Application Report
SPRA599A - November 2000

# DSP/BIOS and TMS320C54x Extended Addressing

*Scott Gary, Don Gillespie, Changlin Chen* *DSP Kernel Technology Group*

**ABSTRACT**

DSP/BIOS provides basic runtime services including real-time analysis functions for instrumenting an application, clock and periodic functions, I/O modules, and a preemptive scheduler. DSP/BIOS™ for the TMS320C54x™ DSP generation was originally developed for the 16-bit addressing model of the early C54x™ devices. Program accesses in this 'near' model were limited to a single 64K word program page. Newer C54x devices incorporate 'far' extended addressing modes, and DSP/BIOS has been modified to work in this environment. This application report describes how to configure and use DSP/BIOS in extended addressing applications.

**Contents**



**List of Figures**

# 1 Background Information

DSP/BIOS implements the DSP/BIOS API, and provides both an assembly language (macro) interface, and a C-callable interface. DSP/BIOS functions can be invoked from software interrupts, background threads, and from interrupt service routines. DSP/BIOS itself is written entirely in hand-optimized assembly language. Wrappers are provided to implement the C-language API. Additionally, function glue stubs, which are used by the IDL and CLK modules of DSP/BIOS, are provided when DSP/BIOS needs to call C-language user functions.

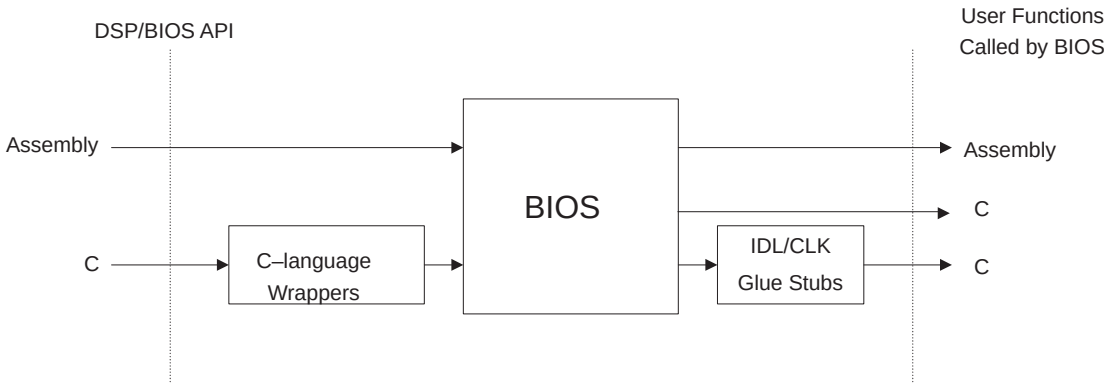


**Figure 1. DSP/BIOS Interfaces**

## 1.1 Extended Program Memory

C54x extended addressing was introduced starting with the C548 to support programs larger that 64K words. A new 7-bit program counter extension register (XPC) is combined with the 16-bit program counter to support up to 8192K of program memory. Extended addressing is only for program space; data space is not extended. Extra instructions are provided for unconditionally branching to, calling, and returning from 23-bit ‘far’ addresses.

The mapping of extended program memory depends upon the state of the overlay (OVLY) bit in the processor mode status register (PMST). If OVLY is zero, on-chip RAM is not mapped into program space (it is only mapped into data space), and there can be up to 128 unique 64K-word program pages, (corresponding to XPC=0 to XPC=127), as shown in Figure 2.

