Appendix D

The ARM Processor

D.1. (a) R8, R9, and R10, contain 1, 2, and 3, respectively.

(b) The values 20 and 30 are pushed onto a stack pointed to by R1 by the two Store instructions, and they occupy memory locations 1996 and 1992, respectively. They are then popped off the stack into R8 and R9. Finally, the Subtract instruction results in 10 ($30 - 20$) being stored in R10. The stack pointer R1 is returned to its original value of 2000.

D.2. (b) A memory operand cannot be referenced in a Subtract instruction.

(d) The immediate value 257 is 100000001 in binary, and is thus too long to fit in the 8-bit immediate field. Note that it cannot be generated by the rotation of any 8-bit value.

D.3. Use register R0 as a counter register and R1 as a work register. A first possible program uses a rotation operation, RRX (Rotate right extended), that was not explained in Appendix D. The operation RRX rotates the contents of a register, extended by the C flag, to the right by one bit position.

```
        MOV     R0, #32          Load R0 with count value 32.
LOOP    MOVS    R2, R2, LSL #1   Shift contents of R2 left
                                   one bit position, setting the
                                   high-order bit into the C flag.
        MOV     R1, R1, RRX      Rotate R1 right one bit
                                   position, including the C flag,
                                   as shown in Figure 2.25d.
        SUBS    R0, R0, #1       Check if finished.
        BNE     LOOP
        MOV     R2, R1           Load reversed pattern
                                   back into R2.
```

Next, we give a second possible program that does not use the RRX operation.

```
        MOV     R0, #32              Load R0 with the count value 32.
LOOP    MOV     R1, R1, LSR #1       Shift contents of R1 right by one
                                       bit position, setting the high-order bit to 0.
        CMP     R2, #0               Set condition codes
                                       to show status of R2 contents.
        ORRMI   R1, R1, #&80000000   Set high-order position of R1
                                       to 1 if high-order position of R2 is 1.
        MOV     R2, R2, LSL #1       Shift contents of R2 left one bit position,
                                       moving next bit into the sign-bit position.
        SUBS    R0, R0, #1           Loop back if all 32 bits
        BNE     LOOP                   have not been processed.
        MOV     R2, R1               Write reversed bits into R2.
```

D.4. Program trace:

TIME	R0	R1	R2
after 1st execution of BGT	3	4	NUM1 + 4
after 2nd execution of BGT	−14	3	NUM1 + 8
after 3rd execution of BGT	13	2	NUM1 + 12

D.5. Assume bytes are unsigned 8-bit values.

```
        LDR     R0, N         R0 is list counter.
        ADR     R1, X         R1 points to X list.
        ADR     R2, Y         R2 points to Y list.
        ADR     R3, LARGER    R3 points to LARGER list.
LOOP    LDRB    R4, [R1], #1  Load X list byte into R4.
        LDRB    R5, [R2], #1  Load Y list byte into R5.
        CMP     R4, R5        Compare bytes.
        STRHSB  R4, [R3], #1  Store X byte if larger or same.
        STRLOB  R5, [R3], #1  Store Y byte if larger.
        SUBS    R0, R0, #1    Check if finished.
        BGT     LOOP
```

D.6. This solution assumes that the last number in the series of *n* numbers can be represented in a 32-bit word, and that $n > 2$.

```
        MOV     R0, N           Use R0 for loop count for numbers
        SUB     R0, R0, #2        generated after 1.
        ADR     R1, MEMLOC      Use R1 as memory pointer.
        MOV     R2, #0          Store first two numbers,
        STR     R2, [R1], #4      0 and 1, from R2
        MOV     R3, #1            and R3 into memory.
        STR     R3, [R1], #4
LOOP    ADD     R3, R2, R3      Starting with number i − 1
        STR     R3, [R1], #4      in R2 and i in R3, compute
                                  and place i + 1 in R3
                                  and store in memory.
        SUB     R2, R3, R2      Recover old i and place
                                  in R2.
        SUBS    R0, R0, #1      Branch back if all numbers
        BGT     LOOP              have not been computed.
```

D.7. Inspection of Table 1.1 of the ASCII characters shows that lowercase letters can be converted to uppercase letters by changing bit b_5 from 1 to 0.

A possible program for the conversion follows. Note that the last three instructions are executed conditionally (condition = NE), as described in Sections D.1.1 and D.9.

```
        LDR     R1, =WORD       Load address WORD.
NEXT    LDRB    R0, [R1]        Load character byte.
        CMP     R0, #&20        Check for space.
        BICNE   R0, R0, #&20    If not space, clear
        STRNEB  R0, [R1], #1      bit b5, store back,
        BNE     NEXT              and process next character.
```

D.8. Memory word location J contains the number of tests, *j*, and memory word location N contains the number of students, *n*. The list of student marks begins at memory word location LIST in the format shown in Figure 2.10. The parameter Stride = $4(j+1)$ is the distance in bytes between scores on a particular test for adjacent students in the list.

The Post-indexed addressing mode [R2], R3, LSL #2 is used to access the successive scores on a particular test in the inner loop. The value in register R2 before each entry to the inner loop is the address of the score on a particular test for the first student. Register R3 contains the value $j + 1$. Therefore, register R2 is incremented by the Stride parameter on each pass through the inner loop.

```
        LDR   R3, J                Load j + 1 into R3 to
        ADD   R3, R3, #1             be used as an address offset.
        ADR   R4, SUM              Initialize R4 to the sum
                                     location for test 1.
        ADR   R5, LIST             Load address of test 1 score
        ADD   R5, R5, #4             for student 1 into R5.
        LDR   R6, J                Initialize outer loop counter
                                     R6 to j.
OUTER   LDR   R7, N                Initialize inner loop
                                     counter R7 to n.
        MOV   R2, R5               Initialize base register R2
                                     to location of student 1 test
                                     score for next inner loop
                                     sum computation.
        MOV   R0, #0               Clear sum accumulator
                                     register R0.
INNER   LDR   R1, [R2], R3, LSL #2 Load test score into R1
                                     and increment R2 by Stride to
                                     point to next test score.
        ADD   R0, R0, R1           Accumulate score into R0.
        SUBS  R7, R7, #1           Check if all student scores
        BGT   INNER                  for current test are added.
        STR   R0, [R4], #4         Store sum in memory.
        ADD   R5, R5, #4           Increment R5 to next test
                                     score for student 1.
        SUBS  R6, R6, #1           Check if sums for all test
        BGT   OUTER                  scores have been accumulated.
```

D.9. A possible program that computes SUM = $580 + 68400 + 80000$.

```
LDR   R0, =580     Load 580.
LDR   R1, =68400   Load 68400.
ADD   R0, R0, R1   Generate 580 + 68400 in R0.
LDR   R1, =80000   Load 80000.
ADD   R0, R0, R1   Generate the final sum in R0.
STR   R0, SUM      Store the sum.
```

