Application Note

C2000™ Position Manager PTO API Reference Guide

Ozino Odharo, Subrahmanya Bharath, Peter Luong and Lori Heustess

ABSTRACT

Pulse-train output (PTO) and pulse-train input (PTI) are generic names for describing various forms of signal pulse streams. The PTO library APIs leverage the C2000 Configurable Logic Block (CLB) to assist in working with various PTO and PTI signals. This includes generating output signals as well as decoding input signals such as QEP, CwCCW, pulse and direction.

Table of Contents

List of Figures

List of Tables

Trademarks

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

1 Introduction

The C2000 Real-Time MCU Pulse Train Output (PTO) APIs leverage the Configurable Logic Block (CLB), Type 1 or later, to generate a specified PTO or to decode a PTI (Pulse Train Input).

Note

Some APIs work with Pulse Train Inputs (PTI) and others with Pulse Train Outputs (PTO). For simplicity, the examples, libraries, and directory structure make use of the suffix "pto" to identify content belonging to this library.

This document describes the implementation and associated software for each modules listed below:

PulseGen: Output a simple pulse and a direction-indication signal.

QepDiv: Scale Quadrature Encoded Pulse inputs (QEP-A, QEP-B and QEP-Index) to output reduced frequency PTO signals.

Abs2Qep: Translate a change in absolute position into equivalent QEP-A/B and QEP-I signals.

QepOnClb Implements a basic QEP decoder by using the CLB.

There are two categories of software provided:

Example Application Projects: Small applications which configure a C2000 Real-Time MCU, incorporates the appropriate reference library, and demonstrates the functionality. Section 6 describes how to access the source code, import the project into CCS, and then build and run the example.

Reference API Libraries: Software implementation of the module. Section 7 includes a description of each API, how to access the source code, and how to rebuild the libraries. Section 8 explains how to incorporate the API into your own project.

Note

You will need the appropriate development tools installed to build the CLB-based projects. For more information, see the *CLB Tool User's Guide*.

2 PTO – PulseGen

The PTO-PulseGen function can be used to generate pulse and direction outputs as required by the application. Figure 2-1 shows the PulseGen output and Figure 2-2 shows the implementation diagram.

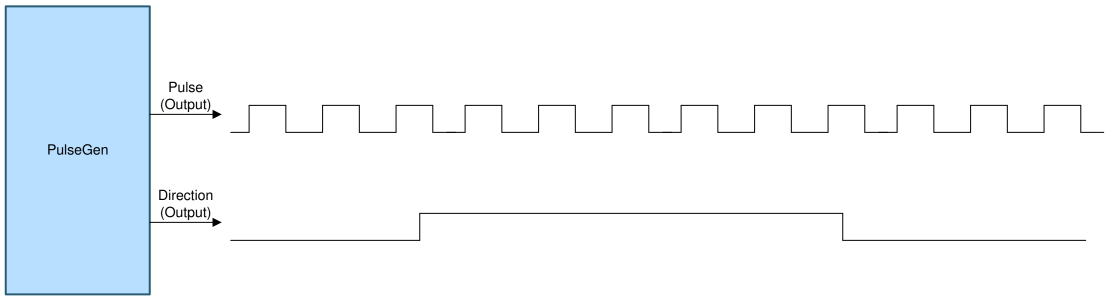


Figure 2-1. PulseGen Output Diagram

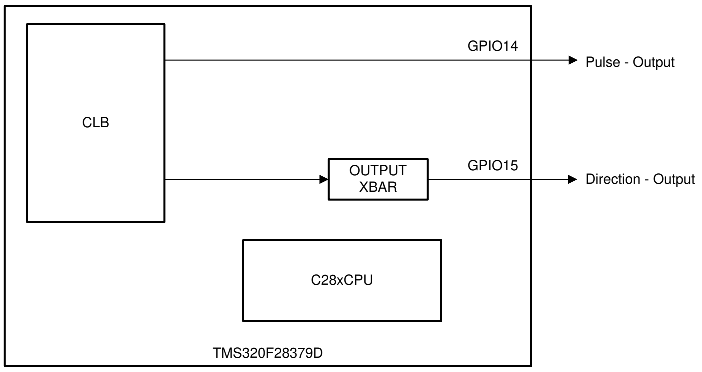


Figure 2-2. PulseGen Implementation Diagram

Note

Interconnect between the CLB and the MCU boundary may differ between devices or between examples. For specific interconnect routing information, see Section 6.

2.1 PulseGen Implementation Overview

This section provides an overview of how the PTO-PulseGen interface is implemented. This interface is achieved by the following components:

- **C28x CPU**
  - Initializes the PulseGen interface, configures the CLB, XBARs, and GPIOs.
  - Provides the number of pulses and the duration of each pulse to the CLB.

- **Configurable Logic Block (CLB) Type 1 or later**
  - Generates the pulses and direction as defined by the software interface function.
- **Device Interconnect (XBARs)**
  - Configured for output-signal routing to and from the CLB, as required.

2.2 PulseGen Limitations

The PTO-PulseGen operation has the following usage limitations:

- The minimum number of cycles must be 1000 cycles for the PTO period (at 200-MHz system clock, this corresponds to 10 µs [for example, 100 KHz]).
- The number of cycles must be between 40% to 60% of the PTO period for the interrupt time to avoid conflicts with PTO updates.
- The maximum frequency of the PTO-PulseGen output is 5 MHz at a 200-MHz CPU CLK. See the example provided in C2000Ware MotorControl SDK.

2.3 PulseGen CLB Configuration

The following resources are used inside the CLB tile to achieve the desired function detailed in Section 2.1.

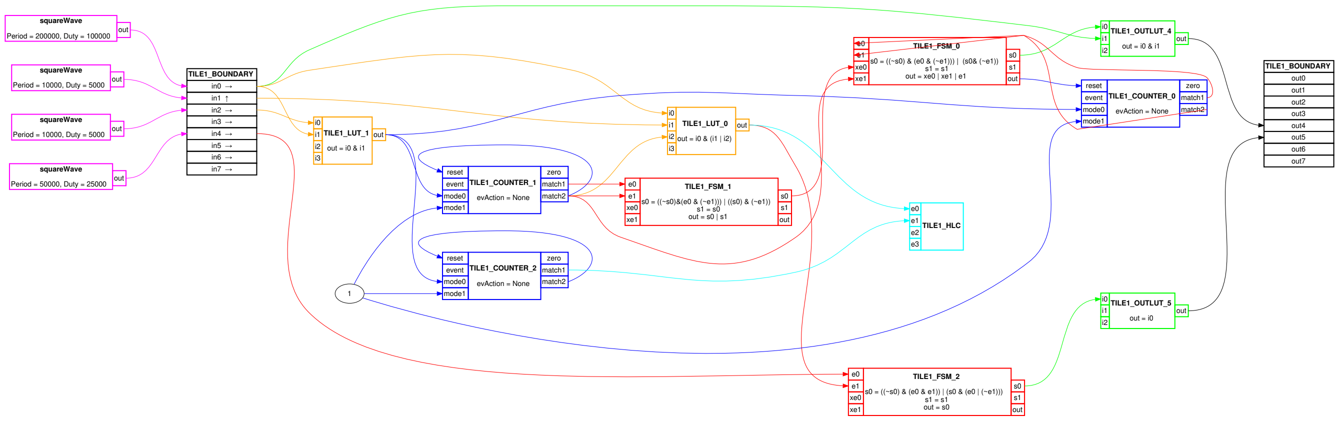


Figure 2-3. PulseGen CLB Tile Diagram

Note

Section 7 describes how to build the library project in Code Composer Studio™. By building the project, CCS will regenerate the CLB tile diagram (clb.svg or clb.html). and object (.lib). The CLB tile diagram will be located in the `RELEASE/syscfg` directory.

Implementation is described in Table 2-1 and visualized in Figure 2-3.

Table 2-1. PulseGen CLB Tile 1

Resource	Function	Notes
		Inputs
In0	On/Off Control via GPREG	Enable CLB
In1	Rising Edge Detect	Via EPWM1A
In2	On/Off Control via GPREG	Run signal (start/stop of PTO)
In3	Not used	Not used
In4	On/Off Control via GPREG	Sets the PTO direction
In5	Not used	Not used
In6	Not used	Not used
In7	Not used	Not used
		Outputs
Out0	Not used	Not used
Out1	Not used	Not used
Out2	Not used	Not used

Table 2-1. PulseGen CLB Tile 1 (continued)

Resource	Function	Notes
Out3	Not used	Not used
Out4	Transmit Enable	Via OUTPUT XBar; PTO pulse output
Out5	Transmit Enable	Via OUTPUT XBar; PTO direction output
Out6	Not used	Not used
Out7	Not used	Not used
		Logic Resources
LUT0	Input for Event0 in HLC	Edge detection on encoder input with either in1 or CNT1 match value. Triggers event in HLC to load new values into HLC registers
LUT1	Mode0 input for CNTs 1,2,3	Logic to determine the selected modes for CNT1, CNT2, and CNT3. Starts all three counters.
LUT2	Not used	Not used
FSM0	Pulse width generation	This state machine together with CNT0 will generate a number of hi and low pulse widths. The output sets the reset value of CNT0.
FSM1	Active and Full Period generation	Sets the values for the active and full period based on match1 and match2 outputs of CNT1. Outputs number of pulses in active period duration and none in between the difference of the full and active periods
FSM2	PTO output direction generation	Generates the PTO output direction. The output direction is held until the end of the full period set by FSM1.
CNT0	Pulse width generation	Counter Match1 and Match2 values determine triggers for hi and low pulse widths. The match values are loaded to FSM0 inputs, e0 and e1.
CNT1	Active and Full Period Clock generation	Generates inputs needed for FSM1 and FSM2. Match1 determines trigger for active period. Match2 determines trigger for full period. Match events are used by FSM1 to generate active and full periods. Match2 is used as extra external input in FSM0 to determine how long to hold PTO output direction.
CNT2	Counter for full period	Match1 event used to trigger interrupt in HLC. Counter is reset when full period of signal is reached
		High Level Controller
HLC	Event0 used to trigger taskEvent1 used to trigger interrupt	Event0 used to load new options for the PTO from C28 core into CLBEvent1 used to generate an interrupt based on match1 event of CNT2, which corresponds to the full period. New PTO options take effect after this event.

2.4 PulseGen Input and Output Signals

Chip-level inputs to the PTO-PulseGen interface: none.

Chip-level outputs from the PTO-PulseGen interface: Pulse Output and Direction. In the examples provided, these outputs are routed through to GPIOs as described in Section 6.

3 PTO – QepDiv

The QepDiv PTO function can be used to generate a divided pulse stream from QEP inputs. Figure 3-1 shows the QepDiv input and output diagram.

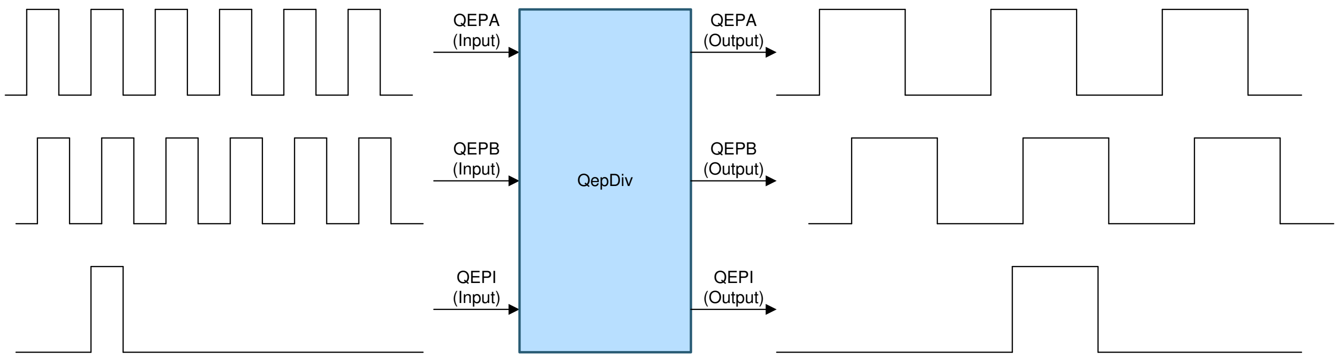


Figure 3-1. QepDiv Input and Output Diagram

Figure 3-2 shows the CLB interconnect diagram.

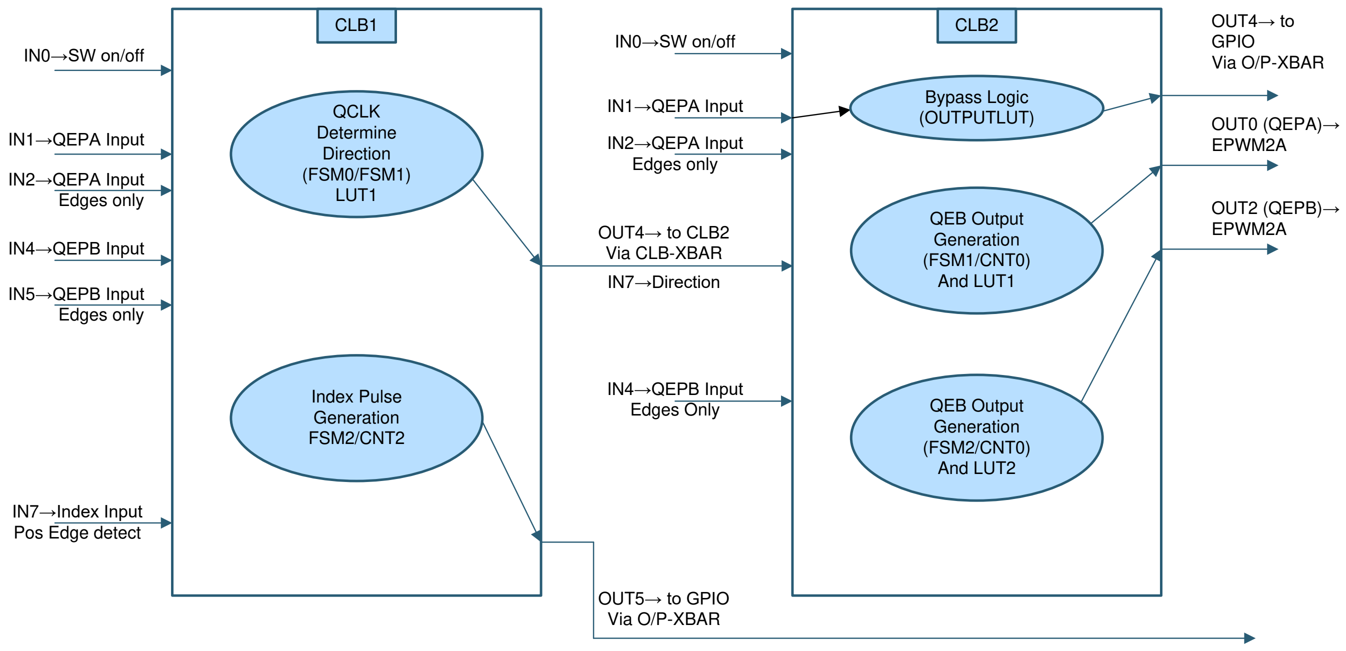

CLB1
IN0→SW on/off
IN1→QEPA Input
IN2→QEPA Input Edges only
IN4→QEPB Input
IN5→QEPB Input Edges only
IN7→Index Input Pos Edge detect
QCLK Determine Direction (FSM0/FSM1) LUT1
Index Pulse Generation FSM2/CNT2
IN0→SW on/off
IN1→QEPA Input
IN2→QEPA Input Edges only
OUT4→ to CLB2 Via CLB-XBAR
IN7→Direction
IN4→QEPB Input Edges Only
OUT5→ to GPIO Via O/P-XBAR
CLB2
Bypass Logic (OUTPUTLUT)
QEB Output Generation (FSM1/CNT0) And LUT1
QEB Output Generation (FSM2/CNT0) And LUT2
OUT4→ to GPIO Via O/P-XBAR
OUT0 (QEPA)→ EPWM2A
OUT2 (QEPB)→ EPWM2A

Figure 3-2. QepDiv Interconnect Diagram

Figure 3-3 shows the implementation diagram of the QepDiv interface.

