TREEHOUSE projects

Setting Up An External Crystal Clock Source (with Fuse Bits) for AVR ATMega Microcontrollers

Posted on August 28, 2012 by — 12 Comments ↓

Hi everyone,

Recently, I came across the need to utilize an external clock source for my ATMega microcontroller. I was initially under the impression that it would be a relatively daunting task after reading about the mounds of bricked AVRs which had fallen into the unforgiving grip of incorrectly set fuse bytes. I didn't find any tutorials geared specifically towards addressing this subject, suitable for beginners. Fortunately my experience wasn't bad at all, and the main reason for that (I suspect) is that I did a lot of research and asked even more questions to understand exactly what I was doing, and what I may be up against. My goal for this tutorial is to teach anyone who wants to learn exactly what each and every aspect of setting up an external clock source is, leaving no stones unturned.

In brief, I needed to set up an external clock source because of the limitations of the internal one; the limitations being speed and accuracy. I needed more speed to support V-USB (expect a tutorial on that in the future) which requires a faster clock than the internal one, and for those who don't know, the USB protocol itself is infamous for its timing accuracy requirements. This tutorial will be based on the ATMega8, but for the most part the information here will be applicable to any other ATMega you have. If you don't have the ATMega8, then I'll teach you how to read the relevant parts of the datasheet so you can easily figure out what you have to do on your own. If you know your stuff, feel free to skip sections you find irrelevant – in which case I also apologize for crowding this tutorial with basic information.

Basic Ideas

Let us first take a little while to understand the basic concepts and ideas associated with setting up an external clock source. I highly recommend opening up the datasheet for the ATMega8 (or your microcontroller) for the duration of this tutorial, and briefly skimming the chapter on System Clock and Clock Options (page 25). Click here for the ATMega8′s datasheet.

What is a clock?
A clock is simply a device that keeps track of time, it kind of gives you a beat to move to. The clock on your wall counts in increments of seconds, for example. A metronome for your instrument might give you a beat every half or full second. The amount of times a clock ticks/cycles per second is called its frequency, measured in Hertz (Hz or cycles/second). Similarly, your ATMega has a clock inside too, and its speed directly relates to how many instructions it can carry out per second (on every tick/cycle of the clock). The default clock speed that comes shipped with most AVRs is 1 MHz (1 million cycles per second) because they have internal clocks which keep time. The internal ATMega8 clock can be set up to a maximum frequency of 8 MHz, and beyond that you

need an external source or else you would be over clocking your AVR – which *could* lead to unpredictable problems.

How can we set a clock speed?
We have two options: use the internal one, or use an external source. If you are writing code that does basic stuff, and you don't require precision timing, the internal clock should suffice. In any other case, particularly for communication (i.e. using the UART for example, or in my case, USB), timing is critical. In that case, we need an alternate method, so we use things like crystals, resonators, oscillators, and clocks. They are all suitable to produce the beat we are looking for, at the frequency we are looking for, but the most common amongst hobbyists are crystals and resonator (you'll see how they are pretty much the same). We will be using a crystal for this tutorial, and it looks like this:

To use the crystal we will also require two ceramic 22 pF capacitors (so have those handy). A resonator, on the other hand, has the capacitors and crystal built into one package, thus making it a little more compact. That's pretty much the only difference, but there may be subtle differences in setting fuse bits if you are using a resonator – just something to look out for in the datasheet. Oscillators require an external power source to operate, and usually have four pins. External clocks, something a lot people don't have (but wish they did), can also be used by pumping in a square wave at the desired frequency.

Start-up time
Clocks sources usually need a little bit of time to warm up and start giving us a reliable signal when the microcontroller is turned on. This is called the start-up time. To play it safe, we will be using the maximum start-up time to give the clock as much time as it needs to get up to speed (no pun intended). Actually, the max start-up time is only a few milliseconds anyway!

What are fuse bytes?
This is the one concept which I noticed catches a lot of beginners off guard (myself included), and is the source of a lot of confusion and mishaps. Typically, there are only two fuse bytes: a high one, and a low one. As you should hopefully know, one byte contains 8 bits. So we have 16 bits to set to on or off. Each of those bits, depending on whether they are on or off, impacts the critical operations of the microcontroller. The mistake most people make is messing around with bits they did not intend to change, or giving bits they intended to change the wrong value. This is particularly easy because a bit set to 0 means it is programmed, and a bit set to 1 means it is unprogrammed – kind of counter intuitive, especially for those who wear binary wrist watches *whistles innocently*. But more on that later.

