Discovery Session 5

Scrolling the LEDs with a microprocessor

Overview

In the earlier sessions, the tricolor LEDs were controlled via FPGA instruments, either manually, or using the NANOBOARD_INTERFACE with a script to drive the instrument. In this session a processor will be introduced to control the LEDs. Rather than placing the processor and surrounding interface logic as traditional schematic components, we will capture the hardware using OpenBus. OpenBus is a high-level capture system that removes all of the low-level wiring and interconnectivity detail, allowing the system to be built very quickly. We will implement the software portion of the design using Altium Designer's Software Platform Builder, which provides an abstract API layer between user code and the hardware.

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
WB_INTERCON	OpenBus Palette	Interconnect
WB_LED_CTRL	OpenBus Palette	LED Controller

WB_MEM_CTRL_SRAM	OpenBus Palette	SRAM Controller
CLK_BRD	FPGA NB3000 Port-Plugin.IntLib	CLOCK_BOARD
TEST_BUTTON	FPGA NB3000 Port-Plugin.IntLib	TEST_BUTTON
U? CLK INIT DELAY[7..0] FPGA_STARTUP8	FPGA Generic.IntLib	FPGA_STARTUP8
U? OR2S	FPGA Configurable Generic.IntLib	GATE (configured as a 2 input OR gate with one input inverted).
LED_R[7..0] LED_G[7..0] LED_B[7..0]	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

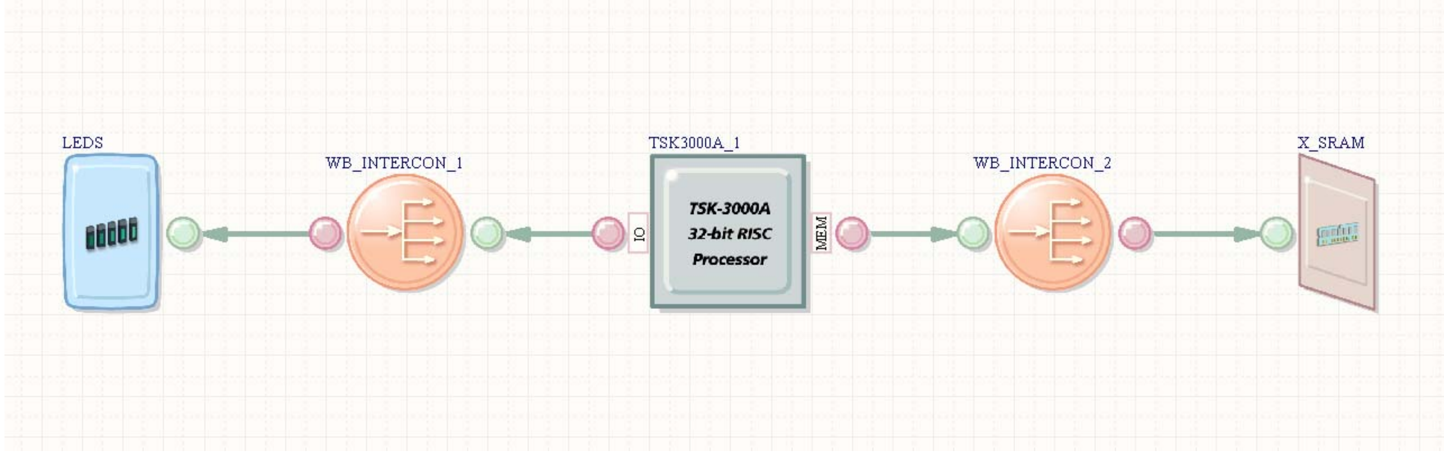

Figure 1. OpenBus document for the uP_KnightRider design.

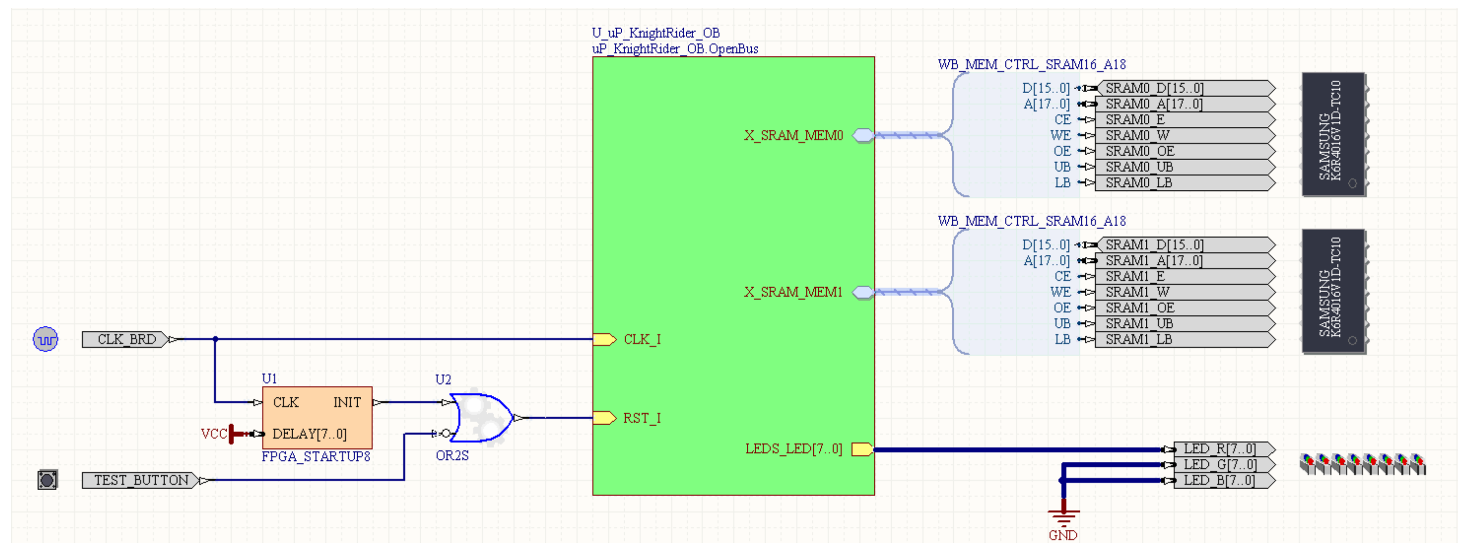

Figure 2. Schematic top sheet for the uP_KnightRider design.

