**과목명: 시스템프로그래밍**

**2분반**

**<<Project #1>>**

**서강대학교 [심리학과]**

**[20140424]**

**[문성혁]**

목 차

1. **프로그램 개요**
2. **프로그램 흐름도**
3. **모듈 정의 및 코드 설명**
4. **전역 변수 등 특수 변수 정의**
5. 프로그램 개요

본 프로그램은 시스템 프로그래밍 수업의 첫 번째 프로젝트 결과물로서, 가상 OS SIC/XE를 구현하는데 그 최종적인 목적이 있습니다. 첫 번째 프로젝트의 구체적인 목표는 쉘 환경을 구성하는 것으로서, 다음과 같은 명령어들이 입력 가능하며, 특정 명령어를 정상적으로 수행 또는 에러 출력한 후 idle 상태로 회귀합니다.

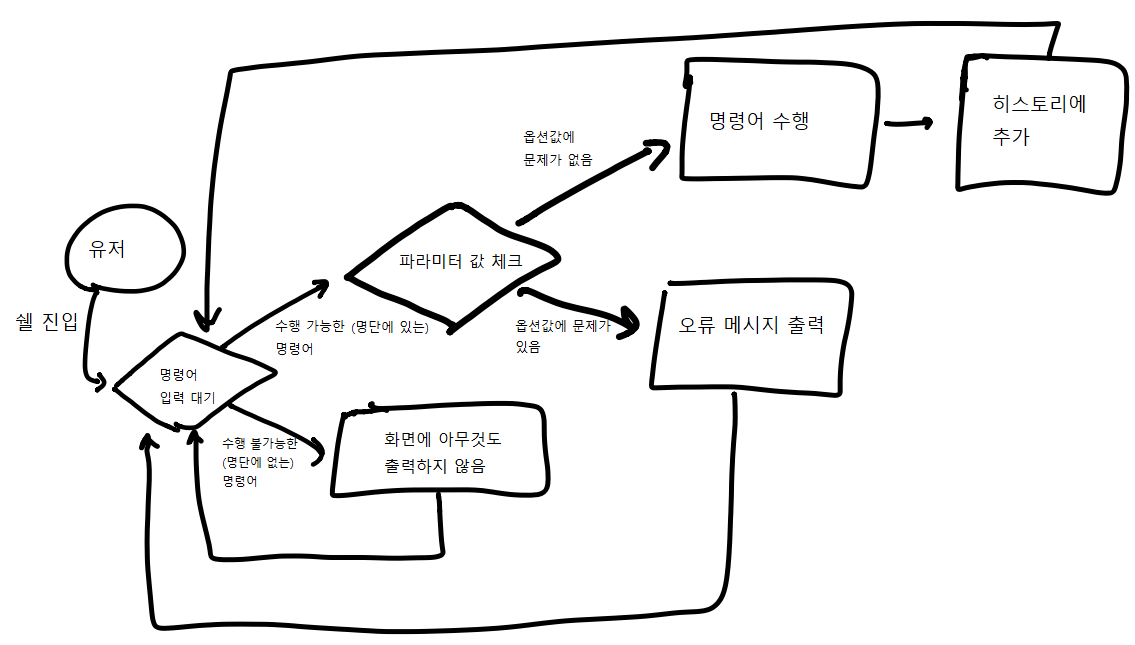
* h[elp]
* d[ir]
* q[uit]
* hi[story]
* du[mp] [start, end]
* e[dit] address, value
* f[ill] start, end, value
* reset
* opcode mnemonic
* opcodelist

위에 언급된 명령어 이외의 입력을 받았다면 해당 명령어를 수행하지 않습니다.

지시된 사항은 아니지만, 다음과 같은 부분을 특히 염두에 두고 코딩을 진행했습니다.

* 각 모듈이 독립적으로 구성될 것 : 각각의 명령어를 별도의 소스파일 및 헤더파일로 분리하였습니다. 이는 확장성과 디버깅의 용이성에 그 목적이 있습니다. 다만 opcodelist는 opcode와 너무나도 밀접한 관련이 있는 바, 하나의 모듈로 간주했습니다.
* 이해하기 쉬울 것 : 최대한 함수 이름만으로 그 목적과 결과물을 예상할 수 있도록 하였습니다. 이는 가독성과 디버깅의 용이성에 그 목적이 있습니다. 함수 이름만으로는 표현이 충분치 않다고 판단한 경우에는 함수 이름 위, 또는 특정 라인 위에 주석을 추가하였습니다.

