Application Report
SPNA091A–December 2005–Revised February 2006

Using Read Margin Modes in TMS470 F05 Flash Microcontrollers

Bob Crosby *AEC Automotive*

ABSTRACT

TMS470 microcontroller devices built using the F05 flash process have a built-in mechanism for detecting weak or degraded flash bits before they fail. A way of using this mechanism is discussed in this application report.

Contents

List of Figures

List of Tables

1 Background

TMS470 F05 flash cells are composed of transistors with an extra isolated gate: the floating gate. If this floating gate is neutrally charged, the transistor allows current flow when a preselected gate voltage is applied. This is the 1 state. If extra electrons are forced onto this floating gate, the field they create keeps the transistor turned off even with the gate voltage applied. This creates the 0 state. Imperfections in the flash cell may, over time, allow the electrons to escape from the floating gate, or allow extra electrons to be added to the floating gate, thus changing the state of the cell. Automotive safety-critical applications, such as airbag controllers and anti-lock brake systems, require a means of detecting these defective cells before they cause a system failure.

Traditional means of detecting failing nonvolatile memory locations, such as doing checksums and parallel signature analysis techniques, are inadequate alone because they require the memory location to completely fail before they are detectable. Small, almost imperceptible, differences in the timing paths between data reads and instruction fetches may allow checksum reads to pass even though instruction fetches from the same location will fail. A better mechanism for early detection of potential failing bits is required.

This application report explains a method of quickly checking the flash memory contents while minimizing the time that interrupts are disabled. A device with 512K bytes of flash in two banks, write protection keys at address 0x3FF0, and a maximum speed of 48 MHz is used in this example.

2 The Read Margin Modes

Control register bits in the F05 flash wrapper allow the user to select one of two modes for checking flash cells: read margin one mode or read margin zero mode. These modes effectively change the reference current used to determine if a bit is a 1 or a 0. Since the current through a flash cell is related to the voltage applied to the gate and the number of elecrons on the floating gate, it is easier to refer to the state of the cell by the gate voltage that would be needed to match the standard reference current. For example, in the normal read mode, if a cell passes less than the reference current when 5.0 V are applied to the gate, the cell is a 0. If the cell passes more than the reference current when 5.0 V are applied to the gate, the cell is a 1.

In read margin zero mode, the current ratio is changed such that it is equivalent to applying 5.2 V to the gate. This voltage checks that the programmed cells have at least 200 mV of margin before the bit might flip to a 1. Normal programmed cells will have a threshold voltage (Vt) of 6.5 V or more.

In read margin one mode, the current ratio is changed such that it is equivalent to applying 4.8 V to the gate. This voltage checks that the erased cells have at least 200 mV of margin before the bit might flip to a 0. Normal erased cells will have a Vt of 4.5 V or less.

Programmed cells have an excess of electrons. The negatively charged floating gate will leak electrons to achieve a neutral state if there are any leakage paths. This leakage is called data retention loss (DRL). Erased cells, on the other hand, have a neutrally charged floating gate. They are less likely to gain or lose electrons. However, some defects allow the floating gate to gain electrons while the read voltage is applied to the gate. Therefore, both read margin zero and read margin one tests should be run to guarantee properly functioning flash. If it is only possible to run one test, the read margin zero test is the more critical of the two, since it catches the more common failure mechanism.

Properly functioning flash cells will read properly in either read margin mode for the full lifetime guaranteed by the device specification.

3 Using the Read Margin Modes

There are several steps to using the read margin modes:

- Initialize the flash.
- Copy the routine to perform the read margin check into RAM.
- Execute the routine.

3.1 Initializing the Flash

Programmed bits (0s) will start to fail at low speed. Erased bits (1s) will start to fail at high speed. If enough execution time is available, it is best to do a read margin zero check at slow speed, or with extra wait states, and a read margin one check at high speed. If there is enough time available, the routine running from RAM can change the wait states from one to fifteen while the read margin zero check is being done. Often, for in-system checks, not much time is available. Then it is acceptable to do both checks, or only the read margin zero check, at the normal operating speed. In this case, the normal flash initialization is used. It consists of the following steps:

- Enter configuration mode.
- Set the flash wait states.
- Enable pipeline mode.
- Match the flash keys.
- Leave configuration mode.
- Change the PLL to increase the clock speed.