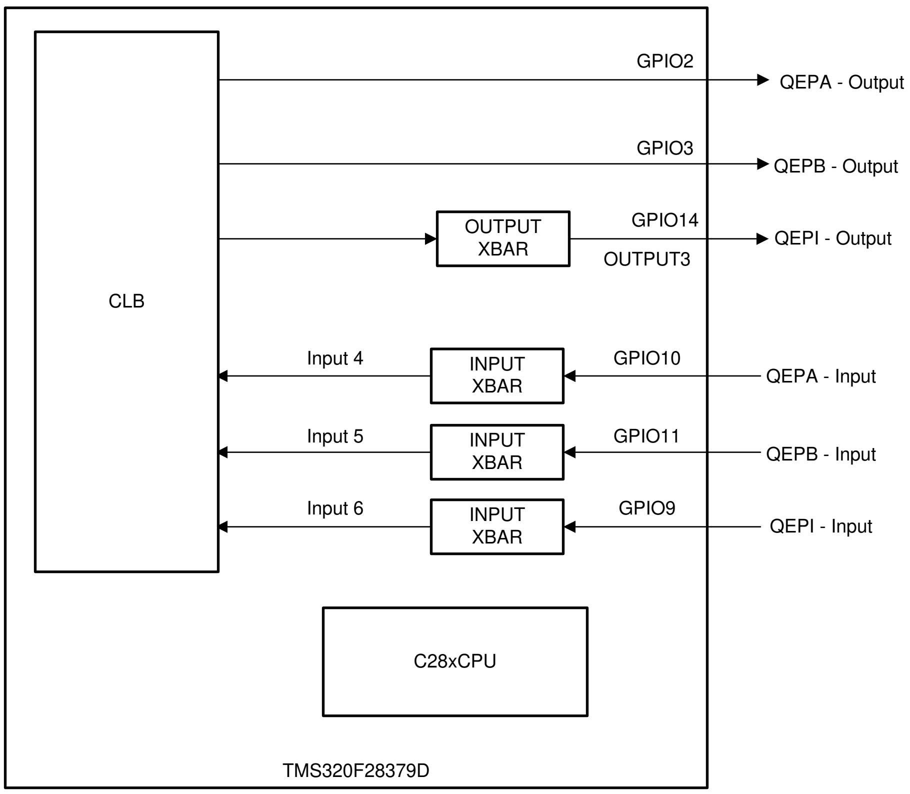


Figure 3-3. Implementation Diagram

Note

Interconnect between the CLB and the MCU boundary may differ between devices or between examples. For specific interconnect routing information, see Section 6.

3.1 QepDiv Implementation Overview

This section provides an overview of how the PTO QepDiv interface is implemented. This interface is primarily achieved by using the following components:

- **C28x CPU**
  - The CPU initializes the function and configuration of the CLB, XBARs, and GPIOs as applicable.
- **Configurable logic block (CLB) Type 1 or later**
  - Monitors input signals, QEP-A, QEP-B, and QEP-I, connected to the GPIO
  - Detects the direction of the motion and any changes in direction
  - Detects the edges of the input signals
  - Implements the division function and generates the scaled outputs: PTO-QEP-A, PTO-QEP-B, and PTO-QEP-I.
- **Device interconnect (XBARs)**
  - Input and output XBARs are used to route signals to and from the CLB as applicable.

3.2 QepDiv Limitations

The PTO-QepDiv operation has the following usage limitations:

- The QepDiv interface implements division factors of /1, /2, /4, /8, ….. up to /1024 and /2048.
- The maximum frequency of the input signals (QEP-A and QEP-B) is limited to 5 MHz.
- The index pulse is generated on the index output when a rising edge is detected on the index input signal.
- The width of the index pulse can be user defined. See the `pto_qepdiv_config` function and the corresponding example provided in C2000Ware MotorControl SDK.
- The divider values work as follows:
  - The frequency of the output QEP-A or QEP-B = frequency of input QEP-A or QEP-B / (2 × divider)

3.3 QepDiv Divider Settings and Initialization

Divider initialization is done via the function below:

- COUNTER_0 in CLB2 is used for divider*4 for match2 value of the counter.
- COUNTER_0 in CLB2 is used for divider*2 for match1 value of the counter.

Index pulse width is controlled using COUNTER_2 in CLB1 for match1 value setting.

```
uint16_t
pto_qepdiv_config(uint16_t divider, uint16_t indexWidth)
{
    CLB_writeInterface(CLB2_BASE, CLB_ADDR_COUNTER_0_MATCH2, divider * 4);
    CLB_writeInterface(CLB2_BASE, CLB_ADDR_COUNTER_0_MATCH1, divider * 2);
    CLB_writeInterface(CLB1_BASE, CLB_ADDR_COUNTER_2_MATCH1, indexWidth - 1);
    return(divider);
}
```

3.4 QepDiv CLB Configuration

The PTO API implementation source files are located under `[C2000Ware_MotorControl_SDK]\libraries\position_sensing\pto\source`.

The following resources are used inside the CLB tile to achieve the desired function detailed in Section 3.1.

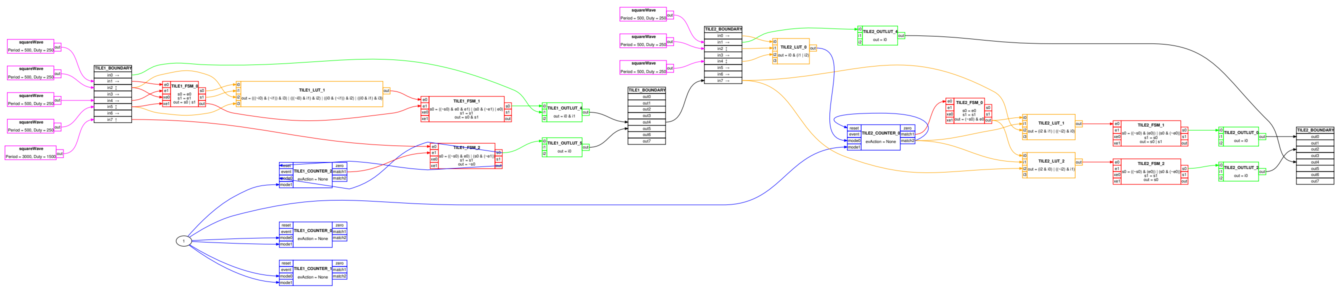


Figure 3-4. QepDiv CLB Tile Diagram

Note

You can import and build the QepDiv API reference project for each respective device, located in `[C2000Ware_MotorControl_SDK]\libraries\position_sensing\pto\ccs`. By rebuilding the compiled object, it will regenerate the CLB tile diagram (clb.svg or clb.html). and object (.lib) The CLB tile diagram will be located in the `RELEASE/syscfg` directory.

Implementation is described in detail, below and visualized in Figure 3-4.

Table 3-1. QepDiv CLB Tile 1

Resource	Function	Notes
		Inputs
In0	On/Off Control via GPREG	Enable CLB
In1	On/Off Control via GPREG	QEPA via EPWM4A
In2	Edge Detect	QEPA via EPWM4A
In3	Not used	Not used
In4	On/Off Control via GPREG	QEPB via EPWM5A
In5	Edge Detect	QEPB via EPWM5A
In6	Not used	Not used
In7	Edge Detect	QEPI via EPWM4B
		Outputs
Out0	Not used	Not used
Out1	Not used	Not used
Out2	Not used	Not used
Out3	Not used	Not used
Out4	Transmit Enable	PTO Direction Via OUTPUT XBar; Input for CLB 2
Out5	Transmit Enable	QEPI output via OUTPUTXBAR3
Out6	Not used	Not used
Out7	Not used	Not used
		Logic Resources
LUT0	Not used	Not used
LUT1	Determines QCLK direction in combo with FSM0 and FSM1	Provides input to FSM1 for external input 0
LUT2	Not used	Not used
FSM0	Alternate inputs between QEPA and QEPB	This state machine checks the QEP signals and alternates between the different signals
FSM1	Set QCLK direction	Uses output of LUT1 and FSM0 to set up the QCLK, which in turn sets the direction. The output will be routed to CLB2 as the input direction
FSM2	Index pulse generation	Takes the QEPI input and uses CNT2 Match2 value to set the QEPI output period and duty cycle.
CNT0	Set index pulse width value	Load indexWidth-1 value set via CLB_writeInterface function
CNT1	Set divider value	Load divider*4 value set via CLB_writeInterface function
CNT2	Set divide value	Load divider*2 value set via CLB_writeInterface function
		High Level Controller
HLC	Not used	Not used

Table 3-2. QepDiv CLB Tile 2

Resource	Function	Notes
		Inputs
In0	On/Off Control via GPREG	Enable CLB
In1	On/Off Control via GPREG	QEPA via EPWM4A
In2	Edge Detect	QEPA via EPWM4A
In3	Not used	Not used
In4	Edge Detect	QEPB via EPWM5A
In5	Not used	Not used
In6	Not used	Not used
In7	On/Off Control via GPREG	PTO direction routed from CLB1 out4
		Outputs
Out0	Transmit Enable	QEPA Output via EPWM2A
Out1	Not used	Not used
Out2	Transmit Enable	QEPB Output via EPWM2B
Out3	Not used	Not used
Out4	Transmit Enable	Bypass Logic
Out5	Not used	Not used
Out6	Not used	Not used
Out7	Not used	Not used
		Logic Resources
LUT0	QEPA/QEPB signal input	When the tile is on, send the selected QEP signal to CNT0 as mode0 input
LUT1	Generate high and low values for	Alternate between high and low
LUT2	Not used	Not used
FSM0	QEP Pulse width generation	This state machine together with CNT0 will generate a number of hi and low pulse widths for LUT1 and LUT2.
FSM1	QEPA signal generation	Generates QEPA output using CNT0 and LUT1.
FSM2	QEPB signal generation	Generates QEPB output using CNT0 and LUT2.
CNT0	Counter for output QEP signal generation	Counter Match1 value is the external input for FSM0. Match2 value is the reset value for the counter. The match2 value is passed to LUT1 and LUT2 .
CNT1	Not used	Not used
CNT2	Not used	Not used
		High Level Controller
HLC	Not used	Not used

4 PTO – Abs2Qep

The PTO-Abs2Qep function translates a change in absolute position into a quadrature encoder pulse train output. Figure 4-1 shows the implementation diagram of the PTO-Abs2Qep interface.

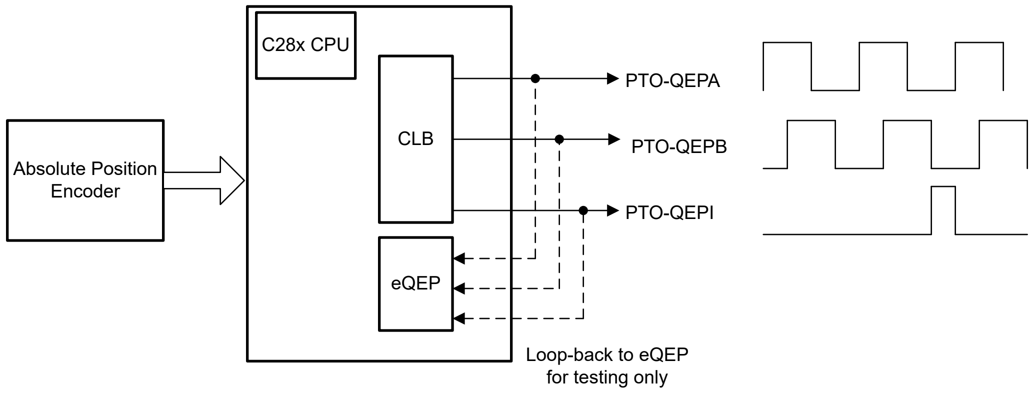


Figure 4-1. Abs2Qep Implementation Diagram

4.1 Abs2Qep Chip resources

The Abs2Qep implementation uses the following C2000 resources:

- **C28x CPU**
  - Initializes the Abs2Qep interface, configures the CLB, input/output XBARS and GPIOs.
  - Translates the change in absolute position into the equivalent QEP-A/QEP-B and QEP-I pulses.
  - Configures the CLB to generate the pulse train output.
- **Configurable Logic Block (CLB) type 1 or later**
  - Generates the PTO-QEP-A/B and QEP-I pulses as defined by the C28x.
  - Indicates the pulse train is complete by setting a CLB interrupt tag.
- **Device interconnect (XBARs)**
  - Input and output XBARs are used to route signals to and from the CLB as applicable.

4.2 Abs2Qep Theory of Operation

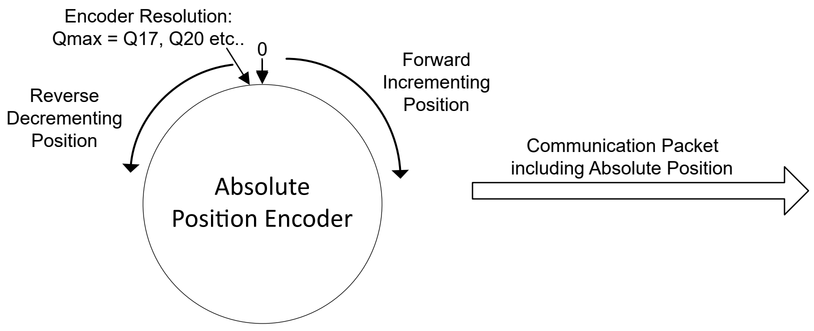

Encoder Resolution: Qmax = Q17, Q20 etc..
0
Reverse Decrementing Position
Forward Incrementing Position
Absolute Position Encoder
Communication Packet including Absolute Position

Figure 4-2. Absolute Position Encoder

An absolute encoder output represents the exact position of a rotating shaft. If Qmax is the resolution of a single rotation, then the position will range from 0 to Qmax. Resolutions in the range Q17 = 2^{17} or Q20 = 2^{20} are common. The absolute position increases when the direction is forward (clockwise) and decreases when the direction is reverse (counter-clockwise).

