FORTH MULTITASKING IN A NUTSHELL

Brad Rodriguez
T-Recursive Technology
September 1992

INTRODUCTION

1. What is multitasking?

Multitasking is the ability to run several independent programs in a single CPU, apparently simultaneously. Of course, a single CPU can only run one instruction at a time. The illusion is created by switching the CPU very quickly -- hundreds of times a second -- among several different programs. Each of these programs is called a "task," hence the name "multitasking."

A popular fallacy is that you need a big CPU, like a 68000, in order to do multitasking. This is simply not true. Any CPU can multitask; I've done it on Z80s and single-chip Z8s.

Another fallacy is that you need a real-time clock interrupt to "time slice" the tasks. While some multitaskers do indeed work this way, the Forth multitasker does not. You can multitask without interrupts, and with no special hardware support at all!

It is important to distinguish between multitasking and multiuser. Multiuser systems are those that support multiple terminals, and give several people the illusion of working each on his own computer. Multitasking is simpler: there is only one user, but he can have several things running simultaneously. Unix is multiuser. Amigas, Macintoshes, and Windows PCs are multitasking.

Occasionally, in a computer science text, you will see multitasking called multiprocessing. I prefer to avoid this usage. To me, multiprocessing means several CPUs operating in parallel. If it's on a single CPU, I call it multitasking.

2. When should multitasking be used?

There are several situations -- some more obvious than others -- which can benefit from multitasking.

2.1 Parallel operations. Some computer applications just naturally have several things running at once. An embedded process controller probably has to keep the control loops running, even while the operator is typing commands on the keypad. Network communications generally have to be maintained while other things are going on. And usually you'd like to be able to do something else while the printer is grinding out a long listing. All of these are candidates for multitasking.

2.2 Using idle time. Many computer programs spend a lot of time waiting -- maybe for a keypress, an external event, or a data transfer to complete. Sitting in a wait loop is a waste of good CPU time! Multitasking allows the CPU to be doing something else while waiting for an event.

2.3 Coroutines. Sometimes you need to break out of a program "in the middle." For example, I recently wrote a command processor which needed to stop and wait for a keypress at a deeply nested level. Multitasking allows a program to be suspended at any point, and later resumed, with no special effort on the part of the programmer.

2.4 Clarity and ease of programming. Often a program is easier to write, and its logic is more obvious, if you use the facilities that a multitasker gives you. Sometimes an algorithm is best expressed procedurally, in terms of waiting for an event -- even when you know the CPU can't just sit there and wait. I could have written the command processor using a huge state table, and entered the routine "at the top" on every keypress -- but it would have been much larger, and impossible to read and maintain. Multitasking adds **WAIT** and **PAUSE** to your "programming toolbox." You just use them naturally, and the multitasker takes care of the details.

HOW FORTH MULTITASKING WORKS "INSIDE"

3. What each task requires

What is needed to support multiple Forth tasks?

3.1 Separate programs. In most multitasking applications, each task will run a different program. In single-user Forth systems, these need not be in separate dictionaries; they can simply be Forth words with different names.

In multiuser Forth systems, several people may be adding definitions at the same time. So, each user needs some RAM for a private dictionary. F83 does this. (How multiple dictionaries are managed, and linked to the "main" Forth dictionary, are beyond the scope of this article.)

3.2 Re-entrant kernel code. We'd really prefer not to have a separate copy of **DUP** or **FIND** for each task! Code which can be shared by several tasks simultaneously is called "re-entrant." Among other things, re-entrant code can't leave important data in global variables. Fortunately, Forth code, with its use of stacks, tends to be naturally re-entrant, and the "cooperative" multitasker relaxes some of the restrictions on temporary variables. Most Forth kernels are fully re-entrant.

3.3 Private stacks. Obviously, each independent Forth program will need its own parameter and return stacks. So, each task has some RAM for its stacks, and also has its own parameter and return Stack Pointers.

3.4 Private "user areas." Some things simply must be kept in variables, and yet will have different values for each task. The "bottom" stack addresses **S0** and **R0** are two examples; the **BASE** for number conversion is another. So, each task needs a private RAM area for variables, called the user area. Different Forth kernels may keep from six to several dozen variables in the user area. Most Forths provide a "user variable" which, instead of returning an absolute address, indexes into the user area of whichever task is currently running.

The user area is essential to hold certain task control data: among other things, the "saved" stack pointer, and the link to the next task. Both of these will be discussed shortly.

3.5 Private buffers. Certain buffers must exist privately for each task. One example is the **PAD** buffer which is used for numeric output. If two tasks tried to display a number at the same time, using a single **PAD**, nonsense would result! Such buffers may be kept in the user area, or (in multiuser systems) in the private dictionary.

