MULTITASKING 8051 CAMELFORTH

Brad Rodriguez, T-Recursive Technology

This article describes a multitasking extension to the 8051 CamelForth system (described in TCJ #71 and #72). The techniques described here are not limited to Forth; they should be useful to any 8051 assembly-language programmer.

CONTEXT SWITCHING

When several tasks are sharing a processor, the information which is specific to any one task is called that task's "context." This includes the current instruction pointer, CPU registers, subroutine return stack, and any other private data. CamelForth uses the 8051 registers as follows:

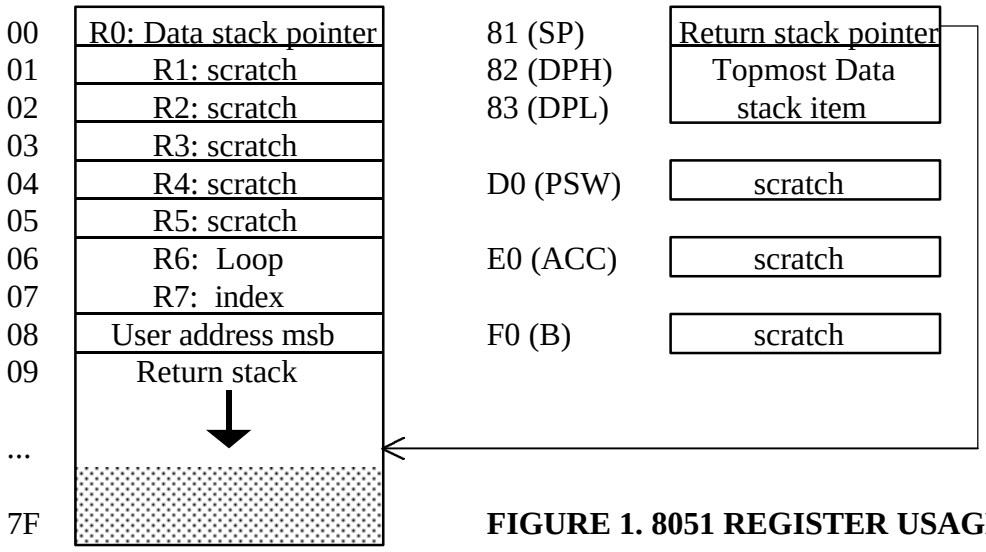

FIGURE 1. 8051 REGISTER USAGE

The registers labelled "scratch" may be used within Forth words, but have no guaranteed value between Forth words (except that the register select bits in PSW must always be 00 in 8051 CamelForth).

The 8051 stack pointer SP points to the last byte actually stored in the Return stack. This stack grows from internal RAM location (register) 09 upward to 7F.

In CamelForth, each task's private data is contained within a 768-byte "task area," organized as in Figure 2.

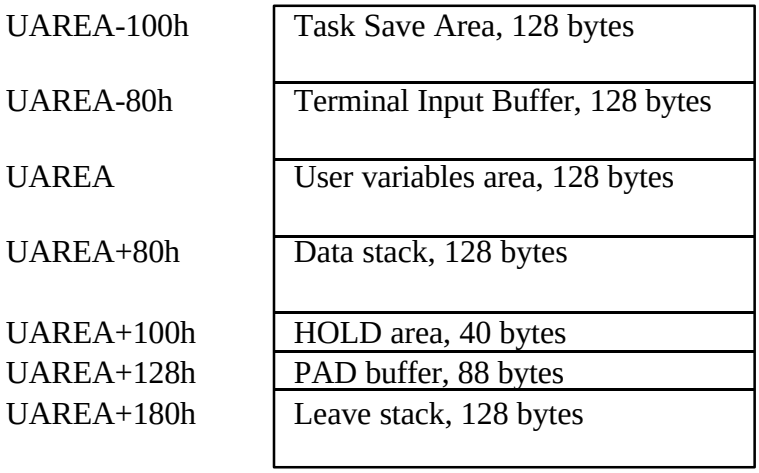

FIGURE 2. CAMELFORTH TASK AREA

All of this data is addressed relative to a pointer, UAREA, whose high byte is stored in register 8. (The Task Area is page-aligned, so the low byte of this pointer is always 00.) To make another Task Area current, only this pointer need be changed. Note that the start of the Task Area is 100h bytes *before* UAREA.

