안녕하세요...

이것은 그냥 제가 아는 한도 내에서 DJGPP의 문법이나 특징등을
요약한 것입니다. 보시기 싫으시면 그냥 지우세요...

박 진홍

1. DJGPP를 알기에 앞서서...

DJGPP는 DJ Delorie라는 분이 만드신 공개용 도스 32비트 C 컴파
일러 입니다. 기본적으로 과거 리눅스에서 쓰였던 a.out 포맷으로
파일을 생성하는데 여기에 도스에서 실행 가능한 프로그램을 덧붙
여서 실행합니다. 컴파일 환경은 유닉스 계열에서 나온 컴파일러
인 만큼 flat 메모리 모델에 가상 메모리를 지원합니다. (flat 메
모리 모델에서는 세그먼트 혹은 셀렉터를 생각해 줄 필요가 없습
니다.) 현재까지 나온 최신 버전은 DJGPP V2.01 입니다. 만약 2.0
을 쓰고 있다면 2.01로 바꾸세요.. 2.0 패키지에 든 GCC 2.7.2에
는 최적화 기능에 일부 문제가 있습니다. 또, DJGPP V2.0 패키지
안에는 볼랜드 컴파일러의 통합환경과 같은 RHIDE라는 것이 들어
있는데 이것은 beta 버전입니다. RHIDE 1.0 이상의 정식판을 따로
구해야 합니다.

2. 우리나라 사설 통신망에서 DJGPP가 배포된 곳

이제 DJGPP 사냥을 떠나야죠... 하이텔, 나우누리, 천리안에
서 알아보죠..

[하이텔]
소프트웨어 동호회 (go soft) 자료실 [DJGPP V2.0, GCC 32Bit]
프로그램 전문 동호회 (go prog) 자료실 [비공개, DJGPP V2.0, 2.01]
OS 동호회 (go osc) 자료실 [DJGPP V2.01]

[나우누리]
리눅스 동호회 (go linux) 자료실 [DJGPP V2.0]
프로그램 동호회 (go prog) 자료실 [DJGPP V2.0, V2.01]
게임 제작 포럼 (go ngm) 자료실 [DJGPP V2.01, RHIDE 1.1]

[천리안]
공개 자료실 (go pds) [DJGPP V2.0 전체]

RHIDE 1.1 정식 버전은 나우누리 게임 제작 포럼에만 있는 것으로 알고 있습니다. 또, DJGPP V2.01만 올려놓은 것은 DJGPP 패키지 중에서 최소한의 부분만 올려 놓은 것입니다. 위에서 GCC 32Bit 라고 적은 것은 윈도우 95용 GNU C 컴파일러 입니다.

3. DJGPP의 설치

DJGPP 패키지를 보시면 상당히 놀라죠... 왜이리 ZIP으로 압축된 파일이 많냐고요.. 하지만 이중에서 기본적으로 설치할 파일은 몇 개 안됩니다. 아래의 목록은 V2.01을 기준으로 한 것입니다. 뒤에 나오는 숫자가 버전이니까 쉽게 알아 보실수 있습니다.

DJDEV201.ZIP, BNU27B.ZIP, GCC2721B.ZIP, GPP2721B.ZIP, LGP271B.ZIP

위에 나열한 파일들을 한 디렉토리에 풀어줍니다. 예를들어 C:\DJGPP 라는 디렉토리를 만들고 PKUNZIP으로 풀어 넣으면 되겠죠. 나머지 패키지들도 같은 디렉토리에서 풀어 주시면 됩니다.

그다음에는 그 아래에 생긴 BIN 디렉토리를 path에 걸어 주어야 합니다. 또, 환경변수 하나를 적어 줘야 하는데요... 위에서 예를 든 것에서 본다면

PATH=<원래있던 경로>;C:\DJGPP\BIN
SET DJGPP=c:/djgpp/djgpp.env

가 되겠죠. 다른 디렉토리에 했다면

PATH=<원래있던 경로>;디렉토리\BIN
SET DJGPP=디렉토리/djgpp.env

두번째 디렉토리는 유닉스 형식으로 \대신 /를 써야 합니다.

