Application Report
SPRABY4–March 2015

Accessing External SDRAM on the TMS320F2837x/2807x Microcontrollers Using C/C++

David M. Alter, Cody Addison, Vishal Coelho

ABSTRACT

The TMS320F2837x and TMS320F2807x microcontrollers provide facilities for interfacing to external SDRAM memory through two external memory interface modules. The address space for this memory lies beyond the 22-bit addressable range traditionally accessed with the C28x CPU. This application report describes how to access the SDRAM memory from C/C++ code.

Contents

1 Introduction

The External Memory Interfaces (EMIF) on the TMS320F2837x and TMS320F2807x families of microcontrollers have the SDRAM address range in far memory, here defined as being above 22 bits address [(1)]. This is beyond the 22-bit range that the C/C++ compiler historically supported, and also incurs some limitations due to the C28x CPU architecture itself. The C28x core has a 22-bit program address bus, a 16-bit stack pointer (SP), and a 16-bit Data Page Pointer (DP). This limits available addressing methods for far memory. Specifically, the following restrictions must be observed:

- TMS320C2000™ C/C++ Compiler version 6.4.0 or later is required
- Only global and static local data can exist in far memory. Code is restricted to the lower 22 bits of memory (henceforth termed near memory), while the stack (for automatic variables, also called non-static local variables) is restricted to the lower 16 bits of memory.
- The direct addressing mode for instructions requires use of the 16-bit DP register (plus 6 least-significant bits from the instruction op-code itself). The combined address is limited to 22-bit range. As a result, only indirect addressing (using the 32-bit wide XARn registers) can be used to access data in far memory. The compiler will take care of using the correct addressing modes.
- Instructions that use the program bus cannot access far memory with this bus. These instructions are: MAC, DMAC, XMAC, XMACD, IMACL, QMACL PREAD, PWRITE, XPREAD, and XPWRITE.

(1) Be careful not to confuse the present usage of the term ‘far’ with earlier compiler usage. Historically, the TMS320C2000 compilers had support for small and large memory models, with small model being defined as 16 bits address range and large model defined as 22 bits address range. With small memory model, there was the ‘far’ keyword which declared a data object as being above 16 bits. This usage of ‘far’ is now deprecated, and small memory model is no longer supported.

2 Compiler Enhancement and Basic Usage

2.1 The Far Data Object Attribute

Starting with the TMS320C2000 C/C++ Compiler v6.4.0, far data objects can be accessed from C/C++ code by declaring the data object with the GCC-style attribute *__attribute__((far))*. This attribute will cause the compiler to use the full 32-bit address of the data object to access it. The far attribute can be applied to any global or static local data object.

To prevent the compiler from employing instructions that uses the program bus to access the data object, the *volatile* keyword must also be applied in addition to the far attribute. The volatile keyword was traditionally used to tell the compiler that the data object may change outside the scope of the compiler, but it also already had the property in the TMS320C2000 C/C++ compiler that program bus instructions were not utilized to access such data objects. Therefore, it was convenient to exploit this property and press volatile into service for far data objects as well.

Here is a fundamental C-code example, first without using the far attribute:

```
int x;                                  // x is in near memory
int *p;                                 // p is in near memory

void foo() {
   p = &x;
}
```

which compiles to:

```
_foo:
        MOVL      XAR4,#_x
        MOVW      DP,#_p
        MOVL      @_p,XAR4
        LRETR
```

Above, the “MOVL XAR4, #x" will only load the 22 least-significant bits of the address of x into the XAR4 register. Hence, x must be linked to near memory for the code to operate correctly.

Here is the same C-code example with x located in far memory:

```
__attribute__((far)) volatile int x;      // x is in far memory
volatile int *p;                          // p is in near memory

void foo() {
   p = &x;
}
```

which compiles to:

```
foo:
        MOV       AL,#_x+0
        MOV       AH,#_x+0 >> 16
        MOVW      DP,#_p
        MOVL      @_p,ACC
        LRETR
```

Above, the full 32 bits of the address of *x* is loaded into the accumulator register, which in turn is assigned to pointer *p*. Hence, *x* can be linked to far memory and the code will work correctly.

Note in the C-code above that both *x* and *p* were declared as volatile (or more correctly for *p*, it was declared as pointing to a volatile object). If the volatile keyword is not applied to a symbol declared with the far attribute, the compiler will apply it automatically and issue a warning to notify the user. A warning will also be issued if a non-volatile pointer is assigned the address of a volatile data object, although in this case volatile is not automatically applied by the compiler. Also note that the declaration of *p* does not make any mention of far. This is because the far attribute applies to a data object itself, and not to pointers that reference it (although one could declare a pointer as being far if the pointer itself is going to live in far memory).

