Chapter 3
Basic Input/Output

3.1. To ensure that the given data are read only once.

3.2. Assuming the display interface registers in Figure 3.3, the following program can be used:

```
            Move          R2, #LOC          Get the address LOC.
            Move          R3, #DISP_DATA    Get the address of display.
            Move          R4, #10           Initialize the byte counter.
LOOP:       LoadByte      R5, (R2)          Load a byte.
            And           R6, R5, #0xF0     Select the high-order 4 bits.
            LShiftR       R6, R6, #4        Shift right by 4 bit positions.
            LoadByte      R6, TABLE(R6)     Get the character for display.
            Call          DISPLAY
            And           R6, R5, #0x0F     Select the low-order 4 bits.
            LoadByte      R6, TABLE(R6)     Get the character for display.
            Call          DISPLAY
            Move          R6, #0x20         ASCII code for SPACE.
            Call          DISPLAY
            Add           R2, R2, #1        Increment the pointer.
            Subtract      R4, R4, #1        Decrement the byte counter.
            Branch_if_[R4]>[R0] LOOP        Branch back if not finished.
            next instruction

DISPLAY:    LoadByte      R7, 4(R3)
            And           R7, R7, #4        Check the DOUT flag.
            Branch_if_[R7]=[R0] DISPLAY
            StoreByte     R6, (R3)          Send the character to display.
            Return

            ORIGIN        0x1000
TABLE:      DATABYTE      0x30,0x31,0x32,0x33   Table that contains
            DATABYTE      0x34,0x35,0x36,0x37    the necessary
            DATABYTE      0x38,0x39,0x0041,0x42  ASCII characters.
            DATABYTE      0x43,0x44,0x45,0x46
```

3.3. A subroutine is called by a program instruction to perform a function needed by the calling program. An interrupt-service routine is initiated by an event such as an input operation or a hardware error. The function it performs may not be at all related to the program being executed at the time of interruption. Hence, it must not affect any of the data or status information relating to that program.

3.4. In the And instruction the immediate value 2 represents a binary value. In the TestBit instruction the immediate operand specifies one of 32 bits that are identified as bits b_0 to b_{31}. Thus, the immediate value 1 corresponds to b_1.

3.5. Assume that the interface registers for the keyboard of each terminal are as shown in Figure 3.3. A list of device addresses is stored in the memory, starting at DEVICES, where the address given in the list, DEVADRS, is that of KBD_DATA. The pointers to data areas, PNTRn, are also stored in a list, starting at PNTRS.

The required task can be realized using the following CISC-style programs.

POLL:	Move	R2, #40	Use R2 as device counter, i.
LOOP:	Move	R3, DEVICES(R2)	Get address of device i.
	TestBit	4(R3), #1	Test input status of a device.
	Branch$\neq$0	NEXTDEV	Skip read operation if not ready.
	Move	R4, PNTRS(R2)	Get pointer to data for device i.
	MoveByte	(R4)+, (R3)	Get and store input character.
	Move	PNTRS(R2), R4	Update pointer in memory.
NEXTDEV:	Subtract	R2, _4	
	Branch>0	LOOP	
	Return		

INTERRUPT: Same as POLL, except that it returns once a character is read. If several devices are ready at the same time, the routine will be entered several times in succession.

In case (a), POLL must be executed at least 100 times per second. Thus $T_{max} = 10$ ms.

For case (b) have to consider the case when all 20 terminals become ready at the same time. The time required for interrupt servicing must be less than the inter-character delay. That is, $20 \times 200 \times 10^{-9} < 1/c$, or $c < 250,000$ char/s.

The time spent servicing the terminals in each second is given by:

Case *a*: Time $= 100 \times 800 \times 10^{-9}$ ns $= 80$ μs
Case *b*: Time $= 20 \times r \times 200 \times 10^{-9} \times 100 = 400r$ ns

Case *b* is a better strategy for $r < 0.2$.

3.6. Setting the interrupt-enable bit in the PS last ensures that the processor will not be interrupted before it completes the initialization of all interrupts.

3.7. False. All requests indicated in the IPENDING register will be serviced.

3.8. While this could be done, it would unnecessarily place the onus on the programmer and complicate the program.

3.9. Assuming the display interface registers in Figure 3.3, a possible RISC-style program is:

```
        Move              R2, #BINARY       Get the address BINARY.
        LoadHalfword      R2, (R2)          Load the 16-bit pattern.
        Move              R3, #DISP_DATA    Get the address of display.
        Move              R4, #16           Initialize the bit counter.
        Or                R5, R0, #0x8000   Set bit 15 to 1.
LOOP:   And               R6, R2, R5        Test a bit.
        Branch_if_[R6]>[R0] ONE             Branch if the bit is 1.
        Move              R7, #0x30         ASCII code for 0.
        Branch            DSPL
ONE:    Move              R7, #0x31         ASCII code for 1.
DSPL:   Call              DISPLAY
        LShiftR           R5, R5, #1        Shift to check the next bit.
        Subtract          R4, R4, #1        Decrement the bit counter.
        Branch_if_[R4]>[R0] LOOP            Branch back if not finished.
        next instruction

DISPLAY: LoadByte         R8, 4(R3)
        And               R8, R8, #4        Check the DOUT flag.
        Branch_if_[R8]=[R0] DISPLAY
        StoreByte         R7, (R3)          Send the character to display.
        Return
```

3.10. Assuming the display interface registers in Figure 3.3, a possible CISC-style program is:

```
        MoveHalfword      R2, BINARY        Load the 16-bit pattern.
        Move              R3, #15           Initialize the bit counter.
LOOP:   TestBit           R2, R3            Test a bit.
        Branch≠0          ONE               Branch if the bit is 1.
        Move              R4, #0x30         ASCII code for 0.
        Branch            DSPL
ONE:    Move              R4, #0x31         ASCII code for 1.
DSPL:   Call              DISPLAY
        Subtract          R3, #1            Decrement the bit counter.
        Branch≥0          LOOP              Branch back if not finished.
        next instruction

DISPLAY: TestBit          DISP_STATUS, #2   Wait for display to become ready.
        Branch=0          DISPLAY
        MoveByte          DISP_DATA, R4     Send the character to display.
        Return
```

