Sections

M. Anton Ertl*
TU Wien

Abstract

A section is a contiguous region of memory, to which data or code can be appended (like the Forth dictionary). Assembly languages and linkers have supported multiple sections for a long time. This paper describes the benefits of supporting multiple sections in Forth, interfaces and implementation techniques.

1 Introduction

A section is a contiguous memory area, to which new data can be appended at the end; the Forth dictionary is a section. Assemblers and linkers have supported multiple sections or segments for many decades [Lev00]. In contrast, Forth traditionally has had only one section; some systems have had separated headers (another section), and cross-compilers have uninitialized memory for `buffer:`, but by and large, Forth systems have made do with just one section: the dictionary. With multiple sections, each section has it's own start, dictionary pointer (what `here` reads), and end (used in `unused`, but otherwise not used much).

This paper presents various uses of sections and why they are better than the current workarounds (Section 2), presents a programming interface (Section 3), and discusses various implementation approaches (Section 4).

2 Uses

2.1 Nested structures

You often build one structure A in memory, and in the middle of that, have to build some out-of-line part B, and afterwards continue building A. If you have two sections, that is easy: you put A in one section, and B in the other section. In Forth, you traditionally use one of the workarounds:

- You select a representation for A that does not require contiguity.
- You put B in `allocate`d memory. Unfortunately, that usually means that B does not survive a `savesystem`, and it's also cumbersome if B is a growable structure.

A particular instance of this problem is when A is a colon definition under construction, and B is the data for a string or floating-point literal. Forth compilers traditionally work around this by not requiring contiguity.

A typical solution is to call a word such as `(s")` or `flit`, and follow that with the inline data. These words get the return address from the return stack, use that to push the relevant data on the data/FP stack, then increment the return address to skip over the data, and then either return to the changed return address or jump to it. Both ways are very expensive on modern CPUs, because they cause mispredictions from the hardware return stack[1]: If the changed address is returned from, the return incurs a branch misprediction (about 20 cycles on a modern Intel or AMD CPU); if the changed address is jumped to, the jump has a chance to predict correctly, but all outer returns will mispredict once (at about 20 cycles per misprediction).

A faster approach is to jump across the data, and then let some code push the data on the data/FP stack. This does not cause significant mispredictions, but the code is bigger (jump plus inlined literal code).

Finally, if you put the data elsewhere (i.e., a different section), you get fewer mispredictions, and you save the space for the jump around the data.

As an example of the benefit of putting the data out-of-line, consider the following micro-benchmark:

```
\ inline variant
: foo1 123e f+ ;

\ out-of-line simulation
123e fconstant x
: foo2 x f+ ;

defer foo
: bench 0e 100000000 0 do foo loop f. cr ;
```

*Correspondence Address: Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria; anton@mips.complang.tuwien.ac.at

[1] The hardware return stack is not the Forth return stack; it is a hardware branch predictor that predicts that returns will return right behind the corresponding calls).

With VFX[2] 4.71 on a Core i3-3227U (Ivy Bridge), the `foo1` version takes 48 cycles, 11 instructions and 1 branch misprediction per iteration, while the `foo2` version takes 6.5 cycles, 7 instructions, and 0 branch mispredictions per iteration. If VFX would put the floating-point number in `foo1` in a separate section instead of inline, it could achieve the same performance as `foo2`.

Quotations are another case of having to build something else in the middle of a colon definition; in this case the "something else" is a colon definition itself. Again, the usual implementation is to jump around it (used in, e.g., Gforth), and putting the quotation in a separate section can save that overhead. In this case, however, two sections are not sufficient, as quotations can be nested arbitrarily deeply, so you need a whole stack of sections.

Locals are another case where you have to build some additional stuff (in this case, headers) in the middle of a colon definition; the headers are no longer needed at the end of the colon definition and their space can be reclaimed, so the usual inline-and-skip approach is particularly inefficient here. Locals in Gforth were developed before sections, and the code for dealing with that problem is complicated; we foresee it becoming much simpler once we take advantage of sections, but we have not made these changes yet.

One way of implementing recognizers is to create a temporary word for each recognized string, then treat the temporary word like an ordinary word (i.e., `execute` or `compile,` it), and finally, delete the temporary if no longer needed [Ert16]. With sections, this is relatively straightforward to implement (especially the case when you cannot delete the "temporary" and have to make it permanent).

