TEXAS INSTRUMENTS

9900

CONCURRENCY WITH MICROPROCESSOR PASCAL

MICROPROCESSOR SERIES™

Application Report

PREFACE

The following Texas Instruments' publications were used in the development of this application report and present additional information relative to process management in the Microprocessor Pascal System. These publications may be ordered from a TI Sales Office or authorized distribution. A complete listing of TI Sales Offices and Distributors is provided in the last section of this document.

- The Microprocessor Pascal System User's Manual, MP351
- Software Development Handbook, MPA29

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

INTRODUCTION

The Microprocessor Pascal System is a complete Pascal development system designed for target execution on a wide spectrum of Texas Instruments' microcomputer-based products; from a 9900 microprocessor chip set to TM990 microcomputer modules to 990 minicomputers. Microprocessor PASCAL is a superset of Wirth's Pascal language offering its users such language extensions as concurrent task execution, expanded input/output capability, and library utilities for microprocessor applications.

The features of Microprocessor Pascal supporting concurrent programming provide greater programming efficiency through improved utilization of the processor. In conventional programming languages, execution proceeds sequentially from instruction to instruction with branches to and from subroutines as program logic dictates. Action takes place within one site of execution or program within the run-time environment. In the Microprocessor Pascal environment, multiple sites of executions exist as separate "processes". Action moves from process to process as CPU time is shared on a priority basis.

The execution of several processes in a single system is termed "multiprogramming" (also called multitasking). Multiprogramming proves extremely useful in the development of process control applications that deal to a large degree with asynchronous occurrences of events. Each independent asynchronous activity can be managed by a separate software module (process). This one-to-one correspondence between external activities and software processes facilitates a modular approach to system development that greatly enhances problem identification, application design, problem resolution, and solution reliability.

Each process in the Microprocessor Pascal run-time environment is in one of two states:

1) ready to execute (This "ready" state includes active processes or the process that is currently executing.)

2) suspended (blocked) and waiting for a condition in the system to change (an event to occur) before it can become ready to execute.

Processes that are ready to execute reside in a ready queue. A priority scheduling policy is used to order processes as they are placed in queue. The most urgent process is placed first and is the active process. The last process in the queue is the IDLE process which is activated when no other process is in the ready state. This scheduling policy switches the attention of the processor from one

ready process to the next. This switching is transparent to the user and results in an interleaving of process execution; all active processes appear to execute concurrently.

Processes that are suspended, waiting for an event to occur, must be notified when that event has taken place. The Executive Run-Time Support uses semaphores for that purpose. A semaphore can be envisioned as representing some event on which processes synchronize. Use of a semaphore can be loosely compared to the role of a signalman in a traffic situation. The signalman waves a flag (performs a signal) when its clear for a car to proceed. A car waits for the signalman to wave his flag (performs a wait) before proceeding. In the above example, synchronization is based on the waving of a flag (event); this synchronization provides for the sharing of a common resource (i.e., the road). The scenario is now changed as follows: The signalman periodically waves his flag regardless of whether or not a car is waiting to proceed. Also, a counter keeps track of each occurrence of flag waving when there is no car to receive the signal. As a car approaches, it checks the counter. If the counter indicates that the signal was given, it proceeds. With the above modification, our example more closely resembles the action of a semaphore in the Microprocessor Pascal Environment.

The information presented above briefly introduces tools used by the Executive Run Time Support to manage processes in the Microprocessor Pascal Environment. In the following sections, the scheduling of the processor and process synchronization are discussed in more detail.

The Executive Run-Time Support (RTS) scheduling policy determines the assignment of the processor to one of several ready processes. Ready processes are inserted in the ready queue and scheduled for execution according to priority.

A process' priority is represented by a user-assigned numeric value. The greatest urgency is represented by 0; the least by 32766, which is reserved for the IDLE process. (IDLE is active only when all other processes in the system are checked.) Integer values up to 15 indicate device processes associated with interrupts. (Interrupts can occur due to a change in some "real world" condition or because they are programmed to occur to prevent a process from executing longer than its time slice. Interrupt handlers are usually time-critical and demand immediate execution.) Integer values greater than 15 represent non-device processes.

