A. Answers to Selected Problems

This file contains answers to some of the problems that appear at the end of the chapters in the text.

A.1 Chapter 1

Chapter 1 contains no problems.

A.2 Chapter 2

1. The program for an intelligent VCR should probably be stored in masked ROM. That form of memory is the cheapest, which will probably be the most important consideration of product of which you hope to sell this many copies.

   A user-configurable name for a network printer should remember even if the power fails probably be stored either in flash memory or in EEROM. Both of these memories will remember the data when the power is turned off. Since is unlikely that the user will change in the name more can be a few dozen times, the limitations on writing to flash memory will probably not be a problem. Particularly if the program for the microprocessor in the printer is in flash memory anyway, flash might be a good choice.

   The program for a beta version of almost anything should probably be either in PROM, EPROM, or flash memory. The likelihood is that you will want to change the program, and these forms of memory make that possible. Most beta tests require few enough units that cost should not be an issue.

   The data that your program just received from the network has to be in RAM. Your program is probably allowed to forget this data if the power fails, and the ability to read and write this data quickly is probably the most important concern.

2. The output for a three-input AND gate is high if the three inputs are high and low otherwise. Therefore the truth table looks like this:

Input 1	Input 2	Input 3	Output
High	High	High	High
High	High	Low	Low
High	Low	High	Low

High	Low	Low	Low
Low	High	High	Low
Low	High	Low	Low
Low	Low	High	Low
Low	Low	Low	Low

3. No answer provided.

4. The engineer would run the HARDWARE_FINISHED/ and SOFTWARE_NOT_WORKING signals through inverters and then route the outputs of the inverters along with the NO_BUGS signal into a three-input NAND gate.

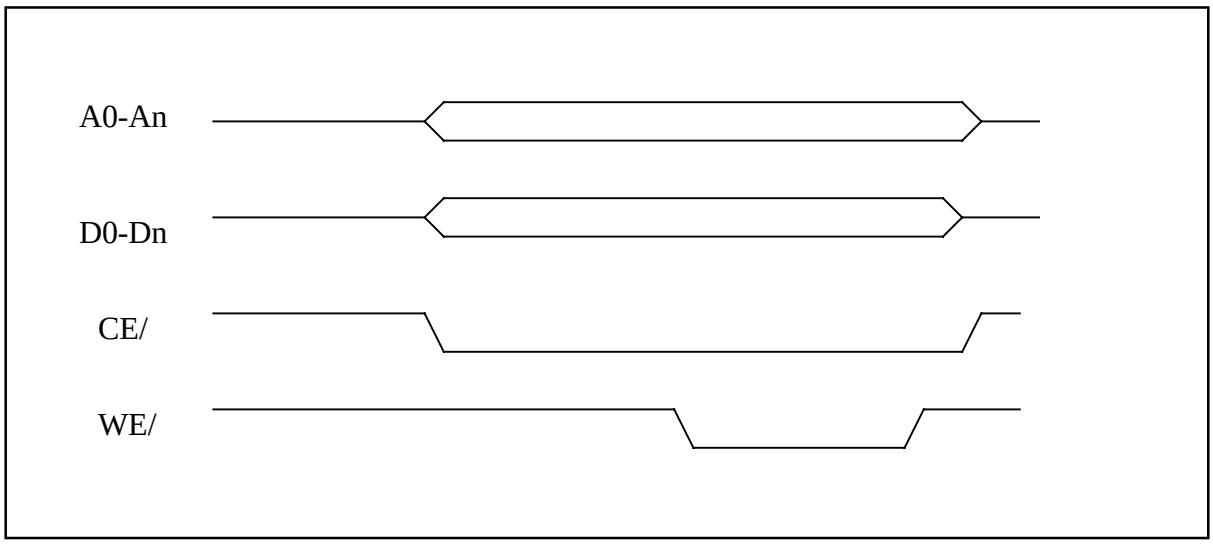
5. No answer provided.

6. This circuit acts as an inverter; its output is always the opposite of its input. The most likely reason that an engineer might do this is that he needs to invert a signal and that his design has a spare NAND gate on it (they are typically packaged in sets of four) and no spare inverter.

7. The problem is that the capacitor is on the wrong side of the switch. As the circuit is drawn, when the switch is first closed, allowing current to flow into the right hand side of the circuit, the current will not only have to power the right-hand side of the circuit but also charge the capacitor. This will cause a momentary brownout for the parts on the left-hand side of the circuit as well. If the capacitor were on the left-hand side of the switch, then when the switch closed, the capacitor would already be charged, and it would be able to give up some of its charge to prevent the brownout.

