Application Report
SPRA726 - April 2001

TMS320C55x Self-check Program: Version 1.0

Caroline L. Rhodes — Technical Staff – DSP Applications

ABSTRACT

The TMS320C55x™ (C55x™) self-check program verifies the operation of DSP devices in the C55x generation 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 possible, this program is not capable of detecting all potential faults. This application report describes the self-check program and the methodology used to check C55x instructions.

Contents

List of Figures

List of Tables

1 Introduction

The TMS320C55x (C55x) self-check program verifies the operation of DSP devices in the C55x generation 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 possible, this program is not capable of detecting all potential faults. This application report describes the self-check program and the methodology used to check C55x instructions. The instructions can be arranged into five major categories:

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

1.1 Scope of the Self-check

The C-callable program verifies the operation of most C55x instructions. Several modules are included to test features of the DSP such as addressing modes, arithmetic operators, bit manipulation, and register utilization. All data paths and functional units are tested at least once, but not with every instruction mode. Several bits of the CPU status registers ST0, ST1, ST2, and ST3 are tested including TC1, TC2, SXMD, RDM, and SATA. System control registers ICR and ISTR are not tested. Memory block tests are included and can be configured for a particular system.

1.2 System Requirements

To use the code for the self-check program, you will need the following tools:

- TMS320C55x target board
- Debugger (e.g. Code Composer Studio v1.20 or later)
- Emulator (e.g. XDS510)
- Code Generation Tools for C55x

The code has been tested using C5000 PC Code Composer Studio Code Generation Tools, version 1.70.

2 Program Files

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

ALU.ASM	Arithmetic operations module
BASIC.ASM	Basic operations module including load and store instructions
BIT.ASM	Bit manipulation module
CHK55x.ASM	Main program control shell
CIRC.ASM	Circular addressing module
COND.ASM	Branch and basic conditional instructions module
CONTROL.ASM	CPU control register manipulation module
FUNC.ASM	Module for complex instructions that utilize parallelism
LOGIC.ASM	Multi-bit operations module
MEM.ASM	Memory check module
MULT.ASM	Multiplier operations module
VECTORS.ASM	Interrupt Vector Table
USER.C	Sample C program that calls the self-check program
CHK55x.CMD	Sample linker command file for use with USER.C sample program
CHK55xCCS.CMD	Sample Code Composer Studio linker command file
OPTIONS.H	Include file containing user selectable options
MAKE.BAT	DOS batch file to assemble and link the code from a command prompt

2.1 How to Call the Self-check Program

The self-check program is designed to be callable from C or assembly. The files USER.C and CHK55x.CMD (or CHK55xCCS.CMD) provide an example of calling the self-check from a C program. If the self-check passes, it will use register T0 to return a pass code to the calling routine. If the self-check fails, it will either use register T0 to return an error code to the calling routine or lock out the calling routine (refer to section 3.4). The C compiler uses register T0 by convention for return values.

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

```
errorcode = chk55x();
```

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

```
extern unsigned int chk55x();
```

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

```
B _chk55x
```

The leading underbar in _chk55x must be included. This is a standard C-language naming convention that is required to allow the routine to be called from C code.

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 using the TMS320C55x Code Generation Tools. It assumes that the self-check program is being called from the C program USER.C, and uses the linker command file CHK55X.CMD. The batch file requires that all program files be in the current directory. The TMS320C55x Code Generation Tools must also be in the current directory or listed in the DOS path. The MAKE.BAT batch file is as follows:

```
cl55 –gs –als vectors.asm
cl55 –gs –als chk55x.asm
cl55 –gs –als basic.asm
cl55 –gs –als control.asm
cl55 –gs –als logic.asm
cl55 –gs –als alu.asm
cl55 –gs –als mult.asm
cl55 –gs –als func.asm
cl55 –gs –als bit.asm
cl55 –gs –als cond.asm
cl55 –gs –als circ.asm
cl55 –gs –als mem.asm
cl55 –gs –als user.c
cl55 –z chk55x.cmd
```

