Marcus Engineering LLC

# Teaching New Tricks to an Old Micro

## Breaking into Chips By Reading the Datasheet

Hardwear.io USA 2024

hardwear.io

# Outline

- Overview
- Approach & Why This Part
- Reading The Manual
- Exploit 1: Checksums
- Exploit 2: Programming
- Demo

# Who Are We?

## Mark Omo

Engineering Director

- Leads the Engineering team at Marcus Engineering
- Expert in hardware and embedded security
- Background in regulated device design in the Medical, Industrial, Aerospace, and Consumer market segments
- Former System Engineering Lead at Google

## James Rowley

Senior Security Engineer

- Leads security programs at Marcus Engineering
- Expert in disassembly and embedded security
- Background in regulated device design in the Medical, Industrial, Aerospace, and Consumer market segments
- Couldn’t be bothered to update his headshot

# How did we get on this topic?

# Researching Safe Locks

Doing some security research on a popular model of low-end electronic safe lock from a major brand.

Need its firmware!

# What’s our first move?

# Teardown Time

# Where's the microcontroller?

Luckily they labeled it!

Here it is!

# Quick PCB Reverse Engineering

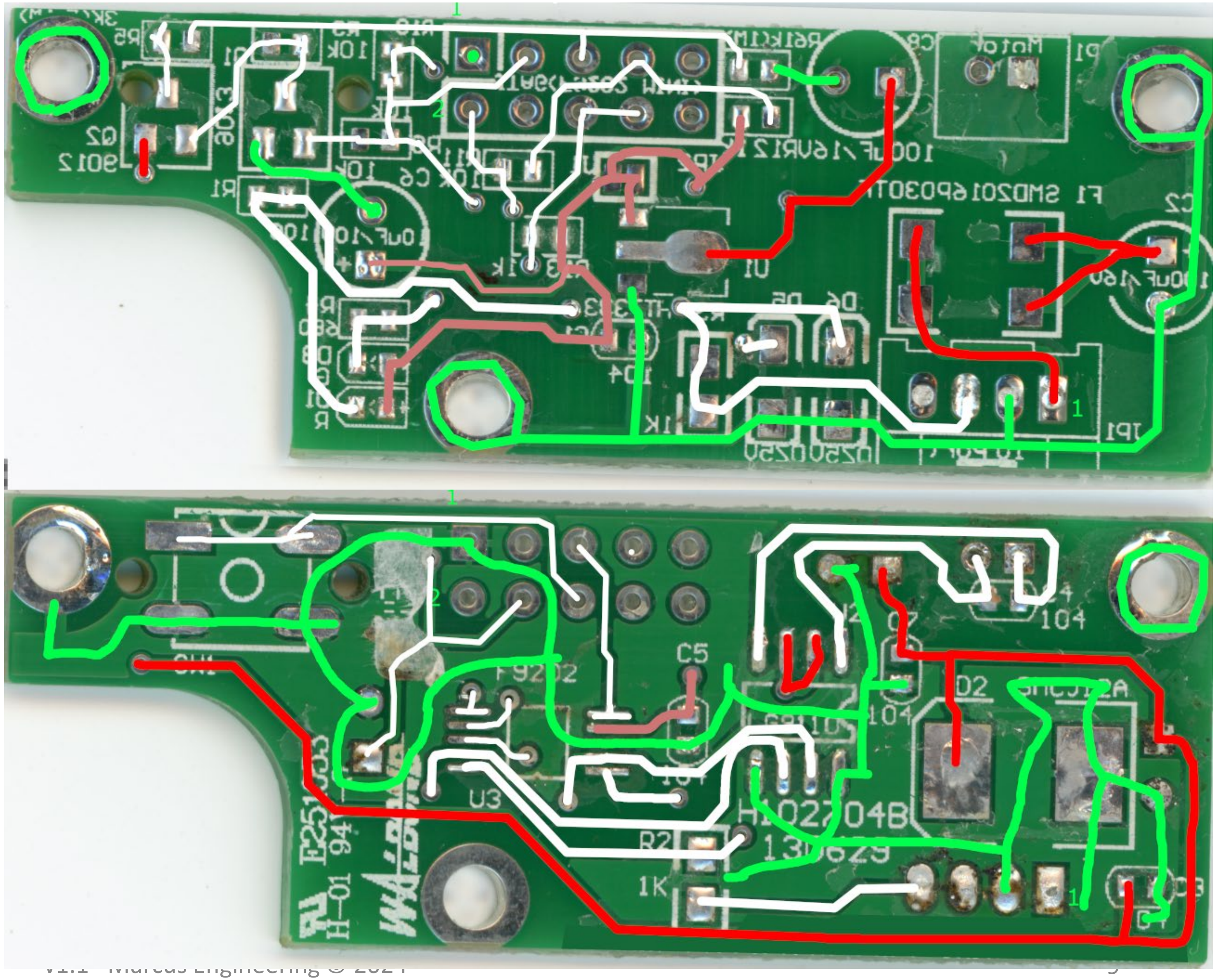
Side note:

They labeled every component with a part number

Part is a

~~NEC~~ Renesas

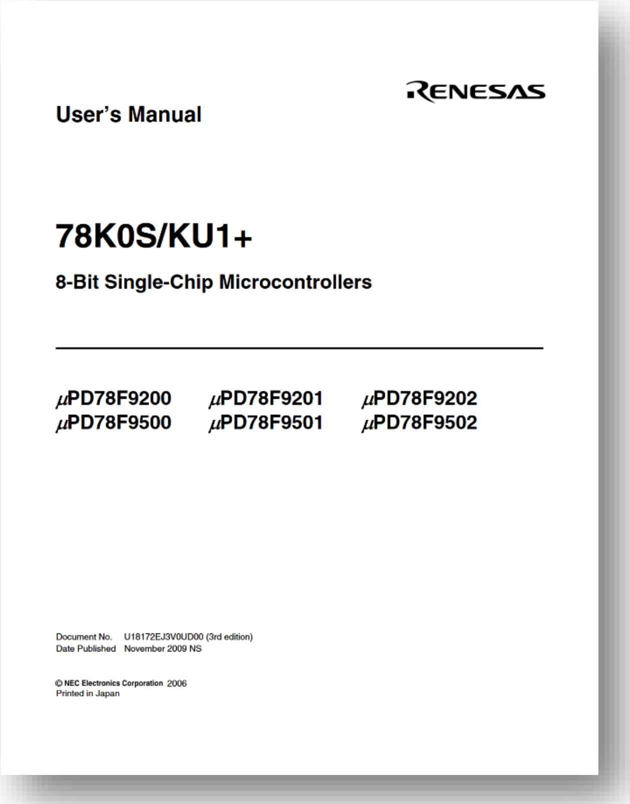
**μPD78F9202**



# Now to get the firmware.

# RTFM

What secrets could it hold?



User's Manual

RENESAS

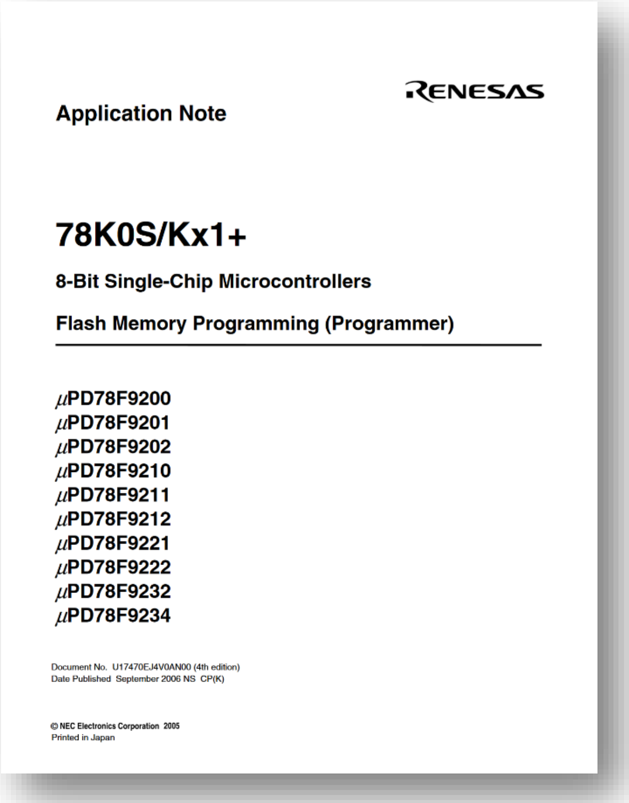
78K0S/KU1+

8-Bit Single-Chip Microcontrollers

μPD78F9200 μPD78F9201 μPD78F9202
μPD78F9500 μPD78F9501 μPD78F9502

Document No. U18172EJ3V0UD00 (3rd edition)
Date Published November 2009 NS

© NEC Electronics Corporation 2006
Printed in Japan



Application Note

RENESAS

78K0S/Kx1+

8-Bit Single-Chip Microcontrollers

Flash Memory Programming (Programmer)

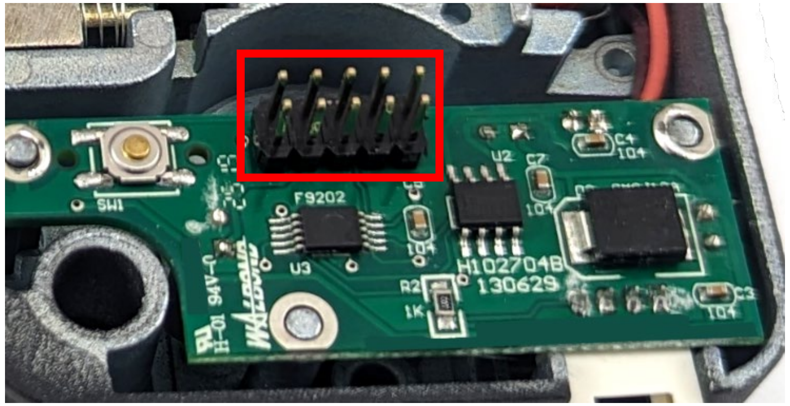
μPD78F9200
μPD78F9201
μPD78F9202
μPD78F9210
μPD78F9211
μPD78F9212
μPD78F9221
μPD78F9222
μPD78F9232
μPD78F9234

Document No. U17470EJ4V0AN00 (4th edition)
Date Published September 2006 NS CP(K)

© NEC Electronics Corporation 2005
Printed in Japan

# Flash Memory Programming

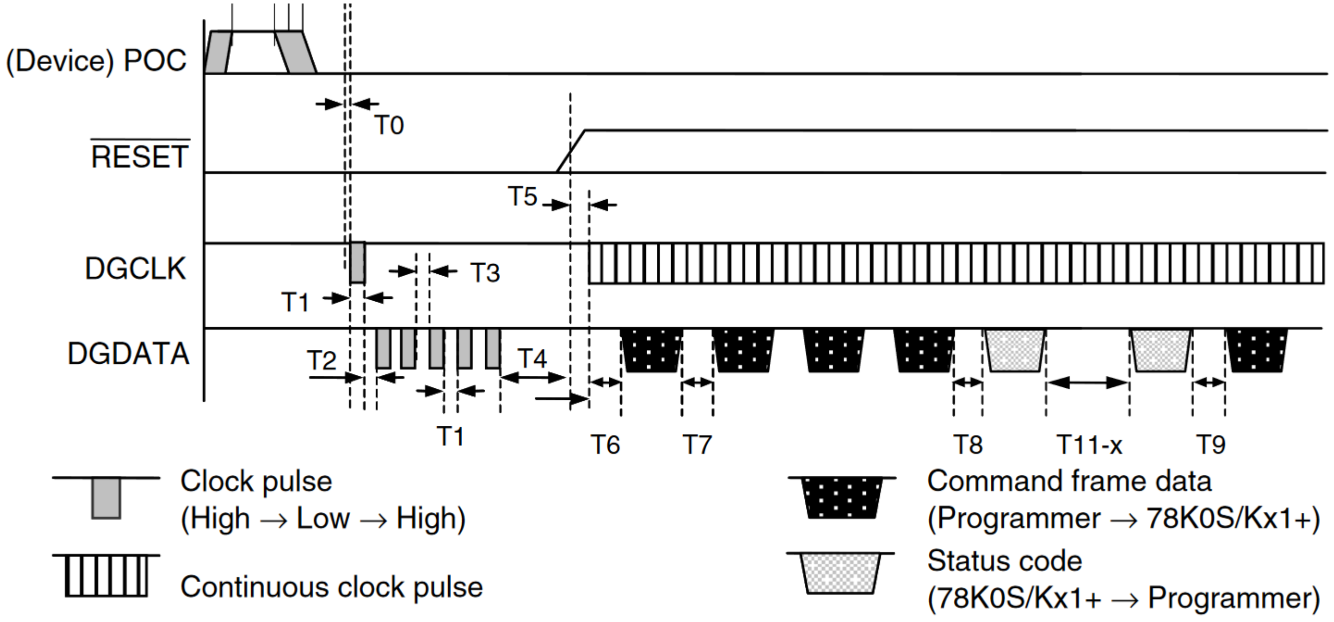
- Fortunately, NEC carefully documented the flash programming protocol.
  - Entry, commands, params, etc.
- The necessary pins are already broken out to a header, too.



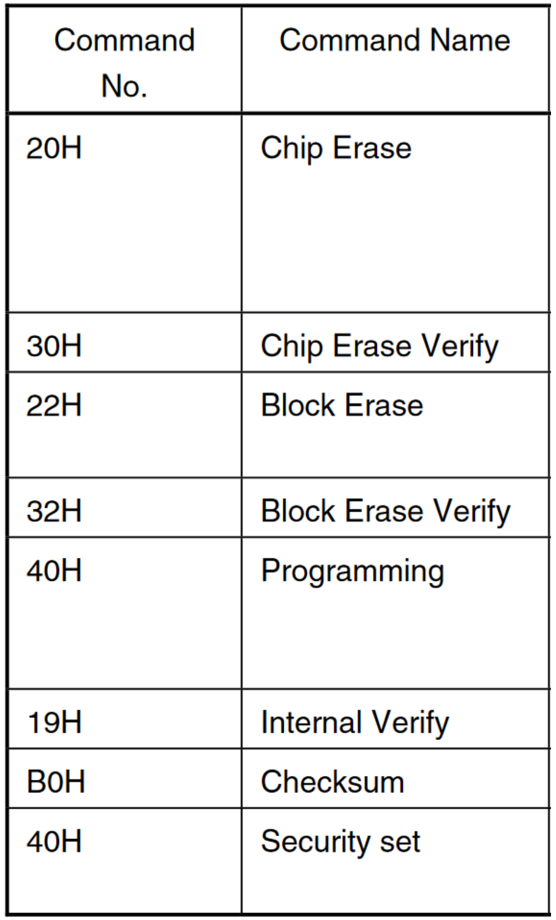
- Programmed a Teensy to do the half-duplex single-line UART communication and setup.