8. Each inverter in the lower left-hand corner of the circuit only has to drive a portion of the parts in the rest of the circuit; therefore, neither inverter has to drive too much. The advantage of this second method is that both parts of the circuit get the signal at the same time; in the first method, the right hand side of the circuit gets the signal a little bit later, later by the amount of propagation delay in the driver. In especially time-critical circuits, this might make the difference.

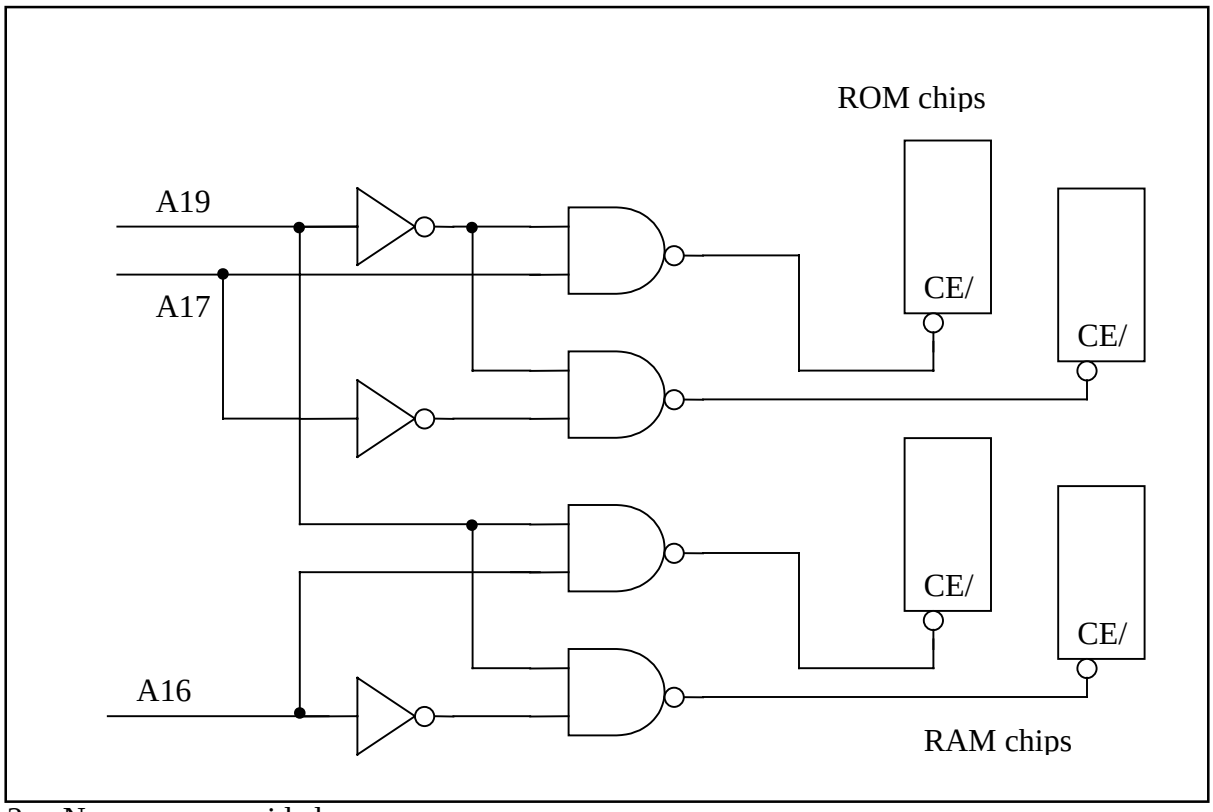
9. The timing diagram for the read cycle for a static RAM is identical to the timing diagram for the read cycle from a ROM. The write cycle is similar, except that the write enable signal will pulse instead of the read enable signal, and the data is likely to appear on the bus at the same time as the address, since the microprocessor will drive both at the same time.



A.3 Chapter 3

1. The first thing to notice in addressing this problem is that the highest-order address line, A19, is low in the addresses that belong to the ROM chips and high in the addresses that belong to the RAM chips. Therefore, one or the other of the ROM chips should be enabled when A19 is low, and one or the other of the RAM chips should be enabled when A19 is high. Since the first of the ROM chips should be enabled as long as the address is less than 0x1ffff, it should therefore be enabled when A17 is low. The second ROM ship should be enabled when A17 is high. Similarly, the first RAM chip should be enabled when A16 is low, and the second should be enabled when A16 is high.