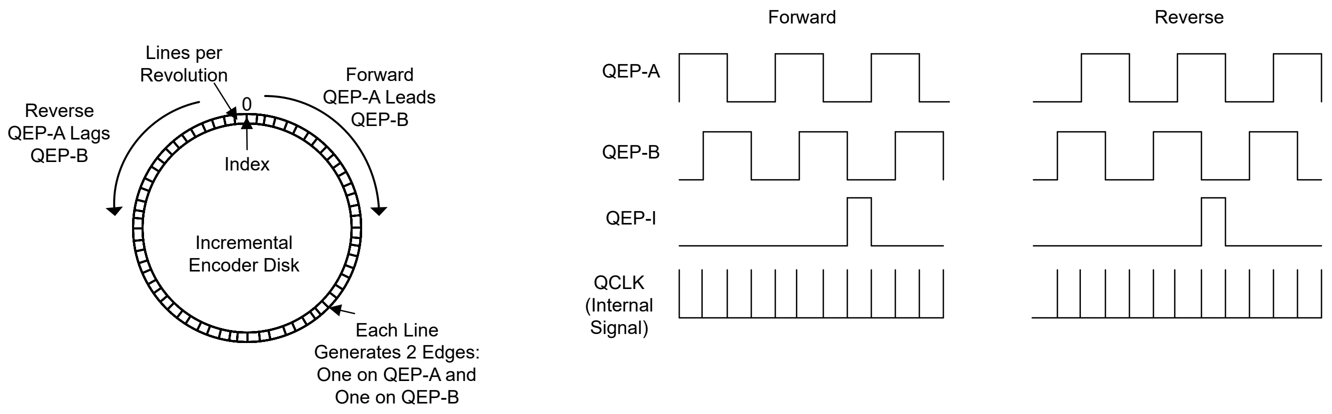


Figure 4-3. Incremental Position Encoder

An incremental encoder output is a quadrature encoder pulse (QEP). This pulse train consists of the following outputs: QEP-A, QEP-B and QEP-I with the following characteristics:

- The phase between QEP-A and QEP-B indicates the direction of movement. If QEP-A leads by 90°, then the direction is forward (clockwise). If QEP-A lags by 90° degrees, then the direction is reverse (counter-clockwise).
- The QEP-A/B frequency is proportional to the disk's velocity.
- The index signal, QEP-I, indicates crossing over absolute zero.

The resolution of the incremental encoder is specified by the number of lines around the disk. As each line passes a sensor, an edge (falling or rising) is generated on QEP-A as shown in Figure 4-3. A second channel can be added by second ring of lines, inside and offset from the outer ring. In such a case, this inner ring of lines generates QEP-B. For example, a 1024 line encoder would have 1024 QEP-A lines and 1024 QEP-B lines for a total of 2048 QEP state changes in a full rotation.

In Abs2Qep, a configurable parameter in the header file defines how many QEP state transitions are generated per line. The QEP state transition is controlled by an internal CLB signal called QCLK as shown in Figure 4-3. The default setting is each line corresponds to two QCLK pulses.

The example in Table 4-1 further clarifies this point.

Table 4-1. Abs2Qep Relationship Between Lines and QCLK (Forward Direction)

Line	QCLK	QEP-A	QEP-B
Line 1 Outer Ring	QCLK 1	Rising Edge	
Line 1 Inner Ring	QCLK 2		Rising Edge
Line 2 Outer Ring	QCLK 3	Falling Edge	
Line 2 Inner Ring	QCLK 4		Falling Edge

4.2.1 Abs2Qep Translation Equations

All of the Abs2Qep translation calculations are handled by the C28x. Based on the results, the CLB tile is then configured to generate the specific QEP signals. The CLB configuration is detailed in Section 4.3.

Note

The parameters used in the translation equations are configurable in the Abs2Qep library header file. This includes: absolute encoder resolution, incremental encoder lines per revolution, drive maximum revolutions per minute.

The Abs2Qep translation uses the ratio ABS_TO_INCR to map a change in absolute position to a corresponding number of QEP edges. Any fraction of an edge is tracked. If the fractional edge accumulation reaches 1, then an additional edge is generated.

$$\text{ABS\_TO\_INCR} = \frac{\text{QCLK\_PER\_LINE x LINES\_PER\_REV}}{\text{ABS\_MAX\_POSITION}} = \frac{\text{QCLK\_PER\_REVOLUTION}}{\text{ABS\_MAX\_POSITION}} \quad (1)$$

where

- LINES_PER_REV is the incremental encoder resolution.
- QCLK_PER_LINE is typically 2. One for QEP-A and one for QEP-B.
- ABS_MAX_POSITION = $2^{\text{ABS\_ENCODER\_RESOLUTION}}$. For example 2^{20}.

The number of QCLKs, or QEP edges, that represent a change in position is:

$$\text{QCLK} = \text{ABS\_TO\_INCR x DELTA\_ABS\_POSITION} \quad (2)$$

where:

- DELTA_ABS_POSITION = ABS_POSITION(n) - ABS_POSITION(n-1) the change in absolute position between the current sample (n) and the previous sample (n-1).
- QCLK is the total number of QEP-A + QEP-B edges required to represent the change in position.

Note

This simple translation in Equation 2 assumes absolute zero was **not** crossed. For zero-cross detection, see Section 4.2.3

For a given position change, the frequency of QCLK is such that the edges are equally divided across the position sampling period. This frequency is expressed in terms of CLB clock cycles.

4.2.2 Abs2Qep Translation Example

Given the following parameters:

- CLB clock = 10 nanoseconds
- Position sampling period = 100 microseconds or 10,000 CLB clocks
- Absolute encoder resolution = ABS_MAX_POSITION = Q20 = 1048576
- Incremental encoder resolution = 1024 lines. Therefore QCLK_PER_REV = 2 x 1024 = 2048

The ABS_TO_INCR ratio is:

$$\text{ABS\_TO\_INCR} = \frac{\text{2 x Lines}}{\text{Qmax}} = \frac{2048}{1048576} = .00195313 \quad (3)$$

Table 4-2 shows example translations from a change in absolute position to QCLKs generated.

Notice at sample 2 and sample 3, the fractional edge accumulation is greater than 1. When this occurs, an additional QCLK is generated and one is subtracted from the fractional edge accumulation.

Note

The absolute position samples shown are for illustration only. Actual position change values may be much larger than shown or may be in the reverse direction.

Table 4-2. Abs2Qep Example Calculations

Sample	Position	Delta Position [1]	QCLKs	Fractional Edges	QCLKs Generated	CLB Clocks per QCLK [2]
0	0	0	0	0	0	0
1	24000	24000	46.875	.875	46	217
2	53000	29000	56.6406	.875 + .6406 = 1.515 → .515 [3]	56+1 [3]	175
3	62000	9000	17.5781	.515 +.5781 = 1.09375 → .09375 [3]	17+1 [3]	555

(1) Position(n) - Position(n-1). In this example all changes are in the forward direction and zero is not crossed. If this value were negative, the direction would be reverse.
(2) Number of CLB clocks betweeen each QCLK pulse. This is based on the sampling frequency expressed in CLB clocks. For this example: 10,000 CLB clocks / QCLKs generated
(3) An additional QCLK is generated and the fractional portion is adjusted by 1.

4.2.3 Abs2Qep Zero Cross Detection

Abs2Qep supports generation of QEP-I to indicate crossing absolute zero. Detection of the zero cross relies on knowing the maximum revolutions-per-minute (RPM) of the absolute encoder. The RPM combined with the position sampling frequency determines the maximum possible delta change. If the delta change is larger than the absolute value of the maximum, then absolute zero has been crossed.

Consider the following example:

- Motor MAX_RPM = 30,000 revolutions per minute = 500 revolutions per second
- Position Sampling Frequency = 100 microseconds
- Absolute encoder resolution = ABS_MAX_POSITION = Q20 = 2^{20}= 1048576

The maximum position change is:

$$500\frac{\text{revolutions}}{\text{second}}\text{ x }100\frac{\text{microseconds}}{\text{sample}} = .05\frac{\text{revolutions}}{\text{sample}} \quad (4)$$

Therefore any position change where the magnitude is greater than .05 x Qmax is assumed to be a zero crossing.

Figure 4-4 illustrates crossing zero in the forward direction. Position(n) is a relatively small number and Position(n-1) is a very large number. Position(n) - Position(n-1) is, therefore, a negative number with a magnitude greater than ABS_MAX_POSITION.

In this case, Abs2Qep uses the sum of two measurements to determine the equivalent QEP pulses:

- (A) delta between Position(n-1) and Qmax
- (B) delta between 0 and Position(n)

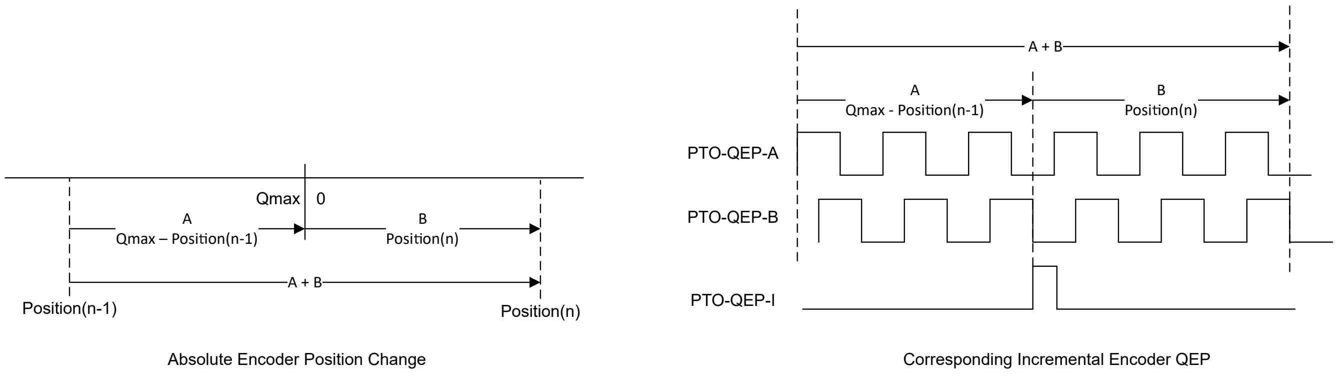


Figure 4-4. Abs2Qep Zero-Cross in the Forward Direction

Figure 4-5 illustrates zero crossed in the reverse direction. Position(n) is a large value and Position(n-1) is relatively small. Therefore Position(n) - Position(n-1) will be positive and have a magnitude > ABS_MAX_POSITION.

In this case, Abs2Qep uses the sum of measurements:

- (A) delta between Position(n-1) and zero
- (B) delta between Qmax and Position(n)

Notice, in the resulting PTO, QEP-B leads by 90° indicating a reverse direction.

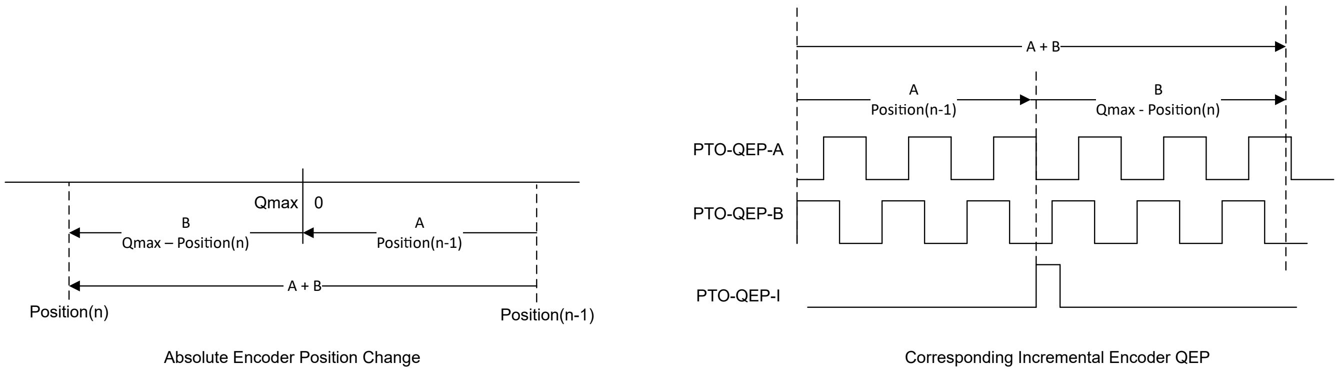


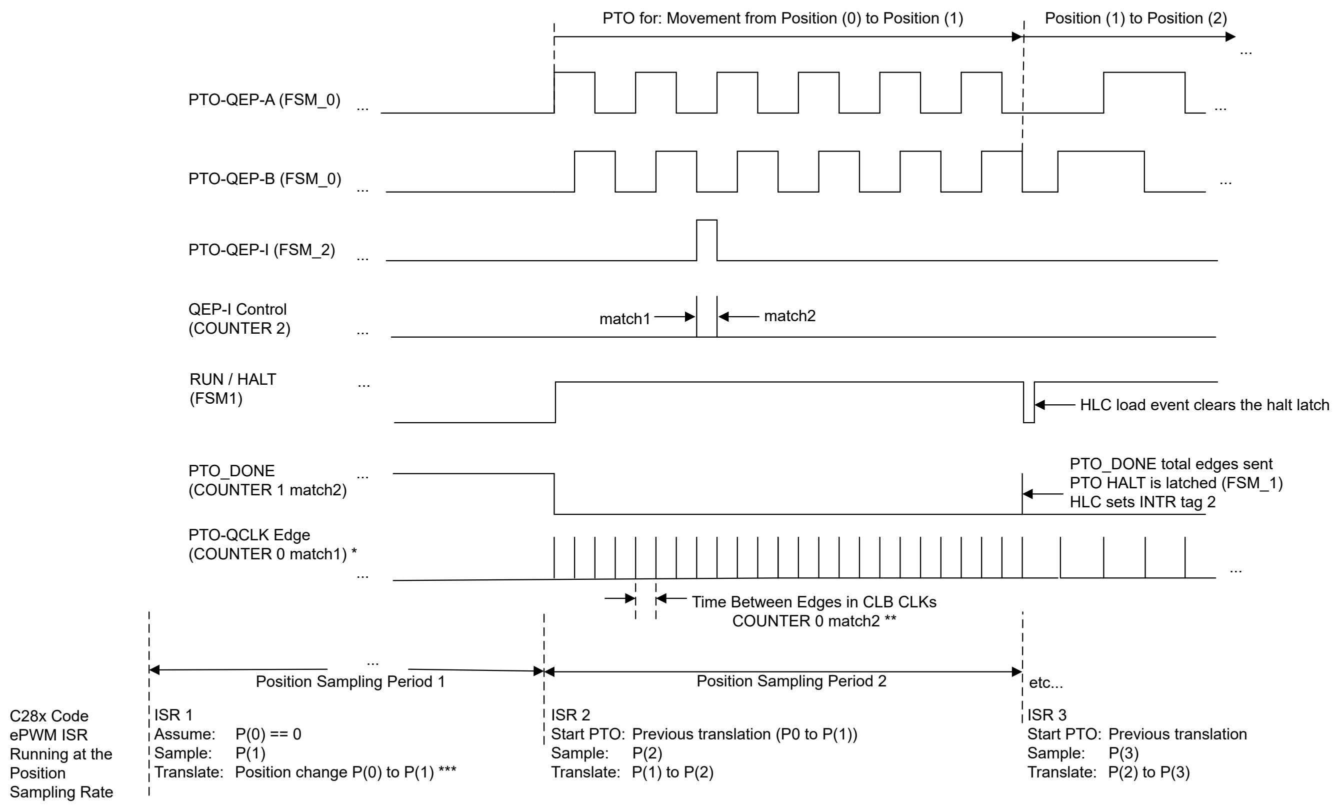
Figure 4-5. Abs2Qep Zero-Cross in the Reverse Direction

4.3 Abs2Qep CLB Configuration

After translating the absolute position into QCLK pulses as described in Section 4.2, the C28x loads the PTO parameters into the HLC's FIFO. When it is time to start the PTO, a command via the GPREG bits will signal the HLC to pull the parameters from the FIFO, to load them into the counters, and then to start the PTO. Once started, the CLB generates the PTO-QEP waveform independently.