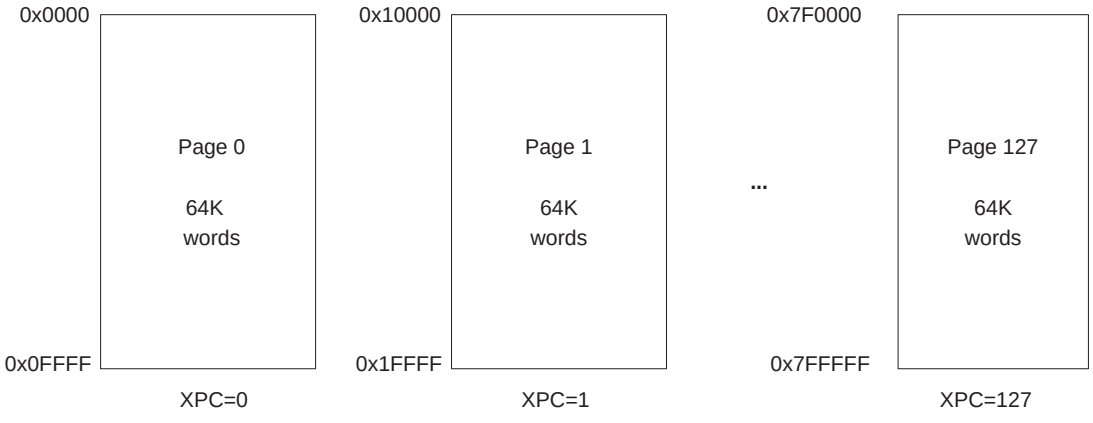


**Figure 2. Extended Program Memory when OVLY=0**

If OVLY=1 on-chip RAM is mapped into both program and data space, resulting in one common (i.e., shared) overlay page, and up to 128 unique 32K-word pages, (corresponding to XPC=0 to 127). The common (overlay) page is visible within the lower 32K-word space of each 64K-word program page.

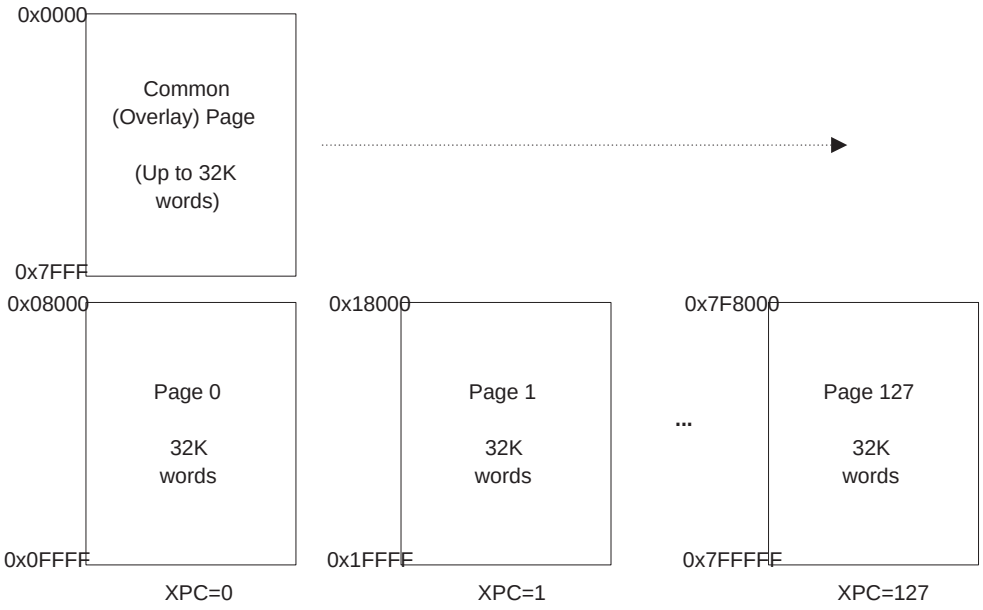


**Figure 3. Extended Program Memory when OVLY=1**

If on-chip ROM is enabled by setting MP/$\overline{MC}$=0 in the PMST, the ROM is only visible on the first extended memory page (i.e., XPC=0).

## 1.2 Interrupts

A critical consideration with extended addressing is interrupt handling (see Reference 1). When an interrupt is acknowledged only the 16-bit program counter is automatically saved on the stack; XPC must be saved by the interrupt service routine as needed. Also, the current value of the XPC register is used to compute the location of the interrupt vector. This means that the vector table must reside in the current 64K program page whenever an interrupt can occur. In the OVLY=0 case this means that a copy of the interrupt vector table is needed on *each* page where interrupts are enabled during code execution. In the OVLY=1 case only a single copy of the vector table is needed if the table is relocated from the default location of 0xFF80 to somewhere in the common overlay page (e.g., 0x7F80).

**Note:** The remainder of this note will focus on the most common usage of extended addressing, which has OVLY=1.

# 2 Using DSP/BIOS in Far Model Applications

This section describes conceptually how DSP/BIOS works in far model applications. Some assumptions are defined, followed by an illustration of a direct DSP/BIOS API call, and an illustration of preemption triggered by a hardware interrupt. In the next section a specific programming example is given.

Separate “far” versions of the C-wrappers are now provided. A global configuration parameter CALLMODEL is provided and the Configuration Tool takes care of linking the appropriate function and glue stubs, and generating the appropriate interrupt stubs. A “near” CALLMODEL is provided for users who do not wish to use extended addressing.