The cl55 command invokes the assembler, compiler, or the linker. Files with the .ASM extension are processed by the assembler to create COFF objects. Files with the .C extension are run through the compiler to create assembly code. The following options were used in the batch file for compiling and assembling source code:

`–g`	Enables src-level debugging

Assembler only option:

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

Compiler only option:

`–s`	Interlist C and assembly source code

To run the linker, the cl55 command must be accompanied by the –z option and a linker command file with the .CMD extension. The command file specifies the linker objects and memory mapping at a minimum. Contents of the linker command file are described below:

`xxxxx.obj`	Object filenames to be linked
`–o chk55x.out`	Specifies the output filename
`–m chk55x.map`	Specifies the map filename
`–c`	Indicates that variables will be autoinitialized at runtime
`–stack 1000`	Specifies stack size
`–sysstack 1000`	Specifies system stack size
`–l C:\ti\c5500\cgtools\lib\rts55.lib`	Links in the appropriate run-time support library included with the C compiler utility
`MEMORY{ }`	Sets up the memory map
`SECTIONS{ }`	Defines section locations

The USER.C and CHK55X.CMD files should be replaced or modified for your specific application. If you change the filenames, you will need to update the batch file MAKE.BAT, or create a new batch file similar to the one shown above. When calling the self-check from assembly, you will need to remove the options in the linker command file for initializing the C environment.

You can also build the self-check program in Code Composer Studio. The first step is to create a new project and add the C, assembly, and header files described in section 2. Then include the linker command file CHK55xCCS.CMD and the run time support library rts55.lib. CHK55xCCS.CMD only specifies the MEMORY{ } and SECTIONS{ } definitions, so you must select linker options using the appropriate Code Composer Studio menus. Compiler and assembler options are selected in the same manner. Code Composer Studio will save the project in a file with the MAK extension. When you are finished creating your project, you can build and download the output file to your target board. See Code Composer Studio User's Guide for more details. Again, the USER.C and CHK55xCCS.CMD files can be modified or replaced.

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 memory blocks and all optionally tested memory. Programs should not be loaded, nor any data sections initialized in tested RAM locations prior to running the self-check. All data loaded in the tested sections 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 RAM tests on the used sections of memory.
- Link and run all code from internal RAM, and disable the RAM tests on the used sections of memory.
- 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 two linker methods mentioned above before bootloading your code and initializing your data. This will require you to modify your bootloader.

3 Operations Tested by the Self-check Program

The self-check is used to demonstrate proper operation of most TMS320C55x instructions and operating modes. The following types of instructions and modes are tested:

- Arithmetic operations, including complex instructions such as LMS and FIRSADD
- Bit manipulation operations
- Extended auxiliary register operations
- Logical operations
- Move operations, including CPU register and stack manipulations
- Program control operations
- Linear and circular addressing
- Memory-mapped register access, using the mmap qualifier

Several instructions are tested under multiple conditions by varying the number and type of operands. This is intended to rigorously test the memory interface, instruction buffer, program flow, address-data flow, and data computation units of the CPU. In addition, internal and external RAM can optionally be tested. Since memory sizes and locations can vary amongst the different TMS320C55x devices, these parameters can be changed prior to running the self-check.

Exceptions to the self-check include testing peripherals, 40-bit operations, coprocessor hardware invocations, forms of parallelism not explicitly identified in the instruction set, and IDLE and NOP conditions. If peripheral testing is desired, a module could be created similar to those included in the self-check program. This module must be specific to a particular device since the on-chip peripherals vary for each member of the C55x generation. Also, no interrupts are tested nor will any maskable interrupts be serviced during self-check execution (interrupts are disabled). Finally, although the C55x device supports the TMS320C54x™ instruction set, the self-check will only test those instructions specific to the C55x.

3.1 How the Self-check Program Works

The self-check program tests the CPU by exercising several functional units and then verifying the correct result. When an incorrect result occurs, the program loads an unsigned error code into register T0, and then returns to the assembly calling routine, _chk55x. Once in this routine, the program can either return the error code to the calling C program (USER.C) or lock itself out from further execution (as specified in OPTIONS.H).

