intel®

AP-325

APPLICATION NOTE

Guide to First Generation Flash Memory Reprogramming

APPLICATIONS ENGINEERING STAFF

March 1994

Order Number: 292059-002

Information in this document is provided in connection with Intel products. Intel assumes no liability whatsoever, including infringement of any patent or copyright, for sale and use of Intel products except as provided in Intel's Terms and Conditions of Sale for such products.

Intel retains the right to make changes to these specifications at any time, without notice. Microcomputer Products may have minor variations to this specification known as errata.

*Other brands and names are the property of their respective owners.

†Since publication of documents referenced in this document, registration of the Pentium, OverDrive and iCOMP trademarks has been issued to Intel Corporation.

Contact your local Intel sales office or your distributor to obtain the latest specifications before placing your product order.

Copies of documents which have an ordering number and are referenced in this document, or other Intel literature, may be obtained from:

Intel Corporation
P.O. Box 7641
Mt. Prospect, IL 60056-7641

or call 1-800-879-4683

GUIDE TO FLASH MEMORY REPROGRAMMING

CONTENTS PAGE

CONTENTS PAGE

INTRODUCTION TO REPROGRAMMING

You Are in Control

Rewriting any type of memory requires hardware or software control. Traditional EEPROM designers combined all control functions into each chip's periphery. This provided a highly functional chip but at a high price. On the other hand, DRAM designers provided a bulk memory with little integrated peripheral circuitry. Each system designer then accommodated the DRAM with external refresh signals and learned quickly that failure to refresh yielded non-functioning memory boards. Initially, software drivers controlled DRAM refresh; today controllers provide the same function.

Similarly, early disk drives required every user to write software to manipulate drive head movement. Failure to follow drive specifications and algorithms caused irreversible head crashes. Leading-edge engineers faced these challenges and triumphed, as evidenced by the sophisticated systems available today.

Since 1988, thousands of engineers have written software to direct flash memory reprogramming. With first generation flash memories, one sends a control signal to a device to begin and end programming or erasure. It is a simple process implemented on more than 40 million units, however care must be taken. If algorithms are not properly followed, a device may be rendered inoperable. This document discusses proper software and debug technique, which yields dependable first generation flash memory operation. First generation products include the 28F256A, 28F512, 28F010 and 28F020. All second and third generation Intel Flash Memories contain automated program and erase routines.

FUNDAMENTALS OF FLASH MEMORY OPERATION

Adaptive vs Brute Force Algorithms

Many designers use EPROMs regularly. Few consider the programming algorithms because the PROM programmer vendors take care of that function.

Two types of algorithms are in use today:

- Adaptive Algorithms
- Brute Force Algorithms

Adaptive algorithms such as Intel's Quick-Pulse Programming and Quick-Erase algorithms reduce programming time. A feedback mechanism recognizes when each byte has been programmed sufficiently. You may ask how is the point of sufficiency determined? One simply adds the net effects of V_{CC} and temperature variations, and superimposes on those factors the normal EPROM charge leakage to obtain the answer. The next question is how can these factors be checked?

NOTE:

EPROM and EEPROM charge leakage occurs over a very long time—typically 100 years. Reliability papers often discuss charge leakage in terms of the memory's data retention characteristic.

If you look at EPROM programming algorithms, you will notice that V_{CC} is elevated during programming. The elevated V_{CC} acts as the feedback mechanism for the adaptive algorithm. Reading the device and checking for program completion is called verification, or margining. (One is checking the margin to V_{CC} fluctuations.) For example, if the part can be verified at 6.25V, then it can withstand the fluctuations and normal charge leakage.

During the past few years most major EPROM manufacturers have converted to adaptive algorithms. The algorithm loops back and programs a byte again if the first program operation does not verify at the elevated voltage.

Brute force algorithms simply program each byte multiple times, typically with long program durations. This type of algorithm has no in-system margin verification. That is they *assume but never verify* program margin to the typical environment effects.

*Many flash memories that specify **a brute force** algorithm may fail to retain data for 10 years. Additionally, they may not read the data correctly even at specified V_{CC} and temperature extremes.*

Intel's flash memory program and erase algorithms are both adaptive. They offer margin verification without requiring users to elevate V_{CC} in-system. When issued a command to program verify, the memory's command register logic taps an internally-generated elevated V_{CC} from the user-supplied external V_{PP} (12V). This is why it is essential that you provide the specified V_{PP} voltage and follow the given adaptive algorithms. Intel's adaptive algorithms, combined with the command register architecture, assures reliable code and data storage and dependable system operation.

Figure 1 shows an example of an adaptive byte programming algorithm. Appendix A compares the algorithm in Figure 1 to a brute force approach.

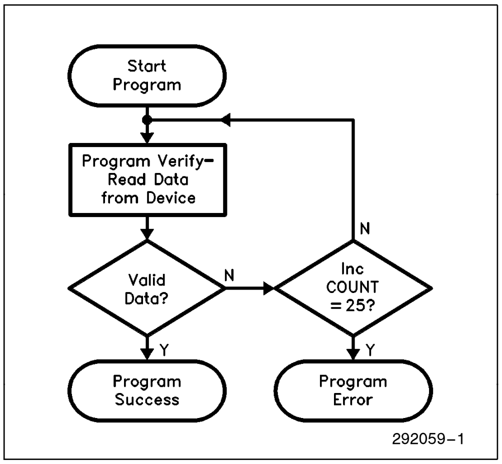


Figure 1. The flow chart shows the fundamental nature of an adaptive algorithm. Based on the outcome of program verification, the flow may loop back for another program operation.

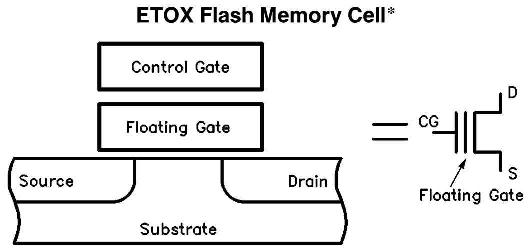
Moving Charge and Other Factors You Should Know

This section discusses the mechanics of flash memory programming. For most system designers, transistor-level discussions were last heard in college. We may recall that DRAM consists of a storage capacitor and a transistor. We remember this clearly because failure to refresh that capacitor causes systems to malfunction. **In like fashion, one should understand the fundamentals of flash memory reprogramming. The understanding will enable error-free memory operation and reliable system performance**.

In simplest terms, each data bit equates to a memory cell. Intel's flash memory uses one transistor per cell with the smallest possible architecture. This delivers the lowest cost per bit and highest capacity, levering system software (rather than bulky, complex cells) for reprogramming control.

Figure 2 shows a simplified cross section of Intel's flash memory transistor. Note the structure; the cell is a stacked gate MOS transistor. An isolated floating gate stores the memory charge. The floating gate consists of a layer of (conductive) polysilicon surrounded by (non-conductive) oxide layers.

