AN_653X_006 OCTOBER 2008

# Code for Bank-Switching with the 653X-Series

This document describes how to use banked code with a 71M6531. Similar principles apply for the 71M6533 and 71M6534.

## Introduction

The 80515 microcontroller contained in the TERIDIAN 71M653X Energy Meter chips can only address 64 KB of code. This Application Note explains how to design firmware with more than 64 KB of code for the 71M653X Energy Meter chips.

## Theory of Operation

### The Hardware

In the 71M6531, there is a 32-KB area from code address 0x0000 to 0x7FFF. The code in this area is always available to the 8051. This area is called “common” and it is the same memory area as “bank 0”. Since it is always present, it never needs to be switched into the bank area.

The upper memory area with the address range 0x8000 to 0xFFFF can be allocated to any 32-KB memory bank. The 32-KB bank is selected by writing the bank’s number in the register *FL_BANK* (SFR at 0xB6). After this, the bank’s code is visible to the MPU in addresses 0x8000 to 0xFFFF.

The 71M6531 has four 32-KB banks (128 KB bytes total). Bank 0 is the common area. Banks 1, 2, and 3 are the banked code areas selected by *FL_BANK* (see Table 1).

The 71M6534 has eight 32-KB banks (256 KB total). Bank 0 is the common area. Banks 1 through 7 are the banked code areas selected by *FL_BANK* (see Table 1).

A reset sets *FL_BANK* to 1, so any 71M653X IC can run 64 KB of non-bank-switching code.

The 71M653X ICs have two write-protect registers to protect ranges at the beginning and end of flash, called *BOOT_SIZE* and *CE_LCTN*, activated by the enable bits *WRPROT_CE* and *WRPROT_BT*, in SFR 0xB2.

| *FL_BANK [2:0]* | Address Range for Lower Bank (Common) (0x000-0x7FFF) | Address Range for Upper Bank (0x8000-0xFFFF) | 6531 128 KB | 6533 128 KB | 6534 256 KB |
|---|---|---|---|---|---|
| 000 | 0x0000-0x7FFF | 0x0000-0x7FFF | X | X | X |
| 001 | 0x0000-0x7FFF | 0x8000-0xFFFF | X | X | X |
| 010 | 0x0000-0x7FFF | 0x10000-0x17FFF | X | X | X |
| 011 | 0x0000-0x7FFF | 0x18000-0x1FFFF | X | X | X |
| 100 | 0x0000-0x7FFF | 0x20000-0x27000 | | | X |
| 101 | 0x0000-0x7FFF | 0x28000-0x2FFFF | | | X |
| 110 | 0x0000-0x7FFF | 0x30000-0x37FFF | | | X |
| 111 | 0x0000-0x7FFF | 0x38000-0x3FFFF | | | X |

**Table 1**

The 71M653X ICs' flash memory are very similar to the ROM arrangement in Keil's example "Banking with Common Area" of chapter 9 (linker) of Keil's "Macro assembler and Utilities" manual.

## The Software

TERIDIAN's demonstration code uses the Keil compiler's standard bank switching system (www.keil.com).

Bank-switching is completely supported by Keil, a major compiler vendor for generic 8051 processors and cores, and Signum, the emulator vendor. Code can be ported from non-banked projects, and full symbolic banked debugging is available.

Keil's scheme puts a "page table" in common memory. Code calls an entry in the page table. The entry is a bit of code that switches to the subroutine's bank, and jumps to the subroutine in the bank.

Keil's linker automatically produces the page table. In TERIDIAN's Demo Code, the size of this table is less than 1 KB.

Code using the page table is slower than native 16-bit code, because it has to set the page register.

Interrupts must start in the common (non-banked) area, because the bank register could have any value.

Calls via function pointers (e.g. "callback routines") are supported, but need to be made global, and mapped to their caller with the linker's overlay functions. Keil's linker often omits callback routines from the page table when it optimizes the page table, and this causes incorrect operation.

