# 모바일 폰 디버깅을 위한 ARM용 실행파일 분석도구 구현에 관한 연구

국종혁*, 김홍규, 문승진
수원대학교 컴퓨터학과
e-mail:zippest* ,exxxfx, sjmoon@suwon.ac.kr

# A Study on the Implementation of ARM-based executable file Analyzing Tool for debugging of mobile phone

Jong-Hyuk Kook*, Hong-Kyu Kim, Seung-Jin Moon
Dept of Computer Science, The University of Suwon

요 약

빠르게 변화하고 있는 모바일 시장에서 개발자들이 응용프로그램 개발을 하면서 가장 자주 사용하고 필요로 하는 것 중 하나가 디버깅이다. 본 논문에서는 기존의 하드웨어 디버거를 통해 ELF 디버깅 정보를 얻어오는 방법 대신 하드웨어 없이 ELF 파일에 대한 정보를 분석하는 툴 구현을 제안한다. ELF 파일을 분석하고 그 정보를 바탕으로 실제 모바일 폰에서 덤프 해온 메모리 값을 보여줌으로써 디버깅을 용이하게 한다. 소프트웨어만으로 디버깅 정보 추출이 가능하므로 고가의 디버깅 장비의 비용 절감의 효과를 가져다 줄 수 있을 것으로 기대된다.

## 1. 서론

개발자들이 모바일 폰 응용프로그램을 개발함에 있어 중요시 되는 것 중 하나가 디버깅이다. 현재는 대부분이 하드웨어 디버거를 통해서 디버깅 정보를 추출하여 디버깅을 하고 있다. 하드웨어 디버거를 이용하면 하드웨어 비용이 증가하게 되고 디버깅이 필요할 때 항상 장비가 구비되어있어야 하는 제약사항이 있기 때문에 본 논문에서는 최종 목표인 소프트웨어 디버거를 제작하기 위해서 그 첫 번째 단계로 ARM용 실행파일 인 ELF 파일을 분석하는 도구를 개발하여 그 도구를 통하여 ELF 파일의 정보를 추출하는 방법을 제안한다. 추후에는 그 정보를 바탕으로 실제 모바일 폰 으로부터 덤프 해온 메모리의 call stack 정보, ISR stack 정보를 개발자들에게 보여주는 것을 목표로 한다. 본 논문의 구성은 다음과 같다. 2장에서는 ELF 파일의 정의와 구조를 기술하고 3장에서는 ELF 파서 구현의 전반적인 시스템 구성도 및 환경구축, ELF 파일을 파싱하는 알고리즘 구현에 대해서 기술한다. 마지막으로 4장에는 결론과 향후 구현해야 할 사항들을 제시한다.

## 2. 관련 연구

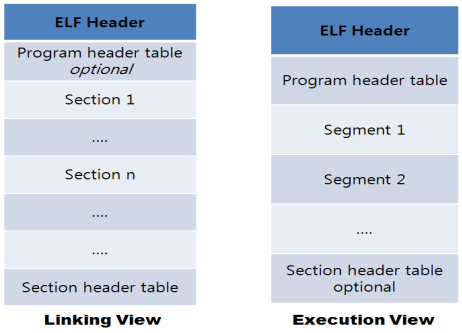
### 2.1 ELF의 정의

ELF(Executable and Linking Format)는 바이너리 파일로서 unix system laboratory에서 개발되고 발전되어왔다. SVR4와 solaris 2.X version의 운영체제에서는 기본적인 실행 파일 포맷으로 사용되고 있다. Tool Interface Standards committee(TIS)에서는 ELF 표준을 다양한 운영체제를 위해서, 32비트 인텔 아키텍처 환경에서 동작하는 이식 가능한 목적파일로 선택하였다. ELF 파일은 개발자가 프로그래밍 한 소스에 관한 모든 정보를 내장하고 있고 컴파일러에 따라 각각 다를 수 있다. ELF 표준은 다양한 운영체제에 걸쳐서 사용될 수 있는 이진 인터페이스를 개발자에게 제공함으로써, 소프트웨어 개발에 연

계성을 주기위해 만들어졌는데 서로 다른 여러 인터페이스의 구현을 방지하고, 프로그램을 다시 짜고 재 컴파일을 해야 할 필요를 줄이도록 하기 위함이다.

