Overview of Mainstream Serial Protocols

By Tyler Lee
Version 1.4
Last Updated: 12/1/17

Abstract

This paper discusses the various serial protocols utilized by the majority of embedded devices. This report includes the various coding implementations necessary for the serial protocols discussed as well as further explanation on the tools and resources used for proper programming, analysis, and debugging of connected embedded devices.

Index Terms

Serial, communication, sub-systems, connected devices, satellite, Linux, ARM, protocol, screen, SSH, SCP, Half Duplex, Full Duplex UART, I2C, USB, Ethernet, RS-232, console, embedded devices, low-level, demonstration, code, common issues, problems, connections, usage, nmap

Table of Contents

Introduction

The purpose of this report is to describe and demonstrate various serial protocols common in embedded applications. The protocols are broken up into several sections to help clarify the functionality and as follows:

- Overview: an in-depth description of the protocol
- Connections: the connections required to implement the protocol
- Common Problems and Issues: a list of common problems encountered when utilizing the protocol
- Demonstration Code: code to exemplify the simplest approach to understanding the protocol

Terminology

- Master: controlling device on the network, dictates commands to slave devices
- Slave: device being controlled on the network, can transmit data back to master but at a lower priority on the bus
- Full Duplex: transmission can occur bi-directionally simultaneously
- Half Duplex: transmission can only occur in one direction at any given moment, for devices to communicate back and forth they must wait their turn
- Packet Switched: data is sent over the network in packets containing a destination address, allowing devices to share a connection bus
- Single Ended: signals connected using two terminals, one as an input and the other grounded as a reference voltage for the signal, cheapest and easiest methodology
- Differential: two complimentary signals with the actual signal being the difference between the two signals, allowing for noise compensation
- Baud rate: the speed in bits per second at which data is transmitted over a line, largely adjustable depending on the devices being utilized
- Console: a text-based application used to monitor and control a computer system which is typically remotely accessible

Usage Guidelines

This page compares the various protocols depicted in this report using several key parameters. The goal is to help depict the optimal scenarios for using each protocol. The parameters being compared are as follows:

- Transmission Type: Half duplex is single direction data flow at any given time, full duplex allows simultaneous bi-directional data flow
- Device Hierarchy: The organization of devices in the network, connected pair simply means only two devices are directly connected, master/slave hierarchies allow masters to send/receive messages from all the connected slaves
- Data Transmission Method: A physical connection means the devices respond to a signal immediately as the connection results in the completion of an electrical circuit, while a packet switched network sends groups of data which can then be sent to a specific address which translates to a device on the network
- Input Signal Type: Signals can be sent either in reference to a ground (usually common to all devices) with incoming data being the voltage, or by sending two signals and calculating the difference; the former is single ended, the latter is differential

**For more details see the terminology section of the introduction*

Protocol	Transmission Type	Device Hierarchy	Data Transmission Method	Input Signal Type
UART	Full Duplex	Connected Pair	Physical Circuit	Single Ended
I2C	Half Duplex	Multi-Master/ Multi-Slave	Packet Switched	Single Ended
Ethernet	Full Duplex	Single Master/ Multi Slave	Packet Switched	Single Ended
RS-232	Full Duplex	Connected Pair	Physical Circuit	Single Ended

Color Key	Best	Ok	Worst

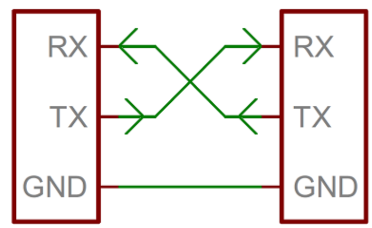
UART

Overview

UART ("Universal Asynchronous Receiver Transmitter") is the simplest of serial connections, implemented primarily for connecting two devices of nearly any type with an adjustable speed. The design simply consists of two data wires connecting the devices, RX for receiving and TX for transmission. The interface calls for both devices to communicate with the baud rates set and equivalent. This rate can be adjusted to transmit data at various speeds with common baud rates being 9600 for low level devices like Arduinos, and 115200 for more mainstream computers.

Connections

Master	Slave
RX	RX
TX	TX
GND	GND



Common Problems and Issues

- Switching RX and TX: extremely common mistake, the master transmitting pin and the receiver receiving pin need to be connected, visa versa for master receiving pin
- Baud rate mismatch: if you are attempting to read a device and getting weird characters as an input, you likely have the devices communicating at different speeds, resulting in incorrect interpretation. This can be fixed by setting the baud rates to the lower of the two devices.

