Freescale Semiconductor

Designing Freescale DSP56xxx Software for Nonrealtime Tests File I/O Using SIM56xxx and ADS56xxx

by

Tom Zudock

**For More Information On This Product,
Go to: www.freescale.com**

freescale™
semiconductor

Freescale Semiconductor, Inc.

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

freescale™
semiconductor

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

TABLE OF CONTENTS

Nonrealtime Software Tests

LIST OF EXAMPLES

SECTION 1

INTRODUCTION

Freescale Semiconductor, Inc.

1.1   INTRODUCTION

The debugging of realtime digital signal processing systems is inherently challenging because of their complex nature and the high performance they demand. By executing portions of a Digital Signal Processor (DSP) application in nonrealtime using the DSP Simulator or the Application Development System (ADS), many bugs can be identified and eliminated before system integration and system test. This application report presents the methods for performing file I/O using the Simulator and ADS. Using conditional assembly allows quick software reconfiguration for simulation, ADS, or realtime execution and is presented here as a solution to multiexecution environment needs.

1.2   SCOPE

This document focuses on using the   tools to promote quality software rather than complex DSP algorithms and systems. In fact, the DSP software and system examples have intentionally been kept simple to maintain the emphasis on the true purpose of the application report.

1.3   NONREALTIME EXECUTION

In this document, nonrealtime execution refers to the execution of code on the DSP Simulator or the use of any DSP with a debugger that prevents continuous DSP execution (i.e., using file I/O through a debugger). The benefits of designing software to be executed in this manner are numerous.

1.4   BENEFITS OF NONREALTIME EXECUTION

When a project reaches the point where DSP coding begins, it is convenient to initiate code development using file I/O. It provides a controlled environment in which input data can be specific and output data can be examined sample by sample. Further, many DSP applications' functionalities are defined by input vectors and the proper bit exact output. By specifying an input data file and saving the output as a data file, it is possible to verify such criteria. Additionally, if bit exact output is obtained only up to a certain point, the DSP software can be executed up to the point of failure and then stopped. Debugging may ensue thereafter.

Benefits of Nonrealtime Execution

DSP applications typically depend upon continuous data input, which is provided via interrupt. Multiple interrupts and processes are often performed by a system. Nonrealtime testing can be used to verify each block individually before integrating an entire system. Time spent at system test debugging subsystem failures is thus saved. Final system integration and test are complex enough without the additional effort of algorithm debug.

Using file I/O also promotes precise characterization of an algorithm without the introduction of the performance limitations of other hardware. Analog stages and codecs introduce artifacts into any DSP system. File I/O can be used to provide a "clean" digital input to evaluate maximum performance of an algorithm. Further, phase shifts, noise, and other real-world signal characteristics can be introduced to the input data to help determine the analog system performance required to maintain the desired overall signal processing capability. In addition, testing can be automated by using operating system scripts to execute debuggers and data analysis programs. This ensures test repeatability and promotes archiving test data. Moreover, computers can test around the clock while most engineers cannot.

If code is executing using the DSP Simulator, profiling information is immediately available. The   DSP profiler provides extensive execution performance statistics that can be used to optimize DSP software. Because the initial implementation of a DSP algorithm exceeds the available realtime resources, nonrealtime execution using file I/O may initially be the only method possible. Using the profiler may allow early implementations to be improved sufficiently to execute them in realtime.

dsp

SECTION   2

FILE INPUT/OUTPUT

Freescale Semiconductor, Inc.

2.1   FILE INPUT/OUTPUT (I/O)

The fundamental purpose of nonrealtime testing is to provide input data to the Digital Signal Processor (DSP) software and store the resulting output. The   DSP Simulator and Application Development System (ADS) software support this functionality differently. Both methods are summarized in this section.

2.1.1   Simulator File I/O

The Simulator supports file I/O by memory mapping. A DSP address is associated with an input or output file using a Simulator command. Each time the DSP software reads that address, the Simulator will input data from the specified file. The Simulator will output data to the specified file each time the DSP software writes that address. Consider a simple DSP program that will copy an input file to an output file and the associated Simulator commands.

