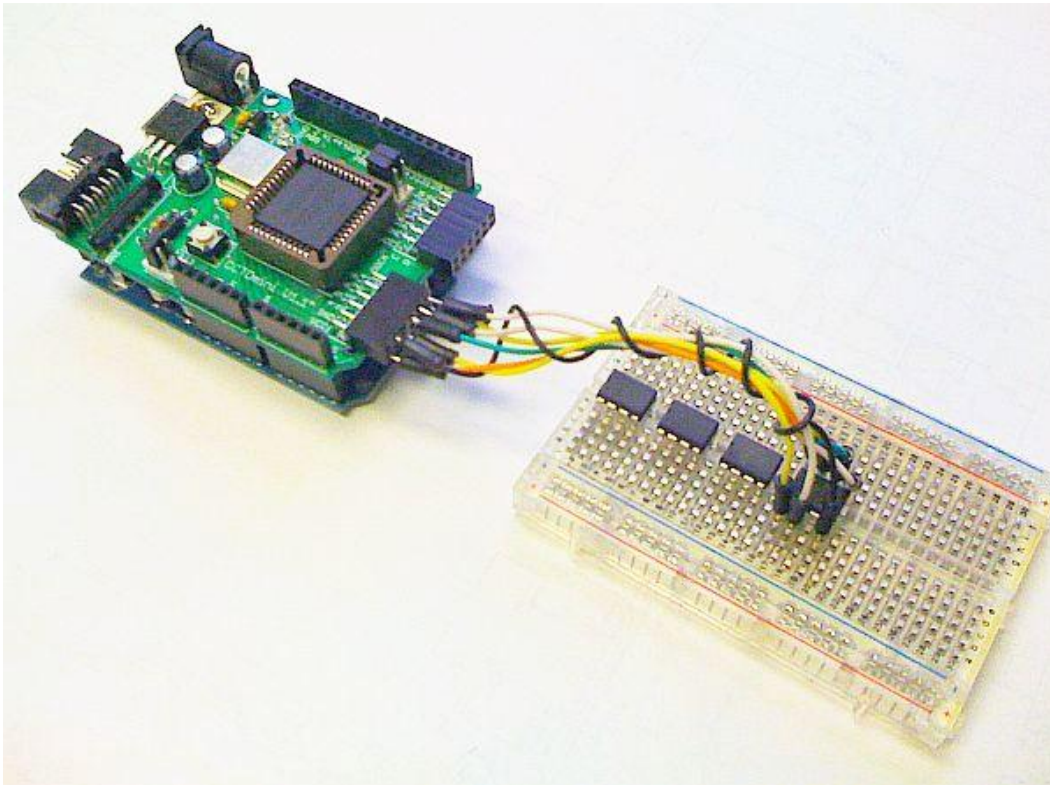


# Arduino SRAM Expansion: Featuring the Amani 64 CPLD Shield

By Eric Rogers for majolsurf.net

**Abstract:** One surprising limitation of the Arduino is that it offers only 2KB of SRAM. Arduino.cc offers a quick and easy [expansion method](#) using the 23K256, a 32KB SPI SRAM device. What happens if more storage is needed while keeping the convenience of the SPI interface? There are few larger SPI SRAM devices available and tend to be more expensive than the equivalent number of 23K256 devices. The larger devices also add transfer time due to additional addressing bits. **In this project you will learn to address multiple 23K256 devices by injecting addressing bits into the existing data packet, without modifying its length, and decoding those bits to select the appropriate SRAM bank.** The Amani 64 will be mostly transparent in this process and cause no data delays. A secondary benefit of using the Amani here is that it provides a 5V to 3.3V interface without the need for additional translation circuitry.



