

TRACE32

Android debugging

MDS Technology
Technical Support Team

목 차

- 1. Introduction**
- 2. Android Platform Overview**
- 3. TRACE32 실행 및 설정**
- 4. Bootloader**
- 5. Linux Kernel**
- 6. Linux Awareness**
- 7. Daemon(Process)**
- 8. Library**
- 9. Kernel Module**
- 10. Exception Debugging**
- 11. 추가 실습**

1. Introduction

본 과정을 통해 Android Platform에서의 TRACE32를 이용한 디버깅 방법을
익히도록 합니다

1. 교육 목표 및 교재 설명
2. BSP 및 Hardware 구성
3. Basic Linux Development Environment

1-1. 교육 목표 및 교재 설명

본 교재는 Android Platform내의 Native Framework와 Kernel 영역에 대한
디버깅 실습으로 구성되어 있습니다

이를 통해 TRACE32를 이용하여 부트로더, 커널, 커널모듈, 라이브러리 그리고 프로세스
디버깅까지 실습을 통해 학습하실 수 있습니다

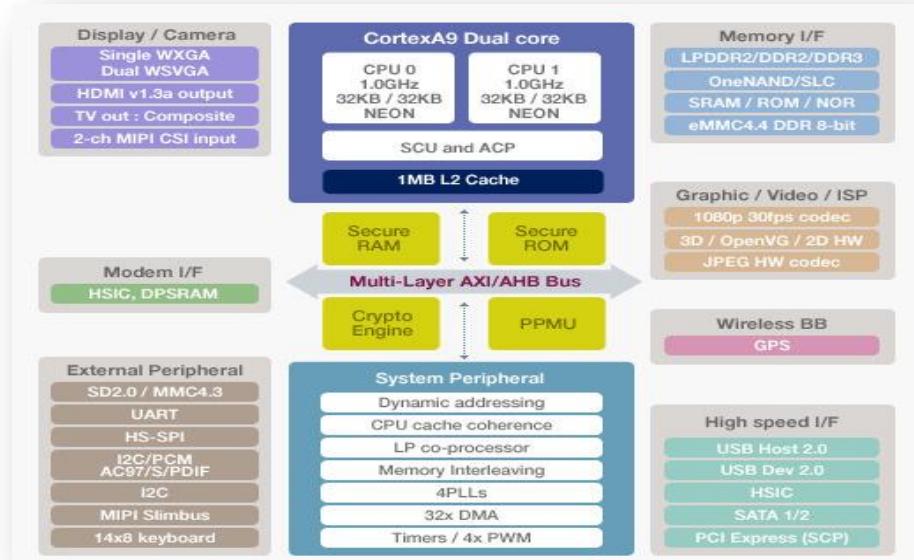


1-2. BSP 및 Hardware 구성

교육을 위해 설정에 사용되는 BSP 및 Hardware에 대해 확인 합니다

Bootloader / Kernel / Platform / File System

- U-boot
- Linux kernel 3.7.0
- Android 4.1.2(JellyBean)
- Ram Disk, YAFFS2 (EXT4)

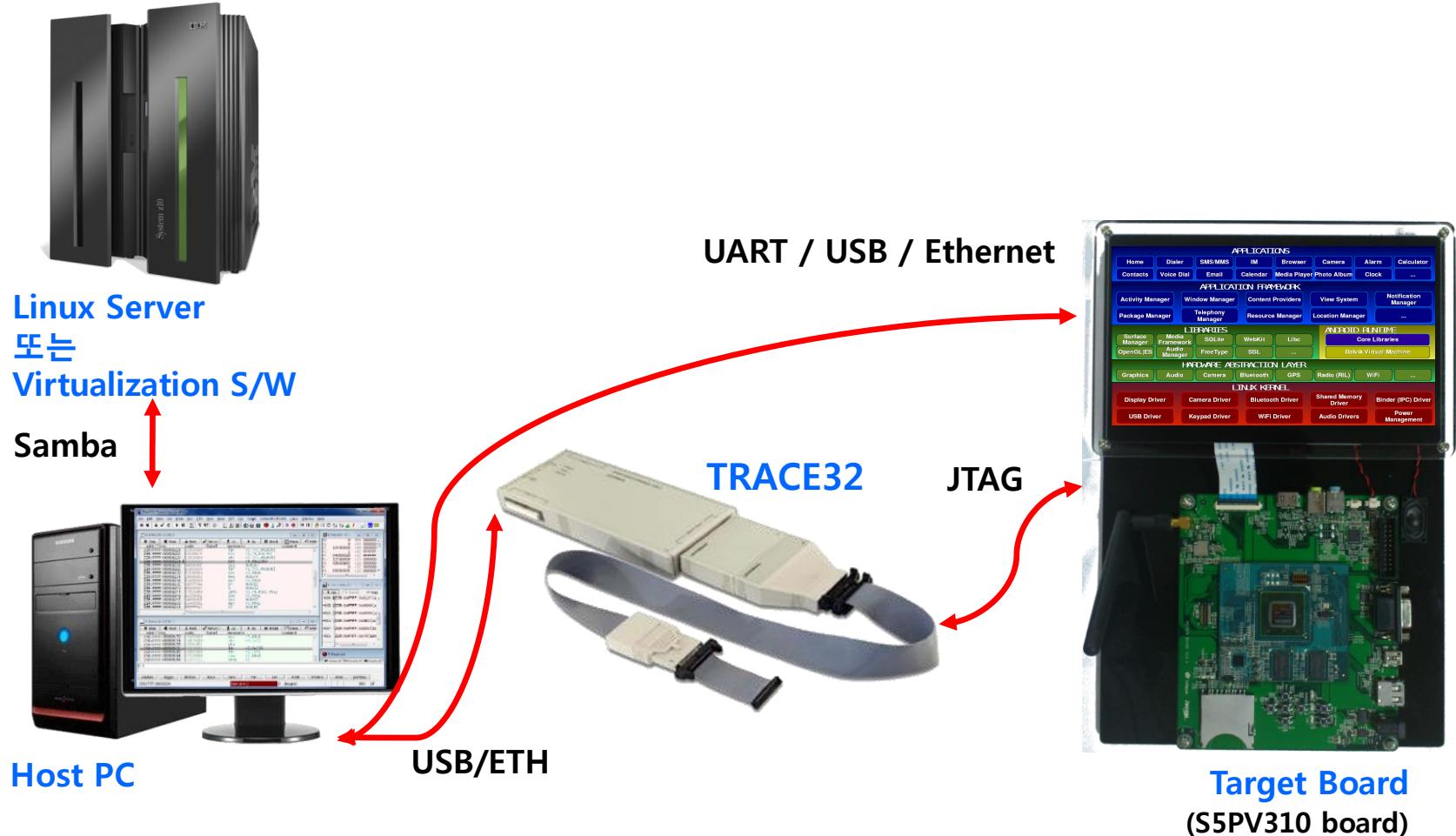


Hardware Specification

- Recommend TRACE32 ICD USB 2.0 Type & 2010년 4월 이후 S/W
- S5PV310 (Samsung Exynos4, ARMv7, Cortex-A9MPcore Dual)
- 1GB DDRSDRAM
- 7" 1024 x 600 LCD
- 2 Port SD/MMC card slot (Uses a SD card for Android Platform)

1-3. Basic Linux Development 환경

Linux 개발 환경 구성 시에 기본적으로 갖추게 되는 환경에 대해 알아 봅니다



2. Android Platform Overview

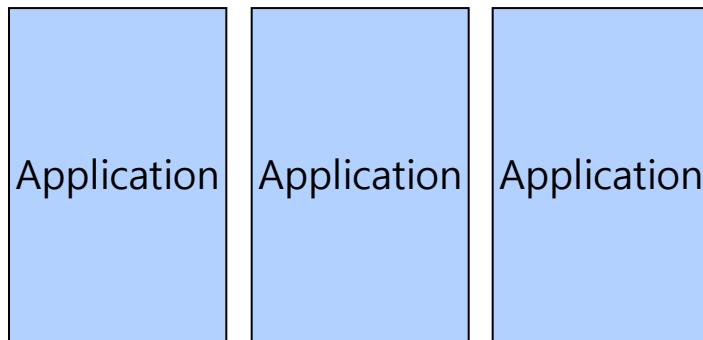
Android System의 원활한 디버깅을 위해 Android Platform에 대한 이해를 합니다

- 1. Linux Platform과의 차이**
- 2. Android Software Stack**
- 3. Android File System**
- 4. Android Boot Sequence**
- 5. Zygote Fork Sequence**

2-1. Linux Platform과의 차이

Linux와 Android의 차이점은 Framework & VM을 이용한 Application의 동작

Multi-Process (Linux)



Multi-Process (Android)



Framework & VM

Library

Middleware & Library

Kernel

Scheduler

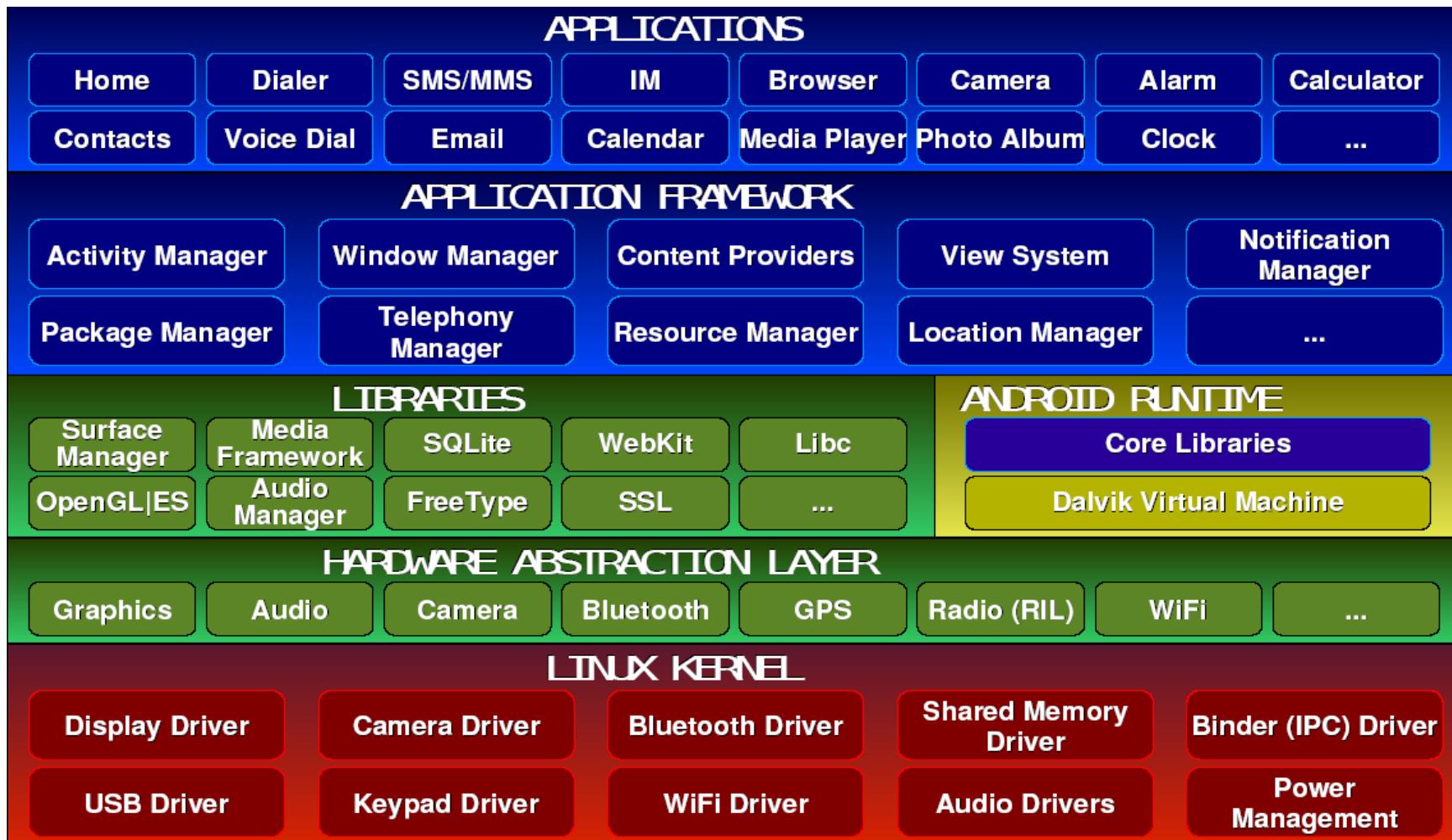
Kernel

Scheduler

[Q] Android Platform에서 Framework의 역할은 무엇일까요?

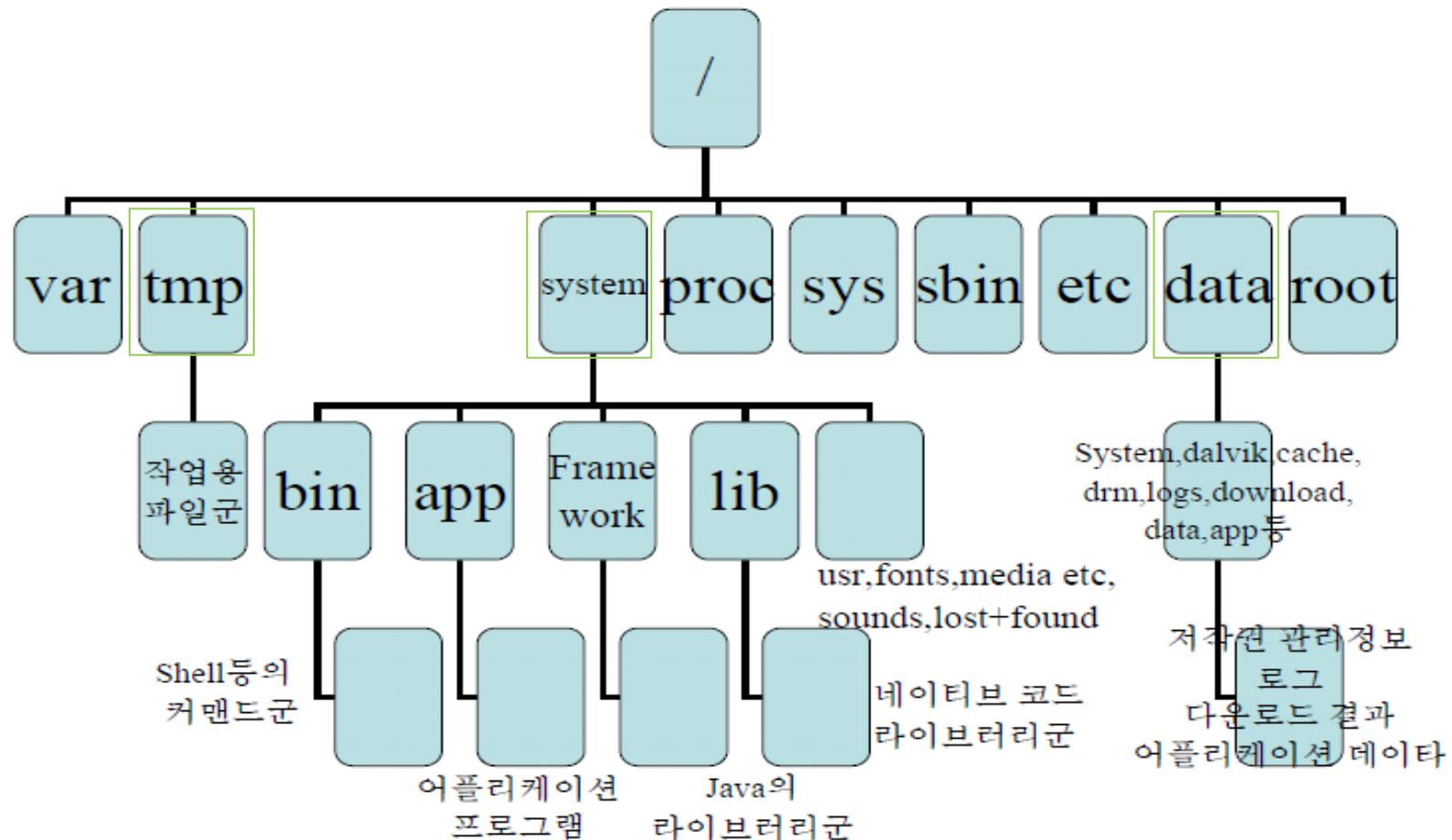
2-2. Android Software Stack

Android는 4개의 Stack으로 구성, JAVA/ Native Code 영역으로 구분



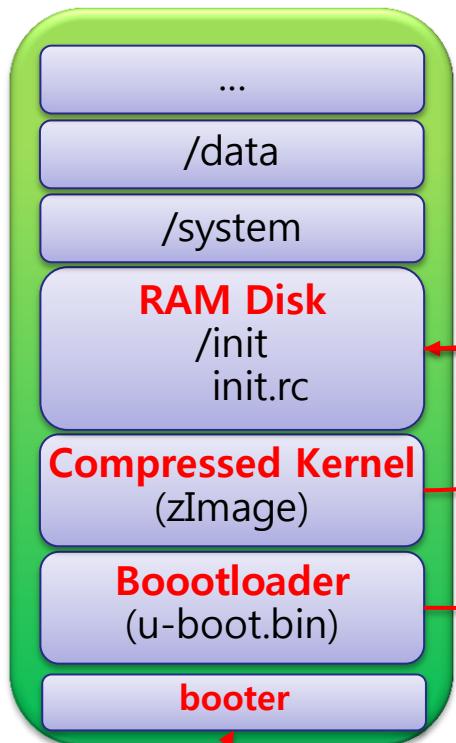
2-3. Android File System

Android에서의 Root file System은 일반 Linux의 Root file System과 비슷하나
Android를 위해 별도로 추가된 File System이 존재합니다(/system, /data)

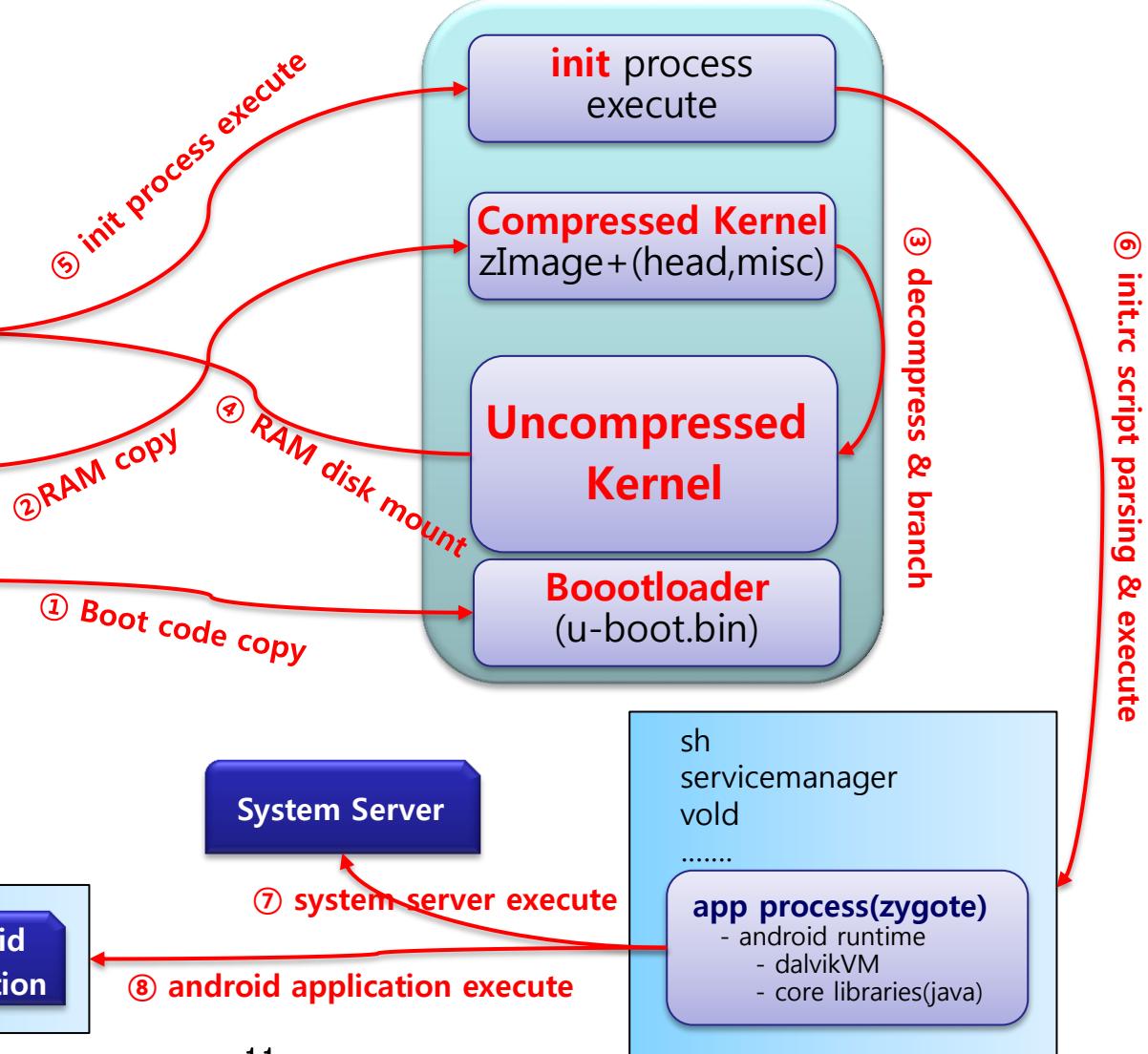


2-4. Android Boot Sequence

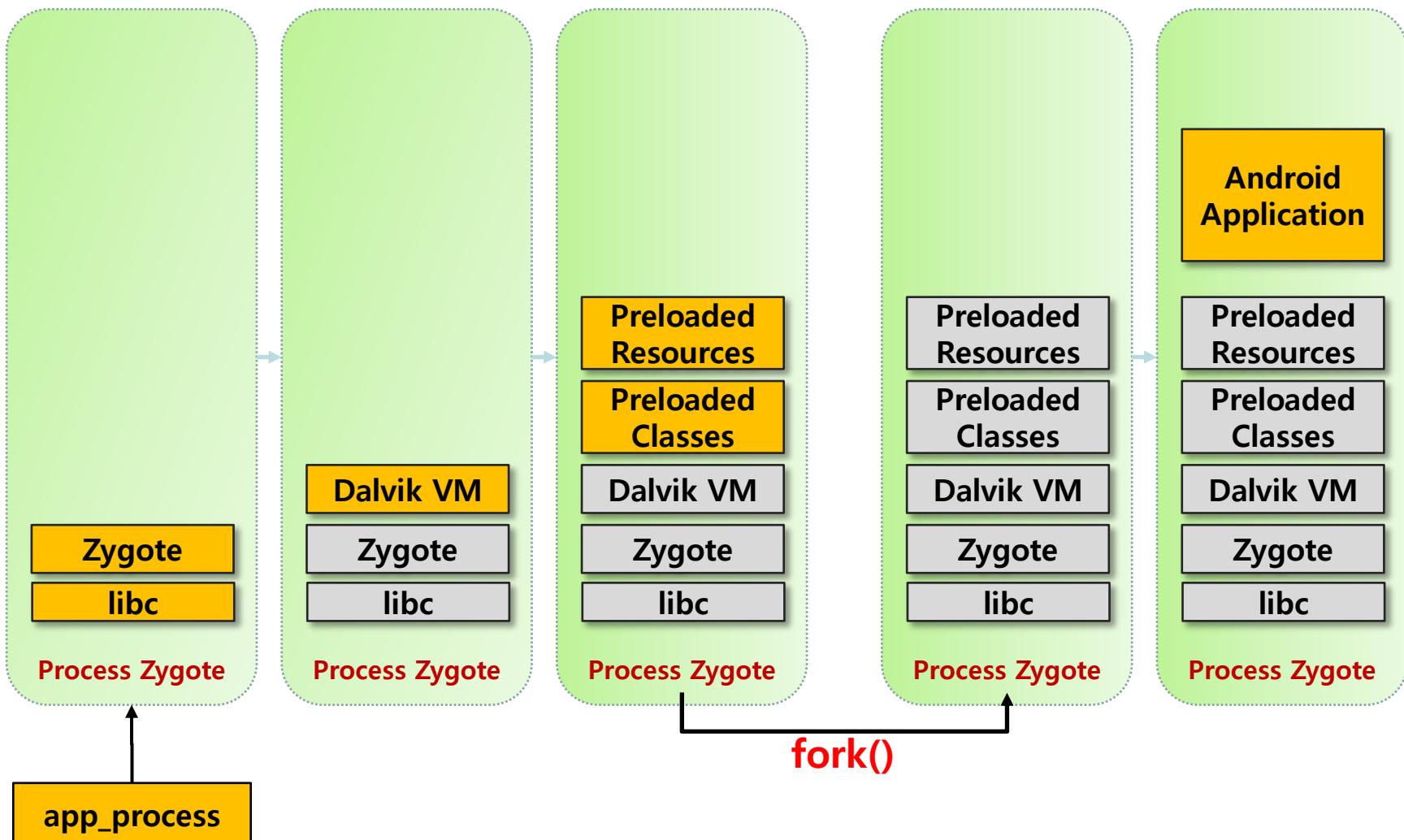
NAND Flash Memory



DRAM Memory



2-5. Zygote fork Sequence



3. TRACE32 실행 및 설정

Android 디버깅은 설정을 쉽게 하는 별도의 TRACE32 Package를 이용합니다

- 1. MTC Manager**
- 2. iTSP (integrated TRACE32 Support Package)**
- 3. TRACE32 실행**
- 4. iTSP 환경설정**
- 5. 실습**

3-1. MTC Manager

Target debug 설정과 TRACE32 S/W 다운로드를 위한 Program 입니다

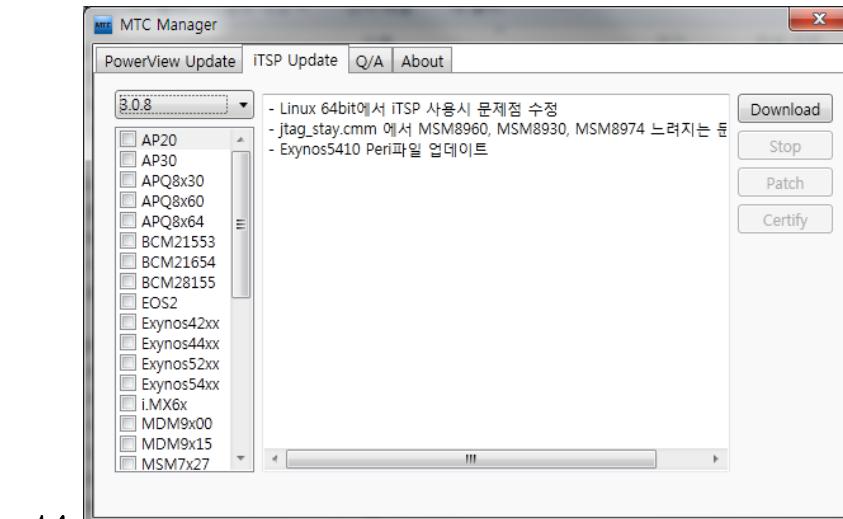
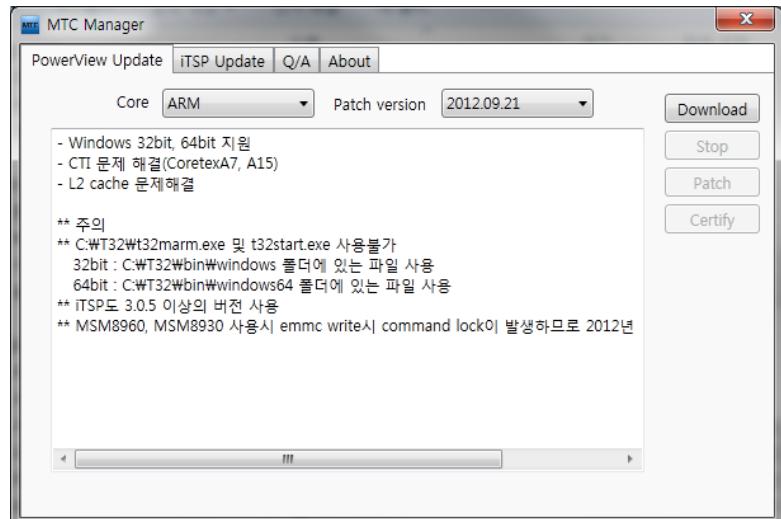
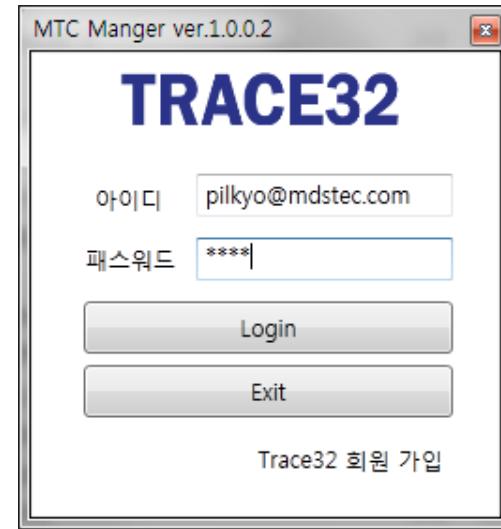
TRACE32 S/W(PowerView) download

- New & Old version 별 선택적 download

S/W(PowerView) update를 위한 patch download

CPU별 환경설정과 디버깅을 쉽게 제공하는 iTSP download

On-line을 통한 기술지원/제품문의 요청

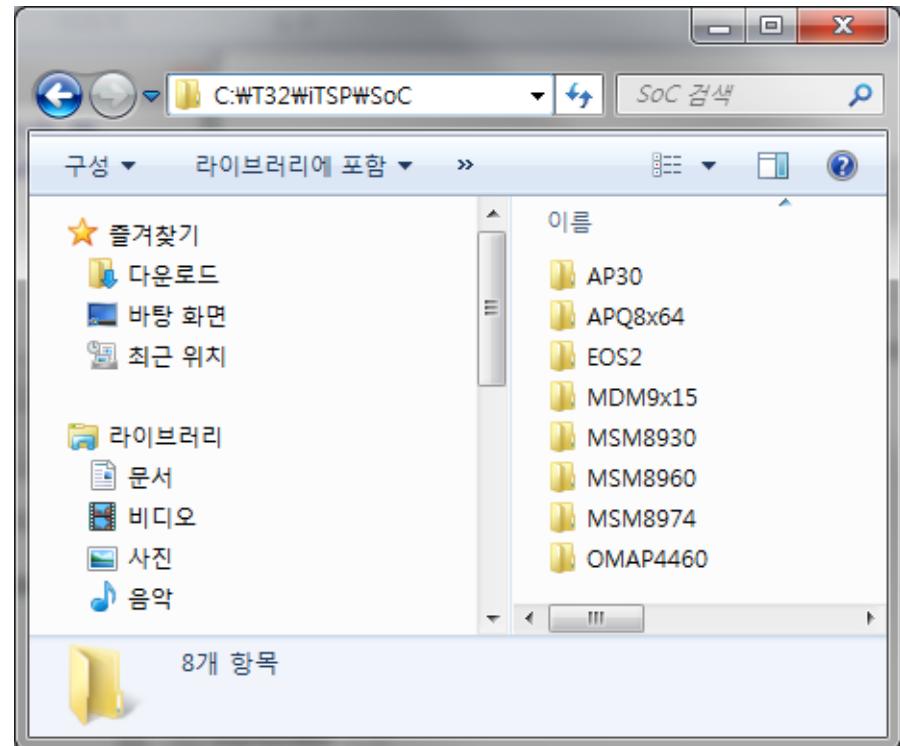


3-2. iTSP(integrated TRACE32 Support Package)

Android Platform 디버깅에는 iTSP(integrated TRACE32 Support Package)를 이용합니다.

iTSP를 통해 안드로이드의 복잡한 개발환경이 단순화 됩니다.

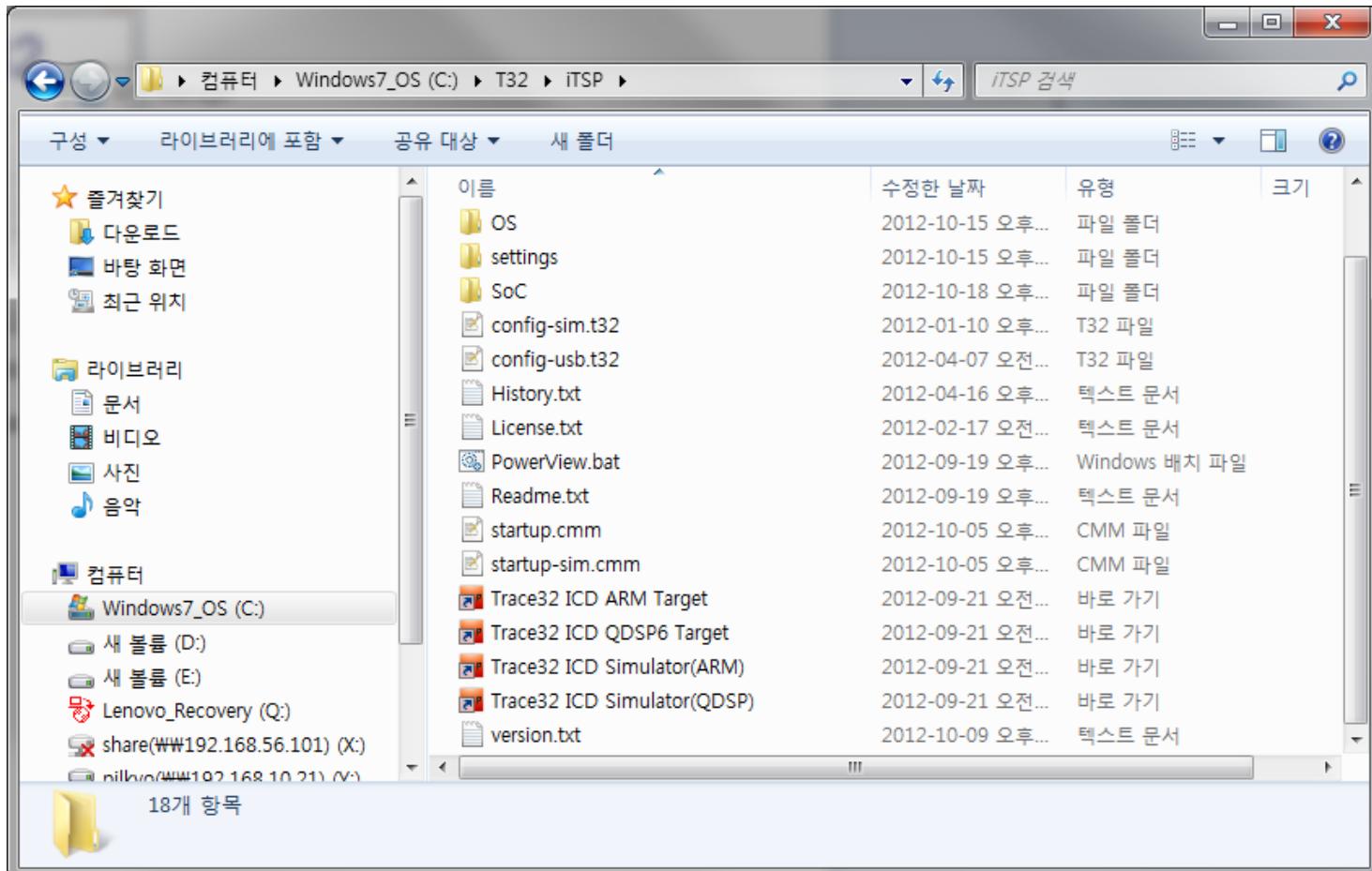
- 다양한 H/W Platform에 대한 환경설정이 용이
- Android Debugging에 꼭 필요한 쉬운 디버깅을 제공
- 여러 환경을 하나의 디렉토리에서 사용하기에 관리 용이



3-3. TRACE32 실행

iTSP를 통해 TRACE32를 실행하고 기존 환경과 다른 점을 살펴 봅니다

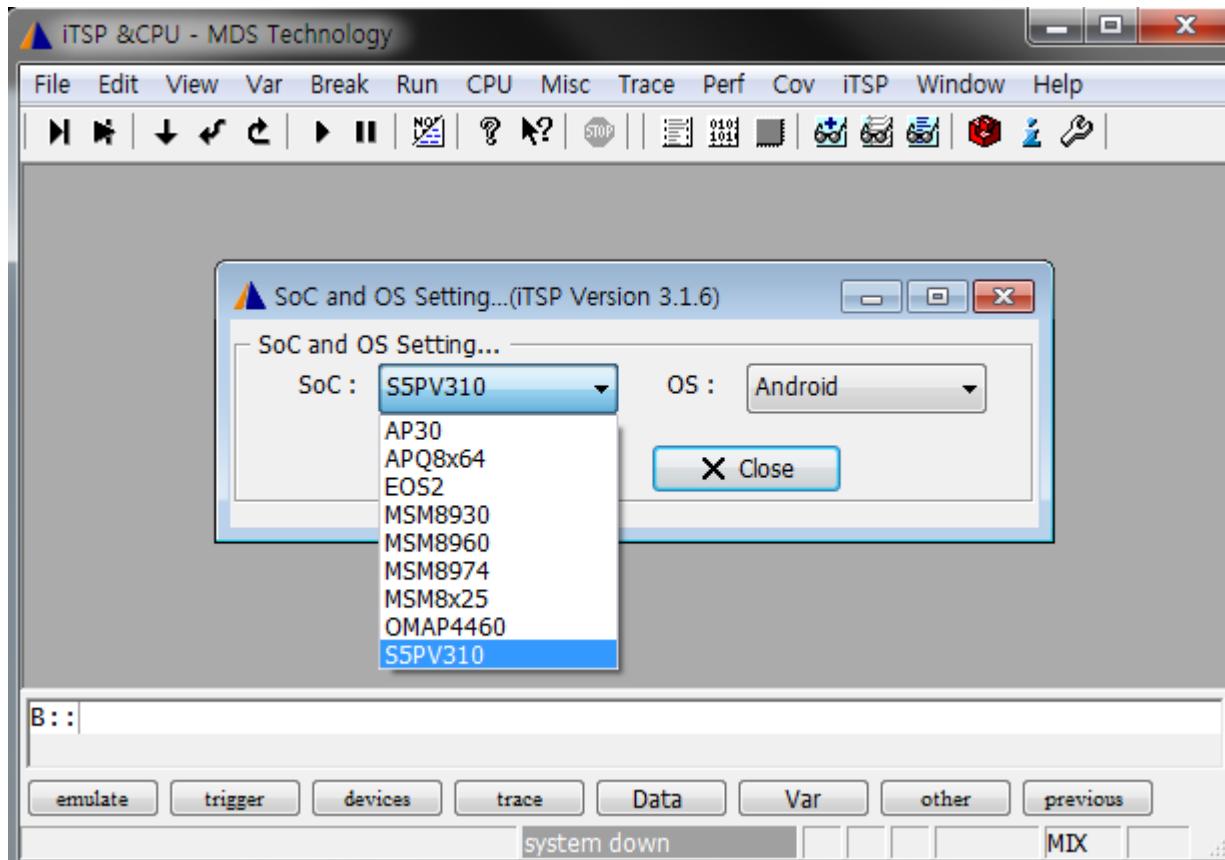
iTSP를 설치하면 해당 C:\WT32 아래에 iTSP 폴더가 생성되며,
내부에 포함되어 있는 실행 파일을 통해 TRACE32를 실행할 수 있습니다.



3-3. TRACE32 실행

iTSP를 통해 TRACE32를 실행하고 기존 환경과 다른 점을 살펴 봅니다.

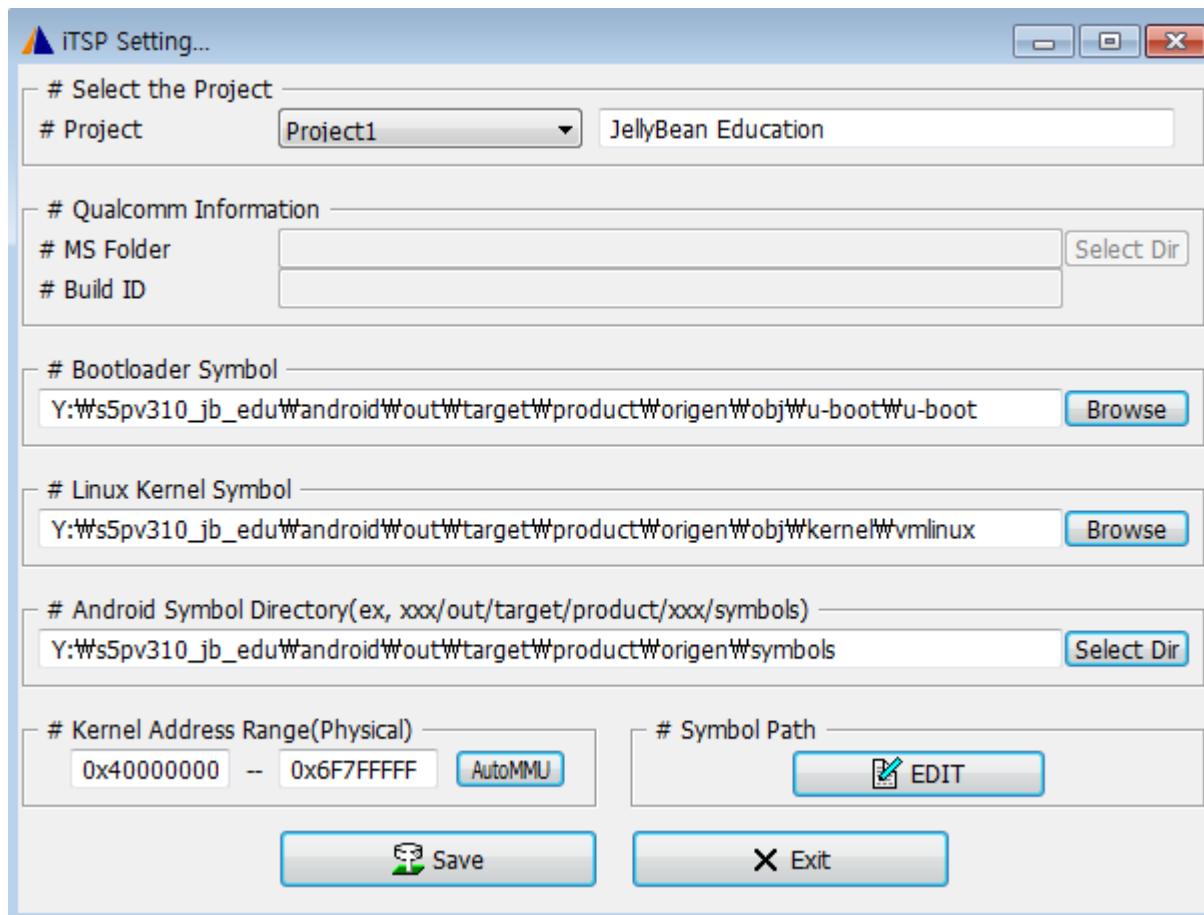
디버깅 하기 위한 CPU와 OS를 선택하면 해당 CPU에 대한 디버깅 환경을 갖출 수 있습니다.



3-4. iTSP 환경설정

본 실습환경에 대한 PowerView 설정을 하시고 Save 버튼을 클릭합니다

디버깅이 필요한 source directory에 대한 정보를 iTSP를 통해 설정하면
디버깅을 쉽게 진행할 수 있습니다



3-4. iTSP 환경설정

iTSP 화면 설정에 필요한 각 항목들의 의미를 통해 정확한 설정을 합니다

Project

- 여러 개의 Project 및 다양한 경로로 설정된 Project별 설정

Qualcomm Information

- Qualcomm CPU인 경우 boot_image에 대한 정보 입력

Bootloader Symbol

- Linux의 bootloader symbol을 입력
- 환경에 따라 u-boot, lk, sbl 를 선택

Linux Kernel Symbol

- Kernel Symbol 정보인 vmlinux 파일을 선택, Kernel 디버깅 정보 입력

Android Symbol Directory

- Android Platform의 symbol 경로 설정 (일반적으로 빌드하고 난 후에 symbols라는 디렉토리를 선택, <android root>/out/target/product/<Project Name>/symbols)

3-4. iTSP 환경설정

iTSP 화면 설정에 필요한 각 항목의 의미를 이해하도록 합니다.

Kernel Address Range (Physical)

- Kernel Image가 Load된 Physical Address를 Start address와 End address로 적어줌 MMU 정보를 통해 해당 정보를 입력합니다.
- MMU Size가 2GB일 경우 0x80000000 부터의 Physical Address를 적고 3GB일 경우 0xC0000000부터의 Physical Address를 적습니다.

본 교육 보드(MMU Size가 3GB)의 경우는 0x40000000과 0x67FFFFFF를 입력합니다.

Symbol Path

- 기본적으로 symbol이 생성되는 경로인 symbols 디렉토리를 제외한 다른 경로에 symbol이 있는 경우 추가적인 path 정보를 입력합니다.

Save

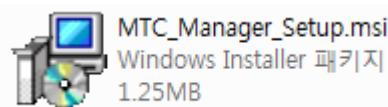
- Save 버튼을 통해 Text 파일로 저장이 되며 한번 저장된 정보는 TRACE32 종료 후 다시 실행해서 입력한 정보를 그대로 보여주게 됩니다.

3-5. 실습

실습을 통해 MTC manager와 iTSP 에 대해서 이해하도록 합니다.

1. MTC Manager 설치

MTC_Manager_Setup.msi 파일을 실행하여 프로그램을 설치 합니다.



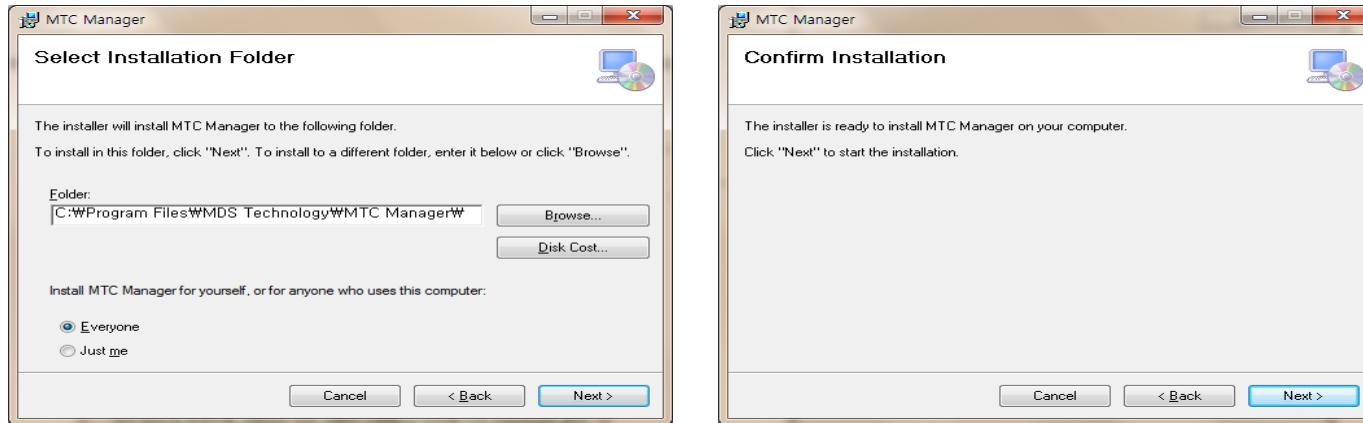
<http://www.trace32.com> [자료실]-[Etc]-[13번 MTC Manager]
<ftp://ftp.trace32.com/MTCManager>

1.1 msi 파일 실행 후 Next 클릭한다.

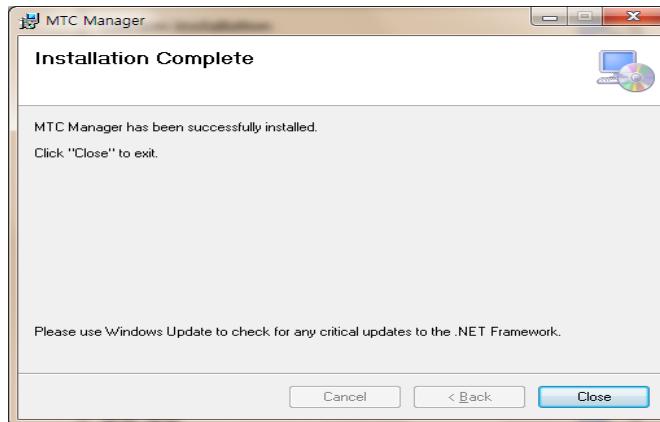


3-5. 실습

1.2 설치 할 디렉토리를 선택하고 Next를 클릭한다. (디폴트 디렉토리를 권장한다.)



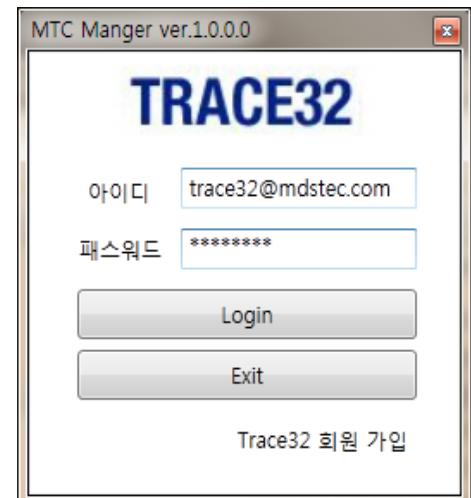
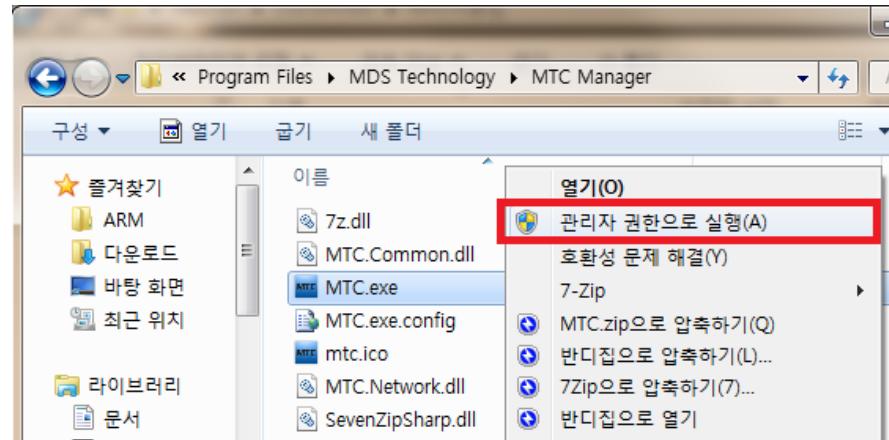
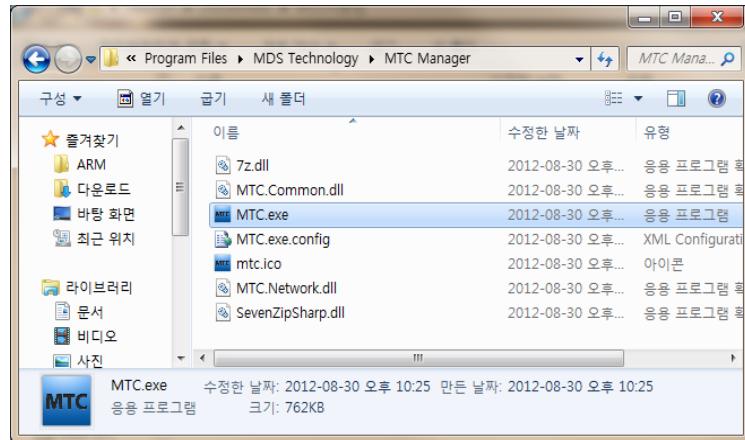
1.3 설치 완료 확인을 하고 Close 클릭한다.



3-5. 실습

2. MTC Manager 실행

2.1 C:\Program Files\MDS technology\MTC Manager\MTCE.exe 클릭하여 실행한다.

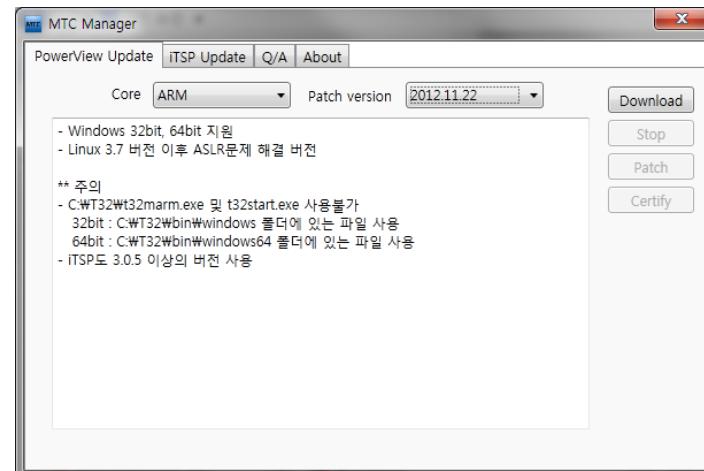


2.2 www.trace32.com에 가입한 ID(e-mail 주소)와 Password를 입력 후 로그인 한다.

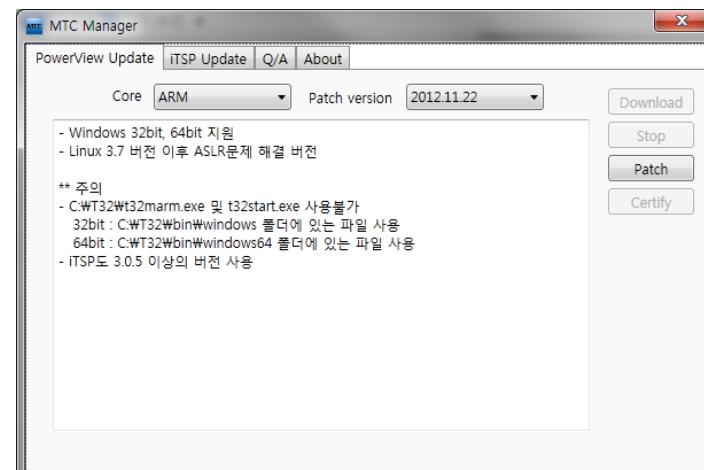
3-5. 실습

3. PowerView Update

- 3.1 로그인 완료 후 PowerView Update 탭에서 원하는 core와 patch version을 선택 한 후 Download를 클릭 한다.
(실습을 위해서 2012.11.22 version을 다운로드한다.)



