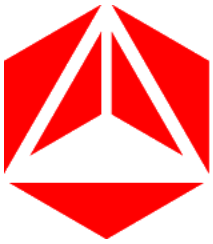
# Obviously a Major Malfunction...

Random musings that would otherwise have no means of escape from my head.

## I am...

**Adam "Major Malfunction" Laurie**
London, United Kingdom
A White Hat Hacker
View my complete profile



Director at Aperture Labs Ltd.

## RFIDIOt

Author of RFIDIOt, a python library for RFID

POC at London DEFCON - DC4420

My GitHub projects

## Sunday, 27 January 2013

### Fun with Masked ROMs - Atmel MARC4

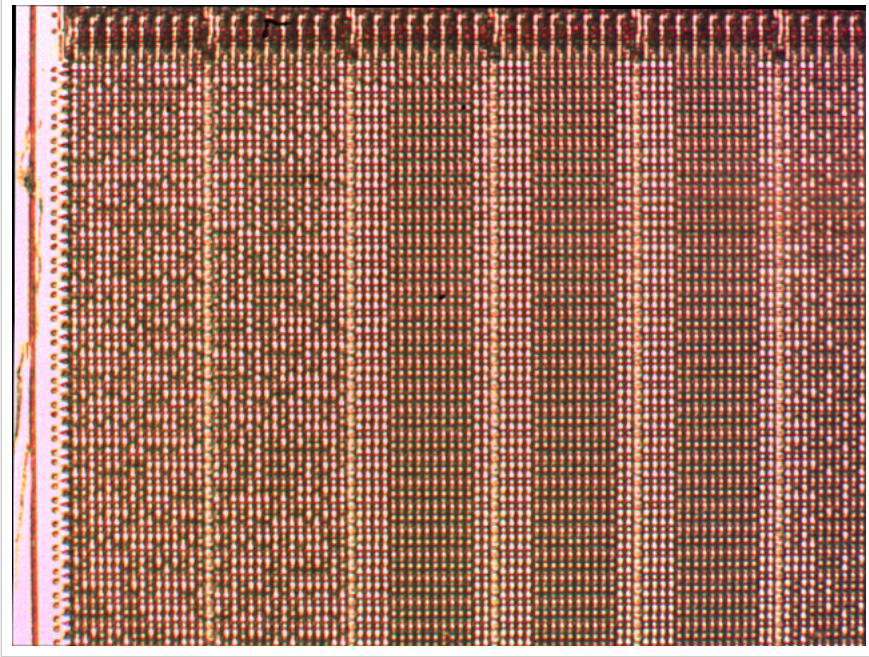
At Aperture Labs we have Code Monkeys and Chip Monkeys. Chip Monkeys do the dangerous stuff, while we Code Monkeys sit in our nice safe offices playing with bits and bytes and other things that can't hurt us...

Now, you're probably wondering what could possible be 'dangerous' in an IT Security business, and that's a perfectly fair question. How about boiling nitric acid for a start? Or fiddling with circuits that are directly connected to mains electricity? Exactly. Dangerous!

So when head Chip Monkey Zac Franken says to me: "Here are some pretty pictures of chips that I've dissolved in some lovely boiling nitric acid", I am quite happy to be a Code Monkey and to leave the nasty smelly dissolvy stuff to him... :)

You might ask why you would want to dissolve chips in the first place? Usually, it's because you are trying to do something like reset a fuse to allow reading/writing of protected areas or probe a data track to observe data being processed by the chip, or even trying to figure out the actual logic of a proprietary chip by viewing and reverse engineering it's construction.

In our case it's a combination of things, but the primary target at this stage is the program code that is stored in Masked ROM. The chip itself is using a known architecture and a published assembly language, so the only reverse engineering required is to recover the actual instructions stored in the ROM. As we can see from the picture below, this should be relatively easy as 'data' is clearly discernible:

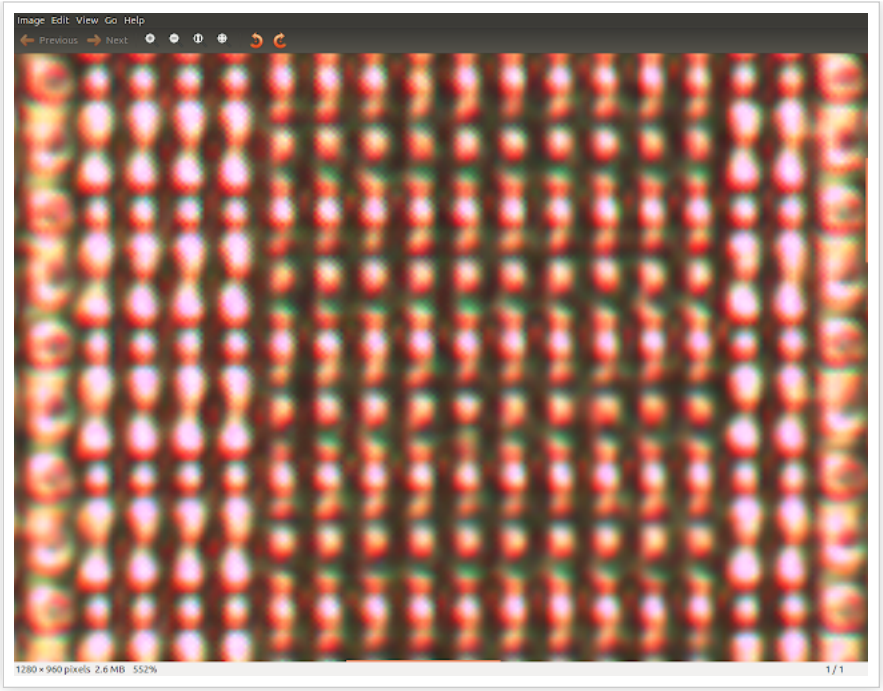


If we zoom in we can see what looks like blobs of solder connecting vias, and we can guess that the presence of a blob represents a '0' or a '1' and the absence vice-versa:

Aperture Labs github projects

DEFCON
Quartermaster

Google code projects (libnfc, pynfc, rfcat)



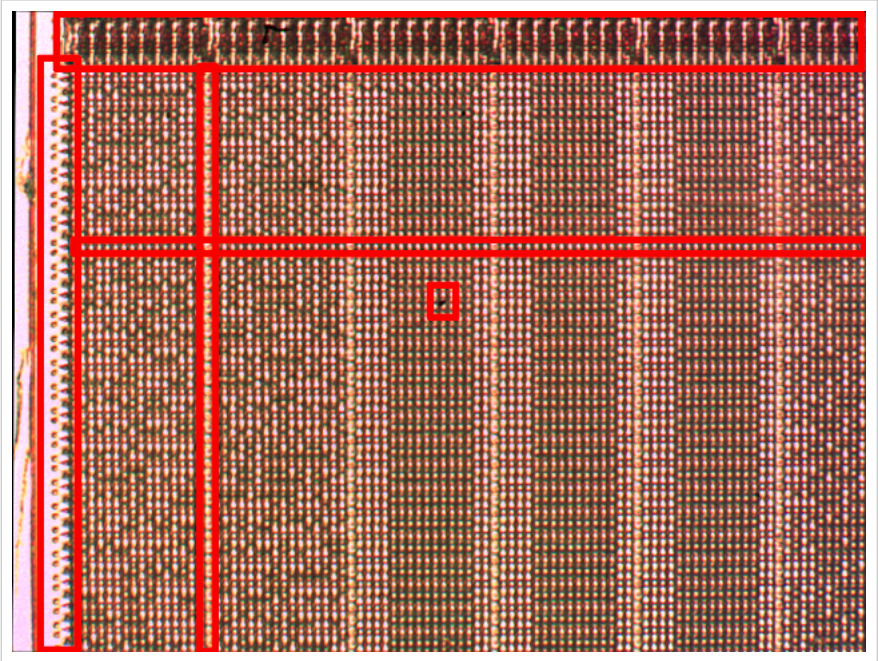

Should be pretty easy to read then. All I need is some pattern recognition software and we're done! Well, that's the theory anyway. The obvious first candidate was degate. This is a very cool piece of software written by Martin Schobert for doing exactly this kind of thing - reverse engineering chips from images. However, after playing around with it I couldn't find any obvious way to get it to read masked ROM. I thought I was being thick, so I contacted Martin and he confirmed:

*"I know from a professional chip hacker that he uses Fiji to detect inter-layer connections in the mask ROM. These vias are circle-shaped and can be detected with the Fiji GUI by just clicking some menu items, which results in a list of coordinates. I am interested to integrate this functionality in degate, but never had a practical use case for this. Anyway, vias can already be detected by Degate. Therefore, the user draws some vias into the image and thus marks their position. Remaing vias are detected by using "via matching", which is based on a cross-correlation with the marked templates. After via matching is finished, the coordinates are written into the logic model XML file. From this, the coordinates can be extracted easily and have to be processed into bytes with some kind of script. But it has to be written."*

Hmmm... OK, so whatever the approach I'm going to end up writing code to extract the data so I figured that I may as well try to do the whole thing in one go - i.e. detect the data bits directly from the image and dump them to binary file for later analysis. Well, at least we now have a plan! :)

As anyone who knows me will already know, I am a big python fan, and if you're a regular at DC4420 you'll also remember that python has some excellent image processing tools which I previously used in our Smoke Detector Random Number Generator project, in the form of OpenCV.

