Application Report
SPRA635 - March 2000

TMS320C62x™ Self-check Program Applications Brief: Version 1.0

Syed A. Maroof *Member, Technical Staff*

ABSTRACT

This program is designed to verify the operation of DSP devices in the Texas Instruments TMS320C62x™ family by checking for proper operation of the CPU core and internal RAM. On-chip peripherals are not tested since doing so would be target system dependent. It is important to note that this program only provides a confidence check. It is not as comprehensive as the tests done at production, which thoroughly check the device's logic, performance, and electrical parameters. While every attempt has been made to provide a compact self-check program that will exercise as much of the device as is possible, this program is not capable of detecting all potential faults.

This application report will cover the methodology used to check all TMS320C62x instructions. There are primarily five main categories into which these instructions can be arranged:

- Arithmetic
- Logical
- Data Management
- Bit Management
- Program Control

Contents

TMS320C62x is a trademark of Texas Instruments

List of Figures

List of Tables

1 Introduction

This program is designed to verify the operation of DSP devices in the Texas Instruments (TI™) TMS320C62x family by checking for proper operation of the CPU core and internal RAM. On-chip peripherals are not tested since doing so would be target system dependent. It is important to note that this program only provides a confidence check. It is not as comprehensive as the tests done at production, which thoroughly check the device's logic, performance, and electrical parameters. While every attempt has been made to provide a compact self-check program that will exercise as much of the device as is possible, this program is not capable of detecting all potential faults.

This application report will cover the methodology used to check all TMS320C62x instructions. There are primarily five main categories into which these instructions can be arranged:

- Arithmetic
- Logical
- Data Management
- Bit Management
- Program Control

1.1 Scope of the Self-check

This C-callable program utilizes all 'C62x instructions. There are routines that check for circular addressing mode (alternative to linear addressing mode) and 40-bit arithmetic instructions. These instruction tests effectively check much of the functional units and data paths. In addition, internal data RAM block, status registers AMR, CSR, IRP and NRP are tested. Since the NMI enable bit is disabled or low (NMIE = 0) at hardware reset, no interrupts will be taken while self-check code is executing. Therefore, control registers ICR, IER and ISR were not tested. Specific blocks of external RAM can be optionally tested.

1.2 System Requirements

To use this code, you will need the following tools:

TMS320C6x Assembly Language Tools
TMS320C62x Simulator or XDS510 Emulator
A suitable host PC to support the above tools

If you intend to call the self-check from a C-language program, you will additionally need:

TMS320C6x Optimizing C Compiler

2 Program Files

The following files are included in the self-check program:

ALU.ASM	Arithmetic operations module
ALU40.ASM	40-bit Arithmetic operations module
BASIC.ASM	Basic operations module
BIT.ASM	Bit management module
CHK6x.ASM	Main program control shell
CIRCULAR.ASM	Circular addressing instructions module
COND.ASM	Branch and conditional instructions module
MEM.ASM	Internal DATA RAM operation module
MULT.ASM	Multiplier operations module
SAT.ASM	Saturation instructions module
VECTORS.ASM	Vector Table (Branching to _c_init00)
USER.C	Sample C program that calls the self-check program
CHK6x.CMD	Sample linker command file for use with USER.C sample program
OPTIONS.H	Include file containing user setable options
MAKE.BAT	DOS batch file to assemble and link the code
CHK6X.DOC	This text file
READ.ME	README text file

2.1 How to Call the Self-check Program

The self-check program is designed to be C-callable, although it can also be easily called from assembly code. The files USER.C and CHK6x.CMD are provided as examples of a self-check calling C program and its linker command file. If the self-check passes, it will use register A4 to return a pass code to the calling routine. If the self-check fails, it will either use register A4 to return an error code to the calling routine, or it can optionally lock out the calling routine (explained in section 3.4). This option is activated in the options.h file. Register A4 containing the return value is part of the standard convention for the TMS320C62x C compiler. Assembly code users should simply expect to find the return value in register A4.

If calling the self-check from C code, use a statement of the form

```
errorcode = chk6x();
```

where errorcode has been declared as an unsigned integer. The C language function prototype for the self-check routine is:

```
extern unsigned chk6x();
```

If calling the self-check from assembly code, use the following instruction:

```
B     .S2   _chk6x
NOP   5
```

The leading underbar in _chk6x above is not a typo, and must be included.

