Application Report
SPNA079A–June 2005

# TMS470R1x Startup Sequence Description

*Trevor Jones and Vladimir Plotkin* *Automotive Group*

**ABSTRACT**

The default C compiler boot routine c_int00 provided by the runtime support is not sufficient for the TMS470R1X family of devices. This application report describes a c_int00 routine written in C that provides the minimum initialization required by the TMS470R1X based on F05 technology.

**Contents**



**List of Tables**



## 1 Introduction

The **c_int00** boot routine supplied with the C compiler routine library only sets up the user mode stack, initializes the C global variables and calls **main( )**. For a device, such as one of the TMS470R1X family of devices, that contain the system architecture reference (SAR) module, the default **c_int00** routine is insufficient. This application report describes a replacement **c_int00** routine written in C that provides the minimum initialization to correctly boot a TMS470R1x device.

The replacement c_int00 routine performs the following tasks:

- Initializes the SAR module (memory map and internal clock frequencies)
- Sets up the system module
- Sets up the address manager
- Sets up the memory controller
- Sets up internal flash (if required)
- Sets up all six stacks
- Initializes the C global variables
- Calls **main( )**
- Defines addresses in the linker command file

**CAUTION**

**When writing the boot code in C, it is important to remember that at the time c_int00 is called, NO STACK and NO RAM is available. c_int00 must be written with this limitation in mind. The version of c_int00 in this application report has been carefully written and does not use any STACK or RAM.**

If additional initialization code is to be added to c_int00, it is recommended that the code be placed in a separate function and that this function be called at the end of the routine just before the call to **main**, or just before the switch to user mode if the code needs to be executed in system mode. Also, bear in mind that because of the reasons listed above, the *Startup.c* file must be compiled using the **–o2** optimization option regardless of your project settings. If the programmer decides to use program-level optimization, this file still mustbe compiled using the**–o2** optimization option .

## 2 Description of Code

The following sections describe how the code performs the necessary adjustments.

### *2.1 Initializing the SAR Module*

The system architecture reference (SAR) module must be initialized in supervisor mode. The device is in supervisor mode immediately after reset, so a good practice is to perform this initialization at the start of the **c_int00** routine, before changing to user mode.

Three items must be initialized:

- The internal system clock SYSCLK: this is initialized in the SYSTEM MODULE.
- The device memory map: this is initialized in the ADDRESS MANAGER.
- The data bus width and wait states: this is initialized in the MEMORY CONTROLLER.

Detailed register descriptions can be found in the *TMS470R1x System Module Reference Guide* (SPNU189).

### *2.2 Setting Up System Module*

In the system module, two registers need to be initialized: the global control register (GLBCTRL) and the SYSCLK exception control register (SYSECR). CLKCNTL may also require initialization if the CLKOUT pin or the low power modes need to be changed from the default. For all other registers, the default values are sufficient. In the example, GLBCTRL, which controls the internal SYSCLK frequency; SYSECR, which enables the illegal peripheral / memory access interrupts; and CLKCNTL, to initialize the CLKOUT pin, are set.

The system module is accessed by defining a C structure **SARSYS_ST**, which defines the register layout and accesses it via a pointer reference **e_SARSYS_ST.**

```
typedef volatile struct
{
    unsigned int   : 16;
    unsigned short ClkCntl_UW;
    unsigned int   : 32;
    unsigned int   : 16;
    unsigned short CramCR_UW;
    unsigned int   : 16;
    unsigned short GlbCtrl_UW;
    unsigned int   : 16;
    unsigned short SysECR_UW;
    unsigned int   : 16;
    unsigned short SysESR_UW;
    unsigned int   : 16;
    unsigned short AbrtESR_UW;
```

```
    unsigned int   : 16;
    unsigned short GlbStat_UW;
    unsigned int   : 16;
    unsigned short Dev_UW;
    unsigned int   : 32;
    unsigned int   : 16;
    unsigned short Ssif_UW;
    unsigned int   : 16;
    unsigned short Ssir_UW;
} SARSYS_ST;
extern SARSYS_ST e_SARSYS_ST;
```

**Note:**

At this stage, only a **reference** is defined. The pointer itself will be defined later in the linker command file.