- 3.2 다운로드 완료 후 Patch가 활성화 되고 Patch를 누르면 C:\WT32 폴더에 패치를 시작한다. 다운로드 후 반드시 patch 버튼을 선택해야지만 update가 완료된다.

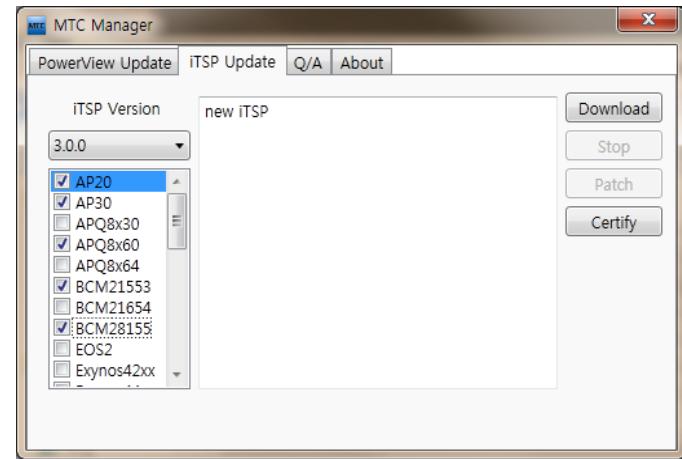


- 3.3 QDSP6 PowerView의 경우 인증을 퀼컴의 인증을 받아야 하므로 Certify를 눌러 인증요청을 해야 한다.
(인증 요청 후 MDS에서 확인 작업이 이루어 진후 다운로드 가능 함.)

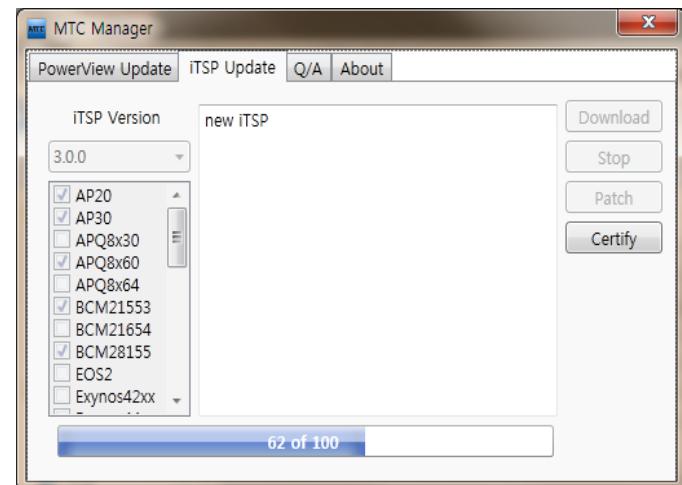
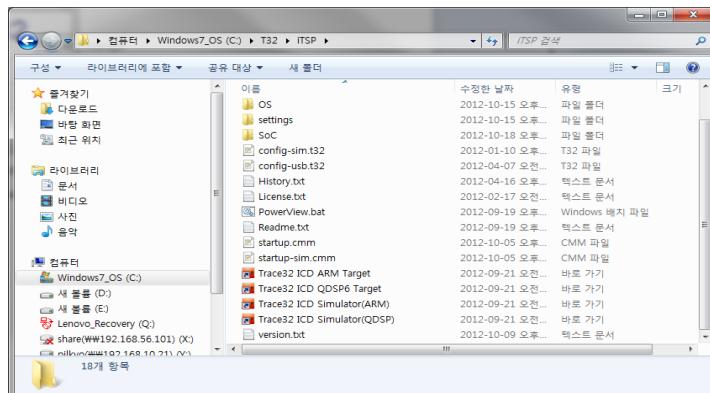
3-5. 실습

4. iTSP 설치

4.1 로그인 완료 후 iTSP Update탭에서 원하는 iTSP 버전과 사용하는 칩을 선택 후 Download를 클릭한다. 교육을 위한 iTSP는 S5PV310을 선택해서 다운로드한다.



4.2 다운로드 완료 후 Patch가 활성화 되고 Patch를 누르면 C:\WT32\iTSP 폴더에 패치를 시작한다.(기존 폴더는 삭제 됨.)



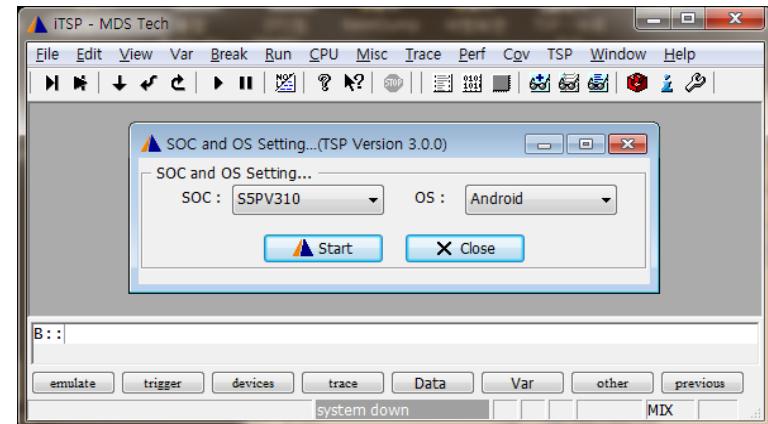
3-5. 실습

5. TRACE32 실행 및 iTSP 설정

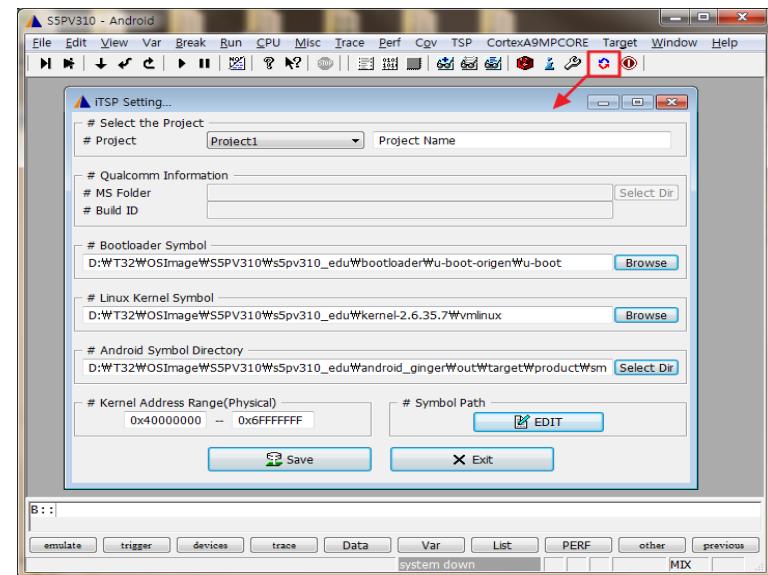
5.1 Trace32 ICD ARM Target.lnk 아이콘을 눌러 PowerView를 실행한다.

5.2 SoC와 OS 선택 창에서 디버깅하고자 하는 SoC와 OS를 선택한다.

교육용 환경의 SOC는 S5PV310이고,
OS는 Android를 선택한 후 start를 누른다.



5.3 프로젝트마다 사용될 iTSP 프로젝트 이름,
각종 심볼 경로 등 필수 정보를 설정해준다.



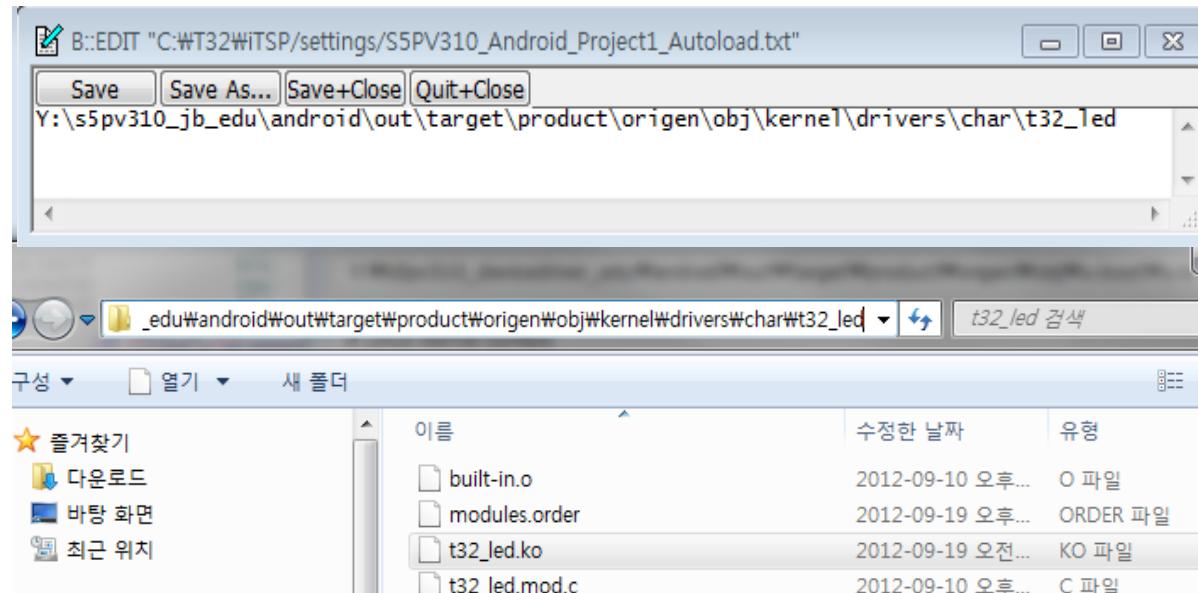
3-5. 실습

Android Symbol Directory에 입력한 경로를 기반으로 library와 daemon의 symbol은 자동으로 load된다.

하지만 NDK로 만들어진 library, kernel module 등 해당 경로에 존재하지 않는 symbol은 자동으로 load되지 않고 symbol을 찾으라는 dialog가 뜨게 되는데 이 부분을 해결하기 위해서는 위의 window 중 symbol path부분의 EDIT 버튼을 눌러 해당 symbol이 존재하는 절대경로를 입력하면 된다.

예로, kernel module인 t32_led.ko가 포함된 directory는 아래와 같다.

그렇다면 edit를 눌러 해당 경로를 입력해 주면 t32_led.ko의 symbol은 자동으로 load되게 된다.



4. Bootloader

부트로더 디버깅을 위해 그의 역할 및 종류에 대한 이해하고 디버깅을 합니다

1. Bootloader의 역할 및 종류
2. 실습

4-1. Bootloader 역할 및 종류

Bootloader가 하는 일과 동작의 흐름을 파악하여 디버깅에 이용합니다

Bootloader는 System Hardware 초기화하고, OS의 Kernel을 Memory에 Load하고 실행시키는 프로그램

- Bootloader의 위치 및 기능
 - ROM, Flash ROM, SRAM 등 정적인 메모리에 위치
 - 초기화 코드는 대부분 Assembler로 작성된다
 - 메모리 초기화 / 하드웨어 초기화
(직렬포트, 네트워크, 프로세서 속도, 인터럽트 등)
 - Kernel과 Ramdisk를 RAM에 Load 및 실행

Linux System에서 다양한 Bootloader가 존재하나 디버깅 방법은 동일

- U-Boot : 유니버설 플랫폼을 위한 Open Source Bootloader
- LK : 퀄컴, TI, nVidia 등 대부분의 칩밴더에서 Release한 Bootloader
- SBL: Samsung Bootloader
 - BOOTP/TFTP (RARP/TFTP)를 이용한 네트워크 부팅
 - Serial을 이용한 다운로드

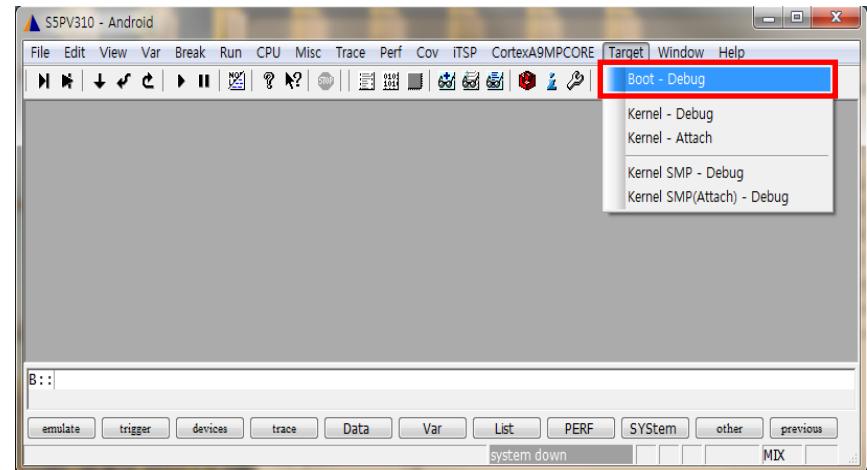
4-2. 실습

실습을 통해 bootloader 디버깅에 대해서 이해합니다.

1. Bootloader Debugging

1.1 Trace32 Menu에서

[Target] – [Boot – Debug] 선택한다.



1.2 타겟이 리셋되고 부트로더의
main함수인 board_init_f() 부터
디버깅 가능하다.

A screenshot of the Trace32 Data.List window. It shows assembly code for the board_init_f() function. The code includes instructions like 'NULL,' and '};'. It also contains C-like code with labels 265, 266, 278, 280, 282, and 284. The code uses labels such as 'bd_t *bd;', 'init_fnc_t **init_fnc_ptr;', 'gd_t *id;', 'ulong addr, addr_sp;', '#ifdef CONFIG_PRAM', 'ulong reg;', '#endif', 'bootstage_mark_name(BOOTSTAGE_ID_START_UBOOT_F, "boa...', '/* Pointer is writable since we allocated a register', 'gd = (gd_t *) ((CONFIG_SYS_INIT_SP_ADDR) & ~0x07);', '/* compiler optimization barrier needed for GCC >= 3', '_asm__ __volatile__("": : :"memory");', 'memset((void *)gd, 0, sizeof(gd_t));', 'gd->mon_len = _bss_end_ofs;', '#ifdef CONFIG_OF_EMBED', and '!!!'. The assembly code is color-coded with syntax highlighting.

5. Linux Kernel

Kernel의 구조와 Kernel 동작 시 사용되는 Memory 구조에 대한 이해를 기반으로 Kernel 디버깅을 할 수 있습니다

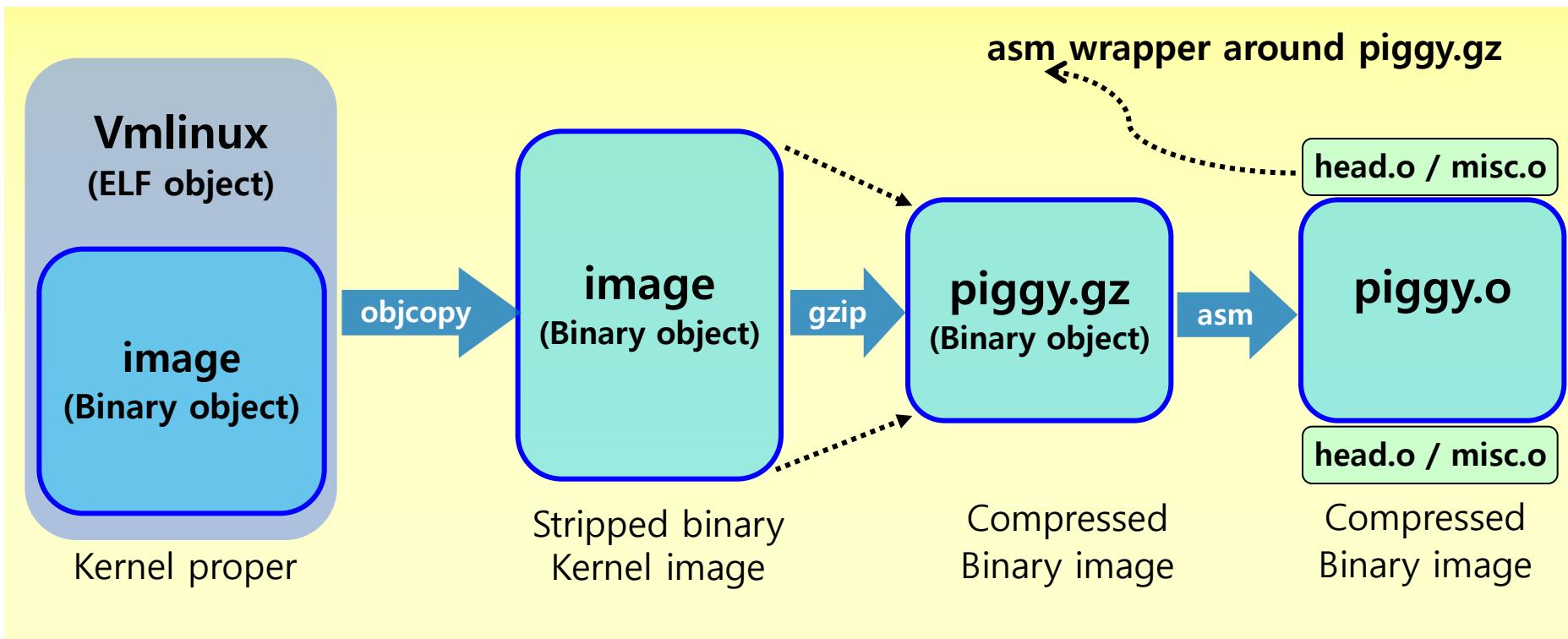
1. Kernel 이미지 구조
2. Linux Memory Model
3. Kernel 디버깅 실습 1 – kernel debugging 환경 실습
4. Kernel 디버깅 실습 2 – mmu page table 실습
5. Built-in Device Driver 디버깅

5-1. Kernel 이미지 구조

Linux Kernel의 경우 Compile이 완료되면 압축된 이미지 형태가 생성되는 구조

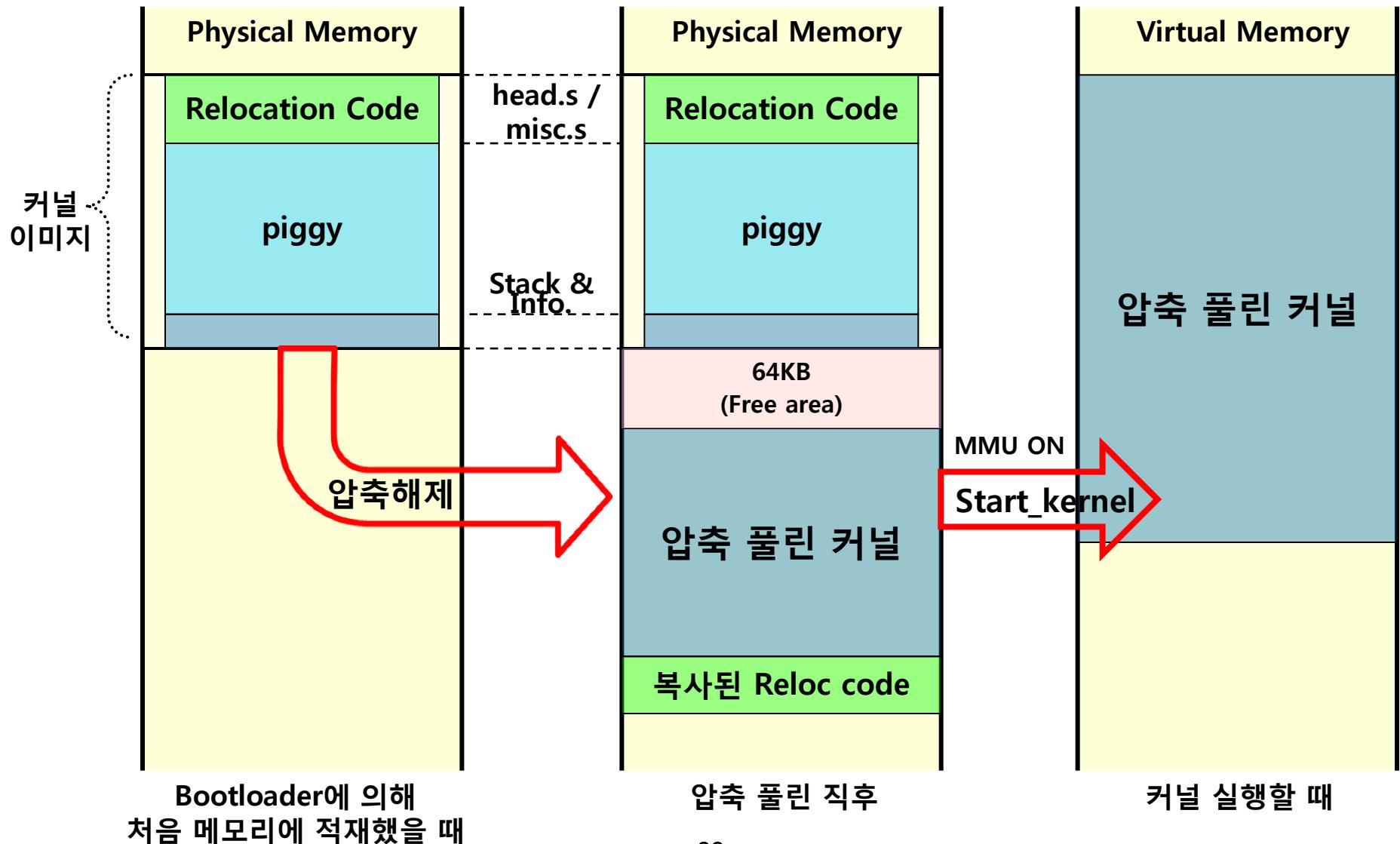
zImage 구성

- Linux kernel Image는 크기를 줄이기 위해 압축된 형태로 저장
- 압축 해제 시에는 지정된 물리주소에 해제
 - kernel/arch/arm/mach-exynos/Makefile.boot
 - zreladdr-y= 0x40008000 값 참조



5-1. Kernel 이미지 구조

압축된 이미지는 Bootloader 또는 Kernel에 의해 압축 해제 및 재 배치됩니다



Bootloader에 의해
처음 메모리에 적재했을 때

압축 풀린 직후

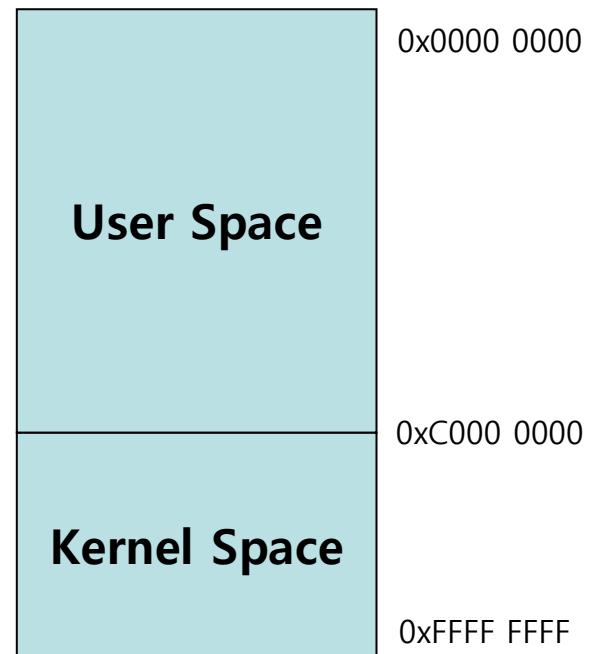
커널 실행할 때

5-2. Linux Memory Model

Linux kernel은 32bit Address 표현 방식으로 4GB의 Virtual 메모리 사용합니다

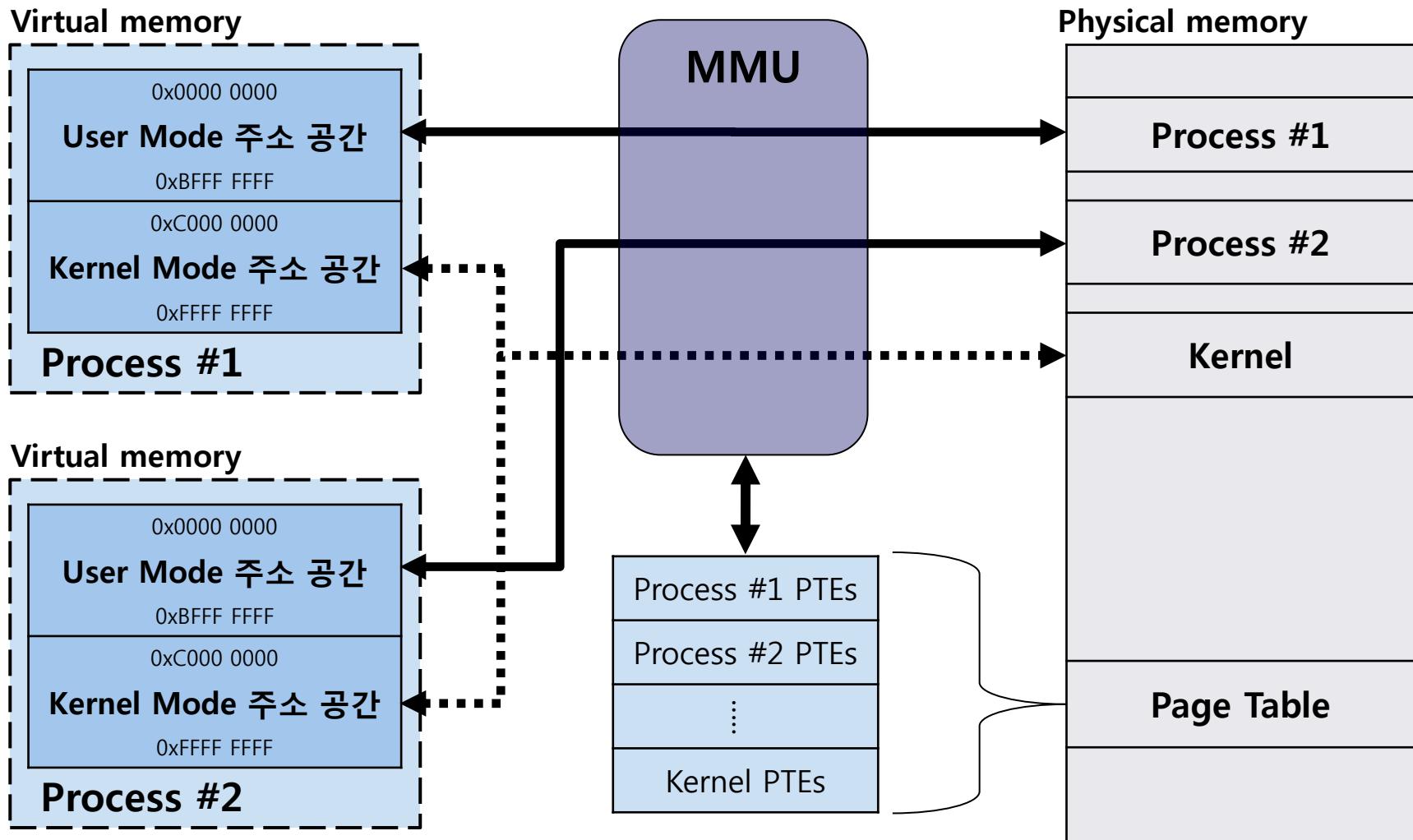
MMU 사용으로 인해 Physical Memory가 아닌 Virtual Memory로 동작함

- 현재 ARM 32bit Architecture로 인해 Virtual Memory 4GB를 사용함
- 통상 User space를 3GB, Kernel Space를 1GB 사용
 - ✓ 0x0000 0000 ~ 0xBFFF FFFF : User Space
프로세스가 사용자 모드에서 동작 중일 때의 어드레스로 프로세스가 사용자 모드이든 커널 모드이든 접근 가능
application, library, stack, heap 등 맵핑
 - ✓ 0xC000 0000 ~ 0xFFFF FFFF : Kernel Space
프로세서가 커널 모드에서 동작 중일 때의 어드레스로, 프로세스가 커널 모드에서만 접근 가능
kernel, device driver, gpio 등 맵핑



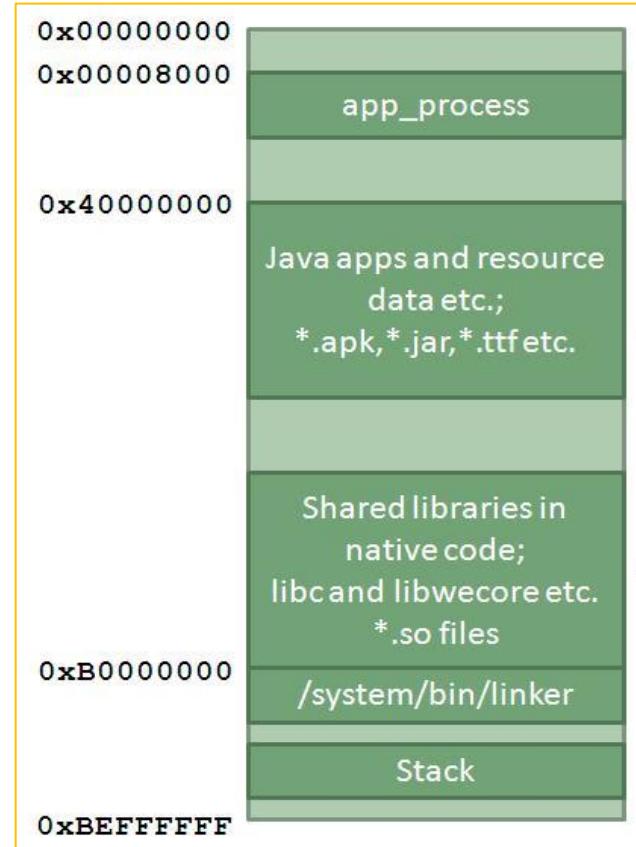
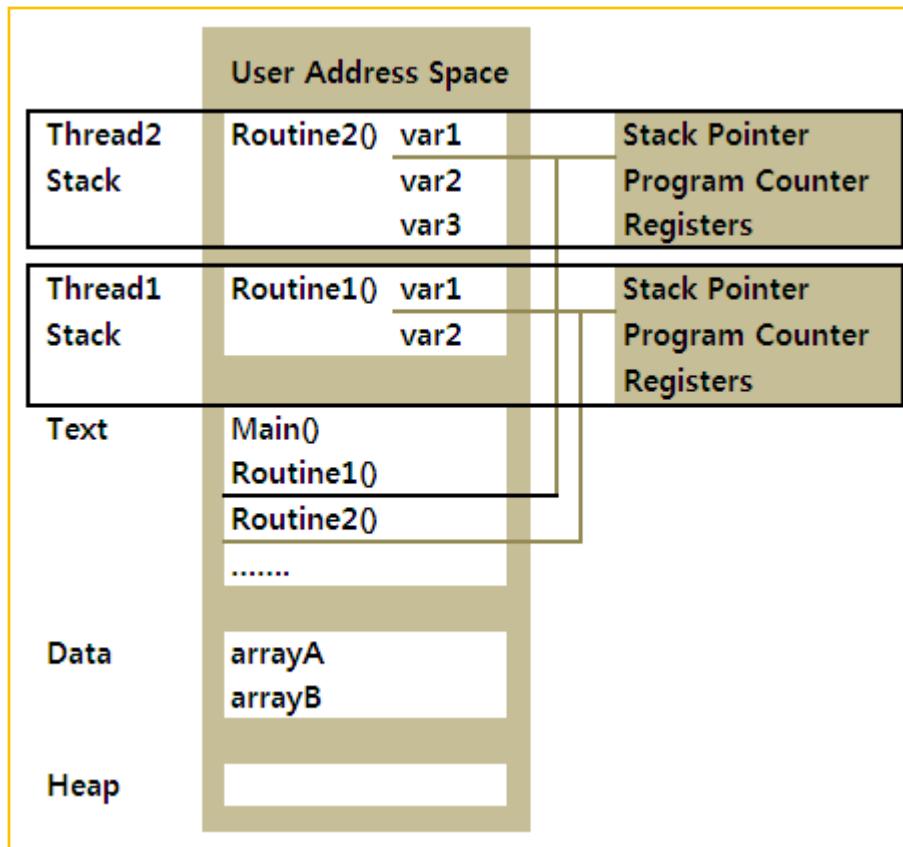
5-2. Linux Memory Model

Linux에서는 가상 메모리를 각 Process 마다 4GB씩 할당 받고, TRACE32에서는 이 메모리 공간을 **Space ID**라는 값으로 구분합니다



5-2. Linux Memory Model

Linux는 Multi-thread 방식을 사용하는데 Thread들은 메모리 영역 중 3GB의 User Space에서 할당된 Stack 영역을 나눠서 사용합니다



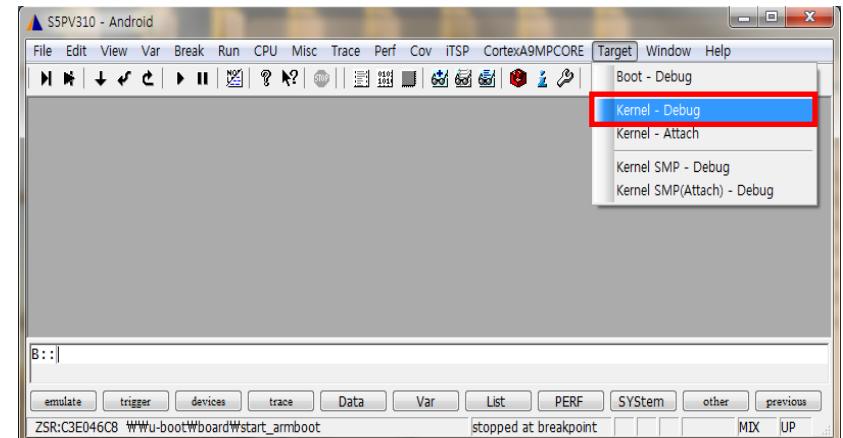
5-3. Kernel 디버깅 실습 1

Kernel 디버깅 시작 및 SMP에 대해서 이해합니다

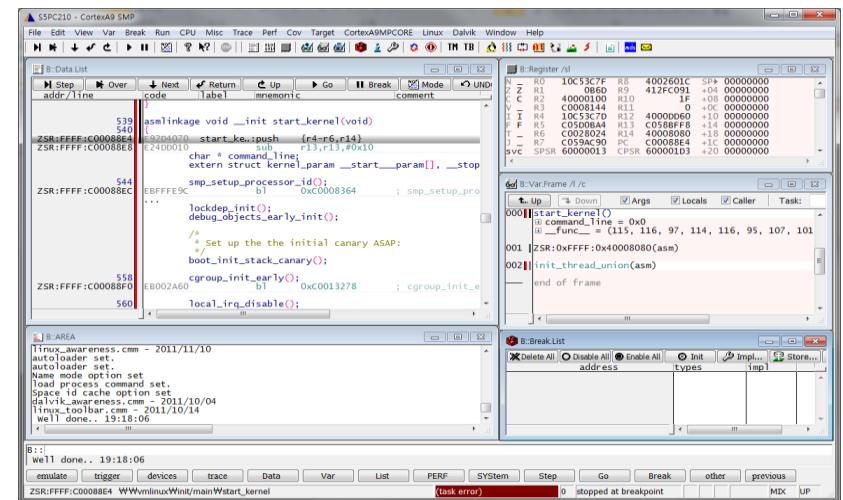
1. Single core Debugging

1.1 Trace32 Menu에서

[Target] – [Kernel – Debug] 선택한다.



* 타겟을 리셋 시키고 cortex A9MPCore 중 core 0만 JTAG으로 연결 한다.
start_kernel() 부터 디버깅 가능 하다.
이후에 SMP가 enable이 되면 core 1에서
수행하는 코드들은 디버깅이 불가능하다.



5-3. Kernel 디버깅 실습 1

1.2 Trace32 Menu에서

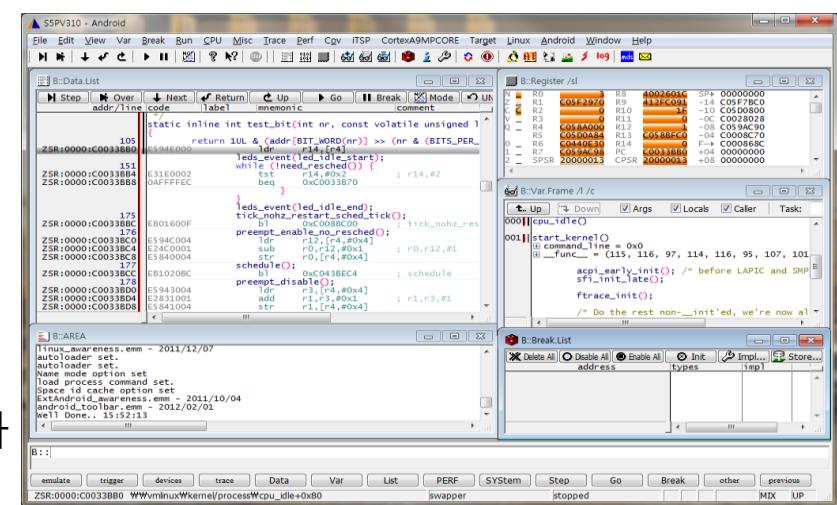
[Target] – [Kernel – Attach] 선택한다.



* 타겟을 리셋 시키지 않고 타겟이 실행 중이던 상태 그대로 core 0만 JTAG으로 연결 한 후 타겟을 멈춘다.

멈춘 시점의 코드부터 디버깅 가능하다.

마찬가지로 SMP가 enable된 후에 attach로 연결이 되었다면 core 1에서 수행하는 코드는 디버깅이 불가능하다.



** 만약 attach를 시도하는 core(여기서는 core0)가 sleep 및 suspend에 들어가 있는 상태에서는 attach가 불가능할 수도 있다.

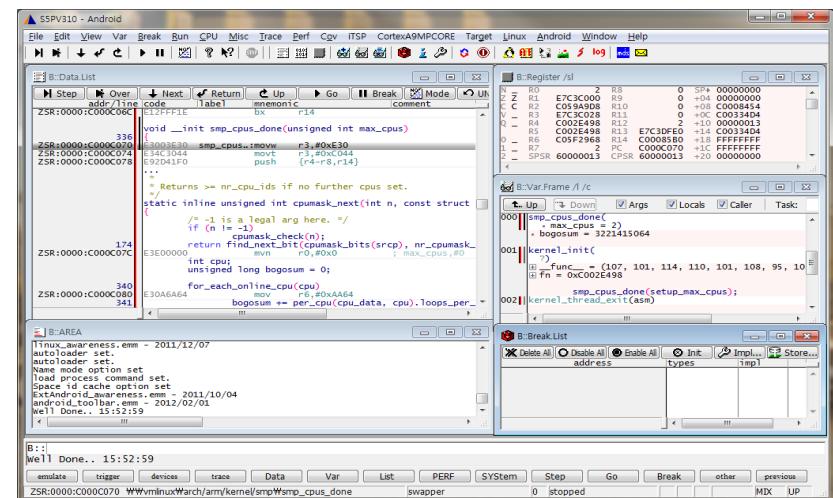
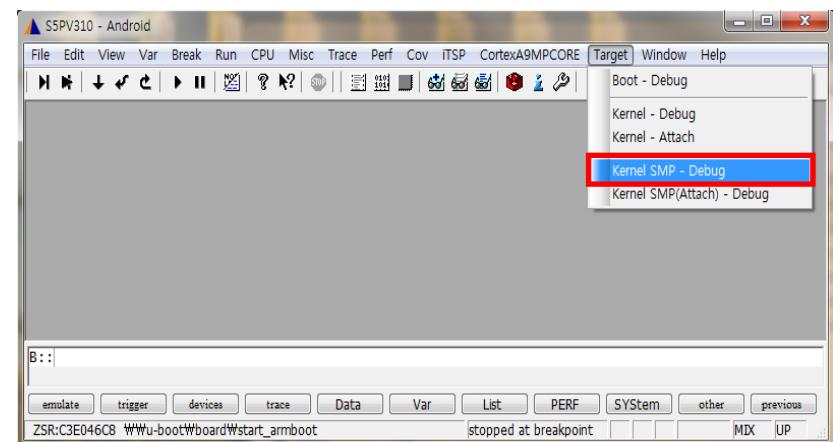
5-3. Kernel 디버깅 실습 1

2. Kernel SMP Debugging

2.1 Trace32 Menu에서

[Target] – [Kernel SMP– Debug]
선택한다.

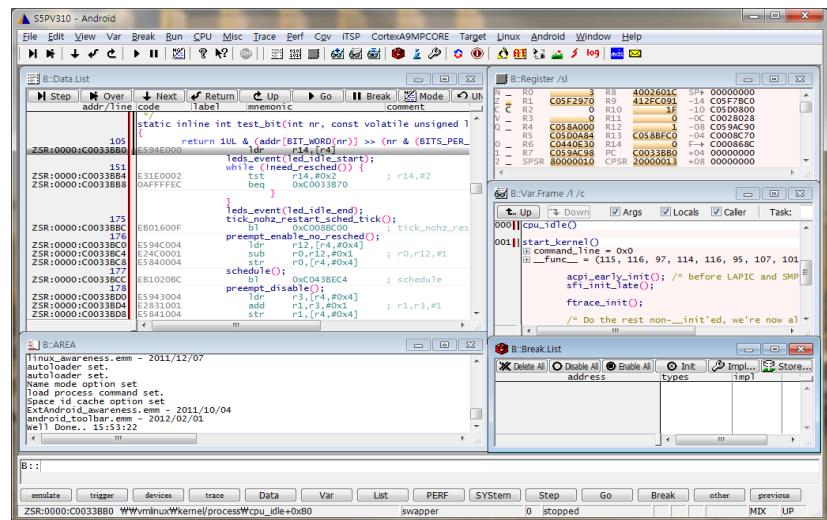
* 타겟 리셋 후 SMP(core0, core1)로
JTAG 연결하고 SMP가 Enable 되는 시점인
`smp_cpus_done()` 함수부터 디버깅 가능하다.
우측 하단의 core를 선택하는 부분이 활성화
되어 **core0 또는 core1 번을 선택 할 수 있다.**



5-3. Kernel 디버깅 실습 1

2.2 Trace32 Menu에서

[Target] – [Kernel SMP(Attach – Debug)]
선택한다.



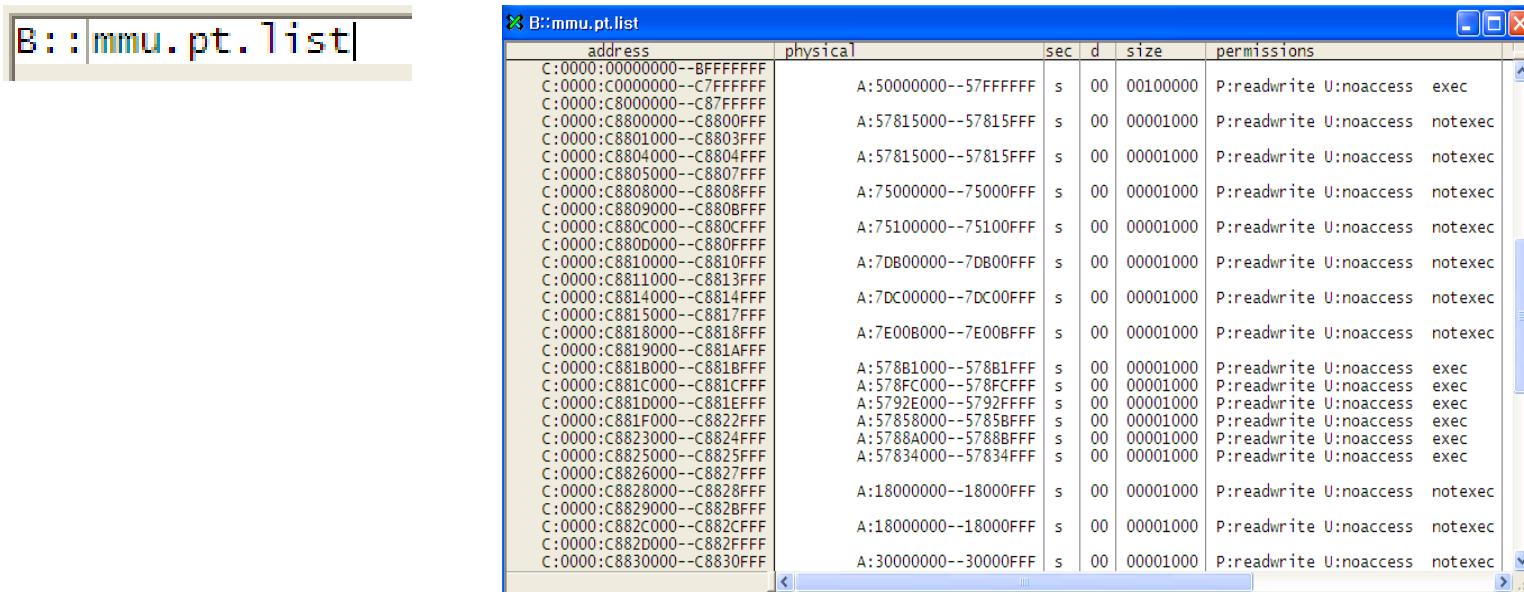
- * 타겟을 리셋 시키지 않고 타겟이 실행 중이던 상태 그대로 SMP(core 0, core 1)로 JTAG으로 연결 한 후 타겟을 멈춘다.
- ** 마찬가지로 두 개의 core중 하나라도 sleep 및 suspend에 진입했다면 attach가 불가능할 수도 있다.
- *** B:: do setup_window.cmm smp 을 명령어 창에 실행 하여 SMP View로 core 0과 core 1 같이 볼 수 있다.

5-4. Kernel 디버깅 실습 2

각 process 별로 존재하는 mmu 정보를 확인하고 이해합니다

1. MMU 정보 확인

MMU 정보를 확인하는 방법은 TRACE32 커맨드를 통해 아래와 같이 입력하면 확인할 수 있습니다. MMU를 확인하는 기본 명령어는 아래와 같다.



address	physical	sec	d	size	permissions
C:0000:0000000-0xFFFFFFFF					
C:0000:C0000000-CFFFFFFF	A:50000000-57FFFFFF	s	00	00100000	P:readwrite U:noaccess exec
C:0000:C8000000-C87FFFFFF					
C:0000:C8800000-C8800FFF	A:57815000-57815FFF	s	00	00001000	P:readwrite U:noaccess notexec
C:0000:C8801000-C8803FFF					
C:0000:C8804000-C8804FFF	A:57815000-57815FFF	s	00	00001000	P:readwrite U:noaccess notexec
C:0000:C8805000-C8807FFF					
C:0000:C8808000-C8808FFF	A:75000000-75000FFF	s	00	00001000	P:readwrite U:noaccess notexec
C:0000:C8809000-C880BFFF					
C:0000:C880C000-C880CFFF	A:75100000-75100FFF	s	00	00001000	P:readwrite U:noaccess notexec
C:0000:C880D000-C880FFFF					
C:0000:C8810000-C8810FFF	A:7DB00000-7DB00FFF	s	00	00001000	P:readwrite U:noaccess notexec
C:0000:C8811000-C8813FFF					
C:0000:C8814000-C8814FFF	A:7DC00000-7DC00FFF	s	00	00001000	P:readwrite U:noaccess notexec
C:0000:C8815000-C8817FFF					
C:0000:C8818000-C8818FFF	A:7E00B000-7E00BFFF	s	00	00001000	P:readwrite U:noaccess notexec
C:0000:C8819000-C881AFFF					
C:0000:C881B000-C881BFFF	A:578B1000-578B1FFF	s	00	00001000	P:readwrite U:noaccess exec
C:0000:C881C000-C881CFFF	A:578FC000-578FCFFF	s	00	00001000	P:readwrite U:noaccess exec
C:0000:C881D000-C881EFFF	A:5792E000-5792FFFF	s	00	00001000	P:readwrite U:noaccess exec
C:0000:C881F000-C8822FFF	A:57858000-5785BFFF	s	00	00001000	P:readwrite U:noaccess exec
C:0000:C8823000-C8824FFF	A:5788A000-5788BFFF	s	00	00001000	P:readwrite U:noaccess exec
C:0000:C8825000-C8825FFF	A:57834000-57834FFF	s	00	00001000	P:readwrite U:noaccess exec
C:0000:C8826000-C8827FFF					
C:0000:C8828000-C8828FFF	A:18000000-18000FFF	s	00	00001000	P:readwrite U:noaccess notexec
C:0000:C8829000-C882BFFF					
C:0000:C882C000-C882CFFF	A:18000000-18000FFF	s	00	00001000	P:readwrite U:noaccess notexec
C:0000:C882D000-C882FFFF					
C:0000:C8830000-C8830FFF	A:30000000-30000FFF	s	00	00001000	P:readwrite U:noaccess notexec

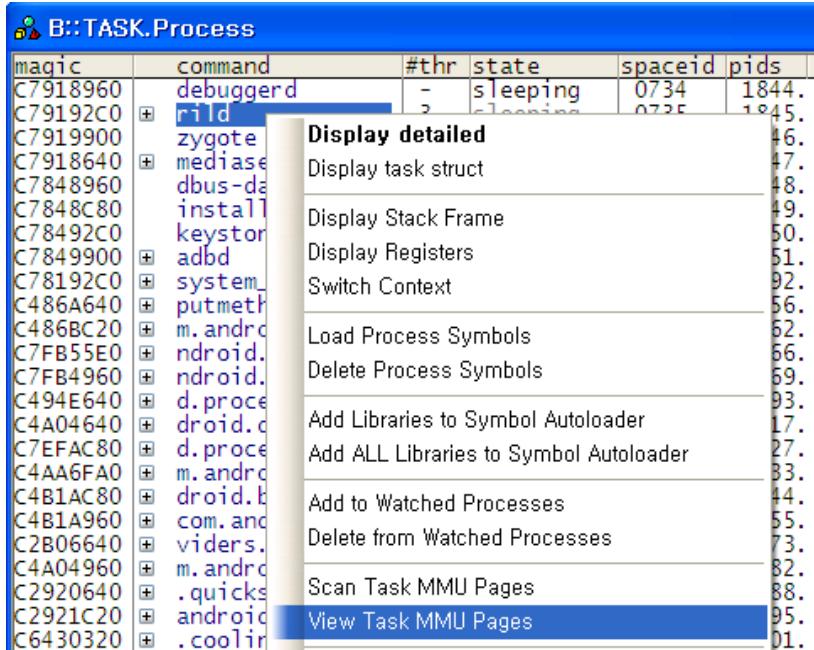
위의 window에서 확인하는 mmu mapping 정보는 기본적으로 kernel에 해당하는 mmu 정보만 확인이 가능하다. 즉, virtual memory 중 kernel 영역인 0xC0000000에서 0xFFFFFFFF에 해당하는 mmu 정보만 확인할 수 있다.

5-4. Kernel 디버깅 실습 2

만약 Process가 사용하는 모든 MMU 정보를 보고 싶다면 아래와 같은 명령어를 입력하면 됩니다.

```
B::mmu.tpt.list "rild"
```

혹은 Process의 경우는 Process 정보에서 메뉴를 통해 확인도 가능 합니다.



address	physical	sec	d	size	permissions
C:0735:00000000-0000BFFF	A:5382E000--5382EFFF	s	01	00001000	P:readwrite U:readwrite exec
C:0735:00000000-0000CFFF					
C:0735:00000000-40108FFF	A:549E8000--549E8FFF	s	01	00001000	P:readwrite U:readonly exec
C:0735:40109000-40109FFF					
C:0735:4010A000-0FFF					
C:0735:C0000000-C7FFFFFF	A:50000000--57FFFFFF	s	00	00100000	P:readwrite U:noaccess exec
C:0735:C8000000-C87FFFFFF					
C:0735:C8800000-C8800FFF	A:57815000--57815FFF	s	00	00001000	P:readwrite U:noaccess notexec
C:0735:C8801000-C8803FFF					
C:0735:C8804000-C8804FFF	A:57815000--57815FFF	s	00	00001000	P:readwrite U:noaccess notexec
C:0735:C8805000-C8807FFF					
C:0735:C8808000-C8808FFF	A:75000000--75000FFF	s	00	00001000	P:readwrite U:noaccess notexec
C:0735:C8809000-C880BFFF					
C:0735:C880C000-C880CFFF	A:75100000--75100FFF	s	00	00001000	P:readwrite U:noaccess notexec
C:0735:C880D000-C880FFFF					
C:0735:C8810000-C8810FFF	A:7DB00000--7DB00FFF	s	00	00001000	P:readwrite U:noaccess notexec
C:0735:C8811000-C8813FFF					
C:0735:C8814000-C8814FFF	A:7DC00000--7DC00FFF	s	00	00001000	P:readwrite U:noaccess notexec
C:0735:C8815000-C8817FFF					
C:0735:C8818000-C8818FFF	A:7E00B000--7E00BFFF	s	00	00001000	P:readwrite U:noaccess notexec
C:0735:C8819000-C881AFFF					
C:0735:C881B000-C881BFFF	A:57881000--57881FFF	s	00	00001000	P:readwrite U:noaccess exec
C:0735:C881C000-C881CFFF					
C:0735:C881D000-C881EFFF	A:578FC000--578FCFFF	s	00	00001000	P:readwrite U:noaccess exec
C:0735:C8821000-C8822FFF	A:5792E000--5792FFF	s	00	00001000	P:readwrite U:noaccess exec
C:0735:C8821F000-C8822FFF					
C:0735:C8823000-C8824FFF	A:57858000--5785BFFF	s	00	00001000	P:readwrite U:noaccess exec
C:0735:C8825000-C8825FFF					
C:0735:C8826000-C8827FFF	A:57834000--57834FFF	s	00	00001000	P:readwrite U:noaccess exec
C:0735:C8828000-C8828FFF					
C:0735:C8829000-C882BFFF	A:18000000--18000FFF	s	00	00001000	P:readwrite U:noaccess notexec

Process별로 확인하는 mmu에 대한 정보는 process가 할당받아 사용하는 4GB(0x00000000~~0xFFFFFFFF) 전체 영역 확인이 가능하다. 여기에는 공통적으로 사용하는 kernel에 대한 mmu정보 또한 포함이 되어 있다.