Once the PTO is complete, the HLC will set a CLB interrupt tag. The C28x can use this to flag to check if the PTO is complete before loading a new configuration.

Figure 4-6 shows PTO-QEP waveforms and their tie to CLB components. In this example, the position sampling period is controlled by an ePWM ISR on the C28x.


PTO for: Movement from Position (0) to Position (1)
Position (1) to Position (2)
PTO-QEP-A (FSM_0)
PTO-QEP-B (FSM_0)
PTO-QEP-I (FSM_2)
QEP-I Control (COUNTER 2)
match1
match2
RUN / HALT (FSM1)
HLC load event clears the halt latch
PTO_DONE (COUNTER 1 match2)
PTO_DONE total edges sent
PTO HALT is latched (FSM_1)
HLC sets INTR tag 2
PTO-QCLK Edge (COUNTER 0 match1) *
Time Between Edges in CLB CLKs
COUNTER 0 match2 **
Position Sampling Period 1
Position Sampling Period 2
etc...
C28x Code ePWM ISR Running at the Position Sampling Rate
ISR 1
Assume: P(0) == 0
Sample: P(1)
Translate: Position change P(0) to P(1) ***
ISR 2
Start PTO: Previous translation (P0 to P(1))
Sample: P(2)
Translate: P(1) to P(2)
ISR 3
Start PTO: Previous translation
Sample: P(3)
Translate: P(2) to P(3)

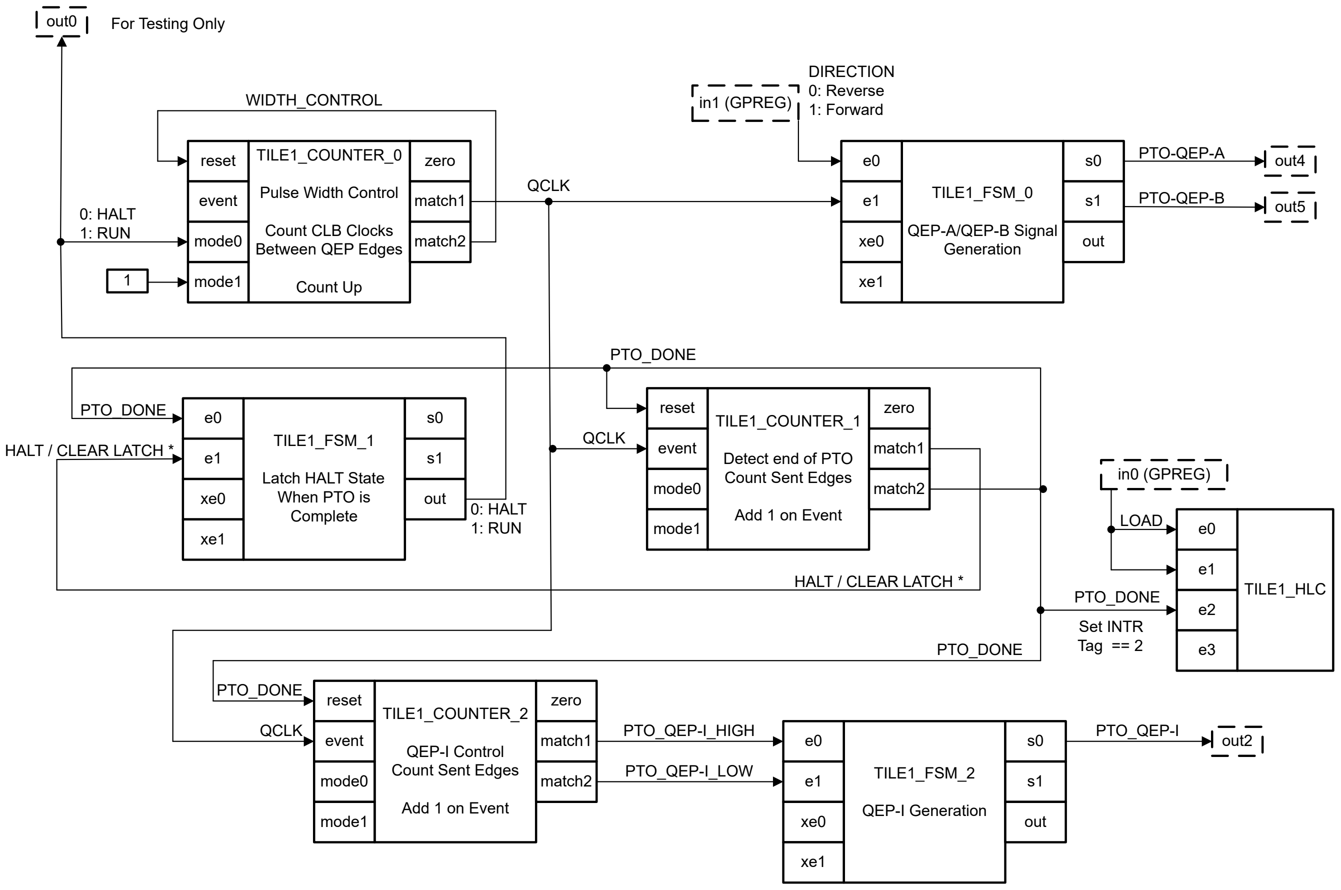
A. Counter 0 match 1 is initialized to a fixed value and is not changed. This match generates the QCLK pulse. This places QCLK toggle at the beginning of the count, reducing the time between the previous PTO halt and the next PTO start.

B. The time between QCLK pulses is controlled by the COUNTER 0 match 2. This match resets COUNTER 0.

C. P(n) stands for Absolute Position at sample n.

Figure 4-6. Abs2Qep Example System Waveform

The CLB Tile block diagram is shown in Figure 4-7 and Table 4-3 describes the functionality of each CLB component in detail.



A. HALT/CLEAR LATCH is controlled directly by HLC during a LOAD event.

B. PTO_DONE is controlled by the COUNTER 1 incrementing and directly by the HLC during a LOAD event.

Figure 4-7. Abs2Qep CLB Tile Block Diagram

Table 4-3. Abs2Qep CLB Tile 1

Resource	Function	Notes
		Inputs
In0	LOAD control Rising edge: Loads new PTO configuration (HLC).	Connected to GPREG bit 0. Before loading a new configuration, check that the last PTO is complete (INTR tag == 2)
In1	DIRECTION control 1: clockwise (forward) 0: counter-clockwise (backward)	Connected to GPREG bit 1. Change only when the last PTO is complete (Intr Tag 2)
In2	Not used	Not used
In3	Not used	Not used
In4	Not used	Not used
In5	Not used	Not used
In6	Not used	Not used
In7	Not used	Not used
		Outputs
Out0	Not used	Not used
Out1	Not used	Not used

Table 4-3. Abs2Qep CLB Tile 1 (continued)

Resource	Function	Notes
Out2	PTO_QEP-I The index transiton from 0 to 1 indicates the absolute zero position has been crossed.	The index output signal.
Out3	Not used	Not used
Out4	PTO_QEP-A	PTO quadrature output A
Out5	PTO_QEP-B	PTO quadrature output B
Out6	Not used	Not used
Out7	Not used	Not used
Logic Resources		
LUT0	Not used	Not used
LUT1	Not used	Not used
LUT2	Not used	Not used
FSM0	Generate PTO_QEP-A and PTO-QEP-B	Generate 1 edge on each QCLK input. The lead/lag of QEP-A/B is based on the current state and the DIRECTION input signal.
FSM1	Generate HALT/RUN signal	Halt the PTO output if either of these conditions is true: • The PTO completes. This halt is latched and the PTO will remain halted until the latch is cleared via the HALT/RUN control input • The HALT/RUN control input is high.
FSM2	Generate PTO_QEP-I signal	Force PTO_QEP-I high and low based on the QEP-I control. Enables the user to configure QEP-I to stay high for more than one QCLK if desired.
CNT0	Generate QCLK (PTO width control) signal	Counts up by 1 each CLB clock. • match1: fixed value. Generates the QCLK signal near the start of the count. This placement reduces the time between the last PTO halt and the next PTO start. • match2: number of CLB clocks between QEP edges. The counter is reset every match2 event.
CNT1	PTO edge-count control	Increments by 1 every QCLK event to count the total QEP-A + QEP-B edges sent during a PTO. • match1: manipulated by the HLC in order to clear a halt latch condition and start the PTO. • match2: number of edges to be sent. Once reached, the PTO_DONE signal is asserted. This latches a halt state and resets the edge count and resets QEP-I control.
CNT2	PTO_QEP-I control	Increments by 1 every QCLK event to count the total QEP-A + QEP-B edges sent during a PTO. • match1: Edge where PTO_QEP-I will go high • match2: Edge where PTO_QEP-I will go low **Note** If PTO-QEP-I should remain low for the whole PTO, then configure match1 and match2 to be a large number to avoid a match. (i.e. 0xFFFFFFFF).
High Level Controller		
HLC	Event 1: LOAD new PTO configuration.	Responds to a rising edge on the LOAD input from the C28x. This will configure and start a new PTO. For all of the steps, refer to the program description in Section 4.3.3
	Event 2: Signal PTO is complete.	Responds to the completion of PTO signal by setting Interrupt Tag 2. At this point, it is safe to load a new PTO counter configuration.

4.3.1 Abs2Qep QEP-A/B Pulse Train Generation

The QEP-A and QEP-B signal are generated by a finite state machine (FSM). The state diagrams are shown in Figure 4-8 and have the following characteristics:

- A state transition occurs when QCLK = 1.
- The state remains the same when QCLK = 0
- Only one QEP signal changes at a time
- Which signal transitions first depends on the direction input from the C28x.

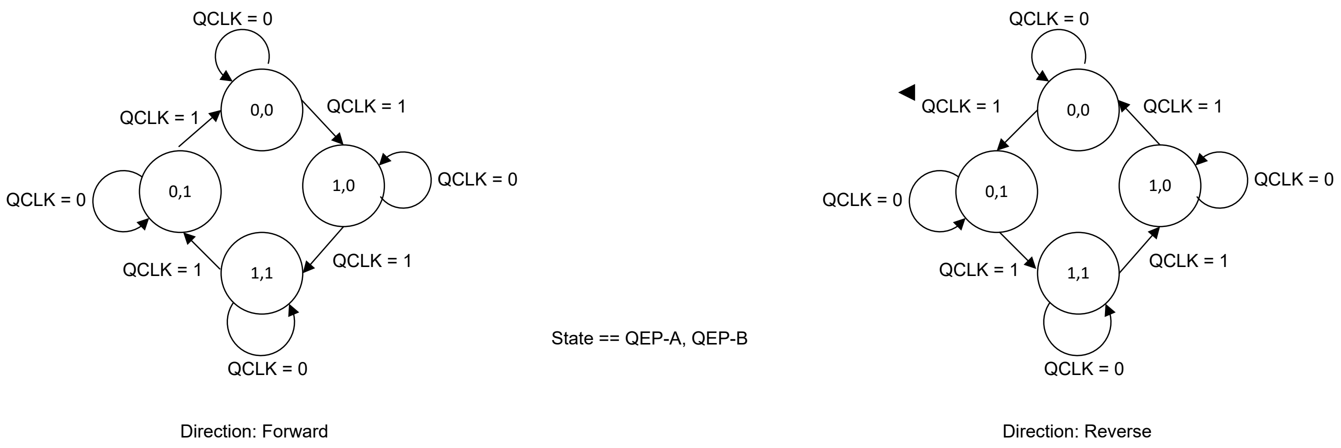


Figure 4-8. Abs2Qep PTO State Diagrams

Table 4-4 and Table 4-5 are the corresponding Karnaugh Maps. The resulting equations are determined by inspecting each "1" within the map or by using a Karnaugh Map solver. x is used to indicate states which are not valid. Note that there is no need to further simplify the equations; they can be entered into the CLB tool as shown. Use the OR operator to build up the full equation from the parts as shown in the simulation results (Figure 4-9).

Table 4-4. QEP-A (s0) Signal Generation Karnaugh Maps

		DIRECTION (e0) = 1 (Forward) Next State QCLK, QEP-B (e1, s1)						**DIRECTION (e0) = 0 (Reverse) Next State QCLK, QEP-B (e1, s1)**			
		00	**01**	**11**	**10**			**00**	**01**	**11**	**10**
Current State s0, s1 (QEP-A, B)	**00**	0	0	x [2]	1 [2]	**Current State s0, s1 (QEP-A, B)**	**00**	0	0	0	x
	01	0	0	x	0		**01**	0	0	1 [3]	x [3]
	11	1 [1]	1 [1]	0	x		**11**	1 [1]	1 [1]	x [3]	1 [3]
	10	1 [1]	1 [1]	1[2]	x [2]		**10**	1 [1]	1 [1]	x	0

(1) s0_1 = (e0 & s0 & !e1) | (!e0 & s0 & !e1) = s0 & !e1
(2) s0_2 = e0 & !s1 & e1
(3) s0_3 = !e0 & s1 & e1

Table 4-5. QEP-B (s1) Signal Generation Karnaugh Maps

		DIRECTION (e0) = 1 (Forward) Next State QCLK, QEP-A (e1, s0)						**DIRECTION (e0) = 0 (Reverse) Next State QCLK, QEP-A (e1, s0)**			
		00	**01**	**11**	**10**			**00**	**01**	**11**	**10**
Current State s0, s1 (QEP-A, B)	**00**	0	0	0	x	**Current State s0, s1 (QEP-A, B)**	**00**	0	0	x [3]	1 [3]
	01	1 [1]	1 [1]	x	0		**01**	1 [1]	1 [1]	1 [3]	x [3]
	11	1 [1]	1 [1]	x [2]	1 [2]		**11**	1 [1]	1 [1]	0	x
	10	0	0	1 [2]	x [2]		**10**	0	0	x	0

(1) s1_1 = (e0 & s1 & !e1) | (!e0 & s1 & e1) = s1 & !e1
(2) s1_2 = e0 & s0 & e1
(3) s1_3 = !e0 & !s0 & e1

Figure 4-9 shows the SystemC simulation results.