At this stage, the system clock must be initialized. The value that should be used in the example below depends on the actual system configuration. For more information on how to initialize these values for the zero-pin phase-locked loop (ZPLL), frequency-modulated phase-locked loop (FMPLL), or asynchronous phase-locked loop (APLL), please refer to *TMS470R1x System Module Reference Guide* (SPNU189).

This example uses ZPLL. The calculation is as follows:

**SYSCLK = (OSCIN * M) / PLL clock divider**

The **M** value is controlled by bit 3 in GLBCTRL, and it could be set to one of two values:

| *Bit 3 Value* | *M Value* |
|---|---|
| 0 | 8 |
| 1 | 4 |

The **PLL clock divider** should be defined in bits 0–2 of GLBCTRL and is specified as shown in Table 1:

**Table 1. PLLClock Divider Definitions**

| *ZPLL Clock Divider* | *Bit 2* | *Bit 1* | *Bit 0* |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 2 | 0 | 0 | 1 |
| 3 | 0 | 1 | 0 |
| 4 | 0 | 1 | 1 |
| 5 | 1 | 0 | 0 |
| 6 | 1 | 0 | 1 |
| 7 | 1 | 1 | 0 |
| 8 | 1 | 1 | 1 |

Thus, assuming an external oscillator at 10MHz and that the goal is a system clock at 20MHz, the following value can be written to GBLCTRL: 0x0011.

In this case, M equals 4 and the clock divider is 2.

10MHz * 4 / 2 = 20 MHz.

This results in the following lines:

```
e_SARSYS_ST.ClkCntl_UW = 0x0060;       /* set CLKOUT to SYSCLK    */
e_SARSYS_ST.GlbCtrl_UW = 0x0011;       /* set SYSCLK to 20MHz     */
e_SARSYS_ST.SysECR_UW  = 0x4007;      /* DISABLE abort interrupts */
```

Writing 0x0060 to CLKCNTL sets the CLKOUT pin to internal SYSCLK. Writing 0x0011 to GLBCTRL set the internal SYSCLK to the (output of the PLL * 4)/2 and writing 0x4007 to SYSECR disables the illegal access interrupts. However, it is not recommended to disable illegal access interrupts during debugging session as the interrupts will help the programmer to find errors.

**Note:**

Do not set SYSCLK to the highest possible value before initializing the flash pipeline. Remember that high frequencies might be only available when the flash is in pipeline mode. For example, if the device's datasheet says "20MHz System Clock (48MHz in Pipeline mode)," you must switch the flash module to pipeline mode first! If, in this example, you would like to configure the flash to work at 40MHz, you need to enable pipeline mode first, then use the following code:

```
e_SARSYS_ST.GlbCtrl_UW = 0x0001;       /* set SYSCLK to 40MHz     */
```

## 2.3 Setting Up Address Manager

Because the TMS470R1x family has many devices, it is impossible to list configurations for all of them in this paper. Therefore, one of these devices is used as an example: the TMS470AVF688A. The TMS470AVF688A features the following memory configuration:

- 256Kbytes of flash
- 12Kbytes of internal static RAM
- RAM 64-instructions (0.75Kbytes) HET RAM

