

---

# Symphony SoundBite: Assembly Project Template

User's Guide

Document Number: SNDBASMTMPL

Version: 2.0

September 2008



#### **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 89 921 03 559 (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  
1-800-521-6274 or 303-675-2140  
Fax: 303-675-2150  
LDCForFreescaleSemiconductor@hibbertgroup.com

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals", must be validated for each customer application by customer's technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners.

© Freescale Semiconductor, Inc. 2008. All rights reserved.

# Contents

<b>About This Document</b> .....	<b>3</b>
<b>Chapter 1 Overview</b> .....	<b>5</b>
1.1 Introduction.....	5
1.2 Symphony SoundBite Hardware .....	5
1.3 Analog/Optical and Digital Conversion .....	5
<b>Chapter 2 Generating the Assembly Project</b> .....	<b>6</b>
2.1 Create and Build the Assembly Project .....	6
2.2 Run the Assembly Project on Hardware .....	7
<b>Chapter 3 High Level Overview</b> .....	<b>9</b>
<b>Chapter 4 Detailed Operation</b> .....	<b>10</b>
4.1.1 The Files and Their Function .....	10
4.1.2 Project Structure .....	11
4.1.3 Application Memory Map .....	11
4.1.4 Input/Output Buffer Structure .....	13
4.1.5 Main Program Flow .....	14
4.1.6 Interrupt Service Routines.....	16
4.1.6.1 Left Data Transmit, esai_tx_isr_even .....	17
4.1.6.2 Right Data Transmit, esai_tx_isr .....	17
4.1.6.3 Left Data Receive, esai_1_rx_even .....	17
4.1.6.4 Right Data Receive, esai_1_rx.....	17
<b>Chapter 5 Customizing the Project</b> .....	<b>18</b>
5.1 Reserved Registers .....	18
5.2 Main Loop.....	19
5.3 Enabling/Disabling Processing .....	19

5.4 Modifying PROCESS\_AUDIO ..... 19

5.5 Buffer Modification..... 20

5.6 Output Channel Mapping ..... 20

Because of an order from the United States International Trade Commission, BGA-packaged product lines and part numbers indicated here currently are not available from Freescale for import or sale in the United States prior to September 2010: i.MX Product Family, DragonBall Product Family

## About This Document

This document describes the organization and operation of the Symphony SoundBite Assembly Project Template so that the user can extend the project template to perform custom digital signal processing applications on the Symphony SoundBite board. Since the source assembly code is not duplicated in this document, the project and the files contained within should be consulted in addition to this text.

## Audience

This document is intended for users who want to build their own assembly-only digital signal processing application code to run on the Symphony SoundBite development board.

## Organization

This document is organized into the following chapters:

- |           |   |
|-----------|---|
| Chapter 1 | Gives a brief overview of the Symphony SoundBite development board                                      |
| Chapter 2 | Describes how to build the Assembly Project from the source files                                       |
| Chapter 3 | Gives a high-level overview of the Assembly Project Template application flow                           |
| Chapter 4 | Describes how the Assembly Project Template application works in detail                                 |
| Chapter 5 | Discusses how to modify the Assembly Project Template for custom digital signal processing applications |

## References

Also see:

- ***Symphony SoundBite: Reference Manual***
- ***Symphony SoundBite: Quick Start With Symphony Studio***
- ***Symphony SoundBite: Demonstration Application***



# Chapter 1 Overview

## 1.1 Introduction

An Assembly Project Template has been developed by Freescale to provide users of the Symphony SoundBite audio development board an application framework upon which to build their own software application. The author would like to acknowledge that while other options exist, this particular template has been limited to and implemented wholly in assembly language. The intent of this document is to familiarize the user with the Assembly Project Template, explain its operation and provide a foundation upon which custom applications may be derived. Please consult the project's source files in addition to this text, because none of the code and comments are duplicated herein.

## 1.2 Symphony SoundBite Hardware

The Symphony SoundBite is an inexpensive but feature-packed solution for audio digital signal processing which is based around the Symphony DSPB56371 digital signal processor. It supports the simultaneous input and output of 8 channels of analog audio or 6 channels of analog with digital stereo optical SP/DIF through the eight 3.5mm jacks on the board. Sample rates of up to 192 kHz are supported with the four 24-bit stereo codecs mounted on the board. A serial I2C EEPROM provides non-volatile storage for the DSP as well as allowing the board to operate stand-alone (without an attached host PC). An 8-position DIP switch and 9 LEDs are connected to GPIO pins on the DSP and are available for user interaction and status indication. An expansion header is also provided to facilitate off-board expansion.

The on-board USB communication interface is built around the FT-2232 dual USB UART. It provides for low-level JTAG/OnCE debugging capability with the use of Symphony Studio, allowing application code development in assembly alone or C alone or a mixture of C and assembly. Additionally, this interface provides a “data pipe” for high-level SPI or I2C serial communication between the host PC and the DSP. Data and/or application code may be transmitted through the data pipe.

## 1.3 Analog/Optical and Digital Conversion

The four stereo codecs provide for the conversion between the analog and digital audio. The digital audio data input and output streams are connected to the DSP. In the case of the optical input and output, the AK4584 codec performs the conversion between the optical digital and electrical digital streams. The DSP processes the streams it receives and then outputs serial digital audio streams back to the codecs.

All the “magic” processing that the Symphony SoundBite is capable of occurs within the DSP in the digital signal processing application code that you, the user, will create for it.

## Chapter 2 Generating the Assembly Project

In order to better understand the Assembly Project Template application code, we shall begin with the source files. The following sections will cover the creation of the project, importing the source files, configuring the project properties, building the project, and running the application code on the Symphony SoundBite hardware.

### 2.1 Create and Build the Assembly Project

Begin by creating an empty Assembly project in Symphony Studio. (Here, it is assumed that Symphony Studio has already been properly installed and configured on the host PC and that the user is somewhat familiar with the application. If not, please refer to the document, *Symphony SoundBite: Quick Start with Symphony Studio*.)