The sections that make up DSP/BIOS and its associated C-wrappers are .sysinit, .bios and .bios:.norptb. The .sysinit and .bios sections must be placed together but may be put on any page, including the overlay page, if desired. However, the .bios:.norptb section must be put on the overlay page to avoid the RPTB hazard.

The RPTB harzard is where a RPTB instruction sets the BRAF (in ST1) and then within the repear block there is a context switch to another thread. That other thread then happens to execute an instruction on a different memory page but with the 16-bit program address matching the value in the repeat ending address (REA) register. The result would be an unintended block repeat.

The current DSP/BIOS Configuration Tool provides an easy way for DSP/BIOS users to move the .sysinit and .bios sections around, but it does not automatically put the .bios:.norptb section on the overlay page. Therefore, DSP/BIOS users have to create a linker command file (cmd) to allocate the .bios:.norptb section into the overlay page. For instance, to allocate the .bios:.norptb section into the overlay page for program progx, the user can create a cmd file named myprogx.cmd with the contents shown below to link and build the executable.

```
-l progxcfg.cmd

SECTIONS {

      .bios:.norptb:  {}  > IPROG  PAGE 0

}
```

where IPROG is a defined MEMORY section located inside the overlay page.

Nevertheless, the future releases of DSP/BIOS Configuration Tool will automatically put the .bios:.norptb section on the overlay page so users can avoid that extra work.

**Note**: To avoid the RPTB hazard, TMS320C54x far model programs using the current version of DSP/BIOS need the patch in the ftp path shown below:

ftp://ftp.ti.com/pub/cs/c54x/bios/12245_ecs/

It is important to note that the C interface is far-callable because of the wrapper functions, whereas the assembly interface is not. The assembly interface is near-callable only, so assembly calls into DSP/BIOS must be made from the same page, unless DSP/BIOS is placed on the overlay page. DSP/BIOS may not be split across multiple pages. The interrupt vector table must always be placed on the overlay page, but the ISRs may be placed on any page. However, the ISR stubs, i.e., HWI_enter and HWI_exit, must be placed on the overlay page.

## 2.1 Requirements

- OVLY=1, mapping on-chip RAM into both data and program space.
- The interrupt vector table is relocated to the overlay page.
- ISRs that do not use HWI_exit must do far returns (i.e., frete or freted).
- The DSP/BIOS C-language API can be called from any page.
- Assembly API calls must be made only from the same page as DSP/BIOS, unless DSP/BIOS is placed on the overlay page.

## 2.2 Illustration 1: DSP/BIOS API Call

In this example, a high priority software interrupt (SWI1) is posted while executing on far page 2 (XPC=2). SWI1 is configured to run on far page 1 (XPC=1), and OVLY is 1. DSP/BIOS is on page 3 (XPC=3). The calling sequence:

1. SWI_post() is invoked, which does a far call to the SWI_post() wrapper. XPC is set to 3 by the far call.
2. The C wrapper posts the SWI, detects that a higher priority SWI is now ready, and does a near branch to the DSP/BIOS scheduler.
3. The scheduler saves the current context and does a far call to SWI1's execute function, setting XPC to 1.
4. When SWI1 runs to completion, it does a far return to the scheduler, setting XPC to 3.
5. The scheduler restores the context for SWI2 and then does a near return to the C wrapper.
6. The C wrapper does a far return, which sets XPC back to 2, and resumes SWI2.

