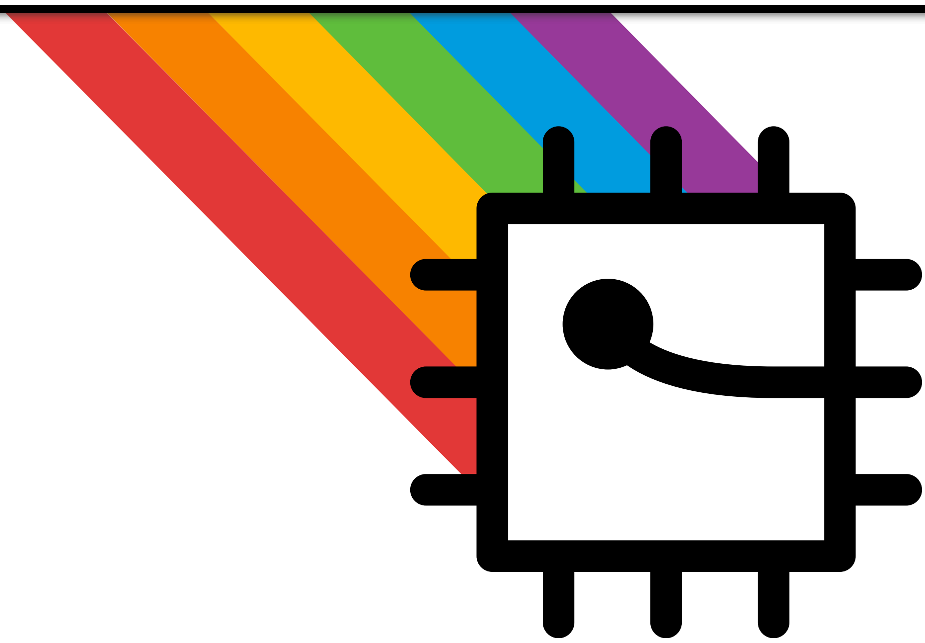
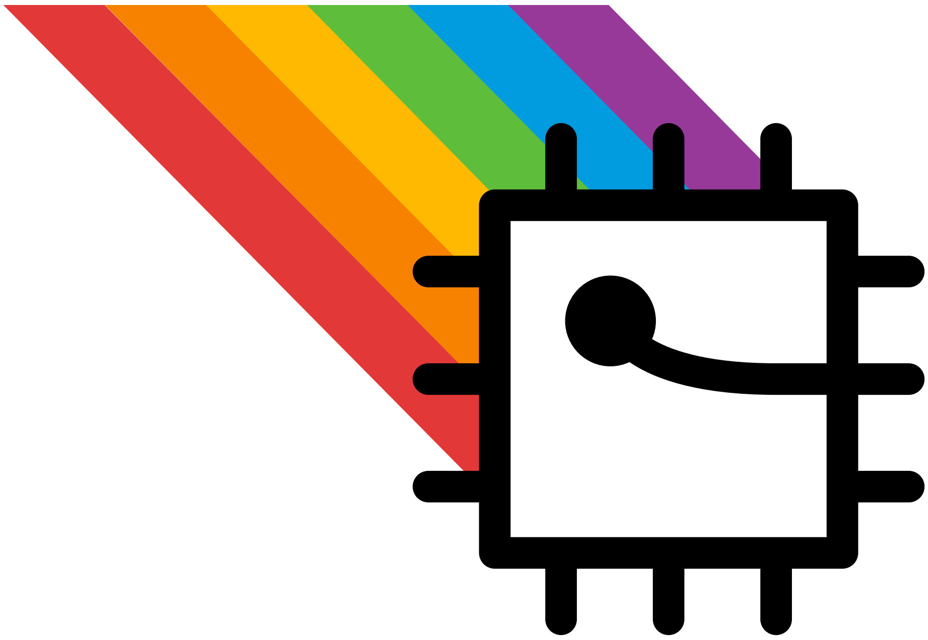




- CHIP
- PocketCHIP
- CHIP Pro

Flashing atmega chips over XIO / GPIO / SPI with avrdude

C.H.I.P.