D.10. Program that computes ANSWER = $\mathrm{A} \times \mathrm{B} + \mathrm{C} \times \mathrm{D}$.

```
            AREA    CODE
            ENTRY
            LDR     R0, A              Load operand A.
            LDR     R1, B              Load operand B.
            MUL     R0, R1, R0         Generate A × B in R0.
            LDR     R1, C              Load operand C.
            LDR     R2, D              Load operand D.
            MULA    R0, R1, R2, R0     Generate C × D + [R0] in R0.
            STR     R0, ANSWER         Store the answer.

            AREA    DATA
A           DCD     100                Test data.
B           DCD     50
C           DCD     20
D           DCD     400
ANSWER      DCD     0                  Space for the answer.
```

D.11. The following program determines the number of negative integers in a list.

```
            AREA    CODE
            ENTRY
            LDR     R1, =NUMBERS       Load address of data list.
            LDR     R2, N              Load size of list.
            MOV     R0, #0             Clear negative number counter.
LOOP        LDR     R3, [R1], #4       Load next data item .
            CMP     R3, #0             Set condition code flags.
            ADDMI   R0, R0, #1         Increment counter if data item negative.
            SUBS    R2, R2, #1         Decrement loop counter
            BNE     LOOP                and branch back if not done.
            STR     R0, NEGNUM         Store result.

            AREA    DATA
N           DCD     6                  Size of list.
NEGNUM      DCD     0                  Count of negative numbers.
NUMBERS     DCD     25, −5, −128       Test data.
            DCD     44, −23, −9
```

D.12. A possible program for byte sorting as described in Example 2.5 follows. It has the same general structure as the solution in Example 2.5. The conditional execution feature of the ARM instruction set is used to advantage in the inner loop when LIST(*k*) must be interchanged with LIST(*j*). The three-instruction sequence STR, STR, MOV is only executed if LIST(*k*) is greater than LIST(*j*), as indicated by the GT suffixes. The forward conditional branch to NEXT in the solution to Example 2.5 is not needed in the ARM program.

```
        ADR     R4, LIST        Load list pointer register R4,
        LDR     R10, N            and initialize outer loop base
        ADD     R2, R4, R10       register R2 to LIST + n.
        ADD     R5, R4, #1      Load LIST + 1 into R5.
OUTER   LDRB    R0, [R2, #-1]!  Load LIST(j) into R0.
        MOV     R3, R2          Initialize inner loop base register
                                  R3 to LIST + n - 1.
INNER   LDRB    R1, [R3, #-1]!  Load LIST(k) into R1.
        CMP     R1, R0          Compare LIST(k) to LIST(j).
        STRGTB  R1, [R2]        If LIST(k) > LIST(j), interchange
        STRGTB  R0, [R3]          LIST(k) and LIST(j), and
        MOVGT   R0, R1            move (new) LIST(j) into R0.
        CMP     R3, R4          If k > 0, repeat
        BNE     INNER             inner loop.
        CMP     R2, R5          If j > 1, repeat
        BNE     OUTER             outer loop.
```

D.13. Assume that register R0 contains the address BOTTOM and that register R1 contains the address TOP.

```
SAFEPUSH  CMP   R5, R1
          BEQ   FULLERROR
          STR   R2, [R5, #-4]!


SAFEPOP   CMP   R5, R0
          BHI   EMPTYERROR
          LDR   R2, [R5], #4
```

D.14. No program is needed for the solution to this problem; so the solution is the same as that given for Problem 2.24.

D.15. See the solution to Problem 2.24 for the procedures needed to perform the append and remove operations.

Register assignment:

R0 – Data byte to append to or remove from queue

R1 – IN pointer

R2 – OUT pointer

R3 – Address of first queue byte location

R4 – Address of last queue byte location (= [R3] + k –1)

R5 – Auxiliary register for address of next appended byte.

Initially, the queue is empty with [R1] = [R2] = [R3]

APPEND routine:

```
MOV     R5, R1
ADD     R1, R1, #1      Increment R1 Modulo k.
CMP     R1, R4
MOVGT   R1, R3
CMP     R1, R2          Check if queue is full.
MOVEQ   R1, R5          If queue full, restore
BEQ     QUEUEFULL         IN pointer and send
                          message that queue is full.
STRB    R0, [R5]        If queue not full,
                          append byte and continue.
```

REMOVE routine:

```
CMP     R1, R2          Check if queue is empty.
BEQ     QUEUEEMPTY      If empty, send message.
LDRB    R0, [R2], #1    Otherwise, remove byte
CMP     R2, R4            and increment R2.
```

D.16. The weights stored in memory are not actually needed because multiplication of inputs by the weights 1/8, 1/4, 1/2 can be accomplished by arithmetic right shifts of the inputs by 3, 2, and 1 bit positions, respectively.

```
        LDR     R0, =IN                 Load addresses of first three
        ADD     R1, R0, #4                inputs into R0, R1, and R2.
        ADD     R2, R1, #4
        LDR     R5, =OUT                Load address of first output into R5.
        LDR     R6, N                   Number of outputs to be computed.
LOOP    LDR     R4, [R0], #4            Load first input.
        MOV     R4, R4, ASR #3          Multiply it by 1/8.
        LDR     R5, [R1], #4            Load second input.
        ADD     R4, R4, R5, ASR #2      Multiply it by 1/4 and
                                          add to first input.
        LDR     R5, [R2], #4            Load third input.
        ADD     R4, R4, R5, ASR #1      Multiply it by 1/2 and
                                          add it to sum.
        STR     R4, [R5], #4            Store output value.
        SUBS    R6, R6, #1              Decrement counter and branch
        BGT     LOOP                      back if not done.
```

D.17. The following program performs the copying operation as specified in the problem statement. However, if the *from* and *to* areas do not overlap, copying can actually be done in either direction. If the problem statement is relaxed to allow that possibility, then the five instructions following the STMFD instruction could be replaced by the two instructions

```
CMP    R2, R1
BLO    FORWARD
```

and the subroutine would perform the copying operation correctly. Also, we assume that the *from* and *to* addresses are not the same; but if they are, the given program does the copying in the BACKWARD loop.