So, to switch the from one task's context to another, the following steps must be taken:

1. Save the "important" registers and the Return stack in external RAM, with the task's private data. This is is purpose of the "Task Save Area" in Figure 2.
2. Switch the UAREA pointer to the new task's User Variables area.
3. Restore the previously-saved registers and Return stack from the new task's Task Save Area.
4. Resume execution.

The code to do this is shown in Listing 1, as a Forth word SWITCH. This word expects the address of the new task's Task Save Area on top of stack (i.e., in DPTR). Obviously, copying the Return stack out to the Task Save Area is the time-consuming step, so SWITCH includes a tight loop to copy registers N to 1 to external RAM, using R0 as the loop counter. The number of the highest register to be saved is contained in the Return stack pointer SP; this number is saved in the first byte of the Task Save Area.

Since SWITCH is called as a subroutine, the program counter will already be pushed on the return stack (and thus saved by the copy loop). Register R0 (the Data stack pointer) is saved by copying it to R1 before the loop. DPTR doesn't need to be saved, since its contents can be discarded after the task is switched. On the other hand, R2-R5 are saved

needlessly, since they don't need to be preserved across Forth words. But including them simplifies the loop, and will come in handy later.

After SWITCH copies the current task's context out to RAM, it fetches the new task's context (specified by the address parameter passed to SWITCH). The process is simply the reverse of what was done to save the context. This assumes that the new Task Save Area contains the data saved when *that* task did a SWITCH. What about a brand-new task?

The Forth word INITTASK sets up the Task Save Area for a new task. It saves 10 bytes in the Task Save Area: two bytes of return stack, and the eight registers 1 through 8. For register 8, the high byte of the User Variables area address is stored. For register 1, the low byte of the Parameter stack pointer, 0FD, is stored (this value will cause an empty stack once the task starts running). The "saved" return stack contains only a return address, which is where execution will begin when the task is started.

So, to create and launch a new task:

1. Reserve 768 bytes of storage for the Task Area. Call this area `taskname`. This name can be defined as a Forth CONSTANT, or by using CREATE if care is taken to keep the area page-aligned.
2. Initialize the task to perform a Forth `word`, with the command
   ```
   ' word taskname INITTASK
   ```
3. Launch the new task with the command
   ```
   taskname SWITCH
   ```

The "main" CamelForth task will be suspended, and the new task will run. Beware: if the new task never does a SWITCH back to the main task, it will retain control forever!

Listing 2 shows an example of how to define a task area in high RAM, by offseting it below the main task's user area (thus ensuring page alignment). It also shows how to write a task that will do something and then return to the main task: after loading this program, every time you type `TASK1 SWITCH`, the 8051 will emit a bell character.

ROUND-ROBIN TASK SWITCHING

For true multitasking, each task must run for a short time, and then hand off control to another task...making sure that all of the defined tasks get a turn. One way to do this is to have each task switch to the "next" task in the list. This can be done, clumsily, with SWITCH statements, as long as the list never changes (since task addresses are hard-coded in the program). A better way is to maintain a dynamic linked list of all tasks in RAM.

Listing 3 shows high-level words to manage a linked list of tasks. The first cell of the user variables (at offset 0 from UAREA) is reserved in CamelForth for a task link. MYTASK always returns the address of the running task area, by offseting -100h from UAREA.

DETACH empties the task list, by making the running task link back to itself. (This causes an inadvertent PAUSE to simply save and restore to the running task's context.) ATTACH adds a task to the list, inserting it immediately after the running task. Finally, PAUSE just causes a switch to the next task in the list. The secret to PAUSE is that it fetches a User variable, TASKLINK. User variables are always addressed relative to the current UAREA pointer, so whenever a task runs PAUSE, it fetches *its* link to the "next" task. In this manner, all of the tasks are linked together in a circular list. (See Figure 3.)