3.1.1 Example of Initialization Routine

Here is an example of the beginning of the initialization code. The complete source code for this example can be found in Appendix A. The register addresses and macros used in this example are defined in the header file *misc.h*.

```
#include "misc.h"
const unsigned int key[]={0xffffffff,0xffffffff,0xffffffff,0xffffffff};
void c_int00()
{  unsigned int i;
    GLBLCNTL = 0x0017;                          // SYSCLK = 12MHz (12MHz in)
    FMMAC2 = 0xFFF8 + 0;                        // Select bank 0
    FMBAC2 = 0x7F11;                            // 1 wait state
    FMMAC2 = 0xFFF8 + 1;                        // Select bank 1
    FMBAC2 = 0x7F11;                            // 1 wait state
    FMREGOPT = 1;                               // ENABLE PIPELINE MODE
    for(i=0;i<4;i++)                            // Match the keys
    {  KEY_LOCATION[i];
       FMPKEY=key[i];
    }
    GLBLCNTL = 0x0001;                          // SYSCLK = 48MHz (12MHz in)
    SYSPCR = 0x07;                              // ICLK = 16MHz, enable peripherals
    ...
```

Matching the flash keys is only required if the key locations are part of the flash memory that will be checked for margin. Matching the keys is done here in the startup code because it must be done in configuration mode.

3.2 Example of Main Code

The main program is responsible for copying the check routine to the RAM and executing this routine. In this example, the parallel signature analysis (PSA) calculation is broken into blocks to allow interrupts to be serviced between blocks. The BLOCK_SIZE parameter determines how long interrupts are disabled. Also, the block size must be chosen such that a single call to the function *ram_psa()* does not cross a bank boundary.

```
#include "misc.h"
#define BLOCK_SIZE 0x800             //Number of words to check
static load(unsigned int *load,unsigned int *start, unsigned int size);
extern unsigned int Load_RAM_PSA, Run_RAM_PSA;
extern unsigned int Size_RAM_PSA,psa_value;
main()
{ unsigned int size,count,bank;
  volatile unsigned int *address;
  load(&Load_RAM_PSA,&Run_RAM_PSA,(unsigned int)&Size_RAM_PSA);
  // Read Margin 0
  ENABLE_PSA;              //Initialize PSA value to zero
  PSA=0;
  DISABLE_PSA;
  address=0;               //start check at address zero
  count=(FLASH_SIZE-4)>>2;  //0x80000 bytes of flash (less 32-bit PSA at end)
  do
  { size=(count>BLOCK_SIZE) ? BLOCK_SIZE: count;
    bank=((unsigned int)address<BANK1_START) ? 0 : 1;
    address=ram_psa(address,size,bank,MARGIN0);       // execute this code in RAM
  } while (count-=size);
  if (PSA!=psa_value)
  { //  Read margin 0 error code goes here
  }
  // Read Margin 1
  ENABLE_PSA;               //Initialize PSA value to zero
  PSA=0;
  DISABLE_PSA;
  address=0;                //start check at address zero
  count=(FLASH_SIZE-4)>>2;   //0x80000 bytes of flash (less 32-bit PSA at end)
  do
```

```
  { size=(count>BLOCK_SIZE) ? BLOCK_SIZE: count;
    bank=((unsigned int)address<BANK1_START) ? 0 : 1;
    address=ram_psa(address,size,bank,MARGIN1);       // execute this code in RAM
  } while (count-=size);
  if (PSA!=psa_value)
  { //  Read margin 1 error code goes here
  }
  // Rest of program goes here

}
static load(unsigned int *load,unsigned int *start, unsigned int size)
{ size=(size+3)>>2; //convert from bytes to words and round up
  do
  { *start++=*load++;
  } while (--size);
}
```