The address manager is defined using a C structure **SARDEC_ST**, and is accessed via a pointer **e_SARDEC_ST**.

```
typedef volatile struct
{
    unsigned int   : 16;
    unsigned short mfbahR0_UW;
    unsigned int   : 16;
    unsigned short mfbalR0_UW;
    unsigned int   : 16;
    unsigned short mfbahR1_UW;
    unsigned int   : 16;
    unsigned short mfbalR1_UW;
    unsigned int   : 16;
    unsigned short mfbahR2_UW;
    unsigned int   : 16;
    unsigned short mfbalR2_UW;
    unsigned int   : 16;
    unsigned short mfbahR3_UW;
    unsigned int   : 16;
    unsigned short mfbalR3_UW;
    unsigned int   : 16;
    unsigned short mfbahR4_UW;
    unsigned int   : 16;
    unsigned short mfbalR4_UW;
    unsigned int   : 16;
    unsigned short mfbahR5_UW;
    unsigned int   : 16;
    unsigned short mfbalR5_UW;
    unsigned int   : 16;
    unsigned short mcbahR0_UW;
    unsigned int   : 16;
```

```
    unsigned short mcbalR0_UW;
    unsigned int   : 16;
    unsigned short mcbahR1_UW;
    unsigned int   : 16;
    unsigned short mcbalR1_UW;
    unsigned int   : 16;
    unsigned short mcbahR2_UW;
    unsigned int   : 16;
    unsigned short mcbalR2_UW;
    unsigned int   : 16;
    unsigned short mcbahR3_UW;
    unsigned int   : 16;
    unsigned short mcbalR3_UW;
    unsigned int   : 16;
    unsigned short mcbahR4_UW;
    unsigned int   : 16;
    unsigned short mcbalR4_UW;
    unsigned int   : 16;
    unsigned short mcbahR5_UW;
    unsigned int   : 16;
    unsigned short mcbalR5_UW;
} SARDEC_ST;
extern SARDEC_ST e_SARDEC_ST;
```

In this example, only the resister pairs MFBAxR0, MFBAxR2, MFBAxR3, and MFBAxR4 are of interest. MFBAHR0 and MFBALR0 define the memory map for the program memory (in this case, flash). MFBAHR2 and MFBALR2 map the first block of RAM. MFBAHR3 and MFBALR3 map the second block of RAM. MFBAHR4 and MFBALR4 map the HET RAM. Table 2 shows the definitions.

**Table 2. Memory Map for Flash and RAM**

| *Memory type* | *Start* | *End* | *Length* |
|---|---|---|---|
| FLASH | 0x00000000 | 0x0003FFFF | 0x40000 |
| RAM 0 | 0x00100000 | 0x001001FFF | 0x2000 |
| RAM 1 | 0x00102000 | 0x00103FFF | 0x1000 |
| HET RAM | 0x00800000 | 0x0080002FF | 0x300 |

To achieve the memory map given in Table 2, the registers should be programmed as follows (for more details on memory fine base address registers, refer to *TMS470R1x System Module Reference Guide* (SPNU189)):

```
e_SARDEC_ST.mfbahR0_UW = 0x0000;       /* address: 0x00000000      */
e_SARDEC_ST.mfbalR0_UW = 0x0090;       /* size:    256k            */
e_SARDEC_ST.mfbahR2_UW = 0x0010;       /* address: 0x00100000      */
e_SARDEC_ST.mfbalR2_UW = 0x0040;       /* size:    8k              */
e_SARDEC_ST.mfbahR3_UW = 0x0010;       /* address: 0x00102000      */
e_SARDEC_ST.mfbalR3_UW = 0x2030;       /* size:    4k              */
e_SARDEC_ST.mfbahR4_UW = 0x0080;       /* address: 0x00800000      */
e_SARDEC_ST.mfbalR4_UW = 0x0010;       /* size:    1k - can not    */
                                       /* specify less             */
```

**Note:**

Although the address manager contains the memory map select bit (MS in MFBALR0), which enables the memory map, it cannot be set at this time. The MS bit (bit 8) should only be set after both the address manager and memory controller have been initialized.

## 2.4 Setting Up Memory Controller

The memory controller defines the access mode and wait states for both the memory and the peripherals. This section discusses how to set up the memory map.

