Designing With Discrete SPI Flash Memory

by Catoblepas

Designing with discrete flash is **1/10th the cost**, uses a much **smaller form factor**, and requires significantly **less specialized hardware** than using SD flash cards.

This Instructable will show you how to add 1MB of discrete external flash memory to your microcontroller project with what I believe to be the least amount of effort possible. This is also a follow-on to my other two data-logging Instructables (an anemometer and a 3-axis wrist accelerometer) that explains how to download the data from the logger flash memory using age-old TTY command line applications found in Linux.

Motivation

Whenever I'm building an Atmel ATMega or Arduino project and I need to record data, I almost always reach for a single SPI WinBond W25Q80BV 1MB flash chip rather than an SD flash subsystem. Many reasons exist to choose a discrete flash chip over an SD subsystem, and vice versa, and you'll need to consider these tradeoffs for your design. The list below contains a few tradeoffs I think about when I need to decide if I want to use a single 8-pin DIP chip or a full-on SD solution:

Hardware Complexity (Choose: Discrete)

One way to add SD flash to an Arduino system is to use a shield, such as this one by Seeed Studio (three 'e's) I bought at my local Radio Shack for $15. While shields provide convenience for prototyping, the final production assembly might not have the budget or the space to include SD hardware. An 8-pin DIP package of a discrete flash chip is much easier to drop on a protoboard than an SD shield, assuming your development board even supports a shield.

Software Complexity (Choose: Discrete)

The SD flash subsystem commonly relies on the SDF at16/32 libraries. While the devices are an SPI interface, it makes sense to use FAT since any PC/MAC can then read this card. These libraries are large and can take up precious EEPROM space on smaller embedded controllers. Compatibility and integration into your build environment may require significant debug. The software required to drive a discrete flash chip with an SPI interface is trivial and very small, as you will soon see. Maybe this says more about me than the SDFat libraries, but I find them cumbersome to work with.

Capacity & Portability (Choose: SD)

SD flash wins big here, simply pop in a larger capacity SD card into the existing design with no modifications. Discrete SPI flash has lower density limits in the 8-pin DIP format. The SDFat library means any PC/MAC can read the files on the card.

Cost (Choose: Discrete)

SD cards range in price dramatically, and with an SD flash shield, can set you back $20-$30. WinBond 1MB chips cost about $2 from Mouser or Digikey.

Power (Choose: Discrete)

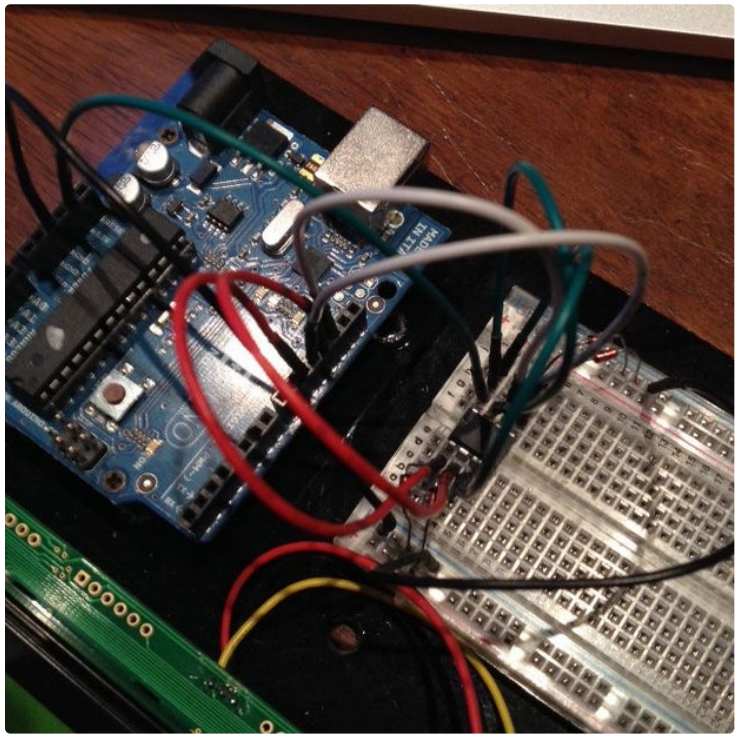
Energy requirements of flash depend on the manufacturer, production lot, device density, and process technology. SD cards are typically higher

leakage power due to the higher densities, and higher dynamic power due to the higher access speeds. The WinBond chips I focus on in this Instructable require very little power, 6uW standby, 60mW page program, and 60mW chip erase. I wasn't able to find power data on the high-end super-fast SD cards, but the write speed is about 100x that of the WinBond. Since dynamic power is proportional to frequency, I can't imagine power would be *less*.

Speed (Choose: SD)

I haven't had any need for very fast flash memory write performance, but SD flash comes in many different product SKUs based on speed (mostly due to the demands of digital photography and the use of raw image formats). The WinBond SPI chips can't really compare: page program speed is 0.7ms for 256 bytes, which translates to 0.360MB/s, which is 100x slower than Team Corp.'s fastest Micro SD cards at ~40MB/s. I suspect they have multiple devices or arrays writing in parallel to achieve those speeds.

While this analysis most likely represents my own lazy biases, I find my brand of laziness to be rather prolific. That being said, any one of these vectors may be more important for your project, but my goal here is to call out the tradeoffs, and then illustrate the simplicity of this wonderful flash chip. (And I haven't even discussed using larger capacity parallel flash chips.)



Step 1: What Is SPI Flash Memory?

I'm going to explain this next part painfully fast. My first job at Intel was in the flash memory group in 1993, and a lot has changed with the technology in the 20 years since then, but some concepts remain consistent.

Flash memory is a type of nonvolatile storage memory based on MOSFET technology. Nonvolatile means the device retains its value when it isn't powered-up.

