The **amforth** Cookbook

Multitasking

Author: Erich Wälde (ew.forth@nassur.net)
Date: 2011-12-19

typeset with LaTeX

Multitasking

Multitasking is a way to execute separate chunks of program code (tasks) apparently simultaneous on a single CPU. Of course, the separate tasks will run one after another. If the CPU can switch between them fast enough, separate tasks may appear to execute in parallel.

Multitasking in `amforth` is achieved as *cooperative multitasking*[1]: In every task the programmer defines places, where control is given up, such that the next task can run. The tasks current state is stored in a piece of memory called the task control block (TCB). TCBs are organized in a simple, linked list and are visited in round–robin fashion.

What is a Task?

Every task owns a piece of RAM, where it finds a set of runtime information (user area) and where it has its own space for the data and return stacks. This space is called a *task control block (TCB)*. Is is referred to by the *task id* or *tid*, which happens to be the start address of the TCB by convention[2].

The runtime information includes:

- status, whether the task is *awake* or *sleeping*
- follower, points to the next task in the list
- where do the stacks start and how many entries are currently on them
- the current value of ”base”
- pointers to deferred words such as **`key`**, **`emit`** and the like
- the content of the stacks is not regarded part of the task control block. They can be located anywhere as long as their location is known. They do belong to the task, however.

Viewed from afar, a task is just a piece of RAM holding a small set of important information.

Switching Tasks

To switch execution from one task to the next, the following things need to happen somehow:

- store the relevant bits of the current runtime in the task control block (stack pointers, mainly)
- look up the next task’s control block
- switch the userarea-pointer to that control block
- unfold the same bits, which were stored before giving up control, back into the runtime
- resume execution at the next instruction of the new task

So the problem is mainly an excercise in saving and restoring all relevant information.

[1] as opposed to *preemptive multitasking*

[2] Technically the TCB is located in the `user` area. However, the first 6 cells of the `user` area are used by the system itself (see section *Multitasker: The Gory Details*)

Recipe 1: Using the Multitasker

Problem

Simultaneous execution of several blocks of code (tasks) is desired on a single CPU.

Solution

Include the file `lib/multitask.frt` in your programm, define separate tasks as separate words. The start of everything needs a little extra code (see **`starttasker`**). This solution is working together with **`turnkey`**. **This recipe requires `amforth`-4.7 or later.**

Sample Program

The following example creates two tasks:

1. the command loop keeps running
2. increment a Counter **`N`**, write its value to **`PORTB`**. The intention is to make connected LEDs blink.

```
\ run_multitask  --  tested with amforth-4.7, atmega-32

$38 constant PORTB
$37 constant DDRB

include lib/multitask.frt       \ load the multitasker
: ms ( n -- ) 0 ?do pause 1ms loop ;  \ call pause on wait

variable N
: init
  $ff PORTB c!                  \ portB: all pins high
  $ff DDRB  c!                  \        all pins output
  0 N !
;
: run-demo                      \ --- task 2 ---
  begin
    N @ invert PORTB c!
    1 N +!
    &500 ms
  again
;
$20 $20 0 task: task_demo       \ create task, allot tcb + stack space
: start-demo
  task_demo tcb>tid activate
  \ words after this line are run in new task
  run-demo
;
: starttasker
  task_demo task-init           \ create TCB in RAM
  start-demo                    \ activate tasks job

  onlytask                      \ make cmd loop task-1
  task_demo tcb>tid alsotask    \ start task-2
  multi                         \ activate multitasking
;
: run-turnkey
```

```
37    applturnkey
38    init
39    starttasker
40  ;
41  ' run-turnkey is turnkey        \ make run-turnkey start on power up
```

When the program is started, LEDs connected to **PORTB** will blink. However, the prompt is presented as well and commands will be handled.

```
> run-turnkey
amforth 4.7 ATmega32
 ok
> tasks
149   running
309   running
Multitasker is running ok
> N @ .
199  ok
>
```