Internal and external data RAM is tested using a checkerboard pattern. The test first sets every other bit in each memory location, and then reads the data back to determine if an error occurred. If no errors are detected, then every bit in memory is inverted from a 0 to a 1 or vice versa. Again, the same check is performed to determine if the bits can be set or cleared. The memory tests can be configured to specify starting addresses and block sizes. See section 3.4 for more details on changing these parameters and enabling or disabling memory tests.

CHK55x.ASM functions as the main control shell for the program and calls each test module passing it a pointer to a table of known values. In most cases, the table arguments are used as input operands to the instructions. Since the arguments are known, expected results can be determined and compared with the actual results. Immediate values are also used for some tests, rather than the table. Again, the arguments are known, so the results can be checked for errors. Refer to the following section of code for an example of how a specific instruction is tested:

TMS320C54x is a trademark of Texas Instruments.

```
;*********************************************************
;*  Arguments passed to routine                          *
;*                                                       *
;*  AR2 –> temp_0                                        *
;*  temp_0(0) = AAAAh                                    *
;*  temp_0(1) = 5452h                                    *
;*  temp_0(2) = 5200h                                    *
;*  temp_0(3) = 5555h                                    *
;*  temp_0(4) = 3333h                                    *
;*  temp_0(5) = 2222h                                    *
;*********************************************************


; test load to auxiliary register using indirect addressing

      MOV   *AR2,AR5          ; AR5 = temp_0(0) = AAAAh
      SUB   *AR2,AR5,AC0      ; AC0 = *AR2 – AR5
      BCC   ERR1, AC0!=#0     ; ERROR, if AC0 != 0
      B     end_basic

ERR1:
      MOV   #11h,T0           ; LOAD FAIL CODE
      B     end_basic         ; Accumulator, Auxiliary, or Temporary
                              ; Register Load
end_basic:  RET
```

This example tests the MOV instruction for loading a value into an auxiliary register. In the first instruction, the test value is loaded into AR5 using indirect addressing. Since AR2 is pointing to the first value in the table temp_0, AAAAh is loaded into AR5. To be sure that the correct value was loaded, AAAAh is subtracted from AR5 and the result is stored in AC0. If AC0 is not 0, then the test failed and an error code is placed in T0. When the test is complete, the module returns to the main control shell.

3.2 Preserved Registers

Since the self-check is C-callable, it must save the context of certain registers upon entry and restore them before returning to the main calling routine. The registers that are preserved on the stack are AR5, AR6, AR7, T2, and T3. Several other status registers are never modified by the self-check and, therefore, can be considered preserved. These registers include DBIER0–1, IER0–1, IFR0–1, IVPD, IVPH, ICR, and ISTR.

3.3 How the Program is Structured

The self-check code is organized into ten modules each containing test routines for similar functional instructions. For example, ALU.ASM includes tests for both the addition (ADD) and subtraction (SUB) instructions. Each module is called in a specific order by the main control shell found in CHK55x.ASM. If an error is found in a particular module, the program will return to the main control shell with an error code. At that point, the program returns to the calling routine with an error code loaded in T0 or locks itself out from further execution. If all tests pass, the next module is called. Figure 1 below illustrates the program flow in a C environment.