CapnBry 2016-02-25 02:56:19 UTC #1

I have a project headed for the CHIP that uses a atmel microcontroller to handle all the real-time stuff that Linux isn't generally well suited for. We've already got one great addon board in the **SALSA DIP** which uses an ATmega328 microcontroller and I think we'll see more and more as more people start getting their hands on their CHIPs. What is inconvenient is having to program the atmega with an external ISP programmer, or having to buy a "bootloaded" chip for relatively big bucks to be able to flash it over the UART. My project currently is hosted on another small linux board and being able to just drop a fresh microcontroller in and let it magically get flashed is a real boon for inexperienced users.

Anyway, clearly we need a way to flash atmegas using the CHIP so I looked at what's available.

- The Debian install can install AVRdude 6.1 from their ARM repository.Problem: GPIO pins in AVRdude are limited to 0-31. GPIOs that low are on PORTA on the CPU and are either used for internal things or not exposed.
- Ok so maybe we just get the latest source and compile it up on the device? Well that's not fun. But nope, GPIO only goes up to 255 in AVRdude 6.3. That could be useful, if you use the linuxspi driver and use one of the Allwinner GPIOs to run the !RESET line to the AVR. You'd also need to build your own dtb to expose the spidev interface. Seems like too many steps right?
- Just write a custom app and just use memory mapped IO to toggle the CPU registers to bitbang the protocol. This is what I did in my current solution. Writing an app for every platform I want to deploy on seems like a lot of work to maintain and also isn't very reusable by others who might use different MCUs.
- There's AVRdude in the buildroot. That's helpful because we can quickly cross-compile it. Also has the 0-31 GPIO limitation but we can patch that out to get all the way up to the magic number 416 to get all the way through all the GPIO/XIO on the CHIP. We can also patch out the bugs that prevent it from working and exporting GPIO0! Still not the greatest solution but one I can roll with.

Step 1: Add a couple patches to the buildroot (package/avrdude)

```
--- avrdude-eabe067c4527bc2eedc5db9288ef5cf1818ec720/avrdude/pindefs.h.orig
+++ avrdude-eabe067c4527bc2eedc5db9288ef5cf1818ec720/avrdude/pindefs.h  2016
@@ -59,10 +59,10 @@
 #define PIN_MIN     0   /* smallest allowed pin number */
 #define PIN_MAX     31  /* largest allowed pin number */

-#ifdef HAVE_LINUX_GPIO
```

```
-#define PIN_MAX      255 /* largest allowed pin number */
+#define PIN_MAX      511 /* largest allowed pin number */
 #endif

 /** Number of pins in each element of the bitfield */
--- avrdude-eabe067c4527bc2eedc5db9288ef5cf1818ec720/avrdude/linuxgpio.c.ori
+++ avrdude-eabe067c4527bc2eedc5db9288ef5cf1818ec720/avrdude/linuxgpio.c
@@ -258,7 +258,7 @@
   //mostry LED status, can't be set to GPIO0. It can be fixed when a better
   //solution exists.
   for (i=0; i<N_PINS; i++) {
-    if ( pgm->pinno[i] != 0 ||
+    if ( (pgm->pinno[i] & PIN_MASK) != 0 ||
          i == PIN_AVR_RESET ||
          i == PIN_AVR_SCK   ||
          i == PIN_AVR_MOSI  ||
```

