Assembler in C/C++ Programmen

Der Einsatz von Assembler kann bei speziellen Aufgaben durchaus sinnvoll sein. Auf einer PC-Plattform ist eine gemischtsprachliche Programmierung eher nicht mehr angebracht, weil die C-Compiler heute einen sehr effizienten Code erzeugen. Die Hardwarenähe, die klassische Domäne von Assembler, bringt in Win32-Systemen wenig, da in der Anwenderebene (Ring 3) keine Vorteile, wie I/O-Privileg, existieren. Zusammengefasst kann mit Assembler auf einer PC-Plattform normalerweise nicht mehr erreicht werden, als mit einem normalen C-Compiler.

In Mikrocontroller- oder Signalprozessoranwendungen ist der Einsatz von Assembler verbunden mit einer Hochsprache jedoch keine Frage. Manches kann in Assembler wesentlich einfacher und/ oder laufzeiteffizienter realisiert werden. In solchen Systemen werden die zeitkritischen Teile in Assembler codiert, der Rest in der Hochsprache.

Eine besondere Bedeutung hat aber die gemischtsprachliche Programmierung bei der Module in C/C++, Fortran, Visual Basic, u.a in einem Programmierprojekt verwendet werden. Dies wird unter dem Begriff „Mixed Language Programming“ zusammengefasst und ist in den Dokumentationen zu den Sprachen und Programmierplattformen zu entnehmen.
Die Prinzipien, Vorgehen und Möglichkeiten sind für alle Produkte und Plattformen etwa gleich. Deshalb wird in den nachfolgenden Kapiteln die Arbeit auf der Basis Microsoft Visual Studio 2003 VC7 gezeigt. Bei diesem Produkt sind alle benötigten Programmier- und Testkomponenten direkt verfügbar.

Gründe, Assembler in einer PC-Umgebung einzusetzen

Assembler wird in PC- und leistungsfähigen Mikrocontrollerumgebungen immer weniger eingesetzt. Die Gründe liegen in der schlechten Portierbarkeit auf andere Plattformen und dem enorm grossen Entwicklungs- und Dokumentationsaufwand.
Dennoch sind verschiedene Gründe für den Einsatz von Assembler denkbar. Aber selbst dann werden keine grossen Assemblerprogramme codiert, sondern nur das minimal Notwendige.

1. Grund: Bereits vorliegendes Material soll (weiter-) verwendet werden.
Dies, weil z. B. für eine Hardware vom Hersteller nur ein Objektmodul geliefert wird. Im Idealfall kann dieses Material direkt in das eigene Projekt eingebunden werden. Bei älteren Objektmodulen wird das Objektfileformat eventuell nicht kompatibel zum verwendeten Linker sein.
Alte 16-Bit Module lassen sich im Prinzip mit etwas Aufwand auch in eine 32-Bit Anwendung integrieren. Bei solchen Fragestellungen sollte man aber immer prüfen, ob ein Neudesign nicht sinnvoller wäre. Der Einsatz solcher Module ist unter Win32 (Win2K, WinXP) im Regelfall problemlos, solange im Modul kein direkter Zugriff auf die Peripherie erfolgt. Ist im Modul I/O notwendig, muss ein Neudesign mit einem WDM-Treiber erfolgen. (Vgl. auch [MSQ-1].)

Assemblerprogramme im Quellcode können direkt migriert werden. Bei 16-Bit Programmen sollte aber eine Erweiterung auf 32-Bit erfolgen.

2. Grund: Nutzung spezieller Prozessorbefehle
Compiler benutzen für die Codegenerierung einen Instruktionssatz, der auf allen Maschinen zur Verfügung steht. Dies sind die Befehle, die der i386 mit dem Gleitkommaprozessor i387 zur Verfügung stellt. Zahlreiche spätere Erweiterungen, wie z. B. MMX-Befehle werden nicht benutzt.

3. Grund: Effizienzverbesserung
Manche rechenzeitintensive Verfahren wie Verschlüsselung oder Datenkompression können durch Einsatz von Assembler etwas beschleunigt werden. Moderne Compiler erzeugen jedoch einen besseren Maschinecode als ein durchschnittlicher Assemblerprogrammierer. Deshalb ist dieser Grund nur eingeschränkt gültig.

Zu erwartende Probleme

Neben der schlechten Portierbarkeit der Programme und dem hohen Dokumentationsaufwand sind Probleme immer an den Schnittstellen zu erwarten. Dies sind Parameter und Resultate bei den Funktionen sowie beim Zugriff im Assemblercode auf in der Hochsprache definierte Daten und Symbole.
Stichworte, die bei der Verwendung von Funktionen in Assembler oder ganzen Assemblermodulen beachtet werden sollen sind:

`extern C {…}`	C++ Name-Mangling für exportierte Symbole deaktivieren
`CDECL, PASCAL`	Parameterreihenfolge, Stackbereinigung

Am Besten erstellt man zuerst eine Minimalfunktion und prüft, ob die Parameter korrekt übergeben werden und das Resultat wie gefordert, retourniert wird.

Das Debugging kann bei `_asm`-Sequenzen noch direkt im integrierten Debugger auf Quellcodeebene erfolgen. Bei externen Modulen wird dies schwieriger. Dasselbe gilt, wenn Module aus einer anderen Sprache wie FORTRAN oder Visual-Basic integriert werden.

Bei schwierigen Fällen kann ein sog. Kernel-Debugger eingesetzt werden (z. B. SoftICE von Compuware). Er erlaubt das symbolische Debugging auf Systemebene. Solche Werkzeuge sind teuer und setzen tief gehende Systemkenntnisse voraus.

Integrierte Assembler-Anweisungen im C/C++ Code

Maschinencode Statements mit _asm _emit

Microsoft- und Intel-Compiler, wie andere Produkte, unterstützen das direkte Einbringen von Maschinencode in Byteform mittels `_asm _emit`.
Die `_asm _emit` Anweisung ist in ihrer Wirkung wie ein `DB` (Define Byte) in Assembler zu verstehen. Sie definiert an der aktuellen Stelle im Codesegment ein einzelnes Byte. Sollen mehrere Bytes definiert werden, muss dies mit aufeinander folgenden `_asm _emit` Anweisungen erfolgen.

```
_asm _emit 0x90 _asm _emit 0x90 _asm _emit 0x90
```

Die `_asm`-Anweisung ist selbstterminierend, d. h. sie benötigt kein Semikolon. Deshalb können auch mehrere `_asm`-Anweisungen direkt aufeinander folgen.

Das Einbinden von Maschinencode hat keine Bedeutung in der praktischen Programmierung, weil mit `_asm` auch direkt Assemblercode eingebunden werden kann. Dies ist in jedem Fall zu bevorzugen.

Beispiel 1: Einbinden von Maschinencode-Bytes mit `_asm _emit`.

Bei Microsoft und Intel können einzelne Bytes mit `_asm _emit` direkt im Code platziert werden. Das nachfolgende Beispiel zeigt wie drei `NOP`-Befehle (Code 0x90) an der aktuellen Stelle definiert werden und wie die Umsetzung im Compiler erfolgt.

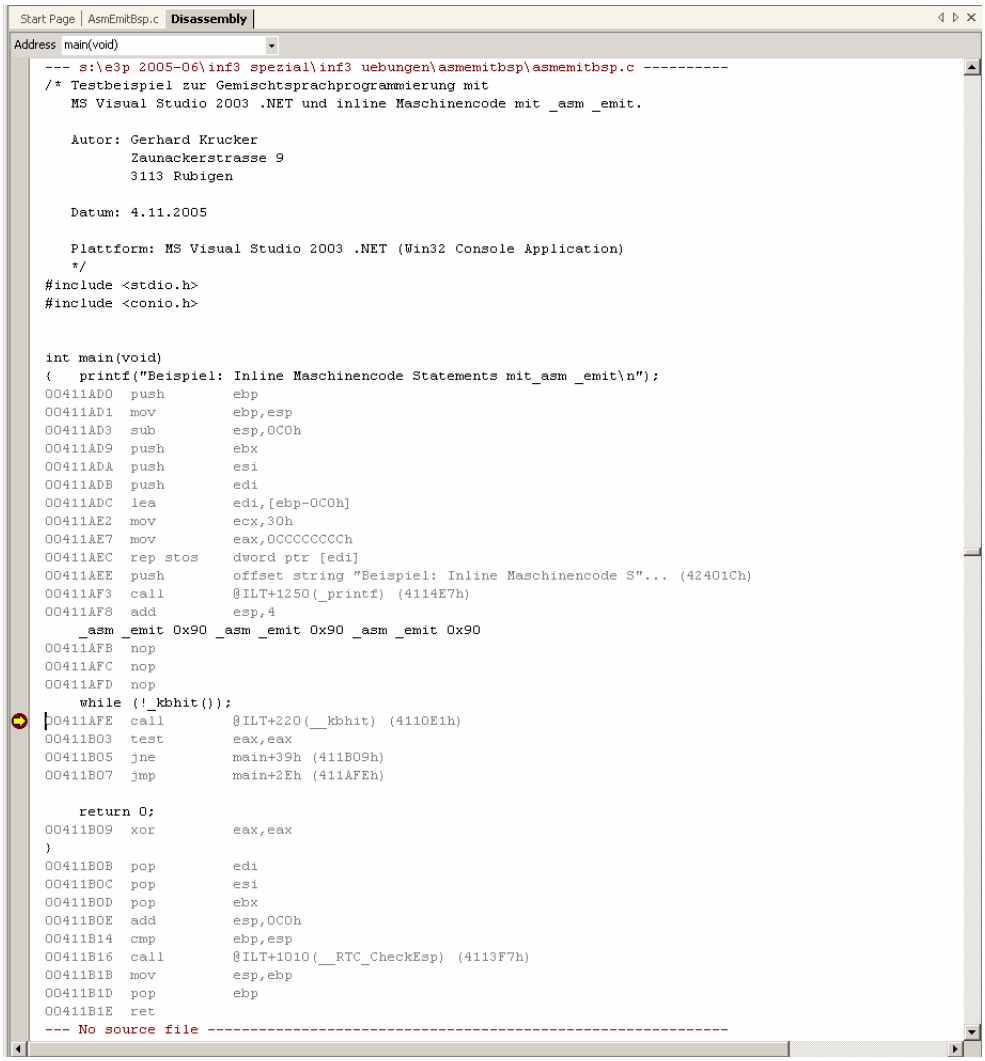


Bild 1: Disassemblierte Darstellung des Programms mit _asm _emit-Anweisungen nach Beispiel 1.
Plattform: Visual C++ V7.0 von VS .NET 2003.

Assemblercode mit _asm-Anweisungen

Durch das reservierte Wort _asm kann ein Assemblercodeblock direkt in den C/C++ Quelltext eingefügt werden. Innerhalb des Assemblerblockes gelten auch hier die Syntaxregeln des Assemblers, der zum Compiler gehört. Ein Zugriff auf C-Symbole ist auf einfache Weise möglich wie auch Funktionsaufrufe der C-Laufzeitbibliothek.

Das reservierte Wort _asm aktiviert den Inline-Assembler. Es muss immer in Verbindung mit einer einzelnen Assembler-Anweisung oder einem Block von Assembleranweisungen stehen.

```
_asm nop
```

Die _asm-Anweisung ist selbstterminierend, d. h. sie benötigt kein Semikolon. Deshalb können auch mehrere _asm-Anweisungen direkt aufeinander folgen.

```
_asm nop _asm nop _asm nop
```

Ein Semikolon nach der letzten Anweisung zu setzen ist aber kein Fehler. Zu beachten ist, dass das Semikolon die Zeile terminiert. Nachfolgende _asm-Anweisungen werden in dieser Zeile nicht mehr erkannt.

Beispiel 2: Einfache _asm-Anweisungen in C/C++ Code.

Das folgende Beispiel zeigt das Einbringen dreier NOP-Befehle in den C/C++ Quellcode:

```
#include <stdio.h>

int main(void)
{   printf("Beispiel: Inline Assemblercode mit_asm\n");
    _asm nop _asm nop _asm nop

    return 0;
}
```

Eine alternative Lösung mit einem Block:

```
#include <stdio.h>

int main(void)
{   printf("Beispiel: Inline Assemblercode mit_asm\n");
   _asm {
     nop          ; Klassischer Assembler-Kommentar ist erlaubt
     nop          /* Klassischer C-Kommentar ist erlaubt */
     nop          // C++-Kommentar ist erlaubt
   }
   return 0;
}
```

Kommentare können als normale C/C++ Kommentare eingebracht werden. Der klassische Assembler-Kommentar mit Semikolon ist ebenfalls zulässig.

Namen und Aufrufkonventionen

Sollen auf Daten oder Funktionen aus einem anderen Modul zugegriffen werden sind die Schnittstellen für den Aufruf zu berücksichtigen. Sie sind grundsätzlich sprach- und einstellungsabhängig.
Klassisch kennt man die Konventionen:

Namenkonvention

Sie besagt, wie der Compiler den Namen verändert bevor er in das Objektmodul exportiert wird. Am Beispiel des Symbols `getTime` wird dies:

SYSCALL
Der Symbolname wird unverändert in das Objektmodul übernommen. Der Linker referenziert die Variable als `getTime`.

C, STDCALL
Der Symbolname wird mit einem führenden Underscore ergänzt in das Objektmodul übernommen. Der Linker referenziert die Variable als `_getTime`.

FORTRAN, PASCAL, Basic
Der Symbolname wird in Grossbuchstaben in das Objektmodul übernommen. Der Linker referenziert die Variable als `GETTIME`.

Name Mangling

Bei C++ erfolgt für jede Funktion eine codierte Erweiterung, die die Anzahl Typ der Argumente beinhaltet. Für extern codierte Funktionen in einer anderen Sprache ist es daher notwendig die Funktionen als extern C { } zu deklarieren.

Aufrufkonvention

Sie besagt, wie der Compiler beim Aufruf einer Funktion oder Prozedur die Parameter ablegt und wie die Prozedur zu Aufrufer zurückkehrt. Bei VS .NET 2003 VC7 sind dies:

`__fastcall`
Die Argumente werden nach Möglichkeit in den Prozessorregistern der aufgerufenen Einheit übergeben.
Die ersten beiden DWORD oder kleiner Argumente werden in ECX and EDX übergeben. Alle anderen Argumente über den Stack Argumente von rechts nach links

`thiscall`
In C++ erfolgen standardmässig alle Funktionsaufrufe in dieser Konvention. Die Argumente werden in `__cdecl` Form auf den Stack gelegt und der `this`-Zeiger wird in `ECX` übergeben.
`thiscall` ist kein Keyword. Deshalb kann eine Funktion nicht explizit als `thiscall` definiert werden und ist nur für C++ Programme relevant.

`__stdcall`
Sie löst die `_pascal`-Aufrufkonvention ab. Praktisch alle Windows-Systemfunktionen arbeiten mit dieser Aufrufkonvention, da sie etwas laufzeit- und speicherlatzeffizienter ist als `__cdecl`.

`__cdecl`
Ist bei C Programmen Standard. Sie wird zwingend benötigt wenn mit Funktionen mit variabler Anzahl Argumenten gearbeitet wird, da nur der Aufrufer die tatsächliche Anzahl übergebener Argumente kennt und die Stackbereinigung durchführen kann.

Die Gross-/Kleinschrift der Symbole bleibt bei allen Verfahren unverändert.

Konvention	`__fastcall`	`__stdcall`	`thiscall`	`__cdecl`
Zugefügte Zeichen	@+@+Anzahl Bytes für Parameter	_+@+Anzahl Bytes für Parameter	_+@+Anzahl Bytes für Parameter	_
Argumente über Register übergeben	ECX, EDX		`this` in ECX	
Argumente von rechts nach links auf den Stack	(X)	X	X	X
Stackbereinigung durch Aufrufer			X	X

Die nachfolgenden Betrachtungen beschränken sich auf Assembler x86 und ANSI-C/C++.

Alle Aufrufkonventionen erzeugen einen Prozedurrahmen, der die Register `ESI, EDI, EBX` und `EBP` sichert und wiederherstellt und wo notwendig die Stackbereinigung durchführt. Aufrufe ohne Prozedurrahmen sind mit Naked Function Calls möglich.