On a DRAM cell, each transistor connects to a capacitor which stores the memory charge. The major difference between flash memory and DRAM derives from their cell structure. The DRAM cell loses its charge if not refreshed within a few milliseconds. On the other hand, the flash memory floating gate maintains its charge for typically 100 years. The structure is isolated and insulated by the field and gate oxides—hence the name "floating" gate.



*Patented Intel Processes

Figure 2. Simplicity of design assures increasing densities, manufacturability and reliability. These are the attributes that drive mainstream memories.

CONTRARY TO INTUITION
CHARGE = DATA "0"
NOT DATA "1"

PROGRAMMING:
ADDS CHARGE TO FLOATING GATE
→ DATA = 0

ERASURE:
REMOVES **SOME** CHARGE FROM FLOATING GATE
→ DATA = 1

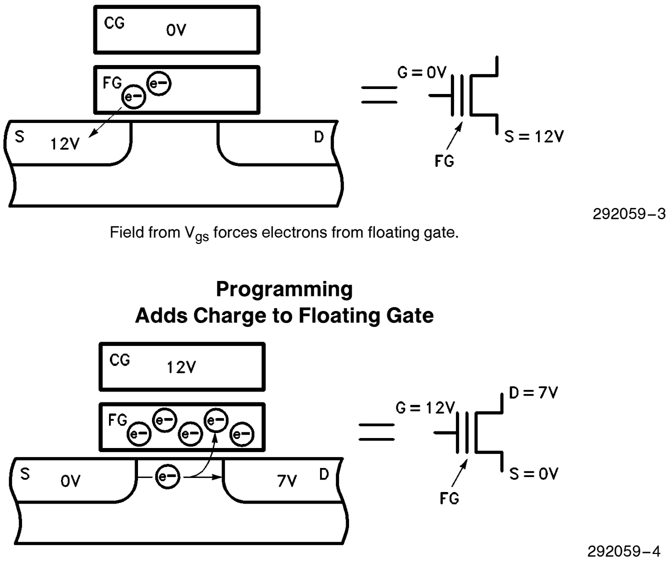
PROGRAMMING DATA WITH MIXED 0s and 1s:
→ ONLY DATA "0" BITS GET CHARGED
→ DATA "1" BITS REMAIN UNCHARGED

Changing the memory contents is simple. Figure 3 shows two memory cells—one being erased and one being programmed. Erasure removes charge from all bits simultaneously. Programming adds charge to selected bits. During erasure, not all charge is removed. The erase verify operation tells the system when enough charge has been removed. At that point, the flash memory behaves like a U.V.-erased EPROM.

Removing too much charge by erasing too long renders the memory unprogrammable. Excessive erasure lowers the cell threshold to the point where the transistor is always on and always reads data "1". (Recall that the cell threshold, V_t, determines when the transistor turns on or off.) You must control the erase timing within the algorithm specifications on first generation flash memories.

A second erase consideration relates to the first. Prior to erasing the chip, you must blanket program all bytes to data 00h, regardless of the previous data. This step equalizes the charge on all transistors.

If you skip this step and proceed directly to erasure, an interesting thing happens. Consider a typical byte programmed with data 0AAh (1010 1010b). While programming this data, bits with data "1" remain erased (charge removed), and bits with data "0" are programmed (charged added). Following programming, normal read operations sense whether a memory transistor has more or less charge and drives the outputs accordingly.



Field from V_{gs} forces electrons from floating gate.

$V_{gs} > V_t$ turns transistor on creating source-drain channel (I_{ds});
Field from V_{gd} attracts electrons from channel current towards floating gate.

Figure 3. Flash memory cells during erasure and programming. Note the movement of charge on and off the floating gate. The charge adjusts the cell threshold, which tells the outputs whether a bit (transistor) is on or off.

Erasure then removes charge from all bits. The bits that have had charge added (data "0") have some quantity of charge removed; bits with less charge (data "1") have charge removed as well. This is akin to excessively erasing the data "1" bits. Pre-programming all bits to data "0" equalizes the charge which allows for controlled, uniform erasure of all bits in the device (i.e., all 1,048,576 bits in a 28F010).

The sections entitled "Margin for Error" and the "Erase Algorithm Interpreted, Program all Bytes to 00h" discuss this concept in greater detail.

ERASURE—THE GOLDEN RULE

Erasure removes charge from all memory cells in parallel. This lowers the cells' threshold voltages from the programmed level (6.5V) below V_{CC} to the erased level (3.2V) **The device continues erasing until told to stop by the verify command or until the integrated stop-timer counts down.**

Margin for Error

Allowing erasure to continue too long depletes the charge in floating gates. So you ask—how long is too long? Figure 4 shows the margin for error of a typical device. Following the algorithm would have stopped erasure after 1 second. Cell depletion occurred after 10 seconds giving a 10x margin for error. This 10x margin exists if the erased cell erases in 1 second or 10 seconds (i.e., within the algorithm limits). This chart shows one typical example where the device happened to take 1 second to erase.

Flash memory has generous margin for error over the stopping point defined by the algorithm. The stopping point is defined as the point when all bytes in the chip verify to FFh data. The erase operation duration (Twhwh2) is specified at 10 ms ± 500 μs. Five hundred microseconds offers substantial allowance for system latency during erasure and even for slop in the timer generation. Processors or controllers can execute many lines of code in 500 μs, and the margin for error simply adds another guardband.

Proper software and system design will never rely on the additional margin for error. Remember, you control the program code and system operation during erasure. Once you have fully debugged your driver code, the issue of software control disappears entirely.

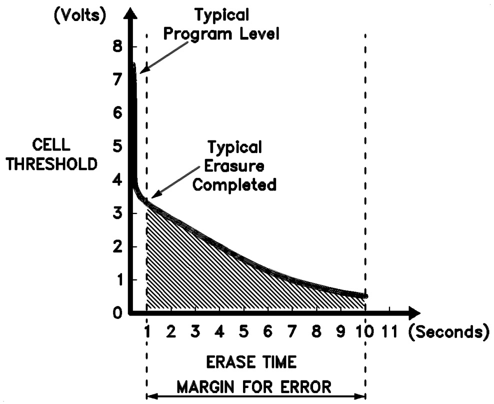


Figure 4. The logarithmic-decaying nature of erasure allows for 10x error in erase time before a device becomes inoperable. Remember, each device has its own erase time, thus the use of an adaptive algorithm.

Most Common Development Issues

Having covered the fundamentals of flash memory reprogramming, let's move on to the system's hardware and software perspectives. The following list of questions might have occurred to you . . .

- You have defined a system power supply with regulated 5V and 12V outputs (V_{CC} and V_{PP}). Due to the smaller capacitive load on the V_{PP} supply, V_{PP} powers up much faster than the V_{CC} supply. Will this affect the device?
- How does the flash command register architecture reset?
- Suppose your code sends a signal to start erasure, and never tells it to stop?
- Suppose your software delay timers are not calibrated. Instead of stopping erasure after ten milliseconds, the code issues the stop command after 10 seconds?
- Suppose your 6 μs timer used between the erase and erase verify modes is only 2 μs?
- Suppose you decide to skip the first erase operation (program all bytes to zero) because the device is already programmed with data?
- Suppose you are programming and erasing devices in a 16- or 32-bit system?

