Multi-Tasking Kernel

Table of Contents

INTRODUCTION: SCOPE AND INTENT

A problem often faced in writing microprocessor software is to integrate the independent functions (or tasks) of a system. One approach is to write independent procedures to handle each task, then merge them together to form a cohesive system by using a multi-tasking operating system.

The concept of multi-tasking in computer systems is commonly shrouded in mystique. The general perception seems to be that multi-tasking is only for large systems, is complex to implement and use, and imposes a great deal of code space and system time overhead. This document demonstrates that multi-tasking organizations may be small, fast, and easy to use on general purpose 8-bit microprocessor systems.

Effective use of this document requires some background in assembly-level programming of at least one of the microprocessors covered (8085, Z-80, 6502, 6800, or 6809) and some experience with designing and using linked data structures. While some formal training in operating systems would be helpful, it is not required.

A basic multi-tasking kernel, accompanying data structures, and task/system interfacing is presented in detail for the five microprocessors. The underlying algorithm for all of the kernels is the same, though the code required to save a task state and traverse the task data structure varies markedly. The code presented herein is fully tested and may be used without changes — however, modifications and enhancements to the basic multi-tasking algorithm may substantially improve the performance of the system's software.

Some of the more common adaptations of the basic kernel are outlined. The discussion on implementing these modifications is necessarily general; the application at hand determines the desirability of additional features and often governs the way such features are instituted.

Most of the changes to the basic kernel are independent of other changes. This allows a step-wise refinement approach in adapting the multi-tasking kernel for an application: implement the kernel with only those features required, then make small, distinct changes so that the effects can be observed and evaluated. By taking this approach, confidence in the operation of the kernel is gained while insuring that only the minimum necessary overhead is included.

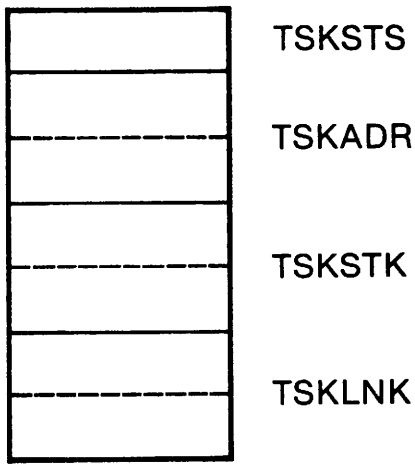
THE BASIC MULTI-TASKING KERNEL

The Basic Multi-Tasking Kernel comprises three segments of code: the task activation block declarations, the task activation block initialization, and the task scheduler.

The Task Activation Block

Each task has its own task activation block (TAB). The TAB contains all the information required by the scheduler to handle a task. The basic kernel's TAB is relatively simple, reflecting the fundamental scheme implemented.

Task Activation Block (TAB)



The TSKSTS field of the TAB is used by the task scheduler to keep track of the status of the task. The basic kernel places a task in one of three states: stopped (symbolic constant TSKSTP) indicating the task is not to be scheduled, new (symbolic constant TSKNEW) indicating the task can be scheduled and is to be started at the address in the two byte field TSKADR, and active (symbolic constant TSKACT) indicating the task is in the middle of processing and the registers must be restored before resuming execution.

The TSKADR field, as indicated in the discussion of TSKSTS, holds the starting address for a task. When a task is first activated, the stack pointer is set to the value in TSKSTK and execution is initiated at the address in TSKADR. Note that task execution is initiated through, effectively, a subroutine call and that a task may be stopped by using a subroutine return as the egress. The value in TSKADR is used only when the scheduler decides to run a task whose status is new.

The TSKSTK field of the TAB holds the value of the stack pointer to be used during the execution of the task. This field is the same size as the microprocessor's stack pointer (two bytes for the 8085, Z-80, 6800, and 6809; one byte for the 6502). Stack space in read/write memory must be allocated for each task with sufficient room for the execution of the task plus storage to save the processor's registers when the task is not active. Since all of the processors' stacks grow downward, the TSKSTK field should be initialized to the address of the topmost byte of the stack area for the associated task.

The TSKLNK field is used by the scheduler to advance from one TAB to the next. The basic kernel uses this field to link all TABs in a circular queue for round-robin scheduling.

TAB Initialization

As part of the system initialization process, before starting the task scheduler, the TABs are initialized. (They must reside in read/write memory.) The task start address and task stack address are set according to actual memory addresses. The task link, for the basic kernel, must be set so as to result in a circular list of TABs for traversal by the task scheduler. The task status is set as per the desires of the user to either TSKNEW, indicating the task is to be run as soon as the scheduler can initiate it, or TSKSTP, indicating that some other task or interrupt service routine will, at a later point, change the status to TSKNEW (thereby making the task a candidate for the CPU).

Some caution must be used when changing the TSKSTS field after initialization. The user should only set TSKSTS to TSKNEW, and should do this only when its value is TSKSTP.

The Task Scheduler

After system initialization, including TAB initialization, the task scheduler may be started by setting the task scheduler variable TSKPTR to point to any TAB in the circular queue and then transferring control to the task scheduler label NXTTSK. The task scheduler is now in control.

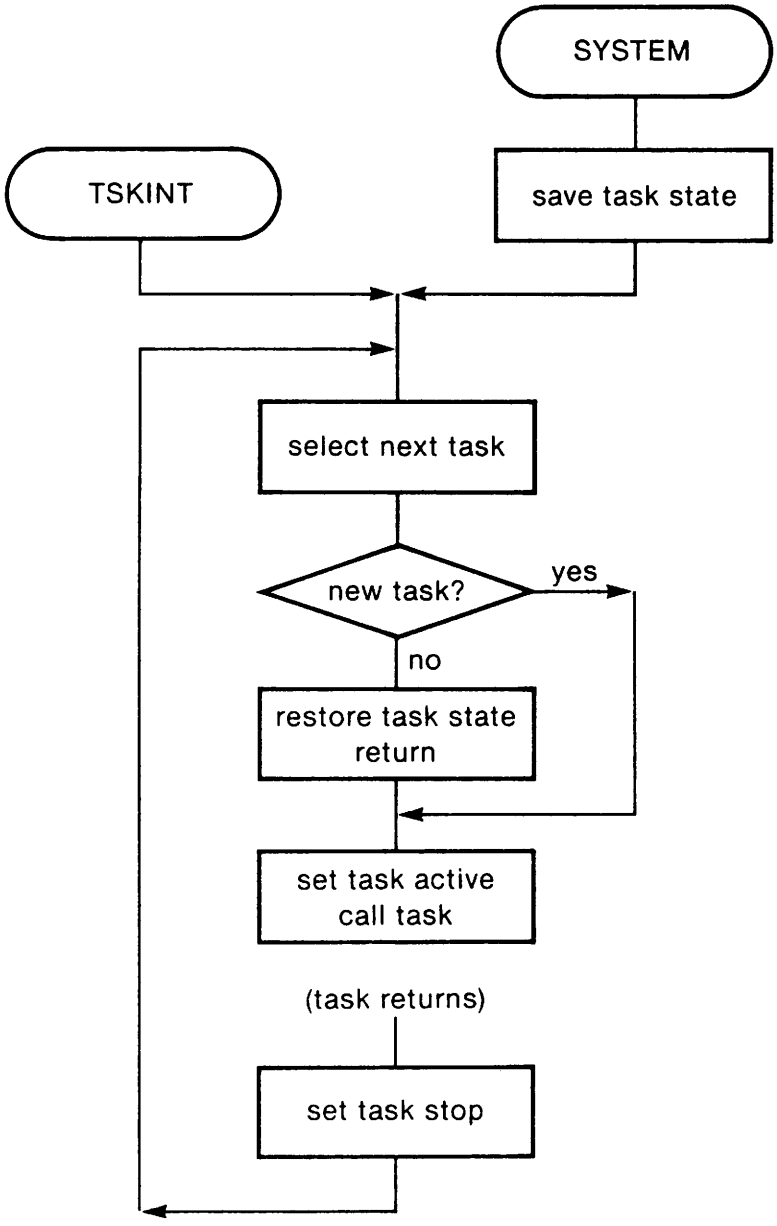
The basic kernel provides the user with a primary entry point: SYSTEM. When a task wishes to relinquish the CPU, but resume operation at some later time, a subroutine call is performed to SYSTEM. At this point the scheduler causes all of the processor's registers to be saved, then begins a search for the next task to activate or resume.