Die Konventionen `_pascal`, `_fortran` und `_syscall` werden von der Seite der Hochsprache nicht mehr unterstützt. Ihre Funktionalität kann mit den bestehenden Methoden und entsprechenden Linkereinstellungen erreicht werden.

Beispiele von Parameterübergaben bei Funktionsaufrufen

Der Ablauf der Parameterübergaben bei den verschiedenen Aufrufkonventionen wird am nachfolgenden Beispiel gezeigt. Beim Funktionsaufruf in der Hochsprache wird `calltype` jeweils durch das entsprechende Aufrufprefix ersetzt.

```
void  calltype MyFunc( char c, short s, int i, double f );
.

void    MyFunc( char c, short s, int i, double f )

    {



    }

.

MyFunc ('x', 12, 8192, 2.7183);
```

`__cdecl`
Der Compiler exportiert den Funktionsnamen für den Linker als `_MyFunc`.Die Register ECX und EDX werden nicht benutzt. Die Parameterreihenfolge im Stack wird:

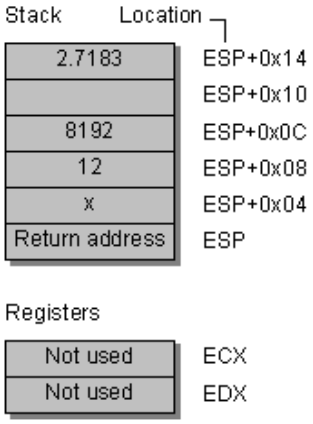


Bild 2: Stack und Register beim Funktionsaufruf mit __cdecl Konvention.
Bildreferenz: MSDN Developer Library CD 2005

`__stdcall` und **`thiscall`**
Bei `__stdcall` der Compiler exportiert den C-Funktionsamen für den Linker als `_MyFunc@20`. Der C++ Name ist proprietär und sollte in Assemblermodulen nicht referenziert werden. Der `this`-Zeiger wird bei C++ in ECX übergeben, EDX wird nicht benutzt. Die restlichen Argumente werden im Stack in der Reihenfolge übergeben:

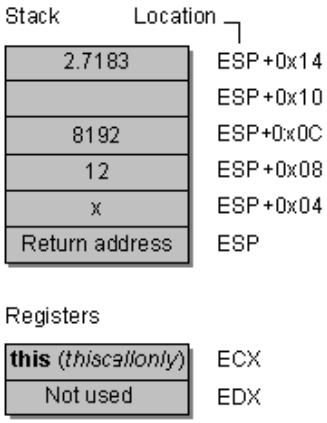


Bild 3: Stack und Register beim Funktionsaufruf mit __stdcall oder thiscall Konvention.
Bildreferenz: MSDN Developer Library CD 2005

`__fastcall`
Der Compiler exportiert den Funktionsnamen die `__fastcall` für den Linker als `@MyFunc@20`. Der C++ Name ist proprietär und sollte in Assemblermodulen nicht referenziert werden.

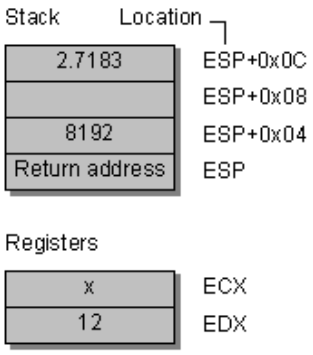


Bild 4: Stack und Register beim Funktionsaufruf mit __fastcall Konvention.
Bildreferenz: MSDN Developer Library CD 2005

Zugriff auf C-Symbole

Innerhalb der `_asm`-Anweisung kann auf C-Symbole (Variablen, Konstanten und Funktionen) zugegriffen werden. Der Programmierer ist selbst dafür verantwortlich, dass dies richtig geschieht. Fehler werden normalerweise nicht erkannt und führen meist zu einem Hänger oder zum sofortigen Programmabsturz.

Ein Zugriff auf C-Symbole ist direkt über den Namen möglich. Ein Zeiger auf eine Variable sollte mittels LEA-Prozessorbefehl gebildet werden (vgl. Beispiel 4).

Bedingung für den möglichen Zugriff ist, dass das Symbol im aktuellen Gültigkeitsbereich (Scope) liegt.

Beispiel 3: `_asm` Anweisungen in C-Code.

Das folgende Beispiel zeigt die Berechnung der Fakultät einer eingelesenen Ganzzahl. Die Rechnung erfolgt iterativ mit einer einfachen Schleife.

```
/* Beispiel zum Einbetten von Assemblercode in C-Programme
   x86 Inline Assemblercode und MS Visual Studio 2003 .NET.

   Autor: Gerhard Krucker
          Zaunackerstrasse 9
          3113 Rubigen
   Datum: 6.11.2005
   Plattform: MS Visual Studio 2003 .NET (Win32 Console Application)
   */
#include <stdio.h>
#include <conio.h>

int main()
{  int eingabeWert;
   int fakWert;
   printf("Eingabewert n fuer Fakultaetsberechnung n!:") ;
   scanf("%d",&eingabeWert);

   _asm{   mov eax,1            // 0! = Initialwert
           mov ecx,eingabeWert  // Zugriff auf C++ Variable
           cmp ecx,2            // Trivialfall n < 2
           jl   end
   $loop:  imul ecx             //  n = n * (n-1)!
           dec ecx              //  n = n -1
           cmp ecx,2            //  Ende wenn n < 2
           jnl $loop
   end:
           mov fakWert,eax      // In C++ Variable speichern
   }
   printf("Resultat: %d!=%d\n", eingabeWert,fakWert);

   while (!kbhit());
   return 0;
}
```

Die Umsetzung des gesamten Programms erfolgt bei VS .NET 2003 VC7:

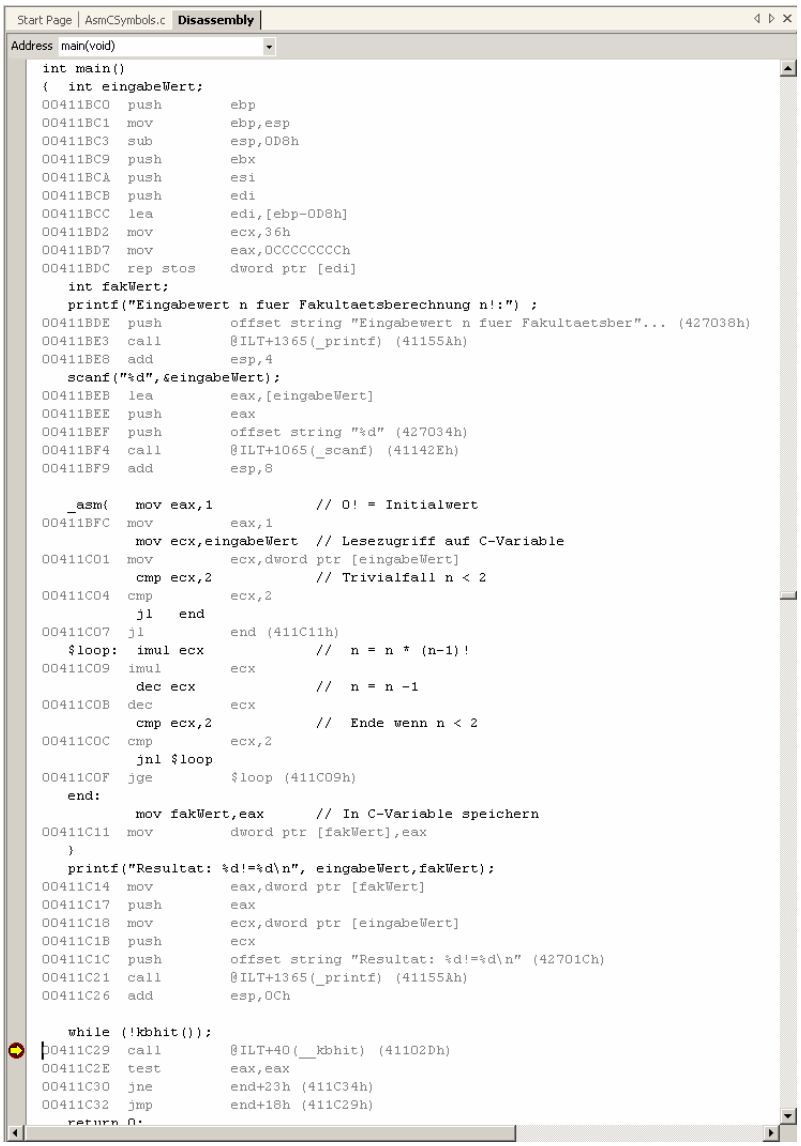


Bild 5: Mixed Language Darstellung im Debugger des Codes nach Beispiel 3.

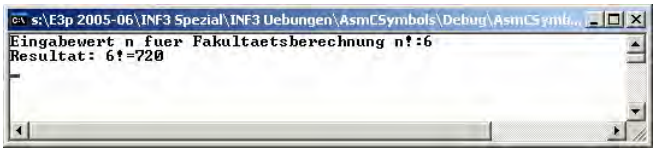


Bild 6: Ablauf und Ausgabe des Programmes nach Beispiel 3.

Zugriff auf C/C++ Runtime Library Funktionen

Im integrierten Assembler können auch C/C++ Funktionen der Runtime-Library verwendet werden. Ebenso sind auch Aufrufe der Win32-API möglich. Leider sind die RTL-Funktionen nicht sehr detailliert dokumentiert. Am besten benutzt man die Funktion zuerst einmal in der Hochsprache. Dann schaut man im disassemblierten Code nach wie die Funktion auf Assemblerebene genau heisst und wie die Parametrisierung erfolgt. Ein Zugriff auf `inline`-definierte -Funktionen ist aus nahe liegenden Gründen nicht möglich.

Beispiel 4: C-Funktionsaufruf in _asm Anweisungen.

Das folgende Beispiel zeigt den Zugriff auf C-Variablen und den Aufruf der `printf(..)`-Funktion. Die Ausgabe erfolgt einmal direkt in der Hochsprache und ein weiteres mal auf Stufe Assembler:

```
/* Beispiel zum Aufruf von C-Libraryfunktionen und Zugriff auf C-Variablen aus dem
   Inline-Assemblerblock.
   Autor: Gerhard Krucker
   Datum: 6.11.2005
   Plattform: MS Visual Studio 2003 .NET (Win32 Console Application)
*/
#include <conio.h>   // Fuer kbhit()
#include <stdio.h>

int main()
{ int a=1,b=2;
  const char fmtStr[]="Werte nach _asm-Sequenz:\na=%d, b=%d\n";

  const char *f=fmtStr;
  _asm{  mov eax, a        // Zugriffe auf die C-Variablen – Werte vertauschen
         mov ebx, b
         mov b,eax
         mov a,ebx
        };
    printf(fmtStr,a,b);  // Direkte Ausgabe
    // Ausgabe in Assembler mit der printf-Library-Funktion
    _asm{ push b
          push a
          lea eax,fmtStr
          push eax
          call printf       // Aufruf der printf-Libraryfunktion
          add esp,0ch       // Stackbereinigung (Parameter wieder entfernen)
         }
  while (!kbhit());
  return 0;
}
```

Bemerkung: Beim Aufruf der `printf`-RTL-Funktion im `_asm`-Block muss ein Zeiger auf den Formatstring `fmtStr` übergeben werden. Dies erfolgt hier mit LEA/PUSH. Ein direktes PUSH OFFSET funktioniert (eigenartigerweise) nicht.

Der gezeigte Code wird durch MS VC umgesetzt:

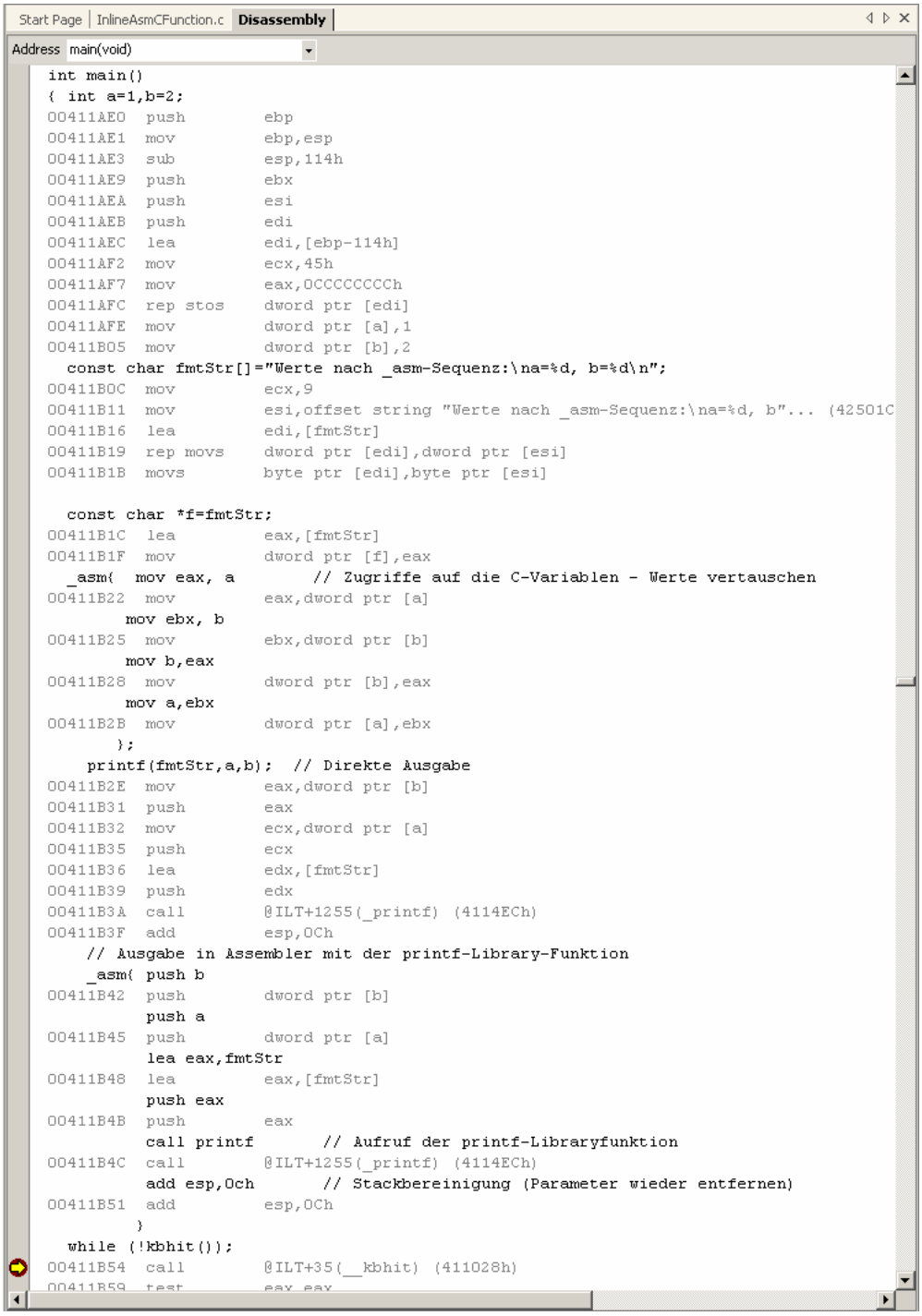


Bild 7: Disassemblierte Darstellung des Programms nach Beispiel 4 mit Aufruf einer Funktion der C-Runtime Library. Plattform:MS Visual Studio .NET 2003 VC7.

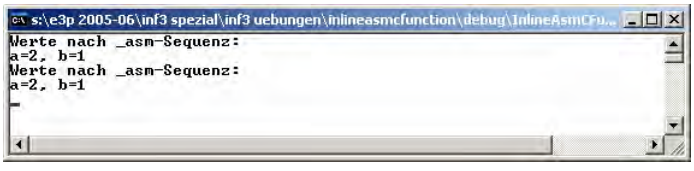


Bild 8: Ablauf und Ausgabe des Programms nach Beispiel 4.

