碩士學位論文

# 임베디드 시스템을 위한 동적 업그레이드

2009 年

漢城大學校 大學院

컴퓨터 工學科

컴퓨터工學 專攻

慶 柱 顯

碩 士 學 位 論 文
指導敎授 李 珉 秭

# 임베디드 시스템을 위한 동적 업그레이드

## Dynamic Software Upgrade for Embedded Systems

2008年 12月 日


漢城大學校 大學院

컴퓨터 工學科

컴퓨터工學 專攻

慶 柱 顯

碩 士 學 位 論 文
指導敎授 李 珉 秔

# 임베디드 시스템을 위한 동적 업그레이드

## Dynamic Software Upgrade for Embedded Systems

위 論文을 컴퓨터工學 碩士學位論文으로 提出함

2008年 12月 日


漢城大學校 大學院

컴퓨터 工學科

컴퓨터工學 專攻

慶 柱 顯

慶柱顯의 工學 碩士學位論文을 인준함

2008年 12月 日

심사위원장 정 인 환 (인)

심사위원 이 민 석 (인)

심사위원 김 성 동 (인)

# 목 차

# 표 목 차

# 그 림 목 차

# 제 1 장 서론

## 1.1 연구 동기

모든 임베디드 시스템에서 소프트웨어의 규모가 커짐에 따라, 이미 현장에서 작동 중인 임베디드 시스템에서 오류의 발생 가능성이 높아지고 있으며, 지속적으로 기능 및 성능개선 요구가 있다. 임베디드 시스템에서 운영체제와 응용프로그램의 업그레이드는 버그수정, 성능향상, 기능추가 등의 이유로 빈번히 발생한다. 하지만 불행하게도 임베디드 시스템에서는 운영체제와 응용프로그램이 플래쉬 메모리 등에 내장되어 설치된 상태에서 제품으로 판매되기 때문에 새로운 버전의 운영체제 또는 응용프로그램의 재설치가 쉽지 않으며 제품 판매 후 업그레이드를 위해서는 많은 시간적 노력과 비용이 필요하다.

현재의 임베디드 소프트웨어 업그레이드 작업은 유무선 네트워크나 메모리 카드를 통해 새로운 버전의 소프트웨어를 대상 임베디드 시스템으로 다운로드하고 시스템을 재부팅시키는 방법으로 업그레이드 작업이 수행된다. 이 경우 시스템의 재부팅으로 인해 작업의 연속성이 끊겨 서비스의 일시 공급중단이라는 문제가 야기됨과 동시에 시간적, 비용적 낭비의 문제가 발생한다.

특히 서비스의 상시성이 요구되며, 사용자가 업그레이드와 관련된 기술적인 절차를 수행하기 어려운 환경에서는 소프트웨어 업그레이드를 최소한의 네트워크 트래픽으로 사용자가 알지 못하는 사이에 시스템 중단 없이, 즉 재부팅 없이 시스템이 서비스를 계속 진행하면서 수행하여야 한다.

또 시스템의 중단이 인명 및 상당한 비용 손실을 일으킬 수 있는 환경인 인공위성, 군용 임베디드 시스템, 보건의료 시스템, 원자력 발전소, 통신장비, 항공 트래픽 관제시스템, 생산라인 로봇, 등에서 시스템의 중단 없이 소프트웨어 업그레이드를 수행하는 것은 매우 중요하다. 따라서 이를 안전하고 효율적으로 처리할 수 있는 프레임워크 및 도구가 절실히 필요하다.

본 논문은 이러한 임베디드 시스템의 운영체제 커널 또는 응용프로그램을 업그레이드 할 때 시스템의 중단 없이 업그레이드를 수행하기 위한 플랫폼 독립적인 동적 업그레이드에 대한 프레임워크를 설계하였으며 OMA-DM 프로토콜을 사용한 함수 단위 동적 업그레이드 프로토타입 시스템을 구현하였다.

## 1.2 연구 목적

본 논문에서는 임베디드 시스템을 위한 동적 업그레이드에 대한 요구 사항을 분석하고 요구 사항을 기반으로 프레임워크를 설계와 프로토타입 시스템을 구현하는데 목적이 있다. 본 논문의 세부적인 목적은 다음과 같다.

- **동적 업그레이드 요구 사항 분석**
  기존 연구되어온 동적 업그레이드 시스템의 문제점과 임베디드 시스템에 적용하기 위한 요구 사항을 분석한다.

- **동적 업그레이드 프레임워크 설계**
  임베디드 시스템의 플랫폼 독립적인 동적 업그레이드 프레임워크를 설계한다.

- **동적 업그레이드 프로토타입 시스템 구현**
  동적 업그레이드 프레임워크를 기반으로 하여 임베디드 리눅스를 대상으로 동적 업그레이드 프로토타입 시스템을 구현한다.

## 1.3 논문 구성

본 논문의 구성은 다음과 같다.

- **2장 연구 배경**
  동적 업그레이드 시스템과 관련된 연구들에 대하여 기술한다.

- **3장 동적 업그레이드 프레임워크**
  동적 업그레이드를 위한 프레임워크 설계에 대하여 기술한다.

- **4장 프로토타입 시스템 구현**
  동적 업그레이드 프레임워크를 기반으로 프로토타입 시스템 구현에 대하여 기술한다.

- **5장 결과 및 토의**
  구현된 동적 업그레이드 시스템의 결과에 대한 문제점과 질문사항들에 대하여 기술한다.

- **6장 동적 업그레이드 시스템 활용**
  본 논문에서 기술한 동적 업그레이드 시스템에 대하여 실제 상용화 방안에 대하여 기술한다.

- **7장 결론 및 향후연구**
  본 논문에서 설계 및 구현한 시스템에 대한 결론을 내리며 향후 연구사항들에 대해 기술한다.

# 제 2 장 연구 배경

## 2.1 일반 소프트웨어 업그레이드

### 2.1.1 정적 업그레이드 기법

현재 까지 일반적으로 소프트웨어를 업그레이드 기법에는 대부분 정적 업그레이드 기법을 사용하여 업그레이드를 수행하고 있다. 이러한 정적 업그레이드 기법은 업그레이드 관리하는 프로그램이 파일 시스템 또는 저장소의 실행 이미지를 수정하여 업그레이드 하는 방법을 사용한다. 이러한 정적 업그레이드 기법에는 기계종속적인 정적 업그레이드 기법과 기계독립적인 업그레이드 기법으로 나누어진다.

**가. 기계종속적인 정적 업그레이드 기법**

기계종속적인 정적 업그레이드 기법은 profiling, tracing, memory debugging등에 사용되는 binary rewriting을 사용하여 업그레이드를 수행하는 기법이다. 이러한 기계어종속적인 정적 업그레이드 기법은 기계어의 제어 흐름을 분석하여 업그레이드를 수행한다. 대표적인 연구로는 EEL(James Larus외 1명, 1995), ATOM(Amitabh Srivastava외 1명, 1994)이 있다. 이러한 기계종속적인 정적 업그레이드 기법은 분석 단계에서는 중간 표현 형태를 사용하여 기계독립성을 보장할 수 있지만 그 하부 구조인 바이너리 코드 해석 및 다시 쓰기 부분은 기계 독립성을 보장할 수 없다. 그리고 제어 흐름 분석이 가능한 경우는 코드 영역이 있는 실행 파일 또는 라이브러리에서 국한된다(이민재, 2007). 또한 이러한 업그레이드는 업그레이드를 적용한 순서에 따라 다른 최종 업그레이드 이미지를 생성한다. 그러므로 업그레이드를 적용 속도는 빠르나 기계어종속적인 방법이므로 구현이 복잡하고 실제 업그레이드 버전관리가 어려운 단점이 있다.

### 나. 기계독립적인 정적 업그레이드 기법

초기 소프트웨어 업그레이드는 수정된 파일을 전송하여 새로운 파일로 교체하는 방법으로 업그레이드를 진행 하였으나 이러한 방법은 새로운 파일 전체를 받아야 하므로 클라이언트 저장 공간 문제와 서버의 네트워크 트레픽 문제를 가지고 있다. 이러한 문제점을 개선하기 위해 업그레이드를 할 때에는 두 이미지의 차이점만 저장하는 방법을 사용하여 업그레이드를 수행한다. 이러한 기계독립적인 정적 업그레이드 기법은 바이너리 파일을 일종의 문자열로 보고 문자열 매칭 알고리즘을 사용하여 패턴을 찾아내는 방식을 사용한다. 따라서 특정 플랫폼에 상관없이 사용할 수 있다. 이러한 바이너리 이미지의 차이점만 저장하는 것을 델타 패키지라고 부르며 델타 패키지의 사이즈를 줄이는 방향으로 연구가 진행되고 있다.

일반적인 델타 패키지는 스크립트 형태로 구성되어 있다. 델타 패키지의 스크립트 형태는 이전 버전의 이미지로부터의 복사할 때 사용하는 COPY 명령어와 새로운 데이터를 추가하는 INSERT 명령어로 구성되어 있다 (Chih-Chieh Han외 3명, 2005).

그림 2.1은 좌측 그림은 이러한 델타 패키지의 구성을 보여주고 있으며 COPY 명령어는 {previous image offset, current image offset, size}형태로 구성되어 있으며 INSERT 명령어는 {current image offset, size}형태와 INSERT 명령을 사용하여 추가할 데이터로 구성된다.

그림 2.1는 이러한 델타 패키지를 활용하여 업그레이드를 수행하는 과정을 보여준다. 델타 패키지의 명령어를 읽은 후 명령어를 해석하고 COPY 명령어이면 이전 버전의 이미지로부터 새로운 버전의 이미지로 복사를 하고 INSERT를 명령어이면 새로운 데이터를 새로운 버전의 이미지로 추가하는 장면을 보여주고 있다.

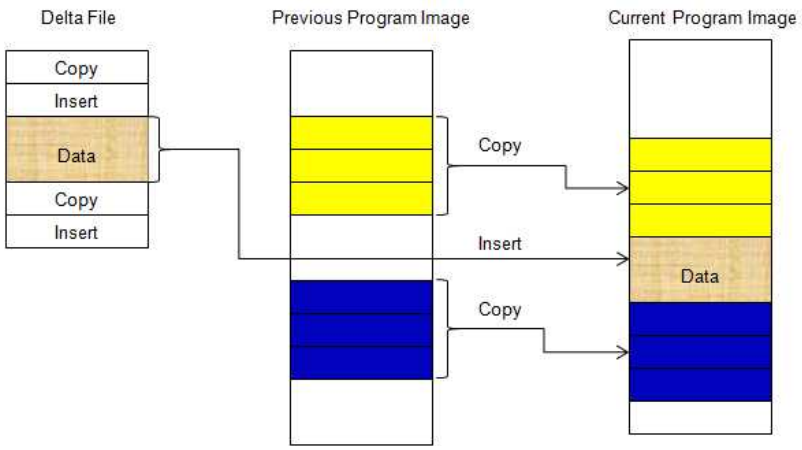



**그림 2.1 델타 업그레이드 수행 방법**

델타 패키지를 생성하는 대표적인 도구는 다음과 같다.

### (1) BSDIF

BSDIFF(Colin Percival, 2006)는 FreeBSD의 패치를 위해 만들어낸 것으로 파이어폭스와 애플의 OS X등 많은 업그레이드 시스템으로 이용되고 있으며 작은 델타 패키지 사이즈를 가지고 효율적인 업그레이드가 가능하다. 단점으로는 메모리를 많이 소요되고 델타 패키지 생성 속도가 느린 것이 있다. BSDIFF는 원본 바이너리 파일에 접미사를 붙이는 방식의 인덱싱 소트인 suffix sorting을 사용하여 원본 파일과 새로운 파일의 블록의 내용상으로 차이가 있는 영역과 매치되는 영역 셋을 찾은 후 다시 블록 안에서 매치되는 부분과 차이나는 영역을 찾는 방식으로 차이점을 추출한다. 이렇게 구해진 매치들은 그림 2.2와 같은 구조로 저장된다.

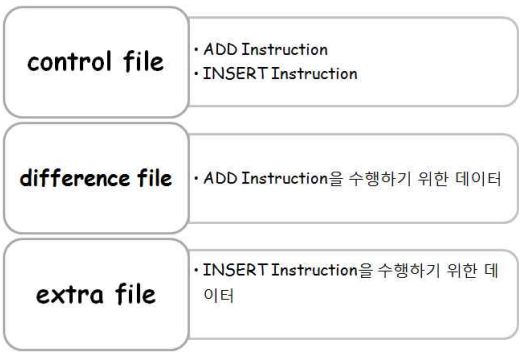



그림 2.2 BSDIFF 델타 파일 구조

control 파일은 ADD 명령어와 INSERT 명령어를 저장한다. ADD명령어는 일반적으로 델타 패키지의 COPY와 비슷한 역할을 하는 명령어이다. difference 파일은 ADD 명령어를 수행하기 위한 데이터이며 extra 파일은 INSERT 명령어를 수행하기 위한 데이터이다.

(2) VCDIFF

VCDIFF는 IETF Standard (RFC3284)에 등록된 델타 압축 방법으로 많은 바이너리 패치 도구들이 이것을 기본으로 하여 만들어졌다. VCDIFF는 압축률을 향상하기 위해 명령 코드 테이블을 사용하여 델타 명령을 1바이트로 인코딩 한다(이민재, 2007). 이러한 VCDIFF는 델타 추출을 위해 윈도우 알고리즘을 이용하고 그 추출된 결과를 명령 코드 테이블을 사용하여 인코딩하며 그림 2.3과 같이 명령어 스크립트 같은 형태로 저장된다.

| | | |
|---|---|---|
| ADD | 5 | abcde |
| COPY | 1024 | 1048 |
| RUN | 5 | Z |
| COPY | 3450 | 3800 |
| RUN | 9 | A |

그림 2.3 VCDIFF 델타 파일 구조

ADD 명령은 문자열을 추가할 때 사용되는 명령이고 사이즈와 삽입할 문자열을 나타낸다. COPY 명령은 원본 위치에서 복사할 위치를 나타내고

RUN명령은 같은 문자가 여러 번 반복되어 추가될 경우 반복되는 문자와 반복 회수를 사용한다.

### 2.1.2 패키지 관리 시스템 (Package management system)

패키지 관리 시스템은 소프트웨어에 대한 설치, 업그레이드, 설정, 삭제를 자동적으로 관리하는 툴이다. 전통적으로 그림 2.4 같은 방법으로 패키지를 설치한다(http://en.wikipedia.org/wiki/Package_management_system).

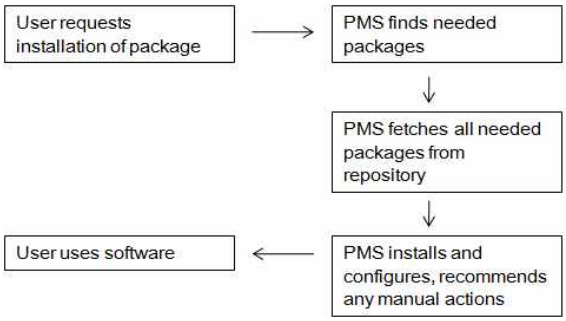



**그림 2.4 패키지 관리 시스템 수행 과정**

이러한 패키지 관리 시스템의 대표적인 예로는 레드햇 기반의 RPM과 데비안 계열의 DPKG 그리고 윈도우의 CYGWIN이 대표적인 패키지 관리 시스템으로 볼 수 있다.

### 2.1.3 PMS (Patch Management System)

PMS는 소프트웨어에서 발견되는 오류와 보안 취약점을 보완하기 위해 소프트웨어 벤더들이 제공하는 패치를 불특정 다수의 수많은 다른 환경을 가진 대상 컴퓨터에 반영시키는 일종의 자동화 패치 시스템을 말한다. 일반적으로 PMS라고 하면 보안관련 패치를 이야기 하나 PMS에는 운영체제의 보안뿐만 아니라 응용프로그램의 프로그램 버그 패치와 일반적인 기

능 향상 업그레이드도 지원한다. PMS가 만들어지면서 2개의 흐름으로 나뉘어졌다. PMS는 에이전트 기반의 PMS와 비 에이전트 기반의 PMS으로 나누어진다.

에이전트 기반의 PMS는 대상 클라이언트마다 에이전트를 설치해서 패치를 하는 방법이다. 이러한 방법은 클라이언트의 리소스를 사용하고 에이전트를 설치해야 하는 번거로움이 있지만 소속 클라이언트에 대한 자세한 정보를 PMS 중앙에 제공하여 사용자 모르게 자동화된 패치를 실행 할 수 있다는 장점이 있으며, 비 에이전트 기반의 PMS는 소속 클라이언트에 대한 자세한 정보를 제공할 수는 없지만, 에이전트를 설치하지 않아도 되는 장점이 있다. 최초 시장에서의 다수 전문가의 의견과는 달리 현재 주류는 에이전트 기반으로 가고 있다. 에이전트를 설치해야 하는 번거로움이 있지만, 관리자들은 언제 어느 상황에서라도 자세한 정보를 요구하고 클라이언트 환경에 따르는 자동화된 패치를 실행할 수 있다.

## 2.2 임베디드 환경에서의 소프트웨어 업그레이드

임베디드 환경에서의 소프트웨어 업그레이드는 크게 모바일 환경 그리고 센서네트워크 환경과 Modems, Routers, Gateways, Set-top Box, VoIP-phones으로 구성되는 CPE(Customer Premises Equipment) 환경으로 나누어진다. 모바일 환경에서는 주로 무선 환경에서의 모바일 소프트웨어 업그레이드의 기술 중 하나인 OMA-DM에서 표준으로 정의한 FUMO 기반의 FOTA를 사용하고 있고 최근에는 파일시스템의 소프트웨어 컴포넌트 업그레이드를 위한 OMA의 SCOMO 기반의 소프트웨어 컴포넌트 업그레이드를 사용한다. 센서네트워크에서는 특별한 표준 없이 센서네트워크 운영체제가 지원하는 업그레이드 기능을 사용하여 업그레이드를 진행하고 있으며 CPE 환경에서는 TR-069 on CPE WAN Management Protocol를 사용하여 원격진단과 소프트웨어 업그레이드를 진행한다. 현재까지 이러한 임베디드 환경에서의 소프트웨어 업그레이드는 동작중인 시스템을 중지한 후 시스템 재부팅 또는 프로그램 재시작을 하는 정적 업그

레이드만 지원하는 수준에 머물러 있다.

### 2.2.1 모바일 환경

#### 가. 모바일 소프트웨어 구조

현재 모바일 소프트웨어는 단순 모놀리틱 환경에서 동작하는 RTOS 기반의 모바일 소프트웨어와 RTOS 위에 브루, Java, Flash Lite등 AEE(Application Execution Environments)에서 동작하는 소프트웨어 마지막으로 스마트폰(Smart Phone)에 동작하는 Advanced OS 기반의 모바일 소프트웨어로 크게 3가지로 나눈다.

#### (1) RTOS

RTOS 기반의 모바일 소프트웨어는 휴대폰의 초기 모델인 일반폰(Vanila phone)에 들어가는 소프트웨어 특징을 가지고 있다. 초기 휴대폰은 단순히 radio signal processing과 delivering the text-based user interface의 특징만 가지고 있었으며 매우 낮은 시스템 성능에 따라 실시간성을 확보하기 위해 RTOS 기반의 소프트웨어 구조를 띄고 있었다. 최근에는 하드웨어 성능이 많이 향상되어서 보다 많은 기능을 넣기 시작하였으며 RTOS를 확장하는 구조로 진행되었다. 이러한 RTOS 기반의 모바일 소프트웨어의 특징은 모든 기능을 static link를 수행하여 소프트웨어를 배포하는 특징이 있다. 이러한 것을 모놀리틱 커널이라고 불리며 RTOS 기반의 소프트웨어는 모든 프로그램의 코드와 데이터를 하나의 이미지로 만든 후 배포하는 방식으로 되어 있다. 하나의 모놀리틱 이미지로 되어 있으므로 소프트웨어 업그레이드를 하기 위해서는 모놀리틱 이미지만 업그레이드를 수행 하면 된다. 하지만 일단 컴파일과 링크가 끝난 후 플래쉬 메모리에 저장되면 기능을 추가하기 힘든 구조로 되어 있다. 최근에는 RTOS 위에 브루 또는 위피와 같은 AEE환경을 추가하여 차후에 소프트웨어를 추가할 수 있는 구조로 발전하고 있다.

### (2) AEE (Application Execution Environment)

전통적으로 모바일 소프트웨어는 모놀리틱 RTOS 기반으로 구성되어 왔으나 기능이 복잡해짐과 휴대폰을 양산 후 게임등 추가적인 기능을 배포하기 위하여 RTOS에 AEE을 추가하여 소프트웨어를 제작하기 시작하였으며 이러한 구조로 되어 있는 휴대폰을 기능폰(Feature Phone)이라고 한다. 대표적인 AEE 환경으로서는 JAVA 머신과 Flash Lite가 있으며 AEE 환경에 속하나 퀄컴 플랫폼 의존적인 브루가 있다. 국내에서는 위피 플랫폼과 각 통신사마다 제작한 AEE환경을 사용하고 있다.

이러한 소프트웨어 특징은 휴대폰에 필수적인 기능은 모놀리틱 RTOS에서 수행하고 기타 추가적인 프로그램은 AEE환경에서 동작하는 응용프로그램이 수행한다. 그러므로 응용프로그램을 휴대폰 양산 후에도 추가할 수 있는 구조로 이루어져 있다.

JAVA 머신과 Flash Lite와는 달리 퀄컴의 브루는 REX운영체제와 AMSS 스택에서만 사용가능 하므로 AEE환경이 아닌 독립적인 모듈러 운영체제의 관점으로 보기도 한다(liz laffan외 1명, 2007).

대부분 AEE환경에서 동작하는 응용 소프트웨어는 해당 휴대폰에 동작하는 파일시스템에 존재하며 파일 시스템을 하나의 펌웨어로 생각하고 업그레이드 하는 방법이 있고 최근에는 이러한 파일시스템에 존재하는 각각의 소프트웨어 컴포넌트를 업그레이드하기 위해서는 기존 모놀리틱 RTOS에서 사용하는 업그레이드 기술과 OMA의 SCOMO을 사용해서 업그레이드를 수행 한다.

### (3) Advanced OS

모놀리틱 RTOS기반과 AEE환경과는 다르게 최근에는 하드웨어 성능향상과 모바일 소프트웨의 복잡성에 따라 Microsoft Mobile, S60/UIQ(based on Symbian)과 Linux기반의 운영체제를 사용하고 있으며 이러한 소프트웨어 구조를 가지는 휴대폰을 스마트폰(Smartphone)이라고 한다. 이러한 Advanced OS의 특징은 운영체제의 기본적인 기능을 가지고 있는 모놀리틱 커널과 파일 시스템에 미들웨어와 응용프로그램을 추가하는 구조로 되

어 있다.

## 나. MDM/MSM

모바일 관리는 현재까지 MDM(Mobile Device Management) 이라고 한다. 전형적으로 MDM이라고 하면 FOTA 서비스, 원격 진단(diagnostics) 서비스, 보안 관리를 이야기 하지만 최근에는 automatic device detection and configureation, backup and restore, lock and wipe and customer-self care와 같은 소프트웨어도 추가 되었으며 앞으로는 단순히 모바일 디바이스만 관리하는 MDM 서비스로부터 소프트웨어의 복잡성이 높아짐에 따라 MSM(Mobile Software Management)로 연구개발이 진행되고 있다. 이러한 MSM의 구성 요소로는 그림 2.5와 같이 기존 MDM의 요소를 포함하며 추가로 Architecture design & process design, Software config management, Software Versioning, API managemnet and security, Software development & integration, Variant Creation and Managemnet의 소프트웨어가 추가된다.

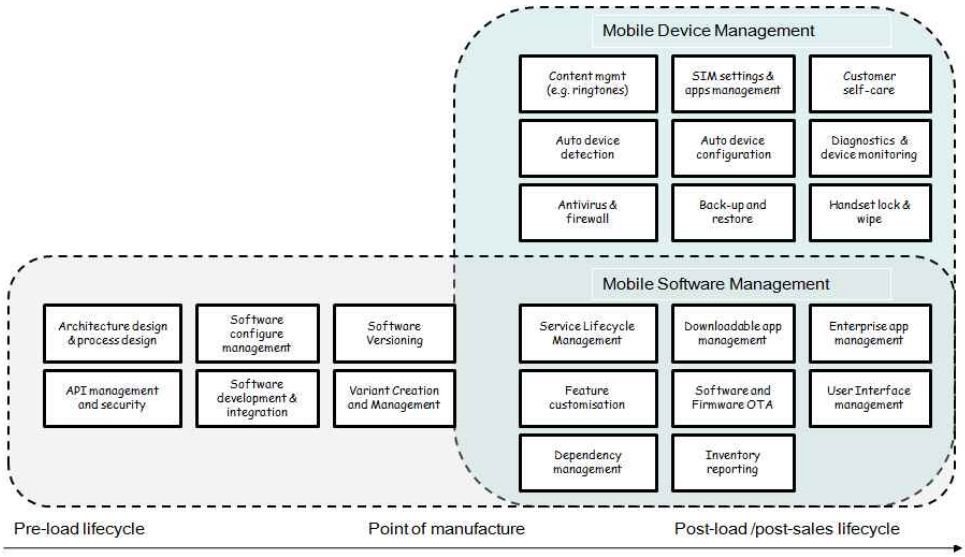



그림 2.5 MSM/MDM 구성 요소

이러한 모바일 소프트웨어 관리를 위한 기술은 크게 단말기 측면의 기술(Device technologies)과 모바일 소프트웨어를 관리하고 단말기로 전송하는 전송 기술(Delivery technologies)로 분류한다.

우선 단말기 측면의 기술은 단말기 의존적인 모바일 진단 서비스 및 펌웨어 또는 응용 소프트웨어 업그레이드를 위한 패치파일을 생성, 적용하는 기술을 말하며 전송 기술은 모바일 소프트웨어를 관리하고 소프트웨어를 전달하는 기술을 말한다. 전자는 단말기 제조사의 입장이고 후자는 통신사의 입장으로 보면 된다.

### 라. OMA-DM

#### (1) OMA (Open Mobile Alliance)

OMA는 2006년 6월에 통신사, 단말기 제조사, 네트워크 기기 관련 업체, IT 기업, 서비스 업체들이 참여하여 설립한 모바일 서비스 표준화 단체이다. WAP, Wireless Village, MGIF, SyncML, MWIF, MMS IOP, LiF등의 7개 산업계 포럼이 합쳐지면서 시작되었다. 이들은 모바일 네트워크 상에서의 서비스 및 서비스를 제공하기 위한 상호연동성(Interoperability)를 지향을 한다. 이러한 OMA의 Working Groups은 표 2.1과 같은 활동을 한다.

**표 2.1 OMA Working Groups and Committees**

| | |
|---|---|
| Architecture | 각 서비스들의 Architecture 제정 및 조율 |
| Browser & Content | 사용자 에이전트, 콘텐츠 형식, 프로그래밍 인터페이스에 대한 규격과 디지털 저작권 및 방송 서비스 규격 등을 제정 |
| Developers Interest Group | 각 응용 서비스의 개발 지원 및 개발 가이드 역할 수행 |
| **Device Management** | 단말의 설정 및 각종 프로그램을 원격으로 유지, 관리하는 규격 제정 |
| Digital Rights Management | DRM에 대한 정의 |
| Games Services | 모바일 게임을 위한 API, Protocol 등에 있어서의 각종 호환성 관련 부분에 대한 작업을 진행하며, 이전의 MGIF(Mobile Game Interoperability Forum)의 업무를 계속적으로 담당 |

| | |
|---|---|
| Interoperability | 각 WG에서 제정한 Candidate Enabler의 신뢰성 및 상호 운용에 대한 시험 표준 제정 및 시험 수행 |
| Location | End-to-End의 위치 기반 서비스 규격 제정하며, 에전의 LIF(Location Interoperability Forum)의 업무를 계속적으로 담당 |
| Messaging | MMS, Instant Messaging, Mobile E-mail 서비스 등 메시지와 관련된 서비스의 규격 제정 |
| Mobile Commerce & Charging | 과금에 관한 표준안을 제정하고, 과금 데이터의 처리, 관련 정보의 흐름에 대한 표준안 관련 작업을 담당함 |
| Mobile Client Environment (MCE) | Content format, Rendering, Browsing등의 클라인어트 환경에 대한 표준을 정의 |
| Presence & Availability | 상태 정보 제공 및 관리 등에 해당하는 상호 호환적인 사용과 관련한 규정을 제정하는 역할을 담당 |
| Push to Talk Over Cellular | 이동통신 네트워크에서 PTT 규격 제정 |
| Release and Planning Management | 릴리즈와 계획에 대한 관리의 표준을 정의 |
| Requirements | 각 서비스들의 요구사항을 정의하고 규정하는 역할을 담당 |
| Security | 응용프로그램 수준의 보안 규격 제정 및 보안 컨설팅 |

### (2) DM

Device를 Management하기 위한 규격은 표 2.2와 같다.