2.2 Linking Requirements

Data objects declared with the far attribute are placed by the compiler into two special sections. The *.farbss* section contains far variables, while the *.farconst* section contains far constant data (those declared with the *const* keyword). The two sections can then be placed in far memory as desired using the linker command file. For the .farbss section, this linking is straightforward. In the following example, a 512K×16 SDRAM is located connected to EMIF1 chip select zone 0 (EMIF1_CS0n). This gives it a starting address of 0x80000000.

```
/*************************************************************************
* MyFile.cmd - linker command file
*************************************************************************/

MEMORY
{
PAGE 1:    /* Data Memory */
   SDRAM    : origin = 0x80000000, length = 0x00080000   /* 512Kx16 SDRAM */
}


SECTIONS
{
   .farbss:     > SDRAM,    PAGE = 1
}
```

However, in the case of .farconst, separate load and run addresses will be needed since constants must be loaded to persistent (non-volatile) memory such as flash. User code will need to copy the section from the load address (in flash) to the run address (in SDRAM) at runtime as part of the software initialization procedure. While the code generation tools provide facilities for implementing this [5], you are strongly discouraged from placing constants in far memory since there is no advantage in doing so. For more information on the impracticality of far constants on the TMS320F2837x/2807x devices, see tip # 3 in Section 5.

3 Far Attribute Examples

3.1 Correct Usage Examples

Example 1. Far Data, Near Pointer

```
__attribute__((far)) volatile int x;              // x is in far memory
volatile int* p;                                  // p is in near memory

void foo(void) {
    p = &x;                                       // p points to x
    *p = 0x1234;                                  // write to x
}
```

In the declaration of *p* above, the volatile indicates that p points to a volatile object. The pointer *p* itself is not volatile. Since *p* was not declared as far, there was no need to make *p* volatile.

Example 2. Far Data, Far Pointer

```
__attribute__((far)) volatile int x;                 // x is in far memory
volatile __attribute__((far)) int* volatile p;       // p is in far memory

void foo(void) {
    p = &x;                                          // p points to x
    *p = 0x1234;                                     // write to x
}
```

In the declaration of *p* above, the left-most volatile indicates that *p* points to a volatile object, while the right-most volatile indicates that *p* itself is volatile. Since *p* was declared with the far attribute, it must also be declared as volatile itself. Notice that for pointers, the right-most volatile must come after the data type, int*. It cannot be placed before the data type:

```
   volatile __attribute__((far)) volatile int* p;    // invalid
```

Also for pointers, the left-most volatile could be placed after the far attribute if desired:

```
   __attribute__((far)) volatile int* volatile p;    // valid, alternate format
```

However, in this application report, the convention that is followed places the volatile before the far attribute for pointers, and this is the suggested format.

```
   volatile __attribute__((far)) int* volatile p;    // valid, suggested format
```

Example 3. Near Data, Far Pointer

```
int x;                                                  // x is in near memory
__attribute__((far)) int* volatile p;                   // p is in far memory

void foo(void) {
    p = &x;                                             // p points to x
    *p = 0x1234;                                        // write to x
}
```

Example 4. Volatile Near Data, Far Pointer

```
volatile int x;                                         // x is in near memory
volatile __attribute__((far)) int* volatile p;          // p is in far memory

void foo(void) {
    p = &x;                                             // p points to x
    *p = 0x1234;                                        // write to x
}
```

Example 5. Far Local Static Objects

```
void foo(void) {
    static volatile __attribute__((far)) int x;         // x is in far memory
    static volatile __attribute__((far)) int* volatile p;   // p is in far memory
    p = &x;                                             // p points to x
    *p = 0x1234;                                        // write to x
}
```

Example 6. Far Constants

```
const __attribute__((far)) volatile int c[4]={0,1,2,3};         // c is in far memory
const static __attribute__((far)) volatile int d[3]={4,5,6};    // d is in far memory

void foo(void) {
  const static __attribute__((far)) volatile int e[2]={7,8};    // e is in far memory
  ...
}
```

By regular ANSI-C conventions in the example above, *c* has global scope, *d* has local scope within the source file, and *e* has local scope within the *foo()* function. For information on the impracticality of far constants on the TMS320F2837x/2807x devices, see tip # 3 in Section 5.