3.3 Routine Run from RAM

The code that actually calculates the PSA exists in a separate file. This separate file makes it easier to identify this routine to the linker for loading into RAM. This is the routine that is copied into RAM in main() and then executed. In this routine, interrupts are disabled by software interrupt (SWI) routines. Two special test registers are written to in configuration mode. Normally, you cannot use configuration mode at frequencies above 24 MHz. Since, in this case, flash is neither read nor written while in configuration mode, you can execute the routine from RAM and write to the flash control registers in configuration mode at the full speed of the part.

```
#include "misc.h"
volatile unsigned int *ram_psa(volatile unsigned int *address,
                               unsigned int length,
                               unsigned int bank,
                               MODE mode)
{
  INT_DISABLE();           // SWI routine
  GLBLCNTL = 0x0011;       // enter config mode
  FMMAC2 = 0xFFF8 + bank;  // Select the bank
  FMTCR=0x2FC0+mode;       // Enter read margin test mode
  FMDTR=0xB;               // TEZ low (active)
  GLBLCNTL = 0x0001;       // leave config mode
  ENABLE_PSA;
  do
  {  *address++;
  } while (--length!=0);
  DISABLE_PSA;
  GLBLCNTL = 0x0011;       // enter config mode
  FMDTR=0xF;               // TEZ high (inactive)
  FMTCR=0x03C0;            // Leave read margin test mode
  GLBLCNTL = 0x0001;       // leave config mode
  INT_ENABLE();            // SWI routine
  return address;
}
```

4 Additional Flash Control Registers

Two flash test control registers are used in this example that are not documented in the *"TMS470R1x F05 Flash Reference Guide*, literature number SPNU213. These are the flash module test control register (FMTCR) and the flash module data test register. These two registers implemented as 16-bit registers. They can be written in word or half word accesses only. They are described below.

4.1 Flash Module Test Control Register (FMTCR)

Figure 1 and Table 1 describe this register.

Figure 1. Flash Module Test Control Register (FMTCR) [offset = 0x3C12h]

15	14	13	12	11	10	9	6	5	0
Reserved		Write Enable Code			Enable TCR	Reserved		Test Mode	
R/W-0		R/W-0			R/W-0	R/W-1		R/W-0	

R = Read; W = Write; -*n* = value after reset

Table 1. Flash Module Test Control Register (FMTCR) Field Descriptions

Bit	Field	Value	Description
15–14	Reserved	00	When these bits are written, they must be written as 0.
13–11	Write Enable Code	0–111	These bits must be written as 101.
10	Enable TCR		This bit must be written as a 1 to enable flash test modes.
9–6	Reserved	1111	When these bits are written, they must be written as 1.
5–0	Test Mode		These bits determine which test mode is entered if bit 10, Enable TCR, is set.
		000000	The flash is in normal read mode.
		010010	The flash is in read margin zero mode.
		010011	The flash is in read margin one mode.
		All other values	These values are reserved and should not be used.

4.2 Flash Module Data Path Test Register (FMDTR)

Figure 2 and Table 2 describe this register. There is a copy of this register for each bank in the flash module. The appropriate bank must be selected in the Module Access Control Register number 2 (FMMAC2) before writing to this register.

Figure 2. Flash Module Data Path Test Register (FMDTR) [offset = 0x0012h]

15	14	5	4	3	2	1	0
Reserved	Reserved		Reserved	Reserved	$\overline{TE}$	Reserved	Reserved
R-0	R-u		R/W-0	R/W-1	R/W-1	R/W-1	R/W-1

R = Read; W = Write; -*n* = value after reset; -*u* = undefinded

Table 2. Flash Module Datapath Test Register (FMDTR) Field Descriptions

Bit	Field	Description
15	Reserved	When this bit is written, it must be written as 0.
14-5	Reserved	Reads of these bits are undefined and writes have no effect.
4	Reserved	When this bit is written, it must be written as 0.
3	Reserved	When this bit is written, it must be written as 1.
2	$\overline{TE}$	Test enable. This bit is active low. When written as 0, the test mode defined in the TCR bits is applied. When written as 1, the test mode is disabled.
1	Reserved	When this bit is written, it must be written as 1.
0	Reserved	When this bit is written, it must be written as 1.

