명령 언어(Command Language)

명령 언어는 링크 프로세스에 대한 명시적인 제어를 제공한다. 링커의 입력 파일들과 출력 사이의 완전한 맵핑 스펙을 지원한다. 이것은 다음과 같은 것들을 제어한다.

- 입력 파일
- 파일 포멧
- 출력 파일 레이아웃
- 섹션들 어드레스
- 공통 블럭들의 위치

링커에게 `-T' 옵션을 통해서 명시적으로 또는 일반 파일처럼 암묵적으로 명령 파일(링커 스크립트라고 하기도 한다)을 제공할 수 있다. 일반적으로 `-T' 옵션을 사용해야만 한다. 묵시적인 링커 스크립트는 디폴트 링커 스크립트를 대체한다기 보다는 확장하고자 할 때만 사용되어야 한다; 전형적으로 묵시적 링커 스크립트는 INPUT 또는 GROUP 명령들만으로 이루어져 있을 것이다.

링커가 지원되는 오브젝트 또는 아카이브 포멧으로 인식할 수 없거나 또는 링커 스크립트로 인식할 수 없는 파일을 오픈하면 이것은 에러를 보고한다.

링커 스크립트(Linker Scripts)

ld 명령 언어는 문장(statement)들의 모임이다; 어떤 것들은 특별한 옵션을 설정하는 단순한 키워드들이고 어떤 것들은 입력 파일들이나 이름 출력 파일들을 선택하고 그룹핑하는 데 사용된다; 그리고 두 문장 타입들이 기초적이며 링크 프로세스에 널치 영향을 미친다.

ld 명령 언어의 가장 기초적인 명령은 SECTIONS 명령(see section 출력 섹션 지정(Specifying Output Sections))이다. 각 의미 있는 명령 스크립트는 SECTIONS 명령을 가져야 한다: 이것은 여러 등급의 자세한 내용들을 가진, 출력 파일의 레이아웃의 "그림(picture)"을 지정한다. 어떤 다른 명령도 모든 경우에 반드시 필요한 것은 아니다.

MEMORY 명령은 타겟 아키텍쳐에서 사용 가능한 메모리를 기술해서 SECTIONS 명령을 보완한다. 이 명령은 옵션이다; MEMORY 명령을 사용하지 않으면 ld는 모든 출력에 대해서 연속된 블럭으로 충분한 메모리가 사용가능할 것이라고 추정한다. See section 메모리 레이아웃(Memory Layout).

링커 스크립트에서 주석을 C에서처럼 넣을 수 있다; `/*'과 `*/' 안에 묶으면 된다. C에서 처럼 주석들은 문법적으로 공백과 동일하다.

Expressions

많은 유용한 명령들은 산술 표현식들을 포함한다. 명령 언어에 있는 표현식에 대한 문법은 다음과 같은 특성을 가지면서 C 표현식의 문법과 동일하다:

- 모든 표현식은 정수로 평가되고 "long" 이나 "unsigned long" 타입이다.
- 모든 상수들은 정수다.
- C 산술 연산자들 모두가 제공된다.
- 여러분은 전역 변수들을 참조하고, 정의하고, 생성할 수 있다.
- 특수 목적의 내장 함수들을 호출할 수 있다.

정수(Integers)

8진수 정수는 `0' 뒤에 따라오고 0개 이상의 8진수 디지트 (`01234567')들로 이루어진다.

```
_as_octal = 0157255;
```

10진수 정수는 0이 아닌 디지트로 시작하고 0개 이상의 디지트들 (`0123456789')로 이루어진다.

```
_as_decimal = 57005;
```

16진수 정수는 `0x'나 `0X' 뒤에 하나 이상의 16진수 디지트들 `0123456789abcdefABCDEF'로 이루어진다.

```
_as_hex = 0xdead;
```

음의 정수를 쓰기 위해서 접두 연산자 `-' (see section 연산자(Operators)) 를 사용한다.

```
_as_neg = -57005;
```

이와 아울러 K 와 M가 상수를 각각 배씩 하는 데 사용될 수 있다. 예를 들어서 다음은 모두 동일한 양을 가리킨다:

```
_fourk_1 = 4K;
_fourk_2 = 4096;
_fourk_3 = 0x1000;
```

심벌 이름(Symbol Names)

따옴표로 묶지 않으면 심벌 이름들은 문자, 밑줄, 또는 점으로 시작하고 임의의 문자들, 밑줄, 디지트, 점 그리고 하이픈을 담을 수 있다. 따옴표로 묶지 않은 심벌 이름들은 반드시 임의의 키워드와 충돌하면 안된다. 이상한 문자들을 담고 있거나 키워드와 동일한 이름을 담고 있는 심벌을, 심벌 이름을 겹따옴표로 싸서, 지정할 수 있다:

```
"SECTION" = 9;
"with a space" = "also with a space" + 10;
```

심벌들은 많은 비-알파벳 문자들을 담을 수 있기 때문에 심벌들을 공백들로 구분하는 것이 가장 안전하다. 예를 들어서 `A-B'는 하나의 심벌이다. 반면에 `A - B'는 빼기를 포함하고 있는 표현식이다.

위치 카운터(The Location Counter)

특수한 링커 변수 *dot* `.' 는 항상 현재 출력 위치 카운터를 담고 있다. .는 항상 출력 섹션에 있는 위치를 참조하기 때문에 이것은 항상 SECTIONS 명령안에 있는 표현식에 나타나야 한다. . 심벌은 일반 심벌이 표현식에서 허용된 위치라면 어디든지 나타날 수 있지만, 이것의 할당(assignment)은 부작용을 가진다. 어떤 값을 . 심벌에 할당하는 것은 위치 카운터가 이동되도록 할 것이다. 이것은 출력 섹션에 구멍(hole)을 생성하는 데 사용될 수 있다. 위치 카운터는 뒤쪽으로 이동되어서는 안된다.

```
SECTIONS
{
  output :
  {
  file1(.text)
  . = . + 1000;
  file2(.text)
  . += 1000;
```

```
  file3(.text)
  } = 0x1234;
}
```

이전 예제에서 `file1`은 출력 섹션의 처음에 위치한다. 그 뒤에 1000바이트의 틈새가 있다. 그리고 `file2` 가 나타난다. 그리고 그 뒤에 또 1000바이트의 틈새가 있고 그 뒤에 `file3`가 로드된다. `` `=0x1234' ``라고 표기하는 것은 어떤 데이터가 그 틈새에 기록되어야 할 것인가를 지정한다 (see section 옵션인 섹션 속성(Optional Section Attributes)).

@vfill

연산자(Operators)

링커는 다음과 같이 표준 바인딩과 우선순위 레벨들과 함께, 수식 연산자들의 표준 C 집합을 인식한다: { @obeylines@parskip=0pt@parindent=0pt @dag@quad Prefix operators. @ddag@quad See section 할당: 심벌 정의(Assignment: Defining Symbols). }

평가(Evaluation)

링커는 표현식들에 대해서 "게으른 평가(lazy evaluation)"를 사용한다; 이것은 절대적으로 필요할 때만 표현식을 계산한다. 링커는 임의의 링크를 하기 위해서 시작 주소의 값과 메모리 영역들의 길이를 필요로 한다; 이런 값들은 링커가 명령 파일을 읽을 때 가능한 한 빨리 계산된다. 그러나 다른 값들(예를 들어 심벌 값들)은 저장소 할당(stroage allocation)이 이루어지기 전까지는 알려지거나 않거나 필요하지 않다. 그런 값들은 나중에 평가된다. 다른 정보(출력 섹션들의 크기등과 같은)가 심벌 할당 표현식에서 사용될 수 있을 때에.

할당: 심벌 정의(Assignment: Defining Symbols)

여러분은 C 할당 연산자들 중 하나를 사용해서, 글로벌 심벌들을 생성할 수 있고 글로벌 심벌들에 값들(주소들)을 할당할 수 있다:

```
symbol = expression ;
symbol &= expression ;
symbol += expression ;
symbol -= expression ;
symbol *= expression ;
symbol /= expression ;
```

두 개의 것들이 `ld` 표현식들에서 다른 연산자들을 구분한다.

- 할당은 표현식의 루트(root)에서만 사용될 수 있다; `` `a=b+3;' `` 는 허용되지만, `` `a+b=3;' `` 는 에러이다.
- 여러분은 마지막 세미콜론 (";")을 할당 문장의 마지막에 놓아야 한다.

할당 문장들은 다음과 같이 보일 수 있다:

- `ld` 스크립트에서 as commands in their own right in an `ld` script; or
- as independent statements within a `SECTIONS` command; or
- as part of the contents of a section definition in a `SECTIONS` command.

첫번째 두 경우들은 효력면에서 동일하다--둘다 절대 주소로 심벌을 정의한다. 두번째 경우는 주소가 특정한 섹션 (see section 출력 섹션 지정(Specifying Output Sections)) 에 상대적인 심벌을 정의한다.

