Проект 1.2

Мартышкин пинг-понг

Теоретическая часть:

После первых шагов в адруино возникает вопрос, как форматировать значения показаний датчиков — как неструктурированные («сырые») двоичные значения или как ASCII? Это зависит от возможностей системы, получающей данные, а также от возможностей системы, через которые они проходят транзитом. Программам для персональных компьютеров часто бывает легче разбираться с неструктурированными двоичными значениями. Но многие сетевые протоколы, используемые в этой книге, основаны на ASCII. Кроме того, символы ASCII могут читаться людьми, вследствие чего данные оказывается более удобно передавать с помощью этого кода. Мы используем форматирование ASCII (первоначальный код) и, читая главу далее, мы поймем, почему это правильный выбор.

Что такое ASCII?

Код ASCII (American Symbolic Code for Information Interchange, американский стандартный код для обмена информацией) был создан в 1967 г. организацией American Standards Association (которая теперь называется ANSI[9]) в качестве средства, которое позволило бы обмениваться текстовыми сообщениями любым компьютерам, независимо от их операционной системы. Этот код присваивает однозначное число каждой букве, цифре и знаку препинания латинского алфавита. Все вводимые с клавиатуры символы преобразовываются в строку чисел, передаются по месту назначения, где эта строка чисел преобразовывается обратно в исходные символы. Кроме букв, цифр и знаков препинания, значения ASCII также присвоены некоторым непечатаемым символам, применяющимся для форматирования страницы — например операциям перехода на новую строку или возврата каретки (к началу строки), которые имеют значения ASCII 10 и 13 соответственно. Таким образом, вместе с сообщением можно передавать дополнительную информацию о формате его отображения. Эти символы называются *символами управления* и занимают первые 32 значения набора ASCII (ASCII 0–32).

Доступные 128 значений ASCII позволяют представить все буквы, цифры и знаки препинания английского алфавита, а также управляющие символы. Но этих значений недостаточно для представления букв неанглийских алфавитов, а имеющиеся символы управления не предоставляют достаточных возможностей управления выводом для графических пользовательских интерфейсов. Поэтому в настоящее время вместо ASCII в качестве нового стандарта для обмена текстовыми сообщениями используется более развитый код, который называется Unicode и является надмножеством ASCII. Unicode содержит несколько наборов символов, позволяющих отображать алфавиты всех языков.

Одним из самых сбивающих с толку моментов передачи данных является разница между двоичным протоколом и текстовым. Код ASCII и его современный родственник Unicode позволяют преобразовывать любой поток текстовой информации в двоичную информацию, читаемую компьютерами, и наоборот. Понимание разницы между текстовыми и нетекстовыми протоколами и того, какой из них выбрать, является необходимыми предварительными условиями для понимания взаимодействия между электронными устройствами.

Разве не все данные двоичные?

Ну, с одной стороны, да. В конце концов, компьютеры работают только на двоичной логике. Поэтому даже текстовые протоколы в своей основе являются двоичными. Но нам легче составлять сообщения в виде обычного текста, предоставляя компьютерам преобразовывать их в двоичные биты самостоятельно. Например, возьмем фразу «1-2-3, go». В следующей таблице показана разбивка этой фразы на составляющие ее символы с соответствующими представлениями ASCII и двоичного кода для каждого символа.

Символ	1	-	2	-	3	,		g	o	!
Код ASCII	49	45	50	45	51	44	32	103	111	33
Двоичный код	00110001	00101101	00110010	00101101	00110011	00101100	01000000	01100111	01101111	01000001

Может показаться, что в двоичном коде слишком много единиц и нулей, но это только до тех пор, пока мы не осознаем, что компьютер может считывать миллионы или даже миллиарды битов в секунду. А как насчет скорости передачи двоичных данных другому компьютеру? Давайте считать. В этой фразе 80 битов. Допустим, что мы передаем их по последовательному TTL-каналу со скоростью 9600 битов в секунду. Если к каждому байту добавить стартовый стоповый биты (как это делается последовательными протоколами TTL и RS-232), мы получим суммарно 96 битов, то есть, с учетом заявленной скорости это сообщение можно передать за $^{1}/_{100}$ долю секунды. В общем-то, довольно быстро.

Ну, а как насчет тех случаев, когда требуется передать нетекстовое сообщение? Пусть мы передаем текстом строку значений пикселов RGB, каждое из которых может быть в диапазоне от 0 до 255:

102,198,255,127,127,212,255,155,127,

Мы знаем, что каждая составляющая цвета передается восемью битами (диапазон чисел от 0 до 255 содержит 256 значений, или 2^8, или 8 битов), поэтому для одного пиксела требуется самое большее три байта. Но при передаче их в текстовой форме, когда для каждого символа требуется один байт, понадобится втрое больше байтов плюс разделяющая запятая. В частности, для приведенной строки потребовалось бы 36 байтов, чтобы передать ее 9 значений. Но если передавать ту же самую информацию в исходном формате, потребовалось бы всего лишь девять байтов. Так что, когда передаются значения миллионов пикселов, разница очень ощутима. В случаях подобных описанному, когда передаются чисто числовые данные, имеет смысл передавать их без преобразования в текстовый формат. Лишь когда нам нужно передать именно текст (например, электронную почту или гипертекст), следует закодировать его в ASCII или Unicode. Впрочем, когда передается небольшой объем данных, кодирование этих данных в ASCCI или Unicode может быть полезным для отладки, поскольку большинство программ терминалов для последовательной или сетевой связи интерпретируют все получаемые ими байты как текст, в чем мы имели возможность убедиться. Соответственно:

- если нужно передать большой объем числовых данных, передавайте их в сыром (не отформатированном) двоичном формате;
- текст передавайте в формате ASCII или Unicode;
- в случае небольшого объема данных любого типа, передавайте их в любом удобном для вас формате

Интерпретация двоичного протокола

Поскольку в этой книге мы будем иметь дело в основном с текстовыми протоколами, нам не потребуется посвящать много времени интерпретации битов, составляющих байты. Но для двоичных протоколов часто требуется знать значения каждого бита, входящего в состав байта, поэтому весьма полезно хоть немного знать об архитектуре байта и представлять, как манипулировать его содержимым. Поэтому давайте поговорим немного о битах.

