TMS320C54x Simulator

Addendum to the TMS320C5xx C Source Debugger User's Guide

SPRU170
February 1996

Printed on Recycled Paper

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software 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.

Certain applications using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

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

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI 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.

Contents

Chapter 1

Changes to the TMS320C5xx C Source Debugger User's Guide

Section 1.8, *Debugger Options*, in the *TMS320C5xx C source Debugger User's Guide*, describes the options that you can use when invoking the debugger. The –mv option has been added for the simulator version of the debugger.

The –mv option specifies which memory map the simulator loads. By default, the simulator loads the memory map contained in the siminit.cmd file, which is a generic memory map. Each of the provided memory maps simulates a different 'C54x device, as described in the following table:

Option	Device Simulated	Initialization File Used	Peripherals Simulated
–mv541	'C541	sim541.cmd	Serial port 0, serial port 1, timer
–mv542	'C542	sim542.cmd	Buffered serial port, TDM serial port, timer
–mv543	'C543	sim543.cmd	Buffered serial port, TDM serial port, timer
–mv544	'C544	sim544.cmd	Serial port 0, serial port 1, timer
–mv545	'C545	sim545.cmd	Buffered serial port, serial port 1, timer
–mv546	'C546	sim546.cmd	Buffered serial port, serial port 1, timer
–mv547	'C547	sim547.cmd	Buffered serial port, serial port 1, timer
–mv549	'C549	sim549.cmd	Buffered serial port, serial port 1, timer

Chapter 2

Defining a Memory Map

Note:

This chapter replaces Chapter 6, *Defining a Memory Map*, in the *TMS320C5xx C Source Debugger User's Guide*.

Before you begin a debugging session, you must supply the debugger with a memory map. The memory map tells the debugger which areas of memory it can and can't access. Note that you can use the Memory pulldown menu to enter the commands described in this chapter.

Topic **Page**

2.1 The Memory Map: What It Is and Why You Must Define It

A memory map tells the debugger which areas of memory it can and can't access. Memory maps vary, depending on the application. Typically, the map matches the MEMORY definition in your linker command file.

Note:

When the debugger compares memory accesses against the memory map, it performs this checking in software, not hardware. The debugger can't prevent your program from attempting to access nonexistent memory.

A special default initialization batch file included with the debugger package defines a memory map for your version of the debugger. This memory map may be sufficient when you first begin using the debugger. However, the debugger provides a complete set of memory-mapping commands that let you modify the default memory map or define a new memory map.

You can define the memory map interactively by entering the memory-mapping commands while you're using the debugger. However, this can be inconvenient because, in most cases, you'll set up one memory map before you begin debugging and will use this map for all of your debugging sessions. The easiest method of defining a memory map is to put the memory-mapping commands in a batch file.

Defining the memory map in a batch file

There are two methods for defining the memory map in a batch file:

- ❏ Redefine the memory map defined in the initialization batch file
- ❏ Define the memory map in a separate batch file of your own

When you invoke the debugger, it follows these steps to find the batch file that defines your memory map:

1) The debugger checks if you've used the –t debugger option. If the debugger finds the –t option, it executes the specified file. (Use the –t option to specify a batch file other than the initialization batch file shipped with the debugger.)

2) If you don't use the –t option, the debugger looks for the default initialization batch file. The batch filename differs for each version of the debugger:

   - ❑ For the emulator, this file is called *emuinit.cmd*.
   - ❑ For the EVM, this file is called *evminit.cmd*.
   - ❑ For the simulator, this file is called *siminit.cmd*.

   If the debugger finds the file corresponding to your tool, it executes the file.

3) If the debugger does not find the –t option or the initialization batch file, it looks for a file called *init.cmd*. This search mechanism allows you to have one initialization batch file for more than one debugger tool. To set up this file, you can use the IF/ELSE/ENDIF commands (for more details, see the controlling command execution in a batch file information in Chapter 5, *Entering and Using Commands*) to indicate which memory map applies to each tool.

Potential memory map problems

You may experience these problems if the memory map isn't correctly defined and enabled:

- ❑ **Accessing invalid memory addresses.** If you don't supply a batch file containing memory-map commands, then the debugger is initially unable to access any target memory locations. Invalid memory addresses and their contents are highlighted in the data-display windows. (On color monitors, invalid memory locations, by default, are displayed in red.)
- ❑ **Accessing an undefined or protected area.** When memory mapping is enabled, the debugger checks each of its memory accesses against the memory map. If you attempt to access an undefined or protected area, the debugger displays an error message. For specific error messages, see Appendix D, *Debugger and PDM Messages.*
- ❑ **Loading a COFF file with sections that cross a memory range.** Be sure that the map ranges you specify in a COFF file match those that you define with the MA command (described on page 2-8). Alternatively, you can turn memory mapping off during a load by using the MAP OFF command (see page 2-12).
- ❑ **Accessing conflict and extra cycles (simulator only).** If two memory read access requests occur simultaneously during an execution, you may be unable to complete both requests within the same clock cycle. If both locations belong to the same physical memory block and the block is single-access memory, both requests cannot be processed within the same clock cycle.

2.2 Customizing the Memory Map

The customizable ’C54x (cDSP) debugger allows you maximum flexibility in configuring a memory map. Because the size and address of the memory map is not fixed in the debugger, you can select any amount of ROM or RAM internally, externally, or both.

The following example shows how you can have both RAM and ROM mapped to the same address:

```
ma 0xc000, 0, 0x1000, R          ;Internal Program ROM

ma 0xc000, 0, 0x1000, R|EX       ;External Program ROM
```

During execution or when the debugger performs memory accesses, the block of memory accessed is based on the ’C54x MP/$\overline{\text{MC}}$ bit located in the PMST register. When this bit is set to 0, the on-chip program ROM is enabled. When it is set to 1, the off-chip program RAM is enabled.

The next example shows you two blocks of RAM, one internal and one external, mapped to the same address.

```
ma 0x0080, 0, 0x0380, R|W        ;Internal Program RAM

ma 0x0080, 0, 0x0380, R|W|EX     ;External Program RAM
```

For the above example, the block of memory is accessed based on the OVLY bit located in the PMST register during execution or when the debugger performs memory accesses. When this bit is set to 1, the on-chip dual-access data RAM is mapped to internal program space. When it is set to 0, the off-chip program RAM is enabled.