5-4. Kernel 디버깅 실습 2

각 process별로 mmu 정보를 확인해보면 user 영역에 대한 mmu 정보가 다른 것을 확인할 수 있다. 즉, exception 등 문제가 발생했을 경우 해당 process에 대한 mmu table을 봐야 한다. 만약 다른 mmu 정보를 열게 되면 다른 방향으로 분석이 이루어지기 때문에 문제가 될 수 있다. 아래 그림의 경우, 예를 들어 0xB000 address에서 void process는 physical에 mapping이 되어 있기 때문에 문제가 없는 address이지만 com.android.phone process의 경우에는 0xB000가 mapping되어 있지 않아 해당 process에서 이 address를 접근하면 문제가 발생한다.
즉, 각 process마다 mmu 정보가 다른 것을 확인할 수 있다.

The image shows two side-by-side windows displaying memory dump data. Both windows have a title bar with a close button and a toolbar with three icons. The left window is titled 'B:mmu.tpt.list "com.android.phone"' and the right window is titled 'B:mmu.tpt.list "void"'. Both windows have a table with columns: address, physical, sec, d, and s.

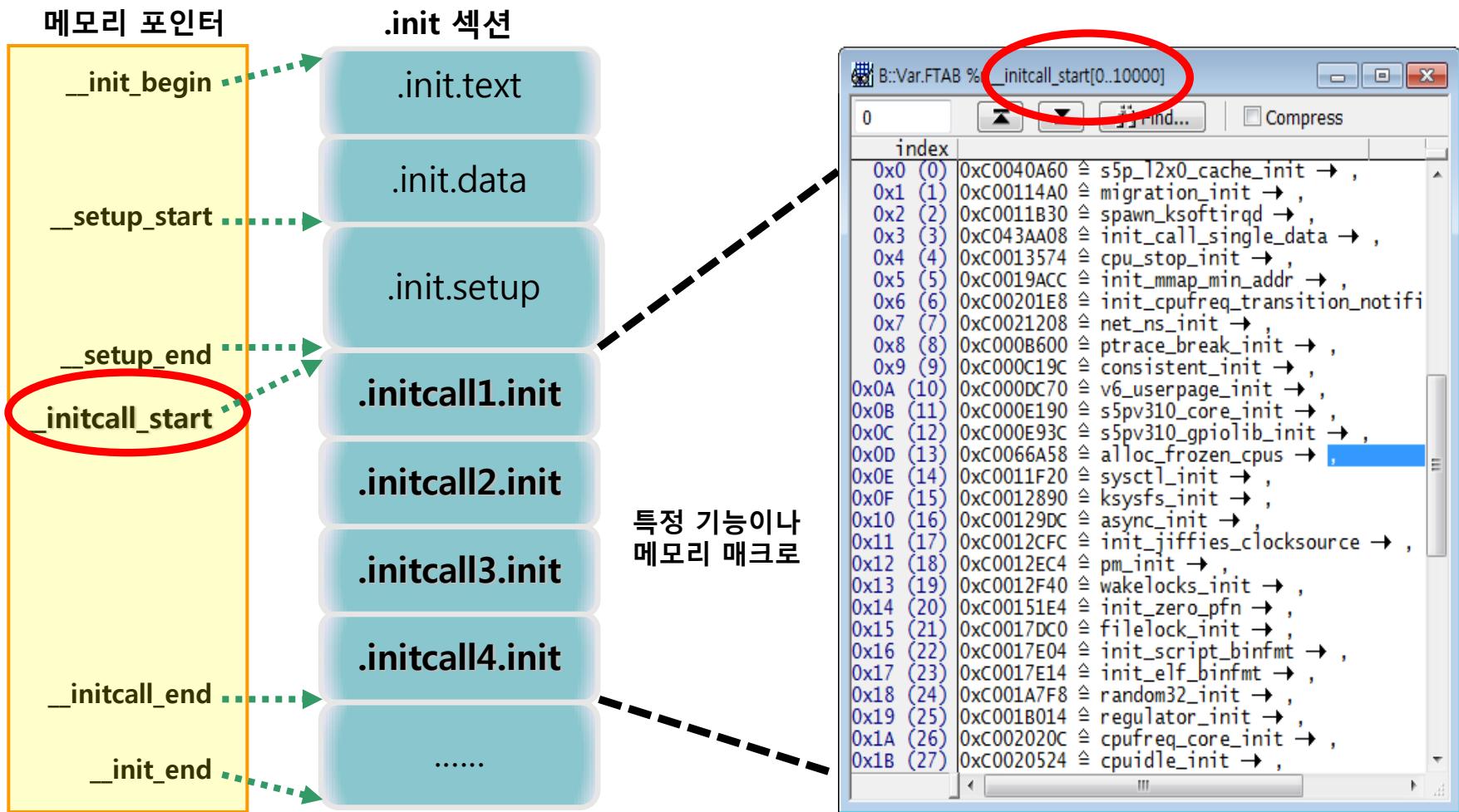
address	physical	sec	d	s
C:09F8:00000000--00007FFF				
C:09F8:00008000--00008FFF	A:42130000--42130FFF	s	01	0
C:09F8:00009000--00009FFF				
C:09F8:0000A000--0000AFFF	A:40730000--4073FFFF	s	01	0
C:09F8:0000B000--00C6CFFF				
C:09F8:00C60000--00C6DFFF	A:65460000--65460FFF	s	01	0
C:09F8:00C6E000--00C6EFFF	A:62813000--62813FFF	s	01	0
C:09F8:00C6F000--00C70FFF	A:6C548000--6C549FFF	s	01	0
C:09F8:00C71000--00C71FFF	A:65797000--65797FFF	s	01	0
C:09F8:00C72000--00C72FFF	A:655EA000--655EAFFF	s	01	0
C:09F8:00C73000--00C73FFF				
C:09F8:00C74000--00C74FFF	A:655E4000--655E4FFF	s	01	0
C:09F8:00C75000--00C75FFF	A:6537A000--6537AFFF	s	01	0
C:09F8:00C76000--00C76FFF	A:6536B000--6536BFFF	s	01	0
C:09F8:00C77000--00C77FFF	A:6535E000--6535EFFF	s	01	0
C:09F8:00C78000--00C78FFF	A:65E36000--65E36FFF	s	01	0
C:09F8:00C79000--00C79FFF	A:64D0A000--64D0AFFF	s	01	0
C:09F8:00C7A000--00C7AFFF	A:64AB4000--64AB4FFF	s	01	0
C:09F8:00C7B000--00C7BFFF	A:64AD9000--64AD9FFF	s	01	0
C:09F8:00C7C000--00C7CFFF	A:64AD4000--64AD4FFF	s	01	0
C:09F8:00C7D000--00C7DFFF	A:64AD5000--64AD5FFF	s	01	0

address	physical	sec	d	s
C:08EA:00000000 00007FFF				
C:08EA:00008000--0000BFFF	A:40735000--40738FFF	s	01	0
C:08EA:0000C000 0000CFFF				
C:08EA:0000D000--0000DFFF	A:420FD000--420FDFFF	s	01	0
C:08EA:0000E000--0000EFFF	A:420FC000--420FCFFF	s	01	0
C:08EA:0000F000--0000FFFF	A:410BD000--410BDFFF	s	01	0
C:08EA:00010000--00010FFF	A:410BC000--410BCFFF	s	01	0
C:08EA:00011000--00011FFF	A:410FD000--410FDFFF	s	01	0
C:08EA:00012000--00012FFF	A:410FC000--410FCFFF	s	01	0
C:08EA:00013000--00013FFF				
C:08EA:00014000--00014FFF	A:42138000--42138FFF	s	01	0
C:08EA:00015000--00015FFF	A:4219B000--4219BFFF	s	01	0
C:08EA:00016000--00016FFF	A:420DF000--420DFFF	s	01	0
C:08EA:00017000--00017FFF	A:4209D000--4209DFFF	s	01	0
C:08EA:00018000--00019FFF				
C:08EA:0001A000--0001AFFF	A:42193000--42193FFF	s	01	0
C:08EA:0001B000--00E18FFF				
C:08EA:00E19000--00E1AFFF	A:42212000--42213FFF	s	01	0
C:08EA:00E1B000--00E1BFFF	A:42220000--42220FFF	s	01	0
C:08EA:00E1C000--00E1CFFF	A:42222000--42222FFF	s	01	0
C:08EA:00E1D000--00E1DFFF	A:42222600--422226FFF	s	01	0

5-5. Built-in Device Driver 디버깅

Kernel build시 등록된 디바이스 드라이버들을 디버깅하는 방법을 학습합니다

Built-in device driver들은 대부분 initcall table에 등록되어 있으며, initcall table의 구조는 아래 그림과 같습니다 (<kernel>/include/linux/init.h 파일 참조)



5-5. Built-in Device Driver 디버깅

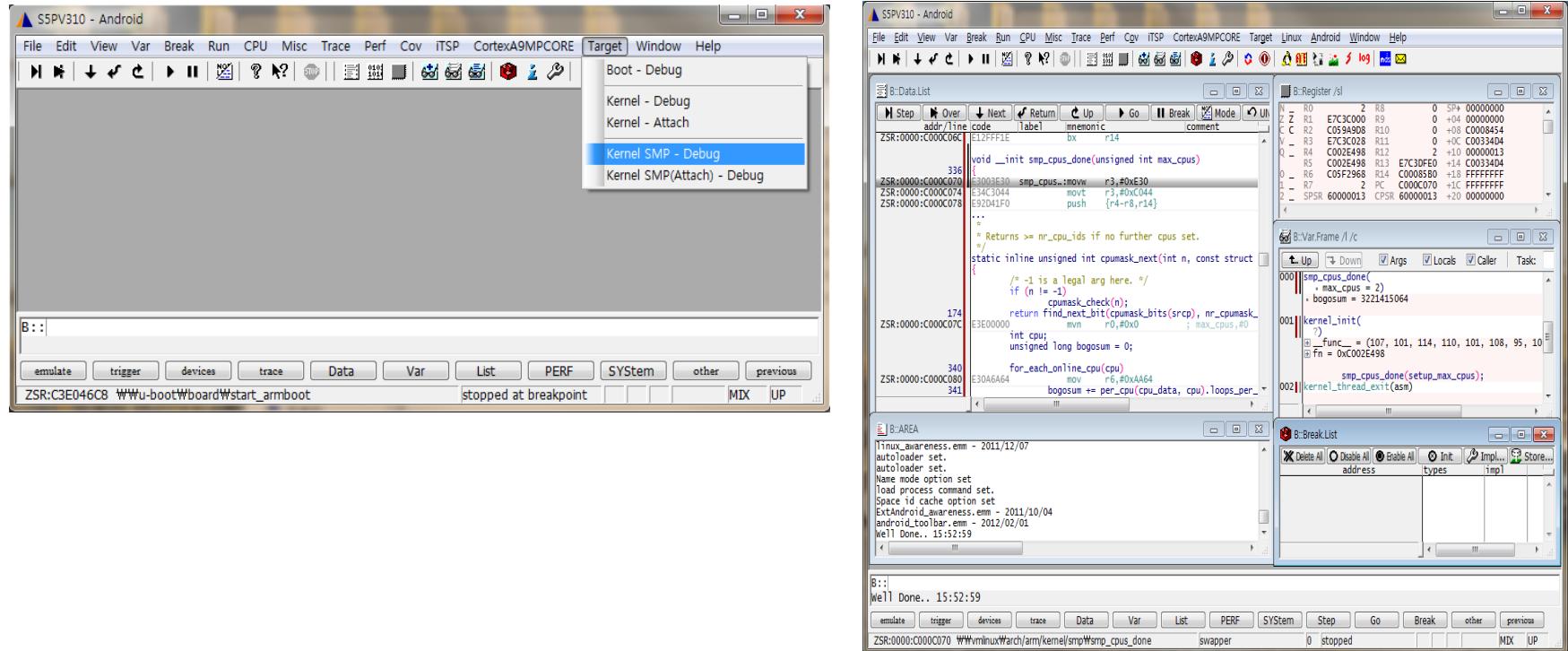
실습을 통해 built-in device driver에 대해서 이해합니다

1. Built-in device driver

1.1 Trace32 Menu에서

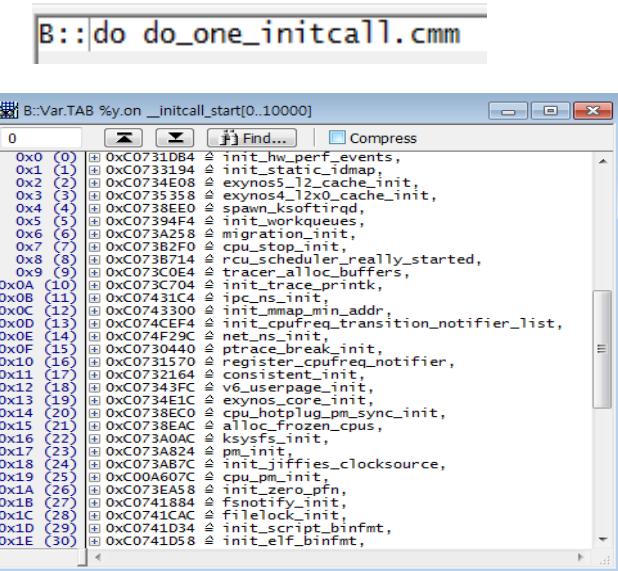
[Target] – [Kernel SMP – Debug]

선택한다.

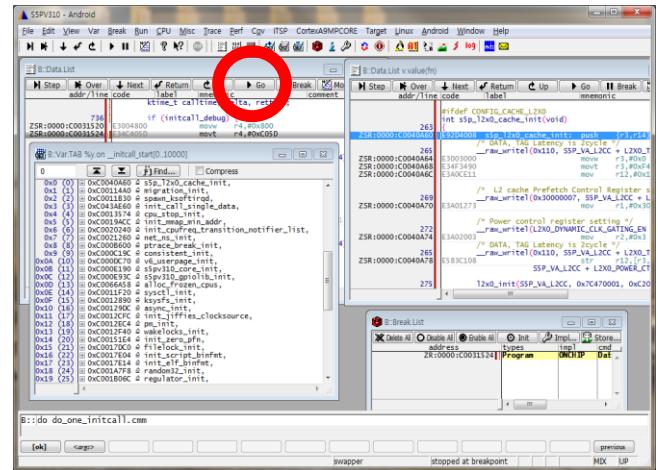


5-5. Built-in Device Driver 디버깅

1.2 Trace32 커マン드 창에 B:: do do_one_initcall.cmm을 실행 한 후 Go버튼을 눌러 타겟을 Running 시킨다.



1.3 Go 버튼을 누른 후 breakpoint에 의해 target이 멈추면 하나씩 초기화 되는 device driver를 확인해 볼 수 있다.



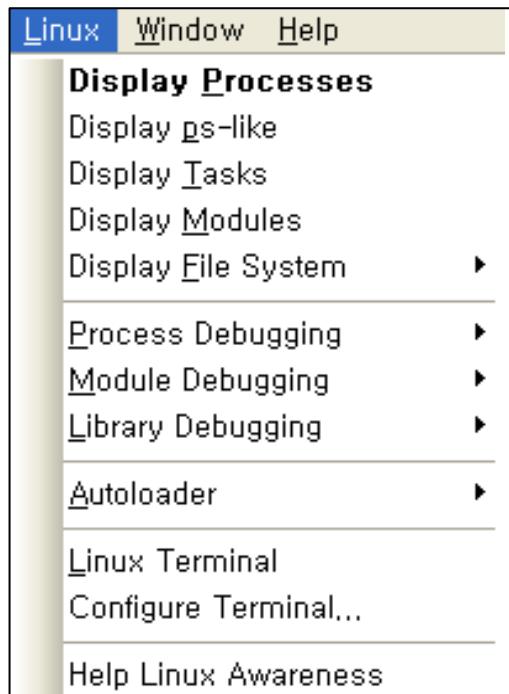
6. Linux Awareness

TRACE32의 Linux Awareness 기능에 대해서 이해하고, Autoloader를 통해 Symbol 정보를 Load하여 Source Level 디버깅을 가능하게 합니다

1. Linux Awareness
2. Autoloader 활용

6-1. Linux Awareness

Linux Awareness는 Linux kernel의 정보를 제공하는 TRACE32의 기능으로
현재 CPU의 상태 및 디버깅을 위한 Symbol Load 등을 할 수 있습니다



Linux Kernel의 특정 정보를 보여주는 기능입니다

Kernel Symbol인 vmlinux가 정상적으로 Load가 되어
있다면 Linux Menu를 통해
아래와 같은 Kernel 정보를 확인할 수 있습니다.

- **Display Processes:** 현재 실행되고 있는 Process
- **Display ps-like:** Linux Terminal에서 ps 명령 결과값
- **Display Tasks:** 실행 중인 Thread를 모두 Display
- **Display Modules:** 동적으로 (insmod) Load 된 Display
- **Display File System:** Mount 된 File System Display
- **Process Debugging:** Process (Daemon) Debugging
- **Module Debugging:** Module (Device driver) Debugging
- **Library Debugging:** Library Debug Function
- **Autoloader :** Symbol Autoload (6-2. 참고)

6-1. Linux Awareness

Display Processes / ps-like / Tasks 관련 기능을 실습해 봅니다

magic	command	#thr	state	spaceid	pids
C0597E80	swapper	66.	current	0000	0. 2. 3. 4. 5. 6. 7. 8.
E7C38000	init	-	sleeping	0001	1.
E7504840	ueventd	-	sleeping	04B1	1201.
E75BDCE0	servicemanager	-	sleeping	0811	2065.
E75BD8C0	vold	2.	disk sleep	0812	2066. 2111.
E75BD4A0	netd	3.	sleeping	0813	2067. 2099. 2100.
E75BD080	debuggerd	-	sleeping	0814	2068.
E75BCC60	app_process	-	disk sleep	0815	2069.
E75BC840	mediaserver	-	running	0816	2070.
E75BC420	dbus-daemon	-	sleeping	0817	2071.
E75BC000	installd	-	sleeping	0818	2072.
E7E2E100	keystore	-	sleeping	0819	2073.
E7D22100	bccmd	-	sleeping	081B	2075.
E7504000	hciattach	-	sleeping	081C	2076.
E75079C0	sh	-	sleeping	0828	2088.
E75075A0	adbd	4.	sleeping	0829	2089. 2094. 2095. 2096.

pid	tty	time	command	uid	%mem
2145.	-	00000000	/system/bin/vold	0.	0.2
2067.	-	00000000	/system/bin/netd	0.	0.1
2099.	-	00000001	/system/bin/netd	0.	0.1
2100.	-	00000000	/system/bin/netd	0.	0.1
2068.	-	00000000	/system/bin/debu	0.	0.0
2069.	-	000002CA	zygote	0.	6.4
2070.	-	0000007D	/system/bin/medi	101.	1.0
2149.	-	00000000	/system/bin/medi	101.	1.0
2150.	-	00000000	/system/bin/medi	101.	1.0
2151.	-	00000007	/system/bin/medi	101.	1.0
2152.	-	00000000	/system/bin/medi	101.	1.0
2071.	-	00000003	/system/bin/dbus	100.	0.1
2072.	-	00000002	/system/bin/inst	0.	0.0
2073.	-	00000003	/system/bin/keys	101.	0.0
2088.	s3c24	00000002	/system/bin/sh	0.	0.0
2089.	-	00000002	/sbin/adbd	0.	0.0
2094.	-	00000000	/sbin/adbd	0.	0.0
2140.	-	00000000	/sbin/adbd	0.	0.0
2141.	-	00000001	/sbin/adbd	0.	0.0
2153.	-	0000001E	system_server	100.	5.6
2154.	-	00000003	system_server	100.	5.6
2155.	-	00000034	system_server	100.	5.6
2156.	-	00000000	system_server	100.	5.6
2157.	-	00000000	system_server	100.	5.6

magic	command	state	uid	pid	spaceid	tty	flags	cpu
C0597E80	swapper	current(0)	0.	0.	0000	0	00200000	0.
E7C38000	init	sleeping	0.	1.	0001	0	00400100	1.
E7C38420	kthreadd	sleeping	0.	2.	0000	0	80208040	1.
E7C38840	ksoftirqd/0	sleeping	0.	3.	0000	0	84208040	0.
E7C38C60	migration/0	sleeping	0.	4.	0000	0	84208040	0.
E7C39080	migration/1	sleeping	0.	5.	0000	0	84208040	1.
E7C394A0	ksoftirqd/1	sleeping	0.	6.	0000	0	84208040	1.
E7C39CEO	events/0	sleeping	0.	7.	0000	0	84208040	0.
E7C3A100	events/1	sleeping	0.	8.	0000	0	84208040	1.
E7C3A520	khelper	sleeping	0.	9.	0000	0	80208040	0.
E7C3A940	async/mgr	sleeping	0.	13.	0000	0	80208040	1.
E7C60C60	pm	sleeping	0.	14.	0000	0	80200040	0.
E7C618C0	suspend	sleeping	0.	17.	0000	0	80208040	0.
E7E2DCE0	sync_supers	sleeping	0.	241.	0000	0	80208040	0.
E7E2E520	bdi-default	sleeping	0.	243.	0000	0	80A01040	0.
E7E2ED60	kblockd/0	sleeping	0.	245.	0000	0	84208040	0.
E7E2F180	kblockd/1	sleeping	0.	246.	0000	0	84208040	1.
E7C60420	khubd	sleeping	0.	260.	0000	0	80200040	1.

모든 Task (Process 또는 Thread)
목록을 확인하고 관련기능과
연동되는 윈도우입니다.

TCB 정보 및 라이브러리 목록 등을
확인할 수 있습니다.

6-1. Linux Awareness

특정 Process 또는 Thread의 task_struct 요약 정보를 확인할 수 있습니다

The screenshot shows the B::Task.DTask window for the process 0xE7E09080, which is the vold process. The window displays various fields and relationships related to the task_struct.

- 프로세스의 상태**
(eg. SUPERPRIV : super-user 권한)
- 가족 관계**
- Argument 정보**
- 환경 변수 정보**
- Open한 파일**
(eg. Device node file)
- Code/Data/Stack 정보**
- library 관련 정보**
(section 정보 포함)
- 시작 시간, sleep기간**

Key data points highlighted in the screenshot:

- Flags:** SUPERPRIV RANDOMIZE
- Parent:** init
- Arguments:** /system/bin/vold
- Environment:** PATH=/sbin:/vendor/bin:/system/sbin:/system/bin:/system/xbin
LD_LIBRARY_PATH=/vendor/lib:/system/lib
- Open files:** null, main, radio, events, system
- Properties:** properties
- Code/addr/size:** 00008000 / 0000B514
- Data/addr/size:** 00014000 / 00000518
- Stack/start:** BE863CC0
- Code file:** vold, vold, properties, libhdmdiservice.so, libhdmdiservice.so, libdiskconfig.so, libdiskconfig.so
- Start address:** 00008000, 00014000, 40000000, 80000000, 80001000, A7600000, A7603000
- Flags:** EX RD, WR RD, RD, EX RD, WR RD, EX RD, WR RD
- Start time:** 3860384D
- Sleep time:** 000001FD
- User time:** 0000000D
- System time:** 00000005

6-1. Linux Awareness

Display Modules / File System 정보를 확인하고 이해합니다

B:TASK.MODULE						
magic	name	state	size	address	refcount	depends
BF062968	ar6000	Live	494746.	BF000000		

커널에 로드 되어 있는 파일 시스템
또는 모듈의 목록을 확인할 수
있습니다

lsmod 같은 역할

B:TASK.FS.Types		
magic	name	#devs
C05ABD48	sysfs	2.
C05AC1B8	rootfs	1.
C05A884C	bdev	1.
C05ABA8	proc	1.
C05A9D90	cgroup	2.
C05AABA8	tmpfs	5.
C05C9B80	sockfs	1.
C05BC9E0	usbfs	1.
C05AB5F8	pipefs	1.
C05AB97C	anon_inode	1.
C05CDE00	rpc_pipef	1.
C05ABDB0	devpts	1.
C05ABDE8	ext3	0.
C05ABE1C	ext2	0.
C05ABE40	ext4	2.
C05AC048	cramfs	0.
C05AC080	ramfs	0.
C05AC22C	vfat	1.
C05AC24C	msdos	0.
C05AC324	nfs	0.
C05AD3E0	romfs	0.
C05AD490	yaffs	0.
C05AD4B0	yaffs2	0.
C05BBFB0	mtd_inode	1.

커널에 등록된 FS

B:TASK.FS.Mount				
magic	device	mountpoint	type	mod
E7C03280	rootfs	/	rootfs	ro
E7FC4780	tmpfs	/dev	tmpfs	rw
E7FC4800	devpts	/pts	devpts	rw
E7FC4880	proc	/proc	proc	rw
E7FC4900	sysfs	/sys	sysfs	rw
E7FC4980	none	/acct	cgroup	rw
E7FC4800	tmpfs	/mnt/asec	tmpfs	rw
E7FC4880	tmpfs	/mnt/obb	tmpfs	rw
E7FC4C00	none	/cpuctl	cgroup	rw
E7FC4E80	/dev/block/mm	/system	ext4	ro
E751AA80	/dev/block/mm	/data	ext4	rw
D37FD400	/dev/block/vo	/mnt/sdcard	vfat	rw
D37FD500	/dev/block/vo	/mnt/secure/	vfat	rw
D37FD580	tmpfs	/.android_se	tmpfs	ro

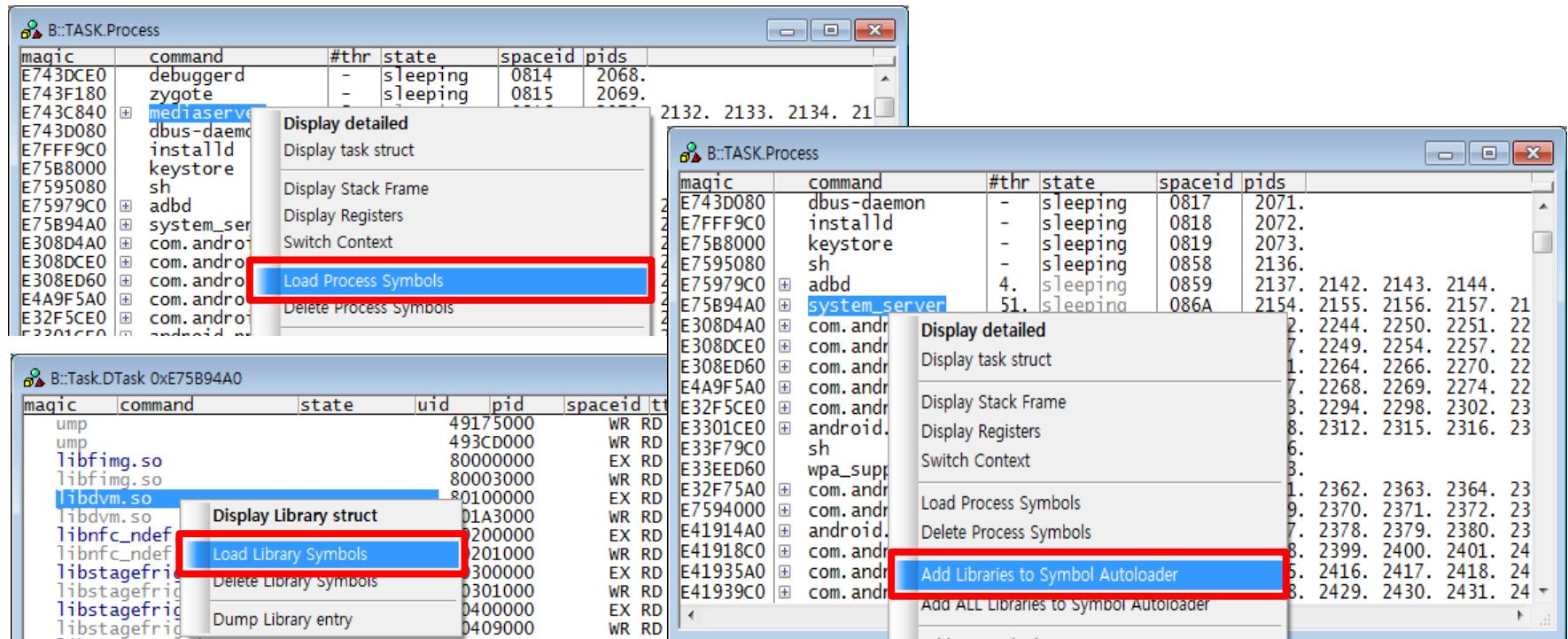
Mount되어 있는 위치

B:TASK.FS.MountDevs				
magic	dev#	fsmagic	type	root
E7C04200	0.	beer	sysfs	/
E7C04400	1.	858458F6	rootfs	bdev:
E7C04600	2.	bdev	bdev	/
E7C04800	3.	00009FA0	proc	/
E7C5C200	4.	01021994	tmpfs	/
E7C5C600	5.	SOCK	sockfs	socket:
E7CFE600	6.	PIPE	pipefs	pipe:
E7CFE400	7.	09041934	anon_ino	anon_inode
E7CD5200	8.	00001CD1	devpts	/
E7CD4800	9.	gYi	rpc_pipe	/
E7F48C00	10.	11307854	mtd_inod	mtd_inode:
E7475C00	11.	00009FA2	usbfs	/
E7457C00	12.	01021994	tmpfs	/
E7510000	13.	beer	sysfs	/
E75A6200	14.	0027E0EB	cgroup	/
E75A6400	15.	01021994	tmpfs	/
E75A6E00	16.	01021994	tmpfs	/
E75A7400	17.	0027E0EB	cgroup	/
E75A7C00	1876.	0000EF53	ext4	/
E75A6000	1876.	0000EF53	ext4	/
E7605400	1876.	00004D44	vfat	/
E7605C00	18.	01021994	tmpfs	/

Mount되어 있는 디바이스

6-2. Autoloder 활용

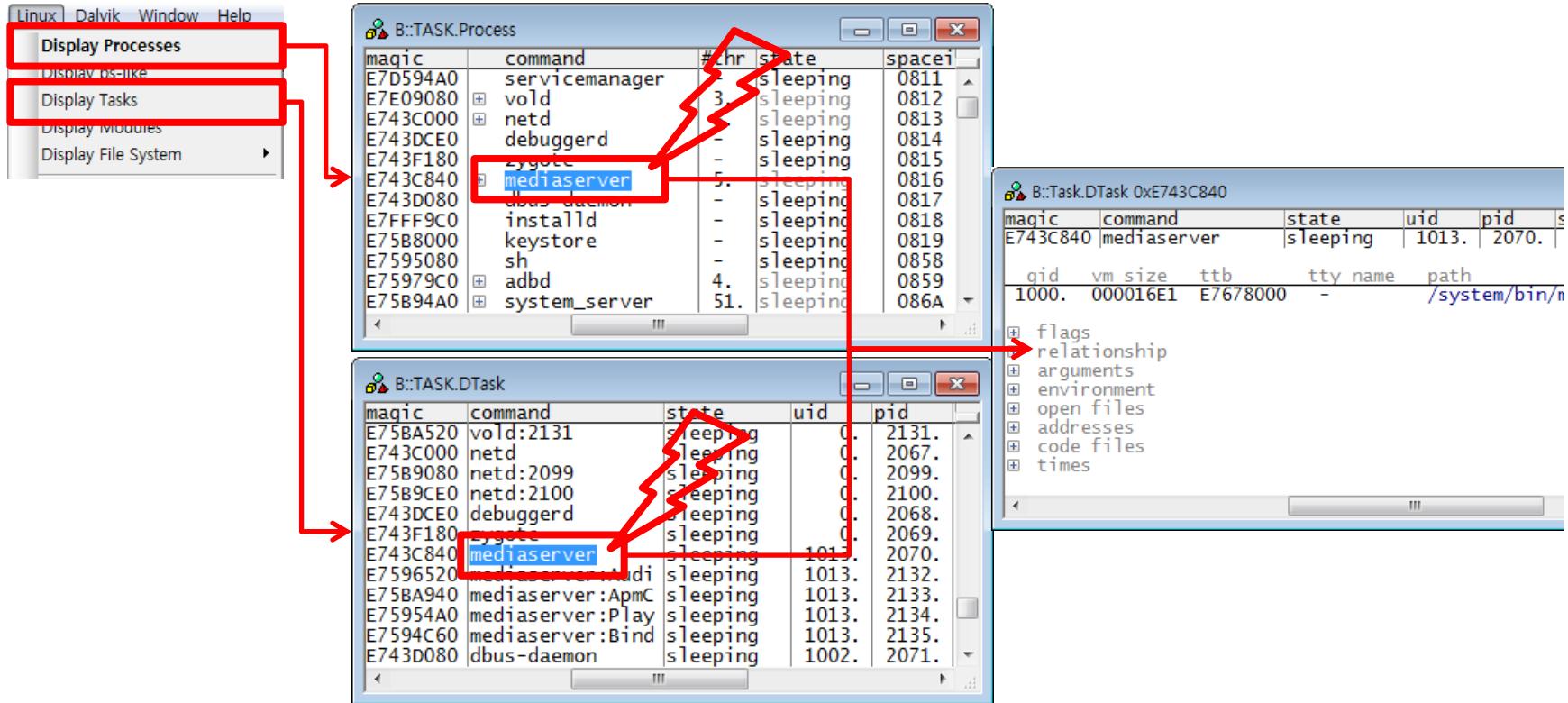
Autoloader는 Kernel Symbol인 VMLinux를 제외한 Process, Module, Library 대한 Symbol 정보를 자동으로 Load하는 기능입니다



이 기능은 Process의 목록에서 특정 디버깅 심볼 정보가 필요할 때 직관적으로 사용할 수 있다는 장점이 있고 또한 다음 장의 Autoloader 메뉴의 List Components 기능과도 관련이 있습니다

6-2. Autoloder 활용

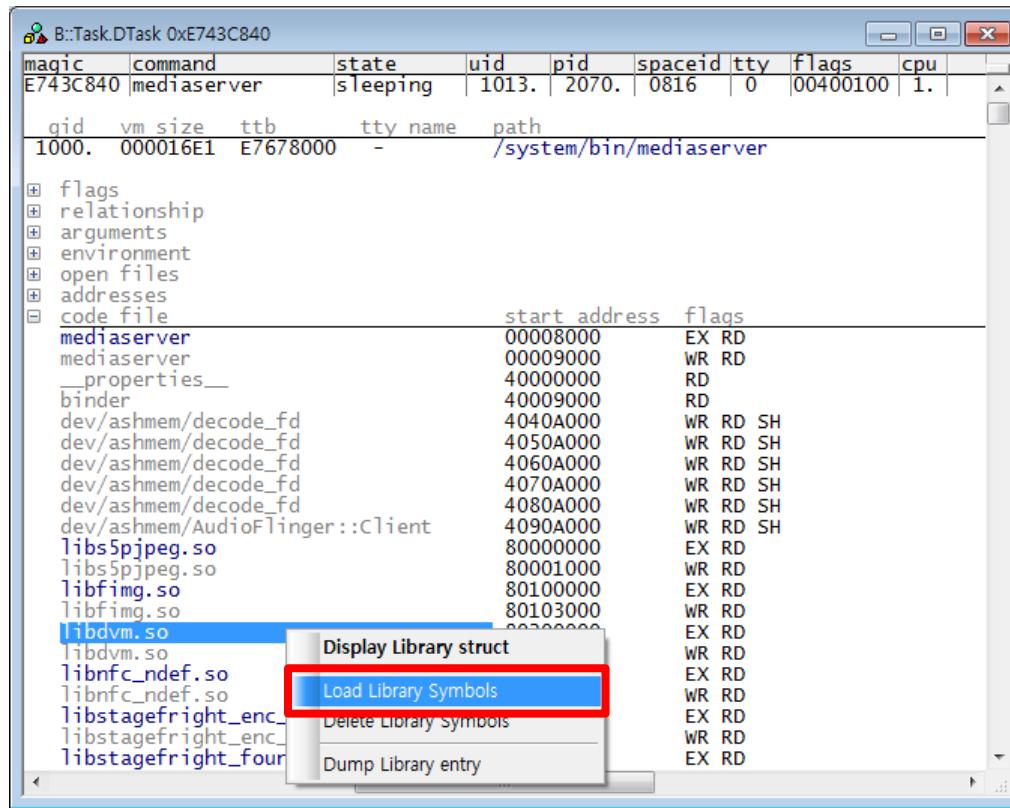
TASK.Process 또는 TASK.DTask 를 통해 Symbol load 하는 방법을 익힙니다



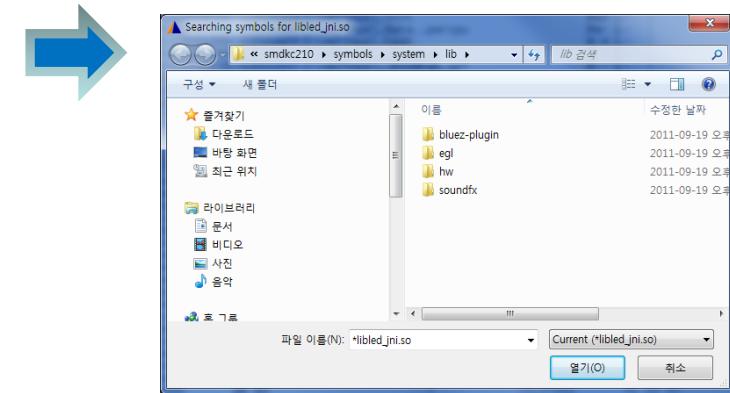
디버깅을 원하는 라이브러리를 사용하는 프로세스 이름 또는 Magic number(address of task_struct)를 더블 클릭하여 특정 프로세스의 자세한 정보를 확인하도록 합니다

6-2. Autoloder 활용

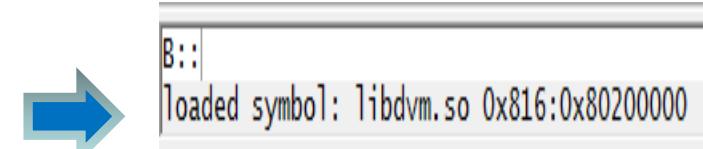
TASK.Process 또는 TASK.DTask 를 통해 Symbol load 하는 방법을 익힙니다



파일이 존재하지 않을 경우



파일이 존재 할 경우



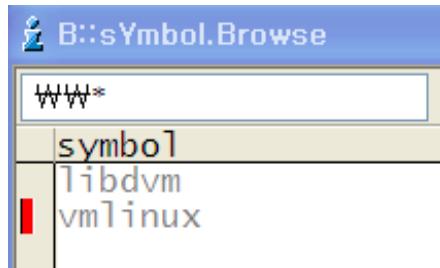
특정 프로세스의 자세한 태스크 정보 중에서 *code files*부분을 확장시켜서 확인하고 디버깅을 원하는 라이브러리를 선택하여 마우스 오른쪽 버튼을 누르고 Load Library symbols를 선택하여 심볼 이미지를 선택하고 로드 합니다

6-2. Autoloder 활용

TASK.Process 또는 TASK.DTask 를 통해 Symbol load 하는 방법을 익힙니다

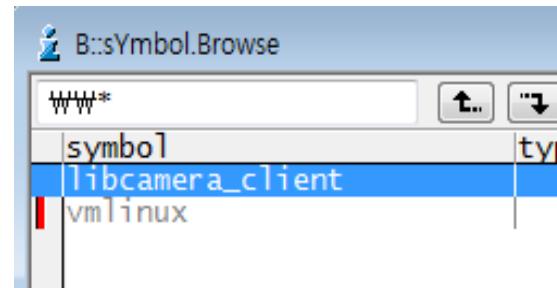
Zygote process의 libdvm.so를 load하는 경우

```
B:::task.symbol.loadlib "zygote" "libdvm.so"
```



Mediaserver의 libcamera_client.so를 load하는 경우

```
B:::task.symbol.loadlib "mediaserver" "libcamera_client.so"
```

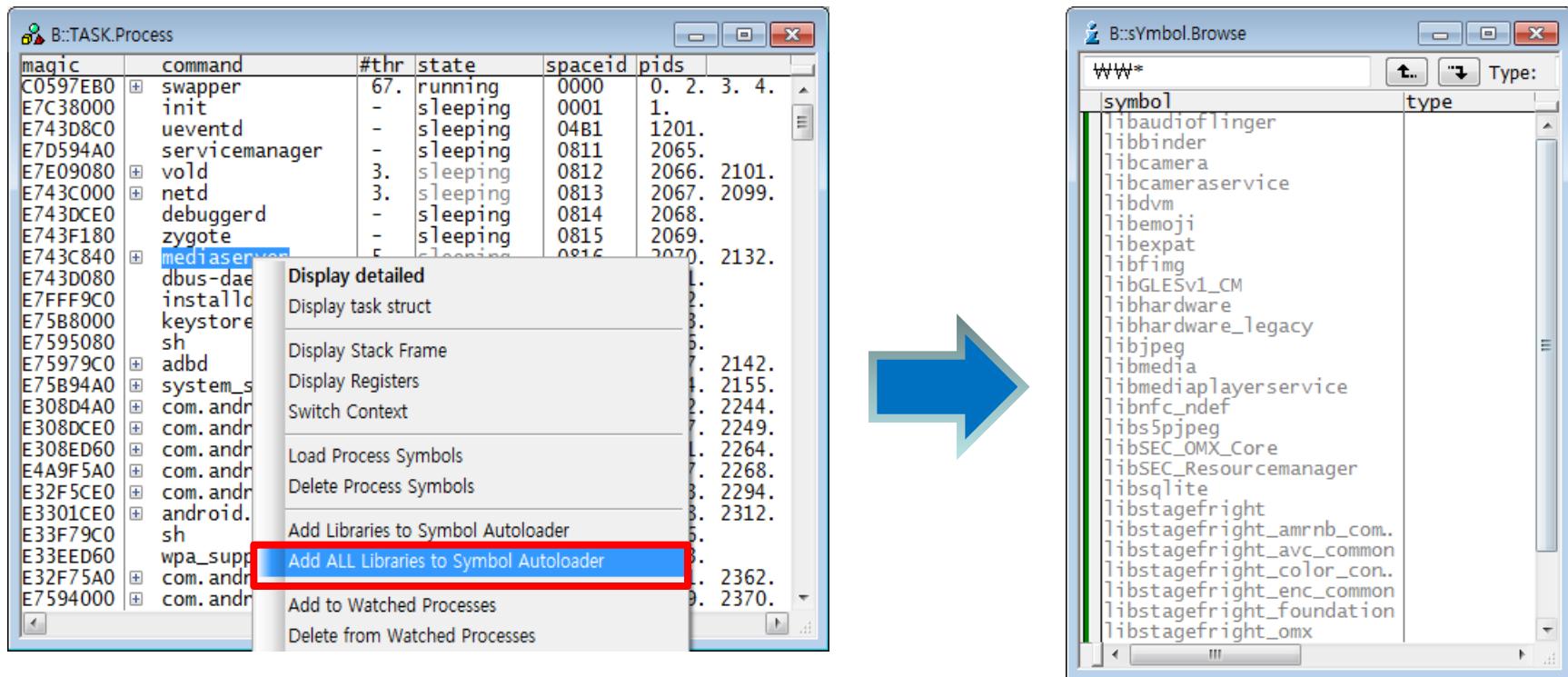


추가적으로 명령어를 통한 Library Symbol Loading 하는 방법이 있다.

Display Processes window는 data를 계속 access하는 window로 open이 되면 느려지는 현상이 발생 할 수 있다. 이 때 symbol을 load하면 느리다는 느낌이 들기 때문에 명령어를 통해 빠른 symbol loading을 할 수 있다.

6-2. Autoloder 활용

TASK.Process 또는 TASK.DTask 를 통해 Symbol load 하는 방법을 익힙니다

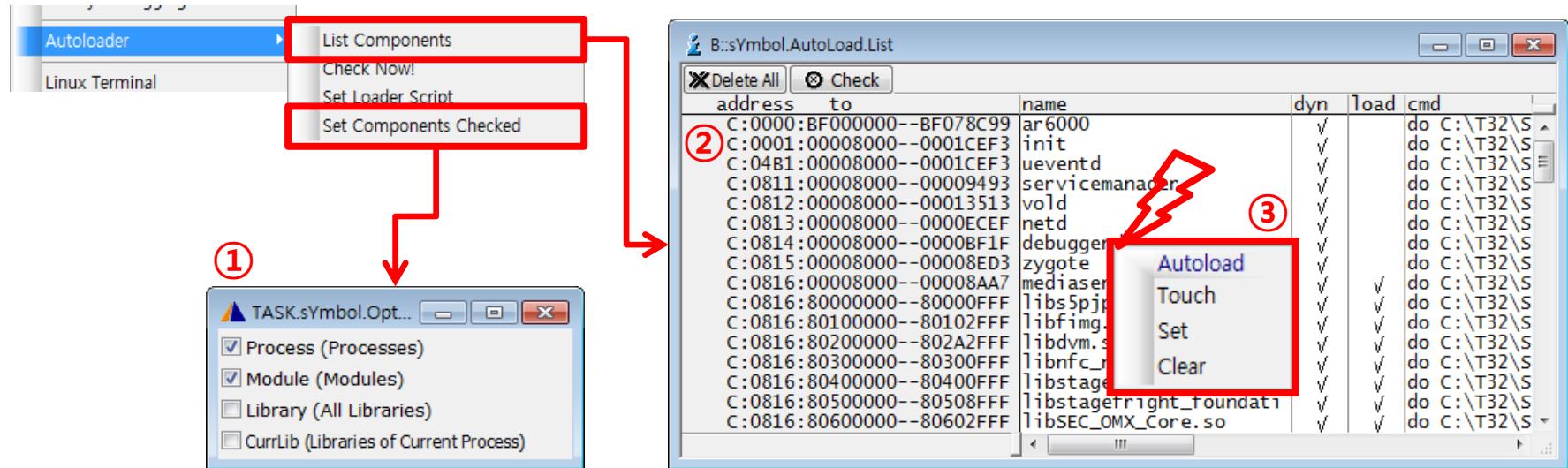


해당 프로세스의 모든 라이브러리 목록의 디버깅 심볼 정보를 로드하고 싶다면
위의 그림처럼 Add ALL Libraries to Symbol Autoloader 버튼을 클릭합니다
(위의 기능은 많은 라이브러리 파일을 로드하므로 시간이 오래 걸립니다)

로딩한 심볼은 sYmbol.Browse 창에서 확인합니다

6-2. Autoloder 활용

List Components (sYmbol.AutoLoad.List) 와 Set Components Checked (TASK.sYmbol.Option AutoLoad) 기능을 통한 Symbol load 방법을 익힙니다



디버깅하길 원하는 구성 요소(Component)를 선택하여 선택 디버깅하는 기능

- Check : 수동적으로 목록의 새로 고침을 수행합니다.
- Touch : 수동적으로 심볼 정보 파일을 로드 하도록 합니다.
- Set : 특정 심볼 정보가 이미 로드 된 것처럼 마크합니다
(심볼 파일이 없을 경우 유용합니다.)
- Clear : Load로 마크된 특정 심볼 정보를 언-마크합니다.

7. Process(Daemon)

Android에서의 Daemon(Process) 디버깅을 위해 Daemon(Process)을 이해합니다

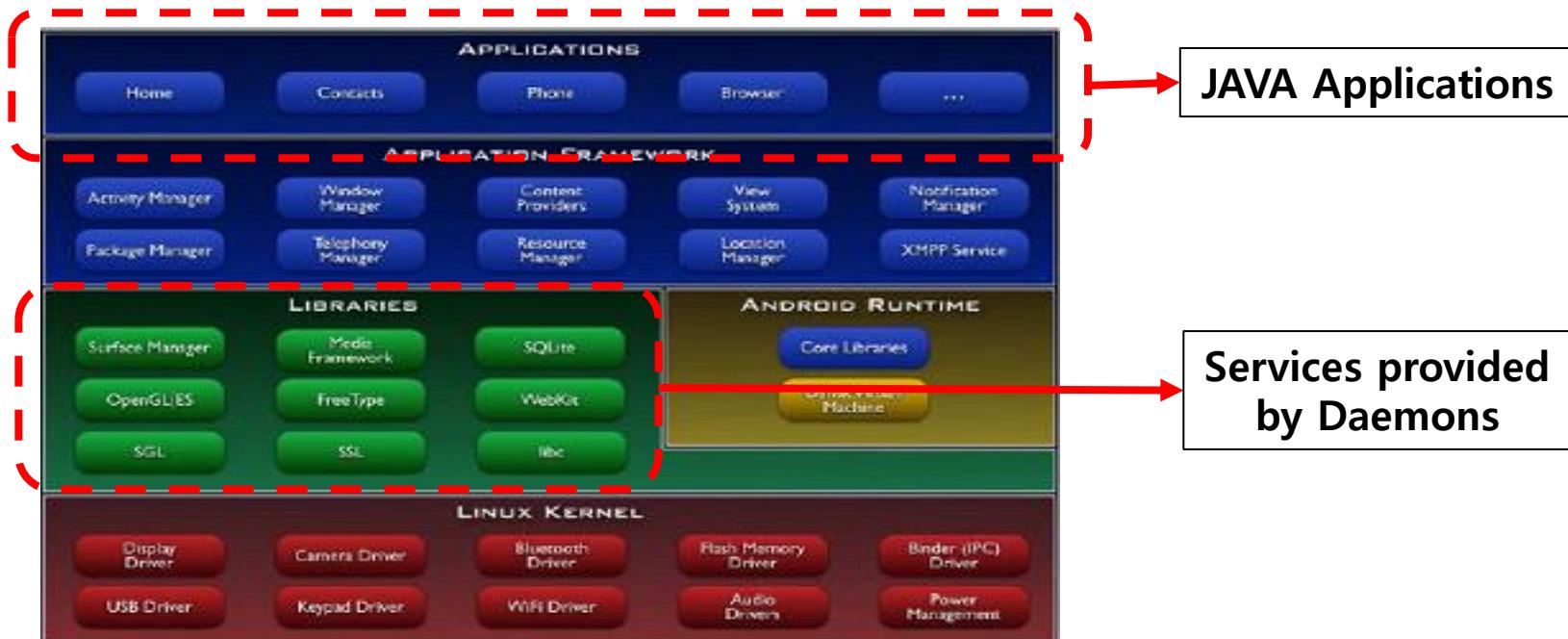
- 1. Java Application & Daemon**
- 2. Daemon 디버깅**
- 3. Daemon의 main 함수부터 디버깅 실습**
- 4. Daemon 실행 후 Daemon 디버깅 실습**

7-1. JAVA application & Daemon

User Application인 Java Application과 Native Process인 Daemon으로 구분

Android에서는 두 종류의 Process(application & daemon)가 존재합니다

- 하나는 JAVA로 되어 있는 apk인 Application이고,
다른 하나는 부팅 중에 init에 의해 실행되는 Daemon (rild, vold, etc)으로 구분됨
- JAVA application은 부팅 후 사용자에 의해 실행되는 User Process이고
Daemon은 보통 Native Library Load를 위한 Native Process입니다



7-2. Daemon Debugging

Android에서 Process Debugging은 Java Application이 아닌 Daemon 디버깅
방식은 일반적인 Linux Application Debugging 방식과 동일합니다

Parent Process	Child Process
bootloader	Linux Kernel
Kernel	init
init	servicemanager usbd mountd adb debuggerd rild mediaserver installd qemuud dbus-daemon
init	zygote
zygote	system_server
system_server	
start-up time launched android applications	com.android.phone android.process.shared com.google.process.shared com.android.mms com.android.alarmclock android.process.media

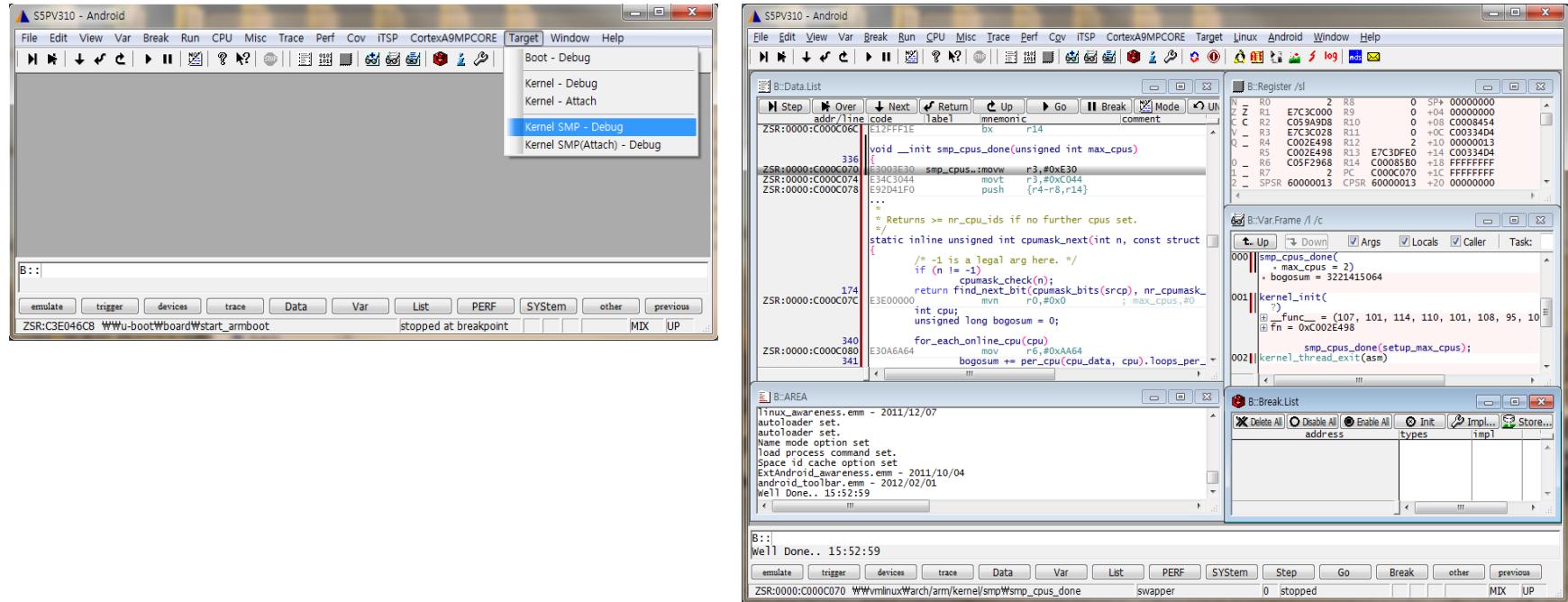
7-3. Daemon의 main 함수부터 디버깅 실습

실습을 통해 daemon의 main 부터 디버깅 방법을 이해합니다

1. Process의 Main부터 디버깅하기

1.1 Init process Main부터 디버깅하기

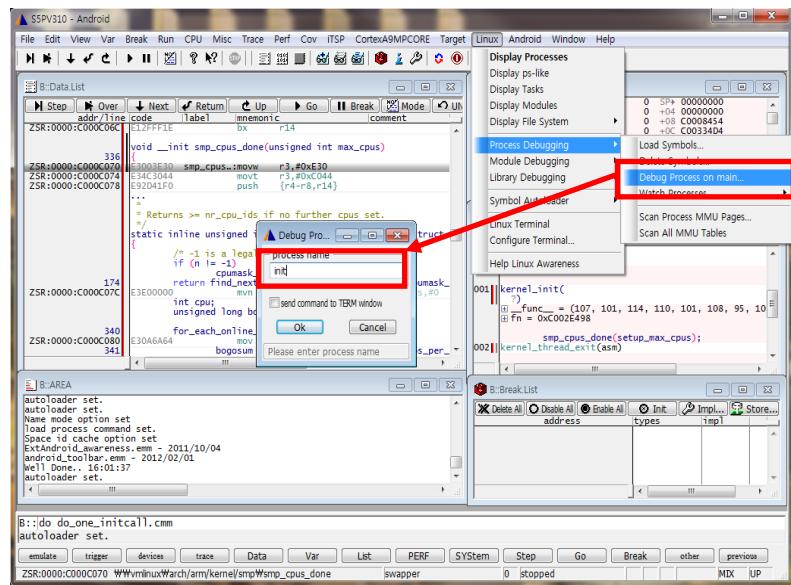
1.1.1. Menu에서 [Target]-[Kernel SMP – Debug] 선택하여 SMP debugging 환경을 만든다.



