Freescale Semiconductor
Application Note

Document Number: AN3754
Rev. 0, 07/2007

Using Symphony™ Studio with the DSP563xxEVM

The DSP563xxEVM hardware development tool is an evaluation module that supports all of the general purpose DSP563xx devices available in the 196-pin Molded Array Process-Ball Grid Array (MAP-BGA) package. The Freescale Symphony™ Studio Development Software tool supports the Symphony audio DSP family, as well as all DSP56300-based DSPs supported by the DSP563xxEVM.

This application note discusses how to use the Symphony Studio tool to edit and run code on the DSP563xxEVM. This application note also presents code to program the Flash memory of the DSP563xxEVM with user code so that the EVM can be used in stand-alone mode.

Contents

freescale™
semiconductor

1 DSP563xxEVM Overview

The DSP563xxEVM was designed to support all of the general purpose DSP563xx devices that are currently available in the 196-pin MAP-BGA package: DSP56303, DSP56309, DSP56311 and DSP56321. Included with the DSP563xxEVM is a sample case that contains all of these devices so the user can choose which device to evaluate. The DSP563xxEVM contains a socket in which any of these devices can be placed. Some of these devices have different core voltage requirements. Table 1-1 shows the jumper settings required to set the correct core voltage for each device.

Table 1-1. DSP563xxEVM Voltage Jumper Settings

Device	Core Voltage	J18 (Voltage Select) Setting	J16 (Low Voltage Select) Setting
DSP56303	3.3 V	1 to 2 (3.3 V)	—
DSP56309	3.3 V	1 to 2 (3.3 V)	—
DSP56311	1.8 V	2 to 3 (Low voltage determined by J16)	3 to 4
DSP56321	1.6 V	2 to 3 (Low voltage determined by J16)	5 to 6

The DSP563xxEVME User's Manual (DSP563xxEVMEUM/D) describes the DSP563xxEVM in more detail. This application note assumes that the DSP563xxEVM has been setup as described in Sections 1.2.1 - 1.2.3 of the User's Manual. It is important for the timing of the Flash memory programming code to verify that the DSP is driven with the 19.6608 MHz clock by setting the jumper at J13 (CLK_SEL) to pins 1 to 2. This application note does not use the Suite56 Software described in Section 1.2.4, so it is not necessary to complete the steps in this section.

The DSP563xxEVM also includes:

- 64K × 24-bit fast static RAM (FSRAM) for expansion memory
- 256K × bit-bit Flash memory for stand-alone operation
- 16-bit CD-quality audio codec
- headers to facilitate direct access for off board devices to the on-chip peripheral ports
- 14-pin JTAG/OnCE connector that allows direct connection of an external command converter
- buttons for asserting RESET and the IRQ pins of the DSP
- switches for setting the DSP boot mode.

This application note presents assembly code to program the Flash memory with user code to allow stand-alone operations after setting the correct boot mode switches. This document also describes how to write code to Flash memory that programs the DSP ESSI ports to communicate to the audio codec and allow music to be passed from the audio input jack (J12) to the DSP and back to the audio output jack (J10) or the headphone jack (J15). Please see **Appendix A** of the *DSP563xxEVME User's Manual* for a full description of the audio codec programming code.

2 Symphony Studio Overview

This application note describes how to use the new Symphony Studio Development Software with the DSP563xxEVM. Symphony Studio uses the extensible development platform Eclipse, which is an open source industry standard, to provide a new way to develop code through an Integrated Development Environment (IDE). The IDE allows code creation and editing as well as project management, debugging and code compilation all in one software suite. Symphony Studio is fully backward compatible with existing application code, as the Suite56 assembler and linker tools are reused in the platform. Since the toolset reuses many components from the Suite56 platform, the learning curve is minimized. The IDE allows quicker code generation and easy program management. In addition, unlike the Suite56 tool that only allows access to the EVM through a parallel port, the Symphony Studio allows communication to the EVM through a USB port.

Symphony Studio can be downloaded from the Freescale web site at http://www.freescale.com/symphonystudio. The *Symphony Studio User Guide* (DSPSTUDIOUG) describes the Symphony Studio in detail. This application note assumes that the Symphony Studio has been installed as described in **Section 2** of the *Symphony Studio User's Guide*.

