Application Report
SPRA754 - May 2001

Debugging Shared Memory Systems

Jeff Hunter — Software Development Systems/Emulation Team

ABSTRACT

Multiple cores on a single processor often share a common block of memory. This application report discusses how to setup the Code Composer Studio™ Integrated Development Environment (IDE) to debug shared memory systems. The configuration and attributes of the shared memory must be defined in the Code Composer Studio memory map.

Once Code Composer Studio is properly set up, several strategies are available for debugging shared memory systems. Users can choose to debug a single core at a time or multiple cores simultaneously.

Code Composer Studio's shared memory options enable users to control whether cores with access to shared memory are halted when writing to shared memory, or stepping over a breakpoint.

Contents

1 Setting Up Code Composer Studio

The shared memory support in Code Composer Studio 2.0 is ISA independent. The only requirement for using the shared memory features in Code Composer Studio is to have a driver that contains shared memory support. You can verify this by looking at the list of capabilities displayed in the Code Composer Studio Setup program. One of the listed capabilities should be shared memory support. If shared memory support is not listed, you must obtain a new driver with shared memory support enabled.

To setup Code Composer Studio 2.0 for debugging a shared memory system, the user must define the shared memory configuration in the Code Composer Studio memory map.

1.1 Defining the Configuration

A shared memory attribute has been added to Code Composer Studio 2.0 to describe the configuration of shared memory to the debugger. The shared memory attribute is used to define:

- The cores that have access to the shared memory block(s).
- The range of the shared memory block(s).
- The access privileges of each core to the shared memory block(s).

The GEL command, GEL_MapAddStr, is used to define shared memory. The syntax is:

GEL_MapAddStr(*start address*, *page*, *length*, “*string*”, *waitstate*)

The *string* parameter is formatted as:

TYPE [**| SH** n [**C**]]

The required *TYPE* attribute designates one of three possible access types:

RAM – Core has Read and Write access to the shared memory block.
ROM – Core has Read Only access to the shared memory block.
WOM – Core has Write Only access to the shared memory block.

When defining a shared memory block, append the optional shared memory attribute by using the | operator.

SH*n*[C] – Core has access to shared memory.

The above *TYPE* attribute determines the type of access the core has to the shared memory block. The *n* represents the block number that uniquely identifies the block on all processors. It should never be zero.

The SH symbol has an optional **C** attribute. The **C** attribute can be used to designate *Common Shared* memory (i.e., the shared memory block contains executable code). The attribute is used to differentiate between memory that contains code or data that will be accessed by two or more cores, and memory that is physically shared but the memory contents are not.

When describing each core’s shared memory access privileges to Code Composer Studio, the user must describe the emulation read/write access to shared memory, not the CPU’s access. For instance, the CPU on the TMS320C5421 cannot write directly to any shared memory. The DMA module must perform all shared memory writes. However, the emulation portion of the TMS320C5421 has direct write access to portions of the shared memory configuration.

Example:

The TMS320C5421 has the following shared memory emulation configuration:

```
 Core 1:     0x08000–0x0ffff   Read/Write Access
             0x18000–0x1ffff   Read/Write Access
             0x28000–0x2ffff   Read Only Access
             0x38000–0x3ffff   Read Only Access

 Core 2:     0x08000–0x0ffff   Read Only Access
             0x18000–0x1ffff   Read Only Access
             0x28000–0x2ffff   Read/Write Access
             0x38000–0x3ffff   Read/Write Access
```

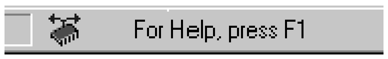
If an application is linked into the entire shared memory section, the GEL entries to Code Composer Studio look like:

```
 Core 1:     GEL_MapAddStr(0x08000,0,0x8000,”RAM|SH1C”,0);
             GEL_MapAddStr(0x18000,0,0x8000,”RAM|SH2C”,0);
             GEL_MapAddStr(0x28000,0,0x8000,”ROM|SH3C”,0);
             GEL_MapAddStr(0x38000,0,0x8000,”ROM|SH4C”,0);

 Core 2:     GEL_MapAddStr(0x08000,0,0x8000,”ROM|SH1C”,0);
             GEL_MapAddStr(0x18000,0,0x8000,”ROM|SH2C”,0);
             GEL_MapAddStr(0x28000,0,0x8000,”RAM|SH3C”,0);
             GEL_MapAddStr(0x38000,0,0x8000,”RAM|SH4C”,0);
```

For your convenience, pre-formatted GEL files for all currently available devices that support shared memory are provided in the cc\gel directory of the Code Composer Studio 2.0 installation.