1. Launch Symphony Studio. Make sure auto-building is turned off by clicking the *Project* menu. If there is a check by *Build Automatically*, select *Build Automatically* to toggle the auto-building off.
2. In the *C/C++ Projects* pane of the *C/C++* perspective, right-click and select *New > Managed Make ASM Project*. Alternatively, the menu *File > New > Managed Make ASM Project* may also be used.
3. Enter a name in the field *Project name*. Here “*assembly\_template*” will be used. Click *Next* when done.
4. Verify that the *Project Type* is *56K ASM COFF*. Leaving everything else unchanged (the defaults), click *Finish*.

A new, blank assembly project should now appear in the *C/C++ Projects* pane. Now, do the following to import the source files into the empty project:

5. Select the project in the *C/C++ Projects* pane and right-click it. Select *Import...* from the menu that pops up.
6. In the dialog box that opens up, expand *General* and select *Archive File*. Click *Next*.
7. Use the *Browse* button to navigate to the location on your hard drive where the file containing the Assembly Project Template source files resides, *SoundBite\_Assy\_Tmpl.zip*. Select it and click *Open*.
8. Make sure that all the check boxes on both halves of the dialog box have check marks in them (they should by default). The field *Into folder* should have the name of the project created above in it. If it does not, use the *Browse* button to select it. Check the box by “Overwrite existing resources without warning” and click *Finish*.

9. Expand the project *assembly\_template* in the *C/C++ Projects* pane. It should contain all the files that were in the Assembly Project Template source archive.

The project files have now been imported but the project properties need to be modified in order for the project to build properly. Do the following:

10. Right click the project *assembly\_template* and select *Properties* from the pop up menu.
11. Select *C/C++ Build* on the left hand side. On the right hand side, select the *Tool Settings* tab.
12. Select *Options* under *56k ASM Linker*. These are the interface to the command line options for the 56300 Linker executable.
13. Type a name for the *Map File* command line option. Here, “*mapfile.txt*” will be used.
14. In the *Memory Control File* option field, enter the following (without the quotes):  
“*..\sb\_linker.ctl*”
15. Click the *OK* button.

The project is now ready to build. Do so by right clicking the project name (*assembly\_template* in this case) and selecting *Build Project*.

Text emitted by the assembler and linker should scroll in the *Console* tab. When successfully built, folders named *Debug* and *Binaries* folder objects should appear in the project, and there should be no red mark on the project icon.

## 2.2 Run the Assembly Project on Hardware

Now that the Assembly Project Template has been imported, configured and built, it can be executed and debugged. (If an external tool has already been configured for the Symphony SoundBite, launch it and continue at step 20.)

16. Switch to the *Debug* perspective by clicking on the *Debug* button at the top right hand side of the window.
17. Create the external tool for making the hardware connection to the Symphony SoundBite board by using the menu *Run > External Tools > External Tools...*
18. In the dialog box that comes up, on the left select *OpenOCD GDB Server* and then click the new button (the page icon with the yellow “+”).
19. Under *OpenOCD Configuration File*, choose *56371* in the *Device* pull down menu and choose *soundbite* in the *Dongle* pull down menu.
20. Click *Run*.

21. After launching, the console tab should display a single status line with no errors similar to the following:

```
Info: openocd.c:82 main(): Open On-Chip Debugger ps001 (2007-10-19 18:00 CEST)
```

If it does not, re-launch the external tool by selecting it from the *Run > Debug History*.

Now that the external tool has been configured and is connected to the hardware, use the debug configuration to download the application and run it. These steps may be skipped once the debug configuration is in place.

22. Create a debug configuration for *assembly\_template* by using the menu *Run > Debug...*
23. Select the debug *Freescale 56371* from the left side of the dialog box. Click the new button (the page icon with the yellow “+”). Make sure the *Main* tab is selected.
24. On the right hand side, assign a new name to the debug configuration, if one different from the default name that was generated is desired, by entering it into the *Name* field.
25. If the project *assembly\_template* does not already appear in the *Project* field, click the *Browse* button next to it and select the project from the list that pops up.
26. For the field *C/C++ Application*, click the *Search Project* button and choose the *assembly\_template.cld* DSP code object file and click *OK*.
27. Verify that *Run at Startup* and *Stop on Startup* are unchecked.
28. Verify that *Download onto Target* is checked.
29. Verify that *Core Index* is *0*.
30. Click the *Commands* tab. Verify the following 4 lines of text are in the ‘*Initialize*’ *commands* text box:

```
M p:0 0x000084
M p:1 0x000200
set $pc=0
cont
```

This text is necessary when using Symphony Studio with the DSP56371 because of errata ED54. See [FAQ-28010](http://www.freescale.com) at <http://www.freescale.com> for more information.

31. Click *Apply* to save the changes.
32. Click *Debug* to launch the debug configuration.

Run, debug, modify and rebuild the Assembly Project Template application as desired. As built from the files in the archive, audio signals present at every one of the 8 analog inputs will be passed unmodified to the corresponding analog output.

# Chapter 3 High Level Overview

The Assembly Project Template application passes analog audio signals present at each of the 8 inputs to the corresponding analog output without modification. Figure 3-1 shows a flow chart of the code executing within the Assembly Project Template application. The following chapter covers the detailed operation of the application flow chart shown graphically below.