Example 7. Structure Declarations

Case 1: The structure lives in near memory, p and q point to near memory.

```
typedef struct {
   unsigned int a;
   unsigned long b[100];
   unsigned int *p;
   long *q;
} MyStruct_t;

MyStruct_t x;
```

Case 2: The structure lives in near memory, p points to far memory, q points to near memory.

```
typedef struct {
   Uint16 a;
   Uint32 b[100];
   Uint16 *p;
   int32 *q;
} MyStruct_t;

__attribute__((far)) volatile MyStruct_t x;
```

Case 3: The structure lives in far memory, p and q point to near memory.

```
typedef struct {
   unsigned int a;
   unsigned long b[100];
   unsigned int *p;
   long *q;
} MyStruct_t;

__attribute__((far)) volatile MyStruct_t x;
```

Case 4: The structure lives in far memory, p points to far memory, q points to near memory.

```
typedef struct {
   unsigned int a;
   unsigned long b[100];
   volatile unsigned int *p;
   long *q;
} MyStruct_t;

__attribute__((far)) volatile MyStruct_t x;
```

With structures, all structure members are far (or near) if the structure itself is far (or near). Basically, the members are the structure, and live wherever the structure lives. Pointer members can be declared as pointing to near or far objects independent of where the structure (and, therefore, the pointer member) itself lives. Remember, for pointers there are two near and far properties: where the pointer itself lives and where the object being pointed to lives. These properties are independent and apply to all pointers whether they are members of a structure or separately declared pointers.

Example 8: Function Parameter Passing

```
int foo(int, volatile int*);                    // function prototype
__attribute__((far)) volatile int x;            // x is in far memory
__attribute__((far)) volatile int y[10];        // y is in far memory

int main(void) {
    int z;                                      // z is local (on stack)
    z = foo(x, y);                              // function call
}

int foo(int a, volatile int* p) {
    return(a + *p);
}
```

It is important to note that the passed parameters are not declared 'far'. The argument *a* is passed by value, while *p* points to a far data object. The parent function passes the entire 32-bit address of the array *y* (since it is a far object), which the child function receives in its entirety, in either a working register (XARn) or on the stack. Due to the way the hardware works, the child function will always manipulate all 32 bits of the address (and not just 22 bits), and also always access the contents of *y* through the 32-bit argument *p* in indirect addressing mode (and never use direct addressing). The use of *volatile* when declaring the argument *p* prevents the compiler from using the program bus when it is being de-referenced.

3.2 Incorrect Usage Examples

Example 9. Failing to Declare a Far Object as Volatile

```
__attribute__((far)) int x;                     // x is in far memory

void foo(void) {
     x = 5;
}
```

A compiler warning will be issued:

```
warning #2618-D: Variables that are far qualified must be declared volatile to prevent using
instructions that require unified memory. Applying the volatile type qualifier to variable "x".
```

The warning is a reminder that far objects must be declared volatile to keep the compiler from using program bus access to such objects. The warning further states that the compiler is applying volatile automatically. It is best practice however to correct the coding mistake:

```
void foo(void) {
__attribute__((far)) volatile int x;            // x is in far memory
x = 5;
}
```

Example 10. Non-Static Local Objects Declared as Far

```
void foo(void) {
__attribute__((far)) volatile int x;          // x is in far memory
x = 5;
}
```

A compiler warning will be issued:

```
error #2130: attribute "__far" does not apply to automatic variables
```

Non-static local objects are always placed on the stack, and therefore they cannot be declared far independently. In addition, the stack is restricted to the lower 16-bit address range so anything on the stack will always be in near memory. The correct code is:

```
void foo(void) {
int x;                                        // x is local (on stack)
x = 5;
}
```

Example 11. Volatile Object Address Assigned to a Non-Volatile Pointer

```
__attribute__((far)) volatile int x;          // x is in far memory
__attribute__((far)) int* volatile p;         // p is in far memory
void foo(void) {
p =*"                                         // p points to x
}
```

A compiler warning will be issued:

```
warning #515-D: a value of type "volatile int *“ cannot be assigned to an entity of
type "int *"
```

If a pointer is assigned to a far (volatile) data object, the pointer must be declared as pointing to a volatile object. Above, *p* itself has been declared as volatile because it is in far memory, but it was not declared as pointing to a volatile object. The corrected C-code is:

```
__attribute__((far)) volatile int x;           // x is in far memory
volatile __attribute__((far)) int* volatile p; // p is in far memory
void foo(void) {
p = &x;                                        // p points to x
}
```