**표 2.2 OMA-DM 규격**

| | | |
|---|---|---|
| 기본 규격 | DM Protocol | DM 서버와 단말 간의 통신 프로토콜 정의 |
| | Representation | DM 메시지의 구성 요소 및 스키마 정의 |
| | Tree and Description | 단말 관리 데이터의 내부 표현 방식 정의 |
| | Standard Object | 필수 단말 관리 객체 정의 |
| | Security | 단말 관리 메시지의 보안 방법 정의 |
| | Bootstrap | 신규 단말을 관리 가능한 상태로 설정하기 위한 방법 정의 |
| | Notification | 단말 관리 세션의 시작을 알리는 메시지 전송 방법 정의 |
| 부가 규격 | FUMO | 펌웨어 업데이트를 위한 관리 객체 및 인터페이스 정의 |
| | SCOMO | 소프트웨어 컴포넌트 관리를 위한 관리 객체 및 인터페이스 정의 |
| | Diagnostic & Monitoring | 단말 장애 정보의 수집, 전송, 진단, 원격 해결을 위한 메커니즘 정의 |
| | ConnMo | 네트워크 접속성 설정을 위한 관리 객체 및 메커니즘 정의 |
| | Scheduling | 스케쥴 기반의 단말 관리를 위한 관리 객체 및 메커니즘 |

| | | |
|---|---|---|
| | | 정의 |
| | Web Service Interface | DM 서버와 외부 시스템 사이의 인터페이스 정의 |
| | Smartcard | 단말기에 장착된 스마트카드 기반의 단말 관리를 위한 관리 객체 및 메커니즘 정의 |

### 라. FOTA (Firmware Over The Air)

#### (1) FOTA

일반적으로 FUMO와 FOTA를 같은 것으로 볼 수도 있으나 FUMO는 펌웨어 업그레이드를 위하여 업데이트 패키지를 DM 서버로부터 단말기로 전송하기 위한 표준을 정의한 것이고 FOTA는 업데이트 패키지 생성부터 단말기로 적용까지 모든 과정을 포함한다. FOTA는 FUMO의 표준 없이 업데이트를 수행하는 경우도 많다. 그림 2.6은 일반적으로 사용되는 모바일 소프트웨어의 구조도이다. 어두운 색으로 표현된 블록은 펌웨어 업데이트를 통해 업데이트되는 부분을 나타낸다.

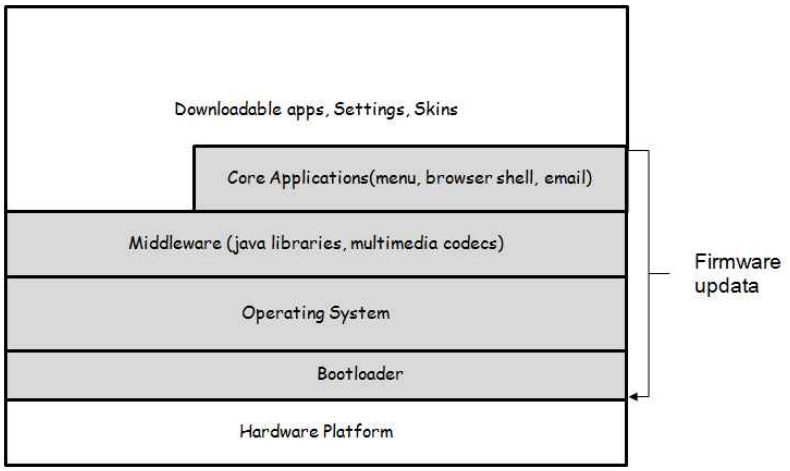



**그림 2.6 모바일 소프트웨어 구조도**

#### (2) FOTA 업데이트 과정

1. 핸드폰 제조사로부터 펌웨어 업데이트 패키지(델타 패키지)를 생성한다.
2. DM 서버로 생성된 업데이트 패키지를 전송한다.
3. 단말기는 새로운 업데이트 패키지를 다운 받는다.

4. 단말기의 업데이트 에이전트로부터 다운 받은 업데이트 패키지를 설치한다.

### 2.2.2 센서네트워크 환경

대부분 모바일 환경과 거의 비슷한 방법으로 업그레이드를 수행한다. 하지만 일반적으로 센서네트워크에서의 업그레이드는 특별한 표준이 없이 진행된다. 센서네트워크 업그레이드의 고려할 점은 모바일 환경보다 리소스가 매우 부족하다. 그러므로 델타 패키지를 전송할 때 전체 패키지를 전송하는 것이 아니라 델타 패키지의 명령어들을 순서대로 전송하여 업그레이드를 수행한다.

### 2.2.3 CPE 환경

현재 CPE 환경에서 소프트웨어 업그레이드는 TR-069를 기반으로 수행하며 각 장치마다 정의된 표준은 그림 2.7과 같이 정의 되어 있다.

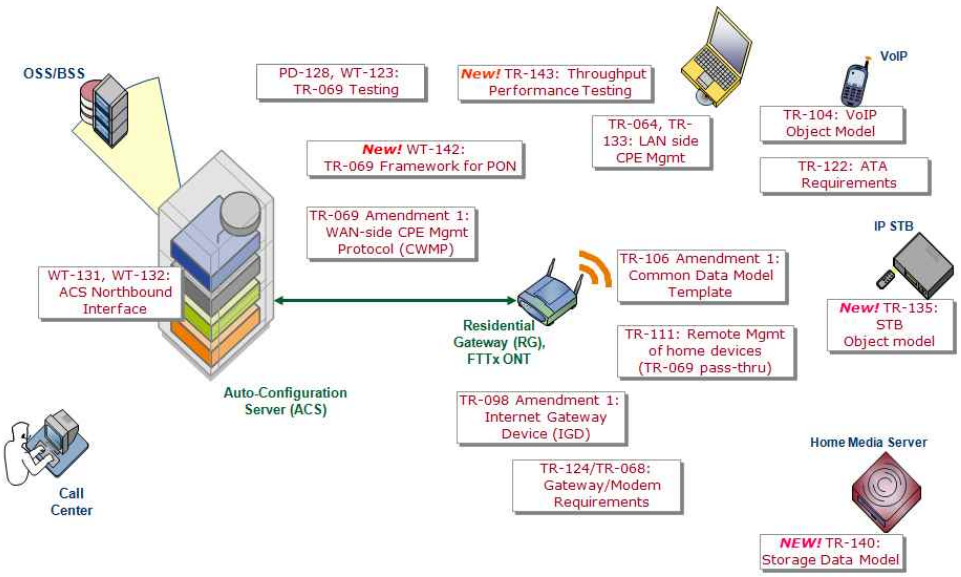



그림 2.7 CPE 표준

TR-098은 인터넷 게이트웨이를 위한 표준 모델이고 TR-104는 VoIP 폰에 대한 표준 모델이다. 또한 TR-135는 셋탑 박스를 위한 표준이다.

TR-069 on CPE WAN Management Protocol은 Broadband Forum(DSL Forum)에서 정의한 CPE WAN Management Protocol (CWMP)이다. TR-069의 기본적인 기능은 표 2.3과 같다.

표 2.3 TR-069의 기본적인 기능

| | |
|---|---|
| Auto-configuration and dynamic service providing | Initial CPE Configuration<br>Re-provisioning at any subsequent time |
| Software/firmware image management | Version identification<br>File download initiation<br>Notification of the success or failure of a file download |
| Status and performance monitoring | Log file, and dynamic notification |

| | |
|---|---|
| Diagnostics | Connectivity and service issues |

이러한 TR-069 프로토콜의 구조는 그림 2.8과 같다.

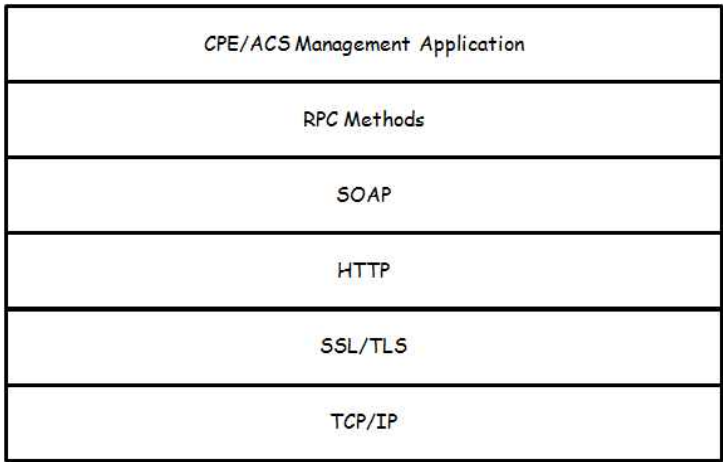



**그림 2.8 TR-069 프로토콜 구조**

일반적인 TCP/IP 프로토콜위에 SSL(Secure Socket Layer) 또는 TLS(Transport Layer Security)가 있고 그 위에 HTTP 프로토콜에 XML 문법에 기반을 둔 SOAP 프로토콜을 사용하여 원격 함수를 호출하는 구조로 이루어진다. 이러한 TR-069를 사용해서 ACS(Auto Configuration Server)와 CPE 단말기간의 상호 통신은 그림 2.9와 같다.

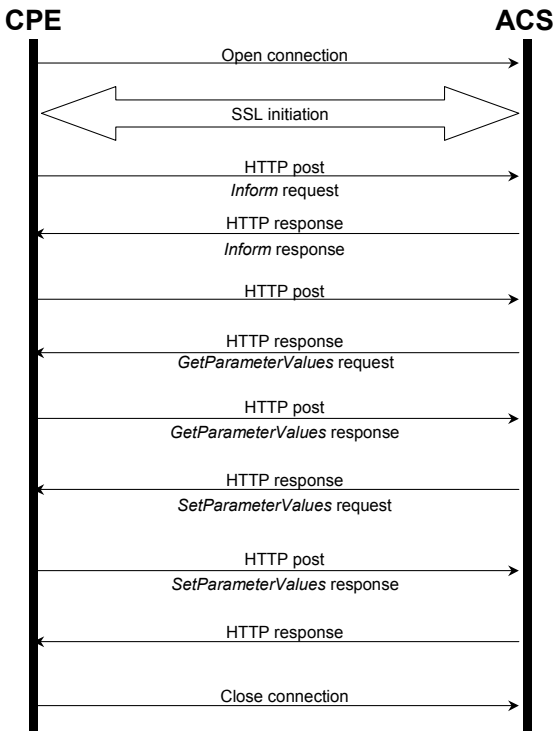



그림 2.9 CPE 단말기간의 상호 통신

### 2.2.4 임베디드 환경에서의 정적 업그레이드

대부분의 임베디드 환경에서의 소프트웨어 업그레이드는 무선 환경에서 진행되며 또한 시스템 자원이 열악한 환경에서 소프트웨어 업그레이드를 지원하므로 업그레이드 패키지는 항상 이전 버전의 소프트웨어와 새로운 버전의 소프트웨어의 차이를 가지고 업그레이드를 진행하는 델타 패키지를 활용한다. 이러한 임베디드 환경에서의 델타 패키지는 일반 PC에서 사용되는 델타 패키지와 다른 구조로 작성되어야 한다.

#### 가. 임베디드 환경에서 델타 적용의 문제점

일반적인 델타 이미지를 생성 방법을 임베디드 시스템에 적용하고자 하면 다음과 같은 문제점을 가지고 있다. 우선 메모리 자원이 열악하므로 일반적인 델타 방법을 사용하면 많은 메모리 공간이 필요하게 된다. 또한 리

눅스 커널 이미지와 같은 압축된 커널 이미지에 대해서도 델타 업그레이드가 가능해야 한다.

### 나. In-Place 델타 이미지

델타 이미지를 적용하기 위해서는 2개의 그림 2.10과 같이 저장 공간이 필요하다. 만약 1개의 이미지를 가지고 In-place copy를 수행 하면 그림 2.10과 같이 충돌이 일어난다.

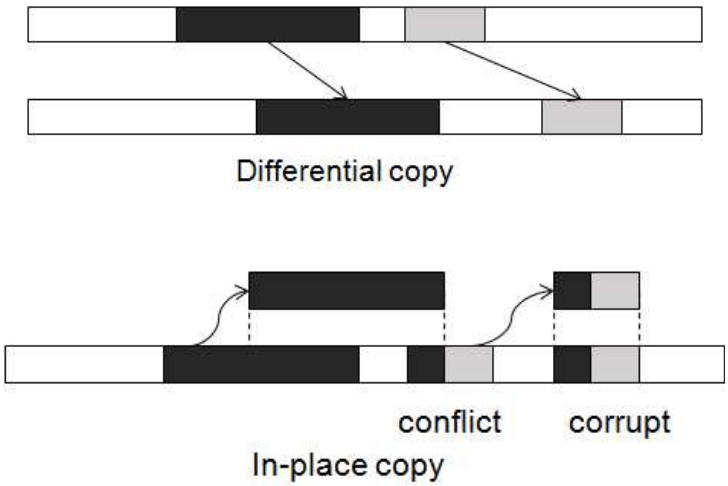



그림 2.10 Differential copy와 In-place copy

이러한 Read-Write 충돌 문제를 가장 쉬운 방법으로 해결하려면 그림 2.11과 같이 충돌이 일어나는 부분을 INSERT 명령어로 삽입하여 해결하면 된다. 하지만 이러한 방법으로 해결하면 델타 사이즈가 커지는 단점을 가진다.

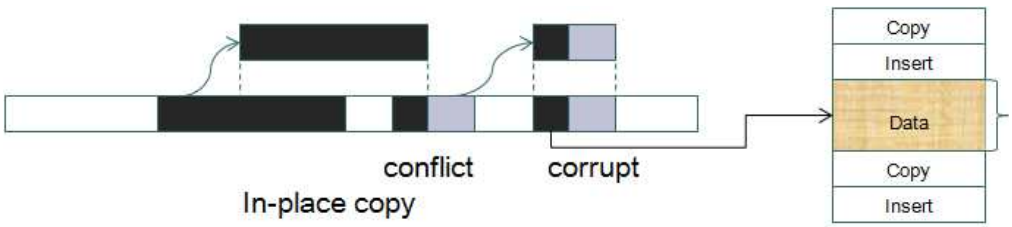



그림 2.11 Read-Write 충돌 해결 방법

델타 사이즈를 줄이기 위한 방법에 대한 연구들이 진행되고 있는데 그중 한가지 방법이 COPY 명령어를 대상으로 left-to-right 방법 또는 right-to-left를 사용하는 방법이 있다(Randal C.Burns 외 1명, 1998).


그림 2.12 right-to-left 방법

그림 2.12는 이러한 문제를 해결하기 위해 right-to-left 방법을 사용하여 해결한 그림이다. 이러한 원리로 Read-Write 충돌 문제를 해결한 방법은 In-place reconstruction 알고리즘(Randal C.Burns 외 1명, 1998)이 있다. 표 2.4는 In-place reconstruction 알고리즘에 대한 의사코드이다.

표 2.4 In-place reconstruction 알고리즘

1. Delta파일 -> copy, add command 구분
2. write offset이 증가하는 방향으로 정렬
3. Copy command를 가지고 digraph(방향성이 존재하는 graph)를 생성
4. Digraph를 가지고 topological sort를 수행
   4.1 수행도중 cycle break 를 찾는다. (그래프 내에 사이클이 존재한다면 순서를 식별할 수 없다. )
   4.2 cycle break를 찾아 내면 locally minimum or constant time cycle breaking policy를 사용하여 삭제
   4.3 삭제된 encoded된 데이터는 Add command로 삽입
5. 정렬된 digraph를 가지고 delta file을 다시 작성

### 다. Compressed 이미지

델타 파일을 생성하는데 압축된 이미지일 경우에는 대부분의 압축이 수행되면 원본이미지와는 전혀 다른 파일이 생성되므로 문자열 비교를 통하여 델타를 생성하는 방법으로는 델타 파일이 생성되지 않는 문제를 가지고 있다. 압축된 이미지의 델타 처리의 정확한 해결책은 찾지 못하였으나

압축을 해제를 한 후 델타 파일을 생성한 다음 델타 적용 때 역시 압축 해제해가면서 델타를 적용하는 방법이 있다.

## 2.3 동적 소프트웨어 업그레이드

### 2.3.1 동적 소프트웨어 업그레이드

앞 절에서 설명한 소프트웨어는 현재 까지 진행되어온 연구와 실제 상용화된 소프트웨어 업그레이드를 설명하였고 본 절은 현재 임베디드 환경에는 적용 되지 않은 동적 소프트웨어 업그레이드를 설명한다. 현재까지의 동적 소프트웨어 업그레이드에 대한 연구는 동적 업그레이드 수행 방법에 대하여 초점이 맞추어 연구가 진행되고 있다. 현재까지 연구되어온 동적 업그레이드 수행 방법에는 커널 모듈를 이용한 동적 업그레이드와 운영체제의 수정을 통한 동적 업그레이드 그리고 컴파일러 수정에 따른 동적 업그레이드 마지막으로 바이너리 패치 기술을 활용한 동적 업그레이드가 있다.

### 2.3.2 기존 동적 업그레이드

#### 가. 커널 모듈를 이용한 동적 업그레이드

모듈 방법의 동적 업그레이드는 리눅스와 같은 운영체제는 LKM(Loadable Kernel Modules)의 기능을 지원하며 이러한 기능을 활용하여 예전에 동작하는 모듈을 내리고 새롭게 수정된 모듈을 올리는 방법을 사용하여 커널을 동적으로 업그레이드를 할 수가 있다. 하지만 이러한 방법은 운영체제의 LKM으로 작성된 코드만 업그레이드를 할 수 있는 매우 제한적인 형태를 가진다.

#### 나. 운영체제의 수정을 통한 동적 업그레이드

운영체제의 수정을 통한 동적 업그레이드 시스템의 대표적인 연구는

K42 운영체제(Andrew Baumann, 2007)가 있다. K42 운영체제는 객체지향 운영체제로 개발 되었으며 운영체제를 동적 업그레이드 가능하도록 처음부터 고려하여 설계되었다. K42 운영체제의 동적 업그레이드 기법은 Interposition과 Hot swapping 기법을 사용하여 동적으로 오브젝트를 교체하는 방법으로 동적 업그레이드를 구현하였다. 최근에는 K42 기반의 동적 업그레이드 기법을 리눅스나 FreeBSD 운영체제로 적용하기 위한 연구가 진행되고 있다.

이러한 방법은 새로운 운영체제를 제안하거나 기존 사용되고 있는 운영체제의 많은 부분을 수정해야 함으로 실제 적용하기는 힘든 문제점을 가지고 있다. 또한 K42 기반의 동적 업그레이드 기법은 업그레이드 하고 싶은 클래스를 새롭게 컴파일한 후 동적 모듈로 삽입하는 방법으로 구현되는데 현재 동작중인 전반적인 운영체제 버전 관리를 하지 못하고 수정된 클래스에 대해서만 버전관리가 되므로 재부팅 후 새로운 버전으로 실행되기 위해 2차 저장소에 있는 실행 이미지의 버전관리와 독립적으로 이루어진다. 이와 같은 이유로 2차 저장소의 이미지 업데이트가 동시에 이루어졌을 경우 상당히 복잡하게 버전을 관리해야 하는 문제점을 가지고 있다.

### 다. 컴파일러 수정을 통한 동적 업그레이드

컴파일러 수정을 통한 동적 업그레이드 시스템의 대표적인 연구는 Ginseng(http://www.cs.umd.edu/projects/PL/dsu/)이 있다. 이러한 연구는 C언어와 같은 특정 언어에 의존적인 업그레이드 시스템을 지원한다. 업그레이드 시점이 오면 예전에 분석한 정보와 이전 버전의 코드 그리고 새로운 코드를 활용하여 패치코드를 생성하고 패치코드를 컴파일러가 다시 업그레이드 가능한 코드 생성한 후 패치 하는 방식으로 구성되어 있다. 이러한 방법으로 동적으로 업그레이드가 가능하도록 하기 위해서는 두 가지를 고려해야 한다. 첫째로 함수를 새로운 버전의 함수로 대체하는 기법이 필요하고 두 번째로 정의된 데이터 스트럭쳐를 변경하기 위한 기법이 필요하다. Ginseng은이 두 가지를 function indirection과 type wrapping을 통해 해결하였다. function indirection는 전역변수인 f는 컴파일 단계에서 업

그레이드가 가능하도록 *f_ptr로 변환하여 실제 함수를 호출할 때 indirection 함수를 호출하는 방식으로 구현된다. type wrapping은 역시 컴파일 단계에서 처음 데이터 스트럭쳐를 선언할 때 업그레이드 가능한 데이터 스트럭쳐를 생성을 한 후 업그레이드 할 때 포인터 조작으로 새로운 데이터 스트럭쳐의 주소로 가리키는 방식을 사용한다. 이러한 동적인 소프트웨어 업그레이드는 응용프로그램에서의 업그레이드를 고려한 방법이다 최근에는 리눅스 커널을 대상으로 연구를 하고 있으나 Ginseng 을 리눅스로 적용하려면 두 가지 문제점이 있다(Andrew Baumann, 2007). 첫째로 Ginseng은 싱글 쓰레드 프로그램만 지원한다. 두 번째로는 Ginseng은 컴파일 당시 모든 함수에 대해서 자동적으로 indirection 함수를 생성한다. 이러한 방법은 운영체제 커널이 사용하기에는 성능에 상당한 문제를 야기시킨다.

### 라. 바이너리 패치 기술을 활용한 동적 업그레이드

바이너리 패치 기술은 기존 커널의 수정 없이 새로운 기능을 추가함으로써 업그레이드 가능한 방법으로 본 논문에 사용한 방법과 매우 유사하다. 이러한 방법은 대부분 프로그램 실행 흐름을 중간에 바꾸어 업그레이드 코드로 분기하는 방법으로 구현된다.

#### (1) KernelInst

바이너리 코드 단위로 동적 패치하는 대표적은 연구로는 KernelInst(Ariel Tamches외 1명, 2001)가 있다. 바이너리 패치 기술에 대한 연구는 성능 측정, 소프트웨어 테스팅의 코드 커버리지, 디버깅의 브레이크 포인트, 코드 최적화, 코드 패치 등으로 사용된다. KernelInst는 코드중간에 분기 명령어를 삽입하여 패치된 코드로 분기하고 수행을 한 후 다시 원래 코드로 복귀한 후 반환하는 방식으로 구현되었다. 특히 Kerninst는 해당 고정된 명령의 구조(SPARC, PowerPC)에서 구현되어있다. SPARC와 같은 CPU에는 브렌치 명령어의 범위가 PC±8MB이기 때문에 Kerninst는 스프링보드(springboard)라는 기술을 사용하여 패치된 커널코드로 점프

를 하였다. Kerninst는 인텔의 x86 CPU와 같은 명령어가 고정되지 않는 CISC 구조에서 원래의 코드에 점프 코드를 삽입 시 문제가 발생한다. 이러한 연구는 함수단위 업그레이드가 아닐 어셈블리 언어로 작성된 패치 파일을 가지고 업그레이드를 수행하기 때문에 실제로 활용하기는 무리가 있다(박현찬외 2명, 2007).

### (2) OPUS

OPUS(Gautam Altekar외 3명, 2005)는 동적인 보안 패치에 관련하여 연구된 시스템이다. OPUS는 C 프로그램의 함수단위로 업그레이드가 가능하도록 구현되었으며 프로그램 실행도중 구 버전의 함수로부터 새로운 버전으로 분기하여 업그레이드를 수행하는 방법을 사용한다. 최상위 함수를 수정할 수 없으며 전역변수 수정도 할 수 없는 단점을 가지고 있다. 하지만 OPUS의 특징은 GCC 컴파일러를 수정하여 GCC 빌드 과정에서 동적으로 업그레이드가 가능한 코드인지 아닌지를 판단한다. 만약 새로운 코드가 전역변수 또는 최상위 함수를 수정하였다면 사용자에게 경고를 알리는 특징을 가지고 있다. OPUS는 본 논문에서 구현한 방법과 상당히 유사하나 2.3.3절에서 설명할 소스레벨 비교에 따르는 문제점을 가지고 있고 2차 저장소에 있는 정적 이미지를 업그레이드에 대한 고려가 없다. 또한 OPUS는 함수 단위로 코드 업그레이드는 지원하나 전역 변수와 데이터 스트럭쳐 업그레이드를 지원하지 못한다.

### (3) DynAMOS

DynAMOS(Kyung Dong Ryu외 1명, 2007)는 현재까지 동적업그레이드에 대한 연구 중 커널 동적업그레이드에 대해 가장 체계적으로 정리를 하였다. DynAMOS에서는 커널 업그레이드에 대해 특징과 요구사항을 12가지로 정리 하였으며 내용은 표 2.5와 같다.

**표 2.5 DynAMOS에서 제안한 커널 업그레이드 요구사항**

| · **Updating a variable value**<br>전역 변수에 대한 수정을 할 경우에 해당한다. 예를 들면 읽기 전용 전역변수인 open files limit |
|---|

| |
|---|
| 최대값을 새로운 최대값으로 변경하고자 하는 경우 이러한 경우에는 state tracking이 필요하다. |
| **· Updating a variable value with synchronized**<br>상호 배제가 필요한 변수를 수정할 경우에 해당한다. 예를 들면 inode의 owner(uid)를 수정할 경우에 state tracking을 포함한 inode의 세마포어를 얻어야 한다. |
| **· Adding a new variable used by a single function**<br>하나의 함수만 접근하는 전역변수를 추가할 경우에 해당한다. 하나의 함수만 접근하므로 추가적인 safe point는 필요치 않다. |
| **· Adding a new variable used by a function group**<br>여러 함수가 접근하는 전역변수를 추가할 경우에 해당한다. 여러 함수가 동시에 접근할 수 있으므로 state tracking이 필요하다. |
| **· Adding a new field in a data structure.**<br>data structure에 새로운 필드가 추가되었을 경우에 해당한다. 예를 들어 inode structure에 새로운 포인터가 추가되었을 경우에 해당된다. 데이터 스트럭쳐의 수정은 state transfer를 또는 shadow data structure를 사용하여 업데이트한다. |
| **· Updating a quiescent single function.**<br>중지하는 함수를 업데이트를 할 경우에 해당한다. |
| **· Updating a non-quiescent single function.**<br>절대 중지하지 않는 함수를 업데이트 할 경우에 해당한다. 예를 들어 쓰레드 함수는 중지하지 않는다. 이러한 함수는 리눅스의 스케줄러 수정을 통하여 업데이트한다. |
| **· Updating interrupt handlers.**<br>인터럽트 핸들러를 업데이트할 경우에 해당한다. 인터럽트 핸들러는 인터럽트를 중지한 후 quiescent function과 같은 방법으로 처리한다. |
| **· Updating a function group**<br>여러 함수를 업데이트 할 경우 업데이트할 함수에서 업데이트할 함수를 호출할 경우에 해당한다. 이러한 경우에는 서로 동기 맞추어 업데이트해야 한다. |
| **· Updating a function signature**<br>어떤 함수가 수정된 함수를 호출할 경우에 수정된 함수의 인자가 변경되었을 경우에 해당한다. |

| |
|---|
| · **Updating a quiescent subsystem.**<br>중지하는 함수 그룹에서 데이터 스트럭쳐 업그레이드를 수행 할 경우에 해당한다. |
| · **Updating a non-quiescent subsystem.**<br>중지하지 않는 함수 그룹에서 데이터 스트럭쳐 업그레이드를 수행 할 경우에 해당한다. |

DynAMOS 커널은 제한적인 데이터 스트럭쳐 업그레이드를 지원한다. 이는  해쉬 테이블을 사용한 Shadow Data Structure 기법을 사용하여 데이터 스트럭쳐 업그레이드를 수행하였다. 함수 단위로 새로운 함수로 분기하는 방법을 사용하여 OPUS와 상당히 유사한 방법으로 업그레이드를 수행한다. 새롭게 수정된 커널은 리눅스 커널 모듈을 사용하여 커널 영역으로 옮긴 다음 adaptive function cloning 기법을 사용하여 새로운 함수 위치로 복사하여 구 버전의 함수에서 신 버전의 함수로 분기하는 방식을 사용한다. DynAMOS는 신 버전의 함수로 분기하기 위한 x86의 절대 분기 코드인 6바이트의 코드를 삽입하는데 동기화 문제를 해결하기 위하여 분기 코드를 삽입하는 동안은 인터럽트를 막는 방식을 사용하였다.