Двоичные протоколы часто применяются для обеспечения взаимодействия между микросхемами в сложных устройствах. И особенно часто при этом используются синхронные последовательные протоколы. Большинство

устройств SPI и I^2C имеют небольшие наборы команд. Их однобайтные коды операций часто объединяются с параметрами команды в одном байте. Команды этих протоколов обычно записываются в шестнадцатеричном или двоичном представлении, или и в том, и в другом. Мы уже видели в этой книге шестнадцатеричное (с базой 16) представление значений. Подобно тому, как шестнадцатеричные значения начинаются с символов 0x, двоичные числа в Arduino, а также в языке C, начинаются с символов `0b`, например, `0b10101010`.

Теперь можете ли вы сказать, какая цифра в следующем числе имеет наибольшее значение?

$2,508

Ответить легко — цифра 2, так как она представляет самое большое количество — две тысячи, или две группы по 10^3. То есть цифра 2 находится здесь в самом старшем разряде. Число в этом примере показано в десятеричном представлении, но тот же самый принцип применяется и к числам в двоичном представлении. Какой бит самый старший в следующем двоичном числе?

0b10010110

Самым старшим битом здесь является самая левая единица, так как она представляет одну группу значений по 128, или 2^7.

Обычно, когда нам приходится иметь дело с битами, нам не столько важны их десятичные значения, сколько их позиция в байте. Например, в рассматриваемом в *главе 7* протоколе API для радио XBee используется 2-байтное значение, называющееся Channel Indicator, которое указывает активные аналоговые и цифровые вводы. В этом 16-битном значении 9 младших битов представляют цифровые вводы, следующие 6 битов — 4 аналоговых ввода, а самый старший бит не задействован. Использование этого значения рассматривается более подробно в последующих главах книги. Если активировать все аналоговые вводы и ни одного цифрового ввода, Channel Indicator будет иметь следующее двоичное значение:

0111111000000000

Как можно видеть, каждый из 9-ти младших битов, представляющих цифровые каналы, имеет значение 0, а каждый из следующих 6-ти битов, представляющих аналоговые каналы, имеет значение 1, означающее, что эти каналы активированы. В данном случае для нас неважно десятичное значение этого 2-байтного числа — что для нас имеет важность, так это значение каждого его бита, чтобы мы могли определить, какие каналы использовать.

Иногда значения двух байтов объединяются в одно 16-битное значение, или четырех байтов — в одно 64-битное значение. Примером такого объединения может служить значение выборки ввода/вывода аналого-цифрового преобразования протокола радио XBee. Разрешение выборки каждого устройства АЦП составляет 10 битов, поэтому для каждой выборки протокол резервирует два байта. Таким образом, максимальное возможное значение выборки равно 1023. Если рассматривать значения побайтно, максимальное значение первого байта будет 3, а второго — 255. В шестнадцатеричном представлении это значение можно записать как 0x3FF. Чтобы получить десятичное значение, умножаем первый байт на 256 (3 × 256 = 768) и складываем его с десятичным значением второго байта (FF = 255), получая в результате то же самое 1023 (768 +255 = 1023). Иными словами, байты можно рассматривать как числа с базой 256.

Польза шестнадцатеричного представления

Поскольку мы можем манипулировать данными двоичных протоколов побитно, вы, наверное, задаетесь вопросом, зачем нам нужно шестнадцатеричное представление? Это представление, конечно же, полезно для работы с группами по 16. Например, протокол MIDI определяет блоки инструментов, каждый из которых содержит 128 инструментов, а количество каналов, на которых может играть каждый инструмент, может быть до 16. Возьмем следующий пример. Шестнадцатеричное значение 0x9n представляет команду включения ноты (Note On), где n означает номер канала от 0 до A в шестнадцатеричном формате. Так, команда 0x9A означает включение ноты на канале A (или на канале 10 в десятичном формате). А команда 0x8A означает выключение ноты (note off) на канале A. Теперь, когда мы знаем, что инструменты MIDI организованы в группы по 16, вполне имеет смысл манипулировать ими в шестнадцатеричном

формате. Подобным образом используемые в веб-страницах цвета представляются значениями их составляющих цветов: красного (r — red), зеленого (g — green) и синего (b — blue). Например:

```
красный = 0xFF0000 зеленый = 0x00FF00 синий = 0x0000FF
```

Зная это, можно легко определить относительный уровень каждого составляющего цвета по его шестнадцатеричному значению. Например, цвет 0x26B0E7 содержит сравнительно низкую красную составляющую (0x26, или 38 из возможных 255), довольно высокую зеленую (0xB0, или 176) и очень высокую синюю (0xE7, или 231). Эти базовые цвета создают конечный цвет приятного зеленовато-голубого оттенка.

Датчик левой руки (0-1023)	***Датчик правой руки (0-1023)***	***Кнопка сброса (0 или 1)***	***Кнопка подачи (0 или 1)***	***Символ возврата каретки, символ перевода строки***
1–4 байта	1–4 байта	1 байт	1 байт	2 байта

Пакеты данных, заголовки, полезные нагрузки и «хвосты»

Теперь, когда мы добились передачи данных от одного устройства (микроконтроллера при мартышке) к другому (компьютеру, исполняющему программу Processing), рассмотрим более внимательно последовательность байтов, которую мы передаем. Обычно, эта последовательность имеет следующую структуру:

Части этой последовательности разделены байтом со значением ASCII 44, то есть запятой. Мы только что создали еще один протокол передачи данных. Байты значений датчиков и разделяющие их запятые — это *полезная нагрузка*, а символы возврата каретки и перевода строки — *«хвосты»*. Запятые служат *разделителями*. Этот протокол не имеет заголовка, но многие другие протоколы — имеют.