Example 2-1   DSP Copy I/O Program A

```
; filename: fileio.asm
        org     p:$80
start
        move    x:simdatain,x0              ; x0=input data
        move    x0,x:simdataout            ; move x0 to output
        jmp     start                      ; repeat process


        org     x:$0                       ;
simdatain       ds      1                  ; memory location for input data
simdataout      ds      1                  ; memory location for output data
```

In the preceding program, the DSP software moves data from the X memory location `datain` to the Y memory location dataout by using the `x0` register.

Note: In the code examples that follow, the terms MSB and LSB refer to Most Significant Byte and Least Significant Byte, respectively. However, normally MSB and LSB refer to bits.

File Input/Output (I/O)

Now examine a Simulator command script that will associate an input and output file to the `datain` and `dataout` memory locations.

Example 2-2   Simulator Command Script

```
reset s                              ; reset simulator
input off                            ; reset all input files
output off                           ; reset all output files
load fileio.cld                      ; load the dsp program
input #1 simdatain infile.dat -rh    ; input from infile.dat from address datain
output #1 simdataout outfile.dat -rh -o ; output to outfile.dat to address dataout
break EOF                            ; stop when an input file reaches end-of-file
go                                   ; execute program
```

In the preceding Simulator command script the following holds true:

- Lines 1–3 reset the state of the Simulator and close any open input or output files.

- Line 4 loads the `fileio.cld` file for execution.

- Line 5 tells the Simulator to read data from the data file `infile.dat` in hexadecimal format, -rh, whenever the DSP software reads memory location `datain`.

- Line 6 provides identical functionality for the output file with the additional -o option, forcing an overwrite of the output file if that file already exists.

- Line 7 indicates that the Simulator should halt execution when the input end-of-file is reached.

- Finally, line 8 begins execution of the program.

2.1.2   Application Development System (ADS) File I/O

The Application Development System (ADS) software supports file I/O differently than the Simulator. To perform file I/O using the ADS software, a debug instruction is used in the program flow. The DSP software must initialize several registers just before the debug instruction is executed. The initialization of these registers indicates the file number and number of words to be transferred (the most significant byte and least significant byte of X0), the starting address of the transfer (R0), and the target memory (R1).

Now examine a DSP program that performs the same function (copies an input file to an output file) as discussed in the Simulator section, as well as the associated ADS commands.

Example 2-3   DSP Copy I/O Program B

```
; filename: fileio.asm
bufsize equ     $20

        org     p:$80
start
        move    #$010000+bufsize,x0             ; MSB=infile #1, LSB=bufsize
        move    #dataio,r0               ;input address is dataio
        move    #>1,r1                         ; input data into X memory
adsdatain       debug                          ;
        move    #$010000+bufsize,x0            ; MSB=outfile #1, LSB=bufsize
        move    #dataio,r0                    ; output address is dataio
        move    #>1,r1                        ; output data into X memory
adsdataout      debug                          ;
        jmp     start                         ; repeat process

        org     x:$0                          ;
dataio  ds      bufsize                       ; buffer for data io
```

At first glance, it would appear that this program doesn't do very much at all. However, by coupling it with the proper ADS command script, it will copy an input file to an output file. When the program is running and a debug instruction is encountered, the ADS is notified. The ADS will check the address of the debug instruction to see if it is associated with an input or output command. If it is, the ADS will use the contents of registers X0, R0, and R1, to provide the information it needs to either read or write data to or from the DSP.

Notice that this program transfers blocks of data, size $20. The block size is user-specified in the least significant byte of the X0 registers. However, be aware that since all of the DSP code is running at full processor speed except for the file I/O, the smaller the block size, the longer it will take to execute the code. Since the block size is a byte-wide entity, the largest block size is 255. If block transfers larger than 255 are required, then a loop can be used that updates the X0, R0, and R1 registers as needed and repeatedly executes the debug instruction.

File Input/Output (I/O)

Now examine the ADS command script that will properly initialize the debugger.