하지만 DynAMOS는 데이터 스트럭쳐의 추가된 필드 부분만 업그레이드 할 수 있는 제한을 가지고 있으며 DynAMOS는 K42와 매우 유사한 방법으로 버전을 함수마다 관리하여 함수 단위 업그레이드를 지원하나 2차 저장소의 정적 이미지와 동시에 버전관리를 할 수 없다. 또한 개발자가 업그레이드를 적용까지 하기 위해서 상당히 복잡한 과정을 거치며 업그레이드 적용 방법의 문제로 자동화된 시스템을 만들 수 없다.

### (4) POLUS

POLUS(Haibo Chen외 4명, 2007)는 응용 프로그램을 대상으로 동적 업그레이드를 지원한다. 함수 단위의 동적 업그레이드를 수행하며 멀티 쓰레드 프로그램에 대한 동적 업그레이드와 데이터 스트럭쳐 업그레이드 그리고 동적 업그레이드 실패에 따르는 문제에서도 복구 할 수 있는 기능을 지원한다.

동적 업그레이드를 위해 patch constructor, patch injector, runtime library의 3가지 구성요소로 나누었으며 patch constructor 는 source-to-souce 컴파일러인 CIL(George C. Necula, 2002)를 사용하여 소스레벨 비교 방법을 사용하여 수정된 함수와 데이터 스트럭쳐을 찾아낸다. 이러한 방법으로 수정된 함수와 데이터 스트럭쳐를 C언어 소스코드로 생성하게 되고 공유 라이브러리 파일로 컴파일을 수행한 후 함수단위 코드 업그레이드는 기존 코드에서 공유라이브러리 파일로 분기 하는 방법을 사용하였고 리눅스의 디버깅 API를 사용하여 전역 변수를 write protect 설정하여 해당 코드로 접근 하는 코드를 새로운 버전의 전역변수로 복사하는 일을 하여 동적 업그레이드를 수행한다.

하지만 POLUS는 동적 업그레이드에서 필요한 함수, 전역변수, 데이터 스트럭쳐 등 모든 요소에 대해서 업그레이드를 제공하나 소스레벨 비교에 따른 인라인 함수가 수정된 사항에 대해서는 동적 업그레이드를 지원하지 못하며 데이터 스트럭쳐 업그레이드 위해서 개발자가 소스레벨 비교로 나온 결과물인 코드를 직접 수정해서 업그레이드가 가능한 코드로 만들어야 개발자와 업그레이드 관리자간에 독립적인 작업이 이루어질 못한다. 또한 응용프로그램에만 적용할 수 있는 방법을 제공하는 문제점을 가지고 있다.

### (5) Ksplice

Ksplice(http://web.mit.edu/ksplice/)는 본 논문과 매우 유사한 방법으로 동적업그레이드를 수행한다. 우선 소스레벨 비교에 따르는 문제점을 오브젝트 파일 비교를 통한 방법으로 해결하였으며 개발 프로세스와 업그레이드 프로세스 독립적으로 수행할 수 있는 장점이 있다. Ksplice는 두 커널간의 프로젝트 단위로 오브젝트 비교를 통한 결과물을 리눅스 커널모듈로 작성 후 커널 영역에 삽입한다. 구 버전 함수는 커널 영역에 삽입된 신 버전의 커널 모듈 함수로 분기하는 방법으로 함수 단위의 동적 업그레이드 수행한다. Ksplice는 데이터 스트럭쳐 업그레이드를 지원하지 못하는 한계가 있지만 2005년 5월부터 2007년 12월까지 리눅스의 50번의 보안 패치에 대해서 84%를 데이터 스트럭쳐 업그레이드 없이 함수 단위 코드 업그

레이드 방법으로 업그레이드가 가능하다고 한다.

하지만 Ksplice 역시 2차 저장소에 있는 새로운 버전의 이미지를 업그레이드를 하기위한 방법과 버전 관리에 대한 언급이 없으며 데이터 스트럭쳐과 전역변수를 수정할 수 없다.

### 2.3.3 동적 업그레이드 델타 비교 방법

현재까지 자동화된 업그레이드 시스템은 크게 두 가지 분류로 나누어진다. 소스레벨 비교를 통하여 수정된 부분을 체크한 후 업그레이드 파일을 생성하는 방법인 Ginseng, OPUS, POLUS 가 있고 컴파일된 결과물인 오브젝트 파일 비교를 통하여 업그레이드 파일을 생성하는 방법은 Ksplice 가 있다.

소스레벨 비교의 장점은 함수 단위의 업그레이드를 뿐만 아니라 데이터 스트럭쳐의 수정된 사항을 알 수가 있으며 데이터 스트럭쳐의 업그레이드를 수행 할 수 있는 장점이 있다. 하지만 소스레벨 비교 방법은 인라인 함수의 경우는 해당 코드는 컴파일 당시 기존 코드에 삽입되는 형태를 가지기 때문에 업그레이드를 수행할 수 없으며 함수의 인자의 형태가 수정되었을 경우에도 처리할 없다(Jeffrey Brian Arnold, 2008). 또한 소스레벨 비교로 나온 결과물을 다시 컴파일을 수행하여야 되는데 기존 프로젝트의 컴파일 옵션을 적용하기 위하여 복잡한 절차를 수행해야 한다.

오브젝트레벨 비교는 소스레벨 비교의 단점인 인라인 함수의 업그레이드 처리와 함수의 인자가 수정 되었을 경우에도 처리할 수 있고 오브젝트 비교를 통하여 수정된 오브젝트 파일을 다시 컴파일을 수행할 필요 없이 바로 사용할 수 있는 장점을 가지나 단점으로는 데이터 스트럭쳐가 수정되었을 경우 찾아내기가 쉽지 않으며 수정된 함수 단위 오브젝트 코드의 심볼 재배치 문제를 해결해야 하는 문제를 가지고 있다.

### 2.3.4 임베디드 환경에서의 동적 소프트웨어 업그레이드

임베디드 시스템 안에 내장되어 있는 운영체제 또는 그 위에 동작하는 응용프로그램을 동적으로 업그레이드하기 위해서는, 업그레이드와 관련된 모든 시스템, 개발 도구에 대하여 다음과 같은 요구 사항들을 만족하여야 한다.

첫째, 플랫폼 독립적인 동적 업그레이드 프레임워크를 제공해야 하며 운영체제 전체에 대한 수정 없이 적용할 수 있는 동적 업그레이드 시스템이 되어야 한다. 또한 업그레이드 과정에서의 보안 문제 해결을 위한 다양한 기술, FOTA, OMA-DM등 주변 기술과 연계될 수 있는 확장성을 가져야 한다.

둘째, 사용자가 쉽게 접근할 수 있도록 동적 업그레이드 서비스를 제공해야 한다. 즉 동적 업그레이드가 가능한 소프트웨어를 구현하는 경우에도 개발 프로세스 상의 변화가 거의 없도록 하여야 한다.

셋째, 동적 업그레이드를 수행하기 전에 현재 업그레이드가 가능한 상태인지 체크를 할 수 있어야 하고 만약 잘못된 업그레이드를 통해 시스템에 문제가 발생할 가능성이 있을 경우 업그레이드 수행하기 전의 시스템 상태로 복원할 수 있어야 하는 유연성을 가져야 한다.

넷째, 어떠한 업그레이드를 수행하더라도 업그레이드된 코드로 인한 시스템 오버헤드를 줄여야 한다.

다섯째, 동적 업그레이드를 위한 고려한 개발을 위해, 또, 소프트웨어 형상과 동적 업그레이드를 위한 이미지의 관계를 효과적으로 관리하기 위하여, 기존의 통합 개발 환경에 동적 소프트웨어 업그레이드 프레임워크가 잘 융합되어야 한다.

# 제 3 장 동적 업그레이드 프레임워크

## 3.1 동적 업그레이드 시스템 프레임워크

임베디드 시스템을 위한 동적 업그레이드 프레임워크에 대한 연구는 일반 PC의 경우와는 달리 플랫폼에 종속적인 부분이 상당히 많다. 이러한 플랫폼 종속적인 부분들은 차후 동적 업그레이드 기술을 다른 플랫폼으로 이식할 때 상당히 많은 시간과 노력이 필요하다. 본 연구에서는 이러한 문제점을 해결하고자 임베디드 시스템을 위한 동적 업그레이드에 대한 프레임워크를 설계하였다. 그림 3.1은 본 논문에서 설계한 동적 업그레이드 프레임워크이다.

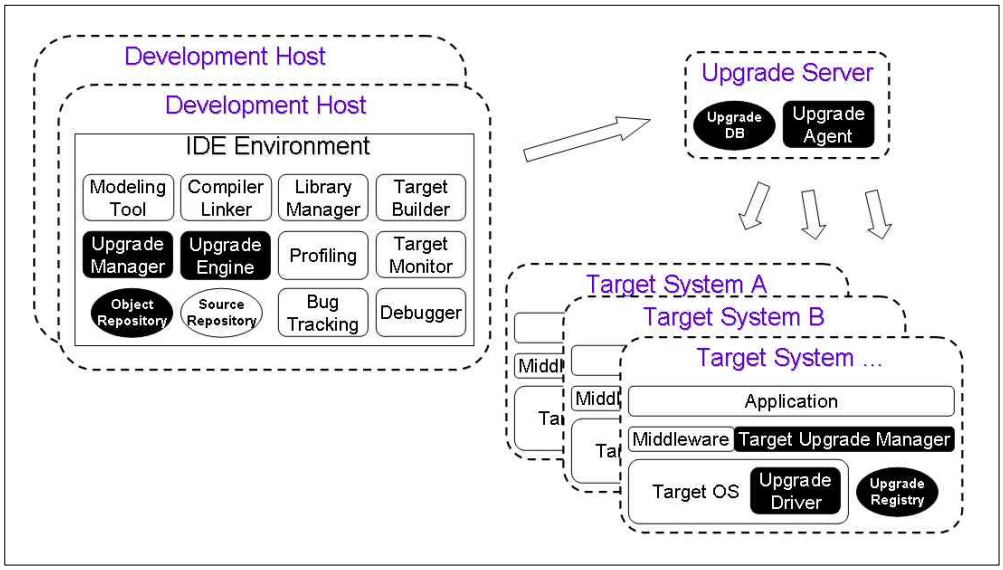



그림 3.1 동적 업그레이드 프레임워크

### 가. 개발 호스트 (Development Host)

동적 업그레이드 시스템을 구성하기 위한 개발 호스트는 기존 통합 개발환경(IDE, Integrated Development Environment)에 추가로 업그레이드 관

리자, 버전 관리자, 오브젝트 저장소를 가지고 있다. 개발 호스트는 빠른 업그레이드 수행을 위해 최소화된 업그레이드 이미지를 생성하는 일을 하며 생성된 업그레이드 이미지를 업그레이드 서버에 전송하는 일을 한다.

**나. 업그레이드 서버 (Upgrade Server)**

업그레이드 서버는 업그레이드 데이터베이스와 업그레이드 에이전트를 가지고 있다. 개발 호스트에서 받은 업그레이드 이미지를 버전별로 관리하고 저장하는 역할을 하며 타겟 시스템과 통신을 하여 업그레이드 이미지를 암호화한 후 전송하는 일을 한다.

**다. 타겟 시스템 (Target System)**

타겟 시스템은 타겟 업그레이드 관리자와 업그레이드 드라이버 그리고 업그레이드 저장소를 가지고 있다. 실제 업그레이드를 수행하는 역할을 하며 업그레이드 서버로부터 받은 업그레이드 이미지를 복호화하여 동적으로 업그레이드를 수행하는 일을 한다.

## 3.2 업그레이드 생성기 (Upgrade Generator)

### 3.2.1 업그레이드 생성기 모듈 구성도

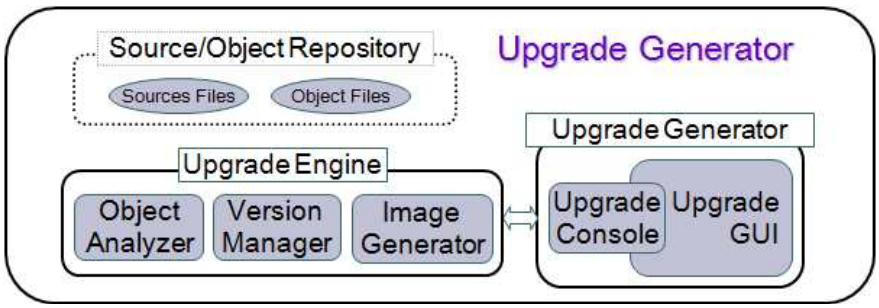



그림 3.2 업그레이드 생성기 모듈 구성도

### 가. 업그레이드 GUI (Upgrade GUI)

업그레이드 GUI (Graphical User Interface) 모듈은 사용자로부터 업그레이드 명령을 입력으로 받아 명령어에 따라 업그레이드 콘솔(upgrade console)에 전달하게 되고 업그레이드 콘솔은 입력 받은 명령어에 따라 업그레이드 엔진에게 보고하는 역할을 한다.

### 나. 업그레이드 콘솔 (Upgrade Console)

업그레이드 콘솔은 업그레이드 GUI와 독립적으로 다른 통합개발환경과 쉽게 통합되기 위해서 존재하며 이러한 업그레이드 콘솔은 업그레이드 GUI로부터 해당 명령을 실제 수행하는 업그레이드 엔진을 호출하는 역할을 한다.

### 다. 업그레이드 엔진 (Upgrade Engine)

업그레이드 엔진은 실제 동적 업그레이드 이미지를 생성하는 모듈이며 오브젝트 분석기(object analyzer)와 버전 관리자(version manager) 이미지 생성기(image generator)를 가지고 있다. 오브젝트 분석기는 업그레이드 이미지를 생성할 때 필요한 심볼 정보를 얻기 위해서 필요하다. 이미지 생성기는 버전 관리자로부터 버전 정보를 요청하고 버전에 따르는 동적 업그레이드 이미지를 생성한다.

### 라. 소스/오브젝트 저장소(Source/Object Repository)

소스/오브젝트 저장소는 업그레이드를 수행하기 위한 정보를 얻기 위해 존재한다. 각 오브젝트에 존재하는 심볼 정보 등을 활용하여 업그레이드 이미지를 작성할 때 필요한 심볼 주소 등 기타 필요한 정보를 제공한다. 이러한 소스/오브젝트 저장소는 업그레이드 생성기에 존재할 수도 있고 SCM (Software Configuration Management)에 존재할 수도 있다.

### 3.2.2 업그레이드 패키지 생성과정

본 논문에서 설계한 업그레이드 이미지 생성 모듈의 수행과정은 그림 3.3과 같다.

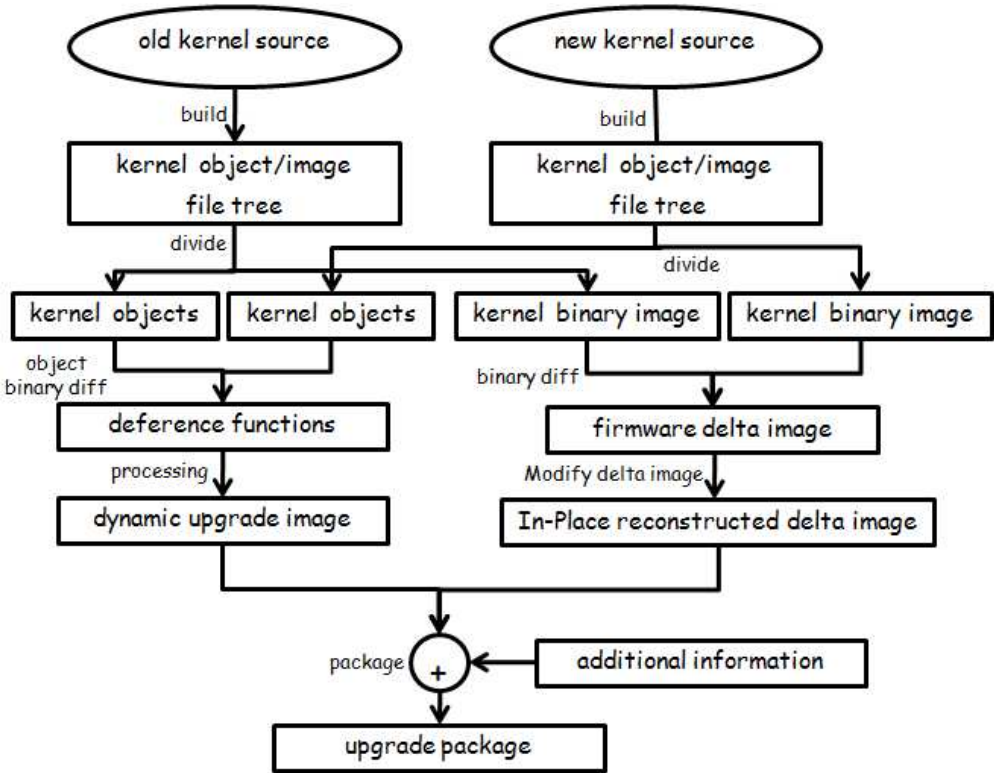



그림 3.3 업그레이드 패키지 생성 과정

업그레이드 생성기에서 업그레이드 패키지를 생성하는 과정은 업그레이드 관리자는 SCM (Software Configuration Management) 시스템으로부터 이전 버전의 소스와 최신 버전의 소스를 다운 받은 후 빌드를 수행한다. 빌드의 결과물인 오브젝트 파일과 실행 바이너리 이미지를 나눈 후 동적 업그레이드 이미지는 오브젝트 파일 비교를 통해 수정된 함수를 찾은 후 심볼 재배치 등 추가적인 작업을 통해 동적 업그레이드 이미지를 생성한다. 실행 바이너리 이미지는 바이너리 비교를 통해 펌웨어 델타 이미지를 생성한 후 임베디드 시스템에서의 델타 업그레이

드를 적용하기 위한 In-Place reconstruction 알고리즘을 사용하여 델타이미지를 수정한다. 최종적으로 생성된 동적 업그레이드 이미지, 펌웨어 델타 이미지 그리고 메타정보를 추가하여 최종 업그레이드 패키지를 작성한다.

## 3.3 업그레이드 서버 (Upgrade Server)

### 3.3.1 업그레이드 서버 시스템 구조도

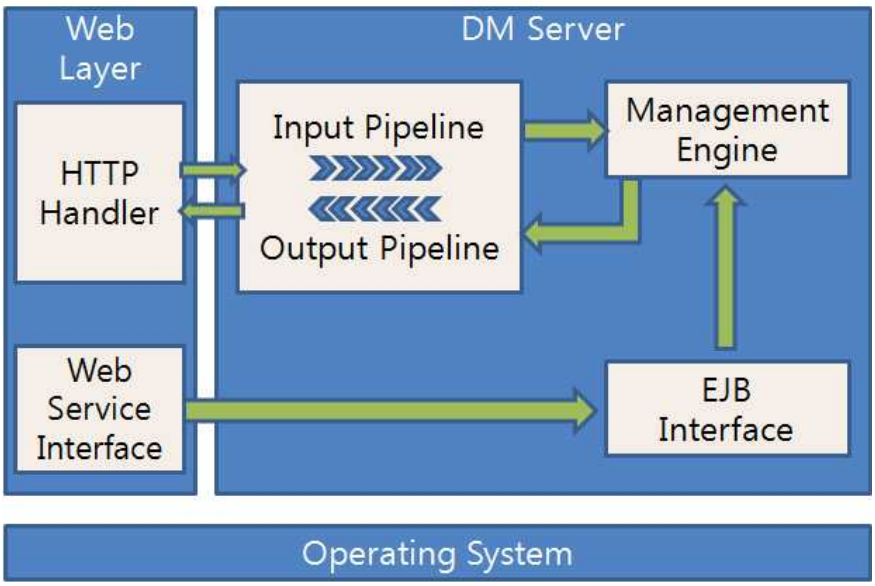



그림 3.4 업그레이드 서버 시스템 구성도

업그레이드 서버 시스템은 웹 서비스 인터페이스를 통해 업그레이드 관리자와의 인터페이스를 제공하며 업그레이드 관리자로부터 받은 입력에 대해 EJB 인터페이스를 이용해 관리 엔진에게 전달한다. 입력을 받은 관리 엔진은 HTTP 핸들러에 맞는 인터페이스를 맞추기 위해 출력 파이프라인, 입력 파이프라인을 거쳐 HTTP 핸들러에게 전달하게 된다. HTTP 핸들러는 단말기와 통신을 위한 인터페이스로서 단말기에 관리 정보를 제공 또는 관리 정보에 대한 단말기의 결과에 대해서 관리 엔진에게 전달하게 된다.

### 3.3.2 업그레이드 서버 모듈 구성도

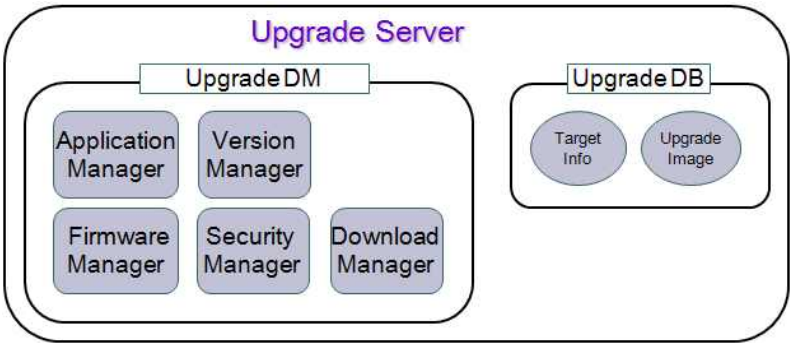



그림 3.5 업그레이드 서버 모듈 구성도

#### 가. 응용프로그램 관리자 (Application Manager)

응용프로그램 관리자는 응용프로그램의 설치와 삭제 등의 응용프로그램을 관리한다.

#### 나. 펌웨어 관리자 (Firmware Manager)

펌웨어 관리자는 응용프로그램 관리와 독립적으로 두어 커널이미지를 관리하는 일을 한다. 응용프로그램과 독립적으로 구분한 이유는 OMA-DM에서는 펌웨어 업그레이드 표준 프로토콜은 FUMO를 사용하고 응용프로그램 업그레이드 SCOMO를 사용하는 구조로 이루어져 있으며 이러한 이유 때문에 응용프로그램 관리자와 펌웨어 관리를 독립적으로 구분하였다.

#### 다. 버전 관리자 (Version Manager)

과거의 소프트웨어 버전과 현재의 소프트웨어 버전에 대해서 어떠한 소프트웨어를 적용해야 되는지에 대한 버전을 관리한다.

#### 라. 보안 관리자 (Security Manager)

단말기와 통신을 하면서 단말기에 대한 정보, 서버에 대한 정보 등이 외부에 노출되지 않도록 암호화 또는 복호화를 담당한다.

### 마. 다운로드 관리자 (Download Manager)

업그레이드 관리자로부터 새로운 펌웨어를 다운받고, 단말기에 새로운 펌웨어를 업로드 하기 위한 역할을 담당한다.

## 3.3.3 업그레이드 서버 관리 시나리오

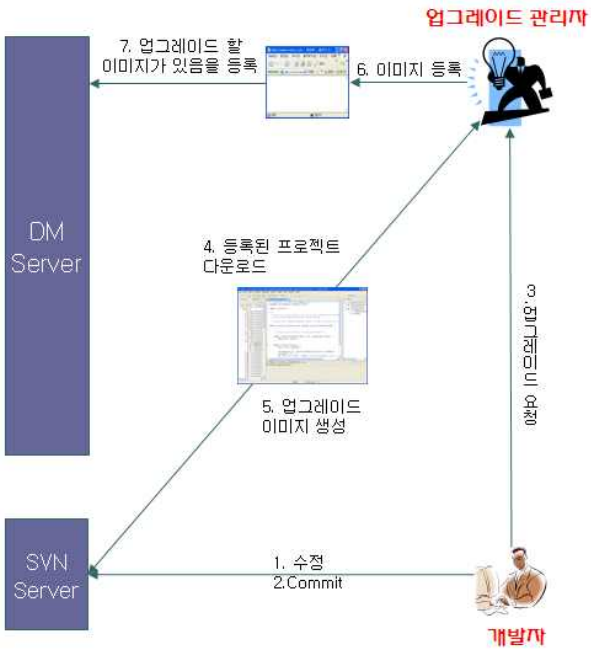



그림 3.6 업그레이드 서버 관리 시나리오

개발자는 Subversion 서버에서 소스코드를 다운받아 수정을 한 후 Commit을 하게 된다. 어느 정도 완성된 버전으로 Commit이 되었으면 업그레이드 관리자에게 업그레이드 이미지가 있음을 알린다. 그러면 업그레이드 관리자는 등록된 프로젝트를 다운받아 업그레이드 이미지를 생성하여 업그레이드 서버의 웹 서비스 인터페이스를 통해 업그레이드 할 이미지가 있음을 등록하게 된다.

## 3.4 타겟 시스템 (Target System)

### 3.4.1 타겟 시스템 구조도

타겟 시스템의 구조도는 그림 3.7과 같이 구성된다.

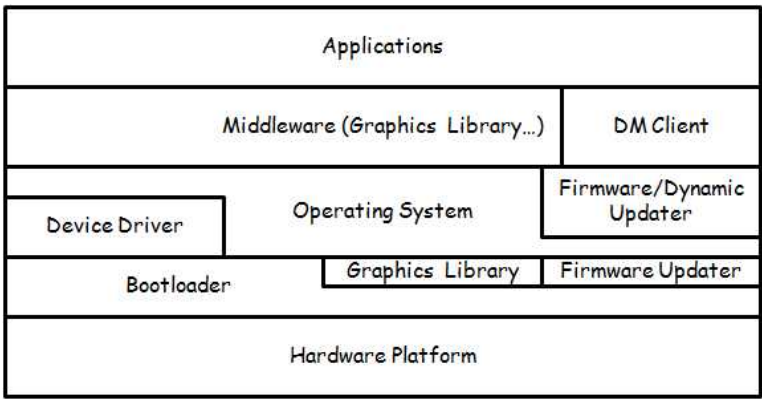



그림 3.7 타겟 시스템 구조도

하드웨어 플랫폼 위에 부트로더가 존재하고 부트로더에 내부에 펌웨어 업그레이드를 위한 모듈과 업그레이드 진행을 표시하기 위한 간단한 그래픽 라이브러리로 구성되며 운영체제 내부에는 플레쉬 메모리와 커널 메모리 영역을 수정하기 위해 펌웨어/동적 업그레이드 모듈이 위치하며 미들웨어 수준에 단말기의 소프트웨 업그레이드를 관리하는 DM 클라인어트 모듈이 존재하는 구조로 이루어진다. 부트로더에 펌웨어 업그레이드 모듈이 존재하는 이유는 운영체제 레벨에서 업그레이드를 진행하다가 도중에 장애가 발생했을 경우에는 부트로더에서 업그레이드를 진행해야 하므로 펌웨어 업그레이드 모듈은 커널영역과 부트로더 영역 모두 존재하는 구조로 이루어진다.

