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.

Die Prinzipien, Vorgehen und Möglichkeiten sind für alle Produkte und Plattformen etwa gleich. Deshalb werden in den nachfolgenden Kapiteln die Arbeit schwerpunktmässig auf der Basis Borland C++ Builder und mit einigen Beispielen in Microsoft Visual C++ gezeigt.

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 eventuelle 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 Periferie 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. Für den direkten Zugriff auf die Periferie gilt dasselbe wie bei den Objektmodulen.

   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, 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 tiefgehende Systemkenntnisse voraus.

Integrierte Assembler-Anweisungen im C/C++ Code

Inline Statements

Dies ist die einfachste Form Maschinencode direkt in das Programm einzubringen. Die Formulierung erfolgt mit dem Schlüsselwort `inline` gefolgt von den Hex-Codes.

```
inline (0$90 / 0$90 /0$90);        /* 3 NOP Befehle */
```

Innerhalb des Inline-Statements gelten nicht mehr die Syntaxregeln von C/C++, sondern diejenigen des Assembler des jeweiligen Sprachproduktes (hier Turbo-C - BASM). Diese Form der Codierung ist veraltet und hat nur Nachteile. Sie wird auch bei C++ nicht unterstützt da `inline` in C++ ein reserviertes Wort ist.

Microsoft- und Intel-Compiler, wie andere Produkte, unterstützen ähnlich dem obigen `inline`--Statement 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;
```

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.

```
10:   #include <conio.h>
11:
12:   int main()
13:   {  cout << "Beispiel zum direkten Einbinden von Maschinencode in C/C++ Programme" << endl;
00401040   push        ebp
00401041   mov         ebp,esp
00401043   sub         esp,40h
00401046   push        ebx
00401047   push        esi
00401048   push        edi
00401049   lea         edi,[ebp-40h]
0040104C   mov         ecx,10h
00401051   mov         eax,0CCCCCCCCh
00401056   rep stos    dword ptr [edi]
00401058   push        offset @ILT+10(endl) (0040100f)
0040105D   push        offset string "Beispiel zum direkten Einbinden "... (0042801c)
00401062   mov         ecx,offset cout (0042e470)
00401067   call        ostream::operator<< (00401400)
0040106C   mov         ecx,eax
0040106E   call        @ILT+0(ostream::operator<<) (00401005)
14:
15:       _asm _emit 0x90;
00401073   nop
16:       _asm _emit 0x90;
00401074   nop
17:       _asm _emit 0x90;
00401075   nop
18:
19:       while(!_kbhit());
00401076   call        _kbhit (00403660)
0040107B   test        eax,eax
0040107D   jne         main+41h (00401081)
0040107F   jmp         main+36h (00401076)
20:       return 0;
00401081   xor         eax,eax
21:
22:
23:
24:   }
00401083   pop         edi
00401084   pop         esi
00401085   pop         ebx
00401086   add         esp,40h
00401089   cmp         ebp,esp
0040108B   call        __chkesp (00403900)
00401090   mov         esp,ebp
00401092   pop         ebp
00401093   ret
```

Bild 1: Disassemblierte Darstellung des Programms mit _asm-Anweisungen nach Beispiel 1. Plattform: Visual C++ V6.0 mit Intel C++ Compiler V7.1.

`_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 jeder Anweisung zu setzen ist aber kein Fehler.

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

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

```
#include <iostream.h>

int main()
{  cout << "Beispiel zum direkten Einbinden von Assemblercode in C/C++ Programme" << endl;

   _asm nop _asm nop _asm nop      // 3 NOP einbringen - Semikolon sind nicht notwendig.
                                   // (Aber auch nicht falsch)

   return 0;
}
```

Eine alternative Lösung mit einem Block:

```
#include <iostream.h>

int main()
{  cout << "Beispiel zum direkten Einbinden von Assemblercode in C/C++ Programme" << endl;

   _asm { nop                       // 3 NOP einbringen
          nop
          nop
        }