링커 표현식이 평가되고 변수에 할당될 때 이것은 절대 또는 재배치 가능한 타입을 받는다. 절대 표현식 타입은 심벌이 출력 파일에 있게 될 값을 가지는 타입이다; 상대적 표현식 타입은 값이 섹션 베이스로부터의 고정된 오프셋으로 표현된다.

표현식의 타입은 스크립트 파일에서의 위치에 의해서 제어된다. 섹션 정의 안에서 할당된 심벌은 섹션의 베이스에 상대적으로 생성된다; 다른 장소에서 할당된 심벌은 절대적인 심벌로 생성된다. 섹션 정의안에서 생성된 심벌은 섹션의 베이스에 상대적이기 때문에 재배치 가능한 출력이 요구되면 이것은 재배치 가능하게 남을 것이다. 심벌은 절대 할당 함수 ABSOLUTE 를 사용하여 섹션 정의내에서 할당될 때에도, 절대 값으로 생성될 수 있다. 예를 들어서 주소가 .data이라는 이름의 출력 섹션의 마지막 바이트인 절대 심벌을 생성하기 위해서는:

```
SECTIONS{ ...
  .data :
    {
      *(.data)
      _edata = ABSOLUTE(.) ;
    }
... }
```

링커는 소스 표현식에 있는 모든 용어들이 알려질 때까지 할당의 평가를 꺼두려고(put off) 시도한다 (see section 평가(Evaluation)). 예를 들어서 섹션들의 크기는 할당 이후까지는 알려질 수 없다. 그래서 이런 것에 종속적인 할당은 할당 이후까지 수행되지 않는다. 위치 카운터 *dot*, `.'에 종속적인 것과 같은 어떤 표현식들은 반드시 할당 도중에 평가되어야 한다. 표현식의 결과가 필요하다면 그러나 그 값이 사용 불가능이라면 에러가 발생한다. 예를 들어서 다음과 같은 스크립트는

```
SECTIONS { ...
  text 9+this_isnt_constant :
    { ...
    }
... }
```

에러 메시지 "Non constant expression for initial address"를 발생시킨다.

어떤 경우 링커 스크립트가 심벌이 참조될 때만, 그리고 이것이 링크에 포함된 임의의 오브젝트에서도 정의되지 않았을 경우에만, 심벌을 정의하는 것이 바람직하다. 예를 들어서 전통적인 링커들은 심벌 `etext'를 정의한다. 그러나 ANSI C는 사용자가 `etext'를 에러 없이 함수로써 사용할 수 있기를 요구한다. PROVIDE 키워드는 `etext'과 같은 심벌을, 이것이 참조되지만 정의되지 않은 경우에만, 정의하는 데 사용될 수 있다. 문법은 PROVIDE(*symbol = expression*) 이다.

산술연산 함수(Arithmetic Functions)

명령 언어는 링크 스크립트 표현식들 안에서 사용되는 다수의 내장 함수들을 포함한다.

ABSOLUTE(*exp*)
: 이것은 표현식 *exp*의 절대값 (비-음수와는 반대로 재배치-불가)을 리턴한다. 이것은 심벌 값들이 일반적으로 섹션-상대적인 섹션 정의안에서 어떤 심벌에 절대값을 할당하는 데 주로 유용하다.

ADDR(*section*)
: *section*이라는 이름의 절대 주소를 리턴한다. 여러분의 스크립트는 반드시 이전에 그 섹션의 위치를 정의해야 한다. 다음 예제에서 symbol_1와 symbol_2는 동일한 값들로 할당된 것이다:

```
SECTIONS{ ...
  .output1 :
    {
    start_of_output_1 = ABSOLUTE(.);
    ...
    }
```

```
    .output :
      {
      symbol_1 = ADDR(.output1);
      symbol_2 = start_of_output_1;
      }
... }
```

LOADADDR(*section*)

*section*라는 이름의 절대 로딩 주소(absolute load address)를 리턴한다. 이것은 일반적으로 ADDR와 동일하다. 그러나 이것은 AT라는 키워드가 섹션 정의에서 사용된다면 서로 다를 수 있다 (see section 옵션인 섹션 속성(Optional Section Attributes)).

ALIGN(*exp*)

다음 *exp* 바운더리에 정렬된 현재 위치 카운터 (.)의 결과를 리턴한다. *exp*는 반드시 그것의 값이 2의 몇제곱인 표현식이어야 한다. 이것은

```
(. + exp - 1) & ~(exp - 1)
```

과 동일하다. ALIGN은 위치 카운터의 값을 변경하지 않는다---이것은 단지 그것에 대해서 산술 연산만을 수행한다. 예를 들어서 출력 .data 섹션을 직전 섹션 뒤의 다음 0x2000 바이트 경계에 정렬하기 위해서, 그리고 그 섹션에 있는 어떤 변수를 입력 섹션들 뒤의 다음 0x8000 경계에 정렬하기 위해서는:

```
SECTIONS{ ...
  .data ALIGN(0x2000): {
    *(.data)
    variable = ALIGN(0x8000);
  }
... }
```

예제의 ALIGN 첫번째 사용은 어떤 섹션의 위치를 지정한다. 왜냐면 이것은 섹션 정의의 옵션인 *start* 속성으로써 사용되었기 때문이다 (see section 옵션인 섹션 속성(Optional Section Attributes)). 두번째 사용은 단순하게 어떤 변수의 값을 정의한다. 내장 NEXT는 ALIGN에 아주 밀접하게 연결되어 있다.

DEFINED(*symbol*)

링커의 글로벌 심벌 테이블안에 *symbol*이 있고 정의되어 있으면 1을 리턴한다. 그렇지 않으면 0을 리턴한다. 여러분은 이 함수를 사용해서 심벌들에 대한 디폴트 값들을 제공할 수 있다. 예를 들어서 다음 명령-파일 조각은 글로벌 심벌 begin를 .text 섹션에 있는 첫번째 위치에 설정하는 방법을 보여준다---그러나 begin이라고 불리는 어떤 심벌이 이미 존재한다면 그 값은 보존된다:

```
SECTIONS{ ...
  .text : {
    begin = DEFINED(begin) ? begin : . ;
    ...
  }
... }
```

NEXT(*exp*)

*exp*의 배수인 할당되지 않은 주소를 리턴한다. 이 함수는 ALIGN(*exp*)에 밀접하게 연결되어 있다; 여러분이 비연속(discontinuous) 메모리를 출력 파일을 위해 정의하기 위해서 MEMORY 명령을 사용하지 않는다면 두 함수들은 서로 동일하다.

SIZEOF(*section*)

*section*라는 이름을 가진 섹션의 크기를, 그 섹션이 할당되었다면, 바이트 단위로 리턴한다. 다음 예제에서 symbol_1과 symbol_2는 동일한 값들로 할당된다:

```
SECTIONS{ ...
  .output {
    .start = . ;
    ...
    .end = . ;
    }
```

```
  symbol_1 = .end - .start ;
  symbol_2 = SIZEOF(.output);
... }
```

SIZEOF_HEADERS
sizeof_headers
: 출력 파일의 헤더들의 크기를 바이트 단위로 리턴한다. 여러분은, 페이징을 쉽게 하기로 선택했다면, 이런 숫자를 첫번째 섹션의 시작 주소로써 사용할 수 있다.

MAX(*exp1*, *exp2*)
: *exp1*과 *exp2*의 최대값을 리턴한다.

MIN(*exp1*, *exp2*)
: *exp1*과 *exp2*의 최소값을 리턴한다.

세미콜론(Semicolons)

세미콜론 (";")은 다음과 같은 장소에서 필요하다. 모든 다른 장소에서 그들은 심미적 이유 (aesthetic reasons)로 나타날 수 있지만 그렇지 않다면 무시된다.

Assignment
: 세미콜론은 할당 표현식 뒤에 반드시 나타나야 한다. See section 할당: 심벌 정의 (Assignment: Defining Symbols)

PHDRS
: 세미콜론은 PHDRS 문장의 마지막에 나타나야 한다. See section ELF 프로그램 헤더(ELF Program Headers)

메모리 레이아웃(Memory Layout)

링커의 디폴트 설정은 모든 사용 가능한 메모리의 할당을 허용한다. 여러분은 이 설정을 MEMORY 명령을 사용해서 오버라이드할 수 있다. MEMORY 명령은 타겟내의 메모리 블럭들의 크기와 위치를 기술한다. 이것을 주의깊게 사용함으로써 여러분은 어떤 메모리 영역들이 링커에 의해서 사용될 수 있는지 그리고 어떤 메모리 영역들을 반드시 피해야 하는지 기술할 수 있다. 링커는 사용가능한 영역들에 맞추기 위해서 섹션들을 서로 섞지 않지만 요구된 섹션들을 정확한 영역들로 이동시키기는 한다. 그리고 그 영역들이 가득 차게 되면 에러들을 발생한다.

