Application Report
SPRACL3–August 2019

Designing With the C2000™ Configurable Logic Block (CLB)

Nima Eskandari

ABSTRACT

The C2000 configurable logic block (CLB) is a collection of configurable blocks that interconnect through software to implement custom digital logic functions. In this application report, a simple custom digital logic system is designed and tested. Each configurable block inside the CLB module is examined and set up to implement the desired custom system.

Contents

List of Figures

List of Tables

Trademarks

C2000, LaunchPad, Code Composer Studio are trademarks of Texas Instruments.

1 Introduction

To understand the CLB, this document examines each of its configurable blocks individually and explains how they can be used together. The CLB subsystem contains a number of identical tiles. There are four identical tiles in the F2837xD CLB subsystem, but other devices may contain more or fewer tiles. Each tile has the following:

- 4-input look-up table (LUT4) submodules
- Counter submodules
- Finite State Machine (FSM) submodules
- Output 3-input lookup table (Output LUT) submodule
- High-level Controller (HLC) submodule

A simple 4-state, state machine is designed. Each of the submodules in a CLB tile are used and their capability and example usage are shown.

2 Supplementary Online Information

For more information on the CLB module on a specific C2000 device, see the device-specific data sheet and the corresponding Technical Reference Manual (TRM).

This application report was written using the TMS320F2837xD family of devices. The data sheet and TRM used for this application report are listed below:

- *TMS320F2837xD Dual-core Delfino™ Microcontrollers Datasheet*
- *TMS320F2837xD Dual-core Delfino™ Microcontrollers Technical Reference Manual*

Additional support is provided by the TI E2E™ Community.

3 Design Overview

The design used in this document is a simple 4-state, state machine. The four states are:

- Opened
- Closing
- Closed
- Opening

The design is similar to a simple garage door opening. There are two external inputs used to interact with the state machine. They are:

- BOUNDARY IN0: Button input to open or close the door based on the current state
- BOUNDARY IN1: Sensor input to indicate whether the closing or opening of the door is completed

Figure 1 shows the states of the FSM and how the external inputs are used to transition between these states.

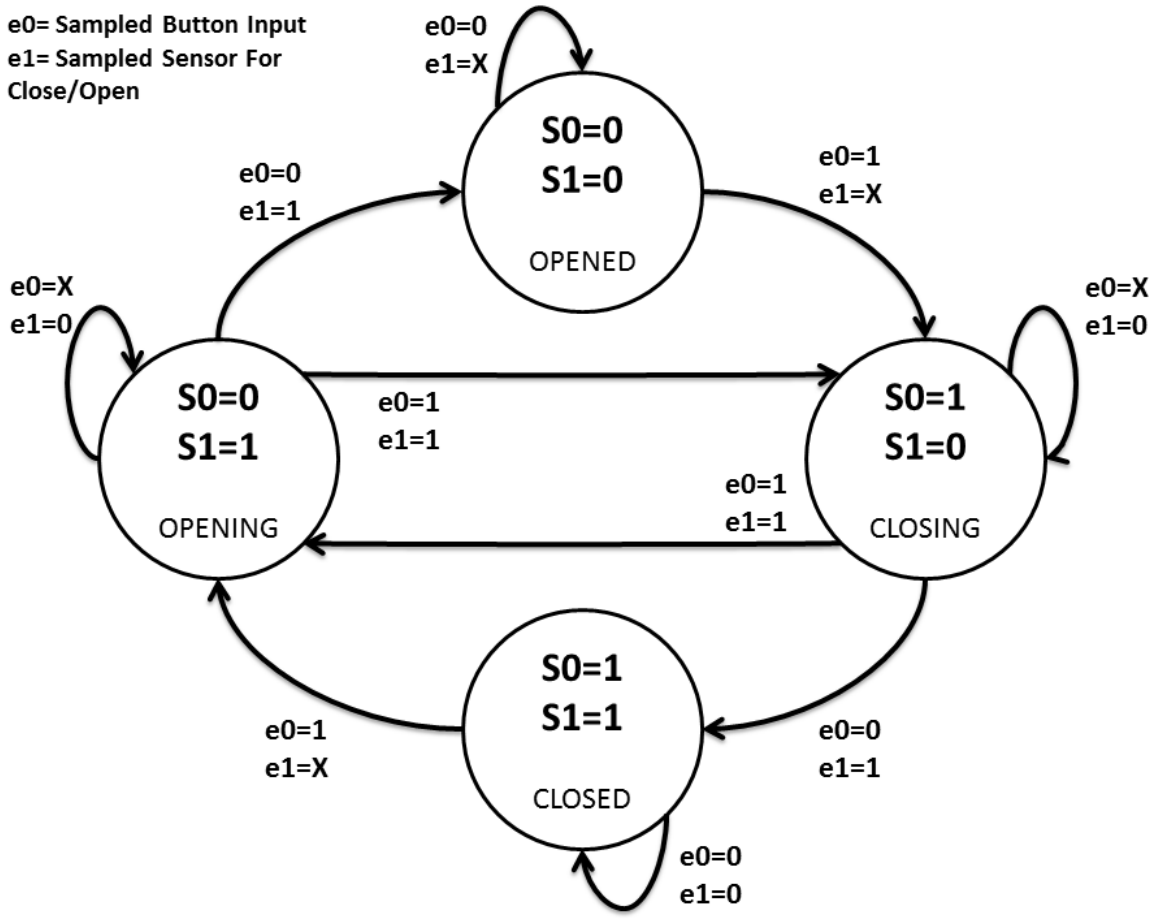


Figure 1. State Machine Diagram

The system transitions from opened or closed states to closing or opening states when the BOUNDARY IN0 (the button input) is HIGH. The system transitions from opening or closing states to opened or closed states when BOUNDARY IN1 (the sensor input) is HIGH. When IN0 and IN1 are HIGH at the same time, the system transitions from closing to opening state OR from opening to closing state. No other input combinations cause a transition between the states of the state machine.

4 Sampling the Inputs

The first step in designing this specific system is sampling the external inputs to make testing the final design easier. Instead of using the raw input from the GPIOs used as BOUNDARY IN0 and IN1, a counter is used to sample BOUNDARY IN0 and IN1 every 50,000 clock cycles. Use an oscilloscope to help view the results of the polling by displaying a transition between closing and opening states. This can be seen in Section 10.

A counter module (COUNTER0) is used alongside LUT0 and LUT1 to generate the sampled input signals, which is then used by the FSM submodule. The requirements for the sampling logic are:

- COUNTER0 is used to generate a signal every 50000 clock cycles (match1_val = 50000)
- The counter is always enabled (mode0 = 1) and counts up (mode1 = 1)
- The counter is reset on MATCH1 event
- LUT0 and LUT1 are used
- The two input signals (BOUNDARY IN0 and IN1) are used with the COUNTER0 MATCH1 event signal to generate the sampled input signals, which are then used by the FSM

Figure 2 and Figure 3 show these requirements and how they are applied in the SysConfig settings of the example project.