Step 2: Enable linuxgpio in the config as well

```
diff --git a/package/avrdude/avrdude.mk b/package/avrdude/avrdude.mk
index 1811893..da2a77a 100644
--- a/package/avrdude/avrdude.mk
+++ b/package/avrdude/avrdude.mk
@@ -16,6 +16,7 @@ AVRDUDE_DEPENDENCIES = elfutils libusb libusb-compat ncurs
        host-flex host-bison
 AVRDUDE_LICENSE = GPLv2+
 AVRDUDE_LICENSE_FILES = avrdude/COPYING
+AVRDUDE_CONF_OPTS = --enable-linuxgpio

 ifeq ($(BR2_PACKAGE_LIBFTDI1),y)
 AVRDUDE_DEPENDENCIES += libftdi1
```

Step 3: Enable the package in your buildroot menuconfig and `make avrdude`

Step 4: Install the prerequisites on your target system libelf1 and libftdi1: `sudo apt-get install libelf1 libftdi1`

Step 5: Copy the avrdude binary over to the target system

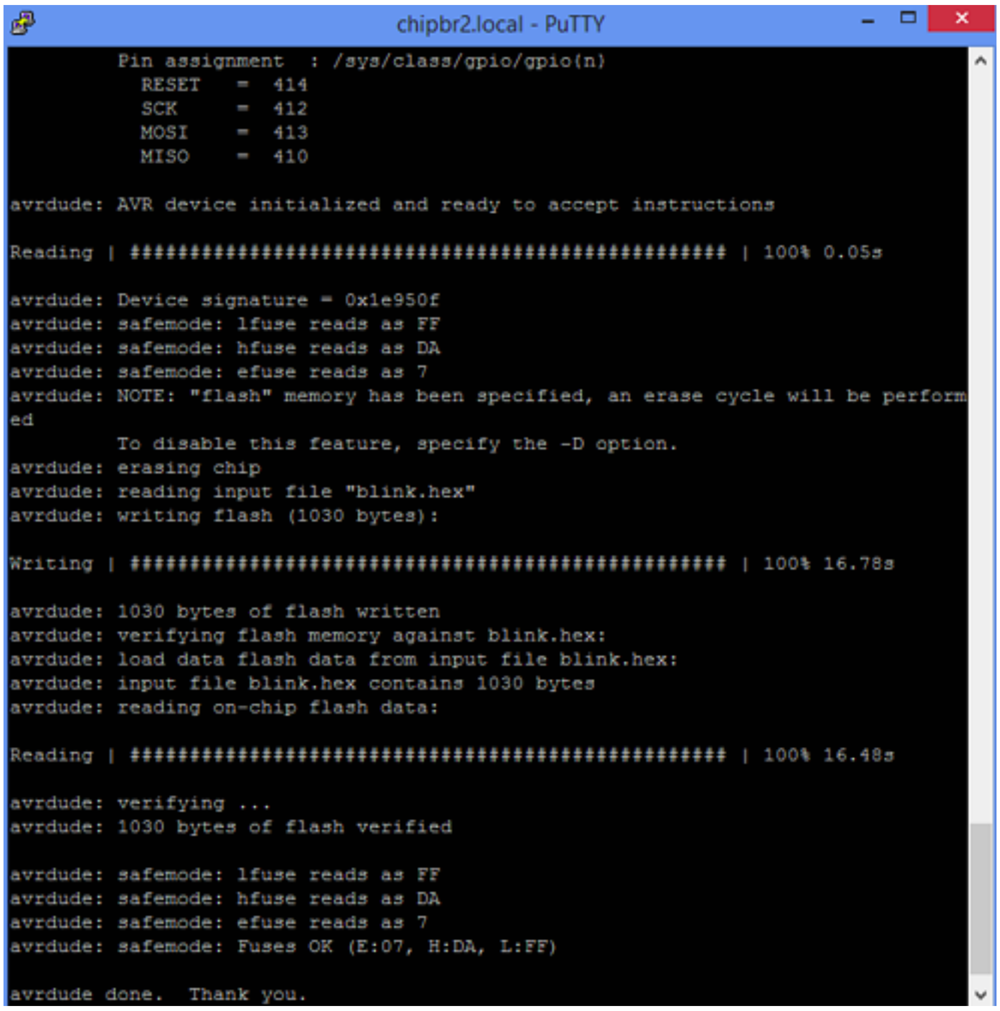
Step 6: Create an avrdude.conf that defines the GPIO programmer (you'll also need to include the definition for your target avr.

```
programmer
  id    = "gpio0";
  desc  = "Use sysfs interface to bitbang GPIO lines";
  type  = "linuxgpio";
```

```
    miso    = 410;
;
```