1. 프로그램 흐름도

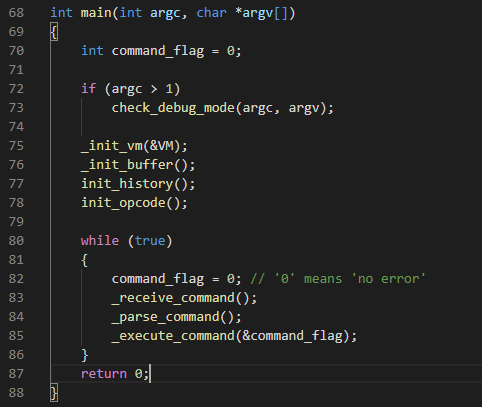


전체적인 흐름도는 위와 같습니다.

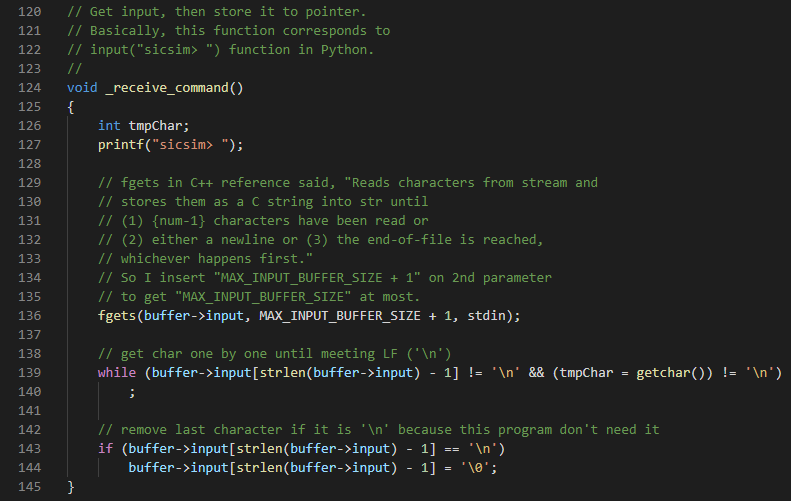
각 모듈을 독립시킨 결과, 해당 모듈 내에서 입력값(파라티머)의 validation check를 진행하게 되었고, 그 결과 각각의 명령어에서 정상 수행 여부를 0(성공) 또는 1(실패) 값으로 리턴하도록 했습니다. 이 결과를 보고 메인 함수에서는 히스토리에 추가할 지 추가할지 않을지를 결정하도록 설계했습니다.

위 흐름도에는 나타나지 않은 두 가지 예외 사항이 있습니다. 첫 번째 예외는 히스토리 명령어(hi[story])를 수행했을 때의 경우입니다. 이 때는 히스토리에 우선 추가한 후 명령어 수행을 진행합니다. 이는 요구조건을 따르기 위함입니다. 두 번째 예외는 종료 명령어(q[uit])를 수행했을 때의 경우입니다. 이 때는 히스토리에 추가하지 않고, 해당 모듈(quit.c) 내에서 프로그램 자체를 exit(0) 함수로 종료시킵니다. History nodes가 프로그램 메모리 상에 존재하므로, quit이 완료된 후 히스토리에 새 노드를 추가할 필요가 없다고 판단했습니다.

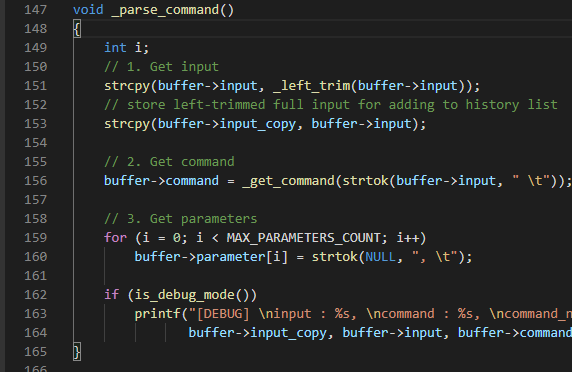
1. 모듈 정의 및 코드 설명
2. 20140424.c (entry point)



