PJRC Store

- Teensy 4.1, $26.85
- Teensy 4.0, $19.95
- Teensy 3.6, $29.25
- Teensy 3.5, $24.25
- Teensy 3.2, $19.80
- Teensy LC, $11.65
- Teensy 2.0, $16.00
- Teensy++ 2.0, $24.00

Teensy

- Main Page
- ⊞ **Hardware**
- ⊞ **Getting Started**
- ⊞ **Tutorial**
- ⊞ **How-To Tips**
- ⊞ **Code Library**
- Projects
- ▶ **Teensyduino**
  - Main
  - Download+Install
  - Basic Usage
  - Digital I/O
  - PWM & Tone
  - ⊞ **Timing**
  - ▶ USB Serial
  - USB Keyboard
  - USB Mouse
  - USB Joystick
  - USB MIDI
  - USB Flight Sim
  - Serial
  - ⊞ **Libraries**
- ⊞ **Reference**

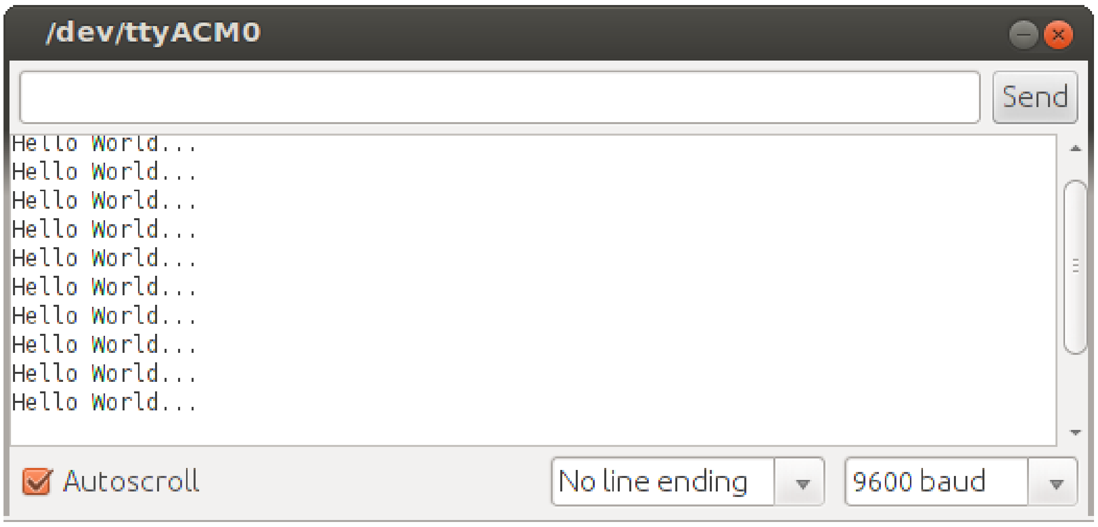
Using USB Serial Communication

Teensyduino provides a Serial object which is compatible with the Serial object on standard Arduino boards. Usually Serial is used to print information to the Arduino IDE Serial Monitor.

```
void setup()
{
  Serial.begin(9600); // USB is always 12 Mbit/sec
}

void loop()
{
  Serial.println("Hello World...");
  delay(1000);  // do not print too fast!
}
```

Unlike a standard Arduino, the Teensy Serial object always communicates at 12 Mbit/sec USB speed. Many computes, especially older Macs, can not update the serial monitor window if there is no delay to limit the speed! In this example, "Hello World..." is printed once per second.



```
Hello World...
Hello World...
Hello World...
Hello World...
Hello World...
Hello World...
Hello World...
Hello World...
Hello World...
Hello World...
```

The Teensy does not actually become a serial device until your sketch is running, so you must select the serial port (Tools -> Serial Port menu) after your sketch begins.

Standard Serial Functions

All of the standard Serial functions are supported.

Serial.begin()

Initialize the Serial object. The baud rate is ignored and communication always occurs at full USB speed.

Serial.print() and Serial.println()

Print a number or string. Serial.print() prints only the number or string, and Serial.println() prints it with a newline character.

```
// Serial.print() can print many different types
int number = 1234;
Serial.println("string");     // string
Serial.println('a');          // single character
Serial.println(number);       // number (base 10 if 16 or 32 bit)
Serial.println(number, DEC);  // number, base 10 (default)
Serial.println(number, HEX);  // number, base 16/hexidecimal
Serial.println(number, OCT);  // number, base 8/octal
Serial.println(number, BIN);  // number, base 2/binary
Serial.println(number, BYTE); // number, as a single byte
Serial.println(3.14);         // number in floating point, 2 digits
```

On a standard Arduino, this function waits while the data is transmitted. With Teensyduino, Serial.print() and Serial.println() typically return quickly when the message fits within the USB buffers. See Transmit Buffering below.

Serial.write()

Transmit a byte. You can also use Serial.write(buffer, length) to send more than one byte at a time, for very fast and efficient data transmission.