Заголовок — это последовательность байтов, которая идентифицирует следующие за ней данные, а также может содержать описание этих данных. В сетевом окружении, где одно и то же сообщение может быть получено многими устройствами, заголовок может содержать адрес отправителя, или получателя, или и то, и другое. Таким образом, любое устройство может сначала прочитать только заголовок, чтобы определить, нужно ли ему читать все сообщение. Некоторые заголовки могут быть весьма простыми — как один байт постоянного значения, указывающий на начало сообщения. В нашем примере, в котором заголовок сообщения отсутствует, эту роль играет хвост сообщения, отделяя одно сообщение от следующего.

В сетевом окружении постоянно отправляется множество сообщений, подобных этому. Каждая отдельная группа байтов сообщения называется *пакетом* и состоит из заголовка, полезной нагрузки и, обычно, «хвоста». Для каждой отдельной сети существует *максимальный размер пакета*. В нашем примере размер пакета определяется размером буфера на персональном компьютере для накопления последовательных данных. Среда Processing может работать с содержимым буфера размером в несколько тысяч байтов, так что наш 16-байтовый пакет не представляет для нее никакой сложности. Сообщения намного большего размера нам пришлось бы разделять на несколько пакетов, а затем собирать исходное сообщение из полученных пакетов. В таком случае заголовок пакета может содержать его номер, чтобы получатель знал, как упорядочить полученные пакеты для восстановления исходного сообщении.

Практическая часть:

В этом проекте мы, по сути, создадим аналог компьютерной мыши. Ведь если рассматривать мышь в качестве объекта данных, она будет выглядеть, как показано на рис. 2.10: есть сигналы на входе, есть реакция на выходе.

Требуемые компоненты

- Резистивные датчики изгиба, 2 шт.
- Кнопочные выключатели, 2 шт.
- Резисторы номиналом 4 кОм, 4 шт.
- Беспаечная макетная плата, 1 шт.
- Совместимая с Arduino плата (на рис. 2.11 *вверху* показана плата MKR1000, а *внизу* — плата Arduino 101), 1 шт.

  Используемые возможности платы: цифровой ввод, аналоговый ввод, УАПП (UART).
- Персональный компьютер.

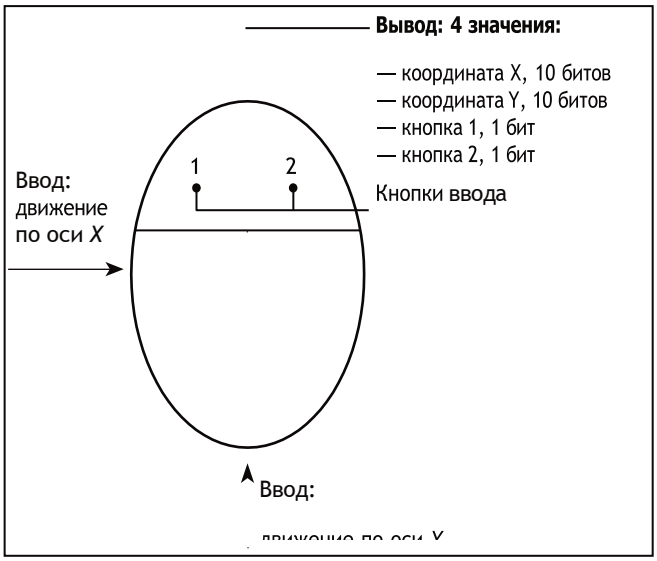
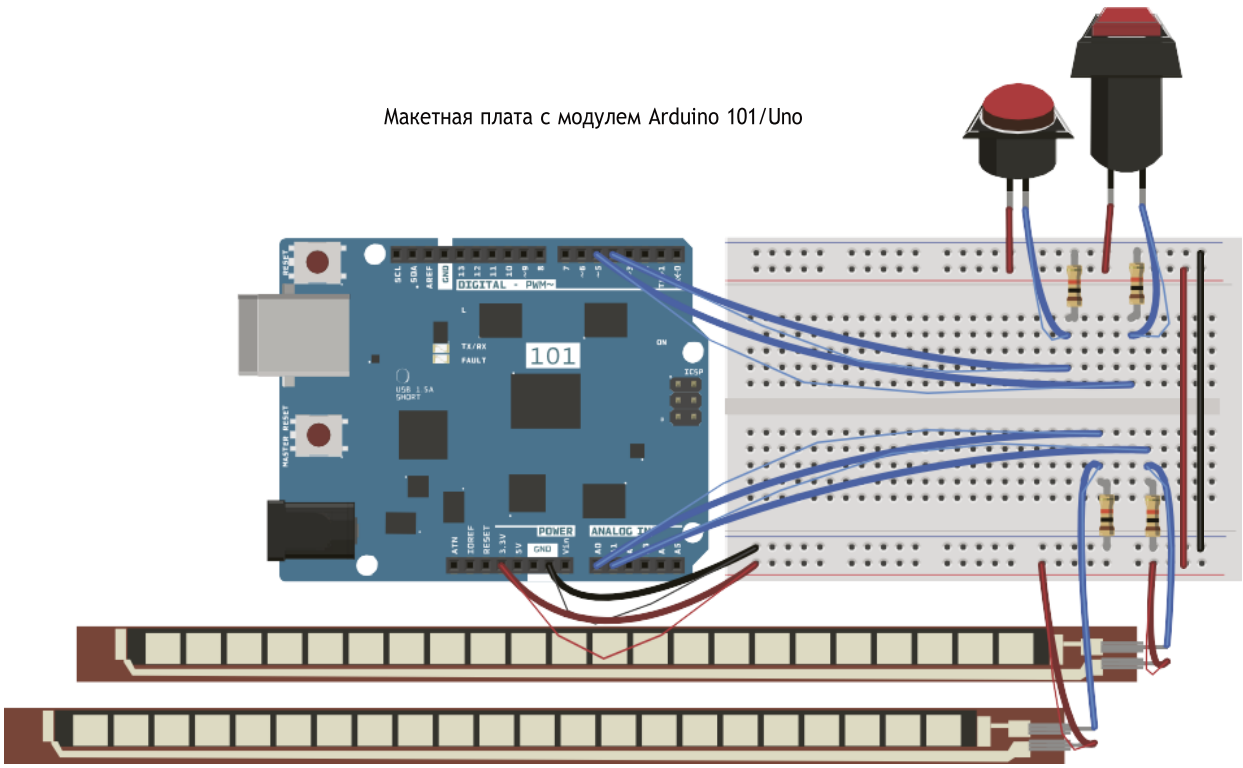


