Embedded RTOS

QEMU를 통해 임베디드 RTOS 구현하기

요약

임베디드 컴퓨터란 무엇인가?

QEMU, RTOS란 무엇인가?

부팅하기

UART, 타이머 개발

스케줄러, 이벤트 개발

**작성자** 53기 문연수

[yyyynoom@gmail.com](mailto:yyyynoom@gmail.com)

1. Narrative

특정 용어를 한글로 옮겼을 때 그 의미가 명확하지 않다면 원어 그대로 사용한다. 그 외의 모든 용어는 한글로 옮겨 작성한다. 가령 후술할 ‘hard real-time’의 경우 ‘단단한 실시간’, ‘강력한 실시간’ 등의 단어로 번역해도 그 의미가 직관적이지 않기에 논문에 나온 ‘hard real-time’ 이라는 용어를 그대로 사용했다.

일부 용어들의 경우 원어를 발음에 따라 한글로 옮겨 적은 경우가 다수 있는데 이는 해당 용어가 원어 혹은 번역된 용어보단 발음 그대로 쓰이는 것이 관용적인 경우이다. 예를 들어 task는 그 뜻이 “a piece of work, especially something unpleasant” 이기에 ‘과업’ 혹은 ‘과제’ 라는 단어로 옮길 수 있으나 컴퓨터, 특히 운영체제에서 task는 “컴퓨터 시스템 내 활동의 기본적인 단위.” 라는 뜻으로 쓰인다. 또한 이러한 의미로 쓰이는 모든 것들을 관용적으로 태스크라 부른다. 이러한 경우 task를 원어 그대로 쓰거나 ‘과업’, ‘과제’ 등으로 옮기지 않고 한글로 ‘태스크’라 작성한다.

약어의 경우 처음 등장할 때에는 생략하지 않고 그대로 작성한다. 그 다음 해당 용어 뒤에 소괄호와 함께 ‘이하 ~(물결표)’ 로 작성한다. 이후 나오는 ‘~’에 해당하는 약어들은 해당 소괄호 앞에 붙은 용어를 의미한다. **2. Background Theory**에 나오는 ‘Operating System’의 경우 “Operating System(이하 OS)” 로 작성하였다. 해당 서술 다음 나오는 모든 약어 ‘OS’는 ‘Operating System’을 뜻한다.

약어가 동일하여 의미가 중복되는 경우가 발생할 수 있으나 문맥상 그 의미를 쉽게 구분이 가능하다면 약어 그대로 작성한다. 그러나 약어의 의미가 중복되면서 그 사이의 구분이 어렵다면 원어로 작성한다.

1. Introduction

임베디드 시스템에서 동작하는 Real-Time Operating System(이하 RTOS)를 개발하는 것이 목적이다. 모든 동작이 언제나 정해진 시간에 수행되진 않으나, 어느 정도의 적시성을 보장하며, 사용자가 발생시킨 이벤트에 따라 적절한 동작을 수행하는 Soft Real-Time OS를 구현한다. 개발 환경은 다음과 같다:

* 개발 PC의 운영체제: Linux
* 컴파일러: ARM-GCC
* 하드웨어: ARM RealView Platform Baseboard for Cortex-A8(이하 RealViewPB)
* 시계이(가) 표시된 사진

  자동 생성된 설명에뮬레이터: QEMU

Figure RTOS개발 순서

개발 순서는 다음과 같다:

1. 개발 PC(Linux)에서 코드를 작성한다.
2. 코드를 크로스 컴파일러(GCC-ARM)를 통해 타깃 아키텍처(GCC, RealViewPB)에 맞춰 컴파일 한다.
3. 생성된 바이너리 이미지를 QEMU 에뮬레이터를 통해 실행시켜 테스트를 진행한다.

1~3 순서를 반복하며 개발을 진행한다.

1. Background Theory

기본적으로 운영체제, 컴파일러, C, ARM어셈블리, 그리고 ARM 아키텍처에 대한 약간의 지식이 있다는 것을 가정하고 학술지를 작성하였다. 따라서 일부 용어들(CPSR, ELF, Context Switching, volatile keyword, etc.)의 경우 그 뜻과 기능에 대해서 따로 설명하지 않고 넘어갈 수 있음을 알린다.[[1]](#footnote-1)