### 3.4.2 타겟 시스템 모듈 구성도

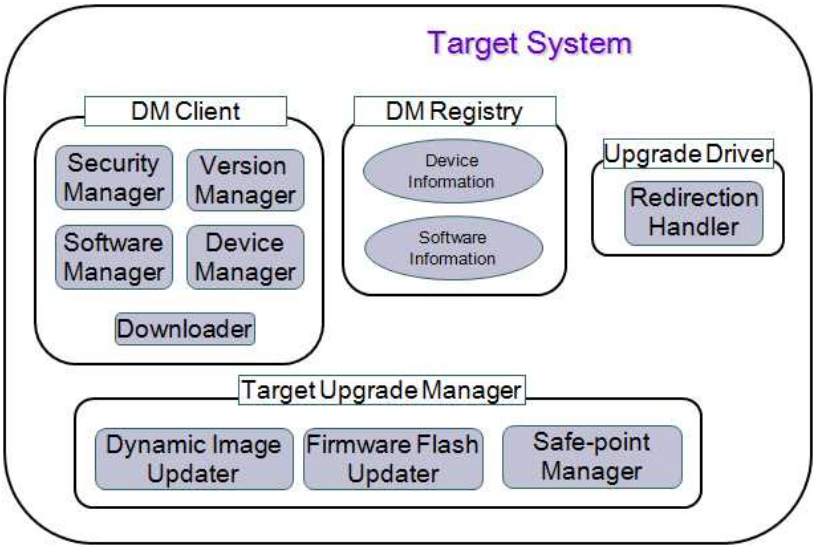



그림 3.8 타겟 시스템 모듈 구성도

#### 가. DM 클라이언트 (Device Manager Client)

DM 클라이언트 단말기에 관련된 소프트웨어와 단말기 정보등을 관리하고 소프트웨어를 다운로드 하는 역할을 수행한다. 내부 모듈로는 보안관리자, 버전관리자, 소프트웨어 관리자, 디바이스 관리자, 다운로드 관리자를 가지며 디바이스 관리자는 DM 저장소를 관리하는 역할을 하며 소프트웨어 관리자는 타겟 업그레이드 관리자를 호출하여 업그레이드를 수행하는 역할을 한다.

#### 나. 업그레이드 드라이버 (Upgrade Driver)

업그레이드 드라이버의 내부에는 redirection handler가 존재하며 redirection handler는 이전 버전의 함수에서 업그레이드된 함수의 실제 주소로 분기할 때 사용되는 모듈이다 .

#### 다. 타겟 업그레이드 관리자 (Target Upgrade manager)

타겟 시스템을 업그레이드 하기 위해 동적 이미지 업데이터, 펌웨어 플래

쉬 업데이터, 안전 지점 관리자를 두어 안전하고 적절한 업그레이드가 수행되기 위한 역할을 담당한다.

### 3.4.3 타겟 시스템 업그레이드 적용 절차

업그레이드 서버로부터 업그레이드 패키지를 DM 클라이언트를 통해 다운받은 후 타겟 업그레이드 관리자는 그림 3.9의 순서로 업그레이드를 수행한다.

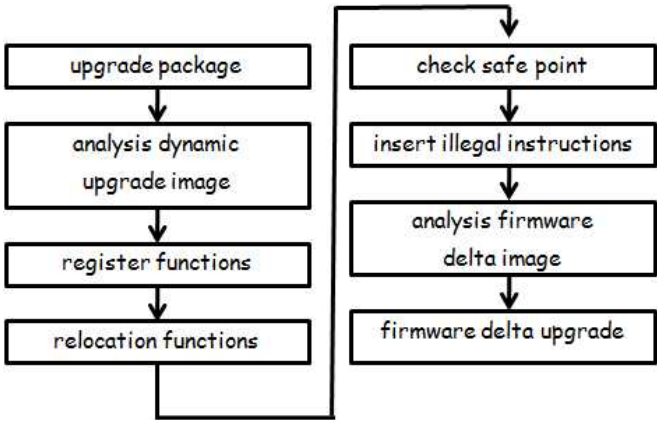



**그림 3.9 타겟 시스템 업그레이드 수행 과정**

업그레이드 수행 과정 중 동적 업그레이드 수행 과정은 동적 업그레이드 이미지를 분석하여 업그레이드 할 함수의 코드를 메모리에 등록한다. 다음으로 등록된 함수의 재배치 작업을 수행한 후 세이프 포인트를 체크를 한다. 마지막으로 업그레이드 된 함수로 분기하기 위한 트랩 코드를 삽입하여 새롭게 추가된 업그레이드 함수로 분기하도록 한다. 동적 업그레이드 수행을 마친 후 다시 펌웨어 델타 업그레이드를 수행 하는 업그레이드 수행 과정을 한다.

### 3.4.4 업그레이드 드라이버 수행 절차

함수단위로 동적업그레이드를 등록하면 실제 수행 중에 이전버전 함수에

서 새로운 함수로 분기를 한다. 이러한 분기는 업그레이드 드라이버에서 수행 하며 수행 절차는 그림 3.10과 같다.

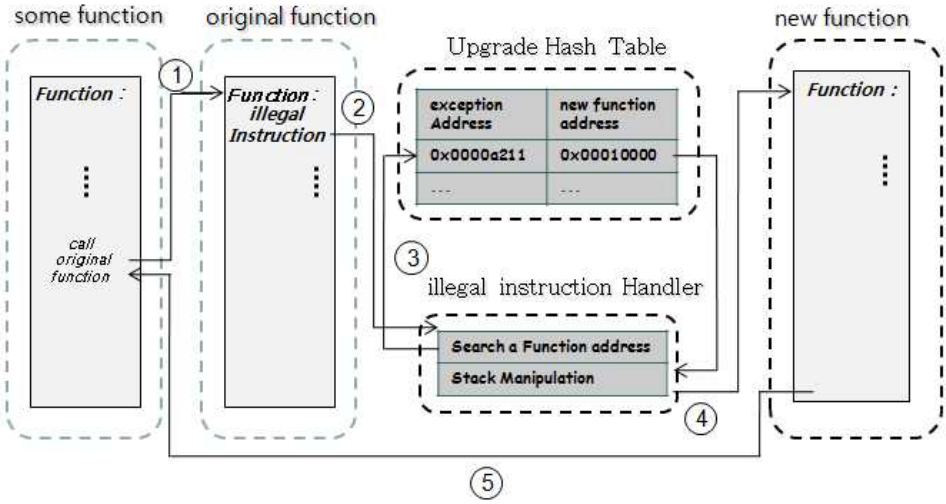



**그림 3.10 업그레이드 드라이버 수행 절차**

먼저 원본 함수를 호출을 하면 동적 업그레이드 등록할 때 삽입한 illegal instruction을 수행하게 되고 시스템은 illegal instruction에 대한 illegal instruction handler로 분기하게 된다. illegal instruction handler에서 Upgrade Hash Table에 존재하는 새로운 함수의 주소를 찾고 새로운 함수의 주소를 illegal exception이 발생할 당시의 스택에 저장되어 있는 복귀 주소를 새로운 함수의 주소로 대체함으로써 illegal exception 처리를 마치고 새로운 함수로 분기하게 된다. 마지막으로 최종 새로운 함수는 수행을 마치고 최초 호출한 함수(some function)로 복귀한다.

### 3.4.5 업그레이드 적용 시나리오

업그레이드 적용 시나리오는 그림 3.11과 같다. 아래 그림의 DM 클라이언트는 타겟 머신이고 DM 서버는 업그레이드 서버를 의미한다. 업그레이드 저장소는 DM 서버와 같이 곳이 위치할 수 있고 다른 곳에 위치할 수

도 있다.

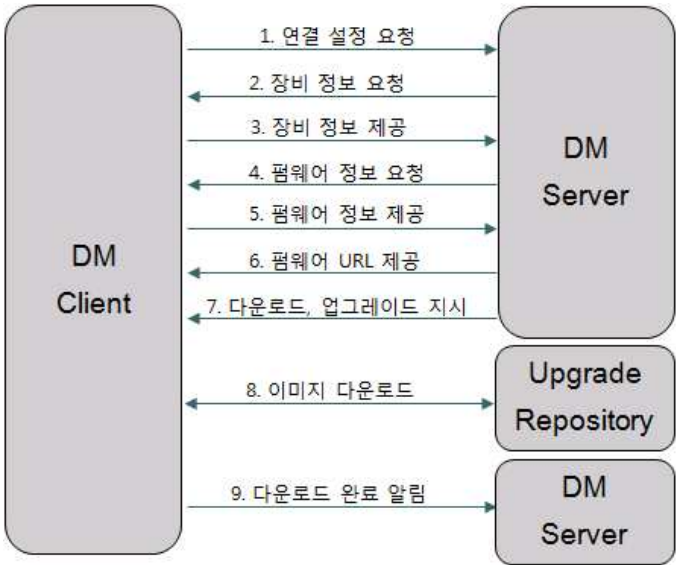



**그림 3.11 업그레이드 적용 시나리오**

최초 DM 클라이언트는 DM 서버에게 연결 설정 요청을 한다. 이 부분은 사용자의 업그레이드 요청에 의해 호출되며 DM 서버는 연결 설정 요청에 대한 응답과 함께 장비 정보를 요청하게 된다. DM 클라이언트는 응답 메시지로 장비 정보를 제공하게 된다. 다음으로 DM 서버는 해당 단말기에 대한 펌웨어 버전 정보를 요청하게 되고 DM 클라이언트는 펌웨어 버전 정보를 DM 서버에 전송하게 된다. DM 서버는 장비 정보와 펌웨어 정보를 통해 업그레이드 패키지를 선택한 후 해당 업그레이드 패키지가 위치한 URL을 단말기에 전송과 동시에 다운로드 업그레이드를 지시하게 된다. DM 클라이언트는 해당 URL을 통해 업그레이드 저장소로부터 업그레이드 패키지를 다운받은 후 다운로드 완료 메시지를 DM 서버에 전송하게 되고 업그레이드를 수행 한다.

# 제 4 장 프로토타입 시스템 구현

## 4.1 프로토타입 시스템 개발 환경

4장에서는 3장에서 설계한 동적 업그레이드 프레임워크를 실제 적용한 내용을 다룬다. 본 논문에서 구현한 프로토타입 시스템 환경은 아래와 같다.

| | | 업그레이드 생성기 |
|---|---|---|
| 하드웨어 | | 인텔 코어 2 듀오 |
| 운영체제 | | 윈도우즈, 리눅스 |
| 개발툴 | 에디터 | 이클립스, 소스 인사이트 |
| | 버전관리 | 서브버전 |
| | 디버거 | GDB 기반의 이클립스 디버거 |
| | 버그 관리 | 엑셀 |
| | 테스팅 | 소스 인스펙션 |
| 개발 언어 | | C , JAVA |

| | | 업그레이드 서버 |
|---|---|---|
| 하드웨어 | | 인텔 코어 2 듀오 |
| 운영체제 | | 리눅스 |
| 개발툴 | 에디터 | 이클립스, 소스 인사이트 |
| | 버전관리 | 서브버전 |
| | 디버거 | GDB 기반의 이클립스 디버거 |
| | 버그 관리 | 엑셀 |
| | 테스팅 | 소스 인스펙션 |
| 개발 언어 | | JAVA |

| | | 타겟 시스템 |
|---|---|---|
| 하드웨어 | | PXA270 기반의 스마트폰 참조 보드 |
| 운영체제 | | 임베디드 리눅스 |
| 개발툴 | 에디터 | 이클립스, 소스 인사이트 |
| | 버전관리 | 서브버전 |
| | 디버거 | JTAG 디버거(A1000), GDB |
| | 버그 관리 | 엑셀 |
| | 테스팅 | 소스 인스펙션 |
| 개발 언어 | | C |

## 4.2 업그레이드 생성기 구현

### 4.2.1 업그레이드 생성기 시퀀스 다이어그램

업그레이드 이미지를 생성하기 위한 과정을 그림 4.1은 보여준다.

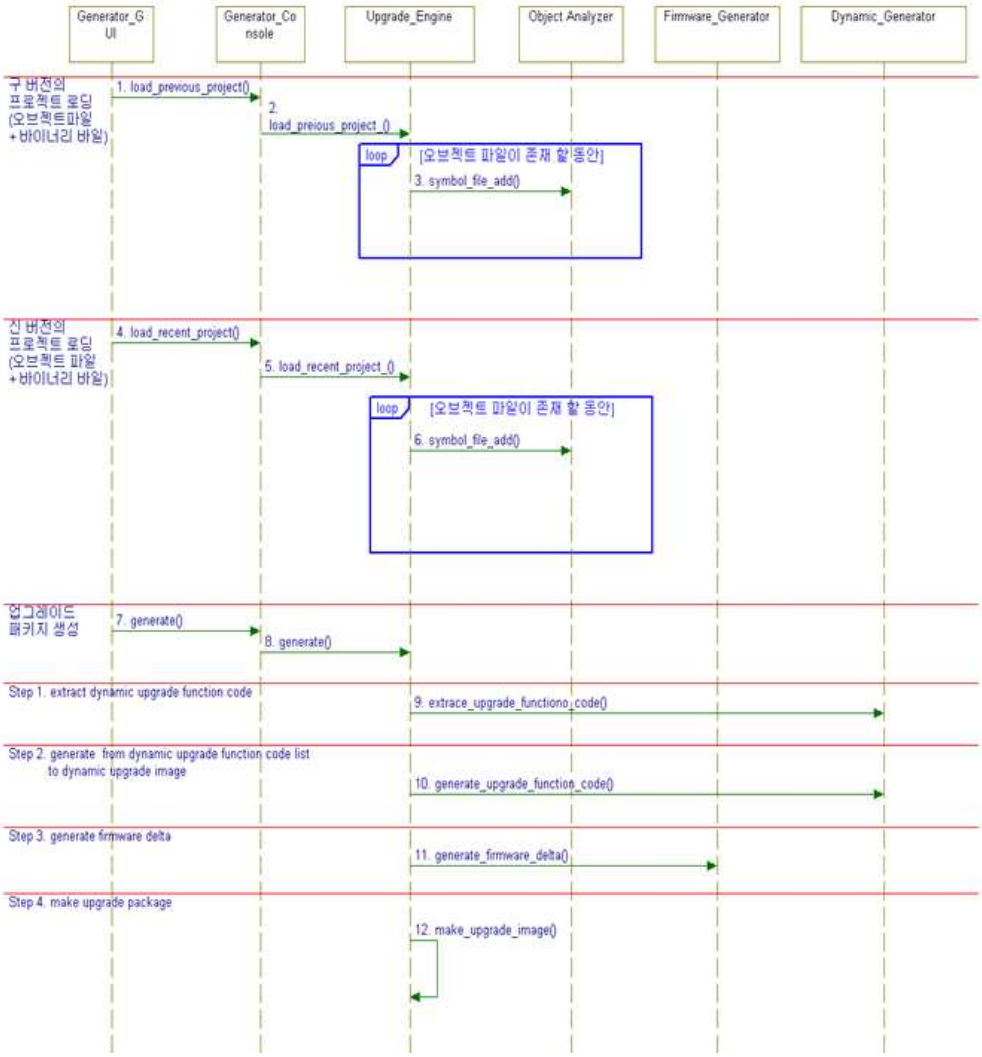



**그림 4.1 업그레이드 생성기 시퀀스 다이어그램**

각 모듈에 대해서 설명하면 Generator_GUI는 사용자로부터 입력을 받은

후 일을 해당 명령어를 Generator_Console에게 전달하는 일을 수행하고 Generator_Console은 Generator_GUI로부터 받은 명령어에 해당하는 업그레이드 엔진 모듈을 호출하는 일을 수행한다. 업그레이드 엔진은 각 오브젝트 분석기와 펌웨어 업그레이드 이미지 생성기와 동적 업그레이드 이미지 생성기를 호출하여 업그레이드를 수행한다.

수행 흐름을 보면 사용자로부터 구 버전의 프로젝트 경로를 입력 받으면 오브젝트 분석기는 프로젝트의 모든 오브젝트를 읽은 후 리스트에 저장한다. 다음으로 신 버전의 프로젝트 경로를 입력 받으면 오브젝트 분석기가 모든 오브젝트를 리스트에 저장한다. 다음으로 사용자의 업그레이드 이미지 생성 요청을 받으면 먼저 구 버전의 오브젝트 리스트와 신 버전의 오브젝트 리스트로부터 수정된 함수 코드를 추출한다. 다음으로 추출된 함수 코드를 가지고 동적 업그레이드 이미지를 생성한다. 동적 업그레이드 이미지가 완성되면 펌웨어 업그레이드 이미지 생성을 수행하며 마지막으로 메타 정보 등을 추가하여 최종 업그레이드 패키지를 생성한다.

### 4.2.2 업그레이드 GUI

#### 가. 이클립스 플러그인

본 논문은 업그레이드 이미지 생성 GUI를 이클립스 플러그인으로 개발하였다. 이클립스는 개발환경과 업그레이드 관리환경을 하나의 툴로 통합할 수 있는 장점을 가지고 있다. 또한 이클립스를 활용하면 버전관리 툴인 CVS 또는 서브버전을 활용하여 버전 관리자를 독립적으로 구현할 필요가 없으므로 본 논문에서는 이클립스 플러그인 기반의 업그레이드 GUI를 구성하였다.

#### 나. 업그레이드 이미지 생성 창

업그레이드 이미지 생성 창은 그림 4.2와 같다. Product name에는 업그레이드 패키지를 만들 때 사용할 정보를 입력하는 곳이고 구 버전 프로젝트 URL와 신 버전 프로젝트의 URL을 입력 받아 이미지를 생성한다.

이미지 생성 버튼을 클릭하면 Option 탭에서 설정된 정보를 입력으로 받아 업그레이드 콘솔에게 커맨드를 전달하여 최종 업그레이드 이미지를 생성한다.

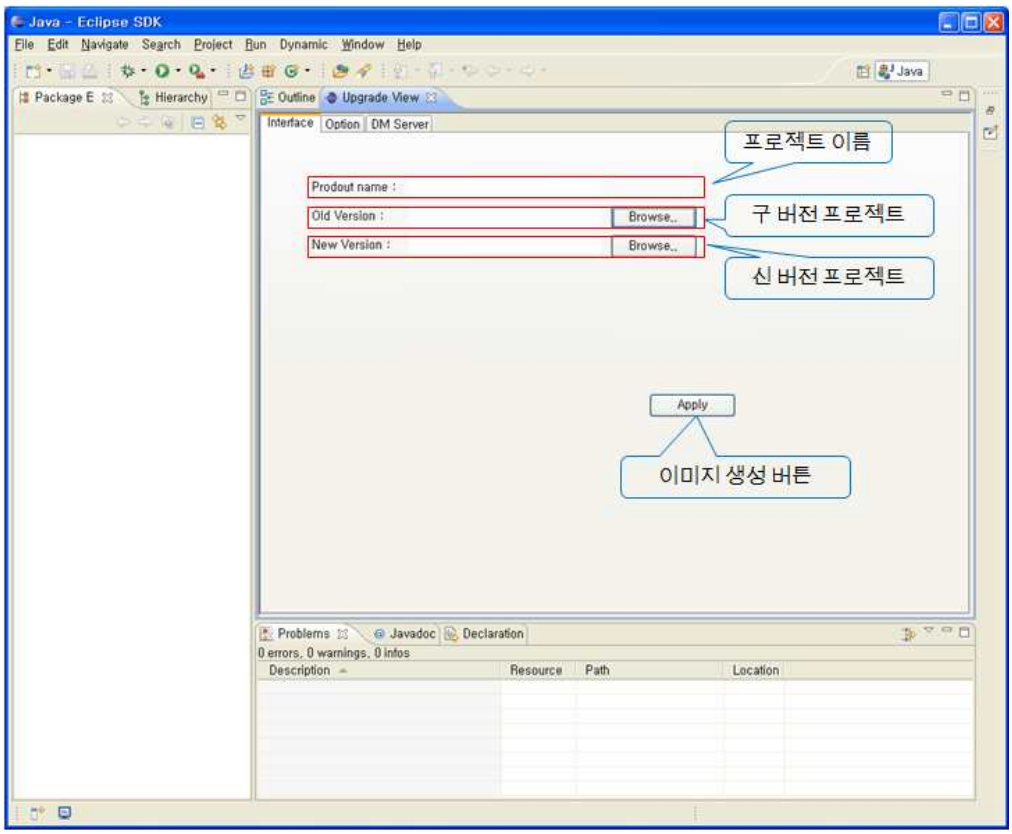



그림 4.2 업그레이드 이미지 생성 창

### 나. 옵션 설정 창

업그레이드 이미지를 생성할 때 필요한 옵션을 설정하는 창은 그림 4.2과 같다. 우선 업그레이드에 필요한 일반적인 옵션으로는 동적 업그레이드를 Disable하는 체크박스를 두었다. 동적 업그레이드를 Disable 체크를 하면 업그레이드 패키지를 생성할 때 펌웨어 업그레이드만 적용하는 업그레이드 패키지를 생성한다. 펌웨어 업그레이드 Disable 체크도 역시 동적 업그레이드만 적용할 때 사용하는 체크박스이고 Rollback Image 체크는 타겟

에서 이전 버전으로 돌아오기 위한 롤백 이미지를 생성하는 창이다.

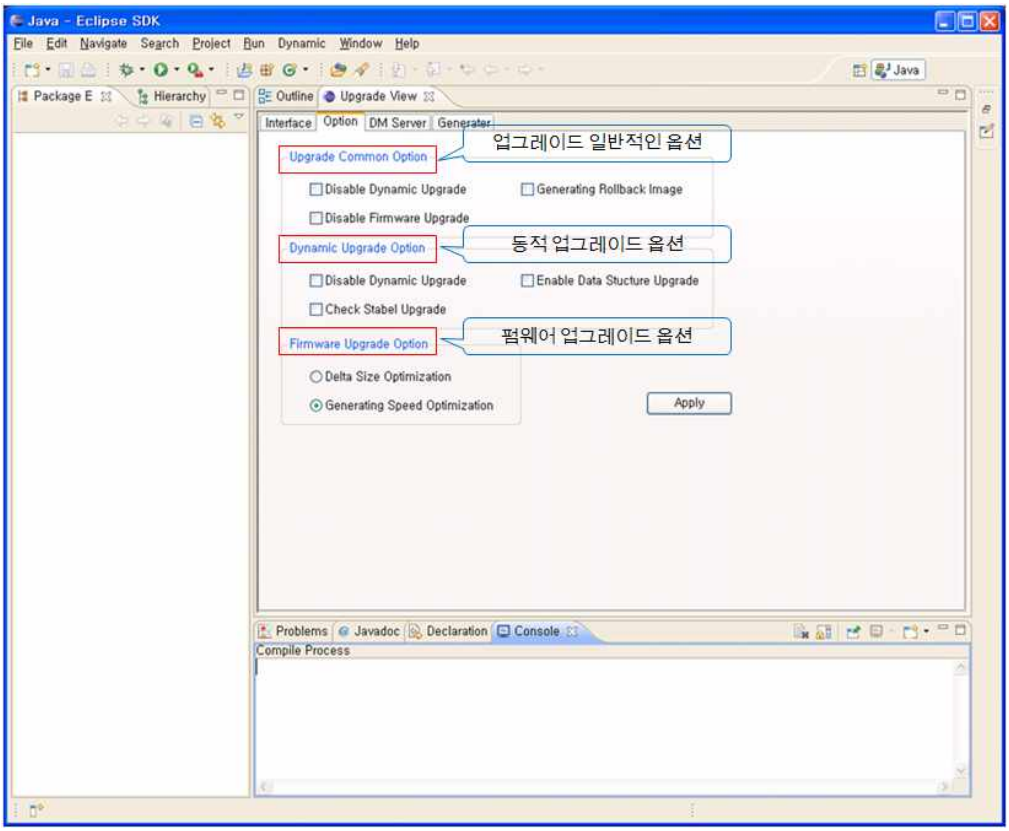



**그림 4.3 옵션 설정 창**

### 다. DM 서버 관리 창

DM 서버 관리 창은 업그레이드 서버인 OMA-DM 서버에 로그인 하여 서버의 DM 또는 서버의 환경을 설정하기 위해 만든 창이다. 본 논문에서는 DM 서버 관리 하는 업그레이드 엔진 부분은 구현하지 못한 상태이며 차후 구현할 예정이다.

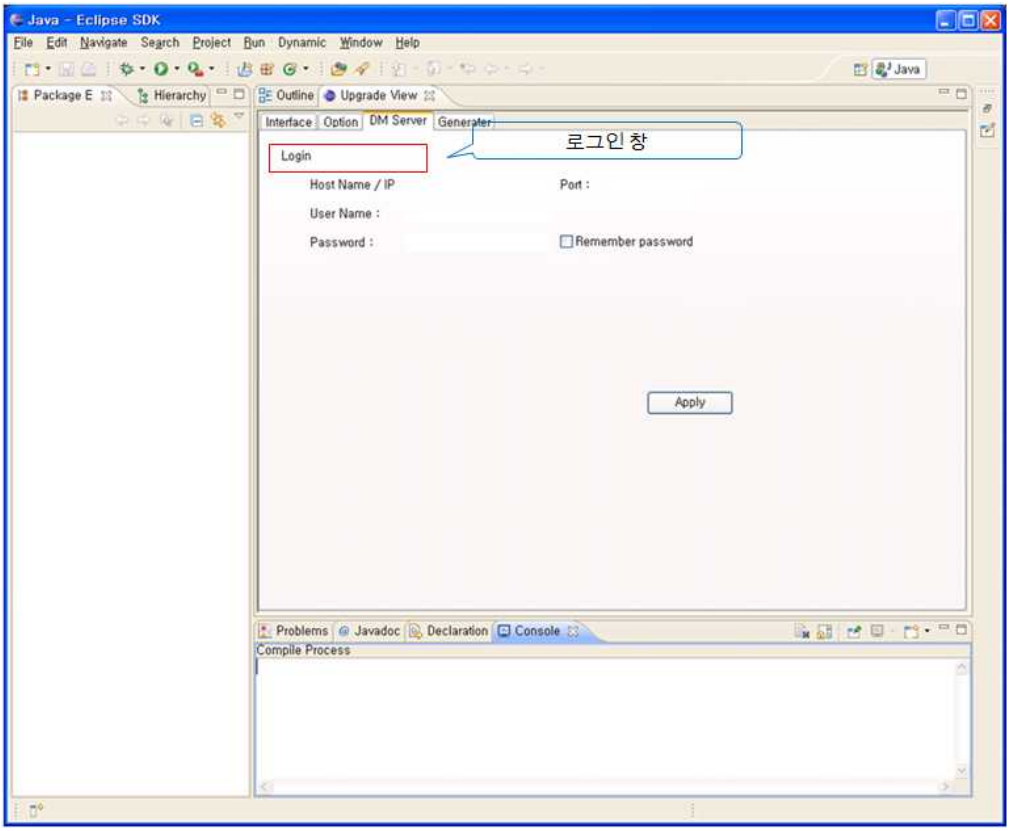



그림 4.4 DM 서버 관리 창

### 4.2.3 업그레이드 콘솔

본 논문에서 제안한 업그레이드 콘솔은 CLI(Command Line Interface)로 구현 하였다. 리눅스 디버거인 GDB와 같은 구조로 이루어진다. 업그레이드 콘솔은 다른 통합개발환경과 쉽게 연동이 가능한 구조로 이루어져있다. 본 논문에서는 이클립스 GUI와 업그레이드 콘솔이 통신하는 방식으로 구현하였다.

업그레이드 콘솔에 해당 명령어를 추가하기 위해서는 add_com() 함수를 사용하여 추가한다. 첫 번째 인자는 업그레이드 콘솔이 처리할 명령어이고 두 번째 인자는 업그레이드 콘솔에서 명령어가 입력될 때 호출되는 함수

이고 세 번째 인자는 설명을 나타낸다.

```
add_com ("load-previous-project",
            class_obscure, load_previous_repository,
            _("Dynamic Upgrade load previous version project repository."));
add_com ("load-recent-project",
            class_obscure, load_recent_repository,
            _("Dynamic Upgrade load recent version project repository."));
```

### 4.2.4 업그레이드 엔진

업그레이드 엔진은 업그레이드 콘솔의 입력을 받아 수행하게 되도록 구현되어 있다. 현재 본 논문에서 구현한 업그레이드 엔진의 세부 구조는 그림 4.5와 같다.

