_stdcall threaded Forth

Dr.-Ing. Egmont Woitzel
Universität Rostock,
Fachbereich Elektrotechnik
Richard-Wagner-Str. 31
18119 Warnemünde
Tel. (0381) 498 3538
email: woi@baltic.e-technik.uni-rostock.de

Aufgrund seines interaktiven und inkrementellen Konzepts findet Forth nur relativ schwierig Anschluß zu Produkten anderer Programmiersystemen, die von einem klassischen Edit-Compile-Link-Zyklus ausgehen. Eine erste Schwierigkeit besteht schon darin, eine in Forth programmierte Anwendung in eine ausführbare Datei zu verwandeln, die vom Betriebssystem gleichberechtigt zu den Ergebnissen der »mainstream«-Produkte benutzt werden kann. Es stellt sich daher die Frage, wie ein Forth-System aufgebaut sein müßte, das alle Spielarten ausführbarer Dateien als Turnkey-Anwendungen generieren kann. Dieser Beitrag stellt eine Lösung dieses Problems für die heute wichtigste Anwendungsplattform Win32 vor.

Programmieren unter Win32

Vergleicht man Win16 und Win32 miteinander, so fällt vor allem die unterschiedliche Verwaltung der Anwendungsprozesse auf. Während sich im Win16 alle Anwendungsprozesse einen gemeinsamen Speicherraum teilten, so erhält unter Win32 jeder Prozeß einen eigenen virtuellen Speicherraum zugeteilt. Dies hat mehrere wesentliche Konsequenzen:

- Prozesse können ohne besondere Vorkehrungen (*shared memory*) nicht mehr auf denselben physikalischen Speicher zugreifen. Der Austausch von Zeigern zwischen Anwendungen ist damit sinnlos geworden.
- Da jedem Prozeß einen (virtuelle) 32-Bit Speicherraum allein zur Verfügung gestellt wird, besteht keine Notwendigkeit mehr für die Verwendung der Segmentierung. Konsequenterweise werden daher alle Segmentregister der Intel-CPU vom Betriebssystem fest eingestellt und dürfen von der Applikation nicht mehr verändert werden. Die Segmentregister können deshalb nicht mehr für eine Basisrelokation einzelner Segmente benutzt werden.

Eine unangenehme Folge ist, daß im allgemeinen Fall zum Zeitpunkt der Übersetzung eines Programms nicht die Adresse vorhergesagt werden kann, auf die es zur Ausführung geladen wird. Die erforderliche Relokation wird normalerweise durch den Applikationslader des Win32-Subsystems ausgeführt.

Innerhalb eines Prozesses können mehrere Threads gestartet werden, die sich alle in demselben Speicheraum befinden. Das Betriebssystem weist weniger Prozessen als vielmehr Threads Rechenzeit zu. Die Umschaltung erfolgt »preemptiv«, so daß keine besonderen Vorkehrungen getroffen werden müssen, um das System reaktionsfähig zu halten. Im

Gegensatz zu Win16 kann man also in einem Thread gefahrlos eine Polling-Schleife ohne »PAUSE« laufen lassen.

Interoperabilität

Betrachtet man die Konstruktion moderner Softwaresysteme, so stellt man fest, daß komplexe Anwendungen nicht mehr als monolithische Giganten hergestellt werden, sondern aus einer großen Anzahl miteinander kommunizierender Komponenten konstruiert werden. Die einzelnen Komponenten werden dabei durchaus auf Basis völlig unterschiedlicher Programmiersysteme implementiert. Möglich wird dies in erster Linie durch die Integration geeigneter Kommunikationsverfahren in das API des Betriebssystems. Möchte man Forth in einer Win32-Umgebung sinnvoll einsetzen, muß man daher unter allen Umständen die Kooperation zwischen Forth-basierenden Programmen und beliebigen anderen mit den Mitteln des Betriebssystems unterstützen.