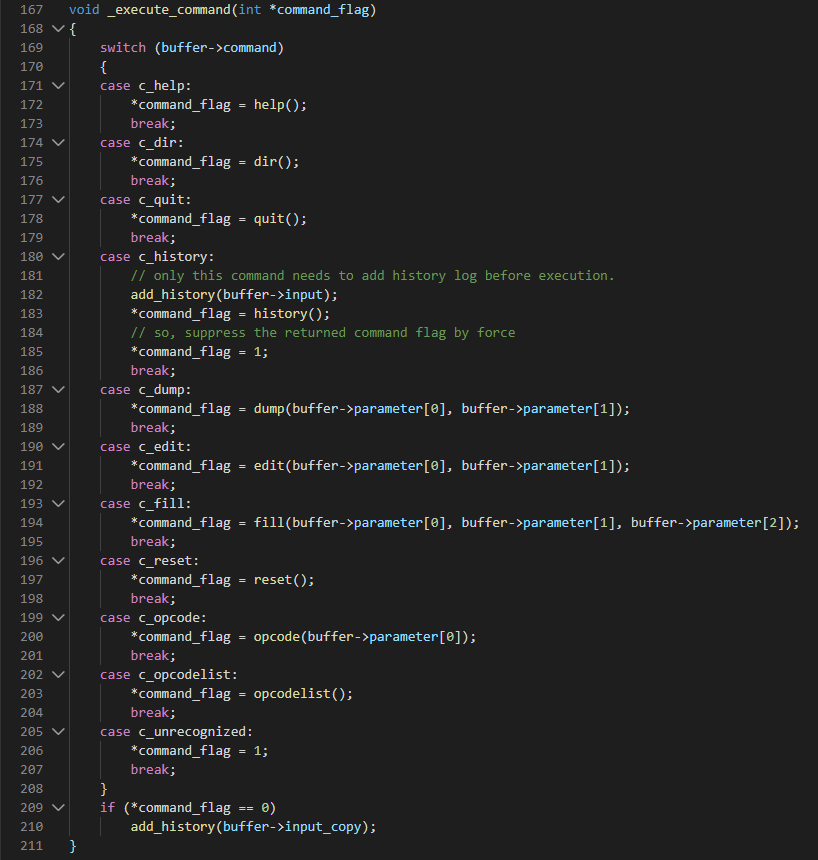
* Line 72 ~ 73 : <디버깅 모드 체크> 디버그 관련 함수는 제가 디버깅을 위해 만든 모듈입니다. debug.h, debug.c로 구성되어 있고, 프로그램 시작 시 ‘-d’ 플래그를 적용할 시 debug mode를 switch on 하여 이후 필요할 때 printf로 적당한 메시지를 출력하도록 했습니다.
* 75 ~ 78 : <초기 메모리 할당, 세팅> 1Mb의 가상메모리를 할당하고, 인풋 버퍼를 받을 구조체인 Buffer에 메모리를 할당하고, 히스토리의 첫 노드를 생성하고, opcode hash table을 만들었습니다.
* 80 ~ 86 : <쉘 가동 및 유지> common\_flag는 특정 모듈이 정상적으로 수행되었는지 판단하는 플래그입니다. 쉘은 특별한 이유 없이는 계속 입력 – 파싱 – 실행의 3단계를 수행합니다.



* 136 : <입력 받기> 최대 MAX\_INPUT\_BUFFER\_SIZE 개의 character를 입력받습니다. fgets(n)은 n-1개의 글자를 입력 받기 때문에 + 1 을 해주었습니다.
* 139 : <잔여 버퍼 정리> 버퍼에 남은 글자가 있다면, 즉 MAX\_INPUT\_BUFFER\_SIZE보다 긴 문자열을 입력 받는다면 받지 못한 만큼은 버립니다.
* 143 ~ 144 : <불필요 문자 제거> 줄바꿈 문자는 필요없다고 판단하여 제거했습니다.

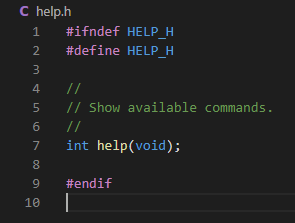


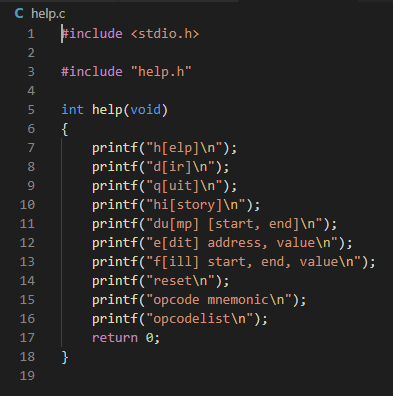
* 151 : <왼쪽 여백 제거> 입력받은 문자열에서 공백이 아닌 문자가 나올때까지 포인터를 우측으로 offset 시킵니다.
* 153 : <히스토리를 위해 문자열 복사> 입력받은 문자열을 input\_copy로 복사합니다. 이는 명령어 정상 수행 이후 히스토리에 노드를 추가하기 위함입니다. Input 자체는 strtok 함수로 인해 많이 훼손됩니다.
* 156 : <명령어 파악> strtok을 이용해 공백 문자로 분리되는 첫 chunk/token을 파악해 이를 통해 명령어를 파악합니다. enum Command를 사용했습니다.
* 159 ~ 160 : <파라미터 저장> strtok을 이용해 나머지 파라미터를 Buffer 구조체에 차례로 담습니다. 파라미터가 없거나 MAX\_PARAMETERS\_COUNT보다 적다면 그 자리에는 NULL이 저장됩니다.