The debugger accesses the three types of memory (data, program ROM, and program RAM) according to the type of memory and the values of the MP/$\overline{\text{MC}}$ bits. The following table summarizes how the debugger accesses memory:

Type of Memory	Memory Access
Data	Accesses internal memory block, then external memory block.
Program ROM	If MP/$\overline{\text{MC}}$ is set to 0, accesses internal memory block, then external memory block; if MP/$\overline{\text{MC}}$ is set to 1, accesses external memory block.
Program RAM	If OVLY is set to 1, accesses internal memory block, then external memory block; if OVLY is set to 0, accesses external memory block.

Mapping on-chip dual-access RAM to program memory

The following steps describe how to map a block of memory to program memory. The memory that is mapped is configured as on-chip dual-access RAM in the data memory.

Step 1: Set the OVLY (overlay bit) in the PMST register to 1.

Step 2: Define the data-memory map before the program-memory map. It is essential to define the data-memory map for the overlay mode.

Step 3: Add a dummy program-memory map in the same region as the external memory. To do this, use the EX attribute.

Note:

The size of the data-memory map and the program-memory map must be the same.

The following is an example of mapping the on-chip dual-access RAM to program memory. The example shows the commands to set the mode to overlay.

```
ma 0x0080, 1, 0x0f80, R|W|DA

ma 0x0080, 0, 0x0f80, R|W|EX

?pmst=0xffc0 ; mp/mc=0, ovly=1
```

Simulating data memory (ROM)

With the ’C54x simulator, you can simulate the DROM bit in the ’C541, ’C543, ’C544, ’C545, ’C546, ’C547, or ’C549 processor. This simulation allows you to map the on-chip program memory (ROM) to the data memory. To map the program memory (ROM) to the data memory, follow these steps:

Step 1: Set the DROM bit in the PMST register to 1.

Step 2: Invoke the simulator with the –mv541, –mv543, –mv544, –mv545, –mv546, –mv547, or –mv549 option.

The following is an example of simulating data memory:

```
?pmst=0x10       ;  DROM bit is set to 1
```

Programming your memory

The easiest time to set up your memory is during the initialization process. However, you can edit your memory map while your program is running.

Use the OVLY and MP/$\overline{MC}$ bits of the status/PMST registers to set the amount of external and internal program memory you need. The values for the OVLY and MP/$\overline{MC}$ bits are as follows:

- OVLY bit

  0 = external program memory

  1 = internal program memory

- MP/$\overline{MC}$ bit

  0 = internal program memory (ROM)

  1 = external program memory

You can edit the the values of the OVLY and MP/$\overline{MC}$ bits by using the debugger or by programming the PMST register. To edit the values of these bits, scroll down the CPU window until you see the PMST register. The CPU window is editable; you can enter the values for each bit.

2.3 A Sample Memory Map

Because you must define a memory map before you can run any programs, it's convenient to define the memory map in the initialization batch files. Example 2–1 shows the memory map commands that are defined in the initialization batch file that accompanies the simulator. If you are using the simulator, you can use the file as is, edit it, or create your own memory map batch file. The files shipped with the emulator and EVM are similar to that of the simulator.

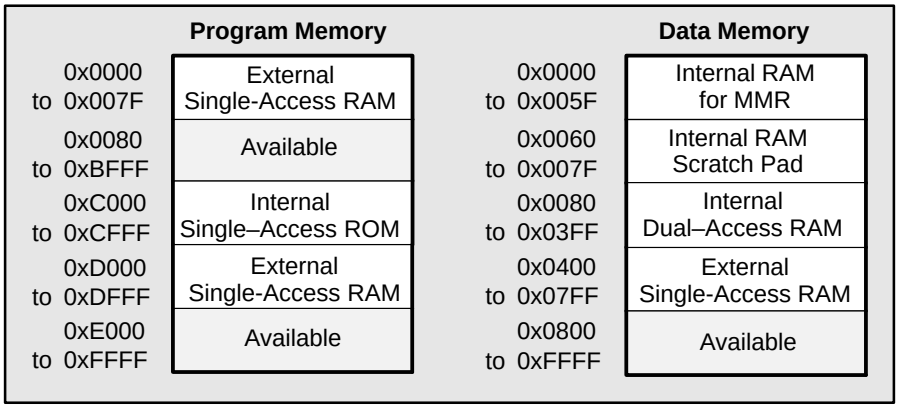
Example 2–1. Sample Initialization Batch File for Use with the TMS320C54x Simulator

```
ma 0x0000, 0, 0x80,   EX|RAM
ma 0xc000, 0, 0x1000, ROM
ma 0xd000, 0, 0x1000, EX|RAM

ma 0x0000, 1, 0x0060, RAM
ma 0x0060, 1, 0x0020, RAM
ma 0x0080, 1, 0x0380, RAM
ma 0x0400, 1, 0x0400, EX|RAM
```

The MA commands (shown in Example 2–1) define valid memory ranges and identify the read/write characteristics of the memory ranges. The MAP command enables mapping (see Section 2.5, *Enabling Memory Mapping*, on page 2-12). By default, mapping is enabled when you invoke the debugger. Figure 2–1 illustrates the memory map defined in Example 2–1.

Figure 2–1. Sample Memory Map for Use With the TMS320C54x Simulator



2.4 Identifying Usable Memory Ranges

ma The debugger's MA (memory add) command identifies valid ranges of target memory. The syntax for this command is:

ma *address, page, length, type*

- The *address* parameter defines the starting address of a range. This parameter can be an absolute address, any C expression, the name of a C function, or an assembly language label.

  A new memory map must not overlap an existing entry. If you define a range that overlaps an existing range, the debugger ignores the new range and displays this error message in the COMMAND window display area:

  ```
  Conflicting map range
  ```

- The *page* parameter is a one-digit number that identifies the type of memory (program, data, or I/O) that a range occupies:

To identify this page,	Use this value as the *page* parameter
Program memory	**0**
Data memory	**1**
I/O space	**2**

- The *length* parameter defines the length of the range. This parameter can be any C expression.

