Discovery Session 9
Sound to Light

Overview

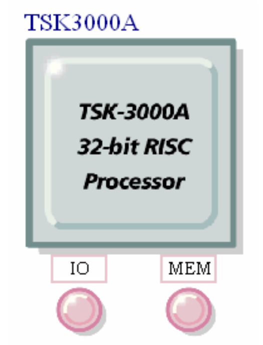
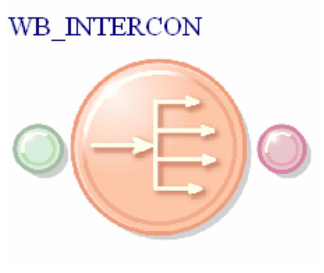
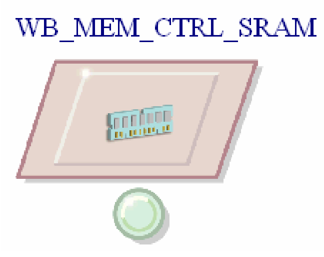
In previous sessions, we learned how to build a basic hardware design for controlling the LEDs, we looked at the use of virtual instruments, and finally processor cores. Now we want to extend our knowledge of the peripherals that are provided on the NB3000; exploring the use of the hi-fidelity audio interface. This tutorial design will take audio samples from the audio codec, pass them through for listening on the NB3000 speakers while driving two LEDs for left and right level indicators.

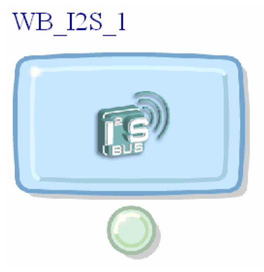
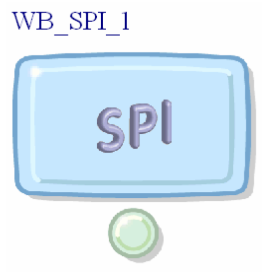
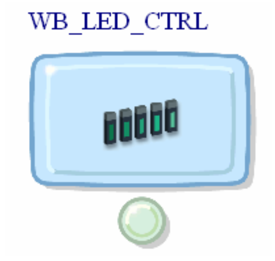
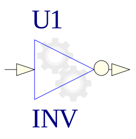
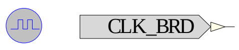
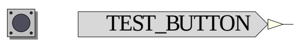
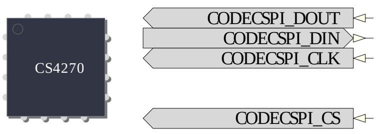
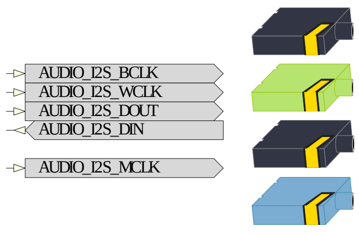
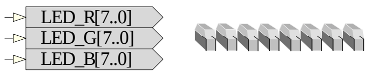
Prerequisites

This tutorial assumes you have a basic understanding of the process of placing and wiring objects in Altium Designer (including components, net labels / net connectivity, and wires / buses) and a basic understanding of the process of configuring and building a design using the Devices View (for specific details on this process, see **Discovery Session 1 – Exploring a Simple LED Driver**). It also assumes basic C programming skills. No additional information is required.

Design detail

This exercise uses the components listed in Table 1, to create the circuits shown in Figure 1 and Figure 2.

Component	Library	Name in Library
 TSK3000A TSK-3000A 32-bit RISC Processor IO MEM	OpenBus Palette	TSK3000A
	OpenBus Palette	Interconnect
	OpenBus Palette	SRAM Controller

	OpenBus Palette	Audio Streaming Controller
	OpenBus Palette	SPI Controller
	OpenBus Palette	LED Controller
	FPGA Configurable Generic.IntLib	INV
	FPGA NB3000 Port-Plugin.IntLib	CLOCK_BOARD
	FPGA NB3000 Port-Plugin.IntLib	TEST_BUTTON
 CS4270 CODECSPI_DOUT CODECSPI_DIN CODECSPI_CLK CODECSPI_CS	FPGA NB3000 Port-Plugin.IntLib	AUDIO_CODEC_CTRL
 AUDIO_I2S_BCLK AUDIO_I2S_WCLK AUDIO_I2S_DOUT AUDIO_I2S_DIN AUDIO_I2S_MCLK	FPGA NB3000 Port-Plugin.IntLib	AUDIO_CODEC
	FPGA NB3000 Port-Plugin.IntLib	LEDS_RGB

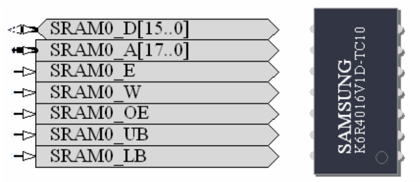
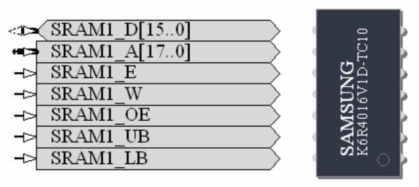
 SRAM0_D[15..0] SRAM0_A[17..0] SRAM0_E SRAM0_W SRAM0_OE SRAM0_UB SRAM0_LB SAMSUNG K6R4016V1D-TC10	FPGA NB3000 Port-Plugin.IntLib	SRAM0
 SRAM1_D[15..0] SRAM1_A[17..0] SRAM1_E SRAM1_W SRAM1_OE SRAM1_UB SRAM1_LB SAMSUNG K6R4016V1D-TC10	FPGA NB3000 Port-Plugin.IntLib	SRAM1

Table 1. List of components required by the design

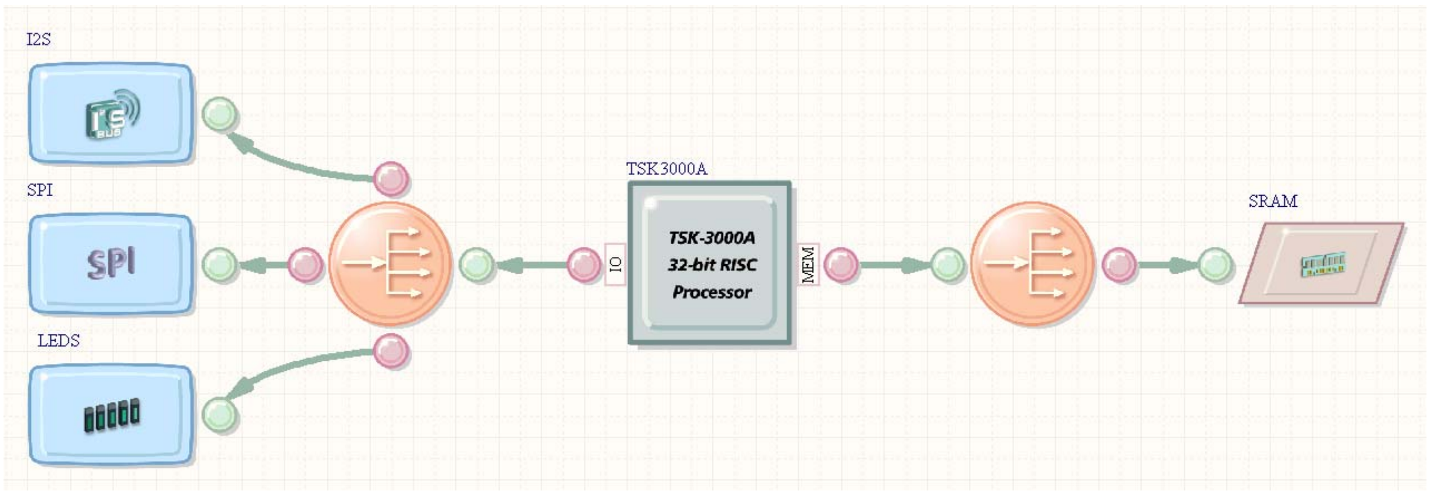


Figure 1. OpenBus document for the Snd2Light design.

