priorities, manage memory, etc. At the lowest level would be the modules that implement primitive operations such as adding and removing tasks or messages from lists, servicing timer interrupts, etc.

Interrupt Service Procedures

Procedures that service external interrupts should be considered differently than those that service internal interrupts. A service procedure that is activated by an internal interrupt, may, and often should, be made reentrant. External interrupt procedures, on the other hand, should be viewed as temporary tasks. In this sense, a task is a single sequential thread of execution; it should not be reentered. The processor's response to an external interrupt may be viewed as the following sequence of events:

- the running (active) task is suspended,
- a new task, the interrupt service procedure, is created and becomes the running task,
- the interrupt task ends, and is deleted,
- the suspended task is reactived and becomes the running task from the point where it was suspended.

An external interrupt procedure should only be interruptable by a request that activates a different interrupt procedure. When the number of interrupt sources is not too large, this can be accomplished by assigning a different type code and corresponding service procedure to each source. In systems where a large number of similar sources can generate closely spaced interrupts (e.g., 500 communication lines), an approach similar to that illustrated in figure 2-70, may be used to insure that the interrupt service procedure is not reentered, and yet, interrupts arriving in bursts are not missed. The basic technique is to divide the code required to service an interrupt into two parts. The interrupt service procedure itself is kept as short as possible; it performs the absolute minimum amount of processing necessary to service the device. It then builds a message that contains enough information to permit another task, the interrupt message processor, to complete the interrupt service. It adds the message to a queue (which might be implemented as a linked list), and terminates so that it is available to service the next interrupt. The interrupt message processor, which is not reentrant, obtains a message from the queue, finishes processing the interrupt associated with that message, obtains the next message (if there is one), etc. When a burst of interrupts occurs, the queue will lengthen, but interrupts will not be missed so long as there is time for the interrupt service procedure to be activated and run between requests.

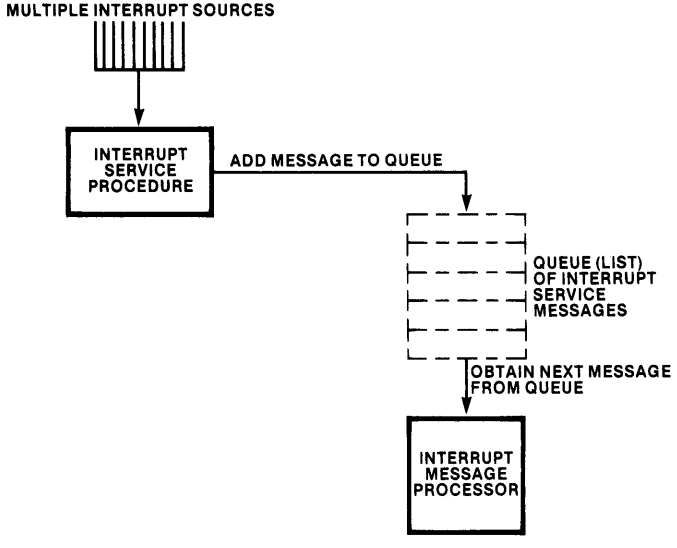

Figure 2-70. Interrupt Message Processor

Stack-Based Parameters

Parameters are frequently passed to procedures on a stack. Results produced by the procedure, however, should be returned in other memory locations or in registers. In other words, the called procedure should "clean up" the stack by discarding the parameters before returning. The RET instruction can perform this function. PL/M-86 procedures always follow this convention.

Flag-Images

Programs should make no assumptions about the contents of the undefined bits in the flag-images stored in memory by the PUSHF and SAHF instructions. These bits always should be masked out of any comparisons or tests that use these flag-images. The undefined bits of the word flag-image can be cleared by ANDing the word with FD5H. The undefined bits of the byte flag-image can be cleared by ANDing the byte with D5H.

Programming Examples

These examples demonstrate the 8086/8088 instruction set and addressing modes in common programming situations. The following topics are addressed:

- procedures (parameters, reentrancy)
- various forms of JMP and CALL instructions
- bit manipulation with the ASM-86 RECORD facility
- dynamic code relocation
- memory mapped I/O
- breakpoints
- interrupt handling
- string operations

These examples are written primarily in ASM-86 and will be of most interest to assembly language programmers. The PL/M-86 compiler generates code that handles many of these situations automatically for PL/M-86 programs. For example, the compiler takes care of the stack in PL/M-86 procedures, allowing the programmer to concentrate on solving the application problem. PL/M-86 programmers, however, may want to examine the memory mapped I/O and interrupt handling examples, since the concepts illustrated are generally applicable; one of the interrupt procedures is written in PL/M-86.

The examples are intended to show one way to use the instruction set, addressing modes and features of ASM-86. They do not demonstrate the "best" way to solve any particular problem. The flexibility of the 8086 and 8088, application differences plus variations in programming style usually add up to a number of ways to implement a programming solution.

Procedures

The code in figure 2-71 illustrates several techniques that are typically used in writing ASM-86 procedures. In this example a calling program invokes a procedure (called EXAMPLE) twice, passing it a different byte array each time. Two parameters are passed on the stack; the first contains the number of elements in the array, and the second contains the address (offset in DATA_SEG) of the first array element. This same technique can be used to pass a variable-length parameter list to a procedure (the "array" could be any series of parameters or parameter addresses). Thus, although the procedure always receives two parameters, these can be used to indirectly access any number of variables in memory.

Any results returned by a procedure should be placed in registers or in memory, but not on the stack. AX or AL is often used to hold a single word or byte result. Alternatively, the calling program can pass the address (or addresses) of a result area to the procedure as a parameter. It is good practice for ASM-86 programs to follow the calling conventions used by PL/M-86; these are documented in *MCS-86 Assembler Operating Instructions For ISIS-II Users*, Order No. 9800641.

EXAMPLE is defined as a FAR procedure, meaning it is in a different segment than the calling program. The calling program must use an intersegment CALL to activate the procedure. Note that this type of CALL saves CS and IP on the stack. If EXAMPLE were defined as NEAR (in the same segment as the caller) then an intra-segment CALL would be used, and only IP would be saved on the stack. It is the responsibility of the calling program to know how the procedure is defined and to issue the correct type of CALL.

```
STACK_SEG       SEGMENT
                DW          20 DUP (?)      ; ALLOCATE 20-WORD STACK

STACK_TOP       LABEL       WORD            ; LABEL INITIAL TOS
STACK_SEG       ENDS

DATA_SEG        SEGMENT
ARRAY_1         DB          10 DUP (?)      ; 10-ELEMENT BYTE ARRAY

ARRAY_2         DB          5 DUP (?)       ; 5-ELEMENT BYTE ARRAY

DATA_SEG        ENDS


PROC_SEG        SEGMENT
ASSUME CS:PROC_SEG,DS:DATA_SEG,SS:STACK_SEG,ES:NOTHING

EXAMPLE         PROC        FAR             ; MUST BE ACTIVATED BY
                                            ;     INTERSEGMENT CALL

; PROCEDURE PROLOG
                PUSH        BP              ; SAVE BP
                MOV         BP, SP          ; ESTABLISH BASE POINTER
                PUSH        CX              ; SAVE CALLER'S
                PUSH        BX              ;      REGISTERS
                PUSHF                       ;      AND FLAGS
                SUB         SP, 6           ; ALLOCATE 3 WORDS LOCAL STORAGE
                ; END OF PROLOG
; PROCEDURE BODY
                MOV         CX, [BP+8]      ; GET ELEMENT COUNT
                MOV         BX, [BP+6]      ; GET OFFSET OF 1ST ELEMENT
                ; PROCEDURE CODE GOES HERE
                ; FIRST PARAMETER CAN BE ADDRESSED:
                ;   [BX]
                ; LOCAL STORAGE CAN BE ADDRESSED:
                ;       [BP-8], [BP-10], [BP-12]
                ; END OF PROCEDURE BODY
; PROCEDURE EPILOG
                ADD         SP, 6           ; DE-ALLOCATE LOCAL STORAGE
                POPF                        ; RESTORE CALLER'S
                POP         BX              ;    REGISTERS
                POP         CX              ;    AND
                POP         BP              ;    FLAGS
                ; END OF EPILOG
; PROCEDURE RETURN
                RET         4               ; DISCARD 2 PARAMETERS

EXAMPLE         ENDP                        ; END OF PROCEDURE "EXAMPLE"

PROC_SEG        ENDS
```

Figure 2-71. Procedure Example 1

```
CALLER_SEG      SEGMENT
; GIVE ASSEMBLER SEGMENT/REGISTER CORRESPONDENCE
ASSUME          CS:CALLER_SEG,
&               DS:DATA_SEG,
&               SS:STACK_SEG,
&               ES:NOTHING              ; NO EXTRA SEGMENT IN THIS PROGRAM

; INITIALIZE SEGMENT REGISTERS
START:          MOV     AX,DATA_SEG
                MOV     DS,AX
                MOV     AX,STACK_SEG
                MOV     SS,AX
                MOV     SP,OFFSET STACK_TOP  ; POINT SP TO TOS

; ASSUME ARRAY_1 IS INITIALIZED
;
; CALL "EXAMPLE", PASSING ARRAY_1, THAT IS, THE NUMBER OF ELEMENTS
;   IN THE ARRAY, AND THE LOCATION OF THE FIRST ELEMENT.
                MOV     AX,SIZE ARRAY_1
                PUSH    AX
                MOV     AX,OFFSET ARRAY_1
                PUSH    AX
                CALL    EXAMPLE

; ASSUME ARRAY_2 IS INITIALIZED
;
; CALL "EXAMPLE" AGAIN WITH DIFFERENT SIZE ARRAY.
                MOV     AX,SIZE ARRAY_2
                PUSH    AX
                MOV     AX,OFFSET ARRAY_2
                PUSH    AX
                CALL    EXAMPLE
CALLER_SEG              ENDS

                END     START
```

Figure 2-71. Procedure Example 1 (Cont'd.)