5 Impact on ROM Devices

TMS470R1x ROM devices built using the ROM pipeline wrapper (RPW) can execute the code in this example without error. ROM memory does not have the same failure mechanism as flash memory, and the read margin modes are not supported in the ROM devices. However, writing to the locations of the flash control registers will not cause a memory exception. On ROM devices, this routine will be a PSA check of the ROM memory, but no margin is applied to the ROM. This lack of margin is acceptable since ROM memory cells, unlike flash cells, are not subject to DRL.

Appendix A Source Code

A.1 misc.h

```
#define PSA         (*(volatile unsigned int *)0xFFFFFF40)
#define ENABLE_PSA  ((*(volatile unsigned int *)0xFFFFFF50)=0)
#define DISABLE_PSA ((*(volatile unsigned int *)0xFFFFFF50)=1)
#define SYSPCR      (*(volatile unsigned int *)0xFFFFFD30)
#define GLBLCNTL    (*(volatile unsigned int *)0xFFFFFFDC)
#define MFBAHR0     (*(volatile unsigned int *)0xFFFFFE00)
#define MFBALR0     (*(volatile unsigned int *)0xFFFFFE04)
#define MFBAHR2     (*(volatile unsigned int *)0xFFFFFE10)
#define MFBALR2     (*(volatile unsigned int *)0xFFFFFE14)
#define HETDIR      (*(volatile unsigned int *)0xFFF7FC34)
#define HETDOUT     (*(volatile unsigned int *)0xFFF7FC3C)
#define FLASH_BASE  0xFFE88000
#define FMREGOPT    (*(volatile unsigned int *)(FLASH_BASE+0x1C00))
#define FMPKEY      (*(volatile unsigned int *)(FLASH_BASE+0x1C0C))
#define FMDTR       (*(volatile unsigned int *)(FLASH_BASE+0x0010))
#define FMTCR       (*(volatile unsigned int *)(FLASH_BASE+0x3C10))
#define FMMAC2      (*(volatile unsigned int *)(FLASH_BASE+0x3C04))
#define FMBAC2      (*(volatile unsigned int *)(FLASH_BASE+0x0004))
#define KEY_LOCATION ((volatile unsigned int *)0x3ff0)
#define BANK1_START 0x40000
#define FLASH_SIZE 0x80000
void INT_ENABLE();
void INT_DISABLE();
#pragma SWI_ALIAS(INT_ENABLE, 5)
#pragma SWI_ALIAS(INT_DISABLE,6)
typedef enum { NORMAL=0, MARGIN0=18, MARGIN1=19} MODE;
volatile unsigned int *ram_psa(volatile unsigned int *address,
                               unsigned int length,
                               unsigned int bank,
                               MODE mode);
```

A.2 intvecs.asm

```
        .state32
        .global _c_int00
        .global _ISR_SWI
              .global _psa_value
              .global PSA

        .sect   ".intvecs"
        b   _c_int00            ;  RESET INTERRUPT
        b   #-8                 ;  UNDEFINED INSTRUCTION INTERRUPT
        b   _ISR_SWI            ;  SOFTWARE INTERRUPT
        b   #-8                 ;  ABORT (PREFETCH) INTERRUPT
        b   #-8                 ;  ABORT (DATA) INTERRUPT
        b   #-8                 ;  RESERVED
        b   #-8                 ;  IRQ INTERRUPT
        b   #-8                 ;  FIQ INTERRUPT
             .sect ".psa_value"
_psa_value
             .word PSA
       .end
```

