Application Report
SPRA577B

Using the TMS320 DSP Algorithm Standard in a Static DSP System

Carl Bergman *Digital Signal Processing Solutions*

Abstract

The TMS320 DSP Algorithm Standard is part of TI's eXpressDSP (XDAIS) technology initiative. It allows system designers to easily integrate algorithms from a variety of sources (e.g., third parties, customers). However, in system design, flexibility comes with a price.

This price is paid in CPU cycles and memory space, both critical in all DSP systems, but perhaps most critical in a static system. For this application note, a static system is defined as one in which memory is allocated once and is used for the remainder of the system's life – there is no effort to reclaim or reuse memory. In contrast, a dynamic system is one in which the memory is reused while the application is executing. A dynamic system takes advantage of the available memory by sharing it between algorithms, by reclaiming it when an algorithm is deactivated, and by reusing it when another algorithm is activated.

Algorithms that comply with the TMS320 DSP Algorithm Standard are tested and awarded an eXpressDSP compliant mark upon successful completion of the test. This application note shows how an eXpressDSP-compliant algorithm may be used effectively in a static system with limited memory. It examines some optimizations and illustrates them with a very simple example: an algorithm that copies input to output. The impact in terms of code size, data size, and CPU cycles will be demonstrated.

Contents

Theory of Operation 3
Review of TMS320 DSP Algorithm Standard Fundamentals 3
Naming Conventions 3
Interface Function Summary 4
Sequence of Builds 6
Build 1: No eXpressDSP Interface – Access Algorithm Directly 6
Build 2: Using the High Level Interface, 'CPY' 8
Build 3: Using and Removing Subsections in the Linker Command File 10
Build 4: Removing the Code from the CPY High-Level Interface 12
Build 5: Using Only the SPI – Creating the Object at Design Time 12
Conclusion 13
References 14
TI Contact Numbers 15

Figures

Figure 1. Test Program 6
Figure 2. Build 1 Code Size 8
Figure 3. Using the TMS320 Algorithm Standard Interface 9
Figure 4. Build 2 Code Size 10
Figure 5. Define Subsections 11
Figure 6. NOLOAD Section in Linker Command File 11
Figure 7. Build 3 Code Size 11

Theory of Operation

The TMS320 DSP Algorithm Standard provides a general-purpose interface that allows efficient use of a large variety of algorithms in a large variety of systems. However, the full capability of the interface may not be useful in all systems. In a static system we might allocate memory at design-time and initialize the algorithm at power-on and never change anything else. In such a system, the code implementing the *create* and *delete* functions, although never used, would take up valuable memory.

This application note follows an example program through a sequence of steps aimed at reducing code size by linking only the required functions. The unused code is assigned to a subsection that will not be loaded by the linker. The steps in the examples involve incrementally more programming effort. The result is that the code is smaller and less memory is used.

We begin with a typical implementation of the interface and then illustrate the process with several optimizations. The first build provides a baseline for comparison. It calls the algorithm directly with no algorithm standard interface. The second build adds the full algorithm standard interface. The remaining examples simplify the use of the interface and recover the memory from the unused functions.

Review of TMS320 DSP Algorithm Standard Fundamentals

Some of the key structures of an eXpressDSP-compliant algorithm are:

- ❒ Memory Table: Describes what memory the algorithm needs in order to operate
- ❒ Creation Parameters: Describes how the algorithm should be initialized
- ❒ Status Information: Describes the current state of the algorithm
- ❒ Function Table: Describes the operations available for the algorithm

There are two levels of access to the algorithm:

1) The service provider interface (SPI) provides the most direct access.

2) The application programmer's interface (API) provides an alternate, more convenient interface.

The high-level functions of the API use the SPI to create and control the algorithm and to process data.

Naming Conventions

The TMS320 Algorithm Standard naming convention ensures that implementations of the same algorithm from different vendors can co-exist without duplicate symbols. This is made possible by defining a two-part prefix to external symbols. Part one of the prefix represents the algorithm and part two represents the vendor.

In our example, the symbol for the 'copy' algorithm is the mnemonic 'CPY'. The symbol for the vendor “Texas Instruments” is the acronym 'TI'. This yields the prefix 'CPY_TI_'. An example of a function name using this prefix would be CPY_TI_create(). This name indicates that TI implements the create function for the copy algorithm.

