Flash Techniques: Designing for System Robustness
Keith Quiring

Agenda

• Basics of flash technology
• Software Robustness
• Limitations of flash technology
• Determining errors
• Locating errors
• Eliminating errors
Flash Memory: Benefits

Flash is growing in popularity. Why?
- Non-volatile memory provides best of ROM/RAM
- Scales well with new lithography: lower cost per byte
  - Better than PROM / OTP fuses, which do not scale well
  - Better density than EEPROM
- Increasing memory sizes enable whole, large programs to be stored directly on MCU die: code security
  - Unlike ROM, no NRE to get code onto the IC
- Ease of code development: time saved!
  - Unlike OTP, no need to throw away every time program changes
  - With JTAG/emulation, no need to remove MCU from system in order to program

Flash Memory: Organization

- MSP430 flash divided into segments & blocks
  - Typical segment size: main -- 512 bytes; info -- 64 bytes
  - Blocks typically 64 bytes
Flash Memory: Technology Basics

- Flash cells can be...
  - Erased (0 → 1)
  - Written (1 → 0)
  - Read

- Erase
  - Performed on whole segments

- Write
  - Bit/byte/word boundaries
  - But, the whole block gets exposed to flash voltage

- Read
  - Evaluate the cell current against a pair of threshold currents (high/low)

Think of a flash cell as a “bucket”

FULLY ERASED

FULLY PROGRAMMED

- NOR flash cell is based on a transistor with a floating gate
  - Positively-charged floating gate results in a ‘1’
  - Negatively-charged floating gate results in a ‘0’

- High energy required to move the charge
Flash Memory: Technology Basics

- Higher voltage required, but MSP430 has an integrated charge pump (no external programming voltage required)
  - 1xx/2xx/4xx: minimum Vcc higher than for CPU (F2xx/47x: 2.2V, F4xx: 2.7V)
  - But, 5xx charge pump can work at 1.8V – same minimum Vcc as MCU
- During write/erase, charge must be applied for a specific length of time
  - Applied for a certain number of f_{FTG} cycles

<table>
<thead>
<tr>
<th>PARAMETER</th>
<th>TEST CONDITIONS</th>
<th>Vcc</th>
<th>MIN</th>
<th>NOM</th>
<th>MAX</th>
<th>UNIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vccc(ERASE)</td>
<td>Program and Erase supply voltage</td>
<td></td>
<td>2.7</td>
<td>3.6</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>f_{FTG}</td>
<td>Flash Timing Generator Frequency</td>
<td></td>
<td>257</td>
<td>476</td>
<td>kHz</td>
<td></td>
</tr>
</tbody>
</table>

FROM: MSP430FG439 datasheet

Agenda

- Basics of flash technology
- Software Robustness
- Limitations of flash technology
- Determining errors
- Locating errors
- Eliminating errors
Corruption of CPU Execution

- Code gone “off in the weeds” could inadvertently write to flash
- Causes:
  - Buggy code
  - Stack overflow
  - Vcc vs. MCLK violations

- Errant code can jump into:
  - Non-code space
  - BSL
  - User code

Built-In Protection Against Errant Flash Writes

- MSP430 implements a “flash key” (FWKEY)
  - Accessing flash control registers requires 0xA5 in the upper byte
  - If not: “flash key violation”, resetting the device (PUC) & setting KEYV
- Flash “LOCK” bit
  - Always lock flash when not writing/erasing!
  - Newer devices have separate lock bit for InfoA memory (LOCKA)
Errant Writes from Non-Code

- Consider effects of an errant jump into “empty” flash space
  - Flash not used for code/data is undefined; interpreted as code, it can cause damage
  - Check how your IDE handles this space; consider adding your own NOP fill
  - Insert a branch to 0xFFFF (RESET vector) prior to vector table
  - Device dependencies may apply (example: 2xx BSL key)

