A method for determining the EPROM contents of a programmed MC68705U3/R3 microcontroller.

Peter Ihnat
pihnat@uow.edu.au
Nov 2010

Background:
Many years ago I mentioned in a newsgroup that I possibly had a way of determining the EPROM contents of the Motorola MC68705x3 microcontrollers. But the family became obsolete so I didn't investigate the idea further and moved onto using other chips. However now I have a micro that I need to copy and I don't have the source or object code. After all that time I finally decided to experiment and see if the idea would work. This document is a summary of what I did.

I assume the reader is familiar with the Motorola MC68705P/U/R series of microcontrollers so I won't go into any introductory discussions of micros here. I'll concentrate on the actual programming procedure and the Bootstrap ROM in the MC68705U3/R3 and what I did to read the information back out of its EPROM. With a few changes this information might also be useful for the P3 micro.

****
DISCLAIMER: Unless you know exactly what you're doing you could overwrite all the data in a micro you're trying to read. All I'm describing here is what I did to read an MC68705U3. If you use any of the following information you do so at your own risk.
****

Bootstrap ROM:
I previously built the suggested Programmer which is shown in the MC68705U/R data sheets (also refer to AN857_REV2.pdf). It runs the code contained in the Bootstrap ROM which burns the contents of an external EPROM into its own internal EPROM and then verifies that the contents are correct. You'll be familiar with the micro's vectors located at addresses 0FF8H to 0FFFH. There is another vector at addresses 0FF6H and 0FF7H which are the last 2 bytes of the Bootstrap ROM. This vector points to the start of the ROM and is fetched if +12V is applied to the TIMER input (pin 8) when the Programmer is powered up. This means the Programmer automatically runs the Bootstrap ROM code.

I wrote a simple program to display the contents of the Bootstrap ROM on 8 LEDs, byte by byte. I disassembled it to understand how it operates. If you want to see how it works refer to my article "MC68705U3 Bootstrap ROM Listing.pdf".

As you know the micro has no built-in 'contents dump' routine and the 'verify' procedure is part of the Bootstrap programming routine. But apparently the micros have a Test mode. I've never seen any information about this mode but I believe it's supposed to enable the micro to run a program from an external EPROM. That means you could write a program to simply dump the contents of the internal EPROM.

Since I don't have info about this mode of operation I looked at exploiting any weaknesses in the Bootstrap ROM code to see if it's possible to determine the contents of the internal EPROM. I did find one.

Some observations:

1. The first thing the Bootstrap code does is copy itself into RAM. It does this so it can modify itself. It basically runs twice. The first time it runs it performs the programming procedure. It then modifies some of its code so that the second time it runs it performs a verify.

2. When the EPROM is being programmed, bytes which are 0 are skipped ie not programmed. Non-EPROM addresses are also skipped (first 128 bytes and the block from 0F3F to 0FF7 inc).

3. Just before each byte is read from the external EPROM the Bootstrap code checks the micro's INT input (pin 3). If it's high it skips reading the EPROM. Note that the MC68705U/R datasheets shows this pin connected to 0V in their suggested Programmer so it always reads the external EPROM.

4. When the programming procedure starts, the Bootstrap code does NOT check to see if the +21V programming voltage on pin 7 (Vpp) is present. This means if you leave pin 7 at +5V the Bootstrap code will perform the whole programming procedure without the EPROM actually being programmed ie the existing internal EPROM data remains intact.

5. When the Bootstrap code does the verify procedure it checks the whole EPROM. At the end, if all the programmed code verified correctly it executes a "bclr" which switches the 'verified' LED on. If during the verify procedure a byte does NOT verify correctly the Bootstrap code modifies itself. It changes the "bclr" to a "bset" so at the end of the procedure it does a "bset" instead which leaves the LED off.
   Now for the interesting part – the Bootstrap code changes the "bclr" to a "bset" EVERY TIME a byte fails verification.

