Booting from NAND Flash Memory

Introduction

NAND flash memory technology differs from NOR flash memory which has dominated the embedded flash memory market in the past. Traditional applications for NOR flash memory are code storage and executing in place (XIP) in many embedded systems. Historically, NAND flash memory has been used extensively in data storage applications such as Solid State Drives (SSD) and flash memory cards. Today, with improvements in endurance, cost, and performance, NAND flash memory is being deployed more frequently in code storage applications in complex embedded systems.

Parallel NOR flash memory was a popular choice for complex embedded systems because its interface contains both address and data buses that enable execute-in-place operations. Execute-in-place capability allows an embedded CPU to fetch code directly from a code storage device and execute the code immediately. With the emergence of the NOR serial interface (SPI), system designers were able to reduce system complexity and cost by reducing the numbers of I/Os, while maintaining system performance.

Most recently, NAND flash memory is employed to further reduce the cost of a complicated embedded systems while maintaining the required system performance. However, booting from NAND can be a challenge in a complex embedded system. NAND flash is a page-oriented memory device that does not inherently support XIP, at least not in the same manner as a typical XIP memory device such as NOR flash. The booting procedure must be modified when migrating from NOR flash in order to fulfill the architectural constraints of NAND flash memory.

Scope

This document describes a typical sequence of steps and discusses special requirements necessary to boot from NAND flash. With a wide variety of CPUs implemented in embedded systems, the operations described below may vary. This application note does not specify any protocols of internal/external buses and timing for any specific embedded microprocessor but it does describe the Macronix NAND flash memory interface.

"Table 1. Macronix NAND Flash" lists Macronix NAND flash memory discussed in this document.

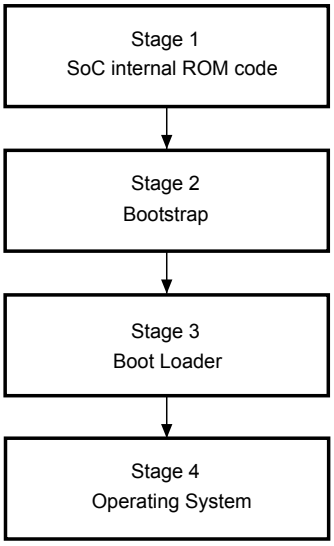
Table 1. Macronix NAND Flash

Part Number	Density	Device ID	Bus Width	Page Size(Bytes)
MX30LF1208AA	512Mb	F0h	x8	2112 = (2048 + 64 bytes per page)
MX30LF1G08AA	1Gb	F1h	x8	2112 = (2048 + 64 bytes per page)

Boot-from-NAND Overview

"Figure1. Boot-from-NAND Stages." shows a typical boot sequence in a complex embedded system.

Figure1. Boot-from-NAND Stages.



Boot Stages

Stage1: SoC Internal ROM Code

After power-on reset or system hardware reset, the embedded CPU will fetch its first code from internal ROM. The main function of ROM code is to transfer the bootstrap code from NAND flash memory into SoC internal SRAM. The ROM code can detect different hardware settings from I/O or latch information during the reset period in order to branch to program code that loads the bootstrap code either from NAND or NOR. In this application note, we focus on NAND flash memory with code storage for booting purposes.

Stage2: Bootstrap

In Stage2, we have bootstrap code sitting in internal SRAM and ready to be executed. The main functions of bootstrap code are to copy the boot loader code from NAND into DRAM and to configure SoC hardware settings for customization purposes.

Stage3: Boot Loader

Stage3 has the boot loader residing in DRAM ready for the embedded CPU to execute it for shadowing the operating system. The main function of the boot loader is to copy the operating system from the NAND flash into DRAM for booting.

