**Curs 9**

**Programare multimodul în limbaj de asamblare**

Contents

[1. Programare modulară 2](#_Toc28948052)

[1. Modular Programmierung 2](#_Toc28948053)

[1.1 Cum se poate împărți o problemă în sub-probleme? 2](#_Toc28948054)

[1.1 Wie können Sie ein Problem in Teilprobleme unterteilen? 2](#_Toc28948055)

[1.2 Tehnici și instrumente 3](#_Toc28948056)

[1.2 Techniken und Werkzeuge 3](#_Toc28948057)

[1.2.1 Includerea statică la compilare/asamblare: directiva %include 3](#_Toc28948058)

[1.2.1 Statische Einbeziehung bei der Kompilierung / Assemblierung: die Direktive %include 3](#_Toc28948059)

[1.2.2 Legarea statică la linkeditare 7](#_Toc28948060)

[1.2.2 Statische Bindung zu verlinkten 7](#_Toc28948061)

[1.2.3 Cerințele NASM 11](#_Toc28948062)

[1.2.3 Cerințele NASM 11](#_Toc28948063)

[1.2.4 Folosirea în practică a directivelor global și extern 11](#_Toc28948064)

[1.2.4 Anwendung von die Direktiven global und extern in der Praxis 11](#_Toc28948065)

[1.2.5 Pașii necesari construirii programului executabil final 13](#_Toc28948066)

[1.2.5 Die zum Erstellen des endgültigen ausführbaren Programms erforderlichen Schritte 13](#_Toc28948067)