A scheduling decision is made by the Executive RTS each time a suspended process becomes ready and a ready process terminates or becomes suspended. (An explanation of process readiness follows this discussion of scheduling.) When the active process (at the top of the queue) terminates execution (or becomes suspended) the next process in the queue becomes the active process. Because the ready queue is ordered by priority, the most urgent process that is ready is given the processor. When a suspended process becomes ready, it is inserted in the ready queue based on its priority. The newly ready process preempts the currently active process (i.e., is placed in front of it in the ready queue) if it is more urgent. Non-device processes that become ready are placed in queue behind processes of equal priority. (When two processes have equal priority, the process that has been ready the longest executes first.) Device processes are placed in front of other processes of equal priority including the active process. Figure 1 illustrates the working of this scheduling policy.

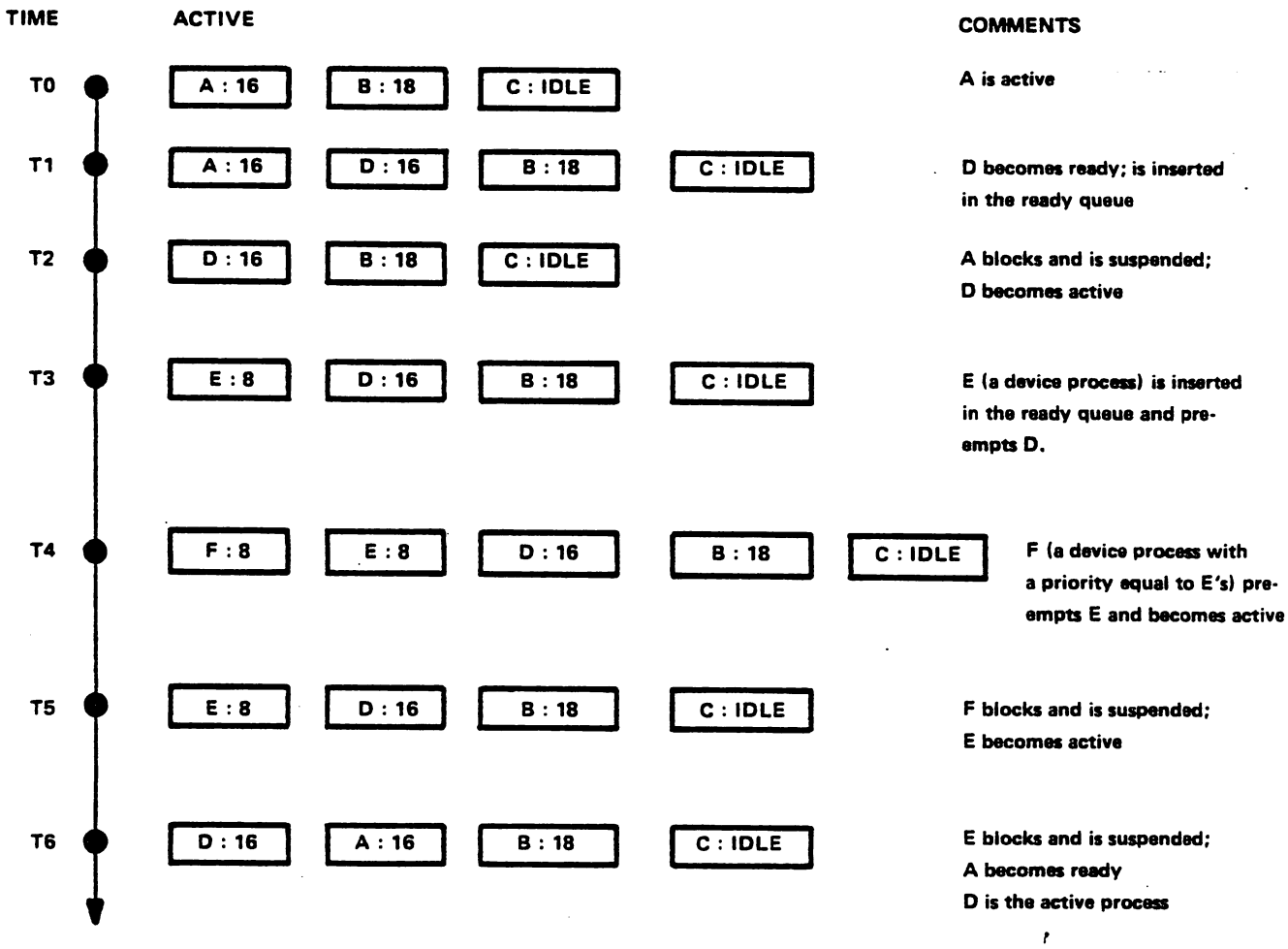

FIGURE 1 – SCHEDULING POLICY

In Figure 1 above, the ready queue is represented as a horizontal series of boxes. Each process (box) is labeled with a letter and a priority number. The first box in the ready queue is the active process. Time moves vertically from top to bottom. Comments to the right of each queue describe the action performed.