The search is simply a traversal of the circular queue, starting with the task after the current one, looking for a task with a status of new or active. If a task with a new status (TSKNEW) is found, the stack pointer is set and a subroutine call to the ad-

dress in TSKADR is performed. If a task with an active status (TSKACT) is found, the stack pointer is set, all the registers are restored, and the task is resumed by performing a subroutine return (since the task entered the active status by making a subroutine call to SYSTEM).

When a task wishes to deactivate itself, it does a subroutine return. This will enter the scheduler at a different point. The scheduler sets the task status to stopped (TSKSTP) and then searches for the next task to activate or resume. The stack pointer must be proper (for the return to function as well as insure TSKSTK is set correctly).

The other entry point into the scheduler is TSKINT, which is discussed later in this document.



Performance Statistics

	8085	Z-80	6502	6800	6809	
Minimum Code Space	128	101	95	84	59	Bytes
Task Switch Time	179	197	129	133	85	usec.
Clock Frequency	2.0	2.0	1.0	1.0	1.0	MHz

The performance figures given reflect non-critical operating parameters. Task switching time can be improved by increasing the clock frequency or, in the case of the 6800 or the 6809, using direct mode addressing for the task scheduler variables.

BASIC MULTI-TASKING KERNEL ADAPTATIONS

The Task Activation Block

All tasks are completely described to the task scheduler by the Task Activation Block (TAB) associated with each task. The basic kernel has the minimum information needed in its TAB. Additional fields may be added as desired. The information in these additional fields could include some form of task identification, a secondary linkage, resource control data, accounting information, and so on.

The interpretation of the task status field may be extended. The basic kernel allows a task to be in one of three states: inactive (stopped), awaiting initiation (new), and in process (active). Possible extensions could be a hold status (not ready for the CPU but, unlike stopped status, the task can be continued without restarting it) or a delay status. The delay status, in conjunction with a periodic interrupt and a countdown field in the TAB, could be used to cause a task to pause a specified time interval.

In systems with dynamic memory configuration, the TAB could contain the memory configuration to be used while performing the task. Careful control of this is required. The memory should be reconfigured after a task has been selected by the scheduler but before any task activation steps are taken. Interrupt routines and the Multi-Tasking Kernel should never be mapped out of the processor's address space.

Interrupt Initiated Task Switching – I

The basic Multi-Tasking Kernel switches tasks only as a result of a task request (either a task decides to temporarily relinquish the CPU by a subroutine call to SYSTEM or it finishes and deactivates the task by doing a subroutine return). Interrupts can be handled with this basic kernel. The interrupt service routines communicate with tasks through data structures and do their processing independently of task processing.

With the basic kernel, interrupts can modify scheduling by changing the task status byte in a TAB. The task status byte may be changed only from the stop state to the new state. The task in the new state will be activated the next time the task scheduler scans its TAB. The delay between the interrupt occurring (which changes task status byte to new) and the activation of the task can be quite long since the current task may not relinquish the CPU (and invoke the task scheduler) for some time.

It is possible to cause task switching to occur upon request by an interrupt service routine. The mechanics to do this are basically the same across the five microprocessors: upon entry to the interrupt service routine, the microprocessor

registers must be placed on the stack in the order the task scheduler expects them and the exit from the interrupt service routine is accomplished by jumping to the label TSKINT. This will cause the current task to relinquish the CPU and the task scheduler to activate the next task.

CPU SPECIFIC REMARKS

8085: Register pairs PSW, B, D, and H PUSHed (in order)

Z-80: Registers AF, BC, DE, HL, IX, and IY PUSHed (in order)

6502: Registers A, X, and Y pushed (in order; interrupt sequence has pushed the condition code register P)

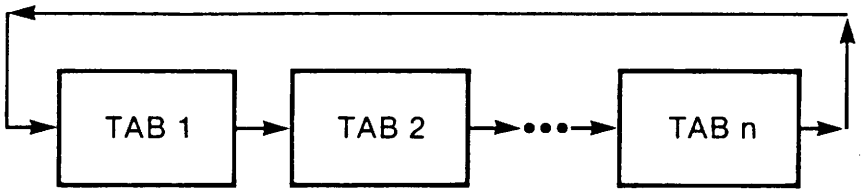
6800: No additional pushes (interrupt sequence pushes all registers)

6809: FIRQs require coding to push all registers (in the same order as NMIs and IRQs).

Priority Ordered Scheduling

The basic kernel schedules tasks in a round-robin ordering. This means that task $n+1$ can run only after task n has run (or had a chance to run). All tasks in a round-robin scheduler have equal access to the CPU. This is the simplest scheduling policy and generally provides the smallest maximum interval between task activations for all tasks. Round-robin scheduling is considered to implement equitable CPU sharing among the tasks.

TAB Linkage For Round-Robin Scheduling

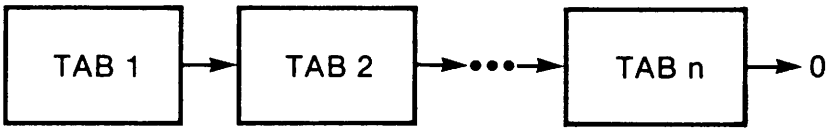


In many cases, all tasks are not created equal — and thus a round-robin policy is not well suited to processing requirements. For example, a task associated with an I/O port that requires rapid response would need better access to the CPU than a task that processes keyboard character entry. This suggests a scheduling policy that activates tasks in a priority order.

As stated previously, the basic kernel links task activation blocks in a circular queue ordering. The round-robin policy is implemented by the scheduler remembering which task was last activated and searching for candidate tasks (to activate) by walking forward through the circular queue. Before a task can be reactivated under this scheduling policy, all other tasks have been activated or given the chance to be activated.

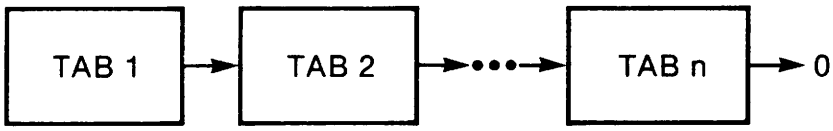
The easiest method of implementing a priority based scheduling policy is to link the TAB in a linear list, with the highest priority task at the head, and modify the scheduler to always start at the head of the list when looking for a task to activate.

TAB Linkage For Priority Scheduling

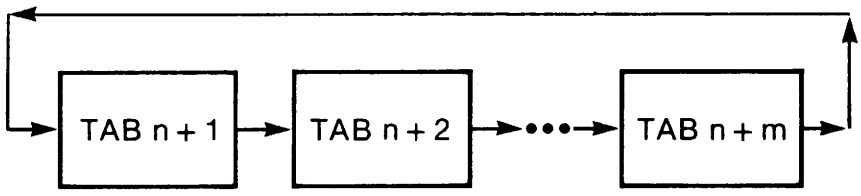


The linear list approach to priority scheduling does not always match the application. A more common case is the one in which there are a set of tasks which require better access to the CPU and another set of tasks for which the round-robin approach makes more sense. A stick-and-ring linkage for the TABs can be used here; the priority tasks are in a linear list and the round-robin tasks are in a circular queue. The scheduler first checks the linear list for any task that can be activated. If none are found, a pointer into the circular queue (to record which task was last activated) is used to start the search for a task in the round-robin set. Some care must be taken in priority scheduling to avoid CPU lock-up. If a priority task never deactivates itself, tasks of lower priority will never get a chance at the CPU. This implies that code exogenous to the priority task, some round-robin task or interrupt service routine, causes a priority task to become ready to be activated. That priority task eventually starts up, does its work, and deactivates itself becoming quiescent until it is needed again.

Priority Stick



Round-Robin Ring



Of course, extensions of this priority scheme abound. The stick-and-ring structure can become a stick-and-ring-and-ring (two levels or priority for round-robin scheduling) and on and on to a stick-of-rings. Generally, great sophistication in priority levels is not required.

Interrupt Initiated Task Switching — II

The use of manual task switching to share the CPU can create a programming burden since calls to SYSTEM, to voluntarily relinquish the CPU, must be strategically placed in each task's code body. An alternative that can be more attractive from the coding standpoint, though requiring additional hardware, is the use of time slice scheduling of tasks in a round-robin task queue.