Multiuser systems also need separate Terminal Input Buffers. Usually, though, a single set of disk buffers is shared by all the tasks.

4. Switching tasks

Most Forth systems use the simplest of multitasking schemes: the "round-robin, cooperative" multitasker. Round-robin means that each task takes its turn at the CPU, one at a time, in a fixed sequence -- a big loop of tasks. Cooperative means that each task has the CPU as long as it wants, and releases the CPU only when it's ready -- nothing will "grab" the CPU away from a task.

Figure 1 shows the RAM allocation for a three-task Forth system. The private areas described above are usually grouped together for convenience. The dictionary and the disk buffers are common to all of the tasks.

Switching from one task to another -- that is, from one program to another -- requires three steps.

a) Save the state of the current program. Everything necessary to restore this program -- to exactly the point where it was suspended -- must be saved. This is called the "task context," and may include the CPU's program counter, flags, and registers, as well as data in RAM.

In Forth, most of the task context is on the (private) parameter and return stacks, and so is safe from alteration. But there are four crucial values which are so frequently used that they are usually kept in CPU registers:

SP - the parameter Stack Pointer
RP - the Return stack Pointer
IP - the Interpreter Pointer
UP - the User Pointer (base address of the user area)

These four registers must be saved and restored when the task is switched. (It turns out that we don't need to save the CPU's Program Counter, since we never change tasks in the middle of a **CODE** word.)

b) Select the next task to run. This may be done in fixed rotation, or according to some priority scheme.

c) Restart the new task according to its saved context. This will resume execution of the new task at the point where it was last suspended.

5. The task switch in F83

The Forth word which switches tasks is traditionally called **PAUSE**. It must do the following:

a) Save the IP, RP, and SP. Typically two of these will be pushed on a stack, and that stack's pointer will then be saved in the user area. We don't need to save UP; you'll see why in a moment.

b) Get the address of the next task's user area. This is what the LINK variable in the user area is for: it contains the address of the next task in the round-robin sequence. (The tasks are thus chained together in a linked list.) Note that this link gives you the UP (user area address) for the new task! This is why UP doesn't need to be explicitly saved.

c) Restore the IP, RP, and SP of the new task.

d) Continue Forth execution!

The time it takes to do this is called the "context switch time," and is an important figure of merit for multitasking systems. Since Forth systems only have to save three CPU registers, their context switching times are quite fast -- under 10 microseconds on some 8-bit CPUs!

F83 on the IBM PC uses some tricks to improve performance. (Refer to Figure 1.) In F83, step (b) is performed by jumping into the next task's user area. In this area is a code fragment (INT 80h) which executes a **RESTART** routine. This **RESTART** routine does the step (c) described above.

Why does F83 do this? A task which is not ready to run can be "put to sleep" by changing the INT 80h to a JMP instruction. Then, when **PAUSE** jumps to this task, it immediately jumps to the next task. The sleeping task is skipped in only one machine instruction!

You can see that if all the tasks had JMP instructions, the round-robin loop would be simply a loop of jumps. Of course, the CPU would then be stuck in an infinite loop! Usually, several tasks will be "awake" with INT 80h instructions.

F83's **PAUSE** and **RESTART** are coded as follows:

```
CODE (PAUSE)
     IP PUSH                        save IP on parameter stack
     RP PUSH                        save RP on parameter stack
     UP #) BX MOV  SP 0 [BX] MOV    save SP in user area
     BX INC  BX INC  BX INC  BX INC  calculate address of next
     0 [BX] BX ADD  BX INC  BX INC    task from (relative) LINK,
     BX JMP  C;                       then jump to that task

CODE RESTART            entered from an INT 80h instruction
     -4 # AX MOV
     BX POP                  get return adrs saved by INT 80h
     AX BX ADD               adjust it to the start of user area
     BX UP #) MOV            make this the current UP
     AX POP  AX POP          clean up INT 80h leftovers
     STI
     0 [BX] SP MOV           restore SP from user area
     CLI
     RP POP                  restore RP from parameter stack
     IP POP                  restore IP from parameter stack
     NEXT  C;                continue Forth execution at new IP
```

USING A FORTH MULTITASKER

6. Creating a task

All Forth systems start with one task. (There's always at least one program running!) New tasks can then be added to the system in two phases, which I call creation and activation.

Creating a new task involves two steps:

a) Reserve RAM for the task. Space must be allocated for its two stacks and its user area. In multiuser systems, a private dictionary must be reserved as well.

b) Link the task into the round-robin list. This is a simple linked-list insertion using the LINK field.

These steps are performed only once. Obviously, reserving RAM twice for the same task is a pointless waste of RAM...and can lead to confusion if other tasks need to know where this task is located. Linking the task into the list twice is more subtle, and more insidious: usually it ends up destroying the round-robin loop!