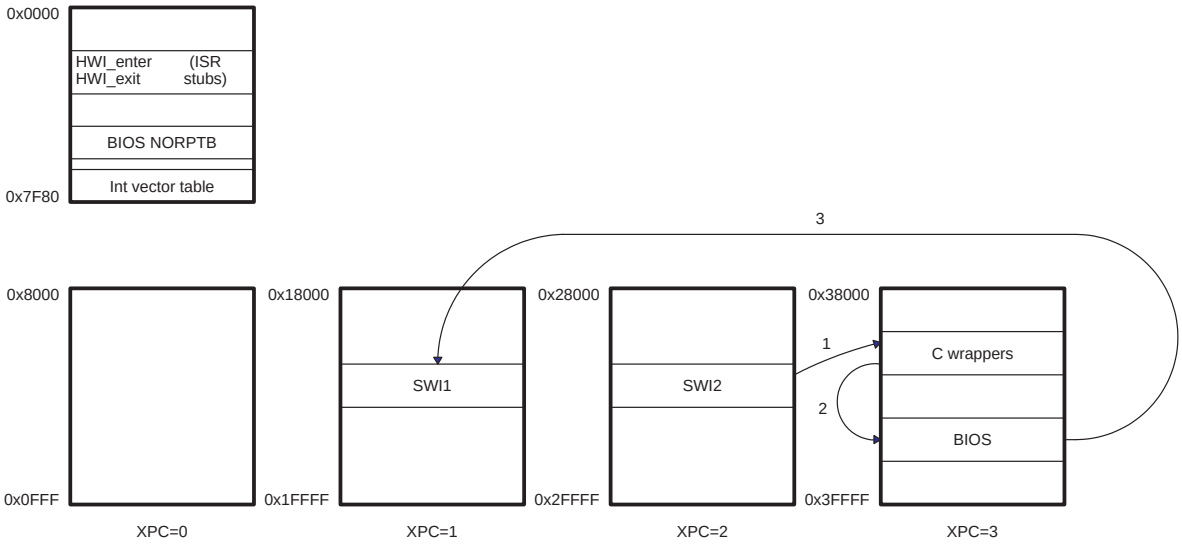


**Figure 4. Posting of SWI, DSP/BIOS**

## 2.3 Illustration 2: Hardware Interrupt Triggers Software Interrupt Preemption

In this example, two software interrupts are configured. SWI1 is currently running on far page 1, and a higher priority software interrupt (SWI2) is configured to run on far page 2. SWI2 is posted by an interrupt and preempts SWI1. DSP/BIOS is on page 3 (XPC= 3). The preemption sequence is as follows:

1. The interrupt occurs. The CPU stores the current (16-bit) PC on the stack, and fetches the instruction at the vector address.
2. The vector stub code pushes the current value of the XPC (i.e., 1) on the stack, and does a far branch to the configured ISR, which sets XPC=0.
3. The ISR calls HWI_enter, posts SWI2, and then calls HWI_exit. HWI_exit determines that a higher priority SWI has been readied, and invokes the DSP/BIOS scheduler, setting XPC=3.
4. The scheduler saves the current context and runs SWI2 which sets XPC=2.
5. When SWI2 runs to completion it does a far return to the scheduler.
6. The scheduler restores the context for SWI1 and then does a far return, which resumes SWI1 execution.

