Embeded Linux
Embeded Linux
Embeded Linux

PARTNER for Linux

Kyoto Microcomputer Co., Ltd.

PARTNER for Linux Manual

Thanks for purchasing PARTNER for Linux, which is a source debugger supporting Linux.
This product was developed by Kyoto Microcomputer Co., Ltd., which also produces and markets it, for the purpose of providing a comfortable debugging environment. We believe that you will find this product a very useful tool for a long time.

Introduction

About this manual

This document describes procedures for debugging the Linux kernel, loadable modules, and applications using PARTNER, and functions added since PARTNER Ver. 5.11.

PARTNER versions earlier than 3.5 do not support Linux-compatible debug environments. Also note that only version 5.1 or later supports debug functions for Linux kernel 2.6. If you have already installed older version software, or you want to use all functions described in this document, acquire the latest version software from the user support site、 http://www.kmckk.co.jp.

Screen shots used in this document are taken from the software for the ARM series.

About supported CPUs

PARTNER for Linux supports AM33/ARM/MIPS/SH series CPUs. This manual is usable with all supported CPUs. Notes are added to descriptions for functions and settings that depend on a specific CPU.

About online help

PARTNER for Linux provides on-screen online help to describe its functions and use. To display the online help, press the F1 key or enter the HELP command, or choose Help -> Contents and click the Help button in the displayed dialog box.

Product configurations and names

● PARTNER-Jet (high-speed JTAG ICE)

PARTNER-Jet is a high-speed JTAG-ICE, optimal in debug environments for large embedded systems such as Linux and T-Engine.

PARTNER-Jet enables debugging by resolving, with a JTAG-ICE, all software programs in the virtual and physical memory spaces that use MMU. Accordingly, real-time trace and hardware breakpoints can be used in any area of the memory space. Debug model, in which only the virtual memory space of applications is treated with an ICE, is provided, so that it is possible to fully support multi-thread and multi-process.

● PARTNER (high-performance source-level debugger software)

PARTNER for Linux is a high-performance source-level debugger, to which functions have been added as extensions to comfortably debug target systems into which Linux has been embedded. It is also the control software for **PARTNER-Jet**.

About notation used in this document

This manual adopts several notational rules.

Quotation and cross-reference

When quoting or referring to descriptions in other parts of this document, double quotes (″ ″) are used.

PARTNER configuration file names

PARTNER configuration files have such names as JETARM_1.JPX, etc. Replace ″ARM″ with ″SH″ or ″MIPS″ depending on your CPU. Numbers indicate numbers assigned to PARTNER windows that were opened with the ″MULTI command″ (Page 173)

Notation specific to individual devices

This is used to indicate which device the operation in question is being performed on. Parts which should be entered by a user are underlined, and the ″ ↓ ″ symbol indicates that a command has been entered.

```
WINPC   :Input at the command prompt of a Windows PC
LINUX8  :Command input on a Linux PC
TGT     :Console input to a target Linux
PT      :Input to a PARTNER command window
PT#     :Input to the #th PARTNER command window when multiple PARTNER windows are open
```

Notes

Notes are provided where special care is needed for operational methods or handling, or as additional explanation for things that can easily be misinterpreted.

Other comments

Used to describe useful operational methods, tips, techniques, columns, etc. that are worth remembering.

Table of Contains

Chapter 1 Tutorial 1

1.1 Tutorial overview 2

1.2 About development environments 3

1.2.1 Setting up PARTNER on the Windows PC for debugging 6

1.2.2 Loading and executing Linux from PARTNER 13

1.3 Observing behavior when the kernel is launching 14

1.4 Simple Application 15

1.5 Interactive application 21

1.6 Parallel processing using the pthreads library 24

1.7 Parallel processing using the fork system call 29

1.8 Creating a shared library 35

1.9 Executing external commands from an application 39

1.10 Creating a loadable module 44

Chapter2 Linux debugging and environment configuration 51

2.1 Overview of the configuration for Linux debugging 52

2.1.1 Configuration settings for Linux debugging 52

2.1.2 Configuration diagnosis 53

2.2 Modifying and configuring the Linux kernel source 54

2.2.1 Necessity of modifying the Linux kernel source 54

2.2.2 Applying the patch 55

2.2.3 Notes on manual modification 55

2.2.4 Notes on manual modification for MIPS series (version 2.4.x) 55

2.2.5 Kernel configuration 57

2.2.6 Effects on behavior that caused by kernel modification 60

2.3 Application debug support file 61

2.3.1 Necessity of the debug support file 61

2.3.2 Preload library method 63

2.3.3 Application direct link method 66

2.3.4 Notes on Linux compatibility 71

2.4 Launching the debug environment 73

2.4.1 Types of target initialization and launch methods 74

2.4.2 Configuring and launching PARTNER 75

2.4.3 Loading the Linux kernel 78

2.4.4 Executing the Linux kernel 81

Chapter 3 Debug procedure reference 83
3.1 Debugging the kernel 84
3.1.1 Configuration required for debugging 84
3.1.2 Debugging when initializing the Linux Kernel 85
3.1.3 Kernel debugging after system boot 86
3.2 Debugging a loadable module (method 1) 87
3.2.1 Configuration required for debuggin 88
3.2.2 Debugging Procedure 89
3.3 Debugging a loadable module (method 2) 93
3.3.1 Configuration required for debugging 93
3.3.2 Debugging Procedure 94
3.4 Debugging an application 99
3.4.1 Configuration required for debugging 99
3.4.2 Debugging Procedure 100
3.5 Debugging a multi-thread application 106
3.5.1 Configuration required for debugging 106
3.5.2 Debugging Procedure 107
3.6 Debugging a multi-process application 116
3.6.1 Configuration required for debugging 116
3.6.2 Debugging Procedure 117
3.7 Debugging a shared library 122
3.7.1 Configuration required for debugging 122
3.7.2 Debugging Procedure 122
3.7.3 Notes on shared library debugging 124
3.8 Linux OS compatible history display 125
3.8.1 Configuration requirements 125
3.8.2 Debugging Procedure 125
3.9 Attaching to a running application 133
Chapter 4 Linux compatible function reference 139
4.1 CFG file extensions 140
4.2 Launch option 146
4.3 Additional command 159
Chapter 5 Special debugging methods 181
5.1 Debugging in Application Mode 182
5.1.1 Configuration required for debugging 182
5.1.2 Procedure for debugging an application in Application Mode 183
5.1.3 Procedure for multi-context debugging in Application Mode 185

5.2 Manually debugging a loadable module 192
5.2.1 Configuration required for debugging 192
5.2.2 Debugging Procedure 192
5.3 Manually debugging an application 200
5.3.1 Configuration required for debugging 200
5.3.2 Debugging Procedure 200
5.4 Manual multi-process/multi-thread debugging 204
5.4.1 Configuration required for debugging 204
5.4.2 Debugging Procedure 205

Chapter 6 Event Tracker 211
6.1 Requirements for the Event Tracker 212
6.2 Linux kernel modification for the Event Tracker 213
6.3 Launching the Event Tracker in the debugger 214
6.4 Operation of the Event Tracker window 215
6.5 Event tracker commands 220

Appendix 223
Appendix A Trouble shooting 224
A-1 Kernel debugging 224
A-2 Loadable module debugging 225
A-3 Application debugging 226
A-4 Real-time trace 228
Appendix B Building a Linux PC for development 229
B-1 Setting up a cross environment on a Linux PC 230
B-2 Setting up an NFS server using the Linux PC for development 234
B-3 Setting up terminal software 235
B-4 Starting up Linux on the target 236
B-5 Setting up a Samba Server 238
Appendix C How to insert the support file into glibc 240
C-1 Modifying the glibc library of the target system 241
C-2 Registering glibc with PARTNER 242
C-3 Debugging method 242
Appendix D Debugging a dynamic linker/loader (ld.so) 243
Appendix E Technology description: Linux architecture and PARTNER 247
E-1 Linux development environment and problems 247
E-2 Debugging methods provided by PARTNER 250
E-3 Kernel mode debugging details 253
E-4 Application mode debugging details 254

Appendix F Selecting the operation mode of PARTNER 255
F-1 Characteristics of each mode and use of purpose 255
F-2 Startup method and necessary conditions 256
F-3 Guidance for operation mode selection 257
F-4 Behavior and transition of Linux debug operation modes 258
Appendix G Supplement for kernel tree compiling 259
G-1 Suppressing compile optimization 259
Appendix H Tips for investigating the causes of segmentation faults 261
H-1 Method of thinking 261
H-2 Tracing SIG_SEGV 261

Index 263

Chapter 1 Tutorial

This chapter describes with examples procedures for using Linux on evaluation/ development boards and debugging using PARTNER.

1.1 Tutorial overview

The quickest and easiest way of learning how to develop and debug the Linux kernel and applications is actually doing it rather than studying it. This chapter describes how to perform debugging in an embedded Linux environment using the KZM-ARM11-01 board and the PARTNER-Jet debugger.
The PARTNER debugger provides many functions and there are a number of debugging techniques that can be used with it. Instead of describing all of them, this chapter describes configurations selected to provide the most comfortable development environment.
For information on detailed configurations and functional descriptions, refer to the subsequent chapters.

The KZM-ARM11-01 board used in examples is a high-end board for an embedded device, which features the ARM11 core and supports various devices. This can boot the OS from the hard drive, SD card, USB storage, onboard NAND Flash, etc., but in this chapter, it uses a file system on a network that uses NFS. Interfaces required for debugging on the target board are JTAG and Ethernet controllers only, so similar configurations can be achieved in many embedded Linux development environments.

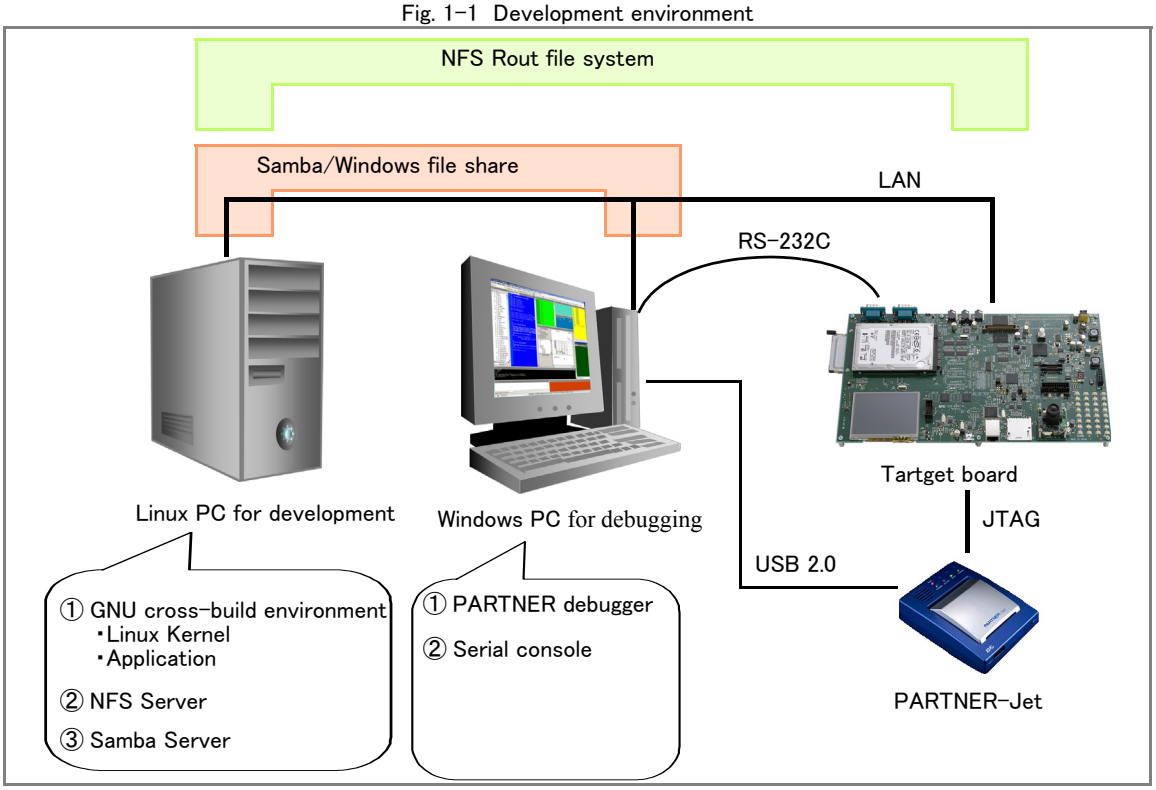
This chapter provides tutorials to describe how to perform debugging in an embedded Linux environment.

- "About development environments (Page 3)"
- "Observing behavior when the kernel is launching (Page 14)"
- "Simple Application (Page 15)"
- "Interactive application (Page 21)"
- "Parallel processing using the pthreads library (Page 24)"
- "Parallel processing using the fork system call (Page 29)"
- "Creating a shared library (Page 35)"
- "Executing external commands from an application (Page 39)"
- "Creating a loadable module (Page 44)"

1.2 About development environments

The KZM-ARM11-01 board comes with a Linux environment CD-ROM for ARM CPUs; thus the software required to launch a Linux environment is already there without having to port the Linux kernel.
However, unlike a Linux environment on a PC, you cannot simply prepare the target board and software CD-ROM to install and execute it. You have to build a cross development environment.
The hardware configuration of the development environment to be built is as shown in Fig. 1-1

Fig. 1-1 Development environment



Use a Linux PC as the development environment (compile environment) for the Linux kernel and applications, and a Windows PC as the operating environment for the PARTNER debugger. If you use a virtual machine such as VMWare on the Windows PC, you can physically remove the Linux PC used for development from the environment. However, two PCs will be required in either case.
The target board and two PCs are connected via LAN. To build a comfortable development environment, it is important to publish to the network a directory that is shared by both NFS and Samba.

1. The target board uses the directory on the Linux PC as the root file system in NFS.
2. In Samba, the Windows PC refers to the same location as the target board refers to.

These two points enable PARTNER to automatically correlate debug information with the source code, so that comfortable debugging can be performed. Because the storage capacity of an embedded system is normally not that large, it is not easy to install a large-sized program with debug information into the target board. This problem can be solved using the NFS root file system.

Needless to say, it is not mandatory to utilize the NFS root file system or file sharing by Samba.

・The target is launched from the root file system (without debug information) on the onboard storage.

・A copy of the root file system tree (with debug information) is placed on the hard drive of the Windows PC.

If these two conditions are met, a LAN connection is not required.

The reason why NFS and Samba are used in conjunction with a LAN connection is to facilitate a procedure that will be repeated (modify a program -> build -> place on the target board a file that has no debug information -> place on the Windows PC for debugging a file that has debug information -> debug).

The Linux OS that comes with the KZM-ARM11-01 board adopts a serial console like other distributed embedded device packages. In Figure 1-1, the RS-232C is connected to the Windows PC, but it can also be connected to the Linux PC.

For details on how to set up a Linux PC for development, refer to "Building a Linux PC for development (Page 229) "Appendix B: Building a Linux PC for development (page 219)".

Table 1-1 shows the file paths used in the development environment.

Table 1-1 File locations on the Linux PC

Software	PATH
Cross tool chain	/opt/kmc/kzm-arm11/staging_dir/ Command: /opt/kmc/kzm-arm11/staging_dir/bin/ Include : /opt/kmc/kzm-arm11/staging_dir/include/ Library : /opt/kmc/kzm-arm11/staging_dir/lib/
Root file system for target	/opt/kmc/kzm-arm11/root/
Root user home on target	/opt/kmc/kzm-arm11/root/root/ Path name on target : /root/
Linux kernel source tree	/opt/kmc/kzm-arm11/build_src/linux/
Linux kernel	/opt/kmc/kzm-arm11/build_src/linux/vmlinux /opt/kmc/kzm-arm11/build_src/linux/arch/arm/boot/zImage
Debug support file (PRELOAD Library)	/opt/kmc/kzm-arm11/root/usr/lib/libkmcsup.so.2.0.0 Path name on target : /usr/lib/libkmcsup.so.2.0.0

This tutorial uses the following setting, which is highly recommended for providing a comfortable development environment.

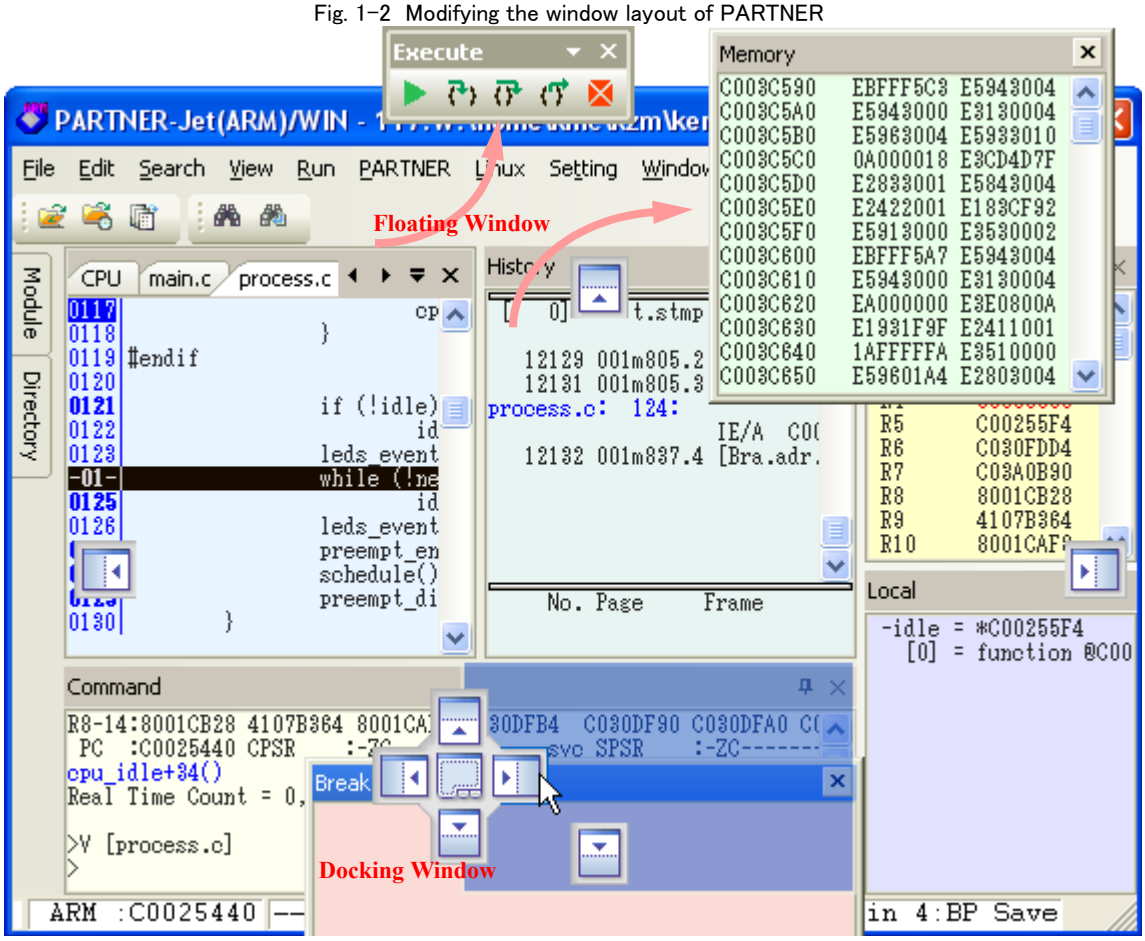
・Use "K2_LINUX_ADD_V26" as the launch mode of PARTNER (refer to "-OS option (Page 147)")

・Use the debug support file as a preload library (refer to "2.3 Application debug support file (page 57)" and ""Application debug support file (Page 61)")

In the subsequent pages, screen shots of PARTNER are frequently used for descriptions. Screen configurations are quite different from each other, because the window layout within PARTNER can be freely arranged as shown in Fig. 1-2 .

Used screens are arranged so that each relevant topic can be intelligibly explained.

Fig. 1-2 Modifying the window layout of PARTNER



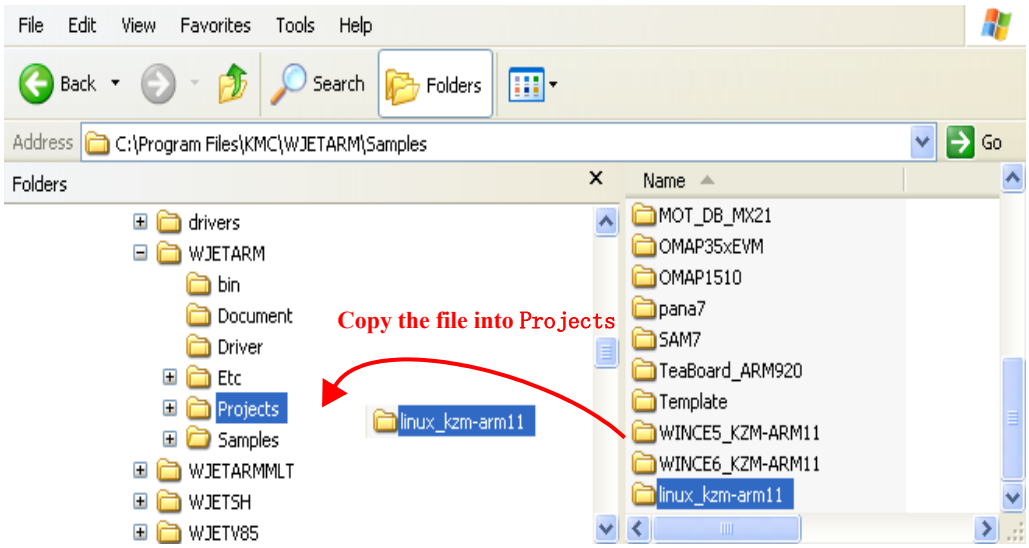
1.2.1 Setting up PARTNER on the Windows PC for debugging

For information on how to install PARTNER software and the PARTNER-Jet driver, refer to their manuals. This section describes how to set up the KZM-ARM11-01 board, assuming that the PARTNER for ARM has already been connected and installed.

Copy the configuration file for PARTNER for the KZM-ARM11-01 board that is located in the Sample directory of PARTNER.

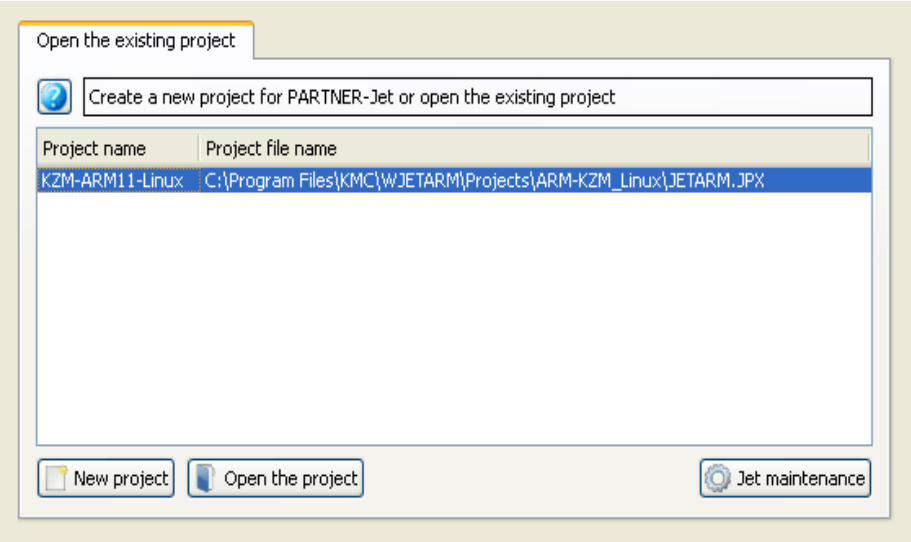
```
WINPC>c: ↓
WINPC>cd "Program Files¥KMC¥WJETARM¥Samples" ↓
WINPC>mkdir ..¥Projects¥linux_kzm-arm11 ↓
WINPC>copy linux_kzm-arm11 ..¥Projects¥linux_kzm-arm11 ↓
linux_kzm-arm11¥init.mcr
linux_kzm-arm11¥jetarm.cfg
linux_kzm-arm11¥jetarm.jpx
        3 files have been copied.
```

Fig. 1-3 Copying the sample project



You see a highly organized configuration for the board when you opened the copied project in PARTNER.

Fig. 1-4 Opening the sample project for PARTNER



The setting for the CFG file depends on the target board, and it is not necessary to modify the sample when the KZM-ARM11-01 board is used.
This section shows how to modify the "launch option of PARTNER" that varies depending on the built development environment.
(For details on this setting, refer to "Launch option (Page 146)" and "Configuring and launching PARTNER (Page 75)".

● Debug information path conversion (-XGX option)

According to the setting described in "Setting up a Samba Server (Page 238)", specify "-XGX/opt/kmc,z:¥kmc¥".

● Application debug information search setting (-RootDir option)

Specify the path by which PARTNER refers to the NFS route to the target (-RootDir z:¥kmc¥kzm-arm11¥root)

● OS debug mode specification (-OS option)

For the PARTNER mode, we recommend "Kernel ADD Mode", which can support both kernel and applications and is easy to use (everything that can be done in Kernel (NON ADD) Mode can also be done in Kernel ADD Mode). In this tutorial, "K2_LINUX_ADD_V26" is specified, as the Linux 2.6 kernel is used.

● Kernel option (-!! option)

Specify default_command_line="" and an empty argument, and write the kernel launch parameters in the init.mcr file.

Fig. 1-5 shows summarized relationships between the shared file setting and PARTNER setting.

Fig. 1-5 Correlations in the file sharing system

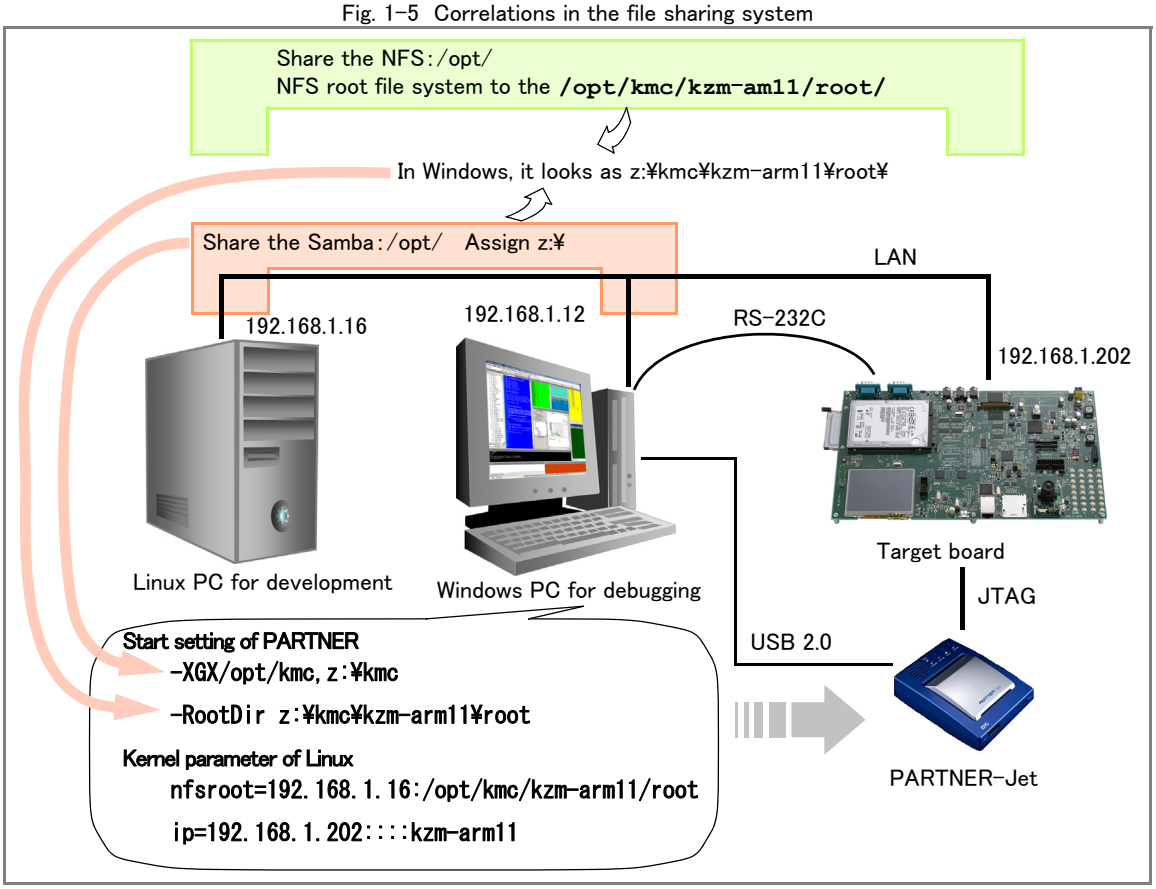
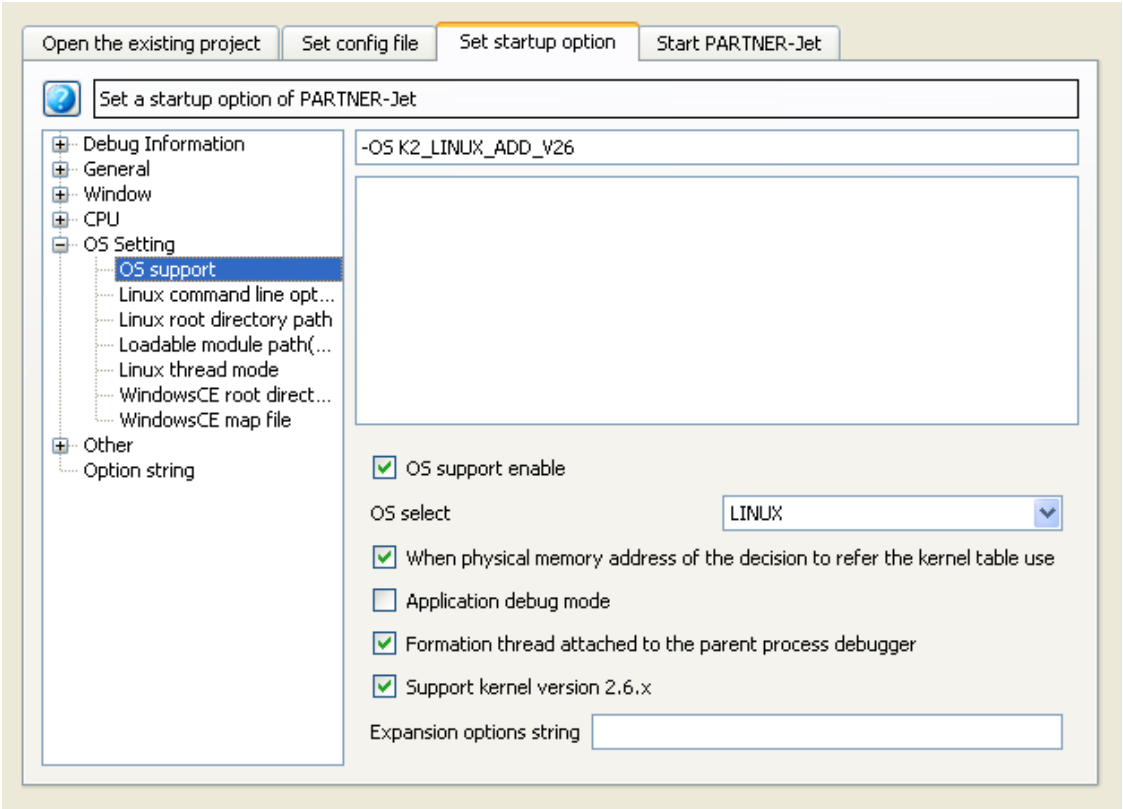
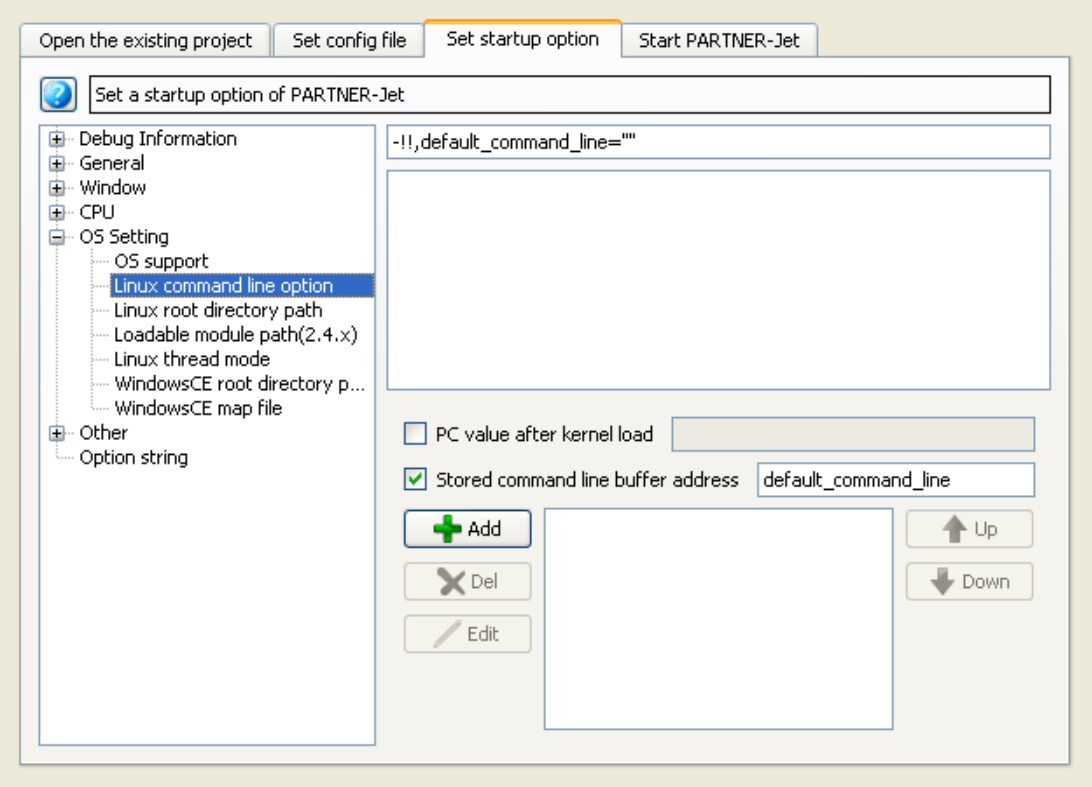
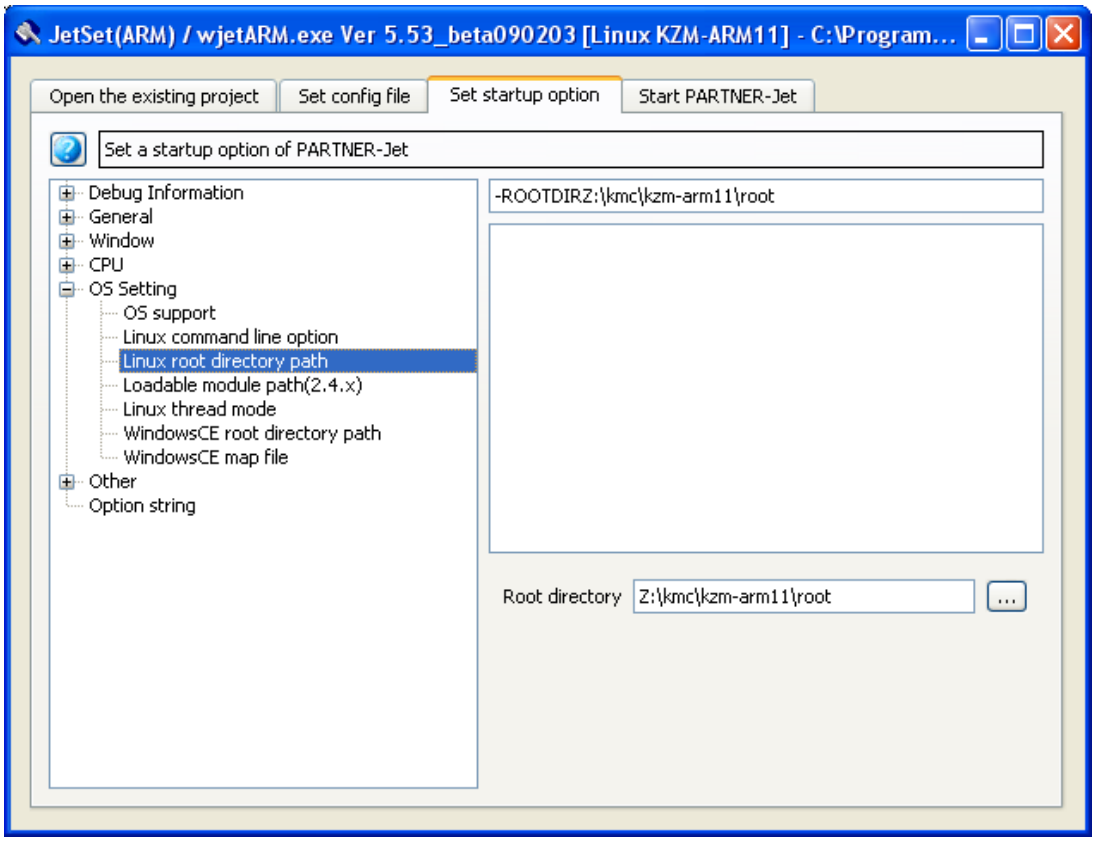
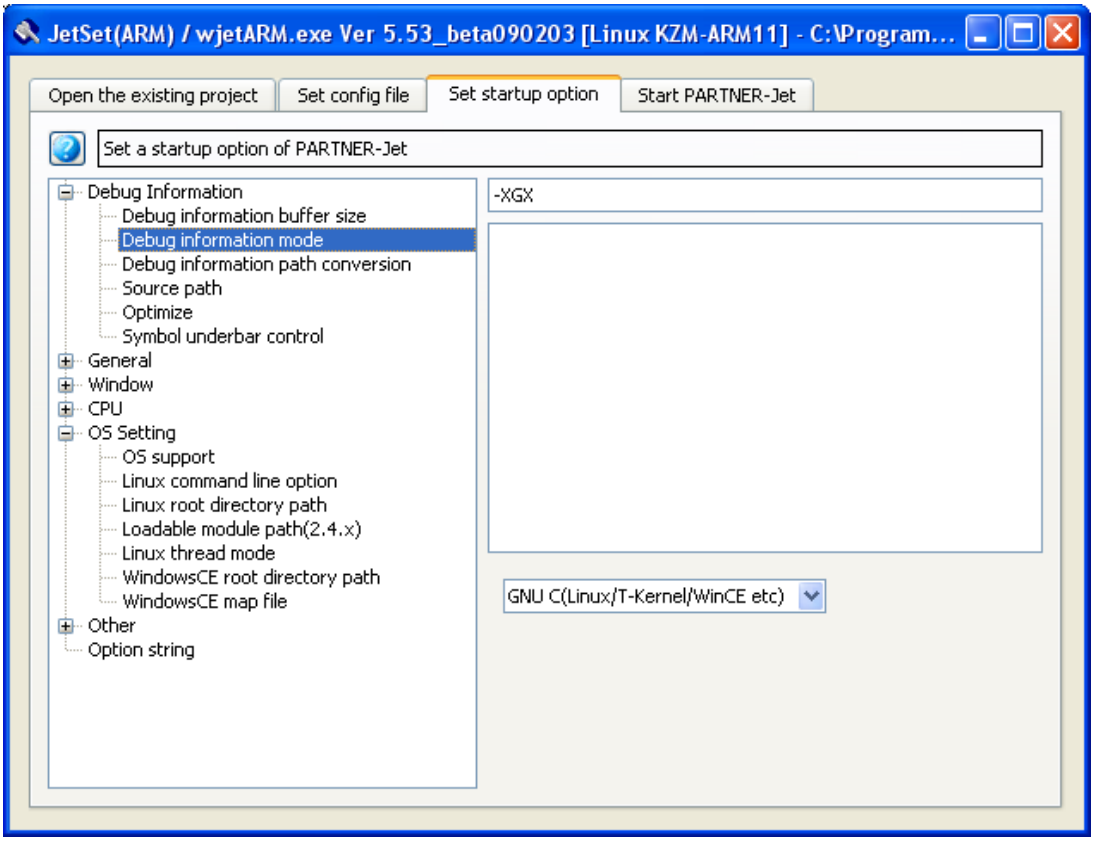


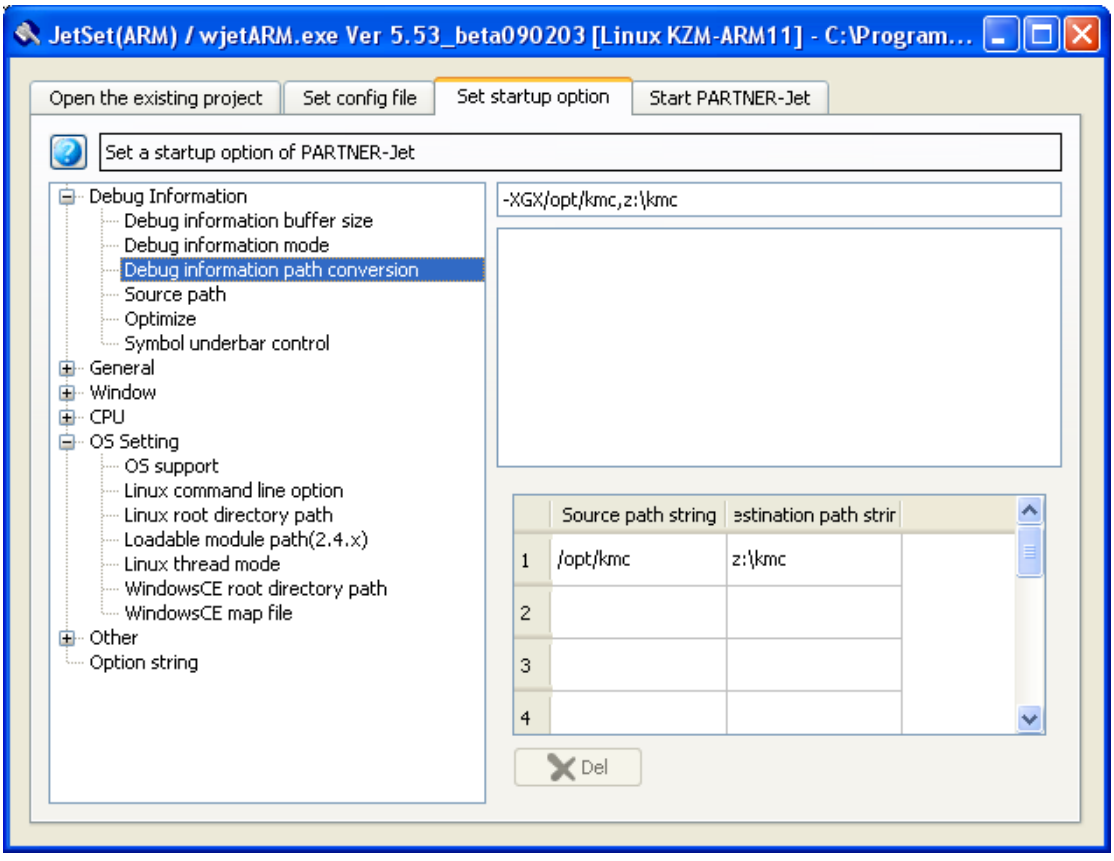
Fig. 1-6 Launching PARTNER Set startup option




Open the existing project
Set config file
Set startup option
Start PARTNER-Jet
Set a startup option of PARTNER-Jet
Debug Information
General
Window
CPU
OS Setting
OS support
Linux command line option
Linux root directory path
Loadable module path(2.4.x)
Linux thread mode
WindowsCE root directory p...
WindowsCE map file
Other
Option string
-!!,default_command_line=""
PC value after kernel load
Stored command line buffer address
default_command_line
Add
Del
Edit
Up
Down


JetSet(ARM) / wjetARM.exe Ver 5.53_beta090203 [Linux KZM-ARM11] - C:\Program...
Open the existing project
Set config file
Set startup option
Start PARTNER-Jet
Set a startup option of PARTNER-Jet
Debug Information
General
Window
CPU
OS Setting
OS support
Linux command line option
Linux root directory path
Loadable module path(2.4.x)
Linux thread mode
WindowsCE root directory path
WindowsCE map file
Other
Option string
-ROOTDIRZ:\kmc\kzm-arm11\root
Root directory
Z:\kmc\kzm-arm11\root
...

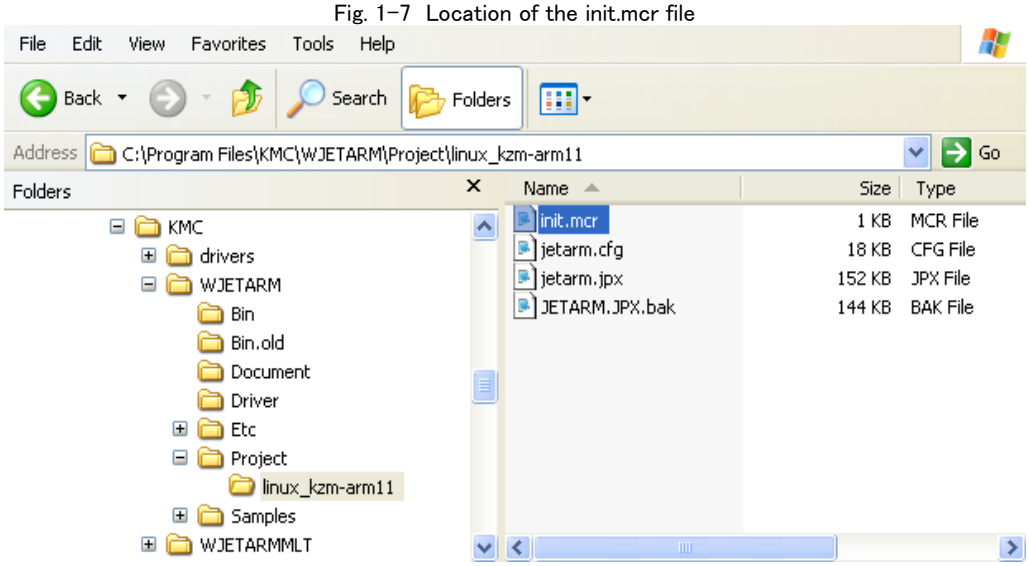

JetSet(ARM) / wjetARM.exe Ver 5.53_beta090203 [Linux KZM-ARM11] - C:\Program...
Open the existing project
Set config file
Set startup option
Start PARTNER-Jet
Set a startup option of PARTNER-Jet
Debug Information
Debug information buffer size
Debug information mode
Debug information path conversion
Source path
Optimize
Symbol underbar control
General
Window
CPU
OS Setting
OS support
Linux command line option
Linux root directory path
Loadable module path(2.4.x)
Linux thread mode
WindowsCE root directory path
WindowsCE map file
Other
Option string
-XGX
GNU C(Linux/T-Kernel/WinCE etc)


JetSet(ARM) / wjetARM.exe Ver 5.53_beta090203 [Linux KZM-ARM11] - C:\Program...
Open the existing project
Set config file
Set startup option
Start PARTNER-Jet
Set a startup option of PARTNER-Jet
Debug Information
Debug information buffer size
Debug information mode
Debug information path conversion
Source path
Optimize
Symbol underbar control
General
Window
CPU
OS Setting
OS support
Linux command line option
Linux root directory path
Loadable module path(2.4.x)
Linux thread mode
WindowsCE root directory path
WindowsCE map file
Other
Option string
-XGX/opt/kmc,z:\kmc
Source path string
estination path strir
1
/opt/kmc
z:\kmc
2
3
4
Del

Then, write the init.mcr file.

Create the init.mcr file in the project folder. The default name is "init.mcr". When the "MULTI command (Page 173)" is used in PARTNER, file names such as "init1.mcr", "init2.mcr", "init.mcr", etc. are used.

Fig. 1-7 Location of the init.mcr file



It is useful to include the loading setting of the Linux kernel as a macro in the init.mcr file.

Fig. 1-8 Description of the init.mcr file

```
{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
}
```

The name of NFS root server and path name

IP address of target board

VA → PA translate offset (ARM appropriate)

Set the entry address of Linux Kernel to the program counter

Target ID (ARM appropriate)

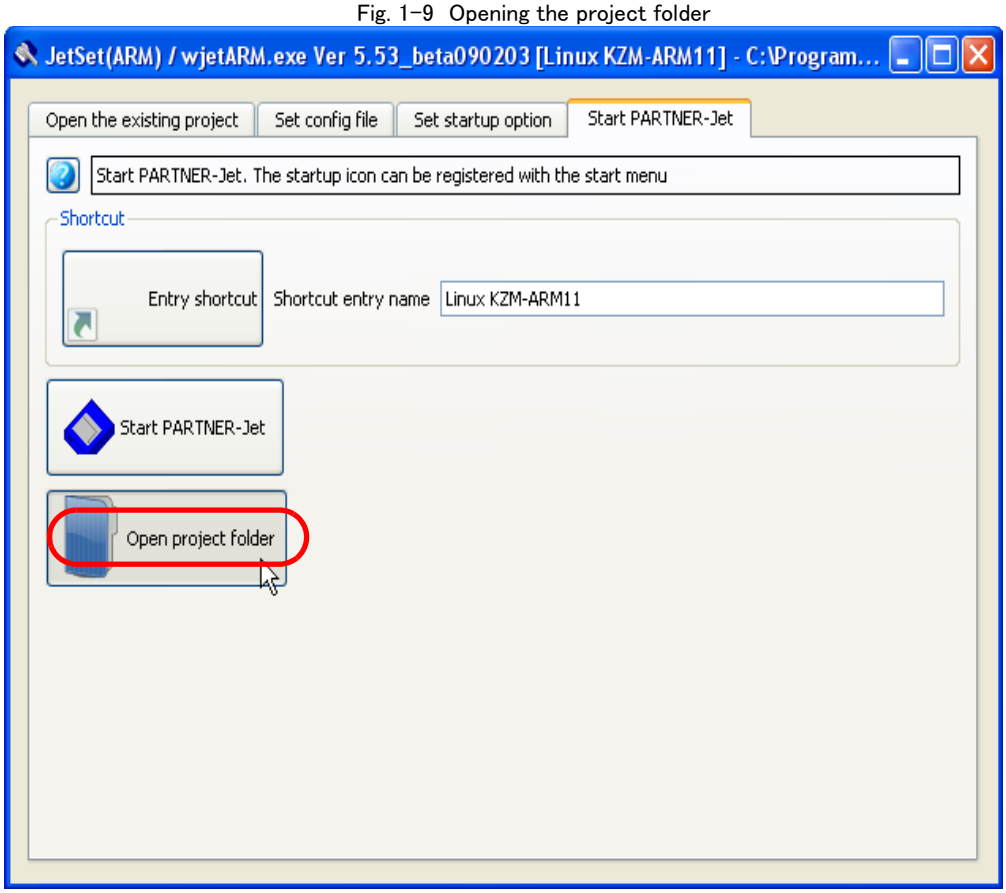
Support file setting

Executable hardware breakpoint in where accessible location with VA

This is the macro that makes application debugging easier. (described

The launch screen of JETSET has a useful button for opening the project folder.

Fig. 1-9 Opening the project folder

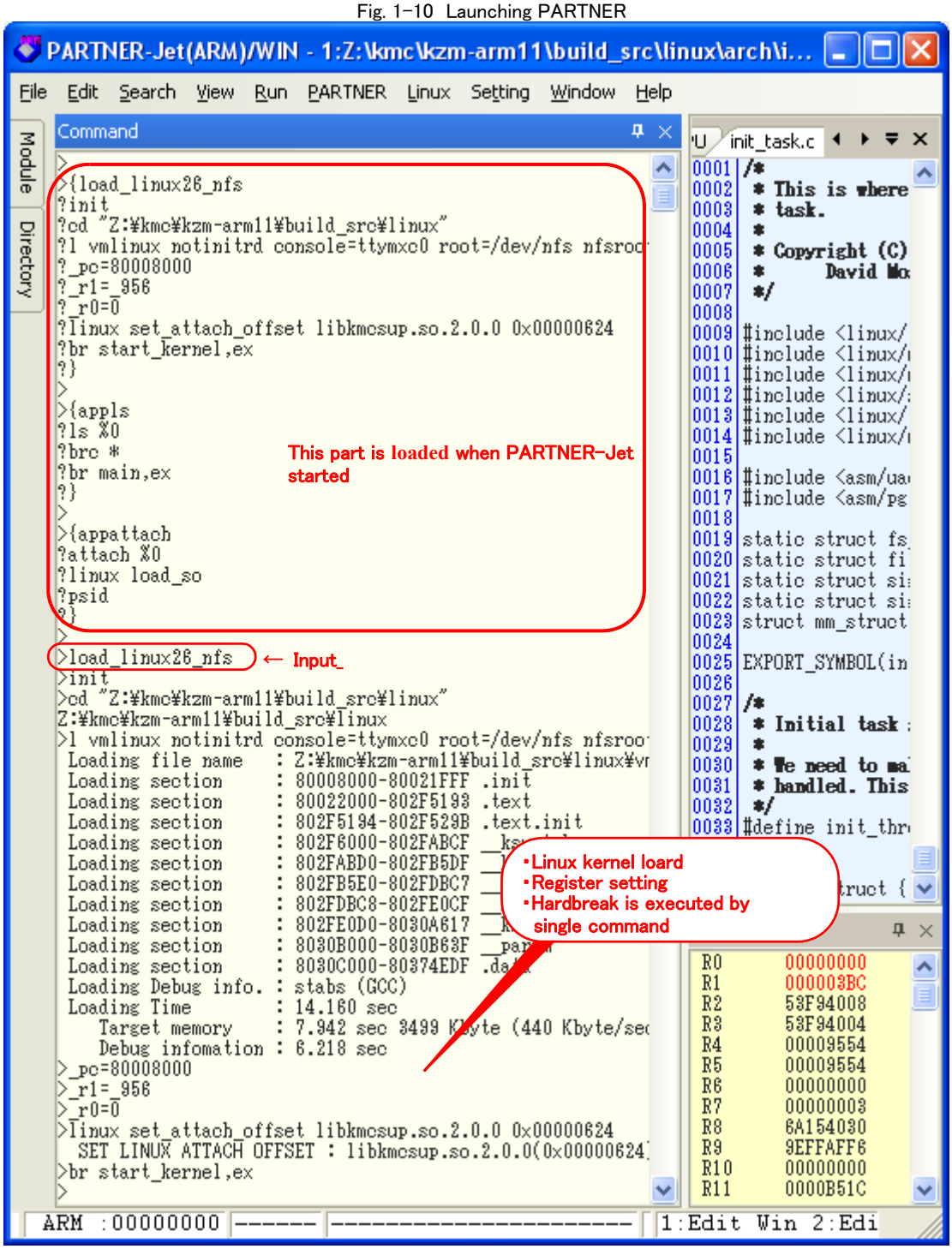


1.2.2 Loading and executing Linux from PARTNER

The macro setting stored in the init.mcr file is loaded when a PARTNER window is opened. Just typing the registered macro name will load and launch Linux using the NFS root file system.

```
PT>load linux26_nfs ↓
PT>g ↓
```

Fig. 1-10 Launching PARTNER

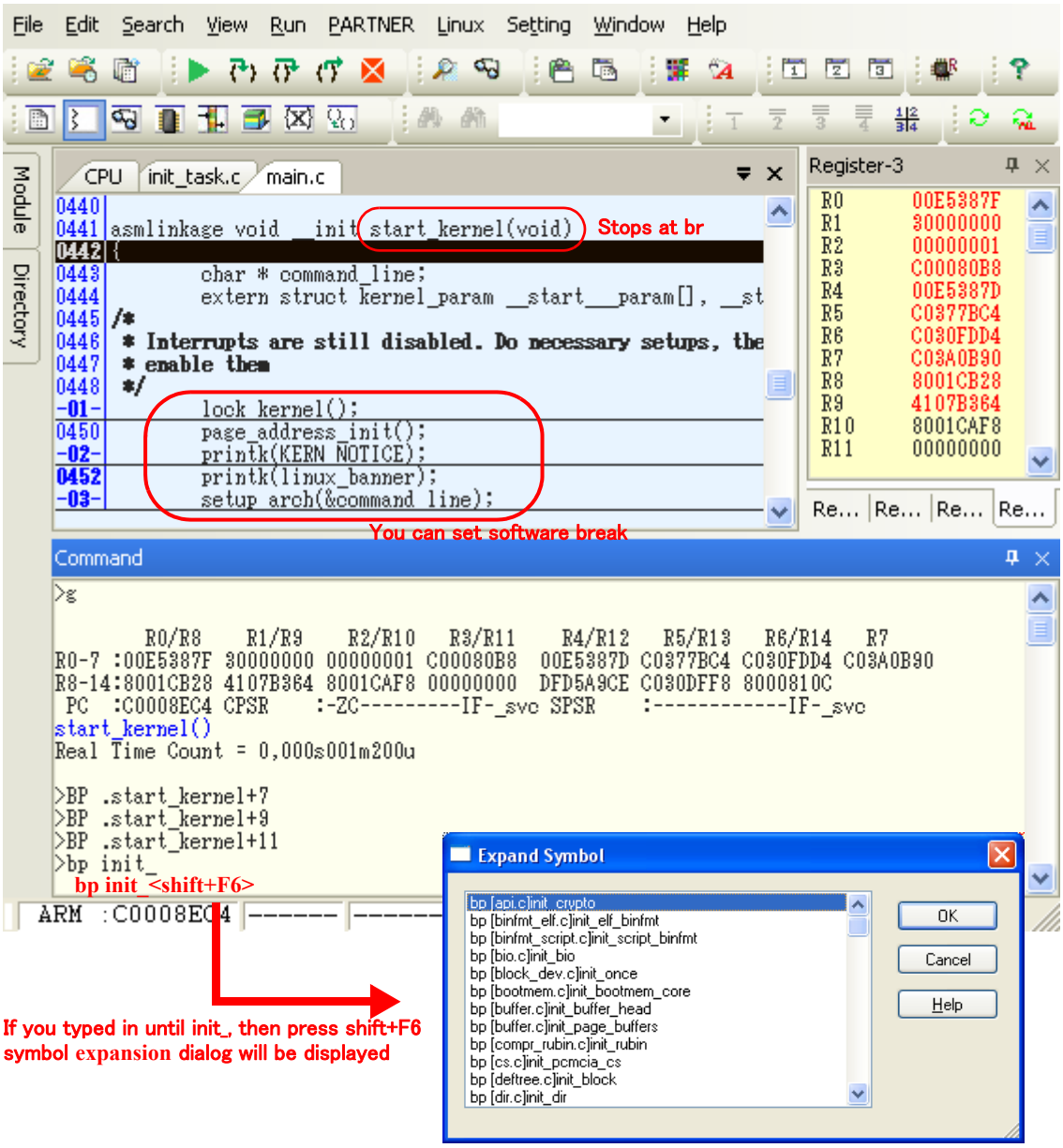


1.3 Observing behavior when the kernel is launching

When executing the Linux kernel following the procedure described in "Loading and executing Linux from PARTNER (Page 13)", the execution stops at the start_kernel symbol.
On ARM CPUs, software breakpoints cannot be used until the MMU is enabled and access is available. However, all debug functions become available after the execution has stopped at the start_kernel symbol.