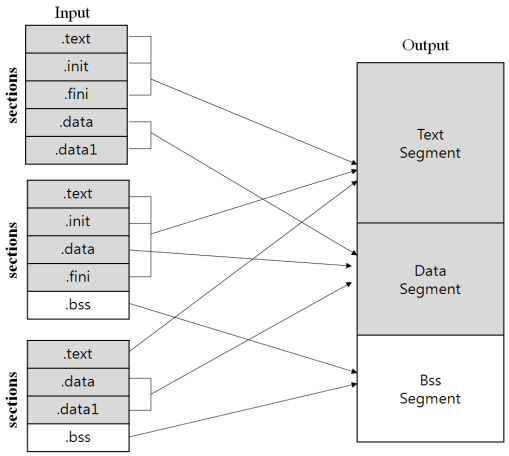
## 2.2 ELF 파일 구조

ELF 파일은 프로그램 목적에 따라 두 가지 관점에서 바라 볼 수 있도록 되어 있다. 어셈블러나 정적링커는 섹션 헤더 테이블을 통해 파일을 섹션들의 모음으로 보게 되고, 로더나 동적링커는 프로그램 헤더 테이블을 통해 파일을 세그먼트의 모음으로 바라보게 된다. 섹션 단위는 기계어나 심볼 테이블과 같은 아주 구체적인 정보를 포함하게 된다. 반면에 세그먼트는 동일한 메모리 속성을 갖는 섹션을 하나 이상 포함한 더 큰 단위이다.




(그림 1) ELF 파일의 구조

그림 1에서와 같이 ELF 파일의 ELF 헤더는 파일 내에 고정된 위치를 가지는 정보로서 헤더정보를 통해 파일의 전체적인 구조와 분석정보를 파악 할 수 있다. ELF header는 ELF 의 버전, 기계정보, 데이터 저장방식, program header table과, section header table의 위치와 크기 등의 정보를 포함하고 있다. section header table은 파일의 각 섹션들의 이름, 형태, 변위, 크기 등의 정보들을 가지며 program header table은 ELF 파일에서 세그먼트에 대한 정보와 메모리에 올리는 방법을 기술한다.

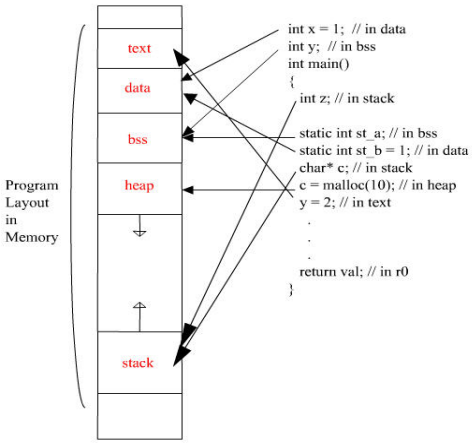



(그림 2) 목적파일 링크과정

프로그램 헤더에서는 그림 2에서와 같이 ELF 의 각 섹션들을 세그먼트 단위로 묶어 메모리에 적재하게 된다.

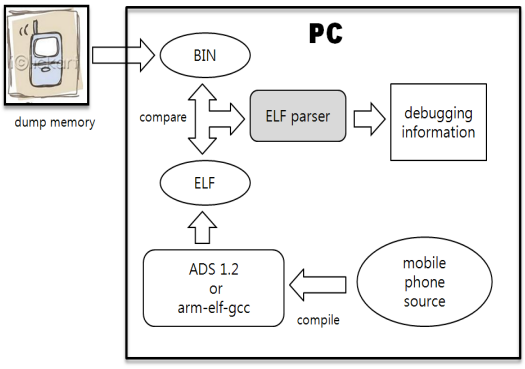
## 2.3 C 소스에서의 메모리 구성

C 소스에서 각 변수의 메모리 적재 단위는 그림 3과 같다. 전역 변수와 정적 변수는 모두 text 세그먼트에 적재되고 초기화 되지 않은 전역 변수와 정적 변수는 bss 라는 세그먼트에 적재된다. bss 세그먼트는 실제 데이터가 없으므로 실행 시 메모리를 차지하지 않고 이름과 크기만을 기억해둔다. *main()* 함수 안에 있는 지역변수들은 스택에 적재 되며 사용자가 *malloc()*과 같은 동적으로 할당한 변수들은 힙이라는 메모리 공간에 적재된다.