- The *type* parameter identifies the read/write characteristics of the memory range. The *type* must be one of these keywords:

To identify this kind of memory,	Use this keyword as the *type* parameter
Read-only memory	**R** or **ROM**
Write-only memory	**W** or **WOM**
Read/write memory	**R|W** or **RAM**
Read/write external memory	**RAM|EX** or **R|W|EX**
Read-only port	**P|R**
Read/write port	**P|R|W**
Single-access memory	**SA**
Dual-access memory	**DA**

Usage notes

- The debugger caches memory that is not defined as a port type (P|R, P|W, or P|R|W). For ranges that you don't want cached, be sure to map them as ports.
- When you are using the simulator, you can use the parameter values P|R, P|W, and P|R|W to simulate I/O ports. See Section 2.9, *Simulating I/O Space*, on page 2-16.
- Be sure that the map ranges that you specify in a COFF file match those that you define with the MA command. A command sequence such as:

  `ma x,0,y,ram; ma x+y,0,z,ram` ↵

  doesn't equal

  `ma x,0,y+z,ram` ↵

  If you plan to load a COFF block that spans the length of y + z, you should use the second MA command example. Alternatively, you can turn memory mapping off during a load by using the MAP OFF command (see Section 2.5, *Enabling Memory Mapping*, on page 2-12)

- Although the address range for both of the following MA commands is the same (0x0400 to 0x0800), one range is internal and the other range is external.

  `ma 0x0400, 0, 0x0800,  ROM` ↵
  `ma 0x0400, 0, 0x0800,  EX|ROM` ↵

  When the simulator is operating in microcomputer mode, the internal program ROM is accessed. Otherwise, if the simulator is running in microprocessor mode, the external program memory module is used.

- If a range of memory is configured as single-access RAM (using the SA attribute with the MA command), it means only one access (read/write) can be performed on any address in the block in one cycle. You can configure more than one single-access RAM block. Simultaneous accesses to different single-access RAM blocks during the same cycle are permitted.

  For example, the following commands create two single-access RAM blocks. The blocks are 0x100 in size. If an instruction performs two accesses, one in the first block (for example, address 0x110) and another in the second block (for example, address 0x230), the instruction executes in only one cycle.

  ```
  ma 0x0100, 1, 0x0100, R|W|SA ⏎
  ma 0x0200, 1, 0x0100, R|W|SA ⏎
  ```

  Contrarily, if the blocks were combined into one block and configured as one single chunk of 0x200 words (as shown in the following command), simultaneous accesses to addresses 0x110 and 0x230 would take two cycles to complete.

  ```
  ma 0x100, 1, 0x200, R|W|SA ⏎
  ```

- If a range of memory is configured as dual-access RAM (using the DA attribute with the MA command), it means two simultaneous accesses (read/write) can be performed during the same cycle to the block.

  For example, the following command creates one dual-access RAM as a data page. If an instruction performs two simultaneous accesses to two addresses in this block, both accesses execute in one cycle.

  ```
  ma 0x0100, 1, 0x0100, R|W|DA ⏎
  ```

Memory mapping with the simulator

Unlike the emulator and EVM, the ’C54x simulator has memory cache capabilities that allow you to allocate as much memory as you need. However, to use memory cache capabilities effectively with the ’C54x, do not allocate more than 20K words of memory in your memory map. For example, the memory map shown in Example 2–2 allocates 64K words of ’C54x program memory.

Example 2–2. Sample Memory Map for the TMS320C54x Using Memory Cache Capabilities

```
MA 0,0,0x5000,R|W
MA 0x5000,0,0x5000,R|W
MA 0xa000,0,0x5000,R|W
MA 0xf000,0,0x1000,R|W
```

The simulator creates temporary files in a separate directory on your disk. For example, when you enter an MA (memory add) command, the simulator creates a temporary file in the root directory of your current disk. Therefore, if you are currently running your simulator on the C drive, temporary files are placed in the C:\ directory. This prevents the processor from running out of memory space while you are executing the simulator.

Note:

If you execute the simulator from a floppy drive (for example, drive A), the temporary files are created in the A:\ directory.

All temporary files are deleted when you exit the simulator using the QUIT command. If, however, you exit the simulator with a soft reboot of your computer, the temporary files will not be deleted; you must delete these files manually. (Temporary files usually have numbers for names.)

With the memory cache capabilities of the simulator, your memory map is now restricted only by your PC’s capabilities. As a result, there should be sufficient free space on your disk to run any memory map you want to use. If you use the MA command to allocate 20K words (40K bytes) of memory in your memory map, then your disk should have at least 40K bytes of free space available. To do this, you can enter:

`ma 0x0, 0, 0x5000, ram` ↵

Note:

You can also use the memory-cache capability feature for the data memory.

2.5 Enabling Memory Mapping

map By default, mapping is enabled when you invoke the debugger. In some instances, you may want to explicitly enable or disable memory. You can use the MAP command to do this; the syntax for this command is:

map on
or **map off**

Note that disabling memory mapping can cause bus fault problems in the target because the debugger may attempt to access nonexistent memory.

Note:

When memory mapping is enabled, you cannot:

- ❏ Access memory locations that are not defined by an MA command
- ❏ Modify memory areas that are defined as read only or as protected

If you attempt to access memory in these situations, the debugger displays this message in the COMMAND window display area:

```
Error in expression
```

2.6 Checking the Memory Map

ml If you want to see which memory ranges are defined, use the ML (memory list) command. The syntax for this command is:

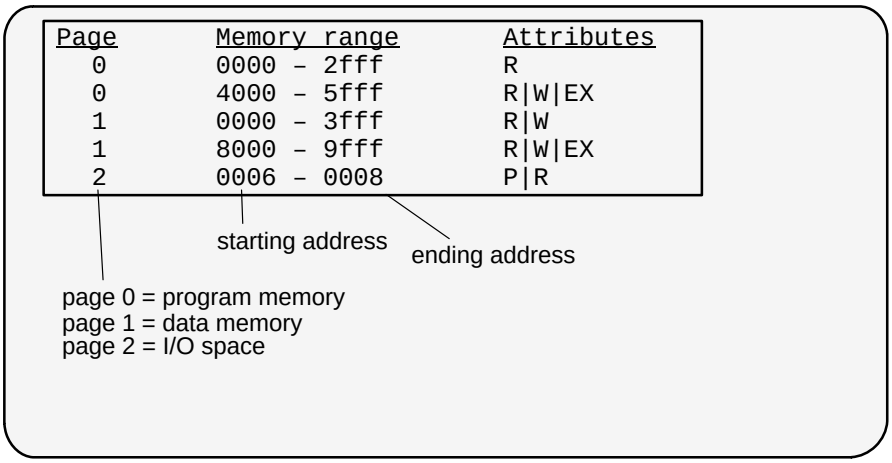
ml

The ML command lists the page, starting address, ending address, and read/write characteristics of each defined memory range.