Calling program:

```
LDR    R0, N          Load the length of the byte sequence.
LDR    R1, =FROM      Load the from address.
LDR    R2, =TO        Load the to address.
BL     MEMCOPY        Call the subroutine.
next instruction
```

Subroutine:

```
MEMCOPY     STMFD    R13!, {R3, R14}    Stack R3 and the link register.
            ADD      R3, R0, R1         Load R3 with from + length.
            CMP      R2, R1             Scan/copy forward if to is outside from area.
            BLO      FORWARD
            CMP      R2, R3
            BHS      FORWARD
BACKWARD    ADD      R1, R1, R0         Otherwise (if to is inside from area),
            ADD      R2, R2, R0           scan/copy backward from end of sequence.
            LDRB     R3, [R1, #−1]!
            STRB     R3, [R2, #−1]!
            SUBS     R0, R0, #1         Branch back if not finished copying.
            BGT      BACKWARD
            B        DONE
FORWARD     LDRB     R3, [R1], #1       Scan/copy forward.
            STRB     R3, [R2], #1
            SUBS     R0, R0, #1         Branch back if not done.
            BGT      FORWARD
DONE        LDMFD    R13!, {R3, R15}    Return.
```

D.18. Assume that the values *first* address, *second* address, and *length* of the sequence are passed to the subroutine in registers R1, R2, and R3, respectively. Use register R0 to return the count of the number of comparisons that do not match. The subroutine also uses registers R4 and R5, so they need to be saved/retored at the start/end of the subroutine.

Subroutine:

```
MEMCMP    STMFD    R13!, {R4, R5, R14}    Stack registers R4, R5, and the link register.
          MOV      R0, #0                 Clear “no-match” count register.
LOOP      LDRB     R4, [R1], #1           Load byte from first list.
          LDRB     R5, [R2], #1           Load byte from second list.
          CMP      R4, R5                 Compare bytes.
          ADDNE    R0, R0, #1             Increment counter if bytes don’t match.
          SUBS     R3, R3, #1             Branch back if not done.
          BGT      LOOP
          LDMFD    R13!, {R4, R5, R15}    Return from subroutine.
```

D.19. Assume that the address value STRNG is passed to the subroutine in register R0, and that the subroutine uses registers R1 and R2 as work registers that need to be saved/restored by the subroutine.

```
EXCLAIM  STMFD   R13!, {R1, R2, R14}   Save R1, R2, and the link register.
         MOV     R2, #&21              Load ASCII code for '!'.
LOOP     LDRB    R1, [R0], #1          Check for ASCII code '.' and
         CMP     R1, #&2E                replace with '!' if yes.
         STREQB  R2, [R0, #−1]
         CMP     R1, #&00              Check for ASCII code 'NUL'
         BNE     LOOP                    and read more characters if no.
         LDMFD   R13!, {R1, R2, R15}   Restore registers and return.
```

D.20. This problem is similar to Problem D.7. In that problem, it is assumed that all of the input text characters are lowercase letters. Here, the text can consist of any ASCII characters. Hence, a check must be made to determine whether or not each character scanned is a lowercase letter.

A possible subroutine program ALLCAPS follows. The subroutine assumes that the address STRNG is passed to it in register R0. The ASCII code for a lowercase letter can be converted to the corresponding uppercase letter by changing bit b_5 from 1 to 0. The lowercase letter codes are in the numerical range 97 through 122.

Subroutine:

```
ALLCAPS  STMFD   R13!, {R1, R14}
SCAN     LDRB    R1, [R0], #1     Load character.
         CMP     R1, #&00         Check for NUL character.
         BEQ     RETURN           Return on NUL.
         CMP     R1, #97          Check for lowercase.
         BLT     SCAN
         CMP     R1 , #122
         BGT     SCAN
         BIC     R1, R1, #&20     If lowercase, clear bit b5
         STRB    R1, [R0, #−1]      and store back.
         B       SCAN             Continue scanning.
RETURN   LDMFD   R13!, {R1, R15}  Return.
```

D.21. The calling program passes the address value STRNG to the subroutine in register R0, and the word count is returned in register R1. A possible subroutine is as follows.

Subroutine:

```
WORDS   STMFD   R13!, {R2, R14}   Stack work register R2 and the link register.
        MOV     R1, #0            Clear word counter.
LOOP    LDRB    R2, [R0], #1      Check for space character and
        CMP     R2, #&20            increment counter if yes.
        ADDEQ   R1, R1, #1
        CMP     R2, #&00          Check for NUL character and
        BNE     LOOP                continue scanning if no.
        LDMFD   R13!, {R2, R15}   Restore R2 and return.
```

D.22. Assume that the starting address of the ordered list of numbers, the length of the list (in words), and the new number to be inserted, are passed to the subroutine INSERT in registers R1, R2, and R3. Also assume that there may be sequences of two or more numbers that are the same at different places in the list, and that the list contains at least one number.

In the solution given, the subroutine moves through the list to locate the position of the first number that is larger than the new number to be inserted. This is the position where the new number is to be inserted after moving the sublist that starts at that position up by one word position.

Register R4 is first set to hold the address of the word after that of the last number in the list; register R1, which initially contains the starting address for the list, is used to step through the list searching for the correct insertion position; and the current list number to be compared to the new number is loaded into register R5. If the new number is equal to a number (or string of numbers) in the list, it will be inserted after that number (or string).

Subroutine:

```
INSERT  STMFD   R13!, {R4, R5, R14}
        ADD     R4, R1, R2, LSL #2    Set R4 to first word address
                                        beyond the end of the list.
        LDR     R5, [R4, #−4]         Check if the new number
        CMP     R3, R5                  should be stored after the
        BGE     STORE                   last number in the list.
LOOP1   LDR     R5, [R1], #4          Load current list number
        CMP     R3, R5                  into R5, increment pointer,
        BGE     LOOP1                   and continue through list
                                        if new number is greater than
                                        or equal to the current number.
        SUB     R1, R1, #4            Set R1 to starting address
                                        of numbers to be moved up.
LOOP2   LDR     R5, [R4, #−4]         Shift numbers up,
        STR     R5, [R4], #−4           starting from end of list.
        CMP     R4, R1
        BGT     LOOP2
STORE   STR     R3, [R4]              Store new number in list.
        LDMFD   R13!, {R4, R5, R15}
```

D.23. The address OLDLIST, the length of the list in words, and the address NEWLIST, are passed to the subroutine INSERTSORT in registers R6, R7, and R8, respectively. Assume that the unordered list contains at least one number.