(그림 3) C 소스에서의 메모리 적재

## 3. 개발 환경 및 ELF Parser 구현




(그림 4) 전체적인 시스템 구성도

전체적인 시스템 구성도는 그림 4와 같다. 모바일 폰 소스를 컴파일한 결과물로 ELF 파일이 생성되면, ELF parser를 이용하여 ELF 파일을 분석하고 모바일 폰에서 덤프 한 BIN 파일과 비교하여 그 위치를 찾아 낸 후 실제 모바일 폰 메모리의 call stack, ISR stack 같은 디버깅 정보를 추출해 낸다.

### 3.1 ARM 크로스 컴파일러 환경 구축

모바일 폰에 사용되는 프로세서가 ARM 계열이기 때문에 ARM용 ELF파일을 생성하기 위해서는 ARM machine 에 맞는 컴파일러로 소스를 컴파일을 해야 한다. 그러기 위해서는 크로스컴파일러 환경구축이 필요하다. 크로스컴파일러란 수행되는 시스템과 다른 시스템에서 수행될 수 있는 코드를 생성하는 컴파일러를 말한다. ARM용 크로스컴파일러 환경을 구축하려면 상용 컴파일러인 ADS(ARM Developer Suite)나 다른 ARM용 컴파일러를 사용하면 된다. 본 논문에서는 크로스컴파일러 환경을 구축하기 위해서 오픈소스인 arm-elf-gcc 패키지를 다운로드 받아 사용하였다. 다른 크로스컴파일러 패키지로 arm-linux-gcc 가 있었으나 이 패키지는 동적으로 소스를 컴파일하여 ELF 파일을 생성하므로 실제 모든 디버깅 정보가 정적으로 올라가야하는 ELF 파일을 생성하기에는 적합하지 않았다.

### 3.2. ELF Parsing Algorithm 구현

바이너리 파일인 ELF 파일에서 구체적인 디버깅 정보를 추출하려면 리눅스의 *"/usr/include/elf.h"* 파일을 참조하여야 한다. elf.h 파일에는 ELF 파일의 구조가 구조체로 정의되어 있기 때문에 그 구조체 형식을 참조하여 ELF 파일의 데이터 엔코딩 방식, 즉 빅 엔디언 방식과, 리틀 엔디언 방식으로 구분하여 ELF 파일을 읽어 와야 한다. 현재 모바일 폰 안의 내장된 프로세서가 ARM 이기 때문에 ARM용에 사용되는 데이터 엔코딩 방식인 리틀 엔디언 방식으로 데이터를 읽어 와야 한다. 만일, 모바일 폰이 문제가 발생하여 정지하였다면, 그것을 디버깅하기위해 현 상태 메모리의 스택이나 힙 등의 공간에 어떠한 값이 들어가 있는지 개발자가 알 필요성이 있다. 이를 위해서 ELF 파싱이 필요하다. ELF 파일을 파싱하게 되면 메모리의 어느 부분에 스택이나 힙 등이 위치해있는지 그 주소값을 알 수 있기 때문이다. 여기서는 가장 중요한 섹션헤더 부분의 파싱 알고리즘만을 기재하였다.

```
/* Section header.  */
typedef struct
{
  Elf32_Word   sh_name;         /* Section name (string tbl index) */
  Elf32_Word   sh_type;         /* Section type */
  Elf32_Word   sh_flags;        /* Section flags */
  Elf32_Addr   sh_addr;         /* Section virtual addr at execution */
  Elf32_Off    sh_offset;       /* Section file offset */
  Elf32_Word   sh_size;       /* Section size in bytes */
  Elf32_Word   sh_link;       /* Link to another section */
  Elf32_Word   sh_info;       /* Additional section information */
  Elf32_Word   sh_addralign;    /* Section alignment */
  Elf32_Word   sh_entsize;      /* Entry size if section holds table */
} Elf32_Shdr;
```