3.11. Since the address 0x10100 exceeds 16 bits, the indexed addressing mode TABLE(R2) cannot be used. The program in Figure 3.18 can be modified as follows:

```
DIGIT    EQU                 0x800               Location of ASCII-encoded digit.
SEVEN    EQU                 0x4030              Address of 7-segment display.
         LoadByte            R2, DIGIT           Load the ASCII-encoded digit.
         And                 R3, R2, #0xF0       Extract high-order bits of ASCII.
         And                 R2, R2, #0x0F       Extract the decimal number.
         Move                R4, #0x30           Check if high-order bits of
         Branch_if_[R3]=[R4] HIGH3                 ASCII code are 0011.
         Move                R2, #0x0F           Not a digit, display a blank.
HIGH3:   OrHigh              R6, R0, #1          Load the value 0x10100
         Or                  R6, R6, #0100         into register R5.
         Or                  R6, R6, R2          Generate the desired address in the table.
         LoadByte            R5, (R6)            Get the 7-segment pattern.
         StoreByte           R5, SEVEN           Display the digit.

         ORIGIN              0x10100
TABLE:   DATABYTE            0x7E,0x30,0x6D,0x79 Table that contains
         DATABYTE            0x33,0x5B,0x5F,0x70   the necessary
         DATABYTE            0x7F,0x7B,0x00,0x00   7-segment patterns.
         DATABYTE            0x00,0x00,0x00,0x00
```

3.12. Since the address 0x10100 exceeds 16 bits, the indexed addressing mode TABLE(R2) cannot be used. The program in Figure 3.19 can be modified as follows:

```
DIGIT    EQU         0x800               Location of ASCII-encoded digit.
SEVEN    EQU         0x4030              Address of 7-segment display.
         MoveByte    R2, DIGIT           Load the ASCII-encoded digit.
         Move        R3, R2
         And         R3, #0xF0           Extract high-order bits of ASCII.
         And         R2, #0x0F           Extract the decimal number.
         CompareByte R3, #0x30           Check if high-order bits of
         Branch=0    HIGH3                 ASCII code are 0011.
         MoveByte    R2, #0x0F           Not a digit, display a blank.
HIGH3:   Move        R4, #0x10100        Base address of the table.
         MoveByte    SEVEN, (R4, R2)     Display the digit.

         ORIGIN      0x10100
TABLE:   DATABYTE    0x7E,0x30,0x6D,0x79 Table that contains
         DATABYTE    0x33,0x5B,0x5F,0x70   the necessary
         DATABYTE    0x7F,0x7B,0x00,0x00   7-segment patterns.
         DATABYTE    0x00,0x00,0x00,0x00
```

3.13. Setting the timer count to indicate one-second periods, and using polling to detect when the timer reaches the end of a period, the following RISC-style program can be used:

```
        Move                R2, #SEVEN
        Move                R3, #TIMER
        OrHigh              R4, R0, #0x5F5      Set the count period in the timer
        Or                  R4, R4, #0xE100       circuit to the value 0x5F5E100,
        Store               R4, 8(R3)             to provide the desired delay.
        Move                R4, #6              Start the timer in the
        StoreByte           R4, 4(R3)             continuous mode.
        Move                R5, R0              Clear the digit counter.
LOOP:   LoadByte            R6, (R3)            Wait for the timer to reach the
        And                 R6, R6, #2            end of the one-second period.
        Branch_if_[R6]=[R0] LOOP
        LoadByte            R7, TABLE(R5)       Look up the 7-segment pattern
        StoreByte           R7, (R2)              and display it.
        Add                 R5, R5, #1          Increment the digit counter,
        Move                R4, #10               and check if >9.
        Branch_if_[R5]<[R4] LOOP
        Move                R5, R0              Clear the digit counter.
        Branch              LOOP

        ORIGIN              0x1000
TABLE:  DATABYTE            0x7E,0x30,0x6D,0x79 Table that contains
        DATABYTE            0x33,0x5B,0x5F,0x70   the necessary
        DATABYTE            0x7F,0x7B,0x00,0x00   7-segment patterns.
        DATABYTE            0x00,0x00,0x00,0x00
```

3.14. Setting the timer count to indicate one-second periods, and using polling to detect when the timer reaches the end of a period, the following CISC-style program can be used:

```
        Move        R3, #TIMER
        Move        8(R3), #0x5F5E100   Set the count period in the timer
                                          to provide the desired delay.
        Move        4(R3), #6           Start the timer in the continuous mode.
        Clear       R5                  Clear the digit counter.
LOOP:   TestBit     (R3), #1            Wait for the timer to reach the
        Branch=0    LOOP                  end of the one-second period.
        MoveByte    SEVEN, TABLE(R5)    Display the digit.
        Add         R5, #1              Increment the digit counter,
        Compare     R5, #10               and check if >9.
        Branch<0    LOOP
        Clear       R5                  Clear the digit counter.
        Branch      LOOP

        ORIGIN      0x1000
TABLE:  DATABYTE    0x7E,0x30,0x6D,0x79 Table that contains
        DATABYTE    0x33,0x5B,0x5F,0x70   the necessary
        DATABYTE    0x7F,0x7B,0x00,0x00   7-segment patterns.
        DATABYTE    0x00,0x00,0x00,0x00
```