Constant values have to be accessed from the same bank, or common code. When accessed from common code the bank has to be switched manually with `switchbank(),` a subroutine in the bank logic.

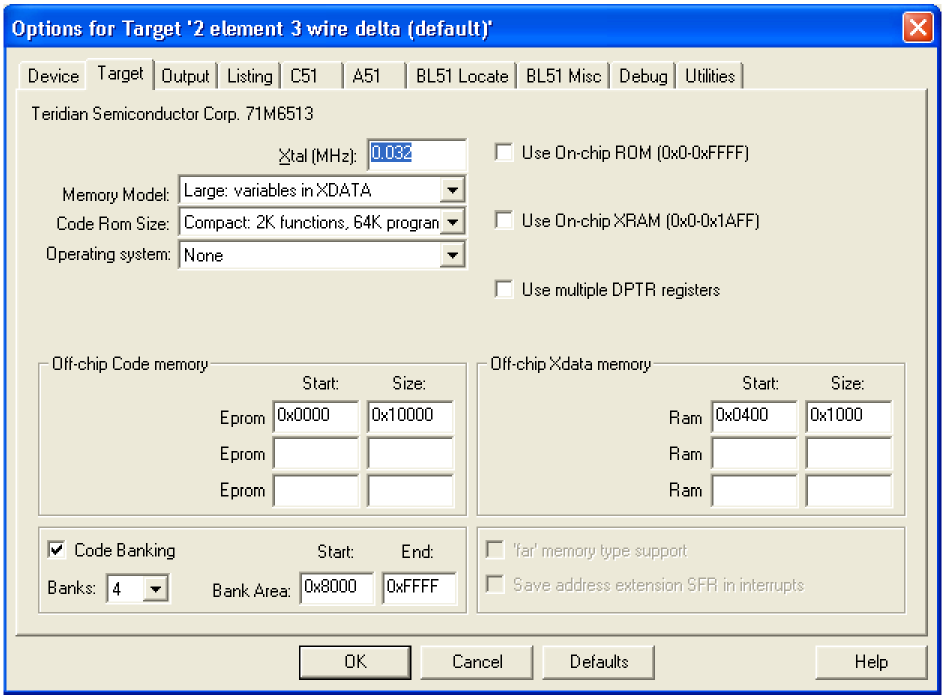
# Design and Coding

## Software Versions

The development software used with these examples was Keil C version 8.03, with the BL51 linker (the Lx51 linker is actually easier to use, but not shown). The Signum emulator software used was version 3.11.04.

## Setup of the Compiler Project

This dialogue is for the project options of a 71M6531, which has four banks (see Figure 1).



Teridian Semiconductor Corp. 71M6513

Xtal (MHz): 0.032

Memory Model: Large: variables in XDATA

Code Rom Size: Compact: 2K functions, 64K progran

Operating system: None

□ Use On-chip ROM (0x0-0xFFFF)

□ Use On-chip XRAM (0x0-0x1AFF)

□ Use multiple DPTR registers

Off-chip Code memory

| | Start: | Size: |
|---|---|---|
| Eprom | 0x0000 | 0x10000 |
| Eprom | | |
| Eprom | | |

Off-chip Xdata memory

| | Start: | Size: |
|---|---|---|
| Ram | 0x0400 | 0x1000 |
| Ram | | |
| Ram | | |