Example: F2619

- Fill unused space with NOP
- Write “0x4030 FFFE” (BR #FFFEh) at 0xFFDC and 0x1FFFFC

ATC 2008
MSP430 Advanced Technical Conference

Errant Writes from User Code

- Execution can jump into BSL code
  - Contains legitimate flash write/erase accesses
  - Relatively rare
  - See “Features of the MSP430 Bootstrap Loader” (slaa089d) for more info

- Execution can jump into user code
  - Can contain legitimate flash write/erase accesses
  - Running them at the wrong time can corrupt flash

<table>
<thead>
<tr>
<th>Memory</th>
<th>MSP430FG437</th>
<th>MSP430FG438</th>
<th>MSP430FG429</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main</td>
<td>128Kb</td>
<td>128Kb</td>
<td>128Kb</td>
</tr>
<tr>
<td>Main interrupt vector</td>
<td>64Kb</td>
<td>64Kb</td>
<td>64Kb</td>
</tr>
<tr>
<td>Main code memory</td>
<td>64Kb</td>
<td>64Kb</td>
<td>64Kb</td>
</tr>
<tr>
<td>Information memory</td>
<td>64Kb</td>
<td>64Kb</td>
<td>64Kb</td>
</tr>
<tr>
<td>Boot memory</td>
<td>Size</td>
<td>4M</td>
<td>4M</td>
</tr>
<tr>
<td>ROM</td>
<td>4M</td>
<td>4M</td>
<td>4M</td>
</tr>
</tbody>
</table>

Texas Instruments
**Errant Writes: Prevention**

- Test your code well, to eliminate bugs
- Help protect against Vcc vs. MCLK violations by:
  - Ensure proper Vcc before changing MCLK
  - Using proper technique when changing DCO speed
  - Can also occur from application-specific transient effects
- To protect against the bugs you didn’t find, use known software robustness techniques
- See “MSP430 Software Coding Techniques” (slaa294a) for more tips on software robustness

---

**Agenda**

- Basics of flash technology
- Software Robustness
- Limitations of flash technology
- Determining errors
- Locating errors
- Eliminating errors

---

**ATC 2008**

MSP430 Advanced Technical Conference
Flash Limitations: Lifetime Retention

- Charge of a programmed cell leaks very slowly over time, causing a 1 to eventually become a 0
- Datasheet indicates >100 years at 25°C
  - Common practice in the industry to spec flash this way
  - Long enough for most apps
- Additional rigorous tests show actual MSP430 flash retention at 25°C is >1000 years!

(from: MSP430FG439 datasheet)

Flash Limitations: Heat

- Lifetime retention is reduced at high temp
- MSP430 flash testing still shows good retention at extreme temps:
  - 27 years at 85°C
  - 10 years at 105°C
  - These values are for continuous exposure -- most apps involve only part-time exposure

(from: “Understanding MSP430 Flash Data Retention” (slaa392)
**Flash Limitations: Improper Vcc**

- 1xx/2xx/4xx flash has minimum Vcc higher than for CPU
- Flash write/erase when Vcc < minimum can lead to corrupted data
  - Whether initiated by CPU, JTAG, or BSL
- Help prevent this by...
  - Implementing Vcc checks, using SVS or ADC
  - Adequate bulk/decoupling caps on Vcc
  - ESD protection at vulnerable points in the system

<table>
<thead>
<tr>
<th></th>
<th>2xx</th>
<th>1xx/4xx</th>
<th>5xx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supply voltage during flash memory programming, Vcc</td>
<td>VDDCC - VDDCC - VDD (see Note 1)</td>
<td>2.2</td>
<td>3.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vcc</td>
<td>Supply voltage during program execution and flash programming, A[2:0] = D[2:0] = 0</td>
<td>1.8</td>
<td>3.6</td>
</tr>
</tbody>
</table>