Figure 2-72 shows the stack before the caller pushes the parameters onto it. Figure 2-73 shows the stack as the procedure receives it after the CALL has been executed.

EXAMPLE is divided into four sections. The "prolog" sets up register BP so it can be used to address data on the stack (recall that specifying BP as a base register in an instruction automatically refers to the stack segment unless a segment override prefix is coded). The next step in the prolog is to save the "state of the machine" as it existed when the procedure was activated. This is done by pushing any registers used by the procedure (only CX and BP in this case) onto the stack. If the procedure changes the flags, and the caller expects the flags to be unchanged following execution of the procedure, they also may be saved on the stack. The last instruction in the prolog allocates three words on the stack for the procedure to use as local temporary storage. Figure 2-74 shows the stack at the end of the prolog. Note that PL/M-86 procedures assume that all registers except SP and BP can be used without saving and restoring.

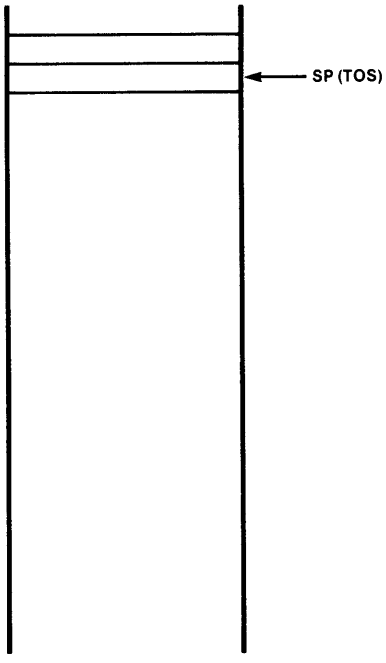

Figure 2-72. Stack Before Pushing Parameters

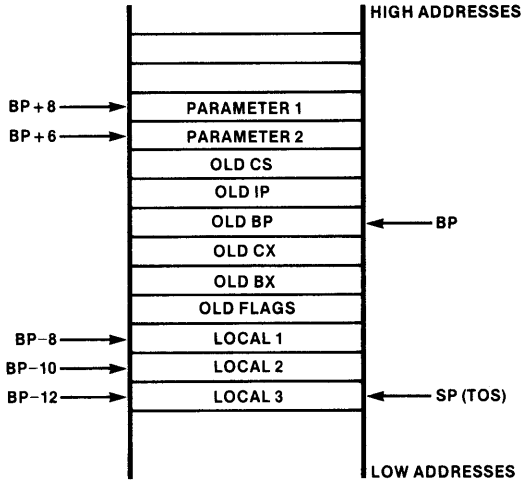

Figure 2-74. Stack Following Procedure Prolog

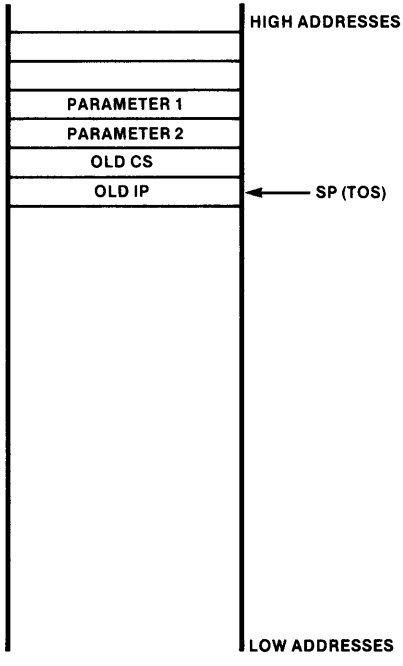

Figure 2-73. Stack at Procedure Entry

The procedure "body" does the actual processing (none in the example). The parameters on the stack are addressed relative to BP. Note that if EXAMPLE were a NEAR procedure, CS would not be on the stack and the parameters would be two bytes "closer" to BP. BP also is used to address the local variables on the stack. Local constants are best stored in a data or extra segment.

The procedure "epilog" reverses the activities of the prolog, leaving the stack as it was when the procedure was entered (see figure 2-75).

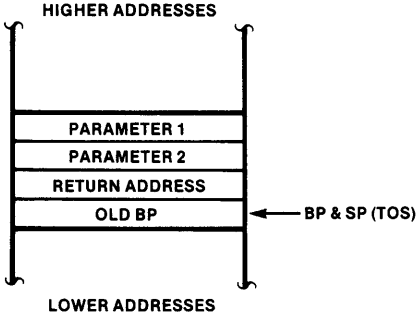

Figure 2-75. Stack Following Procedure Epilog

The procedure "return" restores CS and IP from the stack and discards the parameters. As figure 2-76 shows, when the calling program is resumed, the stack is in the same state as it was before any parameters were pushed onto it.

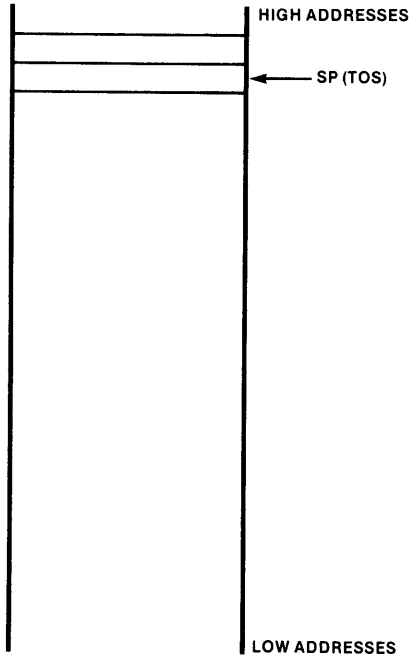

Figure 2-76. Stack Following Procedure Return

Figure 2-77 shows a simple procedure that uses an ASM-86 structure to address the stack. Register BP is pointed to the base of the structure, which is the top of the stack since the stack grows toward lower addresses (see figure 2-78). Any structure element can then be addressed by specifying BP as a base register:

```
[BP].structure_element.
```

Figure 2-79 shows a different approach to using an ASM-86 structure to define the stack layout. As shown in figure 2-80, register BP is pointed at the middle of the structure (at OLD_BP) rather than at the base of the structure. Parameters and the return address are thus located at positive displacements (high addresses) from BP, while local variables are at negative displacements (lower addresses) from BP. This means that the local variables will be "closer" to the beginning of the stack segment and increases the likelihood that the assembler will be able to produce shorter instructions to access these variables, i.e., their offsets from SS may be 255 bytes or less and can be expressed as a 1-byte value rather than a 2-byte value. Exit from the subroutine also is slightly faster because a MOV instruction can be used to deallocate the local storage instead of an ADD (compare figure 2-71).

It is possible for a procedure to be activated a second time before it has returned from its first activation. For example, procedure A may call procedure B, and an interrupt may occur while procedure B is executing. If the interrupt service procedure calls B, then procedure B is *reentered* and must be written to handle this situation correctly, i.e., the procedure must be made reentrant.

In PL/M-86 this can be done by simply writing:

```
B: PROCEDURE (PARM1, PARM2) REENTRANT;
```

An ASM-86 procedure will be reentrant if it uses the stack for storing all local variables. When the procedure is reentered, a new "generation" of variables will be allocated on the stack. The stack will grow, but the sets of variables (and the parameters and return addresses as well) will automatically be kept straight. The stack must be large enough to accommodate the maximum "depth" of procedure activation that can occur under actual running conditions. In addition, any procedure called by a reentrant procedure must itself be reentrant.

A related situation that also requires reentrant procedures is recursion. The following are examples of recursion:

- A calls A (direct recursion),
- A calls B, B calls A (indirect recursion),
- A calls B, B calls C, C calls A (indirect recursion).

```
CODE            SEGMENT
                ASSUME CS:CODE
MAX             PROC
; THIS PROCEDURE IS CALLED BY THE FOLLOWING
;       SEQUENCE:
;           PUSH PARM1
;           PUSH PARM2
;           CALL MAX
; IT RETURNS THE MAXIMUM OF THE TWO WORD
;   PARAMETERS IN AX.

; DEFINE THE STACK LAYOUT AS A STRUCTURE.
STACK_LAYOUT STRUC
OLD_BP          DW ?        ; SAVED BP VALUE—BASE OF STRUCTURE
RETURN_ADDR     DW ?        ; RETURN ADDRESS
PARM_2          DW ?        ; SECOND PARAMETER
PARM_1          DW ?        ; FIRST PARAMETER
STACK_LAYOUT ENDS

; PROLOG
                PUSH    BP              ; SAVE IN OLD_BP
                MOV     BP, SP          ; POINT TO OLD_BP
; BODY
                MOV     AX, [BP].PARM_1 ; IF FIRST
                CMP     AX, [BP].PARM_2 ; > SECOND
                JG      FIRST_IS_MAX    ; THEN RETURN FIRST
                MOV     AX, [BP].PARM_2 ; ELSE RETURN SECOND
; EPILOG
FIRST_IS_MAX:   POP     BP              ; RESTORE BP (& SP)
; RETURN
                RET     4               ; DISCARD PARAMETERS
MAX             ENDP

CODE            ENDS
                END
```