**Figure 1.1 The Amani Stacked with the Arduino, Interfacing to 23K256 Devices**

## Parts List:

- (1) Arduino
- (1) Amani 64 CPLD Shield
- (2-8) 23K256 SPI SRAM IC's

[www.arduino.cc](http://www.arduino.cc)  
[www.majorsurf.net](http://www.majorsurf.net)  
[www.digikey.com](http://www.digikey.com)

## Project Code:

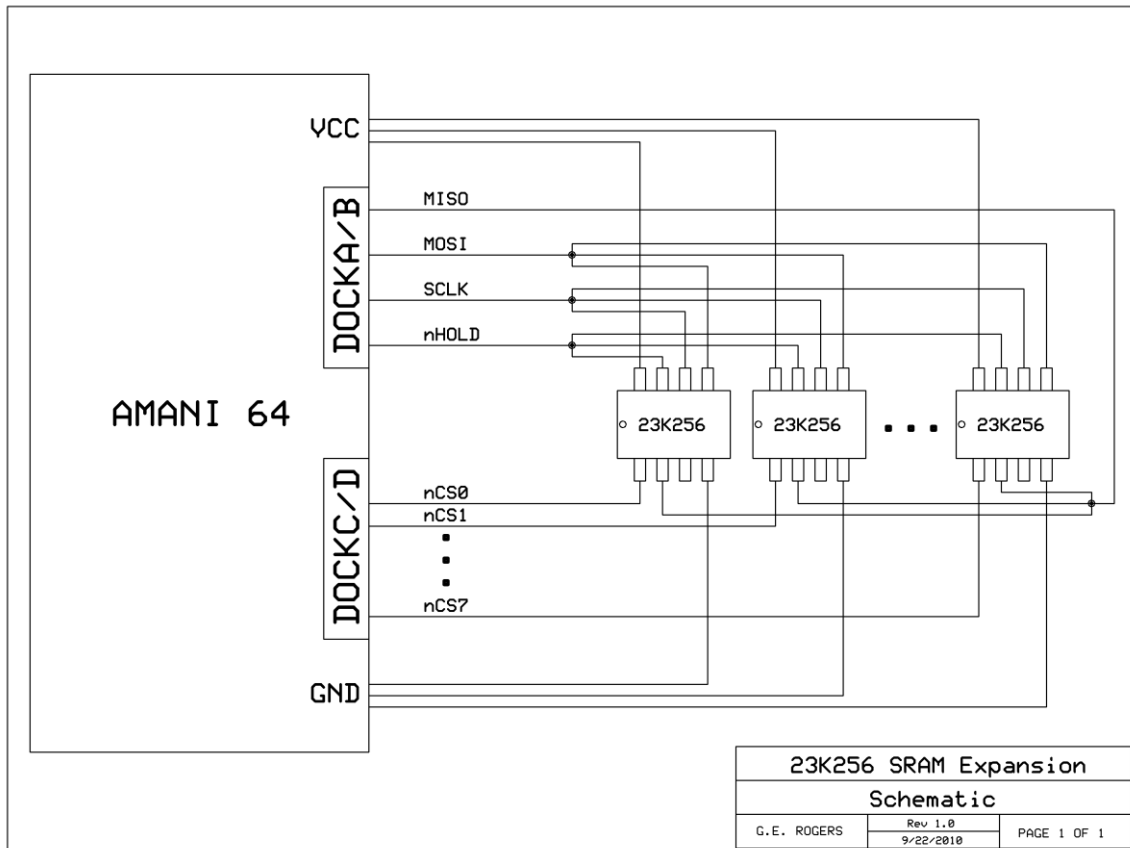
**Arduino:** Simple test code for demonstrating the formation of 24 bit addressing and generating SPI port data transfers.

[http://majorsurf.net/projects/Amani64/cc/100914%20256KB%20SRAM/Arduino/SPI\\_8MEMTEST.zip](http://majorsurf.net/projects/Amani64/cc/100914%20256KB%20SRAM/Arduino/SPI_8MEMTEST.zip)

**Amani64:** Implements a finite state machine to read and strip the packet of the injected addressing, select the appropriate SRAM bank, and handle the SPI data transfer transparent to both the Arduino and 23K256.