   return 0;
}
```

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

Zugriff auf C-Symbole

Innerhalb der `_asm`-Anweisung kann auf C-Symbole (Variablen, Konstanten und Funktionen zugegriffen werden. Der Programmier 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
// Borland C++ Builder V5.0 , Win32 Console Application
// G. Krucker, 2.1.2004
#pragma hdrstop
#include <iostream.h>
#include <conio.h>  // Fuer kbhit()
//---------------------------------------------------------------------------

int main()
{  int eingabeWert;
   int fakWert;
   cout << "Eingabewert n fuer Fakultaetsberechnung n!:" ;
   cin >> 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
   }
   cout << "Resultat: " << eingabeWert <<"! = " << fakWert << endl;

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

Die Umsetzung des gesamten Programms erfolgt bei Borland C++ Builder:

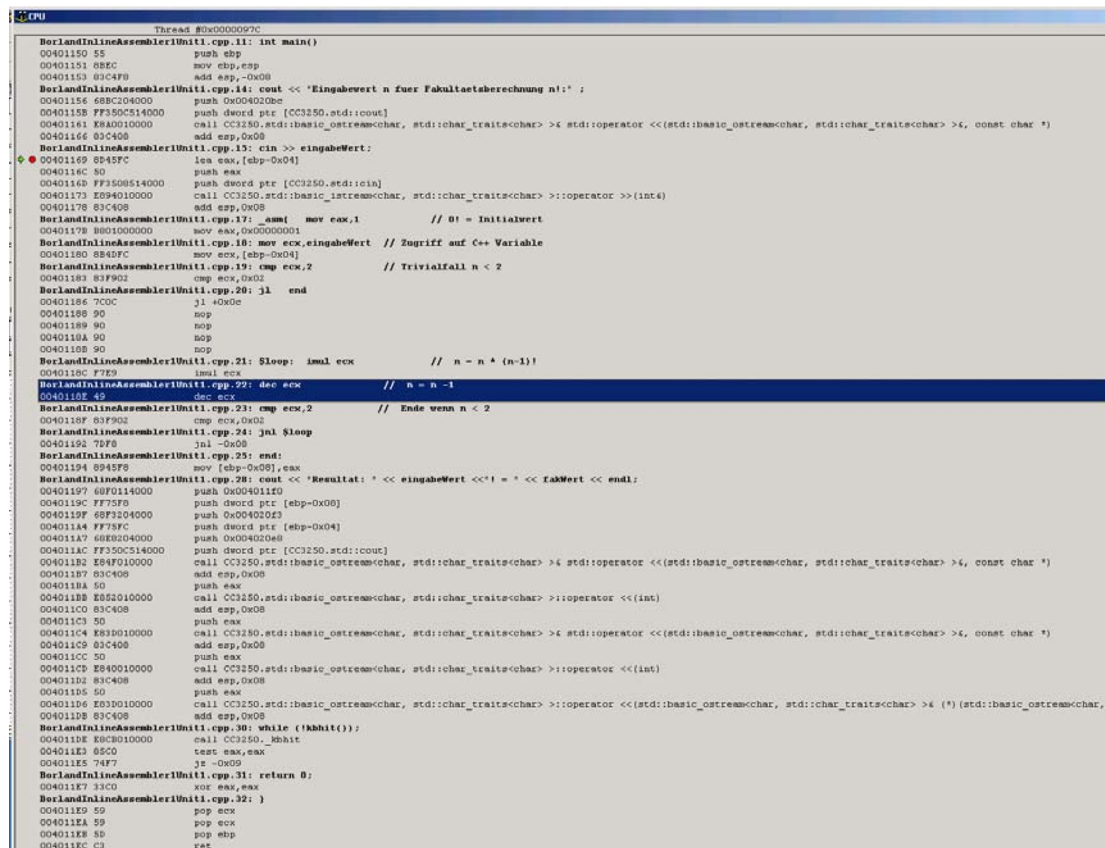


Bild 2: Disassemblierte Darstellung des Programms mit _asm-Anweisung nach Beispiel 3. Plattform: Borland C++ Builder 5.0.

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 und schaut 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.
// Borland C++ Builder V5.0 , Win32 Console Application
// G. Krucker, 2.1.2004
#pragma inline       // Inline-Assembler ohne Meldung verwenden
#include <conio.h>   // Fuer kbhit()
#include <stdio.h>

//-----------------------------------------------------------------------------
// Fuer die Operationen im _asm-Block gilt im Wesentlichen die
// TASM-Sprachdefinition.
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

       };

    // Direkte Ausgabe
    printf("Werte nach _asm-Sequenz:\na=%d, b=%d\n",a,b);

    // Ausgabe in Assembler mit der _printf-Library-Funktion
    _asm{ push b
          push a
          lea eax,fmtStr
          push eax
          call _printf      // Aufruf der print-Libraryfunktion aus CC3250.DLL
          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 Borland C++ Builder 5.0 umgesetzt:

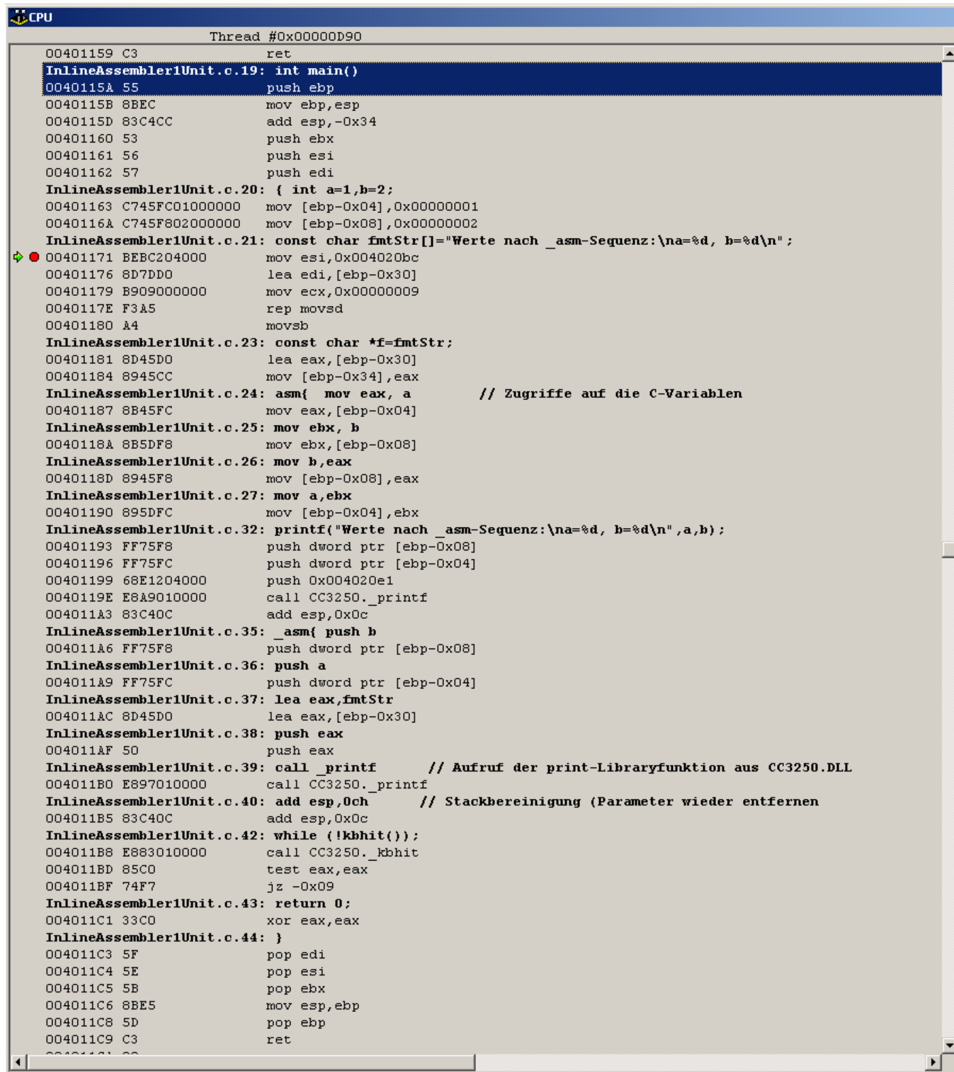


Bild 3: Disassemblierte Darstellung des Programms nach Beispiel 4 mit Aufruf einer Funktion der C-Runtime Library. Plattform: Boland C++ Builder 5.0.

Bei der Verwendung von Inline-Assembler wird bei Borland automatisch ein `*.asm`-File erzeugt. 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.

```
        .386p
        ifdef ??version
        if    ??version GT 500H
        .mmx
        endif
        endif
        model flat
        ifndef    ??version
        ?debug    macro
        endm
        endif
        ?debug    S "\\Wendy\hta-be dok\EB01-1\Informatik 5 Uebergang\Assembler\Borland Inline Assembler I\InlineAssembler1Unit.c"
        ?debug    T "\\Wendy\hta-be dok\EB01-1\Informatik 5 Uebergang\Assembler\Borland Inline Assembler I\InlineAssembler1Unit.c"
_TEXT   segment dword public use32 'CODE'
_TEXT   ends
_DATA   segment dword public use32 'DATA'
_DATA   ends
_BSS    segment dword public use32 'BSS'
_BSS    ends
$$BSYMS segment byte public use32 'DEBSYM'
$$BSYMS ends
$$BTYPES          segment byte public use32 'DEBTYP'
$$BTYPES          ends
$$BNAMES          segment byte public use32 'DEBNAM'
$$BNAMES          ends
$$BROWSE          segment byte public use32 'DEBSYM'
$$BROWSE          ends
$$BROWFILE        segment byte public use32 'DEBSYM'
$$BROWFILE        ends
DGROUP  group     _BSS,_DATA
```

