Designing an Array of Arduino MCUs

By Allan M. Schwartz, allans@CodeValue.net

1 Introduction

The challenge was to build a kinetic art wall of 24x12 pairs of clock hands. A cost effective implementation is 288 Arduino's networked together, each controlling two stepper motors directly. There are several technical hurdles in connecting multiple or in this case hundreds of interconnected Arduinos. Because of the nature of kinetic art, time synchronization are of upmost importance. Efficiency of communication is also important. The cost and complexity of the interconnection is another factor.

For these reasons, we have designed a two-dimension array with I^2C communication buses.

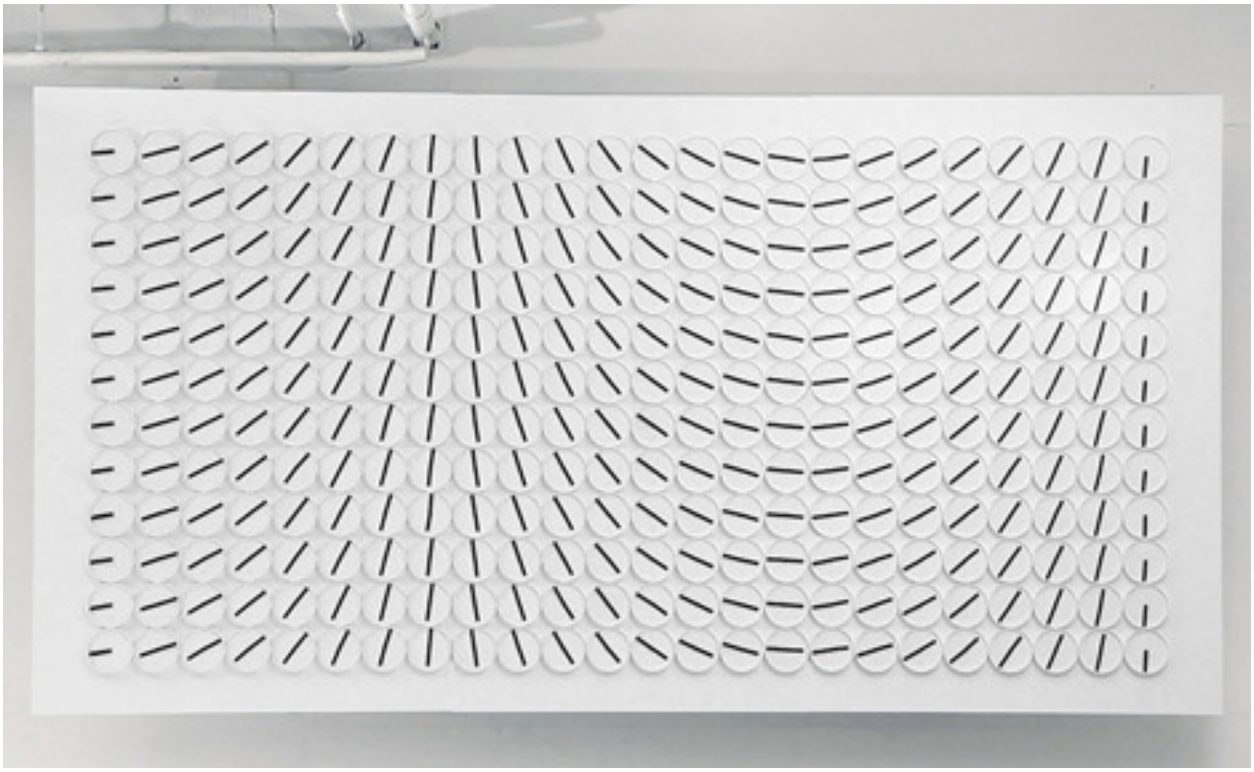

Figure 1: Clock Wall (www.humanssince1982.com)

2 Goal

- The first goal is to support a large array of Arduinos Elements. Each single ***Node Element*** is individually addressable.

- The array is organized into Rows and Columns. We define a special kind of Element called a ***Row Controller***, which is connected to all the row's ***Node Elements*** over a communications bus. We will be using the I2C bus for this purpose, because it is well-supported by the Arduino hardware and library software. The I2C implementation of the Arduino supports 112 device addresses.
- An array using the typical I2C bus with 7-bit Slave Address could support up to 111 rows and 111 columns. There may be practical limitations such as bus-capacitance which would limit the number of Nodes per row or per column.
- Bi-directional Communication from the ***Array Controller*** to the ***Node Elements*** through the ***Row Controller***, acting simply as a liaison is required.
- The application requires Low Latency and precise time-based synchronization.

3 System Architecture

Figure 2: Pictorial Architecture

The above diagram is representative of the array-like architecture, with as an example, 4 rows, and 10 columns of Node Elements.

In this design, the array of controllers is comprised of 3 different elements:

- 288 ***Node Elements*** arranged in 12 rows of 24 elements
- 12 ***Row Controllers*** (depicted as the 4 larger MCUs, above)
- A single ***Array Controller*** (depicted as the single Intel Edison, above)

3.1 Node Elements

The ***Node Element,*** the most numerous element of the design, requires 288 MCUs. The selection is critical. An 8-bit MCU was chosen because of the low cost; and a member of the Arduino family was chosen because of ease of developing, and the ease of programming. In selecting a small-form factor Arduino or Arduino compatible, we have considered these choices:

Model	Processor	SRAM/FLASH	Analog Input Digital IO/PWM	Comm
Arduino Micro 48mm x 18mm	8-bit ATmega32U4 16 MHz	2.5 k / 32 k	Ana: 12 Dig: 20/7	I2C SPI Micro USB
Arduino Mini ***(retired)***	8-bit ATmega328 8 MHz or 16 MHz	1 or 2K / 16 or 32K	Ana: 4 Dig: 16/6	I2C SPI (no lib.) 6-pin header
Arduino Nano v2.x 45mm x 18 mm	8-bit ATmega168 16 MHz	1 k / 16 k	Ana: 8 Dig: 14/6	I2C SPI (no lib.) mini-B USB
Arduino Nano v3.x 44 mm x 18 mm	8-bit ATmega328P 16 MHz	2 k / 32 k	Ana: 8 Dig: 14/6	I2C SPI (no lib.) mini-B USB
Arduino Pro Mini 33 mm x 18 mm	8-bit ATmega328 16 MHz	2 k / 32 k	Ana: 6 Dig: 14/6	I2C SPI (no lib.) 6-pin header
MKR1000 ***(coming soon)***	32-bit Atmel SoC 48 MHz	32 k / 256 k	Ana: 7 Dig: 8/4	I2C SPI USB
Sparkfun Teensy 3.2	32-bit ARM Cortex-M4 72 MHz	64 k / 256 k	Ana: 21 Dig: 34 /12	SPI, I2C, I2S, CAN, IR, UARTs, USB

Table 1: Choices for Node Element Arduinos

The Arduino Mini/Pro Mini are poor choices, because they have been discontinued and are difficult to program. The Teensy is too expensive, in spite of its catchy name. The MKR1000 is not yet available. There are a number of choices based on the ATtiny85, not listed in the table above – all deemed unsuitable because this processor would be under-powered.