The answer to all these questions can be found in the following sections. The questions and the reasons all relate to the discussion of how the cell works.

Device Initialization and Reset

Many logic devices which contain command or control registers also have a reset pin. This pin serves two purposes: it resets the device's internal logic; and it synchronizes the device's clock to the system clock.

Intel's first generation flash memory command register and reprogramming circuitry **reset to the read mode** by three means:

1. raising or lowering V_{PP} with $V_{CC} = 5V$;
2. raising V_{CC} with $V_{PP} = 12V$;
3. issuing the reset command twice in succession.

NOTE:
Method 3 stops erasure or programming as well as resets the chip.

A few cases require closer consideration.

Case 1. The System Controls V_{PP} with a Switch

Assuming V_{CC} is stable with V_{PP} switches on, then the command register defaults to the read mode. No power-on reset is required.

Designers might opt to include the V_{PP} switch for either (or both) of two reasons. The first reason is power minimization. Depending on the technology used, a voltage regulator or pump's efficiency can range from 40%–85%. Switching off the V_{PP} supply minimizes system power consumption. See Appendix B for an example V_{PP} generation circuit with ON/OFF control capability.

The second reason is absolute data protection. This feature is not available to 5V-only EEPROM because the reprogramming voltages are generated internally. On that class of memory device, logic glitches can spuriously change data during system power up or power down. Flash memory's 12V power requirement offers absolute control over these concerns; with V_{PP} below $V_{CC} + 2V$, data protection is guaranteed. Internally, the electric fields are simply too weak to spuriously write data.

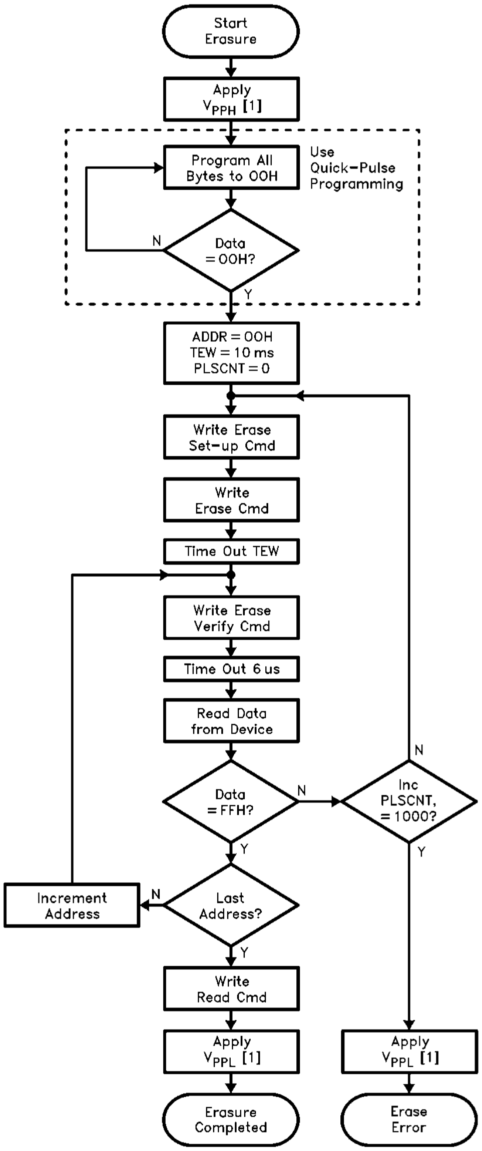
Case 2. V_{PP} Powers Up before V_{CC}

Systems with V_{PP} hardwired to a regulated transformer might encounter this case. Typically, V_{CC} will charge many more bypass capacitors than V_{PP}. V_{CC} will therefore power up much more slowly.

The flash memory power-down ($V_{CC} = 0V$) default state blocks V_{PP} from disturbing the array. These conditions hold while V_{CC} is below ~2V. Once V_{CC} rises above ~2V, the internal logic kicks in and resets the device to the read mode. (This is analogous to the internal V_{PP} reset condition described in Case 1.)

Should the three control pins glitch during the power-up phase ($\overline{CE}$low, $\overline{WE}$low, and $\overline{OE}$high), then the command register acts to filter the data. The command port will only react to the correct command sequence.

Designers might opt to hardwire V_{PP} for a number of reasons. The first reason is cost minimization. A regulated 12V secondary from a transformer is commonly available. Adding a switch or a power supply sequencer adds cost and complexity. The second reason involves consideration of the end application. Using the flash memory as a read/write memory requires optimization for the write cycle. Powering V_{PP} on before each write would waste considerable time.



292059–6

Bus Operation	Command	Comments
Standby		Wait for V_{PP} Ramp to V_{PPH} (= 12.0V)[1]
		Use Quick-Pulse Programming
		Initialize Addresses, Erase Pulse Width, and Pulse Count
Write	Set-Up Erase	Data = 20H
Write	Erase	Data = 20H
Standby		Duration of Erase Operation (T_{whwh2})
Write	Erase Verify	Addr = Byte to Verify; Data = A0H; Stops Erase Operation
Standby		t_{WHGL}
Read		Read Byte to Verify Erasure
Standby		Compare Output to FFH, Increment Pulse Count
Write	Read	Data = 00H, Resets the Register for Read Operations.
Standby		Wait for V_{PP} Ramp to V_{PP} 1[1]

NOTES:

1. The V_{PP} power supply can be hard-wired to the device or switchable. When V_{PP} is switched, V_{PP} 1 may be ground, no-connect with a resistor tied to ground, or less than V_{CC} + 2.0V.
2. Erase verify is performed after chip-erasure. A final read/compare may be performed (optional) after the register is written with the read command.
3. CAUTION: The algorithm MUST BE FOLLOWED to ensure proper and reliable operation of the device.

Figure 5. Quick-Erase Algorithm for Intel's First Generation Product Line

Case 3. Warm Resets

Warm resets, where the system maintains power while rebooting, requires closer inspection. Consider the situation where the system is reprogramming the flash memory and a hardware or software reset occurs.

The boot software would not realize that programming or erasure is ongoing and would not know to stop the reprogramming operation. Therefore safeguard against this condition with one of two means: 1) ensure that control logic switches V_{PP} off during reset; or 2) reset the flash memory before resetting the processor. For a software reset, simply add the flash memory reset command to the interrupt sequence. For hardware resets, wire the reset switch to the interrupt controller instead of directly to the reset input. Hardware resets would then execute the software interrupt sequence. Intel's second and third generation flash memories all include a H/W reset pin (RP#) formally called PWD#. This pin is essential for any device with automation or embedded algorithms.

The Erase Algorithm Interpreted

The following section offers a block by block explanation of the Quick-Erase algorithm shown in Figure 5. Understanding the reasons behind a function will enable you to appreciate the importance of following the algorithm explicitly. Deviations will negatively affect the part's performance and should not be attempted. Note: the effect may not be immediately apparent.