Example 2-4 ADS Command Script

```
force s                                  ; reset command converter and target processor
device dv0 56004                         ; specifiy target device type
change or 0                              ; enable PRAM (for some 56xxx processors)
input off                                ; reset all input files
output off                               ; reset all output files
load fileio.cld                          ; load the dsp program
input #1 adsdatain infile.dat -rh        ; input from infile.dat from address datain
output #1 adsdataout outfile.dat -rh -o  ; output to outfile.dat to address dataout
break EOF                                ; stop when an input file reaches end-of-file
go                                       ; stop when an input file reaches end-of-file
```

In the preceding ADS command script the following holds true:

- Line 1 resets the command converter and target system, which provides a known initial state for the system.

- Line 2 specifies the target processor for the command converter.

- Line 3 sets the omr to zero which for the DSP56004 enables internal Program RAM and disables the bootstrap code.

- Lines 4–5 reset all input and output files by closing them.

- Line 6 loads the program.

- Lines 7–8 specify the file numbers and program memory addresses for the I/O.

- Line 9 tells the ADS to break when the input file reaches its end.

- Finally, the program is started with the GO command.

Notice that the addresses specified are taken directly from the labels used in the DSP program. Also, notice the file numbers are those used to initialize the X0 register just prior to the debug instruction in the DSP program.

2.2 COMMON FILE I/O MISTAKES

The preceding sections provide simple examples of methods that can be used to perform file I/O using the ADS and Simulator software. Although the techniques are straightforward, there are a number of frequently made errors. Hopefully, this section will prevent the reader from making the same mistakes many have already encountered.

2.2.1 Mistakes Common to Both the Simulator and the ADS

The following steps should be taken to avoid most mistakes common to both the Simulator and the ADS:

- Verify that the input file data is the same radix as that specified in the input statement.

- Eliminate any potential infinite loops using conditional assembly. For instance, if the DSP software polls an external device before proceeding and there is no intention to simulate the external device, be certain to bypass the code.

- Close all open input and output files if initiating new execution. Reloading and reexecuting a DSP program does not reset the file pointers to any open input or output files. Hence, input data will resume from the current input file position and data output will be appended to the current output position.

2.2.2 Simulator Mistakes

To avoid Simulator mistakes, take the following steps:

- If simulating peripheral interrupts, confirm that the cycle spacing between simulated interrupts matches the actual sample rate. If interrupted too frequently, the application may experience buffer underruns or overruns.

- Disable interrupts or peripherals if they will interfere with nonrealtime execution of the DSP code. Although it is completely possible and often desirable to fully simulate the peripherals that provide I/O into the application, be sure to disable these routines if I/O will be performed without them.

2.2.3 ADS Mistakes

To avoid the most common ADS mistakes, take the following steps:

- The ADS block I/O is linear and does not examine the contents of M0 to determine if it is set for modulo addressing. Hence, if the data is being transferred to/from a modulo buffer, it must be copied from/to a linear block and then transferred from/to the file. Otherwise, data may be input/output past the top of the modulo buffer.

- The ADS file I/O method uses several registers that may be needed by the DSP program. Be certain that destroying the registers does not affect the program or preventively back them up to make the I/O code transparent to operation.

- Disable interrupts or peripherals if they will interfere with nonrealtime execution of the DSP code. Since the DSP is likely part of a larger system, the peripherals may be receiving clocks or other assertion signals. Alternatively, the peripherals may internally be programmed to generate the signals and interrupts themselves. The result may be undesired executions of an interrupt service routine accessing input and output buffers when the intended I/O is via the debugger.

SECTION   3

CONDITIONAL ASSEMBLY

Nonrealtime Software Tests

3.1   CONDITIONAL ASSEMBLY

Because the Simulator and the ADS use different file I/O techniques and require different support code, conditional assembly should be used to select the proper build for the execution that will ensue. Through the use of a makefile, conditional assembly, and Simulator/ADS command macros, the software can promptly and easily be rebuilt and executed as desired. To facilitate such an approach and maintain readability, the code should be structured to use subroutines for input and output.