VC7 kann beim Kompilieren ein kommentiertes Assemblerlisting als `*.cod`-File erzeugen. Es enthält den vom Compiler erzeugten Zwischencode aus der Hochsprache für den Assembler. Dieses File kann zur Lokalisierung von Fehlern in den `_asm`-Anweisungen nützlich sein.

```
; Listing generated by Microsoft (R) Optimizing Compiler Version 13.10.3077

         TITLE     .\InlineAsmCFunction.c
         .386P
include listing.inc
if @Version gt 510
.model FLAT
else
_TEXT     SEGMENT PARA USE32 PUBLIC 'CODE'
_TEXT     ENDS
_DATA     SEGMENT DWORD USE32 PUBLIC 'DATA'
_DATA     ENDS
CONST     SEGMENT DWORD USE32 PUBLIC 'CONST'
CONST     ENDS
_BSS      SEGMENT DWORD USE32 PUBLIC 'BSS'
_BSS      ENDS
$$SYMBOLS SEGMENT BYTE USE32 'DEBSYM'
$$SYMBOLS ENDS
$$TYPES   SEGMENT BYTE USE32 'DEBTYP'
$$TYPES   ENDS
_TLS      SEGMENT DWORD USE32 PUBLIC 'TLS'
_TLS      ENDS
;         COMDAT ??_C@_0CF@FDELPAMO@Werte?5nach?5_asm?9Sequenz?3?6a?$DN?$CFd?0?5b@
CONST     SEGMENT DWORD USE32 PUBLIC 'CONST'
CONST     ENDS
;         COMDAT _main
_TEXT     SEGMENT PARA USE32 PUBLIC 'CODE'
_TEXT     ENDS
sxdata    SEGMENT DWORD USE32 'SXDATA'
sxdata    ENDS
FLAT      GROUP _DATA, CONST, _BSS
          ASSUME    CS: FLAT, DS: FLAT, SS: FLAT
endif

INCLUDELIB LIBCD
INCLUDELIB OLDNAMES

PUBLIC    _main
PUBLIC    ??_C@_0CF@FDELPAMO@Werte?5nach?5_asm?9Sequenz?3?6a?$DN?$CFd?0?5b@ ; `string'
EXTRN     _kbhit:NEAR
EXTRN     _printf:NEAR
EXTRN     __RTC_InitBase:NEAR
EXTRN     __RTC_Shutdown:NEAR
EXTRN     @_RTC_CheckStackVars@8:NEAR
EXTRN     __RTC_CheckEsp:NEAR
;         COMDAT rtc$IMZ
; File s:\e3p 2005-06\inf3 spezial\inf3 uebungen\inlineasmcfunction\inlineasmcfunction.c
rtc$IMZ   SEGMENT
__RTC_InitBase.rtc$IMZ DD FLAT:__RTC_InitBase
rtc$IMZ   ENDS
;         COMDAT rtc$TMZ
rtc$TMZ   SEGMENT
__RTC_Shutdown.rtc$TMZ DD FLAT:__RTC_Shutdown
rtc$TMZ   ENDS
;         COMDAT ??_C@_0CF@FDELPAMO@Werte?5nach?5_asm?9Sequenz?3?6a?$DN?$CFd?0?5b@
CONST     SEGMENT
??_C@_0CF@FDELPAMO@Werte?5nach?5_asm?9Sequenz?3?6a?$DN?$CFd?0?5b@ DB 'Wer'
          DB        'te nach _asm-Sequenz:', 0aH, 'a=%d, b=%d', 0aH, 00H ; `string'
; Function compile flags: /Odt /RTCsu /ZI
CONST     ENDS
;         COMDAT _main
_TEXT     SEGMENT
_f$ = -80                                                 ; size = 4
_fmtStr$ = -68                                            ; size = 37
_b$ = -20                                                 ; size = 4
_a$ = -8                                                  ; size = 4
_main     PROC NEAR                                       ; COMDAT

; 11   : { int a=1,b=2;

  00000   55                push    ebp
  00001   8b ec             mov     ebp, esp
  00003   81 ec 14 01 00
          00                sub     esp, 276                ; 00000114H
  00009   53                push    ebx
  0000a   56                push    esi
  0000b   57                push    edi
  0000c   8d bd ec fe ff
          ff                lea     edi, DWORD PTR [ebp-276]
  00012   b9 45 00 00 00    mov     ecx, 69                         ; 00000045H
  00017   b8 cc cc cc cc    mov     eax, -858993460                 ; ccccccccH
  0001c   f3 ab             rep stosd
  0001e   c7 45 f8 01 00
          00 00             mov     DWORD PTR _a$[ebp], 1
  00025   c7 45 ec 02 00
          00 00             mov     DWORD PTR _b$[ebp], 2

; 12   :   const char fmtStr[]="Werte nach _asm-Sequenz:\na=%d, b=%d\n";

  0002c   b9 09 00 00 00    mov     ecx, 9
  00031   be 00 00 00 00    mov     esi, OFFSET FLAT:??_C@_0CF@FDELPAMO@Werte?5nach?5_asm?9Sequenz?3?6a?$DN?$CFd?0?5b@
  00036   8d 7d bc   lea     edi, DWORD PTR _fmtStr$[ebp]
  00039   f3 a5             rep movsd
  0003b   a4                movsb

; 13   :
; 14   :   const char *f=fmtStr;

  0003c   8d 45 bc   lea     eax, DWORD PTR _fmtStr$[ebp]
  0003f   89 45 b0   mov     DWORD PTR _f$[ebp], eax

; 15   :   _asm{  mov eax, a        // Zugriffe auf die C-Variablen - Werte vertauschen

  00042   8b 45 f8   mov     eax, DWORD PTR _a$[ebp]
```

```
; 16   :         mov ebx, b

  00045   8b 5d ec   mov    ebx, DWORD PTR _b$[ebp]

; 17   :         mov b,eax

  00048   89 45 ec   mov    DWORD PTR _b$[ebp], eax

; 18   :         mov a,ebx

  0004b   89 5d f8   mov    DWORD PTR _a$[ebp], ebx

; 19   :        };
; 20   :     printf(fmtStr,a,b);  // Direkte Ausgabe

  0004e   8b 45 ec   mov    eax, DWORD PTR _b$[ebp]
  00051   50                 push   eax
  00052   8b 4d f8   mov    ecx, DWORD PTR _a$[ebp]
  00055   51                 push   ecx
  00056   8d 55 bc   lea    edx, DWORD PTR _fmtStr$[ebp]
  00059   52                 push   edx
  0005a   e8 00 00 00 00     call   _printf
  0005f   83 c4 0c   add    esp, 12                                      ; 0000000cH

; 21   :     // Ausgabe in Assembler mit der printf-Library-Funktion
; 22   :     _asm{ push b

  00062   ff 75 ec   push   DWORD PTR _b$[ebp]

; 23   :           push a

  00065   ff 75 f8   push   DWORD PTR _a$[ebp]

; 24   :           lea eax,fmtStr

  00068   8d 45 bc   lea    eax, DWORD PTR _fmtStr$[ebp]

; 25   :           push eax

  0006b   50                 push   eax

; 26   :           call printf       // Aufruf der print-Libraryfunktion

  0006c   e8 00 00 00 00     call   _printf

; 27   :           add esp,0ch       // Stackbereinigung (Parameter wieder entfernen)

  00071   83 c4 0c   add    esp, 12                                      ; 0000000cH
$L876:

; 28   :         }
; 29   :   while (!kbhit());

  00074   e8 00 00 00 00     call   _kbhit
  00079   85 c0              test   eax, eax
  0007b   75 02              jne    SHORT $L877
  0007d   eb f5              jmp    SHORT $L876
$L877:

; 30   :   return 0;

  0007f   33 c0              xor    eax, eax

; 31   : }

  00081   52                 push   edx
  00082   8b cd              mov    ecx, ebp
  00084   50                 push   eax
  00085   8d 15 00 00 00
          00                 lea    edx, DWORD PTR $L894
  0008b   e8 00 00 00 00     call   @_RTC_CheckStackVars@8
  00090   58                 pop    eax
  00091   5a                 pop    edx
  00092   5f                 pop    edi
  00093   5e                 pop    esi
  00094   5b                 pop    ebx
  00095   81 c4 14 01 00
          00                 add    esp, 276                    ; 00000114H
  0009b   3b ec              cmp    ebp, esp
  0009d   e8 00 00 00 00     call   __RTC_CheckEsp
  000a2   8b e5              mov    esp, ebp
  000a4   5d                 pop    ebp
  000a5   c3                 ret    0
$L894:
  000a6   01 00 00 00        DD     1
  000aa   00 00 00 00        DD     $L893
$L893:
  000ae   bc ff ff ff        DD     -68                                  ; ffffffbcH
  000b2   25 00 00 00        DD     37                                   ; 00000025H
  000b6   00 00 00 00        DD     $L892
$L892:
  000ba   66                 DB     102                                  ; 00000066H
  000bb   6d                 DB     109                                  ; 0000006dH
  000bc   74                 DB     116                                  ; 00000074H
  000bd   53                 DB     83                                   ; 00000053H
  000be   74                 DB     116                                  ; 00000074H
  000bf   72                 DB     114                                  ; 00000072H
  000c0   00                 DB     0
_main     ENDP
_TEXT     ENDS
END
```

Beispiel 5: String-Funktionsaufruf in `_asm` Anweisung.
Das folgende Beispiel zeigt die Benutzung einer C-Stringfunktion aus C++. Dazu muss im C/C++ Teil ein `#include <string.h>` erfolgen, auch wenn die Funktion erst auf Assemblerebene verwendet wird. Dies gilt auch für andere Funktionen der Runtime Library.

Start Page | MSSTDCALL.asm | MS STDCALL Test.c | **MS ASM Strlen Aufruf.cpp** | cerr

(Globals) | main

```
//------------------------------------------------------------------------
#include <iostream>
#include <string.h>
//------------------------------------------------------------------------
using namespace std;

int main()
{ int i;
  char string[]="Hallo";
  int len;
   cout << "Zugriff auf Stringfunktionen:" << endl;
   len=(int)strlen(string);
    _asm { lea eax,string
           push eax
           call strlen
           add esp,4          // Parameter (1 DWORD) vernichten
           mov len,eax        // Resultat in C-Variable
         }

   cout << "Stringlaenge (" << string << "): " << len << endl;
   return 0;
}
```

```
s:\e3p 2005-06\inf3 spezial\inf3 uebungen\mixed language programs\ms a...
Zugriff auf Stringfunktionen:
Stringlaenge (Hallo): 5
```

Bild 9, Bild 10,: Programm und Ablauf mit Ausgabe des Programms nach Beispiel 5.

Zugriff auf in Assemblermodulen definierte Symbole

Umgekehrt ist prinzipiell auch aus der Hochsprache ein Zugriff auf Assemblersymbole möglich. Dazu muss bei Datensymbolen (Variablen, Konstanten) eine `extern`-Deklaration erfolgen. Die Referenzen werden dann über den Linker aufgelöst.

Bei Funktion muss im Hochsprachenteil ein Funktionsprototyp definiert werden. Bei den Parametern und Resultatrückgaben ist besondere Vorsicht walten zu lassen, da verschiedene Konfigurationsmöglichkeiten bestehen und Fehler sich meist sofort verheerend auswirken.

In `_asm`-Blöcken definierte Daten liegen grundsätzlich im Codesegment. Daher ist eine Definition von Daten und Datenbereichen in einem `_asm`-Block nicht sinnvoll.

Externe Assemblermodule

Externe Assemblermodule können mit Microsoft Visual Studio einfach erzeugt und in ein C/C++ Projekt eingebunden werden. Dazu wird zuerst das Projekt mit der `main()`-Funktion in der Hochsprache aufgesetzt. Das Assemblermodul wird als Textdatei mit `*.asm`-Erweiterung dem Projekt zugefügt. Das Assemblermodul hat in 32-Bit PC-Plattformen immer denselben schematischen Aufbau:

```
.386                            ; 32 Bit Bit Register
.model flat, c                  ; 32-Bit Adressbereich, C-Konventionen fuer Namenregeln

.code
                                ; Assembler codierte Funktionen
.data
                                ; Assembler Daten
 end
```

Im Gegensatz zu C/C++ Quelltexten müssen `*.asm`-Quelletexte vor dem Kompilieren (Assemblieren) immer explizit gespeichert werden, sonst werden die Änderungen nicht übernommen. Es empfiehlt sich im Modul immer die benutzten Register zu sichern, insbesondere EBP. Sie beinhalten eventuell Werte, die später wieder verwendet werden.

Für das Erstellen von Assemblermodulen muss sichergestellt sein dass der MASM Assembler (`ML.EXE`) im Compilerverzeichnis installiert ist. Er kann von der Setup-CD einfach in das `\bin`-Verzeichnis kopiert werden.

Soll alles aus der IDE bearbeitet und kompiliert werden können, wird das Assembler-Sourcfile mit der Extension `.asm` versehen und Kompilervorschrift von Hand im Property-Sheet des Assembler-Files eingetragen: (Vgl. auch Übungsbeispiel am Ende des Kapitels)

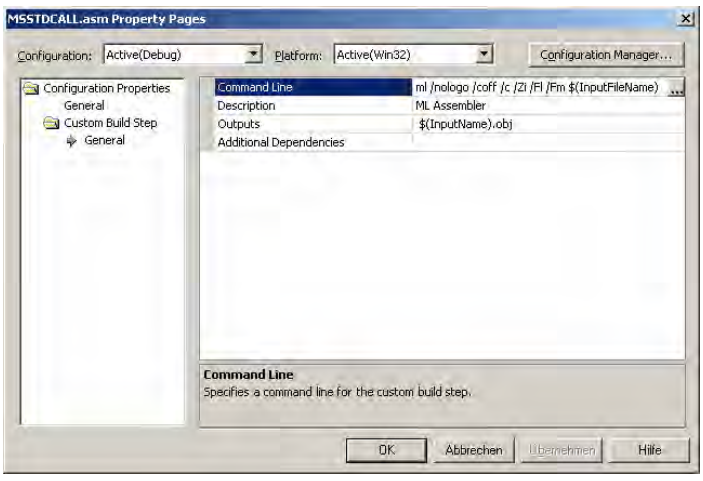


Bild 11: Einzutragende Kommandozeile für den Assembleraufruf beim Kompilieren des Files. Der Ausgabefilename muss ebenfalls angegeben werden. Hierzu wird zweckmässigerweise das $(InputName) benutzt.

Parameterübergaben

Eine nicht zu unterschätzende Fehlerquelle stellen die Datenübergabeschnittstellen dar. Bei den Parameter, die an eine Assemblerfunktion übergeben werden können, muss zwischen folgenden Prinzipien unterschieden werden:

`_cdecl`
Das ist das Standardübergabeprinzip in C/C++. Die Parameter werden in der umgekehrten Reihenfolge der Definition auf den Stack gelegt und die Stackbereinigung, d.h. das Entfernen der Parameter vom Stack erfolgt durch den Aufrufer.

Der folgende Code

```
void func(int var1, char var2, int var3);
```

legt Parameter in der Reihenfolge (`var3,var2,var1`) auf den Stack. Dies sei an einem Minimalbeispiel gezeigt.

C-Hauptmodul:
Im Hauptmodul wird der Funktionsprototyp definiert. Normalerweise gilt für C-Compiler die Einstellung `cdecl` als Standard, dann kann diese Angabe entfallen.

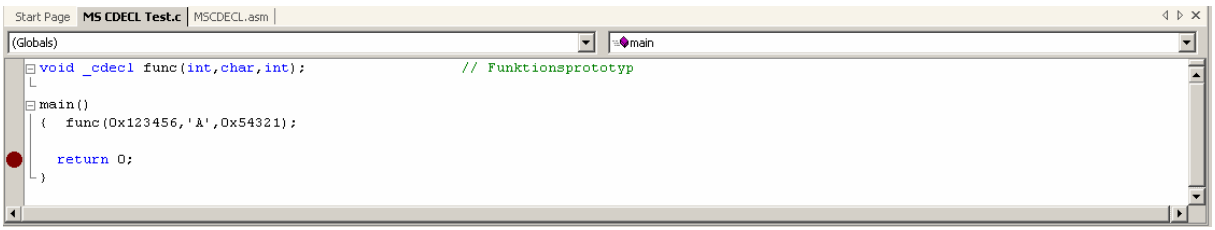


Bild 12: In C standarmässige Parameterübergabe in _cedcl Konvention nach C. Im Hochsprachenteil wird die externe Funktion über den Prototyp definiert.

