Application Note How to Implement Custom Serial Interfaces Using the Configurable Logic Block (CLB)



Gustavo Martinez

ABSTRACT

This application report describes how to design a custom serial interface using the configurable logic block (CLB). A step-by-step methodology for designing a CLB-based serial port is given along with a description of common design challenges and potential solutions. Finally, two serial port design examples are given with full design details, simulation results, and hardware test results. The first example covers the design of a CLB-based serial port to send and receive data over a time-division multiplexing (TDM) bus commonly used in audio systems. The second example uses a CLB-based serial port to send data to an LED matrix display. All example code is included in the C2000ware software development kit (SDK).

Table of Contents

| 1 Introduction | 3 |
|--|-------------------|
| 2 Serial Port Design Methodology | 3 |
| 2.1 Step 1: Understand Design Requirements | 3 |
| 2.2 Step 2: Identify Required Inputs to the CLB Tile | 3 |
| 2.3 Step 3: Identify Required Outputs from CLB Logic | 5 |
| 2.4 Step 4: Design the CLB Logic | 7 |
| 2.5 Step 5: Simulate the Logic Design | 10 |
| 2.6 Step 6: Test the CLB Logic | 11 |
| 3 Example A: Using the CLB to Input and Output a TDM Stream in Audio Applications | 11 |
| 3.1 Example Overview | 11 |
| 3.2 Step 1: Understand Design Requirements | 12 |
| 3.3 Step 2: Identify Required Inputs to the CLB Tile | 12 |
| 3.4 Step 3: Identify Required Outputs from CLB Logic | 13 |
| 3.5 Step 4: Design the CLB Logic | 14 |
| 3.6 Step 5: Simulate the Logic Design | 18 |
| 3.7 Step 6: Test the CLB Logic | 19 |
| 4 Example B: Using the CLB to Implement a Custom Communication Bus for LED Driver in Lighting Applications | s <mark>24</mark> |
| 4.1 Example Overview | 24 |
| 4.2 Step 1: Understand Design Requirements | 25 |
| 4.3 Step 2: Identify Required Inputs to the CLB Tile | 26 |
| 4.4 Step 3: Identify Required Outputs From CLB Logic | 27 |
| 4.5 Step 4: Design the CLB Logic | 28 |
| 4.6 Step 5: Simulate the Logic Design | 35 |
| 4.7 Step 6: Test the CLB Logic | 38 |
| 5 References | 41 |

List of Figures

| Figure 2-1. Using D-Type Flip Flops to Synchronize Outputs | 5 |
|---|----------------|
| Figure 2-2. Synchronizing Outputs Using FSM D-Type Flip-FLops | <mark>6</mark> |
| Figure 2-3. FSM Setup for D-Type Flip-Flop | <mark>6</mark> |
| Figure 2-4. Data Movement Flow | 8 |
| Figure 2-5. Example Interconnect Block Diagram | . 10 |
| Figure 2-6. CLB Simulation Example | 10 |
| Figure 3-1. CLB TDM-8 Example | 11 |
| Figure 3-2. Required TDM Stream Frame-Sync Pulse, Clock Signal, and Data Timing | 12 |

1



| Figure 3-3. CLB Tile Inputs for TDM-8 Example | 12 |
|---|-----------------|
| Figure 3-4. CLB Tile Outputs for TDM-8 Example | 13 |
| Figure 3-5. CLB Tile Logic for TDM-8 Example | 14 |
| Figure 3-6. Assumed TDM Stream Frame-Sync Pulse, Clock Signal, and Data Timing for CLB Logic Design | 14 |
| Figure 3-7. FSM1 State Diagram for TDM-8 Example | 15 |
| Figure 3-8. FSM0 State Machine for TDM-8 Example | 16 |
| Figure 3-9. TDM-8 Data Receive Simulation | 18 |
| Figure 3-10. TDM-8 Data Transmit Simulation | |
| Figure 3-11. FSYNC and DATA1 Synchronization | 19 |
| Figure 3-12. Test Setup for TDM Example | 19 |
| Figure 3-13. Importing CLB TDM Example Project | <mark>21</mark> |
| Figure 3-14. Importing McBSP TDM Example Project | 22 |
| Figure 3-15. TDM-8 Output Without Latch and Delay Logic | 23 |
| Figure 3-16. Final TDM-8 Output. | 23 |
| Figure 3-17. F28388D Monitoring Incoming TDM Traffic | 24 |
| Figure 4-1. Block Diagram for LED Matrix Example | 24 |
| Figure 4-2. CCSI Frame | 25 |
| Figure 4-3. TX Tile Input Signals | <mark>26</mark> |
| Figure 4-4. RX Tile Input Signals | 27 |
| Figure 4-5. TX Tile Output Signals | 27 |
| Figure 4-6. CLB TX Tile Logic | 28 |
| Figure 4-7. FSM0 State Diagram for TX Tile | <mark>29</mark> |
| Figure 4-8. FSM2 State Diagram for TX Tile | 30 |
| Figure 4-9. CLB RX Tile Logic | <mark>31</mark> |
| Figure 4-10. FSM0 State Diagram for RX Tile | 32 |
| Figure 4-11. FSM1 State Diagram for RX Tile | 32 |
| Figure 4-12. Single-Clock Edge Data Transmission and Reception Clock Configuration | 33 |
| Figure 4-13. Dual-Clock Edge Data Transmit/Receive Using PWM_SCLKX2 | |
| Figure 4-14. Dual-Clock Edge Data Transmission and Reception Clock Configuration | <mark>34</mark> |
| Figure 4-15. Data Receive Simulation for LED Driver | <mark>35</mark> |
| Figure 4-16. Check Bit Error Logic Simulation | 36 |
| Figure 4-17. END Frame Detection | <mark>36</mark> |
| Figure 4-18. Data Transmit Simulation for LED Driver | 37 |
| Figure 4-19. Output Glitch | 38 |
| Figure 4-20. Importing CLB LED Driver Example Project | 39 |
| Figure 4-21. LED Driver Example Setup and Hold Time | 40 |

List of Tables

| Table 2-1. Recommended Paths for Input Signals | |
|--|----|
| Table 2-2. GPIO Input Qualification Uses. | 4 |
| Table 2-3. CLB Input Configuration Options | 5 |
| Table 2-4. CLB Input Filter Options. | 5 |
| Table 3-1. FSM1 Truth Table for TDM-8 Example | |
| Table 3-2. FSM0 Truth Table for TDM-8 Example | 17 |
| Table 3-3. FSM2 Truth Table for TDM-8 Example | 17 |
| Table 3-4. F280025C CLB Logic Input TDM Signals | |
| Table 3-5. F28388D McBSP Output TDM Signals | 20 |
| Table 3-6. F280025C CLB Logic Output TDM Signals | |
| Table 3-7. F28388D McBSP Input TDM Signals. | 20 |
| Table 4-1. Design Requirements | |
| Table 4-2. FSMO Truth Table | 29 |
| Table 4-3. FSM2 Truth Table | 30 |
| Table 4-4. FSM0 Truth Table | 32 |
| Table 4-5. FSM1 Truth Table | 33 |
| | |

Trademarks

C2000[™] and Code Composer Studio[™] are trademarks of Texas Instruments. All trademarks are the property of their respective owners.



1 Introduction

Serial interfaces are commonly used in many industrial and automotive applications. Standard serial interfaces available in today's microcontrollers can support access to a multitude of peripheral devices. However, in some cases, a custom serial interface may be required to support a unique serial protocol. The configurable logic block (CLB) included in C2000[™] real-time microcontrollers can be programmed to emulate these custom serial interfaces.

In this application note, a step-by-step design methodology is presented to guide the designer in the design, simulation, and testing of a custom serial interface. Some common design challenges and potential solutions are presented throughout the application report. Finally, two examples are presented with full design details, simulation results, and hardware test results. The example code can be easily obtained from the C2000ware software development kit (SDK).

2 Serial Port Design Methodology

The following is a step-by-step guide to designing a custom serial interface using the CLB. The steps presented here walk the designer from start to finish during the CLB logic design. Solutions are presented to common design challenges and pitfalls.

This application report assumes the reader is already familiar with the architecture of the CLB and with Code Composer Studio (CCS) integrated development environment (IDE). Below is a list of training materials that can be used for a quick overview of the CLB and C2000 real-time microcontrollers:

- Configurable Logic Block (CLB) Introduction (video)
- Configurable Logic Block (CLB) architecture (video)
- Configurable Logic Block (CLB) programming tool training (video)
- SysConfig Development Tool for C2000 real-time microcontrollers
- C2000 Academy Online Training

It is important to keep in mind that the CLB type, number of CLB tiles, and crossbar (XBAR) logic differs between C2000 real-time microcontrollers. For detailed information on the CLB relating to a specific C2000 real-time microcontroller, see the device-specific TRM.

2.1 Step 1: Understand Design Requirements

The key to *any* successful CLB logic design is having a clear understanding of the design requirements. In the case of a serial interface design, a thorough understanding the bus protocol will be required during the CLB logic design phase. This includes basic information such as number of signals, input and output configuration, and frame encoding requirements such as start, stop, parity, and checksum bits.

Second, the timing requirements of the bus interface must be clearly understood to determine the feasibility of implementing the design using the CLB. Special attention should be given to the serial bus operating speed, signal polarities, and required setup and hold times.

Lastly, any support signals that will be required to implement the serial bus interface must be identified. This can include, for example, an input clock signal generated by an on-chip PWM to clock data in and out of the CLB tile, or a periodic timer interrupt to trigger the transmission of a synchronization command on the serial bus.

2.2 Step 2: Identify Required Inputs to the CLB Tile

In most cases, input signals need to be passed to the CLB tile. Generally, there are three categories of signals to consider:

- External serial bus signals sampled through GPIO pins. These include, for example, serial bus clock, frame, and data signals.
- On-chip peripheral signals used in the operation of the CLB logic. For example, a clock signal generated using a PWM*n*A pin or a timer interrupt.
- GPREG bits directly controlled by the CPU. These GPREG bits can be used to trigger an action in the CLB tile or enable/disable a specific feature.

These input signals can be connected to the eight CLB tile inputs using the CLB global and local muxes in combination with the different XBARs on the device. For more information, see the device-specific TRM.