# How do we dump the code?

- Get it into programming mode, and…



- ah, there's no "read" command.



| Command No. | Command Name |
|---|---|
| 20H | Chip Erase |
| 30H | Chip Erase Verify |
| 22H | Block Erase |
| 32H | Block Erase Verify |
| 40H | Programming |
| 19H | Internal Verify |
| B0H | Checksum |
| 40H | Security set |

# What would we normally do?

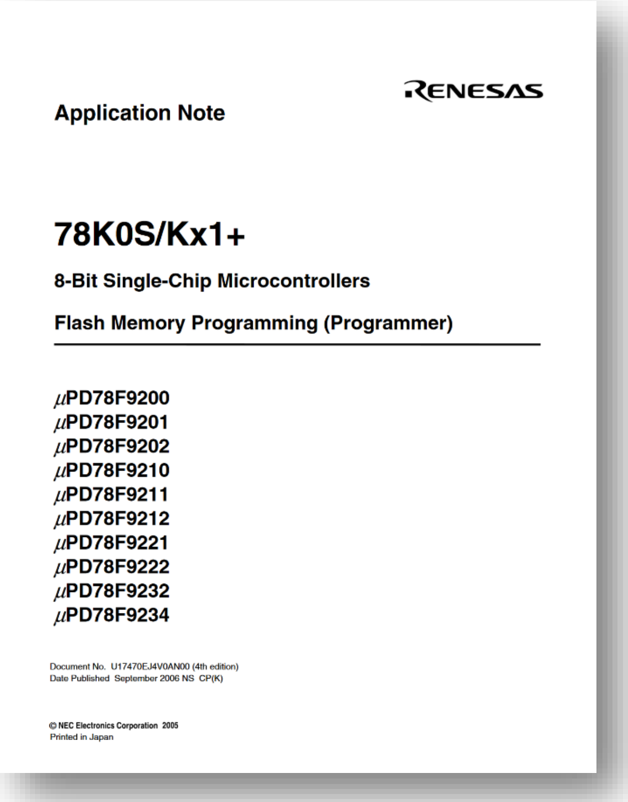
Marcus Engineering LLC

- On the typical micros we work with:
  - There is a read command
  - Or at least a debugging interface
- Those are good glitch targets to bypass security
  - However, this micro is not nearly that advanced.

# Time for a closer read of the programming guide…

# RTFM Part 2: Attack of the Datasheet

Burn Before Reading



RENESAS

Application Note

**78K0S/Kx1+**

**8-Bit Single-Chip Microcontrollers**

**Flash Memory Programming (Programmer)**

μPD78F9200
μPD78F9201
μPD78F9202
μPD78F9210
μPD78F9211
μPD78F9212
μPD78F9221
μPD78F9222
μPD78F9232
μPD78F9234

Document No. U17470EJ4V0AN00 (4th edition)
Date Published September 2006 NS CP(K)

© NEC Electronics Corporation 2005
Printed in Japan

# So there’s no read command.

- What else can we look for?

- Looking for anything that interacts with the flash
    - Can we gain insight about that’s inside?

- What data might unintentionally be leaked?

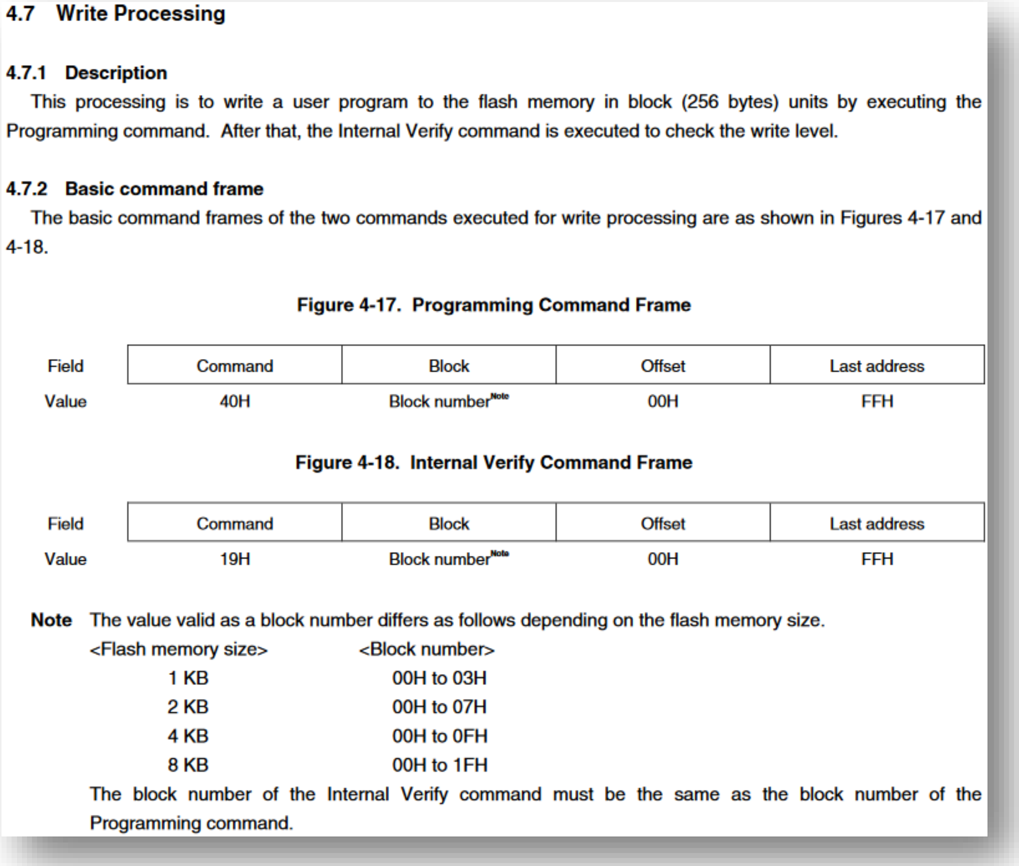
# What commands do we have?

- Chip Erase
- Block Erase
- Chip/Block Erase Verify
- Programming
- Security Set
- Internal Verify
- Checksum

# Write/Programming

- Writes data.
  - That’s a real flash write, so can only change 1’s to 0’s.

- <u>We don’t particularly want to overwrite the data.</u>



4.7 **Write Processing**

4.7.1 **Description**

This processing is to write a user program to the flash memory in block (256 bytes) units by executing the Programming command. After that, the Internal Verify command is executed to check the write level.

4.7.2 **Basic command frame**

The basic command frames of the two commands executed for write processing are as shown in Figures 4-17 and 4-18.

**Figure 4-17. Programming Command Frame**

| Field | Command | Block | Offset | Last address |
|---|---|---|---|---|
| Value | 40H | Block number[Note] | 00H | FFH |

**Figure 4-18. Internal Verify Command Frame**

| Field | Command | Block | Offset | Last address |
|---|---|---|---|---|
| Value | 19H | Block number[Note] | 00H | FFH |

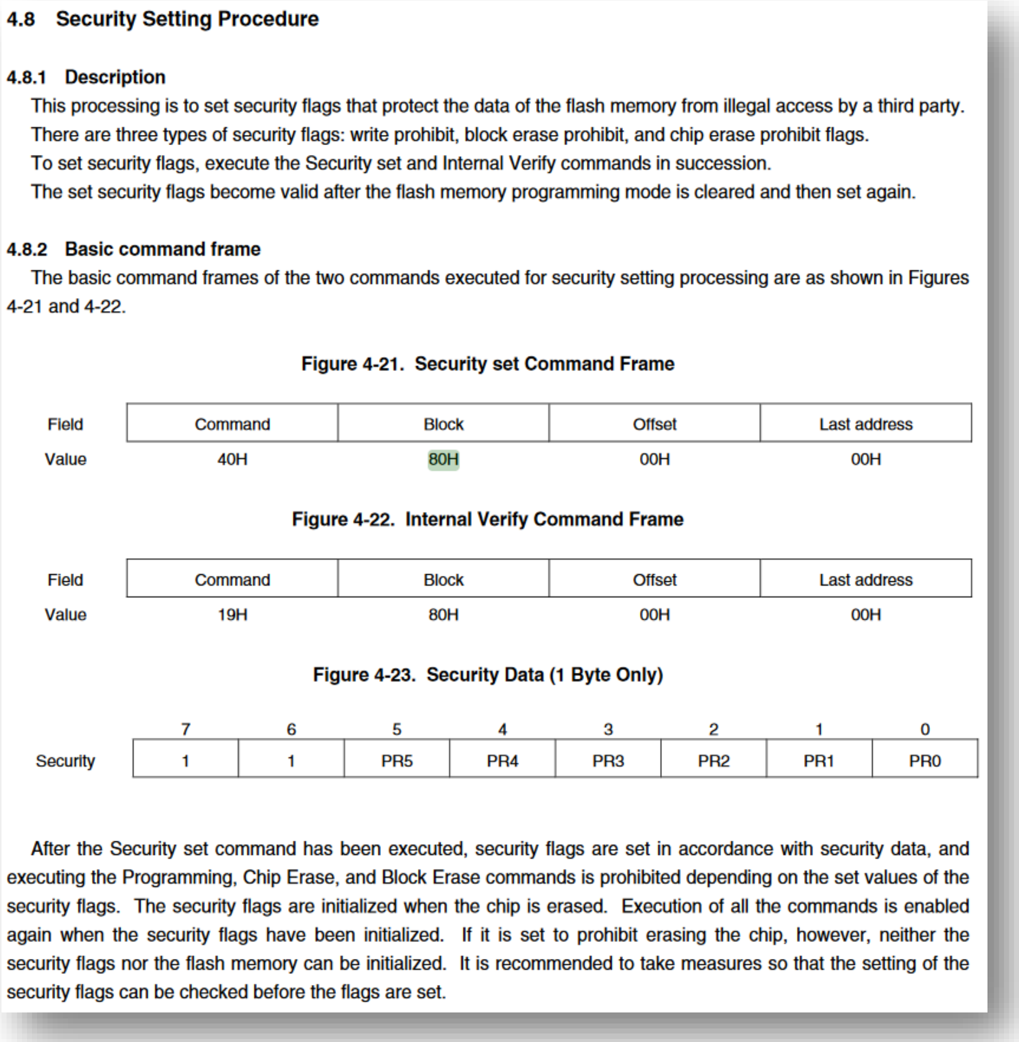
**Note** The value valid as a block number differs as follows depending on the flash memory size.

| <Flash memory size> | <Block number> |
|---|---|
| 1 KB | 00H to 03H |
| 2 KB | 00H to 07H |
| 4 KB | 00H to 0FH |
| 8 KB | 00H to 1FH |

The block number of the Internal Verify command must be the same as the block number of the Programming command.

# Security Set

- Writes data to the special security byte.
    - That’s a real flash write, so can only change 1’s to 0’s.
    - Weird, it’s the same command byte as for Programming…
- <u>We don’t particularly want to activate more security.</u>



**4.8 Security Setting Procedure**

**4.8.1 Description**

This processing is to set security flags that protect the data of the flash memory from illegal access by a third party.
There are three types of security flags: write prohibit, block erase prohibit, and chip erase prohibit flags.
To set security flags, execute the Security set and Internal Verify commands in succession.
The set security flags become valid after the flash memory programming mode is cleared and then set again.

**4.8.2 Basic command frame**

The basic command frames of the two commands executed for security setting processing are as shown in Figures 4-21 and 4-22.

**Figure 4-21. Security set Command Frame**

| Field | Command | Block | Offset | Last address |
|---|---|---|---|---|
| Value | 40H | 80H | 00H | 00H |

**Figure 4-22. Internal Verify Command Frame**

| Field | Command | Block | Offset | Last address |
|---|---|---|---|---|
| Value | 19H | 80H | 00H | 00H |

**Figure 4-23. Security Data (1 Byte Only)**

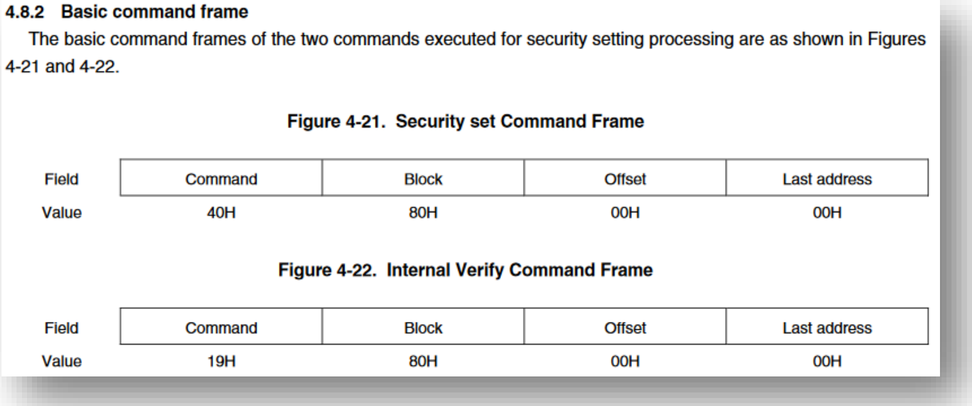
|  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Security | 1 | 1 | PR5 | PR4 | PR3 | PR2 | PR1 | PR0 |

After the Security set command has been executed, security flags are set in accordance with security data, and executing the Programming, Chip Erase, and Block Erase commands is prohibited depending on the set values of the security flags. The security flags are initialized when the chip is erased. Execution of all the commands is enabled again when the security flags have been initialized. If it is set to prohibit erasing the chip, however, neither the security flags nor the flash memory can be initialized. It is recommended to take measures so that the setting of the security flags can be checked before the flags are set.

# Internal Verify

- Run after Programming or Security Set command.
  - Maybe checks if all bytes match what was sent.
    - I don’t think there’s a write buffer, though.
  - Or maybe checks analog write level somehow.
- Can’t send it data to verify, it just does its thing.

“Checks the write level of a specified block.”