Time slice scheduling requires a periodic interrupt from the hardware system. Upon receipt of this interrupt, the associated interrupt service routine will cause the task scheduler to switch the CPU to the next task. The period should be chosen based on the maximum interval a task waits between CPU slices (this is the number of tasks multiplied by the slice period) balanced by the system time overhead needed to switch tasks.

Suppose we had a Z-80 system (2 MHz clock) with four tasks to share the CPU. It would probably take about 250 microseconds to identify the interrupt and cause a task switch to occur. If the time slice interval was 1 millisecond, 25% of the Z-80 processor time would be consumed by time slice task switching allowing a task 750 microseconds to run each time slice. The corresponding overhead value for a 50 millisecond time slice is 0.5% allowing a task 49,750 microseconds to run each time slice. In the first case each task would have access to the CPU every 4 milliseconds while the interval for the latter case would be every 200 milliseconds.

In the case of priority scheduling, time slicing does not function — the CPU is always assigned to the task with the highest priority that is ready. The stick-and-ring organization will time slice the tasks that are in the round-robin ring. A time slice interrupt that occurs during the (brief) intervals when a priority task is running will be ignored by the scheduler and have no effect other than to insert some system time overhead.

Dynamic Tasking

All discussion of tasks that are ready (and candidates to have the CPU) versus those tasks complete or waiting for something have assumed a static TAB linkage structure. This means that the scheduler has to make a decision based on traversal of the TAB structure and thus spend time skipping past tasks that are not candidates for the CPU. Further, a static structure requires all tasks to be known and ordered at the time the code is written. The simplicity a static task set gives often outweighs the constraints it imposes.

In some applications, time critical task switching or dynamic creation of tasks is required. This can mean treating the TAB data structure as a dynamic entity — shuffling the order of tasks and adding or deleting TABs as tasks are created or expire. Note that if only tasks which are candidates for the CPU are in the linked list

traversed by the scheduler, the maximum time for a task switch is the same as the minimum time for a task switch.

The implementation of dynamic tasking falls into three areas. First, modifying the TAB data structure. Since the task scheduler traverses the structure to be modified, clearly it must be blocked from operation while the TAB data structure is being modified. Generally, this means shutting off interrupts at the point linkages are reordered. Clearly, the integrity of the TAB data structure must be maintained.

Second, TABs not in the scheduler's list must be handled in some manner. This can involve the construction of a second linked list along with an identification field added to the TAB (if TABs are created and destroyed) or may not involve any special actions if the TAB in question is statically allocated.

Finally, truly dynamic task creation requires free storage management to allocate memory for the TAB, for the associated stack space, and possibly for the data area of the task. With this task organization, it is possible to have the same task code used in several TABs. The distinct stack areas could also hold data being processed. This is the eight bit corollary to shared code on a time-sharing system.

Of course, if circumstances permit, a fixed number of TABs with their associated stacks could be predefined allowing dynamic allocation and deallocation without the overhead of free storage management.

Inter-Task Communication

Most applications require a degree of communication between tasks. The process used to provide this communication should be adapted to the multi-tasking algorithm in use. The problem faced when implementing the communication scheme is dealing with mutual exclusion — that is, being certain that the sending task has completely built the message before the receiving task begins to process it.

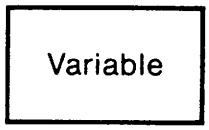
Depending on the type of multi-tasking kernel, the problem of mutual exclusion and methods for dealing with it can range from simple to complex. In the case of a manually scheduled system, (tasks decide when to relinquish the CPU), mutual exclusion is assured by the simple expedient of a task retaining the CPU until the message is completely constructed. Automatic scheduling, (task switching initiated by interrupts), can cause a task to be delayed at any point. This requires some form of interlock on the message transmission mechanism so that a partially constructed message is not available to the receiving task until the sender decides to allow access.

Inter-task messages can take many forms depending on the amount and type of information to be conveyed as well as the extent of an interlock mechanism required. A simple, and often completely adequate, method is the use of what may be called one-way variables. This approach has the sender and receiver share some

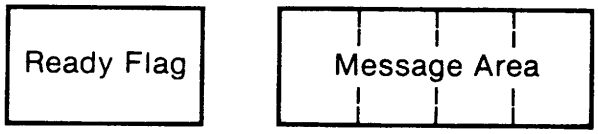
variables but each variable may be written by only one of the tasks. A modification of this scheme is to use a message area and a ready flag. The rules are simple: the sender waits until the ready flag is false (zero), it then builds a message in the message area and when the message is complete the sender sets the ready flag to true (non-zero). The receiver can process the data in the message area only when it has found the ready flag set true. When the receiver does not require the message to remain intact in the message area any longer, it sets the ready flag to false.

The shared variable and message area plus ready flag are simple to implement and quite secure. Note that in an automatically scheduled system the one-way variables must be accessed indivisibly (that is, the task may not be interrupted when it is reading or writing a shared variable). Indivisibility is required to prevent an unintended message resulting from an access of a shared variable in the midst of an update. All single byte and, depending on the microprocessor, some double byte load and store operations are indivisible. However, a multi-byte shared variable should only be accessed when interrupts are (briefly) disabled.

Inter-Task Shared Variable



Inter-Task Ready Flag & Message Area

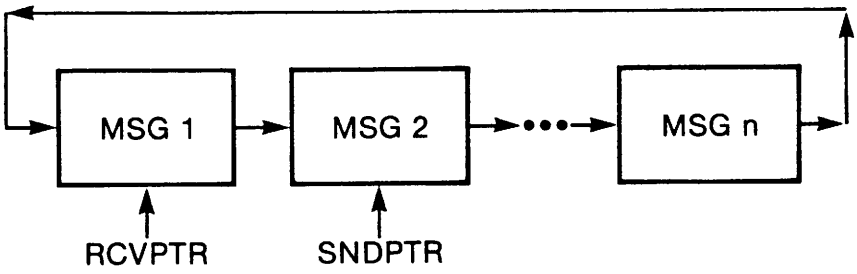


The principle limitation with these two mechanisms is the lock-step synchronism that can result. The sender must wait until the receiver has noted that a message is ready and has processed it. This could cause the sending task to suspend its processing until the receiving task catches up. Significant processing imbalances can occur when the sender wishes to issue several messages or the receiver has a great variation in the time required to process a message.

An alternate communication mechanism which allows a short-term variation in sender and receiver message rates is a circular queue. Note that the circular queue pointer variables are an instance of shared variables — the sending task is the only task that may write the SND pointer and the receiving task is the only task that may write the RCV pointer. With a circular queue the sender will have to wait only when there is not room in the queue for a message. If the average rate at which the sender generates messages is not greater than the average rate the

receiver processes messages, a careful choice of the queue size will permit the sender to occasionally issue a flurry of messages without having to sacrifice its access to the CPU.

Inter-Task Message Queue



The circular queue may comprise either a list of messages or perhaps a list of pointers to message buffers. The latter option in combination with free storage management allows a large queue capacity without tying up read/write memory when it is not needed.

Another inter-task communication technique is to use the scheduler to pass data in what could be called a pipe. When a task has a message to transmit, it points a register to the data area, uses another register to identify the task to receive the data and makes a subroutine call to a special entry in the multi-tasking kernel. A receiving task performs a subroutine call to a different entry point, where it specifies a destination message area and task identification of some type. The scheduler would copy the data when a sender and receiver are ready to transfer data or place either the sending task or the receiving task into a pipe hold status as required. Of course, this scheme could be extended to transfer links to dynamically allocated memory to avoid copying data and to use a circular queue as described previously to allow the sender and receiver to temporarily operate at different rates.

Resource Scheduling

Certain resources may need to be made available to several tasks. How a resource is shared depends on the nature of the resource and the way in which it is to be used.

The sharing of the processing resource, the CPU, has already been discussed. The CPU is able to be switched relatively quickly between tasks that need it. It is possible to resume a task from which the CPU has been taken without impeding the functionality of the task.

Contrast the sharing of the CPU with the sharing of two I/O resources: a line printer and a disk. Clearly, once a task has begun to use a line printer, it must run to completion (it would not do to mix the reports from a number of tasks because

the line printer is switched on a line by line basis). Transfering data to or from a disk is usually done in units of sectors. A disk can be shared among many tasks (with proper support software) but only in quanta of sectors.