3



There are often multiple paths which can be used to connect a CLB tile to an input signal. Table 2-1 lists the recommended path to bring in these signals to the CLB tile boundary.

| Table 2-1. Recommended Faths for input orginals | | | | | |
|--|---|--|--|--|--|
| Input Signal Type | Recommended Input Path | | | | |
| External serial bus signals (for example, serial bus clock, frame, and data signals) | GPIO pin \Rightarrow CLB input XBAR 1 \Rightarrow CLB local mux \Rightarrow CLB input | | | | |
| Internal on-chip peripheral signals (for example, clock signal generated using a PWM <i>n</i> A) | Peripheral signal \Rightarrow CLB global mux or CLB local mux \Rightarrow CLB input | | | | |
| Custom signal directly controlled by CPU | Memory-mapped GPREG bit ⇔ CLB input | | | | |

Table 2-1. Recommended Paths for Input Signals

1. CLB input XBAR is not available on all C2000 real-time microcontrollers. On those devices, external signals can be brought to the tile boundary using the GPIO XBAR and CLB XBAR. For more information, see the device-specific TRM.

2.2.1 GPIO Input Qualification

One important consideration when sampling external serial bus signals is the input qualification settings of the GPIO pin. Table 2-2 summarizes the different input qualification settings available and potential uses.

| Input Qualification Setting | Description | Uses | |
|--|--|---|--|
| Asynchronous input (recommended setting) | The input signal is not synchronized to SYSCLKOUT. Note: Synchronization can be enabled at CLB input level (see Section 2.2.2). | Used for passing signals unchanged to the CLB tile. | |
| Synchronization to SYSCLKOUT (not recommended) | The input signal is synchronized to SYSCLKOUT. | A small analog delay is added by the input XBARs before the signal reaches the CLB tile boundary, essentially making the signal asynchronous to SYSCLKOUT again. Therefore, synchronization at this level only adds unnecessary delay. It is recommended to enable synchronization at the CLB input level instead (see Section 2.2.2). | |
| Qualification Using a Sampling Window (recommended in some cases) | The input signal is synchronized to SYSCLKOUT and qualified for a specified number of cycles before the input is allowed to change. Use for removing noise from the However, the added latency intro the sampling window needs to be when designing CLB logic. Synchronization at the CLB input also be enabled to re-synchroniz to SYSCLKOUT since the input 2 add a small analog delay before reaches the CLB tile boundary. | | |

Table 2-2. GPIO Input Qualification Uses

Additional features to consider:

- GPIO internal pull-up resistor. Enable this feature to avoid floating CLB input signals, if no external pull-up exists on the board.
- GPIO input signal inversion: Enable this feature to invert the signal at the GPIO pin before passing to the CLB.



2.2.2 CLB Input Settings

Each of the eight CLB inputs have different configuration and filter options which must be carefully considered depending on the switching characteristics of the serial bus. Table 2-3 and Table 2-4 list the different options and their possible uses.

| Input Configuration Setting | Description | Uses |
|--|--|---|
| Asynchronous input | The input signal is not synchronized to SYSCLKOUT. | Use for passing signals unchanged through the CLB tile and directly to the CLB outputs. |
| Synchronization to SYSCLKOUT (recommended setting) | The input signal is synchronized to SYSCLKOUT. | Generally, it is required to synchronize all CLB inputs used in tile logic blocks. |

Table 2-3. CLB Input Configuration Options

| Input Filter Setting | Description | Uses | |
|----------------------|---|--|--|
| No filtering | The input signal is passed directly to the CLB tile. | Use for designing logic that depends on the logical state of the input. For example, enable & disable signals. | |
| Rising-edge detect | A single pulse equal to a CLB clock will be generated when a rising-edge is detected on the input signal. | Use for shifting data in or out of the counters in serializer mode or for counting serial clock edges. | |
| Falling-edge detect | A single pulse equal to a CLB clock will be generated when a falling-edge is detected on the input signal. | Use for shifting data in or out of the counters in serializer mode or for counting serial clock edges. | |
| Any-edge detect | A single pulse equal to a CLB clock will be generated when a falling-edge or rising-edge is detected on the input signal. | Use for logic that needs to act on both edges of an input signal. | |

Table 2-4. CLB Input Filter Options

A single external input may need to be mapped to multiple CLB tile inputs with different input filter settings. This allows part of the CLB tile logic to operate off the rising edge of an external signal and separate logic to operate off the falling edge of the same signal. Mapping of a single external signal to multiple CLB inputs is used in the two examples given in this application report.

2.3 Step 3: Identify Required Outputs from CLB Logic

In most cases, the CLB tile needs to drive serial bus signals. These can include, for example, serial bus clock, frame, and data signals. Each CLB tile has eight outputs that can be brought out to the device pins through the CLB output XBAR or the GPIO output XBAR.

2.3.1 Synchronizing Outputs Signals

In all likelihood, the output signals generated by the CLB is generated by different logic blocks in the CLB tile. This can lead to different delay paths in each output signal. In order to maximize the setup and hold times seen by the receiving device, the different CLB outputs can be synchronized to each other using a bus clock and simple flip-flops, see Figure 2-1. The flip-flops can be implemented using finite state machine (FSM) blocks in the CLB tile. However, due to limited CLB tile resources, it is best to use this option only when absolutely needed.





5



As an example of the need for synchronization logic, consider the simulation in Figure 2-2, which shows two inputs, *in1* and *in2*, changing state at slightly different times. These two signals are synchronized to each other using a third "clock" signal, *in0* using two edge-triggered D-type flip-flops implemented using the two FSMs. As can be seen from the simulation, each FSM latches and delays its input signal using the "clock" signal *in0*. As a result, both FSM outputs are synchronized with each other.



Figure 2-2. Synchronizing Outputs Using FSM D-Type Flip-FLops

The logic equations for each FSM output and state variable are shown in Figure 2-3. Notice that both FSM *out* and *s0* are set to the same equation and either signal can be used to drive the final output. There is a 1-cycle delay between both signals due to *out* being a purely combinatorial output, while *s0* is always updated on the next CLB clock cycle. Also note that in this simulation, *in0* input filter is setup for synchronous, rising-edge detect.

| FSM_0 | | ~ |
|--|---|---|
| eqn_out | !e0&s0 e1&e0 | |
| eqn_s0 | !e0&s0 e1&e0 | |
| eqn_s1 | | |
| e0 | BOUNDARY.in0 | Ŧ |
| e1 | BOUNDARY.in1 | Ŧ |
| xe0 | 0 | * |
| xe1 | 0 | * |
| FSM_1 | | ~ |
| 1 | | |
| eqn_out | !e0&s0 e1&e0 | |
| eqn_out eqn_s0 | <u>!e0&s0 e1&e0</u> !e0&s0 e1&e0 | |
| eqn_out eqn_s0 eqn_s1 | <u>!e0&s0 e1&e0 </u> !e0&s0 e1&e0 | |
| eqn_out eqn_s0 eqn_s1 e0 | <u>!e0&s0 e1&e0</u> !e0&s0 e1&e0 BOUNDARY.in0 | • |
| eqn_out eqn_s0 eqn_s1 e0 e1 | <u>le0&s0 e1&e0</u> <u>le0&s0 e1&e0</u> BOUNDARY.in0 BOUNDARY.in2 | * |
| eqn_out eqn_s0 eqn_s1 e0 e1 xe0 | <u>le0&s0 e1&e0</u> <u>le0&s0 e1&e0</u> BOUNDARY.in0 BOUNDARY.in2 0 | * |

Figure 2-3. FSM Setup for D-Type Flip-Flop



2.3.2 Output Signal Conditioning

Each CLB output can be passed through an asynchronous output conditioning (AOC) block where it can be inverted and gated before being passed to the output XBARs. The AOC could be used to invert the signal before passing it to the output.

Note

If a CLB input signal, say a serial clock, needs to be passed through the CLB tile unsynchronized to the internal CLB clock, the input signal must be passed through the CLB tile using boundary input 4 or 5. From the tile input the signal must be passed directly to the AOC where it can be inverted, if desired. Additionally, the AOC output must be brought to the micronctroller pin through the GPIO OUTPUT XBAR, not the CLB OUTPUT XBAR.

2.4 Step 4: Design the CLB Logic

Designing the CLB logic is the most complicated step in the process. Several iterations of logic design, simulation, and testing will be required to finalize the design.

The simplest approach to program the CLB tiles is to leverage the CLB tool. With the CLB tool the user can select the input filter options for each CLB tile input, connect sub-modules in each CLB block, program the FSM blocks, high-level controller (HLC), and other logic blocks, and easily replicate tile logic to other tiles. The tool also allows the user to setup simulations for the tile logic. The CLB tool will flag, in real-time, any potential issues in the logic design such as unsupported logic connections and invalid FSM formulas.

The CLB tool makes use of the "SysConfig" graphical user interface (GUI), which is part of Code Composer Studio[™] (CCS). With SysConfig the user can also configure the MCU's input and output XBARs as well as program the MCU peripherals.

For more information on the CLB tool and SysConfig, refer to the following resources:

- CLB Tool User's Guide
- Configurable Logic Block (CLB) programming tool training (video)
- SysConfig Development Tool for C2000 real-time MCUs

2.4.1 Resource Allocation

Usage of the CLB logic blocks needs to be carefully considered as each CLB tile has finite number of resources. In general, for a serial interface the following will be required:

- One counter is used for data receive. The counter operates in serializer mode to allow the CLB to shift in a serial bit on each serial clock edge.
- One counter is used for data transmit. The counter operates in serializer mode to allow the CLB to shift out a serial bit on each serial clock edge.
- One counter is used for serial bit counting. The counter operates in normal counter mode.
- One FSM block is needed to count receive (and/or transmit) words and trigger events to the HLC.

Although it is possible to implement a serial interface entirely in one CLB tile, in some cases it may be required to use two CLB tiles, in which case it is recommended to implement the receive and transmit functions in separate tiles for easier logic design.

Some serial interfaces require special frame encoding bits such as start, stop, and parity bits. The CLB can support these features, however, it requires additional CLB logic blocks. If the required CLB blocks are not available, consider leveraging the C28x CPU to extract or add these bits in the receive and transmit data.

2.4.2 Exchanging Data Between CLB FIFOs and MCU RAM

A CLB-based serial interface design requires data movement between 1. the CLB serializers and the CLB FIFOs, and 2. the CLB FIFOs and MCU RAM. The HLC in the CLB tile can move data between the serializers and the FIFOs. However, the HLC cannot move data between the FIFOs and the MCU RAM directly since is not a master in the MCU architecture. As of the writing of this application note, the C28x CPU and CLA are the only masters capable of moving data in and out of the CLB FIFOs, such that, the DMA cannot be used for this data movement.

7





Note

The HLC cannot pull data directly into the CLB counter. Instead, it must use an intermediate step of pulling data into one of the HLC registers (Rn), and then move the data to the counter.