* 169 ~ 208 : <명령어 수행> buffer->command를 통해 각 모듈에 인자를 전달해 실행합니다. 리턴값으로 성공여부를 확인합니다(flag).
* 209 ~ 210 : <히스토리 추가> flag를 확인하고 필요하다면 히스토리에 추가합니다. 단, 예외적으로 Line 180 ~ 186에 나온 것처럼 history 명령어는 명령어 수행 전 히스토리 노드를 추가하므로 flag를 1로 만들어 두 번 추가하지 않도록 예외처리하였습니다.

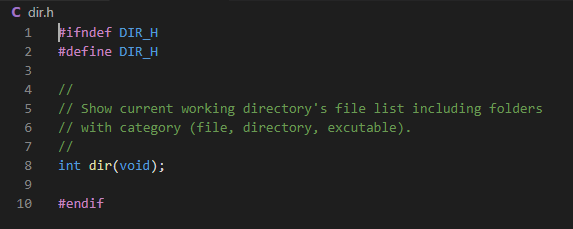
1. help.h, help.c

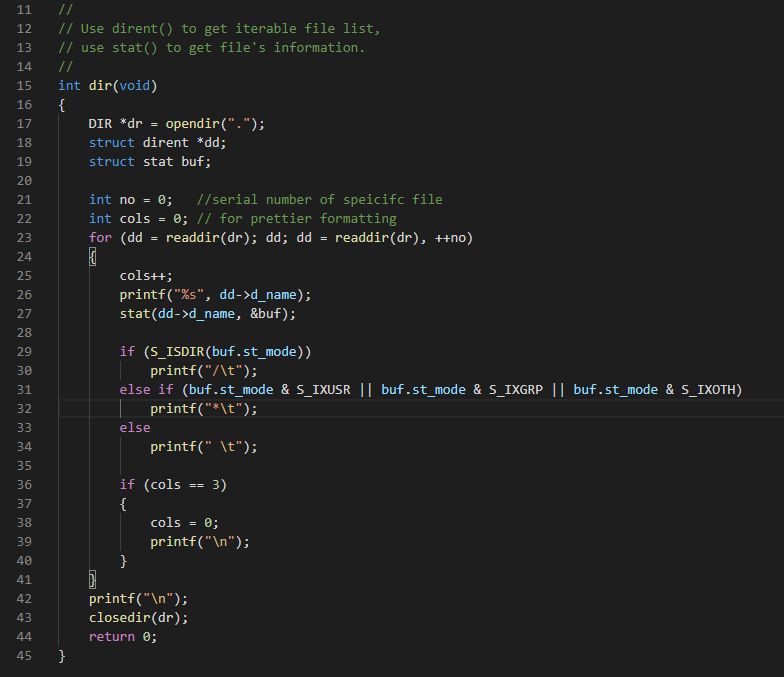




* 7 ~ 16 : <사용가능한 명령어 출력> 간단하게 printf로 구현했습니다.
* 17 : <정상 수행 여부 리턴> 정상적으로 수행하는 것이 자명하므로 0(no error)을 리턴합니다. 이후의 소스파일들에서는 동일한 패턴이 반복되므로 설명을 생략하겠습니다.

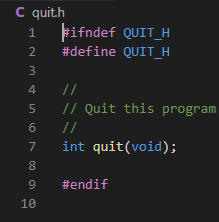
1. dir.h, dir.c

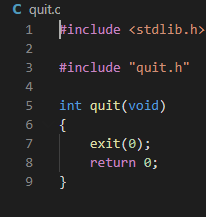




* 23 : <파일 목록 이터레이터 사용> dirent.h의 readdir 명령어를 사용해서, 순차적으로 파일을 하나씩 struct dirent 구조체에 불러왔습니다.
* 27 : <파일 속성 불러오기> sys/stat.h의 stat 명령어로 각 파일의 속성을 가져와 struct stat 구조체에 저장하였습니다.
* 29 ~ 34 : <파일 분류에 따라 다른 출력> 일반파일, 폴더, 실행파일(executable)에 따라 printf에 차이를 두어 식별가능하도록 하였습니다.
* 43 : <메모리 해제> opendir로 불러온 디렉토리 스트림 포인터에 대한 메모리를 해제합니다.