Because the symbol names (function names) of initialization processes of modules that reside in the kernel usually begin with "init", using the symbol extension function of PARTNER facilitates setting breakpoints.

Fig. 1-11 Stops at start_kernel



1.4 Simple Application

In this section, we create a popular sample program "HelloWorld", which is written in C.

Fig. 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;
}
```

This source code is unlikely to pose you any problems. Let's build it with debug options enabled.

```
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
```

If you have logged onto the target's root, you should be able to see "/root/hello".

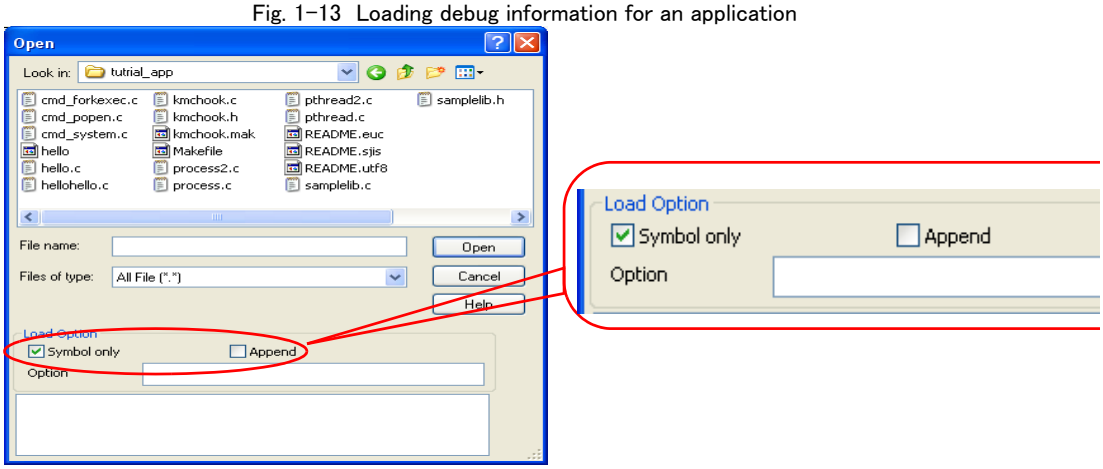
```
TGT> login: root ↓
TGT>pwd ↓
/root
TGT>ls ↓
hello  hello.c
```

Stop the target by pressing the [ESC] key on PARTNER, and set an executable breakpoint for the main function after loading the debug information. When you do so, it is useful to open another window using the "MULTI command (Page 173)".

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

The program code of a Linux application should be loaded and launched on the target. When loading the debug information of a program from the GUI menu, ensure that you only load the debug information by checking Symbol Only.
Also note that to set a breakpoint for the main symbol of an application that has not been executed yet, use br instead of bp. Almost every process has the main symbol. Using bp does not generate an error, but it creates a situation where breaks do not work as designed.

Fig. 1-13 Loading debug information for an application

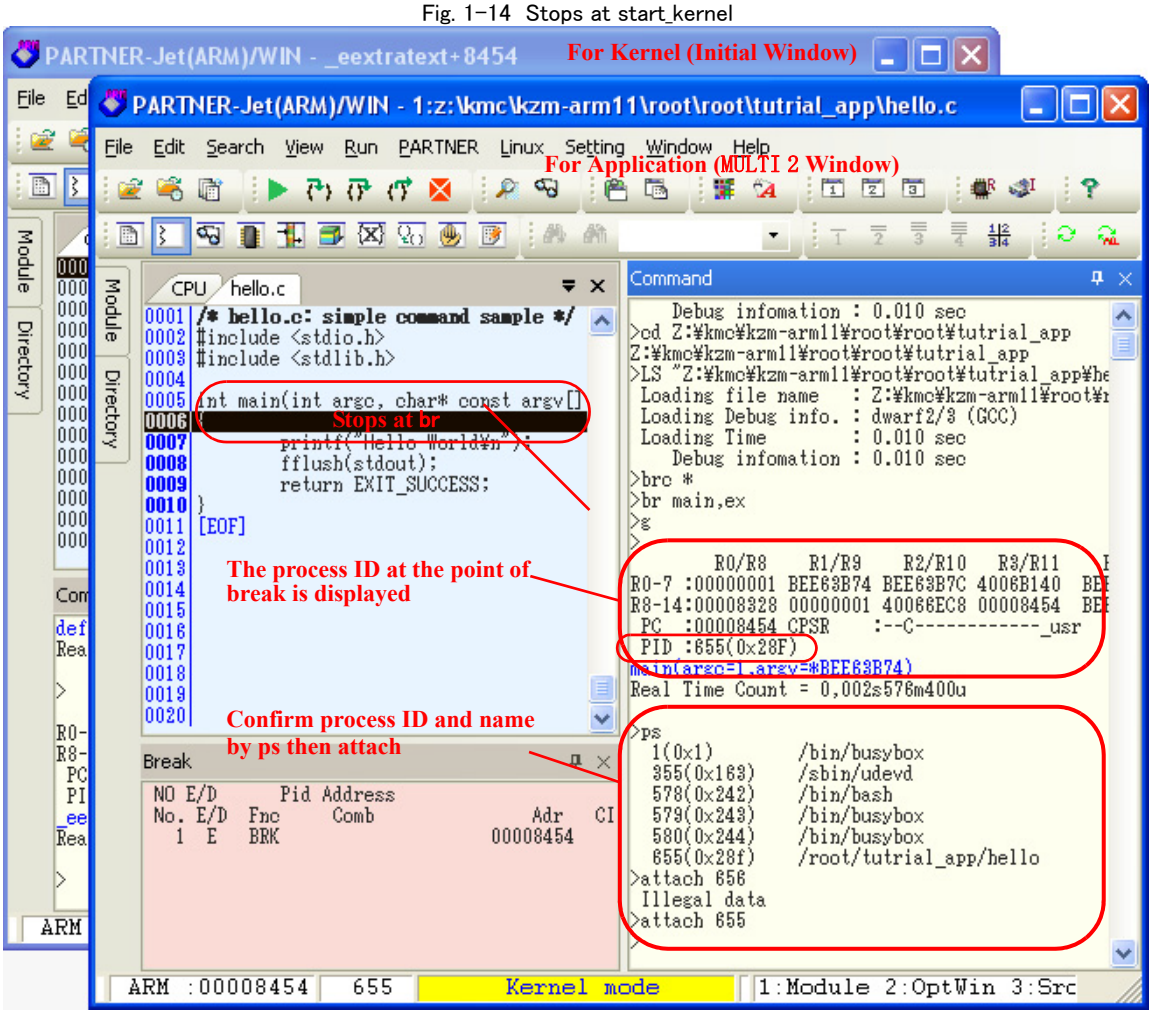


PARTNER remembers in the command history the load file names and location of files that have been loaded, so the subsequent operations can be simplified.

The program stops at the main function when it is executed on the target, so attach to the process using the "ATTACH command (Page 162)".

```
PT2>ps ↓
(You will see from the display result that the process ID is 740.)
PT2>attach 740 ↓

PT2>attach hello ↓
(If only one process of this program is running, the name can also be specified.)
```

Fig. 1-14 Stops at start_kernel



In this state, you can perform normal debugging, such as tracing in the process and setting software breakpoints. But those things can also be done with normal user mode debuggers. In this section, we perform more advanced operation utilizing the features of PARTNER.

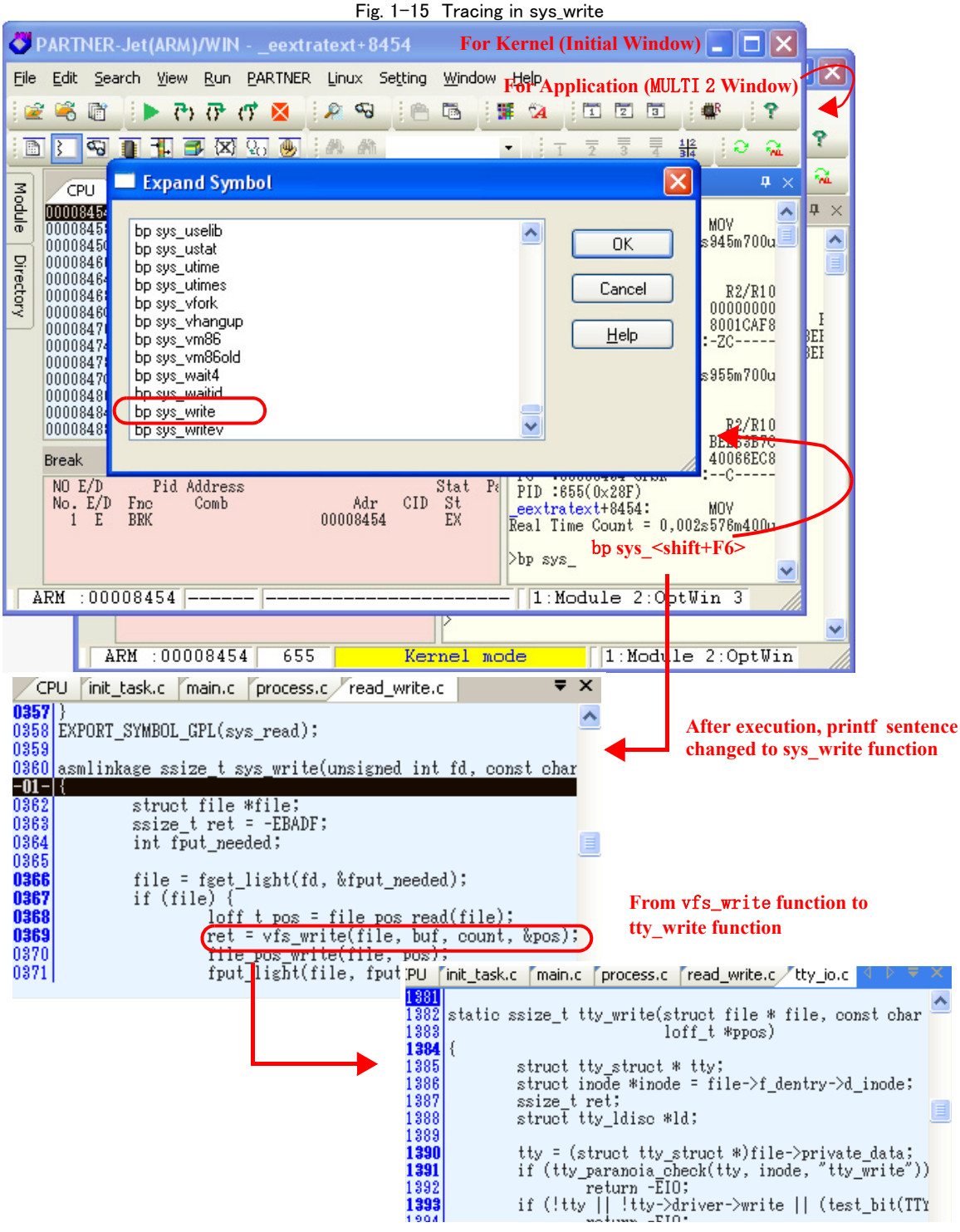
The printf sentence, which is the first sentence in the main function, displays the string "Hello World". Because the standard output is connected to the virtual terminal (tty) in the Linux kernel, the string is actually displayed on the serial console, into which the user has logged. The printf function should be set to write to the tty device file. You can easily trace it using PARTNER.

After setting up some breakpoints in the PARTNER window for the application, set up breakpoints in the PARTNER window, into which the kernel debug information has been loaded.

Although the system call used by the library function printf() to perform file I/O for the last time is write(), set it as "bp sys_write", since the symbol names of Linux system calls begin with "sys_".

```
(PARTNER window for the application)
PT2>bp .main+1 ↓   (printf sentence)
PT2>bp .main+2 ↓   (fflush sentence)

(PARTNER window for the kernel)
PT>bp sys_write ↓
```

Fig. 1-15 Tracing in sys_write



After the printf sentence is executed in the application window, the kernel window becomes active and the execution stops at sys_write. By stepping within the kernel, you see that a function named tty_write has been called. If you execute the G command under this condition, the application window becomes active and the execution stops at the fflush function. In other words, you can perform debugging by seamlessly switching between application processes and the inner kernel.

We recommend reading debug information for shared libraries using the load_so "LINUX command (Page 164)", after attachment. If you register this procedure as a macro as shown in "Description of the init.mcr file (Page 11)", the procedure can be simplified as described below. You also can modify the behavior of ATTACH command itself by making the ATTACH_AUTO_SO setting using the LINUX command Format 5, to read debug information for shared libraries when attachment is performed. However, macro registration procedure can be useful if you would rather perform operations without reading debug information for shared libraries due to a large number of shared libraries being used and the attachment process being slow as a consequence.

```
PT2>appls hello ↓
(Loading debug information and setting breakpoints for the main function)
PT2>g ↓
TGT>./hello ↓
PT2>ps ↓
(From the display result, you will see that a program with the process ID 740 is hello.)
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
>
(After attach and linux load_so are executed, display the current status with psid.)
```

This macro is used in subsequent descriptions in this tutorial.

1.5 Interactive application

The program described in "Simple Application (Page 15)" promptly performs processing and the program exits once the processing is done. (This type of application is sometimes called a command type or a batch processing type). There are programs that run much longer. To name a few, "server programs" that handle connections from network clients, and "GUI applications" that use the X-Window system and suchlike, are included in these programs. The characteristics of these programs are that they are interactive. Details may differ depending on whether the other party is a program or a human. It is safe to say that these programs wait for a period of time until some event occurs or a request is made, rather than always performing 100% processing.

These types of programs tend to be executed immediately once the system is launched and to remain in memory, and the right timing for debugging these kinds of programs tends to be when any operation or event occurs rather than when executing the application.

These kinds of programs can be supported using the attach debug function for running programs that PARTNER provides.

As an example, hellohello.c, which is a modified interactive version of "hello.c (Page 15)", is used to describe interactive programs.

Fig. 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;
}
```

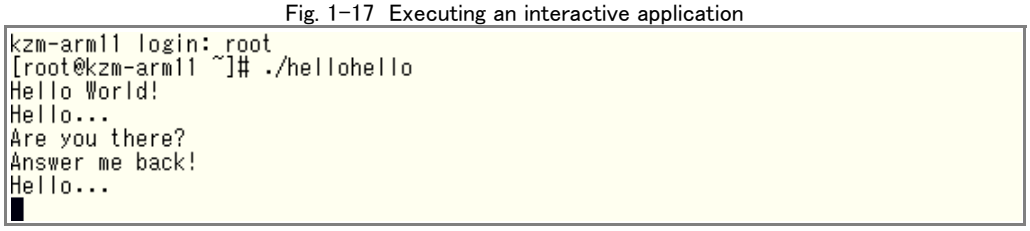
This program keeps displaying a message using the printf sentence every 5 seconds. When input is made on the terminal by a user, it displays the input string, and exits.

Compile it with debug options enabled.

```
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
```

It should be executed on the target.

Fig. 1-17 Executing an interactive application



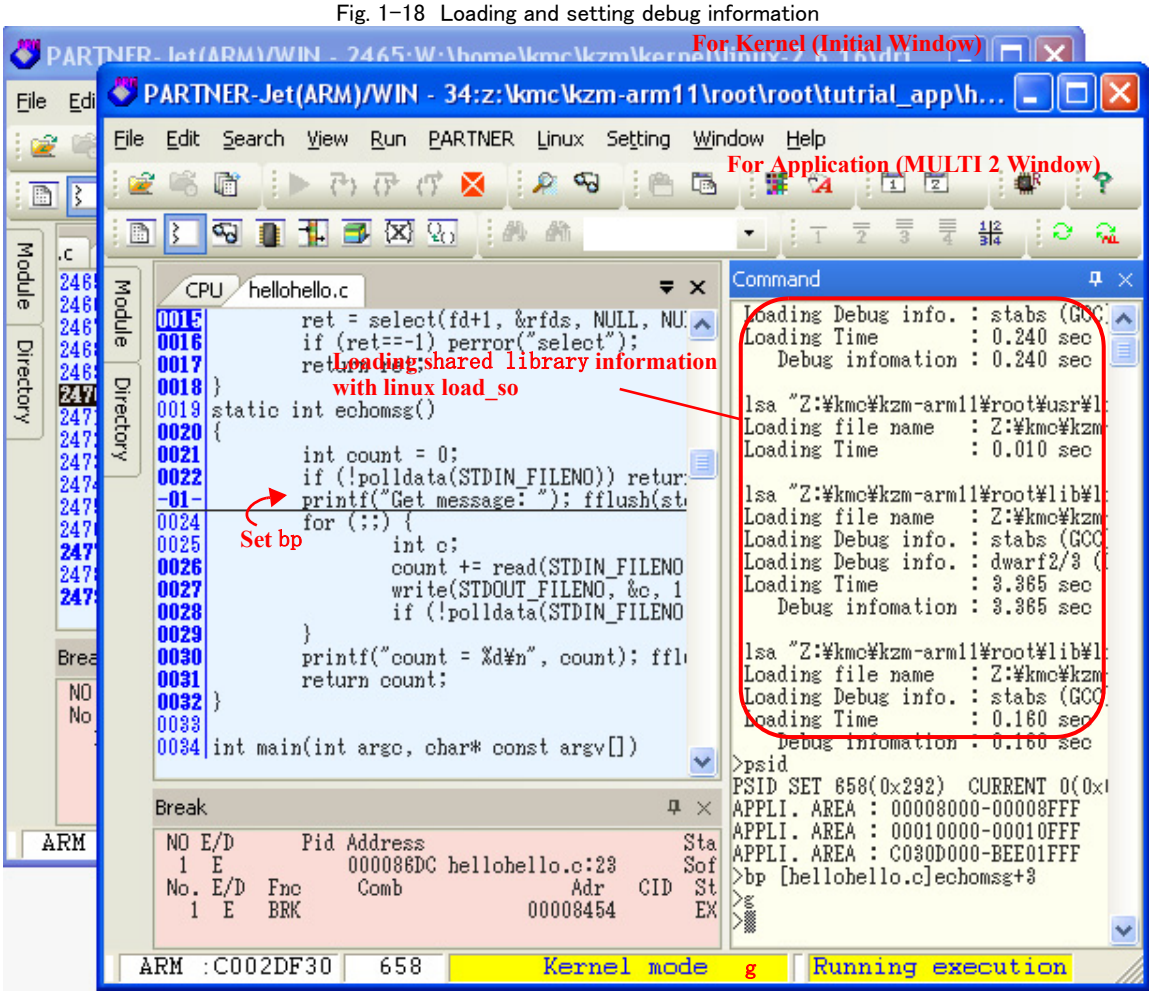
Now, a situation where a program is running has been created.
Press the ESC key in the PARTNER window to stop the target, and then load the debug information.

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

A breakpoint is set at the main function, but it does not stop since the program has already been executed. The process is, also, still running, so attach to the process using the "ATTACH command (Page 162)" (that is to say, the procedure is almost the same as described in "Simple Application (Page 15)".

```
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
(You see from the display result that the process ID is 740.)
PT2>appattach 740 ↓
(The information about shared libraries is loaded with the linux load_so command included
in the macro.)
PT2>BP [HELLOHELLO.C]echomsg+3 ↓
(Sets a breakpoint at a point, which is executed when input is received from the terminal.)
PT2>g ↓
(Execution continues)
```

Fig. 1-18 Loading and setting debug information



Even in this status, the program does not stop and continues to periodically send output to the console.

The program stops at the breakpoint when any letter is input on the terminal.

Fig. 1-19 Stops at the breakpoin

```
0017        return ret;
0018 }
0019 static int echomsg()
0020 {
0021        int count = 0;
0022        if (!polldata(STDIN_FILENO)) return 0;
-01-        printf("Get message: "); fflush(stdout);
0024        for (;;) {
0025                int c;
0026                count += read(STDIN_FILENO, &c, 1)
0027                write(STDOUT_FILENO, &c, 1);
0028                if (!polldata(STDIN_FILENO)) break
0029        }
0030        printf("count = %d¥n", count); fflush(stdc
0031        return count;
0032 }
```

1.6 Parallel processing using the pthreads library

A parallel programming method with multi-threads has often been used, because required specifications for recent programs are getting more complex and highly-functional, and its behavior is close to the RTOS task model, which is frequently used for embedded devices.
However, there has never been an environment in which Linux multi-thread programs can comfortably be debugged. The main reasons for this are as follows: 1) the multi-thread/multi-process functions for the PTRACE system call, which is used by user-mode debuggers that run on the target (such as GDB), are not fully supported. 2) latency cannot be ignored when the number of threads is increased to use the signal distribution function to stop processes (to perform breaks). JTAG debuggers can stop the target CPU, and the debugger does not work as a process on the target. So the above problems do not occur with them.

In this section, multiple-thread contexts are debugged on PARTNER using a sample that uses the pthreads library.

Fig. 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;
}
```

The sample code pthread.c is a very basic way of using the pthreads library.
It creates two threads from the main function, and has each of them process the func1 and func2 functions respectively. That is to say, there are three contexts that are simultaneously executed within the program.
The func1 and func2 functions output strings to the terminal at regular intervals. Both functions perform nearly the same thing. Although it is possible to have the program execute one function in two individual threads, two functions are used, as that makes it easier to understand.
After creating the two threads, the main function waits for the termination of these threads. Usually, normal RTOS tasks cannot wait for the exiting of other tasks (and that reduces the processing load). However, it is actually possible to wait for the termination of other threads using the pthread_join function, because "joinable threads", which make it possible for a task to wait for the termination of other threads as with the case of processes, are used as a default in the pthreads library implementation in Linux (Unix).

Fig. 1-21 shows the display example of this execution

Fig. 1-21 Executing a pthread application

```
[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 ~]#
```

Because it takes a while for this program to terminate, you can attach to the process after this program was executed. But debugging is performed from the beginning of the main function in this example.

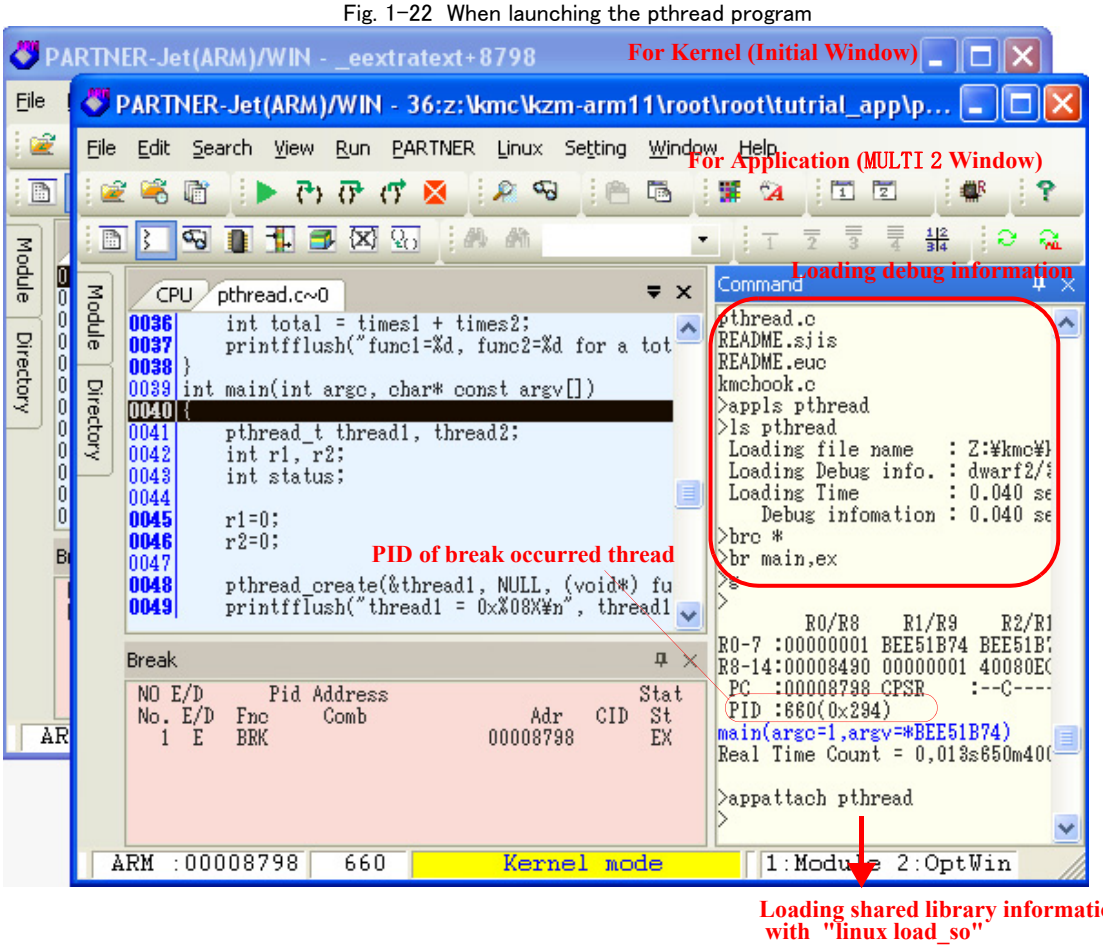
Compile it with debug options enabled.

```
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
```

Load debug information into PARTNER.

PT>multi 2 ↓
PT2>cd z:¥kmc¥kzm-arm11¥root¥root ↓
z:¥kmc¥kzm-arm11¥root¥root
PT2>appls pthreads ↓
PT2>g ↓
TGT>./pthread ↓
(Breaks at main.)
PT2>appattach pthread ↓
(Attaches to the process and loads debug information for the shared library.)

Fig. 1-22 When launching the pthread program



PARTNER is running in ADD mode in this tutorial, so you can set breakpoints in the func1 and func2 functions in the application window.

If it is executed after these breakpoints are set, breaks occur while contexts are switched in the same window as shown in Fig. 1-23 and Fig. 1-24.

Fig. 1-23 When a break occurs in a thread context

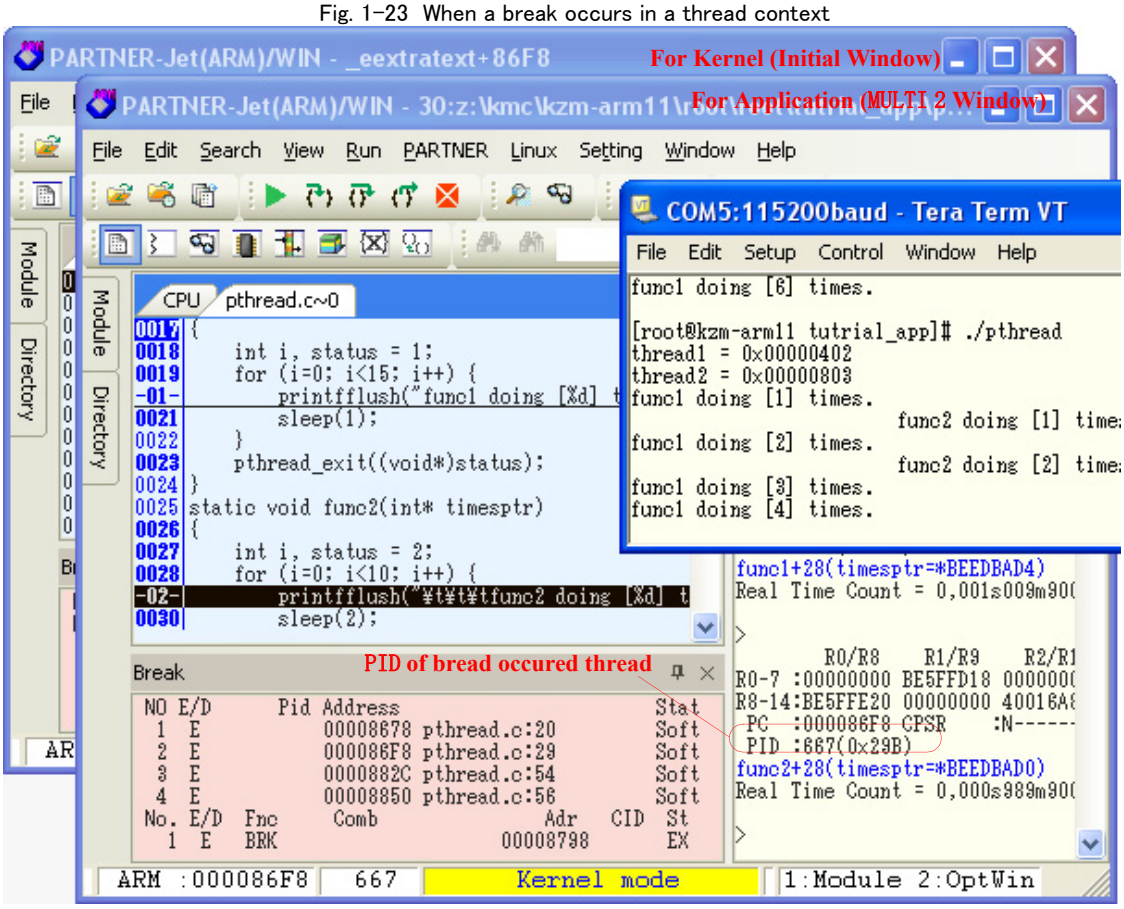
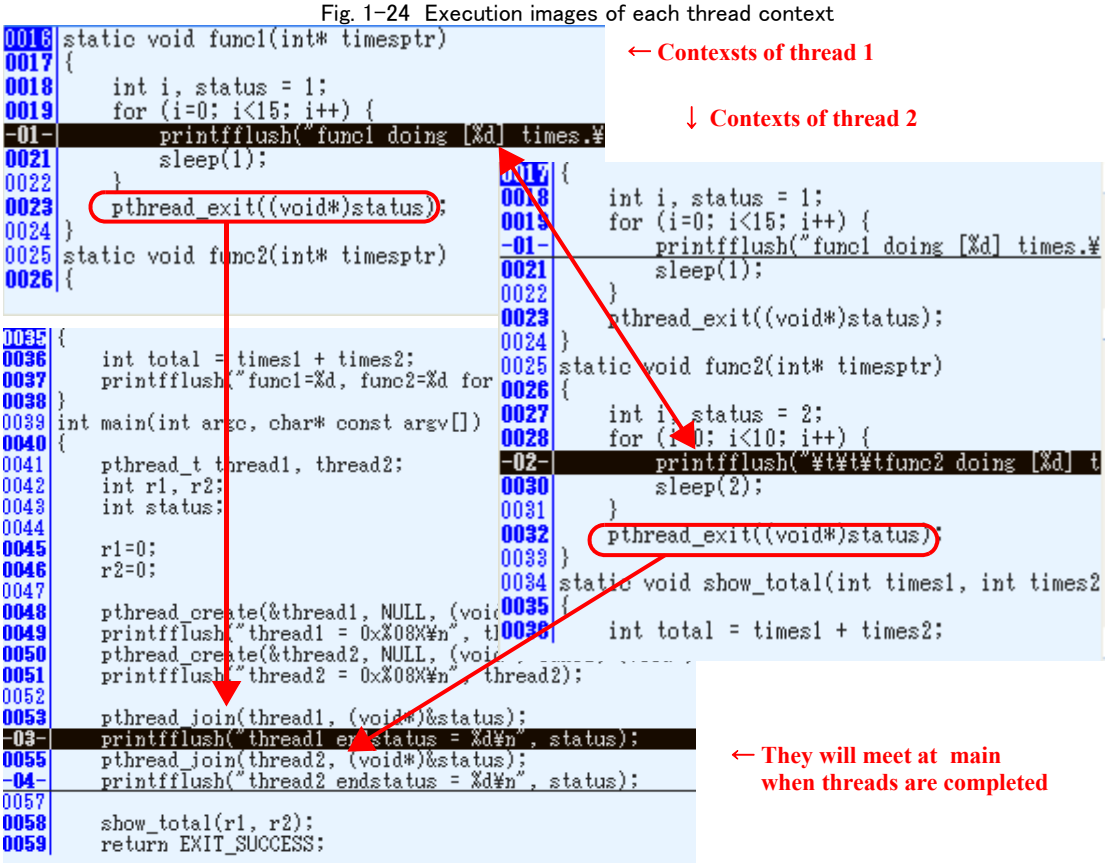


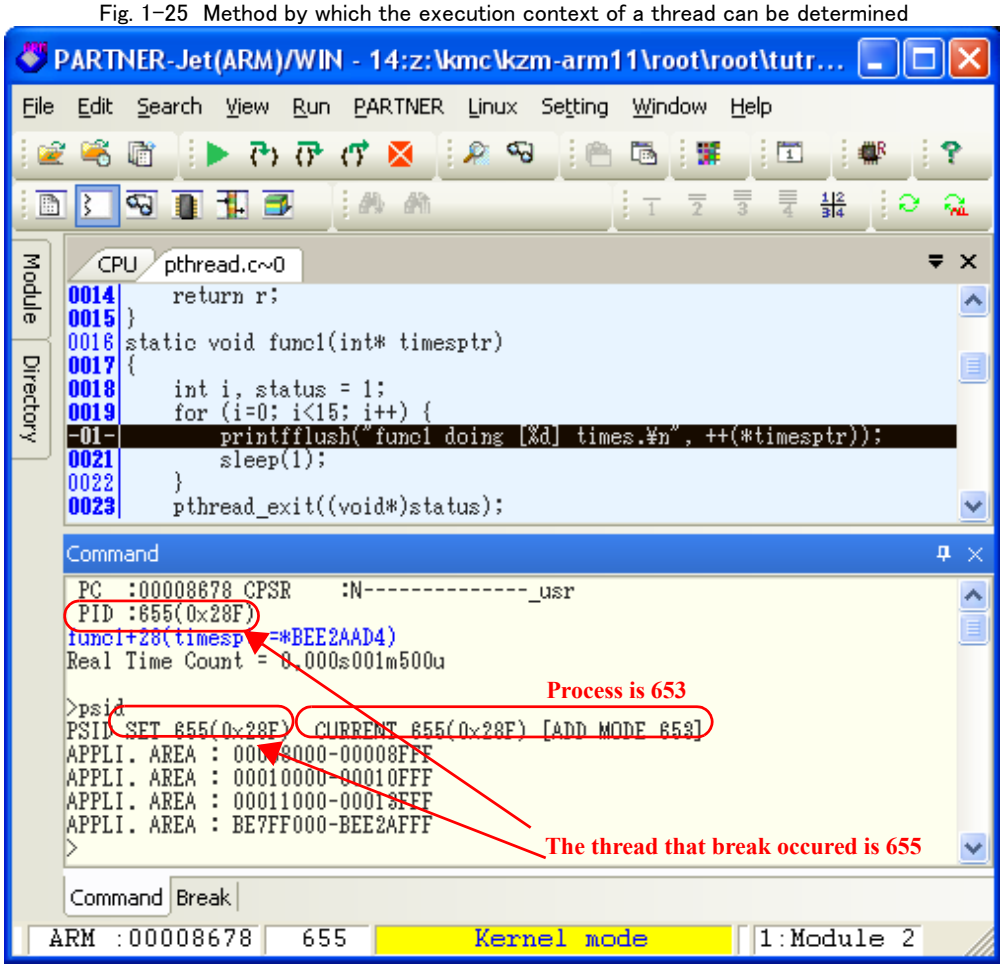
Fig. 1-24 Execution images of each thread context



As far as threads are concerned, the same function may be executed in different threads, and the program names of each thread are the same; so it is not necessarily possible to judge the execution context being broken by displaying the code window or using the "PS command (Page 160)".

You can see in which execution context a break occurred, from the PID display in the command window or using the "PSID command (Page 168)". Information about all threads that are attached within a process can be displayed using the "THREAD command (Page 163)".

Fig. 1-25 Method by which the execution context of a thread can be determined



Moreover, using the "THREAD command (Page 163)" and "K command (Page 174)" let you see the detailed states of threads.

```
PT2>thread ↓
 pid:746(0x2EA) task_struct:C6434CA0 pc:4003EE18
 pid:748(0x2EC) task_struct:C6435220 pc:4000E668
 (Displays the task_struct address in the kernel and program counter.)
PT2>
PT2>k 748 ↓
 KMC-SUPPORT.C: 468 : 4000E71C __kmc_pthread_entry+64(arg=*40001AAC)
 LIB1FUNCS.ASM: 195 : 4000E668 __kmc_sleep_thread+44()
 (Displays the function call history of the thread stack.)
PT2>
```

1.7 Parallel processing using the fork system call

Since Linux is an OS that works with multiple processes, processes can be used as a method of parallel processing. In ″Fig. 1-26 process.c″, nearly the same behavior as ″Fig. 1-20 pthread.c″ is implemented with a process that uses the fork system call. Almost every line corresponds to the counterpart in the other source code, so it is interesting to compare them.

Fig. 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;
}
```

Compile it with debug options enabled.

(Because such values as WEXITSTATUS differ from each other on the Linux PC and the target Linux, the include directory is specified using the -I option to clarify what it refers to.)

```
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
```

The execution image will be almost the same.

Fig. 1-27 Executing a process application

```
[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 ~]#
```

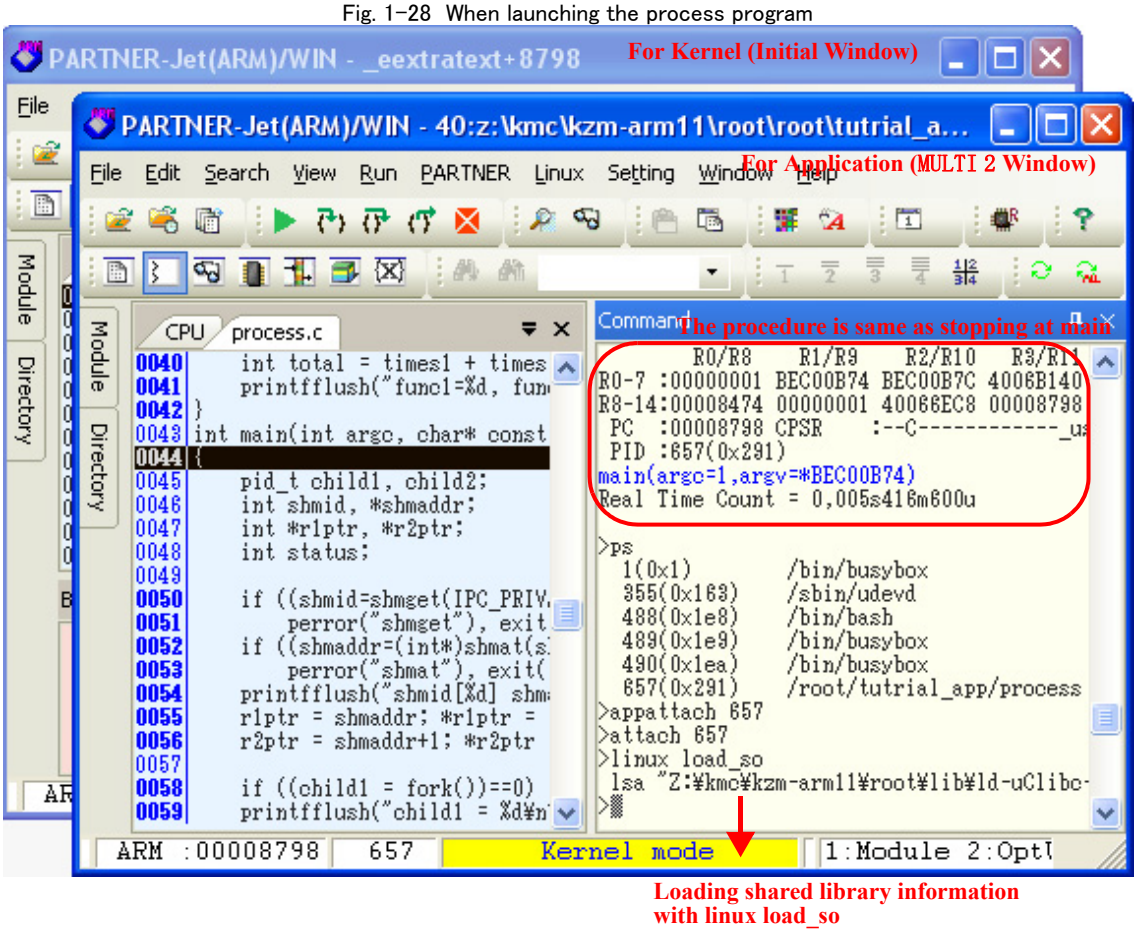
In this tutorial, this program is actually debugged using PARTNER.

The procedure is exactly the same as "1.6 Parallel processing using the pthreads library" until the step where the program is stopped at the main function after loading debug information and executing the program.

Load debug information into PARTNER.

```
PT>multi 2 ↓
PT2>cd z:¥kmc¥kzm-arm11¥root¥root ↓
z:¥kmc¥kzm-arm11¥root¥root
PT2>appls process ↓
PT2>g ↓
TGT>./process ↓ (Breaks at main.)
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 ↓
(Attaches to the process and loads debug information for the shared library.)
```

Fig. 1-28 When launching the process program



At this point, you want to set breakpoints for both func1 and func2 functions, but, unlike the sample for multi-threads, different processes are used for the func1, func2 and main functions in this case, and thus different address space areas are used for each of them. There are two problems that prevent you from using the same procedure as for the multi-thread sample.

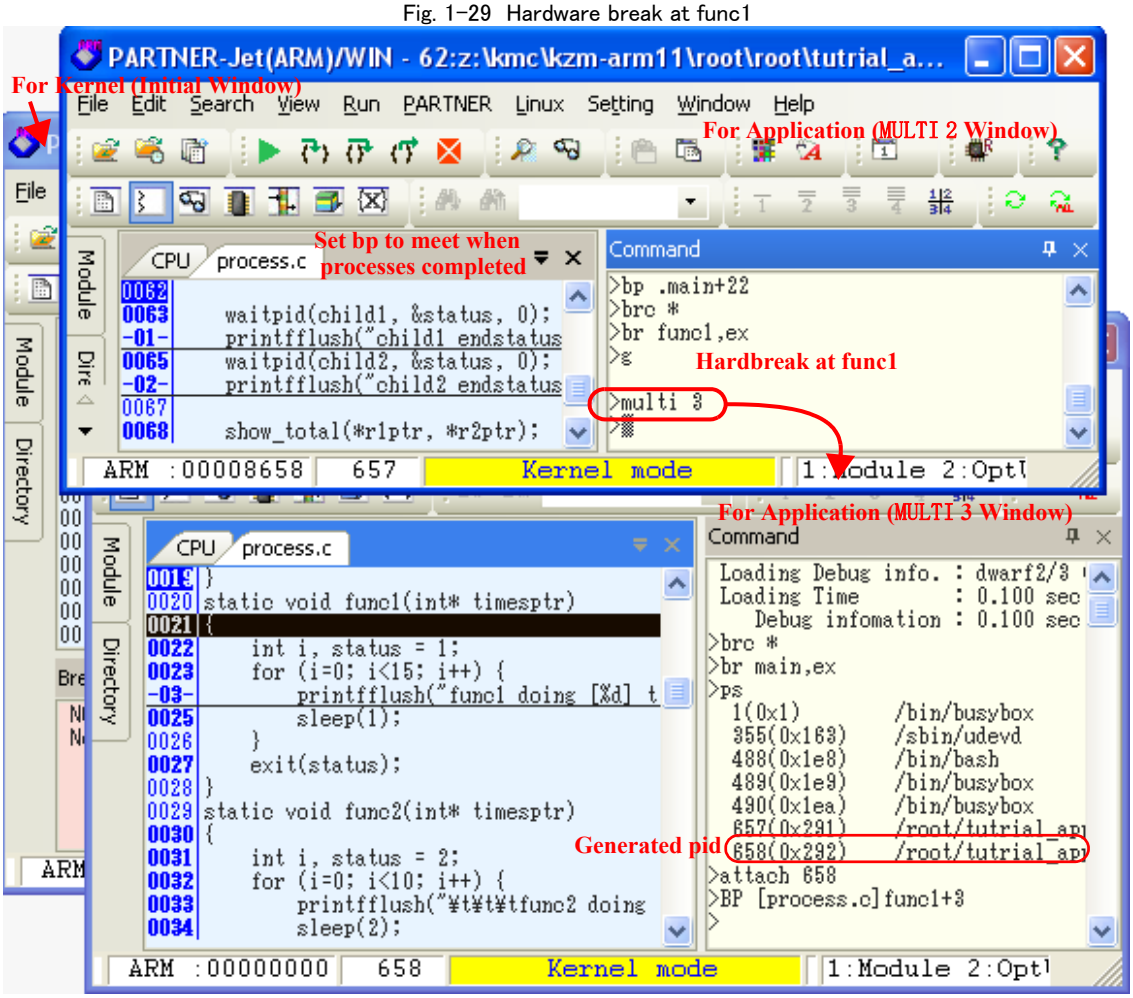
(1) At this point, software breakpoints cannot be set for the func1 and func2 functions.

(2) Debugging multiple processes in one window can confuse the user.

Use hardware breakpoints to stop the program at the func1 and func2 functions, and open an additional PARTNER window to provide dedicated windows for each process, even when the ADD mode is used.

```
(MULTI2 window)
PT2>BP .main+20 ↓   (printf sentence after waitpid(child1,...))
PT2>BP .main+22 ↓   (printf sentence after waitpid(child2,...))
PT2>brc * ↓
PT2>br func1.ex ↓
PT2>g ↓
(A hardware break occurs at func1.)
PT2>multi 3 ↓

(MULTI3 window)
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 ↓
```

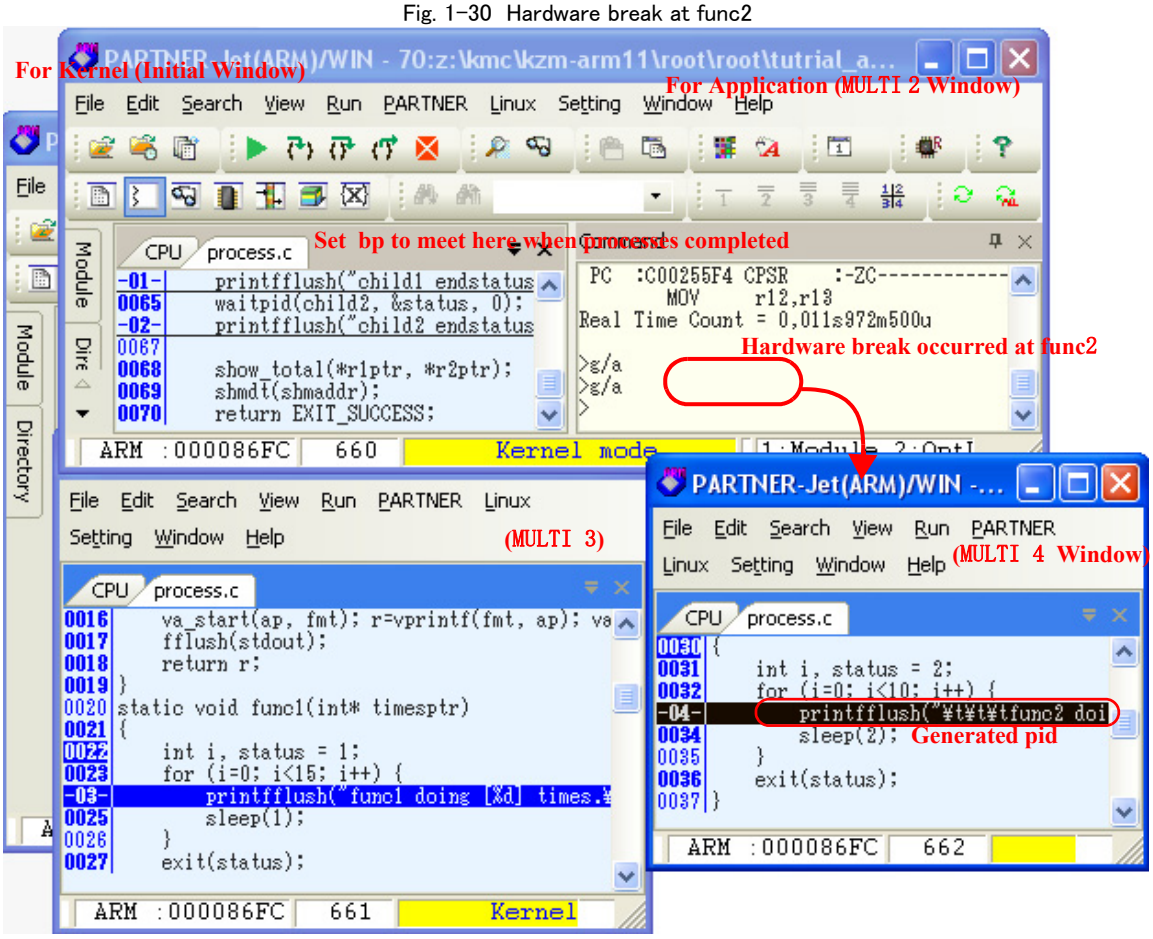
Fig. 1-29 Hardware break at func1



Also, open an additional window for the func2 function to cope with this problem.

```
(MULTI2 window)
PT2>brc * ↓
PT2>br func2.ex ↓
PT2>g/a ↓
(g/a is performed until a hardware break occurs at func2)
PT2>multi 4 ↓

(MULTI4 window)
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 ↓
```

Fig. 1-30 Hardware break at func2



Hereafter, each time the G/A command is executed, the active window is switched and the execution stops at each breakpoint.

Fig. 1-31 Software break in func1 (pid 741)

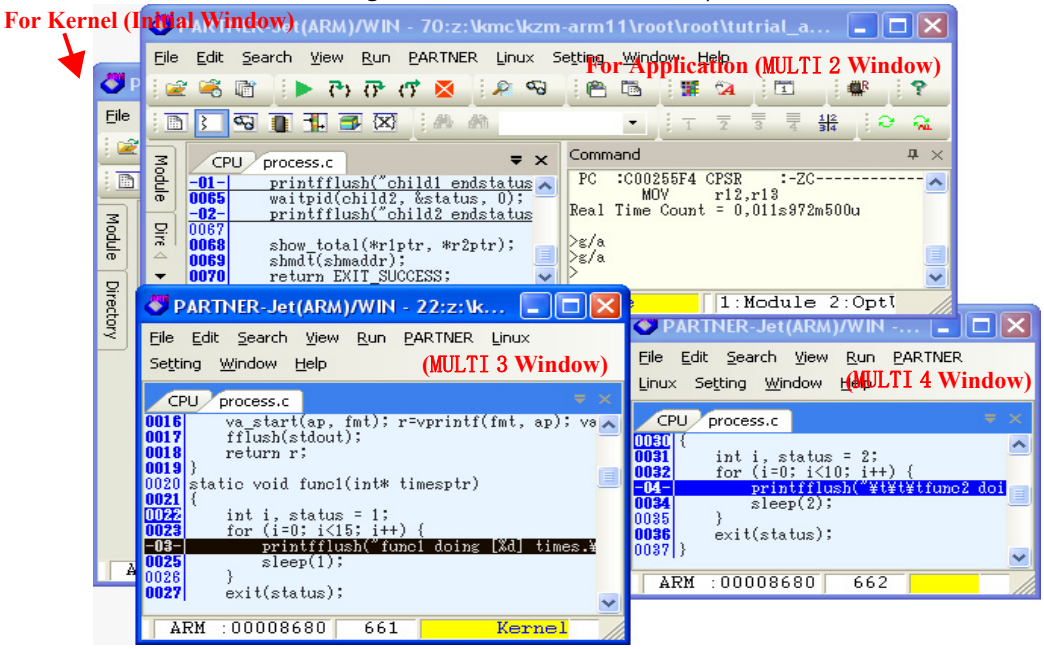
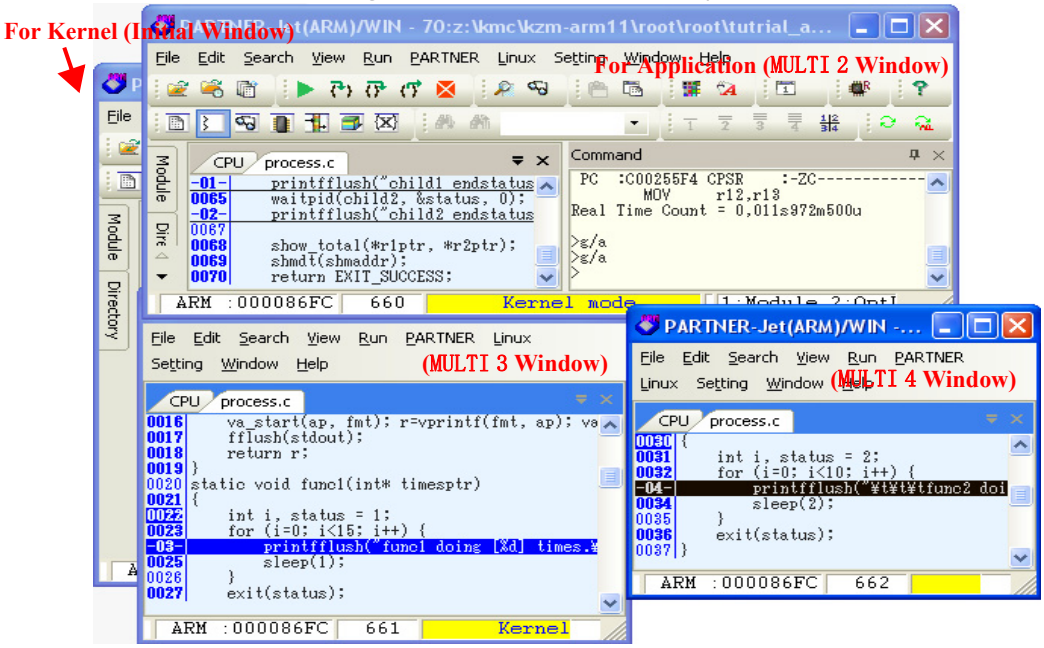


Fig. 1-32 Software break in func2 (pid 742)



If the G command is used instead of the G/A command, only the kernel and the process that is running in the target PARTNER window are affected, and the display of other PARTNER windows is changed as shown in Fig. 1-33 and processes being executed will stop and remain stopped.

Fig. 1-33 Display indicating that this application has stopped

1.8 Creating a shared library

The abovementioned samples "pthread.c (Page 24)" and "process.c (Page 29)" are very similar to each other and so can have many things in common. The commonly used part is made into a library. If it is linked from multiple programs that may run simultaneously, a shared library has an edge over a static library in terms of resource efficiency, such as memory usage.

The following sample codes were changed so that a shared library is used.

Fig. 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___ */
```

Fig. 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;
}
```

The following sample codes are for the part that uses the shared library samplelib.

Fig. 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;
}
```

Fig. 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;
}
```

Compile the shared library with debug options enabled.

```
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
```

Install the shared library.

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

Also, compile the program with debug options enabled.

```
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
```

You can confirm linked libraries using the ldd command on the target.

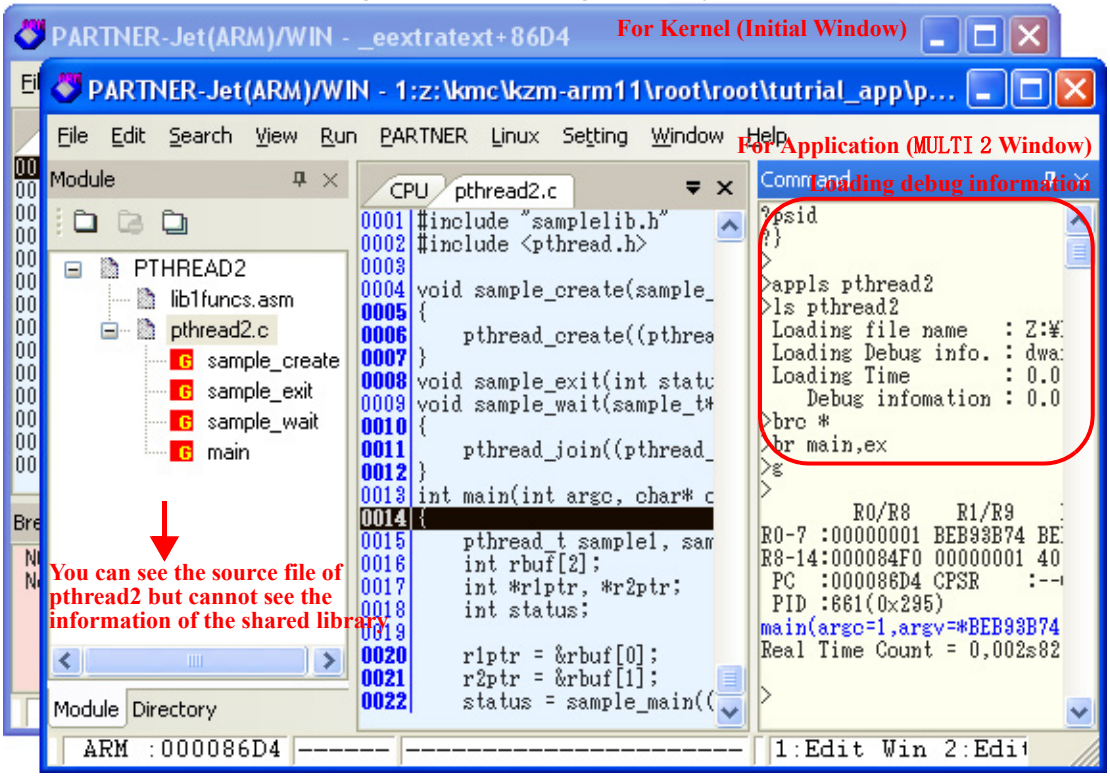
```
TGT>ldd pthread2 ↓
        libsample.so => /usr/lib/libsample.so (0x4000e000)
        libpthread.so.0 => /lib/libpthread.so.0 (0x40017000)
        libc.so.0 => /lib/libc.so.0 (0x40031000)
        libdl.so.0 => /lib/libdl.so.0 (0x40086000)
        ld-uClibc.so.0 => /lib/ld-uClibc.so.0 (0x40000000)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

Since the behavior of pthread2 and process2 is nearly the same, using the pthread2 program, we confirm the debugging done in libsample.so that has been made into a shared library.
The procedure used until a break occurs at the main function is the same as the previous ones.

