

## Extending nasm – tratto dall'asmj

Pubblicato da *mammon\_* il 03/1999

Livello *intermedio*

### Introduzione

/\* Questo articolo e' stato tratto dall'Assembly Programming Journal numero 3 e tradotto da Little-John \*/

```

::/ \:.....
:/___\:.....
/|   \:.....
:|   _/\:.....
:|  _ | \:.....
::\___\:.....THE.UNIX.WORLD
    
```

### Iniziamo

I programmatori che passano a NASM da un assembler commerciale come MASM o TASM notano immediatamente la mancanza di qualsiasi struttura di linguaggio di alto livello — la sintassi assembly accettata dal NASM e' solo leggermente piu' sofisticata di quella che potremmo trovare in un debugger.

Mentre cio' ha un lato positivo — la dimensione del codice inferiore, e nulla e' nascosto al programmatore — rende la programmazione un po' piu' noiosa.

Per questo motivo NASM e' corredato di un preprocessor che e' al contempo semplice e potente; scrivendo delle macro NASM, le funzioni di alto livello degli altri assembler possono essere emulate molto facilmente. Come le macro seguenti dimostreranno, le funzioni di alto livello dei comuni assembler commerciali non fanno nulla di speciale; sono solo piu' convenienti per il programmatore.

Le macro che esporro' piu' giu' forniscono alcuni costrutti di base in C e ASM da usare in NASM. Il file completo e' disponibile all'indirizzo <http://www.eccentrica.org/Mammon/macros.asm> Il file macro puo' essere incluso in un file .asm con la direttiva `NASM %INCLUDE "macros.asm"` I commenti sull'uso di ogni macro sono inclusi nel file.

#### Le Basi delle Macro

La struttura fondamentale di una macro NASM e'

```

%macro {nomemacro} {# parametri}
%endmacro
    
```

Il codice e' contenuto tra i tags `%macro` e `%endmacro`; questo codice sara' inserito nel tuo programma ogniqualvolta NASM trova `{nomemacro}`. Quindi puoi creare una macro per pushare i contenuti di ogni registro come:

```

%macro SAVE_REGS 0
push eax
push ebx
push ecx
push edx
%endmacro
    
```

Una volta che hai definito questa macro, puoi usarla nel tuo codice cosi':

```

SAVE_REGS
call ReadFile
    
```

...che il preprocessor espandera' come

```

push eax
push ebx
push ecx
push edx
call ReadFile
    
```

prima di assemblare. Dovrebbe essere notato che il preprocessing avviene in un unico passo giusto prima che inizi il compiling; per vedere in anteprima cosa il preprocessor mandera' all'assemblatore, puoi lanciare nasm con l'opzione `-e`.

Il tag `%macro` richiede che tu dichiari il numero dei parametri che saranno passati alla macro. Cio' puo' essere un singolo numero oppure un intervallo, con qualche trucco:

```

%macro LilMac 0 ; takes 0 arguments
%macro LilMac 5 ; takes 5 arguments
%macro LilMac 0-3 ; takes 0-3 arguments
%macro LilMac 1-* ; takes 1 to unlimited arguments
%macro LilMac 1-2+ ; takes 1-2 arguments
%macro LilMac 1-3 0, "OK" ; takes 1-3 arguments, 2-3 default to 0 & "OK"

```

Gli ultimi tre esempi necessitano di qualche spiegazione. L'operatore "-" nel tag di %macro specifica che la macro puo' gestire un qualsiasi numero di parametri; in altre parole, non c'e' un numero massimo, e il minimo e' il numero alla sinistra dell'operatore "-". L'operatore "+" significa che ogni parametro addizionale sara' aggiunto all'ultimo parametro invece di generare un errore, quindi:

```
LilMac 0, OK, Questo parametro e' di troppo
```

sara' gestito in modo tale che il primo argomento e' 0 e il secondo sara' "OK, Questo parametro e' di troppo".i  
Nota che questo e' un buon metodo per passare le virgole come parti di un argomento (normalmente sono solo separatori).  
Dando il numero di default di argomenti permette alla macro di essere chiamata con un numero inferiore di argomenti di quanti se ne aspettati.

```
%macro SAVE_VARS 1-4 ecx, ebx, eax
```

riempira' il quarto argomento mancante con eax, il terzo con ebx, e il secondo con ecx.  
Nota che devi passare gli argomenti di default cominciando dall'ultimo e lavorando all'indietro.