7-3. Daemon의 main 함수부터 디버깅 실습

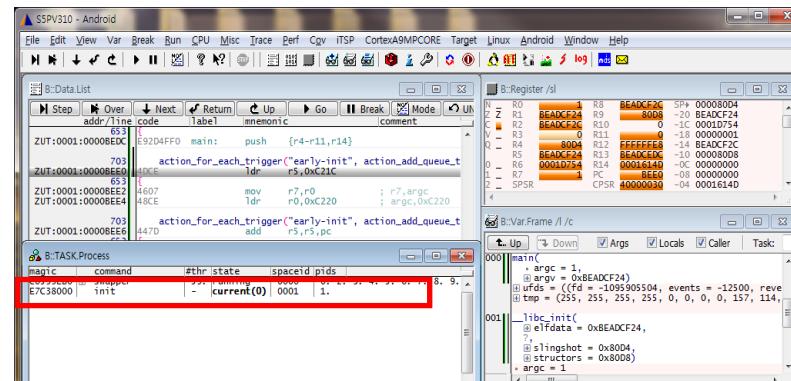
1.1.2. Trace32 Menu에서

[Linux]–[Process Debugging]–
[Debug Process on main]을
선택한다.



1.1.3 Process Name에 원하는 init을

입력 후 OK를 누르면 init 프로세스의
main()부터 디버깅 가능하다.



7-3. Daemon의 main 함수부터 디버깅 실습

1.2 Vold Daemon Main부터 디버깅하기
init process main 디버깅하기와
같은 방식으로 vold daemon을 main부터
디버깅 할 수 있다.

The screenshot shows a debugger interface with two windows. The top window is titled [B::Data.List] and displays assembly code for the main function of the vold daemon. The bottom window is titled B::TASK.Process and shows a list of running processes. The 'vold' process is highlighted with a red box in the task list, indicating it is the current target.

magic	command	#thr	state	spaceid	pids
EC973400	sh	-	sleeping	08E8	2280.
EC971000	comsicmanager	-	sleeping	08E9	2281.
EC970800	vold	-	current(0)	08EA	2282.
EC970400	netd	-	running	08EB	2283.
EC970000	debuggerd	-	sleeping	08EC	2284.

1.3 Media Server Daemon Main부터 디버깅하기
위와 같은 방식으로
MediaServer Daemon의 Main부터
디버깅 할 수 있다.

The screenshot shows a debugger interface with two windows. The top window is titled [B::Data.List] and displays assembly code for the main function of the mediaserver. The bottom window is titled B::TASK.Process and shows a list of running processes. The 'mediaserver' process is highlighted with a red box in the task list, indicating it is the current target.

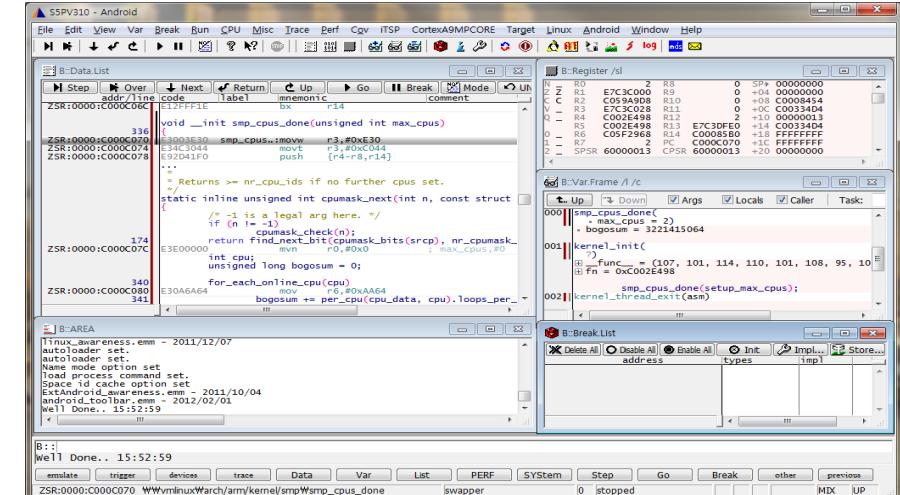
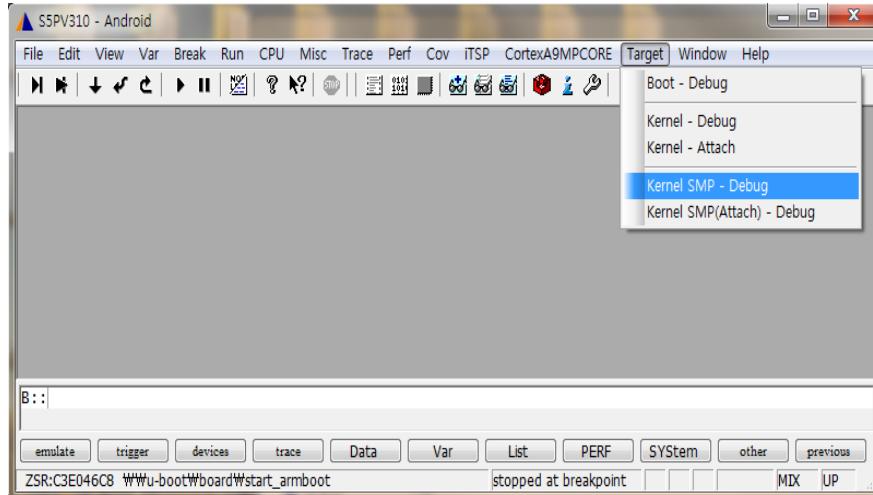
magic	command	#thr	state	spaceid	pids
C78D4960	debuggerd	-	sleeping	0731	1841.
C78D5C20	rild	3.	sleeping	0732	1842. 1873. 1878.
C78D55E0	zygote	-	running	0733	1843.
C78F15E0	mediaserver	-	current	0734	1844.
C78F1900	dbus-daemon	-	sleeping	0735	1845.

7-4. Daemon 실행 중 디버깅 실습

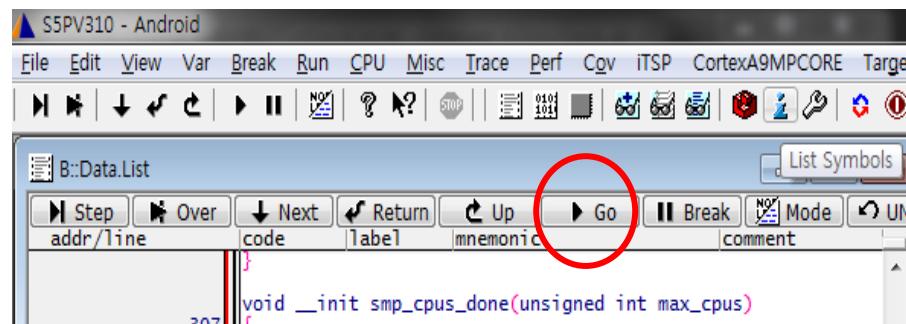
실습을 통해 Daemon 실행 이후의 디버깅 방법을 이해합니다

1. adb daemon 디버깅하기

1.1 [Target]-[Kernel SMP – Debug] 선택하여 SMP debugging 환경을 만든다.



1.2 adb daemon이 실행이 될 때까지 target을 running 시킨다.



7-4. Daemon 실행 중 디버깅 실습

1.3 부팅이 완료되면 break를 눌러 target을 멈춘 후에 linux 메뉴의 Display Processes 메뉴를 선택한다.



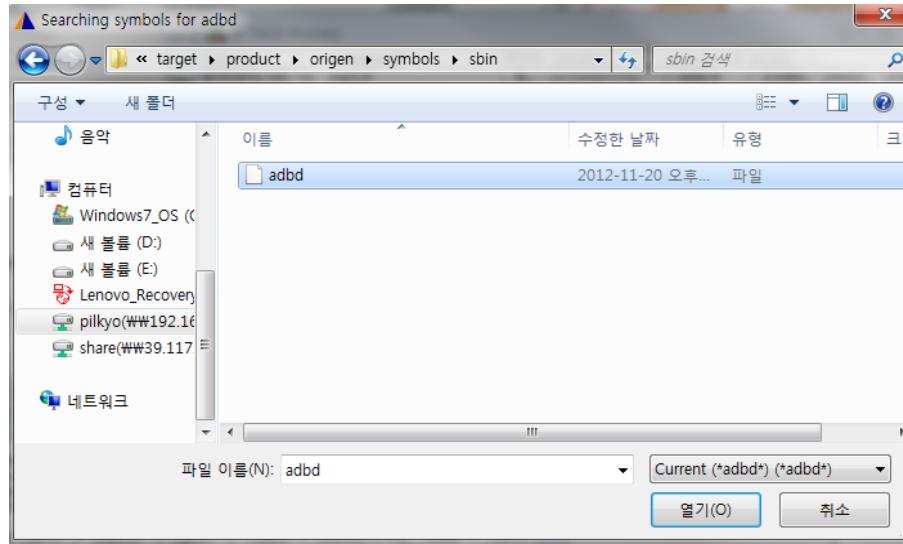
1.4 아래와 같이 현재 실행되고 있는 process 정보 중 debugging이 필요한 adbd daemon symbol을 로드한다.

B::TASK.Process							
magic	command	#thr	state	spaceid		pids	
C049F960	+ swapper	20.	current	0000			
C7818000	+ init	-	sleeping	0001			
C79195E0	+ sh	-	sleeping	0730			
C7918000	+ servicemanager	-	sleeping	0731			
C7919C20	+ vold	3.	sleeping	0732			
C7918FA0	+ netd	3.	sleeping	0733			
C7918960	+ debuggerd	-	sleeping	0734			
C79192C0	+ rild	3.	sleeping	0735			
C7919900	+ zygote	-	sleeping	0736			
C7918640	+ mediaserver	7.	sleeping	0737			
C7848960	+ dbus-daemon	-	sleeping	0738			
C7848C80	+ installd	-	sleeping	0739	1849.		
C78492C0	+ keystore	-	sleeping	073A	1850.		
C7849900	+ adbd	4.	sleeping	073B	1851. 1854. 1855.	1856.	
C78192C0	+ system_server	58.	sleeping	0764	1892. 1893. 1894.	1895. 18	
C486A640	+ putmethod.latin	9.	sleeping	07A4	1956. 1957. 1958.	1961. 19	

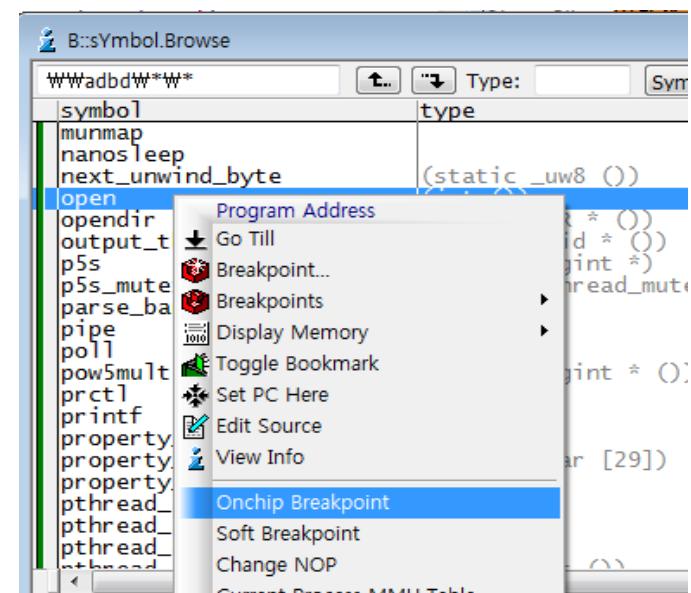
Load Process Symbols

7-4. Daemon 실행 중 디버깅 실습

1.5 adb는 기본적으로 daemon symbol이 존재하는 `\$symbols\$\$system\$\$bin` 아래에 존재하는 것이 아닌 `\$symbols\$\$sbin` 아래에 존재한다.
Dialog가 열리면 해당 경로에 접근해 adbd symbol을 load한다.

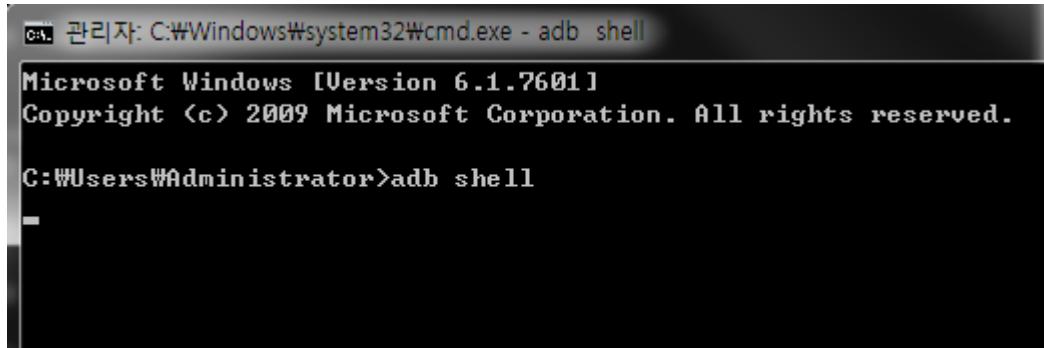


1.6 Symbol Load 후 해당 Symbol에서
open()함수를 찾아서 BreakPoint를 설정하자.
open()함수는 adb shell을 통해 진입하게 되면
동작하는 함수이다.



7-4. Daemon 실행 중 디버깅 실습

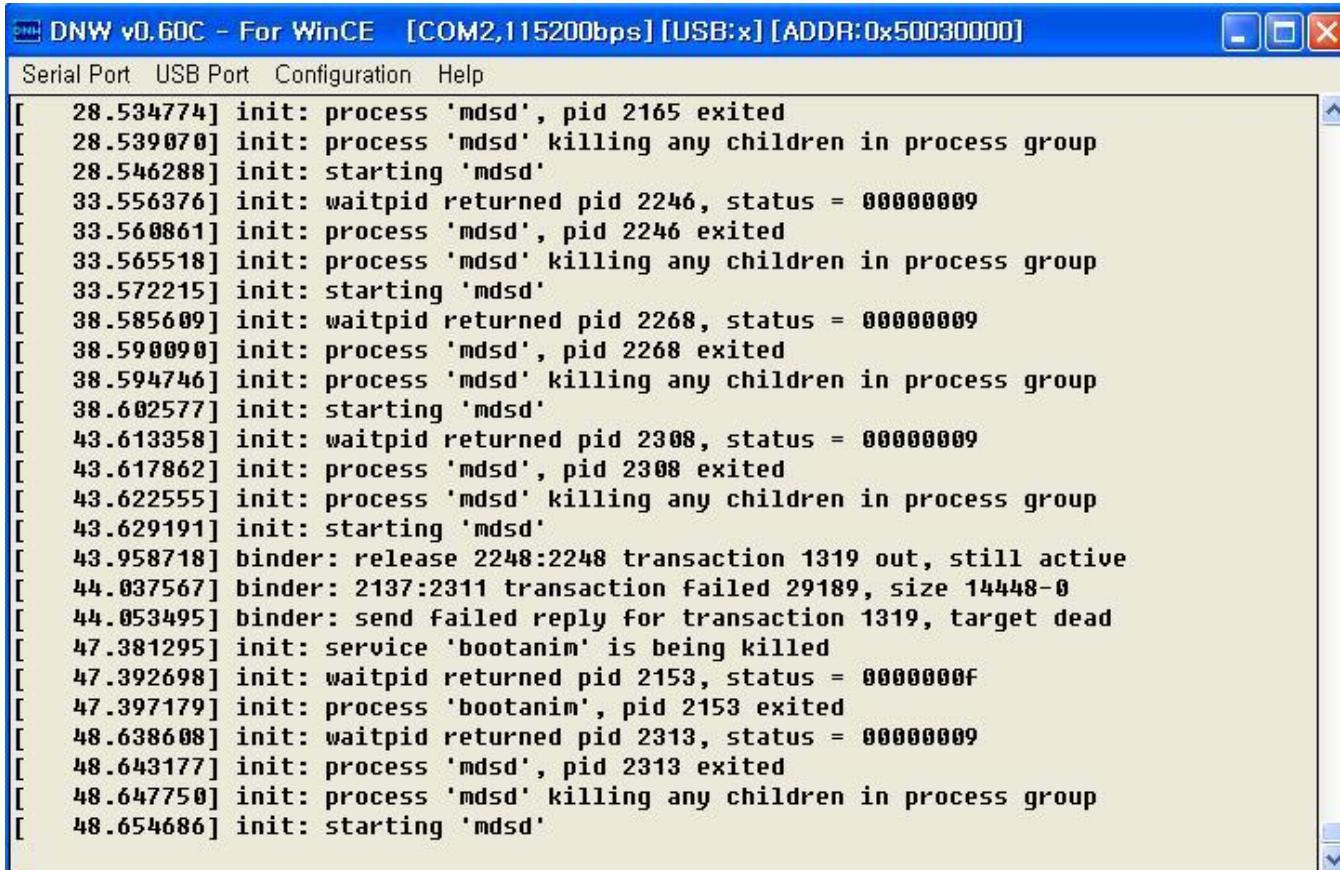
1.7 Target 동작을 시킨 후 adb shell 명령을 입력해서 해당 부분을 디버깅 한다.

A screenshot of a debugger interface showing assembly code for the 'open' function. The code is color-coded with syntax highlighting. The assembly instructions are in blue, and the C language source code is in pink. The code snippet includes lines 35 through 51, which handle file opening flags and mode setting. The debugger toolbar at the top includes buttons for Step, Over, Next, Return, Up, Go, Break, and a search bar.A screenshot of a debugger interface showing a list of processes. The table has columns for magic, command, #thr, state, and S. One row for the process 'adb' is highlighted with a red border. The 'adb' row contains the value 'EC40F000' in the 'magic' column, 'adb' in the 'command' column, '4.' in the '#thr' column, and 'current' in the 'state' column. Other processes listed include dbus-daemon, installld, keystore, system_server, com.android.syst, android.process, and ora.linaro.walln.

7-5. MDS daemon Debugging

실습을 위해 만든 MDS daemon을 디버깅하도록 하도록 합니다

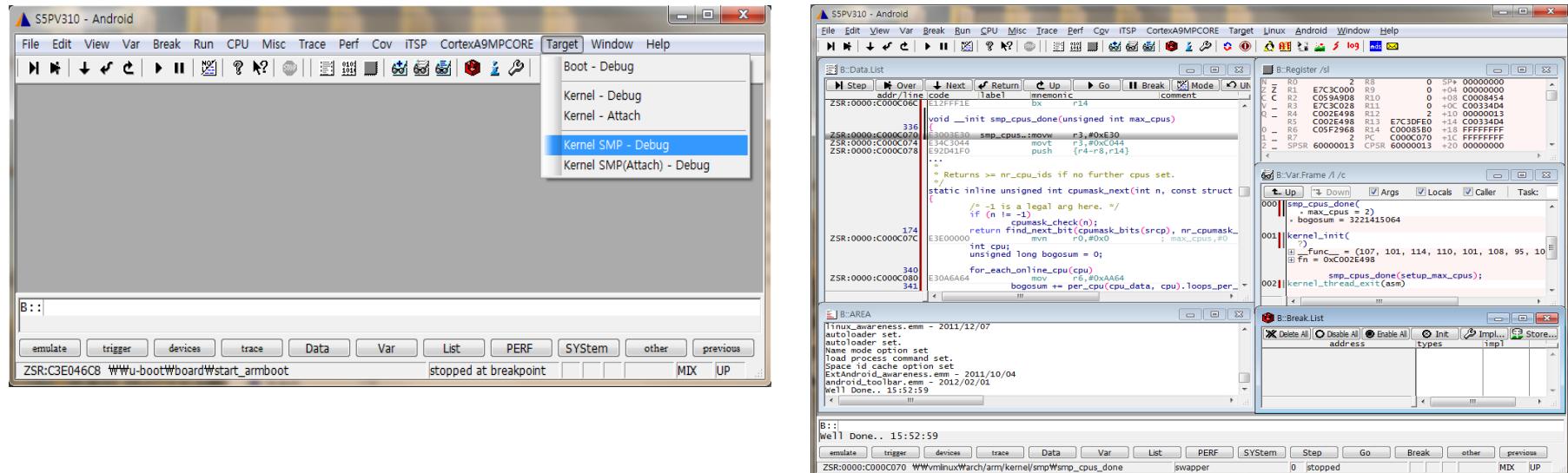
아래 log를 보면 mdssd 가 계속적으로 kill과 start를 반복적으로 하고 있다.
Daemon의 경우 문제없이 동작을 해야 하지만 특정 문제가 발생해서
아래와 같이 동작하는 것을 알 수 있다.



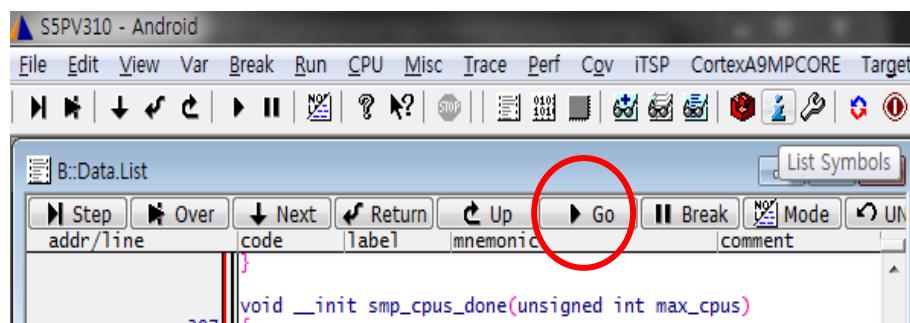
```
[ 28.534774] init: process 'mdssd', pid 2165 exited
[ 28.539070] init: process 'mdssd' killing any children in process group
[ 28.546288] init: starting 'mdssd'
[ 33.556376] init: waitpid returned pid 2246, status = 00000009
[ 33.560861] init: process 'mdssd', pid 2246 exited
[ 33.565518] init: process 'mdssd' killing any children in process group
[ 33.572215] init: starting 'mdssd'
[ 38.585609] init: waitpid returned pid 2268, status = 00000009
[ 38.590090] init: process 'mdssd', pid 2268 exited
[ 38.594746] init: process 'mdssd' killing any children in process group
[ 38.602577] init: starting 'mdssd'
[ 43.613358] init: waitpid returned pid 2308, status = 00000009
[ 43.617862] init: process 'mdssd', pid 2308 exited
[ 43.622555] init: process 'mdssd' killing any children in process group
[ 43.629191] init: starting 'mdssd'
[ 43.958718] binder: release 2248:2248 transaction 1319 out, still active
[ 44.037567] binder: 2137:2311 transaction failed 29189, size 14448-0
[ 44.053495] binder: send failed reply for transaction 1319, target dead
[ 47.381295] init: service 'bootanim' is being killed
[ 47.392698] init: waitpid returned pid 2153, status = 0000000F
[ 47.397179] init: process 'bootanim', pid 2153 exited
[ 48.638608] init: waitpid returned pid 2313, status = 00000009
[ 48.643177] init: process 'mdssd', pid 2313 exited
[ 48.647750] init: process 'mdssd' killing any children in process group
[ 48.654686] init: starting 'mdssd'
```

7-5. MDS daemon Debugging

1.1.1. Menu에서 [Target]-[Kernel SMP – Debug] 선택하여 SMP debugging 환경을 만든다.

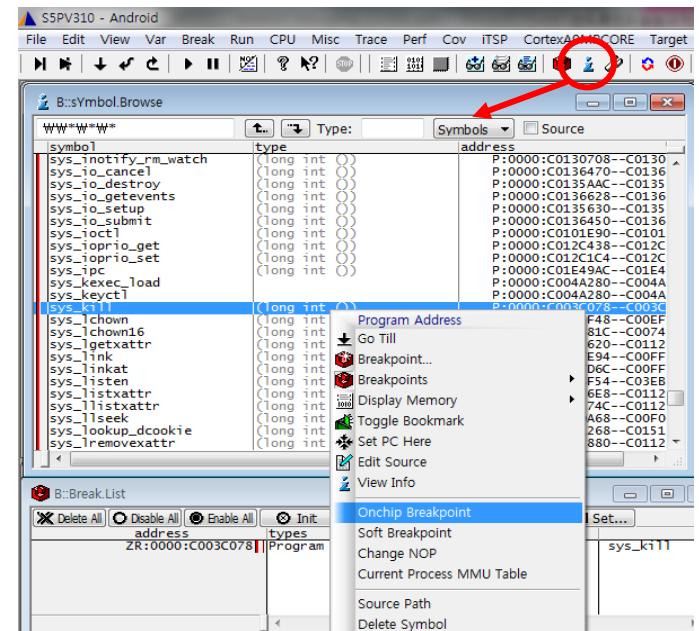


1.1.2 target 을 running 시켜 mdsd daemon이 실행하도록 한다.

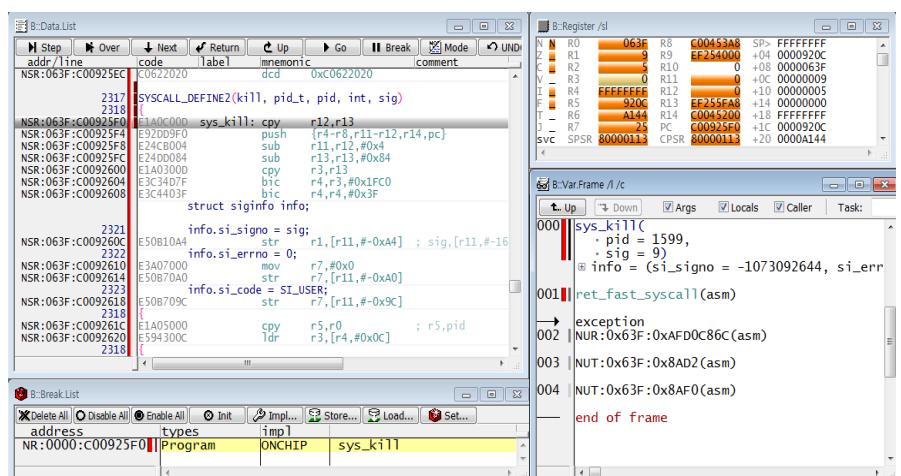


7-5. MDS daemon Debugging

1.1.3. 어느 정도 부팅이 완료되면
break로 target을 멈추고,
symbol browser에서 sys_kill
함수에 breakpoint를 설정한다.
Process가 종료되면 sys_kill을
통해 메모리를 반환하고
종료되기 때문이다.

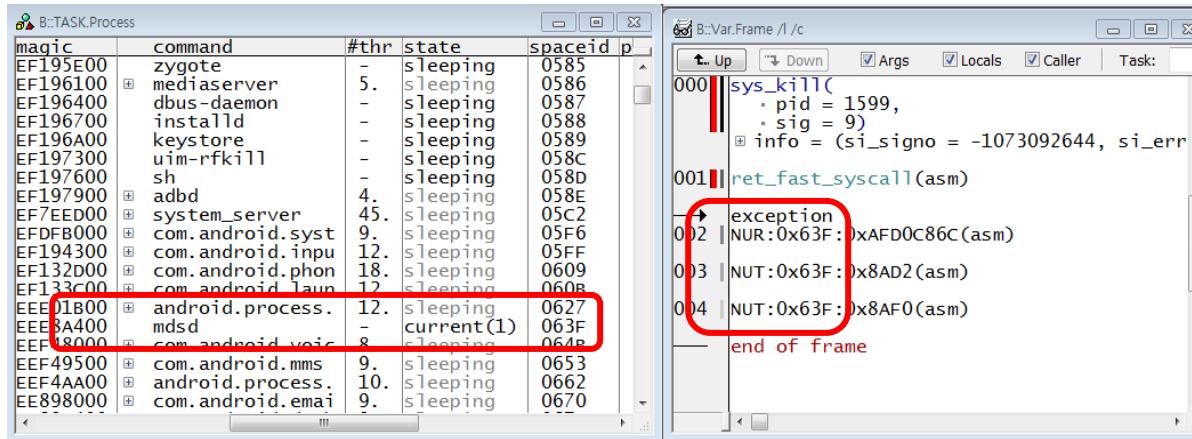


1.1.4 target을 다시 running 시키면
mdsd가 kill이 될 때 설정된
sys_kill에 멈추게 된다.

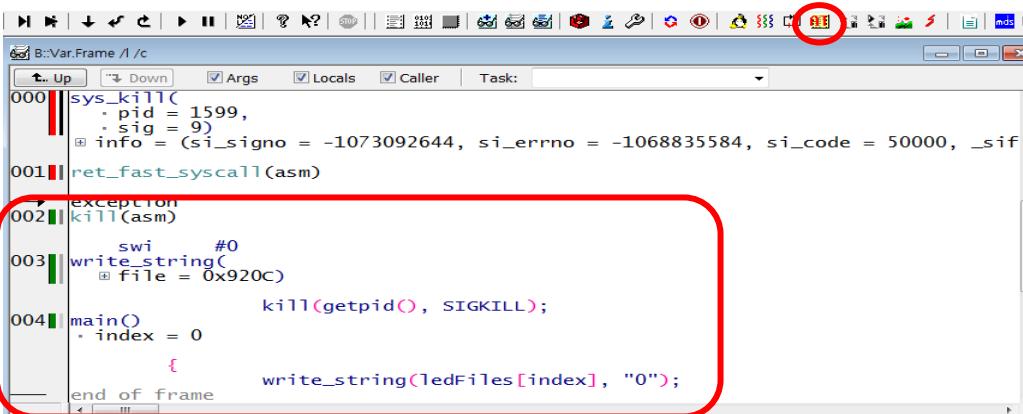


7-5. MDS daemon Debugging

1.1.5. 이 때 현재 kill된 process가 mdsd인지 linux 메뉴의 display process를 통해 확인한다.

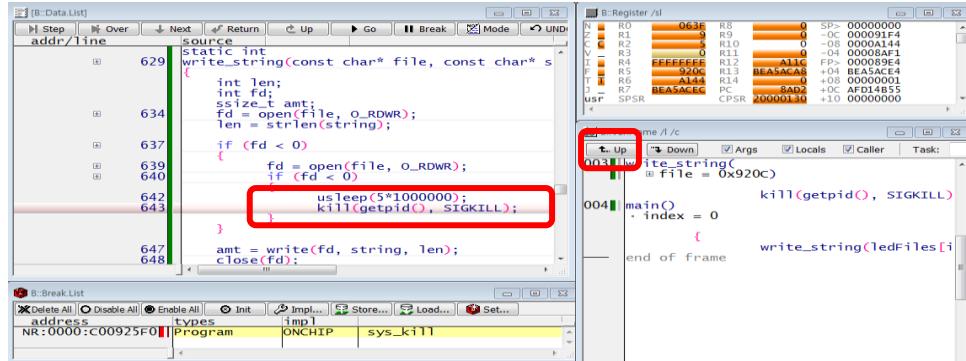


1.1.6. exception이 발생된 이전의 process에 대한 symbol이 안보이기 때문에 버튼을 눌러 symbol을 load한다.

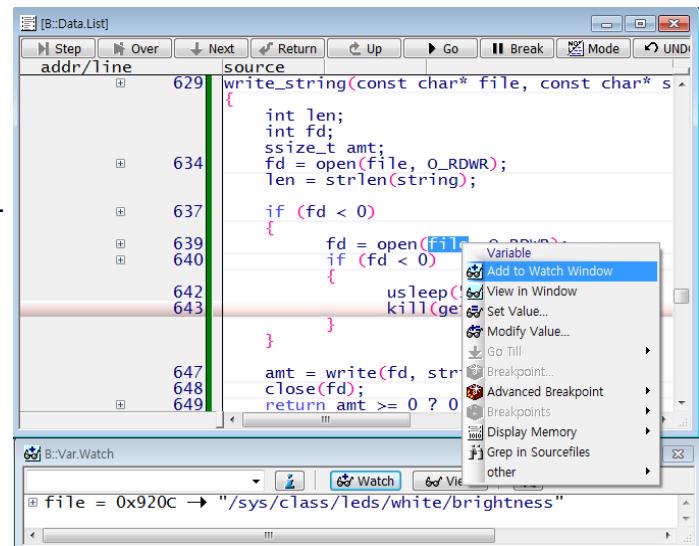


7-5. MDS daemon Debugging

1.1.7. 이제 실제 문제가 발생한 위치를 확인하기 위해 call stack의 up 버튼을 통해 mdsd 코드로 이동한다.



1.1.8. kill이 되는 부분은 fd 변수가 0 이하이기 때문이고 open을 통해서 fd값을 가지고 오는 것을 확인할 수 있다.
그래서 해당 file이 제대로 open이 안 되는 것이기 때문에 file open하는 정보를 확인한다.



7-5. MDS daemon Debugging

1.1.9. file 정보를 통해 실제 filesystem에 해당 file이 존재하는지 확인한다.

The screenshot shows two windows. The top window is titled 'B::Var.Watch' and displays a variable 'file' with the value '0x920c → "/sys/class/leds/white/brightness"'. The bottom window is a terminal window showing a root shell on an Android device. The terminal history includes:

```
root@android:/sys/class/leds # ls
ls
mmc0::
mmc1::
mmc2::
origen::status1
origen::status2
root@android:/sys/class/leds # cd white
cd white
/system/bin/sh: cd: /sys/class/leds/white: No such file or directory
2 !root@android:/sys/class/leds #
```

The line '/system/bin/sh: cd: /sys/class/leds/white: No such file or directory' is highlighted with a red box.

실제 해당 file이 filesystem에 존재하지 않아 open에서 제대로 값을 가지고 오지 못하고 이로 인해 kill 이 되는 것을 확인할 수 있다.

8. Library

Android에서의 중요 기능인 Library에 대해서 이해하고 Library 디버깅을 할 수 있습니다

1. Android Library

2. Library 종류

- Static Library
- Shared Library – Dynamic Linking
- Shared Library – Dynamic Loading

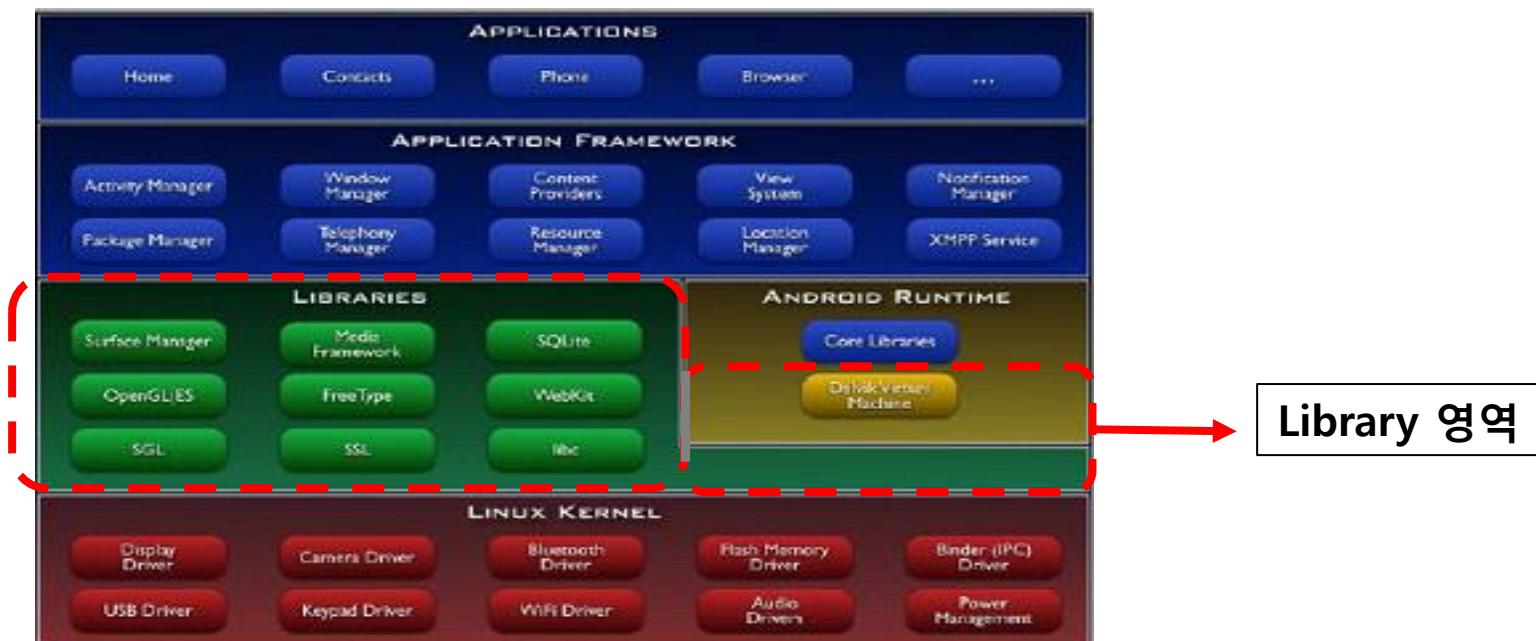
3. Library 실습

- Shared Library - Dynamic Linking 실습
- Shared Library - Dynamic Loading 실습

8-1. Android Library

Android에서 실행되는 대부분의 Application들은 System Resource들을 사용하기 위해 Library에 있는 코드들을 수행하게 됩니다

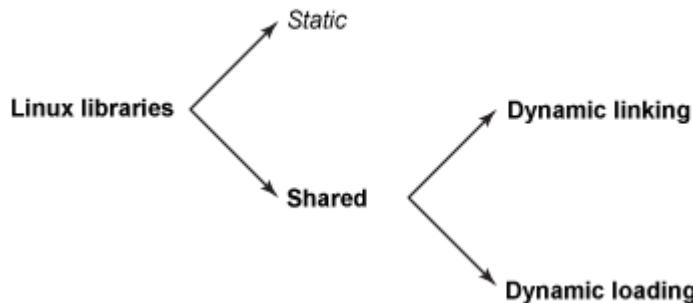
- Android Application에서 Dalvik에 의해 넘어와 Kernel을 수행하기 전 중요한 Routine들을 모아놓은 Object들입니다
- 대부분의 Application들은 Library들을 사용하게 됩니다
- 이 Library들은 부팅 중 Daemon에 의해 대부분 Memory에 Load됩니다
- JNI, Binder, HAL 등을 포함 합니다
- Dalvik 도 Native Library에 포함 됩니다 (libdvm.so)



8-2. Library 종류

Library의 종류에 대해서 알아보고 이해합니다

Library는 Linking과 loading 방식에 따라 구분할 수 있다.



종 류	Symbol reference resolve 시기 즉 프로그램에 라이브러리 적재 시기	라이브러리 사용 ?
정적 라이브러리 (Static Library)	컴파일 타임	컴파일 시 라이브러리를 적재한 그 프로그램만 라이브러리 코드를 사용.
공유 라이브러리 (Dynamic Linking)	런타임 (프로그램 메모리에 적재 시)	메모리에 라이브러리 적재 되 있으면 그 라이브러리 사용하는 프로그램끼리 한 그 메모리영역(라이브러리)를 공유한다.
동적 라이브러리 (Dynamic Loading)	런타임 (프로그램 실행 중 필요할 때) 즉 사용하는 응용프로그램이 결정	메모리에 라이브러리 적재 되 있으면 그 라이브러리 사용하는 프로그램끼리 한 그 메모리영역(라이브러리)를 공유한다.

8-3. Shared Library - Dynamic Linking 실습

실습을 통해 동일한 address에 mapping되는 Dynamic Linking 형태의 library 디버깅을 이해할 수 있습니다

Shared Library - Dynamic Linking 의 경우에는 zygote에 의해서 pre-load된 library를 중심으로 디버깅하는 방법입니다

이런 library의 경우 shared library 형태로 zygote에 load된 library의 address와 application에 load되는 address가 동일합니다.

8-3. Shared Library - Dynamic Linking 실습

실습을 통해 동일한 address에 mapping되는 Dynamic Linking 형태의 library 디버깅을 이해할 수 있습니다

1. zygote에 의해 pre-load된 Shared Library Debugging

Browser application을 실행 했을 때 window를 그려주는 함수를 디버깅해보자.

디버깅하는 함수는 eglcreatewindowsurface() 함수이다.

1.1 먼저 해당 함수가 포함되어 있는 library를 찾아보자.

해당 함수는 android\frameworks\native\opengl\libs\EGL\eglApi.cmm 파일에 포함되어 있다.

The screenshot shows two windows. On the left is a file browser with the path: android > frameworks > native > opengl > libs > EGL. It lists several files: egl.cpp, egl_cache.cpp, egl_cache.h, egl_display.cpp, egl_display.h, egl_entries.in, egl_object.cpp, egl_object.h, egl_tls.cpp, and egl_tls.h. The file eglApi.cpp is highlighted with a red rectangle. On the right is a code editor window titled 'eglApi.cpp' showing the following code:

```
// -----
232 EGLSurface eglCreateWindowSurface( EGLDisplay dpy, EGLConfig config,
233 NativeWindowType window,
234 const EGLint *attrib_list)
235 {
236     clearError();
237
238     egl_connection_t* cnx = NULL;
239     egl_display_ptr dp = validate_display_connection(dpy, cnx);
240     if (dp) {
241         EGLDisplay iDpy = dp->disp.dpy;
242         EGLint format;
```

8-3. Shared Library - Dynamic Linking 실습

1.2 동일한 디렉토리 혹은 상위 디렉토리의 Android.mk 파일을 통해 생성되는 library를 확인하자.

The screenshot shows a file browser window with the path: android > frameworks > native > opengl > libs. In the left pane, there are several directories: EGL, ETC1, GLES_CM, GLES_trace, GLES2, and tools. A file named 'Android.mk' is highlighted with a red box. In the right pane, the content of the 'Android.mk' file is displayed:

```
1 LOCAL_PATH := $(call my-dir)
2
3 #####
4 # Build META EGL library
5 #
6
7 include $(CLEAR_VARS)
8
9 LOCAL_SRC_FILES :=
10    EGL/egl_tls.cpp \
11    EGL/egl_cache.cpp \
12    EGL/egl_display.cpp \
13    EGL/egl_object.cpp \
14    EGL/egl.cpp \
15    EGL/eglApi.cpp \
16    EGL/trace.cpp \
17    EGL/getProcAddress.cpp.arm \
18    EGL/Loader.cpp \
19 #
20
21 LOCAL_SHARED_LIBRARIES += libutils libutils libGLES_trace
22 LOCAL_LDLIBS := -lpthread -ldl
23 LOCAL_MODULE := libEGL
24 LOCAL_LDFLAGS += -Wl,--exclude-libs=ALL
25 LOCAL_SHARED_LIBRARIES += libdl
```

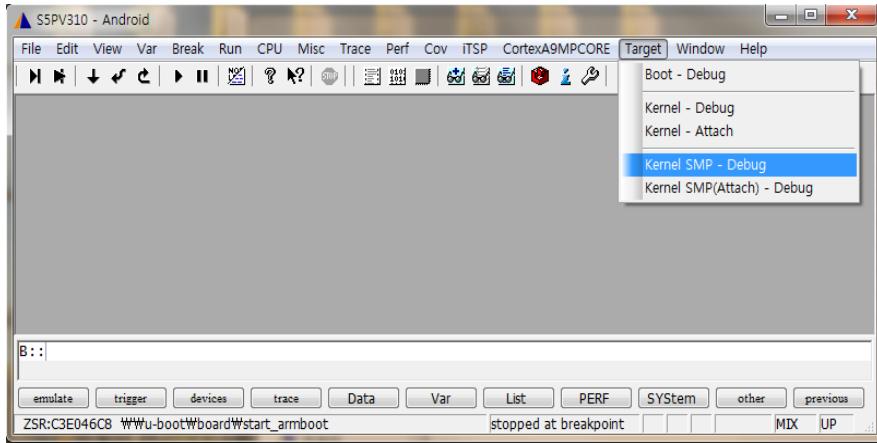
The lines 'LOCAL_MODULE := libEGL' and 'LOCAL_SHARED_LIBRARIES += libdl' are also highlighted with a red box.

Android.mk 파일을 통해 해당 함수가 포함된 파일을 빌드한 후 생성된 library를 LOCAL_MODULE:=libEGL 을 통해 확인할 수 있다.

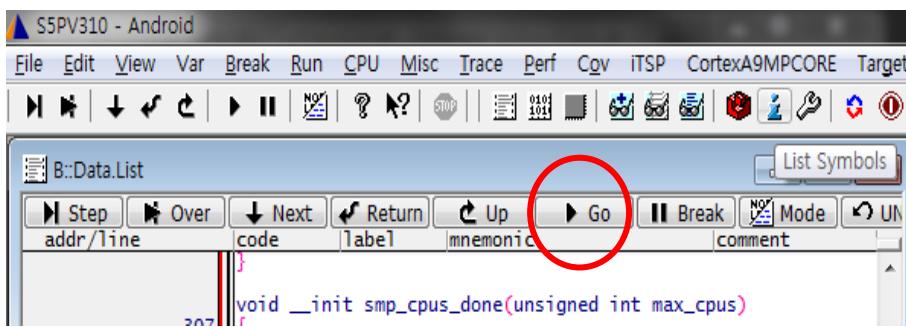
즉, 디버깅해야 하는 eglCreateWindowSurface() 함수는 libEGL.so라는 library에 포함이 되어 있다는 얘기가 된다.

8-3. Shared Library - Dynamic Linking 실습

1.3 kernel SMP debugging 환경을 선택하자.



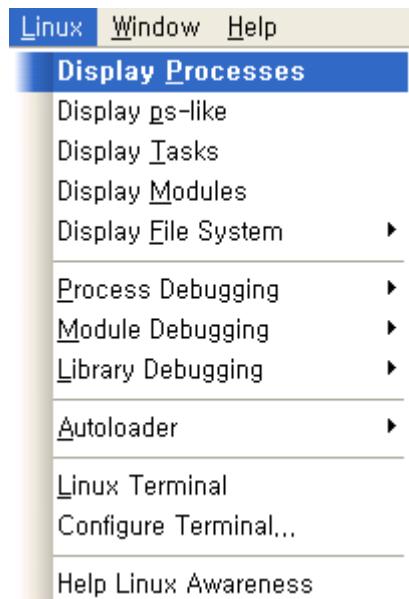
1.4 target을 running 시켜 부팅을 완료시키자.



8-3. Shared Library - Dynamic Linking 실습

1.5 부팅이 완료되면 target을 break하고 libEGL.so Symbol을 Load하자. 해당 library는 shared library이기 때문에 zygote에서 해당 library의 symbol을 load하자. Zygote를 이용하는 이유는 현재 browser application이 실행이 되지 않은 상태이기 때문에 browser application을 통해 해당 symbol을 load할 수 없기 때문이다. 또한 browser application이 실행되면 zygote에서 fork()를 통해 pre-load된 library를 그대로 사용하기 때문에 zygote에서 load된 libEGL.so와 browser application이 실행 되었을 때 load되는 libEGL.so 는 동일한 address를 사용한다.

Linux → Display processes 메뉴를 선택한다. 그리고 zygote를 찾아 double click 한다.



B::TASK.Process		
magic	command	#thr
ED038000	netd	6.
EC9D4000	debuggerd	-
EC9D4800	rild	3.
EC9D4C00	surfaceflinger	8.
EC9D5000	zygote	4.
EC9D5400	drmservice	2.
EC9D5C00	mediaserver	5.
EC9D6000	dbus-daemon	-
EC9D6400	installd	-
EC9D6800	keystore	-
EC9D6C00	gatord	-
EC9D7000	adbd	4.
C19DF800	system_server	65.

B::Task.DTask 0xC78192C0				
magic	command	state	gid	vm_size ttb tty r
C78192C0	system_server	sleepir	1000.	00008F65 C7EA4000 -
+ flags				
+ relationship				
+ arguments				
+ environment				
+ open files				
+ addresses				
+ code file				

새로 열리는 window에서 code file 부분을 확장한다.

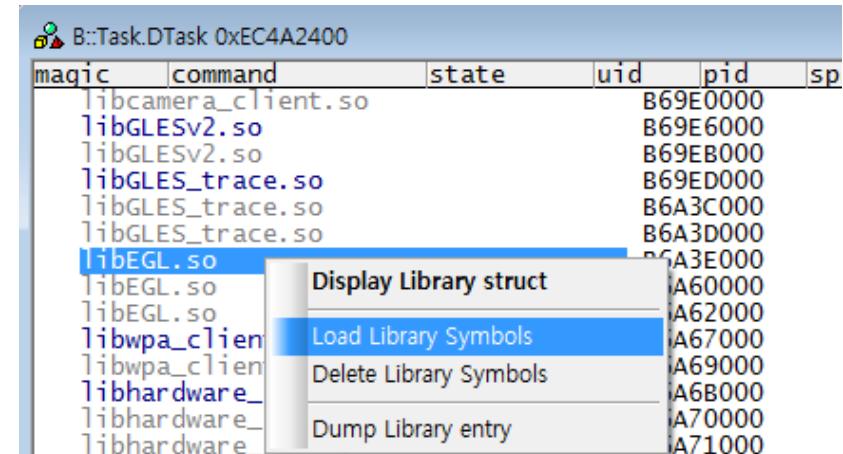
8-3. Shared Library - Dynamic Linking 실습

1.6 library 중 우리가 디버깅해야 하는

libEGL.so library를 찾는다.

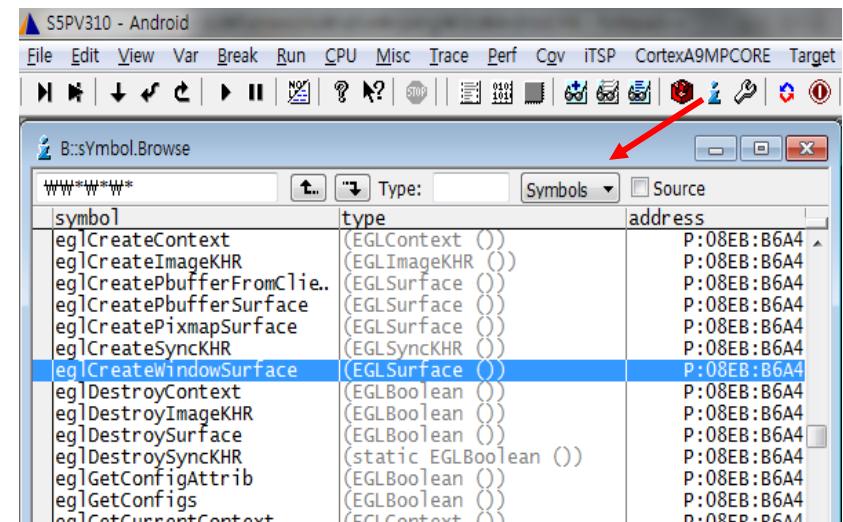
(ctrl+F로 찾을 수 있다.)

해당 library를 찾은 후 오른쪽 마우스 클릭에서 Load Library Symbols 기능을 통해 해당 library의 symbol을 load한다.



