STDCALL Threaded Code and its Impact on Debugging

Egmont Woitzel, Stephan Lange***
** University of Rostock, Department of Electrical Engineering,*
Institute of Applied Microelectronics and Computer Science
Richard-Wagner-Straße 31, D-18119 Rostock
*** FORTecH Software Entwicklungsbüro,*
Joachim-Jungius-Straße 9, D-18059 Rostock

Abstract. Component oriented software construction demands development tools which produce code, which can import and export functions from and to other components by means of the underlaying operating system. This paper focuses on an implementation technique for the Win32 API, which enables Forth systems to import functions, exposed by the Windows nucleus or any other software component, as ordinary Forth primitive words. It will be shown, that the resulting architecture of the virtual Forth machine offers excellent debugging capabilities.

Win32, Component Techniques, and Forth

Recent software products most often consist of a large number of different binary files, executable programs with a graphical user interface, dynamic link libraries, hidden servers, and so on. Even though it is sometimes a little bit difficult to keep track on a large number of files, the support of the underlaying operating system to connect all necessary files at load- or run-time enables the construction of maintainable software systems. One of the most important effects of this kind of operation system support is the freedom to implement each piece of software in the best fitting language.

As an illustration of that approach: We had to deliver an Win16 interface library destined for applications written in C/C++ and Visual Basic for a Forth driven network at the Accelerator Laboratory of the Munich Universities. The central component of that system is a Forth-written DLL with C-conform interface functions, which sends the network commands and interprets the response messages. Whereas C/C++-written front-ends directly use that DLL, Visual Basic written front-ends use that DLL through an additional interface translation DLL written in C, which simplifies the Basic programs. On the opposite end, the user mode part of the network driver is written in C, the kernel mode part is written in assembly language. Cross-language development is the challenge of the day.

In the context of component-oriented software systems the ability of a programming system to generate executable files which can co-operate at operating system level becomes an essential feature for its survival.

STDCALL Register Mapping

The idea behind STDCALL threading [WOI97] is to understand Forth as an integration tool. High-level Forth programs are managing the execution of primitives. From the view of a virtual machine model the set of these primitives is identical to the set of its instructions. The technique used for coding and executing sequences of primitives may be understood as an integration technique. Therefore, all primitives, or instructions, have to satisfy the same rules for parameter passing and keeping control flow information, defined by the integration technique used.

If we switch to the world of Win32 executable files, we will find that the STDCALL calling convention acts as an overall rule for parameter passing and keeping control flow information. If we want to design an integration technique for Win32-»instructions«, we have to choose the STDCALL calling convention. The remaining question is, whether all the instructions defined by Forth systems may be mapped to the STDCALL convention. The answer is clearly »no«, because of the ability of Forth instructions to generate more then one result and the need for two stacks. But there is the opportunity to design a more specific calling convention, which includes the STDCALL convention as a subset. Let's take a closer look on STDCALL for Intel processors, shown at Table 1, and an applicable Forth register mapping.

Caller	Callee
pushes parameters onto the stack from left to right	removes all parameters from stack
provides the return address on top of the stack	returns to the address found on top of the stack
expects a result in EAX	delivers a scalar 32-bit result in EAX, an 8-byte result structure in EDX:EAX or a pointer to other-sized result structure in EAX
may use EBX, EBP, ESI and EDI to store information beyond the scope of a function call	saves and restores register contents of EBX, EBP, ESI, EDI
the direction flag has to be resetted to »incrementing« after a »decrementing« code section before the next call	the direction flag is expected to be set »incrementing« and has to be saved

Table 1: Intel 32-bit STDCALL calling convention [MIC92]

Forth primitives addressing parameters on the return stack (RSP relative addressing mode), and/or at the parameter field of the recently executed word (WA relative addressing mode) and/or inline parameters (IP relative addressing mode) need additional information. Because all »ordinary« or operating system primitives will preserve them, the registers EBX and ESI may be utilized for the return stack pointer RSP and the instruction pointer IP of the virtual Forth machine. The value of the word address register WA does not need to be preserved, so the EAX register may be used to transfer this value to a WA-addressing Forth instruction.