The execution of the RTS scheduling policy displayed in Figure 1 results in process "B" never becoming active. In fact, B will never become active unless all other processes in queue with greater urgency become blocked or terminate execution. A process of higher urgency that becomes ready will always interrupt the (currently) active process. Once the more urgent process terminates (or becomes blocked) the previously active process will resume execution (unless another higher priority process becomes ready). This "preemptive scheduling with resumption" is designed for event-driven systems in which the event is some real-world occurrence that demands the immediate

attention of the computer. (Texas Instruments' Software Development Handbook discusses system design for monitoring and controlling "real-world" actions).

An RTS utility is available to swap non-device processes (with priority values greater than 15) in the ready queue. This "swap procedure" removes the first non-device process from the queue and inserts it behind the last process with the same priority. Figure 2 illustrates this.

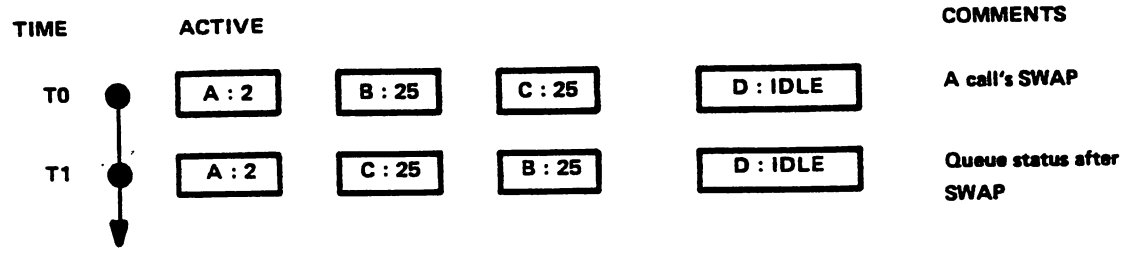

FIGURE 2 – SWAP PROCEDURE

A use of SWAP is to implement time slicing. SWAP is called to force a non-device process that is active to relinquish the processor. This swapping of the active process prevents it from running longer than its user specified execution time (time slice).

Up to this point, the discussion has been concerned with the management of processes that are ready for execution. Processes become suspended or blocked because of a condition in the system. When another process signals that the condition has changed, the waiting process can become ready. The mechanics of this process synchronization are described in the next section.

The semaphore is a Microprocessor Pascal language construct that is the fundamental mechanism for synchronization of processes via the Executive RTS, and can be thought of as representing some event on which processes synchronize. A process that is dependent on the occurence of an event can perform a WAIT to ensure that the event has occured before continuing execution. If the event has already occurred, the process executes; if not, it is suspended in that semaphore's queue until the event does occur. A SIGNAL operation performed on the associated semaphore allows a process to signal the occurence of an event. If some process is waiting for the event, it is made ready for execution; the process is removed from that semaphore's WAIT queue and inserted into the ready queue. If no process is waiting, the occurence of the event is recorded in the semaphore until a WAIT operation occurs for that event. (In both of these cases, the process that called SIGNAL remains in the active state.) The semaphores of the Executive RTS can be thought of as "counting" semaphores in that an occurence of an event is never lost, even if no process is waiting when the event occurs; a count is kept in the semaphore of all events that occurred (by SIGNAL) but were not received (by WAIT).

Microprocessor Pascal predefines semaphores as structures composed of two elements:

1) A non-negative counter of unserviced events, and

2) A queue (possibly empty) of suspended processes. In this queue, processes are made ready on a first-in first-out (FIFO) basis.

A semaphore is operated on by two primitive operations, WAIT and SIGNAL. These operations are implemented as routines, but are executed as though they were single machine instructions. Until these operations have completed, nothing must access the semaphore, the queues, or the operations themselves. This is assured when the interrupt mask is set to zero upon entry to the routines, and reset to its previous state upon exit.