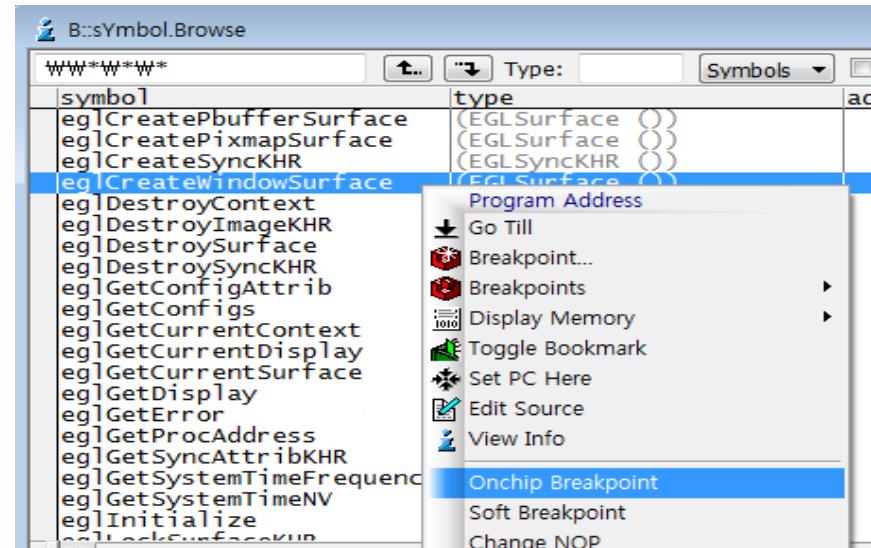
1.7 symbol load가 끝나면

symbol browser에서 디버깅해야 하는 함수를 찾는다.

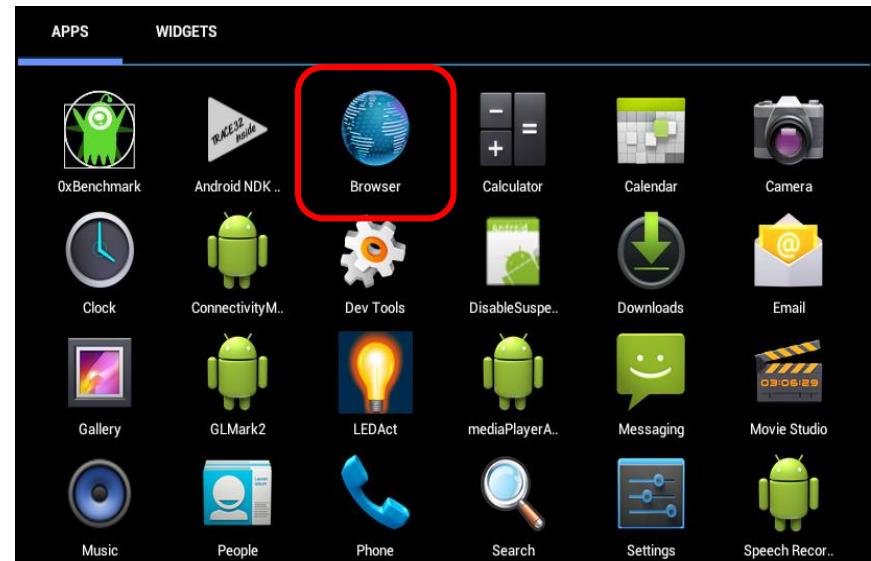


8-3. Shared Library - Dynamic Linking 실습

1.8 해당 함수를 찾은 후 오른쪽 마우스 클릭 후 breakpoint를 설정한다.

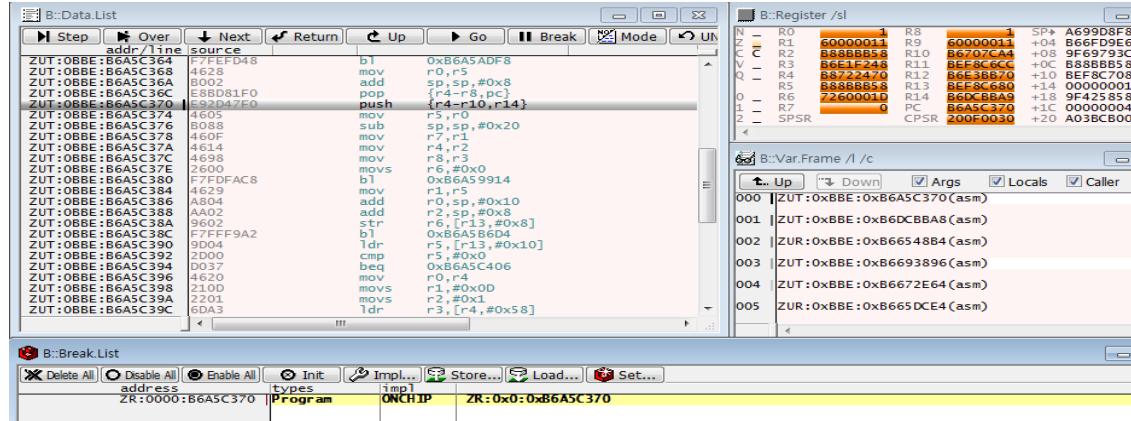


1.9 breakpoint 설정 후 go 버튼으로 target을 running 시키고 디버깅을 위해 browser application 을 실행한다.

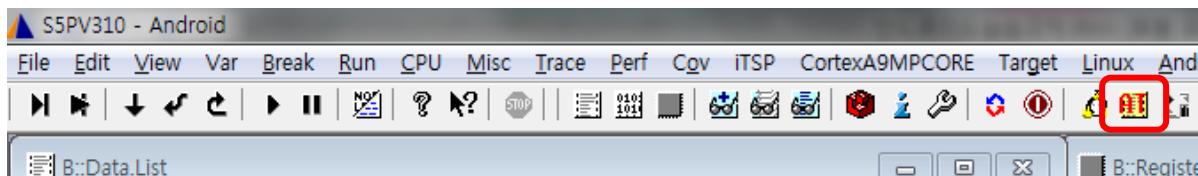


8-3. Shared Library - Dynamic Linking 실습

1.10 browser application을 실행하면 미리 설정되어 있는 breakpoint에 의해 target이 멈추게 된다.

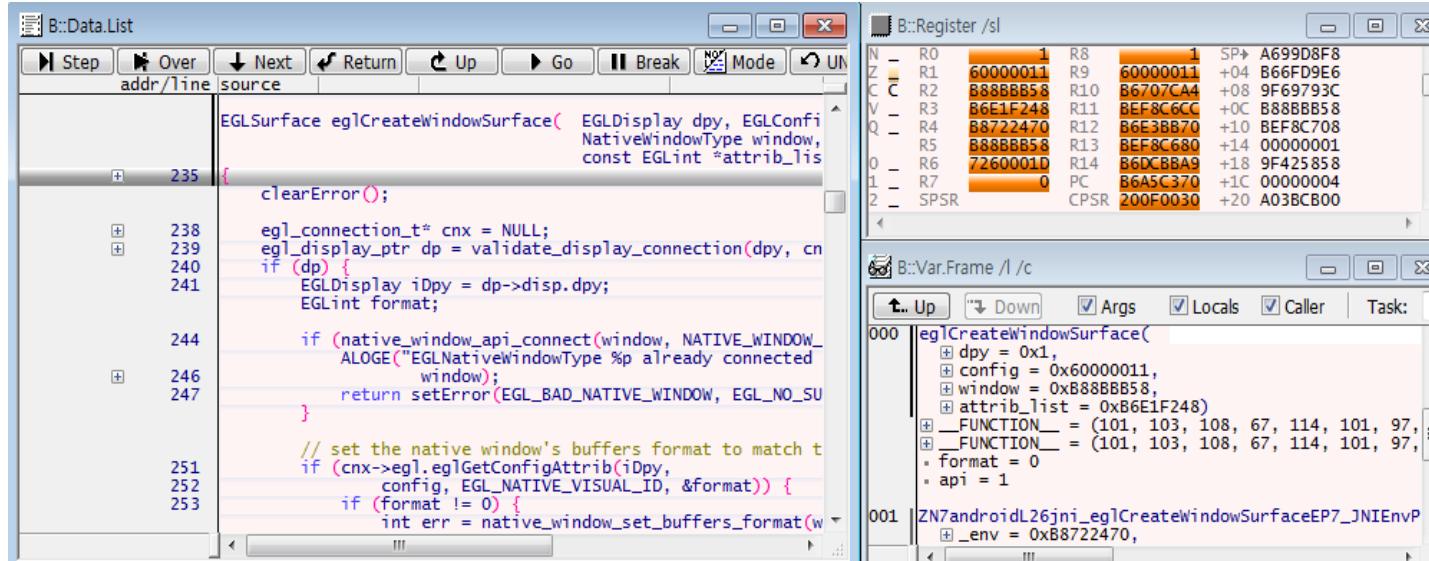


1.11 하지만 이전에 load했던 symbol 정보는 보이지 않게 된다. 이유는 load했던 libEGL.so 의 symbol은 zygote에서 load했기 때문에 해당 symbol은 zygote가 가지고 있는 memory 공간에 load된다. 실제 breakpoint에 의해 멈춘 부분은 zygote가 아닌 browser application이기 때문에 다른 메모리 공간이다. 실제 symbol 정보도 browser application의 메모리 공간으로 다시 load해야 한다. 이를 위해서 아이콘 중 아이콘을 누르면 된다.



8-3. Shared Library - Dynamic Linking 실습

그리면 TRACE32가 현재 process 기준(여기서는 browser application)으로 symbol을 다시 load한다.



- * 위와 같이 zygote에서 사용하는 library와 동일한 address를 사용하는 library의 경우에는 이와 같은 방법으로 디버깅하면 application을 실행되지 않은 상태에서 breakpoint를 설정한 후 디버깅이 가능하다.

8-4. Shared Library - Dynamic Loading 실습

실습을 통해 ASLR 방식인 Dynamic Loading library 디버깅을 이해합니다.

Shared Library - Dynamic Loading의 경우에는 linux kernel 3.0 부터 적용된 ASLR 방식과 dynamic library와 같은 형태로 같은 library이지만 다른 address에 load되는 방식입니다

이런 경우에는 해당 library를 load해주는 libdvm.so(JNI Interface 방식을 사용하는 Library) 혹은 libhardware.so(HAL Interface 방식을 사용하는 Library)를 통해서 해당 library를 load하고 디버깅을 해야 합니다

8-4. Shared Library - Dynamic Loading 실습

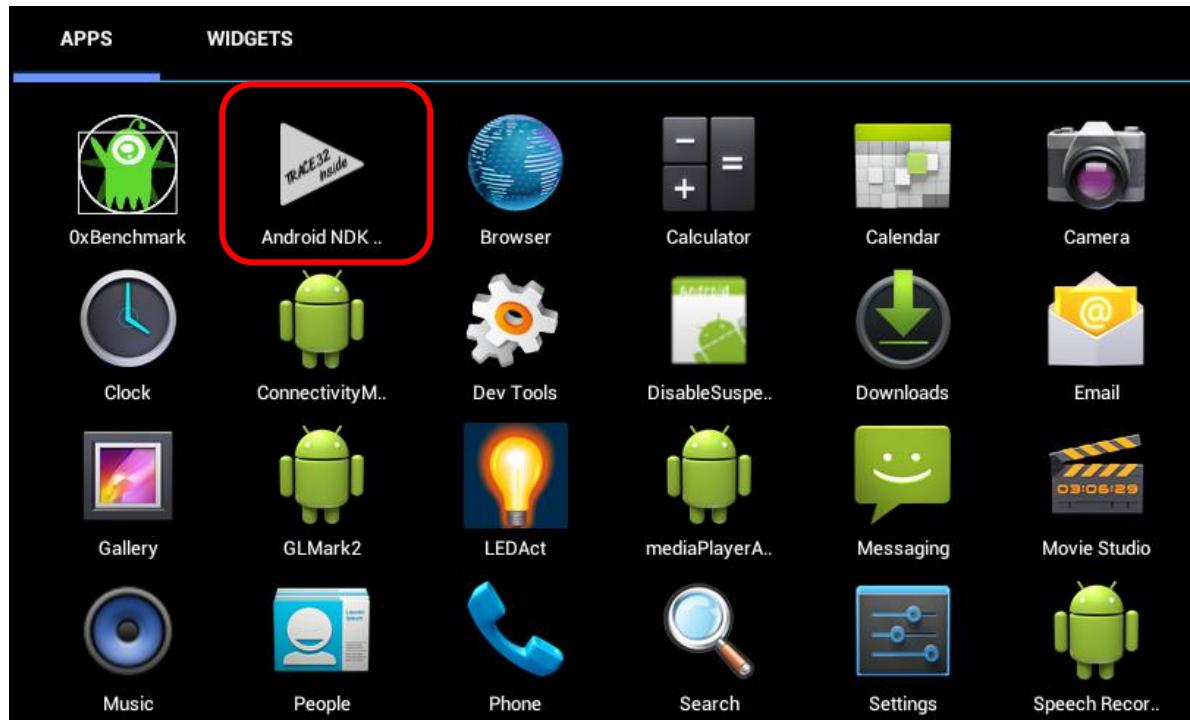
실습을 통해 ASLR 방식인 Dynamic Loading library 디버깅을 이해합니다.

1. Shared Library – Dynamic Loading Debugging – 실습 1

만약 특정 Application에서만 사용하는 JNI interface를 사용하는 Library의 경우 libdvm.so의 dvmLoadNativeCode함수를 이용한다.

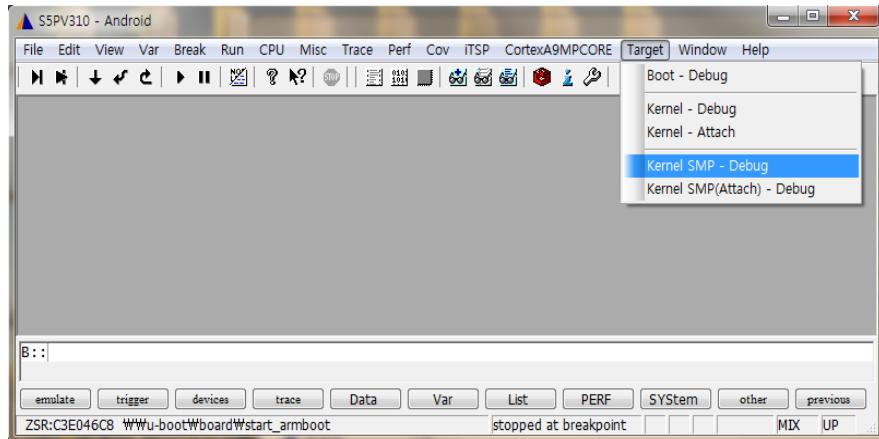
다음 application 실행 시 사용되는 JNI library를 디버깅 해보자.

디버깅해야 하는 library는 libhellolibrary.so 이다.

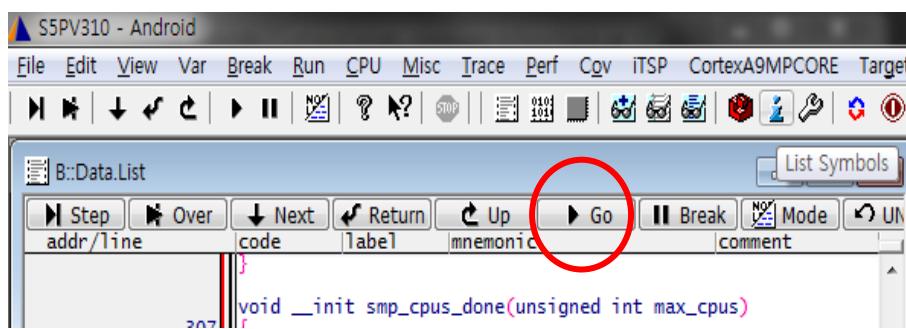


8-4. Shared Library - Dynamic Loading 실습

1.1 kernel SMP debugging 환경을 선택하자.



1.2 target을 running 시켜 부팅을 완료시키자.



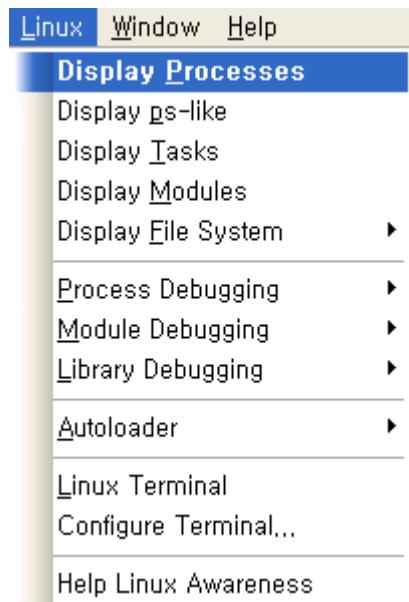
8-4. Shared Library - Dynamic Loading 실습

1.3 부팅이 완료되면 application을 실행하기 전 symbol을 load하자.

Application이 실행되기 전이기 때문에 zygote에서 libdvm.so 를 load한다.

Libdvm.so library symbol을 로드하는 이유는 jni library의 경우 사용되기 위해 memory에 load될 때 libdvm.so를 이용하기 때문이다.

Linux → Display processes 메뉴를 선택한다. 그리고 zygote를 찾아 double click 한다.



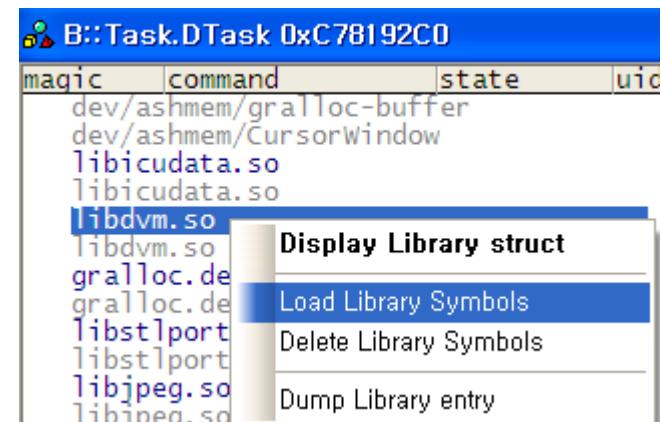
B::TASK.Process		
magic	command	#thr
ED038000	netd	6.
EC9D4000	debuggerd	-
EC9D4800	rild	3.
EC9D4C00	surfaceflinger	8.
EC9D5000	zygote	4.
EC9D5400	drmservice	2.
EC9D5C00	mediaserver	5.
EC9D6000	dbus-daemon	-
EC9D6400	installd	-
EC9D6800	keystore	-
EC9D6C00	gatord	-
EC9D7000	adbd	4.
C19DF800	system_server	65.

B::Task.DTask 0xC78192C0				
magic	command	state	gid	vm size ttb tty r
C78192C0	system_server	sleepir	1000.	00008F65 C7EA4000 -
+ flags				
+ relationship				
+ arguments				
+ environment				
+ open files				
+ addresses				
+ code file				

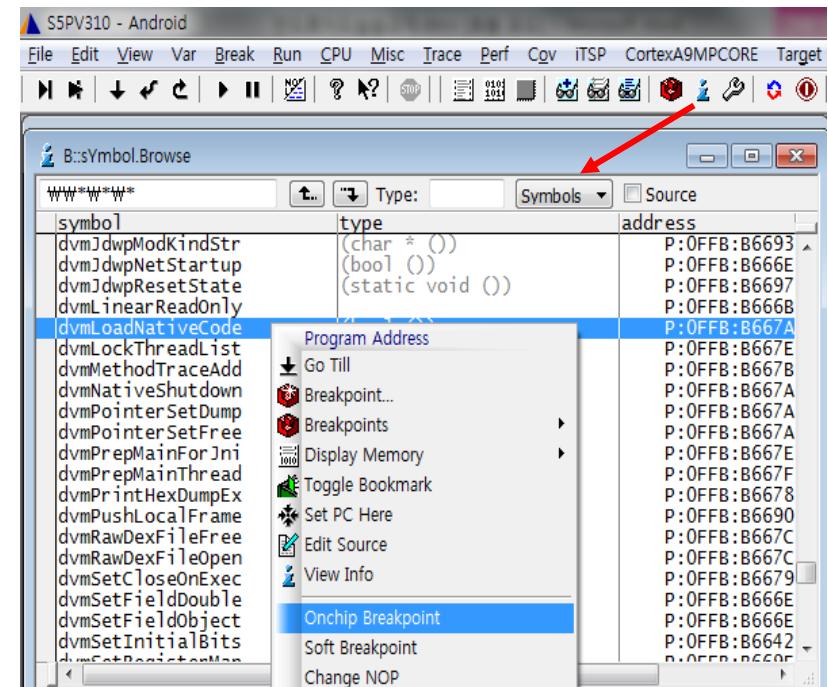
새로 열리는 window에서 code file 부분을 확장한다.

8-4. Shared Library - Dynamic Loading 실습

libdvm.so 를 찾고, 오른 쪽 마우스를 클릭한 후 Load Library Symbols 메뉴를 통해 해당 library symbol을 load한다.

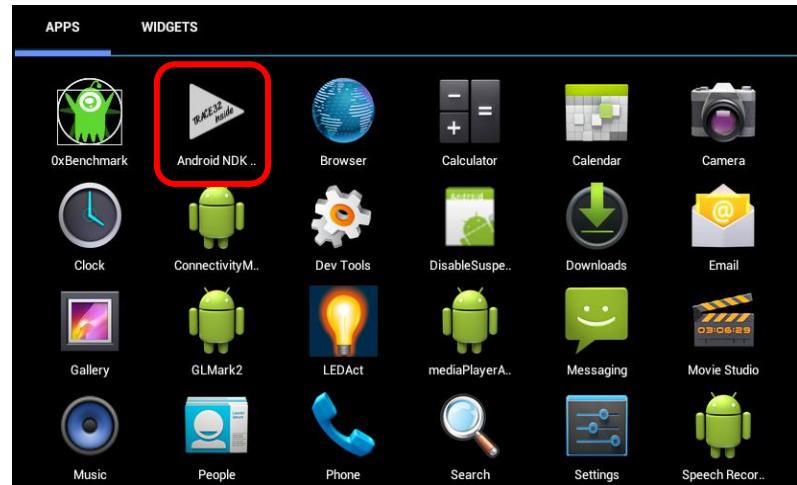


1.4 symbol이 load되면 symbol browser에서 libdvm의 dvmLoadNativeCode함수를 찾아 breakpoint를 설정한다.

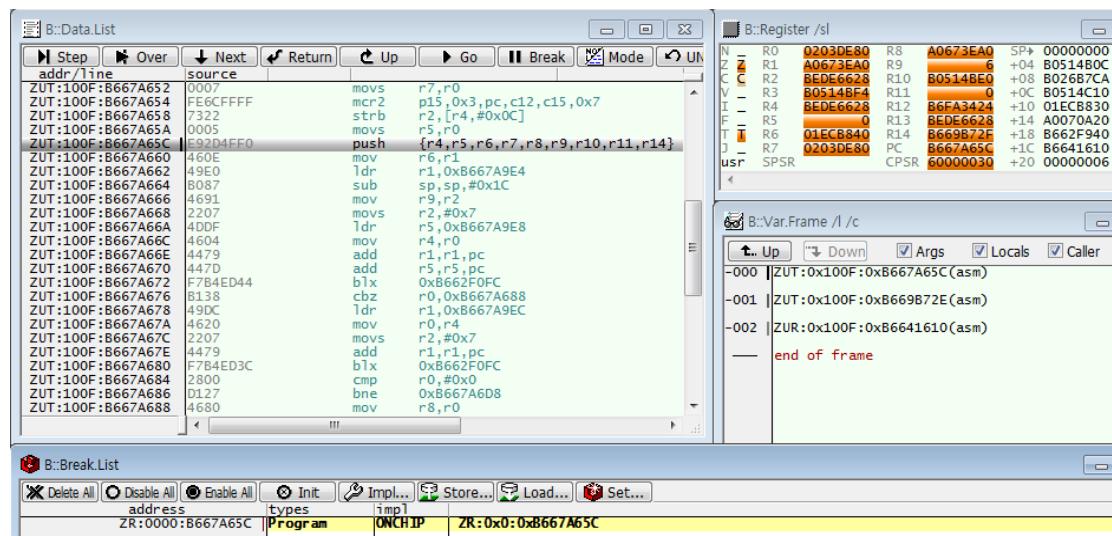


8-4. Shared Library - Dynamic Loading 실습

1.5 breakpoint 설정 후 target을 running하고 해당 application을 실행한다.

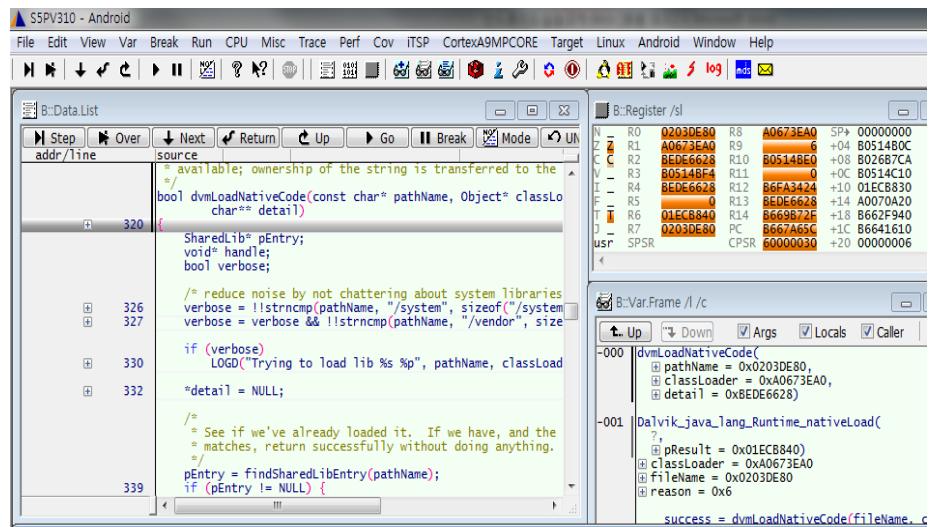


1.6 application을 실행하면 해당 application이 jni library를 사용하기 위해 dvmloadnativecode에 멈추게 된다.

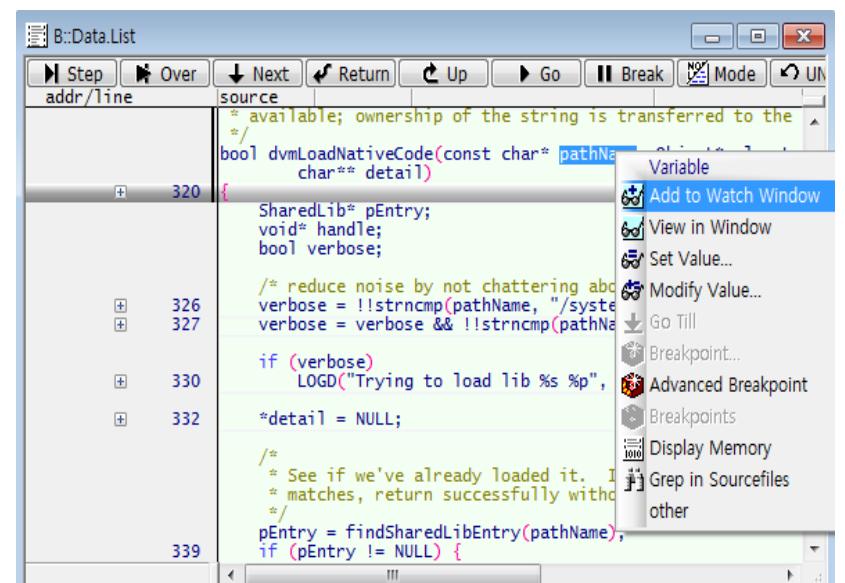


8-4. Shared Library - Dynamic Loading 실습

1.7 마찬가지로 load한 libdvm.so의 symbol은 zygote의 메모리 공간에 load한 것이기 때문에 현재는 application에 해당하는 libdvm.so를 다시 load해야 한다.
Symbol을 다시 로드하기 위해서는  아이콘을 누르면 된다.



1.8 symbol이 load된 후 디버깅해야 하는 library가 맞는지 인자로 들어오는 pathName 변수를 확인해보자.
해당 변수 정보를 확인하기 위해 data.list에서 해당 변수를 선택한 후 오른쪽 마우스 클릭해서 Add to Watch Window 메뉴를 선택하자

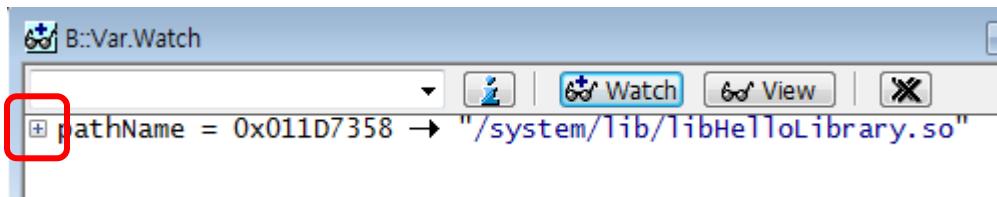


8-4. Shared Library - Dynamic Loading 실습

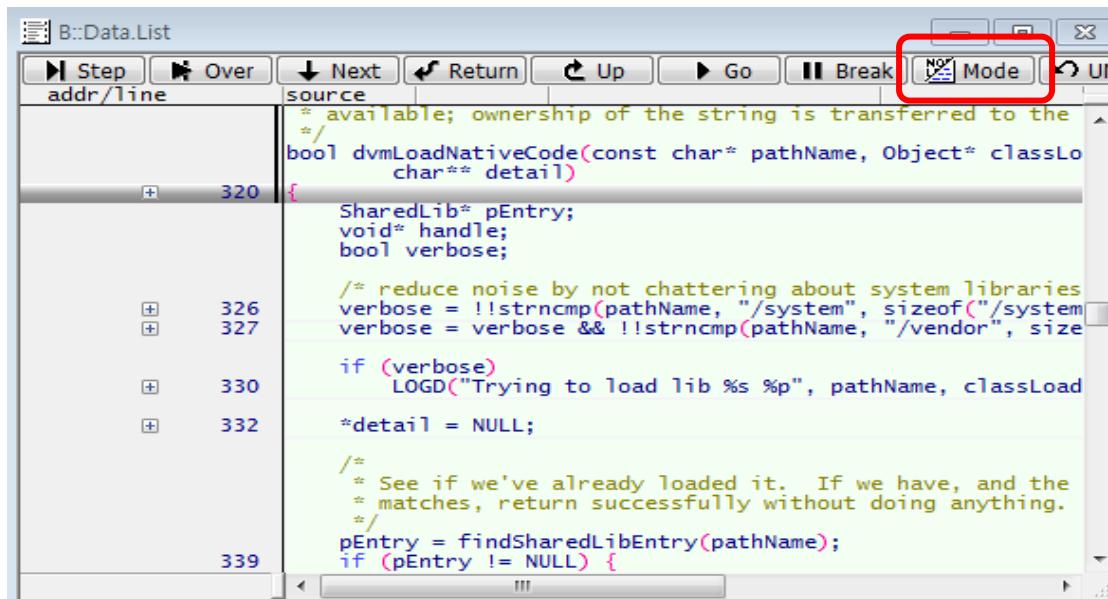
1.9 해당 메뉴를 선택하면 변수를 상세히 볼 수 있는 watch window를 볼 수 있다.

해당 변수의 string을 보기 위해서 앞에 + 부분을 더블클릭하자.

확인해 보면 디버깅해야 하는 library가 libHelloLibrary.so가 맞기 때문에 해당 library를 로드하는 부분까지 동작시킨다.



1.10 data.list의 mode 부분을 눌러 C소스만 보이도록 만든다.



8-4. Shared Library - Dynamic Loading 실습

1.11 scroll 을 내려서 해당 library를 load하는 dlopen() 함수를 호출하는 곳을 찾는다.
Dlopen()함수를 수행하면 위에서 인자로 넘어온 library가 실제 사용되기 위한 memory에 load된다.

```
[B::Data.List]
addr/line source
381     * to it with "strace -p <pid>" while it's
382     * attempting to open nonexistent dependent
383     *
384     * This can execute slowly for a large library
385     * want to switch from RUNNING to VMWAIT while
386     * the GC to ignore us.
387     */
388     Thread* self = dvmThreadSelf();
389     ThreadStatus oldStatus = dvmChangeStatus(self,
390         handle = dlopen(pathName, RTLD_LAZY);
391         dvmChangeStatus(self, oldStatus);
392
393     if (handle == NULL) {
394         *detail = strdup(dlerror());
395         return false;
396     }
397
398     /* create a new entry */
399     SharedLib* pNewEntry;
400     pNewEntry = (SharedLib*) malloc(1, sizeof(SharedLib));
401     pNewEntry->pathName = strdup(pathName);
402     pNewEntry->handle = handle;
403     pNewEntry->classLoader = classLoader;
404     pNewEntry->dvmInitMutex(&pNewEntry->mutex);
405     pthread_cond_init(&pNewEntry->cond, &pthread_mutexattr_default);
```

1.12 해당 library가 memory에 load되어야 symbol을 load할 수 있기 때문에 해당 루틴까지 코드를 수행한다.
해당 코드를 수행하기 위해 dlopen() 아래의 적절한 곳에 breakpoint를 설정하고 go한다.

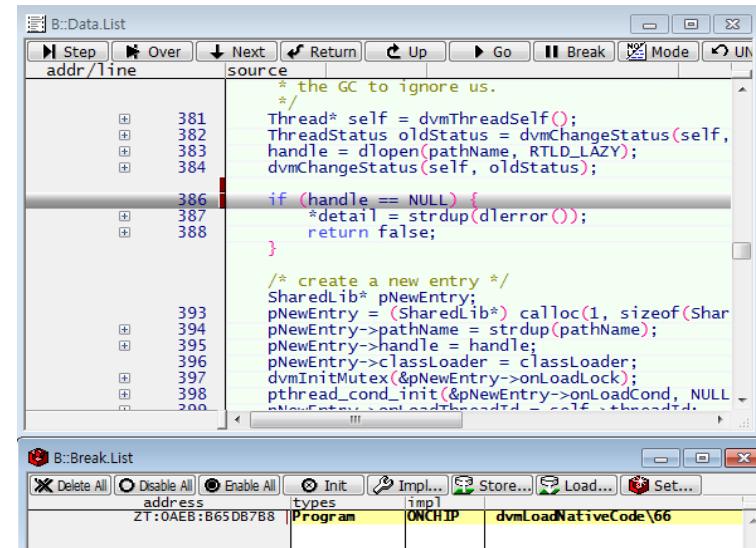
```
[B::Data.List]
addr/line source
381     * the GC to ignore us.
382     */
383     Thread* self = dvmThreadSelf();
384     ThreadStatus oldStatus = dvmChangeStatus(self,
385         handle = dlopen(pathName, RTLD_LAZY);
386         dvmChangeStatus(self, oldStatus);
387
388     if (handle == NULL) {
389         *detail = strdup(dlerror());
390         return false;
391     }
392
393     /* create a new entry */
394     SharedLib* pNewEntry;
395     pNewEntry = (SharedLib*) malloc(1, sizeof(SharedLib));
396     pNewEntry->pathName = strdup(pathName);
397     pNewEntry->handle = handle;
398     pNewEntry->classLoader = classLoader;
399     pNewEntry->dvmInitMutex(&pNewEntry->mutex);
400     pthread_cond_init(&pNewEntry->cond, &pthread_mutexattr_default);
```

- Program Address
- Go Till
- Breakpoint...
- Breakpoints
- Display Memory
- Toggle Bookmark
- Set PC Here
- Edit Source
- View Info

Onchip Breakpoint

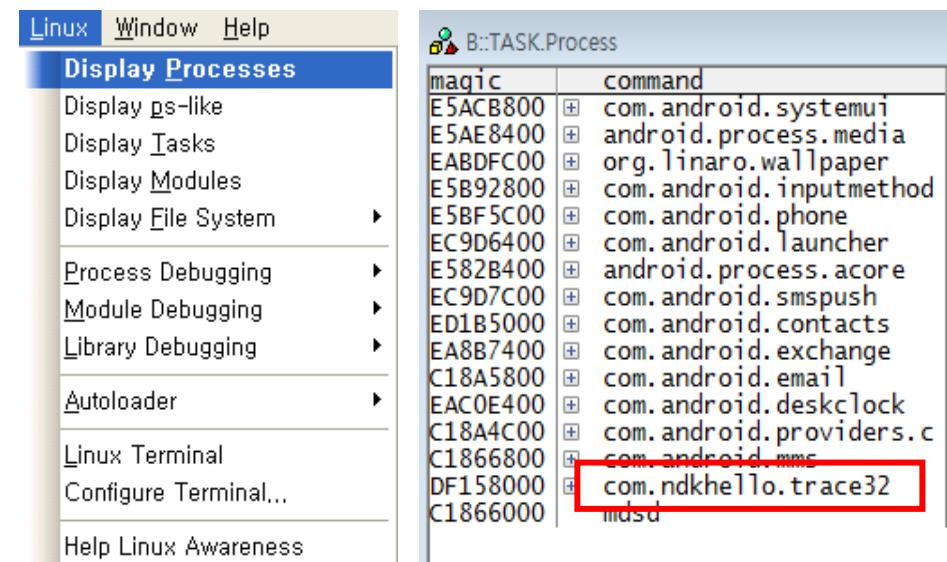
8-4. Shared Library - Dynamic Loading 실습

1.13 target이 동작하다가 breakpoint에 의해 멈추는 것을 확인할 수 있다.



1.14 실제 dlopen() 함수를 수행했기 때문에 해당 library가 load되었다. debugging을 위해 해당 library의 symbol을 로드하자.

Linux → Display processes 메뉴를 선택한다. 그리고 현재 current process인 com.ndkhello.trace32를 찾아 double click 한다.



8-4. Shared Library - Dynamic Loading 실습

새로 열리는 window에서 code file 부분을 확장한다.

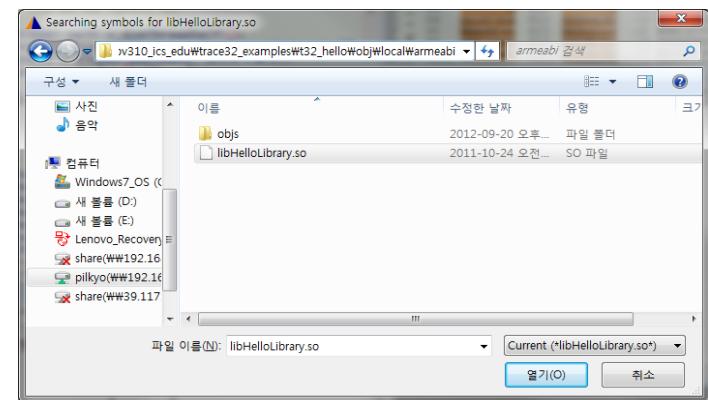
magic	command	state
DF158000	com.ndkHello.tra	current(1)
10024.	0001BC9E	DEC14000
gid	vm size ttb	tty name
<ul style="list-style-type: none">+ flags+ relationship+ arguments+ environment+ open files+ addresses- code file		

1.15 디버깅해야 하는 library인 libHelloLibrary.so를 찾는다.
찾은 후에 오른쪽 마우스 클릭한 후 Load Library Symbols 메뉴를 통해 해당 library symbol을 load한다.

magic	command	state	uid	pid
Roboto-Regular.ttf			9D99E000	
framework-res.apk			9D9E1000	
framework-res.apk			9DEBA000	
icudt46l.dat			9E0FD000	
system@framework@filterfw.jar@classe			9E7B9000	
system@framework@apache-xml.jar@clas			9E808000	
system@framework@services.jar@classe			9E96C000	
services.jar			9EB55000	
libHelloLibrary.so			9E957000	
libHelloLibrary.so				
libexif.so				
libexif.so				
system@framework@and				
system@framework@fra				
system@framework@ext				
system@framework@bou				
dev/ashmem/dalvik-LinearAlloc			9F801000	
dev/ashmem/dalvik-LinearAlloc			9F802000	
dev/ashmem/dalvik-LinearAlloc			9FA8F000	
dev/ashmem/dalvik-LinearAlloc			9FA90000	
dev/ashmem/dalvik-heap			^0001000	

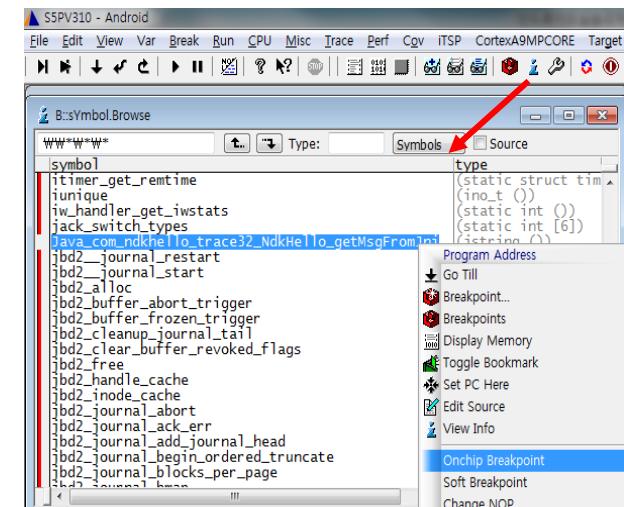
8-4. Shared Library - Dynamic Loading 실습

1.16 만약 자동으로 load가 되지 않고 symbol을 찾는 window가 뜨게 되면 아래 경로에서 해당 symbol을 찾아 선택해 준다. 해당 library는 아래의 경로에 존재한다.



BSP\trace32_examples\t32_hello\obj\local\armeabi\libHelloLibrary.so

1.17 symbol load 후 해당 library에 존재하는 함수를 디버깅하기 위해 symbol browser를 연다. 디버깅해야 하는 함수는

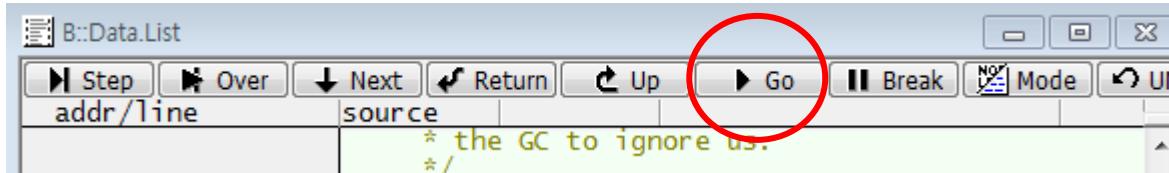


Java_com_ndkhello_trace32_NdkHello_getMsgFromJni()

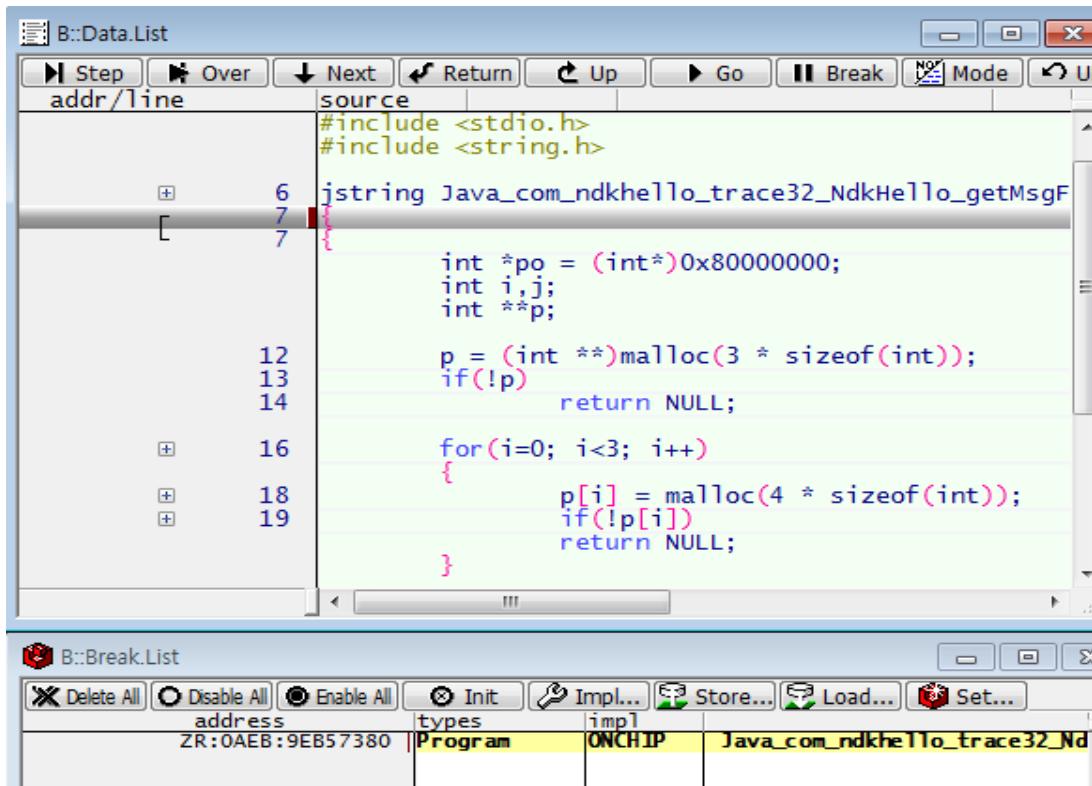
함수이다. 해당 함수를 symbol browser에서 찾고 breakpoint를 설정한다.

8-4. Shared Library - Dynamic Loading 실습

1.18 breakpoint 설정이 완료되면 target을 running 시킨다.



1.19 target이 running되면 동작하다가 설정한 breakpoint에 의해서 멈추게 된다.

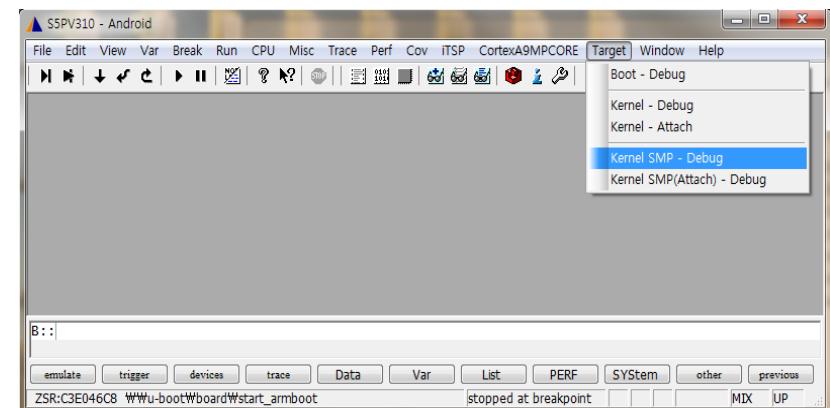


8-4. Shared Library - Dynamic Loading 실습

2. Shared Library – Dynamic Loading Debugging – 실습 2

부팅하는 과정 중에 한번만 실행되는 libwebcore.so의 JNI_OnLoad() 함수를 디버깅해보자. 해당 library는 zygote에서 load된 후 바로 호출되는 함수이기 때문에 timing 잡기가 쉽지 않다. 그래서 process debugging과 함께 활용을 해야 한다.

2.1 Kernel SMP debugging 환경을 만든다.

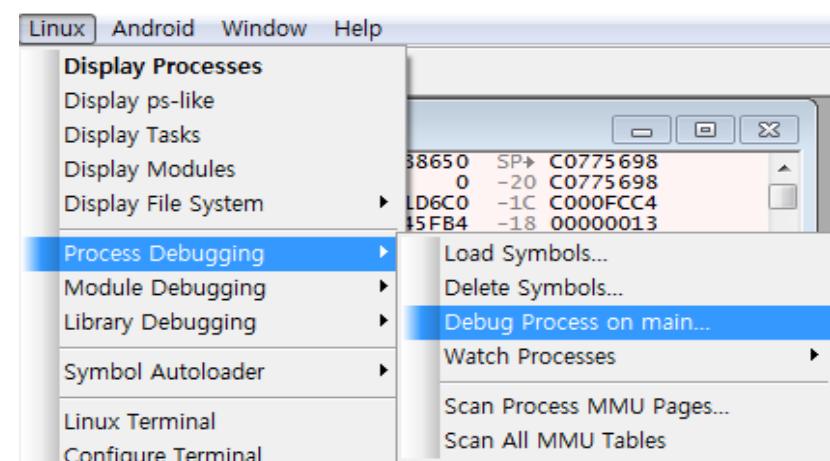


2.2 target을 running하기 전에 process

디버깅 메뉴를 선택한다.

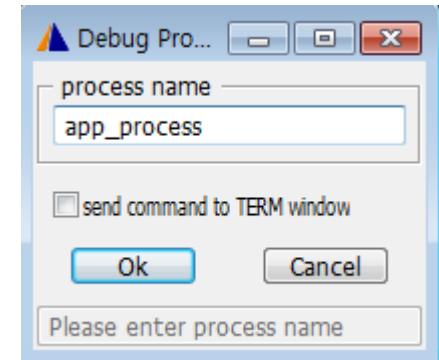
Linux → Process Debugging →

Debug process on main 메뉴를
선택한다.



8-4. Shared Library - Dynamic Loading 실습

2.3 zygote는 app_process가 변경이 되는 process이기 때문에 zygote를 디버깅하기 위해서는 app_process를 디버깅한다. app_process를 입력하고 ok 버튼을 누른다.



2.4 target이 자동으로 실행이 되고 app_process가 실행이 되면 TRACE32에 의해서 멈춘 후에 symbol을 load해준다. 그리고 나서 main에 멈춘 것을 확인할 수 있다.

B::Data.List

addr/line	source
128	strncpy(const_cast<char *>(argv0), newArgv0, strlen)
131	int main(int argc, const char* const argv[])
132	{
134	// These are global variables in ProcessState.cpp
135	mArgC = argc;
136	mArgV = argv;
137	mArgLen = 0;
138	for (int i=0; i<argc; i++) {
139	mArgLen += strlen(argv[i]) + 1;
141	}
142	mArgLen--;
144	AppRuntime runtime;
145	const char* argv0 = argv[0];
146	// Process command line arguments
147	// ignore argv[0]
148	argc--;
	argv++;

B::TASK.Process

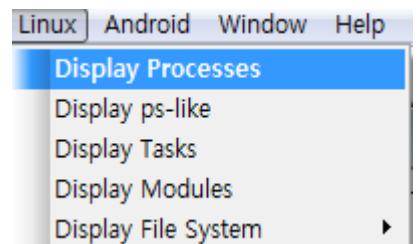
magic	command	#thr	state	sp
C07A0108	swapper/0	44.	running	C
ED00A000	init	-	running	C
ED2CA400	ueventd	-	running	C
C18C1400	sh	-	sleeping	C
C18C0800	servicemanager	-	running	C
C18C0400	void	3.	sleeping	C
C18C0000	netd	6.	sleeping	C
C18C0C00	debuggerd	-	sleeping	C
C181BC00	rild	3.	sleeping	C
C181B000	surfaceflinger	3.	running	C
C181B400	app_process	-	current(0)	C
C181B000	drmserver	-	sleeping	C
C181AC00	mediaserver	-	running	C
C181A800	dbus-daemon	-	sleeping	C
C181A400	installd	-	sleeping	C
C1819C00	keystore	-	sleeping	C
C1819800	gatord	-	sleeping	C
C180B000	adb	4.	sleeping	C

8-4. Shared Library - Dynamic Loading 실습

2.5 app_process가 zygote로 변환된 후 libdvm.so를 통해 libwebcore.so를 load하게 된다. 그래서 app_process가 pre-load한 libdvm.so를 통해 libwebcore.so를 load하는 시점을 찾아야 한다.

Linux → Display Processes

메뉴를 선택해서 app_process를 찾아서 더블클릭한다.



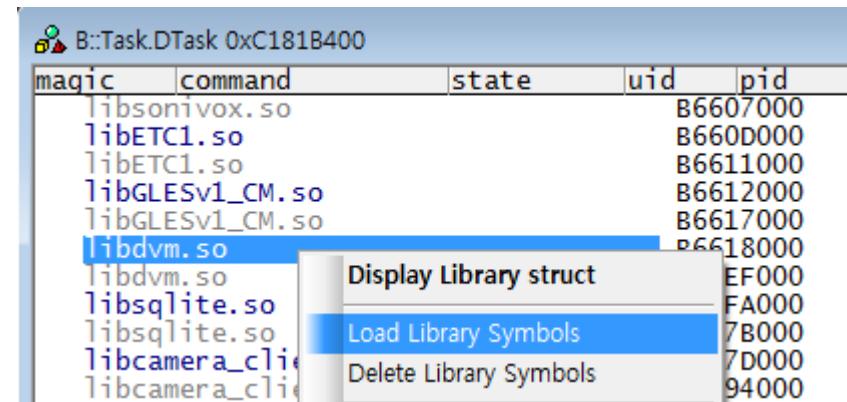
magic	command	#thr	state	sp
C07A0108	swapper/0	44.	running	C
ED00A000	init	-	running	C
ED2CA400	ueventd	-	running	C
C18C1400	sh	-	sleeping	C
C18C0800	servicemanager	-	running	C
C18C0400	vold	3.	sleeping	C
C18C0000	netd	6.	sleeping	C
C18C0C00	debuggerd	-	sleeping	C
C181BC00	rild	3.	sleeping	C
C181B800	surferfinger	3.	running	C
C181B400	app_process	-	current(0)	C
C181B000	drimserver	-	sleeping	C
C181AC00	mediaserver	-	running	C
C181A800	dbus-daemon	-	sleeping	C
C181A400	installld	-	sleeping	C
C1819C00	keystore	-	sleeping	C
C1819800	gatord	-	sleeping	C
C180B000	adbd	4.	sleeping	C

2.6 app_process를 더블클릭 한 후 열려진 window에서 code file 항목을 확장한다.

