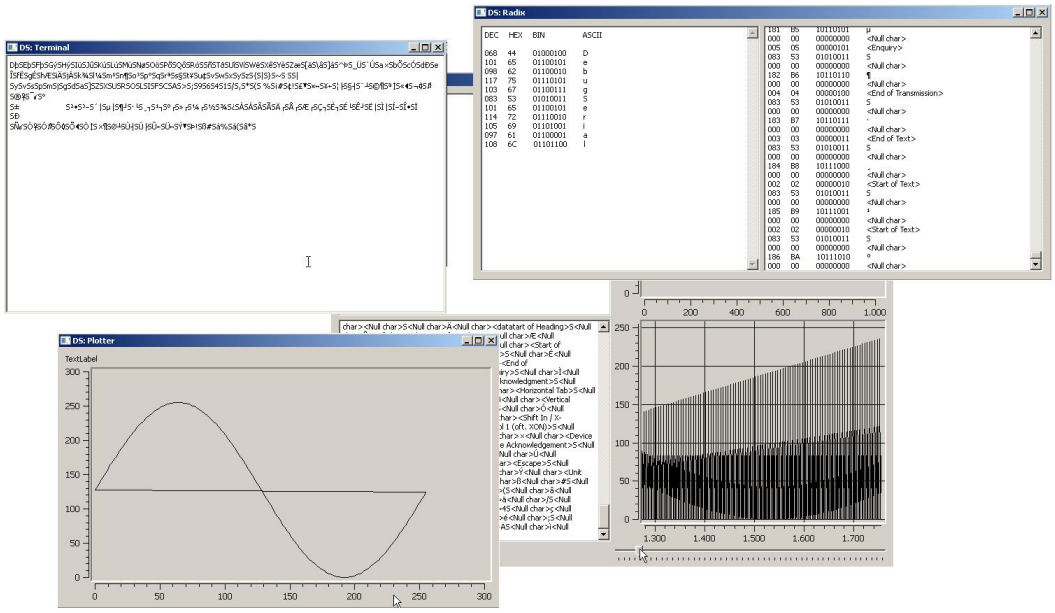
DebugSerial
User Manual

Ashey J. Robinson
ashley181291@gmail.com

10th September, 2014



Abstract

DebugSerial is a tool for debugging embedded software using a simple serial connection. Only a UART with eight bits of data and a single stop bit is permitted but varying baud rates are allowed. Motivation for this tool is to increase design potential when only simple hardware is available. UART protocol is slow but is possible to setup real-time graphs and other interesting analysis techniques. This software relieves the embedded systems engineer from creating higher level protocols for sending data which can be time consuming when managing both ends of the connections. Protocols are defined and a rich GUI is provided so debugging using serial is simple from hardware all the way to visualisation.

Contents

1 Introduction

1.1 Hardware

1.2 Getting Started

`DebugSerial`is invoked as a python script from a terminal window. `DebugSerial`will immediately search the environment for all available serial ports. Failure to find any serial ports will cause `DebugSerial`to exit. If only one serial port is available this will be connected to automatically. When more than one serial port is found a choice will presented. User choices persist between program executions.

When a serial port is selected a baud rate must be chosen. Any value can be taken but this may be limited by chosen hardware.

2 Terminal

Terminal mode is the most basic data handling tool available in `DebugSerial`. An keyboard input the can be taken as an ASCII value if piped down the chosen serial port. Returning data is translated from raw form to the equivalent ASCII character and printed in the terminal. As an example connecting the transmit and receive end of a serial port together will create a text editor like interface that will filter any non-ASCII keys from being printed. Some ASCII characters cannot be visualised easily within a terminal environment so these are replaced by the name of the representation encase by angle brackets; ¡bell¿, ¡alt¿, etc.

3 Radix

When debuggging over a serial connection ASCII representations may be of no concern. In this case the Radix option within `DebugSerial`can be chosen chich display differemt representation of recieved and transmistted data. Data is displated in binary, hexadecimal, ASCII and decimal representations. As the throughput of data is so large it suggestted to use this option in low volume debugging situations.

4 Graph

In the case when data throughput is too large to be analysed by text then visualisation can be used to view raw values. The `DebugSerial`Graph options can be selected. This is a biditectional interface that creates transmission and reception domain raw data plots. Each direction of traffic has a terminal interface paired with the corresponding grapgh.

5 Plotter

Advanced graphing cannot be achieved in the transmission domain so the `DebugSerial`Plotter option allows real-time graphing. This is a receive only option and a higher level protocol must be adhered to in order to creATE THE GRAPHS.

5.1 Protocol

Initially a command byte is sent followed by as many as 8 data bytes. There is a maximum of 4 bytes for each axis allowing for a range of 2^{32} discrete values. Figure 1 and Table 1 describes how this byte can control plotting. Bit 0, the least significant bit, and bit 1 remain fixed at logical high. Bits 2 and 3 control plotting where 00 adds a point, 01 removes a point and 11 will erase the whole plot. The top 4 bits control the trailing data bytes.

