュール PATH>

機能

ローダブルモジュールパス指定

解説

ローダブルモジュールの PATH 情報を指定します。

JETSET 画面の「ローダブルモジュールパス」の指定に相当します。

【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
```

本オプションは『ローダブルモジュールのデバッグ（手法１）(81 頁)』の手法でデバッグ時のみ有効です。

-RootDir オプション

書式　　-RootDir <ルートファイルシステム PATH>

機能

アプリケーションのパス名解決

解説

アプリケーションのデバッグ情報を解決するために、ルートファイルシステムツリーの PATH 情報を指定します。

Windows から見えるルートファイルシステムツリーのトップディレクトリへのパスを指定します。

【使用例】

ルートファイルシステムツリーのトップディレクトリが /opt/kmc/kzm-arm11/root で Z: ドライブに /opt をマウントした場合

```
-RootDir z:¥kmc¥kzm-arm11¥root
```

-MULTI オプション

書式　　-MULTI〈起動ウインドウ数〉

機能

PARTNER ウインドウの起動数指定

解説

PARTNER ウインドウを指定個数起動します。指定可能な最大ウインドウ数は 16 です。

【参考】

MULTI コマンド（163 頁）

-OPTIMIZE オプション

書式　　-OPTIMIZE

機能

最適化コンパイルされたプログラムの行番号補正

解説

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|SJIS|UTF8] オプション

書式 1　-EUC

書式 2　-SJIS

書式 3　-UTF8

機能

コードウインドウの漢字コード設定

解説

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設定します。デフォルトの文字コードは SJIS です。

書式 1 はソースファイルの文字コードが日本語 EUC であるものとして扱います (EUC カナは非サポートです)。

書式 2 はソースファイルの文字コードが Shift-JIS であるものとして扱います。

書式 3 はソースファイルの文字コードが UTF-8 であるものとして扱います。

【参考】

KANJI コマンド（169 頁）

-NPTL オプション

書式　　-NPTL

機能

マルチスレッドデバッグの NPTL 対応

解説

ターゲット上の pthread ライブラリが NPTL でコンパイルされているときに指定します。

本オプションは 2.6 系 Linux のデバッグをするときのみ有効です。

NPTL 方式と Linux スレッド方式を混在させたシステムを構築することも可能なため、将来本オプションは削除される可能性があります。

4.3 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。

Linux デバッグ専用コマンド

Linux デバッグ用に機能が拡張されたコマンド

PS コマンド

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。

Linux のスレッドの実装にはスレッドをプロセスの一種として扱う「Linux スレッド方式」とスレッドをプロセス内の実行コンテキストとして扱う「NPTL 方式」があります。NPTL 方式を使用している場合は、プロセス内のスレッドがインデントされて表示されます。

【使用例】

```
PT>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
PT>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
PT>
```

MAPS コマンド

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】

```
PT>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
PT>maps 1,c000 ↓
00008000-0000f000 r-xp /sbin/init
PT>
```

ATTACH コマンド

書式 1　**ATTACH <PID>**

書式 2　**ATTACH <file name>**

書式 3　**ATTACH ADD**

機能

アプリケーションのデバッガへのアタッチ操作

動作条件

ターゲット上で Linux が起動後

解説

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。
書式 1 はアタッチするアプリケーションを <PID> の整数値で指定します。
書式 2 は指定されたファイル名から <PID> を検索してアタッチします。
書式 3 はアタッチ中の thread とメモリ環境が同じで、まだアタッチされていない thread を自動検索してアタッチします。

このコマンドは、アプリケーションデバッグサポートファイルが Preload ライブラリとしてターゲットにインストールされているか、ターゲット環境で使用する glibc に埋め込まれている場合のみ使用可能です。
詳しくは『デバッグサポートライブラリのインストールと設定（61 頁）』参照してください。

【使用例】

```
PT>attach 100 ↓
PT>
PT>attach sample program ↓
PT>
```

THREAD コマンド

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（プロセス ID）,task_struct（タスク情報構造体のアドレス）,pc（プログラムカウンタ）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】

```
PT>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
PT>
```

LINUX コマンド

書式 1　**LINUX module**

書式 2　**LINUX module <ko モジュールファイルへのパス>**

書式 3　**LINUX module clr**

書式 4　**LINUX load_so**

書式 5　**LINUX [ATTACH|ATTACH_AUTO|ATTACH_AUTO_SO|ATTACH_MANUAL]**

書式 6　**LINUX set_attach_offset <ライブラリ><オフセットアドレス>**

機能

Linux をデバッグするためのターゲット上のファイル制御
・デバッグ対象 2.6 系ローダブルモジュールの登録 / 登録情報確認 / 登録情報削除
・共有ライブラリ情報の読み込み
・アタッチ時の ELF フォーマットの扱い指定
・デバッグ対象共有ライブラリのオフセット情報登録

解説

書式 1 ～書式 3 はカーネルモジュールのデバッグ用です。カーネルが起動した後でこのコマンドを使用して登録されたモジュールは、ターゲット Linux 上で insmod されたときに自動的にデバッグを開始します。
書式 1 はデバッグするモジュールの PARTNER への登録状況を表示します。
書式 2 は指定されたパスのモジュールを PARTNER へ登録します。登録可能なモジュール数は 8 個です(Linux カーネルの KMC パッチ内で定義されているため変更は可能です)。
書式 3 はデバッグするモジュールの PARTNER からの登録削除(全削除)をします。

書式 4 はアプリケーションが使用している共有ライブラリ(.so)のデバッグ情報を読み込みます。『-RootDir オプション（144 頁)』が指定されている場合のみ有効です。
また、読み込まれる共有ライブラリの検索には MAPS コマンド（151 頁）で表示される情報が用いられます。検索できないライブラリやメモリ参照不可能なアドレスの情報は読み込まれません。

書式 5 はアタッチしたときの ELF ファイルの読み方をアタッチ前にコマンドラインで設定します。引数によって指定される設定値は表 4-7 の通りです。

表 4-7 ELF ファイルの扱い指定

LINUX コマンド引数	動作	備考
ATTACH	現在の読み込みモードを表示します。	MANUAL, AUTO, AUTO_SO
ATTACH_AUTO	アタッチ時に実行ファイルのデバッグ情報を読み込む（デフォルト）	『-RootDir オプション（144 頁)』が有効時のみ
ATTACH_AUTO_SO	アタッチ時に実行ファイルと共有ライブラリ (SO) のデバッグ情報を読み込む	『-RootDir オプション（144 頁)』が有効時のみ
ATTACH_MANUAL	アタッチ時にデバッグ情報は読み込まない	

書式 6 はデバッグサポートファイルをライブラリとして使用するときに、そのオフセットアドレス情報を PARTNER へ登録しておくために使用します。このコマンドはデバッグサポートファイルを PRELOAD ライブラリとして使用する場合と glibc に埋め込んで使用する場合にのみ有効です。

書式 1 ～書式 3 は『INSMOD コマンド（156 頁）』とは混在できません。（デバッガの制限ではありませんが、運用が面倒になるので混在できないように、パッチで排他にしてあります）。
書式 6 はデバッグサポートファイル専用です。一般的な使い道はありません。

【使用例】

書式 1

```
PT>linux module ↓
 00000000 : z:¥home¥foo¥bar¥kernel¥linux¥fs¥udf¥udf.ko
```

書式 2

```
PT>linux module z:¥home¥foo¥bar¥kernel¥linux¥fs¥udf¥udf.ko ↓
TGT>insmod /home/foo/bar/kernel/linux/fs/udf/udf.ko ↓
```

書式 3

```
PT>linux module clr ↓
```

書式 4

```
PT>ls program ↓
PT>br main.ex ↓
PT>g ↓
TGT>./program ↓
PT>linux load so ↓
```

書式 5

```
PT>linux attach ↓
  ATTACH AUTO LOAD DEBUG INFORMATION : AUTO
```

書式 6

```
PT>linux set_attach_offset libkmcsup.so.2.0.0 0x00000624 ↓
```

INSMOD コマンド

書式 1　INSMOD

書式 2　INSMOD [GET|CLR|CLRBP|CLRALL]

機能

ローダブルモジュールのアタッチ情報表示
ローダブルモジュールの手動アタッチ制御

動作条件

ターゲットがブレーク中であること

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 はローダブルモジュールの手動デバッグ用（『5.2 手動ローダブルモジュールデバッグ（182 頁）』参照）のための拡張機能です。
引数によって指定できる操作は表 4-8 の通りです。

表 4-8 手動ローダブルモジュールデバッグ操作

INSMOD コマンド引数	動作
GET	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ます。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。
CLR	GET 操作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。
CLRBP	GET 操作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。
CLRALL	完全なクリアです。GET 操作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。

書式 2 は通常のデバッグでは使用しません（手動デバッグの時のみ使用可能です）。
手動でのデバッグ方法は『5.2 手動ローダブルモジュールデバッグ（182 頁）』を参照してください。

【使用例】

書式 1

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

書式 2

```
PT>ls u:¥linux¥drivers¥block¥rd.o ↓
 0xC1800060-0xC18007DB .text
 0xC18007DC-0xC180089B .rodata.str1.4
 0xC1800970-0xC18009B3 .data
 0xC1800A68-0xC1800BAB .bss
PT>bp rd_init ↓
PT>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
PT>insmod get ↓
INSMOD AREA : C1800000-C1800FFF
PT>
```

PSID コマンド

書式 1　**PSID**

書式 2　**PSID ADD**

書式 3　**PSID NON_ADD**

書式 4　**PSID [GET|CLR|CLRBP|CLRALL]**

機能

アプリケーション / デバッグモードの情報表示
PARTNER デバッグモードの制御
アプリケーションの手動アタッチ制御

動作条件

ターゲットがブレーク中であること

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てのプロセス ID も表示します。

書式 2 と書式 3 は PARTNER のデバッグモードを制御します。
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。

【参考】

-OS オプション（138 頁）

書式 4 はアプリケーションの手動デバッグ用（『5.3 手動アプリケーションデバッグ（191 頁)』及び『5.4 手動マルチプロセス / マルチスレッドデバッグ（195 頁)』参照）のための拡張機能です。
引数によって指定できる操作は表 4-8 の通りです。

表 4-9 手動プロセスデバッグ操作

PSID コマンド引数	動作
GET	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 にアタッチします。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。 GET 操作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などです。
CLR	GET 操作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。
CLRBP	GET 操作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。
CLRALL	完全なクリアです。GET 操作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。

書式 4 は通常のデバッグでは使用しません（手動デバッグの時のみ使用可能です)。
手動でのデバッグ方法は『5.3 手動アプリケーションデバッグ（191 頁)』『5.4 手動マルチプロセス / マルチスレッドデバッグ（195 頁)』を参照してください。

【使用例】

書式 1

```
PT>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
（表示される内容は CPU 種別や MAP フィールドの設定によって異なります。）
PT>
```

書式 4

```
PT>ls u:¥appli¥sample ↓
PT>bp main ↓
PT>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)
PT>
PT>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
(表示される内容は CPU 種別や MAP フィールドの設定によって異なります。)
PT>
```

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 ↓
```

TOP コマンド

書式　　TOP

機能

実行中プロセスのメモリ利用状況表示

解説

実行中のプロセスリストとメモリ使用状況を表示します。

Linux (Unix) の top コマンドと似ていますがリアルタイムの表示更新は行いません。

【使用例】

```
PT>
PT>top ↓
 MemTotal:  101848 K,    MemUsed  8140 K,    MemFree:  93708 K

  PID   VIRT    RES     SHR     PROGRAM
  1     1144 K  328 K   256 K   busybox
  398   584 K   304 K   176 K   udevd
  530   1500 K  988 K   780 K   bash
  531   1136 K  328 K   256 K   busybox
  532   1140 K  320 K   260 K   busybox
  742   540 K   180 K   136 K   hellohello
PT>
```

L コマンド

書式　　L[A] ファイル名 [,/OFFS= オフセット]

機能

デバッグプログラムおよびデバッグ情報のロード

動作条件

ターゲットがブレーク中であること

解説

このコマンドは、PARTNER の基本コマンドです。ここでは Linux デバッグ用に有用な「,/OFFS オプション」について解説します。すべての機能を知るには PARTNER のヘルプを参照してください。
「,/OFFS オプション」は特に ARM CPU 用の Linux カーネルで有用です。

一般に ARM の Linux カーネルは、vmlinux のリンク時に割り当てられたアドレス（仮想アドレス：以下 VA）と、そして実際にメモリ上に配置される物理アドレス（以下 PA）は一致しません。したがって、ICE からターゲットに vmlinux を転送するには、カーネルの VA と PA の違いを意識して転送する必要があります。

「,/OFFS オプション」はロードコマンドにリンクされたアドレスに対してオフセットを付加してバイナリをターゲットに転送する機能です。

【使用例】

```
PT>l vmlinux ,/offs=0xc0000000 ↓
PT>
```

上記はデバッガコマンドラインで指定する方法ですが、ロードダイアログのオプションボックスで ,/offs 以下を記述してもかまいません。

,/offs に指定する値は常に正の値（unsigned long）になります。例えば ARM CPU の KZM-ARM11-01 Linux カーネルの場合は、VA が C0008000 番地から始まり、その PA は 80008000 番地になるので、VA に対して +C0000000 します（溢れた 32bit 以上の桁は無視されます）。

MULTI コマンド

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 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】

```
PT>multi 3 ↓
PT>
```

【参考】

-MULTI オプション（145 頁）

K コマンド

書式 1　K 0

書式 2　K <PID>

機能

スタックフレームの内容表示（逆追跡・バックトレース）

動作条件

引数オプションは、デバッグ中プロセスのデバッグしている PARTNER ウインドウでのみ有効

解説

PARTNER に従来よりある K コマンドは引数を取りませんが、Linux デバッグ用に 2 種類の引数付き書式が追加されました。これらの拡張された（引数付きの)K コマンドは、カーネルや他のプロセスでブレークしている時にも利用できます。

書式1はカレントプロセスのユーザー側スタックの関数コールヒストリをコマンドウインドウに表示します。
書式2は <PID> で指定の PID のユーザー側スタックの関数コールヒストリをコマンドウィンドウに表示します。

【使用例】

```
PT>ls hellohello ↓
PT>BP .main+12 ↓
PT>g ↓
（ブレークポイントで停止）
```

書式 1

```
PT>K 0 ↓
 HELLOHELLO.C:  47 : 000087BC main+38(argc=E50B1024,argv=*E50B0020)
PT>
```

書式 2

```
PT>ps ↓
  1(0x1)        /bin/busybox
  398(0x18e)    /sbin/udevd
  530(0x212)    /bin/bash
  531(0x213)    /bin/busybox
  532(0x214)    /bin/busybox
  742(0x2e6)    /root/hellohello
PT>K 742 ↓
 HELLOHELLO.C:  47 : 000087BC main+38(argc=E50B1024,argv=*E50B0020)
PT>
```

Q,EXIT コマンド

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

動作条件

ターゲットがブレーク中であること

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（163 頁）や -MULTI オプション（145 頁）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】

PT>q/a ↓

【参考】

MULTI コマンド（163 頁）,-MULTI オプション（145 頁）

G コマンド

書式　　G/A

機能

プログラムの実行

動作条件

ターゲットがブレーク中であること

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 コマンド

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

動作条件

ターゲットがブレーク中であること

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 コマンド

書式　　**SNAME**

機能

コードウインドウのソースパスの表示

動作条件

ターゲットがブレーク中であること

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】

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

KANJI コマンド

書式　　KANJI [EUC|SJIS|UTF8]

機能

コードウインドウに表示するソースファイルの漢字コードを設定

解説

PARTNER のコードウインドウで表示するソースファイルの漢字コードを指定します。
指定が無い場合は現在の漢字コードが表示されます。
Linux デバッグ用に UTF8 が新たにサポートされました。

【使用例】

```
PT>kanji ↓
KANJI MODE --> SJIS
C:¥Program Files¥KMC¥WJETARM¥Bin¥iconv.dll version 1.10
PT>
PT>kanji euc ↓
PT>kanji ↓
KANJI MODE --> EUC
C:¥Program Files¥KMC¥WJETARM¥Bin¥iconv.dll version 1.10
PT>
```

【参考】

-[EUC|SJIS|UTF8] オプション（147 頁）

第 5 章　特殊なデバッグ手法

この章では、様々な状況に応じたデバッグ方法について説明します。

5.1 アプリケーションモードデバッグ

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。
アプリケーションモードは CPU を停止させずにプロセス毎にブレークしたい場合に使用する、カーネルモードと比べると特殊な用途のモードです。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。

5.1.1 デバッグに必要な設定条件

表 5-1 アプリケーションモードデバッグの条件

* 設定条件 / デバッグ対象	カーネルメニュー						-OS オプション			
	Debug infomation type	Enable patch for PARTNER debug	Loadable module debug by PARTNER-Jet (hook in kernel side)	Loadable module debug by PARTNER-Jet (hook in module side)	use PARTNER GigaTrace (PARTNER-Jet M40, ...)	enable PARNTER-Jet Event Tracker	カーネルモード	カーネル ADD モード	アプリケーションモード	アプリケーション ADD モード
アプリケーション	◎						—†		○	○

* ◎：必須、○：推奨、△：非推奨、×：不可、空欄：どちらでもよい
† 本節はアプリケーションモードを指定した場合の説明です。カーネルモードのデバッグ方法は『3.4 アプリケーションのデバッグ（91 頁）』『3.5 マルチスレッドアプリケーションデバッグ（98 頁）』『3.6 マルチプロセスアプリケーションデバッグ（108 頁）』を参照してください。

『2.2 Linux カーネルソースの修正と設定（50 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、『5.3 手動アプリケーションデバッグ（191 頁）』『5.4 手動マルチプロセス / マルチスレッドデバッグ（195 頁）』を参照してください。

5.1.2 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の手順

アプリケーションモードでデバッグするためには、PARTNER の起動設定(『PARTNER の設定と起動（70 頁)』参照) の OS デバッグモード指定(『 -OS オプション（138 頁)』参照) で「アプリケーション (NON_ADD) モード」または「アプリケーション ADD モード」を選択します。
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、『3.4 アプリケーションのデバッグ（91 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) デバッグ開始の手順（174 頁）
(2) アプリケーションをデバッグする（174 頁）

(1) デバッグ開始の手順

アプリケーションをコンパイルして PARTNER でを開始して main 関数でハードウェアブレークしてアタッチするまでの手順は『アプリケーションのデバッグ（91 頁）』と全く同じです。

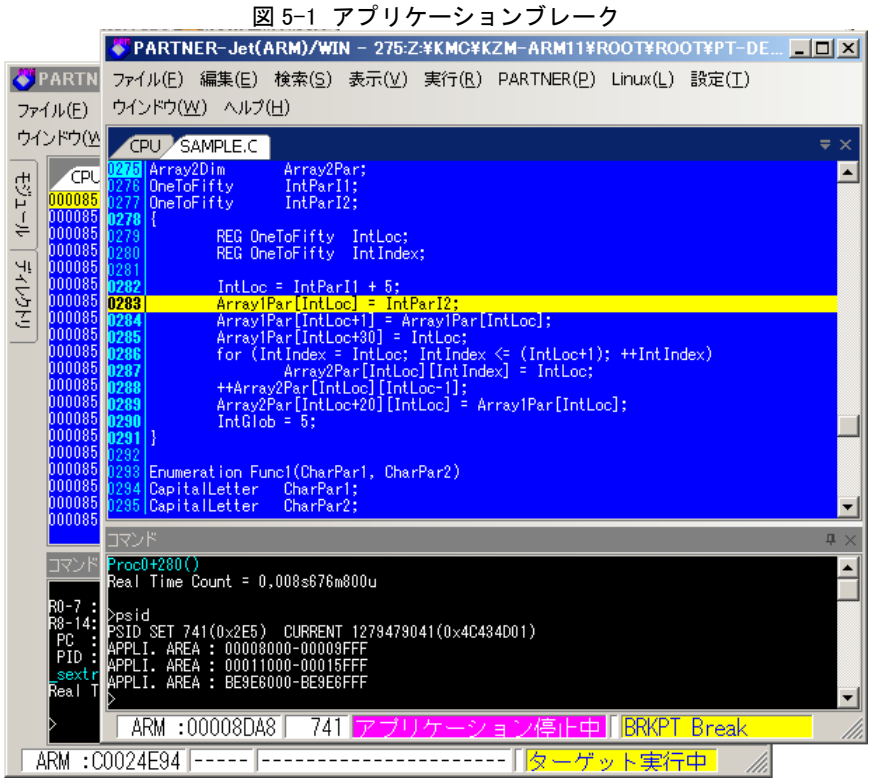
ハードウェアブレークポイントで停止中は CPU が停止します。アタッチ直後はアプリケーション用 PARTNER ウィンドウのステータス表示が『アプリ停止中 (カーネル)』となります。この時もまだ CPU は停止しており、アプリケーションにはソフトウェアブレークポイントが設定できません。1 ステップ実行しようとするとカーネル用ウインドウは『ターゲット実行中』に、アプリケーション用ウインドウは『アプリケーション停止中』へと変化しアプリケーションにソフトウェアブレークポイントの設定ができるようになります。main 関数でアプリケーションのみをブレークしてカーネルは停止させたくない場合は、アプリケーションの中にデバッグ用のスタブ関数を入れておく必要があります。詳しくは『アプリケーション直接リンク方式（62 頁）』を参照してください。