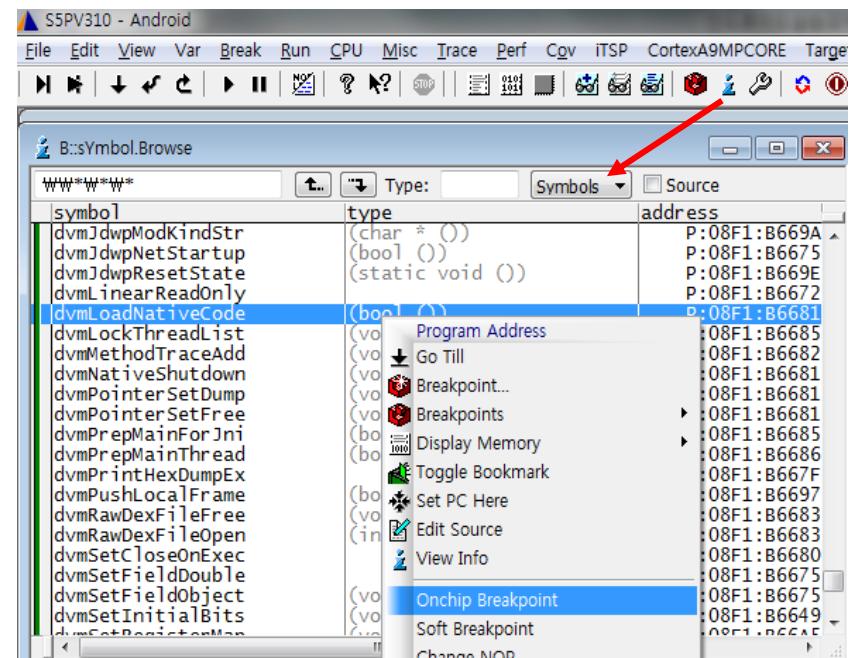
magic	command	state		
C181B400	app_process	current(0)		
gid	vm size	ttb	tty	name
0.	00000B83	C1840000	-	
<ul style="list-style-type: none">+ flags+ relationship+ arguments+ environment+ open files+ addresses- code file				

8-4. Shared Library - Dynamic Loading 실습

2.7 libdvm.so 를 찾고 오른쪽 마우스 클릭 후 Load Library Symbols 메뉴를 통해 symbol을 로드한다.

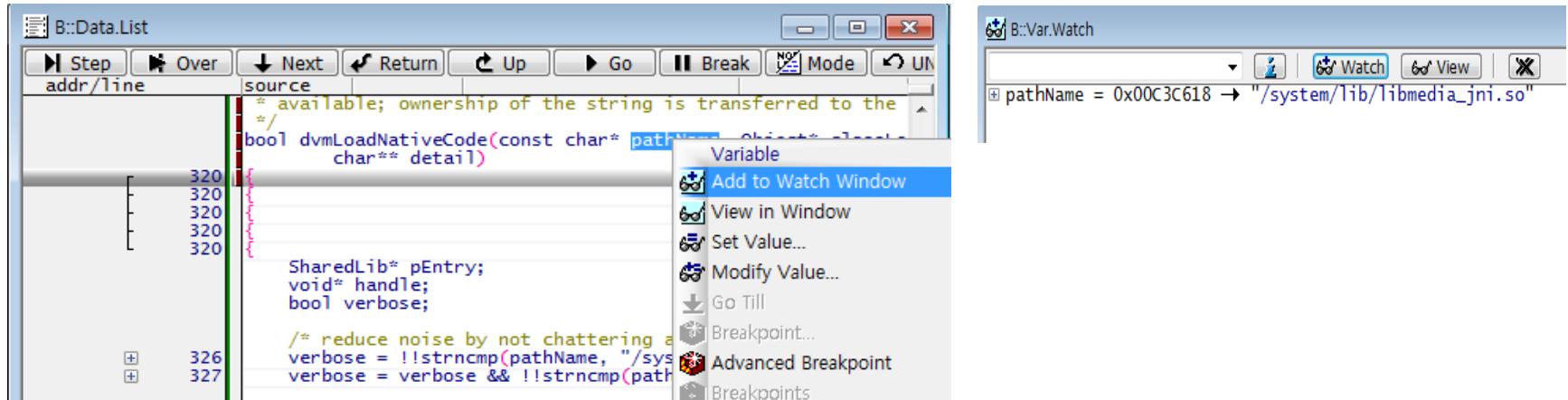


2.8 symbol load 후 symbol browser에서 dvmloadnativecode() 함수를 찾아 breakpoint를 설정한다.



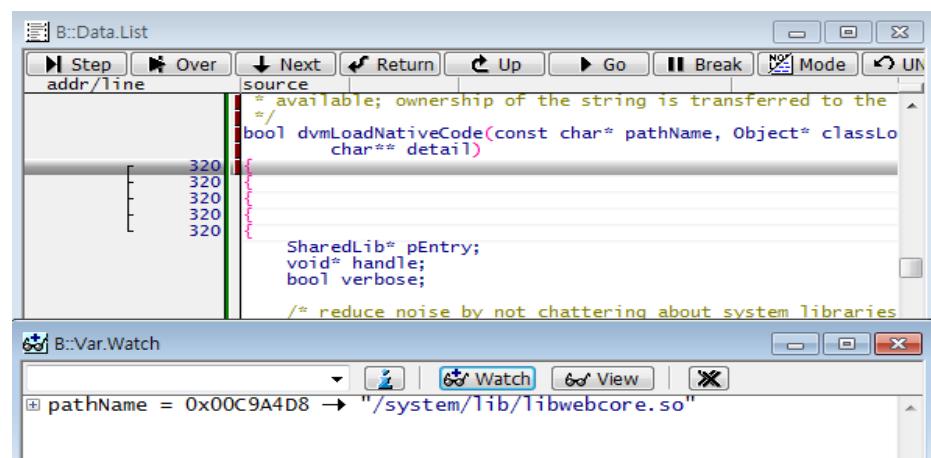
8-4. Shared Library - Dynamic Loading 실습

2.9 breakpoint 설정 후에 target을 running 시킨다. Breakpoint에 의해서 멈추게 되면 디버깅해야 하는 libwebcore.so 인지 pathname을 통해 확인한다.



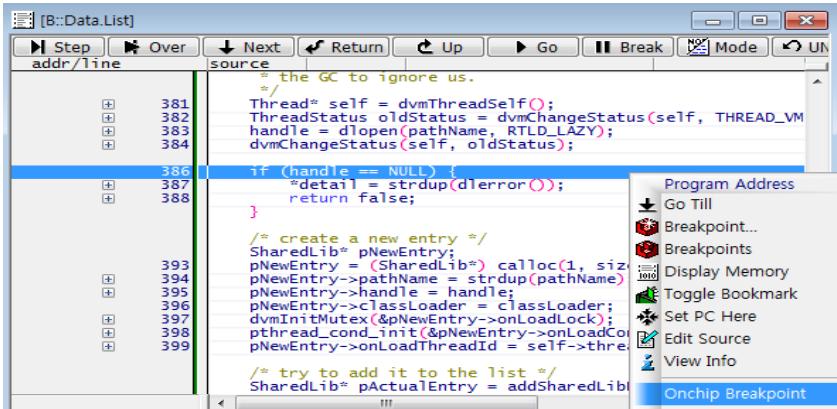
만약 이와 같이 libwebcore.so가 아니면 다시 target을 running해서 libwebcore.so를 로딩할 때까지 진행한다.

libwebcore.so를 로딩하는 시점이 나온다면, data.list의 mode를 눌러 C소스만 보이게 한 후 scroll을 내려 dlopen()함수를 찾는다.

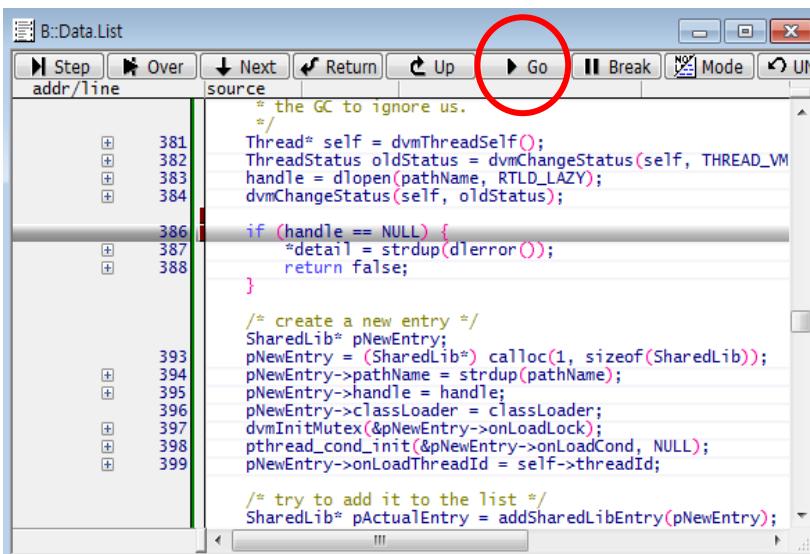


8-4. Shared Library - Dynamic Loading 실습

2.10 dlopen함수를 호출하는 아래 적당한 위치에 breakpoint를 설정한다.



2.11 breakpoint 설정 후 target을 running 시켜 breakpoint까지 동작시킨다.

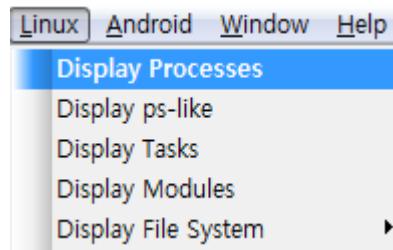


8-4. Shared Library - Dynamic Loading 실습

2.12 여기까지 동작하면 libwebcore.so가 사용되기 위해 memory에 load한 상황이다. 디버깅을 위해 libwebcore library symbol을 로드하면 된다.

Linux → Display Processes 메뉴를 선택해서 zygote를 찾는다.

Zygote를 검색하는 이유는 app_process가 zygote로 변경되었기 때문이다.



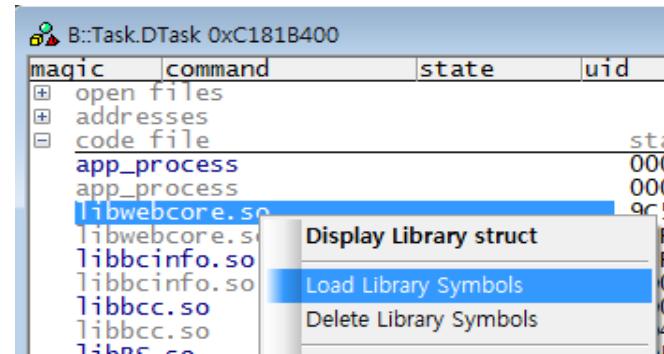
magic	command	#thr	state
C07A0108	swapper/0	44.	running
ED00A000	init	-	sleeping
ED2CA400	ueventd	-	sleeping
C18C1400	sh	-	sleeping
C18C0800	servicemanager	-	sleeping
C18C0400	vold	3.	sleeping
C18C0000	netd	6.	sleeping
C18C0C00	debuggerd	-	sleeping
C181BC00	rild	3.	sleeping
C181B800	surfaceflinger	7.	running
C181B400	zygote	4.	current
C181B000	drmserver	2.	sleeping
C181AC00	mediaserver	5.	sleeping
C181A800	dbus-daemon	-	sleeping
C181A400	installd	-	sleeping
C1810C00		-	-

magic	command	state		
C181B400	zygote	current(0)		
qid	vm size	ttb	tty	name
0.	0001AC6E	C1840000	-	
+ flags				
+ relationship				
+ arguments				
+ environment				
+ open files				
+ addresses				
+ code file				

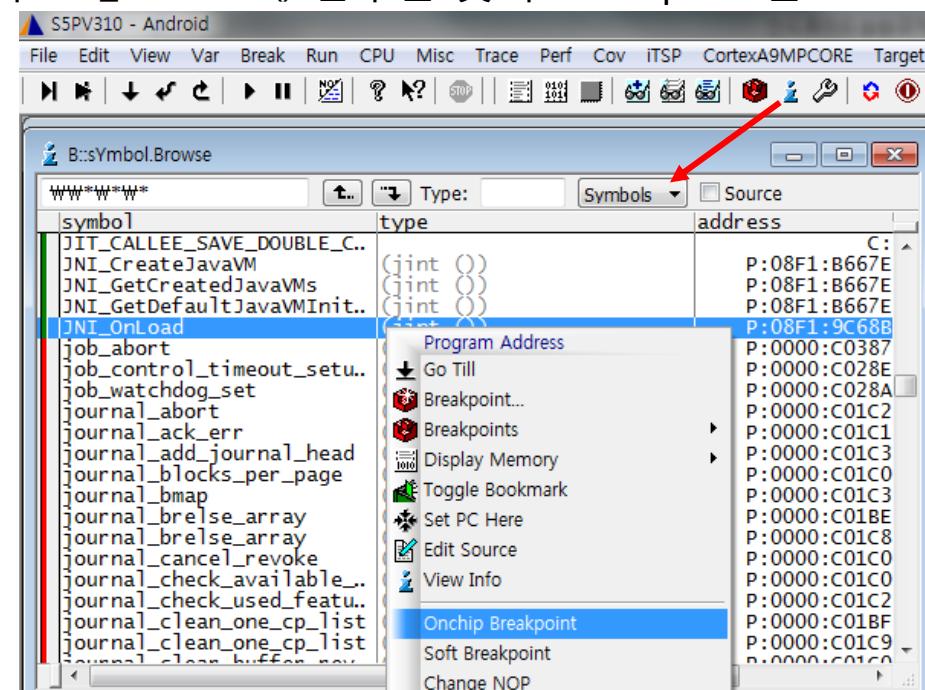
2.13 zygote를 double click 한후 새로 open된 window의 code file 항목을 확장시킨다.

8-4. Shared Library - Dynamic Loading 실습

2.14 libwebcore.so를 찾고 오른쪽 마우스 클릭해서 Load Library Symbols 메뉴를 통해 symbol 정보를 load한다.

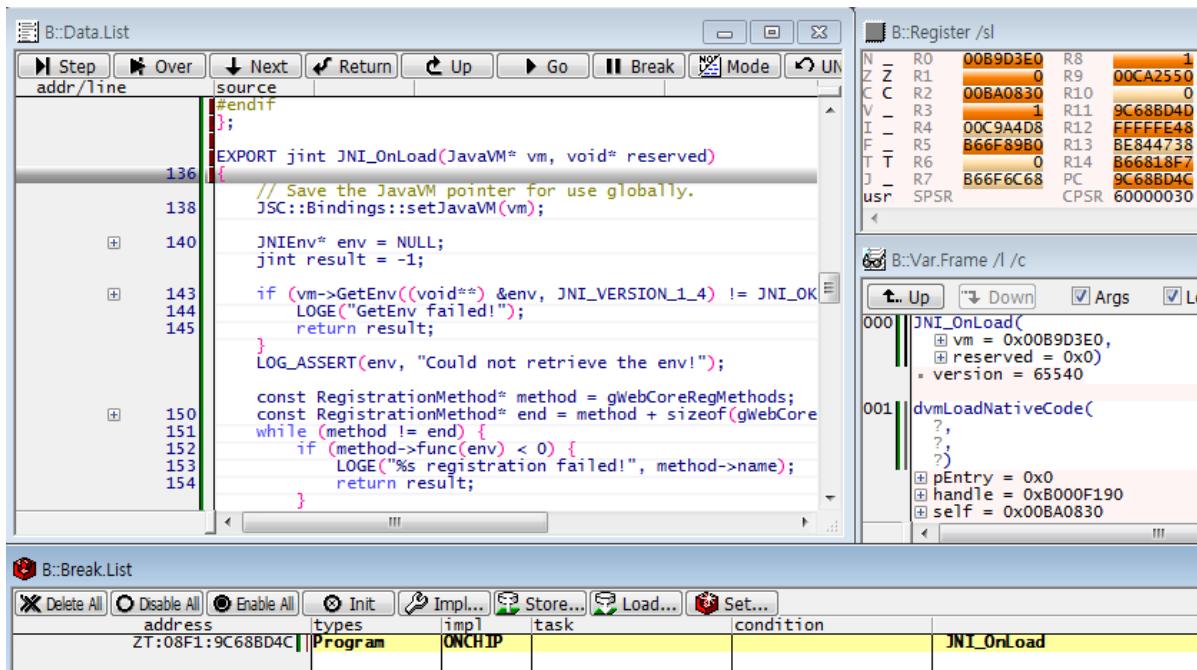


2.15 symbol load후 symbol browser에서 JNI_OnLoad() 함수를 찾아 breakpoint를 설정한다.



8-4. Shared Library - Dynamic Loading 실습

2.16 breakpoint 설정 후 target을 running하면 breakpoint 를 설정한 함수에 멈추는 것을 확인할 수 있다.



* 특정 application에서만 사용하는 library 혹은 ASLR 방식으로 load되는 library의 경우에는 libdvm.so의 `dvmloadnativecode()` 함수를 이용해 해당 library가 load되자마자 symbol을 load해 디버깅하는 방법을 이용해야 한다.

8-4. Shared Library - Dynamic Loading 실습

3. Shared Library – Dynamic Loading Debugging – 실습 3

HAL쪽 Library를 디버깅하기 위해서는 JNI와 마찬가지로 Load되는 함수에 Break Point를 설정한 후 원하는 Symbol을 Load할 수 있다.

HAL Library는 libhardware.so의 Load함수를 통해 확인할 수 있으며 해당 Library는 모든 process에 포함된다. 아래 내용을 통해 디버깅 방법을 확인해 보자.

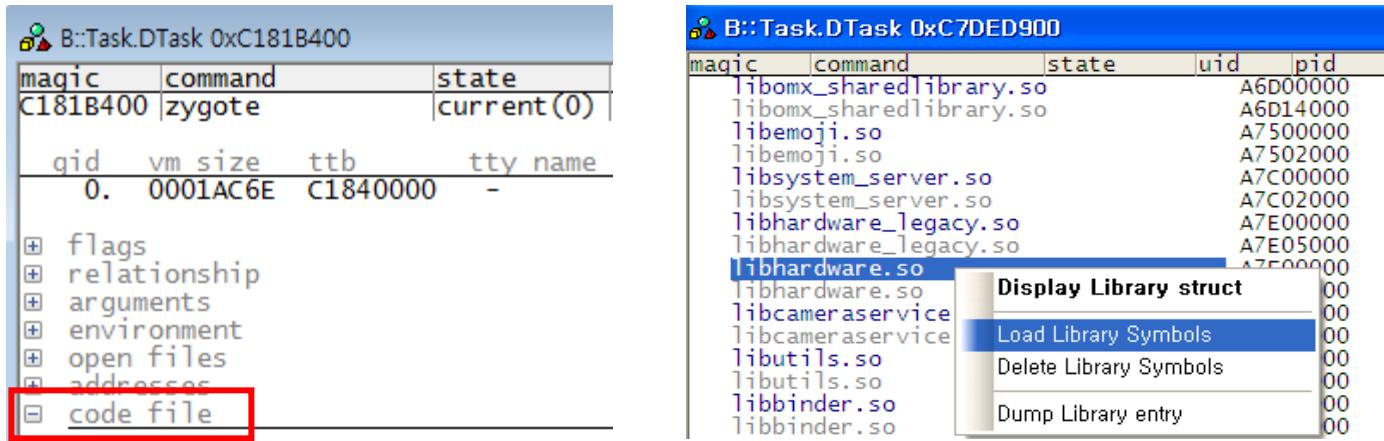
3.1 Linux->Display Processes 메뉴를 Open 한 후 zygote를 Double-click 한다.

The screenshot shows the Android Device Monitor interface. On the left, there is a navigation bar with 'Linux' selected, followed by 'Window' and 'Help'. Below this is a 'Display Processes' menu with several options: 'Display ps-like', 'Display Tasks', 'Display Modules', 'Display File System', 'Process Debugging', 'Module Debugging', 'Library Debugging', 'Autoloader', 'Linux Terminal', 'Configure Terminal...', and 'Help Linux Awareness'. To the right of this menu is a table titled 'B::TASK.Process'. The table has columns: 'magic', 'command', '#thr', and 'state'. The data in the table is as follows:

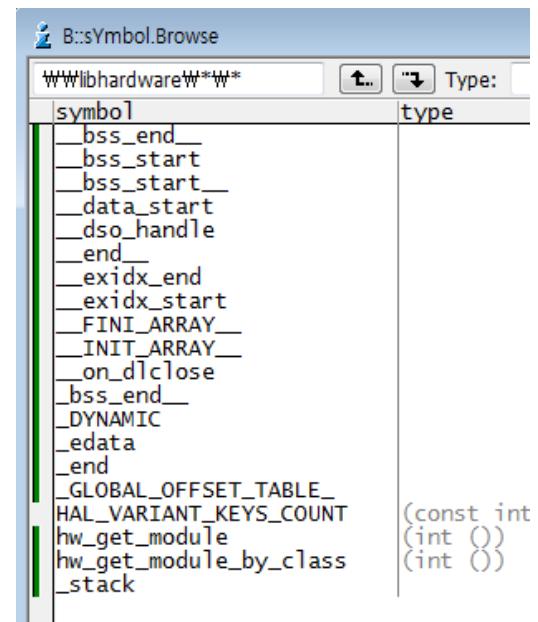
magic	command	#thr	state
C07A0108	swapper/0	44.	running
ED00A000	init	-	sleeping
ED2CA400	ueventd	-	sleeping
C18C1400	sh	-	sleeping
C18C0800	servicemanager	-	sleeping
C18C0400	vold	3.	sleeping
C18C0000	netd	6.	sleeping
C18C0C00	debuggerd	-	sleeping
C181BC00	rild	3.	sleeping
C181B800	surfaceflinger	7.	running
C181B400	zygote	4.	current
C181B000	drmserver	2.	sleeping
C181AC00	mediaserver	5.	sleeping
C181A800	dbus-daemon	-	sleeping
C181A400	installd	-	sleeping
C181A000		-	-

8-4. Shared Library - Dynamic Loading 실습

3.2 Code Files부분을 확장시켜 libhardware.so Symbol을 Load 한다.



3.3 libhardware.so Symbol에서 Load함수에
Break Point를 설정해야 하지만 static 함수의
optimization으로 인해 symbol이 존재하지 않는다.

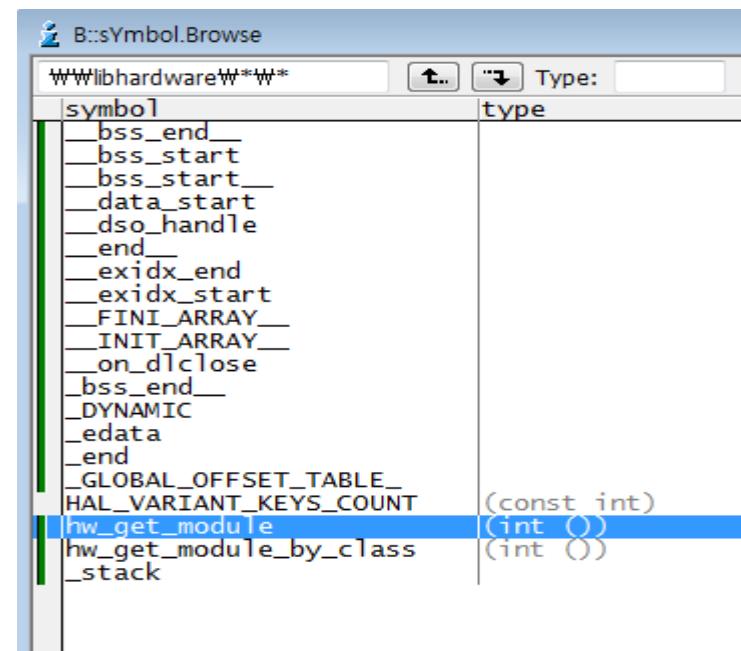


8-4. Shared Library - Dynamic Loading 실습

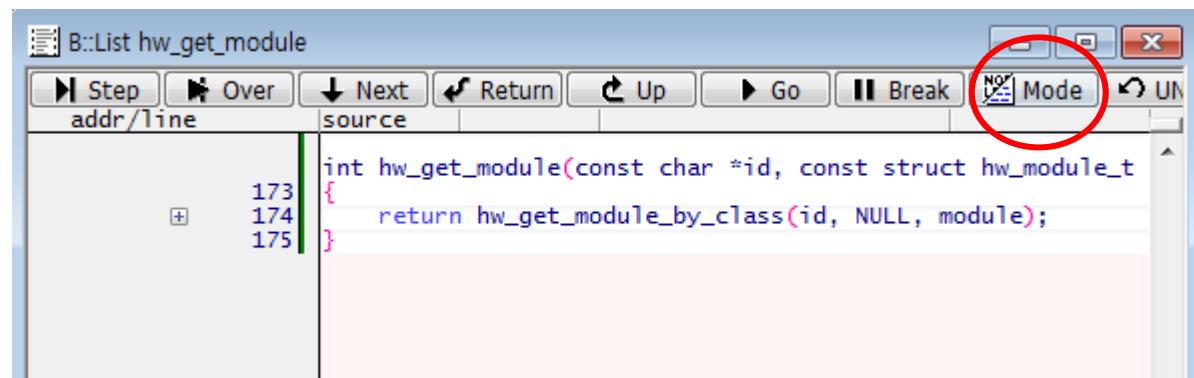
3.4 이런 이유로 load 함수가 포함된 symbol

정보를 통해 data.list를 연 다음 load 함수를
직접 찾아야 한다.

여기서는 hw_get_module을 더블 클릭해서
찾아보자.

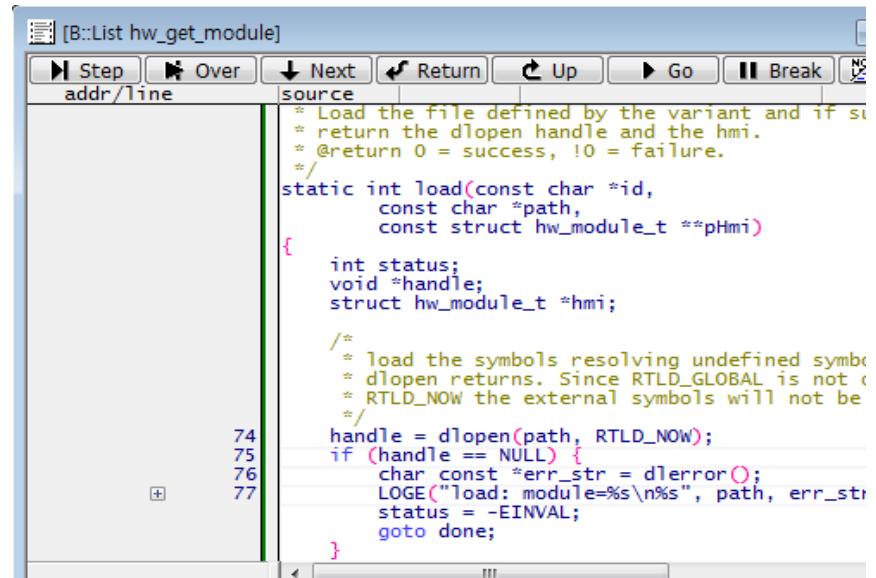


3.5 data.list가 열리면 mode 버튼을 눌러 C소스만 보이게 한 후 위로 scroll해서 load함수를 찾자.



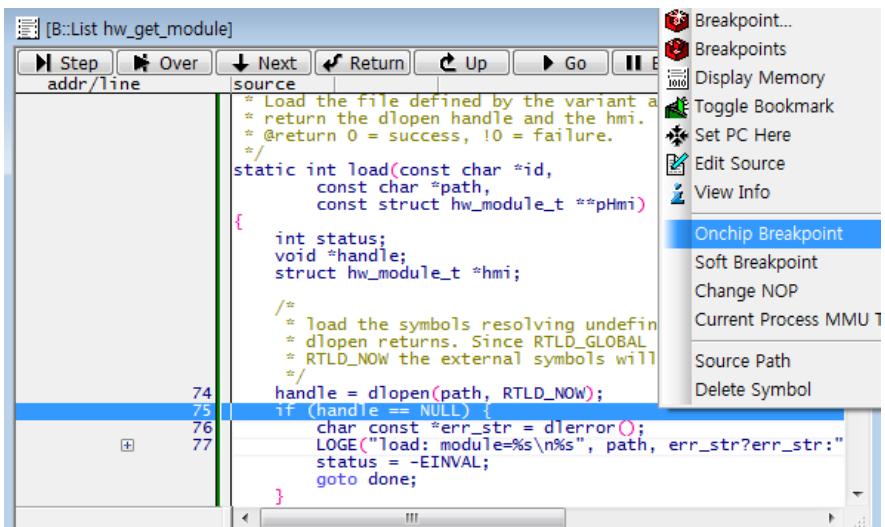
8-4. Shared Library - Dynamic Loading 실습

3.6 scroll을 위로 올리면 load함수를 찾을 수 있다.



```
[B::List hw_get_module]
addr/line source
74     * Load the file defined by the variant and if su
75     * return the dlopen handle and the hmi.
76     * @return 0 = success, !0 = failure.
77     */
static int load(const char *id,
               const char *path,
               const struct hw_module_t **phMi)
{
    int status;
    void *handle;
    struct hw_module_t *hmi;
    /*
     * load the symbols resolving undefined symbols
     * dlopen returns. Since RTLD_GLOBAL is not used
     * RTLD_NOW the external symbols will not be
     */
    handle = dlopen(path, RTLD_NOW);
    if (handle == NULL) {
        char const *err_str = dlerror();
        LOGE("load: module=%s\n%s", path, err_str);
        status = -EINVAL;
        goto done;
    }
```

3.7 load 함수에 breakpoint를 설정한 후 target을 running하면 멈추게 된다.
멈추기 위해서 특정 application(예를 들어 LEDAct application)을 실행해야 하는 경우가 생길 수 있다.

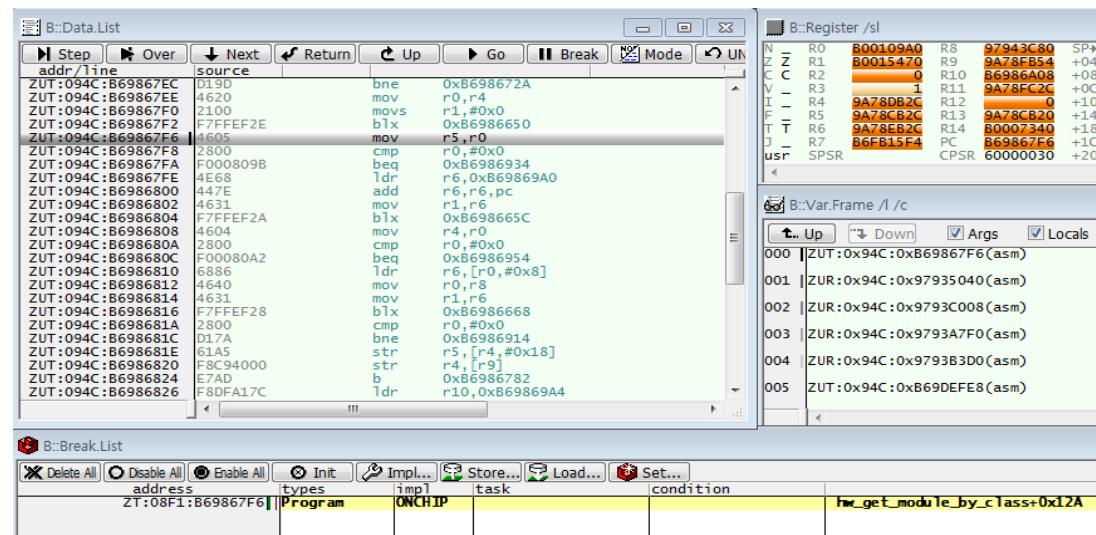


```
[B::List hw_get_module]
addr/line source
74     * Load the file defined by the variant and if su
75     * return the dlopen handle and the hmi.
76     * @return 0 = success, !0 = failure.
77     */
static int load(const char *id,
               const char *path,
               const struct hw_module_t **phMi)
{
    int status;
    void *handle;
    struct hw_module_t *hmi;
    /*
     * load the symbols resolving undefined symbols
     * dlopen returns. Since RTLD_GLOBAL is not used
     * RTLD_NOW the external symbols will not be
     */
    handle = dlopen(path, RTLD_NOW);
    if (handle == NULL) {
        char const *err_str = dlerror();
        LOGE("load: module=%s\n%s", path, err_str);
        status = -EINVAL;
        goto done;
    }
```

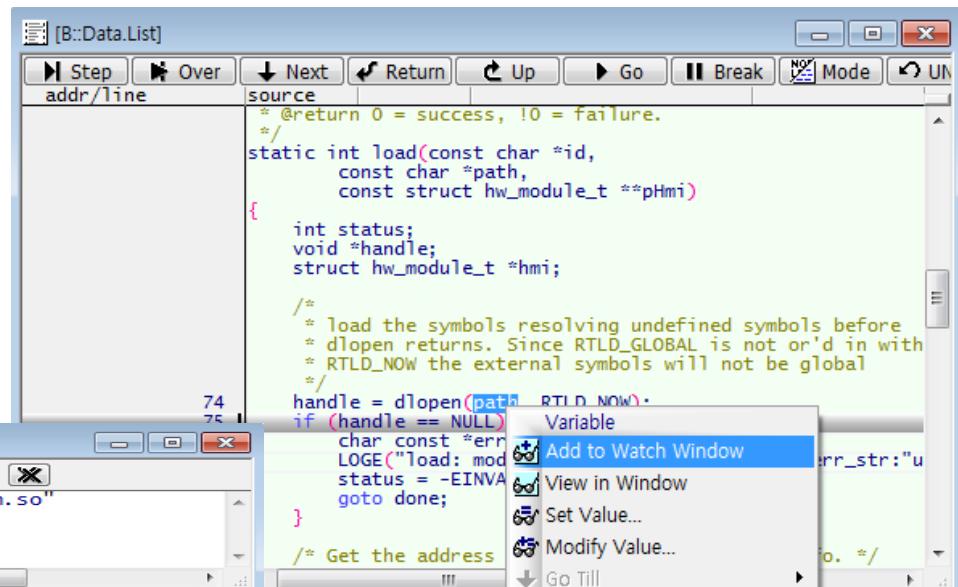
- Breakpoint...
- Breakpoints
- Display Memory
- Toggle Bookmark
- Set PC Here
- Edit Source
- View Info
- Onchip Breakpoint**
- Soft Breakpoint
- Change NOP
- Current Process MMU 1
- Source Path
- Delete Symbol

8-4. Shared Library - Dynamic Loading 실습

3.8 멈쳤을 때 마찬가지로 path 변수를 확인해 보자.



3.9 breakpoint에 의해 멈추고 symbol이 안보이면 아이콘을 선택하자. 그러면 symbol이 다시 갱신해서 load될 것이다.
Symbol이 load되면 인자 값인 path를 다시 확인해 보자.



8-4. Shared Library - Dynamic Loading 실습

만약 path정보가 보이지 않는다면, local 변수가 register로 연산되기 때문에 register를 확인해야 한다.

The screenshot shows the assembly code and registers window of a debugger. The assembly code window displays the following code:

```
void *handle;
struct hw_module_t *hmi;

/*
 * Load the symbols resolving undefined symbols before
 * dlopen returns. Since RTLD_GLOBAL is not or'd in with
 * RTLD_NOW the external symbols will not be global
 */

74 ZUT:09D1:B69867EE handle = dlopen(path, RTLD_NOW);
620 ZUT:09D1:B69867F0 mov r0,r4
Z100 ZUT:09D1:B69867F2 movs r1,#0x0
F7FFF2E ZUT:09D1:B69867F4 b1x 0xB6986650
75 if (handle == NULL) {
4605 ZUT:09D1:B69867F6 mov r5,r0
2800 ZUT:09D1:B69867F8 cmp r0,#0x0
```

	N	R0	B000F3C0	R8	B6A13E5C	
Z	Z	R1	B0015470	R9	BE8443C0	
C	C	R2	0	R10	B6986A08	
V	V	R3	1	R11	BE844578	
I	I	R4	BE84239C	R12	0	+10 00006E65
F	F	R5	BE84139C	R13	BE841390	+14 00000000
T	T	R6	BE84339C	R14	B0007340	+18 00000000
J	J	R7	B6FB15F4	PC	B69867F6	+1C 00000000
usr	usr	SPSR		CPSR	60000030	+20 00000000

코드 정보를 보면 path는 r4를 가리키고 있기 때문에 register window에서 r4값을 확인하면 된다. 해당 값을 변수로 보기 위해서는 다음과 같이 입력하면 된다.
Path의 type은 char *이다.

The screenshot shows the assembly code and registers window of a debugger. The assembly code window displays the following code:

```
void *handle = dlopen(path, RTLD_NOW);
```

The registers window shows the following register values:

	N	R0	B000F3C0	R8	B6A13E5C	
Z	Z	R1	B0015470	R9	BE8443C0	+04 BE84139C
C	C	R2	0	R10	B6986A08	+08 00000000
V	V	R3	1	R11	BE844578	+0C 6769726F
I	I	R4	BE84239C	R12	0	+10 00006E65
F	F	R5	BE84139C	R13	BE841390	+14 00000000
T	T	R6	BE84339C	R14	B0007340	+18 00000000
J	J	R7	B6FB15F4	PC	B69867F6	+1C 00000000
usr	usr	SPSR		CPSR	60000030	+20 00000000

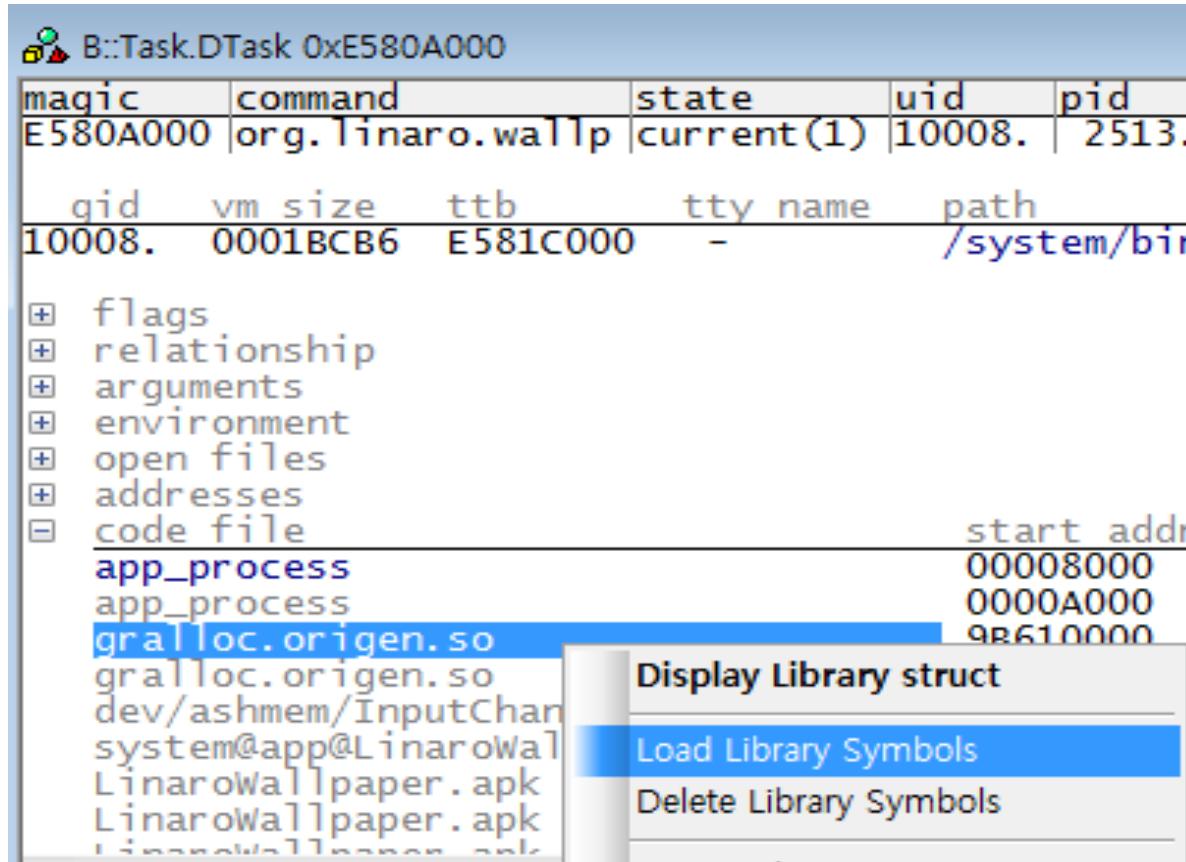
The variable window shows the value of the variable v:

```
B::v.v (char*)0xBE84239C
```

The value of the variable v is 0xBE84239C, which corresponds to the address of the shared library "gralloc.origen.so".

8-4. Shared Library - Dynamic Loading 실습

위와 같이 확인하면 hal library인 gralloc.origen.so library를 load할 수 있다.



* HAL library도 마찬가지로 libdvm과 마찬가지로 load 하는 함수인 Load() 함수가 존재한다. 방식은 위에서 확인한 것처럼 JNI방식과 동일한 것을 확인할 수 있다.

9. Kernel Module

Linux에서는 Kernel에 Built-In되지 않고, 원하는 시점에 동적으로 Load할 수 있는 Module 이 있어 이를 디버깅하는 방법을 학습하도록 합니다

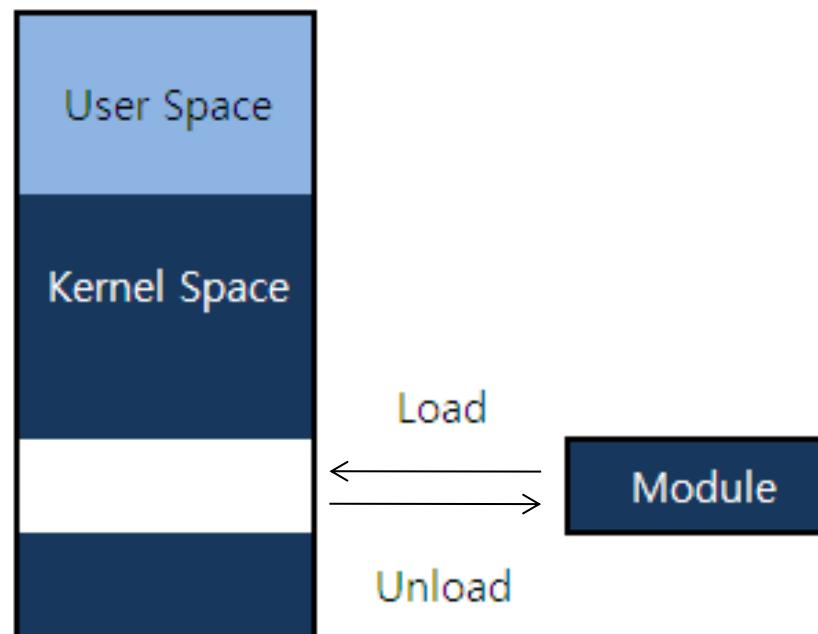
1. Kernel Module의 이해
2. Kernel Module의 init 부터 디버깅
3. Kernel Module 디버깅

9-1. Kernel Module의 이해

Linux 커널에 동적으로 프로그램 코드를 적재하는 LKM에 대해 학습합니다

Device Driver는 H/W를 다루기 위한 Kernel 코드의 종류이며, Linux에서는 동적으로 Load할 수 있는 Kernel Object 형태인 Module로 제공되기도 합니다

- 통상 Device Driver를 Module 방식으로 작성하기 때문에 같은 의미로 통용됩니다
- 부팅 중 Kernel에 의해 Device Driver가 등록되는 경우가 많습니다
- 일부 동적으로 Memory에 Load하는 Module도 존재합니다(eg. Wifi, GPS)
- 동적으로 동작되는 Module은 insmod/rmmod/lsmod 등의 명령을 사용합니다

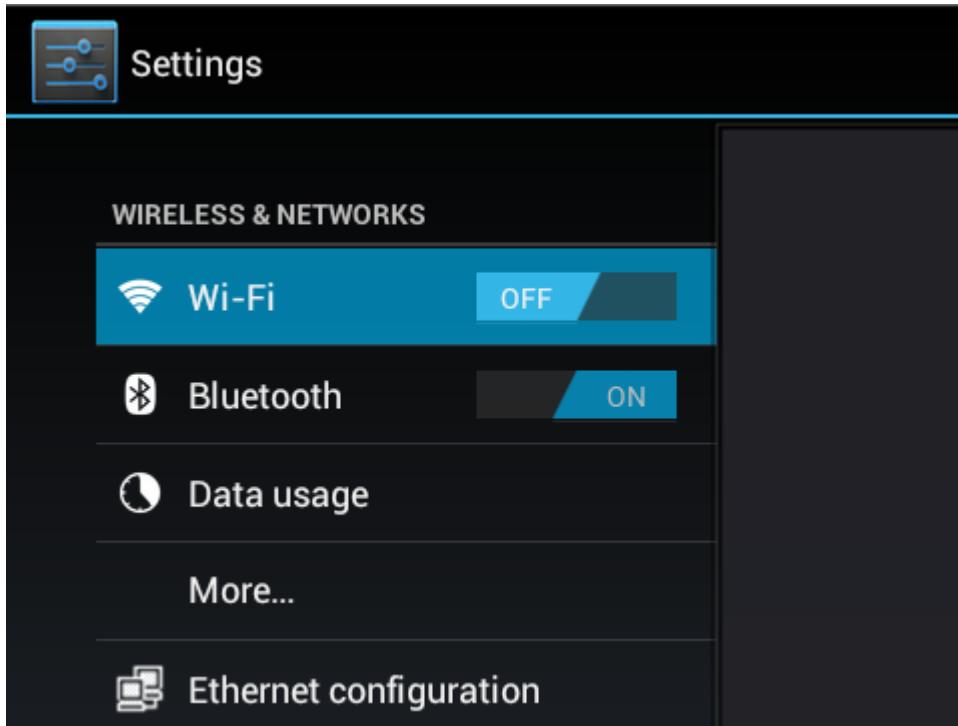


9-2. Kernel Module의 init 부터 디버깅

실습을 통해 module의 init 함수 디버깅을 이해합니다

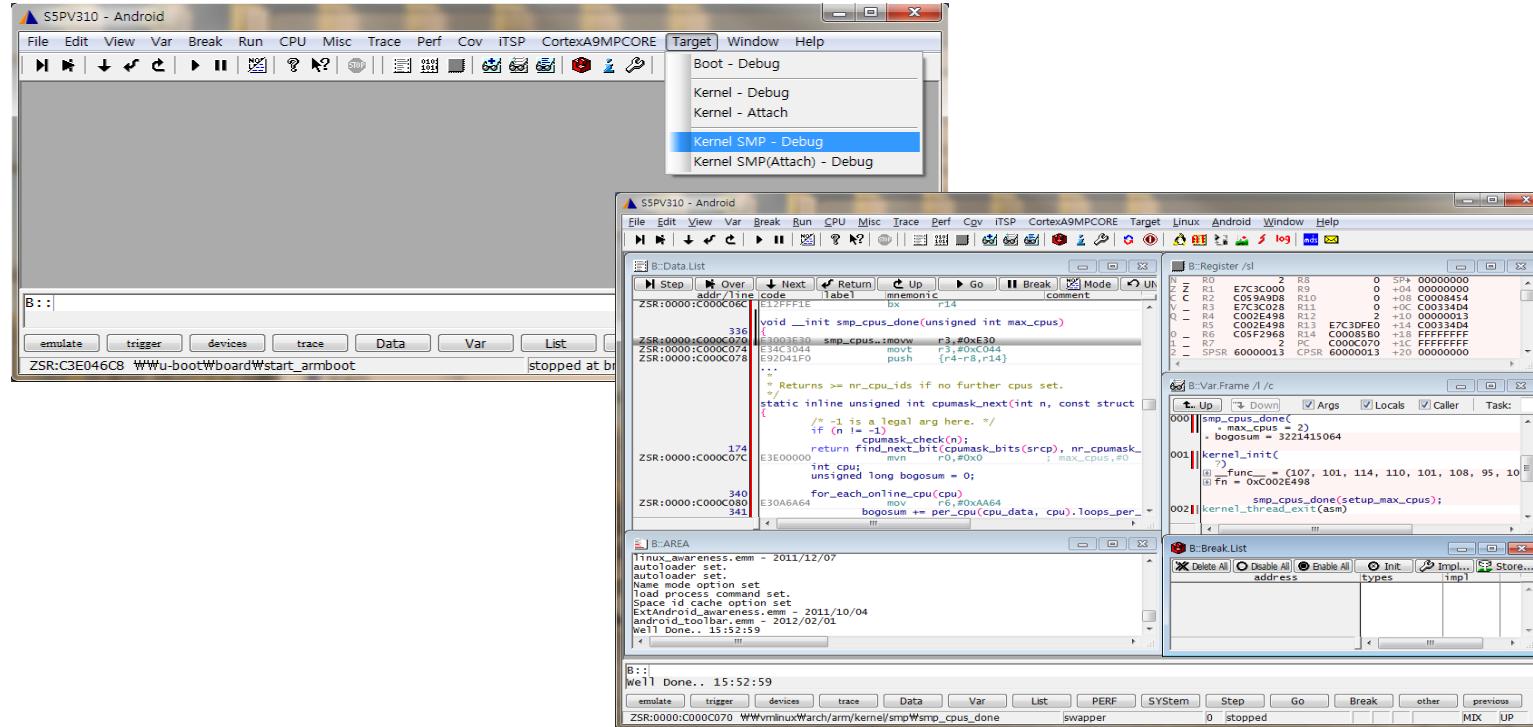
1. Wifi module debugging

setting에서 wifi를 on할 때 load되는 module의 init 부터 디버깅하자.

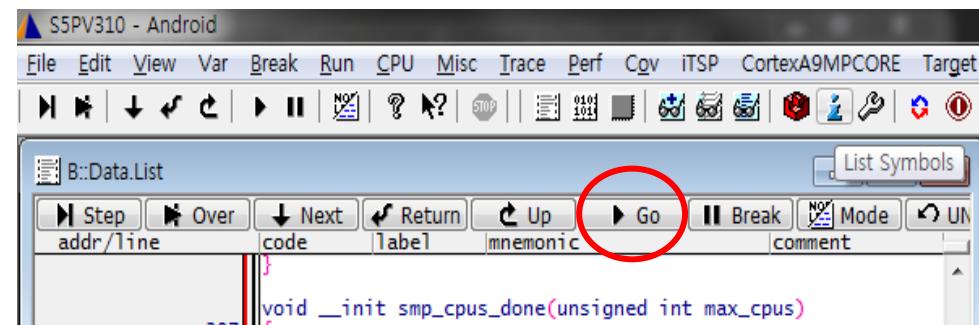


9-2. Kernel Module의 init 부터 디버깅

1.1 kernel SMP debugging 환경을 만든다.



1.2 go 버튼을 눌러 부팅을 시킨다.



9-2. Kernel Module의 init 부터 디버깅

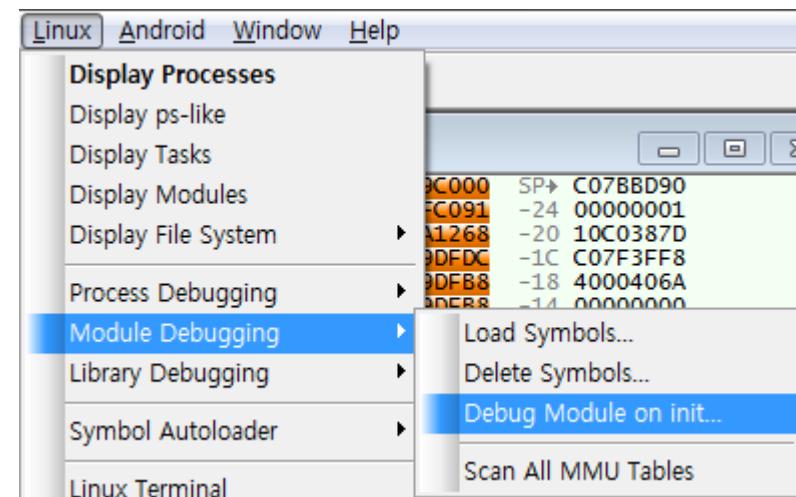
1.3 부팅이 완료된 후 setting에 들어가 wifi를 on할 준비를 한다.

만약 wifi가 on이 되어 있다면 module이 이미 load되어 있기 때문에 init함수를 디버깅 할 수 없다. Wifi가 on이 되어 있다면 off로 변경한다.

1.4 wifi가 on이 되기 전에 TRACE32로
디버깅에 대한 정보를 셋팅한다.

Linux → Module debugging →

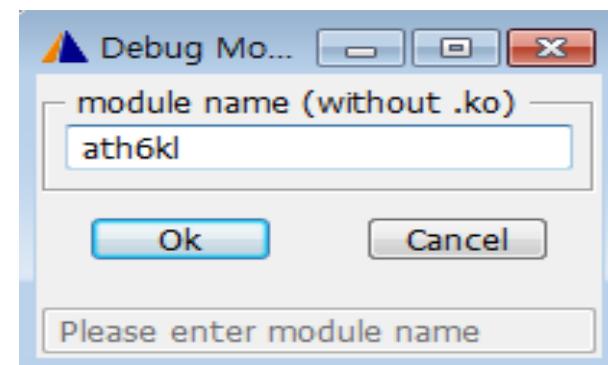
Debug module on Init 메뉴를 누른다.



1.5 열려진 window에서 디버깅할 module의 이름을 입력하고 ok를 누른다.

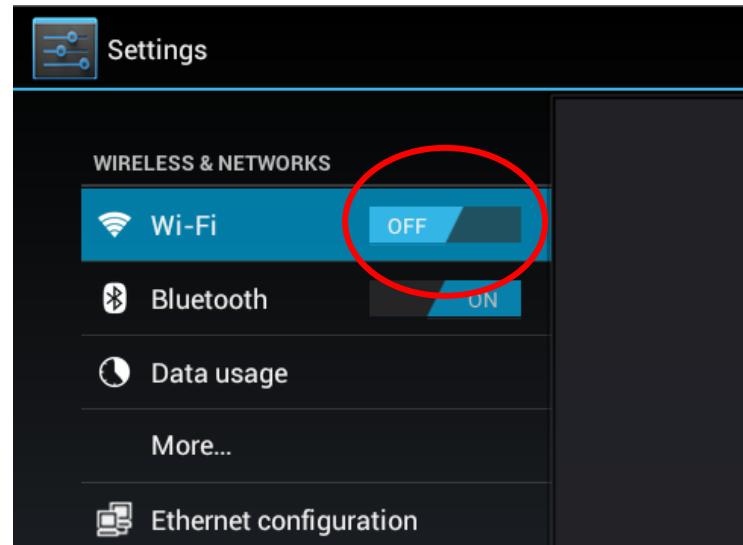
디버깅할 wifi module은 ath6kl.ko 이다.