Tutorial steps – preparing the OpenBus hardware

1. Select **File»New»Projects»FPGA Project** to create a new, blank FPGA Project.
2. Select **File»New»Schematic** to add a new blank schematic to your FPGA project. This schematic sheet will become the top level schematic in the project.
3. Select **File»New»OpenBus System Document** and add a new blank OpenBus document to your FPGA project.
4. Select **File»Save Project As** to save the project documents. You will first be prompted to save the schematic, followed by the OpenBus document, followed by the project itself. Name the documents `uP_KnightRider.SchDoc`, `uP_KnightRider_OB.OpenBus` and `uP_KnightRider.PrjFpg`, respectively.
5. Make the OpenBus document the active document, and display the OpenBus Palette. The Palette can be displayed via the **OpenBus** button, which is down the lower right of the workspace, or though the **View»Workspace Panels»OpenBus** menu entries.

6. From the **Processors** section of the OpenBus Palette, locate and click once to place the **TSK3000A** processor, positioning it approximately in the center of the OpenBus document.
7. From the **Connectors** section of the OpenBus Palette, locate and place two of the **Interconnect** connectors, placing one on either side of the TSK3000A.
8. From the **Peripherals** section of the OpenBus Palette, locate and place the **LED Controller**, placing it to the left of the OpenBus document.
9. From the **Memories** section of the OpenBus Palette, locate and place the **SRAM Controller**, placing it to the right of the OpenBus document.
10. Arrange the OpenBus components as shown in Figure 1. Note that you can rotate an OpenBus component while moving it by pressing the **Spacebar**. The OpenBus Ports (the smaller green or red circles on each component) can also be dragged to change their position, note that there must be a Master (green) port adjacent to each Slave (red) port.
11. To connect the ports on the OpenBus components, select the **Place»Link OpenBus Ports** command, or click the button on the toolbar, as shown in Figure 3. Click once on the Master (green) port, then click on the appropriate Slave (red) port, to connect the ports as shown in Figure 1. You can also wire in the reverse order, clicking on the slave and then the master port, Altium Designer will automatically detect the port relationships and set the direction correctly.

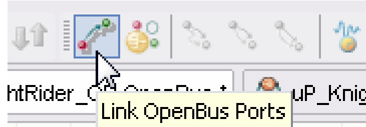

Figure 3. OpenBus ports are linked by running the Link OpenBus Ports command.

12. Each OpenBus component must now be configured. To configure the LED Controller component, double-click on it to open the *Configure OpenBus LED Controller* dialog. Disable the **RGB** option (since we are only using the RED mode), and set the **Component Designator** to `LEDS`, as shown in Figure 4.

Figure 4. Configuring the LED Controller OpenBus component.

13. Double-click on the memory controller on the right of the document, to open the Configure OpenBus SRAM Controller dialog. The memory controller can be configured to support a variety of different memory types, for this design we will be using Asynchronous SRAM – set the Memory Type option as required. The Size will be

1MB, Layout 2x16-bit Wide Devices, and the designator X_SRAM. Leave the other options at their default state, as shown in Figure 5.

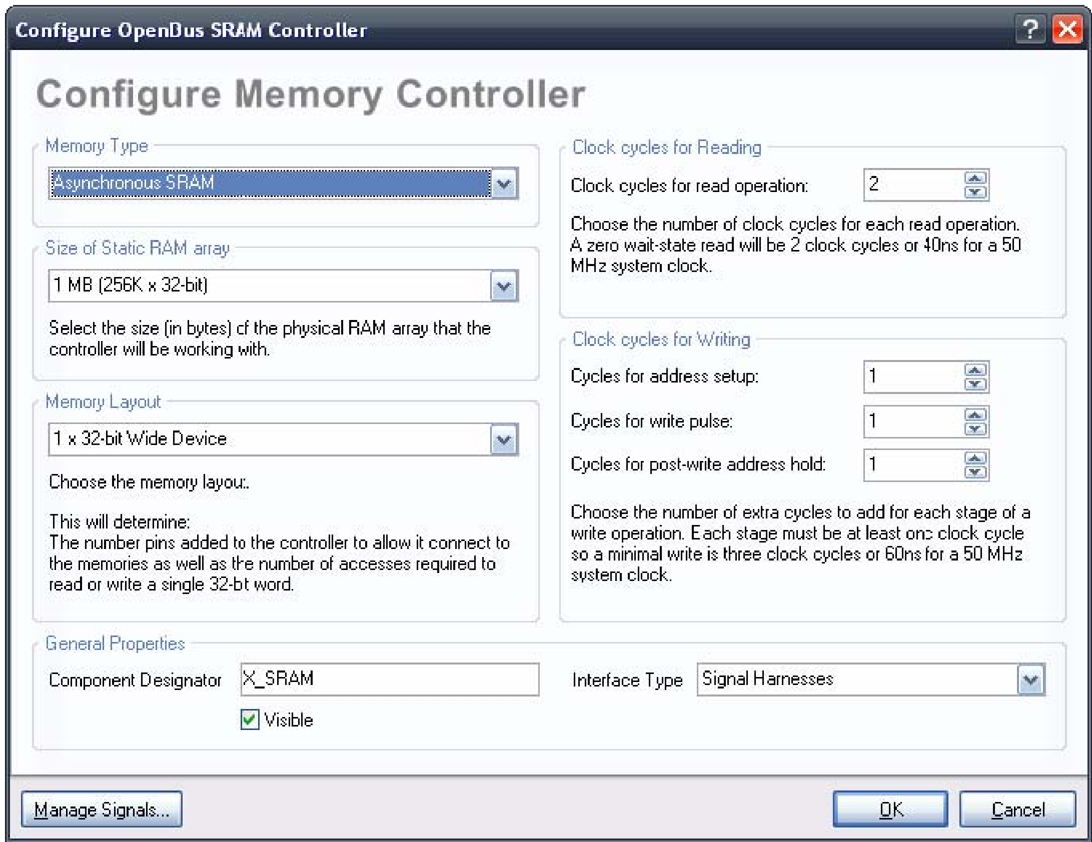

Figure 5. Configure the memory as 1 MB of SRAM.