3.15. Assume that the two 7-segment displays are concatenated into one device that is accessed by loading the segment patterns into a 16-bit data register, called SEVEN, in the device interface. Then, setting the timer count to indicate one-second periods, and using polling to detect when the timer reaches the end of a period, the following RISC-style program can be used:

```
        Move                R2, #SEVEN
        Move                R3, #TIMER
        OrHigh              R4, R0, #0x5F5      Set the count period in the timer
        Or                  R4, R4, #0xE100       circuit to the value 0x5F5E100,
        Store               R4, 8(R3)             to provide the desired delay.
        Move                R4, #6              Start the timer in the
        StoreByte           R4, 4(R3)             continuous mode.
        Move                R5, R0              Clear low and
        Move                R6, R0                high digit counters.
LOOP:   LoadByte            R7, (R3)            Wait for the timer to reach the
        And                 R7, R7, #2            end of the one-second period.
        Branch_if_[R7]=[R0] LOOP
        LoadByte            R8, TABLE(R5)       Look up the 7-segment
        LoadByte            R9, TABLE(R6)         patterns of digits.
        LShiftL             R9, R9, #8          Concatenate the patterns
        Or                  R8, R8, R9            for the two digits, and send
        StoreHalfword       R8, (R2)              them to 7-segment displays.
        Add                 R5, R5, #1          Increment the low-digit counter,
        Move                R4, #10               and check if >9.
        Branch_if_[R5]<[R4] LOOP
        Move                R5, R0              Clear the low-digit counter.
        Add                 R6, R6, #1          Increment the high-digit counter,
        Branch_if_[R6]<[R4] LOOP                  and check if >9.
        Move                R6, R0              Clear the high-digit counter.
        Branch              LOOP

        ORIGIN              0x1000
TABLE:  DATABYTE            0x7E,0x30,0x6D,0x79 Table that contains
        DATABYTE            0x33,0x5B,0x5F,0x70   the necessary
        DATABYTE            0x7F,0x7B,0x00,0x00   7-segment patterns.
        DATABYTE            0x00,0x00,0x00,0x00
```

3.16. Assume that the two 7-segment displays are concatenated into one device that is accessed by loading the segment patterns into a 16-bit data register, called SEVEN, in the device interface. Then, setting the timer count to indicate one-second periods, and using polling to detect when the timer reaches the end of a period, the following CISC-style program can be used:

```
        Move        R3, #TIMER
        Move        8(R3), #0x5F5E100     Set the count period in the timer
                                            to provide the desired delay.
        Move        4(R3), #6             Start the timer in the continuous mode.
        Clear       R5                    Clear low and
        Clear       R6                      high digit counters.
LOOP:   TestBit     (R3), #1              Wait for the timer to reach the
        Branch=0    LOOP                    end of the one-second period.
        MoveByte    R7, TABLE(R5)         Look up the 7-segment
        MoveByte    R8, TABLE(R6)           patterns of digits.
        LShiftL     R8, #8                Concatenate the patterns
        Or          R7, R8                  for the two digits, and send
        MoveHalfword SEVEN, R7              them to 7-segment displays.
        Add         R5, #1                Increment the low-digit counter,
        Move        R4, #10                 and check if >9.
        Compare     R5, R4
        Branch<0    LOOP
        Clear       R5                    Clear the low-digit counter.
        Add         R6, #1                Increment the high-digit counter,
        Compare     R6, R4                  and check if >9.
        Branch<0    LOOP
        Clear       R6                    Clear the high-digit counter.
        Branch      LOOP

        ORIGIN      0x1000
TABLE:  DATABYTE    0x7E,0x30,0x6D,0x79   Table that contains
        DATABYTE    0x33,0x5B,0x5F,0x70     the necessary
        DATABYTE    0x7F,0x7B,0x00,0x00     7-segment patterns.
        DATABYTE    0x00,0x00,0x00,0x00
```

3.17. Assume that the four 7-segment displays are concatenated into one device that is accessed by loading the segment patterns into a 32-bit data register, called SEVEN, in the device interface. Then, setting the timer count to indicate one-second periods, and using polling to detect when the timer reaches the end of a period, the following RISC-style program can be used:

```
            Move            R2, #SEVEN
            Move            R3, #TIMER
            OrHigh          R4, R0, #0x5F5      Set the count period in the timer
            Or              R4, R4, #0xE100       circuit to the value 0x5F5E100,
            Store           R4, 8(r3)             to provide the desired delay.
            Move            R4, #6              Start the timer in the
            StoreByte       R4, 4(R3)             continuous mode.
            Move            R5, R0              Set the minute counter to 0.
LOOP1:      Move            R6, R0              Clear the seconds counter.
LOOP2:      LoadByte        R7, (R3)            Wait for the timer to reach the
            And             R7, R7, #2            end of one-second period.
            Branch_if_[R7]=[R0] LOOP2
            Add             R6, R6, #1          Wait for 60 seconds before
            Move            R4, #60               updating the display.
            Branch_if_[R6]≤[R4] LOOP2
            Call            DISPLAY
            Add             R5, R5, #1          Increment the minute counter.
            Move            R4, #1440           When the time 24:00 is reached,
            Branch_if_[R5]<[R4] LOOP1             have to reset the minute
            Move            R5, R0                counter to 0.
            Branch          LOOP1

DISPLAY:    Move            R11, R5
            Move            R12, #600           To determine the first digit of
            Divide          R13, R11, R12         hours, divide by 600.
            LoadByte        R15, TABLE(R13)     Get the 7-segment pattern.
            LShiftL         R15, R15, #8        Make space for next digit.
            Multiply        R14, R13, R12       Compute the remainder of the
            Subtract        R11, R11, R14         division operation.
            Move            R12, #60            Divide the remainder by 60 to
            Divide          R13, R11, R12         get the second digit of hours.
            LoadByte        R16, TABLE(R13)     Get the 7-segment pattern,
            Or              R15, R15, R16         concatenate it to the first
            LShiftL         R15, R15, #8          digit, and shift.
            Multiply        R14, R13, R12       Determine the minutes that have
            Subtract        R11, R11, R14         to be displayed.
            Move            R12, #10            To determine the first digit of
            Divide          R13, R11, R12         minutes, divide by 10.
            LoadByte        R16, TABLE(R13)     Get the 7-segment pattern,
            Or              R15, R15, R16         concatenate it to the first
            LShiftL         R15, R15, #8          two digits, and shift.
            Multiply        R14, R13, R12       Compute the remainder, which
            Subtract        R11, R11, R14         is the last digit.
            LoadByte        R16, TABLE(R11)     Concatenate the last digit to
            Or              R15, R15, R16         the preceding 3 digits.
            Store           R15, (R2)           Display the obtained pattern.
            Return

            ORIGIN          0x1000
TABLE:      DATABYTE        0x7E,0x30,0x6D,0x79 Table that contains
            DATABYTE        0x33,0x5B,0x5F,0x70   the necessary
            DATABYTE        0x7F,0x7B             7-segment patterns.
```