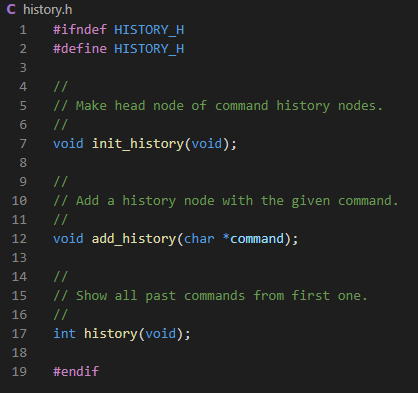
1. quit.h, quit.c

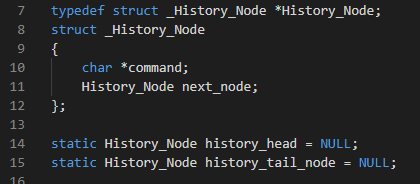




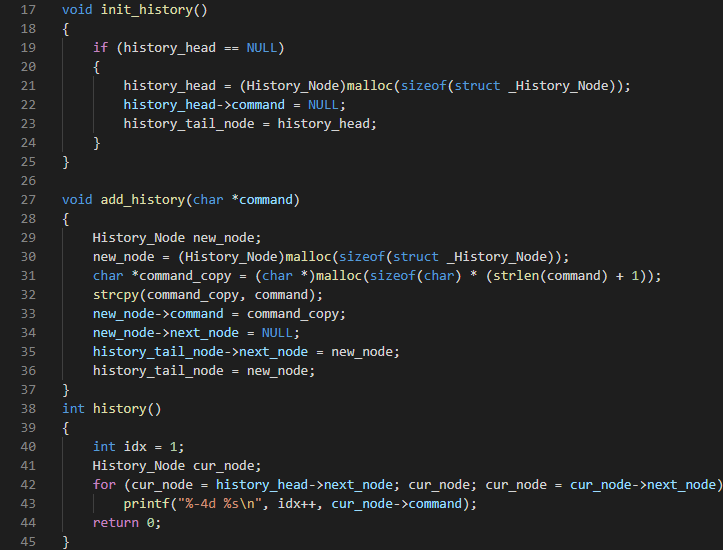
* 7 : <프로그램 종료> stdlib의 exit 명령어로 프로그램을 즉시 종료하도록 하였습니다.

1. history.h, history.c



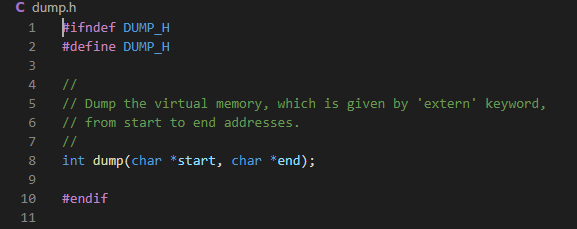


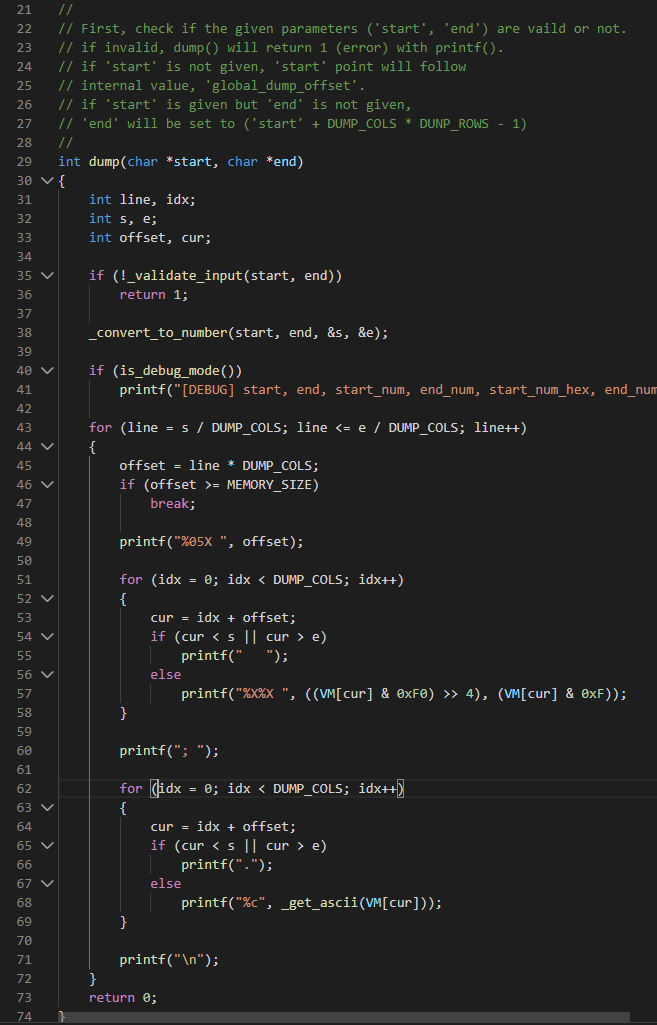
* 7 ~ 15 : <히스토리 노드 선언> 명령어 문자열을 담을 char 포인터와 다음 노드에 대한 포인터를 가지고 있는 단방향 링크드 리스트를 선언했습니다.