Stage4: Operating System

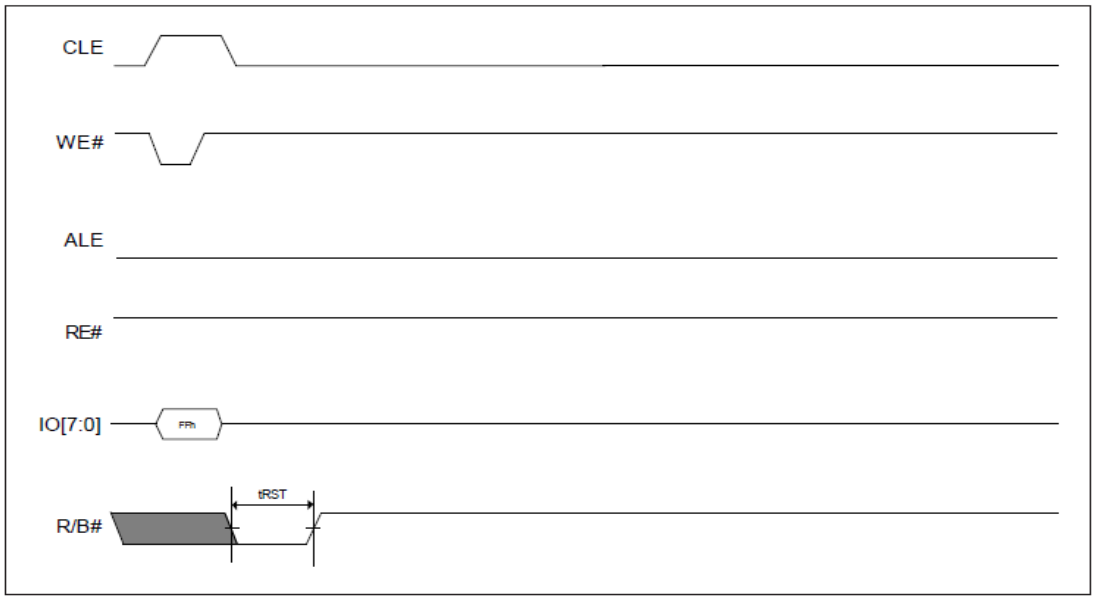
The operating system (ex: OS kernel) is in DRAM and is launched to complete the booting process. The operating system takes control of the whole system at the completion of Stage 4.

Stage1: SoC Internal ROM Code

After power-on reset or system hardware reset, the embedded CPU will begin executing code from ROM in Stage1. The ROM code cannot be modified by the system designer. For security purposes, the, ROM code can be written to support Macronix NAND flash by reading the manufacturer and device IDs of the installed flash to determine which Macronix NAND flash is being used before proceeding with the boot procedure. After a power-on-reset or system-reset is initiated, all registers in the SoC are in their default values.

The very first ROM code instructions issue the RESET command (FFh) to reset the NAND flash memory and waits a tRST time for the NAND flash memory to be ready (*"Figure 2: NAND Flash Memory RESET Command FFh"*). For the value of tRST, please refer to the appropriate Macronix datasheet.

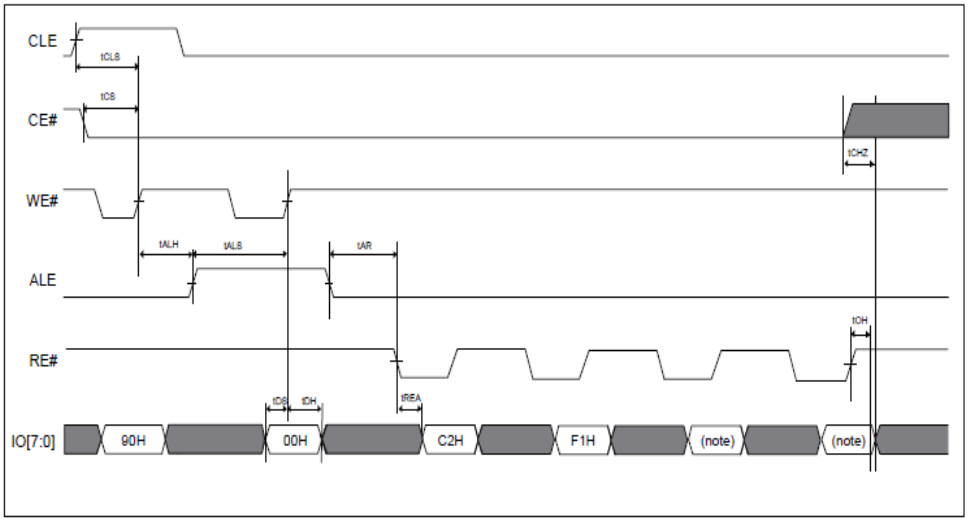
Figure 2: NAND Flash Memory RESET Command FFh



Then ROM code then issues the Read ID command (90h) to read the NAND flash memory manufacture ID and device ID (*"Figure 3: Read ID Operation"*) to ensure that the correct NAND flash memory is in place and is supported by the ROM code.