Figure 2-77. Procedure Example 2

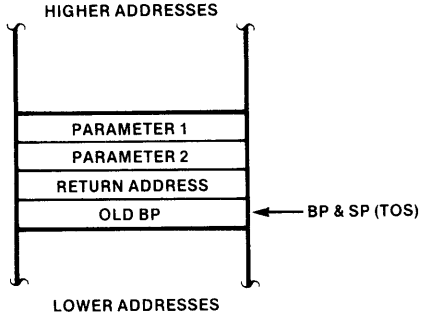

Figure 2-78. Procedure Example 2 Stack Layout

Jumps and Calls

The 8086/8088 instruction set contains many different types of JMP and CALL instructions (e.g., direct, indirect through register, indirect through memory, etc.). These varying types of transfer provide efficient use of space and execution time in different programming situations. Figure 2-81 illustrates typical use of the different forms of these instructions. Note that the ASM-86 assembler uses the terms "NEAR" and "FAR" to denote intrasegment and intersegment transfers, respectively.

```
EXTRA           SEGMENT
; CONTAINS STRUCTURE TEMPLATE THAT "NEARPROC"
;    USES TO ADDRESS AN ARRAY PASSED BY ADDRESS.
DUMMY           STRUC
PARM_ARRAY      DB          256 DUP ?
DUMMY           ENDS
EXTRA           ENDS

CODE            SEGMENT
                ASSUME CS:CODE,ES:EXTRA
NEARPROC        PROC
; LAY OUT THE STACK (THE DYNAMIC STORAGE AREA OR DSA).
DSASTRUC        STRUC
I               DW          ?               ; LOCAL VARIABLES FIRST
LOC_ARRAY       DW          10 DUP (?)      ;
OLD_BP          DW          ?               ; ORIGINAL BP VALUE
RETADDR         DW          ?               ; RETURN ADDRESS
POINTER         DD          ?               ; 2ND PARM—POINTER TO "PARM_ARRAY"
COUNT           DB          ?               ; 1ST PARM—A BYTE OCCUPIES
                DB          ?               ;     A WORD ON THE STACK
DSASTRUC        ENDS

; USE AN EQU TO DEFINE THE BASE ADDRESS OF THE
;    DSA. CANNOT SIMPLY USE BP BECAUSE IT WILL
;    BE POINTING TO "OLD_BP" IN THE MIDDLE OF
;    THE DSA.
DSA             EQU         [BP - OFFSET OLD_BP]

; PROCEDURE ENTRY
                PUSH        BP              ; SAVE BP
                MOV         BP, SP          ; POINT BP AT OLD_BP
                SUB         SP, OFFSET OLD_BP ; ALLOCATE LOC_ARRAY & I

; PROCEDURE BODY
                ; ACCESS LOCAL VARIABLE I
                MOV         AX,DSA.I

                ; ACCESS LOCAL ARRAY (3) I.E., 4TH ELEMENT
                MOV         SI,6            ; WORD ARRAY-INDEX IS 3*2
                MOV         AX,DSA.LOC_ARRAY [SI]

                ; LOAD POINTER TO ARRAY PASSED BY ADDRESS
                LES         BX,DSA.POINTER

                ; ES:BX NOW POINTS TO PARM_ARRAY (0)
                ; ACCESS SI'TH ELEMENT OF PARM_ARRAY
                MOV         AL,ES:[BX].PARM_ARRAY [SI]

                ; ACCESS THE BYTE PARAMETER
                MOV         AL,DSA.COUNT
```

Figure 2-79. Procedure Example 3

```
; PROCEDURE EXIT
                MOV     SP,BP           ; DE-ALLOCATE LOCALS
                POP     BP              ; RESTORE BP
                ; STACK NOW AS RECEIVED FROM CALLER
                RET     6               ; DISCARD PARAMETERS

NEARPROC        ENDP
CODE            ENDS
                END
```

Figure 2-79. Procedure Example 3 (Cont'd.)

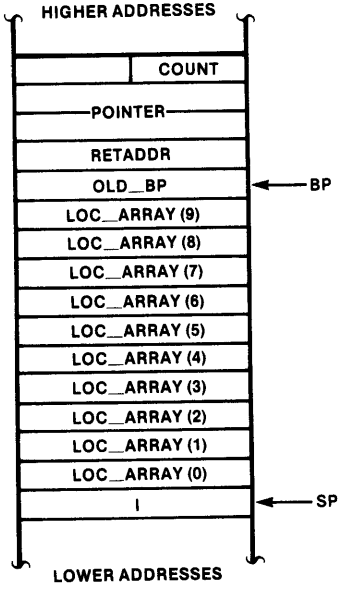

Figure 2-80. Procedure Example 3 Stack Layout

The procedure in figure 2-81 illustrates how a PL/M-86 DO CASE construction may be implemented in ASM-86. It also shows:

- an indirect CALL through memory to a procedure located in another segment,
- a direct JMP to a label in another segment,
- an indirect JMP though memory to a label in the same segment,
- an indirect JMP through a register to a label in the same segment,
- a direct CALL to a procedure in another segment,
- a direct CALL to a procedure in the same segment,
- direct JMPs to labels in the same segment, within −128 to +127 bytes ("SHORT") and farther than −128 to +127 bytes ("NEAR").

```
DATA            SEGMENT
; DEFINE THE CASE TABLE (JUMP TABLE) USED BY PROCEDURE
;       "DO_CASE." THE OFFSET OF EACH LABEL WILL
;       BE PLACED IN THE TABLE BY THE ASSEMBLER.
CASE_TABLE      DW          ACTION0, ACTION1, ACTION2,
&                           ACTION3, ACTION4, ACTION5
DATA            ENDS

; DEFINE TWO EXTERNAL (NOT PRESENT IN THIS
;       ASSEMBLY BUT SUPPLIED BY R & L FACILITY)
;       PROCEDURES. ONE IS IN THIS CODE SEGMENT
;       (NEAR) AND ONE IS IN ANOTHER SEGMENT (FAR).
                EXTRN       NEAR_PROC: NEAR, FAR_PROC: FAR

; DEFINE AN EXTERNAL LABEL (JUMP TARGET) THAT
;       IS IN ANOTHER SEGMENT.
                EXTRN       ERR_EXIT: FAR

CODE            SEGMENT
                ASSUME      CS: CODE, DS: DATA
; ASSUME DS HAS BEEN SET UP
;       BY CALLER TO POINT TO "DATA" SEGMENT.

DO_CASE         PROC        NEAR
; THIS EXAMPLE PROCEDURE RECEIVES TWO
;     PARAMETERS ON THE STACK. THE FIRST
;     PARAMETER IS THE "CASE NUMBER" OF
;     A ROUTINE TO BE EXECUTED (0-5). THE SECOND
;     PARAMETER IS A POINTER TO AN ERROR
;     PROCEDURE THAT IS EXECUTED IF AN INVALID
;     CASE NUMBER (>5) IS RECEIVED.

; LAY OUT THE STACK.
STACK_LAYOUT STRUC
OLD_BP          DW      ?
RETADDR         DW      ?
ERR_PROC_ADDR   DD      ?
CASE_NO         DB      ?
                DB      ?
STACK_LAYOUT ENDS

; SET UP PARAMETER ADDRESSING
                PUSH        BP
                MOV         BP, SP

; CODE TO SAVE CALLER'S REGISTERS COULD GO HERE.

; CHECK THE CASE NUMBER
                MOV         BH, 0
                MOV         BL, [BP].CASE_NO
                CMP         BX, LENGTH CASE_TABLE
                JLE         OK              ; ALL CONDITIONAL JUMPS
                                            ; ARE SHORT DIRECT
```

Figure 2-81. JMP and CALL Examples

```
; CALL THE ERROR ROUTINE WITH A FAR
;    INDIRECT CALL. A FAR INDIRECT CALL
;    IS INDICATED SINCE THE OPERAND HAS
;    TYPE "DOUBLEWORD."
                    CALL        [BP].ERR_PROC_ADDR

; JUMP DIRECTLY TO A LABEL IN ANOTHER SEGMENT.
;    A FAR DIRECT JUMP IS INDICATED SINCE
;    THE OPERAND HAS TYPE "FAR."
                    JMP         ERR_EXIT

OK:
; MULTIPLY CASE NUMBER BY 2 TO GET OFFSET
;     INTO CASE_TABLE (EACH ENTRY IS 2 BYTES).
                    SHL         BX, 1
; NEAR INDIRECT JUMP THROUGH SELECTED
;    ELEMENT OF CASE_TABLE. A NEAR
;    INDIRECT JUMP IS INDICATED SINCE THE
;    OPERAND HAS TYPE "WORD."
                    JMP         CASE_TABLE [BX]

ACTION0:            ; EXECUTED IF CASE_NO = 0
     ; CODE TO PROCESS THE ZERO CASE GOES HERE.
     ; FOR ILLUSTRATION PURPOSES, USE A
     ;    NEAR INDIRECT JUMP THROUGH A
     ;    REGISTER TO BRANCH TO THE POINT
     ;    WHERE ALL CASES CONVERGE.
     ;    A DIRECT JUMP (JMP ENDCASE) IS
     ;    ACTUALLY MORE APPROPRIATE HERE.
                    MOV         AX, OFFSET ENDCASE
                    JMP         AX

ACTION1:            ; EXECUTED IF CASE_NO = 1
     ; CALL A FAR EXTERNAL PROCEDURE. A FAR
     ;    DIRECT CALL IS INDICATED SINCE OPERAND
     ;    HAS TYPE "FAR."
                    CALL        FAR_PROC
     ; CALL A NEAR EXTERNAL PROCEDURE.
                    CALL        NEAR_PROC
     ; BRANCH TO CONVERGENCE POINT USING NEAR
     ;    DIRECT JUMP. NOTE THAT "ENDCASE"
     ;    IS MORE THAN 127 BYTES AWAY
     ;    SO A NEAR DIRECT JUMP WILL BE USED.
                    JMP         ENDCASE

ACTION2:            ; EXECUTED IF CASE_NO = 2
     ; CODE GOES HERE
                    JMP         ENDCASE     ; NEAR DIRECT JUMP
```