**4.8.2 Basic command frame**

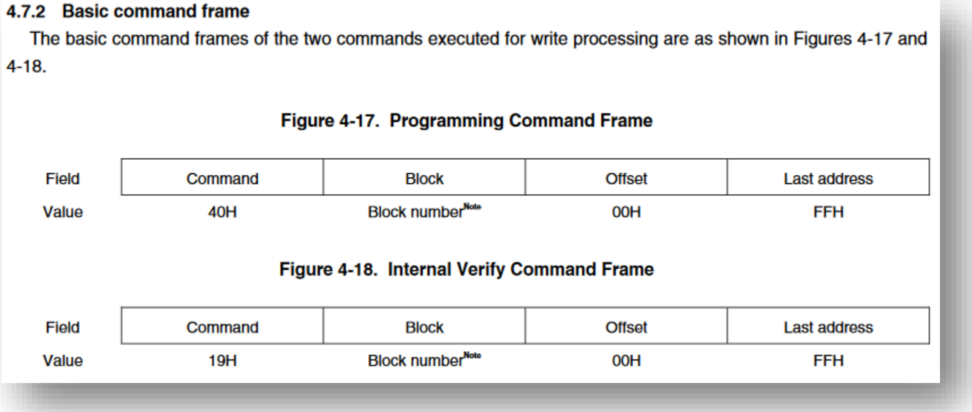
The basic command frames of the two commands executed for security setting processing are as shown in Figures 4-21 and 4-22.

**Figure 4-21. Security set Command Frame**

| Field | Command | Block | Offset | Last address |
|---|---|---|---|---|
| Value | 40H | 80H | 00H | 00H |

**Figure 4-22. Internal Verify Command Frame**

| Field | Command | Block | Offset | Last address |
|---|---|---|---|---|
| Value | 19H | 80H | 00H | 00H |



**4.7.2 Basic command frame**

The basic command frames of the two commands executed for write processing are as shown in Figures 4-17 and 4-18.

**Figure 4-17. Programming Command Frame**

| Field | Command | Block | Offset | Last address |
|---|---|---|---|---|
| Value | 40H | Block number$^{\text{Note}}$ | 00H | FFH |

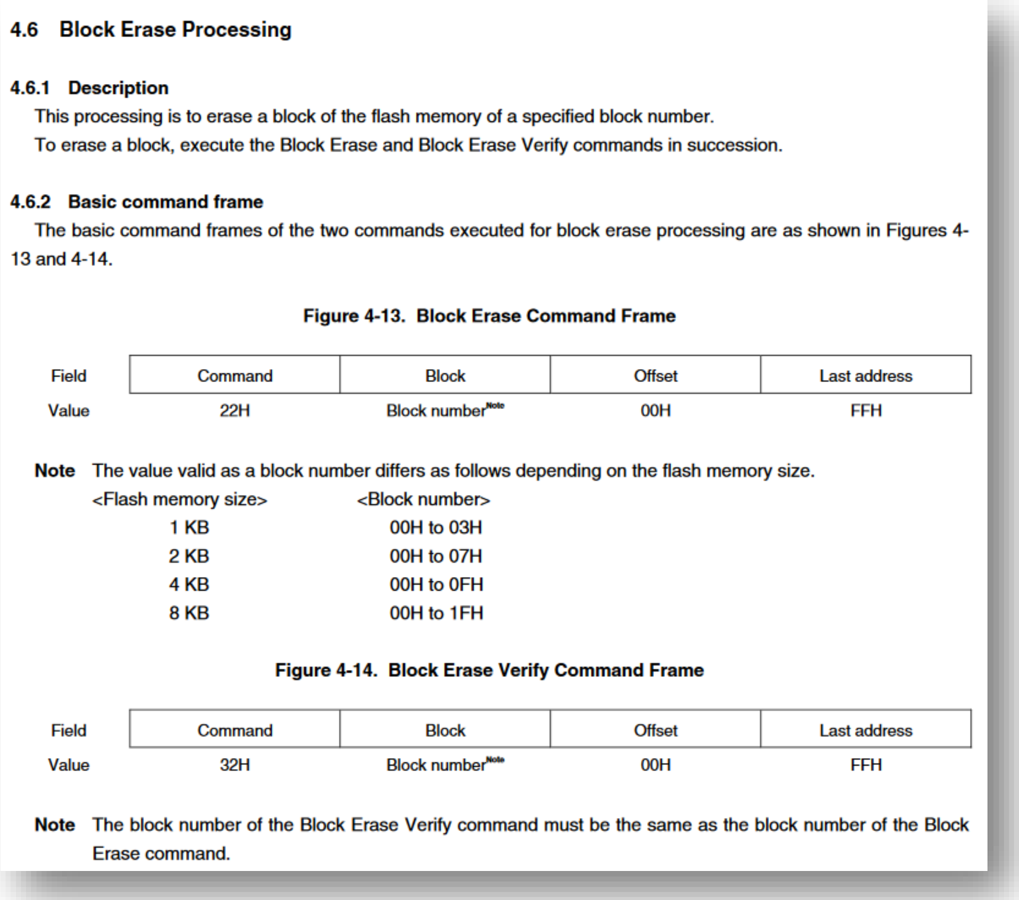
**Figure 4-18. Internal Verify Command Frame**

| Field | Command | Block | Offset | Last address |
|---|---|---|---|---|
| Value | 19H | Block number$^{\text{Note}}$ | 00H | FFH |

# Block Erase

- Erases a block of flash.
  - Don’t want to do that.
- Could this erase security bits independently?
  - Notice that after Chip Erase, you’re supposed to send Block Erase Verify for block 80h.
  - Does 80h correspond to the security bits?
  - Still, there’s no disabled read command for us to get.

“Checks the erasure level of the entire flash memory.”



**4.6 Block Erase Processing**

**4.6.1 Description**

This processing is to erase a block of the flash memory of a specified block number.
To erase a block, execute the Block Erase and Block Erase Verify commands in succession.

**4.6.2 Basic command frame**

The basic command frames of the two commands executed for block erase processing are as shown in Figures 4-13 and 4-14.

**Figure 4-13. Block Erase Command Frame**

| Field | Command | Block | Offset | Last address |
|---|---|---|---|---|
| Value | 22H | Block number[Note] | 00H | FFH |

**Note** The value valid as a block number differs as follows depending on the flash memory size.

| <Flash memory size> | <Block number> |
|---|---|
| 1 KB | 00H to 03H |
| 2 KB | 00H to 07H |
| 4 KB | 00H to 0FH |
| 8 KB | 00H to 1FH |

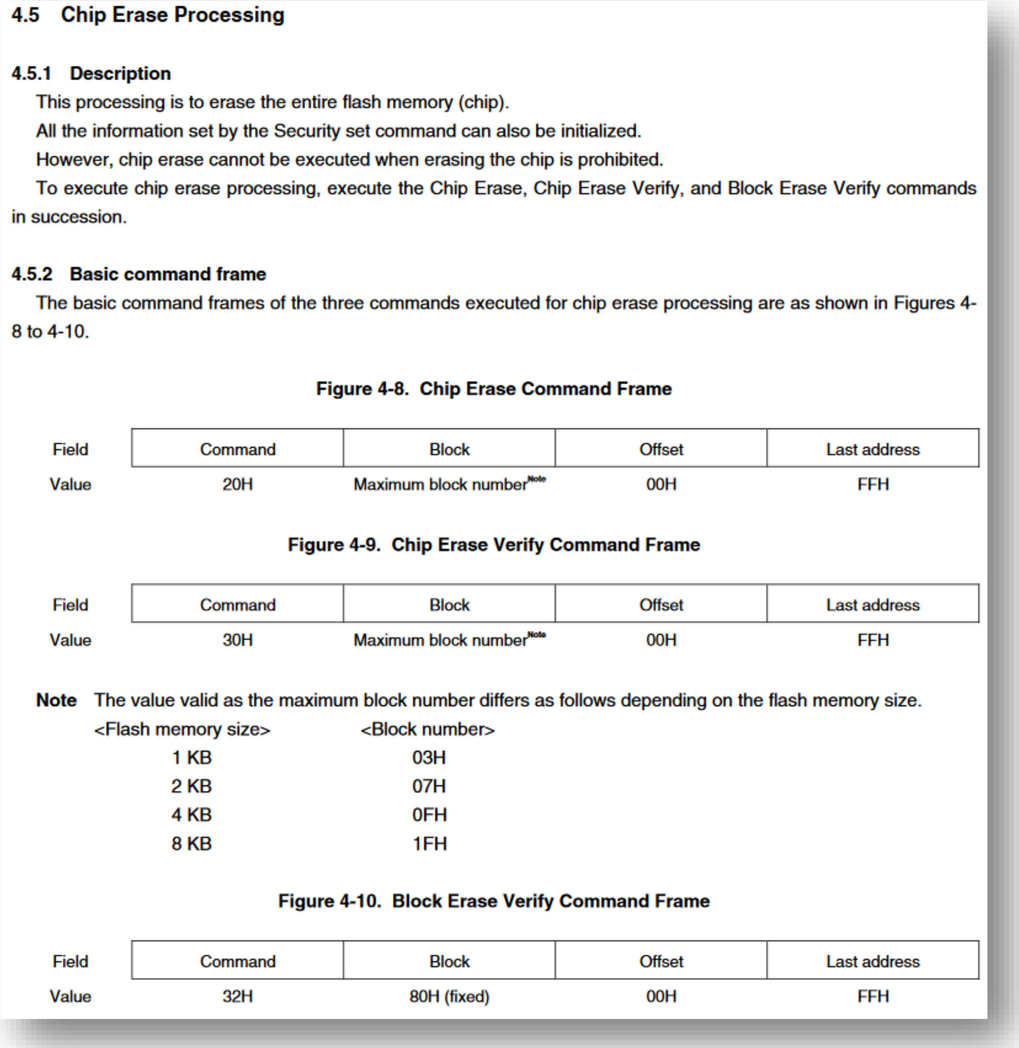
**Figure 4-14. Block Erase Verify Command Frame**

| Field | Command | Block | Offset | Last address |
|---|---|---|---|---|
| Value | 32H | Block number[Note] | 00H | FFH |

**Note** The block number of the Block Erase Verify command must be the same as the block number of the Block Erase command.

# Chip Erase

- Erases all flash.
  - Including security bits.
- Oddly, you have to tell it how many blocks it has?
  - Maybe exploitable?
  - <u>Still, don't want to erase anything.</u>



### 4.5 Chip Erase Processing

**4.5.1 Description**

This processing is to erase the entire flash memory (chip).

All the information set by the Security set command can also be initialized.

However, chip erase cannot be executed when erasing the chip is prohibited.

To execute chip erase processing, execute the Chip Erase, Chip Erase Verify, and Block Erase Verify commands in succession.

**4.5.2 Basic command frame**

The basic command frames of the three commands executed for chip erase processing are as shown in Figures 4-8 to 4-10.

**Figure 4-8. Chip Erase Command Frame**

| Field | Command | Block | Offset | Last address |
|---|---|---|---|---|
| Value | 20H | Maximum block number[Note] | 00H | FFH |

**Figure 4-9. Chip Erase Verify Command Frame**

| Field | Command | Block | Offset | Last address |
|---|---|---|---|---|
| Value | 30H | Maximum block number[Note] | 00H | FFH |

**Note** The value valid as the maximum block number differs as follows depending on the flash memory size.

| <Flash memory size> | <Block number> |
|---|---|
| 1 KB | 03H |
| 2 KB | 07H |
| 4 KB | 0FH |
| 8 KB | 1FH |

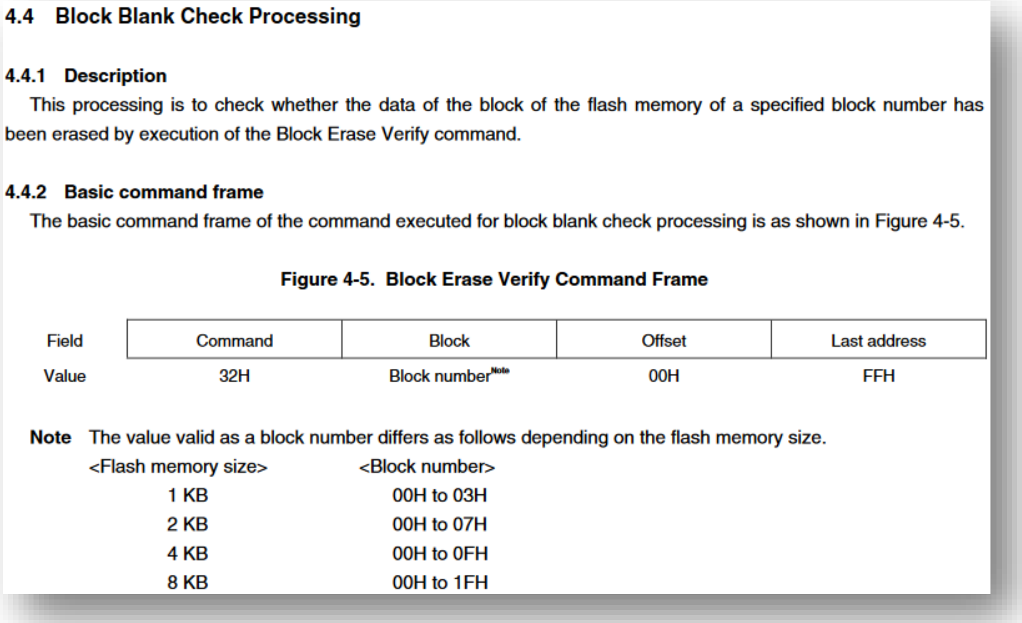
**Figure 4-10. Block Erase Verify Command Frame**

| Field | Command | Block | Offset | Last address |
|---|---|---|---|---|
| Value | 32H | 80H (fixed) | 00H | FFH |

# Block Blank Check (Block Erase Verify)

- Checks if a block was erased.
  - Maybe checks if all bytes are logical 1’s, or…
  - Maybe checks analog erase level somehow.
  - <u>Best case, can tell us if a whole block is blank.</u>

“Checks the erasure level of a specified block.”



**4.4 Block Blank Check Processing**

**4.4.1 Description**

This processing is to check whether the data of the block of the flash memory of a specified block number has been erased by execution of the Block Erase Verify command.

**4.4.2 Basic command frame**

The basic command frame of the command executed for block blank check processing is as shown in Figure 4-5.