This leaves only the Arduino Micro and the Arduino Nano as suitable models. The Arduino Nano v3 has been selected as our *Node Element*, because of its cost, availability and suitability for this project.

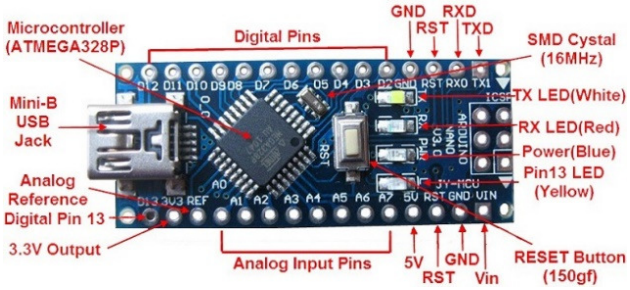

Figure 3: Arduino Nano v3

3.2 Row Controller

The selection of the *Row Controller* is difficult, because this array architecture requires the *Row Controller* to have two channels of I2C bus communication. Here are some of the boards under consideration:

Model	Processor	SRAM/FLASH	Analog Input Digital IO/PWM	Comm
Arduino Due	AT91SAM3X9E 84 MHz	96 k / 512 k	Ana: 12 Dig: 54/12	(2) I2C SPI USB
STM32F0 Discovery *$9.44*	32-bit ARM	8 k / 64 k	Plenty	Plenty
MSP432 Launchpad *$12.99*	MSP432P401R 48 MHz	64 k / 256 k + FRAM	plenty	4 I2C, 8 SPI, 4 UART

Table 2: Choices for Row Controller

One idea we considered, but soon rejected, was to take a standard Arduino Uno, and replace the ATmega328 with an ATmega328NP – a newer chip that supports two channels of I2c. However, there are several complications to this implementation.

Both the STM32F0 and MSP432 boards would be implementable, and would be sufficient in terms of CPU, Memory, Inputs and Communication. Both come on fairly inexpensive evaluation boards from the chip manufacturer, which could be used almost with no modification. Both provide a HAL (Hardware Abstraction Library), and an RTOS environment. So both could be workable solutions.

However, the easiest to implement solution would be using the Arduino Due, which has two I2C interfaces, and a huge number of GPIO pins. Because it is an Arduino, it has the same development environment as the Arduino Nano used as the *Node Elements*. In fact, the two devices could share a substantial amount of code.

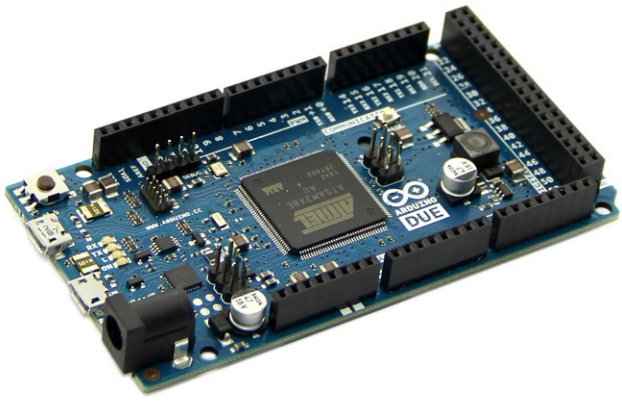

Figure 4: Arduino Due

3.3 Array Controller

The *Array Controller* is the single device, controlling the entire array. It is an ***IoT*** element, therefore a modern 32-bit MCU with at least 128k of FLASH and built in networking, and library support of TCP/IP is absolutely required.

Model	Processor	SRAM/FLASH	Analog Input Digital IO/PWM	Comm
Particle Photon	32-bit STM32F205 120 MHz	128k / 1 Mb flash	18 GPIO/7 PWM	SPI, I2S, I2C, CAN, USB Wi-Fi Ethernet
Raspberry Pi 2B	quad-core 32-bit ARM Cortex-A7 900 MHz	1 GB / µ-SD card	40 GPIO	SPI, I2C, CAN, CSI, DSI, 4 USB, Ethernet
Raspberry Pi Zero	Single core 32-bit ARM Cortex	128 M / µ-SD card	40 GPIO	plenty
TI CC3200	32-bit ARM Cortex- M4 core 80 MHz	256 KB /	4 channel ADC 27 GPIO	SPI, UART, I2C, SD/MMC, I2S/PCM, Wi-Fi Ethernet
Intel Galileo Gen2	32-bit Intel Quark X1000 400 MHz	512 k / 512 k 8 Mb SPI Flash µ-SD card	Ana: 6 Dig: 14/12 PWM	Plenty, incl. 100 Mb Ethernet
Intel Edison	32-bit Intel Atom Z34xx	1 GB RAM / 4 GB Flash	Ana: 6 Dig:40/4	I2C, SPI, USB, Wi-Fi Ethernet
Sparkfun **Think IOT** *(plus other com- peting models)*	ESP8266 (32-bit Xtensa LX106) 80 MHz	64k + 96 k / 512M	5 digital	I2C, SPI Wi-Fi Ethernet
Arduino Zero	32-bit ARM Cortex ATSAMD21G18 48 MHz	32 k / 256 k Flash	Ana: 6 Dig: 20/18	plenty

Table 3: Choices for Array Controller

The specific MCU for the *Array Controller* has not yet been chosen. It has been suggested we implement this using a Raspberry Pi. I recommend, instead, either an Intel Edison or a TI CC3200 or a Particle Photon. These are popular ***IoT*** MCUs, which we should gain experience with.