**Flash Limitations: Improper F_{FTG}**

- F_{FTG} must be 257-476kHz
  - Too fast: voltage not applied long enough – insufficiently charged
  - Too slow: voltage applied too long – “stresses” the cells, which may lead to reduced lifetime retention
  - Check your code, make sure value is chosen properly
- Derived from user-defined system clocks, so choose proper dividers
  - If F_{FTG} is DCO-derived & software changes DCO speeds, be sure to adjust dividers as well
- Never write/erase flash if source clock is in fault or not yet stabilized
  - Prevent this by writing NMI handlers for oscillator faults
- 5xx: F_{FTG} is automatic – no need to configure
Flash Limitations: Endurance

- “Endurance” refers to the number of write/erase cycles a flash cell can endure before failing
- 100k cycles: high enough for most applications
- However, it’s possible to write code that hits the limit
  - Example: data logging that erases flash 25 times a day for 11 years
  - If frequent data writes required, spread it over a wider area of flash to reduce the number of erases

<table>
<thead>
<tr>
<th>PARAMETER</th>
<th>TEST CONDITIONS</th>
<th>V_CC</th>
<th>MIN</th>
<th>NOM</th>
<th>MAX</th>
<th>UNIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program/Erase endurance</td>
<td></td>
<td></td>
<td>10^4</td>
<td>10^5</td>
<td></td>
<td>cycles</td>
</tr>
</tbody>
</table>

Flash Limitations: Cumulative Time

- Writing any part of a block applies flash voltage to entire block
- “Cumulative programming time” refers to time a cell is applied to flash voltage between erase cycles
- Won’t hit limit with byte/word writes; only bit writes
  - Faster F_Tq helps
  - No byte/word can be written to more than twice between erasures
- Exceeding cumulative time results in “write disturb”
  - Unintended bits may get programmed as well

<table>
<thead>
<tr>
<th>PARAMETER</th>
<th>V_CC</th>
<th>MIN</th>
<th>NOM</th>
<th>MAX</th>
<th>UNIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_CUT</td>
<td>2.7 V/3.6 V</td>
<td>10</td>
<td></td>
<td></td>
<td>ms</td>
</tr>
</tbody>
</table>

ATC 2008
MSP430 Advanced Technical Conference
**Agenda**

- Basics of flash technology
- Software Robustness
- Limitations of flash technology
- Determining errors
- Locating errors
- Eliminating errors

**Determining Errors with Checksums**

- A **checksum** is a small value derived from a much larger memory block
  - Example: four byte checksum for 2K block
- Any change in the block changes the checksum
  - Or at least – does so with an extremely high probability
Determining Errors with Checksums

- Update the checksum at powerup & when block is written/erased
- Or use checksum feature in IDE, which calculates checksum & places in memory at link time
- After period of no intended changes, re-calculate checksum & compare with stored value

What if Checksums Don’t Match?

- Means one of two things…
  - Contents of flash block have changed, or…
  - Contents of stored checksum has changed!
  - But – statistically more likely that the block contents changed
- Potential causes
  - Code gone “off in the weeds”, writing/erasing flash data, or…
  - One of the flash cell mechanisms discussed earlier
- Is it a code segment?
  - Can’t trust this code anymore
  - Might choose to display error to user and shut down gracefully
  - Checksum helps avoid a walking-wounded state
What if Checksums Don’t Match?

- Is it a data segment?
  - Can’t trust this data anymore
  - Checksum helps avoid bad results
- Might be able to use a software ECC algorithm to recover data
- Consider storing checksums in more than one location
  - Redundancy helps avoid situation where checksum contents have been altered

Checksum Algorithms

- Many possible algorithms & derivatives
  - CRC-8, CRC-16, CRC-32
  - CRC-16-CCITT
  - Fletcher’s checksum / Adler-32
- CRC types are the most popular
  - Detects any single error burst of <n bits (where n is the size of the checksum)
- Based on division of polynomials, implemented with XORs and bit-shifts
  - Essentially treats block as one large binary number, dividing it by a large prime number & computing the remainder
- In true mathematical form, the algorithm is bit-based
  - Inefficient for an MCU
- But, table-based algorithm allows byte operation
  - Saves cycles and power
**CRC Math**