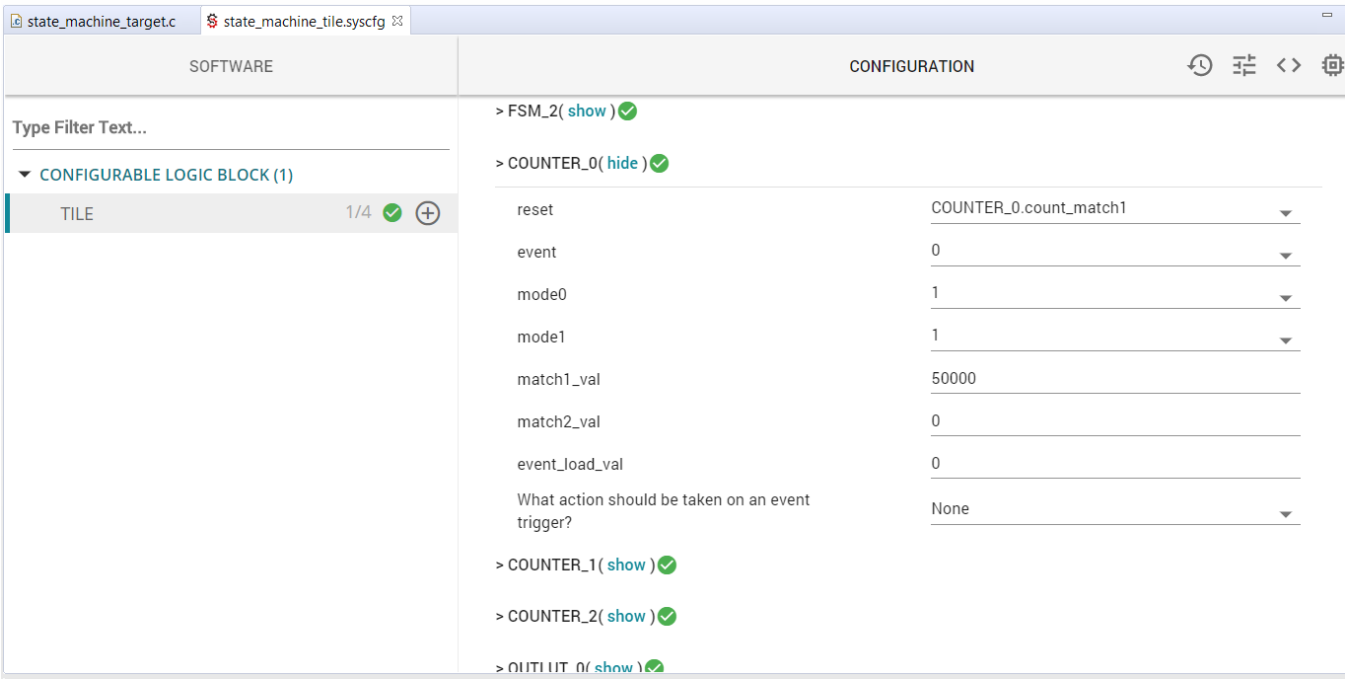


Figure 2. COUNTER0 Configuration

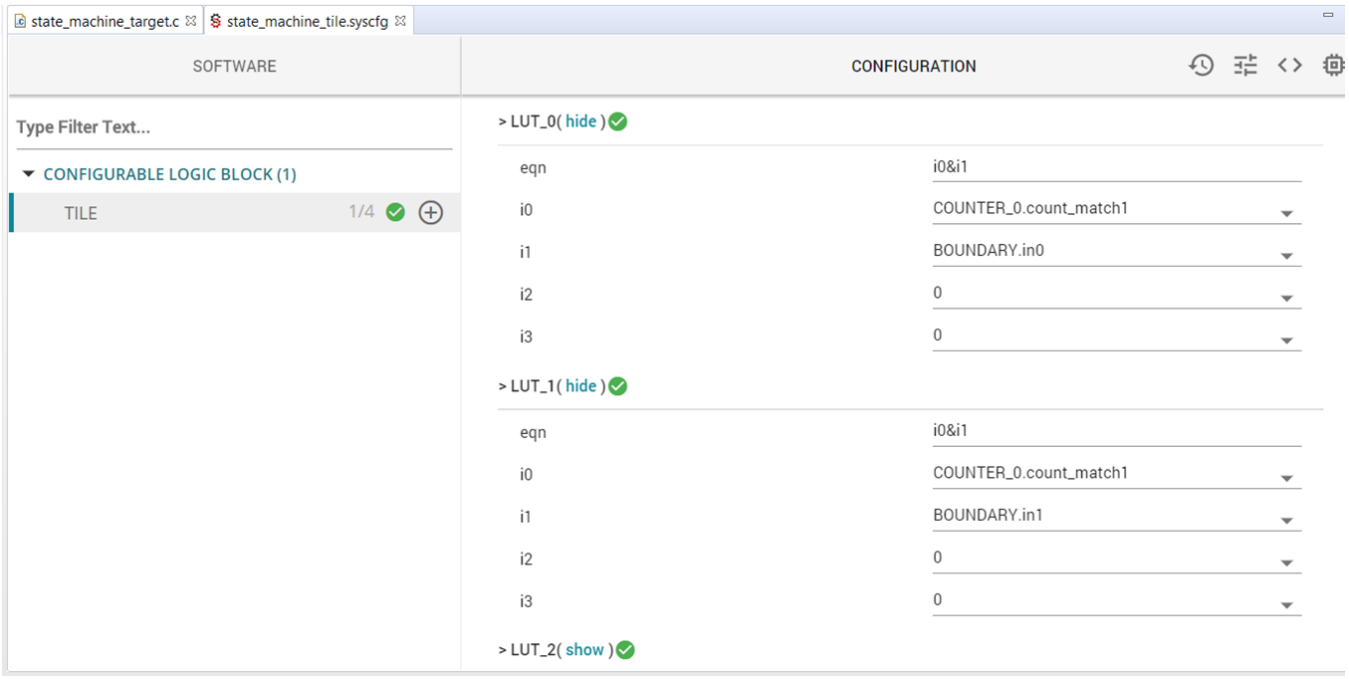


Figure 3. LUT0 and LUT1 Configuration

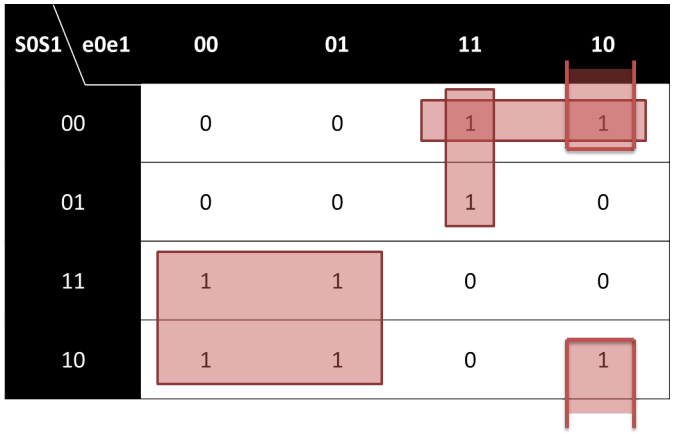
5 Implementing the State Machine in the FSM Submodule

The state machine in Figure 1 is applied using the sampled inputs generated in the previous section. First, create a complete truth table consisting of the inputs, current states, and the next states of the state machine. The FSM submodule also generates an output signal. In this design, the output is set to HIGH when the state of the system is switched to opening or closing. This output is used later in the design (for gating PWM signals). Table 1 shows the complete truth table of the design.

Table 1. FSM Truth Table

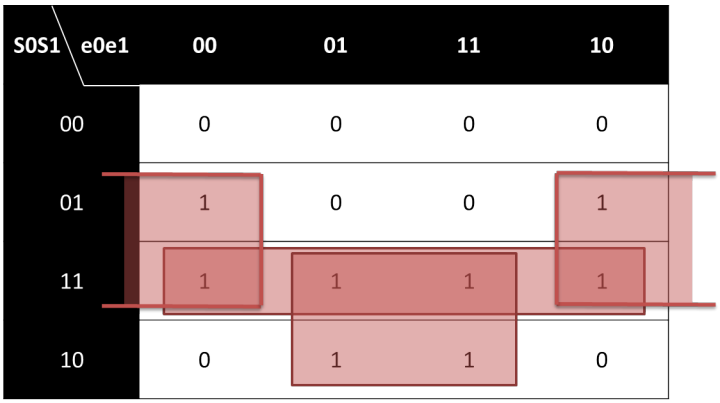
s0	s1	e0	e1	s0 NEXT	s1 NEXT	OUTPUT
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	1	0	1
0	0	1	1	1	1	1
0	1	0	0	0	1	0
0	1	0	1	0	0	0
0	1	1	0	0	1	0
0	1	1	1	1	0	1
1	0	0	0	1	0	0
1	0	0	1	1	1	0
1	0	1	0	1	0	0
1	0	1	1	0	1	1
1	1	0	0	1	1	0
1	1	0	1	1	1	0
1	1	1	0	0	1	1
1	1	1	1	0	1	1

Using the FSM truth table, the Karnaugh map is created for the s0 state, s1 state, and the output of the FSM. The Karnaugh map helps find the FSM equations for s0, s1, and the output.



EQ S0 = (!e0&s0) | (e0&e1&!s0) | (!s0&!s1&e0) | (e0&!e1&!s1)

Figure 4. FSM1 S0 K-Map



EQ S1 = (s1&!e1) | (s0&s1) | (s0&e1)

Figure 5. FSM1 S1 K-Map

S0S1 \ e0e1	00	01	11	10
00	0	0	1	1
01	0	0	1	0
11	0	0	1	1
10	0	0	1	0

EQ OUTPUT = (e0&e1) | (!s0&!s1&e0) | (s0&s1&e0)

Figure 6. FSM1 Output K-Map

The F28379D LaunchPad (used for testing this design) displays s0 and s1 of the FSM1 on the LEDs available on the LaunchPad™. OUTLUT4 and OUTLUT5 export these signals from FSM1 to the OUTPUT X-BAR which is then selected to drive GPIOs. The LEDs on the LaunchPad turn ON when the output of GPIO34 and GPIO31 are pulled LOW. s0 and s1 of the FSM1 are therefore inverted before getting export out of the CLB through OUTLUT4 and OUTLUT5. This inversion turns the LEDs ON when s0 and s1 are HIGH. Figure 7 shows the SysConfig configuration for OUTLUT4 and OUTLUT5.

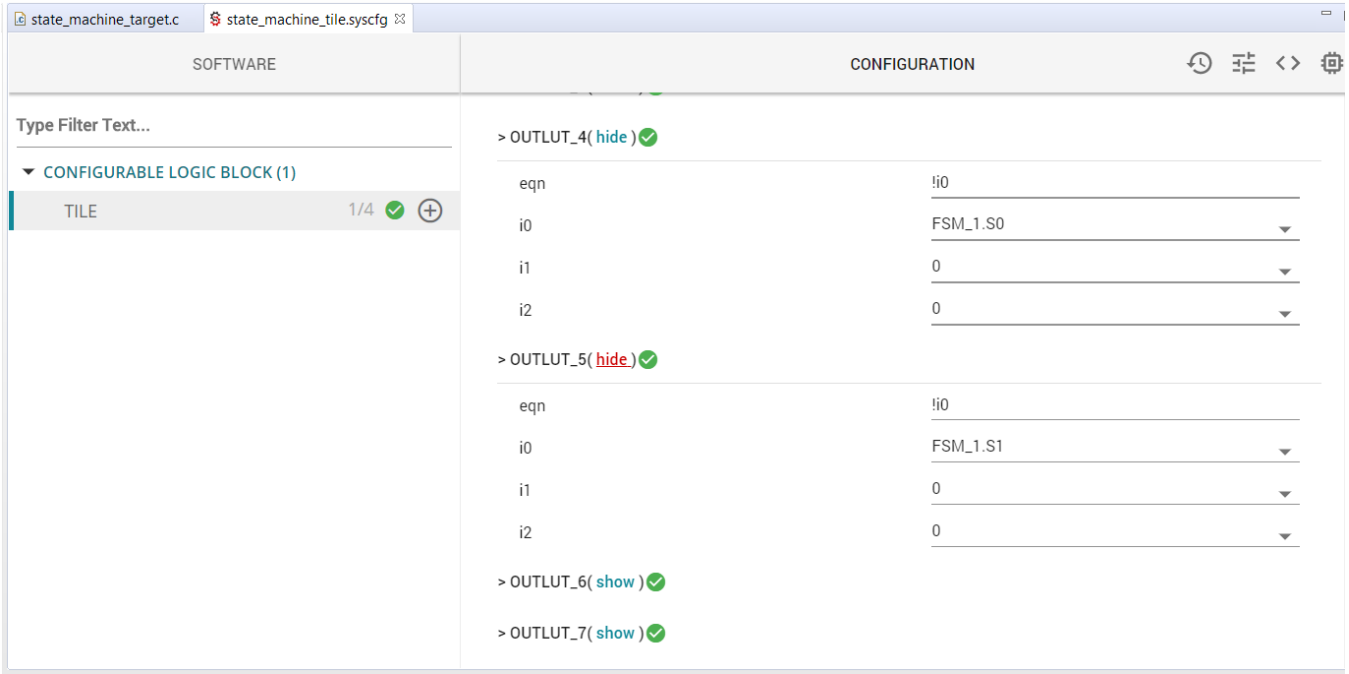


Figure 7. OUTLUT4 and OUTLUT5 Configurations

6 Generating PWM Signals

The CLB can use its configurable blocks to generate PWM waveforms. Assume it is required to generate PWM signals when the state of the system is opening or closing. For this system, the two PWM signals from the CLB tile are:

- Opening state: First PWM signal is active with a modifiable period and duty cycle, while the second PWM signal is held LOW.
- Closing state: First PWM signal is held LOW, while the second PWM signal is active with a modifiable period and duty cycle.