The sharing of a resource, as with the CPU, should have some deterministic rule about the degree of access for each task. A rule for a line printer might be to allow an error logging task first chance at the line printer with other tasks on a first come, first serve basis (FCFS). A disk could be shared on a basis ranging from a simple FCFS to a scheme which would select the next task based on how long each requesting task has waited and how long it would take to service the request.

One method for implementing resource sharing is to provide a set of routines which are called by tasks to request a particular resource and then, once the resource has been used, to release the allocated resource. These routines generally must be protected from automatic task switching. The request routine would allocate the resource immediately if it was not already in use, otherwise it would place the task in a resource queue and set the task status to resource hold. The release routine would either allocate the resource to the task at the top of the waiting queue, should any tasks be waiting, or indicate the resource is idle if no task is currently waiting for it. The queue used to keep track of tasks waiting for a particular resource can be implemented by adding a resource linkage field to the TAB.

WRITING TASKS

Once the decision has been made to construct a software system under a multi-tasking organization, the individual tasks must be designed. Four guidelines should be kept in mind: carefully follow the Multi-Tasking Kernel's interface rules, split the processing work into manageably small tasks generally with just one function apiece, provide individual data areas for each task, and keep inter-task communication to a minimum.

To insure that the CPU is utilized effectively, proper use must be made of the Multi-Tasking Kernel's scheduler. A task that is only needed periodically can always be active and check for work to do, or it may deactivate itself (and be activated at the time it is needed). If the latter method is practical, it will reduce job switching and so improve CPU utilization, although this may increase the complexity of the software slightly.

In a manually scheduled round-robin system, a balance must be struck between sharing the CPU (by calling SYSTEM in the body of tasks) and avoiding excessive task switching. When contemplating the overhead penalty of task switching on manual command, the same considerations apply to calculating system time as with time-slice task switching. Usually, processing segments longer than some fixed interval (10 to 100 milliseconds, typically) should be split into smaller CPU slices with calls to SYSTEM. In the case of a wait loop inside of a task, unless exceptional circumstances such as critical response requirements exist, the CPU should be relinquished within the wait loop so as to overlap useful processing of another task with the waiting.

From a design, code, documentation, and maintenance standpoint, it is often profitable to produce single function tasks and bear the system time overhead of switching between them as opposed to fewer, multi-function tasks. However, if a small set of tasks are truly sequential in nature, with no chance to take advantage of parallelism, or a great deal of attention must be paid to inter-task communication, a larger task may be better.

The greatest single source of difficulties in the debug phase of building a multi-tasking system is improper data sharing, often called a pathologic interconnect. This originates from the single address space nature of most eight bit processor systems. The best rule to apply to tasks is to never share a variable unless it is involved in inter-task communication and all accesses to such a variable are carefully controlled and confined to limited areas in the task body. This idea, in a somewhat different form, applies to code used by more than one task (usually a common subroutine). All data used in shared code should exist only in the processor's registers or on the processor's stack (since both areas are private to the task being executed). This data restriction also applies to parameters passed to common code segments. That is, shared code must be implemented in a re-entrant manner.

An almost certain omen of debugging and maintenance difficulties is a large and cumbersome communication pathway between tasks. A great deal of communication between tasks indicates either a poor splitting into tasks at the design point or an exceptional processing requirement that may not be compatible with the multi-tasking concept. Since inter-task communication increases dependencies between tasks, it tends to defeat the "variable separable" approach to coding tasks and so reduces some of the appeal of multi-tasking.

PREFACE TO THE APPENDICES

Coding Conventions

The example assembly source sequences in appendices A through E conform to the following coding conventions:

Program Counter: The program counter is indicated by an asterisk "*"

Comments: Comment fields start with a semi-colon ";" and terminate at the end of the line

Labels: A label is indicated by an identifier followed by a colon ":"

Pseudo-Ops: The pseudo-ops used are ASEG, ORG, and EQU.

```
        ASEG        ; absolute segment
        ORG   expr  ; program counter = expression
ident   EQU   expr  ; identifier = expression
        DS    expr  ; reserve expr bytes storage
```

Functions: Two functions are used in the operand field:

```
LO(expr)  ; use low order expression byte
HI(expr)  ; use high order expression byte
```

Numbers: Numbers are decimal unless postfixed by a "B", "Q", or "H" in which case they are binary, octal, or hexadecimal, respectively.

Operational Remarks

The example code illustrates the basic multi-tasking scheduler — manual scheduling in round-robin ordering. Three tasks are handled in the examples. The first task is always active; it causes a manual task switch whenever it is executed. The second task immediately stops itself when it is activated. The third task is always active; it checks the status of task 2 and, if it is stopped, task 3 causes task 2 to be restarted.

The code sequences which follow allow interrupts to occur. If interrupt processing is not needed, the enabling and disabling of interrupts by the task scheduler may be eliminated.