* 19 ~ 24 : <헤드 노드, 테일 노드 생성> 기초가 되는 헤드, 테일 노드를 생성했습니다. 테일 노드는 add\_history 연산에 O(1)의 시간 복잡도를 가지기 위해서 추가했습니다.
* 29 ~ 36 : <새 히스토리 노드 추가> 테일 노드를 이용해 O(1)의 시간복잡도로 노드 추가를 구현했습니다. 문자열은 strcpy명령어를 사용하여 복사했습니다. 메모리를 더 사용함에도 불구하고 복사한 이유는 Buffer 객체 하나가 매 명령어마다 재활용 되기 때문입니다. 히스토리를 보존하려면 명령어 문구를 똑같이 복사해야만 했습니다.

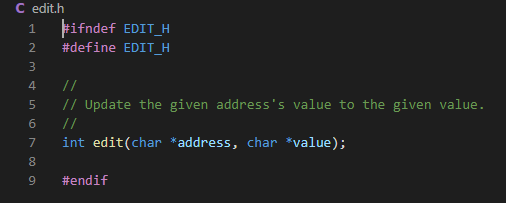
1. dump.h, dump.c

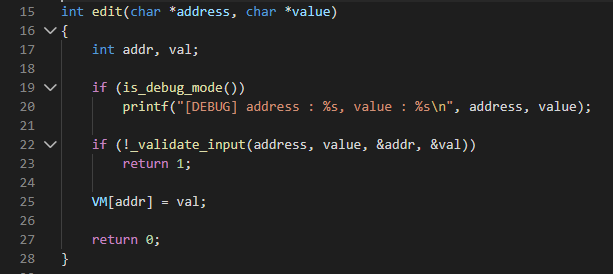




* 35 ~ 36 : <파라미터 유효성 검증> start, end 파라미터에 대해서 유효성 검증을 진행했습니다. dump 명령어는 start, end가 있을 수도 있고 없을 수도 있으며, 있더라도 그 값이 정상적인 hex 값이 아닐 수도 있고, 정상적인 hex 이더라도 표현 가능한 범위를 초과했을 수도 있습니다. 이러한 경우를 모두 고려하여 \_validation\_input 함수로 분리해 검증을 진행했으며, 이에 따라 Invalid parameter, Unreachable address, Wrong range 에러가 발생할 수 있습니다.
* 38 : <문자열 파라미터를 정수로 변환> 위 단계를 거치면 문자열로 입력된 start, end의 값이 해석 가능한 Interger로 변환됩니다. 만약 파라미터가 제시되지 않았다면 적절한 값으로 대체됩니다. 예를 들어 start, end가 모두 입력되지 않았으면 int global\_dump\_offset에 따라 마지막으로 “dump” (with no parameter) 했던 위치로부터 다음 1번째, 다음 160번째 메모리 인덱스를 s, e에 각각 대입하고, start 값만 제시되었다면 그로부터 159번째의 메모리 인덱스를 e에 대입하는 식입니다.
* 43 ~ 72 : <메모리 값 출력> 16 Columns으로 memory dump를 진행합니다. DUMP\_COLS는 extern 키워드로 받아오고 있으며, 실제 값은 메인 함수 (20140424.c)에서 설정합니다. 특히 Line 57에서, 비트 연산을 활용해(shift, mask) half byte 2개로 각 메모리 인덱스의 내용을 표현하는 것을 확인할 수 있습니다.