Figure 2-81. JMP and CALL Examples (Cont'd.)

```
ACTION3:        ; EXECUTED IF CASE_NO = 3
    ; CODE GOES HERE
                JMP         ENDCASE     ; NEAR DIRECT JMP

; ARTIFICIALLY FORCE "ENDCASE" FURTHER AWAY
;   SO THAT ABOVE JUMPS CANNOT BE "SHORT."
                ORG         500

ACTION4:        ; EXECUTED IF CASE_NO = 4
    ; CODE GOES HERE
                JMP         ENDCASE     ; NEAR DIRECT JUMP

ACTION5:        ; EXECUTED IF CASE_NO = 5
    ; CODE GOES HERE.
    ; BRANCH TO CONVERGENCE POINT USING
    ;   SHORT DIRECT JUMP SINCE TARGET IS
    ;   WITHIN 127 BYTES. MACHINE INSTRUCTION
    ;   HAS 1-BYTE DISPLACEMENT RATHER THAN
    ;   2-BYTE DISPLACEMENT REQUIRED FOR
    ;   NEAR DIRECT JUMPS. "SHORT" IS
    ;   WRITTEN BECAUSE "ENDCASE" IS A FORWARD
    ;   REFERENCE, WHICH ASSEMBLER ASSUMES IS
    ;   "NEAR." IF "ENDCASE" APPEARED PRIOR
    ;   TO THE JUMP, THE ASSEMBLER WOULD
    ;   AUTOMATICALLY DETERMINE IF IT WERE REACHABLE
    ;   WITH A SHORT JUMP.
                JMP         SHORT ENDCASE

ENDCASE:        ; ALL CASES CONVERGE HERE.

    ; POP CALLER'S REGISTERS HERE.
    ; RESTORE BP & SP, DISCARD PARAMETERS
    ;   AND RETURN TO CALLER.
                MOV         SP, BP
                POP         BP
                RET         6

DO_CASE         ENDP
CODE            ENDS
                END         ; OF ASSEMBLY
```

Figure 2-81. JMP and CALL Examples (Cont'd.)

Records

Figure 2-82 shows how the ASM-86 RECORD facility may be used to manipulate bit data. The example shows how to:

- right-justify a bit field,
- test for a value,
- assign a constant known at assembly time,
- assign a variable,
- set or clear a bit field.

```
DATA            SEGMENT
; DEFINE A WORD ARRAY
XREF            DW 3000 DUP (?)
; EACH ELEMENT OF XREF CONSISTS OF 3 FIELDS:
;       A 2-BIT TYPE CODE,
;       A 1-BIT FLAG,
;       A 13-BIT NUMBER.
; DEFINE A RECORD TO LAY OUT THIS ORGANIZATION.
LINE_REC        RECORD  LINE_TYPE: 2,
&                       VISIBLE: 1,
&                       LINE_NUM: 13
DATA            ENDS

CODE            SEGMENT
                ASSUME CS: CODE,   DS: DATA
; ASSUME SEGMENT REGISTERS ARE SET UP PROPERLY
;     AND THAT SI INDEXES AN ELEMENT OF XREF.

; A RECORD FIELD-NAME USED BY ITSELF RETURNS
;   THE SHIFT COUNT REQUIRED TO RIGHT-JUSTIFY
;   THE FIELD. ISOLATE "LINE_TYPE" IN THIS
;   MANNER.
                MOV     AL, XREF [SI]
                MOV     CL, LINE_TYPE
                SHR     AX, CL

; THE "MASK" OPERATOR APPLIED TO A RECORD
;    FIELD-NAME RETURNS THE BIT MASK
;    REQUIRED TO ISOLATE THE FIELD WITHIN
;    THE RECORD. CLEAR ALL BITS EXCEPT
;    "LINE_NUM."
                MOV     DX, XREF[SI]
                AND     DX, MASK LINE_NUM

; DETERMINE THE VALUE OF THE "VISIBLE" FIELD
                TEST    XREF[SI], MASK VISIBLE
                JZ      NOT_VISIBLE
; NO JUMP IF VISIBLE = 1
NOT_VISIBLE:    ; JUMP HERE IF VISIBLE = 0

; ASSIGN A CONSTANT KNOWN AT ASSEMBLY-TIME
;     TO A FIELD, BY FIRST CLEARING THE BITS
;     AND THEN OR'ING IN THE VALUE. IN
;     THIS CASE "LINE_TYPE" IS SET TO 2 (10B).
                AND     XREF[SI], NOT MASK LINE_TYPE
                OR      XREF[SI],2 SHL LINE_TYPE
; THE ASSEMBLER DOES THE MASKING AND SHIFTING.
; THE RESULT IS THE SAME AS:
                AND     XREF[SI], 3FFFH
                OR      XREF[SI], 8000H
;   BUT IS MORE READABLE AND LESS SUBJECT
;   TO CLERICAL ERROR.
```

Figure 2-82. RECORD Example

```
; ASSIGN A VARIABLE (THE CONTENT OF AX)
;     TO LINE_TYPE.
                MOV     CL, LINE_TYPE   ; SHIFT COUNT
                SHL     AX, CL   ; SHIFT TO "LINE UP" BITS
                AND     XREF[SI], NOT MASK LINE_TYPE   ; CLEAR BITS
                OR      XREF[SI], AX   ; OR IN NEW VALUE

; NO SHIFT IS REQUIRED TO ASSIGN TO THE
;     RIGHT-MOST FIELD. ASSUMING AX CONTAINS
;     A VALID NUMBER (HIGH 3 BITS ARE 0),
;     ASSIGN AX TO "LINE_NUM."
                AND     XREF[SI], NOT MASK LINE_NUM
                OR      XREF[SI], AX

; A FIELD MAY BE SET OR CLEARED WITH
;   ONE INSTRUCTION. CLEAR THE "VISIBLE"
;   FLAG AND THEN SET IT.
                AND     XREF[SI], NOT MASK VISIBLE
                OR      XREF[SI], MASK VISIBLE

CODE            ENDS
                END     ; OF ASSEMBLY
```

Figure 2-82. RECORD Example (Cont'd.)

The following considerations apply to position-independent code sequences:

- A label that is referenced by a direct FAR (intersegment) transfer is not moveable.
- A label that is referenced by an indirect transfer (either NEAR or FAR) is moveable so long as the register or memory pointer to the label contains the label's current address.
- A label that is referenced by a SHORT (e.g., conditional jump) or a direct NEAR (intrasegment) transfer is moveable so long as the referencing instruction is moved with the label as a unit. These transfers are self-relative; that is they require only that the label maintain the same distance from the referencing instruction, and actual addresses are immaterial.
- Data is segment-independent, but not offset-independent. That is, a data item may be moved to a different segment, but it must maintain the same offset from the beginning of the segment. Placing constants in a unit of code also effectively makes the code offset-dependent, and therefore is not recommended.
- A procedure should not be moved while it is active or while any procedure it has called is active.
- A section of code that has been interrupted should not be moved.

The segment that is receiving a section of code must have "room" for the code. If the MOVS (or MOVSB or MOVSW) instruction attempts to auto-increment DI past 64k, it wraps around to 0 and causes the beginning of the segment to be overwritten. If a segment override is needed for the source operand, code similar to the following can be used to properly resume the instruction if it is interrupted:

```
RESUME:  REP   MOVS     DESTINATION, ES:SOURCE
  ;IF CX NOT = 0 THEN INTERRUPT HAS OCCURRED
           AND  CX,CX     ; CX=0?
           JNZ  RESUME   ;NO, FINISH EXECUTION
  ;CONTROL COMES HERE WHEN STRING HAS BEEN MOVED.
```

If the MOVS is interrupted, the CPU "remembers" the segment override, but "forgets" the presence of the REP prefix when execution resumes. Testing CX indicates whether the instruction is completed or not. Jumping back to the instruction resumes it where it left off. Note that a segment override cannot be specified with MOVSB or MOVSW.

Dynamic Code Relocation

Figure 2-83 illustrates one approach to moving programs in memory at execution time. A "supervisor" program (which is not moved) keeps a pointer variable that contains the current location (offset and segment base) of a position-independent procedure. The supervisor always calls the procedure through this pointer. The supervisor also has access to the procedure's length in bytes. The procedure is moved with the MOVSB instruction. After the procedure is moved, its pointer is updated with the new location. The ASM-86 WORD PTR operator is written to inform the assembler that one word of the doubleword pointer is being updated at a time.

```
MAIN_DATA       SEGMENT
; SET UP POINTERS TO POSITION-INDEPENDENT PROCEDURE
;    AND FREE SPACE.
PIP_PTR         DD          EXAMPLE
FREE_PTR        DD          TARGET_SEG
; SET UP SIZE OF PROCEDURE IN BYTES
PIP_SIZE        DW          EXAMPLE_LEN
MAIN_DATA       ENDS

STACK           SEGMENT
                DW          20 DUP (?)          ; 20 WORDS FOR STACK

STACK_TOP       LABEL       WORD                ; TOS BEGINS HERE
STACK           ENDS

SOURCE_SEG      SEGMENT
; THE POSITION-INDEPENDENT PROCEDURE IS INITIALLY IN THIS SEGMENT.
; OTHER CODE MAY PRECEDE IT, I.E., ITS OFFSET NEED NOT BE ZERO.
ASSUME          CS:SOURCE_SEG
EXAMPLE         PROC        FAR
   ; THIS PROCEDURE READS AN 8-BIT PORT UNTIL
   ; BIT 3 OF THE VALUE READ IS FOUND SET. IT
   ; THEN READS ANOTHER PORT. IF THE VALUE READ
   ; IS GREATER THAN 10H IT WRITES THE VALUE TO
   ; A THIRD PORT AND RETURNS; OTHERWISE IT STARTS
   ; OVER.
STATUS_PORT     EQU         0D0H
PORT_READY      EQU         008H
INPUT_PORT      EQU         0D2H
THRESHOLD       EQU         010H
OUTPUT_PORT     EQU         0D4H
CHECK_AGAIN:    IN          AL,STATUS_PORT      ; GET STATUS
                TEST        AL,PORT_READY       ; DATA READY?
                JNE         CHECK_AGAIN         ; NO, TRY AGAIN
                IN          AL,INPUT_PORT       ; YES, GET DATA
                CMP         AL,THRESHOLD        ; > 10H?
                JLE         CHECK_AGAIN         ; NO, TRY AGAIN
                OUT         OUTPUT_PORT,AL      ; YES, WRITE IT
```