MOSFET

If you aren't familiar with how a MOSFET transistor works, I'll try to explain it in one sentence: a slab of silicon with two terminals on either end doesn't conduct electricity if you place a potential difference between them, but if you stick another piece of metal on top of that slab and sandwich a dielectric between it, and then apply a voltage to that piece of metal it creates a field and current can flow between the two terminals. The terminals are called the source and drain, and the metal is called the gate. That's a super simple explanation that bulldozes 50 years of quantum physics, but from a Michael Farady point-of-view, it is reasonably workable.

FLASH TRANSISTOR

Flash memory operates by blasting a bunch of charge carriers onto the dielectric between the gate and the substrate. This is called *programming*, and is typically done with a much higher voltage. It actually damages the material, and after 100k program cycles, the gate will fail. To remove the charge carriers rom the dielectric, and equally high voltage, but reverse potential, pulls the carriers off the gate. This is called *erasing*.

A programmed flash bit has value 0 and an erased bit has value 1, an erased flash byte is 0xFF in hex. (Nowadays, flash memory can store multiple bits per cell using multiple voltage levels, but that gets really complicated.)

FLASH ARCHITECTURE

Typically, a flash memory contains a giant array of transistors that can be individually programmed, but only erased in groups (sectors, blocks, or the entire chip). This is simply a side effect of how the erase circuitry works: per-bit erase would require too much metal density, and isn't all that useful (in practice, erasing in larger chunks works just fine).

Since programming a single transistor is slow due to ramping up that high voltage and all of the control that goes along with that, flash is usually programmed in *pages*. Typically a flash device will have a small SRAM page buffer (256 bits) which the host will first rapidly fill with data, and then the host issues a page write command, and the flash chip writes all the page bytes out in a large batch job. This batch circuitry amortizes the startup write latency across a larger number of bits. Offering two or more page buffers allows the host to use a double-buffer technique to hide the write latency of the flash device.

SPI

The Serial Peripheral Interface is a brilliant invention. It is a simple serial interface that uses a chip select, a clock, a data IN and a data OUT. There are many kinds of SPI devices, as it is a very popular interface, and all SPI devices use a common library: once you know how to talk to one SPI device, you can talk to any SPI device.

The advantage to SPI is it's software simplicity, the code basically shifts data in and out of the DI and DO pins respectively, on the rising edge of a clock. And the clock is controlled by the host, it doesn't require a fancy clock circuit: the phases can be as asymmetric as you want, as long as you adhere to the minimum cycle width requirements of the device.

FLASH SPI

Flash SPI memory simply combines the best of both worlds. Note that SD cards use SPI as well as this discrete chip. Surprise! The programming interface isn't very different, but the actual instructions and timings differ.

Step 2: The WinBond Device Interface

The pinout shown above is taken from the WinBond datasheet.

Pin 1: Chip Select (/CS, sometimes called /SS, for "serial select")

CS is the "Chip Select" pin. You set the CS pin when you want to talk to that device, because you could have a dozen SPI devices all sharing the same bus, and you identify each one uniquely via their CS pin. The slash in front of CS means "active low": to talk to this device, pull this pin to logic level zero; to remove it from the shared bus, drive logic level one.

Pin 2: Data Out (DO)

Serial data is read from this pin. It will connect to the MISO (Master In / Slave Out) wire of the bus. Typically you write a command to the SPI device in a pre-determined sequence. After that sequence completes, and depending on the instruction in the sequence, data is then read off the DO pin.

Pin 3: Write Protect (/WP)

This pin disables writing. Sometimes you'll see a jumper attached to this pin in order to provide very strict control over the program/erase mechanism: if set low, the device cannot be programmed or erased. I usually hardwire it to Vdd and allow my software to control write enable/disable through serial commands (we'll talk about this later).

EDIT (2016-12-16) Thanks to user velsoft for catching a typo: I had the polarity mixed up.

Pin 4: Ground

This is simply the ground pin.

Pin 5: Data In (DI)

This is the input serial pin. It will connect to the MOSI (Master Out / Slave In) wire of the bus. Commands and data are written to this pin by the host system.

Pin 6: Clock (CLK)

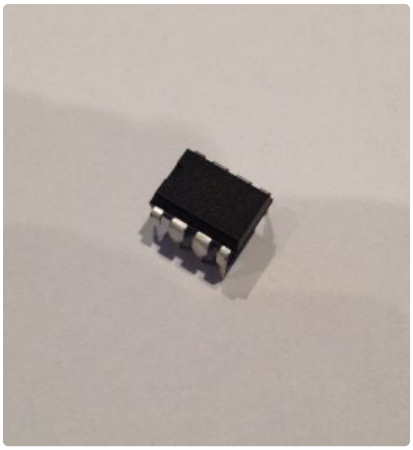
The clock pin determines how data bits are transmitted on the DI and DO pins. The DI/DO pins are sampled on the rise of the clock pin.

Pin 7: Hold (/HD)

I've never used this pin, but it allows a host device to pause whatever transaction is in flight. You'll probably never have to use this pin so I leave it wired to VCC (active low).

Pin 8: VCC

This is simply the source voltage.