The following code defines a structure **SARMMC_ST** and pointer **e_SARMMC_ST** to access the memory controller registers:

```
typedef volatile struct
{
    unsigned int   : 16;
    unsigned short smcR0_UW;
    unsigned int   : 16;
    unsigned short smcR1_UW;
    unsigned int   : 16;
    unsigned short smcR2_UW;
    unsigned int   : 16;
    unsigned short smcR3_UW;
    unsigned int   : 16;
    unsigned short smcR4_UW;
    unsigned int   : 16;
    unsigned short smcR5_UW;
    unsigned int   : 16;
    unsigned short smcR6_UW;
    unsigned int   : 16;
    unsigned short smcR7_UW;
    unsigned int   : 16;
    unsigned short smcR8_UW;
    unsigned int   : 16;
    unsigned short smcR9_UW;
    unsigned int : 32;
    unsigned int : 16;
    union
    {
        unsigned short wcr0_UW;
        struct
        {
            unsigned int             : 14;
            unsigned int WTWSOvr_B1  : 1;
            unsigned int WBEnable_B1 : 1;
        } wcr0_ST;
    } wcr0_UN;
    unsigned int : 16;
    union
    {
        unsigned short pcr0_UW;
        struct
        {
            unsigned int            : 11;
            unsigned int ClkDiv_B4  : 4;
            unsigned int PEnable_B1 : 1;
        } pcr0_ST;
    } pcr0_UN;
    unsigned int   : 16;
    unsigned short plr_UW;
    unsigned int   : 16;
    unsigned short pprot_UW;
} SARMMC_ST;
extern SARMMC_ST e_SARMMC_ST;
```

SMCR0 controls the program memory access, SMCR1 controls the HET RAM access, and SMCR4 controls the external flash. SMCR0, SMCR1, and SMCR4 are initialized as follows:

```
e_SARMMC_ST.smcR0_UW = 0x0002;       /* 0 wait states, 32 bit    */
e_SARMMC_ST.smcR1_UW = 0x0072;       /* 7 wait states, 32 bit    */
```

Next, the write buffer is disabled by writing to WCR0.

```
e_SARMMC_ST.wcr0_UN.wcr0_UW = 0x0002; /* no trailing wait states  */
```

Next, peripheral access must be set up. PPROT enables user mode or supervisor mode only access to the peripherals; user mode access must be enabled for all the peripherals. PLR must be set to indicate that all the peripherals are on chip. The frequency of the peripheral clock (ICLK) is set by writing to PCR0.

```
e_SARMMC_ST.pprot_UW        = 0x0000;  /* user access on all       */
e_SARMMC_ST.plr_UW          = 0x0000;  /* all internal             */
e_SARMMC_ST.pcr0_UN.pcr0_UW = 0x0003;  /* ICLK=SYSCLK/2, enable    */
```

Finally, the chip selects must be enabled and the memory switched to the new memory map. These tasks are accomplished by setting the MS bit (0x0100) in the MFBALR0 register of the address manager.

```
e_SARDEC_ST.mfbalR0_UW     |= 0x0100;
```

## *2.5 Setting Up Flash Mode*

The flash on F05 devices could operate in pipeline mode, which allows the program execution to speed up. To enable it, access to flash wrapper registers set must be obtained. Then, which flash memory banks are active and which ones are inactive must be defined. The inactive banks could be put in low power mode to reduce power consumption. For more details, refer to *TMS470R1x F05 Flash Reference Guide* (SPNU213).

Finally, the pipeline mode and finish flash setup should be enabled or disabled; this is application-dependent.

Flash wrapper registers are defined in FWPROGRAM structure as follows:

```
struct flash
{
    unsigned         : 16;
    unsigned short FMBAC1;
    unsigned         : 16;
    unsigned short FMBAC2;
    unsigned         : 16;
    unsigned short FMBSEA;
    unsigned         : 16;
    unsigned short FMBSEB;
    unsigned         : 16;
    unsigned short FMBRDY;
    unsigned dummy1[1787];
    unsigned FMREGOPT;
    unsigned FMBBUSY;
    unsigned FMPKEY;
    unsigned dummy2[768];
    unsigned         : 16;
    unsigned short FMPTR4;
    unsigned         : 16;
    unsigned short FMPRDY;
    unsigned dummy3[1275];
    unsigned         : 16;
    unsigned short FMMAC1;
    unsigned         : 16;
    unsigned short FMMAC2;
    unsigned         : 16;
    unsigned short FMPAGP;
    unsigned         : 16;
    unsigned short FMMSTAT;
    unsigned         : 16;
    unsigned short FMTCR;
};
#define FWPROGRAM ((volatile struct flash *)(0xFFE88000))
```