```
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 ↓ (Breaks at main.)
```

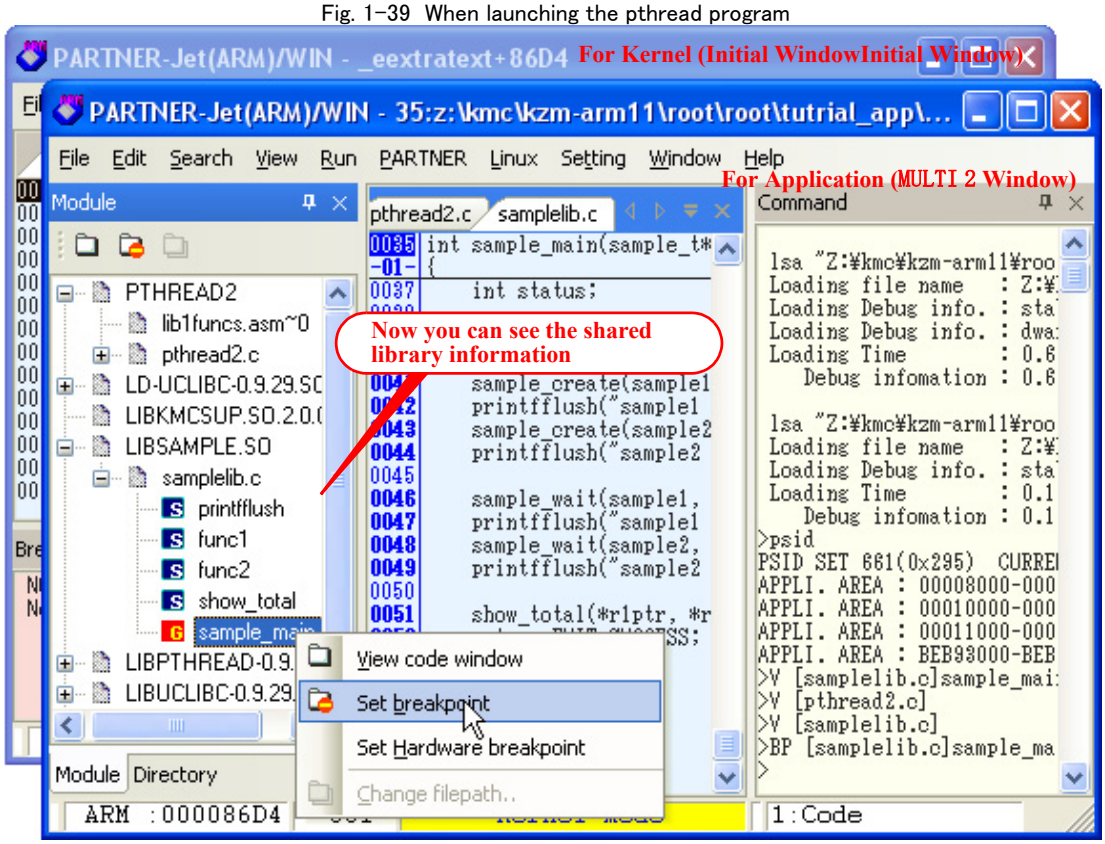
Information of the shared library has not been loaded yet at this point, so samplelib.c is not included in the files and functions that are displayed by choosing Display -> Module.
By attaching to the process ("ATTACH command (Page 162)") and resolving shared library information ("LINUX command (Page 164)"), you can load debug information of the shared library into PARTNER and perform debugging.

Fig. 1-38 When breaking at main in pthread2



PT2>appattach pthread2 ↓ (Attaching to the process and loading debug information of the shared library)

Fig. 1-39 When launching the pthread program



Now you also can set breakpoints to the code within the shared library.

1.9 Executing external commands from an application

In "1.7 Parallel processing using the fork system call", multi-process is used to perform parallel processing for multiple contexts. There is another reason for using multi-process, which is to execute another executable file (external command).
It is necessary to load debug information for the program to be launched at any point in time when debugging an external command. As examples, this section describes three types of APIs used to execute external commands.

When using the system call fork(2) + exec(3)

It is a low-level API; but it is often used because it can handle detailed process behavior. In Linux, exec(3) is implemented as part of a group of functions that call execve(2). In "Fig. 1-40 cmd_forkexec.c", the execlp function is used instead of execve(2), which is rarely used directly.

Fig. 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;
}
```

Compile it with debug options enabled.

```
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
```

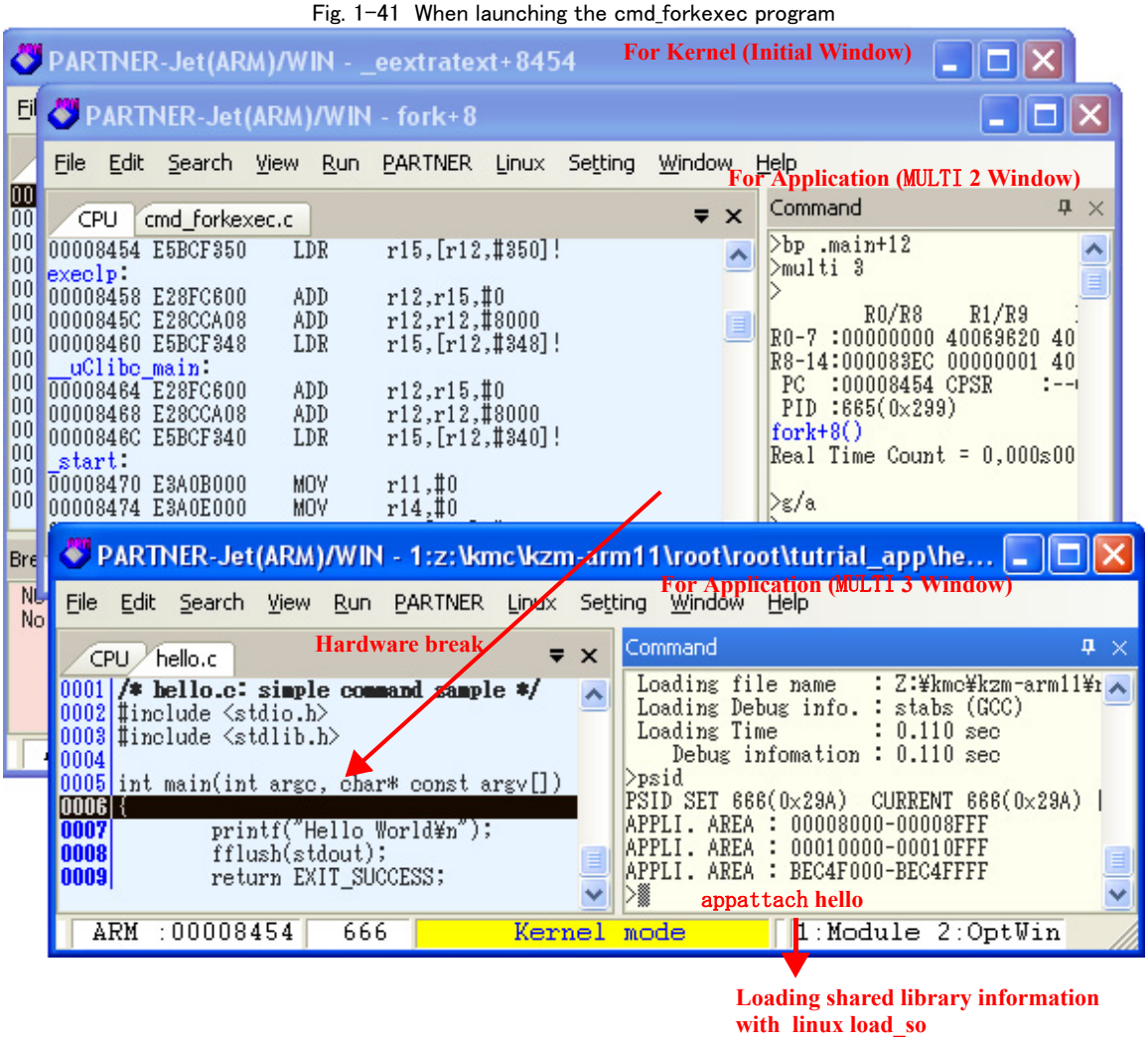
From start up and until stopping at the main function, the process used is the same as the previous procedures.

```
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 ↓  (Breaks at main.)
PT2>appattach cmd_forkexec ↓  (Attaching to the process. Specifying the process ID is also
allowed.)
```

If your program is more complicated than this sample program, the name of an external command that you want to execute should have already been determined before calling the fork function. Open another PARTNER window and load the debug information of the external command (./hello).

```
(MULTI2 window)
PT2>BP .main+7 ↓   (execlp)
PT2>BP .main+12 ↓   (printf sentence after waitpid)
PT2>multi 3 ↓

(MULTI3 window)
PT3>appls hello ↓
PT?>g/a ↓   (Any PARTNER window is OK.)
(A hardware break occurs at main in hello.)
PT3>appattach hello ↓   (Attaching to the process. Specifying the process ID is also allowed.)
```

Fig. 1-41 When launching the cmd_forkexec program



Hereafter, you can debug one process in each window as described in ″Parallel processing using the fork system call″ (Page 29).

When using the popen(3) function of POSIX

There are cases where a process that launched an external command receives the output result to the standard output from the launched external command (child process), and processes the received output result (filter processing). In "Fig. 1-42 cmd_popen.c", the output from the child process is routed to the standard output as is. Nearly the same behavior can be performed with the following shell script.

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

The I/O function (between parent and child processes) that can be used with the popen function supports only unidirectional reading from the standard output of a child process or writing to the standard input of a child process; so it is not suitable for launching a program such as "Fig. 1-16 hellohello.c", shown previously, in which both standard input and standard output are used.

Fig. 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);
}
```

Compile it with debug options enabled.

```
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
```

From start up and until stopping at the main function, the process used is the same as the previous procedures.

```
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 ↓ (Breaks at main.)
```

```
PT2>appattach cmd popen ↓ (Attaching to the process, and loading information on the shared
library)
```

Because parent and child processes are connected via the pipe I/O, if the child process yields a large output, the child process may have to wait for it to write to the standard output, depending on the processing load of the parent program, and thus the behavior of the two processes should differ from "cmd_forkexec.c" (Page 39).
However, the behavior will be nearly the same since the hello.c program yields only a small output.

The debug procedure is also nearly the same, so only the procedure is shown below.

```
(MULTI2 window)
PT2>BP .main+12 ↓  (printf sentence when fgets is done)
PT2>BP .main+14 ↓  (printf sentence after pclose)
PT2>multi 3 ↓

(MULTI3 window)
PT3>appls hello ↓
PT?>g/a ↓  (Any PARTNER window is OK.)
(Stops at a hardware breakpoint.)
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
(hello has not been launched yet.)
PT3>g ↓
(Stops again at a hardware breakpoint.)
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
(This time, hello is already running.)
PT3>appattach 743 ↓
PT3>BP .main+1 ↓  (Set software breakpoints properly.)
PT3>BP .main+2 ↓
PT3>BP .main+3 ↓
PT?>g/a ↓  (Any PARTNER window is OK.)
```

Hereafter, each time the G/A command is executed, the MULTI2 or the MULTI3 windows become active and the execution stops at one of the set software breakpoints.

When using the system(3) function of the C standard library

This function is known as the simplest process handling function, but is different from the previously mentioned two methods in that a shell program is actually executed and the argument of the function is a command string, which is passed on to the shell. (Normally it is executed in the format of "/bin/sh -c command"). This is because the argument string is executed as the child process of the shell (except in cases where the argument string is the exec command), and so at least two processes are launched in order for the argument string to run as a command.

Fig. 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;
}
```

Compile it with debug options enabled.

```
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 ↓
```

The debug procedure is the same as the previous ones.

```
PT>multi 2 ↓
PT2>cd z:¥kmc¥kzm-arm11¥root¥root ↓
z:¥kmc¥kzm-arm11¥root¥root
PT2>appls cmd_system ↓
PT2>g ↓
TGT>./cmd_system ↓ (Breaks at main.)
PT2>appattach cmd_system ↓ (Attaching to the process, and loading information on the shared library)
PT2>BP .main+7 ↓  (Set software breakpoints properly.)
PT2>multi 3 ↓
PT3>appls hello ↓
PT?>g/a ↓  (Any PARTNER window is OK.)
(Stops at a hardware breakpoint.)
PT3>ps ↓  (Confirms that the hello process is running.)
  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 ↓
```

Hereafter, you can debug both processes.

1.10 Creating a loadable module

Although Linux supports many devices, you may have to create your own device driver when developing an embedded device.
Because device drivers run in the Linux kernel space, it is easier for JTAG-ICE debuggers to debug them than applications in the user space. A case that might cause a problem is where a driver is dynamically installed into the kernel in the form of a loadable module after the system boot.
A loadable module is installed into the kernel using the insmod command on the target, but the address will not be determined until the installation is complete. It is difficult for a debugger to start debugging from the module_init function, which is called when the module is initialized.
But PARTNER provides an easy procedure to debug such modules, due to the effects obtained by the [Loadable module auto attach] and [Loadable module auto attach (patch is include in kernel)] items in the [PARTNER Debugging] menu, which were enabled when the Linux kernel was configured.

The loadable module forms are different between Linux 2.4 and 2.6. The procedure described in this chapter is only valid for the kernel 2.6.

Fig. 1-44 kzmled.c is a program that only has module_init and module_exit, which are the simplest interfaces for loadable modules.
In order to intelligibly depict that the loadable module is executed in the kernel space and is able to directly access hardware, the onboard LED of the KZM-ARM11-01 board turns on when the module is loaded or unloaded.

If you want to compile Fig. 1-44 kzmled.c using an environment other than the KZM-ARM11-01 board, the part that controls LED access is disabled. In that case, only message output by printk is performed.

Fig. 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);
```

To compile this source file, first we have to create Makefile shown in Fig. 1-45, because the compile settings in the source tree of the Linux kernel are used to create a Linux kernel 2.6 loadable module (.ko).

Fig. 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
```

This procedure should be performed under the home directory of the root user on the target.

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

Compile it.

```
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]: Moves into the directory `/opt/kmc/kzm-arm11/build_src/linux'.
  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]: Goes out of the directory `/opt/kmc/kzm-arm11/build_src/linux'.
LINUX86> ls ↓
Makefile  kzmled.c  kzmled.ko  kzmled.mod.c  kzmled.mod.o  kzmled.o
(kzmled.ko is output.)
```

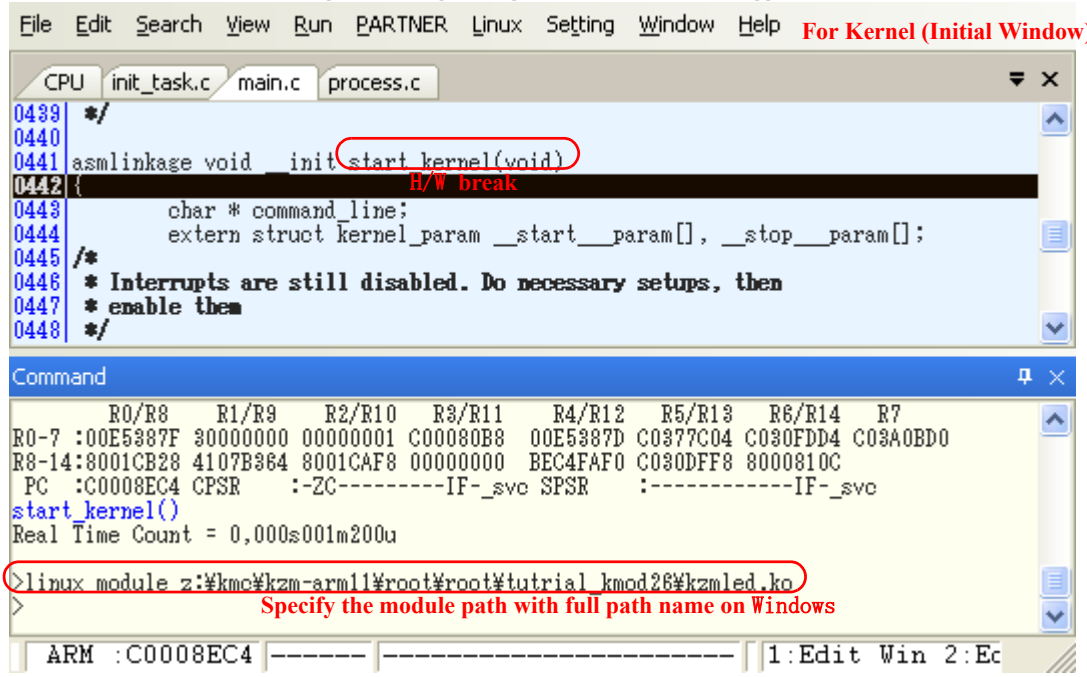
Because the compile settings in the Linux kernel source tree are used, whether or not to build it with debug information will be the same as the kernel source.

Now the actual debugging procedure is described below.

It is different from debugging of other programs in that the module to be debugged must be registered with PARTNER in advance using the "LINUX command" (Page 164) to debug the loadable module.

```
PT>load_linux26_nfs ↓
PT>g ↓
(Breaks at start_kernel)
PT>linux module z:¥kmc¥kzm-arm11¥root¥root¥tutrial_kmod26¥kzmled.ko ↓
(Registers the module to be debugged.)
PT>g ↓
```

Fig. 1-46 Registering the module to be debugged



It is possible to perform the "linux module <path name>" operation at any time before insmod is executed for the loadable module, but the Linux kernel must have already been enabled (after the start_kernel symbol).

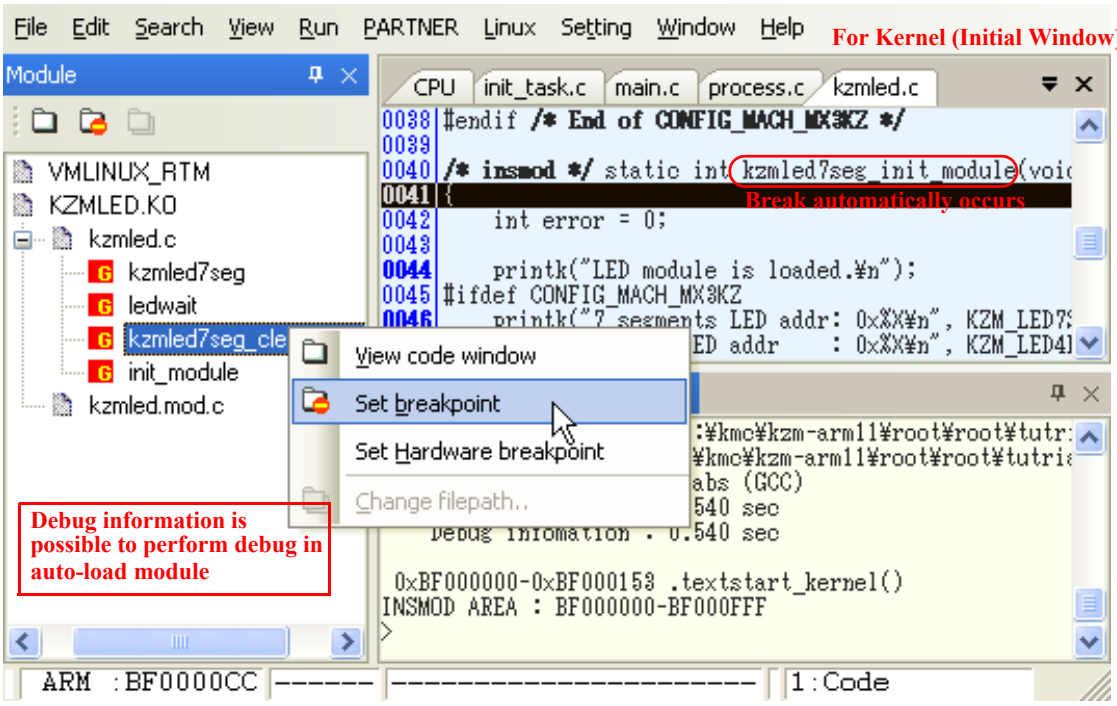
You can confirm the registration of modules to be debugged using the "linux module" command, and delete them using the "linux module clr" command.

For details, refer to "LINUX command" (Page 164).

Once Linux is launched, execute the insmod command as the root user to install the loadable module into the kernel.

```
TGT>login: root ↓
TGT># cd tutrial kmod26 ↓
TGT># insmod kzmled.ko ↓
(A break automatically occurs on PARTNER.)
```

Fig. 1-47 Starting to debug the module



Hereafter, you can freely set software breakpoints.

For example, if you set a breakpoint to cleanup_module, the execution stops at this breakpoint when the module is unloaded on the target.

```
TGT># rmmod kzmled ↓
```

Chapter2 Linux debugging and environment configuration

This chapter describes configuration settings used to perform more advanced debugging on a Linux system using PARTNER for Linux.

2.1 Overview of the configuration for Linux debugging

PARTNER actively utilizes information within the Linux kernel to provide a comfortable debug environment for Linux. To do so, PARTNER needs some settings configured, in addition to debug information for the Linux kernel. This chapter provides an overview of the configuration necessary to perform advanced debugging.

2.1.1 Configuration settings for Linux debugging

Table 2-1 shows a list of configuration settings necessary for Linux debugging.

Table 2-1 Configuration settings for Linux debugging

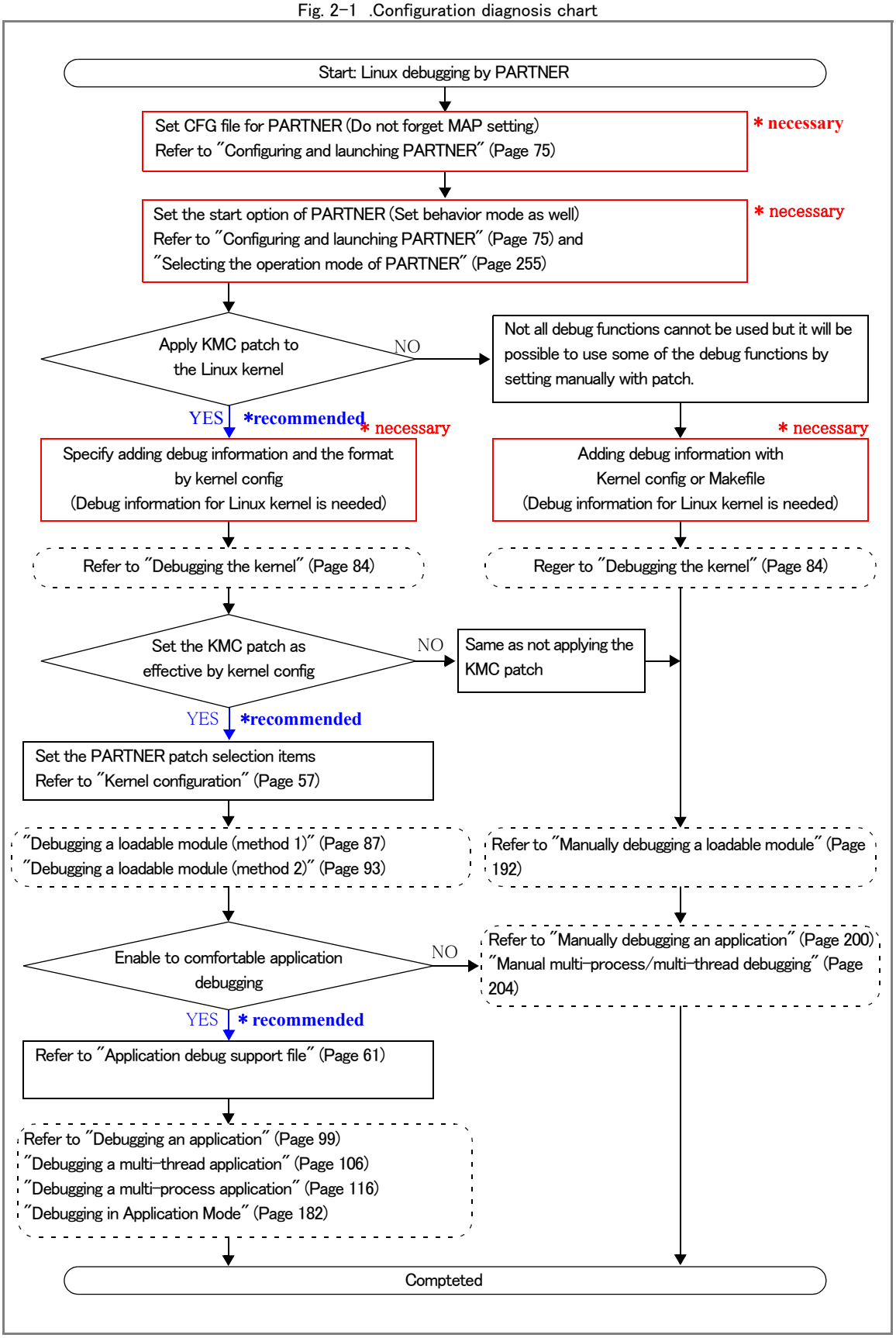
Type	Description
Kernel Modification	Installs the features that make debug kernel body conveniently. It is provided by KMC as a patch, and user can easily switch on and off for each item. Refer to "Modifying and configuring the Linux kernel source" (Page 54)
Application Modification	Installs the features that make debug user mode program (application) conveniently to the application. It is provided as Application debug support file by KMC. How to install to the application, refer to "Application debug support file" (Page 61)
PARTNER Setting	Describs the setting that enables PARTNER to handle target board. The format is expanded for Linux debugging and PARTNER recognizes the relationship between kernl and user mode program. Refer to "CFG file extensions" (Page 140)
PARTNER Start Option	Specify the path name differnce between build envoironment and on target (converting path name in debug information), Linux kernel version on target and operation mode of PARTNER. Refer to "Launch option" (Page 146)

In addition to these configuration settings, as functions specialized for Linux debugging, PARTNER commands that can be used as an operation during debug are added for Linux (refer to "Additional command" (Page 159)".

2.1.2 Configuration diagnosis

Fig. 2-1 shows choices to determine what configuration is needed to provide a more comfortable debugging environment and what can still be done when some configuration settings and modifications for debugging are omitted

Fig. 2-1 .Configuration diagnosis chart



2.2 Modifying and configuring the Linux kernel source

This chapter describes how to modify part of the Linux kernel source in order to perform advanced debugging.

Modification performed for the kernel source tree is provided as a batch file.
Modification consists of an additional directory (KMC/) and the modification of some files.
Hereafter, the directory, in which the Linux kernel was uncompressed, is described as $(TOPDIR)/.

2.2.1 Necessity of modifying the Linux kernel source

To perform Linux debugging using PARTNER, the debug information of the Linux kernel is absolutely necessary.

Note that even if you don't need to debug in the kernel space and only debug applications in User Mode, the debug information of the Linux kernel is also necessary.

Normally, no debug information setting is added to the source tree of the Linux kernel. Even if debug information settings are added to the kernel configuration menu, you cannot specify the type of the debug information. By specifying the debug information type and modifying part of the Linux kernel source, you can perform more advanced debugging. To do so, an item for debug information settings is added to the kernel configuration menu for PARTNER, and a patch that applies modification to part of the kernel is provided.

Automating loadable module debugging

If the Linux kernel has been modified, PARTNER performs a break and automatically loads the debug information when a loadable module has been installed, and then you can start debugging.

Support of Application Mode

Modifying the Linux kernel makes it possible to perform debugging in Application Mode. For details on debugging in Application Mode, refer to "5.1 Debugging in Application Mode (Page 182)".

Displaying real-time trace for each process

If you are debugging the Linux kernel and an application in separate PARTNER windows, you can only display the real-time trace of the relevant process in the history window.

Support of the Event Tracker

You can obtain data necessary to display the Event Tracker by modifying the Linux kernel, and display the process transition in the Event Tracker window.

2.2.2 Applying the patch

The patch file is included in the accompanying CD-ROM.

Apply the patch file to the Linux kernel source using the patch command as shown below.

```
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 ↓
```

The patch file is provided for major versions of the Linux kernel. (Not for all the versions)
Files to be modified differ depending on the kernel version.

2.2.3 Notes on manual modification

If the patch file is not provided for the Linux kernel you use, you are to manually modify the Linux kernel source. The modification to be made should be based on the patch file for the nearest version; but note that there are some parts that require special care.

Modification $(TOPDIR)/kernel/sched.c

Insert _KMC_SCHED_CALL(prev,next). Make sure you insert it before executing switch_to(prev,next,prev); (it is necessary for both 2.4 and 2.6 series Linux).

As for parts to be modified in sched.c, note that the position at which the switch_to() function is written differs greatly according to the kernel version.

2.2.4 Notes on manual modification for MIPS series (version 2.4.x)

If you manually modify a kernel 2.4.x for MIPS CPUs, take special care when modifying the following.
This modification is to make it possible to use the system call sys_gettid().
Confirm that SYS(sys_gettid,0) is included in $(TOPDIR)/arch/mips/kernel/syscalls.h

Modification $(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)
```

The system call sys_gettid() is used in the debug support library file (kmc-support.c). If SYS(sys_gettid,0) is declared in arch/mips/kernel/syscalls.h, it is not necessary to make this modification.
If SYS(sys_gettid,0) is not declared, write it until sys_gettid, according to the order of the _NR_xxxx declaration in include/asm-mips/unistd.h.

2.2.5 Kernel configuration

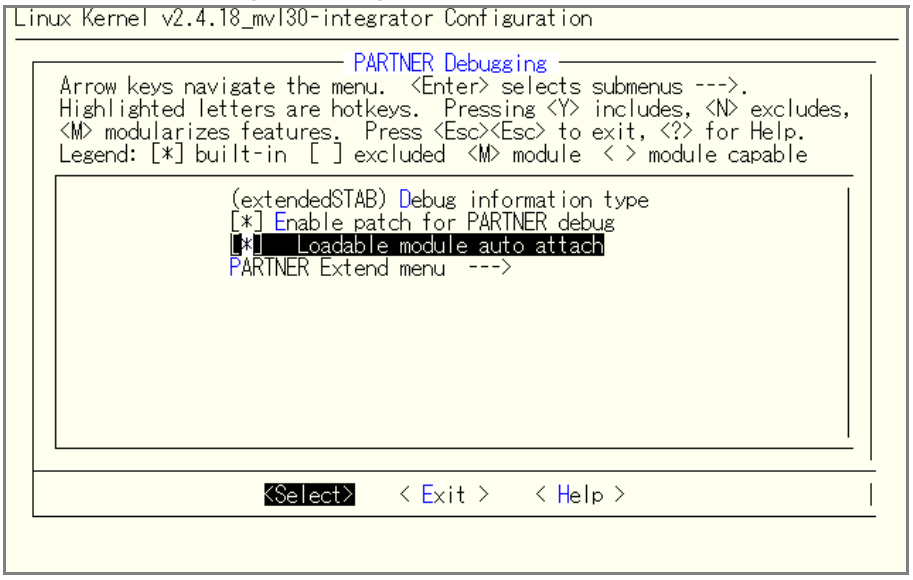
Once the Linux kernel source tree is modified, [PARTNER Debugging] is added to the kernel configuration menu. Using this, you can configure items related to debugging on PARTNER.

```
LINUX86>make menuconfig ↓
```

【For Linux 2.4.x kernels】

[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]

Fig. 2-2 Configuration for Linux 2.4.x kernels



【For Linux 2.6.x kernels】

[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)

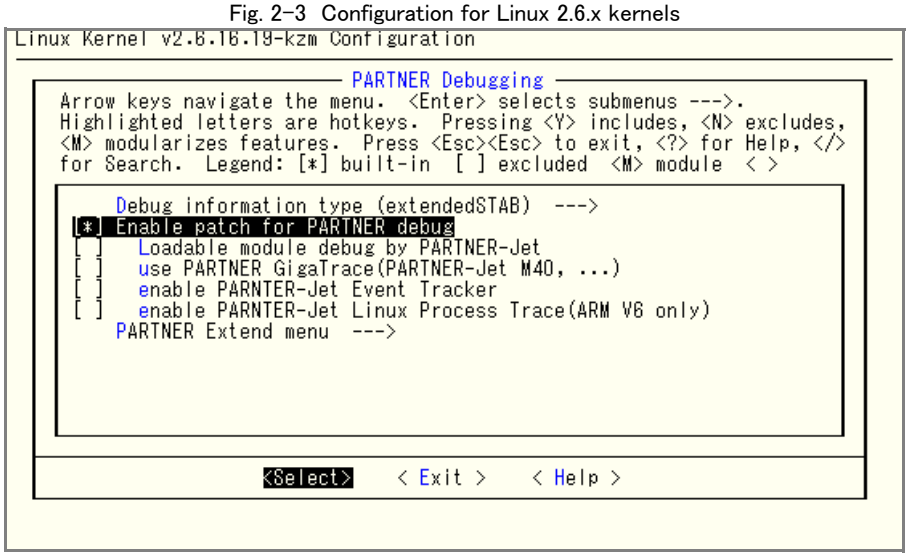
├──→ [use PARTNER GigaTrace(PARTNER-Jet M40, ...)]

├──→ [enable PARNTER-Jet Event Tracker]

├──→ [enable PARNTER-Jet Linux Process Trace(ARM V6 only)]

└──→ [PARTNER Extend menu]

Fig. 2-3 Configuration for Linux 2.6.x kernels



● Debug information type

Select the type of debug information that is added to the Linux kernel.

You can select it from NONE, STAB, extendedSTAB, DWARF-1, extendDWARF-1, and DWARF2.

We recommend extendedSTAB(-gstabs+). If you find that the debug information is not correct after loading the Linux kernel file with debug information into PARTNER, try other formats (e.g. DWARF2).

● Enable patch for PARTNER debug

This setting enables kernel source modification so that the kernel setting can support advanced debugging on PARTNER.

Disabling this item results in the same state as if the patch had not yet been applied.

● Loadable module debug by PARTNER-Jet

This setting breaks on PARTNER when a loadable module is installed, and automatically loads debug information.

For a 2.4.x kernel, only [debug hook in module side] can be selected. But, for a 2.6.x kernel, you can select either [debug hook in module side] or [debug hook in kernel side]. [debug hook in kernel side] is the recommended setting.

For details, refer to "3.2 Debugging a loadable module (method 1) (Page 87)" and "3.3 Debugging a loadable module (method 2) (Page 93)".

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

This setting is to use the Giga Trace function. It requires an ICE that supports the Giga Trace function such as PARTNER-Jet Model 40. For details, refer to the manual provided with each device that supports the Giga Trace function.

● enable PARNTER-Jet Event Tracker

This setting is necessary to visualize the execution history of processes and threads using the Event Tracker function of PARTNER. For details, refer to "Chapter 6 Event Tracker (Page 211)".

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

This item is only displayed for ARM CPUs that support V6 commands. Normally, use the default setting.

● PARTNER Extend menu

This setting is not usually changed, so do not modify it under normal conditions.

Table 2-2 summarizes the scope of application of these configuration items.
◎ indicates that enabled is mandatory; and empty indicates that either enabled or disabled is OK. Enabling all settings in the table does not create any particular problem.

Table 2-2 Scope of application of debugging configurations

Setting items / Debug target	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
Kernal debugging	◎					
Module debugging (Method 1)	◎	◎	◎	—		
Module debugging (Method 2)	◎	◎	—	◎		
Application debugging	◎	◎				

2.2.6 Effects on behavior that caused by kernel modification

[PARTNER Debugging] in the kernel configuration menu allows a user to globally turn on or off the modifications made to the Linux kernel source tree. You can create a release version simply by rebuilding the kernel after reconfiguration, without explicitly deleting the applied modifications from the Linux kernel source tree.

Behavior is not affected even if you use the JTAG debugger without connecting it in the kernel where the debugging configuration using PARTNER is enabled in [PARTNER Debugging] in the kernel configuration menu.
However, a slight overhead occurs when performing context switching and when terminating a process.

2.3 Application debug support file

This chapter describes the application debug support file used to perform advanced debugging.

2.3.1 Necessity of the debug support file

A characteristic of Linux applications viewed from PARTNER is that they are "objects that function in the virtual addresses of logical multiple spaces".
Virtual logical addresses are assigned to each process (application) respectively. Normally, only part of the 4GB space is used.
Processes work on virtual addresses that are determined when an application is created (linked). The virtual address on which a process starts to work is the same for almost every process.

On AM33 CPUs	Around 0x08000000
On ARM CPUs	Around 0x00008000
On MIPS/SH CPUs	Around 0x00400000

That is to say, the initial address of a process is determined and fixed for each CPU architecture.
Naturally, different processes cannot use the same physical memory at the same time. In order to provide the same address that is physically different, the Linux kernel and the MMU of a CPU create such logical space.
From PARTNER, you can only see the physical CPU and memory. To debug a process (application), you therefore have to resolve these logical addresses and notify PARTNER of information to discern what programs run in the same address space.

In order to fully utilize the application debug function, it is necessary to insert a stub function using the debug support file (kmc-support.c) to resolve the virtual addresses of processes. If there is a stub for application support in the target environment, PARTNER automatically resolves the virtual address space for applications.
There are various ways of using the debug support file.

(1) After directly linking the debug support file (kmc-support.c) to an application, insert the debug stub function (_kmc_start()) into the application source code (old method). Or make the support file into a shared library and link to it from the application.

(2) Link the debug support file (kmc-support.c) to a library to which nearly every program links (libc etc.) , and then attach to it.

(3) Use the debug support file (kmc-support.c) as a preload library (libkmcsup.so) and then attach to it (new method).

Every method has both merits and demerits. A comparison of these methods is shown in Table 2-3.

Table 2-3 Comparison between debug support file linking methods

Link method	For-mat	Advantage/Disadvantage	Start debugging
(1) Link directly to the application	.o	○ Modification affects application only. × Must in-line stab function in main() and the top of thread × Need to modify application link setting × An exception is occurred when starting the stab function in-lined program with JTAG debugger unconnected.	Break at stub function
(1)' Link a shared library directly to an application	.so		Break at stub function
(2) Link to a libc	.o	○ No need to modify each application ○ Auto-hoock starting thread × The library as libc (glibc and uclibc) is complicated and difficult to put batch. △ Some people feel it is not good to insert a debug stub in library that affect whole system	Attaching to a process
(3) Preload library	.so	○ No need to modify each application ○ No problem with library in file system of release version ○ Can be off with just changing LD_PRELOAD variables. ○ Auto-hock starting thread × It may stop at the different process when a break performed (br main,ex) at program start. △ Cannot debug if Preload setting is missed.	Attaching to a process

The new method (3) using the preload library also has a problem. But compared with the method (1) that directly links the debug information file to an application, it has a distinct advantage in that threads generated in a shared library can be particularly easily debugged, there being no need to insert the stub function (_kmc_start()) into points where threads are generated.

Although more comfortable debugging can be performed using the method (2) that links the debug support file to libc, the preload library method (3) has an edge on easier implementation and deletion, as all you have to do is change the setting of the LD_PRELOAD environment variable when switching between debugging and non-debugging operations. For the above reasons, we recommend the preload library method (3).

A stub function can also be used with the methods (2) and (3). If you consider a stub function as "a function that automatically breaks in the execution context just once", you can see that it is always useful when developing. Organized information on usage of stub functions is shown in Table 2-4.

Table 2-4 Usage of the debug support file

	Stub function	Link method
①	Null	(2) Link to a libc (3) Preload libkmcsup.so
②	Is it OK to insert it at arbitrary position such as main etc.?	(1) Link kmc-support.c (1) Link libkmcsup.so (2) Link to a libc (3)Preload libkmcsup.so
③	Is it necessary to insert to a thread and a child process?	(1) Link kmc-support.c

In this document, usage is described for the (3) preload library method and the (1) application direct link method, in which the scope to be modified is closed in a single application.

2.3.2 Preload library method

In the preload library method, build the debug support file in the form of a shared library, install it into the target, and then configure it as a preload library.

Creating the library

The support library for application debugging is provided as source code, and consists of two files.

Makefile
kmc-support.c

Create the library following the procedure below.

(1) Compiling

Makefile expects the make command to be executed by specifying the cross compiler (ARCH variable) and CPU type (CPU variable). If you want to change the compiler to be launched, edit Makefile and specify the gcc compiler for the target as CC.
If compiling is successfully performed, two files, libkmcsup.soand libkmcsup.so.2.0.0, are created (libkmcsup.so is a symbolic link to libkmcsup.so.2.0.0).

【Example for ARM series ARM 11 CPU】

```
LINUX86> tar zxvf libkmcsup-XXXXXX.tgz ↓
LINUX86> cd libkmcsup ↓
LINUX86> make ↓
Specify the cross compiler. ARCH={arm,mips,sh}
make: *** [envchk] Error 1
LINUX86> make ARCH=arm ↓
Specify the CPU CPU={arm7,arm9,arm11}
make: *** [envchk] Error 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) Remembering address information

The line next to the displayed PARTNER COMMANDLINE is the offset address information of the symbol _kmc_sleep_thread that is included in the support library.

Remember it, and enter it as a PARTNER command after the Linux kernel has launched.

Installing and configuring the debug support library

Follow the procedure below to make settings so that the debug support library can be used on the target.

(1) Installing the library into the target

Copy the two generated files into the root file system of the target Linux. Any directory is OK.

(2) Setting up the target

This library performs mapping on the process using the ld.so preload function. This will be loaded first by the preload function when an application is executed, and that causes the link function with the debugger to work. So the preload specification of the library has to be made.

The setting should be made according to one of the following methods. For details, refer to the section in the manual for ld.so.

1. Set the library as the /etc/ld.so.preload file
   Create a file named ld.so.preload under /etc, and specify the file path to the shared library (llibkmcsup.so.2.0.0) that you want to preload.
   If necessary, update the cache of the library using the ldconfig command.

2. Set the library as the environment value LD_PRELOAD
   Specify the path to the library by separating it with spaces. Only individual processes, to which environment variables are set, are targeted. To apply it to all programs on the system, rewrite /etc/profile and register it as the environment variable of the shell that launches programs.

The behavior varies depending on the Linux environment, so also refer to "Notes on Linux compatibility" (Page 71).

Column 2-4 Using functions included in the debug support file

The debug stub function (for details, refer to "Application direct link method" (Page 66) that is defined in the debug support file can be also used when building the debug support file as a library.
If the debug stub function is inserted into an application, a break automatically occurs at a point immediately after the debug stub function without using a hardware breakpoint. It can be used just once as a breakpoint in a process or thread.
You also can insert a breakpoint in a program using the _KMC_BRK_CODE_() macro that is defined in the debug support file (you need to increase the program counter value to continue the execution of the program after the break occurred).
Although you have to modify the source code of an application for debugging in both methods, we recommend you to use these techniques.

2.3.3 Application direct link method

If you directly link the application support file to an application, follow the below procedure to insert the debug support file into the debug target application.

Modifying and building an application

Create an application following the procedure below.

(1) Modifying the application source.

Insert the call of the debug stub function at the head of the main() function in the application source.

Fig. 2-5 Debug stub function

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

Specify the file name of the application as the program_name argument of the debug stub function. If the file name has been changed due to a symbolic link, insert the actual file name as a string.
PARTNER compares this argument string with the file name in the debug information that has been loaded by the debugger, and attaches to the debugger if it is the debug target.

If the debug stub function is inserted into an application, a break automatically occurs at a point immediately after _kmc_start() without using a hardware breakpoint.
The name of the debug stub function, _kmc_start() was _kmc_start_debuger() in the previous version. To maintain downward compatibility, the old function name also remains. Also note that "debuger" contains only one "g".

【Example】 Insertion at a point immediately after the main function

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

When debugging each thread in a multi-thread program (using the pthreads library), also insert the debug stub function at the head of the entry functions of each thread. Specify 0 as the program_name argument of the debug stub function that is to be inserted at the head of the entry functions of each thread.

【Example】 Insertion into a multi-thread program

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

If debugging in Kernel Mode (ADD Mode), a break occurs at the debug stub function (_kmc_start()) that was inserted after the main function, but it doesn't occur at the debug stub function that was inserted at the head of the entry function of a thread. If you set a breakpoint after the debug stub function that was inserted into the thread_body() function (after the debug stub at the head of a thread) while stopping at the main() function and execute the application, the newly created thread is automatically attached to PARTNER and a break occurs at the set breakpoint.
If you execute it without setting a breakpoint, the thread into which the debug stub function was inserted will be attached to PARTNER, but a break does not occur automatically. Set a breakpoint when the first application is attached.

When debugging each child process of a multi-process program that uses the fork() system call, insert the calls for the debug stub at the head of the main() function in the application source and the head of the child processes (the return of the fork() function in the child processes). Specify 0 as the program_name argument of the debug stub function that is to be inserted at the head of each child process.

【Example】 Insertion into a multi-process program

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

(2) Linking to the application

Link the support file (kmc-support.c) when the application is linked.kmc-support.c.

.If the "Select Target CPU type" compile error occurs in kmc-support.c, enable only the target CPU in the CPUTYPE symbol declaration in kmc-support.c, and recompile it.

Column 2-1 Tips to modify an application without changing the source file

In the method where the debug support file is directly linked to an application, you usually have to insert the debug stub function. But if you use a macro written in C, you may not have to modify the source file.
This column includes an example of how to insert the stub function by replacing the main function with a macro.
Although the #include directive is normally written in the source code to apply a C-language macro to the source file, you also can do it using a compiler function.
If you use a compiler function, you can replace symbol names without changing the source file. The following methods can be used as a method to insert a stub function.

1. Include the header file using the pre-include feature (the ?include option of gcc) of a compiler.
2. Replace a target symbol using the macro definition feature of a compiler (the -D option of gcc).
3. In conjunction with replacement of symbol names by one of the above methods, use a combinatorial technique whereby the source file that includes the function in which the replacement has been performed is linked to the application.

This column includes an example performed using the above method 1.
This method provides in the header file a macro, in which a stub function has been inserted.
The following two modifications are mainly required for debugging.

- Pre-include the header file (by modifying Makefile).
- Link the debug support file (by modifying Makefile).

In this method, you don't have to directly modify the source file, and changing the content of the Makefile description to switch between the debug and release versions is commonly performed in a normal development project.
Of course, you need to beware of reversed heisenbugs that only occur in a debug build, so you should use this after fully understanding the details of the modifications to be applied.

(1) Preparing the macro file

Fig. 2-2 shows the header file, in which a debug stub function for pre-including has already been inserted.

Fig. 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) Mechanism of stub function insertion

kmchook.h performs the following replacement.

Table 2-1 Debug Mode

Name	Replacement
main	Debug stab function in-lined main function
	Debug stab function in-lined _kmchook_pthread_create() function
	Debug stab function in-lined _kmchook_fork() function
	_main
fork	_kmchook_fork
pthread_create	_kmchook_pthread_create

Because this is a macro, it inevitably has some side effects including the names enumerated here.
Also note that replacement of the main function is tricky. In a normal definition of the main function in C language,

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

note that "main" between "int" and "(" is normally rewritten in the code in many cases. If the prototype declaration of the main function was performed, you may not get the expected result.
You should use this after fully understanding other effects and influences, for example, static variables may increase the object size per file.

(3) Compiling the application

If the application is a single-thread application, pre-include kmchook.h using the -include option.

```
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 ↓
```

If the application uses the pthread library, also define -D_KMCHOOK_WITH_PTHREAD.

```
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 ↓
```

As described above, you only need to specify kmchook.h as the compile option, and so you just have to rewrite Makefile and it will be easier for you to manage the source code of applications.
The positions of the main function and the stub function are moved to a position before the original main function (_main) after the macro was expanded.
It is necessary to perform several steps before the original main function, so the operation of the debugger becomes somewhat complicated.

2.3.4 Notes on Linux compatibility

This section describes compatibility problems between various Linux distributions that can be an impediment when handling the application debug support file.

(1) Compatibility of ld.so (the preload setting does not work)

In environments that use uClibc as the standard C library and glibc is not used, the /etc/ld.so.preload file may not work. In that case, use the environment variable LD_PRELOAD.
Environments that do not support a shared library mechanism are not supported.

(2) Compatibility of libdl (multi-context debugging cannot be performed)

The libdl library is required when building an application debug support file in the form of a shared library (libkmcsup.so).
When a shared library form (libkmcsup.so) is used, the pthread_create() function and fork() with the debug hook function are created in the debug support file, and from those functions the "original" pthread_create() function and fork() in the libpthread and libc libraries are called. At that time, the dlsym interface of the libdl library is used to search these "original functions".
Some Linux distributions have problems with the behavior of functions that are provided by the libdl library (Linux distributions that use an ld.so that is different from the ld.so included with glibc usually fall under this case).
The following is a practical explanation of situations where bugs occur and countermeasures in these situations.
Firstly, "a process that searches for the original functions" included in the debug support file is described as shown below.

【Example】 Searching for the "original" fork()

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

In Linux distributions with no compatible problem, when this was performed, the return value of the dlsym() function is the address of the "original" fork() function and this address is located in libc. Linux distributions with compatibility problems do not return the correct value. For example, if the address of the fork() function that was defined in the debug support file is returned, it causes an infinite recursive call and a stack overflow. If a totally unrelated address is returned, unpredictable behavior will result.
According to the knowledge currently available, the behavior of the RTLD_NEXT specification of the dlsym() function causes this incompatibility. To avoid this problem, modify the code of the debug support file so that the library in which "the original fork() function" is stored can be specified using the dlopen() function without using the RTLD_NEXT specification.
In other words, you need to check in advance which library contains the function that you want to search for.
The code after the avoidance measure was performed is as shown below (replace libYourLibC according to your environment).

【Example】 fork() search process after the avoidance measure was taken

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

The dlopen() function is usually closed with its counterpart, the dlclose() function, but there seem to be cases where bugs occur if the dlclose() function is used. In that case, comment out the call for the dlclose() function.

2.4 Launching the debug environment

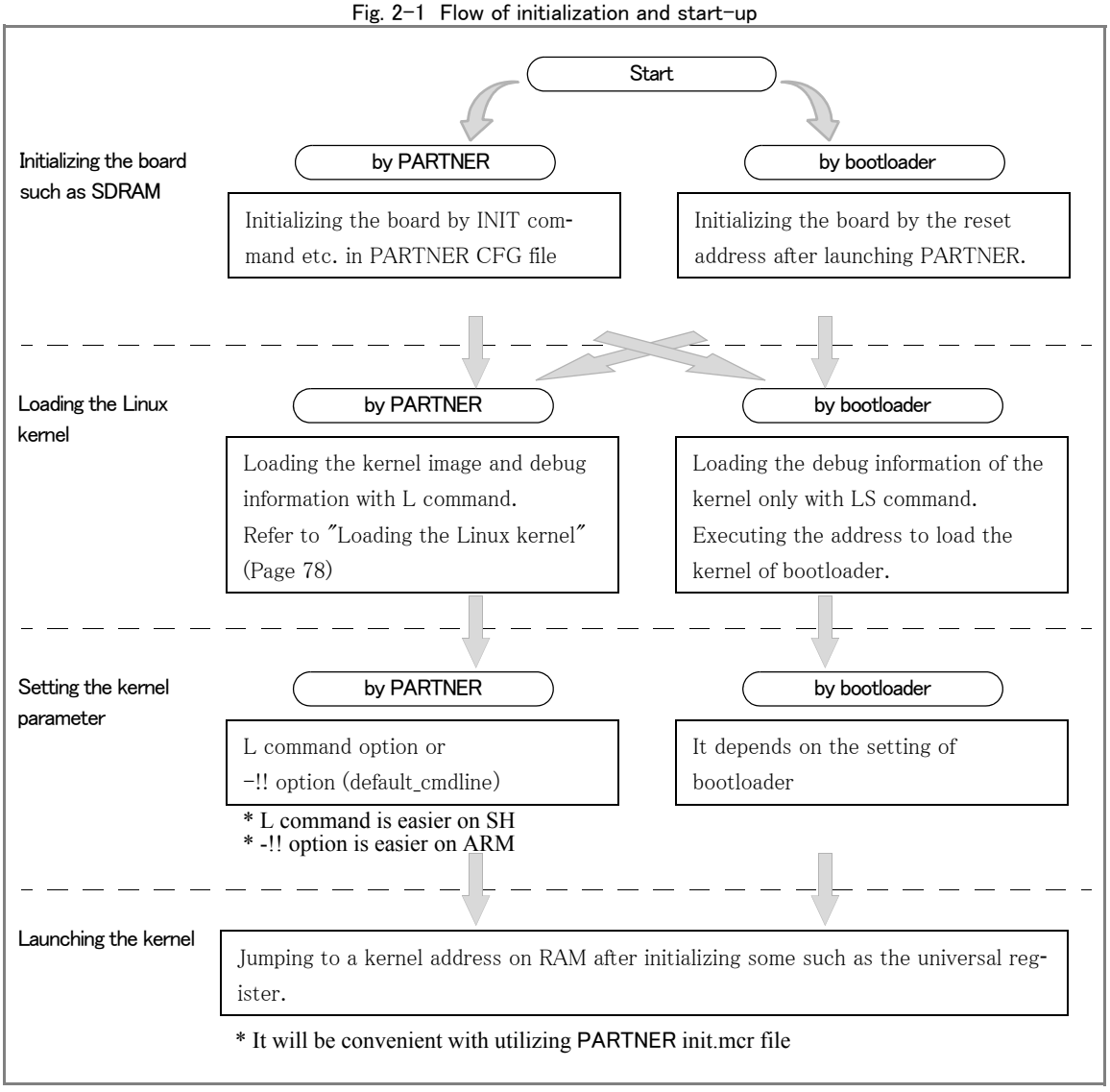
This section describes the procedure for performing debugging in Kernel Mode. It assumes that modification covered in ""2.2 Modifying and configuring the Linux kernel source" (Page 54) has already been made on the kernel. In the Linux kernel setting menu (refer to "Kernel configuration" (Page 57)), make settings necessary for PARTNER, and then, create the Linux kernel (vmlinux) after the configuration settings have been made.

In environments where the kernel source cannot be modified, manually modify akefile so that debug information is added to the kernel object. Necessary Linux kernel configuration settings depend on the debug targets, so they have nothing to do with the launch mode of PARTNER.

2.4.1 Types of target initialization and launch methods

In embedded Linux development environments, target initialization and start-up can be performed in two ways, namely, a method that uses the JTAG debugger (PARTNER) or a method that uses the boot-loader on the target board. The JTAG debugger is not used with the final product, so it can be changed during development.
This section describes a method that uses PARTNER. So, if you want to use the boot-loader, confirm in Figure 2-1 how the procedure and configuration settings would change.

Fig. 2-1 Flow of initialization and start-up



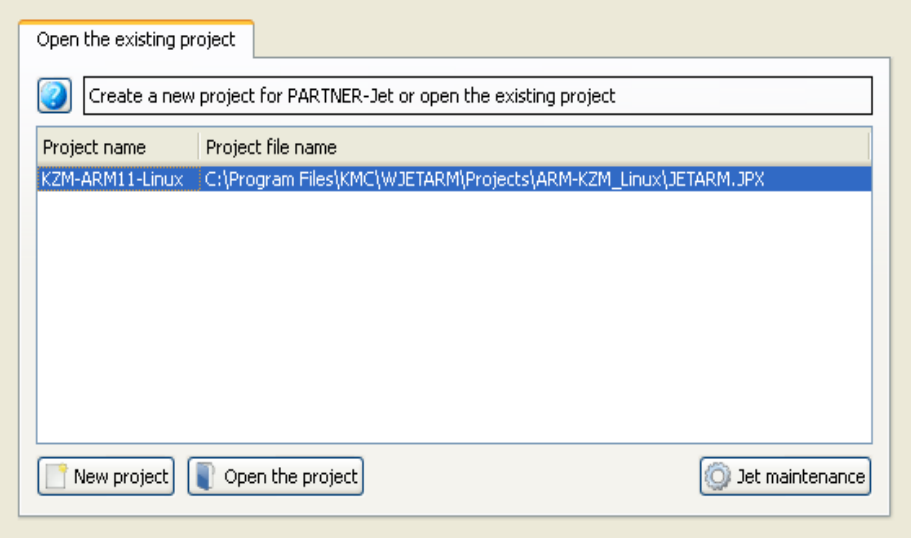
2.4.2 Configuring and launching PARTNER

Follow the procedure below to launch PARTNER in Kernel Mode.

(1) Starting a new project or selecting an existing project

Launch the environment setting program of PARTNER (JETSET) to start a new project or open an existing one.

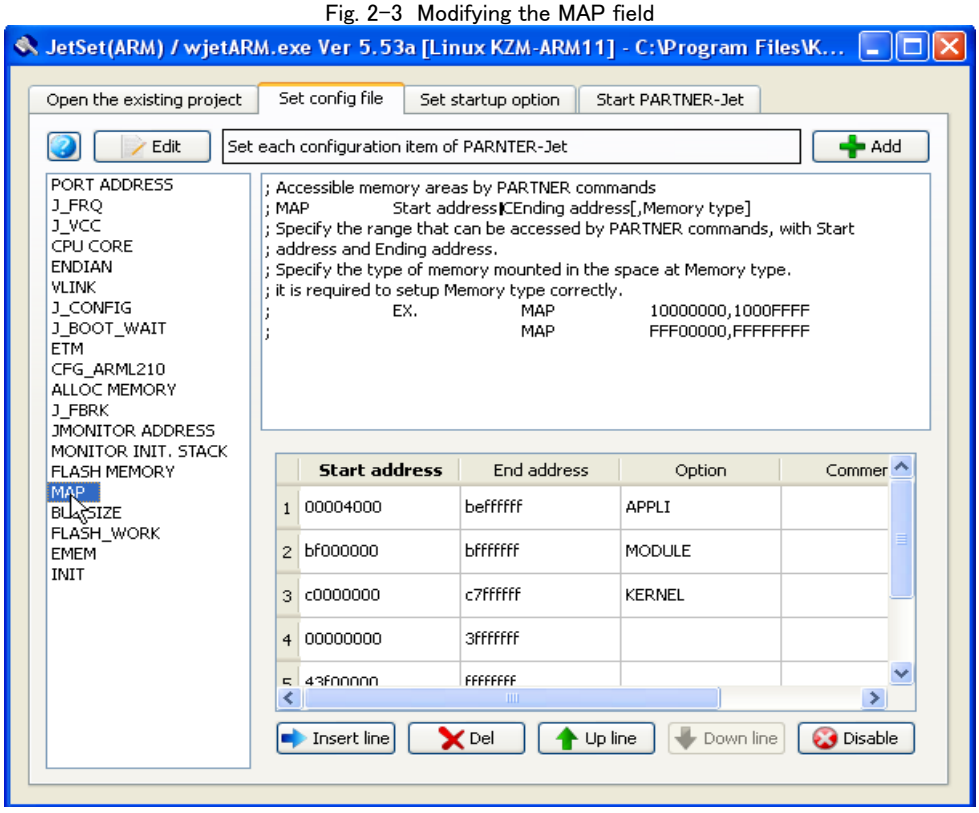
Fig. 2-2 Opening a PARTNER project



(2) Setting the CFG file

Specify map information for Linux in the MAP field. For information on addresses and attributes to be set, refer to "MAP field" (Page 141).

Fig. 2-3 Modifying the MAP field



(3) Setting the launch option

Press the [Expand>>] button to specify the expansion option for Linux debugging.
The following settings must be made when debugging the kernel.

● Debug information buffer size (-B option)

Specify around 100,000 as the buffer size.
If an error message "debug information area is full (refer to the -B option for the time when launching)" is displayed, expand the buffer size.

● Debug information type (-XGX option)

Select "GNU C (Linux etc.)".