FSM0, FSM2, and COUNTER1 generate the PWM signals. The counter submodule acts as the main part of the PWM while FSM0 and FSM2 complete the rest of the logic. These roles are required to generate the PWM waveforms.

COUNTER1 is set up to operate in count-down mode, generating signals when the counter reaches ZERO or a specified match value. The items below summarize the requirements for COUNTER1:

- Count down from load_val (mode1 = 0).
- At the counter=ZERO event, load the load_val into the counter.
- The period of the PWM signal is set by the load_val register of COUNTER1.
- The event generated at counter=match1_val is used in the next steps.
- The duty cycle of the PWM is set by the match1_val register of COUNTER1.
- The enable signal for the counter is mode0. This signal should be active only when the system is in opening or closing states (this signal, which is an input to the counter, is created in the next steps).
- Reset the timer when a transition of states to opening or closing states is detected (this signal is already available from FSM1 OUTPUT).

The time-based counter of the PWM is now set up. Next, create the signal that is active only when the system is in opening or closing states. To create this signal, FSM2 is used as a 3-output LUT instead of a Finite State Machine. FSM2 is set up to fulfill the requirements below:

- S0: Active only when the opening state is active
- S1: Active only when the closing state is active
- OUTPUT: Active when either closing or opening states are active

The output from FSM2 is used as the input to COUNTER1 mode0. Mode0 of COUNTER1 is only active when the system is in either closing or opening states.

Figure 8 and Figure 9 show the SysConfig configurations for FSM2 and COUNTER1.

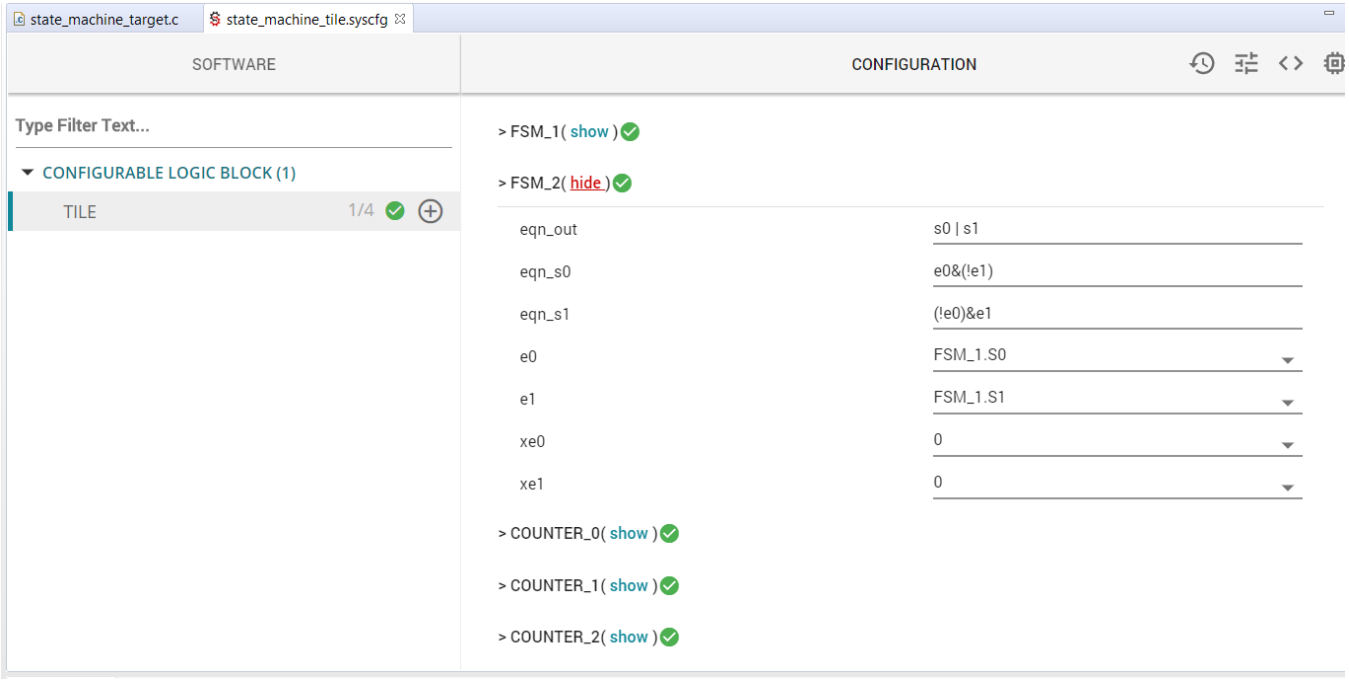


Figure 8. FSM2 Configuration

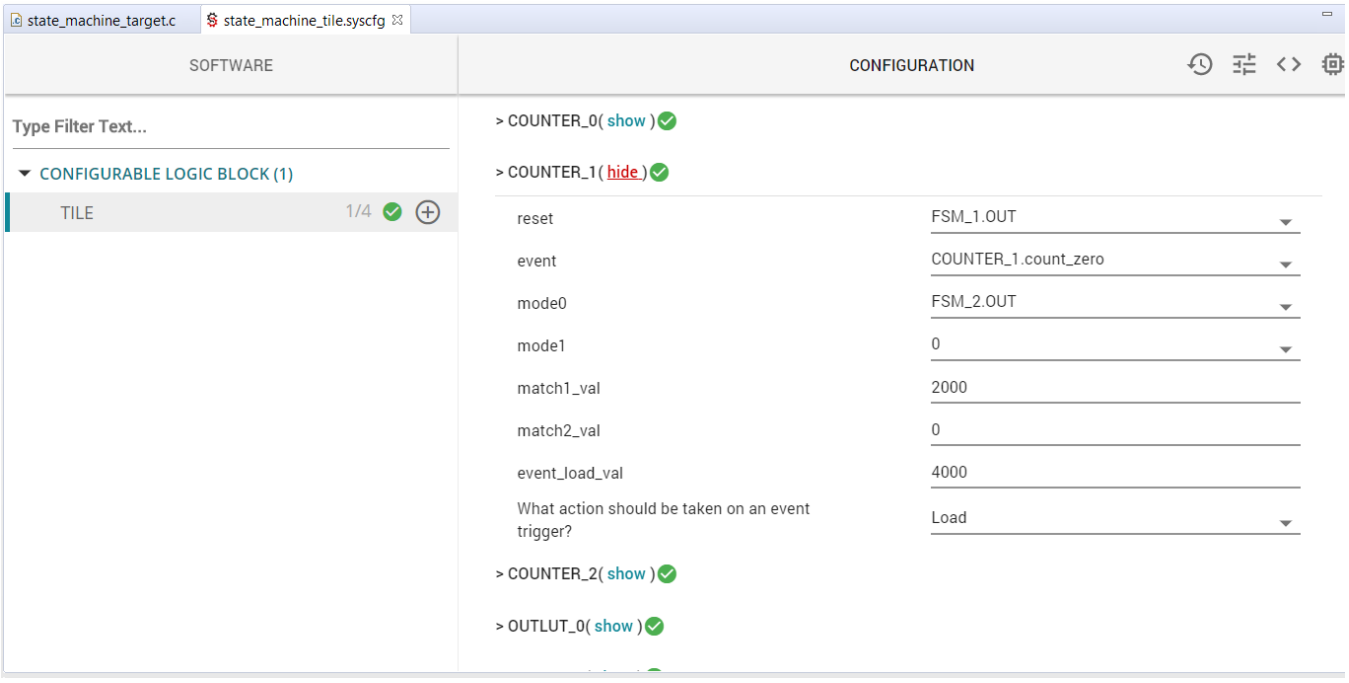


Figure 9. COUNTER1 Configuration

The final step to create the PWM signals is to use the FSM module to generate the PWM waveforms based on the events and signals generated by COUNTER1 and FSM2. The PWM signal is generated by using FSM0 s0. s0 must clear to LOW on the counter=ZERO event of the COUNTER1. s0 must set to HIGH on the counter=match1_val event of the COUNTER1.

Figure 10 shows the SysConfig configuration of FSM0.

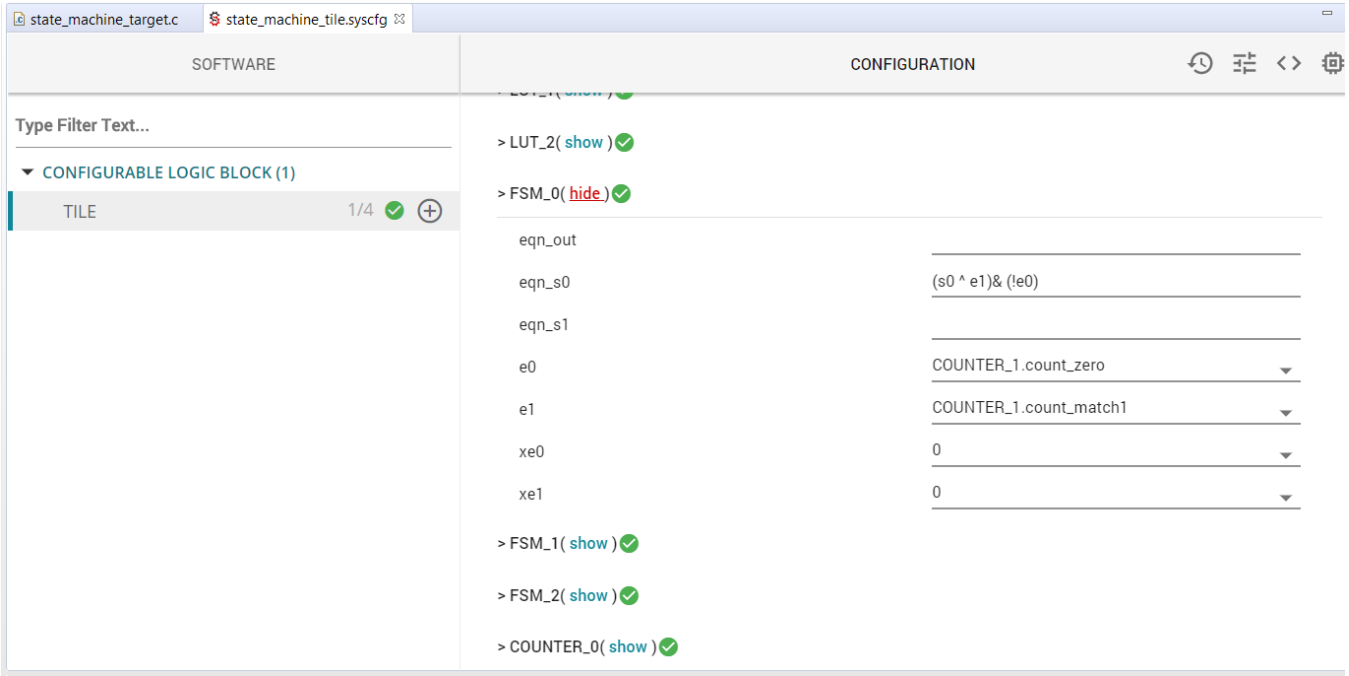


Figure 10. FSM0 Configuration

Finally, the PWM signal is split into two signals: one for the closing state and one for the opening state. OUTLUT0 and OUTLUT2 export the two signals. When the system is in the opening state, OUTLUT0 outputs the PWM signal while OUTLUT2 stays LOW. When the system is in the closing state, OUTLUT2 outputs the PWM signal while OUTLUT0 stays LOW. Figure 11 show the SysConfig configurations for OUTLUT0 and OUTLUT2.