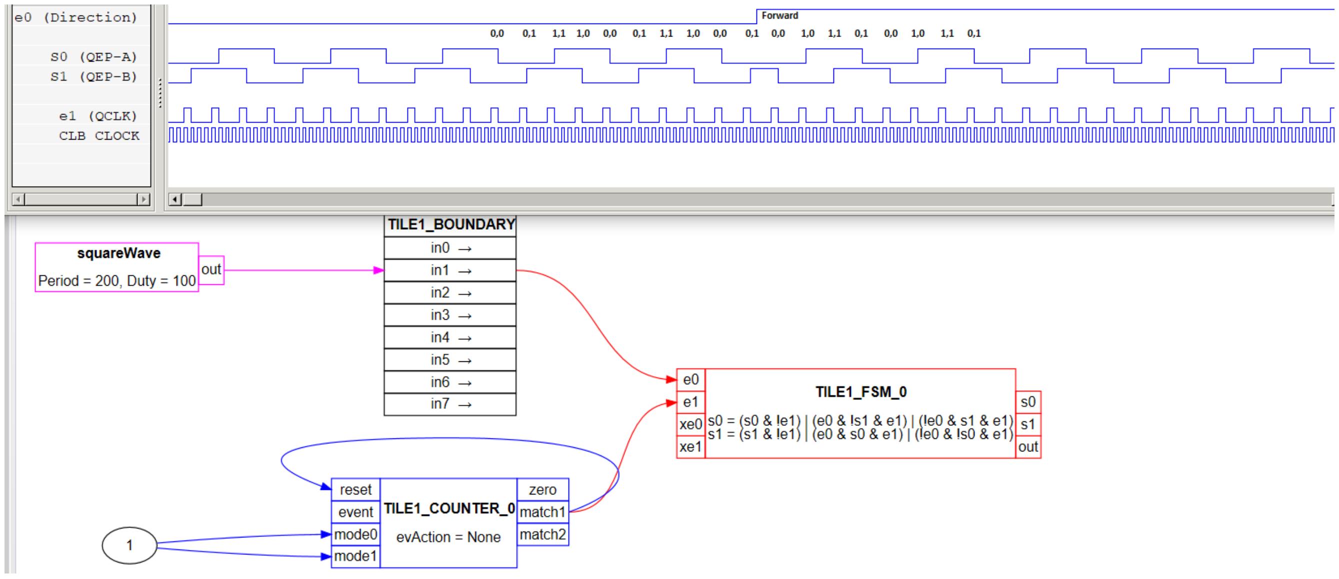


Figure 4-9. Simulation QEP-A and QEP-B Generation

4.3.2 Abs2Qep Halt Latch

The HALT_LATCH and RUN/HALT output is implemented using a finite state machine. The output signal is connected to the mode0 input of the counter which generates QCLK. The RUN/HALT output depends only on the current state of the latch and the HALT signal from the CPU. If the HALT signal is low, and the latch is not set, then QCLK will be generated (COUNTER mode0 = out = 1). In all other cases QCLK will not be generated (COUNTER mode0 = out = 0). This is expressed as out = !(s0 | e1).

Table 4-6. RUN/HALT Output

s0 (LATCH)	e1 (HALT/CLEAR LATCH)	out = !(s0|e1)	QCLK Generation
0	0	1	Run
0	1	0	Halt
1	0	0	Halt
1	1	0	Halt

HALT_LATCH is set on the rising edge of PTO_DONE. It will remain set until cleared by a a rising edge on the HALT/CLEAR_LATCH signal from the CPU.

Table 4-7. HALT_LATCH Karnaugh Map

		PTO_DONE, CLEAR_LATCH (e0, e1)			
		00	**01**	**11**	**10**
s0 (LATCH)	**0**	0	0	0	1 [(2)]
	1	1 [(1)]	0	0	1 [(1),(2)]

(1) s0_1 = s0 & !e1
(2) s0_2 = e0 & !e1

The Halt Latch SystemC simulation is shown in Figure 4-10.

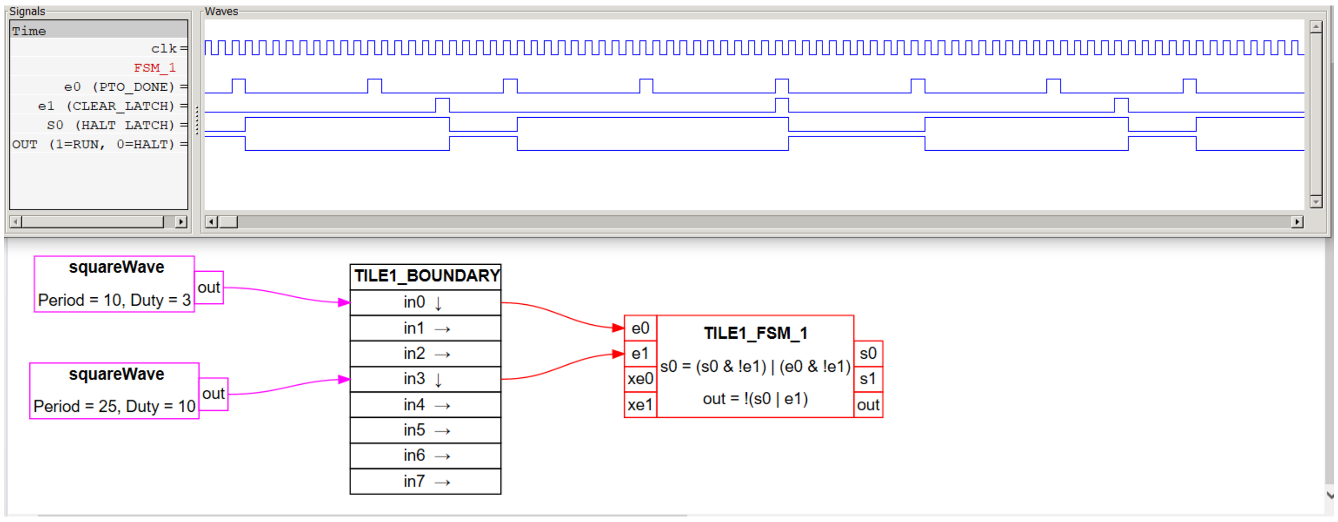


Figure 4-10. Simulation of Halt Latch

4.3.3 Abs2Qep High Level Controller (HLC)

The high level controller is programmed to:

- Start the PTO signal generation
- Modify the HALT / CLEAR_LATCH and PTO_DONE signals
- Load the counter match values required to generate a PTO
- Tag an end of PTO state

Table 4-8. Abs2Qep HLC Register Usage

R0 and R1	Used to PULL data from the FIFO.
R2	Initialized to zero during CLB configuration. Used to load zero to a match reference in order to manipulate a given signal to a high state.
R3	Initalized to 0xFFFFFFFF during CLB configuration. Used to load a large value to a match reference in order to manipulate a given signal to a low state.

Table 4-9. Abs2Qep HLC Programs

LOAD: Event 0, Event 1		
Instruction #	**Opcode(s)**	**Description**
Program0: 0	`MOV_T1 R2, C1`	Assert HALT/CLEAR_LATCH. COUNTER 1 has been reset by PTO_DONE (count == 0). Loading a match1 reference of zero will force a rising edge on HALT/CLEAR_LATCH.
Program0: 1	`MOV_T2 R3, C1`	Force the PTO_DONE signal low.

Program0: 2, 3	`PULL R0`
	`MOV_T2 R0, C1`
		Load the number of QCLKs to be generated. Note: for a case of zero QCLKs: since COUNTER_1 count == 0, a QCLK value of zero will force PTO_DONE back to a high state.

Program0: 4, 5	`PULL R1`
	`MOV_T2 R0, C0`
		Load the number of CLB clocks between two QCLKs. When the counter reaches this value, it will be reset to zero.

Program0: 6, 7
Program1: 0, 1
	`PULL R0`
	`MOV_T1 R0, C2`
	`PULL R0`
	`MOV_T2 R0, C2`
		Configure which QCLK edge will force PTO-QEP-I high and low. A large value will be passed through the FIFO if PTO-QEP-I should remain low.

Program1: 2	`MOV R1, C0`	Set COUNTER_0 to zero. This prevents the counter from incrementing by 1 when a zero pulse configuration is loaded.

Table 4-9. Abs2Qep HLC Programs (continued)

LOAD: Event 0, Event 1		
Instruction #	**Opcode(s)**	**Description**
Program1: 3	`INTR 1`	Tag indicates Event 0 plus Event 1 complete. This is placed next to the last instruction to keep it from being back-to-back with the INTR instruction in Event 2.
Program1: 4	`MOV_T1 R3, C1`	Force the HALT / CLEAR_LATCH signal low. This will start PTO signal generation if the PTO_DONE signal is low. If PTO_DONE is high, then the HALT_LATCH will be set.
	PTO_DONE: Event 2	
Instruction #	**Opcode(s)**	**Description**
Program2: 0	`INTR 2`	Tag to indicate Event 2 is complete or that the PTO has finished.

4.4 Abs2Qep Input and Output Signals

Chip-level inputs to the Abs2Qep interface: For the examples, a simulated absolute position is generated by a test function and no external input is required. If an absolute encoder is used, then connect it as described in the absolute encoder interface documentation.

Chip-level outputs from the Abs2Qep interface: The output signals are PTO-QEP-A, PTO-QEP-B and PTO-QEP-I signals. In the examples provided, these outputs are mapped to GPIOs as described in Section 6

5 PTO – QepOnClb QEP Decoder

The QepOnClb configures the CLB block to implement a simple QEP decoder module. Figure 5-1 shows the implementation diagram of the QepOnClb.

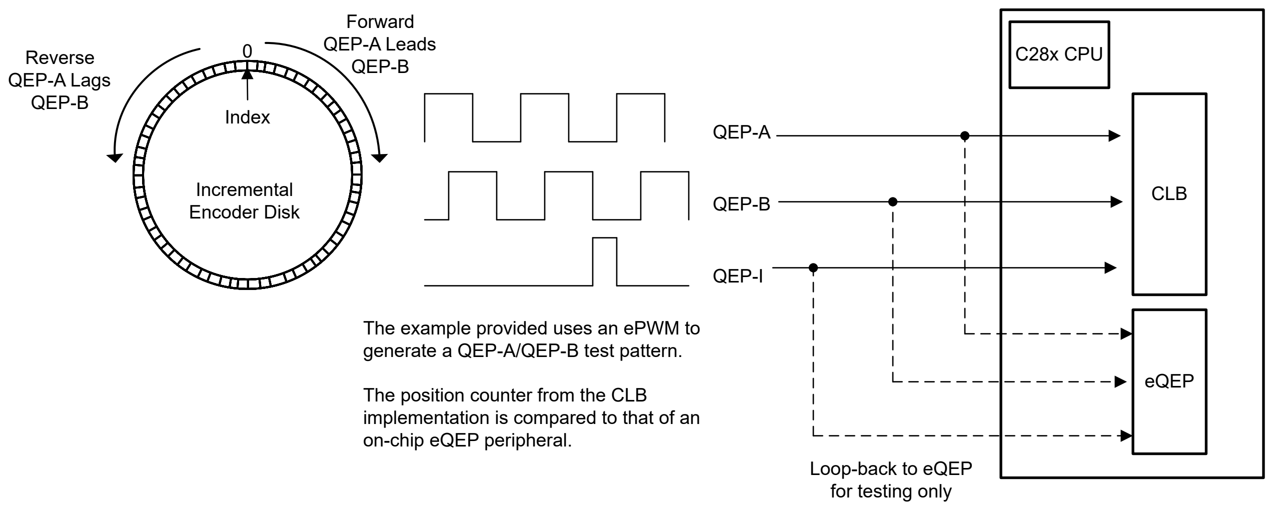

Reverse
QEP-A Lags
QEP-B
0
Forward
QEP-A Leads
QEP-B
Index
Incremental
Encoder Disk
QEP-A
QEP-B
QEP-I
C28x CPU
CLB
eQEP
The example provided uses an ePWM to generate a QEP-A/QEP-B test pattern.
The position counter from the CLB implementation is compared to that of an on-chip eQEP peripheral.
Loop-back to eQEP
for testing only

Figure 5-1. QEP on CLB Implementation Diagram

5.1 QepOnClb and eQEP Comparison

This decoder implementation is similar to the C2000 eQEP peripheral operating in quadrature-count mode. Table 5-1 provides a high-level comparison of the features supported. In some cases, potential modifications have been provided. Refer to the device-specific Technical Reference Manual for a detailed description of the eQEP peripheral.

Table 5-1. QepOnClb and eQEP (Type 0) Comparison Overview

	eQEP Feature	**QepOnClb Support**
Quadrature-clock mode (QEP-A/B signals)	Detect direction, count, and invalid state transitions.	Implemented. The current state of the QEP-A/B signals is compared to the previous state. The phase relationship determines the direction of movement. If both signals change at the same time, then an invalid state transition is detected.
	An invalid state transition sets a flag or optionally generates an interrupt.	Implemented. The CLB will issue an interrupt on an invalid state transition. If instead a flag is desired, the CLB interrupt tag can be used and the interrupt left disabled.
	Configure 4x count: both the rising and falling edge of both QEPA/B	Implemented as: Both the rising and falling edges of QEPA/B generate a count. This behavior can be changed by modifiying the QCLK state machine.
	Reverse count (reverse QEP-A/B inputs)	Not implemented. To implement, modify the CLB configuration to swap the QCLK state machine inputs.
Direction-count mode (XCLK and DIR signals)	QEP-A becomes XCLK and QEP-B becomes DIR.	Not implemented. To implement, modify the CLB configuration and (1) disconnect DIR (QEP-B) from the direction decode LUT and (2) connect DIR directly to the direction control, mode 1, of the counter.
QEP-I (index or zero signal)	Latch the position counter which can then be read using a driverlib function. This can be configured to be rising-edge, falling-edge or event marker/software index marker based.	Implemented as: Latch the position counter on the QEP-I rising-edge. The position counter value can be read from the HLC FIFO using a provided library function. On some devices the HLC can be configured to respond to a falling-edge instead of rising-edge.
	Initialize the position counter.	Not implemented. To implement, modify the HLC program to initalize the counter with a value pulled from the FIFO.
	Reset the position counter.	Not implemented. To implement, route the QEP-I signal to the QEP reset generation LUT and it into the LUT's equation.
QEP-S (strobe signal)	Latch the position counter o reset the position counter.	Not implemented. Can be implemented by following the QEP-I example.
Position counter operating modes	Reset on index, maximum position, first index or unit time-out event.	Implemented as: Reset on a maximum position value. This value is configured through a provided library function pto_qeponclb_configMaxCounterPos(). Refer to Section 7.6.
Position compare unit		Not implemented.
Edge capture unit		Not implemented.
Watchdog		Not implemented.