In einem C++ Programm muss der Funktionsprototyp mit `extern "C "`definiert werden damit keine C++ Typinformation an den Funktionsnamen angehängt wird. Dies würde zu einem Linker-Fehler führen.

Assemblermodul:
Im Assemblermodul wird die Funktion in `proc … endp` codiert. Der Funktionsname wird mit `public c` exportiert, d. h. für die anderen Module sichtbar gemacht. Der Hinweis `c` bei `proc` und `public` bewirkt, dass die Namen automatisch mit einem Underscore (als `_func`) exportiert werden, weil der C-Compiler alle Namen mit einem Underscore verwaltet. Ferner werden c-exportierte Namen case-sensitive behandelt.

Start Page | MS CDECL Test.c* | **MSCDECL.asm**

```
            .386            ; 32 Bit Register
            .model flat, c  ; 32Bit Adressbereich, C-Namenkonvention

            public c func   ; Funktionsnamen aus dem Modul exportieren

            .code
func    proc c              ; Beginn  der Funktion

            mov eax,[ebp+4]                 ; Letztgepushten Parameter var1 in eax
            mov ebx,[ebp+8]                 ;  var2
            mov ecx,[ebp+0Ch]               ; Erstgepushten Parameter var3 laden
             ;
             ;                                Kein weiterer Code
            ret
func   endp

          end
```

Bild 13: Assemblermodul zur Parameterübergabe nach C. Die Funktion wird mit `proc c` definiert. Der Funktionsname wird mit `public c` exportiert.

Das Programm wird gestartet und bis zur `ret`-Anweisung im Assemblermodul mit dem Debugger ausgeführt. Wir sehen nachher wie die Parameter im Stack liegen (Fenster unten rechts).
Ebenso sieht man im disassemblierten Code von `main()`, dass nach dem Funktionsaufruf `call func(int signed char,int)` die Stackbereinigung mit `add esp,0x0c` erfolgt.

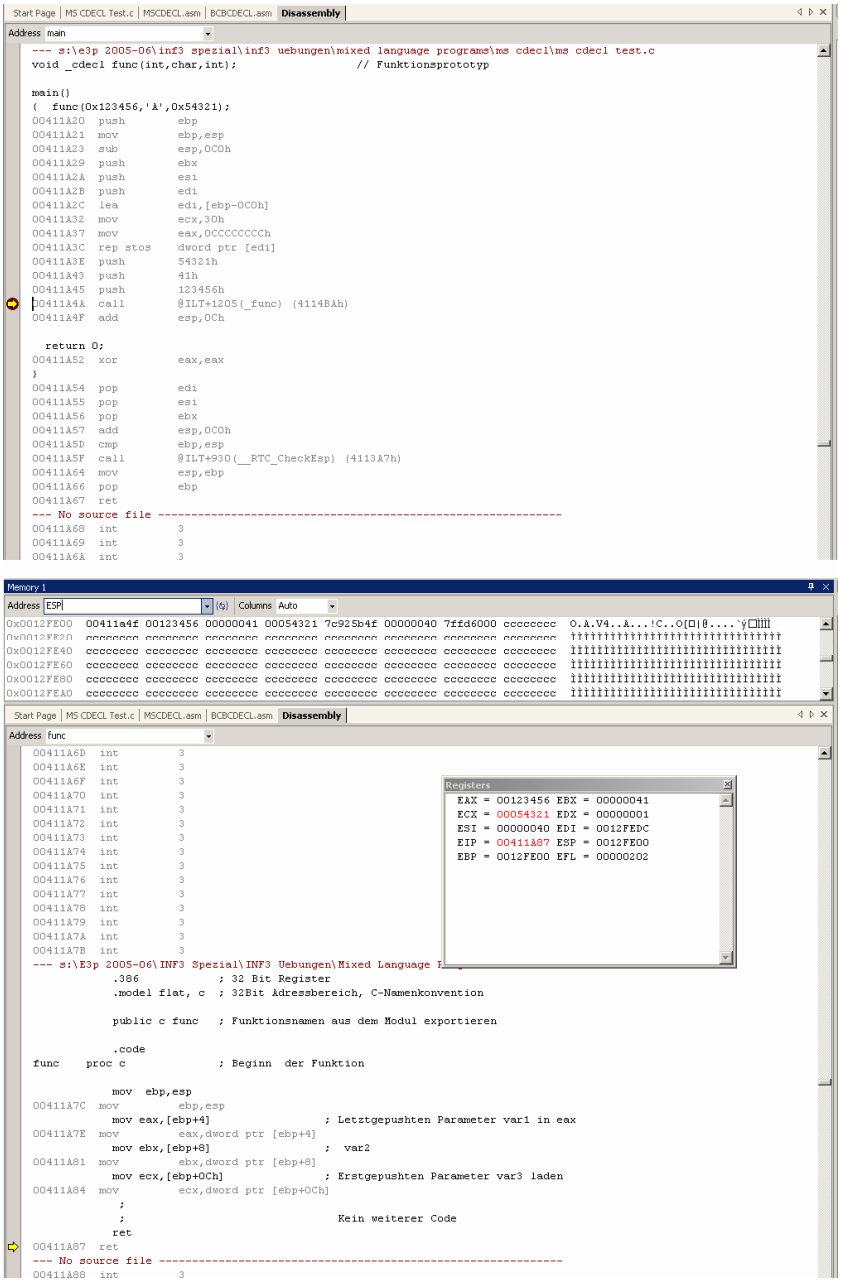


Bild 14: Programmablauf im Debugger mit Parameterübergabe nach C. Plattform: MS Visual Studio .NET 2003 VC7.

_stdcall

Bei der `_stdcall` Konvention erfolgt die Stackbereinigung durch die aufgerufene Einheit selbst. Praktisch alle Betriebssystemfunktionen von Windows arbeiten mit `_stdcall`-Übergaben, weil dies eine kleine Einsparung an Code bringt.

`_stdcall` löst die `_pascal` Aufrufkonvention ab. Bei der erfolgt ebenfalls die Stackbereinigung durch den Aufgerufenen, die Parameter werden aber in umgekehrter Reihenfolge auf den Stack gebracht.
Weiter wurden bei `_pascal` alle Symbole nur in Grossschrift behandelt.

Die Parameterübergabereihenfolge ist genau gleich wie bei _cdecl. Für das vorherige Beispiel mit `_stdcall`

```
void _stdcall func(int var1, char var2, int var3);
```

werden die Parameter in der Reihenfolge (`var3,var2,var1`) auf den Stack gelegt. Dies sei auch an folgendem Minimalbeispiel gezeigt.

C-Hauptmodul:
Der Funktionsprototyp wird mit Namen-Prefix `_stdcall` versehen.

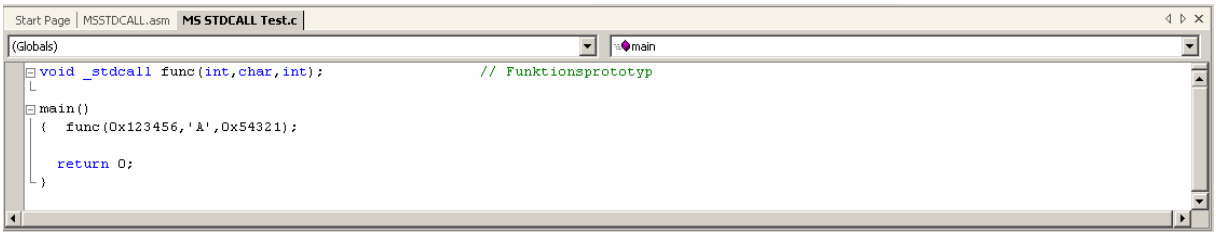


Bild 15: Parameterübergabe mit `_stdcall` Konvention. Im Hochsprachenteil wird die externe Funktion mit dem Prefix `_stdcall` über den Prototyp definiert.

Assemblermodul:
Die Funktion wird hier als `proc` ohne zusätzliche Erweiterungen definiert. Die Publikation verlangt aber nach dem Funktionsnamen ein @, gefolgt von der Anzahl Bytes, die auf dem Stack übergeben werden. Für dieses Beispiel wird `public func@12 eingesetzt`. Der Assembler exportiert das Symbol automatisch mit einem führenden Underscore als `_func@12`.
Wenn man nicht genau weiss, wie viele Bytes über den Stack übergeben werden, kann man anhand des Linker-Fehlers beim nicht aufgelösten Symbol die Anzahl herauslesen.

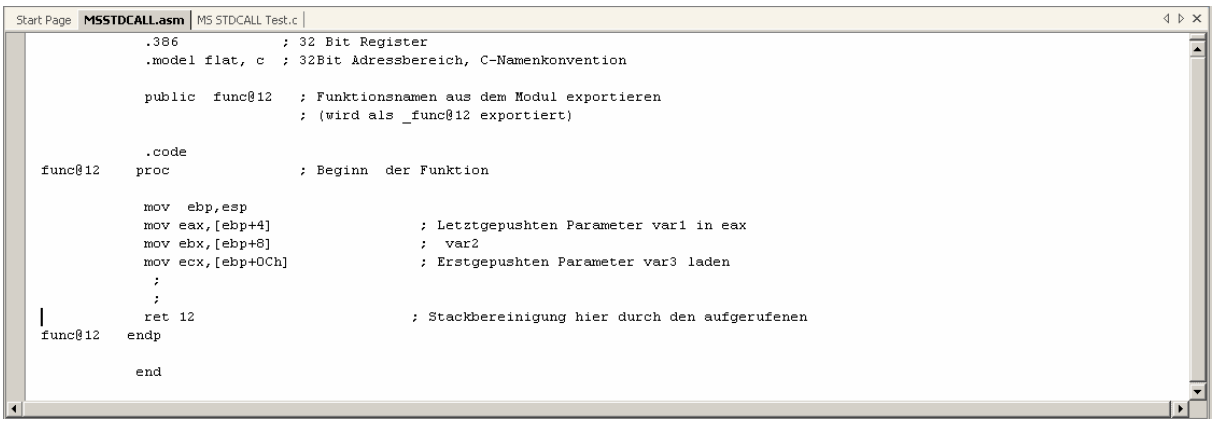


Bild 16: Assemblermodul zur Parameterübergabe mit `_stdcall`. Die Funktion wird mit `proc` definiert. Der Funktionsname wird mit `public` exportiert. Der Funktionsname muss mit @ und der Anzahl im Stack übergebener Bytes erweitert werden.

Das Programm wird gestartet und bis zur `ret 0x0c`-Anweisung im Assemblermodul mit dem Debugger ausgeführt. Wir sehen nachher wie die Parameter im Stack liegen (Fenster unten rechts).

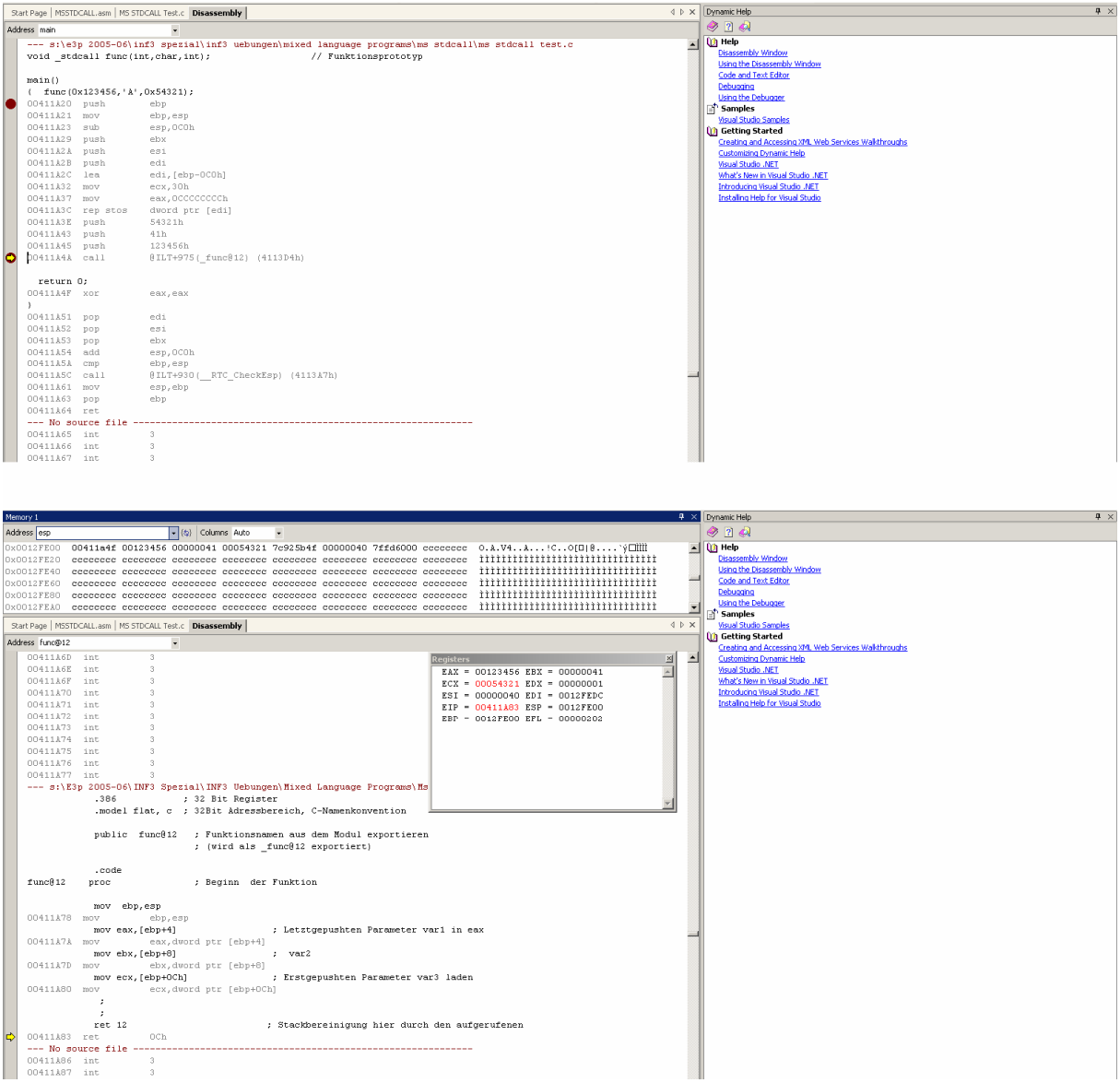


Bild 17: Programmablauf im Debugger mit Parameterübergabe mit `_stdcall`. Plattform: MS Visual Studio .NET 2003 VC7.

Im disassemblierten Code von `main()` sieht man, dass nach dem Funktionsaufruf `call func(int signed char,int)` keine Stackbereinigung erfolgt. Diese wird in der aufgerufenen Funktion mit `ret 0Ch` erledigt. Die Angabe 0Ch besagt, dass beim Rücksprung 0Ch=12=3 Parameter à 4Bytes vom Stack gelöscht werden.

Andere Parameterübergabetechniken

Die Übergabemethoden `_stdcall` und `_cdecl` sind sicherlich die beiden wichtigsten Verfahren. Bei C++ wird standardmässig `thiscall` verwendet. Da aber in C++ sowieso externe Funktionsaufrufe nur im C-Namespace erfolgen sollen, ist diese Form weniger von Bedeutung. Bei `_fastcall` werden Parameer in Prozessorregistern und über den Stack übergeben. Das bringt eine kleine Laufzeitverbesserung. Weitere Informationen zu diesen Techniken werden am besten in der MSDN Library nachgelesen.

Aus Gründen der Kompatibilität zu anderen Programmiersprachen existierten früher noch andere Verfahren.
`SYSCALL, STDCALL, BASIC, FORTRAN`

Auch sie unterscheiden sich in der Art und Reihenfolge wie die Argumente übergeben werden und wie die Stackbereinigung erfolgt. Sie werden heute aber von Microsoft nicht mehr unterstützt. Detaillierte Informationen sind in den produktespezifischen Programmierhandbüchern zu finden. [MASMPG-92],[BORASMBH-92],[BORASMRH-92].