명령 파일은 많아야 한 번 MEMORY 명령을 사용할 수 있다; 그러나 여러분은 원하는 만큼 그 안에 있는 메모리 블럭들을 정의할 수 있다. 문법은 다음과 같다:

```
MEMORY
  {
    name (attr) : ORIGIN = origin, LENGTH = len
    ...
  }
```

name
: 이것은 링커에 의해서 영역에 대한 참조를 하기 위해서 내부적으로 사용되는 이름이다. 임의의 심벌 이름도 사용될 수 있다. 영역 이름들은 분리된 이름 공간에서 저장되고 심벌들, 파일 이름들 또는 섹션 이름들과 충돌하지 않을 것이다. 서로 다른 이름들을 사용해서 여러 영역들을 지정하자.

(*attr*)
: 이것은 특정한 메모리를 링커 스크립트에 없는 섹션을 놓기 위해서 사용할 것인가 말것인가를 지정하는, 옵션인 속성들의 리스트이다. 유효한 속성 리스트는 섹션 속성들과 일치하는 "ALIRWX" 문자들로 구성되어야 한다. 속성 리스트를 생략하면 여러분은 이것을 둘러싸는 괄호들도 생략할 수 있다. 현재 지원되는 속성들은 다음과 같다:

```
`Letter'
     Section Attribute
```

`R'
읽기-전용 섹션
`W'
읽기/쓰기 섹션
`X'
실행 코드를 담고 있는 섹션
`A'
할당된 섹션
`I'
초기화된 섹션
`L'
I와 동일.
`!'
다음에 오는 속성 의미의 반대.

origin
이것은 물리적 메모리에 있는 영역의 시작 주소이다. 이것은 메모리 할당이 이루어지기 전에 상수로 반드시 평가되어야 하는 표현식이다. 키워드 ORIGIN는 org나 o로 약식 표현될 수 있다 (그러나 예를 들어 `ORG'은 아니다).

len
이것은 해당 영역(표현식)의 바이트 단위 크기이다. 키워드 LENGTH는 len나 l로 약식 표현될 수 있다.

예를 들어서 메모리가 할당---하나는 0부터 시작해서 256 KB를, 다른 것은 0x40000000 부터 시작해서 4 MB를--을 위해서 사용 가능한 두 영역들을 사용한다고 지정하려고 한다고 가정하자. rom 메모리 영역은 읽기-전용이거나 코드를 담고 있는 명시적인 메모리 레지스터 없이 모든 섹션들을 획득할 것이지만 ram 메모리 영역은 그 섹션들을 획득할 것이다.

```
MEMORY
  {
  rom (rx)  : ORIGIN = 0, LENGTH = 256K
  ram (!rx) : org = 0x40000000, l = 4M
  }
```

일단 *mem*라는 이름의 메모리 영역을 정의했다면, SECTIONS 명령에서 `>*mem*'로 끝나는 명령을 사용함으로써 특정한 출력 섹션을 다른 곳으로 보낼수 있다 (see section 옵션인 섹션 속성(Optional Section Attributes)). 어떤 영역에 보내져서 결합된 출력 섹션들이 그 영역에 대해서 너무 크다면 링커는 에러 메시지를 출력할 것이다.

출력 섹션 지정(Specifying Output Sections)

SECTIONS 명령은 어디에 입력 섹션들이 출력 섹션들로 정확하게 놓일 것인가와 출력 파일에서의 그들의 순서, 그리고 그들이 할당된 출력 섹션들이 무엇인가를 제어한다.

여러분은 많아야 한번 SECTIONS 명령을 스크립트 파일에서 사용할 수 있다. 그러나 그 안에 여러분이 원하는 만큼의 문장들을 넣을 수 있다. SECTIONS 명령 안에 있는 문장들은 다음과 같은 것들 중의 하나가 될 수 있다:

- 엔트리 포인트를 정의;
- 심벌에 값을 할당;
- 이름이 있는 출력 섹션의 위치, 그리고 어떤 입력 섹션들이 그 안으로 갈 것인가를 기술.

첫번째 두 작업들---엔트리 포인트를 정의하고 심벌들을 정의하는 것---을 SECTIONS 명령 바깥에서 사용할 수 있다: see section 엔트리 포인트(The Entry Point) 그리고 section 할당: 심벌 정의(Assignment: Defining Symbols). 그들은 스크립트를 읽기 쉽도록 하는 편의를 위해서 여기에 허용

되었다. 그래서 그 심벌들과 엔트리 포인트는 여러분의 출력-파일 레이아웃의 의미 있는 위치들에서 정의될 수 있다.

SECTIONS 명령을 사용하지 않는다면 링커는 각 입력 섹션을 동일한 이름의 출력 섹션에, 그 섹션들이 입력 파일들에서 처음으로 나타난 순서대로 놓는다. 모든 입력 섹션들이 첫번째 파일에 다 있으면, 예를 들어서 출력 파일의 섹션들의 순서는 첫번째 입력 파이르이 순서와 일치할 것이다.

섹션 정의(Section Definitions)

SECTIONS 명령에서 가장 자주 사용되는 문장은 *section definition*이다. 이것은 출력 섹션의 속성들을 지정한다: 이것의 위치, 할당, 내용, 채우기 패턴, 그리고 타겟 메모리 영역. 이런 스펙의 대부분은 옵션이다; 어떤 섹션 정의 가장 간단한 형태는 다음과 같다

```
SECTIONS { ...
  secname : {
    contents
  }
... }
```

*secname*은 출력 섹션의 이름이고 *contents*는 거기에 들어갈 것이 무엇인가에 대한 스펙이다---예를 들어서 입력 파일들의 리스트나 입력 파일들의 섹션들 (see section 섹션 놓기(Section Placement)). *secname* 주변에는 공백 문자가 있어야 한다. 그렇게 섹션 이름이 구분된다. 다른 공백문자는 옵션이다. 그러나 콜론 `:'과 중괄호 `{}'가 필요하다.

*secname*는 반드시 여러분의 출력 포멧의 제약 조건(constraints)과 일치해야 한다. a.out와 같이 제한된 개수의 섹션들만을 지원하는 포멧에서 그 이름은 반드시 그 포멧에 의해서 지원되는 이름들(예를 들어서 a.out는 .text, .data, 또는 .bss만을 허용한다) 중의 하나이어야 한다. 출력 포멧이 임의 개수의 섹션들을 지원하지만 임의 개수만 허용하고 이름들은 그렇지 않다면(Oasys와 같은 경우), 이름은 반드시 따옴표로 묶인 숫자 문자열(quoted numeric string)으로 제공되어야 한다. 섹션 이름은 임의의 문자열로 이루어질 수 있지만 표준 ld 심벌 이름 문법에 맞지 않는 이름은 반드시 따옴표로 묶여야 한다. See section 심벌 이름(Symbol Names).

특수한 *secname*인 `/DISCARD/'는 입력 섹션들을 버리는 데 사용될 수 있다. `/DISCARD/'라는 이름의 출력 섹션에 할당된 임의의 섹션들은 마지막 링크 출력에 포함되지 않는다.

링커는 내용을 가지지 않는 출력 섹션들을 생성하지 않는다. 이것은 존재하거나 존재하지 않을 수 있는 입력 섹션들을 참조할 때 편의를 위한 것이다. 예를 들어서,

```
.foo { *(.foo) }
```

이것은 적어도 한 개의 입력 파일에 `.foo' 섹션이 존재한다면 출력 파일에 `.foo' 섹션을 생성할 뿐이다.

섹션 놓기(Section Placement)

섹션 정의에서 특정 입력 파일들을 넣어서, 특정 입력-파일 섹션들을 넣어서, 또는 이 두 가지를 합해서, 출력 섹션 섹션의 내용물을 지정할 수 있다. 여러분은 또한 이 섹션에 임의 데이터를 놓을 수 있고 섹션 시작에 상대적인 심벌들을 정의할 수 있다.

섹션 정의의 *contents*은 다음과 같은 종류의 문장들 중 임의의 것을 포함할 수 있다. 공백 문자로 다른 것들과 구분된, 단일 섹션 정의에 이들 중 원하는 만큼 포함할 수 있다.

filename

단순하게 현재 출력 섹션에 놓일 수 있도록 특정한 입력 파일의 이름을 지정할 수 있다; @이 파일에서 emph{모든} 섹션들은 현재 섹션 정의에 놓일 것이다. 그 파일 이름이 명시적인 섹션 이름 리스트와 함께 이미 다른 섹션 정의에서 언급되었다면 아직 할당되지 않은 그런 섹션들이 사용된다. 특정한 파일들 리스트를 이름으로 지정하려면:

```
.data : { afile.o bfile.o cfile.o }
```

이 예제는 다수의 문장들이 섹션 정의 내용물안에 포함될 수 있다는 것도 예시한다. 왜냐면 각 파일 이름이 분리된 문장이기 때문이다.

filename(*section*)
filename(*section* , *section*, ...)
filename(*section section* ...)

현재 출력 섹션에 삽입하기 위해서 입력 파일들로부터 하나 이상의 섹션들의 이름을 지정할 수 잇다. 입력-파일 섹션들 리스트를 괄호 안에 지정하고자 한다면 그 섹션 이름들을 공백 문자로 구분하자.