2.2 Assembling and Linking the Code

The file MAKE.BAT is a DOS batch file that will assemble, compile and link the various program modules for you. It assumes that the self-check program is being called from the C program USER.C, and uses the linker command file CHK6x.CMD. It requires the program files to be in the current directory, and the directory containing the TMS320C6x Assembly Language Tools to either be in the current directory or be listed in the DOS path. The MAKE.BAT batch file is as follows:

```
cl6x –gs –als vectors.asm
cl6x –gs –als chk6x.asm
cl6x –gs –als cond.asm
cl6x –gs –als alu.asm
cl6x –gs –als basic.asm
cl6x –gs –als mult.asm
cl6x –gs –als bit.asm
cl6x –gs –als sat.asm
cl6x –gs –als circular.asm
cl6x –gs –als alu40.asm
cl6x –gs –als mem.asm
cl6x –gs –als user.c
cl6x –z chk6x.cmd
```

The cl6x commands invoke either the TMS320C6x compiler or assembler as required. One parameter is the name of the source file to be either assembled or compiled. The .ASM extension is routed through the assembler while .C extension is routed through the compiler. The command line options tell the assembler to generate TMS320C62X object code or the compiler to create .ASM for a C code (e.g. USER.ASM for USER.C). A summary of what each command line option does, is as follows:

`–g` Enables src-level symbolic debugging

Assembler only options:

`–als` creates assembler listing file and retains asm symbols for debugging

Compiler only options:

`–s` interlist C statements into assembly listing

The cl6x command with -z option invokes the TMS320C6x linker. The first parameter, “CHK6x.cmd”, is the name of the linker command file. This file lists the names of the object files to be linked, as well as the desired memory mapping. The linker command file contains several parameters. The -c option enables the program to use the ROM initialization model and “-lrts6201.lib” links the program with the required C-environment initialization routines (rts6201.lib is a run-time support library file that comes with the C Compiler). The “–o chk6x.out” specifies the name for the executable output file, and the “–m chk6x.map” specifies the name for the output map file.

USER.C and CHK6x.CMD could be replaced with your own custom routines as necessary. If you instead wish to call the self-check from assembly code, say the files YOURFILE.ASM and YOURFILE.CMD, modify the last two lines of MAKE.BAT so that YOURFILE is assembled and linked. The linker command file would also need some changes so that it does not initialize the C environment. The make.bat file might look like:

```
cl6x –gs –als vectors.asm
cl6x –gs –als chk6x.asm
cl6x –gs –als cond.asm
cl6x –gs –als alu.asm
cl6x –gs –als basic.asm
cl6x –gs –als mult.asm
cl6x –gs –als bit.asm
cl6x –gs –als sat.asm
cl6x –gs –als circular.asm
cl6x –gs –als alu40.asm
cl6x –gs –als mem.asm
cl6x –gs –als YOURFILE.ASM
cl6x –z YOURFILE.CMD
```

2.3 How to Allocate Memory for the Linker

The self-check program destructively overwrites all RAM locations that it either tests or uses. These include on-chip data and program memory blocks and all optionally tested external memory blocks. Therefore, programs should not be loaded nor any data sections initialized in tested RAM prior to running the self-check program. They will be lost. Here are several suggestions for allocating and running the self-check:

- Link and run all code from external RAM (or ROM), and do not exercise the external RAM test option on the used portion.
- The internal program space is not tested in the self-check code (only internal data memory is tested).
- If running the self-check as auxiliary support for some other primary application program, one option is to run the self-check using one of the above two linker methods before boot loading the code and initializing the data for your primary application. This will require you to modify your bootloader.

3 What is Tested

The self-check program is designed to verify the operation of DSP devices in the Texas Instruments TMS320C62x family by checking for proper operation of the CPU core. The internal data RAM test routine is only TMS320C6201 specific and would need modification to test other TMS320C62xx derivatives. The only modification it would need would be the start address and length for the memory blocks. Certain blocks of external RAM can also be optionally tested. The code was written using nearly every instruction in linear addressing mode and few in circular addressing mode. The only instructions not tested were IDLE and NOP. The code is designed to check for proper instruction execution rather than merely using the instruction. For example, the conditional branch instruction B has been implemented to both branch and not branch in the code. These instruction tests effectively check much of the functional units and data paths. The self-check program also tests all registers, all eight functional units and crosspaths. Most of the Read/Write status registers are also tested.

