

CREATE DOES> Discussion

Let's examine the creation and application of CREATE DOES> one word at a time. We will start with a parent word which will create a child word and give that child word its execution process.

I must note a variation in CREATE DOES> specific to W32F. Because of its separation of machine code into the CODE area the contemporary (1982 onward) CREATE DOES> must be adapted. The usual inline JSR, DODOES [See below] is replaced by a code fragment I term <proto-does>. For each CREATE DOES> definer a specific <proto-does> is created.

We will demonstrate with this word pair:

```
: PARENT CREATE , DOES> @ DROP ; \ the definer word
0x1234 PARENT CHILD \ an example defined word
```

These compile into:

```
<PARENT-header> CREATE , (DODOES>) <proto-does> @ DROP UNNEST
```

```
<CHILD-header> <proto-does> 0x1234
```

```
<proto-does>
90 NOP
C7 C1 MOV W, <addr after <proto-does> \ to be the new IP
E9 JMP [long] DODOES
```

```
DODOES \ run-time code
53 push TOS \ make room on stack
89 75 FC mov -4 [RP], IP \ push IP to return stack
8B F1 mov IP, ecx \ new IP
8D 58 04 lea TOS, 4 [W] \ push address of parameter field
8B 46 FC mov W, -4 [IP] \ x on to return stack
ED 04 sub RP, # 4 \ confirm space on return stack
FF 20 exec c;
```

How The CHILD Executes

The execution of CHILD occurs by the machine code <proto-does>. The leading MOV W, <addr> transfers the address two cells after (DODOES>) into register W in preparation to transfer interpretation there. It then calls DODOES, to place the parameter address of CHILD on the data stack and high-level interpretation resumes just after <proto-does> in PARENT. CHILD uses fetch (@) to recover 0x1234 and DROPs it. Nothing fancy, just a demonstration.

How PARENT is created

Let us examine the compiling of PARENT. That is the location of all the action.

After the header is added to the dictionary by ':' (colon), CREATE is compiled. It will later create the CHILD word. The following ',' (comma) is compiled. Next, DOES>, as an immediate word is executed. It handles several tasks:

1. Compile in-line in PARENT the cfa for (DODOES>).
2. Execute the word DOES-CALL, (does call comma) which creates an unnamed code fragment. I will refer to it as 'proto-dodoes' .
 - a. Locate the address of the next cell in the CODE memory area, which will be the beginning of proto-dodoes.
 - b. Compile the address of proto-dodoes in-line in PARENT . This will later serve to locate proto-dodoes to be used by the preceding (DODOES>).
 - c. In the code area for proto-dodoes compile 0xC7 NOP.
 - d. Compile 0xC190 for MOV W, ADDR where ADDR is the address two cells after (DODOES>). This ADDR later will mark the entry for the child word's execution. (Note is is one cell after the actual. entry point).
 - e. Compile 0xE9 as a long relative jump followed by the relative address of the existing W32F dodoes machine code. The relative address is developed by subtracting the address of dodoes from the address of the cell after the completed opcode. (The method for a relative jump.)
3. The compilation of PARENT continues by compiling the cfa's of @ and DROP.
4. The ';' compiles an ending UNNEST and completes the definition of PARENT.

How PARENT Creates CHILD

We will now examine the execution of PARENT as it creates CHILD. First CREATE creates the header for CHILD and comma (,) compiles the parameter 0x1234. (DODOES>) reaches one cell ahead and retrieves the address of <proto-dodoes). That address is written into the code field of the most recently defined word CHILD. PARENT ends execution at that point. The CHILD definition is now complete


CREATE DODES> ala 1970

For completeness and a historical perspective, I'll discuss the prior forms of CREATE DOES>. When originated by Charles Moore, the syntax used <BUILDS DOES>. Following the example above, as compiled it would be in the form:

```

<PARENT-header> <docol> <BUILDS , DOES> @ DROP ENCODE
<CHILD-header> <dodoes> <pointer> 0x1234 (the compiled parameter)
<BUILDS 0 CONSTANT ; \Create header and one parameter for DOES>

```



```

: DOES> \ Rewrite PFA with calling hi-level code address
        \ Rewrite CFA pointing to this dodoes code.
R> LATEST PFA ! ;CODE \ dodoes follows
IP 1+ LDA, PHA, IP LDA, PHA, \ begin Forth nesting
2 # LDY, W )Y LDA, IP STA, \ fetch first parameter
INY, W )Y LDA, IP 1+ STA, \ as NEXT interpreter
CLC, W LDA, 4 # ADC, PHA, \ push address of parameter area
W 1+ LDA, 00 # ADC, PUSH JMP, \ interpret in PARENT word

```

The above is 6502 code reprinted from the fig-FORTH Model by the author (Forth Interest Group, 1980).

