

# Program NAND with ROM Programmer Implementation Guide

by *Multimedia Applications Division*  
*Freescale Semiconductor, Inc.*  
*Austin, TX*

## 1 Introduction

At the time of manufacturing a hardware board, no data is available on the NAND Flash. During the engineering stage, the Advanced ToolKit (ATK) is used to download the XLDR and EBOOT (for WinCE) to the board, and then the board can be run. But, using ATK to download the full run-time NK image is too slow and not appropriate for mass production.

For mass production, the NAND Flash must be programmed with the ROM programmer, prior to mounting it onto to the Printed Circuit Board (PCB). The XLDR, EBOOT, and NK must also be programmed into the NAND Flash simultaneously.

To work with most of the vendors of ROM programmer, the bad block skip format is used to program the NAND Flash data, and all the main data and the spare data is provided to the ROM programmer. Hence, the programmer need not worry about the ECC method, and can just program the given data to the NAND Flash.

This application note discusses methods to:

- Program the full run-time image to NAND Flash by using the ROM programmer.

### Contents

1. Introduction .....	1
2. Design References .....	2
3. Revision History .....	14

- Prepare the full run-time image files with the format supported by the ROM programmer.

The reference codes given in this application note are based on Freescale’s i.MX27ADS F15 BSP, and Micron 2 Kbyte page size NAND Flash MT29F4G08ABC.

## 2 Design References

This section includes the framework description and a porting example. It also describes the NAND Flash driver modification and the NAND Dump application with the help of code, and finally provides the T9000 programmer settings.

### 2.1 Framework Description

The following key points must be considered to make the programmed run-time image executable:

- Each region (XLDR, EBOOT, NK, and so on) can work with flexible start block address, but not with a fixed block address.
- The storage region and the bad block table must be created while running the program for the first time, since it is difficult for the programmer to process them.
- The dumped run-time image file must include all the main data and the spare data (This is different for the initial image file).

### 2.2 Porting Example

In this section, the WinCE system NAND Flash layout is considered as an example:

Table 1 shows the NAND Flash layout.

**Table 1. NAND Flash Layout**

XLDR	EBOOT	IPL	NK	Storage (FATFS)	BOOT CONFIG
------	-------	-----	----	-----------------	-------------

The WinCE system is based on the 2 Kbyte page NAND Flash (MT29F4G08ABC, a total of 4096 blocks) starting from the NAND Flash block 0 which is XLDR (one block), followed by EBOOT (two blocks), IPL (two blocks), NK (20 Mbyte, 160 blocks), and the BOOT CONFIG data which is the last block. The storage region is between the NK and BOOT CONFIG blocks.

For the final run-time image, the XLDR, EBOOT, IPL, and NK regions are read-only regions. These data must be programmed by the ROM programmer, and the data constitutes a total of 165 NAND blocks.

If all the NAND Flash blocks are good blocks, then the corresponding NAND Flash addresses are as follows:

- Block 0—XLDR
- Block 1–2—EBOOT
- Block 3–4—IPL
- Block 5–164—NK
- Block 165–4094—Storage

- Block 4095—BOOT CONFIG

In this ideal case, the programmer programmed the entire 165 blocks of NAND Flash block by block.

Assume that there are bad blocks—1,3,5,7,10, and 100—and the corresponding NAND Flash addresses for these bad blocks are as follows:

- Block 0—XLDR
- Block 2, 4—EBOOT
- Block 6, 8—IPL
- Block 9, 11–99, 101–170—NK
- Block 171–4094—Storage
- Block 4095—BOOT CONFIG

In this case, the programmer skips the bad blocks, but processes the entire 171 blocks, although the raw data consists of only 165 blocks.

In the above two cases, the starting block addresses for the regions EBOOT, IPL, NK, and Storage are different.

Hence, the BSP must be modified to support these different starting block addresses. This implies that the EBOOT region must start from the second good block and not the second block, and the NK region must start from the sixth good block and not the sixth block, and so on.