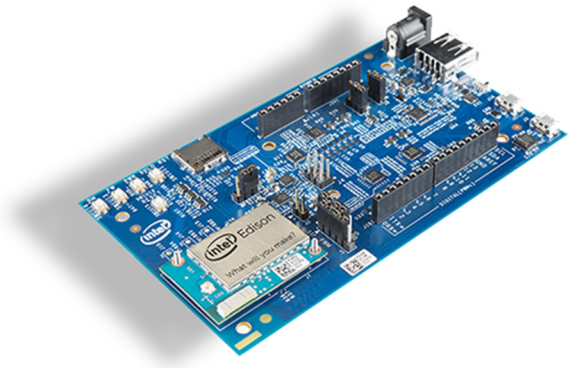

Figure 5: Arduino Due

4 I2C Master/Slave Protocol

Each Message from the Master to a Slave is sent as an I2C message, consisting of several octets (bytes represented in the protocol). For every request message there is a corresponding response message, also sent as an I2C message.

Because the I2C Bus lengths are approach the physical limit of the bus (5m), and also approach the logical limit of I2C (112 addressed clients), we will try to ensure error-free communication. Therefore, each message contains a checksum, to ensure the request message and the reply message are received error-free.

The protocol is completely synchronous – the master transmits the request message, and then requests the response message from the slave. If *NAKed* by the I2C low-level circuitry or not received, or received with an error, the master can resend the frame *n* times. The slave always replies with a reply message, which is requested by the master.

4.1 Message Formats

Each request message looks like this:

```
struct requestMessage {
    _u8     cmd;            // a command byte
    _u8     len;            // length of the frame, always 9
    _u8     naddr[2];       // node address
    _u16    args[2];        // up to 2 additional 16-bit arguments
    _u8     checksum;       // checksum, for safety
}
```

This message (simply a sequence of bytes) is layered on top of I2C hardware framing, addressing and acknowledgement.

The response message is simply an **`ACK`** (packet acknowledgement) or **`NAK`** (packet negative acknowledgement) with a one byte result code.

```
enum responseType { ACK='A', NAK='N' };
```

And the response message looks like this:

```
struct responseMessage {
    _u8    cmd;             // ACK or NAK
    _u8    len;             // length of the frame
    _u8    info;            // 1 additional byte of information
    _u8    checksum;        // checksum, for safety
}
```

Again, these are layered on top of the I2C frame mechanisms. The I2C Master requests the message from the I2C Slave, and the I2C Slave responds with this response message.

4.2 Illustrated Packet Formats

These packet forms, illustrated by `packetdiag`, looks like this:

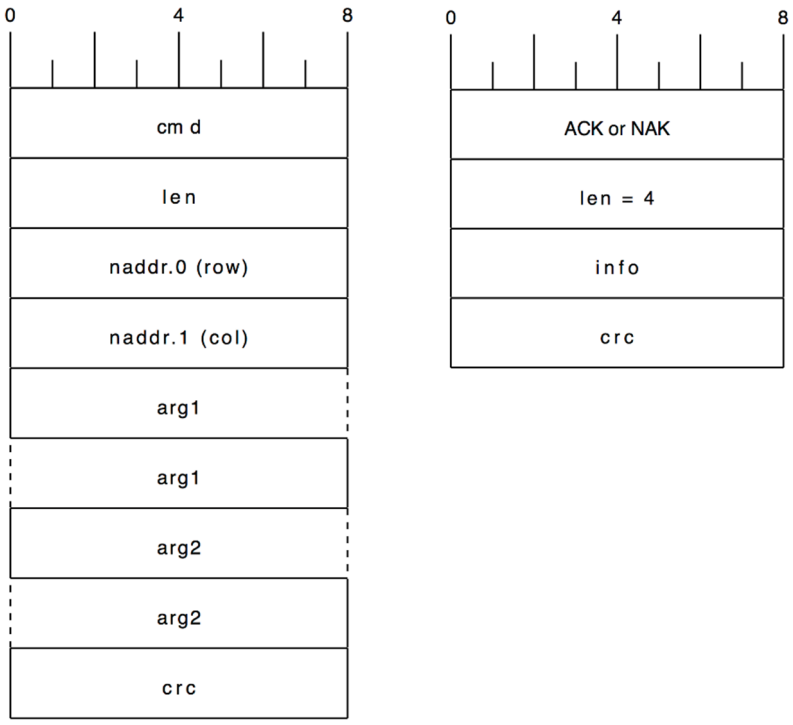

Figure 6: Request and Response Messages

4.3 Remote Procedure Calls

With the I2C Master/Slave Protocol defined above, all of the interactions between the Master and Slave (one of the *Node Elements)* can be defined as Remote Procedure Calls (RPCs). Each RPC is two I2C messages: the Request message diagrammed above, sent from the Master to the specific *Node Element*, followed immediately by Response message diagrammed above.

4.4 Array Controller to Row Controller

The *Node Element* address (`naddr`) is 2 bytes in total, with the row number [1..12] in the first byte, and the column number (or element number) [1..24] in the second byte.

Each *Node Element* knows its element number. It is also equal to its I2C slave address.

Each *Row Controller* knows its row number; it is also equal to its I2C address (on a unique I2C bus).

For messages sent from the *Array Controller* to a *Row Controller*, ***`naddr[0]`*** is set to that Row Controller's row address. Then, the Row Controller re-forwards the message to the ***`naddr[1]`*** *Node Element's* address.

Here are the calls from the *Array Controller* to a *Row Controller:*

function	*Brief Synopsis*	*Params*	*Returns*
byte **LED_OnOff**(*node* **naddr**, *boolean* **onOff**, *boolean* **onSync**)	Send to node ***naddr*** a message to, when synced, turn the LED on or off	**naddr:** node address **onOff**: if true turn on, otherwise turn off **onSync**: if true, do this when synced, otherwise immediately	Return value from the *Node Element*.
void **LED_nop**(*node* **naddr**)	Send to node ***naddr*** a no-op message.	**naddr:** node address	Return value from the *Node Element*.

Table 4: Calls from the Array Controller to the Row Controller

4.5 Row Controller to Node Element

Finally, the *Node Element* executes these commands. In all cases, the received 'naddr' is the address of the *Node Element* receiving this message.

Here are the calls from the *Row Controller* to a *Node Element*:

function	*Brief Synopsis*	*Params*	*Returns*
byte **LED_OnOff**(*node* **naddr**, *boolean* **onOff**, *boolean* **onSync**)	… when synced, turn the LED on or off	**naddr:** our own address **onOff**: if true turn on, otherwise off **onSync**: if true, do this when synced, otherwise immediately	ACK or NAK
void **LED_nop**(*node* **naddr**)	… do nothing on the next sync pulse.	**naddr:** our own address	ACK

Table 5: Calls from the Rows Controller to the Node Element.

