THE MINIX ASSEMBLY LANGUAGE MANUAL

Author Mao yao，Yang songhua
Translated and edited by: Wangzhi e-mail: quakewang@mail.whut.edu.cn

Minix Assembly language is a little different to the Masm, which Most of us have learned. But In Minix OS There is many pieces of assembly code, which is very necessary to understand everything. I have found a Chinese paper talk about it (Posted on the Journal of Southwest University for Nationalities-Natural Science Edition Vol 29,5, Authors are Mao yao and Yang songhua). It is very good and useful. So here I translated and edited it to share them with all the people. I must mention that the author of this paper thanks Kees J. Bot two times.

I added some examples to explain the things clearer (All examples in this document are written by myself), correct several errors, and I change the format of original paper greatly to make it more like a manual. This document will explain almost all aspects of Minix assembly language, they are compiling, structures, syntax, and interchange between assembly and C code. At last I will give two examples using some features of Minix assembly language (All the examples are run able and tested under minix2.0.0).

1 COMPILING

In Minix you can use cc compiler to compile the assembly language code. Because cc must start from a function named “main”, if all of your code is written in assembly language, you must have a function named “_main” to indicate main entry point.

If you save the file as “a.s”, then use "cc a.s" command to generate the executable file “a.out”, and you can type "./a.out" the run it. If you want to just compile the assembly language file, use "cc -c a.s" then you will get an object file “a.o”.

A very useful parameter of cc compiler is "-S". If you have a c file “b.c”, using "cc -S b.c" you will get assembly language file b.s. In another word the c language file “b.c” is converted into assembly language file “b.s”. Using this method you can study all Minix assembly language features.

2 STRUCTURES

A Minix assembly language file includes four sections. They are:

1. Code section (.sect .text): executable code is put here.
2. Read only data section (.sect .rom): read only data is put here. The data in rom section can be changed (write) if the physical medium allows it. In other word the rom section is just suggest these data can be saved in (read only memory) but not forbid the code to be changed (written). And almost all the time the data defined in rom section is just saved on RAM and can be changed.
3. Data section (.sect .data): read and write able data is put here.
4. Global data section (.sect .bss): global data is put here. The data define in this section will be initialed to zero. If you put a nonzero data in this section the compiler will give you a error.

Every section is declared by ".sect" following a section name(“.sect .data” declare a section

and the name of this section is “.data”) . The function of each section is indicated by the **order** of their defined orders in the file (**not** the name of the section). The first section **which compiler meet** is the (code section), the second is read only data section, the third is data section, the fourth is global data section. You can give any name to a section (but conform name rule, for example (**.sect Wangzhi**). All the things of one section may not be all put in one place, the compiler will know them belong to one section according to the section name before them. In Minix these is not stack section, the compiler will allot stack space (you can use “chmem” command to change it later).

For a good program habit, you shall declare the four sections at the file beginning of every assembly language file, and give a standard name to four sections like these (the standard name of section is started by a "."):

```
.sect .text    ! For code
.sect .rom     ! For read only data
.sect .data    ! For read and write able data
.sect .bss     ! For global data
```

3 SYNTAX

3.1 name rule

You can use period"." underline"_" Alphabetic letters (a--z and A--Z) and digits (0--9). You cannot put digit at the beginning of a name, but you can use a digit as an identifier.

3.2 Character string

Using single quotation mark (') or double quotation mark (") to embody the string. like these:

"I like Minix!"

'Happy Chinese NEW YEAR!'

3.3 parentheses and square brackets

Using parentheses"()" to indicate this operand is a address. Using square brackets"[]" to change the priority.

3.4 comments

Using exclamatory mark (!), from the “!” to the end of this line is the comment.

3.5 prefix and postfix

(1) 16/32 operand prefix

Using "o16" and "o32" to indicate the operand is 16/32 bits.

(2) 16/32 address prefix

Using "a16" and "a32" to indicate the address is 16/32 bits.

(3) Instruction repeat prefix

"rep" indicate repeat the instruction when cx register is not zero.

repz/repnz/repe/repne indicate repeat the instruction when cx register is not zero and at the same time the result is zero/not zero/equal/not equal.

(4) Segment prefix

Using cseg/dseg/eseg/fseg/gseg/sseg to indicate addressing in cs/ds/es/fs/gs/ss segment register.

(5) Far jump postfix

Using instruction "jmpf" and "callf" to indicate far jump and far call. In 32 bits OS you should using these instructions like this:

jmpf SEGMENT:OFFSET

callf SEGMENT:OFFSET

(The "SEGMENT" can not be a variable, it is must be a constant.)

(6) 8 bits registers postfix

If you use 8 bits registers you must add a postfix "b" to the instruction. Like:

movb (edi), ah

3.6 Identifier

The declaration of a Identifier is:

Minix:

Wangzhi:

3.7 Operand

(1) Registers

8 bits register (al, ah...), 16 bits register (ax , ...), 32 bits register(eax , ...), flag register (flags), segment register(cs , ds , ...), ...

(2) Expression Like:

6*[3+2]-2 !result=28

(3) (register)

The value of register is the operand's address, and instruction will get the operand from that address. (Only esp and ebp's default addressing segment are ss register, all the others are ds register.)

(4) (expression)

The value of expression is the operand's address, and instruction will get the operand from that address.

(5) expression (register)

The sum of expression and register is the operand's address, and instruction will get the operand from that address.