5. PIN CONFIGURATION PDIP 300-MIL

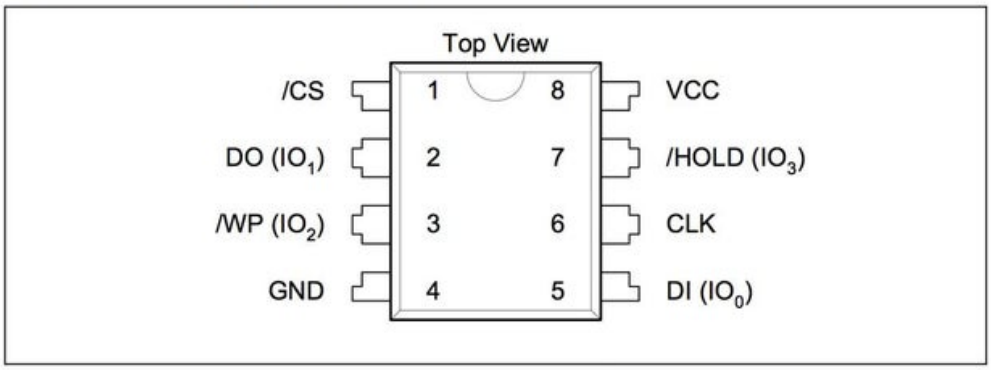


Figure 1c. W25Q80BV Pin Assignments, 8-pin PDIP 300-mil (Package Code DA)

Step 3: How to Read a Timing Diagram

Now that I've explained flash, SPI, and a specific implementation of an SPI flash device, the next things you need to understand are communication timing diagrams*. Timing diagrams explain the sequencing of the data across the pins to issue instructions to the device. Each SPI device responds to its own set of instructions (e.g., a flash device will have a read or erase instruction) and the timing diagram is the link between the conceptual behavior of the instruction and the actual hardware protocol to execute that instruction.

In the diagram for this section I copied the chip erase timing diagram from the datasheet because it is the easiest to understand.

The bottom axis is time, the vertical axes represent four SPI pins and the sequence data should appear on them over time to execute an instruction. Note: "High impedance" means you can ignore that signal (it is driven to not 0 or 1, but extremely high resistance, so it is effectively an open circuit). Cases when two lines appear (like DI) that simple represents that some kind of transitions are happening but are unknown; a single line means a specific high or low value is present.

Let's look at the diagram from left to right and top to bottom.

In order to talk to any SPI device, it's chip-select must be brought high and then driven low (remember /CS means active low). When /CS is brought low, note that the clock in the diagram is very explicitly drawn to show eight phases. This means you must pulse the clock eight times, once per bit. At the time the clock is strobing, data in goes from high to low to high. I think the DI diagram is erroneous, because if you draw a vertical line down the rising edge of each clock and calculate the binary values of DI at those points, you should get value 11000111, or 0xC7. This is the instruction that tells the chip to erase itself. Once chip select is brought high, the internal circuitry will begin executing the 0xC7/Chip Erase function. This instruction takes about 1~2 seconds to complete.

Keep in mind, you don't need to actually toggle the clock pin 8 times to send out 8-bits of a byte, the SPI library does this for you when you use the function SPI.transfer(). You will still need to manually drive /CS with digitalWrite(), but the SCK, MOSI and MISO is all handled by the SPI functions.

You will notice in my source code a function called "not_busy()". This function continually issues a "read control register #1" and checks bit 0 which indicates if the internal operation has completed yet, and the flash is not busy. The timing of this operation matches diagram 9.2.8 of the datasheet.

* Note I am not referring to the electrical timing diagrams, which explain to the nanosecond the setup and hold times for the internal digital logic; the diagrams I'm referring to are the logical diagrams that ignore nanoseconds and describe the *sequence of logical events*. The actual electrical timing of the SPI interface is handled by the Arduino SPI library. And to be honest, that code isn't very complex, and could be further simplified if you are designing to one specific device.

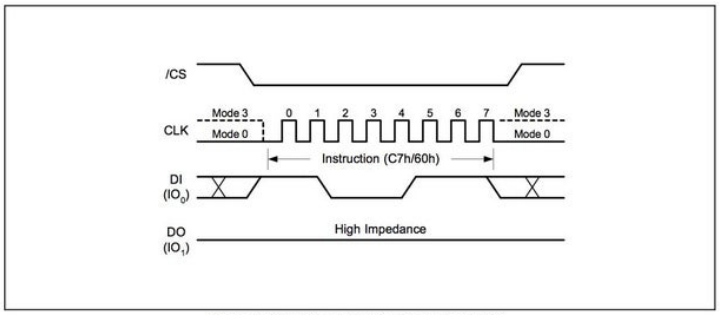


Figure 24. Chip Erase Instruction Sequence Diagram

Step 4: Interfacing to an Arduino Uno With Level-Shifters

The Arduino Uno's digital outputs transmit 0V and 5V as logic levels low and high, respectively. The WinBond flash chip only operates between 2.7V and 3.6V. Whenever logic circuits on different voltage planes need to communicate, we have to use a *level-shifter*.

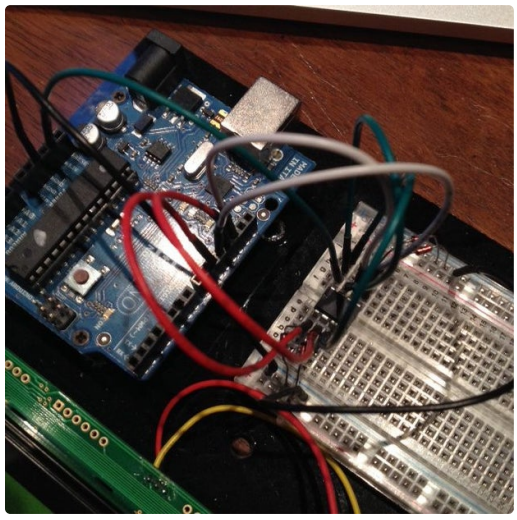
The easiest form of level-shifter is a simple Zener diode clamp. There are many other types of level shifters in the world, some are faster, some use less power, the Zener clamp method is quick and easy.