* **임베디드 컴퓨터 (Embedded Computer)**  
   임베디드 컴퓨터는 다른 장치에 포함되어 있어서 미리 정해진 한 가지 일이나 몇 가지 소프트웨어만 실행하는 컴퓨터이다. 즉 Personal Computer (이하 PC)처럼 여러가지 기능 (음악과 동영상 재생, 게임, 문서 작성, etc.)을 수행하기보단 만들 때부터 그 기능이 한정되어 특정 기능만을 계속 수행하는 장비라는 뜻이다. (David A. Patterson, John L. Hennessy 2018, 5)  
   임베디드 컴퓨팅 장치도 소프트웨어가 있어야 동작한다. 규모가 작거나 극단적인 최적화가 필요한 임베디드 장치는 운영체제 없이 펌웨어로만 동작하기도 하지만 시스템의 자원과 복잡도를 관리하는 것이 더 중요한 임베디드 장치에서는 임베디드 운영체제를 사용하는 것이 일반적이다 (이만우 2019, 4).
* **RTOS (Real Time Operating Systems)** Operating system(이하 OS)이라는 용어에 “real-time” 이 쓰였을 때 의미하는 바는 간단히 말해 임베디드 애플리케이션이 외부 이벤트에 실시간으로 응답할 수 있음을 뜻한다. 그 반대는 다른 태스크(컴퓨터 활동 내 기본 단위, 아래에서 더 자세하게 다룬다.)를 기다리거나 혹은 수행이 완료될 때까지 기다리는 것을 의미한다. 이는 또 다시 ‘Hard real-Time’과 ‘Soft Real-Time’이란 용어로 나뉘게 된다.   
   hard real-time 은 그 행동이 **언제나** 정해진 마감 시간에 반드시 끝나야 함을 의미한다. 이는 비디오 스트리밍 처리, 자동차 엔진의 스파크 플러그의 점화, 도플러 레이더의 에코 처리 등을 포함한다.   
   Soft real-time 은 hard real-time 과는 다르지만 일종의 적시성을 필요로 하는 곳에 적용된다. 그 말인 즉, 마감 시간을 놓치는 것이 시스템의 무결성을 손상시키진 않으나, 해로운 효과를 가질 수 있음을 말한다. 이들은 소매점의 point of sale (POS) 시스템, ATM과 다른 신용 카드 장치, 그리고 PDA 등이 될 수 있다. (D. Stepner, N. Rajan, D. Hui 1999, 2)
* **QEMU**QEMU는 상표로 등록되지 않은, 그리고 오픈 소스인 machine emulator이자 virtualizer이다.  
   machine emulator를 사용하면, QEMU는 OS를, 그리고 하나의 장치(예, ARM 보드)에서 만들어진 프로그램을 다른 장치(예, 독자의 개인 PC)에서 구동이 가능하다. 또한 dynamic translation을 사용함으로써, 좋은 성능을 끌어낼 수 있다.  
   virtualizer를 이용하면, QEMU는 guest code를 호스트 CPU에서 직접 실행함으로써 네이티브와 가까운 성능을 달성할 수 있다. (Main Page, Aug 28, 2020)

1. 메모리 설계

개발에 들어가기 앞서, RTOS의 메모리 구조를 먼저 설계한다. 필자는 QEMU가 기본적으로 할당해주는 128MB를 활용해 메모리의 크기는 신경 쓰지 않고 넉넉하게 사용할 것이다. 메모리의 배치 역시 장치들의 특성을 고려하지 않고 그냥 선형적으로 배치한다.[[2]](#footnote-2)

우선 처음 시작 주소(0x00000000)부터 다음 1MB(0x000FFFFF)까지를 text 영역으로 사용할 것이다. 사실 리눅스나 Windows와 같은 거대한 운영체제를 개발하는 것이 아니기에 text 영역은 대충 수십 KB면 충분하지만 앞서 말했듯이 논의를 간단하게 하기 위해 메모리를 넉넉하게 잡아서 사용할 것이다.

다음으로 ARM의 7가지 동작 모드(Operating Mode) 각각의 스택으로 1MB씩 총 7MB를 할당하겠다. 0x00100000 부터 USR & SYS (2MB), SVC, IRQ, FIQ, ABT, UND 순서대로 각각 1MB씩 배치해서 0x007FFFFF까지를 동작 모드 별 스택으로 할당한다.

RTOS를 개발하는 것이기 때문에 RTOS 위에서 동작하는 태스크의 스택 영역도 할당해야 한다. 전체 태스크 스택의 크기는 64MB, 각 태스크의 스택 크기는 1MB로 책정할 것이다. 따라서 필자가 작성하는 RTOS의 최대 태스크의 개수는 64개가 된다. 마지막으로 전역 변수 영역을 1MB로 할당하겠다.

이렇게 할당하고도 남은 나머지 공간(55MB)을 동적 할당용 메모리로 사용하면 최종적으로 **Figure 2**와 같은 형태의 메모리 구조를 가지게 된다:

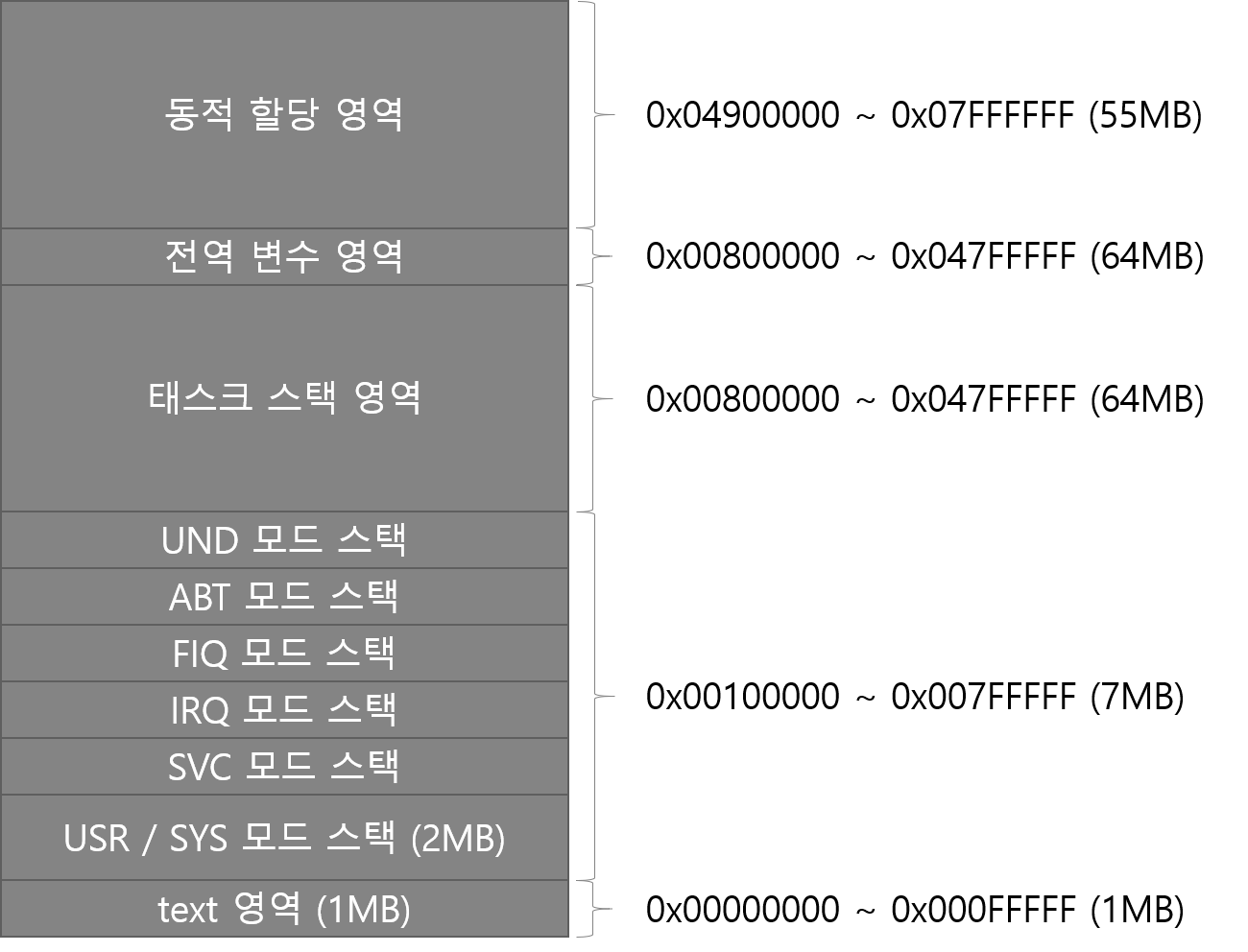
****

Figure 필자가 설계한 RTOS의 메모리 배치와 구성

1. Exception Vector Table

ARM에는 7개의 Exception이 있고 **Table 01**의 **설명**에 해당하는 상황이 발생하면 Program Counter(이하 PC)를 Exception Vector Table(이하 EVT)에 정의된 오프셋 주소의 값으로 강제 변환시킨다.

|  |  |  |
| --- | --- | --- |
| 오프셋 | 이름 | 설명 |
| 0x00 | Reset | 전원이 켜지면 실행됨 |
| 0x04 | Undefined Instruction | 잘못된 명령어를 실행했을 때 |
| 0x08 | SVC (Supervisor Call) | SVC 명령으로 발생시키는 exception |
| 0x0C | Prefetch Abort | 명령어 메모리에서 명령어를 읽다가 문제가 발생 |
| 0x10 | Data Abort | 데이터 메모리에서 데이터를 읽다가 문제가 생김 |
| 0x14 | Not used | 사용하지 않음 |
| 0x18 | IRQ interrupt | IRQ 인터럽트가 발생했을 때 |
| 0x1C | FIQ interrupt | FIQ 인터럽트가 발생했을 때 |

Table Exception Vector Table 정보

전원이 켜지면 ARM은 reset vector의 값을 PC에 대입한다. 일반적으로 EVT의 기본 위치는 0x00000000이고 reset vector의 오프셋도 0x00이기 때문에 보통ARM은 전원이 켜지면 그냥 시작 주소를 읽어 Program Counter(이하 PC)에 대입한다. (ARM Developer Jan 10, 2020)

EVT의 Exception은 각각 4바이트씩 할당되어 있고, 32비트 머신이기 때문에 하나의 Exception에 하나의 명령어만 실행이 가능하다. 명령어 하나로는 Exception을 제대로 다 처리할 수 없기 때문에 보통 여기에 Exception을 처리하는 코드로 분기하는 명령어를 넣는다. Exception을 처리하는 코드(26행의 reset\_handler)를 Exception Handler(이하 EH)라고 부른다.

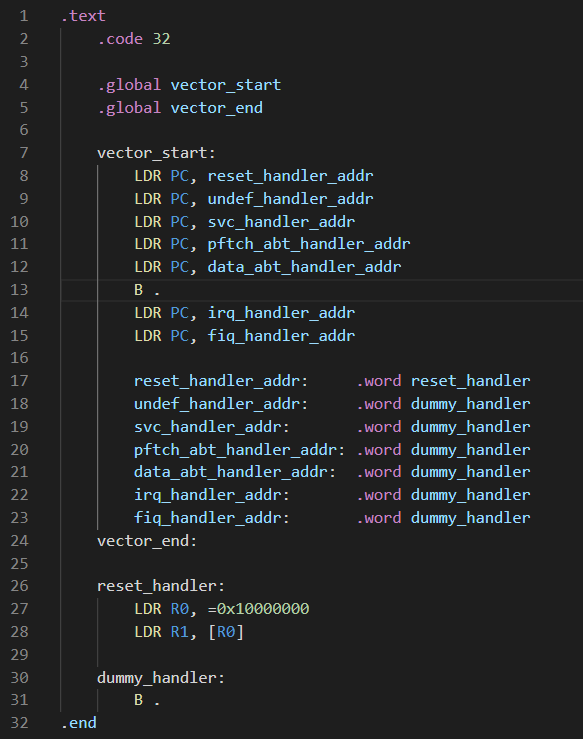
이제 위 논의를 토대로 EVT테이블을 만들어볼 것이다:

Figure EVT를 구현한 어셈블리 코드(Entry.S)

Figure EVT구현 코드

어셈블리 코드가 나와서 당황스러울 수 있겠지만 생각보다 별 것 없다. 처음으로 EVT 테이블을 등록한다. 사실 등록하는 과정이라고 할 필요도 없다. 그냥 Exception 순서대로 명령어를 배치해두면 ARM이 알아서 발생한 Exception에 따라 해당 코드를 실행한다. 이후 reset vector에 등록된 명령어(8행)가 실행되어 reset\_handler가 호출된다. reset\_handler는 0x10000000의 메모리 주소를 읽어 R1 레지스터에 저장한다. 이제 코드 라인별로 조금 더 자세하게 설명을 진행하겠다.

2행은 각 명령어의 크기가 32 비트(ARM 모드)라는 정보를 전달한 것이다. 앞서 EVT의 시작 주소를 기점으로 4바이트씩 떨어진 위치에 Exception을 처리하는 명령어가 배치된다고 했으므로 명령어의 크기를 32비트로 지정했다.

4, 5행의 vector\_start와 vector\_end를 왜 global로 선언했는지는 이후 Linker Script와 관련된 내용을 진행하면서 다시 한번 설명하겠다.

8~15행의 코드가 바로 Exception이 발생했을 때 분기해야 할 주소를 등록하는 코드이다. 지금은 아직 reset exception이 발생했을 때 분기하는 reset handler 만 등록했지만(이 역시 아직 내용이 부실하다) 나머지 EH는 진행하면서 천천히 구현하도록 하겠다. 13행은 0x14위치의 Not Used 영역으로 4바이트를 채우기 위해 그냥 무한 반복 명령어(현재 주소로 분기하는 명령어)를 집어넣었다.

26행부터 28행이 reset handler인데 특별한 내용은 없다. 그냥 0x10000000주소의 값을 읽어서 해당 값을 레지스터에 저장하는 코드이다.

이제 해당 어셈블리 코드를 타깃 플랫폼에 맞춰 컴파일하고 생성된 Executable and Linkable Format(이하 ELF)파일을 QEMU로 실행시키면 제대로 동작할까? 아쉽지만 제대로 된 결과를 보기 어려울 수 있다. 이는 컴파일러의 Linking 방식 때문인데 이는 다시 아래에서 자세하게 설명하도록 하겠다.

1. Linker Script

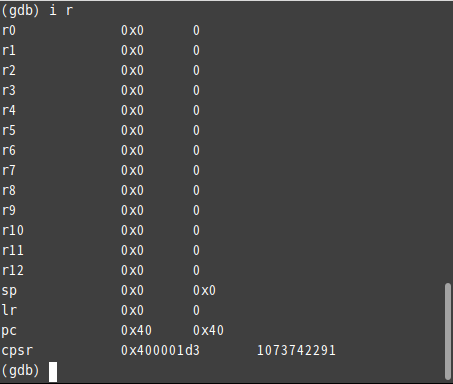
이제 EVT도 완성했으니 QEMU를 통해 실행시키면 될 것 같지만 실행시켜도 아무 것도 출력되지 않는다:

Figure gdb를 통해 확인해본 레지스터의 값

특별한 출력 코드를 넣지 않아서 그런 것도 있지만 reset\_handler에서 0x10000000으로 초기화한 r0 레지스터와 해당 주소를 읽어 그 값을 저장한 r1 레지스터의 값 역시 0이다. 무언가 문제가 있다.