3.18. Assume that the four 7-segment displays are concatenated into one device that is accessed by loading the segment patterns into a 32-bit data register, called SEVEN, in the device interface. Then, setting the timer count to indicate one-second periods, and using polling to detect when the timer reaches the end of a period, the following CISC-style program can be used:

```
        Move        R3, #TIMER
        Move        8(R3), #0x5F5E100    Set the count period in the timer
                                           to provide the desired delay.
        Move        4(R3), #6            Start the timer in the continuous mode.
        Clear       R5                   Set the minute counter to 0.
LOOP1:  Move        R6, #60              Counter for a 60-second interval.
LOOP2:  TestBit     (R3), #1             Wait for the timer to reach the
        Branch=0    LOOP2                 end of the one-second period.
        Subtract    R6, #1               Wait for 60 seconds before
        Branch>0    LOOP2                 updating the display.
        Call        DISPLAY
        Add         R5, #1               Increment the minute counter.
        Compare     R5, #1440            When the time 24:00 is reached,
        Branch<0    LOOP1                 have to reset the minute
        Clear       R5                    counter to 0.
        Branch      LOOP1

DISPLAY: Move       R10, R5
        Move        R11, R5              To determine the first digit of
        Divide      R11, #600             hours, divide by 600.
        MoveByte    R12, TABLE(R11)      Get the 7-segment pattern.
        LShiftL     R12, #8              Make space for next digit.
        Multiply    R11, #600            Compute the remainder of the
        Subtract    R10, R11              division operation.
        Move        R11, R10             Divide the remainder by 60 to
        Divide      R11, #60              get the second digit of hours.
        MoveByte    R13, TABLE(R11)      Get the 7-segment pattern,
        Or          R12, R13              concatenate it to the first
        LShiftL     R12, #8               digit, and shift.
        Multiply    R11, #60             Determine the minutes that have
        Subtract    R10, R11              to be displayed.
        Move        R11, R10             To determine the first digit of
        Divide      R11, #10              minutes, divide by 10.
        MoveByte    R13, TABLE(R11)      Get the 7-segment pattern,
        Or          R12, R13              concatenate it to the first
        LShiftL     R12, #8               two digits, and shift.
        Multiply    R11, #10             Compute the remainder, which
        Subtract    R10, R11              is the last digit.
        MoveByte    R13, TABLE(R10)      Concatenate the last digit to
        Or          R12, R13              the preceding 3 digits.
        Move        SEVEN. R12           Display the obtained pattern.
        Return

        ORIGIN      0x1000
TABLE:  DATABYTE    0x7E,0x30,0x6D,0x79  Table that contains
        DATABYTE    0x33,0x5B,0x5F,0x70   the necessary
        DATABYTE    0x7F,0x7B             7-segment patterns.
```

3.19. The program can be based on the scheme presented in Figure 3.4, assuming the device-interface given in Figure 3.3.

We will store the promt "Type your name" as a string of ASCII-encoded characters, by using an assembler directive. Similarly, we will store "Your name reversed" as the output message that precedes the display of the reversed name. Each string is terminated by a Carriage Return character.

A possible RISC-style program is:

```
KBD_DATA    EQU                 0x4000          Specify addresses for keyboard
DISP_DATA   EQU                 0x4010            and display data registers.

            Move                R2, #LOC        Location where line will be stored.
            Move                R3, #KBD_DATA   R3 points to keyboard data register.
            Move                R4, #DISP_DATA  R4 points to display data register.
            Move                R5, #0x0D       ASCII code for Carriage Return.

/* Display the promt. */
            Move                R8, #PROMPT
PLOOP:      LoadByte            R6, 4(R4)       Read display status register.
            And                 R6, R6, #4      Check the DOUT flag.
            Branch_if_[R6]=[R0] PLOOP
            LoadByte            R7, (R8)        Send a character of the prompt
            StoreByte           R7, (R4)          to the display.
            Add                 R8, R8, #1
            Branch_if_[R7]≠[R5] PLOOP

/* Read the name. */
READ:       LoadByte            R6, 4(R3)       Read keyboard status register.
            And                 R6, R6, #2      Check the KIN flag.
            Branch_if_[R6]=[R0] READ
            LoadByte            R7, (R3)        Read character from keyboard.
            StoreByte           R7, (R2)        Write character into main memory
            Add                 R2, R2, #1        and increment the pointer.
ECHO:       LoadByte            R6, 4(R4)       Read display status register.
            And                 R6, R6, #4      Check the DOUT flag.
            Branch_if_[R6]=[R0] ECHO
            StoreByte           R7, (R4)        Send the character to display.
            Branch_if_[R5]≠[R7] READ            Loop back if character is not CR.

/* Display output message. */
            Move                R8, #MESSAGE
MLOOP:      LoadByte            R6, 4(R4)       Read display status register.
            And                 R6, R6, #4      Check the DOUT flag.
            Branch_if_[R6]=[R0] MLOOP
            LoadByte            R7, (R8)        Send a character of the message
            StoreByte           R7, (R4)          to the display.
            Add                 R8, R8, #1
            Branch_if_[R7]≠[R5] MLOOP
```