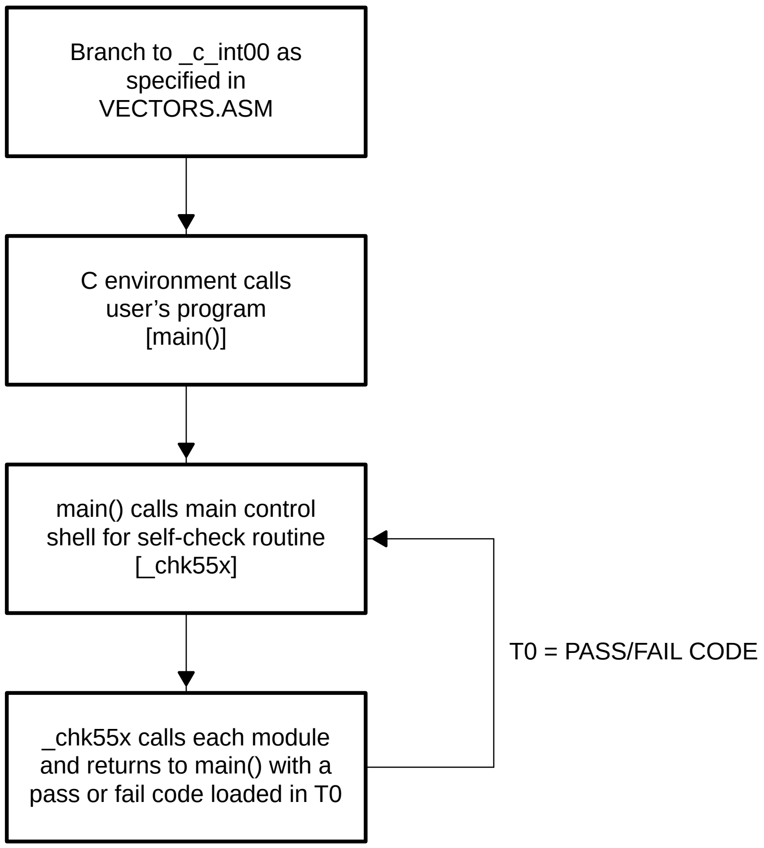
Branch to _c_int00 as specified in VECTORS.ASM
C environment calls user's program [main()]
main() calls main control shell for self-check routine [_chk55x]
_chk55x calls each module and returns to main() with a pass or fail code loaded in T0
T0 = PASS/FAIL CODE

Figure 1. Program Flowchart

In most cases, an instruction is tested before it is used (hence the significance of the module calling order in CHK55x.ASM). It is not possible to employ this method for all instructions since some are required for basic operation. Instructions that are frequently used throughout the self-check routine such as move (MOV) and conditional branch (BCC) are tested in the first module, BASIC.ASM. The following list indicates the order in which the modules are called:

- BASIC.ASM
- CONTROL.ASM
- LOGIC.ASM
- ALU.ASM
- MULT.ASM
- FUNC.ASM
- BIT.ASM
- COND.ASM
- CIRC.ASM
- MEM.ASM

Table 1 identifies which instructions are tested by the self-check routine and where their associated test routines can be found.

Table 1. Opcode Functional Testing by Module

Opcode/Module Name	Basic	Control	Logic	ALU	Mult	Func	Bit	Cond	Circ	Mem	Not Tested
AADD				✓							
ABDST						✓					
ABS				✓							
ADD				✓							
ADD::MOV											✓
ADDSUB				✓							
ADDSUB2CC				✓							
ADDSUBCC				✓							
ADDV					✓						
AMAR				✓							
AMAR::MAC					✓						
AMAR::MAS					✓						
AMAR::MPY					✓						
AMOV				✓							
AND			✓								
ASUB				✓							
B											✓
BAND							✓				
BCC	✓										
BCLR							✓				
BCNT			✓								
BFXPA							✓				
BFXTR							✓				
BNOT							✓				
BSET							✓				
BTST							✓				
BTSTCLR							✓				
BTSTNOT							✓				
BTSTP							✓				
BTSTSET							✓				

Table 1. Opcode Functional Testing by Module (Continued)

Opcode/Module Name	Basic	Control	Logic	ALU	Mult	Func	Bit	Cond	Circ	Mem	Not Tested
CALL											✓
CALLCC								✓			
CMP											✓
CMPAND				✓							
CMPOR				✓							
DELAY	✓										
DMAXDIFF						✓					
DMINDIFF						✓					
EXP						✓					
FIRSADD						✓					
FIRSSUB						✓					
IDLE											✓
INTR											✓
LMS						✓					
MAC					✓						
MAC::MAC					✓						
MAC::MPY					✓						
MACK					✓						
MACM					✓						
MACM::MOV					✓						
MACMK					✓						
MANT::NEXP						✓					
MAS					✓						
MAS::MAC					✓						
MAS::MAS					✓						
MAS::MPY					✓						
MASM					✓						
MASM::MOV					✓						
MAX						✓					
MAXDIFF						✓					