☑ Code Banking

Banks: 4

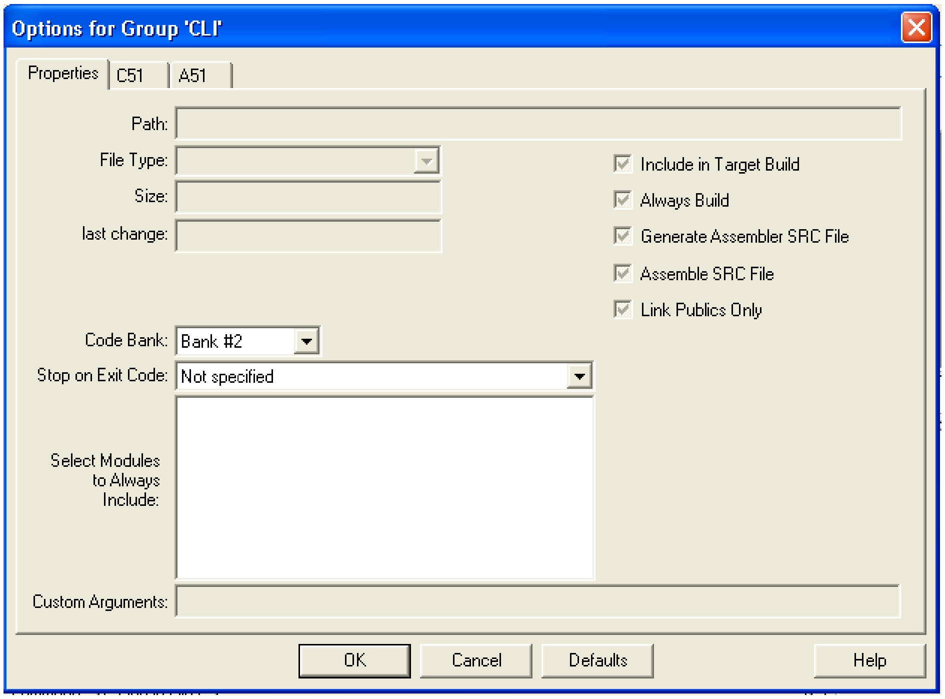
| | Start: | End: |
|---|---|---|
| Bank Area: | 0x8000 | 0xFFFF |

□ 'far' memory type support

□ Save address extension SFR in interrupts

**Figure 1: Project Setup**

When opening individual files by right-clicking on the file names (after opening the group folders listed under “Target”), file options can be edited. These options can be set to assign code to certain pages (as shown in Figure 2: Assigning Bank Numbers to Code



Options for Group 'CLI'

Path:

File Type:

Size:

last change:

☑ Include in Target Build

☑ Always Build

☑ Generate Assembler SRC File

☑ Assemble SRC File

☑ Link Publics Only

Code Bank: Bank #2

Stop on Exit Code: Not specified

Select Modules to Always Include:

Custom Arguments:

**Figure 2: Assigning Bank Numbers to Code**

## Startup Code

TERIDIAN provides special start-up code on the CD-ROMs shipped with the 71M653X Demo Kits. The code can be found at Util\startup_30_banked.a51. This file sets up the bank-switching logic. It must be included in the build. Any other startup.a51 file must be removed.

## Bank-Switching Code

TERIDIAN has already ported Keil’s bank-switching code, Util\L51_bank.a51. TERIDIAN’s version of this file should be included in the build. TERIDIAN has already selected the fastest standard bank-switching method as the default.

During performance testing, TERIDIAN made a good-faith attempt to port the other bank-switching methods in this code, including features needed by Keil’s advanced Lx51 linker. However these versions are not extensively tested.

## Page Table

Keil’s linkers produce the page table automatically, once paging is selected in the Microvision options->target dialogue. The Keil tools usually places the page table at an address of 0x100 in the common bank. It is visible in the linker .m51 file. To see how it works, one can use the emulator to single-step through at a banked function call at the assembly language level.

To call a paged subroutine, Keil’s linker arranges to call one of the entries of the page table. The page table consists of one entry per subroutine. Each entry is a small piece of code that loads the address of the banked subroutine into the 8051 register *DPTR*, and then jumps to paging code. The paging code sets the bank register *FL_BANK*, and then jumps to the banked code’s address contained in the *DPTR*.