Subroutine:

```
INSERTSORT  STMFD  R13!, {R1, R2, R3, R14}
            MOV    R1, R8                    Initialize R1 to address NEWLIST.
            LDR    R3, [R6], #4              Load first number from old list
                                               into R3 and increment pointer.
            STR    R3, [R1]                  Store this number in new list.
            SUBS   R7, R7, #1                Decrement old list count.
            BEQ    RETURN                    Return if old list had
                                               only one number.
            MOV    R2, #1                    Otherwise, load R2 and R3
            MOV    R3, [R6], #4                before calling INSERT.
LOOP        BL     INSERT                    (See Problem 2.32 for a
                                               description of INSERT.)
            MOV    R1, R8                    Set up R1, R2, and R3
            ADD    R2, R2, #1                  before calling INSERT again.
            LDR    R3, [R6], #4
            SUBS   R7, R7, #1                Check if all numbers have
            BNE    LOOP                        been inserted into the new list.
RETURN      LDMFD  R13!, {R1, R2, R3, R15}   If yes, Return.
```

D.24. The following program is similar to that in Figure D.17. There are two differences: A specified number of characters, *n* = [N], are read from the keyboard, instead of reading up to a carriage return character; and the characters are pushed onto a user stack instead of being written into memory before they are displayed.

```
        LDR    R0, N            Use R0 as the loop counter
                                  for reading n characters.
READ    LDRB   R3, [R1, #4]     Load KBD_STATUS byte and
        TST    R3, #2             wait for character.
        BEQ    READ
        LDRB   R3, [R1]         Read character and push
        STRB   R3, [R6, #−1]!     onto stack.
ECHO    LDRB   R4, [R2, #4]     Load DISP_STATUS byte and
        TST    R4, #4             wait for display.
        BEQ    ECHO
        STRB   R3, [R2]         Send character
                                  to display.
        SUBS   R0, R0, #1       Repeat until n
        BGT    READ               characters read.
```

D.25. Assume that almost all of the time between successive characters being struck is spent in the three-instruction wait loop that starts at location READ. The BEQ READ instruction is executed once every 15 ns (3×5 ns) while this loop is being executed. There are $10^9/10 = 10^8$ ns between successive characters. Therefore, the BEQ READ instruction is executed $10^8/15 = 6.7 \times 10^6$ times per character entered.

D.26. Main Program:

```
READLINE    BL      GETCHAR        Load character into R3.
            STRB    R3, [R0], #1   Store character in memory.
            BL      PUTCHAR        Display character.
            TEQ     R3, #CR        Check for carriage return character.
            BNE     READLINE
```

Subroutine GETCHAR:

```
GETCHAR     LDRB    R3, [R1, #4]   Wait for character.
            TST     R3, #2
            BEQ     GETCHAR
            LDRB    R3, [R1]       Load character into R3.
            MOV     R15, R14       Return.
```

Subroutine PUTCHAR:

```
PUTCHAR     STMFD   R13!, {R4, R14}   Save R4 and Link register.
DISPLAY     LDRB    R4, [R2, #4]      Wait for display.
            TST     R4, #4
            BEQ     DISPLAY
            STRB    R3, [R2]          Send character to display.
            LDMFD   R13!, {R4, R15}   Restore R4 and Return.
```

D.27. Address KBD_DATA is passed to GETCHAR on the stack; the character read is passed back in the same stack position. The character to be displayed and the address DISP_DATA are passed to PUTCHAR by being pushed onto the stack in that order. Register R0 contains the address of the memory buffer area.

Main Program:

```
READLINE  LDR    R5, =KBD_DATA     Load address KBD_DATA
          STR    R5, [SP, #−4]!      into R5 and push onto stack.
          BL     GETCHAR           Call subroutine to read character.
          LDRB   R5, [SP]          Load character from top of
          STRB   R5, [R0], #1        stack, leaving it on the stack,
                                     and store it in memory.
          LDR    R5, =DISP_DATA    Load address DISP_DATA into
          STR    R5, [SP, #−4]!      R5, and push onto stack.
          BL     PUTCHAR           Call character display subroutine.
          ADD    SP, SP, #8        Remove parameters from stack.
          TEQ    R5, #CR           Check for carriage return character.
          BNE    READLINE
```

Subroutine GETCHAR:

```
GETCHAR   STMFD  SP!, {R1, R3, LR}   Save registers.
          LDR    R1, [SP, #12]       Load address KBD_DATA into R1.
READ      LDRB   R3, [R1, #4]        Wait for character.
          TST    R3, #2
          BEQ    READ
          LDRB   R3, [R1]            Load character into R3
          STRB   R3, [SP, #12]         and overwrite address KBD_DATA
                                       on stack.
          LDMFD  SP!, {R1, R3, PC}   Restore registers and Return.
```

Subroutine PUTCHAR:

```
PUTCHAR   STMFD  SP!, {R2−R4, LR}    Save registers.
          LDR    R2, [SP, #16]       Load address DISP_DATA into
          LDR    R3, [SP, #20]         R2 and character into R3.
DISPLAY   LDR    R4, [R2, #4]        Wait for display.
          TST    R4, #4
          BEQ    DISPLAY
          STRB   R3, [R2]            Send character to display.
          LDMFD  SP!, {R2−R4, PC}    Restore registers and Return.
```

D.28. The first program section reads the characters, stores them in a 3-byte area beginning at CHARSTR, and echoes them to a display. The second section does the conversion to binary and stores the result in BINARY. The I/O device addresses KBD_DATA and DISP_DATA are in registers R1 and R2.

```
         LDR    R0, =CHARSTR          Initialize memory pointer
         MOV    R5, #3                  R0 and counter R5.
READ     LDRB   R3, [R1, #4]          Read a character and
         TST    R3, #2                  store it in memory.
         BEQ    READ
         LDRB   R3, [R1]
         STRB   R3, [R0], #1
ECHO     LDRB   R4, [R2, #4]          Echo the character
         TST    R4, #4                  to the display.
         BEQ    ECHO
         STRB   R3, [R2]
         SUBS   R5, R5, #1            Check if all three
         BGT    READ                    characters have been read.
CONVERT  LDR    R0, =CHARSTR          Initialize memory pointers
         LDR    R1, =HUNDREDS           R0, R1, and R2.
         LDR    R2, =TENS
         LDRB   R3, [R0], #1          Load high-order BCD digit
         AND    R3, R3, #&F             into R3.
         LDR    R4, [R1, R3, LSL #2]  Load binary value
                                        corresponding to decimal
                                        hundreds value into
                                        accumulator register R4.
         LDRB   R3, [R0], #1          Load middle BCD digit
         AND    R3, R3, #&F             into R3.
         LDR    R3, [R2, R3, LSL #2]  Load binary value
                                        corresponding to
                                        decimal tens value
                                        into register R3.
         ADD    R4, R4, R3            Accumulate into R4.
         LDRB   R3, [R0], #1          Load low-order BCD digit
         AND    R3, R3, #&F             into R3.
         ADD    R4, R4, R3            Accumulate into R4.
         STR    R4, BINARY            Store converted value
                                        into location BINARY.
```

D.29. (*a*) The names FP, SP, LR, and PC, are used for registers R12, R13, R14, and R15 (frame pointer, stack pointer, link register, and program counter). The 3-byte memory area for the characters begins at address CHARSTR; and the converted binary value is stored at BINARY.