* (*section*)
* (*section*, *section*, ...)
* (*section section* ...)

링크 제어 스크립트안에서 명시적으로 입력 파일들의 이름을 지정하는 대신에 `ld` 명령 라인으로부터 *모든* 파일들을 참조할 수 있다: `*'를 괄호로 묶인 입력-파일 섹션 리스트 앞에 특정한 파일 이름 대신 사용한다. 이미 이름으로 명시적으로 포함된 어떤 파일들을 가지고 있다면 `*'은 모든 *남은* 파일들을 참조한다---출력 파일에 놓일 위치가 아직 정의되지 않은 것들. 예를 들어서 Oasys 파일로부터 `1`에서 `4`까지 섹션들을 `a.out` 파일의 `.text` 섹션에 복사하려면 그리고 `13`과 `14` 섹션들을 `.data` 섹션에 복사하려면:

```
SECTIONS {
  .text :{
    *("1" "2" "3" "4")
  }

  .data :{
    *("13" "14")
  }
}
```

`[*section* ...]'는 모든 할당되지 않은 입력 파일들로부터 이름이 있는 섹션들을 지정하기 위한 다른 대안으로써 종종 받아들여진다. 어떤 다른 운영 체제(VMS)는 파일 이름들에 각괄호들을 허용하기 때문에 그 표기법은 더이상 지원되지 않는다.

filename(COMMON)
*(COMMON)

이것은 출력 파일의 어디에다 초기화되지 않은 데이터를 이 표기법으로 놓을 것인가를 지정한다. `*(COMMON)` 홀로 모든 입력 파일들(할당되지 않는 한)로부터 초기화되지 않은 데이터를 참조한다; *filename*`(COMMON)`는 특정한 파일로부터 초기화되지 않은 데이터를 참조한다. 둘 다 어디에 입력-파일 섹션들을 놓을 것인가를 지정하는 일반적인 메카니즘들의 특별한 경우들이다: `ld`는 초기화되지 않은 데이터를 입력 파일의 포멧에 상관없이 `COMMON` 이라는 이름의 입력-파일 섹션안에 있는 것처럼 이것을 참조할 수 있도록 허용한다.

특정 파일이나 섹션 이름을 사용할 수 있는 임의의 위치에 또한 와일드카드 패턴을 사용할 수 있다. 링커는 유닉스 쉘이 그러한 것처럼 와일드카드들을 처리한다. `*' 문자는 임의 개수의 문자들에 대응한다. `?' 문자는 단일 문자에 대응한다. `[*chars*]' *chars* 중 임의의 문자 단일 인스턴스와 대응할 것이다; `-' 문자는 문자들의 범위를 지정하는 데 사용될 수 있다. `[a-z]'가 임의의 소문자와 대응하는 것처럼 말이다. `W' 문자는 다음 문자를 인용(quote)하는 데 사용될 수 있다.

파일 이름이 와일드카드와 대조될 때 와일드카드 문자들은 `/' 문자와 비교되지 않을 것이다(유닉스에서 디렉터리 이름들을 구분하는 데 사용되는 문자). 단일 `*' 문자로 이루어진 패턴은 예외이

다; 이것은 항상 임의의 파일 이름과 일치된다. 섹션 이름에서 와일드카드 문자들은 `/' 문자와 비교될 것이다.

와일드카드들은 명령행에서 명시적으로 지정된, 파일들만 찾는다. 링커는 와일드카드들을 확장해서 디렉터리들을 검색하지 않는다. 그러나 링커 스크립트 안에서 단순한 파일 이름을 지정한다면---와일드카드를 전혀 가지지 않는 이름--- 그리고 그 파일 이름이 또한 명령행에서 지정되지 않았다면 링커는 그 파일이 명령행에 나타난 것처럼 그 파일을 열려고 시도할 것이다.

다음 예제에서 명령 스크립트는 출력 파일을 세개의 연속된 섹션들, .text, .data, 그리고 .bss안으로 정렬한다. 이들 각각에 대한 입력으로 모든 입력 파일들 중에서 대응된 이름을 가진 것을 취해서.

```
SECTIONS {
  .text : { *(.text) }
  .data : { *(.data) }
  .bss :  { *(.bss)  *(COMMON) }
}
```

다음 예제는 all.o 파일로부터 섹션들 모두를 읽어서 그들을 0x10000 위치에서 시작하는 출력 섹션 outputa의 시작에 놓는다. foo.o 파일로부터 온 .input1 섹션 모두는 동일한 출력 섹션에서 즉각 뒤따라 온다. foo.o의 .input2 섹션 모두는 출력 섹션 outputb으로 들어가고 바로 뒤에 foo1.o의 .input1 섹션이 뒤따른다. 임의 파일에서 온 남은 .input1과 .input2 섹션들은 출력 섹션 outputc에 기록된다.

```
SECTIONS {
  outputa 0x10000 :
    {
    all.o
    foo.o (.input1)
    }
  outputb :
    {
    foo.o (.input2)
    foo1.o (.input1)
    }
  outputc :
    {
    *(.input1)
    *(.input2)
    }
}
```

이 예제는 와일드카드 패턴들이 파일들을 나누는 데 어떻게 사용되는가를 보여준다. 모든 .text 섹션들은 .text에 놓이며 모든 .bss 섹션들은 .bss에 놓인다. 대문자로 시작하는 모든 파일들에 대해서 .data 섹션은 .DATA에 놓인다; 모든 다른 파일들에 대해서 .data 섹션은 .data에 놓인다.

```
SECTIONS {
  .text : { *(.text) }
  .DATA : { [A-Z]*(.data) }
  .data : { *(.data) }
  .bss : { *(.bss) }
}
```

섹션 데이터 표현식(Section Data Expressions)

데이터는 입력 파일들에서 가져 와서 여러분의 출력 파일에 앞에서 말한(foregoing) 문장들이 정렬한다. 데이터를 직접 링크 명령 스크립트로부터 출력 섹션에 놓는 것이 가능하다. 이런 추가의 문장들 대부분이 표현식들을 포함한다 (see section Expressions). 비록 이런 문장들이 보여주기 편하게 분리되어 나타나 있지만 그런 분리는 SECTIONS 명령의 섹션 정의안에서 전혀 불필요하다; 여러분은 우리가 방금 기술한 문장들 중에서 어떤 것도 자유로이 서로 섞을 수 있다.

CREATE_OBJECT_SYMBOLS

이것은 현재 섹션에 각 입력 파일에 대한 심벌을 생성하고 그 입력 파일로부터 작성된 데이터의 첫번째 바이트의 주소로 설정한다. 예를 들어서 a.out 파일들의 경우 각 입력 파일에 대한 심벌을 가지는 것이 편하다. 여러분은 이것을, 출력 섹션 .text를 다음과 같이 정의함으로써, 성취할 수 있다:

```
SECTIONS {
  .text 0x2020 :
     {
    CREATE_OBJECT_SYMBOLS
    *(.text)
    _etext = ALIGN(0x2000);
    }
  ...
}
```

sample.ld가 이 스크립트를 포함한 파일이라면, a.o, b.o, c.o, 그리고 d.o는 다음과 같은 내용을 가진 네 개의 입력 파일들이라면---

```
/* a.c */

afunction() { }
int adata=1;
int abss;
```

`ld -M -T sample.ld a.o b.o c.o d.o' 는 다음과 같이 오브젝트 파일 이름들과 일치하는 심벌들을 가지는 맵 파일을 생성할 것이다:

```
00000000 A __DYNAMIC
00004020 B _abss
00004000 D _adata
00002020 T _afunction
00004024 B _bbss
00004008 D _bdata
00002038 T _bfunction
00004028 B _cbss
00004010 D _cdata
00002050 T _cfunction
0000402c B _dbss
00004018 D _ddata
00002068 T _dfunction
00004020 D _edata
00004030 B _end
00004000 T _etext
00002020 t a.o
00002038 t b.o
00002050 t c.o
00002068 t d.o
```

symbol = *expression* ;
symbol f= *expression* ;

*symbol*은 임의의 심벌 이름이다 (see section 심벌 이름(Symbol Names)). "*f*="는 산술 및 할당을 합성한 &= += -= *= /= 연산자들 중의 임의의 것을 가리킨다. 특정 섹션 정의 안에서 심벌에 값을 할당할 때 그 값은 섹션의 시작에 상대적이다 (see section 할당: 심벌 정의 (Assignment: Defining Symbols)). 다음과 같이 작성하였다면

```
SECTIONS {
  abs = 14 ;
  ...
  .data : { ... rel = 14 ; ... }
  abs2 = 14 + ADDR(.data);
  ...
}
```

abs와 rel는 동일한 값을 가지지 않는다; rel는 abs2와 동일한 값을 가진다.