. . .continued on the next page

```
/* Display the reversed name. */
            Subtract            R8, R2, #2      Set the pointer for backward scan.
            Move                R2, #LOC
NLOOP:      LoadByte            R6, 4(R4)       Read display status register.
            And                 R6, R6, #4      Check the DOUT flag.
            Branch_if_[R6]=[R0] NLOOP
            LoadByte            R7, (R8)
            StoreByte           R7, (R4)        Send a character to display.
            Subtract            R8, R8, #1
            Branch_if_[R8]≥[R2] NLOOP
            next instruction

            ORIGIN              0x1000
PROMPT:     DATABYTE  0x54, 0x79, 0x70, 0x65, 0x20, 0x79, 0x6F, 0x75
            DATABYTE  0x72, 0x20, 0x6E, 0x61, 0x6D, 0x65, 0x0D
MESSAGE:    DATABYTE  0x59, 0x6F, 0x75, 0x72, 0x20, 0x6E, 0x61, 0x6D, 0x65, 0x20
            DATABYTE  0x72, 0x65, 0x76, 0x65, 0x72, 0x73, 0x65, 0x64, 0x0D
```

3.20. The program can be based on the scheme presented in Figure 3.5, assuming the device-interface given in Figure 3.3.

We will store the promt "Type your name" as a string of ASCII-encoded characters, by using an assembler directive. Similarly, we will store "Your name reversed" as the output message that precedes the display of the reversed name. Each string is terminated by a Carriage Return character.

A possible CISC-style program is:

```
KBD_DATA    EQU         0x4000          Specify addresses for keyboard
DISP_DATA   EQU         0x4010            and display data registers.

            Move        R2, #LOC        Location where line will be stored.
            Move        R3, #KBD_DATA   R3 points to keyboard data register.
            Move        R4, #DISP_DATA  R4 points to display data register.
            Move        R5, #0x0D       ASCII code for Carriage Return.

/* Display the promt. */
            Move        R8, #PROMPT
PLOOP:      TestBit     4(R4), #2       Wait for display to become ready.
            Branch=0    PLOOP
            MoveByte    (R4), (R8)      Display a character of the prompt.
            Compare     R5, (R8)+       Check if Carriage Return.
            Branch≠0    PLOOP

/* Read the name. */
READ:       TestBit     4(R3), #1       Wait for a character to be entered.
            Branch=0    READ
            MoveByte    (R2), (R3)      Transfer the character into main memory.
ECHO:       TestBit     4(R4), #2       Wait for display to become ready.
            Branch=0    ECHO
            MoveByte    (R4), (R2)      Send the character to display.
            CompareByte R5, (R2)+       Check if Carriage Return.
            Branch≠0    READ            Loop back if character is not CR.

/* Display output message. */
            Move        R8, #MESSAGE
MLOOP:      TestBit     4(R4), #2       Wait for display to become ready.
            Branch=0    MLOOP
            MoveByte    (R4), (R8)      Display a character of the message
            CompareByte R5, (R8)+       Check if Carriage Return.
            Branch≠0    MLOOP

/* Display the reversed name. */
            Subtract    R2, #1          Set the pointer for backward scan.
NLOOP:      TestBit     4(R4), #2       Wait for display to become ready.
            Branch=0    NLOOP
            MoveByte    (R4), −(R2)     Display a character of the name
            Compare     R2, #LOC        Check if last character.
            Branch>0    NLOOP
            next instruction

            ORIGIN      0x1000
PROMPT:     DATABYTE    0x54, 0x79, 0x70, 0x65, 0x20, 0x79, 0x6F, 0x75
            DATABYTE    0x72, 0x20, 0x6E, 0x61, 0x6D, 0x65, 0x0D
MESSAGE:    DATABYTE    0x59, 0x6F, 0x75, 0x72, 0x20, 0x6E, 0x61, 0x6D, 0x65, 0x20
            DATABYTE    0x72, 0x65, 0x76, 0x65, 0x72, 0x73, 0x65, 0x64, 0x0D
```

3.21. The program can be based on the scheme presented in Figure 3.4, assuming the device-interface given in Figure 3.3.

We will store the promt "Type the word" as a string of ASCII-encoded characters, by using an assembler directive. Similarly, we will store "It is a palindrome" and "It is not a palindrome" as the output messages indicating the result of the test. Each string is terminated by a Carriage Return character.

A possible RISC-style program is:

```
KBD_DATA    EQU                 0x4000          Specify addresses for keyboard
DISP_DATA   EQU                 0x4010            and display data registers.

            Move                R2, #LOC        Location where the word will be stored.
            Move                R3, #KBD_DATA   R3 points to keyboard data register.
            Move                R4, #DISP_DATA  R4 points to display data register.
            Move                R5, #0x0D       ASCII code for Carriage Return.

            Move                R8, #PROMPT
            Call                DISPLAY         Display the prompt.

/* Read the word. */
READ:       LoadByte            R6, 4(R3)       Read keyboard status register.
            And                 R6, R6, #2      Check the KIN flag.
            Branch_if_[R6]=[R0] READ
            LoadByte            R7, (R3)        Read character from keyboard.
            StoreByte           R7, (R2)        Write character into main memory
            Add                 R2, R2, #1        and increment the pointer.
ECHO:       LoadByte            R6, 4(R4)       Read display status register.
            And                 R6, R6, #4      Check the DOUT flag.
            Branch_if_[R6]=[R0] R6, R0, ECHO
            StoreByte           R7, (R4)        Send the character to display.
            Branch_if_[R5]≠[R7] READ            Loop back if character is not CR.
```