In the EBOOT and BSP code, the programmer must calculate the starting address as the macros cannot be used to define the starting block address. For the ending block address of each region, the bad blocks also must be considered. The ending block address should be "start block address" + "region size" + "bad blocks in the region".

```
startBlockID = IMAGE_BOOT_EBOOTIMAGE_NAND_OFFSET / flashInfo.dwBytesPerBlock; // This line of code is incorrect.
```

The following code snippet is used to calculate the start block:

```
//-----
//
// Function: NANDGetRealBlockAddress
//
// This function find the real nand flash block address from the input logical
// address, skip bad blocks. If there's no bad block, the read block address is
// same as the logical address.
//
// Parameters:
//     dwBlockLogAddress
//     [in] Nand block logical address in flash memory.
//
// Returns:
//     If success, return the real block address, skipped bad blocks.
//     If failure, return INVALID_BLOCK_ID.
//
//-----
static DWORD NANDGetRealBlockAddress(DWORD dwBlockLogAddress)
{
    FlashInfo flashInfo;
    BLOCK_ID blockID, endBlockID, badblockNum;
```

```

// Check for NAND device availability
if(!g_bNandExist)
{
    EdbgOutputDebugString("WARNING: NAND device doesn't exist - unable to get real
block address.\r\n");
    return (INVALID_BLOCK_ID);
}
if(!FMD_GetInfo(&flashInfo))
{
    EdbgOutputDebugString("ERROR: Unable to get NAND flash information.\r\n");
    return (INVALID_BLOCK_ID);
}
endBlockID = dwBlockLogAddress;
badblockNum = 0;
for(blockID = 0; blockID < endBlockID; blockID ++)
{
    if(blockID >= flashInfo.dwNumBlocks)
    {
        return (INVALID_BLOCK_ID);
    }
    if(FMD_GetBlockStatus(blockID) == BLOCK_STATUS_BAD)
    {
        EdbgOutputDebugString("INFO: Found bad NAND flash block
0x%x].\r\n", blockID);
        endBlockID ++;
        badblockNum ++;
    }
}
return (dwBlockLogAddress + badblockNum);
}

startBlockID = NANDGetRealBlockAddress(IMAGE_BOOT_EBOOTIMAGE_NAND_OFFSET /
flashInfo.dwBytesPerBlock);

```

Since the XLDR, EBOOT, IPL, and NK are the only regions that are programmed, the BOOT CONFIG region is not available for programming. Also, the Storage region is not formatted. When booting up the WinCE system for the first time, the EBOOT can set the default BOOT CONFIG data, and the NK region can create a partition and format it automatically. By this time, all the functions are ready to run.

The most important step in this solution is to prepare the image file for the ROM programmer, which is as follows:

Table 2 shows the main space data and the spare space data of the MT29F4G08ABC NAND Flash page in bytes.

**Table 2. Main Space and Spare Space Data (in bytes)**

2048 bytes	64 bytes
------------	----------

While using the ROM programmer to read/write the NAND Flash, the data in each page is divided into two parts—the first 2048 bytes represent the main space data and the remaining 64 bytes represent the spare space data. Also, the bad block flag is the first byte in the 64 bytes of spare data.

But for Freescale NAND Flash Controller (NFC), when reading/writing the NAND Flash page, the raw NAND page (2048 + 64 bytes) is processed as shown in Table 3:

Table 3 shows the NAND Flash page layout.

**Table 3. NAND Flash Page Layout**

512 bytes	16 bytes						
-----------	----------	-----------	----------	-----------	----------	-----------	----------

This implies that, when writing 2048 bytes of user data into the NAND Flash by the NFC, the data is divided into four discontinuous 512 bytes. Also, when reading/writing data to the ROM programmer, the 2048 bytes of main space data is split into "512 MData + 16 SData + 512 MData + 16 SData + 512 MData + 16 SData + 464 MData" and the 64 bytes of spare space data is split into "48 MData + 16 SData".

For the BISWAP in WinCE, the first byte of the 48 MData is the bad block and the real data in this byte is at the fifth byte of the 16 SData. MData implies the user data and SData implies the spare data, and in WinCE, they include the sector information and ECC data.