This application note discussed two options for the hardware command converter:

- Axiom parallel port DSP JTAG Pod (which is included with the DSP563xxEVM) for which you must install the parallel port driver as described in Section 2.3.2 of the Symphony Studio User Guide
- Xverve USB Signalyzer Tool (with the DSP563xx adapter) for which you must install the drivers that can be downloaded from the Signalyzer web site at http://www.signalyzer.com/. Pricing information and a detailed description of the Signalyzer tool can also be found at the Signalyzer web site.

3 Load and Edit Code: C/C++ Perspective

For details, please see the Symphony Studio documentation that contains information on how to create, build, debug and run assembly projects. There are usually multiple methods to complete any task in the Symphony Studio, for example using a pull down menu, using a right mouse click shortcut, using a button in the tool bar, or using one of the function keys. This document attempts to point out as many of these techniques as possible.

Creating and editing code (including the Flash memory programming assembly code) is done in the **C/C++** perspective of Symphony Studio. Different perspectives are chosen by clicking on the corresponding text in the tab in the top right hand corner of the Symphony Studio (C/C++ or Debug) application window. If you do not see the perspective you need in the top right hand corner, you can use the **Window > Open Perspective > Other...** pull down menu to open either of these perspectives.

3.1 Create the Project

Symphony Studio uses projects to organize different groups of code. To create a new project for the Flash memory programming code, navigate to the C/C++ perspective then choose **File** > **New** > **Managed Make ASM Project**. You can also right click in the **Navigator** View and choose **New** > **Managed Make ASM Project**. The Navigator View is usually on the left hand side of the C++ Perspective. If not, you can open the Navigator View using the **Window > Show View > Other...** pull down menu. You can also create a new project by clicking on the **New** tool bar button (which looks like a little window with a plus in the upper corner).

Type a project name in the **Project name:** field and click on the **Next>** button. Then click on the **Finish** button because the default settings are appropriate for this project. You can now see your empty project in the Navigator View.

3.2 Add the Code

Next, add the Flash memory programming code to your new project. If you already have the Flash memory programming code in a file, you can use the import function to add the file to the project. To use the import function, right click in the Navigator View and choose **Import** or choose **File** > **Import** from the pull down menu. Choose **File System** under **General** and click **Next>**. Make sure the **Into folder:** field contains your project name. Click the **Browse** button next to the **From directory:** field. Navigate to the directory that contains the Flash memory programming code and click the **OK** button. Click in the box next to the Flash memory programming code and click the **Finish** button. You can also add the Flash memory programming code to the project by dragging it from a file explorer window and dropping it into the project folder in Symphony Studio.

If you do not have the Flash memory programming code in a file, you can use the new file option. To create a new file, right click in the Navigator View and choose **New > File** or choose **File** > **New > File** from the pull down menu. Click on your project name, and type a name for the Flash memory programming code file (`flash.asm` works nicely) in the **File name field:**. Then click the **Finish** button. Now you can copy the text of the Flash memory programming code from Example 5 of this document and paste it into the editing window of your new file.

The Flash memory programming code requires that the user code you want to write to the Flash memory is also loaded into the DSP memory. For this document, we write the audio codec programming code that is included with the DSP563xxEVM (on the *DSP563xxEVME Technical Documentation CD*) to the Flash memory. Therefore, the codec code needs to be added to the project so that it can be loaded to the DSP memory before running the Flash memory programming code. Locate where the codec files were saved from the DSP563xxEVM software installation. The codec files should be in a directory called Tests\Codec_Source. To add the codec files to the project, use the **Import** function (as described above) or drag the files from a file explorer window to the Symphony Studio project folder. You need to add the following files from the Codec_Source directory:

- direct.asm
- ada_equ.asm
- ada_init.asm
- ioequ.asm

3.3 Edit the Code

Once the necessary code files are added to the project, you must edit these code files. You must tell the Flash memory programming code which DSP device you will be using and you must change the codec code to allow it to build in the Symphony Studio environment.