14. The Interconnect components will be configured correctly, as Altium Designer automatically detects what is connected to them.
15. The processor has 3 different areas that can be configured: Memory, Peripherals, and the processor itself. To access each of these, right-click on the processor to display the floating menu. From this menu select **Configure TSK3000A**. This dialog is used to configure the hardware aspects of the TSK3000A, including the amount of internal memory, and if the on-chip debug capabilities are enabled. Leave these options at their default state.
16. Right-click again on the TSK3000A and select **Configure Processor Memory** from the floating menu. This dialog shows how the memory is allocated within the processor's address space. Close the dialog without changing any of the options.
17. Right-click again on the TSK3000A and select **Configure Processor Peripheral** from the floating menu. This dialog shows where the peripherals sit within the processor's address space. Close the dialog without changing any of the options.
18. This completes the OpenBus part of the design, save the OpenBus document.

Tutorial steps – preparing the remaining FPGA hardware

19. Switch to the uP_KnightRider 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.
20. 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 uP_KnightRider_OB.OpenBus and click **OK**. A sheet symbol will appear floating on the cursor, position it approximately in the middle of the schematic sheet.
21. Click once on the green Sheet Symbol to select it, then click and drag on the lower middle handle to resize it, as shown in Figure 6.
22. Click and drag on each Sheet Entry to reposition it, as shown in Figure 6. The exact location is not critical.

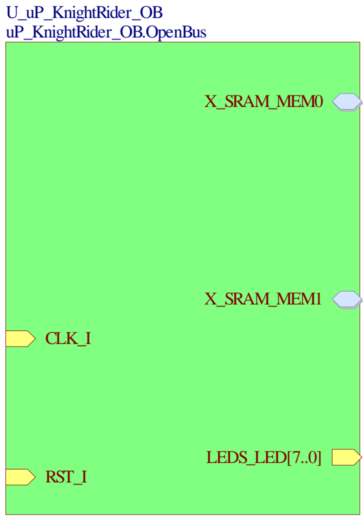

Figure 6. Resize the Sheet Symbol, and position the Sheet Entries.

23. Right-click on the uP_KnightRider.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 7. ***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.*

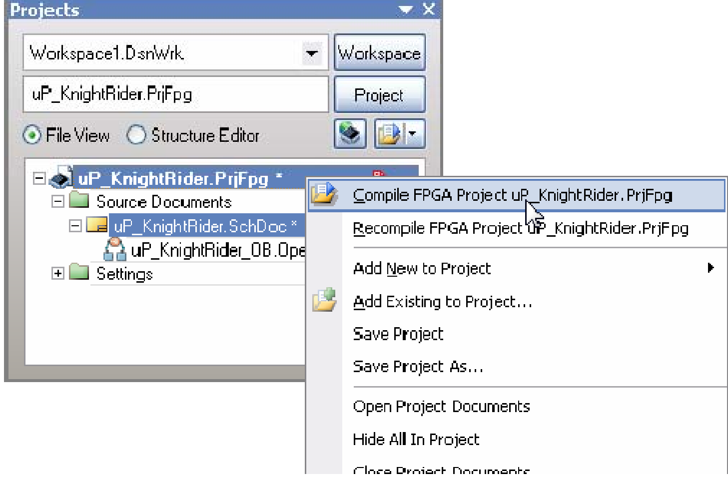

Figure 7. When the project is compiled the OpenBus document will become a child document of the schematic.

24. Place the following components onto the schematic, arranging them approximately in the positions shown in Figure 2:
    a. CLOCK_BOARD
    b. TEST_BUTTON
    c. FPGA_STARTUP8
    d. GATE
    e. LEDS_RGB
    f. SRAM0
    g. SRAM1
25. When you place the GATE component, you'll need to configure it (right-click menu) as a 2-input OR gate with one input inverted (see figure).

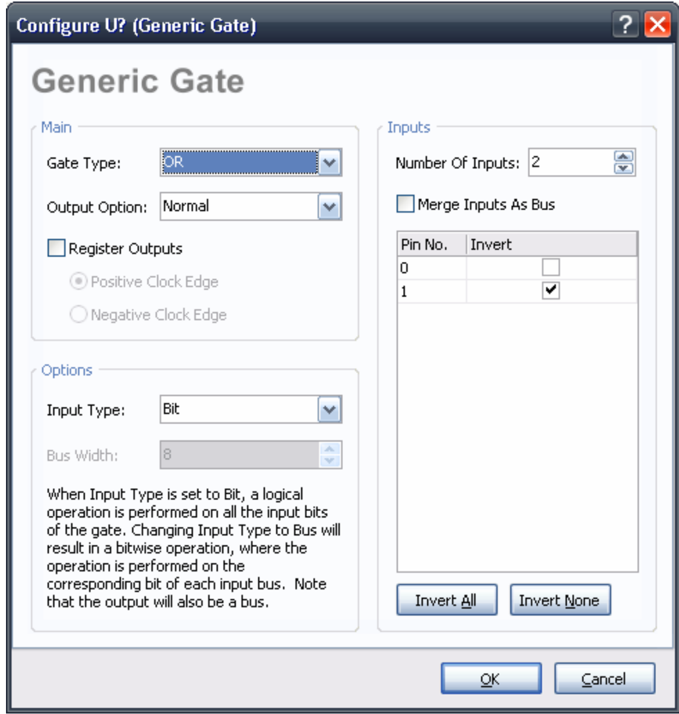

Figure 8. Configuring the GATE component as a 2-input OR gate with one input inverted.

26. Wire up the components on the left side of the sheet symbol, as shown in Figure 9. Note that the Delay pin on the FPGA_STARTUP8 component is wired to VCC using a VCC Bus Power Port.

