Multitasking

das Paket MTASK erlaubt kooperatives Multitasking.

Laden mit

```
1 GET MTASK
1 5 THRU
```

Glossar

```
(PAUSE)   ( -- )
RESTART   ( -- )
6 CONSTANT INT#
LOCAL   ( base adr -- adr' )
@LINK   ( -- adr )
!LINK   ( adr -- )
SLEEP   ( adr -- )
WAKE    ( adr -- )
STOP    ( -- )
SINGLE   ( -- )
MULTI   ( -- )
TASK:  ( size -- )
SET-TASK   ( IP Task -- )
ACTIVATE   ( Task -- ) Activate a Task
BACKGROUND:   ( -- )   Create a Background task
```

Literatur

- https://www.bradrodriguez.com/papers/mtasking.html
- C.H. Thing „Inside F83“ http://forth.org/OffeteStore/1003_InsideF83.pdf
- R. Zech, Forth 83
- http://www.forth.org/fd/FD-V05N4.pdf

aus L/P-F83-Doku:

```
Multitasking low level
(PAUSE)   (S -- )
    Puts a task to sleep by storing the IP and the RP on the
    parameter stack.  It then saves the pointer to the
    parameter stack in the user area and jumps to the code
    pointed at by USER+3, switching tasks.
RESTART     (S -- )
    Sets the user pointer to point to a new user area and
    restores the parameter stack that was previously saved
    in the USER area.  Then pops the RP and IP off of the
    stack and resumes execution.   The inverse of PAUSE.
```

```
Initialize current User area to a single task.

Manipulate Tasks
LOCAL  Map a User variable from the current task to another task
@LINK  Return a pointer the the next tasks entry point
!LINK  Set the link field of the current task (perhaps relative)
SLEEP  makes a task pause indefinitely.
WAKE  lets a task start again.
STOP  makes a task pause indefinitely.
SINGLE  removes the multi-tasker's scheduler/dispatcher loop.
MULTI
   installs the multi-tasker's scheduler/dispatcher loop.
   By patching the appropriate INT vector and enabling PAUSE.

MULTI starts the multi-tasker loop running. SINGLE stops it.
Then type XXX WAKE to start the XXX task.
To put the XXX on hold, use  XXX SLEEP
To restart it, use XXX WAKE

In general, executing the name of a task leaves the address of
its user area on the stack. Words like sleep and wake use that
address.

\ Activate a Task
TASK:  Name, initialize, and allocate a new task.
  Copy the USER Area.  I point to where he pointed.
  He points to me.
  Set initial stack pointers.
  Set dictionary pointer.
  Make task ready to execute. Allocate task in host dictionary.
SET-TASK  assigns an existing task to the code at ip.
  Get top of stack of the task to be used.
  Put IP and RP values on its stack.
  Set its saved stack pointer.

ACTIVATE  assigns an existing task to the following code,
  and makes it ready to execute.

\ Create a Background Task
BACKGROUND:
  Create a new task of default size. Initialize it to execute
  the following code.

Examples:
 The task named counter executes an infinite loop, so STOP is not
 required. Note that you MUST use PAUSE, or no other tasks will
 be executed. PAUSE is built in to all words which do I/O, so
 tasks which do I/O ( like spooler ) do not need to use PAUSE
 explicitly.
```

Beispiel

```
VARIABLE COUNTS
BACKGROUND: COUNTER  BEGIN
    PAUSE 1 COUNTS +!  AGAIN ;
COUNTER WAKE  MULTI

COUNTS ?
COUNTS ?
COUNTS ?
SINGLE
```

Funktionsweise

Das Multitasking-System basiert auf dem des Laxen/Perry-F83 und wurde geringfügig ans FG FORTH83 angepasst. Die Beschreibung zur Funktionsweise und Nutzung des Multitasking-Systems entspricht daher der Dokumentation in C.H. Thing „Inside F83“ (http://forth.org/OffeteStore/1003_InsideF83.pdf).