Figure 2-83. Dynamic Code Relocation Example

```
                RET         ; RETURN TO CALLER
; GET PROCEDURE LENGTH
EXAMPLE_LEN     EQU         (OFFSET THIS BYTE)—(OFFSET CHECK_AGAIN)
                ENDP        EXAMPLE  ENDP
SOURCE_SEG      ENDS

TARGET_SEG      SEGMENT
; THE POSITION-INDEPENDENT PROCEDURE
;   IS MOVED TO THIS SEGMENT, WHICH IS
;   INITIALLY "EMPTY."
; IN TYPICAL SYSTEMS, A "FREE SPACE MANAGER" WOULD
;   MAINTAIN A POOL OF AVAILABLE MEMORY SPACE
; FOR ILLUSTRATION PURPOSES, ALLOCATE ENOUGH
;   SPACE TO HOLD IT
                DB          EXAMPLE_LEN DUP (?)

TARGET_SEG      ENDS

MAIN_CODE       SEGMENT
; THIS ROUTINE CALLS THE EXAMPLE PROCEDURE
; AT ITS INITIAL LOCATION, MOVES IT, AND
; CALLS IT AGAIN AT THE NEW LOCATION.

ASSUME          CS:MAIN_CODE,SS:STACK,
&               DS:MAIN_DATA,ES:NOTHING

; INITIALIZE SEGMENT REGISTERS & STACK POINTER.
START:          MOV         AX,MAIN_DATA
                MOV         DS,AX
                MOV         AX,STACK
                MOV         SS,AX
                MOV         SP,OFFSET STACK_TOP

; CALL EXAMPLE AT INITIAL LOCATION.
                CALL        PIP_PTR

; SET UP CX WITH COUNT OF BYTES TO MOV
                MOV         CX,PIP_SIZE
; SAVE DS, SET UP DS/SI AND ES/DI TO
;   POINT TO THE SOURCE AND DESTINATION
;   ADDRESSES.
                PUSH        DS
                LES         DI,FREE_PTR
                LDS         SI,PIP_PTR
; MOVE THE PROCEDURE.
                CLD                             ; AUTO INCREMENT
                REP MOVSB

; RESTORE OLD ADDRESSABILITY.
                MOV         AX,DS               ; HOLD TEMPORARILY
                POP         DS
; UPDATE POINTER TO POSITION-INDEPENDENT PROCEDURE
                MOV         WORD PTR PIP_PTR+2,ES
                SUB         DI,PIP_SIZE         ; PRODUCES OFFSET
                MOV         WORD PTR PIP_PTR,DI
```

Figure 2-83. Dynamic Code Relocation Example (Cont'd.)

```
; UPDATE POINTER TO FREE SPACE
                MOV     WORD PTR FREE_PTR+2,AX
                SUB     SI,PIP_SIZE          ; PRODUCES OFFSET
                MOV     WORD PTR FREE_PTR,SI

; CALL POSITION-INDEPENDENT PROCEDURE AT
;   NEW LOCATION AND STOP
                CALL    PIP_PTR
MAIN_CODE       ENDS
                END     START
```

Figure 2-83. Dynamic Code Relocation Example (Cont'd.)

Memory-Mapped I/O

Figure 2-84 shows how memory-mapped I/O can be used to address a group of communication lines as an "array." In the example, indexed addressing is used to poll the array of status ports, one port at a time. Any of the other 8086/8088 memory addressing modes may be used in conjunction with memory-mapped I/O devices as well.

In figure 2-85 a MOVS instruction is used to perform a high-speed transfer to a memory-mapped line printer. Using this technique requires the hardware to be set up as follows. Since the MOVS instruction transfers characters to successive memory addresses, the decoding logic must select the line printer if any of these locations is written. One way of accomplishing this is to have the chip select logic decode only the upper 12 lines of the address bus (A19-A8), ignoring the contents of the lower eight lines (A7-A0). When data is written to any address in this 256-byte block, the upper 12 lines will not change, so the printer will be selected.

If an 8086 is being used with an 8-bit printer, the 8086's 16-bit data bus must be mapped into 8-bits by external hardware. Using an 8088 provides a more direct interface.

```
COM_LINES       SEGMENT  AT 800H
; THE FOLLOWING IS A MEMORY MAPPED "ARRAY"
;       OF EIGHT 8-BIT COMMUNICATIONS CONTROLLERS
;       (E.G., 8251 USARTS). PORTS HAVE ALL-ODD
;       OR ALL-EVEN ADDRESSES (EVERY OTHER BYTE
;       IS SKIPPED) FOR 8086-COMPATIBILITY.

COM_DATA        DB      ?
                DB      ?                   ; SKIP THIS ADDRESS
COM_STATUS      DB      ?
                DB      ?                   ; SKIP THIS ADDRESS
                DB      28      DUP (?)     ; REST OF "ARRAY"
COM_LINES       ENDS

CODE            SEGMENT
; ASSUME STACK IS SET UP, AS ARE SEGMENT
;       REGISTERS (DS POINTING TO COM_LINES).
;       FOLLOWING CODE POLLS THE LINES.

CHAR_RDY        EQU     00000010B   ; CHARACTER PRESENT
START_POLL:     MOV     CX, 8       ; POLL 8 LINES ZERO
                SUB     SI, SI      ; ARRAY INDEX
```

Figure 2-84. Memory Mapped I/O "Array"

```
POLL_NEXT:    TEST    COM_STATUS [SI], CHAR_RDY
              JE      READ_CHAR ; READ IF PRESENT
              ADD     SI, 4           ; ELSE BUMP TO NEXT LINE
              LOOP    POLL_NEXT ; CONTINUE POLLING UNTIL
                                ;     ALL 8 HAVE BEEN CHECKED
              JMP     START_POLL; START OVER

READ_CHAR:    MOV     AL,COM_DATA [SI]   ;GET THE DATA
; ETC.
CODE          ENDS
              END
```

Figure 2-84. Memory Mapped I/O "Array" (Cont'd.)

```
PRINTER         SEGMENT
; THIS SEGMENT CONTAINS A "STRING" THAT
;    IS ACTUALLY A MEMORY-MAPPED LINE PRINTER.
;    THE SEGMENT (PRINTER) MUST BE ASSIGNED (LOCATED)
;    TO A BLOCK OF THE ADDRESS SPACE SUCH
;    THAT WRITING TO ANY ADDRESS IN THE
;    BLOCK SELECTS THE PRINTER.

PRINT_SELECT    DB 133      DUP (?)          ; "STRING" REPRESENTING PRINTER
                DB 123      DUP (?)          ; REST OF 256-BYTE BLOCK
PRINTER         ENDS

DATA            SEGMENT
PRINT_BUF       DB 133      DUP (?)          ; LINE TO BE PRINTED
PRINT_COUNT     DB 1        ?                ; LINE LENGTH
; OTHER PROGRAM DATA
DATA            ENDS

CODE            SEGMENT
; ASSUME STACK AND SEGMENT REGISTERS HAVE
;      BEEN SET UP (DS POINTS TO DATA SEGMENT).
;      FOLLOWING CODE TRANSFERS A LINE TO
;      THE PRINTER.

                ASSUME  ES: PRINTER
                MOV     AX, PRINTER      ; PREVENT SEGMENT OVERRIDE
                MOV     ES, AX
                SUB     DI, DI           ; CLEAR SOURCE AND
                SUB     SI, SI           ;    DESTINATION POINTERS
                MOV     CX, PRINT_COUNT
                CLD     ; AUTO-INCREMENT
    REP         MOVS    PRINT_SELECT, PRINT_BUF
                ; ETC.
CODE            ENDS
                END
```

Figure 2-85. Memory Mapped Block Transfer Example

Breakpoints

Figure 2-86 illustrates how a program may set a breakpoint. In the example, the breakpoint routine puts the processor into single-step mode, but the same general approach could be used for other purposes as well. A program passes the address where the break is to occur to a procedure that saves the byte located at that address and replaces it with an INT 3 (breakpoint) instruction. When the CPU encounters the breakpoint instruction, it calls the type 3 interrupt procedure. In the example, this procedure places the processor into single-step mode starting with the instruction where the breakpoint was placed.

```
INT_PTR_TAB   SEGMENT
; INTERRUPT POINTER TABLE-LOCATE AT 0H
TYPE_0        DD        ?                ; NOT DEFINED IN EXAMPLE
TYPE_1        DD        SINGLE_STEP
TYPE_2        DD        ?                ; NOT DEFINED IN EXAMPLE
TYPE_3        DD        BREAKPOINT
INT_PTR_TAB   ENDS

SAVE_SEG      SEGMENT
SAVE_INSTR    DB 1      DUP (?)          ; INSTRUCTION REPLACED
                                         ; BY BREAKPOINT
SAVE_SEG      ENDS

MAIN_CODE     SEGMENT
; ASSUME STACK AND SEGMENT REGISTERS ARE SET UP.

; ENABLE SINGLE-STEPPING WITH INSTRUCTION AT
;    LABEL "NEXT" BY PASSING SEGMENT AND
;    OFFSET OF "NEXT" TO "SET_BREAK" PROCEDURE
              PUSH      CS
              LEA       AX, CS: NEXT
              PUSH      AX
              CALL      FAR SET_BREAK
; ETC.

NEXT:         IN        AL, 0FFFH        ; BREAKPOINT SET HERE
              ; ETC.

MAIN_CODE     ENDS

BREAK         SEGMENT
SET_BREAK     PROC      FAR
; THIS PROCEDURE SAVES AN INSTRUCTION BYTE (WHOSE
;    ADDRESS IS PASSED BY THE CALLER) AND WRITES
;    AN INT 3 (BREAKPOINT) MACHINE INSTRUCTION
;    AT THE TARGET ADDRESS.

TARGET        EQU       DWORD PTR [BP + 6]
```

