CHAPTER 12

Digital Electronics

Before beginning, we must warn you that there is a lot of information in this chapter, and it may be difficult to absorb all this at once. Some information is present largely for historical interest and to provide a better understanding of how complex digital systems such as microcontrollers work. Our advice is to skim to your heart's content, and pull out whatever information you find practical. The basic principles are still the same, but if you find that your design uses more than three ICs, you probably could be using a microcontroller (the subject of Chap. 13).

12.1 The Basics of Digital Electronics

Until now, we have mainly covered the analog realm of electronics—circuits that accept and respond to voltages that vary continuously over a given range. Such analog circuits included rectifiers, filters, amplifiers, simple RC timers, oscillators, simple transistor switches, and so on. Although each of these analog circuits is fundamentally important in its own right, these circuits lack an important feature: they cannot store and process bits of information needed to make complex logical decisions. To incorporate logical decision-making processes into a circuit, you need to use digital electronics.

This chapter is concerned with laying the foundations of digital electronics. The actual implementation of digital electronics these days is either handled by microcontrollers (see Chap. 13) or programmable logic devices (see Chap. 14).

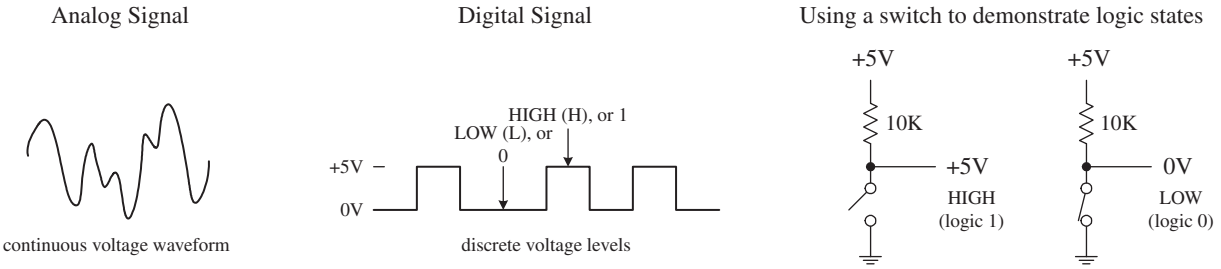


FIGURE 12.1

12.1.1 Digital Logic States

In digital electronics, there are only two voltage states present at any point within a circuit. These voltage states are either *high* or *low*. The voltage being high or low at a particular location within a circuit can signify a number of things. For example, it may represent the on or off state of a switch or saturated transistor, one bit of a number, whether an event has occurred, or whether some action should be taken.

The high and low states can be represented as true and false statements, which are used in Boolean logic. In most cases, high equals true and low equals false. However, this does not need to be the case—you could make high equal to false and low equal to true. The decision to use one convention over the other is a matter left ultimately to the designer. In digital lingo, to avoid people getting confused over which convention is in use, the term *positive true logic* is used when high equals true, while the term *negative true logic* is used when high equals false.

In Boolean logic, the symbols 1 and 0 are used to represent true and false, respectively. Now, unfortunately, 1 and 0 are also used in electronics to represent high and low voltage states, where high equals 1 and low equals 0. As you can see, things can get a bit confusing, especially if you are not sure which type of logic convention is being used: positive true or negative true logic. In Sec. 12.3, you will see some examples that deal with this confusing issue.

The exact voltages assigned to high or low voltage states depend on the specific logic IC that is used (as it turns out, digital components are IC-based). As a general rule of thumb, +5 V is considered high, while 0 V (ground) is considered low. However, as you will see in Sec. 12.4, this does not need to be the case. For example, some logic ICs may interpret a voltage from +2.4 to +5 V as high and a voltage from +0.8 to 0 V as low. Other ICs may use an entirely different range.

12.1.2 Number Codes Used in Digital Electronics

Binary

Because digital circuits work with only two voltage states, it is logical to use the binary number system to keep track of information. A binary number is composed of two binary digits, 0 and 1, which are also called *bits* (for example, 0 = low voltage and 1 = high voltage). By contrast, a decimal number such as 736 is represented by successive powers of 10:

$$736_{10} = 7 \times 10^2 + 3 \times 10^1 + 6 \times 10^0$$

Similarly, a binary number such as 11100 (28_{10}) can be expressed as successive powers of 2:

$$11100_2 = 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$$

The subscript tells which number system is in use (X_{10} = decimal number and X_2 = binary number). The highest-order bit (leftmost bit) is called the *most significant bit* (MSB), while the lowest-order bit (rightmost bit) is called the *least significant bit* (LSB). Methods used to convert from decimal to binary and vice versa are shown in Fig. 12.2.

It should be noted that most digital systems deal with 4, 8, 16, or 32 bits at a time. The decimal-to-binary conversion example given here has a 7-bit answer. In an 8-bit system, you would need to put an additional 0 in front of the MSB (for example, 01101101). In a 16-bit system, nine additional 0s would need to be added (for example, 0000000001101101).