It is possible for the CPU to directly load the counter and HLC registers using the CLB local write interface bus. This is useful to set the initial state of the CLB tile. For example, in Section 4, using the CLB to Implement a Custom Communication Bus for LED Driver in Lighting Applications, a counter is loaded with the total number of transmit words before the CLB TX tile is enabled for transmit. The following code shows an example of how to do this using C2000ware driverlib functions. For information on the CLB local write interface bus, see the device-specific TRM.

CLB_writeInterface(TX_TILE_BASE, CLB_ADDR_COUNTER_1_LOAD, 0xFFFFFFF); CLB_writeInterface(TX_TILE_BASE, CLB_ADDR_HLC_R2, 0x0);

2.4.3 CLB Logic Status and Trigger Flags

When designing a serial interface using CLB, there is often a need to implement status/flag bits and trigger bits.

2.4.3.1 Status/Flag Bits

Status/flag bits allow the CPU to poll the state of the CLB logic. For example, after a CLB interrupt, the CPU can poll a flag bit to determine if the interrupt was due a data receive interrupt or a receive error interrupt.

These status/flag bits can be implemented using the HLC registers. However, one HLC register has to be used per flag since there is no bitwise AND or OR instruction supported by the HLC. For example, a non-zero value in HLC R0 register could be used to indicate a receive interrupt, while a non-zero value in R1 could be used to indicate a receive error.

| Note | |
|---|---------|
| The HLC "INTR <tag>" instruction and CLB_INTR_TAG_REG cannot be used for flag</tag> | s since |
| subsequent uses of this instruction will overwrite any previous tag values. | |

The CLB logic status can also be determined by directly reading the status of the different CLB logic blocks. For example, if an FSM is configured to define multiple states, e.g. IDLE and ACTIVE, the CPU can read the status of S0 and S1 to determine the CLB logic state.

The CPU can use the CLB memory-mapped debug registers CLB_DBG_R*n*, CLB_DBG_C*n*, and CLB_DBG_OUT to determine the status of different blocks within the CLB. An example of using HLC registers as status bits is shown in the following code block. The code block uses C2000ware driverlib functions.

```
uint32_t chkIntFlag = CLB_getRegister(CCSI2_RX_TILE_BASE, CLB_REG_HLC_R1);
uint32_t rcvIntFlag = CLB_getRegister(CCSI2_RX_TILE_BASE, CLB_REG_HLC_R2);
uint32_t endIntFlag = CLB_getRegister(CCSI2_RX_TILE_BASE, CLB_REG_HLC_R3);
...
// Receive interrupt
if (rcvIntFlag & 0x1)
{
...
// Clear the receive interrupt flag register
CLB_writeInterface(CCSI2_RX_TILE_BASE, CLB_ADDR_HLC_R2, 0x0);
}
```



```
// End interrupt
if (endIntFlag & 0x1)
{
    ...
    // Clear the end interrupt flag register
    CLB_writeInterface(CCSI2_RX_TILE_BASE, CLB_ADDR_HLC_R3, 0x0);
}
// Check error interrupt
if (chkIntFlag & 0x1)
{
    ...
    // Clear the check error interrupt flag register
    CLB_writeInterface(CCSI2_RX_TILE_BASE, CLB_ADDR_HLC_R1, 0x0);
}
```

2.4.3.2 Trigger Bits

Trigger bits allow the CPU enable or disable logic in the tile or to trigger a specific action. For example, a receiver enable bit can be implemented to selectively enable or disable receive logic in the CLB at run time. These trigger bits can be implemented using the CLB_GP_REG register.

The CLB_GP_REG bits can be directly connected to the eight CLB tile inputs. The CPU can set any bit in the CLB_GP_REG to trigger an action in the CLB logic. For example, a receiver enable/disable bit can be implemented to enable or disable receive logic in the CLB at run-time. An example of an enable/disable bit implementation is shown in the following code block. The code block uses C2000ware driverlib.

```
#define GPREG ENABLE RCVR
                                  ЗU
void CCSI HAL enableClbReceiver()
   uint32 t gpRegVal = CLB getGPREG(CCSI1 RX TILE BASE);
   //\ensuremath{\mathsf{First}} check that receiver is in IDLE state
   while(HWREG(CCSI1_RX_TILE_BASE + CLB_LOGICCTL + CLB_O_DBG_OUT) &
           (CLB DBG OUT FSM0 S1 | CLB DBG OUT FSM0 S0)) {}
   // Enable receiver
   gpRegVal |= (1U << GPREG ENABLE RCVR);</pre>
   CLB setGPREG(CCSI1 RX TILE BASE, gpRegVal);
void CCSI HAL disableClbReceiver()
   uint32 t gpRegVal = CLB getGPREG(CCSI1 RX TILE BASE);
    // First check that receiver is in IDLE state
   // Disable receiver
   gpRegVal &= ~(1U << GPREG ENABLE RCVR);
   CLB setGPREG(CCSI1 RX TILE BASE, gpRegVal);
```

Note

A CLB input must be dedicated to any CLB_GP_REG bit used in the logic design. Given that the number of CLB inputs is limited to eight, CLB GP_REG bits should be used judiciously.

9

2.5 Step 5: Simulate the Logic Design

A serial interface design can quickly become very complex due to the different logic blocks and interconnections used in the design. In order to facilitate debugging of the CLB logic design, the CLB tool generates an interconnect block diagram (Figure 2-5) and a simulation waveform (Figure 2-6). For more details on how to generate these two files, see the *CLB Tool User's Guide*.



Figure 2-5. Example Interconnect Block Diagram



Figure 2-6. CLB Simulation Example

2.6 Step 6: Test the CLB Logic

The final step in the design process is to test the full design by running it on the intended system.

It is recommended to start with a controlled test setup which will allow for a thorough characterization of the logic design before moving to the target application. The test setup should include aim for complete test coverage of the CLB logic features. This can include varying data patterns and clock frequencies, as well as mechanisms to introduce deliberate error conditions, especially if the CLB logic was designed to detect errors.

The CLB simulator can be used debug any observed test failures by adjusting the simulator stimulus and viewing the signals within the CLB logic blocks. After modifying the CLB logic, compile and run the test again.

CAUTION

Keep in mind that performance on a single system, operating at room temperature, does not guarantee performance over all the operating conditions targeted for the system. It is important to identify all critical design parameters and ensure sufficient test coverage and timing margin is in place to guarantee the logic design can meet those parameters with sufficient margin.

3 Example A: Using the CLB to Input and Output a TDM Stream in Audio Applications

3.1 Example Overview

Time-division multiplexing is a common method used in audio applications to transmit and receive audio data between devices in the system. In this multiplexing scheme a frame is divided into several time slots, or channels, of fixed length. Since the multiple channels separated in time are multiplexed into a single transmission channel, the multiplexing scheme is called time-division multiplexing (TDM).

The multichannel buffered serial port (McBSP) available on some C2000 real-time microcontrollers supports input and output of a single TDM stream. However, the McbSP peripheral is not available on all C2000 real-time microcontrollers. Furthermore, none of the other serial port peripherals commonly included in C2000 real-time microcontrollers support the TDM protocol.

In this example, the CLB is used to input an 8-channel TDM stream (TDM-8), and output a corresponding TDM-8 stream. The C28x CPU moves received data from the CLB FIFOs to internal RAM, and moves transmit data from RAM to the CLB FIFOs. A single CLB tile is used to implement this example. The input and output TDM-8 streams are clocked using an external 12.288 MHz clock.



Figure 3-1. CLB TDM-8 Example



3.2 Step 1: Understand Design Requirements

The CLB tile logic is required to input and output an 8-channel TDM stream with the clock, frame-sync, and data timing shown in Figure 3-2.



Figure 3-2. Required TDM Stream Frame-Sync Pulse, Clock Signal, and Data Timing

Specifically, the following conditions are required:

- External, fixed 12.288MHz serial bit clock (bit clock passed through to output)
- 1 clock frame-sync pulse width with no delay
- · Frame-sync pulse is active high
- Transmit/receive data sampled on the rising edge of clock
- 8 channels per TDM frame
- 32-bit word length per channel

It is easy to support other frame-sync and clock polarities by simply using the GPIO inverting logic to invert the signal before it reaches the CLB tile.

3.3 Step 2: Identify Required Inputs to the CLB Tile

The input TDM-8 stream signals are routed to the CLB tile boundary inputs via the CLB input XBAR as shown in Figure 3-3.



Figure 3-3. CLB Tile Inputs for TDM-8 Example



All GPIOs use for the input TDM stream are configured for asynchronous operation. Synchronization is enabled at the CLB tile input boundary as needed. Furthermore, the internal pull-up is enabled on the BCLK_IN and FSYNC_IN GPIOs to avoid floating input pins.

Note

The CLB logic is designed for opposite clock polarity than that shown in Figure 3-2, hence the signal is BCLK_IN inverted. For more information, see Section 3.5.

The BCLK_IN signal is routed to three different tile inputs with different filtering configurations.

- CLB Input 0: This input is used to trigger data reception on the falling edge of BCLK_IN.
- CLB Input 3: This input is used to trigger data transmission on the rising edge of BCLK_IN.
- CLB Input 5: Filtering and synchronization are disabled on this input to support the BCLK pass-through requirement.

Special consideration must be given to the requirement to pass through the BCLK_IN signal to the BCLK_OUT output. Synchronizing the input 12.288 MHz BCLK signal to the 100 MHz CLB internal clock will introduce jitter in the resulting output BCLK signal since the two clocks are not multiples of each other.

3.4 Step 3: Identify Required Outputs from CLB Logic

The output TDM-8 stream signals are connected to the CLB tile boundary outputs using the CLB OUTPUT XBAR and the GPIO OUTPUT XBAR as shown in Figure 3-4.



Figure 3-4. CLB Tile Outputs for TDM-8 Example

As previously mentioned, the input BCLK signal must be passed through the CLB tile unregistered (i.e. it is not synchronized to any internal clock) to avoid introducing jitter at the output BCLK signal. Only CLB outputs 4 and 5 can be passed unregistered through the CLB tile. Therefore, CLB output 5 is used for BCLK_OUT. Furthermore, the BCLK signal is brought to the GPIO pin through the GPIO OUTPUT XBAR.

3.5 Step 4: Design the CLB Logic

The final CLB tile logic design is shown in Figure 3-5.



Figure 3-5. CLB Tile Logic for TDM-8 Example

The TDM clock, frame-sync, and data timing shown in Figure 3-6 is used in order to keep the CLB logic design simpler. The invert feature of the MCU GPIO pin can be used to support opposite clock and frame-sync polarities.