   Assuming that the chip enable lines on both the ROM and the RAM chips are active low, one possible design that implements the comments in the above paragraph is as follows:



2. No answer provided.

3. The most common reason for connecting more than one I/O device to the same interrupt pin on the microprocessor is that you have more I/O devices than interrupt pins. This mechanism allows you to deal with this situation. The disadvantage is that software will have to have some way of figuring out which of the two devices is causing the interrupt or if both are. Since both devices are connected to the same interrupt signal, the microprocessor will go to the same interrupt routine when either of the devices interrupts. This will also mean that the software will respond to these devices a little more slowly, since it will take a little bit of time to figure out which device needs service.

4. The advantage of the edge-triggered interrupt is that the device that is requesting the interrupt need not continue signaling it; the microprocessor will recognize the edge of the signal even if it is just a pulse. The disadvantage is that if the device wants to signal two interrupts, it must ensure that it stops asserting its interrupt signal for a short time between the two so that the second interrupt will cause an edge and be recognized by the microprocessor.

5. No answer provided.

6. You would certainly find a power pin and a ground pin on any UART. Many UARTs also have output pins that allow them to signal a DMA channel to transfer data into or out of the memory. Finally, since timing is very important to UARTs, many of them have several kinds of input and output clock pins.

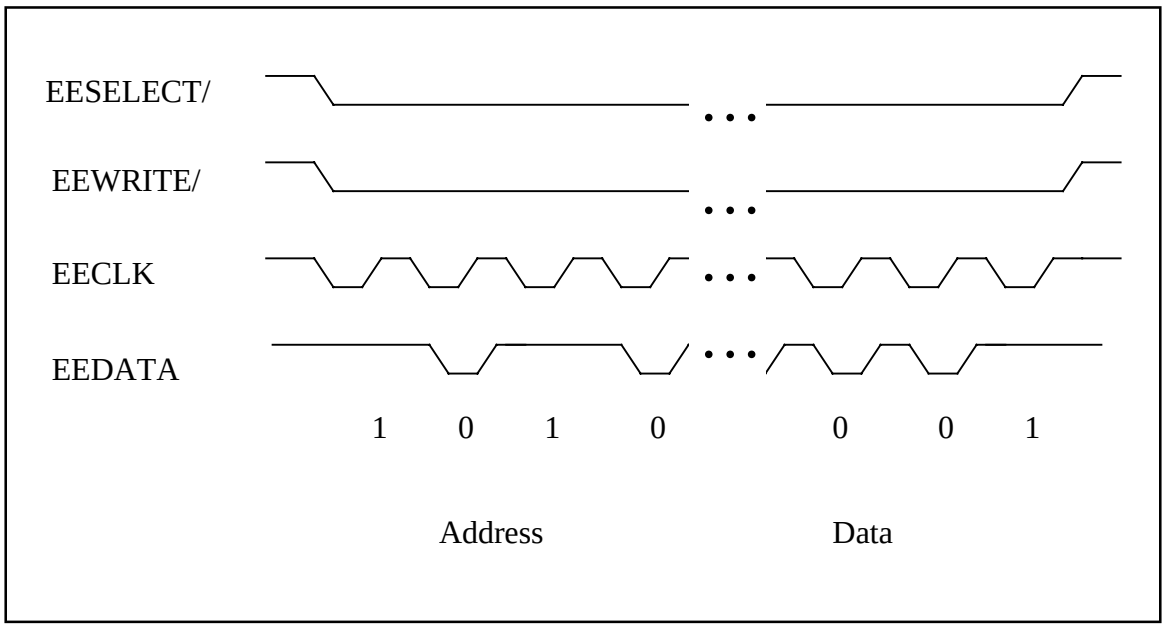
7. A FIFO for received bytes is useful because it allows a certain number of bytes to pile up in the UART while the microprocessor is busy doing other things. A UART that has no FIFO can only remember one received byte at a time; this means that the microprocessor must read each byte from the UART before the next one is received. On a fast serial line, the microprocessor might not be able to keep up if it had to stop what it is doing and execute an interrupt routine each time a single character came in. Even if the microprocessor could keep up, getting into and out of interrupt routines would use up a lot of extra processor cycles. With a FIFO, the microprocessor can wait until several bytes have come in and then transfer them to the memory all at once.