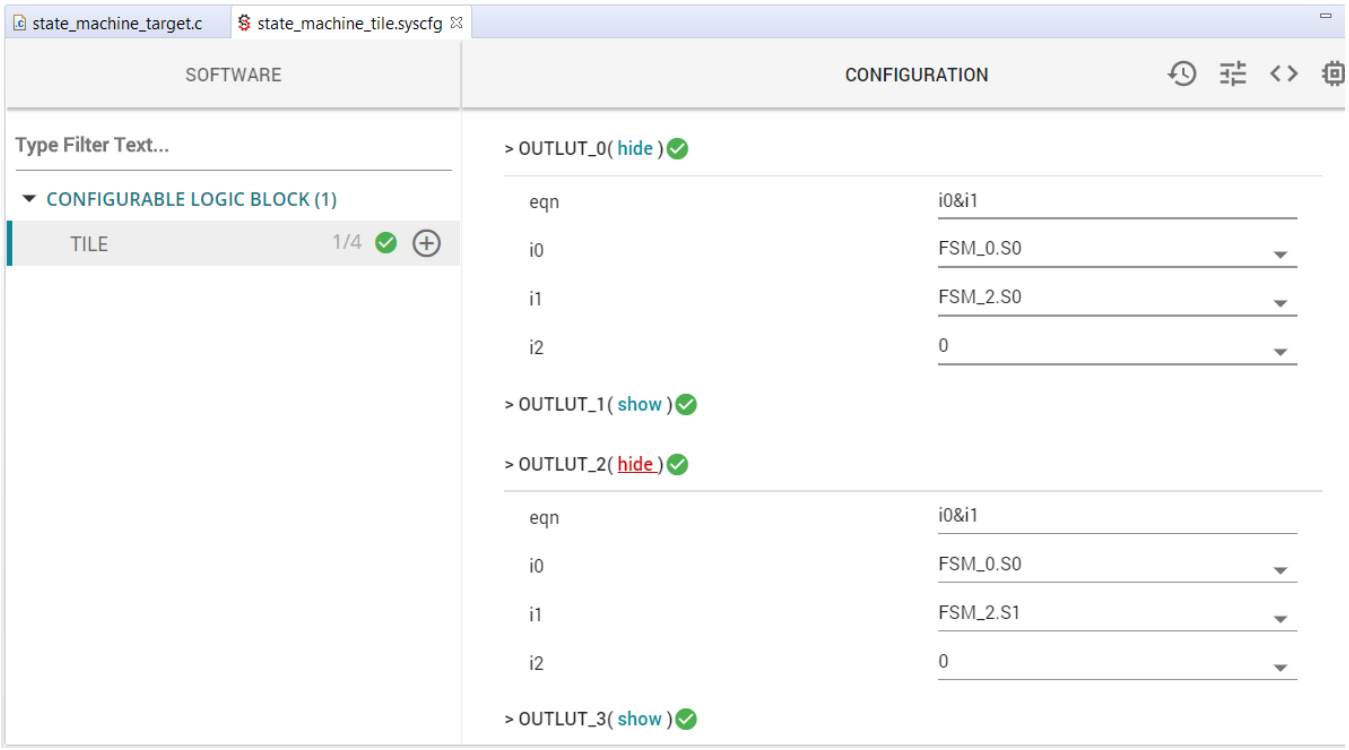


Figure 11. OUTLUT0 and OUTLUT2 Configurations

In the next step, the PWM period and duty cycle can be updated at any runtime by using the C28x core and the HLC submodule.

7 Modifying PWM Period and Duty

It is required to update the load_val and match1_val of the COUNTER1 to modify the period and duty cycle of the PWM generated by the CLB tile. The C28x core has access to both of these registers in COUNTER1 and can updates them as desired. However, it would better if the load_val and match1_val is updated at counter=ZERO event of COUNTER1 to eliminate any unpredictable behavior. In this design, the update of the counter values occur on the next counter=ZERO after it is requested by the C28x core. To load the counter values, the HLC submodule and the PUSH-PULL interface are used. The PUSH-PULL interface is a 4-deep FIFO that the C28x core and the HLC submodule use for data transfer. The C28x core writes the new period and duty cycle values to the PUSH-PULL FIFO. The HLC reads these values and updates the COUNTER1 registers at the next counter=ZERO event. The C28x also must signal to the HLC that new values are available in the PUSH-PULL FIFO. The signal is created by using BIT2 (0b00000100) of the GPREG register (which is a C28x accessible register). The bits of the GPREG register connect to the BOUNDARY input of the CLB tile. HLC performs its tasks when an input event is activated.

In this design, EVENT0 is used to signal the HLC to read the PUSH-PULL FIFO and update the PWM values. If a rising edge is detected on EVENT0 of the HLC, the specified instructions are executed. Since the C28x core must signal to the HLC to start updating the PWM values on the next COUNTER1 counter=ZERO event, the EVENT0 input from the HLC must be the resulting signal from COUNTER1 counter=ZERO "ANDed" with BOUNDARY IN2. LUT2 is used to logically AND these two signals. The output of LUT2 is then connected to the EVENT0 input of the HLC.

Figure 12 shows the SysConfig configuration of LUT2.

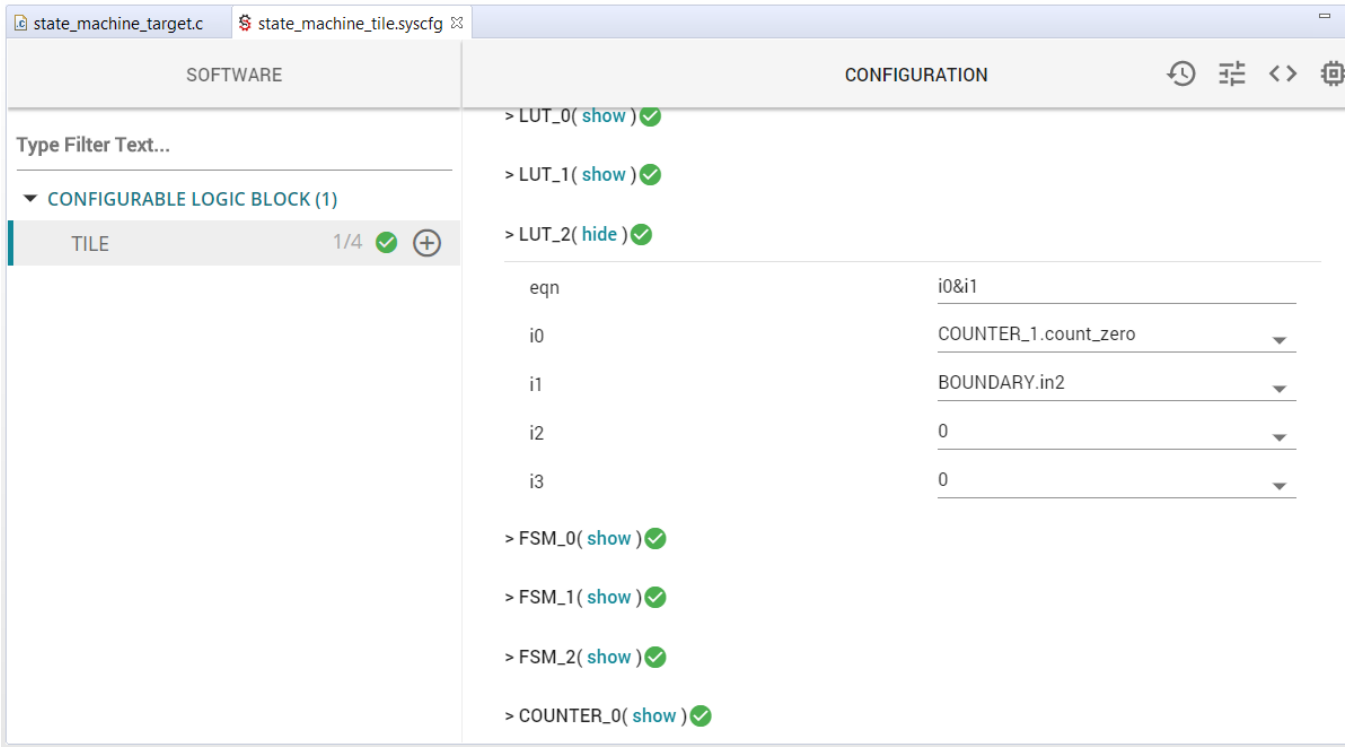


Figure 12. LUT2 Configuration

Next, define the instructions of the HLC. The C28x core updates the FIFO with the new period value, then with the new duty value. The HLC must PULL these values from the FIFO in the same order. First, the HLC must PULL the period value into the active counter register of COUNTER1 (this value is also written to the load_val, discussed in the next step). The next value PULLED from the FIFO must be written to the match1_val of the COUNTER1. However, the PULL instruction cannot pull values into the match1_val of the counter submodule. The only instruction that can write to the match values of the counters are MOV_T1 and MOV_T2. These instructions access match1_val and match2_val, respectively. The value in the FIFO is pulled into the R1 register of the HLC using a PULL instruction. Afterward, the R1 value updates the match1_val of the COUNTER1 using the MOV_T1 instruction.