Going back to fuse bits, basically we modify a few bits in the high byte and the low byte to tell the microcontroller that the next time you start up, expect an external crystal giving you a frequency of *x* MHz, and you have to give it *y* amount of time to start up before you start running the code inside of you. That's pretty much it; look over the section called Memory Programming on page 215 just to get a general idea of what comprises a fuse byte. As you may notice, each bit in each of the two bytes has a specific name (i.e. RSTDISBL, CKSEL, SUT1, etc.). This makes it easier to refer to them, and harder to make mistakes because the name of the bit alone gives you a good idea about what its purpose likely is. For

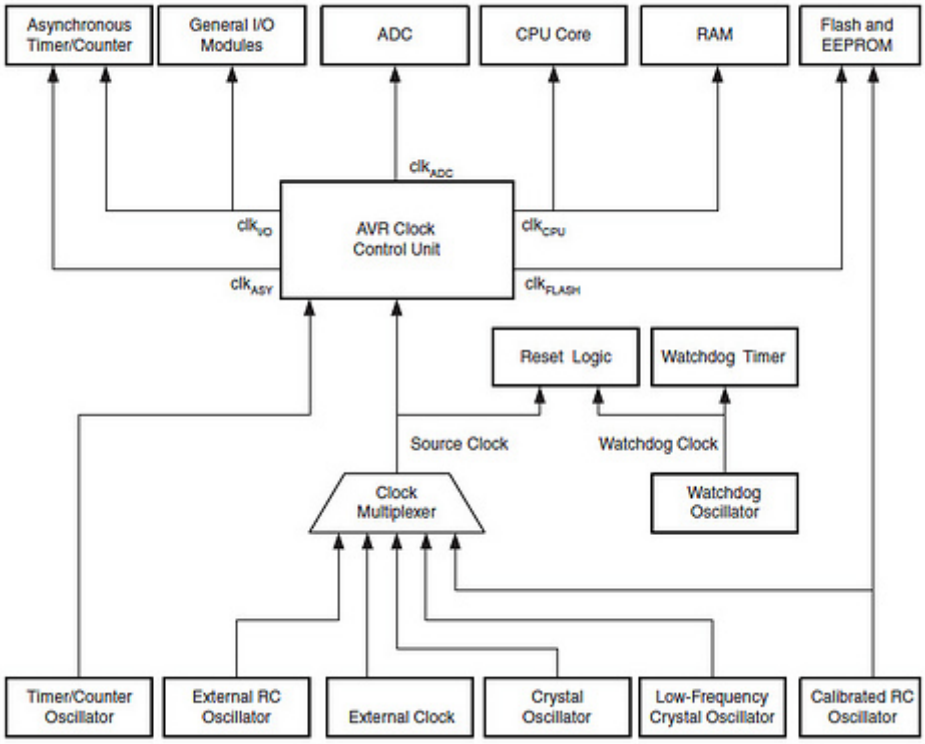
example, RSTDISBL stands for reset disable (you do not want to program this guy, unless you pretty much never want to reprogram your precious microcontroller again), CKSEL for clock select, SUT for start-up time, etc. Finally, it's important to note that the default values for each fuse byte are also listed in the datasheet.

Let's Do It!

So what do we know so far? Well, we now know that we have to set some bits to indicate we are using an external crystal, one greater than a certain frequency, and requiring a certain start-up time. Let's go through the datasheet, and bring our ideas together. Scroll over to the chapter titled System Clock and Clock Options on page 25 (or whichever page it is for your microcontroller's datasheet). So we have 7 bits to change in order to get everything working the way we want: CKOPT, CKSEL3..0 (that means CKSEL3, CKSEL2, CKSEL1, CKSEL0), and SUT1..0. We will address each bit below, with the associated tables to refer to from the datasheet. For convenience I have also added the tables and figures to the post, but they are all copied directly from the datasheet.

