

## 6 Channel Logic Analyzer Project

This project uses an ATMEGA168 microcontroller. The microcontroller uses its pin change interrupt to detect a change in logic level on any of the 6 pins (in this case, the entire port C is used, which only has 6 usable pins). When a change is detected, the value is stored in a 1000 byte buffer (the ATMEGA168 only has 1024 bytes of SRAM memory), and if the serial port is not busy, it will send the data to the computer in a FIFO (first in first out) fashion. The computer can change the interrupt mask register to enable/disable changes triggering a particular pin. The firmware is written in AVR-GCC with AVR Studio, and it can be easily modified to be used with any AVR microcontroller with hardware UART and pin change interrupts.

Hardware wise, all you need is the ATMEGA168, and a 16MHz clock source, a reset button, and also a serial port connection (either a MAX232 converter to use with a RS232 serial port, or a USB to serial converter like the FT232RL chip). You can use the Arduino (including all variations running on 16MHz) if you have one since it's the same thing. The 6 pins on port C (pin number 23 to 28) can be connected to any style of test probes (alligator clips, multimeter probes, hooks, etc), but keep in mind, the circuit you are analyzing must have its ground connected to the analyzer's ground. A buffer chip should be used to avoid damaging the microcontroller but I did not have any when I built this.

The computer software is written in Processing, which means it'll work on Linux, Windows, or Macs. It stores and displays the logic state of each pin as data is transferred from the microcontroller. The user can use buttons to disable/enable triggering on any of the 6 pins, and a scroll bar allows the user to view several hundred of previously saved logic states on a timeline. Note, new data is only received if the state changes, so this timeline is not real time.

I estimate the performance to be sufficient enough to analyze a 400KHz I2C data bus without any problems. Also note that 8 pins can be used, but just not on an ATMEGA168, simple changes in the software will allow for 8 channels to be used instead of 6.

To make your own, simply program an ATMEGA168 with the provided hex file, and run the Processing program. I think I commented the code enough for people to modify so they can use their own processor. Also read the other documents in this project. I assume you understand how to program microcontrollers and understand basic computer science concepts like for loops and such.

### User Interface

The user interface is programmed in Processing. It's a easy to learn Java language that handles graphic functions for you, and also has libraries so you can use the serial port.

First, the settings are stored in the file called "settings", from there you can change various aspects of the user interface like the size of the scroll bar, the height of the bits, width, and stuff like that.

All the global variables are declared inside "vars".

Lets look at the main file.

First thing to do, import the serial port library, and give the serial port object a name, in this case, "port".

Processing will first call the “setup” function.

Inside “setup”, the first thing that happens is that the serial port is opened at 115200 baud.

Then, the “values” array is cleared to all zeros, without giving it a initial value, the program will crash.

All channel triggering are enabled and this data is sent to the microcontroller, which will call it's own interrupt routine and apply the mask.

The user interface window is opened and the viewOffset is given a value so the user is looking at the latest incoming data.

The “draw” function handles everything that is graphical about the program. It draws all the lines, buttons, and bits. This is very intense on the CPU so we call “noLoop” which basically means do not repeatedly call “draw”, instead, every time something happens, we call “redraw” which runs “draw” only once. This keeps your computer running fast and your CPU cool.

Inside “draw”, first thing that is drawn are the guide lines which are just a visual aid. Next, the buttons are drawn, green if the trigger on that channel is enable, red if disabled. Next, the bits are drawn, this is the complicated one, only the bits that are in the viewing area are rendered, the individual variable inside the “values” array are converted to binary, and each bit is displayed as either a yellow or blue rectangle. Next, the scroll bar is drawn, with a green rectangle indicating where you are viewing, right side being the latest and left side being old data.

Several events may occur, the user can choose to drag the scroll bar, the user can click on the buttons, or new data could be received from the microcontroller. After each of these events, “redraw” is called to draw the user interface once.

When the user drags the mouse, the function “mouseDragged” is called, and if the mouse is within the scroll bar area, the new location of the viewing offset is calculated.

When the user clicks the mouse, “mouseClicked” is called, and if the mouse is on a button, that button is toggled, the trigger of that channel is enabled/disabled, a new PCMSK is calculated, and sent to the microcontroller.

When new data comes from the microcontroller, “serialEvent” is called, the “values” array is shifted, and the new data is stored at the end.

## **Firmware**

All AVR-GCC programs starts from a function called “main”. Inside “main” I call a initialization routine called “init\_main”.