8. There are in the number of ways to do this. One straightforward way to do it is to attach the LED to one of the outputs of a D flip flop. Attach the D input to the D0 data signal, and build some circuitry that pulses the CLK input whenever the microprocessor writes to some particular address. Then the microprocessor can turn the LED on by writing the value 1 to the chosen address and turn it off by writing the value 0 (or vice versa, depending upon exactly how you acquire it up).

   Another popular way to control LEDs is to find some unused output on a UART (for example, RTS on a system that does not use RTS) and attach the LED to that. The microprocessor then instructs the UART whether to signal RTS high or low (typically by writing to a register within the UART) and thereby controls the LED.

9. No answer provided.

10. It is a serial, one-bit-at-a-time device. To write to it, you assert EEENABLE/ and EEWRITE/. Then you put bits on the EEDATA signal, lower the EECLK signal, and raise the EECLK signal. The EEROM will treat the first half dozen bits as an address into which to write the next eight bits as data. The mechanism for reading data is similar, except that you do not assert EEWRITE/and that the EEROM will drive the EEDATA with the data whenever you lower EECLK. A partial timing diagram showing how to write to the EEROM is below, showing the beginning of the address and the end of the data being sent to the EEROM.



11. The microprocessor is U1: note that all of the address and data signals are connected to it, that it has several interrupts signals, INT0, INT1, INT2, and NMI, and that it has a clock circuit attached to pins XTAL and EXTAL. The ROM is U2; the RAM is U3. Note that you can tell them apart, because only the RAM is connected to the write signal.

12. The microprocessor has 20 address signals, A0 through A19, and is therefore capable of addressing 1 megabyte. The ROM has 16 address signals and therefore contains 32 kilobytes. The RAM has 15 address signals and therefore has 16 kilobytes. The microprocessor apparently has a separate I/O address space; it presumably uses the IORQ/ signal to distinguish between memory accesses and I/O accesses.

13. In order to enable the RAM, the CSRAM/ signal must be asserted. This signal is driven by one of the three-input NAND gates in the lower left-hand corner of the schematic, and it will be driven low when the A15 address line is high. (The other two inputs to the NAND gate are IORQ, which will be high whenever the microprocessor is using the memory address space as opposed to the I/O address space, and USRRAM/, which is pulled high by the pullup just to the left of the three-input NAND gate unless the user adds a jumper to ground it.) Therefore, the RAM appears at any address in which A15 is high: 0x08000 to 0x0ffff, 0x18000 to 0x1ffff, 0x28000 to 0x2ffff, and so on. The ROM is enabled whenever the CSROM/ signal is asserted; that signal is also driven by a three-input NAND gate in the lower left-hand corner of the schematic, and it is driven low whenever A15 is low (and therefore the inverted A15 is high). The first-half of the ROM therefore appears at addresses 0x00000 to 0x07fff. The second half of the ROM will appear that addresses 0x10000 to 0x17fff (because the A16 signal is attached to be A15 pin on the ROM).

14. No answer provided.

A.4 Chapter 4

1. No. The problem is that the `vSetTimeZone` function assumes that the `iHours` variable doesn't change while the function is doing its work. If the interrupt routine changes `iHours` during the period of time that interrupts are enabled in the middle of `vSetTimeZone`, that change will get lost when `vSetTimeZone` writes to `iHours` at the end. The local variable `iHoursTemp` does not change the fact that `vSetTimeZone` won't work unless `iHours` doesn't change while the function is calculating.

2. No answer provided.

3. If another, higher-priority interrupt routine uses `lSecondsToday`, then that variable is shared between two interrupt routines. The interrupt routine `updateTime` must disable interrupts (or at least disable the interrupt that corresponds to the interrupt routine that uses `lSecondsToday`) while it is using `lSecondsToday`.

4. The interprocessor interrupt has to wait 250 microseconds while interrupts are disabled; then it takes 100 microseconds to do its work. The networking interrupt becomes irrelevant to the interrupt latency for the interprocessor interrupt.

5. That depends how you define *atomic*. When the `main` routine changes `fTaskCodeUsingTempsB`, that use can certainly be interrupted. However, the interrupt routine only cares whether the value is zero, and the change from zero to nonzero and back again can't be interrupted. As the code is written, it doesn't even matter whether the use of `fTaskCodeUsingTempsB` is atomic; even if it is not, the value of `fTaskCodeUsingTempsB` cannot get corrupted, and it will still protect the temperature arrays properly.