So, if the ROM programmer is provided with only an image file (in WinCE, it is NK.NB0, the user data), then it is very difficult for the programmer to process. An easy and reliable way is to provide a combined file, which includes all the user data, WinCE sector information data, and the ECC data. The ROM programmer just programs this combined file into the target NAND blocks, and the programmer need not worry about the BISWAP, bad block table, or ECC. All the data that should be programmed into the NAND Flash spare space are already included in the combined source file.

When preparing the combined file for the ROM programmer, it is required to dump the raw NAND Flash data from a workable device.

In WinCE NAND Flash driver, a new function `NANDDumpBlock()` is added to implement the dump block data function, and an application `NandDump` calls `DeviceIOCTL` to make the NAND driver to dump the Flash data block by block, and write into a file.

## 2.3 NAND Flash Driver Modification

The following code snippet is the reference code added in the NAND Flash driver. Here, a new `IOCTL` is defined for the dump function:

```
#define IOCTL_FMD_DUMP_NAND                IOCTL_DISK_USER(17)
//-----
//
// Function:  NANDGetRealBlockAddress
//
// This function find the real nand flash block address from the input logical
// address, skip bad blocks. If there's no bad block, the read block address is
// same as the logical address.
//
// Parameters:
//     dwBlockLogAddress
//     [in] Nand block logical address in flash memory.
//
// Returns:
//     If success, return the real block address, skipped bad blocks.
//     If failure, return INVALID_BLOCK_ID.
//-----
static DWORD NANDGetRealBlockAddress(DWORD dwBlockLogAddress)
{
```

## Design References

```

BLOCK_ID blockID, endBlockID, badblockNum;
endBlockID = dwBlockLogAddress;
badblockNum = 0;
for(blockID = 0; blockID < endBlockID; blockID ++)
{
    if(blockID >= NAND_BLOCK_CNT)
    {
        return (INVALID_BLOCK_ID);
    }
    if(FMD_GetBlockStatus(blockID) == BLOCK_STATUS_BAD)
    {
        endBlockID ++;
        badblockNum ++;
    }
}
return (dwBlockLogAddress + badblockNum);
}
//-----
//
// Function: NFCDumpSpare
//
// This function reads the NAND flash controller spare area.
//
// Parameters:
//     pSectorSpareBuff
//     [out] Buffer containing sector spare data read.
//
// Returns:
//     None.
//
//-----
static VOID NFCDumpSpare(LPBYTE pSectorSpareBuff)
{
    if(pSectorSpareBuff != NULL)
    {
        memcpy(pSectorSpareBuff, (PBYTE>(&g_pNFC->SPARE[0][0]),
NAND_PAGE_SPARE_SIZE);
    }
}
//-----
//
// Function: NFCDumpSector
//
// This function reads the requested sector data and metadata from the
// flash media.
//
// Parameters:
//     startSectorAddr
//     [in] The starting physical sector address to read.
//
//     pSectorBuff
//     [out] Pointer to the buffer that contains the sector data read
//     from flash memory. Set to NULL if this data is not needed.
//
//     pSectorSpareBuff
//     [out] Buffer for an array of all spare sector data. There
//     are 16 bytes spare data for every sector that is to be read.