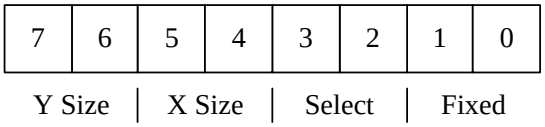


Figure 1: Command byte explained.

The state transition diagram in Figure 2 explains how the command byte controls the data representation. A clear command and any other command bytes not listed in Table 1 will keep the state machine in the `CMD` state. Valid command bytes will move the state machine on the receive data. First the x-axis data is received followed by the y-axis data. The first byte in each axis data transmission becomes the most significant byte for that axis. The three examples in Figure 3 explains how the transmission order is used to shift bytes left in the final representation.

CMD Byte	**Function**	**Y**	**X**
00 00 0? 11	Add(0) or Remove(1) point.	1	1
00 01 0? 11	Add(0) or Remove(1) point.	1	2
00 10 0? 11	Add(0) or Remove(1) point.	1	3
00 11 0? 11	Add(0) or Remove(1) point.	1	4
01 00 0? 11	Add(0) or Remove(1) point.	2	1
01 01 0? 11	Add(0) or Remove(1) point.	2	2
01 10 0? 11	Add(0) or Remove(1) point.	2	3
01 11 0? 11	Add(0) or Remove(1) point.	2	4
10 00 0? 11	Add(0) or Remove(1) point.	3	1
10 01 0? 11	Add(0) or Remove(1) point.	3	2
10 10 0? 11	Add(0) or Remove(1) point.	3	3
10 11 0? 11	Add(0) or Remove(1) point.	3	4
11 00 0? 11	Add(0) or Remove(1) point.	4	1
11 01 0? 11	Add(0) or Remove(1) point.	4	2
11 10 0? 11	Add(0) or Remove(1) point.	4	3
11 11 0? 11	Add(0) or Remove(1) point.	4	4
xx xx 11 11	Clear entire plot.	0	0

Table 1: Command byte options.

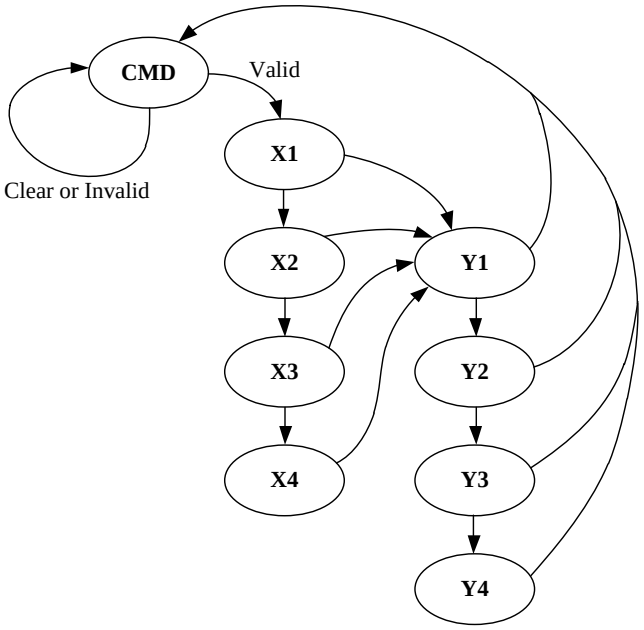


Figure 2: Plotter state transition diagram.

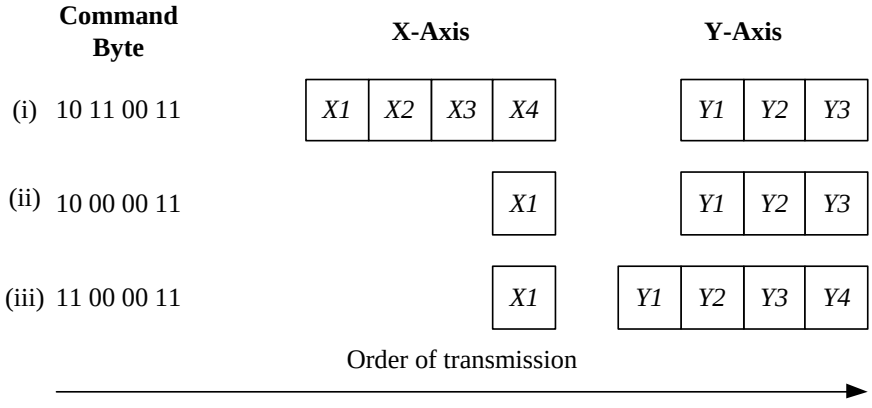


Figure 3: Three example transmissions.

6 Image

This option allows image to visualised but pushes the limit of serial band width. Small greyscale image can be easily transmitted but larger images require higher bandwidth technology which falls beyond the scope of this software.

7 Radix

When debuggging over a serial connection ASCII representations may be of no concern. In this case the Radix option within `DebugSerial`can be chosen chich display differemt representation of recieved and transmistted data. Data is displated in binary, hexadecimal, ASCII and decimal representations. As the throughput of data is so large it suggestted to use this option in low volume debugging situations.