<http://majorsurf.net/projects/Amani64/cc/100914%20256KB%20SRAM/Amani64/100914%2023K256addr8.zip>

## Schematic:



**Figure 1.2 Wiring Multiple 23K256 devices to the Amani**

## Assembly Instructions:

1. Load Arduino with the **SPI\_8MEMTEST** sketch provided above. Depower and set aside.
2. Load the Amani 64 with the **23K256addr8.pof** file provided in the project provided above. Depower the Amani.
3. With both boards unpowered, seat the Amani onto the Arduino.
4. Populate a blank breadboard with as many 23K256 SRAM devices needed.

*The code provided for the Amani 64 handles up to 8 SRAM devices. More are possible with minor code changes.*

*The code provided for the Arduino is capable of addressing up to 2MB. However the test code portion only counts up to 128KB for speed of testing. Modify the (ADDR\_24 == 1) limit condition in the advance\_counters() function to accommodate higher addressing.*

5. Wiring the SRAM devices to the Amani:
  - A. Connect the nCS pin of each SRAM device to DOCKS C-D. Wire the first SRAM device to nCS0, the second to nCS1, and so forth.

*You must start with nCS0 and move up sequentially as the addressing scheme starts from nCS0. nCS selects which SRAM device will be addressed.*

DOCK	Amani Pin	nCS[X]
C3	16	nCS0
C2	18	nCS1
C1	19	nCS2
C0	20	nCS3
D3	21	nCS4
D2	24	nCS5
D1	25	nCS6
D0	26	nCS7

**Figure 1.3 Amani to 23K256 nCS Connections**

- B. Connect the SO (MISO) pins of each SRAM device to a common node on the breadboard. Connect that node via one wire to DOCKA2 (pin 5) of the Amani.
- C. Connect the SI (MOSI) pins of each SRAM device to a common node on the breadboard. Connect that node via one wire to DOCKA3 (pin 4).
- D. Connect the SCLK pins of each SRAM device to a common node on the breadboard. Connect that node via one wire to DOCKA0 (pin 8).

- E. Connect the nHOLD pins of each SRAM device to a common node on the breadboard. Connect that node via one wire to DOCKA1 (pin 6).
- F. Wire each SRAM IC's Vcc and GND connections to a **common distribution point. Do not daisy chain.**

*Daisy-chaining is connecting the power connection of an SRAM device to the previous device's connection. This is known to cause bad data quality. I prefer to connect each 23K256 power connection to separate Vcc and GND connections in DOCKSA-D.*

6. Set Jumpers JP5-JP6 on the Amani to "PB."
7. Set the Amani power jumper to JP2.
8. Connect the Arduino to a USB port, run the Serial Monitor in the Arduino IDE.
9. Observe the results. Errors will be displayed as found.

Hopefully you observed errorless data transfers for the four different test patterns provided. Try experimenting with the test patterns. If you experience multiple errors, check the wiring connections versus the schematic and Quartus pin file. Try experimenting with you power and ground connections, providing separate ground and power paths for each SRAM device. If problems persist, try wiring only one SRAM device with all other devices/connections removed. Once you get this one squared away, try adding additional stages as you feel comfortable.

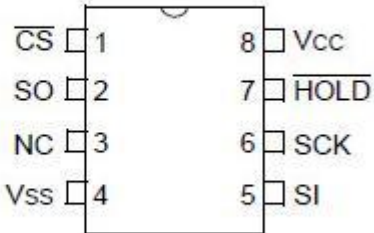
## How It Works:

### 23K256 SRAM Description and Protocol

The 23K256 is a 256Kbit SRAM memory device, accessible via SPI. Memory is organized by 32,768 x 8-bit blocks, thus effectively being a 32KB storage device. Serial data transfers take place via the data-in (SI) and data-out (SO) pins, clocked by SCK. Maximum clock speed is 20MHz, although for the scope of this project the Arduino will dictate a maximum speed of 16MHz. Communications can be paused via the HOLD signal although this function will not be used in this project. Access to the device is controlled via the chip select (CS) input. We will often refer to this signal as “nCS” as this is an active-low control.