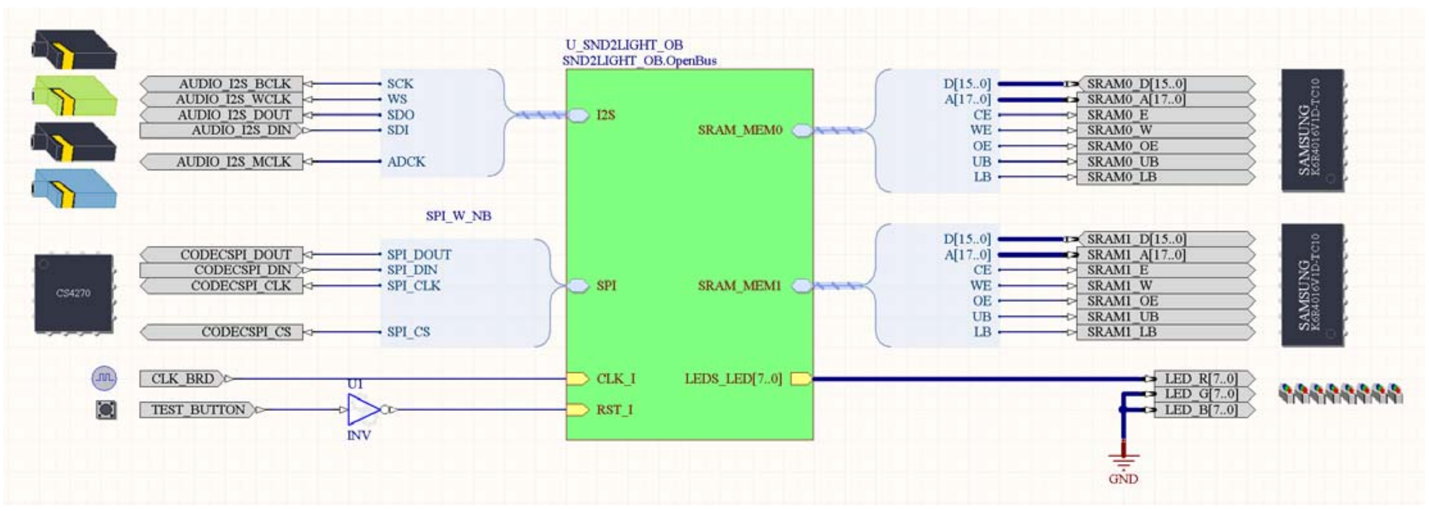
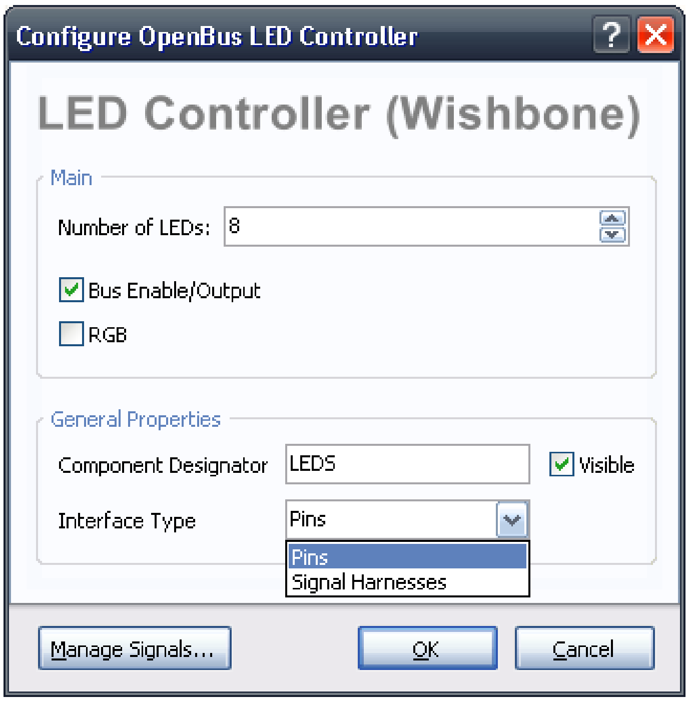


Figure 2. Schematic top sheet for the Snd2Light design.

Tutorial steps – preparing the OpenBus hardware

1. As with earlier tutorials, the hardware for this design will be captured using OpenBus for the processor and supporting I/O interface logic. It will also need a top-level schematic to connect that logic out to the FPGA device pins, and an FPGA project file. Create a new FPGA project, schematic sheet and OpenBus document, and save and name each of them `Snd2Light.PrjFpg`, `Snd2Light.SchDoc` and `Snd2Light_OB.OpenBus`, respectively.
2. Make the OpenBus document the active document, and place the following components from the OpenBus Palette:
   a. TSK3000A processor
   b. 2 Interconnect connectors (place one on either side of the processor)
   c. SRAM Controller
   d. LED Controller
   e. Audio Streaming Controller
   f. SPI Controller
3. Arrange the OpenBus components as shown in Figure 1.
4. The Interconnect on the I/O side of the processor needs a total of 3 slave ports, add these using the **Add Port** button on the toolbar.
5. **Place Links** between the OpenBus components (shown in Figure 1).
6. Each OpenBus component must now be configured. Configure the LED Controller component as shown in Figure 3, then click **OK** to close the dialog.



Configure OpenBus LED Controller

LED Controller (Wishbone)

Main

Number of LEDs: 8

[✓] Bus Enable/Output

[ ] RGB

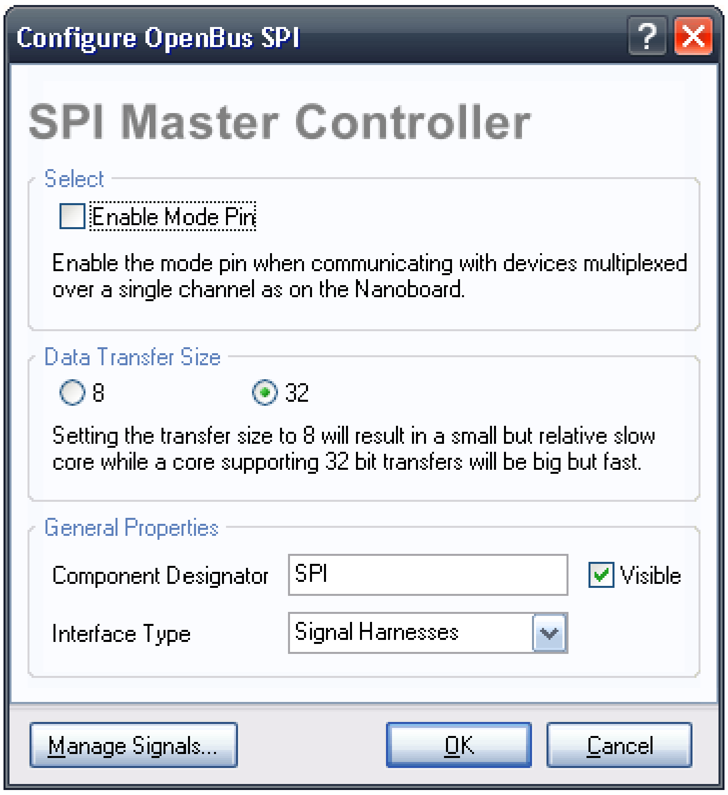
General Properties

Component Designator: LEDS [✓] Visible

Interface Type: Pins

Figure 3. Configure the LED Controller OpenBus component.

7. Configure the SPI Controller as shown in Figure 4 (un-check **Enable Mode Pin**), then click **OK** to close the dialog.


Configure OpenBus SPI

SPI Master Controller

Select

[ ] Enable Mode Pin

Enable the mode pin when communicating with devices multiplexed over a single channel as on the Nanoboard.