Apply V_{PPH} (Optional, see Discussion on Device Initialization)

Switch on the local V_{PP} supply prior to erasure and programming. The time required for V_{PP} to reach its steady state 12 ± 0.6V depends on the capacitive load and the impedance of the printed circuit board trace. If you measure this delay on a wire-wrapped prototype system, remember that temperature, printed circuit traces and the board's layout change the load seen by the V_{PP} generator. Allow V_{PP} sufficient time to ramp before proceeding with the next step.

Program All Bytes to 00h → Data = 00h?

Prior to erasure, blanket program all addresses in the flash memory to 00h (charge state), **regardless of the previous data**. Verify that each address equals 00h before proceeding to the next address. If you use only part of the memory array, you still need to pre-program the **entire** array for erasure. An example where this is an issue is using a 512K in a 256K socket. A second example is a system where internal microcontroller memory overlaps the external flash memory space.

Programming data 00h equalizes the charge on every bit in the array. This is necessary because erasure removes charge from all cells regardless of their previous state.

For example, reconsider the byte containing AAh data (1010 1010b). If you skip the pre-program step, then during erasure when the data "0" bits get charge removed, the previously erased bits (data "1") lose additional charge. This drives the cell threshold a little lower. The next time you erase the chip and change the code, the threshold will drop to 2.8V.

If the memory transistor is not pre-programmed to data "0" before the next erasure, then its threshold will drop on successive reprogramming cycles (denoted by E3, E4, etc. in Figure 6). Repeated violations of the blanket programming requirement drives the threshold to the point where the transistor is stuck on (data = "1").

Variable Initialization

Initialize two variables and a constant: ADDR (address), PLSCNT (pulse count), and TEW (erase pulse width). The pulse count increments from 0 to a maximum of 1000 erase tries. The erase pulse width remains constant at 10 ms. The address increments from the flash memory starting address to the ending address during verification.

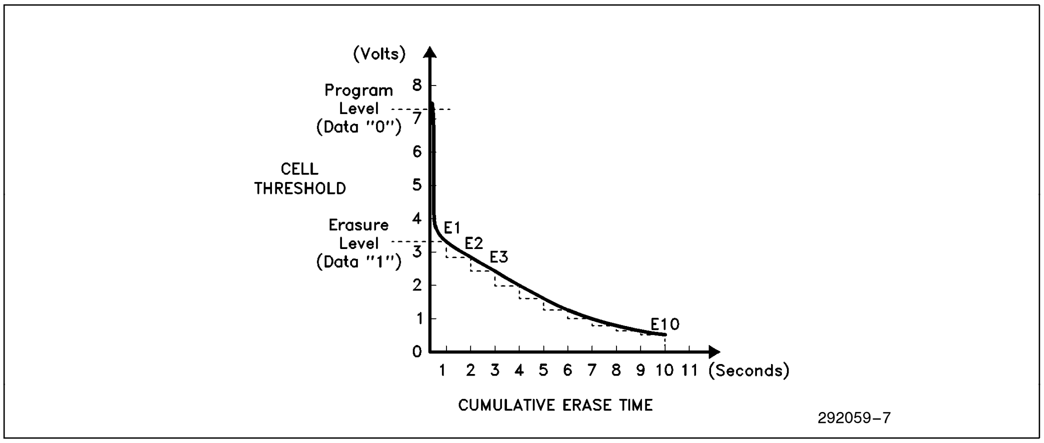


Figure 6. Successful erasure requires blanket programming all bytes to the data "0" level first. This prevents threshold decline on successive erase cycles (E2, E3, etc.). Very low thresholds cause the chip to malfunction.

Write Erase Set-Up Command

Write the erase set-up command (20h) to any flash memory address. This prepares the selected device for erasure, but does not activate the process. A second erase command (20h) is required. Any other data written to the flash memory between the set-up and erase commands will abort the sequence. Once the process is started, it will not stop until told to do so. The correct stop erasure command is Erase Verify (A0h). However, any command including the Reset command is an illegal sequence and will stop erasure as well.

Write Erase Command

The erase command starts the erase process. Internally, the device switches the voltages on all memory cell drains, gates, and sources to the erase configuration.

Time Out Tew (10 ms)

Start your software or hardware timer. Until commanded to verify or until the integrated stop-timer counts down, the flash memory continues the erase process. Therefore, **assign a high priority to the timer interrupt**. If a higher priority interrupt occurs, stop the erase process and switch contexts (store all variables, registers, etc.). This will allow reentry into the erase procedure in a controlled fashion.

Write Erase Verify Command

Write the erase verify command (A0h) to the flash memory at the address given by the ADDR variable.

The erase verify command performs many tasks. Internally, the device stops erasure and latches the given address for verification. Additionally, the command changes the voltages on the memory cell drains, gates and sources to the erase verify configuration.

Time Out 6 μs

This time out accounts for the internal slew rate of switching the memory array from the erase to the erase verify configuration. Do not attempt to read from the device before 6 μs has passed; the device will appear to still be programmed. This is because you have not allowed sufficient time for the memory to change configurations. Your code will then interpret this as a need for extra erase operations, and will continue erasing the device.

NOTE:
6 μs is a minimum specification. You can use the 10 μs timer developed for the programming algorithm.

Read Data from Device

Read the data at the address given by the ADDR variable. This should be the same address driven with the erase verify command.

Data = FFh?

Compare the output data at address ADDR to FFh. If the data equals FFh, then that address has been erased. Continue verification until the last address has been verified or until the maximum erase pulse count (1000) has been reached. Typically, most devices will fully erase within 50–100 erase loops.

Last Address → Increment Address

Check the ADDR variable to see if the last address has been verified. If not, increment the ADDR variable and re-write the erase verify command. Remember to write the erase verify command to address ADDR, since the verify command latches the address. Also, if your system has 64K byte segment boundaries, be sure to increment the base pointer every 64K byte addresses.

Write Read Command

After full chip verification, write the read command (00h) to switch the device to read mode. If you plan to reprogram the device immediately, this step is not necessary.

Apply V_{PPL} (Optional)

Switch V_{PP} off. With V_{PP} left on, the command register offers data protection by requiring a precise sequence to initiate programming or erasure. However, V_{PP} controls overall command register operation. Turning V_{PP} off disables the command register, thus providing absolute data protection. Without the high voltage, the reprogramming mechanisms cannot occur and the component becomes a read only memory (ROM).

Abort/Reset

Whenever a system error condition occurs (reset or reboot), write the Reset command (FFh) to each flash memory twice in succession. This is a good initialization practice in systems leaving V_{PP} at 12V. The processor would be unaware if prior to the reset, it had been in the middle of erasure, and this sequence aborts erasure.

The Program Algorithm Illuminated

The full algorithm will not be interpreted here, although a few items should be noted. You can find a conceptual version of the Quick-Pulse Programming algorithm in Chapter 2, Figure 1, and the complete flow chart in Appendix C.