|  |  |  |
| --- | --- | --- |
| Prezentăm forma generală a unui program în NASM, însoţită de un scurt exemplu: |  | Wir präsentieren die allgemeine Form eines Programms in NASM, begleitet von einem kurzen Beispiel: |
|  | | |
| 1. Programare modulară |  | 1. Modular Programmierung |
| 1.1 Cum se poate împărți o problemă în sub-probleme? |  | 1.1 Wie können Sie ein Problem in Teilprobleme unterteilen? |
| Modularizarea este un concept care are sensuri diferite în funcţie de nivelul de abstractizare la care este aplicat:   * La nivelul programelor, modularizarea constă în crearea de unități logice; * La nivelul codului sursă (al unităților), modularizarea constă în crearea de fișiere distincte; * La nivelul fișierelor, modularizarea constă în crearea de subrutine. |  | Modularisierung ist ein Konzept, das je nach Abstraktionsebene, auf die es angewendet wird, unterschiedliche Bedeutungen hat: • Auf Programmebene besteht die Modularisierung in der Erstellung logischer Einheiten; • Auf der Quellcode-Ebene (der Einheiten) besteht die Modularisierung darin, separate Dateien zu erstellen;  • Auf Dateiebene besteht die Modularisierung aus der Erstellung von Unterprogrammen. |
| G:\didactic\ASC\Babes\Curs ASC OC\Cursuri\modularizare.jpg  Figura 1. *Conceptul de modularizare, la diferite niveluri de abstractizare* (Das Konzept der Modularisierung auf verschiedenen Abstraktionsebenen) | | |
| În momentul creării unei aplicaţii mai complexe, trebuie să ne întrebăm care sunt sub-problemele pentru care există deja rezolvări disponibile, astfel încât să nu rescriem cod în mod nenecesar.  *Reutilizarea codului* poate să se refere la următoarele aspecte:   * La nivelul fișierelor sursă, putem refolosi   + Cod sursă și date din asamblare   + Directiva %**include** * La nivelul fișierelor binare, putem refolosi   + Cod sursă și date din asamblare   + Cod și date din limbaje de nivel înalt   + Biblioteci   Observaţie: termenul „a refolosi” nu trebuie înţeles în sensul reutilizării unor porţiuni de cod scrise de noi, ci a reutilizării unor porţiuni de cod în general (chiar şi scrise de altcineva). |  | Bei der Erstellung einer komplexeren Anwendung müssen wir uns fragen, für welche Unterprobleme bereits Lösungen verfügbar sind, damit wir den Code nicht unnötig umschreiben.  *Die Wiederverwendung des Codes* kann sich auf folgende Aspekte beziehen:   * Auf Quelldateiebene können wir wiederverwenden * Quellcode und Assemblierungsdaten * Die Direktive %**include** * Auf der Ebene der Binärdateien können wir wiederverwenden * Quellcode und Assemblierungsdaten * Code und Daten auf Hochsprachen * Bibliotheken   Hinweis: Der Begriff „Wiederverwendung“ ist nicht in dem Sinne zu verstehen, dass einige von uns geschriebene Codeteile wiederverwendet werden, sondern dass einige Codeteile im Allgemeinen wiederverwendet werden (selbst wenn sie von jemand anderem geschrieben wurden). |
| 1.2 Tehnici și instrumente |  | 1.2 Techniken und Werkzeuge |
| 1.2.1 Includerea statică la compilare/asamblare: directiva %include |  | 1.2.1 Statische Einbeziehung bei der Kompilierung / Assemblierung: die Direktive %include |
| Această directivă este specifică limbajului de asamblare, dar are echivalent și în alte limbaje.  Este foarte important de subliniat faptul că modularizarea permite doar *divizarea codului* scris în limbajul respectiv (nu se pot combina coduri sursă scrise în mai multe limbaje).  Prin definiție, fiecare modul este o unitate de sine stătătoare şi grupează mai multe funcţii şi variabile înrudite.  Principalele avantaje ale programării modulare sunt:   * Abstractizare: de exemplu, în C fiecare modul conține o interfaţă (un *header*, extensia .h, .hpp sau fără nici o extensie cum e cazul header-ului din biblioteca standard cpp) şi o implementare (fişierul sursă). În interfaţă doar se *declară* funcţiile, iar *definirea* lor propriu zisă se realizează în fişierul sursă. Astfel, se asigură abstractizarea: se pune accentul pe elementele esenţiale (CE face modulul) şi se ignoră detaliile (CUM sunt implementate aceste funcţii). * Prin această *ascundere a implementării* în fişierul sursă se asigură *corectitudinea* modulului (utilizatorul nu poate să modifice codul sursă) şi *transparenţa* (se pot face modificări ulterioare asupra codului sursă şi utilizatorul să nu fie conştient de acestea dacă nu se modifică interfaţa funcţiilor). În plus, dacă eventual codul din implementare este furnizat în mod binar (de exemplu, o bibliotecă statică, fişier obiect etc.), utilizatorii (sau eventuali atacatori ai aplicaţiei) nu pot face *reverse engineering* ca să vadă codul sursă. În plus, utilizatorul modulului poate să utilizeze doar funcţiile expuse în interfaţă.   