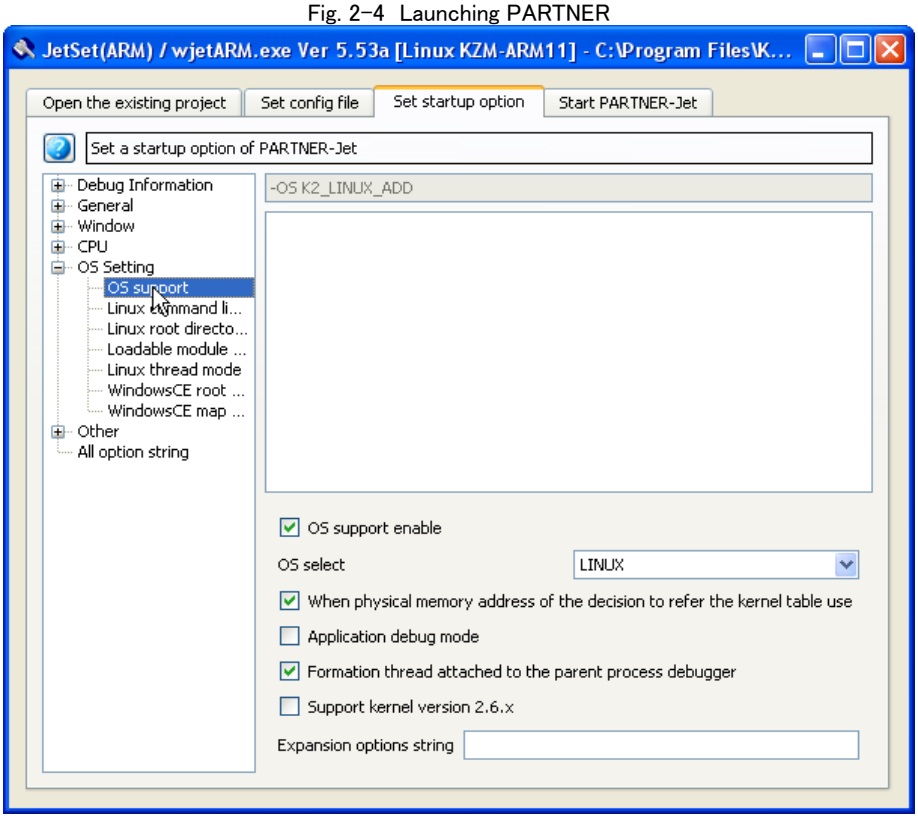
● Debug information path conversion (-XGX option)

Specify this when the path used to build the kernel (vmlinux) and the installation path in Samba are different.
For example, if the kernel is built at /home/foo/work/linux and /home/foo/work is installed on the Z: drive, specify -XGX/home/foo/work/linux/,z:¥linux¥.

● OS debug mode specification (-OS option)

Select the option for "Kernel (NON_ADD) Mode", "Kernel ADD Mode", "Application (NON_ADD) Mode", and "Application ADD Mode". We recommend "K2_LINUX_ADD" or "K2_LINUX_ADD_V26".
For details on each option, refer to " -OS option" (Page 147) and ""Appendix F Selecting the operation mode of PARTNER" (Page 255).

Fig. 2-4 Launching PARTNER



(4) Launching a PARTNER window

Clicking the Launch button on JETSET launches PARTNER.

Fig. 2-5 Launching PARTNER

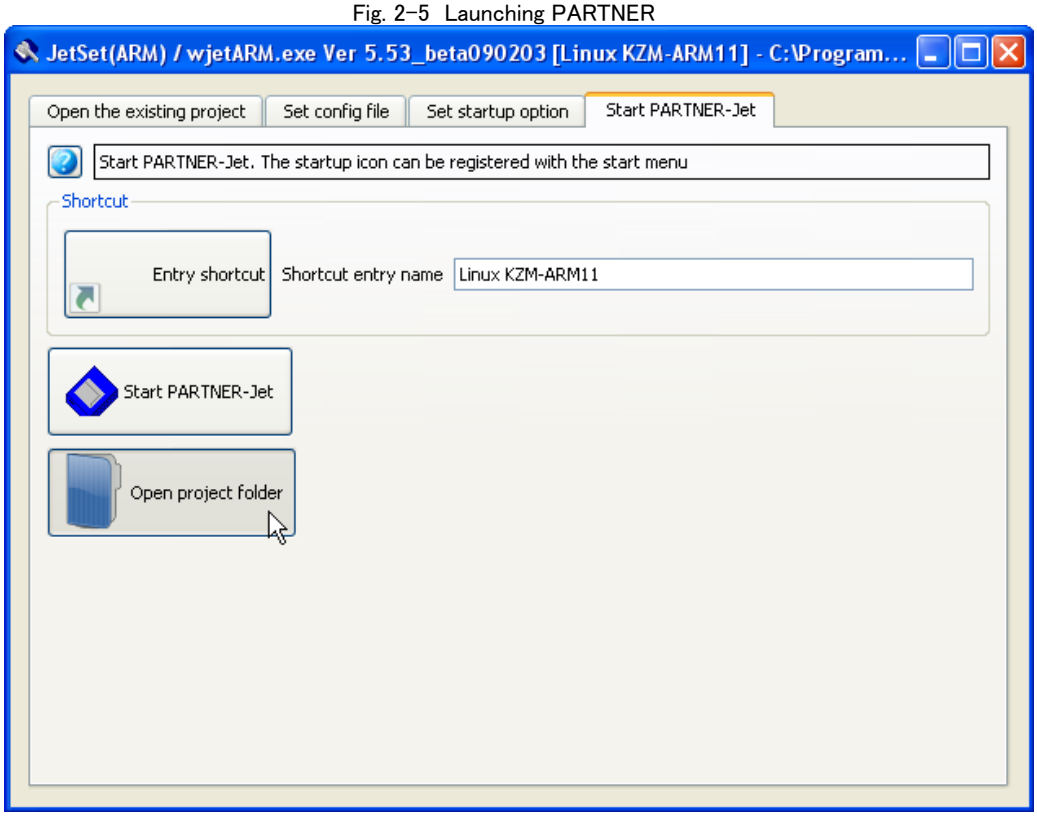
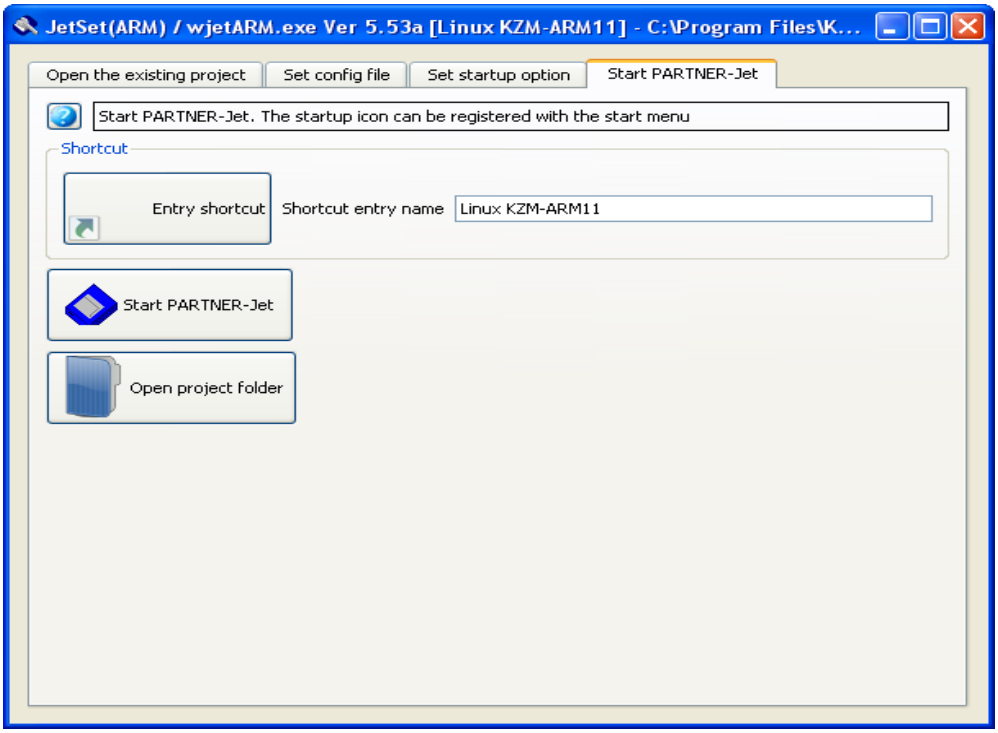


Fig. 2-6 Launching screen of PARTNER

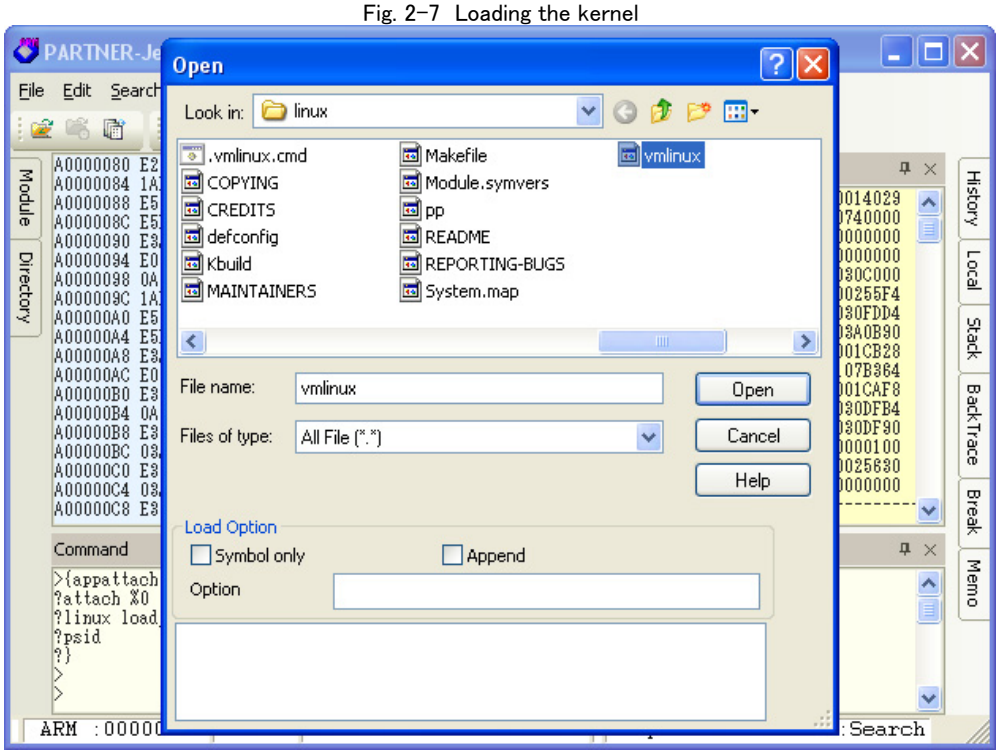


2.4.3 Loading the Linux kernel

Load the kernel created in "2.4.2 Configuring and launching PARTNER" (Page 75) (vmlinux) into the target memory.

PT>l vmlinux ↓

Fig. 2-7 Loading the kernel



When loading the binary file of the HEX file that was created from vmlinux, follow the procedure below to load the kernel.

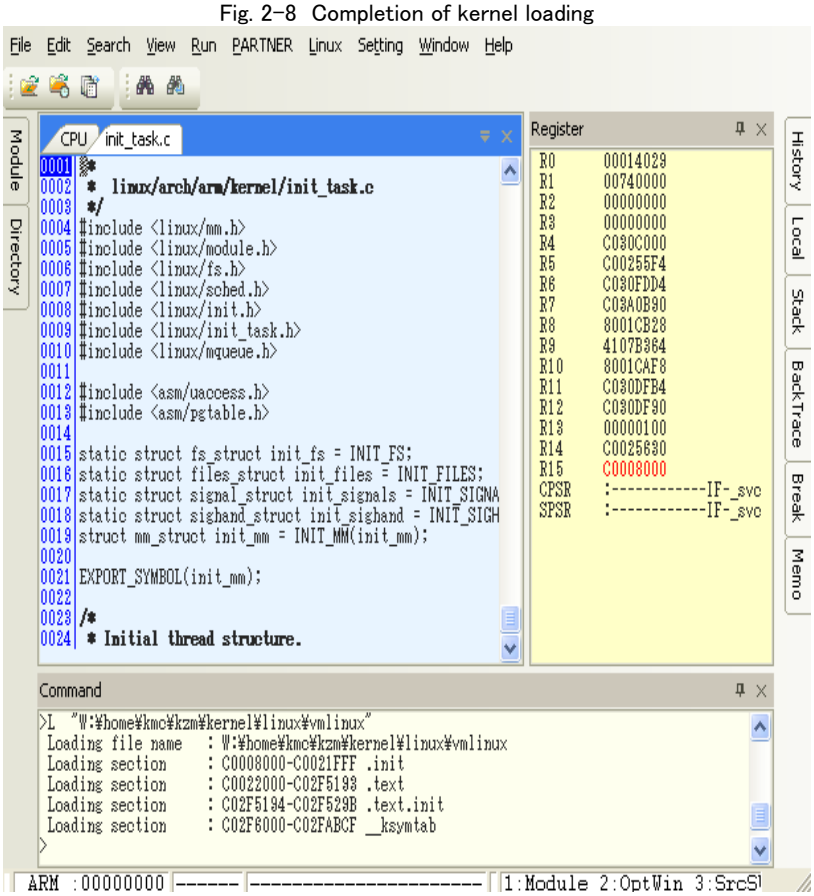
Enter the following command in a PARTNER command window.

[Example] Load the HEX file using the RD command and load the debug information using the LS command.

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

If debug information output is specified when compiling, debug information is also loaded when the kernel is loaded. PARTNER remembers in the command history the load file names and the locations of files that have been loaded, so the subsequent operations can be simplified.
In systems where the Linux kernel is transferred from ROM to RAM, use the LS command instead of the L command to load only the debug information into PARTNER. .

Fig. 2-8 Completion of kernel loading



If loading is normally completed, the Linux source code is displayed in the code window of PARTNER. If nothing is displayed in the code window, the path conversion setting that converts Linux PC directory information to a directory that can be referred from the Windows PC may not be correct (refer to " -XGX option" (Page 150)).

Once the source code of Linux is properly displayed in the code window of PARTNER, PARTNER is in the state that allows full source debugging on the Linux kernel.

In systems where the Linux kernel is transferred from ROM to RAM, you cannot set software breakpoints before execution. Breakpoints can be set after the kernel has been transferred. To set breakpoints before the kernel is transferred, set hardware breakpoints.

Colum 2-9 Using the macro function of PARTNER

PARTNER has the macro command function. With this function, you can combine multiple commands and create a new command using macro control commands. The following commands are provided as macro control commands. For details, refer to PARTNER's help.

- {(definition of macro a command)
- DO{..}WHILE(DO-WHILE macro)
- FOR{..}(FOR macro)
- WHILE{..}(WHILE macro)
- REPEAT{..}(REPEAT macro)
- BREAK(gets out of a macro)
- LALL(specifies macro display output)
- SALL(Specifies macro display prevention)
- MLIST(Displays a macro)
- KILL(Deletes a macro)
- IF{..}(IF macro)
- KEYIN(Specifies key input)

If you create a text file named "init.mcr" in the project folder and include a macro in it, this macro will be loaded when a PARTNER window is launched.

You can simplify the kernel load procedure or switch between procedures using this function.

For example, if you write a macro as shown in Fig. 2-10, executing "PT>load_nor" as a PARTNER command loads the kernel from the Windows PC using PARTNER, and the root file system uses the NOR flash. If you execute "PT>load_ide", the root file system on the HDD is used.

Fig. 2-10 Macro creation example

```
{load_nor          Macro name
l z:¥kzm¥linux¥vmlinux noinitrd console=ttymxc0 root=/dev/mtdblock2
rootfstype=jffs2 init=linuxrc ip=none ,/offs=0xc0000000
                                  Setting the kernel parameter with L command option
_r0=0
_r1=_956          Setting content of registers/ program counter
_r15=0x80008000
br start_kernel,ex      Setting a breakpoint
}

{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 Executing the Linux kernel

When the kernel has been properly loaded, enter the G command or press the Execute button to execute the loaded vmlinux.

```
PT>g ↓
```

If the Windows PC and the target board are properly connected and the terminal software is normally running, the bootup message is displayed.

If the bootup message does not correctly appear or the system hangs up, re-check each configuration setting.

Fig. 2-11 Bootup message for the Linux kernel

```
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:
```

When reloading the Linux kernel, make sure you execute the INIT command from the command window of PARTNER.
Unless you perform initialization using the INIT command, Linux cannot launch even if you reloaded it.

Chapter 3 Debug procedure reference

This chapter describes the procedure for debugging using PARTNER for Linux.

3.1 Debugging the kernel

This section describes how to debug the Linux kernel.

3.1.1 Configuration required for debugging

Table 3-1 Configuration settings for kernel debugging

Setting Condition / Debug Target	Kernel Menu						-OS Option			
	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	Kernel mode	ADD mode of kernel	Application mode	Add mode of application
Kernel	◎						—— *1			

◎ : Required, ○ : Recommended, △ : Not recommended, × : Not Permitted, Empty : Either is OK

*1 The -OS option is not required for kernel debugging, but we recommend that you specify one.

The only requirement for kernel debugging is that the kernel is compiled with debug information. Even in environments that do not allow kernel source modification, it is necessary to create debug information for the kernel.

Colum 3-1 How to create kernel debug information

The Debug Information Type menu of the KMC patch is provided to facilitate adding debug information and making type settings. To do the same without using this menu, you also can use the following methods.
You can build the kernel with debug information using any of the following methods.。

(1) Specify it in the Kernel Hacking menu in the kernel configuration.

(2) Rewrite Makefile and add a line (e.g. "CFLAGS+=-g").

(3) Specify CFLAGS as the command argument, when executing the make command.

3.1.2 Debugging when initializing the Linux Kernel

PYou have to be careful about the system configuration when debugging the Linux booting process with PARTNER.

(1) When executing the kernel after loading it to RAM using PARTNER

Debugging can be performed when the kernel has been loaded (for the procedure, refer to "Loading the Linux kernel" (Page 78)).
There are, however, some things you have to pay attention to depending on the system configuration.

● When using a self-extracting type

There are some systems whose kernel is compressed to save the storage capacity used for storing the kernel image. If the kernel has not yet been decompressed when it is transferred to RAM in your system, you cannot set breakpoints for kernel processing using PARTNER at the point when the kernel has just been loaded. Even if it is a single file, actually it consists of two programs, namely the self-extracting routing and the Linux kernel, and so it should not have debug information.
In this case, load the debug information for the Linux kernel using the LS command when the self-extraction routine has completed the extraction process.

● When the physical address (PA) and virtual address (VA) are different from each other

Some systems link the Linux kernel in the virtual address space (mostly the Linux kernel for ARM CPUs falls under this case).
In such systems, TLB initialization is performed and the MMU is enabled at a very early stage of execution of the kernel in RAM. Since access to the virtual address (VA) is not allowed until this initialization process is completed, you cannot set breakpoints, refer to variables or perform disassembling in the VA.
If you set an executable hardware breakpoint at a position after the VA becomes available (e.g. start_kernel symbol), you can use software breakpoints after it has stopped at the set hardware breakpoint.

[Example]

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

(2) When a boot-loader loads and executes the kernel.

A boot-loader is mainly used to transfer the kernel image from ROM to RAM. In that case, if the debug information of the kernel has already been loaded into PARTNER, you cannot use software breakpoints before transferring the kernel. To set breakpoints before the kernel is transferred, use hardware breakpoints.

(3) When the kernel is executed in ROM

In a system where part of the kernel program or the whole kernel program is located in ROM (it is also called an XIP kernel), you cannot set software breakpoints to the program code that is located in ROM. Set hardware breakpoints instead.

Notes on software breakpoints

A software breakpoint is a technology that stops a program by rewriting with a debugger the code of a program that is located in RAM.

Because of this mechanism, there are limitations such as "it cannot be used before program transfer" and "it cannot be used for programs in ROM".

3.1.3 Kernel debugging after system boot

There are no limitations on kernel debugging after the kernel has been executed (for the procedure, refer to "Executing the Linux kernel" (Page 81)).

You can press the ESC key during operation on PARTNER to stop the CPU, and set breakpoints at any points within the kernel.

3.2 Debugging a loadable module (method 1)

The characteristic of Linux loadable modules (device drivers) viewed from PARTNER is that they are ″re-locatable objects that function in the Linux kernel space″.
The address to which a loadable module is loaded when it is linked (created) does not determine the address at which the loadable module will be located.
The address at which the loadable module is located will not be determined until the Linux kernel has been loaded.
For that reason, in order to debug a loadable module, PARTNER must know where the Linux kernel placed the installed modules (i.e. information about the actual addresses).
PARTNER can automatically obtain the address of a module that the Linux has loaded, and debug it.

The procedure described in this section can be used for both 2.4.x and 2.6.x kernels. However if the 2.6.x kernel is used, we recommend you to use ″Debugging a loadable module (method 2) (Page 93)″ that does not require modification of the source code of loadable modules.

3.2.1 Configuration required for debuggin

Table 3-2 Configuration settings for loadable module debugging (method 1)

Setting condition / Debug target	Kernel menu						-OS option			
	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	Kernel mode	Kernel ADD mode	Application mode	Application ADD mode
Loadable module	◎	◎	—	◎			○	○	△ *1	

◎ : Required, ○ : Recommended, △ : Not recommended, × : Not permitted, Empty: Either is OK

*1 It is not impossible but not recommended. The kernel CPU stops when performing debugging or creating a break within a loadable module.

If you have not yet performed "2.2 Modifying and configuring the Linux kernel source" (Page 54), you cannot perform automatic attachment of loadable modules described below. If you want to use a kernel that is not modified, refer to "5.2 Manually debugging a loadable module" (Page 192).

3.2.2 Debugging Procedure

This section describes the debugging method of loadable modules in the following order, using the case where a RAM disk (rd.o) is used as an example.

(1) "Modifying the source of a loadable module (Page 90)"

(2) "Building a loadable module (Page 90)"

(3) "Preparing for debugging (Page 90)"

(4) "Installing a loadable module (Page 91)"

(5) "Breaking using PARTNER (Page 91)"

(1) Modifying the source of a loadable module

In the source of a debug-target loadable module, insert the following line into the head of the source in which the module_init() function is defined.

```
#define       __KMC_MODULE_DEBUG
```

【Example】 Insertion the declaration for debugging

```
+#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.
 *
```

If there is no module_exit() function, define a dummy module_exit() function.

If you created an original loadable module, define the full path to that loadable module.

```
#define __KMC_MODULE_NAME "Full path to the Module"
```

【Example】 Insertion of the path to the loadable module

```
+#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.
 *
```

As for a loadable module in the Linux kernel source tree, the full path to the loadable module is automatically resolved.
If you failed, define it using the"-SK option (Page 153)" and _KMC_MODULE_NAME, so that the full path to a loadable module can be resolved.

(2) Building a loadable module

Create a loadable module that you want to debug.
When doing so, do not forget to add debug information that is the same as that of the kernel. Debug information specified in "(1) Modifying the source of a loadable module" (Page 90) is added to a loadable module in the Linux kernel source tree.

```
LINUX86>make modules ↓
```

(3) Preparing for debugging

Launch PARTNER and Linux and put them into an operable state (refer to "Launching the debug environment (Page 73)".

(4) Installing a loadable module

Once it is normally launched, insert the debug-target loadable module into the kernel in the target system. Use the insmod command as a super user (root) to install a loadable module.

```
TGT>insmod rd.o ↓
```

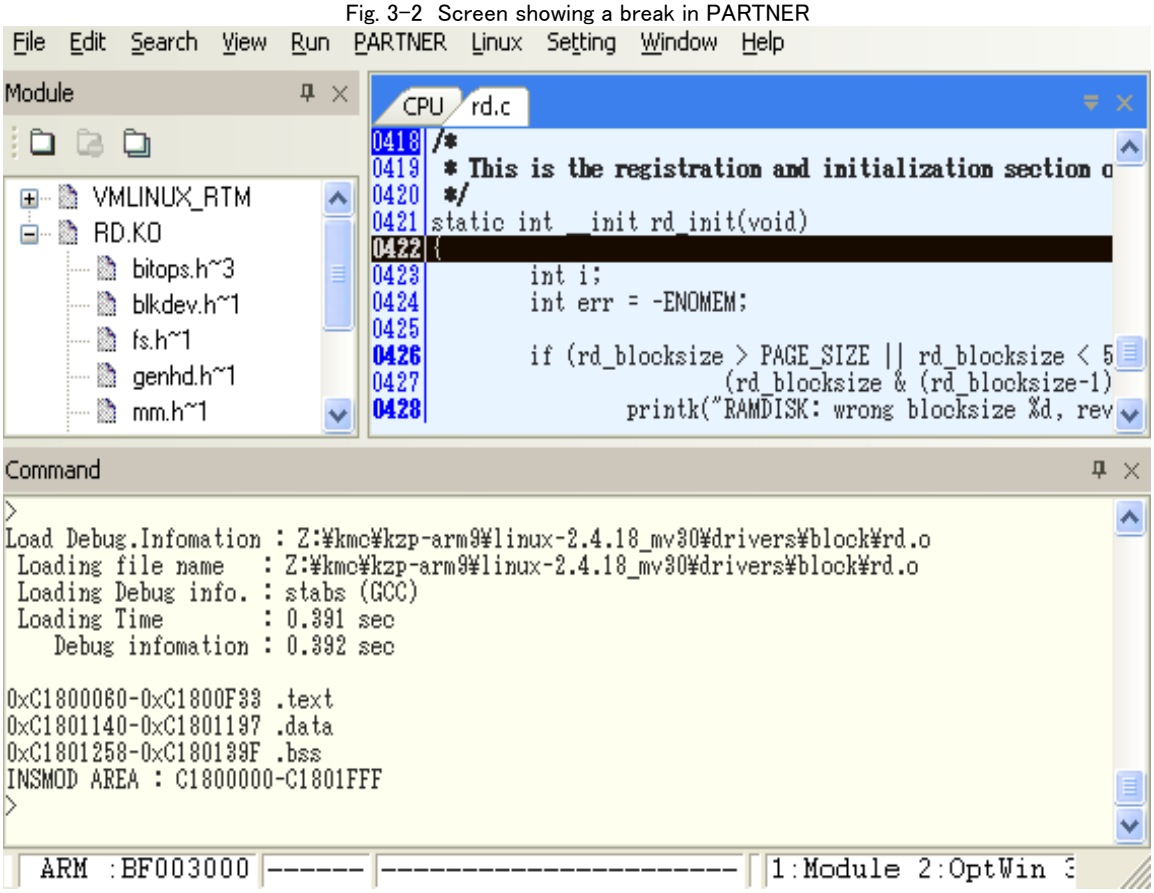
Fig. 3-1 Installing a loadable module

```
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) Breaking using PARTNER

When the debug-target loadable module has been installed in the target system, PARTNER automatically load the debug information and breaks at the module_init() function. At that time, it also obtains the placement information of the loadable module automatically, and thereafter you can refer to the memory in the loadable module area and set breakpoints.

Fig. 3-2 Screen showing a break in PARTNER



If an error message saying "There is no specified file" is displayed, it means that it failed to resolve the file path to the installed loadable module, that or the loadable module file does not exist. If it failed to resolve the path, use the"-SK option (Page 153)" to resolve the problem.

If the source code of the loadable module does not appear in the code window of PARTNER, the source path in the debug information may differ from the source path used by PARTNER to access the module. Use the "-XGX option (Page 150)" to convert the path stored in the debug information.

If you unload (rmmod) the loadable module that you are debugging and install (insmod) the same loadable module again, the originally loaded debug information of the Linux kernel will be lost. If you need the debug information for the Linux kernel, reload the debug information of the Linux kernel using the lsa command.

3.3 Debugging a loadable module (method 2)

The format of a loadable module for a Linux 2.6.x kernel is different from that of a 2.4.x kernel, and it is .ko format. If a 2.6.x kernel is used, PARTNER can start debugging at the point where insmod is performed, without modifying the source code of a loadable module.

The procedure described in this section can be used only for a 2.6.x kernel. It cannot be used for a 2.4.x kernel.
For 2.6.x kernels, you can also use the method described in "Debugging a loadable module (method 1)(page 81)", but we recommend this method.

3.3.1 Configuration required for debugging

Table 3-1 Configuration settings for loadable module debugging (method 2)

Setting condition / Debug target	Kernel menu						-OS option			
	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	Kernel mode	Kernel ADD mode	Application mode	Application ADD mode
Loadable module	◎	◎	◎	—			○	○	△ *1	

◎ : Required, ○ : Recommended, △ : Not recommended, × : Not permitted, Empty: Either is OK

*1 It is not impossible but not recommended. The kernel CPU stops when performing debugging or creating a break within a loadable module.

If you have not yet performed "2.2 Modifying and configuring the Linux kernel source (Page 54)", you cannot perform automatic attachment of loadable modules described here. If you want to use a kernel that is not modified, refer to "5.2 Manually debugging a loadable module (Page 192)".

3.3.2 Debugging Procedure

This section describes the debugging method of loadable modules in the following order, using the case where a RAM disk (rd.ko) is used as an example.

(1) "Building a loadable module (Page 95)"

(2) "Preparing for debugging (Page 95)"

(3) "Installing a loadable module (Page 97)"

(4) "Breaking using PARTNER (Page 97)"

Debugging a loadable module (method 2)

(1) Building a loadable module

A loadable module for 2.6.x kernels is compiled by Makefile in the source tree. Confirm the following two points.

1. Whether the kernel compile setting has debug options
   If the patch for PARTNER is applied to the kernel, you can set it by choosing [PARTNER Debugging] ->[Debug Information Type] from the configuration menu. You also can directly edit Makefile to add a flag for the compiler in CFLAGS.

2. Whether it can be compiled as a loadable module
   If the debug-target device driver is included in the kernel tree, confirm whether or not the configuration allows it to be compiled as a loadable module.
   If a RAM disk (rd.ko) is used, confirm whether the item in [Device Drivers] -> [Block Devices] -> [RAM Disk Support] is set to <M>.

Create a loadable module that you want to debug.

```
LINUX86>make modules ↓
```

Install it into the target file system. For example, if the directory tree of the target file system is located at ${TARGET_ROOT}, enter the following command.

```
LINUX86>make INSTALL_MOD_PATH=${TARGET_ROOT}/ modules_install ↓
(It will be installed under /lib/modules/2.6.16.XXX/ in the target.)
```

(2) Preparing for debugging

Launch PARTNER and Linux and put them into an operable state (refer to "Launching the debug environment (Page 73)". Register the debug-target module using the "LINUX command (Page 164)". The maximum number of modules that can be registered is eight (you can change this as it is defined in the KMC patch in the Linux kernel).

```
PT>linux module z:¥kmc¥kzm-arm11¥root¥lib¥modules¥2.6.16.19-kzm¥kernel¥drivers¥block¥rd.ko ↓
(Registers the module to be debugged.)
```

It is possible to perform the "linux module <path name>" operation at any time before insmod is executed for the loadable module, but the Linux kernel must have already been enabled (after the start_kernel symbol).
You can confirm the registration of the debug-target module using the "linux module" command, and delete it using the "linux module clr" command.
For details, refer to "LINUX command (Page 164)".

Confirm whether the registration has been completed.

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

(3) Installing a loadable module

Once it is normally launched, insert the debug-target loadable module into the kernel in the target system. Use the insmod command as a super user (root) to install a loadable module.

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

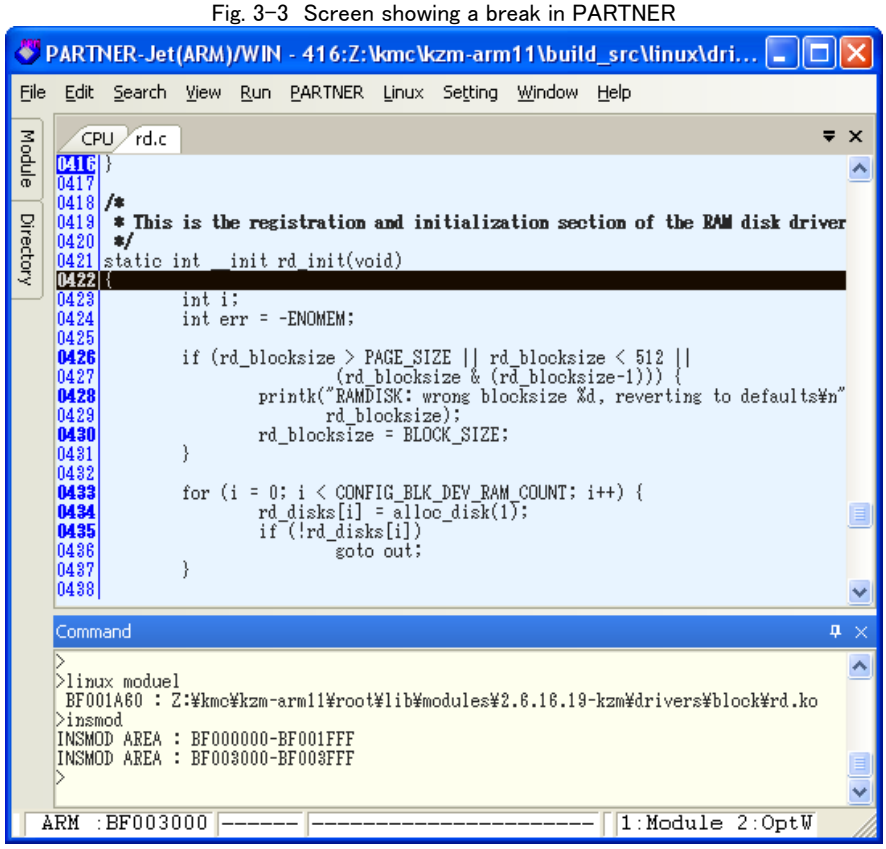
(4) Breaking using PARTNER

When the debug-target loadable module has been installed in the target system, PARTNER automatically loads the debug information and breaks at the module_init() function. At that time, it also obtains the placement information of the loadable module automatically, and thereafter you can refer to the memory in the loadable module area and set breakpoints.

You can confirm the location of the module that has been inserted in the kernel using the "INSMOD command (Page 166)".

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

Fig. 3-3 Screen showing a break in PARTNER



If an error message saying "There is no specified file" is displayed, it means that it failed to resolve the file path to the installed loadable module, or that the loadable module file does not exist. If it failed to resolve the path, use the -SK option (refer to Page 153) to resolve the problem.

If the source code of the loadable module does not appear in the code window of PARTNER, the source path in the debug information may differ from the source path used by PARTNER to access the module. Use the -XGX option (refer to Page 150) to convert the path stored in the debug information.

If you unload (rmmod) the loadable module that you are debugging and install (insmod) the same loadable module again, the originally loaded debug information of the Linux kernel will be lost. If you need the debug information for the Linux kernel, reload the debug information of the Linux kernel using the lsa command.

3.4 Debugging an application

For application debugging, PARTNER provides two debug modes, namely the application mode and the kernel mode (refer to "-OS option (Page 147)"). This section describes the procedure to debug a single-process application in Kernel Mode.

In kernel mode debugging, when an application being debugged stops, the kernel and all other application processes also stop and you are then able to refer to the kernel resources of the time when it stopped.

3.4.1 Configuration required for debugging

Table 3-2 Configuration settings for application debugging

Setting condition / Debug target	Kernel menu						-OS option			
	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	Kernel mode	Kernel ADD mode	Application mode	Application ADD mode
Application	◎	◎					○	○	— *1	

◎ : Required, ○ : Recommended, △ : Not recommended, × : Not permitted, Empty: Either is OK

*1 For information on the debug method in Application Mode, refer to "5.1 Debugging in Application Mode (Page 182)".

If you have not yet performed "2.2 Modifying and configuring the Linux kernel source (Page 54)", you cannot perform the application debugging described here. If you want to use a kernel that is not modified, refer to "5.3 Manually debugging an application (Page 200)".

3.4.2 Debugging Procedure

This section describes the procedure to debug an application (process) in Kernel Mode, using the case where a sample (sample) is used as an example.

(1) "Creating an application (Page 101)"
(2) "Preparing for debugging (Page 101)"
(3) "Opening a PARTNER window for an application (Page 101)"
(4) "Loading application debug information (Page 102)"
(5) "Setting breakpoints (Page 103)"
(6) "Executing an application (Page 104)"
(7) "Breaking using PARTNER (Page 104)"
(8) "Attaching and confirming an application (Page 105)"

(1) Creating an application

Create the debug-target application with debug information.

You don't have to modify any part of the source code of the application.

[Example]

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

If you use the debug support file in "Preload library method (Page 63)", **do not link the debug support library (libkmcsup.so) to the application.** (You can confirm the linked shared library using the ldd command in the target.)

(2) Preparing for debugging

Refer to "Launching the debug environment (Page 73)" and enable PARTNER to perform Linux debugging in Kernel Mode.

(3) Opening a PARTNER window for an application

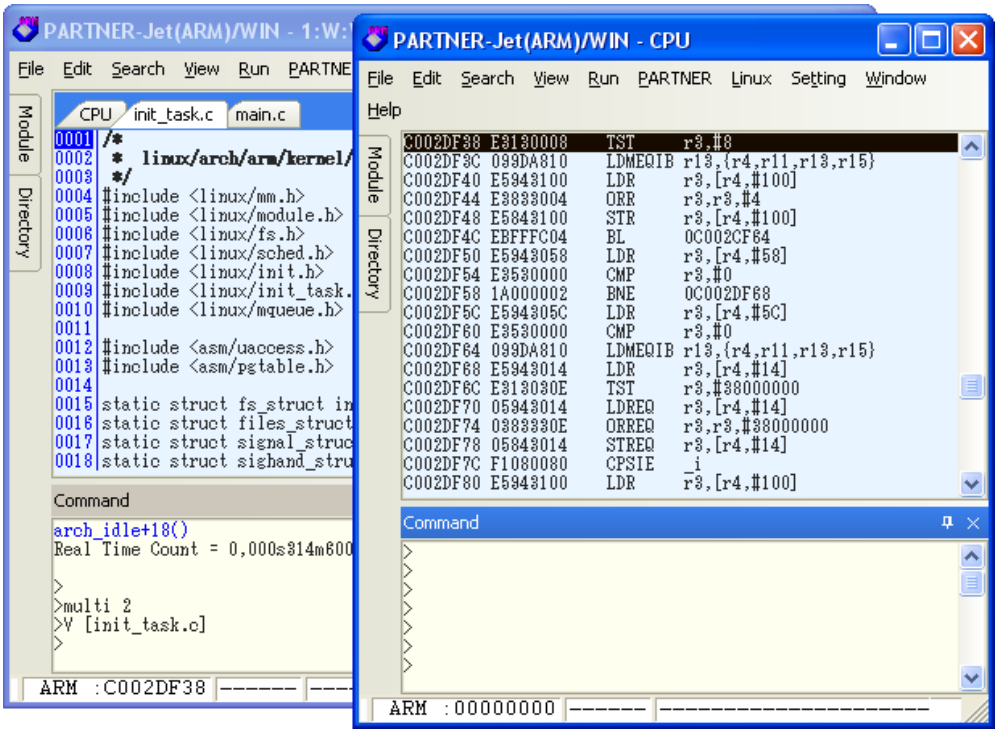
If you want to debug the Linux kernel and an application using separate PARTNER windows, open multiple PARTNER windows using the"MULTI command (Page 173)". To debug the sample program (sample), open two PARTNER windows, one for the kernel and the other for the application.

If the target is running, first press the ESC key to stop the CPU and enter the following command.

```
PT>multi 2 ↓
```

If you are debugging an application that has only one process, using the PARTNER window into which the kernel was loaded, you don't have to open the second PARTNER window.

Fig. 3-4 Opening multiple PARTNER windows



Window information (such as window arrangement) and the command history of PARTNER windows opened using the "MULTI command (Page 173)" or the "-MULTI option (Page 155)" are stored in a newly-created project file (e.g. JETARM_1.JPX) and will be used at the next launch.

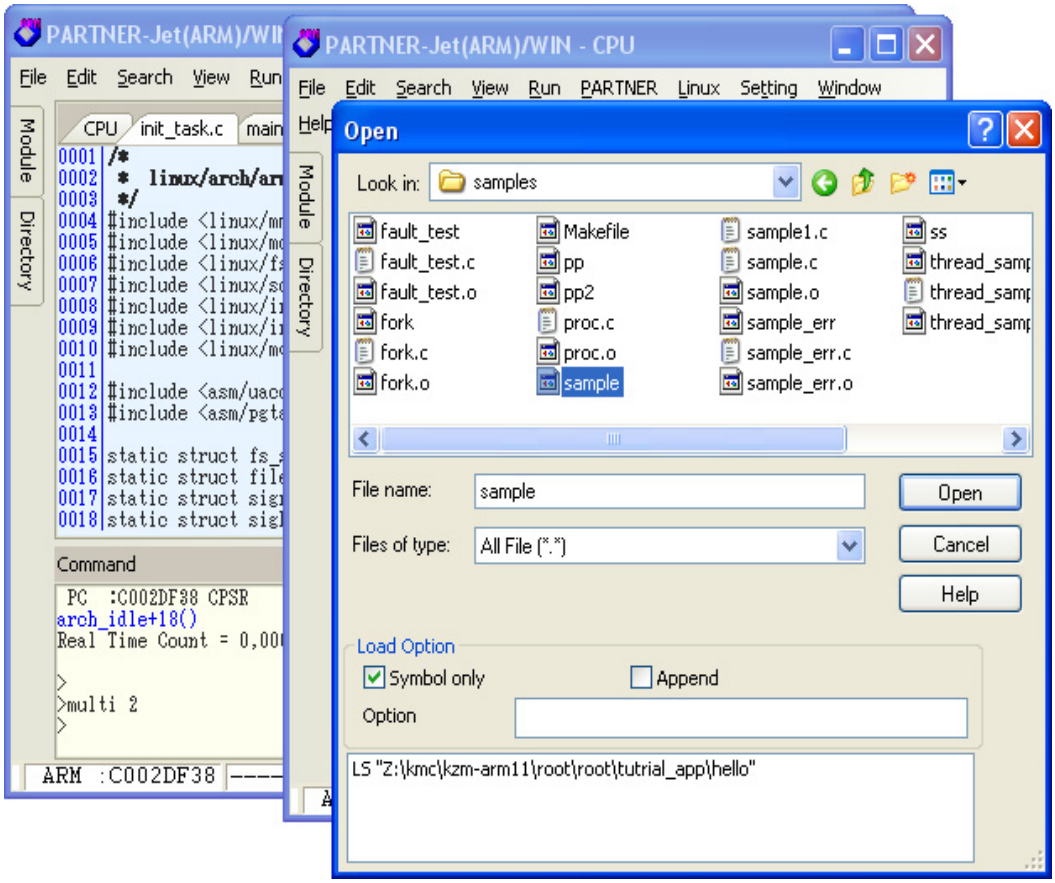
(4) Loading application debug information

Load the debug information of the created application into PARTNER.

Be sure to check Symbol Only when loading it. Also, check Append when loading multiple pieces of debug information into one PARTNER window.

PT>ls sample ↓
PT>lsa sample ↓

Fig. 3-5 Loading application debug information



PARTNER remembers in the dialog and command history the names and locations of files that have been loaded, so the subsequent operations can be simplified.

(5) Setting breakpoints

Set an executable hardware breakpoint at the main() function of the application.

```
PT>br main.ex ↓
```

If the "Verify error" message appears when setting the breakpoint, the maximum allowed number of breakpoints has already been set (the number of breakpoints that can be set depends on the CPU). Set a breakpoint again at the entry of the application after deleting one of the already-set breakpoints.

(6) Executing an application

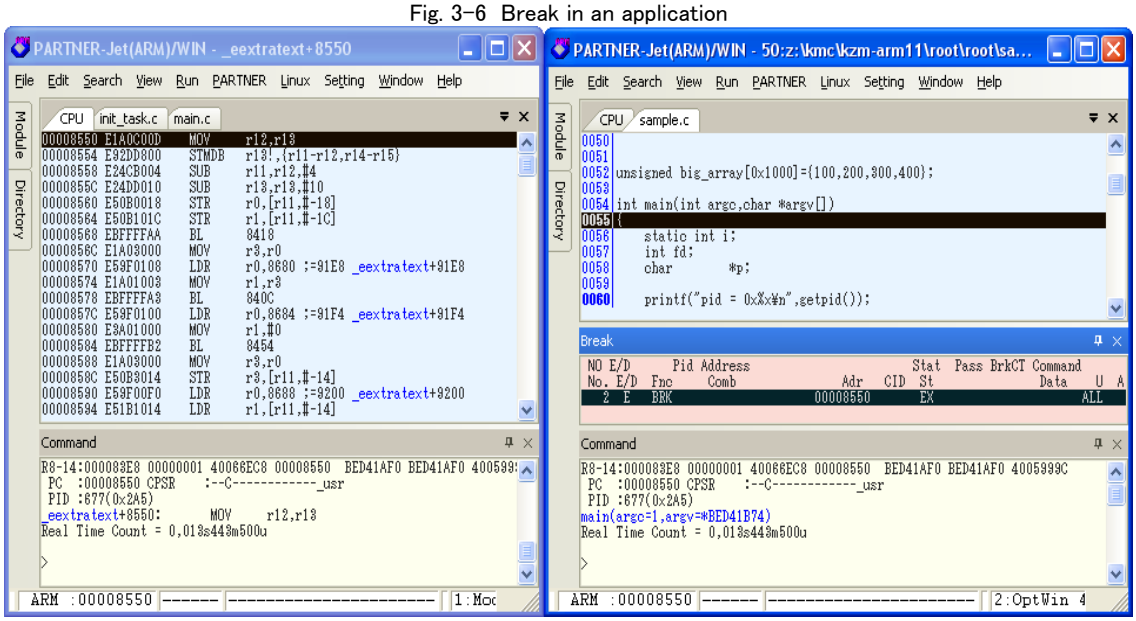
Execute the debug-target application in the target system.

```
TGT>./sample ↓
```

(7) Breaking using PARTNER

If the debug-target application is executed in the target system, PARTNER breaks at the position at which the breakpoint was set.

Fig. 3-6 Break in an application



PARTNER sets a hardware break at an address that was obtained from the debug (symbol) information. When debugging an application, this address may also be used by another program, so that another program may cause a breakIng. That case, keep the program running (by pressing the F5 key) until a break occurs in the target application. .

If PARTNER does not break properly, the kernel configuration may not be correct. Makes sure that the OS debug mode, which was specified in "2.2.5 Kernel configuration (Page 57)" and "2.4.2 Configuring and launching PARTNER (Page 75)", is in Application Mode.

Also confirm that the support library (llibkmcsup.so.2.0.0) has been preloaded.

(8) Attaching and confirming an application

Attach to the application using the "ATTACH command (Page 162)". At that time, PARTNER obtains the PID and the placement information of the application, and thereafter you can refer to the memory in the application area and set breakpoints.

```
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 ↓
Or
PT>attach sample ↓
```

You can confirm whether or not the application is attached to the debugger using the "PSID command (Page 168)". An asterisk (*) is displayed if the application is not attached.

```
PT>psid ↓
PSID ****
```

If the command has been completed normally, and the space of the application (sample) is displayed, you can refer to the memory from the source code and use software breaks.
The result of the "PSID command (Page 168)" when an application has been attached is as shown below.

```
PT>psid ↓
PSID SET 740(0x2E4)  CURRENT 740(0x2E4) [ADD MODE]
APPLI. AREA : 00008000-00009FFF
APPLI. AREA : 00011000-00015FFF
APPLI. AREA : BEAD4000-BEAD4FFF
(The displayed content depends on the CPU type and MAP field settings.)
```

We recommend that you also enter the "LINUX command (Page 164)" so that shared libraries can also be debugged.

```
PT>linux load so ↓
```

If the application (sample) has been terminated, declare that the registered psid space has been deleted in PARTNER.

```
PT>PSID CLRALL ↓
```

3.5 Debugging a multi-thread application

This section describes the procedure for debugging a multi-thread application in Kernel Mode.
The debug procedure for applications that use pthread is nearly the same as was previously described in "3.4 Debugging an application (Page 99)". Only the kernel configuration, application creation and launch option specification, and multi-window debugging are different.

For debugging multi-thread applications, you can use the ADD mode, in which you debug the execution context of each thread in the same PARTNER window, and the NON_ADD mode, in which you debug each in separate PARTNER windows.
You can switch between the ADD mode and NON_ADD mode using the "PSID command (Page 168)" or by specifying the "-OS option (Page 147)".

3.5.1 Configuration required for debugging

Table 3-3 Requirements for debugging of multi-thread applications

Setting condition / Debug target	Kernel menu						-OS option			
	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	Kernel mode	Kernel ADD mode	Application mode	Application ADD mode
Application	◎	◎					○ *1		— *2	

◎ : Required, ○ : Recommended, △ : Not recommended, × : Not permitted, Empty: Either is OK

*1 The behavior differs depending on whether the ADD mode or NON_ADD mode is used.

*2 For information on the debug method in Application Mode, refer to "5.1 Debugging in Application Mode (Page 182)".

If you have not yet performed "2.2 Modifying and configuring the Linux kernel source (Page 54)", you cannot perform application debugging described here. If you want to use a kernel that is not modified, refer to "5.3 Manually debugging an application (Page 200)".

3.5.2 Debugging Procedure

This section describes the procedure for debugging a multi-thread application in Kernel Mode, using the case where a sample (thread_sample) is used as an example.

(1) "Creating an application (Page 108)"
(2) "Preparing for debugging (Page 108)"
(3) "Opening a PARTNER window for an application (Page 108)"
(4) "Loading application debug information (Page 109)"
(5) "Setting breakpoints (Page 110)"
(6) "Executing an application (Page 110)"
(7) "Breaking using PARTNER (Page 110)"

(1) Creating an application

Create the debug-target application with debug information.

You don't have to modify any part of the source code of the application.

[Example]

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

If you use the debug support file in "Preload library method (Page 63)", **do not link the debug support library (libkmcsup.so) to the application.** (You can confirm the linked shared library using the ldd command in the target.

(2) Preparing for debugging

Refer to "Launching the debug environment (Page 73)", and enable PARTNER to perform Linux debugging in Kernel Mode.

(3) Opening a PARTNER window for an application

● When the ADD mode is used.

In ADD Mode, threads created from the debug-target application are attached to a PARTNER window for an application, so open an additional PARTNER window for an application.

```
PT>multi 2 ↓
```

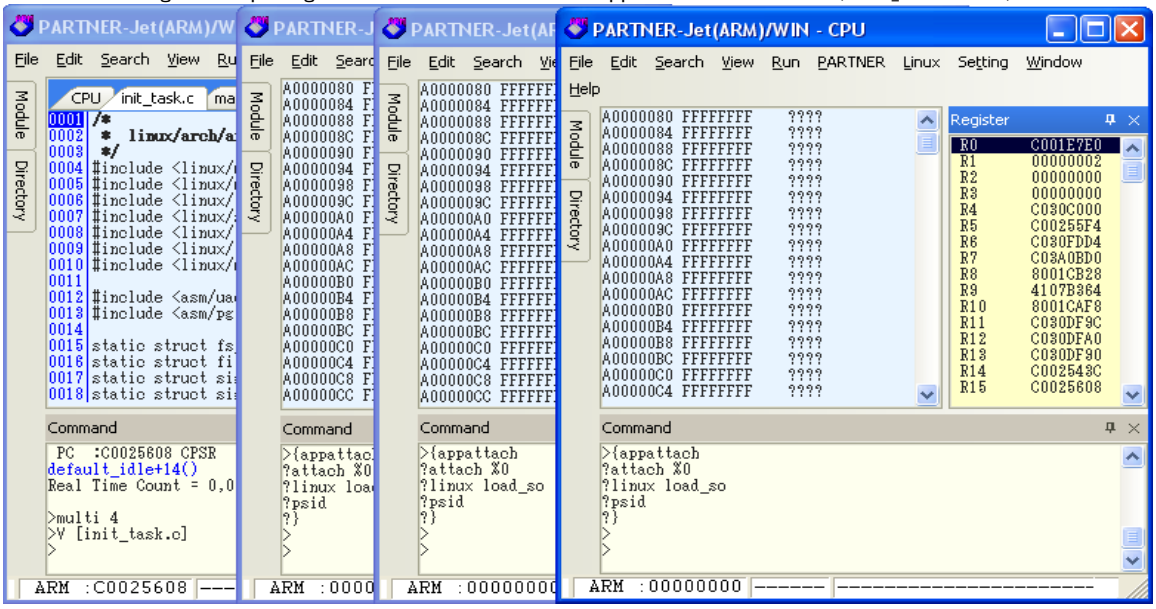
● When the NON_ADD mode is used.

If a mode other than the ADD mode is used, open PARTNER windows as many as "the number of the kernel (1) + the number of the application (1) + the number of processes to be created".

For example, for the sample thread_sample, open four PARTNER windows, because one window is necessary for the kernel and the application respectively and two windows are necessary for the threads to be created.

```
PT>multi 4 ↓
```

Fig. 3-7 Opening PARTNER windows for the application and threads (NON_ADD mode)



Window information (such as window arrangement) and the command history of PARTNER windows opened using the "MULTI command (Page 173)" or the "-MULTI option (Page 155)" are stored in a newly-created project file (e.g. JETARM_1.JPX)

(4) Loading application debug information

Load the debug information of the created application into PARTNER.

● When ADD mode is used

Load only the debug information for the debug-target application into the PARTNER window for the application.

```
PT2>ls thread sample ↓
```

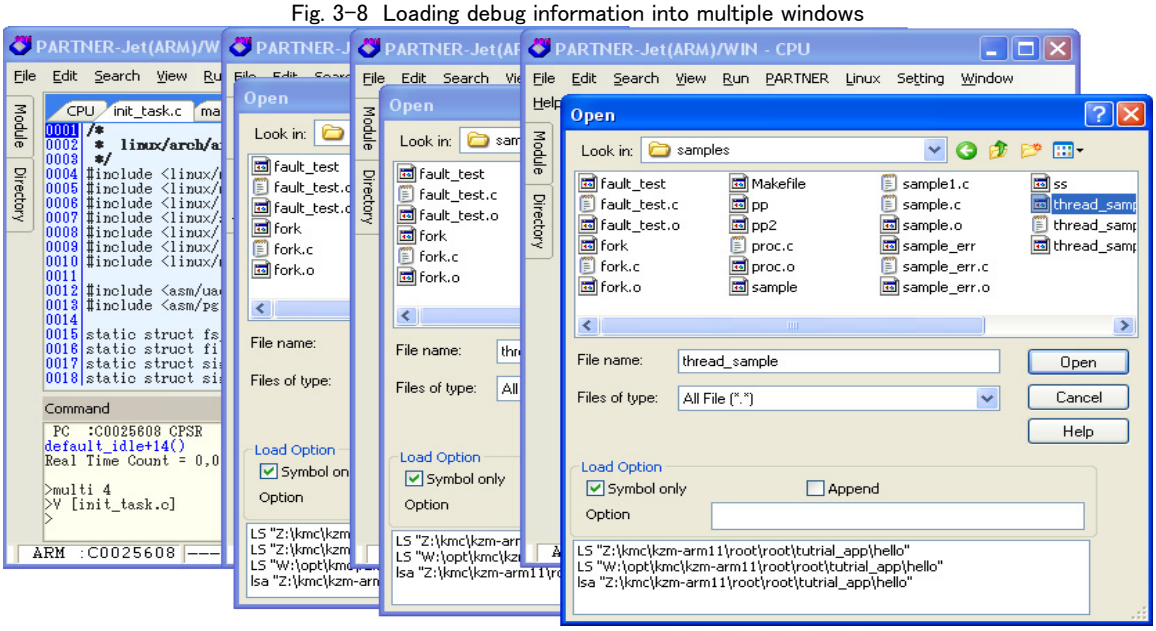
● When the ADD mode is used.

Load only the debug information for the debug-target file into the PARTNER window for the application and the PARTNER windows for the child processes/threads respectively.

Make sure to check [Symbol Only] when loading it.

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

Fig. 3-8 Loading debug information into multiple windows



(5) Setting breakpoints

Set an executable hardware breakpoint at the main() function of the application.

```
PT2>br main.ex ↓
```

(6) Executing an application

Execute the debug-target application in the target system.

```
TGT>./thread_sample ↓
```

(7) Breaking using PARTNER

If the debug-target application is executed in the target system, PARTNER breaks at the position at which the breakpoint was set (i.e. the main function).

PARTNER sets a hardware break at an address that was obtained from the debug (symbol) information.
When debugging an application, this address may also be used by another program, so that another program may cause a break.
In that case, keep the program running (by pressing the F5 key) until a break occurs in the target application.

---

(8) Attaching to and confirming an application

Attach to the application using the "ATTACH command (Page 162)". At that time, PARTNER obtains the PID and the placement information of the application, and thereafter you can refer to the memory in the application area and set breakpoints.

```
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 ↓
Or
PT2>attach thread sample ↓
```

You can confirm whether or not the application is attached to the debugger using the"PSID command (Page 168)".

```
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
(The displayed content depends on the CPU type and MAP field settings.)
```

We recommend you to also enter the "LINUX command (Page 164)") at this point, so that shared libraries can also be debugged.

```
PT2>linux load so ↓
```

● When the ADD mode is used.

Set software breakpoints in the main() function and the thread_body() function. After the application is executed, a break occurs when any of the software breakpoints is executed. If the address is executed in multiple thread contexts, a break occurs in each thread.

You can confirm execution contexts in which breaks have occurred, using the "PSID command (Page 168)".

```
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>
```

● When the NON_ADD mode is used.

Set a software breakpoint in the main() function in the main window, and another software breakpoint in the thread_body() function in the windows for the threads.

You can confirm that the application is attached in the application PARTNER window, by executing the "PSID command (Page 168)".

```
PT>psid ↓     (Kernel Window)
PSID ****
PT2>psid ↓   (Main Window)
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 ↓   (Window for Thread 1)
PSID ****
PT4>psid ↓   (Window for Thread 2)
PSID ****
(The displayed content depends on the CPU type and MAP field settings.)
```

Then, the application is resumed in the application PARTNER window, a thread is created when the create_thread() function is executed, and the PARTNER window for the thread 1 breaks at the entry of the thread 1. At that time, the PID and placement information of the thread is automatically collected, and thereafter you can refer to the memory in the thread area and set breakpoints (unnecessary to attach to the created thread).

You can confirm that the application is attached in the PARTNER window for the thread 1, by executing the "PSID command (Page 168)".

