Application Report
SPRA828A – August 2002

System Initialization for the OMAP5910 Device

Rishi Bhattacharya *C5000 DSP Applications*

ABSTRACT

This document will assist software engineers in understanding the procedure of initializing the OMAP5910 hardware platform. The information provided in this document is specific to the development of low-level software that accesses the OMAP5910 hardware directly.

Contents

1 Introduction 3
1.1 What to Expect From This Guide 3
1.2 Global Architecture 3
2 System Initialization Procedure 6
2.1 Resetting the OMAP5910 6
2.2 Setting the Frequencies and Enabling the OMAP5910 7
2.3 Pin Muxing Configuration 11
2.3.1 Peripheral List 11
2.3.2 Pin Multiplexing Utility 13
2.4 Configuring the OMAP5910 Device in Native Mode 16
3 Examples 16
3.1 Level 2 Interrupt Example 16
3.1.1 Codec Configuration Using the I2C Peripheral (Polling Example) 20
3.1.2 Transferring Data Using the McBSP1 Peripheral (Level 2 Interrupt Example) 22
3.2 DSP Booting Example 26
3.2.1 Converting DSP COFF Files Into ARM C-Language Header Files 26
3.2.2 MPU Code Description 28
3.2.3 Building and Running the Example 30
Appendix A TLV320AIC23 Stereo Codec 32

Figures

Figure 1. OMAP5910 Platform 5
Figure 2. Using the CLKRST_reset() Function to Perform a Global Software-Generated Reset. 6
Figure 3. Using the PLL_setFreq() Function to Set the Output Frequency of the DPLL1 to 150 MHz 7
Figure 4. An Example of How to Use the CLKRST_setupScalableMode() Function to Initialize the OMAP5910 Device 8
Figure 5. An Example of How to Use the DSPCLKRST_setup () Function to Initialize the DSP 10
Figure 6. ARM/DSP Public and Shared Peripherals 12
Figure 7. OMAP5910 GZG Microstar BGA ä Package (Bottom View) 12
Figure 8. Using the OMAP5910 Pin Configuration Utility to Configure the UART1 signals 14
Figure 9. Using the OMAP5910 Pin Configuration Utility to Configure the UART1 signals 14
Figure 10. Using the OMAP5910 Pin Configuration Utility to Configure the UART1 signals 15
Figure 11. Level 2 Interrupt Example 16
Figure 11. Level 2 Interrupt Example (Continued) 17
Figure 11. Level 2 Interrupt Example (Continued) 18
Figure 11. Level 2 Interrupt Example (Continued) 19
Figure 12. MPU Interrupt Handlers 23
Figure 13. Servicing the McBSP1 Transmit Ready Interrupt 24
Figure 14. Bootloader Build Flowchart 27
Figure 15. OUT2BOOT Format for DSP Code and Data 27
Figure 16. MPU Application Code (arm_boot) 28
Figure 17. Figure A-1. 2–Wire I2C Compatible Timing 33

Tables

Table 1. OMAP5910 System Frequencies 9
Table 2. DSP Subsystem Frequencies 11
Table 3. UART1 Pin Muxing Options 13
Table 4. Table A-1. Slave Address Selection 32
Table 5. Table A-2. AIC23 Control Register Addresses 33

1 Introduction

1.1 What to Expect From This Guide

The goal of this document is to help software engineers understand proper initialization of the OMAP5910 hardware platform. It will be of interest primarily to those who are developing low-level software that accesses the OMAP5910 hardware directly.

Section 1 focuses on the OMAP™ dual-core architecture. This section familiarizes the reader with OMAP™ vocabulary and highlights the structure and hierarchy of the platform.

Section 2 reviews the initialization procedures of all hardware components and peripherals of the MPU and DSP subsystems. **Unless otherwise instructed, any procedures listed should be followed in the order listed.**

Section 3 contains examples dealing with servicing Level 2 interrupts and DSP Boot processes.

Low-level functions and macros that are a part of the OMAP5910 Chip Support Library (CSL) are referenced in the module's description. These components are provided as an example. They are not necessarily efficient enough for a real application, but provide a good basis for understanding the hardware programming, and can be used as a starting point for new software development.

Note: References to operating systems are minimized. No knowledge of any specific operating system is required to understand the examples.

1.2 Global Architecture

The OMAP5910 device is a high-performance software and hardware development platform targeted for multimedia applications. It consists of:

- **DSP mega-module** (55x DSP core and subsystems) with internal program and data memory, instruction cache, DMA controller, and hardware accelerators.
- **ARM processor mega-module** (ARM925T) with memory management unit (MMU) and instruction and data cache.
- **Local bus interface** with memory management unit, multi-channel system DMA controller, peripherals (local and shared) that support glueless system interface, and connecting modules that facilitate communication between these mega-modules and system memory (external and internal).

The OMAP5910 platform consists of the MPU and DSP subsystems. Each of these subsystems consist of a mega-module and its standard peripherals. The ARM925T is the master of the entire platform and has access to the entire memory and I/O space of the DSP. Figure 1 illustrates the layout of the OMAP5910 platform.

- MPU subsystem includes:
  - ARM925T mega-module (ARMTDMI core)
  - Traffic controller
  - DSP MMU
  - System DMA controller
  - Three 32-bit timers
  - One 16-bit watchdog timer
  - Interrupt handler
  - Local bus interface
  - Three DPLLs and clock managers
  - Rhea bridges
  - LCD controller
  - API interface
  - Endianism conversion
  - Mailbox (shared)
  - Elastic buffers (55x DSP and ARM925T)
  - Test and configuration

- DSP subsystem includes:
  - C55x core
  - Three 32-bit timers
  - One 16-bit watchdog timer
  - Second-level interrupt handler
  - DSP interrupt interface (first-level interrupt)
  - One UART (shared)
  - Sixteen GPIO (shared)

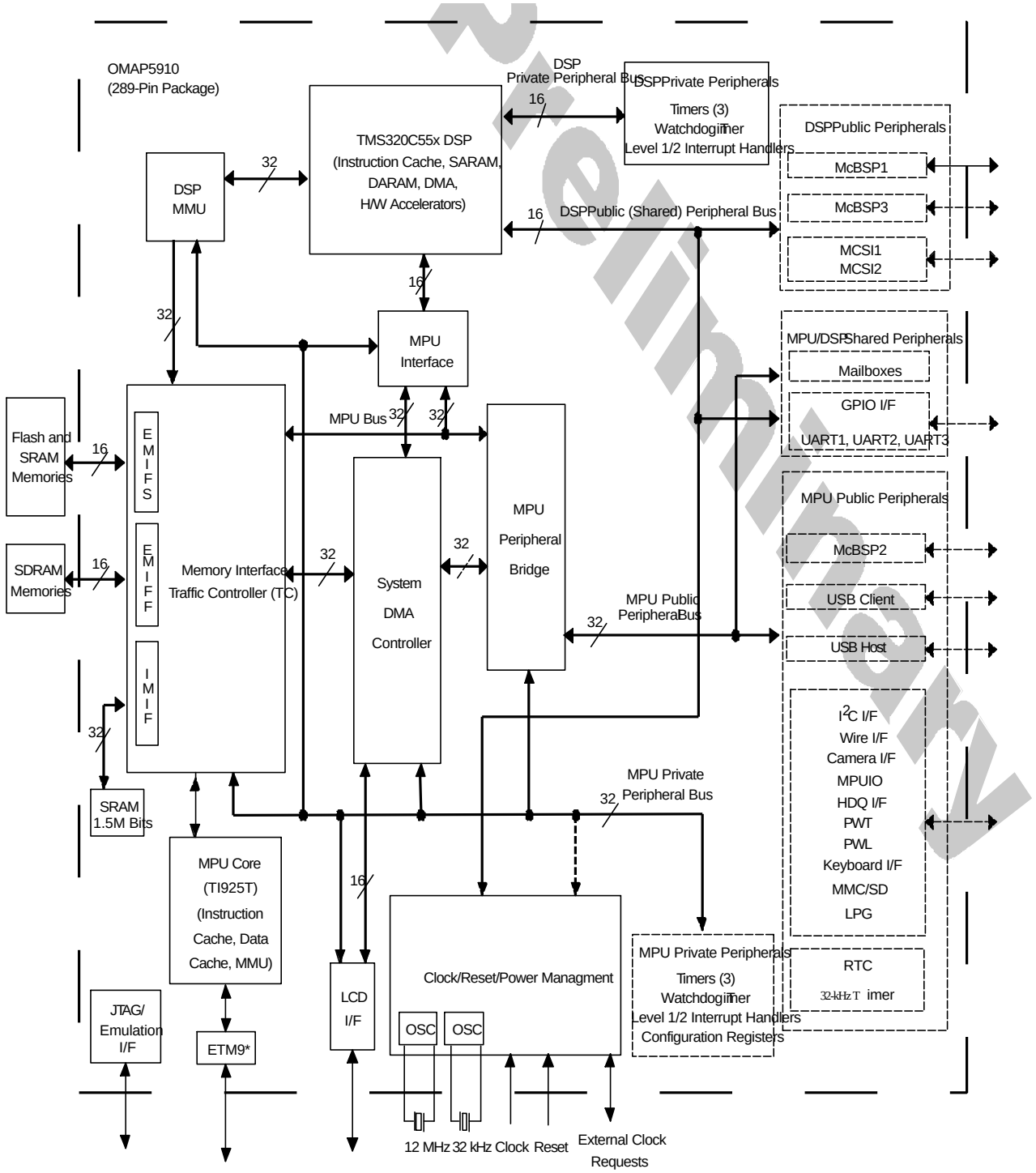


Figure 1. OMAP5910 Platform

2 System Initialization Procedure

Section 2 demonstrates the proper steps to initialize the ARM, DSP, and their associated peripherals, as well as instructions for configuring the corresponding frequencies at which they operate. Four functions from the OMAP5910 CSL library's PLL, CLKRST, and DSPCLKRST modules will be used to demonstrate this procedure:

- CLKRST_reset()
- PLL_setFreq()
- CLKRST_setupScalableMode()
- DSPCLKRST_setup()

Illustrations of these functions and their operations are provided.

2.1 Resetting the OMAP5910

Initially, the CLKRST_reset() function can be used to perform a global software generated reset.

Note: This step is optional and is not required if a power-on reset has just occurred.

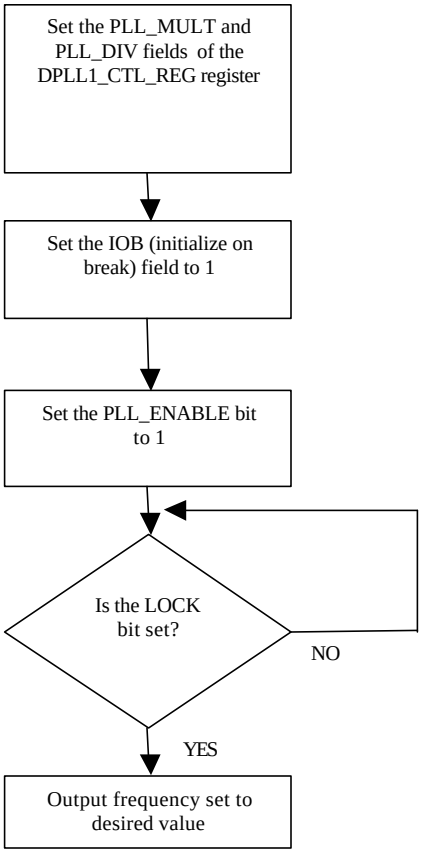
Figure 2 illustrates how the CLKRST_reset() function performs a global reset and the proper method for calling the CLKRST_reset function. After a global software reset is performed, the reset is released for the ARM925T and internal OMAP5910 peripherals, with the exception of the DSP Subsystem and all external peripherals (both ARM and DSP). The subsequent description of the CLKRST_setupScalableMode and DSPCLKRST_setup functions describe the procedure used to release the reset state for the DSP and external peripherals.