The INTR instruction is the final instruction executed by the HLC. The INTR instruction generates an interrupt on the C28x core, which signals the C28x core that the update has taken place. The FIFO is now empty and may be used again. The GPREG BIT2 is also cleared by the C28x core so that the next counter=ZERO does not generate another HLC event (causing the FIFO to be read when it is empty and creating an underflow scenario). The INTR instruction requires one argument called the "interrupt tag". This tag is read on the C28x core side to see what caused the interrupt. The HLC can generate multiple interrupts at different events in the tile; however, in this design only one interrupt is used. The interrupt tag used with the INTR instruction is "1" and the C28x core ensures the interrupt tag is "1" when handling the interrupt service routine.

Figure 13 shows the SysConfig configuration of the HLC submode.

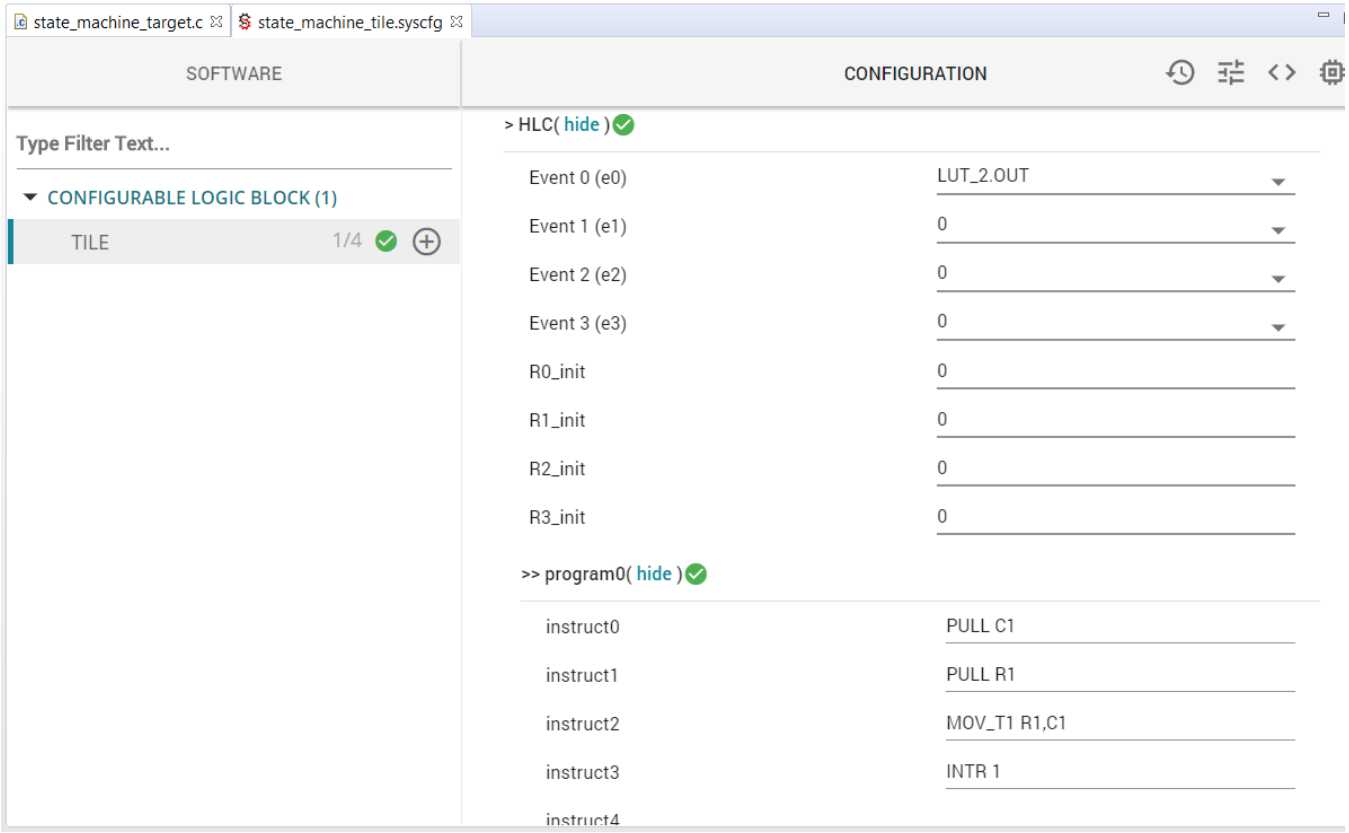


Figure 13. HLC Configuration

The final item left open in this part of the design is updating the load_val of COUNTER1. The HLC does not have access to this register, so the C28x must update this value before it sets the GPREG BIT2. If the load_val is not updated, the period is not updated. The following code snippet shows how the C28x core can update the PWM period and duty cycle, and clear the GPREG after the interrupt is received.

```
void updateClbPwm(uint32_t period, uint32_t duty)
{
if (canUpdate)
   {
       canUpdate = 0;
       CLB_writeInterface(CLB1_BASE, CLB_ADDR_COUNTER_1_LOAD, period);
       HWREG(CLB1_BASE + CLB_LOGICCTL + CLB_O_BUF_PTR) = 0U;
       HWREG(CLB1_BASE + CLB_DATAEXCH + CLB_O_PULL(0)) = period;
       HWREG(CLB1_BASE + CLB_DATAEXCH + CLB_O_PULL(1)) = duty;
       CLB_setGPREG(CLB1_BASE, 1 << TRIGGER_PWM_UPDATE_SHIFT);
   }
}

__interrupt void clb1ISR(void)
{
    uint16_t tag = CLB_getInterruptTag(CLB1_BASE);
    if (tag == UPDATE_PWM_COMPLETED_TAG)
    {
        canUpdate = 1;
        CLB_setGPREG(CLB1_BASE, 0);
    }
    CLB_clearInterruptTag(CLB1_BASE);
    Interrupt_clearACKGroup(INTERRUPT_ACK_GROUP5);
}
```

Section 9 describes how the XBAR modules are used to import and export signals in and out of the CLB tiles.

8 Completed Design

Figure 14 shows the complete block diagram of the design. The connection between the submodules allows you to visualize the complete picture of what the CLB tile is designed to accomplish.

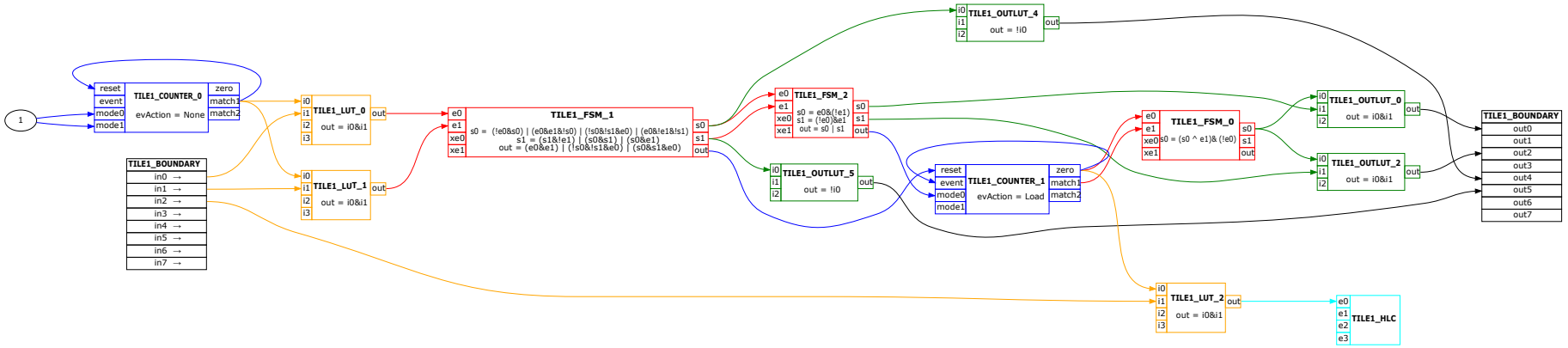


Figure 14. Completed Design Block Diagrams

Be careful when choosing which submodule to use. For example, the design above cannot use FSM0 instead of FSM1 because hardware limitations does not allow the output of LUT1 to be selected as an input to the e1 of the FSM0. This is an example of an invalid configuration. *TMS320F2837xD Dual-core Delfino™ Microcontrollers Technical Reference Manual* shows the complete list of these invalid setups. Some sets of inputs are disconnected in hardware to avoid logic loops. Figure 15 shows the warning the CLB SysConfig tool generates to notify the user of this design error.