Keil's linkers minimize the size of the page table. A subroutine has an entry in the page table only if:

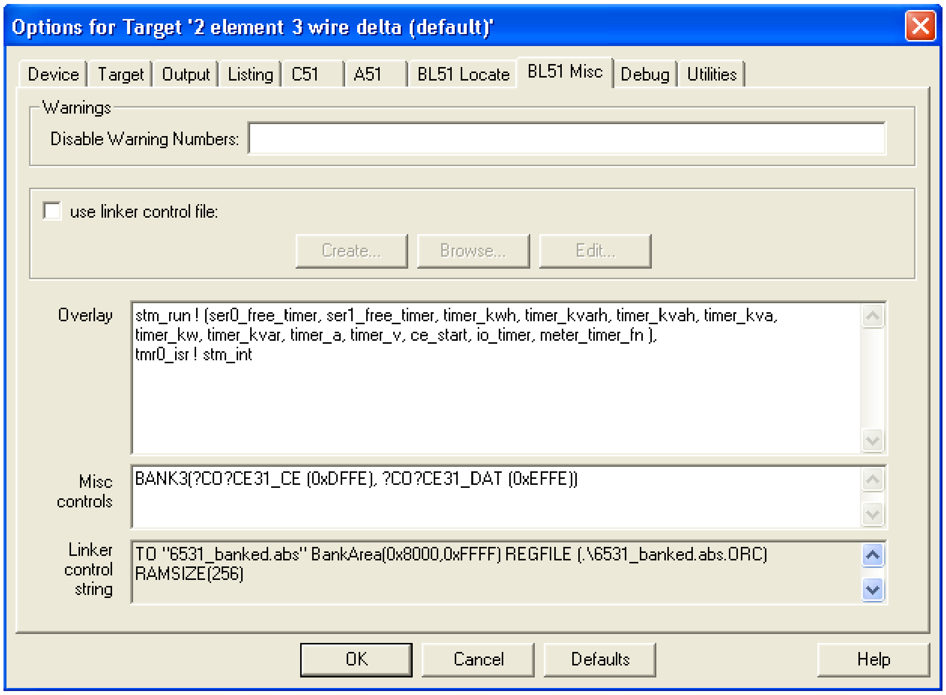
1. The subroutine is in a bank, and
2. The subroutine is called from outside its bank.

Most problems with banking code occur because the linker omits a function from the page table. The result is that the call to a function in a different bank goes to code in the current bank, causing unexpected code in the current bank to be executed.

One major cause of this is a callback subroutine called via a function pointer. Another is an interrupt defined in banked assembly language file (fortunately, Keil detects and flags banked interrupts in C code).

To solve problems stemming from callback routines, all subroutines called from other banks should be made global, so that the linker can use their data.

Next, overlay commands should be used to inform the Keil linker that a banked function is called from a caller in a different bank. This forces the linker to put the callee function into the page table. To use the overlay command in the linker, see the discussion of "overlay" in the Keil linker's documentation. Here is an example of the overlay commands from the Demo Code. These commands map the callback routines that are called from the software timer, and hardware timer interrupt 0.



Warnings

Disable Warning Numbers:

☐ use linker control file:

Overlay

```
stm_run ! (ser0_free_timer, ser1_free_timer, timer_kwh, timer_kvarh, timer_kvah, timer_kva,
timer_kw, timer_kvar, timer_a, timer_v, ce_start, io_timer, meter_timer_fn ),
tmr0_isr ! stm_int
```

Misc controls

```
BANK3(?CO?CE31_CE (0xDFFE), ?CO?CE31_DAT (0xEFFE))
```

Linker control string

```
TO "6531_banked.abs" BankArea(0x8000,0xFFFF) REGFILE (.\6531_banked.abs.ORC)
RAMSIZE(256)
```

**Figure 3: Linker Dialog Box**

However, if there should be other problems, there is a way to isolate them:

1. Remove code from the project until all code fits in common and bank 1.
2. Move modules individually each to a bank until the problem occurs.
3. At some point, the problem is likely to show up as an unexpected reset. What is happening here is that the call to code in bank 1 is probably going to uninitialized code memory in another bank. It will execute until the program counter wraps around to zero and begins executing the reset vector.
4. Place a break point near the end of the other bank, to catch the erroneous execution.
5. After trapping the error, set the program counter to the address of a RET instruction, and single-step. The code will return to the code that called the wrong bank.
6. Fix the calling routine, and all similar problems!