Abweichungen:

- Beim Z80 wird statt INT80 der RST30 (RST 6) genutzt.
- instruction pointer ist Register BC
- return stack pointer ist Register IY

Inside F83, Second Edition, CHAPTER 23. MULTITASKER

The source code of the multitasker is in CPU8086.BLK, screens 22-23, and in UTILITY.BLK, screens 52-54.

23.1. MULTITASKING

Multitasking is the technique to use one computer to do several things at the same time. Most of the microcomputers run rather inefficiently in the single user, interactive mode, because the computer wastes most of its time in waiting the user to type in commands. This waiting time can be utilized to perform useful work, like printing a long file, keeping a timer clock, watching over the heater or the air conditioner and other instruments, etc. If properly scheduled, all these activities can be handled by a single computer, allocating each task sufficient time to do its work and still satisfies the programming needs of the user.

Most mainframe computers and minicomputer operating systems have the multitasking function and can support many users and tasks to run at the same time, using rather complicated hardware and software to schedule and run the tasks, and also manage a host of peripheral devices like disk drives, tape drives, printers, plotters, etc. Scheduling and resource allocation are big headaches in these operating systems, contributing a fair share to the complexity in the operating system.

People generally conceive FORTH as a toy language only suitable for single user microcomputers. This is probably due to the limited capability presented in the public domain fig-FORTH model, which has become the most widely distributed FORTH dialect. However, in many more expensive commercial

FORTH implementations, especially those developed by Charles Moore himself and later marketed by FORTH, Inc. under the trade name poly-FORTH, multitasking was a standard component using the very simple but effective round-robin scheduling technique.

F83 also includes the elementary operators to implement multi- tasking. The basic system design takes the task switching into consideration, so that tasks can be easily added to the system when needed. Task switching is very fast because of the brevity of code involved. Here we will go through the entire system to describe the multitasker in great details.

23.2. USER VARIABLES AND THE USER AREA

Special words managing multitasking or multiuser FORTH system are collected in a special vocabulary USER. Some of them have the same names but different functions as other regular FORTH words. They have to be used with care. They allocate space for user variables in the user area which is a unique memory area for every task in the system.

```
VARIABLE #USER      Count of the number of user variables allocated in a
user area.
VOCABULARY USER     Declare the user vocabulary.
USER DEFINITIONS        Put all subsequent words in the USER vocabulary.
```

```
: ALLOT         ( n --- )     Allocate space in the user area for a task.
    #USER +!            Move the user area pointer forward for n bytes.
    ;

: CREATE        ( --- )     A special header builder for user variables.
    CREATE                Build a regular header.
    #USER @ ,            Compile the current user area pointer in the
parameter field,
                      to be used as the offset of the user variable from
origin of
                       the user area.
    ;USES                Get the code field address of the new user
variable.
    DOUSER-VARIABLE ,        Patch the code field using DOUSER-VARIABLE as
the runtime interpreter for user variables.

: VARIABLE      ( --- )     New defining word for user variables.
    CREATE               This is the newly defined CREATE.
    2 ALLOT               Allocate two bytes in the user area, not on top
of dictionary.    ;

: DEFER         ( --- )        New defining word for deferred words in the
user area.
    VARIABLE            Create a new user variable.
    ;USES                Patch code field with
    DOUSER-DEFER ,            address of the runtime routine DOUSER-DEFER.
```

When tasks are switched, the environment of the task currently under execution must be preserved before the task is put to sleep, so that when the task is woke up the next time around it will be able to pick the execution sequence where was left off and continue the task until finished or put to sleep

again. What defines the environment of a task is a set of parameters stored in a set of 'user variables' and an area where the data stack and the return stack used by the task are allocated. Each task must have its own copy of these user variables and stacks. As these essential information are stored independently for each task, task switching becomes very easy because only a minimal amount of information has to be preserved explicitly during task switching.