Table 1. Opcode Functional Testing by Module (Continued)

Opcode/Module Name	Basic	Control	Logic	ALU	Mult	Func	Bit	Cond	Circ	Mem	Not Tested
MIN						✓					
MINDIFF						✓					
MOV	✓	✓									
MOV::MOV				✓							
MPY					✓						
MPY::MPY					✓						
MPYK					✓						
MPYM					✓						
MPYM::MOV					✓						
MPYMK					✓						
NEG				✓							
NOP											✓
NOT			✓								
OR			✓								
POP		✓									
POPBOTH											✓
PSH		✓									
PSHBOTH											✓
RESET											✓
RET											✓
RETCC											✓
RETI											✓
ROL			✓								
ROR			✓								
ROUND				✓							
RPT								✓			
RPTADD								✓			
RPTB											✓
RPTBLOCAL								✓			
RPTCC											✓

Table 1. Opcode Functional Testing by Module (Continued)

Opcode/Module Name	Basic	Control	Logic	ALU	Mult	Func	Bit	Cond	Circ	Mem	Not Tested
RPTSUB								✓			
SAT				✓							
SFTCC			✓								
SFTL			✓								
SFTS			✓								
SFTSC			✓								
SQA					✓						
SQAM					✓						
SQDST						✓					
SQR					✓						
SQRM					✓						
SQS					✓						
SQSM					✓						
SUB				✓							
SUB::MOV											✓
SUBADD				✓							
SUBC				✓							
SWAP	✓										
SWAP4											✓
SWAPP	✓										
TRAP											✓
XCC								✓			
XCCPART											✓
XOR			✓								

3.4 Program Options

The OPTIONS.H file allows you to select the modules you would like to test and specify several other program features. All changes to these variables must be made prior to compiling and linking the program. The following sections describe the program options and corresponding variables.

3.4.1 Calling Routine Lockout Option

If the self-check fails, the program will return an error code to the calling routine or enter an endless loop as specified by the lockout option. When the lockout option is turned on, LOCKOUT = 1, only a system reset or a non-maskable interrupt will regain control over a failed device. If the self-check passes, the program will always return to the calling routine regardless of the value of LOCKOUT. The default value for LOCKOUT is 0, or disabled.

3.4.2 Memory Test Options

The MEM module can be configured to test specific blocks of internal and external memory. Nine memory tests have been included, and can be expanded if desired. Each test has an enable flag and variables to specify the starting address and block length. To enable the individual tests, set their associated flag, MEMBLKx (x refers to the specific test block), to 1. Setting the flag to 0 will disable the test. The user must also provide valid starting addresses (RAMSTx) and block lengths (RAMLNx) for the memory blocks tested. Failure to do so will produce erroneous results.

CAUTION:
Be aware that the test is destructive and will overwrite all program code and data in the memory sections you specify. The default options disable all memory tests.

3.4.3 Test Disable Options

Any of the ten test modules can be disabled by setting the appropriate flag in the OPTIONS.H file prior to compiling and linking the program. Table 2 describes the flag variables used for this feature. Each flag is checked in the CHK55X.ASM module before running the associated test. If a test is disabled, the self-check skips it and moves on to the next test.

Table 2. List of Test Disable Options

Option	Function
TEST_BA	To disable BASIC test, set equal to 0.
TEST_CT	To disable CONTROL test, set equal to 0.
TEST_LO	To disable LOGIC test, set equal to 0.
TEST_AL	To disable ALU test, set equal to 0.
TEST_MU	To disable MULT test, set equal to 0.
TEST_FN	To disable FUNC test, set equal to 0.
TEST_BI	To disable BIT test, set equal to 0.
TEST_CO	To disable COND test, set equal to 0.
TEST_CI	To disable CIRC test, set equal to 0.
TEST_ME	To enable MEM test, set equal to 1.

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. In addition, 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. Therefore, use of the failed C55x device will be discontinued regardless of the type of error (or number of errors) that may be present.