**Figure 4-5. Block Erase Verify Command Frame**

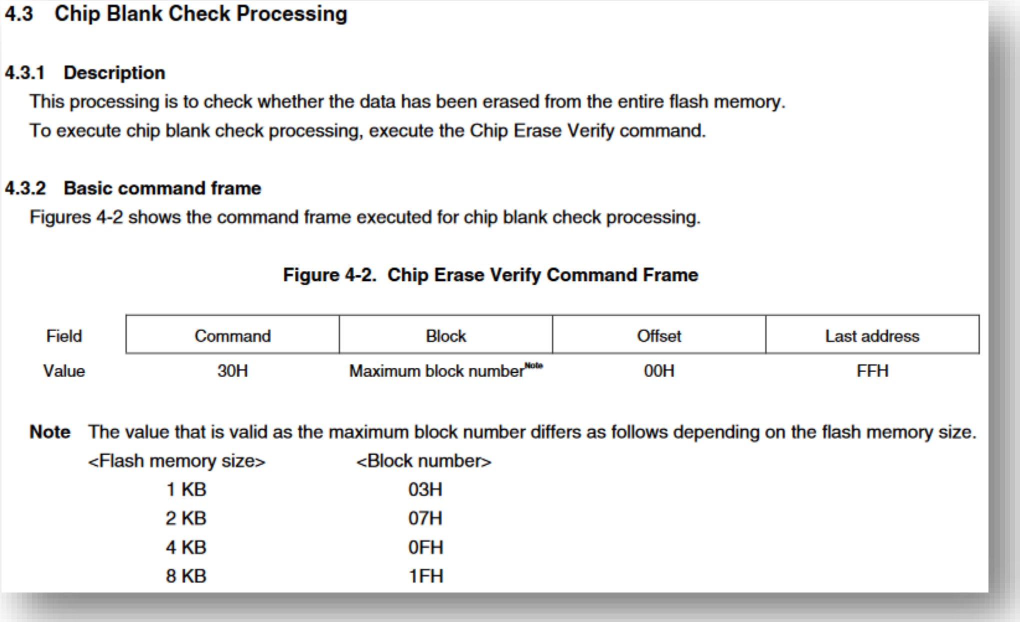
| Field | Command | Block | Offset | Last address |
|---|---|---|---|---|
| Value | 32H | Block number[Note] | 00H | FFH |

**Note** The value valid as a block number differs as follows depending on the flash memory size.

| <Flash memory size> | <Block number> |
|---|---|
| 1 KB | 00H to 03H |
| 2 KB | 00H to 07H |
| 4 KB | 00H to 0FH |
| 8 KB | 00H to 1FH |

# Chip Blank Check (Chip Erase Verify)

- Checks if the whole chip was erased.
  - Maybe checks if all bytes are logical 1's, or…
  - Maybe checks analog erase level somehow.
  - Best case, can tell us if the whole chip is blank.
    - Hopefully, it isn't…



**4.3 Chip Blank Check Processing**

**4.3.1 Description**

This processing is to check whether the data has been erased from the entire flash memory.
To execute chip blank check processing, execute the Chip Erase Verify command.

**4.3.2 Basic command frame**

Figures 4-2 shows the command frame executed for chip blank check processing.

**Figure 4-2. Chip Erase Verify Command Frame**

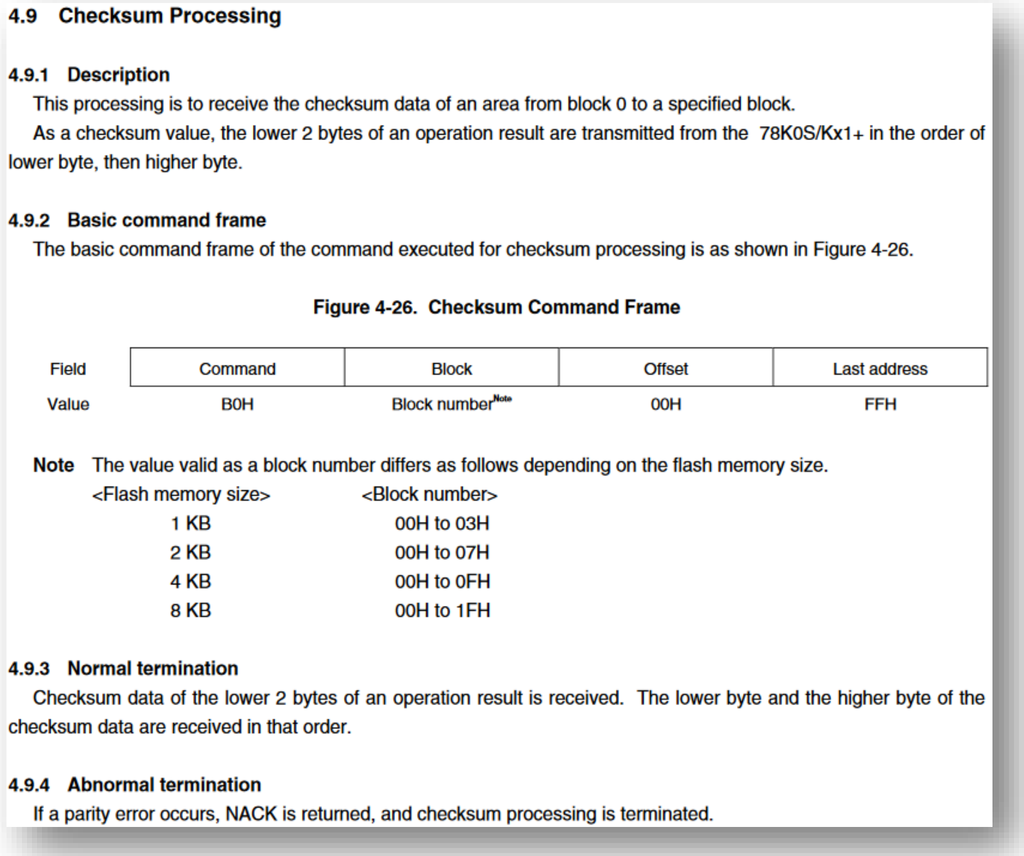
| Field | Command | Block | Offset | Last address |
|---|---|---|---|---|
| Value | 30H | Maximum block number[Note] | 00H | FFH |

**Note** The value that is valid as the maximum block number differs as follows depending on the flash memory size.

| <Flash memory size> | <Block number> |
|---|---|
| 1 KB | 03H |
| 2 KB | 07H |
| 4 KB | 0FH |
| 8 KB | 1FH |

# Checksum

- Computes the checksum of one or more blocks.
  - So it’s reading the flash!
  - Only works on blocks… but you can specify the start and end address?
- **<u>This might work.</u>**



### 4.9 Checksum Processing

**4.9.1 Description**

This processing is to receive the checksum data of an area from block 0 to a specified block.

As a checksum value, the lower 2 bytes of an operation result are transmitted from the 78K0S/Kx1+ in the order of lower byte, then higher byte.

**4.9.2 Basic command frame**

The basic command frame of the command executed for checksum processing is as shown in Figure 4-26.

**Figure 4-26. Checksum Command Frame**

| Field | Command | Block | Offset | Last address |
|---|---|---|---|---|
| Value | B0H | Block number[Note] | 00H | FFH |

**Note** The value valid as a block number differs as follows depending on the flash memory size.

| <Flash memory size> | <Block number> |
|---|---|
| 1 KB | 00H to 03H |
| 2 KB | 00H to 07H |
| 4 KB | 00H to 0FH |
| 8 KB | 00H to 1FH |

**4.9.3 Normal termination**

Checksum data of the lower 2 bytes of an operation result is received. The lower byte and the higher byte of the checksum data are received in that order.

**4.9.4 Abnormal termination**

If a parity error occurs, NACK is returned, and checksum processing is terminated.

# Checksum command looks promising.

- We can get the checksum of each block.
  - That’s something.

- **Could we get the checksum of each byte?**

# To the lab!

# How to test this?

- This chip needs some special inputs to get into programming mode, in a sequence:
    - Power
    - DGCLK pulse
    - DGDATA pulse
    - DGCLK input clock
- Only after all of this is DGDATA used as a UART pin.
- So we'll write an Arduino program to handle all that.
    - Then may as well implement the commands there too.

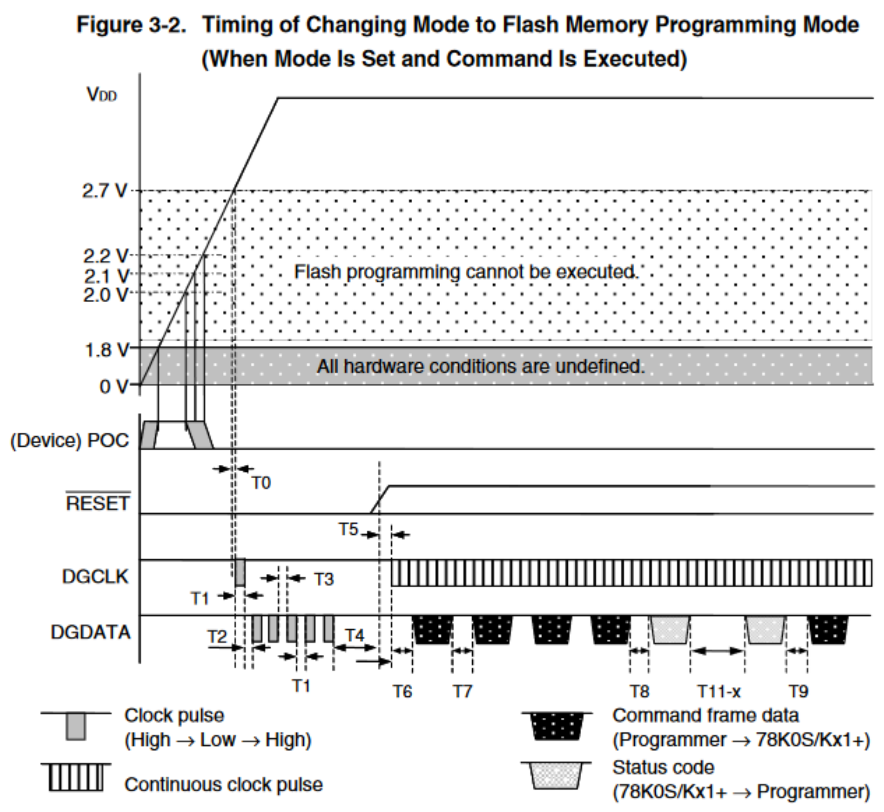
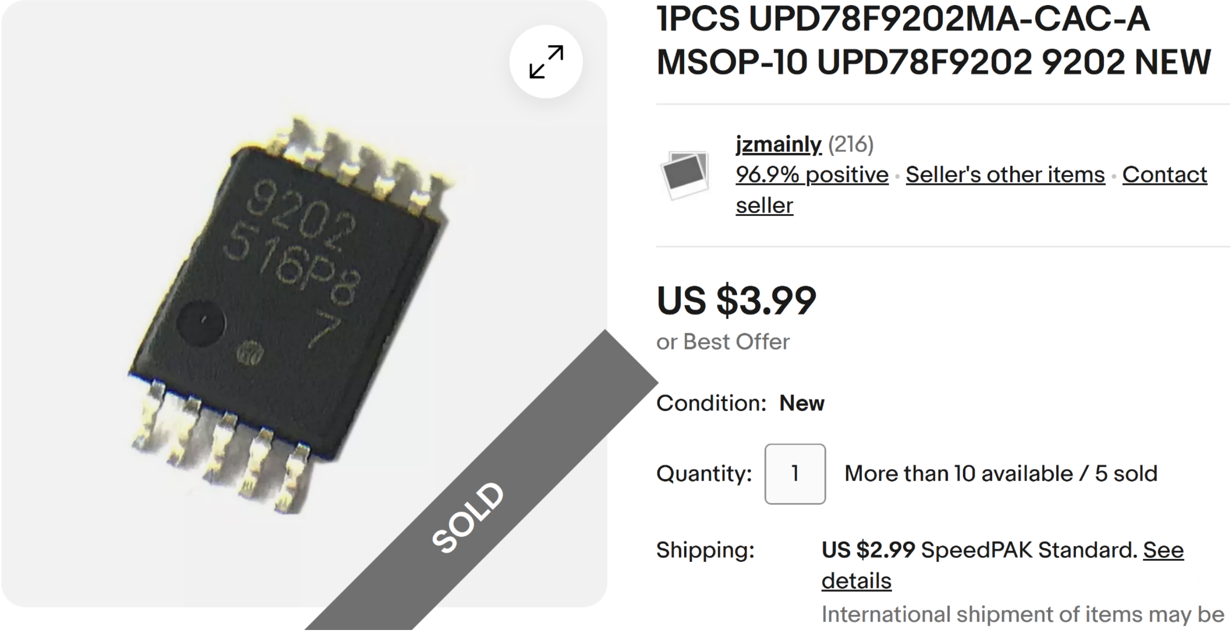


Figure 3-2. Timing of Changing Mode to Flash Memory Programming Mode (When Mode Is Set and Command Is Executed)

# But first, to eBay…

- Who even sells these still?



**1PCS UPD78F9202MA-CAC-A MSOP-10 UPD78F9202 9202 NEW**

jzmainly (216)
96.9% positive · Seller's other items · Contact seller

**US $3.99**
or Best Offer

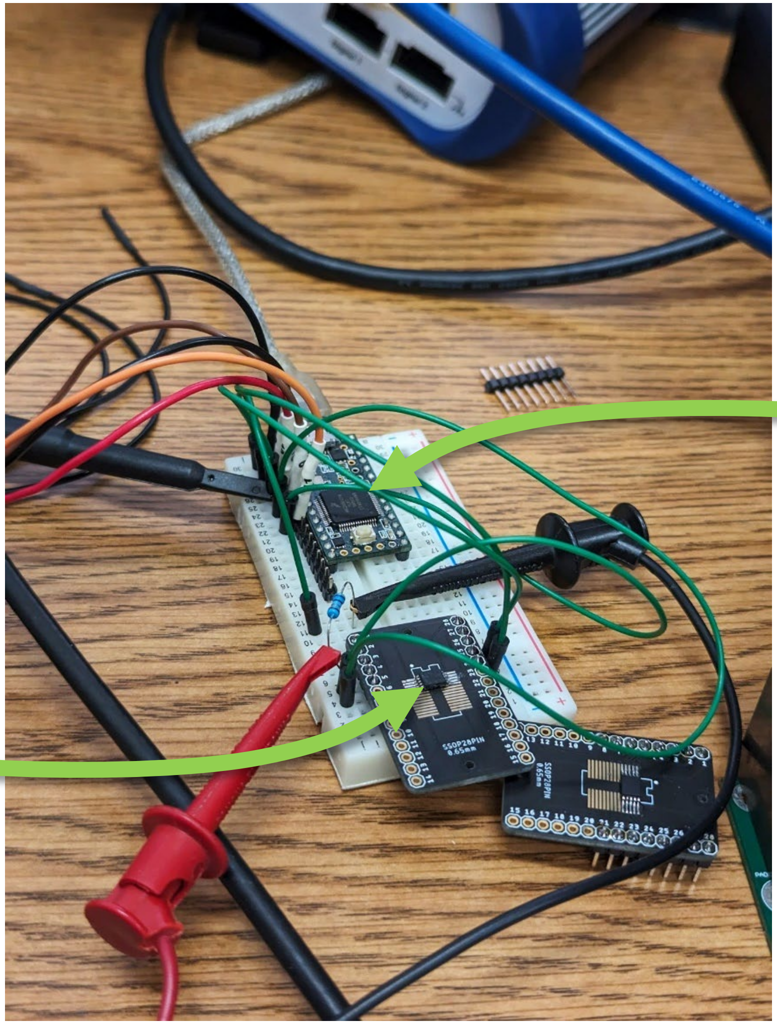
Condition: **New**

Quantity: 1 More than 10 available / 5 sold

Shipping: **US $2.99** SpeedPAK Standard. See details
International shipment of items may be

- Fortunately, not too rare, though gone from the likes of DigiKey and Mouser.