This really shouldn't be too hard - read in an image, detect patterns of rows and columns and extract the points. True in principal, but if you go back and look at the original image it quickly becomes obvious that there is a lot of extraneous data:



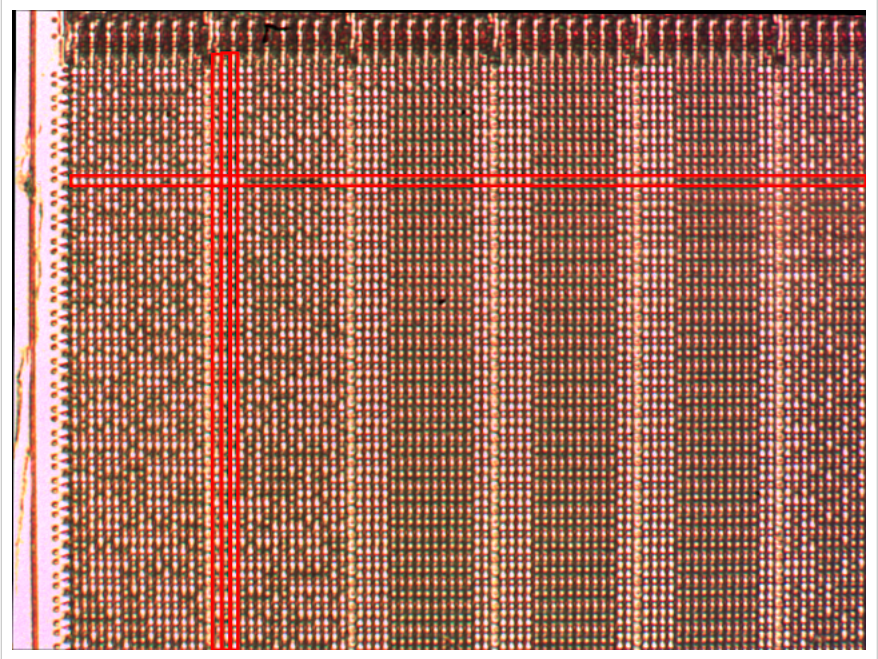
So whatever we do, we're going to end up having to either do a lot of manual intervening to prevent our program from detecting data that we're simply not interested in, or so specific in it's pattern recognition that it's likely to get a lot of read errors due to missing data that was 'corrupted' by a dark spot, poor lighting etc., etc.

I should qualify this a bit: in an ideal world, we would be looking at a perfectly clean image with absolutely uniform lighting and every feature would be easily recognisable. This is achievable if you have something like a university lab at your disposal (e.g. visual6502.org), or you are a legend like Chris Tarnovsky who has spent years building up a multi-million dollar lab, but for us mere mortals we are working with much less sophisticated kit! (I'll leave Zac to expand on that subject). We therefore need to make some compromises and make the most of what we've got.

On that basis, I decided that instead of trying to automate the whole process, I would ***semi-automate*** it. In other words, I'd put the effort into making a tool that enables selection of 'interesting' data rather that trying to recognise it automatically.

And so **rompar** was born...

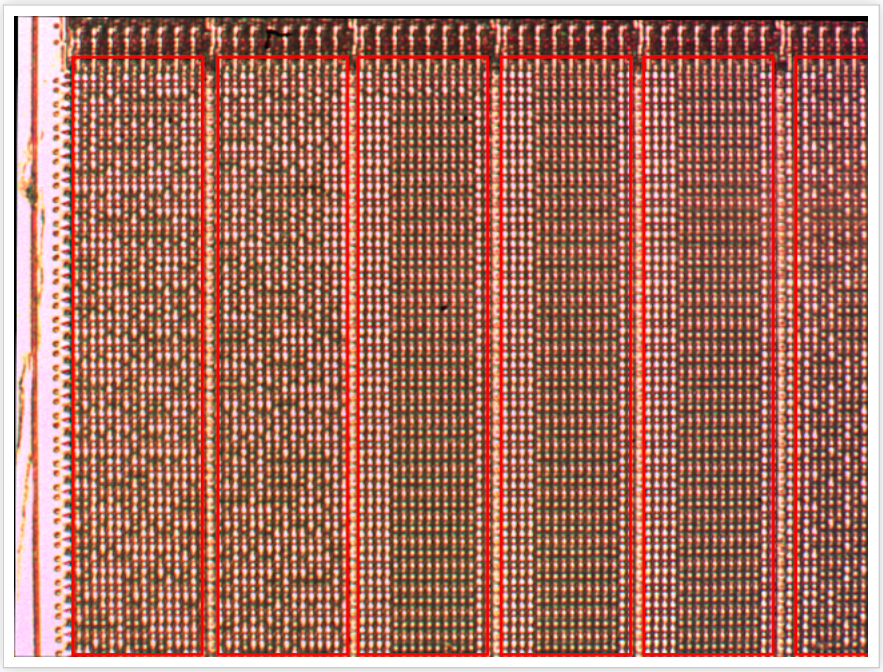
The basic idea was to create a grid that layers over the image and where the lines intersect we have a point of interest which will easier to 'detect':



The full tutorial on rompar is on it's documentation page, but for this process, the steps were as follows:

First we must decide what we think the layout is. The full chip has 10 columns, but looking at the image available via the

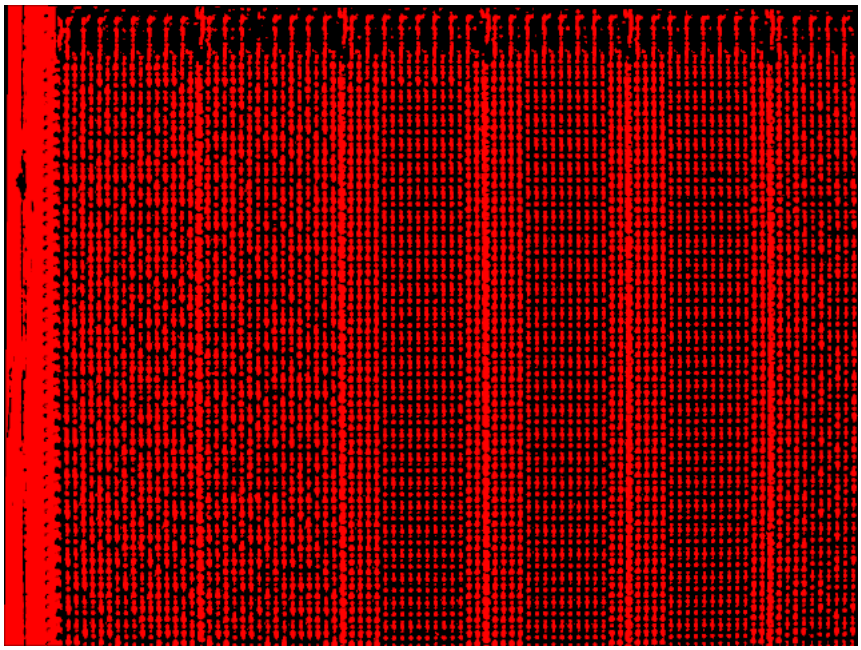
microscope, it would appear that we can see 5 and a half columns of 16 bits in a column, each seperated by a column of vias:



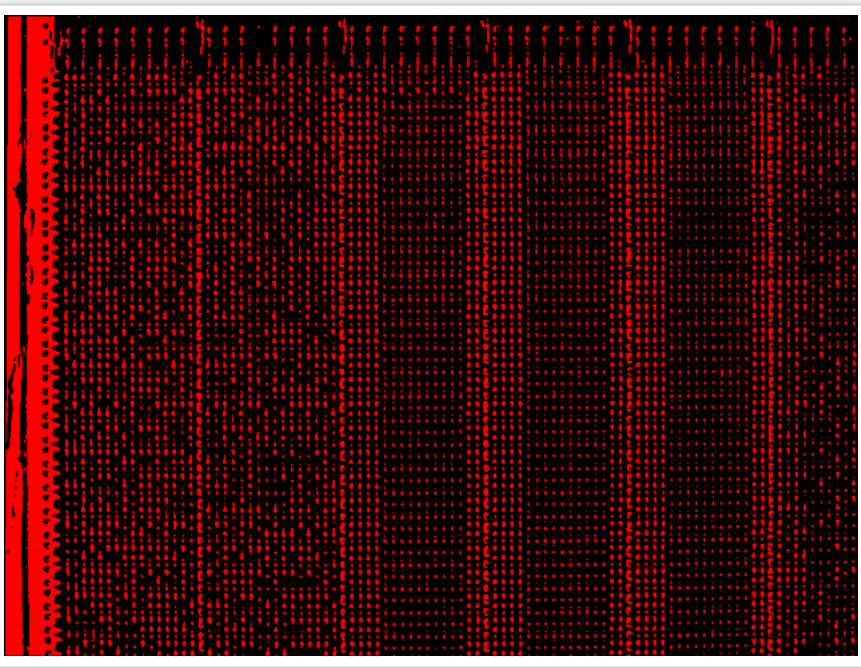
The horizontal lines are basically single rows of bits seperated by lines of vias. If we look a little closer we can see that the gap between the first and second row is a lot larger than the gap between the third and fourth, so as far as a repeating pattern goes, at the very least we need to include two rows of bits and two rows of vias. As will become clear later on, the bigger the pattern group we create, the less 'work' we will need to do to make our grid, so we can increase this figure to something bigger that still fits in a reasonable multiple into the image. In this case I have about 48 rows of bits so I chose 16 which will give me 3 'chunks' per image.