First, similar to erasure, a two-write sequence starts programming. The first write is the Program Set-Up Command which primes the chip for programming. The chip then latches the address and data to be programmed on the second write. You can abort programming by writing the Reset Command twice in succession instead of the data to be programmed.

Second, the device continues programming until commanded to stop by the Program Verify Command. Similar to the Erase Verify Command, this command performs a couple of functions. Internally, it halts programming and latches the given address for verification. Additionally, the command changes the voltages on the memory array's drains, gates, and sources to the program verify configuration.

The cell programming mechanism is self-limiting. However, do not assume that programming all twenty-five 10 μs operations in one pass is the best way to attain reliable operation. To a certain degree, programming stresses the memory cell. The stress is considerably lower than that applied to EEPROM (2 MV/cm lower to be specific). But why stress a component without cause? The adaptive Quick-Pulse Programming algorithm with its fast program operations, minimizes all stresses and affords the greatest reliability.

Finally, after writing the Program Verify Command to the device, wait a minimum of 6 μs before reading the device. The time out accounts for the internal slew rate of switching the memory array from the program to program verify configuration. Do not attempt to read from the device before 6 μs has passed; the device may appear unprogrammed. Your code will then interpret this as a need for extra program operations. It may needlessly reach the 25 operation limit, even though the byte most probably programmed on the first pass.

Ramifications of the Golden Rule

Always follow the erase command with an erase verify command to stop erasure. Interrupt-driven systems must give high priority to servicing the reprogramming timer interrupts. Systems that reset upon a watchdog time-out must reset the flash memory device before rebooting. (See discussion on device initialization.) Likewise any non-maskable interrupt should software- or hardware-reset the flash memory before performing a context switch.

Use an oscilloscope to calibrate all time delays before attempting erasure. The delay modules include the 6 μs, 10 μs, and 10 ms timer routines.

Blanket program all addresses to 00h data before erasing. Verify correct implementation of the programming algorithm with a PROM programmer before attempting erasure. (Chapter 4 explains how this can be done.)

16- and 32-bit systems require special attention. Each flash device has its own erase characteristic. Do not assume that if the low byte of a data word is not erased, then the high byte must not be either.

Always follow the listed guidelines and take care while developing your code to eliminate the erase control issue. Consider it similar to implementing any control function. Once the code is debugged and stable, the issue goes away.

You might ask, is it not possible to control erasure through hardware? The alternative to software control is integrated hardware control or an external controller. Intel ofers a complete line of high density, cost-effective products with integrated hardware control, often called automation. Whether software or hardware controlled, Intel's ETOX Flash Memory offers the most reliable, dense, manufacturable, and fastest read/write nonvolatile memory. Other EEPROM approaches have drawbacks of multiple transistors per memory cell. This property negatively affects all those attributes offered by Intel's ETOX flash memory.

DEBUGGING YOUR CODE AND OTHER TIPS ON TESTING

As with any software checkout, a few simple principles enable complete flash memory algorithm debug. The following sections offer some hints to make your job easier.

Software Drivers Save You Time

Intel saves you time by offering various processor-family flash memory drivers. You simply edit the files to suit your system. Then assemble the driver, link, and locate it, and you are ready for debug.

These drivers offer the framework for successful flash memory reprogramming, and require some customization to fit your particular system. If your processor's driver is not available, you may use the available driver software as an example. One caution in advance: The drivers have been written in assembly language to give the most speed- and memory- efficient code. However, most people prefer high-level programming.

High-level programming can be used for everything except software-timer generation. Compilers may give different routines with different object code on each compilation. Therefore, the timers must be either hardware-based or coded in assembly language. Software timers also present some risk if there is a frequency upgrade change on the controlling processor. Regenerate and check your timer routines whenever the system clock rate changes.

Timers, Test Loops and Assembly Level Programming

Timing circuits or software play the most crucial role in flash memory reprogramming. Good timers precisely control their function; sloppy timers produce faults. An example of a sloppy timer is one produced by a compiler. Each time high-level languages recompile, the low-level object coding may change. Thus, a timing loop may be 10 ms one compilation, and much longer or shorter the next time.

You can check your timing method with the following simple technique. Develop test loops which call the various timers' routines. For example, implement the 6 μs, 10 μs, and 10 ms timers used with the 28F512 and 28F010. If you have a spare port or peripheral output, use it to trigger an oscilloscope. Follow the trigger call with the timer routines. If you do not have a spare port, write to the flash memory address space before and after each timer call. You can trigger the oscilloscope off of the flash memory CE# signal. Remember to power-down the system and remove the flash memory before attempting these methods.

Once the timer code has been verified, you can link and locate it to a higher-level erase/program algorithm implementation.

Programming—The Key to Proper Erasure

Earlier sections described the importance of programming 00's prior to erasure. This procedure equalizes the charge on all memory cells; following this step all bits erase in unison.

A conclusive debug technique can check your programming software. Simply use your software to program zeroes into the flash memory and then verify this step using a conventional PROM programmer. Load the PROM programmer's buffer with all zeroes and compare the buffer to the flash memory. If your programmer does not service the flash memory, call the company for the latest software upgrade. Alternatively, one can easily rig the 512K flash memory to look like a 512K EPROM. Simply jumper V_{CC} on a 32-pin socket to a few pins. Note that the 27512 EPROM and 28F512 flash memory have different intelligent identifier codes. Override the identifier code check to use this method. See Figure 7 for socket details.

Some microcontrollers have limited address space or internal memory that masks certain external address space. Even if you do not use sections of the flash memory, you must still access these sections to program zeroes before erasure. Map and decode port bits to access unused address space, and verify that all bytes are programmed to zero before proceeding with erasure.

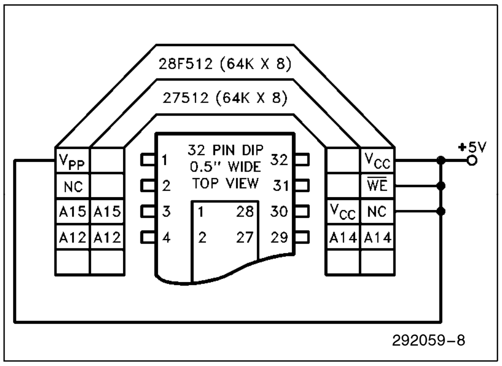


Figure 7. A 28F512 can be read in a PROM programmer as a 27512 by jumpering the appropriate pins to V_{CC}. The same method applies to the 28F010.

16- and 32-Bit Systems—Achieving Optimum Reprogramming Throughput

Erasing flash memory in 16- and 32-bit systems requires special consideration. One could implement the program and erase algorithms in a byte-wise fashion, but this is time-consuming. Alternately, one can treat the multiple flash memory as a data word, and gain optimum performance.

The primary consideration with the latter approach is that one device may program or erase faster than the other(s). Subsequent programming or erasure of a slower device compromises the functionality of the faster device by subjecting it to the slow-device timing.

Consider an example of erasure in a 16-bit system. After 10 passes through the erase operations, both devices verify through address 07C3h. Then at address 07C5h, the processor reads data word 83FFh.