Discussion

The two tasks will happily run along provided, that both tasks call **pause** regularly. This call is built into the command loop already. It is possible to call **run-turnkey** as **turnkey**. The program will survive a power cycle, because **task:** stores the neccessary information in flash memory:

1. the address of the task control block
2. the start of the data stack (**sp0**)
3. the start of the return stack (**rp0**)

The sizes of the stacks are not expizitly stored. They can be inferred from the knowledge that all space is allocated as one chunk. However, `amforth` does not protect the stack from overflows. Underflows are generating an exception since `amforth` version 4.?. Exceeding the allocated stack space does cause unexpected crashes of your programm (see below at `task:`).

task-init prepares the task control block located in RAM. It erases any previous content, stores the addresses of the stacks, the top–of–stack address for the data stack, base, and the status of the task (sleeping). **start-demo** adds the calls to the tasks body into the TCB and stack space.

task: will use three entries from the stack.

1. additional size of the user area in this task. This space can be used to create **user**–variables, which belong to this task only.
2. size of the task's return stack
3. size of the task's data stack. Both stack sizes may be as small as `$20` bytes. However, programs exceeding a certain complexity may experience inexplicable crashes. If the program works *in the foreground* but not as a task, increasing the stack sizes may help.

Please note that calling **ms**, which in turn calls **1ms** will not produce accurate time intervalls any more, depending on how much time is spent in the other tasks.

One might argue that the startup sequence (**starttasker**) is way too long and should not be handled by the programmer. On the other hand, full control over the startup might be useful in unforeseen ways.

Multitasker: The Gory Details

`amforth` ships the file `lib/multitask.frt` featuring a multitasker based on code by Brad Eckert.

Task Control Block

The layout of the task control block is fixed. Technically it is located at the start of the so called *user area*. The first 6 entries (`status` . . . `handler`) are not intented for changes by the programmer. The next 6 entries (`base` . . . `/key`) are commonly changed by the programmer. If more space for user variables is desired, the user area needs to be increased specifically. When defining **`user`** variables, the offset of that variable from the start of the user area needs to be specified. It is the programmers duty to keep track of how many entries have been used.

Also as a consequence the `tid` of a task holds the start address of the user area for that task. Its value is copied into the *user pointer* upon task switch. The user pointer is fetched and stored with **`up@`** and **`up!`**, respectively (see definition of **`wake`** below).

```
\ TCB (task control block) structure, identical to user area
\ Offs_| _Name___ | __Description_____________________________ |
\   0  | status   | xt of word that resumes this task       | <-- UP
\   2  | follower | address of the next task's status       |
\   4  | RP0      | initial return stack pointer            |
\   6  | SP0      | initial data stack pointer              |
\   8  | sp       | -> top of stack                         |
\  10  | handler  | catch/throw handler                     |
\  12  | base     | numerical base                          |
\  14  | emit     | deferred io words                       |
\  16  | emit?    | .                                       |
\  18  | key      | .                                       |
\  20  | key?     | .                                       |
\  22  | /key     | .                                       |
```

Two offsets into the TCB are defined as **`user`** variables. They produce the address of TCB[0] and TCB[2] respectively, correctly using the current TCB's address.

```
decimal
0 user status
2 user follower
```

After that two **`noname:`** words are defined. These words will not have a header in the vocabulary, their execution tokens (**`xt`**s) are stored in the constants **`pass`** and **`wake`**. Their values will be stored in the status field (TCB[0]).

```
:noname ( 'status1 -- 'status2 )
    cell+ @ dup @ 1+ >r
;  constant pass

:noname  ( 'status1 -- )
        up! sp @ sp! rp!
; constant wake
```

Switching Multitasking on and off

To switch between tasks the deferred word **pause** is used. Normally, **pause** does nothing. Therefore turning multitasking off is simple:

```
43 \ stop multitasking
44 : single ( -- )
45     ['] noop is pause
46 ;
```

