USB HOST CONTROLLER FOR A MICROCONTROLLER FOR USE IN ECE 4760

A Design Project Report

Presented to the School of Electrical and Computer Engineering of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering, Electrical and Computer Engineering

Submitted by

Young Hwa Kim

MEng Field Advisor: Dr. Bruce Land

Degree Date: January 2013

Abstract

Master of Engineering Program

School of Electrical and Computer Engineering

Cornell University

Design Project Report

Project Title: USB Host Controller for a Microcontroller for Use in ECE 4760

Author: Young Hwa Kim

Abstract:

Using a microcontroller as a USB host device instead of as a USB peripheral device can be very helpful for students in ECE 4760 (Digital Systems Design Using Microcontrollers) to attach a USB mouse, a USB keyboard or a USB memory stick. Although a full software version of a USB host has been previously implemented on AVR Mega32 microcontrollers as a final project for ECE 4760 by students, the fact that it heavily loads the microcontroller forces us to look for an alternative solution, in other words, a hardware implementation. For example, dedicated host chips such as VNC1L and MAX3421 provide high-speed interface to unload the microcontroller. I implemented a useable USB host interface to Mega1284 using a chip (VNC1L on VDIP1 module) that unloads the microcontroller. The final product includes the libraries that consist of APIs that a host uses to communicate with HID (human interface device) and mass storage class peripheral devices. The hardware implementation can be used by students in ECE 4760 to run a mouse, a keyboard and a memory stick. This will allow students to easily attach peripherals such as a mouse to their final project without having to use a host computer.

Executive Summary

This USB Host Controller for a Microcontroller project was proposed and is created specifically for the use in ECE 4760 class. The project is designed to create a useable USB host interface to Mega1284 using a dedicated chip, VNC1L, on its development module, VDIP1. Compared to a similar previous project the solution of which takes an approach in software, this project is done in hardware to unload the microcontroller. The first and direct beneficiaries of the final product of this project would be the students in ECE 4760 who are expected to design a creative and somewhat complicated final project.

The final product includes the libraries that consist of APIs that an MCU host simply calls to communicate with HID (human interface device) and mass storage class peripheral devices. What enables the mega1284 to be a USB host is the VNC1L chip that is pre-programed with its firmware. The serial communication part of this project between two chips was implemented both in SPI and USART to give students options to choose. All the tasks that are related to USB protocols are performed by the mediator, VNC1L, instead of mega1284. To send data to a slave USB device and to receive data from it, the MCU host simply makes appropriate API calls and communicates with the device through the command monitor interface of the VNC1L. The bulk of this project is getting familiar with the firmware, its monitor commands and their operations.

Mainly three libraries were created for three different types of popular USB devices: a flash drive, a keyboard and a mouse. Also each library was implemented twice: one with SPI and the other with USART interface. Each library comes with unique APIs in a firmware.c file where the appropriate firmware commands are sent to the monitor to control the actions of a device and the response data from a device is parsed to determine the status of events that happened within the device. With a VNC1L, a USB flash drive can be used for a microcontroller project to provide additional memory and eliminate the limit of memory resources available to an MCU. A data logging program could well utilize a USB drive to save enormous amount of data. A USB keyboard as a user input device can provide many more options than a pushbutton or a keypad with only a few keys available on it. A USB mouse can be a perfect choice as an input and control device for game projects or draw-and-paint projects.

Without the broad knowledge in the much complicated USB protocols, students in ECE 4760 will be able to easily include a USB peripheral device of their choice for their final projects, which is expected to help them expand the scope of their projects and give them more freedom to realize their creative design concepts.

Table of Contents

Introduction

ECE 4760 class teaches students how to design various digital systems using Atmel AVR ATmega microcontrollers such as Mega1284. A prototype board that has a type-B USB connector for serial communication is loaned to students at the beginning of the class. This allows students to have the USB connection for serial communication between their running program and the PC that runs PuTTY for debugging and taking user inputs for their programs. In this configuration, the PC is a USB host and the microcontroller takes the role of a slave device.

For their final projects, students in ECE 4760 class design and produce hardware and software combined systems that could require many different kinds of user interfaces. Previously students used items such as push buttons, keypads, microphones, and light detecting phototransistors. Although these items worked quite well as user input devices, many students could have benefited and made their programs more flexible or sophisticated if they could use an HID class device such as a USB keyboard or a USB mouse as a user input device. Also, adding a USB mass storage in a project could solve the problem of Mega1284's limited memory resources that hinders developing a memory intensive project.

From my experience as a previous student of the class, having an option to have a microcontroller as a host to control USB slave devices with a fast and easy interface can allow students to broaden the scope of the design of their final projects or come up with more creative systems. I believed choosing a right USB host controller IC chip that could be easily interfaced with Mega1284 and came with firmware fully supported by a manufacturer would be a key to the success of this design project. This consideration led us to choose a VDIP1 module with a VNC1L chip from FTDI (Future Technology Devices International Ltd.) as a winner.

Issues to be addressed

The previous project, "Software Implemented Atmel Mega32 Universal Serial Bus Host Controller" by Ben Hutton, Devrin Talen and Chris Leary, was a successful final project for the ECE 4760 class. However, with its shortcomings, it cannot fully address the needs of students who would like to use a USB device in their final projects without having to learn the much complicated USB protocols.

Below is the list of problematic issues that the previous software implementation carries:

- The previous project was implemented for Mega32, which is a much older chip with the less resources available compared to Mega1284, which is currently used in ECE 4760. Although I believe the libraries can be exported to be used by Mega1284, the project

implements only a low speed (1.5 Mbps) USB 2.0–compliant host controller and it needs to be updated.
- As the authors of the project state, the libraries were tested only with a USB mouse, and if a student wants to use other USB slave devices such as a USB keyboard or a USB flash drive, he or she needs to write “some sort of storage driver (such as FAT32)” themselves, which could be a daunting task for students who are not familiar with complex USB protocols. Students who want to use a USB device for their project should not need to have a comprehensive knowledge of complicated USB protocols. Preferably, an extensive library of API (application programming interface) functions for each popular USB class (HID and mass storage) should be provided for students so that their program could access USB slave devices by simply calling those API functions without having to know the details of what is done in a driver file.
- The example Mouse Driver code from the previous project shows that how communication between a USB host and a USB slave is done at a very low level, which is translated as a huge load on a microcontroller. For a microcontroller to be able to perform many other tasks than communicating with a USB slave device, it should be freed from such a huge load since the entire implementation was done in software. This fact calls for a hardware implementation of a USB host controller instead of a software implementation.

In summary, the limitations of the previous project mentioned above point to a hardware implementation of a USB host controller using a dedicated IC chip, which is VNC1L for this project.

Preliminary Research

To address the main problem of the previous project on implementing a USB host controller, we needed a dedicated host chip that could ease the load of Mega1284. According to *USB Embedded Hosts* by Jan Axelson[1], there are many hardware options available for implementing a USB host in an embedded system as shown in the table below.

System Type	Host Communications Support
Embedded PC with host controller	Linux or Windows API, other protocols supported by the OS and programming environment
General-purpose microcontroller with on-chip host controller	Libraries from chip provider

[1] Axelson, Jan. (2011). USB Embedded Hosts. Madison, WI: Lakeview Research LLC.

External host interface chip plus general-purpose microcontroller	Libraries from chip provider
Processor with on-chip host module	Vendor-specific API
Host module with interface to external processor	Vendor-specific command set
Processor with USB host and support for .NET Micro Framework and UST host communications	.NET Micro Framework classes

Table 1. Hardware options available for implementing a USB host in an embedded system

To make a decision on which IC chip would be the best for this project, first much research on those chips had to be carried out as well as studies on their datasheets, their compatibility to Mega1284 and support items that each vendor provides such as libraries, APIs and firmware.

The main purpose of this project is to serve the need of students who desire a simple way to utilize their microcontroller as a USB host so that they can employ a USB slave device in their final project without losing much of the resources or the computing power of the MCU to the processing of communication between a USB host and a slave. Considering that, a processor with an on-chip host module or a host module with interface to an external processor seemed to be the best choice for the project. FTDI (Future Technology Devices International) offers VNC1L Vinculum USB Host Controller IC, which can be programed with firmware fully developed by FTDI. The manufacturer also produces a single USB connector based development module VDIP1 for the VNC1L and a double USB connector base development module VDIP2 for the VNC1L. Maxim Integrated offers a single IC with USB functionality, MAX3421E USB Peripheral/Host Controller with SPI interface. Although MAX3421E provides a microcontroller-independent solution, no extensive firmware download for the chip is available from the vendor. Writing a device-specific firmware for a USB host controller is outside of the scope of this design project. Thus, VNC1L was the best choice for the project.

Once a decision was made on which IC was to be the hardware option for the project, studies on the libraries of the specific firmware for VNC1L and its command sets for various USB device classes were carried out. The expectation was that those firmware commands would take care of instantiating the low level interface between the USB host Mega1284 and its USB slave device such as an enumeration of the device, requesting and receiving various types of descriptors and addressing endpoints of the device.

Proposal for Development

Next step would be to write the API libraries for two different USB classes, HID and mass storage, which includes a USB keyboard, a USB mouse and a USB memory stick. Those classes use different transfer types to communicate with a host: HID class uses interrupt transfer while the mass storage class uses bulk transfer. Also, HID class uniquely uses “report” to transfer data. Thus, three sets of API libraries for each device should be written, and this would be the main task for the finished project.

Once the project was successfully completed, the deliverables were expected to be a ready-to-use USB interfacing solution that supported two common USB device classes. The library for a USB memory stick would provide APIs that read a file from a drive, write a file to a drive and append new texts to an existing file on a USB drive. The library for a USB keyboard would have APIs that can recognize a keystroke on a keyboard. The last library for a USB mouse would include APIs that detect a button press (left and right clicks) and a wheel movement on a mouse in terms of X and Y coordinates. With the complete sets of API libraries, students should be able to easily add a USB slave device to their projects. To help students better understand how to utilize the new libraries in their project, sample programs that employ API function calls from each library would be written to show how to interface a USB slave device to a host controller Mega1284.

Developing an API library for a USB printer would not be considered because many printers these days use a printer control language that is unique to themselves. Implementing supports for every possible printer control language and many different features of every printer would be outside of the scope of this project.

USB Protocol Basics

Although VNC1L does take care of all low-level housekeeping that is related to USB protocols for a microcontroller, understanding the basic terminology of USB protocol will help students who plan to use a VDIP1 module and APIs that allow communication between an MCU and VNC1L for their final projects. Thus, here a few of important USB interface are explained.

USB Transfers

A USB device uses a USB transfer to send and receive data for communication. Each transfer has a defined format for sending data, status and control information and more. There are four types of USB transfers: control, bulk, interrupt and isochronous transfers. All USB devices must support control transfer which is used for identification and configuration of a device. USB devices such as USB flash drives and printers use bulk transfer that does not guarantee latency while HID devices such as USB keyboards and USB mice use interrupt transfer that does guarantee low latency. Except for control transfer, interrupt transfer is the only way low-speed devices such as a USB mouse can transfer data.

Important elements of a USB transfer are called endpoints, and there is a pair of endpoints: IN and OUT. Once a transfer is scheduled, all data on a USB bus travels to or from a device endpoint. Simply put, endpoints are buffers that store bytes which are received or waiting to be transmitted. While a host device has buffers to hold received data and data waiting to be transmitted, only a slave device has device endpoints. Before any data can be exchanged between a host and a slave, they must establish a pipe. A pipe connects endpoints of a device to a host. Any transfer type is allowed to use IN and OUT transactions.

Enumeration

Once a USB slave device is connected to a USB host, the host needs to learn about the device before any application can communicate with the device. This process of initialization and exchanging information between a host and a slave device is called enumeration. During the enumeration, a host can assign an address to a device and read all kinds of descriptors from a device. Once the enumeration is completed, a slave device can be ready to transfer data to a host. USB descriptors are data structures that contain information about a device such as interface and endpoints of it. They enable a host to learn about a slave device, and all USB devices are required to respond to requests for the standard USB descriptors from a host. The good news is that VNC1L does execute the process of enumeration in the background for us as soon as it detects a device on its USB port.

Human Interface Device (HID) Report

It was mentioned above that HID devices such as a USB keyboard or a USB mouse use interrupt transfer to send data to a host through an IN interrupt endpoint (the direction of any data transfer is always from a host point of view). This data that a device sends to a host is contained in a report, which HID devices use to exchange data. A report descriptor contains information about the data that is sent and received between a host and a device; however, the descriptor does not include a report. HID class specific requests can be used instead to get a report from a device and send a report to a device. For example, a report from a USB keyboard can tell a host which key has been pressed, and a report from a USB host can turn on an LED for a NUM Lock on a keyboard. The Windows HID API provides a set of functions that applications can use to get and send a report. Conveniently, the firmware for VNC1L comes with a command that can do the same, which will be elaborated on later in this report.

More detailed information on the entire USB protocols can be obtained from documents such as *USB Serial Bus Specification* on USB Implementers Forum (USB-IF) website.

Vinculum VNC1L Embedded USB Host Controller IC

Hardware: VDIP1 module

VNC1L from FTDI is a single chip embedded dual USB host controller that features two independent USB 2.0 Low/Full-speed USB host ports. This chip handles entire USB protocols such as enumeration and various transfers of descriptors at IN and OUT endpoints. It comes with 64k bytes of embedded Flash (E-FLASH) memory to store firmware, which could be programed or updated via UART interface. VNC1L also provides options to interface to external Command Monitor via UART, SPI or FIFO slave interface. It does not require external software control because the free firmware takes care of it all, which is the best advantage of using a VNC1L chip for the project.

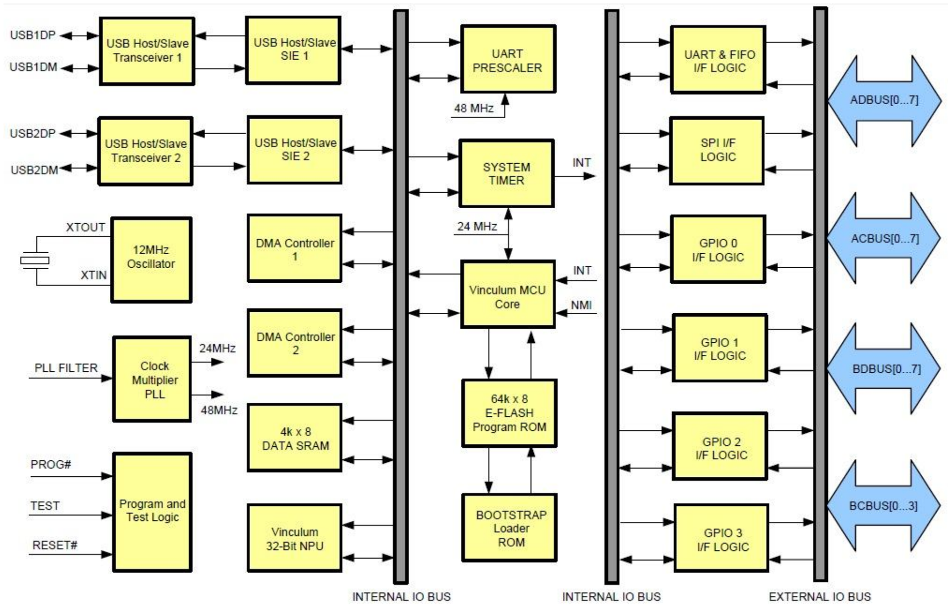


Figure 1. Simplified VNC1L Block Diagram[2]

[2] Excerpted from VNC1L datasheet on page 4, Figure 2.1.

The block diagram is provided here for anyone who wants to know what is inside of a VNC1L. However, after learning about Atmel mega1284 and studying its datasheet, students in ECE 4760 do not have to repeat the same time consuming process with a VNC1L if they choose to use it for their final projects. It is recommended that they read about basics of USB protocols that are explained in the previous section and learn how to use key commands for the firmware that a VCN1L is programmed with. The details on Vinculum Firmware will be explained in the next section.

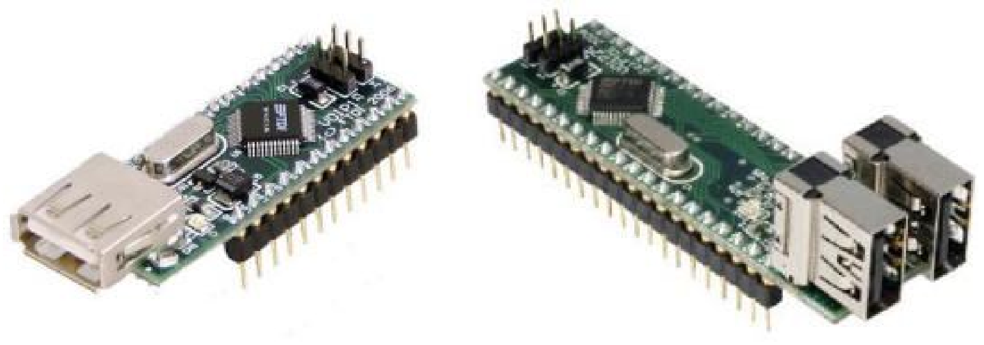


Figure 2. VDIP1 (left) and VDIP2 (right) with VNC1L[3]

Above are the pictures of two development modules of VNC1L. They are both under $25, which serves well the budget constraint for a final project in ECE 4760 class. The size of VDIP2 is 64.8mm by 17.78mm with the height of 25mm. Their compact size and relatively light weight will work well with any final project designs.

Only different between the two modules is that VDIP2 comes with two USB A type receptacles while VDIP1 comes with only one that is connected to the USB Port 2 of a VNC1L. With VDAP firmware, which is used for this project, both HID devices and mass storage devices (a USB memory stick) can be connected to USB Port 2 and be supported. Therefore, this project uses a VDIP1. Mass storage devices can be connected only to Port 2 with VDAP firmware, and it should not be connected to Port 1 on a VDIP2. HID devices connected to Port 1 on VDIP2 proved to work well. Although VDIP1 has only one USB type A receptacle, it has two port pins that can be connected to D- and D+ of the second USB A connector. (D- and D+ are two Data- and Data+ pinouts of a USB port.) Thus, students can add another USB port to VDIP1 with external circuit and a USB type A receptacle although simply replacing VDIP1 with VDIP2 would be much easier. How to add the second USB port to VDIP1 and the external circuit configuration is shown in the next figure as a reference.

[3] Copied from FTDI website

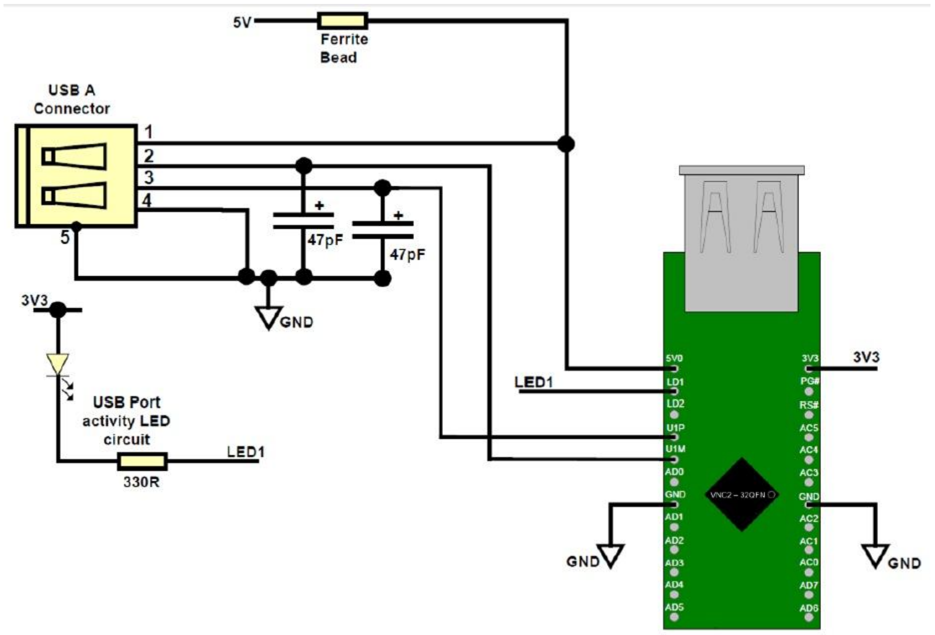


Figure 3. Additional USB port configuration for VDIP1[4]

Software: VDAP Firmware version 3.69

The firmware programed on VNC1L is what makes a VDIP1 module the most attractive solution for this project. Just as the Windows API functions magically take care of all complex USB protocol related transactions between a PC host and a USB device, the command set of firmware can do the same for a microcontroller host and a USB slave device. This unloads the microcontroller, and only with the knowledge of the firmware command set, students can easily make their MCU a USB host and incorporate a USB slave device in their final project.

Firmware Upgrade

A VNC1L chip ships from the manufacture without being programmed while a VDIP1 module is programed with firmware. However, they are often shipped with an older version of firmware, and it is important that you first check if an upgrade is needed. There are two ways to upgrade the firmware installed on a VNC1L. You can download the ROM programmer tool and the ROM files from FTDI website and program the chip via a PC serial port or an FT232, FT245, or ft2232 device. This is demonstrated in the next figure only as a reference in case students use only a VNC1L chip instead of a VDIP1 or VDIP2 module. The easiest way to upgrade the firmware on a VNC1L on a development module is to use a USB flash drive. First download the latest firmware version from FTDI website and save it in a USB flash drive. You must change the

[4] Excerpted from VDIP1 datasheet on page 17, Figure 6.1.

name of the file to “FTRFB.FTD” before you insert the drive to the USB connector (Port 2) on a development module. Make sure that the file is saved in the root directory. Once the VNC1L detects the USB drive, it will check and verify the format of the firmware upgrade file and automatically upgrade the firmware for you. This project employs the latest version of the firmware, 3.69.

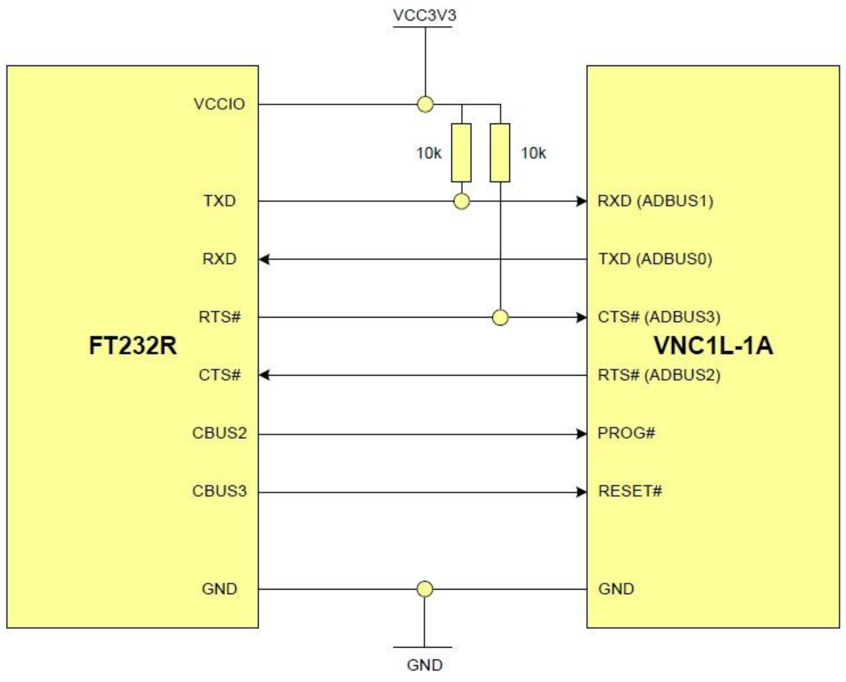


Figure 4. How to program VNC1L using a USB-serial converter[5]

There are six types of firmware available for VNC1L. This project uses VDAP firmware, which is for the most general purpose and supports most USB devices on Port 1 and Port 2. With this firmware, both HID devices and mass storage devices (BOMS: Bulk Only Mass Storage) can be connected to Port 2 on a VDIP1.

Command Monitor

The way to control and communicate with the VNC1L is through a command monitor. The firmware activates a command monitor port for one of the USB ports, and this allows an embedded device such as an MCU to communicate with a USB peripheral device via the VNC1L’s UART, SPI or FIFO interface. An MCU can send instructions (commands) to the command monitor and receive data from it. When the VNC1L is ready to take a command and after a successful execution of any command, the command monitor returns a prompt (D:\>).

[5] Excerpted from VNC1L datasheet on page 26, Figure 7.2.

There are two modes in which the command monitor works: Command Mode and Data Mode. The VNC1L firmware starts in the command mode; to switch to the data mode, the DATAREQ# line on VNC1L must be asserted low. Once in the data mode, the DATAACK# line goes low and the VNC1L simply passes data between a USB device and the monitor port. In the command mode, the VNC1L interprets and executes commands, taking appropriate actions. The operations of these two modes are depicted in Figure 5 and Figure 6 respectively.

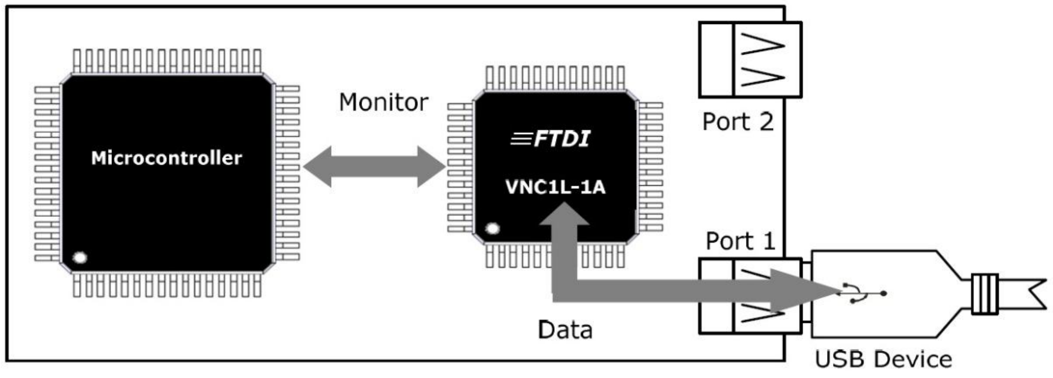


Figure 5. Command Mode connection[6]

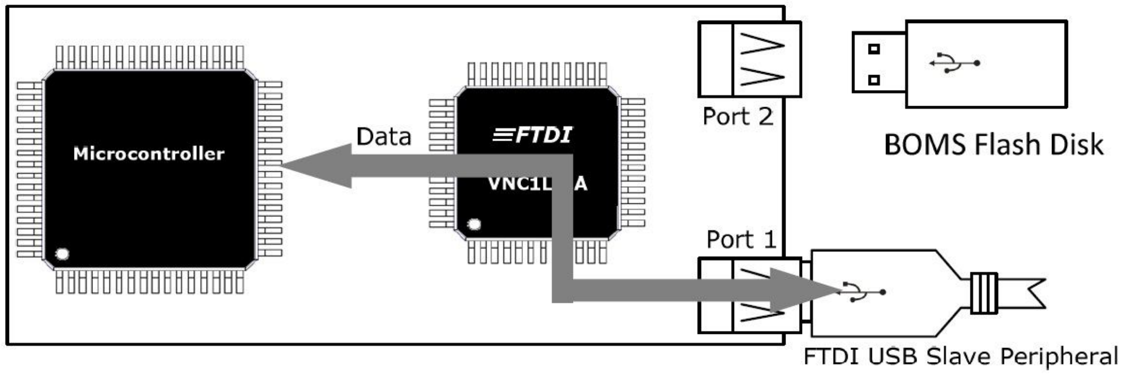


Figure 6. Data Mode connection[7]

The command monitor supports two command entry modes: Extended mode and Short mode. In the extended mode, commands are entered using printable characters such as a string of alphabet letters plus numerical numbers. In the short mode, commands are entered using binary values expressed in hexadecimal numbers. For example,

```
DSD 10↵
01 02 03 04 05 06 07 08 09 0a
```

and

[6] Excerpted from Vinculum Firmware Manual on page 66, Figure 8.2.
[7] Excerpted from Vinculum Firmware Manual on page 14, Figure 4.1.