이름을 입력할 때 확장자인 .ko는 제외하고 입력한다.

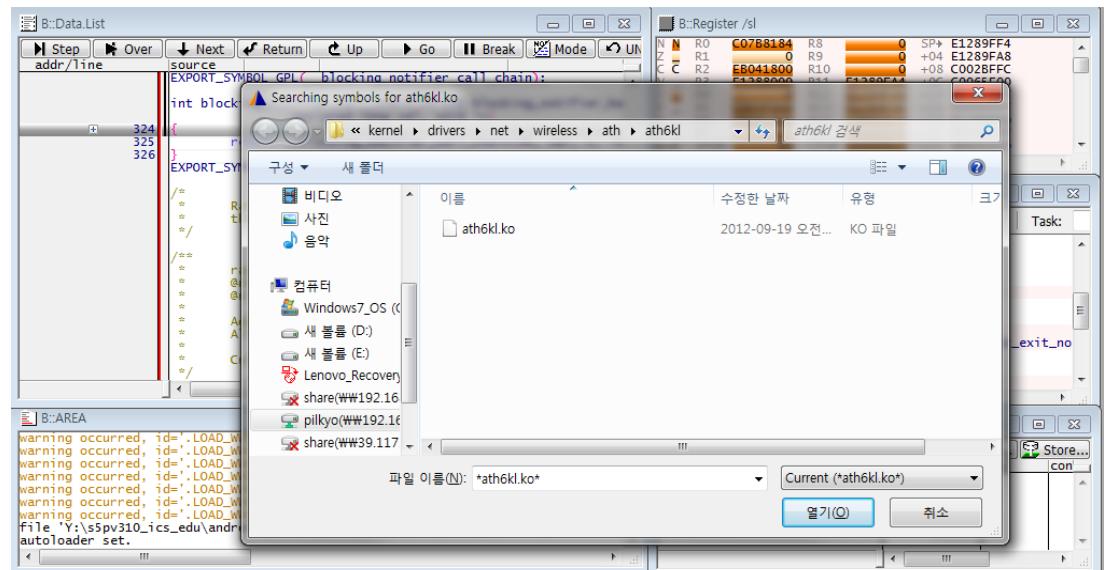


9-2. Kernel Module의 init 부터 디버깅

1.6 ok를 입력하면 target0| 자동으로 running이 된다. Wifi를 on해보자.

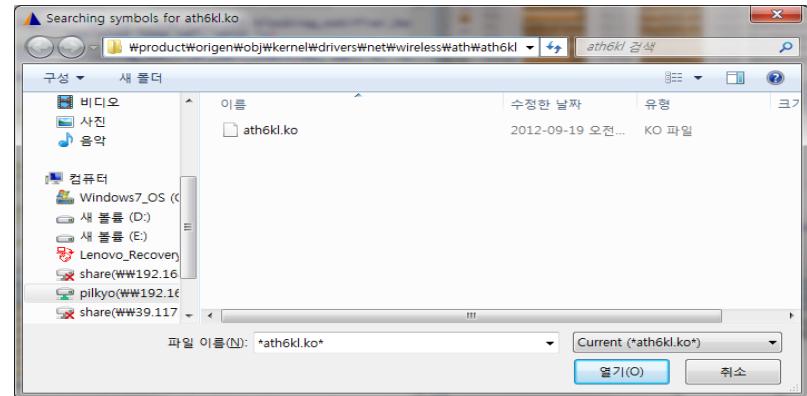


1.7 on을 누르면 TRACE32에
의해서 target이 멈추고
symbol을 찾는
window가 뜬다.



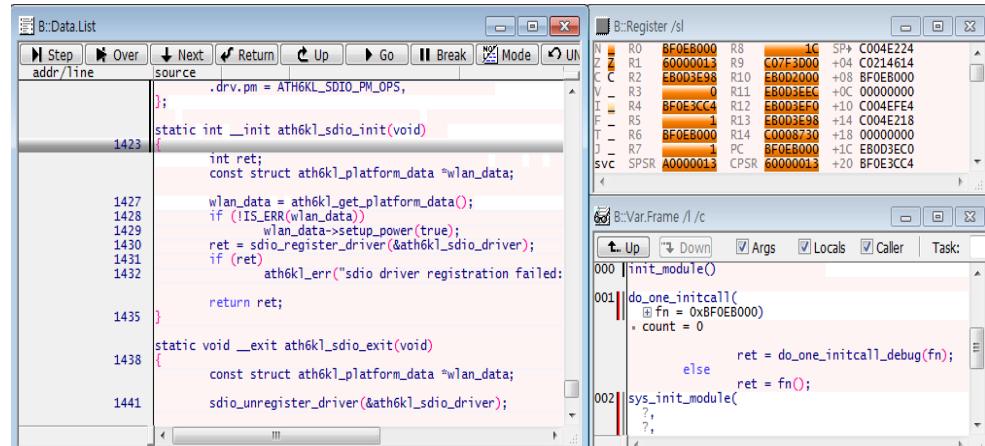
9-2. Kernel Module의 init 부터 디버깅

1.8 아래 경로에 들어가 해당 module의 symbol을 찾아 선택한다.



Android\out\target\product\origen\obj\kernel\drivers\net\wireless\ath\ath6kl\ath6kl.ko

1.9 symbol을 선택하면 TRACE32가 해당 symbol을 load하고 module의 init함수를 찾아 멈춘다.
해당 module의 init함수는 ath6kl_sdio_init() 이다.



9-2. Kernel Module의 init 부터 디버깅

2. T32_led module debugging

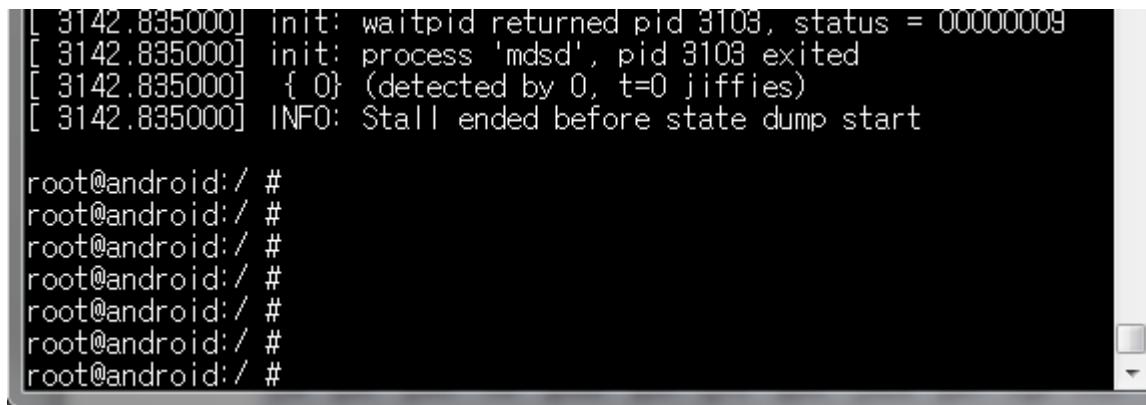
동일한 방법으로 t32_led.ko 를 디버깅해보자.

해당 모듈을 loading하는 별도의 기능이 없기 때문에 terminal을 통해서 수동으로 로드한다.(terminal을 사용해야 하므로 serial cable을 연결하자. Adb를 이용할 때는 usb를 연결해 adb로 접속한다.)

추가로 현재 mdssd가 계속 죽는 메시지가 뜨기 때문에 임시로 해당 문제를 해결하자.
cmm 중 clear_mdssd.cmm을 실행하자. (serial 을 이용할 때만 사용하자.)

```
B:::do clear_mdssd.cmm
```

2.1 target을 부팅한 후에 terminal을 통해 filesystem을 확인하자.



The screenshot shows a terminal window with two sections. The top section displays logcat output from the kernel, showing messages about processes exiting and a stall before a state dump. The bottom section shows a root shell prompt, with several '#' characters indicating commands have been entered but not yet run.

```
[ 3142.835000] init: waitpid returned pid 3103, status = 00000009
[ 3142.835000] init: process 'mdssd', pid 3103 exited
[ 3142.835000] { 0} (detected by 0, t=0 jiffies)
[ 3142.835000] INFO: Stall ended before state dump start

root@android: / #
```

9-2. Kernel Module의 init 부터 디버깅

2.2 filesystem 내의 경로 중
/system/modules 로 이동하자

```
COM24:115200baud - Tera Term VT
File Edit Setup Control Window Help
root
sbin
sdcard
sys
system
ueventd.goldfish.rc
ueventd.origen.rc
ueventd.rc
vendor
root@android:/ # cd system/
root@android:/ # cd system/

root@android:/system # cd modules/
root@android:/system # cd modules/

root@android:/system/modules # ls
ansi_cprng.ko
ath6kl.ko
gator.ko
gspca_main.ko
hid-logitech-dj.ko
scsi_wait_scan.ko
t32_led.ko
root@android:/system/modules # pwd
/system/modules
root@android:/system/modules # cd /system/modules
```

2.3 ls 명령어로 디버깅 해야 하는
t32_led.ko가 있는지 확인한다.

```
COM24:115200baud - Tera Term VT
File Edit Setup Control Window Help
vendor
root@android:/ # cd system/
root@android:/ # cd system/

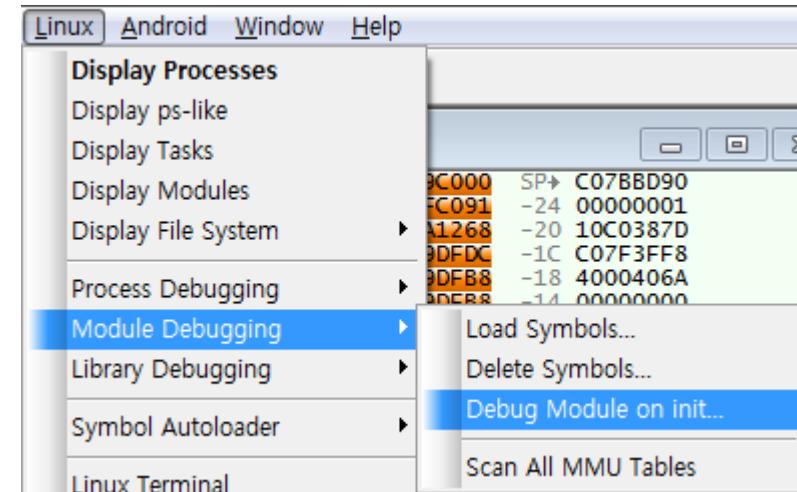
root@android:/system # cd modules/
root@android:/system # cd modules/

root@android:/system/modules # ls
ansi_cprng.ko
ath6kl.ko
gator.ko
gspca_main.ko
hid-logitech-dj.ko
scsi_wait_scan.ko
t32_led.ko
root@android:/system/modules # pwd
/system/modules
root@android:/system/modules # ls
ansi_cprng.ko
ath6kl.ko
gator.ko
gspca_main.ko
hid-logitech-dj.ko
scsi_wait_scan.ko
t32_led.ko
root@android:/system/modules #
```

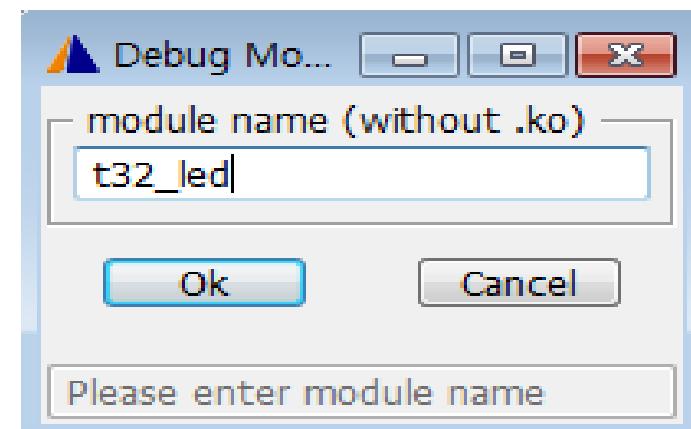
9-2. Kernel Module의 init 부터 디버깅

2.4 해당 module을 load하기 전 TRACE32에서 module debugging 셋팅을 진행하자.

Linux → module debugging → debug module on init 메뉴를 선택하자.

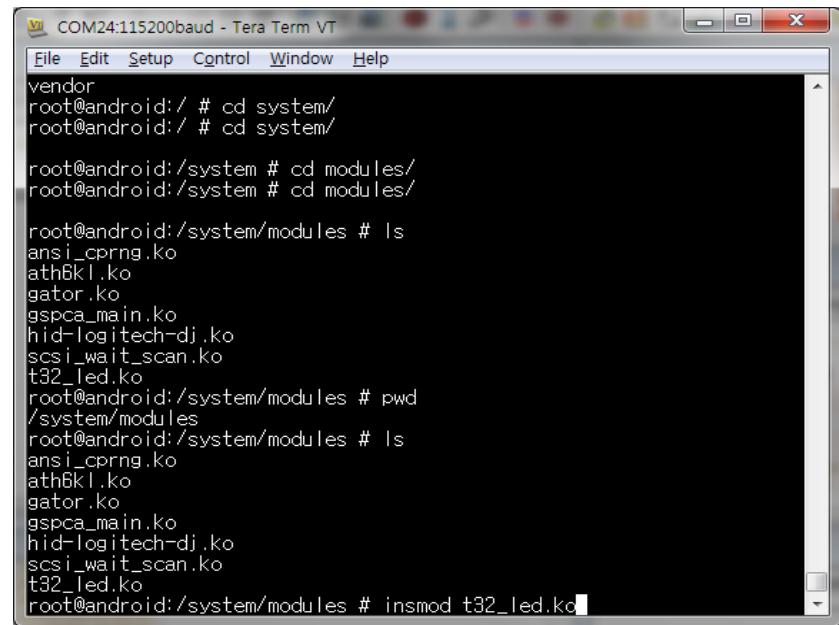


2.5 window가 열리면 t32_led 를 입력하고 ok를 누르자.
확장자인 .ko는 입력하지 않는다.



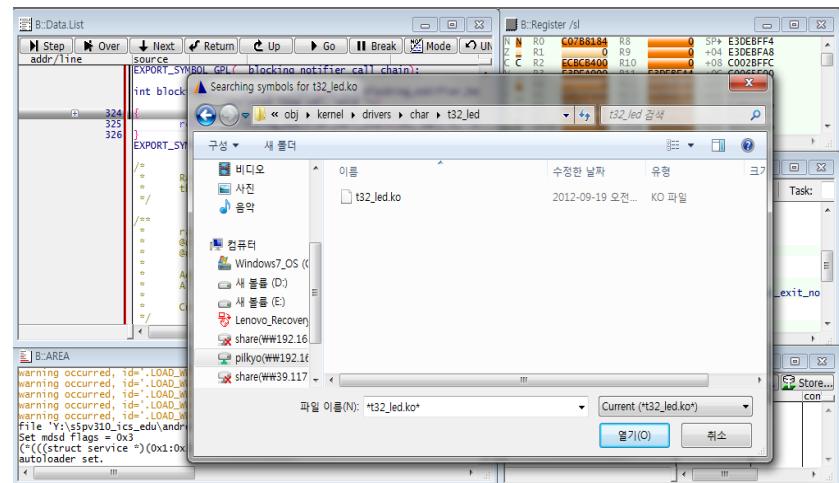
9-2. Kernel Module의 init 부터 디버깅

2.6 target0| running이 되면,
terminal에서 해당 module을
load하자. loading하는 명령은
insmod t32_led.ko를
입력하면 된다.



```
File Edit Setup Control Window Help
vendor
root@android:/ # cd system/
root@android:/ # cd system/
root@android:/system # cd modules/
root@android:/system/modules # ls
ansi_cprng.ko
ath6kl.ko
gator.ko
gspca_main.ko
hid-logitech-dj.ko
scsi_wait_scan.ko
t32_led.ko
root@android:/system/modules # pwd
/system/modules
root@android:/system/modules # ls
ansi_cprng.ko
ath6kl.ko
gator.ko
gspca_main.ko
hid-logitech-dj.ko
scsi_wait_scan.ko
t32_led.ko
root@android:/system/modules # insmod t32_led.ko
```

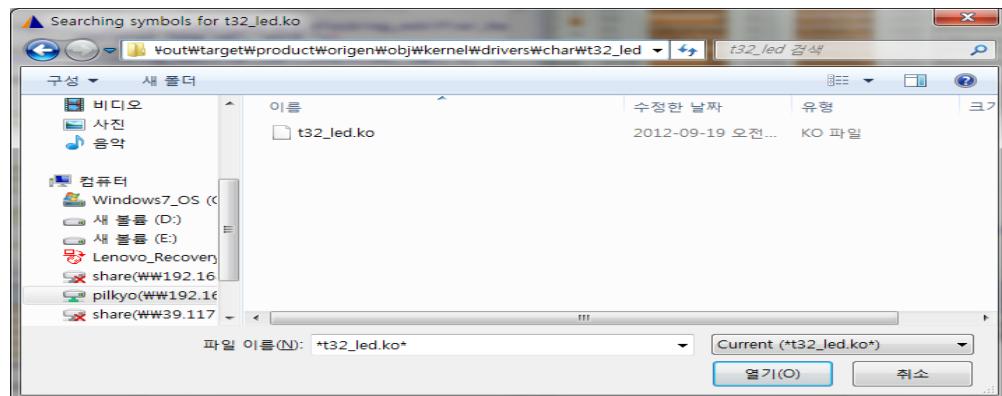
2.7 해당 명령어를 입력하면 TRACE32에서
미리 설정한 부분에 의해 target이
멈추고 symbol을 찾는다.
만약 앞서 setting window의 EDIT에서
해당 경로를 입력했다면 자동으로
symbol이 load된다.



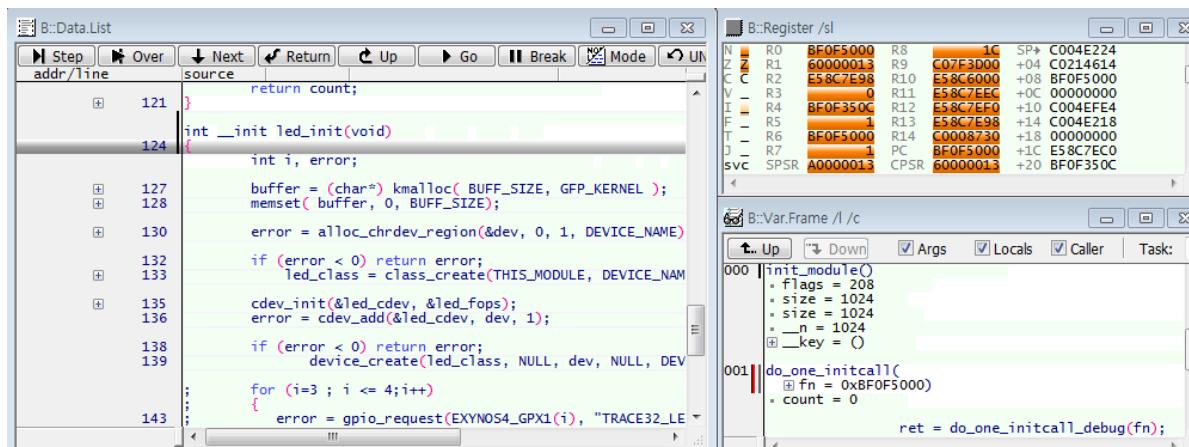
9-2. Kernel Module의 init 부터 디버깅

2.8 해당 symbol을 찾아 선택한다. Symbol은 아래 위치에 있다.

Android\out\target\product\origen\obj\kernel\drivers\char\t32_led\t32_led.ko
해당 파일을 찾아 선택한다.



2.9 파일을 선택하면 TRACE32가 symbol을 load하고 t32_led의 init함수를 찾아 멈춘다.
t32_led.ko의 init 함수는 led_init() 인 것을 확인할 수 있다.



9-3. Kernel Module 디버깅

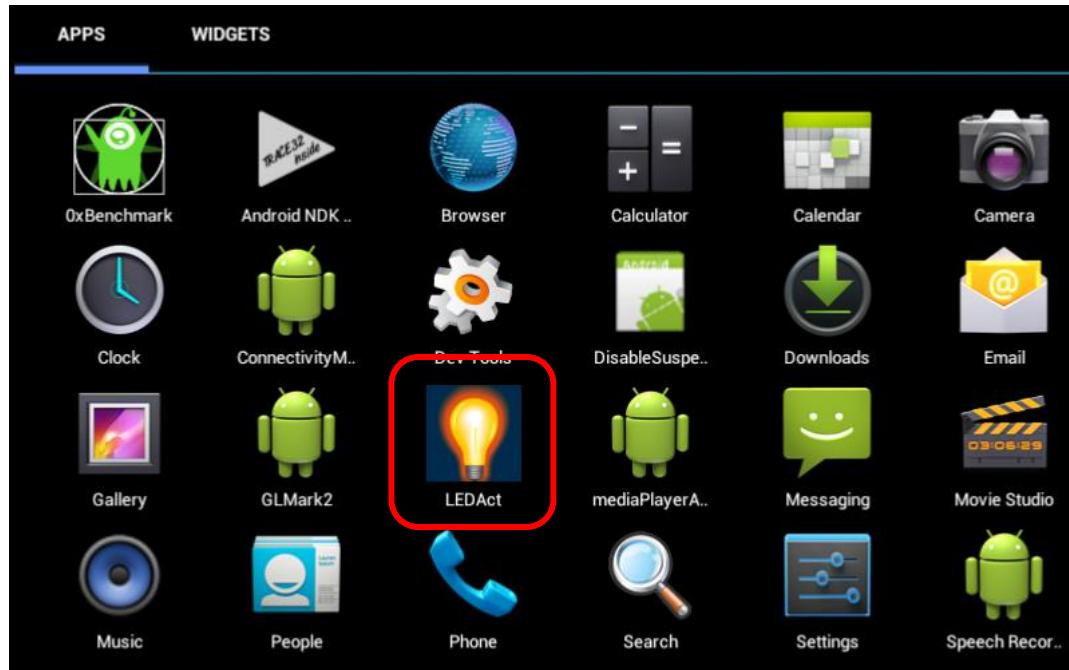
실습을 통해 load된 이후의 module 디버깅을 이해합니다.

1. t32_led module load 후의 debugging

application에서 해당 module을 사용하는 경우 디버깅을 실습해보자.

테스트할 application은 LedAct application이다.

해당 application을 실행할 경우 t32_led.ko 를 사용한다.

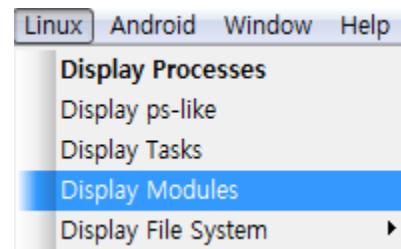


9-3. Kernel Module 디버깅

1.1 먼저 t32_led.ko를 load하자. Loading이 되어 있다면 이 부분은 skip해도 된다.

```
COM24:115200baud - Tera Term VT
File Edit Setup Control Window Help
vendor
root@android:/ # cd system/
root@android:/ # cd system/
root@android:/system # cd modules/
root@android:/system/modules # ls
ansi_cprng.ko
ath6kl.ko
gator.ko
gspca_main.ko
hid-logitech-dj.ko
scsi_wait_scan.ko
t32_led.ko
root@android:/system/modules # pwd
/system/modules
root@android:/system/modules # ls
ansi_cprng.ko
ath6kl.ko
gator.ko
gspca_main.ko
hid-logitech-dj.ko
scsi_wait_scan.ko
t32_led.ko
root@android:/system/modules # insmod t32_led.ko
```

1.2 symbol을 load하기 위해 우측 메뉴를 이용하자.
Linux→display modules 을 선택한다.

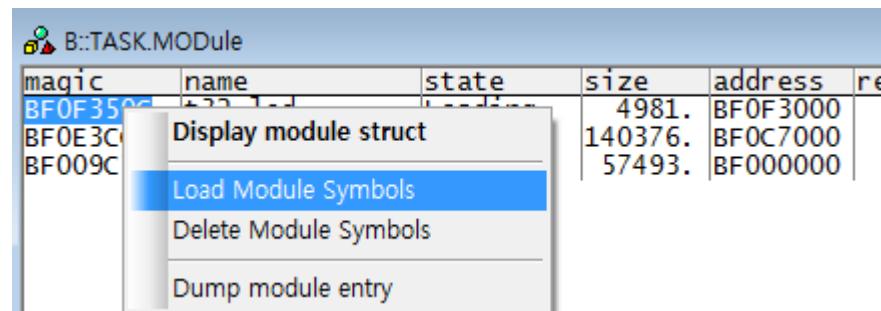


9-3. Kernel Module 디버깅

- 1.3 insmod 를 통해 t32_led.ko를 load했다면 loading된 module 정보를 확인할 수 있다.
만약 해당 module이 보이지 않다면 module이 아직 load되어 있지 않기 때문에
insmod로 해당 module을 load한다.

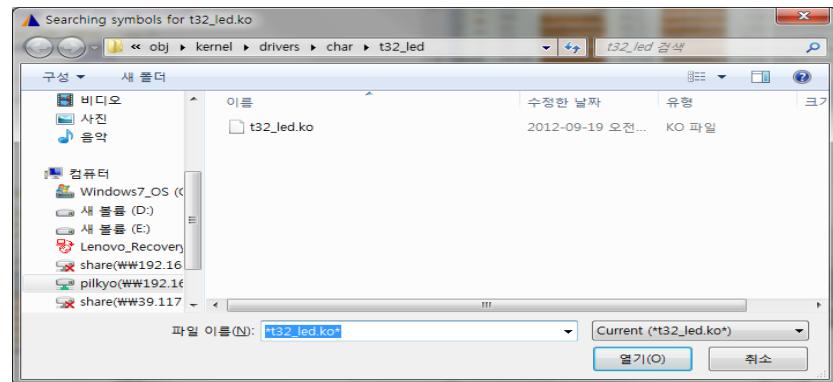
magic	name	state	size	address	refcount	depends
BF0F350C	t32_led	Loading	4981.	BF0F3000		
BF0E3CC4	ath6kl	Live	140376.	BF0C7000		
BF009C74	gator	Live	57493.	BF0000000		

- 1.4 해당 module이 보이면 symbol을 load한다. 모듈을 선택한 후 오른쪽 마우스 클릭해서 Load Module symbols 을 선택한다.



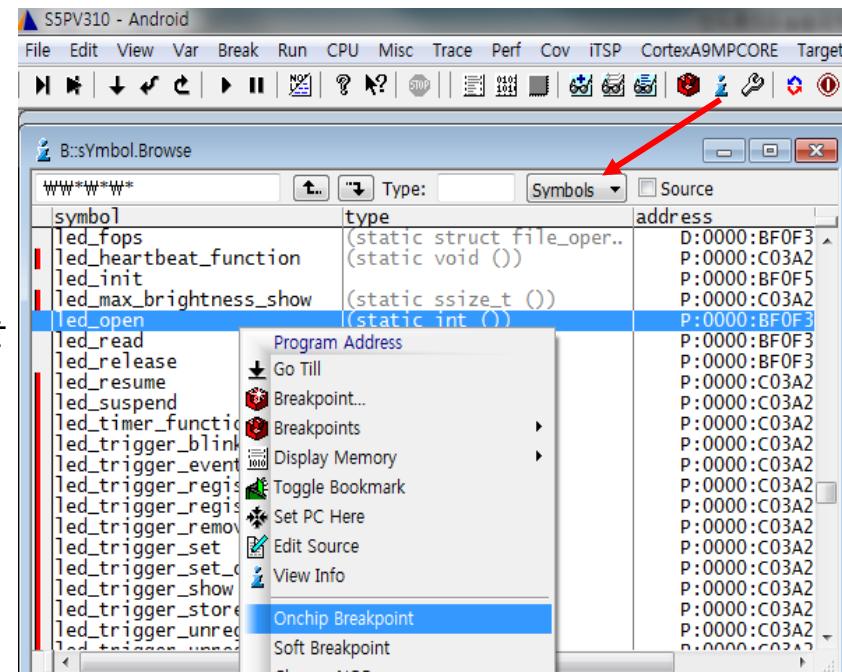
9-3. Kernel Module 디버깅

1.5 해당 menu를 선택하면 symbol을 찾으라는 window가 열리게 되고, 아래 위치에서 해당 symbol을 선택한다.



Android\out\target\product\origin\obj\kernel\drivers\char\t32_led\t32_led.ko

1.6 symbol이 load되고 나면 application을 실행하기 전에 디버깅할 함수를 찾아 breakpoint를 설정한다.
Symbol browser를 열고 led_open 함수를 찾아 오른쪽 마우스 클릭한 후 onchip breakpoint를 설정한다.



9-3. Kernel Module 디버깅

1.7 target을 running하고
해당 application을
실행한다.

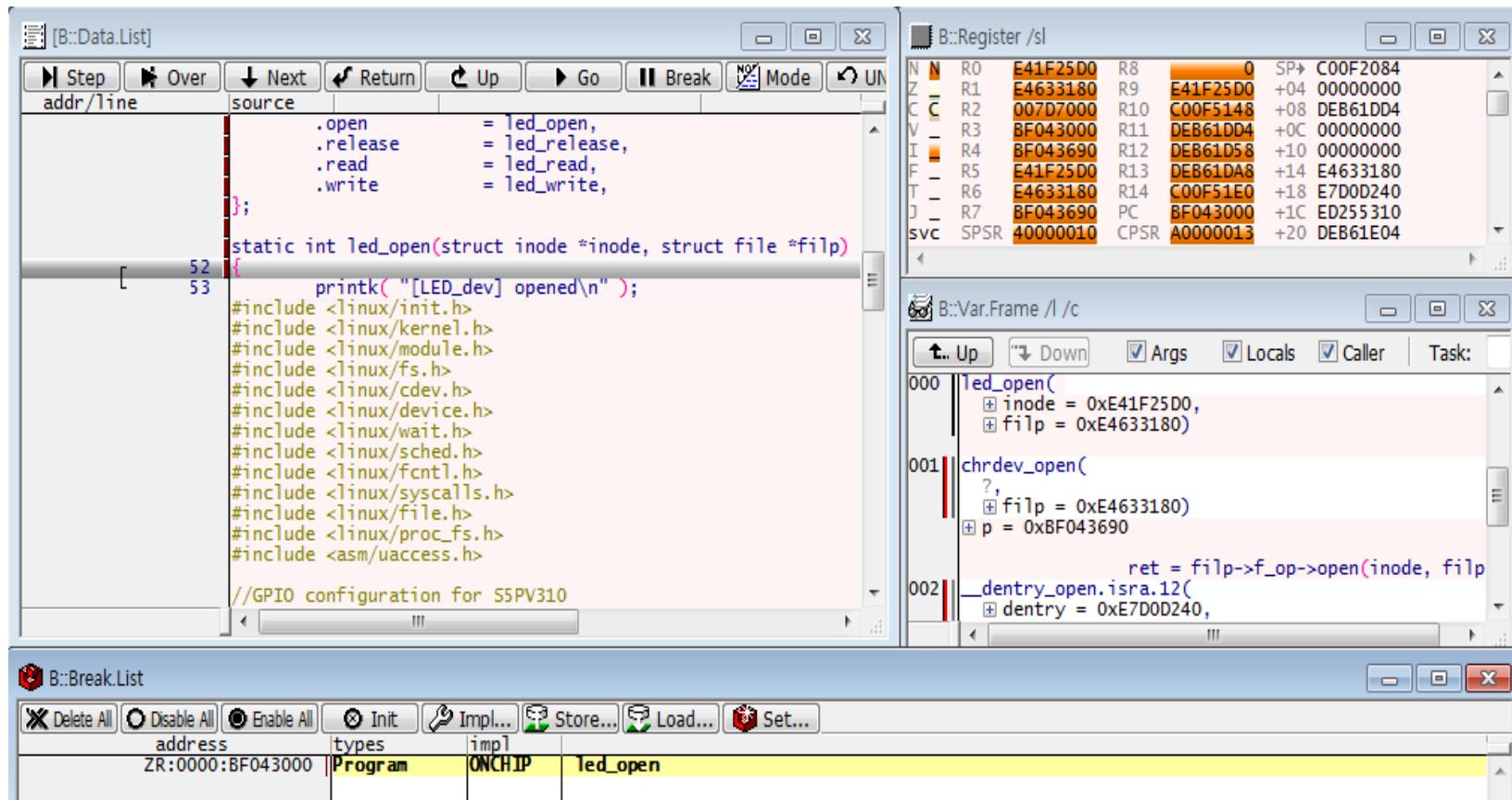


1.8 application 실행 후
LED ON 혹은
LED OFF를 눌러본다.



9-3. Kernel Module 디버깅

1.9 설정된 breakpoint에 멈추는 것을 확인할 수 있다.



10. Segmentation Fault Debugging

지금까지의 Debugging 기능을 이용하여 Exception Debugging을 할 수 있습니다

1. Segmentation Fault 디버깅
2. Android_NDK_Hello 디버깅
3. LEDAct 디버깅

10-1. Segmentation Fault 디버깅

Linux에서는 Exception Vector Table을 통한 방식을 이용하지 않고 Kernel의 정보를 이용해서 Exception Debugging을 합니다

- Linux 기반의 Android Platform에서의 Exception은 발생할 당시의 정보를 가지고 있는 Kernel 정보를 통해 이루어집니다
- Application인 User Space에서 발생한 Exception부터 Kernel Panic 상황까지 디버깅이 가능합니다
- TRACE32 Script를 통해 미리 Breakpoint를 설정한 후 재현을 하면 됩니다
- 자동으로 문제가 되는 Register 정보 복원을 통해 Exception이 발생한 위치를 확인할 수 있습니다

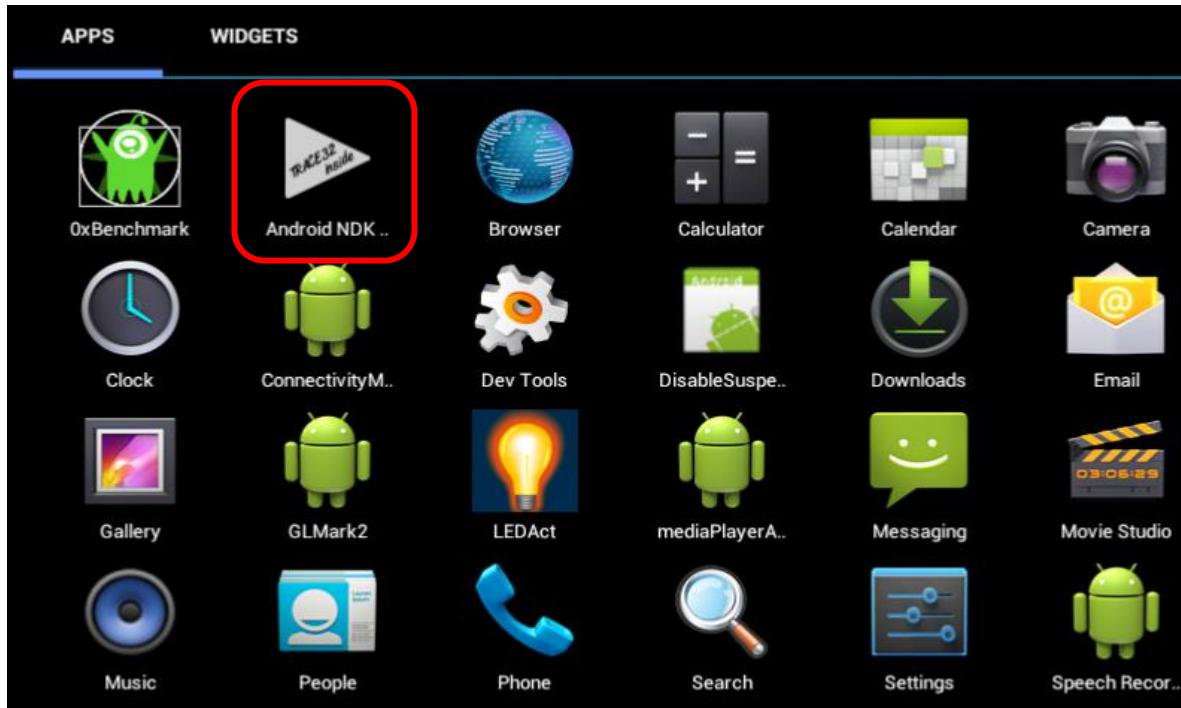
10-2. Android NDK Hello 디버깅

실행하면 바로 종료되는 Android NDK Hello Application의 문제점 원인을 찾아봅니다

1. Exception debugging

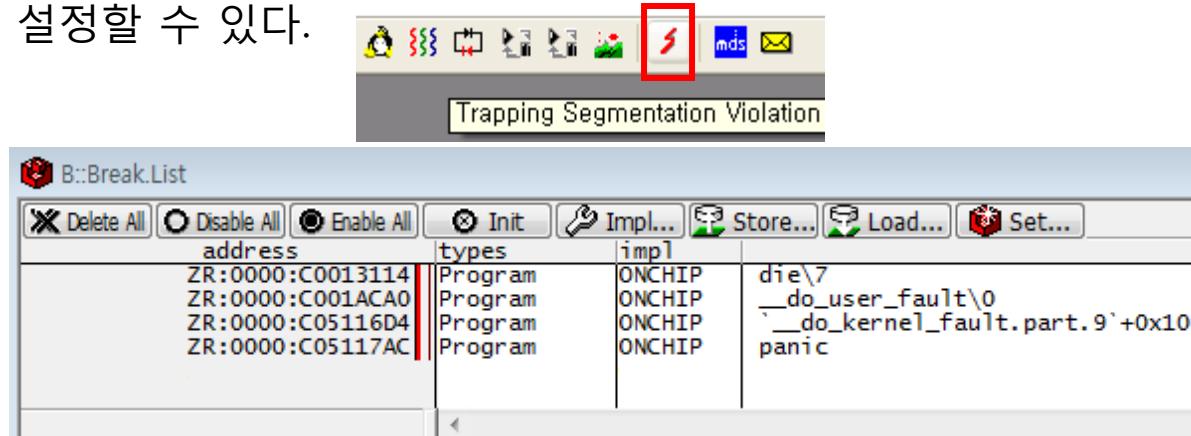
Exception이 발생해서 디버깅을 해야하는 application은 아래에서 확인할 수 있다.
해당 application을 실행하면 jni library에서 문제가 발생해 application이 죽는다.
이 부분을 디버깅해보자.

디버깅해야 하는 application은 android NDK hello application 이다.

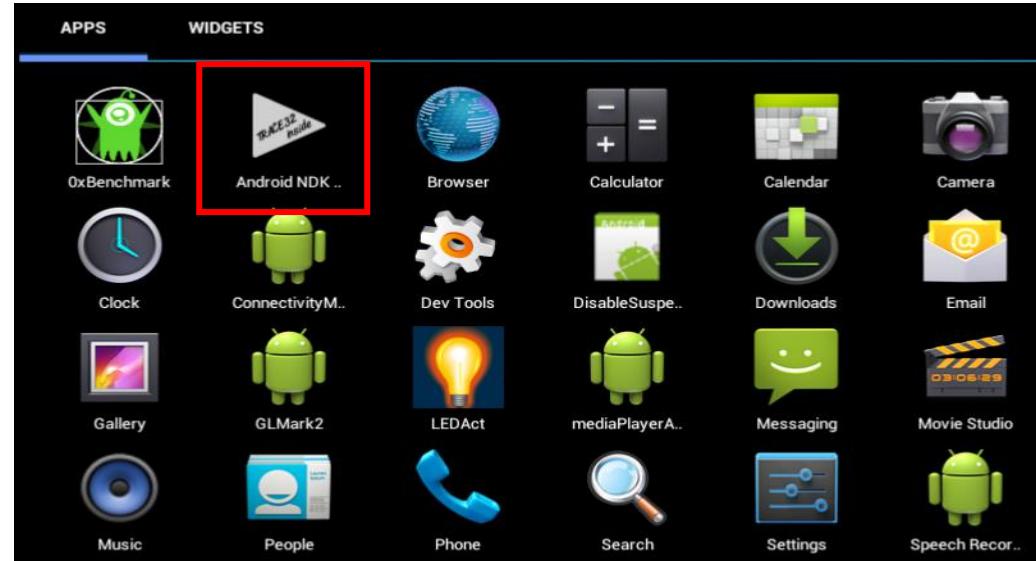


10-2. Android NDK Hello 디버깅

1.1 Exception이 발생되기 전, 즉 재현을 하기 전에 Kernel의 특정 부분에 BreakPoint를 설정한다. TRACE32에서는 아래와 같은 아이콘을 통해 자동으로 BreakPoint를 설정할 수 있다.



1.2 BreakPoint 설정이 완료되면 Target을 Running시켜 Exception이 일어나는 Application을 실행해서 재현을 한다.



10-2. Android NDK Hello 디버깅

1.3 Android NDK Application을 실행하면 Exception이 발생해 미리 설정해 둔 BreakPoint 4곳 중 한군데에서 멈추게 된다.

The screenshot shows the Immunity Debugger interface. The assembly window displays the following code snippet from `do_user_fault`:

```
    /* Something tried to access memory that isn't in our memory
     * User mode accesses just cause a SIGSEGV
     */
    static void __do_user_fault(struct task_struct *tsk, unsigned long addr,
                                unsigned int fsr, unsigned int sig, int code,
                                struct pt_regs *regs)
{
    struct siginfo si;
    #ifdef CONFIG_DEBUG_USER
    if (((user_debug & UDBG_SEGV) && (sig == SIGSEGV)) ||
        ...
    /* Something tried to access memory that isn't in our memory
     * User mode accesses just cause a SIGSEGV
     */
    static void __do_user_fault(struct task_struct *tsk, unsigned long addr,
                                unsigned int fsr, unsigned int sig, int code,
                                struct pt_regs *regs)
{

```

The Registers window shows the following register values:

R0	0E885000	R8	80000000
ZR	80000000	R9	E470E448
C	0805	R10	DEBB5000
V	08	R11	E47B1E3C
I	0805	R12	E47B1E40
F	RS	R13	E47B1D90
T	R6	R14	C001AF40
J	R7	PC	C001ACA0
SPSR	60000010	CPSR	80000113

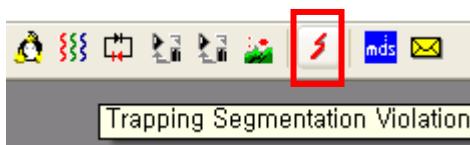
The Stack window shows the current stack frame:

000	do_user_fault	args	loc
000	tsk = 0xDEBB5000,		
	+ addr = 2147483648,		
	+ fsr = 2053,		
	+ sig = 11,		
	+ code = 196609,		
	+ regs = 0xE47B1F80		
	+ si = (si_signo = -1068390500)		

The Breakpoints window shows the current state of breakpoints:

address	types	addr	label	impl.	store	load	set
ZR:0000:C00131141	Program	ONCHIP	die?				
ZR:0000:C001ACAO	Program	ONCHIP	__do_user_fault@0				
ZR:0000:C05116D4	Program	ONCHIP	__do_kernel_fault.part.9'+0x10				
ZR:0000:C05117AC	Program	ONCHIP	panic				

1.4 이 상태에서 처음에 선택했던 아이콘을 다시 한번 클릭하면 문제가 발생했던 위치의 Register를 복원시켜 줍니다.



The screenshot shows the Immunity Debugger interface after a reset. The assembly window displays the assembly code for the application, starting with:

```
ZUR:0B99:9869E420 EBF4FFCD b1 0x9869E35C
ZUR:0B99:9869E424 E3540003 cmp r4,#0x3
ZUR:0B99:9869E428 1AFFFF4 bne 0x9869E400
ZUR:0B99:9869E42C E3A02052 mov r2,#0x52
ZUR:0B99:9869E430 E1CB89080 strb r9,[r11]
ZUR:0B99:9869E434 E5CB2002 strb r2,[r11,#0x2]
ZUR:0B99:9869E438 E2855004 add r5,r5,#0x4
ZUR:0B99:9869E43C E355000C cmp r5,#0x0C
ZUR:0B99:9869E440 1AFFFE8 bne 0x9869E3E8
ZUR:0B99:9869E444 E4980004 ldr r0,[r8],#0x4
ZUR:0B99:9869E448 EBF4FFC9 b1 0x9869E374
ZUR:0B99:9869E44C E15A0008 cmp r10,r8
ZUR:0B99:9869E450 1AFFFFFB bne 0x9869E444
ZUR:0B99:9869E454 E1A00006 cpy r0,r6
ZUR:0B99:9869E458 EBF4FFC5 b1 0x9869E374
ZUR:0B99:9869E45C E59D0004 ldr r0,[r13,#0x4]
ZUR:0B99:9869E460 E59F1024 ldr r1,0x9869E48C
ZUR:0B99:9869E464 E5903000 ldr r3,[r0]
ZUR:0B99:9869E468 E08F0001 add r1,pc,r1
ZUR:0B99:9869E46C E1A0E00F cpy r14,pc
ZUR:0B99:9869E470 E593F29C ldr r3,[r3,#0x29C]
ZUR:0B99:9869E474 E28D000C add r13,r13,#0x0C
ZUR:0B99:9869E478 E8BD8FF0 pop {r4-r11,pc}
ZUR:0B99:9869E47C E3A00000 mov r0,#0x0
```

The Registers window shows the following register values:

R0	2	R8	013773D8
ZR	01274630	R9	5245
C	52	R10	013773E4
V	1	R11	80000000
I	3	R12	FFFFCD80
F	0	R13	BE9D9660
T	013773D8	R14	9869E424
J	9869E490	PC	9869E430
SPSR	60000010	CPSR	60000010

The Stack window shows the current stack frame:

000	ZUR:0B99:0x9869E430(asn)	args	loc
001	ZUR:0B99:0xB65B37B4(asn)		
002	ZUT:0B99:0xB65FBFFA(asn)		
003	ZUT:0B99:0xB65DBD8C(asn)		
004	ZUT:0B99:0xB65FE2EE(asn)		
005	ZUR:0B99:0xB65C5610(asn)		

10-2. Android NDK Hello 디버깅

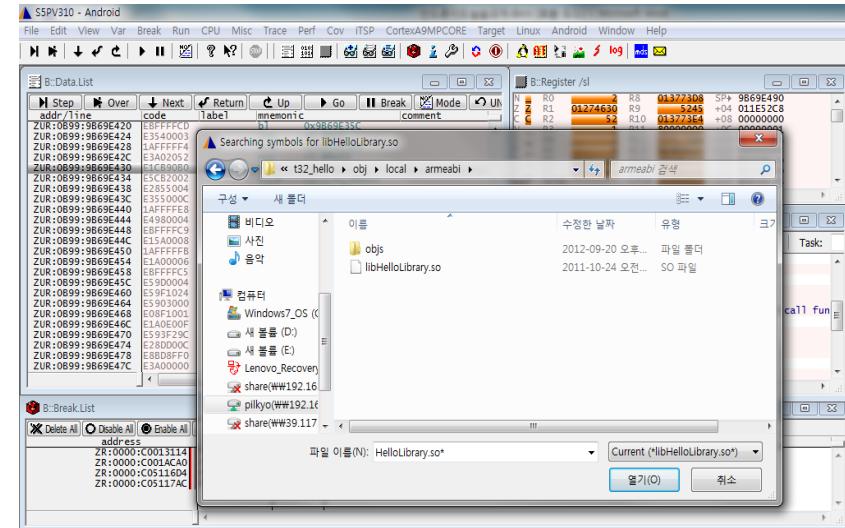
1.5 Register를 복원시킨 후

Symbol이 보이지 않으면

 아이콘을 통해 symbol을 load한다. ndk로 만들어진 application으로 해당 library가 자동으로 load되지 않을 수 있다. 이 때는 해당 library symbol을 찾아서 선택해 준다.

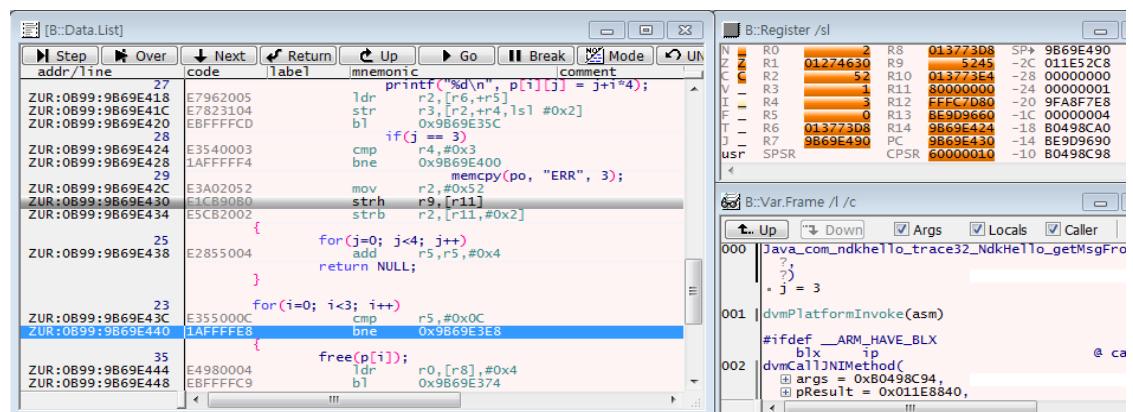
Library symbol 위치는 다음과 같다.

BSP\trace32_examples\t32_hello\obj\local\armeabi\libHelloLibrary.so



1.6 symbol load가 완료되면

오른쪽과 같이 문제가 발생한 위치를 확인할 수 있다.



10-2. Android NDK Hello 디버깅

1.7 실제 분석을 통해 문제의 원인을 찾아보자.

The screenshot shows the assembly code for the `Data.List` function. The code includes the following instructions:

- Line 27: `printf("%d\n", p[i][j] = j+i*4);`
- Line 28: `if(j == 3)`
- Line 29: `memcpy(po, "ERR", 3);`
- Line 30: `mov r2, #0x52`
- Line 31: `strh r9, [r11]`
- Line 32: `strb r2, [r11], #0x2`
- Line 33: `{`
- Line 34: `for(j=0; j<4; j++)`
- Line 35: `add r5, r5, #0x4`
- Line 36: `return NULL;`
- Line 37: `}`
- Line 38: `for(i=0; i<3; i++)`
- Line 39: `cmp r5, #0x0C`
- Line 40: `bne 0x9B69E3E8` (highlighted)
- Line 41: `{`
- Line 42: `free(p[i]);`
- Line 43: `ldr r0, [r8], #0x4`
- Line 44: `bl 0x9B69E374`

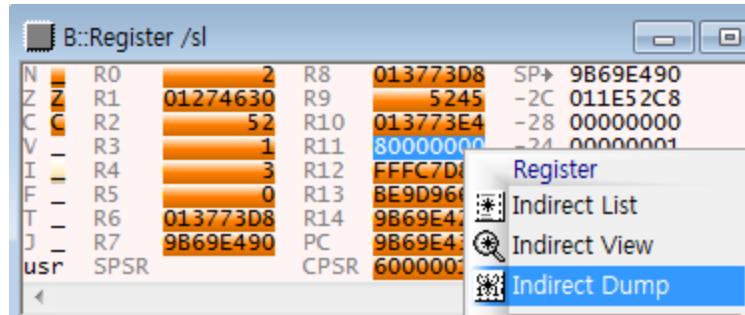
문제가 발생한 부분은 현재 복원된 `Data.List`에서
PC가 위치한 `strh r9,[r11]`에서 문제가
발생했음을 알 수 있다.

B::Register /sl						
N	R0	2	R8	013773D8	SP	9B69E490
Z	R1	01274630	R9	5245	-2C	011E52C8
C	R2	52	R10	013773E4	-28	00000000
V	R3	1	R11	80000000	-24	00000001
I	R4	3	R12	FFFC7D80	-20	9FA8F7E8
F	R5	0	R13	BE9D9660	-1C	00000004
T	R6	013773D8	R14	9B69E424	-18	B0498CA0
J	R7	9B69E490	PC	9B69E430	-14	BE9D9690
usr	SPSR	600000010	CPSR	600000010	-10	B0498C98

10-2. Android NDK Hello 디버깅

Register 정보를 확인해 보면 r11의 값이 0x80000000입니다.

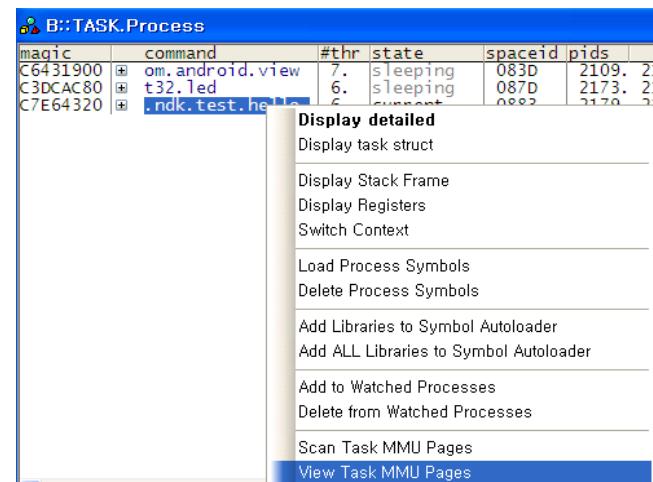
0) Address의 메모리 정보를 확인하면 존재하지 않는 address인 것을 확인할 수 있다.



address	0	4	8	C
ZUD:0B99:80000000	???????	???????	???????	0123456789ABCDEF
ZUD:0B99:80000010	???????	???????	???????	0123456789ABCDEF
ZUD:0B99:80000020	???????	???????	???????	0123456789ABCDEF
ZUD:0B99:80000030	???????	???????	???????	0123456789ABCDEF
ZUD:0B99:80000040	???????	???????	???????	0123456789ABCDEF
ZUD:0B99:80000050	???????	???????	???????	0123456789ABCDEF
ZUD:0B99:80000060	???????	???????	???????	0123456789ABCDEF
ZUD:0B99:80000070	???????	???????	???????	0123456789ABCDEF
ZUD:0B99:80000080	???????	???????	???????	0123456789ABCDEF
ZUD:0B99:80000090	???????	???????	???????	0123456789ABCDEF
ZUD:0B99:800000A0	???????	???????	???????	0123456789ABCDEF
ZUD:0B99:800000B0	???????	???????	???????	0123456789ABCDEF
ZUD:0B99:800000C0	???????	???????	???????	0123456789ABCDEF
ZUD:0B99:800000D0	???????	???????	???????	0123456789ABCDEF
ZUD:0B99:800000E0	???????	???????	???????	0123456789ABCDEF
ZUD:0B99:800000F0	???????	???????	???????	0123456789ABCDEF
ZUD:0B99:80000100	???????	???????	???????	0123456789ABCDEF
ZUD:0B99:80000110	???????	???????	???????	0123456789ABCDEF
ZUD:0B99:80000120	???????	???????	???????	0123456789ABCDEF
ZUD:0B99:80000130	???????	???????	???????	0123456789ABCDEF
ZUD:0B99:80000140	???????	???????	???????	0123456789ABCDEF

이는 MMU 정보를 통해서도 확인할 수 있다.