(2) アプリケーションをデバッグする

ATTACH コマンド（152 頁）でプロセスにアタッチしたら後はプロセス内をデバッグできます。カーネルモードと異なるのは、アプリケーションプロセス内でブレークしている間もカーネルが動作することです。

main 関数でハードウェアブレークしているときは CPU が停止しています。

図 5-1 アプリケーションブレーク



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『アプリケーションモードでのステータスバー表示（139 頁）』ように変化します。

5.1.3 アプリケーションモードでのマルチコンテキストデバッグの手順

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『5.1.2 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の手順（173 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（158 頁）または、-OS オプション（138 頁）で指定します。

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) デバッグ開始の手順（176 頁）
(2) PARTNER のブレーク（176 頁）

(1) デバッグ開始の手順

アプリケーションをコンパイルして PARTNER でを開始して main 関数でハードウェアブレークしてアタッチするまでの手順は『アプリケーションのデバッグ（91 頁）』と全く同じです。

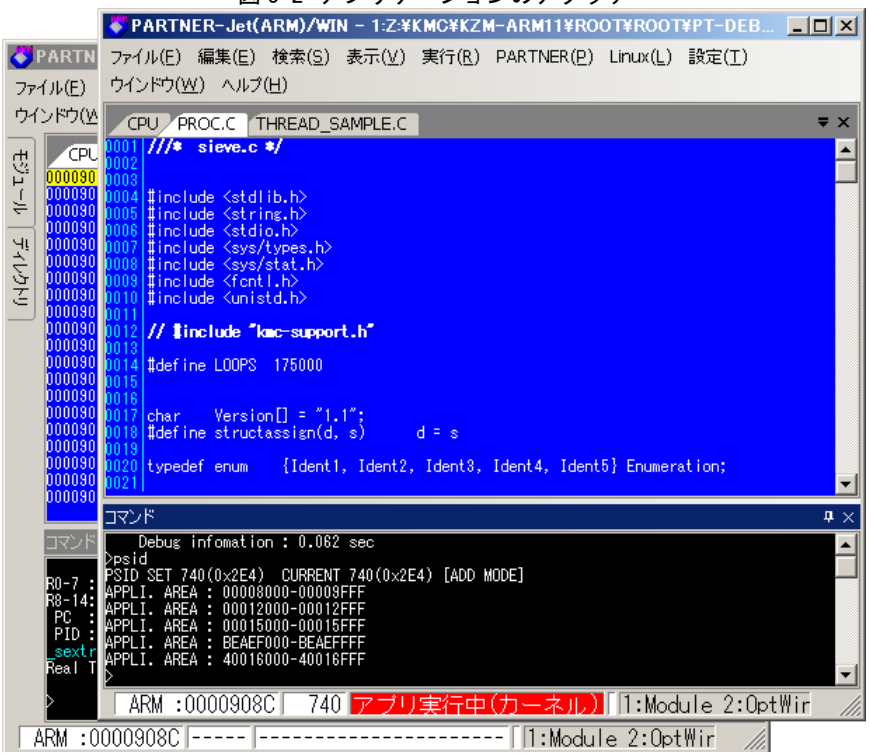
ハードウェアブレークポイントで停止中は CPU が停止します。ATTACH 直後はアプリケーション用 PARTNER ウィンドウのステータス表示が『アプリ停止中（カーネル）』となります。この時もまだ CPU は停止しており、アプリケーションにはソフトウェアブレークポイントが設定できません。1 ステップ実行しようとするとカーネル用ウインドウは『ターゲット実行中』に、アプリケーション用ウインドウは『アプリケーション停止中』へと変化しアプリケーションにソフトウェアブレークポイントの設定ができるようになります。
main 関数でアプリケーションのみをブレークしてカーネルは停止させたくない場合は、アプリケーションの中にデバッグ用のスタブ関数を入れておく必要があります。詳しくは『アプリケーション直接リンク方式（62 頁）』を参照してください。

(2) PARTNER のブレーク

● ADD モードの場合

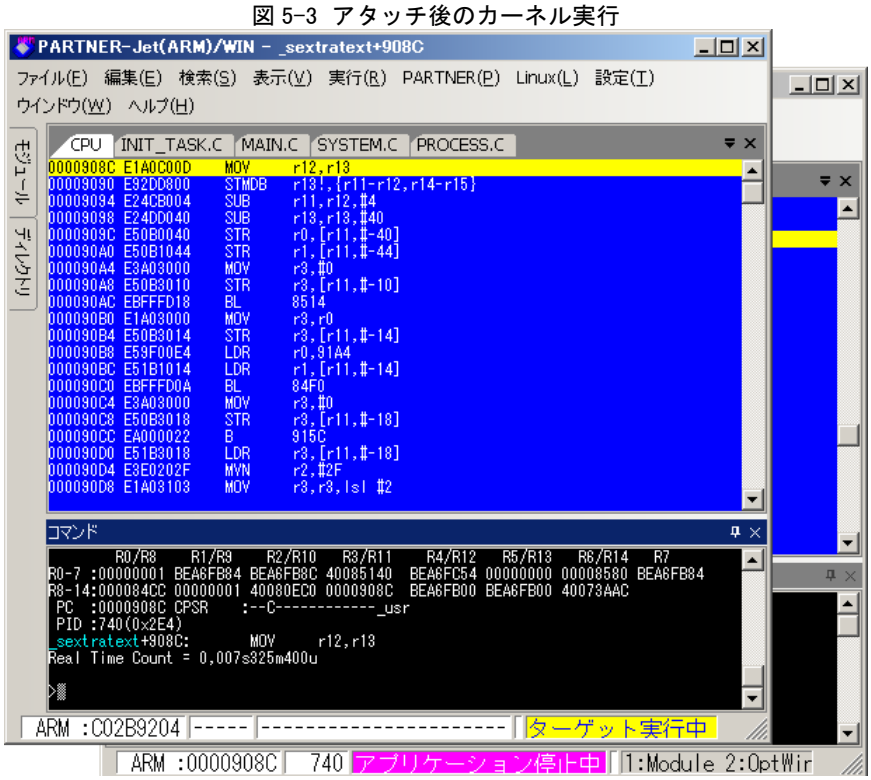
main でハードウェアブレークし、アタッチ直後の状態からブレークポイント設定可能な状態にします。

図 5-2 アプリケーションのアタッチ



トレース (F8) で 1 ステップしようとします。

図 5-3 アタッチ後のカーネル実行



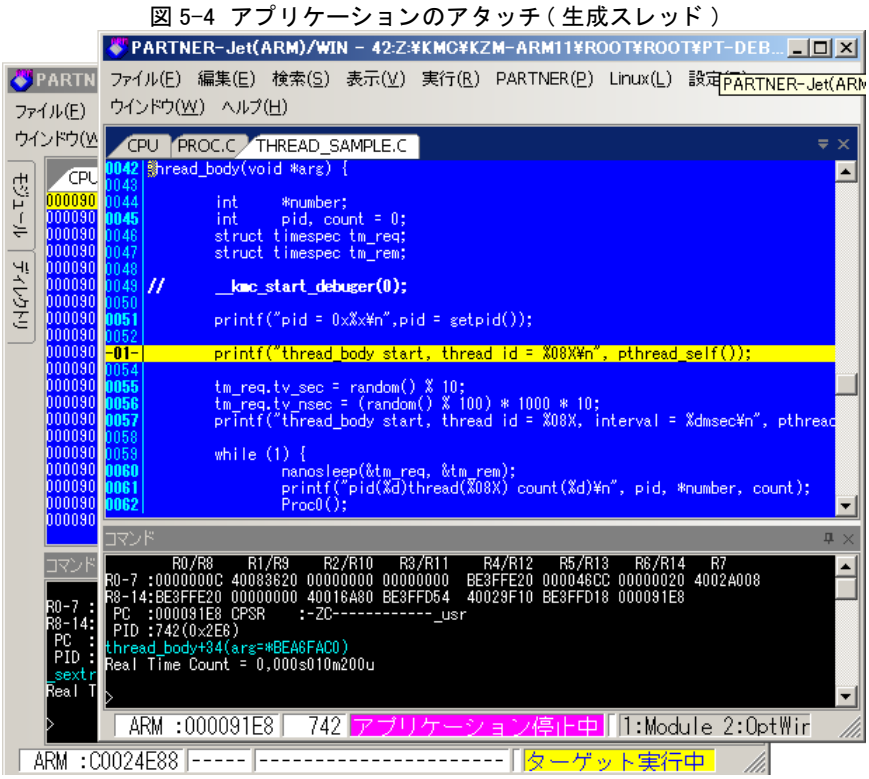
PSID コマンド（158 頁）でデバッガにアプリケーションがアタッチされているか確認します。

```
PT2>psid ↓
PSID SET 740(0x2E4)  CURRENT 740(0x2E4) [ADD MODE]
APPLI. AREA : 00008000-00009FFF
APPLI. AREA : 00012000-00012FFF
APPLI. AREA : 00015000-00015FFF
APPLI. AREA : BEA6F000-BEA6FFFF
APPLI. AREA : 40016000-40016FFF
```

（表示される内容は CPU 種別や MAP フィールドの設定によって異なります）

ここで thread_body() 関数の中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、生成されたスレッドが自動的にアタッチされブレークします。

図 5-4 アプリケーションのアタッチ（生成スレッド）



PSID コマンド（158 頁）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 742(0x2E6)  CURRENT 1129120076(0x434D014C) [ADD MODE 740]
APPLI. AREA : 00008000-00009FFF
APPLI. AREA : 00012000-00012FFF
APPLI. AREA : 00015000-00017FFF
APPLI. AREA : BE3FF000-BEA6FFFF
APPLI. AREA : 40016000-40016FFF
```

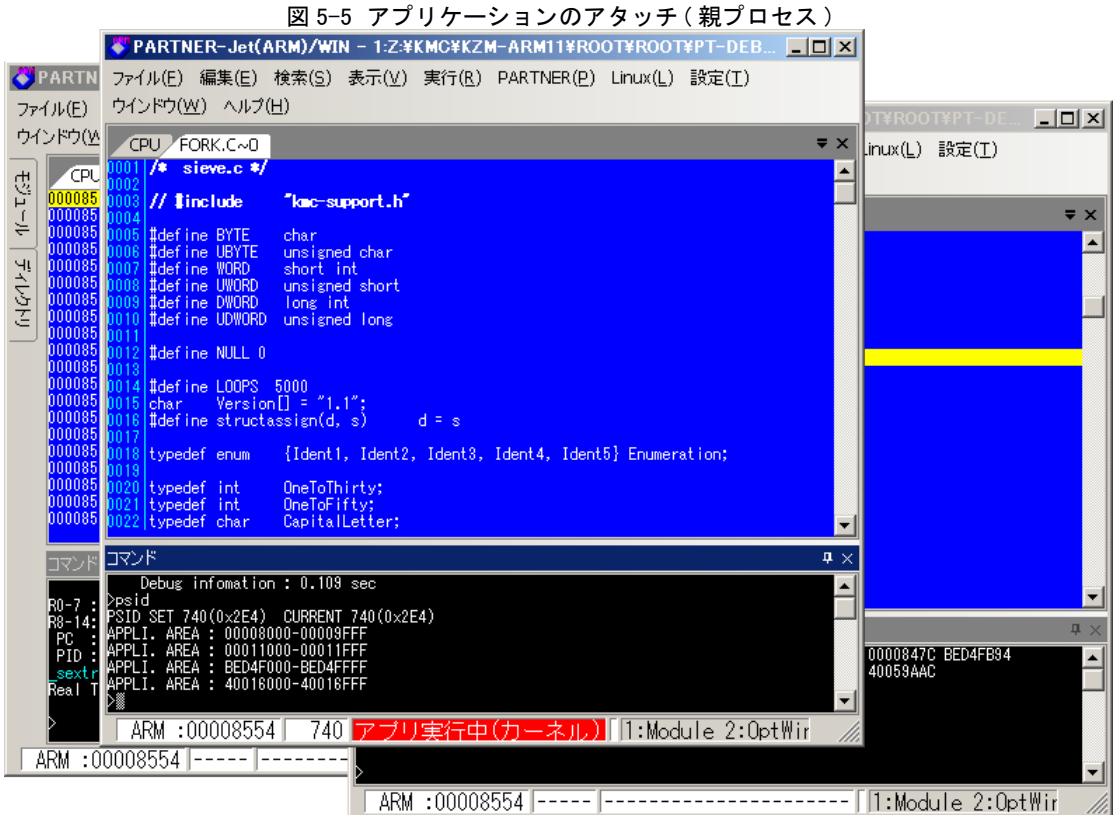
（表示される内容は CPU 種別や MAP フィールドの設定によって異なります）

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

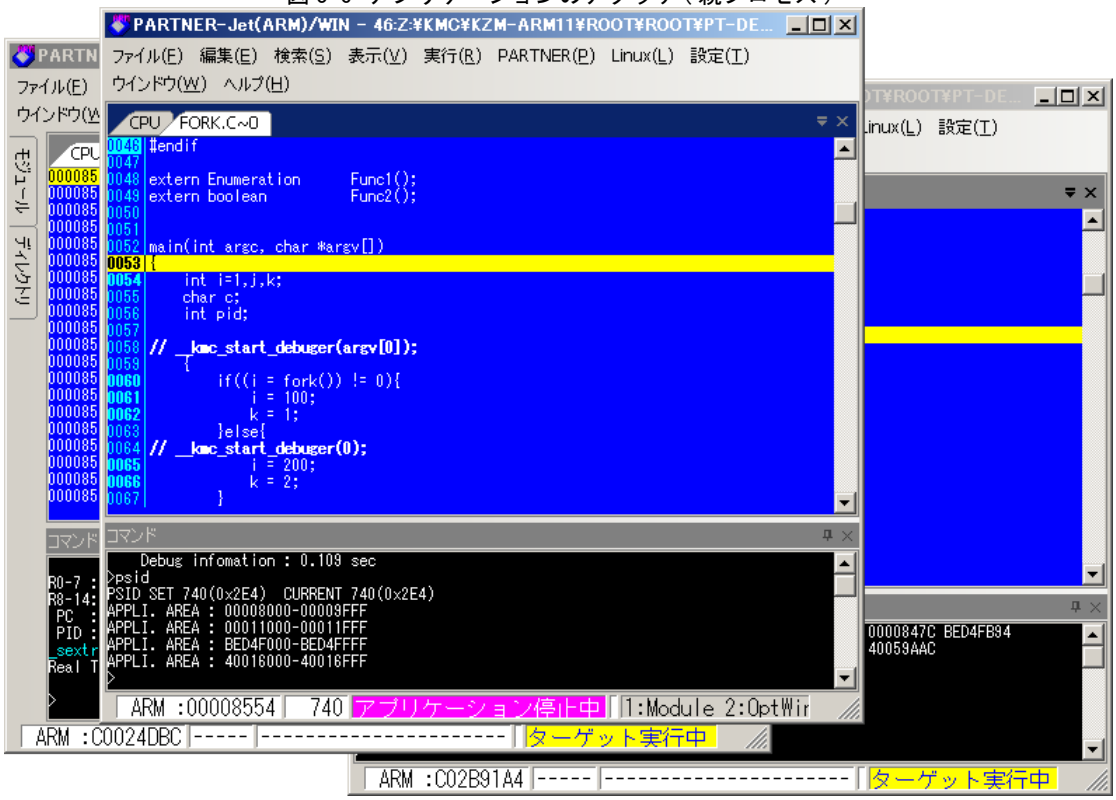
main でハードウェアブレークし、アタッチ直後の状態からブレークポイント設定可能な状態にします。

図 5-5 アプリケーションのアタッチ（親プロセス）



トレース (F8) で 1 ステップしようとします。

図 5-6 アプリケーションのアタッチ（親プロセス）

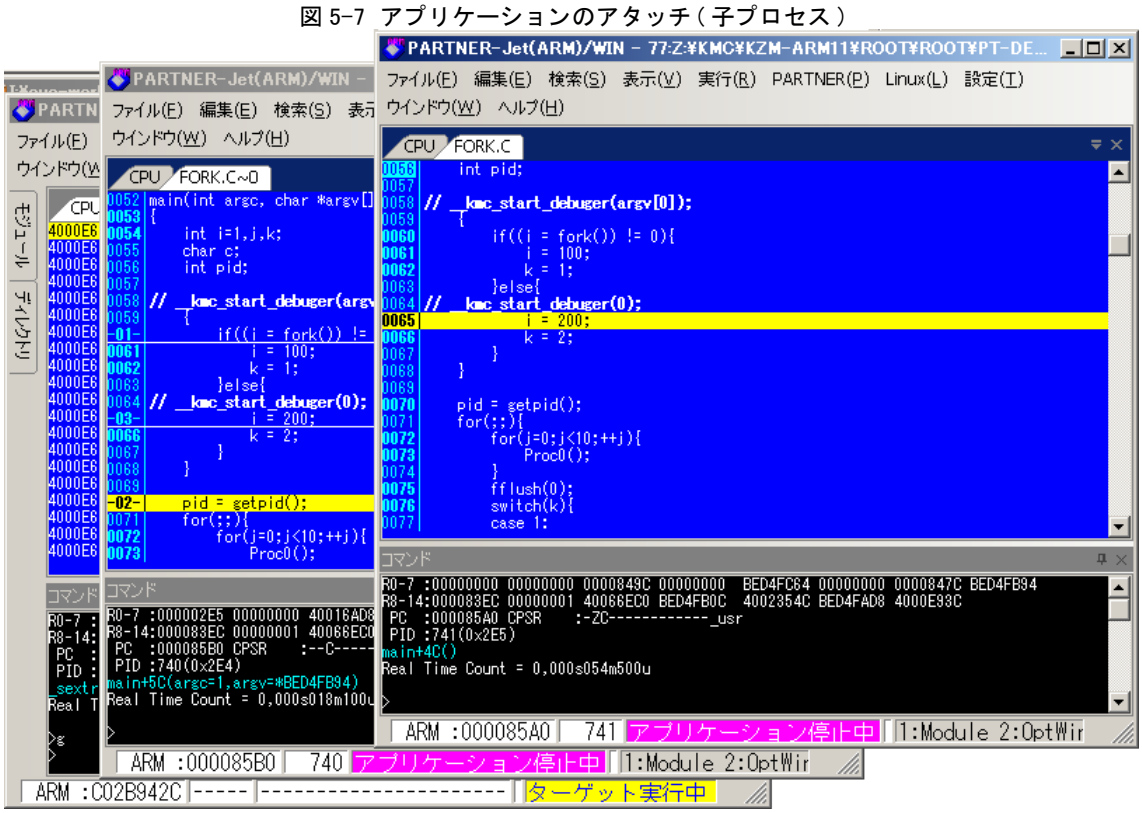


PSID コマンド（158 頁）を実行するとアプリケーション用 PARTNER ウインドウで生成スレッドがアタッチされていることが確認できます。

```
PT>psid ↓
PSID SET 740(0x2E4)  CURRENT 740(0x2E4)
APPLI. AREA : 00008000-00009FFF
APPLI. AREA : 00011000-00011FFF
APPLI. AREA : BED4F000-BED4FFFF
APPLI. AREA : 40016000-40016FFF
（表示される内容は CPU 種別や MAP フィールドの設定によって異なります）
```

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 ウインドウがブレークします。このとき、子プロセスの PID や配置情報は PARTNER によって自動的に収集され、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っています（図 5-7 で、中央の親プロセス用のウインドウで fork() 関数の子プロセス側にソフトウェアブレークポイントを設定しているのに停止せず、右の子プロセス用のウインドウでブレークしているところに注目してください）。

図 5-7 アプリケーションのアタッチ（子プロセス）



PSID コマンド（158 頁）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
PSID SET 741(0x2E5)  CURRENT 1129120076(0x434D014C)
```

```
APPLI. AREA : 00008000-00009FFF
APPLI. AREA : 00011000-00011FFF
APPLI. AREA : BED4F000-BED4FFFF
APPLI. AREA : 4000E000-4000EFFF
```

（表示される内容は CPU 種別や MAP フィールドの設定によって異なります）

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。『2.2.6 カーネルコンフィグレーション（54 頁）』の設定と『2.4.2 PARTNER の設定と起動（70 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、サポートライブラリ (libkmcsup.so.2.0.0) が Preload されているか確認してください。

5.2 手動ローダブル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ローダブルモジュールのデバッグ手順を説明します。

5.2.1 デバッグに必要な設定条件

表 5-2 手動ローダブルモジュールデバッグの条件

* 設定条件 / デバッグ対象	カーネルメニュー						-OS オプション			
	Debug infomation type	Enable patch for PARTNER debug	Loadable module debug by PARTNER-Jet (hook in kernel side)	Loadable module debug by PARTNER-Jet (hook in module side)	use PARTNER GigaTrace (PARTNER-Jet M40, ...)	enable PARNTER-Jet Event Tracker	カーネルモード	カーネル ADD モード	アプリケーションモード	アプリケーション ADD モード
ローダブルモジュール	◎						○	○	×	