So now we load the image into rompar, tell it we're processing a 16x16 grid and apply a threshold filter:

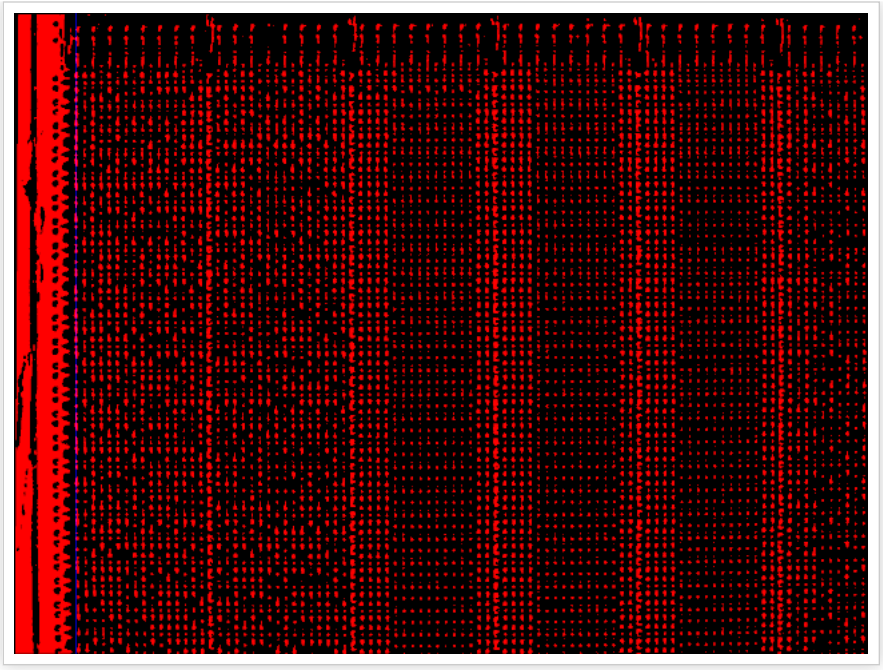
```
$ rompar.py chip.png 16 16
```



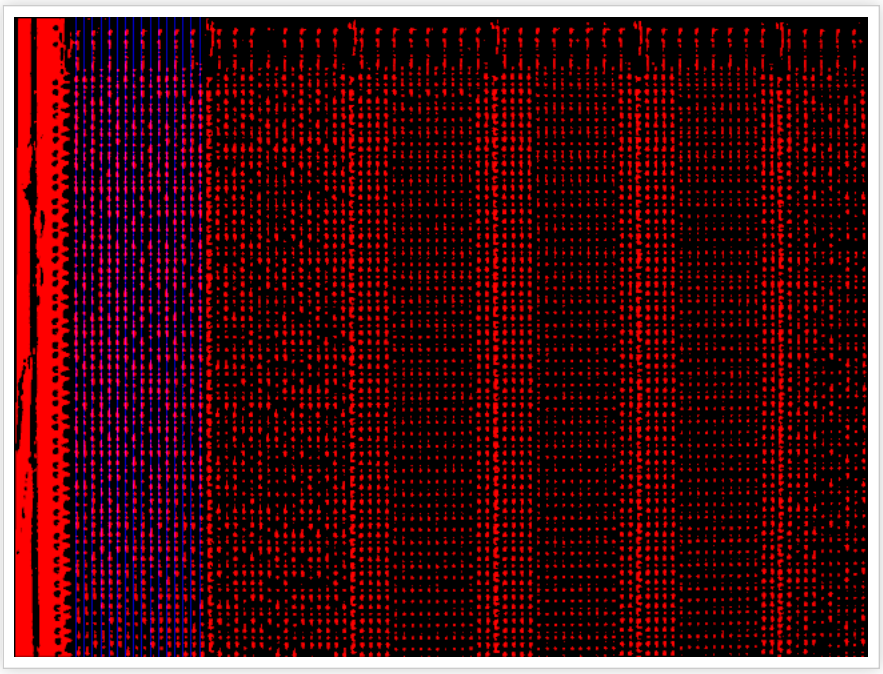
Adjust the threshold until we've got very clear single points:



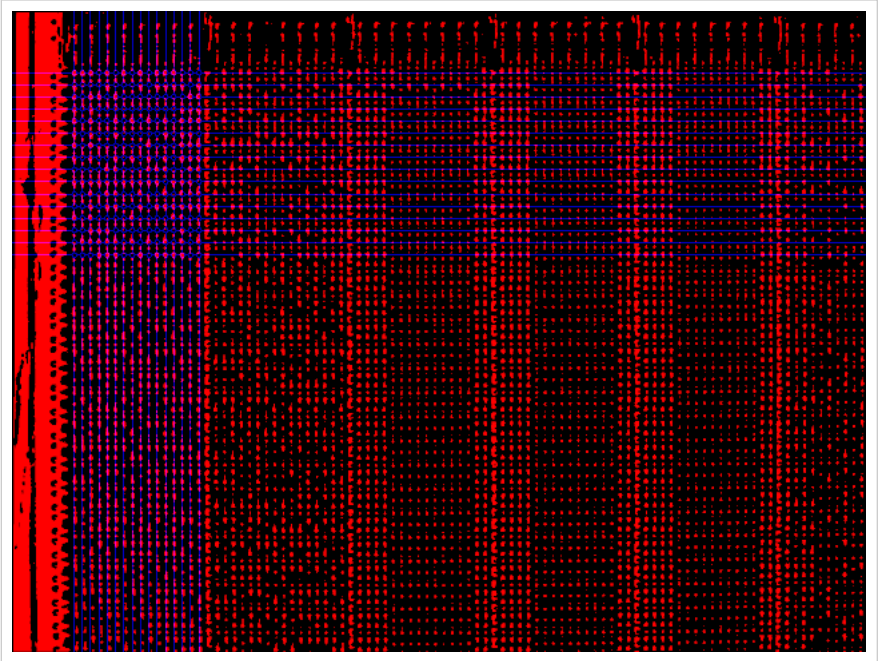
Left-Click on the first bit of the first column of data. This will create a vertical gridline:



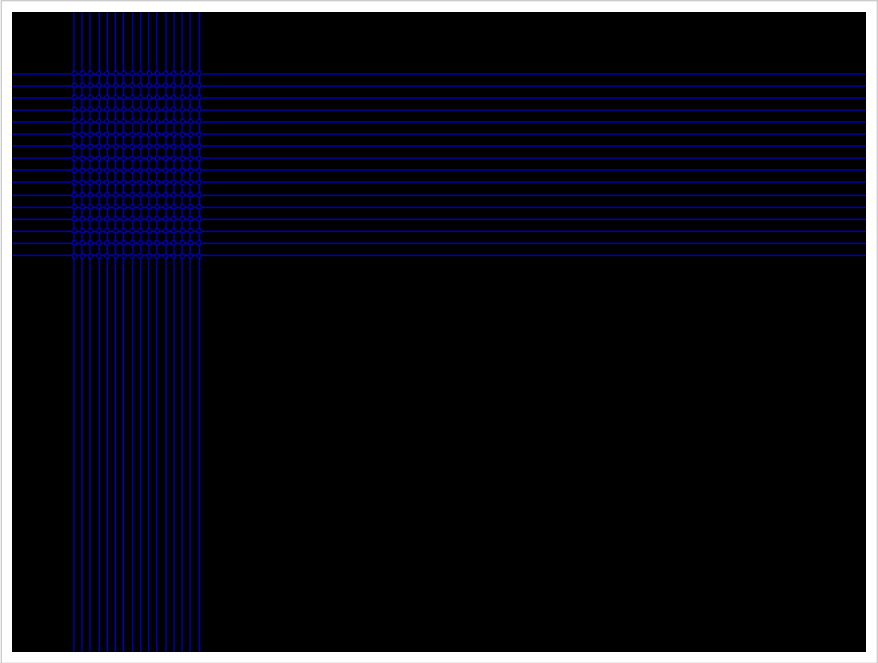
Left-Click on the last bit of the first column of data. This will create the remaining gridlines for that column:



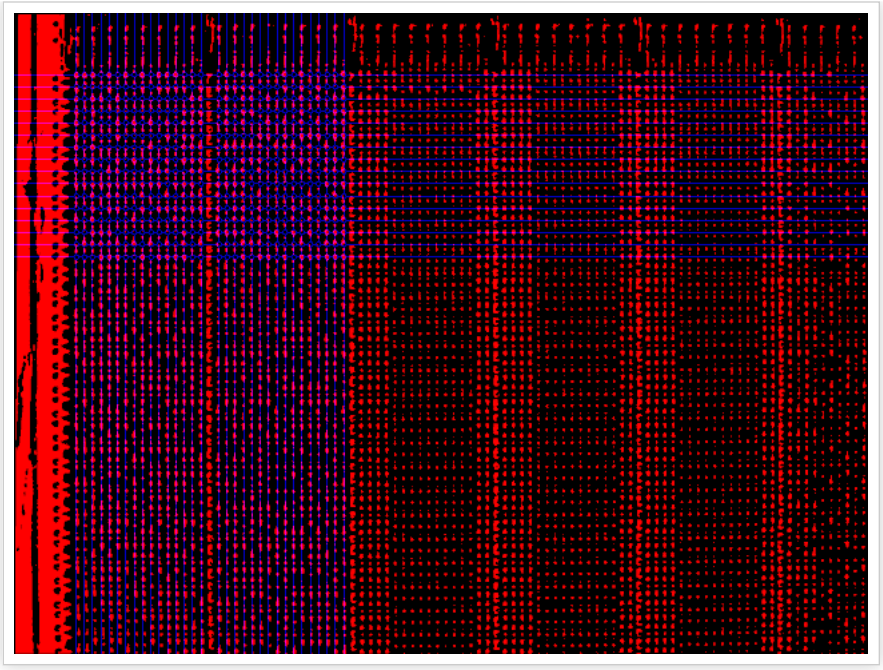
In the same way we can Right-Click on the first and last bits of the first group of rows and this will create our first set of intersections:



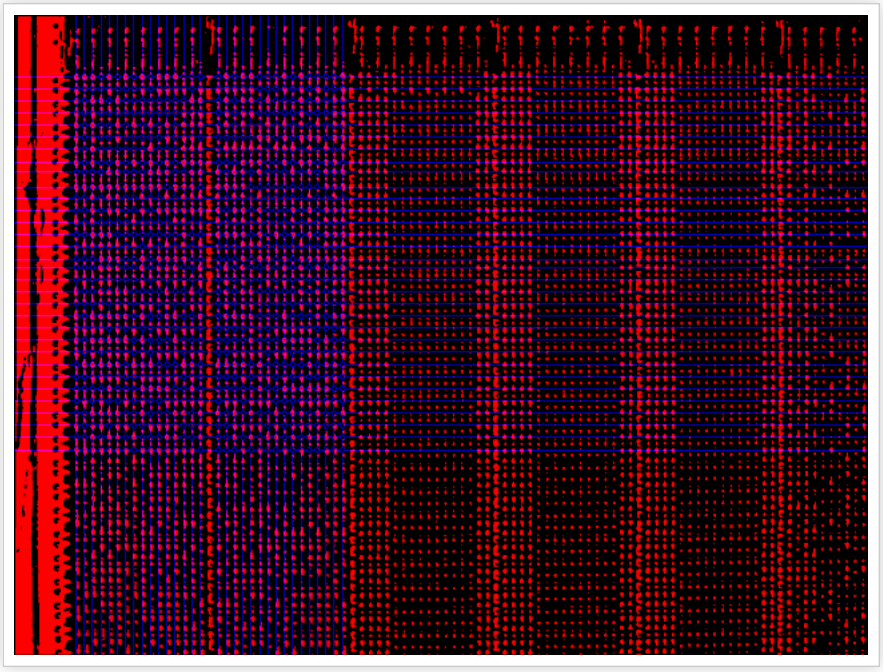
This is easier to see of we blank out the ROM image:



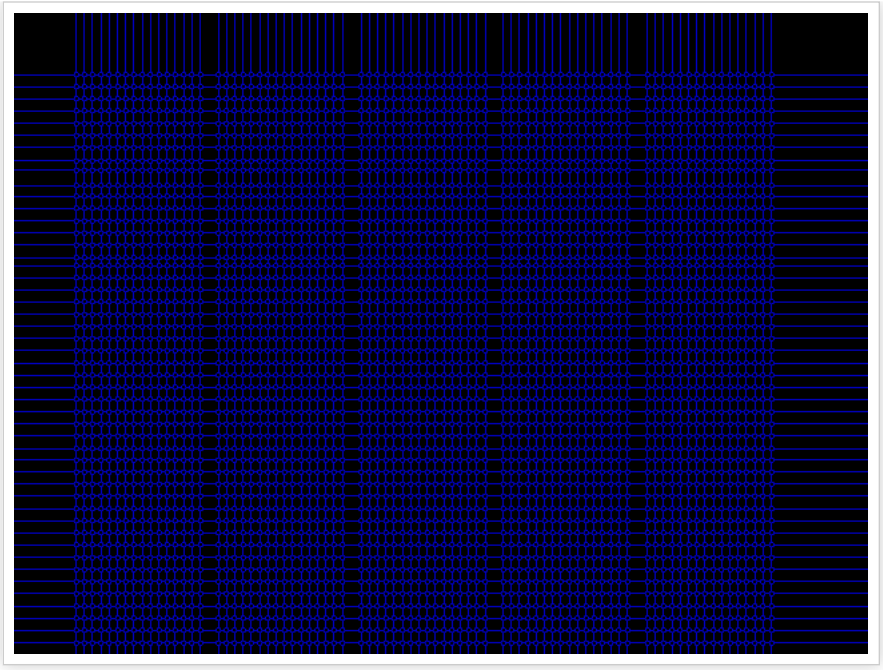
And now this is where the semi-automation starts to kick in. Having created our 16x16 intersection group, we can repeat it by simply Left-Clicking at the start of a new column group:



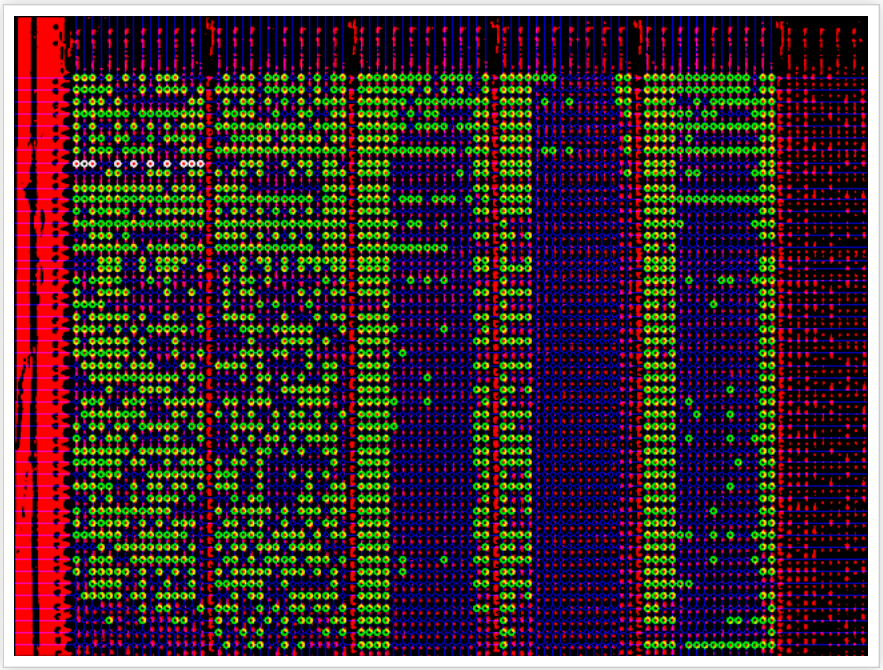
Or Right-Clicking at the start of a new row group:



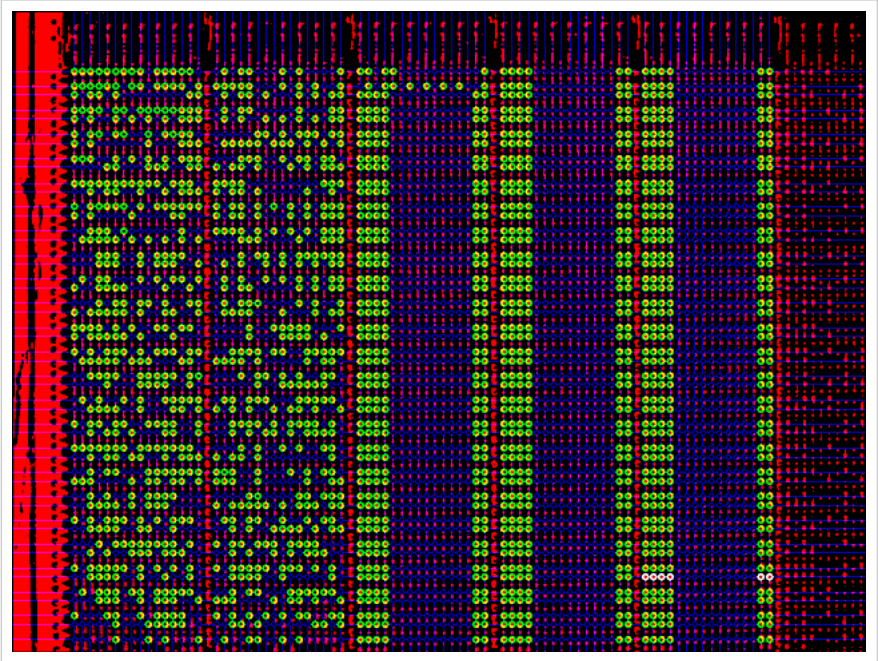
Four mouse clicks later and we've 'gridded' the whole thing:



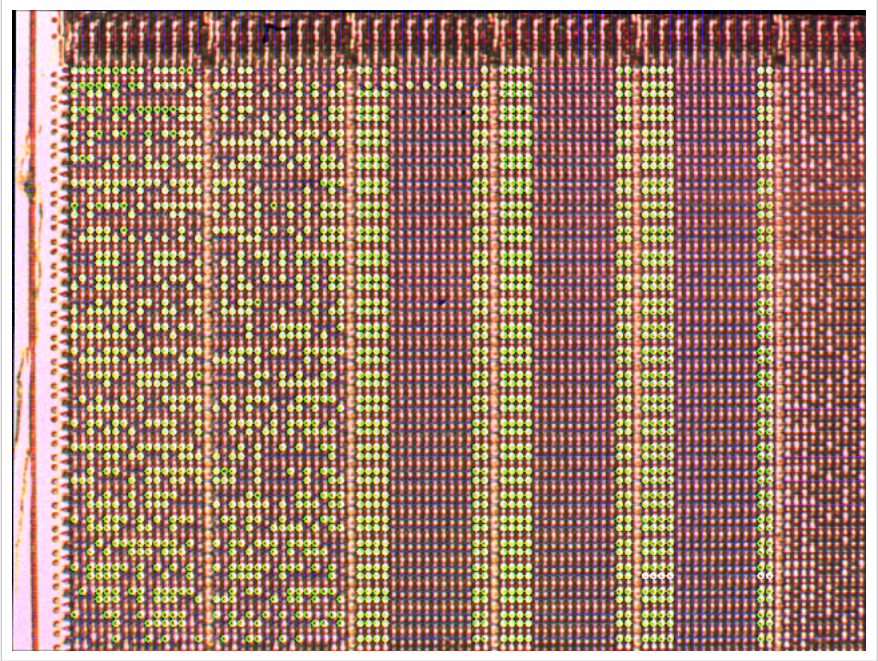
Now we simply hit 'r' for 'read' and the presence or absence of a bright spot within an intersection will be flagged:



Here we can see that we're getting quite a few bit errors as the data in the 'empty' columns three four and five is varying quite a lot, whereas it is clear from the original image that these columns should mostly contain the same values. This is due to small mis-allignements of the grid, and we can adjust those by selecting rows or columns and nudging them left/right or up/down. In the image above the eighth row in the first column is highlighted in white to show it's selected. After fine tuning, our read looks like this:



Much better! There are a few other features that make it easy to check/adjust the data, like being able to toggle the view between the thresholded image and the original, so individual bits can be inspected and/or set:



Magic. So after only a few minutes we have some 'real' data to play with. But now we have to answer some fundamental questions:

- What order are the bits in - i.e. LSB/MSB?
- What order are the bytes in - read all the rows and then the columns, or vice-versa, or something else?
- What is a '0' and a '1' - the presence or the absence of a blob?
- etc.

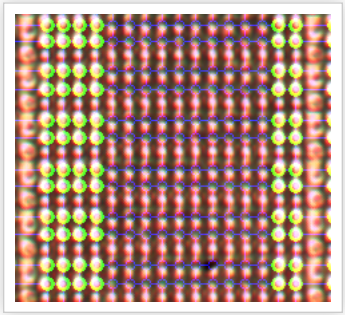
We really have no way of telling, and in fact, if you look around, you'll find examples of pretty much every variation you can think of. Here, Travis Goodspeed describes a Masked ROM which has 16 columns of 8 bits each, but each 16 bit word is made up by taking a single bit from each of the 16 columns. In our case we have a total of 10 columns, so this is an unlikely scenario. There are other examples on siliconpr0n.org.

However, in this case we are lucky: it's a known architecture with a published instruction set, so we should be able to simply compare the bits/bytes and juggle them around until they make sense.

Step one was to figure out what is a '0' and what is a '1'. This should be fairly straightforward as again we are lucky: there is a large section of 'empty' ROM which should have a known/fixed value. The chip is an Atmel MARC4, and we can simply download the programming manual which includes the instruction set. An example project also shows that 'empty' space in a

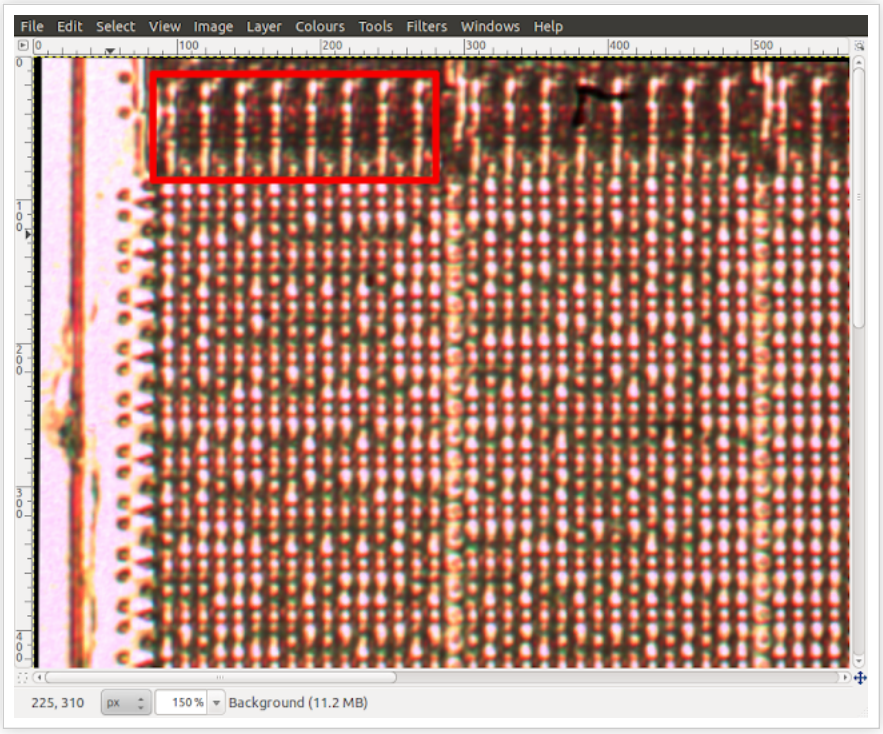
program is likely to contain the HEX value C1, which, according to the manual, is the instruction: 'SCALL $RESET' (Unconditional short CALL to address $008). This seems eminently sensible as any wayward code will end up hitting one of these and resetting the chip.

So lets look at our 'empty' area:



But that doesn't look right. C1 is the eight bit pattern: 11000001 in binary, so 16 bits of repeating C1s would be 1100000111000001, whereas we have a pattern that is either 1111000000000011 or 0000111111111111100 depending how we interpret the bits.

As I said though, anything goes in bit placement, and in this case the clue is in the top of the image:




Here we can see that dropping into our group of 16 columns are 8 traces. Could this mean that we are actually looking at two sets of 8 bits interleaved?

If we interpret our 16 bit pattern on that basis, it looks like this:

```
1111000000000011
----------------
1 1 0 0 0 0 0 1
 1 1 0 0 0 0 0 1
```

Bingo! We have our 11000001 == C1. Sweet! Now I have the option to read as 16 bits and de-interleave in code, or create pairs of 8 bit grids that read every other bit. I opted for the latter just to keep things tidy.

So far, so good: we now know the overall bit arrangement and order within a byte (rompar reads MSB from the left by default). Now we need to answer the question of what order we read the bytes in. Again, the empty areas give us a big clue. If we read them in anything other than column order they will no longer be one large group of C1s, but small bursts of C1s interspersed with other data, which wouldn't make sense, so column order it is. The only remaining question then is where does the program start: top left, bottom left, top right or bottom right?

In order to make figuring this out easier, it would be helpful to be able to read program code rather than HEX values, and to achieve this we need to convert the HEX instructions into their human readable form as per the manual. I was unable to find a MARC4 disassembler on the net, so I ended up writing that too. This is not as tricky as it sounds - there are only a fairly small set of commands, and they convert rigidly into a stream of HEX, so reversing the process and converting the HEX back into commands and corresponding arguments is actually pretty trivial. The hardest/most tedious part is cutting & pasting the descriptions and creating the layout so it's nice and readable.

There is of course another advantage to doing it this way: at some point I'm going to have to wade through the extracted code and figure out what it's doing, and there's nothing quite like writing a disassembler to ensure you really understand how a language works!

And so **marc4dasm** was born...