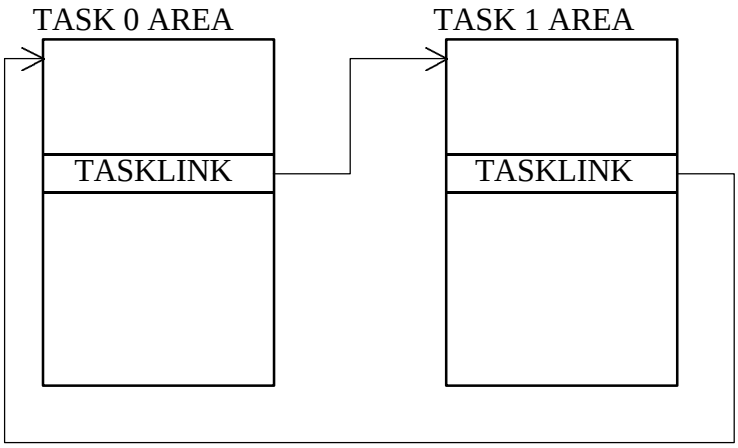

FIGURE 3. LINKED LIST OF TWO TASKS

Listing 3 also includes the "bell" demo, this time written with PAUSE. Every time PAUSE is typed at the keyboard, a bell character is output.

SWITCH can also be used to implement more complex task schedulers, by defining a "dispatcher" task (or a "dispatcher" Forth word). The dispatcher might, for example, select the task having highest-priority according to some rule. This is left as an exercise for the student.

PRE-EMPTIVE TASK SWITCHING

SWITCH and PAUSE require a task to voluntarily give up the CPU to other tasks. This "cooperative" multitasking is simple and efficient, and avoids many of the synchronization problems that can occur when many tasks share a resource. But poorly written tasks can "hog" the CPU, and with even the best-written tasks, it's difficult to parcel out CPU time evenly.

A pre-emptive multitasker uses some external event -- typically a timer interrupt -- to force a task switch. This ensures that tasks are switched regularly. But since a task switch can occur at any time -- not just between Forth words -- we must take pains to save *all* of the working registers that are used by CamelForth.

SWITCH saves all of the working registers except R1-R3, DPH, DPL, ACC, B, and PSW. The simplest solution is for the interrupt to push these onto the return stack, and then call SWITCH with the address of the next task. When the round-robin returns to this task, SWITCH will return to the interrupt service routine, which will then pop these eight

registers and return to wherever this task was suspended. A subroutine PREEMPT to do this is given in Listing 4.

Note that PREEMPT fetches the same task link as does PAUSE. We can't use `ACALL PAUSE,` since the phrase `TASKLINK @` in PAUSE may destroy some registers that we haven't saved. (We could learn this by examining the kernel listing for USER and @, but it's bad form to build such hidden dependencies into kernel words.)

Also, PREEMPT saves all of the "extra" registers on the Return stack, except DPH and DPL. This is because SWITCH, before returning to PREEMPT, pops DPH and DPL from the *Data* stack.

Listing 4 also shows how to link PREEMPT to a timer interrupt, and initialize the timer. Listing 5 is the corresponding high-level test code. Once TASK1 is ATTACHed, the variable TICKS should increment every 65.536 msec (with a 12 MHz oscillator), and yet there should be no visible effect on normal Forth operation. Note the use of PAUSE to return control immediately to the "main" task, after the interrupt is processed.

A disadvantage of this approach is that the RETI instruction, required by the 8051's interrupt processing hardware, is not executed until the preempted task is resumed. For a simple example this is tolerable. To avoid this problem, PREEMPT should call a second copy of SWITCH, identical except that it ends with a RETI. Then PREEMPT should end with a RET.

Also, the preemptive multitasker could be made slightly more efficient if SWITCH saved all working registers (i.e., if PREEMPT were merged into the SWITCH routine). But this adds unnecessary overhead to the "cooperative" SWITCH. The code presented here clearly shows the extra context-switching overhead of a preemptive multitasker.