Since erase data is FFFFh, a few bits in the upper byte/device have not erased. The natural flow of the algorithm would dictate another erase operation. But what about the lower device? Could it be completely erased? Of course it could be; every device erases at a different rate and the algorithm has only checked up to address 07C5h.

You **can** take advantage of the data rate of wider data buses by utilizing the command register, the reset command and an analysis of erasure. Each erase operation is 10 ms. Each byte verification takes 6 μs. Therefore, erasure takes three orders of magnitude longer than verification. Optimization for erasure yields the optimum performance because verification is a second order effect.

Let us reconsider the previous example. At address 07C5h, the data word does not verify. On the next erase operation edit the erase (and erase verify) command word such that only the high byte gets the erase command, and the low byte gets a reset command. (i.e., change the command word from 2020h to 20FFh).

See Application Note AP-316 for a detailed flow chart for this approach. Note this document is based on the 28F256; however, most concepts carry through to the higher density devices. (Literature Order number 292046).

Logic Analyzers and In-Circuit Emulators

Many programmers use logic analyzers and in-circuit emulators to debug code. These approaches are fine for flash memory algorithm debug if certain conditions are met.

1. Check timing routines with an oscilloscope; there is no alternative.
2. Know your code and set breakpoints intelligently. One designer had the bad luck of throwing in a breakpoint in the middle of the program 00's loop. After stepping through a couple of byte program 00 loops, he called the erase flow routine. Can you identify the problem?
3. Single step through your reprogramming code, if and only if, the flash memory device is removed from the system.

Testing Your Software—One More Time

Some flash memories specify 100 erase/program cycles. This is a minimum specification; Intel Flash Memories cycle 100,000 times. With this in mind, feel free to check and recheck reprogramming operations. There is **no** reliability risk in doing so.

One confidence-raising test is similar to that done on systems: stress the system/software by executing test code numerous times consecutively. Set the reprogramming drivers in a loop, and let them run 20–40 times. On each consecutive pass, use a constant data pattern such as 0AAh. This tests the reprogramming code from a quasi-static perspective. Missing is the true system environment. In the true system environment, multiple inputs compete with the flash memory for interrupt priority. Also, RF noise from motors can cause spikes and glitching on V_{PP} or V_{CC}. Additionally, fully loaded systems or partially loaded systems might have different V_{PP} response characteristics or noise levels. Signals that look clean in the lab, might not be all that clean in the true operating environment. Therefore, flash reprogramming tests should be done in the true system environment as a final test.

Watchdog Timer Debug Circuit

This section describes a simple tool you can build for debugging your code. Since Intel's first generation Flash Memories contain integrated stop-timers, this tool offers no real benefit. It is simply shown in case a designer is debugging code for alternate sources. An EPLD watches the flash memory data bus and control signals for the erase sequence—erase set-up, erase, and erase verify commands. Once the CPU initiates erasure, the debug tool starts a 15 ms timer. Should the timer count down prior to receiving the erase verify command, then the circuit switches V_{PP} off.

This tool does not check for the other items discussed in the **Tips on Testing** section; you must still check those yourself.

Figure 8 shows the circuit schematic. The EPLD source code and the name of an Intel EPLD applications engineer is located in Appendix D.

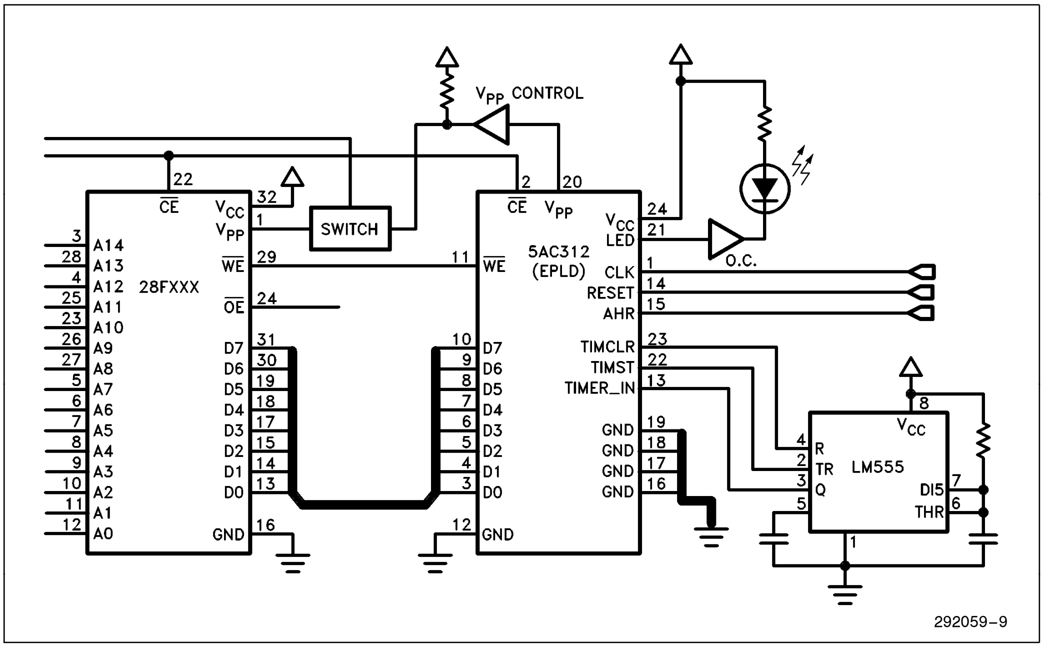


Figure 8. Watchdog Timer Debug Circuit

TROUBLE SHOOTING GUIDE

Determining the Root Cause— Software Error vs Device Damage

The three major indications of a flash memory problem are labeled in the following section. The subsequent paragraphs define potential root causes to investigate.

I. **The Device Does Not Program**

Did it program before?

A. **No.**

1. Trigger your oscilloscope on CE# while probing V_{PP}. Verify that V_{PP} has reached a steady-state 12V when the device is first written.
2. Set the time-base to 10 μs/division (the duration of the program operation). Trigger on CE# and probe WE# (look at both traces). Check the duration of the 10 μs program operation time delay. Also, check the duration of the 6 μs delay between writing the program verify command and read.
3. Look for ringing on V_{PP} when V_{PP} has been switched on. Over-voltage stress on V_{PP} (ringing with amplitude greater than 13V) will destroy V_{PP}'s silicon structure.
4. Power the system down and back up. Look for destructive glitches on V_{CC} or V_{PP} (greater than 7V and 13V respectively).
5. Verify erasure and programming on a PROM programmer (if available). Fill the programmer buffer first with 00h data and program the buffer to the flash memory. Then erase the device, and repeat with AAh data. Repeat the last step with 55h data. This sequence fully exercises the array, the input buffers and the output buffers. If all tests pass then check for a hardware or software error.