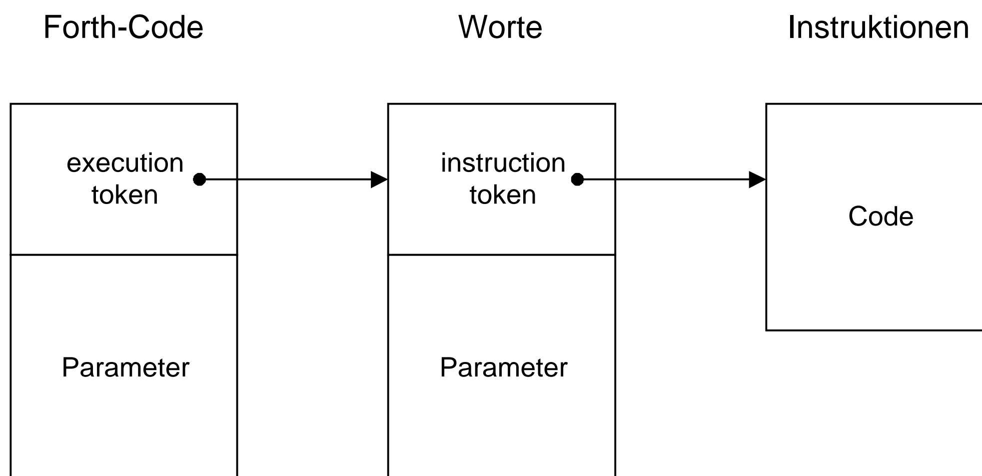
Das elementare Verfahren zur Integration mehrerer Komponenten zu einer Anwendung stellt unter Windows die Benutzung Dynamischer Bibliotheken (*Dynamic Link Library* bzw. DLL) dar. Das Prinzip ist simpel. Eine DLL ist im Prinzip wie eine ausführbare Datei aufgebaut. Allerdings enthält eine DLL kein Programm, das als eigener Prozeß gestartet wird. Statt dessen »exportiert« eine DLL Funktionen. Diese können von einem Anwendungsprogramm oder einer anderen DLL aus »importiert« und benutzt werden. Eine DLL-Funktion läuft immer im Kontext des aufrufenden Prozesses. Das Verbinden von Importen und Exporten organisiert der Lader des Betriebssystems. Jede ausführbare Datei enthält eine Liste der zum Start des Programms benötigten Funktionen aus anderen ausführbaren Dateien sowie eine Liste der eigenen Exporte. Vor dem Programmstart werden in einem rekursiven Verfahren alle benötigten DLLs in den Speicher geladen. Nachdem die Adressen der exportierten Funktionen feststehen, können die aufrufenden Programme aktualisiert werden.

Allerdings wird in einem aufrufenden Programm je Import nur ein einziger Adreßzeiger in der sogenannten *import address table* verändert. Die eigentlichen Aufrufe erfolgen dann über indirekte CALL's. Programmiert man unter C, benutzt man dafür üblicherweise die sogenannten Import-Libraries. In diesen ist für jede importierte Funktion eine Wrapperfunktion enthalten, die diesen indirekten CALL verdeckt.

DLL's bilden die Grundlage aller Kommunikation zwischen verschiedenen Programmkomponenten. Auch das Betriebssystem stellt seine Funktionen den Anwendungsprogrammen in Form von DLL's zur Verfügung. Auch alle höhere Formen der Kommunikation werden letzten Endes durch exportierte Funktionen von DLL's realisiert. Die wesentliche Beschränkung von DLL's besteht in ihrer Zuordnung zum Kontext eines Anwendungsprozesses. Man kann allein mit Hilfe von DLL's nicht zwischen zwei Prozessen kommunizieren. Das Betriebssystem liefert für diesen Zweck jedoch eine umfangreiche Palette verschiedener Verfahren, begonnen bei der Verwendung gemeinsamen Speichers (*shared memory*) und Briefkästen (*mail slots*) bis zum entfernten Prozeduraufruf (*remote procedure call*). Es zeichnet sich jedoch ab, daß diese Vielzahl von Verfahren im Bereich der Applikationsprogrammierung durch das allgemeine Komponentenmodell COM (*component object model*) bzw. DCOM (*distributed COM*) und die darauf aufsetzenden Interfacespezifikationen von OLE verdrängt werden.