The first subroutine, labeled READCHARS, is patterned after the program in Figure D.17. It echoes the characters back to a display as well as reading them into memory. The second subroutine is labeled CONVERT.

The stack frame format used is like Figure D.12.

A possible program, including subroutines, is:

Main program:

```
            LDR     R10, =CHARSTR    Load parameters into
            LDR     R11, =BINARY       R10 and R11 and
            STMFD   SP!, {R10, R11}    push them onto the stack.
            BL      READCHARS        Branch to first subroutine.
RTNADDR     ADD     SP, SP, #8       Remove two parameters
            . . .                      from stack and continue.
```

First subroutine READCHARS:

```
READCHARS   STMFD   SP!, {R0−R5, FP, LR}   Save registers
                                             on stack.
            ADD     FP, SP, #28            Set up frame
                                             pointer.
            LDR     R0, [FP, #4]           Load R0, R1,
            LDR     R1, =KBD_DATA            and R2 with
            LDR     R2, =DISP_DATA           parameters.
            MOV     R5, #3                 Load counter R5.
READ        LDRB    R3, [R1, #4]           Read a character and
            TST     R3, #2                   store it in memory.
            BEQ     READ
            LDRB    R3, [R1]
            STRB    R3, [R0], #1
ECHO        LDR     R4, [R2]               Echo the character
            TST     R4, #4                   to the display.
            BEQ     ECHO
            STRB    R3, [R2]
            SUBS    R5, R5, #1             Check if all three
            BGT     READ                     characters have been read.
            LDR     R0, [FP, #8]           Load R0,R1,R2
            LDR     R5, [FP, #12]            and R5 with
            LDR     R1, =HUNDREDS            parameters.
            LDR     R2, =TENS
            BL      CONVERT                Call second
                                             subroutine.
            LDMFD   SP!, {R0−R5, FP, PC}   Return to
                                             Main program.
```

Second subroutine CONVERT:

```
CONVERT  STMFD  SP!, {R3, R4, FP, LR}  Save registers
                                         on stack.
         ADD    FP, SP, #8             Set up frame
                                         pointer.
         LDRB   R3, [R0], #1           Load high-order BCD digit
         AND    R3, R3, #&F              into R3.
         LDR    R4, [R1, R3, LSL #2]   Load binary value
                                         corresponding to decimal
                                         hundreds value into
                                         accumulator register R4.
         LDRB   R3, [R0], #1           Load middle BCD digit
         AND    R3, R3, #&F              into R3.
         LDR    R3, [R2, R3, LSL #2]   Load binary value
                                         corresponding to
                                         decimal tens value
                                         into register R3.
         ADD    R4, R4, R3             Accumulate into R4.
         LDRB   R3, [R0], #1           Load low-order BCD digit
         AND    R3, R3, #&F              into R3.
         ADD    R4, R4, R3             Accumulate into R4.
         STR    R4, [R5]               Store binary
                                         number.
         LDMFD  SP!, {R3, R4, FP, PC}  Return to
                                         first subroutine.
```

(b) The contents of the top of the stack immediately after the call to the CONVERT subroutine are:

	[R0]
	[R1]
	[R2]
	[R3]
	[R4]
	[R5]
FP →	[FP]
	[LR] = RTNADDR
	CHARSTR
	BINARY
	Original TOS

D.30. A 16-entry table of ASCII character codes, starting at memory location TABLE, for the characters 0, 1, ... , 9, A, B, C, D, E, F, is used by the following program to generate the hexadecimal character codes, two per byte, for the ten bytes starting at memory location LOC.

The display device has the interface depicted in Figure 3.3*b*.

Assume that the character for the high-order four bits of each byte is displayed first. The two character codes for each byte are loaded into registers R1 and R2. Register R0 is initialized to the count value 10.

Program-controlled I/O is used. A display buffer memory area of 30 bytes, starting at DISPBUFF, is used to assemble the two characters, followed by a space character, for each of the ten bytes to be displayed.

```
            AREA    CODE
            ENTRY
            MOV     R0, #10          Load byte count.
            LDR     R3, =TABLE       Load addresses.
            LDR     R4, =LOC
            LDR     R5, =DISPBUFF
            MOV     R7, #&20         Load space character code.
LOOP        LDRB    R6, [R4], #1     Load byte to be displayed.
            MOV     R1, R6, LSR #4   Shift high-order digit
                                       into low-order digit position.
            LDRB    R1, [R3, R1]     Load hex digit code
            STRB    R1, [R5], #1       and store it in buffer.
            AND     R2, R6, #&F      Clear high-order digit.
            LDRB    R2, [R3, R2]     Load hex digit code
            STRB    R2, [R5], #1       and store it in buffer.
            STRB    R7, [R5], #1     Store space character in next byte.
            SUBS    R0, R0, #1       Check if all 10 bytes processed.
            BNE     LOOP
DISPLAY     MOV     R0, #30          Load count of characters
                                       to be displayed.
            LDR     R5, =DISPBUFF    Load buffer address.
            LDR     R2, =DISP_DATA   Load I/O device address.
WRITEWAIT   LDRB    R4, [R2, #4]     Display 30 characters.
            TST     R4, #4
            BEQ     WRITEWAIT
            LDRB    R3, [R5], #1
            STRB    R3, [R2]
            SUBS    R0, R0, #1
            BNE     WRITEWAIT
            next instruction

            AREA    DATA
TABLE       DCB     &30, &31, &32, &33
                    &34, &35, &36, &37
                    &38, &39, &41, &42
                    &43, &44, &45, &46
DISPBUFF    SPACE   30
```

D.31. Assume (big-endian assumption) that the 16 bits to be displayed are in the high-order half of the word at memory location BINARY, and that the high-order bits are to be displayed first.

```
           LDR     R2, =DISP_DATA   Load I/O device address.
           LDR     R1, =&80000000   R1 has a single 1 in
                                      the high-order bit position.
           LDR     R7, BINARY       16-bit pattern is in high-order
                                      half of register R7.
           MOV     R0, #16          R0 is the counter for displayed bits.
           MOV     R5, #&30         Load ASCII code for 0.
           MOV     R6, #&31         Load ASCII code for 1.
LOOP       TST     R7, R1           Check if high-order bit of R7 is 1.
           MOVNE   R3, R6           If it is, load ASCII code
                                      for 1 into R3.
           MOVEQ   R3, R5           If bit is 0, load ASCII code
                                      for 0 into R3.
WRITEWAIT  LDRB    R4, [R2, #4]     Display bit character.
           TST     R4, #4
           BEQ     WRITEWAIT
           STRB    R3, [R2]
           MOV     R7, R7, LSL #1   Move next bit into high-order
                                      position of register R7.
           SUBS    R0, R0, #1       Check if done.
           BGT     LOOP
```