Flash!

```
./avrdude -C ./avrdude.conf -P gpio -c gpio0 -p m328p -V -v -v -U flash:w:bl
```



```
         Pin assignment  : /sys/class/gpio/gpio{n}
           RESET   =  414
           SCK     =  412
           MOSI    =  413
           MISO    =  410

avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.05s

avrdude: Device signature = 0x1e950f
avrdude: safemode: lfuse reads as FF
avrdude: safemode: hfuse reads as DA
avrdude: safemode: efuse reads as 7
avrdude: NOTE: "flash" memory has been specified, an erase cycle will be perform
ed
         To disable this feature, specify the -D option.
avrdude: erasing chip
avrdude: reading input file "blink.hex"
avrdude: writing flash (1030 bytes):

Writing | ################################################## | 100% 16.78s

avrdude: 1030 bytes of flash written
avrdude: verifying flash memory against blink.hex:
avrdude: load data flash data from input file blink.hex:
avrdude: input file blink.hex contains 1030 bytes
avrdude: reading on-chip flash data:

Reading | ################################################## | 100% 16.48s

avrdude: verifying ...
avrdude: 1030 bytes of flash verified

avrdude: safemode: lfuse reads as FF
avrdude: safemode: hfuse reads as DA
avrdude: safemode: efuse reads as 7
avrdude: safemode: Fuses OK (E:07, H:DA, L:FF)

avrdude done.  Thank you.
```

```
MISO  | XIO2 [.  .] XIO3
SCK   | XIO4 [.  .] XIO5 | MOSI
RESET | XIO6 [.  .] XIO7
```

Yes that is right a whopping 450 bits per second. Yeah this isn't going to work. I am used to **literally** 150x that speed on my current platform. Going to keep working on this but thought I'd check in with progress before I forget about all the steps along the way. Next I'll be moving to spidev.

CapnBry 2016-02-26 00:14:41 UTC #2

Before I moved to spidev I considered the though that XIO is on the far end of an I2C interface so maybe

I ran into a problem I'm not sure how to work around. I can export gpio 129 (PORT E1 / CSI-MCLK / SPI2-SCK) but the direction is fixed to "in".

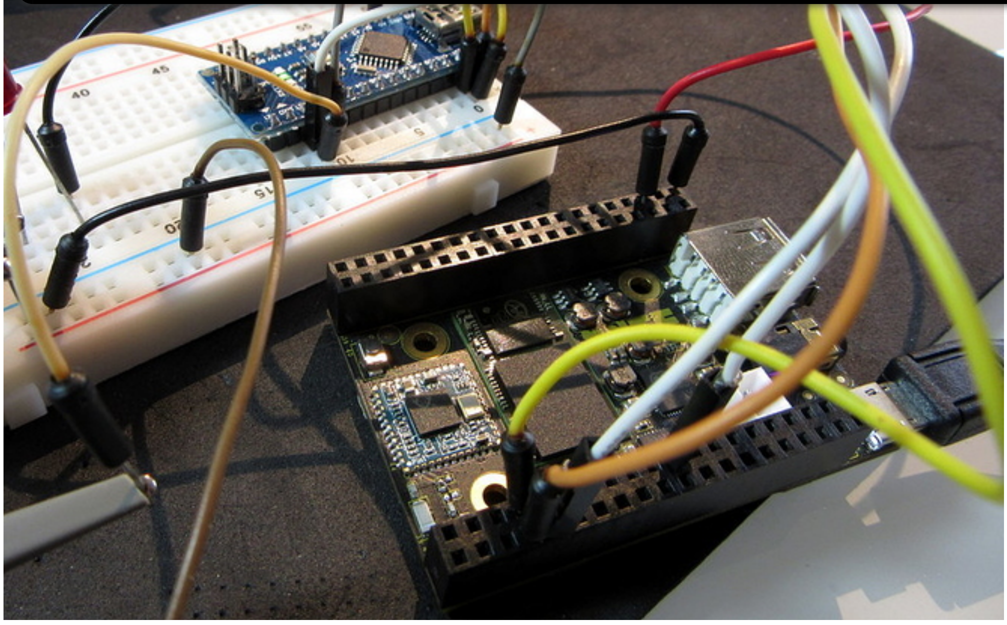
```
root@chipBR2:/home/chip# echo out > /sys/class/gpio/gpio129/direction
bash: echo: write error: Invalid argument
```