Virtuelle _stdcall-Maschine

Ein erfolgversprechender Ansatz zur Implementation eines »kommunikativen« Forth-Systems besteht in der direkten Verwendung der Aufrufkonventionen für DLL- und API-Funktionen innerhalb der virtuellen Forth-Maschine. Hinsichtlich des unten behandelten Problems der Relozierbarkeit kommt als grundlegendes Code-Modell am ehesten das indirekter Modell in Frage. Indirekter Fadencode arbeitet nach folgendem Schema:



Indirekt gefädelter Forth-Code besteht aus einer Folge von Zeigern auf Worte. Die Wortadresse wird üblicherweise als *execution token* verwendet. Innerhalb der Zeigerfolge können sich auch Parameter befinden, die bei Ausführung des Codes des davor liegenden *execution token* IP(*instruction pointer*)-relativ adressiert werden. Worte beginnen immer mit einem Zeiger auf ausführbaren Code für eine Instruktion der virtuellen Maschine. In Analogie zum *execution token* wird der Zeiger auf den Code als *instruction token* bezeichnet. Die im Wortrumpf liegenden Parameter werden WA(*word address*)-relativ adressiert.

Wenn man in diesem Schema die Instruktionen so entwirft, daß sie die für API- und DLL-Funktionsrufe verwendete `_stdcall`-Aufrufkonvention einhalten, so ergibt sich eine natürliche Einbettung dieser Funktionen als Instruktionen der Forth-Maschine ohne Notwendigkeit für ihre »Ummantelung« mit einer Anpassungsschicht. Mit diesem Ansatz kommt man dem Ziel einer einfachen Kooperation mit anderen Programmen schon näher: Ihre Funktionen sind einfach als Instruktionen der Forth-Maschine importierbar, also wie ein mit Hilfe von `CODE` definiertes Wort aufrufbar.

Aufrufkonvention _stdcall

Aufrufkonventionen definieren die Beziehung zwischen einem aufgerufenen Unterprogramm und seinem Aufrufer. Dazu gehören die Art der Parameterübergabe zwischen ihnen, die Verwendung der Register sowie Vereinbarungen über die Symbolnamen. Im Fall der `_stdcall`-Konvention sind dies folgende Festlegungen:

- der Aufrufer legt die Parameter von rechts nach links auf den Stapel, das bedeutet, daß der in der C-Parameterklammer am weitesten rechts stehende Wert auf die Stapelspitze gelangt. Dann erfolgt ein `CALL` zur Funktion.

- Die aufgerufene Routine gibt dem Aufrufer die Register `ESI`, `EDI`, `EBX` und `EBP` unverändert zurück. Alle übrigen Register dürfen verändert werden. Das *direction flag* muß bei Rückkehr auf den Modus »inkrementieren« eingestellt sein.
- Die aufgerufene Routine entfernt die übergebenen Parameter vom Stapel und liefert den Rückgabewert im Register `EAX` ab.
- Der Linker findet die Anfangsadresse der Routine über ein Symbol mit dem Namen `_function@xx`, wobei `function` als Platzhalter für den Funktionsnamen und xx als Platzhalter für die Anzahl übergebener Byte auf dem Stapel steht.

Besonders wichtig ist hierbei, daß insgesamt vier Register durch die aufgerufene Routine nicht verändert werden. Dadurch können diese als Register der virtuellen Forth-Maschine verwendet werden. Es bietet sich folgende Zuordnung an:

i8086	Forth	Bezeichnung
`ESI`	`IP`	*instruction pointer*
`ESP`	`DSP`	*data stack pointer*
`EBX`	`RSP`	*return stack pointer*
`EAX`	`WA`	*word address register*

Das Register `WA` wird nur temporär benötigt und muß daher nicht auf eines der geretteten Register gelegt werden. Das Register `EDI` wird zur Lösung des unten ausführlich diskutierten Relokationsproblems verwendet.