But first, let's take a quick peek at Figure 10:

Figure 10. Clock Distribution



As you can see, the clock multiplexer is given an input from one of the many clock sources. We will be using the Crystal Oscillator. That input goes directly to all the critical parts of the microcontroller, from the CPU and EEPROM, to RAM and your GPIO pins. The clock really is at the heart of everything, giving the entire system a beat to operate at. Pretty cool huh?

Choosing the values for the fuse bits

Just to make sure we are on the same page, CKSEL3..0 means all the CKSEL bits from CKSEL0 to CKSEL3. CKSEL stands for clock select, and tells our microcontroller what

kind of a clock we are using, and what its frequency is. Let's look at Table 2:

Table 2. Device Clocking Options Select[1]

Device Clocking Option	CKSEL3..0
External Crystal/Ceramic Resonator	1111 - 1010
External Low-frequency Crystal	1001
External RC Oscillator	1000 - 0101
Calibrated Internal RC Oscillator	0100 - 0001
External Clock	0000

Note: 1. For all fuses "1" means unprogrammed while "0" means programmed

From this, we see that CKSEL3..0 has to be somewhere between 1111 and 1010. So the four bits (CKSEL0, CKSEL1, CKSEL2, CKSEL3) are somewhere in that range. How do we know what they are exactly? That depends on our crystal's frequency, Table 2 is just telling us that if we use an external crystal, the four bits will be somewhere in that range. Right, moving forward.

Table 4. Crystal Oscillator Operating Modes

CKOPT	CKSEL3..1	Frequency Range (MHz)	Recommended Range for Capacitors C1 and C2 for Use with Crystals (pF)
1	101[1]	0.4 - 0.9	–
1	110	0.9 - 3.0	12 - 22
1	111	3.0 - 8.0	12 - 22
0	101, 110, 111	1.0 ≤	12 - 22

Note: 1. This option should not be used with crystals, only with ceramic resonators

Table 4 gives us some more detail about CKSEL3..1 (note, not CKSEL0 just yet), and tells us what our options for CKOPT are. We will program CKOPT (set it to zero) because a programmed CKOPT is needed when we are dealing with higher frequency ranges, need a full swing signal, or we are driving a second buffer with the same output from the clock. We meet the first criteria (16 MHz is > 1 MHz), so we will set CKOPT to 0. As a result, CKSEL3..1 can be any of 101, 110, or 111, it doesn't matter. We'll go with setting the three CKSEL bits to 111 just for kicks.

Good, let's move on to the final three bits we have to work on: CKSEL0, SUT1, and SUT0.

Table 5. Start-up Times for the Crystal Oscillator Clock Selection

CKSEL0	SUT1..0	Start-up Time from Power-down and Power-save	Additional Delay from Reset (V_{CC} = 5.0V)	Recommended Usage
0	00	258 CK[1]	4.1ms	Ceramic resonator, fast rising power
0	01	258 CK[1]	65ms	Ceramic resonator, slowly rising power
0	10	1K CK[2]	–	Ceramic resonator, BOD enabled
0	11	1K CK[2]	4.1ms	Ceramic resonator, fast rising power
1	00	1K CK[2]	65ms	Ceramic resonator, slowly rising power
1	01	16K CK	–	Crystal Oscillator, BOD enabled
1	10	16K CK	4.1ms	Crystal Oscillator, fast rising power
1	11	16K CK	65ms	Crystal Oscillator, slowly rising power

Notes:
1. These options should only be used when not operating close to the maximum frequency of the device, and only if frequency stability at start-up is not important for the application. These options are not suitable for crystals
2. These options are intended for use with ceramic resonators and will ensure frequency stability at start-up. They can also be used with crystals when not operating close to the maximum frequency of the device, and if frequency stability at start-up is not important for the application

As you may notice, Table 5 is the last piece of the puzzle. It tells us exactly what CKSEL0 and SUT1..0 have to be! So let's see, we decided that we want the maximum start-up time earlier. In this table, that would correspond to a Start-up Time of 16K CK and an Additional Delay of 65ms. Thus, CKSEL0, SUT0, and SUT1 would all be unprogrammed (set to 1).