Table 5-1. QepOnClb and eQEP (Type 0) Comparison Overview (continued)

	eQEP Feature	QepOnClb Support
Unit timer base (QUTMR)		Not implemented.

5.2 QepOnClb Chip resources

The QepOnClb implementation uses the following C2000 resources:

- **C28x CPU**
  - Initializes the QepOnClb interface, configures the CLB, input/output XBARs and GPIOs.
  - Configures the CLB to implement a basic QEP decoder module in quadrature-count mode.
- **Configurable Logic Block (CLB) type 1 or later**
  - 1 CLB tile. Refer to Section 5.4 for specific CLB blocks used.
  - 32-bit QEP position counter with programmable max position.
  - QEP direction decode from QEP-A/B
  - QEP state transition error detection
  - Latch of the position counter on QEP-I
- **Device interconnect (XBARs)**
  - Input and output XBARs are used to route signals to the CLB as applicable.

5.3 QepOnClb Theory of Operation

A QEP decoder intreprets a pulse train output from an incremental encoder. A basic QEP pulse train consists of the signals QEP-A, QEP-B, and QEP-I as shown in Figure 5-3. These signals have the following characteristics:

- The QEP-A/B phase indicates the direction of movement. If the rising edge of QEP-A leads by 90 degrees, then the movement is forward (clockwise). If the rising edge of QEP-A lags, then the direction is reverse (counter-clockwise). This is illustrated in Figure 5-2.
- The QEP-A/B frequency is proportaional to the disk's velocity.
- The index signal, QEP-I, indicates crossing absolute zero.

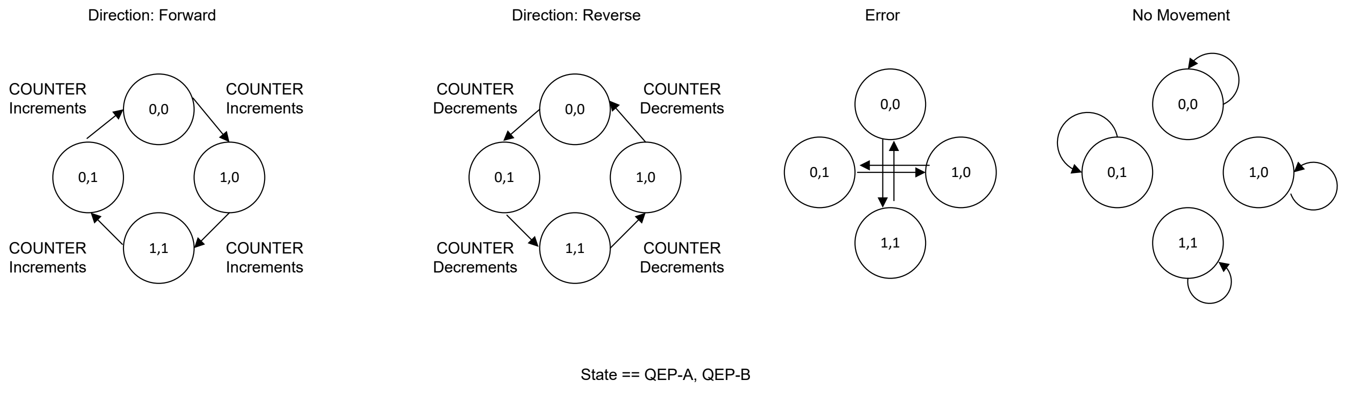


Figure 5-2. QEP-A, QEP-B State Diagrams

Design approach:

1. Select CLB components that map to the requirements of the decoder. Table 5-2 provides an example of this mapping.
2. Draw a waveform to help visualize the desired interaction between CLB blocks. Figure 5-3 includes an example QEP waveform, CLB generated signals, and the corresponding CLB blocks used to implement the feature.
3. Define the equations for LUTs and FSM modules. A detailed description of each is provided in Section 5.4.

Table 5-2. Example Mapping Decoder Features to CLB Blocks

Decoder Function	CLB Block Mapping
32-bit position counter	Maps directly to the CLB 32-bit counter module. By connecting match1 and match2 to reset and an event, a count between 0 and a maximum position (MAXPOS) can be achieved.
Memory of the past state	Detection of a valid state transition, direction and error all depend on the past state of QEP-A/B. This maps to an FSM which has the ability to store the past state.
Comparison between past and present state	Once the past state is available from an FSM, comparison of the current and previous state can be accomplished by a LUT. If a LUT is not available, then an FSM can also provide this functionality. Making a comparison is required for both direction detection and error detection.
Interrupt and counter capture	Capturing the counter value and interrupting the CPU maps to the functionality of the HLC.
CPU input to the decoder such as reset and enable	Control bits from the CPU route through the GPREG to a LUT and combined (either OR or AND) with other system signals.

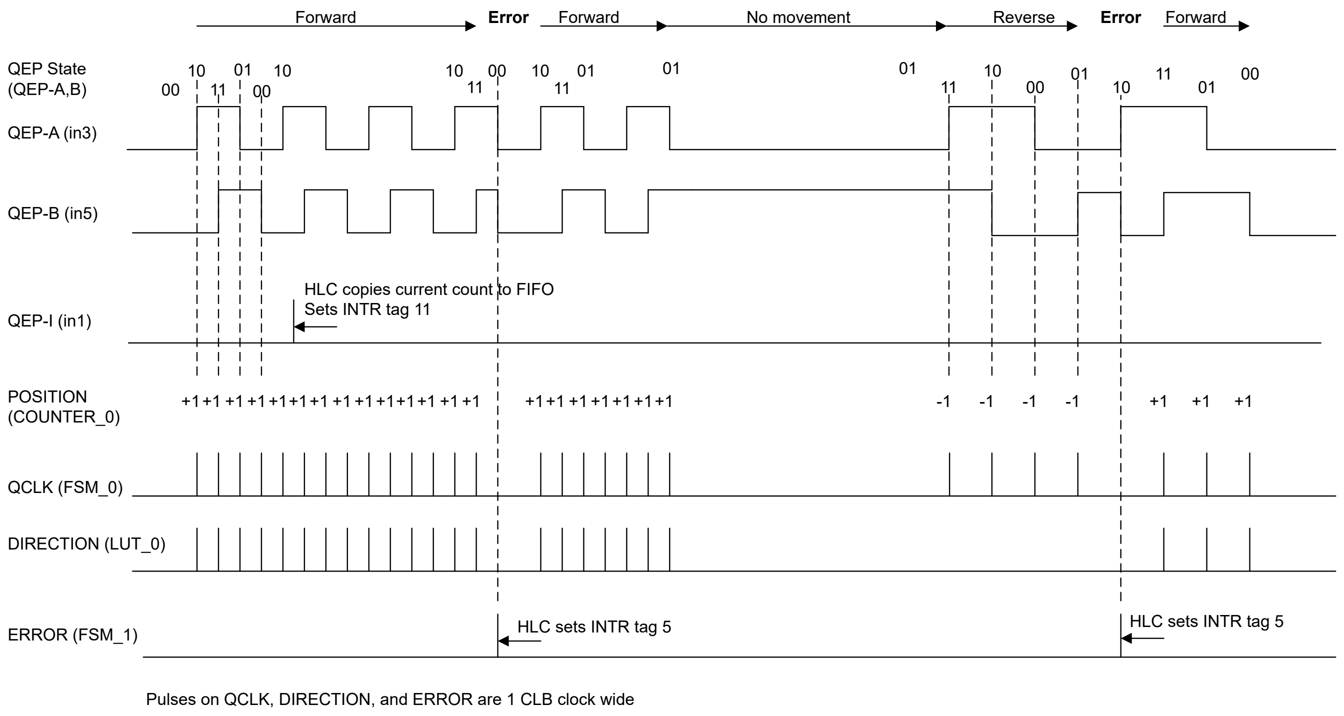


Figure 5-3. QepOnClb Waveform Example

5.4 QepOnClb CLB Resources

The decoder CLB configuration is shown in Figure 5-4 and further described in Table 5-3.

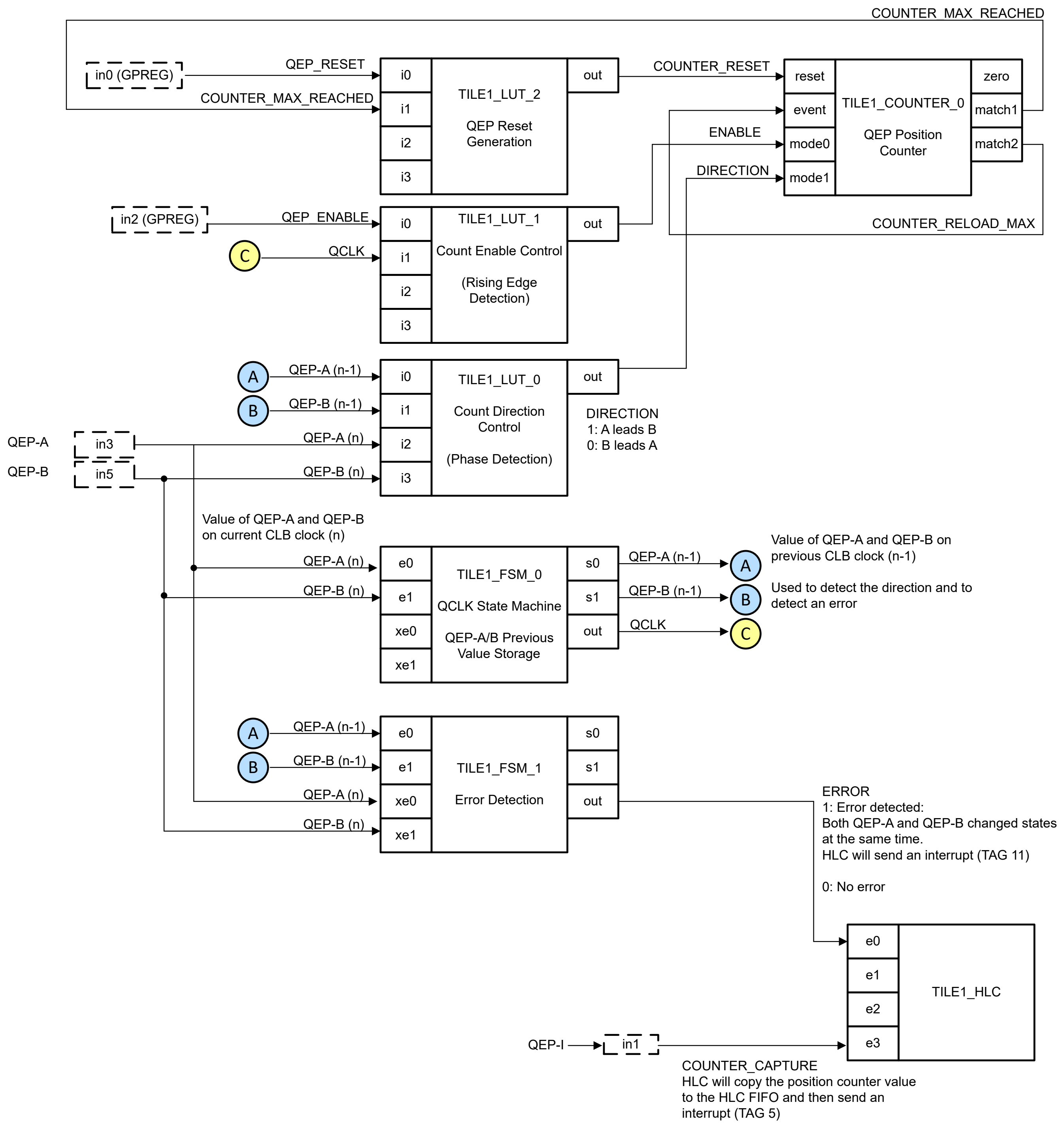


Figure 5-4. QepOnClb Tile Block Diagram

Table 5-3. QepOnClb Tile 1

Resource	Function	Notes
		Inputs
In0	QEP_RESET	Connected to GPREG bit 0 for software control of the position counter reset. • 1: Reset the position counter. The counter will remain in reset until a 0 is written to this bit. • 0: Release the counter from reset. The position counter will increment / decrement as required if QEP_ENABLE is 1.
In1	QEP-I	As designed, the rising edge of this signal will prompt the HLC to store the current position counter in the FIFO. This is similar to the eQEP latch on rising edge mode.
In2	QEP_ENABLE	Connected to GPREG bit 2. Provides an position counter enable/ disable switch from software. • 1: QEP is enabled. The position counter will increment / decrement as required if the reset signal, QEP_RESET, is also 0. • 0: QEP is disabled. The position counter stops incrementing / decrementing.
In3	QEP-A	The state transitions of QEP-A and QEP-B are used to detect movement, direction of the movement, or an error.
In4	Not used	Not used
In5	QEP-B	The state transitions of QEP-A and QEP-B are used to detect movement, direction of the movement, or an error.
In6	Not used	Not used
In7	Not used	Not used
		Outputs
Out0	Not used	Not used
Out1	Not used	Not used
Out2	Not used	Not used
Out3	Not used	Not used
Out4	Not used	Not used
Out5	Not used	Not used
Out6	Not used	Not used
Out7	Not used	Not used
		Logic Resources
LUT0	Count direction control	Determines the direction of movement. Decodes the phase by comparing the current QEP-A, QEP-B state to the previous state. The output sets the position counter's mode appropriately. • QEP-A leads (forward): DIRECTION is 1 and the position counter mode is increment. • QEP-A lags (reverse): DIRECTION is 0 and the position counter mode is decrement.
LUT1	Count enable control	Enables the position counter to increment, or decrement, by one. This occurs when both of these conditions are met: 1. QEP_ENABLE is 1 and 2. QCLK is 1
LUT2	QEP reset generation	Resets the position counter when either of these conditions are met: 1. QEP_RESET is 1 or 2. The position counter match2 is asserted

Table 5-3. QepOnClb Tile 1 (continued)

Resource	Function	Notes
FSM0	QCLK state machine	This FSM has two functions: • Monitors the QEP-A/B signals to detect a valid state change. This change will pull QCLK high which will allow the position counter to increment or decrement. Refer to LUT1. • Stores the previous QEP-A/B levels. The previous state is used to detect the direction of movement or an error. Refer to LUT0 and FSM1.
FSM1	Error detection	Compares the previous QEP-A/B state with the current state. If both signals changed at the same time, then the internal ERROR signal is forced high. As designed, the rising-edge of ERROR will trigger the HLC to send an interrupt with Tag 11.
FSM2	Not used	Not used
CNT0	Position counter	If enabled (QCLK is 1) increment or decrement by one on each CLB clock cycle. The position counter's maximum position (MAXPOS) is specified by the following: • load: Maximum position minus 1 (MAXPOS - 1). This value is loaded into the counter when an event is triggered by match2. • match1: Maximum position (MAXPOS) resets the counter when reached. • match2: 0xFFFFFFFF triggers a counter event which loads the counter with the load value.
CNT1	Not used	Not used
CNT2	Not used	Not used
High Level Controller		
HLC	Event 0: Error detected.	Sent an error interrupt to the CPU with Tag 11.
	Event 3: Counter capture	Responds to the QEP-I rising edge by copying the current position counter to the FIFO. The HLC then interrupts the CPU and Tag 5 is set.