## Generating a Hex File

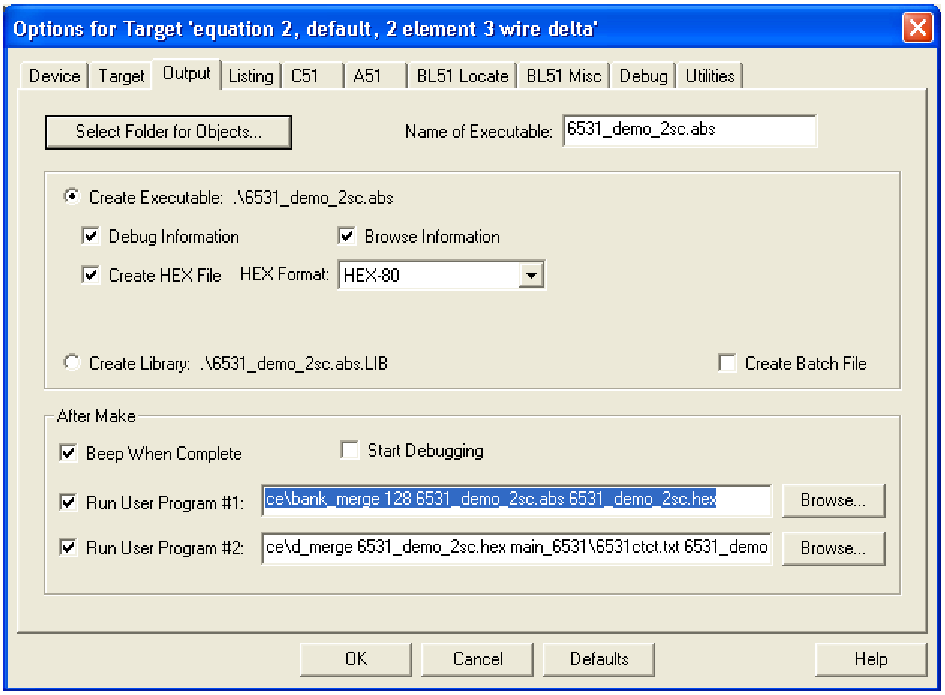
The BL51 linker and its associated hex converter produce a separate Intel hex file for each bank. These files end with names such as .H01 for bank 1, .H02 for bank 2, etc. The emulator and the production programmer expect a single Intel 386 hex file with a .HEX ending.

TERIDIAN's demo CD ROM has a utility called bank_merge.exe. It runs in a DOS command window and merges the bank files from Keil's hex converter into one Intel-386 hex file. This can be placed in the build automatically.

Running bank_merge in the build uses a feature in Keil uVision3. Keil's uVision3 integrated development environment can run DOS command-line programs after it finishes building software. Right-click the project name in the project work-space window to view the project's options. click the output tab. Check the box, "Run user program #1", and fill the associated line with the bank merge command-line (see Figure 4). For example:

```
        ce\bank_merge 128 6531_demo_2sc.abs 6531_demo_2sc.hex
```

"128" is the size of the 6531's flash in 1-KB increments. A 6534 should have "256." The first file name "6531_demo_2sc.abs" tells which name of the .H0x files made by the Keil utilities. The file-type suffix '.abs' is ignored. The second name, "6531_demo_2sc.hex", is the output hex file's name.