**Pin Function Table**

Name	Function
$\overline{\text{CS}}$	Chip Select Input
SO	Serial Data Output
Vss	Ground
SI	Serial Data Input
SCK	Serial Clock Input
$\overline{\text{HOLD}}$	Hold Input
Vcc	Supply Voltage



**Figure 1.4 23K256 Pin Function and Layout**

The 23K256 instruction set is simple, only three bits are used. For the scope of this project we will only be concerned with read and write operations, thus only one bit, the least-significant bit (LSB) of the instruction field, will be manipulated. Two data operation modes are available to the 23K256 user; byte transfers and 32KB page. Only single-byte transfers will be used in the scope of this project. Status Register manipulations will not be performed.

Instruction Name	Instruction Format	Description
READ	0000 0011	Read data from memory array beginning at selected address
WRITE	0000 0010	Write data to memory array beginning at selected address
RDSR	0000 0101	Read STATUS register
WRSR	0000 0001	Write STATUS register

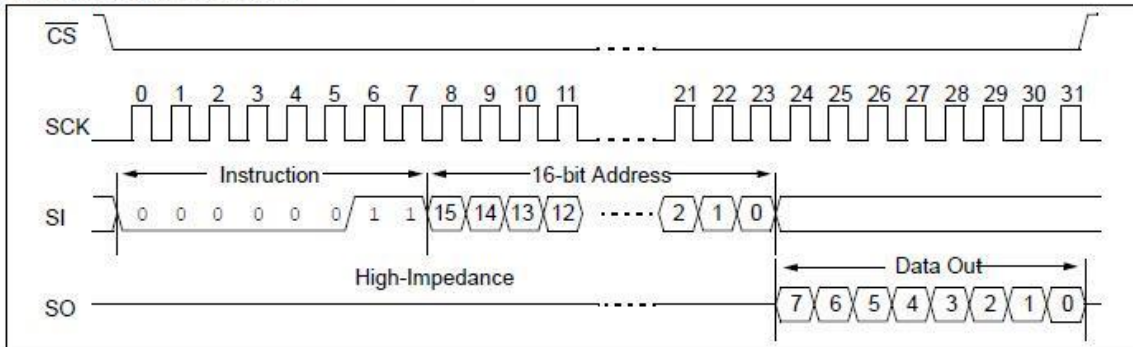
**Figure 1.5 23K256 Instruction Protocol**

Single-data byte memory transfers occur over 32 SCK cycles. This packet is divided into 4 bytes. The first byte of the sequence is the instruction byte, the three least significant bits containing the instruction set. **The first five bits of this instruction byte must be zeros or the 23K256 will ignore the packet.**

The second and third bytes in the packet contain the memory address to be written to or read. Because there is only 32KB of storage on the 23K256, only bits 14 through 0 are used. Bit 15, the most significant bit (MSB), is a “don’t care.”

The last byte of the packet contains the data transfer. If the write instruction, 010, was given in the instruction byte, the master, in our case the Amani, transfers data to memory via SI. Read data, instruction 011, is transferred to the Amani via SO.