D.32. Assume that the 7-segment display device is driven by an 8-bit interface register at address location SEVEN, and that the first register in the timer interface is at address location TIM_STATUS, as shown in Figure 3.14. The number of 100-MHz clock cycles in a 1-second time period is given by the hexadecimal number 5F5E100. The following possible program uses polling to detect when the timer reaches the end of each 1-second period.

```
          AREA    CODE
          ENTRY

          LDR     R2, =SEVEN          Load device interface addresses.
          LDR     R3, =TIM_STATUS
          LDR     R8, =TABLE          Load address of
                                        display patterns.
          LDR     R4, =&5F5E100       Load 1-second timer count.
          STR     R4, [R3, #8]        Store count in device register.
          MOV     R4, #6              Start timer in continuous
          STRB    R4, [R3, #4]          down counting mode.
          MOV     R5, #0              Clear digit counter.
LOOP      LDRB    R6, [R3]            Wait for timer to reach
          TST     R6, #2                end of 1-second period.
          BEQ     LOOP
          LDRB    R7, [R8, R5]        Look up 7-segment pattern
          STRB    R7, [R2]              and display it.
          ADD     R5, R5, #1          Increment digit counter and
          CMP     R5, #10               loop back if less than 10.
          BLT     LOOP
          MOV     R5, #0              Otherwise, reset counter to 0
          B       LOOP                  and loop back.

          AREA    DATA
TABLE     DCB     &7E, &30, &6D, &79
          DCB     &33, &5B, &5F, &70
          DCB     &7F, &7B
```

D.33. Assume that the two 7-segment display devices are driven by a 16-bit interface register at address location SEVENS2, and that the first register in the timer interface is at address location TIM_STATUS, as shown in Figure 3.14. The number of 100-MHz clock cycles in a 1-second time period is given by the hexadecimal number 5F5E100. The following possible program uses polling to detect when the timer reaches the end of each 1-second period.

```
        AREA    CODE
        ENTRY

        LDR     R2, =SEVENS2          Load device interface addresses.
        LDR     R3, =TIM_STATUS
        LDR     R10, =TABLE           Load address of display patterns.
        LDR     R4, =&5F5E100         Load 1-second timer count.
        STR     R4, [R3, #8]          Store count in timer register.
        MOV     R4, #6                Start timer in continuous
        STRB    R4, [R3, #4]            down counting mode.
        MOV     R5, #0                Clear low-order digit counter.
        MOV     R6, #0                Clear high-order digit counter.
LOOP    LDRB    R7, [R3]              Wait for timer to reach
        TST     R7, #2                  end of 1-second period.
        BEQ     LOOP
        LDRB    R8, [R10, R5]         Load 7-segment pattern
                                        for low-order digit.
        LDRB    R9, [R10, R6]         Load 7-segment pattern
                                        for high-order digit.
        ORR     R8, R8, R9, LSL #8    Place high-order digit
                                        pattern in high-order byte
                                        of half word in register R8.
        STRH    R8, [R2]              Display two digit patterns.
        ADD     R5, R5, #1            Increment low-order digit counter.
        CMP     R5, #10               Loop back if count is
        BLT     LOOP                    less than 10.
        MOV     R5, #0                Otherwise, reset counter to 0.
        ADD     R6, R6, #1            Increment high-order digit counter.
        CMP     R6, #10               Loop back if count is
        BLT     LOOP                    less than 10.
        MOV     R6, #0                Otherwise, reset counter to 0
        B       LOOP                    and loop back.

        AREA    DATA
TABLE   DCB     &7E, &30, &6D, &79
        DCB     &33, &5B, &5F, &70
        DCB     &7F, &7B
```

D.34. Assume that the four 7-segment display devices are driven by the bytes of a 32-bit device interface register at address location SEVENS4, and that the first register in the timer interface is at address location TIM_STATUS, as shown in Figure 3.14. The number of 100-MHz clock cycles in a 1-second time period is given by the hexadecimal number 5F5E100. Register pair R1,R0 is used to hold the minute count (00 to 59), and register pair R3,R2 is used to hold the hour count (00 to 23). The following possible program uses polling to detect when the timer reaches the end of each 1-second period.

```
                AREA    CODE
                ENTRY
                LDR     R4, =TABLE          Load address of table of 7-segment
                                              patterns for the digits 0 through 9.
                LDR     R5, =SEVENS4        Load display device address.
                LDR     R6, =TIM_STATUS     Load address of timer device.
                LDR     R7, =&5F5E100       Load 1-second timer count.
                MOV     R0, #0              Clear minute count.
                MOV     R1, #0
                MOV     R2, #0              Clear hour count.
                MOV     R3, #0
                STR     R7, [R6, #8]        Load timer count and mode for
                MOV     R7, #6                continuous down counting operation.
                STRB    R7, [R6, #4]
DELAY           MOV     R7, #0              Delay for 1 minute.
LOOP            LDRB    R8, [R6]            Wait for 1 second.
                TST     R8, #2
                BEQ     LOOP
                ADD     R7, R7, #1          Increment second count.
                CMP     R7, #60             Check if 60 seconds.
                BLT     LOOP                Loop back if not at 60.
                BL      DISPLAY             Otherwise, display time.
INCRCLK         ADD     R0, R0, #1          Increment real time clock.
                CMP     R0, #10             First, increment minutes from 00 to 59,
                BLT     DELAY                 delay for 1 minute, then display time.
                MOV     R0, #0
                ADD     R1, R1, #1
                CMP     R1, #6
                BLT     DELAY
                MOV     R1, #0
                CMP     R3, #2              Then, increment hours from 00 to 23.
                BEQ     TWOON3              Low-order digit of hours cycles
NOT2ON3         ADD     R2, R2, #1            from 0 through 9 while
                CMP     R2, #10               high-order digit is less than 2.
                BLT     DELAY
                MOV     R2, #0
                ADD     R3, R3, #1
                B       DELAY
TWOON3          ADD     R2, R2, #1          When high-order digit is 2,
                CMP     R2, #4                increment R2 only to 3.
                BLT     DELAY
                MOV     R2, #0              Then, hours are reset to 00 on
                MOV     R3, #0                the minute increment after 23 59
                B       DELAY                 has been reached, and time is displayed.
```

```
DISPLAY   STMFD   R13!, {R10, R11, R14}
          LDRB    R10, [R4, R0]            Pack the four 7-segment
          LDRB    R11, [R4, R1]              patterns, corresponding to the
          ORR     R10, R10, R11, LSL #8      hours and minutes digits, into R10.
          LDRB    R11, [R4, R2]
          ORR     R10, R10, R11, LSL #16
          LDRB    R11, [R4, R3]
          ORR     R10, R10, R11, LSL #24
          STR     R10, [R5]                Send packed patterns to display.
          LDMFD   R13!, {R10, R11, R15}

          AREA    DATA
TABLE     DCB     &7E, &30, &6D, &79
          DCB     &33, &5B, &5F, &70
          DCB     &7F, &7B
```

D.35. Assume that the main program starts in the Supervisor mode. This allows the execution of the privileged instruction MSR, which manipulates bits in the status register, CPSR.