Funktionswertrückgaben

Grundsätzlich werden Funktionswerte im `EAX` Register zurückgegeben, ausser bei 8-Byte Strukturen, die im `EDX:EAX` Registerpaar zurückgeben werden.

Für verschiedene C-Datentypen wird die im Speziellen:

C-Datentyp	Register	Wortbreite
char	al	8 Bit
short	ax	16 Bit
long	eax	32 Bit
Zeiger	eax	32 Bit
Struct	Bis 4 Bytes: eax	
	>4 Bytes über Variablenreferenz	
Array	Zeiger in eax	32 Bit

Werden Strukturvariablen >4 Bytes retourniert, werden diese automatisch mit einer `MOVS`-Anweisung übergeben.

In 16-Bit Modulen werden 32-Bit Werte über `DX:AX` zurück gegeben, wobei `DX` das höherwertige Wort ist.

Gleitkommawerte des Typs `double` oder `float` werden in `st(0)` der FPU retourniert. [MSDN]

Beispiel 6: Fakultätsberechnung mit externem Assemblermodul und Funktionswertrückgabe.

In einem C++ Programm soll mit einem externen Assemblermodul eine Funktion `fak` zur Verfügung gestellt werden. Das Argument wird in C-Standardnotation über Parameter übergeben, das Resultat über Funktionswertrückgabe. Mit einem Hauptprogramm wird die Anwendung der Funktion gezeigt.

Lösung:
Da es sich um ein C++ Programm handelt (File-Extension des aufrufenden Moduls ist `*.cpp`), muss der Funktionsprototyp als `extern "C"` definiert werden. Das Assemblermodul wird mit der Funktionalität analog Beispiel 3 erstellt.
Der Funktionsname wird über `fak proc c`…`fak endp` definiert und mit `public c fak` exportiert:

Start Page | FakAsmExternCPPTest.cpp* | FakAsmExt.asm

(Globals) | main

```
//------------------------------------------------------------------------
// Beispiel zu Fakultaetsberechnung mit einem externen Assemblermodul in C++
// Gerhard Krucker
// 5-2-2006 (Borland C++ Builder V5.0: 9-1-2004, 15-1-2004)
//
// MS VC .NET 2003 (Win32 Console Application)
//------------------------------------------------------------------------
#include <iostream>
//------------------------------------------------------------------------
extern "C" int cdecl fak(int);        // Funktionsprototyp als ANSI-C ohne Name-Mangling

using namespace std;
int main()
{  int n, nFak;

   do {cin.clear();                // Eigabepuffer leeren
       cout << "\nEingabe n zur Berechnung von n! =    (Negatives n = Programm beenden)\n";
       cin >> n;
       nFak=fak(n);
       cout << "n! = " << nFak << endl;
      } while (n >= 0);
   return 0;
}
//------------------------------------------------------------------------
```

Start Page | FakAsmExternCPPTest.cpp* | FakAsmExt.asm

```
                .386
                .model flat, c
                public c fak

                .code
                ; Fakultaetsberechnung: extern "C" int cdecl fak(int @n)
                ; Berechnung mit Iteration in Form von sukzessiver
                ; Multiplikation.
                ; Parameter: n ueber Stack uebergeben
                ; Funktionswert: n! in eax retouniert
                ; Geaenderte Register: keine
fak             proc c,
                n:DWORD             ; Parameter n (Beachten Sie das Komma auf der oberen Zeile!)
                push ecx            ; Benutzte Register sichern

                mov eax,1           ; 0! = Initialwert
                mov ecx,n           ; Parameter laden
                cmp ecx,2           ; Trivialfall n < 2
                jl   $end
                ; Iteration
@@:             imul ecx            ;  n = n * (n-1)!
                dec ecx             ;  n = n -1
                cmp ecx,2           ;  Ende wenn n < 2
                jnl @B              ; Zum naechst liegenden anonymous label @@: rueckwaerts springen

$end:           pop ecx
                ret                 ; @n! in eax retournieren
fak             endp

                end
```

Bild 18, Bild 19: Parameterübergabe und Funktionswertrückgabe bei einer extern “C“ definierten Funktion nach Beispiel 6.
Plattform: MS Visual Studio .NET 2003 VC7.

Bild 20: Debuggerfenster der CPU mit Darstellung des erzeugten Codes nach Beispiel 6. Plattform: MS Visual Studio .NET 2003 VC7.

Alternative Lösung:
Man kann die Direktive `extern "C"` weglassen, wenn die C++ Namenerweiterung für die extern definierte Funktion bekannt ist. Die Namenerweiterung wurde notwendig, weil in C++ Funktionen denselben Namen haben dürfen, sofern sie sich in den Parametern unterscheiden. Die Unterscheidung erfolgt in Typ und Anzahl.

Für dieses Beispiel wäre der erweiterte Funktionsname `?fak@@YAHH@Z`. Dieser Name wird am einfachsten aus der Fehlermeldung beim Linken bestimmt. Die Funktion muss auf Stufe Assembler als `syscall` definiert werden. `syscall` bewirkt, dass der Name case-sensitive behandelt wird, aber ohne Underscore exportiert wird. Die Stackbereinigung erfolgt bei `syscall` wie bei `c` durch den Aufrufer. (Vgl. auch [MSAMPG-92], S.309)

Start Page | FakAsmExternCPP2Test.cpp | **FakAsmExternCPP2Test.cpp*** | FakAsmExt2.asm | FakAsmExt2.lst

(Globals) | main

```
//--------------------------------------------------------------------------
// Beispiel zu Fakultaetsberechnung mit einem externen Assemblermodul in C++.
// Speziell: Hier wird die Funktion im C++ Namespace mit dem Name-Mangling aufgerufen.
// Deer Name fuer die externe Referenz kann aus dem Linker-Error oder aud dem Listing des
// Compilers fuer das CPP-File (Poperties-Output-Assembler-Output und ASM List Location)
// entnommen werden. Dito beim Assembler-Modul.
// Gerhard Krucker
// 5-2-2006 (Borland C++ Builder V5.0: 9-1-2004, 15-1-2004)
//
// MS VC .NET 2003 (Win32 Console Application)
//--------------------------------------------------------------------------
#include <iostream>
//--------------------------------------------------------------------------
 int cdecl fak(int);        // Funktionsprototyp als ANSI-C mit Name-Mangling

using namespace std;
int main()
{  int n, nFak;

   do {cin.clear();               // Eigabepuffer leeren
       cout << "\nEingabe n zur Berechnung von n! =    (Negatives n = Programm beenden)\n";
       cin >> n;
       nFak=fak(n);
       cout << "n! = " << nFak << endl;
      } while (n >= 0);
   return 0;
}
//--------------------------------------------------------------------------
```

Start Page | FakAsmExternCPP2Test.cpp | FakAsmExternCPP2Test.cpp | **FakAsmExt2.asm** | FakAsmExt2.lst

```
                .386
                .model flat,syscall   ; syscall fuer C++
                public ?fak@@YAHH@Z   ; Zu exportierendes Symbol

                .code
                ; Fakultaetsberechnung:  int cdecl fak(int n)
                ; Die Funktion wird direkt aus dem C++ Namespace aufgerufen.
                ; Das zu exportierende Symbol mit der Dekoration muss aus dem Linker-Error
                ; bestimmt werden.
                ;
                ; Berechnung mit Iteration in Form von sukzessiver
                ; Multiplikation.
                ; Parameter: n ueber Stack uebergeben
                ; Funktionswert: n! in eax retouniert
                ; Geaenderte Register: keine
?fak@@YAHH@Z    proc ,
                n:DWORD             ; Parameter n (Beachten Sie das Komma auf der oberen Zeile!)
                push ecx            ; Benutzte Register sichern

                mov eax,1           ; 0! = Initialwert
                mov ecx,n           ; Parameter laden
                cmp ecx,2           ; Trivialfall n < 2
                jl   $end
                ; Iteration
@@:             imul ecx            ;  n = n * (n-1)!
                dec ecx             ;  n = n -1
                cmp ecx,2           ;  Ende wenn n < 2
                jnl @B              ; Zum naechst liegenden anonymous label @@: rueckwaerts springen

$end:           pop ecx
                ret                 ; @n! in eax retournieren
?fak@@YAHH@Z    endp

                end
```

Bild 21, Bild 22: Parameterübergabe und Funktionswertrückgabe bei einer C++ definierten Funktion nach Beispiel 6 Der Funktionsname in Assembler trägt die Namenerweiterung `?..@@YAHH@Z`. Plattform: MS Visual Studio .NET 2003 VC7.

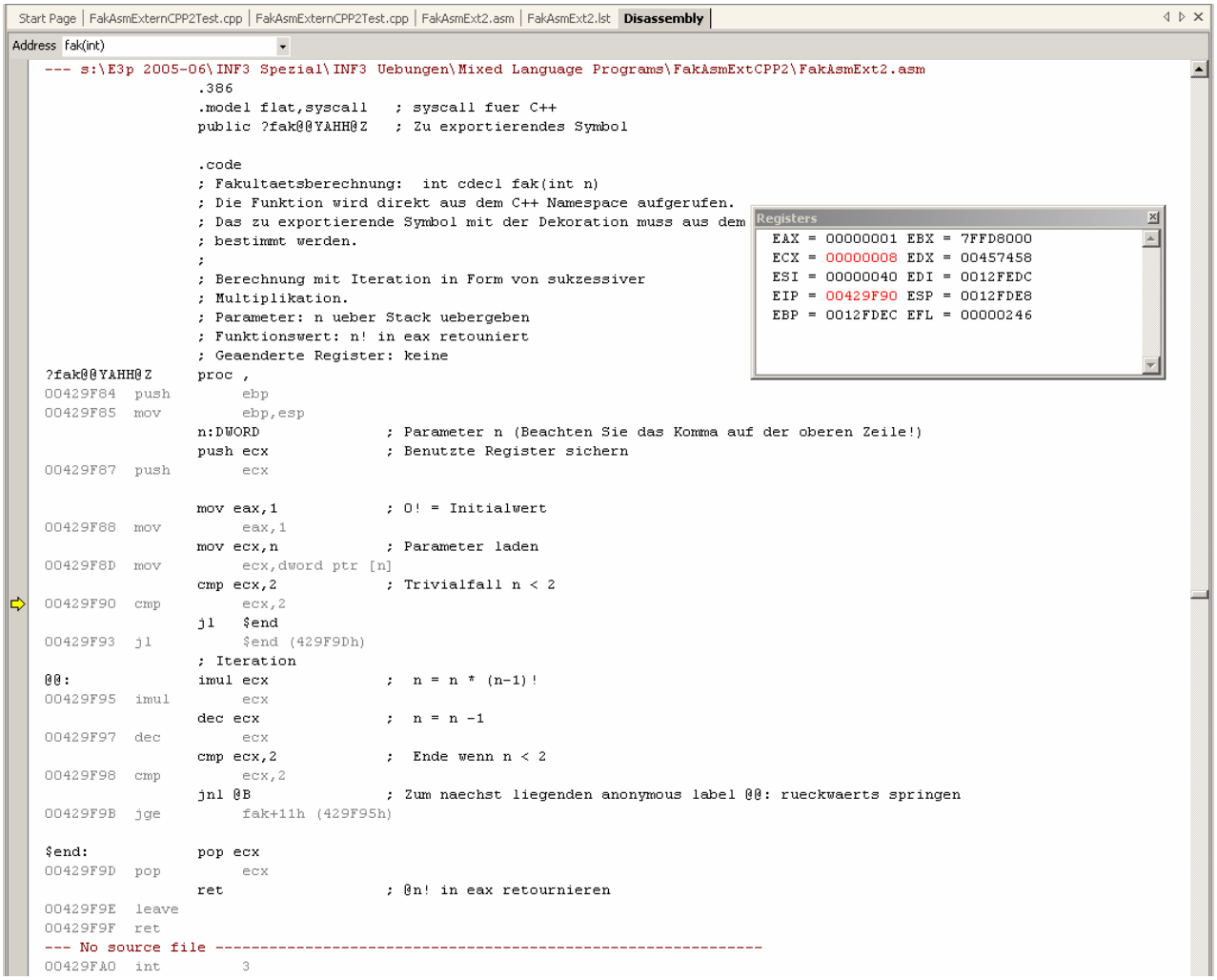


Bild 23: Debuggerfenster der CPU mit Darstellung des erzeugten Codes für die alternative Lösung nach Beispiel 6. Aus Platzgründen wurde auf die Darstellung der main-Funktion verzichtet. Plattform: MS Visual Studio .NET 2003 VC7.

Auf das Fenster der C++.-Rahmenprogrammes wurde verzichtet, da Microsoft Visual C++ für kleine Anwendungen extrem viel Code erzeugt. Borland C++ Builder der Code deutlich kompakter.

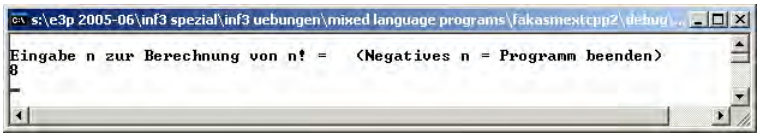


Bild 24: Ablauf des Programmes mit der alternativen Lösung nach Beispiel 6. Plattform: MS Windows XPSP2

Naked Functions

Naked Functions sind Funktionsaufrufe bei denen der Compiler keinen Prolog/ Epilog für das Sichern der Registerinhalte und Stackbereinigung erzeugt. Diese Funktionen werden mit dem Prefix `__declspec(naked)` definiert.

Dadurch kann innerhalb eines C/C++ Programmes mit inline-Assembler eine Funktion definiert werden, die ihren eigenen, nicht standardmässigen Prozedurrahmen haben kann. Die wird benötigt wenn man beispielsweise extern aus einem anderen Sprachkontext als C/C++ auf die Funktion zugreifen will. Als Beispiele wären ein VxD oder ein MS-DOS Device Handler denkbar.

Einschränkungen für Naked Functions

Folgende Aktionen sind in Naked Functions nicht zulässig:

- Die Anweisung return
- C++ Exception Handling
- Alle Formen von `setjmp`
- `_alloca`

Wegen des fehlenden Prozedur-Prologs sind auch initialisierte Lokalvariablen nicht erlaubt. Dies gilt auch für lokale C++ Objekte.

Definition von Naked Functions

Eine Naked Funktion wird durch die erweiterte Attribut-Syntax `__declspec(naked)` gekennzeichnet. Diese Definition erfolgt ausschliesslich bei der Implementierung der Funktion, nie beim Prototyp (Deklaration). Die Grobstruktur der Funktion ist typischerweise:

```
__declspec( naked ) int func( formale_Parameter )
{
    // Funktionskoerper
}
```

Oder unter Verwendung eines Makros:

```
#define Naked   __declspec( naked )

Naked int func( formale_Parameter )
{
    // Funktionskoerper
}
```

Das `naked`-Attribut steuert nur die Codeerzeugung der nachfolgenden codierten Funktion. Sie ist nicht Bestandteil des Funktionstyps. Aus diesem Grund können auch keine `naked`-Funktionszeiger definiert werden. Ebenso ist `naked` beim Funktionsprototyp nicht erlaubt und kann nicht auf andere Daten angewandt werden.

```
__declspec( naked ) int func();  // Fehler: naked ist beim Funktionsprototyp nicht erlaubt
                                 //- nur bei der Implementierung.

__declspec( naked ) int i;  // Fehler: naked Attribute ist bei Daten nicht zulaessig.
```

Da kein Prozedurprolog mit Stackmanagement erzeugt wird, kann weder formale Parameter, noch auf herkömmliche Lokalvariablen direkt Bezug genommen werden.
Um dennoch in Naked Functions mit stackbasierten Lokalvariablen arbeiten zu können, wurde neu das Assemblersymbol `__LOCAL_SIZE` eingeführt.
Es wird benutzt, um Speicherplatz für Lokalvariablen in einer Naked Funktion auf dem Stack bereitzustellen. Es beinhaltet den Platzbedarf aller in der Funktion benutzten Lokalvariablen. Die Bereitstellung des Speicherplatzes auf dem Stack erfolgt nicht automatisch, sondern muss explizit codiert werden. Das Vorgehen ist dabei genau gleich wie beim automatisch erzeugten Prozedur Epilog mit Lokalvariablen. Von der Benutzung her kann `__LOCAL_SIZE` kann als Operand oder in einem Ausdruck verwendet werden.