An example of an interface name would be 'CPY_TI_ICPY'. This name indicates that TI implements the interface to the copy algorithm (ICPY) for the copy algorithm. This may sound redundant, but there are other possible interfaces to the copy algorithm. For example, the test interface (ITST) in this example would be named 'CPY_TI_ITST'.

Interface Function Summary

The functions that implement the two levels of access (API and SPI) may be organized according to whether they apply to all algorithms (generic), apply to a specific algorithm (algorithm-specific), or apply to a specific implementation of an algorithm (vendor-specific). The naming convention helps here as well. The generic create function would be ALG_create(). The algorithm-specific create function would be CPY_create() with the copy algorithm mnemonic as a prefix. The vendor-specific function (if TI was the vendor) would have the name CPY_TI_create().

Functions beginning with 'CPY_' (Algorithm Specific API)

The algorithm-specific API is the most convenient access to the algorithm and is a superset of the TMS320 algorithm standard API.

CPY_activate()	Prepare the algorithm to run
CPY_control()	Command and status mechanism
CPY_create()	Allocate memory and initialize a new algorithm instance
CPY_deactivate()	Prepare the algorithm to be inactive or possibly deleted
CPY_delete()	Remove algorithm instance and deallocate the memory used
CPY_exit()	Finalize module other than deleting algorithm instance
CPY_init()	Initialize module other than creating algorithm instance
CPY_process()	Process data

Functions Beginning With 'ALG_' (Standard API)

The following do not include the algorithm-specific processing function calls.

ALG_activate()	Prepare the algorithm to run
ALG_control()	Command and status mechanism
ALG_create()	Allocate memory and initializes a new algorithm instance
ALG_deactivate()	Prepare the algorithm to be inactive or possibly deleted
ALG_delete()	Remove algorithm instance and deallocate the memory used
ALG_exit()	Finalize module other than deleting algorithm instance
ALG_init()	Initialize module other than creating algorithm instance

The CPY_IALG Interface (Standard SPI)

The IALG interface functions are described in the comments field of the ialg.h file.

```
/*
 *  ======== IALG_Fxns ========
 *  This structure defines the fields and methods that must be supplied by
 *  all XDAIS algorithms.
 *
 *      implementationId  - unique pointer that identifies the module
 *                          implementing this interface.
 *      algActivate()     - notification to the algorithm that its memory
 *                          is "active" and algorithm processing methods
 *                          may be called.  May be NULL; NULL => do nothing.
 *      algAlloc()        - apps call this to query the algorithm about
 *                          its memory requirements. Must be non-NULL.
 *      algControl()      - algorithm specific control operations.  May be
 *                          NULL; NULL => no operations supported.
 *      algDeactivate()   - notfication that current instance is about to
 *                          be "deactivated".  May be NULL; NULL => do nothing.
 *      algFree()         - query algorithm for memory to free when removing
 *                          an instance.  Must be non-NULL.
 *      algInit()         - apps call this to allow the algorithm to
 *                          initialize memory requested via algAlloc().  Must
 *                          be non-NULL.
 *      algMoved()        - apps call this whenever an algorithms object or
 *                          any pointer parameters are moved in real-time.
 *                          May be NULL; NULL => object can not be moved.
 *      algNumAlloc()     - query algorithm for number of memory requests.
 *                          May be NULL; NULL => number of mem recs is less
 *                          then IALG_DEFMEMRECS.
 */
```

The CPY_ICPY Interface (Standard SPI Plus Algorithm Extensions)

The algorithm extensions provide the data processing function.

cpyProcess()	Copy data from input buffer to output buffer

The CPY_TI_ICPY Interface (Standard SPI Plus Algorithm Extensions Plus Vendor's Extensions)

The copy algorithm has no vendor extensions.

Sequence of Builds

Build 1: No eXpressDSP Interface – Access Algorithm Directly

The first build is for comparison purposes. The test program accesses the algorithm directly. The copy algorithm only needs the count field in the object and the input and output data buffers. Note that the eXpressDSP header files are included to support the use of the ICPY_TI_Obj structure, which is expected by the algorithm. Figure 1 shows the code from main() of the test program.

The system resources used are measured in terms of code size and CPU cycles. The code size is shown in the excerpt from the linker map file in Figure 2. The CPU cycles for the data processing function are determined with the profiler in Code Composer Studio [1].