Name of Executable: 6531_demo_2sc.abs

● Create Executable: .\6531_demo_2sc.abs

☑ Debug Information ☑ Browse Information

☑ Create HEX File HEX Format: HEX-80

○ Create Library: .\6531_demo_2sc.abs.LIB □ Create Batch File

☑ Beep When Complete □ Start Debugging

☑ Run User Program #1: ce\bank_merge 128 6531_demo_2sc.abs 6531_demo_2sc.hex

☑ Run User Program #2: ce\d_merge 6531_demo_2sc.hex main_6531\6531ctct.txt 6531_demo

**Figure 4: Output Options Dialog Box**

Another manual solution is to erase the flash, load the .abs file to an emulator, and then use the file->save range: All menu selection to save a copy of flash as an Intel hex file.

When using Keil's Lx51 advanced linker, the output dialog contains a pull-down list of hex formats. Select the "i386" hex file option.

The emulator's verification option is a convenient way to verify that the .abs and .hex file have the same content. To use it, invoke the menu File->Load, then check the verify box, but not the load box.

## Interrupts

The interrupts must start in the common code. TERIDIAN Demo Code starts most interrupts from a “trampoline” routine that saves and restores the bank register. The Demo Code’s trampoline routines are in Meter\io653x.c, with the stub interrupts and decoded interrupts. An example of a trampoline is shown below. It is a serial interrupt code from Meter\io653x.c, used to service UART0:

```
#pragma save
#pragma REGISTERBANK (ES0_BANK)
void es0_trampoline (void) small reentrant interrupt ES0_IV using ES0_BANK
{
    uint8_t my_fl_bank = FL_BANK; // save the bank register
    FL_BANK = BANK_DEFAULT; // BANK_DEFAULT is 1
    es0_isr();
    FL_BANK = my_fl_bank;   // restore the bank register
}
#pragma restore
```

This is clumsy and slow. Why do this?

A trampoline lets most interrupt code be in bank-switched code space with other code from the same file. This makes the code easier to read. It also leaves more space in common memory, and permits a larger system to exist. The penalty is a delay of a few tens of microseconds per interrupt.

Very frequent interrupts should not use trampoline routines. In TERIDIAN’s Demo Code, the CE code, Meter\ce.c has an interrupt that runs every 396 microseconds (ce_busy_isr()). ce.c is placed in the common area, and ce_busy’s trampoline is disabled (in Meter\io653x.c).

If an interrupt calls code in a bank, as shown above, it must save and restore the bank-switching register, *FL_BANK*.

Keil’s Lx51 advanced linker has an option to automatically save and restore the bank register in an interrupt. TERIDIAN’s L51_bank.a51 code provides the necessary symbols.

## Calls via Function Pointers

In a bank-switching system, functions placed in function pointers cannot have the word “static” in front of their definition; they must be global.

Additionally, the linker must be informed of the actual caller of the function in the function pointers. For example, in the TERIDIAN Demo Code, the software timer module (Util\stm.c) calls many routines. There is also an interrupt for hardware timer 1 that calls the software timer’s interrupt code. This linker dialogue tells the linker the true relationship. The command to the linker is “caller ! callee” or “caller ! (callee_1, callee_2, …)” (See the linker command Figure 3).

Why do this? If a function pointer points at a function’s entry in the page table, the function pointer can be executed from any bank, at any time. So, in theory, function pointers are supported via the page table.

In practice, the Keil linker tries to save space in the page table. To do this, it only puts global functions into the page table if they are in a bank area and are called from outside the bank. Often, function pointers are used for callback routines. In these cases, the linker often does not detect the true caller and so cannot detect the cross-bank function call. Then, it places a banked address into the code that sets the function pointer. If this is executed from another bank, the code jumps to code in the current, wrong bank!

Using overlay commands informs the linker of the actual caller, so it can detect cross-bank calls.