Following is the list of user variables needed in every task:

TABLE 23.1. USER VARIABLES

```
    TOS                Top of data stack.
    ENTRY                Entry point to be jumped to when the task is
activated.
    LINK                Point to next task in the round- robin circle.
    SP0                Origin of the data stack.
    RP0                Origin of the return stack.
    DP                Top of dictionary.
    #OUT                Number of characters emitted.
    #LINE                Number of lines typed.
    OFFSET                Block offset from block 0 in the current file.
    BASE                Numeric base for I/O conversion.
    HLD                Point to the last character converted in the PAD
buffer.
    FILE                Point to FCB (file control block) of currently
opened file.
    IN-FILE                Point to the FCB of the input file.
    PRINTING            A flag.  True if printer is active.
    EMIT                Send a character to the output device currently
active.
```

One should note that these user variables have their names in the main dictionary while the parameters are stored in the user area. Only one copy of the names needs to be defined. Every user or task will have its own copy of the user variables. The functions of these user variables will become apparent in the following sections.

23.3. PAUSE AND RESTART

PAUSE and RESTART are the two most crucial operators to effect the task switching process. (PAUSE) stops the current task and passes control on the CPU to the next task in the round-robin loop. It saves the necessary information in the user area so that the task can continue the next time it gains the control of the CPU.

```
CODE  (PAUSE)            Stop executing the current task and pass control to
the next
                task.
    IP  PUSH        Save the Interpretive pointer on the data stack.
    RP  PUSH        Save the return stack pointer.
    UP #) BX MOV        Fetch the user area pointer into BX register.
    SP 0 [BX] MOV        Save the data stack pointer in user variable TOS.
    BX INC   BX INC
    BX INC   BX INC     Increment BX and point it at the user variable LINK.
```

```
    0 [BX] BX ADD        Calculate the address of the user area from the
offset in
                LINK.
    BX INC   BX INC        Now point to the ENTRY user variable of the next
task.
    BX JMP            Execute the next task.
    END-CODE

CODE  RESTART           The reverse of (PAUSE).  Retrieve the stored
information
                and start executing the task left asleep during the last
pass.
    -4 # AX MOV        Store -4 in the AX register for later use.
    BX POP            Pop the LINK address of the next task to be waked up.
    AX BX ADD        Point BX to the TOS user variable.
    BX UP #) MOV        Copy this address into UP as the user pointer for
the next
                task.
    AX POP   AX POP        Discard two items from the stack. They were
pushed on the
                stack by the interrupt routine waking up this task.
    0 [BX] SP MOV        Restore the data stack pointer.
    RP POP            Restore the return stack pointer.
    IP POP            Restore the interpretive pointer.
    NEXT            IP contains the pointer pointing to the next word to be
                executed in this task.  NEXT will continue the execution
left
                off last time.
    END-CODE

CODE PAUSE          A dummy word allowing multitasker to be switch on and
                off.
    NEXT            In the single user mode, PAUSE returns immediately
                without doing anything.
    END-CODE        However, its code field will be patched so that the word
                (PAUSE) will be invoked in the multitasking mode of
                operation.
```

A few operators are defined to manipulate data stored in the user variables to control the tasks:

```
HEX 80 CONSTANT INT#        8086 software interrupt number, used to wake up
a task.
```