Figure 2-86. Breakpoint Example

```
; SET UP BP FOR PARM ADDRESSING & SAVE REGISTERS
                PUSH    BP
                MOV     BP, SP
                PUSH    DS
                PUSH    ES
                PUSH    AX
                PUSH    BX
; POINT DS/BX TO THE TARGET INSTRUCTION
                LDS     BX, TARGET
; POINT ES TO THE SAVE AREA
                MOV     AX, SAVE_SEG
                MOV     ES, AX
; SWAP THE TARGET INSTRUCTION FOR INT 3 (0CCH)
                MOV     AL, 0CCH
                XCHG    AL, DS: [BX]
; SAVE THE TARGET INSTRUCTION
                MOV     ES: SAVE_INSTR, AL
; RESTORE AND RETURN
                POP     BX
                POP     AX
                POP     ES
                POP     DS
                POP     BP
                RET     4
SET_BREAK       ENDP

BREAKPOINT      PROC    FAR
; THE CPU WILL ACTIVATE THIS PROCEDURE WHEN IT
;   EXECUTES THE INT 3 INSTRUCTION SET BY THE
;   SET_BREAK PROCEDURE. THIS PROCEDURE
;   RESTORES THE SAVED INSTRUCTION BYTE TO ITS
;   ORIGINAL LOCATION AND BACKS UP THE
;   INSTRUCTION POINTER IMAGE ON THE STACK
;   SO THAT EXECUTION WILL RESUME WITH
;   THE RESTORED INSTRUCTION. IT THEN SETS
;   TF (THE TRAP FLAG) IN THE FLAG-IMAGE
;   ON THE STACK. THIS PUTS THE PROCESSOR
;   IN SINGLE-STEP MODE WHEN EXECUTION
;   RESUMES.
    FLAG_IMAGE  EQU     WORD PTR [BP+6]
    IP_IMAGE    EQU     WORD PTR [BP+2]
NEXT_INSTR      EQU     DWORD PTR [BP+2]
; SET UP BP TO ADDRESS STACK AND SAVE REGISTERS
                PUSH    BP
                MOV     BP, SP
                PUSH    DS
                PUSH    ES
                PUSH    AX
                PUSH    BX
; POINT ES AT THE SAVE AREA
                MOV     AX, SAVE_SEG
                MOV     ES, AX
; GET THE SAVED BYTE
                MOV     AL, ES: SAVE_INSTR
```

Figure 2-86. Breakpoint Example (Cont'd.)

```
; GET THE ADDRESS OF THE TARGET + 1
;    (INSTRUCTION FOLLOWING THE BREAKPOINT)
                LDS         BX, NEXT_INSTR
; BACK UP IP-IMAGE (IN BX) AND REPLACE ON STACK
                DEC         BX
                MOV         IP_IMAGE, BX

; RESTORE THE SAVED INSTRUCTION
                MOV         DS: [BX], AL
; SET TF ON STACK
                AND         FLAG_IMAGE, 0100H
; RESTORE EVERYTHING AND EXIT
                POP         BX
                POP         AX
                POP         ES
                POP         DS
                POP         BP
                IRET
BREAKPOINT      ENDP

SINGLE  STEP    PROC        FAR
; ONCE SINGLE-STEP MODE HAS BEEN ENTERED,
;    THE CPU "TRAPS" TO THIS PROCEDURE
;    AFTER EVERY INSTRUCTION THAT IS NOT IN
;    AN INTERRUPT PROCEDURE. IN THE CASE
;    OF THIS EXAMPLE, THIS PROCEDURE WILL
;    BE EXECUTED IMMEDIATELY FOLLOWING THE
;    "IN AL, 0FFFH" INSTRUCTION (WHERE THE
;    BREAKPOINT WAS SET) AND AFTER EVERY
;    SUBSEQUENT INSTRUCTION. THE PROCEDURE
;    COULD "TURN ITSELF OFF" BY CLEARING
;    TF ON THE STACK.
; SINGLE-STEP CODE GOES HERE.
; SINGLE_STEP   ENDP

BREAK           ENDS

                END         ;
```

Figure 2-86. Breakpoint Example (Cont'd.)

Interrupt Procedures

Figure 2-87 is a block diagram of a hypothetical system that is used to illustrate three different examples of interrupt handling: an external (maskable) interrupt, an external non-maskable interrupt and a software interrupt.

In this hypothetical system, an 8253 Programmable Interval Timer is used to generate a time base. One of the three timers on the 8253 is programmed to repeatedly generate interrupt requests at 50 millisecond intervals. The output from this timer is tied to one of the eight interrupt request lines of an 8259A Programmable Interrupt Controller. The 8259A, in turn, is connected to the INTR line of an 8086 or 8088.

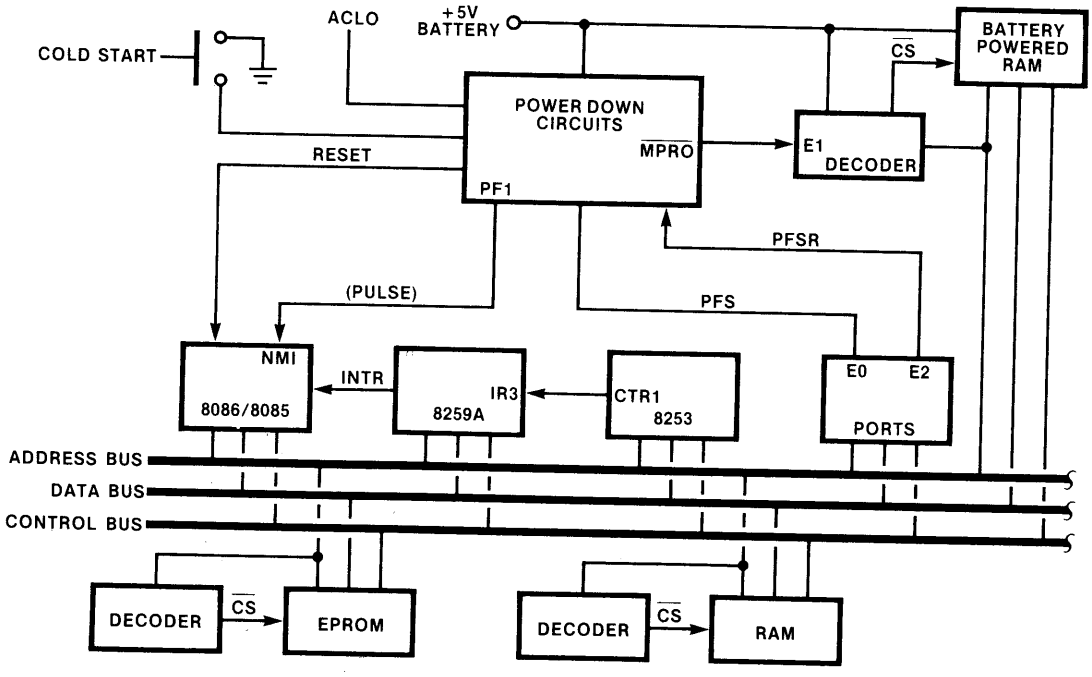

Figure 2-87. Interrupt Example Block Diagram

A power-down circuit is used in the system to illustrate one application of the 8086/8088 NMI (non-maskable interrupt) line. If the ac line voltage drops below a certain threshold, the power supply activates ACLO. The power-down circuit then sends a power-fail interrupt (PFI) pulse to the CPU's NMI input. After 5 milliseconds, the power-down circuit activates MPRO (memory protect) to disable reading from and writing to the system's battery-powered RAM. This protects the RAM from fluctuations that may occur when power is actually lost 7.5 milliseconds after the power failure is detected. The system software must save all vital information in the battery-powered RAM segment within 5 milliseconds of the activation of NMI.

When power returns, the power-down circuit activates the system RESET line. Pressing the "cold start" switch also produces a system RESET. The PFS (power fail status) line, which is connected to the low-order bit of port E0, identifies the source of the RESET. If the bit is set, the software executes a "warm start" to restore the information saved by the power-fail routine. If the PFS bit is cleared, the software executes a "cold start" from the beginning of the program. In either case, the software writes a "one" to the low-order bit of port E2. This line is connected to the power-down circuit's PFSR (power fail status reset) signal and is used to enable the battery-powered RAM segment.