2.2 Separate code and data

Most Forth systems still put code and data in the dictionary in an interleaved way. Since the Pentium (1993) and its separate instruction and data caches, this interleaving has been a performance problem on Intel and AMD CPUs (e.g.,). Forth systems have tried to mitigate this problem by at least not putting code and data in the same cache line (by inserting appropriate padding); e.g., VFX aligns data to 32-byte boundaries, but apparently 64-byte alignment is necessary on recent Intel CPUs to achieve the desired effect. And in some cases an important padding is missing, resulting in 350–500 cycles per iteration in VFX and SwiftForth:

```
0 value x
: foo 10000000 0 do 1 x +! loop ;
here to x 0 ,
foo
```

[2] I use VFX for the performance results in this paper, because it is a high-performance system, where one would expect good performance also for the micro-benchmarks I present.

With sections, the data can just stay in the ordinary dictionary section, and the code can have a separate section (or a stack of them, for quotations), thus separating code and data for good. Moreover, systems can get rid of all the padding they insert at the moment to work around this problem.

2.3 Further uses

The uses above are not an exhaustive list. When I presented sections to other Gforth developers, they came up with uses I had not thought of during development (e.g., simplifying the locals implementation).

3 Progamming interface

In the following, "switching a section" means that the dictionary pointer (what `here` reports, and where `allot` allocates) is now the dictionary pointer of the switched-to section.

The words for working with sections are:

`.sections ( -- )`
: display all sections

`next-section ( -- )`
: switch the current section to the next section in the stack, creating it if necessary

`previous-section ( -- )`
: switch the current section to previous section (the next section still exists afterwards).

`extra-section ( size "name" -- )`
: create a named section stack *name*.
name execution: `( xt -- )`
switch the current section to the first section of *name* if there is no outer call to *name*, or the next section if there is; execute *xt*, and switch the current section back on leaving *name*.

For multi-tasking, the dictionary and the named section stacks should have per-task instances that are instantiated on-demand.

Currently the section implementation in Gforth only supports the dictionary as a section stack, named sections without stack, and no proper handling of per-task section stacks, yet.

4 Implementation

The implementation of sections for use within a session is pretty straightforward: Just a data structure with start, end, and section-dp per section,

and ways to manage named sections and a stack of sections.

Things get more interesting when it comes to implementing `savesystem`. There are two basic options:

- Keep all the sections, and preserve them into the next session.
- Collapse all the sections into one (the dictionary), and start the next session with just the dictionary, and with empty named sections.

The current implementation in Gforth takes the collapsing approach. One advantage is that the loader (which does not know about sections) does not need to be changed.

Traditionally, Gforth creates a relocatable image by running Gforth twice and doing the same things, and finally saving one non-relocatable image per run; the non-relocatable images are for different addresses, and by comparing them, we can tell if a cell is an address (then they differ by the difference in image start addresses), or something else (then they do not differ); if they differ, but by a different amount (e.g., because the cell contains the address of an `allocate`d piece of memory), the process outputs a warning.

With sections, this process had to be enhanced as follows: When saving an image, first the dictionary is written, then the other sections, and sections meta-data last. The sections meta-data contains the length and the original start address of each section, and also allows us to determine where in the non-relocatable image the sections are. If two cells differ, the comparison program looks for each image, whether the value of the cell, interpreted as address points into one of the sections, and computes the offset into the collapsed image from that; if, for both images, this gives the same offset o, then the cell is a relocatable address, with value *image-start*$+o$, and that's what the comparator stores in the relocatable image. I.e., in the relocatable image, the original section structure is no longer present.

Of course, that is not the only option. E.g., a system without relocatable images could just save each section as ELF or COFF section with a fixed virtual start address. The OS loader would then load all the sections where they belong (untested).

5 Conclusion

Supporting multiple sections in a Forth system provides a number of benefits. Forth systems have used workarounds to deal with the absence of sections, but these workarounds have a cost both in complexity and sometimes also in performance that can be eliminated by adding sections.

The interface for working with sections is simple, consisting of just a few words.

The implementation is not that complex, either. Dealing with sections across `savesystem` does take some additional effort, however.

References

[Ert16] M. Anton Ertl. Recognizers: Arguments and design decisions. In *32nd EuroForth Conference*, 2016.

[Lev00] Levine. *Linkers and Loaders*. Morgan Kaufmann, San Francisco, 2000.