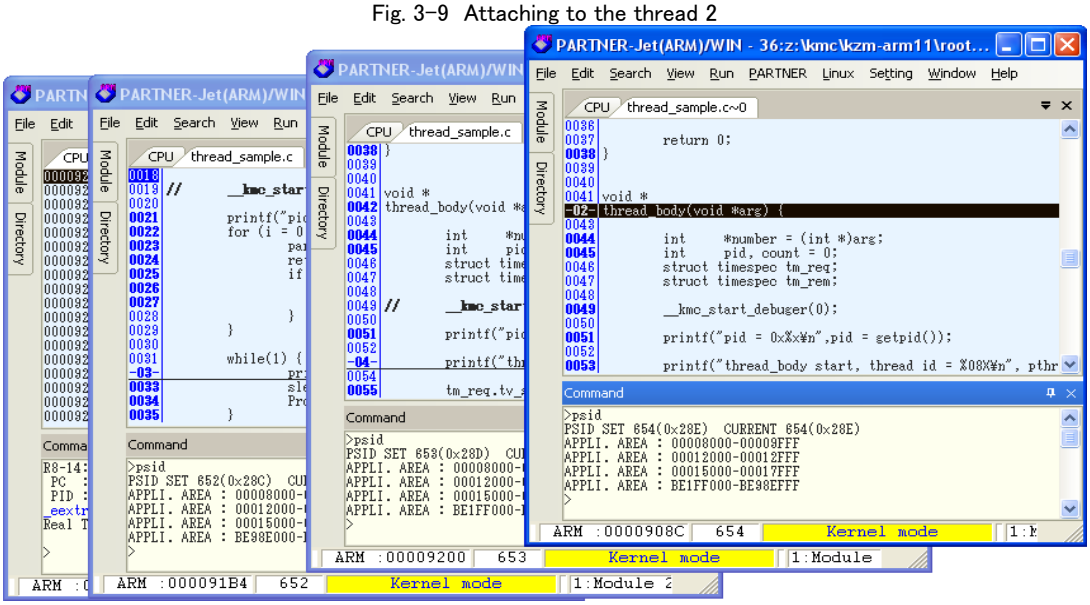
```
PT3>psid ↓
PSID SET 742(0x2E6)  CURRENT 742(0x2E6)
APPLI. AREA : 00008000-00009FFF
APPLI. AREA : 00012000-00012FFF
APPLI. AREA : 00015000-00017FFF
APPLI. AREA : BE1FF000-BE913FFF
APPLI. AREA : 4000E000-4000EFFF
PT3>g
```

When the application is resumed in the application PARTNER window again and a thread is created by the execution of the create_thread() function, the PARTNER window for the other thread breaks at the entry of the thread. At that time, the PID and placement information of the thread is automatically collected, and thereafter you can refer to the memory in the thread area and set breakpoints (unnecessary to attach to the created thread).

You can confirm that the application is attached in the PARTNER window for the thread 2, by executing the "PSID command (Page 168)"

```
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
(The displayed content depends on the CPU type and MAP field settings.)
```

Fig. 3-9 Attaching to the thread 2



If PARTNER does not break properly, the kernel configuration may not be correct. Makes sure that the OS debug mode, which was specified in "2.2.5 Kernel configuration (Page 57)" and "2.4.2 Configuring and launching PARTNER (Page 75)", is in Application Mode.

Also confirm that the support library (libkmcsup.so.2.0.0) has been preloaded.

3.6 Debugging a multi-process application

This section describes the procedure for debugging a multi-process application in Kernel Mode.
The debug procedure for applications that use fork() is nearly the same as was previously described in "3.4 Debugging an application (Page 99)" Only the kernel configuration, application creation and launch option specification, and multi-window debugging are different.

Although both multi-thread and multi-process handle multiple execution contexts that run in parallel, when debugging multi-process applications, there is no difference between the ADD mode and NON_ADD mode, unlike the debugging of multi-thread applications.
The ADD mode of PARTNER is effective only for thread debugging. To debug different processes, you need to open a PARTNER window for each process.

3.6.1 Configuration required for debugging

Table 3-4 Configuration settings for debugging of multi-process applications

Setting condition / Debug target	Kernel menu						-OS option			
	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	Kernel mode	Kernel ADD mode	Application mode	Application ADD mode
Application	◎	◎					○ *1		— *2	

◎ : Required, ○ : Recommended, △ : Not recommended, × : Not permitted, Empty: Either is OK

*1 Debugging can be performed in both ADD Mode and NON_ADD Mode. However, there is no behavioral difference.。

*2 For information on the debug method in Application Mode, refer to "5.1 Debugging in Application Mode (Page 182)"."5.1 Debugging in Application Mode (Page 182)".

If you have not yet performed "2.2 Modifying and configuring the Linux kernel source (Page 54)", you cannot perform application debugging described here. If you want to use a kernel that is not modified, refer to "5.3 Manually debugging an application (Page 200)".

3.6.2 Debugging Procedure

This section describes the procedure for debugging a multi-process application in Kernel Mode, using the case where a sample (fork) is used as an example.

(1) "Creating an application (Page 118)"
(2) "Preparing for debugging (Page 118)"
(3) "Opening a PARTNER window for an application (Page 118)"
(4) "Loading application debug information (Page 118)"
(5) "Setting breakpoints (Page 119)"
(6) "Executing an application (Page 119)"
(7) "Breaking using PARTNER (Page 119)"

(1) Creating an application

Create the debug-target application with debug information.

You don't have to modify any part of the source code of the application.

[Example]

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

If you use the debug support file in "Preload library method (Page 63)", do not link the debug support library (libkmcsup.so) to the application. (You can confirm the linked shared library using the ldd command in the target.)

(2) Preparing for debugging

Refer to "Launching the debug environment (Page 73)", and enable PARTNER to perform Linux debugging in Kernel Mode.

(3) Opening a PARTNER window for an application

Open a PARTNER window for an application.

As many PARTNER windows as "the number of the kernel (1) + the number of the application (1) + the number of processes to be created" are necessary.

For the "fork" sample, open three PARTNER windows, because one window is necessary for both kernel and application respectively and one window is necessary for the child process to be created.

```
PT>multi 3 ↓
```

Regardless of the mode used (ADD mode or NON_ADD mode), as many PARTNER windows are necessary as there are numbers of processes.
However, if you want to debug only one process, you can debug it in the window used for the kernel.

(4) Loading application debug information

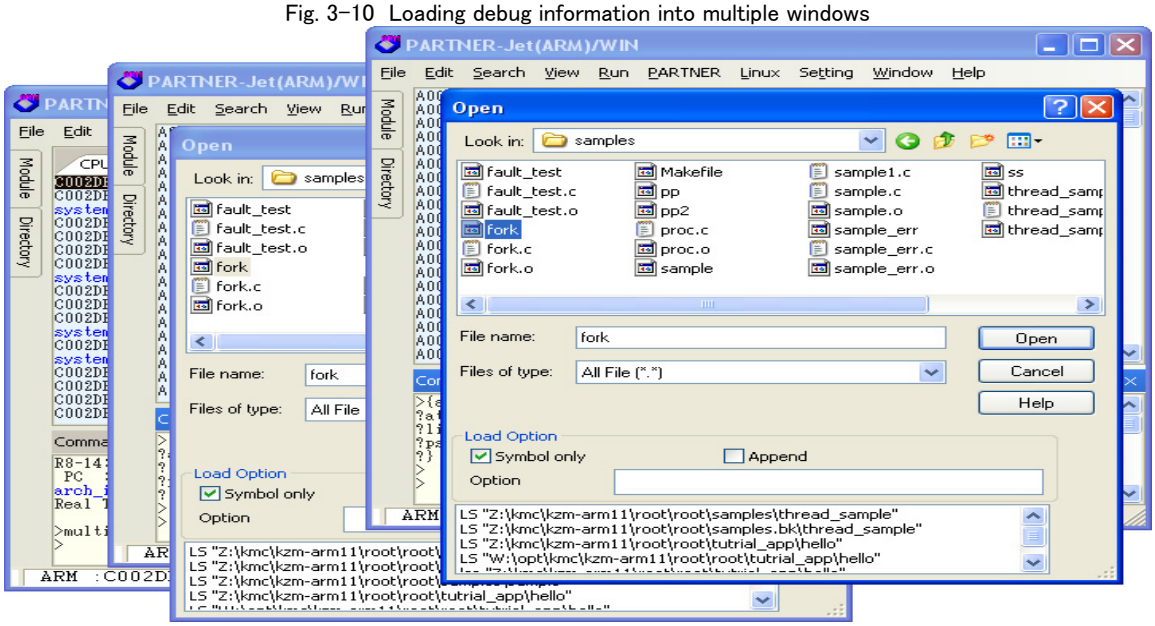
Load the debug information of the created application into PARTNER.

Load only the debug information for the debug-target file into the application PARTNER window and the child process PARTNER windows respectively.

Make sure to check [Symbol Only] when loading it.

```
PT2>ls fork ↓
PT3>ls fork ↓
```

Fig. 3-10 Loading debug information into multiple windows



(5) Setting breakpoints

Set an executable hardware breakpoint at the main() function of the application.

```
PT2>br main.ex ↓
```

(6) Executing an application

Execute the debug-target application in the target system.

```
TGT>./fork ↓
```

(7) Breaking using PARTNER

If the debug-target application is executed in the target system, PARTNER breaks at the position at which the breakpoint was set (i.e. the main function).

PARTNER sets a hardware break at an address that was obtained from the debug (symbol) information.
When debugging an application, this address may also be used by another program, so that another program may cause a break.
In that case, keep the program running (by pressing the F5 key) until a break occurs in the target application.

---

(8) Attaching to and confirming an application

Attach to the application using the "ATTACH command (Page 162)". At that time, PARTNER obtains the PID and the placement information of the application, and thereafter you can refer to the memory in the application area and set breakpoints.

```
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 ↓
Or
PT>attach fork ↓
```

You can confirm whether or not the application is attached to the debugger using the "PSID command (Page 168)".

```
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
(The displayed content depends on the CPU type and MAP field settings.)
```

We recommend you to also enter the "LINUX command (Page 164)" at this point, so that shared libraries can also be debugged.

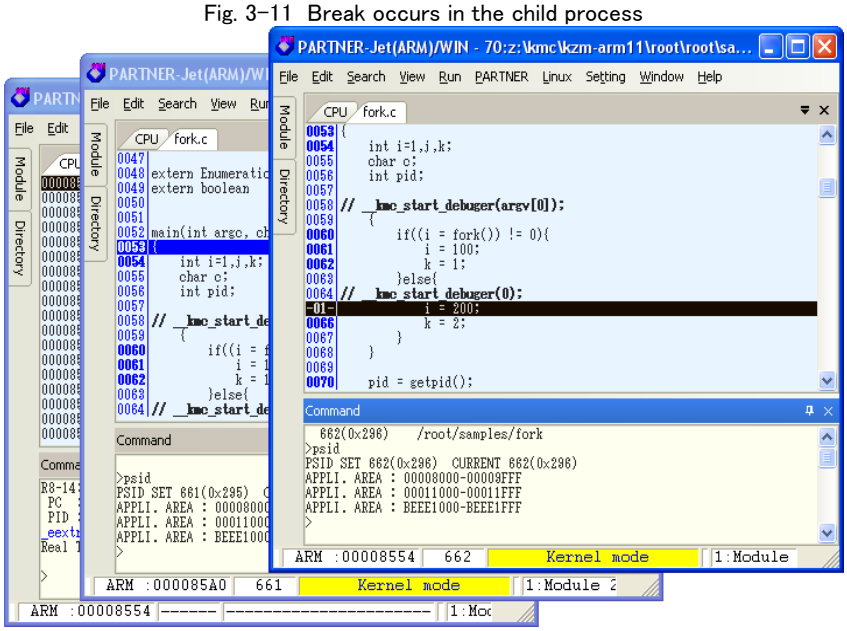
```
PT>linux load so ↓
```

You can confirm that the application is attached in the application PARTNER window, by executing the "PSID command (Page 168)"

```
PT>psid ↓     (Kernel Window)
PSID ****
PT2>psid ↓    (Main Window)
PSID SET 740(0x2E4)  CURRENT 741(0x2E5) [ADD MODE]
APPLI. AREA : 00008000-00009FFF
APPLI. AREA : 00011000-00011FFF
APPLI. AREA : BEFBA000-BEFBAFFF
PT3>psid ↓   (Child process Window)
PSID ****
(The displayed content depends on the CPU type and MAP field settings.)
```

Use the G command or the G/A command to resume the application in the application PARTNER window. When the fork() function is executed and the process is created, a break occurs at the processing of the child process side of the fork() function in the child process PARTNER window. At that time, the PID and placement information of the thread is automatically collected, and thereafter you can refer to the memory in the child process space and set breakpoints (unnecessary to attach to the created process).

Fig. 3-11 Break occurs in the child process



You can confirm that the process is attached in the PARTNER window for the child process, by executing the "PSID command (Page 168)".

```
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
```

If PARTNER does not break properly, the kernel configuration may not be correct. Makes sure that the OS debug mode, which was specified in "2.2.5 Kernel configuration (Page 57)" and "2.4.2 Configuring and launching PARTNER (Page 75)", is in Application Mode.

Also confirm that the support library (libkmcsup.so.2.0.0) has been preloaded.

3.7 Debugging a shared library

A shared library is a program module that is used by multiple processes, but it is part of an application during execution. For that reason, you put it into a state where it can be debugged in an application to which it belongs, and debug it as part of that application.

3.7.1 Configuration required for debugging

As a debug target, it is a process of an application. Required configuration settings are the same as those of application debugging.

The application needs to be attached before starting to debug a shared library.

3.7.2 Debugging Procedure

This section describes the procedure for debugging a shared library, using the case where glibc is debugged as an example.

(1) "Add debug information to a shared library (Page 123)
(2) "Attaching to an application (Page 123)
(3) "Loading debug information for a shared library (Page 123)

(1) Add debug information to a shared library

Add debug information to the debug-target shared library.
Select for the debug information the format that is used for the kernel and application. If PARTNER cannot load the debug information properly, try other formats.

(2) Attaching to an application

Refer to "3.4 Debugging an application (Page 99)", "3.5 Debugging a multi-thread application (Page 106)", "3.6 Debugging a multi-process application (Page 116)" and "5.1 Debugging in Application Mode (Page 182)" to attach the application to the PARTNER window for the application.

(3) Loading debug information for a shared library

Use the "LINUX command (Page 164) while attaching to the application.

```
PT>linux load_so ↓
```

To explicitly load debug information, use the LSA command. Make sure to check [Symbol Only] and [Append] when loading it from the GUI menu.

```
PT>lsa libc-2.2.5.so ↓
```

Once the debug information has been loaded, you can perform source-level debugging on the shared library.
If you execute the application after setting a breakpoint in the shared library, a break occurs at the set breakpoint.
If you execute the "PSID command (Page 168) at that point, you can see that the shared library space is newly attached to the PARTNER window.

```
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
(The displayed content depends on the CPU type and MAP field settings.)
```

3.7.3 Notes on shared library debugging

From a perspective of program execution, a shared library can be seen as a library for which link-address resolution is performed when an application is executed. Since the code within a shared library is executed in a process of an application, you will be able to debug it after attaching to the process.
However, the behavior is different from the main part of an application and a static link library, notes for debugging a shared library is described below.

Threads/processes created within a shared library

The application debug support file of PARTNER (kmc-support.c) can be built as the shared library (libkmcsup.so), because the debugger inserts into the pthread_create() and the fork() functions code to identify execution context generation.
That is to say, if the pthread_create() function or the fork() function is used in the shared library, the pthread_create() function or the fork() function in the libkmcsup.so library has to be executed. If the debug support file is used as a preload library, the pthread_create() function or fork() function in the libkmcsup.so library should be executed before the pthread_create() function in the pthread library or the fork() function in the libc library.
If debugging of threads and processes that are generated within the shared library cannot be performed successfully, check whether or not the functions in the libkmcsup.so library are called.
On the other hand, if the support file (kmc-support.c) is directly linked to the application, the stub functions (_kmc_start()) must be inserted into the places where the pthread_create() function and the fork() function are used. However, since the shared library was built separately from the application, inserting the stub functions into the source code of the application may not be sufficient.

Breakpoints in a shared library

The code within a shared library can be executed in multiple processes. If you set breakpoints in the code within a shared library, a break might occurs in a process other than the process being debugged. PARTNER automatically performs re-execution if a break occurs in a process whose process ID is different from that of the process being debugged, so from an operating standpoint it is not regarded as having stopped. In actuality, a CPU halt and re-execution are performed. So if it occurs at a point that is referred from many processes, it may cause the operation to slow down.

3.8 Linux OS compatible history display

The real-time trace function of PARTNER can perform history display that is compatible with the Linux OS.
If the kernel and an application are debugged in separate PARTNER windows (using multiple windows), you can display the history of processes that are attached (for which debug information is loaded).

3.8.1 Configuration requirements

Table 3-5 Configuration settings for the Linux OS compatible history display

Setting condition / Debug target	Kernel menu						-OS option			
	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	Kernel mode	Kernel ADD mode	Application mode	Application ADD mode
Application	◎	◎					○	○	× *1	

◎ : Required, ○ : Recommended, △ : Not recommended, × : Not permitted, Empty: Either is OK

*1 Perform the Linux OS compatible history display in "Kernel Mode" or "Kernel ADD Mode".

3.8.2 Debugging Procedure

This section describes the procedure for history display of an application (process) in Kernel Mode, using the case where a sample (sample) is used as an example.

(1) "Configuration of the kernel necessary for the Linux OS-compatible history display (page 126)"
(2) "Preparing an application (page 126)"
(3) "Linux OS compatible history display (page 126)"

(1) Configuration of the kernel necessary for the Linux OS-compatible history display

To perform Linux OS compatible history display in Kernel Mode, enable [Debug Information Type] and [Enable Patch for PARTNER Debug] in the Linux kernel configuration.

After the configuration has been done, create the Linux kernel (vmlinux).

```
LINUX86>make ↓
```

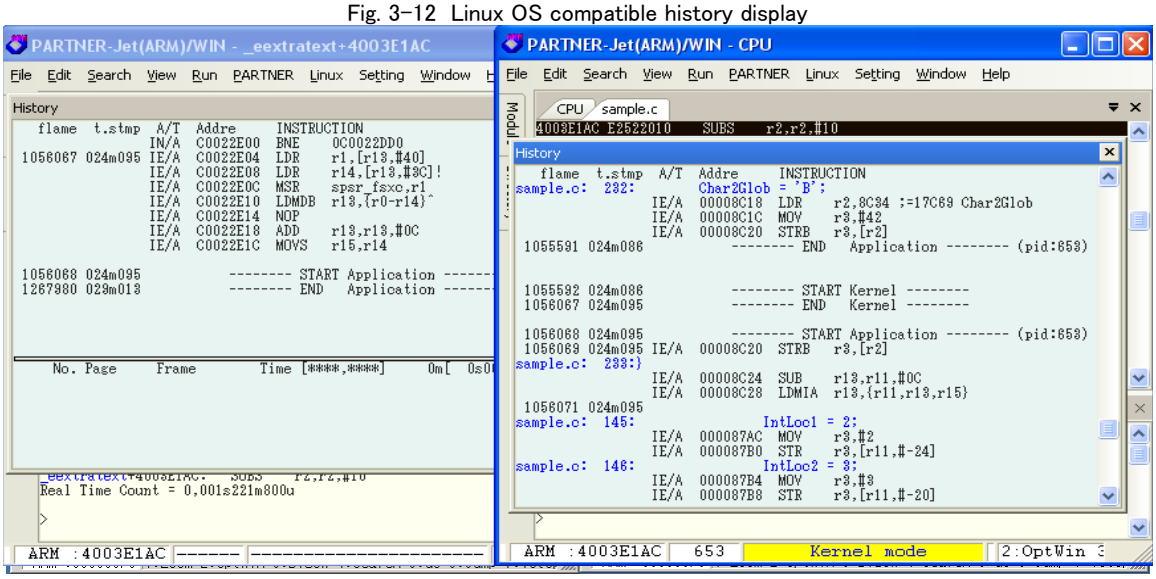
(2) Preparing an application

Refer to "3.4 Debugging an application (Page 99), and adjust the debug environment.

(3) Linux OS compatible history display

It performs a break on the CPU and displays history.

Fig. 3-12 Linux OS compatible history display



The following lines are additionally included in the history display to support Linux OS.

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

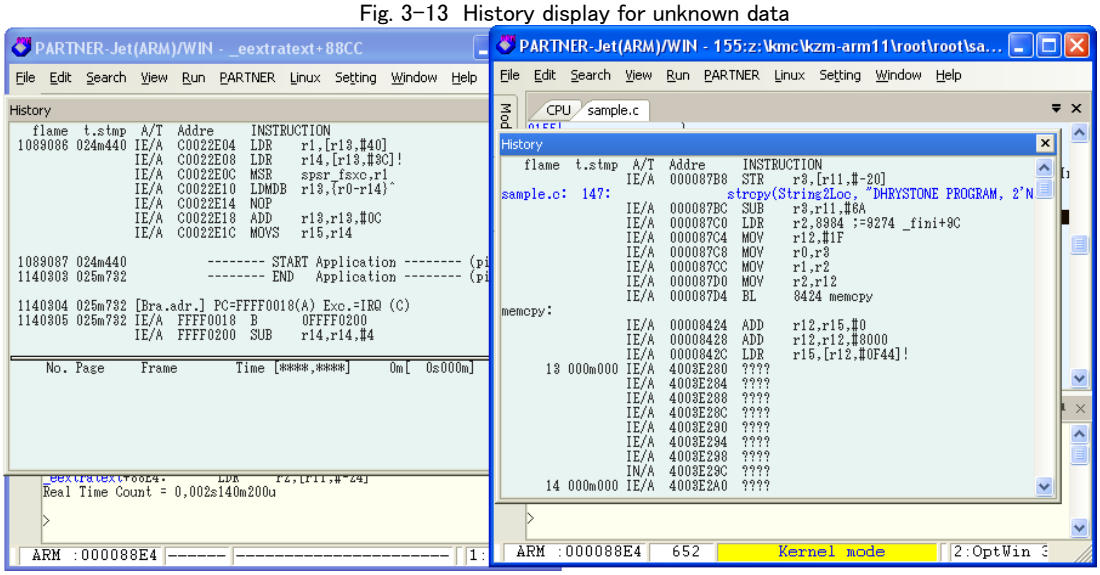
The display is slightly different in some PARTNER versions (e.g. "**** START Kernel ****" and "**** END Kernel ****").

If you double click the above line when there is a PARTNER window in which you are debugging the target kernel or application, the focus moves to that PARTNER window and the same frame number is displayed.

????? appears in some parts of the history display in the PARTNER window for the application. This indicates areas in the process being debugged where debug information has not been loaded, for example, places in a shared library where memory access is not available.

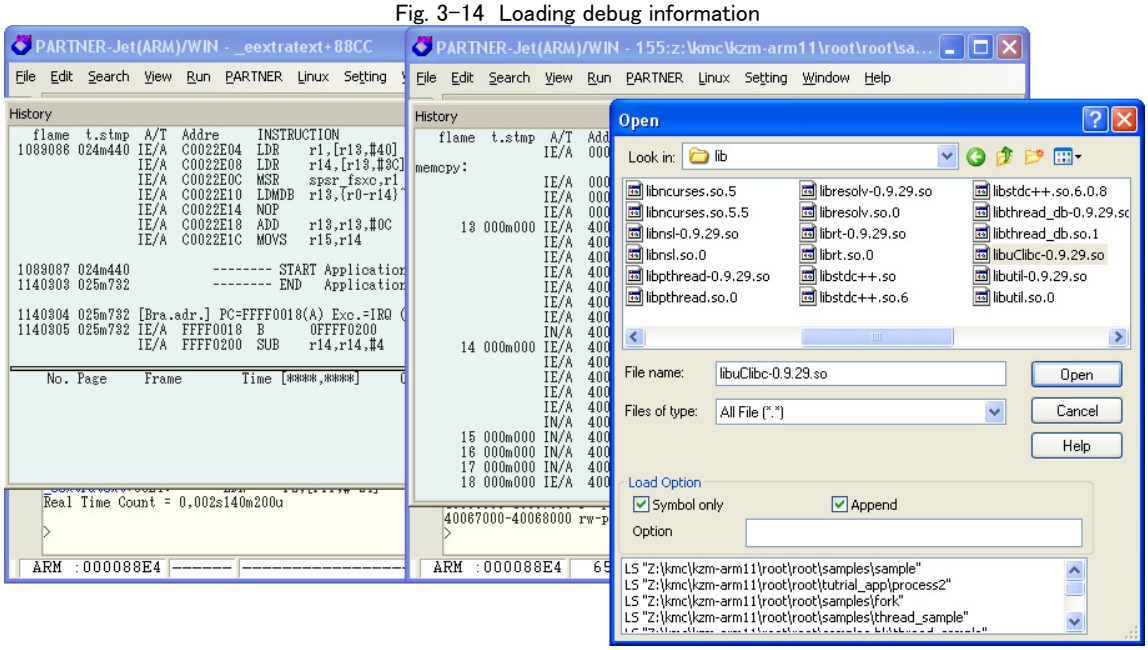
If the K2 Prefix is used for the "-OS option (page 147)" of the PARTNER launch configuration, they may be automatically resolved and ??????' s are not displayed.

Fig. 3-13 History display for unknown data



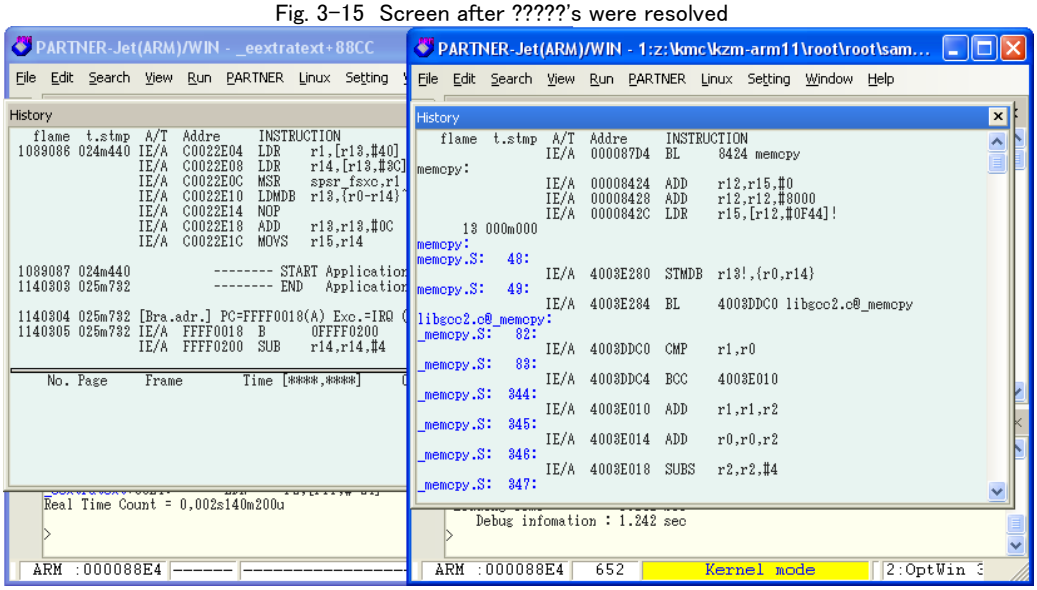
To resolve a ????? display, load the debug information for the corresponding address space. This can be confirmed using the "MAPS command (page 161)".

Fig. 3-14 Loading debug information



Once the debug information has been loaded, ????? display is resolved and normal information is displayed.

Fig. 3-15 Screen after ?????'s were resolved



To display the history of processes to which PARTNER has not been attached in the current history, follow the procedure below:

1. Save the current trace data as binary data from PARTNER.

   PT>tdsbin ↓

2. Open a new PARTNER window using the "MULTI command (page 173)".
3. Load the debug information of the target process in the newly opened PARTNER window.

   PT>ls <File Name> ↓

4. Set the target PID in the newly opened PARTNER window.

   PT>psid test <PID> ↓

5. Load the trace data saved in Step 1 into the newly opened PARTNER window.

   PT>tdlbin ↓

6. The target real-time trace is displayed in the history window.

This method only enables you to view the history of the target process in the history window, and you cannot debug it.

(4) Execution example

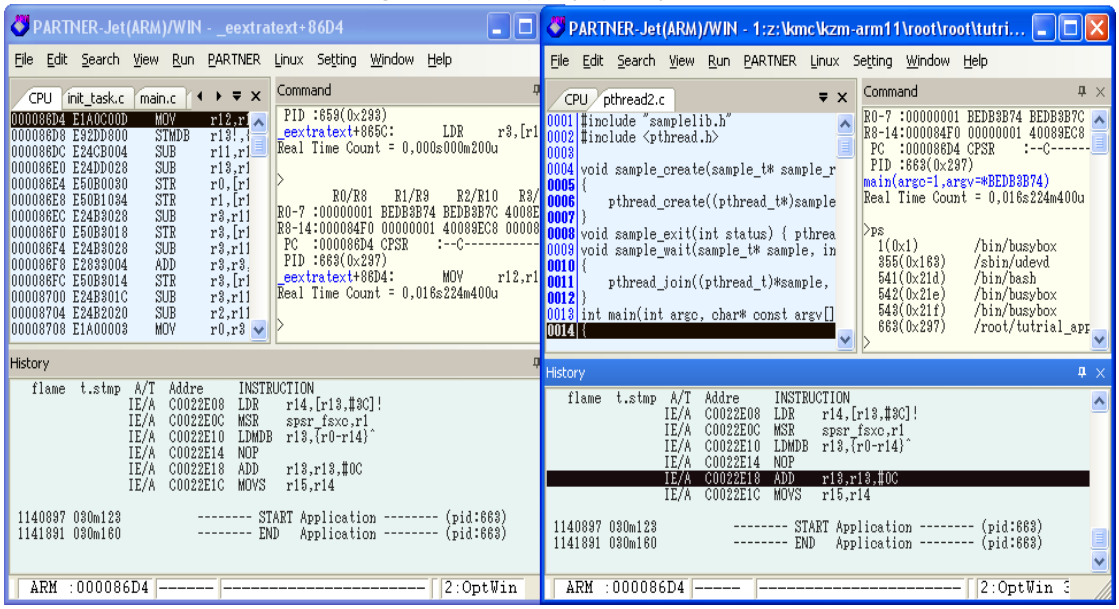
Using the sample program "pthread2.c (page 36)" that was used in "Creating a shared library (page 35)", we explain the behavior of actual execution from the state when Linux has just been launched.

Start debugging from the head of the application.

```
PT>multi 2 ↓
PT2>ls pthread2 ↓
PT2>brc * ↓
PT2>br main.ex ↓
PT2>g ↓
TGT>./pthread2 ↓
(The execution stops at the main function by the hardware breakpoint)
```

Clicking on the history column displays the trace result. But the symbol information of the application process is not displayed, because you have not attached to the process yet.

Fig. 3-16 History display (stops at main)

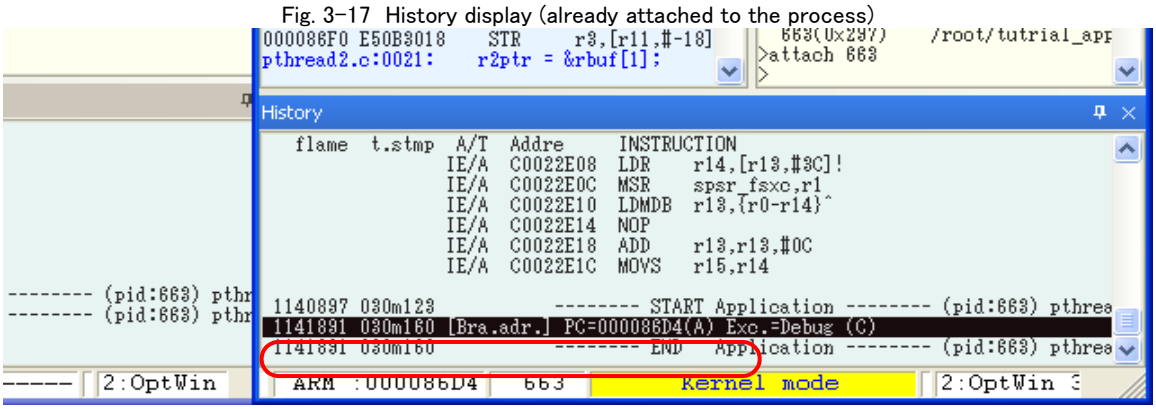


Attach to the process of the application.

```
PT2>ps ↓
PT2>attach 740 ↓
```

If you display the trace result while attaching to the process, the symbol information of the application is displayed

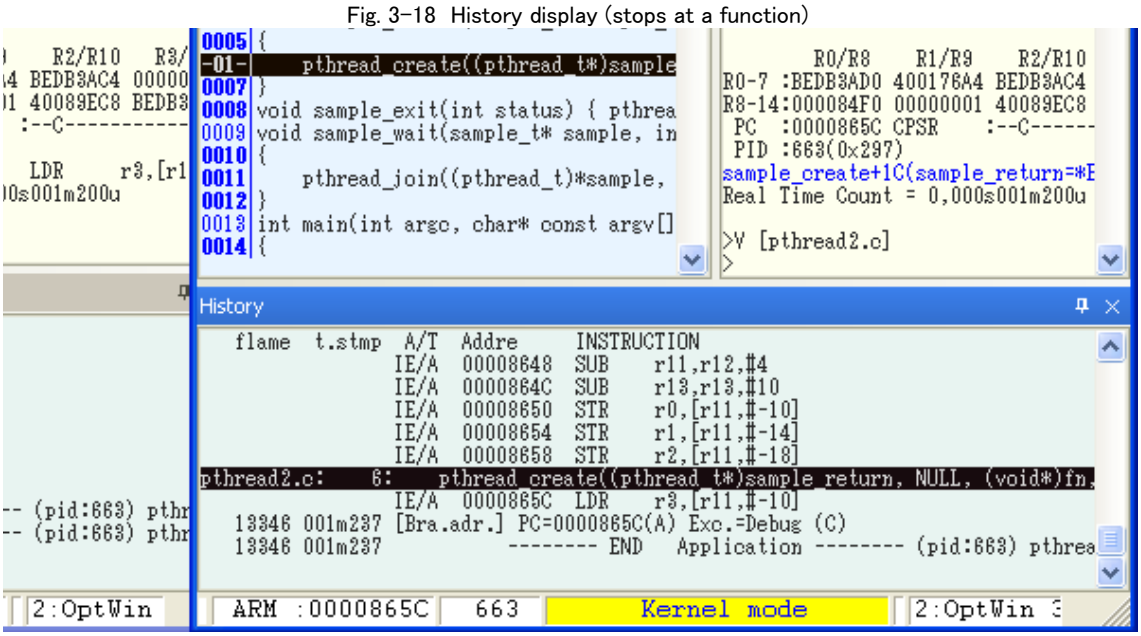
Fig. 3-17 History display (already attached to the process)



PT2>BP .sample_create+1 ↓

PT2>g ↓

Fig. 3-18 History display (stops at a function)



In this program, the sample_create() function is called from the libsample.so shared library. But symbols in the shared library are not shown in Fig. 3-18. If you also use the "LINUX command (page 164)" with load_so specification when attaching to the process, information on the shared library is also displayed.

Here we describe the procedure to restart debugging from the head of the application.

PT>multi 2 ↓

PT2>ls pthread2 ↓

PT2>brc * ↓

PT2>br main.ex ↓

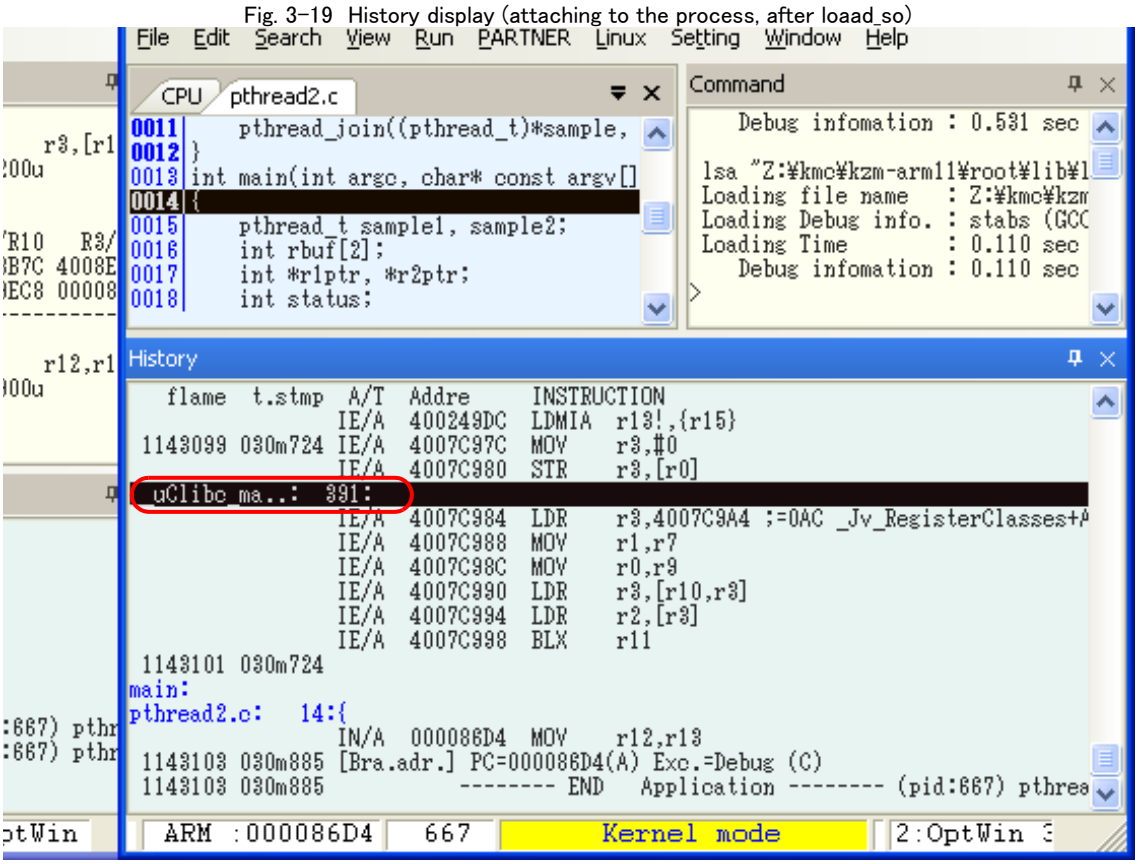
PT2>g ↓

TGT>./pthread2 ↓

(The execution stops at the main function by the hardware breakpoint)

PT2>ps ↓
PT2>attach 740 ↓
(Attaches to the process of the application)
PT2>linux load so ↓
(Loads information of the shared library)

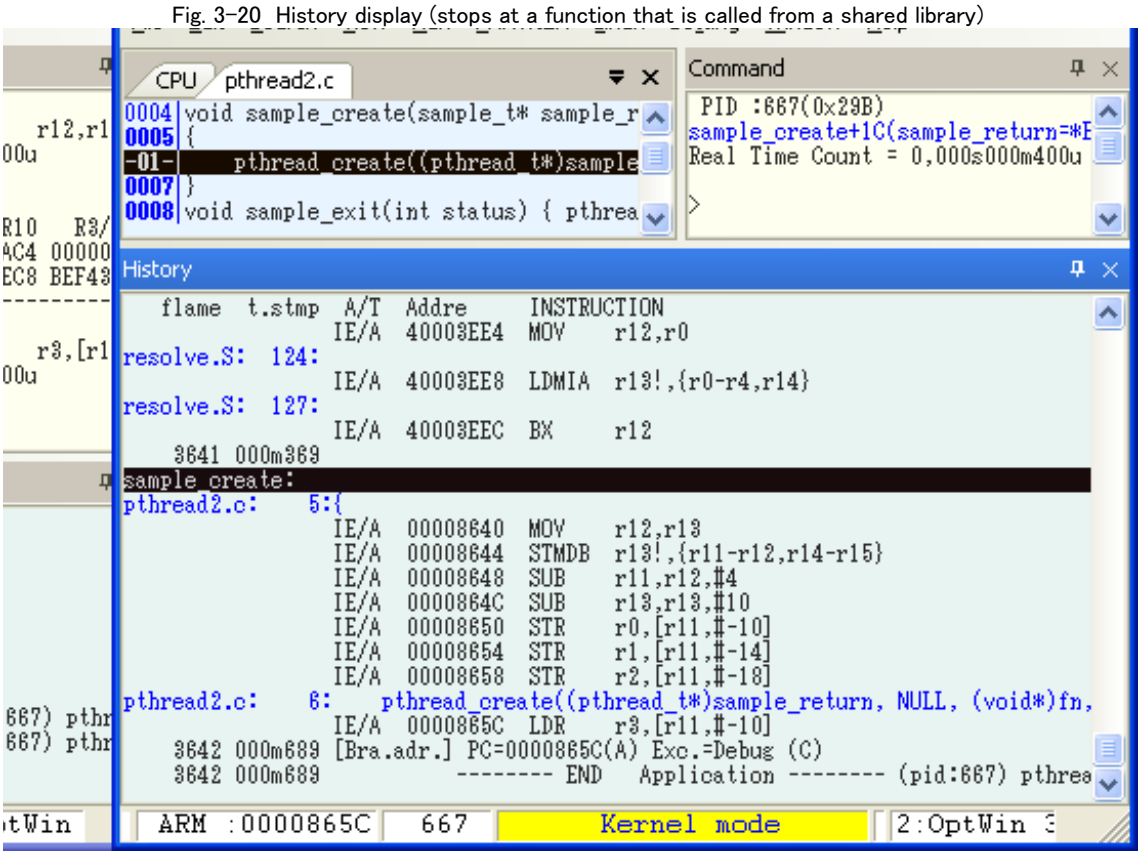
Fig. 3-19 History display (attaching to the process, after loaad_so)



At this point, the symbol information that can be read from shared libraries that use libc is displayed.
Moreover, set a breakpoint in the sample_create() function and execute the application.

PT2>BP .sample_create+1 ↓
PT2>g ↓

Fig. 3-20 History display (stops at a function that is called from a shared library)



Compare Fig. 3-18 and Fig. 3-20. From the history display stopping at the sample_create() function, you see that the symbol information included in the libsample.so shared library is displayed in Fig. 3-20.

3.9 Attaching to a running application

PARTNER can attach to an application already running in the target system, and put it into a state where debugging can be performed.
This has no particular technological difference from normal application debugging where debugging is performed from the head of a program (refer to "Debugging an application (Page 99)" and "Debugging in Application Mode (Page 182)"
The only difference is that setting breakpoints in advance at symbols in an application (e.g. main) is not required.

This section describes the procedure for attaching to a running application and debugging it, in the following order.

(1) "Preparing for debugging (Page 134)"
(2) "Opening a PARTNER window for an application (Page 134)"
(3) "Confirming the PID of a running application (Page 135)"
(4) "Attaching to a running application (Page 135)"
(5) "Loading application debug information (Page 136)"

(1) Preparing for debugging

To attach to a running application, it is necessary that the features of the debug support file kmc-support.c have been made into a shared library.

When using a debug support file in the form of a share library, you need to register the address of the _kmc_sleep_thread symbol with PARTNER using the "LINUX command (Page 164)".

[Example]

```
LINUX86>nm <Library Name> | grep  kmc_sleep_thread ↓
000f2b54 t __kmc_sleep_thread
PT>LINUX set_attach_offset <Library Name> <Address of  __kmc_sleep_thread> ↓
```

For details, refer to "Preload library method (Page 63)" or "How to insert the support file into glibc (Page 240)"

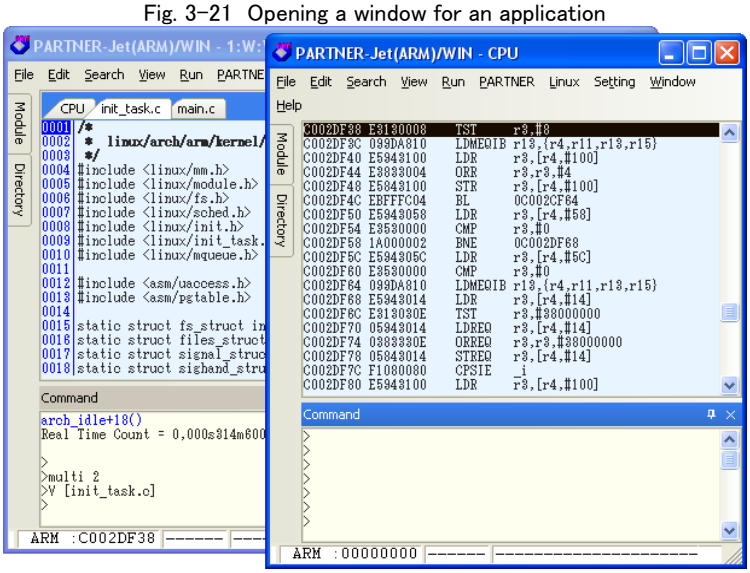
Refer to "3.4 Debugging an application (Page 99)" or "5.1 Debugging in Application Mode (Page 182)", and execute the kernel from PARTNER and the debug-target application.

(2) Opening a PARTNER window for an application

Open a new PARTNER window for the application using the "MULTI command (Page 173)".

```
PT>MULTI 2 ↓
```

Fig. 3-21 Opening a window for an application



Window information (such as window arrangement) and the command history of PARTNER windows opened using the "MULTI command (Page 173)" or the "-MULTI option (Page 155)" are stored in a newly-created project file (e.g. JETARM_1.JPX) and will be used at the next launch.

(3) Confirming the PID of a running application

Confirm the PID of the debug-target application using the "PS command (Page 160)".

```
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
```

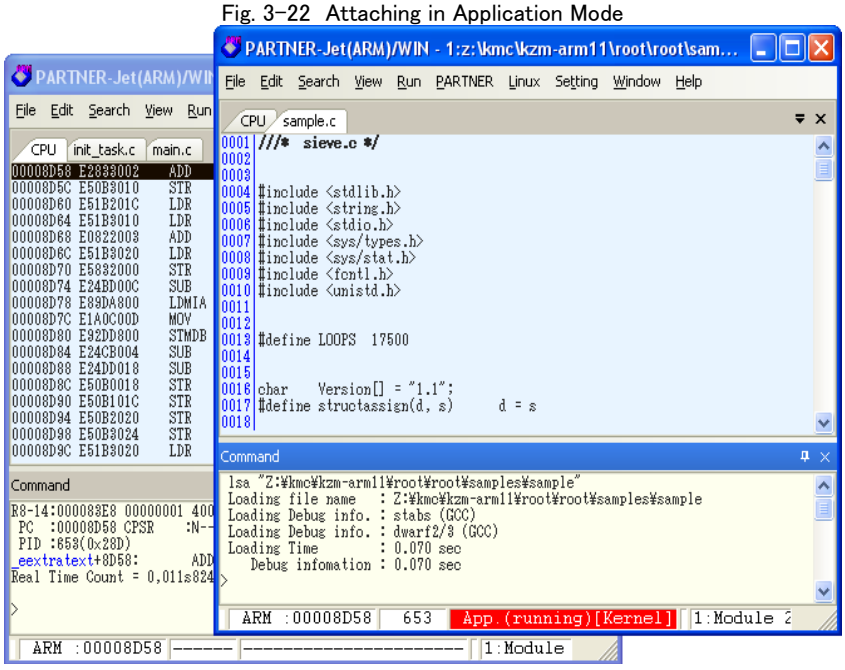
(4) Attaching to a running application

Execute the "ATTACH command (Page 162)" in the PARTNER window for the application to attach to the debug-target application.

```
PT>attach 740 ↓
```

If it is running in Application Mode, the status bar changes from "Running the Target" to "Running the Application" when attaching has been completed. It does not change if it is running in Kernel Mode.

Fig. 3-22 Attaching in Application Mode



The behavior in Application Mode has not been described in this document yet, so refer to "5.1 Debugging in Application Mode (Page 182)".

You can confirm that attaching has been completed using the "PSID command (Page 168)".

```
PT>psid ↓
PSID SET 740(0x2E4)  CURRENT 740(0x2E4)
APPLI. AREA : 00008000-00009FFF
APPLI. AREA : 00011000-00015FFF
APPLI. AREA : BEBF6000-BEBF6FFF
```

(The displayed content depends on the CPU type and MAP field settings.)

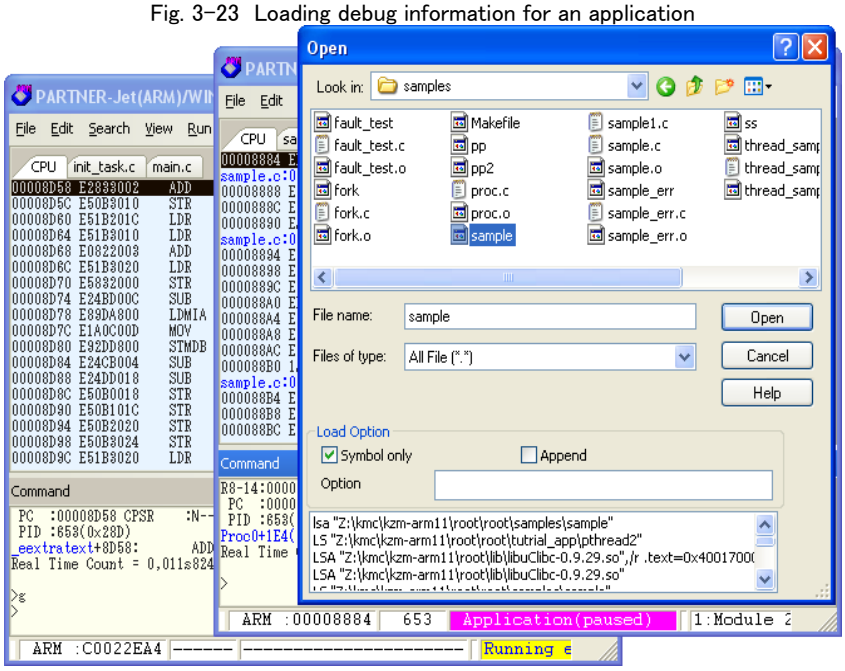
If PARTNER does not break properly, the kernel configuration may not be correct. Makes sure that the OS debug mode, which was specified in "2.2.5 Kernel configuration (Page 57)" and "2.4.2 Configuring and launching PARTNER (Page 75)", is in Application Mode.
Also confirm that the support library (libkmcsup.so.2.0.0) has been preloaded.

(5) Loading application debug information

Load only the debug information for the application that has been attached in the application PARTNER window. Make sure to check [Symbol Only] when loading it.

```
PT>ls sample ↓
```

Fig. 3-23 Loading debug information for an application



Once the debug information has been loaded, you can perform source-level debugging on the running application.

PARTNER remembers in the dialog and command history the names and locations of files that have been loaded, so the subsequent operations can be simplified.

Chapter 4 Linux compatible function reference

This chapter describes features newly added to Linux for PARTNER and the setting methods for these features.

4.1 CFG file extensions

To support Linux debugging, the following setting has been added or extended.

・"MAP field (Page 141)"

MAP field

Format

MAP Start Address, End Address[, Attribute]

Functionality

Specifies the memory area that PARTNER accesses.

Description

The MAP field specifies the memory area to which PARTNER can access. To debug Linux kernels, loadable modules and applications using PARTNER, the specification of the MAP field has been changed.
To perform Linux debugging, you need to specify the memory maps of the logical address area and the physical address area. Do not use 00000000 and FFFFFFFF for memory map specification in the physical address area, and specify existing memory areas.

Specify the logical address area and the attributes of that area according to the format.

Table 4-1 Attribute specification

Attribution	Description
APPLI	The area process and shared library are located.
KERNEL	The area Linux kernel is located.
MODULE	The area loadable module is located.

Follow the explanation below to check each attribute.

● Checking the APPLI attribute area

In the target system, view "/proc/1/maps" and determine the area.
Even if no logical address area is specified in the MAP field, you can view the memory using the "MAPS command (Page 161)" and determine the application area when the kernel can be loaded and executed using PARTNER.

```
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>
```

The display differs depending on the CPU type and Linux kernel version. Since the first column is in the physical address area, set the display for addresses other than 00000000 and FFFFFFFF.

● Checking the KERNEL attribute area

Determine this area by referring to the System.map file that was output when the Linux kernel was built and the link script file used to build the kernel.

● Checking the MODULE attribute area

Check it in the Linux kernel source.

For ARM CPUs:
Determine it by calculating the area ranging from VMALLOC_START to VMALLOC_END that are specified in include/asm/arch/vmalloc.h.
Load and execute the Linux kernel with debug information using PARTNER, make a break happen after the high_memory variable has been initialized, and confirm the high_memory variable using the VAL command of PARTNER.

```
PT>val high_memory ↓
(void *) *C1000000
```

For MIPS/SH/AM33 CPUs:
Determine it in the area ranging from VM_ALLOC_START to VMALLOC_END that are specified in include/asm/pgtable.h.

【Examples of MAP specification in the environment configuration file】

For ARM CPUs:

```
MAP     00008000,bfffffff,APPLI             ; -- Linux Application Space
MAP     c0000000,c17fffff,KERNEL            ; -- Linux Kernel Space
MAP     c1800000,c1ffffff,MODULE            ; -- Linux Loadable Module Space
MAP     ffff0000,ffffffff,KERNEL            ; -- Linux Kernel Space

MAP     00000000,000fffff                   ; -- Area accessible when the MMU is OFF
MAP     c0000000,c1ffffff                   ; -- Area accessible when the MMU is OFF
```

Properly make MAP specification for the case where the MMU is ON and the case where the MMU is OFF. First make MAP specification for the case where the MMU is ON (state in which the Linux OS is running). After that specification, put MAP specification for the MMU OFF state (state where the power has just come on or the system has just been reset). In ARM Linux environments, APPLI, KERNEL and MODULE memory spaces tend to be different depending on the build environment. So check it by referring to the System.map and arc/arm/vmlinux.lds files that were created when the kernel was built.

For MIPS CPUs:

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

As for MIPS CPUs, the space 0x80000000~0x9FFFFFFF and the space 0xA0000000~0xBFFFFFFF refer to the same memory space. So make sure to correctly make MAP specification for those two spaces.

For SH CPUs:

```
MAP     00400000,7fffffff,APPLI
MAP     8c000000,8fffffff,KERNEL
MAP     c0000000,cfffffff,MODULE
MAP     00000000,003fffff
MAP     80000000,83ffffff
MAP     88000000,8bffffff
MAP     8c000000,8fffffff
MAP     90000000,93ffffff
MAP     94000000,97ffffff
MAP     a0000000,a3ffffff
MAP     a8000000,abffffff
```

```
MAP     ac000000, affffffff
MAP     b0000000, b3ffffff
MAP     b4000000, b7ffffff
MAP     f0000000, ffffffff
```

As for SH CPUs, the space 0x80000000~0x9FFFFFFF and the space 0xA0000000~BFFFFFFF refer to the same memory space.. So make sure to correctly make MAP specification for those two spaces.

For AM33 CPUs:

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

Properly make MAP specification for the case where the MMU is ON and the case where the MMU is OFF. First make MAP specification for the case where the MMU is ON (state in which the Linux OS is running). After that specification, put MAP specification for the MMU OFF state (state where the power has just come on or the system has just been reset).

By adding the above MAP specifications, PARTNER distinguishes the space that it accesses through the MMU from the space that it directly accesses.

As far as access to the MMU space from a debugger is concerned, only the space in which no access error occurs with attached loadable modules (drivers) and applications is allowed (it is automatically determined by a debugger). For that reason, a debugger cannot access other spaces.
Consequently, the behavior of PARTNER differs as shown below depending on whether or not a loadable module or an application is attached to a debugger.

【Processing for the module and application spaces when not attached】

- As for breakpoints, hardware breakpoints are automatically set.
  That means, you can set as many breakpoints as the maximum allowed number of hardware breakpoints.
- If the memory space you want to view has not been hit by the MMU, it cannot be viewed.
- Multiple applications are executed in the same virtual memory space. For that reason, a break might occur at elsewhere than the target breakpoint (in that case, keep the program running until reaching the target breakpoint by pressing the F5 key).

【Processing for the module and application spaces when attached】

- The breakpoints set are software breakpoints (there is no limit on the number of breakpoints).
- Even if the memory space you want to view has not been hit by the MMU, it can be viewed using the MMU auto-hit feature.
- Because software breaks are used, breaks occur in a specific application only.

4.2 Launch option

To support Linux debugging, the following launch options have been added or extended.

- "-OS option (Page 147)" *required for loadable module and application debugging.
- "-XGX option (Page 150)" *required for Linux debugging.
- "-!v option (Page 151)"
- "-!! option (Page 152)"
- "-SK option (Page 153)"
- "-RootDir option (Page 154)"
- "-MULTI option (Page 155)"
- "-OPTIMIZE option (Page 156)"
- "-[EUC|SJIS|UTF8] option (Page 157)"
- "-NPTL option (Page 158)"

-OS option

Format　**-OS <Debug mode>[,,4]**

Functionality

Specifies the debug mode.

Description

This option specification is required for Linux loadable module and application debugging.

It is not required when debugging only the kernel, but it is recommended that you always specify this option.

Specify as the argument the kernel mode, the application mode or the ADD mode.

It is recommended that you to specify "K2_LINUX_ADD" or "K2_LINUX_ADD_V26" as the mode.

Table 4-2 Debug mode

Debug mode	Description	Remarks
LINUX	Specify kernel mode.	In case Linux kernel 2.4 series
LINUX_ADD	Specify kernel ADD mode.	
LINUX_APP	Specify application mode.	
LINUX_APP_ADD	Specify application ADD mode.	

You can specify the following prefix and suffix for each debug mode.

Table 4-3 Debug mode additional strings

Additional setting	Description	Remarks
K2_ prefix	Utilizing the function of resolving extended virtual address resolving.	It makes possible to inspect the variables in the other (not current) process.
_V26 suffix	Specifying Linux kernel 2.6 series	Default setting is ver2.4 support.

It is recommended that you add the K2_ prefix under normal conditions. The extended virtual address resolution feature is realized by PARTNER referring to the virtual address conversion table in the Linux kernel. This means that this feature is highly dependent on the kernel; so turn it off if its use causes abnormal behavior. PARTNER automatically detects the Linux kernel version. Normally, debugging a 2.6.x kernel without specifying the _V26 suffix can be performed without any problem. On rare occasions, the automatic detection does not work properly. So if you know the version of the debug-target kernel, specify it.

If necessary, specify the implementation dependent value of the Linux kernel as the argument option [,,4] (excluding the square brackets). The size of the task_struct structure of the Linux kernel is 8192 bytes, and the offset value of the register set that is saved when switching contexts is coded as immediate data. Description may differ depending on the version and the CPU type. But in many cases, it is written as "<End of Task Structure> - <Offset Value>" (e.g. 8192 ? 4). Specify this offset value (in this example, it is 4) after two commas (,,).

In many cases, it is necessary to specify „4 for SH CPUs.
Usually it is not necessary to specify it for CPUs other than SH CPUs.
If you don't understand the usage, contact the KMC support.

【Examples】

```
-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 (For Linux kernels for most SH CPUs)
```

Table 4-4 summarizes the behavior of each debug mode.

Table 4-4 Behavior of each debug mode

	The way of multi-thread debugging	Behavior when application break is occurred
Kernel mode	Debug a single thread with a single debugger window	CPU stops
Kernel ADD mode	Debug multiple threads with a single debugger window	CPU stops
Application mode	Debug a single threads with a single debugger window	Only the thread of debug-target stops (kernel and other threads are executed)
Application ADD mode	Debug multiple threads with a single debugger window	Only the thread of debug target stops (kenel and other threads are executed)

If using PARTNER in Application Mode, the status bar of PARTNER varies as shown below, according to the target status.

Table 4-5 Status bar display in Application Mode

Target status	Status bar
Application is running	
Application is paused	
Linux kernel (CPU) is halted	 

-XGX option

Format　**-XGX <Pre-conversion PATH>,<Post-conversion PATH>**

Function

Resolves the path name included in debug information.

Description

This option specification is required for Linux debugging.