I parametri per le macro sono disponibili come %1 per il primo argomento, %2 per il secondo, e cosi' via, con %0 contenente il conto di tutti gli argomenti.  
C'e' un equivalente del comando "SHIFT" DOS chiamato %rotate che ruotera' i parametri verso sinistra o verso destra a seconda che un numero positivo o negativo sia stato fornito:

```

Prima : %1 %2 %3 %4 Prima : %1 %2 %3 %4 Prima : %1 %2 %3 %4
%rotate 1 %rotate -1 %rotate 2
Dopo : %4 %1 %2 %3 Dopo : %2 %3 %4 %1 Dopo : %3 %4 %1 %2

```

Cosi' la rotazione di 1 mettera' il valore di %1 in %4, e la rotazione di -1 mettera' il valore di %1 in %2.

Chiamate ad Alto Livello (High-level calls)

-----  
Puo' darsi che la lamentela piu' grande riguardo a NASM e' il la sua primitiva sintassi di chiamata. In MASM e TASM, i parametri per una call possono seguire la call stessa:

```
call MessageBox, hOwner, lpszText, lpszTitle, fuStyle
```

dove invece in NASM i parametri devono essere pushati nello stack prima della call:

```

push fuStyle
push lpszTitle
push lpszText
push hOwner
call MessageBox

```

Usando il parametro "-" con la direttiva %rep e' facile replicare una high-level call:

```

%macro call 2-*
%define _func %1
%rep &0-1
%rotate 1
push %1
%endrep
call _func
%endmacro

```

La direttiva %define definisce semplicemente la variabile \_func [gli underscore dovrebbero fare da prefisso ai nomi delle variabili nelle macro cosicche', per errore, non usi lo stesso nome piu' tardi nel programma] come %1, il nome della funzione da chiamare. Le direttive %rep e %endrep racchiudono le istruzioni da ripetere, e %rep prende per parametro il numero delle ripetizioni [in questo caso setta il numero dei parametri della macro meno 1].  
Quindi, la macro sopra cicla attraverso gli argomenti da chiamare e li pusha, l'ultimo argomento per primo [sintassi C] prima di

eseguire la call.

Fare l'overloading di istruzioni esistenti, come una call, causera' dei warnings al momento della compilazione [ricorda, il preprocessor pensa che tu stia facendo una chiamata ricorsiva a macro] quindi spesso vorrai nominare la macro "c\_call" o qualcosa di simile.

Le macro seguenti provvedono degli aiuti per compiere chiamate in stile C, Pascal, fastcall, e stdcall.

```
;=====High-Level Call
; ccall FuncName, param1, param2, param 3... ;Pascal: primo-primo, non pulisce
; pcall FuncName, param1, param2, param 3... ;C: Ultimo-primo,pulisce lo stack
; stdcall FuncName, param1, param2, param 3... ;StdCall: Ultimo-primo, non pulis
; fastcall FuncName, param1, param2, param 3... ;FastCall: registri/stack
%macro pcall 2-*
%define _j %1
%rep %0-1
%rotate -1
push %1
%endrep
call _j
%endmacro

%macro ccall 2-*
%define _j %1
%assign __params %0-1
%rep %0-1
%rotate -1
push %1
%endrep
call _j
%assign __params __params * 4
add esp, __params
%endmacro

%macro stdcall 2-*
%define _j %1
%rep %0-1
%rotate -1
push %1
%endrep
call _j
%endmacro

%macro fastcall 2-*
%define _j %1
%assign __pnum 1
%rep %0-4
%rotate -1
%if __pnum = 1
mov eax, %1
%elif __pnum = 2
mov edx, %1
%elif __pnum = 3
mov ebx, %1
%else
push %1
%endif
%assign __pnum __pnum+1
%endrep
call _j
%endmacro
;=====END
```

## Blocchi Switch-Case (Switch-Case Blocks)

Uno dei piu' scomodi costrutti C da codificare in assembly e' il blocco SWITCH-CASE. E' anche abbastanza difficile ripeterlo come macro per via del numero variabile e della lunghezza dello statement CASE.