5 Experiment

The purpose of the experiment is to determine if the *Row Controller* can communicate with multiple *Node Elements*, and measure latency and throughput. Can the *Row Controller* communicate to 24 *Node Elements*, with predictable and low enough latency?

To perform this experiment, we will place on a breadboard 1 Arduino Uno acting as the *Row Controller*, and 4 Arduino Nanos as the *Node Elements*.

Later we will scale up this experiment to 12 *Node Elements*, and predict the latency with 24 *Node Elements*.

For simplicity of implementation, we will control the pulse of the build-in LED (on pin D13), and also measure these pulses on a logic analyzer. Later, we will control actual clock stepper motors, however the logic analyzer traces will provide a clear indication, in fact the most accurate measurements of latency and synchronization.

5.1 Arduino Signal Hookups

Each Arduino needs 6 pins hooked up to perform this series of experiments.

function	*Arduino UNO (master)*	*Arduino Nano (Element 10)*	*Arduino Nano (Element 11)*	*Arduino Nano (Element 12)*	*Arduino Nano (Element 13)*
SYNC	D2	D2	D2	D2	D2
SDA	SDA	A4	A4	A4	A4
SDL	SDL	A5	A5	A5	A5
LED	*n/c*	D13	D13	D13	D13
GND	GND	GND	GND	GND	GND
+5V (VCC)	+5V	+5V	+5V	+5V	+5V

Table 6: Arduino Signal Hookups

The following photograph shows the 1 Arduino Master and 4 Arduino Slaves hooked up per this table. Note that I have labeled 5 (of the 8) bus strips: SYNC, GND, SDA, SDL, +5V. All of those signals are jumped to those bus strips.

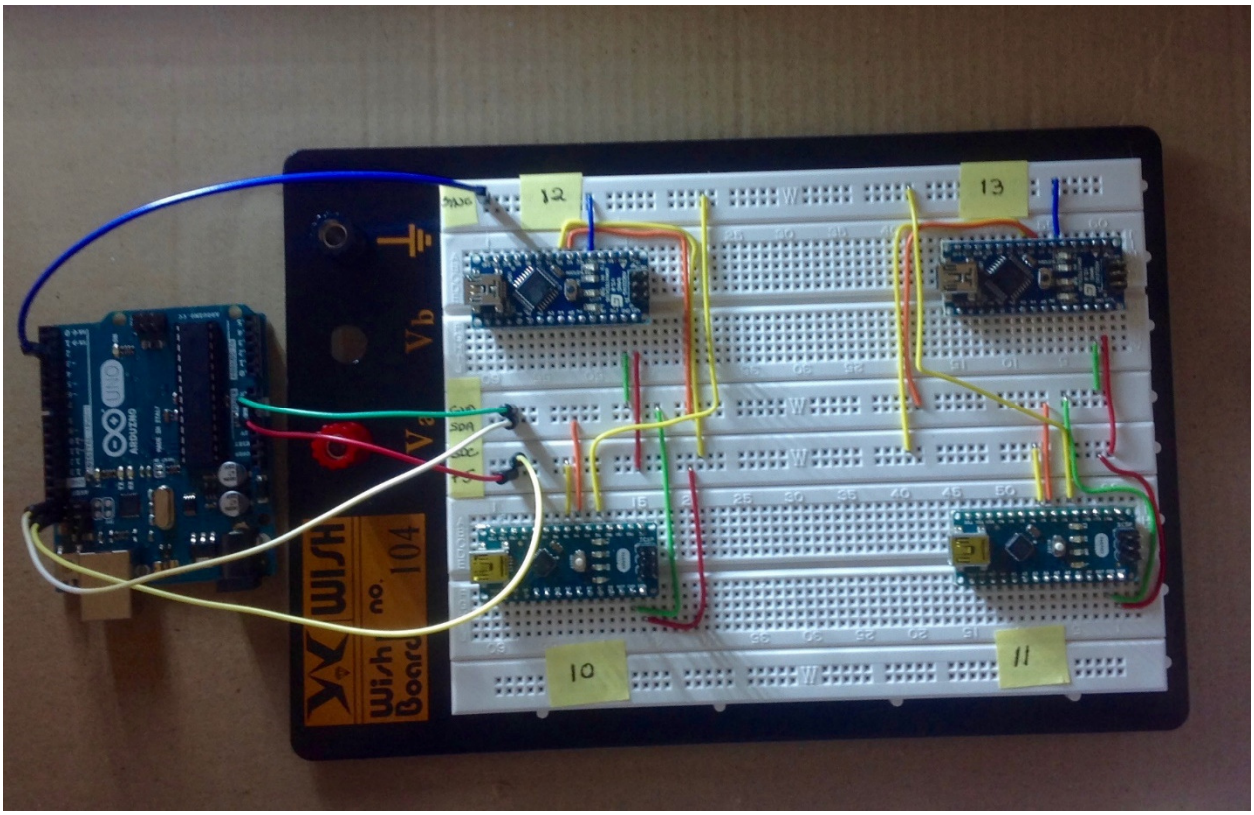

Figure 7: Arduinos on a Breadboard with Signals Connected

5.2 Logical Analyzer Connections

We are using a Saleae Logic Pro-16 to capture and display the signals in performing this experiment. It is capable of monitoring 16 Analog or Digital signals. These are the signals we want to monitor:

Lead	Lead color	Label	Purpose
0	Black	SDA	I2C
1	Brown	SDL	I2C
2	Red	Master MOSI	Master debug SPI
3	Orange	Master SCK	Master debug SPI
4	Yellow	Slave MOSI	Slave debug SPI
5	Green	Slave SCK	Slave debug SPI
6	Blue	Slave #10 LED	Slave LED
7	Purple	Slave #11 LED	Slave LED
8	Black	Slave #12 LED	Slave LED
9	Brown	Slave #12 LED	Slave LED
10	Red	D2 SYNC	SYNC pulse

Table 7: Logic Analyzer Signal Connections

In addition to measuring the result of the experiment, seen as the Slave LEDs, it is very useful to observe the I2C signals SDA and SDL.

Also, in debugging a Master/Slave protocol, it is often useful to have a debug *printf* channel to view. The normal Arduino "Monitor" is too slow and can only be connected to a single Arduino. We use the SPI bus on the Arduino Master, and the SPI bus on a single Arduino Slave as two separate *printf* channels. With the ability to capture both of these with the Logic Analyzer, this enables us to debug the protocol quickly.