* ◎：必須、○：推奨、△：非推奨、×：不可、空欄：どちらでもよい

手動ローダブルモジュールデバッグは、アプリケーションモードでは出来ません。

5.2.2 デバッグの手順

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) ローダブルモジュールの作成（184 頁）

(2) Linux カーネルの起動（184 頁）

(3) ターゲット上でローダブルモジュールのマップファイルの作成（184 頁）

(4) ブレークポイントの設定（187 頁）

(5) ローダブルモジュールのインストール（187 頁）

(6) PARTNER のブレーク（187 頁）

(7) モジュールのアタッチ（188 頁）

(8) モジュールのデタッチ（189 頁）

(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。

(2) Linux カーネルの起動

『デバッグ環境の起動（68 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。

(3) ターゲット上でローダブル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 ↓
```

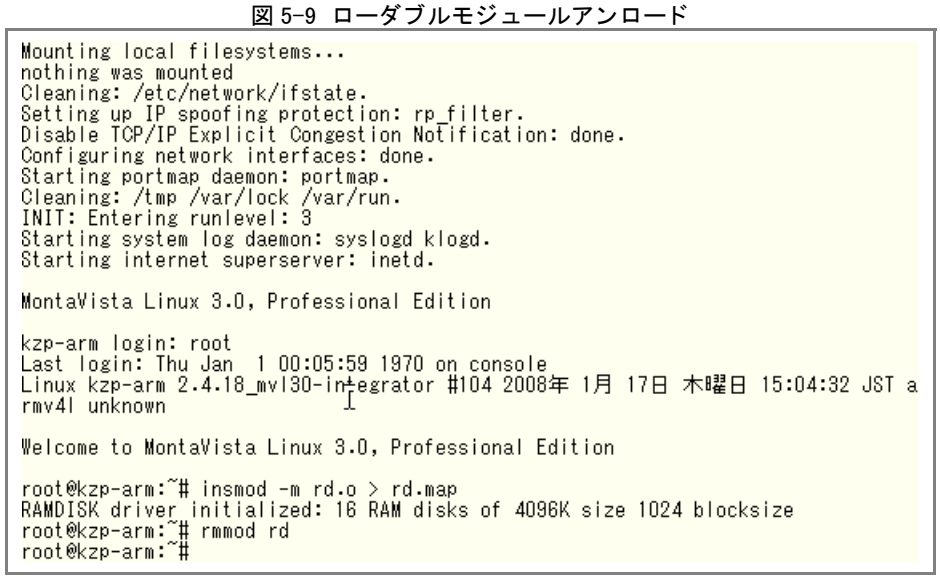
図 5-8 ローダブルモジュールの MAP ファイル作成

```
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 00:05:59 1970 on console
Linux kzp-arm 2.4.18_mvl30-integrator #104 2008年 1月 17日 木曜日 15:04:32 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 5-9 ローダブルモジュールアンロード



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通りです。

- 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

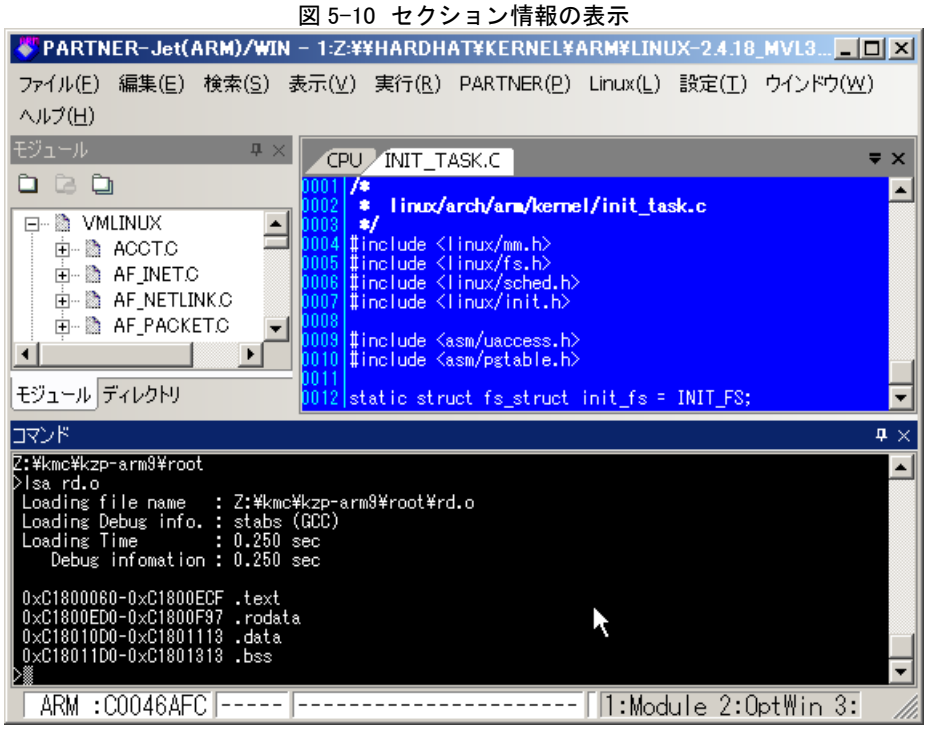
(4) ローダブルモジュールのデバッグ情報の読み込み

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 ↓
```

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 5-10 セクション情報の表示



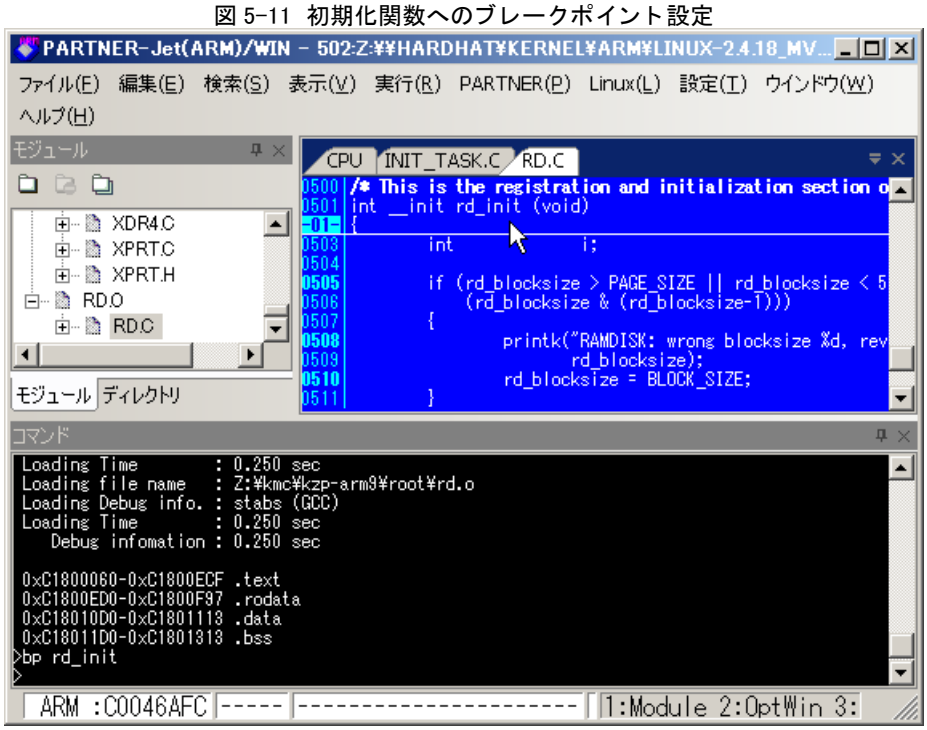
デバッグ情報ロード時に、エラーメッセージ『指定ファイルがありません』が表示された場合は、ロードしたローダブルモジュールと同じディレクトリに MAP ファイルが存在しないしません。再度、『(2) Linux カーネルの起動 (184 頁)』の MAP ファイル作成を行ってください。

(5) 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 5-11 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、既に設定可能な数の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数は CPU により異なります）。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。

(6) 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。

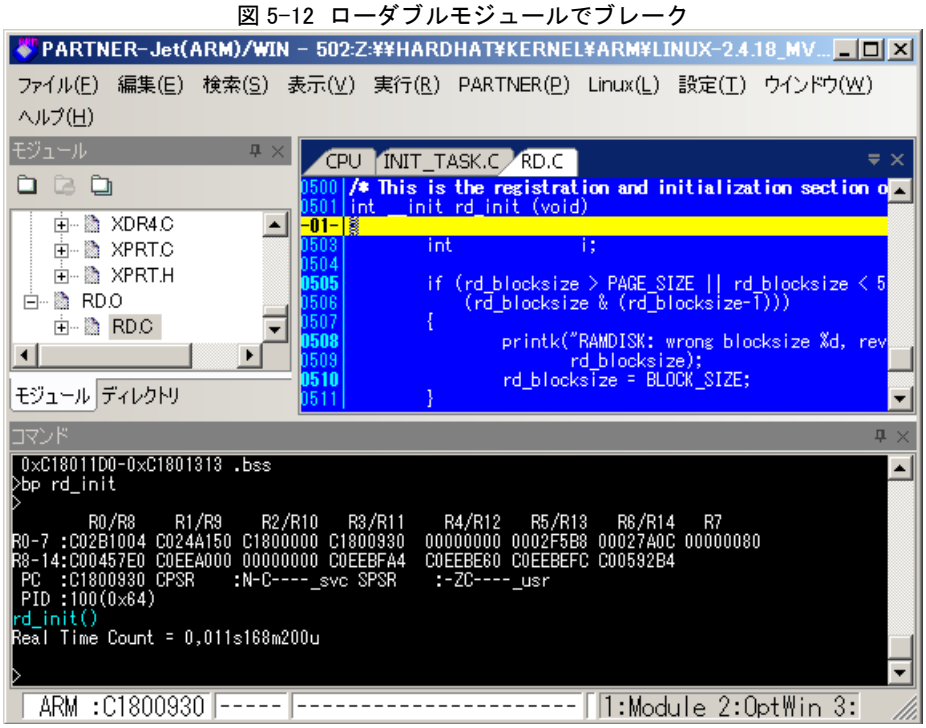
```
TGT>insmod rd.o ↓
```

(7) 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 5-12 ローダブルモジュールでブレーク



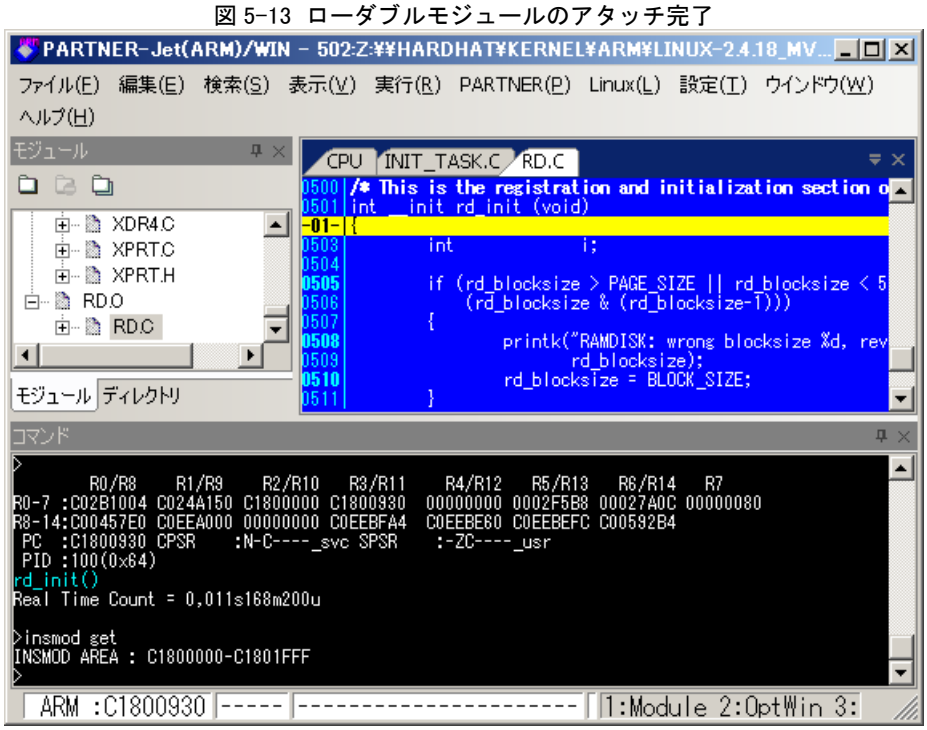
(8) モジュールのアタッチ

INSMOD コマンド （156 頁）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 5-13 ローダブルモジュールのアタッチ完了



(9) 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（156 頁）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 5-14 ローダブルモジュールのアンロード

```
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 00:00:32 1970 on console
Linux kzp-arm 2.4.18_mvl30-integrator #104 2008年 1月 17日 木曜日 15:04:32 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:~# 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                      4884   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 からデタッチしないと、カーネルパニックが発生します。

5.3 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アプリケーションのデバッグ手順を説明します。

5.3.1 デバッグに必要な設定条件

表 5-3 手動アプリケーションデバッグの条件

* 設定条件 / デバッグ対象	カーネルメニュー						-OS オプション			
	Debug infomation type	Enable patch for PARTNER debug	Loadable module debug by PARTNER-Jet (hook in kernel side)	Loadable module debug by PARTNER-Jet (hook in module side)	use PARTNER GigaTrace (PARTNER-Jet M40, ...)	enable PARNTER-Jet Event Tracker	カーネルモード	カーネル ADD モード	アプリケーションモード	アプリケーション ADD モード
アプリケーション	◎						○	○	×	

* ◎：必須、○：推奨、△：非推奨、×：不可、空欄：どちらでもよい

手動アプリケーションデバッグは、アプリケーションモードでは出来ません。

5.3.2 デバッグの手順

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。

(1) アプリケーションの作成（192 頁）

(2) デバッグの準備（192 頁）

(3) アプリケーション用 PARTNER ウインドウを開く（192 頁）

(4) アプリケーションのデバッグ情報の読み込み（192 頁）

(5) ブレークポイントの設定（192 頁）

(6) アプリケーションの実行（193 頁）

(7) PARTNER のブレーク（193 頁）

(8) アプリケーションのアタッチ（193 頁）

(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。

(2) デバッグの準備

『デバッグ環境の起動（68 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。

(3) アプリケーション用 PARTNER ウインドウを開く

MULTI コマンド（163 頁）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 ↓
```

(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。
GUI メニューからロードする時には必ず [Symbol only] のチェックを付けてください。

```
PT>ls sample ↓
```

(5) ブレークポイントの設定

アプリケーションの main() 関数にブレークポイントを設定します。

```
PT>br main.ex ↓
```

図 5-15 ブレークポイントの設定

ソフトウェアブレーク（bp）よりも実行型ハードウェアブレーク（br）の方が確実です。bp の場合、PARTNER はデバッグ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。bp を用いる場合、ブレークポイントの設定は 1 点だけが無難です。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、既に設定可能な数の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数はCPUにより異なります)。ブレークポイントを削除してから再度設定してください。

---

(6) アプリケーションの実行

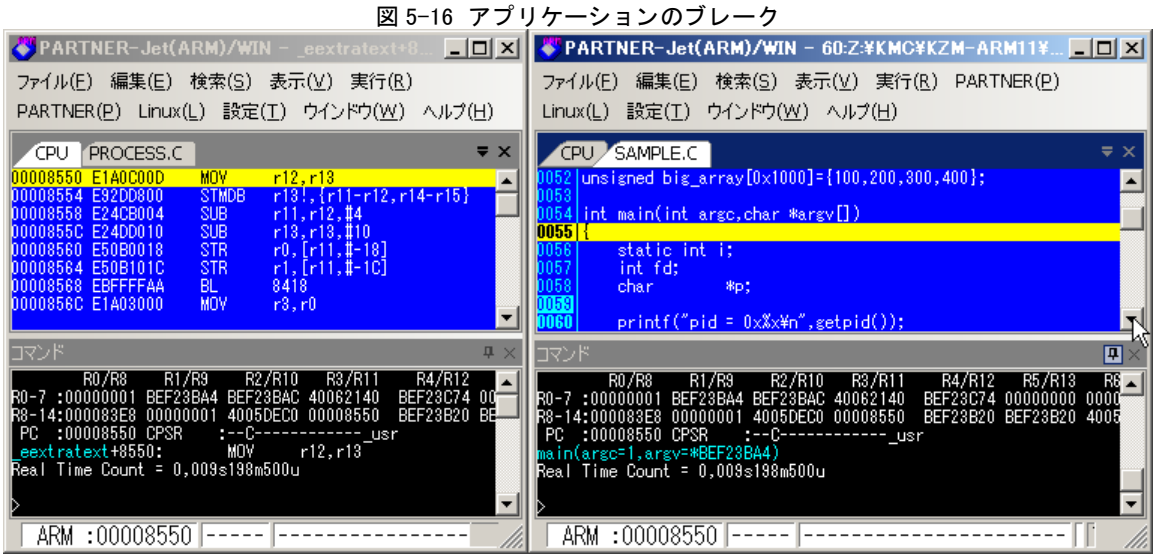
カーネルを実行し、ターゲットシステムでデバッグ対象のアプリケーションを実行します。

```
TGT>./sample ↓
```

(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実行されると、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 5-16 アプリケーションのブレーク



main シンボルのアドレスは複数のプロセスで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で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。

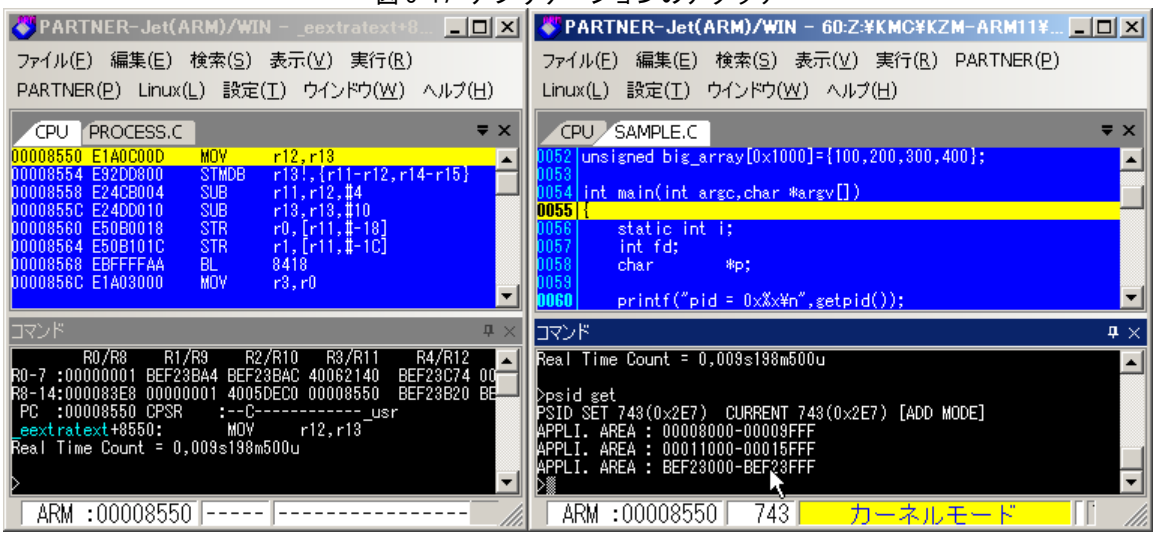
(8) アプリケーションのアタッチ

PSID コマンド （158 頁）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 5-17 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 ↓

5.4 手動マルチプロセス / マルチスレッドデバッグ

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アプリケーションのデバッグ手順を説明します。

5.4.1 デバッグに必要な設定条件

表 5-4 手動マルチコンテキストアプリケーションデバッグの条件

* 設定条件 / デバッグ対象	カーネルメニュー						-OS オプション			
	Debug infomation type	Enable patch for PARTNER debug	Loadable module debug by PARTNER-Jet (hook in kernel side)	Loadable module debug by PARTNER-Jet (hook in module side)	use PARTNER GigaTrace (PARTNER-Jet M40, ...)	enable PARNTER-Jet Event Tracker	カーネルモード	カーネル ADD モード	アプリケーションモード	アプリケーション ADD モード
アプリケーション	◎						○	○	×	

* ◎：必須、○：推奨、△：非推奨、×：不可、空欄：どちらでもよい

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は、アプリケーションモードでは出来ません。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箇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

5.4.2 デバッグの手順

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。

(1) アプリケーションの作成（197 頁）

(2) カーネルの実行（197 頁）

(3) アプリケーション用 PARTNER ウインドウの起動（197 頁）

(4) アプリケーションのデバッグ情報の読み込み（197 頁）

(5) 親プロセスへのブレークポイントの設定（198 頁）

(6) アプリケーションの実行（198 頁）

(7) PARTNER のブレーク（198 頁）

(8) 親プロセスのアタッチ（199 頁）

(9) 子プロセスへのブレークポイントの設定（199 頁）

(10) アプリケーションの再実行（200 頁）

(11) PARTNER のブレーク（200 頁）

(12) 子プロセスのアタッチ（200 頁）

(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。

(2) カーネルの実行

『デバッグ環境の起動（68 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。

(3) アプリケーション用 PARTNER ウインドウの起動

MULTI コマンド（163 頁）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。

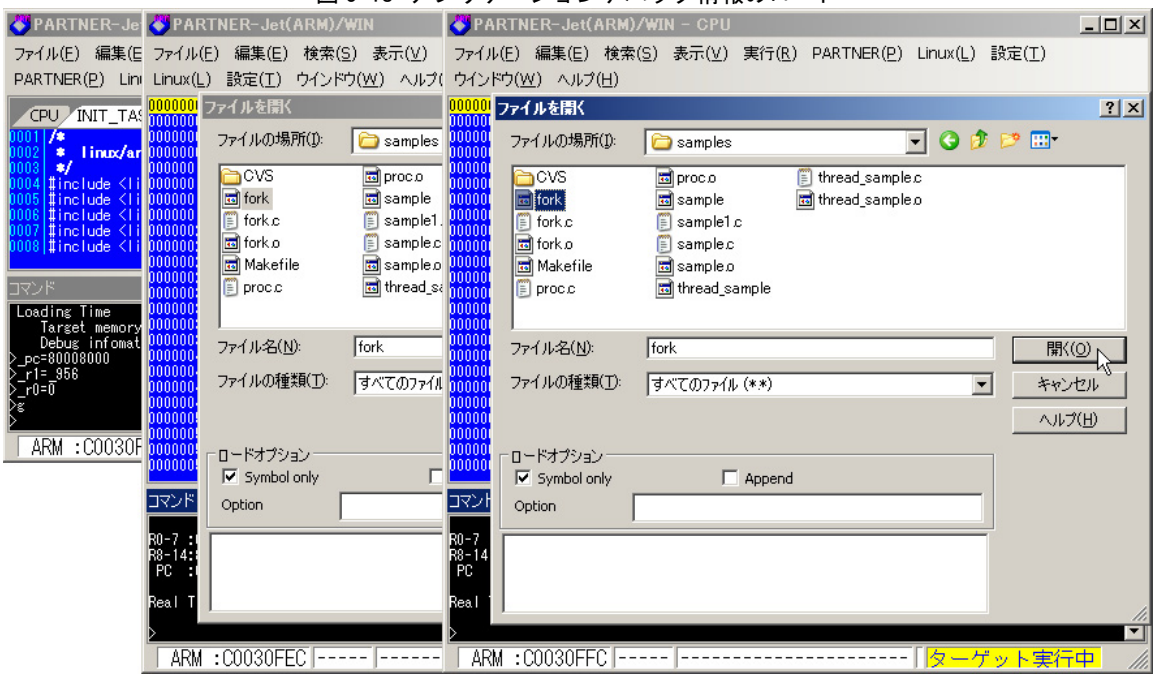
```
PT>MULTI 3 ↓
```

(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] のチェックを付けてください。