Рис. 2.10. Представление мыши в виде объекта данны



Принципиальная схема

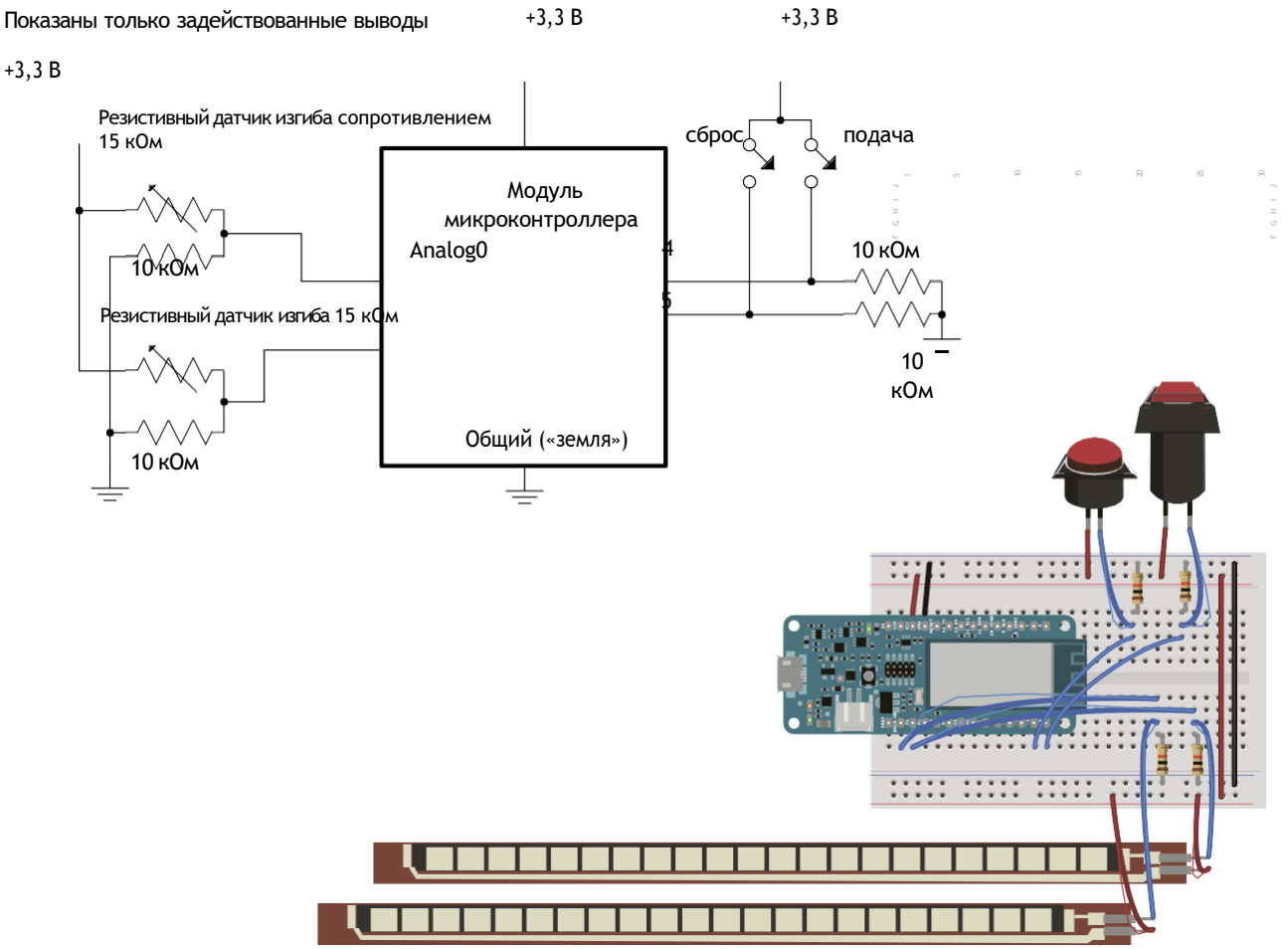


Рис. 2.11. Монтажные (*вверху* и *внизу*) и принципиальная (*в центре*) схемы проекта Датчики здесь — для удобства вычерчивания — показаны с короткими проводами, но для реального проекта датчики нужно подсоединять проводами значительно большей длины

Пишем код

Теперь загрузите и выполните в Arduino следующий код, чтобы проверить работу датчиков изгиба.