For example, assume you issue the following MA commands:

```
ma 0,0, 0x3000, ROM ↵
ma 0x4000, 0, 0x2000, EX|RAM ↵
ma 0, 1, 0x4000, RAM ↵
ma 0x8000, 1, 0x2000, EX|RAM ↵
ma 0x6, 2, 0x3, P|R ↵
```

If you enter the ML command, the debugger displays the following in the COMMAND window display area:



2.7 Modifying the Memory Map During a Debugging Session

If you need to modify the memory map during a debugging session, use these commands.

md To delete a range of memory from the memory map, use the MD (memory delete) command. The syntax for this command is:

md *address, page*

- The *address* parameter identifies the starting address of the range of program, data, or I/O memory. If you supply an *address* that is not the starting address of a range, the debugger displays this error message in the COMMAND window display area:

  ```
  Specified map not found
  ```

- The *page* parameter is a one-digit number that identifies the type of memory (program, data, or I/O) that the range occupies:

To identify this page,	Use this value as the *page* parameter
Program memory	**0**
Data memory	**1**
I/O space	**2**

Note:

If you are using the simulator and want to use the MD command to remove a simulated I/O port, you must first disconnect the port with the MI command (see *Disconnecting an I/O port*, page 2-20).

mr If you want to delete all defined memory ranges from the memory map, use the MR (memory reset) command. The syntax for this command is:

mr

This resets the debugger memory map.

ma If you want to add a memory range to the memory map, use the MA (memory add) command. The syntax for this command is:

ma *address, page, length, type*

The MA command is described in detail on page 2-8.

Returning to the original memory map

If you modify the memory map, you may want to go back to the original memory map without quitting and reinvoking the debugger. You can do this by resetting the memory map and then using the TAKE command to read in your original memory map from a batch file.

Suppose, for example, that you had set up your memory map in a batch file named *mem.map*. You could enter these commands to go back to this map:

```
mr ↵                          Reset the memory map
take mem.map ↵                Reread the default memory map
```

The MR command resets the memory map. (Note that you could put the MR command in the batch file, preceding the commands that define the memory map.) The TAKE command tells the debugger to execute commands from the specified batch file.

2.8 Using Multiple Memory Maps for Multiple Target Systems (Emulator Only)

If you're debugging multiple applications, you may need a memory map for each target system. Here's the simplest method for handling this situation.

Step 1: Let the initialization batch file define the memory map for one of your applications.

Step 2: Create a separate batch file that defines the memory map for the additional target system. The filename is unimportant, but for this example assume that the file is named *filename.x*. The general format of this file's contents should be:

```
mr                            Reset the memory map
MA commands                   Define the new memory map
map on                        Enable mapping
```

(Of course, you can include any other appropriate commands in this batch file.)

Step 3: Invoke the debugger as usual.

Step 4: The debugger reads the initialization batch file during invocation. Before you begin debugging, read in the commands from the new batch file:

```
take filename.x ↵
```

This redefines the memory map for the current debugging session.

You can also use the –t option instead of the TAKE command when you invoke the debugger. The –t option allows you to specify a new batch file to use instead of the default initialization batch file.

2.9 Simulating I/O Space (Simulator Only)

In addition to adding memory ranges to the memory map, you can use the MA command to add I/O ports to the memory map. To do this, use P|R (input port) or P|R|W (input/output port) as the memory type. Use page 2 to simulate I/O space. Then you can use the MC command to connect a port to an input or output file. This simulates external I/O cycle reads and writes by allowing you to read data in from a file and/or write data out to a file. Use page 1 for file connects to data memory.

Connecting an I/O port

mc The MC (memory connect) command connects P|R or P|R|W to an input or output file. MC also allows you to connect any data memory location (except 0x0000–0x001F) to an input or output file to read data from or write data into the file. The syntax for this command is:

mc *portaddress, page, length, filename, fileaccess*

- The *portaddress* parameter defines the address of the I/O space or data memory. This parameter can be an absolute address, any C expression, the name of a C function, the name of a C function, or an assembly language label.

  The *portaddress* must be previously defined with the MA command (described on page 2-8) and have a keyword of either P|R (input port) or P|R|W (input/output port). The length of the address range defined for the port (or peripheral frame) can be 0x1000 to 0x1FFF bytes and does not have to be a multiple of 16.

- The *page* parameter is a one-digit number that identifies the type of memory (data or I/O) that the address occupies:

To identify this page,	Use this value as the *page* parameter
Data memory	**1**
I/O space	**2**

- The *length* parameter defines the length of the range. This parameter can be any C expression.

- The *filename* parameter can be any filename. If you connect a port or memory location to read from a file, the file must exist, or the MC command will fail.

- The *fileaccess* parameter identifies the access characteristics of the I/O memory and data memory. The file access must be one of the keywords identified below:

To identify this file access type	Use this keyword as the *fileaccess* parameter
Input port (I/O space)	**P|R**
Simulator halt at EOF of input space (I/O space)	**R|P|NR**
Output port (I/O space)	**P|W**
Read-only internal memory	**R**
Read-only external memory	**EX|R**
Simulator halt at EOF of input file for internal memory	**R|NR**
Simulator halt at EOF of input file for external memory	**EX|R|NR**
Write-only internal memory	**W**
Write-only external memory	**EX|W**

For I/O memory locations, the file is accessed during a read or write instruction to the associated port address. You can connect any I/O port to a file. A maximum of one input and one output file can be connected to a single port; however, multiple ports can be connected to a single file.