<table>
<thead>
<tr>
<th>Name (Protocols)</th>
<th>Polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRC-16</td>
<td>(x^{16} + x^{12} + 1)</td>
</tr>
<tr>
<td>CRC-CITT (EDLC, HDLC, X.25)</td>
<td>(x^{16} + x^{12} + x^5 + 1)</td>
</tr>
<tr>
<td>CRC-32 (Ethernet)</td>
<td>(x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^7 + x^4 + x^3 + x^2 + x + 1)</td>
</tr>
</tbody>
</table>

**CRC polynomial division built with bit-shifts and XORs**

**CRC Algorithm in Code**

```c
unsigned int crc32(unsigned char *message)
{
    int i, j;
    unsigned int byte, crc;
    i = 0;
    crc = 0xFFFFFFFF;
    while (message[i] != 0)
    {
        byte = message[i];       //Get next byte
        byte = reverse(byte);    //32-bit reversal
        for (j = 0; j <= 7; j++) //Do eight times
        {
            if ((int)(crc ^ byte) < 0)
                crc = (crc << 1) ^ 0x04C11DB7;
            else crc = crc << 1;
            byte = byte << 1; // Ready next msg bit.
        }
        i = i + 1;
    }
    return reverse(~crc);
}
```

Implementation of bitwise method isn't trivial.
CRC Table Algorithm

- Bitstream broken into bytes, XOR-ing bytes at a time instead of bits
- Each XOR product referenced to a lookup table, eliminating bitwise shifts & XORs
- 256-element table is a common choice
- Table can be built at compile-time or run-time

```c
unsigned long crc32(
    unsigned long crc,  // initial value
    TBL_MEM unsigned long *table,  // table
    unsigned char *pbuffer,  // data block
    unsigned int length )  // length of block
{
    while(length--)
        crc = table[((crc >> 24) ^ *pbuffer++)] ^ (crc << 8);
    return(crc ^ CRC32_FINAL_XOR);
}
```

Memory Block Size?

- Are some areas of memory more critical than others?
  - Might want to checksums there more often – arrange blocks accordingly
- Large-block checksums take time; will this disrupt the application?
  - Divide and conquer
- Is recovery possible?
  - Marginal read mode present?
  - Software ECC employed?
  - Then smaller chunks are better; helps locate the error
When to Calculate/Store/Compare?

- Calculate and store...
  - Upon initial execution
  - Whenever contents deliberately changed

- Calculate and compare...
  - Before critical operations
  - Daily/weekly/monthly/??
  - More often if your application is particularly sensitive to failure (i.e., medical, automotive)
  - More often if exposed to heat
  - After watchdog resets (code might have gone “into the weeds”, possibly corrupting flash before resetting the part)

5xx CRC16 Module

- CRC16 hardware module: new to 5xx
- Uses 16-bit CRC-CCITT method
- Functions like MPY module
  - Simply write the data to CRC16 register; CRC updated automatically
- Can be written with MOV instruction or with DMA
  - DMA is lower power: configure DMA to run block to CRC16, then go to sleep
  - Wake up and compare value to old one

```c
#define CRC16RES 0xFFFF;
for(i = blockStartAddr; i < endBlockAddr; i++)
    CRCDI = i;
if(oldCRC != CRC16RES)
    handleBadChecksum;
else oldCRC = CRC16RES;
```
Flash Memory: Technology Basics

- Flash cells can be...
  - Erased (0 → 1)
  - Written (1 → 0)
  - Read
- Erase
  - Performed on whole segments
- Write
  - Bit/byte/word boundaries
  - But, the whole block gets exposed to flash voltage
- Read
  - Evaluate the cell current against a pair of threshold currents (high/low)