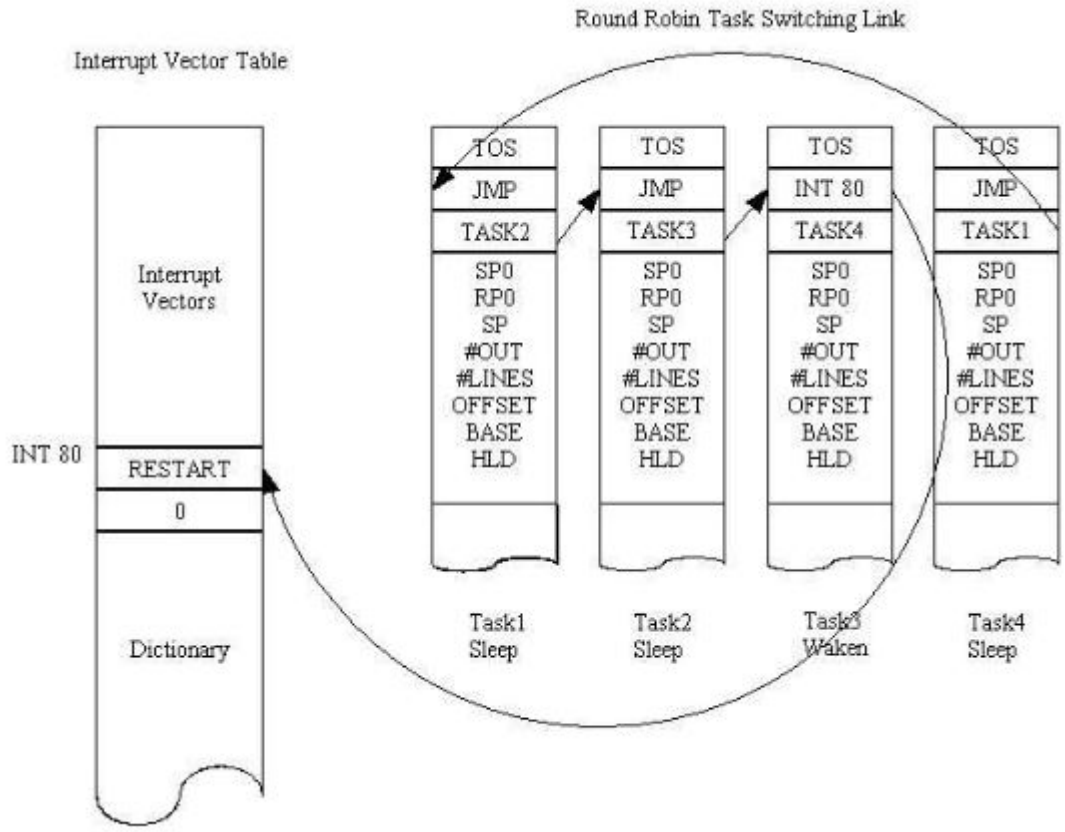


Figure 23.1 The round-robin task scheduler.

```
: LOCAL ( base addr --- addr1 )     Get the address of a user variable in
another task's
                    user area.
    UP @                The origin of current user area.
    -                Offset of current user variable from its origin.
    +                Add offset to the origin of the other user area (base),
                    returning the address of the same user variable in the
other
                    task.
    ;

: @LINK     ( --- addr )     Return a pointer to the entry point in the next
task.
    LINK            Address of LINK in this task.
    DUP @ +            Get the ENTRY in the next task.
    2+            LINK field in the next task.
    ;

: !LINK     ( addr --- )     Set the LINK field of the current task, given
the origin of the
                user area of the next task.
    LINK -             The relative distance from LINK to the other task.
    2+            Relative distance from this LINK to the other LINK.
    LINK !             Store it in the current LINK.
    ;
```

```
: SLEEP     ( addr --- )     Make the addressed task pause indefinitely.
    E990            90 is NOP and E9 is JMP in 8086 machine instruction.
                JMP (link) passes CPU to the next task whose address is in
the
                LINK field right after the ENTRY field.
    SWAP            Get the task address to top.
    ENTRY LOCAL        Get the address of the ENTRY field in the target
task.
    !            Store the sleep code in its ENTRY field and force that task
to
                pass control immediately to its next task.
    ;

: WAKE      ( addr --- )     Wake up a task so that it will execute in its
next turn.
    80CD            Machine instruction INT 80H which wakes up this task by
a
                software interrupt with vector number 80H.
    SWAP            Get the task's user area address.
    ENTRY LOCAL !        Store that wake code in the ENTRY field of the
target task
                and make it an active member in the round-robin loop.
    ;

: STOP      ( --- )     Make the current task pause indefinitely.
    UP @            Get the address of the current task.
    SLEEP            Put it to sleep.
    PAUSE            Stop right now.
    ;
```