No on-chip peripherals are tested since doing so would be target system dependent. These include, but are not limited to, any serial ports, timers, and (DMA) Direct Memory Access.

3.1 How It is Tested

The self-check program tests the CPU by exercising a component of the CPU and then verifying the correct result. When an incorrect result occurs, the program loads an unsigned error code into a register (e.g., B0), and then returns to the calling program. After returning to the main control shell, it can either return the value to the calling C program (user.c) or optionally locks it out (as specified in options.h).

Internal data RAM is tested using a checkerboard pattern. This test sets every other bit in the RAM array to a 1 and then verifies that the pattern is correct by reading each word and checking bits. The test then switches the pattern bits and again checks the memory to verify that each memory bit can be set to a 0 and 1. The user also has an option to test external memory blocks by specifying it in options.h file before compiling and running the program.

NOTE: Future TMS320C6X devices might have a different memory configuration (memory map). Therefore, start address and the length for the external memory blocks as well as the internal data memory can be specified in the options.h file and is not hard coded. It must be configured prior to running the memory test routine (mem.asm).

Each test routine is passed some known values as arguments. These values are used to test each instruction and the result is compared with a known value. If the comparison is false, the test routine generates an error code and returns it to the main control shell. The values passed to each routine as arguments are:

```
A4 = 0FFFFh
B4 = 5454h
A6 = 5151h
B6 = 3333h
A8 = 0AAAAh
B8 = 2222h
A10 = 0FFFFFFFFh
B10 = 00000003h
```

In some code segments, a value is loaded onto a register and then used to test a particular instruction. The result is again compared with a known value. If the comparison fails, then an error code is generated and passed to the main control shell. For example, the following code:

```
;Known Results
            MVK     .S2     0103h, B0       ;B0 = 00000103h
            MVK     .S2     0106h, B1       ;B1 = 00000106h

;preliminary setup for the test routine
            MVK     .S1     0Bh, A2         ;A2 = 0000000Bh
            MVK     .S1     0100h, A5       ;A5 = 00000100h
            MVK     .S2     00020004h, B9   ;B9 = 00000004h
            MVKH    .S2     00020004h, B9   ;B9 = 00020004h

;setting A5 for circular addressing mode with size = 8 using BK0
            MVC     .S2     B9, AMR         ;AMR = B9
            ADDAB   .D1     A5, A2, A2      ;A2 = 00000103h
            CMPEQ   .L2x    B0, A2, B0      ;B0 = 1, if B0 = A2
            SUB     .L2     B0, 1, B0       ;B0 = B0 – 1 = 0
      [B0]  B       .S2     ERRCIR1         ;ERROR, if B0 != 0
            NOP     2
```

3.2 Preserved Registers

Since the self-check is C-callable, the routines are responsible for saving registers A10-A15 or B10-B15 at entry and then restoration prior to returning to the calling routine (main control shell).

In addition to these, some status registers are never modified by the self-check program, and may thus be considered reserved. They include IFR, ISR, ICR, IER, ISTP and PCE1. The NMI bit in the status register IER is zero at reset (disabled), which means that no interrupt will be taken during the operation of the self-check program even if the GIE bit is set while testing the CSR.

3.3 How the Program is Structured

The following diagram elaborates on the flow of the program and also how it can be called from a C program.

