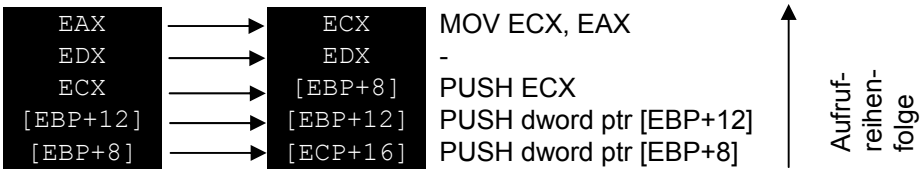
Delphi “register call” vs. C/C++ “fastcall”

Allgemeine Informationen

Die oben genannten Begriffe sind sogenannte Aufrufkonventionen. Darunter versteht man eine Art Konvention, wie die an Unterprogramme übergebenen Parameter im Speicher abgelegt werden müssen, damit das aufgerufene Unterprogramm sie auch versteht. Zu Zeiten von Windows und anderen 32 Bit Betriebssystemen hat die Interoperatibilität zwischen verschiedenen Programmiersprachen drastisch an Bedeutung gewonnen. Ein Beispiel - bei dem Aufrufkonventionen eine wichtige Rolle spielen - sind unter Windows die DLL-Dateien, welche überwiegend in C geschrieben werden. Delphi unterstützt zwar die „**cdecl**" Aufrufkonvention, da diese aber alle Parameter auf den Stack legt, ist diese extrem langsam (Stack-Operationen gehören zu den langsamsten, obwohl diese direkt unterstützt werden).
Um nun schnelleren Zugriff auf Bibliotheken zu erlauben, wurde unter C/C++ die "**fastcall**" Aufrufkonvention eingeführt. Diese legt so viele Parameter wie möglich in Register. Der Rest kommt auf den Stack. Das gleiche geschieht auch unter Delphi durch „**register call**" (wenn nichts angegeben, wird automatisch diese Konvention verwendet). Leider sind beide nicht kompatibel, da sich die Anzahl und die Reihenfolge der verwendeten Register wie folgt unterscheidet:

	Delphi register call	**C/C++ fastcall**
verwendete Register (1., 2., 3. Argument)	EAX, EDX, ECX	ECX, EDX
weitere Parameter auf den Stack **Merke:** der Stack wächst nach unten, d.h. jeder Aufruf von `PUSH` verringert `EBP` um die Länge des Arguments (hier 4 Byte) Die [ ] Klammern stehen für indirekte Adressierung, d.h. in `EBP` steht nicht der Wert, sondern die Adresse des Werts. Durch die Klammer holen wird uns dann den Wert.	von links nach rechts Arg. 4 [EBP+18] Arg. 5 [EBP+16] Arg. 6 [EBP+12] Arg. 7 [EBP+8] Rückspr. [EBP+4] ↓	von rechts nach links Arg. 6 [EBP+18] Arg. 5 [EBP+16] Arg. 4 [EBP+12] Arg. 3 [EBP+8] Rückspr. [EBP+4] ↓
	EAX Arg. 1 EDX Arg. 2 ECX Arg. 3 [EPB+20] Arg. 4 [EBP+16] Arg. 5 [EBP+12] Arg. 6 [EBP+8] Arg. 7 [EBP+4] Rückspr.A.	ACX Arg. 1 EDX Arg. 2 [EBP+8] Arg. 3 [EPB+12] Arg. 4 [EBP+16] Arg. 5 [EBP+20] Arg. 6 [EBP+24] Arg. 7 [EBP+4] Rückspr.A.