Figure 2. Using the CLKRST_reset() Function to Perform a Global Software-Generated Reset

2.2 Setting the Frequencies and Enabling the OMAP5910

- **Step 1:** Set the DPLL1 output frequency:

The PLL_setFreq() function (found in the OMAP5910 CSL PLL module) can be used to set the frequency of the three digital phase-locked-loops (DPLLS) contained in the OMAP5910 device. Since only “fully synchronous” and “synchronous scalable” modes are recommended (see the *OMAP5910 Technical Reference Manual* (SPRU602) for more details), we will configure the frequency of the DPLL1 and divide this value down for the various clock domains as needed. Figure 3 illustrates the process of the PLL_setFreq() function configuring the output frequency and lists example code used to call the function to set the output frequency to 150 MHz.




```
PLL_handle hPll;

hPll = PLL_open(PLL_DEV0,PLL_OPEN_RESET);

PLL_setFreq(hPll,25,1) /*set output
                        frequency at
                        150 MHz */
```

Figure 3. Using the PLL_setFreq() Function to Set the Output Frequency of the DPLL1 to 150 MHz

The PLL_setFreq() function begins by setting the PLL_MULT and PLL_DIV fields of the DPLL1_CTL_REG register with the values that were passed to it as arguments (25 and 1, respectively, in our example). Next, it sets the initialize on break (IOB) field in the aforementioned register to 1. This ensures that the DPLL switches to bypass mode and starts a new locking sequence if the PLL core indicates that it lost the lock. Next, the function starts the locking sequence of DPLL1 by setting the PLL_ENABLE bit to 1. Finally, it continuously polls the LOCK bit to see if the DPLL has reached the desired synthesized clock frequency and exits when the bit is set to 1.

In the example provided with Figure 3, the value of 25 for the PLL_MULT value and 1 for the PLL_DIV is used, resulting in an output frequency of 12 X (25/2), or 150 MHz.

- **Step 2:** Setting the ARM side clock domains and enabling the DSP:
  Next, the CLKRST_setupScalableMode() function (found in the CSL CLKRST module) can be used to initialize the ARM clock domain in the synchronous scalable mode. As mentioned earlier, this is one of the two recommended clocking modes for the OMAP5910 device. In this clocking scheme, the ARM, DSP and Traffic Controller domains are synchronous, but they run at a different clock speeds. This function also enables the ARM peripherals and their corresponding clocks, as well as the DSP.

  Figure 4 illustrates an example of the steps used to call CLKRST_setupScalableMode() and the actions the function takes to accomplish the aforementioned tasks.

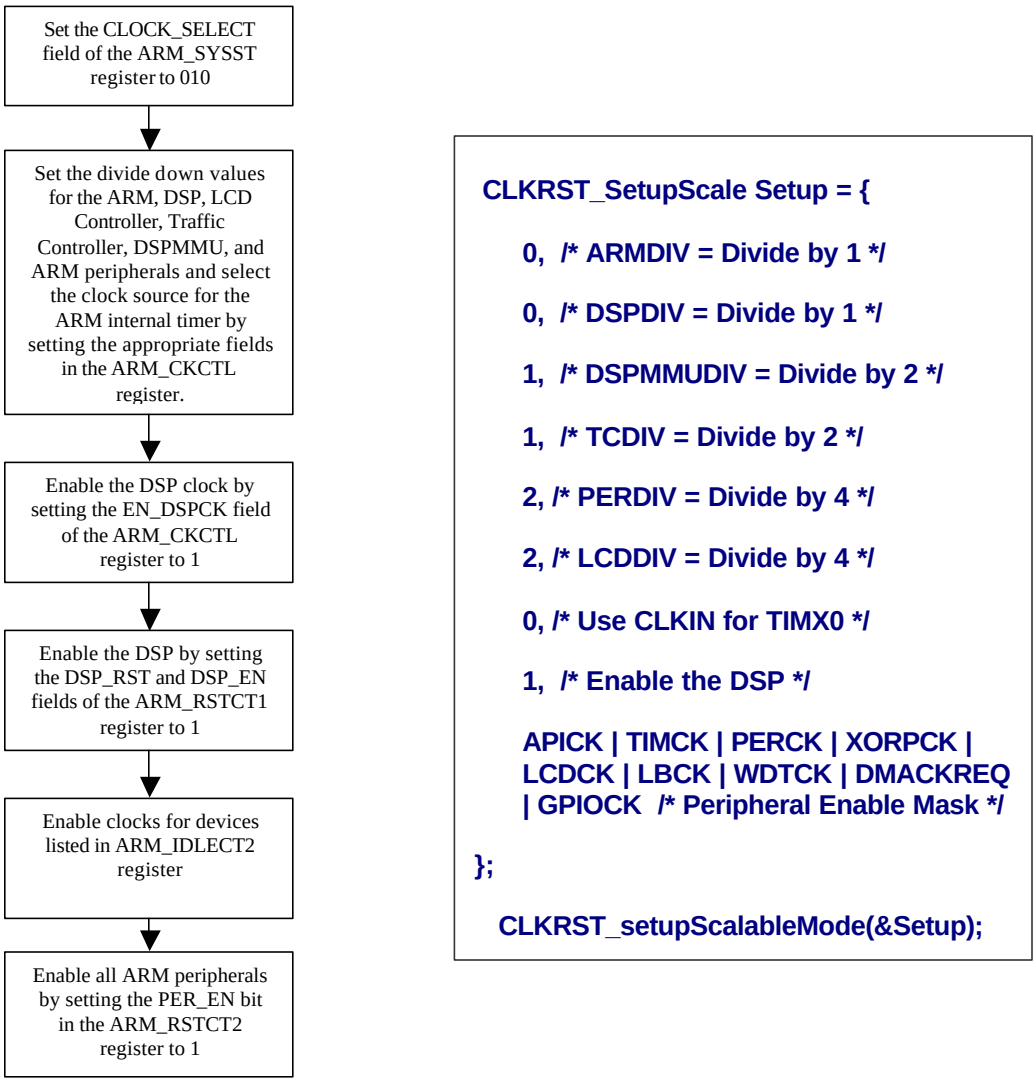


Figure 4. An Example of How to Use the CLKRST_setupScalableMode() Function to Initialize the OMAP5910 Device

The sole argument to the CLKRST_setupScalableMode() function is a structure (Setup) that contains the various parameters used to initialize the device. Initially, the function configures the OMAP5910 device to use synchronous scalable clocking mode by setting the CLOCK_SELECT field in the ARM_SYSST register to 010b. It then proceeds to set the divide down values used to generate the ARM, DSP, DSPMMU, Traffic Controller, ARM peripheral, and LCD controller clocks by using the first six members of the Setup structure and placing them in the corresponding fields of the ARM_CKCTL register. Note that placing a 0 as a divide down value in the field denotes an "actual" divide by 1, 1 (01b) denotes divide by 2, and a value of 2 (10b) denotes divide by 4. If we assume that the DPLL1 output frequency has been set to 150 MHz (as done in the previous step), then the frequencies for each of the components would result as follows:

Table 1. OMAP5910 System Frequencies

Device	Field in ARM_CKCTL Register	Setup structure member value	Actual Divide Value	Resulting Frequency
ARM	ARMDIV	0	1	150 MHz
DSP	DSPDIV	0	1	150 MHz
DSPMMU	DSPMMUDIV	1	2	75 MHz
Traffic Controller	TCDIV	1	2	75 MHz
ARM Peripheral Clock	PERDIV	2	4	37.5 MHz
LCD Controller Clock	LCDDIV	2	4	37.5 MHz

The next member in the structure is used to set the input reference clock as the clock source for the ARM internal timer. This is done by setting the ARM_TIMX0 field of the ARM_CKCTL register to 0.

The function then proceeds to enable the DSP clock (while the DSP itself is still in a reset state) by setting the EN_DSPCK field of the same register to 1. After the DSP clock has been enabled, the function checks the next member of the structure to determine if the user wants to enable the DSP. Since the value is 1, it proceeds to enable the DSP by setting both the DSP_RST and DSP_EN fields of the ARM_RSTCT1 register to 1.

The final member of the Setup structure (APICK | TIMCK | PERCK | XORPCK | LCDCK | LBCK | WDTCK | DMACKREQ | GPIOCK) is an OR mask which allows the user to enable the clocks for the devices in listed in the ARM_IDLECT2 register. In this example, we chose to enable the ARM Port Interface clock (APICK), ARM Timer clock (TIMCK), ARM peripheral clock (PERCK), OS Timer and CLKIN reference peripheral clock (XORPCK), LCD Controller clock (LCDCK), Local Bus clock (LBCK), MPU Watchdog Timer clock (WDTCK), System DMA Clock Request (DMACKREQ), and the MPU GPIO clock (GPIOCK) by setting the EN_APICK, EN_TIMCK, EN_PERCK, EN_XORPCK, EN_LCDCK, EN_LBCK, EN_WDTCK, DMACK_REQ, and EN_GPIOCK fields to 1. Finally, after enabling these clocks, the function enables all of the ARM peripherals by setting the PER_EN bit of the ARM_RSTCT2 register to 1.

- **Step 3:** Setting the DSP side clock domains and enabling the DSP peripherals: The DSPCLKRST_setup() function is used to initialize the various DSP clocks (UART, GPIO, etc) to the desired values as well as to enable DSP peripherals. The sole argument to this function is a structure (DSPSetup) which contains the various parameters used to initialize the DSP subsystem. Figure 5 illustrates an example of how to call the function and the actions the function takes to accomplish the aforementioned tasks.

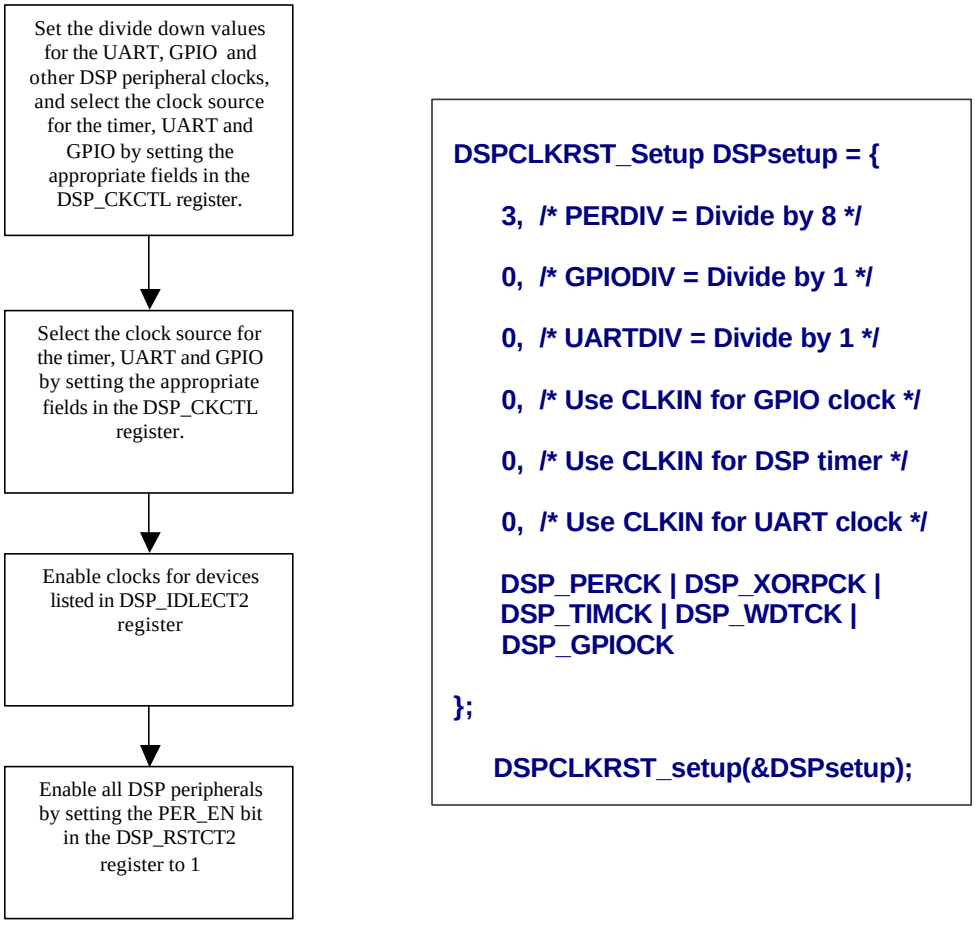


Figure 5. An Example of How to Use the DSPCLKRST_setup () Function to Initialize the DSP