LISTING 1

```
; ================================================
; CamelForth Multitasker for the Intel 8051
; (c) 1996 Bradford J. Rodriguez
; Permission is granted to freely copy, modify,
; and distribute this program for personal or
; educational use.  Commercial inquiries should
; be directed to the author at 115 First St.,
; #105, Collingwood, Ontario L9Y 4W3 Canada
; ================================================

        .equ dr1,h'01   ; r1 as direct register

; The key word of the multitasker is SWITCH.
; It saves the working registers AND the return
; stack of the currently executing task to a
; storage area in external RAM.  Then it gets
; the saved registers and return stack of the
; new task, restores them, and continues
; execution wherever the new task left off.
;
; Registers as they are saved:
; 01 (R1): saved Parameter Stack pointer.
; 02 (R2): future use
; 03 (R3):     "
; 04 (R4):     "
; 05 (R5):     "
; 06 (R6): loop index
; 07 (R7):     "
; 08:      P2, User Area pointer high
; 09...N:  return stack (N is given by SP)
;
; DPTR is not saved, since it is consumed by
; SWITCH.  (It is the address of the new task's
; save area, UAREA-100h.)
;
; Note that these are stored backwards in
; external RAM, starting at address UAREA-100h.
; Thus the save area of a newly created task
; should look like:
; SP:     0Ah
; 0A,09:  init'l Program Counter, hi byte first
; 08:     task's User Pointer high (stack page)
; 07,06:  xxx
; 05,04:  xxx
; 03,02:  xxx
; 01:     0FDh, initial stack pointer
; The initial stack pointer must be FDh because
; of the poptos at the end of SWITCH.
```

```
; SWITCH     a --      switch to new task
        .drw link
        .set link,*+1
        .db  0,6,"SWITCH"
SWITCH: mov r2,dph      ; stash new task adrs
        mov r3,dpl
        mov dph,UP      ; save me at UAREA-100h
        dec dph
        mov dpl,#h'0
        mov dr1,r0      ; save my Pstack pointer
; This loop copies internal RAM, from location
; (SP) down to 01, to external RAM.  6+7n cycles.
; The length is saved as the first byte.
        mov a,sp        ; sp=high address,
        movx @dptr,a    ;   =length.
        inc dptr
        mov r0,a        ; 00 won't be moved
saveregs: mov a,@r0      ; 1 cycle
        movx @dptr,a     ; 2 cycles
        inc dptr         ; 2 cycles
        djnz r0,saveregs ; 2 cycles

        mov dph,r2      ; now get new task
        mov dpl,r3
; This loop copies external RAM to internal RAM,
; and restores SP accordingly.  6+7n cycles.
        movx a,@dptr    ; get high address
        inc dptr
        mov sp,a        ; restore Rstack pointer
        mov r0,a
getregs: movx a,@dptr
        inc dptr
        mov @r0,a
        djnz r0,getregs

; The top of this restored return stack contains
; a return address in the new task.  DPTR no
; longer contains its top-of-stack; so pop the
; new top of stack from RAM.
        mov r0,dr1      ; restore Pstack pointer
        mov p2,UP       ; set new stack page
        ljmp poptos     ; pop TOS and return

; ------------------------------------------------
; INITTASK   xt a --     initialize a task area
; Given the xt (code address) of a Forth word to
; execute, and the address of a task's save area,
; fill in that save area so the given word will
; execute when that task is started.
```

```
        .drw link
        .set link,*+1
        .db  0,8,"INITTASK"
INITTASK: mov a,#h'0a   ; length
        movx @dptr,a
        inc dptr
        movx a,@r0      ; low byte of xt
        inc r0
        mov r2,a
        movx a,@r0      ; high byte of xt
        inc r0
        movx @dptr,a    ; store high byte first
        inc dptr
        mov a,r2
        movx @dptr,a
        inc dptr
        mov a,dph       ; UAREA=SaveArea+100h, so
        inc a           ; DPH+1 = UAREA high byte
        movx @dptr,a
        inc dptr
        inc dptr        ; skip 6 don't-cares
        inc dptr
        inc dptr
        inc dptr
        inc dptr
        inc dptr
        mov a,#h'fd     ; initial Pstack pointer
        movx @dptr,a
        ljmp poptos
```

