The Jasmine OpenSSD Platform

Version 1.2

FTL Developer's Guide

Revision history

Date	Author	Description	Rev.
2011-04-27	임상필 (성균관대)	Initial release	1.0
2011-05-20	임상필 (성균관대)	Section 2.2 update	1.1
2012-01-12	임상필 (성균관대)	Section 2.2.3 update	1.2

Contents
The Jasmine OpenSSD Platform: FTL Developer's Guide

Preface

About this document

본 매뉴얼은 (주)인디링스의 Barefoot™ 컨트롤러 기반의 Jasmine OpenSSD 플랫폼을 위한 FTL 개발 가이드 문서이다. 이 문서에 포함된 주된 내용은 아래와 같다.

- Jasmine OpenSSD 플랫폼을 위한 개발 환경 구축 가이드
- 펌웨어 build 및 설치 과정에 대한 설명
- FTL(Flash Translation Layer) 포팅 및 디버깅 가이드를 포함한 전반적인 FTL 개발 과정 설명

Contents

본 기술문서는 다음과 같은 순서로 작성되어 있다.

Chapter 1. Getting Started to Develop an FTL

이 장에서는 Jasmine OpenSSD 플랫폼에 FTL 을 포팅하기 전, 개발 환경 구축 과정에 대해 설명한다.

Chapter 2. FTL Porting

이 장에서는 Jasmine OpenSSD 플랫폼에 새 FTL 기법을 포팅하는 과정에 대해 소개한다.

Chapter 3. Compile, Build & Install Firmware

이 장에서는 펌웨어를 컴파일 및 build 하는 과정과 Jasmine 보드에 펌웨어를 설치하는 과정에 대해 설명한다.

Chapter 4. Debugging Tips

이 장에서는 Jasmine 보드에 설치된 펌웨어 코드의 동작을 검증하기 위한 몇 가지 디버깅 팁에 대해 소개한다.

Further reading

- RealView Debugger 에 대한 사용자 가이드는 http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.swdev.rvds/index.html 에 제공되는 관련 문서를 참고하도록 한다.

Feedback

- OpenSSD 프로젝트 위키 페이지 - http://www.openssd-project.org/wiki/

Chapter 1.
Getting Started to Develop an FTL

본 장에서는 Jasmine OpenSSD 플랫폼을 위한 FTL 개발을 시작하기 앞서 필요한 개발 환경에 대한 이해와 환경 구축에 대해 소개한다. 이 장에서 설명하는 핵심 내용은 아래와 같다.

- ✓ FTL 개발을 위한 하드웨어 및 소프트웨어 요구사항 설명
- ✓ 개발 환경 구축 가이드 제공

1.1. Development Environment

1.1.1. Requirement specification

Jasmine OpenSSD 플랫폼 상에서 FTL 을 개발하기 위해서는 아래와 같은 환경이 요구된다.

H/W requirement

- Test PC 1 대
  - Jasmine Board 와 통신 수행
- Client PC 1 대
  - RVD(RealView Debugger) 또는 UART 를 이용한 디버깅 수행
- Jasmine Board
  - NAND 플래시 모듈이 장착된 본체, SATA 케이블, RS232 케이블 포함
- RealView In-Circuit Emulator(ICE) 장비 (*optional*)
  - ICE 본체, Ethernet/USB 케이블, JTAG 커넥터 케이블

S/W requirement

- 운영체제: Windows XP SP2
- 펌웨어 컴파일 소프트웨어
  - RVDS(RealView Development Suite) 3.0 이상
  - 또는 GNU Compilation toolchain (The Sourcery G++ Lite Edition)
- Microsoft Visual Studio 8
  - PATH 환경변수 추가: C:\Program Files\Microsoft Visual Studio 8\VC\
- Jasmine OpenSSD 플랫폼 펌웨어
- UART 사용시, 하이퍼터미널 프로그램 필요

1.1.2. Development environment setup

위 1.1.1 절의 요구사항을 토대로, 아래 Figure 1 와 같이 개발 환경을 구축한다.

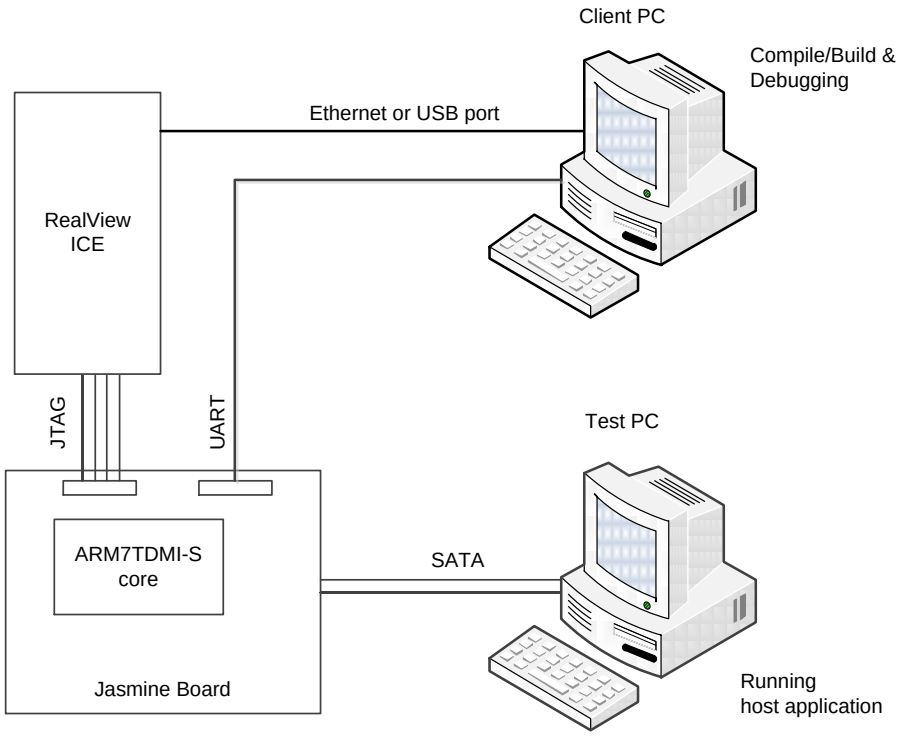


Figure 1) FTL Development Environment

- Test PC 과 Jasmine 보드를 전원 및 SATA 케이블로 연결한다.
- 디버깅 작업을 위해 RV ICE 장비를 Ethernet 또는 USB port 에 연결하고, RS232 시리얼 케이블로 Test PC 의 시리얼 포트와 Jasmine 보드의 UART 인터페이스를 연결한다.
- Client PC 에 RVDS, RV ICE 소프트웨어, 그리고 SSD 펌웨어 소프트웨어를 설치한다.
  - RV 디버거 설정을 통해 RV ICE 와 Jasmine 보드를 인식시킨다.

NOTE: 사실상 RV 디버거로 디버깅을 수행하지 않을 경우 – 즉 호스트 어플리케이션을 이용한 벤치마크 또는 단지 UART 포트를 사용하여 디버깅할 때는 – Client PC 를 따로 사용하지 않고 Test PC 에서 모든 작업을 수행하면 된다.

Chapter 2.
FTL Porting

본 장에서는 Jasmine OpenSSD 플랫폼에 새로운 FTL 기법을 포팅하기 위해 알아야 할 내용에 대해 살펴보도록 한다.

2.1. Porting Guide

FTL 은 동작과정에서 DRAM 과 플래시메모리에 존재하는 데이터를 읽거나 써야 하는 작업등을 수행한다. 본 절에서는 FTL 측면에서 이러한 하드웨어 컴포넌트를 접근할 때 주의해야 할 부분들에 대해 설명한다.