Program Memory Used	37,408 bytes
Data Memory Used	4,480 bytes
CPU Cycles Used	20441 (average of 3 runs on a C6201 EVM card)

Figure 1. Test Program

```
/*
 *  build1.c
 */
#include <stdio.h>              // access to printf()

#include <std.h>                // basic data types
#include <xdais.h>               // XDAIS data types

#include <ialg.h>               // IALG standard
#include <icpy.h>               // ICPY standard
#include <icpy_ti.h>            // ICPY implementation

#include <copydata.h>           // algorithm implementation


/* test data */
#define COPY_COUNT 16
#define BUFFER_SIZE 80
Char * testString = "eXpress DSP Algorithm Standard";
Char buffer[BUFFER_SIZE];


Int main()
{
    ICPY_TI_Handle handle;
    ICPY_TI_Obj cpyObj;
    Char *cp, *input, *output;
    Int i;


    printf("build1 1999 0802 1036\n");
    /* init test buffers ---------------------------------------- */

    input = testString;
    output = buffer;

    /* clear output buffer */
    cp = output;
    for (i = BUFFER_SIZE; i > 0; i--) {
        *cp++ = (Char)0;
    }

    printf("input: %s\n", input);
    printf("output: %s\n", output);

    /* init the algorithm ---------------------------------------*/

    handle = (ICPY_TI_Handle)&cpyObj;

    /* if create failed then exit (can't happen but keep consistent) */
    if (handle == NULL) {
        printf ("object creation failed\n");
        exit(1);
    }
    else {
        printf ("object created\n");
    }

    /* use the algorithm ----------------------------------------*/

    /* set the count of bytes to copy */
    printf ("cpyControl\n");            // for fair code comparison
    printf ("ICPY_SET_COPY_COUNT\n");   // for fair code comparison
    cpyObj.count = 5;

    i = 1;                              // place profile point here

    /* do the copy operation */
    printf ("cpyProcess\n"); // for fair code comparison
    copyData(handle, input, output);

    i = 0;                              // place profile point here

    /* report results ------------------------------------------- */

    printf("copy %d bytes, output: %s\n", cpyObj.count, output);
    printf("end of build1\n");
        /* for fair code size comparison: from the control function */
        printf ("ICPY_READ_STATUS\n");
        printf ("ICPY_WRITE_STATUS\n");
        printf ("default case!\n");
    return(0);
}
```

Figure 2. Build 1 Code Size

```
MEMORY CONFIGURATION

           name      origin    length      used    attributes    fill
         --------  --------  ---------  --------  ----------   --------
         PMEM      00000000  000010000  00000000     RWIX
         EXT0      00400000  000040000  00000000     RWIX
         EXT1      01400000  000300000  00000000     RWIX
         EXT2      02000000  000400000  00009220     RWIX
         EXT3      03000000  000400000  00000000     RWIX
         DMEM      80000000  000010000  00001180     RWIX


SECTION ALLOCATION MAP

section   page    origin      length       input sections
--------  ----  ----------  ----------   ----------------
.text      0    02000000    00008500
.cinit     0    02008500    00000414
.cio       0    02008914    00000120     UNINITIALIZED
.far       0    02008a34    000007ec     UNINITIALIZED
.stack     0    80000000    00000800     UNINITIALIZED
.bss       0    80000800    00000054     UNINITIALIZED
.const     0    80000854    0000012c
.sysmem    0    80000980    00000800     UNINITIALIZED
```

Build 2: Using the High Level Interface, 'CPY'

The second build represents a baseline of using the copy algorithm in a static system with an algorithm standard interface.

The code is built using Code Composer Studio (CCStudio) with the following components:

Build2.mak	Code Composer Studio Project File
Build2.c	Test Program
mem.c	Memory Allocation Utility
cpy.c	Algorithm Specific High Level Interface
alg.c	Standard High Level Interface
cpy.lib	Algorithm Library
Build2.cmd	Linker Command File

System resources used:

	Build 1	**Build 2**	**Change**
Program Memory (bytes)	37408	40456	3048
Data Memory (bytes)	4480	4615	135
CPU Cycles	20441	21343	902

The code in Figure 3 is an excerpt from the function main() in build2.c and shows the following steps:

1) Using the high level CPY API, an instance of the algorithm is created.
2) The copy count is then changed from the default value with a control call and the copy process is run on the input and output buffers.
3) Finally, a control call is made to retrieve status, which in our case simply proves we can find out what the copy count was.