# To the lab, then.



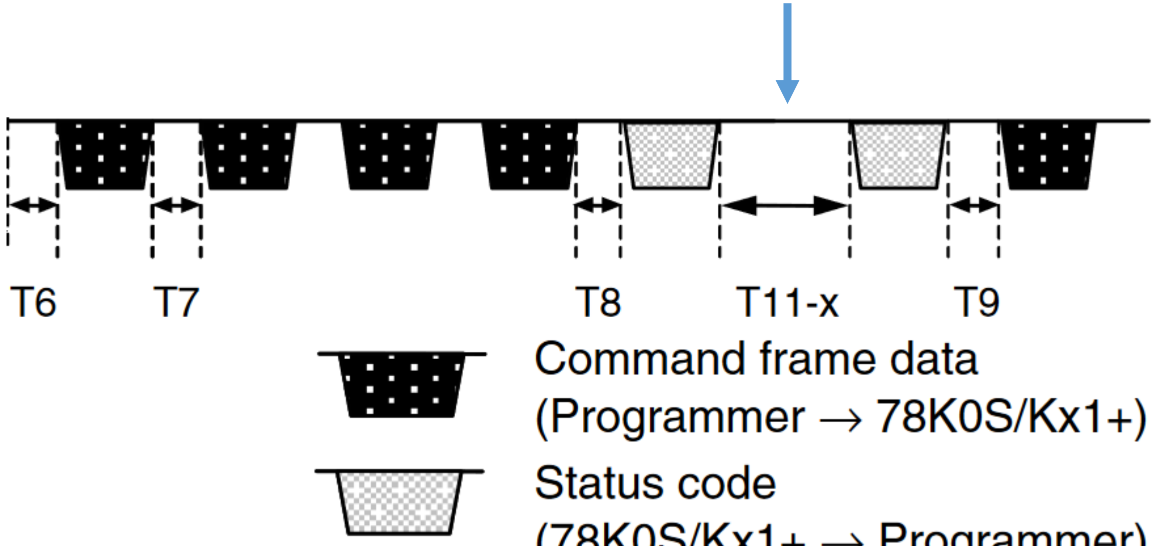
**Teensy 3.2**

**µPD78F9202**

# Command Format

- Fortunately, this chip keeps comms real simple.
  - Send a command byte.
  - Send some parameters.
  - Get a status byte.
  - Send data?
    - Get status, repeat?
  - Get another, final status byte.

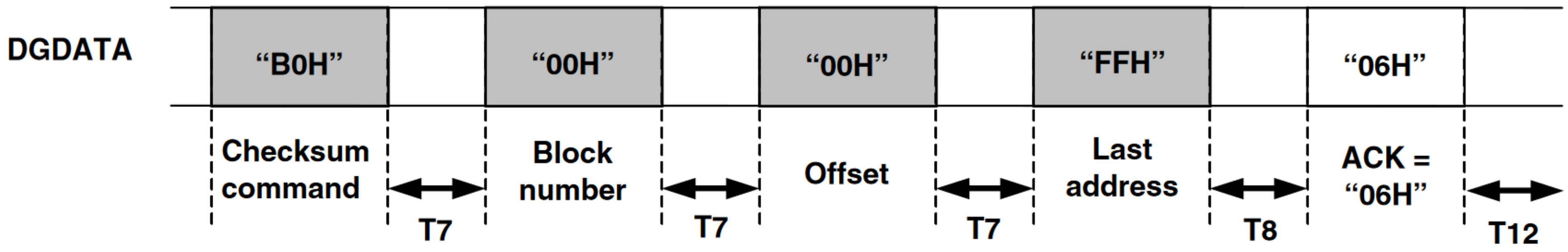
Maybe more data and status here



# Testing the Checksum Command

- B0 00 00 FF
  - Checksum, block 0, start addr. 0, end addr. FFh.
- Does it work?

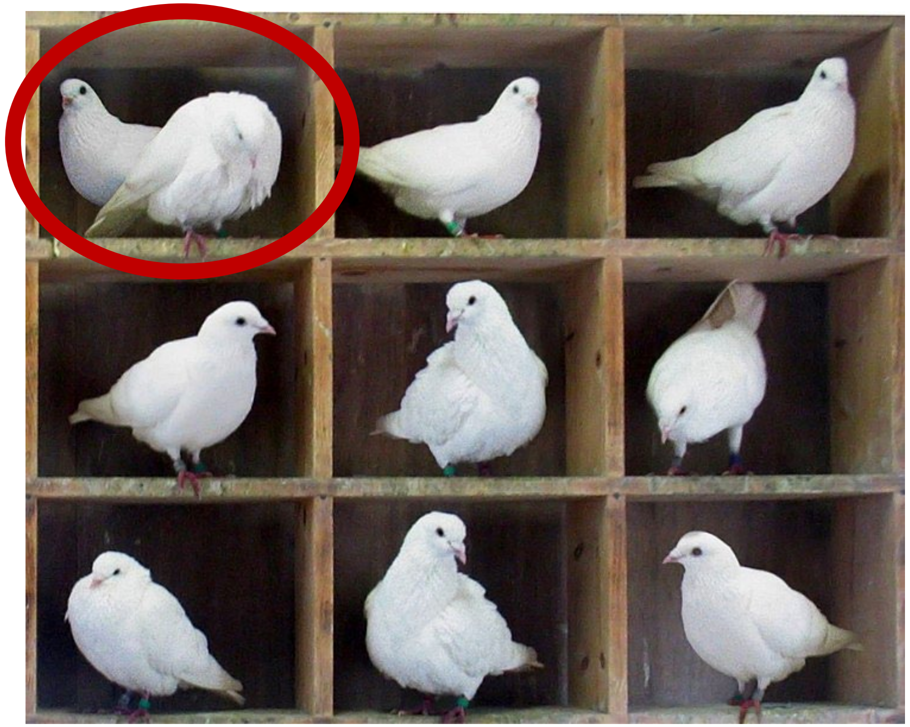
Figure 4-28. Timing Chart of Checksum Processing



- Yes!
  - Ok, comms are working.

# Pigeonhole Principle

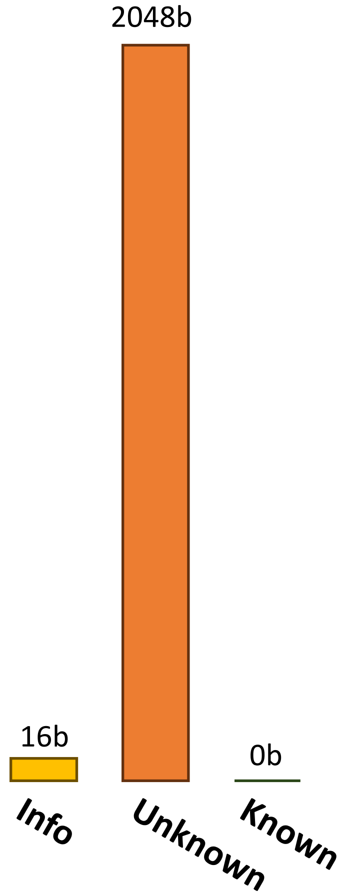
- The checksum output is a function of a data input.
  - We'd like to learn that data from looking at the checksum.
- There are many, many more inputs than there are outputs.
- So, more than one input can result in any given output.
  - So knowing the output doesn't tell us which input is actually present!



Source: Wikipedia;
Pigeons-in-holes.jpg by en:User:BenFrantzDale; this image by en:User:McKay

# Full block checksums only

- Manual says we can only get the checksum of a <u>full 256-byte block</u>.
  - It's not nothing!
- Only 16 bits* output for 2048 bits input.
  - Pigeonhole principle rules out making any meaningful guess about the overall page contents just from that output.

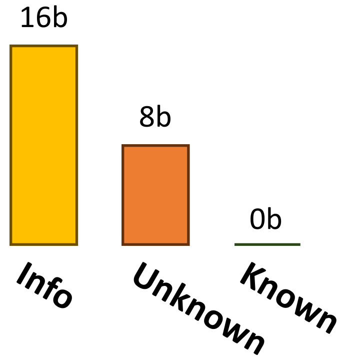


*More like 12 bits…

# If we had a single byte’s checksum…

- We can run every possible byte through the checksum algorithm.
  - And see which one results in that checksum!

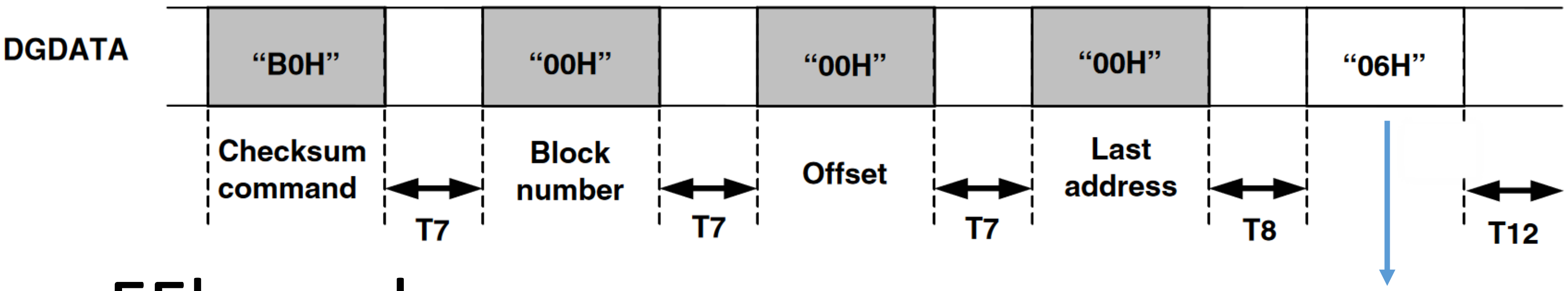
- 16 bits output uniquely identifies 8 bits input.
  - 63356 possible outputs
  - Only 256 possible inputs



# Checksum of a single byte

- B0 00 00 00
  - Checksum, block 0, start addr. 0, end addr. 0.
- <u>Does it work?</u>

**Figure 4-28. Timing Chart of Checksum Processing**



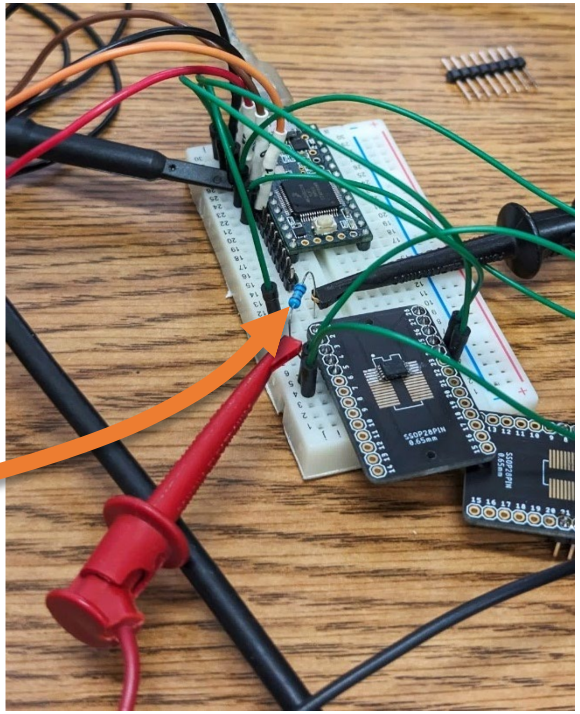
NACK "15h"

- No.
  - Only 00h – FFh works.

# Powerline analysis to the rescue!

Marcus Engineering LLC

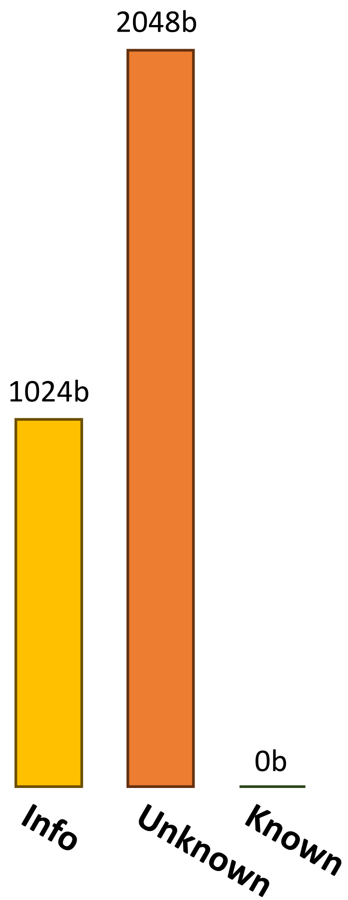
- Perhaps we can learn some information by looking at the CPU's power consumption?
- Instrumented the ground return path of the chip with a resistor and a PC oscilloscope.
- Let's look at the Checksum processing.



Current Shunt

# Powerline analysis to the rescue?

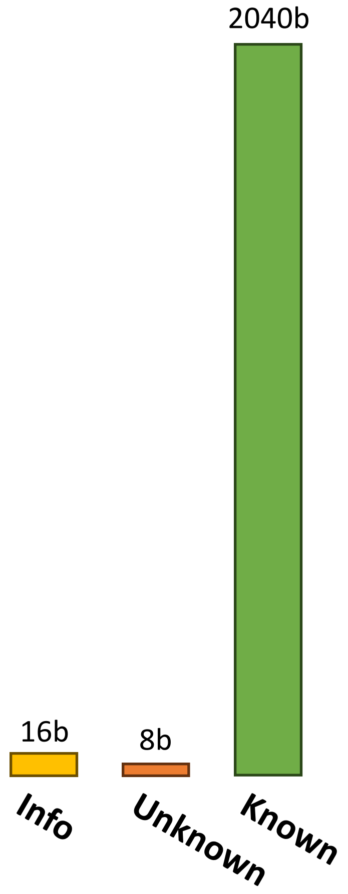
- Can we tell the value of the data that the checksum computation is processing?
  - Somewhat, yes!
- How much can we tell?
  - Only learn about 4 bits per byte.
  - <u>Not enough!</u>
- Well, that’s hard stuff anyways. Let’s keep playing with commands.



