SystemArchitecture

Updated Dec 19, 2007 by cbiffle

Kernel

The kernel is a small chunk of Propeller machine code, written in assembly language. (Other languages might call it an "interpreter," but Forth uses that word for a different part of the system: the part that accepts user input and performs actions.)

Each Cog that wants to execute Forth code must load and run a copy of the kernel. On system startup, Cog 0 runs the kernel, and all other Cogs are dormant -- but any number of Cogs may simultaneously run Forth out of the shared image by loading their own copies of the kernel, and keeping their execution state separate (via small separate areas of RAM known as USER areas).

The kernel provides a few basic services:

1. Access to Propeller hardware operations, such as manipulating memory, performing arithmetic, and I/O;
2. Code fragments to start execution of common types of words (colon-definitions, variables, constants, etc.);
3. "Reflective" operations to write low-level code in high-level Forth, such as manipulating the stack memory directly.

Because it is so simple, most users don't need to modify the kernel. Most development on PropellerForth itself has left the kernel untouched -- even while adding features such as multitasking and control of the Propeller's hardware counter/timers.

All code in the kernel is either part of the *bootstrap sequence,* which loads Forth into a new Cog (including starting the system for the first time), or part of various *primitive words,* which are Forth words implemented in assembly language.

For the purposes of this discussion, all code *outside* the kernel will be referred to as "user code," even if it's provided with the system.

Primitives

A number of "small" Forth words are implemented in the kernel as *primitives* -- short assembly-language routines.

Certain classes of words must be implemented as primitives:

- The NEXT pseudo-word;
- Words that directly access hardware features not exposed through other words: arithmetic, register access, stack manipulation, flow control, and the like;
- Words that will be referenced from the code field of other words, such as DOCOLON or DOVAR (for rationale, see the discussion of code fields below);
- Words that, due to timing requirements, aren't yet suitable for definition as user words (the bootstrap terminal I/O support is the only current example of this).

Some other words, such as multiplication and division, are implemented as primitives for speed.

If part of a system needs to be in assembly language, due to performance or timing requirements, it doesn't necessarily need to be in the kernel as a primitive. Using the COGINIT word in the standard library, any chunk of RAM can be loaded into an available Cog and started as a separate process, independent of the Forth kernel. This is particularly appropriate for device drivers that either run in isolation, or communicate with the rest of the system through shared memory.

Bootstrap Sequence

There are two phases to the bootstrap: *system bootstrap*, used when the PropellerForth image is initially started, and *kernel bootstrap*, when a kernel is started on a new Cog (including the first).

System Bootstrap

The system bootstrap routine (`bootstrap.paf` in the source) constructs an initial USER area for the first kernel, populating it with pointers to stack space in high memory. It sets the USER area's FIRST-WORD field to the outermost word to run on system startup -- by default, INTERACTIVE, but the user may change this to run their program directly.

Once the USER area is set up, an initial copy of the kernel is spawned on Cog 0 using the standard COGINIT sequence, the same way user code might spawn a kernel.

Kernel Bootstrap

(See `kernel-bootstrap.paf`)

When a kernel wakes up in a new Cog, it expects the address of its dedicated USER area to be in the PAR register. It copies the initial values out of the USER area into registers.

Currently, any kernel that starts in Cog 0 assumes it is the boot kernel. It will assume control of the serial terminal using the slow-and-slightly-flaky bootstrap terminal routines (and, on the demo board, set some LEDs). This process is slightly inelegant, and will be revisited in a later version.

Once the kernel has configured itself based on the provided USER area, it loads its starting point from FIRST-WORD and starts it with NEXT.

Stacks

PropellerForth, like most (all?) Forths before it, uses two stacks: data and return. Both are 32 bits wide.

Each kernel has its own data and return stack (and, if multitasking is enabled, each task has its own). The stacks live in shared RAM, but the tops of each are cached in registers. (Implementing the stacks in Cog-local RAM would be a good area of research, but it's difficult due to the Propeller's inability to indirectly address local RAM.)

Multitasking and the USER area

The kernel accesses state of the current task -- such as where input and output come from, the contents of various buffers, and the stack -- through a block of shared memory called the USER area.

This approach, which appeared in Forth several decades ago, is subtle but powerful. While the concept is quite simple, it allows all multitasking support to be written (and tested) in user-defined Forth words. (PropellerForth's multitasking support is at the end of the `core.4th` file, not in the kernel.)

This allows users to try out new schedulers or task-switching implementations without modifying the kernel.

It's worth noting that many of the fields in the USER area, particularly the stack pointers, are cached in registers within the kernel. As a result, peeking or poking at the contents of the USER area may not affect/reflect the current kernel's operations. USER fields are flushed only on a task switch (again, by user code, such as PAUSE in `core.4th`).