For data memory locations, the debugger accesses the data as follows:

- When you're executing code:
  - If you have specified a file, the debugger reads the data from the file and updates the memory location with that data.
  - If you have specified a file, the debugger writes the data to the memory location, as well as to the file.
- When you're using the debugger:
  - The debugger reads the data value from the memory location, *not* from the connected file.
  - If you have specified a file, the debugger writes the data to the memory location, as well as to the file.

If you use the NR parameter, then the simulator halts execution when it reads an EOF. The debugger displays the appropriate message in the COMMAND window display area:

```
<addr> EOF reached – connected at port(I/O_PAGE)
```

or

```
<addr> EOF reached – connected at location (DATA_PAGE)
```

At this point, you can disconnect the file by using the MI command and attach a new file by using the MC command. If you don't do anything, then the input file rewinds automatically, and execution continues until EOF is read.

If you do not specify the NR parameter, execution does not halt, and you are not notified when EOF is reached. The input file rewinds automatically, and the simulator resumes reading from the file.

Example 2–3 shows how input and output ports can be connected to specific memory blocks.

Example 2–3. Connecting Input and Output Ports to Input or Output Files.

Assume that you have two data-memory blocks:

```
ma 0x100,1, 0x10, EX|RAM    ;block1
ma 0x200,1, 0x10, RAM       ;block2
```

- You could use the MC command to set up and connect an input file to block1:

```
mc 0x100, 1, 0x1, my_input.dat, EX|R
```

- You could use the MC command to set up and connect an output file to block2:

```
mc 0x205, 1, 0x1, my_output.dat, W
```

- You could use the MC command to halt simulator at EOF of input file:

```
mc 0x100, 1, 0x1, my_input.dat, EX|R|NR
```

or

```
mc 0x100, 1, 0x1, my_input.dat, R|NR
```

Example 2–4 shows how to connect an input port to an input file named in.dat.

Example 2–4. Connecting an Input Port to an Input File

Assume that the file in.dat contains words of data in hexadecimal format, one per line, like this:

```
0A00
1000
2000
   .
   .
   .
```

Use MA and MC commands to set up and connect an input port:

```
MA     0x50,2,0x1,R|P          Configure port address 50h
                                as an input port.
MC     0x50,2,0x1,in.dat,R     Open file in.dat and
                                connect it to port address 50.
```

Assume that the following instruction is part of your program; it reads from the file in.dat:

```
PORTR  050,data_mem            Read file in.dat, and put the
                                value into the DATA_MEM location.
```

Notes:

1) You can only connect a file to configured location(s).
2) You cannot connect a file to program memory (page 0) locations.
3) You cannot connect a file to the core-memory map register area (0x0000 to 0x001F) of data memory (page 1).
4) While connecting a file to a set of locations:
   - Locations must not spread across memory block boundaries.
   - Two read-only files must not overlap.
   - Two write-only files must not overlap.

Disconnecting an I/O port

Before you can use the MD command to delete a port from the memory map, you must use the MI command to disconnect the port.

mi The MI (memory disconnect) command disconnects a file from an I/O port. The syntax for this command is:

mi *port address, page,* {**R**|**W**|**EX**}

The *port address* and *page* identify the port that will be closed. The read/write/execute characteristics must match the parameter used when the port was connected.

2.10 Simulating External Interrupts (Simulator Only)

The ’C54x simulator allows you to simulate the external interrupt signals $\overline{\text{INT1}}$ to $\overline{\text{INT4}}$ and allows you to select the clock cycle where you want an interrupt to occur. To do this, you create a data file and connect it to one of the interrupt pins, $\overline{\text{INT1}}$ to $\overline{\text{INT4}}$ or the $\overline{\text{BIO}}$ pin.

Note:

The time interval is expressed as a function of CPU clock cycles. Simulation begins at the first clock cycle.

Setting up your input file

In order to simulate interrupts, you must first set up an input file that lists interrupt intervals. Your file must contain a clock cycle in the following format:

[*clock cycle, logic value*] **rpt** {*n* | **EOS**}

Note that the square brackets are used only with logic values for the $\overline{\text{BIO}}$ pin.

- The *clock cycle* parameter represents the CPU clock cycle in which you want an interrupt to occur.

  You can have two types of CPU clock cycles:

  - **Absolute**. To use an absolute clock cycle, your cycle value must represent the actual CPU clock cycle in which you want to simulate an interrupt. For example:

    ```
    12 34 56
    ```

    Interrupts are simulated at the 12th, 34th, and 56th CPU clock cycles. No operation is performed on the clock cycle value; the interrupt occurs exactly as the clock cycle value is written.

  - **Relative**. You can also select a clock cycle that is relative to the time at which the last event occurred. For example:

    ```
    12 +34 55
    ```

    In this example, a total of three interrupts are simulated at the 12th, 46th (12 + 34), and 55th CPU clock cycles. A plus sign (+) before a clock cycle adds that value to the total clock cycles preceding it. You can mix both relative and absolute values in your input file.

- The *logic value* parameter is only for the $\overline{\text{BIO}}$ pin. You must use a value to force the signal to go high or low at the corresponding clock cycle. A value of 1 forces the signal to go high, and a value of 0 forces the signal to go low. For example:

  ```
  [12,1] [23,0] [45,1]
  ```

  This causes the $\overline{\text{BIO}}$ pin to go high at the 12th cycle, low at the 23rd cycle, and high again at the 45th cycle.

- The **rpt** {n | **EOS**} parameter is optional and represents a repetition value.

  Two forms of repetition simulate interrupts:

  - **Repetition on a fixed number of times**. You can format your input file to repeat a particular pattern a fixed number of times. For example:

    ```
    5 (+10 +20) rpt 2
    ```

    The values inside the parenthesis represent the portion that is repeated. Therefore, an interrupt is simulated at the 5th CPU cycle, then the 15th (5 + 10), 35th (15 + 20), 45th (35 + 10), and 65th (45 + 20) CPU clock cycles.

    Note that n is a positive integer value.

  - **Repetition to the end of simulation**. To repeat the same pattern throughout the simulation, add the string EOS to the line. For example:

    ```
    10 (+5 +20) rpt EOS
    ```

    Interrupts are simulated at the 10th CPU cycle, then the 15th (10+5), 35th (15 + 20), 40th (35 + 5), 60th (40 + 20), 65th (60 + 5), and 85th (65 + 20) CPU cycles, continuing in that pattern until the end of simulation.