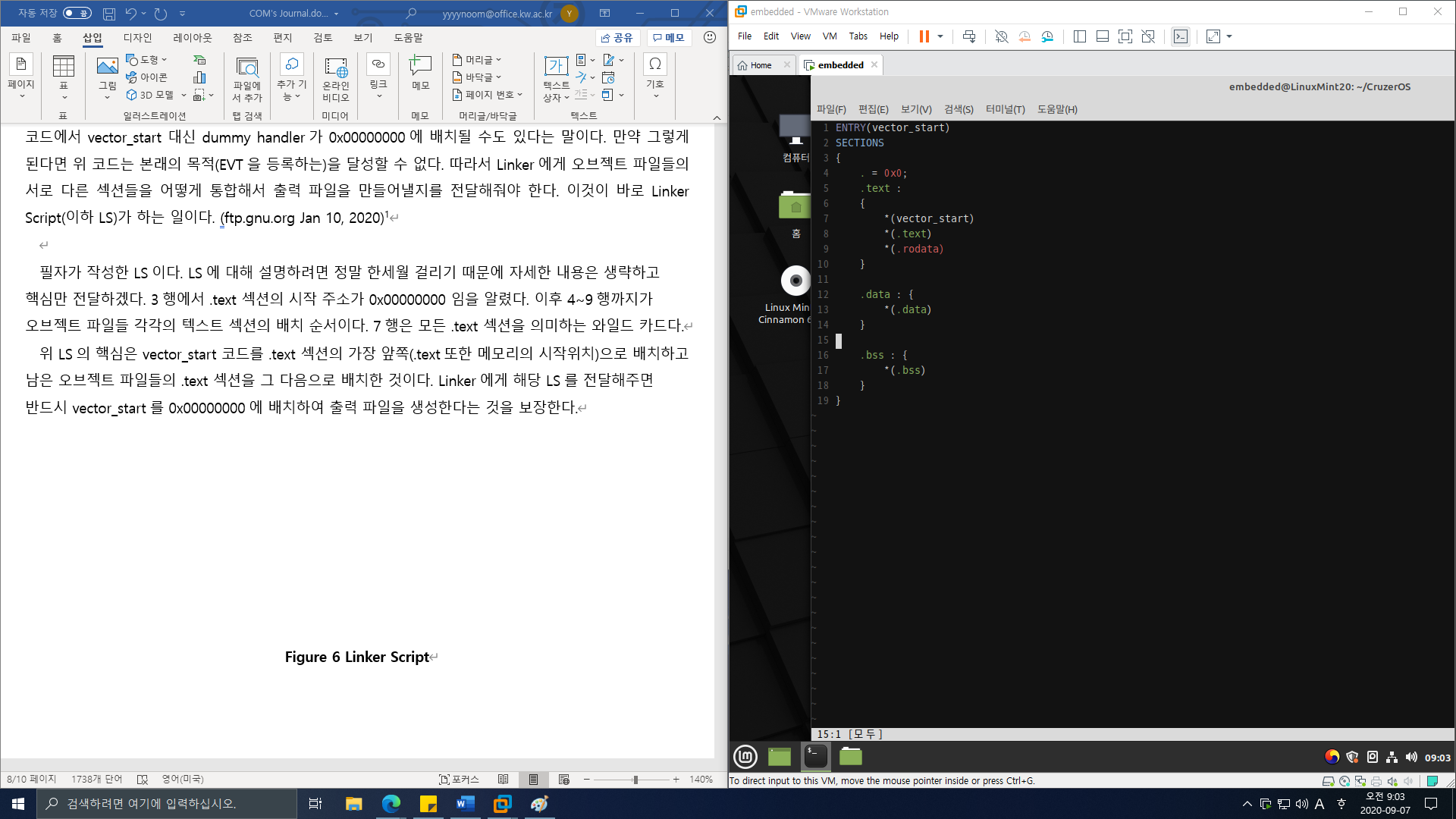
이는 Linker가 코드를 메모리에 배치하기 때문이다. 물론 이는 Linker가 당연히 해야 하는 일이지만 문제 상황은 다음과 같다: 수많은 오브젝트의 섹션을 무슨 순서로 그리고 어디에 배치할지 알 수가 없다. 위 코드에서 vector\_start 대신 dummy handler가 0x00000000에 배치될 수도 있다는 말이다. 만약 그렇게 된다면 위 코드는 본래의 목적(EVT을 등록하는)을 달성할 수 없다. 따라서 Linker에게 오브젝트 파일들의 서로 다른 섹션들을 어떻게 통합해서 출력 파일을 만들어낼지를 전달해줘야 한다. 이것이 바로 Linker Script(이하 LS)가 하는 일이다. (ftp.gnu.org Jan 10, 2020)[[3]](#footnote-3)

Figure Linker Script

LS에 대해 설명하려면 한세월 걸리기 때문에 자세한 내용은 생략하고 핵심만 전달하겠다. 1행은 entry point를 등록하는 코드이다. 전원이 들어오면 다시 reset vector로 분기하지만 의미를 명확하게 하기 위해 넣었다. 3행에서 .text 섹션의 시작 주소가 0x00000000 임을 알렸다. 이후 4~9행까지가 오브젝트 파일들 각각의 텍스트 섹션의 배치 순서이다. 7행은 모든 .text 섹션을 의미하는 와일드 카드다.

위 LS의 핵심은 vector\_start 코드를 .text 섹션의 가장 앞쪽(.text섹션 또한 메모리의 시작위치에 배치됨)으로 배치하고 남은 오브젝트 파일들을 그 다음으로 배치한 것이다. Linker에게 해당 LS를 전달해주면 vector\_start를 0x00000000에 배치하여 출력 파일을 생성한다는 것을 반드시 보장한다.

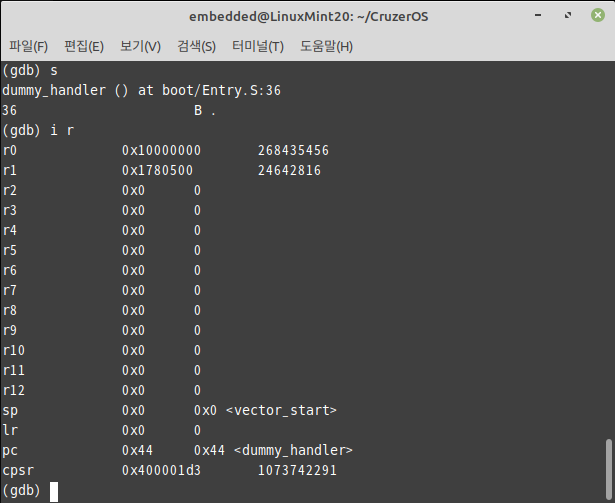
Linker Script를 등록하고 다시 gdb를 통해 레지스터의 값을 확인해본 결과 r0의 값도 0x10000000으로 초기화 되었고, r1 레지스터 역시 해당 주소를 통해 읽어온 값으로 초기화 되어 있다.