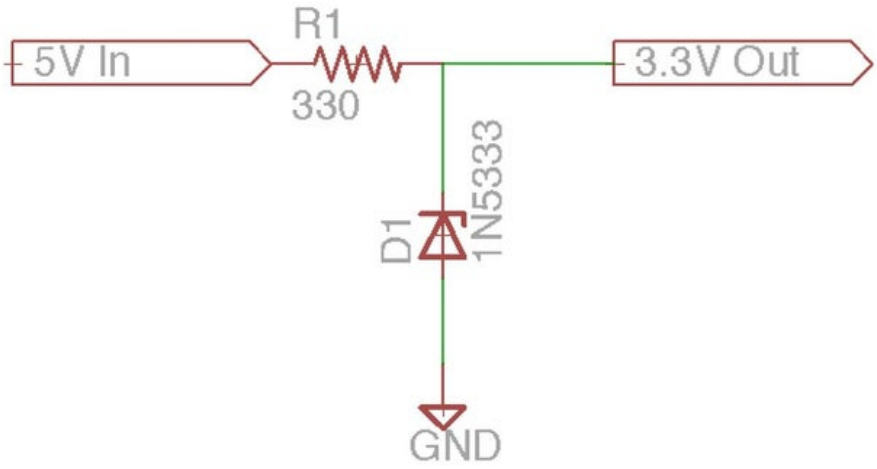
All diodes have a reverse breakdown voltage at which point they begin to conduct. Zener diodes are specifically designed to breakdown at finely tuned voltages. In my case, I connected a 3.3V Zener diode in parallel with each of the chip's digital inputs (see the schematic). (As for the other four pins, ground is 0V, and the Uno board has a 3.3V supply for VCC, so these pins don't need a diode, and I hardwired /WP and /HOLD to 3.3V Vcc.)

UPDATE: I forgot to add the 330 Ohm resistors in series with the output of the Uno drivers. Normally, if you were connecting the digital output of the Uno to a digital input of another device, a simple wire would suffice (since you are connecting one digital logic signal to another, see the ATmega328 datasheet, section 13.1 "I/O Pin Equivalent Schematic"). But since the output path now branches through the Zener, you need a resistor to limit the maximum current driven by the logic output of the Uno/ATmega chip. Without the resistor, this path to ground may exceed the max output current of the device. Which would be bad, Ray.

Now, whenever the Uno drives a 5V logic-high into, say, the /CS pin, the Zener diode switches to breakdown mode, clamping the voltage to 3.3V, thus protecting the input logic of the flash chip.

Using these clamps, I connected the Arduino Uno's digital output pin 10 (SS) to /CS, pin 11 (MOSI) to DI, pin 12 (MISO) to DO, and pin 13 (SCK) to CLK. (Note that the pins of the Atmega328 are NOT the same pins as the Uno, e.g., the Atmega pin #19 is Uno pin #13.) The SPI software library assumes pin 10 = SS, etc.





Step 5: Code Code Code!

I wrote a sketch that allows me to communicate with the Uno via serial TTY communication via the Serial Monitor (or even a Unix prompt, as you well see). This is a helpful method for debugging new hardware, as I can issue commands interactively.

The "serialEvent()" function is a built-in callback, called whenever something happens on the default Serial object. I use this callback to construct a command string and set a boolean flag (the byte-by-byte construction of the string completes when the callback reads as semicolon ";" from the stream; I use this instead of a newline since there's no way to issue a newline from the serial monitor). When the callback constructs the string and sets the flag, the "loop()" function executes a decoder. The decoder determines which function to call based on the command string, and parses any additional parameters from the command string, and calls that function.

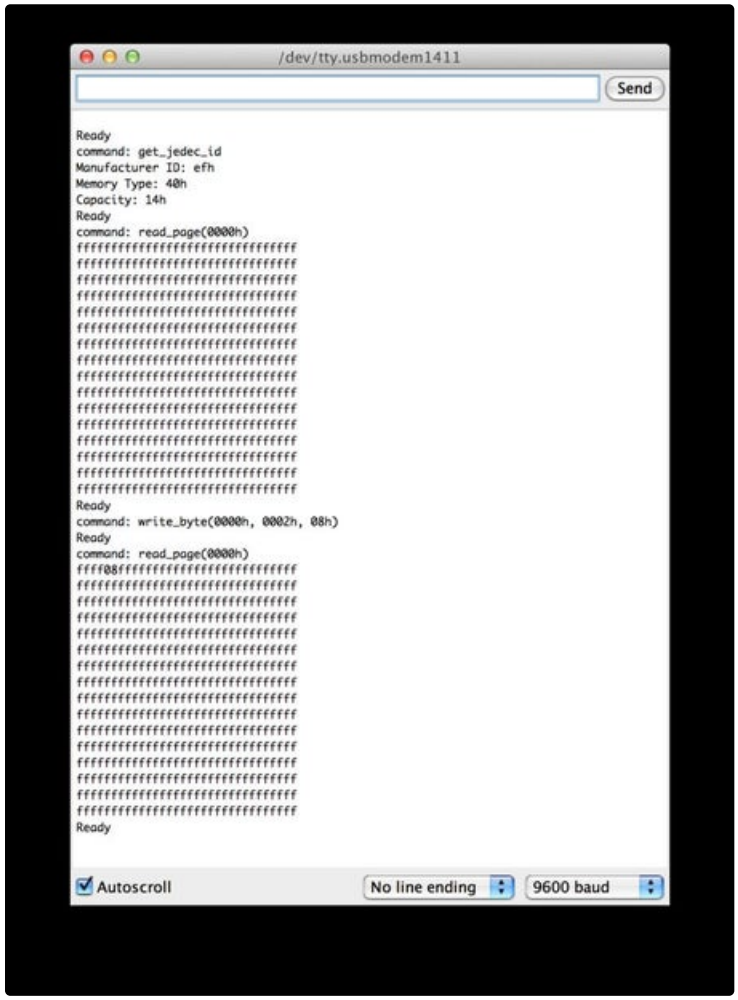
Each function is essentially a wrapper around a low-level implementation of a WinBond SPI functional timing diagram. I used a wrapper so that the low-level functions remain generic: I can use them again in other sketches with a simple cut-and-paste. Plus, the wrapper prints some feedback to the user, which is very useful for debug.

