Chapter 4
Software

4.1. A program in C for the task described in Problem 3.2 is given below.

```
#define    DISP_DATA       (volatile char *) 0x4010
#define    DISP_STATUS     (volatile char *) 0x4014

#define    LOC             (char *) <arbitrary address>

  /* Look-up table to convert 0-15 numbers to ’0’-’F’ characters. */
  char table[] = {
           0x30,0x31,0x32,0x33,0x34,0x35,0x36,0x37,
           0x38,0x39,0x41,0x42,0x43,0x44,0x45,0x46
  };

  void main()
  {
      char ch, byte, *byte_ptr;
      int i;

      byte_ptr = LOC;                            /* Set starting location for bytes to display. */
      for (i = 0; i < 10; i++) {                 /* Display 10 bytes. */
           byte = *byte_ptr;                     /* Read next byte from memory. */
           ch = table[(byte >> 4) & 0xF];        /* Convert high hex digit to character. */
           while ((*DISP_STATUS & 0x4) == 0);    /* Wait for display to become ready. */
           *DISP_DATA = ch;                      /* Transfer the character to the display. */
           ch = table[byte & 0xF];               /* Convert low hex digit to character. */
           while ((*DISP_STATUS & 0x4) == 0);    /* Wait for display to become ready. */
           *DISP_DATA = ch;                      /* Transfer the character to the display. */
           ch = 0x20;                            /* Put space to separate displayed bytes. */
           while ((*DISP_STATUS & 0x4) == 0);    /* Wait for display to become ready. */
           *DISP_DATA = ch;                      /* Transfer the character to the display. */
           ++byte_ptr;                           /* Advance pointer to next byte. */
      }
  }
```

4.2. A program in C for the task described in Problem 3.9 is given below. If the 16-bit quantity is stored as a 16-bit number, rather than in a 32-bit word, the type declaration can be changed to ’unsigned short’ for the address location in memory and the pointer. The other variables may continue to use the ’unsigned int’ type declaration.

```
#define    DISP_DATA       (volatile char *) 0x4010
#define    DISP_STATUS     (volatile char *) 0x4014

  /* Assume that the 16-bit quantity is stored in a 32-bit word. */
#define    BINARY          (unsigned int *) <arbitrary address>
```

```
void main ()
{
    char ch;
    unsigned int *binary_ptr, binary_value, bit;

    binary_ptr = BINARY;                   /* Set pointer to data to display. */
    binary = *binary_ptr;                  /* Read data. */
    for (i = 15; i >= 0; i−−) {            /* Display 16 bits. */
        bit = binary >> i;                 /* Shift right to isolate next bit. */
        if (bit == 0)                      /* Determine display character based on bit. */
           ch = ’0’;
        else
           ch = ’1’;

        while ((*DISP_STATUS & 0x4) == 0); /* Wait for display to become ready. */
        *DISP_DATA = ch;                   /* Transfer the character to the display. */
    }
}
```

4.3. A program in C for the task described in Problem 3.13 is given below.

```
#define    TIM_STATUS    (volatile char *) 0x4020
#define    TIM_CONT      (volatile char *) 0x4024
#define    TIM_INIT      (volatile int *) 0x4028
#define    SEVEN         (volatile char *) 0x4030

  /* This value assumes a 100-MHz clock for the timer. */
#define    ONE_SECOND_COUNT    0x05F5E100


  /* Table that contains the necessary 7-segment patterns. */
  char table[] = {
          0x7E,0x30,0x6D,0x79,0x33,0x5B,0x5F,0x70,
          0x7F,0x7B,0x00,0x00,0x00,0x00,0x00,0x00
  };

  void main ()
  {
      int digit;
      char pattern;

      *TIM_INIT = ONE_SECOND_COUNT;           /* Set count period for one-second delay. */
      *TIM_CONT = 6;                          /* Set continuous mode. */
      digit = 0;                              /* Set initial digit to display. */
      while (1) {
          while ((*TIM_STATUS & 2) == 0);     /* Wait for end of one-second period. */
          pattern = table[digit];             /* Convert digit to pattern for 7-segment display. */
          *SEVEN = pattern;                   /* Display the pattern for the digit. */
          digit = digit + 1;                  /* Increment the digit counter. */
          if (digit > 9)                      /* Reset counter to zero if necessary. */
             digit = 0;
      }
  }
```