After connecting the Logic Analyzer probe lines to the signals we wish to monitor, we now have this setup:

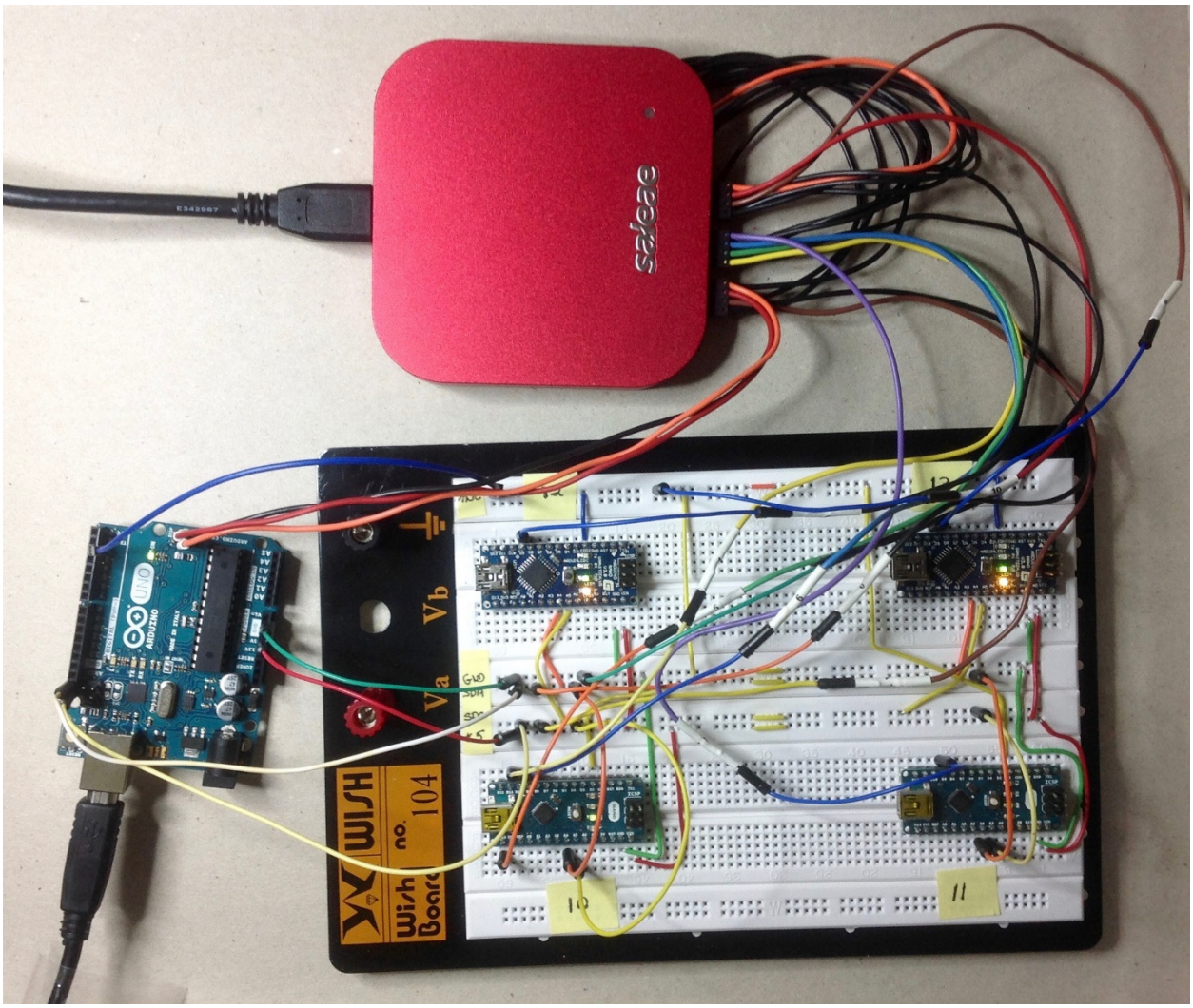

Figure 8: Arduinos on Breadboard with Logic Analyzer Connections

5.3 Experiment #1

The first experiment we will perform is to have each of the *Node Elements* pulse their LEDs for 50 ms. We will do this 3 times in a row, so that there is a clear pattern on the logic analyzer.

The results can be seen in Figure 9, below. The time scale is about 1 second. Here you can see the 3 blips, from the 4 *Node Elements* (in blue, purple, black and brown).

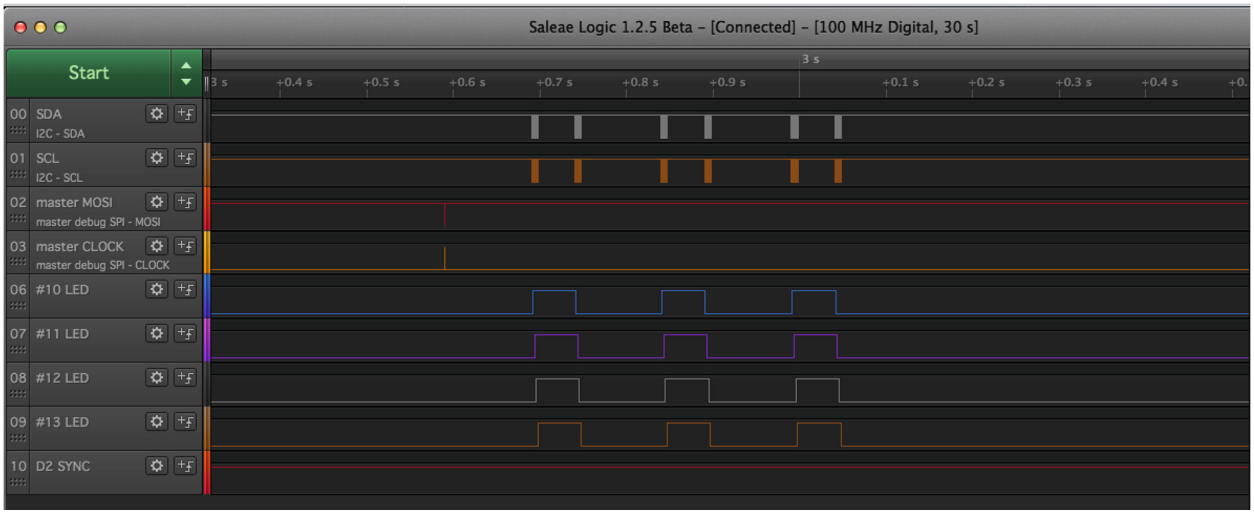

Figure 9: Experiment 1, 1 second scale

Now, let's zoom in (in the timescale) by a factor of 10. In Figure 10 below, the time scale is about 100 ms. across. We can see one pulse (which itself is 50 ms. long), from the 4 *Node Elements*.