A new word **multitaskpause** is defined, which will switch from this to the next task.

```
48 \ switch to the next task in the list
49 : multitaskpause   ( -- )
50     rp@ sp@ sp ! follower @ dup @ 1+ >r
51 ;
52 \ start multitasking
53 : multi ( -- )
54     ['] multitaskpause is pause
55 ;
```

multitaskpause looks short and innocent, but a little explanation is called for:

rp@	**\ -- rp**	fetch the current return stack pointer
sp@	**\ -- rp sp**	fetch the current data stack pointer TOS
sp	**\ -- rp sp tcb[sp]**	get the addr of user variable to store TOS
!	**\ -- rp**	store, TCB[8] := TOS
follower	**\ -- rp tcb[2]**	get the address of TCB[2]
@	**\ -- rp tid'**	fetch it's content, tid of the next task
dup @	**\ -- rp tid' status'**	fetch status of the next task (xt)
1+	**\ -- rp tid' pfa**	xt **>body**
>r	**\ -- rp tid'**	put pfa of **pass** or **wake** on the returnstack

When multitaskpause exits, the interpreter finds the xt of wake or pass on the return stack and will continue execution there.

If status was **pass**, the next task is sleeping, so we need to look for the next next task:

	\ -- rp tid'	these are still on the stack
cell+	**\ -- rp tid'[2]**	point to follower
@	**\ -- rp tid''**	get the tid of the next next task
dup	**\ -- rp tid'' tid''**	
@	**\ -- rp tid'' status''**	fetch status of next next task (xt)
1+	**\ -- rp tid'' pfa**	xt **>body**
>r	**\ -- rp tid''**	put xt of next next tasks status on return stack

This is repeated until an awake task is found.

If status was **wake**, the next task should be running, so we need to unfold it:

	\ -- rp tid'	these are still on the stack
up!	**\ -- rp**	make user pointer point to tid'

This was the magic line. Now the stacks are different stacks! We left the old task's data stack behind with **rp** on top. Now we look at the new task's stack and find **rp'** of that task on top of it.

	\ -- rp'	
sp	**\ -- rp' tid'[sp]**	get addr of TOS location

@	**\ -- rp' sp'**	retrieve stack pointer of now current task
sp!	**\ -- rp'**	store it in (activate) stack pointer
rp!	**\ --**	store rp' of this task in current rp

Switching multitasking on is simply pointing **pause** to **multitaskpause**. The inner workings are far from obvious, but they have been proven to work.

Handling tasks

We need a few words to change the status of tasks:

```
: stop         ( -- )     pass status ! pause ; \ sleep current task
: task-sleep   ( tid -- ) pass swap ! ;         \ sleep another task
: task-awake   ( tid -- ) wake swap ! ;         \ wake another task
```

A little more tricky is setting up a piece of code to be run in a task. **activate** will be used in a snippet similar to this.

```
: run-demo ( interesting work here ... ) ;
$20 $20 0 task: task_demo       \ create task, allot tcb + stack space
: start-demo
  task_demo tcb>tid activate
  \ words after this line are run in new task
  run-demo
;
```

activate will store the xt of **run-demo** on the return stack belonging to the TCB. It will also save the address of top of return stack on top of the data stack belonging to the same TCB, and the address of TOS in the field TCB[sp]. This particular order of information is expected by **wake**.

```
: cell- negate cell+ negate ;
\ continue the code as a task in a predefined tcb
: activate ( tid -- )
   dup    6 + @ cell-
   over   4 + @ cell- ( -- tid sp rp )     \ point to RP0 SP0
   r> over 1+ !       ( save entry at rp ) \ skip all after ACTIVATE
      over  !         (  save rp at sp )   \ save stack context for WAKE
   over 8 + !         ( save sp in tos )
   task-awake
;
```