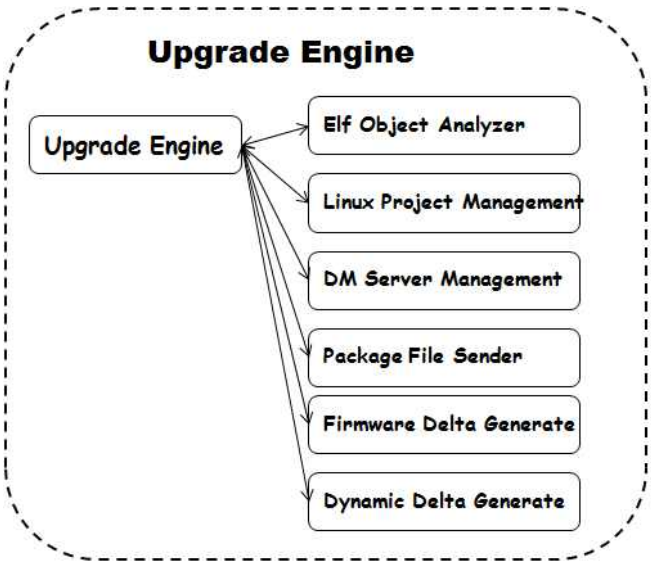



**그림 4.5 업그레이드 엔진 세부 구조**

업그레이드 엔진은 동적 업그레이드 델타 이미지를 생성하기 위한 ELF

오브젝트 분석기, 업그레이드 처리를 위한 리눅스의 프로젝트 관리자, 업그레이드 DM서버를 관리하기 위한 DM 서버 관리자, 업그레이드 패키지를 서버로 전송하기 위한 업그레이드 패키지 전송자, 펌웨어/동적 업그레이드 이미지 생성기로 구성되어 있다. 업그레이드 엔진의 자세한 기능에 대해서는 다음 절 부터 설명한다.

### 4.2.5 ELF 오브젝트 분석기

본 논문에서 구현한 오브젝트 분석기는 다음과 같은 구조로 이루어져 있으며 대부분 공개 소스 프로젝트인 GDB에서 사용한 자료 구조를 수정하여 구현 하였다. 또한 내부적으로 ELF 오브젝트 뿐만 아니라 COFF, PE 파일 포맷등도 지원하기 위해 GNU의 BFD 라이브러리를 사용하였다. BFD 라이브러리란 오브젝트 파일 형식이 어떤 것이던 동일한 루틴을 사용해 오브젝트 파일에 접근하기 위해 GNU에서 만든 라이브러리 패키지이다. BFD는 크게 Front-end와 Back-end 두 부분으로 나누어져 있다. Front-end는 라이브러리 사용자에게 제공되는 Interface이며, Back-end는 특정 오브젝트 파일 형식을 접근할 수 있는 함수와 자료로 이루어져 있다.

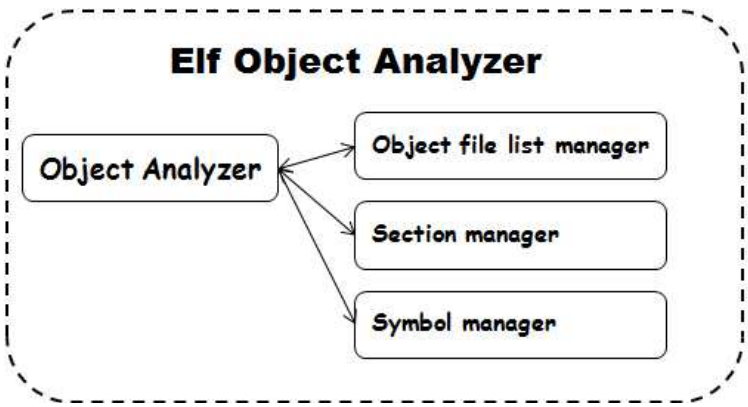
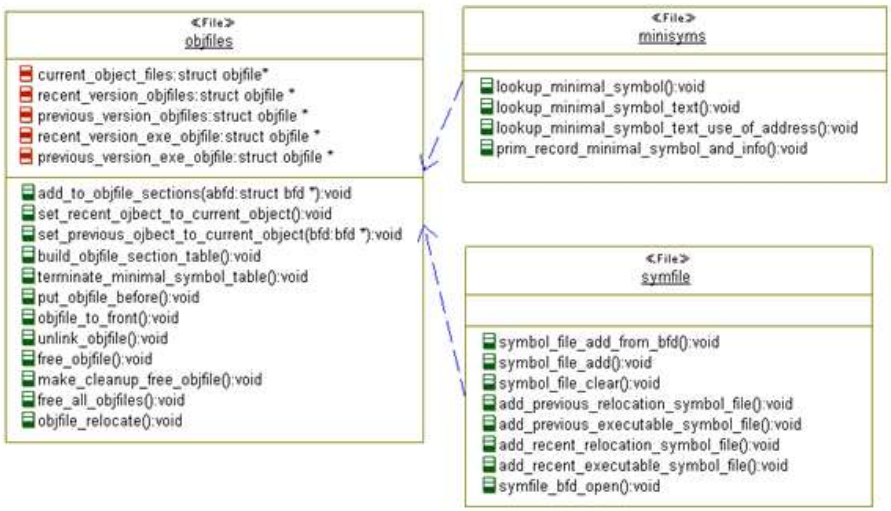



그림 4.6 ELF 오브젝트 분석기

오브젝트 분석기는 크게 오브젝트 파일을 관리하기 위한 오브젝트 파일

리스트 관리자와 ELF 오브젝트에서의 text, data, bss 섹션등을 관리하기 위한 섹션 관리자, 업그레이드 심볼 관리자로 이루어진다.

### 가. File Diagram




objfile.c
ELF 파일의 심볼과 섹션 구조를 리스트로 관리하기 기능을 가진 파일
minisyms.c
오브젝트 파일 내에 있는 심볼 주소를 찾을 때 사용되는 기능을 가진 파일
symfile.c
BFD 라이브러리 초기화와 섹션 처리 그리고 오브젝트 파일을 읽는 기능을 가진 파일

## 4.2.6 리눅스 프로젝트 관리자

본 논문에서 사용한 리눅스 프로젝트 관리자는 다음과 같은 구조의 프로젝트 파일을 읽은 후 처리한다. 프로젝트 설정 파일을 생성하기 위해서는 표 4.1과 같은 방법으로 리눅스 유틸리티인 grep 명령어를 활용하면 되며 실제 본 논문에서는 개발의 편의를 위해 자동적으로 프로젝트 파일을 자동적으로 생성하지 않고 저자가 직접 프로젝트 파일을 수정하여 생성하였

다. 차후 xml 기반으로 자동 생성하도록 수정 할 예정이다. 본 논문에서 생성한 동적 업그레이드를 위한 리눅스 프로젝트 파일의 구조와 해당 프로젝트 파일을 분석하여 처리하는 소스는 다음과 같다.

**표 4.1 리눅스 프로젝트 파일**

```
@execute_symbol:./vmlinux                    # 심볼정보를 포함한 리눅스 이미지
@binary_file:./arch/asm/boot/zImage          # 심볼정보를 제거한 리눅스 바이너리 이미지

@objects:                                    # 오브젝트 파일
fs/aio.o
fs/attr.o
fs/bad_inode.o
fs/binfmt_elf.o
fs/binfmt_misc.o
fs/binfmt_script.o
fs/bio.o
fs/block_dev.o
fs/buffer.o
fs/char_dev.o
fs/dcache.o
fs/devpts/devpts.o
fs/devpts/inode.o
fs/direct-io.o
fs/dnotify.o
fs/eventpoll.o
fs/exec.o
```

### 4.2.7 DM 서버 관리자

본 논문에서는 DM 서버 관리 부분은 구현되지 못한 상태이다. DM 서버 관리자는 웹 서비스를 제공하는 DM 서버로 로그인 하여 웹 서비스를 위한 프로토콜인 SOAP 프로토콜을 사용하여 원격 관리를 할 수 있도록 구현할 예정이다.

### 4.2.8 패키지 파일 전송기

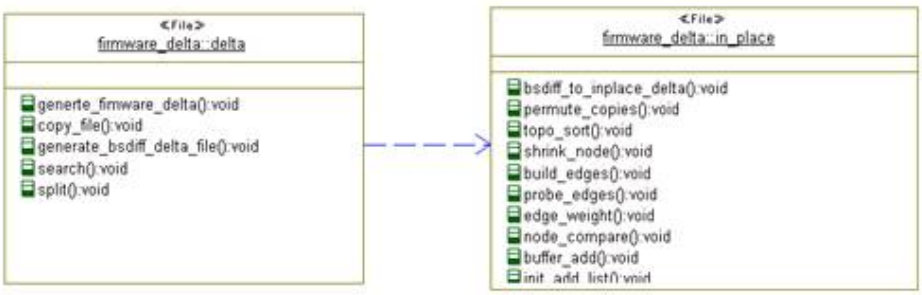
패키지 파일 전송기는 생성된 업그레이드 이미지를 업그레이드 서버로 전송하기 위해 필요하며 본 논문에서는 공개소스 툴인 wput을 활용하여 파일 서버에 업그레이드 패키지를 전송하는 모듈을 구현 하였다.

| wput [options] [file] ftp://[username]:[password]@[hostname]:[prot][/path/][file] |
|---|

### 4.2.9 펌웨어 업그레이드 델타 생성기

펌웨어 델타 생성기는 BSDIFF를 수정하여 작성하였다. BSDIFF에서 사용한 알고리즘을 임베디드 시스템에 적용하려면 In-place 업그레이드가 가능하게 수정해야 한다. 본 논문에서 구현한 델타 생성기는 BSDIFF에서 생성된 델타 파일을 분석하여 In-place 업그레이드가 가능 하도록 작성하였다.

#### 가. File Diagram




| delta.c<br>BSDIFF에서 사용한 알고르즘을 활용하여 펌웨어 델타 이미지를 생성하는 기능을 가진 파일<br>in_place.c<br>BSDIFF로 생성된 델타 이미지를 In-Place 업그레이드가 가능하도록 재구성하는 기능을 가진 파일 |
|---|

#### 나. 활동 다이어그램

펌웨어 델타 생성을 위한 활동 다이어그램은 그림 4.7과 같다. 펌웨어 델

타 생성 타입에 따라 두 종류로 펌웨어를 생성하게 된다. 우선 델타 업그레이드 이미지 생성이 불가능한 경우와 In-place 델타 이미지 생성으로 구분하여 델타 이미지로 생성 안할 경우에는 신 버전의 이미지를 가리지고 펌웨어 업그레이드 이미지로 생성하고 델타를 적용하면 BSDIFF 알고리즘을 사용하여 1차적으로 델타 파일을 생성한다. 이러한 BSDIFF 델타 파일은 2.2.4절에서 설명 했듯이 임베디드 시스템에 적용하려면 메모리가 추가적으로 필요하게 되는 문제가 있다.

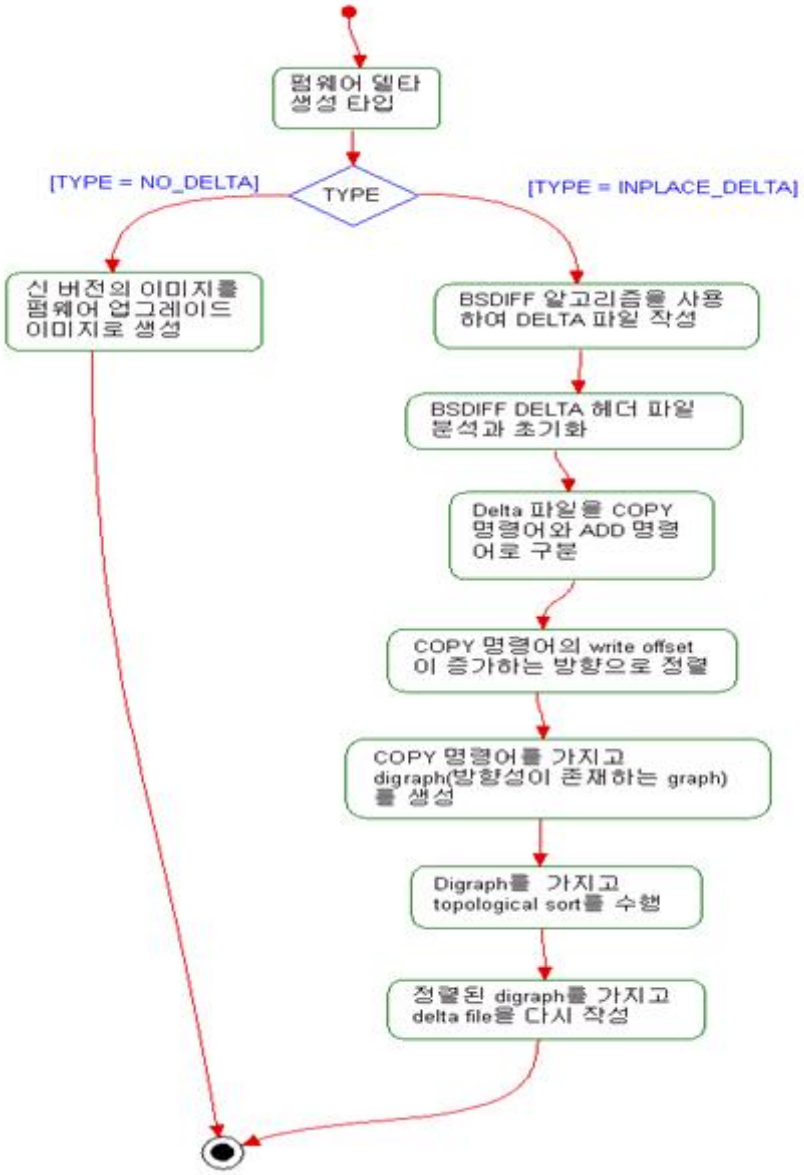



**그림 4.7 펌웨어 업그레이드 델타 생성기 활동 다이어그램**

이러한 문제를 해결하기 위해 BSDIFF 델타 파일을 분석하여 COPY명령어와 ADD명령어로 다시 구분한 후 COPY명령어의 write offset이 증가하는 방향 즉 원본이미지에서 새로운 이미지로 복사하기 위한 메모리 위치가 증가하는 방향으로 정렬을 수행 한 후 COPY명령어를 방향성이 존재

하는 그래프를 생성하게 된다. 다음으로 방향성이 존재하는 그래프를 가지고 위상 정렬을 수행하면 COPY명령어가 수행하기 위한 순서가 정해진다. 마지막으로 정렬된 COPY명령어를 가지고 새로운 델타 파일을 생성하는 과정을 통해 펌웨어 델타 이미지를 생성한다.

### 다. 펌웨어 업그레이드 이미지 구조

펌웨어 델타 파일은 최종적으로 표 4.2와 같은 구조로 생성된다.

**표 4.2 펌웨어 업그레이드 이미지 구조**

| Firwmare Header | Control Block | Difference Block | Extra Block |
|---|---|---|---|

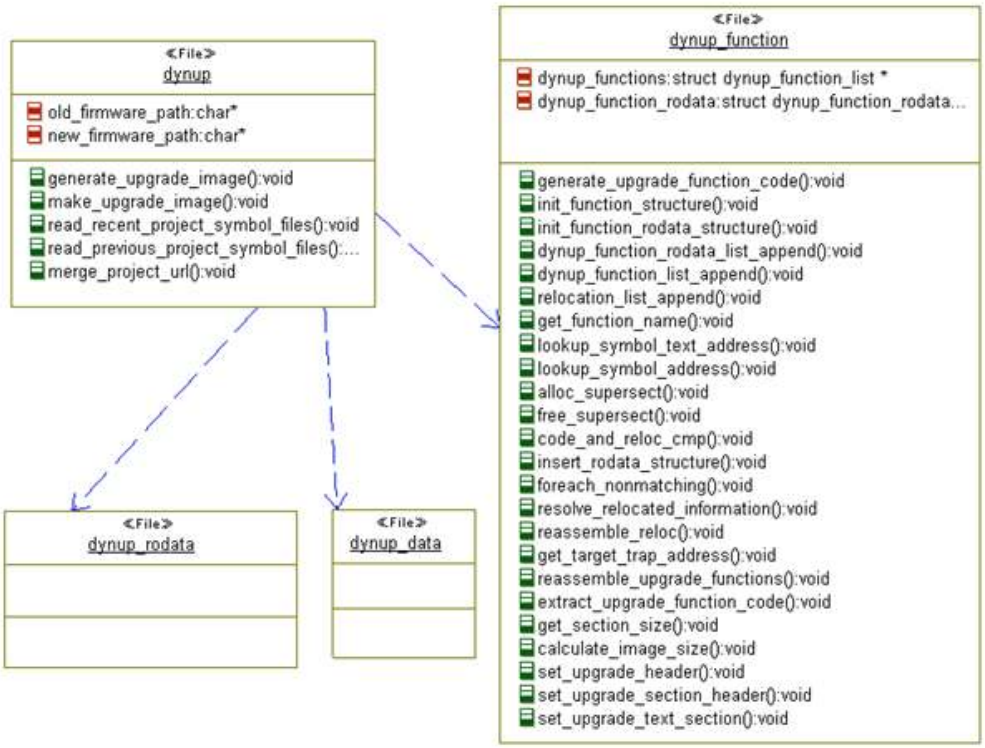
각각에 대해서 설명하면 Firmware Header 필드는 펌웨어 압축방법과 각 블록 사이즈에 대한 정보를 저장한다. Firmware Header의 내용에는 구체적으로 다음과 같은 내용으로 구성 하였다. 총 48 바이트로 구성되어 있으며 8바이트의 "IPDIFF10"이라는 펌웨어 압축방법과 다음 8바이트는 Control Block 내에서 COPY 명령어 길이 그리고 다음 8바이트는 bzip2로 압축된 Control Block의 길이 다음 8바이트는 bzip2로 압축된 Difference Block의 길이 다음 8바이트는 패치가 수행되어 나온 결과물의 크기 다음은 COPY 명령어 길이와 마지막 8바이트는 ADD 명령어의 길이를 나타낸다.

| | | |
|---|---|---|
| 0 | 8 | "IPDIFF10" |
| 8 | 8 | length of copy command size of ctrl block |
| 16 | 8 | length of bzip2ed ctrl block |
| 24 | 8 | length of bzip2ed diff block |
| 32 | 8 | length of new file |
| 40 | 8 | length of copy count |
| 48 | 8 | length of add count |

### 4.2.10 동적 업그레이드 델타 생성기

함수 단위로 동적 업그레이드를 수행하기 위해서는 구 버전의 프로젝트와 신 버전의 프로젝트를 오브젝트 비교하여 수정된 함수의 코드와 함수 내부에서 사용되는 읽기 전용 데이터를 추출한 한다. 추출된 함수의 코드는 오브젝트 파일에서 추출되었기 때문에 함수 코드를 타겟에서 실행 할 수 있도록 심볼을 재배치 한 후  최종 업그레이드 이미지를 생성한다.

#### 가. File Diagram




| |
|---|
| dynup.c<br>업그레이드 콘솔로부터 입력받은 명령어를 해석하고 수행하는 일을 하는 파일<br>dynup_function.c<br>함수 단위로 업그레이드를 수행하기 위해 업그레이드 코드 비교, 동적 업그레이드 이미지 생성에 관한 일을 하는 파일 |

### 나. 오브젝트 파일 비교 활동 다이어그램

동적 업그레이드 델타 생성을 위한 활동 다이어그램은 그림 4.8과 같다.

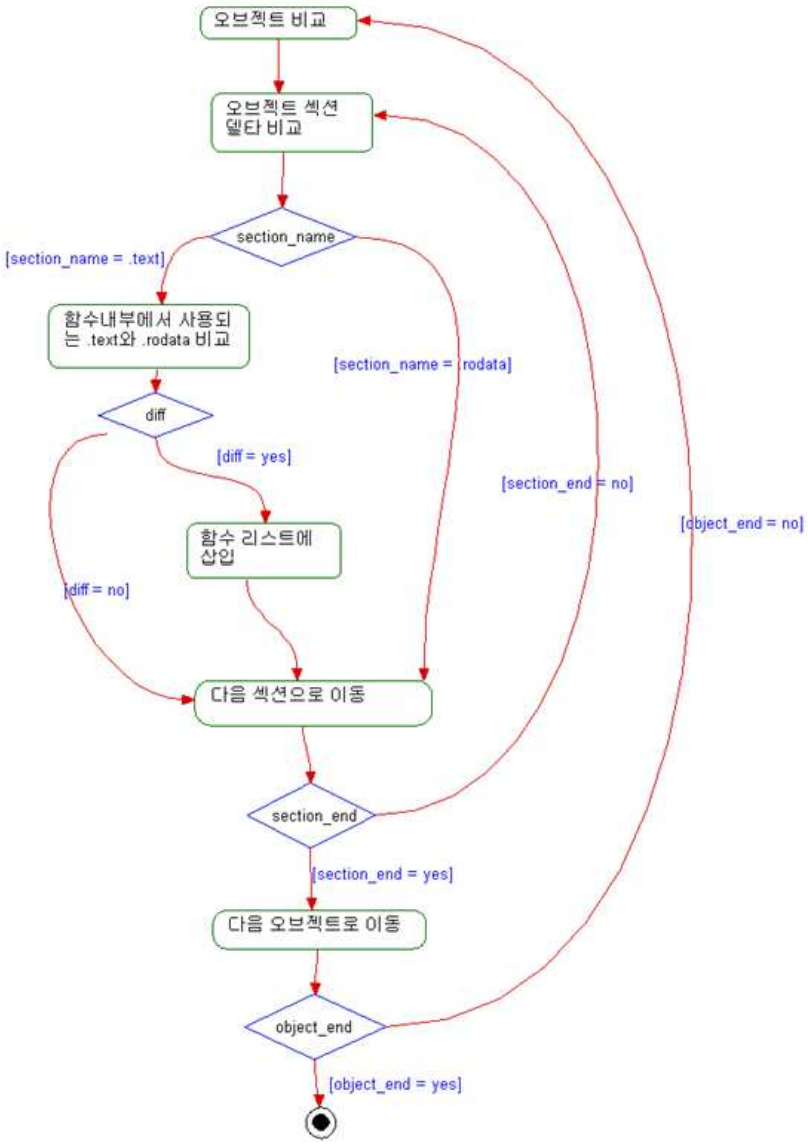



**그림 4.8 오브젝트 비교 활동 다이어그램**

동적 업그레이드 델타 생성은 펌웨어 델타에서 수행한 실행 이미지를 통해 델타 생성을 하는 것이 아니고 재배치가 안 된 오브젝트 파일을 통해

델타를 수행한다. 이러한 오브젝트 단위 델타 추출을 위해 우선 프로젝트의 오브젝트 이름을 비교한 후 이름이 같은 오브젝트로부터 오브젝트 내부에 있는 섹션을 비교하게 된다. 만약 .text 섹션이라면 .text 섹션 내부의 함수코드와 함수 내에서 사용한 읽기전용 데이터인 .rodata 섹션 비교를 수행 하고 내용이 다를 경우 함수 리스트에 삽입한다.

### 다. 업그레이드 함수 코드 재배치 활동 다이어그램

오브젝트 파일 비교를 통해 추출된 오브젝트의 함수 코드는 표 4.3과 같이 재배치가 안 된 코드를 생성한다. 이 함수의 코드는 타겟에 전송되어 실행되면 심볼 해결이 안 되었으므로 문제가 발생하게 된다. 본 논문에서 구현한 동적 업그레이드는 리눅스의 built-in된 커널을 대상으로 구현하였으므로 링크를 수행한 이미지를 보면 함수의 물리적인 메모리 위치를 얻을 수 있다. 우리는 이러한 정보를 활용하여 심볼 해결을 수행 하였다.

**표 4.3 재배치 안 된 코드**

```
 1c:    e5923000   ldr     r3, [r2]
 20:    e1a06001   mov     r6, r1
 24:    e1a0100a   mov     r1, sl
 28:    e88a0018   stmia   sl, {r3, r4}
 2c:    e1a05000   mov     r5, r0
 30:    e5907094   ldr     r7, [r0, #148]
 34:    ebffffe   bl      34 <vfat_mkdir+0x34>
 38:    e2504000   subs    r4, r0, #0  ; 0x0
 3c:    b1a08004   movlt   r8, r4
 40:    ba00003f   blt     144 <.text.vfat_mkdir+0x144>
 44:    e24b9044   sub     r9, fp, #68 ; 0x44
 48:    e1a00005   mov     r0, r5
 4c:    e2861018   add     r1, r6, #24 ; 0x18
 50:    e3a02001   mov     r2, #1      ; 0x1
 54:    e1a03004   mov     r3, r4
 58:    e58da000   str     sl, [sp]
 5c:    e58d9004   str     r9, [sp, #4]
 60:    ebffffe   bl      60 <vfat_mkdir+0x60>
```

그림 4.9는 오브젝트 비교를 통해 추출된 함수 리스트의 함수 코드 중 재배치 안 된 심볼에 대해서 심볼 해결하는 함수 코드 재배치 활동 다이어

그램이다.

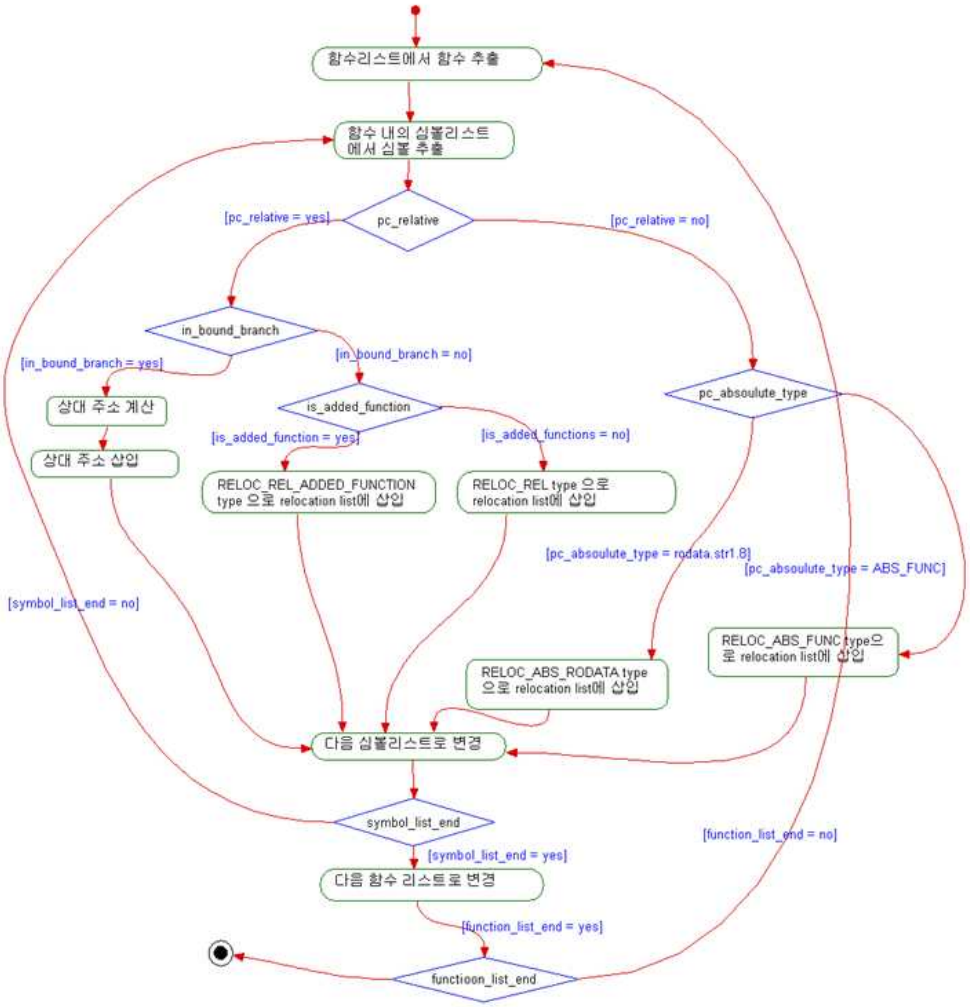



**그림 4.9 업그레이드 코드 재배치 활동 다이어그램**

먼저 함수리스트에서 함수를 얻은 후 해당 함수 내에서 재배치가 안 된 심볼을 얻는다. 해당 심볼이 pc_relative이면 즉 상대주소로 계산된 심볼이라면 다시 함수 내의 코드로 분기하는 심볼과 함수 외부로 분기하는 심볼인지 판별하게 되고 함수 내부로 분기하는 심볼이면 상대 주소를 계산한 후 상대 주소를 삽입을 한다. 만약에 함수 외부로 분기하는 심볼이면 외부

함수가 기존에 있는 함수가아니라 추가된 함수일 수 있는데 이러한 경우에는 RELOC_REL_ADDED_FUNCTION 타입으로 재배치 리스트에 삽입한다. 이 재배치 리스트는 차후 동적 업그레이드 이미지에 포함하여 타겟에 전송한 후 타겟에서 주소를 다시 계산한 후 업그레이드를 수행하는 정보를 가지고 있는 리스트이다. 만약 외부 함수가 기존에 있는 함수라면 RELOC_REL 타입으로 재배치 리스트에 삽입한다.