2.1.1. Implementation

FTL 포팅 시, 개발자는 FTL 프로토콜 API 를 포함한 아래 함수들을 구현해야 한다.

Source file	Function	Description
`./installer/installer.c`	`ftl_install_mapping_table`	FTL 초기 메타데이터를 NAND 플래시에 기록하는 연산 수행. 펌웨어 설치 시 함께 호출됨
`./ftl_[scheme]/ftl.c`	`ftl_open`	FTL 초기화 과정 수행 - NAND 플래시로부터 메타데이터 로드 및 초기화 - VBLK #0 에 포맷 마크가 기록되어 있지 않을 경우, `format` 함수 호출
	`format`	VBLK #0 와 FTL 메타 영역을 제외한 나머지 블록들을 삭제 - 포맷 마크 기록
	`ftl_read`	사용자 데이터 읽기 처리
	`ftl_write`	사용자 데이터 쓰기 처리
	`ftl_flush`	SATA idle/standby time 에 주기적으로 메타데이터를 flush 하는 연산 수행

2.1.2. DRAM host buffer management

SATA 인터페이스를 통해 전달된 실제 사용자 데이터는 DRAM 호스트 버퍼 (i.e. SATA read/write buffer)에 버퍼링된다. 읽기 요청의 경우, event queue 로부터 read ATA 커맨드를 전달받으면, FTL 은 FCP 명령을 플래시 메모리 컨트롤러에 전달하여 NAND 플래시로부터 데이터를 읽어 미리 할당해 놓은 SATA read 버퍼 프레임에 올린다. 쓰기요청의 경우도, 호스트가 요청한 쓰기명령을 가져와 SATA write 버퍼에 데이터가 도착하면 해당 데이터를 플래시메모리에 기록하게 된다.

한편, DRAM 호스트 버퍼의 관리는 SATA, FTL, 그리고 하드웨어 버퍼 매니저 간의 포인터 조정을 통해 이루어 지는데, 플래시 메모리와 IO 성능과 SATA 의 bandwidth 차이로 인해 버퍼 포인터 충돌 현상이 발생할 수 있다. 따라서 FTL 포팅 시, `ftl_read/write` 동작과정에서 FTL 과 SATA 의 버퍼 포인터가 충돌하지 않도록 주의해서 구현해야 한다. 또한 DRAM 호스트 버퍼는 원형 큐 방식으로 동작하기 때문에, 버퍼 포인터는 반드시 단조 증가하도록 조정해야 한다.

NOTE: FTL 포팅 시 LLD 라이브러리를 사용하여 구현할 경우에는 LLD 에서 DRAM 호스트 버퍼의 포인터 조정을 수행하기 때문에 별도로 신경쓰지 않아도 된다. 다만, 펌웨어 개발자가 FCP 를 직접 설정하여 플래시 컨트롤러에 명령을 전달할 경우에는 위에서 언급한 사항들을 유의해야 할 것이다.

2.1.3. DRAM access limitation

Barefoot 컨트롤러는 DRAM 데이터의 신뢰성을 높이기 위해, 하드웨어 ECC 엔진을 사용한다. 이러한 이유로, CPU 가 DRAM 상에 있는 데이터를 직접 변경 하거나 읽으려고 하면, ECC parity 정보로 인해 주변 메모리 데이터가 손실되거나 잘못된 데이터를 읽을 수 있다. 따라서, 제공되는 하드웨어 Memory utility(`./include/mem_util.h`)을 사용하여 DRAM 데이터에 접근해야 한다. 예를 들어, DRAM 에 존재하는 메타데이터를 변경하기 위해서는 `read_dram_xx` 와 `write_dram_xx` 등의 라이브러리를 사용하여, DRAM 에서 SRAM 으로 데이터를 읽어온 후, SRAM 에서 변경한 데이터를 DRAM 에 다시 반영하는 일련의 과정들을 수행해야 한다 (또는 `mem_copy` 를 사용).

덧붙여, `mem_copy` 유틸리티를 사용하여 자주 참조되는 DRAM 데이터를 SRAM 에 캐싱하여 사용하면, 경우에 따라 향상된 성능을 기대할 수 있다.

다음은 DRAM 메모리 작업에 있어 숙지해야 할 몇 가지 제약사항을 정리한 것이다.

1. 기본적으로 128 Byte 당 4 Byte ECC parity 정보가 추가된다.
2. 따라서, 가용가능한 DRAM 영역은 64MB * 128 / 132 = 65075200 Bytes 가 된다.
3. DRAM-to-DRAM, SRAM-to-DRAM 메모리 작업을 할 경우에는 반드시 하드웨어 Memory utility 를 사용하도록 한다.
4. DRAM-to-DRAM 메모리 복사를 수행할 경우에는 복사할 크기는 반드시 `DRAM_ECC_UNIT`(128 Byte) 크기에 정렬되어 있어야 한다.
5. 따라서, DRAM 에 선언할 메타데이터들의 시작 주소는 `DRAM_ECC_UNIT` 또는 `BYTES_PER_SECTOR` 크기로 정렬되도록 하는 것이 바람직하다.

2.1.4. NAND configuration

Barefoot 컨트롤러는 플래시 메모리 내 VBLK #0 을 펌웨어 바이너리 이미지와 메타정보(e.g. scan list)를 저장하는 등의 목적으로 사용한다. 따라서, FTL 은 이 VBLK #0 을 제외한 나머지 블록들을 사용자 데이터 및 메타데이터 저장 용도로 사용해야 한다.

한편, 학계에 제안된 많은 FTL 기법은 POR/SPOR 측면을 고려하여 FTL 메타 데이터를 플래시 메모리의 보조 영역(spare area)에 기록하는 방식을 사용한다. 하지만 Barefoot 컨

트롤러는 플래시 메모리의 보조 영역을 사용할 수 없다는 제약이 있다. 따라서 FTL 은 메타 데이터를 별도의 블록을 할당하여 기록해야 한다.

2.1.5. Flash command

FTL 이 플래시 메모리에 IO 요청을 전달하는 과정을 간략히 설명하면, 먼저 FTL 은 FCP(Flash Command Port)에 플래시 명령을 설정하여 WR(Waiting Room)에 전달한다. 그리고 WR 에 해당 요청이 잠시 대기하였다가, NAND 플래시 컨트롤러가 bank 의 상태를 확인한 후, 유휴상태일 때 플래시 명령을 BSP(Bank Status Port)에 전달하게 된다.

NOTE: FCP, WR, 그리고 BSP 에 대한 설명은 Technical Reference Manual 을 참고 할 것.

한편, WR 의 queue depth 는 '1' 이기 때문에, 새로운 플래시 명령을 전달하기 전에, WR 에 큐잉되어 있는 명령이 있는지 항상 확인해야 한다. 만약, WR 에 이미 기존 명령이 대기하고 있는 상태에서 FTL 이 새로운 플래시 명령을 issue 하면, 기존 명령이 수행되지 않을 수도 있는 등의 오작동이 발생할 수 있으므로 주의해서 구현해야 한다.

NOTE: `./target_spw/flash.c` 의 `flash_issue_cmd` 함수를 참고 할 것.

Auto-select mode (*ADVANCED*)

Auto-select mode 는 쓰기 연산 시, 하드웨어에 의해 자동적으로 유휴 상태의 bank 를 선택하게 함으로써, IO 병렬화를 극대화 할 수 있다. 이 모드로 동작하기 위해서는 FCP 의 FCP_BANK 레지스터를 0x3F 로 설정한 후 전달하면 된다.