. . .continued on the next page

```
/* Determine if palindrome. */
            Subtract                R8, R2, #2          Set the pointer for backward scan.
            Move                    R2, #LOC
            Move                    R9, R2
DLOOP:      LoadByte                R6, (R2)            Scan the word in
            LoadByte                R7, (R8)              opposite directions and check
            Branch_if_[R6]≠[R7]     NOTP                  if there is a match.
            Add                     R2, R2, #1
            Subtract                R8, R8, #1
            Branch_if_[R8]≤[R9]     DLOOP

/* Display the result. */
            Move                    R8, #YESPAL         Display that it is
            Call                    DISPLAY               a palindrome.
            Branch                  DONE
NOTP:       Move                    R8, #NOPAL          Display that it is
            Call                    DISPLAY               not a palindrome.
DONE:       next instruction

DISPLAY:    LoadByte                R6, 4(R4)           Read display status register.
            And                     R6, R6, #4          Check the DOUT flag.
            Branch_if_[R6]=[R0]     R6, R0, DISPLAY
            LoadByte                R7, (R8)            Keep sending characters
            StoreByte               R7, (R4)              to the display until
            Add                     R8, R8, #1            the Carriage Return character
            Branch_if_[R7]≠[R5]     R7, R5, DISPLAY       is reached.
            Return

            ORIGIN                  0x1000
PROMPT:     DATABYTE  0x54, 0x79, 0x70, 0x65, 0x20, 0x74, 0x68, 0x65
            DATABYTE  0x20, 0x77, 0x6F, 0x72, 0x64, 0x0D
YESPAL:     DATABYTE  0x49, 0x74, 0x20, 0x69, 0x73, 0x20, 0x61, 0x20, 0x70, 0x61
            DATABYTE  0x6C, 0x69, 0x6E, 0x64, 0x72, 0x6F, 0x6D, 0x65, 0x0D
NOPAL:      DATABYTE  0x49, 0x74, 0x20, 0x69, 0x73, 0x20, 0x6E, 0x6F, 0x78
            DATABYTE  0x20, 0x61, 0x20, 0x70, 0x61, 0x6C, 0x69, 0x6E, 0x64
            DATABYTE  0x72, 0x6F, 0x6D, 0x65, 0x0D
```

3.22. The program can be based on the scheme presented in Figure 3.5, assuming the device-interface given in Figure 3.3.

We will store the promt "Type the word" as a string of ASCII-encoded characters, by using an assembler directive. Similarly, we will store "It is a palindrome" and "It is not a palindrome" as the output messages indicating the result of the test. Each string is terminated by a Carriage Return character.

A possible CISC-style program is:

```
KBD_DATA    EQU          0x4000          Specify addresses for keyboard
DISP_DATA   EQU          0x4010            and display data registers.

            Move         R2, #LOC        Location where the word will be stored.
            Move         R3, #KBD_DATA   R3 points to keyboard data register.
            Move         R4, #DISP_DATA  R4 points to display data register.
            Move         R5, #0x0D       ASCII code for Carriage Return.

            Move         R8, #PROMPT
            Call         DISPLAY         Display the prompt.

/* Read the word. */
READ:       TestBit      4(R3), #1       Wait for a character to be entered.
            Branch=0     READ
            MoveByte     (R2), (R3)      Transfer the character into main memory.
ECHO:       TestBit      4(R4), #2       Wait for display to become ready.
            Branch=0     ECHO
            MoveByte     (R4), (R2)      Send the character to display.
            CompareByte  R5, (R2)+       Check if Carriage Return.
            Branch≠0     READ            Loop back if character is not CR.

/* Determine if palindrome. */
            Subtract     R2, #1          Set the pointer for backward scan.
            Move         R7, #LOC
DLOOP:      MoveByte     R6, −(R2)       Scan the word in
            CompareByte  R6, (R7)+         opposite directions and check
            Branch≠0     NOTP              if there is a match.
            Compare      R2, #LOC
            Branch≤0     DLOOP
```

. . .continued on the next page

```
/* Display the result. */
            Move        R8, #YESPAL   Display that it is
            Call        DISPLAY         a palindrome.
            Branch      DONE
NOTP:       Move        R8, #NOPAL    Display that it is
            Call        DISPLAY         not a palindrome.
DONE:       next instruction

DISPLAY:    TestBit     4(R4), #2     Wait for display to become ready.
            Branch=0    DISPLAY
            MoveByte    (R4), (R8)    Send the character to display.
            CompareByte R5, (R8)+     Check if Carriage Return.
            Branch≠0    DISPLAY       Loop back if character is not CR.
            Return

            ORIGIN      0x1000
PROMPT:     DATABYTE  0x54, 0x79, 0x70, 0x65, 0x20, 0x74, 0x68, 0x65
            DATABYTE  0x20, 0x77, 0x6F, 0x72, 0x64, 0x0D
YESPAL:     DATABYTE  0x49, 0x74, 0x20, 0x69, 0x73, 0x20, 0x61, 0x20, 0x70, 0x61
            DATABYTE  0x6C, 0x69, 0x6E, 0x64, 0x72, 0x6F, 0x6D, 0x65, 0x0D
NOPAL:      DATABYTE  0x49, 0x74, 0x20, 0x69, 0x73, 0x20, 0x6E, 0x6F, 0x78
            DATABYTE  0x20, 0x61, 0x20, 0x70, 0x61, 0x6C, 0x69, 0x6E, 0x64
            DATABYTE  0x72, 0x6F, 0x6D, 0x65, 0x0D
```

3.23. Assign register R6 to hold the length (up to 78 characters) of the 'sample text' to be displayed. In order to center the display horizontally on an 80-character line, the number of leading space characters needed is given by

$$[80 - ([\mathrm{R6}] + 2)]/2 = 39 - [\mathrm{R6}]/2$$

Assuming the display interface registers in Figure 3.3, a possible RISC-style program is:

	Move	R2, #DISPLAY	Load address of display.
	Move	R5, #STRING	Load address of string.
	Move	R6, #0	Clear length counter.
	Call	LENGTH	String length is now in R6.
	Move	R5, #78	Load 'sample text' length limit.
	Branch_if_[R6]$\leq$[R5]	LEADSP	
	Move	R6, #78	After this point, length of 'sample text' is in R6.
LEADSP:	Move	R5, #39	Load 39.
	AShiftR	R7, R6, #1	Form [R6]/2 in R7.
	Subtract	R7, R5, R7	Number of leading space characters is in R7.
	Move	R10, R6	Load length of 'sample text'.
	Move	R11, R7	Load number of leading space characters.
	Call	DISP13	'+ - - ... - +', followed by carriage return, has been displayed.
	Move	R10, R6	Load length of 'sample text'.
	Move	R11, R7	Load number of leading space characters.
	Move	R3, #0x20	Load ASCII space character.
LOOP1:	Subtract	R11, R11, #1	Decrement space counter.
	Branch_if_[R11]$<$0	DISP2	There may be no spaces.
	Call	DISPLAY	Display space.
	Branch	LOOP1	
DISP2:	Move	R3, #0x7C	Load ASCII '|' character.
	Call	DISPLAY	Display it.
	Move	R5, #STRING	Load address of string.
LOOP2:	LoadByte	R3, (R5)	Load character.
	Call	DISPLAY	Display it.
	Add	R5, R5, #1	Increment string pointer.
	Subtract	R10, R10, #1	Decrement length.
	Branch_if_[R10]$\neq$0	LOOP2	Continue displaying characters if not done.