83 20 0a 0d
01 02 03 04 05 06 07 08 09 0a

are the same command, Device Send Data, with the data size (10) in ASCII Mode for the first case and in Binary Mode for the latter. The data to be sent (01 02 03 04 05 06 07 08 09 0a) is in binary values for both cases. ↵ and 0d represent a carriage return (CR).

There are also two numerical modes available for monitor commands as shown in the example above. They are independent of a selected command set, either extended or short. ASCII Mode can be invoked by the command, “IPA”, and Binary Mode can be set by the command, “IPH”. The ASCII mode uses printable characters while the binary mode uses binary values. In the ASCII mode, unless a number starts with a prefix ‘$’ or ‘0x’, which represent hexadecimal values, a number (parameter data to accompany a command) is assumed to be decimal. An important thing to be noted is that all outputs from the monitor are LSB first regardless of which numerical mode it is in.

Essential Monitor Commands

The Firmware comes with Monitor Command Set that covers various commands for different categories. For this project, mainly Monitor Configuration Commands, Disk Commands and USB Device Commands were used. Here is a list of commands that are used:

Extended Command Set	Function
Monitor Configuration Commands	
ECS (Extended Command Set)	Switches to the extended command set
IPA (Monitor Mode ASCII)	Monitor commands use ASCII values
FWV (Firmware Version)	Display firmware version
Disk Commands	
DIR *file* (Directory)	List specified file and size
OPW *file* (Open File for Write)	Open a file for writing or create a new file
SEK *dword* (Seek)	Seek to the byte position specified by the 1st parameter in the currently open file
WRF *dword* (Write to File) data	Write the number of bytes specified in the 1st parameter to the currently open file
CLF *file* (Close File)	Close the currently open file
OPR *file* (Open File for Read)	Open a file for reading
RDF *dword* (Read From File)	Read the number of bytes specified in the 1st parameter from the currently open file
USB Device Commands	
QP2 (Query Port)	Query port 2
QD *byte* (Query Device)	Query device specified in the 1st parameter

SC *byte* (Set Current)	Set device specified in the 1st parameter as the current device
SSU *qword* (Device Send Setup Data)	Send setup data to device control endpoint with optional follow-on data

Table 2. Monitor Command Set[8]

There are a few important facts to be noted for a successful execution of monitor commands.

- Before writing to a file or reading from a file, the file has to be open by OPW or OPR. After writing to a file or reading from a file, the file has to be closed by CLF.
- Before using RDF to read from a file, DIR *file* command should be executed first to find out the size of the file to be read.
- Before using SSU to set or get report of a HID device, SC with a device number should be executed first to set the device as a current device so that all output from SSU can be routed to that device.

For more detailed descriptions on functions of any of those commands or error messages that the monitor returns when a command fails to execute, Vinculum Firmware Manual offers examples on how they can be used.

Serial Communications

VNC1L provides configuration options to interface to the Command Monitor via UART, SPI or FIFO. When the data and control buses of VNC1L are configured in the UART mode, the interface implements an asynchronous serial UART port with flow control. When the buses are configured in the SPI mode, the interface operates as an SPI slave and thus it will need a master to provide the clock. In this project, both SPI and USART are employed for the serial communication between the microcontroller host (mega1284) and the VNC1L on the VDIP1, which is connected to a USB slave device. Both SPI and USART are tested with a USB flash drive, a USB keyboard and a USB mouse, and they proved to work fine equally. Different wire connections between the mega1284 board and the VDIP1 module for SPI and UART are depicted in the next sections respectively.

SPI (Serial Peripheral Interface)

SPI allows high-speed synchronous data transfer between mega1284 and a peripheral device, in our case, VNC1L. The hardware setup for SPI between a host and a slave is

[8] Excerpted from Vinculum Firmware Manual on pages 22, 26, 27 and 41.

demonstrated in the next figure. Since SPI uses a synchronous serial communication bus, a transmitter and a receiver must use the same clock to synchronize the detection of the bits at the receiver. In SPI, both the master and the slave send and receive data simultaneously; through the shift registers as shown on the right, eight bits flow from the master to the slave and different eight bits flow from the slave to the master. The master always supplies the clock (SCLK) for synchronization and initiates the communication. To get data from the slave, the master has to send out dummy data that is shifted out of MOSI (master-out-slave-in) and into the shift register in the slave, which causes the data in the shift register in the slave to be shifted out of MISO (master-in-slave-out) and into the shift register in the master. More than one slave device can be connected to one master, and $\overline{SS}$ (chip select) can be asserted low to choose an active slave device.

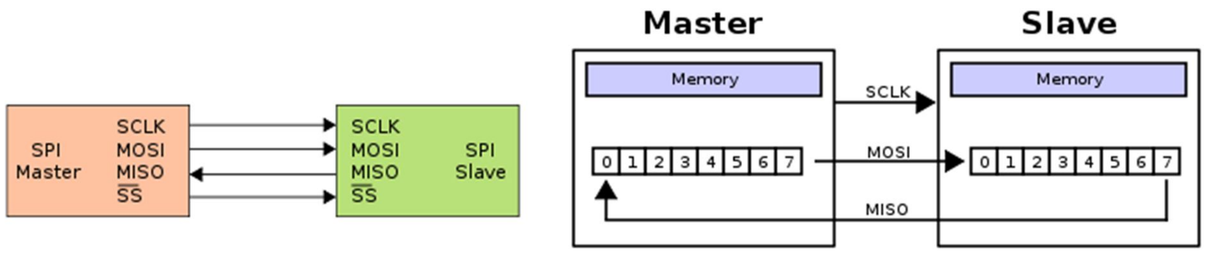


Figure 7. Buses between a host and a slave and hardware setup for SPI[9]

Four I/O pins on VNC1L are used for SPI: SCLK (SPI clock input), SDI (SPI serial data input), SDO (SPI serial data output) and CS (SPI chip select input). The timings of a read operation and a write operation of VNC1L are illustrated in the next two figures. First, during the entire read or write cycle, CS must be held high. Once the operation is over, CS must be pulled low for at least one clock period after the transaction is completed. The first bit on SDI is the write/read bit where a '1' is for read (from VNC1L) and a '0' is for write (to VNC1L). The second bit on SDI is the address bit where a '1' means that the status register will be read from or be written to and a '0' means that the data register will be read from or be written to. During the read cycle, a byte of data is output on SDO with MSB (most significant bit) first. During the write cycle, a byte of data is input to SDI with MSB first. The last status bit on SDO for the read cycle is set to a '1' to indicate that the data read is old data and a '0' for new data. The last status bit on SDO for the write cycle is set to a '0' to indicate that the data write was accepted and a '1' to indicate that the internal buffer is full and the write operation must be repeated.

[9] Copied from Wikipedia on Serial Peripheral Interface Bus

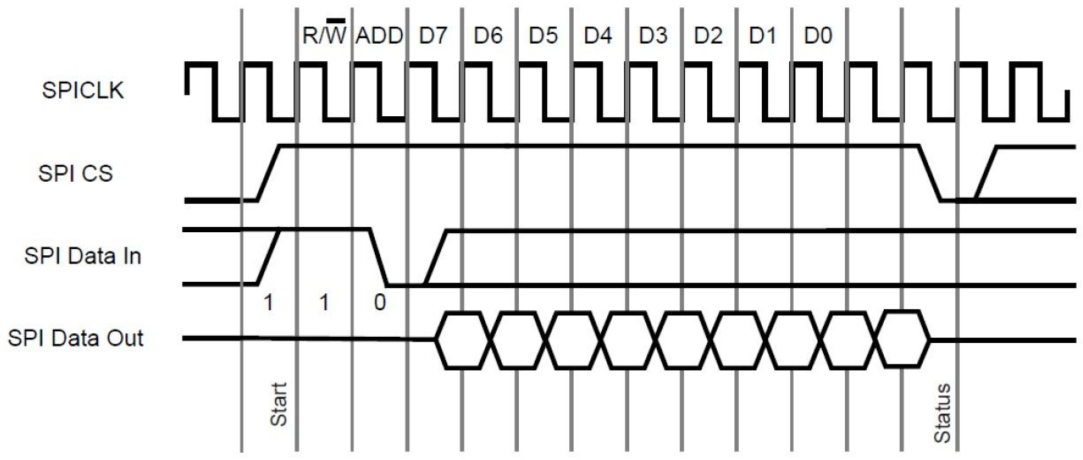


Figure 8. SPI Slave Data Read Cycle[10]

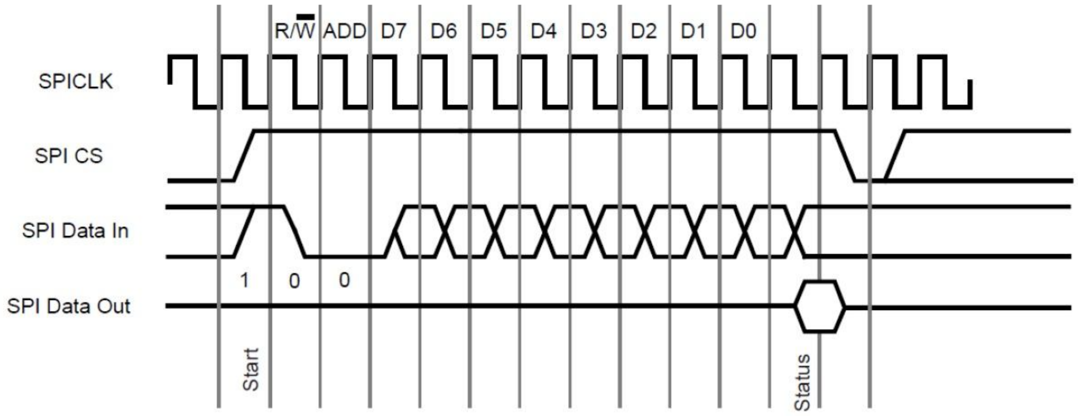


Figure 9. SPI Slave Data Write Cycle[11]

Mega1284 allows four port pins to be configured as SCLK (Port B7), MISO (Port B6), MOSI (Port B5) and $\overline{\text{SS}}$ (Port B4) for SPI and provides corresponding registers. However, in this project five pins on Port A are used to emulate the SPI interface: Port A3 (SCLK), Port A4 (SDI from VDIP1), Port A5 (SDO from VDIP1), Port A6 (CS equivalent to $\overline{\text{SS}}$) and Port A7 (RE). The difference between using SPI pins on Port B with the data and control registers for SPI and emulating the SPI interface is that Port A3 for the clock must be pulled up and down manually every time a host (MCU) wants to transmit any data to a slave (VNC1L) when in emulation. The benefit of using the SPI mode is that both USART channels on mega1284 are freed to be used for other purposes.

[10] Excerpted from VDIP1 datasheet on page 9.
[11] Excerpted from VDIP1 datasheet on page 10.

Two three way jumper pin headers on the VDIP1 module should be correctly configured for SPI interface. The red lines next to J3 and J4 in the figure below show how the jumpers should be mounted for UART along with other port pin configurations.

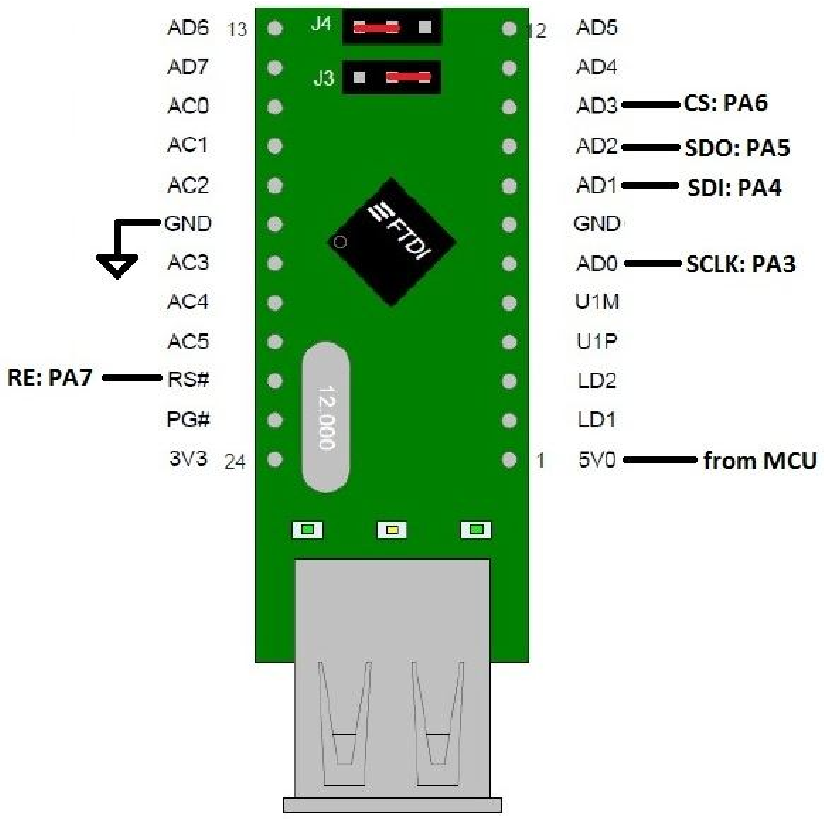


Figure 10. VDIP1 on-board jumper pin configuration and port connections to MCU for SPI[12]

USART (Universal Synchronous/Asynchronous Receiver/Transmitter)

USART that sends multiple bits of data over a single wire is commonly used for communication between a microcontroller and various other devices. UART, which is of asynchronous serial communication, does not require a common clock signal at both a transmitter and a receiver for synchronization of data detection. Instead, it uses a start bit and a stop bit that are added to the data byte.

Mega1284 conveniently comes with two channels of USART. This allows us to use the first channel (channel 0) on Port D0 (RX0) and Port D1 (TX0) for communication between a running program and a PC that runs PuTTy for debugging purposes and printing the status of

[12] Excerpted from VDIP1 datasheet on page 4.

the program execution. The second channel (channel 1) on Port D2 (RX1) and Port D3 (TX1) can be used for communication between mega1284 and VNC1L. To utilize the second channel of USART, the registers that correspond to USART channel 1 should be set up correctly in the main program with options for 8 data bits, no parity, asynchronous UART and 1 stop bit. Also, the two three way jumper pin headers on the VDIP1 module should be correctly configured for UART interface. The red lines next to J3 and J4 in the figure below show how the jumpers should be mounted for UART along with other port pin configurations.

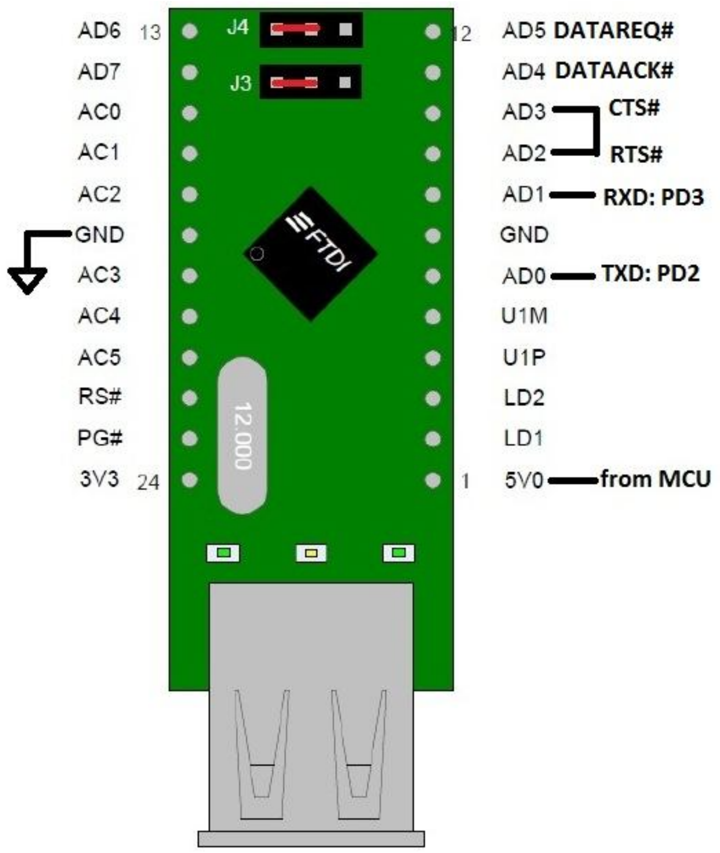


Figure 11. VDIP1 on-board jumper pin configuration and port connections to MCU for UART[13]

It should be noted that pins AD2 (RTS#: request to send) and AD3 (CTS#: clear to send) are connected together. Although the VNC1L supports the flow control functionality of UART, in this project the flow control is not utilized for both of UART channels. However, to enable UART communication, either RTS# and CTS# should be connected or CTS# should be pulled down. The benefit of using the UART mode with VDIP1 is that it requires only two ports (RX and TX) from a microcontroller plus possibly one more output port from the MCU that can pull down AD5

[13] Excerpted from VDIP1 datasheet on page 4.

(DATAREQ#) to make the VNC1L switch from Command Mode to Data Mode, which will be explained more later.

If students choose to use only a VNC1L chip instead of a VDIP1 module for their project, the next figure details how to configure a VNC1L to communicate with a microcontroller using the UART interface. Please note once again that in this project the flow control is not used. Thus ADBUS2 and ADBUS3 should be connected together instead of being connected to RTS# and CTS# of the microcontroller.

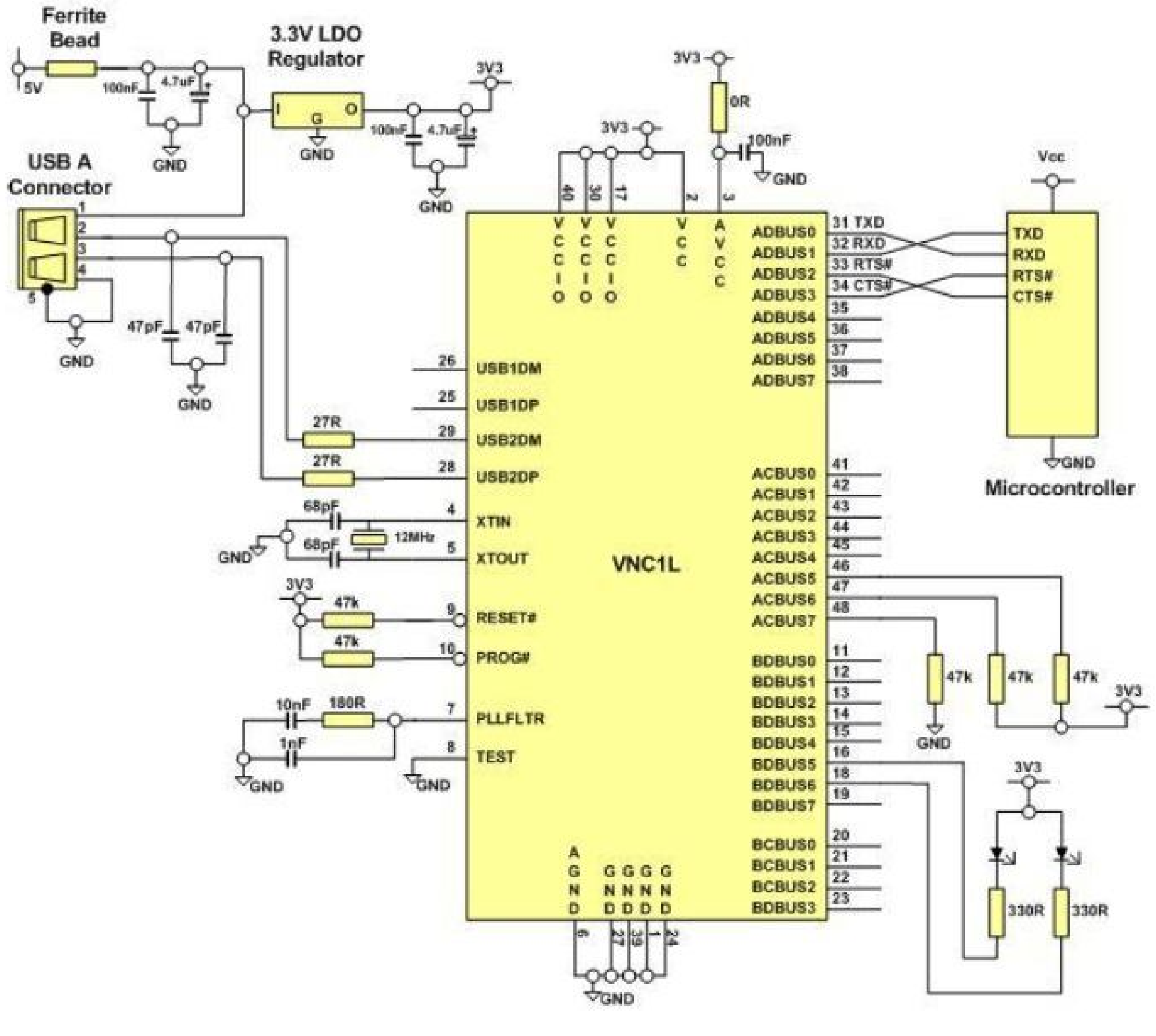


Figure 12. VNC1L schematic (MCU – UART interface)[14]

[14] Excerpted from VNC1L datasheet on page 25.

USB Mass Storage Device

VNC1L USB Host Controller project[15] by Matthias Kahnt was the basis and a good start for this project. Kahnt's project uses emulated SPI for the communication between a mega644 and a VNC1L on a VDIP1 module. It comes with its own LCD and UART libraries, which are different from the ones that are used in ECE 4760 class. Instead of using SCK, MISO, MOSI, and $\overline{SS}$ pins on Port B that mega644 sets apart for SPI, it uses five pins on Port D for SPI and Port A for an LCD display. This project comes with an extensive library that consists of all APIs that are necessary for a microcontroller host to interface with a USB flash drive. With these APIs, firmware commands can be sent to the monitor to write a text file or a binary file to a USB drive, read a text file from a USB drive and append a string of text to an existing file.

The VNC1L_USB_drive project is not much different from Kahnt's original project. It is simply re-written in English with minor changes made to the original project. The VNC1L_USB_drive project for ECE 4760 class uses Port A for the emulated SPI communication and includes the LCD (Port C) and UART (channel 0, PD0 and PD1) libraries that students in ECE 4760 are familiar with and use in class. The main program demonstrates how to write and read a text file to and from a USB drive and append text to an existing file by using a string of characters that are input by a user through PuTTy.

Once the VNC1L_USB_drive project with SPI worked well for mega1284, VNC1L_USB_UART project was implemented using the USART interface to give students another option to choose for the choice of serial communication. USART channel 1 (PD2 and PD3 for RX1 and TX1) without the flow control works fine. Although their firmware.c files have similar APIs for the same functions, there are a few differences between the VNC1L_USB_drive project and the VNC1L_USB_UART project.

- Only in VNC1L_USB_drive, vnc_init() must be called during the initialization process in main to hard-reset the VNC1L and have the SPI interface ready.
- The data that is received from the monitor is saved in a reversed order in VNC1L_USB_drive, and the buffer needs to be printed backwards for readability for a user. In VNC1L_USB_UART, however, the data from the monitor is saved with a MSB first in the buffer with the index 0, and the buffer can be printed as it is, for example, to check a firmware version.
- In VNC1L_USB_drive, vnc_rd_answer() is called in vnc_prompt_check() to check any error messages from the monitor or to see a prompt (D:\>) has been received from the

[15] USB-Stick am Mikrocontroller project from www.mikrocontroller.net.

monitor, which indicates that a command executed successfully. In VNC1L_USB_UART, vnc_prompt_check_UART() itself checks for a prompt.

- In VNC1L_USB_drive, the SPI status bit can be checked to see if the received data on the MCU side is an old one or a new one to decide whether there is any more data to be received or not. In VNC1L_USB_UART, for the program not to hang at **while((UCSR1A & (1<<RXC1)) == 0);**, tested constants are hard coded in the most of 'for' loops that include **while((UCSR1A & (1<<RXC1)) == 0);**. These constants present the number of bytes that the monitor returns such as USB protocol related information on a device or a version of firmware installed on the VNC1L.

There is a list of things that apply to both projects as well:

- First, before the MCU can access a USB drive, some housekeeping procedure should be done. Once the VNC1L is powered up, it will send messages to the monitor port regarding a version of the firmware installed on the chip and a confirmation that a USB device has been detected. These messages should be received and verified before any firmware commands can be sent to the monitor.
- For the better user interface and the easier parsing of data received from the monitor, the VN1L is set in the Command Mode and the ASCII mode by sending the commands, IPA and ECS, to the monitor. FWV command is optional because when the VNC1L is powered up, it always sends out information on the version of firmware that is installed on it as mentioned above.
- Any string of text to be written to a file should end with '\a' because it is an identifier for the end of text. '\a' is chosen so that a user is allowed to input a string of characters including spaces through PuTTy to be written to a file in a USB drive.
- Before calling vnc_rd_file() or vnc_rd_file_UART() to read a file, vnc_rd_dir() or vnc_rd_dir_UART() must be called first to find out the length of the file to be read with the DIR command because the monitor command RDF is used with a parameter for the number of bytes to be read.
- When vnc_wr_txtfile() or VNC_wr_txtfile_UART() is called, it sends OPW, SEK, WRF and CLF commands in order to the monitor. Likewise, when vnc_rd_file() or vnc_rd_file_UART() is called, it sends OPR, SEK, RDF and CLF commands in order to the monitor.
- After executing any monitor command, before the next one can be sent to the monitor, checking for a prompt (D:\>) from the monitor needs to be done first. A prompt must be read; otherwise, the program will stop running because the monitor will not be ready for another command.