To increase reliability, the Demo Code that sets a function pointer also checks to make sure that the pointer is in common memory rather than a bank. A sample of the code is shown below:

```
    if (((uint16_t)fn_ptr) > 0x7FFF) // only accept functions in common
    {
        main_software_error (); // report a software error at a central breakpoint.
        return NULL;  // indicate a failure to the caller
    }
```

## Putting Constants in Banks

Space in the common code area can be precious in a large project. It often helps to put a few large tables in a different code bank. The TERIDIAN Demo Code, for example, places the help text, CE code and the CE's data initialization table into banked flash.

In order for this to work, every reference to the data must be from the same bank or from common.

In the Demo Code, the largest set of constant tables is the help text. The help text (CLI\Help.c) is in the same bank as the printing routines, which copy the text into RAM for use by the serial interrupts (see CLI\io.c). Since the help text is in the same bank as the accessing routines, no other special coding is needed.

In the TERIDIAN Demo Code, the CE code is referenced from the Meter\ce.c. ce.c could not be placed in the last bank with the tables because it also has a very fast interrupt, ce_busy_isr(), so ce.c was placed in common. Also, the CE code is in the last bank, so the code in ce.c had to explicitly switch it in to read it.

Designers must be careful that any code in common is smaller than the data table! Some systems may need a library routine in common to copy part of the banked data to RAM for use by banked code. The TERIDIAN Demo Code does not use this technique because fortunately, the help system fits in the same bank as the serial output routines.

Next, the data should be located in the desired bank, using compiler and linker's BANK commands. (See the compiler and linker command in Figure 2 and Figure 3).

The code accessing the banked data should include Util\bank.h, which defines `switchbank()` to access banked data.

In the code that accesses the data, the bank must be switched in. For example, when the Demo Code copies the starting data for the CE (ce_init() in Meter\ce.c), it executes the following code:

```
switchbank (BANK_CE);
memcpy_cer (
   (int32x_t *)CE_DATA_BASE,
   (int32r_t *)&CeData[0],
   (uint8_t)(0xff & NumCeData)
   );
```

`switchbank ()` sets the bank register without side effects for other bank switching. It is defined in Util\bank.h. `BANK_CE` is the bank number containing the CE initialization table (in Main_6531\options.h or Main_6534\Options.h).

`memcpy_cer () copies` 32-bit words from code to CE memory. `CE_DATA_BASE` is the start of CE memory, 0x0000 in XDATA in 71M653x ICs. `CeData[]` is the array of 32-bit integers containing the CE's default data. `NumCeData` is the count of data words in the table, a constant value that precedes the CE default data table.

## Write-Protecting Flash Memory

In addition to safety interlocks in software that prevent accidental write operations to flash, the 71M653x ICs also have a write-protection mechanism implemented in hardware. Some systems might permit code or customization tables to be downloaded to flash memory, and designers might wish to assure that this process cannot corrupt other code or data.

To protect flash starting at address 0x00000, write the number of 1,024-byte blocks into *BOOT_SIZE* (at XDATA 0x20A7), and set *WRPROT_BT* (bit 5 of SFR 0xB2). Since this range covers the code's interrupt vectors, it is perfect for protecting a boot loader (i.e. code that can load other code into the system). It is also the logical choice for general-purpose write-protection.

To protect flash near the end of memory, place the CE's data area at that point (and set *CE_LCTN*, XDATA 0x20A8), and set the *WRPROT_CE* (bit 4 of SFR 0xB2). This protects not only the CE code, but also all flash memory after it. This is excellent for protecting the CE code and calibration tables stored in flash. The Demo Code uses this method to protect the CE code and its default initialization table.

## Revision History

| Revision | Date | Description |
|---|---|---|
| Rev. 1.0 | 10/16/2008 | First publication. |

*Maxim cannot assume responsibility for use of any circuitry other than circuitry entirely embodied in a Maxim product. No circuit patent licenses are implied. Maxim reserves the right to change the circuitry and specifications without notice at any time.*

***Maxim Integrated Products, 120 San Gabriel Drive, Sunnyvale, CA 94086 408-737-7600***