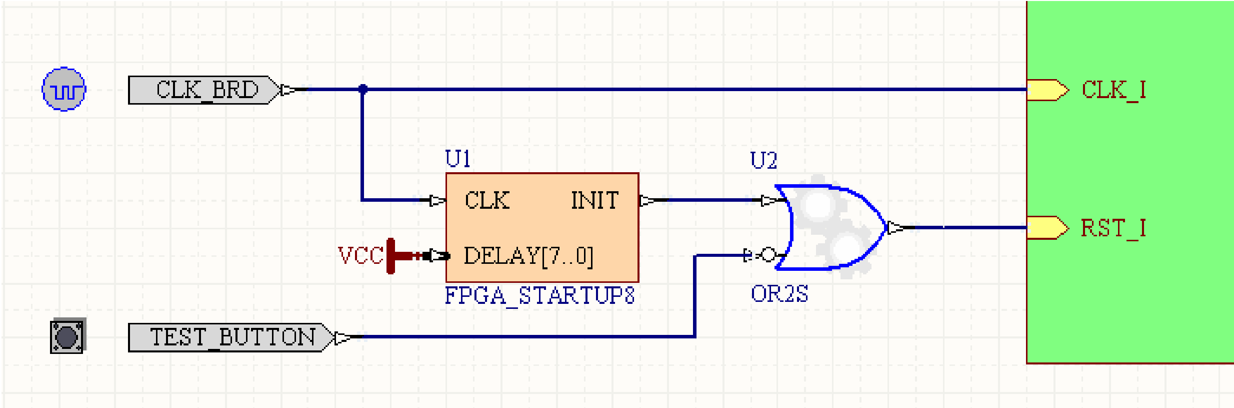

Figure 9. Wire the discrete components.

27. To wire the memory components on the right-hand side of the sheet symbol, right click on the Sheet Entry **X_SRAM_MEM0**, and choose **Sheet Entry Actions»Place Harness Connector of Type** from the floating menu.
28. 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.

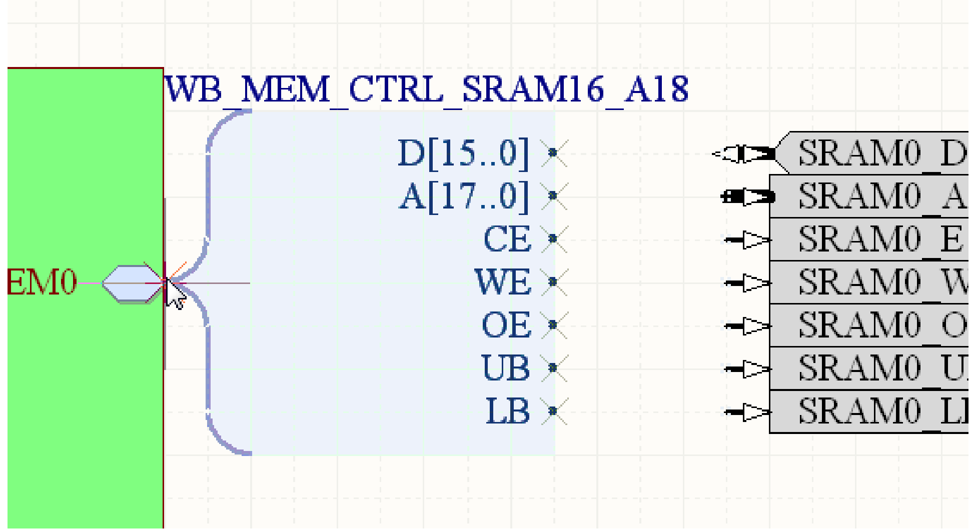

Figure 10. Flip the Signal Harness and place it so that it touches the Sheet Entry.

29. 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.

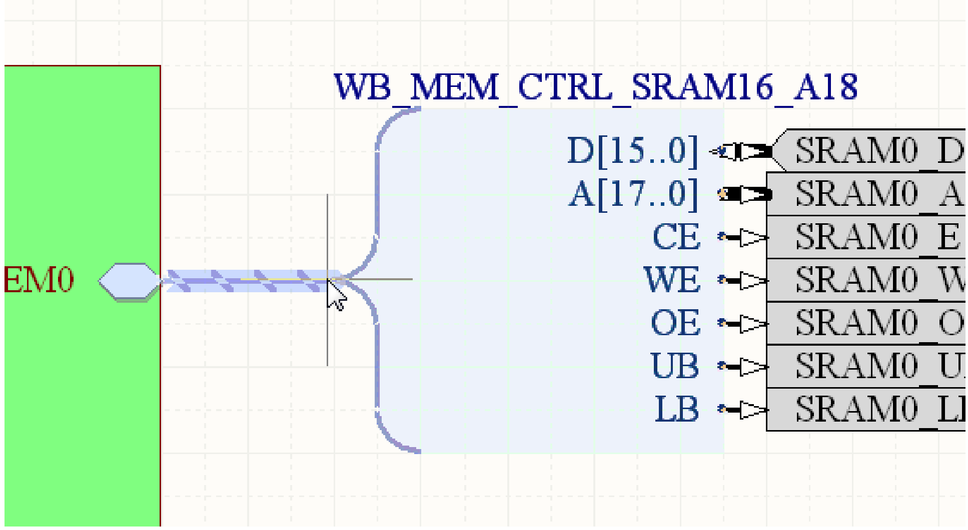

Figure 11. Drag the Harness Connector so that each entry touches a port on the memory port plug-in.

30. Repeat this process for the second memory Sheet Entry, **X_SRAM_MEM1**, connecting it to the second memory port plug-in.
31. Wire the LED_R[7..0] pin on the LEDs to their Sheet Entry, using a Bus line.
32. Wire the 2 unused pins on the LEDs to a Bus Ground Power Port, as shown in Figure 12.

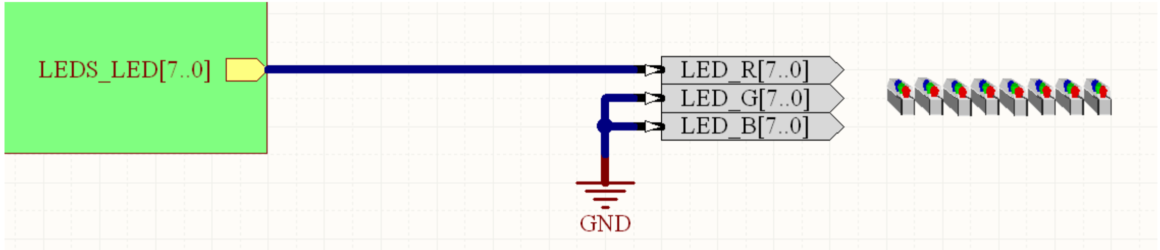

Figure 12. Wire the red pin of the LEDs, using a Bus line, and then wire the unused pins to ground, using a Bus Ground.