Export von Forth-Funktionen

Sollen Forth-Programme nicht nur Funktionen anderer Anwendungen importieren können, sondern auch diesen in Forth implementierte Funktionen zur Verfügung stellen können, so wird ein weiterer Mechanismus benötigt. Die Abarbeitung von Forth-Code muß in einer Funktion »verpackt« werden, die den `_stdcall`-Konventionen genügt. Dieses Problem kann relativ einfach gelöst werden. Im weitesten Sinne können dazu dieselben Verfahren verwendet werden, die zum Aufruf von in Forth implementierten Interruptservice-Routinen benutzt werden. Der einzige nennenswerte Unterschied besteht darin, daß an Interruptserviceroutinen normalerweise keine Parameter übergeben werden. Diese müssen im Fall exportierbarer Funktionen zusätzlich umkopiert werden.

Relokation

Das schwierigste Problem beim der Konstruktion eines Win32-Forth-Systems ist die Sicherung der Relozierbarkeit des Forth-Codes. Die Notwendigkeit dieser Eigenschaft wird besonders bei der Betrachtung von DLL's offensichtlich. Während man beim Start einer Anwendung im allgemeinen davon ausgehen kann, daß dieses auf seine bevorzugte Ladeadresse geladen wird, entscheidet im Falle von DLL's der Win32-Lader über die Reihenfolge der Speichervergabe, werden mehrerer DLL's verwendet, wird dies sehr schnell unübersichtlich. Zudem teilen Windows NT und Windows 95 den Speicher unterschiedliche ein.

Ein nächstes Problem entsteht zusammen mit dem Fehlen einer Typisierung durch die interaktive und inkrementelle Arbeitsweise von Forth. Applikationen werden üblicherweise auf ein interaktives System geladen. Anschließend wird ein Speicherabzug gesichert, der die Applikation zusammen mit dem Systemkern enthält. Es ist weder nachträglich noch während der Übersetzung von Forth-Quelltext feststellbar, an welchen Positionen dieses Speicherabzugs sich zu relozierende Adreßinformationen befinden.

Der einfachste und vermutlich auch effizienteste Ausweg aus diesem Dilemma besteht darin, innerhalb des Forth-Codes mit relativen Adressen zu rechnen. Um die Kosten der Relokation zu minimieren, kann die tatsächliche Ladeadresse im Register `EDI` gespeichert werden. Alle auf den Speicher zugreifenden Instruktionen verwenden dessen Inhalt als zusätzlichen Offset. Eine explizite Transformation ist nur beim Empfang von Adressen durch andere Programme oder API-Funktionen bzw. bei der Übergabe von Adressen an externe Funktionen notwendig.

Von diesem Ansatz wird auch das Fädelungs-Modell berührt. Die innerhalb des Forth-Codes gespeicherten *execution token* dürfen nur relativ adressiert werden. Noch komplizierter wird es im Fall der *instruction token*. Setzt man voraus, daß Instruktionen in DLL's gespeichert werden, so kann man nicht von einer einfachen Verschiebung ausgehen. Allerdings besitzen Worte und *instruction token* eine besonders exponierte Stellung innerhalb eines Forth-Systems. Sowohl der *instruction token* als auch die Position des Worts lassen sich während der Übersetzung von Forth-Quelltext exakt und relativ einfach verfolgen. Wird diese Information in Form einer Tabelle mitgeführt, welche die Position eines Worts mit der von dem Wort verwendeten Instruktion assoziiert, kann beim Start des Forth-Systems eine vergleichsweise einfache Routine die *instruction token* bestimmen und in die Worte eintragen. Dadurch gewinnt man nicht nur eine absolute Adresse als *instruction token*, sondern vermeidet auch jeden Wrapper-Code für den indirekten Aufruf der importierten Funktion über die *import address table*.

Adreßinterpreter

Nachdem die Struktur des Forth-Codes feststeht, läßt sich eine Implementierung für den Adreßinterpreter angeben:

```
push1:
     push eax                      ;Resultat wird ToS

next:
     mov  eax, dword ptr [esi]     ;lese xt
     add  esi, 4                   ;nächste Instruktion
     push offset push1             ;Rückkehr vorbereiten
     jmp  dword ptr [edi+eax]      ;springe zum it
```