Data Transfer Size

8 [✓ 32]

Setting the transfer size to 8 will result in a small but relative slow core while a core supporting 32 bit transfers will be big but fast.

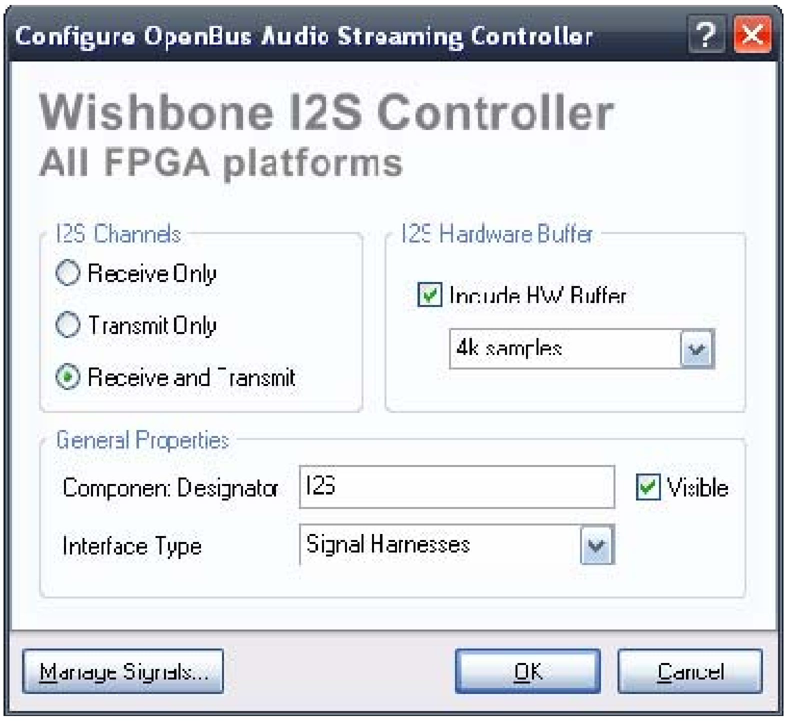
General Properties

Component Designator: SPI [✓ Visible]

Interface Type: Signal Harnesses

Figure 4. Configure the SPI Controller.

8. Configure the Audio Streaming Controller as shown in Figure 5, and then click **Manage Signals** to open the *OpenBus Signal Manager* dialog.


Configure OpenBus Audio Streaming Controller

Wishbone I2S Controller
All FPGA platforms

I2S Channels

Receive Only

Transmit Only

[✓ Receive and Transmit]

I2S Hardware Buffer

[✓ Include HW Buffer]

4k samples

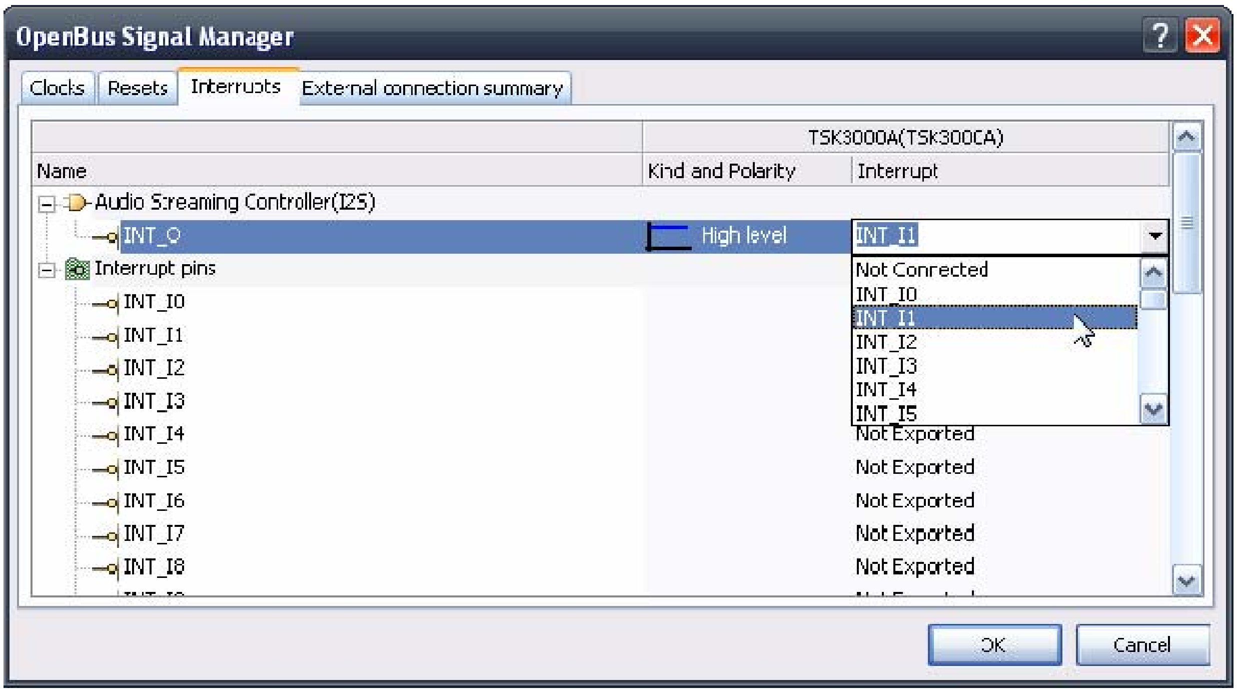
General Properties

Component Designator: I2S [✓ Visible]

Interface Type: Signal Harnesses

Figure 5. Configure the Audio Streaming Controller.

