

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.