Set the debug information mode to the GNU C (stab,stab+,dwarf,dwarf-2) mode with path information.

Also, it has a mechanism that can convert the path information of a file to be loaded to, for example, an installation path in Samba.

You can specify this option a maximum of ten times.

【Examples】

Case where the path used for building the kernel is /home/foo/work/linux and /home/foo/work is installed in the Z: drive.

```
-XGX/home/foo/work/linux/,Z:¥linux¥
```

Case where the path used for building the kernel is /home/foo/work/linux and it is copied into c:¥work¥kernel¥linux.

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

-!v option

Format -!v

Functionality

Sets to open multiple PARTNER windows.

Description

It is necessary to specify this option when opening multiple PARTNER windows.

It is automatically added when launching PARTNER using the "-MULTI option (Page 155)" or the "MULTI command (Page 173)".

-!! option

Format **-!! <Initial PC Value>,<Kernel Option Symbol Name>=<Kernel Option String>**

Functionality

Sets the initial PC value.

Specifies the kernel option string.

Description

You can specify the initial PC (Program Counter) value and the kernel option.

<Initial PC Value> can be omitted. If <Initial PC Value> is omitted, the value specified in the downloaded file is enabled.

<Kernel Option String> writes a string in the memory specified by <Kernel Option Symbol Name> when loading the kernel object (vmlinux).

<Kernel Option Symbol Name> is a globally declared char array with a default value, and needs to be used as the default value of the kernel option in the kernel source.

If <Kernel Option Symbol Name> is omitted, one of the following default values is used.

Table 4-6 Kernel option symbol name

CPU	Default value	Recommended value
ARM	kzp01_arm_command_line	Specify --!!,default_cmdline=""
MIPS	arcs_cmdline	It vary depending on the board
SH	The address that added 0x100 to empty_zero_page	Default value is fine
AM33	cmdline	

【Example】

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

-SK option

Format **-SK <Loadable Module Path>**

Functionality

Specifies the path to the loadable module.

Description

Specifies the path information of a loadable module.

This is equivalent to the "Loadable module path" specification in the JETSET screen.

【Examples】

Case where the build directory of a loadable module is /home/foo/work/modules and /home/foo/work is installed in the Z: drive.

```
-SKZ:¥modules
```

This option is effective only when debugging is performed in the method described in "Debugging a loadable module (method 1) (Page 87)".

-RootDir option

Format **-RootDir <Root File System Path>**

Functionality

Resolves the path name of an application.

Description

This option specifies the path information of the root file system tree to resolve debug information of an application.

It specifies the path to the top directory of the root file system tree, which is viewed from Windows.

【Example】

Case where the top directory of the root file system tree is /opt/kmc/kzm-arm11/root and /opt is installed in the Z: drive.

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

-MULTI option

Format **-MULTI <number of windows to be opened>**

Functionality

Specifies the number of PARTNER windows to be opened.

Description

This option opens as many PARTNER windows as are specified. The maximum number that can be specified is 16.

【Reference】

"MULTI command (Page 173)"

-OPTIMIZE option

Format **-OPTIMIZE**

Functionality

Compensates the line numbers of a program for which optimized compiling has been performed.

Description

When debugging an object that has been compiled with optimization enabled (-02 etc.), the line number data is compensated so that a user can easily understand it.

-[EUC|SJIS|UTF8] option

Format 1 **-EUC**

Format 2 **-SJIS**

Format 3 **-UTF8**

Functionality

Sets kanji code for the code window.

Description

Sets the character code of the source file that is displayed in the code window. The default character code is SJIS.

Format 1 assumes that the character code of the source file is Japanese EUC (EUC kana is not supported).

Format 2 assumes that the character code of the source file is Shift-JIS.

Format 3 assumes that the character code of the source file is UTF-8.

【Reference】

"KANJI command (Page 179)"

-NPTL option

Format **-NPTL**

Functionality

Supports the NPTL method for multi-thread debugging.

Description

Specify this option when the pthread library in the target is compiled in the NPTL method.

This option is effective only when debugging a 2.6.x Linux.

Because it is possible to build a system in which the NPTL method and the Linux thread method are mixed, this option may be abolished in the future.

4.3 Additional command

To support Linux debugging, the following commands have been added or extended.

Commands dedicated to Linux debugging

・"PS command (Page 160)"
・"MAPS command (Page 161)"
・"ATTACH command (Page 162)"
・"THREAD command (Page 163)"
・"LINUX command (Page 164)"
・"INSMOD command (Page 166)"
・"PSID command (Page 168)"
・"TOP command (Page 171)"

Commands that have been expanded to support Linux debugging

・"L command (Page 172)"
・"MULTI command (Page 173)"
・"K command (Page 174)"
・"Q,EXIT command (Page 175)"
・"G command (Page 176)"
・"INS command (Page 177)"
・"SNAME command (Page 178)"
・"KANJI command (Page 179)"

PS command

Format **PS [/R]**

Functionality

Displays process states.

Description

This command displays processes of a running application. If you add the /R option, processes are displayed in descending order.

There are two Linux thread implementations: one is the "Linux thread method", in which threads are treated as a kind of a process, and the other is the "NPTL method", in which threads are treated as execution contexts within processes. If the NPTL method is used, threads within processes are indented when displayed.

【Example】

```
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 command

Format 1 **MAPS [<PID>]**

Format 2 **MAPS [<PID>],<Check Address>**

Functionality

Displays the memory information of an application.

Description

Format 1 displays the MAPS (the same information as /proc/<PID>/maps) of an application whose process ID was specified in <PID>. If <PID> is omitted when an application is attached to the current PARTNER, the MAPS information of that attached application is displayed.

Format 2 only displays the MAPS information that corresponds to the address specified in <Check Address>.

【Example】

```
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 command

Format 1 **ATTACH <PID>**

Format 2 **ATTACH <file name>**

Format 3 **ATTACH ADD**

Functionality

Attaches an application to a debugger.

Requirements

Linux must already be running in the target.

Description

Attaches an application that is running in the target system to PARTNER.

Format 1 specifies the application to attach as an integer in <PID>.

Format 2 searches <PID> using the specified file name and attaches to it.

Format 3 automatically searches and attaches to an unattached thread whose memory environment is equivalent to that of the thread being attached.

This command can be used only if the debug support file has been installed in the target as a preload library or inserted into glibc that is to be used in the target.

For details, refer to "Installing and configuring the debug support library (Page 64)"

【Example】

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

THREAD command

Format **THREAD**

Functionality

Displays thread status information.

Description

This command displays information about the PIDs (process ID), task_struct's (address of task information structures), and PCs (program counter) of all threads that have been attached to the current PARTNER.
This command is effective when debugging a multi-thread application in ADD mode.

【Example】

```
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 command

Format 1 LINUX module

Format 2 LINUX module <path to the ko module file>

Format 3 LINUX module clr

Format 4 LINUX load_so

Format 5 LINUX [ATTACH|ATTACH_AUTO|ATTACH_AUTO_SO|ATTACH_MANUAL]

Format 6 LINUX set_attach_offset <Library> <Offset Address>

Functionality

Manages files for Linux debugging in the target.

- Registers a debug-target loadable module for Linux 2.6.x, confirms registration information, and deletes registration information.
- Loads shared library information.
- Specifies how to treat the ELF format when attaching.
- Registers the offset information of a debug-target shared library.

Description

Format 1- Format 3 are for kernel module debugging. You can automatically start debugging on modules that have been registered using this command after the kernel launch, when the insmod command is executed on the target Linux.

The Format 1 displays the status indicating how the debug-target module is registered with PARTNER.

The Format 2 registers the module specified by the specified path with PARTNER. The maximum number of modules that can be registered is eight (you can change this as it is defined in the KMC patch in the Linux kernel).

The Format 3 deletes the registration of debug-target modules from PARTNER (deletes all registrations).

The Format 4 loads the debug information of shared libraries (.so) that is used by applications. It is effective only if "-RootDir option (Page 154)" is specified.

Information displayed by the "MAPS command (Page 161)" is used to search shared libraries to be loaded. Address information is not loaded for unsearchable libraries and memory that cannot be referred to.

The Format 5 sets how to load the ELF file when attaching, at the command line before attaching. Table 4-7 shows setting values that can be specified by the argument.

Table 4-7 Specifying how to treat the ELF file

LINUX command argument	Behavior	Remarks
ATTACH	Displays current loading mode	MANUAL, AUTO, AUTO_SO
ATTACH_AUTO	Loads debug information of executed file when attach (Default)	Only when "-RootDir option (Page 154)" is effective
ATTACH_AUTO_SO	Loads executed file and debug information of shared library when attach	Only when "-RootDir option (Page 154)" is effective
ATTACH_MANUAL	Doesn't load debug information when attach	

Use the Format 6 when using the debug support file as a library, to register its offset address information with PARTNER. This command is effective only when the debug support file is used as a PRELOAD library or is used after inserting it into glibc.

The Format 1 - Format 3 cannot coexist with "INSMOD command (Page 166)". (This is not a limitation of the debugger. But the patch excludes these combinations in order to maintain simple operation.)
The format 6 is dedicated to the debug support file. There is no normal use for this format.

【Example】

Format 1

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

Format 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 ↓
```

Format 3

```
PT>linux module clr ↓
```

Format 4

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

Format 5

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

Format 6

```
PT>linux set attach offset libkmcsup.so.2.0.0 0x00000624 ↓
```

INSMOD command

Format 1　INSMOD

Format 2　INSMOD [GET|CLR|CLRBP|CLRALL]

Functionality

Displays attachment information for loadable modules.

Controls manual attachment of loadable modules.

Requirements

A break must be generated in the target.

Description

This command performs attachment controls of loadable modules that have been installed using the insmod command on Linux.

The debug information must have been loaded for the debug-target loadable module when using this command.

Format 1 displays the memory range of a loadable module that is currently attached to PARTNER.

Format 2 provides extended capability for manual debugging of loadable modules (refer to "5.2 Manually debugging a loadable module Page 192)").

Table 4-8 shows operations that can be specified with the argument.

Table 4-8 Manual debugging operations for loadable modules

INSMOD command argument	Behavior
GET	Attach the Loadable module that is installed with insmod command on Linux to a debugger. This enables to set breakpoints and memory access in loadable module area.
CLR	Detach the loadable module that is attached with GET operation. However, the breakpoints that is attached in loadable module area with step of Format 2 are remaining.
CLRBP	Delete breakpoints attached with GET operation in Loadble module.
CLRALL	Complete clear. Detach loadable module and delete breakpoints with GET operation.

Format 2 is not used for normal debugging (it can be used only for manual debugging).

For information on how to perform manual debugging, refer to "5.2 Manually debugging a loadable module Page 192)".

【Example】

Format 1

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

Format 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 ↓
/* Installing a loadable module on Linux #insmod rd.o */
/* PARTNER breaks at the initialization entry of the loadable module (rd_init function) */
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 command

Format 1 PSID

Format 2 PSID ADD

Format 3 PSID NON_ADD

Format 4 PSID [GET|CLR|CLRBP|CLRALL]

Functionality

Displays information about the application/debug mode.

Controls the debug mode of PARTNER.

Controls manual attachment of applications.

Requirements

A break must be generated in the target.

Description

Format 1displays the used memory range and the process ID of an application attached to the current PARTNER. In ADD Mode, all IDs of attached processes are also displayed.

Format 2 and Format 3 control the debug mode of PARTNER.

Format 2 switches the mode of the current PARTNER to the ADD mode. (The same status as when it has been launched with -OS LINUX_ADD or -OS LINUX_APP_ADD specified as the launch option.)

Format 3 performs operation opposite to Format 2, and cancels the ADD mode of the current PARTNER.

【Reference】

"-OS option (Page 147)"

Format 4 provides extended capability for manual debugging of an application (refer to "5.3 Manually debugging an application Page 200)" and "5.4 Manual multi-process/multi-thread debugging Page 204)").
Table 4-8 shows operations that can be specified with the argument.

Table 4-9 Manual debugging operations for processes

PSID command argument	Behavior
GET	Attach the process of debug information that current PARTNER loads to PARTNER. This enables to access to the memory and set a breakpoint in process area. The timings for executing GET operation are; starting application (right after main() function), or set a breakpoint right after process/thread are generated, then execute debug-target application and a break occurred at the break point you set.
CLR	Detach the attached process with GET operation. However, the attached breakpoints with Format 2 in process area are remaining.
CLRBP	Delete the breakpoints with GET operation in process area.
CLRALL	Complete clear. Detach the attached process and delete the breakpoints with GET operation.

Format 4 is not used for normal debugging (it can be used only for manual debugging). For information on how to perform manual debugging, refer to "5.3 Manually debugging an application Page 200)" and "5.4 Manual multi-process/multi-thread debugging Page 204)".

【Example】

Format 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
(The displayed content depends on the CPU type and MAP field settings.)
PT>
```

Format 4

```
PT>ls u:¥appli¥sample ↓
PT>bp main ↓
PT>g ↓
/* Executes the application (Sample) on Linux */
/* PARTNER breaks at the main function of the application */
        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
(The displayed content depends on the CPU type and MAP field settings.)
PT>
```

If the application (sample) has been terminated, declare that the registered psid space has been deleted in PARTNER.

```
PT>PSID CLRALL ↓
```

TOP command

Format **TOP**

Functionality

Displays the memory usage status of running processes

Description

Displays a list of running processes and the memory usage status.

Although it resembles the top command of Linux (Unix), it does not perform real-time display update.

【Example】

```
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 command

Format **L[A] File Name[,/OFFS=Offset]**

Functionality

Loads the debug program and debug information.

Requirements

A break must be generated in the target.

Description

This is a basic command of PARTNER. This page describes the ″,/OFFS option″ that is useful for Linux debugging. To acquire knowledge about all the features of this command, refer to the Help of PARTNER.
The ″,/OFFS″ option is particularly useful for Linux kernels for ARM CPUs.

Normally, as for Linux kernels for ARM CPUs, the address that was assigned when vmlinux was linked (Virtual Address, hereinafter VA) and the physical address (PA) that is the address used when the kernel was actually assigned in the physical memory are different. It is therefore necessary to transfer vmlinux from the ICE to the target, with the difference between the VA and the PA of the kernel taken into account.

The ″,/OFFS option″ is a feature that adds the offset to the address that was linked to the load command, and transfers the binary data to the target.

【Example】

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

The above method specifies the offset at the debugger command line, you also can specify the ,/offs option in the dialog box of the load dialog.

Only a positive value (unsigned long) can be specified for the ,/offs option. For example, if the Linux kernel for KZM-ARM11-01 (ARM CPU) is used, the VA starts at C0008000 and its corresponding PA is 80008000, so set +C0000000 as the offset value to the VA (32 bit is the maximum and digits exceeding it are ignored).

MULTI command

Format **MULTI <number of PARTNER windows to be opened>**

Functionality

Opens multiple PARTNER windows.

Description

This command opens PARTNER windows as many as the number specified in <number of PARTNER windows to be opened>. The maximum number of windows that can be opened is 16.

The same operation can be done when launching PARTNER using the -MULTI launch option.

【Example】

```
PT>multi 3 ↓
PT>
```

【Reference】

"-MULTI option (Page 155)"

K command

Format 1　**K 0**

Format 2　**K <PID>**

Functionality

Displays the content of the stack frame (back trace).

Requirements

The argument option is only effective for a PARTNER window in which a process is being debugged.

Description

The conventional K command of PARTNER did not take any argument, but two types of formats that take an argument are newly added to support Linux debugging. This extended K command (with an argument) can be used when a break occurs in the kernel or another process.

The Format 1 displays in the command window the function call history of the user-side stack of the current process.
The Format 2 displays in the command window the function call history of the user-side stack of the PID that was specified in <PID>.

【Example】

```
PT>ls hellohello ↓
PT>BP .main+12 ↓
PT>g ↓
(Stops at a breakpoint.)
```

Format 1

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

Format 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 command

Format 1 **Q/A**

Format 2 **EXIT/A**

Functionality

Terminates all PARTNER windows.

Requirements

A break must be generated in the target.

Description

If you add the /A option to the Q or EXIT command, you can close all PARTNER windows that were opened using the "MULTI command (Page 173)" or the "-MULTI option (Page 155)".

【Example】

PT>q/a ↓

【Reference】

"MULTI command (Page 173)","-MULTI option (Page 155)"

G command

Format **G/A**

Functionality

Executes a program.

Requirements

A break must be generated in the target.

Description

The /A command option executes the kernel and all processes and threads, when manually debugging a multi-process/multi-thread application in Kernel Mode.

INS command

Format 1 **INS LINUX_TASK:<PID>**

Format 2 **INS LINUX_REG:<PID>**

Functionality

Performs inspection display of the task information/register information of processes.

Requirements

A break must be generated in the target.

Description

This command is effective only for PARTNER in which the debug information of the Linux kernel has been loaded using the INS command expanded to support Linux.

The Format 1 displays in the inspection window the task information (task_struct) of the process specified in <PID>.

The Format 2 displays in the inspection window the register information (regs) of the process specified in <PID>.

SNAME command

Format **SNAME**

Functionality

Displays the path to the source displayed in the code window.

Requirements

A break must be generated in the target.

Description

This command displays the path to the source that is currently displayed in the code window.

The displayed source path is the same as the one displayed in the caption in the code window.

【Example】

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

KANJI command

Format **KANJI [EUC|SJIS|UTF8]**

Functionality

Sets the kanji code of the source file that is displayed in the code window.

Description

Specifies the kanji code of the source file that is displayed in the code window of PARTNER.

If nothing is specified, the current kanji code is displayed.

This command newly supports UTF8 for Linux debugging.

【Example】

```
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>
```

【Reference】

"-[EUC|SJIS|UTF8] option (Page 157)"

Chapter 5 Special debugging methods

This chapter describes debugging methods for various situations.

5.1 Debugging in Application Mode

For application debugging, PARTNER provides two debug modes, namely the application mode and the kernel mode. Compared to the kernel mode, the application mode is for special use, and used when you want to generate breaks at each process without stopping the CPU. This section describes how to perform debugging in Application Mode.

5.1.1 Configuration required for debugging

Table 5-1 Configuration settings for application mode debugging

Setting condition / Debug target	Kernel menu						-OS option			
	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	Kernel mode	Kernel ADD mode	Application mode	Application ADD mode
Application	◎						— *1		○	○

◎ : Required, ○ : Recommended, △ : Not recommended, × : Not permitted, Empty: Either is OK

*1 This section describes cases where the application mode is specified. For information on kernel mode debugging, refer to "3.4 Debugging an application (Page 99)", "3.5 Debugging a multi-thread application (Page 106)", and "3.6 Debugging a multi-process application (Page 116)".

If you have not yet performed "2.2 Modifying and configuring the Linux kernel source (Page 54)", you cannot perform application debugging described below. If you want to use a kernel that is not modified, refer to "5.3 Manually debugging an application (Page 200)" and "5.4 Manual multi-process/multi-thread debugging (Page 204)".

5.1.2 Procedure for debugging an application in Application Mode

To perform debugging in Application Mode, specify the "Application (NON_ADD) Mode" or "Application ADD Mode" as the OS debug mode (refer to "-OS option (Page 147)") in the launch configuration of PARTNER (refer to "Configuring and launching PARTNER (Page 75)").
For information on how to debug an application in Kernel Mode, refer to "3.4 Debugging an application (Page 99)".

This section describes the procedure for the debugging method of an application (process) in Application Mode, in the following order, using the case where a sample (sample) is used as an example.

(1) "Procedure for starting debugging (Page 184)"

(2) "Debugging an application (Page 184)"

(1) Procedure for starting debugging

The procedures for compiling an application, starting up PARTNER and attaching by generating a hardware break at the main function are exactly the same as those used in "Debugging an application (Page 99)".

While execution is stopping at a hardware breakpoint, the CPU also stops. Immediately after attachment, the status displayed in the application PARTNER window changes to "Application is stopping (kernel)". At this point, the CPU is still stopped and so you cannot set software breakpoints in the application. When you attempt to execute one step, the status changes to "Executing the target" in the kernel window and to "Application stopped" in the application window", and you are then able to set software breakpoints in the application.

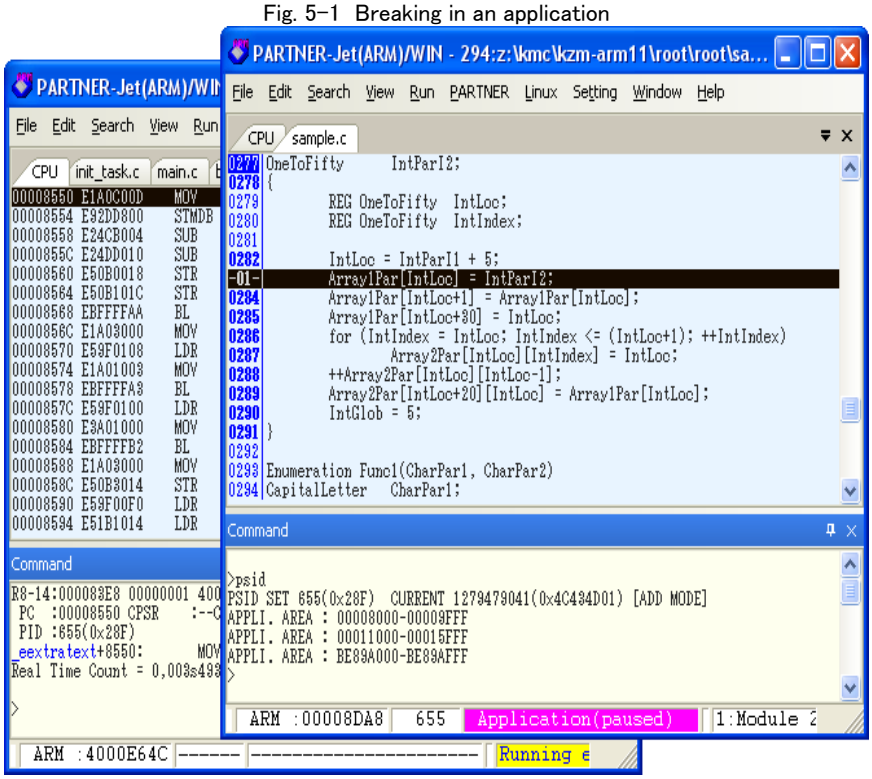
If you want to generate a break at the main function in the application only, and do not want to stop the kernel, you need to insert stub functions for debugging in the application. For details, refer to "Application direct link method (Page 66)".

---

(2) Debugging an application

Once attached to the process using the "ATTACH command (Page 162)", you can debug the process. It is different from the kernel mode in that the kernel runs while a break is generated in an application process. The CPU stops while a hardware break is generated at the main function.

Fig. 5-1 Breaking in an application



If you want to operate PARTNER in Application Mode, the status bar of PARTNER changes as shown in ""Status bar display in Application Mode (Page 149)"" according to the target status.

5.1.3 Procedure for multi-context debugging in Application Mode

The debugging procedure for applications that use fork() or pthread is nearly the same as the procedure previously described in "5.1.2 Procedure for debugging an application in Application Mode (Page 183)". Only kernel configuration, application creation and launch option specification, and multi-window debugging are different.

For debugging multi-process/multi-thread applications, you can use the ADD mode, in which you debug each process/thread in one PARTNER window, and the NON_ADD mode, in which you debug each in separate PARTNER windows.
You can switch between the ADD mode and NON_ADD mode using the"PSID command (Page 168)" or by specifying the "-OS option (Page 147)".

This section describes the procedure for the debugging method of an application (process) in Application Mode, using the case where samples (fork and thread_sample) are used as examples.

(1) "Procedure for starting debugging (Page 186)"

(2) "Breaking using PARTNER (Page 186)"

(1) Procedure for starting debugging

The procedures for compiling an application, starting up PARTNER and attaching by generating a hardware break at the main function are exactly the same as those used in "Debugging an application (Page 99)".

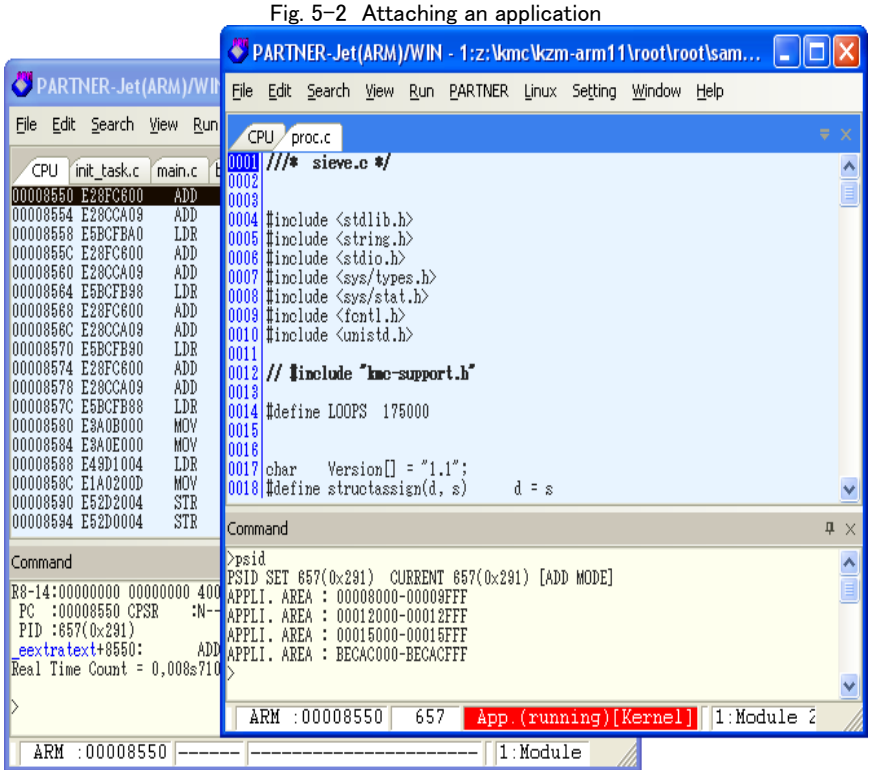
While execution is stopped at a hardware breakpoint, the CPU also stops. Immediately after attachment, the status displayed in the application PARTNER window changes to "Application is stopped (kernel)". At this point, the CPU is still stopped and so you cannot set software breakpoints in the application. When you attempt to execute one step, the status changes to "Executing the target" in the kernel window and to "Application stopped" in the application window, and you are then able to set software breakpoints in the application.
If you want to generate a break at the main function in the application only and do not want to stop the kernel, you need to insert stub functions for debugging in the application. For details, refer to "Application direct link method (Page 66)".

(2) Breaking using PARTNER

● When the ADD mode is used.

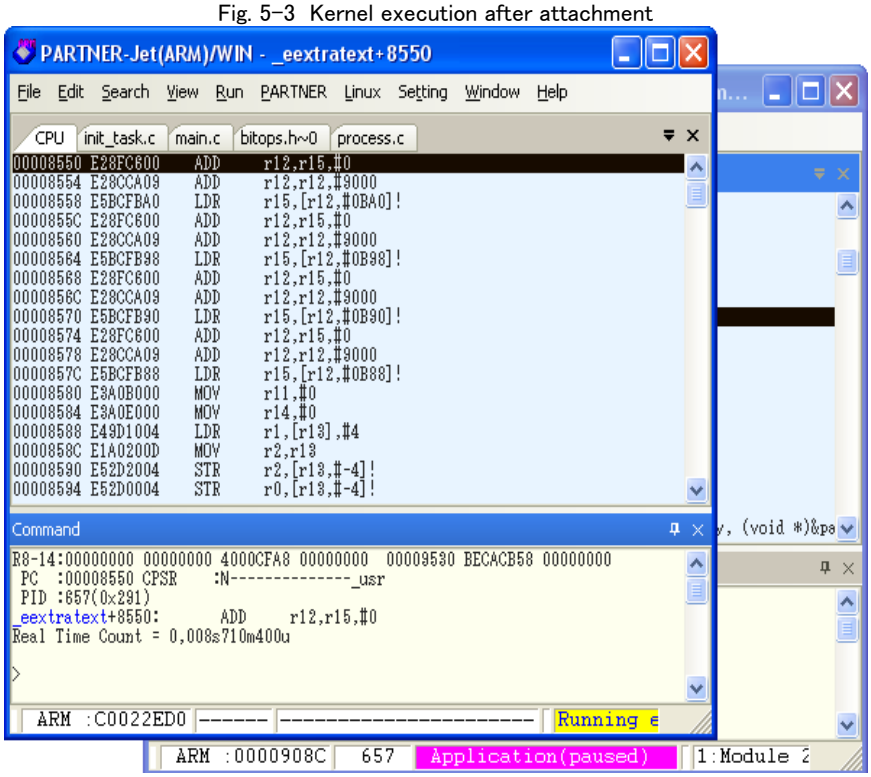
Generate a hardware break at the main function, and put it into the state where breakpoints can be set once attachment has been performed.main

Fig. 5-2 Attaching an application



Move one step forward using Trace (F8).

Fig. 5-3 Kernel execution after attachment

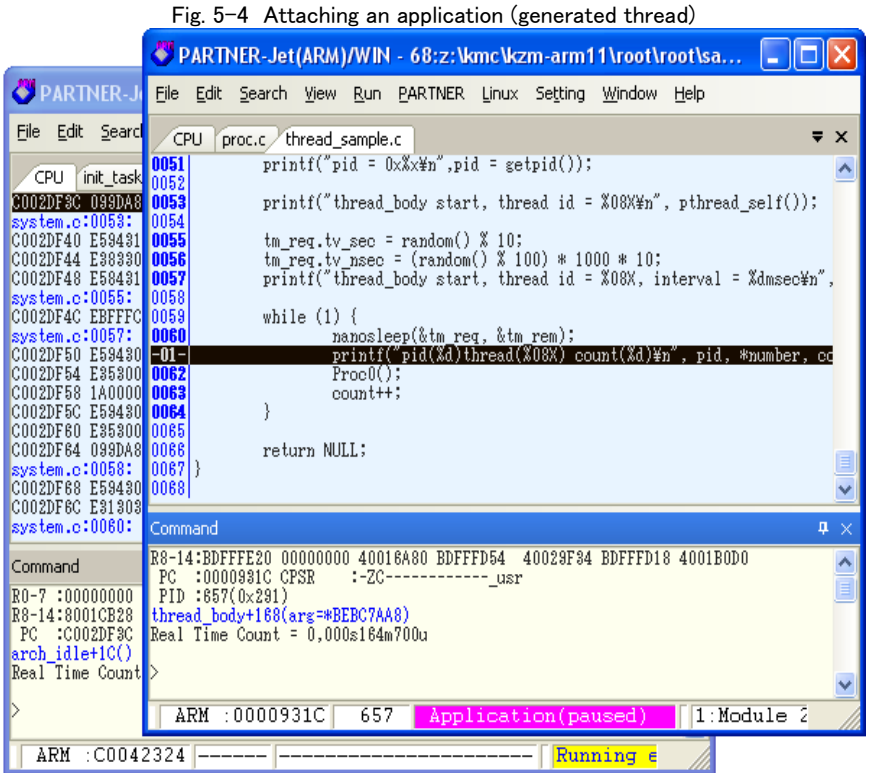


Confirm whether or not the application is attached to the debugger using the "PSID command (Page 168)".

```
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
(The displayed content depends on the CPU type and MAP field settings.)
```

If you set a breakpoint in the thread_body() function and execute the application at this point, the generated thread is automatically attached and a break occurs.

Fig. 5-4 Attaching an application (generated thread)



You can confirm that the generated thread is attached in the application PARTNER window, by executing the "PSID command (Page 168)".

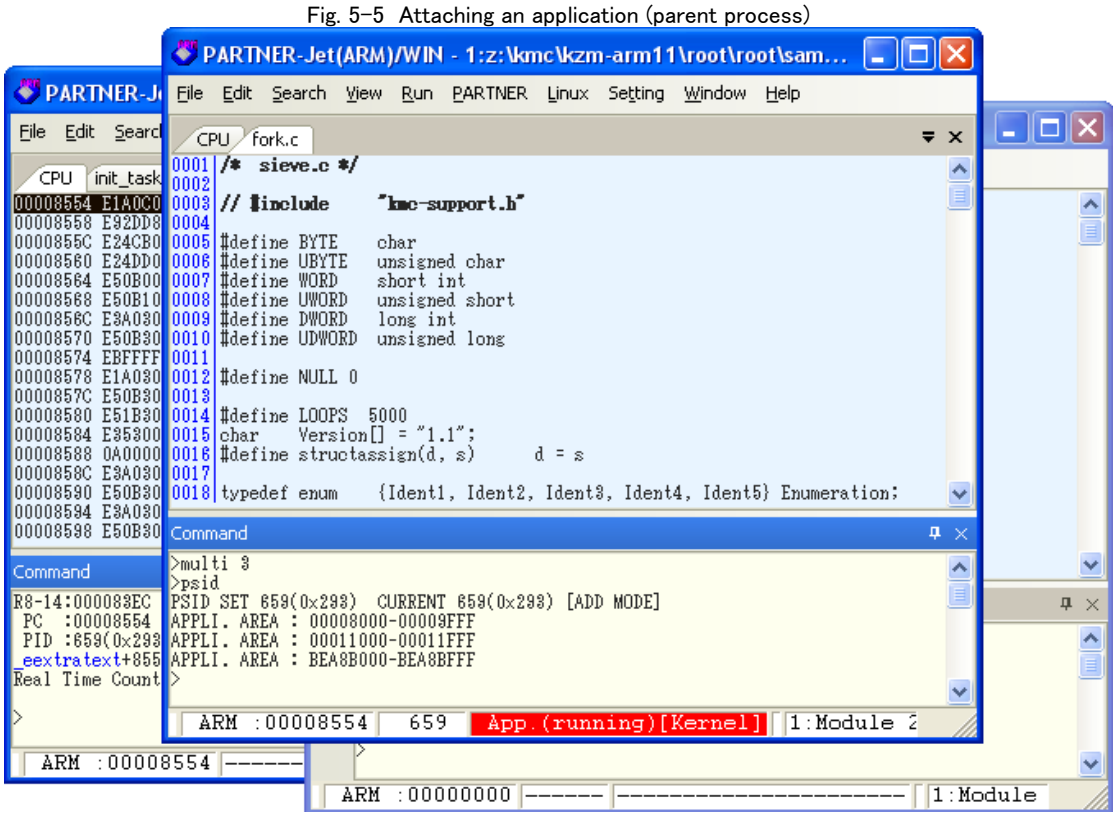
```
PT>psid ↓
PSID SET 742(0x2E6)  CURRENT 1129120076(0x434D014C) [ADD MODE 740]
APPLI. AREA : 00008000-00009FFF
APPLI. AREA : 00012000-00012FFF
APPLI. AREA : 00015000-00017FFF
APPLI. AREA : BE3FF000-BEA6FFFF
APPLI. AREA : 40016000-40016FFF
(The displayed content depends on the CPU type and MAP field settings.)
```

Thereafter, generated threads will be attached to the application PARTNER window, and you will be able to perform normal debugging operations on the attached threads such as viewing/changing thread memory and setting breakpoints.

● When the NON_ADD mode is used

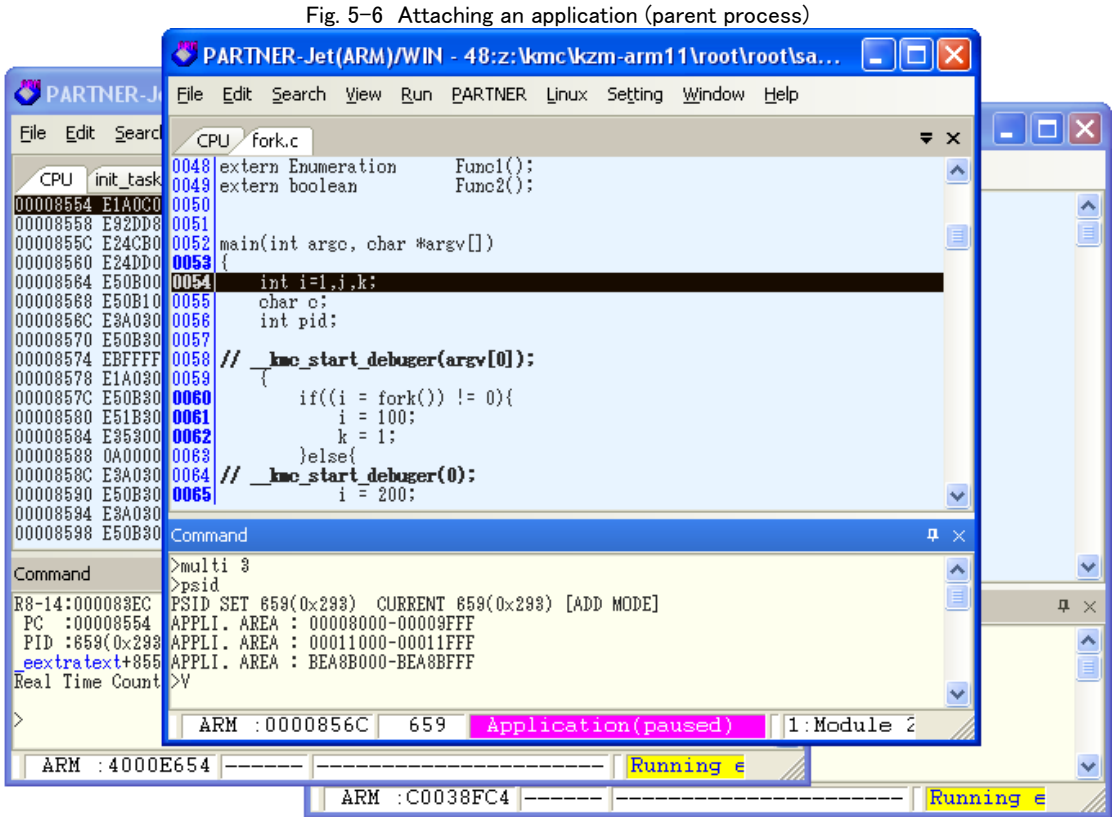
Generate a hardware break at the main function, and put it into the state where breakpoints can be set once attachment has been performed.

Fig. 5-5 Attaching an application (parent process)



Move one step forward using Trace (F8).

Fig. 5-6 Attaching an application (parent process)

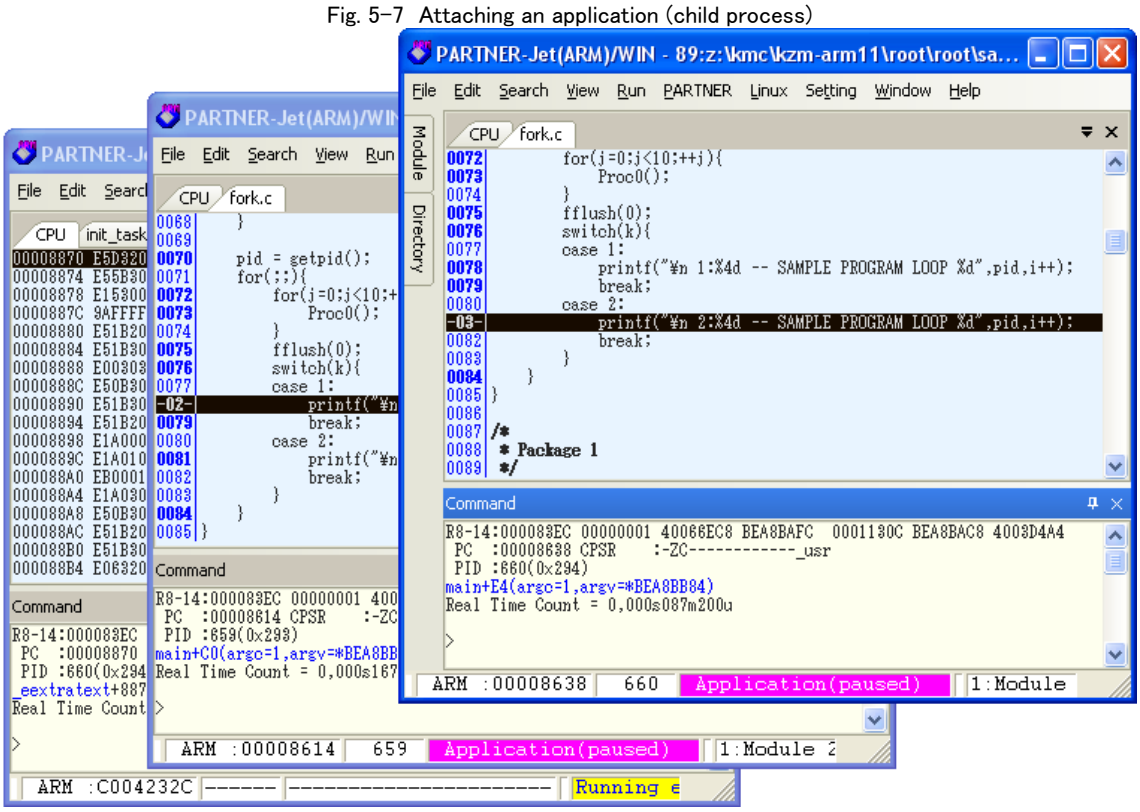


You can confirm that the generated thread is attached in the application PARTNER window, by executing the "PSID command (Page 168)".

```
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
(The displayed content depends on the CPU type and MAP field settings.)
```

Once the fork() function is executed in the PARTNER window for the application and a child process is generated, a break occurs in the PARTNER window for the child process. At that time, PARTNER automatically collects the PID and the placement information of the child process, and thereafter you can view the memory in the child process area and set breakpoints (note that, in Fig. 5-7, a break occurs in the child process window on the right without stopping at a software breakpoint set at the child process side of the fork() function in the window for the parent process at the center).

Fig. 5-7 Attaching an application (child process)



You can confirm that the child process is attached in the PARTNER window for the application, by executing the "PSID command (Page 168)".

```
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
(The displayed content depends on the CPU type and MAP field settings.)
```

If PARTNER does not break properly, the kernel configuration may not be correct. Makes sure that the OS debug mode, which was specified in "2.2.5 Kernel configuration (Page 57)" and "2.4.2 Configuring and launching PARTNER (Page 75)", is in Application Mode.
Also confirm that the support library (libkmcsup.so.2.0.0) has been preloaded.

5.2 Manually debugging a loadable module

This section describes the procedure for debugging a loadable module used when you cannot modify the kernel or loadable modules.

5.2.1 Configuration required for debugging

Table 5-2 Configuration settings for loadable module manual debugging.

Setting condition / Debug target	Kernel menu						-OS option			
	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	Kernel mode	Kernel ADD mode	Application mode	Application ADD mode
Loadable module	◎						○	○	×	

◎ : Required, ○ : Recommended, △ : Not recommended, × : Not permitted, Empty: Either is OK

You cannot manually debug loadable modules in Application Mode.

5.2.2 Debugging Procedure

This section describes the debugging method of loadable modules in the following order, using the case where a RAM disk (rd.o) is used as an example.

(1) "Creating a loadable module (Page 193)"

(2) "Launching the Linux kernel (Page 193)"

(3) "Creating on the target the MAP file for a loadable module (Page 193)"

(4) "Setting breakpoints (Page 196)"

(5) "Installing a loadable module (Page 196)"

(6) "Breaking using PARTNER (Page 196)"

(7) "Attaching a loadable module (Page 197)"

(8) "Detaching a loadable module (Page 198)"

(1) Creating a loadable module

Create the debug-target loadable module with debug information.

(2) Launching the Linux kernel

Follow the procedure described in "Launching the debug environment (Page 73)" to build a debug environment, and launch the Linux kernel.

(3) Creating on the target the MAP file for a loadable module

Install the loadable module (rd.o) on the target Linux using the insmod command, and create the MAP file (rd.map) using the -m option of the insmod command.
The name of the MAP file to be created should be the name of the loadable module file to be installed and specify ".map" as the file extension. Also, it should be created in a directory in which the loadable module file is stored.

```
TGT>insmod -m rd.o >rd.map ↓
```

Fig. 5-8 Creating the MAP file for a loadable module

```
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:~#
```

The -m option of the insmod command can only be used for modules compatible with 2.4.x kernels. If your kernel is a 2.6.x kernel, you cannot use the -m option unless it supports conventional format modules.

After you have created the MAP file, use the rmmod command to unload the loadable module (rd.o).

```
TGT>rmmod rd ↓
```

Fig. 5-9 Unloading a loadable module

```
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:~# rmmod rd
root@kzp-arm:~#
```

PARTNER automatically calculates the relocation destination using the initial address of the module loaded from the MAP file and its size.

Consequently, even if you modified the content of the module, it is not necessary to create the MAP file again unless the initial address of the module is changed. But the following cases are exceptions.

You need to create the MAP file again in the following cases:

- If you changed the order for installing (insmod) the module
- If the module size varied by 4 KB or more
- If the kernel size varied by 4 KB or more

(4) Loading debug information for a loadable module

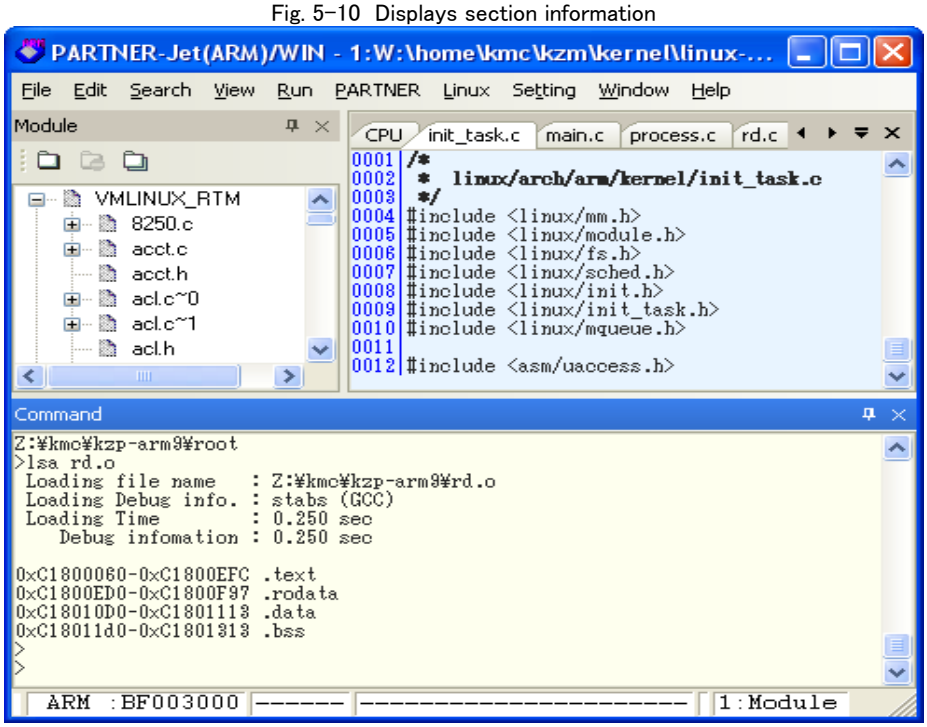
Stop the running kernel by pressing the ESC key and load into PARTNER the debug information for the loadable module.

Make sure to check [Symbol Only] and [Append] when loading it.

```
PT>lsa rd.o ↓
```

Once the debug information has normally been loaded, addresses are displayed for each section of the loadable module (rd.o). At this point, you still cannot view the memory of the loadable module.

Fig. 5-10 Displays section information



If an error message "The specified file does not exist" is displayed when loading the debug information, the MAP file is not in the same directory as the loaded loadable module. Follow the procedure described in (2) Launching the Linux kernel (Page 193) to create the MAP file again.

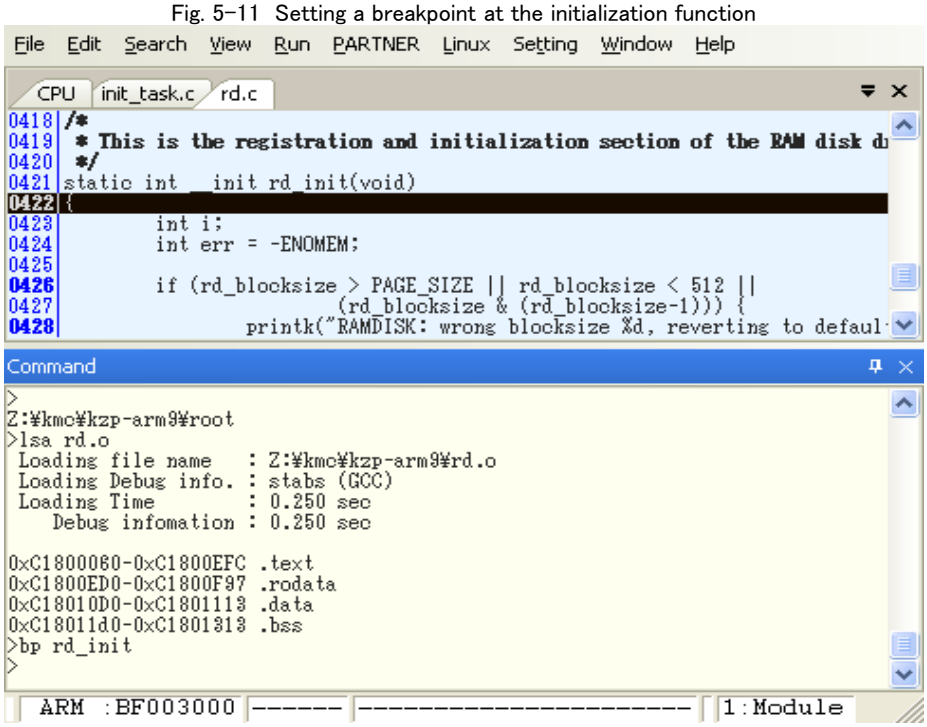
(5) Setting breakpoints

Set a breakpoint in the initialization part (rd_init function) of the loadable module (rd.o).

Set only one breakpoint.

PT>bp rd_init ↓

Fig. 5-11 Setting a breakpoint at the initialization function



If the "Verify error" message appears when setting a breakpoint, the maximum allowed number of breakpoints has already been set (the number of breakpoints that can be set depends on the CPU). Set a breakpoint again at the initialization part of the loadable module after deleting one of the already-set breakpoints.

(6) Installing a loadable module

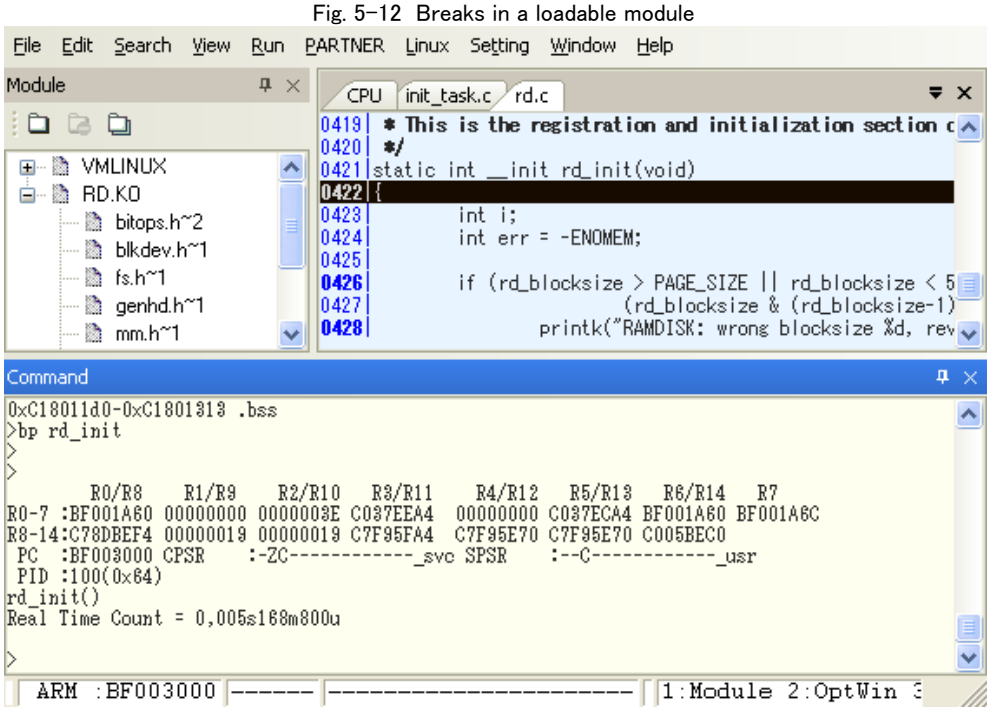
Execute the kernel, and install the debug-target loadable module into the target system.

TGT>insmod rd.o ↓

(7) Breaking using PARTNER

Once the debug-target loadable module has been installed in the target system, PARTNER breaks at the position at which the breakpoint was set.

Fig. 5-12 Breaks in a loadable module



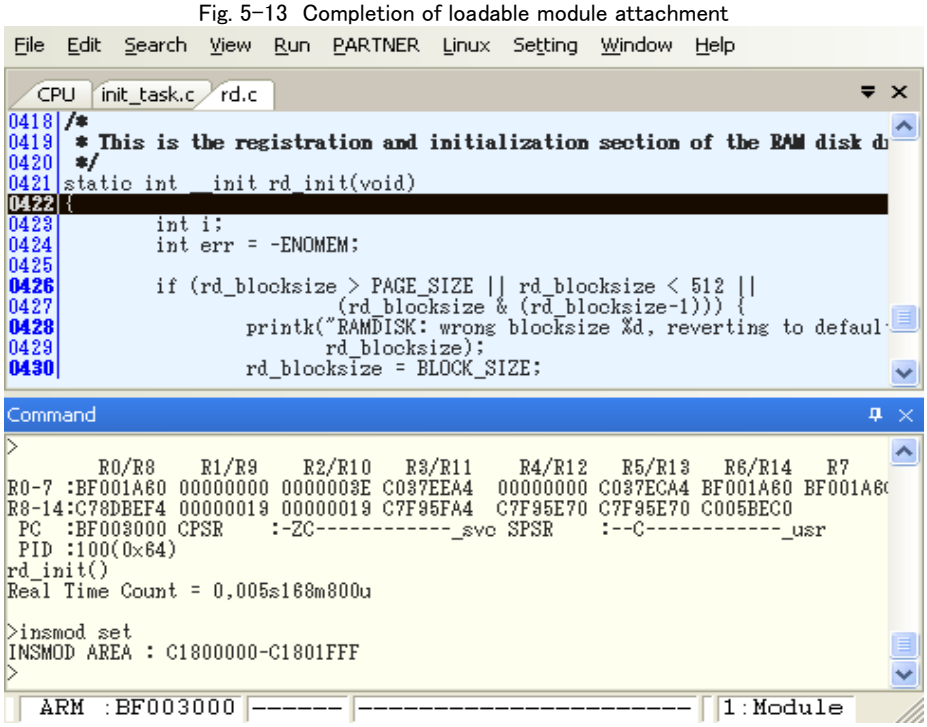
(8) Attaching a loadable module

Use the "INSMOD command (Page 166)" to attach the loadable module (rd.o).

```
PT>INSMOD GET ↓
```

If the command has normally been completed, and the space of the loadable module (rd.o) is displayed, you can view the memory from the source code and use software breaks.

Fig. 5-13 Completion of loadable module attachment



(9) Detaching a loadable module

Before unloading the loadable module from the target using the rmmod command, you need to detach the loadable module attached to PARTNER.

Use the "INSMOD command (Page 166)" to detach to the loadable module.

```
PT>INSMOD CLRALL ↓
```

Once this command has been normally completed, you can execute the rmmod command on the target system.

```
TGT>rmmod rd ↓
```

Fig. 5-14 Unloading a loadable module

```
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:~#
```

If the module had not already been detached from PARTNER before unloading the module using the rmmod command, a kernel panic occurs.

5.3 Manually debugging an application

This section describes the procedure for debugging an application when you cannot modify the kernel or applications.

5.3.1 Configuration required for debugging

Table 5-3 Configuration settings for manual application debugging

Setting condition / Debug target	Kernel menu						-OS option			
	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	Kernel mode	Kernel ADD mode	Application mode	Application ADD mode
Application	◎						○	○	×	

◎ : Required, ○ : Recommended, △ : Not recommended, × : Not permitted, Empty: Either is OK

You cannot perform manual application debugging in Application Mode.

5.3.2 Debugging Procedure

This section describes the procedure for the debugging of an application in the following order, using the case where a sample (sample) is used as an example.

(1) "Creating an application (Page 201)"

(2) "Preparing for debugging (Page 201)"

(3) "Opening a PARTNER window for an application (Page 201)"

(4) "Loading application debug information (Page 201)"

(5) "Setting breakpoints (Page 201)"

(6) "Executing an application (Page 202)"

(7) "Breaking using PARTNER (Page 202)"

(8) "Attaching an application (Page 202)"

(1) Creating an application

Create the debug-target application with debug information.

(2) Preparing for debugging

Refer to "Launching the debug environment (Page 73)", and execute the kernel from PARTNER in Kernel Mode.

(3) Opening a PARTNER window for an application

Open a new PARTNER window for the application using the "MULTI command (Page 173)".

```
PT>MULTI 2 ↓
```

(4) Loading application debug information

Stop the running kernel by pressing the ESC key and load into PARTNER the debug information for the application in the PARTNER window for the application.
Make sure to check [Symbol Only] when loading it from the GUI menu.

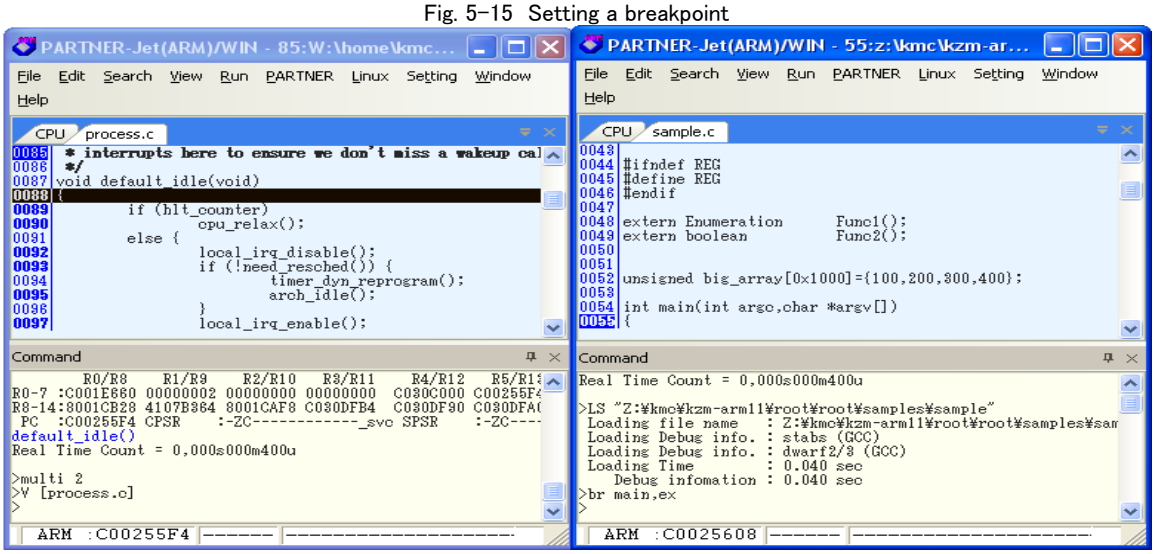
```
PT>ls sample ↓
```

(5) Setting breakpoints

Set a breakpoint at the main() function of the application.

```
PT>br main.ex ↓
```

Fig. 5-15 Setting a breakpoint



An executable hardware break (br) is more reliable than a software break (bp). When a bp is used, PARTNER sets a hardware break at an address that was obtained from the debug information. When a bp is used, it is safer to set only one breakpoint. If the "Verify error" message appears when setting a breakpoint, the maximum allowed number of breakpoints has already been set (the number of breakpoints that can be set depends on the CPU).
In that case, delete one of the already-set breakpoints and set it again.