6. No answer provided.

A.5 Chapter 5

1. If all that the system has to do is to control the traffic lights, it is difficult to imagine that a round-robin architecture would not suffice. Controlling traffic lights does not require any complicated calculation, so even a slow microprocessor doing perhaps less than one million instructions per second should be able to get around its loop in a tiny fraction of a second, much faster than any required response to traffic lights. You might even be able to write this system without interrupts, because the microprocessor might be able to get around its loop quickly enough to poll sensors in the street that detect automobiles and buttons for pedestrians. However, to determine this you would need to know something about how the buttons and sensors work and how fast your microprocessor is. One question that you would certainly want to ask

is “What else does this system have to do?” Another question that might be worth asking is how fast the microprocessor will be.

2. You are obviously going to need interrupts, because the system is required to respond to the network interrupt within 200 microseconds, and this system has enough to do that any kind of loop architecture will take longer than that to get around the loop. As to whether or not you need an RTOS for this system, it depends upon other response requirements to various network events. The answer may well be no. However, one characteristic of the Telegraph system that should push you towards using an RTOS is that this system is relatively complicated. At any one time there may be several timeouts being timed, a number of network frames in various stages of processing, and so on. Even if you could write this system without an RTOS, which, incidentally, it turns out you probably can, you might want to include an RTOS just to simplify your software structure.

3. This code is going to behave most like a function-queue architecture in which functions are assigned priorities and are called in priority order rather than in the order that they were placed on the queue. At the top of the loop in main, the logic simply determines the highest-priority signal that has been set and then calls that function.

4. No answer provided.

5. No answer provided.

A.6 Chapter 6

1. This function is not reentrant (at least on most processors), because `cErrors` will be shared by any task that calls `vCountErrors`, and the use of `cErrors` is not atomic.

2. It is not possible to tell whether this function is reentrant. When it uses the value pointed to by `p_sz`, it may be accessing shared data. There is no way to tell this without seeing the code that calls `strlen`. In theory, it is possible that there is a problem. On the other hand, it is much more likely that different tasks will always pass `p_sz` pointing to different strings, or that the strings passed to `strlen` are never changed while `strlen` is working on them. In these cases `strlen` will work.

3. No answer provided.

4. There are two problems with the code. First, both functions must use the same semaphore. Second, both functions must both take and release the semaphore. The changes are shown below:

```
static int iRecordCount;

void increment_records (int iCount)
{
```

```
   OSSemGet (SEMAPHORE_PLUSSEMAPHORE_COUNT);    /* Use one sem */
   iRecordCount += iCount;
   OSSemGive (SEMAPHORE_COUNT);                 /* Add this */
}

void decrement_records (int iCount)
{
   OSSemGet (SEMAPHORE_COUNT);                  /* Add this */
   iRecordCount -= iCount;
   OSSemGive (SEMAPHORE_MINUSSEMAPHORE_COUNT);  /* Use one sem */
}
```

5. The answer is as shown below. Note that all of the time that `iFixValue` uses `iTemp` as a temporary value to modify `iValue` (which is shared by all of the tasks that call `iFixValue`), it must be protected by a semaphore.

```
static int iValue;

int iFixValue (int iParm)
{
   int iTemp;

   OSSemGet (SEMAPHORE);        /* Get it here */

   iTemp = iValue;
   iTemp += iParm * 17;
   if (iTemp > 4922)
      iTemp = iParm;
   iValue = iTemp;

   OSSemGive (SEMAPHORE);       /* Release it here. */

   iParm = iTemp + 179;
   if (iParm < 2000)
      return 1;
   else
      return 0;
}
```

6. Tasks M and N should probably use a semaphore. Since each of them must update many elements in the array, disabling interrupts may keep interrupts disabled for too long a period of time (obviously depending upon how fast your interrupt routines need to respond). Task P, on the other hand, must disable interrupts, since there is no other mechanism to deal with data that it shared with an interrupt routine.

7. The code works, because although the task and the interrupt routine do share `iLinesPrinted` and `iLinesTotal`, the task only uses them when no print job is printing. Therefore, the interrupt routine will never execute while the task is using `iLinesPrinted` or `iLinesTotal`. The interrupt routine only happens after the task code has called `vHardwarePrinterOutputLine`, and after the task code has made

that call, it blocks on the semaphore until the interrupt routine frees the semaphore, indicating that no more interrupts will occur.

8. No answer provided.

9. It is most like a function-queue architecture in which the system always executes the highest-priority function that is on the queue, regardless of the order in which functions were placed on the queue.

