

# Seminar 6

Rechnerarchitektur

# Multi-Modul Programmierung – Schritt 1

- Multi-Modul: das Programm besteht aus mehreren Modulen (Dateien)
- Man schreibt mehrere Assembler Programme: module1.asm, module2.asm ... moduleN.asm
- Diese werden dann getrennt assembliert mithilfe von ***nasm***:

```
nasm.exe -fobj module1.asm
```

```
.....
```

```
nasm.exe -fobj moduleN.asm
```

- Der Assembler übersetzt und prüft das geschriebene Programm. Wenn es keine Fehler enthält, erzeugt er eine Zwischendatei, die sogenannte ***Objektdatei***. Diese enthält schon den erzeugten Maschinencode aber noch nicht die richtigen Adressen für Unterprogramme und Sprungmarken.

# Multi-Modul Programmierung – Schritt 2

- Um auch das Zusammenbinden mehrerer Assembler- und Hochsprachenprogramme zu ermöglichen, gibt es den Binder (Linker). Im nächsten Schritt wird also der Linker aufgerufen, um alle Objektfiles zusammen zu binden (linken):

```
alink.exe -oPE -subsys console -entry start module1.obj module2.obj ...  
moduleN.obj
```

- Danach erzeugt der Linker eine ausführbare Datei (executable):  
**module1.exe**
- Um das Programm zu debuggen kann man jetzt Ollydebugger benutzen:  
**ollydbg module1.exe**

# Multi-Modul Programmierung

- Ein Modul enthält das **Hauptprogramm** (main program) und alle **andere Modules beschreiben Funktionen/Prozeduren**, die von dem Hauptprogramm aufgerufen werden
- Man kann in einem Modul Daten (Variablen, Prozeduren) benutzen, die in einem anderen Modul definiert wurden falls Folgendes gilt:
  - in dem Modul, in dem sie definiert wurden, werden sie als **GLOBAL** deklariert
  - In einem anderen Modul, in dem sie benutzt werden, werden sie als **EXTERN** deklariert
- Bem. Konstante (**equ**) können nicht exportiert werden, da sie keine Adresse in dem Speicherplatz haben.
- Der Einstiegspunkt des Programms (**entry point**) wird in einem einzigen Modul, in dem Hauptprogramm, definiert

# Parameter Übergabe zwischen Module

- Es gibt drei Möglichkeiten:
  - I. Übergabe von Parameter kann über **Register** erfolgen
    - **Problem:** die Anzahl der Register ist begrenzt und manche können besetzt sein (Daten enthalten, die man noch braucht)
  - II. Parameter können übergeben werden, indem sie als **global** definiert werden
    - **Problem 1:** man verletzt ein wichtiges Prinzip der Programmierung: Modularisierung
    - Module sollten einzeln und unabhängig voneinander programmiert und getestet werden.
    - Man benutzt Modularisierung damit man nicht alles in dem Hauptprogramm schreibt, damit Programme klar strukturiert werden und damit man Programmteilen (Module) wiederverwenden kann.
    - **Problem 2:** wenn viele Variablen als global deklariert werden, dann kann das zu Namenkonflikten führen (dasselbe Symbol wird an verschiedenen Stellen definiert)
  - III. Übergabe von Parameter kann über dem **Stack** erfolgen
    - Wird am meisten benutzt, weil diese Methode flexibel ist

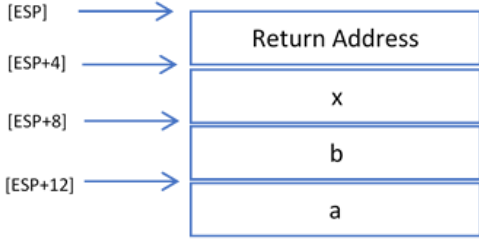
# Übergabe von Parameter über Register

Module main.asm	Module function.asm
<pre>bits 32 global start extern exit import exit msvcrt.dll ; we import the <i>addition</i> function from the function.asm module extern addition segment data use32 class=data public     a db 2     b db 3     x db 0 segment code use32 class=code public start:     ; put the parameters in registers     mov bl, [a]     mov bh, [b]     ; call the function     call addition     ; result is in AL     mov [x], al      ; call exit(0)     push dword 0     call [exit]</pre>	<pre>bits 32 ; we export the <i>addition</i> function in order to be used in the main module global addition  segment code use32 class=code public ; the code segment contains only the addition function addition:     ; the parameters are in: BL=a, BH=b     ; we will return the result in AL     mov al, bl     add al, bh      ; return from function     ; (it removes the Return Address from the stack and jumps to the Return Address)     ret</pre>