Also assume that the timer device is the only device enabled to raise interrupts, and that its interrupt request signal is connected to the IRQ request line. The first register in the timer interface is at address location TIM_STATUS, as shown in Figure 3.14. The number of 100-MHz clock cycles in a 10-second time period is given by the hexadecimal number 3B9ACA00.

Main program:

```
            LDR    R1, =TIM_STATUS   Load timer device address.
            LDR    R2, =&3B9ACA00    LOAD 10-second count
            STR    R2, [R1, #8]        and store in timer device.
            LDR    R2, #7            Load timer mode for continuous
            STRB   R2, [R1, #4]        down counting operation
                                       with interrupt enabled.
            MOV    R2, #&50          Switch processor to User mode
            MSR    CPSR, R2            and clear IRQ disable bit.
COMPUTE     next instruction
```

Interrupt-service routine:

```
IRQLOC   STMFD   R13!, {R0}      Save register R0.
         LDRB    R0, [R1]        Clear TIRQ and ZERO bits
                                   in timer device status register.
         BL      DISPLAY         Display updated results
                                   from COMPUTE routine.
RETURN   LDMFD   R13!, {R0}      Restore register R0
         SUBS    R15, R14, #4      and return.
```

D.36. An ARM program corresponding to Figure 3.18 is given as follows:

```
         AREA    CODE
         ENTRY
         LDR     R1, =&800         Load address DIGIT.
         LDR     R7, =&4030        Load address of 7-segment display.
         LDR     R6, =TABLE        Load address of table of
                                     7-segment patterns.
         LDRB    R2, [R1]          Load ASCII-encoded digit.
         AND     R3, R2, #&F0      Extract high-order bits of code.
         AND     R2, R2, #&F       Extract BCD number.
         CMP     R3, #&30          Check if high-order bits
         BEQ     HIGH3               are 0011.
         MOV     R2, #&F           If no, load index to blank.
HIGH3    LDRB    R5, [R6, R2]      Load 7-segment pattern.
         STRB    R5, [R7]          Display the digit.

         AREA    DATA
TABLE    DCB     &7E, &30, &6D, &79
         DCB     &33, &5B, &5F, &70
         DCB     &7F, &7B, &00, &00
         DCB     &00, &00, &00, &00
```

D.37. The prompt is “Name: ”, after which the user types a name, followed by carriage return, which is echoed back on the same line and stored in memory, starting at location NAME. The next line leads off with the message “Name Reversed: ”, after which the user name is displayed in reverse.

The two subroutines for reading and displaying characters used by the following ARM program use registers R1, R2, R3, and R4. These registers are not used in the main program, so they are not saved/restored on subroutine entry/exit. The subroutines are patterned after the routines in Section D.8.1.

The ASCII characters for the prompt and the message are stored in byte strings starting at memory locations PROMPT and MESSAGE, respectively. Each of these strings is terminated by a space character.

Main program:

```
                AREA    CODE
                ENTRY

                LDR     R0, =NAME           Load address for user name.
                LDR     R1, =KBD_DATA       See Figure 3.3 for I/O
                LDR     R2, =DISP_DATA        device addresses.
                MOV     R10, #&20           Load ASCII code for space.
                MOV     R11, #&0D           Load ACII code for carriage return.
                LDR     R6, =PROMPT         R6 points to prompt characters.
LOOP1           LDRB    R3, [R6], #1        Display prompt, which is
                BL      WRITECHAR             terminated by a space character.
                CMP     R3, R10             Check for space.
                BNE     LOOP1               If not, loop back.
LOOP2           BL      READCHAR            Read user name and store
                STRB    R3, [R0], #1          at NAME.
                BL      WRITECHAR           Echoback name.
                CMP     R3, R11             Check for carriage return.
                BNE     LOOP2               If not, loop back.
                LDR     R6, =MESSAGE        R6 points to message characters.
LOOP3           LDRB    R3, [R6], #1        Display message, which is
                BL      WRITECHAR             terminated by a space character.
                CMP     R3, R10             Check for space.
                BNE     LOOP3               If not, loop back.
                LDR     R6, =NAME           Load address for user name.
                SUB     R0, R0, #1          Decrement R0 to point to
                                              carriage return following user name.
LOOP4           LDRB    R3, [R0, #−1]!      Display user name in reverse.
                BL      WRITECHAR
                CMP     R6, R0              Has first letter of name
                                              been displayed?
                BNE     LOOP4               If not, loop back.
                next instruction
```

Subroutines:

```
READCHAR    LDRB    [R1, #4]            Subroutine to read character.
            TST     R3, #2
            BEQ     READCHAR
            LDRB    R3, [R1]
            MOV     R15, R14            Return.

WRITECHAR   LDRB    R4, [R2, #4]        Subroutine to display character.
            TST     R4, #4
            BEQ     WRITECHAR
            STRB    R3, [R2]
            MOV     R15, R14            Return.


            AREA    DATA
PROMPT      DCB     &4E, &61, &6D, &65
                    &3A, &20
MESSAGE     DCB     &4E, &61, &6D, &65
                    &20, &72, &65, &76
                    &65, &72, &73, &65
                    &64, &3A &20
NAME        SPACE   40
```

D.38. The following possible program assumes that there are at least two characters in the word entered by the user. The characters are stored starting at the memory location WORD. Subroutines are used to read and display characters using the routines shown in Section D.8.1. They use registers R1 through R4, which are not saved/restored in the subroutines. The prompt character is '*'; and the result of the palindrome check is 'Y' (yes) or 'N' (no).