```
PT>ls fork ↓
```

図 5-18 アプリケーションデバッグ情報のロード



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、『3.7 共有ライブラリのデバッグ（114 頁）』を参照してください。

(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() 関数にブレークポイントを設定します。

```
PT2>br main.ex ↓
```

ソフトウェアブレーク（bp）よりも実行型ハードウェアブレーク（br）の方が確実です。bp の場合、PARTNER はデバッグ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。bp を用いる場合、ブレークポイントの設定は 1 点だけが無難です。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、既に設定可能な数の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数はCPUにより異なります)。ブレークポイントを削除してから再度設定してください。

(6) アプリケーションの実行

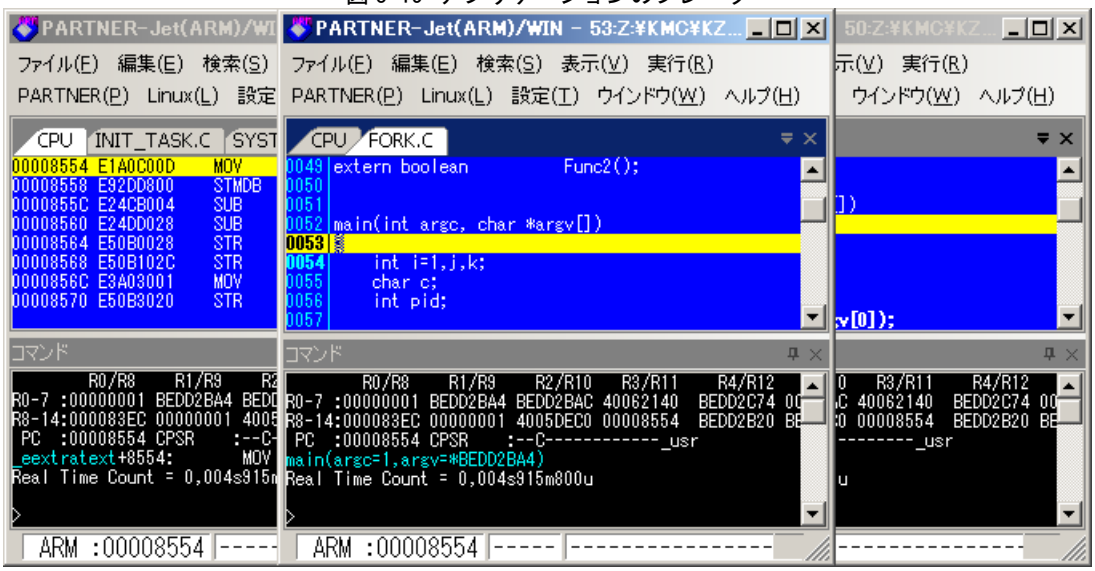
カーネルを再実行し、ターゲットシステムでデバッグ対象のアプリケーションを実行します。

```
TGT>./fork ↓
```

(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 5-19 アプリケーションのブレーク



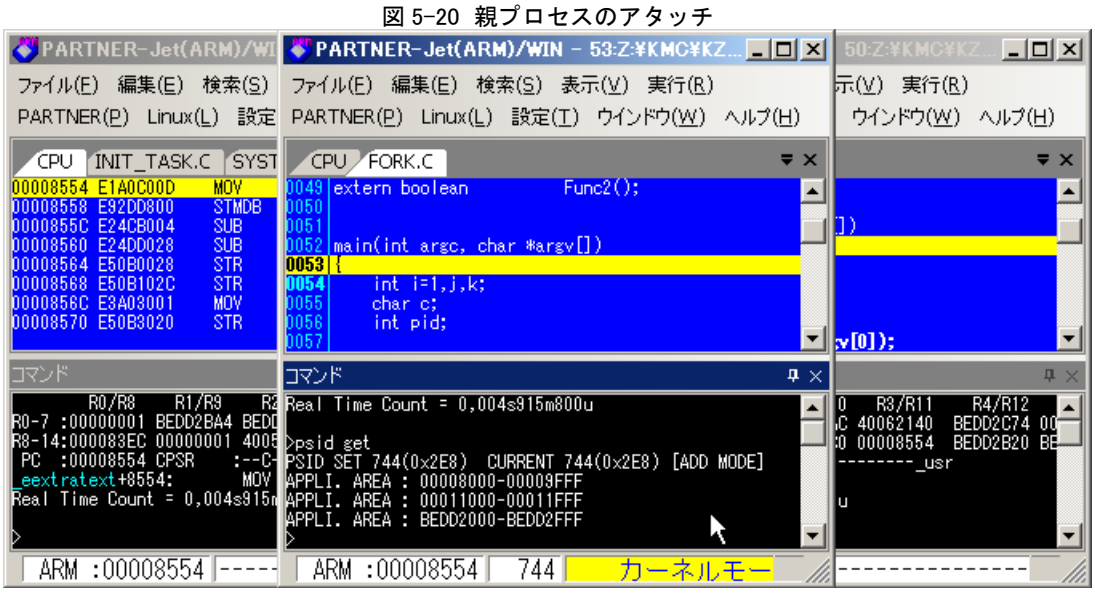
main シンボルのアドレスは複数のプロセスで使用されている可能性があるため、プログラム実行時に別のプログラムでブレークが発生する可能性があります。この場合は、目的のアプリケーションでブレークするまで、プログラムをそのまま続行（[F5] キー）させてください。

(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（158 頁）を実行し、親プロセスをアタッチします。

```
PT2>PSID GET ↓
```

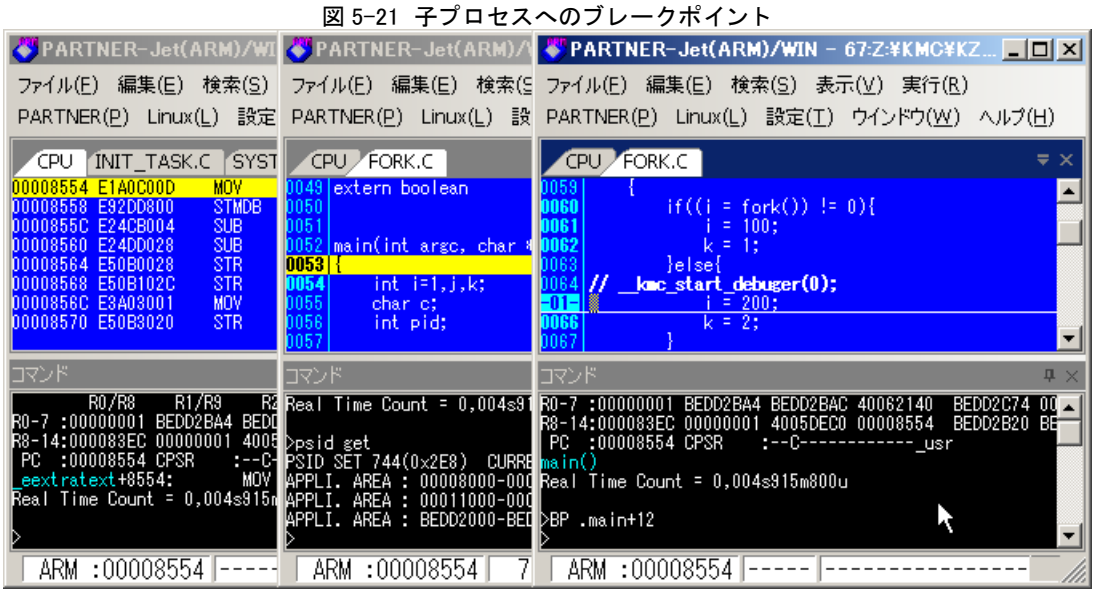
図 5-20 親プロセスのアタッチ



(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。

図 5-21 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、既に設定可能な数の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数は CPU により異なります）。設定数には PSID GET で

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。ブレークポイントを削除してから再度設定してください。

---

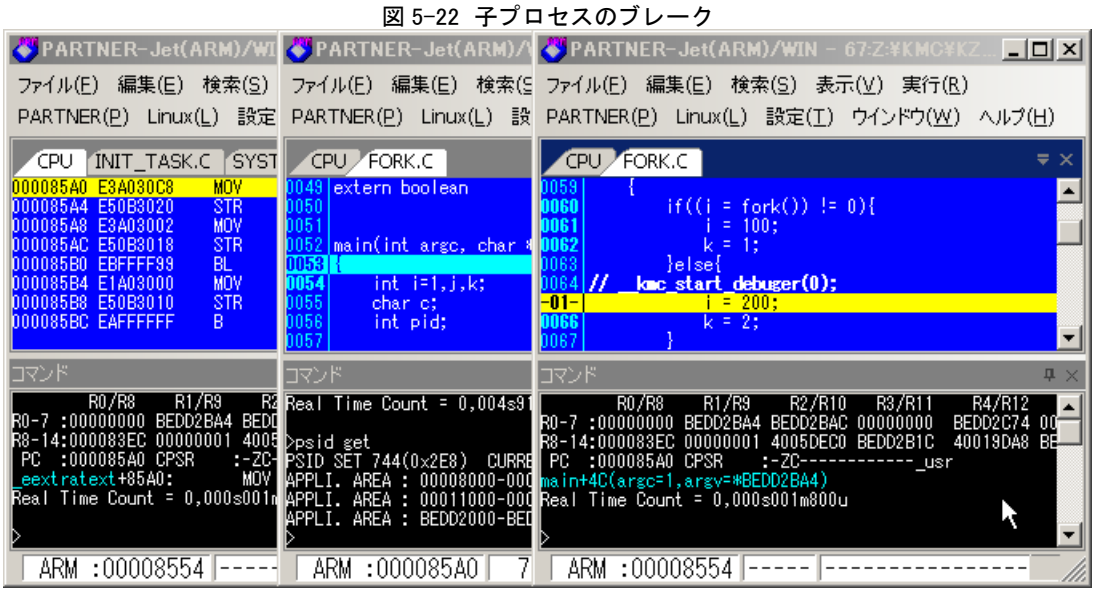
(10) アプリケーションの再実行

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。

(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 5-22 子プロセスのブレーク



PARTNER は、デバッグ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。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セスで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。

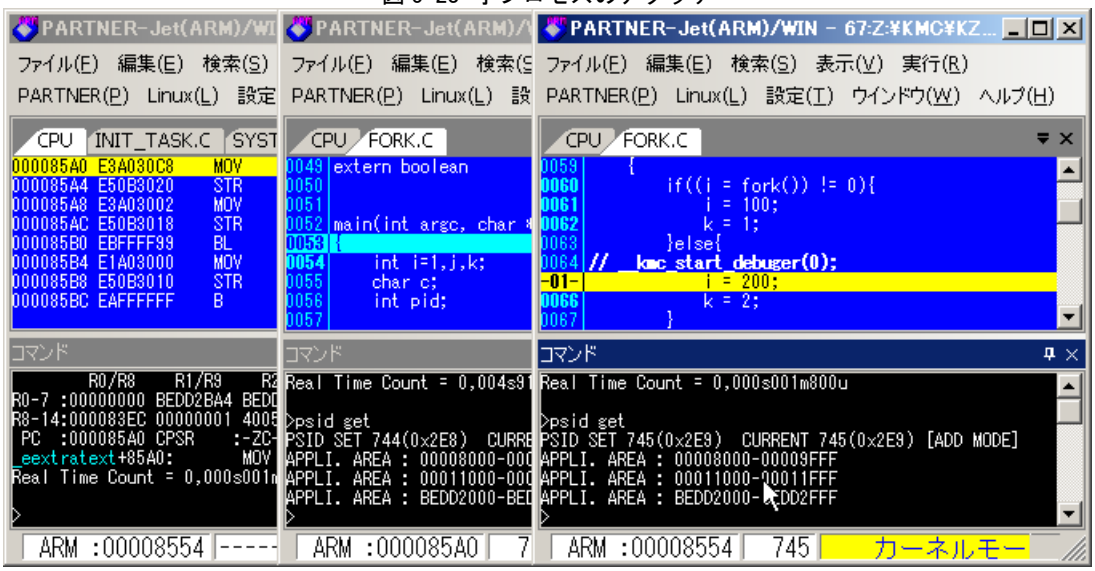
---

(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 (158 頁) を実行し、子プロセスをアタッチします。

```
PT>PSID GET ↓
```

図 5-23 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 ウインドウで、それぞれ PSID GET コマンドを発行した状態) 後は、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。）

第 6 章　イベントトラッカー

この章では、PARTNER のイベントトラッカー機能の利用方法について説明します。

6.1 イベントトラッカーに必要なもの

イベントトレース機能はアプリケーションのプロセスやスレッド動作など、ターゲット OS 上で発生するイベントの履歴を保存し、PARTNER のウインドウでグラフィカルに可視化する機能です。本書で扱っている Linux デバッグ機能とは別のものとして開発されました。
イベントトレース機能を使用するには対応したコンポーネントが揃っている必要があります。
表 6-1 に必要なコンポーネント一覧を掲載します。

表 6-1 イベントトラッカーに必要なコンポーネント

種別	対象	内容
PARTNER	WINPC	イベントトラッカー機能に対応したバージョンの PARTNER ソフトウェアが必要です。
イベント解析フィルタ	WINPC	KMC より提供されている MyFilter.dll もしくはユーザー様でカスタマイズしてビルドしたイベント解析フィルタ DLL が必要です。
カーネル修正	TGT	イベント収集機能をカーネル内に組み込みます。 『イベントトラッカー用の Linux カーネルの修正（205 頁）』参照。
アプリケーション修正	TGT	ユーザーモードのプログラム（アプリケーション）を便利にデバッグするための機能をアプリケーションに組み込みます。 KMC よりアプリケーションデバッグサポートファイルとして提供されます。アプリケーションへの組み込み方法は『アプリケーションデバッグサポートファイル（58 頁）』参照。
PARTNER 設定	PT	Linux デバッグ用の設定がされていれば、イベントトラッカー機能に必要な記述はありません。
PARTNER 起動オプション	PT	Linux デバッグ用の設定がされていれば、イベントトラッカー機能に必要な記述はありません。
PARTNER コマンド	PT	状況に応じて PARTNER のコマンドウインドウに入力します。コマンドが Linux 用に拡張されています。

6.2 イベントトラッカー用の Linux カーネルの修正

カーネル内の修正のほとんどは KMC よりファイルで提供されます。イベント収集の主な機能は以下の３ファイルに記述されています。これらのファイルは本来の Linux カーネルソースには存在せず、KMC より提供のパッチファイルに含まれるファイルです。

・KMC/kmc.c

・KMC/kmc.h

・KMC/Kconfig_KMC

その他、本来の Linux カーネルに含まれるファイルに必要な変更箇所として、プロセス（スレッド）が生成される以下の２カ所があります。KMC より提供されるサンプル実装を参考にして変更してください。

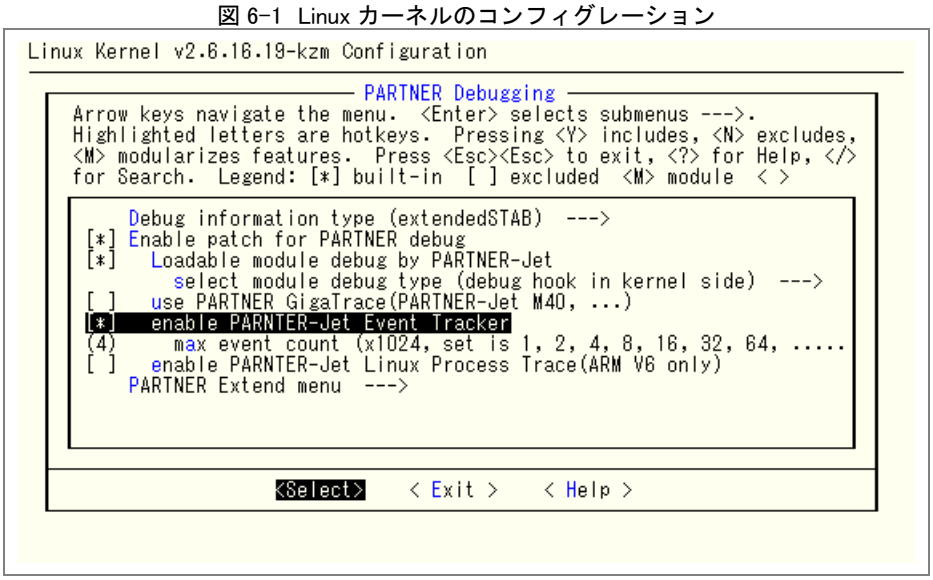
kernel/fork.c	#1206 copy_process() 関数の最後の方
fs/exec.c	#1215 do_execve() 関数の後半

修正を行うとカーネルビルドのコンフィグレーションメニューが追加されます。

[PARTNER Debugging]->[Enable PARTNER-Jet Linux Process trace]

[PARTNER Debugging]->[Enable PARTNER-Jet Linux Profile (only ARM V6)]

図 6-1 Linux カーネルのコンフィグレーション



イベントトラッカーを使用するには [Enable PARTNER-Jet Linux Process trace] を有効にしてください。イベントトラッカーを使用するには、ターゲット上のメインメモリにイベント収集用のバッファを確保する必要があります。バッファサイズは 2 の乗数のサイズで指定してください。

可能ならば [Enable PARTNER-Jet Linux Profile (only ARM V6)] も有効にしてください（適用可能なターゲット・アーキテクチャのみ）。

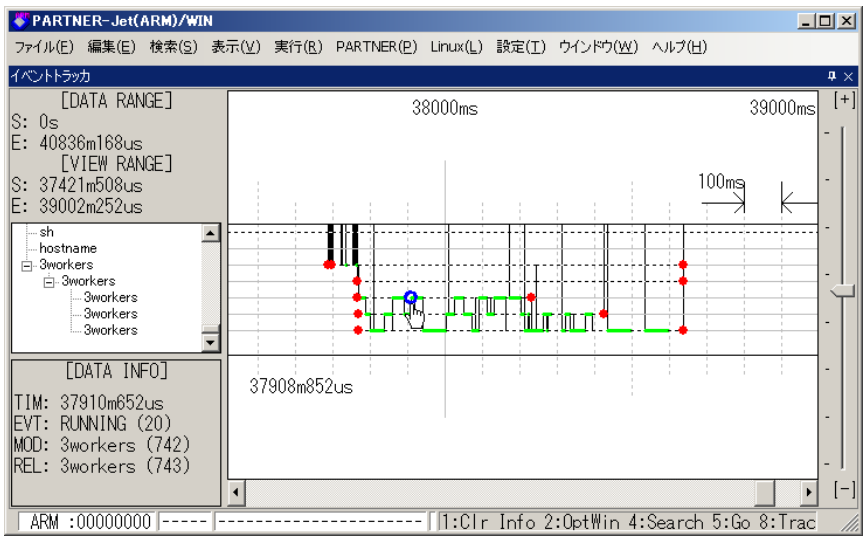
6.3 イベントトラッカーをデバッガで起動する

イベントトラッカーをデバッガで起動するには VIEWLOG コマンド（213 頁）を使用します。イベント解析フィルタ DLL を PARTNER に読み込んだのち、任意のタイミングで CPU を停止し、VIEWLOG コマンド（213 頁）の書式 2 を使用することでターゲット上のイベント履歴バッファの内容を解析して可視化します。なお、フィルタ DLL はカスタムビルドを作成することも可能です。ここでは MyFilter.dll という名前で記載しますので適宜読み替えてください。