0x80000000은 Physical로 Mapping되지 않은 것을 확인할 수 있다.

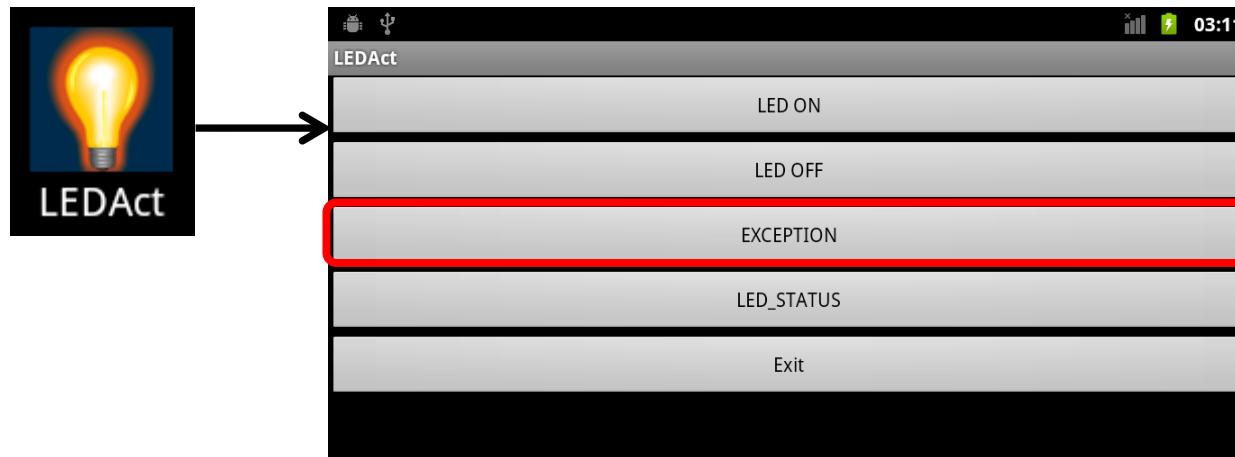


address	physical	sec	d	size
C:0B99:80000000--9B69DFFF	A:5DEE0000--5DEE0FFF	s	01	00001000
C:0B99:9B69E000--9B69EFFF	A:5DEE5000--5DEE5FFF	s	01	00001000
C:0B99:9B69F000--9B69FFFF	A:5DE57000--5DE57FFF	s	01	00001000
C:0B99:9B6A0000--9B6A0FFF	A:66F97000--66F97FFF	s	01	00001000
C:0B99:9B6A1000--9B6A1FFF	A:66F95000--66F95FFF	s	01	00001000
C:0B99:9B6A2000--9B6A2FFF	A:66F97000--66F97FFF	s	01	00001000
C:0B99:9B6A3000--9B6A3FFF	A:66F97000--66F97FFF	s	01	00001000
C:0B99:9B6A4000--9B6A4FFF	A:6BAB4000--6BAB4FFF	s	01	00001000
C:0B99:9B6A5000--9B6A7FFF				
C:0B99:9B6A8000--9B6A8FFF	A:5DE4A000--5DE4AFFF	s	01	00001000
C:0B99:9B6A9000--9B7A7FFF	A:5DE31000--5DE31FFF	s	01	00001000
C:0B99:9B7A8000--9B7A8FFF	A:5DE31000--5DE31FFF	s	01	00001000
C:0B99:9B7A9000--9B7ABFFF	A:5DE2F000--5DE2FFFF	s	01	00001000
C:0B99:9B7AC000--9B7ACFFF	A:5DE1D000--5DE1DFFF	s	01	00001000
C:0B99:9B7AD000--9B8ABFFF	A:5DE1D000--5DE1DFFF	s	01	00001000
C:0B99:9B8AC000--9B8ACFFF	A:5DE1D000--5DE1DFFF	s	01	00001000
C:0B99:9B8B0000--9B8B0FFF	A:5DE1D000--5DE1DFFF	s	01	00001000

10-3. LEDAct 디버깅(Question)

LEDAct application에서 EXCEPTION 버튼을 클릭하여 문제점을 파악해봅시다.

앞서 실습한 방법과 동일한 방법으로 아래 exception도 debugging 해보자.
(exception이 발생하기 위해서는 t32_led.ko 모듈이 load되어야 한다.)



11. 추가 실습

추가 실습을 통해 TRACE32의 활용도를 더 확대해 봅니다

1. FrameBuffer 디버깅 실습
2. Kernel Log 확인하기
3. Logcat Log 확인하기
4. Dalvik List 정보 확인하기

11-1. Frame buffer 디버깅 실습

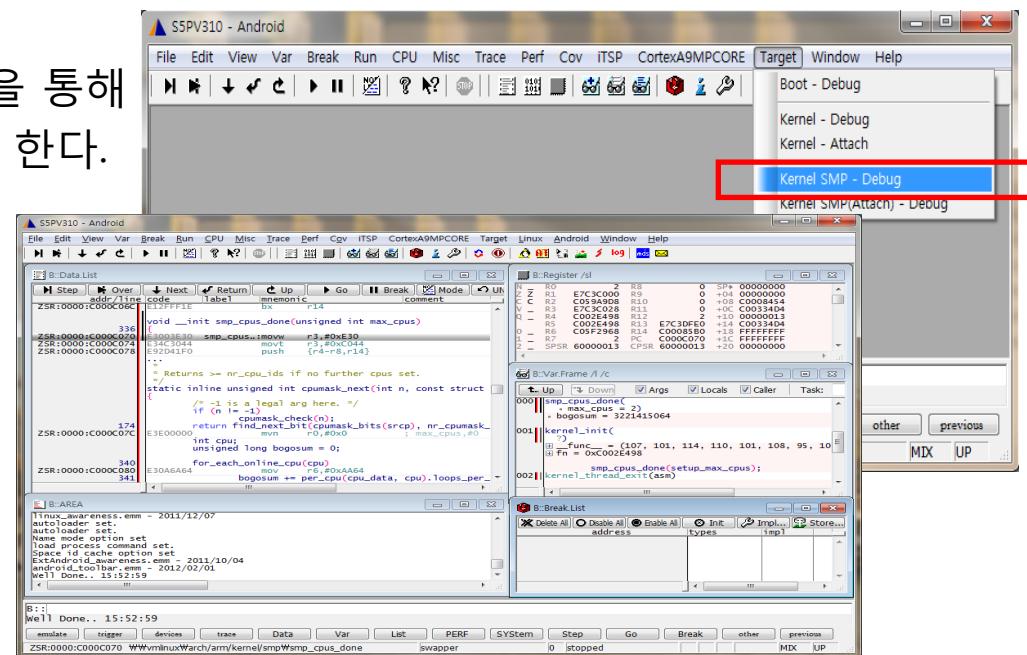
LCD가 없는 상황이거나 LCD에 이미지가 원하는 것과 다르게 표현되는 경우
TRACE32를 이용해 Frame buffer 의 이미지를 볼 수 있습니다

1. Frame Buffer Image debugging

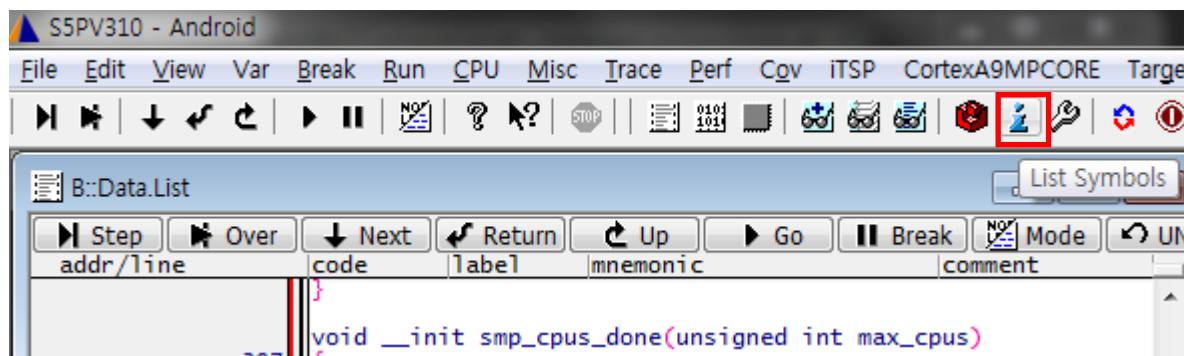
LCD가 없거나 출력되는 이미지를 LCD가 아닌 다른 방법으로 확인하고자 할 때 혹은
ram dump를 TRACE32 simulator로 복원했을 때 TRACE32를 이용하면 Frame Buffer에
있는 Image Data를 확인할 수 있다.

11-1. Frame buffer 디버깅 실습

1.1 Kernel SMP debugging 환경을 통해 smp_cpus_done()에 멈추도록 한다.

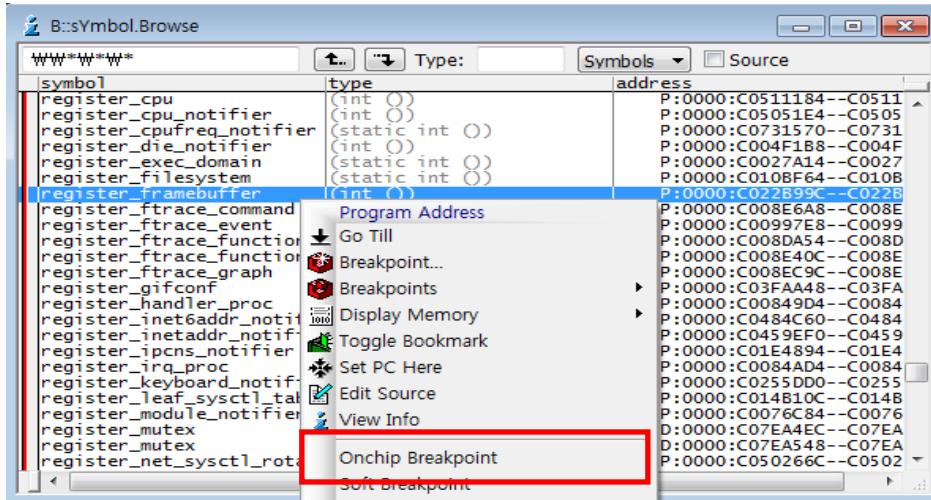


1.2 해당 CPU의 Frame Buffer Address를 구하기 위해 Symbol Browser를 이용해 register_framebuffer()에 Breakpoint를 설정하고 Target Board를 동작시킨다.

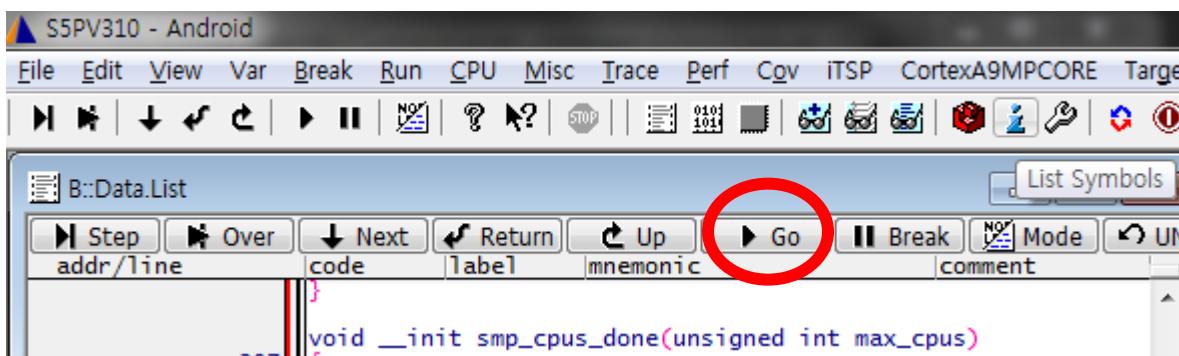


11-1. Frame buffer 디버깅 실습

1.3 Symbol을 찾은 후 오른쪽 마우스 클릭으로 onchip breakpoint를 설정한다.

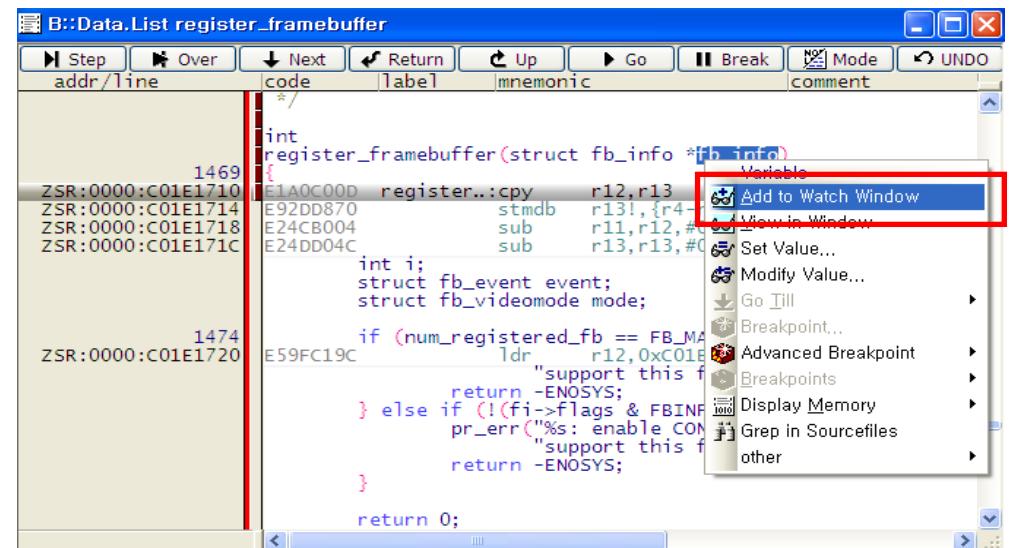


1.4 Breakpoint 설정이 완료되면 target을 running시켜 설정된 register_framebuffer까지 진행시킨다.

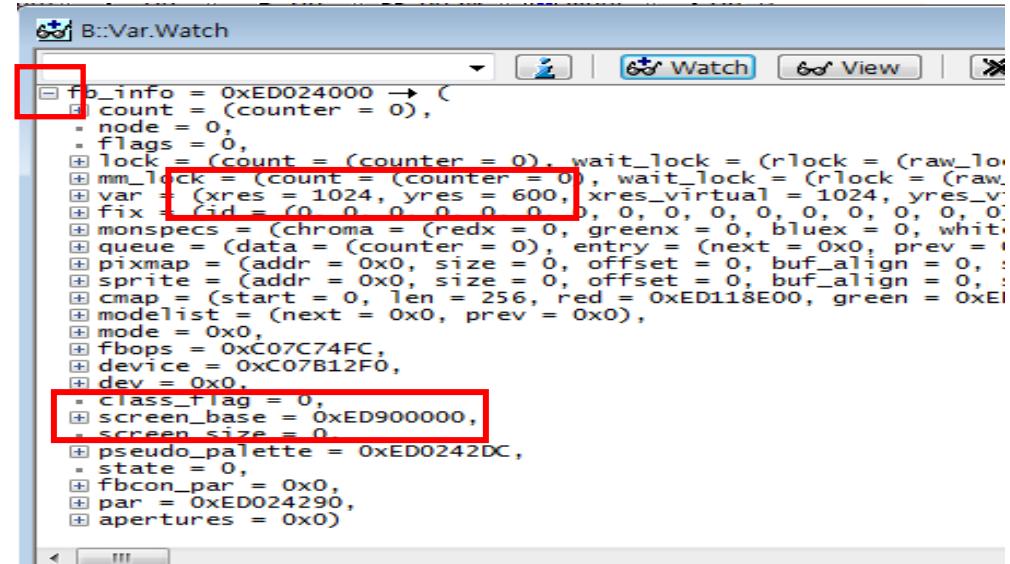


11-1. Frame buffer 디버깅 실습

1.5 Target이 BreakPoint에 의해 멈추게 되면 Parameter로 들어오는 fb_info 변수를 Watch Window에 등록시켜 Framebuffer Address와 LCD Size를 확인한다.



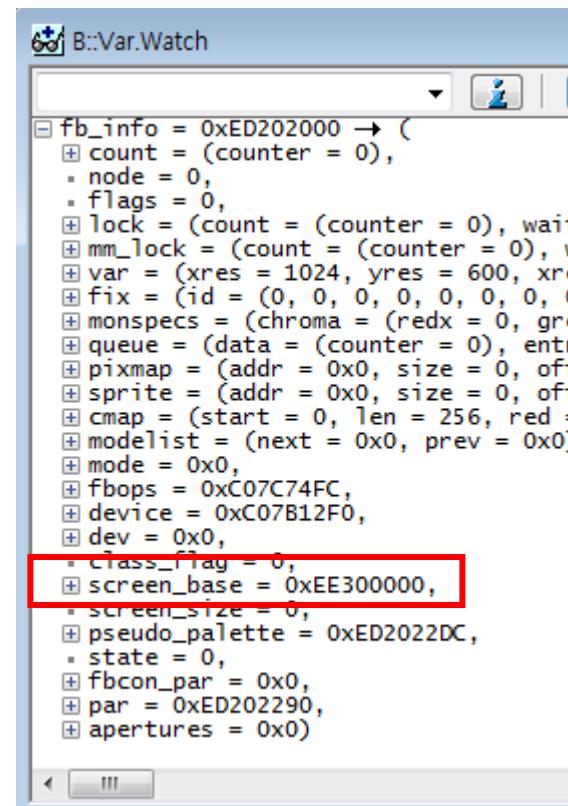
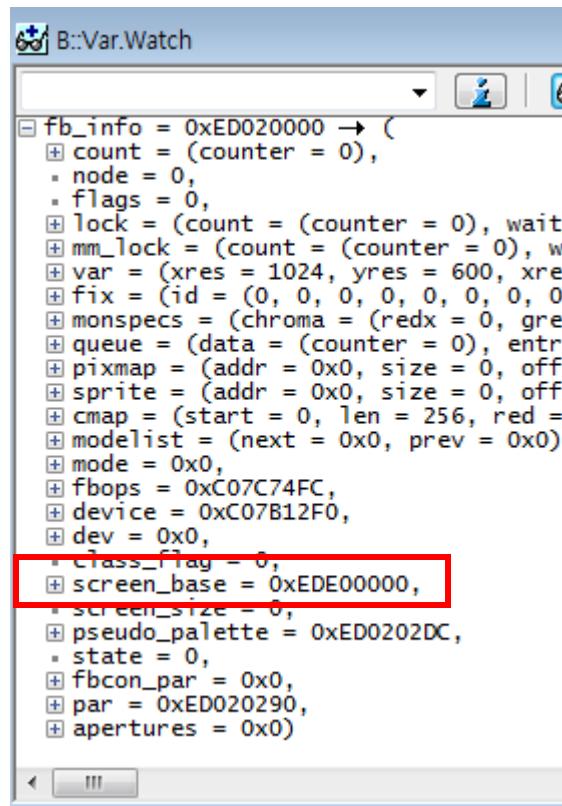
1.6 fb_info 변수를 통해 Frame Buffer Address는 Screen_base로, LCD Size 증가로 가로 Size는 xres, 세로 Size는 yres로 확인 가능하다.



11-1. Frame buffer 디버깅 실습

* 참고

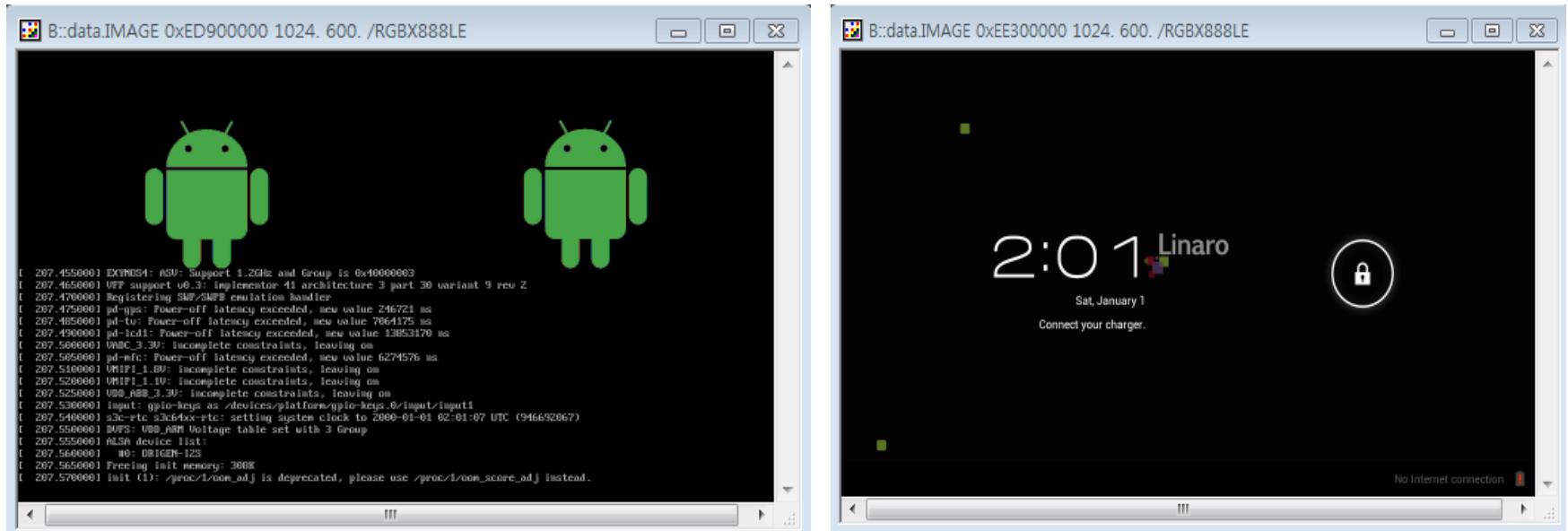
android에서는 여러 개의 framebuffer를 사용할 수 있기 때문에 위의 정보를 확인한 후 다시 go를 하면 해당 함수인 register_framebuffer에 다시 들어오게 된다.
들어올 때마다 해당 정보를 확인하자.



11-1. Frame buffer 디버깅 실습

1.7 어느 정도 부팅을 시킨 후 아래 명령어를 통해 **LCD 화면**을 TRACE32로 확인해 보자.

```
B:::data.IMAGE 0xED900000 1024. 600. /RGBX888LE  
B:::data.IMAGE 0xEDE00000 1024. 600. /RGBX888LE  
B:::data.IMAGE 0xEE300000 1024. 600. /RGBX888LE
```



11-1. Frame buffer 디버깅 실습

Data.IMAGE

Display image data

Format: Data.IMAGE <address> <horiz> <vert> [*I<format>* | <options>]

<format>: MONO, CGA, GrayScale8, JPEG
Palette256 <red> <green> <blue> ...
Palette256X6 <address>
Palette256X12 <address>
Palette256X24 <address>
RGB111, RGB555, RGB555LE, RGB565, RGB565LE,
RGB888, RGB888LE, RGBX888, RGBX888LE,
YUV420, YUV422, YUV422P, YUV422PS, YUV422W, YUV422WS

<options>: BottomUp
FullUpdate

Image Format의 경우 Platform마다 차이가 있기 때문에 이 부분을 개발자들이 직접 확인이 필요하다. 참고로 최근 Android Smartphone에서는 RGBX888 혹은 RGBX888LE 를 많이 사용한다.

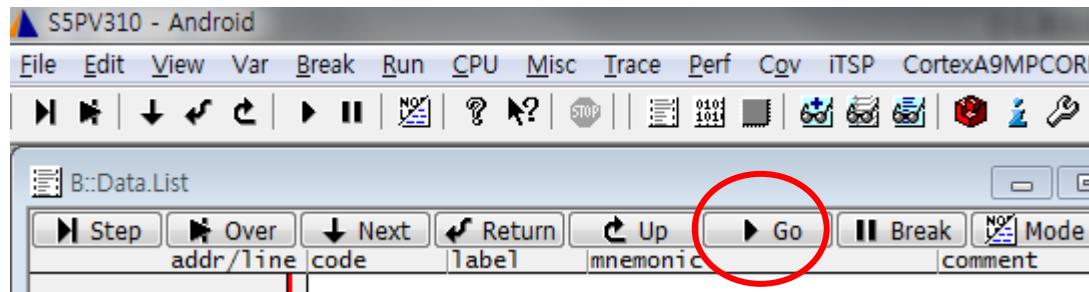
11-2. Kernel Log 확인하기

USB 및 UART를 연결하지 않고 JTAG Interface로 kernel log 정보를 확인해 봅니다

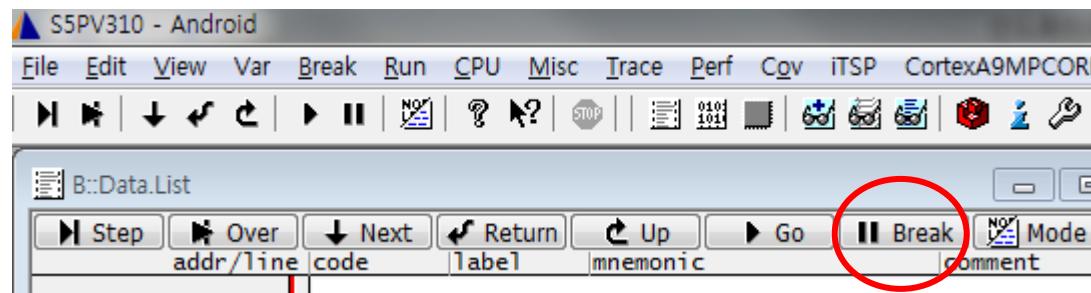
1. Kernel Log 확인하기

미처 USB를 연결하지 않았거나 ram dump를 뜯 후 log를 확인하고 싶은 경우 TRACE32를 통해서 log정보를 확인할 수 있다.

1.1 Log를 확인하고 싶은 시점까지 타겟을 running시킨다.

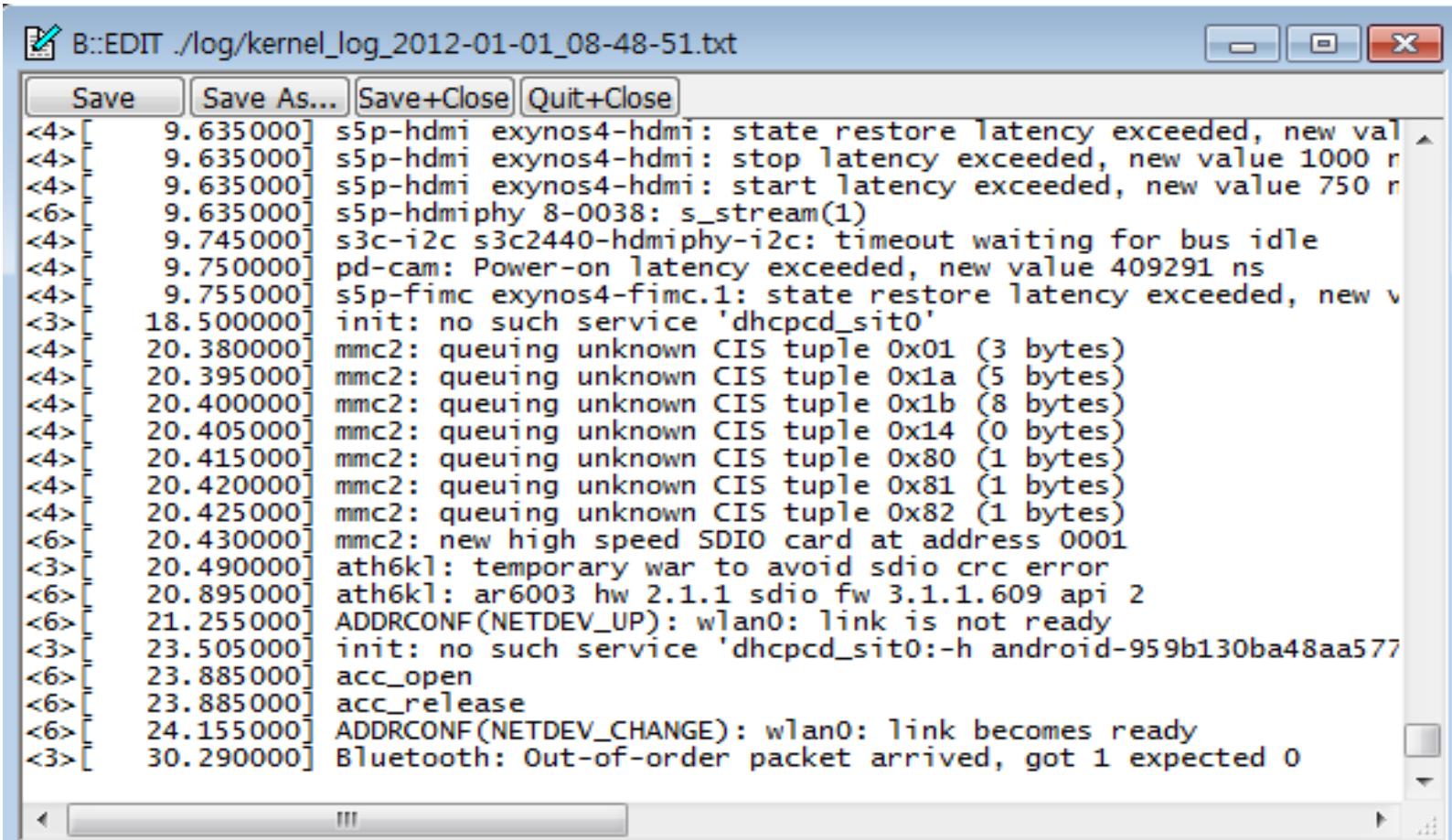


1.2 Target을 멈추기 위해 break 버튼을 누른다.



11-2. Kernel Log 확인하기

1.3 우측 상단의 버튼 중에  버튼을 누르면 현재 멈춘 시점까지의 log를 확인할 수 있다.



The screenshot shows a Windows Notepad window titled "B::EDIT ./log/kernel_log_2012-01-01_08-48-51.txt". The window contains a list of kernel log entries. Each entry consists of a timestamp in brackets followed by a log message. The log messages include various system events such as latency checks, mmc card queueing, and network interface configuration. The window has standard Windows-style buttons for Save, Save As..., Save+Close, and Quit+Close.

```
<4>[ 9.635000] s5p-hdmi exynos4-hdmi: state restore Latency exceeded, new val
<4>[ 9.635000] s5p-hdmi exynos4-hdmi: stop latency exceeded, new value 1000 ns
<4>[ 9.635000] s5p-hdmi exynos4-hdmi: start latency exceeded, new value 750 ns
<6>[ 9.635000] s5p-hdmiphy 8-0038: s_stream(1)
<4>[ 9.745000] s3c-i2c s3c2440-hdmiphy-i2c: timeout waiting for bus idle
<4>[ 9.750000] pd-cam: Power-on latency exceeded, new value 409291 ns
<4>[ 9.755000] s5p-fimc exynos4-fimc.1: state restore latency exceeded, new val
<3>[ 18.500000] init: no such service 'dhpcd_sit0'
<4>[ 20.380000] mmc2: queuing unknown CIS tuple 0x01 (3 bytes)
<4>[ 20.395000] mmc2: queuing unknown CIS tuple 0x1a (5 bytes)
<4>[ 20.400000] mmc2: queuing unknown CIS tuple 0x1b (8 bytes)
<4>[ 20.405000] mmc2: queuing unknown CIS tuple 0x14 (0 bytes)
<4>[ 20.415000] mmc2: queuing unknown CIS tuple 0x80 (1 bytes)
<4>[ 20.420000] mmc2: queuing unknown CIS tuple 0x81 (1 bytes)
<4>[ 20.425000] mmc2: queuing unknown CIS tuple 0x82 (1 bytes)
<6>[ 20.430000] mmc2: new high speed SDIO card at address 0001
<3>[ 20.490000] ath6kl: temporary war to avoid sdio crc error
<6>[ 20.895000] ath6kl: ar6003 hw 2.1.1 sdio fw 3.1.1.609 api 2
<6>[ 21.255000] ADDRCONF(NETDEV_UP): wlan0: link is not ready
<3>[ 23.505000] init: no such service 'dhpcd_sit0:-h android-959b130ba48aa577'
<6>[ 23.885000] acc_open
<6>[ 23.885000] acc_release
<6>[ 24.155000] ADDRCONF(NETDEV_CHANGE): wlan0: link becomes ready
<3>[ 30.290000] Bluetooth: Out-of-order packet arrived, got 1 expected 0
```

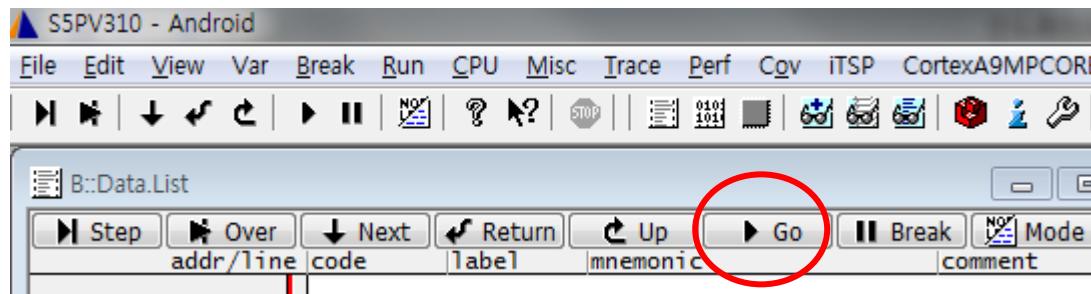
11-3. Logcat Log 확인하기

Adb로 연결하지 않고 Eclipse 또한 사용하지 않은 상태에서 JTAG Interface로 Logcat log 정보를 확인해 봅니다

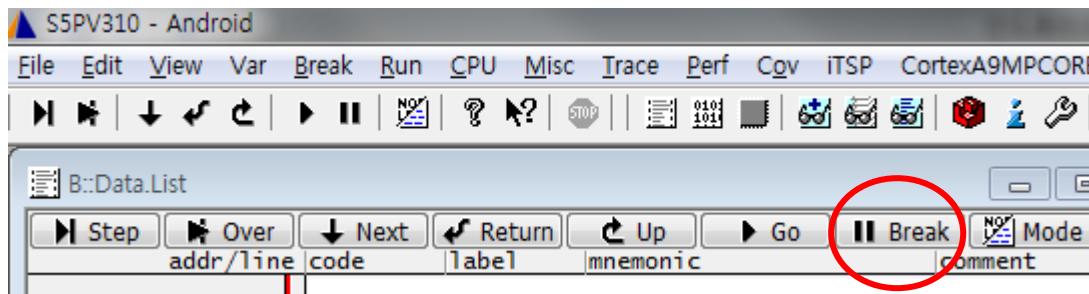
1. Logcat Log 확인하기

Adb 통신을 하지 않고 eclipse를 사용하지 않아도 TRACE32를 이용해 logcat 정보를 확인할 수 있다.

1.1 Log를 확인하고 싶은 시점까지 타겟을 running시킨다.

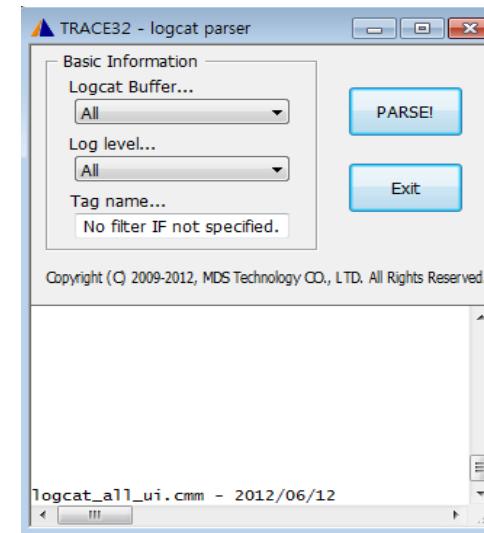
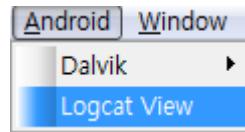


1.2 Target을 멈추기 위해 break 버튼을 누른다.

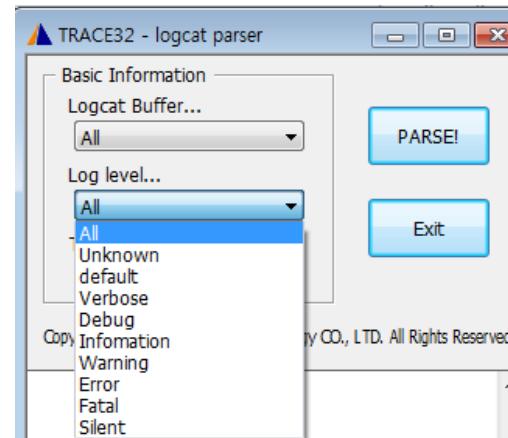
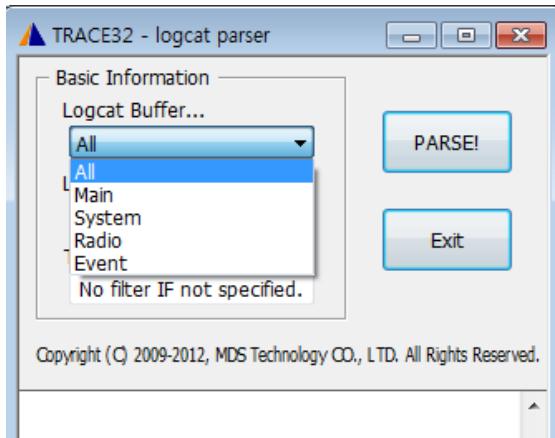


11-3. Logcat Log 확인하기

1.3 메뉴 중 android 메뉴의 Logcat View를 선택한다.



1.4 Log 종류와 log level 별로 parsing해서 해당 정보를 확인할 수 있다.



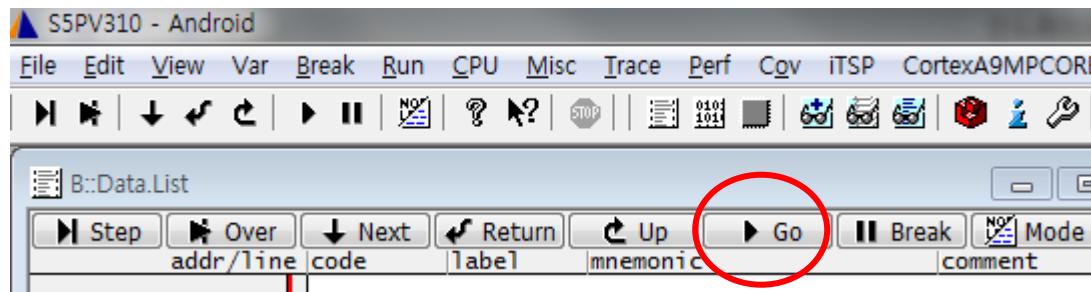
11-4. Dalvik List 확인하기

Dalvik VM을 통해 실행되고 있는 Android application의 실행 list와 call stack을 확인해 봅니다

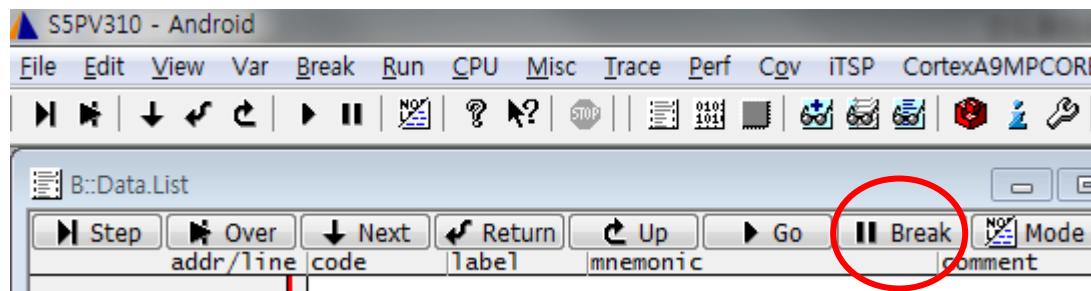
1. Dalvik List 확인하기

TRACE32는 android JAVA를 지원하지는 않지만 dalvik에서 동작되는 application의 실행 정보는 확인할 수 있다.

1.1 Application의 call stack을 확인하고 싶은 시점까지 타겟을 running시킨다.

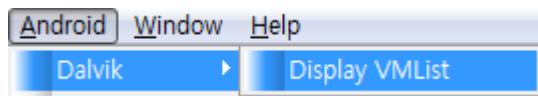


1.2 Target을 멈추기 위해 break 버튼을 누른다.



11-4. Dalvik List 확인하기

1.3 메뉴 중 android 메뉴의 Dalvik→Display VMList 를 선택한다.



1.4 해당 메뉴를 선택하면 현재 실행되고 있는 android application list를 확인할 수 있다.
이 목록은 zygote에 의해 fork 된 application list이다.

magic	spid	pid	process/task name	method name	class descriptor	class source	states
C1949800	093E	2366	system_server	initI	Lcom/android/server	SystemServer.	NATIVE
E5A3F800	09A5	2469	com.android.systemu	nativePollOnce	Landroid/os/Message	MessageQueue.	NATIVE
E59A2400	09B5	2485	android.process.med	nativePollOnce	Landroid/os/Message	MessageQueue.	NATIVE
E5ABC800	09C1	2497	org.linaro.wallpape	nativePollOnce	Landroid/os/Message	MessageQueue.	NATIVE
E5940000	09D1	2513	com.android.inputme	nativePollOnce	Landroid/os/Message	MessageQueue.	NATIVE
E5824800	09E4	2532	com.android.phone	nativePollOnce	Landroid/os/Message	MessageQueue.	NATIVE
E5827C00	09F1	2545	com.android.launcher	nativePollOnce	Landroid/os/Message	MessageQueue.	NATIVE
E4994400	0A1A	2586	android.process.aco	nativePollOnce	Landroid/os/Message	MessageQueue.	NATIVE
E3AB4000	0A38	2616	com.android.smspush	nativePollOnce	Landroid/os/Message	MessageQueue.	NATIVE
E3192C00	0A67	2663	com.android.contact	nativePollOnce	Landroid/os/Message	MessageQueue.	NATIVE
C19C1C00	0A8E	2702	com.android.email	wait	Ljava/lang/Object;	Object.java	WAIT

11-4. Dalvik List 확인하기

1.5 해당 application의 call stack을 보고자 한다면 특정 application을 더블 클릭하면 확인이 가능하다.

method name	class descriptor	class source
spid:0A8E, pid:2702 m.android.email-m.android.email		
wait	Ljava/lang/Object;	Object.java
wait	Ljava/lang/Object;	Object.java
awaitLoadedLocked	Landroid/app/SharedPreferencesImpl;	SharedPreferencesImpl
getBoolean	Landroid/app/SharedPreferencesImpl;	SharedPreferencesImpl
getEnableDebugLogg	Lcom/android/email/Preferences;	Preferences.java
onCreate	Lcom/android/email/Email;	Email.java
callApplicationOnC	Landroid/app/Instrumentation;	Instrumentation.java
handleBindApplicat	Landroid/app/ActivityThread;	ActivityThread.java
access\$1300	Landroid/app/ActivityThread;	ActivityThread.java
handleMessage	Landroid/app/ActivityThread\$H;	ActivityThread.java
dispatchMessage	Landroid/os/Handler;	Handler.java
loop	Landroid/os/Looper;	Looper.java
main	Landroid/app/ActivityThread;	ActivityThread.java
invokeNative	Ljava/lang/reflect/Method;	Method.java
invoke	Ljava/lang/reflect/Method;	Method.java

12. TRACE32 활용 Tip

1. 내부 Serial Terminal 사용하기

Linux Console 이용하기

B:: term.protocol COM com1 115200. 8 NONE 1STOP NONE

B::term.gate

<etc>

1. 터미널 size 조정: **B::term.size 100. 100.**
2. scroll bar 생성: **B::term.scroll on**
3. 터미널 mode 변경: **B::term.mode VT100**
4. 터미널 내용 저장: **B::term.write <filename>**

12. TRACE32 활용 Tip

2. CMM에 대한 Details

2.1 MMU 관련 선언

Linux를 디버깅하기 위해서는 MMU 관련 설정이 꼭 필요하다. 그 이유는 Virtual Address(가상 주소)에서 Physical Address(물리 주소)로 어떻게 값이 변환되는지 TRACE32 Debugger는 알아야 하기 때문이다. 그래서 MMU 관련된 설정을 하게 된다.

MMU FORMAT <format> <base address> <kernel translation>

Virtual Address와 Physical Address를 제대로 접근하기 위해서 TRACE32는 타겟의 MMU Table 포맷의 정보를 정확히 알 필요가 있기에 MMU FORMAT이라는 명령을 사용한다.

12. TRACE32 활용 Tip

(예제) MMU FORMAT

```
Y:\APPS\android\out\target\product\gernal\obj\KERNEL_OBJ\System.map
```

```
C0004000 A swapper_pg_dir
```

```
C0008000 T __init_begin
```

```
C0008000 T _sinittext
```

```
C0008000 T _stext
```

```
C0008000 T stext
```

```
C0008034 t __enable_mmu
```

```
MMU FORMAT LINUX swapper_pg_dir 0xC0000000--0xC1fffff 0x00000000
```

[Format]

LINUX(standard) / LINUX26(PPC) / LINUXEXT(PPC 64bit table) / LINUX32(MIPS32) /
LINUX64(MIPS64)

12. TRACE32 활용 Tip

[Base Address]

커널의 Translation Table의 Base Address가 되는데, 이 Address는 보통 swapper_pg_dir이다.

[Kernel Translation]

Kernel Address Range의 Virtual Address를 Physical Address로의 변환을 위해 사용된다.

Kernel은 ARM/PPC의 경우 0xC000 0000() 에서부터 시작한다.

보통 Range는 Physical Memory의 용량을 Offset으로 하는데, 현재 사용하는 타겟은 64MB를 가지고 있으므로 0x400 0000의 range를 가진다. 참고로, MIPS의 경우 0x8000 0000 부터 시작한다.

TRANSlation.Create <virtual address> <physical address>

이 명령을 사용하여 Virtual address와 Physical Address를 지정해주면 타겟의 MMU Table을 읽어 들일 때 해당 영역(Address Range)은 읽어 들이지 않으므로 약간의 속도 개선이 있다. 즉, Virtual Address와 Physical Address가 고정된 영역인 경우 사용하면 좀 더 디버깅할 때 유리하다. 보통 커널의 경우가 많이 사용하는데 커널은 Static Address Translation을 하기 때문이다.

TRANSlation.Create 0xC0000000--0xC1fffff 0x00000000 ; map kernel pages at RAM start
→ MMU.FORMAT LINUX swapper_pg_dir 0xC0000000--0xC1fffff 0x00000000와 똑같이 하면된다.

12. TRACE32 활용 Tip

TRANSlation.COMMON <virtual address range>

커널 영역에 있는 커널의 코드는 어떤 프로세스에서나 접근이 가능한데, MMU.COMMON 명령을 사용해 이 영역을 설정한다. 보통 0xC000 0000--0xFFFF FFFF로 설정한다.

Y:\WAPPS\Android\kernel\arch\arm\include\asm\memory.h

```
/*
 * PAGE_OFFSET - the virtual address of the start of the kernel image
 * TASK_SIZE - the maximum size of a user space task.
 * TASK_UNMAPPED_BASE - the lower boundary of the mmap VM area
 */
#define PAGE_OFFSET UL(CONFIG_PAGE_OFFSET)
#define TASK_SIZE (UL(CONFIG_PAGE_OFFSET) - UL(0x01000000))
#define TASK_UNMAPPED_BASE (UL(CONFIG_PAGE_OFFSET) / 3)
```

TRANSlation.COMMON 0xBf000000--0xffffffff ; Common Area for Kernel and Processes
(MIPS의 경우에는 0x7F00 0000--0xFFFF FFFF로 설정한다)

12. TRACE32 활용 Tip

MMU.TableWalk <ON/OFF>

MMU table 정보는 종종 바뀔 가능성이 있는데 MMU.TableWalk라는 옵션을 활용하지 않는다면, MMU table 정보가 바뀔 때마다 MMU table 정보를 갱신해야 한다.

이 기능을 사용 하면 Go 또는 Step 기능을 사용할 때마다 MMU table 정보를 임시적으로 갱신한다. 그러나 이 기능을 사용하면 Target에서 table정보를 읽어오는데 시간이 걸리므로 느려지는 현상이 발생할 수 있다. **임시적으로 갱신하므로** MMU.List 정보자체를 갱신해야 하고 싶다면 MMU.SCANALL / MMU.TaskPageTable.SCAN 과 같은 기능들을 활용해야 한다.

12. TRACE32 활용 Tip

2.2 Linux Awareness 설정

MMU 설정만으로 Linux Awareness 기능을 사용할 수 있는 것은 아니다.

demo<arch><kernel>linux 에 들어있는 linux.t32와 linux.men 파일을 활용해야 한다. linux.t32는 Awareness의 중심적인 파일이며, linux.men 파일은 Linux 메뉴를 만들기 위한 파일이다.

TASK.CONFIG <awareness file>

Awareness file을 등록하기 위한 기능이다.

MENU.ReProgram <menu file>

Menu file을 등록하기 위한 기능이다. (여러분들 마음대로 메뉴 구성이 가능하다.)

sYmbol.AutoLoad.CHECKLINUX "<action>"

자동적으로 디버깅 심볼을 로딩시켜주는 기능을 활용하기 위한 기능으로 Autoloader라고 불린다. Autoloader는 등록시켜놓은 리눅스 컴포넌트(application, library, module)들의 주소를 유지하는데 사용자가 일정 address에 접근 시에 해당하는 주소에 알맞은 심볼정보를 자동적으로 로딩시켜 준다.

예를 들자면, Application을 지정해 놓고 해당 address에 접근하면, "<action>"에 지정한 대로 수행한다.

예제에 쓰이는 os.ppd() function은 PRACTICE 스크립트가 동작하는 위치의 디렉토리의 절대경로를 의미하므로 autoload.cmm을 실행하게 되는 것이다

Board Specification

- CPU S5PV310(Exynos4 Series) Cortex-A9 DualCore 1GHz
- Connector 70Pin connector x 4 - CPU/Base board Connector
- Memory DDR3 1GBytes (256 MByte x 4)
- Storage SD Card 2 Slot
- WiFi SWB-A31(삼성전기 모듈), 802.11b/g/n
- Bluetooth V2.1+EDR/BT3.0 (WiFi 통합모듈)
- Sound RealTek ALC5625D IIS/PCM interface Phone용 Audio Codec
- Display LVDS, 7" LCD(1024x600), Touch 정전식
- TV Out HDMI 1 Port
- USB USB 2.0 HostController, USB 2.0 Device 1 Port
- Debug I/F JTAG, SW-DP, ITM,
- Camera 50 Pin connector

TRACE32

제품정보

- 정규교육
- 외부요청교육
- 강의동영상
- 위치안내

교육

- Flash Program
- Peripheral Browser
- S/W Update
- RTOS
- Trace
- CMM
- TSP
- Edu Text/Sample
- 뉴스레터
- 활용사례

자료실

Q&A

FAQ

고객지원

5553 cmm file에서 다른 파일 변수 읽어오는 방법

- Break Point
- Error Handling
- Debugging Tip
- ETC
- Debug Pin Map
- Debug Module Type
- 지원환경

PowerDebug 일반형

디버그 로직을 갖는 모든 Processor를 지원하며
Processor specific한 License를 통해
정밀한 디버깅과 모니터링을 수행합니다



PowerTrace



Logic Analyzer



Solution



Compiler



지원하는
Processor

지원하는
RTOS

지원하는
Compiler

지원하는
Flash Device

TRACE32
활용사례

TRACE32
고객평가

Latest News

- [임베디드월드] TRACE32를 활용한 안드로이드 디버깅
- 가상화 임베디드 운영체제 PikeOS Support
- Xilinx Cortex™-A9 MPCore™ Zynq-7000 Support
- ATMEL 32bit RISC AVR32 Support
- TI-16bit MicroController MSP430 Support
- [전자신문] 임베디드 개발 툴 '트레이스 32' 인기 치솟다

11'06/29

11'05/20

11'04/11

11'04/07

11'04/04

11'03/16

견적요청

기술지원



MDS테크놀로지

[ERR] Emulation Debug Port Fail

고객지원



고객과 함께 생각하고 문제를 해결할때
진정한 보람을 얻는 것이 우리의 고객정신입니다.

Technical Support Request

- Ref. E-mail : trace32@mdstec.com
- Homepage www.trace32.com : Q&A or 기술지원 버튼
 - Training Course : 홈페이지 내부 교육 메뉴
 - Repair Support Service Tel : 031-627-3119



Android Debugging

감사합니다

MDS Technology
Technical Support Team