23.4. THE MULTITASKER

The FORTH multitasking is implemented using a round-robin scheduler and dispatcher technique. All tasks are linked into a loop. A task must use the PAUSE command explicitly to relinquish CPU control to the next task. If a task does not need the CPU service, it will pass the CPU control directly to the next task. When a task needs CPU, it will put a wake code in the ENTRY field in its user area. Next time when the control of CPU is passed to it, it will be able to grab the CPU and restart or continue on the execution left off last time. Each task therefore must include PAUSEs at suitable intervals to let other tasks have a piece of the action. Otherwise a task can hold onto the CPU indefinitely and no other task can use the CPU. The cooperative nature of this scheme thus requires that each task be designed to pause regularly. The advantage is that each task can stop and restart at known points and the code to do the multitasking is simple and fast.

```
CODE MULTI  ( --- )     Install the multitasker's schedule/ dispatcher loop
by
                patching the appropriate interrupt vector and enabling
                PAUSE.
    ' (PAUSE) @ # BX MOV      Copy the contents of code field of (PAUSE) to
BX register.
                It is the starting address of the (PAUSE) code routine.
    BX ' PAUSE #) MOV    Patch the code field of PAUSE with code of (PAUSE)
```

```
so
                that the task will pause at PAUSE.
    ' RESTART @ # BX MOV    Get the RESTART code address.
    DS AX MOV
    AX PUSH            Save DS register on data stack.
    AX AX SUB        Clear AX register.
    AX DS MOV        Clear DS register.
    CS AX MOV        Copy code segment into AX.
    AX INT# 4 * 2+ #) MOV     Store the code segment in the second cell of
the interrupt
                vector for this task.
    BX INT# 4 * #) MOV    Store the RESTART address in the first cell of the
interrupt
                vector. Hereafter, INT 80H will activate this task.
    AX POP
    AX DS MOV        Restore the data segment register.
    NEXT
    END-CODE

: SINGLE    ( --- )     Remove the multitasker's scheduler/dispatcher loop.
    ['] PAUSE >BODY        Get the parameter field address of PAUSE, which
points to a
                simple NEXT routine.
    ['] PAUSE !        Restore the code field of PAUSE so that PAUSE will
return
                immediately rather than go through (PAUSE) for the
                multitasker.
    ;
```

23.5. TASK DEFINITION

A task must be first defined as a word in the dictionary. At the same time, user area and space for two stacks must also be allocated for the task. The new task must then be installed in the round-robin loop. When functions must be performed by this task, the word to be executed by this task must be passed to the multitasker and the task must be waken. The tools to create and activate new tasks are discussed in this section.

```
: TASK:     ( size --- )     Create a new named task and initialize its user
variables.
    CREATE            Build an header for the task.
    TOS            Address of the user area for the current task.
    HERE            The user area of the new task.
    #USER @            Size of the user area.
    CMOVE            Copy the current user variables into the new user area
for the
                new task.
    @LINK            New user area.
    UP @ -ROT        Save the current user area pointer at the bottom of the
data
                stack.
    HERE UP !        Put new user area address in the user area pointer UP.
```

```
    !LINK            Store address of current user area in the LINK field of
the
                 new task.
    DUP HERE +        Address of the space after the new user area and
dictionary
                 space.
    DUP RP0 !        Initialize the return stack pointer of the new task.
    100 - SP0 !        Initialize the data stack pointer of the new task.
    SWAP UP !        Restore the user area pointer to the current user area.
    HERE ENTRY LOCAL !LINK  Store the address of the new task in the LINK
field
                 of the current task.
    HERE #USER @ +        Starting address of the dictionary for the new
task.
    HERE DP LOCAL !        Initialize the dictionary pointer of the new
task.
    HERE SLEEP        Put the new task to sleep first.
    ALLOT            Allocate space for the new task according to the size
                 parameter given on the stack initially.
    ;

: SET-TASK  ( ip task --- )     Assign an existing task to execute the code
pointed to by ip
                     on stack.
    DUP SP0 LOCAL @            Get the top of data stack of the task to be
used.
    2-                Decrement stack pointer preparing a push.
    ROT OVER !            Push ip on data stack.
    2-                Prepare another push.
    OVER RP0 LOCAL @        Get the new tasks return stack pointer.
    OVER !                Push it on the data stack.
    SWAP TOS LOCAL !        Store the data stack pointer in the TOS field of
the new task.
                     These actions simulate (PAUSE) in a way that when the
new
                     task is waken the code pointed to by ip will be
executed.
    ;

: ACTIVATE  ( task --- )         Assign the invoked task to execute the
following code, and
                     wake the task making it ready to execute these code.
    R>                Address of the next code to be executed.
    OVER SET-TASK            Assign this code to the task.
    WAKE                wake the task.
    ;
```