4.4. A program in C for the task described in Problem 3.15 is given below. Two 7-segment displays are assumed to be concatenated into one device for which there is a 16-bit data register in the device interface at the address 0x4030. The `short` data type in C is commonly used for 16-bit integers, hence that type is used in the program below.

```
#define    TIM_STATUS    (volatile char *) 0x4020
#define    TIM_CONT      (volatile char *) 0x4024
#define    TIM_INIT      (volatile int *) 0x4028
#define    SEVEN         (volatile short *) 0x4030

  /* This value assumes a 100-MHz clock for the timer. */
#define    ONE_SECOND_COUNT    0x05F5E100


  /* Table that contains the necessary 7-segment patterns. */
  char table[] = {
           0x7E,0x30,0x6D,0x79,0x33,0x5B,0x5F,0x70,
           0x7F,0x7B,0x00,0x00,0x00,0x00,0x00,0x00
  };

  void main ()
  {
      int digit1, digit0;
      short pattern;

      *TIM_INIT = ONE_SECOND_COUNT;                  /* Set count period for one-second delay. */
      *TIM_CONT = 6;                                 /* Set continuous mode. */
      digit1 = 0;                                    /* Set initial digits to display. */
      digit0 = 0;
      while (1) {
           while ((*TIM_STATUS & 2) == 0);           /* Wait for end of one-second period. */
           pattern = (table[digit1] << 8) | table[digit0];    /* Convert both digits to 7-segment patterns. */
           *SEVEN = pattern;                         /* Display the patterns for the digits. */
           digit0 = digit0 + 1;                      /* Increment the first digit counter. */
           if (digit0 > 9) {
               digit0 = 0;                           /* Reset counter to zero if necessary. */
               digit1 = digit1 + 1;                  /* Increment the second digit counter. */
               if (digit1 > 9) {
                   digit1 = 0;                       /* Reset counter to zero if necessary. */
               }
           }
      }
  }
```

4.5. A program in C for the task described in Problem 3.17 is given below. Four 7-segment displays are assumed to be concatenated into one device for which there is a 32-bit data register in the device interface at the address 0x4030. A separate subroutine is included as a convenience for converting the current values of the counters for the hours and minutes into the appropriate patterns for the 7-segment displays.

```
#define    TIM_STATUS    (volatile char *) 0x4020
#define    TIM_CONT      (volatile char *) 0x4024
#define    TIM_INIT      (volatile int *) 0x4028
#define    SEVEN         (volatile int *) 0x4030

  /* This value assumes a 100-MHz clock for the timer. */
#define    ONE_SECOND_COUNT    0x05F5E100


  /* Table that contains the necessary 7-segment patterns. */
  char table[] = {
            0x7E,0x30,0x6D,0x79,0x33,0x5B,0x5F,0x70,
            0x7F,0x7B,0x00,0x00,0x00,0x00,0x00,0x00
  };

  /* The counters for hours and minutes. */
  int hours, minutes;

  void main ()
  {
      int seconds;

      *TIM_INIT = ONE_SECOND_COUNT;               /* Set count period for one-second delay. */
      *TIM_CONT = 6;                              /* Set continuous mode. */
      seconds = 0;                                /* Initialize time. */
      minutes = 0;
      hours = 0;
      display ();                                 /* Display the initial hours/minutes. */
      while (1) {
          while ((*TIM_STATUS & 2) == 0);         /* Wait for end of one-second period. */
          seconds = seconds + 1;                  /* Increment the seconds counter. */
          if (seconds > 59) {
             seconds = 0;                         /* Reset counter to zero if necessary. */
             minutes = minutes + 1;               /* Increment the minutes counter. */
             if (minutes > 59) {
                minutes = 0;                      /* Reset counter to zero if necessary. */
                hours = hours + 1;                /* Increment the hours counter. */
                if (hours > 23) {
                   hours = 0;                     /* Reset counter to zero if necessary. */
                }
             }
             display ();                          /* Display the current hours/minutes. */
          }
      }
  }
```

```
void display ()
{
    int tens, ones;
    int hours_patterns, minutes_patterns, all_patterns;

    /* Obtain tens/ones digits for hours as quotient/remainder. */
    tens = hours / 10;
    ones = hours − tens * 10;
    hours_patterns = (table[tens] << 8) | table[ones];
    /* Obtain tens/ones digits for minutes as quotient/remainder. */
    tens = minutes / 10;
    ones = minutes − tens * 10;
    minutes_patterns = (table[tens] << 8) | table[ones];
    /* Combine patterns and display all four digits. */
    all_patterns = (hours_patterns << 16) | minutes_patterns;
    *SEVEN = all_patterns;
}
```