3.3.1 Flash Memory Programming Code Device Define

The Flash memory programming code in Example 5 uses a **define** assembly directive, shown in Example 1, to determine on which device the code is going to be run.

Example 1. Flash Memory Programming Code Device Define Directive

```
        DEFINE          DEVICE   '56311'
```

The code can be configured for a DSP56303, DSP56309, DSP56311, or DSP56321 device by changing the text in quotes of the **define** directive to `56303`, `56309`, `56311` or `56321` respectively. To use the Flash memory programming code for a device other than the DSP56311, you must change the text in the quotes to represent the device you are using. The information from the **define** directive is used in the Flash memory programming code with conditional assembly directives to determine the location of the Flash memory programming code in program memory and to program the PLL registers.

The Flash memory programming code is placed at the top of the internal program memory so that the user code can be placed in the low program memory (and use the default interrupt vector locations, if needed). The different DSP devices have different amounts of program memory. Therefore, the Flash memory programming code uses conditional assembly directives to determine where to place the code depending on the device selected in the **define** directive.

The DSP56321 has a different PLL circuit than the other DSP563xx devices. Therefore, the Flash memory programming code uses conditional assembly directives to choose different PLL register programming instructions when the code is to be run on a DSP56321 device.

3.3.2 Codec Code Edits

The codec code include file directives need to be modified to be used with Symphony Studio. First, right click on each of the three include files (ada_equ.asm, ada_init.asm and ioequ.asm) in the Navigator View and choose **Rename**. Change the extension of each of these three files from `.asm` to `.equ`. Your project now contains two `.asm` files (`flash` and `direct`) and three `.equ` files (`ada_equ`, `ada_init` and `ioequ`).

Next, double click on the direct.asm file to open it in the editing window. At the top of the direct.asm file, change the code from Example 2 to be the same as the code from Example 3.

Example 2. Original direct.asm Include Directives

```
include 'ioequ.asm'
include 'intequ.asm'
include 'ada_equ.asm'
; NOTE -- audio passthrough method below uses polling, not interrupts
include 'vectors.asm'
```

Example 3. Modified direct.asm Include Directives

```
include '../ioequ.equ'
include '../ada_equ.equ'
```

The file extensions for the `ioequ` and `ada_equ` files have been updated. The relative reference to the parent directory is required because the Managed Make ASM scripts build the application in the Debug directory (which is created if it does not already exist) within the project directory. The source files are located in the project directory. In addition, the codec code does not use interrupts; therefore, the `intequ.asm` and `vectors.asm` references can be removed.

Finally, scroll down to the bottom of the direct.asm file and make the same changes for the `include` statement for the `ada_init.asm` file. The final statement for the `ada_init include` file should look like Example 4.

Example 4. Modified direct.asm Include Code for ada_init File

```
include '../ada_init.equ'
```

Make sure you save any files that you modify. You can save files using the **File > Save** pull down menu or by clicking on the **Save** button in the tool bar (which looks like a little diskette).

3.4 Build the Code

Now, you can build the project. Symphony Studio has two build choices, which depend on how the **Build Automatically** option is set under the **Project** pull down menu. If this option is selected (check next to the option), Symphony Studio builds the project every time a file within the project is modified and saved. If this option is not selected (no check next to the option), Symphony Studio builds the project only when instructed. If Build Automatically is not selected, you can direct Symphony Studio to build your project by using the **Project > Build All** pull down menu, by clicking on the **Build All** button in the tool bar (which looks like a little piece of paper with 1s and 0s on it) or, by right clicking on the project name in the Navigator View and choosing **Build Project**. If the code was modified correctly, a few warnings may appear in the Console window at the bottom of the Symphony Studio, but no errors.

4 Debug and Run Code: Debug Perspective

After the code is built, debug and run the code by navigating to the **Debug** Perspective. To switch to the **Debug** Perspective, click on the **Debug** tab in the top right hand corner of the Symphony Studio or use the **Window > Open Perspective** pull down menu.