# Übergabe von Parameter über globale Variablen

Module main.asm	Module function.asm
<pre>bits 32 global start extern exit import exit msvcrt.dll ; we import the <i>addition</i> function from the function.asm module extern addition ; we export variables a, b and x in order to be used ; in the other module global a global b global x  segment data use32 class=data public     a db 2     b db 3     x db 0 segment code use32 class=code public start:     ; there is no need to do anything with the parameters. They are     ; already accessible to the other module (because they are global).     ; call the function     call addition     ; the result is already placed in x by the addition function      ; call exit(0)     push dword 0     call [exit]</pre>	<pre>bits 32 ; we export the <i>addition</i> function in order to be used in the main module global addition  ; import the a, b, x variables from the other module extern a, b, x  segment code use32 class=code public ; the code segment contains only the addition function addition:     ; the parameters are directly accessible in global     ; variables a, b and x (which are global)     mov al, [a]     add al, [b]     mov [x], al      ; return from function     ; (it removes the Return Address from the stack and jumps to the Return Address)     ret</pre>

# Übergabe von Parameter über dem Stack

Module main.asm	Module function.asm
<pre>bits 32 global start extern exit import exit msvcrt.dll ; we import the <i>addition</i> function from the <i>function.asm</i> module extern addition segment data use32 class=data public     a db 2     b db 3     x db 0 segment code use32 class=code public start:     ; put the parameters a, b and x on the stack     mov eax, 0    ; we can not put bytes on the stack, we will put <b>doublewords</b>!     mov al, [a]     push eax     mov al, [b]     push eax     mov al, [x]     push eax     ; call the function     call addition     ; the result is in the dword from the top of the stack     pop eax     mov [x], al    ; x := a + b     ; we still have to remove the two parameters from stack     add esp, 4*2     ; instead of the above instruction we could have used two 'pop eax' instructions     ; call exit(0)     push dword 0     call [exit]</pre>	<pre>bits 32 ; we export the <i>addition</i> function in order to be used in the main module global addition segment code use32 class=code public ; the code segment contains only the addition ; function addition:     ; the parameters are on the stack     ; the stack looks like this:      ; remember that a stack element is 4 bytes (doubleword) and the stack grows toward smaller     ; addresses (meaning that the dword from the top of the stack is placed at the smallest memory     ; address).     ; the Return Address was placed on the stack by the 'call addition' instruction in the main module.      mov eax, dword [esp+12]     mov dl, al    ; dl = a     mov eax, dword [esp+8]     add dl, al    ; dl = a + b     mov eax, 0     mov al, dl     mov dword [esp+4], eax    ; place a+b on the stack for the main module      ; return from function     ; ('ret' removes the Return Address from the top of the stack and jumps to the Return Address)     ret</pre> <div></div>



# Bemerkung!

- Wenn man **include** benutzt, dann wird der Code aus einer Datei in der aktuellen Datei “kopiert”. Dadurch besteht aber das Programm **nicht aus mehreren Modulen**, sondern aus einem einzigen Modul, das in mehrere Dateien geschrieben wurde!

# Include ist nicht Multi-Modul!

Module main.asm	Module function.asm
<pre>bits 32 global start extern exit import exit msvcrt.dll ; we import the addition function from the  function.asm module ;extern addition – <b>we do not declare the function as extern, since it is in the same module!!</b>  %include 'function.asm' ; “anywhere” before start  segment data use32 class=data public     a db 2     b db 3     x db 0 segment code use32 class=code public start:     mov eax, 0     mov al, [a]     push eax     mov al, [b]     push eax     mov al, [x]     push eax     call addition     pop eax     mov [x], al        ; x := a + b     add esp, 4*2     push dword 0     call [exit]</pre>	<pre>%ifndef _FUNCTION_ASM_ %define _FUNCTION_ASM_ %endif ; no more code and data segment (this is a small part of the main file that already contains a data ; and a code segment  addition:     mov eax, dword [esp+12]     mov dl, al          ; dl = a     mov eax, dword [esp+8]     add dl, al          ; dl = a + b     mov eax, 0     mov al, dl     mov dword [esp+4], eax    ; place a+b on the stack for the main module     ; return from function     ; ('ret' removes the Return Address from the top of the stack and jumps to the Return Address)     ret</pre>

# Übungen

1. Schreibe ein Modul, das eine Funktion enthält, die zwei Zeichenketten verkettet (concatenate two strings). Schreibe ein Hauptprogramm, das die Funktion aus dem anderen Modul aufruft und das Ergebnis auf dem Bildschirm ausdruckt.
2. Schreibe ein Programm, das die Summe der Ziffern einer Zahl ausdruckt, wobei das Programm eine Funktion aus einem anderen Modul aufruft, welche diese Summe berechnet.
3. Schreibe ein Modul, das eine Funktion enthält, welche überprüft ob eine Zahl prim ist oder nicht. Schreibe ein Programm, das mithilfe dieser Funktion die prime Zahlen aus einer gegebenen Zahlenfolge ausdruckt.