APPENDIX A: 8085 MULTI-TASKING KERNEL

The following code sequence illustrates an 8085 multi-tasking kernel.

```
            ASEG
;******
;
; TASK RECORD DEFINITION
;
;******
            ORG     0
TSKSTS:     DS      1           ;TASK STATUS (STOP,NEW,ACTIVE)
TSKADR:     DS      2           ;TASK ADDRESS (STARTING ADDRESS)
TSKSTK:     DS      2           ;TASK STACK (ASSOCIATED STACK)
TSKLNK:     DS      2           ;TASK LINK (LINK TO NEXT TASK)
TSKSZE      EQU     *-TSKSTS
;
TSKSTP      EQU     0           ;TASK STATUS IS STOP
TSKNEW      EQU     1           ;TASK STATUS IS NEW
TSKACT      EQU     2           ;TASK STATUS IS ACTIVE
;******
;
; TASK VARIABLES AND TASK BLOCKS
;
;******
            ORG     8000H
TSKPTR:     DS      2           ;TASK POINTER
TASK1:      DS      TSKSZE      ;TASK ONE
TASK2:      DS      TSKSZE      ;TASK TWO
TASK3:      DS      TSKSZE      ;TASK THREE
;******
;
; USER DEFINED STACKS FOR ASSOCIATED TASKS
;
;******
            DS      59
STACK1:     DS      1           ;STACK ONE
            DS      59
STACK2:     DS      1           ;STACK TWO
            DS      59
STACK3:     DS      1           ;STACK THREE
;******
;
; TASK INITIALIZATION
;
;******
            ORG     1000H
            LXI     H,CODE1     ;CODE FOR TASK ONE
            SHLD    TASK1+TSKADR
            LXI     H,STACK1    ;STACK FOR TASK ONE
            SHLD    TASK1+TSKSTK
            MVI     A,TSKNEW    ;NEW TASK
            STA     TASK1+TSKSTS
```

```
            LXI     H,TASK2
            SHLD    TASK1+TSKLNK        ;LINK TO TASK2
;
            LXI     H,CODE2             ;CODE FOR TASK TWO
            SHLD    TASK2+TSKADR
            LXI     H,STACK2            ;STACK FOR TASK TWO
            SHLD    TASK2+TSKSTK
            MVI     A,TSKSTP            ;INACTIVE TASK
            STA     TASK2+TSKSTS
            LXI     H,TASK3
            SHLD    TASK2+TSKLNK        ;LINK TO TASK3
;
            LXI     H,CODE3             ;CODE FOR TASK THREE
            SHLD    TASK3+TSKADR
            LXI     H,STACK3            ;STACK FOR TASK THREE
            SHLD    TASK3+TSKSTK
            MVI     A,TSKNEW            ;NEW TASK
            STA     TASK3+TSKSTS
            LXI     H,TASK1
            SHLD    TASK3+TSKLNK        ;LINK TO TASK1
;
            LXI     H,TASK3             ;HL = TSKPTR
            SHLD    TSKPTR              ;TSKPTR = ADDRESS OF TASK3
            JMP     NXTTSK              ;START TASK 1
;*****
;
; SYSTEM TASK SCHEDULER
; ENTERED VIA CALL FROM ACTIVE TASK
;
;*****
SYSTEM:     DI                          ;DISABLE INTERRUPTS
            PUSH    PSW                 ;SAVE TASK STATE
            PUSH    B
            PUSH    D
            PUSH    H
;*****
;
; ENTRY POINT FOR INTERRUPT INITIATED TASK SWITCH
;
;*****
TSKINT:     LHLD    TSKPTR              ;HL = TSKPTR
            LXI     D,TSKSTK
            DAD     D                   ;HL = ADDRESS OF TASK STACK
            XCHG
            LXI     H,0
            DAD     SP
            XCHG                        ;DE = SP
            MOV     M,E
            INX     H
            MOV     M,D                 ;TASK STACK = SP
;
; NEXT TASK
;
NXTTSK:     LHLD    TSKPTR              ;HL = TSKPTR
            LXI     D,TSKLNK
```

```
            DAD     D               ;HL = ADDRESS OF TASK LINK
            MOV     E,M
            INX     H
            MOV     D,M
            XCHG                    ;HL = TASK LINK
            SHLD    TSKPTR          ;TSKPTR = TASK LINK
;
            LXI     D,TSKSTS
            DAD     D               ;HL = ADDRESS OF TASK STATUS
            MOV     A,M             ; A = TASK STATUS
;
            CPI     TSKSTP
            JZ      NXTTSK          ;IF TASK STATUS = STOP THEN NXTTSK
;
            LXI     D,TSKSTK-TSKSTS
            DAD     D               ;HL = ADDRESS OF TASK STACK
            MOV     E,M
            INX     H
            MOV     H,M
            MOV     L,E             ;HL = TASK STACK
            SPHL                    ;SP = TASK STACK
;
            CPI     TSKNEW
            JZ      NEWTSK          ;IF TASK STATUS = NEW THEN NEWTSK
;
; RETURN TO ACTIVE TASK
;
            POP     H               ;RESTORE TASK STATE
            POP     D
            POP     B
            POP     PSW
            EI                      ;ENABLE INTERRUPTS
            RET                     ;RETURN TO ACTIVE TASK
;
; ACTIVATE NEW TASK
;
NEWTSK:     LHLD    TSKPTR          ;HL = TSKPTR
            LXI     D,TSKSTS
            DAD     D               ;HL = ADDRESS OF TASK STATUS
            MVI     M,TSKACT        ;TASK STATUS = ACTIVE
;
            LXI     D,ENDTSK
            PUSH    D               ;RETURN ADDRESS = ENDTSK
;
            LXI     D,TSKADR-TSKSTS
            DAD     D               ;HL = ADDRESS OF TASK ADDRESS
            MOV     E,M
            INX     H
            MOV     H,M
            MOV     L,E             ;HL = TASK ADDRESS
;
            EI                      ;ENABLE INTERRUPTS
            PCHL                    ;ACTIVATE THE TASK
;
; DEACTIVATE ACTIVE TASK
```

```
;
ENDTSK:  DI                          ;DISABLE INTERRUPTS
;
         LHLD    TSKPTR              ;HL = TSKPTR
         LXI     D,TSKSTS
         DAD     D                   ;HL = ADDRESS OF TASK STATUS
         MVI     M,TSKSTP            ;TASK STATUS = STOP
;
         LXI     D,TSKSTK-TSKSTS
         DAD     D                   ;HL = ADDRESS OF TASK STACK
         XCHG
         LXI     H,0
         DAD     SP
         XCHG                        ;DE = SP
         MOV     M,E
         INX     H
         MOV     M,D                 ;TASK STACK = SP
         JMP     NXTTSK              ;SCHEDULE NEXT TASK
; * * * * *
;
; USER DEFINED CODE FOR TASK ONE
;
; * * * * *
CODE1:
         CALL    SYSTEM              ;SCHEDULE OTHER TASKS
         JMP     CODE1               ;DO TASK AGAIN
; * * * * *
;
; USER DEFINED CODE FOR TASK TWO
;
; * * * * *
CODE2:
         RET                         ;ASSUME TASK COMPLETE
; * * * * *
;
; USER DEFINED CODE FOR TASK THREE
; THIS EXAMPLE TASK INITIATES TASK 2 IF IT IS INACTIVE
;
; * * * * *
CODE3:
         LDA     TASK2 + TSKSTS
         CPI     TSKSTP
         JNZ     C3SKP               ;IF TASK2 STATUS ≠ TSKSTP THEN SKIP
         MVI     A,TSKNEW
         STA     TASK2 + TSKSTS      ;TASK2 STATUS = TSKNEW
C3SKP:   CALL    SYSTEM              ;SCHEDULE OTHER TASKS
         JMP     CODE3
         END
```