USB HID Device: Keyboard

A USB keyboard can be a popular choice as an input device for many projects. Making mega1284 a USB host over a USB keyboard as a slave was implemented using both SPI and USART interface to the VNC1L, and they proved to work fine equally. For both projects, polling a report from a keyboard every 50ms using a timer and timer compare ISR (interrupt service routine) worked smoothly. The basic housekeeping procedure, which is the same as the one for a USB drive, is done at the beginning of the program such as getting a version of firmware and a confirmation from the monitor that a device was detected on Port 2.

For a USB keyboard, there is a different set of commands to be used for a host to exchange data with the device connected to the VNC1L. Before the exchange of data between a host and a slave device could happen, some more important housekeeping procedure should be done for both SPI and UART projects. To make sure if the device on Port 2 is recognized as an HID device, QP2 command is sent to the monitor. For a USB keyboard, the monitor returns "$08 $00" while "$20 $00" is returned for a USB drive. To get information on a device interface, QD command is used. QD takes a parameter that presents a device number, which can be between 0 and 15 inclusively. Zero works fine for a device number in both of the projects since VDIP1 comes with only one USB receptacle for one device to be connected to the VNC1L at a time. Among the information that the monitor returns after the QD command are a vendor ID for the device, a product ID, Pipe IN End Point number, Pipe OUT End Point number and a device class, which are part of a block of 32 bytes.

QP and QD commands are not essential and can be left out. However, before sending a command that can fetch data from a device, SC command must be executed (see Table 2). This is very important for both projects and for any mode that the VNC1L might be in. SC with the device number, which is zero for this project, allows all output from the SSU command to be routed to this specific device. Unless SC command is executed first, SSU command, which is the most import command to exchange data with HID devices, does not work at all.

Reports from an HID device can be polled and sent through the Control pipe instead of Interrupt In pipe. All USB devices must respond to requests from a host on their default Control Pipe. Class-specific requests allow a host to get a report from a device to find out the state of it and to send a report to a device to change the state of output from it. Using the SSU command, we can send get_report requests, set_report requests and set_idle requests. Only get_report requests are mandatory and all devices must support them. The detailed information on the formats of each type of requests is found in Appendix A.

After a get_report request is sent as a parameter of the SSU command, the monitor returns an IN report from a keyboard, which has information shown in Figure 13. (The direction of a report is always from a host's point of view.) Only one report is allowed in a single USB transfer, and one IN report can tell us up to six key presses plus if any of Shift, Ctrl or Alt keys is pressed as shown in Table 3. However, all projects created for a USB keyboard check only the Key_array[0] in Byte 2, and this works well with the polling method. In other words, it does not miss a keystroke and also it does not read the same keystroke twice when a user types at a fast pace. Although not all devices support set_report requests, when it is sent by the MCU through SSU, it can turn on one of three LEDs on a keyboard. The bitmap of an OUT report is tabularized in Table 4.

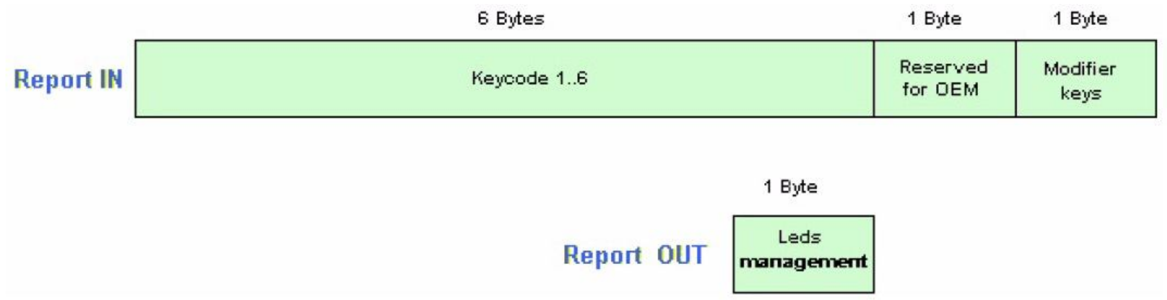


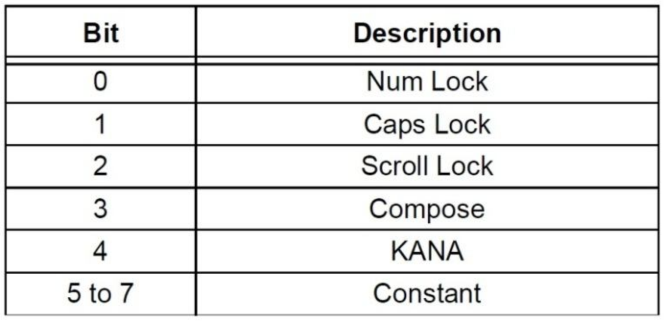
Figure 13. USB keyboard report structure[16]

A USB keyboard must send a report to a host whenever it receives a get_report request or at a set idle rate regardless of no changes in the key codes. Although *Device Class Definition for Human Interface Devices (HID)* document states that the recommended default idle rate for keyboards is 500ms, in this setting, many keystrokes are missed. Thus, the duration of the wValue field in the set_idle request for this project is set to indefinite, which means that only when there is a change in the report data, a new report can be sent to a host. In this setting, whenever the MCU polls a report every 50ms using the SSU command, the keyboard sends out a report.

	Bit7	**Bit6**	**Bit5**	**Bit4**	**Bit3**	**Bit2**	**Bit1**	**Bit0**
Byte0	Right GUI	Right Alt	Right Shift	Right Ctrl	Left GUI	Left Alt	Left Shift	Left Ctrl
Byte1	Reserved							
Byte2	Key_array[0]							
Byte3	Key_array[1]							
Byte4	Key_array[2]							
Byte5	Key_array[3]							
Byte6	Key_array[4]							
Byte7	Key_array[5]							

Table 3. USB keyboard input report format[17]

[16] Excerpted from *AVR271: USB Keyboard Demonstration* by Atmel.



Bit	Description
0	Num Lock
1	Caps Lock
2	Scroll Lock
3	Compose
4	KANA
5 to 7	Constant

Table 4. USB keyboard output report[18]

One thing to be noted is that the keycode (Key_array[] elements) in a report from a keyboard is not the same as ASCII values of printable characters. For example, when the key 'j' is pressed, the report returns a value, 0x0D, not 0x6A, which is an ASCII character value for a lowercase j. firmware.h files in both projects list key codes for a USB keyboard, which is taken from *HID Usage Tables*. More detailed information on USB protocols related to HID devices can be obtained from documents such as *Device Class Definition for Human Interface Devices (HID)* and *HID Usage Tables* on the USB-IF website.

USB HID Device: Mouse

A USB mouse can be a good choice as an input device for projects that allow users to play games. Making mega1284 a USB host over a USB mouse as a slave was implemented using both SPI and USART interface to the VNC1L, and they proved to work well equally. For both projects, polling a report from a mouse every 50ms using a timer and timer compare ISR worked nicely. Other than employing the different types of serial communication interface, both projects work the same way. The basic housekeeping procedure, which is the same as the one for a USB drive and a USB keyboard is done at the beginning of the program. Also, as for a keyboard, QP2 and QD commands are sent to the monitor to confirm that a mouse is recognized as an HID device.

The MCU sends the SD command to the monitor that all class-specific requests that the host makes will be routed to a USB mouse attached to the VDIP1 on Port 2. The SSU command is used the same way to get a report from a mouse and set the idle time to indefinite as for a keyboard. Only difference is that a report from a USB mouse is of 4 bytes when a report from a USB keyboard is of 8 bytes. Also, the host cannot make a set_report request to a mouse since

[17] Excerpted from *USB Keyboard Using MSP430^{TM} Microcontrollers* by Texas Instruments.
[18] Excerpted from *Using USB Keyboard with an Embedded Host* by Microchip Technology Inc.

there is no component on a mouse that an external source should manipulate unlike LED lights on a keyboard.

The structure of an IN report from a mouse is shown in the next figure. The bit for Button middle is set to 1 when a wheel on a mouse is clicked instead of being scrolled. Byte 1 and Byte 2 are X and Y coordinates of the position of a mouse. The value in Byte 3 either increases or decreases as the wheel on a mouse is scrolled up and down. Both a button click and a change in the position of a mouse can be reported to the host in the same IN report.

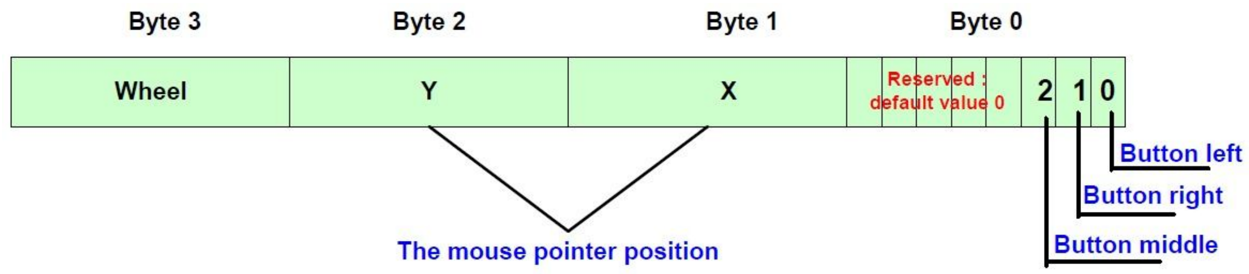


Figure 14. USB mouse report structure

Results

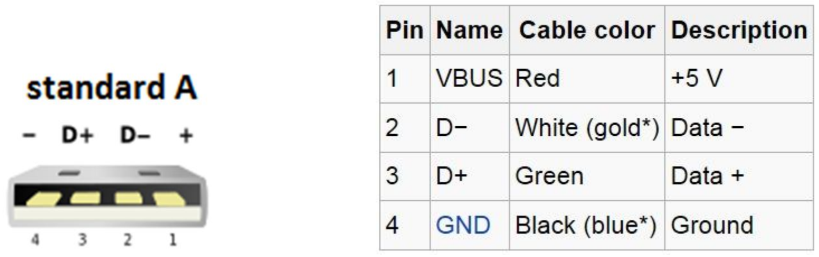
Implementing a USB host interface to Mega1284 using the VNC1L on a VDIP1 module was successful. Overall, eight project files were created to demonstrate how the host MCU can communicate with a USB device through the command monitor by calling APIs that send out firmware commands to the monitor and also parse and process data that a USB device sends to the host. This certainly unloads the host, mega1284. For each type of devices, a USB flash drive, a USB keyboard and a USB mouse, a library of API functions was created. Also, each kind of the libraries was implemented twice: one using SPI and the other in USART interface. The list of the projects files that are provided with this report is as followings:

- VNC1L_USB_drive (SPI)
- VNC1L_keyboard (SPI, polling)
- VNC1L_mouse (SPI, polling)
- VNC1L_USB_drive_UART
- VNC1L_keyboard_UART_polling
- VNC1L_mouse_UART_polling
- VNC1L_keyboard_UART_data
- VNC1L_mouse_UART_data

As mentioned in the earlier section, VNC1L_USB_drive project is based on Kahnt's original project and only minor changes are made to it. However, VNC1L_USB_drive project demonstrates that not only a hard coded string buffer but also a string of characters that is input by a user in real-time can be saved in a new file or be appended to an existing file on a USB drive. This shows that now a USB drive can be used as a good storage for a data logging program and eliminate the limit of memory space for a project that saves much data. One thing to be noted is that USB flash drives usually ship preformatted in FAT32, FAT12 or FAT16 file systems. Although the firmware user manual guarantees that it supports all those three file systems with a sector size of 512 bytes, if a certain USB drive cannot be accessed by the VNC1L, it is recommended that the drive should be re-formatted to be of FAT32 file system with a sector size of 512 bytes before it is connected to the VDIP1.

For a USB keyboard and a USB mouse, in the command mode, polling an IN report every 50 ms with the SSU command after setting the idle time to indefinite worked very well either with SPI or USART. A user can type at a considerably fast speed or click the buttons on a mouse continuously while there was no miss of those new events on the host side. However, in the data mode where data from a slave device is directly sent to a host without going through the monitor, even polling an IN report with the SSU command did not work with SPI. For example, many keystrokes were missed because it seemed that the IN reports did not come in at a regular rate, which made it impossible to parse them correctly. In the SPI mode, only a master provides the clock and thus only a master can initiate the communication. Therefore, even with the idle time set to 50 ms in the data mode, which means that a device is scheduled to send an IN report every 50 ms whether there is a new event or not (just like polling), it does not work for a keyboard or a mouse. On the other hand, polling an IN report in the data mode with USART worked as good as in the command mode for both a keyboard and a mouse.

USART can be used with RX and TX interrupts. The idea that an interrupt on RX complete could leave more time to the MCU to perform other tasks than polling a report every 50 ms was tested in the data mode. First, whether the idle time could be set correctly with the SSU command was checked in hardware because we would want a slave device to send an IN report to the host only when there was a change in the report. USB type A pinout is shown in Figure 15. To scope D+ and D-, two soldered points on the bottom of the VDIP1 module towards a receptacle were connected to an oscilloscope. With the idle time set to indefinite, whenever a key is pressed, a keyboard is expected to send an IN report eight bytes of which should be saved in a buffer in the RX Complete ISR. Interrupts were fired as expected; however, there were other problems. When a user types at a considerably fast speed, the first report contains the first keystroke in the Key_array[0] and at the second keystroke, the second report saves both keystrokes in the Key_array[0] and the Key_array[1]. The problem is that keystrokes are not saved in the same order in which they were pressed. This causes the parsing of a report to be very difficult and it was observed that one keystroke was often mistaken as repeated keystrokes. Therefore, only the polling method is recommended for the projects for a keyboard and a mouse.



Pin	Name	Cable color	Description
1	VBUS	Red	+5 V
2	D−	White (gold*)	Data −
3	D+	Green	Data +
4	GND	Black (blue*)	Ground

Figure 15. USB 1.x/2.0 standard pinout[19]

One last thing about the projects for a keyboard is that print_keystroke() can be modified to return capital letters only when an alphabet key is pressed at the same as a Shift key by checking both the byte for modifying keys (Byte 0) and the byte for a keycode (Byte 1) in an IN report. In a similar way, print_keystroke() can be expanded for other keys such as function keys.

Many USB devices were tested with both SPI and USART interfaces on their compatibility with the VNC1L. Although the documents from USB-IF clearly state which USB protocols or standards apply to all USB devices regardless of manufactures or vendors, the test results of this project prove it to be not true. For example, according to the Table 4, the bitmap of an OUT report, 0x07, used for a set_report request to a keyboard should enable the MCU host to turn on all three LED lights on a keyboard. However, 0x07 turns only two LEDs on a keyboard made by HP while 0x70 turns on all LEDs on a keyboard made by Logitech. Also, out of three USB keyboards that are tested, only one keyboard made by Logitech responded to get_report requests. Thus, included here is the list of devices that shows which ones work with VNC1L and which ones do not.

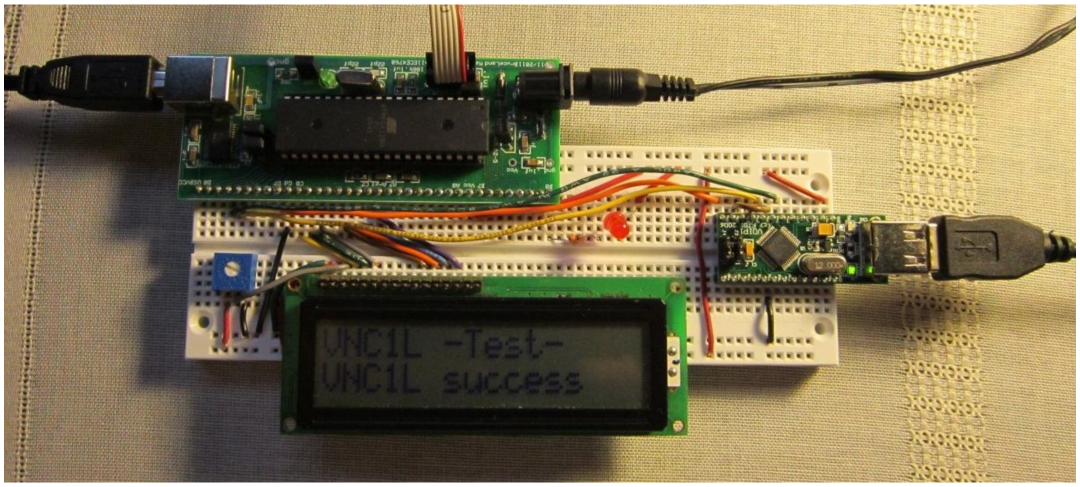
USB flash drive		
SanDisk	16 GB, FAT 32 file system	Works
Cruzer mini	1 GB, FAT 32 file system	Works
Cornell ECE (unknown vendor)	2 GB, FAT file system	Works
Unknown manufacturer	256 MB, FAT 32 file system	Works
USB keyboard		
Logitech	Model #: Y-UR83	Works
HP	Model #: SK-2885	Does not work
DELL	Model #: SK-8115	Does not work
USB mouse		

[19] Copied from Wikipedia on Universal Serial Bus

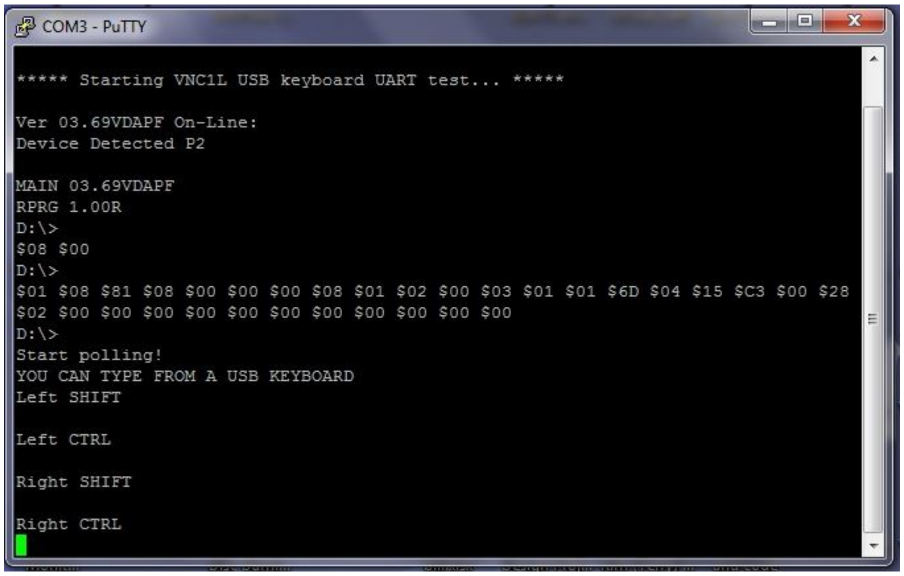
DELL	Model #: M056U0A	Work
Logitech	Model #: M-U0007	Work except for scrolling the wheel
GE	Model #: unknown	Does not work
Unknown manufacturer	Model #: unknown	Does not work

Table 5. Compatibility of the tested USB devices with VNC1L

The next is a picture of the entire system for this project. It includes a mega1284 board with a serial port, an LCD display and a VDIP1 module. The second picture is the screen shot of PuTTY with the outputs from the VNC1L when a keyboard is connected to it.



Picture 1. The project board featuring the mega1284 board, an LCD display and a VDIP1 module



```
***** Starting VNC1L USB keyboard UART test... *****

Ver 03.69VDAPF On-Line:
Device Detected P2

MAIN 03.69VDAPF
RPRG 1.00R
D:\>
$08 $00
D:\>
$01 $08 $81 $08 $00 $00 $00 $08 $01 $02 $00 $03 $01 $01 $6D $04 $15 $C3 $00 $28
$02 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00 $00
D:\>
Start polling!
YOU CAN TYPE FROM A USB KEYBOARD
Left SHIFT

Left CTRL

Right SHIFT

Right CTRL
```

Picture 2. A screenshot of PuTTy with a USB keyboard connected to the VDIP1

Conclusions

This project delivers three main sets of libraries of APIs for each type of USB peripheral devices, a USB flash drive, a USB keyboard and a USB mouse, as specified in the project proposal. The final working products of the project indicate that this was a successful project overall.

The first half of the main bulk of the project development time was spent on researches and studies on the extensive USB protocols. The second half of the development time was spent on coding the APIs in the libraries and debugging them. Although FTDI, the manufacture of VNC1L and VDIP1, claims that VNC1L provides a “ready-to-use USB interfacing solution” and their development module for VCN1L is “ideal for rapid prototyping and development of VNC1L designs”, which is true to a certain extent, learning about and understanding functions of its firmware and the monitor commands was not an easy task. It was rather a somewhat time consuming process because it turned out that much more knowledge on the USB protocols than recognizing a few technical vocabulary from the protocols is required to use the firmware commands correctly. Not all USB HID devices responded the way USB specification documents from the USB-IF explained.

Despite the challenges throughout the process of the project development time, this project has been an interesting and exciting one all along with the good results at the end.

References

Books

1. Axelson, Jan. USB Complete: The Developer's Guide. Madison: Lakeview Research LLC, 2009. Print.

2. Axelson, Jan. USB Mass Storage: Designing and Programming Devices and Embedded Hosts. Madison: Lakeview Research LLC, 2006. Print.

3. Axelson, Jan. USB Embedded Hosts: The Developer's Guide. Madison: Lakeview Research LLC, 20011. Print.

Datasheets

1. Vinculum VNC1L Embedded USB Host Controller IC Datasheet by FTDI

2. VDIP1 Vinculum VNC1L Module Datasheet by FTDI

3. VDIP2 Vinculum VNC1L Module Datasheet by FTDI

4. 8-bit Atmel Microcontroller ATmega1284 datasheet

Manual

1. Vinculum Firmware User Manual by FTDI

Documents (PDF)

1. Universal Serial Bus Specification (2.0) by USB-IF

2. Device Class Definition for Human Interface Devices (HID) by USB-IF

3. HID Usage Tables by USB-IF

4. Data-Logging Using the Vinculum VNC1L by FTDI

5. V-Eval USB Missile Launcher Application Note by USB-IF

6. Embedded USB Design by Example Part 1 &2 by John Hyde

7. Interrupt Driven USART in AVR-GCC by Dean Camera

8. AVR271: USB Keyboard Demonstration by Atmel

9. USB Keyboard Using MSP430TM Microcontrollers by Texas Instruments

10. Using USB Keyboard with an Embedded Host by Microchip Technology Inc.

Websites

1. USB Implementers Forum
www.usb.org

2. VNC1L USB Host Controller project
www.mikrocontroller.net/articles/USB-Stick_am_Mikrocontroller

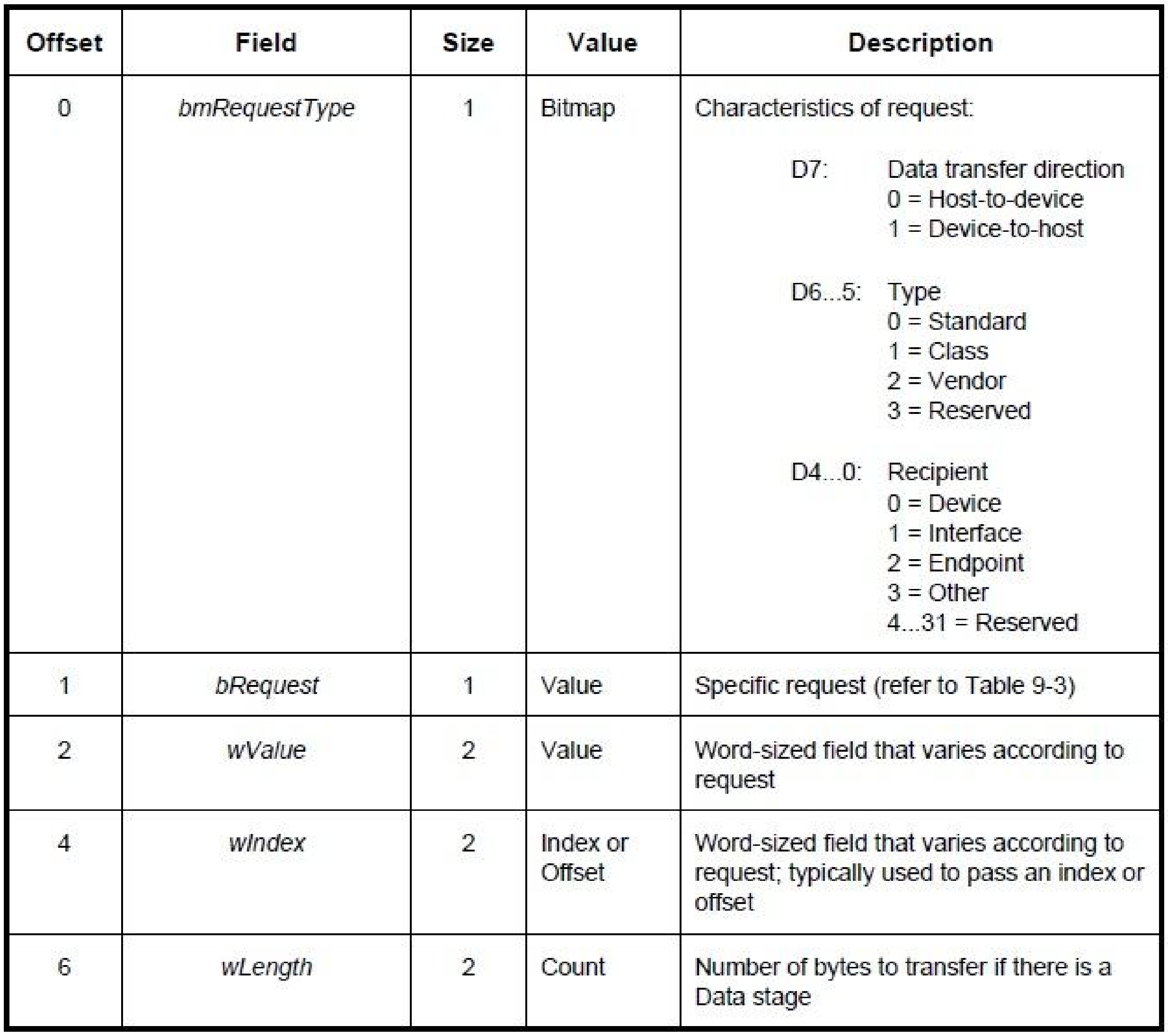
3. ECE 4760 class website
http://people.ece.cornell.edu/land/courses/ece4760/

4. Future Technology Devices International
www.ftdichip.com

5. Wikipedia
http://en.wikipedia.org/wiki/Wikipedia

Appendix A: USB Device Request

1. The format of the Setup packet in which USB device requests are sent to a USB slave device[20]:



Offset	Field	Size	Value	Description
0	*bmRequestType*	1	Bitmap	Characteristics of request: D7: Data transfer direction 0 = Host-to-device 1 = Device-to-host D6...5: Type 0 = Standard 1 = Class 2 = Vendor 3 = Reserved D4...0: Recipient 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4...31 = Reserved
1	*bRequest*	1	Value	Specific request (refer to Table 9-3)
2	*wValue*	2	Value	Word-sized field that varies according to request
4	*wIndex*	2	Index or Offset	Word-sized field that varies according to request; typically used to pass an index or offset
6	*wLength*	2	Count	Number of bytes to transfer if there is a Data stage

[20] Excerpted from *Universal Serial Bus Specification (2.0)* on page 248.