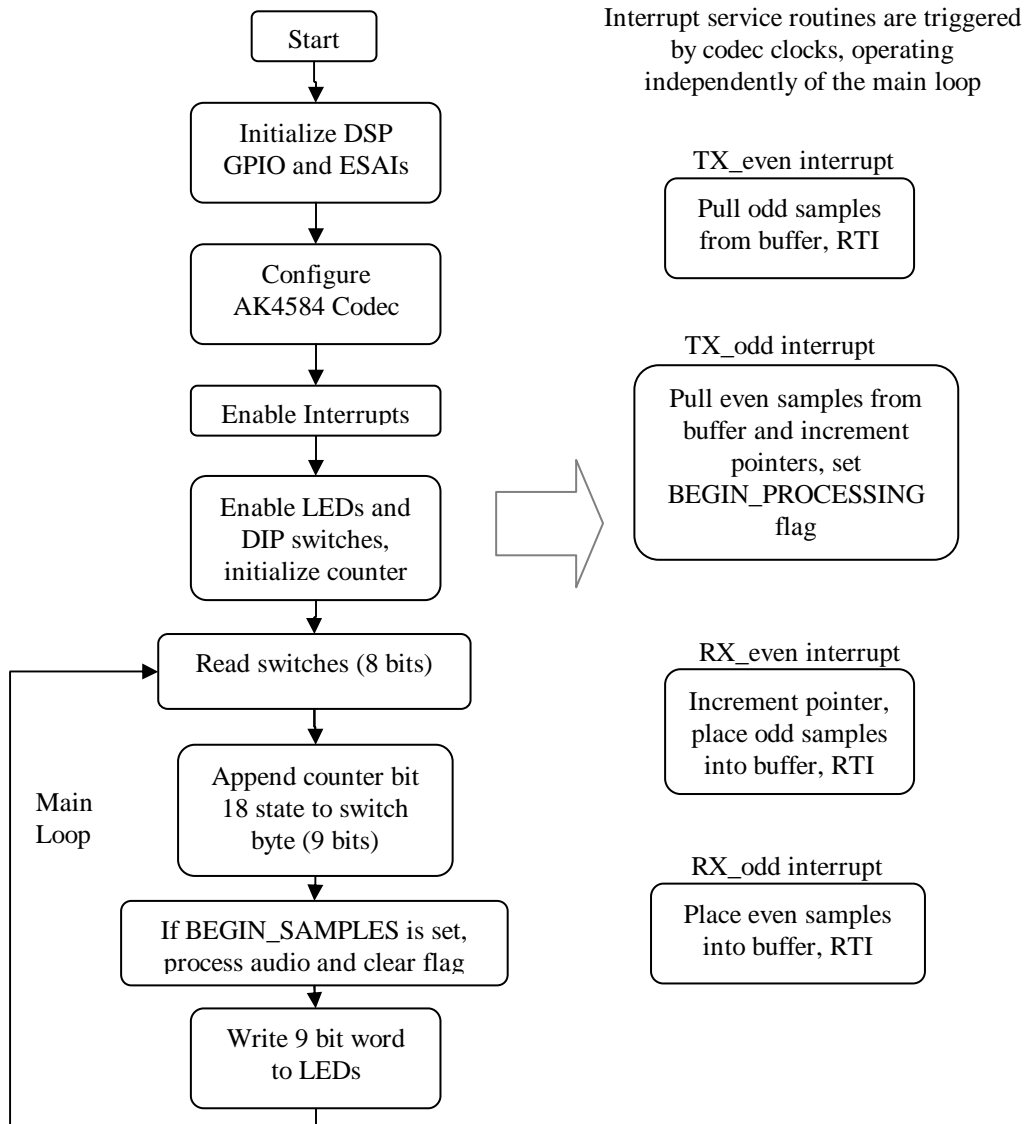


Figure 3-1. Assembly Project Template Flowchart

## Chapter 4 Detailed Operation

The program flow shown in Figure 3-1 will be traced in detail throughout this section. Each assembly file in the project will be mentioned as it is encountered.

### 4.1.1 The Files and Their Function

Table 4-1 lists the name of each assembly file and the general function of that file within the Assembly Project Template.

Table 4-1. The assembly code files and their function in brief.

Filename	Description
main.asm	The <i>main.asm</i> file contains the startup code for the application and the main loop of the application. The main loop contains a counter that is used to flash LED9. It also reads the 8-position DIP switch and displays the state of each switch in LED1-LED8.
process_samples.asm	The <i>process_samples.asm</i> file contains the routines that enable and disable audio processing by the application. When disabled, audio is passed directly from each input to the corresponding output. When enabled, the same thing occurs in the Assembly Project Template until you, the user, modifies that routine to do the audio processing you desire.
sb_codecs.asm	The <i>sb_codecs.asm</i> file contains the subroutines that enable the GPIO pins for the bit-banged serial communication link between the DSP and the AK4584 codec.
sb_isr_esais.asm	The <i>sb_isr_esais.asm</i> file contains the interrupt service routines that receive the digital audio input streams from the codecs and fill the input buffers and pull the audio data from the output buffers for transmission to the codecs. Once enabled, the interrupt service routines do all the work in receiving, transmitting and processing the digital audio. The processing work is done by a call to <i>process_samples.asm</i> after the flag <i>BEGIN_PROCESSING</i> is set.
sb_leds.asm	The <i>sb_leds.asm</i> file contains subroutines that make it easy to enable and turn on and off the general purpose LEDs on the Symphony SoundBite board.
sb_switches.asm	The file <i>sb_switches</i> file contains subroutines that make it easy to enable and reads the status of the 8-position general purpose DIP switch.
sb_eeprogram.asm	The <i>sb_eeprogram.asm</i> file contains subroutines required to write the I <sup>2</sup> C serial EEPROM attached to the DSP56371 in the format required for the

## 4.1.2 Project Structure

It should be noted that the Assembly Project Template application uses features of the DSP56300 assembler and linker to make the assembly code more humanly readable and that facilitate more structured programming. As such, it is assumed that the user has some familiarity with structured programming techniques. For more information about the specific details of the assembler and linker features, please refer to the *Assembler Reference Manual* and the *Linker Reference Manual* documents. The more important and more often used features in the Assembly Project Template are mentioned below.