It basically does four things:

- Convert instructions and arguments to human readable form
- Find and name all called addresses
- Find and name all referenced variable addresses
- Find and name areas of code that are never called (orphans)

Given a binary file, it will produce output like this:

```
\
\
\       Atmel/MARC4(TMEB893)KIT/example/example.hex
\
\
\       ROM ADDRESS       LABEL
\
\       $000              $AUTOSLEEP
\       $008              $RESET
\       $011              LABEL_000
\       $028              ORPHAN_000
\       $030              LABEL_001
\       $040              INTERRUPT_0
\       $048              LABEL_002
\       $080              INTERRUPT_1
\       $0C0              INTERRUPT_2
\       $100              INTERRUPT_3
\       $140              INTERRUPT_4
\       $180              INTERRUPT_5
\       $1C0              INTERRUPT_6
\       $1E0              INTERRUPT_7
\       $1F8              LABEL_003
\
\
\
\       RAM VARIABLE      LABEL
\
\       $03               VAR_02
\       $35               VAR_00
\       $FC               VAR_01
\
\
\
\
0000
0000        ORIGIN $000
0000        : $AUTOSLEEP
0000 7C               NOP                                 \ No operation
0001 0F               SLEEP                               \ CPU in 'sleep mode', interrupts enabled
0002 19               SET_BCF                             \ Set Branch and Carry flag
0003 80               SBRA $AUTOSLEEP                     \ Conditional short branch in page ($000)
0004 C1               SCALL $RESET                        \ Unconditional short CALL ($008)
0005 C1               SCALL $RESET                        \ Unconditional short CALL ($008)
0006 C1               SCALL $RESET                        \ Unconditional short CALL ($008)
0007 C1               SCALL $RESET                        \ Unconditional short CALL ($008)
0008
0008        ORIGIN $008
0008        : $RESET
```

```
0008 78 35            >SP VAR_00                        \ Set Expression Stack Pointer
000A 79 FC            >RP VAR_01                        \ Set return Stack Pointer direct
000C 6A               LIT_A                             \ Push literal/constant $A onto TOS
000D 63               LIT_3                             \ Push literal/constant $3 onto TOS
000E 1F               OUT                               \ Write TOS to 4-bit I/O port
000F 65               LIT_5                             \ Push literal/constant $5 onto TOS
0010 22               >R                                \ Move (loop) index onto Return Stack
0011
0011        ORIGIN $011
0011        : LABEL_000
0011 7C               NOP                               \ No operation
0012 1C               DECR                              \ Decrement index on return stack
0013 91               SBRA LABEL_000                    \ Conditional short branch in page ($011)
0014 2F               DROPR                             \ Remove one entry from the Return Stack
0015 6A               LIT_A                             \ Push literal/constant $A onto TOS
0016 63               LIT_3                             \ Push literal/constant $3 onto TOS
0017 C6               SCALL LABEL_001                   \ Unconditional short CALL ($030)
0018 C9               SCALL LABEL_002                   \ Unconditional short CALL ($048)
0019 60               LIT_0                             \ Push literal/constant $0 onto TOS
001A 3F 03            [>Y]! VAR_02                      \ Direct store into RAM addressed by the Y register
001C 60               LIT_0                             \ Push literal/constant $0 onto TOS
001D 62               LIT_2                             \ Push literal/constant $2 onto TOS
001E 1F               OUT                               \ Write TOS to 4-bit I/O port
001F 60               LIT_0                             \ Push literal/constant $0 onto TOS
0020 62               LIT_2                             \ Push literal/constant $2 onto TOS
0021 C6               SCALL LABEL_001                   \ Unconditional short CALL ($030)
0022 1D               RTI                               \ Return from interrupt routine; enable all interrupts
0023 C1               SCALL $RESET                      \ Unconditional short CALL ($008)
0024 C1               SCALL $RESET                      \ Unconditional short CALL ($008)
0025 C1               SCALL $RESET                      \ Unconditional short CALL ($008)
0026 C1               SCALL $RESET                      \ Unconditional short CALL ($008)
0027 C1               SCALL $RESET                      \ Unconditional short CALL ($008)
0028
0028        ORIGIN $028
0028        : ORPHAN_000
0028 7C               NOP                               \ No operation
0029 7C               NOP                               \ No operation
002A 25               EXIT                              \ Return from subroutine (';')
002B C1               SCALL $RESET                      \ Unconditional short CALL ($008)
002C C1               SCALL $RESET                      \ Unconditional short CALL ($008)
002D C1               SCALL $RESET                      \ Unconditional short CALL ($008)
002E C1               SCALL $RESET                      \ Unconditional short CALL ($008)
```

which is the start of Atmel's example program from their development kit (no longer available to purchase, but easy enough to find on the net). This is handy as I could cross check my output against their example listing, so I'm reasonably confident that my disassembler is working correctly.

The output can then be edited to give the variables and addresses meaningful names (if they can be figured out). You'll notice there are some 'meaningful' names already, namely $AUTOSLEEP, $RESET and $INTERRUPT_0 to $INTERRUPT_7, and this is because they are standard routines that must exist at certain locations in ROM. This will come in very handy later!

Reading the first set of interleaved bits from the chip's top left gives us:

```
FE AE C1 C1 C1
C1 C1 C1 25 FF
69 6F C2 60 60
2E 2E 2E B1 19
FA C5 FA FF 69
6F 2D 1B 6A D6
62 FF 69 67 C6
66 64 FF 6A 1F
6A FF 69 6F 2E
AE 07
```

and running that through the diassember, we get:

```
$ marc4dasm.py a1-0
\
\
```

```
\       a1-0
\
\
\       ROM ADDRESS       LABEL
\
\       $000              $AUTOSLEEP
\       $008              $RESET
\       $009              ORPHAN_000
\       $010              LABEL_003
\       $028              LABEL_006
\       $02E              LABEL_001
\       $030              LABEL_008
\       $031              LABEL_004
\       $040              INTERRUPT_0
\       $080              INTERRUPT_1
\       $0B0              LABEL_007
\       $0C0              INTERRUPT_2
\       $100              INTERRUPT_3
\       $140              INTERRUPT_4
\       $180              INTERRUPT_5
\       $1C0              INTERRUPT_6
\       $1D0              LABEL_005
\       $1E0              INTERRUPT_7
\       $1F0              LABEL_000
\       $1F8              LABEL_002
\
\
\       RAM VARIABLE      LABEL
\
\
\
\
\
0000
0000        ORIGIN $000
0000        : $AUTOSLEEP
0000 FE               SCALL LABEL_000                   \ Unconditional short CALL ($1F0)
0001 AE               SBRA LABEL_001                    \ Conditional short branch in page ($02E)
0002 C1               SCALL $RESET                      \ Unconditional short CALL ($008)
0003 C1               SCALL $RESET                      \ Unconditional short CALL ($008)
0004 C1               SCALL $RESET                      \ Unconditional short CALL ($008)
0005 C1               SCALL $RESET                      \ Unconditional short CALL ($008)
0006 C1               SCALL $RESET                      \ Unconditional short CALL ($008)
0007 C1               SCALL $RESET                      \ Unconditional short CALL ($008)
0008
0008        ORIGIN $008
0008        : $RESET
0008 25               EXIT                              \ Return from subroutine (';')
0009
0009        ORIGIN $009
0009        : ORPHAN_000
0009 FF               SCALL LABEL_002                   \ Unconditional short CALL ($1F8)
000A 69               LIT_9                             \ Push literal/constant $9 onto TOS
000B 6F               LIT_F                             \ Push literal/constant $F onto TOS
000C C2               SCALL LABEL_003                   \ Unconditional short CALL ($010)
000D 60               LIT_0                             \ Push literal/constant $0 onto TOS
000E 60               LIT_0                             \ Push literal/constant $0 onto TOS
000F 2E               DROP                              \ Remove TOS digit from the Expression Stack
0010
0010        ORIGIN $010
0010        : LABEL_003
0010 2E               DROP                              \ Remove TOS digit from the Expression Stack
0011 2E               DROP                              \ Remove TOS digit from the Expression Stack
0012 B1               SBRA LABEL_004                    \ Conditional short branch in page ($031)
0013 19               SET_BCF                           \ Set Branch and Carry flag
0014 FA               SCALL LABEL_005                   \ Unconditional short CALL ($1D0)
0015 C5               SCALL LABEL_006                   \ Unconditional short CALL ($028)
0016 FA               SCALL LABEL_005                   \ Unconditional short CALL ($1D0)
0017 FF               SCALL LABEL_002                   \ Unconditional short CALL ($1F8)
```

```
0018 69               LIT_9                             \ Push literal/constant $9 onto TOS
0019 6F               LIT_F                             \ Push literal/constant $F onto TOS
001A 2D               DUP                               \ Duplicate the TOS digit
001B 1B               IN                                \ Read 4-bit I/O port to TOS
001C 6A               LIT_A                             \ Push literal/constant $A onto TOS
001D D6               SCALL LABEL_007                   \ Unconditional short CALL ($0B0)
001E 62               LIT_2                             \ Push literal/constant $2 onto TOS
001F FF               SCALL LABEL_002                   \ Unconditional short CALL ($1F8)
0020 69               LIT_9                             \ Push literal/constant $9 onto TOS
0021 67               LIT_7                             \ Push literal/constant $7 onto TOS
0022 C6               SCALL LABEL_008                   \ Unconditional short CALL ($030)
0023 66               LIT_6                             \ Push literal/constant $6 onto TOS
0024 64               LIT_4                             \ Push literal/constant $4 onto TOS
0025 FF               SCALL LABEL_002                   \ Unconditional short CALL ($1F8)
0026 6A               LIT_A                             \ Push literal/constant $A onto TOS
0027 1F               OUT                               \ Write TOS to 4-bit I/O port
0028
0028        ORIGIN $028
0028        : LABEL_006
0028 6A               LIT_A                             \ Push literal/constant $A onto TOS
0029 FF               SCALL LABEL_002                   \ Unconditional short CALL ($1F8)
002A 69               LIT_9                             \ Push literal/constant $9 onto TOS
002B 6F               LIT_F                             \ Push literal/constant $F onto TOS
002C 2E               DROP                              \ Remove TOS digit from the Expression Stack
```