4. DJGPP의 문법적인 문제

C 컴파일러에서는 ANSI C 및 볼랜드 컴파일러 계열의 문법을

쓰고 있어서 상당히 친근합니다. 다만 그래픽 환경에 있어서는 매우 독특한 환경을 제공하고 있어서 이 부분에 관심이 많으신 분들은 따로 공부하셔야 합니다.

어셈블러에서는 인ㅌ 문법이 아닌 AT&T 문법을 쓰고 있어서 상당히 틀립니다. DJASM 이라고 해서 인텔 문법을 쓰는 어셈블러도 있지만 이건 실제 컴파일때에 쓰는게 아닙니다. 그리고 C에서 쓰는 인라인 어셈블리도 볼랜드와는 약간의 구조가 틀립니다. 역시 AT&T 문법을 쓰죠..

5. 컴파일의 실제

아무것도 없는 상황에서 컴파일 하려니까 난감하죠.. 여기서는 모든 컴파일을 GCC가 해 줍니다. GCC가 해당 전용 컴파일러를 실행시키는 형태죠.. 우선 쓸 수 있는 GCC의 옵션을 알아 볼까요?

-Ox : x번째의 최적화를 수행한다.
GCC에는 최적화 옵션이 너무 많기 때문에 대략적으로 몇개 모아서 하나의 옵션으로 쓰도록 했습니다. -O1 은 기본적인 최적화 -O2는 중간, -O3는 최대의 최적화 입니다. 4 이상의 숫자를 쓰면 3으로 최적화를 수행합니다.

-v : GCC의 버전을 알려 줍니다. 자신이 쓰고 있는 GCC의 버전을 알고 싶을때 씁니다.

-S : C소스를 어셈블리 소스로 변환합니다. 결과로 확장자가 S인 어셈블리 소스가 나옵니다.

-o 출력파일명
: 기본적으로 컴파일하면a.out과 a.exe가 생성됩니다. 이 옵션을 지정하면 출력 파일명이 바뀌게 됩니다.

-g : 실행 코드에 소스 코드의 디버깅 정보를 넣어 둡니다. 디버깅 할때 편리합니다.

[1] C 언어 사용

그럼 간단히 예제나 하나 만들어 볼까요..?

```
#include

void main()
{
        int x=3, y=2;

        printf(" Multiplied = %d\n", x*y);
}
```

이걸 컴파일 해보죠... 예제가 너무 간단하죠..?

```
gcc -O3 exam1.c -o exam1
```

결과로는 exam1 과 exam1.exe가 나오죠... 이걸 실행하면

```
Multiplied = 6
```

이라고 나올 것입니다. 그럼 일반적인 도스 컴파일러와의 차이점은 무엇일까요..?

가장 큰 차이점은 바로 메모리를 640KB 이상 쓸수 있고 도스에서 처럼 64KB의 세그먼트 한계를 갖지 않는다는 것입니다. 도스에서

```
char *a;

a=malloc(65537);
```

이라고 하면 안되죠. 제대로 메모리가 할당되지 않지요. 물론 farmalloc을 쓰면 최대 640KB까지 되는 만큼 할당할 수 있지요.. 하지만 huge 메모리 모델을 쓰지 않고는 이 메모리를 관리하는데 상당히 문제가 생깁니다. 그러나 DJGPP에서는 이 모든 문제를 생각할 필요가 없습니다.

```
char *a;
int i;
```

```
a=malloc(1024*1024);
```

라고 해도 상관이 없습니다. (물론 메모리가 허용할 경우
입니다.) 그리고

```
for(i=0; i<1024*1024; i++) *(a+i)=NULL;
```

라고 하면 모든 영역이 NULL로 메워지며 자연스럽게 동작
합니다. 이것은 int 형이 4바이트라서 1024*1024를 해도
그 값을 계산할 수 있기 때문입니다. 그리고 세그먼트라는
개념이 없기 때문에 가능한 것이죠. 도스에서는

```
char *a;
long i;

a=farmalloc(65557); /* 문제 없습니다 */

for(i=0; i<65557L; i++) *a++ = NULL; /* 사용 주의 */
```

이라고 하면 상당한 위험 부담이 따르게 됩니다. 물론 huge
모델에서는 괜찮습니다. 다른 메모리 모델에서는 옵셋만 변
하기 때문에 ++ 명령이 제대로 먹히지 않습니다. 이것의 실
행 결과로 시스템이 다운되는 사태까지 벌어질 수 있습니다.