Beispiele:

```
mov     eax, __LOCAL_SIZE           /* Immediate Operand */
mov     eax, __LOCAL_SIZE + 4       /* Ausdruck */
mov     eax, [ebp - __LOCAL_SIZE]   /* Ausdruck */
```

Formale Parameter müssen selbst vom Stack gelesen und verwaltet werden. Die Angabe im Funktionskopf ist nur eine Information für die Stackbereinigung. Der Zugriff auf den letztgepushten Parameter wäre `[esp+4]`, vgl. auch Beispiel 7.

Beispiel 7: Fakultätsberechnung mit Naked Funktion.

Analog Beispiel 3 soll mit einer Assemblersequenz die Fakultät einer Ganzzahl berechnet werden. Die Naked-Funktion wird mit dem Prefix `__declspec(naked)` versehen. In der Funktion kann aber nicht direkt auf den formalen Parameter zugegriffen werden. Er muss selbst mit `mov ..,[esp+4]` vom Stack gelesen werden. Sonst gelten dieselben Regeln wie für normale Inline-Assemblersequenzen.

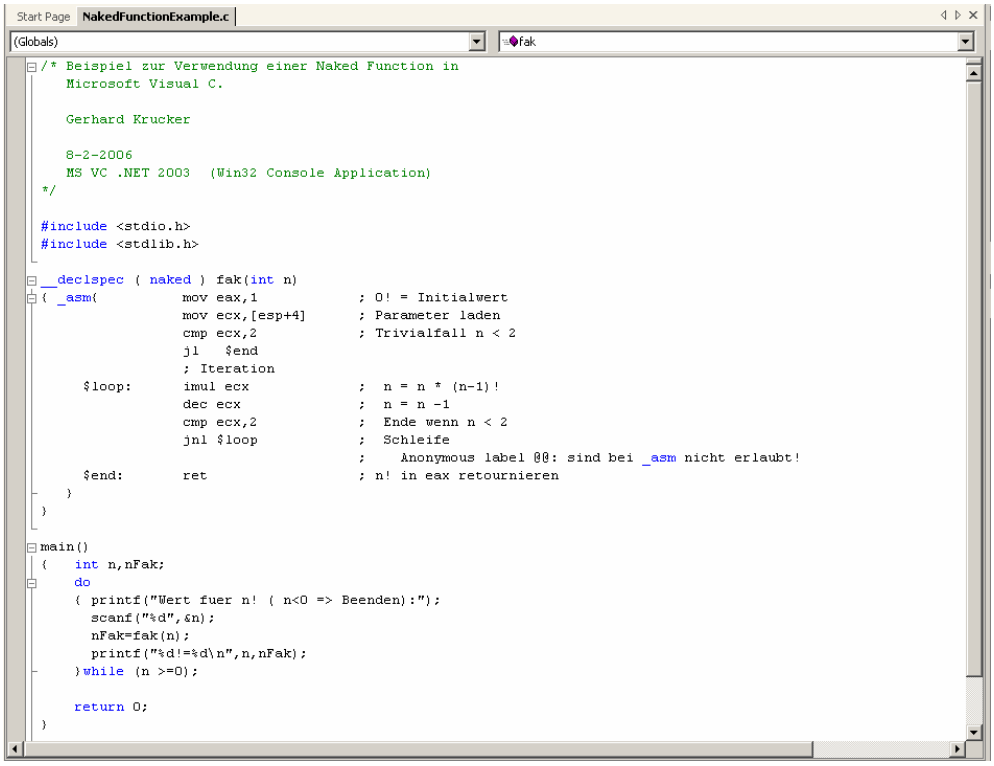


Bild 25: Programm mit einer Naked Function nach Beispiel 7 analog Beispiel 3.

Für eine Eingabe von n=6 wird der Ablauf im Debugger:

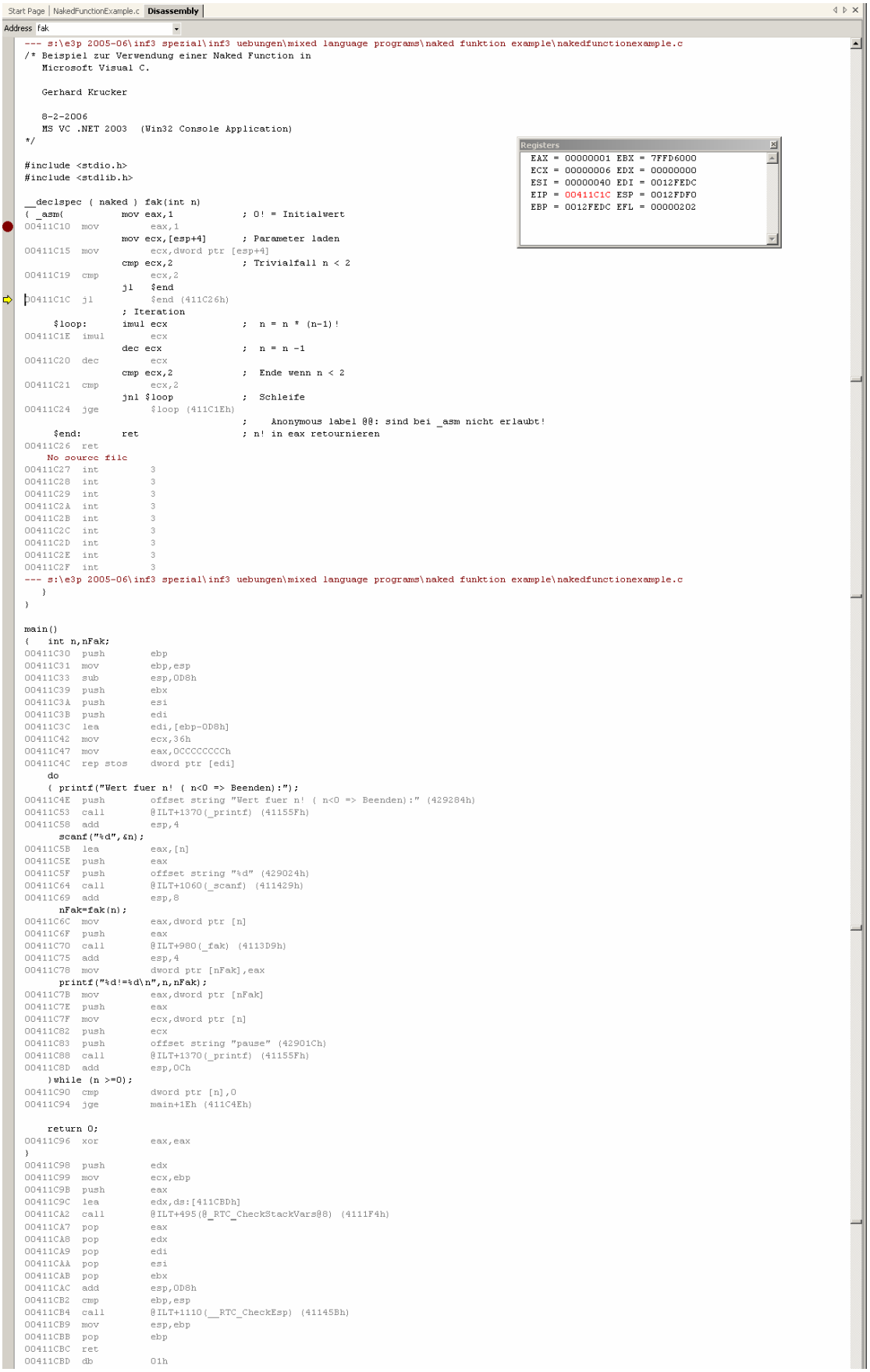


Bild 26: Ablauf des Programmes mit einer Naked Function nach Beispiel 7.

Beispiel 8: Naked Funktion mit Lokalvariablen.

Analog Beispiel 3 soll mit einer Assemblersequenz die Fakultät einer Ganzzahl berechnet werden. Die Naked-Funktion wird mit dem Prefix `__declspec(naked)` versehen. In der Funktion kann aber nicht direkt auf den formalen Parameter zugegriffen werden. Er muss selbst mit `mov ..,[esp+4]` vom Stack gelesen werden. Sonst gelten dieselben Regeln wie für normale Inline-Assemblersequenzen.

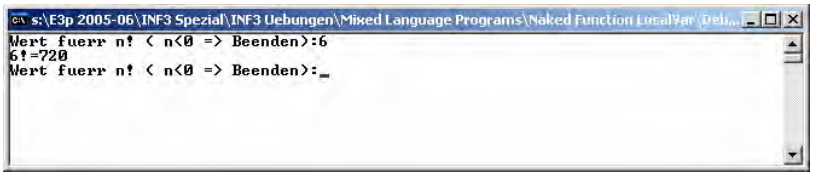
Start Page | NakedFunctionExample.c | NakedFuntionLocalVar.c*

(Globals) | main

```
/* Beispiel zur Verwendung einer naked function in
   Microsoft Visual C.

   Gerhard Krucker
   8-2-2006
   MS VC .NET 2003  (Win32 Console Application)
*/

#include <stdio.h>
#include <stdlib.h>

__declspec ( naked ) int fak(int nDummy)
{ int i,n,nFak;
  _asm
  {  push   ebp                 ; Registerinhalt sichern
     mov    ebp, esp
     sub    esp, __LOCAL_SIZE   ; Platz fuer Lokalvariablen schaffen
     mov    eax,[ebp+8]         ; Parameter holen und in Lokalvariable speichern (+8 weil schon 1 push erfolgte)
     mov    [n],eax
   }

  nFak=1;
  for (i=n;i>1;i--)               // Die Rechnung efolgt in Hochsprache
      nFak=nFak*i;
  _asm
  { mov     eax,[nFak]        ; Funktionswert fuer Rueckgabe in EAX platzieren
    mov     esp, ebp          ; Lokalspeicherplatz wieder freigeben
    pop     ebp               ; Register wieder herstellen

    ret
  }
}


main()
{   int n,nFak;
    do
    { printf("Wert fuerr n! ( n<0 => Beenden):");
      scanf("%d",&n);
      nFak=fak(n);
      printf("%d!=%d\n",n,nFak);
    }while (n >=0);

    return 0;
}
```

Bild 27: Programme mit einer Naked Function und Lokalvariablen in der Naked Function nach Beispiel 8.
Plattform: MS Windows XPSP2

Ein Ablauf für eine Eingabe von n=6 wird mit dem Debugger:



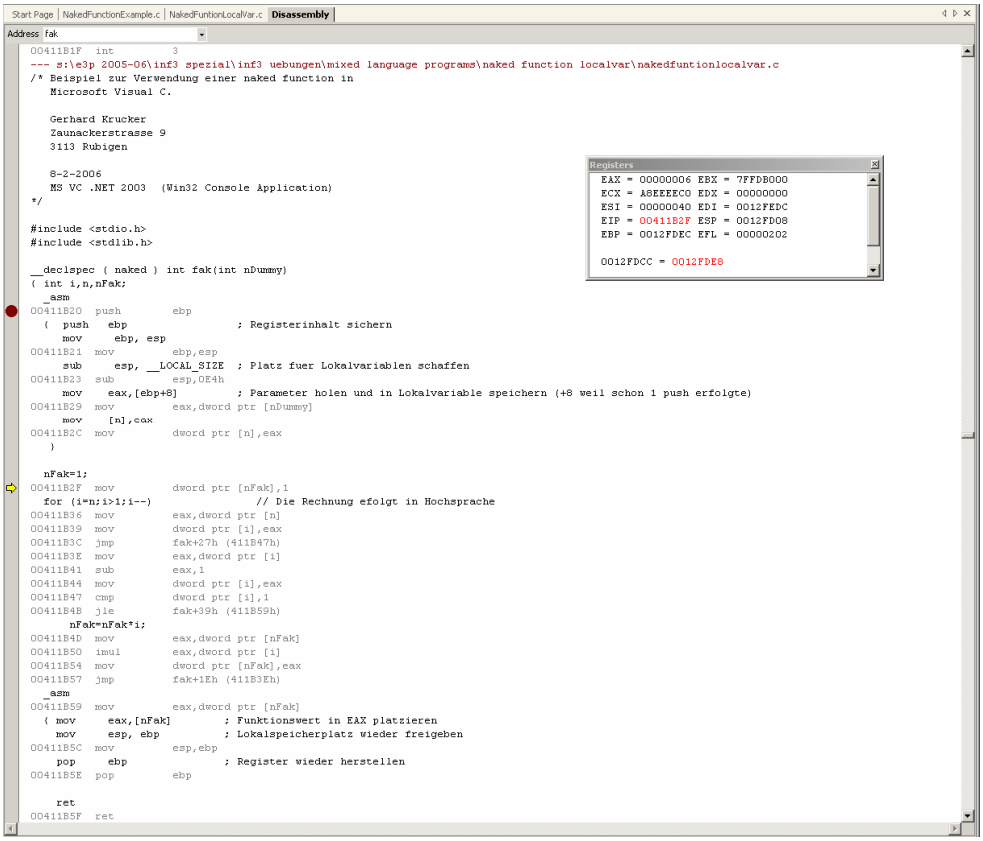


Bild 28: Ablauf des Programmes mit der Eingabe n=6 unter dem Debugger in der Naked Function mit Lokalvariablen nach Beispiel 8.

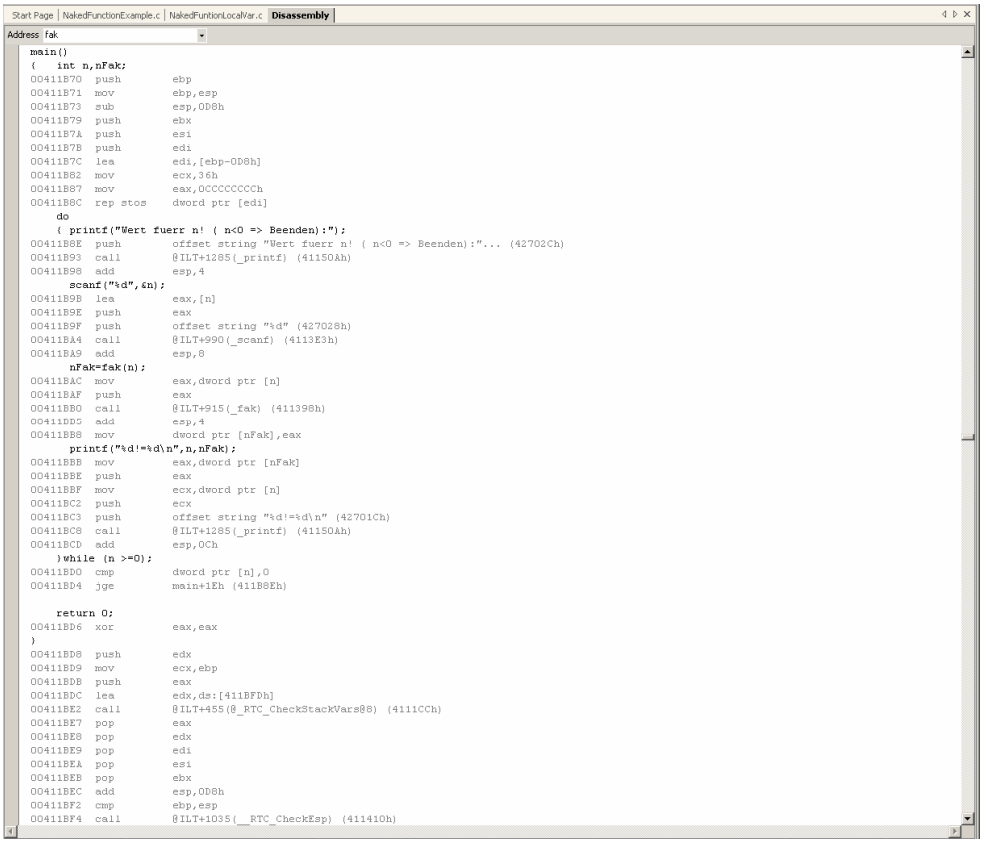


Bild 29: Hauptprogramm mit der Naked Function mit Lokalvariable nach Beispiel 8.