I poked around looking for a way to tell the SOC to not use this as the CSI interface, thinking there's a device on it that's got the gpio locked in that direction, however I was unable to find any device using the pins. Anyone have a clue about this?

I then moved the interface down two rows on U14 to the CSI data lines, and was able to use those just fine. The good news is that this is MUCH faster, although not all the way up to the full 4MHz you can program a 16MHz ATmega at.

```
avrdude: Device signature = 0x1e950f
avrdude: safemode: lfuse reads as FF
avrdude: safemode: hfuse reads as DA
avrdude: safemode: efuse reads as 7
avrdude: NOTE: "flash" memory has been specified, an erase cycle will be per
         To disable this feature, specify the -D option.
avrdude: erasing chip
avrdude: reading input file "heatermeter.hex"
avrdude: writing flash (24106 bytes):

Writing | ################################################## | 100% 14.66s

avrdude: 24106 bytes of flash written
```

So this will get you 13,150 bits per second of throughput. Much better than using XIO. And because posts with images get more likes, here's a messy breadboard photo if it in action. Note that only 6 wires are required between the CHIP and the ATmega, two of which you already probably have (3V3 and GND).



Designers Standard “Arduino” layouts put an LED on the SCK line (Digital 13). This is not great for programming with the XIO. The XIO pins can source *at most* 100uA, but 40uA typical. This means that to drive the SCK line, it’s going to try to put 40uA out through the LED/resistor which is going to pull the voltage down on that line. Because of this I recommend that any custom design NOT put LEDs on any of the programming lines.

---

xtacocorex 2016-02-26 01:04:18 UTC #3

I was going to suggest the other GPIO pins last night, but figured you were on the way to trying that yourself. 🙂

Good work with this, I can see this being beneficial for me in the future should I choose to program microcontrollers with my CHIP.

JKW 2016-02-26 02:23:41 UTC #4

hrhr:

> And because posts with images get more likes, here’s a messy breadboard photo if it in action

---

fordsfords 2016-02-26 12:02:18 UTC #5

> CapnBry:
>
> Standard “Arduino” layouts put an LED on the SCK line (Digital 13). This is not great for programming with the XIO. The XIO pins can source at most 100uA, but 40uA typical. … I

I assume your comment is restricted to the XIO pins, which is driven at “1” by a small current source. The CSI pins appear to be driven to 3.3V through a small (~47 ohm) internal resistance, and should have no trouble driving an LED. (Alternatively, a *custom* design could reverse the LED so that the XIO line sinks current at “0”. But of course, that will leave the LED illuminated and drawing current if the line idles at “0”.)