```
_TEXT   segment dword public use32 'CODE'
__getch proc      near
?live1@0:
        ?debug    T "T:\BORLAND\CBUILDER5\INCLUDE\conio.h"
        ?debug L 183
        push      ebp
        mov       ebp,esp
@1:
        call      _getch
        ?debug L 183
@3:
@2:
        pop       ebp
        ret
        ?debug L 0
__getch endp
_TEXT   ends

$$BSYMS segment byte public use32 'DEBSYM'
        db        2
        db        0
        db        0
        db        0
        dw        46
        dw        516
        dw        0
        dw        0
        dw        0
        dw        0
        dw        0
        dw        0
        dd        ?patch1
        dd        ?patch2
        dd        ?patch3
        df        __getch
        dw        0
        dw        4096
        dw        0
        dw        1
        dw        0
        dw        0
        dw        0
?patch1 equ       @3-__getch+2
?patch2 equ       0
?patch3 equ       @3-__getch
        dw        2
        dw        6
$$BSYMS ends

_DATA   segment dword public use32 'DATA'
$imcmcaia         label     byte
        db        87
        db        101
        db        114
        db        116
        db        101
        db        32
        db        110
        db        97
        db        99
        db        104
        db        32
        db        95
        db        97
        db        115
        db        109
        db        45
        db        83
        db        101
        db        113
        db        117
        db        101
        db        110
        db        122
        db        58
        db        10
        db        97
        db        61
        db        37
        db        100
        db        44
        db        32
        db        98
        db        61
        db        37
        db        100
        db        10
        db        0
_DATA   ends
```

```
_TEXT   segment dword public use32 'CODE'
_main   proc      near
?live1@32:
        ?debug    T "\\Wendy\hta-be dok\EB01-1\Informatik 5 Uebergang\Assembler\Borland Inline Assembler I\InlineAssembler1Unit.c"
        ?debug L 19
        push      ebp
        mov       ebp,esp
        add       esp,-52
        push      ebx
        push      esi
        push      edi
        ?debug L 20
@4:
        mov       dword ptr [ebp-4],1
        mov       dword ptr [ebp-8],2
        ?debug L 21
        mov       esi,offset $imcmcaia
        lea       edi,dword ptr [ebp-48]
        mov       ecx,9
        rep   movsd
        movsb
        ?debug L 23
        lea       eax,dword ptr [ebp-48]
        mov       dword ptr [ebp-52],eax
        ?debug L 24
        mov        eax, dword ptr [ebp-4]
        ?debug L 25
        mov        ebx, dword ptr [ebp-8]
        ?debug L 26
        mov        dword ptr [ebp-8],eax
        ?debug L 27
        mov        dword ptr [ebp-4],ebx
        ?debug L 32
        push      dword ptr [ebp-8]
        push      dword ptr [ebp-4]
        push      offset s@
        call      _printf
        add       esp,12
        ?debug L 35
        push       dword ptr [ebp-8]
        ?debug L 36
        push       dword ptr [ebp-4]
        ?debug L 37
        lea        eax,byte ptr [ebp-48]
        ?debug L 38
        push       eax
        ?debug L 39
        call       _printf
        ?debug L 40
        add        esp,0ch
        ?debug L 42
@5:
@6:
        call      _kbhit
        test      eax,eax
        je        short @5
        ?debug L 43
        xor       eax,eax
        ?debug L 44
@8:
@7:
        pop       edi
        pop       esi
        pop       ebx
        mov       esp,ebp
        pop       ebp
        ret
        ?debug L 0
_main   endp
_TEXT   ends

$$BSYMS segment byte public use32 'DEBSYM'
        dw        52
        dw        517
        dw        0
        dw        0
        dw        0
        dw        0
        dw        0
        dw        0
        dd        ?patch4
        dd        ?patch5
        dd        ?patch6
        df        _main
        dw        0
        dw        4098
        dw        0
        dw        2
        dw        0
        dw        0
        dw        0
        db        5
        db        95
        db        109
        db        97
        db        105
        db        110
        dw        18
        dw        512
        dw        65484
        dw        65535
        dw        4100
        dw        0
        dw        3
        dw        0
        dw        0
        dw        0
        dw        18
        dw        512
        dw        65488
        dw        65535
        dw        4102
        dw        0
```

```
        dw      4
        dw      0
        dw      0
        dw      0
        dw      18
        dw      512
        dw      65528
        dw      65535
        dw      116
        dw      0
        dw      5
        dw      0
        dw      0
        dw      0
        dw      18
        dw      512
        dw      65532
        dw      65535
        dw      116
        dw      0
        dw      6
        dw      0
        dw      0
        dw      0
?patch4 equ     @8-_main+7
?patch5 equ     0
?patch6 equ     @8-_main
        dw      2
        dw      6
        dw      8
        dw      531
        dw      7
        dw      65472
        dw      65535
$$BSYMS ends

_DATA   segment dword public use32 'DATA'
s@      label   byte
        ;       s@+0:
        db      "Werte nach _asm-Sequenz:",10
        ;       s@+25:
        db      "a=%d, b=%d",10,0
        align   4
_DATA   ends

_TEXT   segment dword public use32 'CODE'
_TEXT   ends

 extrn   _getch:near
        public  _main
 extrn   _printf:near
 extrn   _kbhit:near
$$BSYMS segment byte public use32 'DEBSYM'
        dw      16
        dw      4
        dw      117
        dw      0
        dw      0
        dw      7
        dw      0
        dw      0
        dw      0
        dw      16
        dw      4
        dw      116
        dw      0
        dw      0
        dw      8
        dw      0
        dw      0
        dw      0
        dw      16
        dw      4
        dw      33
        dw      0
        dw      0
        dw      9
        dw      0
        dw      0
        dw      0
        dw      16
        dw      4
        dw      33
        dw      0
        dw      0
        dw      10
        dw      0
        dw      0
        dw      0
        dw      16
        dw      4
        dw      33
        dw      0
        dw      0
        dw      11
        dw      0
        dw      0
        dw      0
        dw      16
        dw      4
        dw      18
        dw      0
        dw      0
        dw      12
        dw      0
        dw      0
        dw      0
        dw      ?patch7
        dw      1
        db      3
        db      0
        db      0
        db      24
```