3.5.1 Resource Allocation

The CLB tile resources are allocated as follows:

- COUNTER0 is used to receive the input data. It is operated in serializer mode. DATA_IN is shifted in on BCLK_IN falling edges.
- COUNTER1 is used to output the transmit data. It is operated in serializer mode. COUNTER1 is shifted on BCLK_IN rising edges.
- COUNTER2 is used to count falling edges on BCLK_IN. It is operated in normal counter mode. It resets when an FSYNC + BCLK (low edge) condition is detected OR when the bit count reaches 32.
- LUT1 is used to reset COUNTER2 on FSYNC_IN and word boundaries. FSYNC_IN is logically ANDed with BCLK_IN falling edges to ensure FSYNC_IN is sampled only when it is guaranteed to be valid. LUT1 also triggers the HLC to pull new data from the PULL FIFO.
- FSM1 is used to generate an event to the HLC once 4 32-bit words have been received/transmitted. It resets its count when FSYNC_IN is detected.
- LUT0 and FSM0 are used to generate the FSYNC_OUT signal when FSYNC_IN is detected. The FSYNC_OUT signal is aligned to the BCLK_IN rising edge. Note that FSYNC_OUT is not a pass-through signal of FSYNC_IN as this was not a design requirement.
- FSM2 is used to align the output the DATA1_OUT signal to the BCLK_IN rising edge.
- The HLC is used to move data between the CLB push/pull FIFOs and the two serializer counters (COUNTER0 and COUNTER1). It also generates an interrupt to the CPU when 4 32-bit words are received/ transmitted.

3.5.2 TDM Word Counter

FSM1 is used to trigger an HLC event once four 32-bit words have been received/transmitted. The FSM1 state diagram and truth table are shown in Figure 3-7 and Table 3-1.



Figure 3-7. FSM1 State Diagram for TDM-8 Example



| Table 3-1. FSM1 Truth Table for TDM-8 Example | | | | | | |
|---|----|---------------|-------------------|---------|---------|---------------------|
| S1 | SO | E1 (FSYNC) | E0 (C2_MATCH1) | S1 Next | S0 Next | Output (HLC_INT) |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 |

The reduced logic equations are:

- S1 (NEXT) = S1&!S0&!E1 | S1&!E1&!E0 | !S1&S0&!E1&E0
- S0 (NEXT) = !S0&!E1&E0 | S0&!E1&!E0
- OUTPUT = S1&S0&E0 | S1&S0&E1

3.5.3 FSYNC and DATA1 Output Synchronization

The DATA1 OUT signal will always be delayed with respect to the incoming BCLK IN and FSYNC IN signals given internal data movement delays in the CLB tile (see Figure 3-10). In order to generate FSYNC OUT and DATA1 OUT signals that are aligned to each other, two FSMs were used to latch and hold these signals. The output latches are updated on the rising edge of BCLK IN.

FSM0 is used to detect and delay FSYNC IN. This means the FSYNC IN signal is not directly passed through the FSYNC OUT signal. The state diagram is shown in Figure 3-8 and the corresponding truth table is shown in Table 3-2.



Figure 3-8. FSM0 State Machine for TDM-8 Example

| Q1 | 50 | E1 (FSYNC high & | E0 (BCLK rise) | S1 Novt | S0 Novt | OUT |
|-----------|----|---------------------|-------------------|---------|---------|-----|
| 31 | 30 | BOLK IOW) | (BOLK IISE) | STNext | JUNEX | 001 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 |

Table 3-2. FSM0 Truth Table for TDM-8 Example

The reduced logic equations are:

- S1 (NEXT) = !E1&!E0&S1&!S0 | !E1&E0&!S1&S0
- S0 (NEXT) = !E1&!E0&!S1&S0 | E1&!E0&!S1&!S0
- OUT = !E1&S1&!S0 | !E0&S1&!S0

FSM2 is used to implement a simple D-type flip-flop which will latch and delay the DATA1_OUT signal. The truth table is shown in Table 3-3. Notice from Table 3-3 that S0 (next) is used to drive the final DATA1_OUT signal.

 Table 3-3. FSM2 Truth Table for TDM-8 Example

| E1 (DATA1) | E0 (BCLK rise) | S0 | S0 (next) |
|---------------|-------------------|----|-----------|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

The reduced logic equation is:

• S0 (NEXT) = !E0&S0 | E1&E0



3.6 Step 5: Simulate the Logic Design

A data receive operation is simulated in Figure 3-9. For this simulation a simple pattern of 0xAAAA AAAA is used as data input. The simulation shows the transition between the last bit of the last word in the TDM frame and the start of a new frame.



Figure 3-9. TDM-8 Data Receive Simulation

A data transmit operation is simulated in Figure 3-10. The simulation shows the transition between the last bit of the last word in the TDM frame and the start of a new frame. For this simulation a pattern of 0x5555 5554 is continuously transmitted to highlight the potential for a brief glitch at the output of the serializer in between serial words. Since the serializer must be loaded *after* it has shifted its counter value to avoid losing the most significant bit in the transmit word, there is a brief period of time when the serializer output is invalid.



Figure 3-10. TDM-8 Data Transmit Simulation



Notice that in this TDM example the HLC has been configured to push the value in C0 to the FIFO and then load a new value to C1 (see Figure 3-5). This is done deliberately to ensure C1 is loaded *after* the rising edge of BCLK_IN. Since the HLC action is triggered on the falling edge of BCLK and the serializer action is dependent on the rising edge of BCLK, there is an inherent dependency on the BCLK_IN period. If the BCLK_IN period is extended (BCLK_IN frequency is reduced or CLB clock frequency is increased), there is a risk that the C1 update happens before the rising edge of the clock, which causes a loss of the most-significant bit in the serial word.

As seen in Figure 3-10, there is a delay between the output data and the input FSYNC and BCLK. In order to output a FSYNC and DATA1 signal that are synchronized with each other, two FSMs are used to latch and delay these two signals. The simulation result of this feature is shown in Figure 3-11.



Figure 3-11. FSYNC and DATA1 Synchronization

3.7 Step 6: Test the CLB Logic

This section discusses the instructions on how to test the Code Composer Studio[™] example project provided for the CLB-based TDM-8 example. The example is tested on the F280025C controlCARD (TMDSCNCD280025C). However, the code can be easily ported to any other MCU with support for CLB Types 2 and above.

For test purposes, a separate F28388D MCU is used to generate a test TDM-8 stream using a McBSP port. The F28388D also receives the TDM-8 stream from the CLB and check for errors. An F28388D controlCARD (TMDSCNCD28388D) is used for this test.

In this test, the F28388D is configured for 200MHz operation with the McBSP generating a TDM waveform at 12.5MHz. The F280025C is configured for maximum 100MHz operation.

Figure 3-12 shows the required setup to test the TDM-8 example.



Figure 3-12. Test Setup for TDM Example



3.7.1 Hardware Setup and Connections

The following hardware is needed to run the demo:

- TMDSCNCD280025C controlCARD + TMDSHSECDOCK baseboard docking station + 5 V power supply
- TMDSCNCD28388D controlCARD + TMDSHSECDOCK baseboard docking station + 5 V power supply

Optional

• Logic Analyzer (for viewing TDM bus signals)

Setup the hardware as follows:

- 1. Insert the controlCARDs into their corresponding TMDSHSECDOCK. Both controlCARDs should be set to their default settings (see respective user guide for each EVM).
- 2. Wire the F28388D *output* TDM stream to the F280025C *input* TDM as shown in Table 3-4 and Table 3-5. Table 3-4. F280025C CLB Logic Input TDM Signals

| TDM Input Pin | GPIO (BALL) | DOCK Pin |
|---------------|-------------|----------|
| FSYNC_IN | GPIO00 | 49 |
| BCLK_IN | GPIO01 | 51 |
| DATA1_IN | GPIO02 | 53 |

Table 3-5. F28388D McBSP Output TDM Signals

| TDM Input Pin | GPIO (BALL) | DOCK Pin |
|---------------|-------------|----------|
| MCLKX/BCLK | GPIO22 | 72 |
| MFSX/FSYNC | GPIO23 | 74 |
| MDX/DATA1 | GPIO20 | 68 |

3. Wire the F280025C *output* TDM stream to the F28388D *input* TDM as shown in Table 3-6 and Table 3-7. Table 3-6. F280025C CLB Logic Output TDM Signals

| TDM Output Pin | GPIO (BALL) | DOCK Pin |
|----------------|-------------|----------|
| FSYNC_OUT | GPIO04 | 50 |
| BCLK_OUT | GPIO05 | 52 |
| DATA1_OUT | GPIO06 | 54 |

Table 3-7. F28388D McBSP Input TDM Signals

| | | <u> </u> |
|---------------|-------------|----------|
| TDM Input Pin | GPIO (BALL) | DOCK Pin |
| MCLKR/BCLK | GPIO58 | 108 |
| MFSR/FSYNC | GPIO59 | 110 |
| MDR/DATA1 | GPIO21 | 70 |

- 4. Wire a couple of common GND connections between the two docking stations.
- 5. Connect the corresponding USB cables to each controlCARD.
- 6. Connect the 5V supply to each TMDSHSECDOCK (alternatively use a separate USB cable to power the docking station).
- 7. Set S1 to the "EXT_ON" position on each TMDSHSECDOCK.

3.7.2 Software Setup

Two CCS projects are available in C2000ware to test this CLB example. The path to the TDM projects are:

- <C2000WARE_INSTALL>/driverlib/f28002x/examples/clb
- <C2000WARE_INSTALL>/driverlib/f2838x/examples/c28x/mcbsp

The following software packages are required to build and run the software:

- Code Composer Studio (CCS) version 11.1.00 or later
- C2000ware version 4.3.00.00 or later
- GNU compiler (TDM-GCC) & GTK Wave simulation viewer (optional for running CLB simulations. For more information, see CLB Tool's User Guide.