2.2. Porting Example – Greedy FTL

Jasmine 펌웨어에는 Tutorial FTL 외에 간단한 가비지 콜렉션 연산을 수행하는 페이지 맵핑 FTL 인 'Greedy FTL' (`./ftl_greedy/ftl.c`)이 구현되어 있다. Greedy FTL 은 가비지 콜렉션 기능 뿐만 아니라, normal powe-on/off 가 가능하도록 Power-Off Recovery (POR) 기능도 구현되어 있다. 이 장에서는 Greedy FTL 의 구현 사항을 중심으로 FTL porting 시 구현해야 할 기능들에 대해 설명한다.

2.2.1. FTL initialization

Jasmine 보드의 하드웨어 컴포던트가 초기화가 끝나면 ftl_open 함수를 호출하여 FTL 을 초기화하는 연산을 수행한다. 아래는 Greedy FTL 의 초기화 과정에 대한 설명이다.

1. `build_bad_blk_list` 함수를 호출하여 VBK #0 에 기록되어 있는 scan list 를 읽어와서 bad block bitmap 테이블을 작성.
2. VBLK #0 에 format 마크가 기록되어 있지 않을 경우, format 연산을 수행.
   - format 함수는 SRAM/DRAM 메타데이터를 초기화 하고, VBLK #0 를 제외한 나머지 영역을 삭제하는 연산을 수행하는 기능을 담당.

- 초기화된 메타데이터를 NAND 플래시의 메타 영역에 기록하고, VBLK #0 에 format 마크를 기록.
- 만약, format 을 수행할 필요가 없을 경우, 즉 첫번째 부팅이 아닐 경우에는 NAND 플래시로부터 FTL 메타데이터를 로드.

3. FTL 버퍼 포인터를 초기화.

2.2.2. SRAM metadata

SRAM 에 관리하는 FTL 메타데이터는 매우 빈번히 참조되며 비교적 적은 크기의 정보이다. Greedy FTL 은 FTL 메타 영역과 사용자 데이터 영역에 대한 쓰기 page index pointer 및 free block count, 그리고 통계정보등을 SRAM 에 관리한다.

2.2.3. Address mapping

Greedy FTL 은 Tutorial FTL 과는 다르게, LPN 에 대한 고정 bank 방식으로 주소맵핑이 구현되어 있다. 다시 말하면, 특정 LPN 은 항상 특정 bank 에서만 read/write 연산이 발생하도록 되어 있다. LPN 에 따른 target bank 는 NUM_BANKS 로 modular 연산을 수행하여 구하며, 맵핑 정보는 VPN 정보만을 관리한다.

이러한 특성으로 random access IO 가 발생할 때, 병렬성이 떨어지는 단점이 존재할 수 있지만, sequential access IO, 특히 sequential read 연산일 경우에는 Tutorial FTL 보다는 좋은 성능을 기대할 수 있다.

2.2.4. DRAM metadata

FTL 이 관리해야 하는 메타데이터 크기가 가용한 SRAM 공간보다 클 경우, DRAM 에 별도의 메타 데이터 메모리 영역을 확보해줘야 한다. 이를 위해 `ftl.h` 의 '*DRAM segmentation*' 부분을 수정해야 한다.

한편, Greedy FTL 이 관리하는 DRAM 메타데이터는 아래와 같다.

Table 1) DRAM metadata for Greedy FTL

Metadata	Description
BAD_BLK_BMP	scan list 를 통해 얻어진 bad block list 에 대한 bitmap 테이블.
PAGE_MAP	논리 페이지 번호로 물리 페이지 번호 맵핑을 위한 페이지 맵핑 테이블. PAGE_MAP 의 크기는 (*전체 논리 블록 개수 X 블록 당 페이지 개수 X 4B*) 이다
VCOUNT	블록 내 유효페이지 개수를 관리하기 위한 메타데이터로써, 가비지 콜렉션 시 희생 블록을 선정하기 위해 관리하는 메타데이터. VCOUNT 의 크기는 (*bank 개수 X bank 당 virtual 블록 개수 X 4B*) 이다

2.2.5. NAND Configuration

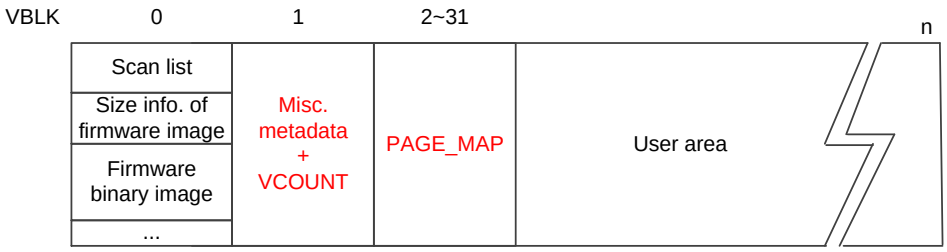


Figure 2) NAND configuration in Greedy FTL

Greedy FTL 은 NAND 플래시를 위 Figure 2 처럼 관리한다. POR 을 위한 FTL 메타 로깅 영역을 사용하여, 이를 제외한 나머지 영역은 사용자 데이터를 기록하기 위한 용도로 사용한다. FTL 메타데이터는 메타 블록 내 순차적으로 기록된다.

한편, 사용자 영역 내 블록들은 아래 Figure 3 과 같은 구조를 갖는다. 블록 내 페이지 개수를 m 으로 가정하면, 페이지 0 부터 m-2 에는 사용자 데이터를 기록하고, 마지막 m-1 번째 페이지에는 페이지 0 부터 m-2 의 LPN 정보를 기록한다. 이러한 메타정보를 별도로 기록하는 이유는 Barefoot 컨트롤러 특성 상 펌웨어가 별도로 페이지 내 보조영역을 사용할 수 없기 때문이다. 이렇게 기록된 LPN 정보들은 추후 가비지 콜렉션 시, 블록 내 사용자 데이터의 유효성을 판별하는데 사용된다.

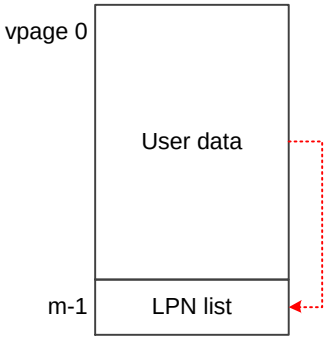


Figure 3) Block structure of user area in Greedy FTL

2.2.6. Garbage collection

Greedy FTL 의 가비지 콜렉션은 더 이상 새로운 데이터를 쓸 여유 공간이 없을 경우 수행한다. 희생 블록 선택 정책은 전체 블록에 대해서 유효 페이지의 개수가 가장 적은 블록을 택하며(`./ftl_greedy/ftl.c` 의 `get_vt_vblock()`), 이는 기본적으로 개별 가비지 콜렉션 시 페이지 복사 연산 비용을 최소화하기 위한 정책이다.

NOTE: Greedy 가비지 콜렉션 정책에 관한 자세한 내용은 아래 논문을 참고할 것.

`Atsuo Kawaguchi and Shingo Nishioka and Hiroshi Motoda, A Flash-Memory Based File System, USENIX Winter, pp. 155-164, 1995.`

아래 Figure 4 은 Greedy FTL 에서 garbage collection 연산을 보여준다.

