ADVANTAGES OF A FULLY SEGMENTED FORTH ARCHITECTURE

Jim Callahan
Harvard Softworks
PO Box 69, Springboro, OH 45Ø66
(513) 748-Ø39Ø

The structure of Forth definitions can be exploited to a far greater extent than generally practiced, yielding improved speed, compactness, and flexibility, and virtually eliminating limits on program size. Except for incidental pointers, Forth definitions consist of machine code, definition or parameter lists, and an optional identifier string. The HS/FORTH Development System provided by Harvard Softworks since 1983 compiles words so that each of these portions is segregated into its own segment. Although 8Ø86 family processors form a very natural fit to this organization, it is also appropriate to any processor capable of partial width addressing. This architecture permits programs larger than 64k to be written using fast, compact 16 bit addressing, and is transparent to most applications. Very large programs can utilize additional code and list segments almost as efficiently as the base ones. Each vocabulary also has its own segment, permitting easy dynamic enhancement of both structure and access (ie, reclaiming space from unneeded headers or grafting of hash tables for 1Ø thousand line/min compilation). Assembled or machine optimized phrases can be very easily included within colon definitions, and colon definitions can be executed from within assembled words. Also, any word can be redefined retroactively at any time, and it is easy to define defining words that create any number of code fields. Finally, the segmented architecture lends itself to the metacompilation of separate systems using any threading or segmentation scheme, or easy use of the system as a cross assembler.

THE ARCHITECTURE

HS/FORTH initially consists of four segments: CODES FORTHS STACKS and LISTS. FORTHS contains the name strings for the FORTH vocabulary. The entry for each word contains a 2 byte pointer into the LISTS segment, the name string, and the count byte. Creation of each new vocabulary causes the creation of an associated segment, and since vocabulary segments are pure there is no need for thread pointers or vocabulary numbers to complicate the structure. The pointer gives the code field address in the LISTS segment. That field is followed by the parameter field, as in conventional Forth implementations. The code field gives the address of the machine code primitive located in the CODES segment. Although each of these segments is limited to 64k, both additional system segments and data segments can be created as needed.

Segment management is accomplished through a linked list of segment descriptors. Each descriptor holds the address of the base paragraph for that segment, the maximum number of bytes available to the segment, the offset of the first free byte, the offset of the beginning of the relevant part of any word being defined, and a link to the previously created segment. All segments are completely relocateable anywhere in memory, and the size of each can be increased or decreased as needed. Reinterpretation of the base paragraph field as a segment selector will allow use under native mode operating systems when they become available, and the segmentation scheme allows easy use with current "expanded" ram as well.

The system segments map naturally to the segment registers of Intel chips with CODES assigned to CS, STACKS to SS, LISTS to DS and vocabulary to ES. This means that considerably more than 64k memory is always available for processing without ever having to waste time changing a segment register. And the appropriate register is always selected by default, never specified. Most non-segmented processors do provide 16 bit addressing modes with a number of base address registers, enabling implementation of this architecture even where the register mapping is not as obvious.

THE ADVANTAGES

With the initial CODES and LISTS segment up to 128k of memory is available for "the program". Since this space is not cluttered with headers, and pointers occupy only 16 bits, and machine code is more compact than on conventional systems, this is sufficient space to contain programs that would occupy most of memory using 32 bit Forths. Having a pure machine code segment allows code primitives to overlay one another. A simple example is the word set containing STEP DUP@ and @ (where STEP is equivalent to 2+ DUP @). All are entry points into a single machine code primitive and were very conveniently defined using:

```
 CODE STEP  BX INC. BX INC.  +CODE DUP@  BX PUSH.
+CODE @     BX [BX] MOV.        END-CODE
```

Obviously this construct is not possible in conventional systems.

Should 64k of pure multiple entry code primitives prove insufficient the easiest solution is to create one or more additional segments to receive all long code primitives. If we choose to offload long primitives to a segment called CODEX we could use the following:

```
CODE BIG-ONE  CODEX TRANSFER.  definition . . . . . . . .
              CODES TRANSFER.  END-CODE
```

The long jumps to and from the other segment have little impact on run time since they are only used for long primitives, and data addressing is not affected since only the code segment register is altered. Actually, the assembler can assemble to any segment and can therefore also be used to create conventional assembly language programs which can then be saved with SAVE-BIN or SAVE-COM as needed.

Another benefit of separating code from the rest of the definition is that machine code can easily be included within a colon definition. This trivial example prints the sum and difference of two numbers:

```
: S&D  [% AX POP. AX PUSH. BX PUSH. BX AX ADD. %] ." Sum= "  .
       [% AX POP. BX AX SUB. %] ."  Difference= "  . ;
```

Similarly, optimizer output which would normally be used to create a code primitive can also be integrated into colon definitions, possibly to speed computation of the sum of squares of numbers in a table:

```
: TOTAL    [%" 0 F=0  DO I DAT-TAB F**2 F+ LOOP " %]  F. ;
```

Even more important is the ability to execute colon definitions from deep inside machine code primitives. This is especially useful in two cases, exception handlers for error conditions occurring within code primitives, and interrupt handlers. The latter can also be handled entirely in high level Forth by the INT: type colon definition available in HS/FORTH. These facilities have been used to add time slice multitasking to HS/FORTH in addition to the more typical round robin multitasking.