9. Set the **Interrupt** for the **Audio Streaming Controller (I2S)** to INT_I1, as shown in Figure 6.

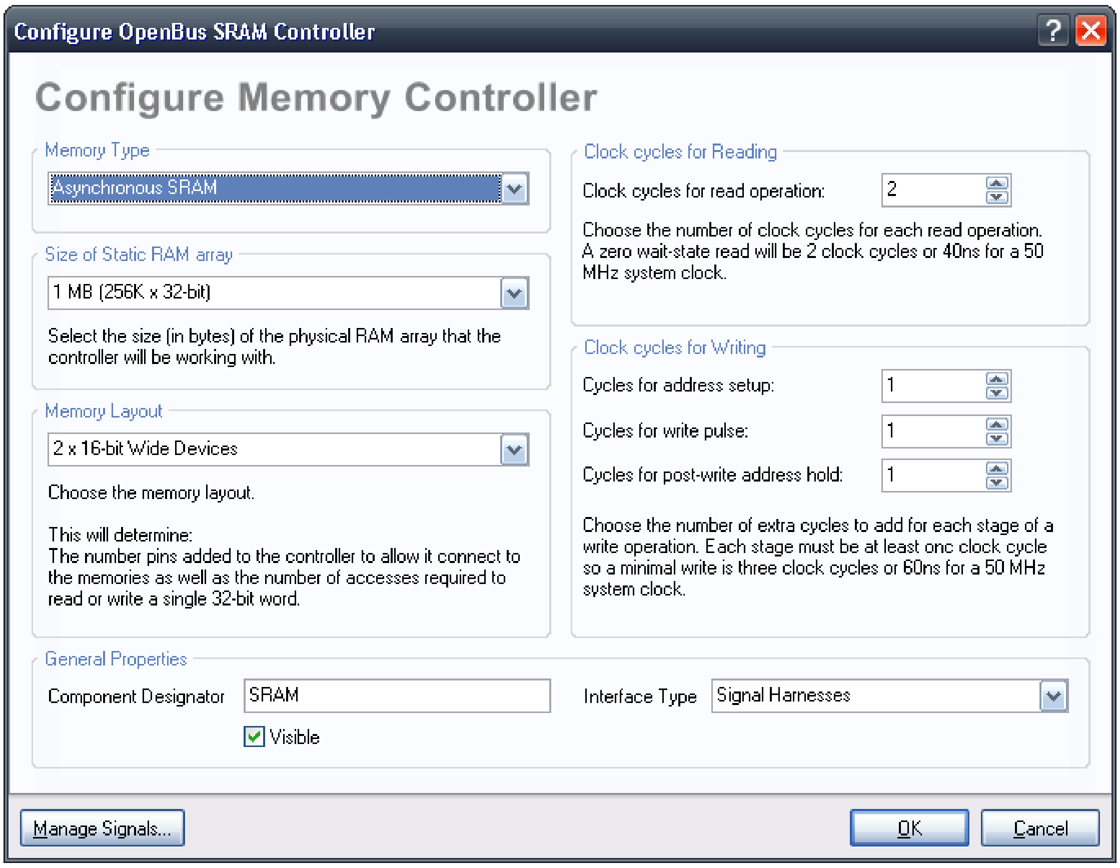

OpenBus Signal Manager

Clocks | Resets | Interrupts | External connection summary

Name	TSK3000A(TSK300CA) Kind and Polarity	Interrupt
Audio Streaming Controller(I2S)		
INT_O	High level	INT_I1
Interrupt pins		
INT_I0		Not Connected
INT_I1		INT_I0
INT_I2		INT_I1
INT_I3		INT_I2
INT_I4		INT_I3 / INT_I4 / INT_I5 / Not Exported
INT_I5		Not Exported
INT_I6		Not Exported
INT_I7		Not Exported
INT_I8		Not Exported

Figure 6. Set the Audio Streaming Controller to use Interrupt 1.

10. Configure the OpenBus SRAM Controller to: **Memory Type** Asynchronous SRAM, **Size** will be 1MB, **Layout** 2x16-bit Wide Devices, and the **Designator** to SRAM. Leave the other options at their default state, as shown in Figure 7.


Configure OpenBus SRAM Controller

Configure Memory Controller

Memory Type: Asynchronous SRAM

Size of Static RAM array: 1 MB (256K x 32-bit)

Select the size (in bytes) of the physical RAM array that the controller will be working with.

Memory Layout: 2 x 16-bit Wide Devices

Choose the memory layout.

This will determine:
The number pins added to the controller to allow it connect to the memories as well as the number of accesses required to read or write a single 32-bit word.

Clock cycles for Reading

Clock cycles for read operation: 2

Choose the number of clock cycles for each read operation. A zero wait-state read will be 2 clock cycles or 40ns for a 50 MHz system clock.

Clock cycles for Writing

Cycles for address setup: 1

Cycles for write pulse: 1

Cycles for post-write address hold: 1

Choose the number of extra cycles to add for each stage of a write operation. Each stage must be at least one clock cycle so a minimal write is three clock cycles or 60ns for a 50 MHz system clock.

General Properties

Component Designator: SRAM

✓ Visible

Interface Type: Signal Harnesses

Figure 7. Configure the memory as 1 MB of SRAM.

11. The Interconnect components will be correctly configured, as Altium Designer automatically detects what is connected to them.
12. Right-click on the TSK3000 processor and select **Configure TSK3000A** from the floating menu. Set the **Internal Processor Memory** to 32 K Bytes. Click OK to close the **Configure OpenBus TSK3000A** dialog.
13. This completes the OpenBus part of the design, save the OpenBus document.

Tutorial steps – preparing the remaining FPGA hardware

14. Switch to the Snd2Light schematic document. The schematic is used to wire the circuitry on the OpenBus document through to the FPGA device pins, and can also include other FPGA hardware that is not available as OpenBus components.
15. To make the OpenBus document a child of the schematic, select **Design»Create Sheet Symbol from Sheet or HDL** from the menus. When the *Choose Document to Place* dialog opens, select Snd2Light_OB.OpenBus and click **OK**. A sheet symbol will appear floating on the cursor, position it approximately in the middle of the schematic sheet.
16. Resize the Sheet Symbol, and reposition the Sheet Entries to approximately match the Sheet Symbol shown in Figure 2.
17. Right-click on the Snd2Light.PrjFpg project file in the Projects panel, and select **Compile** from the menu. When the project is compiled the OpenBus document will move to become a child of the schematic document, as shown in Figure 8. **Note:** *You will receive compiler errors in the Messages panel because we have not completed the wiring yet – you can ignore these and close the Messages panel for now.*

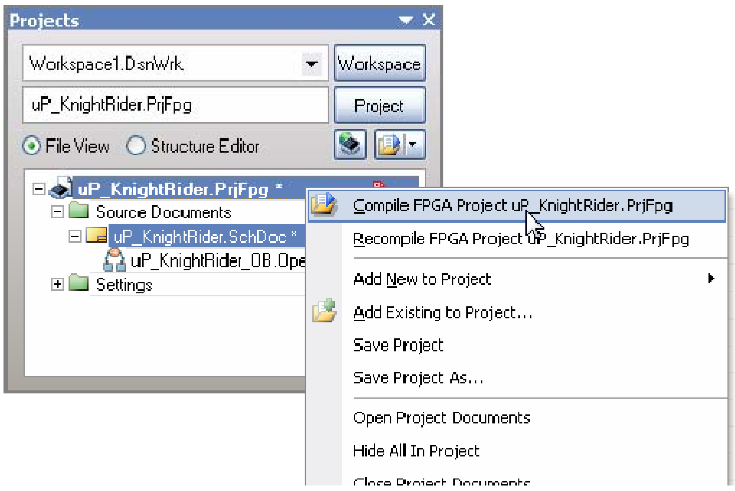


Figure 8. When the project is compiled the OpenBus document will become a child document of the schematic.

18. Using the detail in Table 1 as a reference, locate and place the following components onto the schematic, arranging them approximately in the positions shown in Figure 2:
    a. CLOCK_BOARD
    b. TEST_BUTTON
    c. INV
    d. LEDS_RGB
    e. AUDIO_CODEC_CTRL
    f. AUDIO_CODEC
    g. SRAM0
    h. SRAM1

19. Wire up the components on the lower end of the sheet symbol, as shown in Figure 9.

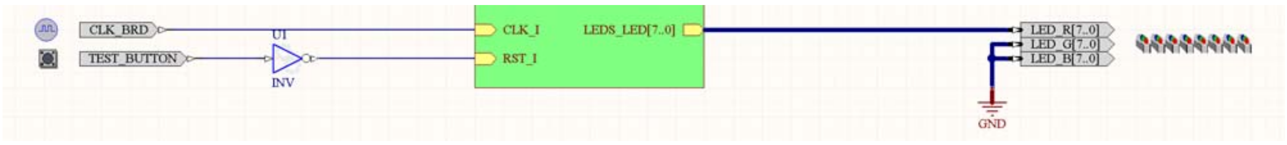


Figure 9. Wire the lower section of circuitry.

20. To wire the memory components on the right-hand side of the sheet symbol, right click on the Sheet Entry **SRAM_MEM0**, and choose **Sheet Entry Actions»Place Harness Connector of Type…** from the floating menu.

21. A Harness Connector will appear floating on the cursor. It may be oriented the wrong way, if it needs to be flipped along the X axis press the X key on the keyboard. Place it so that the tip of the brace touches the Sheet Entry, as shown in Figure 10.

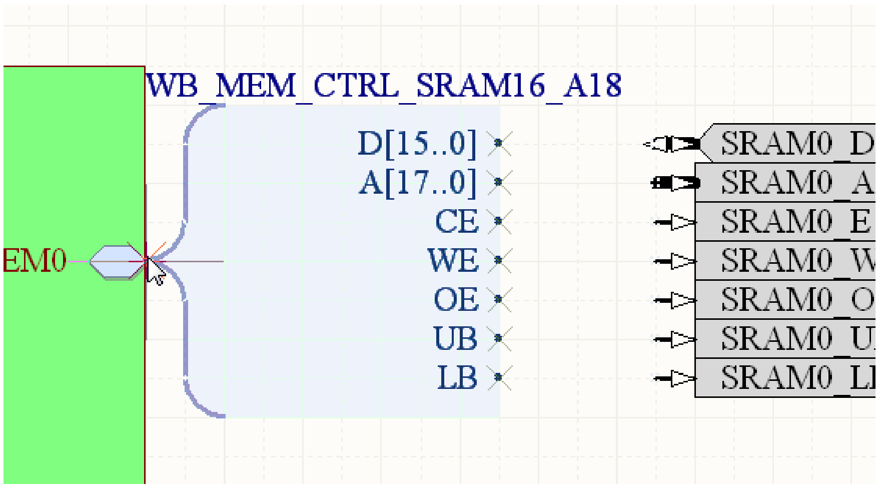


Figure 10. Flip the Signal Harness and place it so that it touches the Sheet Entry.