The files *dsp56371.equ*, *soundbite.equ*, and *soundbite\_macros.equ* contain symbols that are used by the assembly code files to give readable, meaningful names for the peripheral registers and bits, DSP pins, DIP switches and LEDs. These symbols are incorporated into the assembly file with the *INCLUDE* assembler directive. The *INCLUDE* directives are typically surrounded by *LIST* and *NOLIST* directives which suppress the output of these files in the listing output of the assembler.

Each assembly file is bounded by *SECTION* and *ENDSEC* directives which instruct the assembler to assemble the enclosed code into a relative object file in such a way that it can be linked together with other relative objects to form the final absolute code object. This allows external symbols from other assembly files to be used in the current file as well as the ability to expose local symbols in the current file for use in external code files. The directives *GLOBAL*, *XREF*, and *XDEF* are used to define global symbols, externally defined symbols to be used locally and local symbols to expose for external usage, respectively.

The *ORG* directive instructs the compiler to resume assembly using the specified memory space at an optionally specified address (which may be a symbol or an absolute address).

The file *sb\_linker.ctl* is a linker control file that defines where in memory each section of the Assembly Project Template application should be placed when the relative object files are linked together to form the final absolute code object file. It is used to guarantee the ordering of the sections in memory to ensure that the no application code resides in the interrupt vector table of the DSP and that the EEPROM programming section, which is seldom used, is placed at the end of the application object. This makes it easy to determine the size of the final code object which in turn makes programming the EEPROM of the Symphony SoundBite for stand-alone operation a relatively simple matter.

## 4.1.3 Application Memory Map

Figure 4-1 shows how the memory is allocated in the Assembly Project Template application.

The application code itself starts at P:0 with the RESET vector at the base of the vector table. The interrupt vectors also reside in the vector table and pointers to the ESAI interrupt service routines in the section *isr\_esais*.

The beginning of the actual application, in the section *main* is P:100. The rest of the sections of the application follow with the order specified by the linker control file, *sb\_linker.ctf*.

The X and Y memory data spaces hold channel data in circular buffers for each codec. The exact structure and function of the buffers will be discussed next section. X memory also holds the software stack intended for subroutines to use to save and restore registers or any other values as desired. In the setup portion, the AK4584 (U5) codec's register settings before and after initialization are also stored in X memory, as will be discussed later section as well.

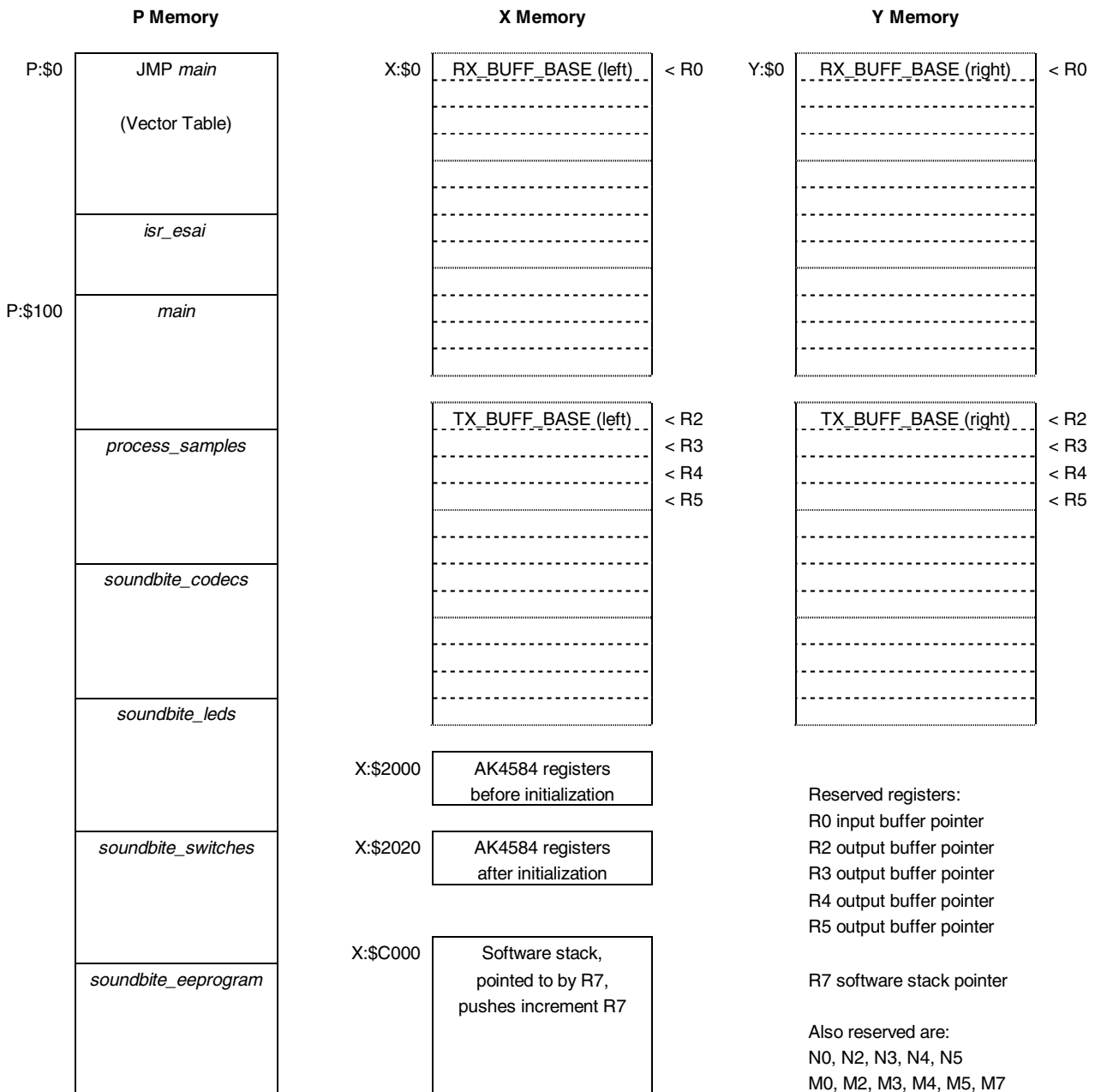


Figure 4-1. Assembly Project Template application memory map in the P, X and Y memory spaces.