5.4.1 QepOnClb QCLK State Machine

The QCLK state machine has two functions: (1) keep a copy of the previous levels of QEP-A and QEP-B, and (2) detect a valid QEP state change and enable the position counter to increment or decrement.

To create a copy of the previous QEP-A and QEP-B signals the following state equations are used:

- s0 next = QEP-A (n) = e0
- s1 next = QEP-B (n) = e1

To determine if a valid QEP state change has occurred, the previous QEP-A/B values are compared to the current values. When a valid QEP state transition is detected, the FSM will pull QCLK high. This signal enables the position counter allowing it to increment or decrement depending on the DIRECTION signal.

There are four possible cases which are mapped in Table 5-4:

1. Invalid state change, QCLK = 0, position counter does not change
2. No movement, QCLK = 0, position counter does not change
3. Forward movement, QCLK = 1, position counter increments or decrements
4. Reverse movement, QCLK = 1, position counter increments or decrements

The resulting equations, combined by the OR operator, are entered into the CLB tool for the output of the FSM.

Table 5-4. QCLK State Machine Karnaugh Map

		Current State e0, e1 QEP-A(n), B(n)			
		00	**01**	**11**	**10**
Previous State s0, s1 QEP-A(n-1), B(n-1)	**00**	0 No movement	1 (2) Reverse	0 Invalid	1 (4) Forward
	01	1 (1) Forward	0 No movement	1 (3) Reverse	0 Invalid
	11	0 Invalid	1 (2) Forward	0 No movement	1 (4) Reverse
	10	1 (1) Reverse	0 Invalid	1 (3) Forward	0 No movement

(1) (!s0 & s1 & !e0 & !e1) + (s0 & !s1 & !e0 & !e1)
(2) (!s0 & !s1 & !e0 & e1) + (s0 & s1 & !e0 & e1)
(3) (!s0 & s1 & e0 & e1) + (s0 & !s1 & e0 & e1)
(4) (!s0 & !s1 & e0 & !e1) + (s0 & s1 & e0 & !e1)

5.4.2 QepOnClb Direction Decode

To determine the direction, a LUT compares the current QEP-A/B signals to the previous QEP-A/B signals. The previous QEP-A/B values are provided by the QCLK state machine (FSM) described in Section 5.4.1.

There are 4 possible cases shown in Table 5-5:

1. Invalid state change: DIRECTION = do-not-care, treat as 0
2. No movement: DIRECTION = do-not-care, treat as 0
3. QEP-A rising edge leads: forward movement, DIRECTION = 1
4. QEP-B rising edge leads: reverse movement, DIRECTION = 0

For case 1 and case 2 the DIRECTION signal does not matter. QCLK is kept low in this case by the QCLK state machine. A low QCLK disables the position counter and it will not increment nor decrement. For the do-not-care cases DIRECTION = 0 has been used.

The resulting equations are combined by an OR operator and can be viewed in the CLB tool.

Table 5-5. Direction Detection Karnaugh Map

		Current State i2, i3 QEP-A(n), B(n)			
		00	**01**	**11**	**10**
Previous State i0, i1 QEP-A(n-1), B(n-1)	**00**	0 No movement	0 Reverse	0 Invalid	1 (4) Forward
	01	1 (1) Forward	0 No movement	0 Reverse	0 Invalid
	11	0 Invalid	1 (2) Forward	0 No movement	0 Reverse
	10	0 Reverse	0 Invalid	1 (3) Forward	0 No movement

(1) (!i0 & i1 & !i2 & !i3)
(2) (i0 & i1 & !i2 & i3)
(3) (i0 & !i1 & i2 & i3)
(4) (!i0 & !i1 & i2 & !i3)

5.4.3 QepOnClb Error Detection

To detect an error, a FSM compares the current QEP-A/B signals to the previous QEP-A/B signals. The previous QEP-A/B values are provided by the QCLK state machine (FSM) described in Section 5.4.1.

Note

A LUT would have also been appropriate for this logic since the previous state of QEP-A and QEP-B values are provided by a seperate FSM. All of the LUTs on the tile were already in use, however, so the output of an unused FSM was leveraged to generate the ERROR signal.

There are three possible cases described in Table 5-6:

1. Valid movement in the forward or reverse direction, ERROR = 0
2. No movement, ERROR = 0
3. Both QEP-A/B change values at the same time, ERROR = 1

Table 5-6. Error Detection Karnaugh Map

		Current State xe0, xe1QEP-A(n), B(n)			
		00	**01**	**11**	**10**
Previous State e0, e1 A(n-1), B(n-1)	**00**	0 No movement	0 Reverse	1[3] Invalid	0 Forward
	01	0 Forward	0 No movement	0 Reverse	1[4] Invalid
	11	1 [1] Invalid	0 Forward	0 No movement	0 Reverse
	10	0 Reverse	1 [2] Invalid	0 Forward	0 No movement

(1) (!xe0 & !xe1 & e0 & e1)
(2) (!xe0 & xe1 & e0 & !e1)
(3) (xe0 & xe1 & !e0 & !e1)
(4) (xe0 & !xe1 & !e0 & e1)

5.4.4 QepOnClb Simulation Waveforms

Simulations for the QEP implementation are provided in this section. For more information on CLB simulations, refer to the CLB Tool User's Guide SPRUIR8.

Note

1. The QEP-A/B input stimulus was generated by CLB tile 4 using a counter and two FSMs as shown in Figure 5-5. Each FSM s0 output toggles when its e0 input changes. The outputs from the FSM modules were connected to the tile 1 simulation inputs in3 and in5.
2. To improve readability, only the last 5 bits of the position counter are shown.
3. The specified maximum position, MAXPOS, for this example is 0xD.

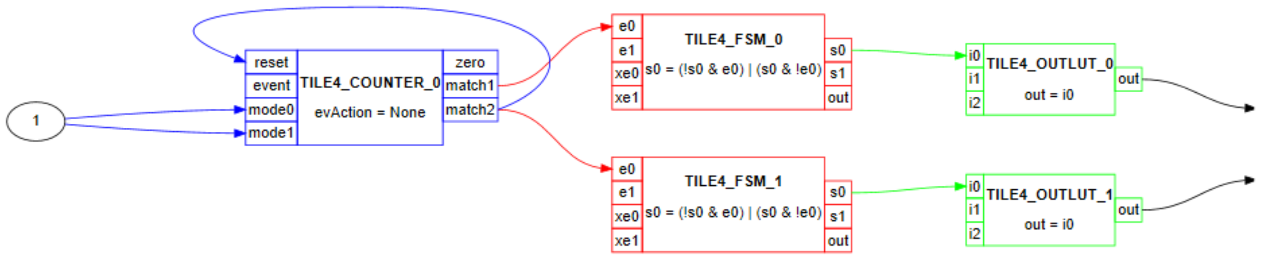


Figure 5-5. QepOnClb Simulation Stimulus

Figure 5-6 and Figure 5-7 show the simulation for movement in the forward direction. The match1 (MAXPOS) output is tied back to the reset (shown in orange) of the counter. When the counter reaches MAXPOS, it is reset to 0 on the next CLB clock.

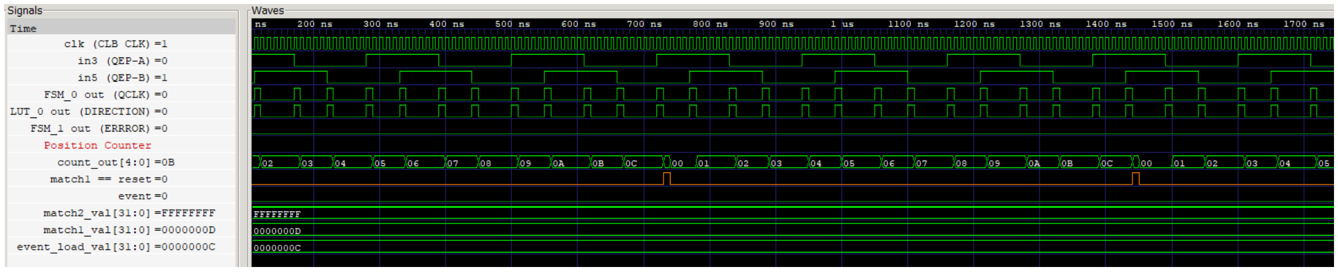


Figure 5-6. QepOnClb Forward Direction Simulation Waveform

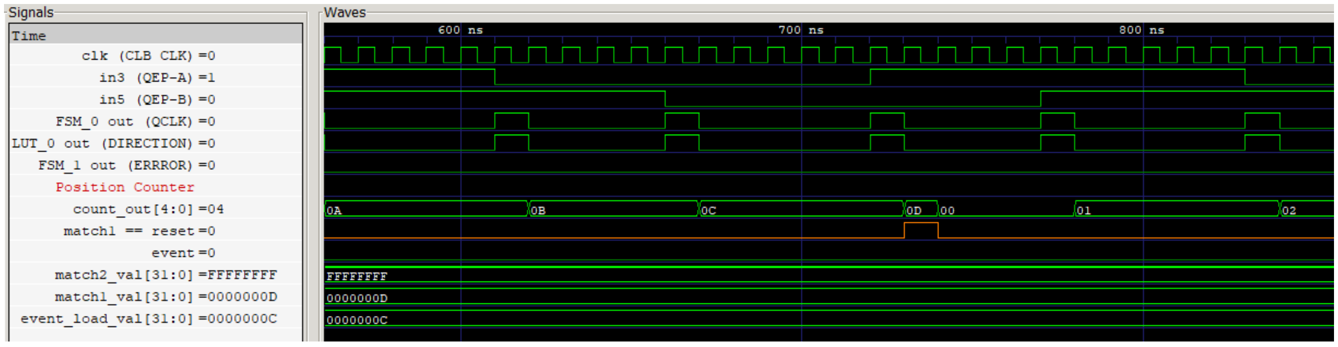


Figure 5-7. QepOnClb Forward Direction MAXPOS Simulation

Figure 5-8 and Figure 5-9 show the simulation for movement in the reverse direction. The match2 output is tied back to the event input of the counter. When the counter transitions to less than zero (0xFFFFFFFF), the load value (MAXPOS -1) is loaded into the counter.

Note

Only the low 5 bits of the position counter are shown for readability. 0x1F corresponds to 0xFFFFFFFF.

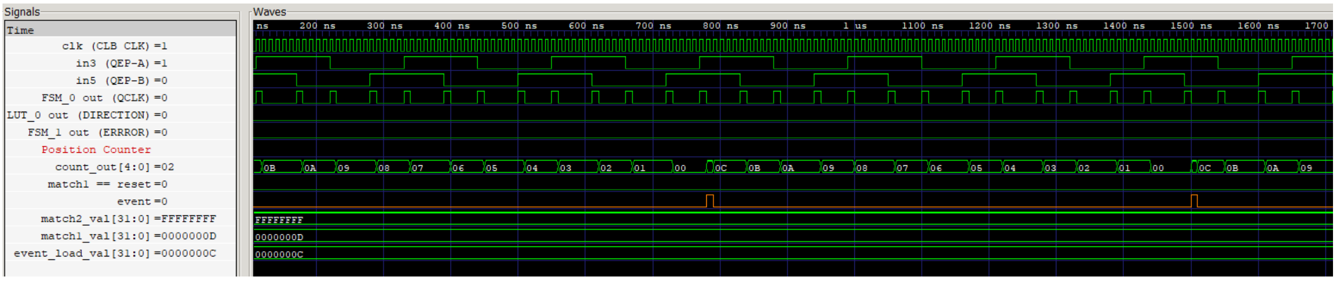


Figure 5-8. QepOnClb Reverse Direction Simulation Waveform

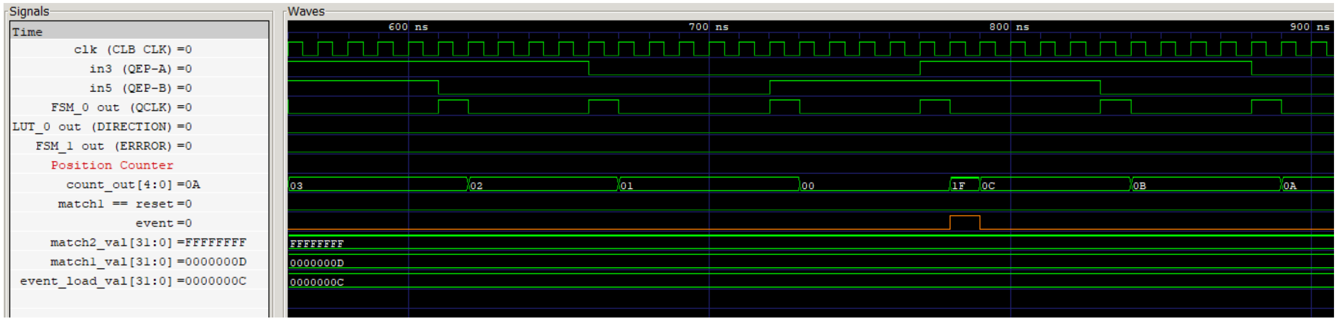


Figure 5-9. QepOnClb Reverse Direction MAXPOS Simulation

Figure 5-10 shows the error detection signal is high anytime QEP-A/B transition at the same time.

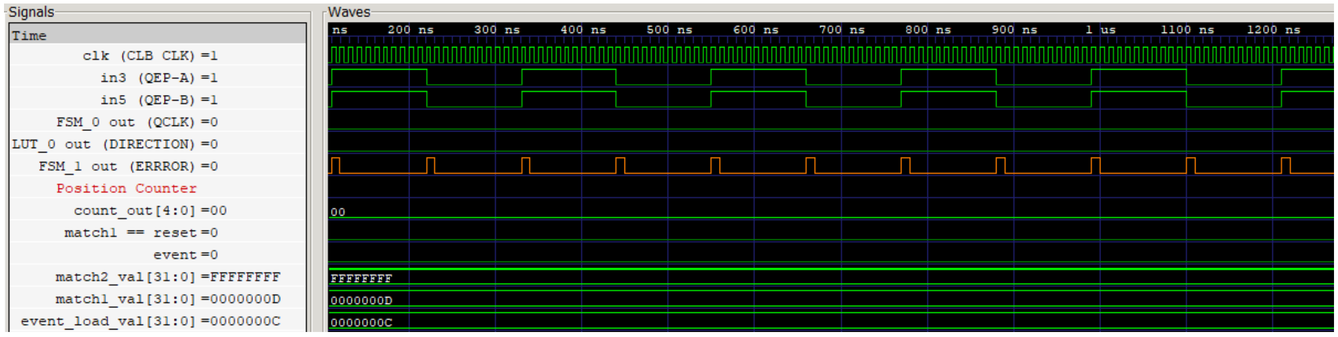


Figure 5-10. QepOnClb Error Detection Simulation Waveform

6 Example Projects

The example projects can be found in the directory shown in Table 6-1. The example projects use the API library projects that are described in Section 7.