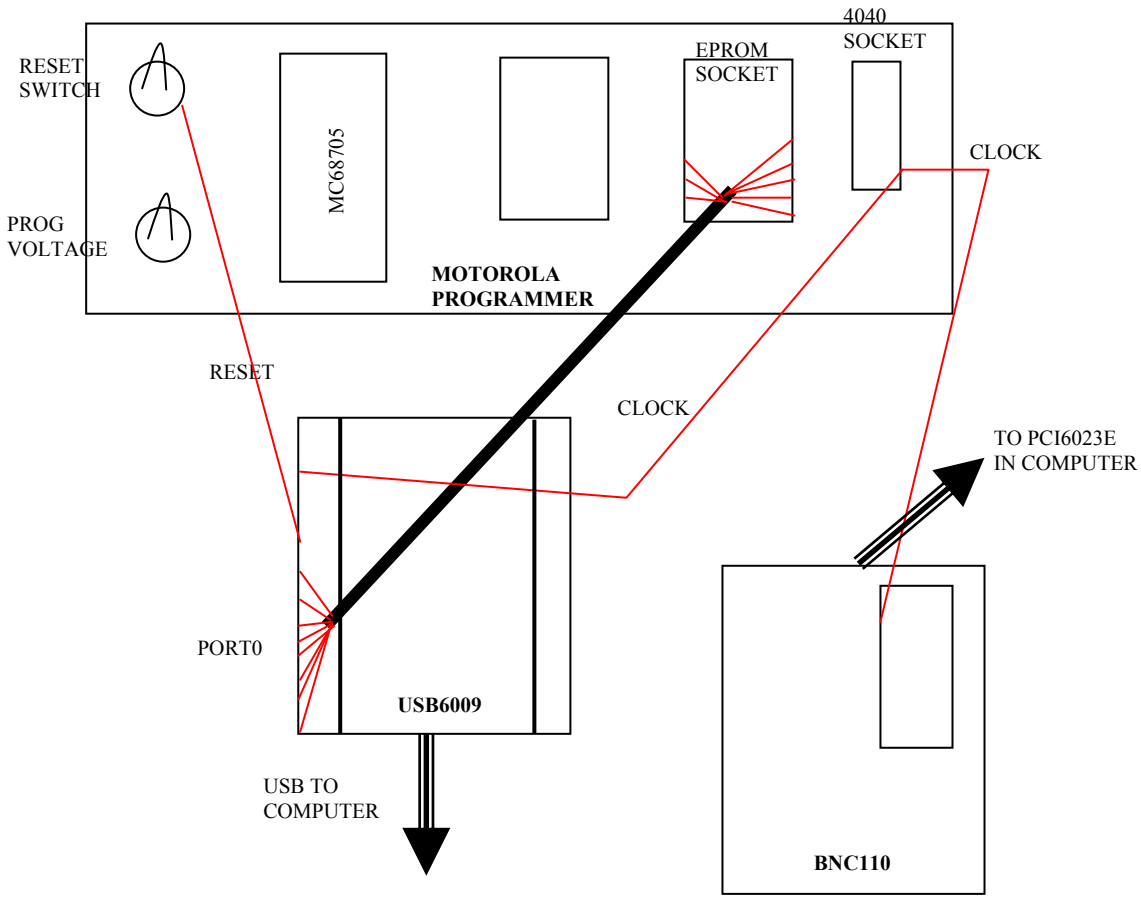
So how did I extract the data from a pre-programmed micro? I used observations #2, #4 and #5 above. Using the Programmer suggested in the Motorola datasheets I removed the external EPROM and 4040 counter and connected it to a computer so I could apply bytes of my choice. Then I measured the time between each of the CLOCK pulses which would normally clock the 4040 counter during the program/verify procedure. Every time a byte fails verification the time to the next clock pulse is slightly longer than the others. This is because of observation #5 – the Bootstrap code takes time to modify the "bclr" to a "bset". The difference in my case was about 28μsec.
So the idea is you apply 00 to the micro and perform a pretend program/verify procedure (Vpp = 5V instead of 21V). All bytes in the internal EPROM which are 00 will have shorter times to the next clock pulse during the verify procedure. Then you apply 01 and do the same, etc for each byte up to FFH. I'll explain my setup in more detail in the next section. There is one catch however – this technique doesn't give the last byte since there's no clock pulse after the last verify. It didn't matter in my case since it was obvious what the value was supposed to be but there is a way to do it. I leave it as an exercise for someone else.