But first, if you are using a parallel port command converter (Axiom DSP JTAG Pod), you must modify the communication timeout values to allow the debugger enough time to download both the Flash memory programming code and the codec code. To increase the timeout interval, choose the **Window > Preferences** pull down menu and then open the **C/C++ > Debug > GDB MI** preferences. Change the Debugger timeout to 30000 ms and the Launch timeout to 90000 ms. This is just an example of timeout values. You can adjust these timeout numbers to appropriate values for your system.

4.1 External Tool Configuration

The first step in debugging is to setup the External Tool. The External Tool launches the Open On-Chip Debugger (OpenOCD), which is a GDB-JTAG server that enables the host computer to communicate with the target hardware. The External Tool configuration is managed via the **Run > External Tools > External Tools...** pull down menu. Highlight the **OpenOCD GDB Server** configuration on the left, then click the **New** button to create a configuration. You can also double click on the **OpenOCD GDB Server** configuration to create the configuration. In the **OpenOCD Configuration File** group, choose **56300** for the **Device**. Then choose the **Dongle** based in the hardware command converter you are using:

- **Signalyzer** for the Signalyzer USB tool
- **Wiggler** for the Axiom parallel port DSP JTAG Pod.

If you are going to be using more than one OpenOCD server configuration, you may want to change the name of the configuration in the **Name:** field to describe each configuration more clearly. Then you can click on the **Run** button to launch the connection to either OpenOCD server (make sure the EVM is powered on and connected to the PC correctly).

You can also use the **External Tools** button in the tool bar (which looks like an arrow in a green circle with a little red toolbox). If you click on this button, Symphony Studio automatically launches the last used External Tool configuration, or you can click on the arrow to the right of the button to access the External Tools options (instead of using the pull down menu).

When you launch an External Tool Configuration, the new link is listed in the Debug View window as well as any notes or errors in the Console window. After you run an External Tool configuration the first time, it becomes available as a choice in the **Run > External Tools** pull down menu (and the External Tools tool bar button menu), therefore, you do not need to create this configuration again.

To stop the External Tool process use the **Run > Terminate** pull down menu or click on the Terminate button in the Debug View window tool bar (which looks like a red square) when the External Tool process is highlighted in the Debug View window. To remove the External Tool process from the Debug View window, you can click on the **Remove All Terminated Launches** button in the Debug View window tool bar (which looks like two grey x's). You can also terminate and remove an External Tool processes by right clicking on it in the Debug View window and choosing Terminate, Terminate and Remove, or even Terminate and Relaunch. You must terminate the process before it can be removed. You should not terminate the External Tool process if you still have a Debug process running.

4.2 Debug

After the connection to the EVM has been established, the code can be downloaded to the DSP. This is accomplished with the Debug configuration. To setup a new Debug configuration, use the **Run > Debug...** pull down menu. Highlight **Freescale 563xx**, then click the **New** button to create a debug configuration or you can double click on the **Freescale 563xx** to create the configuration. The **Name:** and **Project:** fields should already be filled out (you can change the name if you desire). Click on the **Search Project...** button. The object file (*.cld) for this project should already be chosen, so click the **OK** button. Then click the **Debug** button. Be patient: using the parallel command converter it may take as long as 30 seconds to download both programs to the DSP.

You can also use the **Debug** button in the tool bar (which looks like a little green bug). If you click on this button, Symphony Studio automatically launches the last used Debug configuration, or you can click on the arrow to the right of the button to access the Debug options (instead of using the pull down menu).

When you launch the Debug Configuration, the new link is listed in the Debug window as well as any notes or errors in the Console window. After you run a Debug configuration the first time, it becomes available as a choice in the **Run > Debug History** pull down menu (and the Debug tool bar button menu) for you to use; therefore, you do not need to create this configuration again.

After the code is downloaded to the DSP, your Flash memory programming code appears in a window view below the Debug View. A green highlight tells where the program counter is located in the code. The green highlight should appear on the first line of the Flash memory programming code (`move #$100,r0`). You can step through the code and watch the green highlight change one line at a time. You can step by using the **Run > Step Into** or **Step Over** pull down menu, clicking on the **Step Into** or **Step Over** Debug View tool bar buttons (which look like yellow arrows), or using the F5 and F6 function keys.