```
        db      11
        db      66
        db      67
        db      67
        db      51
        db      50
        db      32
        db      53
        db      46
        db      53
        db      46
        db      49
?patch7 equ     18
$$BSYMS ends

$$BTYPES        segment byte public use32 'DEBTYP'
        db      2,0,0,0,14,0,8,0,116,0,0,0,0,0,0,0
        db      1,16,0,0,4,0,1,2,0,0,14,0,8,0,116,0
        db      0,0,0,0,0,0,3,16,0,0,4,0,1,2,0,0
        db      8,0,2,0,10,0,5,16,0,0,8,0,1,0,1,0
        db      16,0,0,0,8,0,1,0,1,0,7,16,0,0,18,0
        db      3,0,16,0,0,0,17,0,0,0,0,0,0,0,37,0
        db      37,0,14,0,8,0,116,0,0,0,0,0,0,0,9,16
        db      0,0,4,0,1,2,0,0,14,0,8,0,116,0,0,0
        db      64,0,2,0,11,16,0,0,12,0,1,2,2,0,4,16
        db      0,0,0,0,0,0,14,0,8,0,116,0,0,0,0,0
        db      0,0,13,16,0,0,4,0,1,2,0,0
$$BTYPES        ends

$$BNAMES        segment byte public use32 'DEBNAM'
        db      6,'_getch'
        db      4,'main'
        db      1,'f'
        db      6,'fmtStr'
        db      1,'b'
        db      1,'a'
        db      6,'size_t'
        db      9,'ptrdiff_t'
        db      7,'wchar_t'
        db      6,'wint_t'
        db      8,'wctype_t'
        db      6,'fpos_t'
$$BNAMES        ends

        ?debug  D "T:\BORLAND\CBUILDER5\INCLUDE\_nfile.h" 10307 8192
        ?debug  D "T:\BORLAND\CBUILDER5\INCLUDE\stdio.h" 10307 8192
        ?debug  D "T:\BORLAND\CBUILDER5\INCLUDE\_null.h" 10307 8192
        ?debug  D "T:\BORLAND\CBUILDER5\INCLUDE\_defs.h" 10307 8192
        ?debug  D "T:\BORLAND\CBUILDER5\INCLUDE\_stddef.h" 10307 8192
        ?debug  D "T:\BORLAND\CBUILDER5\INCLUDE\conio.h" 10307 8192
        ?debug  D "\\Wendy\hta-be dok\EB01-1\Informatik 5 Uebergang\Assembler\Borland Inline Assembler I\InlineAssembler1Unit.c" 12323 29287
        end
```

Beispiel 5: String-Funktionsaufruf in _asm Anweisung.

Das folgende Beispiel zeigt die Benutzung einer C-Stringfunktion. 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.

```
//---------------------------------------------------------------------------
#include <iostream.h>
#pragma inline          // Inline Assembler ohne Meldung aktivieren
#pragma hdrstop
#include <string.h>
//---------------------------------------------------------------------------