Aber es gibt einen Ausweg: Man simuliert „**fastcall**“. Dieser kostet zwar zusätzlich Rechenzeit, in Kombination mit „register“- und „fastcalls“ ist dieser aber wesentlich schneller als reine „**cdecl**" Aufrufe:



Beispiel (Genesis 3D for Delphi):

```
#define GENESISCC _fastcall

void GENESISCC
geEngine_RenderPolyArray(const
geEngine *Engine, const
GE_TLVertex ** pPoints, int *
pNumPoints, int NumPolys, const
geBitmap *Texture, uint32
Flags);
```

```
procedure geEngine_RenderPolyArray(const
Engine: PgeEngine; const pPoints:
PPGE_TLVertex; pNumPoints: PInteger;
NumPolys: Integer; const Texture:
PgeBitmap; Flags: UInt32);
asm
      PUSH dword ptr [ebp+8];
      PUSH dword ptr [ebp+12];
      PUSH dword ptr [ebp+16];
      PUSH ECX;
      MOV ECX, EAX;
      CALL _geEngine_RenderPolyArray;
end;
```

In diesem Beispiel werden die Register wie folgt belegt:

	Delphi	Änderung für fastcall Kompatibilität	C/C++
* Engine	EAX	MOV ECX, EAX	ECX
** pPoints	EDX	- keine Änderung nötigt -	EDX
* pNumPoints	ECX	PUSH dword ptr [ECX]	[EBP+8]
NumPolys	[EBP+16]	PUSH dword ptr [EBP+16]	[EBP+12]
* Texture	[EBP+12]	PUSH dword ptr [EBP+12]	[EBP+16]
Flags	[EBP+8]	PUSH dword ptr [EBP+8]	[EBP+20]

Aufruf des Unterprogramms

Fehlt nur noch der Aufruf des Unterprogramms. Dieses kann aufgrund der Inkompatibilität zwischen Delphi und C/C++ „**fastcall**“ nur manuell geschehen. Ein einfaches

```
procedure geEngine_RenderPolyArray(const Engine: PgeEngine; const pPoints:
PPGE_TLVertex; pNumPoints: PInteger; NumPolys: Integer; const Texture:
PgeBitmap; Flags: UInt32); external <DLLName>;
```

hilft leider nicht, da die o. g. Konvertierung geschehen muss, wenn die DLL-Datei „fastcalls“ beinhaltet. D.h. wir müssen uns die Adresse des Unterprogramms selbst holen. Dazu definieren wir einen Zeiger, der später die Adresse erhalten wird:

```
var _geEngine_RenderPolyArray: pointer;
```

Dieser Pointer muss nur noch mit der Adresse belegt werden. Dazu öffnen wir zuerst die DLL-Datei und holen uns die Adresse. Dann weisen wir diese dem Zeiger zu:

```
DLLHandle: HMODULE;
…
DLLHandle := LoadLibrary(DLLName); // DLL in den Speicher laden
…
_geEngine_RenderPolyArray := GetProcAddress(DLLHandle,
'@geEngine_RenderPolyArray@24');
…
FreeLibrary(DLLHandle); // aber erst, wenn DLL nicht mehr benötigt!
```

Die Namen der Funktionen werden in der Form „@<Funktionsname>@<Größe der Parameter in Byte>“ in der DLL-Datei abgelegt, d.h. die Parameter die an die o. g. Funktion übergeben werden haben insgesamt 24 Byte Länge. Da überwiegend Variablen/Werte mit 4 Byte Länge verwendet werden (meistens Zeiger) benötigt die o. g. Funktion 6 Parameter.

Keine Regel ohne Ausnahme

Leider ist das auch bei den Konventionen der Fall. Trotzdem wird uns das keine Kopfschmerzen bereiten, denn die Ausnahme kann man ganz einfach verallgemeinern. Die Rede ist hier von Fließkommazahlen (Single, Double, …). Diese werden - entgegen der o. g. Regel - nicht in Register, sondern **immer** auf den Stack abgelegt. Dies geschieht bei beiden Programmiersprachen, aber - wie schon erwähnt - in unterschiedlicher Reihenfolge.