Inside “init\_main”, the first thing I do is enable global pull up resistors by clearing bit 4 of the MCUCR register (refer to datasheet page 88)

\*Note that cbi and sbi mean clear bit and set bit, these are macros defined in the “all\_header.h” file.

Next line in the initialization process, I disable the watchdog timer by clearing the WDTCSR register (page 54 of datasheet). This isn't needed since it's not running by default unless your chip has its

watchdog-always-on fuse programmed.

Next, global interrupts are enabled by calling the sei command. Sei is a macro defined in <avr/interrupt.h> that sets bit 7 in the SREG register (page 12 of datasheet), which enables interrupt events to cause the program to jump into a specific interrupt vector.

“serInit0” is called, which is a function found in “usart.h”

Inside “serInit0”, the Rx and Tx pins have their pull up resistors enabled, and the Rx pin is set to be an input while the Tx pin is set to output.

\*Note that the names of all pins used in this program are defined inside “pins.h”

\*Note, read page 73 of the datasheet to understand how to set a pin as input/output, and how to enable pull up resistors on individual pins.

Next in “serInit0”, the baud rate is set to 115200 bits per second. According to page 199 of the datasheet, in order to have a baud rate of 115200 with the U2X0 bit equal to 0 (which it is), we have to set UBRR0 to 8, which means the lower 8 bits of UBRR0 is equal to 8 and the higher 8 bits of UBRR0 is equal to 0. Thus we set UBRR0L to 8 and UBRR0H to 0 (these two registers are described in page 195 of the datasheet).

\*Note, “compBaud” is defined as 8, and is found in “config.h”

The UCSR0C is set, the configuration is 8 bits, no parity bit, and 1 stop bit.

The UCSR0B is set, this enables both the Rx and Tx interrupt, and also enables the USART module.

\*Page 192 and 193 of the datasheet has details on these registers.

The “serBusy” flag is cleared, and the FIFO buffer pointers are cleared.

“serInit0” is now finished, the next lines in “init\_main” is to disable pull up resistors on the logic analyzer port (hence the LA), and also to set to input.

\*Note, LAport, LAddr, and LAPinin, are all defined in “pins.h” as PORTC, DDRC, and PINC

Read page 2 of the datasheet. Since this project uses port C of the ATMEGA168, notice we want to use PCINT8, 9, 10, 11, 12, and 13. Now read page 70, bit 1 of the PCICR register enables the interrupts on PCINT8 to 14. We don't want 14, only 8 to 13, so we look at register PCMSK1, which is responsible for the individual pins between PCINT8 to 14, and we disable 14 and enable 8 to 13 in the next line of the program.

\*Note, LAmask defines PCMSK1, found in the “config.h” file

After setting bit 1 of PCICR like I said before, interrupts are now on. At this point, the logic analyzer is running.

It's nice for the computer to get an initial reading of all the pins, so I call the “serSend0” function once to send the data of the LAPinin. LAPinin contains the current logic state of port C, this is the only data we will ever be interested in.

And the initialization process is now finished.

You will notice after “init\_main” is called, “main\_loop” is called repeatedly inside a while loop that runs forever. “main\_loop” is empty so nothing happens, it just keeps the processor running forever.

After initialization, when a pin changes state, the interrupt routine is entered. This routine is written inside “ISR(SIG\_PIN\_CHANGE1)”. Inside, all it does is send the status of LApinin to the computer, if the serial port is busy, the status is stored in a buffer until the serial port is not busy.

Time to discuss “serSend0” and the FIFO buffer. This function and all functions related to serial port and the FIFO buffer

\*FIFO means first in first out, read Wikipedia about it, I'll do my best to explain it myself.

When “serSend0” is called, the program checks if there's room in the buffer, if there is, the data to be sent is stored at the tail of the buffer (the buffer is like a line, with a head and a tail, think of a grocery store, first in line is first to get out of the line). If the serial port is not busy, the data at the head of the buffer is put into UDR0, which makes the serial port start transmitting the data to the computer.

When a transmission is finished, “ISR(SIG\_USART\_TRANS)” is the interrupt routine that is called. If the buffer is empty, it declares the serial port not busy, but if the buffer is not empty, it takes data from the head of the buffer, and transmits it.

If the serial port detects data coming from the computer, then “ISR(SIG\_USART\_RECV)” is the interrupt routine that is called. The computer interface will send a new pin change mask to enable/disable triggering on any of the pins, and in this event, the mask is applied immediately to LAmask, which we discussed before.

If you feel like changing my code for other processors, “config.h” defines the chip used, the clock speed, the pin change mask, and the baud rate of the serial port, while “pins.h” defines which port is to be used.