다음으로 심볼이 pc_absolute 타입이라면 이 경우도 두 가지 조건으로 다시 나누어지며 하나는 함수내부에서 사용하는 문자열일 경우와 분기할 주소가 절대 주소인 경우로 나누어지며 각각 RELOC_ABS_RODAT와 RELOC_ABS_FUNC 타입으로 재배치 리스트에 삽입한 후 타겟에서 동적 업그레이드 적용 시 다시 주소계산을 수행할 수 있도록 하였다.

### 라. 동적 업그레이드 이미지 구조

동적 업그레이드 이미지는 ELF 파일 포맷 구조와 유사하게 작성하였으며 표 4.4와 같다.

**표 4.4 동적 업그레이드 이미지 구조**

| Dynamic Header | Text Section | Rodata Section | Data Section |
|---|---|---|---|
| Relocation Section | Symbol Section | String Section | Section Header |

#### Dynamic Header

동적 업그레이드 이미지의 전반적인 정보

```
struct dynup_hdr {
    unsigned char ident[NIDENT];      /* magic umber and other info */
    unsigned long version;            /* version */
    unsigned long shoff;              /* section header table file offset */
    unsigned short hsize;             /* header size in bytes */
    unsigned short shentsize;         /* section header table entry size */
    unsigned short shnum;             /* section header table entry count */
};
```

### Text Section

함수 단위의 코드 이미지

### Rodata Section

읽기 전용 데이터

### Data Section

전역 변수와 같은 일반적인 데이터

### Relocation Section

타겟에서 동적업그레이드 이미지를 적용할 때 재배치를 위한 정보

```
struct dynup_reloc_hdr {
    unsigned long r_type;              /* Relocation Type */
    unsigned long r_offset;            /* Relocation position */
    unsigned long r_address;           /* Relocation address */
    unsigned long r_str_offset;        /* String offset */
};
```

### Symbol Section

함수 심볼에 대한 정보

```
struct dynup_symbol_hdr {
    unsigned long st_type;             /* Symbol Type */
    unsigned long st_address;          /* Symbol address */
    unsigned long st_str_offset;       /* Symbol string section offset */
    unsigned long st_size;             /* Symbol size */
    unsigned long st_num_reloc;        /* Symbol relocation count */
};
```

### String Section

심볼 이름에 대한 문자열

### Section Header

각 섹션에 대한 정보를 포함하고 있으며 실제 내용은 다음과 같다.

```
struct dynup_section_hdr {
    unsigned long sh_name;          /* Section name (string tbl index) */
```

```
    unsigned long sh_type;          /* Section type */
    unsigned long sh_offset;        /* Section file offset */
    unsigned long sh_size;          /* Section size in byte */
    unsigned long sh_info;          /* Additional section information */
};
```

### 4.2.11 최종 업그레이드 패키지 구조

최종 생성된 업그레이드 패키지의 구조는 표 4.5와 같다.

**표 4.5 최종 업그레이드 패키지 구조**

| magic | Release Note | Target Version | Original Version | Header Size | Firmware ImageSize | Dynup Image Size | Total Size |
|---|---|---|---|---|---|---|---|
| Firwmare Header | | Control Block | | Difference Block | | Extra Block | |
| Dynamic Header | | Text Section | | Rodata Section | | Data Section | |
| Relocation Section | | Symbol Section | | String Section | | Section Header | |

각 구성 요소를 설명하면 다음과 같다.

**magic**

업그레이드 패키지 종류

**Release Note**

업그레이드 패키지 릴리즈 노트

```
struct release_note {
    char *brief;
    char *detailed;
    int detail_size;
    int brief_size;
};
```

**Target Version**

업그레이드 패키지를 적용할 타겟 신 버전

```
struct upgrade_version {
    int major;
    int minor;
    int revision;
};
```

### Original Version

업그레이드 패키지가 적용될 타겟 구 버전

### Header Size

업그레이드 패키지 해더 사이즈

### Firmware Image Size

펌웨어 이미지 사이즈

### Dynamic Image Size

동적 업그레이드 이미지 사이즈

### Total Size

업그레이드 헤더. 펌웨어/동적 업그레이드 이미지를 합한 사이즈

## 4.2.12 업그레이드 생성기 API

| load_previous_repository | |
|---|---|
| Synopsis | #include “dynup.h”<br>int load_previous_repository(char *args, int from_tty) |
| Description | 구 버전 프로젝트의 모든 오브젝트를 전역 리스트에 삽입 |
| Arguments | args[0] : 프로젝트 URL |
| Return Value | 성공 시 0, 실패시 0 보다 작은 값을 반환 |

| load_previous_repository |
|---|

| Synopsis | #include “dynup.h”<br>int load_previous_repository(char *args, int from_tty) |
|---|---|
| Description | 신 버전 프로젝트의 모든 오브젝트를 전역 리스트에 삽입 |
| Arguments | args[0] : 프로젝트 URL |
| Return Value | 성공 시 0, 실패시 0 보다 작은 값을 반환 |

| **generte_fimware_delta** | |
|---|---|
| Synopsis | #include “delta.h”<br>int generte_fimware_delta(const char *src,<br>const char *dest,<br>const char *delta,<br>int opt) |
| Description | 구버전의 펌웨어 바이너리 이미지와 신 버전의 펌웨어 바이너리 이미지를 비교한 후 델타 파일 생성 |
| Arguments | src : 펌웨어 구 버전 이미지의 파일 경로<br>dest : 펌웨어 신 버전 이미지의 파일 경로<br>delta : 최종 생성할 펌웨어 업그레이드 파일 경로<br>opt : 펌웨어 델타 방법 firmware_option_enum_type 타입<br><br>typedef enum {<br>NO_DELTA = 0,<br>SIZE_OPTIMIZATION,<br>SPEED_OPTIMIZATION,<br>} firmware_option_enum_type; |
| Return Value | 성공 시 0, 실패시 0 보다 작은 값을 반환 |

| extract_upgrade_function_code | |
|---|---|
| Synopsis | #include “dynup.h”<br>int extract_upgrade_function_code(struct dynup *du); |
| Descri | 프로젝트의 모든 오브젝트로부터 변경된 함수를 찾아낸다. |

| ption | |
|---|---|
| Arguments | du : 변경된 함수를 저장하는 구조체 변수, 최종 동적 업그레이드 이미지 생성시 사용한다.<br><br>typedef struct dynup {<br>unsigned long dynup_header_size;<br>unsigned long sec_headers_size;<br>unsigned long text_sec_size;<br>unsigned long rodata_sec_size;<br>unsigned long data_sec_size;<br>unsigned long reloc_sec_size;<br>unsigned long symbol_sec_size;<br>unsigned long string_sec_size;<br>unsigned long dynaup_image_size;<br>unsigned long func_count;<br>unsigned long rodata_count;<br>unsigned long data_count;<br>char *dynup_image;<br>struct dynup_function_list *func_list;<br>} dynup_t; |
| Return Value | 성공시 0을 반환, 실패시 0 보다 작은 값 반환 |

| **generate_upgrade_function_code** | |
|---|---|
| Synopsis | #include “dynup.h”<br>int generate_upgrade_function_code(struct dynup *du) |
| Description | 동적 업그레이드 이미지 생성 함수.<br><br>동적 업그레이드 이미지 구조<br>\|\| Upgrade Header \| Text Section \| Rodate Section \| Data Section \| Relocation Section \| Symbol Section \| String Section \| Section Header \|\| |
| Arguments | du : 변경된 함수를 저장하는 구조체 변수, 최종 동적 업그레이드 이미지 생성시 사용한다.<br><br>typedef struct dynup { |

| | |
|---|---|
| | unsigned long dynup_header_size;<br>unsigned long sec_headers_size;<br>unsigned long text_sec_size;<br>unsigned long rodata_sec_size;<br>unsigned long data_sec_size;<br>unsigned long reloc_sec_size;<br>unsigned long symbol_sec_size;<br>unsigned long string_sec_size;<br>unsigned long dynaup_image_size;<br>unsigned long func_count;<br>unsigned long rodata_count;<br>unsigned long data_count;<br>char *dynup_image;<br>struct dynup_function_list *func_list;<br>} dynup_t; |
| Return Value | 성공시 0을 반환, 실패시 0 보다 작은 값 반환 |

| **make_upgrade_image** | |
|---|---|
| Synopsis | #include “dynup.h”<br>int make_upgrade_image(struct dynup *du, char *name) |
| Description | 최종 업그레이드 패키지를 생성한다.<br><br>최종 업그레이드 패키지<br><br>|| magic number | Release Note | Target Version | Original Version | Header Size | Dynamic Image Size | Firmware Image Size | Total Size | Dynmic Image | Firmware Image ||<br><br><br> |
| Arguments | du : 변경된 함수를 저장하는 구조체 변수, 최종 동적 업그레이드 이미지 생성시 사용한다.<br><br>typedef struct dynup {<br>unsigned long dynup_header_size;<br>unsigned long sec_headers_size;<br>unsigned long text_sec_size;<br>unsigned long rodata_sec_size; |

| | |
|---|---|
| | unsigned long data_sec_size;<br>unsigned long reloc_sec_size;<br>unsigned long symbol_sec_size;<br>unsigned long string_sec_size;<br>unsigned long dynaup_image_size;<br>unsigned long func_count;<br>unsigned long rodata_count;<br>unsigned long data_count;<br>char *dynup_image;<br>struct dynup_function_list *func_list;<br>} dynup_t; |
| Return Value | 성공시 0을 반환, 실패시 0 보다 작은 값 반환 |

## 4.3 업그레이드 서버 구현

### 4.3.1 OMA-DM 서버

OMA-DM 서버는 오픈소스 프로젝트인 Funambol Framework를 이용하여 작성하였다. Funambol Framework는 그림 4.10과 같이 Core, Database, Engine, Transport, Security, Protocol, Logging, Notification, Tools, Config를 제공하며 우리는 그 위에 검은색 블록으로 표현된 부분을 구현하였다. Web Sevice, Processor, Loging, Registration 모듈을 구현하였다.

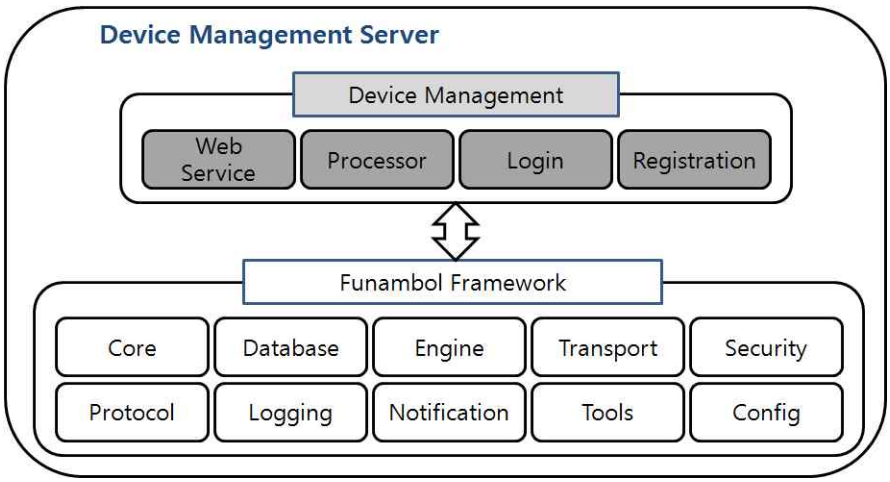



그림 4.10 OMA-DM 서버 구조

### 4.3.2 OMA-DM 서버

#### 가. Core

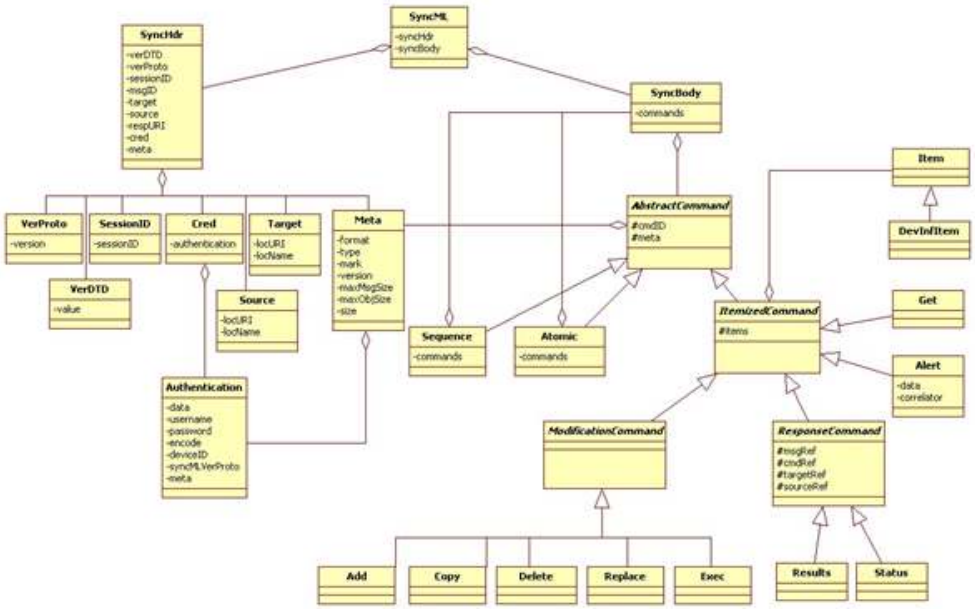



그림 4.11 OMA-DM 서버 Core 클래스 다이어그램

Funambol Framework의 Core 부분으로써 SyncML 메시지에 대한 부분이 어떻게 구성되는 지에 대한 클래스 다이어그램이다.

#### 나. Engine

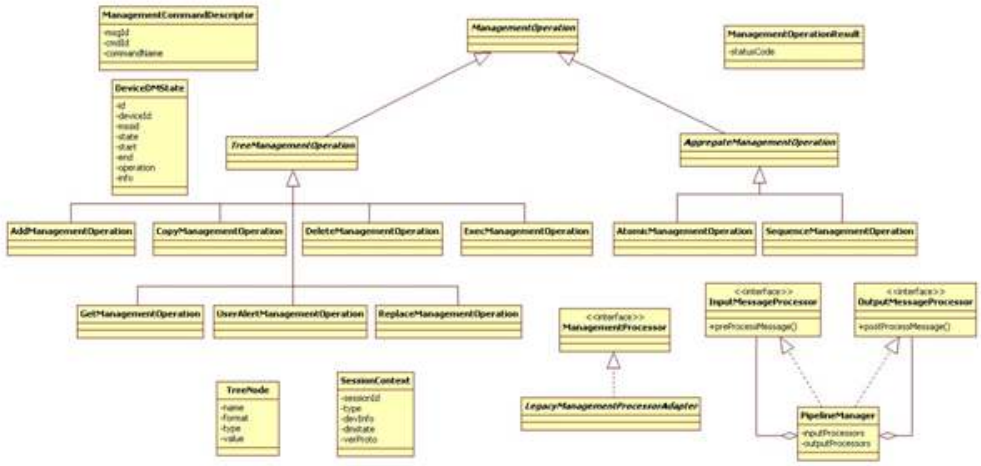



그림 4.12 OMA-DM 서버 Engine 클래스 다이어그램

Funambol Framework의 Engine 부분으로 Management Operation을 위한 클래스들의 구성에 대한 클래스 다이어그램이다.

## 다. Protocol

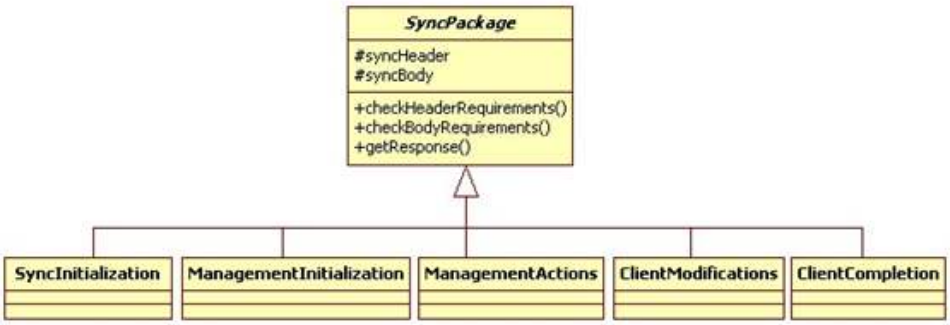



그림 4.13 OMA-DM 서버 Protocol 클래스 다이어그램

Funambol Framework의 Protocol 부분으로 SyncML 메시지를 생성하고 처리하기 위한 프로토콜의 클래스 다이어그램이다.

### 라. Security

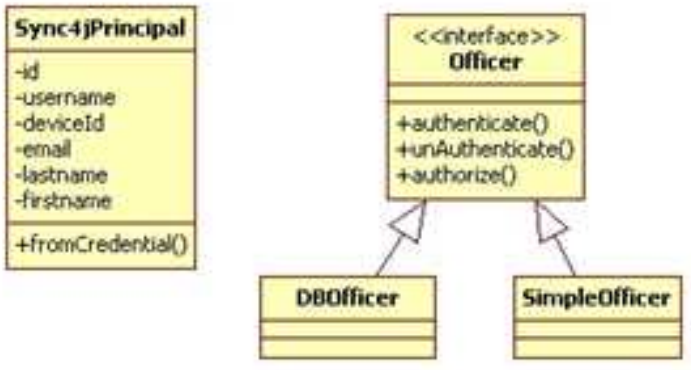



그림 4.14 OMA-DM 서버 Security 클래스 다이어그램

Funambol Framework의 Security에 대한 부분으로 보안을 위해 필요한 정보와 그것에 대한 보안을 관리하기 위한 Officer로 구성되는 클래스 다이어그램이다.

### 마. Transport

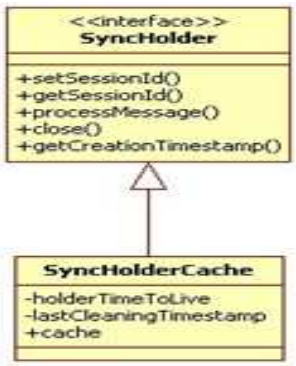



그림 4.15 Transport 클래스 다이어그램

Funambol Framework의 Transport에 관련된 부분으로 HTTP 연결 정보와 시간에 대한 정보를 관리하기 위한 부분의 클래스 다이어그램이다.

## 4.4 타겟 시스템의 구현

### 4.4.1 타겟 시스템 시퀀스 다이어그램

업그레이드 이미지를 적용하기 위한 과정을 그림 4.16은 보여준다.

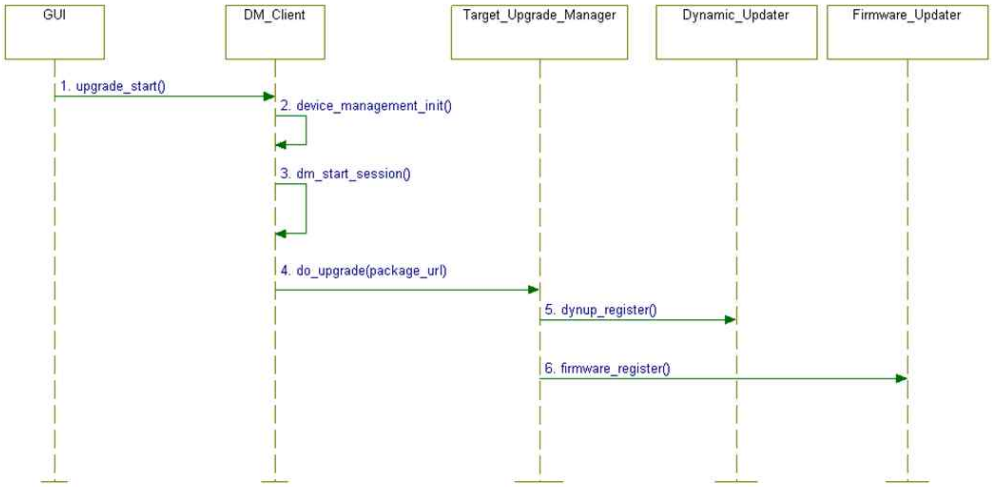



그림 4.16 타겟 시스템 시퀀스 다이어그램

수행 흐름을 보면 업그레이드 GUI를 통해 사용자의 입력을 받게 되고 DM Client는 OMA-DM 프로토콜을 사용하여 신 버전의 업그레이드 패키지 URL을 얻는다. 업그레이드 패키지 URL을 이용하여 타겟 업그레이드 관리자는 동적 업그레이드를 수행하고 동적 업그레이드 수행을 마치면 펌웨어 업그레이드를 수행하는 흐름으로 이루어진다.

### 4.4.2 타겟 시스템 상태 다이어그램

타겟 시스템의 업그레이드를 수행할 때는 다음과 같은 상태을 가진다. 본 논문의 상태 다이어그램는 FUMO의 표준에 맞추어 설계를 하였으며 그림 4.17과 같다.

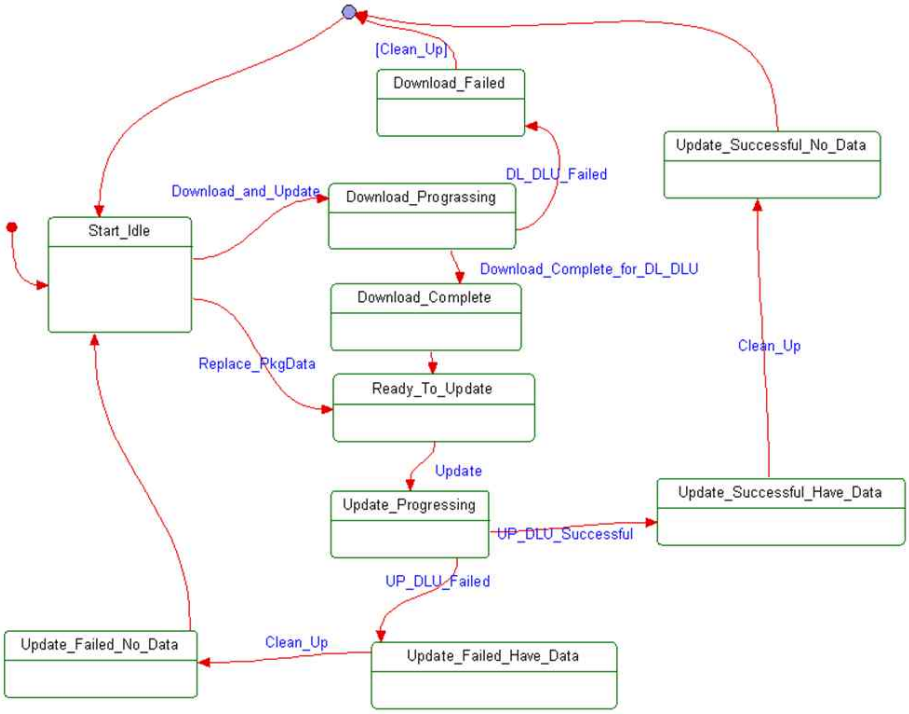



그림 4.17 타겟 시스템 상태 다이어그램

각 상태에 대한 설명은 표 4.6과 같다.

표 4.6 타겟 시스템 상태 표

| 상태명 | 설명 | 다음 상태 |
|---|---|---|
| start/idle | 업데이트를 기다리는 상태 | download progressing<br>ready to update |
| download progressing | 다운로드를 진행하는 상태 | download complete<br>download failed |
| download complete | 다운로드가 완료된 상태 | ready to update |
| ready to update | 업데이트 수행을 기다리는 상태 | update progressing |
| update progressing | 업데이트를 수행하는 상태 | update failed have data<br>update successful have data |
| update successful have data | 업데이트를 성공적으로 수행한 상태<br>사용한 데이터를 아직 가지고 있다. | update successful no data |
| update successful no data | 업데이트를 성공적으로 수행한 상태<br>사용한 데이터를 지운 상태 | start/idle |
| download failed | 다운로드 실패 상태 | start/idle |

| | | |
|---|---|---|
| update failed<br>have data | 업데이트 실패 상태 사용한 데이터를<br>아직 가지고 있는 상태 | update successful no data |
| update successful<br>no data | 업데이트 실패 상태<br>사용한 데이터를 지운 상태 | start/idle |

FUMO 상태 챠트는 다음과 같이 구현하였다. 각 상태에 따른 이벤트에 대한 다음 상태와 핸들러로 이루어진다.

```
struct fumo_state_machine fumo_state_table[NUM_STATE][NUM_INPUT] = {

    {    /* State INIT */
        { DOWNLOAD_PRGRESSING,    download_progressing_handler },
        { READY_TO_UPDATE,    ready_to_update_handler},
        { INIT, nop_handler    },
        { INIT, nop_handler    },
        { INIT, nop_handler    },
    },

    {    /* State DOWNLOAD_FAILED */
        { DOWNLOAD_FAILED,    nop_handler },
        { DOWNLOAD_FAILED,    nop_handler},
        { DOWNLOAD_FAILED,    nop_handler },
        { INIT, init_handler    },
        { DOWNLOAD_FAILED,    nop_handler },
    },

    {    /* State DOWNLOAD_PRGRESSING */
        { DOWNLOAD_PRGRESSING,    nop_handler },
        { DOWNLOAD_PRGRESSING,    nop_handler},
        { DOWNLOAD_PRGRESSING,    nop_handler },
        { DOWNLOAD_FAILED,    download_failed_handler },
        { DOWNLOAD_COMPLETE,    download_complete_handler },
    },

    {    /* State DOWNLOAD_COMPLETE */
        { DOWNLOAD_COMPLETE,    nop_handler },
        { DOWNLOAD_COMPLETE,    nop_handler},
        { DOWNLOAD_COMPLETE,    nop_handler },
        { DOWNLOAD_COMPLETE,    nop_handler },
        { READY_TO_UPDATE,    ready_to_update_handler },
    },

    {    /* State READY_TO_UPDATE */
        { READY_TO_UPDATE,    nop_handler },
        { INIT,    init_handler},
        { UPDATE_PROGRESSING,    update_progressing_handler },
        { READY_TO_UPDATE, nop_handler    },
        { READY_TO_UPDATE,    nop_handler },
    },

    {    /* State UPDATE_PROGRESSING */
        { UPDATE_PROGRESSING,    nop_handler },
        { UPDATE_PROGRESSING,    nop_handler},
```

```
        { UPDATE_PROGRESSING,     nop_handler },
        { UPDATE_FAILED_HAVE_DATA, update_failed_have_data_handler },
        {    UPDATE_SUCCESSFUL_HAVE_DATA, update_successful_have_data_handler },
    },

    {       /* State UPDATE_FAILED_HAVE_DATA */
        { UPDATE_FAILED_HAVE_DATA,     nop_handler },
        {    UPDATE_FAILED_HAVE_DATA, nop_handler},
        {    UPDATE_FAILED_HAVE_DATA, nop_handler },
        { UPDATE_FAILED_NO_DATA,     update_failed_no_data_handler },
        {    UPDATE_FAILED_HAVE_DATA, nop_handler },
    },

    {       /* State UPDATE_FAILED_NO_DATA */
        { UPDATE_FAILED_NO_DATA,     nop_handler },
        { UPDATE_FAILED_NO_DATA,     nop_handler},
        { UPDATE_FAILED_NO_DATA,     nop_handler },
        { INIT, init_handler     },
        { UPDATE_FAILED_NO_DATA,     nop_handler },
    },

    {       /* State UPDATE_SUCCESSFUL_HAVE_DATA */
        {    UPDATE_SUCCESSFUL_HAVE_DATA, nop_handler },
        {    UPDATE_SUCCESSFUL_HAVE_DATA, nop_handler},
        {    UPDATE_SUCCESSFUL_HAVE_DATA, nop_handler },
        {    UPDATE_SUCCESSFUL_HAVE_DATA, nop_handler },
        {    UPDATE_SUCCESSFUL_NO_DATA, update_successful_no_data_handler },
    },

    {       /* State UPDATE_SUCCESSFUL_NO_DATA */
        {    UPDATE_SUCCESSFUL_NO_DATA, nop_handler },
        {    UPDATE_SUCCESSFUL_NO_DATA, nop_handler},
        {    UPDATE_SUCCESSFUL_NO_DATA, nop_handler },
        {    UPDATE_SUCCESSFUL_NO_DATA, nop_handler },
        { INIT, init_handler     },
    },
};
```

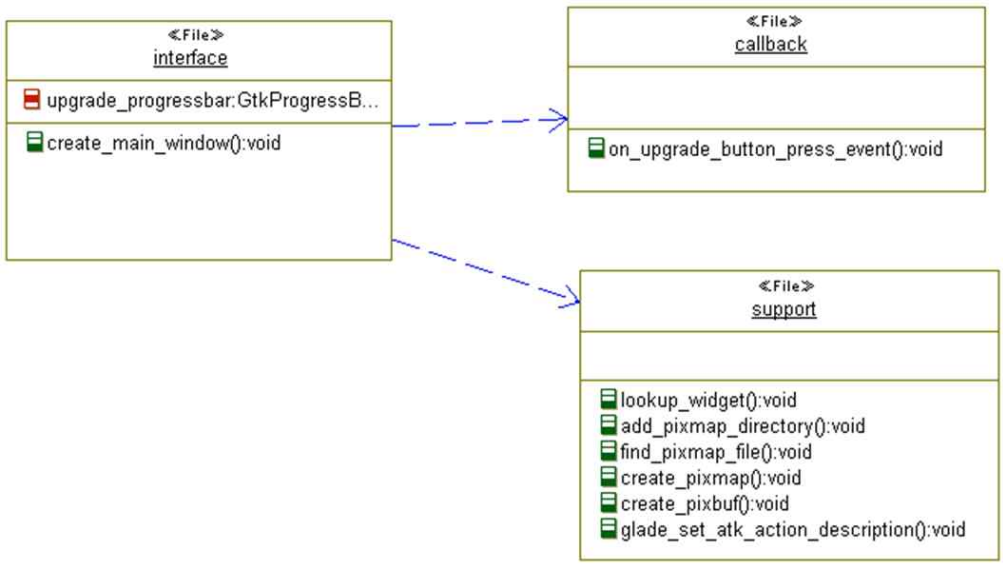
실제 호출은 다음과 같이 구현된다. 이벤트에 대한 상태 머신 테이블의 핸들러를 호출하고 다음 상태로 변경하는 코드이다.

```
wait_event(&event);
/* 1: Generate Input (Event) */
input = event.type;
/* 2: Do action */
fumo_state_table[fumo_state][input].action(event.data);
/* 3: Set Next State */
fumo_state = fumo_state_table[fumo_state][input].next_state;
```

### 4.4.3 타겟 시스템 GUI

타겟 시스템의 타겟 장비의 GTK Library를 사용하여 구현 하였으며 UI는 업그레이드 수행 버튼과 업그레이드 진행바를 구현 하였다.