```

```

//          Set to NULL if this data is not needed.
//
//          dwNumSectors
//          [in] Number of sectors to read.
//
// Returns:
//          Returns TRUE on success. Returns FALSE on failure.
//
//-----
static BOOL NFCDumpSector(SECTOR_ADDR startSectorAddr, LPBYTE pSectorBuff, LPBYTE
SectorSpareBuff, DWORD dwNumSectors)
{
    SECTOR_ADDR SectorAddr = startSectorAddr;
    BOOL bSeqRead = FALSE;
    int i;
    UINT32 PageAddr;
    UINT32 ColAddr;
//    RETAILMSG(1, (TEXT("NFCDumpSector() start.\r\n")));
//    RETAILMSG(1, (TEXT("SectorAddr = %d.\r\n"), SectorAddr));
    if((pSectorBuff == NULL) && (pSectorSpareBuff == NULL))
        return FALSE;
    NFCSetClock(TRUE);
    while(dwNumSectors --)
    {
        PageAddr = SectorAddr;
        ColAddr = 0;
        if(pSectorBuff != NULL)
        {
            // If read access requires command and address to be sent
            if(!bSeqRead)
            {
                NF_CMD(CMD_READ);           // Send page read command.
                NF_ADDR_COL(ColAddr); // Send column address
                NF_ADDR_PAGE(PageAddr); // Send page address
#ifdef NAND_LARGE_PAGE
                NF_CMD(CMD_READ_2CYCLE); // Send 2nd cycle read command
#endif
            }
            // Read page data into NFC buffer
            for(i = 0; i < NAND_PAGE_SIZE / NANDFC_MAIN_BUFFER_SIZE; i ++)
            {
                NF_BUF_ADDR(i);
                NF_RD_PAGE();
                // Check for uncorrectable ECC errors in main and spare
                areas
                if(EXTREG16BF(&(g_pNFC->ECC_STATUS_RESULT),
NANDFC_ECC_STATUS_RESULT_ERM) == NANDFC_ECC_STATUS_RESULT_ERM_2BIT_ERR)
                {
                    //RETAILMSG(TRUE, (_T("Uncorrectable ECC error at
Sector address: 0x%x.\r\n"), SectorAddr));
                    NFCSetClock(FALSE);
                    return FALSE;
                }
            }
            // Move page data from NFC buffer sector buffer
            if(pSectorBuff != NULL)
            {

```

```

        NFCReadMain(pSectorBuff);
        pSectorBuff += NAND_PAGE_SIZE;
    }
    if(pSectorSpareBuff != NULL)
    {
        NFCDumpSpare(pSectorSpareBuff);
        pSectorSpareBuff += NAND_PAGE_SPARE_SIZE;
    }
}
else if(pSectorSpareBuff != NULL)
{
    NF_BUF_ADDR(0);
#ifdef NAND_LARGE_PAGE
    NF_CMD(CMD_READ2);          // Send spare read command.
    NF_ADDR_COL(ColAddr);      // Send column address
    NF_ADDR_PAGE(PageAddr);    // Send page address
#else
    ColAddr = NAND_PAGE_SIZE + 16 * 3;
    NF_CMD(CMD_READ);          // Send read command
    NF_ADDR_COL(ColAddr);      // Send column address
    NF_ADDR_PAGE(PageAddr);    // Send page address
    NF_CMD(CMD_READ_2CYCLE);   // Send 2nd cycle read command
#endif

    // Read page data into NFC buffer
    NF_RD_SPARE();
    NFCDumpSpare(pSectorSpareBuff);
    pSectorSpareBuff += NAND_PAGE_SPARE_SIZE;
}
++ SectorAddr;
bSeqRead = NF_READ_SEQ(SectorAddr);
}
NFCSetClock(FALSE);
// RETAILMSG(1, (TEXT("NFCDumpSector() end.\r\n")));
return TRUE;
}
//-----
//
// Function:  NANDDumpBlock
//
// This function read block data from Nandflash
//
// Parameters:
//     dwTargetBlock
//         [in] Block address in flash memory to be read.
//
//     pMain
//         [out] Buffer for main data.
//
//     pSpare
//         [out] Buffer for spare data.
//
// Returns:
//     TRUE indicates success. FALSE indicates failure.
//
//-----
BOOL NANDDumpBlock(DWORD dwTargetBlock, BYTE *pMain, BYTE *pSpare)
{