Literaturhinweise:

[MSQ-1] Microsoft Knowlwedge Base Q155763
HOWTO: Call 16-bit Code from 32-bit Code Under Windows 95, Windows 98, and Windows Me.

[MSAMPG-92] Microsoft MASM 6.1 Programmer's Guide, Microsoft Corporation 1992, Originaldokumentation zu MASM 6.1.

[BORASMBH-92] Borland Turbo Assembler Benutzerhandbuch, Borland GmbH 1992, Originaldokumentation zu TASM, Bestandteil von Turbo Pascal für Windows 1.0

[BORASMRH-92] Borland Turbo Assembler Referenzhandbuch, Borland GmbH 1992, Originaldokumentation zu TASM, Bestandteil von Turbo Pascal für Windows 1.0

Ergänzend oder produkteorientiert:

[RHO01] Assembler Ge-Packt, Joachim Rhode, mitp-Verlag 2001, ISBN 3-8266-0786-4 (Taschenbuch)

[ROM03] Assembler - Grundlagen der Programmierung, Marcus Roming/ Joachim Rhode, mitp-Verlag 2003, ISBN 3-8266-0671-X

[MÜL02] Assembler - Die Profireferenz, Oliver Müller Franzis Verlag 2002, 3. Auflage, ISBN 3-7723-7507-32002

[POD95] Das Assemblerbuch, Trutz Eyke Podschun, Addison Wesley 1995, 2. Auflage, ISBN 3-89319-853-9

[BAC02] Programmiersprache Assembler – Eine strukturierte Einführung, Rainer Backer, rororo Rowohlt 9.Auflage 2002, ISBN 3-89319-853-9 (Taschenbuch)

[RUS82] Das 8086/8088 Buch – Programmieren in Assembler und Systemarchitektur, Russel Rector/ George Alexy, tewi Verlag 1982, ISBN 3-921803-X

[BRA86] Programmieren in Assembler für die IBM Personalcomputer, David H. Bradley, Verlag Carl Hanser 1986, ISBN 3-446-14275-4

Mikrocontroller:

[MAN00] C für Mikrocontroller, Burkhard Mann, Franzis Verlag 2000, 3-7723-4254-3
(Schwerpunkt IAR-C Compiler mit AVR- und 51er-Mikrocontroller)

[BAL92] MC Tools 7 – Der Keil C-Compiler ab V3.0 Einführung und Praxis, Michael Baldischweiler, Verlag Feger & Co 1992, ISBN 3-928434-10-1
(Nur Keil C-51)

Online:

[MSDN]
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccore98/html/_core_floating_point_coprocessor_and_calling_conventions.asp

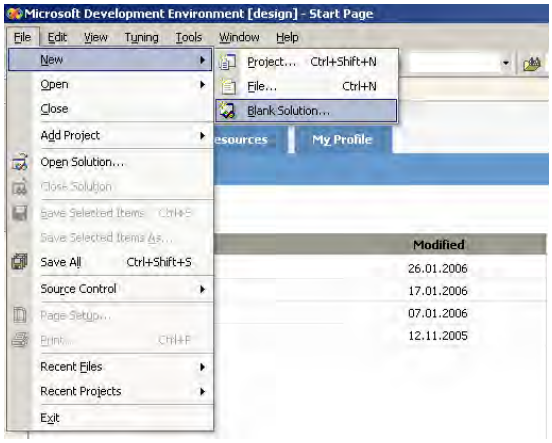
Assemblermodule in MS VC7

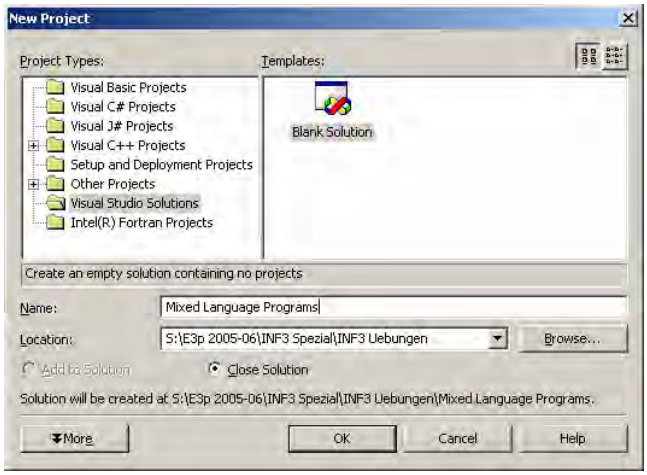
Die nachfolgenden Ausführen zeigen die Schritte zum Erstellen eines C Programmes mit einem externen Assemblermodul mit der Programmierplattform MS Visual Studio 2003 .NET.

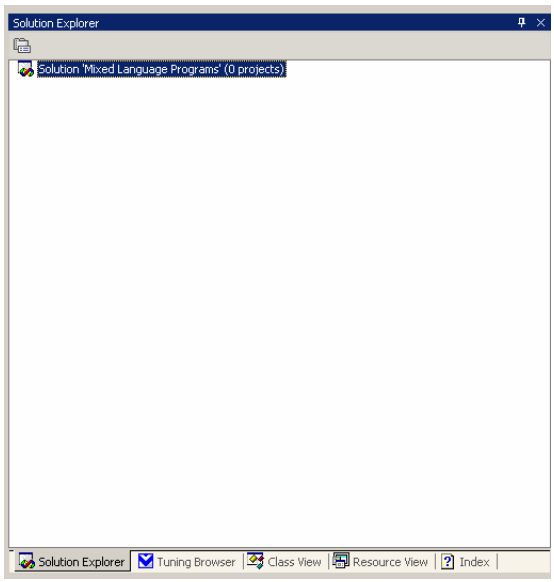
Das Programm ruft aus der Hochsprache eine in Assembler codierte Funktion zur Ausgabe eines String auf. Im Assemblermodul wird dann für die Ausgabe wiederum seinerseits wieder auf eine Bibliotheksfunktion von C Bezug genommen.

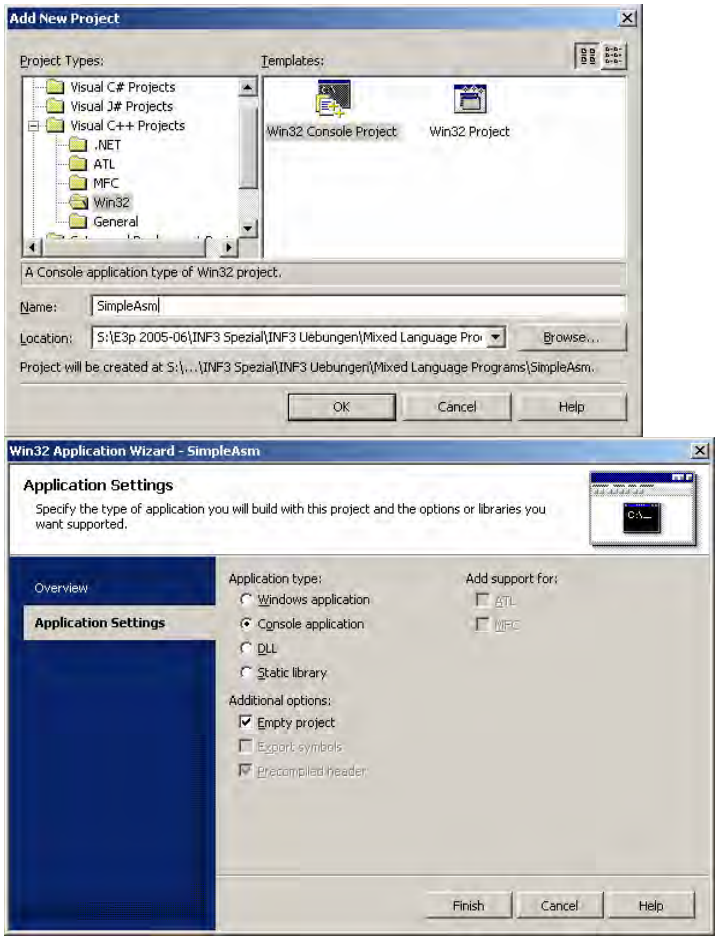
Vorgehen

1. Visual C++ starten. Neues Projekt, ev. neue Solution (Arbeitsmappe) erzeugen.
   Geben Sie der Solution den Namen `Mixed Language Programs`, dem Projekt den Namen `SimpleAsm`.

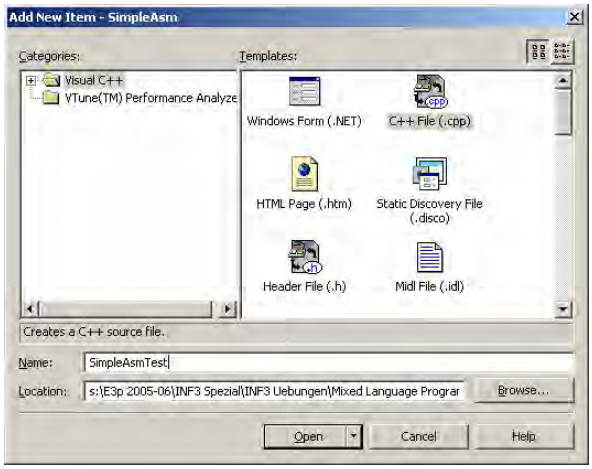








2. Nun C Quelltextfile nach bekanntem Vorgehen erstellen. Wichtig ist, dass das File mit der Extension *.C gespeichert wird, sonst werden die Symbole als dekorierte C++Symbole exportiert.

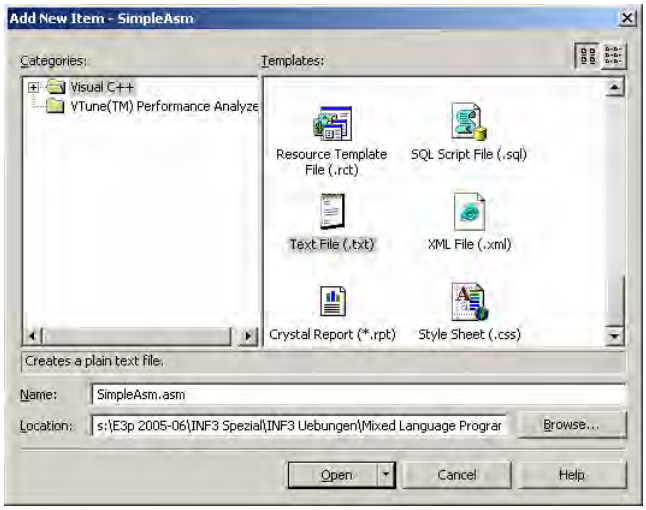


und codieren:

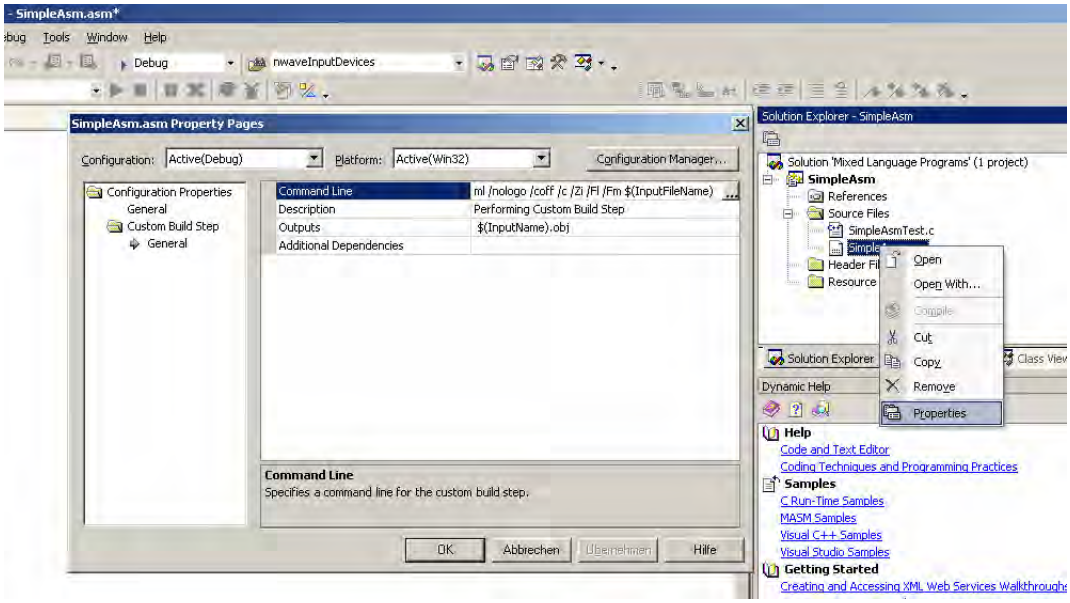
```
Start Page | SimpleAsmTest.c:1 | SimpleAsmTest.c:2 | Disassembly
Address main(void)
--- s:\e3p 2005-06\inf3 spezial\inf3 uebungen\mixed language programs\simpleasm\simpleasmtest.c
/* Testbeispiel zur Gemischsprachprogrammierung C-Assembler x86 mit
   MS Visual Studio 2003 .NET und MASM

   Bemerkung: MASM (ML.EXE) gehört zum Lieferumfang von MS VS .NET und ist im VC7\bin Verzeichnis zu finden.

   Gerhard Krucker
   Zaunackerstrasse 9
   3113 Rubigen

   29-1-2006

   MS Visual Studio 2003 .NET (Win32 ANSI C Console Application)
   */
#include <stdio.h>
#include <conio.h>

int sendData(void);          // Funktionsprototyp fuer die im Assemblermodul extern definierte Funktion


int main(void)
{   int res;
00411AE0  push        ebp
00411AE1  mov         ebp,esp
00411AE3  sub         esp,0CCh
00411AE9  push        ebx
00411AEA  push        esi
00411AEB  push        edi
00411AEC  lea         edi,[ebp-0CCh]
00411AF2  mov         ecx,33h
00411AF7  mov         eax,0CCCCCCCCh
00411AFC  rep stos    dword ptr [edi]

    printf("Mixed language programming example..\n");
00411AFE  push        offset string "Mixed language programming examp"... (42502Ch)
00411B03  call        @ILT+1255(_printf) (4114ECh)
00411B08  add         esp,4
    res=sendData();                                 // Datenwert vom Assemblermodul holen
00411B0B  call        @ILT+145(_sendData) (411096h)
00411B10  mov         dword ptr [res],eax
    printf("Result: %d\n",res);
00411B13  mov         eax,dword ptr [res]
00411B16  push        eax
00411B17  push        offset string "Result: %d\n" (42501Ch)
00411B1C  call        @ILT+1255(_printf) (4114ECh)
00411B21  add         esp,8

    while (!_kbhit());
00411B24  call        @ILT+225(__kbhit) (411OE6h)
00411B29  test        eax,eax
00411B2B  jne         main+4Fh (411B2Fh)
00411B2D  jmp         main+44h (411B24h)
    return 0;
00411B2F  xor         eax,eax
}
00411B31  pop         edi
00411B32  pop         esi
00411B33  pop         ebx
00411B34  add         esp,0CCh
00411B3A  cmp         ebp,esp
00411B3C  call        @ILT+1015(__RTC_CheckEsp) (4113FCh)
00411B41  mov         esp,ebp
00411B43  pop         ebp
00411B44  ret
--- No source file -----------------------------------------------------------
00411B45  int         3
00411B46  int         3
00411B47  int         3
```

```
00411B5D  int         3
00411B5E  int         3
00411B5F  int         3
--- s:\E3p 2005-06\INF3 Spezial\INF3 Uebungen\Mixed Language Programs\SimpleAsm\SimpleAsm.asm
                .386
                .model flat,c       ; 32-Bit Adressraum, C-Namenkonventionen
                                    ; (Case sensitive, _ Underscore vor Namen)
                public c sendData   ; Funktionsnamen exportieren

                .code
sendData        proc

                mov eax,1234
00411B60  mov         eax,4D2h

                ret                 ; Resultat liegt schon EAX
00411B65  ret
--- No source file -----------------------------------------------------------
```

