TaskingOverview

Updated Dec 28, 2007 by cbiffle

A discussion of PropellerForth's multitasking capabilities, with examples.

The Propeller can execute eight independent processes in hardware, running one on each Cog. PropellerForth (like many Forths before it) provides an additional capability: to run many tasks on a *single* Cog, passing control between them cooperatively.

We call this feature *tasking* (short for *multitasking*).

Tasking Concepts

Tasking in PropellerForth is a form of *cooperative multithreading*. All tasks share main memory and the dictionary, but have separate stacks and USER areas. New tasks can be created and (optionally) *scheduled* for execution. A task can elect to temporarily give up control and pass it to some other task using `PAUSE`.

Initially, only one task is scheduled: `OPERATOR`, the interactive task that communicates with the console. Before other tasks are scheduled, `PAUSE` is effectively a no-op.

As additional tasks are scheduled, they form a *task ring*, with each task linked to the tasks scheduled before and after it (a doubly-linked circular list). `PAUSE` passes control to the next task in the ring, and when (if) control comes back around the ring to the task that called `PAUSE`, the task resumes. There is no concept of task priority unless the user rearranges the ring. (The scheduling method will become pluggable in a later version, with this round-robin algorithm

as only one of the options.)

Currently, a task ring is executed by a single Cog, though each Cog may have its own task ring. Tasks share the Cog hardware resources, including the counters and pin I/O registers, so if multiple tasks use these resources, they must be careful to only change the aspects they own.

Blinky: A Brief Example

In this example, we'll create a simple task that toggles a debug LED each time it gets control.

All words below assume the base for number entry is hexadecimal. Before attempting to enter the code, please set the entry base by typing `hex` .

LED manipulation (target-dependent)

First, two basic words: a word to set the debug LEDs to output, and a word to toggle their state. The form of these words differs depending on the target board, but the rest of the code is target-independent.

On the Demo Board or compatible, use:

```
: ledinit   DIRA L@   00FF0000 or  DIRA L! ;
: ledtoggle OUTA L@   00FF0000 xor OUTA L! ;
```

On the HYDRA, use:

```
: ledinit   DIRA L@  1 or  DIRA L! ;
: ledtoggle OUTA L@  1 xor OUTA L! ;
```

Feel free to test these words out before moving on; after `ledinit` is called, `ledtoggle` should toggle your debug LEDs.

Tasks and scheduling (target-independent)

Now, define a simple word that will loop forever, toggling the LED and pausing.

```
: blink-forever
  ledinit
```

```
begin  ledtoggle pause  again ;
```

Don't test this word out just yet. It will do exactly as we told it, and loop forever. This will make it difficult to try the code below.

Now, create a new task to run this word, using the TASK word. TASK is a defining word, like VARIABLE or CONSTANT -- it takes some data from the stack, reads a name (entered after TASK), and creates a new word. In this case, task reads the sizes of the data and return stack and the size of the USER area.

```
8 8 #user task blinky
```

This creates a new task, blinky, with 8 cells each of data and return stack, and a USER area of the default size (provided by #USER). All task information is written to the dictionary, so it will survive a write to EEPROM and reload, even if it's running but paused!

You can test out blinky -- it should return an address. This is the address of the task's *task control block* (TCB), a small block of RAM that defines the task and its scheduling information.

Now, start blinky running our infinite loop using ACTIVATE:

```
' blink-forever blinky activate
```

ACTIVATE consumes a TCB and the address of the word to run in the task, schedules the task in the task ring, and immediately passes it control. If all is well, the debug LEDs should have toggled as blinky started.

The LEDs should have toggled only once, however, and PropellerForth should have responded with a prompt. This is because, even though blink-forever is an infinite loop, it calls PAUSE each time through. blinky is now waiting at that call to PAUSE.

Try entering PAUSE a few times at the console (the OPERATOR task). Each time, blinky takes control and toggles the LED before passing the machine back to you.

Once you're confident that blinky is doing its thing, you can unschedule it using UNSCHEDULE:

```
blinky unschedule
```

If you enter PAUSE now, nothing happens -- because OPERATOR is now the only scheduled task, just like when the system started. blinky was not harmed, however, and if you miss it, you can reschedule it explicitly by using SCHEDULE-NEXT:

```
blinky schedule-next
```

After SCHEDULE-NEXT, blinky is scheduled again, but unlike with ACTIVATE, blinky does not execute automatically (you'll have to PAUSE to see blinky in action again).

Digging Deeper

PropellerForth's tasking code is implemented in Forth code (tasking.4th in the source distribution). The source is a great place to go to understand the nitty-gritty of tasking, or change how it works.

Glossary of Tasking Words Used

ACTIVATE *(xt tcb --)* Schedules 'tcb' next in the ring (as SCHEDULE-NEXT), and configures it to execute 'xt' when it receives control. Immediately passes control; ACTIVATE only returns when control next comes around the ring.

PAUSE *(--)* Passes control to the next scheduled task in the ring. Returns when control comes back around.

SCHEDULE-NEXT *(tcb --)* Schedules 'tcb' next in the ring after the current task. Does not PAUSE to execute it.

TASK *(pstack rstack usize "name" --)* Creates a new named task in the dictionary, with 'pstack' cells of parameter stack, 'rstack' cells of return stack, and 'usize' *bytes* of USER area. Does not schedule or start the task.

UNSCHEDULE *(tcb --)* Removes 'tcb' from the task ring, linking the tasks on either side of it and linking 'tcb' to itself. If 'tcb' is the current task, it is left by itself in its own ring. If 'tcb' is not scheduled, this is effectively a no-op.

► Sign in to add a comment