Initially, the DSPCLKRST_setup () function configures the divide down values used to generate the DSP peripheral, GPIO, and UART clocks by using the first three members of the structure and placing them in the corresponding fields of the DSP_CKCTL register. If we assume that DPLL1 output frequency has been set to 150 MHz (as configured previously), then the frequencies for each of the components would result as follows:

Table 2. DSP Subsystem Frequencies

Device	Field in ARM_CKCTL register	Divide Down Value	Resulting Frequency
DSP Peripherals	PERDIV	8	18.75 MHz
UART	UARTDIV	4	37.5 MHz
GPIO	GPIODIV	4	37.5 MHz

The next three members of the structure direct the function to set the clock source for the GPIO, Timer, and UART respectively. These arguments can be of one of two forms: 0 (clock source and value will be determined by input reference clock), and 1 (clock source will be DPLL1, and value will be determined by the corresponding DIV value set in the previous step). Since we have selected the value of 0 for all three, the divide down values that were previously specified will be ignored and the input reference clock value (12 MHz) will be used as the clock source for these three peripherals.

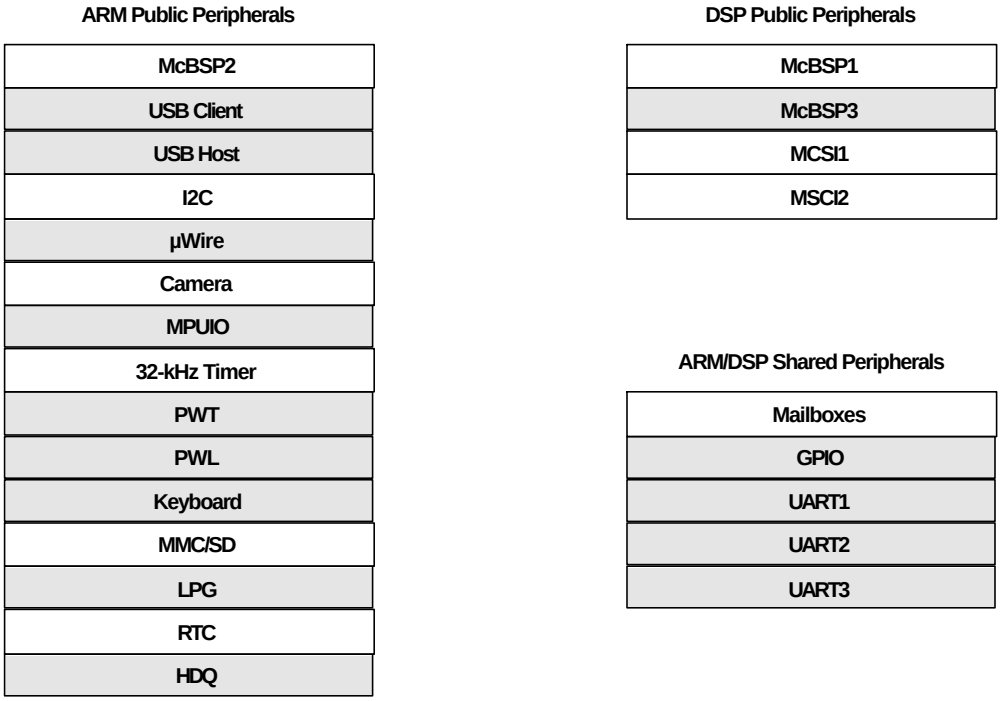
The next argument to the function (DSP_PERCK | DSP_XORPCK | DSP_TIMCK | DSP_WDTCK | DSP_GPIOCK) is an OR mask which allows the user to enable the clocks for those devices in listed in the DSP_IDLECT2 register. In this example, we chose to enable the DSP peripheral clock (DSP_PERCK), external reference clock (DSP_XORPCK), DSP Timer clock (DSP_TIMCK), Watchdog Timer clock (DSP_WDTCK) and GPIO clock (DSP_GPIOCK) by setting the EN_PERCK, EN_XORPCK, EN_TIMCK, EN_WDTCK, and EN_GPIOCK fields to 1. Finally, after enabling these clocks, the function enables all of the DSP peripherals by setting the PER_EN bit of the DSP_RSTCT2 register to 1.

2.3 Pin Muxing Configuration

Due to the complexity of the device, pin multiplexing on the OMAP5910 can be quite difficult. The goal of this section is to present a straightforward procedure that will allow the user to configure the available pins for their particular application's requirements.

2.3.1 Peripheral List

Figure 6 shows a list of the public and shared peripherals on the ARM and DSP. **The shaded peripherals are those whose corresponding pins require configuration in order to achieve full functionality**. That is, some of the signals associated with the shaded peripherals are not enabled by default. The following sections will explain how to configure their corresponding pins for proper use.



Note: The shaded peripherals are those whose associated pins must be configured (multiplexed) in order to achieve full functionality.

Figure 6. **ARM/DSP Public and Shared Peripherals**

Figure 7 illustrates the ball locations for the 289-pin ball grid array (BGA) package. This diagram will be used in conjunction with pin muxing utility that will presented in the next section to help the user determine the proper pin muxing configuration for their application.

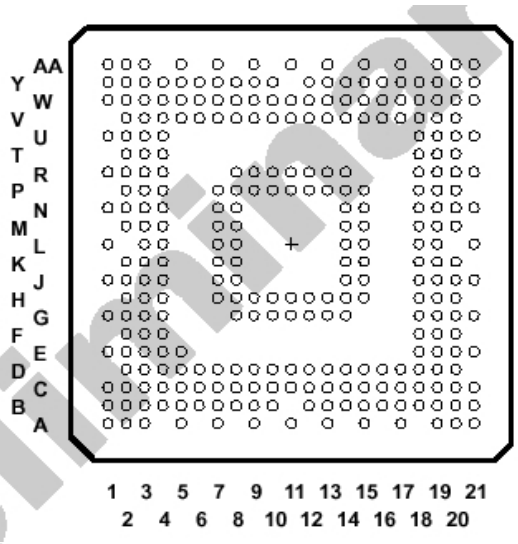


Figure 7. OMAP5910 GZG Microstar BGA ä Package (Bottom View)

As an example, Table 3 lists the different options for configuring the UART1 peripheral (see the OMAP5910 datasheet for more details). Each signal for the UART1 is shown along with the corresponding ball number(s) on which the signal can be enabled. For each ball number on which the signal can be enabled, the corresponding register(s) which controls the muxing of that pin, along with the bit field and value that must be set to mux that signal onto that particular pin is shown.

Table 3. UART1 Pin Muxing Options

Signal Name	Description	Ball	Register	Bit Field	Value to Enable
UART1.TX	UART1 transmit data	Y14	FUNC_MUX_CTRL_9	[23:21]	001
UART1.RX	UART1 receive data	V14	Enabled by default	N/A	N/A
UART1.CTS	UART1 clear-to-send	R14	Enabled by default	N/A	N/A
UART1.RTS	UART1 request-to-send	AA15	FUNC_MUX_CTRL_9	[14:12]	001
UART1.DTR	UART1 data-terminal-ready	W21	FUNC_MUX_CTRL_8	[5:3]	010
		Y13	FUNC_MUX_CTRL_A	[2:0]	010
UART1.DSR	UART1 data-service-ready	U18	FUNC_MUX_CTRL_8	[2:0]	010
		R13	FUNC_MUX_CTRL_9	[29:27]	010

While this table may make the pin multiplexing configuration process seem trivial, there are several other balls in the OMAP5910 device on which up to five different signals can be muxed. This makes the configuration process extremely difficult for those who are attempting to use a multitude of peripherals.

2.3.2 Pin Multiplexing Utility

In order to simplify the pin multiplexing configuration procedure, a GUI based utility has been created. The OMAP5910 Pin Configuration Utility is a standalone application that facilitates the configuration of pins and multiplexing of signals for the OMAP5910 peripherals. In addition, the user can enable/disable the pullups/pulldowns for those pins that have that capability. The following example demonstrates how to use the utility properly.

Figures 8, 9, and 10 are screenshots that were taken while using the utility to configure the UART1 peripheral pins for the Innovator™ development platform. Since both the UART1.TX and UART1.RTS signals are required on balls Y14 and AA15 respectively (refer to the Innovator™ schematic for more details), and neither are available by default on those balls (refer to Table 3), the utility is used to calculate the proper values for the FUNC_MUX_CTRL registers which control the muxing of those particular signals. In the first screen of the utility (Figure 8), the peripherals of interest are selected:

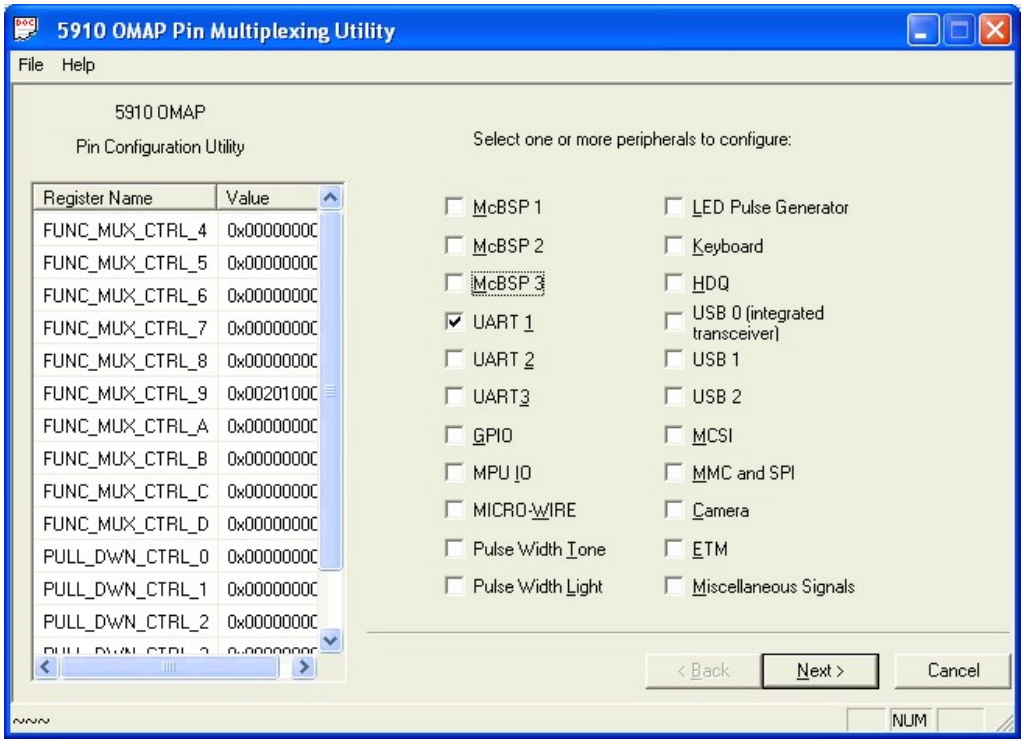


Figure 8. Using the OMAP5910 Pin Configuration Utility to Configure the UART1 signals

After clicking the “Next” button, the utility automatically calculates the various possibilities for each of the signals for the UART1 peripheral and presents them on the next screen (Figure 9).