BYTE(*expression*)
SHORT(*expression*)
LONG(*expression*)
QUAD(*expression*)
SQUAD(*expression*)

섹션 정의에 이런 네 문장들 중 하나를 포함함으로써 명시적으로 그 섹션의 현재 주소에 1, 2, 4, 8 unsigned 또는 8 signed 바이트들을 놓을 수 있다. 64 비트 호스트나 타겟을 사용한다면 QUAD와 SQUAD는 동일하다. 호스트와 타겟이 둘 다 32비트이라면 QUAD는 unsigned 32 비트 값을 사용하고 SQUAD 기호는 그 값을 확장한다. 둘 다 그 값ㅇㄹ 작성할 때 정확한 endianness를 사용할 것이다. 멀티-바이트 값들은 바이트 순서가 출력 파일 포멧에 적절하다면 무엇이든 그 바이트 순서로 표현된다 (see section BFD).

FILL(*expression*)

현재 섹션에 대해서 "채우기 패턴(fill pattern)"을 지정한다. 그 섹션안에서 그렇지 않으면 지정되지 않았을 메모리 영역들(예를 들어서 위치 카운터 `.'에 새로운 값을 할당함으로써 건너 뛴 영역들)은 *expression* 매개변수로부터 온 두 LSB(least significant bytes)들로 채워진다. FILL 문장은 섹션 정의에 있는 위치 *뒤에* 메모리 위치들을 커버한다; 한 개 이상의 FILL 문장을 포함함으로써 출력 섹션의 서로 다른 부분들을 패턴들로 채울 수 있다.

옵션인 섹션 속성(Optional Section Attributes)

다음은 모든 옵션인 부분들을 포함한, 섹션 정의의 완전한 문법이다:

```
SECTIONS {
...
secname start BLOCK(align) (NOLOAD) : AT ( ldadr )
  { contents } >region :phdr =fill
...
}
```

*secname*과 *contents*이 필요하다. *contents*에 대해서는 See section 섹션 정의(Section Definitions), 그리고 section 섹션 놓기(Section Placement). 남은 요소들---*start*, BLOCK(*align*), (NOLOAD), AT (*ldadr*), >*region*, :*phdr*, 그리고 =*fill*---은 모두 옵션이다.

start

start 더하기 섹션 이름을 지정함으로써 출력 섹션이 지정된 주소로 로딩되도록 강제할 수 있다. *start*는 임의의 표현식으로 표현될 수 있다. 다음 예제는 *output* 섹션을 0x40000000에 생성한다:

```
SECTIONS {
  ...
  output 0x40000000: {
    ...
    }
  ...
}
```

BLOCK(*align*)

위치 카운터 .를 섹션 시작점보다 앞서게 하여 그 섹션이 지정된 정렬에서 시작하도록 하는 BLOCK() 스펙을 포함할 수 있다. *align*는 표현힉이다.

(NOLOAD)

`(NOLOAD)' 지시어는 실시간에 어떤 섹션이 로드되지 않도록 마킹한다. 링커는 그 섹션을 정상적으로 처리할 것이지만 그것을 마킹해서 프로그램 로더가 그것을 메모리로 로드하지 않도록 한다. 예를 들어서 아래의 스크립트 샘플에서 ROM 섹션은 메모리 위치 `0'에 위치하고 프로그램이 실행할 때는 로드되지 않아야 한다. ROM 섹션의 내용물들은 링커 출력 파일에서 일반적인 경우와 같이 나타날 것이다.

```
SECTIONS {
  ROM  0  (NOLOAD)  : { ... }
```

```
  ...
}
```

AT (*ldadr*)

AT 키워드 뒤에 따라 오는 표현식 *ldadr*는 그 섹션의 로드 주소를 지정한다. 디폴트(AT 키워드를 사용하지 않는다면)는 위치 재지정 주소와 동일하다. 이 기능은 ROM 이미지를 빌드하기 쉽도록 고안된 것이다. 예를 들어서 이 SECTIONS 정의는 두가지 출력 섹션들을 생성한다: 하나는 `.text'라고 불리는 것이며 이것은 0x1000에서 시작한다. 그리고 다른 하나는 `.mdata'라고 불리는 것이며 이것의 위치 재지정 주소가 0x2000일지라도 `.text' 섹션의 끝에 로드된다. 심벌 _data은 code{0x2000} 값으로 정의된다:

```
SECTIONS
  {
  .text 0x1000 : { *(.text) _etext = . ; }
  .mdata 0x2000 :
    AT ( ADDR(.text) + SIZEOF ( .text ) )
    { _data = . ; *(.data); _edata = . ;  }
  .bss 0x3000 :
    { _bstart = . ;  *(.bss) *(COMMON) ; _bend = . ;}
}
```

이런 식으로 생성된 ROM과 함께 사용하기 위해서, 초기화 데이터를 ROM 이미지로부터 이것의 실시간 주소로 복사하기 위해서, 실-시간 초기화 코드(C 프로그램들의 경우 보통 crt0)는 다음과 같이 어떤 것을 포함해야 한다:

```
char *src = _etext;
char *dst = _data;

/* ROM has data at end of text; copy it. */
while (dst < _edata) {
  *dst++ = *src++;
}

/* Zero bss */
for (dst = _bstart; dst< _bend; dst++)
  *dst = 0;
```

>*region*

이 섹션을 이전에 정의된 메모리 영역으로 할당한다. See section 메모리 레이아웃(Memory Layout).

:*phdr*

이 섹션을 프로그램 헤더에 의해서 기술된 세그먼트에 할당한다. See section ELF 프로그램 헤더(ELF Program Headers). 어떤 섹션이 하나 이상의 세그먼트들에 할당되면, 그들이 명시적으로 :*phdr* 변경자를 사용하지 않는 한, 모든 후속 할당된 섹션들도 같이 이런 세그먼트들로 할당될 것이다. 어떤 섹션이 어떤 세그먼트로 할당되는 것을 막으려면, 보통은 디폴트로 어떤 세그먼트에 할당될 때, :NONE를 사용하자.

=*fill*

=*fill*를 섹션 정의안에 포함하는 것은 초지정하는 것이다. *fill*를 지정하기 위해서 임의의 표현식을 사용할 수 있다. 현재 출력 섹션에 있는 임의의 할당되지 않은 구멍(hole)들은, 출력 파일에 기록될 때, 필요하다면 반복해서, 그 값의 LSB(least significant bytes)로 채워질 것이다. 또한 섹션 정의의 *contents*에 FILL 문장을 써서 값을 채우도록 변경할 수 있다.

오버레이(Overlays)

OVERLAY 명령은 단일 메모리 이미지로써 로드되어야 하지만 동일한 메모리 주소에서 실행되어야 하는 섹션을 기술하는 쉬운 방법을 제공한다. 실행시에 어떤 종류의 오버레이 관리자는 오버레이된 섹션들을 요구된 실시간 메모리 주소로 그리고 이것으로부터 복사할 것이다. 아마 단순하게 어드레싱 비트들을 조작함으로써 그렇게 한다. 이런 접근은, 예를 들어서 메모리의 어떤 영역이 다른 것보다 더 빠를 때 유용할 수 있다.

OVERLAY 명령은 SECTIONS 명령에서 사용된다. 이것은 다음처럼 보인다:

```
OVERLAY start : [ NOCROSSREFS ] AT ( ldaddr )
 {
   secname1 { contents } :phdr =fill
   secname2 { contents } :phdr =fill
   ...
 } >region :phdr =fill
```

모든 것은 OVERLAY (키워드) 를 제외하고 옵션이고 각 섹션은 반드시 이름 (위에서 *secname1* 와 *secname2*) 을 가져야 한다. OVERLAY 구조안의 섹션 정의들은 일반적인 SECTIONS 구조안의 그것들과 동일하다 (see section 출력 섹션 지정(Specifying Output Sections)). 단 어떤 주소들도 그리고 어떤 메모리 영역들도 OVERLAY안에 있는 섹션들에 대해서 정의될 수 없다는 것을 제외하고.

섹션들은 모두 동일한 시작 주소로 정의된다. 섹션들의 로딩 주소들은 그들이 OVERLAY에 대해서 사용된 로드 주소에서 시작해서 메모리에 연속적으로 배치되도록 정렬된다(일반적인 섹션 정의에서처럼 로딩 주소는 옵션이고 디폴트는 시작 주소이다; 시작 주소도 또한 옵션이고 디폴트는 .이다).

NOCROSSREFS 키워드가 사용된다면 그리고 섹션들 사이에 다른 참조들이 있다면 링커는 에러를 보고할 것이다. 섹션들 모두가 동일한 주소에서 실행되기 때문에 하나의 섹션이 다른 것을 직접 참조하는 것은 일반적으로 의미가 없다. See section 옵션 명령(Option Commands).

OVERLAY 에 있는 각 섹션에 대해서 링커는 자동으로 두 개의 심벌들을 정의한다. 심벌 __load_start_*secname*는 그 섹션의 시작 로드 주소로써 정의된다. 심벌 __load_stop_*secname*는 그 섹션의 마지막 로드 주소로써 정의된다. *secname*안에 있는 C identifier들 안에서는 유효하지 않는, 임의의 문자들은 모두 제거된다. C (또는 어셈블러) 코드는 이런 심벌들을 필요한 대로 오버레이된 섹션들을 이동하는 데 사용할 수 있다.