33. To annotate all of the components (assign designators), select **Tools»Annotate Schematics Quietly** from the menus.
34. 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, if there are none (the Messages panel is empty) then you have successfully captured and wired the FPGA hardware. Resolve any errors or warnings before continuing.

Tutorial steps – mapping the hardware design to the target hardware

35. The last stage of hardware capture is to create the connectivity from the ports on the top schematic sheet, through to the actual pins on the target FPGA. This mapping is done by constraint files, which detail port-to-pin mapping, along with other relevant design specifications, such as clock allocations 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.
36. An icon of the NB3000 will appear, right-click on it and select **Configure FPGA Project»uP_KnightRider.PrjFpg** from the menu, as shown in Figure 13.

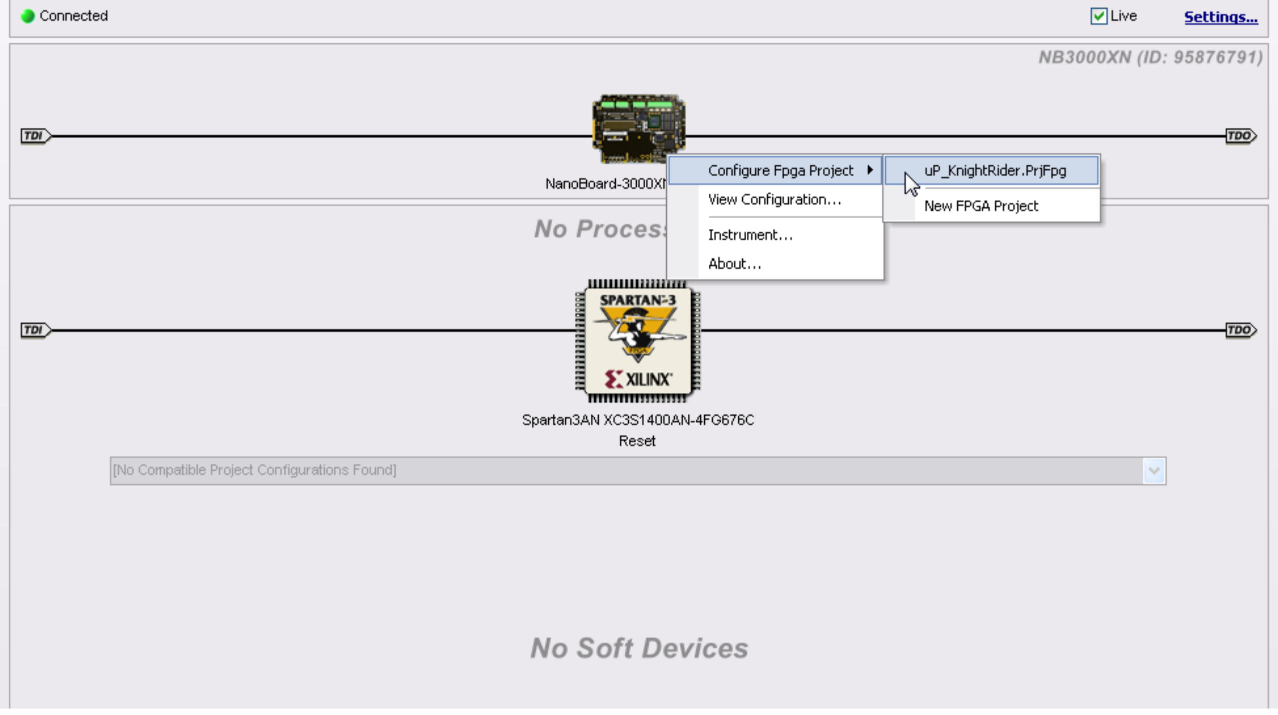

Figure 13. Configure the design to run on the NB3000.

37. 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.
38. 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

39. Create a new embedded project, and save it as `uP_KnightRider.PrjEmb` in a sub-folder below the FPGA project called `\Embedded`.
40. 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 14.

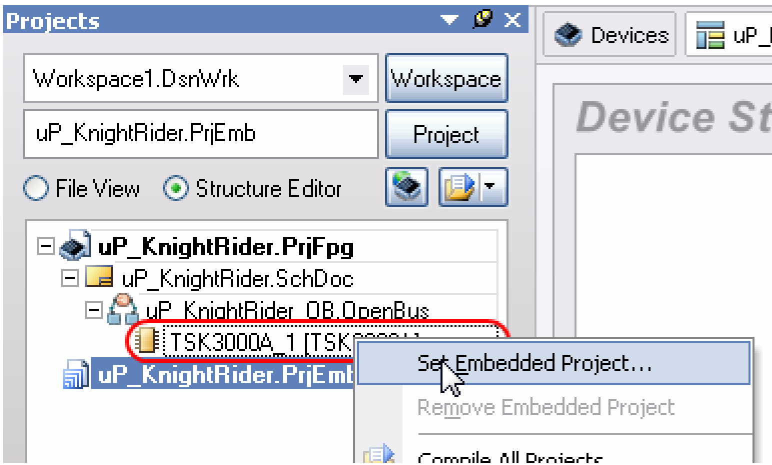

Figure 14. Make the embedded project a child of the FPGA project.

41. Right-click on the **uP_KnightRider.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. (Refer to Figure 15).
42. Double-click on the TSK3000A_1 row in the memory table (see Figure 15) and configure the memory as ROM instead of Non-Volatile RAM (see Figure 16). This forces code we will add later to boot from the TSK3000A internal memory automatically. Click OK twice to apply the changes.

