ATF22V10C Programming Algorithm

This page is about the programming algorithm (what you need to know if you want to build your own programmer) for the ATF22V10C.

Random Conversation

The programming algorithm is essentially undocumented, both officially and unofficially. Search for Atmel documents about how to program the chips and all you end up with are two-page PDF documents that say "buy one of these commercial programmers." Search for random internet information on the topic and all you find are newsgroup postings that say "it's a secret, no one knows."

I was able to find a homebrew programmer with source code, but it didn't support the ATF22V10C. I found another homebrew programmer for which the author said that he didn't know how to make it compatible with ATF22V10, but he did know of a piece of software (closed-source and compiled for Windows 3.1) which could utilize his circuit to program the chips, assuming you still have a PC that has a parallel port. I then found something on sourceforge apparently created much more recently, specifically for the ATF22V10C, but both the source code and a text file of notes from the author proved to be so utterly incomprehensible that I could have figured out how to program the chip in half the time had I simply never found that project. How Atmel manages to sell the chips in such a vacuum of information is beyond my understanding. I realize there are commercial programmers, but judging from what I've read on the internet, most of them fail to program the ATF22V10 as well.

...but that doesn't matter, as I'm not the type to buy chips that I can't program myself, and so using someone else's programmer, homebrew or commercial, was never an option anyway. So I took what information I could find for the GAL22V10, applied that process to the ATF22V10, figured out why that process wasn't working and how to correct it, and this page documents the results.

Programming Pins

During programming, 5 volts should be applied to pin 24 and ground to pin 12, as is done so when not programming.

Pin 1, and pins 15 through 23, are not used during programming. Every document I read suggest that they should be tied low with 4.7 kΩ resistors, and totally not 10 kΩ resistors because they must be pulled low enough to counteract internal pull-up resistors, otherwise invalid logic states may exist, even for the output pins since they might be inputs. Needless to say, this all sounds like a bunch of bullshit, and so I removed the damn resistors from my design and, surprise, it seems to make no difference whatsoever. So if you feel like being pedantic you can put some 4.7 kΩ pull-down resistors on these pins, but I wouldn't bother.

Pin 2 is the programming enable pin. You enable programming by applying 12 volts to this pin. Documents I've read suggest that this should happen 10 ms after 5 volts is applied to pin 24, and so that is what I have done, though I suspect the chip doesn't really care as long as you're sure that it has 5 volts on pin 24 before you apply 12 volts to pin 2.

One circuit design I saw on the internet had a button wired between the programming voltage and pin 2. While the programmer circuit also controlled the programming voltage automatically, this switch served as a fail-safe in case the user left a chip in their programmer when they shouldn't have. I seriously recommend that you include such a button in your programmer. I didn't have one at first, but added one after rendering pin 2 useless for logic input on one of my chips. (The chip otherwise still works, so it isn't a complete loss.) If you decide not to have such a safety button, be sure that you realize that in addition to ensuring that the programming voltage isn't applied during power-up of your programmer, you must ensure that it isn't applied during power-down as well, and also during weird conditions such as slow rise/fall times of supply voltage, brown-outs (which are common with USB power), etc. ...or just add the fucking button.

Pin 3 is the P/$\overline{\text{V}}$ pin, which I assume means "program/$\overline{\text{verify}}$." When low you can read from the chip, and when high you can write to the chip. The state of this pin matters only when the strobe pin is brought low, otherwise it may be in either state without any effect.

Pins 4 through 9 are "row address" pins. In old GAL designs, these pins selected which row of the EPROM data you were writing to. In the ATF22V10, row addresses are instead shifted into the chip along with the data to be programmed into those rows. Thus, these row address pins are almost unused. There are only two patterns the chip pays any attention to, resulting in 3 modes of operation:

> If pins 4, 6, 7, and 9 are high, then bringing the strobe pin low for 10 ms while pin 3 is high will cause the chip to be erased.
>
> If the above condition is not met, and if pin 8 is high, then the 20 bits which control latching and active high/low of the output pins are accessed.
>
> If neither of the above conditions are met, then all of the remaining bits in the chip are accessed.
>
> The state of pin 5 seems to be entirely irrelevant.

Pin 10 is the clock pin. It only took me about 8 hours to figure out the correct polarity of this pin. As best I can tell, the best approach is that at power-up, this pin should be high, then for each bit-shifting operation, you should first sample the chip's data output pin, then lower the clock, then change the chip's data input pin, then raise the clock.

As it presently exists, my programmer never makes use of a clock signal that remains high or low for less than 10 µs. So you can go at least as fast as 100 kHz in some cases, though in other cases my programmer goes much slower (it is just a bit-bang interface), so perhaps 20 kHz is a safer choice if you want to be certain you're going slow enough.

Pin 11 is the serial data input pin. Its state is sampled by the chip on the rising edge of the clock signal.

Pin 13 is the strobe pin. It is active low. Because of this, it must be brought high after 5 volts is applied to pin 24, but before 12 volts is applied to pin 2.