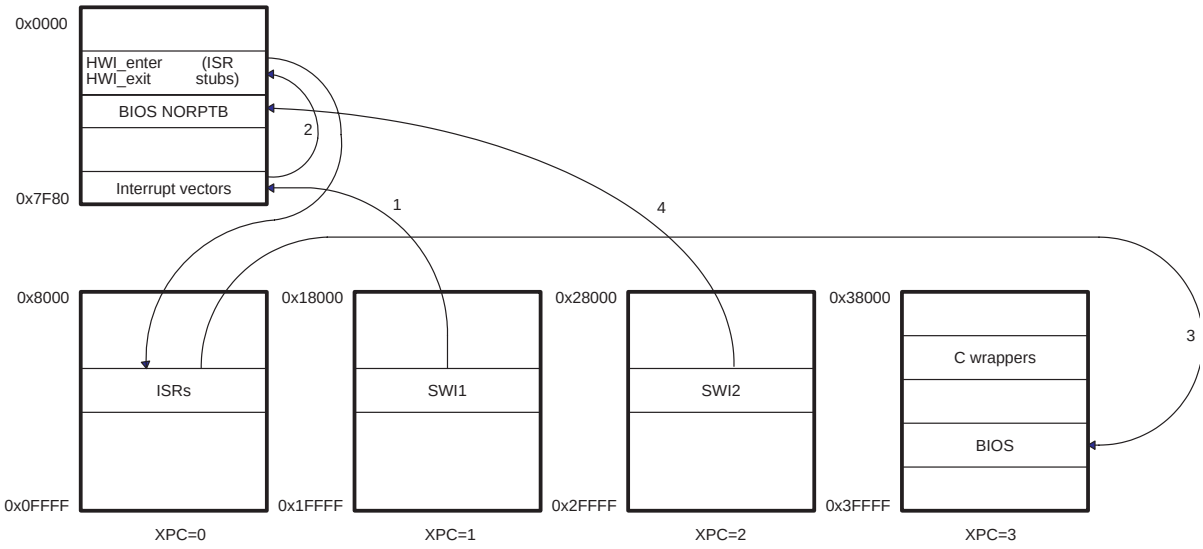


**Figure 5. SWI Preemption Triggered by Hardware Interrupt**

# 3 Programming Example

This section presents a simple example of configuring, building, and running a far model application that uses DSP/BIOS. Two software interrupts are used, and each SWI's execute function resides on a different extended program page. DSP/BIOS is on page 1. **SWI0** is posted from main(); when **SWI0** runs it prints a message and posts a higher priority SWI, **SWI1**. **SWI1** preempts **SWI0**, prints a message, and exits. **SWI0** is reentered, prints another message, and then exits. The source for the application is:

```
/*
 *  ======== fartest.c ========
 */
#include <std.h>
#include <swi.h>
#include <log.h>
extern SWI_Obj SWI0;
extern SWI_Obj SWI1;
extern LOG_Obj trace;
#pragma CODE_SECTION(page0fxn, ".farpage0")
#pragma CODE_SECTION(page1fxn, ".farpage1")
/*
 *  ======== main ========
 */
Void main(int argc, char *argv[])
{
    SWI_post(&SWI0);
}
/*
 *  ======== page0fxn ========
 *
 */
void page0fxn (void)
{
    LOG_printf(&trace,"page0fxn running");
    SWI_post(&SWI1);
    LOG_printf(&trace,"page0fxn exiting");
}
/*
 *  ======== page1fxn ========
 *
 */
Void page1fxn (Void)
{
    LOG_printf(&trace,"page1fxn running");
}
```

## 3.1 Configure the Application

The following steps are used to configure the application:

- Open a new configuration
- Configure the trace buffer
- Configure the software interrupts
- Turn on the OVLY bit
- Specify far CALLMODEL
- Relocate the interrupt vector table to the overlay page
- Relocate DSP/BIOS to page 1
- Configure extended program memory segments
- Save the configuration
- Add files to a project

**Open a new configuration**

1. Start a new Code Composer Studio session. Select File→ New→ DSP/BIOS Config
2. Select one of the configuration seeds. In this example we will use the sd54.cdb seed, for the Spectrum Digital 548 EVM.

**Configure the trace buffer**

3. Right click on the Event Log Manager and select “Insert Log”. A new log, LOG0, appears.
4. Right click on LOG0, select “Rename”, and name the log “trace”.

**Configure the software interrupts**

5. Right click on the Software Interrupt Manager, and select Insert SWI, to create SWI0.
6. Right click on SWI0, and select Properties to bring up the properties page.
7. For the function name type “_page0fxn”, and then click OK to close the page.
8. Repeat the previous steps to create a second SWI, SWI1, with a function name of “_page1fxn”.
9. Once both SWIs are created, left-click on the Software Interrupt Manager to get the SWI objects displayed by priority in the right pane of the Configuration Tool display.
10. In the right pane, left-click on SWI1, and drag it to a higher priority than SWI0.
11. To turn on the OVLY bit, right click on Global Settings icon, and select Properties. In the entry for PMST(6–0), ensure that bit 5 is set to 1. For example, with MP=1 and all other bits off, the value is 0x40. Turning on OVLY will change the value to 0x60.

12. To specify the far CALLMODEL, select “far” from the Function Call Model drop down list. Then click OK to close the global settings property page.

You will use the Memory Section Manager to relocate the interrupt vector table to the overlay page, to configure extended program memory segments and to relocate DSP/BIOS. For the Spectrum Digital 548 EVM, the interrupt vector table is already relocated by default, and two segments EPROG0 and EPROG1 are already defined, so you can skip ahead to the next task.

**Relocate the interrupt vector table**

1. Double-click on the Memory Section Manager
2. Right-click on the VECT object and select Properties
3. In the properties page, specify a new base address on the overlay page (e.g., 0x7f80)
4. Click OK to accept the new address. If you get an error that there is an overlap with another segment you will need to pick another base address for VECT, or adjust the range of the overlapping MEM object.

**Define your own extended program segments**

5. Right click on the Memory Section Manager.
6. Add new MEM objects by selecting Insert MEM.
7. Rename the new objects to EPROG0 and EPROG1, by right-clicking and selecting Rename.
8. For each of the new MEM objects right-click on the object, select Properties.
9. Enter the appropriate base address (e.g., 0x18000), the len length (e.g., 0x8000), and select the space as code.
10. Right-click on Memory Sector Manager; select Properties.
11. Go the the BIOS Code section (.bios) and Startup Code section (.sysinit). Set both to EPROG1.