Totuși, utilizarea directivei **include** nu este programare multimodul autentică! **DE CE?**  Această metodă implică anumite riscuri importante.   1. Prin utilizarea ei se expune codul sursă!   Obţinerea unui fişier executabil trece prin 3 etape principale: preprocesare, compilare (asamblare) şi linkeditare. |  | Diese Direktive ist spezifisch für die Assemblersprache, hat jedoch eine Entsprechung in anderen Sprachen.  Es ist sehr wichtig zu betonen, dass die Modularisierung nur die *Aufteilung des geschriebenen Codes* in die jeweilige Sprache ermöglicht (es ist nicht möglich, in mehreren Sprachen geschriebene Quellcodes zu kombinieren).  Per Definition ist jedes Modul eine eigenständige Einheit und gruppiert mehrere Funktionen und zugehörige Variablen.  Die Hauptvorteile der modularen Programmierung sind:   * Abstraktion: Beispielsweise enthält in C jedes Modul eine Schnittstelle (ein *header*, eine Erweiterung mit der Endung .h, .hpp oder ohne eine Erweiterung wie die *header* der CPP-Bibliothek) und eine Implementierung (Quelldatei). In der Schnittstelle werden nur die Funktionen *deklariert* und in der Quelldatei eine eigene *Definition* realisiert. Somit ist die Abstraktion gewährleistet: Es wird Wert auf das Wesentliche gelegt (WAS stellt das Modul) und Details werden ignoriert (WIE diese Funktionen implementiert werden). * Diese *Verschleierung der Implementierung* in der Quelldatei stellt die *Korrektheit* des Moduls (der Benutzer kann den Quellcode nicht ändern) und die *Transparenz* sicher (spätere Änderungen am Quellcode können vorgenommen werden, und der Benutzer ist sich dessen nicht bewusst, wenn die Funktionsschnittstelle nicht geändert wird). Wenn der Code in der Implementierung in binärer Form bereitgestellt wird (z. B. eine statische Bibliothek, eine Objektdatei usw.), können Benutzer (oder mögliche Angreifer der Anwendung) den Quellcode nicht rückentwickeln (*reverse engineering*). Außerdem kann der Modulbenutzer nur die Funktionen verwenden, die in der Schnittstelle verfügbar sind.   Die Verwendung der Direktive **include** jedoch keine authentische Multimodulprogrammierung! **WARUM**? Diese Methode birgt bestimmte wichtige Risiken.  1. Durch die Verwendung wird der Quellcode verfügbar gemacht!  Das Abrufen einer ausführbaren Datei erfolgt in drei Hauptschritten: Vorverarbeitung, Kompilieren (Assemblierung) und Verknüpfen (*linkediting*). |
| Compile process  Figura 2. *Fazele obţinerii unui fişier executabil* (Die Schritte zum Abrufen einer ausführbaren Datei) | | |
| Mecanismul de preprocesare implică o concatenare textuală a fișierelor, astfel că se poate obţine un fişier sursă mult mai mare decât cel iniţial (care poate fi relativ scurt, dar care poate conţine multe „**include**”-uri). După etapa de preprocesare, un atacator are deja acces la tot codul sursă! În schimb, dacă am împărţi codul sursă în module care să fie asamblate separat, putem „transporta” cu noi doar fişierele .obj, pe care doar noi ştim cum se leagă de fişierul principal (deci putem de pildă transporta mai multe fişiere .obj, dintre care nu toate sunt utile aplicaţiei noastre). Este mult mai greu de făcut *reverse engineering* pe un fişier .obj decât pe un fişier sursă.  2. Expune cu vizibilitate globală toate denumirile de variabile, constante etc. Astfel apar conflicte la redefiniții / redeclarări. De exemplu, dacă avem două variabile cu acelaşi nume declarate în două module diferite, dacă *includem* aceste variabile în acelaşi fişier vom avea un conflict de nume. În schimb, dacă folosim variabile într-un context multi-modul, atunci nu va fi nici o problemă.  Prin directiva **include**, includem fișierul în întregime – și ce este util, și ce nu! |  | Der Vorverarbeitungsmechanismus beinhaltet die Verkettung von Dateien in Textform, sodass eine viel größere Quelldatei als die ursprüngliche Datei erhalten werden kann (die relativ kurz sein kann, aber viele „**include**“ enthalten kann). Ein Angreifer hat nach der Vorverarbeitung bereits Zugriff auf den gesamten Quellcode! Wenn wir stattdessen den Quellcode in Module aufteilen, die separat assemblieren werden sollen, können wir nur die .obj Dateien „mitnehmen“, die wir nur mit der Hauptdatei verknüpfen können (sodass wir beispielsweise mehr .obj-Dateien führen können, von denen nicht alle für unsere Anwendung nützlich sind). *Reverse Engineering* ist für eine .obj-Datei viel schwieriger als für eine Quelldatei.  