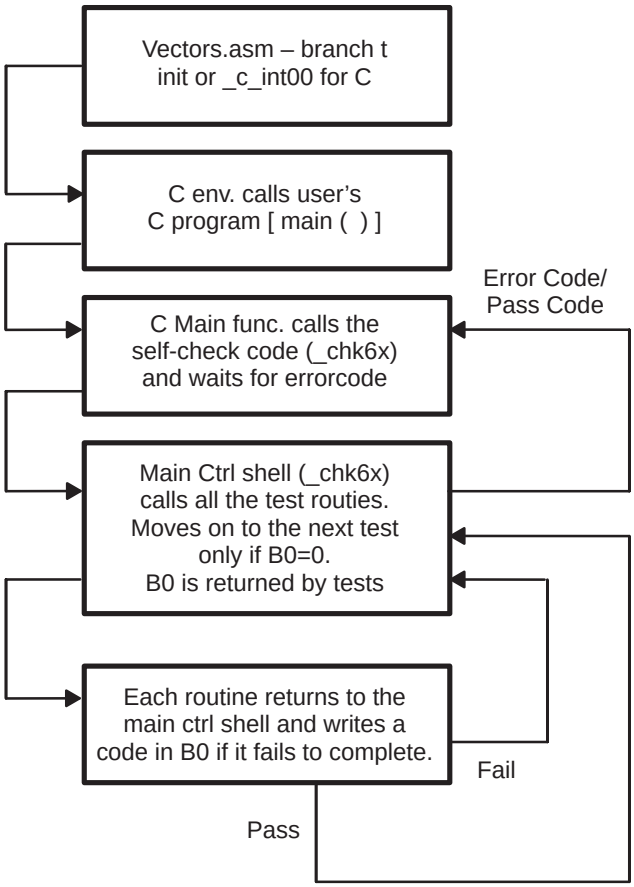

Figure 1. Flow Chart of the Program

The self-check program can be called using either the sample code (USER.C) or any other C program. There is a file called “vectors.asm” that is responsible for branching to either the C runtime support library function that initializes the C environment or to an assembly initialization section (as decided by the user). The C runtime support library initializing function calls the main function of a C program. In this case, it calls the main function in USER.C, which is as follows:

```
int main()
{
      errorcode = chk6x();
      return errorcode;
}
```

This main function calls the self-check code that is written in assembly and is formulated as a series of modules that test different components of the CPU (or different functional groupings of instructions). The main control shell (CHK6X.ASM) calls these test routines. The following is the order in which the test modules are called:

BASIC.ASM	Basic operations module
ALU.ASM	Arithmetic operations module
MULT.ASM	Multiplier operations module
BIT.ASM	Bit management module
SAT.ASM	Saturation instructions module
COND.ASM	Branch and conditional instructions module
CIRCULAR.ASM	Circular addressing instructions module
ALU40.ASM	40-bit Arithmetic operations module
MEM.ASM	Internal DATA RAM operation module

Each test routine could either fail or pass. If it fails, it sends a unique error code to the main control shell that is written in register B0. If it passes, it simply returns to the main shell. Returning from the test routine, the main control shell checks the value in B0. If B0 = 0, then it goes to a section PASS6X (see the code) which writes a pass code (0FFh) to A4 before returning to the C program. If B0 != 0, then it gives a choice to the user. Either the program can be locked out or return the error code to the calling C program. The option to lock out the process must be specified BEFORE compilation and running the program in the options file (as discussed in section 3.4).

In general, the code is designed to functionally check each instruction before that instruction is used to test a different operation. The above calling order meets this design, and therefore lends some validity to the error code returned. All tested instructions are checked in either the first six modules. The modules CIRCULAR.ASM, ALU40.ASM and MEM.ASM are targeted at hardware testing.

Table 1 identifies the module in which each instruction is tested. Note that the table attempts to identify where each instruction is functionally tested, rather than simply where each is used. For example, the following code checks for condition subtract and shift instruction (SUBC). It also uses other instructions but they have already been tested. Therefore, there is a clear distinction between using an instruction and testing it.

```
        MVK     .S1     01F63Fh, A3   ;A3 = 0000F63Fh
        MVKH    .S1     01F63Fh, A3   ;A3 = 0001F63Fh
        MVK     .S2     021A31h, B2
        MVKH    .S2     021A31h, B2   ;B2 = 00021A31h
        SUBC    .L2x    B2, A3, B13   ;B13 = 000047e5h
        SUB     .L1x    B2, A3, A15   ;A15 = 000023F2h
        CMPLT   .L2x    A15, 0h, B1   ;A15 is > 0 so B1 = 0
[!B1]   SHL     .S1     A15, 1, A15     ;A15 = 000047e4h
        ADD     .S1     A15, 01h, A15   ;A15 = 000047e5h
        CMPEQ   .L2x    B13, A15, B0  ;B0 =1, if A15 = B13
        SUB     .L2     B0, 01h, B0   ;B0 = 0
[B0]    B       .S1     ERRALU5       ;ERROR, if B0 != 0
        NOP     5
```

When writing the self-check program, a reasonable attempt was made to avoid or minimize non-detectable instruction failures.