4.1 The All Tests Passed Code

If no errors are found in any of the modules, the self-check returns a pass code of 0FFh in register T0. This register is loaded after all tests have been completed, and just before returning to the calling program. See module CHK55X.ASM for more details.

Note: Failure to obtain the pass code, 0FFh, upon completion of the self-check indicates that an error is present.

Table 3. Error Codes

Code	Description	Module
11h	Load error [MOV Smem/K16 to Accumulator, Auxiliary, or Temporary Register] or Conditional branch error [BCC Lx, cond]	BASIC
12h	Move between registers error [MOV src, dst]	BASIC
13h	Swap register content error [SWAP Accumulator, Auxiliary, or Temporary Registers]	BASIC
14h	Store error [MOV Accumulator, Auxiliary, or Temporary Register to Smem]	BASIC
15h	Memory to Memory Move/Initialization error [MOV Smem/K Smem]	BASIC
21h	Specific CPU register load or store error	CONTROL
22h	Push to top of stack/Pop from top of stack error [PSH src, POP src]	CONTROL
31h	Logical instructions error [OR, XOR, AND, etc.]	LOGIC
32h	Shift instructions error	LOGIC
41h	Saturation or Round instruction error [SAT or ROUND]	ALU
42h	Negate or Absolute Value instruction error [NEG or ABS]	ALU
43h	Addition instruction error [ADD]	ALU
44h	Subtraction instruction error [SUB]	ALU
45h	Conditional Addition/Subtraction instruction error [ADDSUBCC]	ALU
46h	Register Comparison instruction error [CMP, CMPAND, CMPOR]	ALU

Table 3. Error Codes (Continued)

Code	Description	Module
51h	Multiply instruction error [MPY, SQR]	MULT
52h	Multiply and Accumulate instruction error [MAC, SQA, etc.]	MULT
53h	Multiply and Subtract instruction error [MAS, SQS, etc.]	MULT
54h	Dual Multiply and Accumulate/Subtract instruction error	MULT
55h	Implied Parallel instruction error [MPYM || MOV, MACM || MOV, MASM || MOV]	MULT
61h	Absolute Distance, Maximum/Minimum instruction error [ABDST, MAX, MIN, etc.]	FUNC
62h	Normalization instruction error [MANT || NEXP, EXP]	FUNC
63h	Square Distance, LMS, and FIRS instruction error	FUNC
71h	Bit Compare, Extract, and Expand instruction error [BAND, BFXTR, BFXPA]	BIT
72h	Memory Bit Test/Set/Clear/Not instruction error [BTST, BSET, BCLR, etc.]	BIT
73h	Register Bit Test/Set/Clear/Not instruction error	BIT
74h	Status Bit Set/Clear instruction error	BIT
81h	Conditional Branch, Call, or Execute instruction error [BCC, CALLCC, XCC]	COND
82h	Repeat Single instruction error [RPT]	COND
83h	Repeat Block instruction error [RPTBLOCAL]	COND
91h	Circular Addressing error	CIRC
A1h	Memory error for MEMBLK1	MEM
A2h	Memory error for MEMBLK2, MEMBLK3, MEMBLK4, and MEMBLK5	MEM
A3h	Memory error for MEMBLK6, MEMBLK7, MEMBLK8, and MEMBLK9	MEM

5 Miscellaneous Program Information

- The program may be single-stepped through, if desired.
- Verify the memory map of your TMS320C55x target system before loading the program. Changes to the memory map can be made in the linker command file.
- The ARMS mode bit (bit 15 of ST2) is cleared to 0 by the self-check program. This bit affects indirect addressing functionality.
- When using the memory check found in module MEM, be sure to configure each test for the appropriate starting address and block length. All parameters are set in the OPTIONS.H file prior to compiling and linking the program.
- Although maskable interrupts are disabled, interrupt flag register bits will still be set if an interrupt occurs. Any pending interrupts will be serviced once the self-check returns to the calling program and the INTM bit is cleared.

6 Technical Support

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

Telephone: (972) 644-5580

Email: dsph@msg.ti.com

World Wide Web Page: http://www.ti.com