```

```

BLOCK_ID blockID;
SECTOR_ADDR sectorAddr;
DWORD dwBlockState;
// Calculate the physical block
blockID = NANDGetRealBlockAddress(dwTargetBlock);
RETAILMSG(1, (TEXT("INFO: Dump NAND flash blocks [0x%x].\r\n"), blockID));
while(blockID < NAND_BLOCK_CNT)
{
    dwBlockState = FMD_GetBlockStatus(blockID);
    if(dwBlockState == BLOCK_STATUS_BAD)
    {
        blockID ++;
        continue;
    }
    else
    {
        break;
    }
}
// Compute sector address based on current physical block
sectorAddr = BLOCK_TO_SECTOR(blockID);
if(!NFCDumpSector(sectorAddr, pMain, pSpare, NAND_PAGE_CNT))
{
    RETAILMSG(1, (TEXT("\r\nERROR: Failed to dump nand.\r\n")));
    return (FALSE);
}
return (TRUE);
}
BOOL FMD_OEMIoControl(DWORD dwIoControlCode, PBYTE pInBuf, DWORD nInBufSize, PBYTE pOutBuf,
DWORD nOutBufSize, PDWORD pBytesReturned)
{
    BOOL bRet = FALSE;
    DWORD dwErrorCode = ERROR_SUCCESS;
    UPDATE_PACKAGE * pPackage = NULL;
    DWORD dwTargetBlock;
    EBOOT_CFG eBootCfg = {0};
// RETAILMSG(1, (TEXT("OEMFMD_OemIoControl() start.\r\n")));
// RETAILMSG(1, (TEXT("dwIoControlCode = 0x%x.\r\n"), dwIoControlCode));
    switch(dwIoControlCode)
    {
        case IOCTL_FMD_DUMP_NAND:
            RETAILMSG(1, (TEXT("INFO: IOCTL_FMD_DUMP_NAND.\r\n")));
            if((pInBuf == NULL) || (nInBufSize < sizeof(DWORD)) || (pOutBuf ==
NULL) || (nOutBufSize < NAND_BLOCK_SIZE + NAND_PAGE_SPARE_SIZE * NAND_PAGE_CNT))
            {
                dwErrorCode = ERROR_INVALID_PARAMETER;
                if(pBytesReturned)
                {
                    *pBytesReturned = sizeof(DWORD);
                }
                break;
            }
            dwTargetBlock = *((DWORD *)pInBuf);
            bRet = NANDDumpBlock(dwTargetBlock, pOutBuf, pOutBuf +
NAND_BLOCK_SIZE);
            break;
        default:

```

```

        dwErrorCode = ERROR_NOT_SUPPORTED;
        break;
    }

    if(bRet == FALSE)
        SetLastError(dwErrorCode);
//    RETAILMSG(1, (TEXT("OEMFMD_OemIoControl() end.\r\n")));
    return bRet;
}

```

## 2.4 NAND Dump Application

The following is the sample code of the `NandDump` application. To fit different ROM programmers, the NAND data is dumped into two formats:

- Two file format—one for the main space data and other for the spare space data
- One file format—the combined file format: 2 Kbyte main data + 64 Kbyte spare data + 2 Kbyte main data + 64 Kbyte spare data + ... ..

In this example, the total dumped data comprises 165 blocks, starting from block 0. Here, 165 blocks imply 165 good block data. If there are no bad blocks from block 0 to block 164, then the total processed blocks are 165. Supposing there are two bad blocks, then the total processed blocks are 165+2:

```

#include <windows.h>
#include <ceddk.h>
#include <Storemgr.h>
#include <diskio.h>
#include "bsp.h"
#include "socarm_types.h"
#include "mx27_gpio.h"
#include "mx27_ddk.h"
#include "MT29F4G08ABC.h"

#define NAND_READ_BLOCK_START0
#define NAND_READ_BLOCK_NUMBERS((IMAGE_BOOT_XLDRIMAGE_NAND_SIZE +
IMAGE_BOOT_EBOOTIMAGE_NAND_SIZE + IMAGE_BOOT_IPLIMAGE_NAND_SIZE +
IMAGE_BOOT_NKIMAGE_NAND_SIZE) / NAND_BLOCK_SIZE)

#define IOCTL_FMD_DUMP_NAND          IOCTL_DISK_USER(17)

static BOOL DumpToFile(DWORD dwFileNum)
{
    HANDLE hDevice = INVALID_HANDLE_VALUE;
    BYTE * pDataBuffer = NULL;
    TCHAR pszMainFileName[] = TEXT("\\ROMIMAGE.IMG");
    TCHAR pszSpareFileName[] = TEXT("\\ROMIMAGE.SPR");
    TCHAR pszComposeFileName[] = TEXT("\\ROMIMAGE.BIN");
    HANDLE hMainFile = INVALID_HANDLE_VALUE;
    HANDLE hSpareFile = INVALID_HANDLE_VALUE;
    DWORD dwBlock, i, dwBufferSize;
    DWORD dwBytesRequired;
    BOOL bRet;

    if(dwFileNum < 1 || dwFileNum > 2)
    {
        RETAILMSG(1, (TEXT("Invalid file number.\r\n")));
        return FALSE;
    }
}