오버레이 마지막에서 .의 값은 오버레이의 시작 주소에 가장 큰 섹션의 크기를 더한 것으로 설정된다.

다음은 예제이다. 이것은 SECTIONS 구조안에서 나타날 것이다라는 것을 기억하자.

```
OVERLAY 0x1000 : AT (0x4000)
 {
   .text0 { o1/*.o(.text) }
   .text1 { o2/*.o(.text) }
 }
```

이것은 .text0와 .text1 둘다 0x1000 주소에서 시작하도록 정의할 것이다. .text0는 주소 0x4000에 로드될 것이고 .text1는 .text0 바로 뒤로 로드될 것이다. 다음 기호들이 정의될 것이다: __load_start_text0, __load_stop_text0, __load_start_text1, __load_stop_text1.

오버레이 .text1를 오버레이 영역안으로 복사하는 C 코드는 다음과 같이 보일 수 있다.

```
extern char __load_start_text1, __load_stop_text1;
memcpy ((char *) 0x1000, &__load_start_text1,
        &__load_stop_text1 - &__load_start_text1);
```

OVERLAY 명령은 단지 문법적인 설탕(sugar)이다. 왜냐면 이것이 하는 모든 일들이 좀 더 기본적인 명령들을 사용ㅎ서 이루어질 수 있기 때문이다. 위의 예는 다음과 같은 것으로 동일하게 작성될 수 있다.

```
.text0 0x1000 : AT (0x4000) { o1/*.o(.text) }
__load_start_text0 = LOADADDR (.text0);
__load_stop_text0 = LOADADDR (.text0) + SIZEOF (.text0);
.text1 0x1000 : AT (0x4000 + SIZEOF (.text0)) { o2/*.o(.text) }
__load_start_text1 = LOADADDR (.text1);
```

```
__load_stop_text1 = LOADADDR (.text1) + SIZEOF (.text1);
. = 0x1000 + MAX (SIZEOF (.text0), SIZEOF (.text1));
```

ELF 프로그램 헤더(ELF Program Headers)

ELF 오브젝트 파일 포멧은 *프로그램 헤더(program headers)* 를 사용한다. 이것은 시스템 로더에 의해서 읽히고, 프로그램이 메모리로 적재되는 방법을 기술한다. 프로그램 헤더는 프로그램을 ELF 시스템에서 실행할 수 있도록 정확하게 설정되어야 한다. 링커는 타당한 프로그램 헤더들을 디폴트로 생성할 것이다. 그러나 어떤 경우에는 그 프로그램 헤더들을 좀 더 자세하게 명시하는 것이 바람직하다; PHDRS 명령은 이런 목적으로 사용될 수 있다. PHDRS 명령이 사용되면 링커는 어떤 프로그램 헤더들도 스스로 생성하지 않을 것이다.

PHDRS 명령은 단지 ELF 출력 파일을 생성할 때만 의미가 있다. 이것은 다른 경우에는 무시된다. 이 매뉴얼은 시스템 로더가 프로그램 헤더들을 해석하는 자세한 사항들을 기술한다; 좀 더 자세한 정보들을 보려면 ELF ABI를 보자. ELF 파일의 프로그램 헤더들은 objdump 명령의 `-p' 옵션을 사용해서 디스플레이될 수 있다.

다음은 PHDRS 명령의 문법이다. PHDRS, FILEHDR, AT, 그리고 FLAGS 단어들은 키워드들이다.

```
PHDRS
{
  name type [ FILEHDR ] [ PHDRS ] [ AT ( address ) ]
        [ FLAGS ( flags ) ] ;
}
```

*name*는 링커 스크립트의 SECTIONS 명령에서 참조를 위해서만 사용된다. 이것은 출력 파일에 들어가지 않는다.

어떤 프로그램 헤더 타입들은 파일로부터 시스템 로더에 의해서 로드되는 메모리 세그먼트들을 기술한다. 링커 스크립트에서 이런 세그먼트들의 내용물은 할당된 출력 섹션들이 그 세그먼트 안에 들어가도록(역자주: 리다이렉트) 함으로써 지정된다. 이렇게 하기 위해서 SECTIONS 명령에서 출력 섹션을 기술하는 명령은 `:*name*'를 사용해야 한다. 여기서 `:*name*'는 PHDRS 명령에 나타난 것과 같이 프로그램 헤더의 이름이다. See section 옵션인 섹션 속성(Optional Section Attributes).

어떤 섹션들이 하나 이상의 세그먼트 안에서 나타나는 것은 일반적이다. 이것은 `:*name*'를 반복함으로써 지정된다. 섹션이 나타날 프로그램 헤더 각각에 대해서 한번씩 이것을 사용하는 식으로 반복한다.

어떤 섹션이 `:*name*'를 사용하여 하나 이상의 세그먼트들안에 놓인다면, `:*name*'를 지정하지 않은 모든 후속 할당된 섹션들은 동일한 세그먼트들 안에 놓인다. 이것은 편의를 위한 것이다. 왜냐면 일반적으로 연속된 섹션들의 전체 집합은 단일 세그먼트 안에 놓일 것이기 때문이다. 디폴트로 하나에 할당되는 것이 관례이지만, 어떤 섹션이 하나의 세그먼트에 할당되는 것을 방지하려면, :NONE를 사용하자.

프로그램 헤더 타입 뒤에 나타날 수 있는 FILEHDR와 PHDRS 키워드들은 또한 메모리의 세그먼트 내용들을 가리킨다. FILEHDR 키워드는 그 세그먼트가 ELF 파일 헤더를 포함해야 한다는 것을 의미한다.PHDRS 키워드는 그 세그먼트가 ELF 프로그램 헤더 자신들을 포함해야 한다는 것을 의미한다.

*type*는 다음과 같은 것들 중의 하나일 수 있다. 숫자들은 키워드의 값을 나타낸다.

PT_NULL (0)
: 사용되지 않는 프로그램 헤더를 가리킨다.

PT_LOAD (1)

이 프로그램 헤더가 파일로부터 로드되는 세그먼트를 기술한다는 것을 가리킨다.

PT_DYNAMIC (2)
: 동적 링크 정보를 찾을 수 있는 세그먼트를 가리킨다.

PT_INTERP (3)
: 프로그램 해석기(interpreter)의 이름을 를 찾을 수 있는 세그먼트를 가리킨다.

PT_NOTE (4)
: 참고(note) 정보를 갖고 있는 세그먼트를 가리킨다.

PT_SHLIB (5)
: 예약된 헤더 타입. 정의된 것이지만 ELF ABI에 의해서 지정되지 않은 것.

PT_PHDR (6)
: 프로그램 헤더들을 찾을 수 있는 세그먼트를 가리킨다.

expression
: 프로그램 헤더의 숫자 타입을 제공하는 표현식. 이것은 위에서 정의되지 않은 타입들에 대해서 사용될 수 있다.

어떤 세그먼트가 메모리의 특정 주소에서 로드되어야 한다는 것을 지정하는 것은 가능하다. 이것은 AT 표현식을 사용해서 가능하다. 이것은 SECTIONS 명령안에서 사용된 AT 명령에 동일하다 (see section 옵션인 섹션 속성(Optional Section Attributes)). 프로그램 헤더에 AT를 사용하는 것은 SECTIONS 명령에 있는 임의의 정보를 오버라이드한다.

일반적으로 세그먼트 플래그들은 섹션들에 기초해서 설정된다. FLAGS 키워드는 명시적으로 세그먼트 플래그들을 지정하는데 사용될 수 있다. *flags*의 값은 정수이어야 한다. 이것은 프로그램 헤더의 p_flags를 지정하는 데 사용될 수 있다.

다음은 PHDRS 사용의 예이다. 이것은 원(native) ELF 시스템에서 사용되는 프로그램 헤더들의 전형적인 모습을 보여준다.

```
PHDRS
{
  headers PT_PHDR PHDRS ;
  interp PT_INTERP ;
  text PT_LOAD FILEHDR PHDRS ;
  data PT_LOAD ;
  dynamic PT_DYNAMIC ;
}

SECTIONS
{
  . = SIZEOF_HEADERS;
  .interp : { *(.interp) } :text :interp
  .text : { *(.text) } :text
  .rodata : { *(.rodata) } /* defaults to :text */
  ...
  . = . + 0x1000; /* move to a new page in memory */
  .data : { *(.data) } :data
  .dynamic : { *(.dynamic) } :data :dynamic
  ...
}
```

엔트리 포인트(The Entry Point)

링커 명령 언어는 결과 파일에서 첫번째로 실행될 명령(*entry point*)을 정의하기 위해서 특별히 고안된 명령을 포함한다. 이것의 매개변수는 심벌 이름이다:

ENTRY(*symbol*)

심벌 할당처럼, ENTRY 명령은 명령 파일 안에서 독립 명령으로써 또는 SECTIONS 명령안에 있는 섹션 정의들 중에 있을 수 있다---어떤 것이든 레이아웃에 대해서 가장 의미있으면 된다.