Il preprocessor di NASM ha uno stack di contesto che ti permette di creare un set di variabili locali e indirizzi che e' specifico per una particolare chiamata di una macro.

Quindi diventa possibile far riferimento a labels che saranno create in una prossima macro, dando loro nomi che dipendano dal contesto (context-dependent names)

```
%macro MacPart1 0
```

```
%push mac ;crea un contesto chiamato "mac"
jmp %$loc ;salto alla context-specific label "loc"
%endmacro
```

```
%macro MacPart2 0
%ifctx mac ;se siamo nel contesto 'mac'
%$loc: ;definisci la label 'loc'
xor eax, eax ;codice in questa label...
ret
%endif ;chiudi il blocco if
%pop ;distrugge il contesto 'mac'
%endmacro
```

Come puoi vedere, il contesto e' creato e denominato con una direttiva %push, e distrutto con una direttiva \$pop. NASM ha un certo numero di statement condizionali IF/ELSE a livello di preprocessor; nell'esempio sopra, la direttiva %ifctx [if current context equals, se il contesto attuale uguaglia] e' usata per determinare se un contesto 'mac' e' stato creato [Nota che i condizionali NASM di 'base' includono %if, %elif, %else, e %endif; questi potano poi alle direttive %ifctx, %ifnctx, %elifctx, %elifnctx, %else, e %endif; tutte le direttive %if directives devono essere chiuse con una direttiva %endif]. Infine, %\$ e' utilizzato come prefisso del nome di una variabile o label context-specific.

Le labels locali non-context-specific usano il prefisso %%:

```
%macro LOOP_XOR
%%loop:
pop eax
xor eax, ebx
test eax, eax
jnz %%loop
%endmacro
```

La macro SWITCH-CASE che segue usa la sintassi:

```
SWITCH Variable
CASE Int
BREAK
CASE Int
BREAK
DEFAULT
ENDSWITCH
```

Che puo' essere implementata come segue:

```
card db 0 ;card_variable
Jack EQU 11
Queen EQU 12
King EQU 13
...
SWITCH card
CASE Jack
add edx, Jack
BREAK
CASE Queen
add edx, Queen
BREAK
CASE King
add edx, King
BREAK
DEFAULT
add d, [card]
ENDSWITCH
```

Nota che SWITCH muove la variabile in eax e CASE in ebx.

```
;=====SWITCH-CASE Blocks
%macro SWITCH 1
%push switch
%assign __curr 1
mov eax, %1
jmp %$loc(__curr)
%endmacro

%macro CASE 1
```

```

%ifctx switch
%$loc(__curr):
%assign __curr __curr+1
mov ebx, %1
cmp eax, ebx
jne %$loc(__curr)
%endif
%endmacro

%macro DEFAULT 0
%ifctx switch
%$loc(__curr):
%endif
%endmacro

%macro BREAK 0
jmp %$endswitch
%endmacro

%macro ENDSWITCH 0
%ifctx switch
%$endswitch:
%pop
%endif
%endmacro
;=====END

```

### Blocchi If-Then (If-Then Blocks)

Mentre il preprocessor fornisce supporto per le direttive if-then, c'è un po' di lavoro da compiere per generare l'equivalente in linguaggio assembly del codice 'if' [l'if del preprocessor è svolto prima della compilazione, non in esecuzione].