Programming the simulator

After creating your input file, you can use debugger commands to connect, list, and disconnect the interrupt pin to your input file. Use these commands as described below, or use them from the PIN pulldown menu.

pinc To connect your input file to the pin, use the following command:

pinc *pinname, filename*

- The *pinname* identifies the pin and must be one of the simulated pins ($\overline{\text{INT1}}$–$\overline{\text{INT4}}$) or the $\overline{\text{BIO}}$ pin.
- The *filename* is the name of your input file.

Example 2–5 shows you how to connect your input file using the PINC command.

Example 2–5. Connecting the Input File With the PINC Command

Suppose you want to generate an $\overline{\text{INT2}}$ external interrupt at the 12th, 34th, 56th, and 89th clock cycles.

First, create a data file with an arbitrary name, such as myfile:

```
12 34 56 89
```

Then use the PINC command in the pin pulldown menu to connect the input file to the $\overline{\text{INT2}}$ pin.

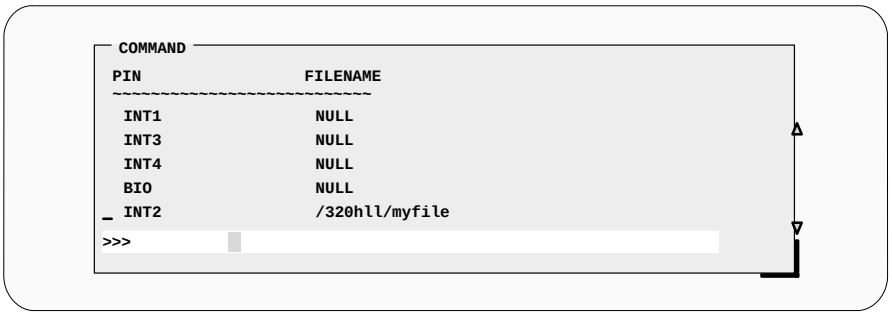
```
pinc  myfile, int2                    Connects your data file
                                      to the specific interrupt pin
```

This command connects myfile to the $\overline{\text{INT2}}$ pin. As a result, the simulator generates an $\overline{\text{INT2}}$ external interrupt at the 12th, 34th, 56th, and 89th clock cycles.

pinl To verify that your input file is connected to the correct pin, use the PINL command. The syntax for this command is:

pinl

The PINL command displays all of the unconnected pins first, followed by the connected pins. For a pin that has been connected, it displays the name of the pin and the absolute pathname of the file in the COMMAND window.



pind To end the interrupt simulation, disconnect the pin. You can do this with the following command:

pind *pinname*

The *pinname* parameter identifies the interrupt pin and must be one of the external interrupt pins ($\overline{\text{INT1}}$–$\overline{\text{INT4}}$) or the $\overline{\text{BIO}}$ pin. The PIND command detaches the file from the input pin. After executing this command, you can connect another file to the same pin.

2.11 Simulating Peripherals (Simulator Only)

With the ’C54x simulator, you can simulate the timer, standard serial port, buffered serial port, and TDM serial port. The peripherals simulated depend upon the device that you simulate. You simulate a device by starting the simulator (sim5xx command) with the appropriate option. Table 2–1 summarizes the options used to simulate the peripherals for each device.

Table 2–1. Debugger Options for the Simulator

Option	Device Simulated	Peripherals Simulated
–mv541	’C541	Serial port 0, serial port 1, timer
–mv542	’C542	Buffered serial port, TDM serial port, timer
–mv543	’C543	Buffered serial port, TDM serial port, timer
–mv544	’C544	Serial port 0, serial port 1, timer
–mv545	’C545	Buffered serial port, serial port 1, timer
–mv546	’C546	Buffered serial port, serial port 1, timer
–mv547	’C547	Buffered serial port, serial port 1, timer
–mv549	’C549	Buffered serial port, serial port 1, timer

Detailed information about simulating the different types of serial ports is discussed in the following sections:

Type of Serial Port	See Section . . .
Standard	2.12 on page 2-26
Buffered	2.13 on page 2-30
TDM	2.14 on page 2-33

2.12 Simulating Standard Serial Ports (Simulator Only)

The ’C54x simulator supports standard serial port transmission and reception by reading data from, and writing data to, the files associated with the DXR/TDXR and DRR/TDRR registers, respectively.

The simulator also provides limited support for the simulation of the serial port control pins (frame synchronization pins) with the help of external event simulation capability. Frame synchronization pin values for receive and transmit operations at various instants of time are fed through the files associated with the pins.

The ’C54x simulator supports the following operations in the standard serial port simulation:

- **Internal clocks (1/4 CPU clock) and external clocks for the transmit and receive operations.** External clocks are simulated by using the DIVIDE command (described on page 2-27) in the files connected to the FSX/TFSX and FSR/TFSR pins.
- **External frame synchronization pulses** (FSX/TFSX transmit and FSR/TFSR receive frame synchronization pulses). Transmit and receive operations are initiated when the signals for these values go high.
- The operations associated with the following memory-mapped registers:

Register	Memory	Bits Used	Description
SPC	0x22	FO	Format specifier (8/16 bits)
TSPC	0x32	MCM XRST/RRST XRDY/RRDY XSREMPTY RSRFULL	Internal/external clock Transmit/receive reset Transmit/receive ready Transmit register empty flag Receive register full flag
DXR TDXR	0x20 0x30	All bits are used	Transmit data register
DRR TDRR	0x21 0x31	All bits are used	Receive data register

Setting up your transmit and receive operations

The ’C54x simulator supports the simulation of the following pins using external event simulation. The pulses occurring on the FSX and FSR pins initiate the standard serial port transmit and receive operations, respectively.

- **FSR/TFSR**—Frame synchronization pulses for the receive operation
- **FSX/TFSX**—Frame synchronization pulses for the transmit operation

Connect the files to the pins using the PINC (pin connect) command (described on page 2-23). Use the following command syntax, selecting the appropriate command for the pin you want:

```
pinc FSX,filename
pinc TFSX,filename
pinc FSR,filename
pinc TFSR,filename
```

filename is the name of the file that contains the CPU cycles at which the pin value goes high. Use the following syntax in the files to define clock cycles:

[*clock cycle*] **rpt** {*n* | **EOS**}

Note that the square brackets are used only with logic values for the $\overline{\text{BIO}}$ pin. For more information about defining clock cycles, see Section 2.10 on page 2-21.