For erasing the chip, it seems that the strobe pin must be held low for 10 ms, as a simple pulse won't erase the chip, even if I delay for 100 ms after giving that short pulse. I'm not certain that the pulse length is essential. It's possible that its timing is used to time the erase operation, but it's also possible that because the erase process is so easy to set up (just raise pins 3, 4, 6, 7, and 9, then lower pin 13) that there is a minimum strobe length simply to prevent circuit glitches from erasing the chip. I've so far only tested pulses of 10 µs and 50 ms in addition to the recommended 10 ms, and the only ill effects I noticed was that the 10 µs pulse failed to erase the chip. The 50 ms pulse didn't seem to over-erase the chip, as subsequent programming succeeded as usual.

For general programming and for reading the chip, a 10 µs pulse seems to work just fine, even though one source I read claims that the programming pulse must last for 5 ms. I've tested 10 µs pulses and 50 ms pulses and I saw no ill effects from either extreme. So I suspect the chip times the write operations itself. It likely still needs some time to perform the operation, but it isn't clear to me how much time that is. I've waited as little as 2 ms before shifting more bits into the chip for the next write operation, but with that bit shifting requiring 6 ms, the chip still has 8 ms between write strobes. Just to be on the safe side, I recommend making the pulse 5 ms anyway, and waiting at least another 5 ms before issuing another write pulse, as 10 ms is commonly how much time EEPROM chips require to program data into themselves.

Pin 14 is the serial data out pin. It should be sampled only when the clock pin is high. When the clock pin is low, the output is on some occasions merely an inverted echo of the serial data input pin, and so the pin cannot be sampled on either edge of the clock, but must be sampled when the clock is high.

Programming Process

Entering Programming Mode

Apply 5 volts to pin 24, wait 5 ms, raise the strobe and clock signals, wait another 5 ms, then apply 12 volts to pin 2.

Reading / Verifying

1. Set a low voltage on the P/$\overline{\text{V}}$ pin.
2. Set one or all of pins 4, 6, 7 and 9 low. Set pin 8 according to the type of data you want to read.
3. Clock a zero bit into the chip for each data bit you plan to read.
4. If the data is addressed, clock the 6 address bits, most-significant bit first, into the chip.
5. Bring the strobe pin low, then high again.
6. Clock the data bits out of the chip.

Erasing / Wiping

1. Set a high voltage on the P/$\overline{V}$ pin.
2. Set pins 4, 6, 7 and 9 to a high state.
3. Bring the strobe pin low, keep it low for 10 ms, then raise it.
4. Wait another 10 ms just for shits and giggles. (no idea if its necessary, but just to be safe)

Writing / Programming

1. Set a high voltage on the P/$\overline{V}$ pin.
2. Set one or all of pins 4, 6, 7 and 9 low. Set pin 8 according to the type of data you want to write.
3. Clock the data bits into the chip.
4. If the data is addressed, clock the 6 address bits, most-significant bit first, into the chip.
5. Bring the strobe pin low, keep it low for 5 ms, then raise it.
6. Wait another 10 ms just for shits and giggles. (no idea if its necessary, but just to be safe)

Leaving Programming Mode

Remove the 12 volts from pin 2, wait 5 ms, lower the strobe and clock signals, wait another 5 ms, then remove the 5 volts from pin 24.

Fuse Data

The main data area is accessed when pin 8 and at least one of pins 4, 6, 7 and 9 are low. It consists of 64 rows (requiring a 6 bit address) of 132 bits each. Rows 0 through 43 contain the bulk of the fuse data which configures the operation of the device. Row 44 will store data but is not utilized by the chip, so you can store identification information there if you'd like, in particular you might want to utilize that area to track how many times each chip has been reprogrammed, in order to know when a chip is near its end of life. Row 58 contains the text "1F22V10C3" stored in reverse order, with both the bits and the bytes reversed. This row can be written to, but unlike the rest of the chip which is reset to the default state whenever the chip erase command is used, any writes to row 58 are irreversable. Row 59 has no data, but writing to it disables the power-down feature of the ATF22V10C, which is otherwise activated by default. Row 61 also has no data, but writing to it actiaves the "security fuse" which prevents the data in the chip from being read by returning all 1 bits whenever the chip is read. All other rows contain no data and writing to them has no effect.

The remaining 20 bits of fuse data are accessed by making pin 8 high and at least one of pins 4, 6, 7 and 9 low. In this mode, row addresses are not shifted into the chip, as there are no row addresses in this mode. Instead you simply shift the data in and out of the chip before or after activating the strobe.

The bulk of the fuse data configures an array of AND gates which feed in groups of 8 to 16 (depending on which output pin we're speaking of) into an OR gate, the output of which goes into "macrocells" which are configured by those extra 20 bits of data, with two bits configuring each pin's macrocell.