In F83 on the IBM PC, steps (a) and (b) are performed by the word **TASK:**. This word expects, on the stack, the number of bytes to reserve for the new task. 256 bytes are reserved for the return stack, and the rest is divided between the user variables, the private dictionary, and the parameter stack. Thus:

HEX 400 TASK: SCREEN-CLOCK

defines a task and allocates a total of 1024 bytes to it. The newly defined word **SCREEN-CLOCK** will return the base address of this 1024-byte area (the base address of the user area).

7. Activating a task

Once the task has been created, it can be "activated" any number of times. This involves two steps:

c) Initialize the task context. Several user variables must be initialized, and in F83 the INT 80h instruction must be inserted in the user area. Also, the initial values for the SP and RP must be stored in the stacks or user area in such a way that they will be correctly loaded into the CPU registers when this task is "resumed" for the first time.

d) Specify the code to be executed by that task. The initial value for the IP must be stored in the stacks or user area, too. When the task is "resumed" for the first time, this will be where execution begins. It must point to a fragment of high-level Forth code.

(Some Forths may only perform step (c) once, in which case it may be done as part of task creation.)

After the task has been activated, it will lie dormant until its turn in the round-robin loop. Then it will begin executing the Forth code specified in step (d).

In F83, the RP, SP, and IP are initialized by the word **ACTIVATE**. F83's **ACTIVATE** must be used within a Forth word -- it cannot be used interpretively. It expects the address of the task on the stack, and is immediately followed by the high-level code the new task is to run. For example:

```
: START-CLOCK   SCREEN-CLOCK ACTIVATE BEGIN .CLOCK PAUSE AGAIN ;
<----------------------------------> <---------------------->
     code performed by "main" task       code performed by
                                         SCREEN-CLOCK task
```

START-CLOCK is executed by some other Forth task -- typically the "main" (initial) task. It then "activates" the **SCREEN-CLOCK** task to perform the code fragment **BEGIN .CLOCK PAUSE AGAIN**. The main task exits this word immediately after **ACTIVATE**.

8. Other task control in F83

The functions described above -- **PAUSE**, **TASK:**, and **ACTIVATE** -- are all that you need to establish a multitasking Forth system. But F83 provides some additional words for convenience:

taskaddr SLEEP puts a task to sleep, by inserting the JMP instruction in the user area.

taskaddr WAKE awakens a task, by inserting the INT 80h instruction.

STOP just puts the running task to sleep, and then switches to the next task. This is equivalent to **my-task SLEEP PAUSE**.

MULTI enables the multitasker.

SINGLE disables the multitasker, by changing the action of **PAUSE** to a "no-op." (**PAUSE** is a DEFERred word for just this purpose.) Whichever task does **SINGLE** will keep control of the CPU.

APPLYING THE FORTH MULTITASKER

9. The PAUSE that refreshes: running programs in parallel (2.1)

The simplest use of a multitasker is to have several programs running in parallel. This requires only that every program have its own task, and that every program obey two rules:

a) Every task must **PAUSE** periodically! This is how the task voluntarily surrenders the CPU to the other tasks in the system. If there are no **PAUSE**s, this task will never release the CPU, and no other task will ever run!

Most Forth I/O words, such as **EMIT**, **KEY**, and **BLOCK**, contain a **PAUSE**. The assumption here is that I/O is slow, and so other tasks should have some time at the CPU.

b) The code executed by the new task must never return! Remember, this code was not entered from a subroutine call -- it is the very first code executed by this task. Thus, there is no return information on the return stack! (Boom.) In general, the "topmost" Forth code of any task must be an endless loop.

In F83, the word **STOP** can be used to end a task, instead of looping.

There is also a rule for Forth programmers during the test and development phase: do not **FORGET** tasks! Remember, once a task is created, it is linked into the round-robin list. If you -- accidentally or intentionally -- reclaim a task's RAM area with **FORGET**, and then put other definitions into that RAM, you will destroy the round-robin linked list! (Again, boom.) It's safest to put all the task creation first in your code, and then never **FORGET** back that far.

10. Example #1: the on-screen clock

Note: the example code given in this article is written for F83 version 2.1.0 for the IBM PC. F83.COM seems to be distributed with the multitasker already installed. Type

' TASK: .

to find out. If **TASK:** is not defined, you will need to install the multitasker by typing

OPEN CPU8086.BLK 21 LOAD OPEN UTILITY.BLK 52 LOAD

Screen 1 of the listing is the code for a simple on-screen clock. This code creates a second task which continually displays the time of day in the upper right-hand corner of the screen.

@TIME is a Forth word to return the IBM PC clock time.