Now the code is ready to run. There are several ways to run the code:

- use the **Run > Resume** pull down menu.
- use the **Resume** Debug View tool bar button which looks like a green triangle.
- right click on the debug process list in the Debug View and choose **Resume.**
- use the F8 function key.

When using any of these methods, the Flash memory programming code runs and it loads the codec code to the Flash memory of the EVM. When the code is running, control is given to the DSP; therefore, the windows of the debugger do not update. The Flash memory programming code was written to stop automatically when it is finished and return control back to the debugger. Thus, when the Flash memory programming is complete, the debugger windows update, and you can see that the Flash memory programming code has stopped at a debug instruction.

To stop the Debug process, use the **Run > Terminate** pull down menu, or click on the Terminate button in the Debug View window tool bar (this button looks like a red square) when the Debug process is highlighted in the Debug View window. To remove the Debug process from the Debug View window, click on the **Remove All Terminated Launches** button in the Debug View window tool bar (the button looks like two grey x's). You can also terminate and remove a Debug processes by right clicking on it in the Debug View window and choosing Terminate, Terminate and Remove, or even Terminate and Relaunch. You must terminate the process before it can be removed.

5 Run Code from Flash Memory

After the codec code has been programmed to the Flash memory, the code can be run in stand-alone mode, that is, with no connection to the PC via a hardware command converter. Therefore, you can remove the command converter connected to J3 of the EVM. You must make sure that the correct boot mode is set on the boot mode switches at SW4. The DSP checks the pins connected to these switches (MODA-D) after reset to determine how to boot the DSP. To boot the DSP from the Flash memory, use boot mode 9: bootstrap from byte-wide memory, by setting the switches on SW4 to be 4 = OFF, 3 = ON, 2 = ON, and 1 = OFF.

The audio codec programming code requires music input to the EVM and a set of headphones for the output. Connect a music source to the audio input jack at J12 using an audio interface cable with 1/8-inch stereo plugs and connect a set of headphone to the jack at J15. Then, you simply press the Reset button, SW1, and you can hear the music in the headphones. Now, you have successfully programmed the DSP563xxEVM Flash memory and run the DSP in stand-alone mode using the Symphony Studio development tools.

6 Flash Memory Programming Code Description

Example 5 is an example of code used to program the Flash memory. This code has three sections: the main code, the bootloader code, and the Flash memory write routine. The following subsections describe in detail the code sections used in this example.