Table 1. Opcode Functional Testing by Module

Opcode/ Module Name	Basic	ALU	Mult	Bit	Sat	Cond	Circular	ALU 40	Mem	Not Tested
ABS		√								
ADD		√						√		
ADDU		√						√		
ADDAB							√			
ADDAH							√			
ADDAW							√			
ADDK		√								
ADD2		√								
AMD		√								
B						√				
B (.unit) IRP	√									
B (.unit) NRP	√									
CLR				√						
CMPEQ		√								
CMPGT (U)		√								
CMPLT (U)		√								
EXT				√						
EXTU				√						
IDLE										√

Table 1. Opcode Functional Testing by Module (Continued)

Opcode/ Module Name	Basic	ALU	Mult	Bit	Sat	Cond	Circular	ALU 40	Mem	Not Tested
LDBU	√									
LDB	√									
LDHU	√									
LDH	√									
LDW	√									
LMBD				√						
MPYU			√							
MPYUS			√							
MPYSU			√							
MPYHL			√							
MPYLH			√							
MV	√									
MVC	√									
MVK	√									
MVKH	√									
MVKLH	√									
NEG		√								
NOP										√
NORM				√						
NOT		√								
OR		√								
SADD					√					
SAT					√					
SET				√						
SHL		√								
SHR		√								
SHRU		√								
SMPY					√					
SMYLH					√					

Table 1. Opcode Functional Testing by Module (Continued)

Opcode/ Module Name	Basic	ALU	Mult	Bit	Sat	Cond	Circular	ALU 40	Mem	Not Tested
SMPYHL					√					
SSHL					√					
SSUB					√					
STBU	√									
STB	√									
STHU	√									
STH	√									
STW	√									
SUB		√						√		
SUBU		√						√		
SUBAB		√								
SUBAH		√								
SUBAW		√								
SUBC		√								
SUB2		√								
XOR		√								
ZERO	√									

3.4 Program Options

The file OPTIONS.H is an include file containing user selectable option variables. All changes to these variables must be made prior to compiling and linking the program by editing OPTIONS.H with any standard text editor. The following is a description of the program options, and the corresponding variable settings:

Calling routine lockout option:

variable name: LOCKOUT
default value: 0
module affected: CHK6X

If the self-check fails, the program will either return the error code to the calling routine, or enter an endless loop to lockout the calling routine. The lockout option is selected by setting LOCKOUT = 1 in the options.h file. Only a system reset will regain control over a failed device when using the LOCKOUT option. If the self-check passes, the program will always return to the calling routine irrespective of how LOCKOUT is set in the options.h file.

External RAM test options:

variable name: RAM_EXT1
variable name: RAM_EXT2
variable name: RAM_EXT3
default value: 0module affected: MEM (.ASM)

Provisions have been made in the code to check the three external memory blocks. To check any one of the external memory blocks, set RAM_EXT1 = 1, RAM_EXT2 = 1, and RAM_EXT3 = 1, respectively. Block start address and the length is defined in the module (mem.asm). Be aware that the RAM test is destructive. Any code or data residing in the external memory blocks will be overwritten.

The memory test does not modify the external memory interface control registers.

Therefore, the default values are chosen (ie., 32-bit asynchronous memory with maximum setup, strobe and hold times).

Disabling a particular test or tests:

variable name: TEST_xx
default value: 1
module affected: CHK6X

You can disable either one or any number of test routines within the self-check code. To disable, you must specify it in the options.h file before compiling and running the program. The list of routines and their corresponding variables in the options.h file is as follows:

Table 2. List of TEST DISABLE Options

Option	Function
TEST_BA	to disable basic.asm, set it to 0
TEST_AL	to disable “ALU“ test routine, set it to 0
TEST_MU	to disable “MULT” test routine, set it to 0
TEST_BI	to disable “BIT” test routine, set it to 0
TEST_SA	to disable “SAT” test routine, set it to 0
TEST_CO	to disable “COND” test routine, set it to 0
TEST_CI	to disable “CIR” test routine, set it to 0
TEST_A4	to disable “ALU40” test routine, set it to 0
TEST_ME	to disable “MEM” test routine, set it to 0

4 Interpreting Error Codes

Table 3 lists all possible error codes that the self-check can return. It also gives the name of the module that generates the error code. It is important to note that the code descriptions identify only potential causes of the error. They should not be taken as absolute. Any number of actual malfunctions could generate a particular error code. For example a bad memory location would cause every test using it to fail.