Auf den ersten Blick erscheint diese Vorgehensweise zu extrem langsamen Instruktionen der Forth-Maschine zu führen. Obwohl kein register mapping der Stapelspitze auf ein Register möglich ist und die Rückkehradresse auf dem Stapel vergleichsweise viel »Bewegung« verursacht, erweist sich die Implementierung von Forth-Instruktionen nach diesem Modell doch als relative einfach, wie die folgenden Quelltextausschnitte belegen dürften:

```
DUP proc ;ds: x -- x x
     mov  eax, [esp+4]              ;ToS als Rückgabewert
     ret
DUP endp

DROP proc ;ds: x --
     pop  edx                       ;Adresse von push1
     inc  edx                       ;Adresse von next
     add  esp, 4                    ;ToS vernichten
     jmp  edx                       ;Sprung zu next
DROP endp
```

Die dynamische Verwaltung der »Adreßinterpreter-Adresse« besitzt einen weiteren Vorteil. Man kann sehr einfach den »*release*«-Adreßinterpreter durch einen »*debug*«-Adreßinterpreter ersetzen, der zum Beispiel Einzelschrittbetrieb organisiert. Allerdings kann die Systemleistung durch einen dezentralen Ansatz um ca. 20...30% gesteigert werden. Forth-Primitive hängen zwecks Einsparung eines Sprunges einfach die Instruktionen des Adreßinterpreters an die eigenen Instruktionen. Dabei sind häufig weitere Einsparungen erzielbar, insbesondere bei IP-adressierenden Forth-Instruktionen. Der folgende Quelltext zeigt die Implementation der zu `(LITERAL)` gehörenden Instruktion:

```
oparLITERALcpar proc
     pop  edx                       ;Adresse push1
     push [esi]                     ;Literal -> Tos
     push edx                       ;Rückehr vorbereiten
     mov  eax, dword ptr [esi+4]    ;xt lesen
     add  esi, 8                    ;ip weitersetzen
     jmp  dword ptr [edi+eax]       ;Sprung zum it
oparLITERALcpar endp
```

`DOES>`-definierte Instruktionen

Eine letzte Tücke hält die Implementation von `DOES>` bereit. Möchte man die mit Forth-83 aufgekommene elegante Implementation beibehalten, wird zusammen mit jedem `DOES>`-Part eine Forth-Instruktion definiert. Diese besteht normalerweise nur aus einem Call auf den gemeinsam verwendeten Einsprungcode. Diese Instruktionen entstehen dynamisch während der Übersetzung und können nicht einer DLL zugeordnet werden. Als sinnvoller Ausweg hat sich erwiesen, diese Instruktionen analog zu den Callbacks zu verwalten und den beim Startup-Vorgang zwischen DLL-Importen und Does-Eintrittspunkten zu unterscheiden.

Systemarchitektur

Ein dem oben vorgestellten Entwurf folgendes Forth-System besteht damit aus den in der folgenden Tabelle zusammengestellten Bestandteilen, die gleichzeitig Sektionen der ausführbaren Datei darstellen:

Sektion	Inhalt

`.image`	Forth-Code
`.text`	Startup-Code, Callback-Code, Does-Code
`.idata`	Windows-Import-Sektion
`.reloc`	Windwos-Relokationen in .text und .image
`.words`	Tabelle der Worte und zugehörigen Instruktionen
`.cback`	Callback-Eintritte
`.does`	Does-Eintritte

Programmierumgebung

Zur Herstellung der Programmdatei wird das Programmiersystem fieldFORTH zusammen mit zwei speziellen Linkern eingesetzt. Die auf der nächsten Seite folgende Abbildung zeigt den Entwicklungsfluß. Während fieldFORTH zur Übersetzung des Forth-Codes verwendet wird, werden die Instruktionen mit Hilfe von VC++ und MASM in DLL's übersetzt. Der Linker FFLINK verbindet den Forth-Code mit den Instruktionen und produziert eine Objektdatei, die in ihrem Aufbau einem dem Ergebnis einer SAVE-Operation gleicht. Mit Hilfe des Linkers CFLINK wird das gesicherte Objekt mit dem Startupcode verbunden und kann zusätzlich in seinen Speichereinstellungen verändert werden.

Die Ergebnisse der »Fingerübungen« waren bisher zufriedenstellend. Die beschriebene Technologie kann für Produktentwicklungen unter Win32 eingesetzt werden.

Win32-Entwicklung mit fieldFORTH