int main()
{ int i;
  char string[]="Hallo";
  int len;
   cout << "Zugriff auf Stringfunktionen:" << endl;
   len=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;
}
```

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 Borland 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.

Beispiele zur Verwendung externer Assemblermodule sind in Beispiel 6 und der Übung zu diesem Thema gezeigt.

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.

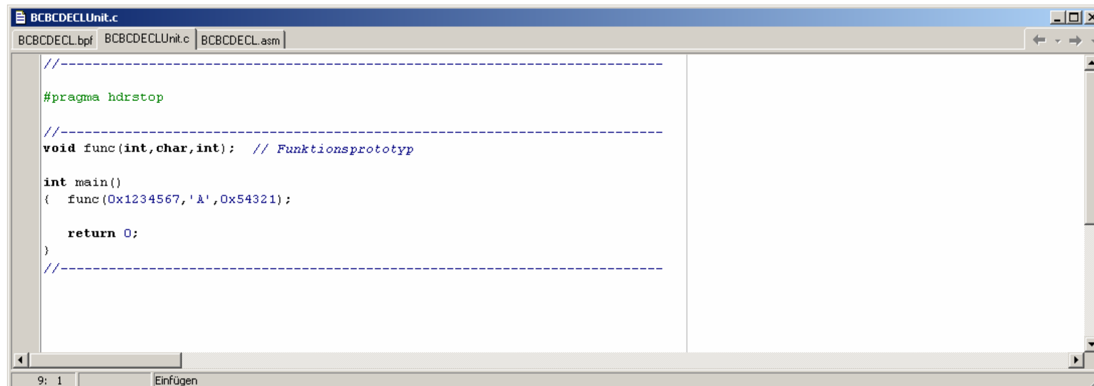


Bild 4: Parameterübergabe nach C. Im Hochsprachenteil wird die externe Funktion über den Prototyp definiert.

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.

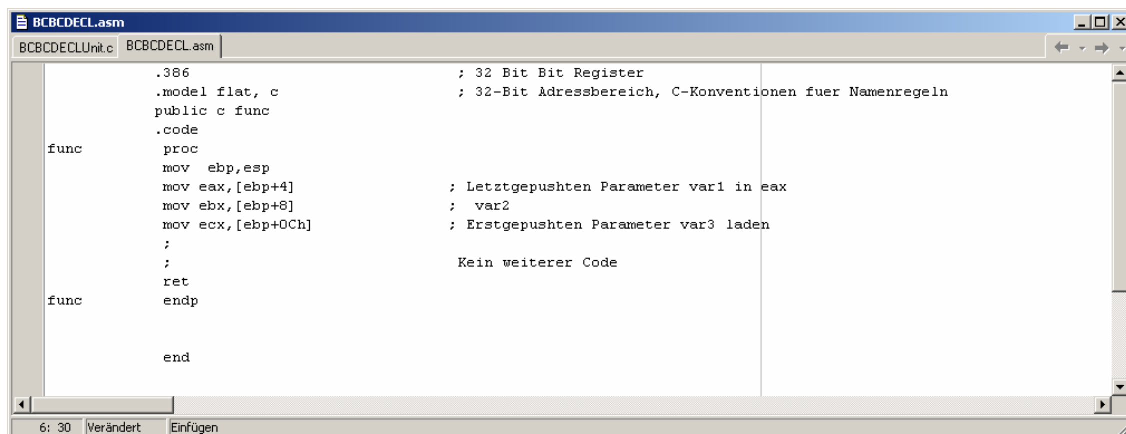


Bild 5: Assemblermodul zur Parameterübergabe nach C. Die Funktion wird mit `proc c` definiert. Der Funktionsname wird mit `public c` exportiert.

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.

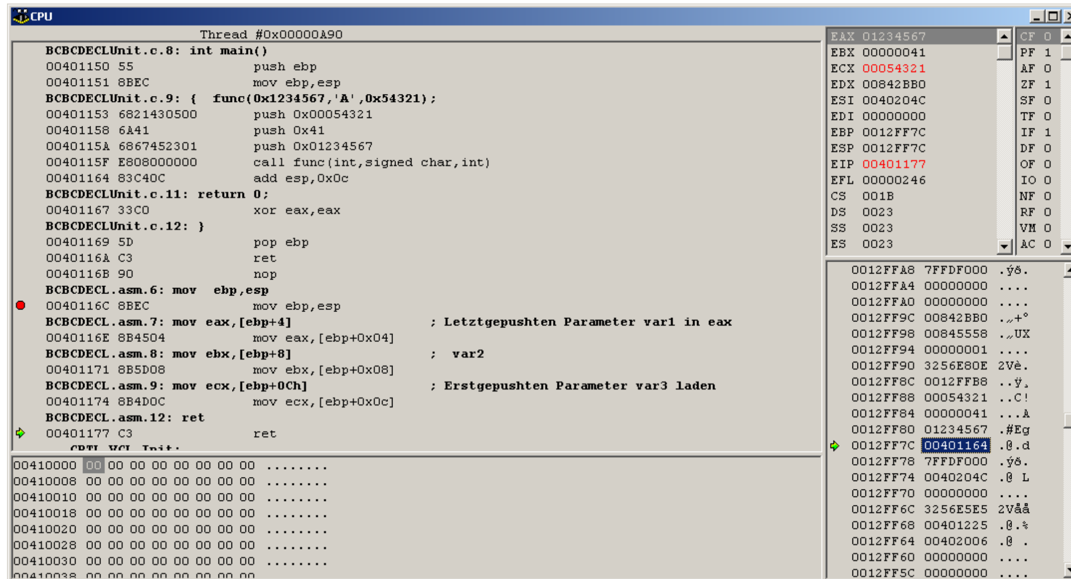


Bild 6: Programmbeispiel mit Parameterübergabe nach C im Debugger. Plattform: Borland C++ Builder 5.0

PASCAL

Bei Pascal-Konvention werden die Parameter in Reihenfolge der Definition auf den Stack gelegt. Die Stackbereinigung erfolgt durch die aufgerufene Einheit selbst.
Praktisch alle Betriebssystemfunktionen von Windows arbeiten mit Pascal-Übergaben, weil dies eine kleine Einsparung an Code bringt.

Für das vorherige Beispiel

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

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

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

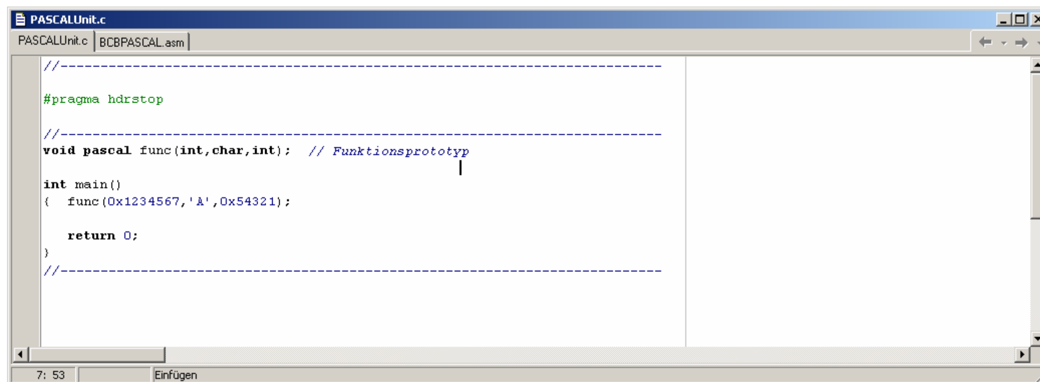