2. The format of the Setup packet for Class-Specific requests[21]:

Part	Offset/Size (Bytes)	Description
bmRequestType	0/1	Bits specifying characteristics of request. Valid values are 10100001 or 00100001 only based on the following description: 7 Data transfer direction 0 = Host to device 1 = Device to host 6..5 Type 1 = Class 4..0 Recipient 1 = Interface
bRequest	1/1	A specific request.
wValue	2/2	Numeric expression specifying word-size field (varies according to request.)
wIndex	4/2	Index or offset specifying word-size field (varies according to request.)
wLength	6/2	Numeric expressions specifying number of bytes to transfer in the data phase.

Valid values of bRequest field:

0x01	GET_REPORT
0x09	SET_REPORT
0x0A	SET_IDLE

3. Get_Report Request[22]:

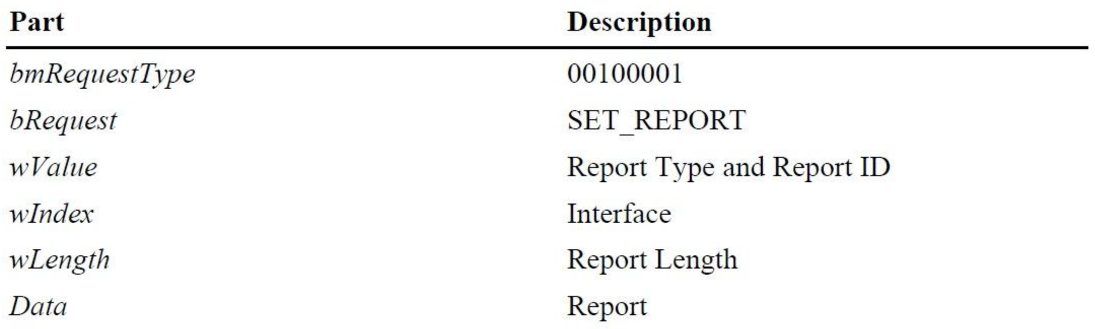
Part	Description
bmRequestType	10100001
bRequest	GET_REPORT
wValue	Report Type and Report ID
wIndex	Interface
wLength	Report Length
Data	Report

[21] Excerpted from *Device Class Definition for Human Interface Devices (HID)* on page 50.
[22] Excerpted from *Device Class Definition for Human Interface Devices (HID)* on page 51.

Valid values for Report Type in the wValue field:

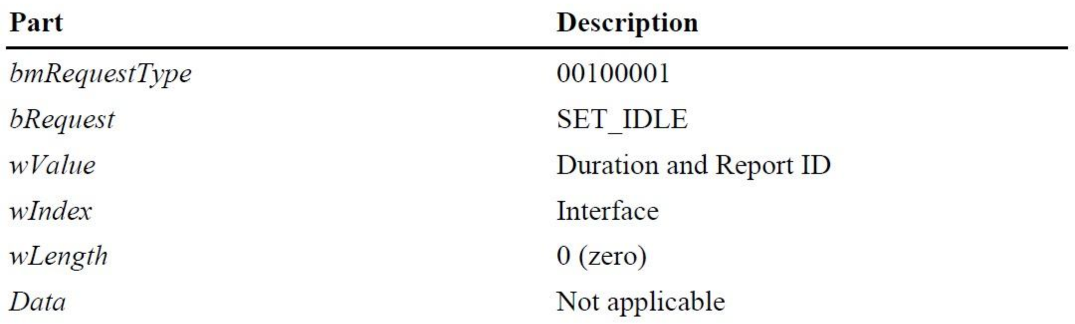
01	Input
01	Output

4. Set_Report Request[23]:



Part	Description
bmRequestType	00100001
bRequest	SET_REPORT
wValue	Report Type and Report ID
wIndex	Interface
wLength	Report Length
Data	Report

5. Set_Idle Request[24]:



Part	Description
bmRequestType	00100001
bRequest	SET_IDLE
wValue	Duration and Report ID
wIndex	Interface
wLength	0 (zero)
Data	Not applicable

[23] Excerpted from *Device Class Definition for Human Interface Devices (HID)* on page 52.
[24] Excerpted from *Device Class Definition for Human Interface Devices (HID)* on page 52.

Appendix B: User's Manual

1. Before the microcontroller is powered up, a USB device should be connected to the VDIP1, which is powered by Vcc from the MCU board.

2. Two jumper pin headers on the VDIP1 need to have two jumpers placed correctly to pull down or pull up the ACBUS5 (VNC1L pin 46) and ACBUS6 (VNC1L pin 37), depending on which interface is used, SPI or UART.

3. For the emulated SPI interface, different ports other than Port A can be used for SCLK, SDI, SDO, CS and Re by modifying firmware.h. The LCD library designates Port C for an LCD display; however, this can be changed to other ports by modifying lcd_lib.h.

4. It is highly recommended that the USART channel 0 of the MCU to PuTTy is used to print the status (including a prompt, D:\>) of the Command Monitor and vnc.receive_buffer that contains data received from the USB device through the Monitor.

5. For the UART interface, for the USART channel 1 of the MCU to be utilized, uart1.c and uart1.h should be added to the project folder along with uart.c and uart.h for the USART channel 0.

6. For the UART interface, either the CTS# pin from the VDIP1 has to be physically pulled down to GND or CTS# and RTS# pins from the VDIP1 have to be tied together because the flow control is not used in this project.

7. To switch from the Command Mode to the Data Mode (VNC1L), DATAREQ# from the VDIP1 should be pulled down to GND. One output pin from the MCU can pull DATAREQ# down just before the MCU starts polling a report from a slave device connected to the VDIP1. Additionally, DATAACK# from the VDIP1 can be observed to see if the VNC1L has switched to the Data Mode when DATAACK# is asserted low by the VNC1L (this is optional). Data Mode is used only for a USB keyboard and a USB mouse.

8. Polling every 50ms for both a keyboard and a mouse works fine without missing a change in a report from the device.

9. When the VDIP1 is powered on, two on-board LEDs, LED1 and LED2 flash alternately for 2 seconds and it repeats until the monitor connects. When commands from the monitor port are sent to a USB device, LED2 flashes. A USB drive should not be removed from the USB type A receptacle of the VDIP1 if LED2 is blinking. It is safe to remove it when LED2 is on.

Appendix C: Code

1. VNC1L_USB_drive (SPI)

```
****************** VNC1L_USB.c *******************

// Original project by Matthias Kahnt az51@gmx.net
// Modified by Terry Young Kim
// ATmega1284
// VNC1L/VDIP1 Vinculum (Port2)
// VNC1L - USB HOST CONTROLLER (Mega1284: Master, VNC1L: Slave)
// VDAP firmware version 3.68 or 3.69
// SPI interface
// VDIP1: Jumper-Set: J3 - Pulldown / J4 - Pullup for SPI

#include <avr/io.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <util/delay.h>     // for LCD

#include "uart.c"  // UART
#include "lcd_lib.c"  // LCD
#include "firmware.c" // VNC1L APIs

#define F_CPU 16000000UL

// LED: PB0
// LCD: PORT C

// VNC: SCLK PA3
//      SDI  PA4
//      SDO  PA5
//      CS   PA6
//      RS   PA7

// UART file descriptor
// putchar and getchar are in uart.c
FILE uart_str = FDEV_SETUP_STREAM(uart_putchar, uart_getchar, _FDEV_SETUP_RW);

const int8_t LCD_initialize[] PROGMEM = "LCD Initialized\0";
const int8_t LCD_VNC1L[] PROGMEM = "VNC1L -Test-\0";
const int8_t LCD_mega1284[] PROGMEM = "AVR Atmega1284\0";
const int8_t LCD_success[] PROGMEM = "VNC1L success \0";
```

```
int8_t lcd_buffer[17]; // LCD display buffer

volatile char my_text[50];
volatile char my_str[50];
volatile char read_str[50];

//*****************************************************
// LCD setup
void init_lcd(void)
{
	LCDinit();	//initialize the display
	LCDcursorOFF();
	LCDclr();				//clear the display
	LCDGotoXY(0,0);
	CopyStringtoLCD(LCD_initialize, 0, 0);
}

// Textfile 1
char *file1[] = {
	"VNC1L USB Drive test\n\r",
   "It can write to and read from a USB flash drive.\n\r",
	"Text file 1 VNC1L/VDIP1\n\r\a"};	// \a is the EOF (end of file)

// Textfile 2
char *file2[] = {
	"Text file 2,\n\r",
	"Another text file to be created.\n\r\a" };

// Textfile 3
char *file3[] = {
	"Text file 3 can be appended to an existing file.\n\r\a" };

// Bin file
char file4[] = {
	0xFA,0x89,0x11,0xFF,0xAA,0x02,0x00,0x1F,
	0x1D,0xBA,0xA6,0x24,0x99,0x2D,0x1E,0x1F,
	0x0D,0x0C,0xA0,0x55,0x30,0x1F,0x16,0xEF,
	0x04,0x22,0x00,0x00,0xCF,0x12,0x5F,0xDF,
	0x69,0x00,0x00,0x1F,0xEF,0x15,0x3F,0xCF,
	0x5F,0x44,0x1F,0x1F,0xBF,0x16,0x13,0x7F,
	0x00,0x10,0x1F,0xDF,0x1F,0x8F,0x16,0xDF,
	0x1F,0x32,0x3F,0x56,0x1F,0x19,0x17,0xFF,
	0x3F,0x42,0xFE,0x46,0x6F,0x99,0x10,0xAF,
        0x6F,0x62,0x36,0x66,0x10,0xAA,0xBB,0xCC };

// Textfile 5
char *file5[] = {
        "MacGyver left the building. He ain't coming back.\n\r\a" };

char file_buffer[FILE_BUFFER_SIZE];  // buffer for testing
unsigned int file_length;                   // byte length of a file (for reading)

// system initialization
void init_system (void)
{
  // LED for debugging
  LED_DDR |= (1 << LED_PIN); // output
  LED_D1_OFF; // turn off LED

  // VNC1L SPI interface setup
  SPI_DDR |= ((1 << SCLK_PIN) | (1 << SDI_PIN) | (1 << CS_PIN) | (1 << RE_PIN)); // outputs
  SPI_DDR &= ~(1 << SDO_PIN);        // input

    //init the UART -- uart_init() is in uart.c
  uart_init();
  stdout = stdin = stderr = &uart_str;
  fprintf(stdout,"\n\r***** Starting VNC1L USB drive test... *****\n\r");
}

// ***********************************************************************
// List of errors
// ***********************************************************************
//      1       = Command was not accepted by VNC1L
//      2       = Command Failed
//      3       = Bad Command
//      4       = Disk Full
//      5       = Filename Invalid
//      6       = Read Only
//      7       = File Open
//      8       = Directory Not Empty
//      9       = No Disk
//      20      = No VNC1L with VDAP firmware available
//      21      = No USB drive plugged
//      50      = Buffer is too small to read a file (buffer overflow)
//      51      = File length = 0 or > 64kByte
//      99      = Timeout
// ***********************************************************************
```

```
void vnc_error_message(unsigned char error_type, unsigned char error_num)
{
 LED_D1_ON;  // LED turns on if there's an error

   fprintf(stdout,"\n\rError type: %d,    Error number: %d\n\r", error_type, error_num);

 // error occured
 while(1); // error keeps the system hanging
}

// ***************************************************************
// Main program
// ***************************************************************
int main (void)
{
 init_system();

 // start the LCD
 init_lcd();
 LCDclr();

 // initialize VNC1L
 vnc_init();

 // put some stuff on LCD
 CopyStringtoLCD(LCD_VNC1L, 0, 0);//start at char=0 line=0
 CopyStringtoLCD(LCD_mega1284, 0, 1);//start at char=0 line=1

        my_text[47] = '\n';
        my_text[48] = '\r';
        my_text[49] = '\a'; // '\a' marks EOF (end of file)
        // Regardless of the size of a buffer, the last character of the buffer should be '\a'

        fprintf(stdout, "\n\rType something: ");
        fscanf(stdin, "%[^\n]s\n\r", my_text) ; // [^\n] allows user to type multiple words
separated by a space

 // Check for the presence of VDAP firmware on VNC1L
 if (vnc_wait_for("VDAP")) vnc_error_message(20,1);                    // check confirmation

  // Wait until a USB drive is plugged and detected
 if (vnc_wait_for("Device Detected")) vnc_error_message(21,2);    // check confirmation

 // Wait for the first prompt that monitor returns (D:\>)
```

```
  vnc.status = vnc_prompt_check();                                  // check confirmation
  if (vnc.status) vnc_error_message(vnc.status,3);     // error message

  // Extented Command Set
  vnc.status = vnc_wr_cmd("ECS");
  if (vnc.status) vnc_error_message(1,4);                     // error: command was not accepted
by VNC1L
  vnc.status = vnc_prompt_check();                                  // check if monitor returns
prompt
  if (vnc.status) vnc_error_message(vnc.status,4);
  // monitor returns prompt if command executed correctly

  // Monitor commands in ASCII: IPA mode
  vnc.status = vnc_wr_cmd("IPA");
  if (vnc.status) vnc_error_message(1,5);                     // error: command was not accepted
by VNC1L
  vnc.status = vnc_prompt_check();                                  // check if monitor returns
prompt
  if (vnc.status) vnc_error_message(vnc.status,5);

  // Display firmware version
  vnc.status = vnc_wr_cmd("FWV");
  if (vnc.status) vnc_error_message(1,6);                     // error: command was not accepted
by VNC1L
  vnc.status = vnc_prompt_check();                                  // check if monitor returns
prompt
  if (vnc.status) vnc_error_message(vnc.status,6);

   // List files in current directory
  vnc.status = vnc_wr_cmd("DIR");
  if (vnc.status) vnc_error_message(1,7);                     // error: command was not accepted
by VNC1L
  vnc.status = vnc_prompt_check();                                  // check if monitor returns
prompt
  if (vnc.status) vnc_error_message(vnc.status,7);

// To print what monitor returns:
//  fprintf(stdout,"\n\r%s\n\r", vnc.rec_buffer);
// note: All output from the command monitor is LSB first
//      If monitor return multiple lines of data, they are followed by carriage return,
//      which overwrites a previous line on fprintf

  // Write text file 1 to USB disk
  vnc.status = vnc_wr_txtfile("VNCFILE1.TXT", file1[0], FILE_NEW);
  if (vnc.status) vnc_error_message(vnc.status,8);

  // Write text file 2 to USB disk
  vnc.status = vnc_wr_txtfile("VNCFILE2.TXT", file2[0], FILE_NEW);
  if (vnc.status) vnc_error_message(vnc.status,9);

  // Append text string to existing text file 1
  vnc.status = vnc_wr_txtfile("VNCFILE1.TXT", file3[0], FILE_APPEND);
  if (vnc.status) vnc_error_message(vnc.status,10);

  // Write a binary (bin) file to disk
  vnc.status = vnc_wr_binfile("VNCFILE3.BIN", file4, 80); // file length = 80
  if (vnc.status) vnc_error_message(vnc.status,11);

  // Before reading a file, you must get the length of it
  vnc.status = vnc_rd_dir("VNCFILE2.TXT", file_buffer, &file_length);
  if (vnc.status) vnc_error_message(vnc.status,12);
  // Now the file length is saved in file_length

  if (!vnc.status) {
    // Read a text file of known length into file_buffer
    vnc.status = vnc_rd_file("VNCFILE2.TXT", file_buffer, file_length);
    if (vnc.status) vnc_error_message(vnc.status,13);
  }

  // Write text file 5 to disk
//  vnc.status = vnc_wr_txtfile("VNCFILE5.TXT", file5[0], FILE_NEW);
//  if (vnc.status) vnc_fehlermeldung(vnc.status,14);

  // Write text file 5 to disk using user input string
  vnc.status = vnc_wr_txtfile("VNCFILE5.TXT", my_text, FILE_NEW);
  if (vnc.status) vnc_error_message(vnc.status,14);

    // Get the length of file
  vnc.status = vnc_rd_dir("VNCFILE5.TXT", read_str, &file_length);
  if (vnc.status) vnc_error_message(vnc.status,15);

  if (!vnc.status) {
    // Read a text file of known length into my_str
    vnc.status = vnc_rd_file("VNCFILE5.TXT", my_str, file_length);
    if (vnc.status) vnc_error_message(vnc.status,16);
  }

//  fprintf(stdout, "Echo back: \n\r%s\n\r", my_str);
```

```
  // Append a text string to an existing file
  vnc.status = vnc_wr_txtfile("VNCFILE5.TXT", file5[0], FILE_APPEND);
  if (vnc.status) vnc_error_message(vnc.status,17);

/*
  // Append a text string to an existing file
  vnc.status = vnc_wr_txtfile("VNCFILE5.TXT", my_str, FILE_APPEND);
  // append twice for some reason?!?!?!?!?
  // my_str seems to cause the problem
  if (vnc.status) vnc_fehlermeldung(vnc.status,17);
*/

  // Append a text string to an existing file
  vnc.status = vnc_wr_txtfile("VNCFILE5.TXT", my_text, FILE_APPEND);
  if (vnc.status) vnc_error_message(vnc.status,17);

  // If you reached here, then no error has occured
  // LED blinks every second
  // USB drive can be removed
  while(1) {
   LED_D1_ON;
   _delay_ms(1000); // 1 sec
   LED_D1_OFF;
   _delay_ms(1000);
  }
}
```

***************** *firmware.h* ******************

```
#define LED_PIN            0
#define SCLK_PIN    3
#define SDI_PIN            4
#define SDO_PIN            5
#define CS_PIN             6
#define RE_PIN             7

#define LED_PORT    PORTB
#define SPI_PORT    PORTA // PORTD

#define LED_DDR            DDRB
#define SPI_DDR            DDRA  // DDRD
#define SPI_PIN            PINA  // PIND
```

```
//**********************************************************************
//**********************************************************************
#define LED_D1_ON   LED_PORT &= ~(1<<LED_PIN) /* LED */
#define LED_D1_OFF  LED_PORT |=  (1<<LED_PIN)

#define SCLK_L                SPI_PORT &= ~(1<<SCLK_PIN)             /* SCLK VNC1L */
#define SCLK_H                SPI_PORT |=  (1<<SCLK_PIN)

#define SDI_L          SPI_PORT &= ~(1<<SDI_PIN)          /* SDI  VNC1L */
#define SDI_H          SPI_PORT |=  (1<<SDI_PIN)

#define SDO_VNC               (SPI_PIN & (1<<PD5))                 /* SDO  VNC1L */

#define CS_VNC_L       SPI_PORT &= ~(1<<CS_PIN)           /* CS   VNC1L */
#define CS_VNC_H       SPI_PORT |=  (1<<CS_PIN)

#define RE_VNC_L       SPI_PORT &= ~(1<<RE_PIN)           /* RS   VNC1L */
#define RE_VNC_H       SPI_PORT |=  (1<<RE_PIN)

#define PROMPT_CHECK                     2
#define PROMPT_AND_ERROR_CHECK           1
#define PROMPT_OK                        1
#define FILE_NEW                         0
#define FILE_APPEND                      1
#define TIMEOUT_SPI                      50            /* Number of SPI read/write tests */
#define TIMEOUT_READ_WRITE               500           /* Number of read/write tests if
buffer is full */
#define TIMEOUT_COMMUNICATION            5             /* Timeout if no communication has
happened */

#define FILE_BUFFER_SIZE     100              /* Buffer size is at least 100 */
```