The screenshot above shows an interactive session with the Serial Monitor. I have issued four commands, "get_jedec_id;", "read_page 0;", "write_byte 0 2 8;", and "read_page 0;" You don't actually see the commands (the serial monitor doesn't have an echo, and I didn't print the exact command.. I probably should have), but you do see the response. It should be most clear when I read/write/read page 0. The "read_page ;" command simply dumps the specified page (in decimal). The "write_byte ;" function is a little weird, as the parameters specify a page number, an offset into that page, and then the byte. Since there is no native 32-bit register in the 16-bit Atmega, I didn't bother doing logical to physical translation, but you'll need to consider this translation at some point. Anyway, notice that the third byte of page zero is now "08h".

I could have also issued "chip_erase;" and then "read_page 0;" to illustrate an erase cycle, but hopefully you get the picture.

The low level functions start with "_", and are named "_read_page" or "_write_page" or "_erase_chip". These functions explicitly sequence out the SPI commands found in the datasheet timing diagrams. Each function ends with a call to "not_busy()" to prevent execution from proceeding before the chip has completed its internal operation.

EDIT (11-MAR-2014): There was an issue with the _read_page low-level function, I had forgotten to pull CS HIGH before pulling it LOW at the start of the function, like the other functions. This means if _read_page is the first function you call, CS may not already be high, so without a valid /CS 1->0 transition _read_page will not function properly, the first time it is called. The second time it would work fine because it leaves /CS as 1. Small but annoying bug.



https://www.instructabl...

Download

Step 6: Downloading the Data With a TTY

The real reason for this Instructable is demonstrate how to download the entire flash memory to a single file. To do this, I used a Unix function, "tail -f" and a redirect.

The Unix function "tail" prints the last 10 lines of a text file. When given the parameter "-f", "tail" will remain connected to the redirect until it catches a SIGINT (e.g., Ctrl-C).

There are three windows open in this screenshot: the Arduino IDE on the left, the Serial Monitor on the upper-right, and an OSX POSIX terminal in the lower-right. In OSX/POSIX land, the USB controller of the Uno shows up as a /dev/tty device, in this case "/dev/tty.usbmodem1411". I connect "tail -f" to this device and redirect the output to a file.

I then issued a "read_page 0;" command in the serial monitor, and the output is sent through "tail" since it is connected to the output of the TTY, and then sent to the file. I then "cat" the file to prove the serial stream was captured.

Now all I need to do to dump the ENTIRE flash chip is to type this in the terminal prompt:

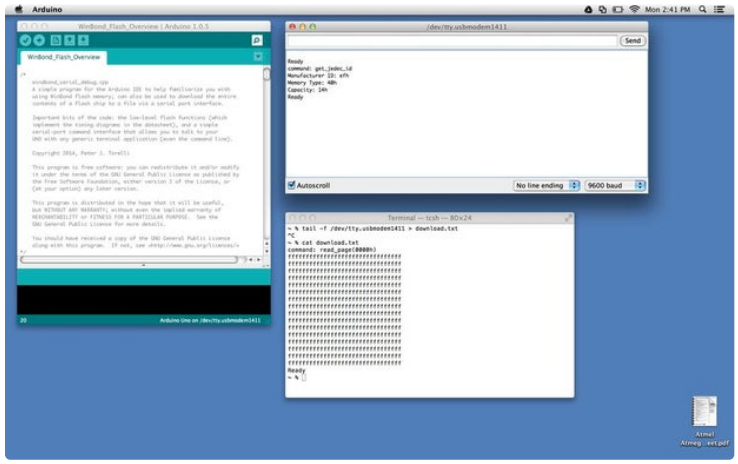
% tail -f /dev/tty.usbmodem1411 > 1MB_of_flash.txt

And then type this in the Serial Monitor window:

read_all_pages;

Then type CTRL-C in the terminal window to stop the "tail" process.

Done and done! *This is why Unix is so vastly superior to any other operating system, IMHO*.



Step 7: Conclusion

This concludes the long-awaited sequel to my data logger Instructables. I had promised a forthcoming method for pulling the data out of the flash chip, here it is.

I hope you found this Instructable useful: 99% of what I know was learned from reading things like this on the web that other people took the time to write, and I'm very grateful for their efforts.

I ported the project to Azure Sphere. Using a AMIC Technology A25L080-F, SPI 8Mbit Flash Memory, 8ns, 8-Pin PDIP

Hi,i cant get the commands to work,i get iinvalid command sent.

is this right "erase_chip;" it doesnt work invalid command sent i get back.None of the commands work.

I had the same problem. I have to reset the Nano after each command. It does work if I do that. Just for others to know since its been 10 months since your post and the author doesn't want to be bothered with questions.

I saw the problem, but I don't have time to debug everyone's software on a project I gave to the community for free four years ago. I did initially help people, if you see my replies the first couple of years. Maybe you could debug it and post the answer? Or you were you just satisfied with not understanding the issue to any significant depth?

I moved on too. Found a nice library with a well written code.
https://github.com/Marzogh/SPIMemory
Also, its not "everyone's software project", its your own code that doesn't run properly.

The code works, but my nano has to be reset after each command.
Otherwise it doesn't recognize the next command.

That sketch was awesome! Thanks for doing this. There is actually very little out there for Arduino SPI flash programmers.