# Single byte checksum works

- We can’t do that directly
- How can we do it anyways?

- Imagine deleting everything except 1 byte.
  - Then it’s really only a checksum of that byte.
- Could do that to 256 chips, or…

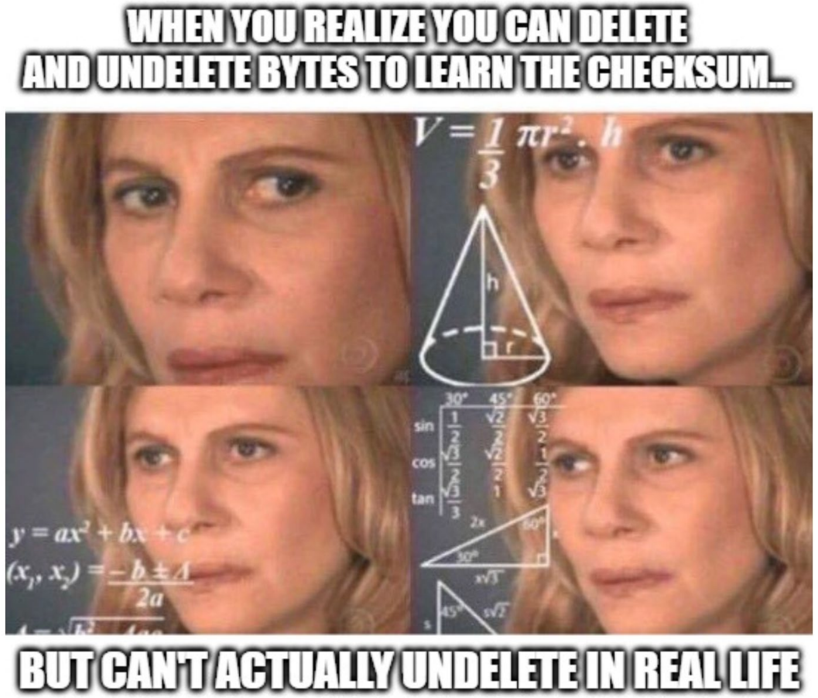


# Delete and Undelete

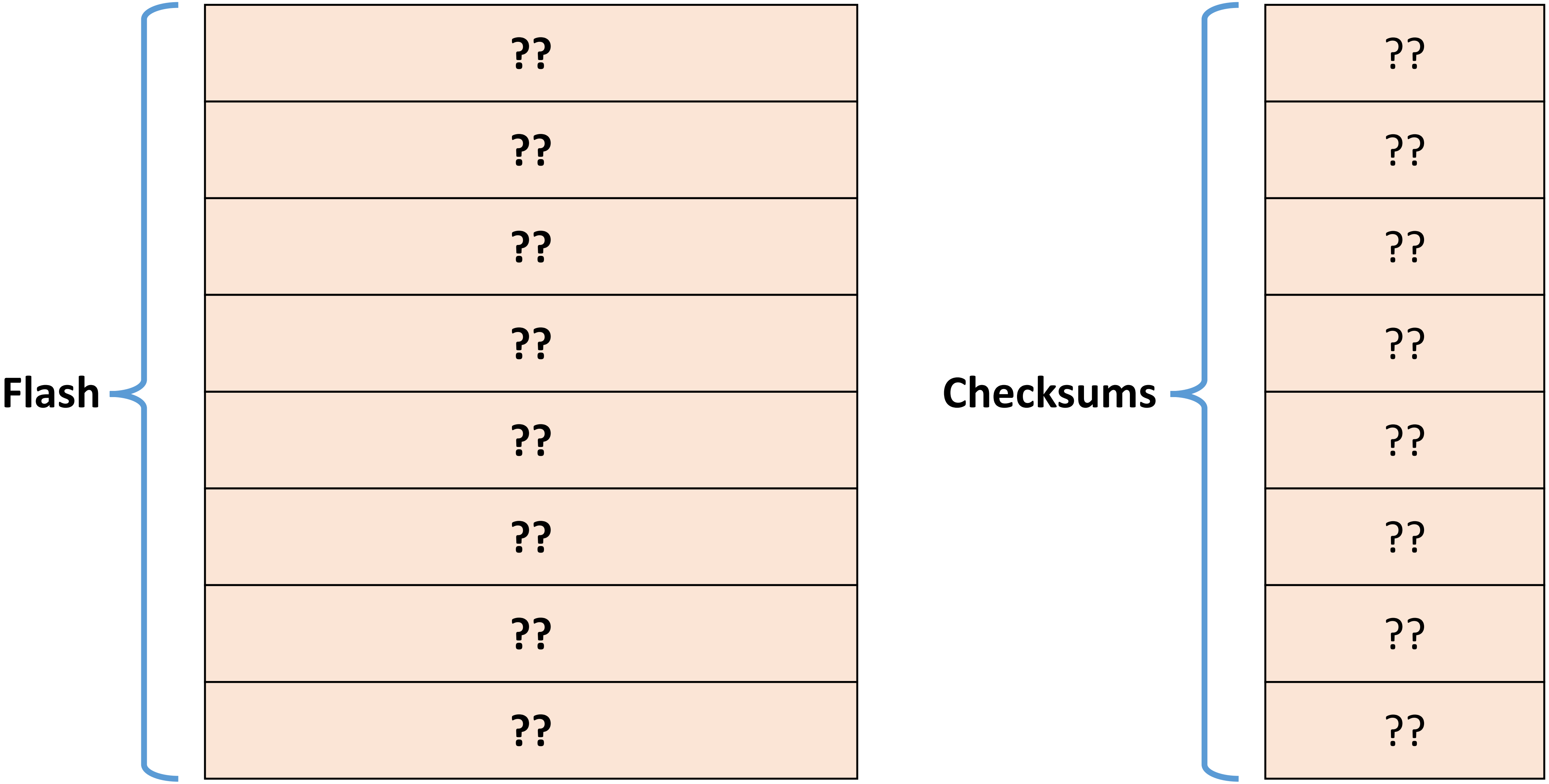
Marcus Engineering LLC

- Can delete the first 255 bytes.
  - Then only 1 is unknown.
  - Then we learn it from the checksum.

- Imagine *un*deleting the byte before it…
  - Still only 1 is unknown, now we learn the next.
  - And so on…



# Burn Before Reading



**Flash**

??

??

??

??

??

??

??

??

**Checksums**

??

??

??

??

??

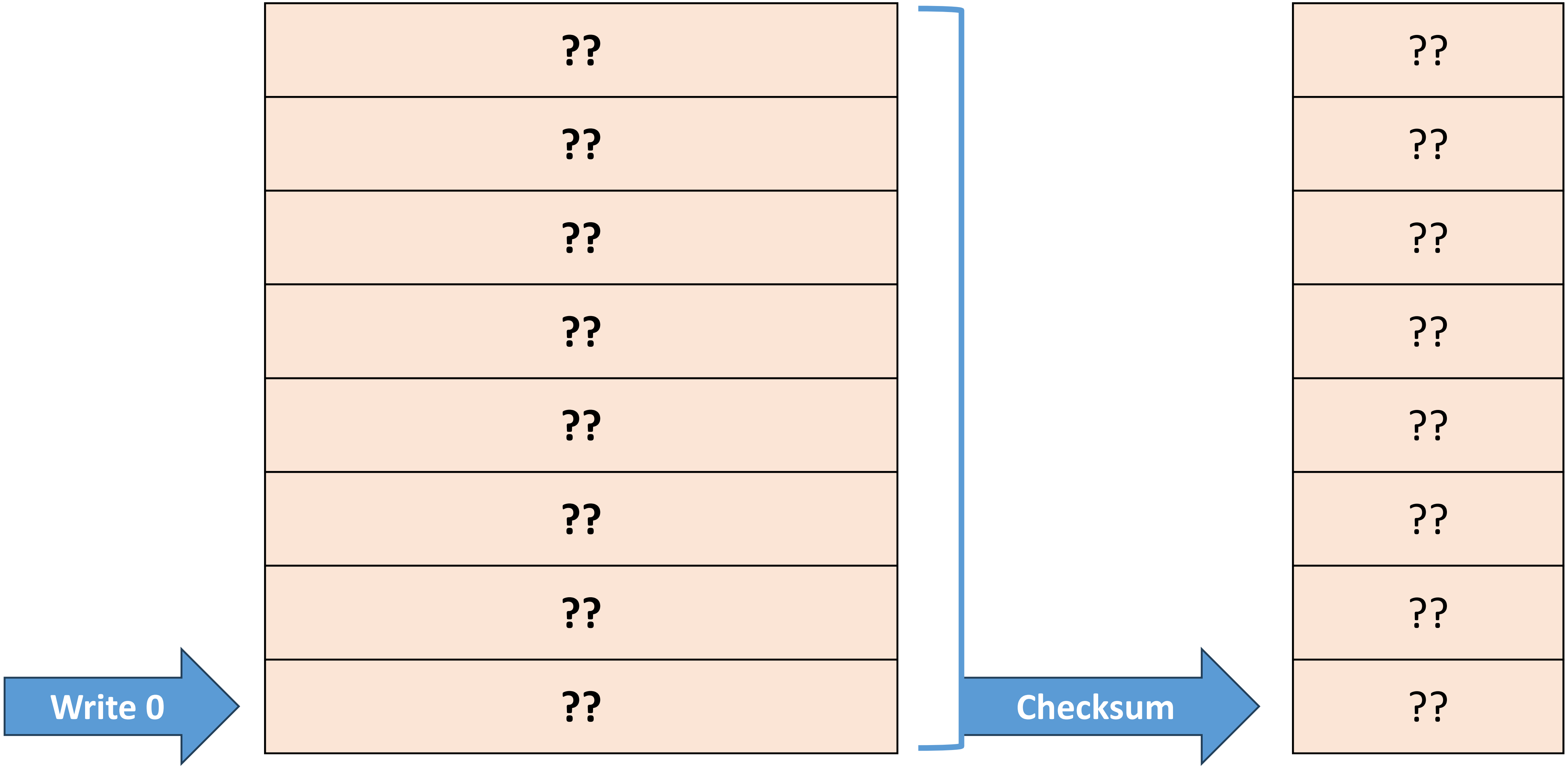
??

??

??

# Burn Before Reading

Marcus Engineering LLC



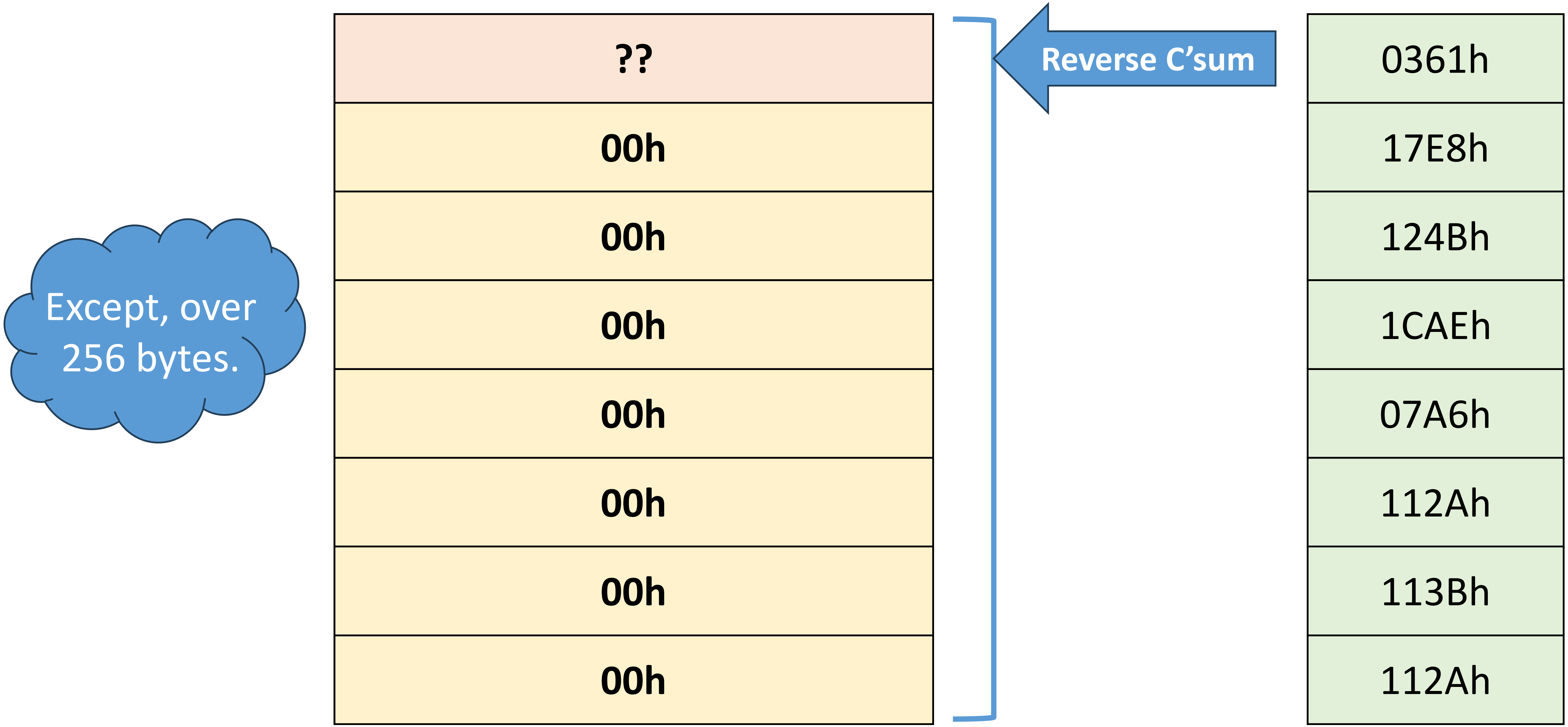
??
??
??
??
??
??
??
??

Write 0

Checksum

??
??
??
??
??
??
??
??