Combine the two applications discussed in prior sections into one application that can be built to execute using file I/O on either the Simulator or the ADS.

Example 3-1   Combined Application Code

```
; filename: fileio.asm
bufsize    equ         $20


           org         p:$80
start
           if          ("exec"=="adsio")||("exec"=="simio")
           jsr         datain                       ; call input data subroutine
           endif                                    ;
           ; core purpose of DSP application here   ;
           if          ("exec"=="adsio")||("exec"=="simio")
           jsr         dataout                      ; call output data subroutine
           endif                                    ;
           jmp         start                        ; repeat process

           if ("exec"=="simio")||("exec"=="adsio")  ;
datain                                              ;
           if ("exec"=="simio")                     ;
           move        #dataio,r0                   ; r0=base address of I/O buffer
           move        #-1,m0                       ; set linear addressing mode
           do          #bufsize,_input_data         ; read in bufsize words
           move        x:simdatain,x0               ;
           move        x0,x:(r0)+                    ;
_input_data                                         ;
           endif                                    ; endif "simio"
           if ("exec"=="adsio")                     ;
           move        #$010000+bufsize,x0           ; MSB=infile #1, LSB=bufsize
           move        #dataio,r0                    ; input address is dataio
           move        #>1,r1                        ; input data into X memory
adsdatain  debug                                    ;
           endif                                    ; endif "adsio"
           rts                                      ;
```

Example 3-1 Combined Application Code (Continued)

```
dataout                                       ;
        if ("exec"=="simio")                  ;
        move       #dataio,r0                 ; r0=base address of I/O buffer
        move       #-1,m0                     ; set linear addressing mode
        do         #bufsize,_output_data      ; read in bufsize words
        move       x:(r0)+,x0                 ;
        move       x0,x:simdataout            ;
_output_data                                  ;
        endif                                 ; endif "simio"
        if ("exec"=="adsio")                  ;
        move       #$010000+bufsize,x0        ; MSB=outfile #1, LSB=bufsize
        move       #dataio,r0                 ; output address is dataio
        move       #>1,r1                     ; output data into X memory
adsdataout debug                              ;
        endif                                 ; endif "adsio"
        rts
        endif                                 ; endif all file I/O routines


        org        x:$0                       ;
        if ("exec"=="simio")||("exec"=="adsio") ;
dataio  ds         bufsize                    ;
        if ("exec"=="simio")                  ;
simdatain ds       1                          ;
simdataout ds      1                          ;
        endif
        endif
```

In the preceding example, there are essentially two conditional assembly variables used: `simio` and `adsio`. Notice that in some portions of the code the conditional inclusion is dependent upon only one of the variables, while in others the dependency is upon both variables. In short, the code sections included by either variable are used in both Simulator and ADS file I/O, while those included by only one of the variables are needed for only that type of I/O.

Also notice that if neither variable is defined, no file I/O code is included and the bare DSP application is left. A third conditional assembly variable could be used that indicates the code build is for realtime I/O. An example of this would perhaps be `rtio`. This variable could be used to include peripheral initialization when the build is for the final system.

Specifying the section of code to be included at assembly time can be done in two ways. It is recommended that both be used. First, the define assembler directive can be used in the code itself. An advantage of this is that a default value is always found in the code. For instance, the final system build variable (`rtio` suggested earlier) may be in the code to define the default build. When the code is built, a different conditional assembly variable may be specified to override the default variable in the code.

Below, the code is built for ADS file I/O using the -d assembler option (which overrides any internally defined value for the exec variable).

```
asm56000 -a -b -l -d exec adsio fileio.asm
```

The code was intentionally set up to use the same Simulator and ADS macros presented in previous sections. The user simply needs to use the proper macro for a given code build. All that remains for the software presented to truly become useful is for the user to add the spice of a signal processing algorithm.

3.2   EXAMPLE CODE

The example code presented in this application report is available via the website at the following address:

```
http://www.freescale-dsp.com/documentation/appnotes
```

Freescale Semiconductor, Inc.