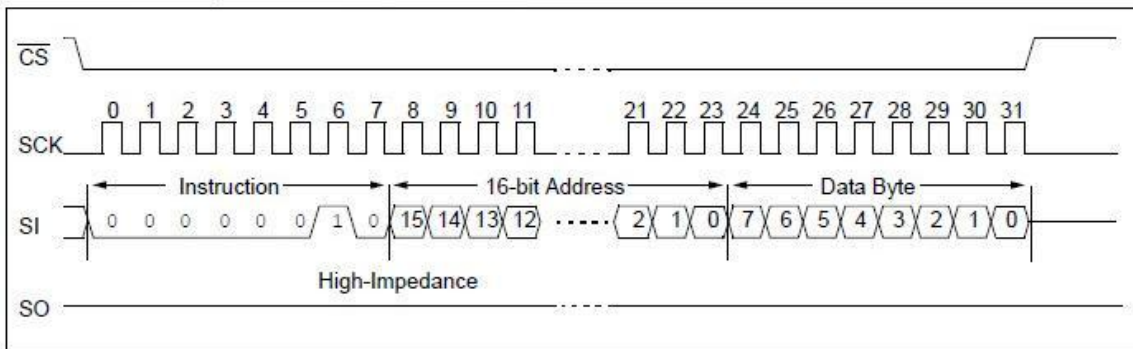
### BYTE READ SEQUENCE



**Figure 1.6 23K256 Read-Byte Transfer Protocol**

As can be seen here the write and read sequences are similar in length and structure excepting that the 23K256 releases its SO high-impedance state during the last 8 SCK cycles during the read sequence, shifting memory data to the master. During this time SI data is "don't care."

### BYTE WRITE SEQUENCE



**Figure 1.7 23K256 Write-Byte Transfer Protocol**

The key to both operations is that  $nCS$  goes low prior to the instruction byte and remains low throughout the entirety of the packet. Should the  $nCS$  signal go high at any time during the message, the 23K256 will disregard SCK and SI data. It instead will wait for  $nCS$  to go low once again as well as look for five '0's to be shifted in via SI.

Data transfer is dependent on SCK. Pauses of varying lengths between SCK ticks will not nullify the message provided  $nCS$  stays low.

## Arduino Packet Formation and Address Injection

We will first consider a single SRAM device transfer. While the 23K256 data packet is 32bits in length, the Arduino SPI interface performs serial transfers 8-bits per SPI data-register load. Because data is clocked into the 23K256 via SCK, which is ultimately generated by the Arduino, pausing the data stream to reload the SPI data register is acceptable. Thus the Arduino can successfully transfer data to and from the SRAM device by calling the **spi\_transfer** function four times.

Four successive SPI calls will contain one of the four sequential stages inherent to the transfer protocol discussed in the previous section:

Byte 1: Instruction Byte. 00000010 Write, 00000011 Read  
Byte 2: Memory Address Upper Byte, MSB = "don't care"  
Byte 3: Memory Address Lower Byte  
Byte 4: Data to be transferred

The **spi\_transfer** routine does not perform nHOLD or nCS signal manipulations which is convenient for our purposes. We can package the four transfer bytes by resetting and setting the nCS prior to and after the four byte transfers.

Below is a simple memory-write routine that makes use of this convention:

```
void write_sram(int addr_16, int addr_8, unsigned int data)
{
    char spi_chaff;
    digitalWrite(nCS, LOW);
    spi_chaff = spi_transfer(B00000010);
    spi_chaff = spi_transfer(addr_16);
    spi_chaff = spi_transfer(addr_8);
    spi_chaff = spi_transfer(data);
    digitalWrite(nCS, HIGH);
}
```

Three variables are passed to the **write\_sram** routine:

**addr\_16:** Memory Address Upper Byte, MSB = "don't care"  
**addr\_8:** Memory Address Lower Byte  
**data:** Data to be transferred

One variable is defined within the routine, **spi\_chaff**, the recipient of SO data from the **spi\_transfer** routine. Because this is a write operation, spi\_chaff data presented by SO is a nonsensical byproduct. **spi\_chaff** is only used during the last byte of a memory-read operation.

The nCS signal is written low to tell the 23K256 device that a message is about to begin. The line is returned high only after the entire message has been received. Premature setting of nCS will disrupt the message and the 23K256 resets its receive mechanism and will ignore further actions until nCS goes low again. This will prove useful as we will see later.

Read operations are performed by the **read\_sram** routine. This functions similarly to the **write\_sram** routine excepting that it returns an unsigned integer value that is the data read from the specified memory address:

```

unsigned int read_sram(int addr_16, int addr_8)
{
    unsigned int spi_chaff;
    digitalWrite(ardnCS, LOW);
    spi_chaff = spi_transfer(B00000011);
    spi_chaff = spi_transfer(addr_16);
    spi_chaff = spi_transfer(addr_8);
    spi_chaff = spi_transfer(B00000000);
    digitalWrite(ardnCS, HIGH);
    return spi_chaff;
}

```

The final byte transferred to the 23K256 device via the SI is inconsequential, as this is a memory-read function. In this routine spi\_chaff is utilized in the last SPI transfer to return the unsigned integer value to the calling agent.