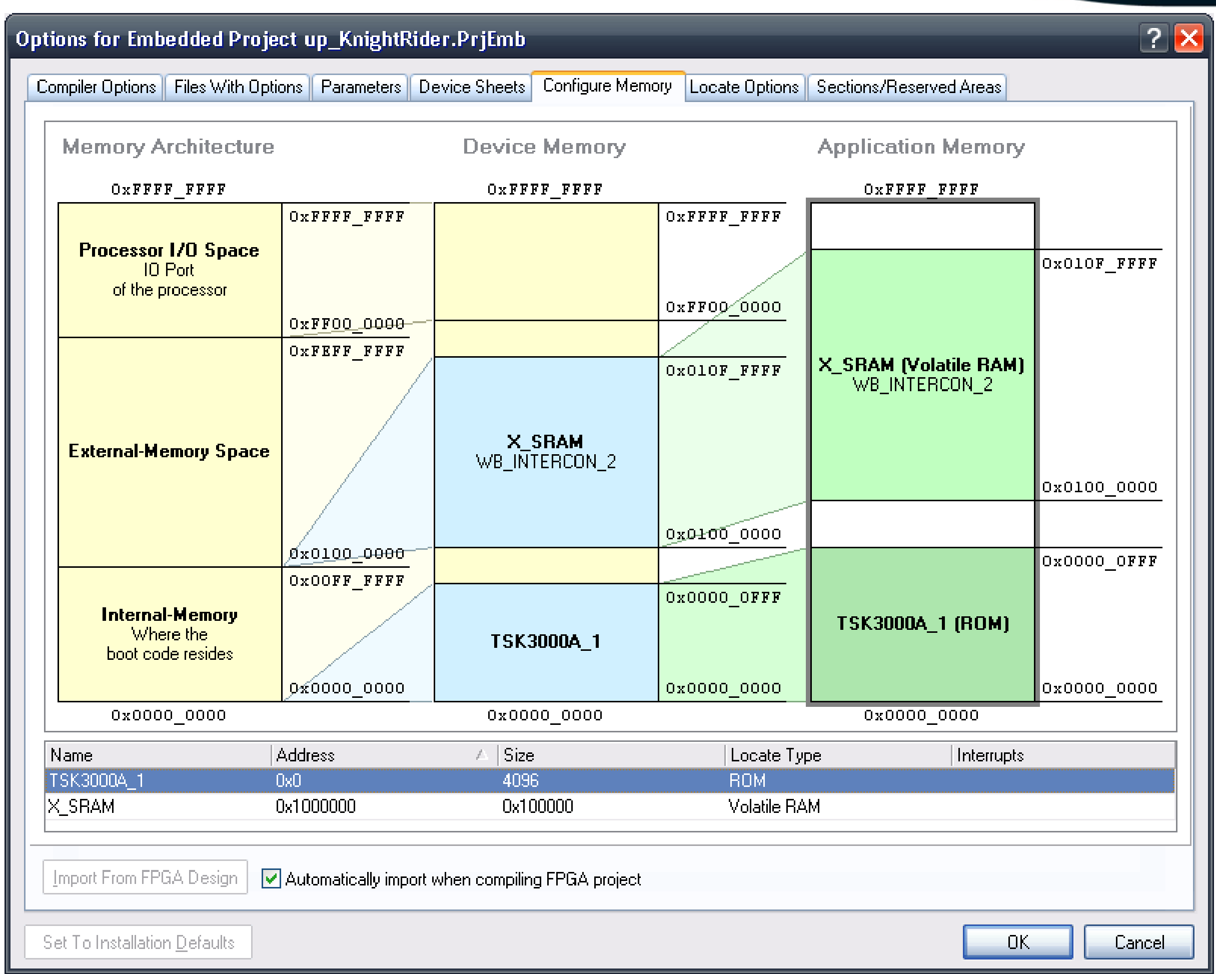

Figure 15. Embedded project memory map.

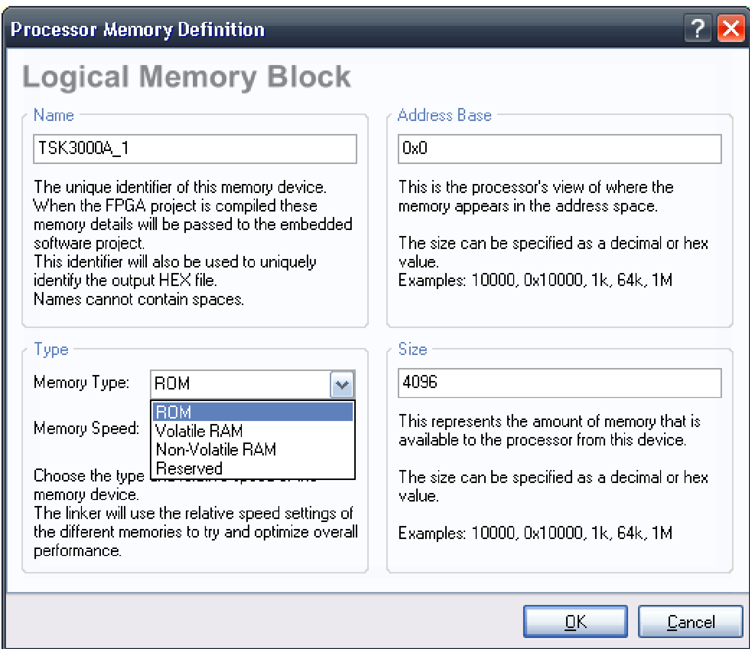

Figure 16. Setting the boot memory to Memory Type: ROM.

43. Switch the Projects panel back to **File View** mode.

44. Add a SwPlatform file to the embedded project, and save it as `uP_KnightRider.SwPlatform` in this embedded sub-folder.
45. 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.
46. The low-level wrapper for the LED Controller will be added, click once to select this wrapper and then click the **Grow Stack Up** button.
47. The *Grow Stack* dialog will open, showing the software stack that is available, **LED Controller Driver**. Click to select it, and click **OK** to close the dialog. The Device Stack should now appear as shown in Figure 17.