그리고 또하나 중요한 점은 할당되지 않은 메모리 영역에 접
근할 경우에는 자동적으로 폴트를 일으키며 종료합니다. 즉,
이것은 도스상의 인터럽트 벡터 같은 것이라도 포인터를 써서
변경할 수 없다는 것입니다. 이것이 상당한 제약이 될 수 있
으나 방법이 없는 것도 아니므로 그리 크게 신경 쓸 문제는
아닐 것입니다.

DJGPP는 메모리 상에서 보호를 위반하는 (자신의 영역 밖의
메모리에 대해 간섭하는) 것 외에는 거의 다 허용합니다. 심
지어 int86()을 이용하여 리얼 모드 인터럽트까지 쓸 수 있
습니다. 이번 기회에 볼랜드 사용자 분들이 많이 이용하시면
좋겠네요...

[2] 어셈블리어 사용

그리고 그 다음으로 중요한 문제가 남았군요... 바로 어셈
블리어에 대한 것입니다. 위에서도 언급했듯이 여기서는 AT&T
문법을 쓴다고 했습니다. 그럼 과연 AT&T 문법은 어떤 것인지
한번 알아보죠...

[기본 형태]
명령어+명령어크기지시어 왼본, 대상부분

[예]
AT&T〉 movl %eax, %ecx
Intel〉 mov ecx, eax

AT&T〉 movw -14(%ebp), %ax
Intel〉 mov ax, word ptr [ebp-14]

AT&T〉 movw $1, %eax
Intel〉 mov eax, 1

위에서 볼 수 있듯히 인텔 문법과는 상당히다릅니다. 여기에
대해서 대강 알아보고 넘어가죠...

레지스터를 쓸때는 반드시 앞에 %를 붙여야 합니다. 메모리
참조와 구분하기 위한 것 입니다. 그리고 상수는 앞에 $를
붙입니다. 안붙이면 메모리 참조로 알고 그 숫자의 옵셋에다
멀 써넣어 버리죠.. 그러면 세그먼트 폴트가 일어나죠.

기본형태에서 명령어 부분은 인텔과 동일합니다. mov나 shl등
이름이 같습니다. 그 다음에 나오는 크기지시어는 몇개가 있습
니다.

8bit (1 byte) : b
16bit (2 byte) : w
32bit (4 byte) : l

이 규칙이 모든 경우에 적용됩니다. 심지어 부동 소숫점 연산에
있어서도 real형이 4바이트 이므로 l을 쓰면 됩니다. 그리고 몇
몇 명령어에서는 조금 다른 형태를 취합니다. 왼본과 대상의 크

기가 다른 명령어에서는 원본의 크기지시어와 대상의 크기 지시어를 둘다 지정합니다. movzxbw %al, %cx 와 같이 쓰이는 거죠.. b가 원본의 크기, w가 대상의 크기가 되는것입니다.

메모리 참조에 있어서는 [] 대신 ()를 쓰며 숫자는 앞에 넣고 두개 이상의 레지스터 참조의 경우에는 (레지스터, 레지스터) 의 형식을 씁니다.

AT&T〉 movl %eax, (%ebx, %edi)
Intel〉 mov [ebx+edi], eax

여기서 한가지 짚고 넘어갈 일이 있습니다. 이것만으로는 어셈 프로그래밍은 무리죠... 데이터 기억 장소도 잡아 줘야 하고 여러가지 많이 필요한데 그중에서 기본적인 틀을 아는것도 중요하죠. 과연 AT&T 어셈의 기본 템플릿은 어떨까요?
그건 다음과 같습니다.

```
        .data

#       data added to here!

        .bss

#       data added to here!

        .text
        .globl _main

_main:
        call ___main

#       main program added to here!

        ret
```

너무 간단했나요..? 이것은 c로 껍데기 함수를 만들고 나서 gcc -S 로 뽑아낸 것입니다. 아마 call ___main은 초기화 함수인 것 같네요. 어셈블리 코드도 역시 gcc가 만들어낸 초기화 프로그램 아래에서 동작하므로 _main이 없으면 동작하지 않습니다.

