It has been said in the past that a “smart” device was one that was capable of some level of computation. That definition is dated. Raj Jain, one of the early developers of congestion control protocols for networks, describes a “smart” device as one that is connected. Connected to other devices, connected to the Internet, connected to the world. In this chapter, we will investigate techniques for communicating between computers, both microcontrollers and desktop computers. We’ll address byte stream concepts, as well as communication protocols that enable higher-level abstract communications.

One of the fundamental notions that has to be considered when dealing with computer to computer communications is that each computer (whether it be a simple microcontroller or a sophisticated desktop or server machine) is executing its own program, and both programs are running at the same time. In effect, there is concurrency present in the complete system.

This situation is different than everything we have seen so far. Up to this point, an individual program was running on the microcontroller, and while the actions that the computer takes are fast, only one thing is happening at a time, and they have a strict ordering. This is no longer the case. When two actions happen on distinct computers, it is entirely possible that we don’t know (or can’t know) precisely which one happened first.

The presence of concurrency in the system often brings with it more complicated reasoning about the correctness of the system as a whole. Here, we will address these issues by constraining the scope of designs that we consider, staying within the realm of operations that are relatively easy to reason about. A word of caution, however, as arbitrary concurrent techniques can be very difficult to understand.

In addition, communications between two different computers also encompasses the likelihood that the two computers are not just two copies of the
same type of computer, but are in reality two different types of computer as well. To help us understand and deal with these issues, in this chapter we will assume that our Arduino microcontroller is communicating with a desktop machine (or maybe a laptop machine) that is running a Java program. For those unfamiliar with Java, Appendix A compares the Java language with the Arduino C language.

10.1 Stream Concepts

We will organize our description of computer to computer communications in terms of an abstract concept called a *stream*. A *stream* is an arbitrary sequence of bytes, delivered from one computing entity to another. A stream has a source, which generates the sequence of bytes to be delivered, and a destination, which receives the sequence of bytes.

One of the advantages of the abstract concept of a stream is that one can author code to serve as a stream source without knowing the stream destination, and one can author code to serve as a stream destination without having to know the stream source. This improves the composability of software that uses the stream abstraction.

Abstract streams follow a few simple rules:

1. The data elements explicitly delivered via streams are *bytes*. Any other data type must be built on top of the byte stream.

2. The data bytes are delivered in order. That is, if the source sends byte A followed by byte B, the destination will receive byte A ahead of byte B.

3. Some streams guarantee reliable delivery. That is, if the source sends byte A, the destination will eventually receive byte A. This is not always the case, however, as some streams do not guarantee reliable delivery. In this case, a byte sent by the source might or might not eventually be received by the destination, or it might be delivered but have the wrong value (e.g., one or more bits within the byte might have been altered).

10.1.1 Streams on the Microcontroller

The [Serial] class is an example of a stream on the microcontroller. We have used [Serial.print()] and [Serial.println()] in a number of previous chapters as a mechanism for writing output to the serial port that connects the microcontroller to the IDE executing on a desktop machine. When running
the Serial Monitor in the IDE, code executing on the microcontroller is serving as the stream source, and the Serial Monitor is serving as the stream destination.

10.2 Delivery of Streams

One of the benefits of the stream abstraction is that the endpoints of a stream do not need to know about one another (i.e, the source’s implementation is independent of the destination and the destination’s implementation is independent of the source). In a similar way, another benefit is that neither endpoint needs to be aware of the physical mechanism used to deliver the bytes from source to destination.

Possible delivery mechanisms include copying bytes in the memory of a processor (e.g., from one program to another), sending bytes over a local area network (LAN) or the Internet, delivering bytes wirelessly, using a serial port implemented on top of a USB link, or a host of other paths.

10.3 Protocols

For our purposes, we will define a protocol as an agreement between the source and the destination of a communication path (or stream) as to how the data are to be interpreted.

10.3.1 Byte Delivery

The stream abstraction described above (in Section 10.1) essentially provides a sequence of individual bytes, reliably delivered, in order, from the source to the destination. In some circumstances, reliability of the byte delivery is not assured, and it is the responsibility of the higher-level protocol to deal with that issue.

In what follows, we will take a middle ground position on reliability, and assume that a byte sent by the source might or might not make it to the destination, but dropped bytes (as they are called) are relatively infrequent. We will also assume that any bytes that do get delivered are correct (i.e., not altered in transit).

In Sections 10.4 and 10.5 below, we discuss the mechanisms available to send and receive individual bytes.
10.3.2 Delivering Larger Data Items

Clearly, if we have the ability to send and/or receive individual bytes, to deliver larger data items it is necessary to use more than one byte for each larger data item. It is also important that both the source and the destination use the same convention for sending and receiving multi-byte data items.