An additional problem will be caused by the way of returning values. Without the knowledge of the high-level language specification of a STDCALL function, it is impossible to determine the kind of return value, especially whether there is any or not. Because the Win32-API does not contain any function which returns a structure, and only a very small number of functions without any return value, it is only a little drawback if the virtual Forth machine lets every ordinary primitive

return a 32-bit value. However, this behavior is not acceptable for Forth primitives. Popping the top of stack into the EAX register causes non-acceptable run-time costs. A better solution is to take into consideration the calling instructions. If the callee knows the intructions to return to, it is able to control its behavior by modifying the return address.

The EDI and EBP registers may be utilized to store additional information. Unfortunately, there is an urgent need to apply a relocation on addresses inside the Forth system, wich will be discussed below. Therefore, one of these registers must hold the absolute start address of the Forth dictionary. EDI is the best choice. The EBP register may be used most effectively for fast access to local variables or instance data of objects in an object oriented extension.

STDCALL Forth Code Layout

The main reason for using relative addressing inside the Forth dictionary is the impossibility to decide whether a given memory cell does or does not contain an address, due to the typeless nature of the Forth data stack. On the other hand, in the flat memory address model of Win32 it is not practicable to guarantee, that a saved dictionary image will be mapped to the same memory address when it will be reloaded. Even though the executable file format allows to specify a required load address, it is not practicable for creating dynamic link libraries. When creating two different DLLs starting with the same dictionary image, both DLLs will obtain the same required load address. So, it would become impossible to use them simultaneously inside a single process.

The situation becomes more complicated if we take into account that the machine code of an instruction of the virtual Forth machine may reside in a separate DLL. In fact, that is true for all Win32 API functions. Fortunately, the affected addresses are well-known and simply trackable. The sole part depending on the code address of a Forth word is its code field, identified by the execution token of the word. The creation and modification of a new execution token is typically encapsulated in only few Forth words, including :NONAME, CREATE, DOES> and ;CODE. Because there is no standard compatible method to create execution tokens without these words, it is possible to create relocation information for code addresses. This observation completes the information needed to implement an STDCALL threaded address interpreter. Using an indirect threading scheme, words will be addressed relative to the image base address, whereas the code fields will contain absolute addresses to the corresponding machine code routines. Combined with the register mapping given above, the address interpreter may be implemented as shown in Picture 1.

The instruction pointer ESI points absolutely to the Forth code. Due to the return instruction which finalizes the pseudo-called machine code routines, the address interpreter loops infinitely. The run-time overhead of this solution is much smaller than expected. In fact, this STDCALL threading implementation runs nearly as fast as usual indirect threaded code implementations. The implementation of some typical Forth instructions given by Picture 2 should illustrate the coding practice.

```
    ...
push1:                                 ;entry with one return value
    push    eax                        ;save the result
next:                                  ;entry without any result
    mov     eax, dword ptr [esi]       ;read next word address
    add     esi, 4                     ;step to next word in Forth code
    push    offset push1               ;provide return address
    jmp     dword ptr [edi+eax]        ;jump to the machine code
```