APPENDIX B: Z-80 MULTI-TASKING KERNEL

The following code sequence illustrates a Z-80 multi-tasking kernel. Many times the alternate register set is used exclusively in foreground which means that it need not be saved by the multi-tasking kernel. If this is not the case then alternate register set should be saved and restored in the indicated places.

```
            ASEG
;******
;
; TASK RECORD DEFINITION
;
;******
            ORG     0
TSKSTS:     DS      1                   ;TASK STATUS (STOP,NEW,ACTIVE)
TSKADR:     DS      2                   ;TASK ADDRESS (STARTING ADDRESS)
TSKSTK:     DS      2                   ;TASK STACK (ASSOCIATED STACK)
TSKLNK:     DS      2                   ;TASK LINK (LINK TO NEXT TASK)
TSKSZE      EQU     *-TSKSTS
;
TSKSTP      EQU     0                   ;TASK STATUS IS STOP
TSKNEW      EQU     1                   ;TASK STATUS IS NEW
TSKACT      EQU     2                   ;TASK STATUS IS ACTIVE
;******
;
; TASK VARIABLES AND TASK BLOCKS
;
;******
            ORG     8000H
TSKPTR:     DS      2                   ;TASK POINTER
TASK1:      DS      TSKSZE              ;TASK ONE
TASK2:      DS      TSKSZE              ;TASK TWO
TASK3:      DS      TSKSZE              ;TASK THREE
;******
;
; USER DEFINED STACKS FOR ASSOCIATED TASKS
;
;******
            DS      59
STACK1:     DS      1                   ;STACK ONE
            DS      59
STACK2:     DS      1                   ;STACK TWO
            DS      59
STACK3:     DS      1                   ;STACK THREE
;******
;
; TASK INITIALIZATION
;
;******
            ORG     1000H
            LD      HL,CODE1            ;CODE FOR TASK ONE
            LD      (TASK1 + TSKADR),HL
            LD      HL,STACK1           ;STACK FOR TASK ONE
```

```
            LD      (TASK1 + TSKSTK),HL
            LD      A,TSKNEW            ;NEW TASK
            LD      (TASK1 + TSKSTS),A
            LD      HL,TASK2
            LD      (TASK1 + TSKLNK),HL ;LINK TO TASK2
;
            LD      HL,CODE2            ;CODE FOR TASK TWO
            LD      (TASK2 + TSKADR),HL
            LD      HL,STACK2           ;STACK FOR TASK TWO
            LD      (TASK2 + TSKSTK),HL
            LD      A,TSKSTP            ;INACTIVE TASK
            LD      (TASK2 + TSKSTS),A
            LD      HL,TASK3
            LD      (TASK2 + TSKLNK),HL ;LINK TO TASK3
;
            LD      HL,CODE3            ;CODE FOR TASK THREE
            LD      (TASK3 + TSKADR),HL
            LD      HL,STACK3           ;STACK FOR TASK THREE
            LD      (TASK3 + TSKSTK),HL
            LD      A,TSKNEW            ;NEW TASK
            LD      (TASK3 + TSKSTS),A
            LD      HL,TASK1
            LD      (TASK3 + TSKLNK),HL ;LINK TO TASK1
;
            LD      IX,TASK3            ;IX = TSKPTR
            LD      (TSKPTR),IX         ;TSKPTR = ADDRESS OF TASK 3
            JR      NXTTSK              ;START TASK1
; * * * * *
;
; SYSTEM TASK SCHEDULER
; ENTERED VIA CALL FROM ACTIVE TASK
;
; * * * * *
SYSTEM:     DI                          ;DISABLE INTERRUPTS
            PUSH    AF                  ;SAVE TASK STATE
            PUSH    BC
            PUSH    DE
            PUSH    HL
; **** ALTERNATE REGISTER SET MAY BE SAVED HERE AS NECESSARY
            PUSH    IX
            PUSH    IY
; * * * * *
;
; ENTRY POINT FOR INTERRUPT INITIATED TASK SWITCH
;
; * * * * *
TSKINT:     LD      IX,(TSKPTR)         ;IX = TSKPTR
            LD      HL,0
            ADD     HL,SP               ;HL = SP
            LD      (IX + TSKSTK + 0),L
            LD      (IX + TSKSTK + 1),H ;TASK STACK = SP
;
; NEXT TASK
```

```
;
NXTTSK:  LD      L,(IX + TSKLNK + 0)
         LD      H,(IX + TSKLNK + 1)      ;HL = LINKAGE TO NEXT TASK
         LD      (TSKPTR),HL
         LD      IX,(TSKPTR)              ;IX = TSKPTR
;
         LD      A,(IX + TSKSTS)          ; A = TASK STATUS
         CP      TSKSTP
         JR      Z,NXTTSK                 ;IF TASK STATUS = STOP THEN NXTTSK
;
         LD      L,(IX + TSKSTK + 0)
         LD      H,(IX + TSKSTK + 1)      ;HL = TASK STACK
         LD      SP,HL                    ;SP = HL
;
         CP      TSKNEW
         JR      Z,NEWTSK                 ;IF TASK STATUS = NEW THEN NEWTSK
;
; RETURN TO ACTIVE TASK
;
         POP     IY                       ;RESTORE TASK STATE
         POP     IX
; **** ALTERNATE REGISTER SET MAY BE RESTORED HERE AS NECESSARY
         POP     HL
         POP     DE
         POP     BC
         POP     AF
         EI                               ;ENABLE INTERRUPTS
         RET                              ;RETURN TO ACTIVE TASK
;
; ACTIVATE NEW TASK
;
NEWTSK:  LD      (IX + TSKSTS),TSKACT     ;TASK STATUS = ACTIVE
;
         LD      HL,ENDTSK
         PUSH    HL                       ;RETURN ADDRESS = ENDTSK
;
         LD      L,(IX + TSKADR + 0)
         LD      H,(IX + TSKADR + 1)      ;HL = TASK ADDRESS
         EI                               ;ENABLE INTERRUPTS
         JP      (HL)                     ;ACTIVATE THE TASK
;
; DEACTIVATE ACTIVE TASK
;
ENDTSK:  DI                               ;DISABLE INTERRUPTS
         LD      IX,(TSKPTR)              ;IX = TSKPTR
         LD      (IX + TSKSTS),TSKSTP     ;TASK STATUS = STOP
         LD      HL,0
         ADD     HL,SP                    ;HL = SP
         LD      (IX + TSKSTK + 0),L
         LD      (IX + TSKSTK + 1),H      ;TASK STACK = SP
         JR      NXTTSK                   ;SCHEDULE NEXT TASK
```

```
;*****
;
; USER DEFINED CODE FOR TASK ONE
;
;*****
CODE1:
        CALL    SYSTEM              ;SCHEDULE OTHER TASKS
        JP      CODE1               ;DO TASK AGAIN
;*****
;
; USER DEFINED CODE FOR TASK TWO
;
;*****
CODE2:
        RET                         ;ASSUME TASK COMPLETE
;*****
;
; USER DEFINED CODE FOR TASK THREE
; THIS EXAMPLE TASK INITIATES TASK 2 IF IT IS INACTIVE
;
;*****
CODE3:
        LD      A,TASK2+TSKSTS
        CP      TSKSTP
        JR      NZ,C3SKP            ;IF TASK2 STATUS ≠ TSKSTP THEN SKIP
        LD      A,TSKNEW
        LD      (TASK2+TSKSTS),A    ;TASK2 STATUS = TSKNEW
C3SKP:  CALL    SYSTEM              ;SCHEDULE OTHER TASKS
        JP      CODE3
        END
```

APPENDIX C: 6502 MULTI-TASKING KERNEL

The following code sequence illustrates a 6500 series multi-tasking kernel.

```
            ASEG
; * * * * *
;
; TASK RECORD DEFINITION
;
; * * * * *
            ORG     0
TSKSTS:     DS      1                   ;TASK STATUS (STOP,NEW,ACTIVE)
TSKADR:     DS      2                   ;TASK ADDRESS (STARTING ADDRESS)
TSKSTK:     DS      1                   ;TASK STACK (ASSOCIATED STACK)
TSKLNK:     DS      2                   ;TASK LINK (LINK TO NEXT TASK)
TSKSZE      EQU     *-TSKSTS
;
TSKSTP      EQU     0                   ;TASK STATUS IS STOP
TSKNEW      EQU     1                   ;TASK STATUS IS NEW
TSKACT      EQU     2                   ;TASK STATUS IS ACTIVE
; * * * * *
;
; TASK VARIABLES AND TASK BLOCKS
;
; * * * * *
            ORG     0080H               ;ABSOLUTE PAGE
TSKPTR:     DS      2                   ;TASK POINTER
TSKJMP:     DS      2                   ;INDIRECT JUMP POINT FOR TASK START-UP
            ORG     0200H               ;RAM, NOT REQ'D ON ABSOLUTE PAGE
TASK1:      DS      TSKSZE              ;TASK ONE
TASK2:      DS      TSKSZE              ;TASK TWO
TASK3:      DS      TSKSZE              ;TASK THREE
; * * * * *
;
; USER DEFINED STACKS FOR ASSOCIATED TASKS
;
; * * * * *
            ORG     0100H               ;STACK PAGE
            DS      59
STACK1:     DS      1                   ;STACK ONE
            DS      59
STACK2:     DS      1                   ;STACK TWO
            DS      59
STACK3:     DS      1                   ;STACK THREE
; * * * * *
;
; TASK INITIALIZATION
;
; * * * * *
            ORG     8000H               ;ROM SPACE
            LDA     #LO(CODE1)          ;EXECUTION ADDR FOR TASK 1
            STA     TASK1+TSKADR+0
            LDA     #HI(CODE1)
```

```
            STA     TASK1+TSKADR+1
            LDA     #LO(STACK1)         ;STACK FOR TASK ONE
            STA     TASK1+TSKSTK
            LDA     #TSKNEW             ;NEW TASK
            STA     TASK1+TSKSTS
            LDA     #LO(TASK2)          ;LINK TO TASK 2
            STA     TASK1+TSKLNK+0
            LDA     #HI(TASK2)
            STA     TASK1+TSKLNK+1
;
            LDA     #LO(CODE2)          ;EXECUTION ADDR FOR TASK 2
            STA     TASK2+TSKADR+0
            LDA     #HI(CODE2)
            STA     TASK2+TSKADR+1
            LDA     #LO(STACK2)         ;STACK FOR TASK TWO
            STA     TASK2+TSKSTK
            LDA     #TSKNEW             ;NEW TASK
            STA     TASK2+TSKSTS
            LDA     #LO(TASK3)          ;LINK TO TASK 3
            STA     TASK2+TSKLNK+0
            LDA     #HI(TASK3)
            STA     TASK2+TSKLNK+1
;
            LDA     #LO(CODE3)          ;EXECUTION ADDR FOR TASK 3
            STA     TASK3+TSKADR+0
            LDA     #HI(CODE3)
            STA     TASK3+TSKADR+1
            LDA     #LO(STACK3)         ;STACK FOR TASK THREE
            STA     TASK3+TSKSTK
            LDA     #TSKNEW             ;NEW TASK
            STA     TASK3+TSKSTS
            LDA     #LO(TASK1)          ;LINK TO TASK 1
            STA     TASK3+TSKLNK+0
            LDA     #HI(TASK1)
            STA     TASK3+TSKLNK+1
;
            LDA     #LO(TASK3)          ;TSKPTR = ADDR OF TASK3
            STA     TSKPTR+0
            LDA     #HI(TASK3)
            STA     TSKPTR+1
            JMP     NXTTSK              ;START TASK 1
;*****
;
; SYSTEM TASK SCHEDULER
; ENTERED VIA CALL FROM ACTIVE TASK
;
;*****
SYSTEM:     SEI                         ;DISABLE INTERRUPTS
;
            PHP                         ;SAVE WORKING REGISTERS
            PHA
            TXA
            PHA
            TYA
            PHA
```

```
;*****
;
; ENTRY POINT FOR INTERRUPT INITIATED TASK SWITCH
;
;*****
TSKINT:   TSX
          TXA
          LDY     #TSKSTK
          STA     (TSKPTR),Y          ;SAVE TASK STACK POINTER
;
; NEXT TASK
;
NXTTSK:   LDY     #TSKLNK             ;SET TSKPTR = ADDR OF NEXT TASK
          LDA     (TSKPTR),Y
          TAX
          INY
          LDA     (TSKPTR),Y
          STX     TSKPTR+0
          STA     TSKPTR+1
;
          LDY     #TSKSTS
          LDA     (TSKPTR),Y
          CMP     #TSKSTP
          BEQ     NXTTSK              ;IF TASK STATUS = STOP THEN NXTTSK
;
          LDY     #TSKSTK
          LDA     (TSKPTR),Y
          TAX
          TXS                         ;SP = TASK STACK
;
          LDY     #TSKSTS
          LDA     (TSKPTR),Y
          CMP     #TSKNEW
          BEQ     NEWTSK              ;IF TASK STATUS = NEW THEN NEWTSK
;
; RETURN TO ACTIVE TASK
;
          PLA                         ;RESTORE TASK STATE
          TAY
          PLA
          TAX
          PLA
          PLP
;
          CLI                         ;ENABLE INTERRUPTS
          RTS                         ;RETURN TO TASK
;
; ACTIVATE NEW TASK
;
STRTUP:   CLI                         ;COMPLETE ACTIVATION STEPS
          JMP     (TSKJMP)
;
NEWTSK:   LDY     #TSKSTS
          LDA     #TSKACT
          STA     (TSKPTR),Y          ;TASK STATUS = ACTIVE
```

```
;
            LDY     #TSKADR
            LDA     (TSKPTR),Y
            STA     TSKJMP+0
            INY
            LDA     (TSKPTR),Y
            STA     TSKJMP+1
            JSR     STRTUP              ;ACTIVATE TASK
;
; DEACTIVATE ACTIVE TASK
;
ENDTSK:     SEI                         ;DISABLE INTERRUPTS
;
            LDY     #TSKSTS
            LDA     #TSKSTP
            STA     (TSKPTR),Y          ;TASK STATUS = STOP
;
            TSX
            TXA
            LDY     #TSKSTK
            STA     (TSKPTR),Y          ;TASK STACK = SP
;
            JMP     NXTTSK              ;SCHEDULE NEXT TASK
;
;*****
;
; USER DEFINED CODE FOR TASK ONE
;
;*****
CODE1:
            JSR     SYSTEM              ;SCHEDULE OTHER TASKS
            JMP     CODE1               ;DO TASK AGAIN
;*****
;
; USER DEFINED CODE FOR TASK TWO
;
;*****
CODE2:
            RTS                         ;ASSUME TASK COMPLETE
;*****
;
; USER DEFINED CODE FOR TASK THREE
; THIS EXAMPLE TASK INITIATES TASK 2 IF IT IS INACTIVE
;
;*****
CODE3:
            LDA     TASK2+TSKSTS
            CMP     #TSKSTP
            BNE     C3SKP               ;IF TASK2 STATUS ≠ TSKSTP THEN SKIP
            LDA     #TSKNEW
            STA     TASK2+TSKSTS        ;TASK2 STATUS = TSKNEW
C3SKP:      JSR     SYSTEM              ;SCHEDULE OTHER TASKS
            JMP     CODE3
            END
```