... continued on the next page

```
            Move            R3, #0x7C       Load ASCII '|' character.
            Call            DISPLAY         Display it.
            Move            R3, #0x0D       Load ASCII carriage return
            Call            DISPLAY          character and display it.
                                             '|sample text|' has now been displayed.
            Move            R10, R6         Load length of 'sample text'.
            Move            R11, R7         Load number of leading space
                                             characters.
            Call            DISP13          The third line '+ - - ... - +', followed by
                                             carriage return, has now been displayed.
            next instruction

LENGTH:     Move            R7, #0x00       Load ASCII 'NUL' character.
LOOP:       LoadByte        R8, (R5)        Load character.
            Branch_if_[R8]=[R7] DONE        Return if 'NUL'.
            Add             R5, R5, #1      Increment string pointer.
            Add             R6, R6, #1      Increment character counter.
            Branch          LOOP            Go back to scan.
DONE:       Return

DISP13:     Move            R3, #0x20       Load ASCII space character.
LOOP3:      Subtract        R11, R11, #1    Decrement space counter.
            Branch_if_[R11]<0 DISP1         There may be no spaces.
            Call            DISPLAY         Display a space.
            Branch          LOOP3           Check to display another space.
DISP1:      Move            R3, #0x2B       Load ASCII '+' character.
            Call            DISPLAY         Display it.
            Move            R3, #0x2D       Load ASCII '-' character.
LOOP4:      Call            DISPLAY         Display it.
            Subtract        R10, R10, #1    Decrement counter.
            Branch_if_[R10]≠0 LOOP4         Go back to display '-'.
            Move            R3, #0x2B       Load ASCII '+' character.
            Call            DISPLAY         Display it.
            Move            R3, #0x0D       Load ASCII carriage return character.
            Call            DISPLAY         Display it. Now '+ - - ... - +'
                                             has been displayed.
            Return

DISPLAY:    LoadBytye       R4, 4(R2)       Wait until DOUT flag is 1.
            And             R4, R4, #4
            Branch_if_[R4]=0 DISPLAY
            Store           R3, (R2)        Display character and return.
            Return
```

3.24. Assign register R6 to hold the length (up to 78 characters) of the 'sample text' to be displayed. In order to center the display horizontally on an 80-character line, the number of leading space characters needed is given by

$$[80 - ([\mathrm{R6}] + 2)]/2 = 39 - [\mathrm{R6}]/2$$

Assuming the display interface registers in Figure 3.3, a possible CISC-style program is:

	Move	R5, #STRING	Load address of string.
	Clear	R6	Clear length counter.
	Call	LENGTH	String length is now in R6.
	Compare	R6, #78	
	Branch≤ 0	LEADSP	
	Move	R6, #78	After this point, length of 'sample text' is in R6.
LEADSP:	Move	R7, #39	Load 39.
	Move	R5, R6	
	AShiftR	R5, #1	Form [R6]/2 in R5.
	Subtract	R7, R5	Number of leading space characters is in R7.
	Move	R10, R6	Load length of 'sample text'.
	Move	R11, R7	Load number of leading space characters.
	Call	DISP13	'+ - - ... - +', followed by carriage return, has been displayed.
	Move	R10, R6	Load length of 'sample text'.
	Move	R11, R7	Load number of leading space characters.
	Move	R3, #0x20	Load ASCII space character.
LOOP1:	Decrement	R11	Decrement space counter.
	Branch< 0	DISP2	There may be no spaces.
	Call	DISPLAY	Display space.
	Branch	LOOP1	
DISP2:	Move	R3, #0x7C	Load ASCII '|' character.
	Call	DISPLAY	Display it.
	Move	R5, #STRING	Load address of string.
LOOP2:	MoveByte	R3, (R5)+	Load character, increment R5.
	Call	DISPLAY	Display it.
	Decrement	R10	Decrement length.
	Branch$\neq 0$	LOOP2	Continue displaying characters if not done.

... continued on the next page

```
            Move        R3, #0x7C       Load ASCII '|' character.
            Call        DISPLAY         Display it.
            Move        R3, #0x0D       Load ASCII carriage return
            Call        DISPLAY          character and display it.
                                         '|sample text|' has now been displayed.
            Move        R10, R6         Load length of 'sample text'.
            Move        R11, R7         Load number of leading space
                                         characters.
            Call        DISP13          The third line '+ - - ... - +', followed by
                                         carriage return, has now been displayed.
            next instruction

LENGTH:     CompareByte (R5)+, #0x00    Check for 'NUL' character.
            Branch=0    DONE            Return if 'NUL'.
            Increment   R6              Increment character counter.
            Branch      LENGTH          Go back to scan.
DONE:       Return

DISP13:     Move        R3, #0x20       Load ASCII space character.
LOOP3:      Decrement   R11             Decrement space counter.
            Branch<0    DISP1           There may be no spaces.
            Call        DISPLAY         Display a space.
            Branch      LOOP3           Check to display another space.
DISP1:      Move        R3, #0x2B       Load ASCII '+' character.
            Call        DISPLAY         Display it.
            Move        R3, #0x2D       Load ASCII '-' character.
LOOP4:      Call        DISPLAY         Display it.
            Decrement   R10             Decrement counter.
            Branch≠0    LOOP4           Go back to display '-'.
            Move        R3, #0x2B       Load ASCII '+' character.
            Call        DISPLAY         Display it.
            Move        R3, #0x0D       Load ASCII carriage return character.
            Call        DISPLAY         Display it. Now '+ - - ... - +'
                                         has been displayed.
            Return

DISPLAY:    TestBit     DISP_STATUS, #2 Wait until DOUT flag is 1.
            Branch=0    DISPLAY
            MoveByte    DISP_DATA, R3   Display character and return.
            Return
```

3.25. Assume that the text to be displayed is stored in the memory at location TEXT, and that a line feed, LF, is automatically sent to the display after a carriage return, CR, has been sent. Also assume that words consist of from 1 to 80 characters, and that the text contains at least one word.