```

```

RETAILMSG(1, (TEXT("Dump Start.\r\n")));
DeleteFile(pszMainFileName);
DeleteFile(pszSpareFileName);
DeleteFile(pszComposeFileName);
Sleep(1);
dwBufferSize = NAND_BLOCK_SIZE + NAND_PAGE_SPARE_SIZE * NAND_PAGE_CNT;
pDataBuffer = (BYTE *)malloc(dwBufferSize);
if(pDataBuffer == NULL)
{
    RETAILMSG(1, (TEXT("Failed to malloc buffer.")));
    return FALSE;
}
// Open device
hDevice = OpenStore(TEXT("NSFlash"));
if(hDevice == INVALID_HANDLE_VALUE)
{
    hDevice = CreateFile(
        TEXT("DSK1:"),
        GENERIC_READ | GENERIC_WRITE,
        FILE_SHARE_READ | FILE_SHARE_WRITE,
        NULL,
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED,
        INVALID_HANDLE_VALUE);
    if(hDevice == INVALID_HANDLE_VALUE)
    {
        RETAILMSG(1, (TEXT("Can't find nand flash driver.\r\n")));
        free((VOID *)pDataBuffer);
        return FALSE;
    }
    RETAILMSG(1, (TEXT("CreateFile for DSK1: successfully.\r\n")));
}
else
{
    RETAILMSG(1, (TEXT("OpenStore for NSFlash successfully.\r\n")));
}
// Dump Block Main Data and spare
if(dwFileNum == 2)
{
    hMainFile = CreateFile(pszMainFileName, GENERIC_WRITE, 0, NULL, CREATE_NEW,
FILE_ATTRIBUTE_NORMAL, NULL);
    hSpareFile = CreateFile(pszSpareFileName, GENERIC_WRITE, 0, NULL, CREATE_NEW,
FILE_ATTRIBUTE_NORMAL, NULL);
}
else
{
    hMainFile = CreateFile(pszComposeFileName, GENERIC_WRITE, 0, NULL,
CREATE_NEW, FILE_ATTRIBUTE_NORMAL, NULL);
}
for(dwBlock = NAND_READ_BLOCK_START; dwBlock < NAND_READ_BLOCK_START +
NAND_READ_BLOCK_NUMBERS; dwBlock ++)
{
    bRet = DeviceIoControl(hDevice, IOCTL_FMD_DUMP_NAND, &dwBlock, sizeof(DWORD),
pDataBuffer, dwBufferSize, &dwBytesRequired, NULL);
    if(bRet == TRUE)
    {
        for(i = 0; i < NAND_PAGE_CNT; i ++)
```

```

        {
            // Write Main data
            if(hMainFile != INVALID_HANDLE_VALUE)
            {
                WriteFile(hMainFile, pDataBuffer + (i *
NAND_PAGE_SIZE) + (512 * 0), 512, &dwBytesRequired, NULL);