We're done! To summarize our bit settings, we have:

- CKSEL0 = 1
- CKSEL1 = 1
- CKSEL2 = 1
- CKSEL3 = 1
- SUT1 = 1
- SUT2 = 1
- CKOPT = 0

Setting the fuse bytes

We know exactly which bits we need to change in the fuse bytes, so let's look at how the fuse bytes are organized. Head over to the chapter called Memory Programming on page 215. As I said, there are two fuse bytes, the high fuse byte, and the low. The bits we need to set are scattered inside these two bytes, so we need to find which one is where, and change it without changing any of the other ones. Take note of the default values in the fourth columns of the following two tables.

Table 87. Fuse High Byte

Fuse High Byte	Bit No.	Description	Default Value
RSTDISBL[4]	7	Select if PC6 is I/O pin or RESET pin	1 (unprogrammed, PC6 is RESET-pin)
WDTON	6	WDT always on	1 (unprogrammed, WDT enabled by WDTCR)
SPIEN[1]	5	Enable Serial Program and Data Downloading	0 (programmed, SPI prog. enabled)
CKOPT[2]	4	Oscillator options	1 (unprogrammed)
EESAVE	3	EEPROM memory is preserved through the Chip Erase	1 (unprogrammed, EEPROM not preserved)
BOOTSZ1	2	Select Boot Size (see Table 82 on page 213 for details)	0 (programmed)[3]
BOOTSZ0	1	Select Boot Size (see Table 82 on page 213 for details)	0 (programmed)[3]
BOOTRST	0	Select Reset Vector	1 (unprogrammed)

Table 87 shows the bits inside the high fuse byte. The only one we want to change is CKOPT (bit 4) to 0. So the high fuse byte should be 11001001 in binary, or C9 in hexadecimal.

Table 88. Fuse Low Byte

Fuse Low Byte	Bit No.	Description	Default Value
BODLEVEL	7	Brown out detector trigger level	1 (unprogrammed)
BODEN	6	Brown out detector enable	1 (unprogrammed, BOD disabled)
SUT1	5	Select start-up time	1 (unprogrammed)[1]
SUT0	4	Select start-up time	0 (programmed)[1]
CKSEL3	3	Select Clock source	0 (programmed)[2]
CKSEL2	2	Select Clock source	0 (programmed)[2]
CKSEL1	1	Select Clock source	0 (programmed)[2]
CKSEL0	0	Select Clock source	1 (unprogrammed)[2]

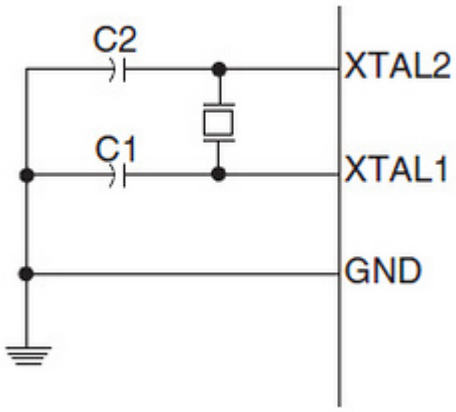
Similarly, Table 88 is the table for the low fuse byte. We have 6 bits to change in here, and in the end, our low fuse byte should be 11111111 in binary, or FF in hexadecimal.

We're almost at the final step!

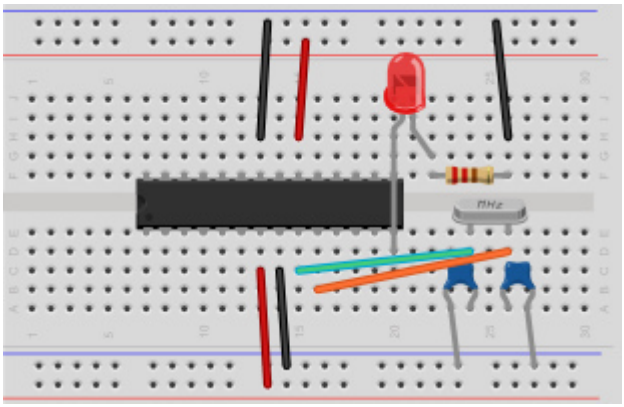
Setting Up the Hardware

Finally, you are at the point at which you are ready to burn the fuse bits into your microcontroller. But you want to make sure you have the crystal connected to your AVR too, so follow the schematic below:

Figure 11. Crystal Oscillator Connections



C1 and C2 are 22 pF ceramic capacitors. Have this setup as close together as possible, don't have wires running around the breadboard to reach the crystal and ground. You could use the following setup to try out the impact of the crystal along with the following code (skip this little bit if you don't want to try out a simple example):



DATA HOSTED WITH ♥ BY PASTEBIN.COM - DOWNLOAD RAW - SEE ORIGINAL

```
1.  /*
2.   * CrystalTest.c
3.   * Author: Treehouse Projects (www.treehouseprojects.ca)
4.   * This simple code is meant to teach how an external crystal works for an AVR ATMega
5.   * by flashing an LED on and off every 200 milliseconds
6.   */
7.
8.  //this tells the AVR how many cycles
```

So if you want, give the above minimalistic setup and code a go, and you should see the LED blink every 200 milliseconds (note that you should not connect the crystal just yet if you are going through the example, so simply don't connect the green and orange wires for now). I ignored the pull-up on the reset pin, decoupling capacitors, etc. just to keep it simple, but you should have that stuff (if you don't, this setup should still work if you did everything else right). Also, make sure you have the correct pins for your AVR model. One

mistake in the diagram is the resistor value; sorry, I forgot to change it. Let me know if you need help choosing the correct resistor, but I am sure you can handle that 😉

Now we will run the commands in avrdude to burn the fuse bits. Of course, if you don't use avrdude, use whatever tool you are comfortable with because the bytes will be the same.

```
avrdude -c usbasp -p m8 -U lfuse:w:0b11111111:m -U hfuse:w:0b11001001:m
```

Notice that we are using the actual binary values because the bits being changed are very clear. However, it is recommended to use hex values, like this:

```
avrdude -c usbasp -p m8 -U lfuse:w:0xff:m -U hfuse:w:0xC9:m
```

Just to make it clear, you wouldn't be using -c usbasp and/or -p m8 if you are not using the USBasp programmer and/or the ATMega8.

If you followed the above example, go ahead and connect the wires to the XTAL1 and XTAL2 pins on your AVR (the green and orange ones in the diagram). If you did everything right and used the exact code as the one pasted above, you will now notice the LED is not flashing. Well, that's because the F_CPU value in your code is *wrong*. Well, it isn't wrong, but your F_CPU value of 1000000 now means 1/16th of a real second is actually one second to the AVR. So your delay is no longer 200 milliseconds, but 1/16th of 200 milliseconds. The delay results in a "frame rate" of 80 flashes per second! That's too fast for the human eye to comfortably process.

So to fix this, we have to tell the AVR that because we changed the clock speed to 16 MHz, one second actually is now made up 16 million cycles, not 1 million. All we have to do is change F_CPU to 16000000UL, and we're done! Your LED should blink ever 200 milliseconds, just like it was before.

Conclusion

That was fun! I hope you learned something today, and I would like to sincerely apologize to anyone who found this tutorial filled with too much basic information. One last note: if you would like to return your fuse bit settings to the way they were from the factory, just run the same process again but use the default bit values found in Table 87 and 88 instead of the ones we came up with.

I would love to read your comments and answer your questions as well, so drop me a line below. Keep it cool everyone 😃

References

I learned everything I wrote in this tutorial from the datasheet, and my friends on the Electrical Engineering StackExchange, AVR Freaks, and Ladyada. A big thank you to everyone who helped me!

Posted in Tutorials

12 COMMENTS ON "SETTING UP AN EXTERNAL CRYSTAL CLOCK SOURCE (WITH FUSE BITS) FOR AVR ATMEGA MICROCONTROLLERS"

budi says:

December 17, 2012 at 5:01 am

thank you very much,helping me a lot…

Reply

sojib says:

January 24, 2013 at 1:12 pm

great tutorial! it helps me a lot.

Reply

Soubheek Nath says:

March 13, 2013 at 12:30 am

Really a good tutorial.. But I'm cann't write it up on BASCOM language for ATMega8.. If you please help, then thankfull to you..

Reply

Treehouse Projects says:

March 17, 2013 at 8:49 am

I don't know how BASCOM works, unfortunately. Really sorry about that. I actually did not even know about BASCOM until you mentioned it, but it looks intriguing. I guess WINAVR and the entire GCC toolchain for AVR just took over.