22. To drag the Harness Connector and automatically add the Harness line, hold Ctrl and click and hold on the Harness Connector, then drag it across so that each Harness Entry touches a pin on the memory port plug-in, as shown in Figure 11.

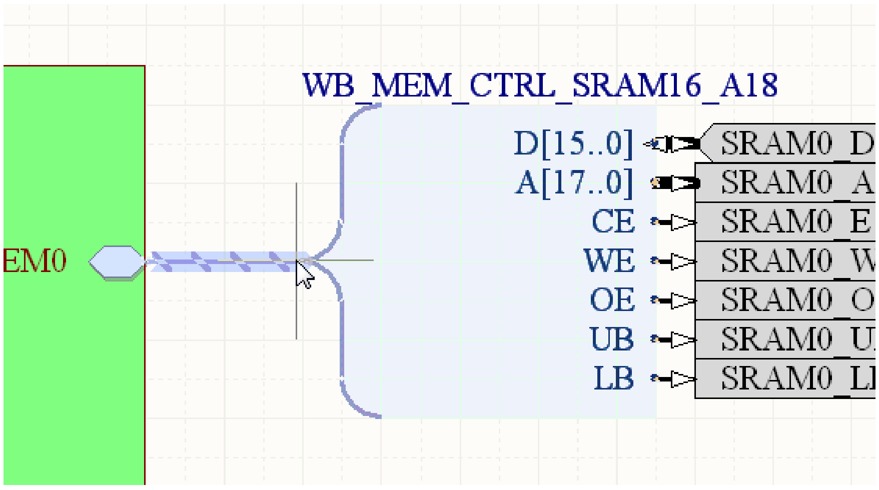


Figure 11. Drag the Harness Connector so that each entry touches a port on the memory port plug-in.

23. Repeat this process for the second memory Sheet Entry, **SRAM_MEM1**, connecting it to the second memory port plug-in.

24. Place Harness Connectors and Signal Harnesses for the AUDIO_CODEC and the SPI_BUS using the same technique.
25. To annotate all of the components (assign designators), select **Tools»Annotate Schematics Quietly** from the menus.
26. To check that there are no errors in the schematic, compile the project using the **Project»Recompile FPGA Project** command. The Messages panel will detail any errors or warnings.
27. Resolve any other errors or warnings, and **Save All** the files.

Tutorial steps – mapping the hardware design to the target hardware

28. To create the connectivity from the ports on the top schematic sheet through to the actual pins on the target FPGA, the design must be *constrained*. This is done by constraint files, which detail port-to-pin mapping, along with other relevant design specifications, such as clock allocations, target device, and so on. To constrain the design you will need an NB3000 connected to your PC via a USB cable, once you have this, open the Devices view (**View»Devices**) in Altium Designer and enable the Live checkbox at the top right of the view.
29. An icon of the NB3000 will appear, right-click on it and select **Configure FPGA Project»Snd2Light.PrjFpg** from the menu, as shown in Figure 12.

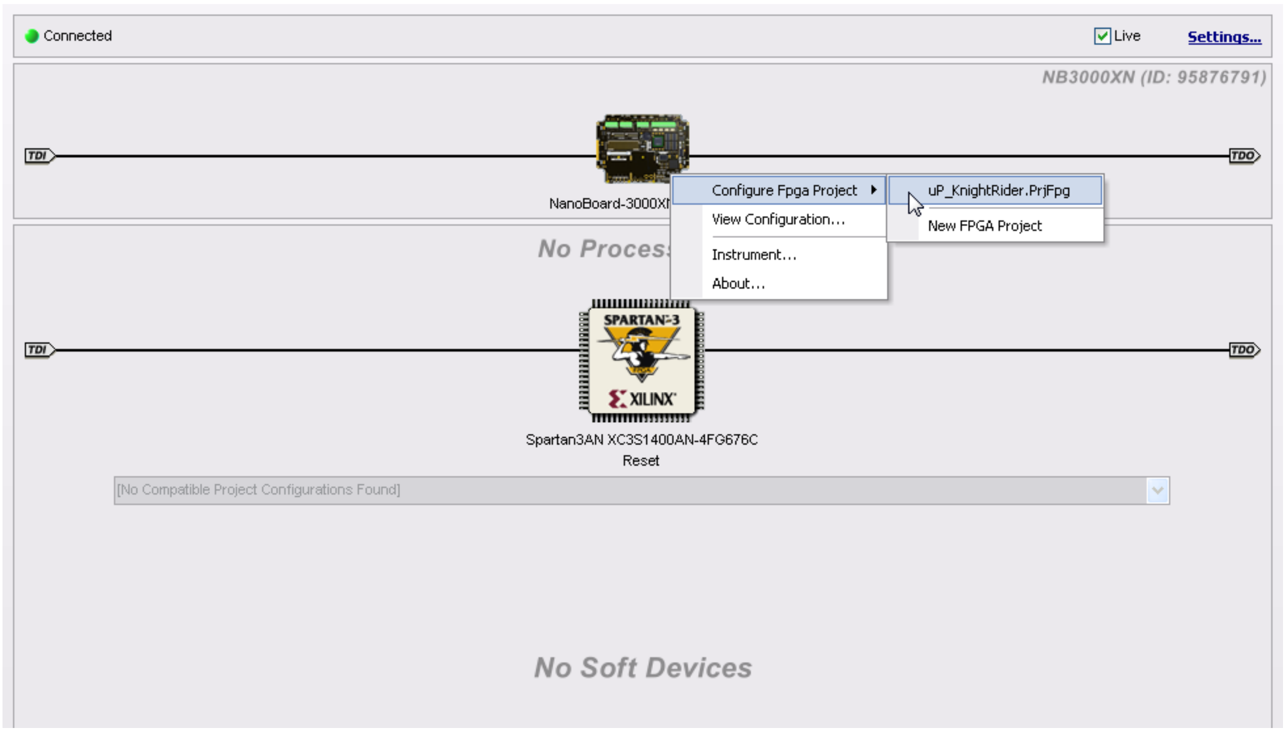


Figure 12. Configure the design to run on the NB3000.

30. The *Configuration Manager* will open automatically, showing the constraint files that have been detected and added to the project, and the configuration that has been created. A configuration is simply a set of constraint files, using configurations allows you to divide your constraints into separate constraint files. Click OK to close the dialog.
31. Select **File»Save All** to save your work. The hardware design is now complete; the next step is to write the embedded code.

Tutorial steps – creating the embedded project

32. Create a new embedded project, and save it as `Snd2Light.PrjEmb` in a sub-folder below the FPGA project called `\Embedded`.
33. To make the embedded project a child of the FPGA project, switch the Projects panel to the **Structure Editor** mode, then right click on the icon for the TSK3000A processor and select **Set Embedded Project** from the menu, as shown in Figure 13.