10. This statement is true. The shared data problem arises when the CPU is switched away from one task while that task was using the shared data. Since this does not happen in a nonpreemptive RTOS, there is no shared data problem among tasks. (Of course, if there is data shared between tasks and interrupt routines, there is still a potential problem with that.)

A.7 Chapter 7

1. The problem is that `vGetKey` passes the *address* (not the value) of `ch` to `vHandleKeyCommandsTask` through the queue. That address is a location on the stack, and the contents of that location will potentially become garbage as soon as `vGetKey` returns. There is no assurance that `vHandleKeyCommandsTask` will read the character before it turns into garbage. The correct way to write this code is to cast the *value* of `ch` to a void pointer when it is sent, and cast the value that is returned from `rcvmsg` back to a char:

```
sndmsg (KEY_MBOX, (void *) ch, PRIORITY_NORMAL);

ch = (char) rcvmsg (KEY_MBOX, WAIT_FOREVER);
```

2. Here is the code for Figure 7-20. The code for Figure 7-8 is left to you.

```
/* Handle for semaphore. */
SEMID semaphore;

void main
{
   . . .
   /* Create a semaphore already taken */
   OSSemCreate (&semaphore, TAKEN);
   . . .
}

void interrupt vTriggerISR (void)
{
   /* The user pulled the trigger.  Release the semaphore. */
   OSSemPost (&semaphore);
}

void vScanTask (void)
{
```

```
        . . .
        while (TRUE)
        {
           /* Wait for the user to pull the trigger. */
           OSSemPend (&semaphore);

           !! Turn on the scanner hardware and look for a scan.
        }

    }
```

3. The problem with the code is that `vUseCharactersTask` blocks in two places: waiting for the urgent queue and waiting for the regular queue. If `vUseCharactersTask` is blocked on the regular queue and `vGetCharactersTask` puts a message on the urgent queue, `vUseCharactersTask` will still be blocked on the regular queue and will never get around to reading the urgent queue until it is sent a command on the regular queue. There are a number of solutions to this. The most obvious is to have two tasks, one that handles the regular commands and one that handles the urgent commands.

4. No answer provided.

5. Although the suggested solution of using several memory buffer pools uses memory more efficiently than using just one pool, it is still not as effective as `malloc` and `free`. Any task that needs memory will have to select from one of a set of pre-determined sizes, and some tasks will no doubt waste some of the memory that they have allocated. Further, you will have to decide ahead of time how many of each size of buffer your system will have. If the task goes after a buffer of a certain size, and the system has run out of buffers of that size, then the memory allocation will fail, even if there are plenty of other buffers of different sizes.

6. No answer provided.

7. This code is no better than the code in the previous problem. The semaphores have no effect on the underlying difficulty.

8. No answer provided.

9. The primary advantage of a nonconforming interrupt routine is that it can be faster, since it avoids the overhead associated with letting the RTOS know when it enters and exits. The disadvantages are that the nonconforming interrupt routines may not call any of the RTOS functions and that they must not allow themselves to be interrupted by any higher-priority interrupt routine that might call the RTOS.

A.8 Chapter 8

1. The `vInterpretCommandTask` task must have some way of knowing when each command is over. The easiest way is probably to have `vGetCommandCharacter` write the carriage returns into the buffer.

2. No answer provided.

A.9 Chapter 9

Chapter 9 contains no problems.

A.10 Chapter 10

1. The first thing to try might be to connect to the chip enable pin on the UART and see if it ever goes low, enabling the microprocessor to access the UART. If not, that would indicate that the hardware is not working or that your software doesn't know the address range at which the UART appears in the address space. If the chip enable pin does go low, then the next thing to try might be to connect to the write (WR) pin on the UART and see if it ever goes low at the same time as the chip enable pin. You might be able to manage this with an oscilloscope, triggering it on the falling edge of the chip enable signal, but it will be easier to do with a logic analyzer (also triggering it on the falling edge of the chip enable signal). If the chip enable signal and the write signal both are low at the same time, then the next thing to do would be to hook up the eight data signals and the two address signals into the UART—now you will definitely need a logic analyzer—and capture the data that you are writing to the UART. You can use your logic analyzer in state mode and clock it on the rising edge of the write signal.

2. It would seem in this case that the interrupt signal from the UART is probably worth testing. If it is never asserted, it indicates that you have not set up the UART correctly to interrupt. If it is asserted, then the likely problem is that your software has not set the microprocessor up to jump to your interrupt routine.

A.11 Chapter 11

No answers provided for the problems in Chapter 11.