****************** *firmware.c* *******************

```
#include "firmware.h" // defines macros
#define VNC_BUFFER_SIZE 60           /* Transmit/receive buffer size is at least 60 */

struct vnc_struc
{
        char  receive_buffer[VNC_BUFFER_SIZE];      // Receive buffer
        char send_buffer[VNC_BUFFER_SIZE];          // Send buffer
        unsigned char status;                       // Status
};
```

```
struct vnc_struc vnc; // instantiate vnc_struc

//int8_t lcd_buffer[17];          // LCD display buffer

// *******************************************************************
// VNC1L - Initialize SPI-Interface and VNC1L
// SPI Mode
// *******************************************************************
void vnc_init(void)
{
  CS_VNC_L;                       // SPI-Interface inactive
  SCLK_L;                         // idle status of SPI lines
  SDI_H;
  RE_VNC_L;                       // Hardware-reset VNC1L
  _delay_ms(100);       // Pause
  RE_VNC_H;                       // start VNC1L
  _delay_ms(500);       // Pause
}

// *******************************************************************
// VNC1L - 13-Bit-SPI Sending a character
// SPI Mode
// Parameter: wr_byte   : character to be output
// Return: status    : 0 = Accepted (OK),  1 = Byte is not accepted by VNC1L
// *******************************************************************
unsigned char vnc_wr_spi(unsigned char wr_byte)
{
        unsigned char bit_pos, status, count;

  count = TIMEOUT_SPI;          // timeout after unsuccessful number of attempts to write

  do {
   CS_VNC_H;                      // SPI-Interface active
   SDI_H;                                  // Start bit (always 1)
   SCLK_H;  SCLK_L;     // falling/rising edge
   SDI_L;                                  // Read/Write bit (0 = Write)
   SCLK_H;  SCLK_L;
   SDI_L;                                  // ADDR (0 = Data)
   SCLK_H;  SCLK_L;

   // send 8 data bits (MSB first)
   for ( bit_pos = 0x80; bit_pos; bit_pos >>= 1 )
        {
    if (wr_byte & bit_pos) SDI_H; else SDI_L;
```

```
     SCLK_H;  SCLK_L;
    }
    // read status
    status = SDO_VNC;  // Status bit (0 = Accepted, 1 = Rejected)
    SCLK_H;  SCLK_L;
    // End sequence
    CS_VNC_L;                          // SPI-Interface inactive
    SCLK_H;  SCLK_L;

    count--;

         if (status) _delay_us(150);       // Pause if a byte transfer has not be confirmed
  } while (status && count);

/*
#ifdef  USART_MONITORING
         fprintf(stdout, "%c", wr_byte);
#endif
#ifdef LCD_MONITORING
         sprintf(lcd_buffer, "%s", wr_byte);
         LCDGotoXY(0, 0);
         LCDstring(lcd_buffer, strlen(lcd_buffer));
#endif
*/
  return status; // 0: no error
}

// *********************************************************************
// VNC1L - 13-Bit-SPI receiving a character
// SPI Mode
// Parameter: rd_byte    : type of read byte (0 = Data byte, 1 = Status byte)
// Store the data byte in the vnc.receive_buffer of vnc struct variable
// Return: status    : 0 = new byte in the buffer (OK), 1 = a new byte is not available from VNC1L
// *********************************************************************
unsigned char vnc_rd_spi(unsigned char rd_byte)
{
         unsigned char bit_pos, data, status;
         unsigned int count;

  count = TIMEOUT_SPI;          // timeout after unsuccessful number of attempts to write

  do {
    data  = 0;
    CS_VNC_H;                          // SPI-Interface active
```

```
   SDI_H;                              // Start bit (always 1)
   SCLK_H;  SCLK_L;     // falling/rising edge
   SDI_H;                              // Read/Write bit (1 = Read)
   SCLK_H;  SCLK_L;

   if (rd_byte) SDI_H; else SDI_L;        // ADDR (0 = Data, 1 = SPI Status information)
   SCLK_H;  SCLK_L;

   // receive 8 data bits (MSB first)
   for ( bit_pos = 0x80; bit_pos; bit_pos >>= 1 )
         {
     if (SDO_VNC) data |= bit_pos;       // store data bits
     SCLK_H;  SCLK_L;
   }

   // read status
   status = SDO_VNC;  // Status bit (0 = New Data, 1 = Old Data)
   SCLK_H;  SCLK_L;
   // End sequence
   CS_VNC_L;                    // SPI-Interface inactive
   SCLK_H;  SCLK_L;

   if (!status) {
     // received character is a new character
     // make room in the vnc.receive_buffer for new data received
     for ( bit_pos = VNC_BUFFER_SIZE - 1; bit_pos; bit_pos-- )
                   vnc.receive_buffer[bit_pos] = vnc.receive_buffer[bit_pos - 1];
     vnc.receive_buffer[0] = data;        // new charater is store in the first spot of the buffer
(backwards)

/*
#ifdef  USART_MONITORING
                   if (!rd_byte) {
                             fprintf(stdout, "%c", vnc.receive_byte[0]);
                   }
#endif
#ifdef LCD_MONITORING
                   if (!rd_byte) {
                             sprintf(lcd_buffer, "%s", vnc.receive_byte[0]);
                             LCDGotoXY(0, 0);
                             LCDstring(lcd_buffer, strlen(lcd_buffer));
                   }
#endif
*/
```

```
   }
   count--;
          if (status) _delay_us(150);      // pause if there is not a new byte
  } while (status && count);

  return status;
}

// ************************************************************************
// VNC1L - Sending a character
// Parameter: wr_byte   : character to be output
// Return: status    : 0 = Accepted (OK), 1 = Not accepted by VNC1L
// ************************************************************************
unsigned char vnc_wr_byte(unsigned char wr_byte)
{
          unsigned char status;
          unsigned int count;

  count = TIMEOUT_READ_WRITE;                          // number of attempts to send before
timeout

  do {
   // read status byte from VNC1L
   if (vnc_rd_spi(1)) return 1;             // error, byte is not accepted
          // Status byte is now in the buffer
          status = vnc.receive_buffer[0] & 0x01;          // test for VNC1L RECEIVE BUFFER full (RXF#
Bit0=1)
   count--;
          if (status) _delay_us(150);                          // Pause if RECEIVE BUFFER full (RXF#)
  } while (status && count);
  if (status || (!count)) return 1;                   // error, byte is not accepted
  // VNC1L is now ready to receive
  if (vnc_wr_spi(wr_byte)) return 1;    // error, byte is not accepted

  return 0;        // byte has been sent
}

// ************************************************************************
// VNC1L - Receiving a character
// Return: status    : 0 = New byte in the buffer (OK), 1 = a new byte is not available from VNC1L
// ************************************************************************
unsigned char vnc_rd_byte(void)
{
          unsigned char status;
```

```
        unsigned int count;

  count = TIMEOUT_READ_WRITE;                 // number of attempts to receive before timeout

  do {
   // read status byte from VNC1L
   status = vnc_rd_spi(0);                    // If status = 1, there is not a new byte
        // Testing for TXE # in the status byte has no advantage
   count--;
        if (status) _delay_us(150);           // Pause if there is no new byte
  } while (status && count);
  if (status || (!count)) return 1;        // error, byte is not received

  return 0;        // byte is received
}

// *********************************************************************
// VNC1L - Sending a command
// Parameter: wr_cmd : command (address)
// Return: status    : 0 = OK,  1 = command not accepted by VNC1L
// *********************************************************************
unsigned char vnc_wr_cmd(char *wr_cmd)
{
        unsigned char offset;  // vnc_offset, length of command

  for(offset = strlen(wr_cmd); offset; offset--)
  {
   if (vnc_wr_byte(*wr_cmd)) return 1;        // error (overflow), command was not accepted
   wr_cmd++;
  }
  // send carrage return (a command to monitor ends with 0x0D)
  if (vnc_wr_byte(0x0D)) return 1;            // error (overflow), command was not accepted

  return 0;
}

// *********************************************************************
// VNC1L - Receive a string
// Wait until the monitor returns info from firmware (ex. Firmware Version, Device Detected)
// Last 12 digits are significant
// Parameter: wr_string : String (address)
// Return: status    : 0 = OK,  1 = string was not received from VNC1L
// *********************************************************************
unsigned char vnc_wait_for(char *wr_string)
```

```
{
        unsigned char length;               // String length / Offset
        unsigned char match_index;          // number of matches
        unsigned char buff_index;           // Buffer index
        unsigned char i;
        unsigned char status;
        unsigned int rd_answer_count;

 rd_answer_count = TIMEOUT_COMMUNICATION; // TIMEOUT error / abort if necessary
 status      = 0;
 match_index  = 0;
 buff_index   = 0;
 length = strlen(wr_string);

 if (length > 12) length = 12;   // length limit

 while(rd_answer_count && (length != match_index)) {
  status = vnc_rd_byte();
        if (!status) {
         // A new character is now available
         buff_index   = 0;
         match_index  = 0;

   for(i = 0; i < length; i++)
         {
                if (vnc.receive_buffer[length - i] == *(wr_string + i))
                        match_index++;          // match
                buff_index++;
   }

         rd_answer_count = TIMEOUT_COMMUNICATION; // reset timeout count
        } else rd_answer_count--;
 }

 if (status || (match_index != length)) return 1;       // error, string is not received

 return 0;       // String was received OK
}

// ******************************************************************
// VNC1L - Wait for response string and parse it
// The last 4 digits of the answer are significant. (possibly expand it)
// Parameter: buffer_check
//                              1 = Testing for prompt and error messages
//          2 = Testing only for prompt
// Return: reply
//                    1 = Prompt received
//        2 = Command Failed received
//        3 = Bad Command
//        4 = Disk Full
//        5 = Filename Invalid
//        6 = Read Only
//        7 = File Open
//        8 = Directory Not Empty
//        9 = No Disk
//       99 = Timeout
// ******************************************************************
unsigned char vnc_rd_answer(unsigned char buffer_check)
{
            unsigned char reply;
            unsigned char status;
            unsigned int rd_answer_count;

  rd_answer_count = TIMEOUT_COMMUNICATION; // TIMEOUT error / abort if necessary
  reply = 0;

  for (status = VNC_BUFFER_SIZE - 1; status; status-- )
                        vnc.receive_buffer[status] = 0;           // clear buffer (using status variable)

  while(rd_answer_count) // not reached timeout limit
  {
    status = vnc_rd_byte();

            if (!status)
            {
              // new character is now available
              switch (buffer_check)
              {
         case 1: // Command Failed
                if ((vnc.receive_buffer[3] == 'i') && (vnc.receive_buffer[2] == 'l') &&
(vnc.receive_buffer[1] == 'e') && (vnc.receive_buffer[0] == 'd')) reply = 2;
                // Bad Command
                if ((vnc.receive_buffer[3] == 'm') && (vnc.receive_buffer[2] == 'a') &&
(vnc.receive_buffer[1] == 'n') && (vnc.receive_buffer[0] == 'd')) reply = 3;
                // Disk Full
                if ((vnc.receive_buffer[3] == 'F') && (vnc.receive_buffer[2] == 'u') &&
(vnc.receive_buffer[1] == 'l') && (vnc.receive_buffer[0] == 'l')) reply = 4;
                // Invalid Filename
                if ((vnc.receive_buffer[3] == 'a') && (vnc.receive_buffer[2] == 'l') &&
(vnc.receive_buffer[1] == 'i') && (vnc.receive_buffer[0] == 'd')) reply = 5;
                // Read Only
                if ((vnc.receive_buffer[3] == 'O') && (vnc.receive_buffer[2] == 'n') &&
(vnc.receive_buffer[1] == 'l') && (vnc.receive_buffer[0] == 'y')) reply = 6;
                // File Open
                if ((vnc.receive_buffer[3] == 'O') && (vnc.receive_buffer[2] == 'p') &&
(vnc.receive_buffer[1] == 'e') && (vnc.receive_buffer[0] == 'n')) reply = 7;
                // Directory Not Empty
                if ((vnc.receive_buffer[3] == 'm') && (vnc.receive_buffer[2] == 'p') &&
(vnc.receive_buffer[1] == 't') && (vnc.receive_buffer[0] == 'y')) reply = 8;
                // No Disk
                if ((vnc.receive_buffer[3] == 'D') && (vnc.receive_buffer[2] == 'i') &&
(vnc.receive_buffer[1] == 's') && (vnc.receive_buffer[0] == 'k')) reply = 9;
                                        // Here you can check for other error messages ...
                                        // No 'break;' here!! After error check, prompt check should be
done as well

            case 2:// Only Prompt Check
                if ((vnc.receive_buffer[2] == 0x5C) && (vnc.receive_buffer[1] == '>') &&
(vnc.receive_buffer[0] == 0x0D))
                                        {
                  // Test for \>, Carriage Return is received
                  reply = 1;
                  // vnc.receive_buffer[0] = 0;        // Buffer discard (delete the last character from the
last string!)
                                }
      }

      if (reply >= 2)
            {
              if(vnc_wr_byte(0x0D)) return 0;     // send Carriage Return to confirm error, then
comes a prompt, exit on error
      }

            if (reply) break;

            rd_answer_count = TIMEOUT_COMMUNICATION; // Timeout reset
    } else rd_answer_count--;
  }

  if (!reply) reply = 99;  // Timeout

  return reply;
```

```
}

// ********************************************************************
// VNC1L - Waiting for PROMPT
// If there is an error, return the error number
// Return: 0 = Prompt received
//         xx = error number from vnc_rd_answer function
// ********************************************************************
unsigned char vnc_prompt_check(void)
{
        unsigned char status_1, status_2;

  status_1 = vnc_rd_answer(PROMPT_AND_ERROR_CHECK);

  if (status_1 == 1) return 0;    // Prompt was received, no previous error, command was
executed

  // Error occurred, now just waiting for prompt, but confirming error
  status_2 = vnc_rd_answer(PROMPT_CHECK);

  if (status_2 == 1) return status_1;

  return status_2;                                    // no Prompt, Timeout
}

// ********************************************************************
// VNC1L - Creating a text file on a USB drive
// Maximum file length: 64 kByte
// Parameters: file_name    : address of file_name
//          file_content : address of text block to be written
//          option       : File 0 = new or 1 = append content
// Return: status                  : error number (0 = OK)
// ********************************************************************
unsigned char vnc_wr_txtfile(char *file_name, char *file_content, char option)
{
        unsigned int vnc_i                = 0;
        unsigned int file_length = 0;
        unsigned int file_CR    = 0; // carriage return
        char *file_ptr;
        unsigned char wr_steps          = 0;
        unsigned char status   = 0;

  // calculate number of bytes
  file_ptr = file_content;
```

```
 file_length = 0;
 file_CR  = 0;

 while (*file_ptr != '\a') // '\a' is an identifier for the end of text
 {
   if (*file_ptr) file_length++; else file_CR++;
   file_ptr++;
 }

 do {
   if (wr_steps == 4) return 0; // Done, OK (last prompt character exists)

   switch (wr_steps) // write process
         {
     case 0:  // Open or create file
           sprintf (&vnc.send_buffer[0], "OPW %s", file_name);
           if (vnc_wr_cmd(vnc.send_buffer)) return 1;// Error, command was not accepted
           break;

     case 1:  // Seek offset byte position of 0 in the open file
           if (!option) // new file
                         {
             if (vnc_wr_cmd("SEK 0")) return 1;                    // Error, command was not accepted
                                 break;
           }
                         else wr_steps++;      // skip to case 2 (append at the end of existing file)

     case 2:  // Write the number of bytes to an open file
           sprintf (&vnc.send_buffer[0], "WRF %u", file_length);
           if (vnc_wr_cmd(vnc.send_buffer)) return 1;// Error, command was not accepted
           file_ptr = file_content;

           for(vnc_i = 0; vnc_i < (file_length + file_CR); vnc_i++)
                         {
             if (*file_ptr)
                                 {
                                   // hide NULL termination
                                   if (vnc_wr_byte(*file_ptr)) return 1;                    // Error,
command was not accepted
             }
                                 file_ptr++;
           }
           break;
```

```
    case 3:  // Close file
            sprintf (&vnc.send_buffer[0], "CLF %s", file_name);
            if (vnc_wr_cmd(vnc.send_buffer)) return 1;// Error, command was not accepted
            break;
   } // Switch

   wr_steps++; // next step
          status = vnc_rd_answer(PROMPT_AND_ERROR_CHECK);
  } while (status == PROMPT_OK);       // Testing for next prompt character >

  return status;// error number (while 0 = OK)
}

// *******************************************************************
// VNC1L - Reading a file of a specific length from an USB drive
// Length of the file should be known, so determine it beforehand with vnc_rd_dir!
// Parameters: file_name   : address of file_name
//             file_content : address of destination buffer where the read data will be stored
//             file_length    : number of bytes
// Return: status                      : error number, 0 = OK
// *******************************************************************
unsigned char vnc_rd_file(char *file_name, char *file_content, unsigned int file_length)
{
          unsigned char rd_steps           = 0;
          unsigned char status   = 0;

  if (file_length >= (FILE_BUFFER_SIZE - 1))
  {
   return 50;     // error, buffer is too small
  }

  do {
   if (rd_steps == 4) return 0; // Done, OK (last prompt character exists)

   switch (rd_steps) // read process
          {
    case 0:  // Open file for reading
            sprintf (&vnc.send_buffer[0], "OPR %s", file_name);
            if (vnc_wr_cmd(vnc.send_buffer)) return 1;// Error, command was not accepted
            break;

    case 1:  // Seek offset byte position of 0 in the open file (you can also choose other offset)
            if (vnc_wr_cmd("SEK 0")) return 1;                          // Error, command was not accepted
            break;
```

```
     case 2:  // Read the number of bytes from an open file
           sprintf (&vnc.send_buffer[0], "RDF %d", file_length);
           if (vnc_wr_cmd(vnc.send_buffer)) return 1;// Error, command was not accepted

           while (file_length)
                             {
            if (vnc_rd_byte()) return 1;                        // error, byte is not received
            // new character exists
            *file_content = vnc.receive_buffer[0];
            file_content++;
            file_length--;
           }
           // The characters read are now in the buffer and can be evaluated.
           break;

     case 3:  // close file
           sprintf (&vnc.send_buffer[0], "CLF %s", file_name);
           if (vnc_wr_cmd(vnc.send_buffer)) return 1;// Error, command was not accepted
           break;
   } // Switch
   rd_steps++;  // next step
          status = vnc_rd_answer(PROMPT_AND_ERROR_CHECK);
 } while (status == PROMPT_OK);       // Testing for next prompt character >

 return status;// error number (while 0 = OK)
}

// ******************************************************************
// VNC1L - Convert hex string to a number
// Example: "$11 $22 $33 $44 D:\>" --> 0x44332211
// Maximum file length: 64 kByte (easily expandable to 32-bit)
// Parameter: value   : Address of hex string
// Return: file_length (16 bit),  0 = error
// ******************************************************************
unsigned int str_to_int(char *value)
{
          unsigned char temp[4];
          unsigned char position = 0;
          unsigned char nibble = 0;
          unsigned char error  = 0;
          unsigned char mid_byte = 0;

 for (position = 4; position; position--)
 {
   if ( (*value == '0') && ( *(value + 1) == '$') ) value += 2;                  // skip $00
   if (*value == '$') value += 1;                                                                     // skip
$
   mid_byte = 4;
   temp[position - 1] = 0;

   for (nibble = 2; nibble; nibble--)
         {
     if ((*value >= 'a') && (*value <= 'f')) *value = *value - 0x20;        // Convert to uppercase
     if ((*value >= 'A') && (*value <= 'F')) *value = *value - 0x07;       // normalize (hex based)
     if ((*value < '0') || (*value > '?')) error = 1;                                        // error
     temp[position - 1] |= (*value - 0x30) << mid_byte; // string to a number (subtract 0x30 from
a character 0-9)
     mid_byte -= 4; // next lower 4 bits out of a byte
     value++;
   }
   value++;
 }

 if ((temp[1] != 0) || (temp[0] != 0 )) error = 1; // error > 16-bit
 if (error)
 {
   // *(value - 16) = 0x00; // No response necessary via buffer
   temp[2] = 0;
         temp[3] = 0;
 }

 return temp[2] * 256 + temp[3];
}

// ***********************************************************************
// VNC1L - DIR determines file length
// Maximum 64 kByte!!!
// Parameters: file_name   : Address of file_name
//            file_content : Address of the intermediate stores (temporary use)
//            file_length    : Address of the length of the file to be played back
// Return: status                     : error number,  0 = OK
// ***********************************************************************
unsigned char vnc_rd_dir(char *file_name, char *file_content, unsigned int *file_length)
{
         unsigned char status              = 0;
         unsigned int buff_index                    = 0;
         char *start;                                                  // Address of hex strings (beginning)
```

```
  sprintf (&vnc.send_buffer[0], "DIR %s", file_name);
  if (vnc_wr_cmd(vnc.send_buffer)) return 1;            // Error, command was not accepted
  // Response string to DIR is read
  status = vnc_rd_answer(PROMPT_AND_ERROR_CHECK);

  if (status == PROMPT_OK)
  {
    // Response string is now in the buffer (the last character at the forefront)
    for (buff_index = VNC_BUFFER_SIZE; buff_index; buff_index--)
          {
            *file_content = vnc.receive_buffer[buff_index - 1];// flip the buffer content
            if (!(*file_content)) *file_content = ' ';                    // avoid string end identifier (0x00)
            file_content++;
    }

          *(file_content) = 0x00;                                                  // put string end identifier
          file_content -= (VNC_BUFFER_SIZE - 2);         // Buffer set to start back
          // Response string is now in the receive buffer, example: "VNCFILE2.TXT $52 $00 $00
$00 D:\>"
          // char *strchr(const char *s, int c); it locates the first occurrence of c in the string
pointed to by s.
          //        it returns a pointer to the byte, or a null pointer if the byte was not found.
    start = strchr(file_content,'$'); // find the index of the first occurance of $ (starting from LSB)
          *file_length = str_to_int(start); // convert hex string to a number (currently up to 64
kByte!)
          if (!(*file_length)) return 51;             // error, file is too long or has length of 0
    return 0;                                                                    // Ok, valid file length
  }

  return status;// error number (while 0 = OK)
}

// ***********************************************************************
// VNC1L - Generating a binary file on a USB drive
// (It can also write a text file.)
// Maximum file length: 64 kByte
// Parameters: file_name   : Address of file_name
//             file_content : Address of data block
//             file_length    : nnumber of bytes
// Return: status                    : error number,  0 = OK
// ***********************************************************************
unsigned char vnc_wr_binfile(char *file_name, char *file_content, unsigned int file_length)
{
        unsigned int vnc_i              = 0;
        unsigned char wr_steps          = 0;
        unsigned char status   = 0;

  if (file_length == 0xFFFF) return 51;  // error, file is too long

  do {
   if (wr_steps == 4) return 0; // done, OK (the last prompt character exists)

   switch (wr_steps) // write process
        {
    case 0:  // Open or create a file
         sprintf (&vnc.send_buffer[0], "OPW %s", file_name);
         if (vnc_wr_cmd(vnc.send_buffer)) return 1;// Error, command was not accepted
         break;
    case 1:  // Seek offset byte position of 0 in the open file (you can also choose other offset)
         if (vnc_wr_cmd("SEK 0")) return 1;                    // Error, command was not accepted
         break;
    case 2:  // Write the number of bytes to an open file
         sprintf (&vnc.send_buffer[0], "WRF %d", file_length);
         if (vnc_wr_cmd(vnc.send_buffer)) return 1;// Error, command was not accepted
         // No check for buffer limit, if necessary, expand it
         for(vnc_i = 0; vnc_i < file_length; vnc_i++)
                        {
          if (vnc_wr_byte(*file_content)) return 1;   // Error, command was not accepted
          file_content++;
         }
         break;
    case 3:  // close file
         sprintf (&vnc.send_buffer[0], "CLF %s", file_name);
         if (vnc_wr_cmd(vnc.send_buffer))  return 1;           // Error, command was not accepted
         break;
   } // Switch
   wr_steps++; // next step
        status = vnc_rd_answer(PROMPT_AND_ERROR_CHECK);
  } while (status == PROMPT_OK);       // Testing for next prompt character >
  return status;// error number (while 0 = OK)
}
```

2. VNC1L_keyboard (SPI, polling)

****************** *VNC1L_keyboard.c* *******************

```
// Original project by Matthias Kahnt az51@gmx.net
// Modified by Terry Young Kim
// ATmega1284
// VNC1L/VDIP1 Vinculum (Port2)
// VNC1L - USB HOST CONTROLLER (Mega1284: Master, VNC1L: Slave)
// VDAP firmware version 3.68 or 3.69
// SPI interface
// VDIP1: Jumper-Set: J3 - Pulldown / J4 - Pullup for SPI

#include <avr/io.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <avr/interrupt.h>
#include <util/delay.h>     // for LCD

#include "uart.c"  // UART
#include "lcd_lib.c"  // LCD
#include "firmware.c" // VNC1L APIs

#define F_CPU 16000000UL

//timeout value for a task
#define t1 50 // 50ms works fine

// LED: PB0
// LCD: PORT C

// VNC: SCLK PA3
//      SDI  PA4
//      SDO  PA5
//      CS   PA6
//      RS   PA7

// UART file descriptor
// putchar and getchar are in uart.c
FILE uart_str = FDEV_SETUP_STREAM(uart_putchar, uart_getchar, _FDEV_SETUP_RW);

const int8_t LCD_initialize[] PROGMEM = "LCD Initialized\0";
const int8_t LCD_VNC1L[] PROGMEM = "VNC1L -Test-\0";
const int8_t LCD_mega1284[] PROGMEM = "AVR Atmega1284\0";
const int8_t LCD_success[] PROGMEM = "VNC1L success \0";

int8_t lcd_buffer[17]; // LCD display buffer
```

```
volatile unsigned int time1;    //timeout counter
volatile int test[16];
volatile int pre = 0; // previous key stroke
volatile int pre_special = 0; // previous SHIFT, CONTROL key stroke

//the task subroutine
void task1(void);

//**********************************************************
//timer 0 compare ISR
ISR (TIMER0_COMPA_vect)
{
  //Decrement the time if it is not already zero
  if (time1>0)    --time1;
}

//**********************************************************
// LCD setup
void init_lcd(void)
{
        LCDinit();        //initialize the display
        LCDcursorOFF();
        LCDclr();                             //clear the display
        LCDGotoXY(0,0);
        CopyStringtoLCD(LCD_initialize, 0, 0);
}

// system initialization
void init_system (void)
{
  // LED for debugging
  LED_DDR |= (1 << LED_PIN); // output
  LED_D1_OFF; // turn off LED

  // VNC1L SPI interface setup
  SPI_DDR |= ((1 << SCLK_PIN) | (1 << SDI_PIN) | (1 << CS_PIN) | (1 << RE_PIN)); // outputs
  SPI_DDR &= ~(1 << SDO_PIN);         // input

    //init the UART -- uart_init() is in uart.c
  uart_init();
  stdout = stdin = stderr = &uart_str;
  fprintf(stdout,"\n\r***** Starting VNC1L USB keyboard test... *****\n\r");

  //crank up the ISRs
  sei();
}

// *********************************************************************
// List of errors
// *********************************************************************
//       1        = Command was not accepted by VNC1L
//       2        = Command Failed
//       3        = Bad Command
//       4        = Disk Full
//       5        = Filename Invalid
//       6        = Read Only
//       7        = File Open
//       8        = Directory Not Empty
//       9        = No Disk
//       20       = No VNC1L with VDAP firmware available
//       21       = No USB drive plugged
//       50       = Buffer is too small to read a file (buffer overflow)
//       51       = File length = 0 or > 64kByte
//       99       = Timeout
// *********************************************************************
void vnc_error_message(unsigned char error_type, unsigned char error_num)
{
  LED_D1_ON;  // LED turns on if there's an error

    fprintf(stdout,"\n\rError type: %d,    Error number: %d\n\r", error_type, error_num);

  // error occured
  while(1); // error keeps the system hanging
}

// ******************************************************************
// Main program
// ******************************************************************
int main (void)
{
  init_system();

  // start the LCD
  init_lcd();
  LCDclr();

  // initialize VNC1L
  vnc_init();
    //set up timer 0 for 1 mSec timebase
  TIMSK0= (1<<OCIE0A);          //turn on timer 0 cmp match ISR
  OCR0A = 249;                  //set the compare reg to 250 time ticks
  //set prescalar to divide by 64
  TCCR0B= 3;
  // turn on clear-on-match
  TCCR0A= (1<<WGM01) ;

  // put some stuff on LCD
  CopyStringtoLCD(LCD_VNC1L, 0, 0);//start at char=0 line=0
  CopyStringtoLCD(LCD_mega1284, 0, 1);//start at char=0 line=1

  char count, x;

  // Check for the presence of VDAP firmware on VNC1L
  if (vnc_wait_for("VDAP")) vnc_error_message(20,1);                    // check confirmation

    // Wait until a USB drive is plugged and detected
  if (vnc_wait_for("Device Detected")) vnc_error_message(21,2);    // check confirmation

  // Wait for the first prompt that monitor returns (D:\>)
  // for HID devices, do not run prompt check here
//  vnc.status = vnc_prompt_check();                              // check confirmation
  if (vnc.status) vnc_error_message(vnc.status,3);     // error message

    /* get the length of the string and it's last character index */
    count = strlen(vnc.receive_buffer) - 1;

         for(x=count; x >= 0; x--)
         {
                         if(vnc.receive_buffer[x] == '\r')
                         {
                                 fprintf(stdout, "\n", vnc.receive_buffer[x]);
                         fprintf(stdout, "%c", vnc.receive_buffer[x]);
                         }
                         else
                         fprintf(stdout, "%c", vnc.receive_buffer[x]);

                         if(x == 0) break;
         }

  // run IPA command before running ECS command
  // Monitor commands in ASCII: IPA mode
```

```
  vnc.status = vnc_wr_cmd("IPA");
  if (vnc.status) vnc_error_message(1,5);                    // error: command was not accepted
by VNC1L
  vnc.status = vnc_prompt_check();                                   // check if monitor returns
prompt
  if (vnc.status) vnc_error_message(vnc.status,5);

  // Extented Command Set
  vnc.status = vnc_wr_cmd("ECS");
  if (vnc.status) vnc_error_message(1,4);                    // error: command was not accepted
by VNC1L
  vnc.status = vnc_prompt_check();                                   // check if monitor returns
prompt
  if (vnc.status) vnc_error_message(vnc.status,4);
  // monitor returns prompt if command executed correctly

  // Display firmware version
  vnc.status = vnc_wr_cmd("FWV");
  if (vnc.status) vnc_error_message(1,6);                    // error: command was not accepted
by VNC1L
  vnc.status = vnc_prompt_check();                                   // check if monitor returns
prompt
  if (vnc.status) vnc_error_message(vnc.status,6);

  count = strlen(vnc.receive_buffer) - 1;

        for(x=count; x >= 0; x--)
        {
                      if(vnc.receive_buffer[x] == '\r')
                      {
                             fprintf(stdout, "\n", vnc.receive_buffer[x]);
                      fprintf(stdout, "%c", vnc.receive_buffer[x]);
                      }
                      else
                      fprintf(stdout, "%c", vnc.receive_buffer[x]);

                      if(x == 0) break;
        }

// To print what monitor returns:
//  fprintf(stdout,"\n\r%s\n\r", vnc.receive_buffer);
// note: All output from the command monitor is LSB first
//      If monitor return multiple lines of data, they are followed by carriage return,
//      which overwrites a previous line on fprintf
```

```
  // Query port 2 for HID
  vnc.status = vnc_wr_cmd("QP2");
  if (vnc.status) vnc_error_message(1,7);                      // error: command was not accepted
by VNC1L
  vnc.status = vnc_prompt_check();                                       // check if monitor returns
prompt
  if (vnc.status) vnc_error_message(vnc.status,7);
  // monitor returns prompt if command executed correctly

  fprintf(stdout,"\n\r%s\n\r", vnc.receive_buffer);

  // Query Device
  vnc.status = vnc_wr_cmd("QD 0");
  if (vnc.status) vnc_error_message(1,8);                      // error: command was not accepted
by VNC1L
  vnc.status = vnc_prompt_check();                                       // check if monitor returns
prompt
  if (vnc.status) vnc_error_message(vnc.status,8);
  // monitor returns prompt if command executed correctly

        count = strlen(vnc.receive_buffer) - 1;

        for(x=count; x >= 0; x--)
        {
                        if(vnc.receive_buffer[x] == '\r')
                        {
                                fprintf(stdout, "\n", vnc.receive_buffer[x]);
                        fprintf(stdout, "%c", vnc.receive_buffer[x]);
                        }
                        else
                        fprintf(stdout, "%c", vnc.receive_buffer[x]);

                        if(x == 0) break;
        }

  // Set current device
  vnc.status = vnc_wr_cmd("SC 0");
  if (vnc.status) vnc_error_message(1,9);                      // error: command was not accepted
by VNC1L
  vnc.status = vnc_prompt_check();                                       // check if monitor returns
prompt
  if (vnc.status) vnc_error_message(vnc.status,9);
  // monitor returns prompt if command executed correctly
```

```
    fprintf(stdout,"\n\r%s\n\r", vnc.receive_buffer);

// Device Send Setup Data - Set Idle
  vnc.status = vnc_wr_cmd("SSU $210A000000000000");
  // no interrupt transfer unless there is a change in the keycode
  // $210A007D00000000 => interrupt transfer every 500ms (500ms/4ms = 0x7D)
  if (vnc.status) vnc_error_message(1,120);
  vnc.status = vnc_prompt_check();                          // check if monitor returns
prompt
  if (vnc.status) vnc_error_message(vnc.status,120);

        CopyStringtoLCD(LCD_success, 0, 1);//start at char=0 line

    fprintf(stdout,"\n\rStart polling!\n\r");

  // If you reached here, then no error has occured
  // LED blinks every second
  // main task scheduler loop
  while(1)
  {
        // reset time and call task
    if (time1==0){ time1=t1; task1();}

//    LED_D1_ON;
//    _delay_ms(1000); // 1 sec
//    LED_D1_OFF;
//    _delay_ms(1000);
  }
}

//*******************************
//Task subroutine: Task 1
void task1(void)
{
  // Device Send Setup Data - Get Report
  vnc.status = vnc_wr_cmd("SSU $A101000100000800"); // monitor returns 8 bytes
  if (vnc.status) vnc_error_message(1,12);

        for(int t=0; t < 23; t++) // this includes prompt check as well as a report
        {
                vnc_rd_spi(0);
                test[t] = (int)vnc.receive_buffer[0];
```

```
	//	fprintf(stdout, "\n%d", test[t]);
	//	fprintf(stdout, "\n%c", vnc.receive_buffer[0]);
	}

	// only one key press is checked here, but you can check 5 more key presses
	if((pre != test[12]) && (test[12] != NO_KEY_PRESSED))
		fprintf(stdout, "%c", print_keystoke(test[12]));

	if((pre_special != test[10]) && (test[10] != 0))
	{
		switch (test[10])
		{
			case KEY_LEFT_CTRL:
				fprintf(stdout, "\n\rLeft CTRL\n\r");
				break;

			case KEY_LEFT_SHIFT:
				fprintf(stdout, "\n\rLeft SHIFT\n\r");
				break;

			case KEY_RIGHT_CTRL:
				fprintf(stdout, "\n\rRight CTRL\n\r");
				break;

			case KEY_RIGHT_SHIFT:
				fprintf(stdout, "\n\rRight SHIFT\n\r");
				break;
		}
	}

	pre = test[12];
	pre_special = test[10];
}
```