The previously described routines are adequate for handling memory transfers to one 23K256 SRAM device. **Let us now explore a situation where the user needs more memory than 32KB without adding time to the transfer process. Furthermore the design is not to consume more Arduino I/O than the four specified pins of the SPI Interface.** Reviewing Digikey stocks we find very few SPI-capable memory devices larger than 64KB. A multi-23K256/CPLD combo is the solution to our problem.

Assume that the following are our design-specification requirements:

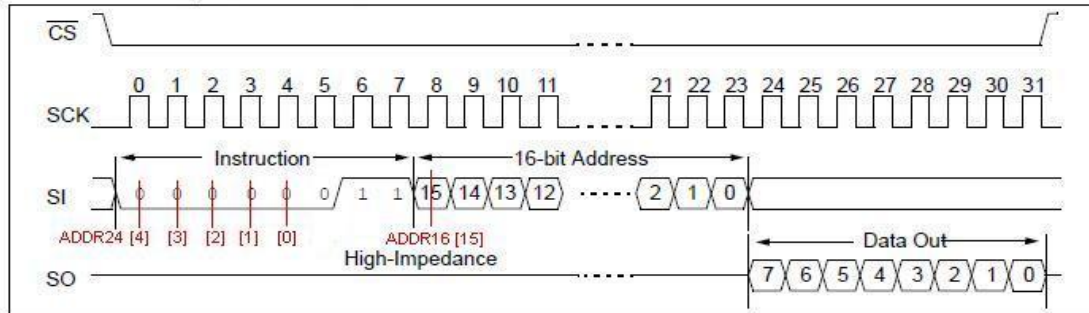
1. The Arduino will not expend additional I/O lines to accommodate additional 23K256 devices.
2. The Arduino will not extend the four-byte message to allow for additional addressing.
3. The translation-device must receive the additional addressing information from the Arduino in the existing four-byte packet.
4. The Arduino will not pause for external translational processing of the additional addressing.
5. The Arduino is no longer responsible for validity of the instruction-byte delivered to the 23K256.
6. The 23K256 devices all are governed by the protocols specified in their datasheet.

As should become practice when involving the Amani 64 in a design, the bulk of added complications should be incumbent upon the Amani to resolve. The sequential nature of microcontrollers makes their processing time valuable. The highly-configurable nature of CPLD's with respect to combinatorial and sequential logic makes them the perfect relief workhorse.

To accommodate more 23K256 devices, additional addressing is needed. Because the design is limited to the original four message bytes, the additional addressing must be injected into the existing packet. The **only bits available for manipulation** are the **upper five bits of the instruction-byte** and the **MSB of the upper-address byte**. This leaves us 6 bits in total for the extra addressing. This provides a theoretical total of 64 SRAM devices; over 2MB of storage.



## BYTE READ SEQUENCE



**Figure 1.8 Injecting Additional Addressing into the Existing 23K256 Message**

In terms of the Arduino Wiring code examples provided previously, we have now added the need for an additional addressing byte called `addr_24`. Furthermore `addr_16` has full use of its bit range as bit-15 will be used to select between the two 23K256 devices at the tips of each addressing branch. It is important to note here that the 23K256 expects to see "00000" as the first five bits of the instruction byte otherwise the packet will be rejected. We will leave it to the Amani to correct this as the packet is passed.