Additionally, you can use the DIVIDE command to specify the clock divide ratio for the device. Use the following syntax in the files for the DIVIDE command:

DIVIDE *r*

r is a real number or integer specifying the ratio of serial port clock versus the CPU clock. Use the divide ratio when the serial port is configured to use the external clock. When you use the DIVIDE command, it must be the first command in the file.

The following example specifies the clock ratio of the transmit clock and the clock cycles for the occurrence of TFSX pulses (if this file is connected to the TFSX pin):

```
DIVIDE 5
100  +200  +100
```

The DIVIDE command specifies the divide-down ratio of the clock against the CPU clock. That is, the CLKX frequency is 1/5 of the CPU clock. The second line indicates that the TFSX should go high at the 100th, 300th (100 + 200), and 400th (300 + 100) CPU cycles. The TFSX pin goes high in the 500th, 1500th, and 2000th cycles of the serial port clock.

Connecting input/output files

Input and output files are connected to DRR/TDRR and DXR/TDXR registers for receive and transmit operations, respectively. To simulate the transmit operation, data is written to the file that is connected to the DXR/TDXR register. To simulate the receive operation, data is read from the file that is connected to the DRR/TDRR register.

The input and output file formats for the standard serial port operation requires at least one line containing an hexadecimal number. The following is an acceptable format for an input file:

```
0055
aa55
efef
dead
```

Note:

To simulate the standard serial port 0, use the DXR and DRR registers and the FSX and FSR pins. To simulate the standard serial port 1, use the TDXR and TDRR registers and the TFSX and TFSR pins.

Programming the simulator

To simulate the standard serial port, configure the DXR/TDXR and DRR/TDRR registers as the output port (OPORT) and the input port (IPORT), respectively. Connect these ports to an output file and an input file. Also, connect files to the TFSX/FSX and TFSR/FSR pins to specify the clock cycles during which the frame synchronization pins go high.

To make these connections, use the following commands in the simulator initialization batch file (siminit.cmd):

```
ma DRR,1,1,R|P
ma DXR,1,1,W|P

mc DRR,1,1,receive filename,READ
mc DXR,1,1,transmit filename,WRITE

pinc FSX,fsx timing filename
pinc FSR,fsr timing filename
```

Variable	Description
receive filename	The file to read data from, which simulates the input port
transmit filename	The file to write data to, which simulates the output port
fsx timing filename	The file that contains the CPU cycles at which the FSX frame synchronization pin goes high
fsr timing filename	The file that contains the CPU cycles at which the FSR frame synchronization pin goes high

2.13 Simulating Buffered Serial Ports (Simulator Only)

The ’C54x simulator supports buffered serial port transmission and reception by reading data from and writing data to the files associated with the DXR and DRR registers, respectively.

The simulator also provides limited support for the simulation of the serial port control pins (frame synchronization pins) with the help of external event simulation capability. Frame synchronization pin values for receive and transmit operations at various instants of time are fed through the files associated with the pins. The ’C54x simulator supports the following operations in the buffered serial port simulation:

- **Automatic buffering and standard serial port modes**
- **Internal clocks (1/(CLKDV + 1) CPU clock) and external clocks for the transmit and receive operations**
- **External frame synchronization pulses** (FSX transmit and FSR receive frame synchronization pulses). Transmit and receive operations are initiated when the signals for these values go high.
- The operations associated with the following memory-mapped registers:

Register	Memory	Bits Used	Description
SPC	0x22	FO MCM XRST/RRST XRDY/RRDY XSREMPTY RSRFULL	Format specifier (8/16 bits) Internal/external clock Transmit/receive reset Transmit/receive ready Transmit register empty flag Receive register full flag
DXR	0x21	All bits are used	Transmit data register
DRR	0x20	All bits are used	Receive data register
SPCE	0x23	CLKDV FE RH/TH BXE/BRE HALTX/HALTR	Clock divide ratio Extended format specifier Buffer half received or transmitted Enable/disable automatic buffering Switch to standalone mode after the current half is transmitted/received
AXR	0x38	All bits are used	Address register for transmit
ARR	0x3a	All bits are used	Address register for receive
BKX	0x39	All bits are used	Block size register for the transmit
BKR	0x3b	All bits are used	Block size register for the receive

Setting up your transmit and receive operations

The ’C54x simulator supports the simulation of the following pins using external event simulation. The pulses occurring on the FSX and FSR pins initiate the buffered serial port transmit and receive operations, respectively.

- **FSR**—Frame synchronization pulses for the receive operation
- **FSX**—Frame synchronization pulses for the transmit operation

Connect the files to the pins using the PINC (pin connect) command (described on page 2-23). Use the following command syntax, selecting the appropriate command for the pin you want:

```
pinc FSX,filename
pinc FSR,filename
```

filename is the name of the file that contains the CPU cycles at which the pin value goes high. Use the following syntax in the files to define clock cycles:

[*clock cycle*] **rpt** {*n* | **EOS**}

Note that the square brackets are used only with logic values for the $\overline{\text{BIO}}$ pin. For more information about defining clock cycles, see Section 2.10 on page 2-21.

Additionally, you can use the DIVIDE command to specify the clock divide ratio for the device. Use the following syntax in the files for the DIVIDE command:

DIVIDE *r*

r is a real number or integer specifying the ratio of serial port clock versus the CPU clock. Use the divide ratio when the serial port is configured to use the external clock. When you use the DIVIDE command, it must be the first command in the file.

The following example specifies the clock ratio of the transmit clock and the clock cycles for the occurrence of TFSX pulses (if this file is connected to the TFSX pin):

```
DIVIDE 5
100  +200  +100
```

The DIVIDE command specifies the divide-down ratio of the clock against the CPU clock. That is, the CLKX frequency is 1/5 of the CPU clock. The second line indicates that the TFSX should go high at the 100th, 300th (100 + 200), and 400th (300 + 100) CPU cycles. The TFSX pin goes high in the 500th, 1500th, and 2000th cycles of the serial port clock.