```
PT>viewlog dll MyFilter.dll ↓
TGT>./ls ↓
TGT>./pwd ↓
TGT>./3workers ↓
PT>[ESC] 押下
PT>viewlog val( kmc_linux_event_data),val( kmc_linux_event_data_size) ↓
（イベント解析結果が表示されます）
PT>
PT>viewlog val( kmc_linux_event_data),val( kmc_linux_event_data_size) ↓
（イベント解析結果が更新されます）
PT>
PT>viewlog clr ↓
（イベント解析結果がクリアされます）
```

これで PARTNER のイベントトラッカーウインドウに情報が表示されます。

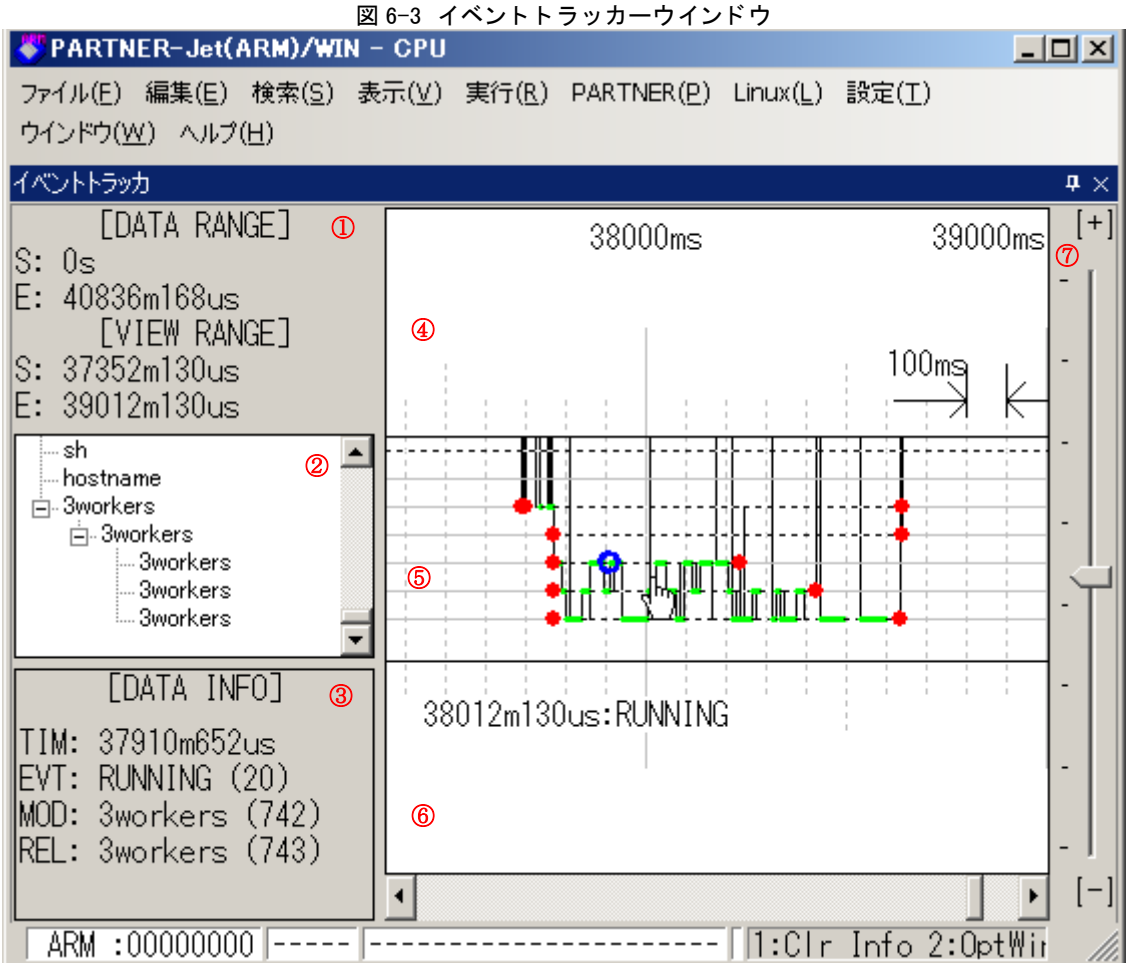
図 6-2 イベントトラッカーの情報表示



6.4 イベントトラッカーウインドウの操作

Linux OS 対応版のイベントトラッカーウインドウを図 6-3 に示します。イベントトラッカーウインドウは GUI によってきめ細かな表示の切り替えができますので、操作方法を図 6-3 に書き込んだ GUI エリア番号にしたがって説明します。

図 6-3 イベントトラッカーウインドウ



グラフ情報画面（①）

イベントトラッカーが保持しているデータの先頭時間、最後の時間、表示している時間の範囲を表示します。

モジュール（タスク）部分（②）

イベントトラッカーに登録されたモジュールが名前で表示されます。
モジュールをクリックして選択状態にし、マウスの副ボタンをクリック -> モジュールを隠す、または [DEL] キーでモジュールを非表示にします（表示しなくなるだけで、データは残っています）。

目盛り部分（④）

時間の目盛りと、ゲージの単位が表示されます。（ゲージの単位は表示 / 非表示可能）

グラフ部分（⑤）

マウスカーソルは通常手の形をしています。マウスの主ボタンクリックで掴んでマウス移動でスクロールします（図 6-4）。
マウスホイールの↑回転：ズームアップ（画面中央を中心にズームアップ 縮尺は x1/2）
マウスホイールの↓回転：ズームダウン（画面中央を中心にズームダウン 縮尺は x2）

ダブルクリックで近傍のイベントを選択し、カーソルを表示させます（図 6-5）。

[SHIFT] キー押下でマウスカーソルが十字になり、マーキーズームモードになります。[SHIFT]+ 範囲選択でその部分をズームします。マウスの主ボタンを押下して矩形表示中は、２ 点間の時間が左下に表示されます。[SHIFT] キーを放すとマーキーズームモードを終了しますので、簡単な２点間時間計測としても使用できます。[SHIFT] キーを押下したままマウス主ボタンを放すと範囲選択を確定してズームします。（図 6-6）

[CTRL] キー押下でマウスカーソルが矢印形になり、セレクトモードになります。[CTRL]+ マウスのドラッグで縦線を表示し、２本の縦線でイベント間時間を表します（図 6-7）。

タイムバー部分（⑥）

何も表示されていない箇所をダブルクリックすると時間計測用の線（タイムバー）が表示されます。１つが親となり絶対時間を指し、他のものは子となり親の絶対時間からの相対時間を指します（図 6-8）。
タイムバーをドラッグするとタイムバーが移動します（図 6-8 右下）。親を移動させると子のすべての相対時間が変わります。
タイムバーの近くをダブルクリックすると親のタイムバーに変更します。親であったものは子に変更されます。（図 6-9）
マウス副ボタンのクリックで表示のメニューから個別のタイムバーの削除や全てのタイムバーのクリアを行うことができます（図 6-8 最右のバーを図 6-9 で消去）。

ズーム用スライダー（⑦）

グラフ部分の時間単位の拡大 / 縮小ができます。マウスの副ボタンクリックで表示のメニューから非表示にすることも可能です。

図 6-4 グラフ移動（左右）

図 6-5 イベント選択

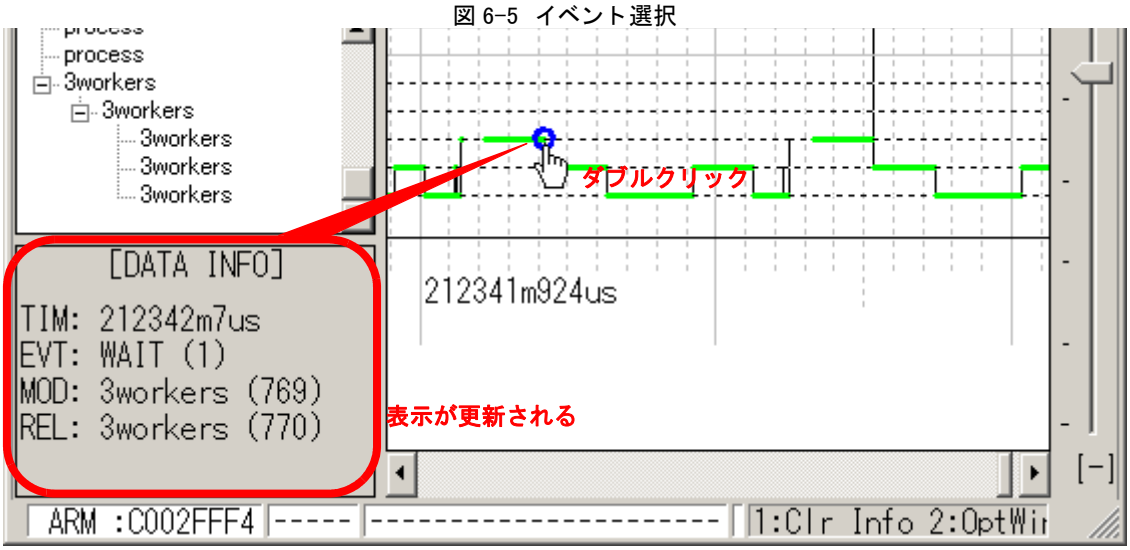


図 6-6［SHIFT- ドラッグ］マーキーズームで表示拡大

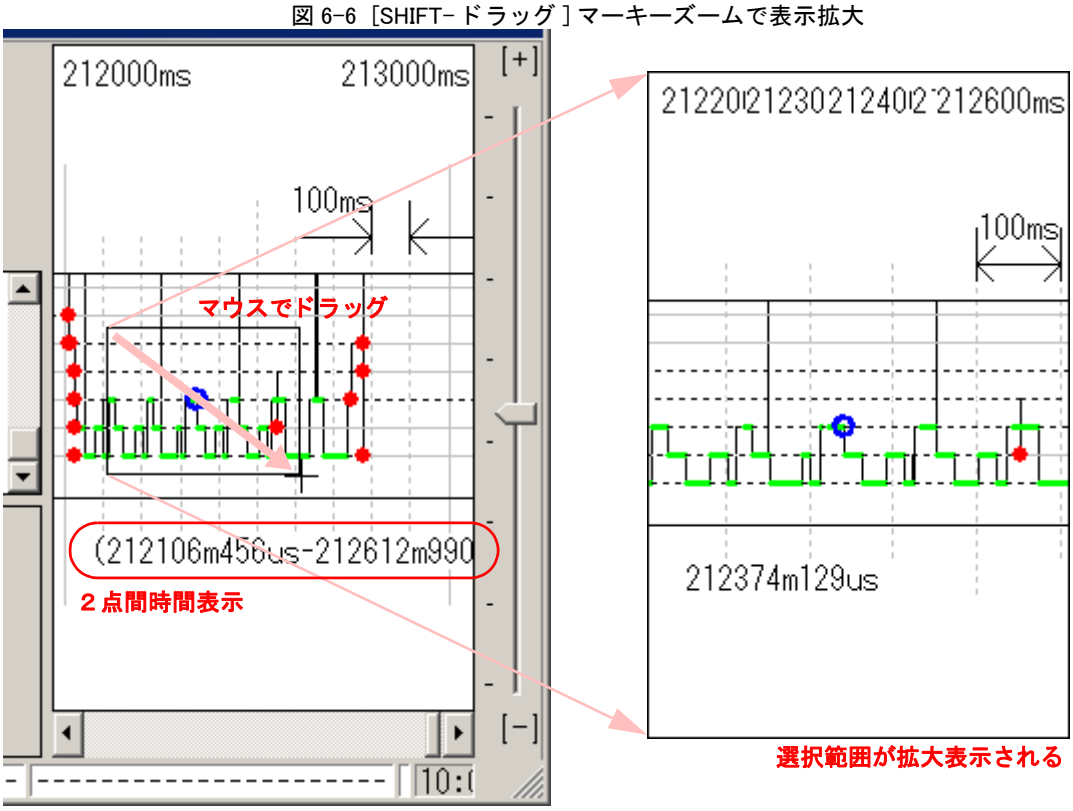


図 6-7［CTRL- ドラッグ］セレクトモードで時差計測

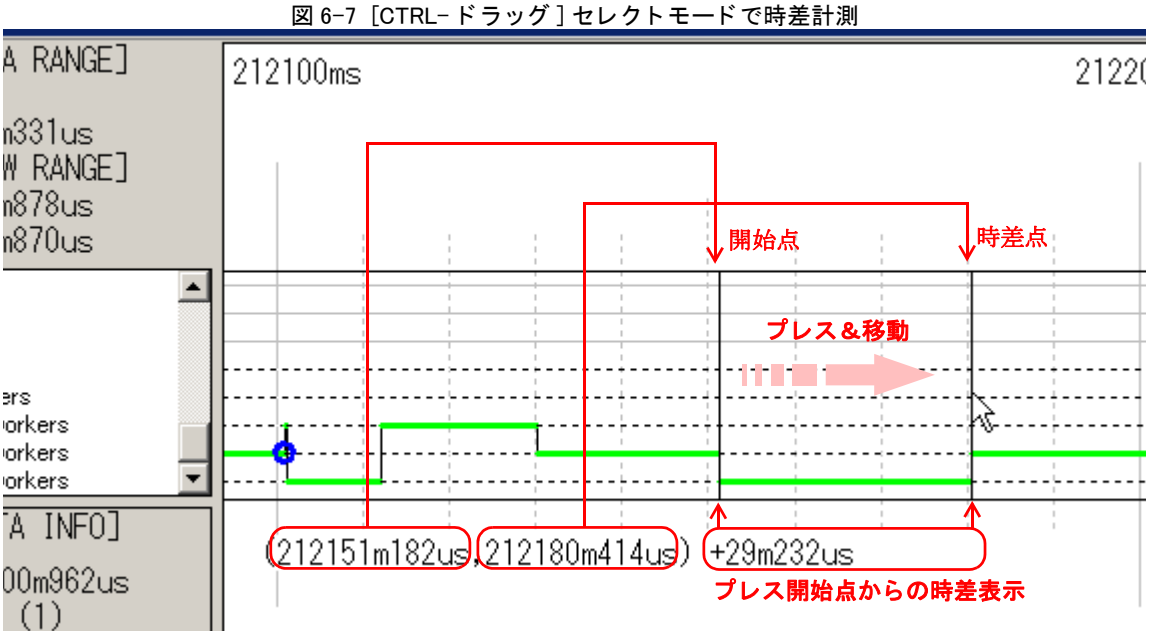


図 6-8 タイムバー表示

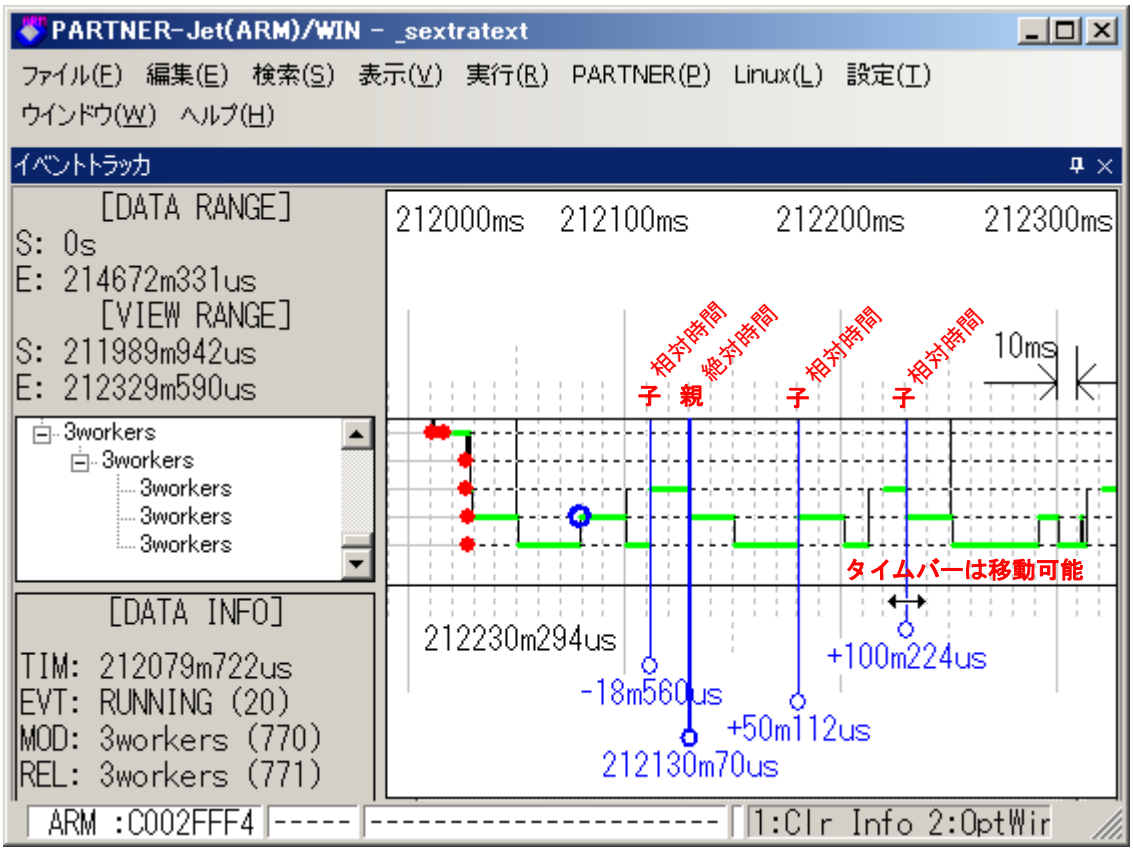
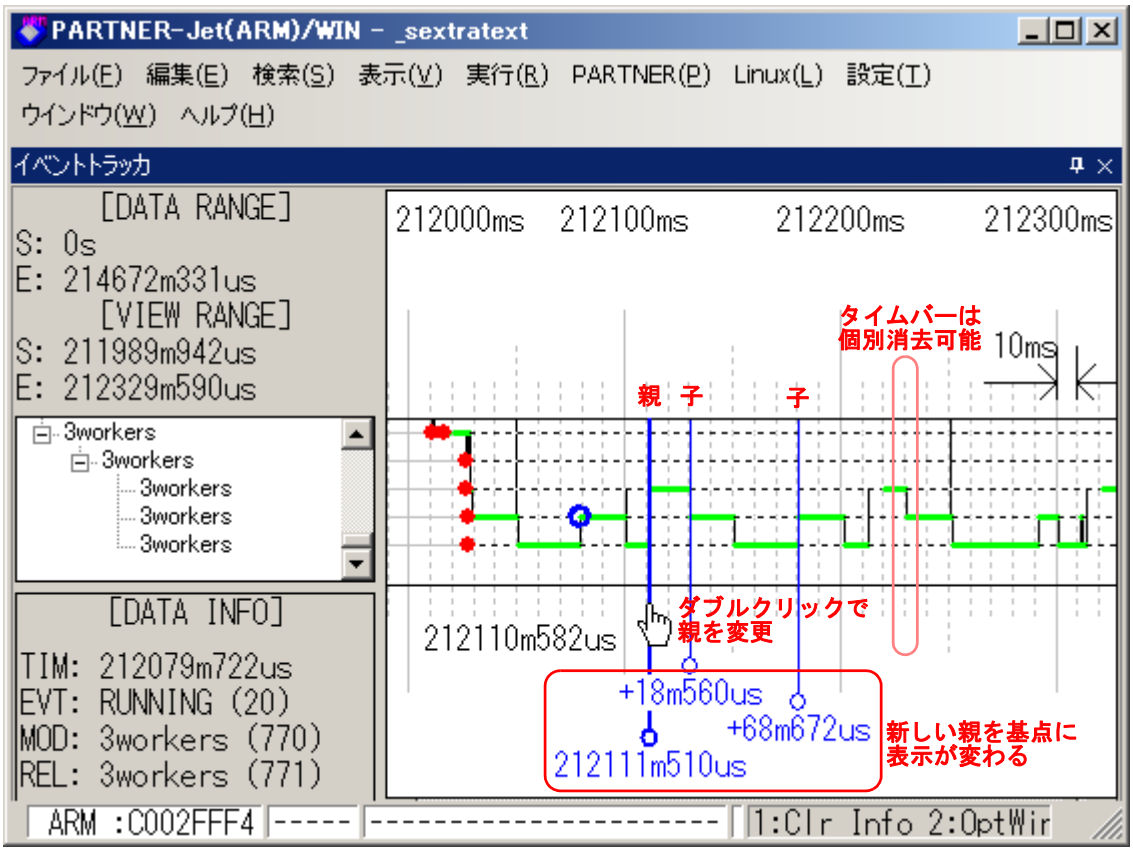


図 6-9 タイムバー表示の変更



6.5 イベントトラッカー用コマンド仕様

イベントトラッカー機能を操作するためには以下の VIEWLOG コマンドが追加されています。イベントトラッカーに関連する操作はすべて VIEWLOG コマンドで、その第 1 引数によって別の動作になります。つまりサブコマンドに分割されているといえます。

・VIEWLOG コマンド（213 頁）

VIEWLOG コマンド

書式 1　**VIEWLOG DLL <dll ファイルへのパス >**

書式 2　**VIEWLOG VAL(< シンボル名 >),VAL(< シンボル名 >)**

書式 3　**VIEWLOG CLR**

書式 4　**VIEWLOG PROF [<starttime>[,<endtime>]]**

書式 5　**VIEWLOG CSV <filename>[,<starttime>[,<endtime>]]**

書式 6　**VIEWLOG {SAVE|LOAD} <filename>[,<starttime>[,<endtime>]]**

書式 7　**VIEWLOG FIND [T<starttime>,<endtime>] [E< 対象イベント >] [M< 対象モジュール >] [F< 検索処理 >]**

機能

イベントトラッカー機能の操作

・イベントトラッカーの DLL 初期化 / 情報解析＆表示 / 解析情報削除

・イベントトラッカーの解析＆表示情報の保存 / プロファイル表示

・イベントトラッカーのコマンド情報のログ保存開始 / 保存終了 / 保存済み情報のロード＆実行

解説

書式 1 ～書式 3 はイベントトラッカーの機能の基本操作をするためのコマンドです。

書式 1 はイベント解析フィルタの DLL を PARTNER にロードします。イベントトラッカー機能を使用するには必ずイベント解析フィルタをロードしてから使用する必要があります。

書式 2 はイベント履歴バッファの内容を解析して PARTNER のイベントトラッカーウインドウに表示します。必ずイベント解析フィルタが PARTNER にロードされている状態で使用してください。引数にはターゲット上に組み込んだイベント履歴バッファのシンボル名を指定します。履歴バッファのシンボル名はカンマ「,」で区切ることで複数指定可能です。

書式 3 はイベントトラッカーが持っているデータを全てクリアします。DLL がロード時に登録したデータもクリアされるため、このコマンドを呼んだ後は再度書式 1 を実行してください。

書式 4 ～書式 6 はイベントトラッカーが表示している情報をさらに活用するための機能です。start-time,endtime はイベントラッカーのが表示している情報の一部分を指定するためのオプションで、指定しない場合は表示情報全体になります。

書式 4 はイベントトラッカーが表示している情報から、各モジュールのイベントの積算時間 (及び比率) をコマンドラインに表示します。

書式 5 はイベントトラッカーが表示している情報を CSV 形式でファイルに保存します。イベントトラッカーで収集したデータを他のアプリケーション (グラフや表計算等) で活用するための機能です。

書式 6 はイベントデータをバイナリ形式でセーブします。これによってターゲット上の OS を動作させなくても再び同じ画面を表示できます。フィルタ DLL の位置も記録しているため、ロード時には同じ位置にフィルタ DLL が必要になります。

書式 7 は収集済みのデータ内を検索する機能です。イベント、モジュールは、複数 (","区切り)、否定 (! を付ける) 及び範囲指定 (A-B で A ～ B の ID の範囲) が可能です。

IDを指定したい場合は、ID:<ID番号>を名前の代わりに入れるとIDとして認識します。逆にID:xxをIDでなく処理させる場合には、""で括ると、名前として処理されます。

検索処理には下記の項目が指定可能です。

ALL: 該当するもの全て
FIRST: 時間が一番早いもの
LAST: 時間が一番遅いもの
LONG: 次のイベントまでの時間が一番長いもの
SHORT: 次のイベントまでの時間が一番短いもの
SUM(MAXS): 積算時間が一番長いもの
MINS: 積算時間が一番短いもの
MAXC: イベント発生回数が一番多いもの
MINC: イベント発生回数が一番少ないもの

現状、FIRST,LAST,LONG,SHORT に関しては対象の各イベント、各モジュールにつき 1 つづつ該当するイベントを出力します。つまり、イベントが 2 つ、モジュールを 2 つ選択して検索処理を行うと、4 つ結果が出てくることになります。

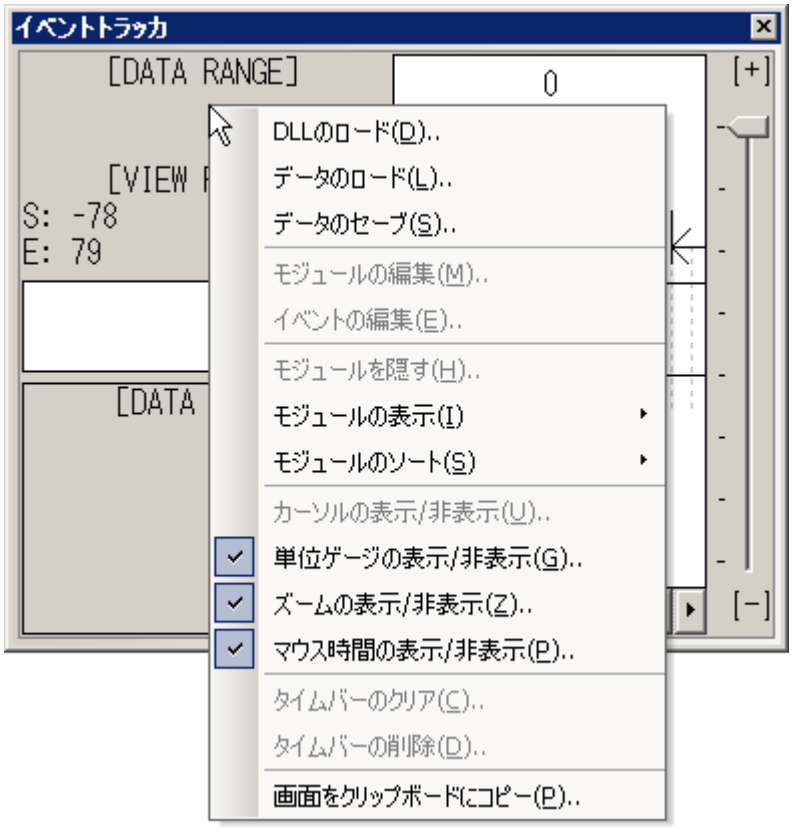
SUM,MINS,MAXC,MINC に関しては、該当する組み合わせを 1 つだけ出力し、その組み合わせに該当するイベントを画面上に表示します。

コラム 6-10 VIEWLOG コマンドと GUI の関係

PARTNER の GUI やメニューから行えるほとんどの操作は PARTNER のコマンドからも行うことができます。

イベントトラッカーの場合も同様の設計方針になっており、イベントトラッカー用の VIEWLOG コマンドは、イベントトラッカーウインドウ上でマウスの副ボタンクリックをすると表示できるポップアップメニューで行える操作に対応しています。

【例】
書式 1 は「DLL のロード」に対応。
書式 6 は「データのロード」「データのセーブ」に対応。



【使用例】

書式１～書式３イベントトラッカー機能の基本操作