Example 5. Flash Memory Programming Code

```
;---------------------------------------------------------------------------
; Flash.asm: routine to make user code bootable from FLASH on the DSP563xxEVM
;
; Execute with: R0 pointing to first word of user code
;               R1 containing size of the user code in WORDS
;
; Copyright Freescale 2008
;---------------------------------------------------------------------------
         DEFINE            DEVICE   '56311'
; Put I/O equates here for flash program so codec program and use ioequ.file
; and everything compiles in one big project
M1_PCTL           EQU $FFFFFD                ; PLL Control Register
M1_PCTL_321       EQU $FFFFD1                ; DSP56321 DPLL Control Register
M1_DSCR_321       EQU $FFFFD0                ; DSP56321 DPLL Static Control Register
M1_AAR1           EQU $FFFFF8                ; Address Attribute Register 1
M1_BCR            EQU $FFFFFB                ; Bus Control Register
; Register Equates
AAR1V             EQU $040711                ; maps FLASH from x:$040000-x:$05ffff
BCRV              EQU $0126A1                ; 21 wait states in AAR1 domain

;---------------------------------------------------------------------------
         if (DEVICE==56303)
         org      p:$F00                    ; put flash code high in memory for DSP56303
         endif
         if (DEVICE==56309)
         org      p:$4F00                   ; put flash code high in memory for DSP56309
         endif
         if (DEVICE==56311)
```

```
        org     p:$7F00                 ; put flash code high in memory for DSP56311
        endif
        if (DEVICE==56321)
        org     p:$7F00                 ; put flash code high in memory for DSP56321
        endif

main_flash                              ; can go in low memory
        move    #$100,r0                ; first word of user code at p:$100
        move    #$100,r1                ; user code is 0x100 words long

        if (DEVICE==56321)
        movep   #$000008,x:M1_PCTL_321  ; DSP56321 PLL enabled
        movep   #$000005,x:M1_DSCR_321  ; DSP56321 PLL = 5x19.6608MHz = ~100MHz
        else
        movep   #$040004,x:M1_PCTL      ; PLL =(5/(1*1)) x 19.6608MHz = ~100MHz
        endif

        movep   #AAR1V,x:M1_AAR1        ; maps FLASH from x:$040000-x:$05ffff
        movep   #BCRV,x:M1_BCR          ; 21 wait states in AAR1 domain

        move    #>42,x0                 ; calculate number of flash sectors
        clr     b       r1,a            ; for user code
_cmpsec
        inc     b
        sub     x0,a
        bgt     _cmpsec
        move    b0,p:NUM_SEC            ; save number of sectors
;-------------------------------------------------------------------------
; write the bootloader code into the FLASH
        move    r0,p:CODE_START         ; save code start for user code
        move    r1,p:CODE_SIZE          ; save code size for user code
        move    #BOOT,r0                ; starting location of data to write
        move    #$040000,r2             ; starting location of flash memory
        jsr     WR_FLASH                ; jump to flash write routine
;-------------------------------------------------------------------------
; write the user code into the FLASH
        move    #$040080,r2             ; user code begins at flash address $80
        move    p:CODE_START,r0         ; where to get the user code
        move    p:NUM_SEC,r3            ; the number of sectors to write
        dor     r3,_wrsec
        bsr     <WR_FLASH               ; WR_FLASH loads 42 words each time
        nop
_wrsec
                nop
                debug                   ; program end
                nop
;-------------------------------------------------------------------------
; Bootloader code
; The following code moves the user code from the FLASH to P:RAM.
; The users code starts in the 2nd sector of the FLASH at x:$040080
BOOT
        dc      BOOT_END-BOOT           ; storage for size of boot
        dc      bootgo                  ; storage for address of boot
bootgo
        movep   #AAR1V,x:M1_AAR1        ; maps FLASH from p:$040000-$05ffff
        movep   #BCRV,x:M1_BCR          ; 31 wait states in AAR1 domain
```

```
load     move     p:NUM_SEC,r3              ; number of sectors to read
         move     p:CODE_START,r0           ; starting address of user code
         move     p:CODE_SIZE,r2            ; R2 contains the length of code in WORDS
         move     #$040080,r1               ; user code goes into FLASH at $040080

         dor      r3,_sector_loop           ; get all sectors
         dor      #42,_word_loop            ; 42 words per FLASH sector
         dor      #3,_byte_loop             ; each word requires 3 bytes
         move     x:(r1)+,a2                ; Load byte from FLASH
         asr      #8,a,a                    ; Shift 8 times to move byte into
                                            ; correct position.
_byte_loop
         movem    a1,p:(r0)+                ; Stores 24-bit word into P mem
         nop                                ; dummy NOP to avoid do loop restrict.
_word_loop
         move     (r1)+
         move     (r1)+
_sector_loop
         move     p:CODE_START,r1
         jmp      (r1)                      ; Jump to the users code

CODE_SIZE         ds       1               ; user code size
CODE_START        ds       1               ; starting location of user code
NUM_SEC           ds       1               ; number of sectors to save

BOOT_END          equ      *
;-------------------------------------------------------------------------
;        WR_FLASH
;        This routine writes one 128-byte FLASH sector
;
;        enter with:       r2 pointing to starting FLASH Byte Address
;                          r0 pointing to first DSP word to save
;
;        corrupts:         r0 - points to next P:memory location to save
;                          r2 - points to next FLASH byte address to write a,b,x0
WR_FLASH ; disable_protect - Software data Protection routine
disable_protect
         move     #>$8D,x0                  ; $8D at DSP ==> $AA at FLASH
         move     x0,x:$45555               ; Atmel address $5555
         move     #>$72,x0                  ; $7d at DSP ==> $55 at FLASH
         move     x0,x:$42AAA               ; Atmel address $2AAA
         move     #>$05,x0                  ; $05 at DSP ==> $A0 at FLASH
         move     x0,x:$45555               ; Atmel address $5555
                                            ;--- writes are now enabled
         dor      #42,move_code             ; move 128 bytes
         move     p:(r0)+,a                 ; get byte from SRAM
         nop
         nop
         move     a1,x:(r2)+                ; move low byte to FLASH
         lsr      #8,a
         move     a1,x:(r2)+                ; move mid byte to FLASH
         lsr      #8,a
         move     a1,x:(r2)+                ; move high byte to FLASH
         nop
move_code
         move     a1,x:(r2)+                ; dummy write
         move     a1,x:(r2)+                ; dummy write
```

```
        bsr     <delay_20m                ; wait 20 ms to finish program cycle
        rts
;-------------------------------------------------------------------------
; delay_20m --
; 21.0 ms of delay guarantees that the write cycle has finished.
; Icycles for a 98.304 MHz clock. (10.17 ns/Instruction_cycle)
; The Write Cycle Time is 20ms, so we just wait long enough here for the
; write cycle to have started AND finished...
delay_20m
        dor     #280,_l1                  ; 280 x 7374 x 27.13ns = 21.0 ms
        move    #7374,x0
        rep     x0
        nop
_l1     rts
;-------------------------------------------------------------------------

        end     main_flash
```