## 4.1.4 Input/Output Buffer Structure

The input and output circular buffers, coupled with the ESAI interrupt service routines, are at the center of the “action” in the Assembly Project Template application. The input and output process is shown pictorially in Figure 4-2.

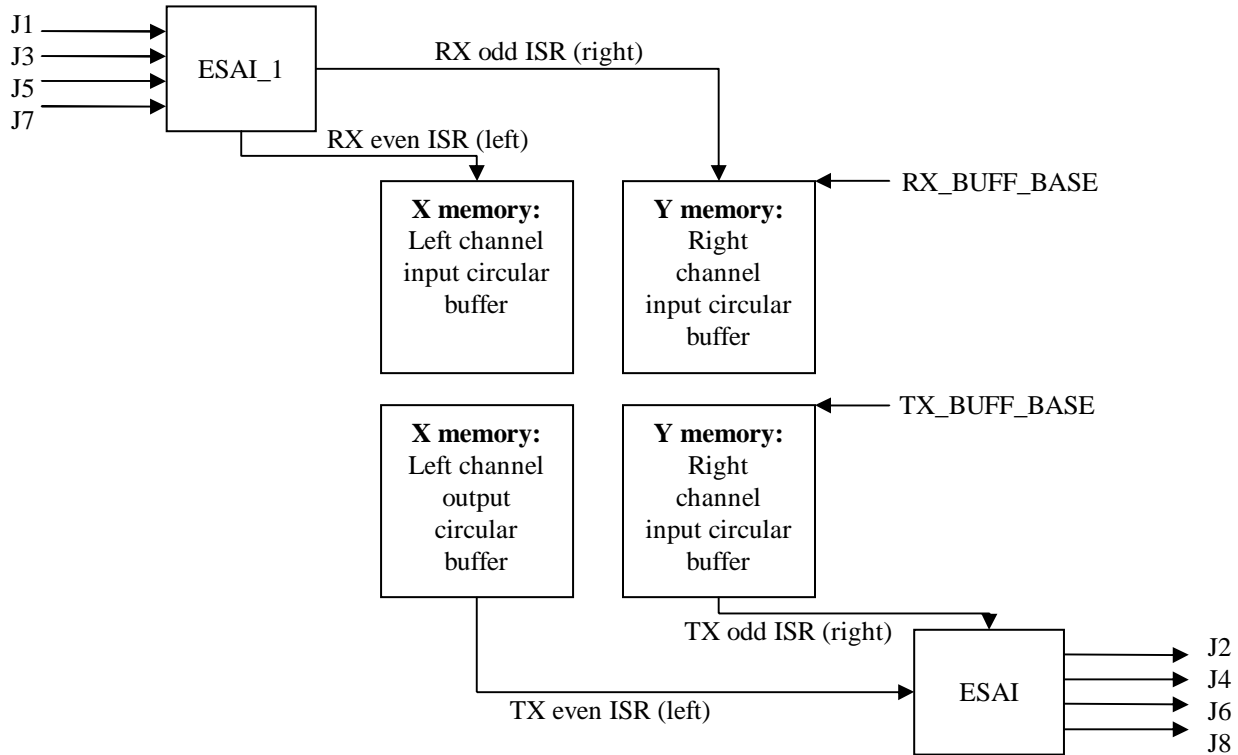


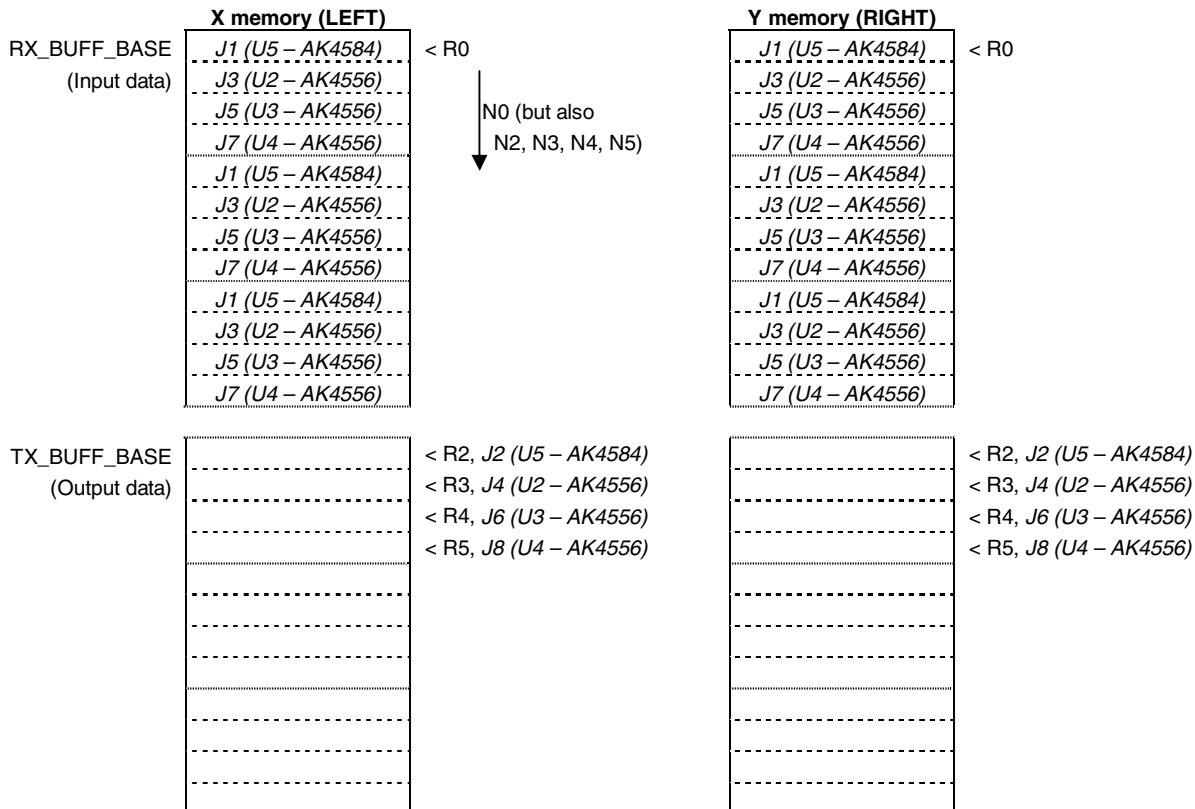
Figure 4-2. Paths of the digital audio to and from the various circular buffers.