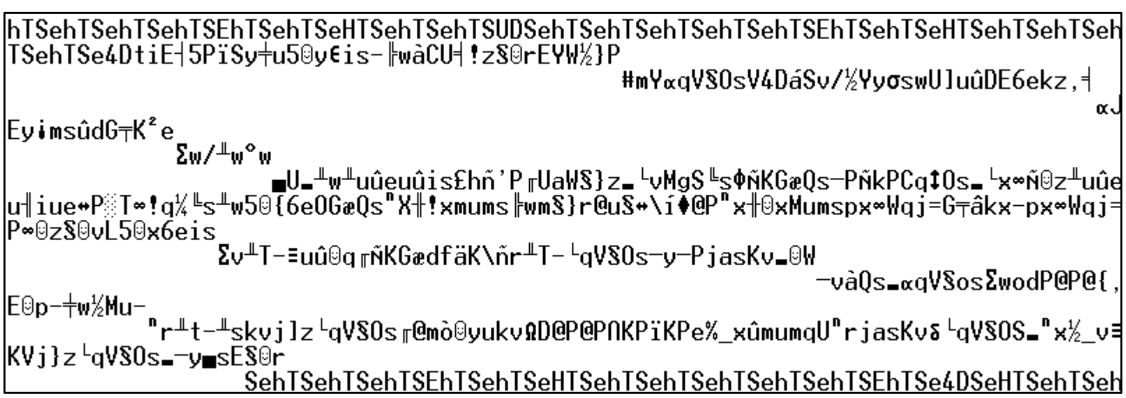


Figure: Console output showing a baud rate mismatch

Demonstration Code

Master – Transmit/Receive	Source Code: Python	Raspberry Pi/ Beaglebone Black

```
import serial

print("Program Running")

# initialization UART interface
serialConnection = serial.Serial("/dev/ttyAMA0")

if serialConnection.isOpen(): # ensure serial connection is ready
    print("Sending on UART connection...")
    while True: # runs in real time
        userInput = input("Transmit: ") # obtain user input
        serialConnection.write(userInput) # transmit

print("Program Ending")
```

Slave – Transmit/Receive	Source Code: Python	Raspberry Pi/ Beaglebone Black

```
import Adafruit_BBIO.UART as UART
import serial

print("Program running...")

# initialization of pins and UART interface
UART.setup("UART1")
serialConnection = serial.Serial("/dev/ttyO1")

print("Reading UART connection...")

while True: # runs in real time
    character = serialConnection.read() # obtain input from connection
    if (character != ""): # check to see if there is an actual input
        print ("Received: " + character)

print("Program ending.")
```

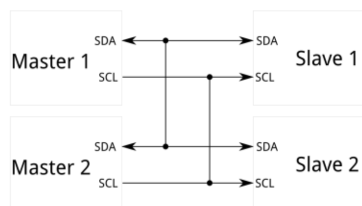
I2C

Overview

I2C ("I Squared C") was a protocol developed to allow a broad array of devices to connect to a single master device (occasionally more) at a high speed using minimal lines. Since the master can easily address up to 114 devices individually on a shared bus it is much more useful than UART beyond two devices. The single and shared Data and Clock wires also reduced the cable clutter of individual wires connecting each slave to the master in protocols such as SPI. Finally, I2C is typically implemented using only jumper cables making it far less bulky than a RS-232 connector. Overall, these advantages make I2C one of the most useful and common protocols implemented in embedded applications.

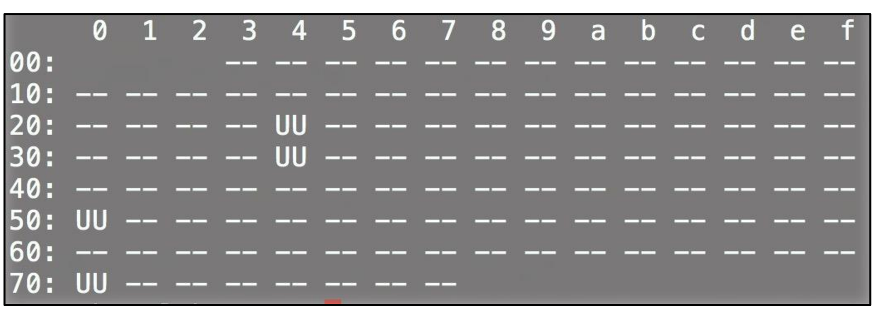
Connections

Master	Slave (1)	Slave (2)
Data	Data	Data
Clock	Clock	Clock
GND	GND	GND



Common Problems and Issues

- Incorrect address: use the **`i2cdetect`** command in the terminal to ensure your device is properly connected. This can be installed via **`sudo apt install i2c-tools`** on Debian-based Linux systems.
- Due to current limitations on the Linux kernel, embedded Linux devices cannot be I2C slave devices.



```
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- UU -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- UU -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: UU -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: UU -- -- -- -- -- -- --
```

Figure: Command line output of **`i2cdetect`** command showing the 114 possible output devices addresses. No device is currently connected.