В окне монитора порта среды Arduino — или другой используемой вами программы терминала последовательной связи с установленной скоростью обмена 9600 битов в секунду, как мы делали в *главе 1* — должен выводиться поток данных следующего вида:

```
284,284,1,1
285,283,1,1
286,284,1,1
289,283,1,1
```

Как и было запрограммировано, значения показаний разделены запятыми, а каждый набор показаний выводится отдельной строкой.

```
/*
Считыватель показаний датчиков
Контекст: Arduino

Считывает показания с двух аналоговых входов и двух
цифровых входов и выводит их значения.

Подключения:
Аналоговые датчики — к контактам аналогового ввода A0 и A1
Кнопки — к контактам цифрового ввода 4 и 5
/*



const int leftSensor = A0;  // аналоговый ввод для левой руки
const int rightSensor = A1; // аналоговый ввод для правой руки
const int resetButton = 4;  // цифровой ввод для кнопки сброса
const int serveButton = 5;  // цифровой ввод для кнопки подачи

int leftReading = 0;     // показания датчика левой руки
int rightReading = 0;    // показания датчика правой руки
int resetReading = 0;    // данные кнопки сброса
int serveReading = 0;     // данные кнопки подачи

void setup() {
   // настраиваем последовательное соединение:
   Serial.begin(9600);
   // configure the digital inputs:
   pinMode(resetButton, INPUT);
   pinMode(serveButton, INPUT);
}

void loop() {
   // считываем показания аналоговых датчиков:
   leftReading = analogRead(leftSensor);
   rightReading = analogRead(rightSensor);

   // считываем показания цифровых датчиков:
   resetReading = digitalRead(resetButton);
   serveReading = digitalRead(serveButton);

   // Выводим результаты на экран:
   Serial.print(leftReading);
   Serial.print(',');
   Serial.print(rightReading);
   Serial.print(',');
   Serial.print(resetReading);
   Serial.print(',');
   /* выводим на экран последнее показание датчика
с помощью функции println(), чтобы каждый набор из четырех
показаний был на отдельной строке:
   /*
   Serial.println(serveReading);
}
```

Теперь попробуйте заменить фрагмент кода, который выводит на экран результаты, следующим кодом:

```
Serial.write(leftReading);
Serial.write(44);
Serial.write(rightReading);
Serial.write(44);
Serial.write(resetReading);
Serial.write(44);
/* выводим на экран последнее показание датчика с помощью
функции println(), чтобы каждый набор из четырех показаний
был на отдельной строке:
/*
Serial.write(serveReading);
Serial.write(10);
Serial.write(13);
```

Результаты исполнения этого кода в окне монитора порта будут примерно такими:

```
.,P,,
(,F,,
(,A,,
),I,,
```

Что же здесь происходит? В исходном примере используется метод `Serial.print()`, который выводит на экран показания датчиков в виде их значений кода ASCII, а модифицированный код с помощью метода `Serial.write()` выводит на экран «сырые» двоичные значения. Монитор порта (или другая программа терминала последовательной связи) предполагает, что каждый полученный ею байт является символом ASCII, поэтому в модифицированном коде он отображает символы ASCII соответственно их «сырым» двоичным значениям. Например, значения 13 и 10 соответствуют символам ASCII «возврат каретки» и «новая строка», а значение 44 — символу запятой. Эти значения и посылаются между значениями показаний датчиков в модифицированном коде. Источником загадочных символов являются переменные показаний датчиков (`leftValue`, `rightValue`, `reset` и `serve`). В третьей строке вывода, когда значение показаний второго датчика станет равно, например, 65, мы увидим символ «A», так как этому символу ASCII соответствует значение 65. Полный список значений ASCII для каждого символа можно найти на сайте **www.asciitable.com**.

Добившись стабильной отправки микроконтроллером данных от датчиков на терминал, можно попытаться отправлять эти данные в программу, которая будет интерпретировать их как игру типа пинг-понга. Эта программа долж на исполняться на компьютере, к которому подключена плата Arduino. Для создания такой программы хорошо подойдет среда Processing.

Возвратите прежний код

Прежде чем переходить к следующему разделу, в котором для интерпретации вывода этой программы мы создаем код в среде Processing, произведенную замену нужно убрать и возвратить прежний код.

Пишем код

Создайте приложение Processing и введите в него следующий код:

```
 Serial myPort;    // Переменная последовательного порта
String resultString; // Строчная переменная для результатов


void setup() {

   size(480, 130); // устанавливаем размер окна апплета
   printArray(Serial.list()); // Выводим на экран все

                              // доступные последовательные порты
```

```
   // На моем компьютере порт микроконтроллера обычно
   // первый порт в списке,
   // поэтому я открываю Serial.list()[0].
   // Измените 0 на номер последовательного порта,
   // к которому подключен ваш микроконтроллер:
   String portName = Serial.list()[0];
   // открываем последовательный порт:
   myPort = new Serial(this, portName, 9600);


   // считываем байты в буфер, пока не дойдем до символа
   // перевода строки (ASCII 10):
 myPort.bufferUntil('\n');
}

void draw() {
   // задаем цвет фона и заливки для окна апплета:
   background(#044f6f);
   fill(#ffffff);
   // выводим строку в окне:
   if (resultString != null) {
      text(resultString, 10, height/2);
   }
}

/* Метод serialEvent() исполняется автоматически
в программе каждый раз, когда в буфер записывается
байт со значением, определенным в методе bufferUntil()
в процедуре setup():
/*

void serialEvent(Serial myPort) {
   // Считываем данные из последовательного буфера:
   String inputString = myPort.readStringUntil('\n');

   // Отбрасываем символы возврата каретки
   // и перевода строки из строки ввода:
   inputString = trim(inputString);
   // Очищаем переменную resultString:
   resultString = "";

   // Разделяем входную строку по запятым и преобразовываем
   // полученные фрагменты в целые числа:
   int sensors[] = int(split(inputString, ','));

   // Добавляем значения к строке результата:
   for (int sensorNum = 0; sensorNum < sensors.length;
sensorNum++) {
      resultString += "Sensor " + sensorNum + ": ";
      resultString += sensors[sensorNum] + '\t';
   }
   // Выводим результат на экран:
   println(resultString);
}
```

Отвлечемся на время от создания кода и протестируем уже имеющийся скетч. Обязательно закройте окно монитора порта (или последовательный порт, если вы используете другую программу терминала последовательной связи), чтобы освободить последовательный порт компьютера. Теперь выполните нашу программу Processing. Если она не содержит никаких ошибок, то должна вывести в консоль и в окно апплета список значений датчиков, пример которого показан на рис. 2.13.