Picture 1: STDCALL threaded address interpreter

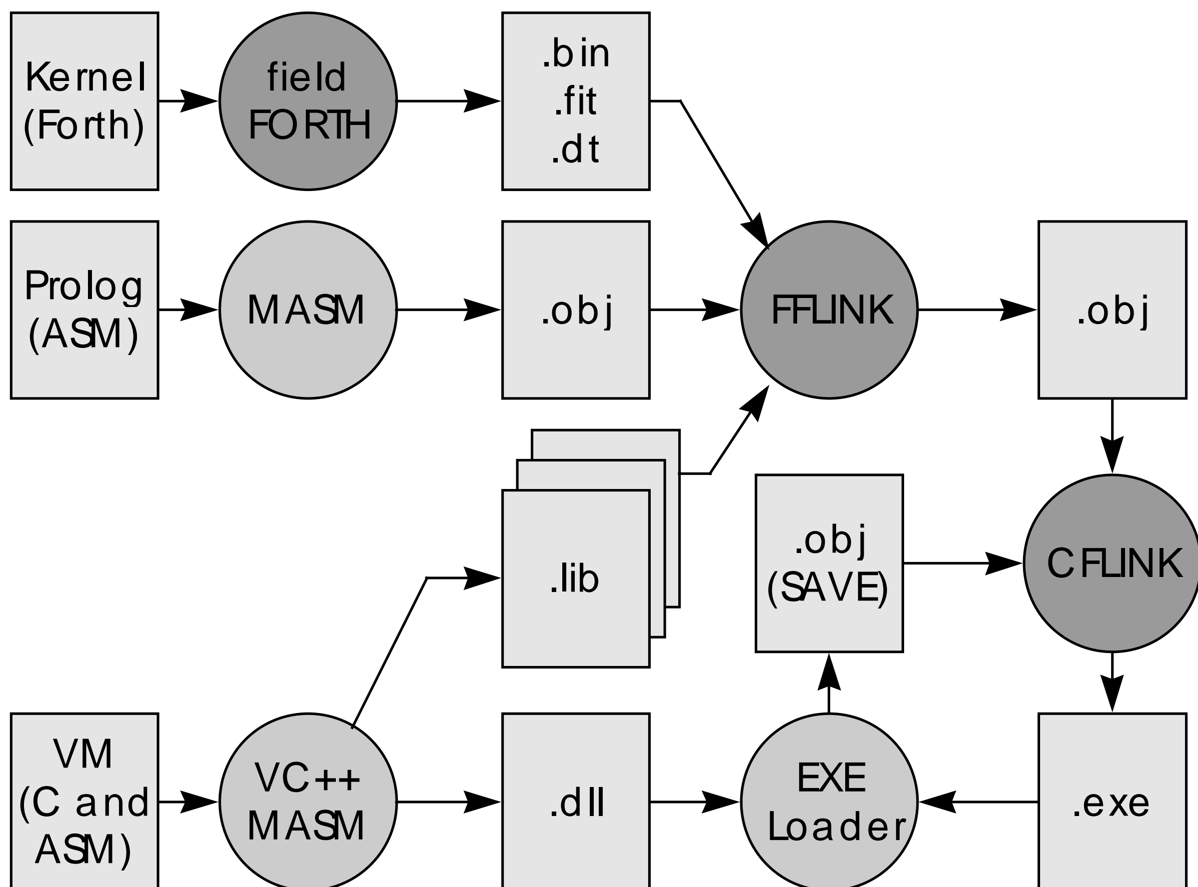
```
colon proc
    sub     ebx, 4                  ;save a return stack cell
    pop     edx                     ;get address of push1
    mov     [ebx], esi              ;save the instruction pointer
    inc     edx                     ;translate push1 to next
    lea     esi, [edi+eax+4]        ;set IP to the parameter field
    jmp     edx                     ;return to next
colon endp


EXIT proc
    pop     eax                     ;get address of push1
    mov     esi, [ebx]              ;reload IP from return stack
    inc     eax                     ;translate push1 to next
    add     ebx, 4                  ;release the top of return stack
    jmp     eax                     ;return to next
EXIT endp


CREATE proc
    add     eax, 4                  ;compute the parameter field address
    ret                             ;return to push1
CREATE endp


DROP proc
    pop     eax                     ;get address of push1
    inc     eax                     ;translate push1 to next
    add     esp, 4                  ;release the top of data stack
    jmp     eax                     ;return to next
DROP endp


_DUP proc
    mov     eax, [esp+4]            ;get top of data stack
    ret                             ;return to push1
_DUP endp


TUCK proc
    pop     edx                     ;get address of push1
    pop     eax                     ;get top of data stack
    pop     ecx                     ;get second of data stack
    push    eax                     ;push a copy of old ToS
    push    ecx                     ;push a copy of old SoS
    jmp     edx                     ;return to push1
TUCK endp
```