APPENDIX D: 6800 MULTI-TASKING KERNEL

The following code sequence illustrates a 6800 multi-tasking kernel.

```
            ASEG
;******
;
; TASK RECORD DEFINITION
;
;******
            ORG     0
TSKSTS:     DS      1                   ;TASK STATUS (STOP,NEW,ACTIVE)
TSKADR:     DS      2                   ;TASK ADDRESS (STARTING ADDRESS)
TSKSTK:     DS      2                   ;TASK STACK (ASSOCIATED STACK)
TSKLNK:     DS      2                   ;TASK LINK (LINK TO NEXT TASK)
TSKSZE      EQU     *-TSKSTS
;
TSKSTP      EQU     0                   ;TASK STATUS IS STOP
TSKNEW      EQU     1                   ;TASK STATUS IS NEW
TSKACT      EQU     2                   ;TASK STATUS IS ACTIVE
;******
;
; TASK VARIABLES AND TASK BLOCKS
;
;******
            ORG     0100H
TSKRA:      DS      1                   ;A REGISTER TEMP
TSKCC:      DS      1                   ;CONDITION CODE TEMP
TSKRX:      DS      2                   ;X REGISTER TEMP
TSKPTR:     DS      2                   ;TASK POINTER
TASK1:      DS      TSKSZE              ;TASK ONE
TASK2:      DS      TSKSZE              ;TASK TWO
TASK3:      DS      TSKSZE              ;TASK THREE
;******
;
; USER DEFINED STACKS FOR ASSOCIATED TASKS
;
;******
            DS      59
STACK1:     DS      1                   ;STACK ONE
            DS      59
STACK2:     DS      1                   ;STACK TWO
            DS      59
STACK3:     DS      1                   ;STACK THREE
;******
;
; TASK INITIALIZATION
;
;******
            ORG     8000H
            LDX     #CODE1              ;CODE FOR TASK ONE
            STX     TASK1+TSKADR
            LDX     #STACK1             ;STACK FOR TASK ONE
```

```
            STX     TASK1+TSKSTK
            LDAA    #TSKNEW             ;NEW TASK
            STAA    TASK1+TSKSTS
            LDX     #TASK2
            STX     TASK1+TSKLNK        ;LINK TO TASK2
;
            LDX     #CODE2              ;CODE FOR TASK TWO
            STX     TASK2+TSKADR
            LDX     #STACK2             ;STACK FOR TASK TWO
            STX     TASK2+TSKSTK
            LDAA    #TSKNEW             ;NEW TASK
            STAA    TASK2+TSKSTS
            LDX     #TASK3
            STX     TASK2+TSKLNK        ;LINK TO TASK3
;
            LDX     #CODE3              ;CODE FOR TASK THREE
            STX     TASK3+TSKADR
            LDX     #STACK3             ;STACK FOR TASK THREE
            STX     TASK3+TSKSTK
            LDAA    #TSKNEW             ;NEW TASK
            STAA    TASK3+TSKSTS
            LDX     #TASK1
            STX     TASK3+TSKLNK        ;LINK TO TASK1
;
            LDX     #TASK3
            STX     TSKPTR              ;TSKPTR = ADDRESS OF TASK3
            BRA     NXTTSK              ;START TASK 1
; * * * * *
;
; SYSTEM TASK SCHEDULER
; ENTERED VIA CALL FROM ACTIVE TASK
;
; * * * * *
SYSTEM:     NOP
            SEI                         ;DISABLE INTERRUPTS
;
; SAVE REGISTERS ON THE STACK IN INTERRUPT ORDER, SEE RTI BELOW
;
            PSHA                        ; SAVE A WITHOUT CHANGING CC REG
            TPA
            STAA    TSKCC
            PULA
            STAA    TSKRA               ;SAVE TASK STATE (IRQ ORDER)
            STX     TSKRX
            LDAA    TSKRX+1
            PSHA                        ;X REG LOW BYTE
            LDAA    TSKRX+0
            PSHA                        ;X REG HIGH BYTE
            LDAA    TSKRA
            PSHA                        ;A REG
            PSHB                        ;B REG
            LDAA    TSKCC
            PSHA                        ;CONDITION CODE
```

```
;*****
;
; ENTRY POINT FOR INTERRUPT INITIATED TASK SWITCH
;
;*****
TSKINT:   LDX     TSKPTR
          STS     TSKSTK,X          ;TASK STACK = SP
;
; NEXT TASK
;
NXTTSK:   LDX     TSKLNK,X
          STX     TSKPTR            ;TSKPTR = ADDRESS OF NEXT TASK
;
          LDAA    TSKSTS,X
          CMPA    #TSKSTP
          BEQ     NXTTSK            ;IF TASK STATUS = STOP THEN NXTTSK
;
          LDS     TSKSTK,X          ;SP = TASK STACK
;
          CMPA    #TSKNEW
          BEQ     NEWTSK            ;IF TASK STATUS = NEW THEN NEWTSK
;
; RETURN TO ACTIVE TASK
;
          PULA
          ANDA    #0EFH             ; CLEAR INTERRUPT MASK BIT IN CC REG
          PSHA
          RTI                       ;RETURN TO TASK (WITH IRQ ENABLED)
;
; ACTIVATE NEW TASK
;
NEWTSK:   LDX     TSKPTR
          LDAA    #TSKACT
          STAA    TSKSTS,X          ;TASK STATUS = ACTIVE
;
          LDX     TSKADR,X          ;X = ADDRESS OF TASK
          NOP
          CLI                       ;ENABLE INTERRUPTS
          JSR     0,X               ;ACTIVATE TASK
;
; DEACTIVATE ACTIVE TASK
;
ENDTSK:   NOP
          SEI                       ;DISABLE INTERRUPTS
;
          LDX     TSKPTR
          LDAA    #TSKSTP
          STAA    TSKSTS,X          ;TASK STATUS = STOP
;
          STS     TSKSTK,X          ;TASK STACK = SP
;
          BRA     NXTTSK            ;SCHEDULE NEXT TASK
```

```
;*****
;
; USER DEFINED CODE FOR TASK ONE
;
;*****
CODE1:
        JSR     SYSTEM                  ;SCHEDULE OTHER TASKS
        JMP     CODE1                   ;DO TASK AGAIN
;*****
;
; USER DEFINED CODE FOR TASK TWO
;
;*****
CODE2:
        RTS                             ;ASSUME TASK COMPLETE
;*****
;
; USER DEFINED CODE FOR TASK THREE
; THIS EXAMPLE TASK INITIATES TASK 2 IF IT IS INACTIVE
;
;*****
CODE3:
        LDAA    TASK2+TSKSTS
        CMPA    #TSKSTP
        BNE     C3SKP                   ;IF TASK2 STATUS ≠ TSKSTP THEN SKIP
        LDAA    #TSKNEW
        STAA    TASK2+TSKSTS            ;TASK2 STATUS = TSKNEW
C3SKP:  JSR     SYSTEM                  ;SCHEDULE OTHER TASKS
        JMP     CODE3
        END
```