WAIT decrements the counter if it is non-zero; or if it is zero, suspends the currently active process (the process is moved from the ready queue to the semaphore queue). SIGNAL increments the counter if the semaphore queue is empty; or if it is not empty, activates the first process in the queue (which will always be the process that has been in the queue the longest). This activation consists of moving the

first process from the semaphore queue to the ready queue.

When semaphores are used to ensure exclusive access to two or more resources, extreme caution must be exercised to prevent a condition known as "deadlock". This takes place when a situation is created in which two or more processes are suspended, awaiting a condition that cannot happen because there is no active process to cause the needed event to occur.

For example, if two simultaneously executing processes (A and B) both require exclusive access to resources (X and Y), the following sequence will result:

```
A gets X  ..  A requests Y
B gets Y  ..  B requests X
```

In the above example, neither A nor B will ever resume execution, as A will be waiting for Y (which B has) and B will be waiting for X (which A has). To prevent a situation such as this, either and/or both processes must check the availability of succeeding resources and, if unavailable, release those already acquired.

The listing on the following page is an example of process synchronization at work. This procedure receives a message from a mailbox and removes it from the queue. "M" is a pointer that (upon return from RCVMSG) points to the received message, and ADDRESSEE is a pointer to the mailbox from which the message is to be received. If no messages exist at call time, the process is suspended until a message is entered into the mailbox queue (via SNDMSG).

```
0 (* MAP, DEBUG *)
0
0 PROGRAM MAILBOXES;
0
0 CONST
0   MSGSIZ = 80;
0
0 TYPE
0   MSGPTR = @MSG;
0   MBPTR  = @MAILBOX;
0
0   MSG     = RECORD
0           NEXTMSG : MSGPTR;
0           RESPONSE : SEMAPHORE;
0           MSGSIZE : INTEGER;
0           CMD : ( R, W );
0           MSGTEXT : PACKED ARRAY (.1..MSGSIZ.) OF CHAR;
0           END;
0
0 MAILBOX = RECORD
0         MAILPRESENT, MUTEX : SEMAPHORE;
0         MSGHEAD, MSGTAIL : MSGPTR;
0         END;
0
0 PROCEDURE SIGNAL ( S : SEMAPHORE ); EXTERNAL;
0 PROCEDURE WAIT   ( S : SEMAPHORE ); EXTERNAL;
0 PROCEDURE INITSEMAPHORE ( VAR S : SEMAPHORE ; VALUE : INTEGER);
4           EXTERNAL;
0 PROCEDURE TERMSEMAPHORE ( VAR S : SEMAPHORE ); EXTERNAL;
2
0 PROCEDURE SNDMSG ( M : MSGPTR ; ADDRESSEE : MBPTR ); FORWARD;
0 PROCEDURE RCVMSG ( VAR M : MSGPTR ; ADDRESSEE : MBPTR ); FORWAR
0 PROCEDURE DELMSG ( M : MSGPTR ; ADDRESSEE : MBPTR ); FORWARD;
0 PROCEDURE SNDMSG (* M : MSGPTR ; ADDRESSEE : MSPTR *);
4 (*-------------------------------------------------------
4 PURPOSE:
4   ENTER A MESSAGE INTO A MAILBOX QUEUE.
4 INPUTS:
4   M         : POINTER TO THE MESSAGE.
4   ADDRESSEE : POINTER TO THE MAILBOX.
4 PROCEDURES CALLED:
4   WAIT, SIGNAL.
4 OUTPUTS:
4   NONE.
4 EXCEPTIONS:
4   NONE.
4 HISTORY:
4   04/08/79: ORIGINAL.
4 -------------------------------------------------------
4
1 BEGIN
1   WITH M@ DO BEGIN
2     NEXTMSG := NIL;
3     WITH ADDRESSEE@ DO BEGIN
4       WAIT (MUTEX );
```

```
 5        IF MSGHEAD <> NIL