# Burn Before Reading



??
00h
00h
00h
00h
00h
00h
00h

Reverse C’sum

0361h
17E8h
124Bh
1CAEh
07A6h
112Ah
113Bh
112Ah

Except, over 256 bytes.

# Burn Before Reading

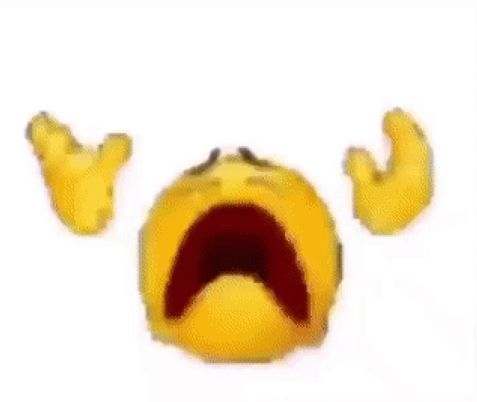
- We know the flash contents if we write it ourselves.
  - Programming command can set any 1 to a 0, and can operate on less than a full block.
  - We can progressively zero out bytes.
- **Zero a byte, read and save the checksum. Repeat.**
  - End up with 256 stored checksums and fully zeroed flash.
  - Working backwards, only 8 bits at a time are unknown, so a byte can be learned.
- Eventually, we learn the contents of the whole block!

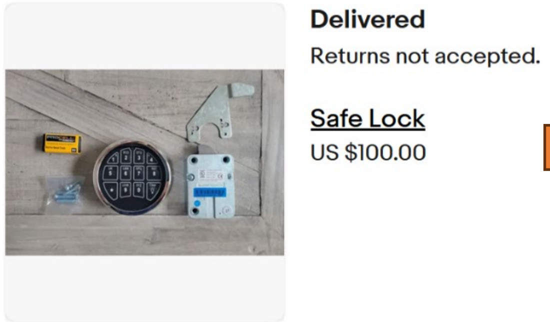
# Kind of risky…

- Only really get one shot at this.
  - Per lock, at least…
- Made our Teensy tool run the checksum three times and make sure all three match, before deleting the next byte.
- Serial output from the tool was routed to a logging Tera Term session.

# MVP, finally dumping the memory!

- It works on our test chips!
- Now to test the real thing!
- We issue the chip erase command…
  - Whoops!
  - Turns out it also erases $100…



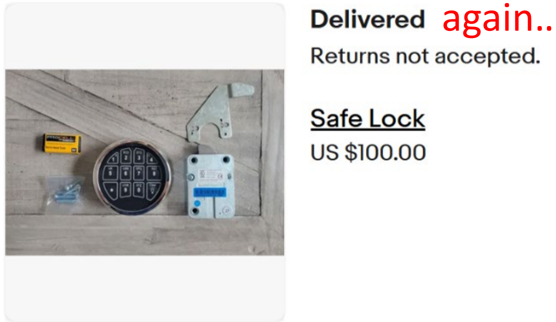


**Delivered**

Returns not accepted.

Safe Lock

US $100.00



**Delivered** again…

Returns not accepted.

Safe Lock

US $100.00

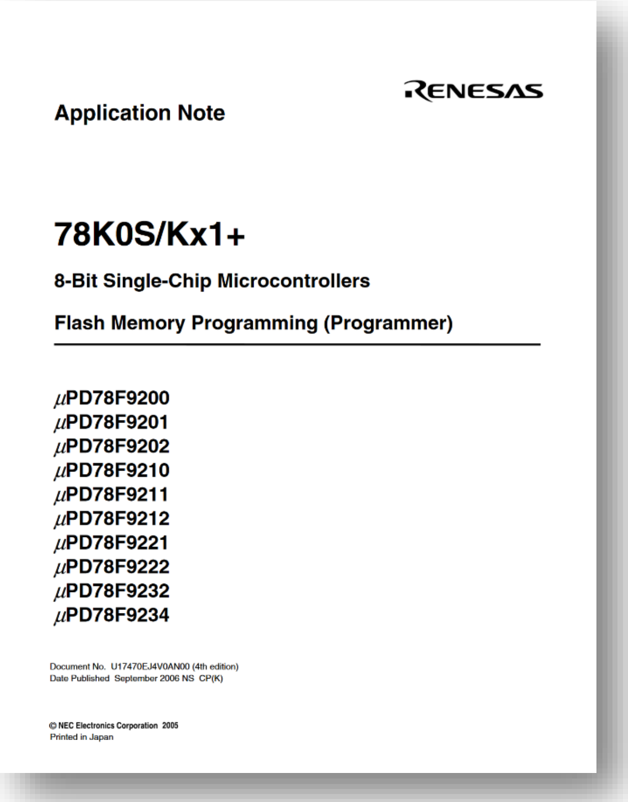
# Dumping the memory (part two)

- Received a new lock and *didn’t* run chip erase.
- And the attack… immediately doesn’t work.
  - Our chip has writing disabled…
  - Would have checked that, but, we chip erased the last one, including its security bits.

- We need a new plan…

# RTFM Part 3: Revenge of the App Note

Can’t write, right?



RENESAS

Application Note

**78K0S/Kx1+**

**8-Bit Single-Chip Microcontrollers**

**Flash Memory Programming (Programmer)**

μPD78F9200
μPD78F9201
μPD78F9202
μPD78F9210
μPD78F9211
μPD78F9212
μPD78F9221
μPD78F9222
μPD78F9232
μPD78F9234

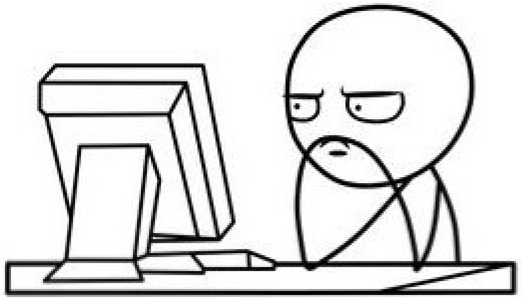
Document No. U17470EJ4V0AN00 (4th edition)
Date Published September 2006 NS CP(K)

© NEC Electronics Corporation 2005
Printed in Japan

# Time for a new plan…

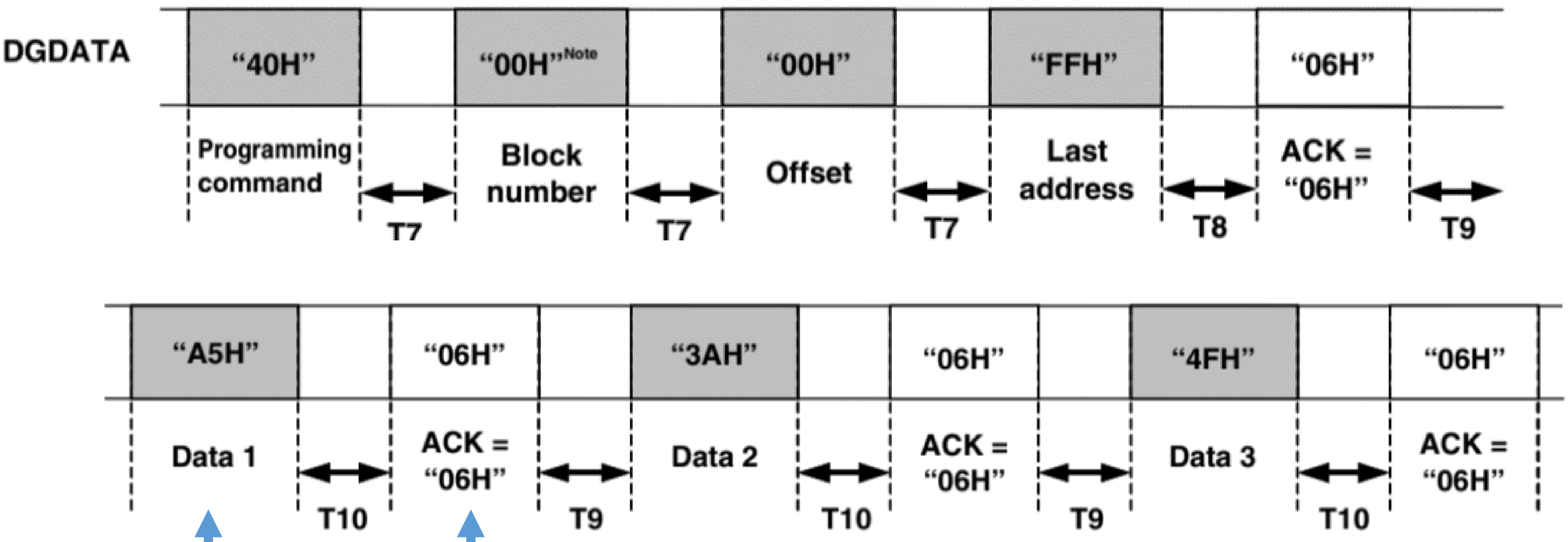
- Chip Erase
- Block Erase
- Chip/Block Erase Verify
- Programming
- Internal Verify
- Checksum
- Security Set

What other commands interact with the memory?



# What about Program?

Figure 4-20. Timing Chart of Write Processing



Data byte to be written

Ack "If Write Successful"

# What happens if we try?

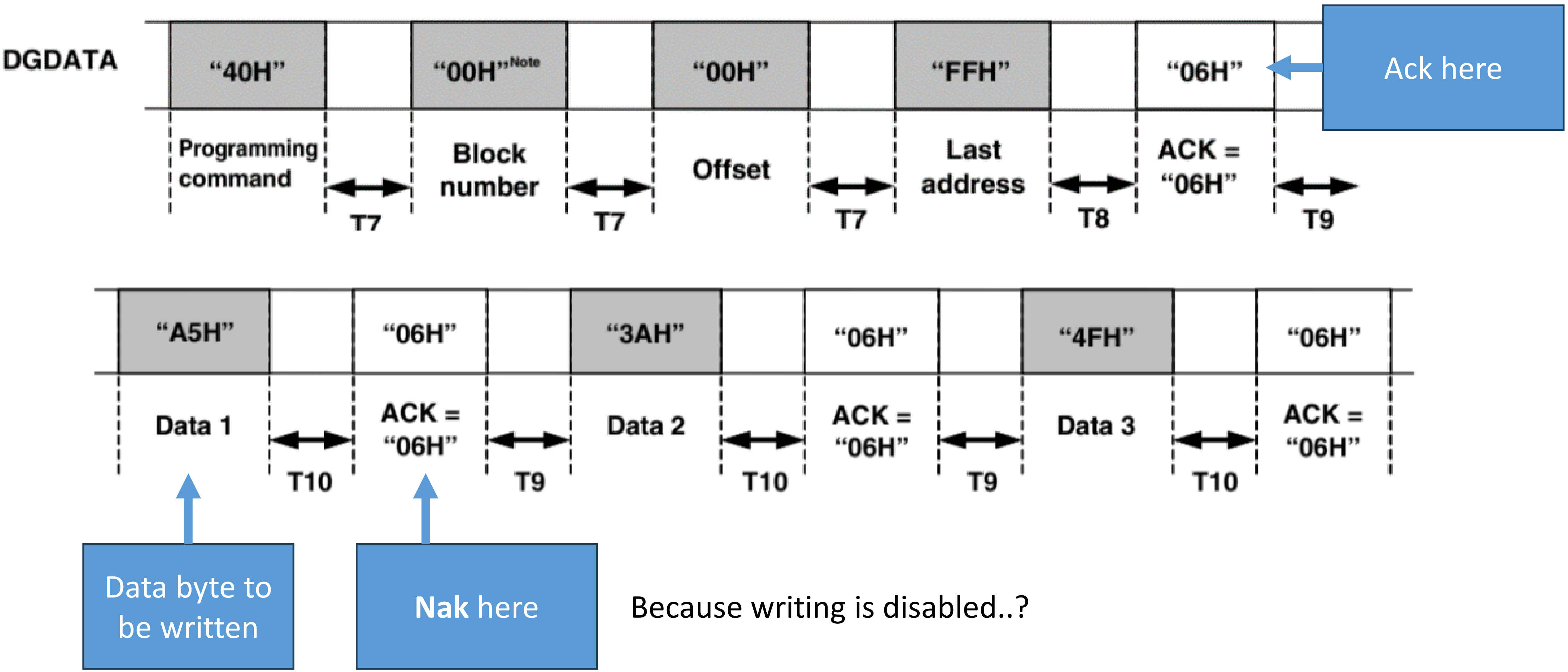


Figure 4-20. Timing Chart of Write Processing

Ack here

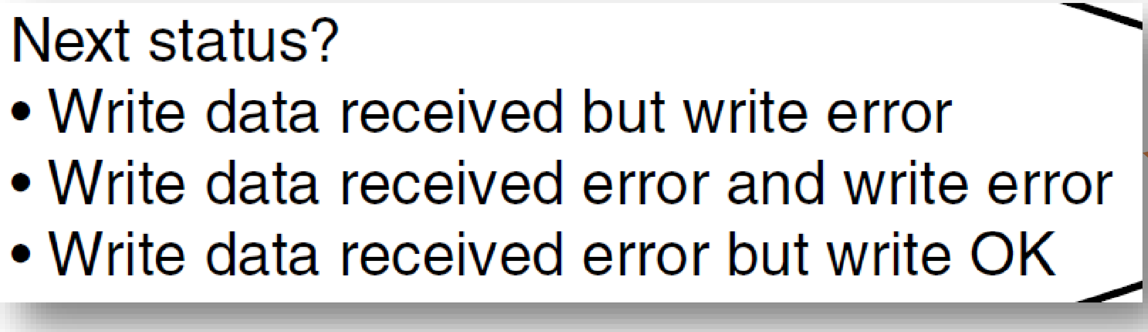
Data byte to be written

**Nak** here

Because writing is disabled..?