**Save the configuration**

12. Use File→ Save As to save the configuration in a new directory, using the file name “fartest.cdb”.
13. Create a new project by selecting Project→ New, navigating to the new directory where you saved fartest.cdb, and specifying the new project name as “fartest”.
14. Type in the example code above and save the file as fartest.c. Use Project→ Add files to Project and add the following files: fartest.c, fartest.cdb, fartestcfg.s54, and fartestcfg.cmd.
15. Configure the compiler and assembler to generate far model code by selecting Project → Options, and adding the flags “–v548 –mf” (these flags specify to generate far model code) to both the compiler and assembler build options.
16. Tell the linker to generate a map file by adding the following to the linker options: –m fartest.map.
17. Click OK to close the build option pages.

## 3.2 Edit the Linker Command File

There are different ways to load program code to the extended program pages. For example, if the code is broken into multiple source files, the following line could be included in the linker command file to load the executable code derived from a file page.c, to the EPROG0 memory segment:

page0text: { page.obj(.text)} > EPROG0 PAGE 0

In this example we put all code in a single source file, and will use a different method. The CODE_SECTION pragma is used to specify the destination sections for page0fxn() and page1fxn(). To get the linker to relocate these functions to the extended pages we need to add the following lines inside the SECTIONS specification at the end of the linker command file (fartestcfg.cmd):

.farpage0: {} > EPROG0 PAGE 0

.farpage1: {} > EPROG1 PAGE 0

As mentioned before, the .bios:.norptb section must be put on the overlay page to avoid the RPTB hazard.

The current DSP/BIOS Configuration Tool does not automatically put the .bios:.norptb section on the overlay page. Therefore, DSP/BIOS users have to allocate the .bios:.norptb section into the overlay page. For instance, to allocate the .bios:.norptb section into the overlay page for program fartest, the user can create a cmd file named myfartest.cmd with the contents shown below to link and build the executable.

```
-l fartestcfg.cmd

SECTIONS {

    .bios:.norptb:  {}  >  IPROG  PAGE 0

}
```

where IPROG is a defined MEMORY section located inside the overlay page.

Nevertheless, the future releases of the DSP/BIOS Configuration Tool will automatically put the .bios:.norptb section on the overlay page so users can avoid that extra work.

## 3.3 Build the Application

Build the application by selecting Project→ Rebuild All. Once the program is built you should look at the map file and verify that code will be loaded to the extended program pages. For example, in the memory configuration summary near the top of the file you should see that space has been used in EPROG0 and EPROG1. Also, in the section allocation map you should see that the .farpage0 and .farpage1 sections reside at the proper addresses, and include code from fartest.obj:

```
.farpage0   0    00008000    00000017
                 00008000    00000017     fartest.obj (.farpage0)
```

```
.farpage1   0    00018000    0000000c

                 00018000    0000000c     fartest.obj (.farpage1)
```

## 3.4 Enable Extended Addressing in Code Composer

To use extended addressing within Code Composer two configuration steps are needed: GEL_XMDef() has to be called to define the mapping register (i.e., XPC), and to define the beginning of mapped extended memory; and GEL_XMOn() has to be called to turn on extended memory mapping. These two commands can be included in the GEL Startup() function in the file init.gel. Example commands are shown below.

```
GEL_XMDef(0,0x1e,1,0x8000,0x7f);

GEL_XMOn();
```

In this example, the parameters for GEL_XMDef() are:

```
0       = program space mapping

0x1e    = address of XPC

1       = XPC resides in data space

0x8000 = beginning of mapped memory

0x7f    = bit mask indicating size of XPC
```

## 3.5 Load and Run the Application

Load the program to the target using File→ Load Program. Run the program for a few seconds and then halt. Select Tools→ DSP/BIOS→ Message Log to open a message log window. Right click on the window, select “Property Page”, and select “trace” from the log name drop down box. Click OK to close the property page. Right click on the log window and select “Refresh Window”. The following should be displayed:

```
0  page0fxn running

1  page1fxn running

2  page0fxn exiting
```

# 4 References

1. *Interrupt Handling Using Extended Addressing of the TMS320C54x Family*, Application Report SPRA492A, July 1999.
2. *TMS320C54x DSP CPU and Peripherals, Reference Set Volume 1*, SPRU131, June 1998.

## 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.