#### 가. File Diagram




interface.c
  GTK 라이브러리를 사용하여 윈도우 위젯을 생성하는 파일
callback.c
 버튼든 GTK 이벤트에 따른 핸들러들에 대한 파일
support.c
  GTK에서 추가적인 기능을 위한 유틸 함수에 대한 파일

### 4.4.4 DM 클라이언트

DM Client는 OMA-DM 1.2 표준 프로토콜을 사용하여 업그레이드 서버로부터 통신을 하며 인증과 현재 단말기에 필요한 업그레이드 패키지 URL을 얻어오는 일을 한다.

본 논문에서 구현된 DM Client는 그림 4.18와 같은 구조로 구현되었다.

검은색 블록은 본 논문에서 구현한 부분이고 검은색이 아닌 블록들은 SyncML C Reference Toolkit Library(http://sourceforge.net/projects/syncml-ctoolkit)에서 제공되는 기능이다.

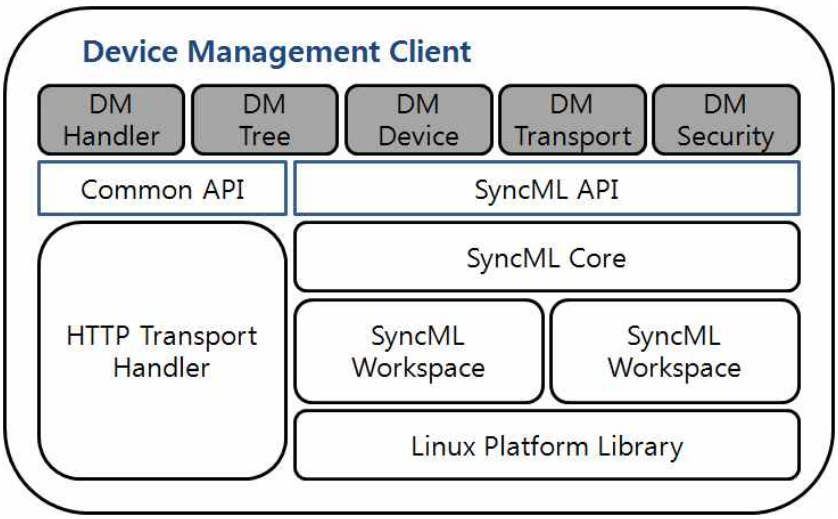
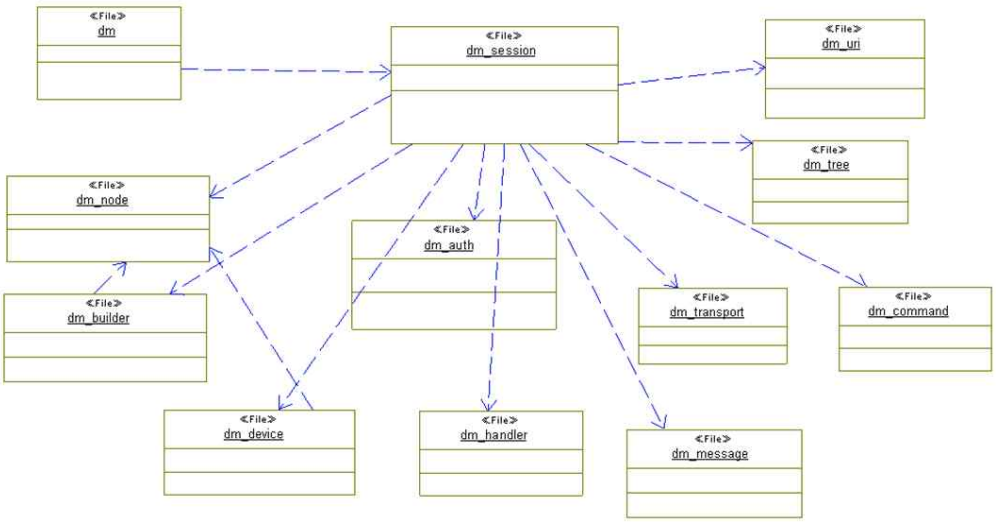



그림 4.18 DM 클라이언트 구조

## 가. File Diagram



dm_auth.c
  MD5를 이용해 암호화 / 복호화 하는 기능에 대한 파일
dm_builder.c
  SyncML 메시지를 생성하는 기능에 대한 파일
dm_command.c
  SyncML 명령어를 관리하는 기능에 대한 파일
dm_device.c
  단말기의 정보를 관리하는 기능에 대한 파일
dm_error.c
  에러메시지를 관리하는 기능에 대한 파일
dm_handler.c
  SyncML 메시지의 각 명령을 분석하여 처리하는 기능에 대한 파일
dm_message.c
  여러 메시지를 관리하는 기능에 대한 파일
dm_node.c
  단말기의 정보를 노드로 관리하는 기능에 대한 파일
dm_session.c
  접속 정보를 관리하는 기능에 대한 파일
dm_transport.c
 서버와 연결, 메시지 전송의 기능에 대한 파일
dm_tree.c
 단말기 정보가 담긴 노드를 관리하기 위한 트리에 대한 파일
dm_uri.c
 서버의 정보를 관리하는 기능에 대한 파일
dm_util.c
  DM 유틸 함수에 대한 파일
dm_verify.c
 서버로 부터 받은 메시지가 DM 규격에 맞는지를 확인하는 기능에 대한 파일

### 4.4.5 펌웨어 델타 적용

본 프로토타입 시스템의 커널은 NOR 플레쉬 메모리에 저장되어 있으며 사용자 영역에서 NOR 플레쉬 메모리를 직접 접근하기 위해서 리눅스의 MTD를 사용하여 구현하였다. 본 논문에서 구현된 펌웨어 델타 적용을 리눅스의 커널 이미지를 대상으로 구현하였으나 리눅스 커널 이미지는 압축된 이미지이다. 그래서 압축된 파일에 대한 델타 생성은 못하는 관계로 델타파일이 아닌 새로운 버전 커널 이미지를 받아서 적용하는 방법으로 구현하였다.

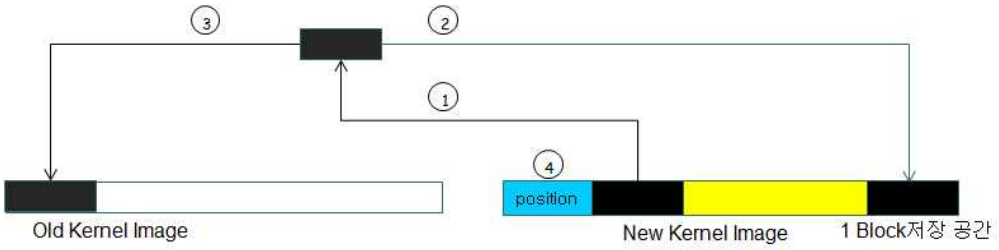
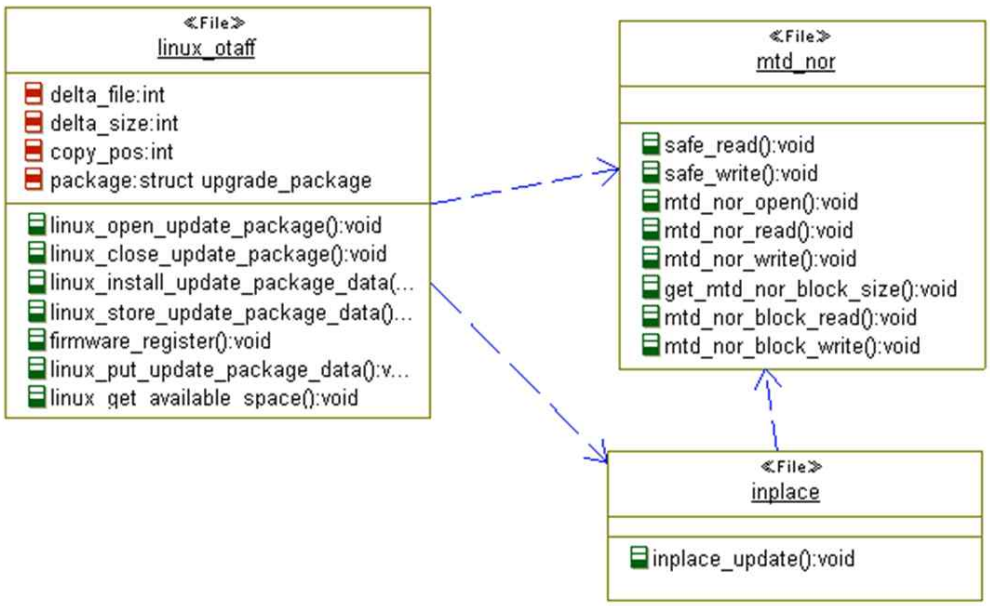



그림 4.19 타겟 시스템 펌웨어 적용 순서

그림 4.19는 실제 구현된 펌웨어 적용 순서이다. 먼저 새로운 커널 이미지의 위치에서 1 블록을 읽은 후 임시 저장 공간에 저장한다. 다음으로 새로운 커널 이미지가 위치한 곳에 1 블록의 데이터를 쓴다. 마지막으로 현재 적용한 플래쉬 블록 위치를 저장하는 순서로 펌웨어 업그레이드를 수행하였다.

### 가. 파일 다이어그램



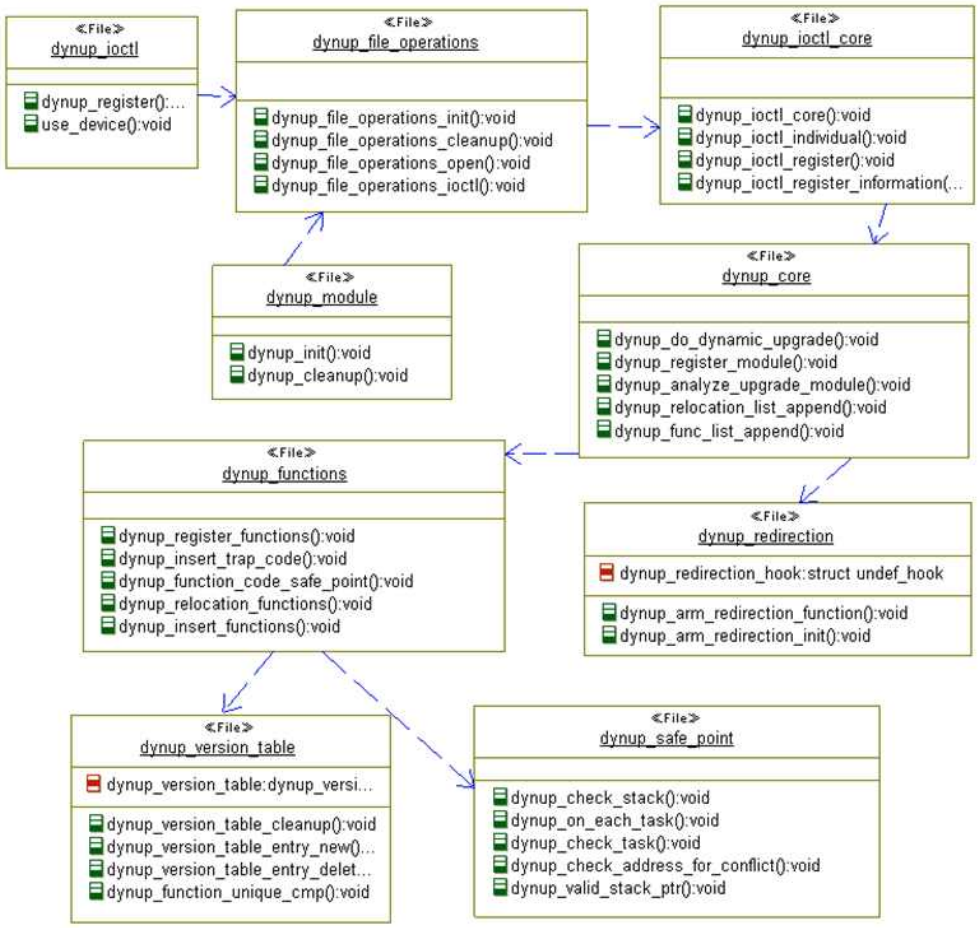

| linux_otaff.c |
|---|

펌웨어를 업그레이드 하기위한 함수에 대한 파일
mtd_nor.c
리눅스 mtd를 사용하여 nor 메모리의 읽고 쓰기 처리하기 위한 함수에 대한 파일
inplace.c
inplace 델타를 적용하기 위한 함수에 대한 파일

### 4.4.6 동적 업그레이드 델타 적용

동적 업그레이드의 델타를 적용하기 위해서 동적 업그레이드 적용 부분은 커널 메모리 영역을 수정해야 함으로 리눅스 커널 모듈로 작성하였다. 사용자가 동적 업그레이드 이미지를 커널 모듈 함수로 전달하여 동적 업그레이드를 수행 하였다. 이러한 동적 업그레이드 델타 적용은 동적 업그레이드 설계에서 설명한 그림 3.9와 같이 수행하였다.

#### 가. 파일 다이어그램




dynup_ioctl.c
 유저 영역에서 커널영역으로 진입하기 위한 파일
dynup_file_operations.c
 유저의 요청에 따라 커널영역에서 수행하는 핸들러에 대한 파일
dynup_ioctl_core.c
 유저의 ioctl의 인자에 따라 호출되는 핸들러에 대한 파일
dynup_module.c
 리눅스 모듈 스타일의 초기화와 종료 코드에 대한 파일
dynup_core.c
 동적 업그레이드를 수행하는 핵심적인 일을 하는 파일
dynup_function.c
 함수 코드 단위로 동적 업그레이드를 수행하기 위한 파일
dynup_redirection.c
 구 버전 함수에서 신 버전 함수로 분기하기 위한 트랩 핸들러 파일

dynup_version_table.c
  해쉬 테이블에 대한 파일
dynup_safe_point.c
  동적 업그레이드의 safe point를 위한 파일

### 4.4.7 타겟 시스템 API

| otaff_get_dl_prompt_text | |
|---|---|
| Synopsis | #include “otaff.h”<br>char * otaff_get_dl_prompt_text(const char *name) |
| Description | 다운로드 전에 사용자에게 메시지를 출력하는 함수 |
| Arguments | name : FUMO update 패키지 이름 |
| Return Value | 출력한 메시지 |

| otaff_get_dl_prompt_counter | |
|---|---|
| Synopsis | #include “otaff.h”<br>unsigned char otaff_get_dl_prompt_counter(const char *name); |
| Description | 사용자에게 표시할 다운로드를 시간을 얻는다. |
| Arguments | name : FUMO update 패키지 이름 |
| Return Value | 다운로드 시간 0~255 값으로 설정 |

| otaff_get_dl_prompt_timeout | |
|---|---|
| Synopsis | #include “otaff.h”<br>unsigned int otaff_get_dl_prompt_timeout(const char * name); |
| Description | 다운로드 최대 시간을 얻는다. 최대 시간에 초과하면 다운로드 실패처리 |
| Arguments | name : FUMO update 패키지 이름 |
| Return Value | 다운로드 최대 시간 |

| otaff_get_update_type | |
|---|---|
| Synop | #include “otaff.h” |

| | |
|---|---|
| sis | unsigned char otaff_get_update_type(const char *name); |
| Description | 업데이트 방식을 얻는다.<br>Mandatory 0x01<br>Optional 0x02<br>NoPrompt 0x03 |
| Arguments | name : FUMO update 패키지 이름 |
| Return Value | 업데이트 타입 |

| **otaff_init_download _client** | |
|---|---|
| Synopsis | #include “otaff.h”<br>int otaff_init_download_client(const char *name, const char *url); |
| Description | 다운로드 초기화와 실제 업데이트 패키지를 다운 받는다. |
| Arguments | name : 패키지 이름<br>url : 패키지 주소 |
| Return Value | 성공 시 0 실패 시 0 보다 작은 값 반환 |

| **otaff_get_dl_status_code** | |
|---|---|
| Synopsis | #include “otaff.h”<br>unsigned int otaff_get_dl_status_code(const char *name); |
| Description | 현재 다운로드 상태를 반환한다. |
| Arguments | name : 패키지 이름 |
| Return Value | 업데이트 상태,. OMA FUMO 표준에 해당하는 상태 값 |

| **otaff_init_update_client** | |
|---|---|
| Synopsis | #include “otaff.h”<br>int otaff_init_update_client(const char *name); |
| Description | 업데이트를 위한 초기화를 수행한다. |
| Arguments | name : 패키지 이름 |
| Return Value | 성공 시 0 실패 시 0 보다 작은 값 반환 |

| **otaff_get_update_status_code** | |
|---|---|
| Synops | #include “otaff.h” |

| is | double otaff_get_update_status_code(const char *name); |
|---|---|
| Description | 현재 업데이트 상태를 얻는다. |
| Arguments | name : 패키지 이름 |
| Return Value | 업데이트 상태 |

| **otaff_open_update_package** | |
|---|---|
| Synopsis | #include “otaff.h”<br>int otaff_open_update_package(const char *name, unsigned int length); |
| Description | 업데이트를 적용하기 위한 초기화를 수행한다. |
| Arguments | name : 패키지 이름<br>length : 패키지 사이즈 |
| Return Value | 성공 시 0 실패 시 0 보다 작은 값 반환 |

| **otaff_close_update_package** | |
|---|---|
| Synopsis | #include “otaff.h”<br>void otaff_close_update_package(const char *name); |
| Description | 업데이트 적용을 마친 후 메모리 해제등 마무리 작업을 한다. |
| Arguments | name : 패키지 이름 |
| Return Value | 없음 |

| **otaff_put_update_package_data** | |
|---|---|
| Synopsis | #include “otaff.h”<br>int otaff_put_update_package_data(const char *name, void *data, unsigned int offset, unsigned int length); |
| Description | 업데이트를 적용한다. |
| Arguments | name : 패키지 이름<br>data : 패키지 데이터<br>offset : 적용할 Flash 메모리상의 패키지 위치<br>length : 패키지 길이 |
| Return Value | 성공 시 0 실패 시 0 보다 작은 값 반환 |

| otaff_get_available_space | |
|---|---|
| Synopsis | #include “otaff.h”<br>unsigned int otaff_get_available_space(const char *name); |
| Description | 사용가능한 저장공간을 반환한다. |
| Arguments | name : 패키지 이름 |
| Return Value | 남은 공간 사이즈 |

## 4.5 실행 및 평가

본 논문에서 구현한 동적 업그레이드를 프로토타입 시스템을 실제 사용하고 평가하기 위해 커널의 do_fork() 함수와 vfat_mkdir() 함수를 대상으로 업그레이드를 시도해보았다. 우선 do_fork() 함수는 코드 수정과 추가된 함수가 문제없이 동작함을 보이기 위해 수정하였으며 static으로 선언된 함수가 업그레이드 가능한 것을 보이기 위해 리눅스 커널의 static 함수인 vfat_mkdir()함수를 대상으로 업그레이드를 수행하였다.

### 4.5.1 업그레이드 이미지 생성

우선 업그레이드 이미지 생성은 표 4.7과 같이 리눅스 커널의 do_fork 코드를 수정하였다. 진한 글씨체가 추가된 코드이다.

**표 4.7 do_fork 함수 수정 코드**

```
long do_fork(unsigned long clone_flags,
          unsigned long stack_start,
          struct pt_regs *regs,
          unsigned long stack_size,
          int __user *parent_tidptr,
          int __user *child_tidptr)
{
  struct task_struct *p;
```

```
int trace = 0;
long pid = alloc_pidmap();

if (pid < 0)
  return -EAGAIN;
if (unlikely(current->ptrace)) {
  trace = fork_traceflag (clone_flags);
  if (trace)
    clone_flags |= CLONE_PTRACE;
}

p = copy_process(clone_flags, stack_start, regs, stack_size, parent_tidptr, child_tidptr, pid);
/*
 * Do this prior waking up the new thread - the thread pointer
 * might get invalid after that point, if the thread exits quickly.
 */
if (!IS_ERR(p)) {
  struct completion vfork;

  if (clone_flags & CLONE_VFORK) {
    p->vfork_done = &vfork;
    init_completion(&vfork);
  }

  if ((p->ptrace & PT_PTRACED) || (clone_flags & CLONE_STOPPED)) {
    /*
     * We'll start up with an immediate SIGSTOP.
     */
    sigaddset(&p->pending.signal, SIGSTOP);
    set_tsk_thread_flag(p, TIF_SIGPENDING);
  }

  if (!(clone_flags & CLONE_STOPPED))
    wake_up_new_task(p, clone_flags);
  else
    p->state = TASK_STOPPED;

  if (unlikely (trace)) {
    current->ptrace_message = pid;
    ptrace_notify ((trace << 8) | SIGTRAP);
  }

  if (clone_flags & CLONE_VFORK) {
    wait_for_completion(&vfork);
    if (unlikely (current->ptrace & PT_TRACE_VFORK_DONE))
```

```
            ptrace_notify ((PTRACE_EVENT_VFORK_DONE << 8) | SIGTRAP);
        }
    } else {
        free_pidmap(pid);
        pid = PTR_ERR(p);
    }

    printk("do_fork update\n");
    static_function_b();

    return pid;
}
```

추가된 함수를 테스트해보기 위하여 표 4.8과 같은 코드를 추가하였으며 do_fork 함수에서 호출되도록 수정하였다.

**표 4.8 업그레이드 추가 코드**

```
static void static_function_c()
{
    printk("static fork c\n");
}

static void static_function_b()
{
    printk("b()\n");
    static_function_c();
    printk("static fork b\n");
}
```

리눅스의 static 함수인 vfat_mkdir 함수를 표 4.9와 같이 수정하였다.

**표 4.9 static 함수 수정 코드**

```
static int vfat_mkdir(struct inode *dir, struct dentry *dentry, int mode)
{
    struct super_block *sb = dir->i_sb;
    struct inode *inode;
    struct fat_slot_info sinfo;
    struct timespec ts;
    int err, cluster;

    lock_kernel();
```

```
    ts = CURRENT_TIME_SEC;
    cluster = fat_alloc_new_dir(dir, &ts);
    if (cluster < 0) {
        err = cluster;
        goto out;
    }
    err = vfat_add_entry(dir, &dentry->d_name, 1, cluster, &ts, &sinfo);
    if (err)
        goto out_free;
    dir->i_version++;
    dir->i_nlink++;

    inode = fat_build_inode(sb, sinfo.de, sinfo.i_pos);
    brelse(sinfo.bh);
    if (IS_ERR(inode)) {
        err = PTR_ERR(inode);
        /* the directory was completed, just return a error */
        goto out;
    }
    inode->i_version++;
    inode->i_nlink = 2;
    inode->i_mtime = inode->i_atime = inode->i_ctime = ts;
    /* timestamp is already written, so mark_inode_dirty() is unneeded. */

    dentry->d_time = dentry->d_parent->d_inode->i_version;
    d_instantiate(dentry, inode);

    unlock_kernel();
    printk("vfat_update function\n");
    return 0;

out_free:
    fat_free_clusters(dir, cluster);
out:
    unlock_kernel();
    return err;
}
```

위와 같이 코드를 수정한 후 컴파일을 수행 후 4.1.2 절에서 설명한 업그레이드 GUI를 통하여 업그레이드 이미지를 생성한 후 업그레이드 적용 테스트를 해보았다.

### 4.5.2 업그레이드 적용

#### 가. OMA-DM 서버 설정

그림 4.20은 업그레이드 서버 설정에 대한 그림이다. 본 논문의 프로토타입 시스템은 버전관리자가 구현되지 않았으므로 버전 설정 화면이 없이 바로 업그레이드를 수행하는 서버 화면이다.

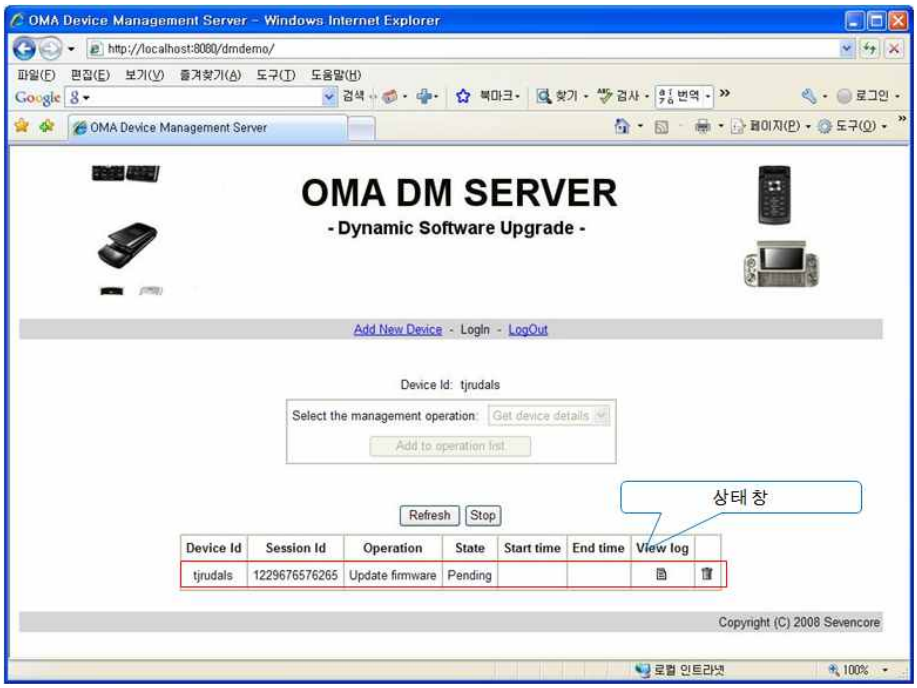



그림 4.20 OMA-DM 서버 설정

#### 나. 타겟 업그레이드 실행

업그레이드 실행은 그림 4.21과 같은 스마트폰 참조보드의 GUI를 통하여 사용자의 업그레이드 수행 버튼을 통하여 수행하도록 하였고 업그레이드 수행 버튼을 누르면 업그레이드를 수행 한다.