First, setup the F280025C by following these steps in CCS:

- 1. In the CCS menu, click 'Project -> Import CCS Projects...'.
- 2. Enter the path to the CLB example projects in the 'Select search-directory'.
 - a. Path: <C2000WARE_INSTALL>/driverlib/f28002x/examples/clb
- 3. Click 'Refresh'.
- 4. Select the 'clb_ex31_tdm_serial_port' project.

| 😚 Import CCS Projects | | | | × |
|---|--|----|----------|-----|
| mport CCS Projects Import existing CCS Project | s or example CCS Projects. | | P | ŝ |
| | | | | 2 |
| Select search-directory: | C:\ti\c2000\C2000Ware_4_02_00_00\driverlib\f28002x\exa | mj | Browse. | |
| ○ Select <u>a</u> rchive file: | | | Browse. | •• |
| Discovered projects: | | | | |
| Clb_ex21_clockpr | escalar_nmi [clb_ex21_clockprescalar_nmi.projectspec] r [clb_ex22_serializer.projectspec] | ^[| Select A | JI. |
| Clb_ex23_lfsr [cll | _ex23_lfsr.projectspec] | | Deselect | AII |
| Clb_ex24_lock_ou | <pre>tput_mask [clb_ex24_lock_output_mask.projectspec] ipeline [clb_ex25_input_pipeline.projectspec]</pre> | [| Refresh | i) |
| Clb_ex26_clockin | g_pipeline [clb_ex26_clocking_pipeline.projectspec] | | | |
| CID_ex27_spi_dat | a_export_fctb_ex27_spi_data_export.projectspecj a_export_dma_[clb_ex28_spi_data_export_dma.projectspec | _ | | |
| Clb_ex29_timesta | mp [clb_ex29_timestamp.projectspec] | | | |
| CID_ex3_auxiliary | rial_port [clb_ex31_tdm_serial_port.projectspec] | | | |
| Clb_ex4_pwm_pr | otection [clb_ex4_pwm_protection.projectspec] | | | |
| CID_exo_siggen | clb_exo_siggen.projectspec] ichine [clb_ex7_state_machine.projectspec] | | | |
| Clb_ex8_external | signal_AND_gate [clb_ex8_external_signal_AND_gate.proj | | | |
| 🗌 🚟 clb_ex9_timer [c | lb_ex9_timer.projectspec] | ~ | | |
| < | > | | | |
| Automatically import ref | erenced projects found in same search-directory | | | |
| Copy projects into works | pace | | | |
| 0 0 0 0 0 0 0 0 0 | | | | |
| Open <u>Resource Explorer</u> to t | rowse a wide selection of example projects | | | |
| | | | | |
| | | | | |
| ? | <u> </u> | | Cancel | |

Figure 3-13. Importing CLB TDM Example Project

- 5. Check 'Copy projects into workspace'.
- 6. Click 'Finish' to complete importing the project into the workspace.

Optional: Click 'Project -> Build Configurations -> Set Active -> CPU1_FLASH' to build the code for flash execution. Set the TMDSCNCD280025C controlCARD to boot from flash (see the TMS320F28388D controlCARD Information Guide).

- 7. In the CCS menu, click 'Project -> Build Project' to build the example project.
- 8. Click 'Run -> Debug' to load the executable to the F280025C target device.



9. Lastly, click 'Run -> Resume' in the CCS Debug perspective to run the code.

Second, setup the F28388D to generate the test TDM stream by following these steps in CCS:

- 1. In the CCS menu, click 'Project -> Import CCS Projects...'.
- 2. Enter the path to the McBSP example projects in the 'Select search-directory'.
- a. Path: <C2000WARE_INSTALL>/driverlib/f2838x/examples/c28x/mcbsp
- 3. Click 'Refresh'.
- 4. Select the 'mcbsp_ex7_tdm8_test' project.

| 😚 Import CCS Projects | | |
|---|--|--------------|
| mport CCS Projects Import existing CCS Project | s or example CCS Projects. | |
| Select search-directory: | C:\ti\c2000\C2000Ware_4_02_00_00\driverlib\f2838x\examp | Browse |
| Select <u>a</u> rchive file: | | Browse |
| <u>Discovered projects:</u> | | |
| mcbsp_ex1_loop | back [CCS/mcbsp_ex1_loopback.projectspec] | Select All |
| mcbsp_ex2_loop | back_interrupts_sysconfig [CCS/mcbsp_ex3_loopback_interr | Deselect All |
| mcbsp_ex3_loop mcbsp_ex4_spi_l | back_interrupts [CCS/mcbsp_ex3_loopback_interrupts.projecopback [CCS/mcbsp_ex4_spi_loopback.projectspec] | Refresh |
| < | <pre>8_test [CCS/mcbsp_ex7_tdm8_test.projectspec]</pre> | - |
| Automatically import ref | erenced projects found in same search-directory | |
| Open <u>Resource Explorer</u> to b | prover a wide selection of example projects | |
| ? | Einish | Cancel |

Figure 3-14. Importing McBSP TDM Example Project

- 5. Check 'Copy projects into workspace'.
- 6. Click 'Finish' to complete importing the project into the workspace.
 - a. Optional: Click 'Project -> Build Configurations -> Set Active -> CPU1_FLASH' to build the code for flash execution. Set the TMDSCNCD28388D controlCARD to boot from flash (see controlCARD user's guide).
- 7. In the CCS menu, click 'Project -> Build Project' to build the example project.
- 8. Click 'Run -> Debug' to load the executable to the F28388D target device.
 - a. Optional: Add the following global variables to the Expressions window: txData, rxData, testWordDetected, and errCountGlobal.
- 9. Lastly, click 'Run -> Resume' in the CCS Debug perspective to run the code.

3.7.3 Testing Output Setup and Hold Times

A key consideration in the output TDM stream from the CLB is the expected setup and hold time seen by the receiving device. To measure the setup and hold time, an oscilloscope was used to continuously capture the output of the CLB. Figure 3-15 shows the TDM output without the latch and delay logic added to the FSYNC_OUT and DATA1_OUT signals. The setup and hold time of DATA1_OUT with respect to BCLK_OUT reduced due to the delay in the DATA1_OUT signal.





Figure 3-15. TDM-8 Output Without Latch and Delay Logic

The oscilloscope plot shown in Figure 3-16 shows the final TDM output with the latch and delay logic added to the FSYNC_OUT and DATA1_OUT signals. The setup and hold time are greatly improved using this approach.



Figure 3-16. Final TDM-8 Output

Since BCLK_OUT must be passed through the CLB tile without jitter, it is not possible to accurately control the timing between BCLK_OUT and the other two signals without introducing jitter on BCLK_OUT. However, in this case, the setup and hold time are acceptable. If needed the BCLK_OUT signal can be inverted using the AOC in the CLB tile to trade setup for hold time.

Finally note the 10 ns jitter on the FSYNC_OUT and DATA1_OUT signals. This is due to the CLB tile sampling the incoming BCLK_IN signal at 100 MHz. The jitter reduces the setup and hold time further.



3.7.4 Testing Data Integrity

To test data integrity, the F28388D sends an incrementing data pattern to the F280025C. The F280025C checks the incoming data against expected data and generates a flag if any errors are detected. The F280025C echoes back all data received to the F28388D. Finally, the F28388D checks all incoming data for errors. Figure 3-17 shows a snap shot of the F28388D monitoring in progress. Note that no errors were detected by either the F280025C or the F28388D during the transmission of the entire 32-bit range.

| Expression | Туре | Value | Address | |
|--|---------------|------------------|-----------------|--|
| 🕪 txData | unsigned long | 0x843B2B6A (Hex) | 0x0000A804@Data | |
| (x)= testWordDetected | unsigned int | 1 | 0x0000A800@Data | |
| Interest end of the second | unsigned long | 0 | 0x0000A802@Data | |
| Add new expression | | | | |

Figure 3-17. F28388D Monitoring Incoming TDM Traffic

4 Example B: Using the CLB to Implement a Custom Communication Bus for LED Driver in Lighting Applications

4.1 Example Overview

In this example, a C2000 real-time microcontroller is used to display a simple pattern on a 16 x 16 RGB LED matrix display. The LED matrix display is driven by one LP5891-Q1 LED driver that communicates with the C2000 real-time microcontroller using a serial bus.

Two CLB tiles are used to support the Continuous Clock Series Interface (CCSI) bus protocol required to communicate with the LP5891-Q1 LED driver devices. The CLB-based CCSI bus implementation requires minimum CPU overhead to encode a bus frame since the IDLE, START, END, and CHECK bits required to communicate across the CCSI bus are automatically added by the CLB.

The example code can be modified to cascade an additional LED matrix display.



Figure 4-1. Block Diagram for LED Matrix Example

4.2 Step 1: Understand Design Requirements

The design parameters used for this demo are listed in Table 4-1.

| Design Parameter | Value |
|---------------------|---------------------------|
| Display module size | 16 x 16 RGB LEDs |
| Frame rate | 60 Hz |
| Refresh rate | 7680 Hz |
| PWM resolution | 16 bits |
| Cascaded devices | 1 (can be increased to 2) |
| CCSI bus count | 1 |
| SCLK frequency | 5.0 MHz |
| GCLK frequency | 70.0 MHz |
| T _{SW} | 643 ns |

The CCSI bus protocol supported by the LP5891-Q1 LED driver device is shown in Figure 4-2. The CCSI protocol includes the following requirements:

- Continuous serial clock
- · Variable frame lengths with start, idle, data, and end states
- 16-bit head and data words
- · Check-bit appended on all head and data words





Two additional requirements are added. First, configurable single and dual clock-edge transmission and reception is required to support different LED driver devices. For example, the TLC6983 LED display driver uses dual clock-edge data transmission and reception. Second, the ability to generate simultaneous VSYNC commands on multiple CCSI buses is desired. The VSYNC is used to sync the display of each frame for the devices in a cascaded chain.



4.3 Step 2: Identify Required Inputs to the CLB Tile

The input signals to the TX tile are shown in Figure 4-3.



Figure 4-3. TX Tile Input Signals

The TX tile use the following inputs:

- CLB input 0: This input is internally driven by the PWMnA output. The PWMnA output is configured to generate a clock at 2x the target SCLK frequency. The TX tile transmits data using the PWMnA clock.
- CLB input 3: This input is internally driven by the PWMnA output. The input is passed through the tile and used to drive the CLB_SCLKX2 output.
- CLB input 4: This input is internally driven by a PWM*n*B output. The PWM*n*B output is configured to generate a clock at the target SCLK frequency. The input is passed through the tile and used to drive the CLB_SCLK output.

The TX tile also uses three auxiliary signals for synchronizing data transmission on all TX tiles:

- CLB input 1: The GPREG.1 bit is used to start a data transfer on the TX tile. It is set/cleared by the CPU.
- CLB input 5: This input used as an external transfer start signal to the TX tile. This input is always taken from CLB tile 2, output 5.
- CLB input 6: The GPREG.6 bit drives the external transfer start signal on all TX tiles in cases where more than one TX tile is used. Only GPREG.6 in CLB tile 2 can be used to start transfers in other TX tiles. The GPREG.6 bit is set/cleared by the CPU.

The input signals to the RX tile are shown in Figure 4-4.



Figure 4-4. RX Tile Input Signals

The RX tile uses the following inputs:

- CLB input 1: This input is used to receive data from the SIN pin.
- CLB input 2: This input is used to trigger data reception on the falling edge of CLB_SCLKX2. The CLB_SCLKX2 signal is an external clock provided by the TX CLB tile which runs at 2x the frequency of SCLK.
- CLB input 3: The GPREG.3 bit is used to enable/disable data reception. It is set/cleared by the CPU.