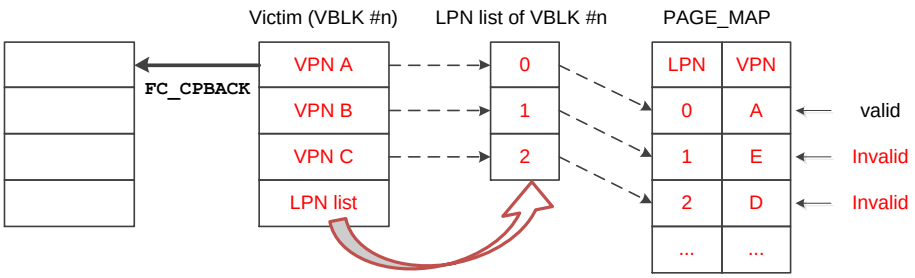


Figure 4) Garbage Collection in Greedy FTL

먼저, DRAM 내 VCOUNT 메타 테이블을 탐색하여 가장 낮은 VCOUNT 값을 갖는 블록을 희생 블록(VBLK #n)으로 선택한다. 그리고 블록 내 마지막 페이지에서 LPN 리스트를 읽는다. 희생 블록 내 데이터의 유효성 판별은 LPN 리스트 정보와 해당 LPN 에 대한 최신 데이터를 담고있는 VPN 정보가 일치하는지 PAGE_MAP 정보 비교를 통해 알아내게 된다. 그림에서 희생 블록의 첫번째 VPN 번호 A 에 기록된 사용자 데이터는 LPN 0 번에 대한 것이라고 알 수 있고, 이 데이터가 최신 데이터인지 판별하기 위해서 PAGE_MAP 의 LPN 0 에 맵핑된 VPN 번호가 A 임을 확인함으로써 유효하다는 것을 판별하게 된다. 만약 VPN B 와 C 의 경우와 같이 주소 맵핑 정보와 일치하지 않을 경우에는 해당 VPN 의 데이터는 invalid 하다는 사실이 자명하게 된다.

이렇게 판별한 유효 페이지들은 LLD 라이브러리의 copy-back 함수를 이용하여 미리 할당한 빈 블록에 복사해준 후, 기존 블록을 삭제한다. 그리고 변경된 FTL 메타정보를 반영하기 위해 PAGE_MAP 과 VCOUNT 메타 테이블을 갱신함으로써 garbage collection 연산을 마친다. 이후, 해당 블록부터 새로운 데이터를 써 나가게 된다.

2.2.7. POR (Power-Off Recovery)

Greedy FTL 은 Jasmine 보드가 정상적으로 종료되었을 경우에서 POR 을 지원한다. 호스트로부터 전달받은 IO 요청이 더 이상 없는 경우, 즉 SATA idle/standby 인 상태에 Figure 2 에서 설명한 FTL 메타영역에 전체 FTL 메타데이터를 NAND 플래시에 로깅한다 (ftl_flush 함수 참고).

이후, 정상적으로 종료되고 부팅되면 format 을 수행하지 않고, FTL 메타 영역에 로깅된 FTL 메타데이터를 로드하여 사용자 IO 요청을 처리할 준비를 한다 (load_metadata 함수 참고).

2.3. How to verify FTL operations?

2.3.1. FTL logic test

FTL 동작을 논리적으로 검증하기 위해서는 기본적으로 쓰기연산을 수행한 후, 동일한 영역에 대한 읽기 연산을 수행하여 두 버퍼 데이터를 비교함으로써, FTL 동작에 버그가 없는지 확인해 볼 수 있다. 이를 구현하기 위한 방법은 다음과 같은 두 가지가 있을 수 있다.

1. 별도의 검증 프로그램을 사용하거나, 호스트 어플리케이션(e.g. IOmeter)에 read-after-write 연산을 수행하는 코드를 삽입하여 검증한다.
2. 펌웨어의 FTL 테스트 코드를 사용한다.

- `./include/jasmine.h` 의 `OPTION_FTL_TEST` 를 1 로 설정하여 FTL 내부 코드를 테스트하도록 한다. 이 옵션을 설정하면 Jasmine 초기화 후, `ftl_test` 함수를 호출하여 SATA 를 bypass 하여 FTL 코드를 검증할 수 있다. 이 함수는 기본적으로 `ftl_write` 함수를 연속하여 수행한 후에, `ftl_read` 함수를 호출하여 동일한 데이터 인지 비교하는 연산을 수행한다.
- 좀더 다양한 경우에 대해 테스트를 하기 위해서는 펌웨어 코드에 다양한 test case 를 구현하여 검증하도록 한다.

2.3.2. POR test

FTL 의 POR 코드의 정상 동작 여부를 간단히 테스트를 위해서는 `OPTION_FTL_TEST` 를 1 로 설정하고, 펌웨어를 부팅시켜 `ftl_test` 를 수행하게 한다. 이후 전원 버튼을 껐다 다시 킴으로써 명시적으로 POR 코드를 수행하도록 하여, `ftl_test` 를 재수행하게 하는 방법으로 테스트를 수행할 수 있다..

2.4. Setting up to build

Jasmine 펌웨어에 포팅한 FTL 기법을 build 하기 위해서는 ./ftl_xxx 의 형태로 폴더를 생성하여, 해당 FTL 관련 헤더파일과 소스코드 파일을 추가시켜야 한다.

기본적으로 build 폴더(`./build_rvds` 또는 `./build_gnu`)의 `file_list.via` 에는 컴파일할 펌웨어 소스파일 리스트가 작성되어 있다. 만약, `ftl.c` 외에 다른 소스파일이 존재한다면 `file_list.via` 에 해당 소스파일을 추가시켜 함께 컴파일될 수 있도록 하자.

또한 Jasmine 관련 헤더파일(`./include/jasmine.h`)의 컴파일 옵션을 수정하도록 한다. 이러한 설정을 통해 Jasmine 보드에 장착된 NAND 플래시 칩 종류, bank 구성, 2-plane 모드, DRAM 크기, clock 동작 속도 등을 결정할 수 있고, 또한 FTL 테스트 모드, assert 검증, UART 디버깅, SATA 2.0/1.0, SATA NCQ 등을 활성화/비활성화할 수 있다.

이렇게 build 설정을 마쳤으면 다음 Chapter 3 을 참고하여 펌웨어를 build 하고, 생성된 펌웨어 바이너리 이미지를 Jasmine 보드에 설치하도록 한다.

Chapter 3.
Compile, Build & Install Firmware

본 장에서는 펌웨어 컴파일 및 build 하는 과정과 Jasmine 보드에 펌웨어를 설치하는 과정에 대해 살펴본다.

3.1. Compile & Build Firmware

먼저, 아래 경로에서 Jasmine OpenSSD 플랫폼의 최신 펌웨어 소스파일을 Client PC 에 다운로드한다.

- http://www.openssd-project.org/wiki/Downloads

Jasmine OpenSSD 플랫폼은 두 가지 컴파일 환경을 지원한다. 첫 번째는 RV ICE 장비 및 RVDS 를 사용하는 방법과 다른 하나는 GNU tool-chain 을 사용하는 방법이다.

3.1.1. Build firmware using RVDS tool-chain

ARM RVDS 를 사용하여 펌웨어를 build 하기 위해서는 Client PC 에 RVDS 3.0 이상이 설치되어 있어야 한다.

NOTE: 참고로, 정식버전의 RVDS 를 구입하지 않았다면 ARM Ltd., 홈페이지 (www.arm.com)에 등록한 후, ARM RVDS 4.1 Professional 의 evaluation version 을 다운받을 수 있다. 참고로 Evaluation version 은 30 일동안 사용할 수 있으며, 만료된 이후에는 라이센스를 구입하여야 한다.

펌웨어 build 는 명령창에서 아래와 같이 명령을 수행하면 된다.

```
> cd ./build_rvds
> build.bat [tutorial | greedy]
```

위와 같이 펌웨어 build 과정을 마치면 해당 폴더에 펌웨어 바이너리 이미지인 `firmware.bin` 파일이 생성된다.