23.6. BACKGROUND TASKS

A background task is some functions the computer does while still allowing the user to use the computer interactively doing programming or execute other programs. F83 allows the user to create

many of such tasks to run concurrently, using the task defining and control words. Here are some examples illustrating the command sequence to create background tasks and activate them.

```
: BACKGROUND:   ( --- )         Create a new task with 400 bytes of
dictionary space.
                    Initialize this task to execute the following code.
    400 TASK:            Define a new task with the name following.
    HERE                IP for code to be compiled so that it can be
executed by the
                    new task.
    @LINK 2-            Address of the new task.
    SET-TASK            Initialize the new task.
    !CSP                Compiler error checking initial- ization.
    ]                Turn on the compiler to compile following code to be
                    executed by the new task.
    ;

BACKGROUND:  SPOOLER    ( --- )     Create a new task named spooler which
will print the
                    current file.
    1                First screen to be listed.
    CAPACITY            Last screen in the current file.
    SHOW                List the entire file.
    STOP                Stop the task here.  STOP is needed at the end of a
task.
    ;
```

Executing SPOOLER will print out the entire current file on the printer while the user can still user the terminal for normal activities. Low level input/output codes in SHOW contain enough PAUSE commands to alternate the spooling task with the terminal task to allow the user seemingly full control over the computer while it is also printing a long file. SPOOLER can also be assigned a different function by the following definition:

```
: SPOOL-THIS    ( --- )         Assign new functions to an existing task.
    SPOOLER ACTIVATE        Force the task SPOOLER to execute the following
code:
    3 15
    [ SHADOW ]  SHOW        Invoke the special SHOW command in the SHADOW
                    vocabulary to print shadow screens along the source
                    screens.
    STOP                Terminate the task.
    ;
```

Another example is to use the computer to keep a counter which keeps the number of circles the multitasker running around the round-robin loop:

```
VARIABLE COUNTS

BACKGROUND:  COUNTER        Define a new background task.
    BEGIN            Beginning of an infinite loop.
        PAUSE        Allow other tasks to run once.
```

```
        1 COUNTS +!    Increment the counter.
    AGAIN    ;
```

The task COUNTER executes an infinite loop, so STOP is not required at the end of the definition. However, note that you must use PAUSE in the loop, or no other task will be executed. PAUSE is built in all the input/output words, so that tasks which do I/O like SPOOLER do not have to use PAUSE explicitly.

From:
https://hc-ddr.hucki.net/wiki/ - **Homecomputer DDR**

Permanent link:
https://hc-ddr.hucki.net/wiki/doku.php/forth/fgforth/multitasking?rev=1605508287

Last update: **2020/11/16 06:31**