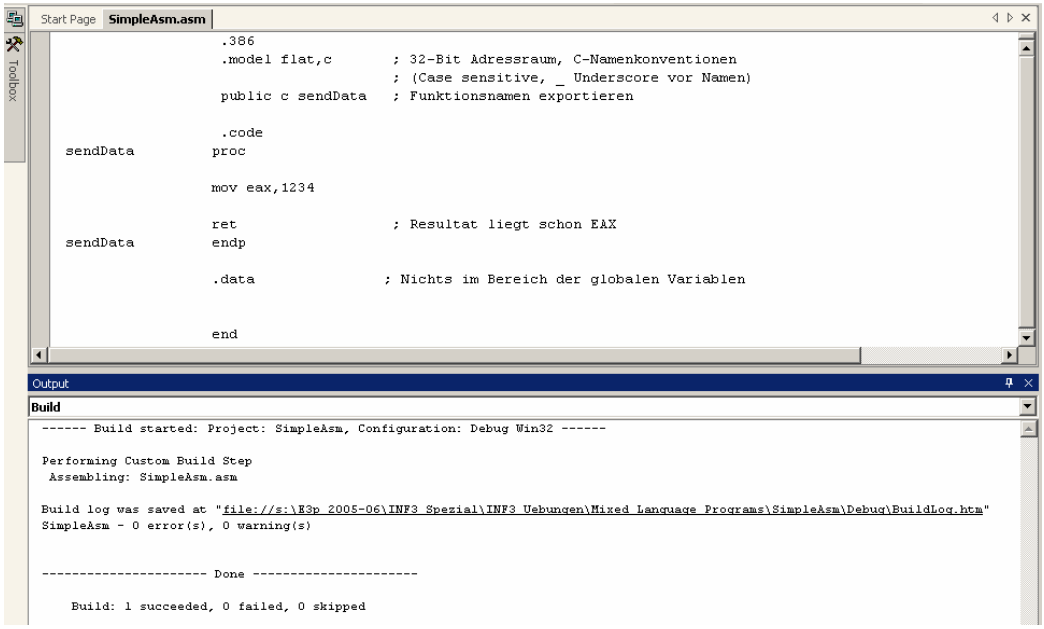
3. Nun das Assemblerquelltextfile erstellen. Es wird wie ein C-File erstellt, nur das man hier Als Typvorgabe “Text File“ wählt und die Extension .asm benutzt:



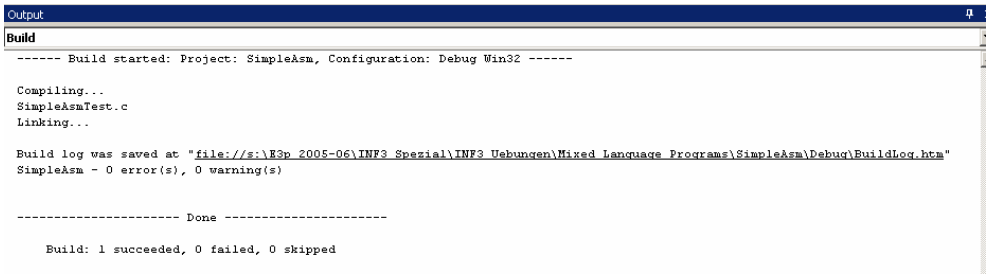
Damit das File im Projekt automatisch assembliert wird, müssen dem File im Solution Explorer die Eigenschaften eingestellt werden.
Hierzu bei `SimpleAsm.asm` mit der rechten Maustaste die Properties öffnen und im Eintrag "General" die Kommandozeile und das Objektverzeichnis genauso eingeben wie nachfolgend gezeigt. Dieser Eintrag ist für alle Assemblerfiles genauso vorzunehmen.



4. Nun den Assemblercode eingeben. Hier retourniert die aus C aufgerufene Funktion den Wert 0x1234 im Register EAX. Dann das Modul mit „Compile" assemblieren. Es sollte ohne Fehlermeldung erstellt ein Objektmodul `SimpleAsm.obj` werden



5. Nun auch das C-File kompilieren und die gesamte Anwendung builden.
   Ein Linkerfehler deutet auf eine Fehler in der Namenschreibung der Funktion hin. Die kann im C-Modul oder im Assemblermodul liegen.



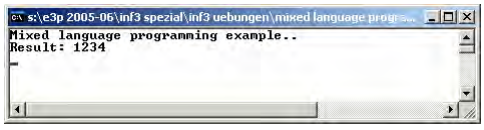
6. Dann das Programm mit einem Breakpoint am Ende starten. Im Disassembly-Window kann der Code des Assemblermoduls und der Hochsprache direkt angeschaut werden. Ebenso können auch im Assemblermodul Haltepunkte gesetzt werden.

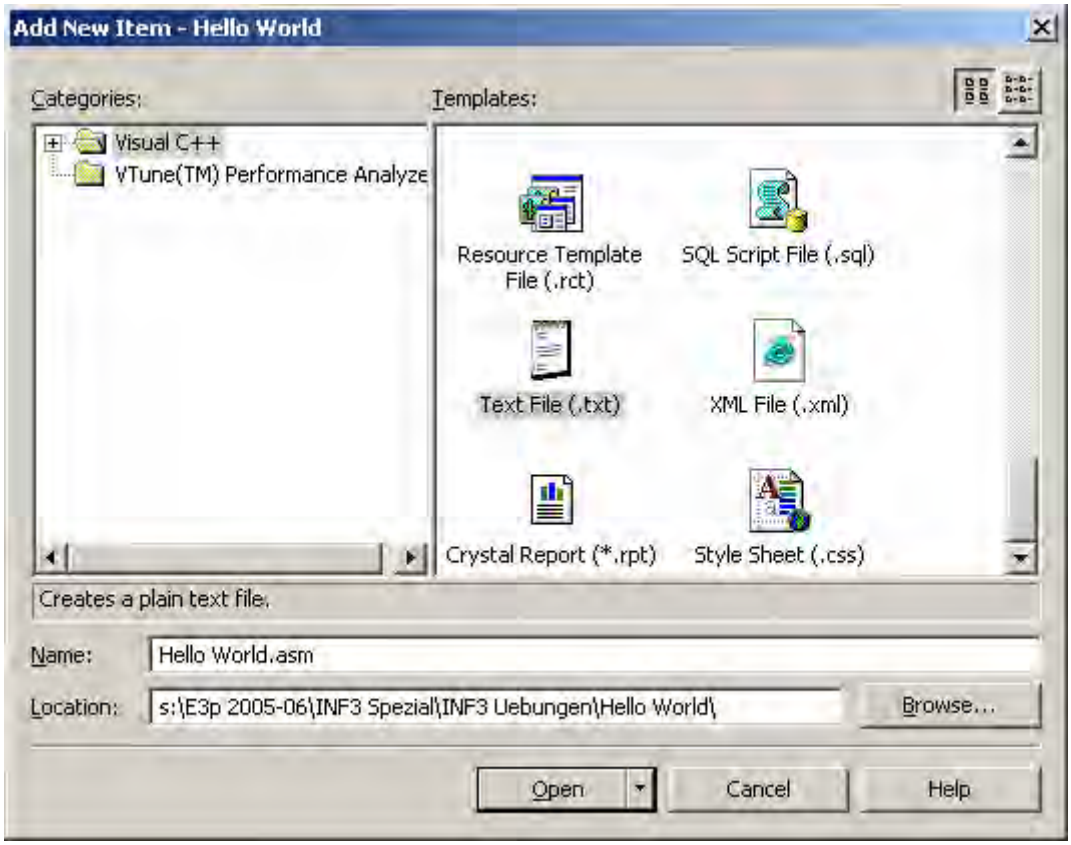
Start Page | SimpleAsm.asm | SimpleAsmTest.c | **Disassembly**

Address: main(void)

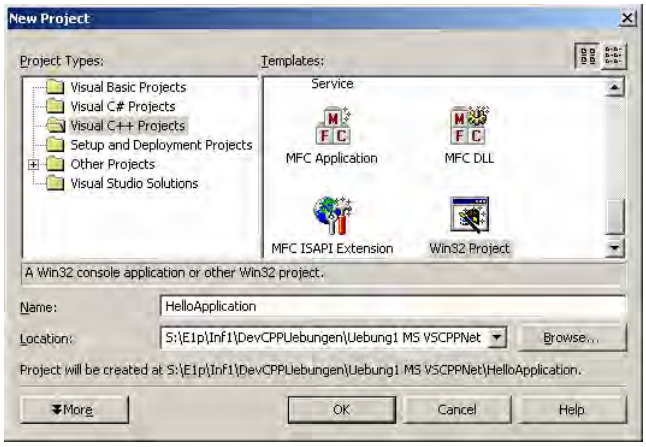
```
00411A3F  int         3
--- s:\e3p 2005-06\inf3 spezial\inf3 uebungen\mixed language programs\simpleasm\simpleasmtest.c
/* Testbeispiel zur Gemischsprachprogrammierung C-Assembler x86 mit
   MS Visual Studio 2003 .NET und MASM

   Bemerkung: MASM (ML.EXE) gehört zum Lieferumfang von MS VS .NET und ist im VC7\bin Verzeichnis zu finden.

   Gerhard Krucker
   Zaunackerstrasse 9
   3113 Rubigen

   29-1-2006

   MS Visual Studio 2003 .NET (Win32 ANSI C Console Application)
   */
#include <stdio.h>

int sendData(void);           // Funktionsprototyp fuer die im Assemblermodul extern definierte Funktion


int main(void)
{   int res;
00411A40  push        ebp
00411A41  mov         ebp,esp
00411A43  sub         esp,0CCh
00411A49  push        ebx
00411A4A  push        esi
00411A4B  push        edi
00411A4C  lea         edi,[ebp-0CCh]
00411A52  mov         ecx,33h
00411A57  mov         eax,0CCCCCCCCh
00411A5C  rep stos    dword ptr [edi]

    printf("Mixed language programming example..\n");
00411A5E  push        offset string "Mixed language programming examp"... (4240Ch)
00411A63  call        @ILT+1175(_printf) (41149Ch)
00411A68  add         esp,4
    res=sendData();                                 // Datenwert vom Assemblermodul holen
00411A6B  call        @ILT+145(_sendData) (411096h)
00411A70  mov         dword ptr [res],eax
    printf("MResult: %d\n",res);
00411A73  mov         eax,dword ptr [res]
00411A76  push        eax
00411A77  push        offset string "MResult: %d\n" (4240Ch)
00411A7C  call        @ILT+1175(_printf) (41149Ch)
00411A81  add         esp,8

    return 0;
00411A84  xor         eax,eax
}
00411A86  pop         edi
00411A87  pop         esi
00411A88  pop         ebx
00411A89  add         esp,0CCh
00411A8F  cmp         ebp,esp
00411A91  call        @ILT+940(__RTC_CheckEsp) (4113B1h)
00411A96  mov         esp,ebp
00411A98  pop         ebp
00411A99  ret
--- No source file -----------------------------------------------------------
00411A9A  int         3
00411A9B  int         3
00411A9C  int         3
00411A9D  int         3
```

```
00411AAB  int         3
00411AAC  int         3
00411AAD  int         3
00411AAE  int         3
00411AAF  int         3
--- s:\E3p 2005-06\INF3 Spezial\INF3 Uebungen\Mixed Language Programs\SimpleAsm\SimpleAsm.asm
                .386
                .model flat,c       ; 32-Bit Adressraum, C-Namenkonventionen
                                    ; (Case sensitive, _ Underscore vor Namen)
                public c sendData   ; Funktionsnamen exportieren

                .code
sendData        proc

                mov eax,1234
00411AB0  mov         eax,4D2h

                ret                 ; Resultat liegt schon EAX
00411AB5  ret
--- No source file -----------------------------------------------------------
```



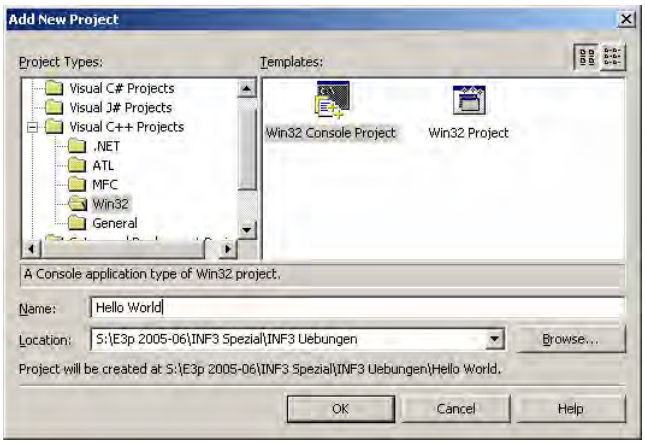


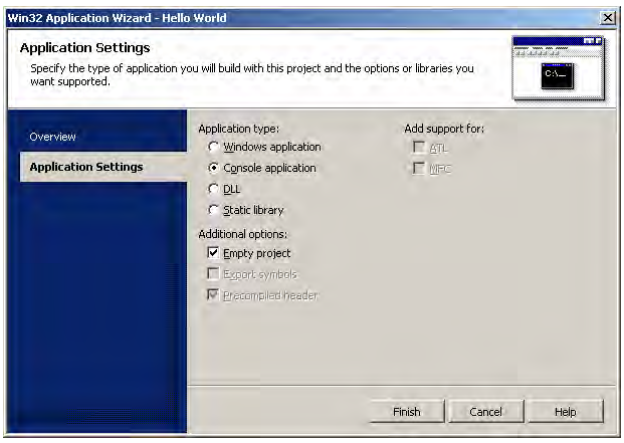
7. Über den Knopf *New Project* wird ein neues Programmierprojekt eröffnet. Das Projekt verwaltet alle zugehörigen Files und Einstellungen. Die Programmerstellung erfolgt immer projektorientiert.

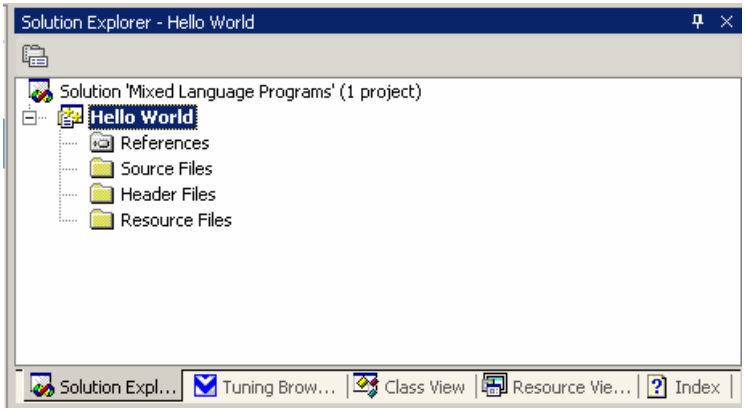


Im Dialogfenster *Visual C++ Projects* und das *Win32 Projec*t Icon anklicken, den Namen für das Projekt eingeben und den Speicherpfad wählen. Mit *OK* das Dialogfenster schliessen.

8. Im folgenden Wizard-Dialog Application Settings anklicken:


Add New Project
Project Types:
Templates:
Visual C# Projects
Visual J# Projects
Visual C++ Projects
.NET
ATL
MFC
Win32
General
Win32 Console Project
Win32 Project
A Console application type of Win32 project.
Name:
Hello World
Location:
S:\E3p 2005-06\INF3 Spezial\INF3 Uebungen
Browse...
Project will be created at S:\E3p 2005-06\INF3 Spezial\INF3 Uebungen\Hello World.
OK
Cancel
Help


Win32 Application Wizard - Hello World
Application Settings
Specify the type of application you will build with this project and the options or libraries you want supported.
Overview
Application Settings
Application type:
Windows application
Console application
DLL
Static library
Additional options:
Empty project
Export symbols
Precompiled header
Add support for:
ATL
MFC
Finish
Cancel
Help


Solution Explorer - Hello World
Solution 'Mixed Language Programs' (1 project)
Hello World
References
Source Files
Header Files
Resource Files
Solution Expl...
Tuning Brow...
Class View
Resource Vie...
Index