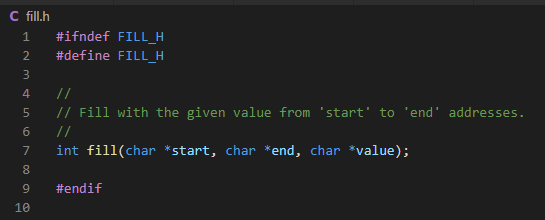
1. edit.h, edit.c

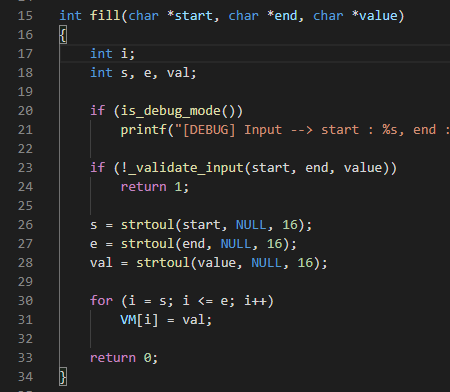




* 22 ~ 23 : <파라미터 유효성 검증> dump 함수와 크게 다르지 않습니다. \_validation\_input 함수로 Insufficient parameter, Invalid parameter, Unreachable address, Unwritable address 에러를 검출g합니다.
* 25 : <메모리 값 수정> 값을 수정합니다. 이 때 사용되는 addr, value는 유효성 검증과 동시에 값을 가져오도록 구현하였습니다. dump와 다르게 address, value가 필수 요소이기 때문입니다.

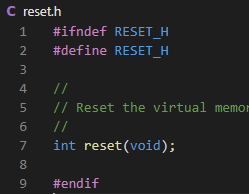
1. fill.h, fill.c

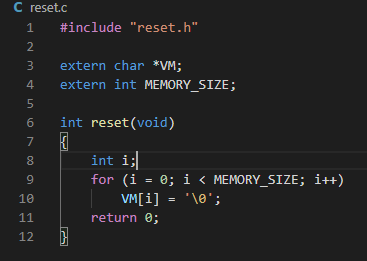




* 23 ~ 24 : <파라미터 유효성 검증> edit 함수와 크게 다르지 않습니다. Insuffcient parameter, Invalid parameter, Unreachable address, Wrong range, Unwritable address 이상 5개 오류에 대해서 검증을 진행합니다.
* 26 ~ 28 : <문자열 파라미터를 정수로 변환> strtoul 함수를 이용해 문자열을 16진수로 변환합니다. 이미 위에서 검증이 끝났으므로 완벽하게 변환될 것이 보장됩니다.
* 30 ~ 31 : <특정 범위의 메모리 값 수정> s, e, val을 이용해 특정 범위의 메모리 인덱스의 값을 하나씩 변경합니다.