Beispiel:

```
function Test2 (a, b, c: Single): Single;
begin
  Result := a + b + c;
end;
…
Test2(1.0, 2.0, 3.0);
```

Man würde jetzt annehmen, dass jetzt `EAX = 1.0, EDX = 2.0, ECD = 3.0` ist. Diese Annahme ist falsch, da wie oben schon erwähnt, die Fließkommazahlen **immer** auf den Stack kommen:

Delphi	
[EBP+16]	1.0
[EBP+12]	2.0
[EBP+8]	3.0

C/C++	
[EBP+8]	1.0
[EBP+12]	2.0
[EBP+16]	3.0

Auch hier gilt die Regel **von links nach rechts für Delphi** und von **rechts nach links für C/C++**. Es muss also die gleiche Änderung geschehen, wie oben bereits erwähnt, d. h. vor dem Aufruf des Unterprogramms die Werte auf den Stack in umgekehrter Reihenfolge ablegen.

Sollten sich in der Parameterliste auch ganze Zahlen (Integer, Zeiger(Adresse), usw.) befinden, werden diese natürlich in Register abgelegt.

Beispiel:

```
function Test4 (a: Single; b: Integer): Single;
begin
  Result := a + b;
end;
…
writeln (Test4 (1.0, 5));
```

Ergebnis:

Delphi	
EAX	5
[EBP+8]	1.0

C/C++	
ECX	5
[EBP+8]	1.0

D. h. Fließkommazahlen kommen **immer** auf den Stack. Ganze Zahlen werden in Register geschrieben. Falls keines mehr frei ist, kommen diese auch auf den Stack. Die Reihenfolge, wie dann ganze Zahlen und Fließkommazahlen auf dem Stack liegen, hängt wiederum von der verwendeten Programmiersprache und der Reihenfolge der Parameter ab.

Sonderformen (Fließkommazahlen mit doppelter Genauigkeit und ganze Zahlen mit 8 und 16 Bit)

Bei Fließkommazahlen mit doppelter Genauigkeit passiert das gleiche. Nur werden - da nicht 4 Byte sondern 8 Byte groß - für einen Parameter zwei „PUSH“ benötigt. Da diese Genauigkeit selten benötigt wird bzw. wieder Kompatibilitätsprobleme mit C/C++ auftreten können, gehe ich nicht genauer darauf ein. Falls es jemanden doch interessieren sollte, kann mir derjenige eine E-Mail schreiben.

Falls als Parameter ganze Zahlen vorkommen, die für die 32 Bit Register zu klein sind, kommen die „niederbitigen“ Register zum Zuge. Auch hier gelten die bereits oben erwähnten Konventionen. Hier wird EBP nur um 2 bzw. 1 erhöht, die Rücksprungadresse (32 Bit) liegt wie aber immer bei [EBP+4]:

	Delphi	C/C++
16 Bit	AX, DX, CX [EBP+10] [EBP+8]	CX, DX [EBP+8] [EBP+10]
8 Bit	AL, DL, CL [EBP+9] [EBP+8]	CL, DL [EBP+8] [EBP+9]

Testroutinen

Hier die Testroutinen, die ich zum Testen verwendet habe. Zum ausprobieren, einfach einen Breakpoint auf die entsprechende „writeln“ Routine setzen und nachdem Delphi gestoppt hat, mit Strg+Alt+C das CPU-Fenster aufrufen.

```
program Project1;

{$APPTYPE CONSOLE}

uses
  SysUtils;

function Test (a, b, c, d, e, f, g: Integer): Integer;
begin
  Result := a + b + c + d + e;
end;

function Test2 (a, b, c: Single): Single;
begin
  Result := a + b + c;
end;
```

```
function Test3 (a, b: Double): Double;
begin
  Result := a + b;
end;

function Test4 (a: Single; b: Integer): Single;
begin
  Result := a + b;
end;

function Test5 (a, b, c: Single; d, e, f, g, h: Integer): Single;
begin
  Result := a + b + c + d + e + f + g + h;
end;

function Test6 ( d, e, f, g, h: Integer; a, b, c: Single): Single;
begin
  Result := a + b + c + d + e + f + g + h;
end;

begin
  writeln (Test ($100, $200, $300, $400, $500, $600, $700));
  writeln (Test2 (1.0,2.0,3.0));
  writeln (Test3 (0.000000001, 100000000.0));
  writeln (Test4 (1.0, 5));
  writeln (Test5 (1.0, 2.0, 3.0, 4, 5, 6, 7, 8));
  writeln (Test6 (4, 5, 6, 7, 8, 1.0, 2.0, 3.0));
  readln;
end.
```