Now the ROM code is ready to load the bootstrap code from the NAND data blocks to internal SRAM. Before we move forward to load the bootstrap code, we have to understand NAND flash memory characteristics such as Bad Block management and ECC schemes for NAND flash physical memory management which are not required by NOR flash memory.

Figure 3: Read ID Operation



Note: The 1st code returned (C2h) is the Macronix manufacturer ID. The 2nd code will be F0h for a 512Mb device or F1h (as shown) for a 1Gb device. Please refer to the appropriate Macronix datasheet for a detailed definition of the 3rd code and 4th code of the ID table.

ECC and Bad Blocks Management

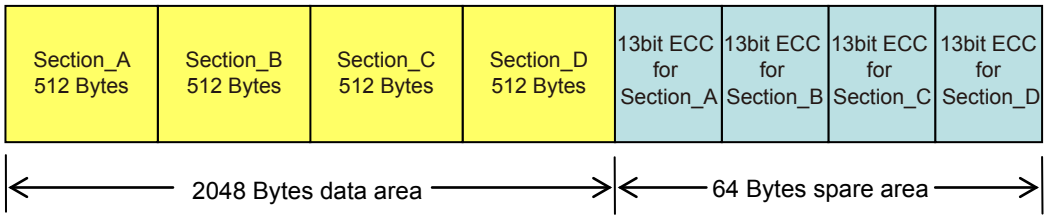
The ROM code expects bootstrap code to reside in block 0, 1, 2 or 3 of the NAND flash memory. The bootstrap image may be duplicated in those 4 blocks for reliability reasons. Macronix NAND flash memory block 0 is guaranteed to be good up to 1K cycles with 1bit ECC per 528 bytes. Therefore, in most cases, the ROM code will not have to look beyond block 0 for a bootable image.

When programming boot code and the OS into a NAND flash memory, the system programmer must first scan for bad blocks and build up a bad block table, which usually resides in block 0 and other duplicate locations of the NAND flash memory. The location of the bad block table is defined by the user. Therefore, when ROM code transfers bootstrap code into internal SoC SRAM, it also loads the bad block table into SRAM to ensure that only good blocks are used when accessing blocks beyond block 0 at a later time.

For those users who do not use all 64 bytes of spare area of each NAND page, the first byte of a spare area can be reserved for bad block marking. If the first byte of a spare area in the 1st or 2nd page is not FFh, the system designer may consider the “marked” blocks to be a “bad blocks”. Skipping the bad block and moving on to the next block and checking that it is good, is a scheme that should be implemented for NAND flash devices, if the system designer does not want to build and use a bad block table. Remember not to erase the bad blocks, otherwise the system may lose the bad block marking, and be unable to identify an inadvertently use the bad blocks in the future.

Macronix NAND flash memories, MX30LF1208AA and MX30LF1G08AA, require 1 bit error correction capability per 528 bytes of data (including ECC parity bits). For a 2048 + 64 bytes per page data structure, we can divide a page into 4 data sections (section A, B, C, D) and each section can accommodate a 528 (512 + 16) bytes data unit (*"Figure 4: Page Level Code Storage Structure"*). A 1 bit error correction capability within 528 bytes requires 13 parity bits using the Hamming code ECC algorithm. While programming the code image into NAND flash, 13 ECC parity bits for each 512 byte section must be calculated and stored within 16 bytes of that section’s spare area. Once all 4 sections are done for each page (2048 + 64 bytes per page), the programmer continues programming at the next page with the same scheme for NAND flash memory until the entire code has been programmed into data blocks successfully. The ECC algorithm can be implemented by either hardware or within ROM code depending on the architecture of the SoC.