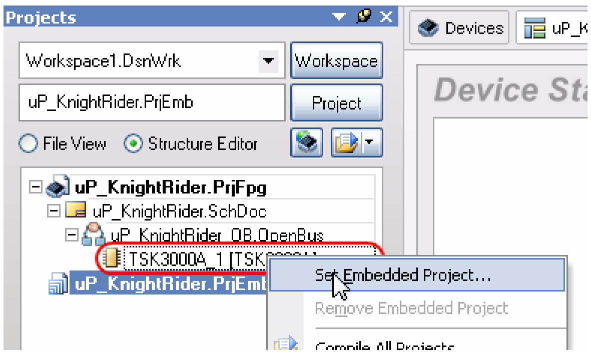
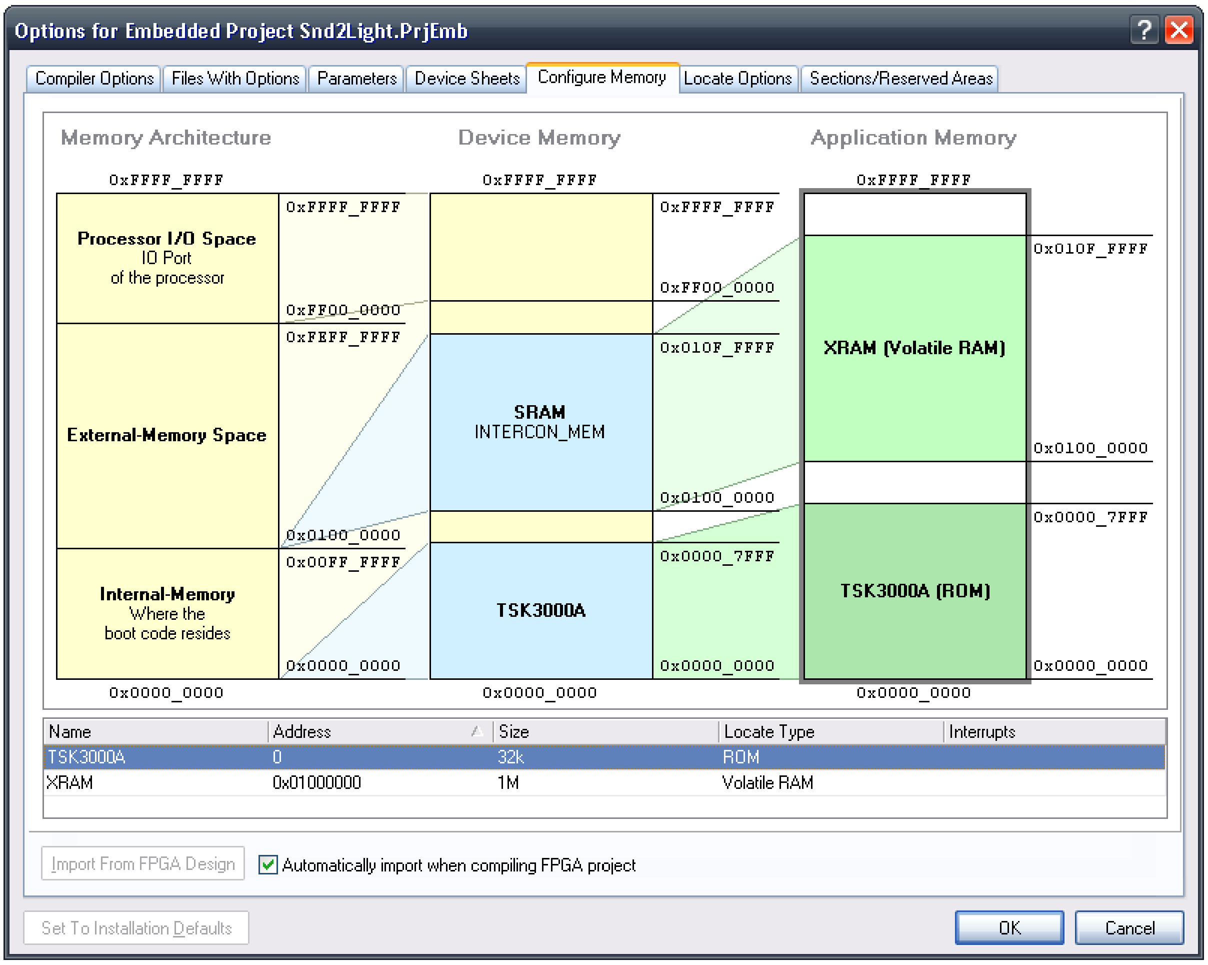


Figure 13. Make the embedded project a child of the FPGA project.

34. Switch the Projects panel back to **File View** mode.
35. Right-click on the **Snd2Light.PrjEmb** embedded project, and click **Project Options**. Click on the **Configure Memory** tab – you can see that the memory map defined in hardware has been automatically imported into the embedded project, as shown in the upper part of Figure 14.
36. Optionally you can rename the SRAM section to XRAM (to clarify it is external to the FPGA) though this is not strictly necessary.

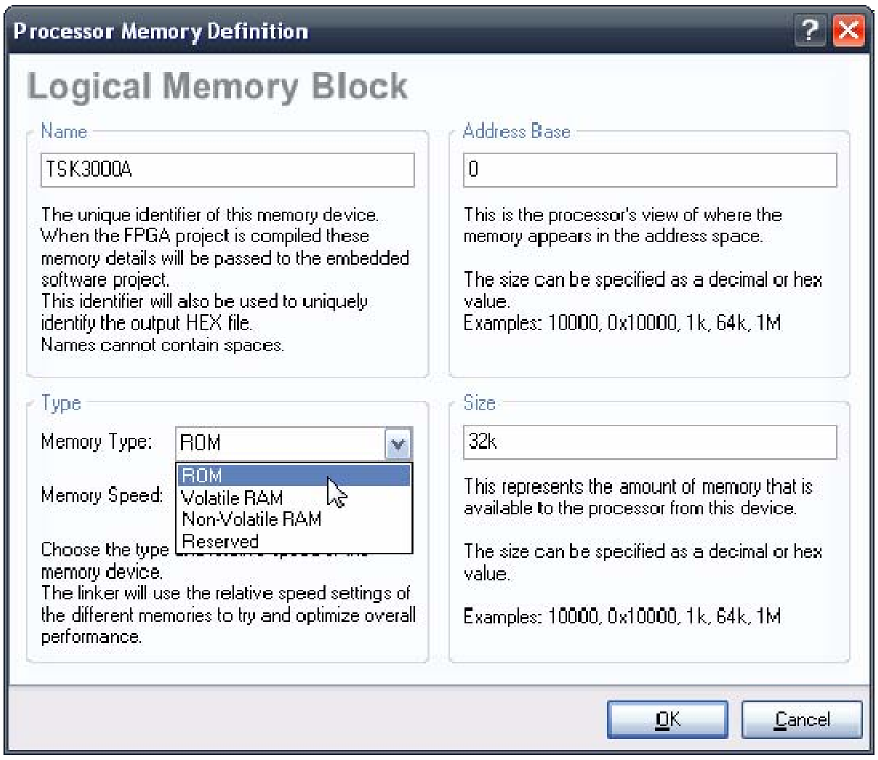


Name	Address	Size	Locate Type	Interrupts
TSK3000A	0	32k	ROM	
XRAM	0x01000000	1M	Volatile RAM	

☑ Automatically import when compiling FPGA project

Figure 14. Embedded project memory map.

37. Double-click on the TSK3000A row in the memory table (Figure 14) and configure the memory as ROM instead of Non-Volatile RAM, as shown in Figure 15. Doing this means that the TSK3000A will boot from internal memory automatically, whenever the design is downloaded to the NB3000. Click **OK** to apply the changes and close both dialogs.


Processor Memory Definition

Logical Memory Block

Name: TSK3000A

The unique identifier of this memory device. When the FPGA project is compiled these memory details will be passed to the embedded software project. This identifier will also be used to uniquely identify the output HEX file. Names cannot contain spaces.

Address Base: 0

This is the processor's view of where the memory appears in the address space.

The size can be specified as a decimal or hex value. Examples: 10000, 0x10000, 1k, 64k, 1M

Type

Memory Type: ROM

Memory Speed:

Choose the type … memory device. The linker will use the relative speed settings of the different memories to try and optimize overall performance.