Рис. 2.13. Вывод списка значений датчиков в окне апплета

Теперь мы можем использовать эти данные, чтобы играть в пинг-понг. Прежде всего, добавьте несколько переменных в начало скетча Processing — перед методом `setup()` — и от- корректируйте этот метод для установки размера окна и инициализации некоторых переменных:

Числа с плавающей запятой

В качестве переменных для диапазона действия ракеток в этом примере используются числа с плавающей запятой (`floats`), поскольку в результате деления чисел целого типа также полу- чается целое число. Например, в результате деления целых чисел: 480 на 400 — получится 1, а не 1,2. Точно так же частное от деления двух целых чисел: 400 на 480 — будет 0, а не 0,833. Соответственно, деление двух целых чисел одинакового порядка величины дает бесполезные для нас результаты. Имейте это в виду при использовании функций масштабирования наподобие `map()`.

```
float leftPaddle, rightPaddle; // переменные для значений
                               // датчиков изгиба
int resetButton, serveButton; // переменные для кнопок
int leftPaddleX, rightPaddleX; // горизонтальные позиции
                               // ракеток
int paddleHeight = 50; // вертикальный размер ракеток
int paddleWidth = 10;  // горизонтальный размер ракеток
float leftMinimum = 120; // минимальное значение левого
                         // датчика изгиба
float rightMinimum = 100; // минимальное значение правого
                          // датчика изгиба
float leftMaximum = 530;  // максимальное значение левого
                          // датчика изгиба
float rightMaximum = 500; // максимальное значение правого
                          // датчика изгиба

void setup() {
   size(640, 480); // устанавливаем размер окна апплета

   String portName = Serial.list()[0];
   // открываем последовательный порт:
   myPort = new Serial(this, portName, 9600);

   // считываем байты в буфер, пока не дойдем до
   // символа перевода строки (ASCII 10):
   myPort.bufferUntil('\n');

   // инициализируем значения датчиков:
   leftPaddle = height/2;
   rightPaddle = height/2;
   resetButton = 0;
   serveButton = 0;

   // инициализируем горизонтальные позиции ракеток:
   leftPaddleX = 50;
   rightPaddleX = width - 50;

   // рисуем фигуры без охантовки:
   noStroke();
}
```

Теперь заменим метод `serialEvent()` следующей версией, которая помещает полученные значения в переменные датчиков:

```
void serialEvent(Serial myPort) {
   // считываем данные из последовательного буфера:
   String inputString = myPort.readStringUntil('\n');

   // отбрасываем символы возврата каретки и перевода
строки
   // из строки ввода:
   inputString = trim(inputString);
   // очищаем переменную resultString:
   resultString = "";

   // разделяем входную строку по запятым и преобразовываем
   // полученные фрагменты в целые числа:
   int sensors[] = int(split(inputString, ','));
   // если получены все строки значений датчиков,
используем их:
   if (sensors.length == 4) {
      // масштабируем данные датчиков изгиба к диапазону
      // ракеток:
      leftPaddle = map(sensors[0], leftMinimum,
leftMaximum, 0, height);
      rightPaddle = map(sensors[1], rightMinimum,
rightMaximum, 0, height);

      // присваиваем значения кнопок соответствующим
      // переменным:
      resetButton = sensors[2];
      serveButton = sensors[3];

      // добавляем значения к строке результата:
      resultString += "left: "+ leftPaddle + "\tright: " +
rightPaddle;
      resultString += "\treset: "+ resetButton + "\tserve: "
+ serveButton;
   }
}
```

Наконец, модифицируем метод `draw()`, добавив в него код для рисования ракеток (новый код выделен полужирным шрифтом).

```
void draw() {
   // задаем цвет фона и заливки для окна апплета:
   background(#044f6f);
   fill(#ffffff);

   // рисуем левую ракетку:
   rect(leftPaddleX, leftPaddle, paddleWidth, paddleHeight);

   // рисуем правую ракетку:
   rect(rightPaddleX, rightPaddle, paddleWidth, paddleHeight);
}
```

Простейшая сеть

Ракетки могут не отображаться до тех пор, пока не поступят данные от датчиков изгиба, т. е. до тех пор, пока мы их не согнем. Функция `map()` сопоставляет диапазонам значений датчиков изгиба диапазоны движений ракеток, но диапазоны датчиков изгиба нужно еще определить. Для этого важно, чтобы датчики изгиба были надежно прикреплены к рукам мартышки, так как для тонкой настройки системы нам необходимо иметь их в том месте, в котором они будут использоваться. Закрепив датчики на руках мартышки, снова выполните программу, сгибая ее руки и наблюдая за данными левого и правого датчиков. Запишите максимальное и минимальное значения для каждой руки. Затем присвойте эти значения переменным `leftMinimum`, `leftMaximum`, `rightMinimum` и `leftMaximum` метода `setup()`. Когда эти переменные настроены должным образом, при движении рук мартышки движения ракеток должны покрывать всю высоту экрана.