NOTE: 만약 build 시 아래 에러가 발생할 경우, 백신 프로그램을 중단 후 수행할 것.
mt.exe : general error c101008d: Failed to write the updated manifest to the resource of file...

3.1.2. Build firmware using GNU tool-chain

또한 Jasmine OpenSSD 플랫폼은 GNU 컴파일 툴을 사용해서 펌웨어를 build 할 수 있다. 먼저 아래 CodeSourcery 홈페이지에서 최신버전의 Sourcery G++ Lite Edition 을 다운받고, Client PC 에 설치한다.

- http://www.codesourcery.com/sgpp/lite_edition.html

기본적으로 Makefile 은 Tutorial FTL 을 build 하게 설정되어 있다. 따라서 포팅한 새 FTL 기법으로 펌웨어를 build 하기 위해 `Makefile` 의 첫번째 줄을 다음과 같이 수정한다.

```
FTL = new_scheme
...
```

그리고 명령창에서 아래와 같이 명령을 수행한다.

```
> cd ./build_gnu
> build.bat
```

위와 같이 펌웨어 build 과정을 마치면 해당 폴더에 펌웨어 바이너리 이미지인 `firmware.bin` 파일이 생성된다.

3.1.3. Compile firmware installer

펌웨어의 설치는 `install.exe` 프로그램에 의해 수행된다. `install.exe` 파일을 생성하기 위해서는 먼저, 아래 경로에서 Visual C++ 2010 Express Free Edition 을 다운받아 설치한다.

- http://www.microsoft.com/express/Downloads/#2010-Visual-CPP

이후 Visual C++ 2005 솔루션 파일(`./installer/installer.sln`)을 build 하면 해당 경로에 `install.exe` 파일이 생성된다. 한편, 이렇게 생성된 펌웨어 인스톨러를 사용하기 위해, build 하고자 하는 FTL 경로(e.g. Tutorial FTL 을 설치하고자 할 경우, `./ftl_tutorial/`)로 옮기도록 한다.

NOTE: Jasmine 보드의 channel/way configuration 을 변경할 경우에는, `./include/jasmine.h` 의 BANK_BMP 를 변경 후, installer 를 반드시 re-build 하여 사용하도록 한다..

3.2. Install Firmware

위 3.1 절에서 build 한 펌웨어 바이너리 이미지를 `install.exe` 를 사용하여 Jasmine 보드에 설치하도록 한다. 펌웨어를 설치하는 과정은 다음과 같다.

1. Jasmine 보드를 'Factory mode'로 부팅
   - Factory mode 부팅을 위해서는 Jasmine 보드의 J2 점퍼를 아래와 같이 설정하고, 전원 및 SATA 케이블을 연결하고 전원 스위치를 켠다.

- Factory mode 로 부팅이 되면, Jasmine 보드가 호스트 PC 의 장치관리자–디스크 드라이브에서 '*YATAPDONG BAREFOOT-ROM*'으로 인식된다.

2. 인스톨러(`install.exe`)를 실행하여 펌웨어를 Jasmine 보드에 설치

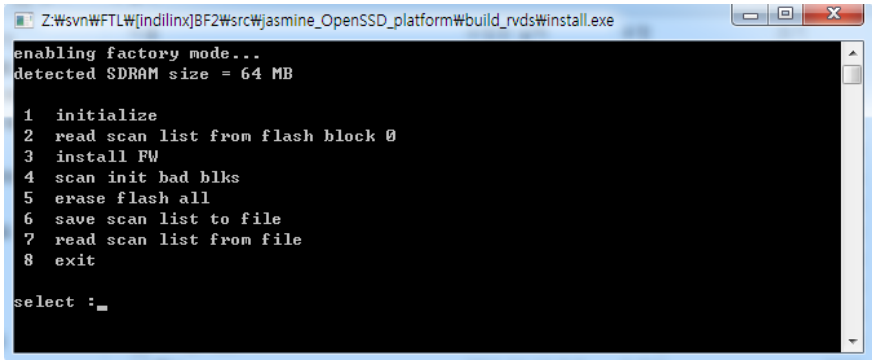


Figure 5) Jasmine Firmware Installer

- 각 메뉴에 대한 설명은 다음과 같다.
  1. `initialize`
     Jasmine 보드를 초기화하는 작업을 수행한다.
  2. `read scan list from flash block 0`
     Jasmine 보드에 장착된 bank 0 번째 NAND 플래시 칩의 block 0 에 설치된 bad block list 를 로드한다.
  3. `install FW`
     펌웨어를 Jasmine 보드에 설치한다.
  4. `scan init bad blks`
     전체 NAND 플래시 메모리를 스캔하면서 bad block list 를 작성한다.
  5. `erase flash all`
     전체 NAND 플래시 칩의 모든 블록을 삭제한다. 이 때, block 0 에 설치된 펌웨어 및 bad block list 도 함께 삭제되므로 이점 유의하여야 한다.
  6. `save scan list to file`
     메뉴 2 또는 4 를 통해 작성된 bad block scan list 를 작업 PC 에 파일의 형태로 저장한다. 이 때 저장할 폴더 명을 입력받게 된다.
  7. `read scan list from file`
     메뉴 6 을 통해 PC 에 저장된 scan list 파일을 읽어서 bad block list 를 작성한다.
  8. `exit`
     인스톨러를 종료한다.

- 한편, Jasmine 보드는 출고 시점에 이미 NAND 플래시 메모리에 scan list 가 설치되어 있다. 따라서, 펌웨어를 설치하기 전에 scan list 를 PC 에 백업받는 작업이 선행되어야 한다. 이는 메뉴 `1-2-6-3` 순서로 수행하면 된다.
- 만일 block 0 에 설치된 scan list 가 손상되었다면, 메뉴 2 를 수행하는 과정에서 에러가 발생하게 된다. 따라서, PC 에 저장된 scan list 파일을 로드한 후에 펌웨어를 설치하도록 한다. 이는 메뉴 `1-7-3` 순서로 수행하도록 한다.
- 만약 새로운 NAND 플래시 모듈을 Jasmine 보드에 설치했을 경우, NAND 플래시 메모리의 bad block 을 scan 하여 scan list 를 저장한 후에 펌웨어를 설치하도록 한다. 이는 메뉴 `1-4-6-3` 순서를 따르면 된다.

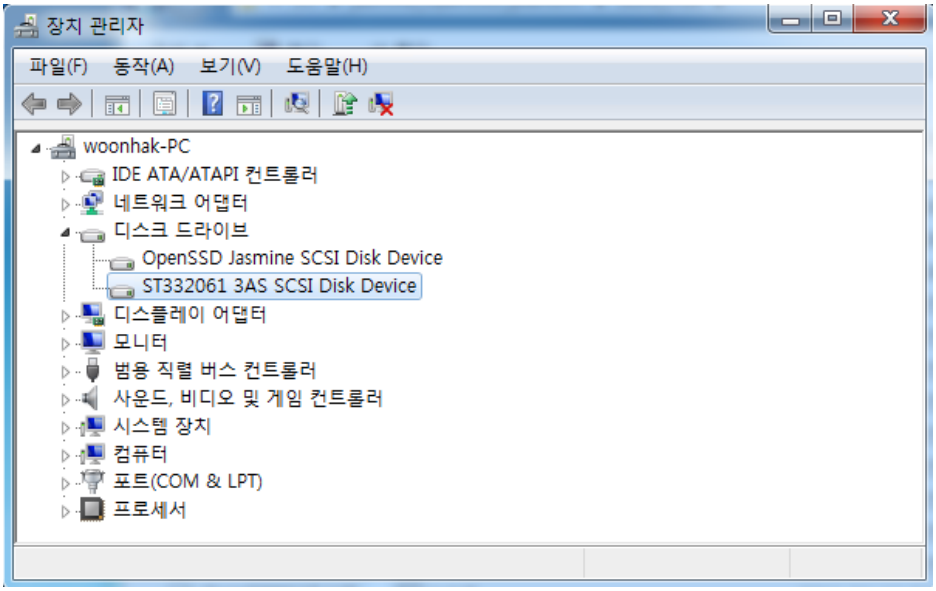
펌웨어 설치가 정상적으로 완료되면 아래와 같이 Normal mode 로 Jasmine 보드의 J2 점퍼를 설정한다.