Picture 2: Implementation of some typical Forth instructions

Providing Relocation Information

As explained above, the placement of Forth instructions inside a DLL does not allow any prediction of its virtual address after a reload of the whole system. That's why the addresses of all used instructions have to be updated on load-time. The main part of this update process will be done by the loader for executables, which is part of the operating system. This loader uses the information stored in the header of the executable file to load all necessary dynamic link libraries. Additionally, the loader fills a so-called import address table with the addresses of all imported functions.

To utilize these preparations, the Forth system has to provide an import name table which contains the names of used DLLs and the names of the imported functions. Additionally, it has to provide a »word table« which associates every execution token with its corresponding entry in the import address table. So, a little piece of startup code is able to update all execution tokens with the actual code addresses. Even if this description a little bit simplifies the whole loading process, especially for DOES>-defined instructions, it should be sufficient for the scope of this paper.



Picture 3: Development Process of comFORTH 4

There are two distinct situations where the import name and word table have to be created. At first, these tables have to be created from scratch after a recompile of the Forth nucleus, most often by means of a meta-compiler. In the second case, an extended version of these tables have to be created when an extended system or a turnkey application should be created after an interactive Forth session. It's a

good idea to design a two-stage process. Picture 3 shows the process of linking and relocating for the experimental comFORTH 4 environment, which uses STDCALL technology.

The first tool, called FFLINK (fieldFORTH Linker), generates an object file from the fieldFORTH [WOI95] meta-compiler output (binary image and unresolved instructions) and the import libraries of the referenced DLLs, which owns the same format as an image SAVEd during an interactive session. A second tool, called CFLINK (comFORTH Linker), adds the executable header and allows a final readjustment of the preferred load address and table sizes. Additional effort arises from execution token tracking inside the Forth compiler. Every new or modified execution token has to be registered in the word table, and every new imported function has to be registered in the import name table. In that way we have to pay for operating system conformance.

Debugging STDCALL Threaded Code

As outlined above, STDCALL threaded code allows the seamless integration of operating system calls as well as functions exported by software packages written in any other language. On the other hand, we have to pay with an execution performance located in the logarithmic middle of the 1:100 gap between optimized C-code and interpreted Java-bytecode and a considerable programming effort for a full-sized programming environment. That's why it is obvious to ask, whether there are additional benefits of STDCALL threaded code. These can be found for the design of debugging tools.

The word table offers fascinating possibilities for debugging Forth code. A simple lookup allows to decide, whether or not a given integer value may be interpreted as an execution token. This feature can be used to secure the indirect execution of code. Addresses stored on the return stack can be roughly verified in the same way, because they have to point to a cell which contains an execution token. Furthermore, the textual information stored in the import name table can be used by SEE to display the name and the exporting DLL of the function bound to the execution token of a given word.