Usando le macro, puoi creare blocchi if-then con la struttura seguente:

```

IF Value, Cond, Value
;codice if qui
ELSIF Value, Cond, Value
;codice else-if qui
ELSE
;codice else qui
ENDIF
Per esempio:
IF [Passwd], e, [GoodVal] ;e == uguale o je
jmp Registered
ELSE
jmp FormatHardDrive
ENDIF

```

La parte più ingegnosa di questa sequenza macro è il parametro 'Cond'. NASM permette codici di condizionamento (condition codes) [cc in 'jcc' che trovi nelle reference degli opcodes] da esser passati alla macro; questi codici di condizionamento sono semplicemente il 'jcc' senza 'j' — 'jnz' diventa 'nz', 'jne' diventa 'ne', 'je' diventa 'e', e così via.

La ragione di ciò è che il codice di condizionamento è apposto ad una 'j' dopo nella macro:

```

%macro Jumper %1 %2 %3 ;JUMPER Reg1, cc, Reg2
cmp %1, %3
j%+2 Gotcha
jmp error
%endmacro

```

Il codice sopra appone %2 alla 'j' con la direttiva j%+2. Nota che se tu usi j%- invece di j%+, NASM inserirà il codice di condizionamento \*inverso\*, sì che jz diventi jnz, ecc. Per esempio, chiamando la macro

```

%macro Jumper2 %1
j%-1 JmpHandler
%endmacro

```

con l'invocazione 'Jumper2 nz' dovrebbe assemblare il codice 'jz JmpHandler'.

I codici di condizionamento potrebbero essere un po' insidiosi da usare; è consigliabile aggiungere una sequenza come la seguente al file di macro:

```

#define EQUAL e

```

```

#define NOTEQUAL ne
#define G-THAN g
#define L-THAN l
#define G-THAN-EQ ge
#define L-THAN-EQ le
#define ZERO z
#define NOTZERO nz

```

cosi' che puoi chiamare la macro IF come segue:

```

IF PassWd, EQUAL, GoodVal
;codice if qui

```

...ecc ecc. Nota che anche le macro IF-THEN-ELSE macros mettono i valori passati in eax e ebx per il confronto, quindi questi registri devono essere preservati.

```

;=====IF-THEN-ELSE

Loops
%macro IF 3
%push if
%assign __curr 1
mov eax, %1
mov ebx, %3
cmp eax, ebx
j%+2 %%if_code
jmp %$loc(__curr)
%%if_code:
%endmacro

%macro ELSIF 3
%ifctx if
jmp %$end_if
%$loc(__curr):
%assign __curr __curr+1
mov eax, %1
mov ebx, %3
cmp eax, ebx
j%+2 %%elsif_code
jmp %$loc(__curr)
%%elsif_code:
%else
%error "'ELSIF' can only be used following 'IF'"
%endif
%endmacro

%macro ELSE 0
%ifctx if
jmp %$end_if
%$loc(__curr):
%assign __curr __curr+1
%else
%error "'ELSE' can only be used following an 'IF'"
%endif
%endmacro

%macro ENDIF 0
%$loc(__curr):
%$end_if:
%pop
%endmacro
;=====END

```

### Cicli Fo/While (Fo/While Loops)

Il DO...FOR e DO...WHILE non fanno alcunché di diverso dalla macro precedente, ma sono soltanto una applicazione diversa degli stessi principi.

La sintassi per chiamare queste macro è:

```

DO
;codice da eseguire qui
FOR min, Cond, max, step

```

```
DO
; codice da eseguire qui
WHILE variable, Cond, value
```

E' comunque molto piu' facile illustrare cio' confronto la macro con il codice C.

```
for ( x = 0; x <= 100; x++) { SomeFunc() }
E' uguale a:
DO
call SomeFunc
FOR 0, 1, 100, 1
Altrettanto,
for( x = 0; x != 100; x--) { SomeFunc() }
E' uguale a:
DO
call SomeFunc
FOR 0, e, 100, -1
La macro WHILE e' simile:
while( CurrByte != BadAddr) {SomeFunc() }
E' uguale a:
DO
call SomeFunc
WHILE CurrByte, ne, BadAddr
```