3. VNC1L_mouse (SPI, polling)

****************** *VNC1L_mouse.c* *******************

```
// Original project by Matthias Kahnt az51@gmx.net
// Modified by Terry Young Kim
// ATmega1284
// VNC1L/VDIP1 Vinculum (Port2)
// VNC1L - USB HOST CONTROLLER (Mega1284: Master, VNC1L: Slave)
// VDAP firmware version 3.68 or 3.69
// SPI interface
// VDIP1: Jumper-Set: J3 - Pulldown / J4 - Pullup for SPI

#include <avr/io.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <avr/interrupt.h>
#include <util/delay.h>     // for LCD

#include "uart.c"  // UART
#include "lcd_lib.c"  // LCD
#include "firmware.c" // VNC1L APIs

#define F_CPU 16000000UL

//timeout value for a task
#define t1 50 // 50ms works fine

volatile unsigned int time1;    //timeout counter

//the task subroutine
void task1(void);

//**********************************************************
//timer 0 compare ISR
ISR (TIMER0_COMPA_vect)
{
  //Decrement the time if it is not already zero
  if (time1>0)    --time1;
}

// LED: PB0
// LCD: PORT C

// VNC: SCLK PA3
//      SDI  PA4
//      SDO  PA5
//      CS   PA6
//      RS   PA7

// UART file descriptor
// putchar and getchar are in uart.c
FILE uart_str = FDEV_SETUP_STREAM(uart_putchar, uart_getchar, _FDEV_SETUP_RW);
```

```
const int8_t LCD_initialize[] PROGMEM = "LCD Initialized\0";
const int8_t LCD_VNC1L[] PROGMEM = "VNC1L -Test-\0";
const int8_t LCD_mega1284[] PROGMEM = "AVR Atmega1284\0";
const int8_t LCD_success[] PROGMEM = "VNC1L success \0";

int8_t lcd_buffer[17]; // LCD display buffer

volatile int test[16];
volatile int pre = 0;
volatile int pre_x = 0;
volatile int pre_y = 0;
volatile int pre_wheel = 0;

//**********************************************************
// LCD setup
void init_lcd(void)
{
	LCDinit();	//initialize the display
	LCDcursorOFF();
	LCDclr();				//clear the display
	LCDGotoXY(0,0);
	CopyStringtoLCD(LCD_initialize, 0, 0);
}

// system initialization
void init_system (void)
{
  // LED for debugging
  LED_DDR |= (1 << LED_PIN); // output
  LED_D1_OFF; // turn off LED

  // VNC1L SPI interface setup
  SPI_DDR |= ((1 << SCLK_PIN) | (1 << SDI_PIN) | (1 << CS_PIN) | (1 << RE_PIN)); // outputs
  SPI_DDR &= ~(1 << SDO_PIN);	// input

   //init the UART -- uart_init() is in uart.c
  uart_init();
  stdout = stdin = stderr = &uart_str;
  fprintf(stdout,"\n\r***** Starting VNC1L USB mouse test... *****\n\r");

  //crank up the ISRs
  sei();
}
```

```
// ***********************************************************************
// List of errors
// ***********************************************************************
//      1       = Command was not accepted by VNC1L
//      2       = Command Failed
//      3       = Bad Command
//      4       = Disk Full
//      5       = Filename Invalid
//      6       = Read Only
//      7       = File Open
//      8       = Directory Not Empty
//      9       = No Disk
//      20      = No VNC1L with VDAP firmware available
//      21      = No USB drive plugged
//      50      = Buffer is too small to read a file (buffer overflow)
//      51      = File length = 0 or > 64kByte
//      99      = Timeout
// ***********************************************************************
void vnc_error_message(unsigned char error_type, unsigned char error_num)
{
 LED_D1_ON;  // LED turns on if there's an error

   fprintf(stdout,"\n\rError type: %d,    Error number: %d\n\r", error_type, error_num);

 // error occured
 while(1); // error keeps the system hanging
}

// ********************************************************************
// Main program
// ********************************************************************
int main (void)
{
 init_system();

 // start the LCD
 init_lcd();
 LCDclr();

 // initialize VNC1L
 vnc_init();

   //set up timer 0 for 1 mSec timebase
 TIMSK0= (1<<OCIE0A);          //turn on timer 0 cmp match ISR
 OCR0A = 249;                  //set the compare reg to 250 time ticks
 //set prescalar to divide by 64
 TCCR0B= 3;
 // turn on clear-on-match
 TCCR0A= (1<<WGM01) ;

 // put some stuff on LCD
 CopyStringtoLCD(LCD_VNC1L, 0, 0);//start at char=0 line=0
 CopyStringtoLCD(LCD_mega1284, 0, 1);//start at char=0 line=1

 char count, x;

 // Check for the presence of VDAP firmware on VNC1L
 if (vnc_wait_for("VDAP")) vnc_error_message(20,1);                    // check confirmation

  // Wait until a USB drive is plugged and detected
 if (vnc_wait_for("Device Detected")) vnc_error_message(21,2);    // check confirmation

 // Wait for the first prompt that monitor returns (D:\>)
 // for HID devices, do not run prompt check here
//  vnc.status = vnc_prompt_check();                              // check confirmation
 if (vnc.status) vnc_error_message(vnc.status,3);     // error message

  /* get the length of the string and it's last character index */
  count = strlen(vnc.receive_buffer) - 1;

        for(x=count; x >= 0; x--)
        {
                        if(vnc.receive_buffer[x] == '\r')
                        {
                                fprintf(stdout, "\n", vnc.receive_buffer[x]);
                        fprintf(stdout, "%c", vnc.receive_buffer[x]);
                        }
                        else
                        fprintf(stdout, "%c", vnc.receive_buffer[x]);

                        if(x == 0) break;
        }

 // run IPA command before running ECS command
 // Monitor commands in ASCII: IPA mode
 vnc.status = vnc_wr_cmd("IPA");
```

```
  if (vnc.status) vnc_error_message(1,5);                    // error: command was not accepted
by VNC1L
  vnc.status = vnc_prompt_check();                                 // check if monitor returns
prompt
  if (vnc.status) vnc_error_message(vnc.status,5);

  // Extented Command Set
  vnc.status = vnc_wr_cmd("ECS");
  if (vnc.status) vnc_error_message(1,4);                    // error: command was not accepted
by VNC1L
  vnc.status = vnc_prompt_check();                                 // check if monitor returns
prompt
  if (vnc.status) vnc_error_message(vnc.status,4);
  // monitor returns prompt if command executed correctly

  // Display firmware version
  vnc.status = vnc_wr_cmd("FWV");
  if (vnc.status) vnc_error_message(1,6);                    // error: command was not accepted
by VNC1L
  vnc.status = vnc_prompt_check();                                 // check if monitor returns
prompt
  if (vnc.status) vnc_error_message(vnc.status,6);

  count = strlen(vnc.receive_buffer) - 1;

        for(x=count; x >= 0; x--)
        {
                      if(vnc.receive_buffer[x] == '\r')
                      {
                            fprintf(stdout, "\n", vnc.receive_buffer[x]);
                      fprintf(stdout, "%c", vnc.receive_buffer[x]);
                      }
                      else
                      fprintf(stdout, "%c", vnc.receive_buffer[x]);

                      if(x == 0) break;
        }

// To print what monitor returns:
//  fprintf(stdout,"\n\r%s\n\r", vnc.receive_buffer);
// note: All output from the command monitor is LSB first
//      If monitor return multiple lines of data, they are followed by carriage return,
//      which overwrites a previous line on fprintf
```

```
 // Query port 2 for HID
 vnc.status = vnc_wr_cmd("QP2");
 if (vnc.status) vnc_error_message(1,7);                    // error: command was not accepted
by VNC1L
 vnc.status = vnc_prompt_check();                                  // check if monitor returns
prompt
 if (vnc.status) vnc_error_message(vnc.status,7);
 // monitor returns prompt if command executed correctly

  fprintf(stdout,"\n\r%s\n\r", vnc.receive_buffer);

 // Query Device
 vnc.status = vnc_wr_cmd("QD 0");
 if (vnc.status) vnc_error_message(1,8);                    // error: command was not accepted
by VNC1L
 vnc.status = vnc_prompt_check();                                  // check if monitor returns
prompt
 if (vnc.status) vnc_error_message(vnc.status,8);
 // monitor returns prompt if command executed correctly

        count = strlen(vnc.receive_buffer) - 1;

        for(x=count; x >= 0; x--)
        {
                        if(vnc.receive_buffer[x] == '\r')
                        {
                                fprintf(stdout, "\n", vnc.receive_buffer[x]);
                        fprintf(stdout, "%c", vnc.receive_buffer[x]);
                        }
                        else
                        fprintf(stdout, "%c", vnc.receive_buffer[x]);

                        if(x == 0) break;
        }

 // Set current device
 vnc.status = vnc_wr_cmd("SC 0");
 if (vnc.status) vnc_error_message(1,9);                    // error: command was not accepted
by VNC1L
 vnc.status = vnc_prompt_check();                                  // check if monitor returns
prompt
 if (vnc.status) vnc_error_message(vnc.status,9);
 // monitor returns prompt if command executed correctly
```

```
    fprintf(stdout,"\n\r%s\n\r", vnc.receive_buffer);

// Device Send Setup Data - Set Idle
  vnc.status = vnc_wr_cmd("SSU $210A000000000000"); // no interrupt unless there's a change
of event
  // $210A007D00000000 (500ms/4ms = 0x7D)
  if (vnc.status) vnc_error_message(1,120);
  vnc.status = vnc_prompt_check();                          // check if monitor returns
prompt
  if (vnc.status) vnc_error_message(vnc.status,120);

            CopyStringtoLCD(LCD_success, 0, 1);

    fprintf(stdout,"\n\rStart polling!\n\r");

  // If you reached here, then no error has occured
  // LED blinks every second
   //main task scheduler loop
  while(1)
  {
            // reset time and call task
    if (time1==0){ time1=t1; task1(); }

//    LED_D1_ON;
//    _delay_ms(1000); // 1 sec
//    LED_D1_OFF;
//    _delay_ms(1000);
  }
}

//********************************
//Task subroutine: Task 1
void task1(void)
{
// Device Send Setup Data - Get Report
  vnc.status = vnc_wr_cmd("SSU $A101000100000400"); // monitor return 4 bytes
  if (vnc.status) vnc_error_message(1,12);

            for(int t=0; t < 19; t++)   // this includes prompt check as well as report
            {
                        vnc_rd_spi(0);
                        test[t] = (int)vnc.receive_buffer[0];

            //          fprintf(stdout, "\n%d", test[t]);
```

```
	//	fprintf(stdout, "\n%c", vnc.receive_buffer[0]);
	}

	if((pre != test[10]) && test[10] != 0) // buttons
	{
		if(test[10] == 1)
			fprintf(stdout, "\n\rLeft click");
		else if(test[10] == 2)
			fprintf(stdout, "\n\rRight click");
		else if(test[10] == 4)
			fprintf(stdout, "\n\rWheel");
		//else
		//	fprintf(stdout, "\n\rNo movement\n\r");
	}

	if((pre_x != test[11]) && test[11] != 0)  // X coordinate
	{
		fprintf(stdout, "\n\rX = %d", test[11]);
	}

	if((pre_y != test[12]) && test[12] != 0)  // Y coordinate
	{
		fprintf(stdout, "\n\rY = %d", test[12]);
	}

	if((pre_wheel != test[13]) && test[13] != 0)  // wheel rotation
	{
		fprintf(stdout, "\n\rWheel = %d", test[13]);
	}

	pre = test[10];
	pre_x = test[11];
	pre_y = test[12];
	pre_wheel = test[13];
}
```

4. VNC1L_USB_drive_UART

****************** *VNC1L_USB_UART.c* *******************

```
// Original project by Matthias Kahnt az51@gmx.net
// Modified by Terry Young Kim
// ATmega1284
// VNC1L/VDIP1 Vinculum (Port2)
// VNC1L - USB HOST CONTROLLER (Mega1284: Master, VNC1L: Slave)
// VDAP firmware version 3.69
// USART interface
// VDIP1: Jumper-Set: J3 - Pullup / J4 - Pullup for UART
// VNC1L: connect CTS (AD3) to ground (pull down low) or connect CTS (AD3) and RTS (AD2)

// WARNING: if the prompt check is carried out properly, the program will hang!
// A file name must be at least two characters long
// otherwise, vnc_rd_dir_UART() won't work.

#include <avr/io.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <avr/interrupt.h>
#include <util/delay.h>     // for LCD

#include "uart.c"  // UART channel 0 ==> PuTTy
#include "uart1.c"  // UART channel 1 <==> VNC1L
#include "lcd_lib.c"  // LCD
#include "firmware.c" // VNC1L APIs

#define F_CPU 16000000UL

// LED: PB0
// LCD: PORT C

// VNC: TX         PD3
//      RX    PD4

// UART file descriptor
// putchar and getchar are in uart.c
FILE uart_str = FDEV_SETUP_STREAM(uart_putchar, uart_getchar, _FDEV_SETUP_RW);

const int8_t LCD_initialize[] PROGMEM = "LCD Initialized\0";
const int8_t LCD_VNC1L[] PROGMEM = "VNC1L -Test-\0";
const int8_t LCD_mega1284[] PROGMEM = "AVR Atmega1284\0";
const int8_t LCD_success[] PROGMEM = "VNC1L success \0";

int8_t lcd_buffer[17]; // LCD display buffer

// Textfile 1
char *file1[] = {
        "VNC1L USB Drive test\n\r",
   "It can write to and read from a USB flash drive.\n\r",
          "Text file 1 VNC1L VDIP1\n\r\a"};      // \a is the EOF (end of file) // \0"}; //

// Textfile 2
char *file2[] = {
          "Text file 2,\n\r",
          "Another text file to be created.\n\r\a"};

// Textfile 3
char *file3[] = {"Text file 3 can be appended to an existing file.\n\r\a"};

// Bin file
char file4[] = {
          0xFA,0x89,0x11,0xFF,0xAA,0x02,0x00,0x1F,
          0x1D,0xBA,0xA6,0x24,0x99,0x2D,0x1E,0x1F,
          0x0D,0x0C,0xA0,0x55,0x30,0x1F,0x16,0xEF,
          0x04,0x22,0x00,0x00,0xCF,0x12,0x5F,0xDF,
          0x69,0x00,0x00,0x1F,0xEF,0x15,0x3F,0xCF,
          0x5F,0x44,0x1F,0x1F,0xBF,0x16,0x13,0x7F,
          0x00,0x10,0x1F,0xDF,0x1F,0x8F,0x16,0xDF,
          0x1F,0x32,0x3F,0x56,0x1F,0x19,0x17,0xFF,
          0x3F,0x42,0xFE,0x46,0x6F,0x99,0x10,0xAF,
          0x6F,0x62,0x36,0x66,0x10,0xAA,0xBB,0xCC };

// Textfile 5
char *file5[] = {"MacGyver left the building. He ain't coming back.\n\r\a"};

char file_buffer[FILE_BUFFER_SIZE];  // buffer for testing
unsigned int file_length;                 // byte length of a file (for reading)

volatile char my_text[50];
volatile char my_str[50];
volatile char read_str[50];

//*********************************************************
// LCD setup
void init_lcd(void)
{
          LCDinit();        //initialize the display
          LCDcursorOFF();
          LCDclr();                               //clear the display
          LCDGotoXY(0,0);
          CopyStringtoLCD(LCD_initialize, 0, 0);
}
```

```
// system initialization
void init_system (void)
{
  // LED for debugging
  LED_DDR |= (1 << LED_PIN); // output
  LED_D1_OFF; // turn off LED

     // 8-bit, no parity, asynchronous UART, 1 stop bit
     UCSR1C |= (1 << UCSZ11) | (1 << UCSZ10);

    //init the UART -- uart_init() is in uart.c
  uart_init();
  uart_init1();
  stdout = stdin = stderr = &uart_str;
  fprintf(stdout,"\n\r***** Starting VNC1L USB drive UART test... *****\n\r");
}

// ***********************************************************************
// List of errors
// ***********************************************************************
//       1        = Command was not accepted by VNC1L
//       2        = Command Failed
//       3        = Bad Command
//       4        = Disk Full
//       5        = Filename Invalid
//       6        = Read Only
//       7        = File Open
//       8        = Directory Not Empty
//       9        = No Disk
//       20       = No VNC1L with VDAP firmware available
//       21       = No USB drive plugged
//       50       = Buffer is too small to read a file (buffer overflow)
//       51       = File length = 0 or > 64kByte
//       99       = Timeout
// ***********************************************************************
void vnc_error_message(unsigned char error_type, unsigned char error_num)
{
  LED_D1_ON;  // LED turns on if there's an error

    fprintf(stdout,"\n\rError type: %d,    Error number: %d\n\r", error_type, error_num);

  // error occured
  while(1); // error keeps the system hanging
```

```
}

// ******************************************************************
// Main program
// ******************************************************************
int main (void)
{
 init_system();

 // start the LCD
 init_lcd();
 LCDclr();

 // put some stuff on LCD
 CopyStringtoLCD(LCD_VNC1L, 0, 0);//start at char=0 line=0
 CopyStringtoLCD(LCD_mega1284, 0, 1);//start at char=0 line=1

	my_text[47] = '\n';
	my_text[48] = '\r';
	my_text[49] = '\a'; // '\a' marks EOF (end of file)
	// Regardless of the size of a buffer, the last character of the buffer should be '\a'

// To print what monitor returns:
//  fprintf(stdout,"\n\r%s\n\r", vnc.receive_buffer);
// note: All output from the command monitor is LSB first
//	If monitor return multiple lines of data, they are followed by carriage return,
//	which overwrites a previous line on fprintf

 // Wait for VNC1L to respond with firmware version and message if a device is detected
 vnc.status = vnc_wait_for_VNC1L_UART();
 if (vnc.status) vnc_error_message(vnc.status,5);

 // run IPA command before running ECS command
 // Monitor commands in ASCII: IPA mode
 vnc.status = vnc_wr_cmd_UART("IPA");
 if (vnc.status) vnc_error_message(1,5);			// error: command was not accepted
by VNC1L
 vnc.status = vnc_prompt_check_UART();				// check if monitor
returns prompt
 if (vnc.status) vnc_error_message(vnc.status,5);

 // Extented Command Set
 vnc.status = vnc_wr_cmd_UART("ECS");
```

```
  if (vnc.status) vnc_error_message(1,4);                              // error: command was not accepted
by VNC1L
  vnc.status = vnc_prompt_check_UART();                                                    // check if monitor
returns prompt
  if (vnc.status) vnc_error_message(vnc.status,4);
  // monitor returns prompt if command executed correctly

  // Display firmware version
  vnc.status = vnc_wr_cmd_UART("FWV");
  if (vnc.status) vnc_error_message(1,6);                              // error: command was not accepted
by VNC1L

  // print firmware version returned by VNC1L and prompt check
  vnc.status = vnc_firmware_UART();

  fprintf(stdout, "\n\rType something: ");
  fscanf(stdin, "%[^\n]s\n\r", my_text) ; // [^\n] allows user to type multiple words separated by
a space//          fscanf(stdin, "%[^\n]s\n\r", my_text) ; // [^\n] allows user to type multiple words
separated by a space

  // Write text file 1 to USB disk
  vnc.status = vnc_wr_txtfile_UART("VNCFILE1.TXT", file1[0], FILE_NEW);
  if (vnc.status) vnc_error_message(vnc.status,8);

  // Write text file 2 to USB disk
  vnc.status = vnc_wr_txtfile_UART("VNCFILE2.TXT", file2[0], FILE_NEW);
  if (vnc.status) vnc_error_message(vnc.status,9);

  // Append text string to existing text file 1
  vnc.status = vnc_wr_txtfile_UART("VNCFILE1.TXT", file3[0], FILE_APPEND);
  if (vnc.status) vnc_error_message(vnc.status,10);

  // Write a binary (bin) file to disk
  vnc.status = vnc_wr_binfile_UART("VNCFILE3.BIN", file4, 80); // file length = 80
  if (vnc.status) vnc_error_message(vnc.status,11);

  // Before reading a file, you must get the length of it
  vnc.status = vnc_rd_dir_UART("VNCFILE2.TXT", &file_length, 12); // file name length = 12
  if (vnc.status) vnc_error_message(vnc.status,12);
  // Now the file length is saved in file_length

  if (!vnc.status)
  {
    // Read a text file of known length into file_buffer
    vnc.status = vnc_rd_file_UART("VNCFILE2.TXT", file_buffer, file_length);
    if (vnc.status) vnc_error_message(vnc.status,13);
  }

  fprintf(stdout, "\n\r%s\n\r", file_buffer);

  // Write text file 5 to disk
//  vnc.status = vnc_wr_txtfile_UART("VNCFILE5.TXT", file5[0], FILE_NEW);
//  if (vnc.status) vnc_error_message(vnc.status,14);

  // Write text file 5 to disk using user input string
  vnc.status = vnc_wr_txtfile_UART("VNCFILE5.TXT", my_text, FILE_NEW);
  if (vnc.status) vnc_error_message(vnc.status,14);

    // Get the length of file
  vnc.status = vnc_rd_dir_UART("VNCFILE5.TXT", &file_length, 12);
  if (vnc.status) vnc_error_message(vnc.status,15);

  if (!vnc.status)
  {
    // Read a text file of known length into my_str
    vnc.status = vnc_rd_file_UART("VNCFILE5.TXT", my_str, file_length);
    if (vnc.status) vnc_error_message(vnc.status,16);
  }

  fprintf(stdout, "Echo back: \n\r%s\n\r", my_str);

  // Append a text string to an existing file
  vnc.status = vnc_wr_txtfile_UART("VNCFILE5.TXT", file5[0], FILE_APPEND);
  if (vnc.status) vnc_error_message(vnc.status,17);

/*
  // Append a text string to an existing file
  vnc.status = vnc_wr_txtfile("VNCFILE5.TXT", my_str, FILE_APPEND);
  // append twice for some reason?!?!?!?!?
  // my_str seems to cause the problem
  if (vnc.status) vnc_fehlermeldung(vnc.status,17);
*/

  // Append a text string to an existing file
  vnc.status = vnc_wr_txtfile_UART("VNCFILE5.TXT", my_text, FILE_APPEND);
  if (vnc.status) vnc_error_message(vnc.status,17);

        CopyStringtoLCD(LCD_success, 0, 1);
```

```
   fprintf(stdout,"\n\rDone!\n\r");

  // If you reached here, then no error has occured
  // LED blinks every second
   //main task scheduler loop
  while(1)
  {
    LED_D1_ON;
    _delay_ms(1000); // 1 sec
    LED_D1_OFF;
    _delay_ms(1000);
  }
}
```