 6          THEN MSGTAIL@.NEXTMSG := M
 7          ELSE MSGHEAD := M;
 8        MSGTAIL := M ;
 9        SIGNAL ( MAILPRESENT );
10        SIGNAL ( MUTEX );
11      END; (* WITH ADDRESSEE@ *)
11    END; (* WITH M@ *)
11  END; (* SNDMSG *)
11
 0  PROCEDURE RCVMSG (* VAR M : MSGPTR ; ADDRESSEE : MBPTR *);
 4  (*-----------------------------------------------------------
 4  PURPOSE:
 4    RECEIVE A MESSAGE FROM A MAILBOX AND REMOVE IT FROM THE QUEUE
 4  INPUTS:
 4    ADDRESSEE: POINTER TO THE MAILBOX FROM WHICH A MESSAGE IS
 4    TO BE RECEIVED.
 4  PROCEDURES CALLED:
 4    WAIT, SIGNAL.
 4  OUTPUTS:
 4    M : A POINTER TO THE RECEIVED MESSAGE.
 4  EXCEPTIONS:
 4    THE PROCESS IS SUSPENDED UNTIL A MESSAGE IS ENTERED INTO THE
 4    QUEUE IF NONE EXIST AT CALL TIME.
 4  HISTORY:
 4    04/08/79:  ORIGINAL.
 4  -----------------------------------------------------------*)
 4
 1  BEGIN
 1    WITH ADDRESSEE@ DO BEGIN
 2      WAIT ( MAILPRESENT );
 3      WAIT ( MUTEX );
 4      M := MSGHEAD;
 5      IF MSGTAIL = MSGHEAD
 6        THEN MSGTAIL := NIL;
 7      MSGHEAD := M@.NEXTMSG;
 8      SIGNAL ( MUTEX );
 9    END; (* WITH ADDRESSEE@ *)
 9  END; (* RCVMSG *)
 9
 0  PROCEDURE DELMSG (* M : MSGPTR ; ADDRESSEE : MBPTR *);
 4  (*-----------------------------------------------------------
 4  PURPOSE:
 4    DELETE A MESSAGE FROM A MAILBOX QUEUE.
 4  INPUTS:
 4    M          : POINTER TO THE MESSAGE TO BE DELETED.
 4    ADDRESSEE : POINTER TO THE MAILBOX CONTAINING THE QUEUE
 4    FROM WHICH THE MESSAGE IS TO BE DELETED.
 4  PROCEDURES CALLED:
 4    WAIT, SIGNAL.
 4  OUTPUTS:
 4    NONE.
 4  EXCEPTIONS:
 4    NONE.
 4  HISTORY:
```

```
4    04/08/79: ORIGINAL.
4    ------------------------------------------------------------
4
4  VAR
4    LAST   : MSGPTR;
6    FOUND  : BOOLEAN;
8
1  BEGIN
1    FOUND := FALSE;
2    WITH ADDRESSEE@ DO BEGIN
3      WAIT ( MUTEX );
4      IF MSGHEAD = M
5        THEN BEGIN
5          MSGHEAD := M@.NEXTMSG;
6          FOUND := TRUE;
7        END (* IF MSGHEAD THEN *)
7        ELSE BEGIN
7          LAST := MSGHEAD;
8          WHILE LAST <> NIL AND LAST@.NEXTMSG <> M
9            DO LAST := LAST@.NEXTMSG;
10         IF LAST<>NIL THEN BEGIN;
12           LAST@.NEXTMSG := M@.NEXTMSG;
13           FOUND := TRUE;
14           IF LAST@.NEXTMSG = NIL
15             THEN MSGTAIL := LAST;
16           END; (* IF LAST *);
16         END; (* IF MSGHEAD ELSE *)
16       IF FOUND
17         THEN WAIT ( MAILPRESENT );
18       SIGNAL ( MUTEX )
19     END; (* WITH ADDRESSEE@ *)
19   END; (* DELMSG *)
19
 1 BEGIN (* MAILBOXES *)
 1   (* NULLBODY *)
 1 END.   (* MAILBOXES *)
```

CONCLUSION

The software tools discussed in this document provide interprocess scheduling and coordination of system resources in the Executive Run-Time Environment. While simple in terms of language constructs, these tools provide a sophistication in capability that enables a higher productivity in the real-time programming environment. Further information on these and other components of the Microprocessor Pascal System can be obtained in the Microprocessor Pascal System User's Guide and the TMS9900 Family Software Development Handbook.

TEXAS INSTRUMENTS
INCORPORATED

Post Office Box 1443 • Houston, Texas 77001
Semiconductor Group

Printed in U.S.A.