Normal Mode (Default)

그리고 SATA 케이블을 뺀 상태에서 전원 스위를 켠다. 이 과정에서는 FTL 포맷이 수행되며, 포맷이 완료되면 D4 위치의 LED 가 켜지게 된다. 이후 SATA 케이블을 연결하면 Jasmine 보드는 호스트로부터 SATA 커맨드를 수행할 준비가 완료된다.

NOTE: Jasmine 보드에 전원이 들어오면 내부적으로 `ftl_open` 함수가 호출되는데, 구현에 따라 해당 연산을 수행하는데 긴 시간이 소요될 수 있다. 이로 인해 호스트에서 *response time-out error* 가 발생할 가능성이 있기 때문에, `ftl_open` 이 완료되는 시점, 즉 D4 위치의 LED 가 켜진 후에 SATA 케이블을 연결하도록 한다.

Jasmine 보드에 펌웨어 설치가 정상적으로 완료되면 아래 그림과 같이 장치 관리자 - 디스크 드라이브에서 '*OpenSSD Jasmine*'으로 인식되는 것을 확인할 수 있고, 이제 완전한 플래시 SSD 로 사용 가능하게 된다.



Chapter 4.
Debugging Tips

Jasmine OpenSSD 플랫폼은 UART 인터페이스를 통한 메시지 출력과 ICE 장비와 RV Debugger 를 사용한 실시간 펌웨어 디버깅이 가능하다. 본 장에서는 이러한 두 가지 방법으로 Jasmine 보드에 설치한 FTL 의 동작을 검증하는 방법에 대해 설명한다.

4.1. Debugging with UART

본 절에서는 Jasmine 보드의 UART 인터페이스를 통해 터미널 창에 출력되는 정보를 활용하여 디버깅하는 과정에 대해 설명한다.

4.1.1. Debugging setting

먼저, Jasmine 보드의 UART 포트(P1)와 Client PC 의 시리얼 포트를 RS232 케이블로 연결한다. 그리고 UART 인터페이스를 사용하기 위해 Jasmine 보드의 온-보드 스위치(SW2,3,4)를 아래와 같이 설정한다.

- SW2: No. 1,2,3,4 (ON)
- SW3: No. 1,2,3,4 (OFF)
- SW4: No. 1,2,3,4 (OFF)

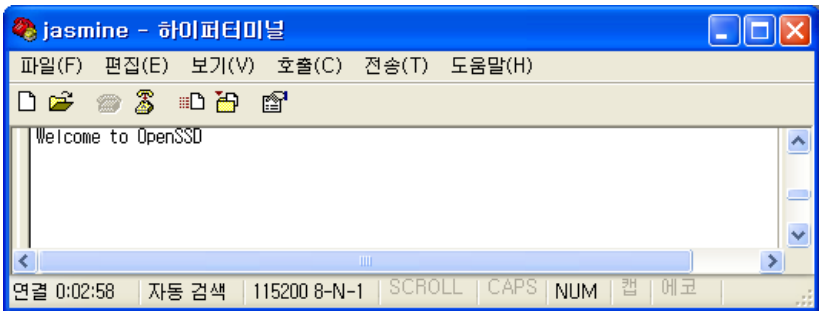
Jasmine 보드 설정이 끝났으면 터미널 프로그램의 시리얼 포트를 아래와 같이 설정한다.

- 비트/초(Baud rate): 115200
- 데이터 비트: 8
- 패리티: 없음
- 정지비트: 1
- 흐름제어: 하드웨어 (또는 Yes)

UART 인터페이스를 통해 메시지 출력을 받으려면 펌웨어에서 `OPTION_UART_DEBUG` 를 활성화하여야 한다. `./include/jasmine.h` 의 `OPTION_UART_DEBUG` 를 1 로 설정한다.

4.1.2. Debugging by printing message

UART 포트를 활성화했다면 UART 메시지 프린트 함수(`./target_spw/uart.c` 의 `uart_print`) 를 이용하여 오류 발생 시 특정 메모리 영역을 dump 한다거나, 디버깅 메시지를 출력하는 등의 방법으로 펌웨어의 동작을 디버깅 할 수 있다. Jasmine 보드의 UART 포트가 정상적으로 초기화 되었고, Normal mode 로 Jasmine 보드가 부팅되면 아래와 같은 메시지가 터미널 창에 출력된다.



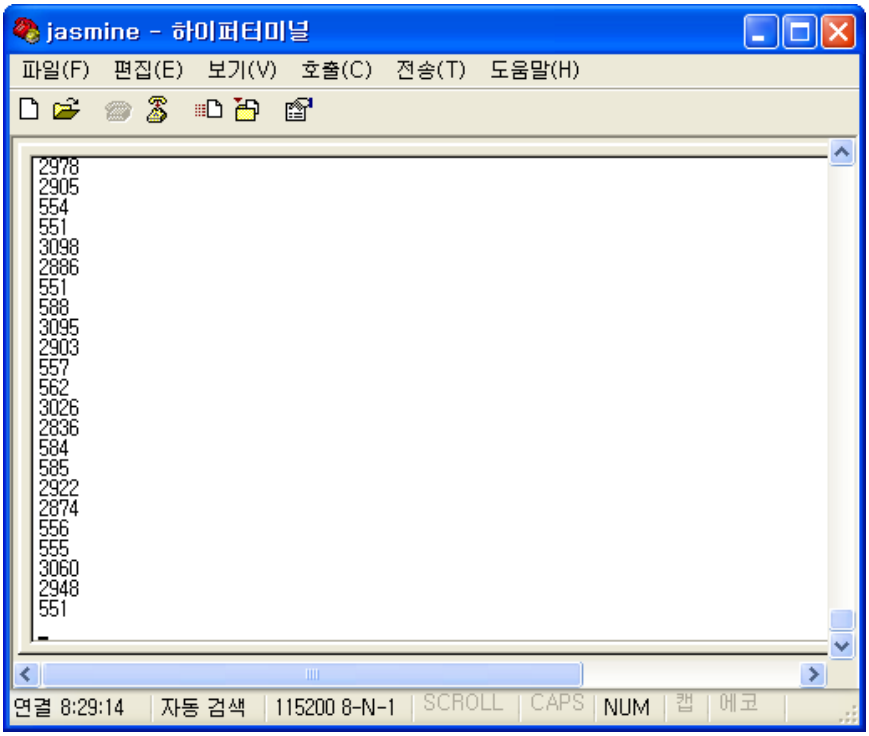
4.1.3. Measure Response Time

Jasmine 펌웨어는 Timer 를 이용한 FTL 성능 측정 관련 함수도 제공한다. 이 함수를 이용하여 FTL 읽기/쓰기 그리고 가비지 콜렉션에 대한 오버헤드를 측정하여 측정된 지연 시간 또는 응답 시간을 UART 포트로 출력함으로써 FTL 의 논리적 오류를 디버깅할 수 있다.