The inputs and outputs of **ESAI\_1** and **ESAI** are in  $I^2S$  format, that is, both the left and right channel data is on the one data line, differentiated by the **LRCLK** signal. Both **ESAI\_1** and **ESAI** are configured so that two interrupts are generated, one for the left channel data and one for the right channel data. For both **ESAI** peripherals, these add up to a total of four interrupts corresponding to receiving left and right data, and transmitting left and right data. These are indicated in Figure 4-2 by the arrows connecting the buffers to the **ESAI** blocks. These interrupt service routines place the data in or pull the data out from the appropriate circular buffer. Additionally, the right transmit interrupt service routine sets the flag **BEGIN\_PROCESSING**. In the main loop of the application, a call is made to *process\_samples* when the **BEGIN\_PROCESSING** flag is observed as being set. It should also be noted that all the buffers are circular, which means that as the pointers increment, they wrap around automatically back to the beginning when the end of the buffer is reached (controlled by the **Mx** registers). The details of the interrupt service routines, the buffers, the pointers, and the audio processing will be covered in subsequent sections.

Each circular buffer holds the sample data in blocks of four samples. In the default configuration of the Assembly Project Template, each buffer contains 3 blocks of 4 samples each, which provides two

previous samples in addition to the current sample for each channel's input and output data. This sizing readily allows for the implementation of second-order filters.

The pointer register (R0) is used for pointing to the current block in the input buffer, as indicated in Figure 4-3. The four pointer registers R2, R3, R4, and R5 are each dedicated to a single ESAI output, which corresponds to a single codec output as shown in Figure 4-3. The corresponding modifier registers for these pointers are configured to make the pointers wrap around, thus creating circular buffers. After each sample, the pointers are incremented by an offset of 4 (using the corresponding Nx offset register) so they point to the next buffer block.



**Figure 4-3: Diagram of the default left and right input and output buffer structure with the data sources and destinations.**

### 4.1.5 Main Program Flow

Program execution begins in *main.asm* at the label *RESET* with a jump into the main code at *Fmain*.

At *Fmain*, the initialization of the DSP begins with the explicit masking of interrupts. This prevents any interrupts from occurring while the application is starting up. The PLL is programmed for one half of full

speed to prevent overshoot, per the data sheet requirements of the DSP56371. Several core registers are explicitly written with their reset values and the interrupt priority registers are written such that both ESAIs are enabled with an interrupt priority of 2. The software stack, which is used to save and restore registers during subroutines, is instantiated by writing the stack base address into the R7 register.

Next, a jump to the subroutine *INIT\_ESAIS* (in *sb\_codec.asm*) occurs. This subroutine initializes ESAI for output only and ESAI\_1 for input only with the data format being 24-bit I<sup>2</sup>S. Both peripherals receive their I<sup>2</sup>S clocks from the AK4584 codec (U5).

Following the return from *INIT\_ESAIS*, the PLL is programmed for operation at 178 MHz. A delay loop follows to allow the PLL to settle before program execution continues.

A subroutine call to *DISABLE\_PROCESSING* (in *process\_samples.asm*) is made to configure the application for default pass-through behavior where each output channel is mapped to the corresponding input channel. The pointer registers, address modifier and offset registers for the buffer pointers are all initialized here.

Program execution resumes in *main.asm* at the label *STRAIGHT*, where custom output channel mapping settings may be applied: the AK4584 (U5) is initialized for analog input and output and each output channel is mapped to the corresponding input channel by the initialization of the pointer registers R2, R3, R4 and R5, similar to what was done in *DISABLE\_PROCESSING*. A call is then made to *DISABLE\_PROCESSING* (in *process\_samples.asm*). Performing this seemingly redundant configuration allows for the modification of the Assembly Project Template to perform customized audio processing and channel mapping as well as on-the-fly enabling and disabling of the audio processing (for more details, please refer to Chapter 5, Customizing the Project).

The AK4584 codec (U5) is configured with a subroutine call to *SETUP\_AK4584* (in *sb\_codec.asm*) where the GPIO pins that perform the bit-bang serial communication with the AK4584 are configured. This is followed by a subroutine call to *GET\_REGS\_AK4584* (in *sb\_codec.asm*), which returns a dump of all the internal registers of the AK4584 codec to X memory beginning at x:\$2000.

The AK4584 is then reset and configured with a subroutine call to *INIT\_AK4584* (in *sb\_codec.asm*). This subroutine takes as an argument a value in Y0 which configures the inputs and outputs of the AK4584 (see assembly code and the AK4584 data sheet for more information). Recall that this value was placed in Y0 following the *STRAIGHT* label above.

Following the initialization of the AK4584, another subroutine call to *GET\_REGS\_AK4584* (in *sb\_codec.asm*) is made, this time dumping the block of the AK4584's registers to x:\$2020. These dumps of register values can be inspected to verify that the codec's internal registers were indeed written correctly with the debugger, if desired.