(그림 5) section header

그림 5는 section header 의 구조체이다. 이 구조체에는 section header 의 이름이나 타입, 변위, 크기 등의 정보가 정의 되어 있다. 구조체를 보면 *Elf32_Word* 처럼 변수의 타입을 선언하고 있는데 이는 4바이트를 의미한다.

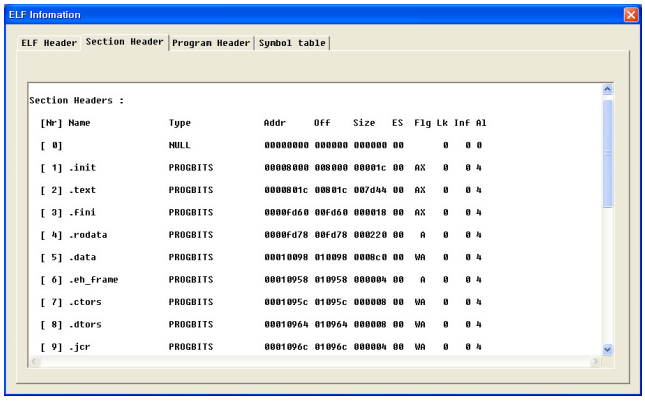
<표 1> section header parsing code

```
int CMainFrame::process_section_headers (FILE * file) {
  Elf_Internal_Shdr * section;
  int i;
  if (elf_header.e_shnum == 0) {
    AfxMessageBox("There are no Sections in this file");
    return 0;
  }
  if (! get_32bit_section_headers (file))
    return 0;
  section = section_headers + elf_header.e_shstrndx;
  if (section->sh_size != 0)  {
    unsigned long string_table_offset;
    string_table_offset = section->sh_offset;
    if (fseek (file, section->sh_offset, SEEK_SET))     {
                    return 0;
    }
          string_table = (char *) malloc (section->sh_size);
    if (string_table == NULL){
      return 0;
    }
    if (fread (string_table, section->sh_size, 1, file) != 1) {
                    free (string_table);
      string_table = NULL;
      return 0;
    }
    dynamic_symbols = NULL;
    dynamic_strings = NULL;
    dynamic_syminfo = NULL;
    for (i = 0, section = section_headers;(unsigned int)i <
elf_header.e_shnum;i ++, section++){
      char * name = SECTION_NAME (section);
      if (section->sh_type == SHT_DYNSYM)           {
        if (dynamic_symbols != NULL)    {
            continue;
        }
        num_dynamic_syms = section->sh_size / section->sh_entsize;
      }
      else if (section->sh_type == SHT_STRTAB && strcmp (name,
".dynstr") == 0) {
            if (dynamic_strings != NULL){
                AfxMessageBox("File contains multiple dynamic string
tables₩n"));
              continue;
             }
            if (fseek (file, section->sh_offset, SEEK_SET)){
                        return 0;
            }
            dynamic_strings = (char *) malloc (section->sh_size);
                    if (dynamic_strings == NULL){
              return 0;
            }
            if (fread (dynamic_strings, section->sh_size, 1, file) != 1) {
                        free (dynamic_strings);
                                dynamic_strings = NULL;
                                    return 0;
             }
           }
         }
                  ...................//중략
      }
      return 0;
}
```