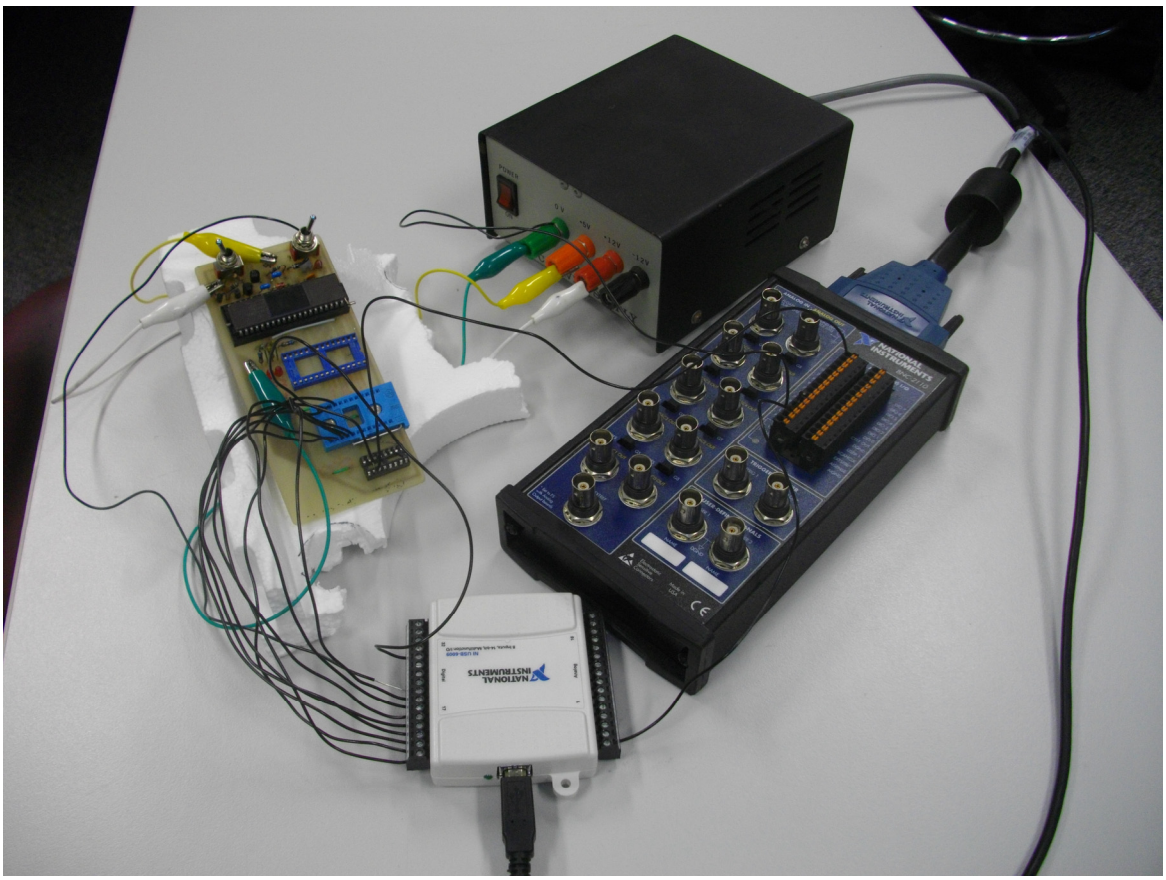
Details:
The best way to perform the above procedure is to use a Data Acquisition System. I use National Instrument's LabView so as a first attempt I tried their USB6009 interface. This has several digital outputs so I used Port0 to feed bytes to the micro in place of the external EPROM. I also used bit0 from Port1 to switch the Programmer's RESET input (otherwise

you have to operate the reset switch manually for each byte being tested). The problem with this DAQ is that even though it has a built-in counter it doesn't support period measurements. So I fed the CLOCK signal from the micro to AI0, one of the analogue inputs. I then wrote a program in LabView to apply a byte to the micro starting with 00, removing the RESET and then sampling as fast as possible to capture the CLOCK signal. Then I repeated this for each byte up to FFH.

Unfortunately this method was unreliable. I got periods between clock pulses of about 14-16 samples when the applied byte matched the EPROM byte and 16-17 samples when it didn't. The problem was the sampling rate which is 48KHz max for this unit. It needed to be much faster.