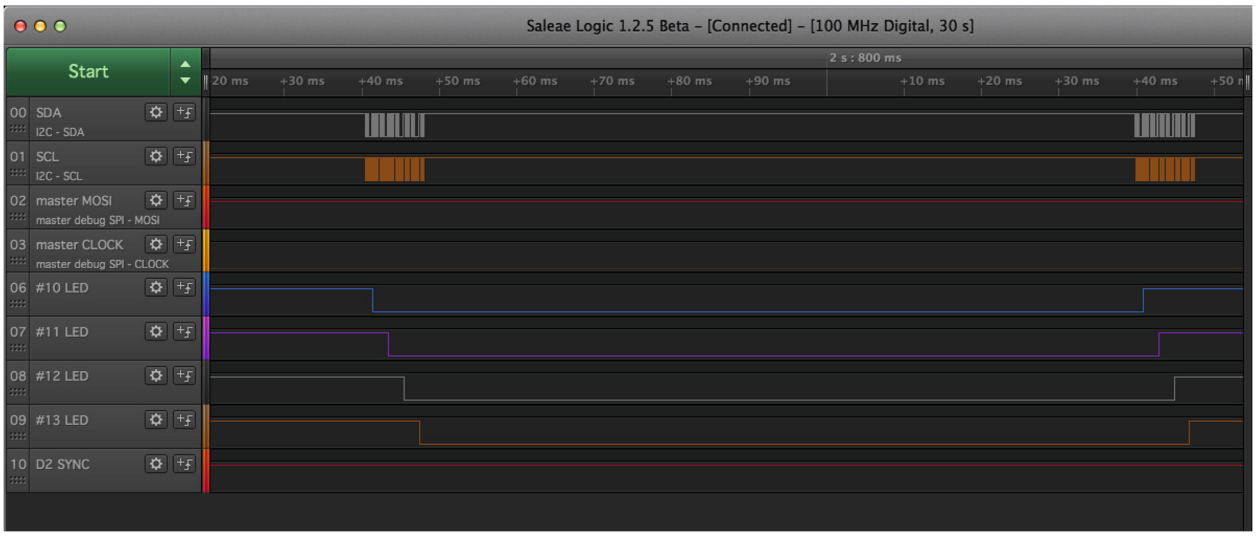

Figure 10: Experiment 1, 100ms scale

Notice that in Figure 10, it is quite apparent that the pulses (in blue, purple, black, and brown) are not synchronized, they are staggered by about 2ms off each.

We can zoom in again, by another factor of 10. In Figure 11, below, the time scale is about 10 ms. across.

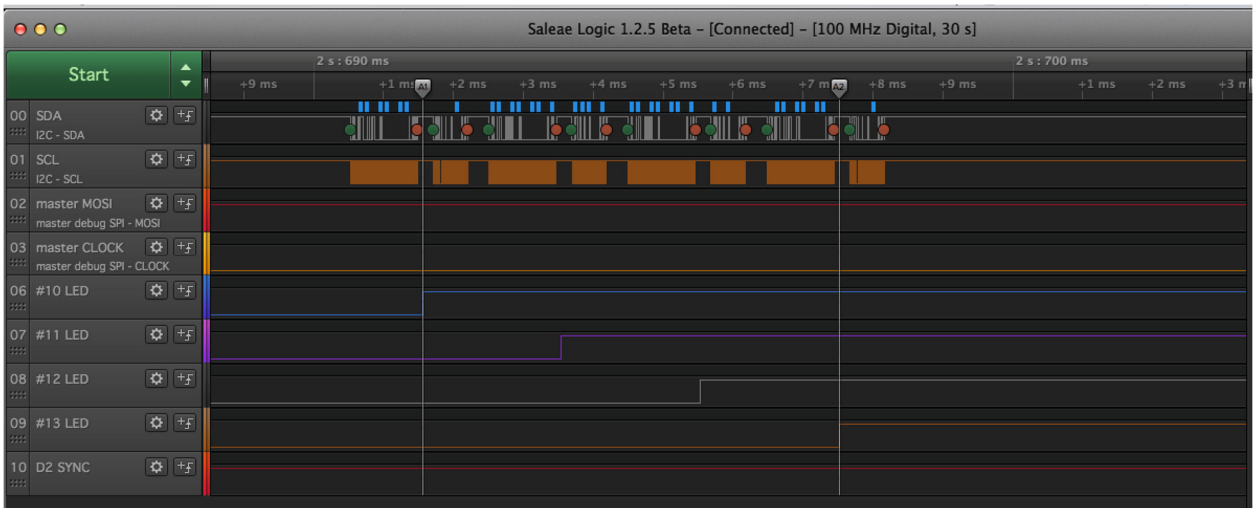

Figure 11: Experiment 1, 1 ms scale

In this picture we can see the beginning of the Slave pulses (in blue, purple, black, and brown). We can also see the 8 I2C messages: 4 pairs of a request message followed by a response message. I've placed timing marks at the start of the blue (Node Element #10) pulse, and the start of the brown (Node Element #13) pulse, and the difference between these timing marks is 6 ms.

Even though the delays are small, with many *Node Elements* the delays would add up to being visually noticeable.

5.4 Experiment #2

In the second experiment we will have each of the *Node Elements* pulse their LED for 75 ms. And again we will do this 3 times in a row, so that there is a clear pattern on the logic analyzer. However, they will be synchronized by an additional timing signal from the Master (acting as a *Row Controlle)r*, which we can see on logical analyzer captures, as the red signal labeled as "D2 SYNC".

The results can be seen on Figure 12, below. The time scale is about 1 second. Here you can see the 3 blips, from the 4 Node Elements (in blue, purple, black and brown).