                WriteFile(hMainFile, pDataBuffer + (i *
NAND_PAGE_SPARE_SIZE) + NAND_BLOCK_SIZE + (16 * 0), 16, &dwBytesRequired, NULL);
                WriteFile(hMainFile, pDataBuffer + (i *
NAND_PAGE_SIZE) + (512 * 1), 512, &dwBytesRequired, NULL);
                WriteFile(hMainFile, pDataBuffer + (i *
NAND_PAGE_SPARE_SIZE) + NAND_BLOCK_SIZE + (16 * 1), 16, &dwBytesRequired, NULL);
                WriteFile(hMainFile, pDataBuffer + (i *
NAND_PAGE_SIZE) + (512 * 2), 512, &dwBytesRequired, NULL);
                WriteFile(hMainFile, pDataBuffer + (i *
NAND_PAGE_SPARE_SIZE) + NAND_BLOCK_SIZE + (16 * 2), 16, &dwBytesRequired, NULL);
                WriteFile(hMainFile, pDataBuffer + (i *
NAND_PAGE_SIZE) + (512 * 3), (512 - 16 * 3), &dwBytesRequired, NULL);
            }
            // Write Spare data
            if(dwFileNum == 2)
            {
                if(hSpareFile != INVALID_HANDLE_VALUE)
                {
                    WriteFile(hSpareFile, pDataBuffer + (i *
NAND_PAGE_SIZE) + (512 * 3) + (512 - 16 * 3), (16 * 3), &dwBytesRequired, NULL);
                    WriteFile(hSpareFile, pDataBuffer + (i *
NAND_PAGE_SPARE_SIZE) + NAND_BLOCK_SIZE + (16 * 3), 16, &dwBytesRequired, NULL);
                }
            }
            else
            {
                if(hMainFile != INVALID_HANDLE_VALUE)
                {
                    WriteFile(hMainFile, pDataBuffer + (i *
NAND_PAGE_SIZE) + (512 * 3) + (512 - 16 * 3), (16 * 3), &dwBytesRequired, NULL);
                    WriteFile(hMainFile, pDataBuffer + (i *
NAND_PAGE_SPARE_SIZE) + NAND_BLOCK_SIZE + (16 * 3), 16, &dwBytesRequired, NULL);
                }
            }
        }
    }
    else
    {
        RETAILMSG(1, (TEXT("Read Block %d failed.\r\n"), dwBlock));
    }
}
if(hMainFile != INVALID_HANDLE_VALUE)
{
    CloseHandle(hMainFile);
}
if(hSpareFile != INVALID_HANDLE_VALUE)
{
    CloseHandle(hSpareFile);
}
free((VOID *)pDataBuffer);
RETAILMSG(1, (TEXT("Dump finished.\r\n")));