Наконец, настало время добавить в программу теннисный мячик. Он будет перемещаться слева направо по диагонали. При столкновении с верхним или нижним краем экрана мячик отскакивает и изменяет вертикальное направление движения. По достижении левого или правого края экрана мячик возвращается в центр. При касании одной из ракеток мячик отскакивает и изменяет горизонтальное направление движения. Для всего этого нам потребуется добавить пять новых переменных в начало скетча, непосредственно перед методом `setup()`:

```
int ballSize = 10; // размер мячика
int xDirection = 1; // горизонтальное направление движения
                    // мячика
                    // влево: -1, вправо: 1
int yDirection = 1; // вертикальное направление движения
                    // мячика
                    // вверх: -1, вниз: 1
int xPos, yPos;     // горизонтальное и вертикальное
                    // положение мячика
```

А в конце метода `setup()` необходимо задать мячику начальную позицию в центре окна:

```
// инициируем позицию мячика в центре окна:
xPos = width/2;
yPos = height/2;
```

Теперь добавим в конец скетча два метода: `animateBall()` и `resetBall()`. Эти методы вызываются из метода `draw()`:

```
void animateBall() {
   // если мячик движется влево:
   if (xDirection < 0) {
      // если мячик находится слева от левой ракетки
      if ((xPos <= leftPaddleX)) {
      // если мячик находится между верхом и низом левой
ракетки:
            if((leftPaddle - (paddleHeight/2) <= yPos) &&
(yPos <= leftPaddle + (paddleHeight /2))) {
               // изменяем направление горизонтального
движения на обратное:
               xDirection =-xDirection;
            }
         }
      }
```

»

```
   // если мячик движется вправо:
   else {
      // если мячик справа от правой ракетки
      if ((xPos >= ( rightPaddleX + ballSize/2))) {
         // если мячик находится между верхом и низом
         // правой ракетки:
         if((rightPaddle - (paddleHeight/2) <=yPos) &&
(yPos <= rightPaddle + (paddleHeight /2))) {
            // изменяем направление горизонтального
            // движения на обратное:
   xDirection =-xDirection;
         }
      }
    }

   // если мячик выходит за пределы окна слева:
   if (xPos < 0) {
      resetBaVL();
   }
   // если мячик выходит за пределы окна справа:
   if (xPos > width) {
      resetBall();
   }

   // не даем мячику выйти за верхний или нижний предел окна
   if ((yPos - ballSize/2 <= 0) || (yPos +ballSize/2 >=height)) {
      // изменяем направление вертикального движения мячика
      // на обратное:
      yDirection = -yDirection;
   }
   // обновляем местонахождение мячика:
   xPos = xPos + xDirection;
   yPos = yPos + yDirection;

   // рисуем мячик:
   rect(xPos, yPos, ballSize, ballSize);
}

void resetBall() {
   // возвращаем мячик обратно в центр окна:
   xPos = width/2;
   yPos = height/2;
}
```

Мы уже почти готовы запустить мячик в игру. Но сначала нужно позаботиться о кнопках сброса и подачи мячика. Добавьте в начало кода (непосредственно перед методом `setup()` со всеми другими объявлениями переменных) еще одну переменную, кото- рая будет отслеживать, движется ли мячик. Добавьте также еще две переменные для ведения счета.

```
boolean ballInMotion = false; // мячик движется?
int leftScore = 0;
int rightScore = 0;
```

Мячик должен начинать движение только с подачи. Обеспечивающий это код нужно разместить в конце метода `draw()`: первый оператор `if()` запускает мячик в движение при нажатии на кнопку подачи, второй оператор `if()` перемещает запущенный мячик и, наконец, третий оператор `if()` возвращает мячик обратно в центр окна и обнуляет счет при нажатии кнопки сброса:

```
// вычисляем местонахождение мячика и прорисовываем его:
if (ballInMotion == true) {
   animateBaH();
}

// если нажата кнопка подачи, запускаем мячик в движение:
if (serveButton == 1) {
   ballInMotion = true;
}

// если нажата кнопка сброса, обнуляем счет и запускаем
// мячик в движение:
if (resetButton == 1) {
   leftScore = 0;
   rightScore = 0;
   ballInMotion = true;
}
```

Модифицируйте метод `animateBall()`, чтобы увеличивать счет, когда мячик выходит за пределы окна слева или справа (добавленный код выделен полужирным шрифтом):

```
// если мячик выходит за пределы окна слева:
if (xPos < 0) {
   rightScore++;
   resetBall();
}
// если мячик выходит за пределы окна справа:
if (xPos > width) {
   leftScore++;
   resetBall();
}
```

Для вывода счета добавьте перед методом `setup()` новую глобальную переменную:

```
int fontSize = 36; // размер шрифта для отображения счета
```

Добавьте в конец метода `setup()` две строки кода, инициализирующие шрифт:

```
// создаем шрифт из третьего шрифта, доступного системе:
P5ont my5ont = create5ont(P5ont.list()[2], fontSize);
text5ont(my5ont);
```

Наконец, добавьте в конец метода `draw()` две строки кода для отображения счета:

```
// выводим счет на экран:
text(leftScore, fontSize, fontSize);
text(rightScore, width-fontSize, fontSize);
```

Все! Теперь мы можем играть в «Мартышкин пинг-понг» (рис. 2.14). Чтобы сделать игру более увлекательной, возьмите вторую мартышку и один из датчиков изгиба вставьте в нее, чтобы можно было играть с кем-нибудь на пару.