4.6. A program in C for the task described in Problem 3.17 is given below, where an interrupt-service routine is associated with the timer. It is assumed that the compiler supports the `interrupt` keyword for generating a function that ends with a return-from-interrupt instruction. In a similar manner as in Problem 4.5, four 7-segment displays are assumed to have a 32-bit data register in the device interface at the address 0x4030.

```
#define   IVECT            (volatile unsigned int *) 0x20
#define   TIM_STATUS       (volatile char *) 0x4020
#define   TIM_CONT         (volatile char *) 0x4024
#define   TIM_INIT         (volatile int *) 0x4028
#define   SEVEN            (volatile int *) 0x4030

  /* This value assumes a 100-MHz clock for the timer. */
#define   ONE_SECOND_COUNT    0x05F5E100


  /* Table that contains the necessary 7-segment patterns. */
  char table[] = {
            0x7E,0x30,0x6D,0x79,0x33,0x5B,0x5F,0x70,
            0x7F,0x7B,0x00,0x00,0x00,0x00,0x00,0x00
  };

  /* The counters for hours, minutes, and seconds. */
  int hours, minutes, seconds;

  interrupt void timer_isr ();                        /* Forward declaration. */

  void main ()
  {
     seconds = 0;                                     /* Initialize time. */
     minutes = 0;
     hours = 0;

     /* Initialize for interrupt-based time-of-day display. */
     *TIM_INIT = ONE_SECOND_COUNT;                    /* Set count period for one-second delay. */
     *TIM_CONT = 7;                                   /* Set continuous mode with interrupts. */
     *IVECT = (unsigned int) &timer_isr;              /* Set interrupt vector. */
     asm (”Subtract    SP, SP, #4”);                  /* Save register R2. */
     asm (”Store    R2, (SP)”);
     asm (”Move    R2, #0x8”);                        /* Allow processor to recognize timer interrupts. */
     asm (”MoveControl    IENABLE, R2”);
     asm (”Move    R2, #0x1”);                        /* Enable interrupts for processor. */
     asm (”MoveControl    PS, R2”);
     asm (”Load    R2, (SP)”);                        /* Restore register R2. */
     asm (”Add    SP, SP, #4”);

     while (1)                                        /* Continuous loop. */
     {
        /* Update time-of-day display every minute using interrupts. */
     }
  }
```

```
interrupt void timer_isr ()
{
    seconds = seconds + 1;          /* Increment the seconds counter. */
    if (seconds > 59) {
        seconds = 0;                /* Reset counter to zero if necessary. */
        minutes = minutes + 1;      /* Increment the minutes counter. */
        if (minutes > 59) {
            minutes = 0;            /* Reset counter to zero if necessary. */
            hours = hours + 1;      /* Increment the hours counter. */
            if (hours > 23) {
                hours = 0;          /* Reset counter to zero if necessary. */
            }
        }
        display ();                 /* Display the current hours/minutes. */
    }
}

void display ()
{
    int tens, ones;
    int hours_patterns, minutes_patterns, all_patterns;

    /* Obtain tens/ones digits for hours as quotient/remainder. */
    tens = hours / 10;
    ones = hours − tens * 10;
    hours_patterns = (table[tens] << 8) | table[ones];
    /* Obtain tens/ones digits for minutes as quotient/remainder. */
    tens = minutes / 10;
    ones = minutes − tens * 10;
    minutes_patterns = (table[tens] << 8) | table[ones];
    /* Combine patterns and display all four digits. */
    all_patterns = (hours_patterns << 16) | minutes_patterns;
    *SEVEN = all_patterns;
}
```

Two variations of the above program include: (a) using a timer count value equivalent to one minute rather than one second, which would allow eliminating the `seconds` counter variable, and (b) having the interrupt-service routine set a software flag once a minute which is checked (and cleared) within the `while` loop of the `main` function to call the `display` function.

4.7 To compute X = X + Y $\times$ Z through explicit use of the MultiplyAccumulate instruction in a C function, additional instructions must be included to read the values of the variables X, Y, and Z into separate registers. Because the registers selected for this purpose may contain valid data due to the compiler-generated instructions in other parts of the program, the register contents should be saved on the stack, and then restored before returning from the function.