4.4 Step 3: Identify Required Outputs From CLB Logic

The output signals of the TX tile are shown in Figure 4-5.







The SCLK and SCLKX2 clocks generated internally by the PWM are passed through the TX tile and driven on the GPIO pins. The output of the TX tile is driven on the CLB_SOUT pin. Finally, output 5 of the tile is driven to the CLB XBAR where it can be used to synchronize data transmission across multiple tiles.

The RX tile logic does not have any output signals.

4.5 Step 4: Design the CLB Logic

The final design uses two CLB tiles to implement a single CCSI serial port. One tile is used for transmit operations and a second tile is used for receive operations. The TX and RX tile logic can be replicated to multiple tiles to create additional CCSI buses.

4.5.1 TX Tile Logic

The logic for the TX tile is shown in Figure 4-6.



Figure 4-6. CLB TX Tile Logic

The following CLB tile resources are used to implement the CCSI bus transmit (TX) functionality:

- · COUNTER0 is used to count the number of words to transmit. It is operated in normal counter mode.
- COUNTER1 is used to transmit the output data on SCLKX2 falling edges. It is operated in serializer mode.
- COUNTER2 is used to count SCLKX2 falling edges. It is operated in normal counter mode.
- The HLC is used to move data into COUNTER1 from the CLB PULL FIFO. It also generates interrupts to the CPU when four 16-bit words are transmitted and when the full frame has been transmitted.
- FSM0 is used to cycle through IDLE, ACTIVE, and END states.

FSM2 is used to count when four 16-bit words have been transmitted. An event is generated to the HLC
when the transmit count reaches four.

Several LUTs and output LUTs are used for combinatorial logic.

FSM0 define three states of operation for the TX tile. The FSM0 state diagram and truth table are shown in Figure 4-7 and Table 4-2.



Figure 4-7. FSM0 State Diagram for TX Tile

| | Table | 4-2. | FSM0 | Truth | Table |
|--|-------|------|------|-------|-------|
|--|-------|------|------|-------|-------|

| S1 | SO | E1 (TRANSFER_ START) | E0 (C0_MATCH1) | S1 Next | S0 Next | Output (FSM0_OUT) |
|----|----|----------------------------|-------------------|---------|---------|----------------------|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 |

The reduced logic equations are:

- S1 = !S1&S0&E0 | S1&!S0&E1
- S0 = !S1&E1&!E0 | !S1&S0&!E0
- OUTPUT = !S0 | S1

FSM2 is used to trigger an HLC event once four 16-bit words have been transmitted. The FSM2 state diagram and truth table are shown in Figure 4-8 and Table 4-3.





E1/E0 = 0/1 (C2_MATCH1)

Figure 4-8. FSM2 State Diagram for TX Tile

| Table | 4-3. | FSM2 | Truth | Table | |
|-------|------|------|-------|-------|--|
| | | | | | |

| S1 | S0 | E1 (FSM0_OUT) | E0 (C2_MATCH1) | S1 Next | S0 Next | Output (HLC_INT) |
|----|----|------------------|-------------------|---------|---------|---------------------|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 |

The reduced logic equations are:

- S1 = S1&!S0&!E1 | S1&!E1&!E0 | !S1&S0&!E1&E0
- S0 = !S0&!E1&E0 | S0&!E1&!E0
- OUTPUT = S1&S0&!E1&E0

4.5.2 RX Tile Logic

The logic for the receive tile is shown in Figure 4-9.



Figure 4-9. CLB RX Tile Logic

The following CLB tile resources are used to implement the CCSI bus receive (RX) functionality:

- COUNTER0 is used to detect the end of a receive frame.
- COUNTER1 is used to receive the input data on SCLKX2 falling edges. It is operated in serializer mode.
- COUNTER2 is used to count SCLKX2 falling edges. It is operated in normal counter mode.
- The HLC is used to move data from COUNTER1 to the CLB PUSH FIFO. It also generates interrupts to the CPU when four 16-bit words are received and when a CHECK bit error and END of frame condition are detected.
- FSM0 is used to define and cycle through two states, IDLE and ACTIVE.
- FSM1 is used to count when four 16-bit words have been received. An event is generated to the HLC when the receive count reaches four.
- FSM2 is used to generate an END of frame interrupt only when in the ACTIVE state.
- LUT0 is used to detect a "0" bit corresponding which is used in detecting the START bit as well as the END byte. The LUT0 output can be forced low, effectively disabling receive operations, by clearing the ENABLE_RX input.
- LUT1 is used to verify the CHECK bit on each 16-bit word. If the CHECK bit does not match the expected state, the LUT1 output will trigger an HLC interrupt.
- LUT2 is used to keep COUNTER2 in reset when receive operations are disabled (i.e. ENABLE_RX = 0).

FSM0 define two states of operation for the RX tile. The FSM0 state diagram and truth table are shown in Figure 4-10 and Table 4-4.



Figure 4-10. FSM0 State Diagram for RX Tile

| SO | E1 (LUT0_OUT) | E0 (C0_MATCH1) | S0 Next | Output (FSM0_OUTPUT) |
|----|---------------|----------------|---------|----------------------|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 |

The full logic equations are:

- S0 = !S1&!S0&E1&!E0 | !S1&!S0&E1&E0 | !S1&S0&!E1&!E0 | !S1&S0&E1&!E0
- OUTPUT = !S1&S0&!E1&!E0 | !S1&S0&!E1&E0 | !S1&S0&E1&!E0 | !S1&S0&E1&E0

FSM1 is used to trigger an HLC event once four 16-bit words have been received. The FSM1 state diagram and truth table are shown in Figure 4-11 and Table 4-5.



Figure 4-11. FSM1 State Diagram for RX Tile

| | | - able 4 | 4-5. FSM1 Iruth | Table | | |
|----|----|---------------|-----------------|---------|---------|---------------------|
| S1 | SO | E1 (FSM0_OUT) | E0 (C2_MATCH2) | S1 Next | S0 Next | Output (HLC_INT) |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 |

The logic equations are:

- S1 = !S1&S0&E1&E0 | S1&!S0&E1&!E0 | S1&!S0&E1&E0 | S1&S0&E1&!E0
- S0 = !S1&!S0&E1&E0 | !S1&S0&E1&!E0 | S1&!S0&E1&E0 | S1&S0&E1&!E0
- OUTPUT = S1&S0&E1&E0

4.5.3 Data Clocking

The clocks needed for the CCSI bus communication are generated using an on-chip PWM. The PWM*n*A output generates a PWM_SCLKX2 clock running at double the target SCLK frequency, while the PWM*n*B output is generates a PWM_SCLK clock at the required SCLK frequency.

The TX and RX CLB tiles always transmit and receive data using the PWM_SCLKX2 clock. Only PWM_SCLK is eventually driven to the LED drivers in the system.

Dual- or single-clock edge data transmission and reception is achieved by configuring the source of the PWM_SCLK input to the TX tile. When single-clock edge transmission and reception is required, for example, when using the LP5891-Q1 LED driver, the TX input tile configuration can be modified to drive the PWM*n*A output to both PWM_SCLK and PWM_SCLKX2 inputs, see Figure 4-12. The PWM*n*B output can be left unconnected from the CLB tiles.



Figure 4-12. Single-Clock Edge Data Transmission and Reception Clock Configuration

Dual-clock edge data transmit/receive allows for the transmission and reception on both edges of the clock, see Figure 4-13.



Figure 4-13. Dual-Clock Edge Data Transmit/Receive Using PWM_SCLKX2

When dual-clock edge transmission and reception is required, the TX input tile configuration can be modified to drive the PWM*n*A output to the PWM_SCLKX2 input and the PWM*n*B output to the PWM_SCLK input, see Figure 4-14.



Figure 4-14. Dual-Clock Edge Data Transmission and Reception Clock Configuration

To mitigate timing delays introduced by the CLB logic, both of the PWM clock inputs are passed through the TX CLB logic.



4.6 Step 5: Simulate the Logic Design

A data receive operation is simulated in Figure 4-15. For this simulation a simple pattern of 0xAAAA is used as data input. The simulation shows the beginning of the frame indicated by the START bit, the capture of the full received word, and the verification of the CHECK bit.



Figure 4-15. Data Receive Simulation for LED Driver

A second data receive simulation is shown in Figure 4-16. In this simulation, an incorrect CHECK bit is added to the incoming data stream to verify the operation of the CHECK bit logic. The simulation shows the output of the LUT1 block go high to indicate a CHECK bit error is detected.



Figure 4-16. Check Bit Error Logic Simulation

Finally, the simulation in Figure 4-17 shows the detection of the END bits. The FSM0 block transitions the CLB logic to the IDLE state upon detection of the END bits.



Figure 4-17. END Frame Detection

Note

Both a receive interrupt and CHECK bit error (not shown) will be generated during the END bits. The CPU code will have to always discard the last word in a frame as this corresponds to the END bits.



A data transmit operation is simulated in Figure 4-18. The simulation shows the transmission of a single 0x5555 word, starting from the START bit and ending with the END frame. To simplify the CLB logic, the START bit is implemented by allowing the tile to transmit a 0xFFFF word. The START bit is generated on the 17th clock cycle using the CHECK bit logic of the tile. Similarly, to generate the END frame, the CLB logic simply sets the CLB_SOUT signal to 1 at the end of the last word transmission. The CLB logic depends on the CPU to wait at least 18 PWM_SCLKX2 cycles before starting a new data transmission.



Figure 4-18. Data Transmit Simulation for LED Driver



Example B: Using the CLB to Implement a Custom Communication Bus for LED Driver in Lighting Applications

An output glitch is observed during data transmit operations due to the delay in updating the output serializer (Counter 1). Figure 4-19 shows the internal delays starting from the PWM_SCLKX2 low edge to the final update of the output serializer. The result of the glitch is a reduced setup time for the receiving device. The setup time should not be calculated from the rising edge of the PWM_SCLKX2 signal since this clock will also be delayed as it passes through the CLB tile. Instead, the setup time should be calculated from the CLB_SCLKX2 signal. This signal is a delayed version of PWM_SCLKX2 which the receiving device will observe.



Figure 4-19. Output Glitch

The glitch can be elimited by latching the serializer output before passing to CLB_SOUT. However, this approach is not used since at least two CLB cycles of setup time are expected per the transmit simulation which is enough to meet the 10ns setup time required by the LP5891-Q1 device.

4.7 Step 6: Test the CLB Logic

This section discusses the instructions on how to test the Code Composer Studio[™] example project provided for the LED driver example. The example is tested on the F280039C LaunchPad (LAUNCHXL-F280039C).