A tradeoff exists between code length and the number of possible error codes. A large number of codes provides more detailed error information to the user, but increases the length of the code. In this program, the number of error codes has been kept moderate, and to emphasize what was stated previously, the self-check program aborts execution at the first such error code that is generated. This program design is based on the belief that any self-check error brings into question the reliability of the device, and therefore use of the failed C6x device will be discontinued regardless of the type of error (or number of errors) that may be present.

Table 3. Error Codes

Code	Description	Module
11h	LOAD instruction error [LD(B/BU)(H/HU)(W)]	BASIC
12h	STORE instruction error [ST(B/BU)(H/HU)(W)]	BASIC
13h	MV and MVC instructions error	BASIC
14h	ZERO instruction error	BASIC
15h	MVK, MVKH, MVKLH instructions error	BASIC
21h	Shift and CMP instructions error	ALU
22h	Logical instructions error (OR, XOR, AND etc)	ALU
23h	Addition instructions error (ADDU, ADDK, ADD2 etc)	ALU
24h	Subtraction instructions error (SUB, SUBA etc)	ALU
25h	SUBC instruction error	ALU
26h	SUB2 instruction error	ALU
31h	MPY instruction error	MULT
32h	MPYH, MPYHUS instructions error	MULT
33h	MPYHU and MPYHSU instructions error	MULT
34h	MPYHL instruction error	MULT
35h	MPYLH instruction error	MULT
41h	CLR instruction error	BIT
42h	EXT instruction error [EXT(U)]	BIT
43h	LMBD instruction error	BIT
44h	NORM instruction error	BIT
45h	SET instruction error	BIT
51h	SSHL instruction error	SAT
52h	SADD instruction error	SAT
53h	SAT instruction error	SAT
54h	SSUB instruction error	SAT
55h	SMPY(L)(H) instructions error	SAT
56h	SMPYHL instruction error	SAT
61h	B instruction [Conditional/Unconditional] error	COND
62h	Conditional Instructions error	COND
71h	ADDAB instruction error	CIRCULAR
72h	ADDAH instruction error	CIRCULAR
73h	ADDAW instruction error	CIRCULAR
74h	CIRCULAR BUFFER AUTHENTICITY error (LDW)	CIRCULAR
81h	ADDU instruction (for 40 bit) error	ALU40
82h	CMPEQ instruction (for 40 bit) error	ALU40
83h	SUBU instruction (for 40 bit) error	ALU40
91h	IDRAM SANITY CHECK ERROR (TEST PATTERN 1)	MEM
92h	IDRAM SANITY CHECK ERROR (TEST PATTERN 2)	MEM
93h	External Memory Block 1 Sanity Check Error	MEM
94h	External Memory Block 2 Sanity Check Error	MEM
95h	External Memory Block 3 Sanity Check Error	MEM

4.1 The All Tests Passed Code

In order to identify any potential malfunctions, the last action of the main control shell CHK6X before completing execution of the self-check is to call the subroutine PASS6X, which resides in the file CHK6X.ASM. PASS6X loads the register A4 with the number 0ffh and returns it as a confirmation that all test modules have passed.

NOTE: Failure to obtain this code upon completion of the self-check program indicates that some error is present.

5 Miscellaneous Program Information

1. In the module ALU, the numbers used in the Logical Operations Test were chosen to account for all possible pairs of 1s and 0s for each logical test (i.e., logical operation of 0 with 0, 0 with 1, 1 with 0, and 1 with 1).
2. The program may be single stepped through if desired.
3. This self-check program uses the default values for the external memory interface control registers. Therefore, the optional external memory test may be invalid for a device supporting different memory types.
4. Future TMS320C6X devices might have a different memory configuration (memory map). Therefore, the start address and the length of the external memory blocks as well as the internal data memory can be specified in the options.h file and is not hard coded.
5. When running the code on a TMS320C6x target system, confirm the memory map it is configured to and change the linked command file (chk6x.cmd) accordingly. The linked command file has both memory maps so one has to be commented out before using the other one.

6 Technical Support

Technical support may be obtained from the Texas Instruments DSP Hotline:

Telephone: (281) 274-2320

Email: dsph@msg.ti.com

World Wide Web Page: *http://www/ti/com/sc/docs/dsps/expsys.htm*

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.