Think of a flash cell as a “bucket”

Agenda

- Basics of flash technology
- Software Robustness
- Limitations of flash technology
- Determining errors
- Locating errors
- Eliminating errors
Locating Errors: Marginal Read Mode

- On newer MSP430’s
  - F261x, F24x, F47x, 5xx, and all future devices
- When active, makes the read thresholds “more extreme”
- Subsequent reads use the new threshold
- Two sub-modes
  - MRG1 detects insufficiently-erased (1) cells
  - MRG0 detects insufficiently-programmed (0) cells

Locating Errors: Marginal Read Mode

- Cell is exposed if it’s not fully charged/discharged
  - Insufficiently-programmed (0) cells may read as 0 when MRG disabled, but read as 1 when enabled
  - Insufficiently-erased (1) cells may read as 1 when MRG disabled, but read as 0 when enabled
  - Sufficiently-charged/discharged cells show same results whether MRG enabled or disabled

MRG1 = 0

MRG1 = 1
Locating Errors: Marginal Read Mode

- **Cell Charge Level**
  - Fully erased
  - Non-MRG
  - MRG0
  - MRG1
  - MRG threshold
  - Non-MRG threshold
  - MRG0 threshold

Modes return different values
All modes return the same value

Limitations of Marginal Read Mode

- If a bit reads differently in non-MRG mode than in either MRG1 or MRG0…
  - The bit is not strong and should be re-programmed
- If a bit reads same in non-MRG as in either MRG1 or MRG0…
  - Then the cell is OK…
  - OR, the cell might be weak enough that it returns the same wrong value in both MRG/non-MRG modes ("extremely weak")
- If cause of an “extremely weak” cell was bad programming conditions…
  - A “verify” immediately after programming will detect it
  - But prevention is better than detection: eliminate bad programming conditions!
- But if it was caused by lifetime retention leakage…
  - Performing MRG with sufficient periodicity can detect such a failure
**Basic MRG Algorithm**

- Comparing against an old checksum ensures that even extremely weak cells will be caught
- Code must be executed from RAM when MRG active

1. Copy this code to RAM and resume execution from there
2. Set MRG1
3. Calculate checksum & compare to stored non-MRG value
   - DIFFERENT
   - SAME
4. Clear MRG1 and set MRG0
5. Calculate checksum & compare to stored non-MRG value
   - DIFFERENT
   - SAME
   - Clear MRG1 and MRG0.
6. Flash block is OK. Return execution to flash
7. Calculate checksum & compare to stored non-MRG value
   - DIFFERENT
   - SAME
8. Weak cell found. Clear MRG1 and MRG0; resume execution from flash
9. Calculate checksum & compare to stored non-MRG value
   - DIFFERENT
   - SAME
   - Recoverable error
10. Calculate checksum & compare to stored non-MRG value
    - DIFFERENT
    - SAME
    - Cell is extremely weak, irrecoverable error

**Agenda**

- Basics of flash technology
- Software Robustness
- Limitations of flash technology
- Determining errors
- Locating errors
- Eliminating errors
Fixing Errors with Flash Refresh

• If MRG identifies an insufficiently-charged bit….
• Then the bit can be “fixed”
• Be sure the erase/write are done under proper Vcc/clock conditions
• If power is interrupted between flash erase and re-writing, segment is lost

MRG: When to Run?

• Before the non-MRG checksum reveals an error!
  – After this, the cell is too corrupted for MRG to help
• Other than this, the same guidelines apply as for checksum calculate/compare
Summary

- Best defense against flash corruption is proper programming conditions
- Test systems thoroughly for bugs
- Add robustness features
- Consider lifetime retention, if operating at high temps
- Check programming setup at manufacturing
- Some conditions can be hard to guarantee 100%
- Therefore, checksums & marginal read mode provide additional layer of protection

Thank you