```
PT>viewlog dll myfilter.dll ↓
TGT>./ls ↓
TGT>./pwd ↓
TGT>./3workers ↓
PT>[ESC]押下
(CPUをブレークした状態でイベント情報を表示を行います)
PT>viewlog val( kmc linux event data),val( kmc linux event data size) ↓
(イベント解析結果が表示されます)
PT>
PT>viewlog val( kmc linux event data),val( kmc linux event data size) ↓
(イベント解析結果が更新されます)
PT>
PT>viewlog clr ↓
(イベント解析結果がクリアされます)
```

書式４

```
PT>viewlog prof ↓
Start:0s End:32850m930us

Module: CPU (-20)
  RUNNING (20) : 722m888us (2%)
  WAIT (1) : 2463m263us (7%)

Module: PID: 0 (0)

Module: /linuxrc (1)
  WAIT (1) : 24241m372us (73%)

Module: PID: 0 (2)
  WAIT (1) : 24311m934us (74%)

Module: PID: 0 (3)
  RUNNING (20) : 630us (0%)
  WAIT (1) : 3186m880us (9%)

Module: PID: 0 (4)
  WAIT (1) : 28915m580us (88%)

Module: PID: 0 (5)
  WAIT (1) : 26342m55us (80%)

Module: khelper (6)
  END (2) : 32848m388us (99%)
～中略～
Module: 3workers (742)
  CREATE (0) : 11us (0%)
  EXEC (10) : 378us (0%)
  RUNNING (20) : 22m676us (0%)
```

```
    WAIT (1) : 834m382us (2%)
    END (2) : 1673m963us (5%)

  Module: 3workers (743)
    CREATE (0) : 69us (0%)
    RUNNING (20) : 1m199us (0%)
    WAIT (1) : 817m993us (2%)
    END (2) : 1674m53us (5%)

  Module: 3workers (744)
    CREATE (0) : 1m66us (0%)
    RUNNING (20) : 131m620us (0%)
    WAIT (1) : 2360m275us (7%)
    END (2) : 102us (0%)

  Module: 3workers (745)
    CREATE (0) : 681us (0%)
    RUNNING (20) : 187m626us (0%)
    WAIT (1) : 417m664us (1%)
    END (2) : 1886m415us (5%)

  Module: 3workers (746)
    CREATE (0) : 10m495us (0%)
    RUNNING (20) : 375m689us (1%)
    WAIT (1) : 431m371us (1%)
    END (2) : 1674m577us (5%)

  PT>
```

書式5

（イベント解析結果が表示されている状態で）

PT>viewlog csv mylog.csv ↓

（プロジェクトフォルダに mylog.csv が保存されます）

書式6

（イベント解析結果が表示されている状態で）

PT>viewlog save mylog.dat ↓

（プロジェクトフォルダに mylog.dat がバイナリ形式でファイルに保存されます）

PT>q ↓

（PARTNER を再起動します）

PT>viewlog dll myfilter.dll ↓

（mylog.dat 保存時と同じ dll をロードします。この操作は必須ではありませんが、dll が存在しているのを確認するため。）

PT>viewlog load mylog.dat ↓

（保存時と同じデータが表示されます）

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　トラブルシューティング

以下にトラブルシューティングの項目別一覧を掲載します。

A-1　カーネルデバッグ

表 A-1 カーネルデバッグトラブルシュート一覧

症状	チェックポイント	参照・参考
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。	2.2 Linux カーネルソースの修正と設定（50 頁）
	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。	-XGX オプション（140 頁）
	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 コマンド（168 頁）
	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。	—
ソースコード表示が実行内容と一致しない	実行しているバイナリとデバッグ情報がずれていないか確認してください。	—
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 ↓	—

A-2 ローダブルモジュールデバッグ

表 A-2 ローダブルモジュールトラブルシュート一覧

症状	チェックポイント	参照・参考
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。	(1) ローダブルモジュールソースの修正（83 頁）
	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。	(3) デバッグの準備（83 頁） -OS オプション（138 頁）
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。	(1) ローダブルモジュールソースの修正（83 頁）
	_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。	
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。	-XGX オプション（140 頁） -SK オプション（143 頁）
	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が表示しようとしているファイル名です。	SNAME コマンド（168 頁）
	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。	
strip したらデバッグできなくなった	ローダブルモジュールをデバッグするためにはモジュール内に「module_init」シンボルが必要なため、全てのシンボルを strip してしまうとデバッグできなくなります。 ローダブルモジュールを strip する際は -g オプションをつけてください。	アプリケーションの場合 LINUX>strip objfile ローダブルモジュールの場合 LINUX>strip -g objfile

A-3　アプリケーションデバッグ

表 A-3 アプリケーションデバッグトラブルシュート一覧

症状	チェックポイント	参照・参考
ソースコード表示が実行内容と一致しない	実行しているバイナリとデバッグ情報がずれていないか確認してください。	―
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（133 頁）
	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も一度はロードしたか確認してください。	ロードしていない場合は、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。
	デバッグサポートファイルを Preload ライブラリ方式で使用している場合は、/etc/ld.so.preload や環境変数 LD_PRELOAD が正しく設定されているか確認してください。 正しく設定されていないと、ATTACH コマンド（152 頁）を使用する際に「データ指定が誤りです」というエラーになります。	2.3 アプリケーションデバッグサポートファイル（58 頁） ATTACH コマンド（152 頁）
	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() または _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。	2.3 アプリケーションデバッグサポートファイル（58 頁）
	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（138 頁）
	【MIPS シリーズのみ】 カーネルモードでは自動アタッチ可能で、アプリケーションモードでは自動アタッチされない場合は、システムコール (gettid) が正しく利用できない可能性があります。 arch/mips/kernel/syscalls.h 内の gettid の宣言されている順番と、 include/asm-mips/unistd.h 内の gettid の宣言されている順番が一致しているか確認してください。	2.2.4 MIPS シリーズの注意事項（53 頁）

症状	チェックポイント	参照・参考
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 / スレッドのエントリーの先頭にデバッグスタブ関数 _kmc_start_debuger() または _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。	2.3 アプリケーションデバッグサポートファイル（58 頁）
	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（138 頁）
	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るかどうか確認してください。	MULTI コマンド（163 頁）
	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。 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。	PSID コマンド（158 頁）
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() または _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。	2.3 アプリケーションデバッグサポートファイル（58 頁）
	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（138 頁）
	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るか確認してください。	MULTI コマンド（163 頁）
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。	C-1 ターゲットシステムの glibc の修正（237 頁）
	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。	C-2 PARTNER への glibc の登録（238 頁）
	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。	PS コマンド（150 頁） ATTACH コマンド（152 頁）

A-4　リアルタイムトレース

表 A-4 リアルタイムトレーストラブルシュート一覧

症状	チェックポイント	参照・参考
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（138 頁）
	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。	2.2 Linux カーネルソースの修正と設定（50 頁）
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。	3.8 Linux OS 対応ヒストリ表示（117 頁）

付録 B　開発用 Linux PC の構築

『開発環境について（3 頁）』で使用している KZM-ARM11-01 ボード用のクロス開発環境（『図 1-1』参照）の開発用 Linux PC をセットアップする手順を紹介します。
使用する PC-Linux のディストリビューションやターゲット用の Linux ディストリビューションによっては多少手順やパス名などが異なる可能性がありますので必要な箇所は読み替えてください。

開発環境用 Linux PC のセットアップ方法を以下の項目で説明します。

B-1　Linux PC でクロス環境をセットアップする

クロス開発環境というからにはホスト PC(ix86 CPU) 上でターゲット CPU 用の命令コードを出力するようなソフトウェア (ツールチェイン) と、それを使ってクロスコンパイルしたソフトウェア環境が必要です。
クロスコンパイル用のソフトウェアとしては binutils や gcc といった GNU Tools を用いるのが一般的です。
GNU Tools のソースコードを入手してクロスコンパイル用ツールチェインをビルドするためには、以下のようなソフトウェアパッケージが必要です。(PC 用 Linux ディストリビューションによってパッケージ名は多少異なるかもしれません)

```
make(3.8 以上 ), gcc, autoconf, texinfo, ncurses(tic), bison, flex, zlib-devel
```

KZM-ARM11-01 ボードの場合は、ビルド済みバイナリが提供されていますので Linux PC にインストールするだけで済みます。
ここでは提供されているファイルが /tmp/ に置かれているものとして、/opt/kmc/kzm-arm11/ ディレクトリ以下にインストールしていきます。

(1) クロスツールチェインのインストール

スーパーユーザ (root) でツールチェインを展開します。

```
LINUX86>$ su ↓
LINUX86># cd / ↓
LINUX86># tar xvzf /tmp/kzm-arm11_toolchain-xxxxxx.tgz ↓
(/opt/kmc/kzm-arm11/staging_dir/ 以下に展開される )
```

クロスコンパイル用ツールの所在を確かめます。

```
LINUX86># ls /opt/kmc/kzm-arm11/staging_dir/bin ↓
arm-linux-addr2line  arm-linux-objcopy                arm-linux-uclibcgnueabi-gcc
arm-linux-ar         arm-linux-objdump                arm-linux-uclibcgnueabi-gcc-4.1.1
arm-linux-as         arm-linux-ranlib                 arm-linux-uclibcgnueabi-gccbug
arm-linux-c++        arm-linux-readelf                arm-linux-uclibcgnueabi-gcov
arm-linux-c++filt    arm-linux-size                   arm-linux-uclibcgnueabi-gprof
arm-linux-cc         arm-linux-strings                arm-linux-uclibcgnueabi-ld
arm-linux-cpp        arm-linux-strip                  arm-linux-uclibcgnueabi-nm
arm-linux-g++        arm-linux-uclibcgnueabi-addr2line arm-linux-uclibcgnueabi-objcopy
arm-linux-gcc        arm-linux-uclibcgnueabi-ar       arm-linux-uclibcgnueabi-objdump
arm-linux-gcc-4.1.1  arm-linux-uclibcgnueabi-as       arm-linux-uclibcgnueabi-ranlib
arm-linux-gccbug     arm-linux-uclibcgnueabi-c++      arm-linux-uclibcgnueabi-readelf
arm-linux-gcov       arm-linux-uclibcgnueabi-c++filt  arm-linux-uclibcgnueabi-size
arm-linux-gprof      arm-linux-uclibcgnueabi-cc       arm-linux-uclibcgnueabi-strings
arm-linux-ld         arm-linux-uclibcgnueabi-cpp      arm-linux-uclibcgnueabi-strip
arm-linux-nm         arm-linux-uclibcgnueabi-g++
```

クロスツールチェインを使用できるように PATH を通します。

```
LINUX86>$ export PATH=/opt/kmc/kzm-arm11/staging_dir/bin:$PATH ↓
```

もしくは、~/.bash_profile などに上記パスを追加し、source コマンドで環境ファイルを読み直します。

```
LINUX86>$ source ~/.bash_profile ↓
```

(2) ルートファイルシステムを構築

KZM-ARM11-01 ボードの場合は、Buildroot(http://buildroot.uclibc.org/) を使用した構築済みアーカイブが用意されていますので展開します。

```
LINUX86>$ su ↓
LINUX86># cd / ↓
LINUX86># tar xvzf /tmp/kzm-arm11_rootfs-xxxxxx.tgz ↓
(/opt/kmc/kzm-arm11/root/ 以下に展開される)
LINUX86>$ ls -F /opt/kmc/kzm-arm11/root/ ↓
bin/  etc/   lib/      mnt/  proc/  sbin/  tmp/  var/
dev/  home/  linuxrc@  opt/  root/  sys/   usr/
```

さらに NFS ルートファイルシステム用に必要な /dev/null と /dev/console デバイススペシャルファイルを作成します。

```
LINUX86>$ su ↓
LINUX86># cd /opt/kmc/kzm-arm11/root/dev/ ↓
LINUX86># tar zxvf /tmp/target_rootfs_dev-xxxxxx.tgz ↓
./console
./null
```

NFS ルートを使用しない場合でも、Windows PC から参照可能なルートファイルシステムのツリーを用意してください。例えば、ターゲットボードの HDD にデバッグ情報が無いファイルのみで構成されたルートファイルシステムがある場合、同じプログラムファイル群でデバッグ情報付きのファイルで構成されたルートファイルシステムツリーが Samba 共有、もしくは Windows PC の HDD に存在していれば結構です。PARTNER はアプリケーションデバッグ時に -RootDir オプションによってデバッグ情報付きのルートファイルシステムツリーからデバッグ情報を検索します。

(3) ターゲット用 Linux カーネルの準備

用意されているカーネルソースを展開します。

```
LINUX86>$ su ↓
LINUX86># mkdir -p /opt/kmc/kzm-arm11/build_src ↓
LINUX86># cd /opt/kmc/kzm-arm11/build_src ↓
LINUX86># tar xvzf /tmp/linux-2.6.16-xxxxxx.tgz ↓
(/opt/kmc/kzm-arm11/build_src/linux-2.6.16-xxxxxx/ 以下に展開される)
LINUX86># ln -s linux-2.6.16-xxxxxx linux ↓
(以後わかりやすいようにシンボリックリンクを張ります)
```

KZM-ARM11-01 ボード用に提供されている Linux カーネルには既に PARTNER デバッグ用のパッチが組み込まれています。デバッグ用パッチが組み込まれていないカーネルを使用する場合は、『2.2 Linux カーネルソースの修正と設定 (50 頁)』の手順で組み込んでください。

デフォルトの設定を生成します。