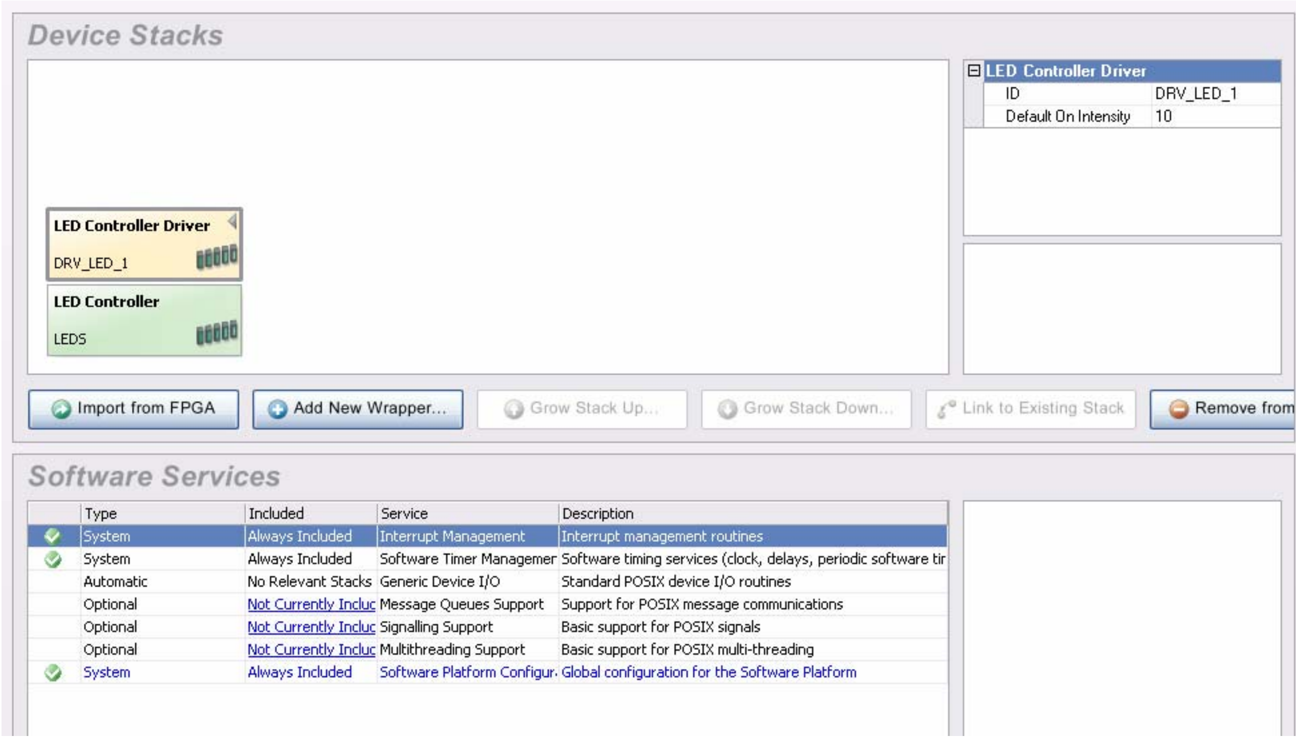

Figure 17. The I/O wrappers and low-level drivers are managed by the Software Platform.

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

Tutorial steps – writing the embedded code

To build our embedded code, we will need to include and use functions provided by the Software Platform Builder that we have just configured. Each low-level wrapper and driver in the Software Platform has a set of data types and functions pre-defined, the details of which are documented extensively via Altium Designer's Knowledge Center panel. At any time you can select a wrapper, driver or context software stack object and press **F1** to open the Knowledge Center panel at the page detailing that object. You can also select any driver functions within the code editor and press **F1** for help on that function.

49. In the file `uP_KnightRider.SwPlatform` select the LED Controller Driver as shown in Figure 17 and press **F1**. You should see the Knowledge Center panel open with the LED Controller Driver details shown as in Figure 18.

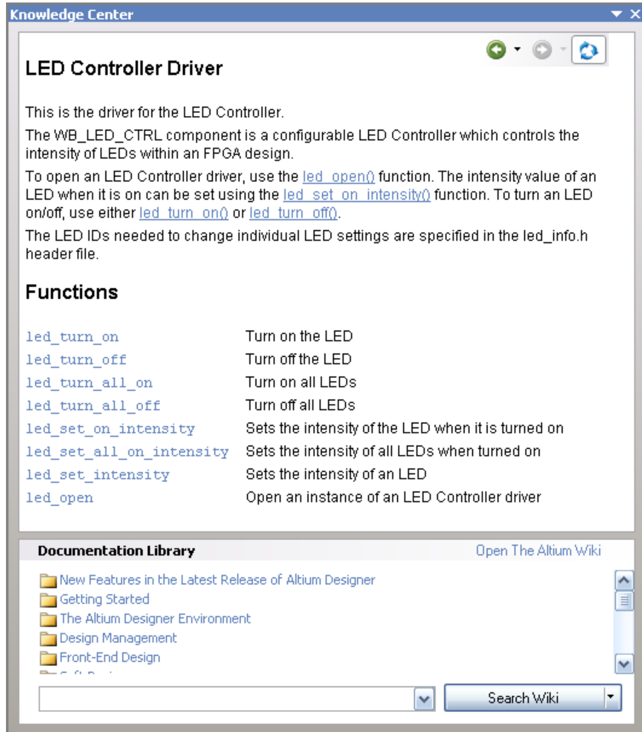

Figure 18. LED Controller Driver reference.

50. In the Knowledge Center panel, click on the link to the `led_open` function. Note that the Knowledge Center panel now displays the information for the function required to open an instance of the LED driver (or another way of putting it would be to *connect to the LED driver*).

51. Note that the syntax shows an `led_t*` data type for accessing the LED controller. This – like any other driver in the Software Platform Builder – is a predefined data type pointer that is used to address each instance of the associated hardware. The ID parameter is the number indicating which instance you wish to initialize a driver *for*. For example, if you have three LED controllers in your hardware design, their ID's would be 0, 1 and 2 respectively. You would call this function three times, each with the respective ID of the LED controller tied to it in hardware, assigning the returned pointer values to three `led_t*` pointers. These pointers are then used whenever calling other driver functions for writing brightness values to various LEDs. Click the LED Controller Driver link to return to the driver overview, and make a note of the functions provided to turn LEDs on and off, or set their intensity (PWM) value. Close the Knowledge Center panel.