(6) expression (register*SCALE)

The sum of expression and register*SCALE is the operand's address, and instruction will get the operand from that address. The SCALE must be one of 2, 4 or 8.

(7) expression (register1)(register2)

The sum of expression and register1 and register2 is the operand's address, and instruction will get the operand from that address. The register1 is the base register, the default segment of this operand is the default segment of the register1.

(8) expression (register1)(register2*SCALE)

The sum of expression and register1 and register2*SCALE is the operand's address, and instruction will get the operand from that address. The register1 is the base register, the default segment of this operand is the default segment of the register1.

3.8 Instruction

Any instruction printed in Intel 80x86 assembly language manual can be used. Using comma"," to separate two operands. If you write two statements on one line, you must use semicolon”;” to separate them. If a statement is too long to be written in one line, Using backlash”\” to indicate the next line is follow this line and do not begin a new line.

3.9

(1) .extern

Declare a global variable or function defined in other files and can use them in this file.

(2) .define

Declare a global variable or function defined in this files and can be referenced by other files.

(3) .data .data2 .data4

.data

Define a one-byte data.

.data2

Define a two-byte data.

.data4

Define a four-byte data.

(4) .ascii .asciz

.ascii define a character string.

.asciz define a character string, and add a 0 at the end of this string.

(5) .align NUM

Make the compiler put the things on the integral multiple address.

(6) .space NUM

Define a memory space which initialized contain NUM bytes zero.

(7) .comm VAR NUM

Define a variable VAR and initialized it to NUM bytes zero.

(8) .sect XXXX

Define a section, named “XXXX”.

3.10 Macro

It is used as the same as the macro used in ANSI C language.

4. Interchange between assembly and C code

4.1 call ANSI C function from assembly language code

First The C function must be declared by .extern. If that C function has parameters, you must push the parameters on the stack before call the function. After you return from the function the eax register contain the return value of the C function and you must resume the stack to continue. And you must add an underline (_) at the beginning of the C function name when you call it. Like:

```
(in C code file:)
extern int c_function(int a, int b, int c);
(in assembly code)
push c
push b
push a
call _c_function
add  esp,3*4
```

4.2 call assembly language function from ANSI C code

First you must declare this assembly language function to make it can be used outside its file. Then you must add an underline at the beginning of the assembly language function, which want to be used outside, remove this underline when you call it from the C language code. Like:

```
(in assembly code)
.define _s_function
```

```
  _s_function:
    !get the parameters
  (in C code file:)
  s_function(a,b,c)
```

5. EXAMPLES

5.1 example one

```
! Begin of example1 assembly code
.sect .text; .sect .rom; .sect .data; .sect .bss
.define _main
.sect .text
_main:
    mov eax,msg !get and pass the parameter to the C function
    push    eax
    call _printf !call C function add a _
    add esp,4 !resume the stack

    xor eax,eax !pass parameter to exit function
    push eax
    call    _exit
.sect .data
msg: .ascii "Be happy in Minix!!\n"
! End of example1 assembly code
```

5.2 example two

```
! Begin of example2 assembly language code
.sect .text; .sect .rom; .sect .data; .sect .bss
.define _s_f !the function and variables in this file
.defiane _D4
.define _D5
.define _D6
.extern _c_pchar !the function outside will be used
.extern _c_pint

.sect .text
_s_f:
  !get the C functions parameters, it is not a good method
  !why put the parameters in data section, why not just put
  !it in registers? Because when you call C functions
  ! Registers will be used and your data will lose.
  mov edx,esp
  add esp,4
```

```
  pop eax
  mov (D1),eax !push data to the address of D1
  pop eax
  mov (D2),eax
  pop eax
  mov (D3),eax

  !resume the stack
  mov esp,edx

  push (D1)
  call _c_pchar
  add esp,4

  push (D2)
  call _c_pint
  add esp,4

  push (D3)
  call _printf
  add esp,4

  ! Use the variable assigned in C code
  push (_D4)
  call _c_pchar
  add esp,4

  push (_D5)
  call _c_pint
  add esp,4

  push (_D6)
  call _printf
  add esp,4

  ret

.sect .data
D1: .data4 0 !define D1 and alloce it 4 bytes
D2: .data4 0
D3: .data4 0
_D4: .data4 0 !must add underline
_D5: .data4 0
_D6: .data4 0
```

```
! End of example2 assembly code

/* Begin of the example2 C code */
#include <stdio.h>

extern int s_f(char p1, int p2, void *p3);
extern int c_pchar(char c);
extern int c_pint(int d);
extern char D4;    /* tell the C compiler what these data are */
extern int D5;
extern void *D6; /* MUST use void*, otherwise will get error at runtime */

int main (void)
{
  char p1= 'A';
  int p2=44;
  void *p3 = (void *)"Be happy in Minix!\n";

/* you see, I assign the variable define in assembly code */
  D4 = 'B';
  D5 = 77;
  D6 = (void *)"Happy Chinese NEW YEAR!\n";

  s_f(p1,p2,p3);

return 0;
}

int c_pchar(char c)
{
  printf("%c\n", c);
  return 0;
}

int c_pint(int d)
{
  printf("%d\n", d);
  return 0;
}
/* End of the example2 C code */
```

----------End of Manual----------

Wangzhi

PostGraduate Of Wuhan University of Technology (China)

Research direction: Operating System And Network

E-mail: quakewang@mail.whut.edu.cn

2004.5.19