Bild 7: Parameterübergabe nach Pascal. Im Hochsprachenteil wird die externe Funktion über den Prototyp mit pascal definiert.

Assemblermodul:
Die Funktion wird hier als `proc pascal` definiert. Ebenso die Publikation `public pascal FUNC`. Namen werden hier in Grossschrift geschrieben. Pascal-bezogene Symbole werden in den einzelnen Modulen immer in Grosschrift importiert und exportiert.

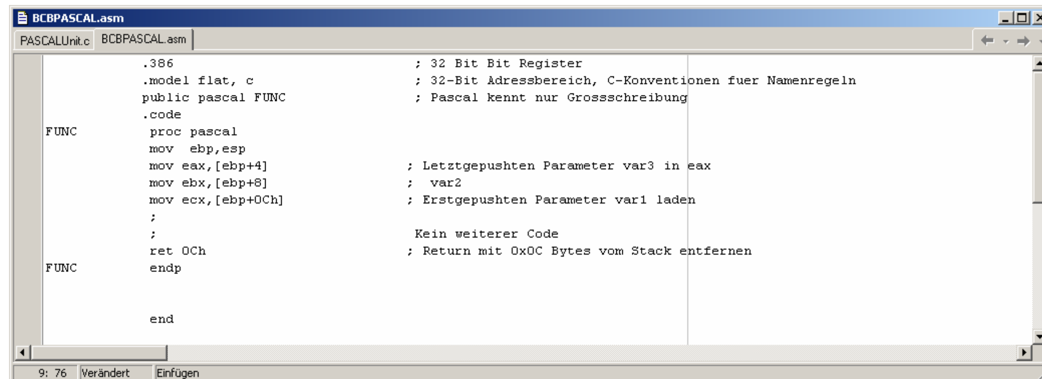


Bild 8: Assemblermodul zur Parameterübergabe nach Pascal. Die Funktion wird mit `proc pascal` definiert. Der Funktionsname wird mit `public pascal` exportiert.

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).

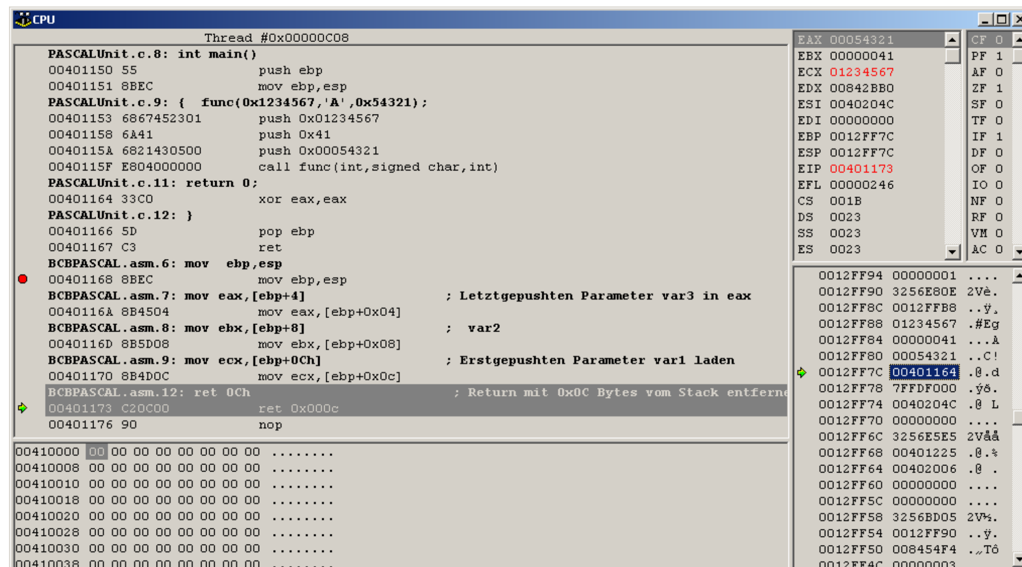


Bild 9: Programmbeispiel mit Parameterübergabe nach Pascal im Debugger. Plattform: Borland C++ Builder 5.0

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 `pascal` und `c` sind sicherlich die beiden wichtigsten Verfahren. Aus Gründen der Kompatibilität zu anderen Programmiersprachen existieren 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. Detaillierte Informationen sind in den produktespezifischen Programmierhandbüchern zu finden. [MASMPG-92],[BORASMBH-92],[BORASMRH-92].

Funktionswertrückgaben

Funktionswerte werden je nach Typ direkt über Prozessorregister oder über Zeiger auf eine Variable übergeben.

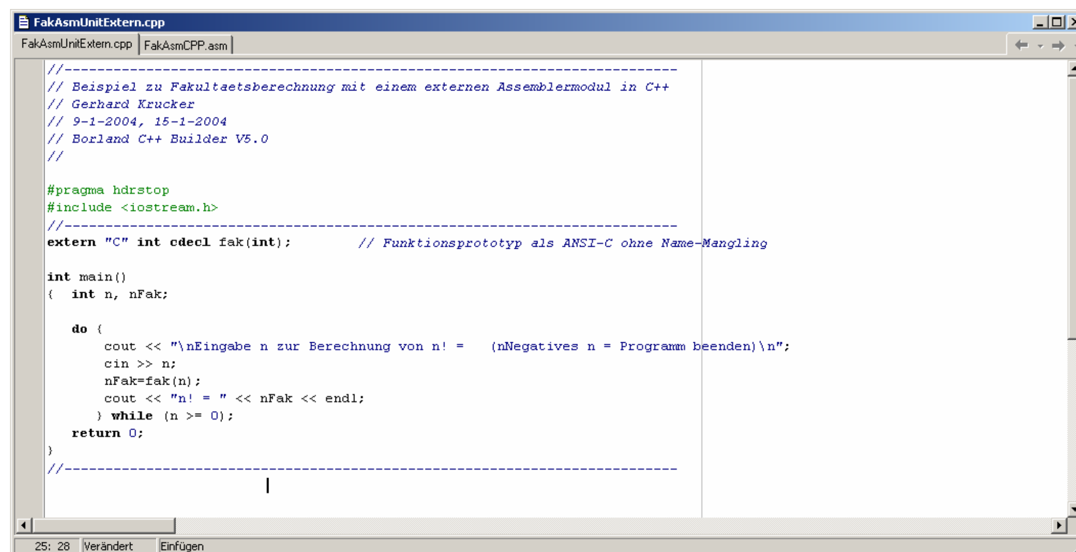
char	al	8 Bit
short	ax	16 Bit
long	eax	32 Bit
Zeiger	eax	32 Bit
Struct	Bis 4 Bytes: eax	
	>4 Bytes über Variablenreferenz	
Array	Zeiger in eax	32 Bit

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

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:



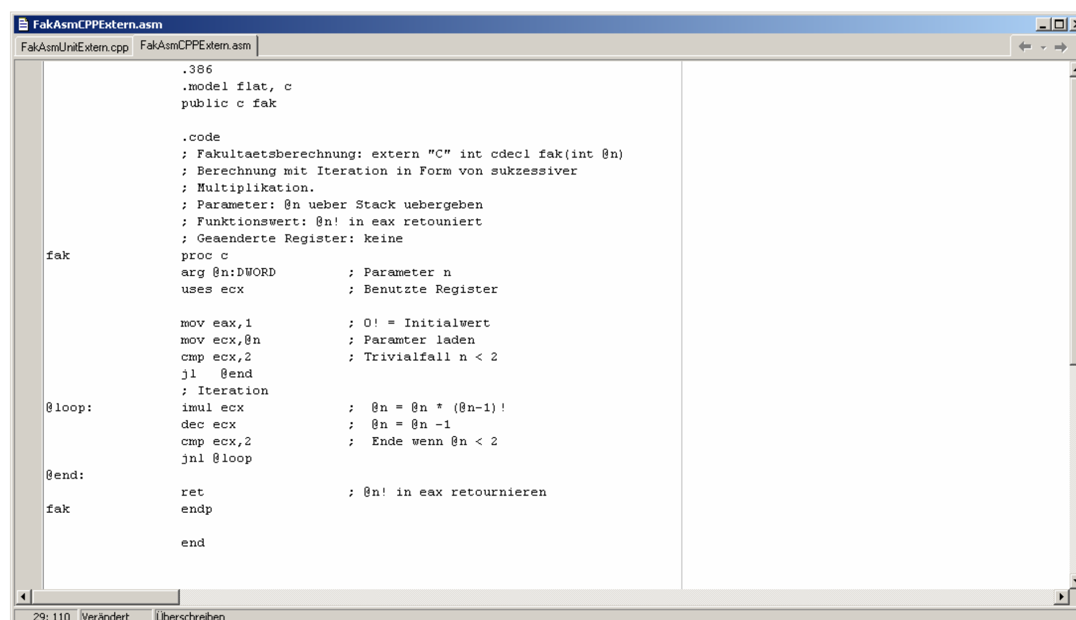
FakAsmUnitExtern.cpp