On speed... I would not say that SD cards beat embedded flash for speed. At least, if you're using SPI to talk to it, you're not going to get good results at all. The SDIO protocol is the only one for which the advertised speeds of 40 or 100 MB/s is actually guaranteed. Over SPI, the controller behaves differently and those speeds are out the window. In between sector writes, if the SD card's controller decides to flash a block to make room for the next sector, you will be stuck in a waiting mode for usually at least 10 ms, and I've seen it take 130 ms. If that's not a problem, then great--it does ok on average. But if you are logging data in real time and don't have the RAM to buffer it all, watch out.

I think you made your point backwards, because you say SD doesn't beat a discrete embedded device, but then you offer points that support SD! (e.g., SDIO vs. SPI and FFS impact).

Block cycling typically happens behind the scenes and the intent of the flash file system is to hide the impact. While it can happen during a read or write, it is purely a function of the flash file system policy and not the memory itself: it is all the same flash!

Also, while SPI maximum speed is only limited by the MCU (compare Silicon Labs, PIC, STMicro and TI, they all offer different SPI speeds).
An SD card, using the SDIO protocol will access in the devices in parallel, not serial/SPI, which is why you will notice the write speed as a multiple of the flash device, and may be up to 64 bits wide depending on the number of parallel-accessible bits available on the die/package and chips in the card. This parallelism can be done with discrete flash chips, but once you start going down that path, SDIO solves a lot of problems.

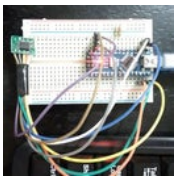
Thanks for the feedback.

Hello(Excuse me for my english), I dont know nothing about these things but I need to know If i can reset the BIOS password in this chip 25x40CLNIG 1311

Hi ptorelli, I want to know if is possible read and write a SPI EEPROM Winbond W25Q64FVSIG because I need to show hidden pages of my BIOS in my laptop. Tell me how please. Thanks.

Thanks mate for this.

I was able to read a different chip (EN25QH16). I double checked the instruction set and they're alright. The serial monitor commands works just fine but once I entered a second command, the returns invalid command. The first entered command works (any commands).



Hi to all, does any one can make a code for the tensy with schematics? so the tensy can be used has a spi programmer? regards

I think you are asking if it works on the Teensy? Shouldn't be a problem, SPI is well-defined protocol, and the sketch should compile for a Teensy if the IDE recognizes it. Let me know if it works. Cheers.

Just a note about the speed: in this case it is limited by the SPI speed, which is limited by the microcontroller clock, much lower than the 80MHz speed of common Serial Flash memory ICs. When using a SD card with an Arduino (taking advantage of the SPI port) you are limited to the same speed, so no improvement.

I benchmarked the sequential page read/write speed with the following modified code (faster) and it takes about 2us for each byte:

```
/*******************************/

void _read_page(word page_number, byte *page_buffer) {

digitalWrite(SS, HIGH);

digitalWrite(SS, LOW);

SPI.transfer(WB_READ_DATA);

// Construct the 24-bit address from the 16-bit page

// number and 0x00, since we will read PAGE_LENGTH bytes (one

// page).

SPI.transfer((page_number >> 8) & 0xFF);

SPI.transfer((page_number >> 0) & 0xFF);

SPI.transfer(0);

unsigned long mil = 0;

unsigned long mil2 = 0;

int i = 0;

byte *page_buffer_prt = page_buffer;

mil = micros();

while ( i < PAGE_LENGTH) {

SPDR = 0x0; // Start the transmission
```

```
while (!(SPSR & (1<<SPIF))) ;
*page_buffer_prt++ = SPDR;
i++;
}
mil2 = micros();
Serial.print(F("Elapsed: "));
Serial.println(mil2-mil);
digitalWrite(SS, HIGH);
not_busy();
}
/*******************************/
void _write_page(word page_number, byte *page_buffer) {
digitalWrite(SS, HIGH);
digitalWrite(SS, LOW);
SPI.transfer(WB_WRITE_ENABLE);
digitalWrite(SS, HIGH);
digitalWrite(SS, LOW);
SPI.transfer(WB_PAGE_PROGRAM);
SPI.transfer((page_number >> 8) & 0xFF);
SPI.transfer((page_number >> 0) & 0xFF);
SPI.transfer(0);
unsigned long mil = 0;
unsigned long mil2 = 0;
int i = 0;
byte *page_buffer_prt = page_buffer;
mil = micros();
while ( i < PAGE_LENGTH) {
SPDR = *page_buffer_prt++; // Start the transmission
while (!(SPSR & (1<<SPIF))) ;
i++;
}
mil2 = micros();
Serial.print(F("Elapsed: "));
Serial.println(mil2-mil);
digitalWrite(SS, HIGH);
not_busy();
}
```

P.S:

```
#define PAGE_LENGTH 256
```

Hi dasankir-

You are absolutely correct about the MCU and I/O frequencies being the limiters. Thanks for the code contribution, too!

-Peter

Thanks for this excellent Tutorial! I really learned a lot!

Hi sorry to bother but can someone assist. this is the code of my chip but can seem to program. I get the following

HELLO

0 - Bytes

My chip details

Man ID : c2h

Mem Type : 20h

Capacity 14h

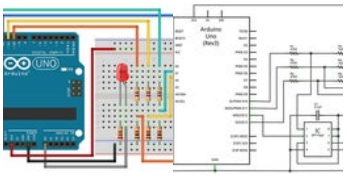
I got one , in a old low cost cellphone . is SMD version and the number is:
(25Q64BWIG) I dont know if can I use it and I cant found the datasheet anywhere but i found some similar (25q64fvsig)

Excellent instructable - Thanks very much!

In your discussion regarding the \CS pin, you said that many devices maybe connected to the SPi bus. Does each device connected to the SPi bus require a dedicated I/O line from the microprocessor? Seems like you'd quickly consume I/O pins.