(6) Executing an application

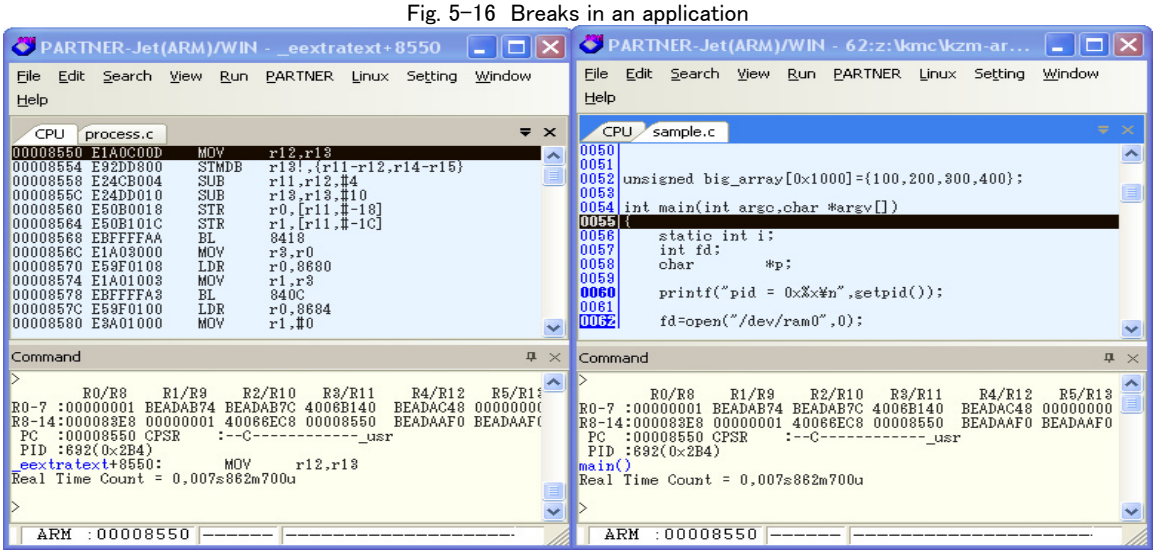
Execute the kernel, and then, execute the debug-target application on the target system.

```
TGT>./sample ↓
```

(7) Breaking using PARTNER

If the debug-target application is executed in the target system, PARTNER breaks at the position at which the breakpoint was set.

Fig. 5-16 Breaks in an application



The address of the main symbol might be used by multiple processes, so a break might occur in another program when executing the target program.
In that case, keep the program running (by pressing the [F5] key) until a break occurs in the target application.

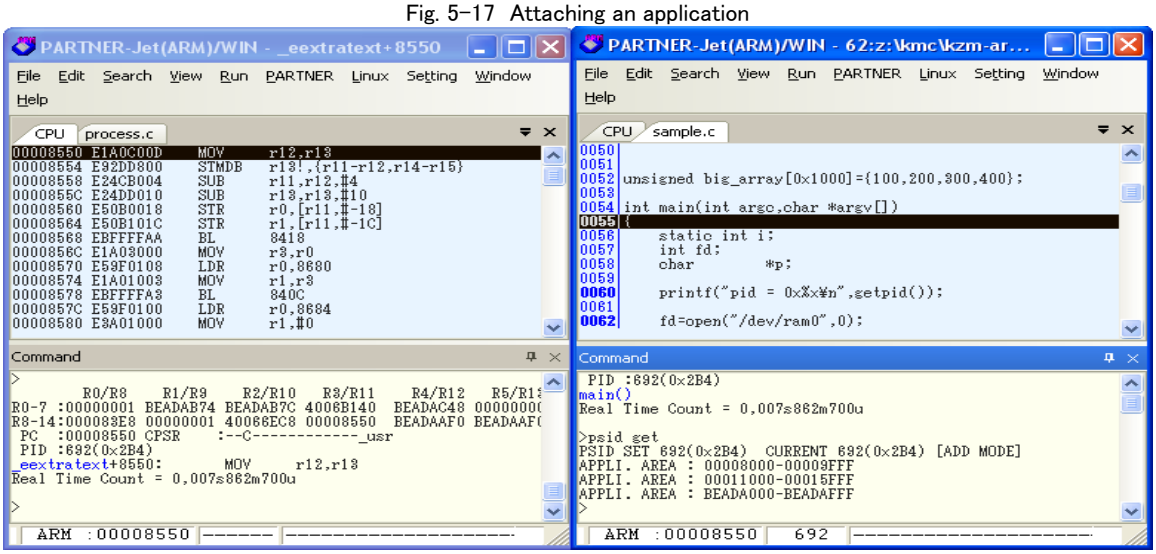
(8) Attaching an application

Attach the application (sample) using the "PSID command (Page 168)".

```
PT>PSID GET ↓
```

If the command has normally been completed, and the space of the application (sample) is displayed, you can view the memory from the source code and use software breaks.

Fig. 5-17 Attaching an application



If the application (sample) has been terminated, declare that the registered psid space has been deleted in PARTNER.

```
PT>PSID CLRALL ↓
```

5.4 Manual multi-process/multi-thread debugging

This section describes the procedure for debugging an application used when you cannot modify the kernel or applications.

5.4.1 Configuration required for debugging

Table 5-4 Configuration settings for manual multi-context application debugging

Setting condition / Debug target	Kernel menu						-OS option			
	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	Kernel mode	Kernel ADD mode	Application mode	Application ADD mode
Application	◎						○	○	×	

◎ Required, ○ : Recommended, △ : Not recommended, × : Not permitted, Empty: Either is OK

You cannot manually debug multi-process/multi-thread applications in Application Mode.
Also, if two or more threads or processes execute the command address at which a breakpoint was set, the debugger might not handle it properly.
1) If another thread or process executes a breakpoint at which the target application is stopped.
2) If the address at which a breakpoint was set was executed by a process or a thread that has not been attached to the debugger.

5.4.2 Debugging Procedure

This section describes the procedure for the debugging of an application in the following order, using the case where a sample (fork) is used as an example.

(1) "Creating an application (Page 206)"

(2) "Executing the kernel (Page 206)"

(3) "Opening a PARTNER window for an application (Page 206)"

(4) "Loading application debug information (Page 206)"

(5) "Setting breakpoints in the parent process (Page 207)"

(6) "Executing an application (Page 207)"

(7) "Breaking using PARTNER (Page 207)"

(8) "Attaching the parent process (Page 208)"

(9) "Setting breakpoints in a child process (Page 208)"

(10) "Re-executing an application (Page 209)"

(11) "Breaking using PARTNER (Page 209)"

(12) "Attaching a child process (Page 209)"

(1) Creating an application

Create the debug-target application with debug information.

(2) Executing the kernel

Refer to "Launching the debug environment (Page 73)", and execute the kernel from PARTNER in Kernel Mode.

(3) Opening a PARTNER window for an application

Open application PARTNER windows using the "MULTI command (Page 173)" as many as the number of processes and the kernel. In the case of this sample (fork), open three PARTNER windows.

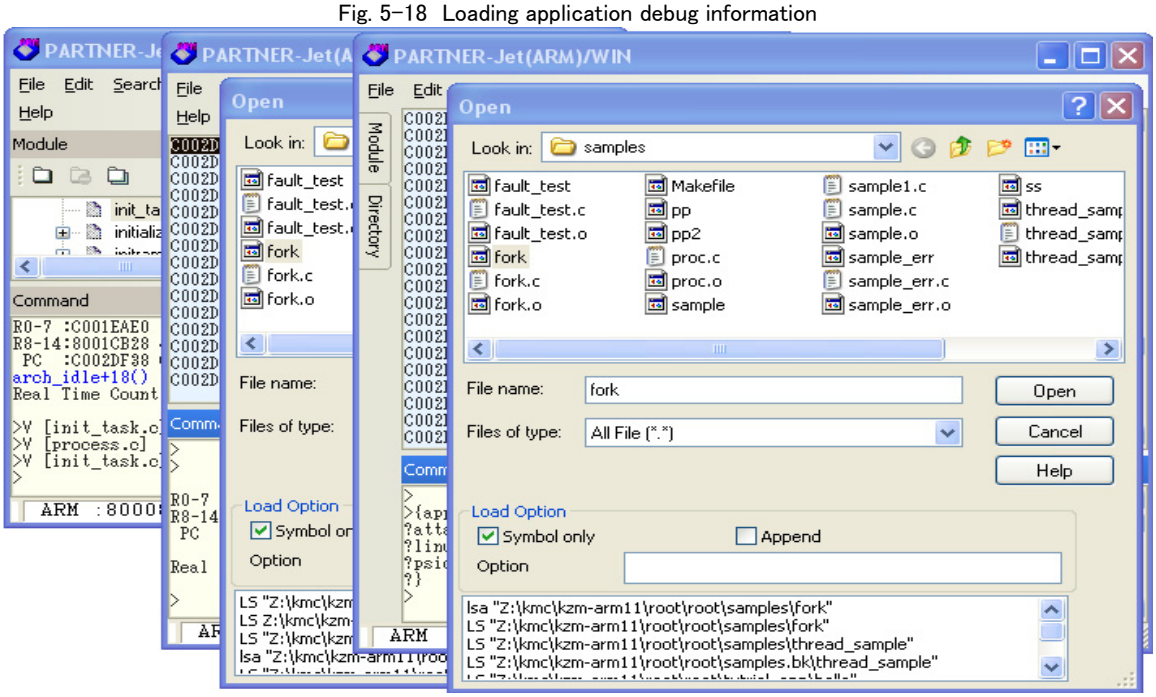
```
PT>MULTI 3 ↓
```

(4) Loading application debug information

Stop the running kernel by pressing the ESC key and, in the parent process PARTNER window and the child process PARTNER window, load the debug information of the application into each PARTNER window.
Make sure to check [Symbol Only] when loading it.

```
PT>ls fork ↓
```

Fig. 5-18 Loading application debug information



If you want to attach a thread generated from a shared library, you also need to load the debug information for the shared library. For information on how to load debug information for a share library, refer to "3.7 Debugging a shared library (Page 122)".

(5) Setting breakpoints in the parent process

Set a breakpoint at the main() function of the parent process.

```
PT2>br main.ex ↓
```

An executable hardware break (br) is more reliable than a software break (bp). When a bp is used, PARTNER sets a hardware break at an address that was obtained from the debug information. When a bp is used, it is safer to set only one breakpoint. If the "Verify error" message appears when setting a breakpoint, the maximum allowed number of breakpoints has already been set (the number of breakpoints that can be set depends on the CPU).
In that case, delete one of the already-set breakpoints and set it again.

(6) Executing an application

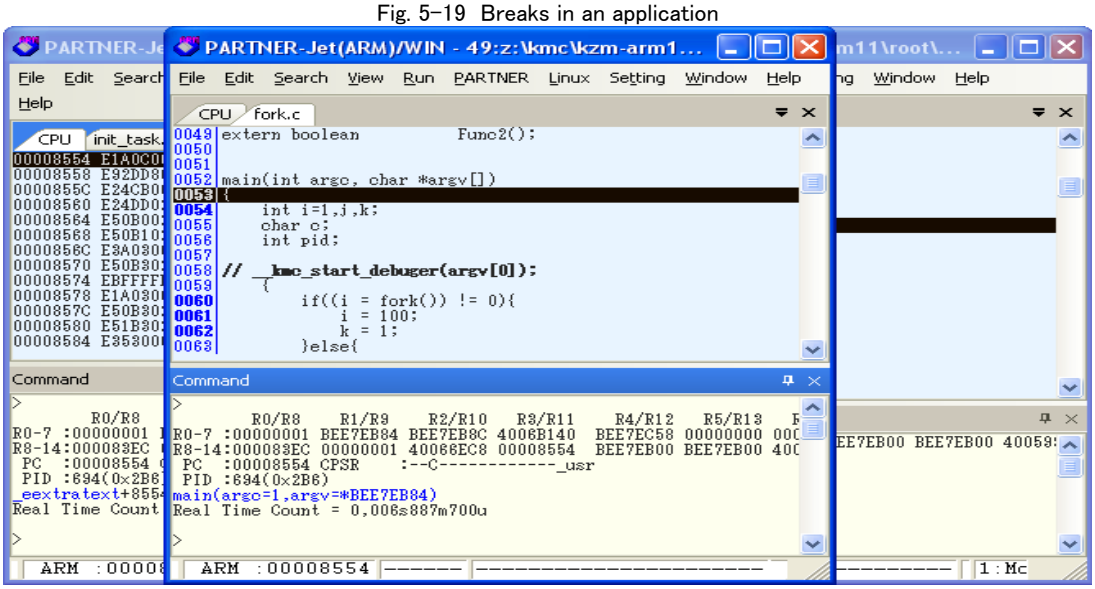
Execute the kernel again, and then, execute the debug-target application on the target system.

```
TGT>./fork ↓
```

(7) Breaking using PARTNER

Once the debug-target application has been installed on the target system, PARTNER breaks at the position at which the breakpoint was set.

Fig. 5-19 Breaks in an application



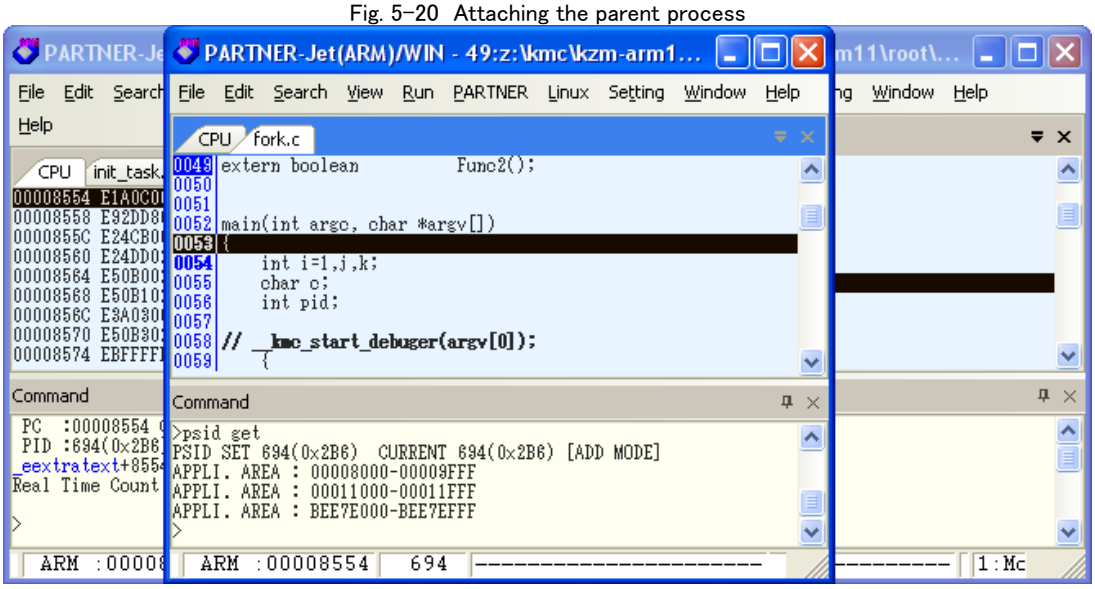
The address of the main symbol might be used by multiple processes, so a break might occur in another program when executing the target program. In that case, keep the program running (by pressing the F5 key) until a break occurs in the target application.

(8) Attaching the parent process

Execute the"PSID command (Page 168)" in the PARTNER window for the parent process, to attach the parent process.

```
PT2>PSID GET ↓
```

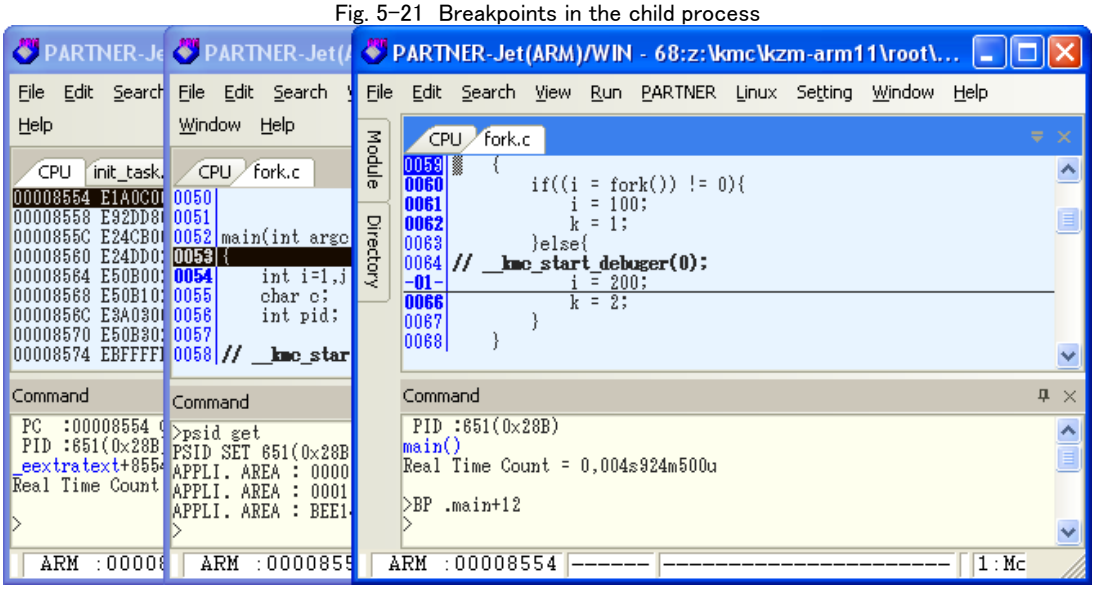
Fig. 5-20 Attaching the parent process



(9) Setting breakpoints in a child process

In the PARTNER window for the child process, set a breakpoint at the return address on the child process side in the part that calls the fork() system call. When debugging a multi-thread application, set a breakpoint at the head of each thread.

Fig. 5-21 Breakpoints in the child process



If the "Verify error" message appears when setting a breakpoint, the maximum allowed number of breakpoints has already been set (the number of breakpoints that can be set depends on the CPU). Breakpoints set in space that is attached to PARTNER using

PSID GET are not included in the number of breakpoints that can be set. In that case, delete one of the already-set breakpoints and set it again.

---

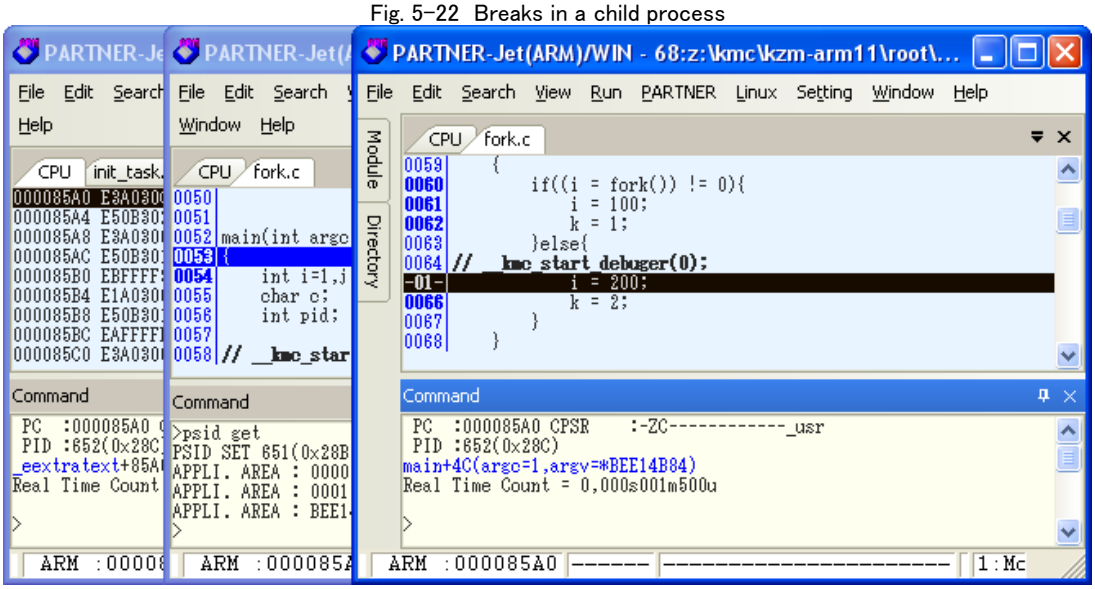
(10) Re-executing an application

Restart the application in the PARTNER window for the parent process.

(11) Breaking using PARTNER

Once a child process has been created, PARTNER breaks at the set breakpoint.

Fig. 5-22 Breaks in a child process



PARTNER sets a hardware break at an address that was obtained from the debug information. This address may also be in use by another program, so a break could occur in another process when executing the program. In that case, keep the program running (by pressing the F5 key) until a break occurs in the target application.

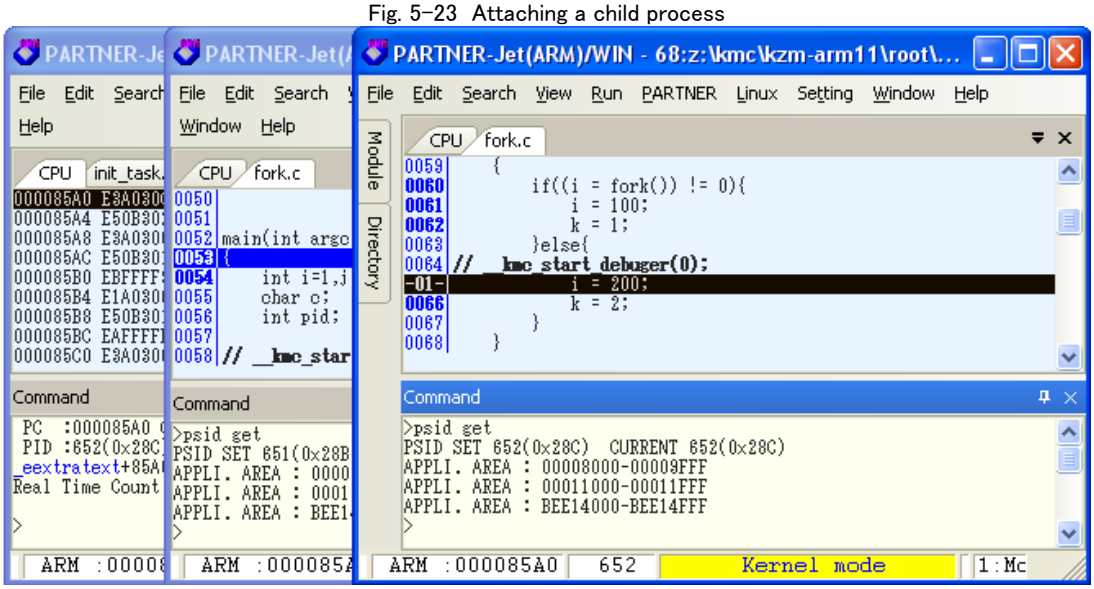
---

(12) Attaching a child process

Execute the "PSID command (Page 168)" in the PARTNER window for the child process, to attach the child process.

PT>PSID GET ↓

Fig. 5-23 Attaching a child process



Now you can performing debugging by individually controlling the child process and the parent process that uses the fork() system call. If there are multiple child processes, you can perform debugging by repeating the procedure.

After multi-process support has been achieved (when the PSID GET command has been issued in each separate PARTNER window), perform the following actions.

1. Set a breakpoint in the Process A and execution stops in the Process A.
2. At this point, execute the program using the G command in the window for the Process B.
3. The status of the Process A changes to "Application stopping" and the execution is virtually suppressed.

This means that, by executing the G command in a window other than the window in which the process is stopped, you can suppress the execution of that process. Also, if you issue the G/A command instead of the G command in that another window, you can execute all processes without suppressing that process.

Pay attention to the following notes when performing multi-process or multi-thread debugging using the fork() system call.

- Do not set breakpoints at the same address in multiple processes.
(Normal processing may not be possible at certain points.)
- Do not set breakpoints at the return addresses of functions.
(If a process other than the stopped process returns from a function, the system may not be able to continue processing.)

Chapter 6 Event Tracker

This chapter describes how to use the Event Tracker feature of PARTNER.

6.1 Requirements for the Event Tracker

The event trace feature saves the history of events (application processes, thread behavior, etc.) that occur on the target, and graphically displays it in a PARTNER window. This feature was developed separate from the Linux debugging features described in this document.
To use the event trace feature, there must be components that support this feature.

Table6-1shows a list of required components.

Table6-1 Components required for the Event Tracker

Type	Target	Description
PARTNER	WINPC	Required PARTNER debugger software which supports Event Tracker function
Event analysis filter	WINPC	EvtFilter_KMC.dll which provided by KMC or event analysis filter, Filter DLL which is customized by user
Kernel modification	TGT	Install the event collection feature on kernel. Refer to "Linux kernel modification for the Event Tracker" (Page213).
Application modification	TGT	Install the feature to debug conveniently the user mode program (application) on application. This is provided by KMC as application debug support file. Refer to "Application debug support file" (Page61)
PARTNER setting	PT	If the setting for Linux debugging is made, it is not needed.
PARTNER start option	PT	If the setting for Linux debugging is made, it is not needed.
PARTNER command	PT	Input PARTNER command window based on the situation.The commands are extended for Linux.

6.2 Linux kernel modification for the Event Tracker

Most kernel modifications are provided as a patch file by KMC. The main functions of the event collection feature are written in the following three files. These files do not exist in original Linux kernel sources, and are included in patch files provided by KMC.

- KMC/kmc.c
- KMC/kmc.h
- KMC/Kconfig_KMC

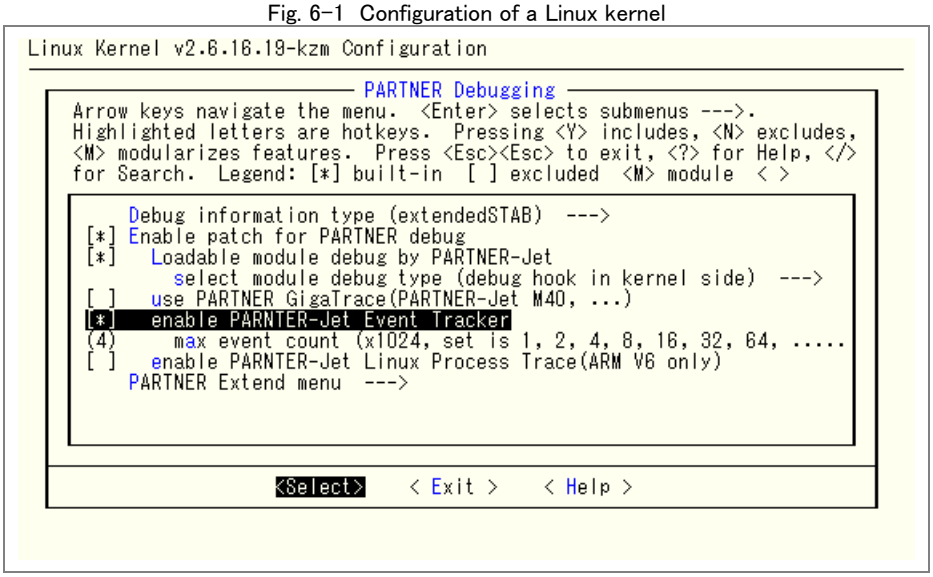
In addition, in the file that is included in original Linux kernels, the following two parts generate processes (threads) and require modification. Refer to the implementation of the samples provided by KMC to modify these parts.

kernel/fork.c	Around the end of the kernel/fork.c #1206 copy_process() function
fs/exec.c	In the last half of the fs/exec.c #1215 do_execve() function

Once modification has been completed, the configuration menu will be added.

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

Fig. 6-1 Configuration of a Linux kernel



To use the Event Tracker, enable [Enable PARTNER-Jet Linux Process trace]. You need to retain a buffer in the main memory of the target for event collection, to use the Event Tracker. Specify a power of two as the buffer size. If possible, also enable [Enable PARTNER-Jet Linux Profile (only ARM V6)] (only if the target architecture allows this application).

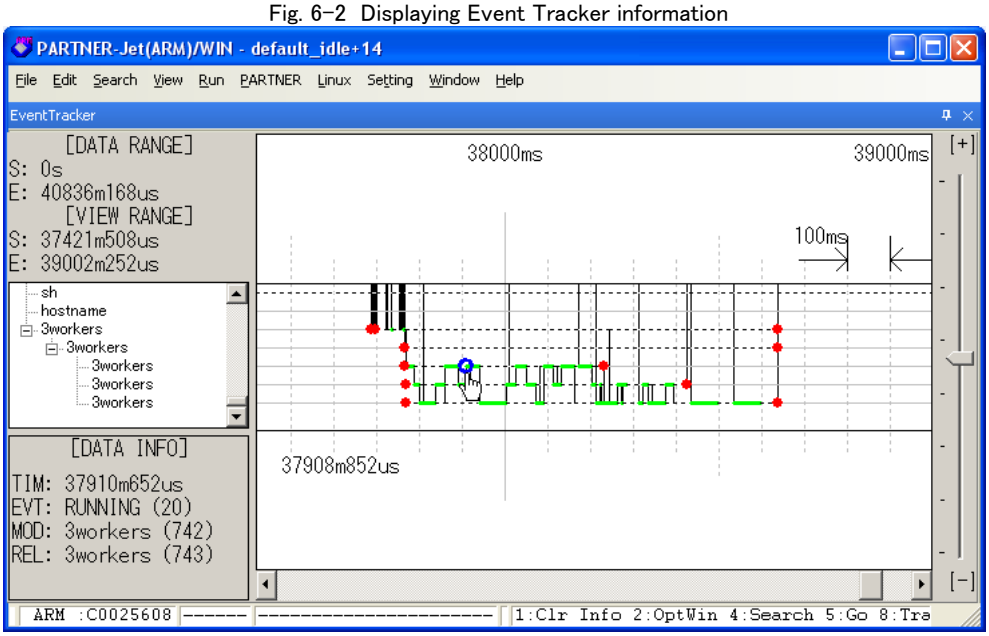
6.3 Launching the Event Tracker in the debugger

To launch the Event Tracker in the debugger, use the VIEWLOG command (refer to the manual of the debugger). After loading the event analysis filter DLL into PARTNER, it stops the CPU at given timing, and analyzes and visualizes the content of the history buffer on the target using the format 3 of the VIEWLOG command. You can also create a custom build filter DLL. In this example, EvtFilter_KMC.dll is used as its name. Replace the name with the name of your file as your read.

```
PT>viewlog dll EvtFilter_KMC.dll ↓
TGT>./ls ↓
TGT>./pwd ↓
TGT>./3workers ↓
(Pressing [ESC] Key)
PT>viewlog val( kmc linux event data),val( kmc linux event data size) ↓
(The result of event analysis is displayed)
PT>
PT>viewlog val( kmc linux event data),val( kmc linux event data size) ↓
(The result of event analysis is updated)
PT>
PT>viewlog clr ↓
(The result of event analysis is cleared)
```

This displays information in the Event Tracker window of PARTNER.

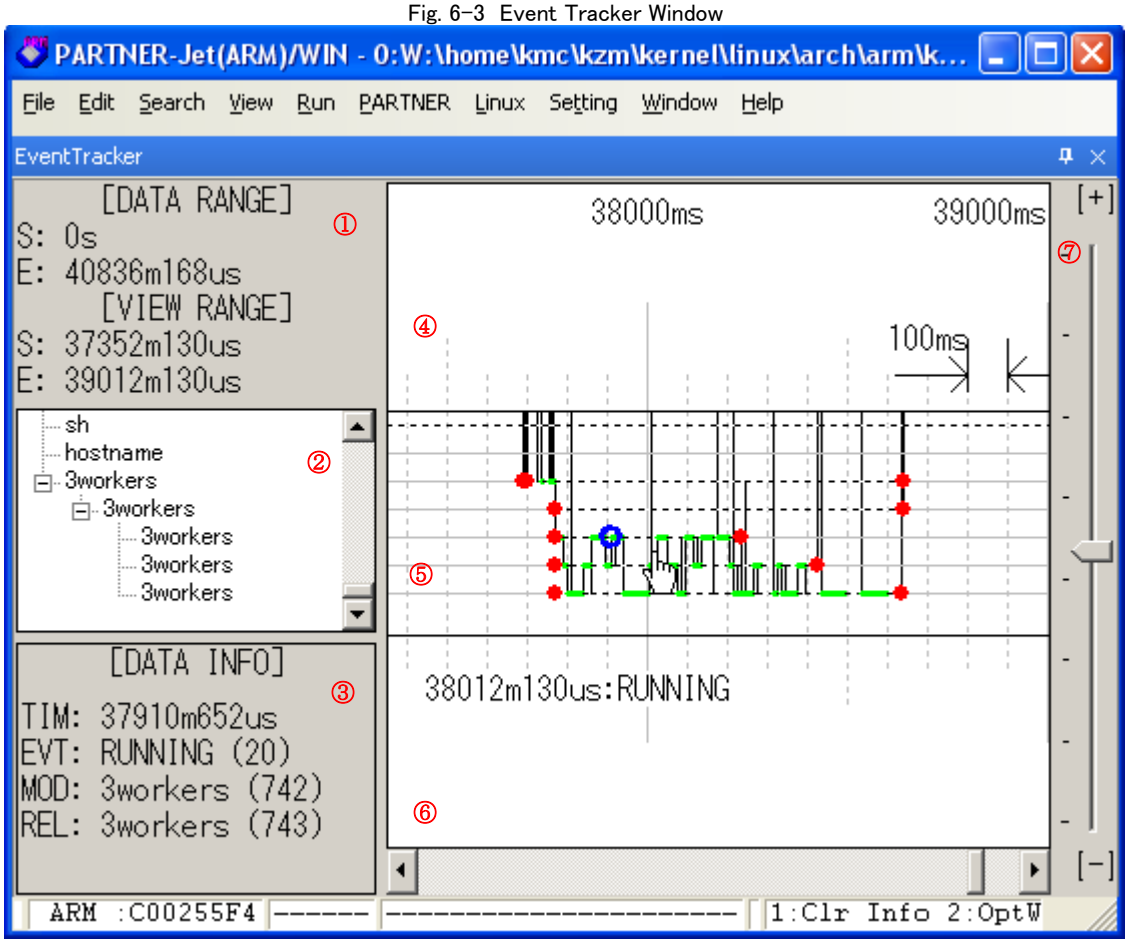
Fig. 6-2 Displaying Event Tracker information



6.4 Operation of the Event Tracker window

The Event Tracker window for Linux OS is displayed in Fig. 6-3. As you can finely customize the display of the Event Tracker window through the GUI, this section describes the operation methods according to the GUI area numbers shown in Figure 6-3.

Fig. 6-3 Event Tracker Window



Graph information area ((1))

This area displays the initial time, the last time, and the range of time of data retained in the Event Tracker.

Module (Process/Thread) display area((2))

This area displays the names of modules that have been registered with the Event Tracker.

Click a module to select it, and click the right button of the mouse, and select Hide Module or press the Delete key to hide the selected module (it only hides, and the data remains).

Data display area ((3))

This area displays the details of the data at the position where the cursor is placed.

Scale ((4))

Displays the time scale and the measurement unit. (You can display or hide the measurement unit.)

Graph ((5))

Normally, the mouse cursor appears as a hand in this area. You can grab an object by clicking the left button of the mouse, and scroll by moving the mouse (Fig. 6-4).

Rolling the mouse wheel forward: Zoom in (Zooming in to the center of the screen, the scale is halved)
Rolling the mouse wheel backward: Zoom out (Zooming out from the center of the screen, the scale is doubled)

Double-clicking selects the nearest event and displays the cursor (Fig. 6-5).

Pressing the Shift key changes the mouse cursor to a cross, and you enter Marquee Zoom Mode. The region selected with the Shift key pressed will be zoomed. While the left mouse button is pressed and a rectangle is displayed, the time between two points is displayed at the left bottom. Since releasing the Shift key terminates Marquee Zoom Mode, you can use it to quickly measure the time between given two points. Releasing the left mouse button while keeping the Shift key down will determine the range selection and zoom into it. (Fig. 6-6)

Pressing the Ctrl key changes the mouse cursor to an arrow, and you enter Select Mode. Dragging the arrow while pressing the Ctrl key will display the time between events using two vertical lines (Fig. 6-7).

Time bar ((6))

Double-clicking an empty place displays a line (time bar) used to measure the time. One of the time bars is the parent time bar that displays the absolute time, and others are child time bars and display relative times from the absolute time (Fig. 6-8).
Dragging a time bar moves the time bar (Fig. 6-8 right bottom). Moving the parent time bar changes all the relative times of its child time bars.
Double-clicking around a child time bar changes it into the parent time bar (main line). The previous parent time bar changes to a child time bar (sub line) (Fig. 6-9).
Clicking the right mouse button opens a popup menu from which you can delete individual time bars or clear all time bars (the rightmost time bar in Fig. 6-8 has been deleted in Fig. 6-9).

Zoom slider ((7))

Zooms in or out the graph display (by changing the time unit). You can hide this by clicking the right mouse button and selecting from the displayed popup menu.

Fig. 6-4 Moving the graph (from side to side)

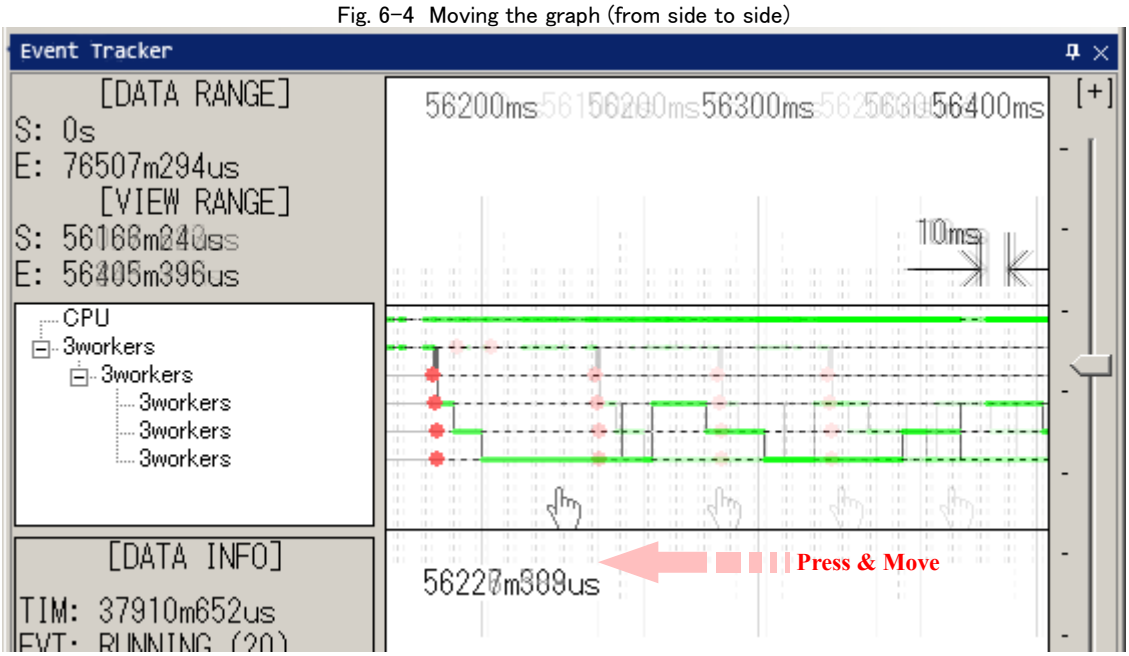


Fig. 6-5 Selecting an event

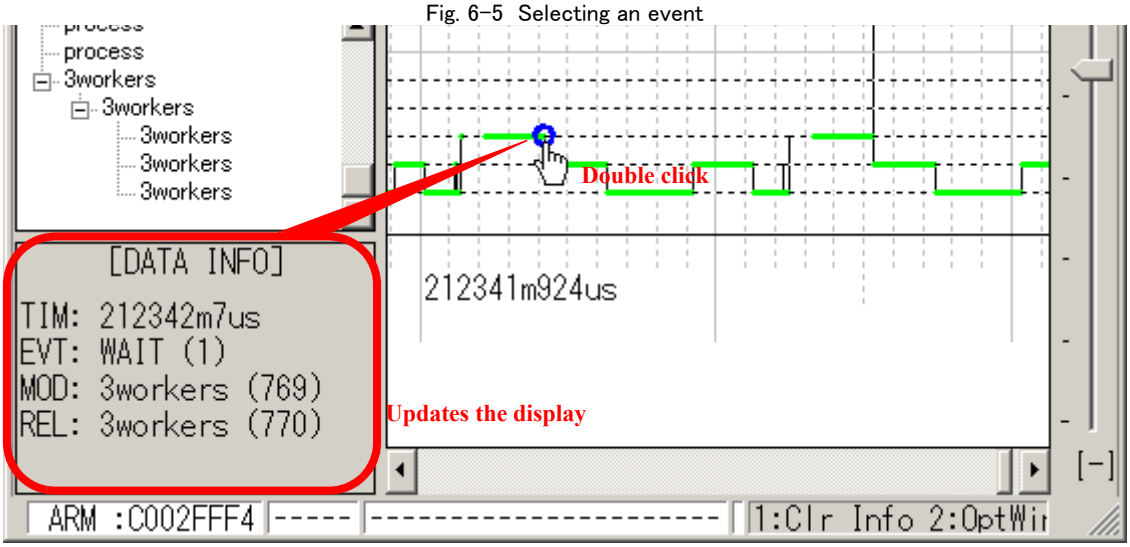


Fig. 6-6 [Zooming in using Marquee Zoom (Shift + dragging)

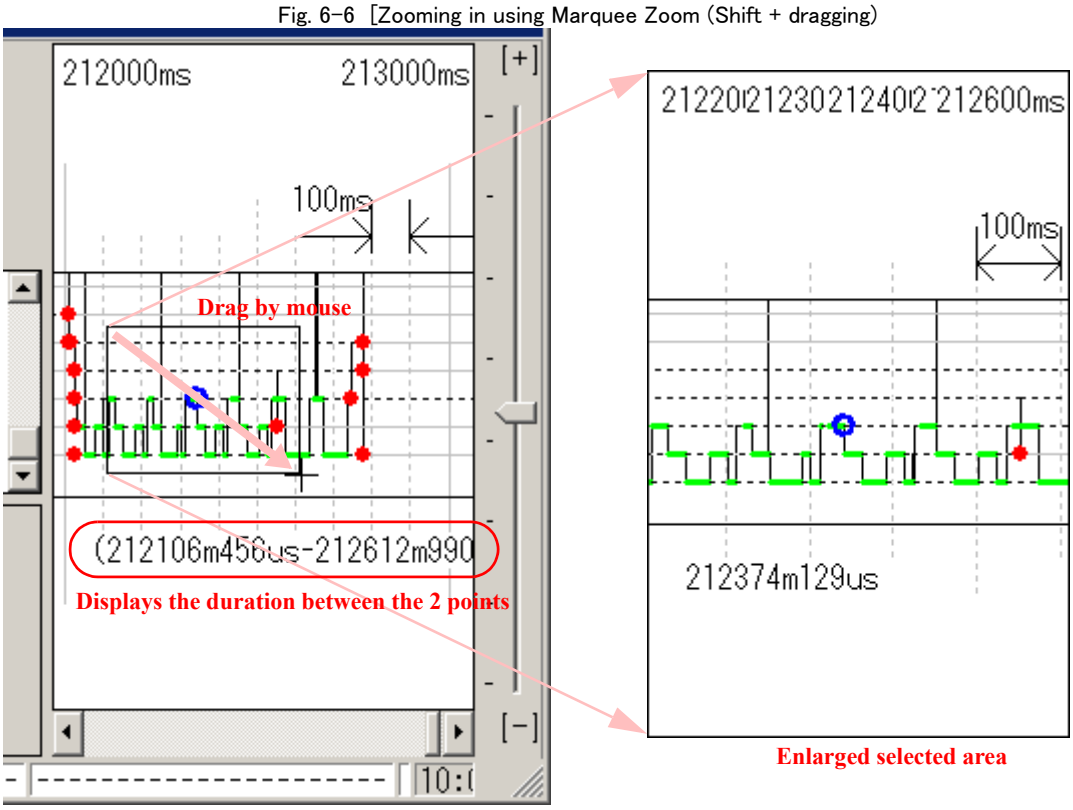


Fig. 6-7 [Measuring the time difference in Select Mode (Ctrl + dragging)

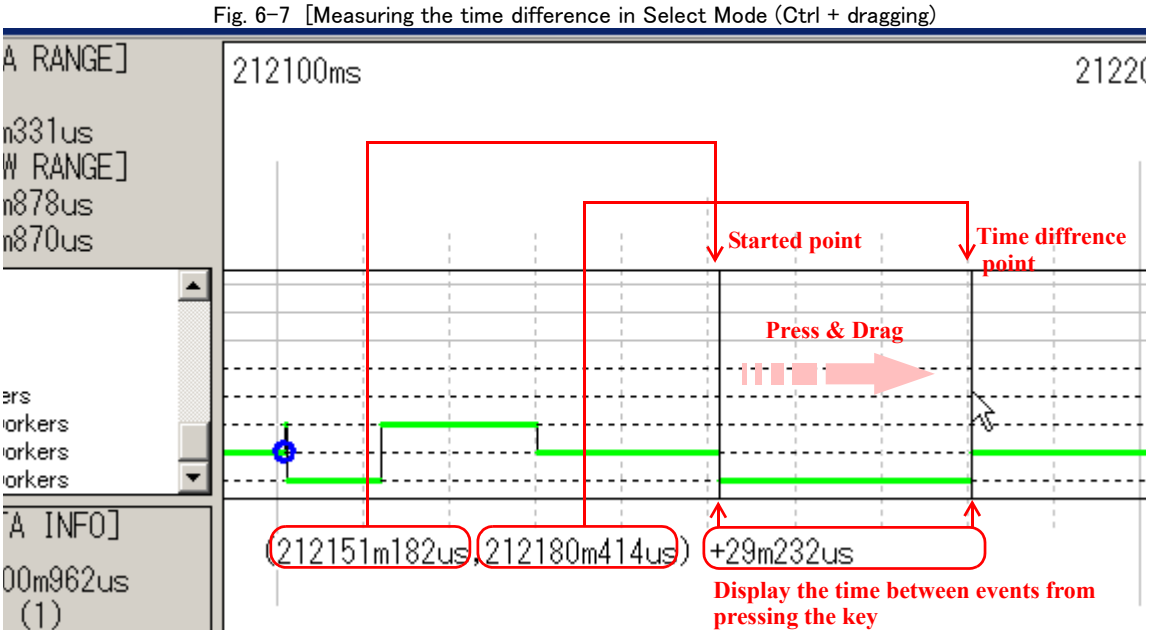


Fig. 6-8 Time bar display

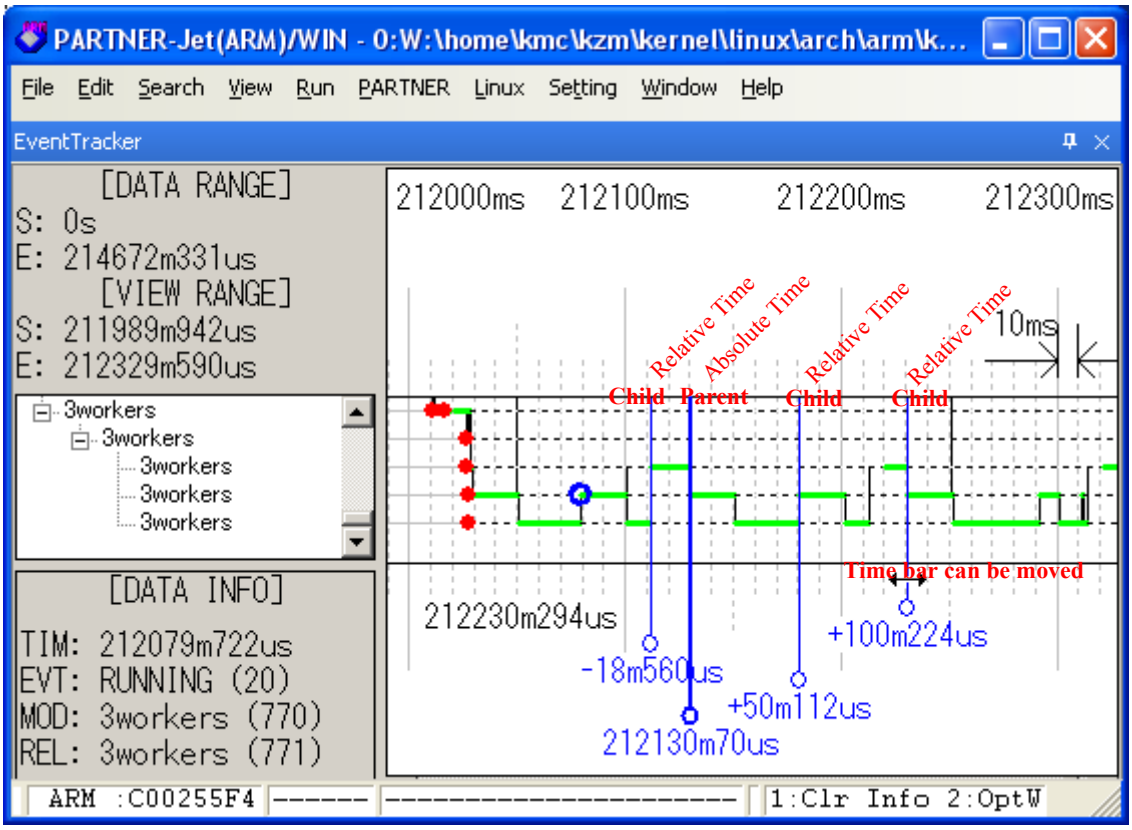
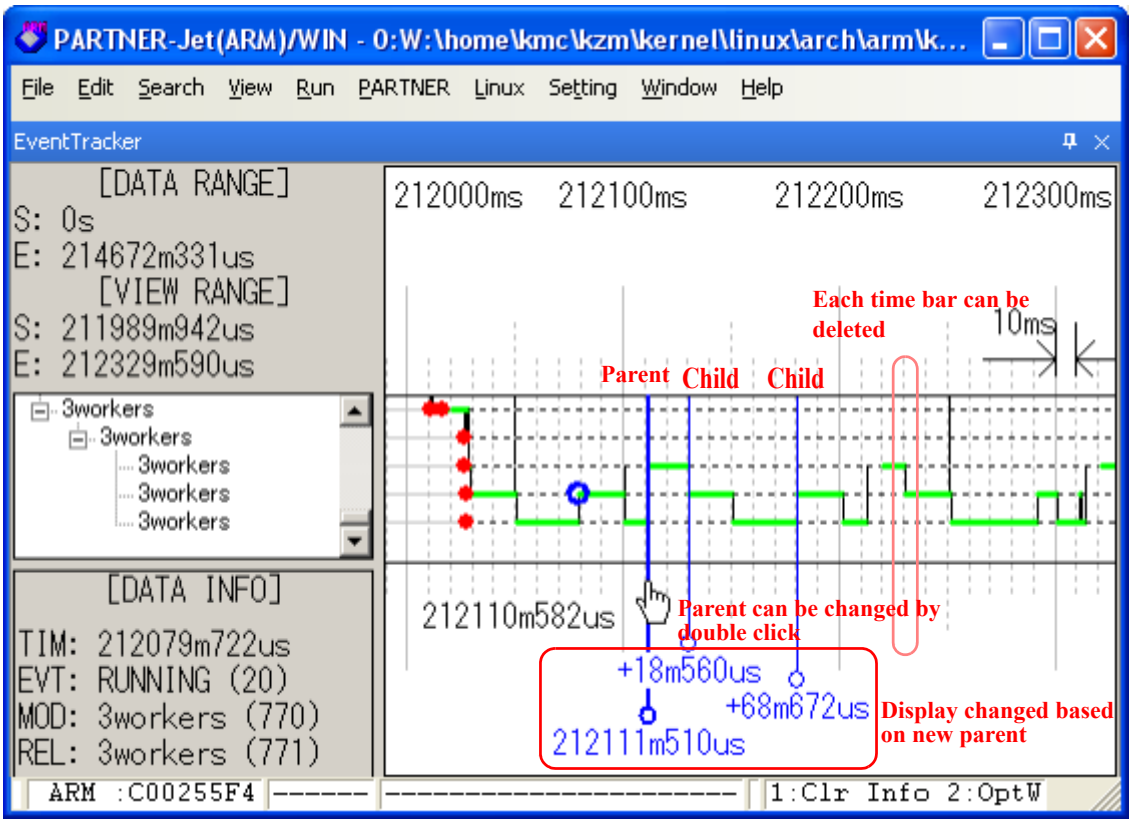


Fig. 6-9 Changing time bar display



6.5 Event tracker commands

The VIEWLOG command is added to operate the event tracker feature. All operations related to the event tracker can be specified with the first argument of the VIEWLOG command. In other words, this command is divided into subcommands.

For details on this command, refer to the VIEWLOG command in the debugger manual.

Colum 6-10 Relationship between the VIEWLOG command and GUI

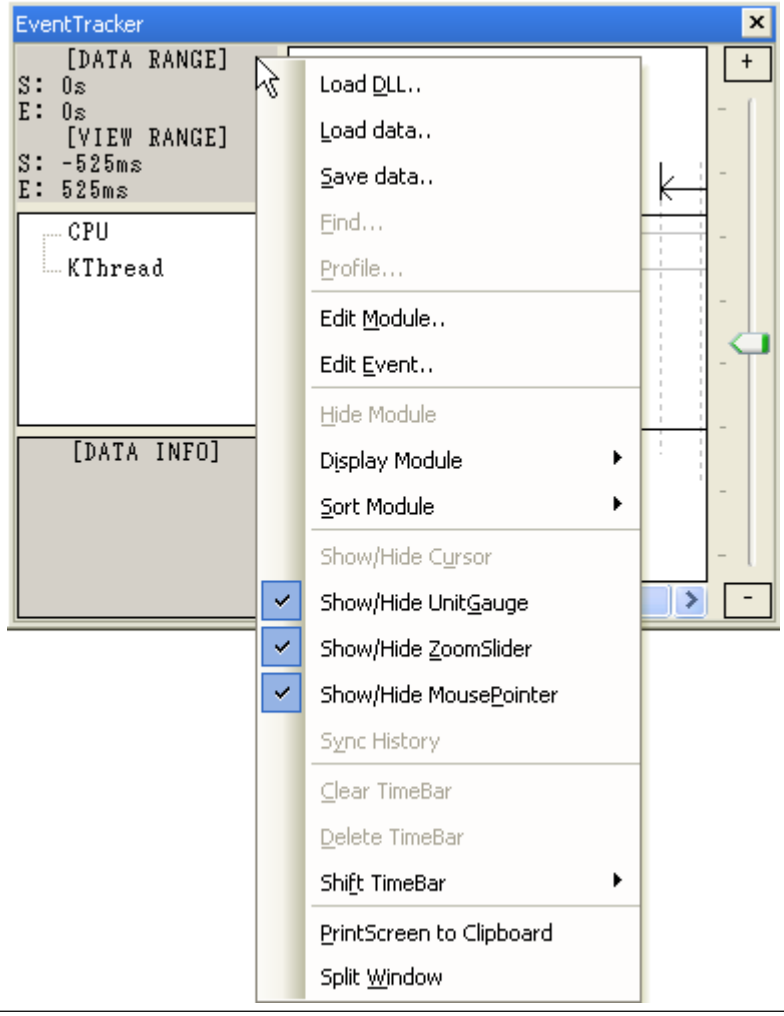
Most operations that can be done from the GUI and the menu of PARTNER can also be done using PARTNER commands.

The same design philosophy is also adopted for the event tracker, the VIEWLOG command for the Event Tracker corresponds to operations available in the popup menu displayed when the right mouse button is clicked in the Event Tracker window.

[Example]

The format 2 corresponds to "Load DLL".

The formats 5 and 6 correspond to "Load data" and "Save data" respectively.



【Example】

Basic operations of the event tracker feature.

```
PT>viewlog dll EvtFiletrLinux_KMC.dll ↓
TGT>./ls ↓
TGT>./pwd ↓
TGT>./3workers ↓
PT> Press the [ESC] key
(Displays the event information while breaking the CPU)
PT>viewlog val(_kmc_linux_event_data),val(_kmc_linux_event_data_size) ↓
(The result of event analysis is displayed)
PT>
PT>viewlog val(_kmc_linux_event_data),val(_kmc_linux_event_data_size) ↓
(The result of event analysis is updated)
PT>
PT>viewlog clr ↓
(The result of event analysis is cleared)
```

Profiling

```
PT>viewlog prof ↓
MOD: CPU(-20) EVT: RUNNING(20) TIM: 722m888us(2%)
MOD: CPU(-20) EVT: WAIT(1) TIM: 2463m263us(7%)
MOD: /linuxrc(1) EVT: WAIT(1) TIM: 24241m372us(73%)
MOD: PID:0(2) EVT: WAIT(1) TIM: 24311m934us(74%)
MOD: PID:0(3) EVT: RUNNING(20) TIM: 630us(0%)
MOD: PID:0(3) EVT: WAIT(1) TIM: 3186m880us(9%)
MOD: PID:0(4) EVT: WAIT(1) TIM: 28915m580us(88%)
MOD: PID:0(5) EVT: WAIT(1) TIM: 26342m55us(80%)
MOD: khelper(6) EVT: END(2) TIM: 32848m388us(99%)
*Snip*
MOD: 3workers(742) EVT: CREATE(0) TIM: 11us(0%)
MOD: 3workers(742) EVT: EXEC(10) TIM: 378us(0%)
MOD: 3workers(742) EVT: RUNNING(20) TIM: 22m676us(0%)
MOD: 3workers(742) EVT: WAIT(1) TIM: 834m382us(2%)
MOD: 3workers(742) EVT: END(2) TIM: 1673m963us (5%)
MOD: 3workers(743) EVT: CREATE(0) TIM: 69us(0%)
MOD: 3workers(743) EVT: RUNNING(20) TIM: 1m199us(0%)
MOD: 3workers(743) EVT: WAIT(1) TIM: 817m993us(2%)
MOD: 3workers(743) EVT: END(2) TIM: 1674m53us(5%)
MOD: 3workers(744) EVT: CREATE(0) TIM: 1m66us(0%)
MOD: 3workers(744) EVT: RUNNING(20) TIM: 131m620us(0%)
MOD: 3workers(744) EVT: WAIT(1) TIM: 2360m275us(7%)
MOD: 3workers(744) EVT: END(2) TIM: 102us(0%)
MOD: 3workers(745) EVT: CREATE(0) TIM: 681us(0%)
MOD: 3workers(745) EVT: RUNNING(20) TIM: 187m626us(0%)
MOD: 3workers(745) EVT: WAIT(1) TIM: 417m664us(1%)
MOD: 3workers(745) EVT: END(2) TIM: 1886m415us(5%)
MOD: 3workers(746) EVT: CREATE(0) TIM: 10m495us(0%)
MOD: 3workers(746) EVT: RUNNING(20) TIM: 375m689us(1%)
MOD: 3workers(746) EVT: WAIT(1) TIM: 431m371us(1%)
MOD: 3workers(746) EVT: END(2) TIM: 1674m577us(5%)
```

PT>

Saving/loading data

(When the event analysis result is displayed)
PT>viewlog save mylog.dat ↓
(mylog.dat is saved as a binary data file in the project folder)
PT>q ↓
(Reboots PARTNER)
PT>viewlog dll EvtFilterLinux_KMC.dll ↓
(Loads the dll file that was used when saving mylog.dat. This operation is not required, but just checks that the dll file exists.)
PT>viewlog load mylog.dat ↓
(Displays the data that was displayed when saving)

Appendices

This chapter describes how to debug loadable modules and applications when the Linux kernel cannot be modified, and other debugging techniques.