The modified Wiring routine accommodates the new addressing byte as seen below. The user passes the routine via `addr_24` without having to manually move the address to the upper 5 bits of the instruction byte. The routine shifts the address to the left by three positions and appends the read/write command to the lower three. For a write operation, 010 (2) is written. `nCS` operations remain unchanged and are left for the Amani to address.

```
void write_sram(int addr_24, int addr_16, int addr_8, unsigned int data)
{
  char spi_chaffe;
  addr_24 = ((addr_24 << 3) + 2);
  digitalWrite(nCS, LOW);
  spi_chaffe = spi_transfer(addr_24);
  spi_chaffe = spi_transfer(addr_16);
  spi_chaffe = spi_transfer(addr_8);
  spi_chaffe = spi_transfer(data);
  digitalWrite(nCS, HIGH);
}
```

For a read operation, the address is shifted to the left thrice and appended with 011 (3) for read:

```
unsigned int read_sram(int addr_24, int addr_16, int addr_8)
{
  unsigned int spi_chaffe;
  addr_24 = ((addr_24 << 3) + 3);
  digitalWrite(nCS, LOW);
  spi_chaffe = spi_transfer(addr_24);
  spi_chaffe = spi_transfer(addr_16);
  spi_chaffe = spi_transfer(addr_8);
  spi_chaffe = spi_transfer(B00011000);
  digitalWrite(nCS, HIGH);
  return spi_chaffe;
}
```

That's the extent of the work the Arduino has to perform in order to incorporate additional 23K256 devices. The task is now up to the Amani to extract addressing from and correct the packets as they are passed in real-time to the targeted 23K256.

## Amani Address Extraction, Device Selection, and Packet Correction

In order to remain transparent to the Arduino and 23K256 devices the SCK output of the Arduino is passed untouched through the Amani CPLD to each of the SRAM devices. Moreover each SI output of the 23K256's is passed directly through the CPLD to the Arduino MISO pin. The Amani must correct the instruction byte as it is passed to the SRAM while extracting addressing data and selecting the 23K256 that is being targeted via the corresponding nCS signal.

The nCS signal is the first order of business. Because there are multiple 23K256 devices, the Amani must handle unique nCS lines for each device. The rest of the SRAM signals share common nodes. The intended SRAM's nCS line is set by the Amani via addressing passed from the Arduino.

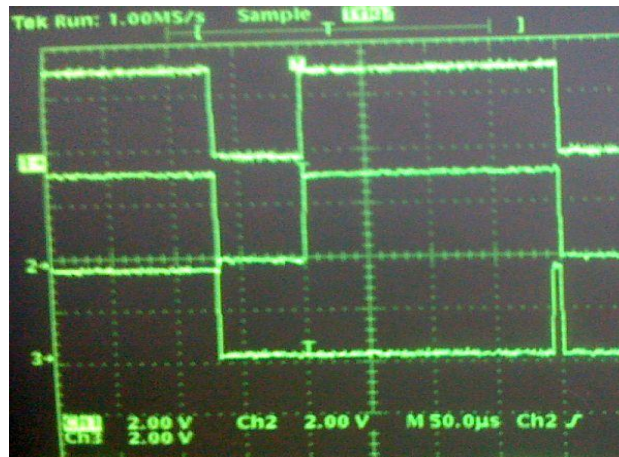
ADDR24	ADDR16	ADDR8	nCS[X]
00000000	0XXXXXXXX	XXXXXXXX	nCS0
00000000	1XXXXXXXX	XXXXXXXX	nCS1
00000001	0XXXXXXXX	XXXXXXXX	nCS2
00000001	1XXXXXXXX	XXXXXXXX	nCS3
00000010	0XXXXXXXX	XXXXXXXX	nCS4
00000010	1XXXXXXXX	XXXXXXXX	nCS5
00000011	0XXXXXXXX	XXXXXXXX	nCS6
00000011	1XXXXXXXX	XXXXXXXX	nCS7

Device Selection
Internal Addressing

**Figure 1.9 SRAM Addressing, Bank Selected by nCS[X]**

The Amani is unaware of which SRAM device is being targeted as the SPI transfer begins. **The Amani buys itself time while extracting the address by enabling every SRAM device at the start of the packet.** As soon as the Arduino sets nCS, the Amani resets all nCS signals low. The 23K256's do not cause conflict on their sole output pin, SO, as SO is always high-impedance until the last byte of a read-operation. This allows the Amani to enable all SRAMs to listen to the data packet until the Amani can determine the exact device being addressed. Once the address is loaded, the Amani disables all SRAMs mid-packet, except for the addressed device, which causes the remaining 23K256's to stand down. Disrupted SRAM's will neither store data being presented via SI nor attempt to present data to SO.