We want to use the Forth display words, but we don't want the clock task to interfere with the main task's display. After we reposition the cursor to the top right of the screen, we need to be able to put it back where it was. The words **@CURS** and **!CURS** invoke BIOS functions to get and set the current display cursor. **.TIME** displays the time in hh:mm:ss format. It illustrates the use of **@CURS** and **!CURS** to get the cursor position, set a new position, and restore the original position. Note the use of **SINGLE** and **MULTI** around **TYPE**. **TYPE** does many **EMIT**s, and each **EMIT** does a **PAUSE**. This would switch back to the main task while the cursor is in the wrong position! Rather than redefine **EMIT** to eliminate the **PAUSE**, we can simply shut off multitasking for the duration of the **TYPE**. (This is, however, quite crude.)

The definition and activation of the **SCREEN-CLOCK** task have already been described. Note that you must specify **DECIMAL** from within the clock task's code. The number base is a user variable, and typing **DECIMAL** from the keyboard will change the main task's number base, not the clock task's.

After loading this screen, type

START-CLOCK MULTI

to activate the clock. You will see the clock appear in the corner of the screen. Type **WORDS** and observe that the displays do not interfere with each other. Then type **SINGLE WORDS** and compare the speed when the multitasker is switched off. The clock display in this example consumes far too much CPU time; it redisplays the clock dozens of times every second, when only once per second would be adequate. A better version would wait for the time to change before redisplaying.

11. Example #2: the round-robin cycle counter

Screen 2 is Forth code to count passes through the round-robin list. Similar code is supplied as an example with F83; this code is different only in that it displays the cycle count in the upper right corner. Load this screen, and type

START-COUNTER

to activate the counter. (You may have to turn **MULTI** back on.) On my 12 MHz AT, with the on-screen clock task also running, I see about 50 counts per second. This means that each task is getting the CPU every 20 milliseconds.

12. Waiting without pain (2.2, 2.4)

Programmers seem to write a lot of wait loops. Most wait loops fall into one of three categories:

a) polling an I/O device. Sometimes you must use polled I/O. The hardware may not have a "data ready" interrupt. Operating system software may only offer a "check status" function -- such as the keyboard under MS-DOS.

b) awaiting an interrupt. When the hardware supports interrupts, you may have to wait for an interrupt to occur. For example, a disk controller using DMA will issue an "end of transfer" interrupt when the operation is complete.

c) waiting for another task. In multitasking applications, occasionally one task has to wait until another task has accomplished something.

Sometimes, waiting is just the obvious way to write a program! Recall the example of the command processor, which has to wait for a keypress to be received and processed.

Wait loops burn up CPU time. Even worse, trying to wait for more than one thing at a time can lead to some Byzantine compound loops! A multitasker solves both of these problems.

One simple addition changes the wait loop from a CPU hog to an efficient programming construct. Simply insert a **PAUSE** in the loop! This ensures that, while this task is waiting, all the other tasks in the system will get to use some CPU time.

Most multitasking Forth kernels write their I/O this way. For example, the basic definition of **KEY**

: (KEY) BEGIN (KEY?) UNTIL 0 8 BDOS ;

is changed in F83 to

: (KEY) BEGIN PAUSE (KEY?) UNTIL 0 8 BDOS ;

As long as no keypress is ready, this task will **PAUSE** repeatedly, surrendering the CPU to the other tasks in the system. The polling represents a slight overhead: on every pass through the task list -- typically every few milliseconds -- this task will execute **PAUSE (KEY?) UNTIL**. If **(KEY?)** is not too complex, this overhead is negligible.

Note that, even if a keypress is ready, the word **(KEY)** will do at least one **PAUSE**. This is usually desirable. If this is not desirable, a **BEGIN..WHILE..REPEAT** loop can be used instead.

What if all of the tasks are waiting for something? Then the system ends up spinning in a much larger wait loop -- checking each task in turn for its "wake up" condition, and moving on to the next. The first task whose poll is successful will then continue execution. With no special programming effort, the multitasker automatically checks a set of events, and starts a different program for each event!

Be aware that putting a **PAUSE** in the loop will significantly reduce the polling rate -- from microseconds to milliseconds. If fast response is essential, or there is a chance that an event can be "missed," some other approach is needed. Interrupts are usually best in this case.

13. Shared resources and semaphores

13.1 The problem of mutual exclusion

Once you have several tasks running in parallel, a new problem can arise: access conflicts. This happens when two tasks attempt to use, simultaneously, some device or resource which can only be used by one task at a time.