Main program:

```
            LDR     R1, =KBD_DATA    Load addresses for I/O devices.
            LDR     R2, =DISP_DATA
            LDR     R5, =WORD
            MOV     R6, R5           R6 is a pointer that is used to
                                       step through the character locations.
            MOV     R3, #&2A         Display the prompt character,
            BL      WRITECHAR          '*' (&2A), followed by
            MOV     R3, #&0D           a carriage return character (&0D).
            BL      WRITECHAR
GETCHARS    BL      READCHAR         Read a character typed by the user.
            STRB    R3, [R6], #1     Store it in memory and
                                       increment address pointer.
            BL      WRITECHAR        Echo character back to display.
            CMP     R3, #&0D         Check for carriage return and
            BNE     GETCHARS           proceed to palindrome analysis
                                       if yes; otherwise, continue reading.
            SUBS    R6, R6, #2       Decrement R6 to point to last
                                       character of word entered
                                       and start palindrome analysis.
            MOV     R3, #&59         Anticipate "yes" answer.
COMPCHARS   LDRB    R7, [R5], #1     Working in from both ends of
            LDRB    R8, [R6], #−1      word, check if characters match.
            CMP     R7, R8
            MOVNE   R3, #&4E         If they do not, change "yes" to "no".
            CMP     R6, R5           See Note below.
            BGT     COMPCHARS        If not finished checking,
                                       branch back to continue.
            BL      WRITECHAR        Otherwise, display answer.
            next instruction
```

Note: Character checking has been completed when the pointers become equal (for an odd length word) or they cross each other (for an even length word). Otherwise, checking continues while R6 is greater than R5.

Subroutines:

```
READCHAR    LDRB    R3, [R1, #4]
            TST     R3, #2
            BEQ     READCHAR
            LDRB    R3, [R1]
            MOV     R15, R14        Return.

WRITECHAR   LDRB    R4, [R2, #4]
            TST     R4, #4
            BEQ     WRITECHAR
            STRB    R3, [R2]
            MOV     R15, R14        Return.
```

D.39. The first part of the main program below determines the length of the sample text to be displayed and writes the length in register R0. The expression

$$[80 - ([\mathrm{R0}] + 2)]/2 = 39 - [\mathrm{R0}]/2$$

gives the number of leading space characters needed to center the display.

Three subroutines are used. LENGTH determines the length of the given character string located at memory location STRING. DISPA displays the upper or lower line of the display box. WRITECHAR displays a single character, using the routine given in Section D.8.1.

Main program:

```
        LDR     R1, =STRING      Load address of string.
        MOV     R0, #0           Clear register R0.
        BL      LENGTH           Compute and write length of string into R0.
        CMP     R0, #78          If length greater than 78,
        MOVGT   R0, #78            truncate to 78.
                                 Length of sample text is now in R0.
        MOV     R1, R0, ASR #1   Write [R0]/2 into R1.
        RSB     R1, R1, #39      Number of leading spaces
                                   is now in R1.
        MOV     R10, R0          Load length of sample text into R10.
        MOV     R11, R1          Load number of leading spaces into R11.
        BL      DISPA            Display upper line of bounding box,
                                   followed by carriage return.
        MOV     R10, R0          Display sample text between
        MOV     R11, R1            vertical lines.
        MOV     R3, #&20         Load space character
LOOP1   SUBS    R11, R11, #1       and display 0 or more leading spaces.
        BLT     DISP2
        BL      WRITECHAR
        B       LOOP1
DISP2   MOV     R3, #&7C         Display vertical character.
        BL      WRITECHAR
        LDR     R5, =STRING      Display sample text.
LOOP2   LDRB    R3, [R5], #1
        BL      WRITECHAR
        SUBS    R10, R10, #1
        BNE     LOOP2
        MOV     R3, #&7C         Display vertical character.
        BL      WRITECHAR
        MOV     R3, #&0D         Send carriage return.
        BL      WRITECHAR
        MOV     R10, R0          Display bottom line of
        MOV     R11, R1            bounding box.
        BL      DISPA
        next instruction
```

Subroutines:

```
LENGTH      LDRB    R5, [R1], #1    Determine length of string
            CMP     R5, #&00          and return in R0.
            ADDNE   R0, R0, #1
            BNE     LENGTH
            MOV     R15, R14        Return.

DISPA       MOV     R3, #&20        ASCII code for space.
LOOP3       SUBS    R11, R11, #1    Display zero or more
            BLT     DISP1             leading space characters.
            BL      WRITECHAR
            B       LOOP3
DISP1       MOV     R3, #&2B        Display '+' character.
            BL      WRITECHAR
            MOV     R3, #&2D        Display one or more '-' characters.
LOOP4       BL      WRITECHAR
            SUBS    R10, R10, #1
            BNE     LOOP4
            MOV     R3, #&2B        Display '+' character.
            BL      WRITECHAR
            MOV     R3, #&0D        Send carriage return.
            BL      WRITECHAR
            MOV     R15, R14        Return.

WRITECHAR   LDRB    R4, [R2, #4]    Assume R2 contains DISP_DATA
            TST     R4, #4            and the next word contains DISP_STATUS.
            BEQ     WRITECHAR
            STRB    R3, [R2]        R3 contains character to be displayed.
            MOV     R15, R14        Return.
```

D.40. Assume that the text to be displayed is stored in the memory starting at location TEXT. Also, assume that words consist of from 1 to 80 characters, and that the text contains at least one word.

The following program is a possible solution. The program does a complete scan of the text, replacing some of the space characters with carriage returns to meet the stated requirements. During this preprocessing of the text, no lines are sent to the display. After the preprocessing has been completed, the text is displayed. The NUL character is replaced by a carriage return when the text is displayed.

Registers R2, R3, and R4 are used by the single character output subroutine, patterned after the routine shown in Section D.8.1, assuming the interface arrangement shown in Figure 3.3*b*.

```
           LDR     R2, =DISP_DATA   Load I/O device address.
           LDR     R0, =TEXT        Register R0 points to start of text.
           MOV     R1, #&0D         R1 contains carriage return character.
RESET      MOV     R5, #0           Clear character counter.
SCAN       LDRB    R6, [R0], #1     Start/continue scanning text.
           ADD     R5, R5, #1       Increment character count.
           CMP     R6, #00          If character is NUL,
           BEQ     DISPLAY            display text.
           CMP     R5, #81          If not at 81st character,
           BLT     SCAN               continue scanning.
           CMP     R6, #&20         If 81st character is a space,
           STREQB  R1, [R0, #−1]      replace with carriage return,
           BEQ     RESET              reset counter, and continue scanning.
BACKUP     LDRB    R6, [R0, #−1]!   Otherwise, backup until
           CMP     R6, #&20           space is encountered.
           BNE     BACKUP
           STRB    R1, [R0], #1     When space is encountered,
           B       RESET              replace it with a carriage return,
                                      reset counter, and continue scanning.
DISPLAY    LDR     R0, =TEXT        Starting at TEXT, display
DISPLAY1   LDRB    R3, [R0], #1       characters until NUL
           CMP     R3, #& 00          is reached.
           BEQ     FINISH
           BL      WRITECHAR
           B       DISPLAY1
FINISH     MOV     R3, R1           When NUL is reached,
           BL      WRITECHAR          display carriage return.
           next instruction


WRITECHAR  LDRB    R4, [R2, #4]     Subroutine to display character.
           TST     R4, #4
           BEQ     WRITECHAR
           STRB    R3, [R2]
           MOV     R15, R14         Return.
```