A software interrupt is used to update a simple real-time clock. This procedure is written in PL/M-86, while the rest of the system is written in ASM-86 to demonstrate the interrupt handling capability of both languages. The system's main program simply initializes the system following receipt of a RESET and then waits for an interrupt. An example of this interrupt procedure is given in figure 2-88.

```
INT_POINTERS                    SEGMENT
; INTERRUPT POINTER TABLE, LOCATE AT 0H, ROM-BASED
TYPE_0          DD              ?               ; DIVIDE-ERROR NOT SUPPLIED IN EXAMPLE.
TYPE_1          DD              ?               ; SINGLE-STEP NOT SUPPLIED IN EXAMPLE.
TYPE_2          DD              POWER_FAIL      ; NON-MASKABLE INTERRUPT
TYPE_3          DD              ?               ; BREAKPOINT NOT SUPPLIED IN EXAMPLE.
TYPE_4          DD              ?               ; OVERFLOW NOT SUPPLIED IN EXAMPLE.
; SKIP RESERVED PART OF EXAMPLE
                ORG             32*4
TYPE_32         DD              ?               ; 8259A IR0 - AVAILABLE
TYPE_33         DD              ?               ; 8259A IR1 - AVAILABLE
TYPE_34         DD              ?               ; 8259A IR2 - AVAILABLE
TYPE_35         DD              TIMER_PULSE     ; 8259A IR3
TYPE_36         DD              ?               ; 8259A IR4 - AVAILABLE
TYPE_37         DD              ?               ; 8259A IR5 - AVAILABLE
TYPE_38         DD              ?               ; 8259A IR6 - AVAILABLE
TYPE_39         DD              ?               ; 8259A IR7 - AVAILABLE
;
; POINTER FOR TYPE 40 SUPPLIED BY PL/M-86 COMPILER
;
INT_POINTERS                    ENDS

BATTERY                         SEGMENT
; THIS RAM SEGMENT IS BATTERY-POWERED. IT CONTAINS VITAL DATA
;    THAT MUST BE MAINTAINED DURING POWER OUTAGES.
STACK_PTR       DW              ?               ; SP SAVE AREA
STACK_SEG       DW              ?               ; SS SAVE AREA
; SPACE FOR OTHER VARIABLES COULD BE DEFINED HERE.
BATTERY                         ENDS

DATA                            SEGMENT
; RAM SEGMENT THAT IS NOT BACKED UP BY BATTERY
N_PULSES        DB              1 DUP (0)       ; # TIMER PULSES

; ETC.
DATA                            ENDS

STACK                           SEGMENT
; LOCATED IN BATTERY-POWERED RAM
                DW              100 DUP (?)     ; THIS IS AN ARBITRARY STACKSIZE

STACK_TOP       LABEL           WORD            ; LABEL THE INITIAL TOS
STACK                           ENDS

INTERRUPT_HANDLERS      SEGMENT
; INTERRUPT PROCEDURES EXCEPT TYPE 40 (PL/M-86)

                ASSUME:         CS:INTERRUPT_HANDLERS,DS:DATA,SS:STACK,ES:BATTERY

POWER_FAIL                      PROC            ; TYPE 2 INTERRUPT
; POWER FAIL DETECT CIRCUIT ACTIVATES NMI LINE ON CPU IF POWER IS
;    ABOUT TO BE LOST. THIS PROCEDURE SAVES THE PROCESSOR STATE IN
;    RAM (ASSUMED TO BE POWERED BY AN AUXILIARY SOURCE) SO THAT IT
;    CAN BE RESTORED BY A WARM START ROUTINE IF POWER RETURNS
```

Figure 2-88. Interrupt Procedures Example

```
; IP, CS, AND FLAGS ARE ALREADY ON THE STACK.
;   SAVE THE OTHER REGISTERS.
                    PUSH        AX
                    PUSH        BX
                    PUSH        CX
                    PUSH        DX
                    PUSH        SI
                    PUSH        DI
                    PUSH        BP
                    PUSH        DS
                    PUSH        ES

; CRITICAL MEMORY VARIABLES COULD ALSO BE SAVED ON THE STACK AT THIS
;    POINT. ALTERNATIVELY, THEY COULD BE DEFINED IN THE "BATTERY"
;    SEGMENT, WHERE THEY WILL AUTOMATICALLY BE PROTECTED IF MAIN POWER
;    IS LOST.

; SAVE SP AND SS IN FIXED LOCATIONS THAT ARE KNOWN BY WARM START ROUTINE.
                    MOV         AX,BATTERY
                    MOV         ES,AX
                    MOV         ES:STACK_PTR,SP
                    MOV         ES:STACK_SEG,SS
; STOP GRACEFULLY
                    HLT
POWER_FAIL                      ENDP

TIMER_PULSE                     PROC                ; TYPE 35 INTERRUPT
; THIS PROCEDURE HANDLES THE 50MS INTERRUPTS GENERATED BY THE 8253.
;    IT COUNTS THE INTERRUPTS AND ACTIVATES THE TYPE 40 INTERRUPT
;    PROCEDURE ONCE PER SECOND.
;
; DS IS ASSUMED TO BE POINTING TO THE DATA SEGMENT
;
; THE 8253 IS RUNNING FREE, AND AUTOMATICALLY LOWERS ITS INTERRUPT
;    REQUEST. IF A DEVICE REQUIRED ACKNOWLEDGEMENT, THE CODE MIGHT GO HERE.
;
; NOW PERFORM PROCESSING THAT MUST NOT BE INTERRUPTED (EXCEPT FOR NMI).
                    INC         N_PULSES
; ENABLE HIGHER-PRIORITY INTERRUPTS AND DO LESS CRITICAL PROCESSING
                    STI
                    CMP         N_PULSES,200        ; 1 SECOND PASSED?
                    JBE         DONE                ; NO, GO ON.
                    MOV         N_PULSES,0          ; YES, RESET COUNT.
                    INT         40                  ; UPDATE CLOCK
; SEND NON-SPECIFIC END-OF-INTERRUPT COMMAND TO 8259A, ENABLING EQUAL
;    OR LOWER PRIORITY INTERRUPTS.
DONE:               MOV         AL,020H             ; EOI COMMAND
                    OUT         0C0H,AL             ; 8259A PORT
                    IRET
TIMER_PULSE                     ENDP

INTERRUPT_HANDLERS              ENDS


CODE                SEGMENT
; THIS SEGMENT WOULD NORMALLY RESIDE IN ROM.

                    ASSUME      CS:CODE,DS:DATA,SS:STACK,ES:NOTHING
```

Figure 2-88. Interrupt Procedures Example (Cont'd.)

```
INIT            PROC        NEAR
; THIS PROCEDURE IS CALLED FOR BOTH WARM AND COLD STARTS TO INITIALIZE
;    THE 8253 AND THE 8259A. THIS ROUTINE DOES NOT USE STACK, DATA, OR
;    EXTRA SEGMENTS, AS THEY ARE NOT SET PREDICTABLY DURING A WARM START.
;    INTERRUPTS ARE DISABLED BY VIRTUE OF THE SYSTEM RESET.

; INITIALIZE 8253 COUNTER 1 - OTHER COUNTERS NOT USED.
; CLK INPUT TO COUNTER IS ASSUMED TO BE 1.23 MHZ.

LO50MS          EQU         000H            ; COUNT VALUE IS
HI50MS          EQU         0F0H            ;    61440 DECIMAL.
CONTROL         EQU         0D6H            ; CONTROL PORT ADDRESS
COUNT_1         EQU         0D2H            ; COUNTER 1 ADDRESS
MODE2           EQU         01110100B       ; MODE 2, BINARY

                MOV         DX,CONTROL      ; LOAD CONTROL BYTE
                MOV         AL,MODE2
                OUT         DX,AL
                MOV         DX,COUNT_1      ; LOAD 50MS DOWNCOUNT
                MOV         AL,LO50MS
                OUT         DX,AL
                MOV         AL,HI50MS
                OUT         DX,AL
                ; COUNTER NOW RUNNING, INTERRUPTS STILL DISABLED.

; INITIALIZE 8259A TO: SINGLE INTERRUPT CONTROLLER, EDGE-TRIGGERED,
;    INTERRUPT TYPES 32-40 (DECIMAL) TO BE SENT TO CPU FOR INTERRUPT
;    REQUESTS 0-7 RESPECTIVELY, 8086 MODE, NON-AUTOMATIC END-OF-INTERRUPT.
;    MASK OFF UNUSED INTERRUPT REQUEST LINES.

ICW1            EQU         00010011B       ; EDGE-TRIGGERED, SINGLE 8259A, ICW4 REQUIRED.
ICW2            EQU         00100000B       ; TYPE 20H, 32 - 40D
ICW4            EQU         00000001B       ; 8086 MODE, NORMAL EOI
OCW1            EQU         11110111B       ; MASK ALL BUT IR3
PORT_A          EQU         0C0H            ; ICW1 WRITTEN HERE
PORT_B          EQU         0C2H            ; OTHER ICW'S WRITTEN HERE

                MOV         DX,PORT_A       ; WRITE 1ST ICW
                MOV         AL,ICW1
                OUT         DX,AL
                MOV         DX,PORT_B       ; WRITE 2ND ICW
                MOV         AL,ICW2
                OUT         DX,AL
                MOV         AL,ICW4         ; WRITE 4TH ICW
                OUT         DX,AL
                MOV         AL,0CW1         ; MASK UNUSED IR'S
                OUT         DX,AL
; INITIALIZATION COMPLETE, INTERRUPTS STILL DISABLED
                RET
INIT            ENDP


USER_PGM:
; "REAL" CODE WOULD GO HERE. THE EXAMPLE EXECUTES AN ENDLESS LOOP
;    UNTIL AN INTERRUPT OCCURS.
                JMP         USER_PGM

; EXECUTION STARTS HERE WHEN CPU IS RESET.
POWER_FAIL_STATUS       EQU     0E0H        ; PORT ADDRESS
ENABLE_RAM              EQU     0E2H        ; PORT ADDRESS
```