2. Es macht mit globaler Sichtbarkeit alle Namen von Variablen, Konstanten usw. verfügbar. Somit ergeben sich Konflikte in den Redefinitionen / Redeklarationen. Wenn beispielsweise zwei Variablen mit demselben Namen in zwei verschiedenen Modulen deklariert sind und diese Variablen in derselben Datei *enthalten* sind, liegt ein Namenskonflikt vor. Im Gegensatz dazu gibt es kein Problem, wenn wir Variablen in einem Kontext mit mehreren Modulen verwenden.  Durch die **include**-Direktive wird die gesamte Datei eingeschlossen – und was ist nützlich und was nicht! |
| Figura 3. *Obţinerea fişierului executabil: faza de preprocesare* (Erhalten der ausführbaren Datei: Vorverarbeitungsphase) | | |
| 1) Exemplu de folosire **%include** |  | 1) Anwendungsbeispiel **%include** |
| ; fișierul constante.inc  ; gardă dublă-includere  **%ifndef** \_CONSTANTE\_INC\_ ; la prima includere, \_CONSTANTE\_INC\_ nu este definit  ; *Beim ersten Einschluss ist* \_CONSTANTE\_INC\_ *nicht definiert*  **%define** \_CONSTANTE\_INC\_ ; definim \_CONSTANTE\_INC\_ -> condiție falsă la viitoare includeri  ; *wir definieren \_CONSTANTE\_INC\_ -> falsche Bedingung für zukünftige Aufnahme*  ; se recomandă ca astfel de fișiere (incluse de către altele) să conțină (doar) declarații!  ; *Es wird empfohlen, dass solche Dateien (von anderen eingeschlossen) (nur) Deklarationen enthalten!*  MAX\_BYTE equ 0xFF  MAX\_WORD equ 0xFFFF  MAX\_DWORD equ 0xFFFFFFFF  MAX\_QWORD equ 0xFFFFFFFFFFFFFFFF  **%endif** ; \_CONSTANTE\_INC\_ | | |
| 2) Exemplu folosire **%include** – împachetare EAX într-un BYTE / WORD / DWORD, conform magnitudinii valorii acestuia |  | 2) Anwendungsbeispiel **%include -** EAX-Packung in einem BYTE / WORD / DWORD, abhängig von der Größe seines Wertes |
| ; fișierul program.asm  **%include** "constante.inc"  **cmp** EAX, MAX\_BYTE  **ja** .nu\_incape\_in\_octet ; încape valoarea din EAX într-un byte?  ; *Passt der Wert in EAX in ein Byte?*  .incape\_in\_octet:  **mov** [rezultat\_octet], AL ; dacă da, salvăm AL în rezultat\_octet  **jmp** .gata ; *Wenn ja, speichern wir AL in rezultat\_octet*  .nu\_incape\_in\_octet:  **cmp** EAX, MAX\_WORD  **ja** .nu\_incape\_in\_cuvant ; altfel verificăm dacă este suficient un WORD  ; *ansonsten prüfen wir, ob ein WORD ausreicht*  .incape\_in\_cuvant:  **mov** [rezultat\_word], AX ; dacă da, salvăm AX în rezultat\_word  **jmp** .gata ; *Wenn ja, speichern wir AX in rezultat\_word*  .nu\_incape\_in\_cuvant:  **mov** [rezultat\_dword], EAX ; dacă nu este suficient un WORD, salvăm întregul EAX  .gata: ; *Wenn ein WORT nicht ausreicht, speichern wir den gesamten EAX* | | |
| 1.2.2 Legarea statică la linkeditare Acesta este un pas realizat de către un linkeditor după asamblare/compilare. |  | 1.2.2 Statische Bindung zu verlinkten Dies ist ein Schritt, den ein Linker nach der Assemblierung / Kompilierung ausführt. |
| Figura 4. *Obţinerea fişierului executabil: faza de compilare sau asamblare* (Erhalten der ausführbaren Datei: Kompilierungs- oder Assemblierungsphase)    Figura 5. *Obţinerea fişierului executabil: faza de linkeditare* (Erhalten der ausführbaren Datei: Verlinktensphase) | | |
| Preprocesorul: transformă text în text.  Efectuează prelucrări asupra textului sursă, rezultând un text sursă intermediar. Se poate imagina ca fiind o componentă a compilatorului sau asamblorului.  Poate lipsi, multe limbaje nu au un preprocesor!  Asamblorul: transformă instrucțiunile (text) în codificare binară (fișier obiect).  Codifică instrucțiunile și datele (variabilele) din textul sursă preprocesat și construiește un fișier obiect care conține cod mașină și valori de variabile, alături de informații despre conținut (denumiri de variabile, subrutine, informații despre tipul și vizibilitatea acestora etc.).  Compilatorul: transformă instrucțiunile (text) în codificare binară (fișier obiect)  Identifică secvențe de instrucțiuni de procesor prin care se pot obține funcționalitățile descrise în textul sursă iar apoi, precum un asamblor, generează un fișier obiect care conține codificarea binară a acestora și a variabilelor din program.  *Asamblarea este un caz special de compilare, unde instrucţiunile de procesor sunt gata oferite direct în textul programului și ca atare nu necesită să fie alese de către compilator!*  Linkeditor: fișiere obiect => bibliotecă sau program  Construiește rezultatul final, adică un program (.exe) sau bibliotecă (.dll sau .lib) în care leagă împreună (include) codul și datele binare prezente în fișierele obiect.  Nu contează ce compilatoare sau ce limbaje de programare au fost folosite! Legarea necesită doar ca fișierele de intrare să respecte formatul standard al fișierelor obiect!  Legarea statică la linkeditare permite unirea mai multor module binare (fișiere obiect sau biblioteci statice) într-un singur fișier.  La intrări putem avea oricâte fișiere obiect (.OBJ) și / sau biblioteci statice (.LIB).  *Atenție, nu toate fișierele .LIB sunt biblioteci statice!*  La ieșiri se obţin fişiere .EXE sau .LIB sau .DLL (Dynamic-Link Library).  Programarea multimodul înseamnă că oricâte fișiere pot fi compilate separat și linkeditate împreună. Este un pas realizat de către linkeditor, după compilare / asamblare, deci nu depinde de limbaj!  *Reutilizarea codului*:   1. În formatul binar nu este expus codul sursă. 2. Permite inter-operabilitate între limbaje diferite!   Alte avantaje și dezavantaje:   1. Editorul de legături poate identifica și elimina resurse neutilizate sau efectua alte optimizări; 2. Dimensiune mare a programului: programul înglobează resursele externe reutilizate; 3. Dimensiune mare a programelor: bibliotecile populare duplicate în multe programe.   În NASM avem directivele **global** (mecanism export) și **extern** (mecanism import)   * **global nume** – oferă posibilitatea de utilizare din exterior a resursei respective, specificate prin nume * **extern nume** – solicitare de acces la resursa specificată; necesită să fie publică! |  | Präprozessor: Konvertiert Text in Text.  Es Führt eine Verarbeitung des Quelltextes durch, was zu einem Zwischenquelltext führt. Es kann als eine Komponente des Compilers oder Assemblers vorgestellt werden. Es kann fehlen, viele Sprachen haben keinen Präprozessor!  Assemblierer (Assembler): wandelt Anweisungen (Text) in binäre Codierung (Objektdatei) um.  Es codiert Anweisungen und Daten (Variablen) im vorverarbeiteten Quelltext und erstellt eine Objektdatei, die Maschinencode und Variablenwerte sowie Inhaltsinformationen (Variablennamen, Unterprogramme, Informationen zu Typ und Sichtbarkeit usw.) enthält.  Compiler: wandelt Anweisungen (Text) in binäre Codierung (Objektdatei) um.  Identifiziert Sequenzen von Prozessoranweisungen, über die die im Quelltext beschriebenen Funktionalitäten abgerufen werden können, und generiert dann als Assembler eine Objektdatei, die ihre binäre Codierung und Programmvariablen enthält.  *Assemblierung ist ein spezieller Kompilierungsfall, bei dem Prozessoranweisungen direkt im Programmtext bereitgestellt werden und als solche nicht vom Compiler ausgewählt werden müssen!*  Linkeditor: Objektdateien => Bibliothek oder Programm  Erstellt das Endergebnis, deshalb ein Programm (.exe) oder eine Bibliothek (.dll oder .lib), die den in den Objektdateien vorhandenen Binärcode und die darin enthaltenen Daten verknüpft (enthält).  Es ist egal, welche Compiler oder Programmiersprachen verwendet wurden! Das Binden setzt nur voraus, dass die Eingabedateien dem Standardformat der Objektdateien entsprechen!  Durch die statische Bindung an einen Link können Sie mehrere Binärmodule (Objektdateien oder statische Bibliotheken) in einer einzigen Datei zusammenführen.  Bei Einträgen können wir beliebige Objektdateien (.OBJ) und / oder statische Bibliotheken (.LIB) haben.  *Achtung, nicht alle .LIB-Dateien sind statische Bibliotheken!*  Die Ausgaben erhalten .EXE- oder .LIB- oder .DLL-Dateien (Dynamic Link Library).  Multimodale Programmierung bedeutet, dass jede Datei separat kompiliert und miteinander verknüpft werden kann. Dies ist ein Schritt, den der Link nach dem Kompilierung / Assemblierung ausführt, es kommt also nicht auf die Sprache an!  *Wiederverwendung von Code*:   1. Der Quellcode wird nicht im Binärformat angezeigt. 2. Ermöglicht die Interoperabilität zwischen verschiedenen Sprachen!   Andere Vor- und Nachteile:   1. Der Link-Editor kann nicht verwendete Ressourcen identifizieren und beseitigen oder andere Optimierungen durchführen. 2. Große Programmgröße: Das Programm enthält wiederverwendete externe Ressourcen 3. Große Programmgröße: Beliebte Bibliotheken, die in vielen Programmen doppelt vorhanden sind.   In NASM gibt es **global** (Exportmechanismus) und **extern** (Importmechanismus) Direktiven   * **global** **Name** – bietet die Möglichkeit der externen Verwendung der jeweiligen Ressource, angegeben durch den Namen * **extern Name** – Anforderung des Zugriffs auf die angegebene Ressource; muss öffentlich sein! |