See:

> **Electrical characteristics of CHIP's GPIO**
>
> This post is for hardware dabblers who want to connect things to GPIO lines. There are two sets of GPIOlines, 8 XIO lines, and 8 CSI lines. The CSI lines are connected to the CPU chip, and the XIO lines are connected to an external chip which communicates to the CPU via an I2C bus. You might think that the two sets are electrically the same, but they are not. CSI input: very high impedance float (no easily detectable resistance to 3.3v or ground). output 1: active 3.26v through equiv 72 o…

---

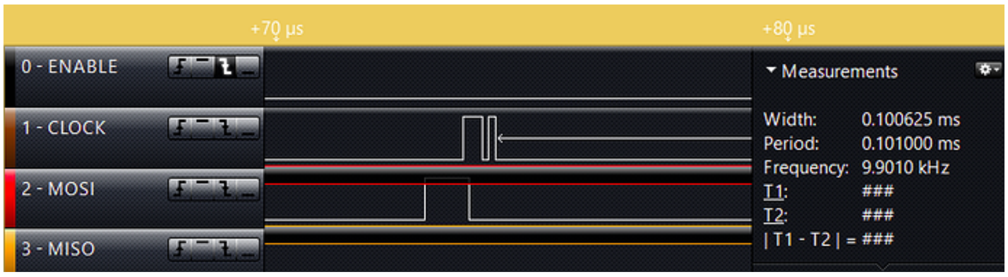
bleepbloop 2016-03-04 22:23:56 UTC #6

Hey CapnBry,
Great work!
May I ask if you had success with spidev?
Also, can you post a pin-out diagram of how you connected the ICSP pins to CHIP?
Thanks!

CapnBry 2016-03-05 19:13:27 UTC #7

> bleepbloop:
>
> May I ask if you had success with spidev?

I have not had any success with spidev. I’ve got spidev compiled as a module and loading, I’ve updated the dts file to include the spi2 definition, and I get a /dev/spidev32766.0 but when I try to use it I get garbage output. Like this doesn’t compute at all for any definition of sending a byte over SPI:



Width: 0.100625 ms
Period: 0.101000 ms
Frequency: 9.9010 kHz
T1: ###
T2: ###
| T1 - T2 | = ###

The !RESET line goes down, and there’s definitely *something* wiggling the lines but clearly there’s

```
MISO (12) | CSI_HSYNC (130) [.  .] 3V3
SCK (13)  | CSI_D0 (132)    [.  .] CSI_D1 (133) | MOSI (11)
RESET (1) | CSI_D2 (134)    [.  .] GND
```

That might look a bit confusing so here's a list
Atmega Reset Pin 1 = CSI_D2 (GPIO 134)
Atmega MOSI Pin 11 = CSI_D1 (GPIO 133)
Atmega MISO Pin 12 = CSI_HSYNC (GPIO 130)
Atmega SCK Pin 13 = CSI_D0 (GPIO 132)

Actually before I hit "Post" on this I had hooked it up to the ole oscilloscope and saw data over SPI. Only problem is that the clock period is 20ns which is… 50MHz! That just so happens to be the max speed of the A13 spi bus so what looks like is happening is the struct spi_ioc_transfer.speed_hz property isn't being respected because any speed I give it, it always comes out 50MHz.

SALSA DIP for C.H.I.P. - I2C Arduino/Ws2812/PWM/Motor

CapnBry 2016-03-05 21:40:23 UTC #8

ARGH it looks to be a bug in the sun4i SPI driver. When setting up the registers to do a transfer, the driver uses spi_device.max_speed_hz to calculate the hardware clock rate. This means you can't overwrite the value using spidev's ioctl SPI_IOC_WR_MAX_SPEED_HZ. spidev stores this value locally and calls spi_setup with it,but restores the max_speed_hz value so that the actual SPI device will always retain its maximum speed parameter (because otherwise if a user queries for the maximum *supported* speed, it will always get the last speed the device ran at!).