Figure 3. Using the TMS320 Algorithm Standard Interface

```
   /* init the algorithm --------------------------------------- */

   /* create an instance of the algorithm object */
   handle = CPY_create(&CPY_ICPY, &paramDefaults);

   /* if create failed then exit */
   if (handle == NULL) {
printf ("object creation failed\n");
exit(1);
   }
   else {
printf ("object created\n");
   }

   /* use the algorithm ---------------------------------------- */

   /* set the count of bytes to copy */
   CPY_control(handle, ICPY_SET_COPY_COUNT, (Void *)5);

   i = 1;    /* place profile point here */

   /* do the copy operation */
   CPY_process(handle, input, output);

   i = 0;    /* place profile point here */

   /* read back the copy count */
   CPY_control(handle, ICPY_READ_STATUS, &status);
```

Figure 4 is an excerpt from the Build 2 linker map file.

Figure 4. Build 2 Code Size

```
MEMORY CONFIGURATION

           name     origin    length      used    attributes    fill
         --------  --------  ---------  --------  ----------  --------
         PMEM      00000000  000010000  00000000     RWIX
         EXT0      00400000  000040000  00000000     RWIX
         EXT1      01400000  000300000  00000000     RWIX
         EXT2      02000000  000400000  00009e08     RWIX
         EXT3      03000000  000400000  00000000     RWIX
         DMEM      80000000  000010000  00001207     RWIX


SECTION ALLOCATION MAP

section   page    origin      length       input sections
--------  ----  ----------  ----------   ----------------
.text      0    02000000    00009080
.cinit     0    02009080    0000047c
.cio       0    020094fc    00000120     UNINITIALIZED
.far       0    0200961c    000007ec     UNINITIALIZED
.stack     0    80000000    00000800     UNINITIALIZED
.bss       0    80000800    0000008c     UNINITIALIZED
.const     0    8000088c    0000017b
.sysmem    0    80000a08    00000800     UNINITIALIZED
```

Build 3: Using and Removing Subsections in the Linker Command File

In the next build, the code is the same as Build 2 (refer to Figure 3). We now add pragma directives to all the interface levels to assign an input subsection for each function statement (see Figure 5). This allows us to selectively include or exclude the subsections in the link process.

System resources used:

	Build 1	**Build 3**	**Change**
Program Memory (bytes)	37408	39432	2024
Data Memory (bytes)	4480	4615	135
CPU Cycles	20441	21132	691

Figure 5. Define Subsections

```
#pragma CODE_SECTION(CPY_activate, ".text:algActivate")
#pragma CODE_SECTION(CPY_apply, ".text:algApply")
#pragma CODE_SECTION(CPY_control, ".text:algControl")
#pragma CODE_SECTION(CPY_create, ".text:algCreate")
#pragma CODE_SECTION(CPY_deactivate, ".text:algDeactivate")
#pragma CODE_SECTION(CPY_delete, ".text:algDelete")
#pragma CODE_SECTION(CPY_exit, ".text:algExit")
#pragma CODE_SECTION(CPY_init, ".text:algInit")
```

Subsections are selected in the linker command file. By specifying a NOLOAD output section, the unused code is removed from the program image (see Figure 6). The code is built the same way as in Build 2.

Figure 6. NOLOAD Section in Linker Command File

```
SECTIONS
{
    ...

    .notUsed {
 * (.text:algActivate)
 * (.text:algApply)
 * (.text:algDeactivate)
 * (.text:algDelete)
 * (.text:algExit)
 * (.text:algInit)
 * (.text:algMoved)
 * (.text:algNumAlloc)
    } type = NOLOAD > EXT3

    ...
}
```

Figure 7 is an excerpt from the Build 3 linker map file.