| 1.2.3 Cerințele NASM Este foarte important de reţinut că resursele sunt partajate de comun acord.  Exportul se realizează prin directiva **global** nume1, nume2, ... Prin aceasta resursele respective se pun la dispoziţia oricărui fișier ar fi interesat.  Importul se realizează prin directiva **extern** nume1, nume2, ... Prin aceasta se solicită accesul la resursele respective, indiferent din ce fișier vor fi oferite.  **Solicitare fără disponibilitate = eroare!**  Nu se pot importa decât resurse care sunt exportate undeva. Însă disponibilitate fără solicitare este caz permis, deoarece chiar dacă nici un modul din program nu solicită / folosește resursele respective, poate ele vor fi utilizate într-o versiune viitoare sau de către un alt program.  Limbajele de programare de nivel mai înalt oferă și ele la rândul lor construcții sintactice cu rol echivalent (exemplu: în limbajul C, disponibilitatea este automată / implicită, putându-se însă opta pentru a bloca accesul prin folosirea cuvântului cheie **static**. Solicitarea de acces se face (tot) prin intermediul cuvântului cheie **extern**). |  | 1.2.3 Cerințele NASM Es ist sehr wichtig, sich daran zu erinnern, dass die Ressourcen in gegenseitigem Einvernehmen geteilt werden.  Der Export erfolgt über die Direktive **global** name1, name2, ... Hierdurch werden die jeweiligen Ressourcen für jede interessierte Datei zur Verfügung gestellt.  Der Import erfolgt über die Direktive **extern** name1, name2, ... Damit wird der Zugriff auf die jeweiligen Ressourcen angefordert, unabhängig davon, welche Datei angeboten wird.  **Anfrage ohne Verfügbarkeit = Fehler!**  Es können nur exportierte Ressourcen importiert werden. Eine Verfügbarkeit ohne Anfrage ist jedoch zulässig, da selbst wenn kein Modul im Programm die entsprechenden Ressourcen anfordert / verwendet, diese möglicherweise in einer zukünftigen Version oder von einem anderen Programm verwendet werden.  Höhere Programmiersprachen bieten auch syntaktische Konstruktionen mit gleicher Funktion (z. B. in der Sprache C ist die Verfügbarkeit automatisch / standardmäßig, Sie können den Zugriff jedoch mithilfe des Schlüsselworts **static** blockieren. Die Anforderung für den Zugriff lautet mache (alles) durch das **extern** Schlüsselwort). |
| 1.2.4 Folosirea în practică a directivelor global și extern |  | 1.2.4 Anwendung von die Direktiven global und extern in der Praxis |
|  | | |
|  | | |
| Exemplu de program multimodul NASM + NASM |  | Beispiel eines multimodul Programms NASM + NASM |
|  | | |
| 1.2.5 Pașii necesari construirii programului executabil final  1. Se asamblează fișierul main.asm |  | 1.2.5 Die zum Erstellen des endgültigen ausführbaren Programms erforderlichen Schritte  1. Assemblieren Sie die Datei main.asm |
| *nasm.exe -fobj* ***main.asm*** | | |
| 1. Se asamblează fișierul sub.asm |  | 1. Assemblieren Sie die Datei sub.asm |
| *nasm.exe -fobj* ***sub.asm*** | | |
| 1. Se editează legăturile dintre cele două module |  | 1. Die Verbindungen zwischen den beiden Modulen werden bearbeitet |
| *alink.exe main.obj sub.obj –oPE –entry:start –subsys:console* | | |
| Observații: cele două module pot fi asamblate în orice ordine! Abia în timpul linkeditării este necesar ca simbolurile referite să aibă, toate, implementare disponibilă în unul dintre fișierele obiect oferite linkeditorului.  Linkeditarea, în mod evident, este posibilă doar după asamblare / compilare! |  | Anmerkungen: Die beiden Module können in beliebiger Reihenfolge assemblieren werden! Nur während der Verlinkung ist es erforderlich, dass die referenzierten Symbole alle in einer der dem Verlinker angebotenen Objektdateien implementierbar sind.  Eine Verknüpfung ist natürlich nur nach Assemblierung / Kompilierung möglich! |