```
//---------------------------------------------------------------------------
// Beispiel zu Fakultaetsberechnung mit einem externen Assemblermodul in C++
// Gerhard Krucker
// 9-1-2004, 15-1-2004
// Borland C++ Builder V5.0
//

#pragma hdrstop
#include <iostream.h>
//---------------------------------------------------------------------------
extern "C" int cdecl fak(int);         // Funktionsprototyp als ANSI-C ohne Name-Mangling

int main()
{   int n, nFak;

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



FakAsmCPPExtern.asm

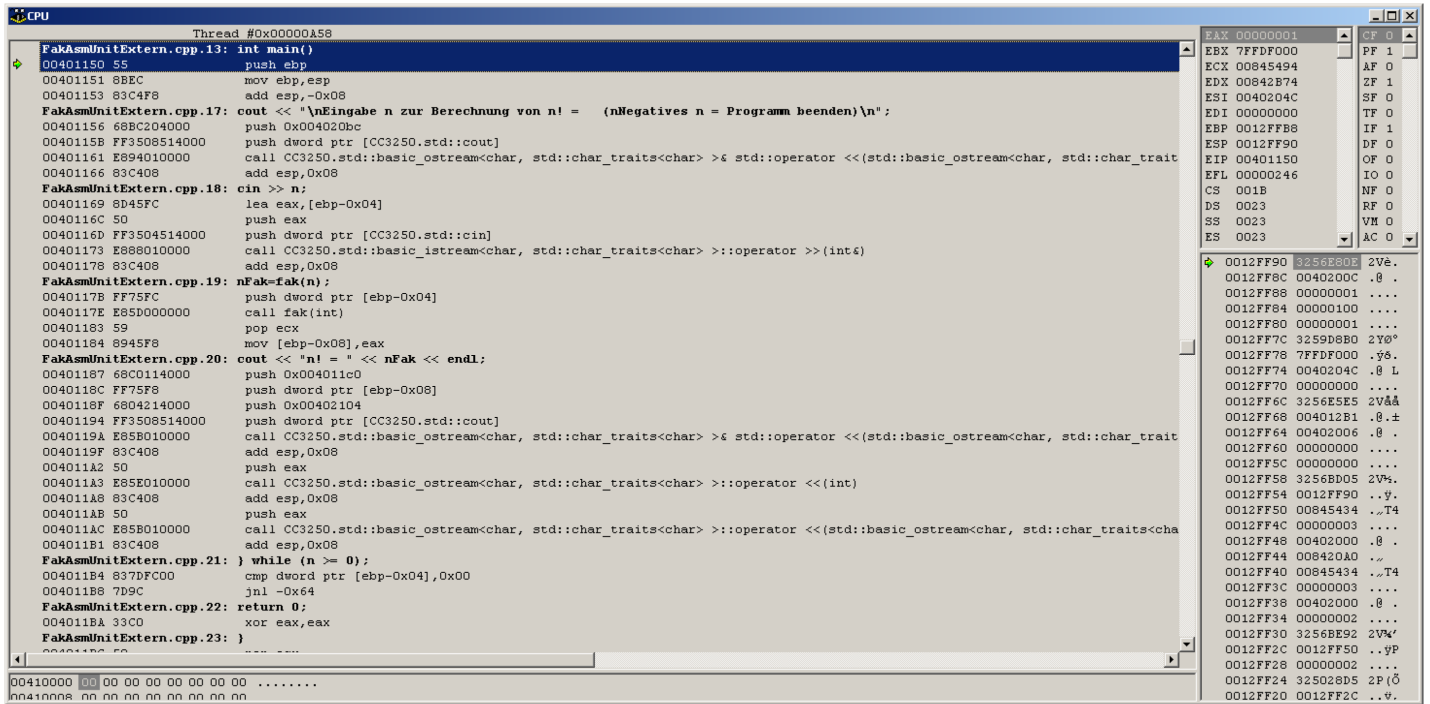
```
                .386
                .model flat, c
                public c fak

                .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
                arg @n:DWORD            ; Parameter n
                uses ecx                ; Benutzte Register

                mov eax,1               ; 0! = Initialwert
                mov ecx,@n              ; Paramter laden
                cmp ecx,2               ; Trivialfall n < 2
                jl   @end
                ; Iteration
@loop:          imul ecx                ;  @n = @n * (@n-1)!
                dec ecx                 ;  @n = @n -1
                cmp ecx,2               ;  Ende wenn @n < 2
                jnl @loop
@end:
                ret                     ; @n! in eax retournieren
fak             endp

                end
```

Bild 10: Parameterübergabe und Funktionswertrückgabe bei einer extern “C“ definierten Funktion nach Beispiel 6.



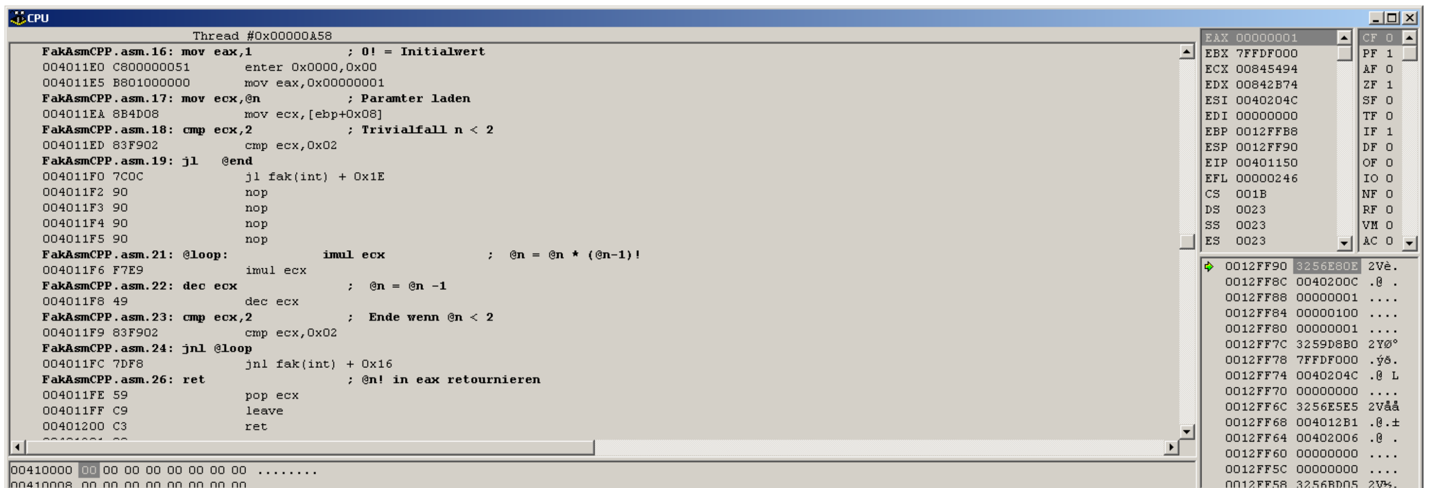


Bild 11: Debuggerfenster der CPU mit Darstellung des erzeugten Codes nach Beispiel 6. Aus Platzgründen wurden die Funktionen main und fak in separaten Fenstern dargestellt.

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$qi`. Die Funktion muss auf Stufe Assembler als `syscall` definiert werden. `syscall` bewirkt, dass der Name casesensitive behandelt wird, aber ohne Underscore exportiert wird. Die Stackbereinigung erfolgt bei `syscall` wie bei `c` durch den Aufrufer. (Vgl. auch [MSAMPG-92], S.309)

FakAsmUnit.cpp

```
//---------------------------------------------------------------------------
// Beispiel zu Fakultaetsberechnung mit einem externen Assemblermodul in C++
// Gerhard Krucker
// 9-1-2004, 15-1-2004
// Borland C++ Builder V5.0
//

#pragma hdrstop
#include <iostream.h>
//---------------------------------------------------------------------------
int cdecl fak(int);        // Funktionsprototyp als ANSI-C mit Name-Mangling

int main()
{   int n, nFak;

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

24: 49 Verändert Einfügen

FakAsmCPP.asm

```
                .386
                .model flat, syscall
                public syscall @fak$qi     ; C++ Name mit Parameter-/Typinformation

                .code
                ; Fakultaetsberechnung: extern 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$qi         proc syscall
                arg @n:DWORD         ; Parameter n
                uses ecx             ; Benutzte Register

                mov eax,1            ; 0! = Initialwert
                mov ecx,@n           ; Paramter laden
                cmp ecx,2            ; Trivialfall n < 2
                jl   @end
                ; Iteration
@loop:          imul ecx             ;  @n = @n * (@n-1)!
                dec ecx              ;  @n = @n -1
                cmp ecx,2            ;  Ende wenn @n < 2
                jnl @loop
@end:
                ret                  ; @n! in eax retournieren
@fak$qi         endp

                end
```

3: 44 Verändert Einfügen

Bild 12: Parameterübergabe und Funktionswertrückgabe bei einer C++ definierten Funktion nach Beispiel 6. Der Funktionsname in Assembler trägt die Namenerweiterung @..$qi.