Consider, for instance, a disk controller which requires two time-consuming operations to access the disk: a seek, and then a read or write. (Many controllers are like this; the Western Digital 1791 is one example.) Suppose Task A does a seek, which takes a few hundred milliseconds. While it is waiting, other tasks are running. Now suppose that Task B tries at this moment to access the disk. It issues its own seek command, conflicting with the command issued by Task A. When the seek completes, and Task A resumes, it will be at the wrong location on the disk. (Worse, since only one completion signal is issued by the controller, one of the tasks may wait forever for its seek to finish.)

Or consider the on-screen clock and cycle counter examples: while one task had altered the display cursor, other tasks had to be prevented from using it.

Or consider the printer. If one task -- maybe a background printing task -- is outputting to the printer, we certainly don't want another task sending output at the same time.

In the example programs we used the crude solution of switching off the multitasker. This is usually not practical -- it defeats the whole purpose of multitasking! We need a better solution.

Fortunately, this "mutual exclusion" is a classic problem in computer science, and several solutions have been devised over the decades [TAN87]. One of the simplest and most elegant is the "semaphore," invented by by E. W. Dijkstra in 1965.

13.2 The semaphore

In its simplest form, a "binary semaphore" is a flag associated with a resource. Two operations act on semaphores: **WAIT** and **SIGNAL**. **WAIT** checks to see if the resource is available. If so, it is marked "unavailable"; if not, the CPU is released to other tasks until the resource becomes available. **SIGNAL** just marks the resource "available."

In Forth, these can be written

```
: WAIT ( addr -- )
        BEGIN PAUSE DUP C@ UNTIL         \ wait for nonzero = available
        0 SWAP ! ;                       \ make it zero = unavailable

: SIGNAL ( addr -- )
        1 SWAP ! ;                       \ make it nonzero = available
```

(These are also in screen 3 of the listing.)

Dijkstra observed that the operations of testing and setting the semaphore must be indivisible. In the cooperative Forth multitasker, all Forth code is indivisible until a **PAUSE** is executed, so these definitions will work as written. In pre-emptive multitaskers, or any application where interrupts can affect semaphores, interrupts must be disabled within **WAIT** and **SIGNAL** (except for the **PAUSE**).

It is a subject of some debate whether **SIGNAL** should include a **PAUSE**. Adding a **PAUSE** ensures that a task which is blocked on that semaphore will be awakened soonest, thus maximizing the use of the resource. Sometimes, however, this is not what is desired.

13.3 How semaphores are used

Every task, before using a shared resource, does a **WAIT** on its semaphore, and after using the resource, does a **SIGNAL**. This is sufficient to ensure that only one task can use that resource at any time -- and yet even if one task is blocked, the other tasks can run normally.

The semaphore is defined as a simple Forth variable. The semaphore variable must be initialized to a nonzero value, to indicate that the resource is initially "available." This should be done when all other variables are initialized: when you load the application (on a PC), or as part of the startup code (in an embedded application.)

For the disk example described above, the Forth code may look something like

```
VARIABLE DISK-SEMAPHORE

: READ-SECTOR
        DISK-SEMAPHORE WAIT
        SEEK
        READ
        DISK-SEMAPHORE SIGNAL ;
```

Now consider the situation with two tasks trying to access the disk simultaneously:

```
        Task A                  Task B

        WAIT
        SEEK (100 msec)
                                WAIT
        READ (100 msec)          |
                                 |----task #2 is blocked!
        SIGNAL                   |
                                SEEK (100 msec)
                                READ (100 msec)
                                SIGNAL
```

Task A performs a **WAIT**, sees that the disk is available, and proceeds to do its **SEEK**. Task A then waits with **PAUSE** until the seek is complete, so that other tasks can use the CPU. At this point Task B requests the disk resource with **WAIT**. Thanks to Task A, the resource is busy, so Task B is put into a wait loop -- waiting not for the seek to complete, but for **DISK-**

SEMAPHORE to be set. This won't happen until Task A is finished seeking and reading. So, Task A is waiting for the disk, Task B is waiting for Task A's **SIGNAL**, and the other tasks in the system are free to use the CPU.

The beauty of this approach is that the **WAIT** and **SIGNAL** are identical for all tasks, so they can be made part of the common routine **READ-SECTOR** that all tasks use.

13.4 Semaphores for intertask signalling

The **WAIT** operator is general-purpose: it can be used any time you need to wait for a RAM location to become nonzero. Recall that two of the uses of wait loops are to wait for an interrupt to occur, and to wait for a signal from another task.

To wait for an interrupt, simply **WAIT** on a variable you have defined for the purpose. The interrupt routine then just needs to store a nonzero value in that variable. (Often this can be done with a single machine instruction, e.g., increment.) Waiting for another task is exactly the same, except that the other task can use the high-level **SIGNAL** word to store the non-zero value.

Note that, in these cases, the semaphore should be initialized to a zero value.