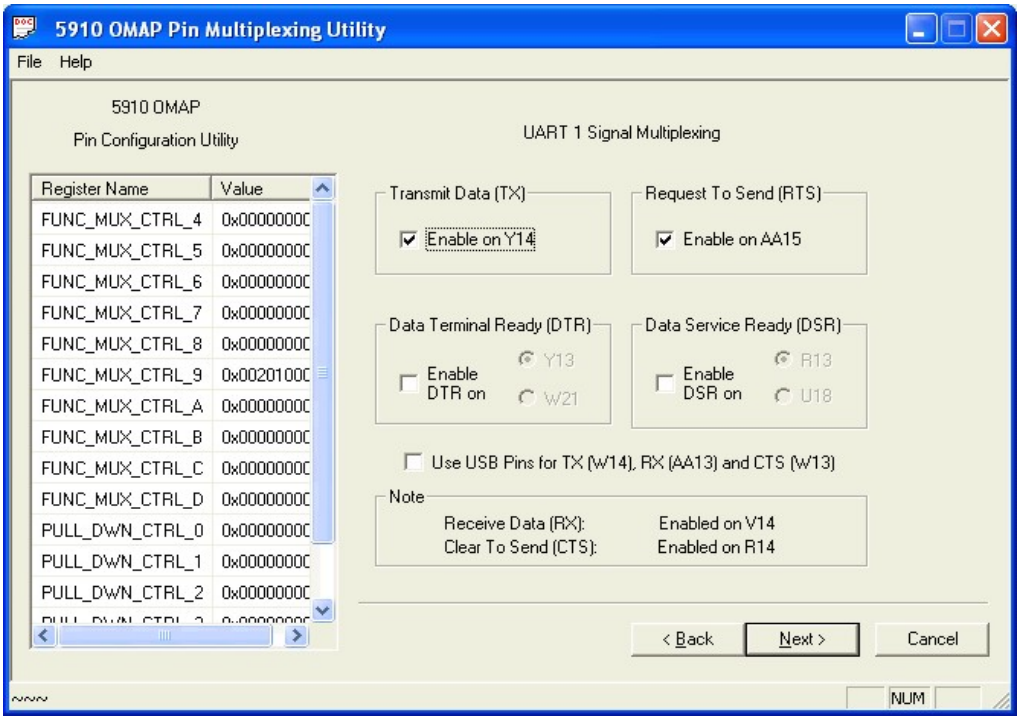


Figure 9. Using the OMAP5910 Pin Configuration Utility to Configure the UART1 signals

The desired balls for the UART1.TX and UART1.RTS signals (Y14 and AA15 respectively) are then chosen and the “Next” button is pressed. In the final screen, the user is presented with a summary of the options selected (Figure 10).

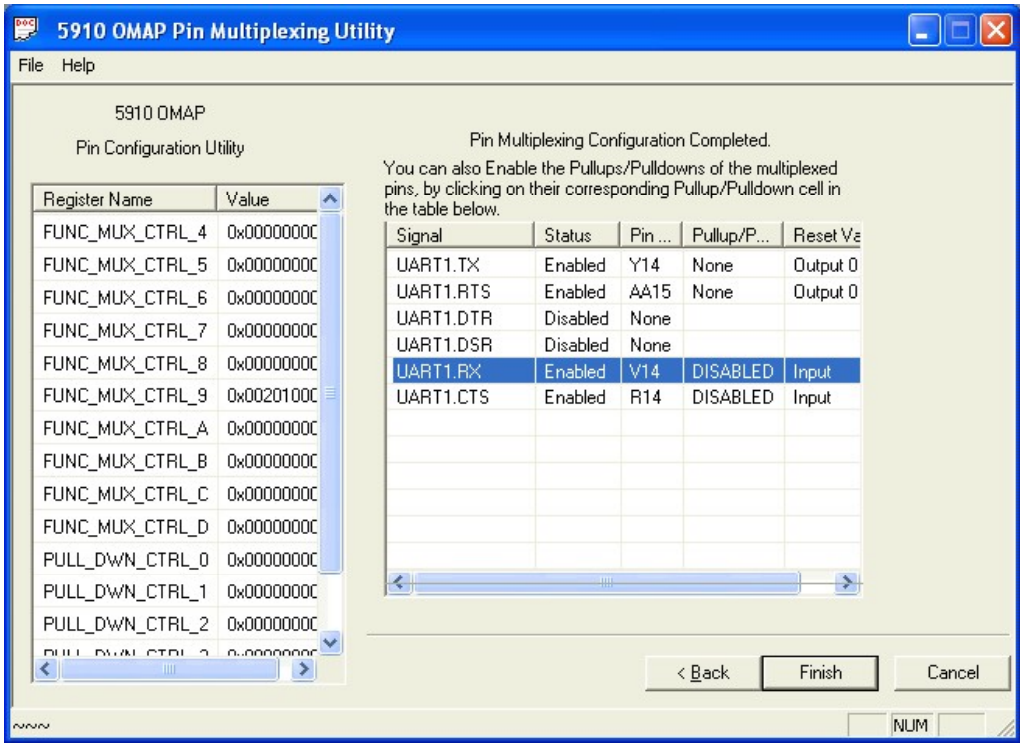


Figure 10. Using the OMAP5910 Pin Configuration Utility to Configure the UART1 signals

After clicking the “Finish” button, a dialog box is presented that allows the user to save the settings created as an include file. This include file contains the appropriate value for each of the FUNC_MUX_CTRL and PULL_DWN_CTRL registers. In this example, the only configuration register that was modified was the FUNC_MUX_CTRL_9 register, and the include file produced the following:

```
#define  FUNC_MUX_CTRL_9          0x00201000    //FUNC_MUX_CTRL_9 register value
```

Note that bit fields [23:21] and [14:12] of this register are both set to 001b, which is consistent with the requirements of Table 3. This value can then be written to the FUNC_MUX_CTRL_9 register by using the following macro in the OMAP CSL library:

```
Example:          CHIP_RSET(CTRL_9, FUNC_MUX_CTRL_9);
```

2.4 Configuring the OMAP5910 Device in Native Mode

In order to retain backwards compatibility, the OMAP5910 device configuration registers are software compatible with previous prototype devices at reset. That is, many of the peripherals and additional capabilities of the device are disabled at power on. Writing the 32-bit value 0x0000EAEFh to the compatibility mode control 0 register (COMP_MODE_CTRL_0, Address = 0xFFFE100Ch) configures the device in native mode and allows access to the multiplexing and device configuration registers found at address 0xFFFE1000h and above.

Use caution when enabling the OMAP5910 native mode. All OMAP5910 configuration registers reset to 0x0000h at power on reset. It is advisable to follow this procedure before enabling OMAP5910 native mode:

- Determine the desired values for each OMAP5910 configuration register.
- Program the desired values by writing to the appropriate register.
- Program the COMP_MODE_CTRL_0 register to 0x0000EAEFh.
- The desired modes are now active.

This procedure allows the user to select all OMAP5910 configuration settings with a series of register writes before enabling all of the modes simultaneously.

Example: CHIP_RSET(COMP_CTRL_0, 0x0000EAEF); /* Set Native mode */

3 Examples

3.1 Level 2 Interrupt Example

Figure 11 provides an example of how to use the OMAP5910 I2C and MCBSP1 peripherals to configure a TLV320AIC23 codec and play a tone. A polling based CSL I2C function will be used to configure the codec's settings (sampling frequency, volume, etc), while an interrupt based MCBSP1 function will be used to transmit data to the codec. This example can be run on Innovator™ development platform.

```
#include <csl_i2c.h>
#include <csl_clkrst.h>
#include <csl_mcbsp.h>
#include <csl_pll.h>
#include <csl_dspclkrst.h>
void write_isr(void);

CLKRST_SetupScale Setup = {
   0,  /* ARMDIV = Divide by 1 */
   0,  /* DSPDIV = Divide by 1 */
   1,  /* DSPMMUDIV = Divide by 2 */
   1,  /* TCDIV = Divide by 2 */
   2, /* PERDIV = Divide by 4 */
   2, /* LCDDIV = Divide by 4 */
   0, /* Use CLKIN for TIMX0 */
   1,  /* Enable the DSP */
   APICK | TIMCK | PERCK | XORPCK | LCDCK | LBCK | WDTCK | DMACKREQ  | GPIOCK
  /* Peripheral Enable Mask */
  };
```

CLKRST Setup Structure

Figure 11. Level 2 Interrupt Example

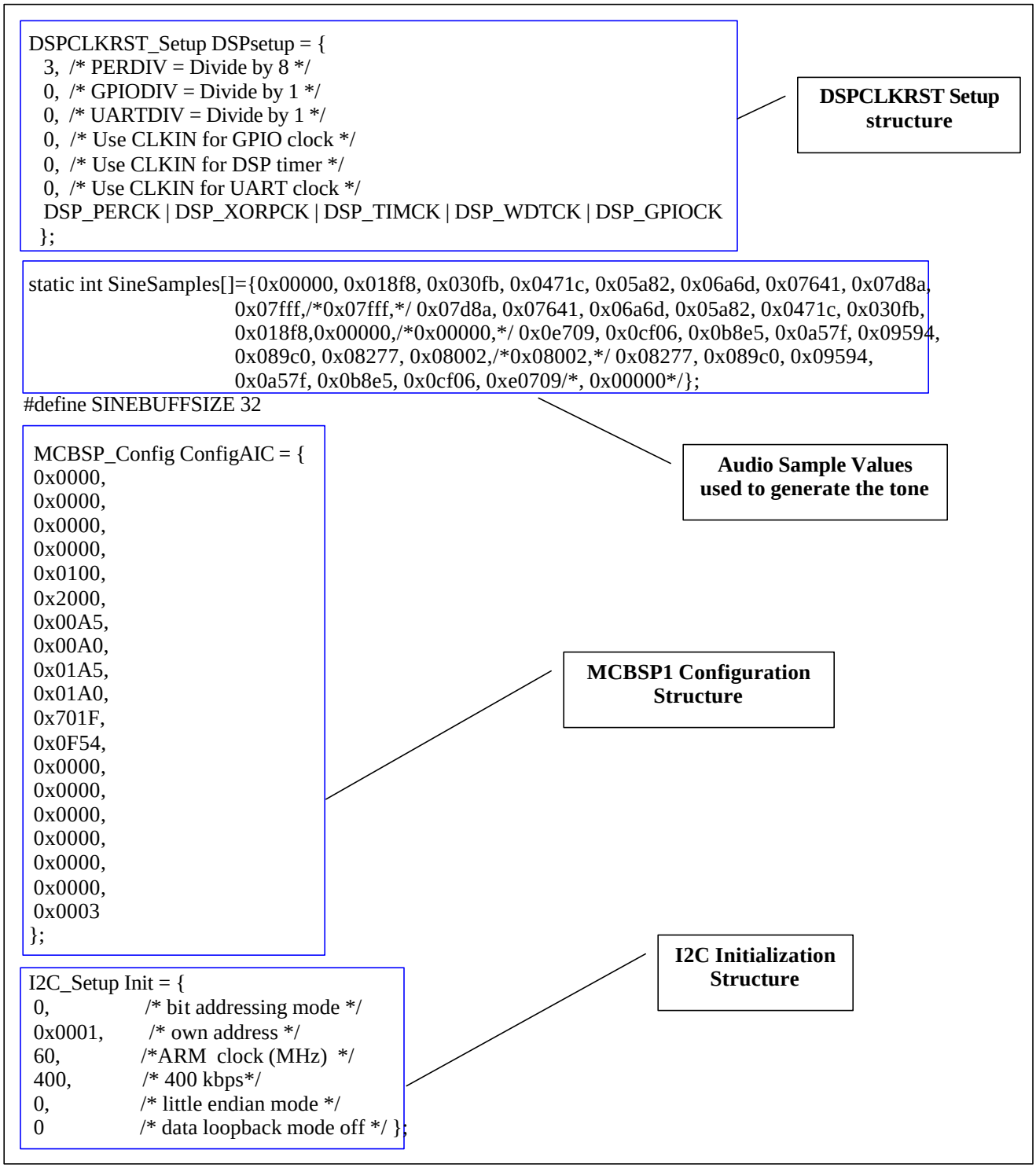


Figure 11. Level 2 Interrupt Example (Continued)