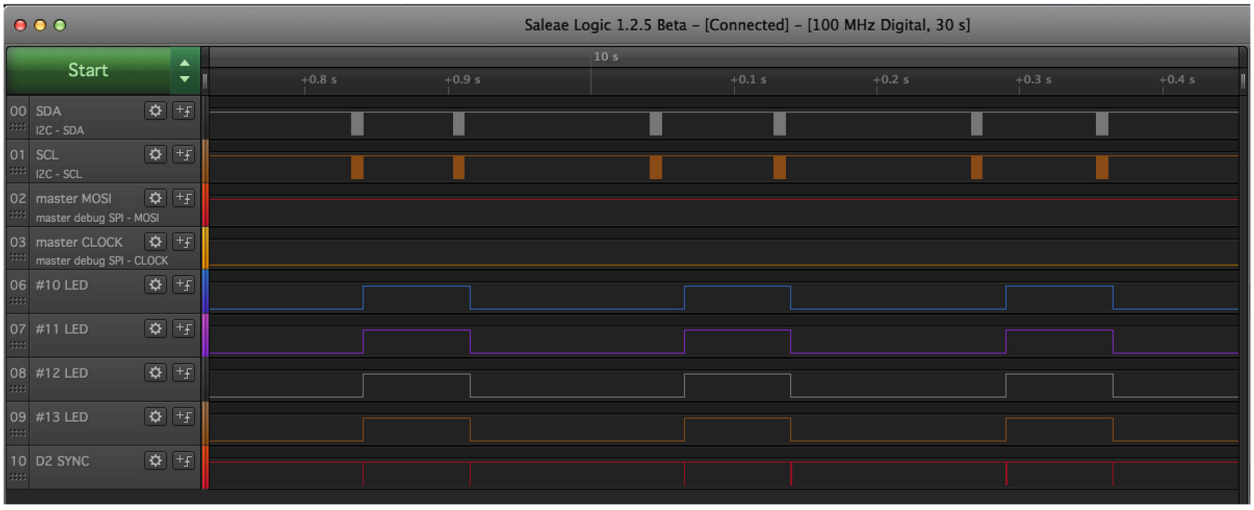

Figure 12: Experiment 2, 1 second scale

Now, let zoom in, in the timescale, by a factor of 10. Now the time scale is about 100 ms. across. We can see one pulse (which itself is 75 ms. long), from the 4 Node Elements.

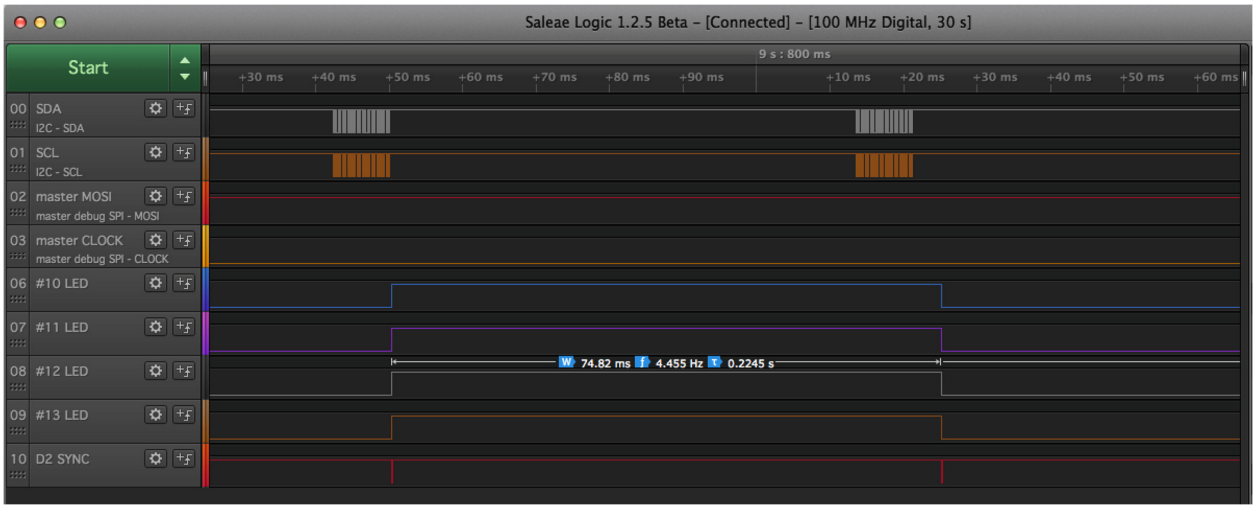

Figure 13: Experiment 2, 100 ms scale

Notice that in Figure 13 above, it is quite apparent that the pulses (in blue, purple, black, and brown) are now perfectly synchronized.

We can zoom in again, by another factor of 10. Now the time scale is about 10 ms across.

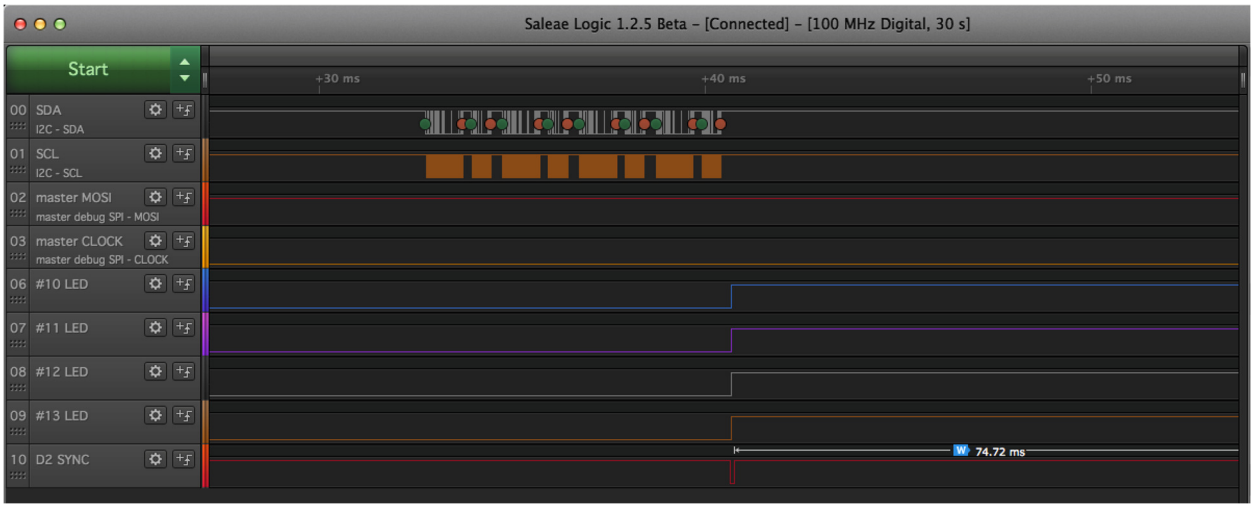

Figure 14: Experiment 2, 10 ms scale

In Figure 14 above, we can see the 8 I2C messages: 4 pairs of a request message followed by a response message. Then, there is a "D2 SYNC" pulse, which is created by the Row Controller, and triggers an interrupt in each of the Node Elements. Then, we can see the Slave pulses start (in blue, purple, black and brown) all perfectly synchronized.