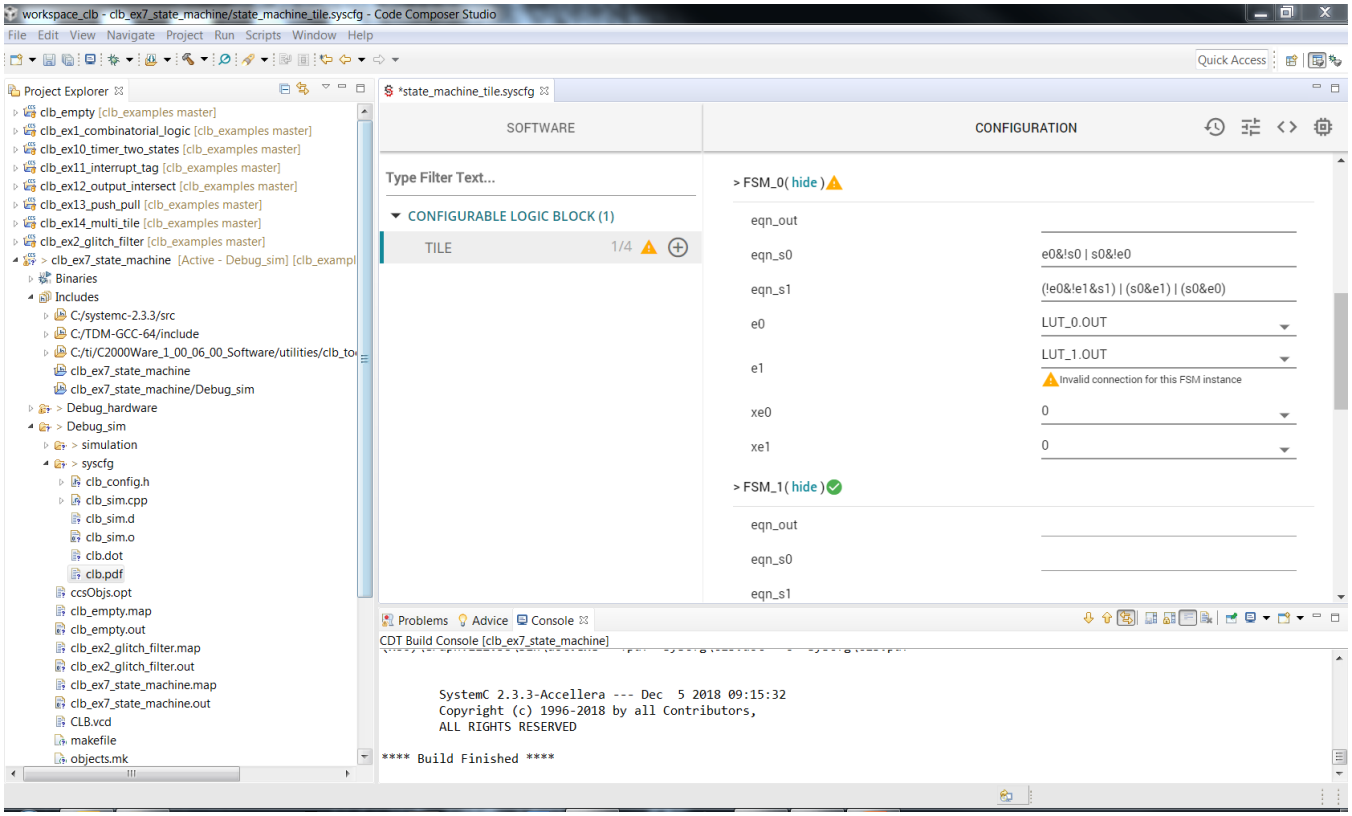


Figure 15. Invalid Connection for This Instance

9 Input X-BAR, Output X-BAR, and CLB X-BAR

The X-BAR module can be used to import external signals into the CLB or export signals out of the CLB. In this section, the X-BAR modules (INPUT X-BAR, OUTPUT X-BAR, and CLB X-BAR) are configured for the CLB tile designed in this application report.

9.1 Using X-BAR to Import Signals to the CLB Tiles

To import signals into the CLB (BOUNDARY INz, where z is any number between 0–7) from the GPIOs, you must configure the INPUT X-BAR and CLB X-BAR. The steps required to import signals from GPIOs to the CLB are:

1. Configure the GPIO as usual:
   a. Set the direction: INPUT/OUTPUT.
   b. Enable/disable the PULL-UP.
   c. Set other GPIO configurations.
2. Use the INPUT X-BAR (for example, INPUTx, where x is any number from 1 to the maximum number of INPUTs) and select the GPIO required.
3. Inside the CLB module, select GLOBAL input for BOUNDARY INz instead of the LOCAL input.
4. Use the CLB X-BAR to select the INPUTx of the INPUT X-BAR as AUXSIGy (where y is any number between 0–7).

5. Select AUXSIGy as GLOBAL input for BOUNDARY INz.
6. Disable the GPREG input and enable the external input (GLOBAL input = AUXSIGy input = INPUTx input = GPIO input).

Figure 16 shows how the CLB BOUNDARY inputs are selected from GLOBAL inputs, LOCAL inputs, or GPREG bits.

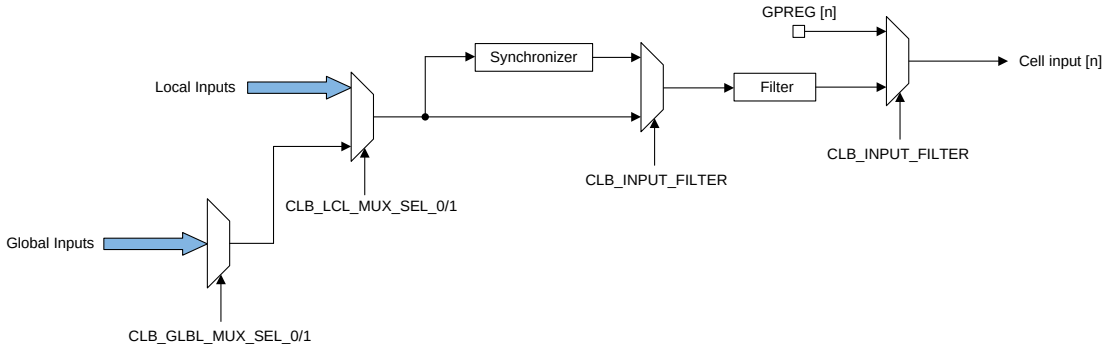


Figure 16. CLB BOUNDARY Input Multiplexing

The following code snippets show how the preceding steps are accomplished using the device driver library.

```
//
// Configure GPIO4 for Button
//
GPIO_setPinConfig(GPIO_4_GPIO4);
GPIO_setDirectionMode(4, GPIO_DIR_MODE_IN);
GPIO_setPadConfig(4, GPIO_PIN_TYPE_PULLUP);
//
// Configure Input-XBAR INPUT1 to GPIO4
//
XBAR_setInputPin(XBAR_INPUT1, 4);
//
// Configure CLB-XBAR AUXSIG0 as INPUT1
//
XBAR_setCLBMuxConfig(XBAR_AUXSIG0, XBAR_CLB_MUX01_INPUTXBAR1);
XBAR_enableCLBMux(XBAR_AUXSIG0, XBAR_MUX01);

CLB_configLocalInputMux(CLB1_BASE, CLB_IN0, CLB_LOCAL_IN_MUX_GLOBAL_IN);
CLB_configGlobalInputMux(CLB1_BASE, CLB_IN0, CLB_GLOBAL_IN_MUX_CLB_AUXSIG0);
CLB_configGPInputMux(CLB1_BASE, CLB_IN0, CLB_GP_IN_MUX_EXTERNAL);
```

9.2 Using X-BAR to Export Signals from the CLB Tiles

To export signals from the CLB (BOUNDARY OUTx) to the GPIOs, use the OUTPUT X-BAR for OUT4 and OUT5 while OUT0–3 and OUT6–7 intercept specific peripheral outputs at the GPIO.

If OUT4 and OUT5 are used, use the OUTPUT X-BAR with the following steps:

1. Configure the GPIO as usual.
   a. Set the direction: INPUT/OUTPUT.
   b. Enable/disable the PULL-UP.
   c. Set other GPIO configurations.

2. Configure the PINMUX to use the OUTPUT X-BAR.

The following code snippets show how the preceding steps are accomplished using the device driver library.

```
//
// Configure GPIO31 for OUTPUTXBAR8
//
GPIO_setPinConfig(GPIO_31_OUTPUTXBAR8);
GPIO_setDirectionMode(31, GPIO_DIR_MODE_OUT);
GPIO_setPadConfig(31, GPIO_PIN_TYPE_STD);
//
// Configure OUTPUT-XBAR OUTPUT8 as CLB1_OUT4
//
XBAR_setOutputMuxConfig(XBAR_OUTPUT8, XBAR_OUT_MUX01_CLB1_OUT4);
XBAR_enableOutputMux(XBAR_OUTPUT8, XBAR_MUX01);
//
// Configure GPIO34 for OUTPUTXBAR1
//
GPIO_setPinConfig(GPIO_34_OUTPUTXBAR1);
GPIO_setDirectionMode(34, GPIO_DIR_MODE_OUT);
GPIO_setPadConfig(34, GPIO_PIN_TYPE_STD);
//
// Configure OUTPUT-XBAR OUTPUT1 as CLB1_OUT5
//
XBAR_setOutputMuxConfig(XBAR_OUTPUT1, XBAR_OUT_MUX03_CLB1_OUT5);
XBAR_enableOutputMux(XBAR_OUTPUT1, XBAR_MUX03);
```

If you have peripheral output interception (tile OUT0–3 and OUT6–7):

1. Configure the PINMUX as usual for that specific peripheral.
    a. Example: Configure GPIO0 for EPWM1A.
2. Enable the output of the CLB OUT0–3 and OUT6–7 using the OUT_EN register.

The output of the GPIO is the CLB OUT instead of the peripheral output. The Peripheral Signal Multiplexer table in the TRM mentions which CLB OUT corresponds to which peripheral output. The following code snippets show how the preceding steps are accomplished using the device driver library.

```
//
// Configure GPIO0 for EPWM1A which will be overriden by CLB0_OUT0
//
GPIO_setPinConfig(GPIO_0_EPWM1A);
GPIO_setDirectionMode(0, GPIO_DIR_MODE_OUT);
GPIO_setPadConfig(0, GPIO_PIN_TYPE_STD);
//
// Configure GPIO1 for EPWM1B which will be overriden by CLB0_OUT1
//
GPIO_setPinConfig(GPIO_1_EPWM1B);
GPIO_setDirectionMode(1, GPIO_DIR_MODE_OUT);
GPIO_setPadConfig(1, GPIO_PIN_TYPE_STD);
```

Finally, the output of the CLB tile must be enabled. The following code snippet accomplishes this task using the device driver library.

```
CLB_setOutputEnableMask(CLB1_BASE, 1 << 0 | 1 << 2);
```

10 Running the Example Project

This section discusses the instructions on how to test the Code Composer Studio™ example project provided for the system designed here. The example is tested on the TMS320F28379D LaunchPad.

10.1 Setup and Connections

Figure 17 shows the pins used for BOUNDARY IN0 and IN1, the PWM outputs for the closing and opening states, and the LEDs used to show the current state of the FSM1 submodule.