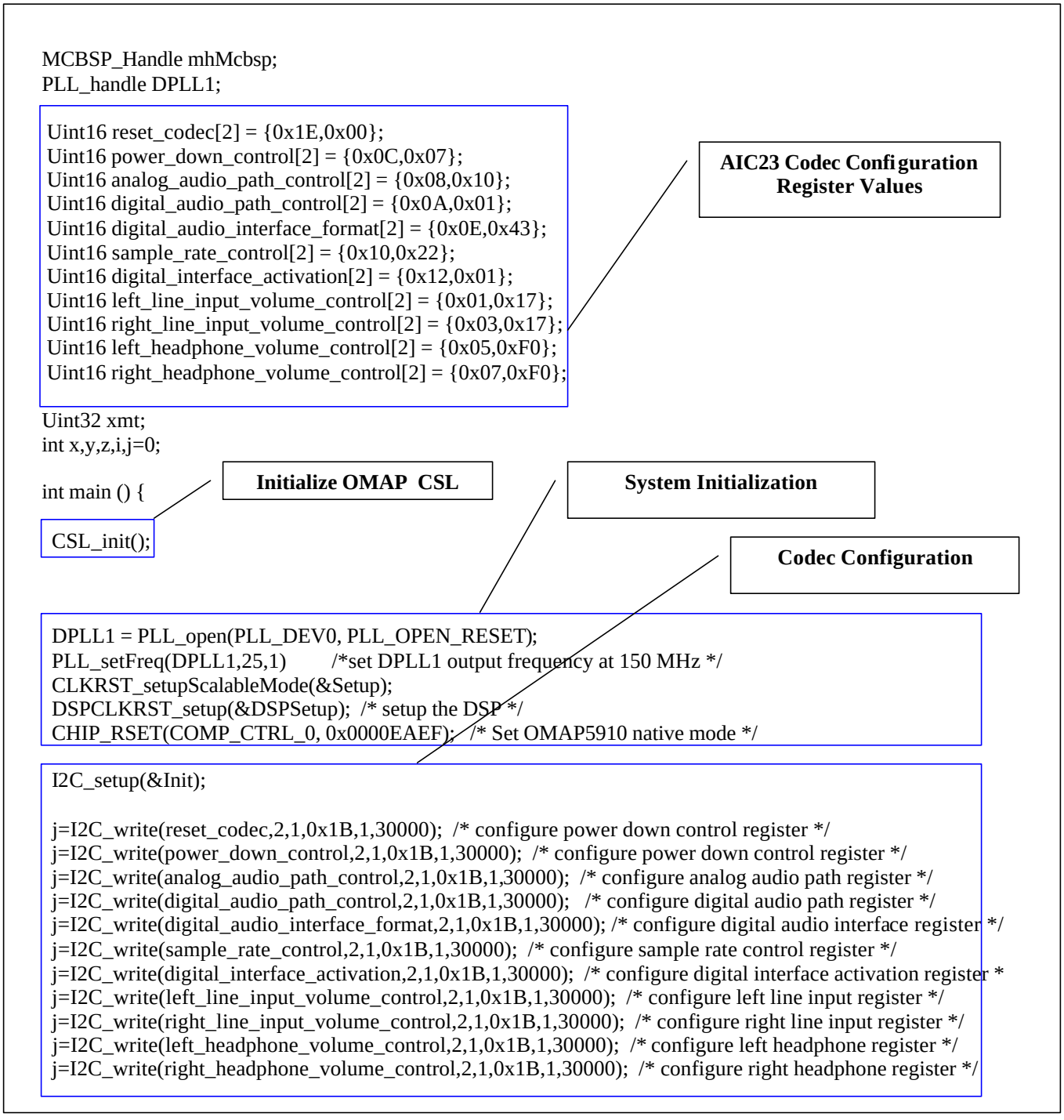


Figure 11. Level 2 Interrupt Example (Continued)

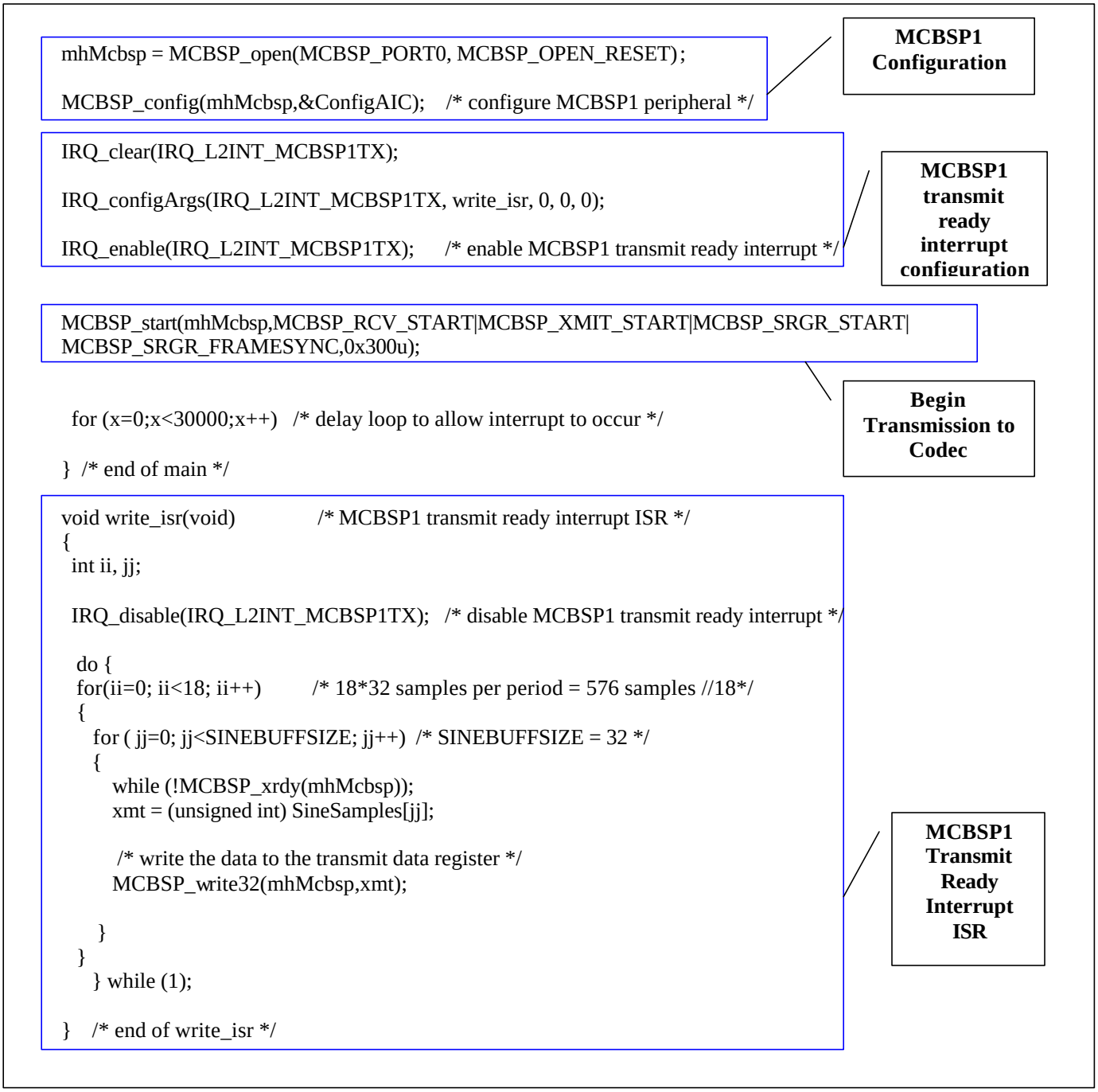


Figure 11. Level 2 Interrupt Example (Continued)

The example code shown in Figure 11 begins by initializing the various configuration structures and arrays that will be used in the following sections. At the beginning of main(), the example initializes the OMAP Chip Support Library (CSL) by calling the CSL_init() function. **This function initializes the CSL data structures and hooks up the Level 1 interrupt handlers — it must be called if CSL is to be used.**

Next, the code performs OMAP5910 system initialization by using CSL functions described in sections 2.2 and 2.3. These functions configure the ARM to run at 150 MHz, and the ARM peripherals to run at 37.5 MHz (see Table 1).

3.1.1 Codec Configuration Using the I2C Peripheral (Polling Example)

This section will describe the example code shown in Figure 11 that is used to configure the codec registers by the use of the I2C peripheral. A step-by-step guide to configuring the AIC23 codec registers using the I2C interface is given.

Step 1 AIC23 Control Register Value Definition

First, it is necessary to set up the eleven arrays that will be used to configure the eleven registers found on the AIC23 codec.

In the first array, configure the reset register to take the device out of reset. Referring to Figure A–1 (in Appendix A) notice that the address of the register and the values of the fields it contains must be concatenated together into two bytes of data. Next, the register address and field values are sent in S-A-D-P (Start-Address-Data-Stop) mode to the codec. If the register address is 0001111, then the first seven bits of the first byte sent out must be 0001111. The registers on the codec have nine fields; therefore, the LSB of the first byte sent counts as the value for field number nine, and the second byte counts as the value for fields 1–8. Taking notice of the reset register description, writing all 1's to the register will place it in reset mode. Therefore, in order to take the device out of reset mode, you must write zero values to the device's nine fields. For simplicity, write all 0's to the register. With the address being the first seven bits, you have: 0001111000000000, or 0x1E and 0x00.

Uint16 reset_codec[2] = {0x1E,0x00};

The second array is set up in a similar manner. The power down control register has an address of 0000110. Field nine in the power down control register is reserved; therefore, place a 0 in that field and turn the power on (field 8 = 0), clock on, oscillator on, outputs on, DAC on, ADC off, microphone input off, and line input off. Again, with the address being the first seven bits of the two bytes to be transferred, you have: 0000110000000111, or 0x0C and 0x07.