Demonstration Code

Master – Transmit/Receive	Source Code: Python	Raspberry Pi/ Beaglebone Black

```
import smbus
import time

bus = smbus.SMBus(1) # instantiate bus
slaveAddress = 0x04 # assign a slave address

def writeNumber(value): # function to write value
    bus.write_byte(slaveAddress, value)

def readNumber(): # function to read value
    number = bus.read_byte(slaveAddress)
    return number

while True: # continuous loop
    var = input("Enter 1 – 9: ") # get user value to write
    if not var: # quick error check
        continue

    writeNumber(var) # write value
    print "RPI: Hi Arduino, I sent you ", var
    time.sleep(1) # wait one second

    number = readNumber() # read value
    print "Arduino: Hey RPI, I received a digit ", number
    print # spacing
```

**Code modified from Oscar Liang Tutorial [1]*

Slave – Transmit/Receive	Source Code: C++	Arduino Uno/Nano/ Mega/Due

```
#include <Wire.h>

#define SLAVE_ADDRESS 0x04 // initialize slave address
int number = 0; // initialize global data value

void setup() {
   Serial.begin(9600); // initialize serial port with baud rate
   Wire.begin(SLAVE_ADDRESS); // initialize slave address

   Wire.onReceive(receiveData); // define callbacks for i2c communication
   Wire.onRequest(sendData);
}

void loop() {
   delay(100); // for stability
}

void receiveData(int byteCount) { // callback for received data
   while(Wire.available()) {
      number = Wire.read();
      Serial.print("data received: ");
      Serial.println(number);
   }
}

void sendData() { // callback for sending data
   Wire.write(number);
}
```

**Code modified from Oscar Liang Tutorial [1]*

Ethernet

Overview

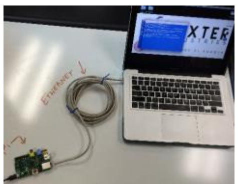
Ethernet is most commonly known for being the standard connection implemented by the internet, at least from an end user perspective. Connections are packet-switched meaning the data is broken up into smaller chunk before being sent to alleviate traffic. An Ethernet network can be expanded via switches making it highly versatile for larger scale applications and larger devices. Modern standards also have a high bandwidth for data pass through making it ideal for file transfers and remote connections to a console.

Ethernet communication on such networks require devices to have large addresses which are known as IP addresses. There are two mainstream sets of addresses, IPv4 and IPv6 with IPv6 fixing issues and the limited address space of IPv4. Despite the advancement, IPv4 addresses are still the de facto standard. IP address are set up by the controller of the network, typically a router in a home setting, the internet service provider in a community setting, and simply a computer in an embedded setting. When a new device is connected to a network, its hardware encoded MAC address (non-changing) becomes known to the controller and the device is assigned an IP address. A good way to think about this is to envision the MAC address as your physical mailbox and an IP address as your home address. Your home address is assigned to you by the government and can change if you move but your physical mailbox doesn’t change.

For the purposes of this report, the use case described for Ethernet in embedded applications is to access the console of the embedded device in order to send commands and files to the low-level devices. The bulkiness and power consumption of the hardware to create a network means it’s not typically implemented in low level systems, however, the functionality lends itself highly to setting up and monitoring systems during the design and building phases.

Connections

Figure: RJ45 port with indicator LEDs typically used for Ethernet

Direct Connection	Repeater	Switch	Router
Directly connected two devices, typically creating a DHCP server on one of the devices to assign IP addresses	A device used to boost the signal strength, used for long distance connections	A device that switches all input from any of the inputs to all other inputs with no concept of IP addresses	A device that assigns IP addresses to connected devices and only sends packets to the intended recipient
			

Common Problems and Issues

- No actual connection: check the orange and green indicator lights on the devices to see if there is a physical connection. If there is, use the **`ping`** command to test whether the devices are visible on the network.
- Connection Refused Over USB: Many embedded devices such as the Beaglebone Black can implement Ethernet over USB. If the connection doesn't exist or is being refused, check to ensure the proper drivers for Ethernet over USB are installed on your master device.
- Unknown IP address: use the **`nmap`** command to show the connected devices on the network using the following syntax:

  `nmap –sP <master device ip>/24`

  `nmap` can be installed via **`sudo apt install nmap`** `on Debian-based Linux systems.`

```
Tyler@Tylers-MacBook-Pro-3:~$ nmap -sP 192.168.7.1/24

Starting Nmap 7.60 ( https://nmap.org ) at 2017-12-05 21:39 EST
Nmap scan report for 192.168.7.1
Host is up (0.00063s latency).
Nmap scan report for 192.168.7.2
Host is up (0.0011s latency).
Nmap done: 256 IP addresses (2 hosts up) scanned in 2.98 seconds
Tyler@Tylers-MacBook-Pro-3:~$
```