Ancora una volta, eax e ebx sono usati nelle macroofe WHILE.

```
;=====DO-FOR e DO-WHILE
Loops
%macro DO 0
%push do
jmp %$init_loop
%$start_loop:
push eax
%endmacro

%macro FOR 4
%ifctx do
pop eax
add eax, %4
cmp eax, %3
j%-2 %%end_loop
jmp %$start_loop
%$init_loop:
mov eax, %1
jmp %$start_loop
%%end_loop:
%pop
%endif
%endmacro

%macro WHILE 3
%ifctx do
pop eax
mov ebx, %3
cmp eax, ebx
j%+2 %%end_loop
jmp %$start_loop
%$init_loop:
mov eax, %1
jmp %$start_loop
%%end_loop:
%pop
%endif
%endmacro
;=====END
```

#### Dichiarazione di dati (Data Declarations)

Dichiarare i dati e' relativamente facile in assembler, ma a volte aiuta rendere il codice piu' chiaro se crei macro che assegnano tipi di dati significativi alle variabili, anche se quelle macro si risolvono in semplici DB o DD.

La macro seguente dimostra il concetto. Questi sono invocati come segue:

```
CHAR Name, String ;e.g. CHAR UserName, "Joe User"
INT Name, Byte ;e.g. INT Timeout, 30
WORD Name, Word ;e.g. WORD Logins
DWORD Name, Dword ;e.g. DWORD Password
```

Nota che quando invocate con un nome ma non con un valore, queste macro creano variabili vuote [DB 0].

```

;-----Data Declarations
%macro CHAR 1-2 0
%1: DB %2,0
%endmacro

%macro INT 1-2 0
%1: DB %2
%endmacro

%macro WORD 1-2 0
%1: DW %2
%endmacro

%macro DWORD 1-2 0
%1: DD %2
%endmacro
;-----END
```

### Dichiarazione di Procedure (Procedure Declarations)

Le dichiarazioni delle procedure sono un'altra questione di convenienza. E' spesso utile nel tuo codice delineare chiaramente l'inizio e la fine di una procedura; ognuna delle macro PROC sotto fa cio', come anche creare un frame di stack per la procedura. La macro ENTRYPROC macro crea una procedura chiamata 'main' e dichiara main come un simbolo globale (global symbol); il PROC standard dichiara il nome dato come globale. Queste macro possono essere usate come segue:

```
PROC ProcName Parameter1, Parameter2, Parameter3
;codice della procedura qui
ENDP

ENTRYPROC
; codice della entry-procedure qui
ENDP
```

Nota che i parametri (Parameters) alla PROC sono settati con EQU agli offset negativi da ebp, p.e. ebp-4, ebp-8, ecc. Ho anche incluso il supporto per le variabili locali, che con EQU sono settate agli offset positivi da ebp, questi potrebbero essere usati come segue:

```
PROC ProcName Parameter1, Parameter2, Parameter3...

LOCALDD Dword_Variable
LOCALDW Word_Variable
LOCALDB Byte_Variable
;codice della procedura qui
ENDP

;-----Procedure Declarations
%macro PROC 1-9
GLOBAL %1
%1:
%assign _i 4
%rep %0-1
%2 equ [ebp-_i]
%assign _i _i+4
%rotate 1
%endrep
push ebp
mov ebp, esp
%push local
%assign __ll 0
%endmacro

%macro ENDP 0
%ifctx local
%pop
```

```

%endif
pop ebp
%endmacro

%macro ENTRYPROC 0
PROC main
%endmacro

%macro LOCALVAR 1
sub esp, 4
%1 equ [ebp + __ll]
%endmacro

%macro LOCALDB 1
%assign __ll __ll+1
LOCALVAR %1
%endmacro

%macro LOCALDW 1
%assign __ll __ll+2
LOCALVAR %1
%endmacro

%macro LOCALDD 1
%assign __ll __ll+4
LOCALVAR %1
%endmacro
;=====END

```

## **Conclusioni**

### Ulteriori Estensioni (Further Extension)

---

Sperimentazioni continue saranno, naturalmente, fruttuose. E' raccomandabile che tu legga/stampi il capitolo 4 del manuale di NASM come reference.

Inoltre, e' vantaggioso testare le tue macro compilando il source con "nasm -e", che mandera' in output il sorgente preprocessato a stdout e non compilara' il programma.