13.5 Problems with semaphores

Semaphores are not foolproof. They are susceptible to "deadlock," where two tasks, each having grabbed one of two resources, are waiting for the other resource to become available. One solution to this is the "monitor," also described in [TAN87], which can be implemented if you have semaphores.

14. Breaking out of the depths (2.3)

What about the problem of "breaking out" of a deeply nested program, preserving all of its nesting information, and then returning to it at some future time?

Surprise! This is exactly what **PAUSE** does! All of the return information, temporary variables, etc. which need to be preserved, are all part of the task context. The task context, you will recall, is the information which is saved when tasks are switched. So, to suspend a task until some future event, simply do a **WAIT** or some other multitasking wait loop.

Returning to the command processor example: when I rewrote this application, I created a word **KEYPRESS** which waited -- with **PAUSE**s -- for a keypress to become available. I also converted all of the variables used in command processing to user variables. **KEYPRESS** is used in dozens of places in the program, as the command processor works its way through its logic tree. The command processing code is written "normally," with no special attention given to saving the state, or releasing the CPU to other tasks!

15. Message passing

Often parallel tasks need to communicate with each other. One of the most-used schemes is the "mailbox," an agreed-upon place where messages can be left for a task. Mailboxes may be statically allocated to each task -- in which case you must be careful to have enough mailboxes -- or they can be dynamically allocated from a pool.

Perhaps the simplest scheme, which ensures that there are sufficient mailboxes for all the tasks, is to include a mailbox in each task's user area. Then, as tasks are added, new mailboxes are too. Each task can reach its own mailbox easily, as a user variable. Other tasks can reach that mailbox by offset from the given task's address. (Recall that the "task address" is usually the base address of its user area.)

For simplicity, we can limit messages to a single cell (16 bits on the IBM PC). An F83 implementation of this would be:

```
: SEND ( message taskadr -- )
        MYTASK
        OVER SENDER LOCAL       get adrs of destination's SENDER
        BEGIN
        PAUSE                   loop with PAUSE,
        DUP @                   until his SENDER is zero
        0= UNTIL
        !                       store my adrs in his SENDER
        MESSAGE LOCAL ! ;       store the message in his MESSAGE

: RECEIVE ( -- message taskadr )
        BEGIN
        PAUSE                   loop with PAUSE,
        SENDER @                until my SENDER is nonzero
        UNTIL
        MESSAGE @              get the message from my MESSAGE
        SENDER @               get his task adr from my SENDER
        0 SENDER ! ;           indicate mailbox empty & ready
```

MESSAGE and **SENDER** are user variables. (See screen 4 of the listing for their definition.) The convention here is that, if **SENDER** is nonzero, a message is in the task's mailbox. This works because most Forth systems can't have a task user area at address 0000.

SEND sends the given message to the given task. Before it can do so, it must be sure that the destination mailbox is empty -- otherwise it would destroy some other mail. So it waits, with **PAUSE**s, until the destination task's **SENDER** variable is zero. Then it stores the message, and the sending task's address (**MYTASK**), in the destination user area.

RECEIVE waits for any message to appear, as indicated by **SENDER** going nonzero. It then reads the message and the sending task's address. Finally, it clears the **SENDER** field, to indicate that the mailbox is now empty.

Note that if two tasks try to send messages to the same destination, one of the tasks will be blocked until the destination task has read its mail!

This is a very simplistic message system, although adequate for many applications. More sophisticated schemes allow a receiver to select only messages from a specific sender, or allow multiple messages to be queued at the receiver so that multiple senders do not have to wait. Other extensions would allow variable-length or string messages.

16. Example #3: the Print utility

Screen 5 shows a simple use of messages. This is a modification of F83's screen-printing utility **SHOW**, to run as a "background" task. Load this screen, and type

START-SHOW

to activate the printing task. (Don't forget to turn **MULTI** on.) At this point, nothing will happen. You can now type

1 4 SHOW

and a printout of screens 1 to 4 will start on the printer. At the same time, you will get the "ok" prompt on the screen, and you can type Forth commands. Type **WORDS**, for instance.

When the printing task was activated, its first action (in **START-SHOW**) was to clear its mailbox. It then waits for two messages in a row, which it expects to be the first and last screen to print, respectively. (We don't care who sends these messages, so the sending task address is **DROP**ped after each **RECEIVE**.) Since no message has yet been sent, it just waits forever, invisible to the "main" Forth task.

SHOW is redefined to simply send two messages to the printing task. So, when the **SHOW** command is typed at the keyboard, the printing task is awakened and begins to print the requested screens. When the listing is complete, the printing task loops, waiting for a new print request (two more messages).