The following RISC program is a possible solution. The program does a complete scan of the text and replaces some of the space characters with carriage return characters to meet the wraparound requirements. During this preprocessing of the text, no lines are sent to the display.

It is assumed that the display cursor has been properly positioned at the left edge of the display. The NUL character is replaced by a CR character when the text is displayed after the preprocessing has been completed.

Registers R2 and R4 are used by the single-character DISPLAY subroutine, and are not used in the main program, so they are not saved/restored in the subroutine. The character to be displayed is passed in register R3. The display interface registers are as shown in Figure 3.3*b*, and the subroutine is patterned after the RISC-style WRITEWAIT routine shown at the end of Section 3.1.2.

```
            Move                  R2, #DISP_DATA   Load address of display.
            Move                  R5, #TEXT        Load address of text.
            Move                  R6, #0x00        Load ASCII NUL, SP,
            Move                  R7, #0x20         and CR characters.
            Move                  R8, #0x0D
            Move                  R9, #81          Load line limit +1.
RESET:      Move                  R10, #0          Clear character counter.
SCAN:       LoadByte              R11, (R5)        Scan text.
            Add                   R5, R5, #1       Increment text pointer.
            Add                   R10, R10, #1     Increment character counter.
            Branch_if_[R11]=[R6]  OUTPUT           If character is NUL,
                                                    display text.
            Branch_if_[R10]<[R9]  SCAN             If not at 81st character,
                                                    continue scanning.
            Branch_if_[R11]≠[R7]  BACKUP           If 81st character
                                                    is not SP, backup to SP.
            StoreByte             R8, −1(R5)       If 81st character is SP,
            Branch                RESET             replace with CR, reset
                                                    counter, continue scanning.
BACKUP:     Subtract              R5, R5, #1       Backup in text
            LoadByte              R11, (R5)         until SP is found.
            Branch_if_[R11]≠[R7]  BACKUP
            StoreByte             R8, (R5)         Then replace SP with CR,
            Add                   R5, R5, #1        move character pointer
            Branch                RESET             forward, resume scanning.
OUTPUT:     Move                  R5, #TEXT        Load address of text.
LOOP:       LoadByte              R3, (R5)         Load character.
            Branch_if_[R3]=[R6]   FINISH           If NUL reached, finish.
            Call                  DISPLAY          Otherwise, display character.
            Add                   R5, R5, #1       Increment character counter
            Branch                LOOP              and continue.
FINISH:     Move                  R3, R8           Load ASCII CR character.
            Call                  DISPLAY          Send to display.
            next instruction

DISPLAY:    LoadByte              R4, 4(R2)        Wait until DOUT flag
            And                   R4, R4, #4        is 1.
            Branch_if_[R4]=0      DISPLAY
            StoreByte             R3, (R2)         Display character and
            Return                                  return.
```

3.26. Assume that the text to be displayed is stored in the memory at location TEXT, and that a line feed, LF, is automatically sent to the display after a carriage return, CR, has been sent. Also assume that words consist of from 1 to 80 characters, and that the text contains at least one word.

The following CISC program is a possible solution. The program does a complete scan of the text and replaces some of the space characters with carriage return characters to meet the wraparound requirements. During this preprocessing of the text, no lines are sent to the display.

It is assumed that the display cursor has been properly positioned at the left edge of the display. The NUL character is replaced by a CR character when the text is displayed after the preprocessing has been completed.

A character to be displayed is passed to the DISPLAY subroutine in register R3. The display interface registers are as shown in Figure 3.3*b*, and the subroutine is patterned after the CISC-style output code in Figure 3.5.

```
            Move            R5, #TEXT           Load address of text.
RESET:      Clear           R10                 Clear character counter.
SCAN:       Increment       R10                 Increment character counter.
            CompareByte     (R5)+, #0x00        Check for NUL character.
            Branch=0        OUTPUT              If character is NUL,
                                                 display text.
            Compare         R10, #81            Check character count.
            Branch<0        SCAN                If not at 81st character,
                                                 continue scanning.
            CompareByte     −1(R5), #0x20       Check for SP.
            Branch≠0        BACKUP              If 81st character
                                                 is not SP, backup to SP.
            MoveByte        −1(r5), #0x0D       If 81st character is SP,
            Branch          RESET                replace with CR, reset
                                                 counter, continue scanning.
BACKUP:     CompareByte     −(R5), #0x20        Backup until SP
            Branch≠0        BACKUP               is found.
            MoveByte        (R5)+, #0x0D        Then replace SP with CR,
            Branch          RESET                reset counter, resume scanning.
OUTPUT:     Move            R5, #TEXT           Load address of text.
LOOP:       MoveByte        R3, (R5)+           Load character.
            Compare         R3, #0x00           Check for NUL.
            Branch=0        FINISH              If NUL reached, finish.
            Call            DISPLAY             Otherwise, display character.
            Branch          LOOP                 and continue.
FINISH:     MoveByte        R3, #0x0D           Load ASCII CR character.
            Call            DISPLAY             Send to display.
            next instruction

DISPLAY:    TestBit         DISP_STATUS, #2     Wait until DOUT flag
            Branch=0        DISPLAY              is 1
            MoveByte        DISP_DATA, R3       Display character and
            Return                               return.
```