그림 4.21 타겟 시스템 GUI

### 다. 업그레이드 다운로드

업그레이드 패키지 다운로드는 OMA-DM 프로토콜을 사용하여 다운로드 되며 업그레이드 서버로부터 다운로드 서버의 업그레이드 패키지 URL을 얻는다. 그림 4.22는 업그레이드 패키지 URL을 얻는 타겟 머신 상에서의 OMA-DM 로그 메시지이다.

```
receive data : <?xml version="1.0" encoding="UTF-8"?>
<SyncML>
<SyncHdr>
<VerDTD>1.1</VerDTD>
<VerProto>DM/1.1</VerProto>
<SessionID>1</SessionID>
<MsgID>3</MsgID>
<Target>
<LocURI>tjrudals</LocURI>
</Target>
<Source>
<LocURI>http://192.168.0.13:8080/funambol/dm?sid=W0JANTcxY2M0LTEyMjk2NzcxOTQ2OD</LocURI>
</Source>
<RespURI>http://localhost:8080/funambol/dm?sid=W0JANTcxY2M0LTEyMjk2NzcxOTQ2OD</RespURI>
</SyncHdr>
<SyncBody>
<Status>
<CmdID>1</CmdID>
<MsgRef>3</MsgRef>
<CmdRef>0</CmdRef>
<Cmd>SyncHdr</Cmd>
<TargetRef>http://192.168.0.13:8080/funambol/dm?sid=W0JANTcxY2M0LTEyMjk2NzcxOTQ2OD</TargetRef>
<SourceRef>tjrudals</SourceRef>
<Data>200</Data>
</Status>
<Replace>
<CmdID>2</CmdID>
<Item>
<Target>
<LocURI>./DevDetail/FwV</LocURI>
</Target>
<Meta>
<Format xmlns='syncml:metinf'>chr</Format>
<Type xmlns='syncml:metinf'>text/plain</Type>
</Meta>
<Data>1.1_1.1</Data>
</Item>
</Replace>
<Replace>
<CmdID>3</CmdID>
<Item>
<Target>
<LocURI>./FwPkg/Sevencore/PXA270/DownloadAndUpdate/PkgURL</LocURI>
</Target>
<Meta>
<Format xmlns='syncml:metinf'>chr</Format>
<Type xmlns='syncml:metinf'>text/plain</Type>
</Meta>
<Data>http://192.168.0.13:8080/dynup</Data>
</Item>
</Replace>
<Final></Final>
</SyncBody>
```

Upgrade Package URL

그림 4.22 OMA-DM 로그 메시지

타겟 머신은 업그레이드 패키지의 URL을 가지고 wget 명령어로 다운을 받는다. 그림 4.23은 업그레이드 패키지의 URL를 통하여 업그레이드 패키지를 다운받는 그림이다.

```
receive data : <?xml version="1.0" encoding="UTF-8"?>
<SyncML>
<SyncHdr>
<VerDTD>1.1</VerDTD>
<VerProto>DM/1.1</VerProto>
<SessionID>1</SessionID>
<MsgID>4</MsgID>
<Target>
<LocURI>tjrudals</LocURI>
</Target>
<Source>
<LocURI>http://192.168.0.13:8080/funambol/dm?sid=W0JANTcxY2M0LTEyMjk2NzcxOTQ20D</LocURI>
</Source>
<RespURI>http://localhost:8080/funambol/dm?sid=W0JANTcxY2M0LTEyMjk2NzcxOTQ20D</RespURI>
</SyncHdr>
<SyncBody>
<Status>
<CmdID>1</CmdID>
<MsgRef>4</MsgRef>
<CmdRef>0</CmdRef>
<Cmd>SyncHdr</Cmd>
<TargetRef>http://192.168.0.13:8080/funambol/dm?sid=W0JANTcxY2M0LTEyMjk2NzcxOTQ20D</TargetRef>
<SourceRef>tjrudals</SourceRef>
<Data>200</Data>
</Status>
<Exec>
<CmdID>2</CmdID>
<Item>
<Target>
<LocURI>./FwPkg/Sevencore/PXA270/DownloadAndUpdate</LocURI>
</Target>
</Item>
</Exec>
<Final></Final>
</SyncBody>
</SyncML>

Command ID : 1, Message Ref.ID : 4, Command Ref.ID : 0, Command : SyncHdr
download_progressing_handler
wget http://192.168.0.13:8080/dynup/dynup.pkg
Connecting to 192.168.0.13[192.168.0.13]:8080
dynup.pkg             78% |**********************       |  1324 KB    00:04 ETA
```

Upgrade Package Download

그림 4.23 업그레이드 패키지 다운로드 로그

### 라. 업그레이드 서버

업그레이드 서버는 업그레이드 진행을 마치면 수행완료 상태로 수정 되며 그림 4.24는 타겟 장비의 업그레이드 수행 완료 상태를 보여준다.

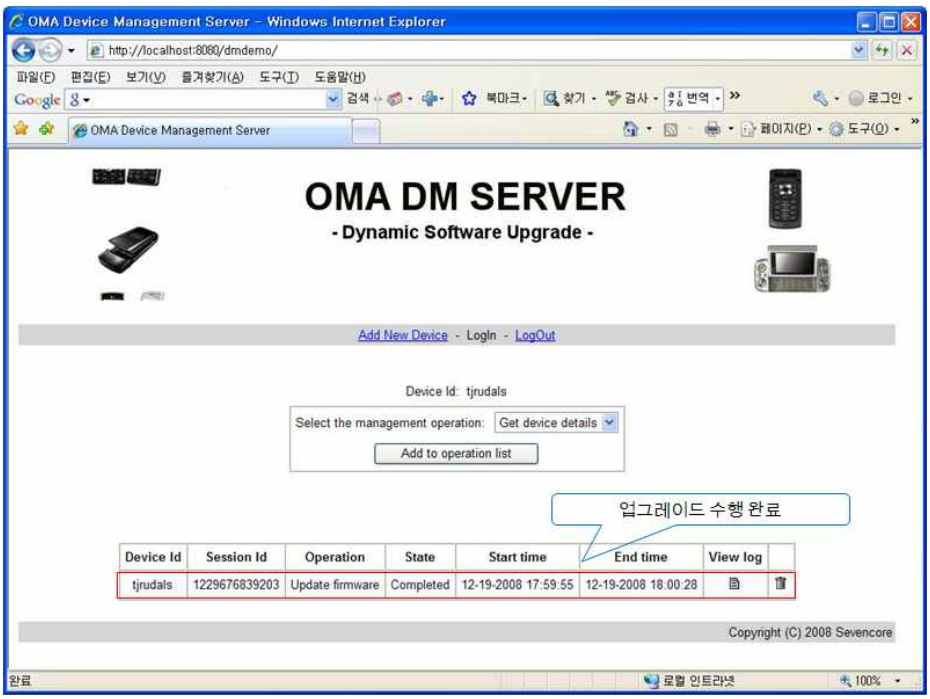



그림 4.24 업그레이드 서버 수행 완료 상태

### 마. 업그레이드 적용 화면

타겟 장비의 업그레이드 적용을 하여 리눅스의 ls 명령어를 사용하여 do_fork() 함수가 정상적으로 업그레이드를 수행되었는지 확인하였다. 그림 4.25는 업그레이드 수행 전 ls 명령어에 대한 콘솔 메시지이고 그림 4.26는 업그레이드 수행 후 ls 명령어에 대한 콘솔 메시지이다. do_fork() 함수에서 출력한 "do_fork update" 메시지와 추가된 함수에서 출력한 "static fork c", "b()", "static fork b"의 메시지가 정상적으로 찍히는 것을 볼 수 있다.

```
~ # cd /
/ # ls
bin         etc         minicom.log  proc        sys         var
boot        home        mnt          root        tmp
dev         lib         opt          sbin        usr
/ #
```

그림 4.25 업그레이드 수행 전 ls 콘솔 메시지

```
~ # cd /
/ # ls
do_fork update
static fork c
b()
static fork b
bin          etc          minicom.log  proc         sys          var
boot         home         mnt          root         tmp
dev          lib          opt          sbin         usr
/ #
```

그림 4.26 do_fork 업그레이드 수행 후 ls 콘솔 메시지

그림 4.27은 static 함수인 vfat_mkdir() 함수에 대한 업그레이드 콘솔 메시지이다. vfat_mkdir() 함수가 호출되기 위해 SD카드를 마운트하여 mkdir 명령어를 사용하였으며 “vfat_update function” 메시지가 정상적으로 출력되는 것을 볼 수 있다.

```
/ # mount -t vfat /dev/mmcblk0 /mnt/card/
do_fork update
static fork c
b()
static fork b
/ # cd /mnt/card/
/mnt/card # mkdir upgrade
do_fork update
static fork c
b()
static fork b
vfat_update function
/mnt/card #
```

그림 4.27 vfat_mkdir 업그레이드 수행 후 콘솔 메시지

### 4.5.3 평가

do_fork() 함수와 vfat_mkdir() 함수 이외에도 본 논문에서는 구현된 시

스템은 어셈블리어로 작성된 함수와 커널의 schedule() 함수를 제외하고는 거의 대부분의 함수가 업그레이드 가능하다는 것을 확인하였다. schedule() 함수는 리눅스에서 일반적인 .text 섹션이 아닌 .text.schedule 섹션으로 처리하여 오브젝트 비교를 수행할 수 없었으며 차후 업그레이드 가능하도록 수정할 예정이다. 어셈블리어 구현된 함수를 업그레이드하는 것은 Ksplice에서는 제공하나 어셈블리어 구현된 함수를 업그레이드하는 것은 거의 일어나지 않는다고 판단하여 구현하지 않았다.

## 4.6 구현상의 문제점과 해결 방안

### 4.6.1 함수 단위 코드영역 비교

함수 단위 동적 업그레이드를 위해서는 수정된 함수를 찾아내야 한다. 이러한 수정된 함수를 찾기 위해 함수 단위 코드영역을 비교해야 하는데 본 논문에서는 수정된 함수를 찾아내기 위해 심볼 재배치가 안 끝난 오브젝트 파일 비교 방법을 사용하였다. 소스 비교가 아닌 오브젝트 비교 방법은 2.3.3절에서 설명한 바와 같은 장점이 있으며 오브젝트 비교 부분의 구현은 BSD Library를 사용하여 함수의 심볼 정보를 가지고 코드 영역을 비교하였으며 수정된 함수 코드를 추출하여 심볼 재배치를 한 후 차후 업그레이드를 수행하는  방법으로 구현 하였다.

### 4.6.2 추출된 함수의 심볼 재배치

함수 단위 코드영역 비교를 통해 추출된 함수 코드는 표 4.10과 같이 심볼 재배치가 수행되지 않은 상태로 이루어져 있다. 이러한 함수 코드는 타겟 장비에 동작하기 위해서는 심볼 재배치를 수행해야 한다.

표 4.10 심볼 재배치 수행 전 코드

```
00000000 <vfat_mkdir>:
   0:   e1a0c00d    mov     ip, sp
   4:   e92ddff0    stmdb   sp!, {r4, r5, r6, r7, r8, r9, sl, fp, ip, lr, pc}
   8:   e24cb004    sub     fp, ip, #4    ; 0x4
   c:   e24dd02c    sub     sp, sp, #44   ; 0x2c
                    ...
  2c:   e1a05000    mov     r5, r0
  30:   e5907094    ldr     r7, [r0, #148]
  34:   ebfffffe    bl      34 <vfat_mkdir+0x34>
  38:   e2504000    subs    r4, r0, #0    ; 0x0
  3c:   b1a08004    movlt   r8, r4
                      ...
  58:   e58da000    str     sl, [sp]
  5c:   e58d9004    str     r9, [sp, #4]
  60:   ebfffffe    bl      60 <vfat_mkdir+0x60>
  64:   e2508000    subs    r8, r0, #0    ; 0x0
  68:   1a000042    bne     178 <.text.vfat_mkdir+0x178>
  fc:   e59f0020    ldr     r0, [pc, #32]; 124 <.text.vfat_mkdir+0x124>
                    ...
 100:   ebfffffe    bl      100 <vfat_mkdir+0x100>
 104:   e1a00008    mov     r0, r8
 108:   e24bd028    sub     sp, fp, #40   ; 0x28
 10c:   e89daff0    ldmia   sp, {r4, r5, r6, r7, r8, r9, sl, fp, sp, pc}
 110:   e1a00005    mov     r0, r5
 114:   e1a01004    mov     r1, r4
 118:   ebfffffe    bl      118 <vfat_mkdir+0x118>
 11c:   ea00003f    b       220 <.text.vfat_mkdir+0x220>
 120:   00000000    andeq   r0, r0, r0
 124:   00000078    andeq   r0, r0, r8, ror r0
```

심볼 재배치는 현재 타겟에서 동작 중인 커널 버전의 함수 또는 전역변수의 메모리 주소를 얻어야하는 문제가 있다. 이러한 문제를 본 논문에서는 현재 타겟에서 동작 중인 버전의 리눅스의 심볼 정보를 포함하고 있는 vmlinux라는 ELF 파일을 읽은 후 해당 심볼 정보를 얻은 후 재배치를 수행하는 방법으로 구현하였다.

### 4.6.3 함수 내부에서 사용되는 심볼 정보가 없는 rodata 영역

함수 내부에서 사용되는 rodata영역이 수정되었을 경우 예를 들어 printf("Hello orld")가 printf("Hello World")로 수정되었을 경우에는 "Hello

World"라는 문자열에 대한 심볼 정보를 얻을 수 없다. 이러한 경우에는 수정된 rodata 즉 "Hello World" 문자열을 패치를 수행할 함수코드 뒤에 위치시킨 후 함수코드 뒤에 있는 문자열 주소로 심볼 재배치를 수행 하였다. 표 4.11은 함수 단위 업그레이드를 위해 타겟에 전송할 업그레이드 함수 이미지를 나타낸다.

**표 4.11 rodata 추가 후 업그레이드 함수 이미지**

| | | | |
|---|---|---|---|
| 함수 코드 | 0:<br>4:<br>8:<br>c:<br><br>2c: | mov<br>stmdb<br>sub<br>sub<br><br>mov | ip, sp<br>sp!, {r4, r5, r6, r7, r8, r9, sl, fp, ip, lr, pc}<br>fp, ip, #4 ; 0x4<br>sp, sp, #44 ; 0x2c<br>...<br>r5, #34 |
| 함수 내부에서 참조하는 rodata | 34: | 00000000 | "Hello World" |

### 4.6.4 타겟 시스템 리눅스 redirection handler 등록

redirection handler는 리눅스의 register_undef_hook을 사용하여 커널에 등록하여 사용을 했다. 표 4.12는 redirection handler의 실제 소스이다.

**표 4.12 redirection handler 소스 코드**

```
#define DYNUP_REDIRECTION_INST   0xe7ffdefe

static struct undef_hook dynup_redirection_hook = {
      .instr_mask = 0xffffffff,
      .instr_val = DYNUP_REDIRECTION_INST,
      .fn = dynup_arm_redirection_function
};

static int dynup_arm_redirection_function(struct pt_regs *regs, unsigned int instr)
{
      dynup_version_table_entry_t *table_entry;
      table_entry = dynup_version_table_find_entry(regs->ARM_pc);
      regs->ARM_pc = table_entry->trampoline_target_address;

      return 0;
```

```
}

int dynup_arm_redirection_init(void)
{
	register_undef_hook(&dynup_redirection_hook);
	return 0;
}
```

struct undef_hook 구조체 필드 중 instr_val에는 0xe7ffdefe 값을 넣었으며 0xe7ffdefe 명령어에 대한 핸들러를 fn 필드에 dynup_arm_redirection_function을 호출 하도록 등록하는 코드이다.

### 4.6.5 펌웨어 업그레이드 도중 장애 처리

펌웨어 업그레이드를 수행 도중 재 부팅등의 이유로 장애가 발생하였을 경우에는 다음에 부팅 시 업그레이드가 완료되지 않은 상태이므로 문제가 발생한다. 실제 펌웨어 업그레이드 장애 처리는 부트로더에서 나머지 펌웨어 업그레이드를 수행하는 방식으로 구현하였다.

# 제 5 장 결과 및 토의

본 논문에서는 기존 동적 업그레이드 시스템의 문제점인 동적 업그레이드를 위한 프레임워크를 제공하지 못하는 문제를 해결하기 위해 동적 업그레이드 프레임워크를 설계하였다. 본 논문에서 설계한 프레임워크를 실제 임베디드 리눅스를 대상으로 프로토타입 시스템을 구현하였다.

우선 구현한 동적업그레이드 프로토타입 시스템은 리눅스 커널 중 built-in된 커널 부분만 적용할 수 있는 시스템으로 구현 되었다. 또한 built-in된 커널 중 DynamOS에서 제안한 커널 동적 업그레이드에서 함수 내부에 사용되는 코드와 읽기전용 데이터가 업그레이드가 가능한 상태이다. 본 논문에서 구현된 프로토타입 시스템에 대해서 많은 질문들이 있었으며 내용은 다음과 같다.

built-in 커널 코드가 아닌 LKM으로 작성된 커널 함수에 대한 업그레이드 지원이다. LKM으로 작성된 커널일 경우에는 함수의 코드가 모듈을 등록할 때 결정된다. 이러한 함수는 주소는 리눅스의 /sys/module/<모듈이름>/section/.text 의 파일을 읽으면 텍스트 영역의 주소를 알 수 있으며 이러한 주소를 가지고 해당 함수를 동적으로 업그레이드 할 수 있다. 이러한 LKM으로 작성된 커널에 대해서도 차후 구현할 예정이다.

다음으로 업그레이드 코드를 업그레이드 할 경우에 대한 질문이 있었다. 본 논문에서 구현한 업그레이드 모듈을 업그레이드 할 경우에는 본 논문에서 구현한 방법이 함수의 제일 첫 코드에서 새로운 함수로 분기하는 방법이므로 업그레이드 코드를 업그레이드 한다고 해도 문제가 없다.

또한 본 논문에서 구현한 함수 단위의 코드 업그레이드는 매우 제한적인 업그레이드를 지원하지 않는가에 대한 질문이 있었다. 이는 Ksplice에서 리눅스 커널의 보안 패치 중 84%가 함수 단위로 코드 업그레이드로 가능하다고 하였으며 OPUS와 Ksplice를 제외한 동적 업그레이드 관련 논문에서는 100%의 동적 업그레이드 지원을 위해 데이터 스트럭쳐 업그레이드를 시도 하였다. 하지만 이러한 방법은 개발과정과 업그레이드 과정의 구

분 없이 개발자가 업그레이드를 위해 소스를 분석하고 업그레이드 코드를 추가적으로 작성해야 하는 등의 일을 수행해야 한다. 특히 데이터 스트럭쳐 업그레이드를 위해 업그레이드 코드를 추가적으로 작성하는 행동은 Ksplice 논문에서 주장한 바와 같이 개발자가 잘못된 코드를 삽입할 가능성이 있다. 본 논문에서는 함수 단위로 코드 업그레이드만 가능하나 기존 논문과 차이점은 프레임워크를 통해 동적 업그레이드가 불가능한 경우에는 정적 업그레이드 수행하는 방법을 사용하도록 하였다.

마지막으로 응용프로그램을 업그레이드에 대한 질문이 있었다. 2장에서 설명한 Ginseng과 POLUS는 소스 단위 비교를 통한 응용프로그램을 업그레이드하는 방법에 대한 기존 연구의 수행 방법을 본 연구에서 수행한 프레임워크에 통합하여 구현하면 문제가 없을 것이라고 본다.

현재 구현된 동적 업그레이드의 한계는 동적 업그레이드의 특성상 바이트 레벨로 비교하는 정적 업그레이드와 달리 기계어 의존적으로 업그레이드 패키지가 작성된다는 것이다. 또한 동적 업그레이드를 수행하면 메모리상의 수정된 코드를 교체하는 방식이 아니라 새로운 코드를 삽입한 후 해당 코드로 점프하는 방법으로 수행하여 더 많은 메모리를 요구한다는 단점이 있다.

# 제 6 장 동적 업그레이드 시스템 활용

본 시스템에서 구현한 동적 업그레이드 시스템은 앞으로 임베디드 소프트웨어의 복잡성에 따른 시스템 오류가 증가하고 있으며 또한 휴대폰과 셋탑박스 등의 시스템에서는 임베디드화된 범용운영체제의 사용이 증가되는 추세이다. 이러한 운영체제는 예전과 같이 제조사 독립적으로 개발한 운영체제가 아니므로 보안 문제가 대두되고 있다(Mikko Hypponen, 2006).

특히 현재 모바일 분야와 IPTV 셋탑박스에서의 운영체제 보안 패치를 수행하면 플래쉬 메모리에 있는 운영체제 펌웨어를 수정해야 하므로 상당한 시간을 요구된다. 예를 들어 모바일의 FOTA 업그레이드를 수행하려면 앞 절에서 설명한 in-place update와 같은 기술을 적용하기 때문에 적어도 1분이상의 시간이 소요된다. 이러한 문제는 사용자의 보안에 대한 인식 결여와 함께 보안 업그레이드를 수행하지 않을 것으로 예상된다. 이러한 문제를 해결할 방안으로는 현재 PC또는 서버환경에서 사용하고 있는 PMS와 같이 방법으로 모든 모바일 또는 셋탑박스 시스템에 에이전트 기반의 업그레이드 시스템을 설치하여 사용자가 모르게 동적으로 보안 패치를 수행하는 것이 반드시 필요하다.

또한 24시간 가동되는 교환기 시스템과 생산라인의 로봇과 같은 시스템은 재부팅에 따른 손해로 인해 업그레이드를 진행하지 못하는 상황이다. 업그레이드를 수행한다고 해도 동적 라이브러리만 바꿔서 업그레이드를 수행하는 상황이다. 이러한 시스템에 전반적인 동적 업그레이드가 가능하도록 만드는 것은 중요하다.

# 제 7 장 결론 및 향후 연구

동적 업그레이드에 관한 연구는 서비스의 상시성이 요구되며, 사용자가 업그레이드와 관련된 기술적인 절차를 수행하기 어려운 환경과 사용자의 필요성 인식이 부족한 보안 업그레이드에 관련해서 시스템 중단 없는 업그레이드 진행이 반드시 필요하다.

이러한 동적 업그레이드에 대한 연구는 현재 동적업그레이드에 대한 프레임워크를 제공하고 있지 못한 상태이다. 우리는 이러한 동적 업그레이드 프레임워크를 설계하였으며 프레임워크를 실제 바이너리 패치 기반의 동적 업그레이드를 사용하여 임베디드 시스템에 적용하였다. 이클립스와 같은 통합 개발 환경에 개발 호스트 측 소프트웨어 모듈을 통합하였고 OMA-DM 프로토콜을 사용하는 서버 측과 타겟 클리이언트를 구현하였다.

우리는 현재 바이너리 패치 기반의 동적 업그레이드에서의 함수 코드 단위로 업그레이드가 가능하나 데이터의 업그레이드를 지원하지 못한다. 이러한 데이터의 동적 업그레이드와 프레임워크의 버전 관리자와 보안 관리자 부분은 앞으로 향후 연구를 진행하여 구현 할 예정이다. 또한 임베디드화 된 범용 운영체제와 상용 실시간 운영체제 센서네트워크에 사용되는 운영체제를 대상으로 우리가 만든 프레임워크를 적용하여 임베디드 시스템에서 사용되는 모든 운영체제에 적용할 수 있는 플랫폼 독립적인 동적 업그레이드 프레임워크를 구성해 나갈 것이다.

# 참 고 문 헌

## <국내문헌>


• 경주현, 이민석, 「임베디드 시스템을 위한 동적 업그레이드 프레임워크에 관한 연구」, 정보과학회 학술발표 논문집, 2008

• 박현찬, 김세원, 유혁, 「리눅스 환경에서의 함수 단위 동적 커널 업데이트 시스템의 설계와 구현」, 정보과학회 학술발표 논문집 2007

• 이민재, 「점진적 대치를 이용한 소프트웨어 업데이트의 개선」, 한국산업기술대 지식기반기술·에너지대학원, 석사학위논문, 2007


## <국외문헌>


• Andrew Baumann et al., 「Module Hot-Swapping for Dynamic Update and Reconfiguration in K42」, LCA 2005

• Ariel Tamches, Barton P.Miller, 「Fine-grained dynamic instrumentation of commodity operating system kernels」, PhD thesis, University of Wisconsin, 2001

• Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. 「Sos: A dynamic operating system for sensor nodes」. 2005

• Kristis Makris, Kyung Dong Ryu,「Dynamic and Adaptive Updates of Non-Quiescent Subsystems in Commodity Operating SystemKernels」. 2007

• George C. Necula, Scott McPeak, Shree Prakash Rahul, Westley Weimer,「Cil:Intermediate language and tools for analysis and transformationof c programs」, 2002

• Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew.

「POLUS: A POwerful Live Updating System」.In Proc. ICSE, 2007

• Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang and Pen-Chung Yew, 「Live. updating operating systems using virtualizatio n」. In Proc. VEE, Ottawa, Canada, 2006

• Iulian Neamtiu, Michael W. Hicks, Gareth Stoyle, Manuel Oriol. 「Practical dynamic software updating for C」. In Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI), 2006

• Koshy J. and Pandey R. Remote Incremental Linking for Energy-Efficient Reprogramming of Sensor Networks」. In Proceedings. of the European Workshop on Sensor Networks, Istanbul, Turkey, 2005

• Jeong J and Culler D. 「Incremental network programming for wireless sensors」. In Proceedings of the First IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks (IEEE SECON), Santa Clara, CA, 2004

• 「Ksplice: Rebootless Linux kernel security updates」, http://web.mit.edu /ksplice/, 2008

• Larus, J., and Schnarr, E. 「EEL: Machine-Independent Executable Editing」. In ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation, ACM SIGPLAN, 1995

• Liz Laffan, Andreas Constantinuou, 「Mobile Software management」, 2007

• Hicks. M. W, 「Dynamic Software Updating」. PhD thesis, The University of Pennsylvania, 2001

• Reijers N. and Langendoen K, 「Efcient code distribution in wireless

sensor networks」. In Proceedings of the Second ACM International Conference on Wireless Sensor Networks and Applications, 2003

• Srivastava, A., and Eustace, A. 「ATOM: A System for Building Customized Program Analysis Tools」. In ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, ACM SIGPLAN,1994

• Young-Pil Kim, Jin-Hee Choi and Chuck Yoo, 「A Trap-based Mechanism for Runtime Kernel Modification」, 6th International Conference on Computer and Information Technology, 2006

## <관련 웹 사이트>

• http://sourceforge.net/projects/syncml-ctoolkit
• http://www.redbend.com/
• http://www.funambol.com/
• http://www.broadband-forum.org/
• http://www.openmobilealliance.org/
• http://en.wikipedia.org/wiki/Package_management_system

# ABSTRACT

## Dynamic Software Upgrade for Embedded Systems

Kyung, Ju-Hyun
Major in Computer Engineering
Dept. of Computer Engineering
Graduate School
Hansung University

OS kernels and applications aiming of serving uninterrupted service require frequent additions of functions, performance improvement, and bug fixes. These kinds of systems currently need costs that result from system-stop and reboot when being upgraded. In particular, it seems not easy for OS kernels and applications for embedded systems to be reinstalled. This dynamic upgrade for embedded systems is dependent on platforms in lots of parts, unlike a personal computer. The parts which are dependent on platforms need much time and efforts for a porting to other platforms.

In this paper, to solve these problems, a framework of dynamic upgrade, for an embedded system, is designed and prototype system is implemented.