If flash memory is left uninitialized, it will be running with maximum wait states, which can significantly slow down application execution. First, access to flash wrapper registers needs to be enabled:

```
e_SARSYS_ST.GlbCtrl_UW |= 0x0010;
```

Then, voltage parameters need to be configured (set VREAD to 5V):

```
FWPROGRAM->FMTCR  = 0x2FC0;
FWPROGRAM->FMPTR4 = 0xA000;
FWPROGRAM->FMTCR  = 0x03C0;
```

The device has two flash memory banks; they are initialized here. The FLASHWS value is defined as 0 for non-pipelined mode and 0x11 for pipelined mode.

```
FWPROGRAM->FMMAC2 = 0xFFF8;              /* bank0                   */
FWPROGRAM->FMBAC1 = 0x00FF;                      /* active                  */
FWPROGRAM->FMBAC2 = 0x7F00 | FLASHWS;    /* wait states             */
FWPROGRAM->FMMAC2 = 0xFFF9;                      /* bank1                   */
FWPROGRAM->FMBAC1 = 0x00FF;                         /*     active              */
FWPROGRAM->FMBAC2 = 0x7F00 | FLASHWS;       /*     wait states         */
```

Now, pipeline mode is enabled or disabled, depending on the application. The value in FLASHPM should be set to 0 for normal mode and to 1 for pipeline mode.

```
FWPROGRAM->FMREGOPT = FLASHPM & 1;
```

Finally, flash wrapper access needs to be disabled:

```
e_SARSYS_ST.GlbCtrl_UW &= ~0x0010;
```

## 2.6 Setting Up the Stacks

The TMS470R1X devices have six modes of operation: USER, FIQ, IRQ, ABORT, UNDEFINED INSTRUCTION, and SUPERVISOR mode. Each mode has a separate stack pointer and all of these stack pointers need to be initialized. The mode cannot be changed, nor can you write to the stack pointers in C, so this part of the **c_int00** routine is written in inline assembly code. Also, 32-bit values cannot be directly loaded into the stack pointers, so a table of constants is set up that defines the stack start address. This constant table is before the **c_int00** function definition:

```
asm("         .text");
asm("         .global _StackUSER_");
asm("u_stack: .long   _StackUSER_");
asm("         .global _StackFIQ_");
asm("f_stack: .long   _StackFIQ_");
asm("         .global _StackIRQ_");
asm("i_stack: .long   _StackIRQ_");
asm("         .global _StackABORT_");
asm("a_stack: .long   _StackABORT_");
asm("         .global _StackUND_");
asm("d_stack: .long   _StackUND_");
asm("         .global _StackSUPER_");
asm("s_stack: .long   _StackSUPER_");
```

The stack pointer values _StackUSER_, _StackFIQ_, _StackIRQ_, _StackABORT_, _StackUND_ and _StackSUPER_ refer to the start addresses of the stacks for USER, FIQ, IRQ, ABORT, UNDEFINED INSTRUCTION, and SUPERVISOR modes, respectively. These pointers are defined in the linker command file. After reset, the device is in supervisor mode, so the supervisor mode stack pointer is loaded:

```
asm(" ldr sp,s_stack");
```

Next, the mode is changed to FIQ mode and the FIQ stack is loaded:

```
asm(" mrs r5,cpsr");              /* FIQ stack                   */
asm(" bic r5,r5,#0x0E");
asm(" msr cpsr,r5");
asm(" ldr sp,f_stack");
```

Similarly, the other interrupt/exception stack pointer is set up:

```
asm(" add r5,r5,#1");              /* IRQ stack                    */
asm(" msr cpsr,r5");
asm(" ldr sp,i_stack");
asm(" add r5,r5,#5");              /* abort stack                  */
asm(" msr cpsr,r5");
asm(" ldr sp,a_stack");
asm(" add r5,r5,#4");              /* undefined instruction stack  */
asm(" msr cpsr,r5");
asm(" ldr sp,d_stack");
```

Finally, the user mode stack is set up by changing to SYSTEM mode. System mode uses the same stack as user mode, but it remains in privilege mode of operation. This allows the user stack to be set up, but the rest of **c_int00** can be executed in privilege mode still:

```
asm(" add r5,r5,#4");              /* user system stack          */
asm(" msr cpsr,r5");
asm(" ldr sp,u_stack");
```