Figure gdb를 통해 다시 확인한 레지스터의 값

참고로 해당 주소(0x10000000)는 system control register 인 SYS\_ID에 해당하는 주소이다. 0x01780500 은 각각 REV(0x0), HBI(0x178), BUILD(0x0), ARCH(0x5), FPGA(0x00)의 해당하는 값들이다. (ARM Developer Jan 10, 2020)

1. UART

UART는 Universal Asynchronous Receiver/Transmitter의 약자로 범용 비동기화 송수신기이다. UART를 가장 먼저 작업하는 이유는 UART가 보통 콘솔 입출력 용으로 사용되기 때문이다. 물론 UART가 문자 통신용으로 만들어진 프로토콜은 아니지만 어떤 데이터 값이든 UART로 주고받을 수 있는 때문에 아스키 데이터를 주고받아 터미널을 통해 입출력이 가능해진다.

UART를 사용하기에 앞서 UART 하드웨어 레지스터를 코드로 작성해서 관리해야 한다. RealViewPB의 UART인 PL011의 datasheet를 확인하여 관련 데이터를 코드로 옮겨 보겠다.

필자가 가져온 데이터는 **Figure 8**은 ARM Developer의 PrimeCell UART(PL011) Technical reference Manual에서 가져온 것이다. 누구든 자유롭게 접근해서 열람이 가능하니 들어가서 한번 찾아보길 바란다. 해당 레지스터를 클릭하면 더 디테일한 정보가 나온다. 시험삼아 UART의 가장 첫 번째 레지스터인 UARTDR의 정보를 확인해보겠다. (ARM Developer Sep 10, 2020)

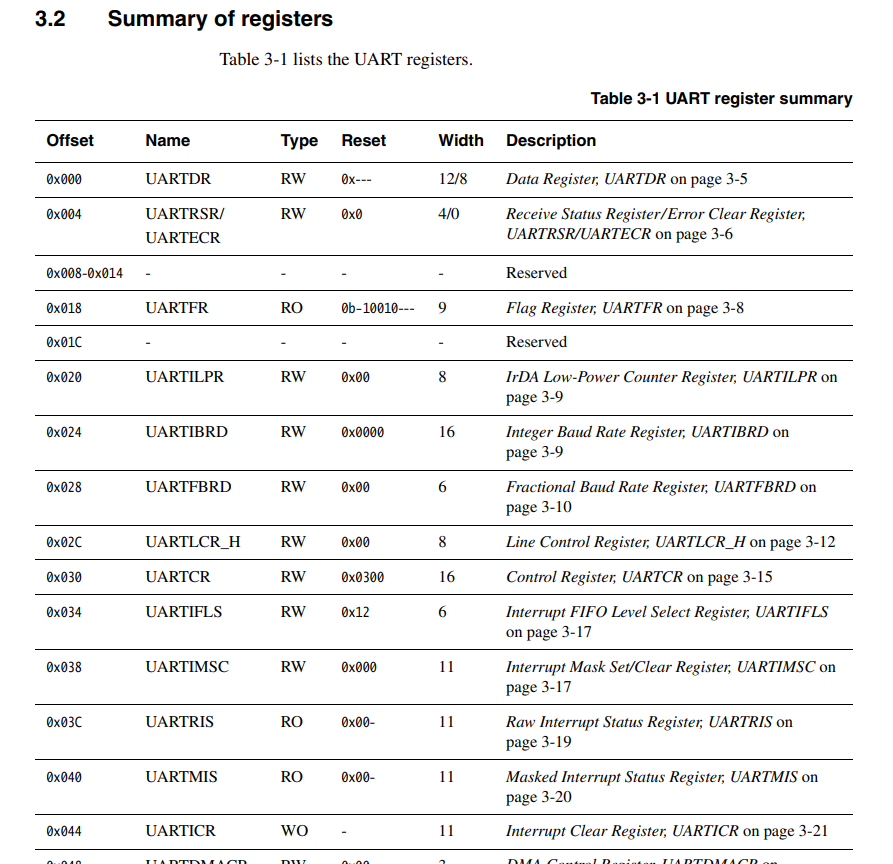
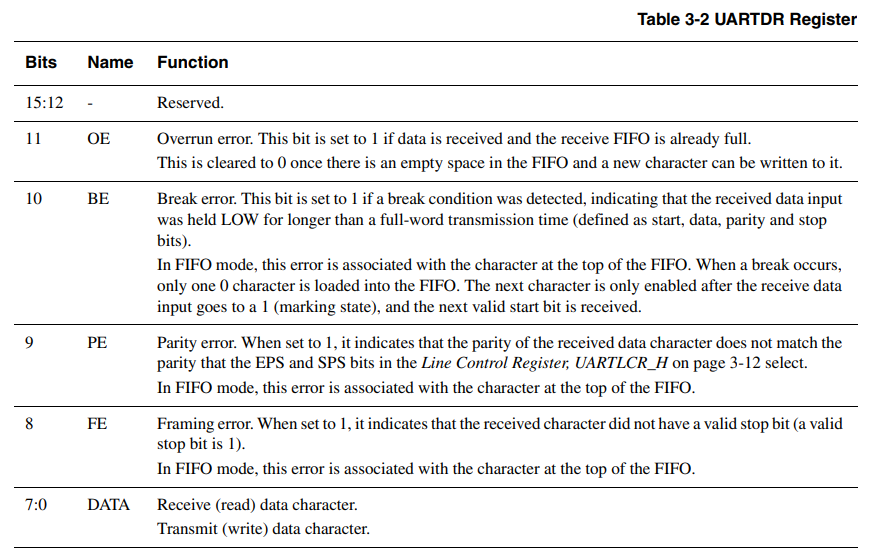
이러한 정보들을 모두 모아서 구조체를 작성하면 된다. 실제로 하드웨어의 레지스터를 코드로 옳기는 일은 단순 반복적인 작업이 주를 이루며 코드의 라인 수도 많아 보통 직접 작성하지는 않고 하드웨어 제조사에서 제공해주는 코드를 쓰는 경우가 많다. 그러나 필자는 학술지도 작성하고 설명도 하는 겸 직접 작성해보겠다. 레지스터와 그 하위 비트들에 대한 설명은 이후 실제로 어떻게 쓰이는지 확인하면서 더 자세하게 기술하겠다. 설명이 부족하다고 느껴지거나 지금 당장 각각의 레지스터와 그 하위 비트들의 쓰임이 궁금하다면 ARM Developer에 들어가 해당 datasheet를 읽고 와도 괜찮다.