We can see straight away that this is not the start of the program, as we are missing the $AUTOSLEEP routine, so let's try reading from the bottom right instead:

```
$ marc4dasm.py b6-4
\
\
\       b6-4
\
\
\       ROM ADDRESS       LABEL
\
\       $000              $AUTOSLEEP
\       $001              ORPHAN_000
\       $002              ORPHAN_001
\       $008              $RESET
\       $00C              ORPHAN_002
\       $013              ORPHAN_003
\       $014              ORPHAN_004
\       $019              LABEL_000
\       $01D              ORPHAN_005
\       $020              ORPHAN_006
\       $021              ORPHAN_007
\       $02B              ORPHAN_008
\       $040              INTERRUPT_0
\       $080              INTERRUPT_1
\       $0C0              INTERRUPT_2
\       $100              INTERRUPT_3
\       $140              INTERRUPT_4
\       $180              INTERRUPT_5
\       $1C0              INTERRUPT_6
\       $1E0              INTERRUPT_7
\       $3FC              LABEL_001
\
\
\
\       RAM VARIABLE      LABEL
\
\       $35               VAR_00
\       $C1               VAR_01
\
\
\
\
0000
0000        ORIGIN $000
```

```
0000        : $AUTOSLEEP
0000 C1             SCALL $RESET                    \ Unconditional short CALL ($008)
0001
0001        ORIGIN $001
0001        : ORPHAN_000
0001 25             EXIT                            \ Return from subroutine (';')
0002
0002        ORIGIN $002
0002        : ORPHAN_001
0002 1F             OUT                             \ Write TOS to 4-bit I/O port
0003 1F             OUT                             \ Write TOS to 4-bit I/O port
0004 27             OVER                            \ Push a copy of TOS-1 onto TOS
0005 26             SWAP                            \ Exchange the top 2 digits
0006 69             LIT_9                           \ Push literal/constant $9 onto TOS
0007 C1             SCALL $RESET                    \ Unconditional short CALL ($008)
0008
0008        ORIGIN $008
0008        : $RESET
0008 C1             SCALL $RESET                    \ Unconditional short CALL ($008)
0009 C1             SCALL $RESET                    \ Unconditional short CALL ($008)
000A C1             SCALL $RESET                    \ Unconditional short CALL ($008)
000B C1             SCALL $RESET                    \ Unconditional short CALL ($008)
000C
000C        ORIGIN $00C
000C        : ORPHAN_002
000C 20             TABLE                           \ Fetch an 8-bit ROM constant and performs an EXIT to Ret_PC
000D 2B             3R@                             \ Copy 3 digits from Return to Expression Stack
000E 29             3>R                             \ Move top 3 digits onto Return Stack
000F C1             SCALL $RESET                    \ Unconditional short CALL ($008)
0010 C1             SCALL $RESET                    \ Unconditional short CALL ($008)
0011 C1             SCALL $RESET                    \ Unconditional short CALL ($008)
0012 C1             SCALL $RESET                    \ Unconditional short CALL ($008)
0013
0013        ORIGIN $013
0013        : ORPHAN_003
0013 25             EXIT                            \ Return from subroutine (';')
0014
0014        ORIGIN $014
0014        : ORPHAN_004
0014 2E             DROP                            \ Remove TOS digit from the Expression Stack
0015 1B             IN                              \ Read 4-bit I/O port to TOS
0016 63             LIT_3                           \ Push literal/constant $3 onto TOS
0017 C1             SCALL $RESET                    \ Unconditional short CALL ($008)
0018 C1             SCALL $RESET                    \ Unconditional short CALL ($008)
0019
0019        ORIGIN $019
0019        : LABEL_000
0019 C1             SCALL $RESET                    \ Unconditional short CALL ($008)
001A C1             SCALL $RESET                    \ Unconditional short CALL ($008)
001B C1             SCALL $RESET                    \ Unconditional short CALL ($008)
001C C1             SCALL $RESET                    \ Unconditional short CALL ($008)
001D
001D        ORIGIN $01D
001D        : ORPHAN_005
001D 20             TABLE                           \ Fetch an 8-bit ROM constant and performs an EXIT to Ret_PC
001E 29             3>R                             \ Move top 3 digits onto Return Stack
001F C1             SCALL $RESET                    \ Unconditional short CALL ($008)
0020
0020        ORIGIN $020
0020        : ORPHAN_006
0020 1D             RTI                             \ Return from interrupt routine; enable all interrupts
0021
0021        ORIGIN $021
0021        : ORPHAN_007
0021 99             SBRA LABEL_000                  \ Conditional short branch in page ($019)
0022 43 FC          CALL LABEL_001                  \ Unconditional long CALL ($3FC)
0024 79 35          >RP VAR_00                      \ Set return Stack Pointer direct
0026 78 C1          >SP VAR_01                      \ Set Expression Stack Pointer
0028 C1             SCALL $RESET                    \ Unconditional short CALL ($008)
0029 C1             SCALL $RESET                    \ Unconditional short CALL ($008)
```

```
002A C1               SCALL $RESET                          \ Unconditional short CALL ($008)
002B
002B        ORIGIN $02B
002B        : ORPHAN_008
002B 80               SBRA $AUTOSLEEP                       \ Conditional short branch in page ($000)
002C 19               SET_BCF                               \ Set Branch and Carry flag
```

This is still not right, but remember we are looking for the start of the program to be:

```
0000 7C               NOP                                   \ No operation
0001 0F               SLEEP                                 \ CPU in 'sleep mode', interrupts enabled
0002 19               SET_BCF                               \ Set Branch and Carry flag
0003 80               SBRA $AUTOSLEEP                       \ Conditional short branch in page ($000)
```

So if the data were reversed we would expect our last instructions to be:

```
0003 80               SBRA $AUTOSLEEP                       \ Conditional short branch in page ($000)
0002 19               SET_BCF                               \ Set Branch and Carry flag
0001 0F               SLEEP                                 \ CPU in 'sleep mode', interrupts enabled
0000 7C               NOP                                   \ No operation
```

Although as the last two bytes are treated as CRC they won't be included in the listing, so we would only expect to see the first two of the above lines, which indeed we do - the last two lines of our listing are:

```
002B 80               SBRA $AUTOSLEEP                       \ Conditional short branch in page ($000)
002C 19               SET_BCF                               \ Set Branch and Carry flag
```