When the parent word is created all of the words are compiled, in-line. Nothing unusual at that point.

When the PARENT word executes, <BUILDS uses 0 CONSTANT to creates the CHILD word and reserves the next cell in the parameter field. It then comma (,) compiles the child's parameter 0x1234 after that reserved cell. DOES> executes in conjunction with ;CODE to rewrite the code field of the child word pointing to the code immediately following (dodoes). In the cell immediately after the code field it places the address located in PARENT just after DOES> and ends. This address specifies location of the the high-level code after DOES> be executed by the CHILD word.

When the CHILD word executes it uses W pointing to its code field to load to the data stack with the address two cells later, holding the parameter 0x1234. It then adjusts W and IP to execute the high-level compiled code following DOES> in the parent word.

All was well and good for most applications. As expected, an application would have the address of the parameter field (located one cell later than usual) allowing for the <pointer> parameter. However, in the unusual case, if you were to tick (‘) the child word you would receive the address of <pointer>. You just had to know the parameter field started one cell ahead. This bothered Chuck. Also, that you needed a special word <BUILDS to do the creation.

The New DOES> ala 1982

At the 1982 FORML Conference at the Asilomar Conference Center Chuck gave his new syntax in about four sentences. The group went silent for a moment and the burst into animated discussion. We now had a regular, simpler format. Here is the ‘new’ DOES>:

```

: PARENT CREATE , DOES> @ DROP ; \ which compiles into:
<PARENT-header> <docol> CREATE , (DODOES>)
      here+cell JSR, DODOES @ DROP ENDCODE \ CHILD runtime

```



(CHILD-header) <pointer> 0x1234 (the compiled parameter)

When PARENT executes it creates the child header and commas its parameter 0x1234. DOES> is an immediate word which had written in-line (DODOES>), a pointer one cell ahead and a placed a sub-routine call to DODOES. This structure mimics the runtime for a code word, written inline!

When CHILD executes, its code field points to the here+cell which points to the JSR, DODOES. This follows the form of a code word execution. As DODOES executes the address of the following code is obtained by popping the machine-stack for the call's return address. The address in register W plus one cell is the CHILD parameter field address. These values are used by DODOES to place the parameter field address on the data stack and transfer interpretation to the code in PARENT after the DODOES call. The child parameter field is standard and the control flow more obvious. Hooray!

**** reference material ****

This is a simplified execution sequence for PARENT, as compiled..

CREATE , (DOES>) <address-of-proto-dodoes> @ DROP UNNEXT

0x1234 PARENT CHILD

```
: PARENT CREATE ( -- ) , \ parent definer word
      DOES> ( addr -- ) DROP ;
```

```
: DOES> ( -- ) \ in a parent word compile (DODOES>, a code fragment
in the CODE space and (in line) the address of the code fragment
  COMPILE (DOES>)
  DOES>_B ; IMMEDIATE
```

```
: DOES>_B ( -- ) \ create a code fragment in CODE which links
execution to the DOES> area of the parent word.
  DODOES-CALL, ;
```

This creates a code fragment to direct execution for the CHILD word to the PARENT portion after DOES>, really the compiled (DOES>) and following address parameter. DODOES sets up the pfa address.

```
: DODOES-CALL, ( -- ) \ compile call to does> (in code-only section)
   code-align code-here , \ for (;code) to pick up
   0xC790 code-w, \ NOP
   0xC1 code-c,
   HERE cell+ code-, \ nop mov ecx, # ? cell+ ; new IP
   0xE9 code-c,
   DODOES code-here CELL+ - code-, \ JMP (long) to DODOES
;

: (DOES>) ( -- )
  LatestXT @ R> TUCK @ OVER !
  turnkeyed? if 2drop else Sys-warn-does? then ;
```

\ DODOES-CALL, builds code (in code-only section) that loads the
 \ cfa following DOES> and jumps to DODOES

This a headless code fragment

```
CFA-CODE DODOES ( -- a1 ) \ runtime for DOES>
53      push  TOS          \ make room on stack
89 75 FC  mov   -4 [RP], IP \ push IP to return stack
8B F1     mov   IP, ecx     \ new IP
8D 58 04  lea  TOS, 4 [W]   \ push address of parameter field
8B 46 FC  mov   W, -4 [IP]  \ x on to return stack
ED 04     sub  RP, # 4      \ confirm space on return stack
FF 20     exec  c;
```