This is a fantastic tutorial, i spent days searching for something like this last year! i did not have diodes handy so used the attached schematic (courtesy of adafruit - trinket audio player)

which works flawlessly with this tutorial if anyone else is in the same situation. Thanks again ptorelli for taking the time to post this detailed information.



Also : anyone using windows and wants to read all to a text file this is how i did it.

Install Cygwin.

determine the port used (unfortunately different from ptorelli's above) with the following command.

ls -lah /dev

this will list all the devices, mine for com3 was /dev/ttyS2 (found by running this command then running it again without the usb plugged in to determine which was missing)

Because windows does not allow you to share com ports with multiple programs you cannot do what ptorellis has done above. instead i edited the read_all_pages function to add a delay(5000) and called this function from the main loop.

In cygwin, type the following :

tee </dev/ttyS2 ouput.txt (where /dev/ttyS2 is what you have determined)

before pressing enter in cygwin, open the serial monitor to initiate the read_all_pages function then close serial monitor and press enter in cygwin.

text file will begin populating! (if you press enter while serial monitor is open you will get an access is denied in cygwin).

ptorelli if there is an easier way than this in windows please let me know and thanks again for your tutorial.

I hardwired the get_jedec_id, and read_all_pages to a hardware button, so when it is pressed, it will call the above functions. I then setup realterm to listen on my USB serial port and capture the data using realterm. a bi

a bit better than switching between cygwin and serial monitor.

Thanks for both the kind words and the contribution to the project! I have learned a lot more about serial communication since I wrote this instructable. I now use Node.js because not only does it provide simple serial API, but it dovetails into a web framework and is cross-platform. In my other instructable, the Arduinolyzer.js, I use the serial interface connected to a localhost webserver instead of using the console, and I've received word it works seamlessly on Windows thanks to node. Something else to consider!

as is the connection?

While this analysis most likely represents my own lazy biases, I find my brand of laziness to be rather prolific. That being said, any one of these vectors may be more important for your project, but my goal here is to call out the tradeoffs, and then illustrate the simplicity of this wonderful flash chip. (And I haven't even discussed using larger capacity parallel flash chips.)

This is fantastic tutorial! I think it's first time I actually understood how SPI communication works :) Is there any way to actually write data to flash chip from computer? i.e. upload small WAV files. I work a lot with ATMega and Sound (i.e. my Xronosclock.com project) and been dreaming of switching to SPI flash chips instead of cumbersome microSD cards used my clocks...

Glad to hear I could help!

The quickest method for this would look like:

computer -> usb -> (Uart) Atmega -> SPI -> Flash

You could potentially use any scripting language that supports a serial port interface (perl, node, python), then write buffers to the target device, and have the target device push them out the SPI port.

Check out my arduinolyzer instructable for an example of using node.js to control an arduino and read/write data via serial comm (nice thing scripting languages, if you keep it (relatively simple) it'll work on Linux, Windows, and Mac OSX)

Good luck!
P

w25q128fvssig reading speed ??

Great instructable!

In case you're interested/didn't know, W25Q 32Mbit(4MB) / 64Mbit(8MB) / 128Mbit (16MB)versions are available on ebay for 45c / 90c / $2.50each , in lots of 10, delivered.

Seems like some kind of basic file allocation table or some way of indexing what is stored would be useful with this.. it would have to be written and optimised around the block-erase limitation of

flash... any thoughts on how that might be achieved , all without it getting too heavy and turning into sdfat?

Thanks for the info!

Funny thing, I wrote an article on how to do a lightweight flash file system in 1994 (back when MS FFS and FTL were duking it out for flash-card storage), but it required a more complex file system on the host. (http://www.drdobbs.com/windows/the-microsoft-flash-file-system/184409501?pgno=8)

But it depends what you want to do with it. If you want to avoid sectors and block reclaim you can fake a file system. If you need CRUD access you really can't beat SDFAT.

Or you could have a go at writing something new! There's always room for innovation and fresh perspectives!!!

Hello, thanks for writing this instructable.

I have been trying to work with this code to write and rewrite to the same location in the chip. It only seems to allow me to write to one location one time before requiring an erase to write to the same location again. Is there a way to overwrite the same location without requiring an erase? If not is there a way to just erase one page instead of the entire chip, or is this just a limitation of this type of memory chip?

Hello,

You've discovered the nature of flash memory: byte write, block erase.

Flash devices don't allow you to erase anything smaller than a block because the circuitry required to raise the memory transistor's voltage high enough to perform an erase is gigantic (by comparison). As a result, this huge circuit's cost is amortized over thousands of flash transistors, aka: a block.

Typically blocks are 64KB but there may be newer devices out there with smaller block sizes. It generally isn't a problem because Flash File Systems (in software and firmware) are constantly shuffling around data anyway for wear-leveling and block reclamation (and in some cases weird issues with multi-level cell architectures), so free blocks are always available.

Good luck, thanks for reading and commenting!

Thanks for the response!

I have an additional question, since I should be able to erase individual blocks on the chip (using w25q80bv) how do I manage to do this? Your code doesn't appear to support less than an entire chip erase and I'm unfamiliar on how to code the low level type functions I would need to implement a single block erase.

Actually I think I managed to figure it out. I can erase sectors (16 pages of 256 bytes: 4k total) at a time.