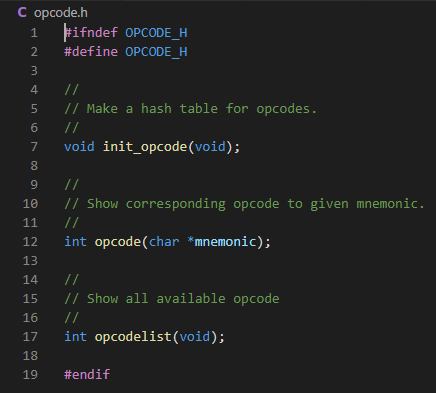
1. reset.h, reset.c

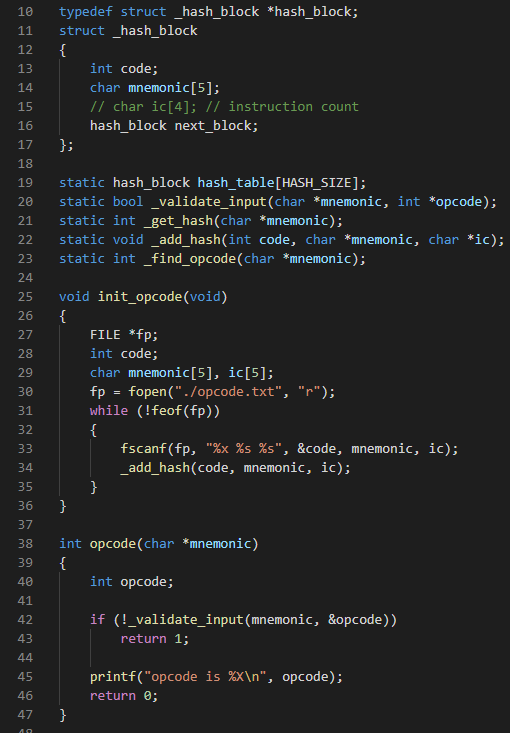




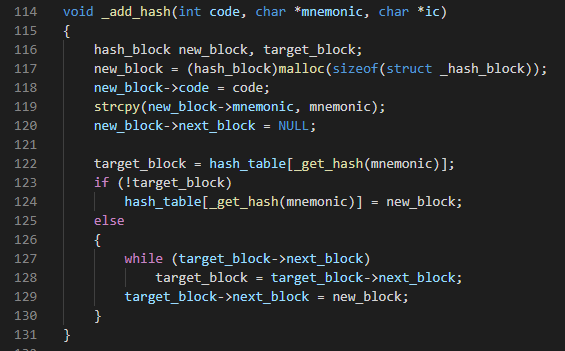
* 3 ~ 4 : <필요 변수 호출> 메모리에 접근할 때 사용하는 char 포인터와 메모리 사이즈를 extern 키워드로 호출하고 있습니다. 이 두 값은 메인 함수 (20140424.c)에서 할당되어, global 변수의 역할을 하고 있습니다.
* 9 ~ 10 : <메모리 초기화> 전체 메모리 범위에 대해 모든 값을 NULL(‘\0’)로 합니다.

1. opcode.h, opcode.c (opcodelist를 포함합니다)

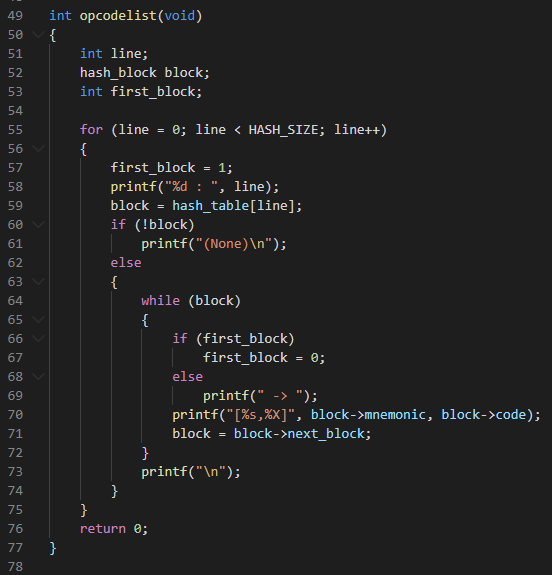




* 10 ~ 11 : <해쉬 엘리먼트(블록) 구조체 선언> 코드, 니모닉 문자열 배열, 다음 노드 (중복 해쉬 값을 위한 단방향 링크드 리스트)로 이루어진 해쉬 블록 구조체를 선언합니다.
* 30 ~ 35 : <해쉬 블록 추가> 파일을 읽어와 해쉬 블록을 만들어 미리 선언된 hash\_table에 하나씩 추가합니다. 이 함수는 메인 함수 (20140424.c)에서 최초 1회 실행됩니다.
* 42 ~ 43 : <파라미터 유효성 검증> Insufficient parameter, No corresponding opcode to the given value 에러에 대하여 검증을 진행합니다. 또한 동시에 검증이 성공적으로 수행 되었다면 해당 mnemonic을 opcode로 변환합니다.



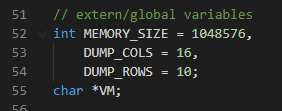
* 116 ~ 120 : <해쉬 블록 생성> 해쉬에 추가하는 것은 history와 마찬가지로 문자열 값을 복사하는 방식을 사용했습니다.
* 122 ~ 130 : <해쉬 블록을 해쉬 테이블에 추가> hash\_table에서, 해당 해쉬 값에 노드가 하나도 없다면 바로 추가하고 하나라도 존재하면 해당 인덱스의 링크드 리스트를 따라가 tail에 추가하도록 했습니다.



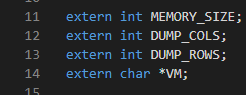
* 55 ~ 75 : <해쉬 테이블 출력> 해쉬 테이블의 인덱스를 하나씩 돌며, 링크드 리스트 유무 확인 및 존재 시 순서대로 출력 후 줄바꿈 하는 과정을 거쳐 해쉬 테이블 출력을 구현했습니다.

1. 전역 변수 등 특수 변수 정의

메인 함수에 있는 MEMORY\_SIZE, DUMP\_COLS, DUMP\_ROWS, VM 등 4개 변수는 다른 모듈에서도 사용하는 전역 변수입니다. 타 모듈에서는 extern 키워드로 호출해 사용하고 있습니다. 다음과 같은 식입니다.



(dump.c의 예시)



debug.c에서는 외부에서 직접 접근하지 못하게 막기 위해 file scope의 static variable을 사용했습니다.



opcode.c에서는 HASH\_SIZE를 define 키워드로 설정해, file scope의 global 변수라고 말할 수 있습니다.



이상으로 프로그램 보고서를 마칩니다.