For a somewhat simpler solution, assume that the locations in memory for X, Y, and Z can be represented in a 16-bit address.

```
void mult_acc_XYZ(void)
{
    asm("Subtract    SP, SP, #12");    /* Save registers on stack. */
    asm("Store       R8, 0(SP)");
    asm("Store       R7, 4(SP)");
    asm("Store       R6, 8(SP)");
    asm("Load        R8, Z");          /* Load values of variables from memory. */
    asm("Load        R7, Y");
    asm("Load        R6, X");
    asm("MultiplyAccumulate    R6, R7, R8");
    asm("Store       R6, X");          /* Store result in memory. */
    asm("Load        R6, 8(SP)");      /* Restore registers from stack */
    asm("Load        R7, 4(SP)");
    asm("Load        R8, 0(SP)");
    asm("Add         SP, SP, #12");
}
```

If the memory locations for variables X, Y, and Z require 32-bit addresses, then the MoveImmediateAddress instruction can be used with another register, such as R9 (whose contents must also be saved and restored), to generate a full 32-bit address. The three Load instructions above for variables X, Y, and Z must be preceded by a MoveImmediateAddress instructions. The Store instruction for variable X can take advantage of the fact that register R9 was most recently set with the address of X before executing the MultiplyAccumulate instruction.

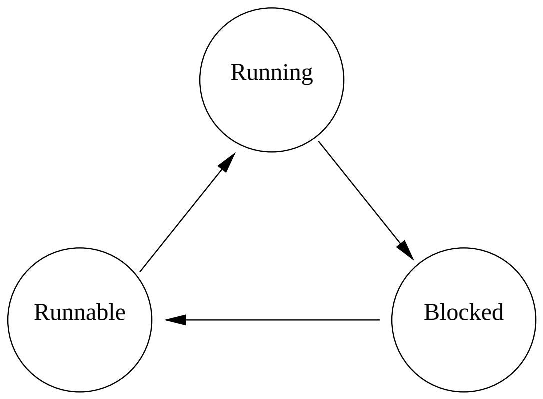
4.8. The time for executing one program consists of $1 + 3 + 1 = 5$ units for the start of execution and for the input phase, $2 + 1 + 3 + 1 + 2 = 9$ units for the compute phase, and $1 + 3 + 1 = 5$ units for the output phase. With no overlap, the total to complete each program in a long sequence is 19 units.

However, the input and output phases for different programs can overlap because they involve different devices. The final 4 units of time for output for program $i - 1$ can overlap with the initial 4 units of time for program i. In a long sequence of programs, the time between completions of successive programs is 15 time units.

The ratio of the best overlapped time to nonoverlapped time is 15/19.

4.9. In the discussion in Section 4.9.2, the overlap was only between input and output activity of two successive jobs or programs. If it is possible to perform output from job $i - 1$, computation for job i, and input to job $i + 1$ at the same time —involving all three units of printer, processor, and disk continuously—then potentially the ratio of overlapped and nonoverlapped times could be reduced closer to 1/3. The OS routines needed to coordinate multiple unit activity cannot be fully overlapped with other activity because those routines use the processor to execute instructions. Therefore, the ratio cannot actually be reduced to 1/3.

4.10. The transitions given in the question are shown below.

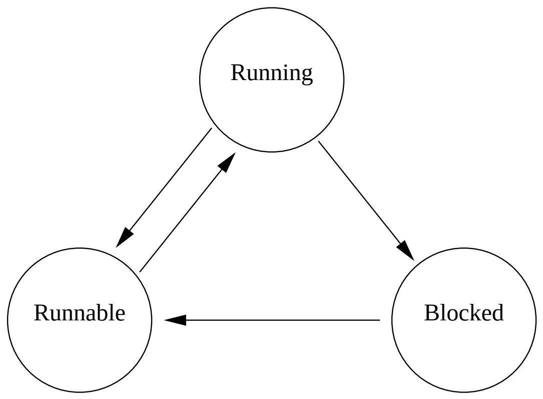


A possible additional transition from Running to Runnable occurs as a result of a timer interrupt at the end of a time slice. This causes the current process to be suspended so that another process can run.

A transition from Runnable to Blocked cannot occur directly. An I/O request must be made while in the Running state to cause a transition to the Blocked state.

A transition from Blocked to Running also cannot occur directly. An I/O request made by a process that is currently blocked eventually completes while that process is still blocked and while some other process is in the Running state. The process whose I/O request has been completed then moves from the Blocked state to the Runnable until it is granted an opportunity to resume execution as a consequence of time slicing.

The one additional transition discussed above is shown in the complete diagram below.



With three states, there can be only six possible transitions. Three valid transitions are described in the question. One valid transition and two invalid transitions are described above. All possible transitions have therefore been considered.