A.3 startup.c

```
#include "misc.h"
/*---------------------------------------------------------------------------*/
/* extern reference to cinit section                                         */
/*---------------------------------------------------------------------------*/
/* stack pointers (initial values)                                           */
```

```
asm("         .text");
asm("         .global _StackSUPER_");
asm("s_stack: .long   _StackSUPER_");
const unsigned int key[]={0xffffffff,0xffffffff,0xffffffff,0xffffffff};
/* the name c_int00 has a special meaning for the TMS470 compiler           */
/* DONT CHANGE IT!                                                          */
void c_int00()
{  unsigned int i;
    /* --- SYSTEM MODULE SETUP ------------------------------------------- */
    /* setup system registers SAR module                                    */
    GLBLCNTL = 0x0017;                          // SYSCLK = 12MHz (12MHz in)
    FMMAC2 = 0xFFF8 + 0;                        // Select bank 0
    FMBAC2 = 0x7F11;                            // 1 wait state
    FMMAC2 = 0xFFF8 + 1;                        // Select bank 1
    FMBAC2 = 0x7F11;                            // 1 wait state
    FMREGOPT = 1;                               // ENABLE PIPELINE MODE
    for(i=0;i<4;i++)                            // Match the keys
    {  KEY_LOCATION[i];
       FMPKEY=key[i];
    }
    GLBLCNTL = 0x0001;                          // SYSCLK = 48MHz (12MHz in)
    SYSPCR = 0x07;                              // ICLK = 16MHz, enable peripherals
    *(volatile unsigned int *)0x24;             // dummy read to flush data pipeline
    /* setup ROM/RAM chip selects                                           */
    /* nEMUCS2 internal RAM block0                                          */
    MFBAHR2 = 0x0050;                           // address: 0x00500000
    MFBALR2 = 0x0040;                           // size:    8k
    /* nEMUCS/nCS0 is the internal FLASH/ROM                                */
    MFBAHR0 = 0x0000;                           // address: 0x00000000
    MFBALR0 = 0x01A0;                           // size:    512K
    /* --- STACKS -------------------------------------------------------- */
    /* set supervisor stack (INITIAL MODE AFTER RESET)                      */
    asm(" ldr sp,s_stack");
    main();

}
```

A.4 Main.c

```
#include "misc.h"
#define BLOCK_SIZE 0x800            //Number of words to check
static load(unsigned int *load,unsigned int *start, unsigned int size);
extern unsigned int Load_RAM_PSA, Run_RAM_PSA;
extern unsigned int Size_RAM_PSA,psa_value;
main()
{ unsigned int size,count,bank;
  volatile unsigned int *address;
  load(&Load_RAM_PSA,&Run_RAM_PSA,(unsigned int)&Size_RAM_PSA);
  HETDIR=0x0000000F;
  HETDOUT=0x00000000;
  // Read Margin 0
  ENABLE_PSA;       //Initialize PSA value to zero
  PSA=0;
  DISABLE_PSA;
  address=0;                 //start check at address zero
  count=(FLASH_SIZE-4)>>2;  //0x80000 bytes of flash (less 32-bit PSA at end)
  do
  { size=(count>BLOCK_SIZE) ? BLOCK_SIZE: count;
    bank=((unsigned int)address<BANK1_START) ? 0 : 1;
    address=ram_psa(address,size,bank,MARGIN0);       // execute this code in RAM
  } while (count-=size);
  if (PSA!=psa_value)
    HETDOUT=1;
  // Read Margin 1
  ENABLE_PSA;       //Initialize PSA value to zero
  PSA=0;
```

```
  DISABLE_PSA;
  address=0;             //start check at address zero
  count=(FLASH_SIZE-4)>>2;  //0x80000 bytes of flash (less 32-bit PSA at end)
  do
  { size=(count>BLOCK_SIZE) ? BLOCK_SIZE: count;
    bank=((unsigned int)address<BANK1_START) ? 0 : 1;
    address=ram_psa(address,size,bank,MARGIN1);       // execute this code in RAM
  } while (count-=size);
  if (PSA!=psa_value)
    HETDOUT=2;
  for(;;);
}
static load(unsigned int *load,unsigned int *start, unsigned int size)
{ size=(size+3)>>2; //convert from bytes to words and round up
  do
  { *start++=*load++;
  } while (--size);
}
```