Since a program is more likely to run out of room in LISTS than in CODES, we have provided words to create additional definition list segments. MAKE-LIST and USE-LIST are used to create and select alternate list segments. MAKE-LIST copies to the new segment that small portion of the old LISTS segment which has already been defined. Just as HS/FORTH can assemble into any segment, it can also compile into any segment. Since there is some memory overhead for having copies of the root definitions in each list segment, this approach is appropriate for applications that can be reasonably divided into separate subtasks, each of which requires only the basic words in common. This is also the basis for a multi-user system currently under development in which each user has their own separate definition list segment.

This approach is, however, not always appropriate. It is inefficient when large parts of the application are all so interrelated that there is no easy subdivision of words into separate segments. EXPORT-SEGS also creates an alternate list segment, but only copies the lowest few hundred bytes containing the cfa's of the most essential words such as DOCOL, EXIT, and the various literal, branch and loop primitives, and the inter-segment thread return primitive. It also creates a root vocabulary for the words that will be defined in the new segment. Any word defined in any segment can be used in any colon definition in any other segment, provided that it has been declared as imported. Code primitives can be imported with IMPORT-CODE which establishes a two byte local cfa. There is no run time overhead for importing a code primitive and the data segment in effect during execution is that of the host word.

More generally, any word can be imported with the universal import word:

```
        IMPORT  word-name  word-source-segment-name
```

This creates a six byte transfer vector in the target definition list segment, the first two bytes serving as the imported word's local cfa. Run time overhead is minimal, about the same as for a single colon nesting level. Since execution would typically pass through several colon nesting levels in one segment, then cross into another segment and pass through several more nesting levels before returning, run time overhead will generally be insignificant.

An additional benefit to multiple well defined code and list segments is that temporarily unneeded ones can be easily swapped out to disk or expanded memory - providing essentially unbounded program size with very minimal overhead. This also applies to data segments and vocabulary segments. Remember that each vocabulary has a segment of its own.

Since vocabulary segment entries form a simple list, and there are no pointers into the vocabulary segments, it is possible to delete word headers at any time and slide the rest of the vocabulary entries down to reclaim the space vacated. Besides saving space, this provides a facility for local definition where all word names that should remain private within a module are beheaded at the end of that module. It is also simple to create synonyms for previously defined words. Since they do not occupy program space, and since the space they do occupy can be reclaimed at any time, there is absolutely no reason not to use synonyms wherever they would make your program more readable.

A final benefit of the independent vocabulary segments is the ease with which they can be modified for even greater performance. We recently added another short utility to the system which when loaded converts the vocabularies in the running system for hashed access -- and increases compilation speed to about ten thousand lines per minute on an AT.

Integrating all the segment management features mentioned above gives a system which is almost inherently a metacompiler. We provide several files which illustrate use of the system as a metacompiler ranging from a simple few hundred byte meta-compiled utility to a complete Forth execution core and demo message that fits in under 2k. That example features interleaved codes and lists, but separate headers, as is appropriate for small utilities. It would also be possible to metacompile direct, subroutine, or token threaded systems in the target segments.

Direct threading is appropriate for small systems, but colon nesting time dominates large deeply nested systems and in this case direct threaded systems are either slow and waste a lot of extra space or waste some space and are very slow, except of course for very shallowly nested tasks such as advertising benchmarks. That is why our main system is indirect threaded. If the extra 5 to 10% time for the indirect threaded NEXT is unacceptable, it is better to speed execution by a factor of 5 or more with the optimizer, which would generate the same code whether the system were direct or indirect threaded.

Now that we've squeezed about all that we can out of the system segments, it's time to attack the data segments. When it comes to large arrays, segments are indeed a bit of a nuisance. But not an insurmountable one. Small arrays can usually be kept in the parameter field in the definition list segments. Being firm believers in multiple code field definitions we provide 1 through 4 dimensional arrays for all data types (through quad length integer plus all 8087 types) with fetch as the default action and store and offset as options. This would take a lot of space for the primitives in most systems, but using multiple entry points into collective code primitives compacts things immensely. Another complete set is available for data arrays in other segments, including arrays that span segments. The only limit is 64k items of whatever size per array. A double precision floating point array could span a linear address space of half a megabyte with no need to ever worry about which particular segment was being used. All you need is the indices, the array name, and possibly a prefix to select the store or offset option.

CONLUSION

We continue to discover more and more advantages to intelligent segmentation of Forth definitions. We have yet to discover any significant disadvantages. The real proof of its value is that competitors have finally begun attempting to adopt the architecture that we pioneered.