Figure 2-88. Interrupt Procedures Example (Cont'd.)

```
; ENABLE BATTERY-POWERED RAM SEGMENT
START:          MOV     AL,001H
                OUT     ENABLE_RAM,AL

; DETERMINE WARM OR COLD START
                IN      AL,POWER_FAIL_STATUS
                RCR     AL,1                    ; ISOLATE LOW BIT
                JC      WARM_START

COLD_START:
; INITIALIZE SEGMENT REGISTERS AND STACK POINTER.
                ASSUME  CS:CODE,DS:DATA,SS:STACK,ES:NOTHING
                ; RESET TAKES CARE OF CS AND IP.
                MOV     AX,DATA
                MOV     DS,AX
                MOV     AX,STACK
                MOV     SS,AX
                MOV     SP,OFFSET STACK_TOP

; INITIALIZE 8253 AND 8259A.
                CALL    INIT

; ENABLE INTERRUPTS
                STI

; START MAIN PROCESSING
                JMP     USER_PGM


WARM_START:
; INITIALIZE 8253 AND 8259A.
                CALL    INIT

; RESTORE SYSTEM TO STATE AT THE TIME POWER FAILED
                ; MAKE BATTERY SEGMENT ADDRESSABLE
                    MOV     AX,BATTERY
                    MOV     DX,AX
                ; VARIABLES SAVED IN THE "BATTERY" SEGMENT WOULD BE MOVED
                ;   BACK TO UNPROTECTED RAM NOW. SEGMENT REGISTERS AND
                ;   "ASSUME" DIRECTIVES WOULD HAVE TO BE WRITTEN TO GAIN
                ;   ADDRESSABILITY.

                ; RESTORE THE OLD STACK
                    MOV     SS,DS:STACK_SEG
                    MOV     SP,DS:STACK_PTR
                ; RESTORE THE OTHER REGISTERS
                    POP     ES
                    POP     DS
                    POP     BP
                    POP     DI
                    POP     SI
                    POP     DX
                    POP     CX
                    POP     BX
                    POP     AX
                ; RESUME THE ROUTINE THAT WAS EXECUTING WHEN NMI WAS ACTIVATED.
                ;   I.E., POP CS, IP, & FLAGS, EFFECTIVELY "RETURNING" FROM THE
                ;   NMI PROCEDURE.
                    IRET
CODE            ENDS

                ; TERMINATE ASSEMBLY AND MARK BEGINNING OF THE PROGRAM.
                                END     START
```

Figure 2-88. Interrupt Procedures Example (Cont'd.)

```
TYPE$40:  DO;
  DECLARE (HOUR, MIN, SEC) BYTE PUBLIC;
  UPDATE$TOD: PROCEDURE INTERRUPT 40;
    /*THE PROCESSOR ACTIVATES THIS PROCEDURE
     *TO HANDLE THE SOFTWARE INTERRUPT
     *GENERATED EVERY SECOND BY THE TYPE 35
     *EXTERNAL INTERRUPT PROCEDURE. THIS
     *PROCEDURE UPDATES A REAL-TIME CLOCK.
     *IT DOES NOT PRETEND TO BE ''REALISTIC''
     *AS THERE IS NO WAY TO SET THE CLOCK.*/

  SEC = SEC + 1;
  IF SEC = 60 THEN DO;
    SEC = 0;
    MIN = MIN + 1;
    IF MIN = 60 THEN DO;
      MIN = 0;
      HOUR = HOUR + 1;
      IF HOUR = 24 THEN DO;
        HOUR = 0;
        END;
      END;
    END;
  END UPDATE$TOD;
END;
```

Figure 2-88. Interrupt Procedures Example (Cont'd.)

String Operations

Figure 2-89 illustrates typical use of string instructions and repeat prefixes. The XLAT instruction also is demonstrated. The first example simply moves 80 words of a string using MOVS. Then two byte strings are compared to find the alphabetically lower string, as might be done in a sort. Next a string is scanned from right to left (the index register is auto-decremented) to find the last period (".") in the string. Finally a byte string of EBCDIC characters is translated to ASCII. The translation is stopped at the end of the string or when a carriage return character is encountered, whichever occurs first. This is an example of using the string primitives in combination with other instructions to build up more complex string processing operations.

```
ALPHA           SEGMENT
; THIS IS THE DATA THE STRING INSTRUCTIONS WILL USE
OUTPUT          DW 100      DUP (?)
INPUT           DW 100      DUP (?)
NAME_1          DB 'JONES, JONA'
NAME_2          DB 'JONES, JOHN'
SENTENCE        DB 80       DUP (?)
EBCDIC_CHARS    DB 80       DUP (?)
ASCII_CHARS     DB 80       DUP (?)
CONV_TAB        DB 64       DUP(0H)              ; EBCDIC TO ASCII
```

Figure 2-89. String Examples

```
        ; ASCII NULLS ARE SUBSTITUTED FOR "UNPRINTABLE" CHARS
                        DB 1        20H
                        DB 9        DUP (0H)
                        DB 7        '¢', '.', '<', '(', '+', 0H, '&'
                        DB 9        DUP (0H)
                        DB 8        '!', '$', '*', ')', ';', '¬', '-', '/'
                        DB 8        DUP (0H)
                        DB 6        '¦', ',', '%', '_', '>', '?'
                        DB 9        DUP (0H)
                        DB 17       '¨', ':', '#', '@', ''', '=', '"',
                                    0H, 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i'
                        DB 7        DUP (0H)
                        DB 9        'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r'
                        DB 7        DUP (0H)
                        DB 9        '≃', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'
                        DB 22       DUP (0H)
                        DB 10       ' ', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I'
                        DB 6        DUP (0H)
                        DB 10       ' ', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R'
                        DB 6        DUP (0H)
                        DB 10       ' ', 0H, 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z'
                        DB 6        DUP (0H)
                        DB 10       '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'
                        DB 6        DUP (0H)

ALPHA                   ENDS

STACK                   SEGMENT
                        DW 100      DUP (?)             ; THIS IS AN ARBITRARY STACK SIZE
                                                        ; FOR ILLUSTRATION ONLY.
STACK_BASE              LABEL       WORD                ; INITIAL TOS
STACK                   ENDS

CODE                    SEGMENT
BEGIN:                  ; SET UP SEGMENT REGISTERS. NOTICE THAT
                        ; ES & DS POINT TO THE SAME SEGMENT, MEANING
                        ; THAT THE CURRENT EXTRA & DATA
                        ; SEGMENTS FULLY OVERLAP. THIS ALLOWS
                        ; ANY STRING IN "ALPHA" TO BE USED
                        ; AS A SOURCE OR A DESTINATION.
                        ASSUME CS: CODE, SS: STACK,
&                               DS: ALPHA, ES: ALPHA
                        MOV         AX, STACK
                        MOV         SS, AX
                        MOV         SP, OFFSET STACK_BASE ; INITIAL TOS
                        MOV         AX, ALPHA
                        MOV         DS, AX
                        MOV         ES, AX

; MOVE THE FIRST 80 WORDS OF "INPUT" TO
;     THE LAST 80 WORDS OF "OUTPUT".
                        LEA         SI, INPUT           ; INITIALIZE
                        LEA         DI, OUTPUT + 20     ; INDEX REGISTERS
```

Figure 2-89. String Examples (Cont'd.)

```
                MOV     CX, 80                  ; REPETITION COUNT
                CLD                             ; AUTO-INCREMENT
        REP     MOVS    OUTPUT, INPUT

; FIND THE ALPHABETICALLY LOWER OF 2 NAMES.
                MOV     SI, OFFSET NAME_1  ; ALTERNATIVE
                MOV     DI, OFFSET NAME_2  ; TO LEA
                MOV     CX, SIZE NAME_2    ; CHAR. COUNT
                CLD                        ; AUTO-INCREMENT
        REPE    CMPS    NAME_2, NAME_1     ''WHILE EQUAL''
                JB      NAME_2_LOW
NAME_1_LOW:             ; NOT IN THIS EXAMPLE
NAME_2_LOW:             ; CONTROL COMES HERE IN THIS EXAMPLE.
                        ; DI POINTS TO BYTE ('H') THAT
                        ; COMPARED UNEQUAL.

; FIND THE LAST PERIOD ('.') IN A TEXT STRING.
                MOV     DI, OFFSET SENTENCE +
&                           LENGTH SENTENCE   ; START AT END
                MOV     CX, SIZE SENTENCE
                STD                        ; AUTO-DECREMENT
                MOV     AL, '.'            ; SEARCH ARGUMENT
        REPNE   SCAS    SENTENCE           ; ''WHILE NOT =''
                JCXZ    NO_PERIOD          ; IF CX=0, NO PERIOD FOUND
PERIOD:         ; IF CONTROL COMES HERE THEN
                ;   DI POINTS TO LAST PERIOD IN SENTENCE.
NO_PERIOD:      ; ETC.

; TRANSLATE A STRING OF EBCDIC CHARACTERS
;       TO ASCII, STOPPING IF A CARRIAGE RETURN
;       (0DH ASCII) IS ENCOUNTERED.
                MOV     BX, OFFSET CONV_TAB   ; POINT TO TRANSLATE TABLE
                MOV     SI, OFFSET EBCDIC_CHARS   ; INITIALIZE
                MOV     DI, OFFSET ASCII_CHARS    ;    INDEX REGISTERS
                MOV     CX, SIZE ASCII_CHARS      ;    AND COUNTER
                CLD                        ; AUTO-INCREMENT
NEXT:           LODS    EBCDIC_CHARS       ; NEXT EBCDIC CHAR IN AL
                XLAT    CONV_TAB           ; TRANSLATE TO ASCII
                STOS    ASCII_CHARS        ; STORE FROM AL
                TEST    AL, 0DH            ; IS IT CARRIAGE RETURN?
                LOOPNE  NEXT               ; NO, CONTINUE WHILE CX NOT 0
                JE      CR_FOUND           ; YES, JUMP
                ; CONTROL COMES HERE IF ALL CHARACTERS
                ;       HAVE BEEN TRANSLATED BUT NO
                ;       CARRIAGE RETURN IS PRESENT.
                ; ETC.

CR_FOUND:
                ; DI-1 POINTS TO THE CARRIAGE RETURN
                ;   IN ASCII_CHARS.

CODE            ENDS
                END
```

Figure 2-89. String Examples (Cont'd.)