Size: 32k

This represents the amount of memory that is available to the processor from this device.

The size can be specified as a decimal or hex value.

Examples: 10000, 0x10000, 1k, 64k, 1M

Figure 15. Setting the boot memory to Memory Type: ROM.

38. Add a SwPlatform file to the embedded project, and save it as `Snd2Light.SwPlatform` in this embedded sub-folder.
39. In the Software Platform document, the **Import from FPGA** button will now be active, click this to instruct Altium Designer to examine the FPGA project and attach any required I/O wrappers. The low-level wrappers for the I2S Master Controller, the LED Controller and the SPI Master Controller will be added, as shown in Figure *16*.

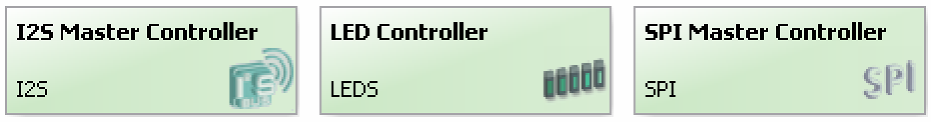


Figure 16. First add the wrappers.

40. Click on each wrapper in turn, then click the **Grow Stack Up** button, adding the driver of the same name to each wrapper, as shown in Figure *17*.

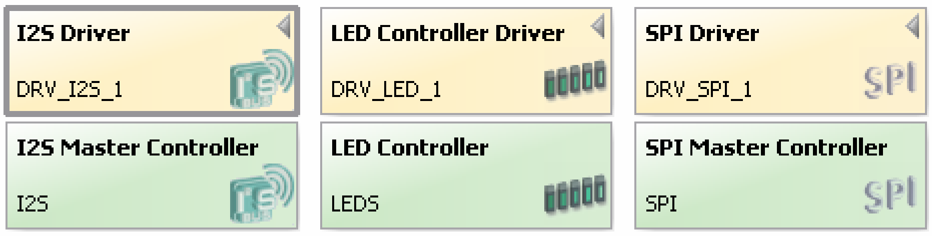


Figure 17. Then add the drivers.

41. Select the SPI Driver, and configure the **SPI channels** to 18 in the settings on the right hand side of the SwPlatform.

42. Select the SPI Driver and click the **Grow Stack Up** button to open the *Grow Stack* dialog. The audio codec on the NB3000 is a Cirrus Logic CS4270, select the driver for this and click OK to close the dialog, as shown in Figure 18. This device uses SPI for configuration and control, and I2S for digitized audio data transfer.

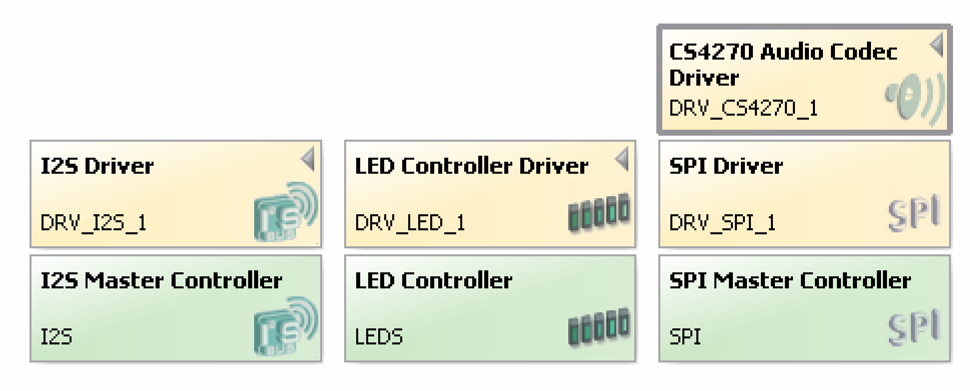


Figure 18. Then add the specific audio driver.

43. With this driver selected in the SwPlatform, click the **Grow Stack Up** button again to open the *Grow Stack* dialog and add the Audio Service, as shown in Figure *19*.

44. Set the **Audio Context** settings on the right hand side of the SwPlatform to: **Default sample frequency** `44100`, **Default mode** `Stereo`, **Default sample size** `16`.

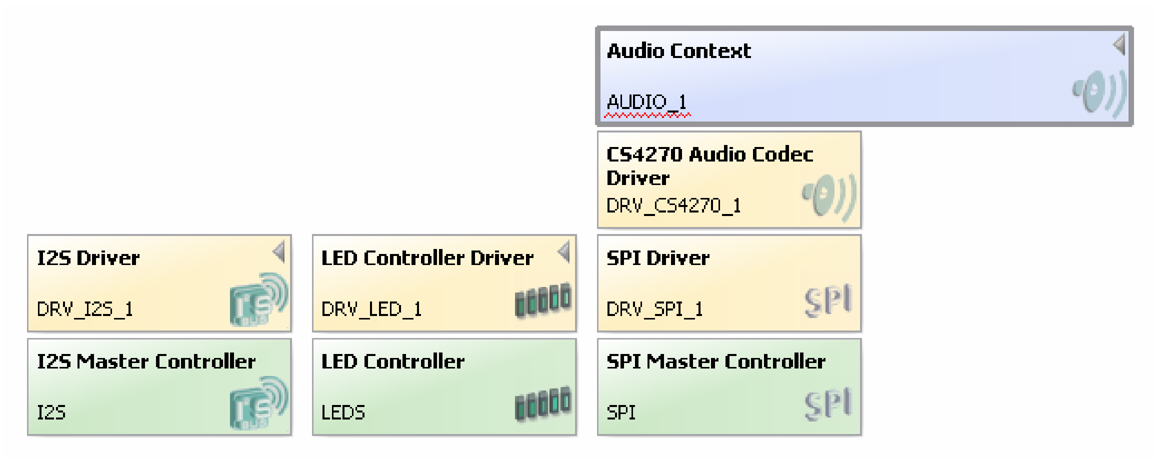


Figure 19. Then add the Audio Context, to configure the Audio Codec.

45. This Audio Context must now be linked to the I2S driver, to do this click the **Link to Existing Stack** button. All stack elements will grey out, except the I2S Driver, select it to create the link, as shown in Figure 20.

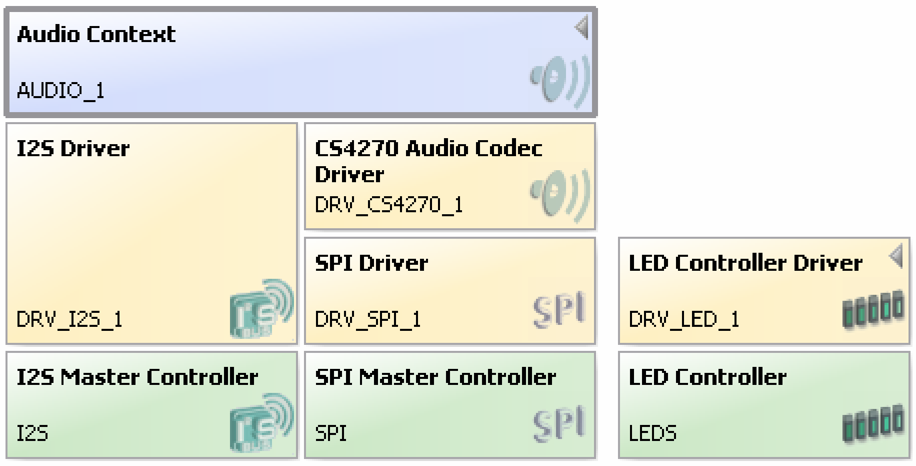


Figure 20. Then Link the Context to the I2S, so that Altium Designer knows both of these stacks are used with the 4270 Audio Codec.

46. Select **File»Save All** to save all your work.

Tutorial steps – writing the embedded code

Just as in previous sessions, we need to include and use functions provided by the Software Platform Builder for our embedded code. It is strongly recommended to explore the driver and context function documentation by using the **F1** help key within the Software Platform Builder.