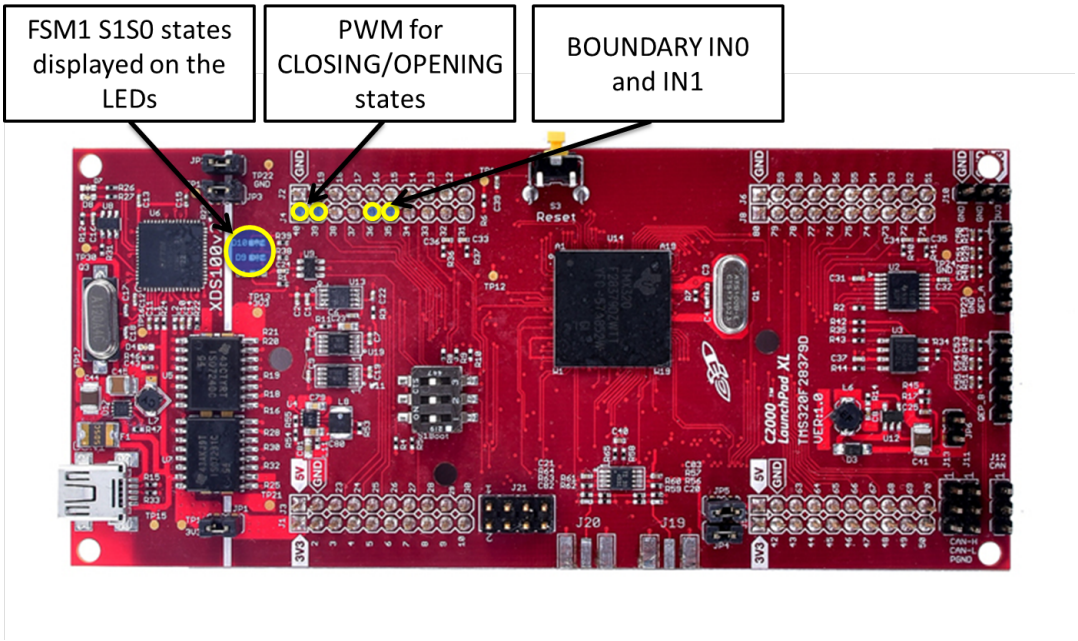


Figure 17. TSM320F28379D LaunchPad Connections

10.2 Testing States And Transitions

Before running the code, pull both BOUNDARY IN0 and IN1 to GND using a jumper wire. The internal PULLUP is enabled on these GPIOs. Removing the connection to GND pulls the pins to HIGH. Connect oscilloscope probes to both GPIO0 and GPIO1 to view the PWM signals when the system is in closing or opening states. The following sections show how to transition between the steps and see the results of these changes on the LaunchPad.

10.2.1 Step 1

Follow these steps to see the transition between the opened state to closing state:

1. Run the core. Both LEDs are OFF and no PWM signal is shown on the scope.
2. Disconnect BOUNDARY IN0 from GND.
   a. The state machine transitions from opened to closing. The PWM signal displays on the corresponding GPIO for closing. The state LEDs are in OFF-ON state to show S1 = 0, S0 = 1.
3. Connect BOUNDARY IN0 to GND again. Nothing occurs from this change.

10.2.2 Step 2

Follow these steps to see the transition between the closing state to closed state:

1. Disconnect BOUNDARY IN1 from GND.
   a. The state machine transitions from closing to closed. The PWM signal stops. The state LEDs are in ON-ON state to show S1 = 1, S0 = 1.
2. Connect BOUNDARY IN1 to GND again. Nothing occurs from this change.

10.2.3 Step 3

Follow these steps to see the transition between the closed state to opening state:

1. Disconnect BOUNDARY IN0 from GND.
   a. The state machine transitions from closed to opening. The PWM signal shows on the corresponding GPIO for opening. The state LEDs are in ON-OFF state to show S1 = 1, S0 = 0.
2. Connect BOUNDARY IN0 to GND again. Nothing occurs from this change.

10.2.4 Step 4

Follow these steps to see the transition between the opening state to opened state:

1. Disconnect BOUNDARY IN1 from GND.
   a. The state machine transitions from opening to opened. The PWM signal stops. The state LEDs are in OFF-OFF state to show S1 = 0, S0 = 0.
2. Connect BOUNDARY IN1 to GND again. Nothing occurs from this change.

10.2.5 Step 5

In this last test, inputs IN0 and IN1 are disconnected from GND, causing the state of the system to jump between closing and opening, continuously. This jumping toggles the states and the PWM signals with the frequency of the sampling counter.

10.3 Testing PWM Period and Duty Cycle

The last thing to test is the ability to change the PWM period and duty cycle at run time. Follow these steps to test this feature:

1. Run the code and open the expressions window.
2. Add “clbPwmUpdateNow”, “clbPwmDuty”, and “clbPwmPeriod".
3. Enable Auto Refresh.

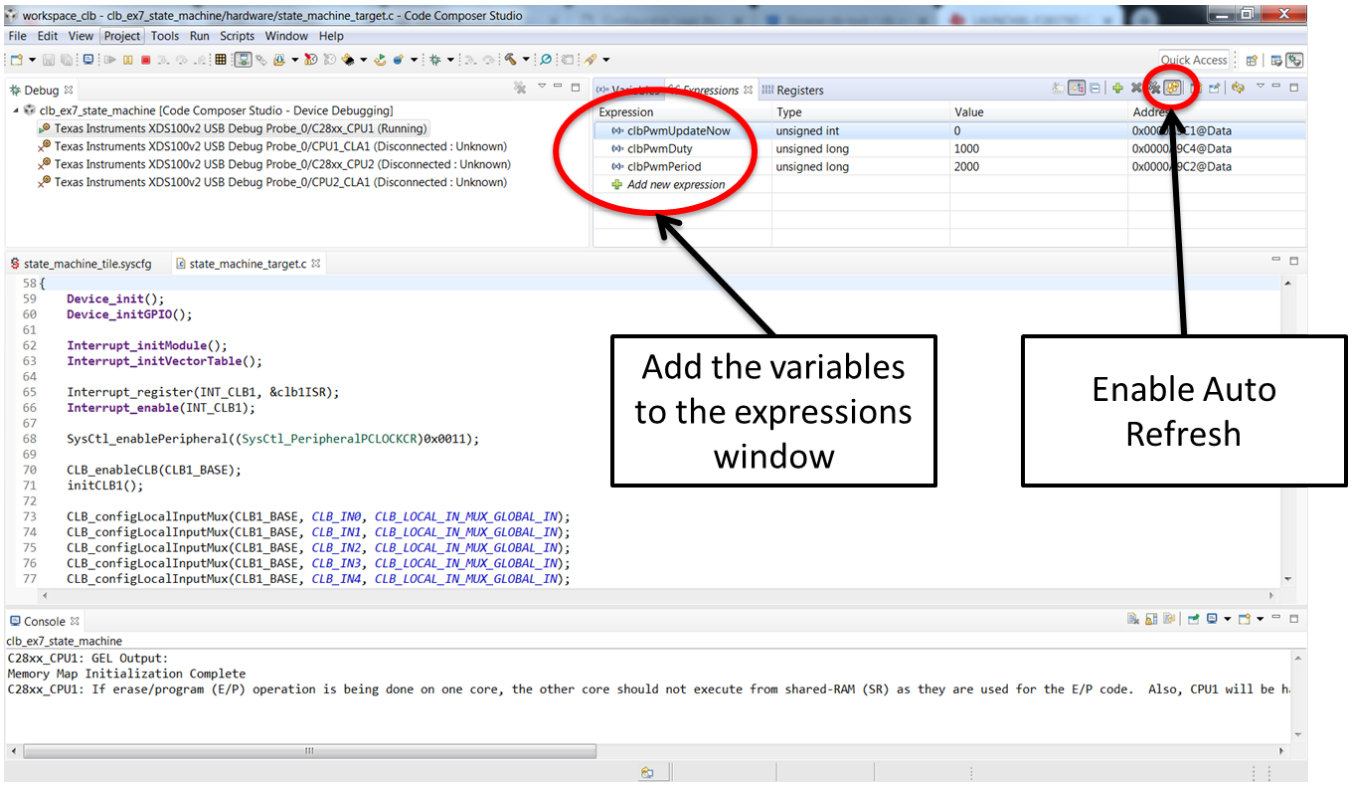


Figure 18. Changing PWM Period and Duty Using the Expressions Window

4. Click on the value of the clbPwmDuty and clbPwmPeriod and update them to new values.
5. Click on the value of clbPwmUpdateNow and write the value “1”.
6. The PWM updates and is seen on the oscilloscope.

Figure 19 and Figure 20 show an example of setting the PWM duty cycle to 1000 and the period to 2000.