Integers and Other Primitive Types

The convention for sending primitive types (integers, floats, and the like) is to send the bytes in order from most significant byte to least significant byte. This convention is sometimes referred to as “network order.” Always sending in network order ensures that the endpoints (either sender or receiver) don’t need to know the endianness of the other endpoint.

This convention, however, does not address the need for both endpoints to know the size of the primitive data type. For example, an integer data type on the Arduino platform is 2 bytes, while a Java integer is 4 bytes.

Arrays of primitive types are typically ordered from the lowest index to the highest index. As with the primitive data types, this doesn’t help the endpoints know the length of the array, which must therefore be communicated via some other mechanism.

Strings

Strings are data types that are frequently represented in noticeably different ways on different machines. For example, a string in C is stored as an array of chars with a null termination (i.e., the string’s length is represented by a ‘\0’, or null character, after the last valid character of the string. Each character in the string therefore occupies one byte of space (plus the additional byte to store the null termination).

In contrast, a string is Java is stored within a String object. In the standard implementation, the String class implements the underlying representation of the string as an array of characters, char[], plus a separate instance variable that retains the length of the array. Null termination is not used. Also, in Java, the char data type is 2 bytes long (using UTF-16 encoding of characters).

Given that different languages use different conventions for internal representations of strings, when one wishes to communicate a string from one computer to another it is insufficient to simply send the bytes using the sender’s internal representation and expect the receiver to interpret them correctly.
What is needed is an encoding of the string that is agnostic to the type of computer or language used by the sender and the receiver.

As described in Chapter 7, UTF-8 is a variable length character encoding mechanism that is widely used on the web. As such, it is well defined how to encode C strings and Java strings into UTF-8. A reasonable string communication protocol could then have the following form. First, the initial two bytes represent a 16-bit value (in network order, high-order byte first and low-order byte second) that describes the length of the string (in bytes). Next, this is followed by the UTF-8 encoded code units (bytes) that represent the individual characters of the string.

10.3.3 Messages

It is quite common to have a circumstance where more than one thing is to be communicated between two endpoints. For example, a microcontroller might be measuring temperature and pressure, and it wishes to send both values (possibly including a timestamp of when the measurements were taken) to a desktop computer. To accomplish this, we frequently will encapsulate the information to be delivered into one or more messages.

A message protocol that has been agreed to by both endpoints allows a range of capabilities that are, at the very least, more difficult without messaging.

1. Recovery from transmission errors.
2. Delivery of distinct data elements (e.g., temperature, pressure, time).
3. Delivery of distinct data types (e.g., integer, float).

In the discussion below, we will cover how to address each of the above capabilities in a message protocol.

Magic Numbers

Consider the following circumstance. A source is sending a series of 2-byte integer values, high byte first and low byte second, and at some point during the delivery process, the high byte of one value is lost. (Recall that our reliability assumption is that occasionally a byte that is sent isn’t received.)

What happens in the above circumstance? All of the integers that follow will be erroneously received by the destination. Low bytes of one integer are paired with high bytes of the following integer, and none of it is correct.
In a message protocol, one of the things we would like to accomplish is to recover from errors like the one above, and while some data might be irrevocably lost, at the very least we get the source and destination back in sync with one another so that correct information delivery can resume.

One of the ways we do this is to encapsulate any information we send in a message. A message will have header information that facilitates the delivery of the entire message (the header is typically independent of the content of the message), and payload information that is the data to be communicated from sender to receiver.

An element that is included in the header of many messaging protocols is a magic number. A magic number is a fixed byte pattern that is always present at the beginning of a message and is used to signify to the receiver that this is the beginning of a message.

While often the magic number is multiple bytes long (4 bytes is a common size), let us consider the use of a single-byte magic number, for example, 0x21. When the sender is preparing a message for delivery, it starts the first byte of the message with the byte 0x21. When the receiver is reading individual bytes from the stream, it can expect that the first byte of any message has the value 0x21. If not, it knows that the byte that it has just read is not the beginning of a message, and an appropriate response would be to discard all incoming bytes until it does see a 0x21.

A good choice of a magic number is a byte pattern than is relatively unlikely to appear anywhere else in the message. While this is impossible to guarantee in general (e.g., encrypted data can have any value), making the magic number infrequent improves the odds that the receiver’s actions in the above paragraph result in erroneously trying to start reading a message while actually still within the body of some other message. Clearly, the use of multiple-byte magic numbers can help this somewhat.

Magic numbers are not constrained to messaging protocols at all. One common use is in files, helping to identify the type of data stored in a file. For example, the first four bytes of a PDF document are 0x25, 0x50, 0x44, and 0x46, which are the ASCII encoding of the characters %PDF.

**Fields**

If we have defined the header of our messages (in the case we just described above, the magic number is the only thing in the header, other protocols might specify additional information) it is now time to specify how the data elements are to be delivered. We need to describe the payload.
10.4 Sending Messages: Composition

A common approach to data delivery is to not just send the data element itself, but also include additional descriptive information so that the receiver can more readily understand what the data represent. For example, what is the data type: two-byte integer vs. four-byte floating point value vs. string? Or, what is the meaning of the data, temperature reading, pressure reading, or timestamp?