****************** *firmware.h* *******************

```
#define LED_PIN          0
#define LED_PORT     PORTB
#define LED_DDR          DDRB

#define RTS_DDR          DDRA
#define RTS_PORT     PORTA
#define RTS_PIN          0

#define CTS_DDR          DDRA
#define CTS_PORT     PORTA
#define CTS_PIN          1

//**************************************************************************
//**************************************************************************
#define LED_D1_ON   LED_PORT &= ~(1<<LED_PIN) /* LED */
#define LED_D1_OFF  LED_PORT |=  (1<<LED_PIN)

#define PROMPT_CHECK                2
#define PROMPT_AND_ERROR_CHECK      1
#define PROMPT_OK                   1
#define FILE_NEW                    0
#define FILE_APPEND                 1
#define TIMEOUT_READ_WRITE          500            /* Number of read/write tests if
buffer is full */
#define TIMEOUT_COMMUNICATION       5              /* Timeout if no communication has
happened */
```

```
#define FILE_BUFFER_SIZE      100                  /* Buffer size is at least 100 */

****************** firmware.c *******************

#include "firmware.h" // defines macros
#define VNC_BUFFER_SIZE 160          /* Transmit/receive buffer size is at least 60 */
//  60

struct vnc_struc
{
        char  receive_buffer[VNC_BUFFER_SIZE];      // Receive buffer
        char send_buffer[VNC_BUFFER_SIZE];          // Send buffer
        unsigned char status;                       // Status
};

struct vnc_struc vnc; // instantiate vnc_struc

// ******************************************************************
// VNC1L - Sending a command
// Parameter: wr_cmd : command (address)
// Return: status    : 0 = command accepted by VNC1L
// ******************************************************************
unsigned char vnc_wr_cmd_UART(char *wr_cmd)
{
        unsigned char offset;  // vnc_offset, length of command

  for(offset = strlen(wr_cmd); offset; offset--)
  {
        while((UCSR1A & (1<<UDRE1)) == 0);
        UDR1 = *wr_cmd;
   wr_cmd++;
  }
  // send carrage return (a command to monitor ends with 0x0D)
  while((UCSR1A & (1<<UDRE1)) == 0);
  UDR1 = 0x0D;

  return 0;
}

// ******************************************************************
// VNC1L - Receive a string
// Wait until the monitor returns info from firmware (ex. Firmware Version, Device Detected)
// Return: status    : 0 = string was received from VNC1L
// ******************************************************************
```

```
unsigned char vnc_wait_for_VNC1L_UART()
{
 char count, x;

 // Wait for VNC1L to respond with firmware version
 // and message if a device is detected
        for(int z=0; z < 45; z++)
        {
                // wait until a byte has been received
                while((UCSR1A & (1<<RXC1)) == 0);
                vnc.receive_buffer[z] = UDR1;
                // printing in this loop could cause the program to break
//              fprintf(stdout, "\n\r%c", vnc.receive_buffer[z]);
        }

  // get the length of the string and it's last character index
   count = strlen(vnc.receive_buffer) - 1;

        for(x = 0; x < count; x++)
        {
                        if(vnc.receive_buffer[x] == '\r')
                        {
                                fprintf(stdout, "\n", vnc.receive_buffer[x]);
                        fprintf(stdout, "%c", vnc.receive_buffer[x]);
                        }
                        else
                        fprintf(stdout, "%c", vnc.receive_buffer[x]);
        }

        return 0;
}

// ***********************************************************************
// VNC1L - Get firmware version
// Return: 0 = Prompt received
// ***********************************************************************
unsigned char vnc_firmware_UART(void)
{
        int z;

 // get message from VNC1L and check prompt
        for(z = 0; z < 34; z++)
        {
                // wait until a byte has been received
			while((UCSR1A & (1<<RXC1)) == 0);
			vnc.receive_buffer[z] = UDR1;
//			fprintf(stdout, "\n\r%c", vnc.receive_buffer[z]);
		}

//  count = strlen(vnc.receive_buffer) - 1;

		for(z = 0; z < 34; z++) // for(x = 0; x < count; x++)
		{
				if(vnc.receive_buffer[z] == '\r')
				{
					fprintf(stdout, "\n", vnc.receive_buffer[z]);
				fprintf(stdout, "%c", vnc.receive_buffer[z]);
				}
				else
				fprintf(stdout, "%c", vnc.receive_buffer[z]);
		}

		return 0;
}

// *****************************************************************
// VNC1L - Waiting for PROMPT
// Return: 0 = Prompt received
// *****************************************************************
unsigned char vnc_prompt_check_UART(void)
{
		int done = 0;
		int rd_count = 10; //TIMEOUT_COMMUNICATION;  // TIMEOUT error / abort if
necessary
		int count = 0;

		while(rd_count)
		{
			if(!done)
			{
				// wait until a byte has been received
				while((UCSR1A & (1<<RXC1)) == 0);
				vnc.receive_buffer[count] = UDR1;
				// printing here could cause the program to break
				//fprintf(stdout, "\n\r%c", vnc.receive_buffer[count]);

				// prompt received
				if(vnc.receive_buffer[count] == '>')//0x0D') // D:\>
```

```
                              {
                                        //if(vnc.receive_buffer[count-1] == '>' &&
vnc.receive_buffer[count-2] == '0x5C' && vnc.receive_buffer[count-3] == ':' &&
vnc.receive_buffer[count-4] == 'D')
                                                  return 0;
                              }
                              count++;
                    }
                    rd_count--;
          }

          return 1; // no prompt, timeout
}

// *********************************************************************
// VNC1L - Creating a text file on a USB drive
// Maximum file length: 64 kByte
// Parameters: file_name    : address of file_name
//            file_content : address of text block to be written
//            option       : File 0 = new or 1 = append content
// Return: 0 = OK
// *********************************************************************
unsigned char vnc_wr_txtfile_UART(char *file_name, char *file_content, char option)
{
          unsigned int vnc_i              = 0;
          unsigned int file_length = 0;
          unsigned int file_CR    = 0; // carriage return
          char *file_ptr;
          unsigned char wr_steps          = 0;
          unsigned char status   = 0;

  // calculate number of bytes
  file_ptr = file_content;
  file_length = 0;
  file_CR  = 0;

  while (*file_ptr != '\a') // '\a' is an identifier for the end of text
  {
    if (*file_ptr) file_length++; else file_CR++;
    file_ptr++;
  }

          // open a file
  sprintf(&vnc.send_buffer[0], "OPW %s", file_name);
  status = vnc_wr_cmd_UART(vnc.send_buffer);
  status = vnc_prompt_check_UART();

  // Data is usually appended to the end of an existing file or a new empty file is created if it
doesn't exist
  // SEK command: move to an arbitrary point in a file
  if(!option) // new file
  {
              status = vnc_wr_cmd_UART("SEK 0");
              status = vnc_prompt_check_UART();
  }

            // write to a file
  sprintf (&vnc.send_buffer[0], "WRF %u", file_length);
  status = vnc_wr_cmd_UART(vnc.send_buffer);
//  status = vnc_prompt_check_UART(); // do not check for prompt here

   file_ptr = file_content;

   for(vnc_i = 0; vnc_i < (file_length + file_CR); vnc_i++)
   {
     if (*file_ptr)
            {
                        // wait until the last byte has been transmitted
                        while((UCSR1A & (1<<UDRE1)) == 0);
                        UDR1 = *file_ptr;
     }
            file_ptr++;
   }

   status = vnc_prompt_check_UART();

            // close a file and prompt check
  sprintf (&vnc.send_buffer[0], "CLF %s", file_name);
  status = vnc_wr_cmd_UART(vnc.send_buffer);
  status = vnc_prompt_check_UART();

  return 0;
}

// ***********************************************************************
// VNC1L - Reading a file of a specific length from an USB drive
// Length of the file should be known, so determine it beforehand with vnc_rd_dir!
// Parameters: file_name   : address of file_name
//          file_content : address of destination buffer where the read data will be stored
//          file_length    : number of bytes
// Return: 0 = OK
// ******************************************************************
unsigned char vnc_rd_file_UART(char *file_name, char *file_content, unsigned int file_length)
{
          unsigned char rd_steps          = 0;
          unsigned char status   = 0;

  if (file_length >= (FILE_BUFFER_SIZE - 1))
  {
    return 50;     // error, buffer is too small
  }

  // Open file for reading
  sprintf (&vnc.send_buffer[0], "OPR %s", file_name);
  status = vnc_wr_cmd_UART(vnc.send_buffer);
  status = vnc_prompt_check_UART();

  // Seek offset byte position of 0 in the open file (you can also choose other offset)
  status = vnc_wr_cmd_UART("SEK 0");
  status = vnc_prompt_check_UART();

  // Read the number of bytes from an open file
  sprintf (&vnc.send_buffer[0], "RDF %d", file_length);
  status = vnc_wr_cmd_UART(vnc.send_buffer);
  status = vnc_prompt_check_UART();

  while (file_length)
  {
                    // wait until a byte has been received
                    while((UCSR1A & (1<<RXC1)) == 0);
                    *file_content = UDR1;
                    file_content++;
      file_length--;
  }
  // The characters read are now in the buffer and can be evaluated.

  // close file
  sprintf (&vnc.send_buffer[0], "CLF %s", file_name);
  status = vnc_wr_cmd_UART(vnc.send_buffer);
  status = vnc_prompt_check_UART();

 return 0; // 0 = OK
```

```
}

// *********************************************************************
// VNC1L - Convert hex string to a number
// Example: "$11 $22 $33 $44 D:\>" --> 0x44332211
// Maximum file length: 64 kByte (easily expandable to 32-bit)
// Parameter: value   : Address of hex string
// Return: file_length (16 bit),  0 = error
// *********************************************************************
unsigned int str_to_int_UART(char *value)
{
        unsigned char temp[4];
        unsigned char position = 0;
        unsigned char nibble = 0;
        unsigned char error  = 0;
        unsigned char mid_byte = 0;

  for (position = 4; position; position--)
  {
    if ( (*value == '0') && ( *(value + 1) == '$') ) value += 2;                  // skip $00
    if (*value == '$') value += 1;                                                           // skip
$
    mid_byte = 4;
    temp[position - 1] = 0;

    for (nibble = 2; nibble; nibble--)
        {
      if ((*value >= 'a') && (*value <= 'f')) *value = *value - 0x20;        // Convert to uppercase
      if ((*value >= 'A') && (*value <= 'F')) *value = *value - 0x07;       // normalize (hex based)
      if ((*value < '0') || (*value > '?')) error = 1;                                     // error
      temp[position - 1] |= (*value - 0x30) << mid_byte; // string to a number (subtract 0x30 from
a character 0-9)
      mid_byte -= 4; // next lower 4 bits out of a byte
      value++;
    }
    value++;
  }

  if ((temp[1] != 0) || (temp[0] != 0 )) error = 1; // error > 16-bit
  if (error)
  {
    // *(value - 16) = 0x00; // No response necessary via buffer
    temp[2] = 0;
        temp[3] = 0;
```

```
 }

 return temp[2] * 256 + temp[3];
}

// *****************************************************************
// VNC1L - DIR determines file length
// Maximum 64 kByte!!!
// Parameters: file_name   : Address of file_name
//            file_content : Address of the intermediate stores (temporary use)
//            file_length    : Address of the length of the file to be played back
// Return: 0 = OK
// *****************************************************************
unsigned char vnc_rd_dir_UART(char *file_name, unsigned int *file_length, unsigned int
file_name_length)
{
        unsigned char status              = 0;
        unsigned int buff_index                 = 0;
        char *start;                                        // Address of hex strings (beginning)
        int vnc_i;

  sprintf (&vnc.send_buffer[0], "DIR %s", file_name);
  status = vnc_wr_cmd_UART(vnc.send_buffer);
  //status = vnc_prompt_check_UART();

  // clean up the buffer
  for(int k = 0; k < 60; k++)
  {
                vnc.receive_buffer[k] = 0;
  }

   // DIR file_name command returns two CRs, the file name followed by $xx $xx $xx $xx CR and
a prompt
  for(vnc_i = 0; vnc_i < (file_name_length + 22); vnc_i++) // file name length + 18 + 4
  {
         // wait until a byte has been received
         while((UCSR1A & (1<<RXC1)) == 0);
         vnc.receive_buffer[vnc_i] = UDR1;
  }
  // Response string to DIR is read

        // char *strchr(const char *s, int c); it locates the first occurrence of c in the string
pointed to by s.
        //        it returns a pointer to the byte, or a null pointer if the byte was not found.
   start = strchr(vnc.receive_buffer,'$'); // find the index of the first occurance of $ (starting
from LSB)
          *file_length = str_to_int_UART(start); // convert hex string to a number (currently up to
64k bytes!)

  return 0; // 0 = OK
}

// ******************************************************************
// VNC1L - Generating a binary file on a USB drive
// (It can also write a text file.)
// Maximum file length: 64 kByte
// Parameters: file_name   : Address of file_name
//             file_content : Address of data block
//             file_length    : nnumber of bytes
// Return: 0 = OK
// ******************************************************************
unsigned char vnc_wr_binfile_UART(char *file_name, char *file_content, unsigned int
file_length)
{
          unsigned int vnc_i                  = 0;
          unsigned char wr_steps         = 0;
          unsigned char status   = 0;

  if (file_length == 0xFFFF) return 51;  // error, file is too long

          // open a file
  sprintf(&vnc.send_buffer[0], "OPW %s", file_name);
  status = vnc_wr_cmd_UART(vnc.send_buffer);
  status = vnc_prompt_check_UART();

  // Data is usually appended to the end of an existing file or a new empty file is created if it
doesn't exist
  // SEK command: move to an arbitrary point in a file
  status = vnc_wr_cmd_UART("SEK 0");
  status = vnc_prompt_check_UART();

          // write to a file
  sprintf (&vnc.send_buffer[0], "WRF %d", file_length);
  status = vnc_wr_cmd_UART(vnc.send_buffer);
//  status = vnc_prompt_check_UART();        // do not check for prompt here

   for(vnc_i = 0; vnc_i < file_length; vnc_i++)
   {
```

```
                    // wait until the last byte has been transmitted
                    while((UCSR1A & (1<<UDRE1)) == 0);
                    UDR1 = *file_content;
                    file_content++;
  }

  status = vnc_prompt_check_UART();

          // close a file and prompt check
  sprintf (&vnc.send_buffer[0], "CLF %s", file_name);
  status = vnc_wr_cmd_UART(vnc.send_buffer);
  status = vnc_prompt_check_UART();

  return 0;
}
```

5. VNC1L_keyboard_UART_polling

****************** *VNC1L_keyboard_UART.c* *******************

```
// Original project by Matthias Kahnt az51@gmx.net
// Modified by Terry Young Kim
// ATmega1284
// VNC1L/VDIP1 Vinculum (Port2)
// VNC1L - USB HOST CONTROLLER (Mega1284: Master, VNC1L: Slave)
// VDAP firmware version 3.69
// USART interface
// VDIP1: Jumper-Set: J3 - Pullup / J4 - Pullup for UART
// VNC1L: connect CTS (AD3) to ground (pull down low) or connect CTS (AD3) and RTS (AD2)

// WARNING: if the prompt check is carried out properly, the program will hang!

#include <avr/io.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <avr/interrupt.h>
#include <util/delay.h>     // for LCD

#include "uart.c"  // UART channel 0 ==> PuTTy
#include "uart1.c"  // UART channel 1 <==> VNC1L
#include "lcd_lib.c"  // LCD
#include "firmware.c" // VNC1L APIs
```

```
#define F_CPU 16000000UL

#define KEYBOARD     22
#define FIRST_KEY    12
#define SPECIAL_KEY  10

//timeout value for a task
#define t1 50 // 50ms works fine

// LED: PB0
// LCD: PORT C

// VNC: TX       PD3
//      RX    PD4

// UART file descriptor
// putchar and getchar are in uart.c
FILE uart_str = FDEV_SETUP_STREAM(uart_putchar, uart_getchar, _FDEV_SETUP_RW);

const int8_t LCD_initialize[] PROGMEM = "LCD Initialized\0";
const int8_t LCD_VNC1L[] PROGMEM = "VNC1L -Test-\0";
const int8_t LCD_mega1284[] PROGMEM = "AVR Atmega1284\0";
const int8_t LCD_success[] PROGMEM = "VNC1L success \0";

int8_t lcd_buffer[17]; // LCD display buffer

volatile unsigned int time1;    //timeout counter
volatile int pre = 0; // previous key stroke
volatile int pre_special = 0; // previous SHIFT, CONTROL key stroke

//the task subroutine
void task1(void);

//**********************************************************
//timer 0 compare ISR
ISR (TIMER0_COMPA_vect)
{
  //Decrement the time if it is not already zero
  if (time1>0)   --time1;
}

//**********************************************************
// LCD setup
void init_lcd(void)
```

```
{
        LCDinit();        //initialize the display
        LCDcursorOFF();
        LCDclr();                                   //clear the display
        LCDGotoXY(0,0);
        CopyStringtoLCD(LCD_initialize, 0, 0);
}

// system initialization
void init_system (void)
{
 // LED for debugging
 LED_DDR |= (1 << LED_PIN); // output
 LED_D1_OFF; // turn off LED

   // 8-bit, no parity, asynchronous UART, 1 stop bit
   UCSR1C |= (1 << UCSZ11) | (1 << UCSZ10);

  //init the UART -- uart_init() is in uart.c
 uart_init();
 uart_init1();
 stdout = stdin = stderr = &uart_str;
 fprintf(stdout,"\n\r***** Starting VNC1L USB keyboard UART test... *****\n\r");
}

// ***********************************************************************
// List of errors
// ***********************************************************************
//      1       = Command was not accepted by VNC1L
//      2       = Command Failed
//      3       = Bad Command
//      4       = Disk Full
//      5       = Filename Invalid
//      6       = Read Only
//      7       = File Open
//      8       = Directory Not Empty
//      9       = No Disk
//      20      = No VNC1L with VDAP firmware available
//      21      = No USB drive plugged
//      50      = Buffer is too small to read a file (buffer overflow)
//      51      = File length = 0 or > 64kByte
//      99      = Timeout
// ***********************************************************************
void vnc_error_message(unsigned char error_type, unsigned char error_num)
```

```
{
 LED_D1_ON;  // LED turns on if there's an error

   fprintf(stdout,"\n\rError type: %d,    Error number: %d\n\r", error_type, error_num);

 // error occured
 while(1); // error keeps the system hanging
}

// *****************************************************************
// Main program
// *****************************************************************
int main (void)
{
 init_system();

 // start the LCD
 init_lcd();
 LCDclr();

   //set up timer 0 for 1 mSec timebase
 TIMSK0= (1<<OCIE0A);          //turn on timer 0 cmp match ISR
 OCR0A = 249;                   //set the compare reg to 250 time ticks
 //set prescalar to divide by 64
 TCCR0B= 3;
 // turn on clear-on-match
 TCCR0A= (1<<WGM01) ;

 //crank up the ISRs
 sei();

 // put some stuff on LCD
 CopyStringtoLCD(LCD_VNC1L, 0, 0);//start at char=0 line=0
 CopyStringtoLCD(LCD_mega1284, 0, 1);//start at char=0 line=1

// To print what monitor returns:
//  fprintf(stdout,"\n\r%s\n\r", vnc.receive_buffer);
// note: All output from the command monitor is LSB first
//      If monitor return multiple lines of data, they are followed by carriage return,
//      which overwrites a previous line on fprintf

 // Wait for VNC1L to respond with firmware version and message if a device is detected
 vnc.status = vnc_wait_for_VNC1L_UART();
 if (vnc.status) vnc_error_message(vnc.status,5);
```

```
  // run IPA command before running ECS command
  // Monitor commands in ASCII: IPA mode
  vnc.status = vnc_wr_cmd_UART("IPA");
  if (vnc.status) vnc_error_message(1,5);                   // error: command was not accepted
by VNC1L
  vnc.status = vnc_prompt_check_UART();                                 // check if monitor
returns prompt
  if (vnc.status) vnc_error_message(vnc.status,5);

  // Extented Command Set
  vnc.status = vnc_wr_cmd_UART("ECS");
  if (vnc.status) vnc_error_message(1,4);                   // error: command was not accepted
by VNC1L
  vnc.status = vnc_prompt_check_UART();                                 // check if monitor
returns prompt
  if (vnc.status) vnc_error_message(vnc.status,4);
  // monitor returns prompt if command executed correctly

  // Display firmware version
  vnc.status = vnc_wr_cmd_UART("FWV");
  if (vnc.status) vnc_error_message(1,6);                   // error: command was not accepted
by VNC1L

  // print firmware version returned by VNC1L and prompt check
  vnc.status = vnc_firmware_UART();

  // Query port 2 for HID
  vnc.status = vnc_wr_cmd_UART("QP2");
  if (vnc.status) vnc_error_message(1,7);                   // error: command was not accepted
by VNC1L

  // print device class and prompt check
  vnc.status = vnc_query_port_UART();

  // Query Device
  vnc.status = vnc_wr_cmd_UART("QD 0");
  if (vnc.status) vnc_error_message(1,8);                   // error: command was not accepted
by VNC1L

  vnc.status = vnc_query_device_UART();

  // Set current device
  vnc.status = vnc_wr_cmd_UART("SC 0");
  if (vnc.status) vnc_error_message(1,9);                            // error: command was not accepted
by VNC1L
  vnc.status = vnc_prompt_check_UART();                                          // check if monitor
returns prompt
  if (vnc.status) vnc_error_message(vnc.status,9);
  // monitor returns prompt if command executed correctly

// Device Send Setup Data - Set Idle
  vnc.status = vnc_wr_cmd_UART("SSU $210A000000000000");
  // no interrupt transfer unless there is a change in the keycode
  // $210A007D00000000 => interrupt transfer every 500ms (500ms/4ms = 0x7D)
  if (vnc.status) vnc_error_message(1,120);
  vnc.status = vnc_prompt_check_UART();                                          // check if monitor
returns prompt
  if (vnc.status) vnc_error_message(vnc.status,120);

        CopyStringtoLCD(LCD_success, 0, 1);

    fprintf(stdout,"\n\rStart polling!\n\r");

  // If you reached here, then no error has occured
  // LED blinks every second
   //main task scheduler loop
  while(1)
  {
        // reset time and call task
    if (time1==0){ time1=t1; task1();}

//    LED_D1_ON;
//    _delay_ms(1000); // 1 sec
//    LED_D1_OFF;
//    _delay_ms(1000);
  }
}

//*******************************
//Task subroutine: Task 1
void task1(void)
{
  // Device Send Setup Data - Get Report
  vnc.status = vnc_wr_cmd_UART("SSU $A101000100000800"); // monitor returns 8 bytes
  if (vnc.status) vnc_error_message(1,12);

        vnc.status = vnc_get_report_UART(KEYBOARD);
```

```
	// only a new key press will be printed
	// here only one key press is checked but you can check 5 more presses
	if((pre != vnc.receive_buffer[FIRST_KEY]) && (vnc.receive_buffer[FIRST_KEY] !=
NO_KEY_PRESSED))
		fprintf(stdout, "%c", print_keystoke(vnc.receive_buffer[FIRST_KEY]));

	if((pre_special != vnc.receive_buffer[SPECIAL_KEY]) &&
(vnc.receive_buffer[SPECIAL_KEY] != NO_KEY_PRESSED))
	{
		switch (vnc.receive_buffer[SPECIAL_KEY])
		{
			case KEY_LEFT_CTRL:
				fprintf(stdout, "\n\rLeft CTRL\n\r");
				break;

			case KEY_LEFT_SHIFT:
				fprintf(stdout, "\n\rLeft SHIFT\n\r");
				break;

			case KEY_RIGHT_CTRL:
				fprintf(stdout, "\n\rRight CTRL\n\r");
				break;

			case KEY_RIGHT_SHIFT:
				fprintf(stdout, "\n\rRight SHIFT\n\r");
				break;
		}
	}

	pre = vnc.receive_buffer[FIRST_KEY];
	pre_special = vnc.receive_buffer[SPECIAL_KEY];
}
```

****************** *firmware.h* *******************

```
#define LED_PIN			0
#define LED_PORT	PORTB
#define LED_DDR			DDRB

#define RTS_DDR			DDRA
#define RTS_PORT	PORTA
#define RTS_PIN			0
```

```
#define CTS_DDR                 DDRA
#define CTS_PORT       PORTA
#define CTS_PIN                 1

//*******************************************************************
//*******************************************************************
#define LED_D1_ON   LED_PORT &= ~(1<<LED_PIN) /* LED */
#define LED_D1_OFF  LED_PORT |=  (1<<LED_PIN)

#define PROMPT_CHECK                      2
#define PROMPT_AND_ERROR_CHECK            1
#define PROMPT_OK                         1
#define TIMEOUT_READ_WRITE                500             /* Number of read/write tests if
buffer is full */
#define TIMEOUT_COMMUNICATION             5               /* Timeout if no communication has
happened */

#define FILE_BUFFER_SIZE     100               /* Buffer size is at least 100 */

#define NO_KEY_PRESSED     0

#define KEY_CTRL       0x01
#define KEY_SHIFT      0x02
#define KEY_ALT                 0x04
#define KEY_GUI                 0x08
#define KEY_LEFT_CTRL           0x01
#define KEY_LEFT_SHIFT          0x02
#define KEY_LEFT_ALT            0x04
#define KEY_LEFT_GUI            0x08
#define KEY_RIGHT_CTRL          0x10
#define KEY_RIGHT_SHIFT         0x20
#define KEY_RIGHT_ALT           0x40
#define KEY_RIGHT_GUI           0x80

#define KEY_A          4
#define KEY_B          5
#define KEY_C          6
#define KEY_D          7
#define KEY_E          8
#define KEY_F          9
#define KEY_G          10
#define KEY_H          11
#define KEY_I          12
#define KEY_J          13
```

```
#define KEY_K		14
#define KEY_L		15
#define KEY_M			16
#define KEY_N		17
#define KEY_O		18
#define KEY_P		19
#define KEY_Q		20
#define KEY_R		21
#define KEY_S		22
#define KEY_T		23
#define KEY_U		24
#define KEY_V		25
#define KEY_W			26
#define KEY_X		27
#define KEY_Y		28
#define KEY_Z		29
#define KEY_1		30
#define KEY_2		31
#define KEY_3		32
#define KEY_4		33
#define KEY_5		34
#define KEY_6		35
#define KEY_7		36
#define KEY_8		37
#define KEY_9		38
#define KEY_0		39
#define KEY_ENTER	40
#define KEY_ESC			41
#define KEY_BACKSPACE		42
#define KEY_TAB			43
#define KEY_SPACE	44
#define KEY_MINUS	45
#define KEY_EQUAL	46
#define KEY_LEFT_BRACE		47
#define KEY_RIGHT_BRACE		48
#define KEY_BACKSLASH		49
#define KEY_NUMBER		50
#define KEY_SEMICOLON		51
#define KEY_QUOTE	52
#define KEY_TILDE	53
#define KEY_COMMA	54
#define KEY_PERIOD	55
#define KEY_SLASH	56
#define KEY_CAPS_LOCK		57
```

```
#define KEY_F1                  58
#define KEY_F2                  59
#define KEY_F3                  60
#define KEY_F4                  61
#define KEY_F5                  62
#define KEY_F6                  63
#define KEY_F7                  64
#define KEY_F8                  65
#define KEY_F9                  66
#define KEY_F10                 67
#define KEY_F11                 68
#define KEY_F12                 69
#define KEY_PRINTSCREEN         70
#define KEY_SCROLL_LOCK         71
#define KEY_PAUSE       72
#define KEY_INSERT      73
#define KEY_HOME        74
#define KEY_PAGE_UP             75
#define KEY_DELETE      76
#define KEY_END                 77
#define KEY_PAGE_DOWN           78
#define KEY_RIGHT       79
#define KEY_LEFT        80
#define KEY_DOWN        81
#define KEY_UP                  82
#define KEY_NUM_LOCK            83
#define KEYPAD_SLASH            84
#define KEYPAD_ASTERIX          85
#define KEYPAD_MINUS            86
#define KEYPAD_PLUS             87
#define KEYPAD_ENTER            88
#define KEYPAD_1        89
#define KEYPAD_2        90
#define KEYPAD_3        91
#define KEYPAD_4        92
#define KEYPAD_5        93
#define KEYPAD_6        94
#define KEYPAD_7        95
#define KEYPAD_8        96
#define KEYPAD_9        97
#define KEYPAD_0        98
#define KEYPAD_PERIOD           99
```