Appendix A Trouble shooting

The following table shows a list of trouble shooting items.

A-1 Kernel debugging

Table A-1 Kernel debugging

Case	Things to check	Reference
Cannot display source code	Make sure that you've added debug information option in kernel compiling options.	Modifying and configuring the Linux kernel source (Page 54)
	Make sure that path translation is done properly in -XGX startup option.	-XGX option (Page 150)
	PARTNER will open the source file which path is displayed in the caption of the code window.	SNAME command (Page 178)
	Check file sharing (such as Samba) settings both for Linux and Windows.	—
The displayed source code doesn't match what executed	Make sure that the executing binary and debug information match.	—
Cannot perform breaking	If your system transfers the kernel from FRASH memory to RAM, you cannot control such system with software breakpoints until start_kernel() function is executed. Use hardware breakpoints instead. PT>br start_kernel,ex ↓ PT>brc * ↓ PT>bp sys_write ↓	—

A-2 Loadable module debugging

Table A-2 Loadable module debugging

Case	Points to check	Reference
Cannot automatically attach to PARTNER	Make sure that the modification and patches to loadable module source code is done properly.	Modifying the source of a loadable module (Page 90)
	Make sure that -OS LINUX option is set in startup option.	Preparing for debugging (Page 90) -OS option (Page 147)
Cannot load debug information	Make sure that the loadable module object exists in the directory where the loadable module has been built.	Modifying the source of a loadable module (Page 90)
	If you have specified the path to the loadable module path in _KMC_MODULE_NAME, make sure that the loadable module object is available in the path.	
Cannot display source code	Make sure that you've added debug information option in loadable module compiling options.	
	Make sure that path translation is done properly in -XGX and -SK startup options.	-XGX option (Page 150) -SK option (Page 153)
	PARTNER will open the source file which path is displayed in the caption of the code window.	SNAME command (Page 178)
	Check file sharing (such as Samba) settings both for Linux and Windows.	
Cannot debug after performing strip	If you perform strip all symbols, you cannot debug since to debug Loadable module, it is necessary to have module_init symbol in a module. When you perform strip loadable module, add -g option.	For Application: LINUX>strip objfile For Loadable module: LINUX>strip -g objfile

A-3 Application debugging

Table A-3 List of trouble shooting tips for application debugging

Case	Things to check	Reference
The displayed source code doesn't match what executed	Make sure that the executing binary and deug information match.	—
Cannot automatically attach Application	If PARTNER hangs up on starting an application, MAP declaration in CFG file may be incorrectly declared. Check MAP field settings.	MAP field (Page 141)
	You need at least once to load the kernel debug information of your Linux system.	If you didn't, just load the debug information of the kernel currently being debugged.
	If you user debug support file as Preload library method, make sure that the setting is done properly for /etc/ld.so.preload and environment variable LD_PRELOAD. If the setting is done incorrectly, an error "The data specification is not correct" will be occurred when performing ATTACH command (Page 162).	Application debug support file (Page 61) ATTACH command (Page 162)
	Make sure that there is a debug stub function (_kmc_start_debuger() or _kmc_start()) inserted at the top of main() function of the application. You also need to make sure that there is proper application name supplied in a parameter of debug stab function.	Application debug support file (Page 61)
	Make sure that -OS LINUX option is set in startup option.	-OS option (Page 147)
	【For MIPS Series only】 If it can automatically attach in kernel mode but cannot do it in application mode, it may not be able to use system call (gettid) properly. Make sure that the order of gettid declation in arch/mips/kernel/syscalls.h matches the order of gettid declation in include/asm-mips/unistd.h.	Notes on manual modification for MIPS series (version 2.4.x) (Page 55)

Case	Things to check	Reference
Cannot debug multi-threaded application (pthread)	Make sure that there is a debug stub function (_kmc_start_debuger() or _kmc_start()) inserted at the top of main() function of the application or at the top of the thread body. You also need to make sure that there is proper application name supplied in a parameter of debug stab function.	Application debug support file (Page 61)
	Make sure that -OS LINUX option is set in startup option.	-OS option (Page 147)
	Launch PARTNER window for the application using MULTI command and make sure that the debug information for the application has been loaded in that PARTNER window.	MULTI command (Page 173)
	Make sure the PARTNER window is working in ADD mode when debugging in one PARTNER window. In NON_ADD mode, you need to launch multiple PARTNER windows for each thread and load debug information for them.	PSID command (Page 168)
Cannot debug multi-processed application (fork)	Make sure that there is a debug stub function (_kmc_start_debuger() or _kmc_start()) inserted at the top of main() function of the application or at the top of the thread body. You also need to make sure that there is proper application name supplied in a parameter of debug stab function.	Application debug support file (Page 61)
	Make sure that OS LINUX option is set in startup option.	-OS option (Page 147)
	Launch PARTNER window for the application using MULTI command and make sure that the debug information for applicaion has been loaded in that PARTNER window..	MULTI command (Page 173)
Cannot attach running application.	Make sure that glibc used by the application is linked to kmc-support.c support file.	Modifying the glibc library of the target system (Page 241)
	You should register the glibc used by the application to PARTNER when it is modified.	Registering glibc with PARTNER (Page 242)
	ATTACH command is applicable only when applications to be attached to PARTNER is running. You can use PS command to confirm it.	PS command (Page 160) ATTACH command (Page 162)

A-4 Real-time trace

Table A-4 List of trouble shooting tips for real-time trace

Case	Things to check	Reference
Cannot display trace information	Make sure that your PARTNER-Jet supports trace function.	
	Make sure the JTAG probe connected to the target is a trace support probe.	
	Make sure that the target CPU supports trace function. Some CPUs require Trace Enable flag set from JTAG ICE or debugger.	
Cannot distinguish trace information for application and kernel	Make sure that each PARTNER instance has loaded required debug information for the application and the kernel.	
	Make sure that -OS LINUX option is set in startup option.	-OS option (Page 147)
	Make sure that the modification of your Linux kernel has been applied correctly and properly.	Modifying and configuring the Linux kernel source (Page 54)
Cannot display application's trace information	Make sure that the application's debug information has been loaded. You should load debug information for shared libraries before trace them.	Linux OS compatible history display (Page 125)

Appendix B Building a Linux PC for development

This appendix describes a procedure for setting up a Linux PC for development used in a cross development environment for the KZM-ARM11-01 board (Refer to Figure 1-1) described in "About development environments (Page 3)"
The procedure and path names may differ, depending on the distribution of the used PC Linux and the distribution of Linux for the target. If necessary, replace the appropriate words as you read.

The setup procedure for a Linux PC for a development environment is described in the following sub-sections.

- "Setting up a cross environment on a Linux PC (Page 230)"
- "Setting up an NFS server using the Linux PC for development (Page 234)"
- "Setting up terminal software (Page 235)"
- "Starting up Linux on the target (Page 236)"
- "Setting up a Samba Server (Page 238)"

B-1 Setting up a cross environment on a Linux PC

A cross development environment requires software (toolchain) that outputs operation code for the target CPU on the host PC (ix86 CPU) and a software environment cross-compiled using that software.
In general, GNU tools such as binutils and gcc are used as software for cross-compiling.
To obtain the source code of a GNU tool and build the toolchain for cross-compiling, the following software packages are needed (it may differ depending on the PC Linux distribution).

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

As for the KZM-ARM11-01 board, a built binary file is provided so you can simply install it into a Linux PC.
This section assumes that the provided file is placed under /tmp/, and install it under the /opt/kmc/kzm-arm11/ directory.

(1) Installing the cross toolchain.

Expand the toolchain as a super user (root).

```
LINUX86>$ su ↓
LINUX86># cd / ↓
LINUX86># tar xvzf /tmp/kzm-arm11_toolchain-xxxxxx.tgz ↓
(It is expanded under /opt/kmc/kzm-arm11/staging_dir/)
```

Check out the location of your cross-compiling tool.

```
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++
```

To use the cross toolchain, specify the path.

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

Alternatively, add the above path to ~/.bash_profile, and reload the environment file using the source command.

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

(2) Building a root file system

For the KZM-ARM11-01 board, a built archive created using Buildroot(http://buildroot.uclibc.org/) is provided, so simply expand it.

```
LINUX86>$ su ↓
LINUX86># cd / ↓
LINUX86># tar xvzf /tmp/kzm-arm11_rootfs-xxxxxx.tgz ↓
(It is expanded under /opt/kmc/kzm-arm11/root/)
LINUX86>$ ls -F /opt/kmc/kzm-arm11/root/ ↓
bin/  etc/   lib/      mnt/  proc/  sbin/  tmp/  var/
dev/  home/  linuxrc@  opt/  root/  sys/   usr/
```

Create device special files ("/dev/null" and "/dev/console") that are required for an NFS root file system.

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

Even if you do not use NFS root, prepare a root file system tree that can be viewed from a Windows PC. For example, if there is a root file system consisting of only files that have no debug information in the HDD of the target board, a root file system tree consisting of the same program files with debug information should be stored in the Samba shared folder or the HDD of the Windows PC. PARTNER searches debug information from a root file system that contains debug information using the -RootDir option, when application debugging is performed.

(3) Preparing a Linux kernel for the target

Expand the prepared kernel source.

```
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 ↓
(It is expanded under /opt/kmc/kzm-arm11/build_src/linux-2.6.16-xxxxxx/)
LINUX86># ln -s linux-2.6.16-xxxxxx linux ↓
(Provides a symbolic link so that it can be easily understood hereafter)
```

A patch for PARTNER debugging is already applied to the Linux kernel provided for the KZM-ARM11-01 board. If you use a kernel to which a patch for debugging has not yet been applied, apply the patch following the procedure described in "2.2 Modifying and configuring the Linux kernel source (Page 54)"

Generate the default configuration.

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

Configure the kernel.
In the [PARTNER Debugging] menu, enable [Debug infomation type], [Enable patch for PARTNER debug], [Loadable module auto attach] and [Loadable module auto attach (patch is include in kernel)].
For details on each item, refer to "Kernel configuration (Page 57)".

```
LINUX86># make menuconfig ↓
```

Build the kernel.

```
LINUX86># make ↓
```

Install the device driver module into the root file system.

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

(4) Preparing the application debug support file.

Build the debug support file as a library (refer to "Preload library method (Page 63)").

```
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
-------------------------------------------------------
```

Install the built library into the root file system for the target.
You can place it anywhere you want, but in this document it is placed under "/usr/lib/".

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

Set the environment variable LD_PRELOAD to use it as a PRELOAD library.

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

Because the Linux distribution provided for the KZM-ARM11-01 board does not use glibc (uses uClibc), the setting cannot be made in the /etc/ld.so.preload file. In this example, the LD_PRELOAD environment variable is registered with /etc/profile that is always used by /bin/sh. But if only a specific user performs debugging, you can write it in the setting file for each user (e.g. /home/foo/.profile).

You have now prepared the software necessary for the target Linux PC.
The configuration of installed files is the same as "File locations on the Linux PC (Page 4)"

B-2 Setting up an NFS server using the Linux PC for development

To set up an NFS server, you need to install NFS server software.

```
LINUX86>$ su ↓
LINUX86># apt-get install nfs-user-server ↓
(The installation method and package name may differ depending on the distribution.)
```

Edit the /etc/exports file to set a directory shared through NFS. In this example, the hierarchy under the /opt/ directory is shared through NFS.
(Just sharing the hierarchy under the /opt/kmc/kzm-arm11/ directory is sufficient.)

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

The no_root_squash option is added, so that the root user on the target can manipulate files that require root permission.
There are three protocol versions (2, 3 and 4) for the NFS protocol, and the format of /etc/exports differs depending on the used version. If not inconvenient, it is safer to support the behavior of earlier versions and avoid using a format such as a wild card (*). For details on the format, refer to "man(5) exports". Confirm the protocol version used during operation using the rpcinfo command.

"kzm-arm11" written in the /etc/exports file is the name of the target board. Write it in the /etc/hosts file, or register it with DNS.

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

Some screen shots contain IP addresses. The IP addresses used are shown for reference in the following table.

Table B-1 Setting examples of IP addresses

Devices	IP address	name
Linux PC for developing	192.168.1.16	linuxpc
KZM-ARM11-01 target board	192.168.1.202	kzm-arm11
Windows PC for debugging	192.168.1.XXX	-

When the setting has been made, start up the NFS server.

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

You have now completed the NFS server setting.

B-3 Setting up terminal software

As long as the serial console that the target board uses is available, any terminal software will do the job. If the KZM-ARM11-01 board is used, set the serial port as shown in Table B-2.

Table B-2 Communication settings of the serial port

Items	Setting Value
Communication rate	115200bps
Data	8bit
Binary	NA
Stop	1bit
flow control	NA

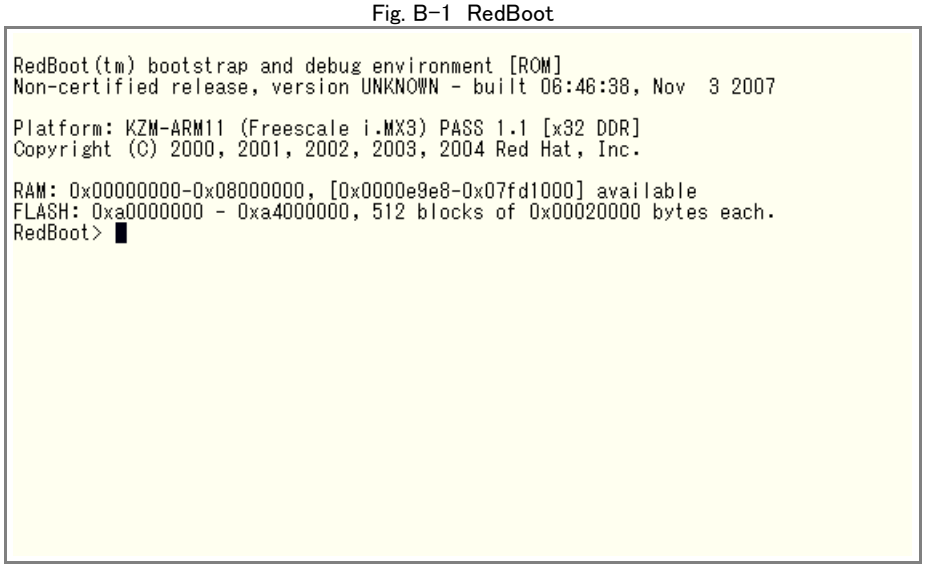
B-4 Starting up Linux on the target

The KZM-ARM11-01 board is shipped in a state that the RedBoot program automatically launches when the power is turned on (see "Fig. B-1").

As the NFS boot can be performed from RedBoot, you can try to see if Linux can start up at this point.

(Evaluation boards tend to be shipped with some kind of monitoring program installed. If the monitoring program supports a network boot feature such as the NFS boot and tftp boot, similar confirmation can be performed. For details, check out the specifications of your board.)

Fig. B-1 RedBoot

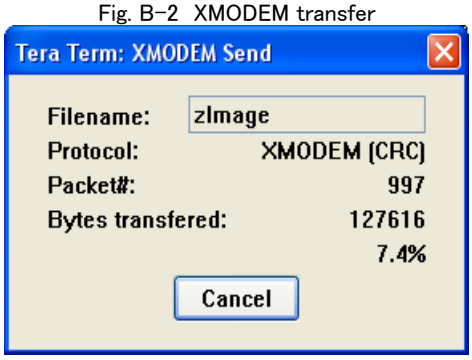


RedBoot can transfer the Linux kernel image via an RS-232C cable to RAM using the XMODEM protocol. If an RS-232C cable has already been connected to the Windows PC for debugging, transfer the file from the Linux PC for development in advance and use the load command at the prompt of RedBoot.

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

When the state has transitioned to standby, transfer the file using terminal software. For example, if TeraTerm is used, select [File]->[Transfer]->[XMODEM]->[Send], add [Option]->[Checksum], and specify the file to be written for [zImage].

Fig. B-2 XMODEM transfer




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

Once the transfer has completed, set kernel parameters for the NFS root file system and start it up.

```
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"
↓
(One line without a linefeed up to here)
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
```

When the boot prompt has been displayed, log into the system to check out whether or not the file has been properly written.

Fig. B-3 Starting up Linux

```
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 Setting up a Samba Server

To set up a Samba server, you need to install Samba server software.

```
LINUX86>$ su ↓
LINUX86># apt-get install samba samba-doc ↓
(The installation method and package name may differ depending on the distribution.)
```

Edit the /etc/samba/smb.conf file to set a shared directory.

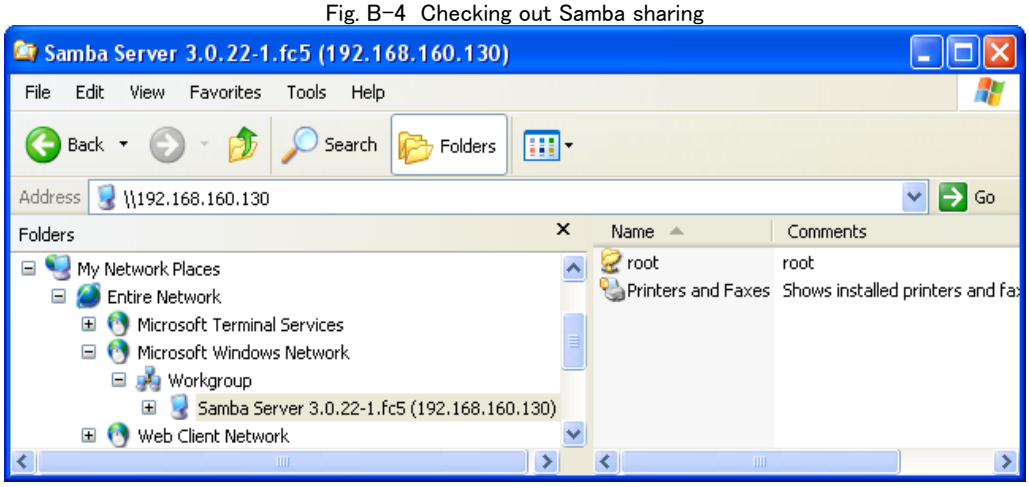
```
LINUX86>$ su ↓
LINUX86># vi /et/samba/smb.conf ↓
~ [global]
~    workgroup = WORKGROUP
~    server string = %h server (Samba %v)
~    security = share
~    socket options = TCP_NODELAY
~ [opt]
~    comment = opt directory
~    path = /opt
~    browseable = yes
~    guest ok = yes
~    public = yes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

In this example, the hierarchy under the /opt/ directory will be shared.
Since this configuration does not take security considerations into account and the write permission is not necessary, refer to man(7) samba to configure it properly. When the configuration has been completed, start up the Samba server.

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

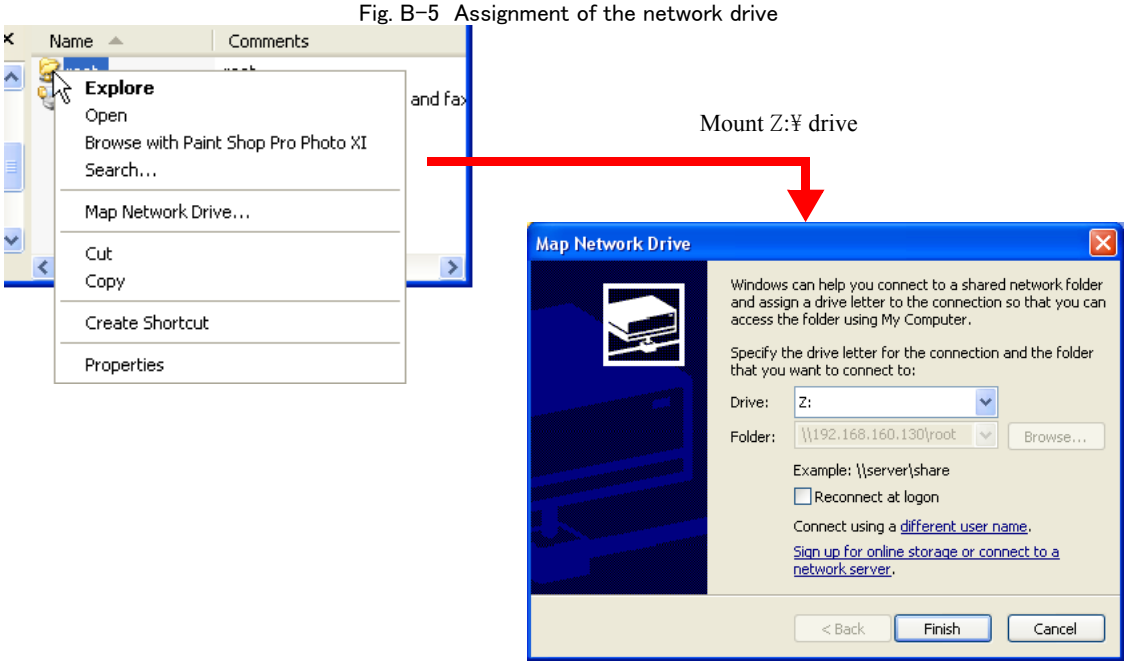
To see if you can view the directory from the Windows PC, check it using the Explorer.

Fig. B-4 Checking out Samba sharing



When you have confirmed that you can access the opt directory, assign the network drive.

Fig. B-5 Assignment of the network drive



Now you can view the /opt/kmc/kzm-arm11 directory on the Linux PC for development from the Windows PC for debugging (from the Windows PC the directory path is z:¥kmc¥kzm-arm11).

Appendix C How to insert the support file into glibc

You can put the debug support file for PARTNER application debugging in a frequently-used shared library.

This section describes, in the following order, how to put the support file into glibc, and, as an example, how to debug a running application after attaching it, which cannot be done by linking the support file directly to an application.

- "Modifying the glibc library of the target system (Page 241)"
- "Registering glibc with PARTNER (Page 242)"
- "Debugging method (Page 242)"

C-1 Modifying the glibc library of the target system

To attach a running application, it is necessary to link the support file (kmc-support.c) to glibc.

Follow the procedure below to create the glibc library for PARTNER attachment.

1. Copying the support file (kmc-support.c)

   Copy the support file (kmc-support.c) into glibc-x.x.x/sysdep/unix/sysv/linux/<cpu> in the glibc build environment.

2. Modifying Makefile

   Add a line "routines += kmc-support" into "glibc-x.x.x/csu/M

【Example】 Modifying Makefile

```
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. Creating glibc

   Create glibc by entering the following commands.

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

If you comment out the line you inserted in glibc-x.x.x/csu/Makefile, you can restore the original build without deleting the support file (kmc-support.c).

If a "Select Target CPU type" compile error occurred in kmc-support.c, enable only the target CPU in the CPUTYPE symbol declaration in kmc-support.c, and recompile it.

4. Checking glibc

   Confirm whether or not the created glibc library has correctly linked the support file (kmc-support.c).

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

   If a symbol is displayed, it has been properly done.

5. Installing glibc

   Install the created glibc library onto the target system.

C-2 Registering glibc with PARTNER

Specify the file name of the glibc library created in "C-1 Modifying the glibc library of the target system (Page 241)" and the symbol address (_kmc_sleep_thread) in the support file (kmc-support.c) using the LINUX command (Page 164).

```
PT>LINUX set attach offset <Library Name> <Address of  kmc sleep thread> ↓
```

Specify the address that was confirmed in "Checking glibc (Page 241)" as the _kmc_sleep_thread address.

[Example]

```
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)
(The displayed content differs depending on such things as the CPU type.)
```

Unless the glibc library, to which the support file (kmc-support.c) was linked is modified, this LINUX command (Page 164) can be omitted.

C-3 Debugging method

Refer to "3.4 Debugging an application (Page 99)" or "5.1 Debugging in Application Mode (Page 182)", and execute the kernel from PARTNER and the debug-target application.

The procedure for debugging an application is the same as the case where a preloaded application debug support file is used.

For details, refer to ""Debugging an application (Page 99)"", ""Debugging a multi-thread application (Page 106)"", and ""Debugging a multi-process application (Page 116)"".

Appendix D Debugging a dynamic linker/loader (ld.so)

For debugging a dynamic linker/loader (ld.so), automatic address resolution cannot be performed in PARTNER, unlike debugging a shared library that was described in "3.7 Debugging a shared library (Page 122)". Consequently, you need to manually attach it to PARTNER.

This section describes the procedure for debugging a linker/loader in the following order, using the case where a sample (ld.so/sample) is used as an example.

(1) "Add debug information to a linker/loader (Page 244)"
(2) "Creating an application (Page 244)"
(3) "Confirming the address of the _dl_start symbol in a linker/loader (Page 244)"
(4) "Executing the kernel (Page 244)"
(5) "Confirming the initial address of the .text of a linker/loader (Page 244)"
(6) "Forcibly breaking the kernel (Page 245)"
(7) "Setting a hardware break (Page 245)"
(8) "Executing the debug-target application (Page 245)"
(9) "Loading debug information for a linker/loader (Page 245)"
(10) "Canceling a hardware break (Page 245)"
(11) "Attaching a process (Page 245)"
(12) "Loading additional debug information for an application/shared library (Page 246)"

(1) Add debug information to a linker/loader

Add debug information to the debug-target linker/loader.
Select for the debug information the format that is used for the kernel and application. If PARTNER cannot load the debug information properly, try other formats.

(2) Creating an application

Create a debug-target application. When debugging an application with a linker/loader, do not insert debug stubs (_kmc_start()) in the application.

(3) Confirming the address of the _dl_start symbol in a linker/loader

Confirm the address of the _dl_start symbol in the linker/loader created in "(1) Add debug information to a linker/loader (Page 244)".

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

Fig. D-1 Confirming the address of the _dl_start symbol

```
[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) Executing the kernel

Refer to the part from the begging to "(4) Loading application debug information (Page 102)" in "3.4 Debugging an application (Page 99)", and put the system into a state where the PARTNER window for the application is open and the kernel is running.

(5) Confirming the initial address of the .text of a linker/loader

Check the memory map of the application using PARTNER to confirm the initial position of the .text of the linker/loader.

```
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
(The displayed content depends on the CPU type and MAP field settings.)
```

(6) Forcibly breaking the kernel

Press the ESC key in the PARTNER window to forcibly break the kernel.

(7) Setting a hardware break

Set a hardware breakpoint at the _dl_start in the linker/loader. The _dl_start address is calculated by adding the offset value confirmed in "(3) Confirming the address of the _dl_start symbol in a linker/loader (Page 244)" to the .text initial address described in "(5) Confirming the initial address of the .text of a linker/loader (Page 244)".

```
PT>br 0x1eac + 0x40000000.ex ↓
```

Note that setting a hardware break at _dl_start causes breaks to occur in every process.

(8) Executing the debug-target application

Execute the kernel again, and then, execute the debug-target application on the target system.

```
PT>./sample ↓
```

(9) Loading debug information for a linker/loader

Load the debug information for the debug-target linker/loader in the command window of the application PARTNER window, when your PARTNER breaks.

```
PT>ls ld-2.2.5.so./r .text=0x40000000 ↓
```

When loading debug information for a linker/loader, you always need to specify the relocation address.

(10) Canceling a hardware break

Cancel the hardware breakpoint set in "(7) Setting a hardware break (Page 245))".

```
PT>brc * ↓
```

(11) Attaching a process

Attach the current application process using the "PSID command (Page 168)".

```
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
(The displayed content depends on the CPU type and MAP field settings.)
```

(12) Loading additional debug information for an application/shared library

Load additional debug information for the debug-target application or shared library in the PARTNER window for the application. Now you can debug a linker/loader, application and share library together.
Make sure to check [Symbol Only] and [Append] when loading it.

```
PT>lsa sample ↓
```

Appendix E Technology description: Linux architecture and PARTNER

PARTNER for Linux is a high-performance source-level debugger, which was created by adding debugging features for embedded Linux to the conventional PARTNER debugger.
This chapter describes why Linux-compatible features are necessary for the debugger, what technological background the features of PARTNER for Linux are based upon, and the details of the development and debugging of embedded Linux made available thanks to the new Linux-compatible features.

E-1 Linux development environment and problems

Nowadays, embedded systems are getting smaller and more complicated, and their price is falling; so Linux is attracting more attention as the next generation OS for embedded devices.
Linux is a UNIX clone OS that features multi-task processing, virtual memory, shared libraries, demand loading, memory management, and the TCP/IP network function, and the standard version includes file system components and network system components.
Moreover, because all provided source code is open source, developing embedded systems using Linux costs much less than development done with conventional embedded systems such as a Real-Time OS (RTOS).
Linux uses the MMU (Memory Management Unit) of a CPU, and so it can build a reliable system using the MMU to create virtual memory space, protect memory in each space, and set memory protection to unassigned areas in each space.
On the other hand, in the current normal development of embedded Linux, use of the MMU causes complicated address management of virtual memory space (multiple logical spaces). For that reason, there is a problem that consistent debugging is difficult, because you have to use multiple debuggers according to the debug target (Linux kernels, loadable modules (device drivers) and applications (processes)) (see Table E-1).

Table E-1 Software programs that run on Linux

—	Space	Executing address	Execution
(1) Linux kernel	Logical space matches Physical space (Logical address and physical address mapped one by one.)	Address is decided when compile/ link	Direct execution at the specified address through writing to ROM or transfer from ICE.
(2) Loadable module (Device driver)	Logical space matches Physical space (Logical address and physical address mapped one by one.)	Executable address is decided when insmod command is used on target.	Execute via Linux kernel on file system
(3) Application (Process)	Logical multiplexed space	Address decided when compile/link	Execute via Linux kernel on file system

(1) Linux kernel

If the allocation address is determined when compiled or linked, logical addresses and physical addresses are mapped to each other one by one and execution can be performed from the specified program counter. So you can use a conventional ICE and a debugger for embedded system development as is, and develop a Linux kernel.

(2) Loadable module (device driver)

It is executed in the same space as a (1) Linux kernel, but its characteristic is that the address to which it is actually allocated is not determined until execution (relocatable).

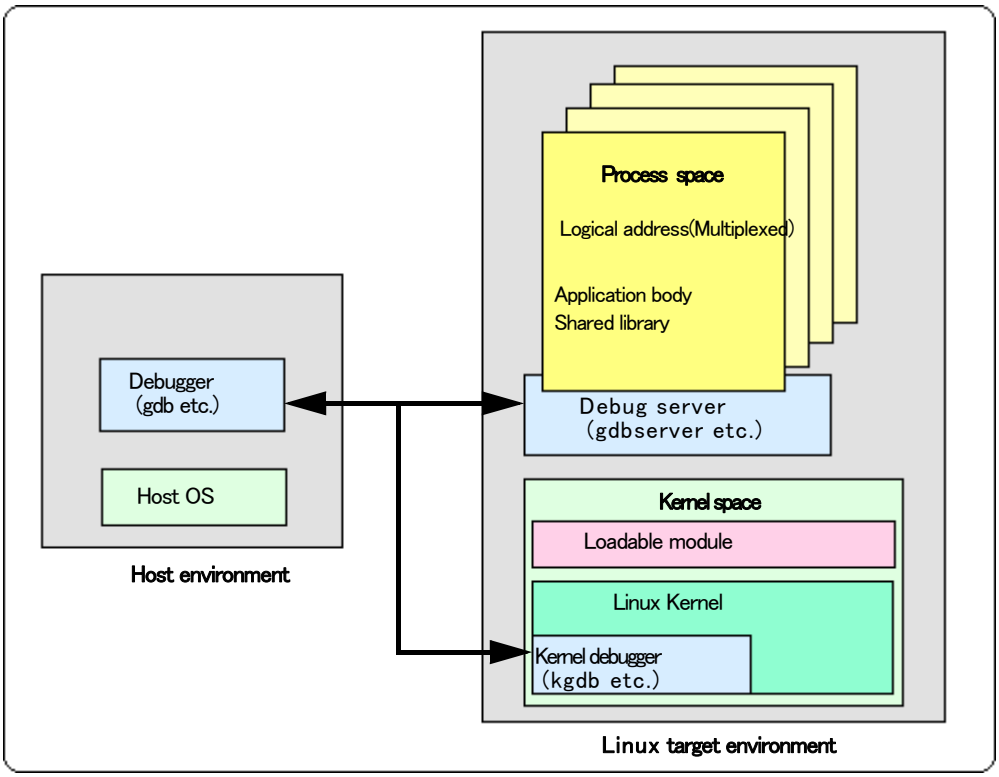
If a conventional ICE and debugger for embedded system development are used as is, they identify addresses that are not actually used, and so proper debugging may not be possible.

(3) Application (process)

Normally, logical addresses and physical addresses do not correspond to each other one to one, and logical addresses are used differently in multiple programs. Because of this, it is very difficult to identify addresses from the outside, and so a conventional ICE and a debugger for embedded system development cannot be used as is.

Also, a normal debug environment for embedded Linux (see Fig. E-1) that uses the debug daemon utilizing an existing kernel debugger or the ptrace() system call has the following problems.

Fig. E-1 Example of a normal debugging environment for embedded Linux



● Difficult to perform consistent debugging

Debugging of an application running on the target Linux is performed by communicating between the debug server running on the target (debug daemon) and the host PC. On the other hand, debugging of a kernel/loadable module is normally performed by communicating between the kernel debugger included in the Linux kernel and the host PC.

Inevitably, different debuggers are used respectively for a kernel, a loadable module and an application. So when this debugging method is used, it is impossible to perform step-in execution within a kernel/loadable module when an application is running.

Also, the debug server running on the target provides most of its debugging features using the ptrace() system call of the Linux kernel. This means that the Linux kernel must be running for the debug server to function.

Accordingly, if a debug feature uses the debug server and the kernel breaks during kernel debugging, the states of an application (variable etc.) cannot be viewed since the debug server cannot work.

● Difficult to perform real-time debugging

While the kernel debugger is communicating with the debugger on the host PC, only the kernel debugger is running and interrupts are prohibited. So debugging of interrupt processing timing may differ from the actual behavior.

Also, even if a breakpoint stops a program on the debug server, with which you are debugging an application, the kernel/loadable module operation does not stop in real-time.

● Impossible to use high-performance debugging assistance features

If you are using a kernel debugger or a debug server that provides debugging features only as software programs, the following features cannot be used.

・Real-time execution history (real-time trace function, etc.)

・Hardware breaks and trigger condition settings

・High-speed download of programs

E-2 Debugging methods provided by PARTNER

Linux for PARTNER provides two debug modes thanks to the expanded features of PARTNER-Jet and the debugger software PARTNER (for details on features of each debug mode, refer to ""E-3 Kernel mode debugging details (Page 253)"" and ""E-4 Application mode debugging details (Page 254)"")

【Kernel mode debugging】

PARTNER manages all different memory spaces for the Linux kernel, loadable modules and applications in physical memory. This makes it possible to equally debug all types of programs (the kernel, loadable modules and applications).

【Application mode debugging】

This mode manages only the virtual memory space of applications.
As PARTNER only manages logical addresses, it can perform debugging fully-compatibly with multi-process and multi-thread applications.

These expanded features enable PARTNER to perform the following types of debugging.

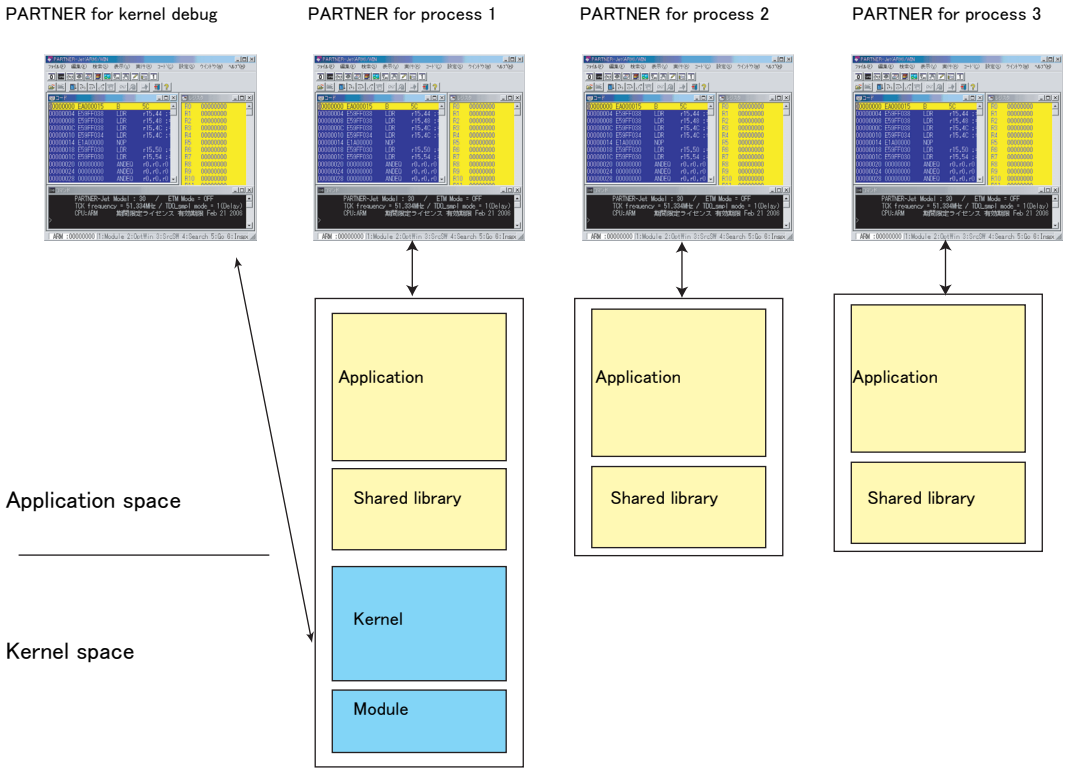
● All types of programs can be debugged using only one debugger (PARTNER)

You can perform debugging in all memory spaces of the Linux kernel, loadable modules (device drivers) and applications using only PARTNER.
You can consistently trace problems by checking the states of the contents of the kernel, loadable modules and applications. Consequently, you can easily debug the linkage between loadable modules (device drivers) and other programs, which is a very important process in product development.
You also can view the status of a specific part of an application (variable, etc.) when breaking the kernel or a loadable module.
As a result, because every type of debugging can be performed with one debugger, the Linux kernel, loadable modules and applications can be transparently debugged in the same manner as a debug environment provided by an embedded system such as a real-time OS (RTOS).

● Possible to perform debugging using multiple debuggers (PARTNER)

You can debug the Linux kernel, a loadable module and an application process/thread in separate PARTNER windows.

Fig. E-2 Multi window debugging



● Possible to perform real-time debugging

When a break is generated in an application, execution of the Linux kernel and loadable modules (device drivers) is also stopped at the same time (when debugging in kernel mode).

Any of the Linux kernel, loadable modules or applications can be debugged under the same conditions.

● Support of multi-process/multi-thread

Because PARTNER manages all virtual memory in physical memory, it does not need the assistance of the ptrace() system call when debugging an application and so multi-thread debugging is possible.

Also, because the application debug mode has a feature that allows to manage only virtual memory (refer to"E-4 Application mode debugging details (Page 254)"), complete multi-process/multi-thread debugging can be performed.

● Provides high-performance debug assistance features

Since the entire memory space is managed in PARTNER, you can use advanced debugging features such as hardware breaks and real-time trace for all types of applications (the Linux kernel, loadable modules and applications).

● Possible to perform high-speed transfer

A Linux kernel can be downloaded in a few seconds.

・2M - 4M Byte/Sec

【Other characteristics】

・Supports various debug information formats such as stubs+.

・Supports USB 2.0

E-3 Kernel mode debugging details

In kernel mode debugging, PARTNER-Jet manages all different memory spaces for the Linux kernel, loadable modules (device drivers) and applications in physical memory.

Even if a program uses virtual memory space, it always exists somewhere in physical memory while running.

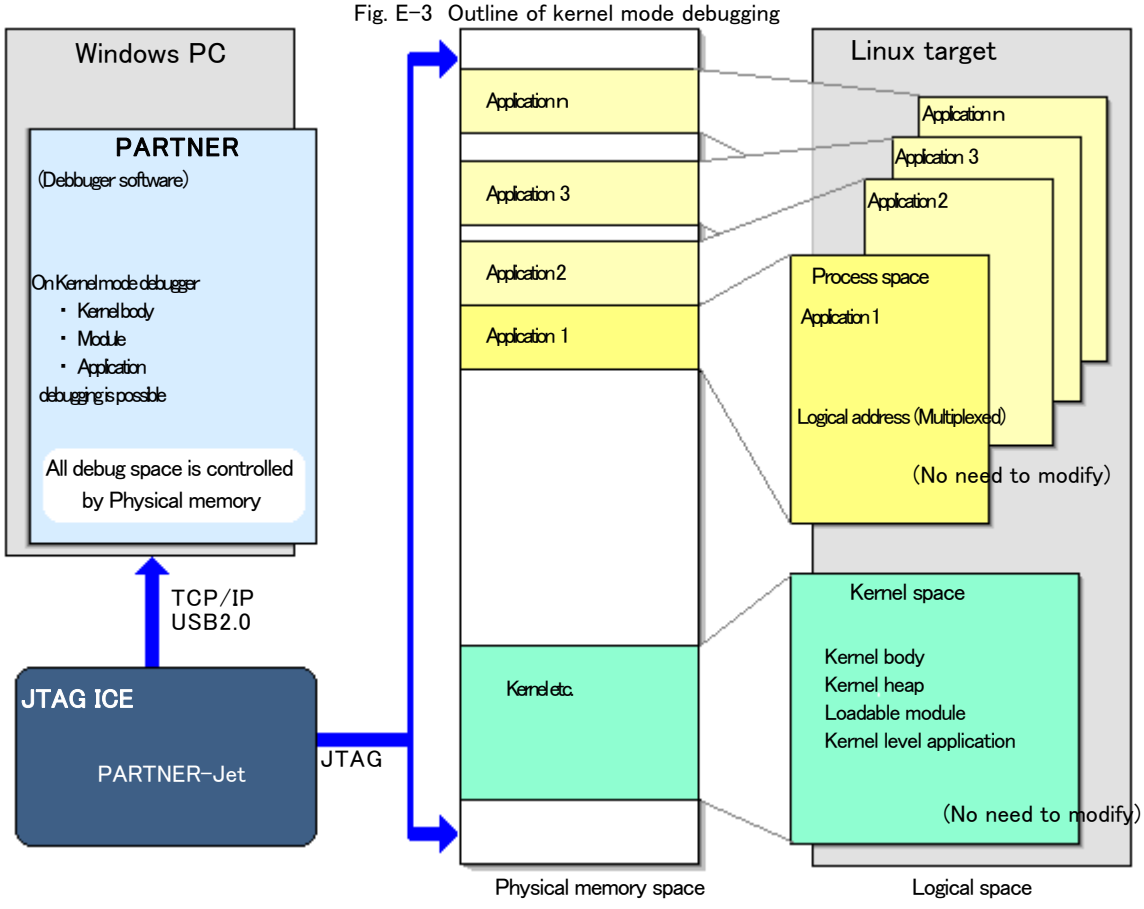
PARTNER automatically sets a correspondence relationship between logical address and physical memory.

This means that PARTNER-Jet interfaces all programs in physical memory, and so you can equally debug all types of programs (kernel, loadable modules (device drivers) and applications (processes)) without discriminating between them. In this mode, if one of the running applications stops (a break occurs), all other applications will also stop.

This mode also fully supports multi-process/multi-thread applications, so you can perform advanced debugging such as hardware breaks and real-time trace.

These features are implemented in PARTNER without depending on any specific OS, so it is not necessary to modify your Linux kernel for debugging. However, if you perform modification described in "2.2 Modifying and configuring the Linux kernel source (Page 54)" that is also described later in this chapter, you will be able to automatically attach loadable modules and applications to PARTNER, and thus development efficiency will also be improved.

Fig. E-3 Outline of kernel mode debugging



E-4 Application mode debugging details

In application mode debugging, only the virtual memory space of an application (process) is managed. However, the Linux kernel and loadable modules (device drivers) are also managed and debugged in physical memory in Application Mode, as with the case of the kernel debug mode.

In an OS that uses virtual memory, normally one virtual memory space is assigned to one process. At that time, allocated execution contexts are also assigned to processes, so, combined with corresponding virtual spaces, you can regard them as individual virtual machines.

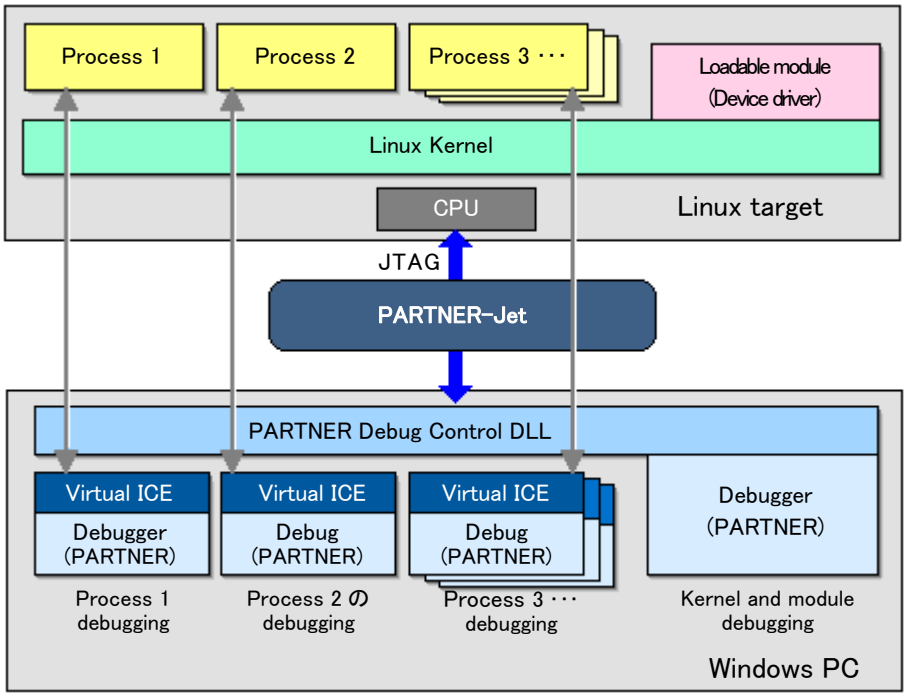
In application debug mode, PARTNER-Jet and the debugger software work together to create virtual ICE modules for each virtual memory space. Because debuggers run for each process on these virtual ICEs, each debugger will only perform debugging in its corresponding virtual memory space.

The characteristic of this virtual ICE module method is that you can perform completely independent debugging per virtual memory space.

"Independent debugging" means that you do not have to stop (break) other processes or the kernel to debug one of the processes, to view variables, or to change memory. Debugging in this method prevents communication overflow during application (process) debugging.

This mode also fully supports multi-process/multi-thread applications, so you can perform advanced debugging such as hardware breaks and real-time trace.

Fig. E-4 Outline of application mode debuggin



Appendix F Selecting the operation mode of PARTNER

For application debugging, PARTNER provides two debugging modes, namely the application mode and the kernel mode, and provides two thread debugging methods, namely the NON_ADD mode and the ADD mode (refer to "-OS option (Page 147)"). In other words, there are four patterns of combinations ("Kernel (NON_ADD) Mode", "Kernel ADD Mode", "Application (NON_ADD) Mode" and "Application ADD Mode") that you can switch according to the debug target and your operational preference.
"Kernel Add Mode" is recommended, but if you have trouble with selecting the mode, read this section to determine the appropriate one.

F-1 Characteristics of each mode and use of purpose

Kernel Mode

The kernel mode stops the CPU when a break occurs (for details, refer to "Technology description: Linux architecture and PARTNER (Page 247)"). This mode allows a user to uniformly debug all types of applications, namely the kernel, device drivers and user mode applications (in addition to kernel space programs, you also can debug user space programs).
During application debugging, you can easily enter into a system call, check out device driver processes, and then, return to the application.

Application Mode

When debugging an application, there are cases where you want to stop only an application without stopping the CPU and the kernel.
For example, you can use this mode in the following cases.

- You want a clock and a file sharing service to work while breaking an application.
- You want to debug a GUI application (X-Client) running under the X-Window system (you want the X-Server to work during debugging).

Normally, user mode debuggers that are implemented as an application on the target (GDB, etc.) are used to perform such debugging for individual applications, but there are overhead problems and various limitations. The application mode of PARTNER is a feature useful for application debugging, in which mode you can break each individual process while keeping the capability of a JTAG debugger.

NON_ADD Mode

If you want to automatically and forcibly separate PARTNER windows for each execution context when debugging a multi-thread/multi-process application, use this mode. When generating a new execution context, an unused PARTNER window is automatically assigned to it and a break occurs.
Also, breakpoints in user space are specific to each PARTNER window (execution of each context stops, even if multiple breakpoints are set at the same address).

ADD Mode

Unlike the ADD mode, PARTNER windows are not forcibly separated for each execution context. As automatic processing is not performed when new execution contexts are generated, you need to set breakpoints if you want to stop the debug target.

Also, breakpoints in user space are not closed each PARTNER window.

F-2 Startup method and necessary conditions

Launching in Kernel Mode

It assumes that modification instructed in "2.2 Modifying and configuring the Linux kernel source (Page 54)" has already been made on the kernel. In the Linux kernel setting menu (refer to "Kernel configuration (Page 57)"), make settings necessary for PARTNER, and then, create the Linux kernel (vmlinux) after the configuration settings have been made.

To perform debugging in Kernel Mode, specify "Kernel (NON_ADD) Mode" or "Kernel ADD Mode" as the OS debug mode (refer to " -OS option (Page 147)") in the launch configuration of PARTNER (refer to "Configuring and launching PARTNER (Page 75)").

In environments where the kernel source cannot be modified, manually modify Makefile so that debug information is added to the kernel object.

Launching in Application Mode

It assumes that modification instructed in "2.2 Modifying and configuring the Linux kernel source (Page 54)" has already been made on the kernel. In the Linux kernel setting menu (refer to "Kernel configuration (Page 57)"), make settings necessary for PARTNER, and then, create the Linux kernel (vmlinux) after the configuration settings have been made.

To perform debugging in Application Mode, specify "Application (NON_ADD) Mode" or "Application ADD Mode" as the OS debug mode (refer to " -OS option (Page 147)") in the launch configuration of PARTNER (refer to "Configuring and launching PARTNER (Page 75)).

Linux kernel configuration settings that are necessary to set depend on the debug target range. So enable [Debug information type] and [Enable patch for PARTNER debug] because the purpose of launching PARTNER in Application Mode is to debug an application, although it has nothing to do with the launch mode of PARTNER.

F-3 Guidance for operation mode selection

To guide a user to select an appropriate operation mode, Table F-1 shows what debug targets each mode is suited for.

Table F-1 Qualification of each debug mode

Debug target \ PARTNER mode	Kernel (NON_ADD) mode	Kernel ADD mode	Application (NON_ADD) mode	Application ADD mode
Kernel body debugging	◎	◎	○	○
Loadable module debugging	◎	◎	○	○
Application debugging	○	○	○	○
Multi-thread application debugging	○	○	○	○
Trace the process by stepping in from application into system call	◎	◎	△	△
Break and debug the process with running device drivers and server program running	×	×	◎	◎
Simultaneous stop multiple executing contexts at software break	◎	◎	×	×
Simultaneous debug multi-thread	△	○	△	○
Attach to running application	○	○	○	○
Shared library debugging	○	○	○	○
Multi-thread debugging in Shared library	○	○	○	○

Due to the limitation of the number of PARTNER windows that can be opened using the MULTI command (Page 173).

If you don't know which PARTNER operation mode you should use, we recommend "Kernel ADD Mode" in a comprehensive perspective.

F-4 Behavior and transition of Linux debug operation modes

The behavior of each operation mode and mode switching methods of PARTNER are organized in Table F-2.

Table F-2 Behavior and transition of each debug mode

Behavior etc. / Process mode	Multi-thread debugging method	Behavior when break occured in Application	-OS option Setting
Kernel mode	Debug a single thread with a single debugger window	CPU stops	LINUX
	Debug multiple thread with a single debugger window	CPU stops	LINUX_ADD
Application mode	Debug a single thread with a single debugger window	Only target thread stops (Kernel and other application are running)	LINUX_APP
	Debug multiple thread with a single debugger window	Only target thread stops (Kernel and other application are running)	LINUX_APP_ADD

Mode switching between the Kernel Mode and Application Mode requires rebooting the entire debugger. Switching between the ADD and NON_ADD modes can be done while operating in Kernel or Application Mode, from the command window of PARTNER (for details, refer to "PSID command (Page 168)").

・Transition from Single Mode to ADD Mode

```
PT>psid add
```

・Transition from ADD Mode to Single Mode

```
PT>psid non_add
```

Appendix G Supplement for kernel tree compiling

This section describes topics for compiling a kernel tree.

G-1 Suppressing compile optimization

When performing source-level debugging on a Linux kernel or a loadable module using PARTNER, sometimes it is difficult to understand if compile optimization has been performed, because executable lines are different from the description in the source and variable scopes are changed. Also there is a problem that source-level debugging cannot be performed on functions located in the _init section. You may be able to avoid this problem by building the debug-target while suppressing compile optimization.

Suppressing only specified files

Add the following line to Makefile of the directory in which the target source is stored, to remove the optimization option from CFLAGS.

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

[Example]

To suppress optimization of kernel/sched.c, add the following line in the kernel/Makefile.

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

Suppressing optimization causes kernel execution to slow down. If this suppression causes a problem in the target system, be careful when choosing files for which optimization suppression is performed.

Suppressing the entire kernel tree

If you compile the entire kernel tree without optimization, add the following line into $(TOPDIR)/Makefile to remove the optimization option from CFLAGS.

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

In some cases, the optimization option specified in CFLAGS is -Os instead of -O2.
In that case, modify it so that the -Os string will be removed.

Notes on kernel creation

In some parts of the kernel source, assembler language code is directly written using the asm declaration in the C source. If the assembler language code directly uses register arguments of a function in accordance with optimization, the kernel cannot run properly. In that case, modify the assembler language code or rewrite that function in C.

If compiled with optimization suppressed, an undefined symbol error may occur during linking. To avoid this error, check the following points.

- If an extern declaration is used for the inline function, change it to a static declaration.

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

- Add the description for the no-optimization case for parts determined by _OPTIMIZE_, using a preprocessor directive (such as #if,#ifdef).

Appendix H Tips for investigating the causes of segmentation faults

You frequently face the situation where an application freezes while it is running due to a bug, but where all you know is that it happened because a segmentation fault occurred.
Investigation of the cause tends to be difficult in an environment where collecting core dumps and operation logs cannot be easily done.

This section describes topics for investigating segmentation faults.

H-1 Method of thinking

The main reason why the causes of bugs cannot be easily found is that the application process is already over once the bug occurs.
A brief event flow from the occurrence of a bug to process termination is as shown below.

1. If the Linux kernel detects a segmentation fault, it sends the SIG_SEGV signal to the application process.
2. When the application receives the SIG_SEGV signal, the default operation of the application is a core dump.
3. If no signal handler is set for the application, the process terminates.

PARTNER has an edge, in that it can simultaneously debug both application processes and the Linux kernel. So it can seize the timing described in the above step 1 without having to modify the application program.

H-2 Tracing SIG_SEGV

The following procedure describes how to trace the SIG_SEGV signal.

(1) Set a software break at a position in the kernel where the SIG_SEGV signal will be detected.

You cannot predict when SIG_SEGV will occur, so always leave this setting on. The position where a software break should be set differs depending on the CPU.

```
【For ARM CPU】
"__do_suer_fault()" in "linux/arch/arm/mm/fault.c"
【For SH CPU】
The "bad_area" label in "linux/arch/sh/mm/fault.c"
【For MIPS CPU】
The "bad_area" label in "linux/arch/mips/mm/fault.c"
```

(2) When a break occurs, open a new PARTNER window.

Increase the number of debugger windows, by opening a new window using the start menu of Windows or the MULTI command (Page 173).

(3) Attaching to the current process

Attach to the current process by specifying the PID that is displayed in the register list in the PARTNER command window at that time.

```
PT>attach <pid> ↓
```

(4) Automatically loading debug information for a shared library

Load the debug information for the shared library of the attached process.

```
PT>linux load so ↓
```

(5) Investigating around the place where exception occurred

By following the procedure described above, you have prepared to investigate the status of the process "immediately before freezing".

Investigate around the place where exception occurred using AUD/ETM Trace or the K command (Page 174).

As a first step, check out the stack trace using the K 0 command.

Index

Symbols

-!! option 152
-!v option 151

A

ADD mode 106, 116, 147, 185
Application 61, 248
Application mode debugging 250, 254
Attach 133, 162, 163, 168
ATTACH command 162
Auto attach 253

C

Creating MAP File 193

D

Debug daemon 248
Debug information 58
Debug stub 66
Device driver 87, 248

E

-EUC option 157
Event trace 212, 220
EXIT command 175

F

fork() 185

G

G command 176
glibc 240

H

Hardware breakpoint 144
History display 126

I

INIT command 81
INS command 177
INSMOD command 91, 97, 166, 193

K

Kernel configuration 57
Kernel mode debugging 250, 253
_KMC_MODULE_DEBUG 90
_KMC_MODULE_NAME 90
_kmc_start 66
_kmc_start_debuger 66
kmc-support.c 63, 241

L

LAUNCH option 146
ld.so 243
Linux kernel 248
Loadable module 87, 248

M

-M option 193
MAP field (CFG) 141
MAPS command 162
MMU 61, 143
module_exit() 90
module_init() 90
MULTI command 174
-MULTI option 155
Multi-process 204, 212, 251, 254
Multi-thread 204, 251, 254

N

NON_ADD Mode 116

O

-OPTIMIZE option 156
-OS option 147

P

PARTNER 3
Process 61, 248
PS command 160
PSID command 168, 171
pthread 106, 185

Q

Q command 175

R

RMMOD comand 194

S

Shared library 122
-SK option 153
SNAME command 178
Software breakpoint 145
Status bar display 149
Support file 67

T

THREAD command 176

V

VMWare 3

X

-XGX option 150