Figure 9 UARTDR 레지스터의 각 비트별 정보

Figure 8 UART 레지스터 정보

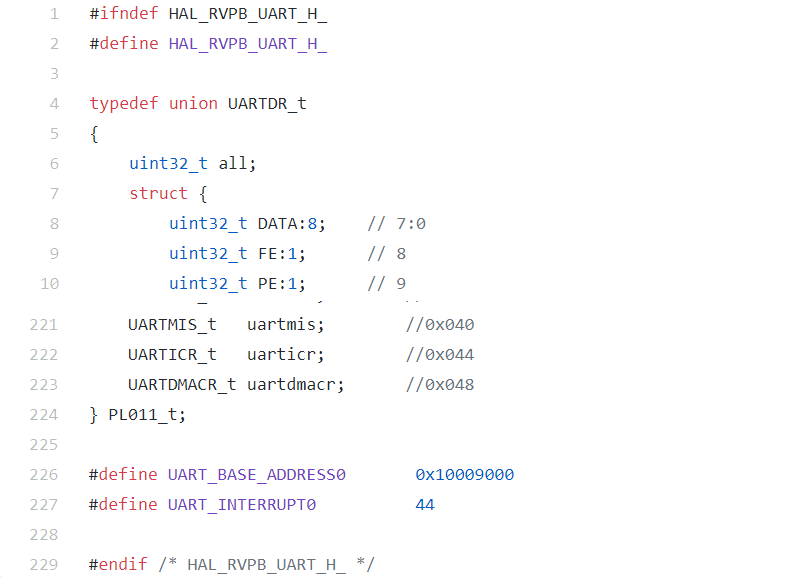
후... 다음의 코드가 UART 레지스터 정보를 datasheet에 따라 옳긴 코드이다. 보면 알겠지만 코드가 무려 229줄이다. 이렇게 번거롭게 레지스터를 구조체로 추상화한 이유는 이후 변수 선언과 특정 비트 접근에 매우 용이하기 때문이다. 이제 남은 일은 UART 하드웨어의 base address(하드웨어에 접근하기 위한 주소)를 할당해두는 것이다. 그렇게 되면 나머지 레지스터는 구조체 각각의 멤버 변수의 이름으로 쉽게 접근이 가능하다. 그 하위 비트 역시 마찬가지이다. 코드 전문은 내주로 달아 둔 출처를 통해 확인하기 바란다. (github Sep 10, 2020)

Figure 10 필자가 피땀 흘려 작성한 UART 레지스터 구조체 정보

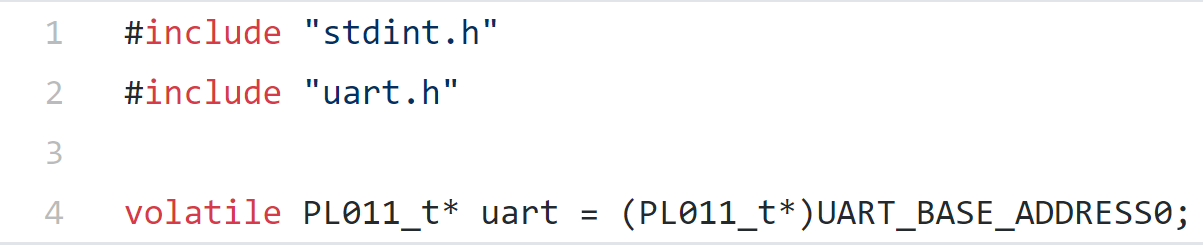
UART 하드웨어를 제어할 변수를 선언해 base address만 등록하면 UART 하드웨어 셋팅은 이로서 끝이 난다. 현재 UART 챕터의 학술지 분량은 2~3 페이지 내외로 극히 짧지만 코드의 분량과 필자가 갈아 넣은 시간은 그러하지 않으니 잠시 숨을 돌리고UART 공용 인터페이스 설계로 넘어가도록 하겠다.