A.5 ram_psa.c

```
#include "misc.h"
volatile unsigned int *ram_psa(volatile unsigned int *address,
                               unsigned int length,
                               unsigned int bank,
                               MODE mode)
{
  INT_DISABLE();           // SWI routine
  GLBLCNTL = 0x0011;       // enter config mode
  FMMAC2 = 0xFFF8 + bank;  // Select the bank
  FMTCR=0x2FC0+mode;       // Enter read margin test mode
  FMDTR=0xB;               // TEZ low (active)
  GLBLCNTL = 0x0001;       // leave config mode
  ENABLE_PSA;
  do
  {  *address++;
  } while (--length!=0);
  DISABLE_PSA;
  GLBLCNTL = 0x0011;       // enter config mode
  FMDTR=0xF;               // TEZ high (inactive)
  FMTCR=0x03C0;            // Leave read margin test mode
  GLBLCNTL = 0x0001;       // leave config mode
  INT_ENABLE();            // SWI routine
  return address;
}
```

A.6 swi.c

```
/*--------------------------------------------------------------------------*/
/* NOTE: this must be compiled with -o2 in 32-bit mode and can't have more  */
/*            than three parameters                                         */
#pragma INTERRUPT(ISR_SWI, SWI)
void ISR_SWI(unsigned r0, unsigned r1, unsigned r2, unsigned r3)
{
     asm(" ldrb  r3, [lr, #-1]");
     switch (r3)
     {
     case 0:
          /* WRITE_REGISTER_SVC */
          *(unsigned int *)r0 = r1;
          return;
     case 1:
          /* enable FIQ interrupt */
          asm(" mrs   r0, spsr");
          asm(" bic   r0, r0, #0x40");
          asm(" msr   spsr, r0");
```

```
            return;
        case 2:
            /* disable FIQ interrupt */
            asm(" mrs   r0, spsr");
            asm(" orr   r0, r0, #0x40");
            asm(" msr   spsr, r0");
            return;
        case 3:
            /* enable IRQ interrupt */
            asm(" mrs   r0, spsr");
            asm(" bic   r0, r0, #0x80");
            asm(" msr   spsr, r0");
            return;
        case 4:
            /* disable IRQ interrupt */
            asm(" mrs   r0, spsr");
            asm(" orr   r0, r0, #0x80");
            asm(" msr   spsr, r0");
            return;
        case 5:
            /* enable both interrupts */
            asm(" mrs   r0, spsr");
            asm(" bic   r0, r0, #0xC0");
            asm(" msr   spsr, r0");
            return;
        case 6:
            /* disable both interrupts */
            asm(" mrs   r0, spsr");
            asm(" orr   r0, r0, #0xC0");
            asm(" msr   spsr, r0");
            return;
        }
}
```

A.7 link32.lcf

```
/*****************************************************************************/
/* LINKER COMMAND FILE                                                       */
/*                                                                           */
/*****************************************************************************/
/*****************************************************************************/
/* OPTIONS                                                                   */
/*****************************************************************************/
-c                      /* ROM AUTOINITIALIZATION MODEL                      */
-l rts32.lib
-x
-stack  0x800              /* SOFTWARE STACK SIZE                             */
/*****************************************************************************/
/* SPECIFY THE SYSTEM MEMORY MAP                                             */
/*****************************************************************************/
MEMORY
{
       ROM         (RX) :     origin=0x00000000   length=0x00080000
       RAM         (RW) :     origin=0x00500000   length=0x00002000

}
/*****************************************************************************************/
/*****************************************************************************************/
/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY                                           */
/*****************************************************************************************/
SECTIONS
{
    .intvecs : {} > 0     /* INTERRUPT VECTORS                */
    .text : {} > ROM
    .PSA: palign=4, run=RAM, load=ROM,
       LOAD_START(_Load_RAM_PSA),
```

```
          RUN_START(_Run_RAM_PSA),
          SIZE(_Size_RAM_PSA)
         { ram_psa.obj
            }
          .const: {} > ROM
      .psa_value : {} > 0x7fffc
      .bss : {} > RAM
      .stack : { .+=0x800;
                 _StackSUPER_=.;
                 } >RAM
}
PSA  =  0xE9577DEB;
```

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments
Post Office Box 655303 Dallas, Texas 75265