B. **Yes.**

1. Have you done anything that may have ESD zapped the devices (i.e., touched the devices while not being grounded, re-wired the protoboard with the components socketed, etc.)? If yes, check part as outlined in section 1.A.5.
2. Have you attempted erasure? If yes, verify your algorithm as outlined in Chapter 4. Also, implement the in-system intelligent identifier mode. If the device outputs an incorrect code, then either an output has been zapped or the golden rule has been violated. Section 1.A.5 describes a method of checking for ESD damage.

II. **The Device Does not Erase**

Did it erase before?

A. **No.**

Follow steps 1–5 outlined in section I. When performing step 2, adjust the oscilloscope time base to 10 ms/div.

B. **Yes.**

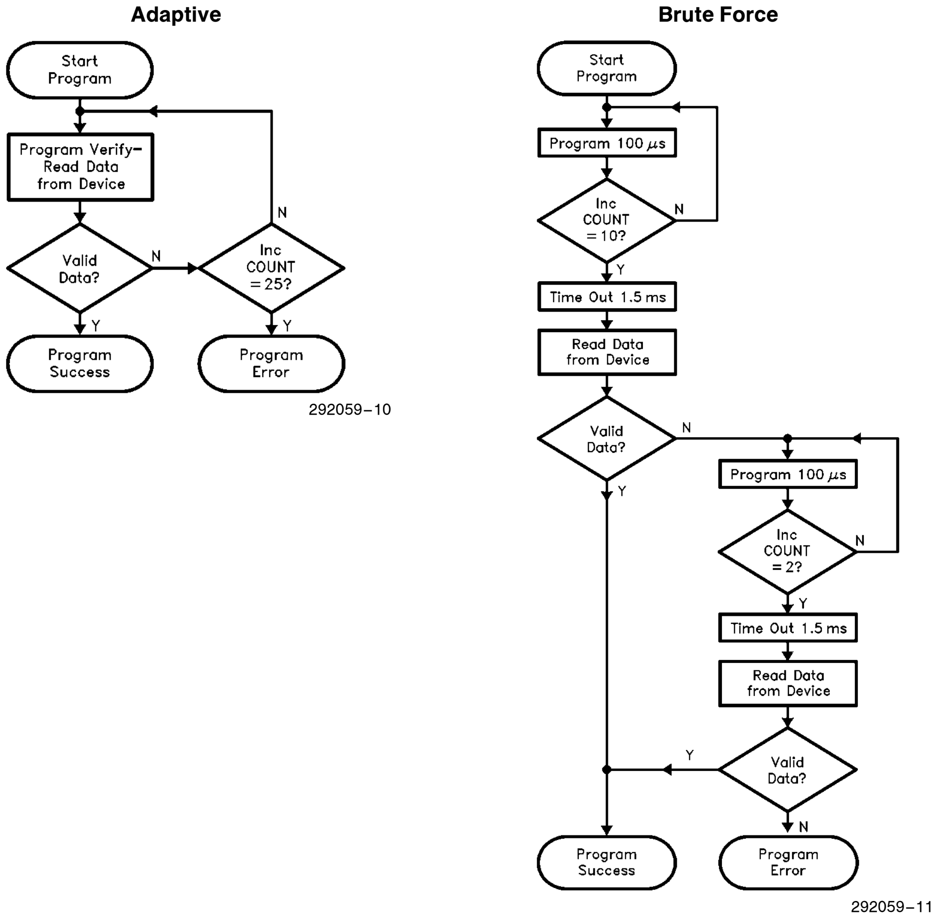
Has the board design, clock rate or software changed? System clock rates directly affect the accuracy of software timers. See AP-316 for a discussion on software timing versus clock rate.

III. **The Device Erases Spuriously**

Exercise all system functions while monitoring the flash memory chip selects. Verify that I/O mapped addresses or logic are not accidentally selecting the flash memory. For example, the space bar character sent from a keyboard controller happens to be 20h. If the flash memory is accidentally selected while this data is on the bus, then erasure will commence on the following cycle when the condition occurs again.

APPENDIX A

TWO APPROACHES TO ALGORITHMS



□ Efficient—

Max Time = (10 + 6 μs) ∗ 25
= 400 μs

Typ Time = (10 + 6 μs) ∗ 1
= 16 μs

□ Reliable—

Verify Command slews internal voltages to simulate elevated V_{CC}.

□ Slow—

Max Time = (100 μs ∗ 12) + 3 ms
= 4200 μs

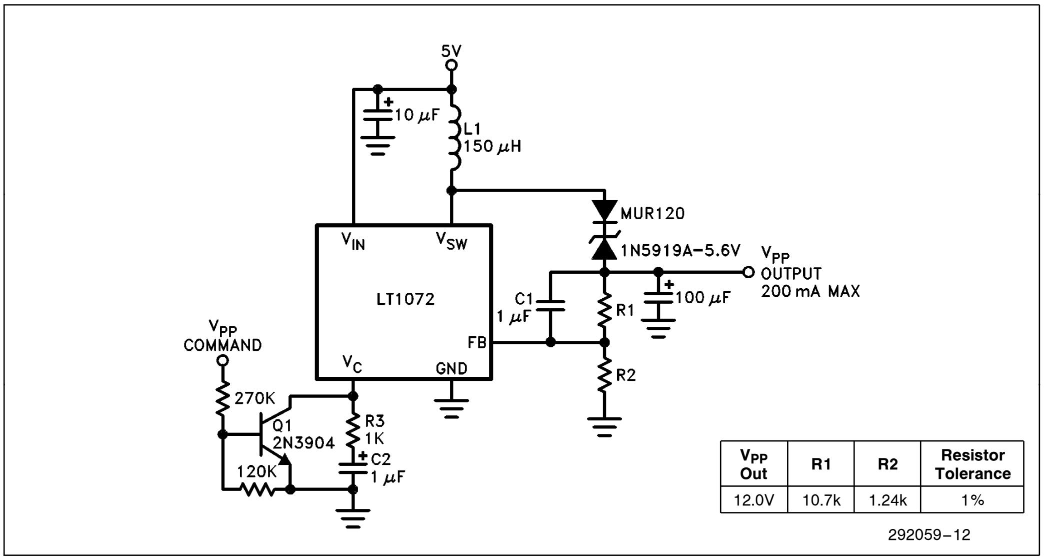
Typ Time = (100 μs ∗ 10) + 1.5 ms
= 2.5 ms

□ Questionable—

Unknown margin to V_{CC} and temperature swings, as well as cell leakage.

Figure 9. Left and right flow charts compare Intel's (adaptive) Quick-Pulse Programming algorithm and another company's (brute force) approach to flash memory programming.