For test purposes, one LP5891-Q1 EVM is used to receive and display the image data from the F280039C MCU. The LP5891-Q1 EVM includes a 16x16 matrix LED display.

4.7.1 Hardware Setup and Connections

The following hardware is needed to run the demo:

- 1x LAUNCHXL-F280039C LaunchPad
- 1x LP5891Q1EVM matrix LED display

Optional

• Logic Analyzer (for viewing CCSI bus signals)

Setup the hardware as follows:

- 1. Setup the LAUNCHXL-F280039C EVM with default settings:
 - a. Populate JP1 to connect the 5 V power and GND from the USB-C connector to the XDS110 side of the board.
 - b. Populate JP2 to connect the 5 V power on the XDS110 side of the board to the rest of the LaunchPad.
 - c. Populate the TCK and TMS jumpers on J101 to connect the XDS110 to the F280039C device (the other jumpers on J101 can be also populated if desired).
 - d. S3 boot mode set to Flash boot (optional).

- 2. Setup the LP5891Q1EVM as follows:
 - a. Switch S1 set to SOUTHOST to route the serial data output from the LP5891-Q1 device back to the host controller.
 - b. Jumper J6 set to MCU5V to select the 5 V supply from the host controller.
 - c. Jumper J7 set to "3V3 VR" to select 3.3 V power supply for VLEDR.
- 3. Plug in the LP5891Q1EVM to site 1 on the LAUNCHXL-F280039C EVM. Use the 5 V, 3V3, and GND markings on both boards as a guide to correctly orient the boards.
- Connect a USB-C cable to the USB header on the LAUNCHXL-F280039C EVM. When the USB cable is
 plugged into a USB2.0/USB3.x port, the LaunchPad and LP5891Q1EVM receives power from the USB port.
- 5. Proceed to the software setup.

4.7.2 Software Setup

The following software packages are required to build and run the software:

- Code Composer Studio (CCS) version 11.1.0 or later
- C2000ware version 4.02.00.00 or later

(optional) GNU compiler (TDM-GCC) and GTK Wave simulation viewer for running simulations. For more information, see the *CLB Tool's User's Guide*.

Setup the LED example software by following these steps in CCS:

- 1. In the CCS menu, click 'Project -> Import CCS Projects...'.
- 2. Enter the path to the F28003x driverlib CLB example projects in the 'Select search-directory'.
 - a. Path: <C2000WARE_INSTALL>\driverlib\f28003x\examples\clb
- 3. Click 'Refresh'.
- 4. Select the 'clb_ex32_led_driver' project.