So this is a good candidate for the start of code. In theory that means we should now be able to simply reverse the order of bytes and we're done:

```
$ marc4dasm.py b6-4-reversed
\
\
\       b6-4-reversed
\
\
\       ROM ADDRESS       LABEL
\
\       $000              $AUTOSLEEP
\       $008              $RESET
\       $010              ORPHAN_000
\       $018              ORPHAN_001
\       $020              ORPHAN_002
\       $028              ORPHAN_003
\       $040              INTERRUPT_0
\       $080              INTERRUPT_1
\       $0C0              INTERRUPT_2
\       $100              INTERRUPT_3
\       $140              INTERRUPT_4
\       $180              INTERRUPT_5
\       $1C0              INTERRUPT_6
\       $1E0              INTERRUPT_7
\       $399              LABEL_000
\
\
\       RAM VARIABLE      LABEL
\
\       $35               VAR_00
\       $FC               VAR_01
\
\
\
\
0000
0000        ORIGIN $000
0000        : $AUTOSLEEP
0000 7C               NOP                                   \ No operation
0001 0F               SLEEP                                 \ CPU in 'sleep mode', interrupts enabled
0002 19               SET_BCF                               \ Set Branch and Carry flag
0003 80               SBRA $AUTOSLEEP                       \ Conditional short branch in page ($000)
0004 C1               SCALL $RESET                          \ Unconditional short CALL ($008)
```

```
0005 C1               SCALL $RESET                    \ Unconditional short CALL ($008)
0006 C1               SCALL $RESET                    \ Unconditional short CALL ($008)
0007 C1               SCALL $RESET                    \ Unconditional short CALL ($008)
0008
0008        ORIGIN $008
0008        : $RESET
0008 78 35            >SP VAR_00                      \ Set Expression Stack Pointer
000A 79 FC            >RP VAR_01                      \ Set return Stack Pointer direct
000C 43 99            CALL LABEL_000                  \ Unconditional long CALL ($399)
000E 1D               RTI                             \ Return from interrupt routine; enable all interrupts
000F C1               SCALL $RESET                    \ Unconditional short CALL ($008)
0010
0010        ORIGIN $010
0010        : ORPHAN_000
0010 29               3>R                             \ Move top 3 digits onto Return Stack
0011 20               TABLE                           \ Fetch an 8-bit ROM constant and performs an EXIT to Ret_PC
0012 C1               SCALL $RESET                    \ Unconditional short CALL ($008)
0013 C1               SCALL $RESET                    \ Unconditional short CALL ($008)
0014 C1               SCALL $RESET                    \ Unconditional short CALL ($008)
0015 C1               SCALL $RESET                    \ Unconditional short CALL ($008)
0016 C1               SCALL $RESET                    \ Unconditional short CALL ($008)
0017 C1               SCALL $RESET                    \ Unconditional short CALL ($008)
0018
0018        ORIGIN $018
0018        : ORPHAN_001
0018 63               LIT_3                           \ Push literal/constant $3 onto TOS
0019 1B               IN                              \ Read 4-bit I/O port to TOS
001A 2E               DROP                            \ Remove TOS digit from the Expression Stack
001B 25               EXIT                            \ Return from subroutine (';')
001C C1               SCALL $RESET                    \ Unconditional short CALL ($008)
001D C1               SCALL $RESET                    \ Unconditional short CALL ($008)
001E C1               SCALL $RESET                    \ Unconditional short CALL ($008)
001F C1               SCALL $RESET                    \ Unconditional short CALL ($008)
0020
0020        ORIGIN $020
0020        : ORPHAN_002
0020 29               3>R                             \ Move top 3 digits onto Return Stack
0021 2B               3R@                             \ Copy 3 digits from Return to Expression Stack
0022 20               TABLE                           \ Fetch an 8-bit ROM constant and performs an EXIT to Ret_PC
0023 C1               SCALL $RESET                    \ Unconditional short CALL ($008)
0024 C1               SCALL $RESET                    \ Unconditional short CALL ($008)
0025 C1               SCALL $RESET                    \ Unconditional short CALL ($008)
0026 C1               SCALL $RESET                    \ Unconditional short CALL ($008)
0027 C1               SCALL $RESET                    \ Unconditional short CALL ($008)
0028
0028        ORIGIN $028
0028        : ORPHAN_003
0028 69               LIT_9                           \ Push literal/constant $9 onto TOS
0029 26               SWAP                            \ Exchange the top 2 digits
002A 27               OVER                            \ Push a copy of TOS-1 onto TOS
002B 1F               OUT                             \ Write TOS to 4-bit I/O port
002C 1F               OUT                             \ Write TOS to 4-bit I/O port
```

We now have a correct $AUTOSLEEP routine as well as a reasonable $RESET, so we can be confident that we've identified our start of program. The orphans are probably only there because I haven't yet stitched all the data together so we're missing most of the code.

Now all I need to do is figure out their CRC algorithm and we can be 100% sure we've got all the data as per the original. *Update: CRC was reverse engineered from a qForth implementation, so marc4dasm now includes CRC checking.*

BTW, rompar & marc4dasm can be found at the Aperture Labs Tools page.

Posted by Adam "Major Malfunction" Laurie at 08:11

## 25 comments:

**Ken S** 31 January 2013 at 15:13

I'll start out by being honest. tl;dr.

But, I think I read enough to know that this is absolutely awesome.

Reply

**beardbastard** 31 January 2013 at 18:20

have used the same tekniq back in 2010 for successfully retriving a key from a freescale chip. bad ass post!

Reply

**iMs** 2 February 2013 at 07:29

*i felt sexually aroused by this, i even had shivers, *punches himself so that he knows he's not dreaming* i'm just sick of high level crap, finally something worth my reading. ACK to you ACK to you

Reply

**Brady** 4 February 2013 at 09:37

Awesome write up!

Reply

**Joxean Koret** 4 February 2013 at 09:50

Awesome!

Reply

**Unknown** 4 February 2013 at 10:56

It would be awesome to see this done to the speech chips used on the Mattel's Intellivoice. SP0256-012 as it does not contain the allophones found in the SP0256-AL2 or SP0256-019. The masked roms are very different, Would be very interesting to see how different they are and figure out the special undocumented codes available. There is also the very undocumented SPB-640 buffer interface chip that connects to the SP0256-012. Find out how these 2 chips fully interact and you will make many hackers happy.
http://www.intellivisionbrasil.com/.%5Cdocs%5Cmanuais-hardware%5CService_Intellivoice-3330.pdf

Reply

Replies

**WilliamF** 6 February 2013 at 05:33

I have a handful of SPO256 variants including the SPO256-012 and SPB-640 removed from an IntelliVoice. I'll donate the pair if those folks are interested.

**Reply**

**Anonymous** 4 February 2013 at 13:49

Fascinating post. Why MARC4 though?

Reply

**Anonymous** 4 February 2013 at 13:51

Great work man! Very well done and written!

Reply

**Kelly Black** 4 February 2013 at 20:30

Are you friends with this guy?

http://uvicrec.blogspot.com/2012/07/st-24c02-sector-17r-clock.html

Or him?
- interesting around 25:10

http://www.youtube.com/watch?v=K5miMbqYB4E

Reply

Replies

**Adam "Major Malfunction" Laurie** 5 February 2013 at 00:17

Yes, John McMaster is the guy that runs http://siliconpr0n.org.

The 6502 stuff is very interesting - I'm just glad that all I was dealing with was ROM! :P

**Reply**

**Guru** 4 February 2013 at 23:50

We've been doing just that for a number of years.....
http://members.iinet.net.au/~lantra9jp1/gurudumps/decap/index.html
and
http://members.iinet.net.au/~lantra9jp1/gurudumps/oldnews2009.html
and several other pages across my site :-)

Reply

Replies

**Adam "Major Malfunction" Laurie** 5 February 2013 at 00:35

First off I'm a huge MAME fan (and in a past life I even used to help build pub tables with MAME emulators in them.) Props to you! Thank you for all your work... :)

I looked at this site when I was first looking for existing code to extract the ROM data, and maybe I was being impatient by I couldn't find any links apart from one broken one to something call odump.zip (the .zip implied it was a windows thing, and I try to work on Linux where possible so I didn't try hard to find it).

I would be very interested to know more about how you extract the data, and also to see how well rompar does against your images (again, I couldn't find links to the actual images - just comments to say the ROM had been photographed)...

**Reply**

**Guru** 5 February 2013 at 13:32

it's all secret ;-)

from what I remember we had typing monkies do the bits. For some other chips there were a few devs working on the decoding and someone wrote a piece of software to extract the bits (it might have been Mooglyguy). Then the code was put through the MAME disassembler and the remaining unknown bits figured out to make good code. For others the decoding took a long time (months).
you could ask on one of the MAME-related forums and maybe one of the devs that worked on it will give you some more info about the actual process (I didn't work on the decoding)

Reply

Replies

**Adam "Major Malfunction" Laurie** 6 February 2013 at 06:56

If any MAME devs read this and would like to send me images of ROMs waiting to be decoded I'd be more than happy to help (and to tweak rompar if necessary).

**bernoulli** 6 February 2013 at 08:24

I was just about to plug the DSP16A but a quick search and I came across (almost literally) this post:
http://ajg.mameworld.info/

**Reply**

**Dave Nelson** 6 February 2013 at 04:42

Back in the early '70s, when Hughes Semiconductor was playing with PMOS chips, we used a technique of placing a powered up chip under a scanning electron microscope at fairly low magnification. The resulting image was very clear as to which traces were at 5V and which were at ground. Made recovering ROM data almost trivial, and worked great.

Reply

Replies

**Anonymous** 6 February 2013 at 13:58

That sounds like an incredible method of doing this. Do you have any more information regarding this process?

**Reply**

**Anonymous** 26 February 2013 at 14:23

I have a complaint. When I posted this to the facebook wall of a famous woman electrical engineer, the snippet delivered under the title is http://adamsblog.aperturelabs.com/2013/01/fun-with-masked-roms.html?showComment=1359818970972#c6171455941513719829 which is a pretty obnoxious thing to post without obvious context.

Reply

Replies

**Adam "Major Malfunction" Laurie** 27 February 2013 at 04:39

http://www.facebook.com/pages/The-Complaint-Department-Make-your-complaints-here/354354471577

**Reply**

**tz** 18 July 2013 at 14:56

Code monkeys solve for X. Chip monkeys dis-solve for X.

Reply

**Sean Riddle** 5 December 2013 at 14:02

Cool! I will try your code next time. This time I did it all manually:
http://www.seanriddle.com/psu.html

Reply

Replies

**Adam "Major Malfunction" Laurie** 5 December 2013 at 14:14

Wow! Nice work...

Since I wrote this I've always thought that game roms would be the most useful thing to focus on, so I'd be delighted if you or anyone else in this field helped to move it in that direction...

**Reply**

**john ceena** 12 August 2017 at 01:12

hmmmm

Reply

**sherrily6 Lane** 17 May 2018 at 22:32

For example, you can use your movie maker software to determine if you need to increase the intensity of your exercise routine if your heart rate is too slow, or exercise at a slower pace to avoid experiencing injuries or health problems such as cardiac arrest if your heart rate is too fast.

Reply

Enter your comment...

Comment as: andrey@xdel.r ▼ Sign out

Publish Preview Notify me

Newer Post Home

Subscribe to: Post Comments (Atom)