그러니까 위에 나온 것들은 기본적으로 지켜야 하는 것이죠. 그리고 초기화 프로그램에 의해 실행되므로 종료함수 같은 것은 필요가 없습니다. 단지 ret만 하면 종료됩니다. 강제 종료의 경우 C에서 쓰는 함수인 exit 함수를 호출하면 됩니다. 즉,

```
        pushl $0
call _exit
```

만 하면 그만이죠. $0 대신 아무거나 넣어도 상관은 없죠.

그럼 위에서 나온 예약어들을 알아보겠습니다.

[세그먼트(?) 선언]
.text : 코드의 시작을 알립니다. 세그먼트의 개념이 없으므로 일종의 시작 레이블 정도로 생각하시면 됩니다. 도스상에서는 code segment와 같습니다.
.bss : 초기화 되지 않은 데이터가 여기에 들어갑니다.
.data : 초기화 된 데이터가 여기에 들어갑니다.

[레이블 범위 지정]
.globl : 뒤에 나온 레이블(label)이 다른 외부의 오브젝트 모듈에도 알려지게 합니다. 도스상의 public과 같습니다.
.extern : 도스상에서도 extern은 쓰죠. 말그대로 이 레이블은 다른 오브젝트 모듈에 존재함을 선언합니다.

[코드 최적화]
.align : 도스용 어셈블러에도 이 기능이 있는 것으로 알고 있습니다. 이 기능은 뒤에 나온 숫자의 배수가 되도록 오브젝트 코드를 정렬합니다. 32비트 프로세서에서는 4의 배수가 되는 지점의 코드가 조금 더 빨리 실행되죠. 이때 .align 4를 하면 됩니다. gcc에서는 점프 명령 이후에 습관적으로 .align 명령을 씁니다.

[형식적인 선언]
.file : 그냥 소스 파일명 적어주는 곳입니다. 필요는 없습니다.

[데이터 선언]
.byte : 1바이트의 기억공간을 할당합니다. MASM의 db와 같습니다.
.word : 2바이트의 기억공간을 할당합니다. MASM의 dw와 같습니다.
.long : 4바이트의 기억공간을 할당합니다. MASM의 dd와 같습니다.
.ascii : 문자열을 저장할 공간을 만듭니다. C에서 쓰는 모든 ESC

문자를 쓸 수 있습니다. (\r, \n, \0 등...)

여기서 데이터 선언 방법을 자세히 알아보죠.

```
LABEL:
        .ascii "하하하... 백수 만세!\0"
```

이런식으로 하면 됩니다. MASM에서는 변수의 경우 :는 붙일 필요가 없었지만 AT&T에서는 그렇게 하더군요... 그런데... 왜 데이터형이 4바이트까지 밖에 없는 것일까요? double 형은 어떻게 나타낼까요? 그건 단순하게 해결 합니다. 그냥 .long으로 선언해서 4byte 값 두 개로 선언하는 것입니다. 상당히 황당하지만 그 방법밖에는 없더군요.

그럼 이제는 할당된 데이터를 활용하는 방법을 알아보죠... AT&T 문법에서는 꼭 레지스터 앞에 %를 붙이고 상수앞에 $를 붙였습니다. 그럼 왜 붙였을까요? 바로 메모리 참조와 구분하기 위한 것이었습니다. 위에서 LABEL 이라고 정의된 곳에 뭘 좀 넣어 볼까요?

```
movl $0xFFFFFFFF, LABEL
```

아주 쉽죠? LABEL에다 0FFFh를 네개 넣은 것이죠. 그리고 MASM등에서는 꿈도 꿀 수 없었던 메모리 번지 지정이 가능합니다. 바로 번지 수만 윈본이나 대상에 넣으면 끝입니다.

```
movl $0, 0
```

도스의 인터럽트 0번 벡터에 0:0을 채웠습니다. 다만 이건 세그먼트 폴트감이라서 실전에서는 쓸수 없습니다. 사실 직접 메모리 지정은 거의 활용할 곳이 없죠... 고정된 벡터를 가진 부분이 아닐 경우에는요...