Anyway, sorry I can't help Soubheek. I would suggest looking through here, if you haven't already http://avrhelp.mcselec.com/index.html

Reply

shaggy says:

December 10, 2013 at 4:04 am

Ur tuto is very helpful but i cant read fuse bytes of atmega128 pls help

Reply

JanGeox says:

April 29, 2013 at 3:51 pm

Great info, very clear. Thanks a lot !

Reply

Kamran says:

September 10, 2013 at 11:57 pm

Hi,
Thanks for the tutorial. I learnt a little bit about the fuse bytes. But now I'm not sure if I know how to modify them correctly.

I did all the steps as mentioned, but at the end got this error when tried to program my avr:

avrdude: initialization failed, rc=-1
Double check connections and try again, or use -F to override this check.

And I couldn't fix it. I'm sure that the problem is not coming from improper connections. I looked online and it seems that many people are struggling with that. Probably I broke my micro.

I'm just curious to know what I did wrong. Any idea?

Reply

Treehouse Projects says:

September 18, 2013 at 4:34 pm

Hi Kamran,

I often get this error also, and it often really is due to a problem in the connections. Sometimes just unplugging and re-plugging the USBasp works for me. You can try triangulating the problem by using another microcontroller. Is the microcontroller you are trying to use a fresh one?

Someone once suggested connecting all the grounds together. Although it doesn't make much sense to me, who knows – give it a try if nothing else works.

Keep me posted.

Reply

SERAJ says:

October 23, 2013 at 5:44 am

Great one! can you tell me how the fuses will look like for 32.768khz and atmega8?
is it the one that say "External low Frequency Crystal"

just tell me the values of Hfuse and lfuse, so I can compare them with mine to see if I'm thinking right,
which timer will fit better for this settings?
Thank you so much.

Reply

Treehouse Projects says:
November 7, 2013 at 10:51 pm

Hey,

I am getting: -U lfuse:w:0xfb:m -U hfuse:w:0xd9:m

This calculator is very helpful: http://www.engbedded.com/fusecalc

Reply

Brijesh says:
January 18, 2014 at 5:24 am

Hey thanks for giving such information about Fuse. Also I want to know how to set fuse for atmega644A PU running on 16Mhz crystal. From datasheet when I calculated it comes out lfuse-0x7F and hfuse-0×99, is it correct? since for atmega644A I did not find CKOPT.
I am little bit confuse how to calculate using http://www.engbedded.com/fusecalc/
how to include Brown out detection do u have any idea?
datasheet
add:http://images.ihscontent.net/vipimages/VipMasterIC/IC/ATML/ATMLS06085/ATMLS06085-1.pdf

Reply

Treehouse Projects says:
February 8, 2014 at 12:26 pm

Hmm I'm getting `-U lfuse:w:0xff:m -U hfuse:w:0x99:m -U efuse:w:0xff:m` for a 16 MHz with no clock divide. If you want to add brown out, in the calculator just click on the brown out level detection you want in the "Manual Fuse Bit Configuration".

Reply

Leave a Reply

Your email address will not be published. Required fields are marked *

Name *

Email *

Website

Comment

Post Comment

2 PINGS/TRACKBACKS FOR "SETTING UP AN EXTERNAL CRYSTAL CLOCK SOURCE (WITH FUSE BITS) FOR AVR ATMEGA MICROCONTROLLERS"

Reprogramming AVRs: Preliminaries « Tesla UIs says:

January 8, 2013 at 7:44 am

[...] which serves as the clock source while reprogramming the chip. I found one thread and a blog post about it. In another blog post they use PonyProg to set the fuse bits. More to come, when I have [...]

CATazine Live » lock/fuse bits in AVR says:

January 26, 2014 at 10:39 am

[...] http://www.codingwithcody.com/2011/04/arduino-default-fuse-settings/ http://treehouseprojects.ca/fusebits/ http://embedderslife.wordpress.com/2012/08/20/fuse-bits-arent-that-scary/ [...]

In Archive

- September 2013
- June 2013
- May 2013
- April 2013
- March 2013
- February 2013

- January 2013
- September 2012
- August 2012
- July 2012
- June 2012

↑