ENTRY 는 엔트리 포인트를 선택하는 여러가지 방법들 중에서 유일한 것이다. 여러분은 이것을 다음 방법들 중의 하나로 지정할 수 있다(우선순위 내림차순으로 보여진: 이 리스트에서 좀 더 높은 방법들은 더 아래에 있는 방법들을 오버라이드한다).

- `-e' *entry* 명령-라인 옵션:
- 링커 제어 스크립트의 ENTRY(*symbol*) 명령:
- 존재한다면 start 심벌의 값;
- 존재한다면 .text 섹션의 첫번째 바이트 주소;
- 주소 0.

예를 들어서 여러분은 이런 규칙들을 사용해서 할당 문장 안에서 엔트리 포인트를 생성할 수 있다: 입력 파일 안에 start 심벌이 정의되지 않았다면 이것에 적장한 값을 할당함으로써 정의할 수 있다---

```
start = 0x2020;
```

이 예제는 절대 주소를 보여주지만 임의의 표현식을 사용할 수 있다. 예를 들어서 여러분의 입력 오브젝트 파일들이 엔트리 포인트에 대해서 어떤 다른 심벌-이름 관례를 사용한다면 여러분은 start에 대한 시작 주소를 담고 있는 심벌이면 무엇이든 이것의 값을 할당할 수 있다:

```
start = other_symbol ;
```

버전 스크립트(Version Script)

링커 명령 스크립트는 특별히 버전 스크립트를 지정하기 위한 명령을 포함하며 공유 라이브러리들을 지원하는 ELF 플랫폼들에서만 의미가 있다. 버전 스크립트는 링크하는 시간에 링커에 대한 다른 입력 파일처럼, 여러분이 사용하는 링커 스크립트안에 직접 만들어 넣어질 수 있다. 명령 스크립트 문법은 다음과 같다:

```
VERSION { version script contents }
```

버전 스크립트는 `--version-script' 링커 명령행 옵션을 사용해서 링커에게 지정될 수 잇따. 버전 스크립트들은 공유 라이브러리들을 생성할 때만 의미가 있다.

버전 스크립트의 포멧은 솔라리스 2.5에서 Sun의 링커에 의해서 사용되는 그것과 동일하다. 버전을 매기는 것(versioning)은 이름을 가진 버전 노드들 트리와 버전 스크립트안에서 지정한 상호 의존성을 정의함으로써 수행된다. 버전 스크립트는 어떤 심벌들이 어떤 버전 노드에 의존하는지 지정할 수 있고 지정된 심벌 집합을 로컬 범위로 축소해서 그들이 공유 라이브러리 외부에서 글로벌하게 보이지 않도록 할 수 있다.

버전 스크립트 언어의 데모를 보여주는 가장 쉬운 방법은 다음과 같은 몇가지 예제들을 보는 것이다.

```
VERS_1.1 {
        global:
                foo1;
        local:
                old*;
                original*;
                new*;
};
```

```
VERS_1.2 {
                foo2;
} VERS_1.1;

VERS_2.0 {
                bar1; bar2;
} VERS_1.2;
```

이 예제에서 세 버전 노드들이 정의되어 있다. `VERS_1.1'은 정의된 첫번째 버전 노드이고 다른 종속물들을 가지지 않는다. `foo1'는 버전 노드에 종속적이고 여러 오브젝트 파일들안에 보이는 많은 심벌들은 그 범위가 로컬로 줄어들어서 그들은 공유 메모리 바깥에서 보이지 않는다.

다음으로 `VERS_1.2'가 정의되어 있다. 이것은 `VERS_1.1'에 종속적이다. `foo2' 심벌은 이 버전 노드에 종속적이다.

마지막으로 `VERS_2.0' 노드가 정의된다. 이것은 `VERS_1.2'에 종속적이다. `bar1'와 `bar2' 심벌들은 이 버전 노드에 종속적이다.

어떤 버전 노드에 종속적이지 않는, 라이브러리내에서 정의된 심벌들은 그 라이브러리의 지정되지 않은 베이스 버전에 효과적으로 종속적이다. 모든 그렇지 않은 지정되지 않은 심벌들을 버전 스크립트 어딘가에 `global: *'를 사용한 주어진 버전 노드에 종속적이게 하는 것이 가능하다.

어휘적으로 버전 노드들의 이름은 그들이 그것을 읽는 사람에게 제시할 의미와 다른 구체적인 의미를 가지지 않는다. `2.0' 버전은 `1.1'과 `1.2' 사이에서 나타날 수도 있다. 그러나 이것은 버전 스크립트를 혼란스럽게 작성하는 방법이 될 것이다.

어플리케이션과 버전이 붙은 심벌(versioned symbol)을 가지는 공유 라이브러리를 링크할 때 어플리케이션 자신은 이것이 필요로 하는 각 심벌의 버전이 무엇인지를 알고 있으며, 링크하는 각 공유 라이브러리로부터 필요한 버전 노드가 무엇인가도 알고 있다. 그래서 실행시 동적 로더는 링크한 라이브러리들이 실제로 어플리케이션이 모든 동적 심벌들을 해독(resolve)해야 하는 모든 버전 노드들을 제공하는지 빨리 검사할 수 있다. 이런식으로 동적 로드는 이것이 필요로하는 모든 외부 심벌들이 각 심벌 참조에 대해서 조사를 할 필요없이 해독가능(resolvable)일 것이라고 생각하는 것이 가능하다.

심벌의 버전을 매기는 것(symbol versioning)은 SunOS가 하는 마이너 버전 검사(minor version checking)를 수행하는, 좀 더 복잡한 방법이다. 여기에 기술된 기본적인 문제는, 때 전형적으로 외부 함수들에 대한 참조가 필요한 바에 따르는 기초(as-needed basis)에 기반하고, 어플리케이션이 시작할 때 모두가 종속적이지는 않는다는 것이다. 공유 라이브러리가 오래된 것이라면 요구된 인터페이스가 없을 수 있다; 어플리케이션이 그 인터페이스를 사용하려고 할 때, 갑자기 예측하지 못한 실패를 겪을 것이다. 심벌에 버전을 매기는 것으로 사용자는, 사용된 라이브러리들이 너무 오래된 것이라면 그들의 프로그램을 시작할 때 경고를 얻을 것이다.

Sun의 버전 매기는 접근법에 대한 몇가지 GNU 확장판들이 있다. 이들중 첫번째는 심벌을 버전 스크립트 대신 심벌이 정의된 소스 파일에서 버전 노드에 바인딩하는 능력이다. 이것은 주로 라이브러리 관리자의 짐을 덜어주기 위해서 만들어졌다. 이것은 다음과 같은 것을 C 소스 파일안에 넣어서 이루어질 수 있다:

```
__asm__(".symver original_foo,foo@VERS_1.1");
```

이것은 함수 `original_foo'를 버전 노드 `VERS_1.1'에 바인딩된 `foo'에 대한 알리어스로 이름을 바꾸었다. `local:' 지시어는 심벌 `original_foo'이 익스포트되는 것을 막는 데 사용될 수 있다.

두번째 GNU 확장은 동일한 함수에 대한 여러 버전들이 주어진 공유 라이브러리안에 나타나도록 허락한다는 것이다. 이런식으로 어떤 인터페이스에 대한 비호환 변화는 공유 라이브러리의 주 버

전 번호를 증가시키지 않고서 발생할 수 있다. 여전히 예전 인터페이스에 링크된 어플리케이션들이 계속 작동할 수 있도록 한다.

이것은 어셈블러 안에 다수의 `.symver' 지시어들을 사용함으로써만 성취될 수 있다. 이것의 예제는 다음과 같다:

```
__asm__(".symver original_foo,foo@");
__asm__(".symver old_foo,foo@VERS_1.1");
__asm__(".symver old_foo1,foo@VERS_1.2");
__asm__(".symver new_foo,foo@@VERS_2.0");
```

이 예제에서 `foo@'는 지정되지 않은 심벌의 베이스 버전에 묶인 `foo' 심벌을 나타낸다. 이 예제를 담고 잇는 소스 파일은 4개의 C 함수들을 정의할 것이다: `original_foo', `old_foo', `old_foo1', 그리고 `new_foo'.

주어진 심벌의 여러 정의들을 가지고 있다면 이 심벌에 대한 외부 참조들이 바인딩될 디폴트 버전을 제시하는 몇가지 방법들이 있어야 한다. 이것은 `.symver' 지시어의 `foo@@VERS_2.0' 타입으로 성취될 수 있다. 어떤 심벌의 한 버전만이 이런 식으로 '디폴트'로 선언될 수 있다 - 그렇지 않으면 동일한 심벌의 여러 정의들을 효과적으로 가지게 될 것이다.

공유 라이브러리안에 심벌의 특정 버전에다 참조를 바인딩하고자 한다면 여러분은 편리한 알리어스들을 사용할 수 있거나 (i.e. `old_foo'), 또는 `.symver' 지시어를 사용해서 명시적으로 문제의 함수 외부 버전에 바인딩할 수 있다.