APPENDIX E: 6809 MULTI-TASKING KERNEL

The following code sequence illustrates a 6809 multi-tasking kernel.

```
            ASEG
;******
;
; TASK RECORD DEFINITION
;
;******
            ORG     0
TSKSTS:     DS      1                   ;TASK STATUS (STOP,NEW,ACTIVE)
TSKADR:     DS      2                   ;TASK ADDRESS (STARTING ADDRESS)
TSKSTK:     DS      2                   ;TASK STACK (ASSOCIATED STACK)
TSKLNK:     DS      2                   ;TASK LINK (LINK TO NEXT TASK)
TSKSZE      EQU     *-TSKSTS
;
TSKSTP      EQU     0                   ;TASK STATUS IS STOP
TSKNEW      EQU     1                   ;TASK STATUS IS NEW
TSKACT      EQU     2                   ;TASK STATUS IS ACTIVE
;******
;
; TASK VARIABLES AND TASK BLOCKS
;
;******
            ORG     0100H
TSKPTR:     DS      2                   ;TASK POINTER
TASK1:      DS      TSKSZE              ;TASK ONE
TASK2:      DS      TSKSZE              ;TASK TWO
TASK3:      DS      TSKSZE              ;TASK THREE
;******
;
; USER DEFINED STACKS FOR ASSOCIATED TASKS
;
;******
            DS      59
STACK1:     DS      1                   ;STACK ONE
            DS      59
STACK2:     DS      1                   ;STACK TWO
            DS      59
STACK3:     DS      1                   ;STACK THREE
;******
;
; TASK INITIALIZATION
;
;******
            ORG     8000H
            LDX     #CODE1              ;CODE FOR TASK ONE
            STX     TASK1+TSKADR
            LDX     #STACK1             ;STACK FOR TASK ONE
            STX     TASK1+TSKSTK
            LDA     #TSKNEW             ;NEW TASK
            STA     TASK1+TSKSTS
```

```
            LDX     #TASK2
            STX     TASK1+TSKLNK        ;LINK TO TASK2
;
            LDX     #CODE2              ;CODE FOR TASK TWO
            STX     TASK2+TSKADR
            LDX     #STACK2             ;STACK FOR TASK TWO
            STX     TASK2+TSKSTK
            LDA     #TSKNEW             ;NEW TASK
            STA     TASK2+TSKSTS
            LDX     #TASK3
            STX     TASK2+TSKLNK        ;LINK TO TASK3
;
            LDX     #CODE3              ;CODE FOR TASK THREE
            STX     TASK3+TSKADR
            LDX     #STACK3             ;STACK FOR TASK THREE
            STX     TASK3+TSKSTK
            LDA     #TSKNEW             ;NEW TASK
            STA     TASK3+TSKSTS
            LDX     #TASK1
            STX     TASK3+TSKLNK        ;LINK TO TASK1
;
            LDX     #TASK3
            STX     TSKPTR              ;TSKPTR = ADDRESS OF TASK3
            BRA     NXTTSK              ;START TASK 1
;******
;
; SYSTEM TASK SCHEDULER
; ENTERED VIA CALL FROM ACTIVE TASK
;
;******
SYSTEM:     ORCC    #01010000B          ;DISABLE INTERRUPTS
;
            PSHS    U,Y,X,DP,B,A,CC     ;SAVE TASK STATE
;******
;
; ENTRY POINT FOR INTERRUPT INITIATED TASK SWITCH
;
;******
TSKINT:     LDX     TSKPTR
            STS     TSKSTK,X            ;TASK STACK = SP
;
; NEXT TASK
;
NXTTSK:     LDX     TSKLNK,X
            STX     TSKPTR              ;TSKPTR = ADDRESS OF NEXT TASK
;
            LDA     TSKSTS,X
            CMPA    #TSKSTP
            BEQ     NXTTSK              ;IF TASK STATUS = STOP THEN NXTTSK
;
            LDS     TSKSTK,X            ;SP = TASK STACK
;
            CMPA    #TSKNEW
            BEQ     NEWTSK              ;IF TASK STATUS = NEW THEN NEWTSK
```

```
;
; RETURN TO ACTIVE TASK
;
            PULS    CC,A,B,DP,X,Y,U         ;RESTORE TASK STATE
;
            ANDCC   #10101111B              ;ENABLE INTERRUPTS
            RTS                             ;RETURN TO TASK
;
; ACTIVATE NEW TASK
;
NEWTSK:     LDX     TSKPTR
            LDA     #TSKACT
            STA     TSKSTS,X                ;TASK STATUS = ACTIVE
;
            ANDCC   #10101111B              ;ENABLE INTERRUPTS
            JSR     [TSKADR,X]              ;ACTIVATE TASK
;
; DEACTIVATE ACTIVE TASK
;
ENDTSK:     ORCC    #01010000B              ;DISABLE INTERRUPTS
;
            LDX     TSKPTR
            LDA     #TSKSTP
            STA     TSKSTS,X                ;TASK STATUS = STOP
;
            STS     TSKSTK,X                ;TASK STACK = SP
;
            BRA     NXTTSK                  ;SCHEDULE NEXT TASK
; * * * * *
;
; USER DEFINED CODE FOR TASK ONE
;
; * * * * *
CODE1:
            LBSR    SYSTEM                  ;SCHEDULE OTHER TASKS
            BRA     CODE1                   ;DO TASK AGAIN
; * * * * *
;
; USER DEFINED CODE FOR TASK TWO
;
; * * * * *
CODE2:
            RTS                             ;ASSUME TASK COMPLETE
```

```
; * * * * *
;
; USER DEFINED CODE FOR TASK THREE
; THIS EXAMPLE TASK INITIATES TASK 2 IF IT IS INACTIVE
;
; * * * * *
CODE3:
            LDA     TASK2+TSKSTS
            CMPA    #TSKSTP
            BNE     C3SKP               ;IF TASK2 STATUS ≠ TSKSTP THEN SKIP
            LDA     #TSKNEW
            STA     TASK2+TSKSTS        ;TASK2 STATUS = TSKNEW
C3SKP:      LBSR    SYSTEM              ;SCHEDULE OTHER TASKS
            BRA     CODE3
            END
```