Here, we suggest using a payload convention that goes by several names: key-value pair, name-value pair, tag-value pair, or sometimes attribute-value pair. Just to keep things simple, we will use the key-value pair terminology, but don’t be surprised if you see any of the above terms used elsewhere.

The basic idea is that each data element is communicated as an ordered pair, a key followed by a value. In our example protocol, the key indicates both the data type and the meaning of the value that follows it (e.g., the value is a four-byte integer that represents a timestamp, the number of milliseconds that has elapsed since the source program was started).

What is required then is that a list of keys (and what they represent) must be known both at the sending and receiving end of the communication. I.e., they must be listed explicitly as part of the protocol.

10.4 Sending Messages: Composition

Given that the source of a message knows the content prior to the actual composition and sending of the message, this task is actually fairly straightforward.

The low-level requirements include how does one send a single byte to the stream. In the Arduino libraries, the Serial class supports the delivery of an individual byte via the Serial.write() method.

In Java, the delivery of an individual byte depends upon which library is being used. If using the JSSC library, which is the one used by the Arduino IDE when communicating between the host computer and the microcontroller, the SerialPort class has a writeByte() method.

In the example that is included below, we will use the generic sendByte() syntax, with the understanding that it gets replaced in the actual code with one of the above options.

Once we have the ability to send individual bytes, there are two paths to designing message delivery. Option 1 is to formulate (compose) the message is memory (in an array of bytes) and then send the individual bytes out the data stream. Option 2 is to send bytes out the data stream as the message is
being formulated (composed). Either method works just fine. In the example below, we’ll use option 1.

Consider the task of sending a message that contains a timestamp value. Assume that the key for timestamp is 0x74 and the timestamp value is a 4-byte integer (e.g., the return value from `millis()`). If the byte array we are using to compose the message is named `msg`, the first two things to include in the message are the magic number and the key. This might look like the following:

```c
msg[0] = 0x24;
msg[1] = 0x74;
```
or if the appropriate constants have been defined:

```c
msg[0] = MAGIC_NUMBER;
msg[1] = TIMESTAMP_KEY;
```
The next four bytes of the message should contain the timestamp value. If that value is in the `unsigned long int` `time`, the logic to compose the message value is:

```c
msg[2] = (time >> 24) & 0xff; // most significant byte
msg[3] = (time >> 16) & 0xff;
msg[4] = (time >> 8) & 0xff;
msg[5] = time & 0xff; // least significant byte
```
What the above code does is to shift the appropriate 8 bits into the least significant byte and mask off any higher-order bits.

And the last task is to send the message to the stream:

```c
int msgLength = 6;
for (i=0; i<msgLength; i++) {
    sendByte(msg[i]);
}
```
which is accomplished by looping over the `msg` array.

## 10.5 Receiving Messages: Parsing

The challenge in receiving messages is that the receiver doesn’t know what message is coming next, and therefore the code to receive messages must recognize any legal messages. It is fairly straightforward to do this using a
finite state machine (FSM), and the diagram for such an FSM is illustrated in Figure 10.1.

In the protocol that this FSM recognizes, there are two message keys: 0x74 (with a 4-byte value representing a timestamp) and 0x54 (with a 2-byte value representing an integer temperature in degrees C). In the diagram, an edge is traversed upon the receipt of a byte. If that edge is labeled, the value of the incoming byte must match the label, otherwise the FSM traverses the unlabeled edge. While actions (other than state transitions) are not shown on the diagram, assume that the edges departing states C to H store the received byte into the appropriately named variable: first, second, third, or fourth (each of type byte).

When traversing the outbound edge from state F, the FSM will store the received timestamp into a 4-byte long integer (if on the microcontroller):

\[
\text{unsigned long int timestamp} = (\text{first} \ll 24) | (\text{second} \ll 16) | (\text{third} \ll 8) | \text{fourth};
\]
and when traversing the outbound edge from state H, the FSM will store the received timestamp into a 2-byte integer:

\[
\text{unsigned int temperature} = (\text{first} \ll 8) | \text{second};
\]

To implement the FSM described above, one still needs the capability of receiving individual bytes. Again, this will be different in C on the microcontroller versus in Java on a desktop machine. In Arduino C, the `Serial.available()` method returns the number of bytes that are available to be read from the serial port, and the `Serial.read()` method returns the first byte in the input buffer. As a result, a code structure like the following

```c
void loop() {
  if (Serial.available() > 0) {
    byte inputByte = Serial.read();
    switch (FSMstate) { // code to implement FSM
      case A:
        ...
      }
  }
}
```

will put each received byte into `inputByte` and then implement one transition (one step) of the finite state machine (whose state is retained in `FSMstate`).