There are two possible solutions to this:

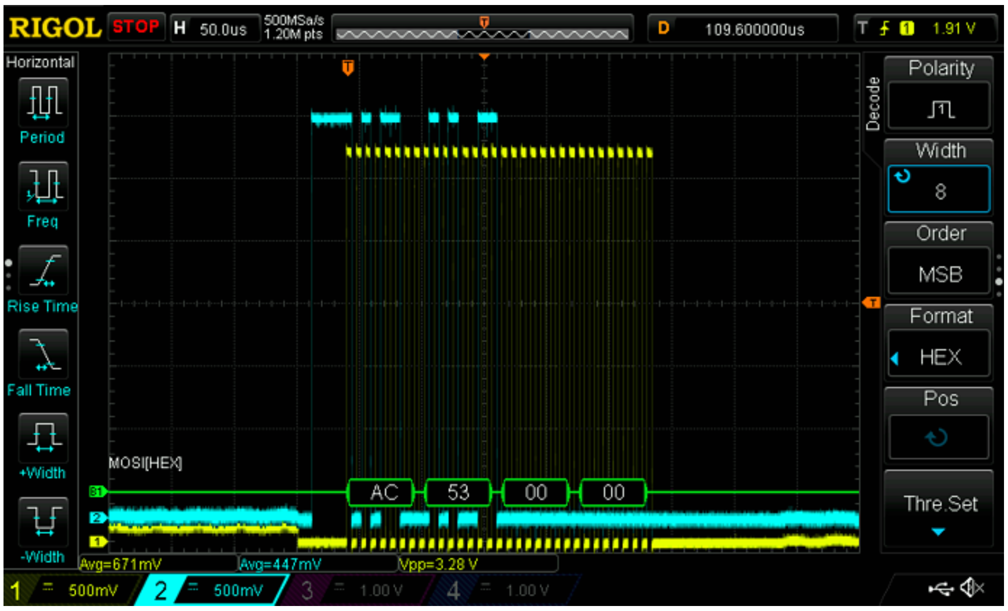
- Fix the source code to use the transfer speed_hz (which is set to the appropriate max by the spidev driver).
- Change the device tree to limit the max speed the spi bus can run at overall.

Obviously the first is the more desirable because it means any user can change the bus speed without a recompile of the dtb and reboot. Here's the source code change needed for that:

```
--- drivers/spi/spi-sun4i.c.orig        2016-03-05 16:07:00.474191693 -0500
+++ drivers/spi/spi-sun4i.c     2016-03-05 16:09:43.989979214 -0500
@@ -229,8 +229,8 @@

        /* Ensure that we have a parent clock fast enough */
        mclk_rate = clk_get_rate(sspi->mclk);
-       if (mclk_rate < (2 * spi->max_speed_hz)) {
-               clk_set_rate(sspi->mclk, 2 * spi->max_speed_hz);
```

```
        }

@@ -248,14 +248,14 @@
          * First try CDR2, and if we can't reach the expected
          * frequency, fall back to CDR1.
          */
-        div = mclk_rate / (2 * spi->max_speed_hz);
+        div = mclk_rate / (2 * tfr->speed_hz);
         if (div <= (SUN4I_CLK_CTL_CDR2_MASK + 1)) {
                 if (div > 0)
                         div--;

                 reg = SUN4I_CLK_CTL_CDR2(div) | SUN4I_CLK_CTL_DRS;
         } else {
-                div = ilog2(mclk_rate) - ilog2(spi->max_speed_hz);
+                div = ilog2(mclk_rate) - ilog2(tfr->speed_hz);
                 reg = SUN4I_CLK_CTL_CDR1(div);
         }
```

And to prove that Bob is in fact my uncle (128kHz)



And flashing an ATmega328 (Arduino Nano) at 4MHz, effective 71,425 bits per second throughput (5.4x the speed of GPIO)

```
$ ./avrdude -c linuxspi -C ./avrdude.conf -P /dev/spidev32766.0 -p m328p -V
hm.hex:i

avrdude: AVR device initialized and ready to accept instructions
```

```
avrdude: NOTE: "flash" memory has been specified, an erase cycle will be per
         To disable this feature, specify the -D option.
avrdude: erasing chip
avrdude: reading input file "hm.hex"
avrdude: writing flash (24106 bytes):

Writing | ################################################## | 100% 2.70s

avrdude: 24106 bytes of flash written

avrdude: safemode: Fuses OK (E:07, H:DA, L:FF)

avrdude done.  Thank you.
```