이것은 그 타입에 따른 바이트 수만큼 ELF 파일에 그 정보가 공간을 차지하고 있다는 것으로 해석된다. 그래서

각 바이트 수를 더하면 ELF header의 크기가 총 52byte 임을 알 수 있다. 그림 5에서 알 수 있듯이 섹션헤더의 크기는 40byte이다. ELF 헤더는 파일의 맨 첫 부분에 위치하므로 파일의 첫 52byte 가 ELF 헤더라는 것을 알 수 있다. 여기서 알아두어야 할 사항은 파일의 고정된 위치를 차지하는 것은 ELF 헤더뿐이라는 사실이다. ELF 헤더만이 파일의 고정된 위치를 차지하고 나머지 프로그램 헤더나 섹션헤더는 ELF 헤더에 파일 내 변위 정보가 있기 때문에 그것을 참조하여 따라가야만 정보를 추출 할 수 있다. 섹션헤더를 예로 들면 ELF 헤더부분에 *e_shoff*라는 변수에 섹션헤더의 시작주소가 저장되어 있기 때문에 ELF 파일에서 그 시작주소를 따라가면 섹션헤더를 찾을 수 있다는 것이다. ELF와 마찬가지로 모바일 폰 memory에도 0번지부터 데이터가 저장되기 때문에 ELF에서 추출한 변위 정보를 참조하면 실제 memory에서의 주소를 가지고 섹션헤더의 위치를 찾을 수가 있다.

표 1은 visual c++ 6.0 툴로 작성한 ELF section header table을 파싱하여 출력하는 코드의 일부분이며 그림 6은 C로 작성한 간단한 코드를 arm-elf-gcc를 이용하여 ELF 파일을 생성하고, 그 ELF 파일을 파싱 하여 섹션헤더 정보를 출력해준 결과 화면이다.




(그림 6) section header parsing 결과

그림 6을 보면 각 섹션에 대한 타입과 변위, 크기 등의 정보를 알 수 있다. 이 정보를 토대로 모바일 폰에서 덤프해온 바이너리 파일의 call stack 주소를 비롯한 다른 디버깅 정보를 찾아 낼 수 있다.

## 4. 결론 및 향후 연구 방향

본 논문에서는 모바일 폰 을 비롯한 다른 ARM 프로세서를 사용하는 머신, 또는 ELF 파일을 사용하는 모든 환경에서 디버깅 정보를 추출하기 위하여 ELF 파일을 파싱 하는 알고리즘을 제안하고 구현하였다.

현재 구현된 도구는 ELF 파일의 ELF header, program header, section header, symbol table 의 정보를 개발자들에게 보여준다. 기존의 개발자들은 하드웨어 디버거를 통해 파싱된 ELF 파일의 정보로 메모리의 스택이나 힙 등에 저장되어 있는 값을 확인 할 수 있었으나 본 논문에서 제안한 ELF 파싱 알고리즘을 통해 ELF 정보를 추출하면 하드웨어 없이 디버깅 정보를 확인할 수 있기 때문에 하드웨어에 대한 비용절감의 효과를 얻을 수 있고 어느 곳에서도 사용가능하기 때문에 공간에 대해서도 효율적이다. 향후 연구에서는 ELF 파일을 통해 실제 모바일 폰에서 덤프 해온 바이너리 파일의 스택 정보를 개발자들이 좀 더 쉽게 알아 볼 수 있도록 트리 뷰 형태로 구현하고 그 스택 정보를 덤프 된 바이너리 파일 과 맵핑시켜 수정 가능하게 할 것이며, 추후에는 오류정보를 스크립트로 저장하여 자동으로 오류수정을 가능하게 하는 auto tester의 개발도 고려중이다.

## 참고문헌

[1] TIS Committe, *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2*

[2] TIS Committe, *Tool Interface Standard (TIS) DWARF Debugging Information Format Specification version 2.0*

[3] Wookey, Chris Rutter, Jeff Sutherland, paul Webb, *The GNU Toolchain for ARM targets HOWTO*

[4] Development Systems Business Unit Engineering Software Group, *ARM ELF*, 8 June, 2001

[5] www.arm.com, AXD and armsd Debuggers Guide(ARM DUI 0066) & ARM Developer Suite Debug Target Guide (ARM DUI 0058D)

[6] "KLDP ", (http://www.kldp.org)

[7] "Korea Embedded Linux Project", (http://www.kelp.or.kr)