LISTING 2

```
( MULTITASKER TEST)

HEX U0 100 - CONSTANT TASK0  ( start of "main" Task Area)
TASK0  300 - CONSTANT TASK1  ( new Task Area, 768 bytes )
                             (  lower)

: TEST1   BEGIN 7 EMIT  TASK0 SWITCH AGAIN ;
' TEST1 TASK1 INITTASK
```

LISTING 3

```
( ROUND-ROBIN TASK LIST)

HEX -100 USER MYTASK    ( start of current Task Area)
       0 USER TASKLINK  ( link to next task in list)

( Initialize the task list to "empty".)
: DETACH
   MYTASK TASKLINK ! ;
```

```
( Insert a new task into the linked list, immediately)
( after the current task.)
: ATTACH ( a -- )
   TASKLINK @           ( my previous successor)
   OVER TASKLINK !      ( new task becomes my successor)
   SWAP 100 + ! ;       ( prev. successor becomes new)
                        (  task's successor)

( Switch to the next task in the list.)
: PAUSE   TASKLINK @ SWITCH ;

( EXAMPLE)
DETACH
MYTASK CONSTANT TASK0
TASK0 300 - CONSTANT TASK1
: TEST1   BEGIN 7 EMIT  PAUSE AGAIN ;
' TEST1 TASK1 INITTASK
TASK1 ATTACH
```

LISTING 4

```
; ------------------------------------------------
; PREEMPT                      force a task switch
; If entered from an interrupt, this will cause
; a switch to the next task in the round robin.
; The task link must be in the first cell of the
; user area (user variable U0).  Note that this
; is an assembler subroutine and must not be
; called as a Forth word.
PREEMPT: push psw     ; save regs used by SWITCH
        push acc
        push b
        push dr1
        push dr2
        push dr3
        lcall pushtos ; DPTR saved on Data stack!
        mov dph,UP    ; fetch task link...
        mov dpl,#0
        movx a,@dptr
        mov r2,a
        inc dptr
        movx a,@dptr
        mov dpl,r2
        mov dph,a     ; ...to DPTR
        acall SWITCH  ; switch to next task
; Execution will resume here when the round-robin
; comes back to this task.  Note that the last
; action of SWITCH is to restore DPH:DPL from the
; Data stack, with "poptos".
        pop dr3       ; restore regs
```

```
        pop dr2
        pop dr1
        pop b
        pop acc
        pop psw
        reti

; ------------------------------------------------
; Sample timer 0 interrupt, entered when timer 0
; rolls over from FFFF to 0000.  The interrupt
; flag is automatically cleared when the
; interrupt service routine is entered.
CLOCK:    sjmp PREEMPT

; CLOCKON starts timer 0 & enables the interrupt
        .drw link
        .set link,*+1
        .db  0,7,"CLOCKON"
CLOCKON: mov tmod,#h'21  ; T1 mode 2, T0 mode 1
        mov th0,#h'0
        mov tl0,#h'0
        setb tcon.4      ; enable timer 0
        mov ie,#h'82     ; enable timer 0 irpt
        ret

; CLOCKOFF stops timer 0 & disables the interrupt
        .drw link
        .set link,*+1
        .db  0,8,"CLOCKOFF"
CLOCKOFF: clr tcon.4    ; disable timer 0
        clr ie.1        ; enable timer 0 irpt
        ret
```

LISTING 5

```
( PREEMPTIVE MULTITASKING TEST)
( requires round-robin support words)

DETACH                    ( reset task list)
VARIABLE TICKS
: TEST2   BEGIN 1 TICKS +!  PAUSE AGAIN ;
' TEST2 TASK1 INITTASK
TASK1 ATTACH              ( add TASK1 to task list)
```