Serial.available()

Returns the number of received bytes available to read, or zero if nothing has been received.

On a standard Arduino, Serial.available() tends to report individual bytes, whereas large blocks can become instantly available with Teensyduino. See Receive Buffering below for details.

Usually the return value from Serial.available() should be tested as a boolean, either there is or is not data available. Only the bytes available from the most recently received USB packet are visible. See Inefficient Single Byte USB Packets below for details.

Serial.read()

Read 1 byte (0 to 255), if available, or -1 if nothing available. Normally Serial.read() is used after Serial.available(). For example:

```
if (Serial.available()) {
  incomingByte = Serial.read();  // will not be -1
  // actually do something with incomingByte
}
```

Serial.flush()

Wait for any transmitted data still in buffers to actually transmit. If no data is waiting in a buffer to transmit, flush() returns immediately.

Arduino 0022 & 0023: flush() discards any received data that has not been read.

Teensy USB Serial Extensions

Teensyduino provides extensions to the standard Arduino Serial object, so you can access USB-specific features.

Serial.send_now()

Transmit any buffered data as soon as possible. See Transmit Buffering below.

Serial.dtr()

Read the DTR signal state. By default, DTR is low when no software has the serial device open, and it goes high when a program opens the port. Some programs override this behavior, but for normal software you can use DTR to know when a program is using the serial port.

```
void loop()
{
  pinMode(6, OUTPUT);      // LED to show if a program is using serial port
  digitalWrite(6, HIGH);   //   (active low signal, HIGH = LED off)
  while (!Serial.dtr()) ;  // wait for user to start the serial monitor
  digitalWrite(6, LOW);
  delay(250);
  Serial.println("Hi there, new serial monitor session");
  while (Serial.dtr()) {   // wait for the user to quit the serial monitor
    Serial.print('.');
    delay(500);
  }
}
```

On a standard Arduino, the DTR and RTS signals are present on pins of the FTDI chip, but they are not connected to anything. You can solder wires between I/O pins and the FTDI chip if you need these signals.

Serial.rts()

Read the RTS signal state. USB includes flow control automatically, so you do not need to read this bit to know if the PC is ready to receive your data. No matter how fast you transmit, USB always manages buffers so all data is delivered reliably. However, you can cause excessive CPU usage by the receiving program is a GUI-based java application like the Arduino serial monitior!

For programs that use RTS to signal some useful information, you can read it with this function.

Serial.baud()

Read the baud rate setting from the PC or Mac. Communication is always performed at full USB speed. The baud rate is useful if you intend to make a USB to serial bridge, where you need to know what speed the PC intends the serial communication to use.

Serial.stopbits()

Read the stop bits setting from the PC or Mac. USB never uses stop bits.

Serial.paritytype()

Read the parity type setting from the PC or Mac. USB uses CRC checking on all bulk mode data packets and automatically retransmits corrupted data, so parity bits are never used.

Serial.numbits()

Read the number of bits setting from the PC or Mac. USB always communicates 8 bit bytes.

USB Buffering and Timing Details

Usually the Serial object is used to transmit and receive data without worrying about the finer timing details. It "just works" in most cases. But sometimes communication timing details are important, particularly transmitting to the PC.

Transmit Buffering

On a standard Arduino, when you transmit with Serial.print(), the bytes are transmitted slowly by the on-chip UART to a FTDI USB-serial converter chip. The UART buffers 2 bytes, so Serial.print() will return when all but the last 2 bytes have been sent to the FTDI converter chip, which in turn stores the bytes into its own USB buffer.

On a Teensy, Serial.print() writes directly into the USB buffer. If your entire message fits within the buffer, Serial.print() returns to your sketch very quickly.

Both Teensyduino and the FTDI chip hold a partially filled buffer in case you want to transmit more data. After a brief timeout, usually 8 or 16 ms on FTDI and 3 ms in Teensyduino, the partially filled buffer is scheduled to transmit on the USB.

The FTDI chip can be made to immediately schedule a partially filled buffer by toggling any of the handshake lines (which are not connected on a standard Arduino board). It can also be configured (by the PC device driver) to schedule when an "event character" is received. Normally it is difficult to control when the FTDI chip schedules its partially filled transmit buffer.

Teensyduino immediately schedules any partially filled buffer to transmit when the Serial.send_now() function is called.

All USB bandwidth is managed by the host controller chip in your PC or Mac. When a full or partially filled buffer is ready to transmit, it actual transmission occurs when the host controller allows. Usually this host controller chip requests any scheduled transfers 1000 times per second, so typically actual transmission occurs within 0 to 1 ms after the buffer is scheduled to transmit. If other devices are using a lot of USB bandwidth, priority is given to "interrupt" (keyboard, mouse, etc) and "isychronous" (video, audio, etc) type transfers.

When the host controller receives the data, the operating system then schedules the receiving program to run. On Linux, serial ports opened with the "low latency" option are awakened quickly, others usually wait until a normal "tick" to run. Windows and MacOS likely add process scheduling delays. Complex runtime environments (eg, Java) can also add substantial delay.