Example 12. Volatile Object Address Passed to a Function as a Non-Volatile Pointer

```
int foo(int, int*);                            // function prototype
__attribute__((far)) volatile int x;           // x is in far memory
__attribute__((far)) volatile int y[10];       // y is in far memory

int main(void) {
    int z;                                     // z is local (on stack)
    z = foo(x, y);
}

int foo(int a, int* p) {
    return(a + *p);
}
```

A compiler warning will be issued:

```
warning #169-D: argument of type "volatile int *" is incompatible with parameter of
type "int *"
```

This mistake is very similar to Example 11. Instead of an explicit assignment, however, the non-volatile pointer is assigned the address of a volatile object as a passed function parameter. It is tempting to use typecasting to fix the warning, but this is not correct.

```
    z = foo(x, (int*)y);                       // incorrect and dangerous!
```

While the typecast eliminates the warning, it does not protect against the *foo()* function using the program bus instructions with the pointer *p*. The proper way to correct this code is to declare *p* as pointing to a volatile object:

```
int foo(int, volatile int*);                  // function prototype
__attribute__((far)) volatile int x;          // x is in far memory
__attribute__((far)) volatile int y[10];      // y is in far memory

int main(void) {
    int z;                                    // z is local (on stack)
    z = foo(x, y);
}

int foo(int a, volatile int* p) {
    return(a + *p);
}
```

4 Recommended Use of Far Memory

It is recommended to only use SDRAM for storage of large data blocks (arrays), and that a given block be copied to internal memory prior to operating on it. Operating on the data directly in the SDRAM requires you to consistently apply volatile to all pointers that might reference the data. The volatile type qualifier can have a significant performance impact on compiler generated code. There is also the potential performance degradation from the SDRAM itself to consider, which on TMS320F2837x/2807x devices is relatively slow. Additionally, there are currently no plans to enhance the majority of Texas Instruments™ provided C2000™ software collateral (for example, libraries) with support for far memory data objects. Therefore, using a store-and-copy approach to the SDRAM is preferred.

When implementing a store-and-copy approach with the SDRAM, either the DMA or the CPU can be used to copy the block. If using the CPU, the *memcpy_fast_far()* function in the C28x FPU DSP library [(2)] v1.50.00.00 (and later) provides a cycle efficient mechanism. There are also two example projects in controlSUITE™ that illustrate the store-and-copy approach. The project *emif1_16bit_sdram_far* uses the CPU to move the data with the *memcpy_fast_far()* routine. The project *emif1_16bit_sdram_dma* uses the DMA to do the data movement.

5 Performance Tips

Although the recommended usage for far memory (SDRAM) on TMS320F2837x/2807x devices is store-and-copy (see Section 4) you may have no alternative than to make wider use of it in your application. For such situations, the following performance tips are provided:

1. Restrict ‘far’ usage to global objects accessed with pointers (arrays).

   Avoid making random access data far since the direct addressing mode cannot be used with far memory. The indirect addressing mode that is employed is less efficient since the XARn auxiliary register needs to be reloaded for access to each data object (as opposed to with arrays that are typically sequentially accessed and, therefore, can utilize the increment and decrement hardware modifiers at no cost, such as *XARn++).

2. Keep pointers in near memory (even if they point to far objects).

   (a) Pointers in far memory can incur performance degradation because of the volatile designation. They are stored and recalled for each use, and pointer adjustment can even be done manually (read, modify, write-back) rather than using the no-cost increment and decrement hardware modifiers for indirect addressing (XARn++).

   (b) Pointers are generally limited in number so they can be easily stuffed into internal RAM.

   (c) Non-static local pointers are always placed on the stack and will always be near.

[(2)] The C28x FPU library is available in the controlSUITE software package, which can be downloaded from the following URL: http://www.ti.com/tool/CONTROLSUITE.

3. Keep constants in near memory.

   The only far memory on TMS320F2837x/2807x devices is external SDRAM. There is no benefit to placing constants there since the access performance of the flash is better than that of the SDRAM. In addition, the SDRAM is not persistent when power is removed so any constants located there would need to be copied from non-volatile memory (Flash) at runtime anyway. The ability to locate constants in far memory is really just provided by the compiler for completeness.