When shared memory support is enabled in Code Composer Studio, the shared memory icon is visible at the bottom of the Code Composer window in the center of the status bar.


For Help, press F1

If you configure the GEL file for shared memory, but the driver does not contain shared memory support, Code Composer Studio reports:

```
The driver does not support shared memory.
All shared memory support will be disabled
for the remainder of this session.
```

1.2 Note to Users of CCS 1.2x

The separate shared memory drivers (5421_shared.dvr, 5440_shared.dvr and 5441_shared.dvr files) are no longer needed in CCS 2.0. The shared memory support present in these files has been moved into the Code Composer executable file.

2 Debugging Multi-Core Systems

There are several strategies that can be used to debug target devices containing multiple cores that share the same memory block(s).

Many users of multi-core devices run the same code on each core (i.e., the use of shared memory). The cores may or may not interact or depend on what the other cores are doing. If there is no interaction between cores, the best strategy is to debug the code on a single core, then simply load and run it on the other cores. In fact, the majority of users debug multi-core systems by debugging code on a single core at a time. However, when individual cores interact or depend on what other cores are doing, it is desirable to debug multiple cores simultaneously.

There are two ways to debug multiple cores simultaneously. These methods can be used exclusively or interchangeably:

- Use the Parallel Debug Manager for broadcasting commands to multiple cores simultaneously.

  The Parallel Debug Manager allows users to perform synchronous execution on multiple cores or multiple devices. This includes stepping, running, and halting in parallel.
- Use the Parallel Debug Manager to start a separate Code Composer Studio debug session for each core, then switch back and forth between the separate sessions running on different cores**.**

  In this case, the user executes debug commands within one debug session at a time.

Normally, a debug session is unaffected by the actions being performed in another debug session. However, when debugging shared memory systems, any activity in shared memory affects all of the other debug sessions running on cores with access to the shared memory. Support has been added to Code Composer Studio 2.0 to deal with these situations.

3 Debugging Shared Memory

When writing to shared memory or reading from shared memory, the debugger knows what read/write access is allowed for each core and chooses a correct core to do the write or read.

Breakpoints set in shared memory are active in all debuggers running on cores with access to the shared memory.

4 Setting Shared Memory Options

The Code Composer Studio Shared Memory Configuration dialog box enables users to control whether cores with access to shared memory are halted when:

- Writing to shared memory.
- Stepping over a breakpoint.

4.1 Writing to Shared Memory

Code Composer Studio enables users to force all the cores with access to shared memory to halt while shared memory is modified. This feature is useful when debugging multiple cores simultaneously or while other cores are running code or reading/writing data to shared memory. This is the default behavior of Code Composer Studio.

When debugging a single core at a time, while no other cores are accessing shared memory, this feature is not needed and can be turned off.

To toggle this option on or off:

1. Select Options→Customize.
2. In the Customize dialog box, select the Shared Memory Configuration tab.
3. Click the check box to select or de-select "Affected cores will be halted during a write to shared memory".

Alternatively, the following GEL commands can be used to turn this feature on and off:

```
GEL_SharedMemHaltOnWriteOn()
GEL_SharedMemHaltOnWriteOff()
```

4.2 Stepping Over a Breakpoint

If a breakpoint set in shared memory must be stepped over, all cores that have access to the shared memory must be halted. The breakpoint will be cleared, the core(s) stepping over the breakpoint will be stepped, and the breakpoint will be set again.

Code Composer Studio enables users to force all the cores with access to shared memory to halt while a breakpoint in shared memory is being stepped over. This feature is useful when users are debugging multiple cores/devices simultaneously, or while other cores are running code or reading/writing data to shared memory. This is the default setting in Code Composer Studio.

When debugging a single core at a time, while no other cores are accessing shared memory, this feature is not needed and can be turned off.

NOTE: Turning off this setting may result in missed breakpoints if other cores execute code where the breakpoint has been removed to perform the step-over function.

To toggle this option on or off:

1. Select Options→Customize.
2. In the Customize dialog box, select the Shared Memory Configuration tab.
3. Click the check box to select or deselect "Affected cores will be halted when a breakpoint in shared memory is stepped over."

Alternatively, the following GEL commands can be used to turn this feature on and off:

```
GEL_SharedMemHaltOnStepOn()
GEL_SharedMemHaltOnStepOff()
```