Figure 7. Build 3 Code Size

```
MEMORY CONFIGURATION

           name     origin    length      used    attributes    fill
         --------  --------  ---------  --------  ----------  --------
         PMEM      00000000  000010000  00000000     RWIX
         EXT0      00400000  000040000  00000000     RWIX
         EXT1      01400000  000300000  00000000     RWIX
         EXT2      02000000  000400000  00009a08     RWIX
         EXT3      03000000  000400000  00000400     RWIX
         DMEM      80000000  000010000  00001207     RWIX
```

```
SECTION ALLOCATION MAP

section   page    origin      length       input sections
--------  ----  ----------  ----------   ----------------
.text      0    02000000    00008c80
.cinit     0    02008c80    0000047c
.cio       0    020090fc    00000120     UNINITIALIZED
.far       0    0200921c    000007ec     UNINITIALIZED
.stack     0    80000000    00000800     UNINITIALIZED
.bss       0    80000800    0000008c     UNINITIALIZED
.const     0    8000088c    0000017b
.sysmem    0    80000a08    00000800     UNINITIALIZED
.notUsed   0    03000000    00000400     NOLOAD SECTION
```

Build 4: Removing the Code from the CPY High-Level Interface

In the fourth build, we replace the calls to the CPY interface (CPY_* functions) with macros that call the standard API (ALG_* functions) and the SPI. The three macros shown replace the corresponding function calls to CPY_control(), CPY_create() and CPY_process().

```
#define CPY_CONTROL(alg, cmd, status) \
    ((alg->fxns->ialg.algControl)((IALG_Handle)alg, cmd, status));

#define CPY_CREATE(fxns, prms) \
    (CPY_Handle)ALG_create((IALG_Fxns *)fxns, (IALG_Params *)prms);

#define CPY_PROCESS(alg, input, output) \
    (alg->fxns->cpyProcess)((ICPY_Handle)alg, input, output);
```

This allows us to eliminate the file cpy.c from our build. The rest remains the same.

System resources used:

	Build 1	**Build 4**	**Change**
Program Memory (bytes)	37408	39224	1816
Data Memory (bytes)	4480	4611	131
CPU Cycles	20441	20725	284

Build 5: Using Only the SPI – Creating the Object at Design Time

In Build 5, we remove the remaining API code in alg.c from the program. We can do this because we are going to 'create' the object and declare the data structures the algorithm requires at design time in the test program. Four steps are required for this build:

1) Allocate the space for the memory descriptor table.
```
memTab =
(IALG_MemRec *)malloc(sizeof(memTab[IALG_DEFMEMRECS]));
```

2) Plug in the addresses of our allocated object and working memory to the memory descriptor table.

```
memTab[CPY_OBJ_DATA].base =
(void *)malloc(sizeof(ICPY_TI_Obj));
memTab[CPY_DATA_RAM].base =
(void *)malloc(sizeof(cpyDataRam));
```

3) Set the value of our handle to the algorithm. We also set the address of the function table in the object. Previously ALG_create() set the function table address and returned the value of our handle.

```
handle = (CPY_Handle)memTab[CPY_OBJ_DATA].base;
handle->fxns = &CPY_ICPY;
```

4) Initialize the algorithm. For this, we call the SPI directly with the parameters it expects. If the initialization fails, the handle is set to NULL.

```
    if (handle->fxns->ialg.algInit(
        (IALG_Handle)handle, memTab,
        NULL, (IALG_Params *)&paramDefaults
    ) != IALG_EOK) {
        handle = NULL;
    }
```

Now with alg.c and mem.c removed from the program (memory allocation is no longer used) and with the subsection .text:algAlloc placed in the .notUsed section, we build the program as before.

System resources used:

	Build 1	Build 5	Change
Program Memory (bytes)	37408	38232	824
Data Memory (bytes)	4480	4611	131
CPU Cycles	20441	20653	212

Conclusion

These build techniques allowed us to reduce the program memory overhead for using the algorithm standard from 3048 bytes to 824 bytes, a 70% reduction. This was accomplished by using only the service provider interface (SPI) and by placing unused code in a NOLOAD section.

	Build 1	**Build 2**	**Build 3**	**Build 4**	**Build 5**
Program Memory	37408	40456	39432	39224	38232
Change		3048	2024	1816	824
Percent of XDAIS		100.00%	66.40%	59.58%	27.03%

The data memory was not really affected, with a change of only 4 bytes from Build 3 to Build 4.