52. Right-click on the embedded project **up_KnightRider.PrjEmb** and click **Add New to Project » C File**. Save the newly created C code document as `main.c` in your Embedded Project folder. Add the following code to the project, shown in Table 2.

```
// #include for LED driver:
#include <drv_led.h>
// #include for SwPlatform device names:
#include <devices.h>
// #include SwPlatform-generated LED hardware configuration info:
#include "led_info.h"
// led_t pointer for attaching to driver instance:
led_t * ptrLEDs;
// Another useful variable:
unsigned char brightness = 0;

void main(void)
{
    ptrLEDs = led_open(DRV_LED_1);                      // initialize driver
    led_set_intensity(ptrLEDs, LEDS_LED7, 0x80);        // set LED0 to 50%

    while (1)
    {
        // waste some time:
        for (int i = 0; i < 0xffff; i++){ __asm("NOP");}

        // Ramp up LED7 brightness:
        led_set_intensity(ptrLEDs, LEDS_LED0, brightness++);
    }
}
```

Table 2. Initial LED test code.

53. In the Projects panel, drill down into the **Generated»Header Documents** section and open `devices.h` and `led_info.h`. If they are not showing you will need to first re-compile your embedded project. These documents are automatically generated from the Software Platform Builder to make code more readable as well as provide a re-configurable abstraction of user code from low-level drivers. Note that the naming convention for these C macros is to use all upper-case characters. With syntax-highlighting it's easy to see where these are used in the code (again, refer to Table 2).
54. Switch to **Devices View** (**View»Devices View**) and build and download the project to the NB3000. You will see the LED at the far left on at half brightness, and the one at far right ramping up brightness continuously.
55. The initial test code uses a software loop to implement a timing delay but that consumes unnecessary processor cycles. A better implementation is to use a timer. Open the uP_KnightRider Software Platform document again and in the *Software Services* section, locate and select the **Software Timer Management** service. In the far right pane check the **Use Software Timers** box.
56. Delete the code in `main.c` and add the code in Table 3.
57. While in the code editor, click the **Compile and Download** button to re-compile and launch the new code. Note that you must have **Debug** selected instead of **Simulate**.

Figure 19. recompile and re-download, once you have switched to Debug mode.

```
#include <drv_led.h>
#include <timers.h>
#include <devices.h>
#include <stddef.h>
#include "led_info.h"
#define PRE_SCAN_VALUE  100
#define SCAN_ARRAY_SIZE 7

timer_handler_t TimerTick (void* context);
void UpdateKnightRiderLEDs (void);
void init(void);

led_t* ptrLEDs;
unsigned char ScanArray[LEDS_NUM_LED_IDS];
volatile unsigned char Tick = 0;

void main(void)
{
   init();
   led_turn_on(ptrLEDs, 0);
   while(1)
      {
         if (Tick)
            {
               UpdateKnightRiderLEDs();
               Tick = 0;  // Clear timer flag.
            }
      }
}

// Initialize LED driver and TSK3000A timer.
void init(void)
{
   ptrLEDs = led_open(LEDS);
   timer_register_handler(0, 20000L, TimerTick);
}

// Callback function for Timer Interrupt - sets flag.
timer_handler_t TimerTick (void* context)
{
   Tick = 1;        //indicate to mainline that the timer tick has occurred.
   return NULL;
}

// Function to shift LED brightness pattern.
void UpdateKnightRiderLEDs (void)
{
   static unsigned char ScanIndex = 0;
   static unsigned char LEDScanIndexModifier = 1;

   // Loop to set eight LEDs in linear brightness pattern
   for (unsigned char i = 0; i < LEDS_NUM_LED_IDS; i++)
   {
      if (ScanArray[i] == PRE_SCAN_VALUE)
         ScanArray[i] = 255;

      else
         ScanArray[i] = ScanArray[i] >> 1;
    }
   ScanArray[ScanIndex] = PRE_SCAN_VALUE;
   ScanIndex += LEDScanIndexModifier;
   if ((ScanIndex == (LEDS_NUM_LED_IDS-1)) || (ScanIndex == 0))
         LEDScanIndexModifier *= -1;
    for (unsigned char i = 0; i < LEDS_NUM_LED_IDS; i++)
    {
        led_set_intensity(ptrLEDs, i, ScanArray[i]);
    }
}
```

Table 3. Final Code for uP_KnightRider.

58. You will now see the KnightRider LED chaser pattern moving back and forth on the LEDs. If it's not working, go back over the code you have entered and ensure nothing is missing and no typographical errors exist.

Code Explanation

In this tutorial we are using the TSK3000A processor's internal timer. We can access the timers in any of the supported 32-bit processors by including the `timers.h` header file in our code. When our main program begins, the timer is initialized by the `init()` function which enables the timer and configures the interrupt service routine to call our function `TimerTick()` whenever a timer interrupt occurs (set to every 20000µS). The LED driver is also initialized.

Each time the timer ISR calls `TimerTick()` a flag is set by way of the variable `Tick`. This variable is declared as **`volatile`** since it is being modified by the timer interrupt routine – this tells the compiler not to optimize it as it could change at any point in time (that is, asynchronously from the main program loop). The main program loop checks this flag continually and when set it updates the LED display using `UpdateKnightRiderLEDs()`, and then clears the flag. This is a typical example of how you can synchronize two processes in an embedded system.

For some fun, experiment with the `timer_register_handler()` function to change the interrupt frequency and hence the speed of the LEDs updating.

Revision History

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

Software, hardware, documentation and related materials:

Copyright © 2009 Altium Limited.

All rights reserved. You are permitted to print this document provided that (1) the use of such is for personal use only and will not be copied or posted on any network computer or broadcast in any media, and (2) no modifications of the document is made. Unauthorized duplication, in whole or part, of this document by any means, mechanical or electronic, including translation into another language, except for brief excerpts in published reviews, is prohibited without the express written permission of Altium Limited. Unauthorized duplication of this work may also be prohibited by local statute. Violators may be subject to both criminal and civil penalties, including fines and/or imprisonment. Altium, Altium Designer, Board Insight, Design Explorer, DXP, LiveDesign, NanoBoard, NanoTalk, P-CAD, SimCode, Situs, TASKING, and Topological Autorouting and their respective logos are trademarks or registered trademarks of Altium Limited or its subsidiaries. All other registered or unregistered trademarks referenced herein are the property of their respective owners and no trademark rights to the same are claimed.