옵션 명령(Option Commands)

명령 언어는 특별한 목적들에 사용할 수 있는 다른 여러 명령들을 포함한다. 그들은 명령행 옵션들과 목적상 비슷하다.

CONSTRUCTORS

a.out 오브젝트 파일 포멧을 사용하여 링크할 때 링커는 C++ 글로벌 생성자와 파괴자(destructor)를 지원하기 위해서 비일상적인 집합 구조를 사용한다. ECOFF와 XCOFF같은 임의 섹션을 지원하지 않는 오브젝트 파일 포멧을 링크할 때 링커는 자동으로 C++ 글로벌 생성자와 파괴자를 이름으로 인식한다. 이런 오브젝트 파일 포멧들에 대해서 CONSTRUCTORS 명려은 링커에게 이 정보가 놓여야 할 위치를 말한다. CONSTRUCTORS 명령은 다른 오브젝트 파일 포멧들에 대해서 무시된다. __CTOR_LIST__ 심벌은 글로벌 생성자의 시작점을 마킹하고 __DTOR_LIST는 끝을 마킹한다. 리스트의 첫번째 단어는 엔트리들의 개수이고 그 뒤에 각 생성자나 파괴자의 주소가 따라오며 그 뒤에는 zero word가 따라 온다. 컴파일러는 반드시 그 코드를 실제로 실행하도록 정렬되어야 한다. 이런 오브젝트 파일 포멧들에 대해서 __main 서브루틴으로부터 GNU C++가 생성자를 호출한다; __main에 대한 호출이 자동으로 main의 시작 코드안으로 삽입된다. GNU C++는 atexit를 사용하거나 함수 exit로부터 직접적으로 파괴자들을 실행한다. 다수 섹션들을 지원하는 COFF 또는 ELF와 같은 오브젝트 파일 포멧들에 대해서 GNU C++는 일반적으로 글로벌 생성자와 파괴자의 주소들을 .ctors와 .dtors 섹션들안에 넣는다. 다음을 여러분의 링커 스크립트안에 넣는 것은 GNU C++ 실시간 코드가 보게될 것으로 기대하는 테이블 종류를 빌드할 것이다.

```
__CTOR_LIST__ = .;
LONG((__CTOR_END__ - __CTOR_LIST__) / 4 - 2)
*(.ctors)
LONG(0)
__CTOR_END__ = .;
__DTOR_LIST__ = .;
LONG((__DTOR_END__ - __DTOR_LIST__) / 4 - 2)
*(.dtors)
```

```
LONG(0)
__DTOR_END__ = .;
```

일반적으로 컴파일러와 링커는 이런 이슈들을 자동으로 처리할 것이고 여러분은 그것들에 관심을 가질 필요가 없을 것이다. 그러나 여러분은 C++을 사용하고 사용자 정의 링커 스크립를 작성하고 있다면 이것을 고려해야 할 것이다.

FLOAT
NOFLOAT

이런 키워드들은 어떤 더 오래된 링커들에서 특별한 수학 서브루틴 라이브러리를 요청하기 위해서 사용되었다. ld는 대신에 필요한 서브루틴들이 모두 아카이브로 링크하기 위한 일반 메카니즘들을 사용하여 지정된 라이브러리들안에 있다는 것을 가정하고, 이런 키워드들을 사용하지 않는다: 그러나 더 오래된 링커들에 대해서 쓰여진 스크립트의 사용을 허용하기 위해서 FLOAT과 NOFLOAT 키워드들이 받아들여지지만 무시된다.

FORCE_COMMON_ALLOCATION

이 명령은 `-d' 명령행 옵션과 동일한 효과를 가진다: ld가 재배치 가능한 출력 파일이 지정되더라도 (`-r') 공용 심벌들에 공간을 ld가 할당하도록 하기 위해서 사용한다.

INCLUDE *filename*

링커 스크립트 *filename*를 이 시점에 포함한다. 그 파일은 현재 디렉터리에서, 그리고 -L 옵션으로 주어진 임의의 디렉터리에서 검색될 것이다. 여러분은 INCLUDE에 대한 호출을 10단계까지 겹칠 수 있다.

INPUT (*file*, *file*, ...)
INPUT (*file file* ...)

이 명령을 사용해서, 그들을 특별한 섹션 정의에 포함하지 않고서, 바이너리 입력 파일들을 링크에 포함한다. 각 *file*에 대한 완전한 이름을 지정하자. 필요하다면 `.a'를 포함해서. ld는 각 *file*을, 명령행에서 지정한 파일들에 대해서 하는 것과 동일하게, 아카이브-라이브러리 검색 경로에서 찾는다. `-L'에 대한 설명을 section 명령행 옵션들(Command Line Options)에서 보자. `-l*file*'를 사용한다면 ld는 lib*file*.a의 이름을 명령행 매개변수 `-l'에서와 같이 변환할 것이다.

GROUP (*file*, *file*, ...)
GROUP (*file file* ...)

이 명령은 INPUT와 비슷하다. 이름있는 파일들은 모두 아카이브들이어야 한다는 것과 그들이 새로운 정의되지 않은 참조들이 생성되지 않을 때까지 반복해서 검색된다는 것을 제외하고 비슷하다. `-('에 대한 설명은 section 명령행 옵션들(Command Line Options)를 보자.

OUTPUT (*filename*)

이 명령은 링크 출력 파일의이름을 *filename*로 지정할 때 사용한다. OUTPUT(*filename*)의 효과는, 이것을 오버라이드하는 `-o *filename*'의 효과와 동일하다. 여러분은 이 명령을 사용해서 a.out이 아닌 디폴트 출력-파일 이름을 지정할 수 있다.

OUTPUT_ARCH (*bfdname*)

BFD 백-엔드 루틴들(see section BFD)에 의해서 사용되는 이름들 중의 하나로, 특별한 출력 머쉰 아키텍쳐를 지정한다. 이 명령은 종종 불필요하다; 아키텍쳐는 종종 시스템 BFD 설정이나 OUTPUT_FORMAT 명령의 부대 효과로써 암묵적으로 설정된다.

OUTPUT_FORMAT (*bfdname*)

ld가 다수의 오브젝트 코드 포멧들을 지원하도록 설정될 때 여러분은 이 명령을 사용해서 특별한 출력 포멧을 지정할 수 있다. *bfdname*는 BFD 백-엔드 루틴(see section BFD)들에 의해서 사용되는 이름들 중의 하나이다. `--oformat' 명령행 옵션의 효과와 동일하다. 이 선택은 출력 파일에만 영향을 미친다; 관련된 명령 TARGET는 주로 입력 파일들에 영향을 미친다.

SEARCH_DIR (*path*)

*path*를 ld가 아카이브 라이브러리들을 찾는 경로 리스트에 추가한다. SEARCH_DIR(*path*)는 명령행의 `-L*path*'과 동일한 효과를 가진다.

STARTUP (*filename*)

*filename*이 링크 과정에서 사용된 첫번째 입력 파일이도록 한다.

TARGET (*format*)

ld가 다수의 오브젝트 코드 포멧들을 지원하도록 설정될 때, 이 명령을 사용해서 입력-파일 오브젝트 포멧을 변경할 수 있다(명령-행 옵션 `-b'나 또는 유사어 `--format'를 사용하는 것과

같이). *format* 매개변수는 BFD에 의해서 바이너리 포멧들의 이름을 지정하는 데 사용되는 문자열들 중의 하나이다. TARGET이 지정되지만 OUTPUT_FORMAT은 아니라면 마지막 TARGET이 또한 ld 출력 파일에 대한 디폴트 포멧으로써 사용된다. See section BFD. TARGET 명령을 사용하지 않는다면 ld는 출력 파일 포멧을 선택하기 위해서 GNUTARGET 환경변수의 값을, 이것이 가능하다면, 사용한다. 그 변수도 없다면 ld는 BFD 라이브러리들안에서 여러분의 머쉰을 위해서 설정된 디폴트 포멧을 사용한다.

NOCROSSREFS (*section section* ...)

이 명령은 ld에게 어떤 섹션들 사이에 있는 참조들에 대해서 에러를 발생하도록 하라고 지시하는 데 사용될 수 있다. 어떤 프로그램 타입들에서는, 특별히 임베디드 시스템들에서는, 한 섹션이 메모리로 로드될 때 다른 섹션은 그렇지 않을 것이다. 두 섹션들 사이의 직접적인 참조들은 어떤 것이든 에러를 발생할 것이다. 예를 들어서 한 섹션에 있는 코드가 다른 섹션에 있는 함수를 호출했다면 에러가 발생할 것이다. NOCROSSREFS 명령은 섹션 이름들 리스트를 취한다. ld가 이런 섹션들 사이의 교차 참조를 하나라도 검출한다면 이것은 에러를 보고하고 0이 아닌 종료 상태값을 리턴한다. NOCROSSREFS 명령은 SECTIONS 명령에서 정의된 출력 섹션 이름들을 사용한다. 이것은 입력 섹션들의 이름은 사용하지 않는다.