47. Firstly, include the necessary header files for the C standard integer definitions as well as the drivers:

```
#include <stdint.h>
#include <devices.h>
#include <audio.h>
#include <drv_led.h>
#include <led_info.h>
```

48. Next we'll define our internal audio buffer size:

```
#define AUDIO_BUF_SIZE      512
#define LEFT                0
#define RIGHT               1
```

49. Add the following arrays for our stereo pass-through buffer, as well as the left and right buffers used for the absolute averaging filters:

```
/* bufffers */
// audio
int16_t stereo_buf[AUDIO_BUF_SIZE] = {0};
int16_t audio_buf_l[AUDIO_BUF_SIZE/2] = {0};
int16_t audio_buf_r[AUDIO_BUF_SIZE/2] = {0};
```

50. Of course, we need handles for the audio and LED drivers:

```
// contexts for drivers
audio_t *audio;
led_t   *leds;
```

51. Now, add a few function prototypes for some handy functions:

```
/* function prototypes */
int get_audio(int16_t *buffer, int size);
int put_audio(int16_t *buffer, int n);
uint8_t abs_ave(int16_t *buffer, int n);
void update_intensity(uint8_t intensity, uint8_t channel);
```

52. Now for our main function:

```
void main(void)
{
    int i, j;
    audio = audio_open(AUDIO_1);
    leds  = led_open(LEDS);
    led_turn_on(leds, LEDS_LED0);
    led_turn_on(leds, LEDS_LED7);

    while (1)
    {
        // get audio to left and right buffers for processing.
        get_audio(stereo_buf, AUDIO_BUF_SIZE);
        for (i = 0, j = 0; i < AUDIO_BUF_SIZE ; i++, j++)
        {
          audio_buf_r[j] = stereo_buf[i++];
          audio_buf_l[j] = stereo_buf[i];
        }

        // Loop left and right channels through.
        put_audio(stereo_buf, AUDIO_BUF_SIZE);

        // Put the D.C. average value of the wavelet on LEDs
        update_intensity(abs_ave(audio_buf_l, AUDIO_BUF_SIZE/2),LEFT);
        update_intensity(abs_ave(audio_buf_r, AUDIO_BUF_SIZE/2),RIGHT);
    }
}
```

53. Now, we add the full function definitions for `get_audio` and `put_audio`, which use the `audio_record` and `audio_play` functions of the audio context to fill our software buffers and write them out, respectively. These offer some ability to fill a much larger software buffer than what would be available in the hardware buffer:

```
int get_audio(int16_t *buffer, int n)
{
    int s;


    do
    {
        s = audio_record(audio, buffer, n);
        n -= s;
        buffer += s;
    }while (n != 0);

    return s;
}

int put_audio(int16_t *buffer, int n)
{
    int s;


    do
    {
        s = audio_play(audio, buffer, n);
        n -= s;
        buffer += s;
    } while (n != 0);

    return 0;
}
```

54. Finally, we add the function definitions for the absolute average filter that drives the LEDs from the `main` function, as well as the LED level decision function:

```
uint8_t abs_ave(int16_t *buffer, int n)
{
    int16_t cusum = 0;
    for (int i = 0; i < n; i++)
    {
        cusum += (buffer[i] < 0) ? -(buffer[i]/n) : buffer[i]/n;
    }
    return (uint8_t)(cusum);
}

void update_intensity(uint8_t intensity, uint8_t channel)
{
    uint8_t b0 = 0;
    uint8_t b1 = 0;
    uint8_t b2 = 0;
    uint8_t b3 = 0;
    if (intensity > 0xF0)
    {
        b0 = b1 = b2 = b3 = 0xFF;
    } else {
        if (intensity > 0xC0)
        {
            b0 = b1 = b2 = 0xFF;
            b3 = 0;
        } else {
            if (intensity > 0xA0)
            {
                b0 = b1 = 0xFF;
                b2 = b3 = 0;
            } else {
                if (intensity > 0x40)
                {
                    b0 = 0xFF;
                    b1 = b2 = b3 = 0;
                } else {
                    if (intensity < 0x10)
                    {
                        b0 = b1 = b2 = b3 = 0;
                    }
                    // if not down to 0x10 then no change.
                }
            }
        }
    }
    switch (channel)
    {
        case LEFT:
             led_set_intensity(leds, LEDS_LED7, b3);
             led_set_intensity(leds, LEDS_LED6, b2);
             led_set_intensity(leds, LEDS_LED5, b1);
             led_set_intensity(leds, LEDS_LED4, b0);
             break;
        case RIGHT:
             led_set_intensity(leds, LEDS_LED0, b3);
             led_set_intensity(leds, LEDS_LED1, b2);
             led_set_intensity(leds, LEDS_LED2, b1);
             led_set_intensity(leds, LEDS_LED3, b0);
             break;
        default: break;
    }
}
```

Tutorial steps – running your design

Now it's time to build and download the project:

55. Switch to **Device View** and click the **Program FPGA** button to build and download the design to the NB3000.
56. Once you have built and downloaded the design, you need to plug in a sound source (such as your PC or an MP3 Player) to the NB3000's LINE IN on the front edge of the board (see Figure 21).
57. Optionally, plug some stereo headphones into the HEADPHONES output on the front edge of the board, or listen to the sound play through on the NB3000's speakers (the speakers are disabled if headphones are plugged in).
58. Play some sounds and observe the LED's brightness on LED array, with respect to the sound source that's playing.

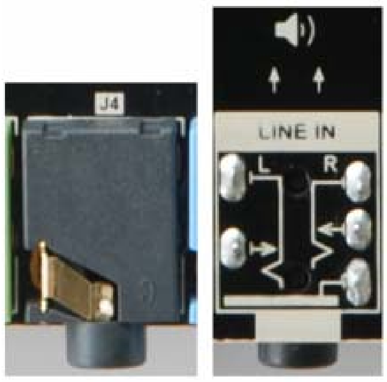


Figure 21. NB3000 Line Input.

Code Explanation

In this tutorial, we are configuring the Audio Context and underlying drivers and hardware to take care of configuring the codec and buffering the audio samples. The main C function simply passes the audio samples from input to output unchanged. The audio buffer has left and right audio samples interleaved - which is the nature of most digital audio interfaces - so our code has to split these into separate left and right buffers for actual stereo processing. In this case, we are simply taking a small segment of sound defined by the buffer length, and passing it through Absolute Value Averaging filters. The output of these filters then drives the LED intensity for left and right channel LEDs (see Figure 22).

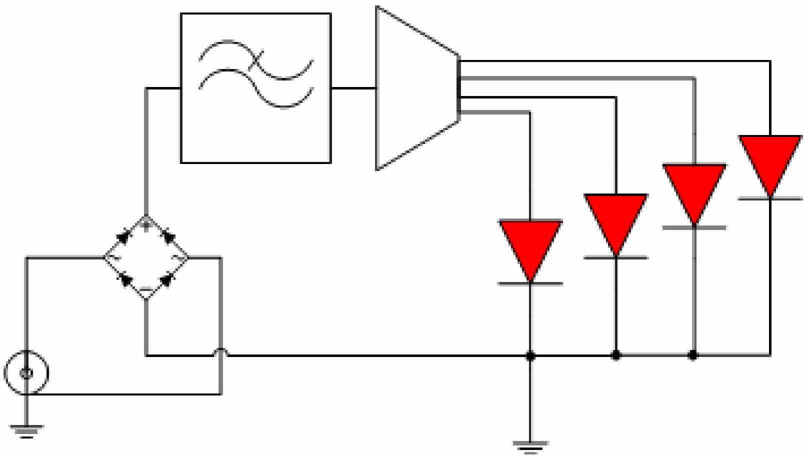


Figure 22. VU Meter Block Diagram (one channel).

Revision History

Date	Revision No.	Changes
23-Jul-2009	1.0	New document release