이론은 이정도에서 끝내고 실전 연습을 해야죠... a와 b에 값을 넣고 그 합을 출력하는 프로그램을 만들어 볼까요?

```
        .data

a:
        .long 12345
```

```
b:
        .long 54321

format:
        .ascii "The result of %d + %d = %d\n\0"

        .text
        .globl _main

_main:
        call ___main

#       push a+b
        movl a, %eax
        addl b, %eax
        pushl %eax

#       push b
        pushl b

#       push a
        pushl a

#       get the pointer of 'format' and push it
        lea format, %eax
        pushl %eax

#       call the C function printf
        call _printf

#       restore the stack
        addl $16, %esp

        ret
```

에구.. 말하자 말자 프로그램이 튀어나오는군요... ^^; 참고로 # 이 붙은 부분은 주석입니다. 어셈 프로그램은 도스에서와 같습니다. 초기화 프로그램에 C의 기본 함수들이 포함되어 있기 때문에 그 중에서 printf 함수를 썼습니다. 도스와 다른 점이 있다면 포인터의 경우 세그먼트(프로텍티드 모드에서는 셀렉터로 바뀌죠)를 쓸 필요가 없다는 것입니다. 혹시나 해서 ds 나

es를 바꾸는 것은 자살행위일 뿐입니다. 절대 세그먼트를 쓰려고 하면 안됩니다. 도스에서 small 메모리 모델로 작성한 프로그램처럼 만들어야 합니다. 그리고 위 소스에서 스택에 푸쉬하는 순서가 역으로 되어 있는 것은 C언어의 함수 호출 규약에 따른 것입니다.

[3] 인라인 어셈블리 사용

그런데 위의 소스를 보시면 아시겠지만 차라리 printf 함수를 쓸 것이면 출력부분은 C언어로 하고 어셈블리는 인라인 어셈블리로 구현하는게 낫지 않겠느냐는 의문이 생기실 것입니다. 그래서 여기서 부터는 인라인 어셈블리에 대한 설명입니다.

C 컴파일러라고 하면 기본적으로 들어가 있는 기능이 인라인 어셈블 기능입니다. 그렇다면 DJGPP에도 인라인 어셈블 기능이 있겠죠. 그럼.. 그것이 알고싶다.. 인라인 어셈블리... 시작해 보죠..

C 언어에서 인라인 어셈블리를 쓸려면 기본적으로 다음과 같이 해 주어야 합니다. 아.. 그리고 여기서 부터는 나우누리 유경상님의 글을 표절했습니다. ^^;

asm ("어셈명령\n" "어셈명령\n" ...);

혹은 C에서 쓰는 데이터를 써야 할 경우

asm ("어셈명령\n" "어셈명령\n" ... : "=m" (변수명1), "=m" (변수명2) ...);

로 해 줍니다.

그럼 이걸로 끝이냐... 그것은 아닙니다. 인라인 어셈블리에서 또 AT&T 문법이 한번더 탈바꿈 합니다. 정말 무지막지 합니다. 이제는 레지스터 앞에 %%를 붙여야 합니다. 이건 또 무엇 때문일까요? 위에서 C의 변수를 쓸려면 ''=m" (변수명)'의 형태로 써야 한다고 했는데 이건 어떻게 이용할까요? 바로 %0, %1, %2의 순서로 이용합니다. 위에서 변수명1은 %0, 변수명2는 %1 등의 형식을 취하죠.. 그럼... 장황하게 말로 쓰지말고 실제 예를 들어보죠.

```
#include
#include

void Memcpy(void *, void *, int);

void main()
{
        char a[128], b[128];

        /* a에 문자열을 입력 받는다 */
        printf("Enter the string : ");
        gets(a);

        /* 그 문자열을 그대로 복사한다 */
        Memcpy(b, a, strlen(a));

        /* 복사된 문자열을 출력한다 */
        printf("The copied string is %s\n", b);
}

void Memcpy(void *b, void *a, int x)
{
        asm ("movl %0, %%esi\n"
             "movl %1, %%edi\n"
             "movl %2, %%ecx\n"
             "rep\n"
             "movsb\n"
             "movb $0, (%%edi)\n"
             : "=m" (a), "=m" (b), "=m" (x)
        );
}
```