Luckily I also had access to a PCI6023E DAQ. This only has 8 digital outputs (I needed 9) but its counters support pulse width measurements – I could measure times between consecutive rising edges, falling edges or both. So I ended up using the USB6009 for the digital part and the PCI6023E with a BNC2110 adapter for the clock part. My set up was as follows:

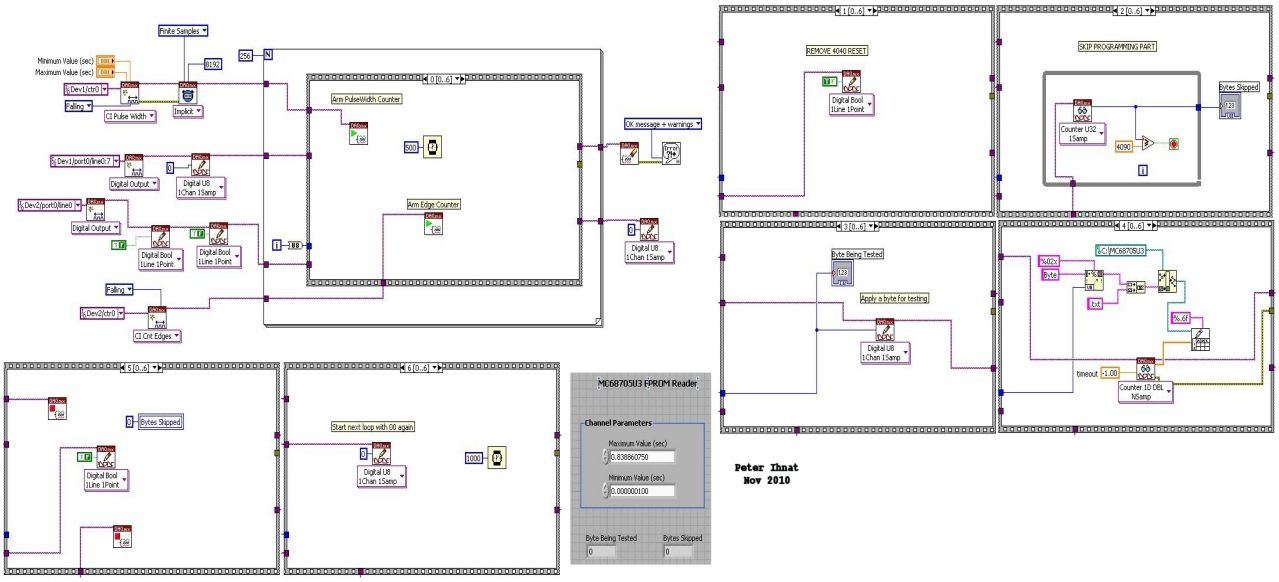




Basically:

1. I removed the EPROM and 4040 from the Motorola Programmer.
2. I connected a +5/+12V power supply to the Programmer. The +5V powers the micro. The +12V is used instead of the +26V. This supplies just under 12V to the TIMER input (pin 8) and produces +5V for pin 7 (Vpp).
   VERY IMPORTANT: don't operate S2 on the Programmer – leave it in its default position so only +5V is applied to pin 7. If you apply a higher voltage then you might overwrite the micro's EPROM.
3. I wired the RESET switch to P1.0 on the USB6009. This way I could start the programming/verify procedure using LabView.
4. I wired the external EPROM's data lines to Port0 on the USB6009.
5. I wired the CLOCK signal from pin 10 of the 4040 socket to the PFI8 input on the BNC110 (which goes to the PCI6023E) and also to the PFI0 input on the USB6009. I used the PCI6023E to measure the time between falling edges of the CLOCK and I used the USB6009 to count the number of CLOCK pulses so I knew when I'd passed the programming part of the procedure.

I modified the LabView example called “Meas Pulse Width-Buffered-Finite.vi”. I used the default max & min values and set the number of counts to 8192 (remember the Bootstrap ROM runs through the EPROM locations twice and 4096*2 = 8192).