Uint16 power_down_control[2] = {0x0C,0x07};

Here are the other nine register field values:

- Analog Audio Path Control:
  Sidetone enable off, sidetone attenuation 6db, DAC select on, bypass off, input select ADC line, microphone mute off, microphone boost 0db.
  Uint16 analog_audio_path_control[2] = {0x08,0x10};
- **Digital Audio Path Control:**
  DAC soft mute off, deemphasis disabled, ADC high pass filter on.
  Uint16 digital_audio_path_control[2] = {0x0A,0x01};

- **Digital Audio Interface Format:**
  Master mode, input length 16 bits.
  Uint16 digital_audio_interface_format[2] = {0x0E,0x43};
- **Sample Rate Control:**
  Sample rate 44.1Khz, clock mode master.
  Uint16 sample_rate_control[2] = {0x10,0x22};
- **Digital Interface Activation:**
  Active Interface.
  Uint16 digital_interface_activation[2] = {0x12,0x01};
- **Left Line Input Volume Control:**
  Simultaneous volume mute on, left line input mute off, left line input volume.
  Uint16 left_line_input_volume_control[2] = {0x01,0x17};
- **Right Line Input Volume Control:**
  Simultaneous volume mute on, right line input mute off, right line input volume.
  Uint16 right_line_input_volume_control[2] = {0x03,0x17};
- **Left Headphone Volume Control:**
  Left-right headphone simultaneous volume mute on, left zero crossing detect on, left headphone volume.
  Uint16 left_headphone_volume_control[2] = {0x05,0xFF};
- **Right Headphone Volume Control:**
  Right-left headphone simultaneous volume mute on, right zero crossing detect on, right headphone volume.
  Uint16 right_headphone_volume_control[2] = {0x07,0xFF};

Step 2 I2C Initialization

Begin the next step by initializing various parameters that will be used during the transfer. The initialization structure (Figure 11) that is passed to the I2C_init function to initialize the device has the following values:

```
I2C_Setup Init = {

      0,              /* 7 bit addressing mode */
      0x0001,         /* own address */
      150,            /* ARM clock (MHz)  */
      400,            /* 400 kbps*/
      0,              /* little endian mode */
      0               /* data loopback mode off */
};
```

Call the initialization function with the Init structure as an argument to initialize the I2C device.

```
I2C_setup(&Init); /* configure the I2C */
```

Step 3 Write Operation

Referring again to Figure A–1 it is necessary to use S-A-D-P mode to control the registers. Call the I2C_write function with the reset array, the length (which will always be fixed at two), master mode on, slave address 0x1B (refer to table A-1), S-A-D-P mode, and a timeout of 30000 to configure the reset register.

```
j=I2C_write(reset,2,1,0x1A,1,30000); /* configure reset register */
```

Configure the other registers using the same format:

```
/* configure power down control register */
j=I2C_write(power_down_control,2,1,0x1A,1,30000);

/* configure analog audio path register */
j=I2C_write(analog_audio_path_control,2,1,0x1A,1,30000);

/* configure digital audio path register */
j=I2C_write(digital_audio_path_control,2,1,0x1A,1,30000);

/*configure digital audio interface register */
j=I2C_write(digital_audio_interface_format,2,1,0x1A,1,30000);

/* configure sample rate control register */
j=I2C_write(sample_rate_control,2,1,0x1A,1,30000);

/*configure digital interface activation register*/
j=I2C_write(digital_interface_activation,2,1,0x1A,1,30000);

/* configure left line input register */
j=I2C_write(left_line_input_volume_control,2,1,0x1A,1,30000);

/* configure right line input register */
j=I2C_write(right_line_input_volume_control,2,1,0x1A,1,30000);

/* configure left headphone register */
j=I2C_write(left_headphone_volume_control,2,1,0x1A,1,30000);

/* configure right headphone register */
j=I2C_write(right_headphone_volume_control,2,1,0x1A,1,30000);
```

For more information on the I2C_write function, please refer to the *Programming the TMS320VC5509 I2C Peripheral App Note* (SPRA785).

3.1.2 Transferring Data Using the McBSP1 Peripheral (Level 2 Interrupt Example)

This section will describe the example code that uses the MCBSP1 transmit ready interrupt to send data to the codec and generate a tone (Figure 11). This code is meant to serve as an example of how a Level 2 interrupt should be serviced. The proper procedure used to handle this interrupt will be shown, and will be followed by a description of the MCBSP1 peripheral configuration and the data transferred to the codec to generate the tone.

The OMAP5910 device has two layers of MPU interrupt handlers, as shown in Figure 12. If an unmasked MCBSP1 transmit ready interrupt (IRQ 12 in Level 2) occurs on the level 2 interrupt handler, it asserts IRQ_0 of the level 1 interrupt handler, which in turn generates either an IRQ or FIQ to the MPU depending on a configurable value in the corresponding interrupt level register (ILR12).

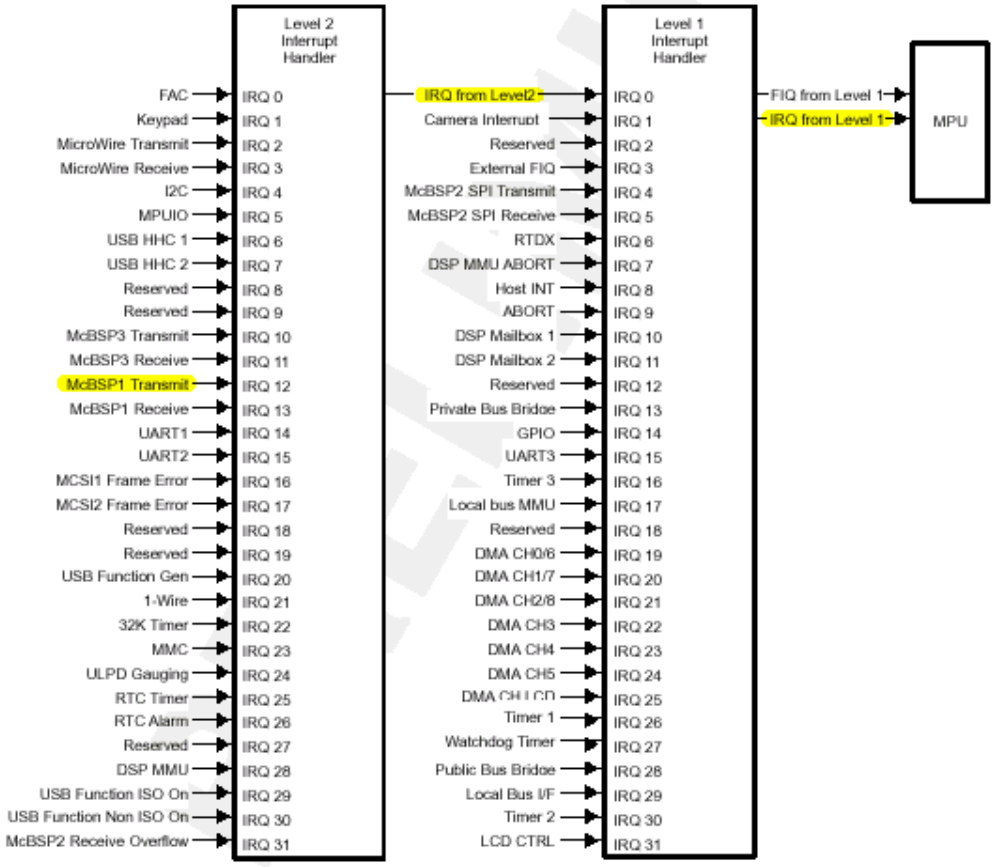


Figure 12. MPU Interrupt Handlers

Figure 13 demonstrates the proper procedure to set up and handle the MCBSP1 transmit ready interrupt and the corresponding CSL functions which perform these actions.

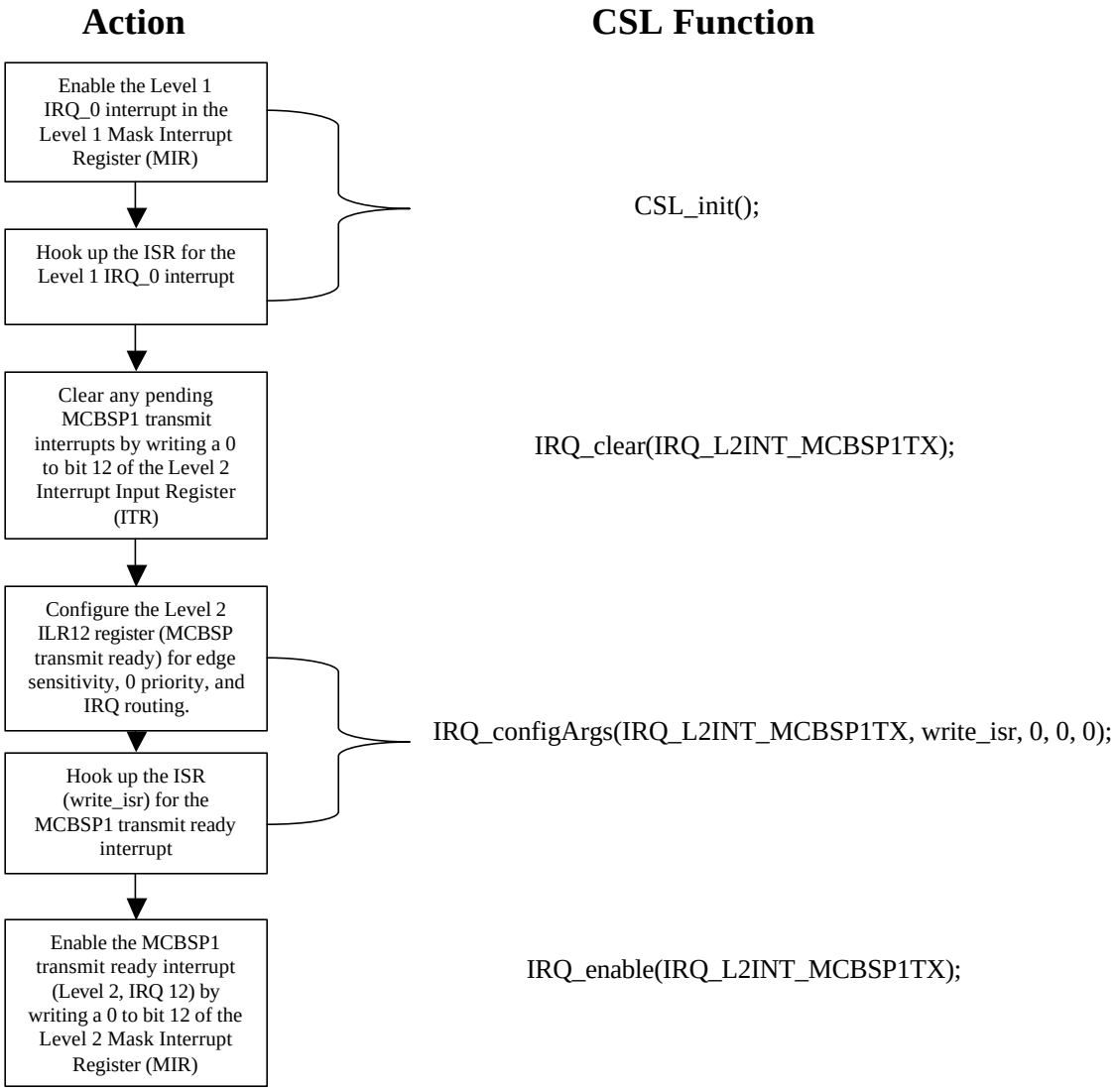


Figure 13. Servicing the McBSP1 Transmit Ready Interrupt

The first step involves enabling the Level 1 IRQ_0 interrupt, which, as mentioned previously, is asserted when any Level 2 interrupt occurs. This is done by writing a 0 to bit 0 of the Level 1 Mask Interrupt Register (MIR).

Next, the corresponding ISR for the Level 1 IRQ_0 interrupt should be “hooked.” In order to properly service the MCBSP1 Level 2 interrupt this interrupt service routine must be written in such a way that it reads the Level 2 SIR_IRQ_CODE register and calls the corresponding Level 2 ISR. This Level 1 IRQ_0 ISR, although not shown here, is implemented in the CSL IRQ_setupHandler() function, which is called within the CSL_init() function.

As a good programming practice, any pending MCBSP1 transmit ready interrupts should be cleared before the transmit process begins. This is done by writing a 0 to bit 12 of the Level 2 Interrupt Input Register (ITR).