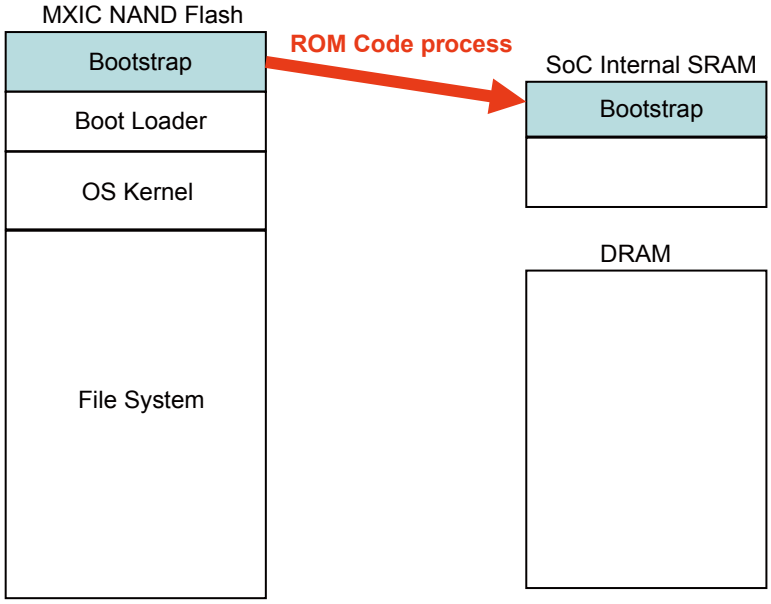
Figure 4: Page Level Code Storage Structure



Bootstrap Code Shadowing within SoC SRAM

After NAND flash memory has been verified and a bad block table has been established, the process of copying (shadowing) from NAND flash memory to internal SRAM starts (*"Figure 5: Shadowing Bootstrap Code from NAND Flash Memory to Internal SRAM"*).

Figure 5: Shadowing Bootstrap Code from NAND Flash Memory to Internal SRAM



After the copying process is done, the embedded CPU will jump to the internal SRAM address where the first byte of bootstrap cade was stored by ROM code.

ECC Error Handing During Bootstrap Copy Process

In the event an un-correctable ECC error (e.g. 2 or more bits of errors) is found during the copy process, the duplicate bootstrap code image stored in a different data block can be used by the embedded CPU to attempt a recover from this error. For example, if there is a block 0 reading error (a very rare event), ROM code can search the next data block for a valid bootstrap image and attempt to load it into SRAM. Error handling requires technical skills that are beyond the scope of this application note and should be handled very carefully to prevent any glitches during the boot process.

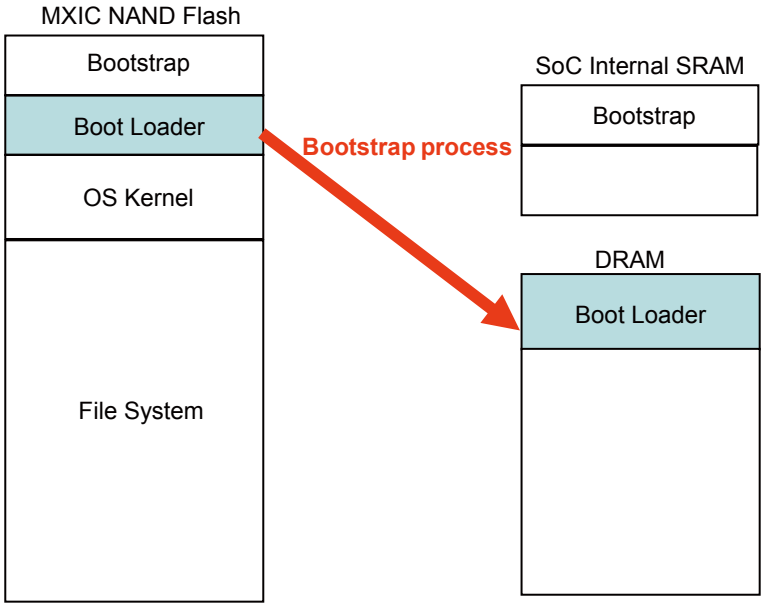
Stage 2: Bootstrap

At the onset of the 2nd stage of the boot process, bootstrap code has already been placed in the appropriate SRAM location for the embedded CPU to execute for the next step. In most cases, the bootstrap should include all information needed to support the current embedded CPU, including PLL clock frequency, essential SoC configuration registers, NAND flash driver, and memory management in embedded system, etc.

In reference to *"Figure 4: Page Level Code Storage Structure"*, note that the NAND flash memory data programming structure is different from NOR flash memory due to the ECC function and parity bits. For reliability reasons, the bootstrap image can be duplicated in other good blocks such as the first four good blocks of the NAND flash device.