Speed vs. Size

The design of the kernel must balance speed with size:

- All code, even the simplest user-defined words, interacts with the kernel. Thus, any bottleneck in the kernel is a bottleneck in the system as a whole.
- The entire kernel must fit in 2KiB, so it can be loaded into local memory on a Cog. The kernel is designed such that, once it has been loaded into a Cog, the copy of the kernel in shared memory can be modified or destroyed.

In general, the current implementation favors speed over size where possible, except in a few rarely-executed primitives. All primitives are carefully tuned to match the Propeller's *memory access rhythm:* since each Cog can access main memory during a brief window every 16 cycles, the ordering and timing of instructions is very important. Subtle changes in instruction layout can change the system's speed by several orders of magnitude.

Shared Memory

Besides a small amount of runtime state (task stacks and USER areas), shared memory is used to hold the *dictionary* of defined words and their implementations. All memory not occupied by the dictionary, stacks, and USER areas is available for use by programs (currently about 24KiB on the default install).

Dictionary

The Dictionary grows from the base of RAM, and contains the definition of all Forth words. The dictionary is a singly-linked list, and can be searched from the most-recently-defined word back.

The design of the dictionary is quite traditional; each entry contains (in order)

- A *link field* pointing to the previous word in the dictionary,
- A *name field* containing its name,
- A *code field* with the Cog-local address of the primitive to use to start the word, and
- A *parameter field*, containing data to be processed by the primitive in the code field.

(The addresses of these fields are often abbreviated NFA, CFA, LFA, and PFA, respectively.)

Most fields in the dictionary are *halfcells* -- 16-bit values. This is because the dictionary tends to be mostly pointers, and pointers on the Propeller have only 16 significant bits. (This results in a space savings of nearly 50% over full cell pointers.)

The Link Field

The *link field* is a halfcell pointer to the previous word in the dictionary. Specifically, it points to the previous word's *code field*.

Thus, to traverse the dictionary, one must convert code field addresses into link field addresses using the `cfa>lfa` word:

```
latest H@   \ Get CFA of latest word
begin dup while   \ Follow until we hit end-of-dictionary (0)
  \ ...do some work here...
  cfa>lfa  \ Convert the CFA pointer to LFA
  H@    \ Load the half-cell link field
repeat
```

The Name Field

The *name field* contains the name of the word.

Specifically, it consists of eight bytes:

1. **Flags and Length:** the MSB (bit 7) is set if the word is IMMEDIATE, and bit 6 is set if the word is "smudged" (hidden from dictionary searches). The remaining six bits indicate the length of the name.
2. **Name Bytes:** the first seven bytes of the name. If the name is shorter than seven bytes, the name field is padded with unspecified data.

Thus, only the first seven characters of names (plus their length) are significant: X2345678 and X2345679 can't be distinguished by the system.

The Code Field

The *code field* contains the Cog-local address (i.e. address within the kernel) of the primitive to use to start the word.

For dictionary entries describing primitive words themselves, this points directly to the primitive, which executes and returns.

For colon-definitions, it points to the special primitive DOCOLON, which saves the state of the calling word and moves execution into the new word.

Other special word types (such as VARIABLE, CONSTANT, and USER definitions) have their own primitives in their code fields.

The Parameter Field

The parameter field contains some data for use by the primitive referenced by the code field.

How the parameter field gets interpreted is completely up to the primitive. Some common examples:

- Colon definitions (DOCON) interpret it as a string of execution tokens making up a user word.
- Variable and constant definitions (DOVAR, DOCON) use it as storage space for the variable or constant value.
- User definitions (DOUSER) use it to store the offset of the word's value in the USER area.

Threaded Code

PropellerForth compiles colon-definitions into *indirect threaded code*, or IDTC.

IDTC is typically slower than the other two threading methods, direct threaded (DTC) and subroutine threaded (STC). IDTC was chosen over the others because of the Propeller's peculiar architecture:

- Because executable code lives in a separate address space from user definitions, DTC could reference one or the other, but not both (without some sort of tagging scheme, which would be slow).
- Similarly, because subroutine calls branch within Cog-local memory, STC is difficult. Even using the proposed "large memory model" execution method, a single 32-bit instruction can't load a 16-bit literal address, requiring at least two instructions to jump to a location in shared RAM.

In preliminary research, PropellerForth's implementation of IDTC is *faster* than all implementations of DTC/STC we tried.

IDTC has one primary disadvantage: the code field of words must point into Cog-local RAM, so user words defined in assembly language (CODE words) are difficult to implement. In the future, we might be able to circumvent this by using the "large memory model" approach (which unfortunately carries a 4x performance penalty, best-case), or copying CODE words into a Cog-local buffer.

IDTC does, however, have an advantage: because all locations in shared RAM can be uniquely addressed with 16 bits, PropellerForth uses 16-bit dictionary fields and execution tokens where possible. This shrinks the dictionary by nearly 50%.

► Sign in to add a comment