In a simple design, the software definition, like the hardware definition, may describe the software for a single board. In a more complex design, where different software engineers work on different parts of the code for a single board, there may be a software definition for each individual engineer's code. In a complex multiprocessor system, there may be an overall software document, which I consider to be part of the system engineering specification. The software specifications should include the following:

- A statement of the requirements, including the requirements definition, engineering specifications, and hardware definition, as appropriate.
- The communication protocol to any other software, whether to another processor or to another piece of the software for this processor. This should include descriptions of buffer interface mechanisms, command/response protocols, semaphore definitions, and, in short, anything to which the complementing code needs to talk.
- A description of how the design was implemented, using flowcharts, pseudocode, or other methods. (Chapter 3 describes these in more detail.)

Because software can be broken down more flexibly than hardware, it is difficult to pin down a single software definition format that works for everybody all the time. The key is to define any interfaces that other engineers need to know about and identify the design details that engineers in the future might need to know.

This discussion assumes that the hardware and software are fairly independent. In a simple system like the pool timer, that is a good model. The hardware is designed, the software is written around that hardware, and that is that. While the actual design implementations may proceed in parallel, the software engineer basically writes code around the available hardware. In a more complex system, the process may be iterative. For example, the software and hardware engineers may have a meeting at which they jointly decide what hardware is required to perform the function. Large amounts of memory may be required for data buffers, or the software group may request a specific peripheral IC because an interface library already has been developed for it. There are tradeoffs in this game between ease of software development and cost or complexity of hardware.

Hardware/Software Partitioning

Once, while having lunch with a group of engineers, I jokingly made the statement that my design philosophy was to put everything under software control. That way, bugs in the design were by definition the fault of the software engineer.

This flippant conversation touches on a real problem in any embedded system: Which functions should be performed in hardware and which should be performed in software? An example of this can be found in the pool timer. As we will see in the next chapter, the pool timer displays time information on four seven-segment LED displays. There are display decoder ICs that accept a four-bit input and produce the signals necessary to drive the display. This design takes a different approach and drives the display segments directly from a register, which is under software control. When the software wants to display a number, it must convert the number to the seven-segment pattern and write that pattern to a register. The savings was a single IC in the design. This approach also allows the code to display nonnumeric symbols on the display (A, C, H, J, L, P, U), which I used for debugging the system.

While this decision saved an IC, it had three costs: ROM space was needed for the lookup table, extra code had to be included for the hex to seven segment conversion, and the software needed extra time to perform the translation. Given the simplicity of this function, none of these was a serious problem. The table was 16 bytes long, so the code took a few more bytes and needed only a few microseconds to execute. But the principle described is at the heart of the software/hardware tradeoff: The more functions that can be pushed into software, the lower will be the product cost, up to the point where a faster processor or more memory is required to implement the added functionality. The pool timer demonstrates another example of this kind of tradeoff, which we'll discuss in Chapter 4.

As the saying goes, there is no such thing as a free lunch. Pushing functionality into the software increases software complexity, development time, and debug time. This is an NRE, just like the mask ROM charges described earlier. However, given the increasing speed and power of microprocessors, I expect to see an ever-increasing trend toward including as much functionality in the software as possible.

In a more complex system, these tradeoffs can create heated discussion. Should the software handle regular timer interrupts at a high rate and count them to time low-rate events, or should an external timer be added that can be programmed to interrupt the software when it times out? Should the software drive the stepper motor directly, or should an external stepper controller be used? If the software drives the motor, should protection logic be included to prevent damage to the motor drive transistors if the software turns on the wrong pair? And if the processor runs out of throughput halfway though the project, did the design place too much of a burden on the software, or did the software engineer write inefficient code? The answers to these questions depend on your design. If you stay in this field very long, be prepared to get into one of these discussions.

While doing everything in software increases development costs, moving functionality to the hardware increases product cost, and these costs are incurred with every unit built. In a low-cost design, addition of any extra hardware can have a significant effect on product cost, so the software/hardware tradeoff can be extremely important. In an extremely cost-sensitive design, such as a low-cost