***************** *firmware.c* ******************

```
#include "firmware.h" // defines macros
#define VNC_BUFFER_SIZE 160          /* Transmit/receive buffer size is at least 60 */
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% 60

struct vnc_struc
{
        char  receive_buffer[VNC_BUFFER_SIZE];      // Receive buffer
        char send_buffer[VNC_BUFFER_SIZE];          // Send buffer
        unsigned char status;                       // Status
};

struct vnc_struc vnc; // instantiate vnc_struc

// *****************************************************************
// VNC1L - Sending a command
// Parameter: wr_cmd : command (address)
// Return: status    : 0 = command accepted by VNC1L
// *****************************************************************
unsigned char vnc_wr_cmd_UART(char *wr_cmd)
{
        unsigned char offset;  // vnc_offset, length of command

  for(offset = strlen(wr_cmd); offset; offset--)
  {
        while((UCSR1A & (1<<UDRE1)) == 0);
        UDR1 = *wr_cmd;
    wr_cmd++;
  }
  // send carrage return (a command to monitor ends with 0x0D)
  while((UCSR1A & (1<<UDRE1)) == 0);
  UDR1 = 0x0D;

  return 0;
}

// *****************************************************************
// VNC1L - Receive a string
// Wait until the monitor returns info from firmware (ex. Firmware Version, Device Detected)
// Return: status    : 0 = string was received from VNC1L
// *****************************************************************
unsigned char vnc_wait_for_VNC1L_UART()
{
  char count, x;
  // Wait for VNC1L to respond with firmware version
  // and message if a device is detected
        for(int z=0; z < 44; z++)
        {
              // wait until a byte has been received
              while((UCSR1A & (1<<RXC1)) == 0);
              vnc.receive_buffer[z] = UDR1;
              // printing in this loop could cause the program to break
//            fprintf(stdout, "\n\r%c", vnc.receive_buffer[z]);
        }

   // get the length of the string and it's last character index
   count = strlen(vnc.receive_buffer) - 1;

        for(x = 0; x < count; x++)
        {
                    if(vnc.receive_buffer[x] == '\r')
                    {
                          fprintf(stdout, "\n", vnc.receive_buffer[x]);
                    fprintf(stdout, "%c", vnc.receive_buffer[x]);
                    }
                    else
                    fprintf(stdout, "%c", vnc.receive_buffer[x]);
        }

        return 0;
}

// ********************************************************************
// VNC1L - Get firmware version
// Return: 0 = Prompt received
// ********************************************************************
unsigned char vnc_firmware_UART(void)
{
        int z;

  // get message from VNC1L and check prompt
        for(z = 0; z < 34; z++)
        {
              // wait until a byte has been received
              while((UCSR1A & (1<<RXC1)) == 0);
              vnc.receive_buffer[z] = UDR1;
//            fprintf(stdout, "\n\r%c", vnc.receive_buffer[z]);
        }
```

```
//  count = strlen(vnc.receive_buffer) - 1;

        for(z = 0; z < 34; z++) // for(x = 0; x < count; x++)
        {
                        if(vnc.receive_buffer[z] == '\r')
                        {
                                fprintf(stdout, "\n", vnc.receive_buffer[z]);
                        fprintf(stdout, "%c", vnc.receive_buffer[z]);
                        }
                        else
                        fprintf(stdout, "%c", vnc.receive_buffer[z]);
        }

        return 0;
}

// ********************************************************************
// VNC1L - Waiting for PROMPT
// Return: 0 = Prompt received
// ********************************************************************
unsigned char vnc_prompt_check_UART(void)
{
        int done = 0;
        int rd_count = 10; //TIMEOUT_COMMUNICATION;  // TIMEOUT error / abort if
necessary
        int count = 0;

        while(rd_count)
        {
                if(!done)
                {
                        // wait until a byte has been received
                        while((UCSR1A & (1<<RXC1)) == 0);
                        vnc.receive_buffer[count] = UDR1;
                        // printing here could cause the program to break
                        //fprintf(stdout, "\n\r%c", vnc.receive_buffer[count]);

                        // prompt received
                        if(vnc.receive_buffer[count] == '>')//0x0D') // D:\>
                        {
                                //if(vnc.receive_buffer[count-1] == '>' &&
vnc.receive_buffer[count-2] == '0x5C' && vnc.receive_buffer[count-3] == ':' &&
vnc.receive_buffer[count-4] == 'D')
```

```
                                        return 0;
                                }
                                count++;
                        }
                        rd_count--;
                }

                return 1; // no prompt, timeout
}

// ******************************************************************
// VNC1L - Get device class from VNC1L (for example, $08 &00 for HID)
// Return: 0 = Prompt received
// ******************************************************************
unsigned char vnc_query_port_UART(void)
{
                int z;

                // get message from VNC1L and check prompt
                for(z = 0; z < 13; z++)
                {
                        // wait until a byte has been received
                        while((UCSR1A & (1<<RXC1)) == 0);
                        vnc.receive_buffer[z] = UDR1;
//                      fprintf(stdout, "\n\r%c", vnc.receive_buffer[z]);
                }

//  count = strlen(vnc.receive_buffer) - 1;

                // get message from VNC1L and check prompt
                for(z = 0; z < 13; z++) //for(x = 0; x < count; x++)
                {
                                if(vnc.receive_buffer[z] == '\r')
                                {
                                        fprintf(stdout, "\n", vnc.receive_buffer[z]);
                                fprintf(stdout, "%c", vnc.receive_buffer[z]);
                                }
                                else
                                fprintf(stdout, "%c", vnc.receive_buffer[z]);
                }

                return 0;
}
```

```
// ******************************************************************
// VNC1L - Get USB related info of a device
// Return: 0 = Prompt received
// ******************************************************************
unsigned char vnc_query_device_UART(void)
{
        int z, count;

  // get message from VNC1L and check prompt
        for(z = 0; z < 135; z++)
        {
                // wait until a byte has been received
                while((UCSR1A & (1<<RXC1)) == 0);
                vnc.receive_buffer[z] = UDR1;
//              fprintf(stdout, "\n\r%c", vnc.receive_buffer[z]);
        }

  count = strlen(vnc.receive_buffer) - 1;

        for(z = 0; z < count; z++) //for(x = 0; x < count; x++)
        {
                        if(vnc.receive_buffer[z] == '\r')
                        {
                                fprintf(stdout, "\n", vnc.receive_buffer[z]);
                        fprintf(stdout, "%c", vnc.receive_buffer[z]);
                        }
                        else
                        fprintf(stdout, "%c", vnc.receive_buffer[z]);
        }

        return 0;
}

// ******************************************************************
// VNC1L - Get class specific report
// Return: 0 = Prompt received
// ******************************************************************
unsigned char vnc_get_report_UART(char HID_class)
{
        int z;

  // This includes prompt check as well as parsing a report
        for(z = 0; z < HID_class; z++)
        {
```

```
			// wait until a byte has been received
			while((UCSR1A & (1<<RXC1)) == 0);
			vnc.receive_buffer[z] = UDR1;
//			fprintf(stdout, "\n\r%c", vnc.receive_buffer[z]);
		}

		return 0;
}

unsigned char print_keystoke(char key_pressed)
{
		switch (key_pressed)
		{
			case KEY_0: return '0';

			case KEY_1: return '1';

			case KEY_2: return '2';

			case KEY_3: return '3';

			case KEY_4: return '4';

			case KEY_5: return '5';

			case KEY_6: return '6';

			case KEY_7: return '7';

			case KEY_8: return '8';

			case KEY_9: return '9';

			case KEY_A: return 'A';

			case KEY_B: return 'B';

			case KEY_C: return 'C';

			case KEY_D: return 'D';

			case KEY_E: return 'E';

			case KEY_F: return 'F';
case KEY_G: return 'G';

case KEY_H: return 'H';

case KEY_I: return 'I';

case KEY_J: return 'J';

case KEY_K: return 'K';

case KEY_L: return 'L';

case KEY_M: return 'M';

case KEY_N: return 'N';

case KEY_O: return 'O';

case KEY_P: return 'P';

case KEY_Q: return 'Q';

case KEY_R: return 'R';

case KEY_S: return 'S';

case KEY_T: return 'T';

case KEY_U: return 'U';

case KEY_V: return 'V';

case KEY_W: return 'W';

case KEY_X: return 'X';

case KEY_Y: return 'Y';

case KEY_Z: return 'Z';

case KEY_ENTER: return '&';

case KEY_SPACE: return ' ';
```

```
                case NO_KEY_PRESSED: return '^';

                default: return '#';
        }
}
```

6. VNC1L_mouse_UART_polling

```
****************** VNC1L_mouse_UART.c *******************

// Original project by Matthias Kahnt az51@gmx.net
// Modified by Terry Young Kim
// ATmega1284
// VNC1L/VDIP1 Vinculum (Port2)
// VNC1L - USB HOST CONTROLLER (Mega1284: Master, VNC1L: Slave)
// VDAP firmware version 3.69
// USART interface
// VDIP1: Jumper-Set: J3 - Pullup / J4 - Pullup for UART
// VNC1L: connect CTS (AD3) to ground (pull down low) or connect CTS (AD3) and RTS (AD2)

// WARNING: if the prompt check is carried out properly, the program will hang!

#include <avr/io.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <avr/interrupt.h>
#include <util/delay.h>     // for LCD

#include "uart.c"  // UART channel 0 ==> PuTTy
#include "uart1.c"  // UART channel 1 <==> VNC1L
#include "lcd_lib.c"  // LCD
#include "firmware.c" // VNC1L APIs

#define F_CPU 16000000UL

#define MOUSE                           18
#define BUTTONS                         10
#define X_COORDINATE            11
#define Y_COORDINATE            12
#define WHEEL_COORDINATE        13
#define BIT_1_ON                1
#define BIT_2_ON                2
```

```
#define BIT_4_ON                    4

//timeout value for a task
#define t1 50 // 50ms works fine

volatile unsigned int time1;    //timeout counter

// LED: PB0
// LCD: PORT C

// VNC: TX       PD3
//      RX    PD4

// UART file descriptor
// putchar and getchar are in uart.c
FILE uart_str = FDEV_SETUP_STREAM(uart_putchar, uart_getchar, _FDEV_SETUP_RW);

const int8_t LCD_initialize[] PROGMEM = "LCD Initialized\0";
const int8_t LCD_VNC1L[] PROGMEM = "VNC1L -Test-\0";
const int8_t LCD_mega1284[] PROGMEM = "AVR Atmega1284\0";
const int8_t LCD_success[] PROGMEM = "VNC1L success \0";

int8_t lcd_buffer[17]; // LCD display buffer

volatile int pre = 0;
volatile int pre_x = 0;
volatile int pre_y = 0;
volatile int pre_wheel = 0;

//the task subroutine
void task1(void);

//********************************************************
//timer 0 compare ISR
ISR (TIMER0_COMPA_vect)
{
  //Decrement the time if it is not already zero
  if (time1>0)   --time1;
}

//********************************************************
// LCD setup
void init_lcd(void)
{
        LCDinit();      //initialize the display
        LCDcursorOFF();
        LCDclr();                               //clear the display
        LCDGotoXY(0,0);
        CopyStringtoLCD(LCD_initialize, 0, 0);
}

// system initialization
void init_system (void)
{
  // LED for debugging
  LED_DDR |= (1 << LED_PIN); // output
  LED_D1_OFF; // turn off LED

    // 8-bit, no parity, asynchronous UART, 1 stop bit
    UCSR1C |= (1 << UCSZ11) | (1 << UCSZ10);

   //init the UART -- uart_init() is in uart.c
  uart_init();
  uart_init1();
  stdout = stdin = stderr = &uart_str;
  fprintf(stdout,"\n\r***** Starting VNC1L USB mouse UART test... *****\n\r");
}

// **********************************************************************
// List of errors
// **********************************************************************
//      1       = Command was not accepted by VNC1L
//      2       = Command Failed
//      3       = Bad Command
//      4       = Disk Full
//      5       = Filename Invalid
//      6       = Read Only
//      7       = File Open
//      8       = Directory Not Empty
//      9       = No Disk
//      20      = No VNC1L with VDAP firmware available
//      21      = No USB drive plugged
//      50      = Buffer is too small to read a file (buffer overflow)
//      51      = File length = 0 or > 64kByte
//      99      = Timeout
// **********************************************************************
void vnc_error_message(unsigned char error_type, unsigned char error_num)
{
  LED_D1_ON;  // LED turns on if there's an error

    fprintf(stdout,"\n\rError type: %d,    Error number: %d\n\r", error_type, error_num);

  // error occured
  while(1); // error keeps the system hanging
}

// ******************************************************************
// Main program
// ******************************************************************
int main (void)
{
  init_system();

  // start the LCD
  init_lcd();
  LCDclr();

    //set up timer 0 for 1 mSec timebase
  TIMSK0= (1<<OCIE0A);          //turn on timer 0 cmp match ISR
  OCR0A = 249;                  //set the compare reg to 250 time ticks
  //set prescalar to divide by 64
  TCCR0B= 3;
  // turn on clear-on-match
  TCCR0A= (1<<WGM01) ;

  //crank up the ISRs
  sei();

  // put some stuff on LCD
  CopyStringtoLCD(LCD_VNC1L, 0, 0);//start at char=0 line=0
  CopyStringtoLCD(LCD_mega1284, 0, 1);//start at char=0 line=1

// To print what monitor returns:
//  fprintf(stdout,"\n\r%s\n\r", vnc.receive_buffer);
// note: All output from the command monitor is LSB first
//      If monitor return multiple lines of data, they are followed by carriage return,
//      which overwrites a previous line on fprintf

  // Wait for VNC1L to respond with firmware version and message if a device is detected
  vnc.status = vnc_wait_for_VNC1L_UART();
  if (vnc.status) vnc_error_message(vnc.status,5);
  // run IPA command before running ECS command
  // Monitor commands in ASCII: IPA mode
  vnc.status = vnc_wr_cmd_UART("IPA");
  if (vnc.status) vnc_error_message(1,5);                    // error: command was not accepted
by VNC1L
  vnc.status = vnc_prompt_check_UART();                                 // check if monitor
returns prompt
  if (vnc.status) vnc_error_message(vnc.status,5);

  // Extented Command Set
  vnc.status = vnc_wr_cmd_UART("ECS");
  if (vnc.status) vnc_error_message(1,4);                    // error: command was not accepted
by VNC1L
  vnc.status = vnc_prompt_check_UART();                                 // check if monitor
returns prompt
  if (vnc.status) vnc_error_message(vnc.status,4);
  // monitor returns prompt if command executed correctly

  // Display firmware version
  vnc.status = vnc_wr_cmd_UART("FWV");
  if (vnc.status) vnc_error_message(1,6);                    // error: command was not accepted
by VNC1L

  // print firmware version returned by VNC1L and prompt check
  vnc.status = vnc_firmware_UART();

  // Query port 2 for HID
  vnc.status = vnc_wr_cmd_UART("QP2");
  if (vnc.status) vnc_error_message(1,7);                    // error: command was not accepted
by VNC1L

  // print device class and prompt check
  vnc.status = vnc_query_port_UART();

  // Query Device
  vnc.status = vnc_wr_cmd_UART("QD 0");
  if (vnc.status) vnc_error_message(1,8);                    // error: command was not accepted
by VNC1L

  vnc.status = vnc_query_device_UART();

  // Set current device
  vnc.status = vnc_wr_cmd_UART("SC 0");
  if (vnc.status) vnc_error_message(1,9);                          // error: command was not accepted
by VNC1L
  vnc.status = vnc_prompt_check_UART();                                        // check if monitor
returns prompt
  if (vnc.status) vnc_error_message(vnc.status,9);
  // monitor returns prompt if command executed correctly

// Device Send Setup Data - Set Idle
  vnc.status = vnc_wr_cmd_UART("SSU $210A000000000000");
  // no interrupt transfer unless there is a change in the keycode
  // $210A007D00000000 => interrupt transfer every 500ms (500ms/4ms = 0x7D)
  if (vnc.status) vnc_error_message(1,120);
  vnc.status = vnc_prompt_check_UART();                                        // check if monitor
returns prompt
  if (vnc.status) vnc_error_message(vnc.status,120);

        CopyStringtoLCD(LCD_success, 0, 1);

   fprintf(stdout,"\n\rStart polling!\n\r");

  // If you reached here, then no error has occured
  // LED blinks every second
   //main task scheduler loop
  while(1)
  {
        // reset time and call task
   if (time1==0){ time1=t1; task1();}

//    LED_D1_ON;
//    _delay_ms(1000); // 1 sec
//    LED_D1_OFF;
//    _delay_ms(1000);
  }
}

//********************************
//Task subroutine: Task 1
void task1(void)
{
  // Device Send Setup Data - Get Report
  vnc.status = vnc_wr_cmd_UART("SSU $A101000100000400"); // monitor returns 4 bytes
  if (vnc.status) vnc_error_message(1,12);

        vnc.status = vnc_get_report_UART(MOUSE);
```

```
	// left click, right click, wheel click
	if((pre != vnc.receive_buffer[BUTTONS]) && vnc.receive_buffer[BUTTONS] != 
NO_BUTTON_PRESSED) // buttons
	{
		if(vnc.receive_buffer[BUTTONS] == BIT_1_ON)
			fprintf(stdout, "\n\rLeft click");
		else if(vnc.receive_buffer[BUTTONS] == BIT_2_ON)
			fprintf(stdout, "\n\rRight click");
		else if(vnc.receive_buffer[BUTTONS] == BIT_4_ON)
			fprintf(stdout, "\n\rWheel");
		//else
		//	fprintf(stdout, "\n\rNo movement\n\r");
	}

	// X coordinate
	if((pre_x != vnc.receive_buffer[X_COORDINATE]) && 
vnc.receive_buffer[X_COORDINATE] != NO_MOVEMENT)
	{
		fprintf(stdout, "\n\rX = %d", vnc.receive_buffer[X_COORDINATE]);
	}

	// Y coordinate
	if((pre_y != vnc.receive_buffer[Y_COORDINATE]) && 
vnc.receive_buffer[Y_COORDINATE] != NO_MOVEMENT)
	{
		fprintf(stdout, "\n\r	Y = %d", vnc.receive_buffer[Y_COORDINATE]);
	}

	// wheel rotation
	if((pre_wheel != vnc.receive_buffer[WHEEL_COORDINATE]) && 
vnc.receive_buffer[WHEEL_COORDINATE] != NO_MOVEMENT)
	{
		fprintf(stdout, "\n\r		Wheel = %d", 
vnc.receive_buffer[WHEEL_COORDINATE]);
	}

	pre = vnc.receive_buffer[BUTTONS];
	pre_x = vnc.receive_buffer[X_COORDINATE];
	pre_y = vnc.receive_buffer[Y_COORDINATE];
	pre_wheel = vnc.receive_buffer[WHEEL_COORDINATE];
}
```

****************** *firmware.h* *******************

```
#define LED_PIN             0
#define LED_PORT     PORTB
#define LED_DDR             DDRB

#define RTS_DDR             DDRA
#define RTS_PORT     PORTA
#define RTS_PIN             0

#define CTS_DDR             DDRA
#define CTS_PORT     PORTA
#define CTS_PIN             1

//*********************************************************************
//*********************************************************************
#define LED_D1_ON   LED_PORT &= ~(1<<LED_PIN) /* LED */
#define LED_D1_OFF  LED_PORT |=  (1<<LED_PIN)

#define PROMPT_CHECK                    2
#define PROMPT_AND_ERROR_CHECK          1
#define PROMPT_OK                       1
#define TIMEOUT_READ_WRITE              500             /* Number of read/write tests if
buffer is full */
#define TIMEOUT_COMMUNICATION           5               /* Timeout if no communication has
happened */

#define FILE_BUFFER_SIZE     100            /* Buffer size is at least 100 */

#define NO_BUTTON_PRESSED       0
#define NO_MOVEMENT                     0
```

****************** *firmware.c* *******************

```
#include "firmware.h" // defines macros
#define VNC_BUFFER_SIZE 160         /* Transmit/receive buffer size is at least 60 */
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% 60

struct vnc_struc
{
        char  receive_buffer[VNC_BUFFER_SIZE];      // Receive buffer
        char send_buffer[VNC_BUFFER_SIZE];          // Send buffer
        unsigned char status;                       // Status
};

struct vnc_struc vnc; // instantiate vnc_struc
```

```
// *******************************************************************
// VNC1L - Sending a command
// Parameter: wr_cmd : command (address)
// Return: status    : 0 = command accepted by VNC1L
// *******************************************************************
unsigned char vnc_wr_cmd_UART(char *wr_cmd)
{
	unsigned char offset;  // vnc_offset, length of command

  for(offset = strlen(wr_cmd); offset; offset--)
  {
	while((UCSR1A & (1<<UDRE1)) == 0);
	UDR1 = *wr_cmd;
    wr_cmd++;
  }
  // send carrage return (a command to monitor ends with 0x0D)
  while((UCSR1A & (1<<UDRE1)) == 0);
  UDR1 = 0x0D;

  return 0;
}

// *******************************************************************
// VNC1L - Receive a string
// Wait until the monitor returns info from firmware (ex. Firmware Version, Device Detected)
// Return: status    : 0 = string was received from VNC1L
// *******************************************************************
unsigned char vnc_wait_for_VNC1L_UART()
{
  char count, x;

  // Wait for VNC1L to respond with firmware version
  // and message if a device is detected
	for(int z=0; z < 44; z++)
	{
		// wait until a byte has been received
		while((UCSR1A & (1<<RXC1)) == 0);
		vnc.receive_buffer[z] = UDR1;
		// printing in this loop could cause the program to break
//		fprintf(stdout, "\n\r%c", vnc.receive_buffer[z]);
	}

   // get the length of the string and it's last character index
   count = strlen(vnc.receive_buffer) - 1;

          for(x = 0; x < count; x++)
          {
                              if(vnc.receive_buffer[x] == '\r')
                              {
                                        fprintf(stdout, "\n", vnc.receive_buffer[x]);
                              fprintf(stdout, "%c", vnc.receive_buffer[x]);
                              }
                              else
                              fprintf(stdout, "%c", vnc.receive_buffer[x]);
          }

          return 0;
}

// ********************************************************************
// VNC1L - Get firmware version
// Return: 0 = Prompt received
// ********************************************************************
unsigned char vnc_firmware_UART(void)
{
          int z;

  // get message from VNC1L and check prompt
          for(z = 0; z < 34; z++)
          {
                    // wait until a byte has been received
                    while((UCSR1A & (1<<RXC1)) == 0);
                    vnc.receive_buffer[z] = UDR1;
//                  fprintf(stdout, "\n\r%c", vnc.receive_buffer[z]);
          }

//  count = strlen(vnc.receive_buffer) - 1;

          for(z = 0; z < 34; z++) // for(x = 0; x < count; x++)
          {
                              if(vnc.receive_buffer[z] == '\r')
                              {
                                        fprintf(stdout, "\n", vnc.receive_buffer[z]);
                              fprintf(stdout, "%c", vnc.receive_buffer[z]);
                              }
                              else
                              fprintf(stdout, "%c", vnc.receive_buffer[z]);
```

```
	}

	return 0;
}

// ***********************************************************************
// VNC1L - Waiting for PROMPT
// Return: 0 = Prompt received
// ***********************************************************************
unsigned char vnc_prompt_check_UART(void)
{
	int done = 0;
	int rd_count = 10; //TIMEOUT_COMMUNICATION;	// TIMEOUT error / abort if
necessary
	int count = 0;

	while(rd_count)
	{
		if(!done)
		{
			// wait until a byte has been received
			while((UCSR1A & (1<<RXC1)) == 0);
			vnc.receive_buffer[count] = UDR1;
			// printing here could cause the program to break
			//fprintf(stdout, "\n\r%c", vnc.receive_buffer[count]);

			// prompt received
			if(vnc.receive_buffer[count] == '>')//0x0D') // D:\>
			{
				//if(vnc.receive_buffer[count-1] == '>' &&
vnc.receive_buffer[count-2] == '0x5C' && vnc.receive_buffer[count-3] == ':' &&
vnc.receive_buffer[count-4] == 'D')
					return 0;
			}
			count++;
		}
		rd_count--;
	}

	return 1; // no prompt, timeout
}

// ***********************************************************************
// VNC1L - Get device class from VNC1L (for example, $08 &00 for HID)
// Return: 0 = Prompt received
// ***************************************************************
unsigned char vnc_query_port_UART(void)
{
        int z;

        // get message from VNC1L and check prompt
        for(z = 0; z < 13; z++)
        {
                // wait until a byte has been received
                while((UCSR1A & (1<<RXC1)) == 0);
                vnc.receive_buffer[z] = UDR1;
//              fprintf(stdout, "\n\r%c", vnc.receive_buffer[z]);
        }

//  count = strlen(vnc.receive_buffer) - 1;

        // get message from VNC1L and check prompt
        for(z = 0; z < 13; z++) //for(x = 0; x < count; x++)
        {
                        if(vnc.receive_buffer[z] == '\r')
                        {
                                fprintf(stdout, "\n", vnc.receive_buffer[z]);
                        fprintf(stdout, "%c", vnc.receive_buffer[z]);
                        }
                        else
                        fprintf(stdout, "%c", vnc.receive_buffer[z]);
        }

        return 0;
}

// ***************************************************************
// VNC1L - Get USB related info of a device
// Return: 0 = Prompt received
// ***************************************************************
unsigned char vnc_query_device_UART(void)
{
        int z, count;

  // get message from VNC1L and check prompt
        for(z = 0; z < 135; z++)
        {
                // wait until a byte has been received
```

```
			while((UCSR1A & (1<<RXC1)) == 0);
			vnc.receive_buffer[z] = UDR1;
//			fprintf(stdout, "\n\r%c", vnc.receive_buffer[z]);
		}

  count = strlen(vnc.receive_buffer) - 1;

		for(z = 0; z < count; z++) //for(x = 0; x < count; x++)
		{
				if(vnc.receive_buffer[z] == '\r')
				{
					fprintf(stdout, "\n", vnc.receive_buffer[z]);
				fprintf(stdout, "%c", vnc.receive_buffer[z]);
				}
				else
				fprintf(stdout, "%c", vnc.receive_buffer[z]);
		}

		return 0;
}

// *********************************************************************
// VNC1L - Get class specific report
// Return: 0 = Prompt received
// *********************************************************************
unsigned char vnc_get_report_UART(char HID_class)
{
		int z;

  // This includes prompt check as well as parsing a report
		for(z = 0; z < HID_class; z++)
		{
			// wait until a byte has been received
			while((UCSR1A & (1<<RXC1)) == 0);
			vnc.receive_buffer[z] = UDR1;
//			fprintf(stdout, "\n\r%c", vnc.receive_buffer[z]);
		}

		return 0;
}
```