After SoC configuration is done, shadowing Boot Loader from NAND flash to DRAM is the next major bootstrap task (*"Figure 6: Shadowing Boot Loader from NAND Flash to DRAM"*). The embedded CPU will jump to the appropriate DRAM address to execute the code once the Boot Loader shadowing task is done.

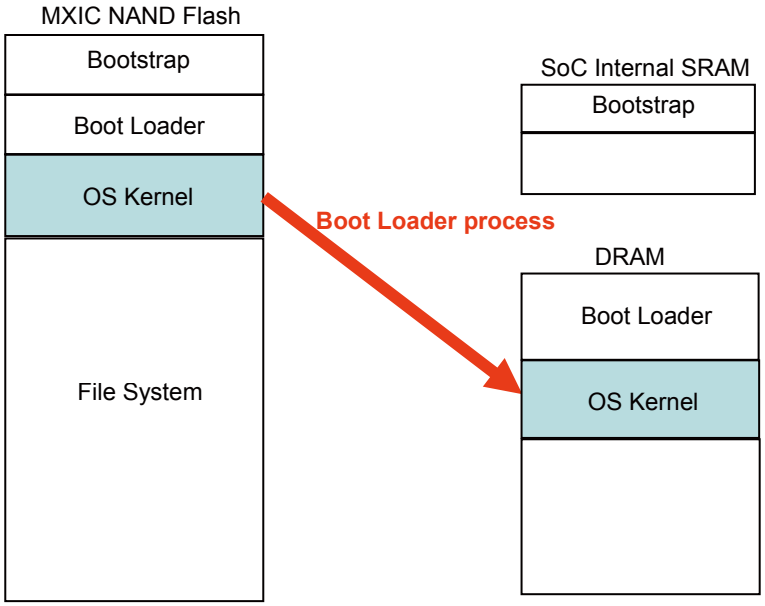
Figure 6: Shadowing Boot Loader from NAND Flash to DRAM



Stage 3: Boot Loader

The boot process of Stage 3 is heavily dependent on the OS. Usually the boot loader shadows the OS kernel first before completing its boot process. There are a number of factors the boot loader should take into account such as memory mapping (to support boot-from-NAND), NAND flash operations supported (so that it can read and write to the flash memory device), and environment data storage location (within a single block), etc. The Boot Loader shadows OS kernel code from NAND flash memory to DRAM (*"Figure 7: Shadowing OS Kernel from NAND Flash to DRAM"*). The embedded MCU (Memory Control Unit) will jump to the shadowing OS kernel in DRAM after the boot loader completes its task.

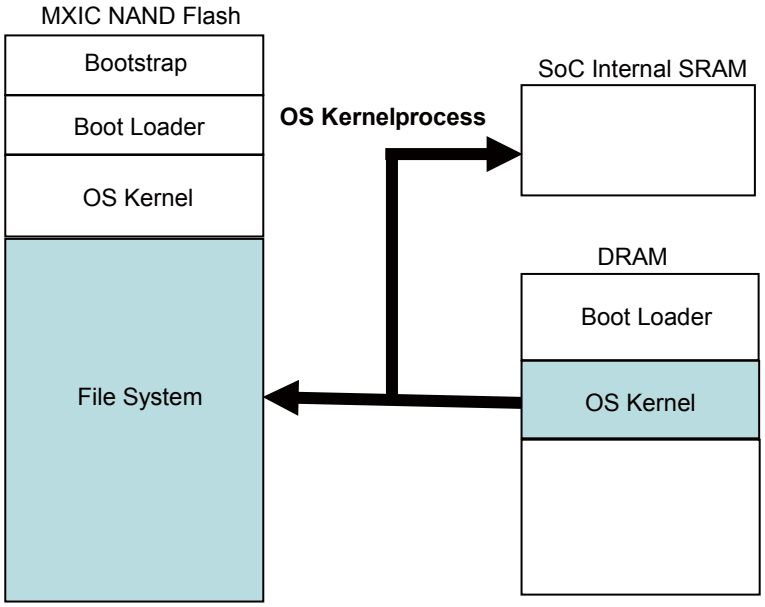
Figure 7: Shadowing OS Kernel from NAND Flash to DRAM



Stage 4: Operating System

The final stage of the boot process starts with OS kernel initial execution within DRAM. From this stage, the OS begins to control the whole system (*"Figure 8: OS Kernel Process Taking System Control"*). The SoC internal SRAM is fully utilized by the OS Kernel. The boot process is complete and ready for user applications after the final stage is finished.