APPENDIX B

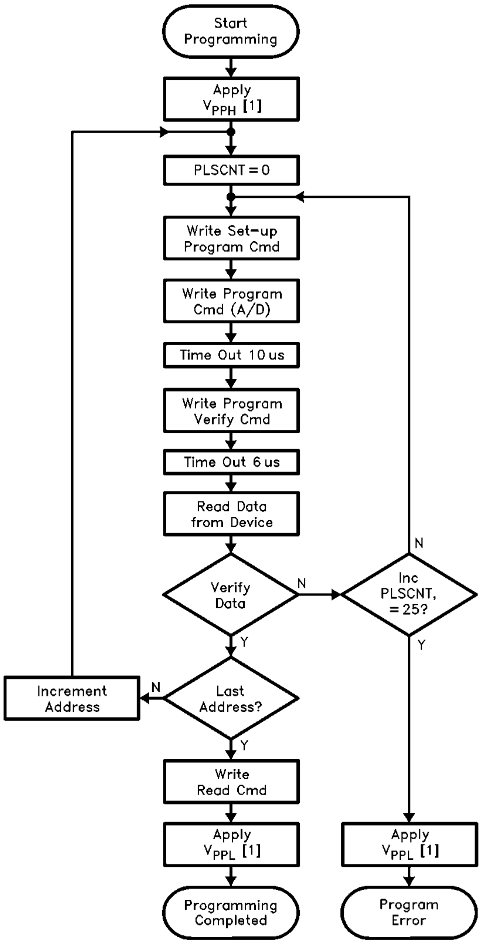


V_{PP} Out	R1	R2	Resistor Tolerance
12.0V	10.7k	1.24k	1%

292059–12

Figure 10. Basic flash memory V_{PP} voltage supply with ON/OFF control. When V_{PP} COMMAND goes low, the Linear Technology LT1072 switching regulator produces 12V. This circuit is just one example of a V_{PP} supply.

APPENDIX C
QUICK-PULSE PROGRAMMING ALGORITHM



292059-13

Bus Operation	Command	Comments
Standby		Wait for V_{PP} Ramp to V_{PPh} (= 12.0V)(1)
		Initialize Pulse-Count
Write	Set-Up Program	Data = 40H
Write	Program	Valid Address/Data
Standby		Duration of Program Operation (t_{WHWH1})
Write	Program(2) Verify	Data = COH; Stops Program Operation
Standby		t_{WHGL}
Read		Read Byte to Verify Programming
Standby		Compare Data Output to Data Expected
Write	Read	Data = 00H, Resets the Register for Read Operations.
Standby		Wait for V_{PP} Ramp to V_{PPl} (1)

NOTES:

1. The V_{PP} power supply can be hard-wired to the device or switchable. When V_{PP} is switched, V_{PP} I may be ground, no-connect with a resistor tied to ground, or less than V_{CC} + 2.0V.
2. Program Verify is only performed after byte programming. A final read/compare may be performed (optional) after the register is written with the Read command.
3. CAUTION: The algorithm MUST BE FOLLOWED to ensure proper and reliable operation of the device.

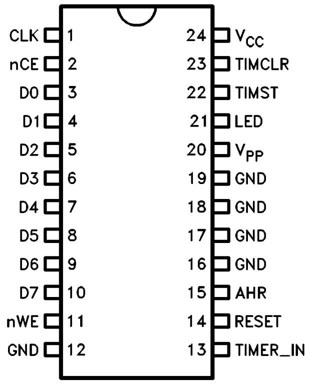
APPENDIX D
WATCHDOG TIMER CIRCUIT

EPLD Source Code and Applications Contact Person

```
Thom Bowns_PLFG Applications
Intel
January 5, 1989
Rev. 008
5AC312
Watchdog timer to cut VPP from FLASH if erase cycle too long.
OPTIONS: TURBO = ON
PART: 5AC312

INPUTS:       CLK, D0@3, D1@4, D2@5, D3@6, D4@7, D5@8, D6@9, D7@10,
              nCE@2, nWE@11, TIMER_IN@13, RESET@14, AHR@15

OUTPUTS:      TIMCLR@23, TIMST@22, LED@21, VPP@20

NETWORK:
              CLK = INP (CLK)
              D0 = INP (D0)
              D1 = INP (D1)
              D2 = INP (D2)
              D3 = INP (D3)
              D4 = INP (D4)
              D5 = INP (D5)
              D6 = INP (D6)
              D7 = INP (D7)
              nCE = INP (nCE)
              nWE = INP (nWE)
              TIMER_IN = INP (TIMER_IN)
              RESET = INP (RESET)
              AHR = INP (AHR)          % RESET active high if AHR is high %
              COND1 = NOCF (C1d)       % Conditions 1-4 are routed       %
              COND2 = NOCF (C2d)       % through combinatorial feedbacks  %
              COND3 = NOCF (C3d)       % to reduce product term count.    %
              COND4 = NOCF (C4d)       %                                  %
              CLR = NOCF (CLRd)

EQUATIONS:
              CLRd = RESET * AHR + !RESET * !AHR;
              TIMEOUT = /TIMER_IN;
              C1d = (/nCE * /nWE * 20H);      % Write 20 %
              C2d = (/nCE * /nWE * a0H);      % Write A0 %
              C3d = (/nCE * /nWE * /20H);     % Write other than 20 %
              C4d = (/nCE * /nWE * /A0H);     % Write other than A0 %
              20H = /D7 * /D6 * /D5 * /D4 * /D3 * /D2 * /D1 * /D0;
              A0H = /D7 * /D6 * /D5 * /D4 * /D3 * /D2 * /D1 * /D0;
```

```
MATCHING:  WATCHDOG
CLOCK:     CLK

STATES:    [ VPP  LED  TIMST  TIMCLR  XSB ]
 START     [  0    0     0       0     0  ]
    S1     [  1    0     0       0     0  ]
    S2     [  1    0     1       0     0  ]
    S3     [  1    0     1       1     0  ]
    S4     [  1    0     0       1     0  ]
    S5     [  1    0     1       1     1  ]
    S6     [  1    0     1       0     1  ]
    S7     [  0    1     1       0     0  ]

           % TRANSITION STATEMENTS %

START: S1                            % From power up, go to S1 right away %
S1:    IF COND1    THEN S2           % If write 20, go to next state      %
S2:    IF /COND1   THEN S3           % Until not write 20, hold           %
       IF CLR      THEN S1
S3:    IF COND1    THEN S4           % If another write 20, start timer   %
       IF COND3    THEN S7           % If write other than 20, error      %
       IF CLR      THEN S1
S4:    S5                            % Trigger timer then go to S5 loop   %
S5:    IF COND2    THEN S6           % If write A0, stop timer            %
       IF TIMEOUT  THEN S7           % If timer times out,
                                       go to error state                  %
       IF COND4    THEN S7           % If write other than A0, error      %
S6:    S1                            % Stop timer and go back to S1       %
S7:    IF CLR      THEN S1           % Error state. wait for a RESET.     %

END$
```

EPLD Pinout
5AC312



292059-14

APPENDIX E

Checklist: Most Common Mistakes that May Lead to Excessive Erasure

- not programming all bytes to 00 data prior to erasure;
- not observing the 6 μs set-up times between programming or erasure and verification;
- attempting to program before V_{PP} is at 12V (Capacitive Load)
- not latching the erase verify address with the erase verify command, or changing the address on the subsequent read cycle;

Chapters three and four discuss the correct methods of developing and debugging code to diminish the possibility of making these mistakes.