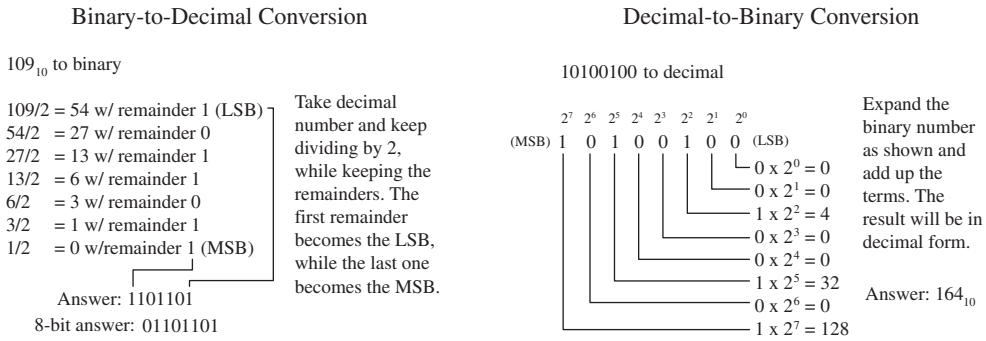


FIGURE 12.2

As a practical note, the easiest way to convert a number from one base to another is to use a calculator. For example, to convert a decimal number into a binary number, type in the decimal number (in base 10 mode) and then change to binary mode (which usually entails a second function key). The number will now be in binary (1s and 0s). To convert a binary number to a decimal number, start out in binary mode, type in the number, and then switch to decimal mode.

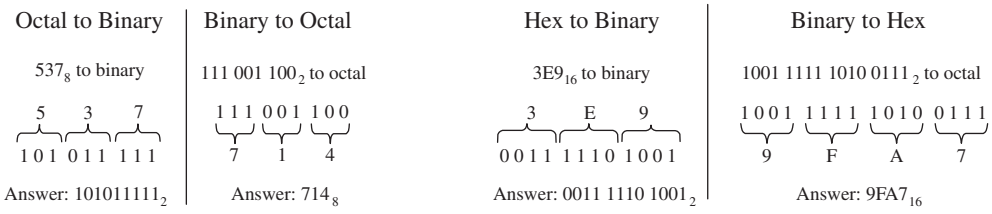
Octal and Hexadecimal

Two other number systems used in digital electronics include the octal and hexadecimal systems. In the octal system (base 8), there are 8 allowable digits: 0, 1, 2, 3, 4, 5, 6, and 7. In the hexadecimal system (base 16), there are 16 allowable digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. Here are examples of octal and hexadecimal numbers with decimal equivalents:

$$247_8 \text{ (octal)} = 2 \times 8^2 + 4 \times 8^1 + 7 \times 8^0 = 167_{10} \text{ (decimal)}$$
$$2D5_{16} \text{ (hex)} = 2 \times 16^2 + D\ (=13_{10}) \times 16^1 + 9 \times 16^0 = 725_{10} \text{ (decimal)}$$

Of course, binary numbers are the natural choice for digital systems, but since these binary numbers can become long and difficult to interpret by our decimal-based brains (a result of our ten fingers), it is common to write them out in hexadecimal or octal form.

Unlike decimal numbers, octal and hexadecimal numbers can be translated easily to and from binary. This is because a binary number, no matter how long, can be broken up into 3-bit groupings (for octal) or 4-bit groupings (for hexadecimal). You simply add zero to the beginning of the binary number if the total numbers of bits is not divisible by 3 or 4. Figure 12.3 should paint the picture better than words.



A 3-digit binary number is replaced for each octal digit, and vice versa. The 3-digit terms are then grouped (or octal terms are grouped).

A 4-digit binary number is replaced for each hex digit, and vice versa. The 4-digit terms are then grouped (or hex terms are grouped).

FIGURE 12.3

Today, the hexadecimal system has essentially replaced the octal system. The octal system was popular at one time, when microprocessor systems used 12-bit and 36-bit words, along with a 6-bit alphanumeric code, which are all divisible by 3-bit units (1 octal digit). Today, microprocessor systems mainly work with 8-bit, 16-bit, 20-bit, 32-bit, or 64-bit words, which are all divisible by 4-bit units (1 hex digit). In other words, an 8-bit word can be broken down into 2 hex digits, a 16-bit word into 4 hex digits, a 20-bit word into 5 hex digits, and so on.

Hexadecimal representation of binary numbers pops up in many memory and microprocessor applications that use programming codes (for example, within assembly language) to address memory locations and initiate other specialized tasks that would otherwise require typing in long binary numbers. For example, a 20-bit address code used to identify one of a million memory locations can be replaced with a hexadecimal code (in the assembly program) that reduces the count to five hex digits. Note that a compiler program later converts the hex numbers within the assembly language program into binary numbers (machine code), which the microprocessor can use. Table 12.1 shows a conversion table.

TABLE 12.1 Decimal, Binary, Octal, Hex, BCD Conversion Table

DECIMAL	BINARY	OCTAL	HEXADECIMAL	BCD
00	0000 0000	00	00	0000 0000
01	0000 0001	01	01	0000 0001
02	0000 0010	02	02	0000 0010
03	0000 0011	03	03	0000 0011
04	0000 0100	04	04	0000 0100
05	0000 0101	05	05	0000 0101
06	0000 0110	06	06	0000 0110
07	0000 0111	07	07	0000 0111
08	0000 1000	10	08	0000 1000
09	0000 1001	11	09	0000 1001
10	0000 1010	12	0A	0001 0000
11	0000 1011	13	0B	0001 0001
12	0000 1100	14	0C	0001 0010
13	0000 1101	15	0D	0001 0011
14	0000 1110	16	0E	0001 0100
15	0000 1111	17	0F	0001 0101
16	0001 0000	20	10	0001 0110
17	0001 0001	21	11	0001 0111
18	0001 0010	22	12	0001 1000
19	0001 0011	23	13	0001 1001
20	0001 0100	24	14	0010 0000