Receive Buffering

When the PC transmits, usually the host controller will send at least the first USB packet within the next 1ms. Both Teensyduino and the FTDI chip on Arduino receive a full USB packet (and verify its CRC check). The FTDI chip then sends the data to a standard Arduino via slow serial. A sketch repetitively calling Serial.available() and Serial.read() will tend to see each byte, then many calls to Serial.availble() will return false until the next byte arrives via the serial communication.

```
void loop()
{
  // On a serial-based Arduino, bytes tend to arrive one at a time
  // so this while loop usually only processes 1 byte before allowing
  // the rest of the loop function to do its work
  while (Serial.available()) {
    incomingByte = Serial.read();
    // do something with incomingByte
  }
  // do other unrelated stuff
  time_sensitive_task1();
  another_urgent_thing();
  still_yet_even_more_stuff();
}
```

On a Teensy, the entire packet, up to 64 bytes, becomes available all at once. Sketches that do other work while receiving data might depend on slow reception behavior, where successive calls to Serial.available() are very unlikely to return true. On a Teensy receiving large amounts of data, it may be necessary to add a variable to count the number of bytes processed and limit the delay before other important work must be done.

```
void loop()
{
  // On a Teensy, large groups of bytes tend to arrive all at once.
  // This bytecount prevents taking too much time processing them.
  unsigned char bytecount = 0;
  while (Serial.available() && bytecount < 10) {
    incomingByte = Serial.read();
    // do something with incomingByte
    bytecount++;
  }
  // do other unrelated stuff
  time_sensitive_task1();
  another_urgent_thing();
  still_yet_even_more_stuff();
}
```

Inefficient Single Byte USB Packets

When transmitting, Serial.write() and Serial.print() group bytes from successive writes together into USB packets, to make best possible use of USB bandwidth. You can override this behavior using Serial.send_now(), but by default multiple writes are merged to form packets.

Microsoft Windows and Linux unfortunately do NOT provide a similar function when transmitting data. If an application writes inefficiently, such as a single byte at a time, each byte is sent in a single USB packet (which could have held 64 bytes). While this makes poor use of USB bandwidth, a larger concern is how this affects buffering as seen by Serial.available().

The USB hardware present in Teensy can buffer 2 USB packets. Serial.available() reports the number of bytes that are unread from the first packet only. If the packet contains only 1 byte, Serial.available() will return 1, regardless of how many bytes may be present in the 2nd package, or how many bytes may be waiting in more packets still buffered by the PC's USB host controller.

This code will not work on Teensy when the PC transmits the expected 11 byte message in more than 1 USB packet.

```
// This may not work on Teensy if USB packets have less than 11 bytes!
boolean getNumbersFromSerial() {
  while (Serial.available() < 11) { ;} // wait for an 11 byte message
  if (Serial.read() == '@') {
    time_t pctime = 0;
    for(int i=0; i < 10; i++) {
      char c = Serial.read();
      if (c >= '0' && c <= '9') {
        pctime = (10 * pctime) + (c - '0') ; // convert digits to a number
      }
    }
    pctime += 10;
    DateTime.sync(pctime);   // Sync clock to the time received
    return true;   // return true if time message received
  }
  return false;  //if no message return false
}
```

This code can be rewritten to always read a byte when Serial.available() returns non-zero.

```
// This will always work on Teensy, does not depend on buffer size
boolean getNumbersFromSerial() {
  int count=0;
  char buf[11];
  while (count < 11) {
    if (Serial.available()) {  // receive all 11 bytes into "buf"
      buf[count++] = Serial.read();
    }
  }
  if (buf[0] == '@') {
    time_t pctime = 0;
    for(int i=1; i < 11; i++) {
      char c = buf[i];
      if (c >= '0' && c <= '9') {
        pctime = (10 * pctime) + (c - '0') ; // convert digits to a number
      }
    }
    pctime += 10;
    DateTime.sync(pctime);   // Sync clock to the time received
    return true;   // return true if time message received
  }
  return false;  //if no message return false
}
```

Of course, there are always many ways to write a program. The above versions look for a '@' character to begin the message, but do not handle the case where additional bytes (incorrectly) appear before the 10 digit number. It is also not necessary to store the entire message in a buffer, since the work can be done as the bytes are read. Here is a more robust and more efficient version.

```
// This version is more robust
boolean getNumbersFromSerial() {
  int count = 0;
  time_t pctime = 0;
  while (count < 10) {
    if (Serial.available()) {
    char c = Serial.read();
      if (c == '@') {
        pctime = 0;  // always begin anew when '@' received
        count = 0;
      } else {
        if (c >= '0' && c <= '9') {
          pctime = (10 * pctime) + (c - '0') ; // convert digits to a number
          count++;
        }
      }
    }
  }
  DateTime.sync(pctime);   // Sync clock to the time received
}
```