아래 코드처럼 Timer 관련 함수(`./target_spw/misc.c` 의 `ptimer_start`, `ptimer_stop`, `_uart_print`)를 이용하여 `ftl_write` 수행 시, 응답 시간을 측정할 수 있다.

```
ptimer_start();
ftl_write(lba, num_sectors);
ptimer_stop_and_uart_print();
```

아래 그림은 FTL 테스트 함수(`ftl.c` 의 `ftl_test`)에 위 코드를 삽입하여 UART 를 통해 출력되는 응답 시간(단위: `us`)을 보여준다.



4.2. Debugging with RVD

본 절에서는 RealView 디버거와 RealView ICE 장비를 이용하여 펌웨어 디버깅하는 과정에 대해 설명한다.

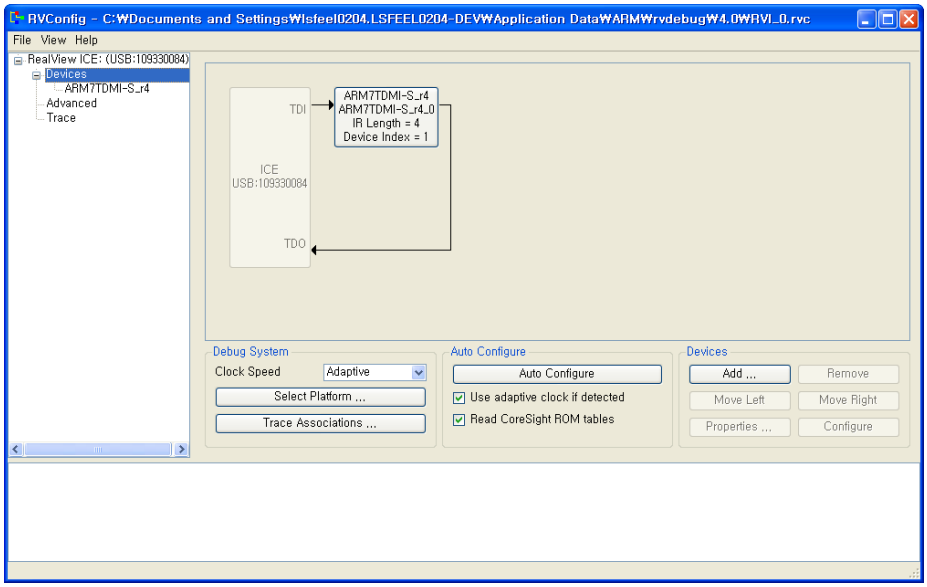
NOTE: 호스트 어플리케이션이 Jasmine 보드에 전달한 IO 명령을 처리하는 도중에 명시적으로 breakpoint 를 걸어 펌웨어 동작을 멈추면, busy waiting 이 발생하여 운영체제가 멈추게 될 가능성이 있다. 따라서 Client PC 를 별도로 사용하여 Client PC 에서 디버깅을 수행한다.

4.2.1. Debugging setting

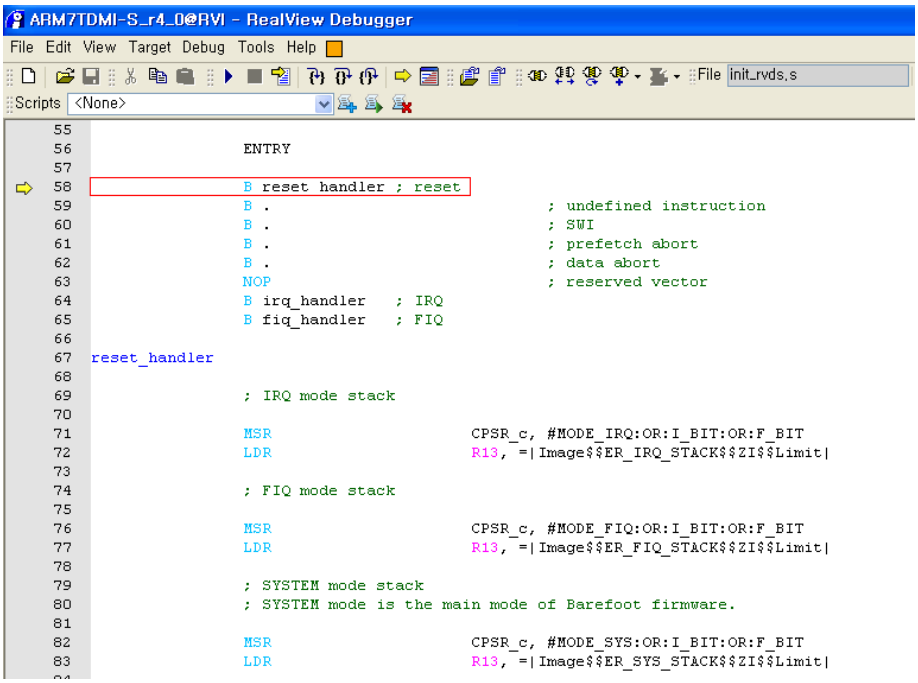
먼저, 정확한 line-by-line 디버깅을 위해 펌웨어의 컴파일 환경 설정 파일을 수정해야 한다. RVDS 컴파일 환경 설정 파일인 `./build_rvds/armcc_opt.via` 파일을 열어 아래와 같이 두 옵션을 수정한 후, 펌웨어 build 를 수행하도록 한다.

```
-O3 // -O1 으로 수정
-Otime // 삭제
```

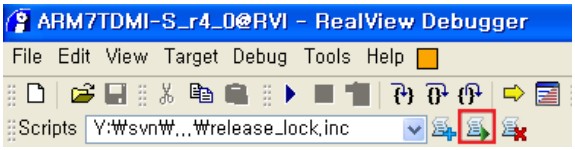
계속해서, Figure 1 처럼 RVD 설정을 수행한 후, 아래 그림과 같이 RV 디버거의 설정을 통해 ICE 장비와 Jasmine 보드가 정상적으로 연결되었는지 확인한다.



모든 연결을 마치면, 메뉴에서 **`Target->Load Image`** 로 build 된 펌웨어 이미지를 로드 한다. 이미지가 정상적으로 로드 되면, 아래 그림과 같이 펌웨어 startup 코드 (`./target_spw/init_rvds.s`)가 보여지게 된다.



마지막으로 아래 그림과 같이 ./release_lock.inc 스크립트를 추가한 후, 실행하여 Jasmine 보드의 JTAG 포트를 열어주면 디버깅을 위한 설정이 모두 끝나게 된다.



4.2.2. Debugging tip #1 – "Use a breakpoint statement"

Jasmine 보드에 전원을 인가하면 startup 코드에 의해 ./target_spw/initialize.c 의 init_jasmine 함수가 가장 먼저 호출된다. 이 함수는 각종 하드웨어 컴포넌트 및 SRAM/DRAM 영역을 초기화하며 ftl_open 함수를 호출하고 나서 사용자 IO 요청을 기다리게 된다.

만약 전원이 들어온 후 사용자 IO 요청을 기다리는 함수가 호출되기 전에 펌웨어 버그로 인해 오류가 발생한 경우를 가정해보자. 아래 그림과 같이, 예를 들어 ftl_open 함수내에 참조할 수 없는 메모리 영역에 데이터를 기록하는 버그가 있다고 하자.

```
void ftl_open(void)
{
    *(UINT32*)0xFFFFFFFE = 10; // data abort 발생
    ...
}
```

위와 같은 경우는 Jasmine 보드에 전원이 인가되자마자 해당 코드가 수행되면서 *data abort* 인터럽트가 발생해 버리기 때문에, 이후 RV 디버거를 실행하여 디버깅을 하려고 해도 정작 문제가 발생한 코드 위치를 찾을 수 없게 된다.