If you have a slow printer with a small buffer, you will notice occasional interruptions in the **WORDS** listing being displayed by the main task. This might appear as if the multitasker is not working as advertised. Actually, this is an unfortunate byproduct of the MS-DOS printer routine. MS-DOS assumes that a program which is printing can afford to wait, so there is no BIOS call to test the "printer ready" flag. All Forth can do is request a character be output to the printer. If the printer is not ready, MS-DOS (not Forth) will tie up the computer while it waits for the printer. To fix this problem, we would have to bypass MS-DOS and write our own direct printer I/O routines.

17. Advanced topics

The round-robin, cooperative multitasker which has been described is the most common Forth multitasker. Most public-domain Forths and many commercial Forths use this approach -- it's simple, versatile, and efficient. But some Forth systems have extended the multitasker to add new capabilities; and most non-Forth multitaskers offer some of these "advanced features."

17.1 Preemptive multitaskers

The problem with cooperative multitaskers is that each task keeps the CPU until it does a **PAUSE**. It can be very difficult to distribute CPU time equally among the tasks; poorly written code can "hog" the CPU for inordinate amounts of time. Placement of **PAUSE** instructions becomes something of an art.

One solution is to allow a task switch to be forced upon a task, say by a real-time clock interrupt. If tasks are switched every 10 milliseconds, it's easy to allocate CPU time accurately. This also ensures that no task can tie up the CPU; and in a round-robin system, it guarantees an upper bound on how long a task must wait for the CPU.

Preemptive multitaskers have several disadvantages, too:

a) since a task switch can occur at any time -- even in the middle of a **CODE** word -- all of the CPU registers and flags need to be saved with the task context. On a big processor like the 68000, this can make the context switch time much longer. (Recall that the cooperative multitasker only needs to save four registers.)

b) words such as **WAIT** and **SIGNAL** become more complicated, since they must switch interrupts off to be indivisible.

c) since Forth code is no longer "normally" indivisible, many more semaphores are needed to enforce mutual exclusion, and more temporary variables must be moved into the user area.

17.2 Prioritized multitaskers

Another problem is: how do you tell the system that some tasks are more important than others?

Perhaps you want to allocate more CPU to one task than another. Or, it may be desirable to run one task in preference to all others, as long as it's not waiting for an event. Or possibly there is a "background" task which should only run when no other task is able to run (i.e., when all the other tasks are waiting for something). What you need is a way to assign priorities to tasks. High-priority tasks should run in preference to low-priority tasks -- either by receiving a bigger share of the CPU time, or by taking the CPU completely away from lower-priority tasks.

Priority schemes are usually associated with preemptive multitaskers, but this need not be the case. Here's one possible modification to the basic Forth multitasker to implement a priority scheme: instead of **PAUSE** switching to the next task in the round-robin list, make it switch to a designated "first" task in the list. This task then is the highest-priority task; as long as it is able to run, no other task will get service. **WAIT** must be modified to link to the "next" task in the round-robin list.

The disadvantages of priority schemes are:

a) there's usually more overhead involved in task switching, since the multitasker must decide which task to run next.

b) assigning priorities is not simple. Sometimes a small change can have a disproportionate effect on the amount of CPU time various tasks receive.

c) in some systems, it's possible that some tasks will never get any CPU time.

17.3 Interrupts

How interrupts are handled can vary from one multitasker to another.

The simplest approach is to treat interrupts completely apart from the multitasker. Interrupts do not affect the execution of tasks; when an interrupt occurs, the interrupt routine executes, and then returns to whichever task was running. (I.e., interrupts work the same as they do in a non-multitasking system.) Interrupts can set flags which will later be seen by various tasks. This approach works best when the interrupt handlers can be kept simple. It offers the fastest interrupt service time; however, if the purpose of the interrupt is to "wake up" a task, it may be many milliseconds before that happens.

A preemptive multitasker can allow an interrupt to wake up a specific task. If the service to be performed is complex -- say, interpreting a long message received via DMA -- then it's usually better to do that service in a task, rather than in an interrupt handler. (This is because, usually, some or all interrupts are blocked for the duration of an interrupt handler.) The downside: in addition to the disadvantages of preemptive taskers, this approach has a longer interrupt service time, since the interrupt handler is more complex.

There are other variations, but most interrupt service schemes fall loosely into one of these two categories.

18. Conclusion

Multitasking is a powerful tool, and like a good tool, it can dramatically increase your productivity. The basic Forth multitasker is more than adequate for many applications. Like Forth itself, it is simple, versatile, and efficient...and easy for one person to grasp. Add multitasking to your "toolkit": you will find that many programming problems become simpler, and some become trivial!

19. References

[LAX84] Laxen, H. and Perry, M., F83 for the IBM PC, version 2.1.0 [1984]. Distributed by the authors.