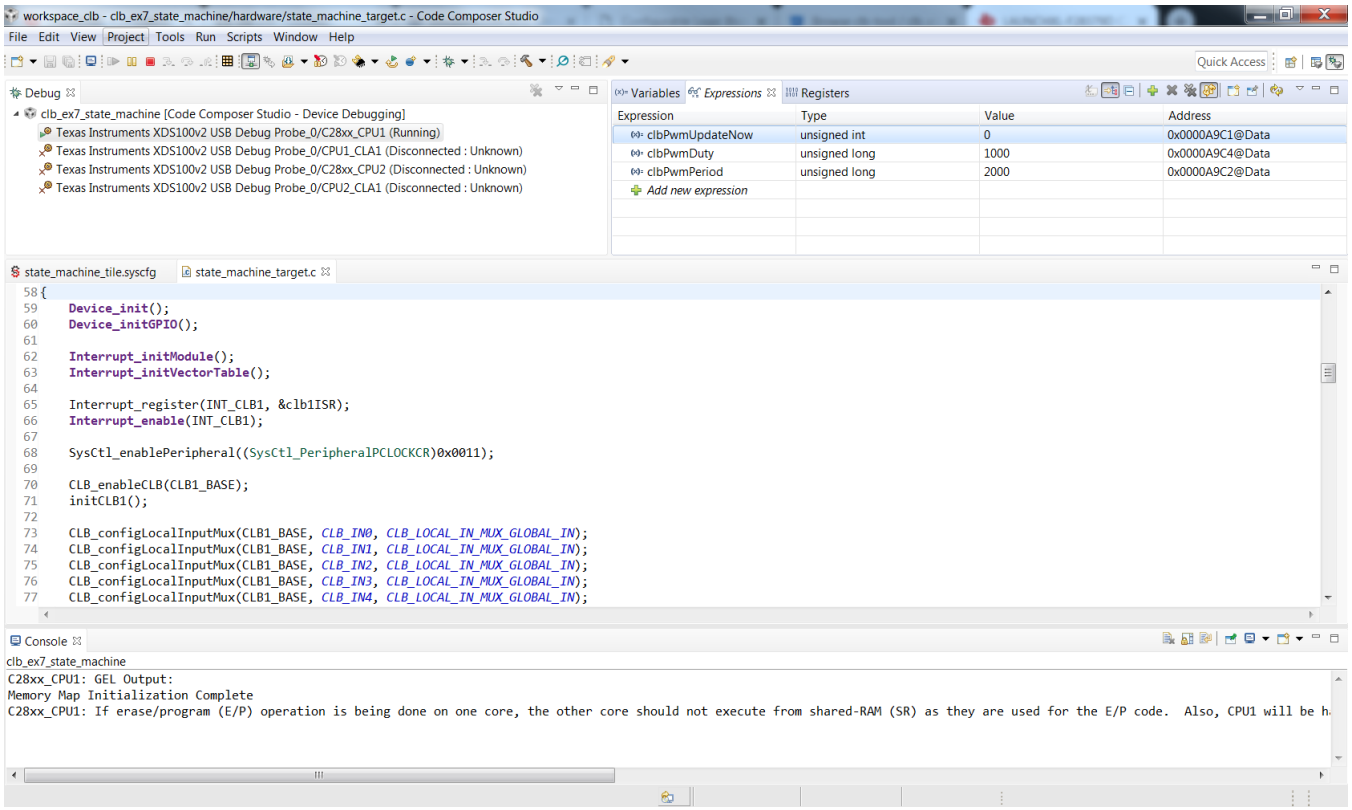


Figure 19. PWM Duty 1000, Period 2000

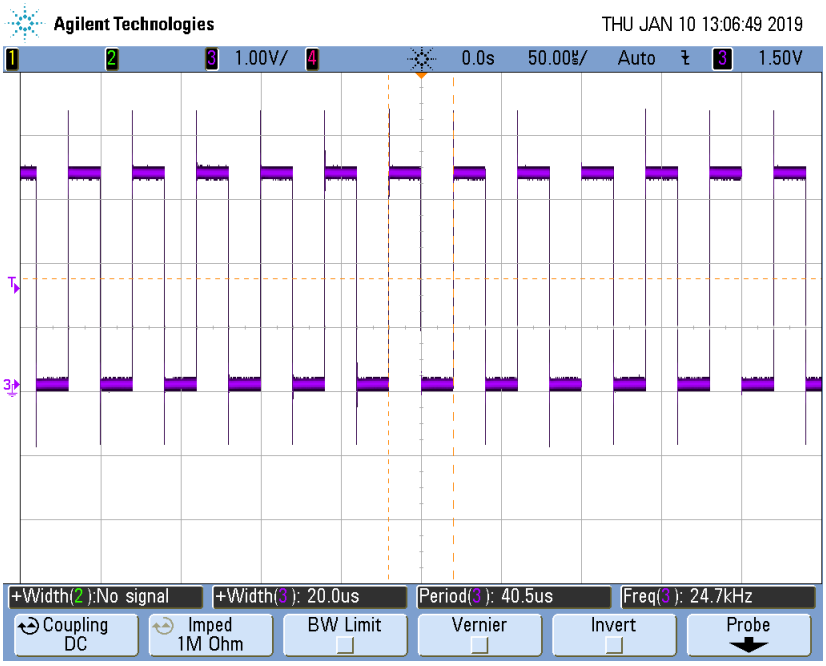


Figure 20. Oscilloscope - PWM Duty 1000, Period 2000

Figure 21 and Figure 22 show an example of setting the PWM duty cycle to 3000 and the period to 4000.

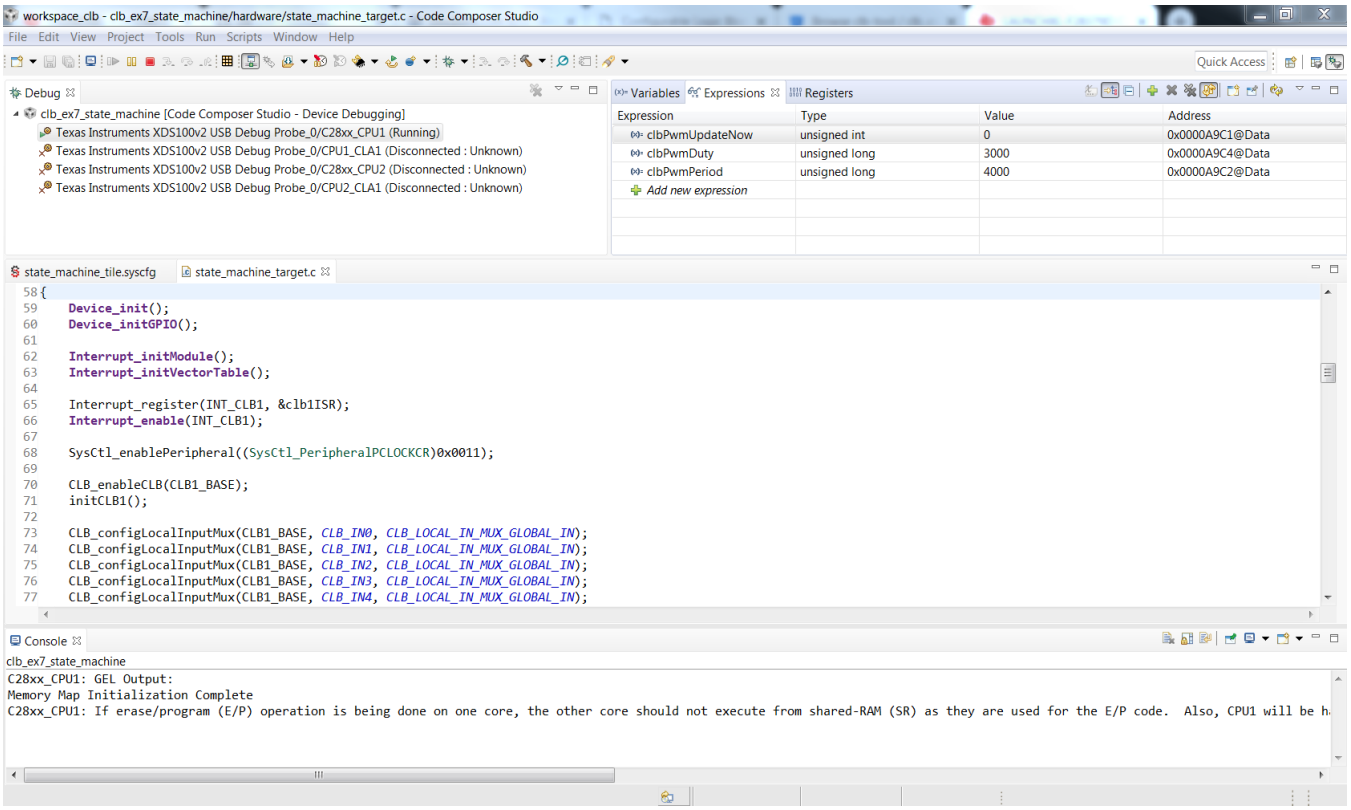


Figure 21. PWM Duty 3000, Period 4000

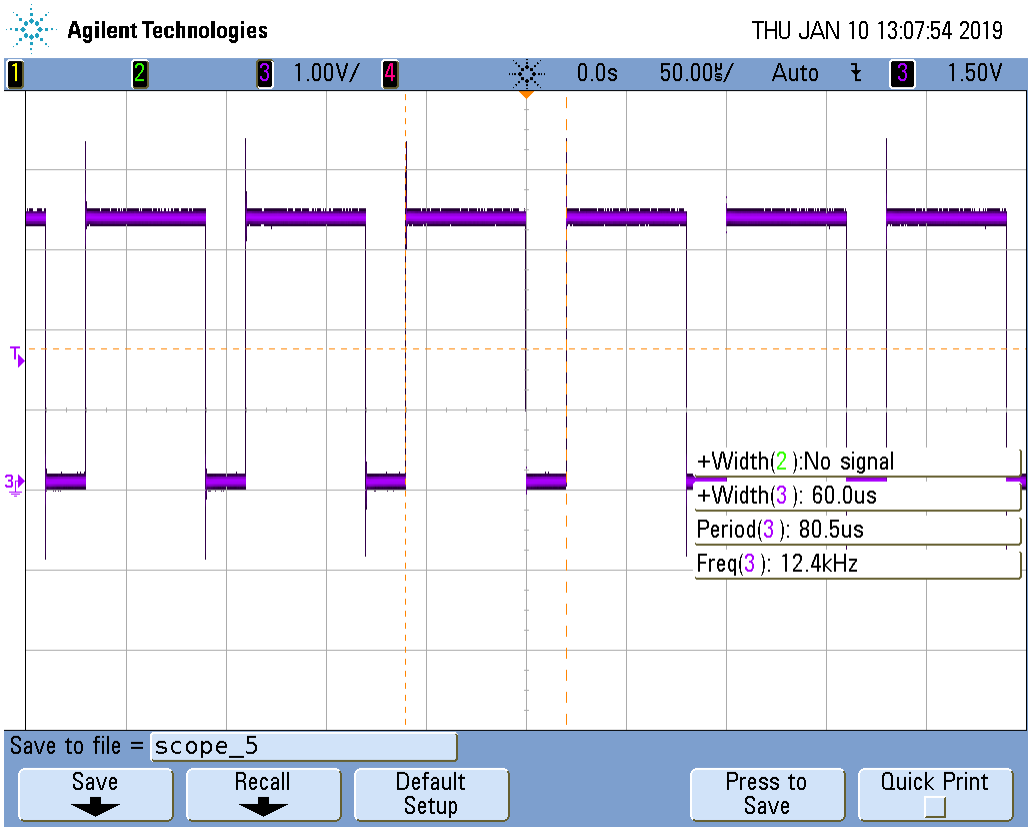


Figure 22. Oscilloscope - PWM Duty 3000, Period 4000

11 Summary

The CLB module is a powerful and highly flexible peripheral. It can be used to create incredibly powerful designs which can operate independently or as a complement to the C28x core. The use of the CLB Sysconfig Tool is also showcased, alongside the configurations needed by the XBAR module to import signals in or export signals out of the CLB module.

12 References

- *TMS320F2837xD Dual-Core Delfino™ Microcontrollers Technical Reference Manual*
- *TMS320F28004x Piccolo Microcontrollers Technical Reference Manual*

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), 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, 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 (www.ti.com/legal/termsofsale.html) 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.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265