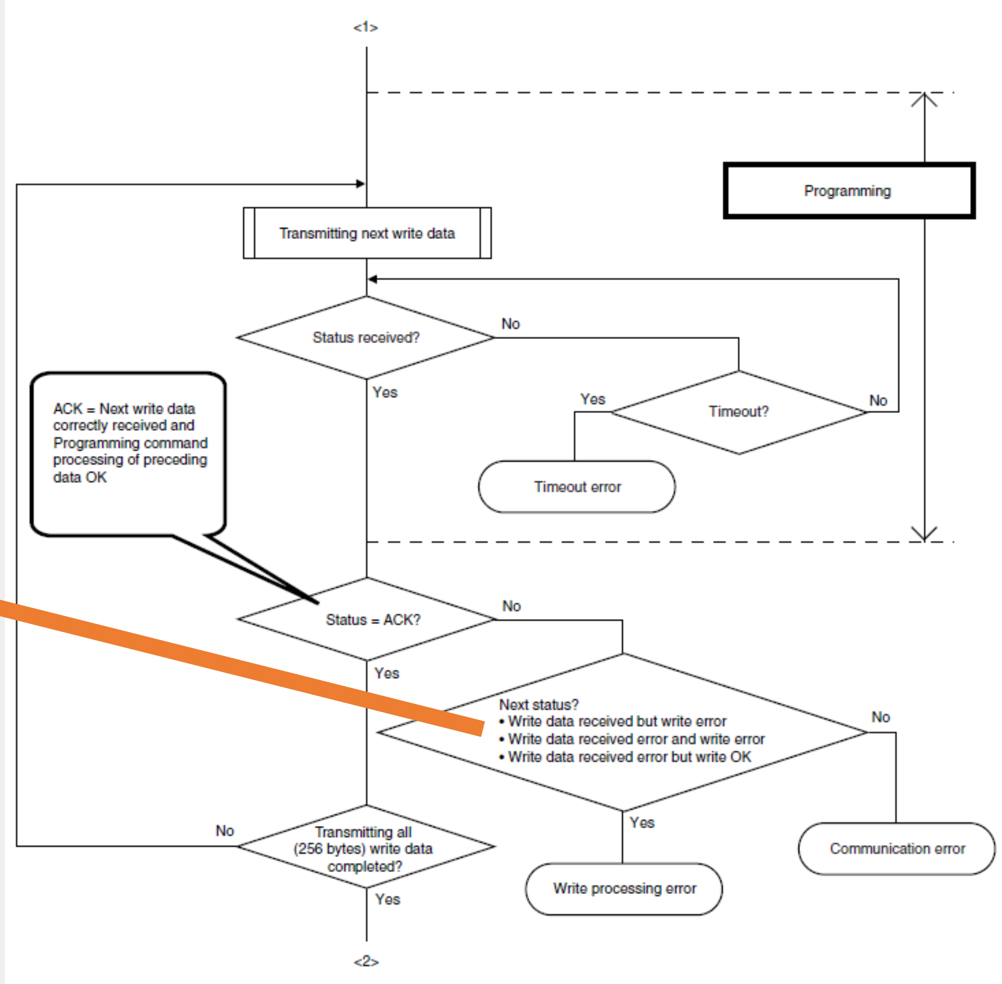
# How might this be implemented?

Let’s take a closer look at the datasheet…



Next status?
- Write data received but write error
- Write data received error and write error
- Write data received error but write OK

Figure 4-19. Write Processing Flow (2/5)



# Weirdly specific errors

Marcus Engineering LLC

- "Write data received but write error"

How does this happen?

- "Write data received error and write error"

- "Write data received error but write OK"

No error for "security bit set"…

# What’s going on inside the chip?

- IC designers are ~~lazy~~ **efficient**
- Every bit of logic takes up space, and
  **Space = Money**

- No more logic than *absolutely* required to implement what’s in the datasheet.

- How would you implement this command?

# Educated Guess

- Command has three steps

Step 1:
Receive the command

Step 2:
Receive and write a data byte to memory

Step 3:
Verify the data byte was written correctly

# How could write protect work?

Marcus Engineering LLC

You could block the programming command…

We get an ACK for the command…

Step 1:

Receive the command

Or you could block the actual programming process

Error says:
“Write data received but write error”

Step 2:

Receive and write a data byte to memory

How do we know?

Step 3:

Verify the data byte was written correctly

# Hypothesis

Step 3:

Verify the data byte was written correctly

- Step 3 must read the actual byte in flash and compare to the received data

- How can we exploit this?

# What if we program the same data?

- If we program a byte to the value we already know it is, we get an ACK!

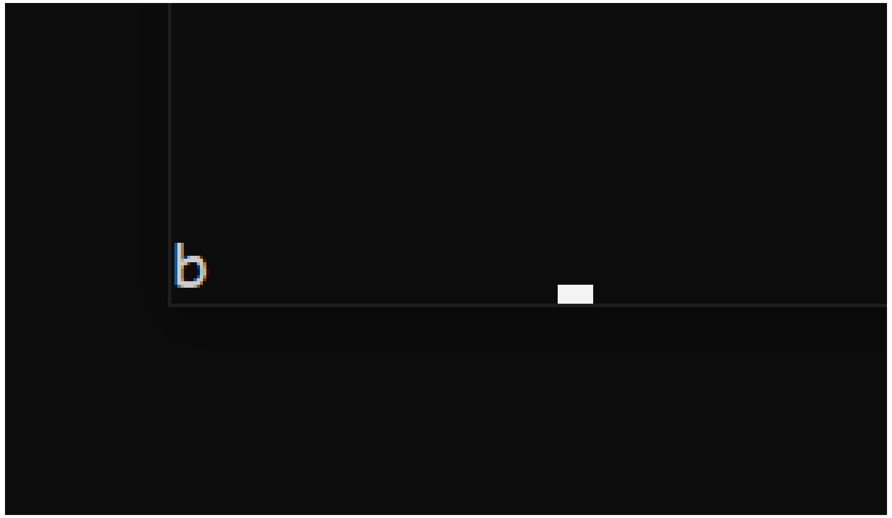
# Pouring out the Memory

# How does it work?

- Attempt to write 00h to the first address
    - If NAK then try the next value (01h, 02h, etc…)
    - If ACK then that’s the real value!

# How does it work?

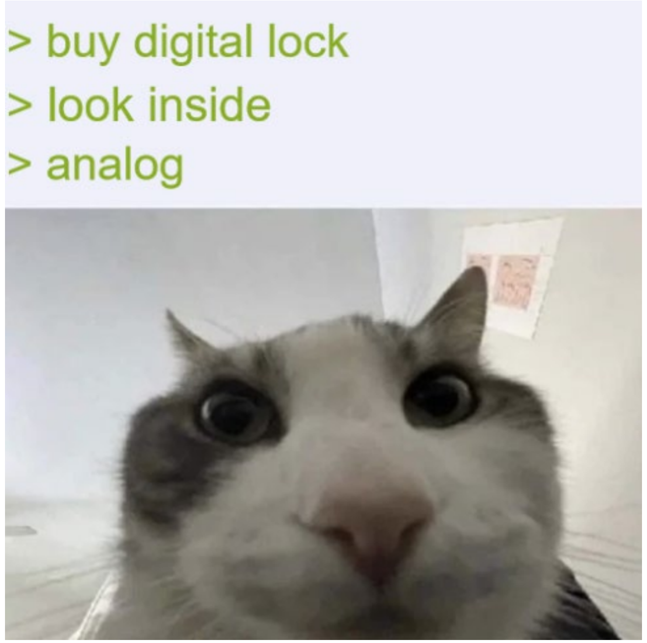
- Attempt to write 00h to the first address
  - If NAK then try the next value (01h, 02h, etc…)
  - If ACK then that's the real value!
  - Move to next byte…



gif

# Now what?

- We gotta disassemble the code…
  - To reveal the secrets inside



> buy digital lock
> look inside
> analog

- Aaaaand we gotta make our own tools…
  - No one has a disassembler for this 40 year old part

# Making ISA Machine Readable

Marcus Engineering LLC

## XCHW

**Exchange Word**
**Word Data Exchange**

**[Instruction format]** XCHW dst, src

**[Operation]** dst ↔ src

**[Operand]**

| Mnemonic | Operand (dst, src) |
|---|---|
| XCHW | AX, rp Note |

**Note** Only when rp = BC, DE or HL

**[Flag]**

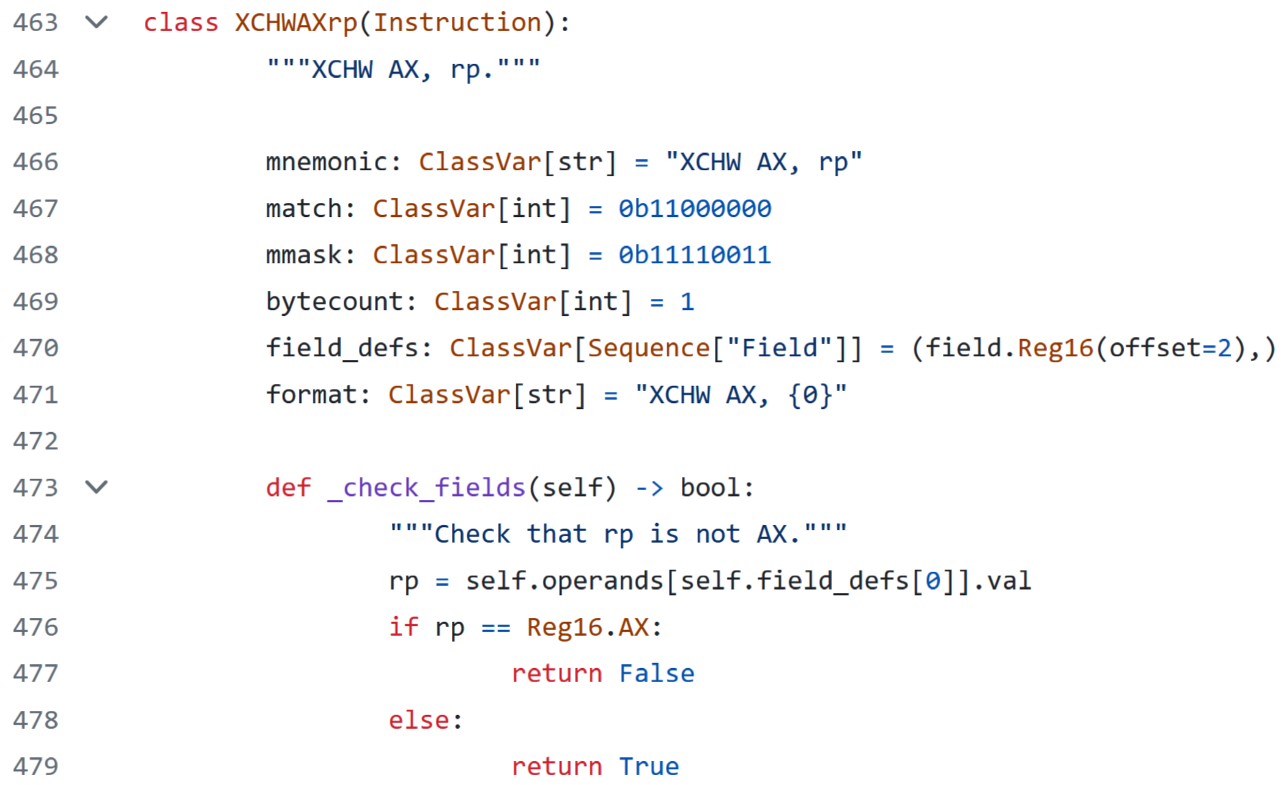
| Z | AC | CY |
|---|---|---|
| | | |

**[Description]**

- The 1st and 2nd operand contents are exchanged.

**[Description example]**

XCHW AX, BC; The memory contents of AX register are exchanged with those of the BC register.



```python
class XCHWAXrp(Instruction):
        """XCHW AX, rp."""

        mnemonic: ClassVar[str] = "XCHW AX, rp"
        match: ClassVar[int] = 0b11000000
        mmask: ClassVar[int] = 0b11110011
        bytecount: ClassVar[int] = 1
        field_defs: ClassVar[Sequence["Field"]] = (field.Reg16(offset=2),)
        format: ClassVar[str] = "XCHW AX, {0}"

        def _check_fields(self) -> bool:
                """Check that rp is not AX."""
                rp = self.operands[self.field_defs[0]].val
                if rp == Reg16.AX:
                        return False
                else:
                        return True
```

# Disassembling, then deciphering

Marcus Engineering LLC

```
.org 0082H

vec_Reset:
        MOVW AX, #FEE2H              ;0082  F0 E2 FE
        MOVW SP, AX                  ;0085  E6 1C
        CALL FUN_NOP                 ;0087  22 F6 00
        XOR A, A                     ;008A  0A 43
        MOV !FE92H, A                ;008C  E9 92 FE
        MOV !FE93H, A                ;008F  E9 93 FE
        MOVW HL, #FED8H              ;0092  FC D8 FE
        MOV B, #28H                  ;0095  0A F7 28
        MOV A, #00H                  ;0098  0A F3 00
    label_009B:
        MOV [HL], A                  ;009B  EF
        INCW HL                      ;009C  8C
        DBNZ B, $009BH               ;009D  36 FC
        MOVW HL, #0129H              ;009F  FC 29 01
        MOVW DE, #FEB6H              ;00A2  F8 B6 FE
    label_00A5:
        MOVW AX, HL                  ;00A5  DC
        CMPW AX, #0129H              ;00A6  E2 29 01
        BZ $00B1H                    ;00A9  3C 06
        MOV A, [HL]                  ;00AB  2F
        MOV [DE], A                  ;00AC  EB
        INCW HL                      ;00AD  8C
        INCW DE                      ;00AE  88
        BR $00A5H                    ;00AF  30 F4
    label_00B1:
        MOVW HL, #FE94H              ;00B1  FC 94 FE
    label_00B4:
        MOVW AX, HL                  ;00B4  DC
        CMPW AX, #FEB6H              ;00B5  E2 B6 FE
        BZ $00C1H                    ;00B8  3C 07
        MOV A, #00H                  ;00BA  0A F3 00
        MOV [HL], A                  ;00BD  EF
        INCW HL                      ;00BE  8C
        BR $00B4H                    ;00BF  30 F3
```

```c
void vec_Reset (void)
{
    uchar * src;
    uchar * dst;
    uchar cnt;

    _SP = 0xFEE2;
    *(uchar*)(0xFE92) = 0;
    *(uchar*)(0xFE93) = 0;

    // [0xFED8, 0xFEFF] = { 0 };
    dst = 0xFED8;
    cnt = 0x28;
    do {
        *(dst++) = 0;
    } while(!--cnt)

    // no effect?
    // this one is actually weird
    // init data at 0x0129.
    src = 0x0129;
    dst = 0xFEB6;
    while (src != 0x0129)
    {
        *(dst++) = *(src++);
    }

    // [0xFE94, 0xFEB5] = { 0 };
    dst = 0xFE94;
    while (dst != 0xFEB6)
    {
        *(dst++) = 0;
    }
```

# Sharing is caring

- You can find our disassembler here:
  - https://github.com/pixelfelon/78k0s-dasm
- And the flash dump code is here:
  - https://github.com/pixelfelon/78k0s-dumper

Special thanks to the Renesas PSIRT, they were very responsive to our disclosure.

# Demo!

And questions!

# The Device