Much more impressive are the possibilities offered by the »floating« address of the address interpreter. Because none of the primitive instructions has explicit knowledge about the location of the address interpreter, it is possible to replace it by a debugging version. Picture 4 shows a simple version of such a replacement. STEPIN is a forth instruction executed by a primitve word with the same name. It it has to be used in conjunction with stepper code like the following:

```
HERE ] CR ." ip: " DUP U. >R .S  R>
    CR ." continue? (y/N) "
    KEY BL OR CHAR y <> EXIT [
    CONSTANT %STEPPER
```

```
.data

aNEXT       DD  ?   ;contains previous NEXT address
rlSTEPPER   DD  ?   ;points to STEPPER handler code


.code

STEPRET proc
    ;return from high-level STEPPER handler
    add     esp, 4                  ;drop 'real' push1 adress
    add     ebx, 8                  ;drop xt and threaded code fragment
    pop     eax                     ;STEPPER handler result
    pop     esi                     ;next forth instruction to step in
    or      eax, eax                ;test STEPPER handler result
    jnz     stepout                 ;step on further?
    mov     eax, dword ptr [esi]    ;read next execution token
    add     esi, 4                  ;adjust instruction pointer
    push    offset STEP             ;push 'stepping' push1
    jmp     dword ptr [edi+eax]     ;execute next instruction

stepout:                            ;terminate stepping mode
    mov     eax, dword ptr [aNEXT]  ;reload 'real' push1 address
    inc     eax                     ;compute next
    jmp     eax                     ;continue without stepping
STEPRET endp


STEP proc
    ;stepping address interpreter
    push    eax                     ;push result, 'push1' address
    ;call high-level STEPPER code first, it behaves as follows
    ;ds: xxx ip -- yyy ip' ?        ;ip' is the new instruction pointer
                                    ;? is true for end of stepping mode
    ;rs: uuu -- vvv                 ;above STEPPER will find dynamically
                                    ;created xt and code fragment
    sub     ebx, 0Ch                ;allocate return stack space
    mov     dword ptr [ebx+8], offset STEPRET   ;create a xt (of STEPRET)
    lea     eax, dword ptr [ebx+8]  ;absolute address of that xt
    sub     eax, edi                ;make it relative
    mov     dword ptr [ebx+4], eax  ;create a threaded-code fragment
    lea     eax, dword ptr [ebx+4]  ;absolute address of that fragment
    mov     dword ptr [ebx], eax    ;create a return address pointing to it
    mov     eax, esi                ;pass IP as a parameter to STEPPER
    mov     esi, dword ptr [rlSTEPPER]          ;rlSTEPPER becomes IP
    jmp     dword ptr [aNEXT]       ;execute STEPPER using the 'real' NEXT
STEP endp                           ;fall into STEPRET on return


STEPIN proc
    ;start stepping mode with handler stepper on debuggee
    ;ds: rl-debuggee rl-stepper --
    pop     dword ptr [aNEXT]       ;save 'real' next
    pop     eax                     ;get STEPPER procedure
    add     eax, edi                ;make it absolute and save it
    mov     dword ptr [rlSTEPPER], eax
    sub     ebx, 4                  ;push current IP to return stack
    mov     [ebx], esi
    pop     esi                     ;load new IP
    add     esi, edi                ;make it absolute
    jmp     STEP+1                  ;step in, skipping push1
STEPIN endp
```

Picture 4: Simple single-step framework

%STEPPER provides (very limited) single-stepping capabilities wich can be activated as follows:

```
: DEBUGGEE ( -- )
    1 2 + . ;

' DEBUGGEE >BODY  %STEPPER STEPIN
```

Besides single-stepping a broad range of profiling activities may be performed with STEPIN. So it is very simple to determine the calling frequency of any given forth word, or the maximum stack depth during execution, and so on.

Conclusion

The STDCALL threading scheme enables seamless integration of foreign code into a Forth system by means of the underlaying operating system. The register mapping and coding scheme were discussed in detail for the Win32 API and Intel processors. The execution speed of the resulting code can be compared to that of typical indirect threaded code systems. Code optimization techniques such as unrolling the address interpreter may be applied, but were beyond the scope of this paper. There is considerable effort to create the necessary data structures, especially on creating and maintaining the import name table and word table. The additional data structures and the floating address interpreter address allow the simple implementation of powerful debugging tools.

References

[MIC92] Microsoft Corporation: „MASM Programmers Guide, Version 6.1“, Program documentation, Redmond 1992

[WOI95] Woitzel, Egmont: „Emulating Forth: Interactive Cross-Development.“, EuroForth ’95 conference proceedings, Dagstuh Castle 1995

[WOI97] Woitzel, Egmont: „_stdcall threaded Forth.“, Forth ’97 conference proceedings, Ludwigshafen 1997