[TAN87] Tanenbaum, Andrew S., Operating Systems: Design and Implementation, Prentice-Hall [1987], 719 pp. Any worthwhile book on operating systems should describe semaphores. This book is recent, quite complete, and very readable. It has an good discussion of the various problems of mutual exclusion.

[TIN86] Ting, C.H., Inside F83, 2nd ed., Offete Enterprises, 1306 South "B" Street, San Mateo, CA, 94402, USA [1986], 287 pp. This explains the F83 multitasker quite well. Actually, it explains most of F83 quite well: a "must" for F83 users.

F83 for the IBM PC, CP/M, and 68000 are available on the Forth Roundtable on GEnie. F83 for the IBM PC is also available from the Forth Interest Group, P.O. Box 2154, Oakland, CA 94621, and from many shareware vendors.

The source code for this article is below.

```
Screen 1
========
\ MULTITASKER EXAMPLE 1 - ON SCREEN CLOCK             bjr09sep92
CODE @TIME ( -- h m s)   HEX 2C # AH MOV  21 INT   AH AH SUB
   CH AL MOV  AX PUSH  ( hrs)   CL AL MOV  AX PUSH  ( mins)
   DH AL MOV  1PUSH  ( secs)   END-CODE
CODE @CURS ( -- n)   0 # BH MOV  3 # AH MOV  10 INT  DX PUSH
   NEXT  END-CODE
CODE !CURS ( n --)   DX POP  0 # BH MOV  2 # AH MOV  10 INT
   NEXT  END-CODE
: .TIME ( h m s --)   0 <# # # 3A HOLD  NIP # # 3A HOLD  NIP
   # # #>  @CURS >R  48 !CURS  SINGLE TYPE MULTI  R> !CURS ;

HEX 400 TASK: SCREEN-CLOCK
: START-CLOCK   SCREEN-CLOCK ACTIVATE
   DECIMAL BEGIN @TIME .TIME PAUSE AGAIN ;
: STOP-CLOCK   SCREEN-CLOCK SLEEP ;
DECIMAL

Screen 2
========
\ MULTITASKER EXAMPLE 2 - ROUND-ROBIN CYCLE COUNTER   bjr09sep92
VARIABLE CYCLES   0 CYCLES !  HEX
: .CYCLES   1 CYCLES +!   CYCLES @
   @CURS >R  40 !CURS  SINGLE 6 U.R MULTI  R> !CURS ;

HEX 400 TASK: CYCLE-COUNTER
: START-COUNTER   CYCLE-COUNTER ACTIVATE
   DECIMAL BEGIN .CYCLES PAUSE AGAIN ;
: STOP-COUNTER   CYCLE-COUNTER SLEEP ;
DECIMAL

Screen 3
========
\ SEMAPHORES IN F83                                   bjr09sep92
: WAIT ( addr -- )         \ WAIT for semaphore ready
   BEGIN PAUSE DUP C@ UNTIL   \ wait for nonzero = available
   0 SWAP ! ;                 \ make it zero = unavailable

: SIGNAL ( addr -- )       \ SIGNAL that semaphore is ready
   1 SWAP ! ;                 \ make it nonzero = available

Screen 4
========
\ MESSAGES IN F83                                     bjr09sep92
USER VARIABLE MESSAGE     \ a 16-bit message
     VARIABLE SENDER      \ holds address of the sending task
FORTH
: MYTASK ( -- a)   UP @ ;     \ returns addr of the running task

: SEND ( msg taskadr -- )     \ send msg to the given task
   MYTASK  OVER SENDER LOCAL  \ -- msg taskadr mytask SENDERadr
   BEGIN PAUSE DUP @ 0= UNTIL \ wait until his SENDER is zero
   !   MESSAGE LOCAL ! ;      \ store mytask,msg in his user var
```

```
: RECEIVE ( -- msg taskadr)     \ wait for a message from anyone
   BEGIN PAUSE SENDER @ UNTIL   \ wait until my SENDER nonzero
   MESSAGE @  SENDER @          \ get message and sending task
   0 SENDER ! ;                 \ now ready for another message!

Screen 5
========
\ MULTITASKER EXAMPLE 3 - BACKGROUND PRINT UTILITY    bjr09sep92
HEX 400 TASK: SHOW-TASK

: START-SHOW   SHOW-TASK ACTIVATE
   DECIMAL  0 SENDER !   \ must clear message buffer first
   BEGIN
      RECEIVE DROP  RECEIVE DROP   \ -- first last
      SHOW
   AGAIN ;

: STOP-SHOW   SHOW-TASK SLEEP ;

: SHOW ( first last -- )
   SWAP  SHOW-TASK SEND  SHOW-TASK SEND ;
DECIMAL
```