Using this technique, the Row Controller can precisely time the execution of all commands to the Node Elements.

6 Alternative Architectures

Some discussion of the data rates and throughput amounts is necessary. If the Array Controller has to construct and send messages to all 288 Node Elements over I2C, and each 10-byte message and 4-byte response takes a total of 2ms, then it would take the Array Controller almost .58 seconds to send all the commands to the Row Controllers, which would then take another 48 ms to communicate with their 24 Node Elements. So this would not be suitable if there was a requirement to communicate commands more often than once every second. The protocol could likely be optimized to yield, at best, 2 or 3 commands per second.

The standard Arduino I2C bus rate uses the slower 100-kHz rate allowed in the spec, which allows for compatibility with more I2C devices. This communication rate could easily be increased to the 400 k-Hz Fast-mode, and this would quadruple our throughput.

6.1 Using SPI

The SPI communication bus is considerably faster. In the standard Arduino implementation, the I2C bus speed in 100KHz, and the SPI bus speed is 4Mhz, so its 40 times faster. Using the SPI bus, or using a Hybrid approach of both I2C and SPI bus in the design, we can come up with seven alternative designs.

6.1.1 Array Controller -> Row Controller over SPI; Row Controller -> Node Element over I2C

This design would require 3 + 12 comm. signals from the Array Controller. A full set of data from the Array Controller (288 10-byte messages, with 4-byte responses) would take a total of 14 ms, then the row controllers would still take 48 ms to communicate with their 24 node Elements. This would allow one command (to each Node Element) every 64 ms. Or one command (to each Node Element) every 26 ms using I2C Fast mode.

6.1.2 Array Controller -> Row Controller over SPI; Row Controller -> Node Element over SPI

This design would require 3 + 12 comm. signals from the Array Controller. And also would require 3 + 24 comm. signals from the Row Controller. The Arduino Due is still a good choice for a Row Controller, and has sufficient GPIO bits for this purpose. A full set of data from the Array Controller would still take a total of 14 ms, then the row controllers would then take less than 2 ms to communicate with their 24 node Elements. This would allow one command (to each Node Element) every 15 ms. This appears to be optimal in terms of communication speed, however, the worst in terms of wiring. Point-to-point wires for all the chip selects would be required.

6.2 SPI Broadcast Only, no response path

We could relax the requirement of bi-directional communication, because actually, only the Array Controller needs to talk to the Node Elements commands. The message response adds no value to the design, except error reporting.

In this design, the Array Controller would broadcast the packets use SPI as a transport, however, only MOSI and SCLK and SS would be connected, no MISO. The Node Elements would have no way to communicated with the Array Controller. Each Node Element would hear all messages, so each

message would be addressed to one or more Node Elements in the message bytes. A full set of data from the Array Controller would take between 2 and 3 ms (because no acknowledgement is read). There would be a minimum of signal wires required, and the Row Controllers are not needed (reducing the CPU count from 301 to 289).

6.3 Using an H-Bridge

An H-Bridge is a custom chip, designed to control 1 or more motors; controlling clockwise and counter-clockwise rotation, braking and stopping. One purpose of an H-Bridge is to supply more current to the motor than is possible with a processor GPIO port. Another purpose of an H-Bridge is to off-load the processor, and deal with motor commands at a higher level, including doing whatever PWM (pulse-width modulated) commands necessary.

A possible design is to construct a Node Element with just an H-Bridge, (such as the TB6612FNG) and a SPI-compatible shift register and latch (such as a 74595).

Then each Row Controller would transmit commands to the H-Bridge, via the shift registers. The commands would naturally be synchronized using the Latch mechanism in the shift register. This would eliminate the Arduino Nanos, and reduce the processor count from 301 to 13.

Another design possibility would be to eliminate the Row Controllers, and the Array Controller would transmit to all 288 Node 74595 registers, at 4 MHz, or roughly .8 µs per byte.

6.4 Summary

A summary of these different Architectures and the speeds is in table 8, below:

	Array Cont. -> Row Cont.	***Time for Msgs***	***Row Cont. ->Nodes***	***Time for Msgs***	***total all msgs***
1	100-kHz I²C	2 ms * 288	100-kHz I²C	2 ms * 24	624 ms
2	400-kHz I²C	.5 ms * 288	400-kHz I²C	.5 ms * 24	156 ms
3	4-Mhz SPI	.05 ms * 288	100-kHz I²C	2 ms * 24	62 ms
4	4-Mhz SPI	.05 ms * 288	400-kHz I²C	.5 ms * 24	26.4 ms
5	4-Mhz SPI	.05 ms * 288	4-Mhz SPI	.05 ms * 24	15.6 ms
			Array Cont. ->Nodes		
6		Via Broadcast SPI, 10 byte msgs	4-Mhz SPI	8 µs * 288	2.5 ms
7		Via 74595 & H-Bridge, 1 byte per H-Bridge	4-Mhz SPI	.8 µs * 288	.25 ms

Table 8: Architecture Comparison, speeds

Note that we are considering the case where the Array Controller sending individual messages to each Node Element, every time a message is to be sent. There might also be a way to leverage the fact that many messages will be the same. A form of a broadcast protocol, or wild-carded node addresses could be invented, to further reduce message traffic.

7 Conclusions

Using this array architecture, with a hierarchy of *Array Controller*, *Row Controller*, and *Node Elements*, we can produce coordinated synchronized LED pulses or other GPIO actuated devices, such as stepper motor. Messages to all 288 Node Elements using 100-kHz I2C would be relatively slow, approximately ½ second. The SPI bus promises much faster communication and better throughput. Additional research is required to find the optimal design.

7.1 Future Research

Three sets of experiments would move us closer to the final design

1. Redo the experiments in Section 5.3 and 5.4 using the SPI bus instead of I2C.
2. Experiment with the TB6612FNG H-Bridge.
3. Experiment with the actual stepper motor, and verify the Nano can control it. Design a higher level command set to operate the stepper motors.