Figure: example nmap command output to find the embedded system IP address

Demonstration Code

Master – Connect/Control	Source Code: Bash Terminal Input	Linux PC/Mac

```
Tyler@Tylers-MacBook-Pro-3:~$ uname -a
Darwin Tylers-MacBook-Pro-3.local 16.7.0 Darwin Kernel Version 16.7.0: Wed Oct  4 00:17:00 PDT
2017; root:xnu-3789.71.6~1/RELEASE_X86_64 x86_64
Tyler@Tylers-MacBook-Pro-3:~$ ssh debian@192.168.7.2
debian@192.168.7.2's password:
Linux beaglebone 4.9.45-ti-r57 #1 SMP PREEMPT Fri Aug 25 22:58:38 UTC 2017 armv7l

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Thu Aug 31 16:56:58 2017 from 192.168.7.1
debian@beaglebone:~$ uname -a
Linux beaglebone 4.9.45-ti-r57 #1 SMP PREEMPT Fri Aug 25 22:58:38 UTC 2017 armv7l GNU/Linux
debian@beaglebone:~$
```

**An assumption is made that the controlling device is on the same network as the embedded devices being controlled and the IP addresses known*

Master – File Transfer	Source Code: Bash Terminal Input	Linux PC/Mac

```
Tyler@Tylers-MacBook-Pro-3:~/demo$ ls
test.file
Tyler@Tylers-MacBook-Pro-3:~/demo$ scp test.file debian@192.168.7.2:~/demo/
debian@192.168.7.2's password:
test.file                    100%    0     0.0KB/s   00:00
Tyler@Tylers-MacBook-Pro-3:~/demo$ ssh debian@192.168.7.2
debian@192.168.7.2's password:
Linux beaglebone 4.9.45-ti-r57 #1 SMP PREEMPT Fri Aug 25 22:58:38 UTC 2017 armv7l

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Thu Aug 31 17:02:58 2017 from 192.168.7.1
debian@beaglebone:~$ cd demo
debian@beaglebone:~/demo$ ls
test.file
debian@beaglebone:~/demo$
```

**An assumption is made that the controlling device is on the same network as the embedded devices being controlled and the IP addresses known*

RS-232

Overview

RS-232 is similar in most ways to UART with the major difference being the cable implementation standard. RS-232 utilizes either a DB-25 or the more common DB-9 connection utilized today instead of jumper cables. This cable can easily be adapted to USB and plugged into a master device.

Connections

DB-9	DB-25	DE-15
		

**DE-15 cable shown for reference as it is the standard cable for VGA monitors*

Common Problems and Issues

- No visible console output: press a character and hit enter to see if you are indeed at the console but the input is hidden
- Screen won't attach to console: use the following command to list screen sessions

  ```
  screen -ls
  ```

  and then run the following command to kill the screen session

  ```
  screen –X –S <session #> quit
  ```

  where session # is the number proceeding the session you want to kill given from the first command

```
Tyler@Tylers-MacBook-Pro-3:~$ screen /dev/tty.usbserial-A104S1N4 115200
[screen is terminating]
Tyler@Tylers-MacBook-Pro-3:~$ screen -ls
There is a screen on:
        11938.ttys000.Tylers-MacBook-Pro-3     (Detached)
1 Socket in /var/folders/hd/_5wv4gh94_30n57t1591mj7m0000gp/T/.screen.

Tyler@Tylers-MacBook-Pro-3:~$ screen -X -S 11938 quit
Tyler@Tylers-MacBook-Pro-3:~$ screen -ls
No Sockets found in
/var/folders/hd/_5wv4gh94_30n57t1591mj7m0000gp/T/.screen.

Tyler@Tylers-MacBook-Pro-3:~$
```

Figure: demonstration of quitting a problematic screen session by identifying the session and sending the quit signal

Demonstration Code

Master – Connect/Control	Source Code: Bash Terminal Input	Linux PC/Mac

```
Tyler@Tylers-MacBook-Pro-3:~$ screen /dev/tty.usbserial-A104S1N4 115200

Welcome to System Board - Linux

System SW Release v2.01
System HW Revision v5 Min

Built: Fri Jan 22 15:06:42 EST 2016

****************************************************************************
****** WARNING: initrd and NAND version mismatch.  Reflash NAND ******
system login: root
Password:
# uname -a
Linux system 1.3.40.2 #1 Fri Jan 22 14:02:17 EST 2016 GNU/Linux
#
```

Sources

[1] https://oscarliang.com/raspberry-pi-arduino-connected-i2c/