```
LINUX86># cd /opt/kmc/kzm-arm11/build_src/linux ↓
LINUX86># make kzm-arm11_defconfig ↓
```

カーネルのコンフィグレーションを行います。
[PARTNER Debugging] メニューの [Debug infomation type]、[Enable patch for PARTNER debug]、[Loadable module auto attach]、[Loadable module auto attach (patch is include in kernel)] を有効にしてください。
各項目の詳細は『カーネルコンフィグレーション (54 頁)』を参照してください。

```
LINUX86># make menuconfig ↓
```

カーネルをビルドします。

```
LINUX86># make ↓
```

デバイスドライバモジュールをルートファイルシステムにインストールします。

```
LINUX86># make INSTALL_MOD_PATH=/opt/kmc/kzm-arm11/root/ modules_install ↓
```

(4) アプリケーションデバッグサポートファイルの準備

デバッグサポートファイルを(『Preload ライブラリ方式 (60 頁)』を参照)してライブラリとしてビルドします。

```
LINUX86>$ tar zxvf libkmcsup.tgz ↓
LINUX86>$ cd libkmcsup ↓
LINUX86>$ make ARCH=arm CPU=arm11 ↓
                :
---- PARTNER COMMANDLINE ------------------------------
linux set_attach_offset libkmcsup.so.2.0.0 0x00000624
-------------------------------------------------------
```

ビルドしたライブラリをターゲット用ルートファイルシステムにインストールします。
どこに配置してもかまいませんが一般的なライブラリファイルを置く場所「/usr/lib/」に置いておきます。

```
LINUX86>$ su ↓
LINUX86># cp libkmcsup* /opt/kmc/kzm-arm11/root/usr/lib/ ↓
```

さらに PRELOAD ライブラリとして使用するために環境変数 LD_PRELOAD を設定します。

```
LINUX86>$ su ↓
LINUX86># echo "export LD_PRELOAD=/usr/lib/libkmcsup.so.2.0.0" >> ¥
 /opt/kmc/kzm-arm11/root/etc/profile ↓
```

KZM-ARM11-01 ボード用に提供されている Linux ディストリビューションは glibc を使用していない(uClibc を使用)ため、/etc/ld.so.preload ファイルでの設定ができません。ここでは /bin/sh が必ず使用する設定ファイル /etc/profile に LD_PRELOAD 環境変数を登録しましたが、デバッグを行うユーザが決まっているならば /home/foo/.profile などの各ユーザ用設定ファイルへの記述でもかまいません。

ここまでで Linux PC に必要なターゲット用のソフトウェアは揃いました。
インストールされたファイルの構成は『Linux PC 上のファイル配置 (4 頁)』と同じになります。

B-2　開発用 Linux PC で NFS Server をセットアップする

NFS をセットアップするためには NFS サーバソフトウェアをインストールする必要があります。

```
LINUX86>$ su ↓
LINUX86># apt-get install nfs-user-server ↓
（インストール方法とパッケージ名はディストリビューションによって異なるかもしれません）
```

次に、NFS で共有するディレクトリの設定をするために /etc/exports ファイルを編集します。ここでは /opt/ ディレクトリ以下を NFS で開示する設定にしています。
(/opt/kmc/kzm-arm11/ 以下のみでも不都合ありません)

```
LINUX86>$ grep kzm /etc/exports ↓
/opt    kzm-arm11(rw,sync,no_root_squash)
```

ターゲット上の root ユーザが root パーミッションのファイルを操作できるように、no_root_squash オプションを付けています。
NFS のプロトコルには 2,3,4 のプロトコルバージョンがあり、使用するバージョンによって /etc/exports の書式が異なります。不都合が無ければ低いバージョンの動作をサポートする設定にし、ワイルドカード「*」等の書式は使用しないほうが無難だと思われます。書式の詳細は man(5) exports を参照してください。動作中のプロトコルバージョンは rpcinfo コマンドなどでご確認ください。

/etc/exports ファイルに記述した「kzm-arm11」はターゲットボード名です。/etc/hosts ファイルに記述するか、DNS への登録をします。

```
LINUX86>$ grep kzm /etc/hosts ↓
192.168.1.202 kzm-arm11
```

IP アドレスはスクリーンショットにも表示される箇所がありますので、使用しているアドレスを参考用に記載します。

表 B-1 IP アドレス設定例

機器	IP address	name
開発用 Linux PC	192.168.1.16	linuxpc
ターゲットボード KZM-ARM11-01	192.168.1.202	kzm-arm11
デバッグ用 Windows PC	192.168.1.XXX	-

設定ができたら NFS サーバを起動します。

```
LINUX86>$ su ↓
LINUX86># /etc/init.d/nfs-user-server start ↓
Starting NFS servers: nfsd mountd.
```

これで NFS サーバの設定は完了です。

B–3 ターミナルソフトをセットアップする

ターゲットボードが使用するシリアルコンソールを使用できれば良いので、ターミナルソフトウェアは何でもかまいません。

KZM-ARM11-01 ボードの場合は、表 B-2 のようにシリアルポートの設定をします。

表 B-2 シリアルポートの通信設定

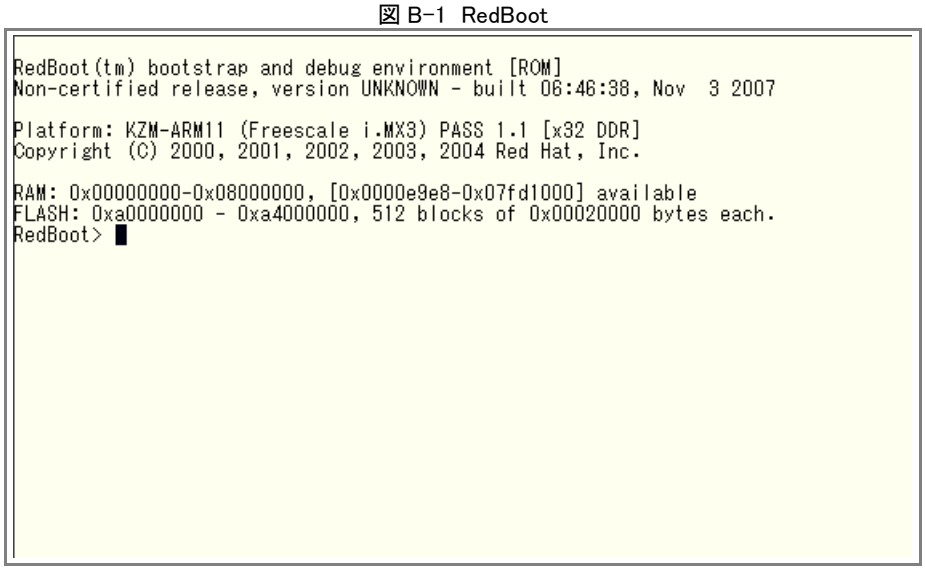
項目	設定値
通信速度	115200bps
データ	8bit
パリティ	なし
ストップ	1bit
フロー制御	なし

B-4　ターゲットで Linux を起動してみる

KZM-ARM11-01 ボードは、電源を入れると RedBoot プログラムが起動するように設定された状態で出荷されています（『図 B-1』参照）。RedBoot から NFS ブートすることができますのでこの時点で Linux が起動するかどうかを試してみることができます。

（評価用ボードの場合は何らかのモニタプログラムが搭載された状態で出荷されることは良くあります。モニタプログラムが NFS ブートや tftp ブートのようなネットワークブート機能をサポートしていれば同様の確認を行えると思います。詳しくはお使いのボードの仕様をご確認ください。）

図 B-1　RedBoot

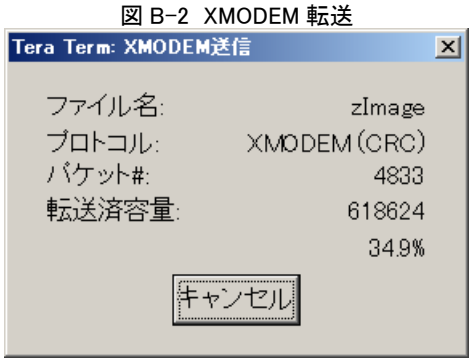


RedBoot では XMODEM プロトコルで Linux カーネルイメージを RS-232C ケーブル経由で RAM 上に転送できます。RS-232C ケーブルをデバッグ用 Windows PC に接続している場合はあらかじめ開発用 Linux PC からファイルを転送しておき、RedBoot のプロンプトで load コマンドを使用します。

```
RedBoot> load -r -b 0x100000 -m xmodem ↓
```

待ち受け状態になりますので、通信ソフトからファイルを送信します。例えば TeraTerm ならば [File]-[Transfer]-[XMODEM]-[Send] を選択して、[Option]-[Checksum] を付けて書き込むファイル [zImage] を指定します。

図 B-2　XMODEM 転送




```
CCRaw file loaded 0x00100000-0x002b0d43, assumed entry at 0x00100000
xyzModem - CRC mode, 13851(SOH)/0(STX)/0(CAN) packets, 10 retries
```

転送が完了したら、NFS ルートファイルシステムをカーネルパラメータに設定して起動します。

```
RedBoot> exec -b 0x00100000 -l 0x00200000 -c "noinitrd console=ttymxc0 root=/dev/nfs
nfsroot=192.168.1.16:/opt/kmc/kzm-arm11/root init=/linuxrc ip=192.168.1.202::::kzm-arm11"
↓
（ここまで一行です）
Uncompressing Linux.....................................................................
..................................................... done, booting the kernel.
Linux version 2.6.16.19-kzm (foobar@linuxpc) (gcc version 4.1.1) #2 PREEMPT Mon Dec 10
02:46:38 JST 2007
CPU: Some Random V6 Processor [4107b364] revision 4 (ARMv6TEJ)
Machine: KMC KZM-ARM11-01
Memory policy: ECC disabled, Data cache writeback
CPU0: D VIPT write-back cache
CPU0: I cache: 16384 bytes, associativity 4, 32 byte lines, 128 sets
CPU0: D cache: 16384 bytes, associativity 4, 32 byte lines, 128 sets
Built 1 zonelists
Kernel command line: noinitrd console=ttymxc0 root=/dev/nfs nfsroot=192.168.1.16:/opt/kmc/
kzm-arm11/root init=/linuxrc ip=192.168.1.202::::kzm-arm11
```

ブートプロンプトが表示されたら、ログインして正しくファイルの読み書きができているか確認します。

図 B-3 Linux の起動

```
TCP established hash table entries: 4096 (order: 2, 16384 bytes)
TCP bind hash table entries: 4096 (order: 2, 16384 bytes)
TCP: Hash tables configured (established 4096 bind 4096)
TCP reno registered
TCP bic registered
NET: Registered protocol family 1
VFP support v0.3: implementor 41 architecture 1 part 20 variant b rev 2
IP-Config: Guessing netmask 255.255.255.0
IP-Config: Complete:
      device=eth0, addr=192.168.1.202, mask=255.255.255.0, gw=255.255.255.255,
     host=kzm-arm11, domain=, nis-domain=(none),
     bootserver=255.255.255.255, rootserver=192.168.1.16, rootpath=
Looking up port of RPC 100003/2 on 192.168.1.16
Looking up port of RPC 100005/1 on 192.168.1.16
VFS: Mounted root (nfs filesystem).
Freeing init memory: 112K
Starting the hotplug events dispatcher udevd
Synthesizing initial hotplug events
Initializing random number generator... done.


Welcome to the Erik's uClibc development environment.

kzm-arm11 login:
```

B-5　Samba Server をセットアップする

Samba をセットアップするためには Samba サーバソフトウェアをインストールする必要があります。

```
LINUX86>$ su ↓
LINUX86># apt-get install samba samba-doc ↓
（インストール方法とパッケージ名はディストリビューションによって異なるかもしれません）
```

次に、共有するディレクトリの設定をするために /etc/samba/smb.conf ファイルを編集します。

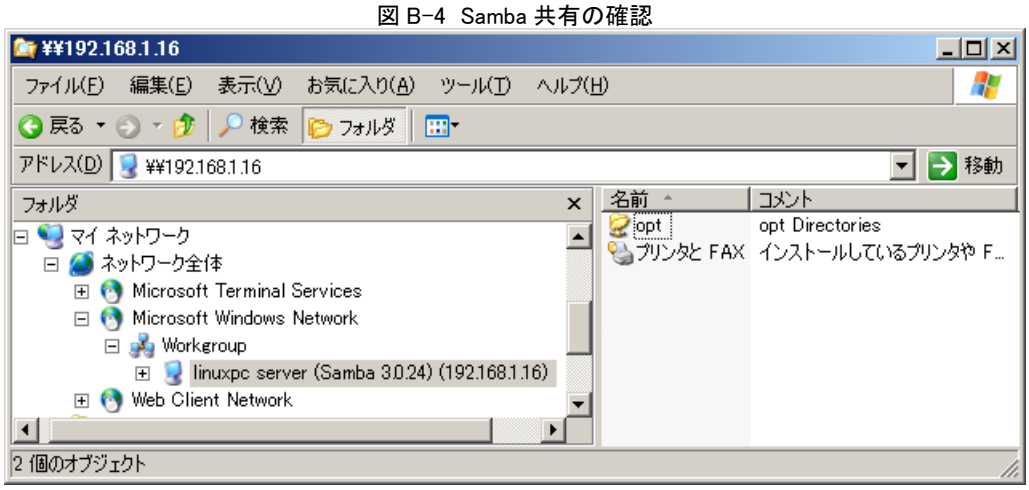
```
LINUX86>$ su ↓
LINUX86># vi /et/samba/smb.conf ↓
~ [global]
~    workgroup = WORKGROUP
~    server string = %h server (Samba %v)
~    security = share
~    socket options = TCP_NODELAY
~ [opt]
~    comment = opt directory
~    path = /opt
~    browseable = yes
~    guest ok = yes
~    public = yes
~    writable = yes
~    create mask = 0777
~    directory mask = 0777
~    available = yes
```

この例では /opt/ ディレクトリ以下を開示する設定にしています。
これはセキュリティを考慮した設定ではありませんし、書き込み許可はしなくてもかまいませんので man (7) samba 等を参照して適切な設定をしてください。設定できたら Samba サーバを起動します。

```
LINUX86>$ su ↓
LINUX86># /etc/init.d/samba start ↓
Starting Samba daemons: nmbd smbd.
```

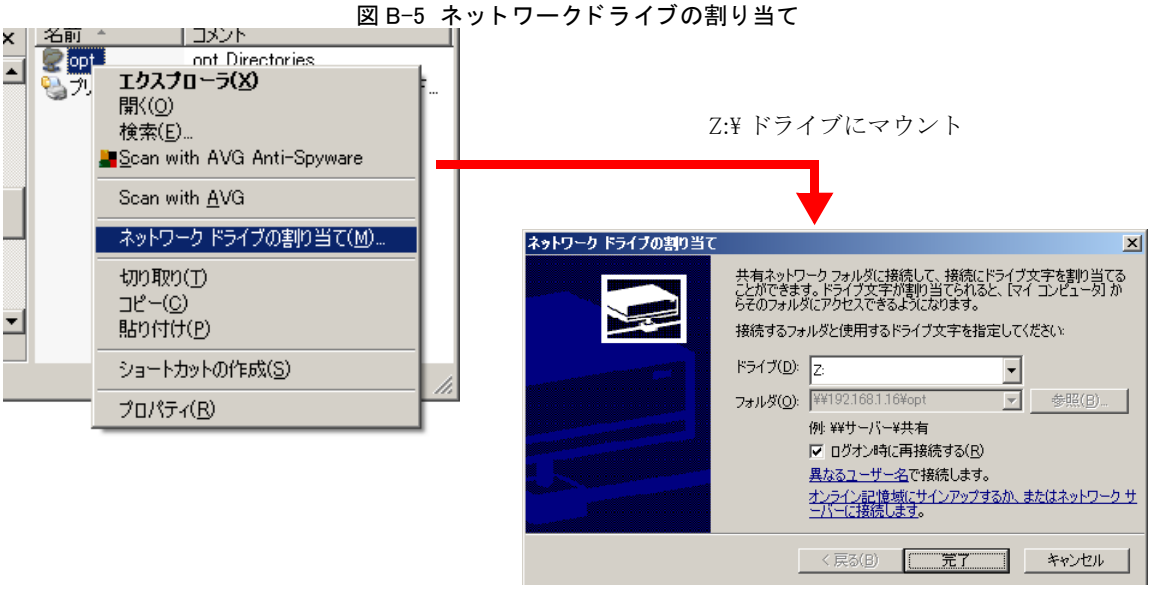
デバッグ用 Windows PC から参照できるかエクスプローラで確認します。

図 B-4 Samba 共有の確認



opt ディレクトリにアクセスできることを確認したら、ネットワークドライブの割り当てをします。

図 B-5 ネットワークドライブの割り当て



開発用 Linux PC の /opt/kmc/kzm-arm11 ディレクトリをデバッグ用 Windows PC からは z:¥kmc¥kzm-arm11 で参照できるようになりました。

付録 C　サポートファイルを glibc に入れる方法

PARTNER のアプリケーションデバッグ用サポートファイルはよく使用される共有ライブラリの中に入れておくこともできます。

この節では、glibc にサポートファイルを入れる方法を説明し、サポートファイルをアプリケーションに直接リンクする方法では実現できない、実行中のアプリケーションをアタッチしてデバッグする方法を例に次の流れで説明します。

・**ターゲットシステムの glibc の修正**（237 頁）

・**PARTNER への glibc の登録**（238 頁）

・**デバッグ方法**（238 頁）

C-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/<cpu> 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】 Makefile 修正

```
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。

C-2　PARTNER への glibc の登録

LINUX コマンド（154 頁）で、『C-1 ターゲットシステムの glibc の修正（237 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（237 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 ↓
000f2b54 t __kmc_sleep_thread
PT>linux set_attach_offset libc-2.2.5.so 0xf2b54 ↓
    SET LINUX ATTACH OFFSET : libc-2.2.5.so(0x000F2B54)
```

（表示される内容は CPU 種別等によって異なります。）

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（154 頁）を省略することが出来ます。

C-3　デバッグ方法

『3.4 アプリケーションのデバッグ（91 頁）』もしくは『5.1 アプリケーションモードデバッグ（172 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。
アプリケーションをデバッグする手順はアプリケーションデバッグサポートファイルを Preload して使用する場合と同じになります。
『アプリケーションのデバッグ（91 頁）』『マルチスレッドアプリケーションデバッグ（98 頁）』『マルチプロセスアプリケーションデバッグ（108 頁）』等を参照してください。

付録 D　動的なリンカローダ (ld.so) のデバッグ

動的なリンカローダ (ld.so) のデバッグは、『3.7 共有ライブラリのデバッグ（114 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。

(1) リンカローダ(ld.so)にデバッグ情報をつける

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。

(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())を記述しないでください。

(3) リンカローダ内のシンボル _dl_start のアドレス確認

『(1) リンカローダ(ld.so)にデバッグ情報をつける (240 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]#
```

(4) カーネルの実行

『3.4 アプリケーションのデバッグ (91 頁)』の『(4) アプリケーションデバッグ情報のロード (94 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。

(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 ↓
00008000-0000f000 r-xp /sbin/init
00016000-00017000 rw-p /sbin/init
00017000-0001b000 rwxp
40000000-40016000 r-xp /lib/ld-2.2.5.so
4001d000-4001e000 rw-p /lib/ld-2.2.5.so
4001e000-4001f000 rwxp
4001f000-4012b000 r-xp /lib/libc-2.2.5.so
4012b000-4012f000 ---p /lib/libc-2.2.5.so
4012f000-40138000 rw-p /lib/libc-2.2.5.so
40138000-4013c000 rw-p
bffff000-c0000000 rwxp
```

（表示される内容は CPU 種別や MAP フィールドの設定によって異なります）

(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。

(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、『(3) リンカローダ内のシンボル _dl_start のアドレス確認（240 頁）』で確認したオフセット値と『(5) リンカローダの .text 開始位置の確認（240 頁）』の .text 開始位置を足した値です。

```
PT>br 0x1eac + 0x40000000.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。

(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 ↓
```

(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=0x40000000 ↓
```