EDIT: And looking at Linux 4.5 looks like this bug was **fixed by Marcus Weseloh**! Although he fixed something else I noticed, which is that the !CS_0 is activated and deactivated for a short time before the transfer, which doesn't affect avrdude because it uses an external GPIO (atmegas need to keep the reset line pulled down the entire time during programming, not just during data transfer). However, CHIPsters interfacing to actual SPI devices might experience strange behavior because of it.

[solved] Enable SPI and UART on CHIP

bleepbloop 2016-03-08 03:24:41 UTC #9

Hi CapnBry,
Can you double check which pins you used for SPI?
The R8 datasheet lists different pins. Page 19 shows the following:
PE0 = SPI2_CS0 = CSIPCK
PE1 = SPI2_CLK = CSICK
PE2 = SPI2_MOSI = CSIHSYNC
PE3 = SPI2_MISO = CSIVSYNC

CapnBry 2016-03-08 05:20:45 UTC #10

> bleepbloop:
>
> Can you double check which pins you used for SPI?

Yup I should have been more clear. Each method uses a different set of pins. The SPI method uses the standard pins except not CS0 because chip select is only asserted during transfer (negative polarity), and the atmega needs the line held down through the entirety of programming. Not just during the 4 byte "write" commands.

```
SCK   (13) | XIO4 (412) [.   .] XIO5 (413) | MOSI (11)
RESET (1) | XIO6 (414) [.   .] GND
```

For linuxgpio (using the A13, 13150 bps)

```
MISO (12) | CSI_HSYNC (130) [.   .] 3V3
SCK  (13) | CSI_D0    (132) [.   .] CSI_D1 (133) | MOSI (11)
RESET (1) | CSI_D2    (134) [.   .] GND
```

For linuxspi (using the A13, 71425 bps)

```
MISO (12) | CSI_VSYNC   (131) [.   .] 3V3
SCK  (13) | CSI_MCLK/CK (129) [.   .] CSI_HSYNC (130) | MOSI (11)
RESET (1) | CSI_D0      (132) [.   .] GND
```

These little diagrams are for the ICSP header on the atmega, not the CHIP side (where the numbers in parens indicate the gpio number on the CHIP). It would be awesome if I could use the same pins for A13 gpio and spi, fallback to gpio if spi isn't available, but for some reason I can not set GPIO 129 to output from the sysfs interface.

I know all these pins, names, and gpio numbers are confusing. Now that the CHIP 1.0 is baked, have you guys considered contacting **PighiXXX** to see if he will make a beautiful layout diagram for the pins with the GPIO numbers and alt functions for your documentation? I think he does it just for funsies but if he needs to get paid, I'd be happy to split the bill with you guys, up to a couple hundred bucks. Contact me directly if you want my help **bmayland@capnbry.net**.

Connecting Arduino Pro Mini to Pocket Chip via UART?

---

knopsj 2016-08-19 14:08:53 UTC #11

I feel like an absolute dork for having to post this, but I have a few questions.

(1) I've never had to work with patching or buildroots - I honestly have no idea what to do with Step 1 [add a couple packages to the buildroot]. Is there a configuration file to edit?

(2) I recently flashed my chip to the 4.4 kernel, which changes the GPIO base to 1016, and through my extensive googling for fixes, I've seen some mentions that avrdude for other platforms (beaglebone, I think) is restricted to gpio range 0-255. Does that limitation apply here - in other words, should I just downgrade to the 4.3 kernel?

(3) You mentioned that you had a fair degree of success flashing over the CSI lines using spidev - I followed the directions in **this post**, am I following along correctly?

I may have a few more questions as time goes by, but for now I'd be indescribably grateful for any advice you can offer.