Next, the MCBSP1 transmit ready interrupt's corresponding Interrupt Level Register (Level 2 ILR12) must be configured. Since this particular interrupt is an edge triggered interrupt (does not remain asserted, and gets cleared upon reading the SIR_IRQ_CODE register or by writing a 0 to the corresponding field in the ITR register), the SENS_EDGE field of the Level 2 ILR12 register must be set to 0. In addition, the priority of the interrupt must be configured – in this example it is defined as 0 (highest level), since it is the only interrupt of interest. Finally, the routing of the interrupt to either the IRQ or FIQ on the MPU also needs to be configured. In this example the MCBSP1 transmit ready interrupt is routed to the IRQ. Therefore, a 0 is written to the FIQ field of ILR12.

The ISR for the MCBSP1 transmit ready interrupt, write_isr, is then "hooked." This function will be used to write the transmit data to the MCBSP1 DXR registers. Finally, the MCBSP1 transmit ready interrupt is enabled by writing a 0 to bit 12 of the Level 2 Mask Interrupt Register.

Now that the proper procedure for servicing a Level 2 interrupt has been demonstrated, the example code used to transmit data using MCBSP1 peripheral along with the method for generating the audio sample values will be shown.

The data sent to the codec are sampled values of a sine wave (tone) in Q15 format.

These values (shown in Figure 11) are determined by using the following procedure:

Step 1 Pick an arbitrary frequency, from 300 Hz to 3 kHz (this allows a tone to be heard without distortion).

Step 2 For the purposes of this example, select the frequency 1.378 kHz. Next, use the following formula to generate samples in decimal format:

$$m[n] = Sin\left(\frac{2\pi f_{in}}{f_s}\right)$$

where *m[n]* is the sampled value, f_{in} is the selected frequency (1.378 kHz), and f_s is the sampling frequency of the codec (the codec was configured for a sampling frequency of 44.1 kHz).

Step 3 After obtaining the sample value, convert to the Q15 format by multiplying by 32767 and convert the resulting value into a hexadecimal number.
For example:

n = 0:
m[n] = 0.0
m[n] * 32767 = 0.0
= 0x00000

n = 1:
m[n] = .19507.
m[n] * 32767 = 6391.95223
6391.95223 = 0x018F8.

n = 2:
m[n] = .38265.

m[n] * 32767 = 12538.30973
12538.30973 = 0x30FB

Using this procedure, the following values are defined in the SineSamples array :

```
static int SineSamples[]={0x00000, 0x018f8, 0x030fb, 0x0471c, 0x05a82, 0x06a6d, 0x07641,
                          0x07d8a, 0x07fff,/*0x07fff,*/ 0x07d8a, 0x07641, 0x06a6d, 0x05a82,
                          0x0471c, 0x030fb, 0x018f8,0x00000,/*0x00000,*/ 0x0e709, 0x0cf06,
                          0x0b8e5, 0x0a57f, 0x09594,  0x089c0, 0x08277, 0x08002,/*0x08002,*/
                          0x08277, 0x089c0, 0x09594, 0x0a57f, 0x0b8e5, 0x0cf06, 0xe0709/*,
                          0x00000*/};
```

Next, the MCBSP_open() and MCBSP_config() functions are used to configure the MCBSP1 peripheral.

Finally, the transmission is initiated by calling the MCBSP_start function. A delay loop is introduced to allow the interrupt to occur, and when it does, the write_isr function is invoked. This function continuously writes the previously defined audio samples into the MCBSP1 transmit registers, thereby generating a tone that is played on the speakers indefinitely.

3.2 DSP Booting Example

This example will be used to demonstrate the proper procedure for booting the DSP. MPUI boot download, one of the six boot modes supported by the DSP, will be used. The bootload is accomplished through the MPU while the DSP is held in reset. The DSP application code is transferred on-chip through one of several interfaces to the MPU and sent through an internal memory interface to the DSP. The DSP application code writes a predefined value into the DSP2ARM1 mailbox register, which generates an interrupt and alerts the MPU. The MPU then proceeds to check the value which was written and compare it with the expected value. If the two match, the MPU will cause the power LED on the front of the Innovator™ development platform to flash green indefinitely. If not, the LED will flash red.

The procedure for converting the DSP application code into a format that is usable by the MPU will be shown first, and will be followed by a description of the MPU code which downloads the converted DSP application code into the internal memory of the DSP.

3.2.1 Converting DSP COFF Files Into ARM C-Language Header Files

Since the MPU and the DSP are different types of processors, the DSP application code needs to be converted into a format usable by the MPU. To do this, the DSP COFF (common object file format) file output is converted into MPU C language header files, which declare arrays initialized with the contents of the COFF file. This conversion can be accomplished by using the converter program *OUT2BOOT*. This converter uses the TI assembly language tool *HEX55* to extract the DSP code from the COFF file. See the *TMS320C55x Assembly Language Tools User's Guide* (SPRU280D) for information on the COFF file format and the *HEX55* conversion utility. The *OUT2BOOT* program calls *HEX55* to convert the COFF file into one binary output file. Then, *OUT2BOOT* converts this binary file into a C language header file, which is used as an include file with the MPU C code. The combined MPU and DSP code can then be placed into flash or ROM to be accessed from the MPU external memory interface. Figure 14 shows a graphical representation of this process.

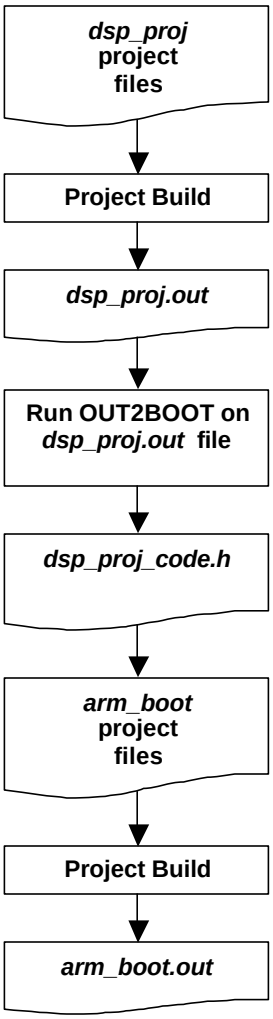


Figure 14. Bootloader Build Flowchart

The header file that OUT2BOOT produces (dsp_proj_code.h) consists of an array that holds the sections of code/data (see Figure 15). A record consisting of the following fields represents each section:

- The section length in 16-bit words
- The DSP destination run-time address for the section.
- The code/data words for the section.

```
Array [] = {
length, address, data, …
length, address, data, …
length, address, data, …
0 // ‘0’ in the final line signifies the end of program code or data
}
```

Figure 15. OUT2BOOT Format for DSP Code and Data

3.2.2 MPU Code Description

Figure 16 shows the MPU project code which downloads the DSP application code created by the OUT2BOOT utility into DSP internal memory (SARAM).

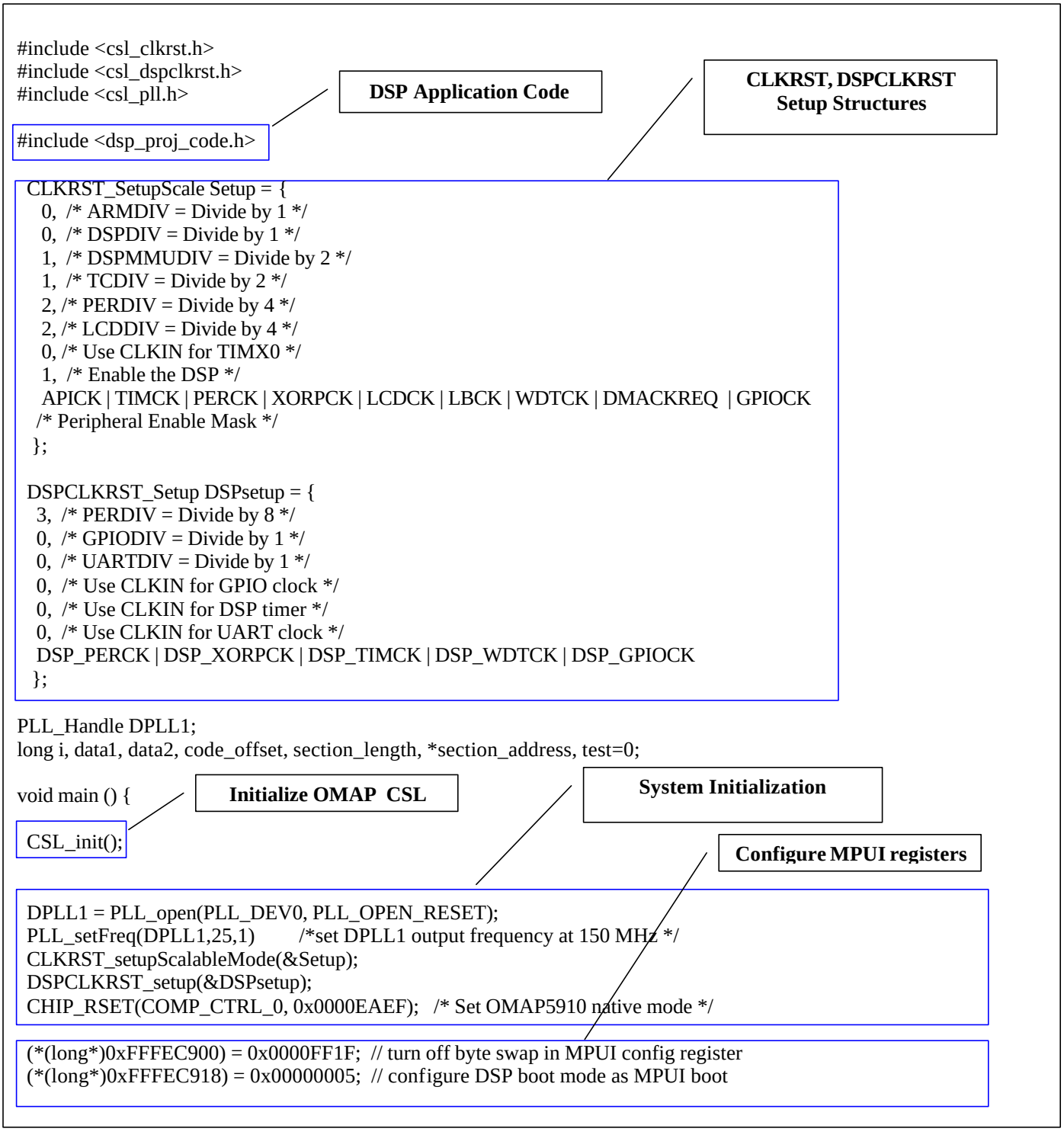


Figure 16. MPU Application Code (arm_boot)