위에 나온 Memcpy 함수는 일반적인 memcpy 함수와는 달리 문자열을 지정한 갯수만큼 복사하고 나서 끝에 NULL 문자를 붙여 줍니다. 아주 단순한 함수죠. GCC의 최적화 성능이 좋기 때문에 달리 이렇게 함수를 만들 필요는 없지만 아주 빠른 성능을 요구하는 프로그램이 있다면 이런식으로 만들어 쓰면 좋겠죠. 위의 예에서 %0는 a 이고 %1은 b 입니다. 또, %2는 x가 되죠. 그리고 위에서 주목할 것이 있는데 rep movsb 같은 것은 두개가 독립이므로 다른 줄에 적어야 한다는 것입니다.

6. DJGPP 에서의 디버깅

DJGPP 패키지에는 디버거가 많이 있습니다. 그중에서는 도스 DEBUG 스타일의 GDB와 Turbo Debugger를 떠올리게 하는 FSDB가 있습니다. 특히 FSDB의 경우 디버깅 정보가 없어도 그냥 어셈블리 소스 형태로 나타내며 소스는 MASM 의 인텔 문법을 사용합니다. GDB의 경우 명령어 형태가 틀리며 메뉴 방식이 아니라서 상당히 불편합니다. 따라서 FSDB를 쓰는게 훨씬 낫겠죠. 자, 그럼 FSDB를 실행해 봅니다..

```
C:\TEMP>fsdb
Usage : 어쩌구 저쩌구 이러쿵 저러쿵..

...

C:\TEMP>
```

에구... 완벽한 메뉴 방식이 아니죠... ^^; FSDB 뒤에다가 디버깅 대상 파일을 선택합니다. 그러면 터보 디버거랑 비슷하게 생긴 디버깅 화면이 나옵니다. 키 배열도 터보 디버거랑 비슷해서 배우기 쉽습니다. 주로 쓰는 키들은

[모든 곳에서]

Tab : 다른 창으로 이동
F1 : 도움말
F10 : 메뉴

[코드 창에서]

F2 : 브레이크 포인트 지정/삭제 (bpx)
F4 : 현재 커서가 있는 곳까지 실행 (here)
F7 : 한개의 명령 실행 (t)
F8 : 한묶음의 명령 실행 (p)
F9 : 그냥 실행 (g)

[데이터 창에서]

Ctrl-G : 데이터 창이 나타날때 원하는 주소를 입력하면 그 주소

로 이동합니다. 단, 16진수를 입력할 때는 C언어에서 쓰
듯이 0x12 등의 형식으로 씁니다.

이정도면 쓸 수 있겠죠. ()안에 든것은 소프트 아이스에서 쓰는 명령을 적은 것입니다. 브레이크 포인트는 실행되다가 그 부분에 가면 멈추라는 것이지요. g 명령을 주어도 브레이크 포인트에서는 멈춥니다. 또, 한묶음의 명령을 실행한다는 것은 현재의 명령어가 call 이나 int, rep 같은, 다음 줄로 되돌아 올 수 있는 명령에 대해서는 중간 과정을 추적하지 말라는 것입니다. 그리고 디버깅을 편리하게 하기 위해서는 컴파일 할때 -g 옵션을 써서 컴파일 하는게 좋습니다. 그렇게 하면 소스 코드가 어셈블리어 코드위에 나타납니다.

7. 마치면서

제가 알고 있는 지식이 얼마 없어서 더 이상 알려드릴 수 없네요. 이제 각자 실력을 쌓을 때 입니다. 직접 C로 코딩을 해서 gcc -S 옵션으로 어셈블리 출력을 알아보면 실력 향상에 큰 도움이 될 것 같네요...