```
CPU
                    Thread #0x00000804
FakAsmUnit.cpp.17: cout << "\nEingabe n zur Berechnung von n! =    (nNegatives n = Programm beenden)\n";
00401156 68BC204000        push 0x004020bc
0040115B FF3508514000      push dword ptr [CC3250.std::cout]
00401161 E894010000        call CC3250.std::basic_ostream<char, std::char_traits<char> >& std::operator <<(std::basic_ostream<char, std::char_traits<cha
00401166 83C408            add esp,0x08
FakAsmUnit.cpp.18: cin >> n;
00401169 8D45FC            lea eax,[ebp-0x04]
0040116C 50                push eax
0040116D FF3504514000      push dword ptr [CC3250.std::cin]
00401173 E888010000        call CC3250.std::basic_istream<char, std::char_traits<char> >::operator >>(int&)
00401178 83C408            add esp,0x08
FakAsmUnit.cpp.19: nFak=fak(n);
0040117B FF75FC            push dword ptr [ebp-0x04]
0040117E E85D000000        call fak(int)
00401183 59                pop ecx
00401184 8945F8            mov [ebp-0x08],eax
FakAsmUnit.cpp.20: cout << "n! = " << nFak << endl;
00401187 68C0114000        push 0x004011c0
0040118C FF75F8            push dword ptr [ebp-0x08]
0040118F 6804214000        push 0x00402104
00401194 FF3508514000      push dword ptr [CC3250.std::cout]
0040119A E85B010000        call CC3250.std::basic_ostream<char, std::char_traits<char> >& std::operator <<(std::basic_ostream<char, std::char_traits<cha
0040119F 83C408            add esp,0x08
004011A2 50                push eax
004011A3 E85E010000        call CC3250.std::basic_ostream<char, std::char_traits<char> >::operator <<(int)
004011A8 83C408            add esp,0x08
004011AB 50                push eax
004011AC E85B010000        call CC3250.std::basic_ostream<char, std::char_traits<char> >::operator <<(std::basic_ostream<char, std::char_traits<char> >&
004011B1 83C408            add esp,0x08
FakAsmUnit.cpp.21: } while (n >= 0);
004011B4 837DFC00          cmp dword ptr [ebp-0x04],0x00
004011B8 7D9C              jnl -0x64
FakAsmUnit.cpp.22: return 0;
004011BA 33C0              xor eax,eax
FakAsmUnit.cpp.23: }
004011BC 59                pop ecx
004011BD 59                pop ecx
004011BE 5D                pop ebp
004011BF C3                ret
std::endl<char, std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> > &):
004011C0 55                push ebp
004011C1 8BEC              mov ebp,esp
ostream.h.590:
004011C3 6A0A              push 0x0a
004011C5 FF7508            push dword ptr [ebp+0x08]
004011C8 E84B010000        call CC3250.std::basic_ostream<char, std::char_traits<char> >::put(char)
004011CD 83C408            add esp,0x08
ostream.h.591:
004011D0 FF7508            push dword ptr [ebp+0x08]
004011D3 E83A010000        call CC3250.std::basic_ostream<char, std::char_traits<char> >::flush()
004011D8 59                pop ecx
ostream.h.592:
004011D9 8B4508            mov eax,[ebp+0x08]
ostream.h.593:
004011DC 5D                pop ebp
004011DD C3                ret

00410000 00 00 00 00 00 00 00 00  ........
00410008 00 00 00 00 00 00 00 00  ........
00410010 00 00 00 00 00 00 00 00  ........
00410018 00 00 00 00 00 00 00 00  ........
00410020 00 00 00 00 00 00 00 00  ........
```

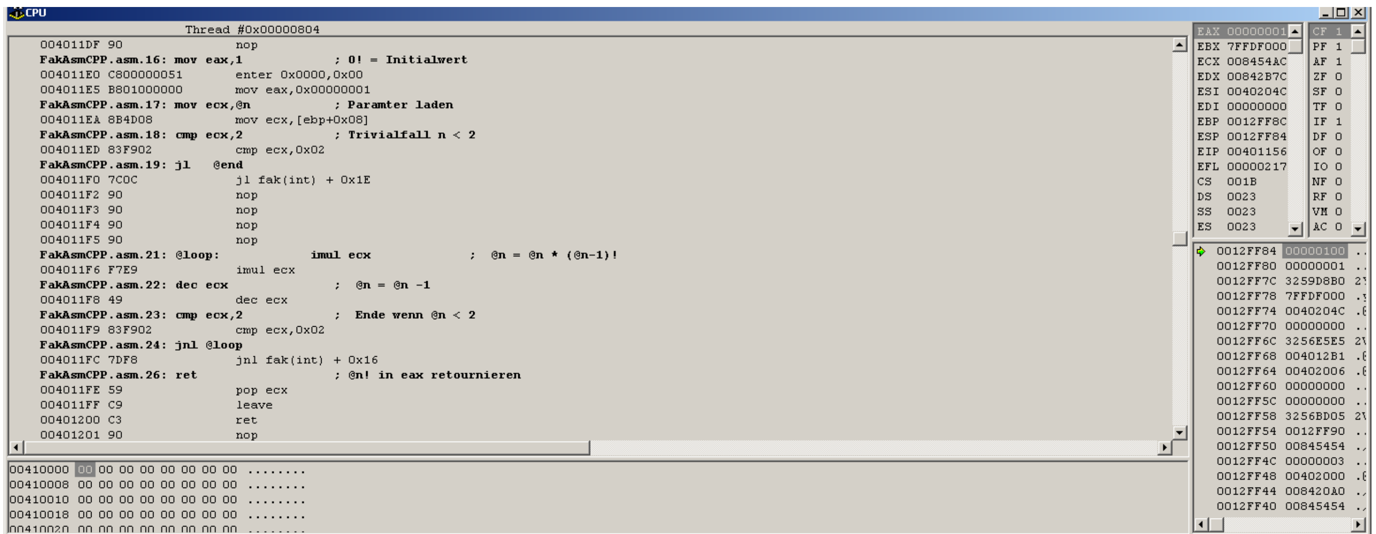


Bild 13: Debuggerfenster der CPU mit Darstellung des erzeugten Codes für die alternative Lösung nach Beispiel 6. Aus Platzgründen wurden die Funktionen main und fak in separaten Fenstern dargestellt.

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)

(Redigierte Version vom 18.1.2004)