```

```

        return TRUE;
    }
    int WINAPI WinMain(
        HINSTANCE hinst,
        HINSTANCE hinstPrev,
        LPWSTR szCmdLine,
        int iCmdShow
    )
    {
        // DumpToFile(2); // This will dump the NAND into two files, one for main data and one
        // for spare data.
        DumpToFile(1); // This will dump the NAND into one file, 2Kbyte main data + 64Kbyte
        // spare data + 2Kbyte main data + 64Kbyte spare data + ... ..
        return 0;
    }

```

## 2.5 T9000 Programmer Setting

T9000 is a ROM programmer and the details are available at <http://www.sinaentek.com/>.

As an example, the dumped NAND Flash file is 165 (0xA5) blocks in one combined file format. The ROMIMAGE.BIN file is 22,302,720 bytes (165 × (128 Kbyte + 4 Kbyte)) in size.

Figure 1 shows the T9000 programmer settings.

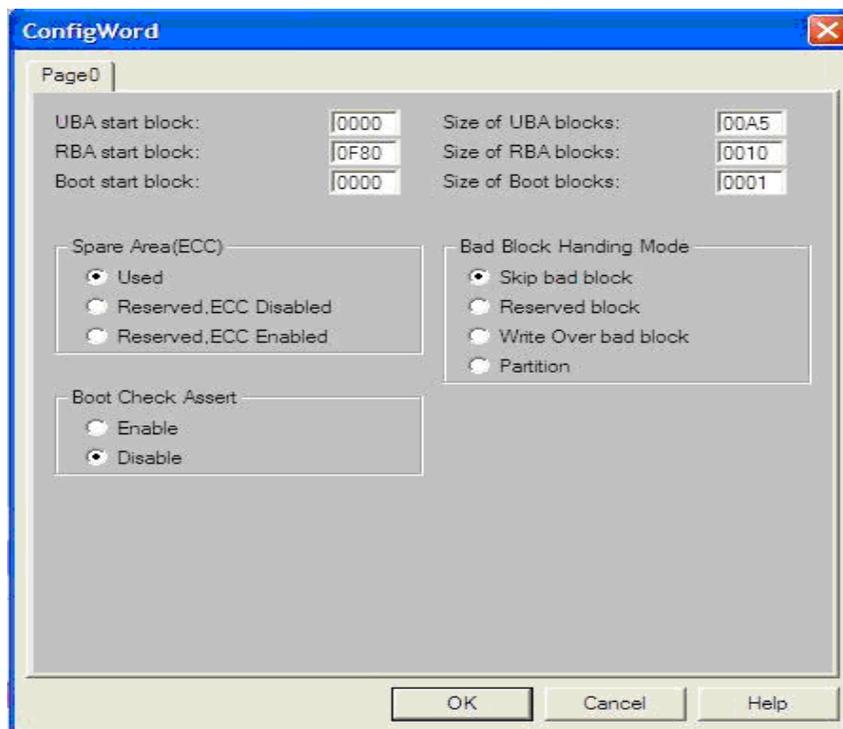


Figure 1. T9000 Programmer Settings

### 3 Revision History

Table 4 provides a revision history for this application note.

**Table 4. Document Revision History**

Rev. Number	Date	Substantive Change(s)
0	05/2010	Initial release

**THIS PAGE INTENTIONALLY LEFT BLANK**

## How to Reach Us:

### Home Page:

[www.freescale.com](http://www.freescale.com)

### Web Support:

<http://www.freescale.com/support>

### USA/Europe or Locations Not Listed:

Freescale Semiconductor, Inc.  
Technical Information Center, EL516  
2100 East Elliot Road  
Tempe, Arizona 85284  
1-800-521-6274 or  
+1-480-768-2130  
[www.freescale.com/support](http://www.freescale.com/support)

### Europe, Middle East, and Africa:

Freescale Halbleiter Deutschland GmbH  
Technical Information Center  
Schatzbogen 7  
81829 Muenchen, Germany  
+44 1296 380 456 (English)  
+46 8 52200080 (English)  
+49 89 92103 559 (German)  
+33 1 69 35 48 48 (French)  
[www.freescale.com/support](http://www.freescale.com/support)

### Japan:

Freescale Semiconductor Japan Ltd.  
Headquarters  
ARCO Tower 15F  
1-8-1, Shimo-Meguro, Meguro-ku  
Tokyo 153-0064  
Japan  
0120 191014 or  
+81 3 5437 9125  
[support.japan@freescale.com](mailto:support.japan@freescale.com)

### Asia/Pacific:

Freescale Semiconductor China Ltd.  
Exchange Building 23F  
No. 118 Jianguo Road  
Chaoyang District  
Beijing 100022  
China  
+86 10 5879 8000  
[support.asia@freescale.com](mailto:support.asia@freescale.com)

### For Literature Requests Only:

Freescale Semiconductor  
Literature Distribution Center  
1-800 441-2447 or  
+1-303-675-2140  
Fax: +1-303-675-2150  
[LDCForFreescaleSemiconductor@hibbertgroup.com](mailto:LDCForFreescaleSemiconductor@hibbertgroup.com)

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters which may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals" must be validated for each customer application by customer's technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

Freescale, the Freescale logo, CodeWarrior, ColdFire, PowerQUICC, StarCore, and Symphony are trademarks of Freescale Semiconductor, Inc. Reg. U.S. Pat. & Tm. Off. CoreNet, QorIQ, QUICC Engine, and VortiQa are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners. ARM is the registered trademark of ARM Limited.

© 2010 Freescale Semiconductor, Inc.