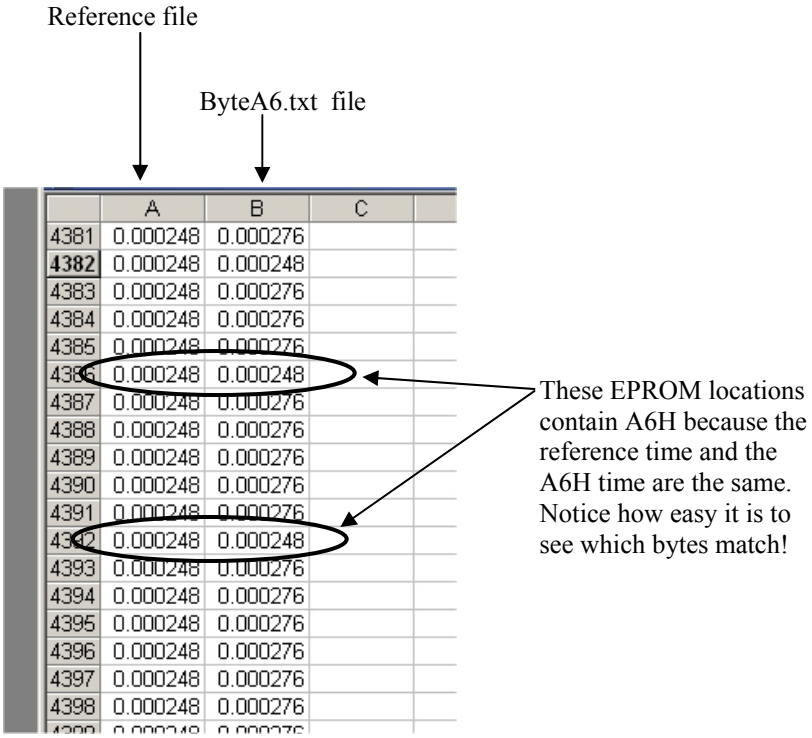
The above program does the following:

1. apply RESET to the micro.
2. apply 00 to the micro (this is to speed up the programming part of the procedure since the micro doesn’t program zeros).
3. arm the PCI6023E and USB6009 counters.
4. remove RESET to the micro. The Bootstrap ROM starts the program/verify procedure.
5. in a loop, check the count in the USB6009 counter. When it reaches around 4090 apply the byte you want to test to the micro.
6. when the 8192 periods have been collected write them to a file. The name of the file reflects which byte was tested.
7. apply RESET to the micro, increment the byte to be tested, wait 1 second then go to step 2. Loop until all 256 bytes have been checked.

This procedure took about 15 minutes to run and I ended up with 256 files named Byte00.txt, Byte01.txt through to ByteFF.txt. Each file had 8192 values which are the times between the falling edges of all the CLOCK pulses.

That’s not all. I also needed a reference file. This is because not all pulse timings which verify correctly are the same length. For example when the Bootstrap code increments the address pointer to the internal EPROM it has to increment the upper byte after every 256 increments of the lower byte. So every 256th clock pulse is different to the others. The simplest way to handle this was to create a reference file by having every byte verify correctly. For this I plugged in a blank micro (remember, an erased micro has 00 in all EPROM locations) and performed the procedure just once with a test byte value of 00. This time the ‘verified’ LED on the Programmer lit because all bytes verified correctly. Then all I did was compare each of the 8192 values in the reference file with each of the 256 ‘byte’ files to re-create the EPROM contents.

Here's an example of part of a reference file loaded into column A in Excel and the adjacent column loaded with the file ByteA6.txt. It doesn't matter what the units are as long as there's a difference between verified and non-verified bytes.



	A	B	C
4381	0.000248	0.000276	
4382	0.000248	0.000248	
4383	0.000248	0.000276	
4384	0.000248	0.000276	
4385	0.000248	0.000276	
4386	0.000248	0.000248	
4387	0.000248	0.000276	
4388	0.000248	0.000276	
4389	0.000248	0.000276	
4390	0.000248	0.000276	
4391	0.000248	0.000276	
4392	0.000248	0.000248	
4393	0.000248	0.000276	
4394	0.000248	0.000276	
4395	0.000248	0.000276	
4396	0.000248	0.000276	
4397	0.000248	0.000276	
4398	0.000248	0.000276	

So to automate the process of re-creating the EPROM I wrote another LabView program. I could have done it in Excel but I chose LabView instead.
My program did the following:

1. initialize a single dimensional array of size 8192 (called the output array)
2. read the reference file
3. read the 00 file. Compare each value in the reference file to the corresponding value in the 00 file. Each value which is the same means that location must have a 00 in it. So put 00 in the corresponding position in the output array.
4. do this for the remaining 255 files. The output array fills byte by byte.
5. remove the first 4224 values from the output array. The array now holds the contents of the MC68705 internal EPROM (minus the last byte).
   Note that the non-EPROM section of 185 bytes near the end can be ignored.

The procedure I just described did what I wanted it to do only for my immediate application so it may or may not suit what you want to do.
If you use any of the information you do so at your own risk.

Good luck with your experimentation.
Peter