onlytask initializes the linked list with the current task only. It copies the tid of the current task into the field TCB[follower] to create a circular list.

```
\ initialize the multitasker with the current task only
: onlytask ( -- )
    wake status !   \ own status is running
    up@  follower ! \ point to myself
;
```

alsotask links a new task given by its **tid** into the list behind the current task.

```
: alsotask      ( tid -- )
   [’] pause defer@ >r \ stop multitasking
   single
   follower @   ( -- tid f)
   over         ( -- tid f tid )
   follower !   ( -- tid f )
   swap cell+   ( -- f tid-f )
   !
   r> is pause  \ restore multitasking
;
```

And then there is **tasks** to print the **tid** of every task in the list and its state to the serial console. It will also report, whether the multitasker is switched on or not (see page 3). If you uncomment the three commented lines, then the values of top–of–stack and start–of–stack for the data and return stacks are also printed out. This might be useful for debugging.

```
: tasks ( -- )
    status ( -- tid ) \ starting value
    dup
    begin      ( -- tid1 ctid )
        dup u. ( -- tid1 ctid )
        dup @  ( -- tid1 ctid status )
            dup
            wake = if ."   running" drop else
            pass = if ."  sleeping" else
                  abort"   unknown" then
        then
\        dup 4 + @ ."   rp0=" dup u. cell- @ ."  TOR=" u.
\        dup 6 + @ ."   sp0=" dup u. cell- @ ."  TOS=" u.
\        dup 8 + @ ."    sp=" u.
        cr
        cell+ @ ( -- tid1 next-tid )
        over  over =     ( -- f flag)
    until
    drop drop
    ." Multitasker is "
    [’] pause defer@ [’] noop = if ." not " then
    ." running"
;
```

Creating a TCB

So there is only one thing left to do, namely create space for a TCB and the stacks.

```
: task: ( C: dstacksize rstacksize add.usersize "name" -- )
        ( R: -- addr )
  create
    here ,                      \ store address of TCB
    ( add.usersize ) &24 + allot \ default user area size
                                \ allocate stacks
    ( rstacksize ) allot here , \ store sp0
    ( dstacksize ) allot here , \ store rp0

```

```
124     1 allot                    \ keep here away, amforth specific
125   does>
126                                  \ leave flash addr on stack
127 ;
128 : tcb>tid  ( f -- tid )      @i ;
129 : tcb>sp0  ( f -- sp0 )  1+  @i ;
130 : tcb>rp0  ( f -- rp0 )  2 + @i ;
131 : tcb>size ( f -- size )
132   dup tcb>tid swap tcb>rp0 1+ swap -
133 ;
```

`task:` allots memory for the task control block and its associated stacks. The sizes of the stacks are taken from the data stack. The start of the data stack (SP0) is stored in TCB[6], the start of the return stack (RP0) is stored in TCB[4]. Then new tid is moved from the return stack to the data stack. The task is marked as sleeping and one more byte is alloted *to keep* `here` *out of the way*. This is an implementation feature of `amforth`. Also please note that stacks are growing downwards.

`task-init` initializes a TCB and copies the information stored in flash into their correct locations.

```
134 : task-init ( f -- )
135   dup tcb>tid  over tcb>size  0 fill \ clear RAM for tcb and stacks
136   \ fixme: possibly use init-user?
137   dup tcb>sp0 over tcb>tid &6 + !       \ store sp0    in tid[6]
138   dup tcb>sp0 cell- over tcb>tid &8 + ! \ store sp0--  in tid[8], tos
139   dup tcb>rp0 over tcb>tid &4 + !       \ store rp0    in tid[4]
140       &10  over tcb>tid &12 + !         \ store base   in tid[12]
141       tcb>tid task-sleep                \ store 'pass' in tid[0]
142 ;
```

Versions of `lib/multitask.frt` prior to `amforth`-4.7 are broken in that there is no permanent storage as described above. These versions of the multitasker work, but they do not survive a power cycle.