6.1 Main Code

The main code starts with its own I/O register equates for the PLL and program bus registers (i.e it does not use an ioequ include file) so that the codec code file can use the I/O register equate include file without conflicts in the project. The main code also includes equates for the AAR1 and BCR control register values. The Flash memory programming code is located at the top of the internal program memory so that the user code can be placed in the low program memory (and use the default interrupt vector locations if needed). See Section 3.3.1, “Flash Memory Programming Code Device Define for more detail of these instructions.

The first two move instructions after the main_flash label define the location of the first word of the user code and the number of 24-bit words in the user code. The codec programming code starts at location p:0x100 and is a little less than 0x100 words long. These values are used in the bootloader code and must be modified if the user code is changed.

The next instructions in the main code program the PLL control register(s) using conditional assembly directives. See Section 3.3.1, “Flash Memory Programming Code Device Define for more detail of these instructions. Then the main code programs the external memory interface control registers (AAR1 and BCR) to enable access to the Flash memory. The main code also calculates number of Flash memory sectors that are needed to contain the user code.

After the initialization, the main code uses the Flash memory write routine to write the bootloader code to the first sector of the Flash memory. Then the main code uses the Flash memory write routine to write the user code to the Flash memory starting at the second sector.

The main code ends at a debug instruction that passes control of the DSP to the debugger so that you know when the code is complete.

6.2 Bootloader Code

The bootloader code is loaded by the main code to the first sector of the Flash memory. The bootloader transfers the user code from the Flash memory (starting at the second sector of the Flash memory) to the DSP internal program memory and then the bootloader code jumps to the beginning of the user code. As

it is currently written, the bootloader code only moves user code from Flash memory to internal program memory. This code can easily be modified to add data transfers from Flash memory to X or Y data memory.

For stand-alone operation, the DSP performs the following steps following a reset with boot mode 9 (boot from byte wide memory):

- jump to 0xFF0000
- run the internal bootstrap code and download the bootloader code
- jump to the bootloader code
- run the bootloader and download the user code
- jump to the user code
- run the user code.

6.3 Flash Memory Write Routine

The Flash memory write routine (WR_FLASH) disables the Flash memory software data protection and then loads one Flash memory sector with data. The Flash memory write code includes a small delay routine. The delay routine is called at the end of the Flash memory write routine to guarantee that the write cycle has finished before any other data is loaded to the Flash memory. The Flash memory write routine is called by the main code as many times as necessary to write the entire user code to the Flash memory depending how many sectors are required to accommodate the user code.

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or
+1-480-768-2130
www.freescale.com/support

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)
www.freescale.com/support

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 China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 010 5879 8000
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
+1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor
@hibbertgroup.com

Document Number: AN3754
Rev. 0
07/2007

freescale™
semiconductor

Mouser Electronics

Authorized Distributor

Click to View Pricing, Inventory, Delivery & Lifecycle Information:

Freescale Semiconductor:

DSP56309EVM