4. Heed the compiler warnings.

   The compiler is quite diligent about issuing warnings where far data object usage is concerned. These warnings should never be ignored. It almost always means the code will not function correctly. Properly fix the code to eliminate the warning.

5. Beware of unintended consequences of far data being ‘volatile’.

   The volatile keyword forces the compiler to do more than just avoid program bus instructions. It causes the compiler to do exactly what the C-code indicates, and eliminates many optimizations that the compiler might otherwise make. For example, the following three code snippets will generate different runtime behavior:

   Case 1: Original code - x will be read 3 times, and written 2 times

```
__attribute__((far)) volatile float x;

void foo(void) {
   x = x*x;
   x = x + 6;
}
```

   Case 2: Combine operations into a single instruction - x will be read 2 times, and written 1 time

```
__attribute__((far)) volatile float x;

void foo(void) {
   x = x*x + 6;
}
```

   Case 3: Use non-volatile intermediate variables - x will be read 1 time and written 1 time

```
__attribute__((far)) volatile float x;

void foo(void) {
   float tmp = x;
   tmp = tmp*tmp;
   x = tmp + 6;
}
```

6. Library functions may not work correctly with far data objects.
   (a) The root cause of the failure is the called function using a program bus instruction to access the far data.
   (b) Any far argument passed as a non-volatile pointer argument may have issues (pointers that point to far memory, arrays located in far memory where the array address is passed as a pointer). However, the compiler should issue a warning when the function is called since a pointer accessing far memory will have been declared as volatile, and the function prototype and declaration are not expecting this (see Example 12 in Section 3.2).
   (c) Any argument passed by value will be handled correctly, and will produce no compiler warnings. This is because the value of the argument is accessed by the parent function, and passed to the child. The child function never accesses the far data object directly. Library functions that only have pass-by-value arguments are safe to use. For example, such standard runtime support library math functions as *sin()* and *pow()* will work.

```
double sin(double x);
double pow(double x, double y);
```

   (d) To make an existing C function work with far data, adjust the function prototype and declaration to add volatile to any passed pointers that point to far data objects. The function should then be re-compiled. If any local pointers have been declared in the function and are assigned (equated) to a passed pointer to far data, a compiler error will be generated. The local pointer declaration should then be modified to declare the pointer as pointing to far data.
   (e) To make an existing ASM function work with far data, examine the assembly code source for instructions that use the program bus (MAC, DMAC, XMAC, XMACD, IMACL, QMACL, PREAD, PWRITE, XPREAD, XPWRITE), and modify the function as needed. If source code is not available, the disassembly window in the debugger can be used to inspect for these instructions and at least determine whether or not the function is safe to use.
   (f) Be aware that library functions can change if the library is revised.
   (g) Far global and static local objects with initialization values will be correctly initialized by the runtime support library C-environment initialization routine, *_c_int00()*. For example:

```
__attribute__((far)) volatile float x = 5;                    // x is in far memory

void foo(void) {
    static volatile __attribute__((far)) int y = 8;           // y is in far memory
  …
}
```

   In the above, *x* and *y* are initialized to 5 and 8, respectively, during execution of the *_c_int00()* routine; nothing special needs to be done for this to occur.

6 Conclusion

The coding methodology for accessing far memory data objects on the TMS320F2837x/2807x devices using the C/C++ language has been presented. The technique utilizes the GCC-style attribute *__attribute__((far))* combined with the *volatile* keyword to notify the compiler of far memory objects. While the compiler and device architecture themselves place few restrictions on far memory usage, from a performance perspective it is recommended that far memory to a store-and-copy mode of operation is limited.

7 References

1. *TMS320C28x CPU and Instruction Set Reference Guide* (SPRU430)
2. *TMS320F2837xD Dual-Core Delfino™ Microcontrollers Data Manual* (SPRS880)
3. *TMS320F2837xS Delfino™ Microcontrollers Data Manual* (SPRS881)
4. *TMS320F2807x Piccolo™ Microcontrollers Data Manual* (SPRS902)
5. *Running an Application from Internal Flash Memory on the TMS320F28xxx DSP* (SPRA958)

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as “components”) are sold subject to TI’s terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI’s terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

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

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license 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 significant portions of TI 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. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

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

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI’s goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or “enhanced plastic” are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have ***not*** been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

Products		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Applications Processors	www.ti.com/omap	**TI E2E Community**	e2e.ti.com
Wireless Connectivity	www.ti.com/wirelessconnectivity		

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