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。

(10) ハードウェアブレークの解除

『(7) ハードウェアブレークの設定（241 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 * ↓
```

(11) プロセスのアタッチ

PSID コマンド（158 頁）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 : 00008000-00009FFF
APPLI. AREA : 00011000-00051FFF
APPLI. AREA : 00053000-00053FFF
APPLI. AREA : BFFFF000-BFFFFFFF
（表示される内容は CPU 種別や MAP フィールドの設定によって異なります）

(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します。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 ↓

付録 E　技術解説：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、なぜデバッガに Linux 対応機能が必要なのか、Linux 対応 PARTNER の機能がどのような技術的背景によって成り立っているか、また Linux 対応機能によってあらたに実現できる組み込み Linux の開発・デバッグ内容などを説明します。

E-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、一貫性のある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 E-1 参照)。

表 E-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)	ターゲット上で 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 E-1 参照) では、次のような問題点が挙げられます。

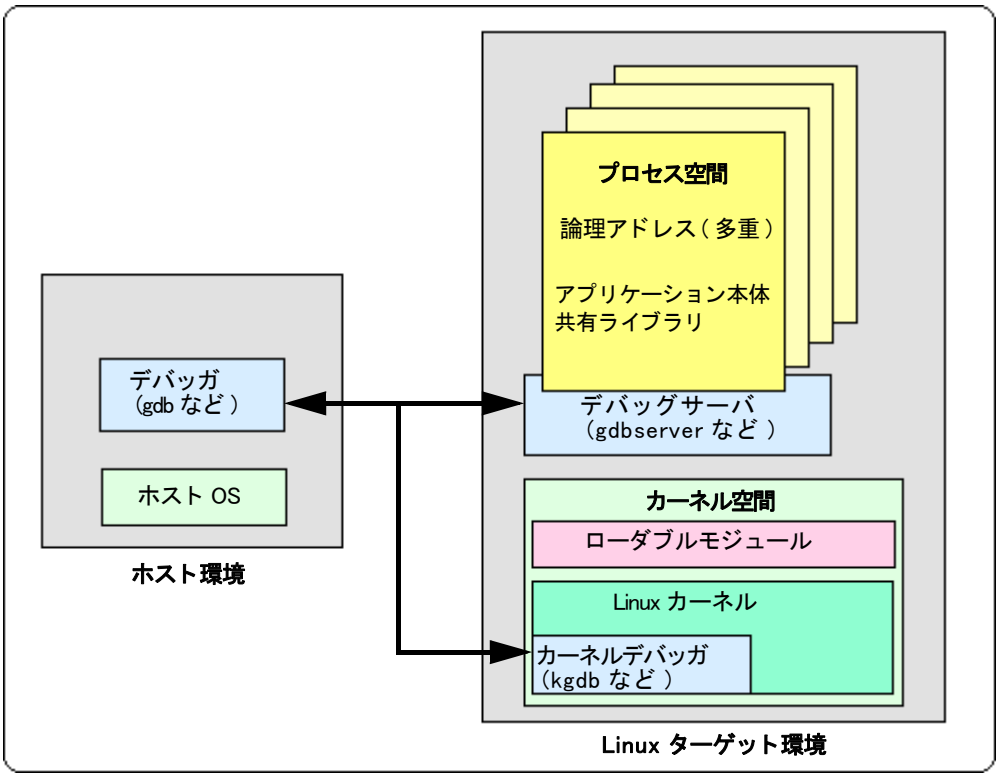


図 E-1 組み込み Linux の一般的なデバッグ環境例

● 一貫性のあるデバッグが困難

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。必然的にカーネル / 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、デバッグサーバを使用するデバッグ機能の場合は、カーネルのデバッグ中にカーネルがブレークした時には、デバッグサーバが動作できない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ので、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

E-2　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『E-3　カーネルモードデバッグの詳細（249 頁)』『E-4　アプリケーションモードデバッグの詳細（250 頁)』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います。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 は論理アドレスのみを扱うため、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。

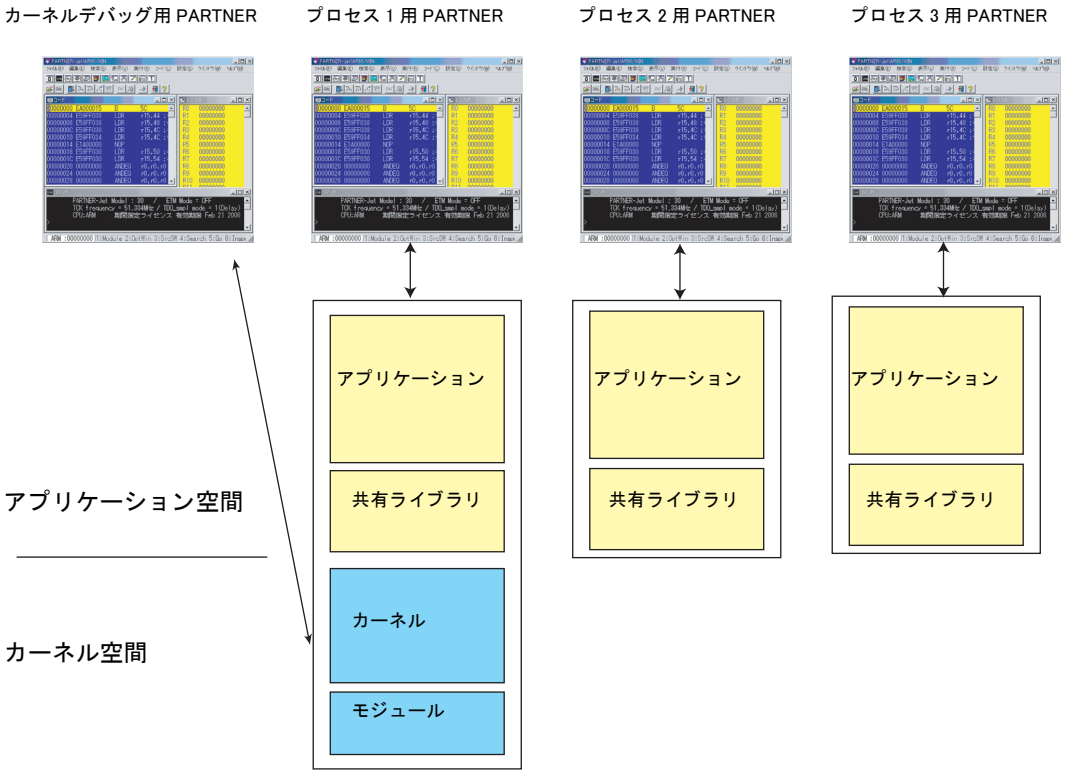
● 1 つのデバッガ (PARTNER) ですべてのデバッグが可能

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。
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透過的にデバッグすることができます。

● 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 E-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)。

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 (『E-4 アプリケーションモードデバッグの詳細 (250 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

E-3 カーネルモードデバッグの詳細

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。
ということは、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 に全く依存せずに PARTNER に実装されていますので、Linux カーネルにデバッグのための変更を加える必要はありません。しかし、後述する『2.2 Linux カーネルソースの修正と設定 (50 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。

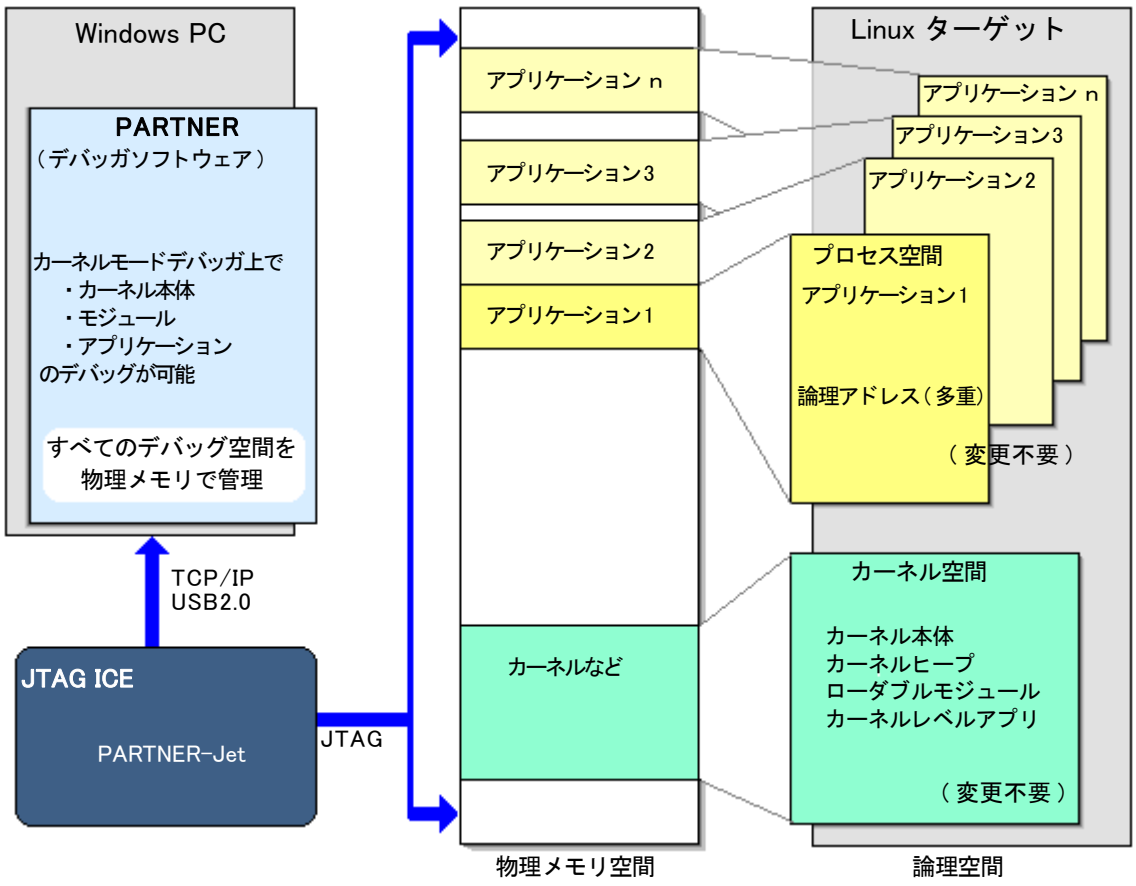


図 E-3 カーネルモードデバッグの概念

E-4　アプリケーションモードデバッグの詳細

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。
「独立」とはつまり、あるプロセスをデバッグするために停止（ブレーク）させ、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がありません。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、通信がオーバー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。

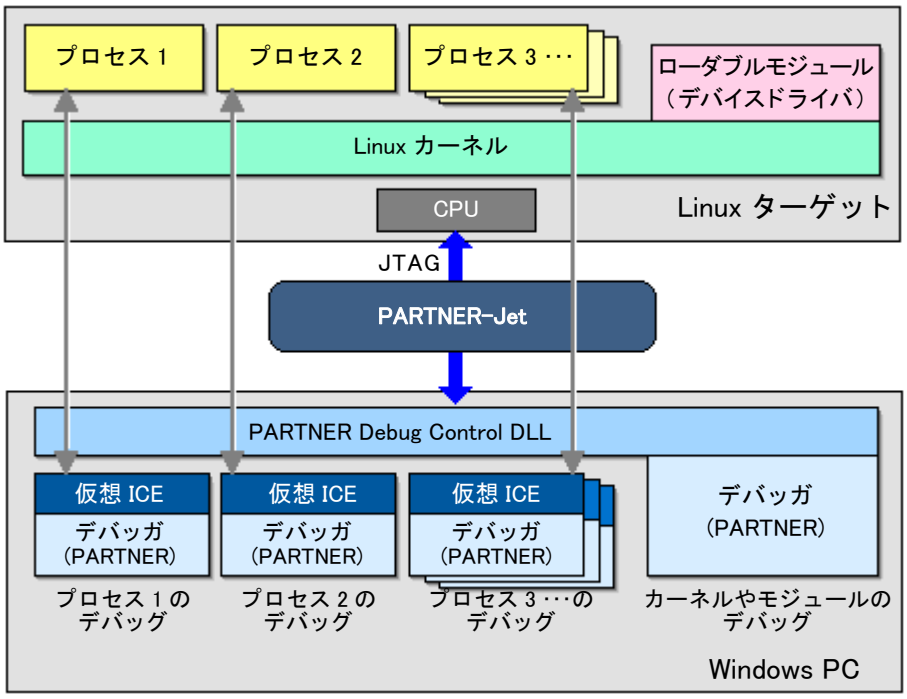


図 E-4　アプリケーションモードデバッグの概念

付録 F　PARTNER の動作モード選択について

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、スレッドデバッグ機能に NON_ADD モードと ADD モードの 2 種類があります（『-OS オプション（138 頁）』参照）。つまり、デバッグの対象や使い勝手の好みで切り替えられるモードの組み合わせは「カーネル (NON_ADD) モード」「カーネル ADD モード」「アプリケーション (NON_ADD) モード」「アプリケーション ADD モード」の 4 組になります。
推奨は「カーネル ADD モード」ですが、もし設定の選択に迷う場合は本節の記述を参考にしてください。

F-1　各モードの特徴と使用目的

カーネルモード

カーネルモードはブレーク時に CPU を停止させるモードです（詳細は『技術解説：Linux アーキテクチャと PARTNER（243 頁）』参照）。カーネルやデバイスドライバからユーザーモードアプリケーションまで統一的にデバッグ可能です（カーネル空間だけでなくユーザー空間のデバッグもできます）。
アプリケーションをデバッグ中にシステムコールの内部に入り、さらにデバイスドライバの処理を確認して再びアプリケーションに戻る、といったことが簡単にできます。

アプリケーションモード

アプリケーションのデバッグ中には、CPU やカーネルを停止させずにアプリケーションのみを停止させながらデバッグしたい場合があります。
例えば、下記のようなケースが想定できます。

- **アプリケーションのブレーク中も時計やファイル共有サービスは動作していて欲しい**
- **X-Window 上の GUI アプリケーション (X-Client) をデバッグする (X-Server は動作中であって欲しい)**

このようなアプリケーション単体のデバッグをするためには一般的には GDB をはじめとするターゲット上のアプリケーションとして実装されたユーザーモードデバッガが用いられますが、オーバーヘッドや様々な制限があります。PARTNER のアプリケーションモードは JTAG デバッガの性能を保ったまま個別のプロセスをブレークをできるようにした、アプリケーションデバッグのための機能です。

NON_ADD モード

マルチスレッド / マルチプロセスのアプリケーションをデバッグするときに、実行コンテキスト毎に PARTNER ウインドウを自動的・強制的に分けたい場合に使用します。新規実行コンテキストの生成時には自動的に未使用の PARTNER ウインドウが割り当てられてブレークします。
また、ユーザー空間のブレークポイントは PARTNER ウインドウ毎に閉じています（同一アドレスでもコンテキスト毎に停止）。

ADD モード

NON_ADD モードと違い、実行コンテキスト毎の PARTNER ウインドウの分割を強制されません。新規実行コンテキストの生成時の自動処理は行われませんので、停止したい場合はブレークポイントを設定する必要があります。
また、ユーザー空間のブレークポイントは PARTNER ウインドウ毎に閉じていません。

F-2　起動方法と必要な条件

カーネルモードでの起動

『2.2 Linux カーネルソースの修正と設定（50 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が前提です。Linux カーネルの設定メニュー（『カーネルコンフィグレーション（54 頁）』参照）の PARTNER に必要な設定を行い、コンフィグレーション設定後、Linux カーネル（vmlinux）を作成してください。

カーネルモードでデバッグするためには、PARTNER の起動設定（『PARTNER の設定と起動（70 頁）』参照）の OS デバッグモード指定（『 -OS オプション（138 頁）』参照）で「カーネル（NON_ADD）モード」または「カーネル ADD モード」を選択します。

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。

アプリケーションモードでの起動

『2.2 Linux カーネルソースの修正と設定（50 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が前提です。Linux カーネルの設定メニュー（『カーネルコンフィグレーション（54 頁）』参照）の PARTNER に必要な設定を行い、コンフィグレーション設定後、Linux カーネル（vmlinux）を作成してください。

アプリケーションモードでデバッグするためには、PARTNER の起動設定（『PARTNER の設定と起動（70 頁）』参照）の OS デバッグモード指定（『 -OS オプション（138 頁）』参照）で「アプリケーション（NON_ADD）モード」または「アプリケーション ADD モード」を選択します。

Linux カーネルのコンフィグレーション設定の必要な設定項目はデバッグしたい対象範囲に依存するので、PARTNER の起動モードとは関係ありませんが、アプリケーションモードで起動する目的はアプリケーションのデバッグですので [Debug infomation type] と [Enable patch for PARTNER debug] を有効にしてください。

F-3 動作モード選択の指針

最適な動作モード選択のための指針となるよう、それぞれのモードのデバッグ対象への適性を表 F-1 にまとめます。

表 F-1 各デバッグモードの適性

デバッグ対象 ＼ PARTNER モード	カーネル (NON_ADD) モード	カーネル ADD モード	アプリケーション (NON_ADD) モード	アプリケーション ADD モード
カーネルの中のデバッグ	◎	◎	○	○
ローダブルモジュールのデバッグ	◎	◎	○	○
アプリケーションのデバッグ	○	○	○	○
マルチスレッドアプリケーションのデバッグ	○	○	○	○
アプリケーションからシステムコールの中にステップインして動作を追う	◎	◎	△	△
デバイスドライバやサーバープログラムを動作させつつプロセスをブレークしデバッグする	×	×	◎	◎
ソフトウェアブレークでの複数の実行コンテキストの同時停止	◎	◎	×	×
多数のスレッドの同時デバッグ*	△	○	△	○
実行中のアプリケーションへのアタッチ	○	○	○	○
共有ライブラリのデバッグ	○	○	○	○
共有ライブラリ中のマルチスレッドデバッグ	○	○	○	○

* MULTI コマンド（163 頁）による PARTNER ウインドウ数の制限による

もし PARTNER の動作モードを何にするか迷う場合は、総合的に見て「カーネル ADD モード」をお勧めします。

F-4　Linux デバッグ動作モードの挙動と遷移

PARTNER の動作モードの挙動やモードの切り替え方法を表 F-2 にまとめます。

表 F-2 各デバッグモードの挙動と遷移

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

カーネルモードとアプリケーションモードの変更には全デバッガの再起動が必要です。
カーネルモードまたはアプリケーションモードで実行中の add モードの変更は、PARTNER のコマンドウインドウから変更可能です（詳細は『PSID コマンド（158 頁）』参照）。

・シングルモードから ADD モードへの遷移

```
PT>psid add
```

・ADD モードからシングルモードへの遷移

```
PT>psid non_add
```

付録 G　カーネルツリーのコンパイルについての補足

本節ではカーネルツリーをコンパイルする際のトピックを記述します。

G-1　最適化コンパイルを抑制する

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ことがあります。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。最適化コンパイルを抑制してビルドすることで問題を回避できる場合があります。

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))
```

最適化を抑制するとカーネルの実行速度が遅くなります。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。

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。

カーネル作成時の注意事項

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箇所があります。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_ で判断している箇所の最適化なしの場合の記述を追加してください。

付録 H　セグメンテーションフォルトの原因調査方法の Tips

アプリケーションが「動作中に落ちる」不具合で、直接の原因としては「セグメンテーションフォルトが発生したため」ということだけわかっているという状況はありがちだと思います。
コアダンプや動作ログの採取などが困難な環境では原因の究明が難しかったりするものです。

本節ではセグメンテーションフォルトを調査する際のトピックを記述します。

H-1　考え方

不具合の原因を見つけにくい一番の要因は不具合が発生したときには既にアプリケーションのプロセスが存在していないからです。
不具合発生からプロセス終了までの大まかな流れは以下のようになります。

1. Linux カーネルはセグメンテーションフォルトを検出すると、アプリケーションプロセスに対して SIG_SEGV シグナルを送信します。
2. アプリケーションが SIG_SEGV を受けとった時のデフォルトの動作は Core ダンプです。
3. シグナルハンドラを設定していないアプリケーションではプロセスが終了します。

PARTNER を使用ている場合は、アプリケーションプロセスも Linux カーネルも同時にデバッグできるのが強みです。アプリケーションプログラムに修正を加えなくても上記の 1 のタイミングを捉えることができます。

H-2　SIG_SEGV を捉える

SIG_SEGV を追う手順を以下に示します。

(1) カーネル内の SIG_SEGV 検出箇所にソフトウェアブレークを設定しておく

動作中のいつ SIG_SEGV が発生するかは予測できないので、常に設定しておきます。ソフトウェアブレークの設定場所は CPU によって異なります。

【ARM CPU の場合】
linux/arch/arm/mm/fault.c の __do_suer_fault()
【SH CPU の場合】
linux/arch/sh/mm/fault.c の bad_area ラベル
【MIPS CPU の場合】
linux/arch/mips/mm/fault.c の bad_area ラベル

(2) ブレークしたら、新規に PARTNER ウィンドウを一つ開く

Windows のスタートメニューから起動や、MULTI コマンド（163 頁）を使用してデバッガウィンドウを増やします。

(3) カレントプロセスにアタッチ

そのときの PARTNER のコマンドウインドウのレジスタリストに表示されている PID を指定し、カレントプロセスにアタッチします。

```
PT>attach <pid> ↓
```

(4) 共有ライブラリのデバッグ情報自動読み込み

アタッチしたプロセスの共有ライブラリのデバッグ情報を読み込みます。

```
PT>linux load so ↓
```

(5) 例外箇所近辺の調査

ここまでの手順で SIG_SEGV で「落ちる直前」のプロセスの状態を調べる準備が整いました。
AUD/ETM トレースなどや、K コマンド（164 頁）を使って例外箇所の近辺を調査します。
まずは K 0 コマンドを使用して、スタックトレースを確認するのが良いでしょう。

索引

か

き

さ

し

す

そ

て

は

ひ

ふ

ま

ろ