	Build 1	Build 2	Build 3	Build 4	Build 5
Data Memory	4480	4615	4615	4611	4611
Change From Build 1		135	135	131	131

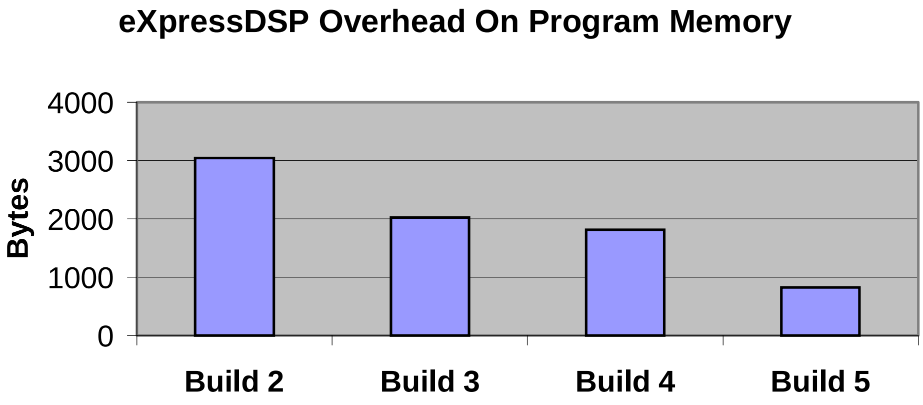
The CPU cycle count for the data processing call was measured with the profiler in Code Composer Studio. Because our copy algorithm has only 160 bytes of code, it is important to note that the percentage of overhead of the algorithm standard interface in a more realistic algorithm would be much smaller than what is shown.

With that in mind, the most direct use of the SPI gives us a cycle count of 212 – a little more than 1% of the total cycles used in the data processing call. This is less than 24% of the 902 cycles used with the full algorithm standard interface in Build 2.

	Build 1	Build 2	Build 3	Build 4	Build 5
CPU Cycles	20441	21343	21132	20725	20653
Change		902	691	284	212
Percent of Total		4.41%	3.38%	1.39%	1.04%

In an actual case with a G.723 algorithm, the processing call takes an average of 375,000 cycles, and the overhead of 212 cycles would be less than 0.1%. The overhead of the full interface at 902 cycles would be less than 0.25%.

Finally, the following chart summarizes the improvements in program memory for the examples given.



References

1. *Code Composer Studio User's Guide*, SPRU328.
2. *TMS320C6000 Assembly Language Tools User's Guide*, SPRU186.
3. *TMS320 DSP Algorithm Standard Rules and Guidelines*, SPRU352.
4. *TMS320 DSP Algorithm Standard API Reference, SPRU360.*

TI Contact Numbers

INTERNET

TI Semiconductor Home Page
www.ti.com/sc

TI Distributors
www.ti.com/sc/docs/distmenu.htm

PRODUCT INFORMATION CENTERS

Americas

Phone	+1(972) 644-5580
Fax	+1(972) 480-7800
Email	sc-infomaster@ti.com

Europe, Middle East, and Africa

Phone	
Deutsch	+49-(0) 8161 80 3311
English	+44-(0) 1604 66 3399
Español	+34-(0) 90 23 54 0 28
Francais	+33-(0) 1-30 70 11 64
Italiano	+33-(0) 1-30 70 11 67
Fax	+44-(0) 1604 66 33 34
Email	epic@ti.com

Japan

Phone	
International	+81-3-3344-5311
Domestic	0120-81-0026
Fax	
International	+81-3-3344-5317
Domestic	0120-81-0036
Email	pic-japan@ti.com

Asia

Phone	
International	+886-2-23786800
Domestic	
Australia	1-800-881-011
TI Number	-800-800-1450
China	10810
TI Number	-800-800-1450
Hong Kong	800-96-1111
TI Number	-800-800-1450
India	000-117
TI Number	-800-800-1450
Indonesia	001-801-10
TI Number	-800-800-1450
Korea	080-551-2804
Malaysia	1-800-800-011
TI Number	-800-800-1450
New Zealand	000-911
TI Number	-800-800-1450
Philippines	105-11
TI Number	-800-800-1450
Singapore	800-0111-111
TI Number	-800-800-1450
Taiwan	080-006800
Thailand	0019-991-1111
TI Number	-800-800-1450
Fax	886-2-2378-6808
Email	tiasia@ti.com

TI is a trademark of Texas Instruments Incorporated.

Other brands and names are the property of their respective owners.

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

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

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. 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 of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.