This data can be obtained from what is referred to as a JEDEC file (as apparently electronics people just aren't happy if they aren't naming file formats after standards organizations) which is produced from software like WinCUPL which is available for free from Atmel. Documentation for the JEDEC file is easy to find, so I won't describe it in depth here, but there are two details you won't find in descriptions of the file format:

1. The chip expects 44 rows of 132 bits, but the data in the JEDEC file is arranged as 132 rows of 44 bits. So you need to reorganize it. Row 0 in the chip should receive bits 0, 44, 88, 132, ..., 5676, 5720, 5764, from the JEDEC file. Then row 1 contains bits 1, 45, 89, ..., 5677, 5721, 5765. Finally row 43 contains bits 43, 87, 131, ..., 5719, 5763, 5807.

2. The 20 bits that program the macrocells are at offset 5808 in the JEDEC data. They're in the wrong order too, in that each pair of bits needs to be reversed. So first you shift bit 1 into the chip, then bit 0, then bit 3, then bit 2, then bit 5, then bit 4, etc.

If you wish to construct your own fuse data, have a look at page 5 of this datasheet. You'll note numbers written next to various lines. These are the address in the JEDEC file where each bit belongs. Set all of the bits to 1, then put a 0 at addresses that coorespond to where you want to join lines in the matrix. For AND gates which you do not wish to use, you can either make all bits 1 to cause the AND gate to output a high level, or make all bits 0 to cause the AND gate to output a low level. The two bits that configure the macrocells are explained on page 4. Keep in mind when trying to design circuits that (a | b | c) == !(!a & !b & !c) as this allows you to put complex equations into a single AND gate row rather than use multiple rows. There are really enough AND gate rows in there that I can't imagine running out, but if you want to, for example, make pin 23 output the logical OR of all 12 of the input pins, you'll have to use that trick to make it happen as it only has 8 AND gates assigned to it, and since those AND gates are the inputs to the OR gate, there are only 8 inputs to that OR gate, so you have to perform the operation with one of the AND gates instead, which have many more inputs. (...and WinCUPL is incapable of figuring that out, so if you don't realize it can be done, WinCUPL isn't going to help you.) Another useful equation, as there aren't XOR gates in there, is that a ^ b = (a & !b) | (!a & b). Having both the inverted and non-inverted version of each input pin available makes constructing complex logic out of such a simple array rather easy.

Unsolved Mysteries

The ATF22V10C does many strange things.

Strange Behavior when Clock Input is Low

While the data on the serial data output pin during the low half of the clock cycle is correct in most cases, for very particular bits of very particular rows (repeatable for every run with every chip I have), it is merely the inverse of the current state of the serial data input pin. Exactly why it does this I have no idea, but it only happens on certain bits of certain rows and it's the same with every chip I have. Whatever the reason, the result is that the serial data output pin must be sampled when the clock is high. Sampling it when it is low, or on either the rising or falling edge, will cause random failures on those bits. I'm tempted to think that this is because of special hidden features in the chip, but absent of any reason to suspect that the chip has hidden features (as I've already figured out how to make it do everything it is documented to do) it's a lot easier to just assume the chip is glitchy.

The Second Bit of Row 31

Correctly reading bit 1 of row 31 takes some extra effort. This mystery is similar to the previous mystery, except that the chip always outputs a zero bit initially, but then, if the bit is actually programmed as a 1, and if the serial data input is changed while the chip is outputting the false 0 bit, it will switch the output to a 1 bit and keep it a 1 bit regardless of further changes on the serial data input. There are four other bits with the same problem (bits 0 of rows 45 and 62, and bit 1 of row 63) but thankfully there's nothing to read in those locations, so they can be ignored. Bit 1 of row 31 is annoying in that it's part of the fuse array that controls the device's function.

I have discovered a totally ridiculous algorithm which is able to correctly read all of the bits:

1. sample the serial data output
2. lower the clock input
3. apply a high level to the serial data input
4. sample the serial data output again
5. apply a low level to the serial data input
6. sample the serial data output again
7. raise the clock input

After these steps are complete, you have three versions of this particular bit. If they are "000" or "111" then there is no question as to whether the bit is a 0 or a 1. However, in a small number of cases, you'll have one of "001", "010", "101" or "110". In these cases, the chip is simply doing its rare but usual thing of making the serial data output mimic the serial data input when the clock is low, which is why I recommend above that you sample the serial data output pin when the clock is high. However, on bits like bit 1 of row 31, you'll sometimes have "011" or "100". In these cases, the bit that the chip output when the clock was high is incorrect, and so you should accept the bit that it output after you started playing with the serial data input while the clock was low.

Half-Disabled Power-Down Pin

The power-down pin, after being disabled, sometimes is only half-disabled, in that the chip will be in power-down mode when it is first turned on and remain in that mode until pin 4 becomes high, at which point it latches the non-powered-down mode and never returns to it regardless of what happens to pin 4. I've seen this happen to two chips, but both have since ceased to do it, leaving me with nothing to experiment with to find a solution.