I wrote the code to erase a sector without erasing the whole chip. Here it is if you would like to use it and/or add it to your code.

```
//================================================

void erase_sector(unsigned int sector){//256 sectors of 4k bytes, 1 sector = 16 pages

Serial.print("erasing sector: ");Serial.println(sector);
```

```
_erase_sector(sector);
Serial.println("ready");
}
//================================================
void _erase_sector(word sector){
digitalWrite(SS, HIGH);
digitalWrite(SS, LOW);
SPI.transfer(WB_WRITE_ENABLE);
digitalWrite(SS, HIGH);
digitalWrite(SS, LOW);
SPI.transfer(WB_SECTOR_ERASE);
sector=sector*16;//sectors are in multiples of 16 pages
SPI.transfer((sector >> 8) & 0xFF);//send byte from front 8 bits
SPI.transfer((sector >> 0) & 0xFF);//send byte from bottom 8 bits
SPI.transfer(0); // send byte 0x00
digitalWrite(SS, HIGH);
not_busy();
}
```

Nice instrucable. But the data is actually 11000111 so the picture is correct. Counting 1's on the rising clock gives you two, not three 1's in the start of the stream.

Ah, right. Thank you. Correcting it now.

hi, i followed all your instructions and it works great.

however, i wanted to modify your program to download flash file from pc. can you explain how to do it?

thanks

Please see the comments from user GlenL, he has a very good solution for your problem.

Hi again Ptorelli, my stupid memory was damage, thats all, thanks for the help :)

Hi ptorelli, Im using a flash memory chip (serial Flash proto from Mikroelectronika) for my project buy Im having kind of problems.... First of all this is the memory I'm using: http://www.mikroe.com/add-on-boards/storage/serial...

When I tried to write on my flash memory and then read what I have already write what I see is trash, for example take a look at this.

I had tried everything but I dont Know how to solve this problem

I apologize for my english, I just speak spanish.

I would be very grateful if you can help me

Hi,

In the instruction manual for your device, Table 5 indicates the Jedec ID codes for instruction 0x9f ... get_jedec_id should return 0x1c, 0x31, 0x14.

Does that command work correctly?

Not sure if you're familiar with flash, but erased flash memory is '1', not '0'. So a "clean" flash device is full of 0xFF bytes. Programming a bit takes it from 1 to 0, erasing takes you from 0 to 1.

What is odd here is that FE03h has 7 zeros in it, and so does 08h. So 7 zeroes are being programmed, just shifted and in the write bit order! :-)

Crazy debug Idea #1.

Can you try writing different bytes to page 0, offset 2, so we can see if there is a relationship between the bits of the requested byte and the bits that were actually programmed? For example, if you write 0x55 the bit pattern would be 01010101, and if you wrote 0x95 the bit pattern would be 10010101. If the bit patterns you read back from the flash are similar maybe it will be a clue... like there might be some kind of weird timing issue on the read page command? I dunno, I'm grasping at straws here. :)

Crazy debug idea #2.
It is highly unlikley that these two commands will fix the problem but you could try explicitly setting the SPI transfer mode and slow down the clock:

```
SPI.setDataMode(0);
SPI.setClockDivider(SPI_CLOCK_DIV8);
```

Mode 0 should be the default, and your device supports mode 0 and 3. Your device also runs at 100MHz so it is very unlikely that slowing down the SPI transfer rate will do anything, but it is worth a shot.

Let me know...
Thanks!,
p

Hi, thanks for the answer, when I tried to use the command Jedec Id it returns 18h, 40h, 03h, at this moment i dont have pictures of programmed data, but the first thing I think was that data was shifted, but not, it writes just thrash in my memory...

about your crazy debug idea #2 i tried it but its just the same :/

Sorry to bother you yet again - but...

I've got it all working great (with some significant help from you). However, it seems I've burned out three of my SPI flash IC's. They work great for a while - and then fail. I'm currently working without the logic level shifter (which I realize is risky). I'll go through the level shifter going forward. My question is this... you mention using the level shifters on CS, MOSI, MISO, and SCK. But isn't MISO an output? That would mean I'd want to shift up from 3.3V to 5V (to the Arduino) right? Or do I have it wrong? I would think I'd skip the zener level shifter on this pin.

Thanks again.

No worries, please, ask away! (It actually helps me to design better to hear the problems other people encounter.) Sorry to hear you've burned out so many parts! In my anemometer I drove the chip at 6V for literally a month, some fabbed lots of silicon just work out of spec better than others. You are correct, the MISO output of the flash will be at 3.3V, and you should be OK without level shifting for any of the ATMega processors: I believe logic level high threshold for digital inputs is 2.8~2.6V for parts powered at 5V and 1.6V for 3.3V Vcc. But the other pins, yeah, downshift and drive the flash at 3.3V.

Thanks again. I'm going to solder 3 more of the IC's onto breakout boards today and give them a go with the logic level shifter. I'm also thinking about a simple resistor voltage divider as a level shifter since I don't have any 3.3V zener diodes lying around.

I'm excited to find this instructable. I wish I'd run across it several months ago. But I have a couple of questions... You mention using a 1 MB SPIFlash chip. It seems most or all of the chips available at DigiKey and Mouser are bigger than 1 MB. Is there any problem going with the bigger ones?

Mouser seems to offer only surface mount chips. Digikey has some 8-pin dips, but doesn't seem to stock any of these at all.

Jameco has this:

http://www.jameco.com/1/1/44476-m45pe10-vmn6p-m45p...

But not sure if it will work with your example code.

Can you suggest any other places to get these? Thanks!

Hi,

Thanks for the feedback! Three things to consider:

1) Flash memory is programmed by writing opcodes to the input. Not all the manufacturers use the same opcodes. You will need to read the datasheet to understand if the instructions differ from WinBond.

2) I haven't encountered any SPI flash larger than 1MB in an 8-pin DIP. if you find any, i'd be very interested.

3) The device size does impact the programming protocol. In other words, the read/write instructions issued to a WinBond flash device are the same for all densities. The only thing you need to be aware of is that reading/writing to memory outside the addressable range won't work, so you need to set the bounds of code to understand that (by polling the JEDEC device ID and determining which device is present).

Make sense?

Good luck!

P