The graphic below shows the nCS lines of three SRAM devices during the first of two subsequent message transfers. The top two traces demonstrate the interrupting of the SRAM nCS signals mid-packet, while the third trace, belonging to the nCS signal of the targeted device, remains low throughout the entire packet. All three traces go low once again as they begin their second message.



**Figure 1.10 nCS Signals, Two Interrupted vs. Targeted Device**

A finite state machine (FSM) implemented in the Amani extracts and stores the addressing bits, the 5 addr\_24 bits as well as the MSB of the addr\_16 byte, as they are received. The Amani then interrupts the non-targeted devices on the subsequent SCK cycle. This occurs fifteen SCK cycles before the data byte.

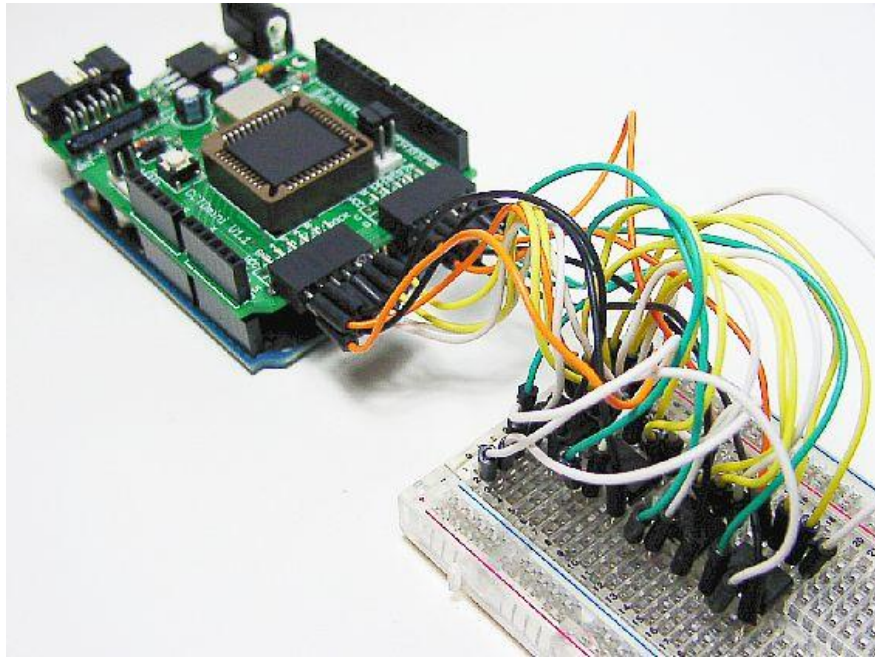
In the same FSM the Amani auto-corrects the instruction-byte by simply holding the SI line low for the first 5 SCK cycles then releasing it to be controlled by the Arduino MOSI.

Upon setting and releasing the appropriate nCS lines, the Amani goes into passive mode and allows the Arduino to drive the rest of the transfer. The Amani FSM resets when the nCS line from the Arduino goes high. It restarts upon the Arduino nCS line going low again, signifying a new message.

### **Summary**

For the sake of cost and efficiency, this project only addresses 8 23K256 devices. The Amani 64 has enough available I/O leftover to address 12 additional devices, provided it can power them. The theoretical addressing limit, being six digits, can address a total of 64 SPI SRAM devices, enough for 2MB of storage. This approach is not practical due to real estate, power, and price issues as well as the availability of other technologies that would utilize less I/O and offer faster transfer times.

The main purpose of this project was to demonstrate a simple and cheap way to expand memory with extra 23K256 devices. Moreover it serves as yet another example of how versatile CPLD's can be as glue logic between microcontrollers and systems that may need assistance fitting together.



**Figure 1.11 Four Wired 23K256 SRAM Devices; Amani Stacked on Arduino**