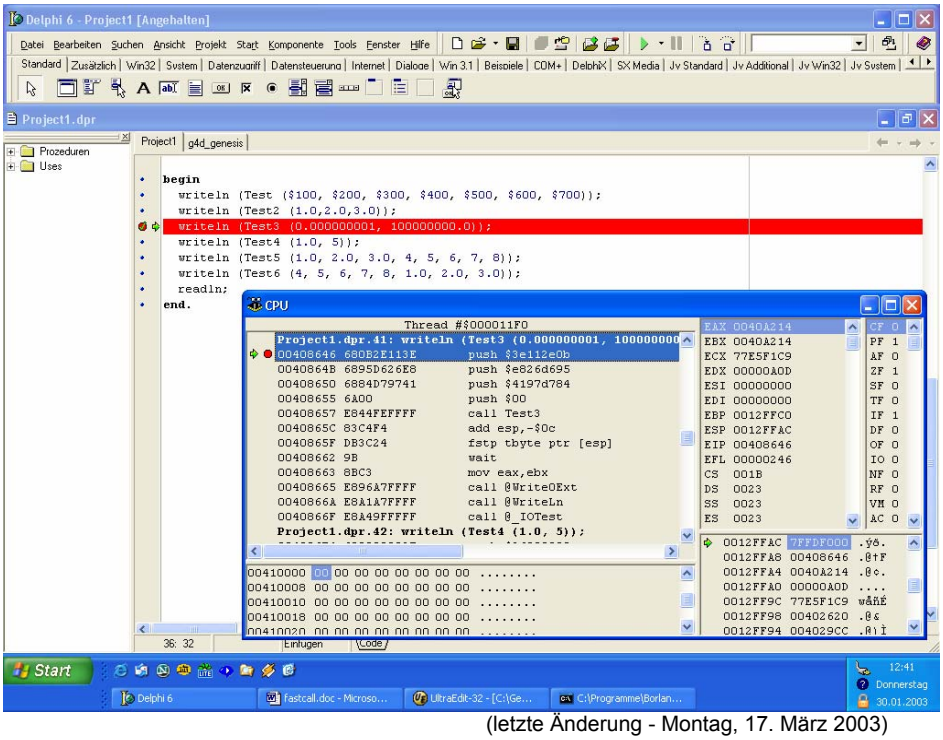

Danksagung

Die meisten Informationen basieren auf eigenen Tests mit Delphi 6. Deshalb möchte ich Borland für Delphi 6 und dem ausgezeichnetem Debugger danken.
Außerdem möchte ich Niels Vanspauwen für seine Pionierarbeit in Sachen „fastcall“ unter Delphi und seiner „Genesis For Delphi“ Unit-Sammlung danken, die mir den Ansporn gegeben hat, mein Wissen zu vertiefen.

Kontaktinformationen

Ich hoffe, ich konnte ein bisschen Wissen vermitteln. Berichtigungen und Korrekturen bitte an wummy@gmx.net schicken. Weitere Informationen gibt es auf meiner Homepage unter www.wummy-online.de.

-W



Das CPU-Fenster von Delphi 6 hat mir bei meinen Recherchen ungemein geholfen.
Kleiner Tipp am Rande: Man kann im Stack-Fenster (rechts unten) durch einen Mausklick rechts einstellen, wie der Inhalt dargestellt werden soll. Hilfreich für Fließkommazahlen auf dem Stack (geht auch im Fenster für das Datensegment links unten).