The setup and configuration of the Assembly Project Template application is now complete. Audio data will not be passed until interrupts are unmasked, which occurs next. Once unmasked, the ESAI\_1 and ESAI interrupts occur and their interrupt service routines handle the input and output of the audio data along. The audio processing occurs when the flag *BEGIN\_PROCESSING* is set within the right transmit ISR. In the default configuration, audio data is copied directly from each input buffer to each output buffer without modification, so that each output channel is a copy of the corresponding input channel.

Following the unmasking of interrupts, the GPIO pins connected to the general purpose LEDs and DIP switches on the Symphony SoundBite are properly configured as outputs and inputs (respectively) with subroutine calls to *SETUP\_LEDS* and *SETUP\_SWITCHES* (in *sb\_leds.asm* and *sb\_switches.asm*, respectively). The B register is cleared so that it may be used as a counter.

In the main loop itself, the B register is incremented. The switch status is read with a subroutine call to *READ\_SWITCHES* (in *sb\_switches.asm*), which returns the state of each switch in the lower 8 bits of the A1 register. To make the ninth LED blink, bit 18 of register B1 is tested. If it is set, bit 8 of register A1 is set (which is the ninth bit from the right of A1). The value in the lowest 9 bits of A1 is then used to update the state of the general purpose LEDs with a subroutine call to *SET\_LEDS* (in *sb\_leds.asm*).

Next, the state of the flag *BEGIN\_PROCESSING* is tested. If it is set, it indicates that a frame of left and right data has been received and just transmitted. A subroutine call is made to *PROCESS\_SAMPLES* (in *process\_samples.asm*) which processes the most recent samples by copying each newest input sample to the top of each corresponding output buffer.

On entry into *PROCESS\_SAMPLES*, at the label *PROCESS\_ENABLE*, there is a jump instruction that governs the execution path through the subroutine. This jump instruction is modified by the *ENABLE\_PROCESSING* and *DISABLE\_PROCESSING* (both in *process\_samples.asm*), which will be discussed at greater length in Chapter 5, *Customizing the Template*. This subroutine is invoked once after both sets of left and right channel sample data is received (that is, after the flag *BEGIN\_PROCESSING* is set). As configured, the Assembly Project Template application has the same code that copies the input data to the corresponding output buffer in the code blocks *PASS\_THROUGH* and *PROCESS\_AUDIO* in *process\_samples.asm*.

Upon return to the main loop, the flag *BEGIN\_PROCESSING* is cleared (it is reset so that no further audio processing will occur until the next frame of data occurs). The DSP is then instructed to jump back to the beginning of the loop.

While the application is running, the main loop monitors the state of the DIP switches and reflects them in the lowest eight of the general purpose LEDs. Changing the state of the DIP switches will be immediately reflected (at least to the human observer) by a corresponding change in illumination in the LEDs. The ninth LED will blink, as the 18<sup>th</sup> bit of B1 is set and clear with the incrementing of B in the main loop.

#### 4.1.6 Interrupt Service Routines

As stated previously, the interrupt service routines for *ESAI\_!* and *ESAI* handle the inputting and outputting of the digital audio data from and to the codecs. Additionally, the *TX odd interrupt* (right channel) interrupt service routine sets the flag *BEGIN\_PROCESSING*, which in turn triggers a subroutine call in the main loop to the audio processing subroutine *PROCESS\_SAMPLES* (in *main.asm*), which performs the actual audio signal processing. In the default Assembly Project Template configuration, *PROCESS\_SAMPLES* copies the input data directly to each corresponding output. The following section describes the operation of each interrupt service routine in more detail.

#### 4.1.6.1 Left Data Transmit, `esai_tx_isr_even`

The `esai_tx_isr_even` (in `sb_isr_esais.asm`) interrupt service routine for ESAI handles copying the left channel output data from the output buffer in X memory pointed to by registers R2, R3, R4, and R5 to the ESAI transmit registers. No other processing occurs in this ISR; the output data is presumed to be already present in the output buffer.

#### 4.1.6.2 Right Data Transmit, `esai_tx_isr`

The `esai_tx_isr` (in `sb_isr_esais.asm`) interrupt service routine for ESAI handles copying the right channel output data from the output buffer in Y memory pointed to by registers R2, R3, R4, and R5 to the ESAI transmit registers. The buffer pointers are incremented by the buffer size, which in the case of the Symphony SoundBite is 4, corresponding to the number of codecs on the board and the number of I<sup>2</sup>S output lines. No other processing occurs in this ISR; the output data is presumed to be already present in the output buffer. The incrementing occurs following the writing of the right channel data so as to keep the left and right sample data in the same frame. The flag `BEGIN_PROCESSING` is then set to signal to the main loop of the application that it is time to begin processing the latest set of input samples.

#### 4.1.6.3 Left Data Receive, `esai_1_rx_even`

The `esai_1_rx_even` (in `sb_isr_esais.asm`) interrupt service routine for ESAI\_1 handles copying the left channel input data from the ESAI\_1 receive registers to the left channel input buffer in X memory. The pointer register R0 is incremented by the buffer size (4) and stored on the software stack. The left channel sample data is then moved and the value of R0 is restored prior to exiting the ISR so that the corresponding right channel sample data will reside at the same addresses within the input buffer in Y memory.

#### 4.1.6.4 Right Data Receive, `esai_1_rx`

The `esai_1_rx` (in `sb_isr_esais.asm`) interrupt service routine for ESAI\_1 handles copying the right channel input data from the ESAI\_1 receive registers to the right channel input buffer in Y memory. The pointer register R0 preserved on the software stack before the data moves and restored afterwards so as to not require the use of any other address registers for the indexing.