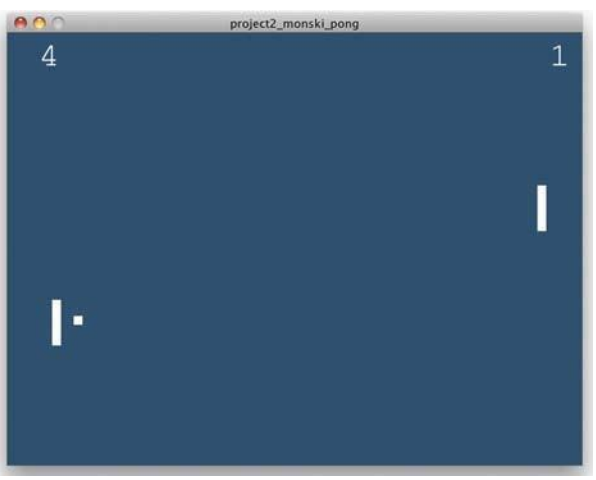


Рис. 2.14. Игра «Мартышкин пинг-понг» на экране компьютера

Иногда ракетки движутся с отставанием от них, а могут на долю секунды и застывать. Причина этому — асинхронная связь между компьютером и микроконтроллером.

Хотя оба эти устройства предварительно согласовали скорость обмена данными, это вовсе не означает, что получающая данные программа должна ими воспользоваться сразу же вслед за получением.
Входящие данные в действительности обрабатываются специальным процессом операционной системы и сохраняются в буфере памяти, называющемся *последовательным буфером*.
В большинстве персональных компьютеров каждому последовательному порту выделяется отдельный буфер памяти, который может содержать несколько тысяч байтов данных. Программа, которая использует эти данные (Processing в нашем случае), жонглирует несколькими задачами, — такими как перерисовка экрана, выполнение соответствующих математических вычислений, а также делит с другими программами процессорное время, которое выделяется операционной системой каждой программе по очереди и по приоритету. Соответственно, она может извлекать данные из буфера реже, чем сотни раз в секунду, хотя данные поступают в буфер намного быстрее.
Существует иной способ реализации взаимодействия между двумя устройствами, который может уменьшить масштаб этой проблемы. В частности, если программа Processing будет запрашивать данные только тогда, когда она в них нуждается, и, если микроконтроллер в ответ на запрос будет посылать только один пакет данных, синхронизация взаимодействия компьютера и микроконтроллера станет более тесной.

Для реализации этого способа сначала добавьте следующий код в метод `startup()` скетча Arduino (см. скетч *«Считыватель показаний датчиков»* в начале *разд. «Проект 2. Мартышкин пинг-понг»*). Исполняя этот код, Arduino посылает сообщение до тех пор, пока не получит ответ от Processing:

```
while (Serial.available() <= 0) {
   Serial.println("hello"); // отправляем начальное
   сообщение
}
```

Затем поместим весь метод `loop()` скетча Arduino в оболочку оператора `if()`, как показано здесь (добавленный код выделен жирным шрифтом):

```
void loop() {
   if (Serial.available() > 0) {
      // считываем данные из последовательного буфера:
      // значение байта не имеет значения — только факт
      его
      // наличия:
      int inByte = Serial.read();
      // остальная часть старого кода основного цикла идет
      сюда
      // ...
   }
}
```

На следующем шаге добавим выделенный полужирным код в конец метода `serialEvent()` скетча Processing, реализующего игру «Мартышкин пинг-понг»:

```
void serialEvent(Serial myPort) {
  // сюда вставляется остальной код метода
  myPort.write('\r'); // посылаем символ возврата
  каретки
}
```

Теперь ракетки на экране компьютера должны двигаться намного более плавно. Наше решение работает следующим образом: микроконтроллер посылает строку `"hello"` до тех пор, пока не получит какие-либо данные, после чего начинает исполнять основной цикл. Этот цикл считывает полученный байт просто для того, чтобы очистить последовательный буфер, отправляет свои данные один раз, а затем ожидает прибытия других данных. Если цикл ничего не получает, он также ничего и не отправляет.

Тем временем среда Processing запускает свою программу, ожидая входящие данные. Когда программа получает любую строку, оканчивающуюся символом новой строки, вызывается метод `serialEvent()`, как это делалось и ранее. Программа считывает строку и, если она содержит запятые, разбивает строку по запятым, извлекая значения датчиков, — и ранее это делалось так же. Если строка не содержит запятых (например, если это строка `"hello"`), программа Processing ничего с ней не делает.

Изменение в скетче Processing находится в конце метода `serialEvent()`. Наш дополнительный код отправляет байт обратно микроконтроллеру, который, получив новый байт, отправляет другой пакет данных, и цикл снова повторяется. Таким образом, последовательный буфер на стороне Processing никогда не заполняется и всегда содержит самые последние показания датчиков.

Значение полученного микроконтроллером байта несущественно. Этот байт служит лишь сигналом от скетча Processing, извещающим микроконтроллер, что скетч готов получать новые данные. Несущественно и сообщение `"hello"`, посылаемое микроконтроллером, поскольку оно лишь служит для Processing триггером, инициирующим отправку первоначального байта, поэтому Processing просто игнорирует его. Такой метод управления потоком данных иногда называется *методом рукопожатия* или *вызова и ответа*. Этот метод может быть полезным для обеспечения надежного обмена при передаче пакетов данных.

Усложняем задачу

В предыдущем проекте мы управляли микроконтроллером с компьютера, используя для этого очень простой протокол. Давайте попробуем добавить возможность игры для 4 сторон квадрата.

1) Добавьте «ракетку» не только на боковые стороны, но и на верхнюю и нижнюю
2) Добавьте возможность управлять каждой из ракеток «одному из игроков»
3) Добавьте возможность для 4 игроков, чтобы каждый управлял своей «ракеткой»

***

4) Реализуйте возможность игры по сети. 1 команда заходит с одного компьютера, другая с другого, и они играют в 1 игру – команда на команду.