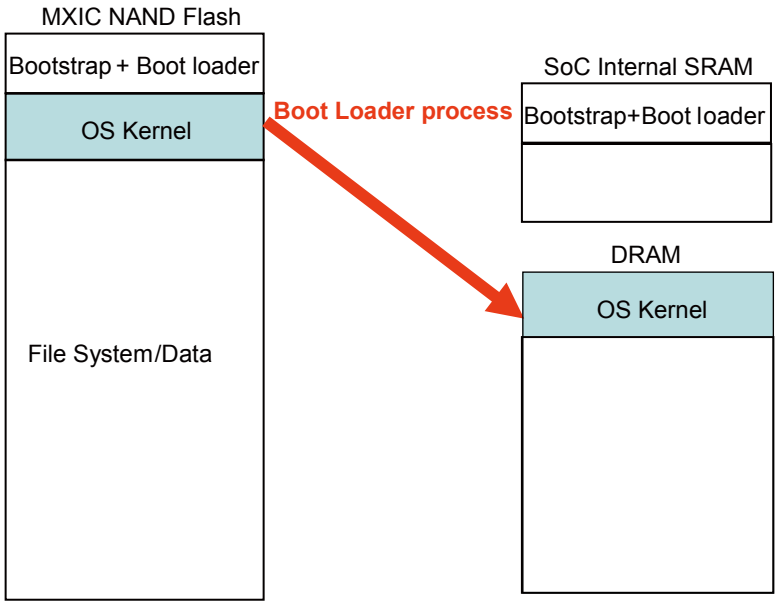
Figure 8: OS Kernel Process Taking System Control



Combination of Bootstrap and Boot Loader

There are some embedded systems that focus on a narrow sets of applications with limited interrupts, tasks and events. The bootstrap and boot loader can be combined as one unit in the SoC's internal SRAM, if the size of the combined code will fit in the SRAM. In this way, the system designer can simplify the boot process and load the OS Kernel to DRAM directly, after the boot loader has successfully beencopied into SRAM (*"Figure 9: Combined Bootstrap and Boot Loader Operation"*).

Figure 9: Combined Bootstrap and Boot Loader Operation



Maximum Reliability of Code in NAND Flash Memory

Typically, NAND flash memory is not as reliable as NOR flash memory. NAND flash memory has higher raw bit errors rates (RBER) and needs error correction codes (ECC) to reduce it. NAND devices also suffer internal Read/Write disturbance effects that reduce data retention time. To maximize the reliability of code, system designers should implement some of the following recommendations:

- To reduce the numbers of reads from a NAND device, copy the code into SRAM/DRAM if the code needs to be repeatedly accessed. This will reduce the possibility of read disturbance errors in NAND flash memory.
- Always enable the ECC function within the SoC for the NAND flash interface. Even a single bad bit in the code can cause a system failure, so error correction should be maximized in code storage areas of NAND flash memory.
- When programming code into NAND flash memory, it is imperative to perform a read-verify operation to compare the contents in NAND flash against the original binary image.
- To avoid multiple programming operations within a single page, program each page in its entirety using a single program operation. This will reduce the possibility of program disturb data degradation.

Summary

To take advantage of NAND's low cost and high performance, more frequently, embedded systems are adopting NAND as the boot device. By employing Macronix's SLC NAND flash memory and implementation recommendations, system designers can achieve a reliable boot sequence from their Macronix NAND flash memory.

Revision History

Revision No.	Description	Page	Date
REV. 1	Initial Release	ALL	12th, Dec., 2013

Except for customized products which have been expressly identified in the applicable agreement, Macronix's products are designed, developed, and/or manufactured for ordinary business, industrial, personal, and/or household applications only, and not for use in any applications which may, directly or indirectly, cause death, personal injury, or severe property damages. In the event Macronix products are used in contradicted to their target usage above, the buyer shall take any and all actions to ensure said Macronix's product qualified for its actual use in accordance with the applicable laws and regulations; and Macronix as well as it's suppliers and/or distributors shall be released from any and all liability arisen therefrom.

For the contact and order information, please visit Macronix's Web site at: http://www.macronix.com