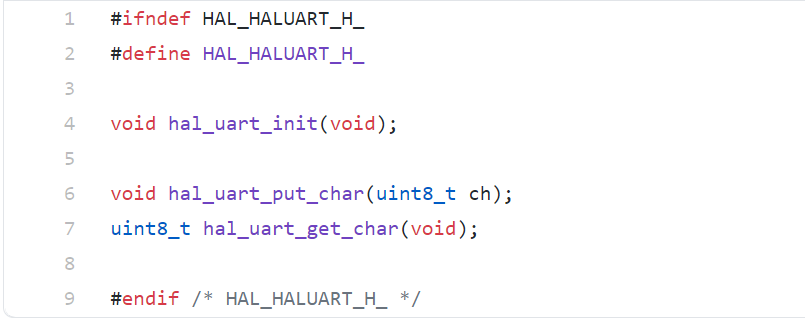
Figure 11 UART 하드웨어 제어에 사용할 uart 변수

표지판, 소화전이(가) 표시된 사진

자동 생성된 설명 PL011은 RealViewPB의 UART이다. 다른 플랫폼은 다른 종류의 UART 하드웨어를 가지고 있을 수 있기 때문에 (그럼 또 위에서 했던 것처럼 하드웨어 셋팅을 해야 한다.) 개별 하드웨어는 각자의 방식으로 동작하되 코드는 공용 인터페이스를 통해서 같은 방식으로 사용할 수 있게 만들 것이다. 일종의 디바이스 드라이버와 같은 역할을 하는 **UART 공용 인터페이스**를 설계하도록 하겠다. 보통 이러한 공용 인터페이스 혹은 API 설계를 Hardware Abstraction Layer(HAL)이라고 부른다.

Figure 12 Hardware Abstraction Layer (HAL)

지금 우리의 예에서 보자면 PL011은 Device 1에 해당하고 설계하고자 하는 것은 중간의 큰 박스인 UART API이다. 이렇게 Device1의 기능을 통해 UART API를 설계하면 Application Layer에서는 내부적인 실행 원리를 몰라도 쉽게 기능을 가져다 쓸 수 있고, 하드웨어가 바뀌더라도 (e.g. RealViewPB의 UART에서 Arduino의 UART로 변경) UART API만 해당 하드웨어를 통해 구현하면 같은 HAL 코드로 동일한 동작을 수행할 수 있게 된다. (Wikipedia Sep 8, 2020)

우선 세 가지 함수를 설계

Reference

1. D. Stepner, N. Rajan and D. Hui, "Embedded application design using a real-time OS," Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361), New Orleans, LA, USA, 1999, pp. 151-156, doi: 10.1109/DAC.1999.781301.
2. David A. Patterson and John L. Hennessy. *컴퓨터 구조 및 설계:하드웨어/소프트웨어 인터페이스*. n.p.: 한티미디어, 2018.
3. "Main Page." *QEMU Wiki*. last modified Jul 9, 2020, accessed Aug 28, 2020, https://wiki.qemu.org/Main\_Page.
4. 이만우. *임베디드 OS개발 프로젝트:ARM 기반 펌웨어/RTOS의 원리와 구조*. n.p.: 인사이트, 2019.
5. "Cortex-A8 Technical Reference Manual." (Sep 10, 2020). *ARM Developer*. last modified Jun 6, 2010, <https://developer.arm.com/documentation/ddi0344/i?lang=en&pdf=DDI0344I_cortex_a8_r3p1_trm.pdf.pdf>.
6. "Using LD, the GNU linker ." (Sep 10, 2020). *ftp.gnu.org*. last modified Nov 7, 1998, <https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_chapter/ld_toc.html#TOC6>.
7. "RealView Platform Baseboard for Cortex-A8 User Guide." (Sep 10, 2020). *ARM Developer*. last modified April 2011, <https://static.docs.arm.com/dui0417/d/DUI0417D_realview_platform_baseboard_for_cortex_a8_ug.pdf>.
8. "CruzerOS." (Sep 10, 2020). *github*. last modified Sep 2, 2020, https://github.com/Cruzer-S/CruzerOS/.
9. "PrimeCell UART (PL011) Technical Reference Manual." (Sep 10, 2020). *ARM Developer*. last modified Dec 18, 2007, <https://developer.arm.com/documentation/ddi0183/g/programmers-model/register-descriptions>.
10. "Hardware abstraction." (Sep 8, 2020). *Wikipedia*. last modified Jun 20, 2020, https://en.wikipedia.org/wiki/Hardware\_abstraction#In\_operating\_systems.

1. 최대한 쉽게 쓰려고 노력했으나 다소 어려운 내용이 있을 수 있습니다. ARM 아키텍처에 대한 지식이 없다면 ARM Cortex-A8 Technical Reference Manual (Chapter 2. Programmer Model)을 읽고 오는 것을 추천 드립니다. [↑](#footnote-ref-1)
2. 마른 수건의 물기를 쥐어짜는 듯한 극한의 최적화를 요구하는 실제 임베디드 시스템에서 이렇게 큰 메모리를 마구잡이로 할당하는 것은 사실상 불가능 합니다. 메모리의 배치 역시 장치의 특성(성능과 용량)에 따라 그 주소를 달리 하는 경우가 많습니다. [↑](#footnote-ref-2)
3. 사실 **Figure 5**는 필자가 LS를 수정해서 .text 영역의 위치를 변환한 다음 실행시킨 결과이다. 실제로는 LS가 vector\_start를 먼저 실행시킨다. 당연히 코드상 vector\_start가 앞서기 때문에 그렇지만 이러한 Linker의 행동방식을 예상하여 프로그램을 작성하는 것은 매우 위험하고 이식성 없는 행동이다. [↑](#footnote-ref-3)