따라서 이러한 문제를 해결하기 위해서는, 오류가 발생하기 전 위치에서 명시적으로 펌웨어 동작을 멈춘 후에 디버깅 작업을 해야 하는데, 이는 아래 그림과 같이 dummy while 문을 삽입하는 방법이 있다.

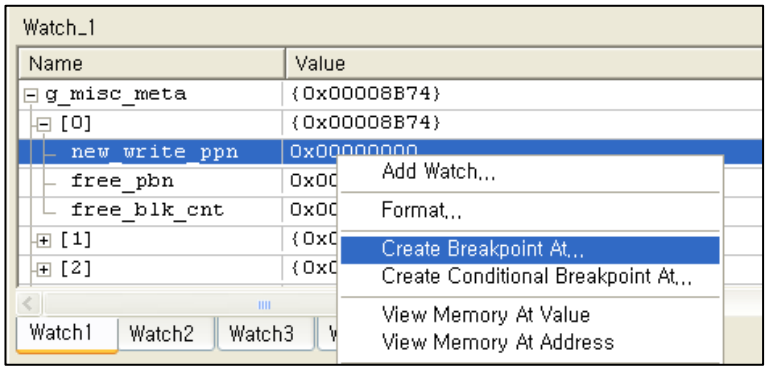
```
volatile UINT32 g_barrier;

void init_jasmine(void)
{
    ...
    g_barrier = 0;
    while (g_barrier == 0);

    ftl_open();
    ...

}
```

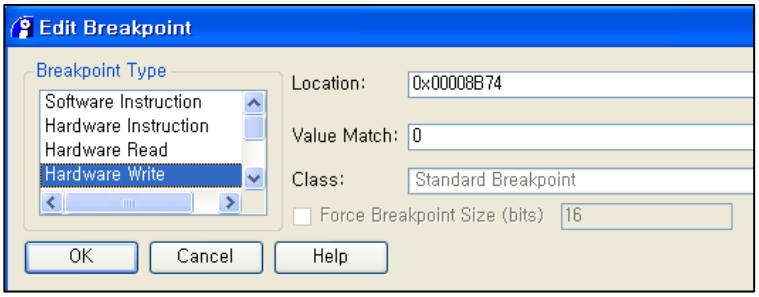
`ftl_open` 함수가 수행하기 전에 dummy while 문을 이용하여 펌웨어의 흐름을 멈춘 후, RV 디버거를 실행하여 g_barrier 의 값을 1 로 변경하고 line-by-line 디버깅을 진행할 수 있다.

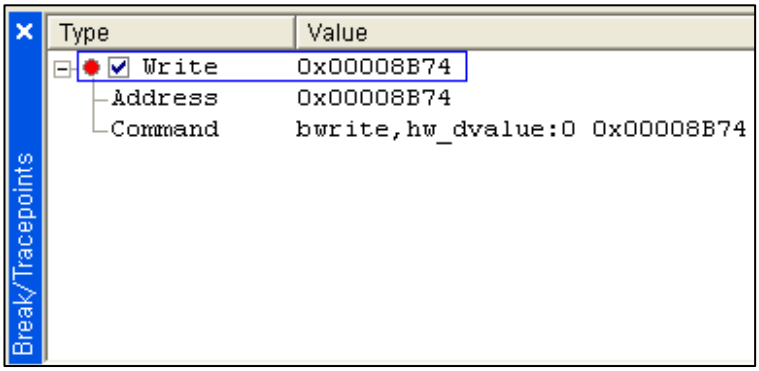
4.2.3. Debugging tip #2 – "Use a H/W breakpoint"

만약 FTL 코드에 논리적인 오류가 있을 경우, 메타 데이터 또는 버퍼의 메모리 값을 잘못 변경하는 문제가 발생할 수 있다. 이러한 경우에는 RV 디버거의 `hardware breakpoint'를 사용하여 디버깅을 수행할 수 있다.

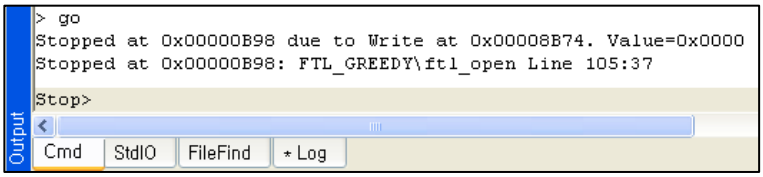
특정 메타 데이터 값이 '0'으로 변경되는 시점에 펌웨어 흐름을 멈추고 싶다면, 아래와 같은 순서로 해당 메모리 주소에 'hardware write' breakpoint 를 등록한다.







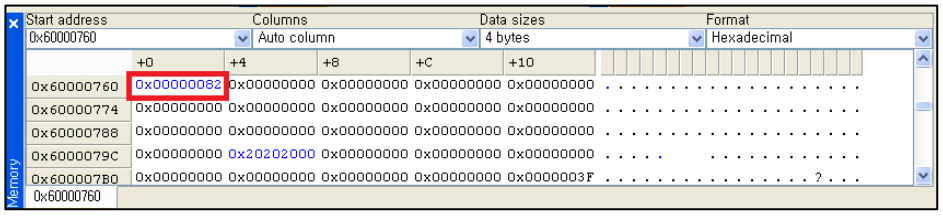
Breakpoint 를 등록한 후 디버깅을 재개하면 관찰 대상이 되는 메모리 주소에 '0'값을 쓰는 라인에서 펌웨어 흐름을 멈출 수 있어 버그를 찾을 수 있다.



4.2.4. Debugging tip #3 – "Watch status registers"

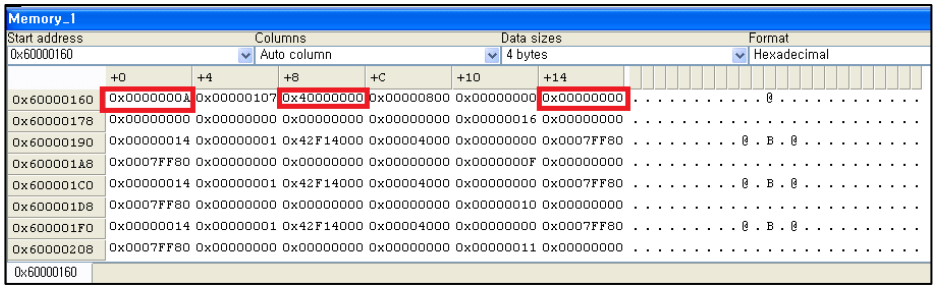
BSP 에는 바로 이전에 수행한 플래시 명령이 기록되고 오동작으로 인해 인터럽트가 발생할 경우, BSP_INTR 레지스터에 인터럽트 정보를 관찰함으로써 FTL 디버깅이 가능하다.

먼저 RV 디버거의 메뉴 **View->Memory** 로 메모리 창을 연 후, BSP_INTR 레지스터의 메모리 주소를 입력하면 BSP 인터럽트 정보를 관찰할 수 있다. 아래 그림처럼 bank 0 에서 FIRQ_DATA_CORRUPT(0x82) 인터럽트가 발생하였다는 것을 확인 할 수 있다.



또한, BSP 에 남아있는 플래시 명령을 확인하여 바로 이전에 수행한 플래시 명령의 종류 및 target bank, 블록, 페이지 번호, 버퍼 주소 등을 확인할 수 있다. 아래 그림을 살펴보면

페이지 읽기 연산인 `FC_NORMAL_READ_OUT(0x0A)` 도중에 인터럽트가 발생했다는 것을 알 수 있다.



NOTE: 메모리 값을 관찰할 때는 little endian 임을 주의해야 하며, BSP 레지스터 및 플래시 명령 매크로 및 DRAM 메모리 맵은 Technical Reference Manual 을 참고할 것.