| apport CCS Projects mport existing CCS Projects or example CCS Projects. Select sgarch-directory: C:\ti\c2000\C2000Ware_4_02_00_00\driverlib\f28003x\example Bgowse. Select archive file: bselect archive file: Bgowse. Select archive file: Bgowse. Select archive file: Bgowse. Select archive file: Bgowse. Select archive file: Bgowse. Select archive file: Bgowse. Select archive file: Bgowse. Select Archive file: Bgowse. Select Archive file: Bgowse. Select Archive file: Select Archive file: Select Archive file: CS/clb_ex25_input_pipeline [CCS/clb_ex26_clocking_pipeline.projectspec] Gcb_ex32_spi_data_export_dma [CCS/clb_ex3_auxiliary_pwm.projectspec] Gcb_ex3_auxiliary_pwm [CCS/clb_ex3_auxiliary_pwm.projectspec] Gcb_ex3_gied_driver [CCS/clb_ex3_sigen_projectspec] Gcb_ex3_gied_driver [CCS/clb_ex6_siggen_projectspec] Gcb_ex5_siggen [CCS/clb_ex6_siggen_projectspec] Gcb_ex5_siggen [CCS/clb_ex6_siggen_projectspec] Gcb_ex5_siggen [CCS/clb_ex6_siggen_projectspec] Gcb_ex5_siggen [CCS/clb_ex6_siggen_projectspec] Gcb_ex5_siggen [CCS/clb_ex6_siggen_projectspec] Gcb_ex6_siggen [CCS/clb_ex6_siggen_projectspec] Gcb_ex6_siggen [CCS/cl | | | |
|--|--|--|------------|
| Select sgarch-directory: C:\ti\c2000\C2000Ware_4_02_00_00\driverlib\f28003x\examplexam | mport CCS Projects Import existing CCS Project | s or example CCS Projects. | |
| Select sgarch-directory: C:(t)(2000(C2000Ware_4_0_2_00_00)(driverlib/t28003x/exam] Browse. Select archive file: Browse. iscovered projects: Select Archive file: Browse. ic clb_ex25_input_pipeline [CCS/clb_ex24_lock_output_mask.projectspec] Select Archive file: Select Archive file: ic clb_ex25_input_pipeline [CCS/clb_ex26_clocking_pipeline.projectspec] Select Archive file: Select Archive file: ic clb_ex27_spi_data_export dma [CCS/clb_ex26_clocking_pipeline.projectspec] Select Archive file: Select Archive file: ic clb_ex32_input_pipeline [CCS/clb_ex29_timestamp.projectspec] Refresh Refresh ic clb_ex30_cyclic_redundancy_check [CCS/clb_ex3_auxiliary_pwm.projectspec] Refresh ic clb_ex30_ycuti_redundancy_check [CCS/clb_ex3_led_driver.projectspec] Refresh ic clb_ex30_ycuti_redundancy_check [CCS/clb_ex3_led_driver.projectspec] Refresh ic clb_ex30_ycuti_redundancy_check [CCS/clb_ex3_led_driver.projectspec] Refresh ic clb_ex31_tdm_serial_port [CCS/clb_ex3_sevent_window.projectspec] Refresh ic clb_ex4_pwm_protection.projectspec] Refresh ic clb_ex5_event_window [CCS/clb_ex7_state_machine.projectspec] Refresh ic clb_ex6_siggen [CCS/clb_ex6_siggen.projectspec] Refresh ic clb_ex6_siggen | | | |
| Select archive file: Browse. iscovered projects: | Select search-directory: | C:\ti\c2000\C2000Ware_4_02_00_00\driverlib\f28003x\examp | Browse |
| iscovered projects: Conversed | Select archive file: | | Browse |
| Image: clb_ex24_lock_output_mask [CCS/clb_ex24_lock_output_mask.projectspec] Image: clb_ex25_input_pipeline [CCS/clb_ex25_input_pipeline.projectspec] Image: clb_ex26_clocking_pipeline [CCS/clb_ex26_clocking_pipeline.projectspec] Image: clb_ex27_spi_data_export_dma [CCS/clb_ex27_spi_data_export.projectspec] Image: clb_ex28_spi_data_export_dma [CCS/clb_ex23_spi_data_export_dma.projectspec] Image: clb_ex29_timestamp [CCS/clb_ex32_spi_data_export_dma.projectspec] Image: clb_ex3_auxiliary_pwm [CCS/clb_ex3_auxiliary_pwm.projectspec] Image: clb_ex3_cyclic_redundancy_check [CCS/clb_ex3_cyclic_redundancy_che Image: clb_ex3_tim_serial_port [CCS/clb_ex3_tim_serial_port.projectspec] Image: clb_ex3_tedm_serial_port [CCS/clb_ex3_tedm_serial_port.projectspec] Image: clb_ex3_signal_And_gate [CCS/clb_ex3_tedm_serial_port.projectspec] Image: clb_ex3_signal_AND_gate [CCS/clb_ex3_textermal_signal_AND_gate] Image: clb_ex4_pimer [CCS/clb_ex5_tot] Image: clb_ex4_pimer [CCS/clb_ex9_timer.projectspec] Image: clb_ex4_tot] Image: clb_ex4_tot] Image: clb_ex4_tot] Image: clb_ex5_signen [CCS/clb_ex6_signen.projectspec] Image: clb_ex4_tot] Image: clb_ex4_tot] Image: clb_ex4_tot] Image: clb_ex4_tot] Image: clb_ex4_tot] Image: clb_ex4_tot] | Discovered projects: | | |
| Imple_pipeline [CCS/clb_ex22_input_pipeline.projectspec] Imple_clb_ex26_clocking_pipeline [CCS/clb_ex26_clocking_pipeline.projectspec] Imple_clb_ex27_spi_data_export [CCS/clb_ex26_clocking_pipeline.projectspec] Imple_clb_ex28_spi_data_export [CCS/clb_ex27_spi_data_export_projectspec] Imple_clb_ex32_spi_data_export_dma [CCS/clb_ex29_timestamp.projectspec] Imple_clb_ex3_uxiliary_pwm [CCS/clb_ex29_timestamp.projectspec] Imple_clb_ex3_uxiliary_pwm [CCS/clb_ex3_auxiliary_pwm.projectspec] Imple_clb_ex3_uxiliary_pwm [CCS/clb_ex3_auxiliary_pwm.projectspec] Imple_clb_ex3_uxiliary_pwm [CCS/clb_ex3_auxiliary_pwm.projectspec] Imple_clb_ex3_uxiliary_pwm [CCS/clb_ex3_auxiliary_pwm.projectspec] Imple_clb_ex3_uxiliary_pwm [CCS/clb_ex3_auxiliary_pwm.projectspec] Imple_clb_ex3_uxiliary_pwm [CCS/clb_ex3_led_driver.projectspec] Imple_clb_ex3_uxiliary_pwm_protection.projectspec] Imple_clb_ex3_uxiliary_pwm_protection.projectspec] Imple_clb_ex5_event_window [CCS/clb_ex3_sevent_window.projectspec] Imple_clb_ex6_siggen [CCS/clb_ex7_state_machine.projectspec] Imple_clb_ex3_uxitiary_pwm [CCS/clb_ex7_state_machine.projectspec] Imple_clb_ex6_siggen [CCS/clb_ex8_external_signal_AND_gate [CCS/clb_ex8_external_signal_AND_gate [CCS/clb_ex8_external_signal_AND_gate [CCS/clb_ex8_external_signal_AND_gate [CCS/clb_ex8_external_signal_aND_gate [CCS/clb_ex8_external_signal_aND_gate [CCS/clb_ex8_external_signal_aND_gate [CCS/clb_ex8_external_signal_aND_ | clb_ex24_lock_ou | utput_mask [CCS/clb_ex24_lock_output_mask.projectspec] | Select All |
| Image: clb_ex27_spi_data_export ICCS/clb_ex27_spi_data_export.projectspec] Image: clb_ex28_spi_data_export_dma ICCS/clb_ex28_spi_data_export_dma.projectspec] Image: clb_ex29_timestamp ICCS/clb_ex29_timestamp.projectspec] Image: clb_ex30_cyclic_redundancy_check ICCS/clb_ex32_uclilary_pwm.projectspec] Image: clb_ex31_tdm_serial_port ICCS/clb_ex31_tdm_serial_port.projectspec] Image: clb_ex32_led_driver ICCS/clb_ex32_led_driver.projectspec] Image: clb_ex32_led_driver ICCS/clb_ex32_led_driver.projectspec] Image: clb_ex4_pwm_protection ICCS/clb_ex4_pwm_protection.projectspec] Image: clb_ex5_siggen ICCS/clb_ex7_state_machine.projectspec] Image: clb_ex5_siggen ICCS/clb_ex7_state_machine.projectspec] Image: clb_ex6_siggen ICCS/clb_ex7_state_machine.projectspec] Image: clb_ex6_siggen ICCS/clb_ex7_state_machine.projectspec] Image: clb_ex9_timer ICCS/clb_ex9_timer.projectspec] | Clb_ex26_clockin | g_pipeline [CCS/clb_ex26_clocking_pipeline.projectspec] | Deselect A |
| cb_ex28_spi_data_export_dma [CCS/cb_ex28_spi_data_export_dma.project cb_ex29_timestamp [CCS/clb_ex29_timestamp.projectspec] cb_ex3_auxiliary_pwm [CCS/clb_ex3_auxiliary_pwm.projectspec] cb_ex31_tdm_serial_pott [CCS/clb_ex3_auxiliary_pwm.projectspec] cb_ex31_tdm_serial_pott [CCS/clb_ex32_ted_driver.projectspec] cb_ex4_pwm_protection [CCS/clb_ex32_ted_driver.projectspec] cb_ex5_event_window [CCS/clb_ex5_event_window.projectspec] cb_ex5_siggen [CCS/clb_ex6_siggen.projectspec] cb_ex6_siggen [CCS/clb_ex6_siggen.projectspec] cb_ex7_state_machine [CCS/clb_ex7_state_machine.projectspec] cb_ex8_external_signal_AND_gate [CCS/clb_ex8_external_signal_AND_gate] cb_ex9_timer [CCS/clb_ex9_timer.projectspec] cd_ex9_timer [CCS/clb_ex9_timer.projectspec] cd_ex9_timer.projectspec] cd_ex9_timer.projectspec] cd_ex9_ | 🔲 🚘 clb_ex27_spi_dat | a_export [CCS/clb_ex27_spi_data_export.projectspec] | Refresh |
| <pre>cl_exs_initiaty_pwm [CCS/clb_ex3_auxiliary_pwm.projectspec] clb_ex3auxiliary_pwm [CCS/clb_ex3_auxiliary_pwm.projectspec] clb_ex31_tdm_serial_port [CCS/clb_ex31_tdm_serial_port.projectspec] clb_ex4_pwm_protection [CCS/clb_ex32_led_driver.projectspec] clb_ex4_pwm_protection [CCS/clb_ex4_pwm_protection.projectspec] clb_ex5_event_window [CCS/clb_ex5_event_window.projectspec] clb_ex6_siggen [CCS/clb_ex6_siggen.projectspec] clb_ex7_state_machine [CCS/clb_ex7_state_machine.projectspec] clb_ex8_external_signal_AND_gate [CCS/clb_ex8_external_signal_AND_gate clb_ex8_external_signal_AND_gate [CCS/clb_ex8_external_signal_AND_gate clb_ex9_timer [CCS/clb_ex9_timer.projectspec] clb_ex9_timer [CCS/clb_ex9_timer.projectspec] cdb_ex9_timer [CCS/clb_ex9_timer.projectspec] cdb_ex9_timer.projectspec</pre> | □ 🔄 clb_ex28_spi_dat | a_export_dma [CCS/clb_ex28_spi_data_export_dma.projec mp_[CCS/clb_ex29_timestamp_projectspec] | - |
| Image: clb_ex30_cyclic_redundancy_check [CCS/clb_ex31_tdm_serial_port.projectspec] Image: clb_ex31_tdm_serial_port [CCS/clb_ex31_tdm_serial_port.projectspec] Image: clb_ex32_led_driver [CCS/clb_ex32_led_driver.projectspec] Image: clb_ex4_pwm_protection [CCS/clb_ex5_event_window.projectspec] Image: clb_ex5_event_window [CCS/clb_ex5_event_window.projectspec] Image: clb_ex6_siggen [CCS/clb_ex6_siggen.projectspec] Image: clb_ex7_state_machine [CCS/clb_ex7_state_machine.projectspec] Image: clb_ex8_external_signal_AND_gate [CCS/clb_ex8_external_signal_AND_gate Image: clb_ex9_timer [CCS/clb_ex9_timer.projectspec] Image: clb_ex9_timer [clb_ex9_timer.projectspec] Image: clb_ex9_timer.projectspec] Image: clb_ex9_timer.projectspec] Image: clb_ex9_timer | Clb_ex3_auxiliary | _pwm [CCS/clb_ex3_auxiliary_pwm.projectspec] | |
| | Clb_ex30_cyclic_i | redundancy_check [CCS/clb_ex30_cyclic_redundancy_che | |
| Cib_exs2_led_onver [CCS/clb_exs2_led_onver.projectspec] Cib_exs4_pwm_protection [CCS/clb_exs4_pwm_protection.projectspec] Cib_exs5_event_window [CCS/clb_exs5_event_window.projectspec] Cib_exs6_siggen [CCS/clb_exs5_event_window.projectspec] Cib_exs7_state_machine [CCS/clb_exs7_state_machine.projectspec] Cib_exs8_external_signal_AND_gate [CCS/clb_ex8_external_signal_AND_gate Clb_exs9_timer [CCS/clb_ex9_timer.projectspec] Cib_exs9_timer [CCS/clb_ex9_timer.projectspec] Conv_projects into workspace | Clb_ex31_tdm_se | rial_port [CCS/clb_ex31_tdm_serial_port.projectspec] | |
| Image: Clb_ex5_event_window [CCS/clb_ex5_event_window.projectspec] Image: Clb_ex6_siggen [CCS/clb_ex6_siggen.projectspec] Image: Clb_ex7_state_machine [CCS/clb_ex7_state_machine.projectspec] Image: Clb_ex8_external_signal_AND_gate [CCS/clb_ex8_external_signal_AND_gate Image: Clb_ex9_timer [CCS/clb_ex9_timer.projectspec] Image: Clb_ex9_timer [CLB_ex9_timer.projectspec] Image: Clb_ex9_timer [CLB_ex9_timer.projectspec] | CID_ex32_led_driv | otection [CCS/clb_ex32_led_driver.projectspec] | |
| Gib_ex6_siggen [CCS/clb_ex6_siggen.projectspec] Gib_ex7_state_machine [CCS/clb_ex7_state_machine.projectspec] Gib_ex8_external_signal_AND_gate [CCS/clb_ex8_external_signal_AND_gate Gib_ex9_timer [CCS/clb_ex9_timer.projectspec] v | Clb_ex5_event_w | indow [CCS/clb_ex5_event_window.projectspec] | |
| Clock Clock / state_machine [CCS/clb_ex/_state_machine.projectspec] Clock & Sexternal_signal_AND_gate [CCS/clb_ex8_external_signal_AND_gate CCS/clb_ex9_timer.projectspec] Clock & Sexternal_signal_AND_gate [CCS/clb_ex9_timer.projectspec] Automatically import referenced projects found in same search-directory Conv projects into workspace | Clb_ex6_siggen | [CCS/clb_ex6_siggen.projectspec] | |
| Automatically import referenced projects found in same search-directory | Clb_ex7_state_ma | signal AND gate [CCS/clb ev8 external signal AND gate | |
| Automatically import referenced projects found in same search-directory | Clb_ex9_timer [C | CCS/clb_ex9_timer.projectspec] | |
| Automatically import referenced projects found in same search-directory | | × | |
| Automatically import referenced projects found in same search-directory | < | > | |
| The last state of the state of | Automatically import ref Copy projects into works | erenced projects found in same search-directory | |
| | Open <u>Resource Explorer</u> to b | prowse a wide selection of example projects | |
| pen Resource Explorer to browse a wide selection of example projects | | | |
| pen <u>Resource Explorer</u> to browse a wide selection of example projects | | | |
| pen <u>Resource Explorer</u> to browse a wide selection of example projects | | | |

Figure 4-20. Importing CLB LED Driver Example Project



- 5. Check 'Copy projects into workspace'.
- 6. Click 'Finish' to complete importing the project into the workspace.

Optional: Click 'Project -> Build Configurations -> Set Active -> CPU1_LAUNCHXL_FLASH' to build the code for flash execution.

- 7. In the CCS menu, click 'Project -> Build Project' to build the example project.
- 8. Right-click the TMS320F280039C_LaunchPad.ccxml file in the project and select "Set as Active Target Configuration".
- 9. Click 'Run -> Debug' to load the executable to the F280039C target device.
- 10. Lastly, click 'Run -> Resume' in the CCS Debug perspective to run the code.

4.7.3 Testing Output Setup and Hold Times

A key consideration in the output CCSI bus from the CLB is the expected setup and hold time seen by the receiving device. To measure the setup and hold time, an oscilloscope was used to continuously capture the output of the CLB.

As shown in Figure 4-21, the CLB output has a minimum setup time of approximately 55 ns which is more than enough to meet the 10 ns setup time required by the LP5891-Q1 LED driver. Note the variation in the CLB_SOUT_1 output. As described in Section 4.6, this is due to the varying internal delays in updating the serializer output. The glitch can be eliminated by latching the serializer output before passing to CLB_SOUT. However, this method is not used given that the final output has enough setup time to meet the LP5891-Q1 LED driver requirements.

The hold time is at least half of an SCLK cycle.



Figure 4-21. LED Driver Example Setup and Hold Time



5 References

- Configurable Logic Block (CLB) Introduction (video)
- Configurable Logic Block (CLB) architecture (video)
- Configurable Logic Block (CLB) programming tool training (video)
- CLB Tool User's Guide
- TMS320F28388D controlCARD Information Guide
- SysConfig Development Tool for C2000 real-time MCUs
- C2000 Academy Online Training
- Code Composer Studio
- C2000ware
- TMDSCNCD280025C
- TMDSCNCD28388D
- TMDSHSECDOCK
- LAUNCHXL-F280039C
- LP5891Q1EVM

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES "AS IS" AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI's products are provided subject to TI's Terms of Sale or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI's provision of these resources does not expand or otherwise alter TI's applicable warranties or warranty disclaimers for TI products.

TI objects to and rejects any additional or different terms you may have proposed.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265 Copyright © 2023, Texas Instruments Incorporated