## 2.7 Initalizing the C Global Variables

The C compiler generates a table in the **.cinit** section that is used to initialize the pre-initialized global variables. Symbol **___cinit__** is generated by the linker to access the **.cinit** section from C. A reference is generated to it:

```
extern unsigned *__cinit__;
```

The table is fairly simple; it contains values to initialize each variable, the length in bytes of the start address, and the data itself. The end of the table is indicated by a zero length. If the table is completely empty, the first length will be set to -1.

```
if ((unsigned *)&__cinit__ != (unsigned *)0xFFFFFFFF)
{
    unsigned *table  = (unsigned *)&__cinit__;
    unsigned  length = *table++;

    while (length != 0)
    {
        unsigned char *address = (unsigned char *)*table++;
        unsigned char *data    = (unsigned char *)table;
        while (length > 0)
        {
            *address++ = *data++; length--;
        }
        /* realign cinit pointer to point to next entry */
        table  = (unsigned *)(((unsigned)data + 3) & ~3);
        length = *table++;
    }
}
```

## 2.8 Calling Main( )

Finally, user mode needs to be invoked and the application **main( )** routine needs to be called. This is not absolutely necessary because many applications are running in privileged mode. However, switching to user mode will provide your application with additional protection as some of the registers (usually the critical ones) cannot be changed in user mode.

```
asm(" mrs r5,cpsr");               /* user mode                    */
asm(" bic r5,r5,#0x0F");
asm(" msr cpsr,r5");
main();
exit();
```

## 2.9 Defining Addresses in Linker Command File

In the MEMORY section of the linker command file, the address for the memory map, the HET RAM and the SAR module addresses needs to be generated. In this example, the only peripheral with onboard RAM is defined as HET. But if the device you are using has other peripherals with RAM, such as the CAN controller, you will need to define these as well.

**CAUTION**

**The programmer must take care not to overwrite the flash protection keys (see *TMS470R1x F05 Flash Reference Guide,* literature number SPNU213), as overwriting those keys may render the device unusable.**

These keys reside at the end of the first sector and occupy 4 words. In the case of TMS470AVF688A, these keys are placed at address 0x1FF0 as the first sector occupies 8Kbytes.

```
VECTORS  (X)    :  origin=0x00000000   length=0x20
 ROM1     (RX)   :  origin=0x00000020   length=0x1FD0
 ROM_KEYS (RX)   :  origin=0x00001FF0   length=0x00000010
 ROM2     (RX)   :  origin=0x00002000   length=0x0003Dfe0
 STACKS   (RW)   :  origin=0x00100000   length=0x00001000
 RAM      (RW)   :  origin=0x00101000   length=0x00002000
 /* peripherals */
 HETPROG (RW)   :  origin=0x00800000   length=0x00000400

 /* sar module addresses */
 MMC     (RW)   :  origin=0xFFFFFD00   length=0x40
 DEC     (RW)   :  origin=0xFFFFFE00   length=0x60
 SYS     (RW)   :  origin=0xFFFFFFD0   length=0x30
```

In the SECTIONS part of the linker command, all the address pointers required by the boot code need to be defined.

First, the stack addresses are defined by splitting up the default **.stack** into several areas and assigning each area to one of the stack address pointers (e.g. _StackUSER_, … ).

```
.stack   : {
                _StackUSER_  = .+ (0x1000 - (4+128+4+4+128));
                _StackFIQ_   = _StackUSER_  + 4;
                _StackIRQ_   = _StackFIQ_   + 128;
                _StackABORT_ = _StackIRQ_   + 4;
                _StackUND_   = _StackABORT_ + 4;
                _StackSUPER_ = _StackUND_   + 128;
           } > STACKS
```

Then the code needs to generate to pointers to the SAR modules:

```
.MMC     : {_e_SARMMC_ST = .;}  > MMC
.DEC     : {_e_SARDEC_ST = .;}  > DEC
.SYS     : {_e_SARSYS_ST = .;}  > SYS
```

## 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