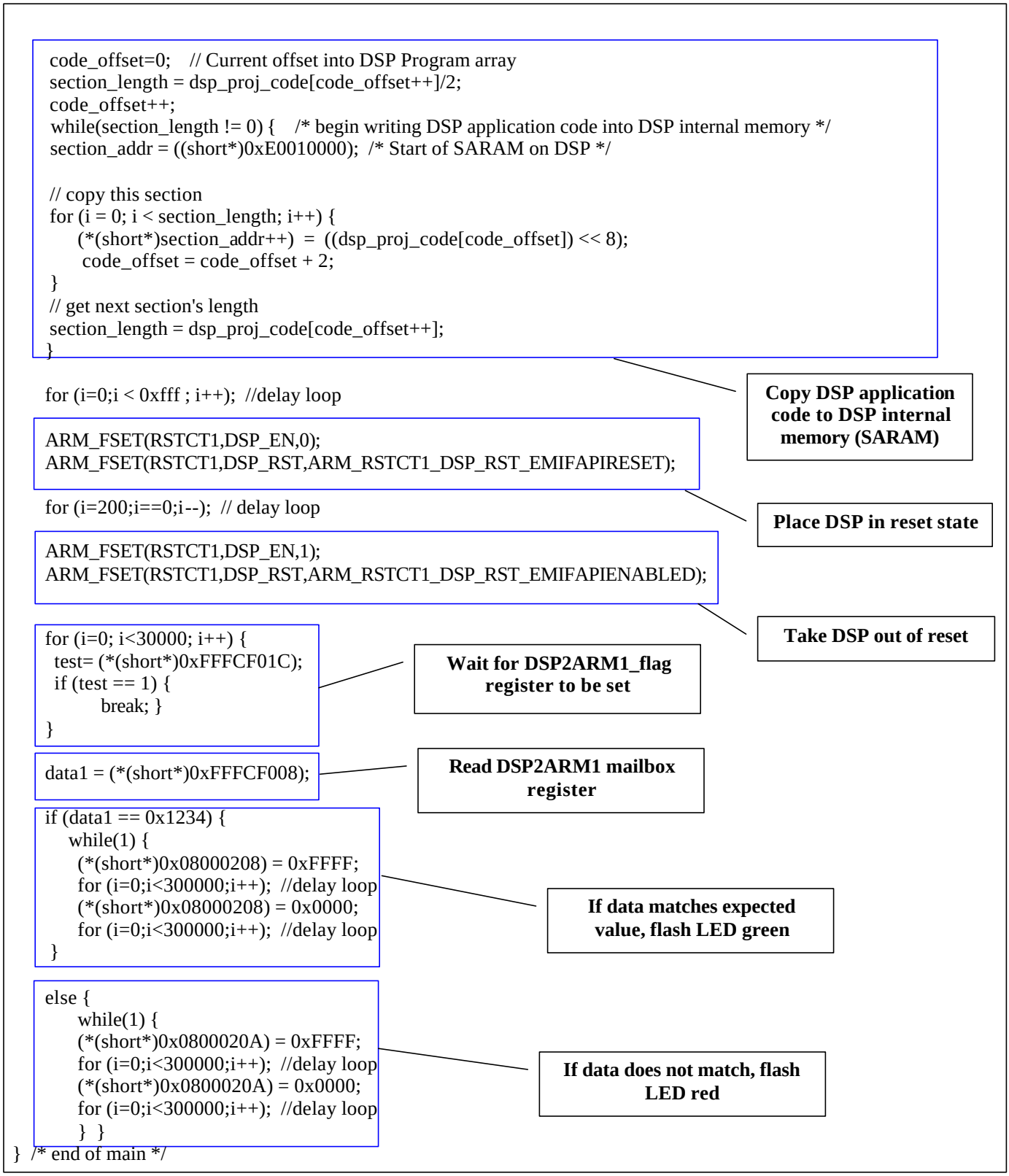


Figure 16. MPU Application Code (arm_boot) (Continued)

The example code shown in Figure 16 begins by initializing the various configuration structures and arrays that will be used in the following sections. At the beginning of main(), the example initializes the OMAP Chip Support Library (CSL) by calling the CSL_init() function. **This function initializes the CSL data structures and hooks up the Level 1 interrupt handlers — it must be called if CSL is to be used.** The code then performs OMAP5910 system initialization by using CSL functions described in sections 2.2 and 2.3. These functions configure the ARM and DSP to run at 150 MHz (see Table 1) and set the device in native mode.

After system initialization is performed, the byte swap feature of the MPUI (MPU interface) is turned off. This is done by writing 00b to bit fields [17:16] of the MPUI CTRL_REG. This will prevent the MPUI logic from swapping bytes on the MPUI/DSP interface, and as a result, allow the writing of the DSP application code produced by the OUT2BOOT utility directly into DSP internal memory.

Next, the DSP is configured for MPUI boot mode. This is done by setting the BOOT_MOD field of the DSP_BOOT_CONFIG register to 0101b. In MPUI boot mode the DSP reset vector address is mapped to the start of the SARAM, or 0x10000 (DSP byte address). This memory location can be accessed from the ARM by using the address 0xE0010000.

The DSP application code is then read from the array that the OUT2BOOT utility produced in the dsp_proj_code.h header file and written into the DSP internal memory, starting at location 0xE0010000. After the application code has been downloaded, the DSP is placed into reset and taken out of reset after a short delay. After the reset is released, the DSP will branch to the reset vector at 0x10000 (ARM address 0xE0010000) and begin execution of the application code.

The DSP application code invokes the DSP to write the values 0x1234 and 0x00FF into the DSP2ARM1 and DSP2ARM1b mailbox registers respectively. The action of writing to the DSP2ARM1b register causes the DSP2ARM1_flag register to be set to 1. After the DSP is released from reset, the MPU code continuously reads this register until it is set. When it is set, it reads the value in the DSP2ARM1 register and compares it with the expected value (0x1234). If it matches, the MPU causes the power LED on the front of the Innovator™ to flash green indefinitely. If not, the LED will flash red.

3.2.3 Building and Running the Example

Listed below are the MS VC++ Studio and Code Composer Studio projects, which build the individual components of this example. NOTE: The compressed file which accompanies this application report, *5910_bootloader.zip*, should be copied to the Code Composer Studio *<drive>:\ti\myprojects* directory and extracted there into*<drive>:\ti\myprojects\5910_bootloader.* All directories referenced below and in the following build example are relative to the 5910_bootloader directory.

• .**out2boot:** MS VC++ Studio project which builds the *out2boot.exe* conversion utility. This tool converts a DSP COFF File into an ARM C-language header. The source code is provided, as well as the executable*, .\out2boot\out2boot.exe*.

• .**dsp_proj:** Code Composer Studio project which builds a simple DSP application to write to the DSP2ARM mailbox register. The DSP application can be modified or replaced to include new code.

• .**arm_boot:** Code Composer Studio project which builds the ARM-side bootloader program, *arm_boot.out.* This program initializes the device and copies the DSP application into ARM port interface memory. This project can be modified to include other ARM or DSP code (as modified by out2boot) to be loaded by the MPU.

- Building the DSP Bootloader Example

1. If needed, use the MS VC++ Studio workspace file, .*out2boot**out2boot.dsw,* to build the conversion utility **.\out2boot\out2boot.exe.** Otherwise, just use the existing executable and skip this step.

2. Using the C55x Code Composer Studio project file, .*dsp_proj**dsp_proj.pjt,* build the sample DSP application **.\dsp_proj\dsp_proj.out**. Next, run the out2boot utility on the **dsp_proj.out** image, and copy the resulting C-header file, *dsp_proj_code.h,* into the. *arm_boot* directory.

3. Using the ARM Code Composer Studio project file, .*arm_boot**arm_boot.pjt,* build the ARM bootloader program, **.\arm_boot\arm_boot.out**.

- Executing the DSP Bootloader Example

To execute the bootloader example, follow these steps (NOTE: All directories referenced below are relative to the Code Composer Studio root installation directory and project sub-directory):

1. Start Code Composer Studio. This should run the Parallel Debug Manager.

2. From the Parallel Debug Manager open ARM Code Composer Studio.

3. From the Parallel Debug Manager open C55x Code Composer Studio.

4. In the C55x Code Composer Studio debug window, make sure that the DSP is in the running state. This will be indicated at the bottom of the window. If it is not, then execute Debug–>Reset CPU, followed by Debug–>Run Free.

5. In the ARM Code Composer Studio debug window, load the executable: **.\arm_boot\arm_boot.out**.

6. In the ARM Code Composer Studio debug window, execute Debug–>Run. The power LED on the front of the Innovator™ platform should begin to flash green or red.

Appendix A TLV320AIC23 Stereo Codec

The TLV320AIC23 is a high–performance stereo audio codec that is found on the Skywalker board. For more details, please refer to the *TLV320AIC23 Stereo Audio D/A Converter, 8-to 96-kHz, With Integrated Headphone Amplifier* (SLWS106) data manual.

The analog–to–digital converters (ADCs) and digital–to–analog converters (DACs) within the TLV320AIC23 use multibit sigma–delta technology with integrated oversampling digital interpolation filters. Data–transfer word lengths of 16, 20, 24, and 32 bits, with sample rates from 8 kHz to 96 kHz, are supported.

The ADC sigma–delta modulator features third–order multibit architecture with up to 90–dBA signal–to–noise ratio (SNR) at audio sampling rates up to 96 kHz, enabling high–fidelity audio recording in a compact, power–saving design.

The DAC sigma–delta modulator features a second–order multibit architecture with up to 100–dBA SNR at audio sampling rates up to 96 kHz, enabling high–quality digital audio–playback capability, while consuming less than 23 mW during playback only.

The TLV320AIC23 is the ideal analog input/output (I/O) choice for portable digital audio–player and recorder applications, such as MP3 digital audio players.

The TLV320AIC23 has many programmable features.

The control interface is used to program the registers of the device. The control interface complies with I2C specifications.

In I2C mode, the data transfer uses SDIN for the serial data and SCLK for the serial clock. The start condition is a falling edge on SDIN while SCLK is high. The seven bits following the start condition determine which device on the I2C bus receives the data. R/W determines the direction of the data transfer.

The TLV320AIC23 is a write only device and responds only if R/W is 0. The device operates only as a slave device whose address is selected by setting the state of the CS pin as follows.

Table 4. Table A-1. Slave Address Selection

CS State (Default = 0)	Address
0	0011010
1	0011011

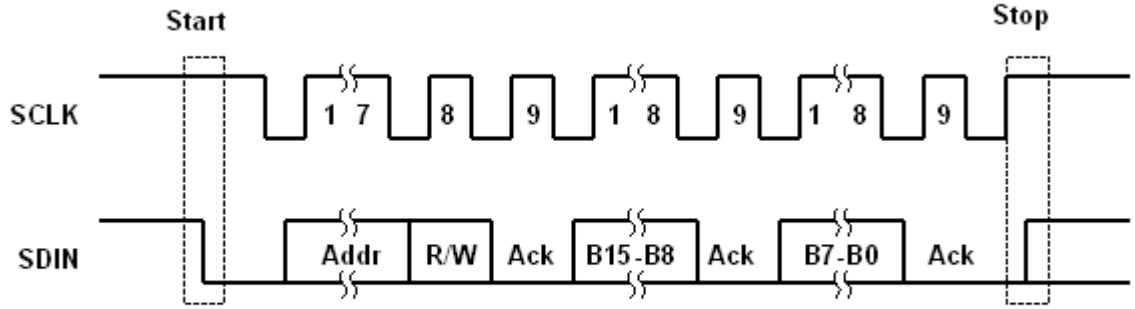
Based on the default setting of the CS pin (used in the TMS320VC5509 EVM), the slave address for the AIC23 should be 0011010, or 0x1A. The device that recognizes the address responds by pulling SDIN low during the ninth clock cycle, acknowledging the data transfer.

The control follows in the next two eight–bit blocks. The stop condition after the data transfer is a rising edge on SDIN when SCLK is high (see Table B-1).

The 16–bit control word is divided into two parts. The first part is the address block, the second part is the data block:

B[15:9] Control Address Bits

B[8:0] Control Data Bits



B[15:8] Control Address Bits
B[7-0] Control Data Bits

Figure 17. Figure A-1. 2–Wire I2C Compatible Timing

The AIC23 has eleven registers that are used to control operation. The addresses for the eleven registers are shown in Table B-2.

Table 5. Table A-2. AIC23 Control Register Addresses

Address	Register
0000000	Left line input channel volume control
0000001	Right line input channel volume control
0000010	Left channel headphone volume control
0000011	Right channel headphone volume control
0000100	Analog audio path control
0000101	Digital audio path control
0000110	Power down control
0000111	Digital audio interface format
0001000	Sample rate control
0001001	Digital Interface activation
0001111	Reset Register

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third–party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265