## Chapter 5 Customizing the Project

The Assembly Project Template application was written to provide a working example application that can be readily extended to perform custom audio processing. As configured, the implementation of second-order filters is a simple matter since the input and output buffers contain the current sample and two previous samples of the input and output for all of the eight channels. The application used for testing the Symphony SoundBite board as described in *Symphony SoundBite: Demonstration Application* is based on the Assembly Project Template application (actually, to be historically accurate, the demonstration application came first; the Assembly Project Template was derived from it later).

The following sections are intended to provide some considerations when using the Assembly Project Template as a starting point for custom audio processing applications.

### 5.1 Reserved Registers

The digital audio processing as well as the audio data input and output of the Assembly Project Template application occurs exclusively through the interrupt service routines that service the ESAI peripherals. In order for these ISRs to operate independently, there are a number of core registers dedicated to their operation. These registers are listed in Figures 4-1 and 4-3 but they are re-listed here in Table 5-1 with a more descriptive explanation of their function.

Table 5-1: Reserved Registers and their usage.

Register	Description of Usage
R0	Input buffer pointer used for all ESAI_1 input channels
R2	Output pointer used for ESAI SDO0 [AK4584 codec (U5), J2]
R3	Output pointer used for ESAI SDO1 [AK4556 codec (U2), J4]
R4	Output pointer used for ESAI SDO2 [AK4556 codec (U3), J6]
R5	Output pointer used for ESAI SDO3 [AK4556 codec (U4), J8]
R7	Software stack pointer used by interrupt service routines
N0, N2, N3, N4, N5	Offset registers used with the input and output pointers used for incrementing the Rx registers by <i>buffsize</i> (see <i>sb_isr_esais.asm</i> )
M0, M2, M3, M4, M5	Modifier registers set with the value <i>keep</i> (see <i>sb_isr_esais.asm</i> ) to make the Rx registers behave as circular buffer pointers
M7	Modifier register for R7, keeps default linear arithmetic value of \$FFFFFF

All registers other than those listed in Table 5-1 are available for use by code both in the main loop as well as the audio processing algorithm code in *PROCESS\_AUDIO* (see *process\_samples.asm*). It should be noted that any registers used in *PROCESS\_AUDIO* that are not in the reserved list of registers should be placed on the software stack in order to avoid clobbering any that might be used in the main loop of the application.

## 5.2 Main Loop

The switch reading and LED lighting in main loop of the Assembly Project Template application is provided as a means of displaying whether or not the Symphony SoundBite is running application code.

Application code in the main loop must also be careful not to modify or tamper with the reserved registers used by the interrupt service routines. Processing in the main loop other than the subroutine call to *PROCESS\_SAMPLES* needs to be carefully managed to ensure that there are enough free MIPS available for the DSP to do the desired processing within the time period of a frame of data. If *PROCESS\_SAMPLES* is delayed too long, input buffer overrun or output buffer under-run may occur.

## 5.3 Enabling/Disabling Processing

The subroutines *ENABLE\_PROCESSING* and *DISABLE\_PROCESSING* (see *process\_samples.asm*) are provided so that the audio processing may be enabled and disabled under software control. Since these routines modify the application code in the subroutine *PROCESS\_SAMPLES* (also in *process\_samples.asm*), interrupts should be disabled prior to calls to these subroutines to avoid the unhappy event of changing the code while it is being executed which would likely lead to unpredictable results.

When processing is disabled, the reserved registers are configured such that audio signals presented at the inputs of the codecs are passed directly through to the corresponding outputs. Before enabling processing, the reserved registers need to be configured as desired.

## 5.4 Modifying PROCESS\_AUDIO

As configured, the *PROCESS\_AUDIO* (see *process\_samples.asm*), subroutine is a copy of the *PASS\_THROUGH* (see *process\_samples.asm*) subroutine. *PASS\_THROUGH* copies the audio input data directly to the corresponding position in the output buffer. To implement custom audio processing routines, the code within the *PROCESS\_AUDIO* routine must be changed.

When entering *PROCESS\_AUDIO*, the pointer registers R0 points to the base of the block that contains the most recent input samples obtained from the codecs as shown in Figure 4-3. The offset *TX\_BUFF\_BASE-RX\_BUFF\_BASE*, the difference between the start addresses of the input and output buffers, is the offset used to obtain the address of the next available output buffer block. The registers R2, R3, R4 and R5 should not be used as pointers into the output buffer for the audio processing routine since these are subject to change depending on how the output channels are mapped (see Section 5.6). A

channel offset of 0, 1, 2, or 3 must be added to address the specific channels within the input and output buffer blocks.

*PROCESS\_AUDIO* must complete all processing within a sample period in order to keep from losing valid input or output data when the next interrupt occurs. Failure to complete in time may cause stutters or other noise in the output audio.

## 5.5 Buffer Modification

In the default configuration, the input and output buffers in the Assembly Project Template application are sized so that there is enough space for one current sample and two previous samples of both the input and output digital audio data for all eight channels on the board. Each buffer consists of *keep* blocks of *buffsize* words each (see *sb\_isr\_esais.asm*), as shown in Figure 4-3. In the default configuration, there are 3 blocks of 4 words.

If more sample history is desired (i.e., to implement higher order filters), the symbol *keep* may be changed. The assembler directives in *sb\_isr\_esais.asm* will align the buffers to proper boundaries so that the reserved pointer registers will operate in a circular fashion. As they are configured by default, the buffering structure readily allows second-order filtering to be implemented on all eight channels. The symbol *buffsize* should not be changed unless fewer channels are used or the buffer structure is changed.

## 5.6 Output Channel Mapping

There is room in the output buffers for eight independent channels of audio. By using separate pointers, one for each EASI output, each output channel pair does not need to pull its data from the corresponding input. Furthermore, all the outputs may be exactly the same as each other since they can pull from the same output buffer channels. An example of the usage of this mapping can be observed in the Demonstration Application (see *Symphony SoundBite: Demonstration Application*) where the DIP switch settings result in various mappings from the input to the output.