not as important as knowing the upper limit on code size. One rule of thumb is that if the ROM is more than 80 percent full, it is too full. Unless you can guarantee that the system requirements will never change, leave some margin. In many cases, it is worthwhile to write portions of the code just to see how big they will get. In microcontroller-based systems with internal ROM, you are limited to whatever program memory the part contains.

Like RAM usage, code size depends somewhat on the development (programming) language selected. A program written in assembler takes less space than one written in Pascal, for example. Again, this depends on the language and even on the specific brand of software.

It is not a good idea to let the language drive the design, at least in low-cost systems. The languages easiest to use, debug, and maintain are often those that require the most memory and processing speed. Choosing the wrong language can turn a simple, inexpensive, single-chip design into something that requires an embedded 64-bit powerhouse with megabytes of RAM. However, sometimes company policy or a customer contract specifies the use of a high-level language. In these cases, you just have to live with the increased cost and complexity that this implies.

A real-life example will illustrate the potential problems you can run into here. An embedded system was to be controlled by an x86-family processor. We had settled on an off-the-shelf CPU board, based on a 386SX. Then one of the software people noticed that the 386SX has no floating-point coprocessor (FPU). The software engineers were from the PC world, where everything ran in Windows 95/98, on a 400 MHz Pentium. They couldn't conceive of not having hardware for floating-point calculations. The only way to get a hardware floating point was to go up to a 486DX or Pentium processor, which doubled the cost of the CPU board. This was an *embedded* application, with no keyboard, display, or hard drive attached. The CPU was reading sensors, controlling motors, and communicating with a PC host. There was no reason to believe that floating-point calculations would ever be needed. But, because C makes it easy to define floating-point variables, they were expected to be available in hardware. In fact, the code wasn't designed or written yet, so we didn't *know* whether any floating-point calculations would actually be required.

This same design had some embedded microcontrollers for very low-level functions. What if a software engineer had decided that those needed hardware floating point and a deep stack for recursion? We'd have turned a requirement for a cheap 8-bit microcontroller into Pentium-class overkill.

Number of Interrupts Required

We'll cover this subject in more detail in Chapter 5; however, a few comments are worth mentioning here.

Many designers overuse interrupts. An interrupt does just that—it interrupts program execution. Interrupts are best used for those things that cannot wait for the processor to get to them. In some cases, an interrupt can be used just to reduce the hardware complexity (and the associated costs), but almost always it is at the expense of increased debug time and higher potential for hard-to-find intermittent errors. In those cases where interrupts are required, it is important to know how many really are needed. Interrupts are used to notify the processor of special events such as a timer that timed out or a piece of hardware that needs attention. Counting the events that need interrupts is straightforward, but be sure to take into account internal interrupt sources as well. Some tricks can be played to reduce the number of interrupt *signals* required when there are more interrupt *sources* than the processor has interrupt inputs. Again, we'll discuss these in Chapter 5.

Real-Time Considerations

This subject covers a lot of territory and is closely connected to the issue of processing speed. Real-time events are what embedded microprocessors generally are intended to handle. However, some specific events deserve special consideration. For example, you might have a subsystem that controls a motor using pulse-width modulation. In this scheme, the motor current is controlled by switching the current at a very high rate and using the duty cycle to control the motor speed. The motor, being a relatively slow mechanical device, responds to the time-average of the current (see Figure 1.2). Lower-duty cycles result in lower average current and slower rotation. (This is a very high-level description; entire books have been written about PWM and motor control. Read one of those for all the details.)

In our hypothetical motor-control system, say that the microprocessor cannot keep up with the motor on a real-time basis. That is, the *chopping* rate, the rate at which the motor current is switched on and off, is faster than the microprocessor can handle. But the other required tasks, such as communicating with whatever is controlling the motor-processor subsystem, are no problem for our processor. It

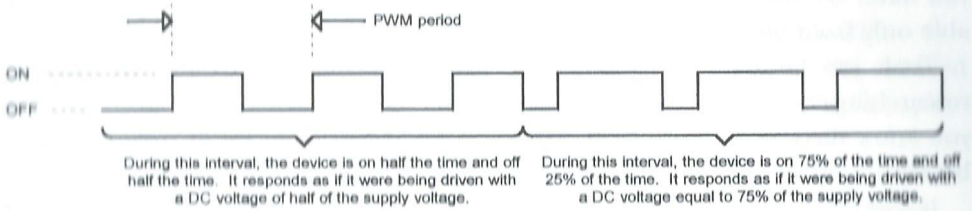

Figure 1.2
PWM Operation.