Connecting input/output files

Input and output files are connected to DRR and DXR registers for receive and transmit operations respectively. To simulate the transmit operation, data is written to the file that is connected to the DXR register. To simulate the receive operation, data is read from the file that is connected to the DRR register.

The input and output file formats for the buffered serial port operation requires at least one line containing an hexadecimal number. The following example shows an acceptable format for an input file:

```
0055
aa55
efef
dead
```

Programming the simulator

To simulate the buffered serial port, configure the DXR and DRR registers as the output port (OPORT) and the input port (IPORT), respectively. Connect these ports to an output file and an input file. Also, connect files to the TFSX/FSX and TFSR/FSR pins to specify the clock cycles during which the frame synchronization pins go high.

To make these connections, use the following commands in the simulator initialization batch file (siminit.cmd):

```
ma DRR,1,1,R|P
ma DXR,1,1,W|P

mc DRR,1,1,receive filename,READ
mc DXR,1,1,transmit filename,WRITE

pinc FSX,fsx timing filename
pinc FSR,fsr timing filename
```

Variable	Description
receive filename	The file to read data from, which simulates the input port
transmit filename	The file to write data to, which simulates the output port
fsx timing filename	The file that contains the CPU cycles at which the FSX frame synchronization pin goes high
fsr timing filename	The file that contains the CPU cycles at which the FSR frame synchronization pin goes high

2.14 Simulating TDM Serial Ports (Simulator Only)

The ’C54x simulator supports TDM serial port transmission and reception by reading data from and writing data to the files associated with the TDXR and TDRR registers, respectively.

The simulator also provides limited support for the simulation of the TDM port control pins (frame synchronization pins) with the help of external event simulation capability. Frame synchronization pin values for receive and transmit operations at various instants of time are fed through the files associated with the pins.

The ’C54x simulator supports the following operations in the TDM serial port simulation:

- **TDM and standard serial port modes**
- **Internal clocks (1/4 CPU clock) and external clocks for the transmit and receive operations.** External clocks are simulated by using the DIVIDE command in the files connected to the TFSX and TFSR pins.
- **External frame synchronization pulses** (TFSX transmit and TFSR receive frame synchronization pulses). Transmit and receive operations are initiated when the signals for these values go high.
- The operations associated with the following memory-mapped registers:

Register	Memory	Bits Used	Description
TSPC	0x32	TDM	Multiprocessor/normal mode
		MCM	Internal/external clock
		XRST/RRST	Transmit/receive reset
		XRDY/RRDY	Transmit/receive ready
		XSREMPTY	Transmit register empty flag
		RSRFULL	Receive register full flag
TCSR	0x33	All bits are used	Channel select register
TRTA	0x34	All bits are used	Receive/transmit address register
TRAD	0x35	All bits are used	Receive address register
TDXR	0x31	All bits are used	Transmit data register
TDRR	0x30	All bits are used	Receive data register

Setting up your transmit and receive operations

The ’C54x simulator supports the simulation of the following pins using external event simulation. The pulses occurring on the TFSX and TFSR pins initiate the TDM serial port transmit and receive operations, respectively.

- **TFSR**—Frame synchronization pulses for the receive operation
- **TFSX**—Frame synchronization pulses for the transmit operation

Connect the files to the pins using the PINC (pin connect) command (described on page 2-23). Use the following command syntax, selecting the appropriate command for the pin you want:

```
pinc TFSX,filename
pinc TFSR,filename
```

filename is the name of the file that contains the CPU cycles at which the pin value goes high. Use the following syntax in the files to define clock cycles:

[*clock cycle*] **rpt** {*n* | **EOS**}

Note that the square brackets are used only with logic values for the $\overline{\text{BIO}}$ pin. For more information about defining clock cycles, see Section 2.10 on page 2-21.

Additionally, you can use the DIVIDE command to specify the clock divide ratio for the device. Use the following syntax in the files for the DIVIDE command:

DIVIDE *r*

r is a real number or integer specifying the ratio of serial port clock versus the CPU clock. Use the divide ratio when the serial port is configured to use the external clock. When you use the DIVIDE command, it must be the first command in the file.

The following example specifies the clock ratio of the transmit clock and the clock cycles for the occurrence of TFSX pulses (if this file is connected to the TFSX pin):

```
DIVIDE 5
100  +200  +100
```

The DIVIDE command specifies the divide-down ratio of the clock against the CPU clock. That is, the CLKX frequency is 1/5 of the CPU clock. The second line indicates that the TFSX should go high at the 100th, 300th (100 + 200), and 400th (300 + 100) CPU cycles. The TFSX pin goes high in the 500th, 1500th, and 2000th cycles of the serial port clock.

Connecting input/output files

Input and output files are connected to TDRR and TDXR registers for receive and transmit operations, respectively. To simulate the transmit operation, data is written to the file that is connected to the TDXR register. To simulate the receive operation, data is read from the file that is connected to the TDRR register. Use the following syntax to create the files:

channel-address data

channel-address specifies the TDM channel in which transmission/reception takes place. *data* specifies the value that is written or read from the file. Each field is in hexadecimal format separated by spaces. The following is an acceptable format for an input file:

```
10  0055
34  aa55
80  efef
01  dead
```

Programming the simulator

To simulate the TDM serial port, configure the TDXR and TDRR registers as the output port (OPORT) and the input port (IPORT), respectively. Connect these ports to an output file and an input file. Also, connect files to the TFSX/FSX and TFSR/FSR registers to specify the clock cycles during which the frame synchronization pins go high.

To make these connections, use the following commands in the simulator initialization batch file (siminit.cmd):

```
ma TDRR,1,1,R|P
ma TDXR,1,1,W|P

mc TDRR,1,1,receive filename,READ
mc TDXR,1,1,transmit filename,WRITE

pinc TFSX,fsx timing filename
pinc TFSR,fsr timing filename
```

Variable	Description
receive filename	The file to read data from, which simulates the input port
transmit filename	The file to write data to, which simulates the output port
fsx timing filename	The file that contains the CPU cycles at which the FSX frame synchronization pin goes high
fsr timing filename	The file that contains the CPU cycles at which the FSR frame synchronization pin goes high