Note

Some APIs work with Pulse Train Inputs (PTI) and others with Pulse Train Outputs (PTO). For simplicity, the examples, libraries, and directory structure make use of the suffix "pto" to identify content belonging to this library.

Table 6-1. Location of Example Solutions

`C:\ti\c2000\C2000Ware_MotorControl_SDK_[version]\`	Default install location for the SDK. (`[SDK]`)
`[SDK]\solutions\boostxl_posmgr\`	Device-specific solutions base install directory (`[pto_base]`)
`[pto_base]\shared\source`	Source code that is device independent and used across different device examples.
`[pto_base]\[device]\source`	Source code for the example projects. Includes both .c and .syscfg files.
`[pto_base]\[device]\include`	Example-specific header files.
`[pto_base]\[device]\ccs\[pto_example]`	Code Composer Studio (CCS) projectspec files. Used to import the project into your CCS workspace.
`[pto_base]\[device]\cmd`	Example project linker command files (.cmd).

6.1 Hardware Requirements

Table 6-2 describes the hardware used to run and test the PTO examples.

Table 6-2. Hardware

Device	Hardware
TMS320F28388D	F28388D controlCARD evaluation module (TMDSCNCD28388D) and docking station (TMDSHSECDOCK)
TMS320F28379D	F28379D LaunchPad development kit (LAUNCHXL-F28379D)
TMS320F280025	F280025C LaunchPad development kit (LAUNCHXL-F280025C)
TMS320F280039C	F280039C LaunchPad development kit (LAUNCHXL-F280039C [(1)])
TMS320F280049	F280049C LaunchPad development kit (LAUNCHXL-F280049C)

(1) The LAUNCHXL-F280039C device is an upcoming LaunchPad that will be released in the second quarter of 2022.

6.2 Installing Code Composer Studio and C2000WARE-MOTORCONTROL-SDK™

Install required software to build and run the PTO examples:

1. Install Code Composer Studio v11.0.0 or later, if it is not already installed on the PC
2. Install C2000WARE-MOTORCONTROL-SDK v4.00.00.00 or later, if it is not already installed on the PC

Note

To build the examples, only the above software is required. To re-build the CLB-based libraries, the CLB Tool is also required. This tool is included in Code Composer Studio (sysconfig) and the C2000Ware sub-component of the SDK (support utilities). To run CLB-based simulations requires installation of additional tools which are documented in the *CLB Tool User's Guide*.

6.3 Import and Run Example Project

1. In CCS or higher: click '*Project -> Import CCS Projects…*'.
2. Navigate to the device-specific PTO solutions CCS directory. The path is shown in Table 6-1.
3. Select the projectspec of choice and click '*Finish*'.
4. Build the project:
   a. Right-click on the project name in the project manager window
   b. Select '*Rebuild Project*'
   c. Observe the Console window for any errors or successful completion of the build.
5. Once the build completes, without errors, execute the project by selecting '*Run –>Debug*'.
6. Run the code by pressing the Run button.
7. Monitor the signals and variables as described in the following, example-specific, sections.

6.4 PulseGen Example

Verify and monitor the output waveform signals. The GPO mapping used is shown in Table 6-3:

Table 6-3. PulseGen Output Signal to GPIO Mapping

Device	Direction (Routed though OUTPUTXBAR)	Pulse Output (Routed through OUTPUTXBAR)
TMS320F280025C	GPIO15	GPIO45
TMS320F280039C	GPIO31	GPIO26
TMS320F280049C	GPIO15	GPIO26
TMS320F28379D	GPIO15	GPIO14
TMS320F28388D	GPIO15	GPIO26

6.5 QepDiv Example

The test inputs and the PTO outputs are routed internally as shown in the mapping tables. The code that routes the signals can be found in the functions listed in Table 6-4.

Table 6-4. QepDiv Example Input/Output Signal Routing

Function	Location	Notes
Input Signal Routing: GPIO to CLB		
`pto_qepdiv_setup_GPIO()`	Example application	Connect input GPIO to an INPUTXBAR.
`pto_qepdiv_initCLBXBAR()`	Library	Route INPUTXBARs to the global CLB AUXSIGx signals.
`pto_qepdiv_setupPeriph()`	Library	Connect tile inputs to the CLB global MUX, CLB local MUX or the tile's GPREG.
Output Routing: CLB to GPIO		
Function	**Location**	**Notes**
`pto_qepdiv_initCLBXBAR`	Library	Connect tile's out4 or out5 to OUTPUTXBAR
`pto_qepdiv_startOperation()`	Library	Enable CLB output to override peripheral signals via `setOutputMask()`
`pto_qepdiv_setup_GPIO()`	Example application	Connect GPIO output to a peripheral or an OUTPUTXBAR

Table 6-5. F28002x, F28003x, F28004x, F2837x and F2838x QepDiv Output GPIO Mapping

QepDiv Input			QepDiv Output		
Input Signal	**Connect to for Demo [(1)]**	**Routing to CLB**	**Output Signal**	**Routing From the CLB**	**GPIO Pin**
QEP-A: GPIO10	EPWM4A/GPIO6 or External Signal	INPUTXBAR4 → AUXSIG0 → Tile1 in1, in2 and Tile2 in1, in2	PTO_QEP-A	Override PWM2A	GPIO2
QEP-B: GPIO11	EPWM5A/GPIO8 or External Signal	INPUTXBAR5 → AUXSIG1 → Tile1 in4, in5 and Tile2 in4	PTO_QEP-B	Override PWM2B	GPIO3
QEP-I: GPIO9	EPWM4B/GPIO7 or External Signal	INPUTXBAR6 → AUXSIG2 → Tile1 in7	PTO_QEP-I	Tile1 out5 → OUTPUTXBAR3	GPIO5

(1) In the example, spare EPWMs are used to provide QEP inputs. These are for test purposes and do not correspond to real-time usage. You can choose to connect these EPWM outputs to the QepDiv input signals or you can choose to connect other external signals.

Table 6-6 lists the connections that need to be made to use the EPWMs as inputs to QepDiv.

Table 6-6. QepDiv Test Input Connections

Board	EPWM4A to QEP-A	EPWM5A to QEP-B	EPWM4B to QEP-I
LAUNCHXL-F280025C	78 (IO.6) to 14 (IO.10)	76 (IO.16) to 15 (IO.11)	77 (IO.7) to 7 (IO.9)
LAUNCHXL-F280039C	78 (IO.6) to 36 (IO.10)	76 (IO.16) to 35 (IO.11)	77 (IO.7) to 7 (IO.9)
LAUNCHXL-F280049C	78 to 40	38 to 39	77 to 37
LAUNCHXL-F28379D	80 to 76	78 to 75	79 to 77
TMDSCNCD28388D	54 to 61	57 to 63	56 to 59

6.6 Abs2Qep Example

The example uses a PWM timer to simulate the position sampling rate. The absolute position value is provided by a test function. The pass/fail criteria assumes the output from the PTO is connected externally to a eQEP peripheral.

6.6.1 Watch Variables

The following watch variables provide pass/fail information:

- passCount
- failCount
- deltaMax
- EQep1Regs.QPOSCNT

The angle of the absolute position is compared to the angle corresponding to the eQEP position counter (QPOSCNT). If the difference is less than a specified threshold, then passCount is incremented. If not, then failCount is incremented. The maximum difference is logged in deltaMax.

6.6.2 Test Signals

Two test signals are provided to aid in viewing waveforms:

- **Test signal 1:** Toggles at the start of each PWM ISR. Remains high if the PTO direction is forward. Remains low if the PTO direction is reverse.
- **Test signal 2:** The HALT/RUN signal internal to Abs2Qep. This signal can be used to visualize exactly where the PTO halts and when it restarts with respect to the ISR toggle of test signal 1.

6.6.3 Pin Usage and Test Connections

The output from the PTO is internally routed to GPIOs via the OUTPUTXBARs and by overriding PWM1-B. The code that routes the signals can be found in the functions listed in Table 6-7.

Table 6-7. Abs2Qep Output Signal Routing

Function	Location	Notes
`pto_abs2qep_initCLBXBAR`	Library	Tie Tile out4/5 to OUTPUTXBARs
`pto_abs2qep_setupPeriph`	Library	Enable output to override peripheral.
`pto_setupGPIO`	Example application	Connect OUTPUTXBARs to GPIO outputs

Table 6-8. F2838xD Abs2Qep Output GPIO Mapping

Abs2Qep Output			Test Connections
Abs2Qep Signal	**Routing From CLB to GPIO**	**GPIO / TMDSHSECDOCK Pin**	**Connect Abs2Qep Output to:**
PTO-QEP-A	Tile1 out4 to OUTPUTXBAR7	GPIO16 / Pin 67	GPIO20_EQEP1A / Pin 68
PTO-QEP-B	Tile1 out5 to OUTPUTXBAR2	GPIO3 / Pin 55	GPIO21_EQEP1B / Pin 70
PTO-QEP-I	Tile1 out2, override PWM1-B	GPIO1 / Pin 51	GPIO99_EQEP1I / Pin 96
Test signal 1: System PWM ISR / Direction	None	GPIO32 / Pin 85	Monitor with Oscilloscope
Test signal 2: Internal HALT/RUN	Tile1 out0, override PWM1-A	GPIO0 / Pin 49	Monitor with Oscilloscope

Table 6-9. F2837xD Abs2Qep Output GPIO Mapping

Abs2Qep Output			Test Connections
Abs2Qep Signal	**Routing From CLB to GPIO**	**GPIO / LAUNCHXL-F28379D Pin**	**Connect Abs2Qep Output to:**
PTO-QEP-A	Tile1 OUT4 to OUTPUTXBAR7	GPIO16 / J4-33	GPIO20_EQEP1A / J14-1
PTO-QEP-B	Tile1 OUT5 to OUTPUTXBAR2	GPIO3 / J4-37	GPIO21_EQEP1B / J14-2
PTO-QEP-I	Tile1 OUT2, override PWM1-B	GPIO1 / J4-39	GPIO99_EQEP1I / J14-3
Test signal 1: System PWM ISR / Direction	None	GPIO32 / J1-2	Monitor with Oscilloscope
Test signal 2: Internal HALT/RUN	Tile1 OUT0, override PWM1-A	GPIO0 / J4-40	Monitor with Oscilloscope

Table 6-10. F280049C Abs2Qep Output GPIO Mapping

Abs2Qep Output			Test Connections
Abs2Qep Signal	**Routing From CLB to GPIO**	**GPIO / LAUNCHXL-F280049C Pin** (1)	**Connect Abs2Qep Output to:**
PTO-QEP-A	Tile1 OUT4 to OUTPUTXBAR1	GPIO24 / J8-55	GPIO10_EQEP1A / J4-40
PTO-QEP-B	Tile1 OUT5 to OUTPUTXBAR2	GPIO3 / J6-75	GPIO11_EQEP1A / J4-39
PTO-QEP-I	Tile1 OUT2, override PWM1-B	GPIO1 / J6-79	GPIO9_EQEPI / J4-37
Test signal 1: PWM ISR / Direction	-	GPIO13 / J1-3	Monitor with Oscilloscope.
Test signal 2: HALT/RUN	Tile1 OUT0, override PWM1-A	GPIO0 / J6-80	Monitor with Oscilloscope.

(1) J8 and J6 are swapped on the silkscreen of RevA F28004x LaunchPad. J8 and J6 Pin numbers in this table refer to the silkscreen on Rev A. To confirm whether this applies to your board, see the "known issues" in the revision section of *C2000™ Piccolo™ F28004x Series LaunchPad™ Development Kit*.

Table 6-11. F280025C Abs2Qep Output GPIO Mapping

Abs2Qep Output			Test Connections
Abs2Qep Signal	**Routing From CLB to GPIO**	**GPIO / LAUNCHXL-F280025C Pin**	**Connect Abs2Qep Output to:**
PTO-QEP-A	Tile1 OUT4 to OUTPUTXBAR1	GPIO24 / J5-44/45	GPIO10_EQEP1A / J2-14
PTO-QEP-B	Tile1 OUT5 to OUTPUTXBAR2	GPIO3 / J4-37	GPIO11_EQEP1A / J2-15
PTO-QEP-I	Tile1 OUT2, override PWM1-B	GPIO1 / J4-39	GPIO9_EQEPI / J1-7
Test signal 1: PWM ISR / Direction	-	GPIO13 / J8-79	Monitor with Oscilloscope.
Test signal 2: HALT/RUN	Tile1 OUT0, override PWM1-A	GPIO0 / J4-40	Monitor with Oscilloscope.

Table 6-12. F280039C Abs2Qep Output GPIO Mapping

Abs2Qep Output			Test Connections
Abs2Qep Signal	**Routing From CLB to GPIO**	**GPIO / LAUNCHXL-F280039C Pin**	**Connect Abs2Qep Output to:**
PTO-QEP-A	Tile1 OUT4 to OUTPUTXBAR1	GPIO24 / J1-8	GPIO10_EQEP1A / J4-36
PTO-QEP-B	Tile1 OUT5 to OUTPUTXBAR2	GPIO3 / J4-37	GPIO11_EQEP1A / J4-35
PTO-QEP-I	Tile1 OUT2, override PWM1-B	GPIO1 / J4-39	GPIO9_EQEPI / J1-7
Test signal 1: PWM ISR / Direction	-	GPIO13 / J8-79	Monitor with Oscilloscope.
Test signal 2: HALT/RUN	Tile1 OUT0, override PWM1-A	GPIO0 / J4-40	Monitor with Oscilloscope.

6.7 QepOnClb Example

This example implements a simple QEP decoder module on the CLB peripheral. The output of this CLB-based QEP decoder is compared with the output of an eQEP-based QEP decoder. The device generates EPWM signals, which are used as test inputs to simulate QEP-A and QEP-B signals.

6.7.1 Watch Variables

The following watch variables provide QepOnClb information to compare the performance of the eQEP and CLBQEP:

- eqepPosition
- clbqepPosition
- deltaPosition
- maxDeltaPosition
- interruptCount
- sendIndex

During the execution of the example, eqepPosition and clbqepPosition track the position from the eQEP and CLBQEP, respectively. If eqepPosition and clbqepPosition differ from each other, the difference is displayed in the deltaPosition variable. The largest deltaPosition found throughout the program execution is stored in maxDeltaPosition.

6.7.2 Header Pin Connections

In this example, EPWMs are generated by the device and are meant to simulate test QEP signals. These EPWM signals need to be externally routed to both the CLB INPUTXBARs and the on-board eQEP peripheral. The following two tables describe the necessary pin connections that need to be made depending on the device being used.

Note

The EPWM signals serve as test inputs to showcase the functionality of the QepOnClb example. If desired, users can instead route external QEP-A and QEP-B signals to the INPUTXBARs and eQEP peripheral.

The CLB-based QEP decoder module is configured to accept three inputs corresponding to QEP-A, QEP-B, and QEP-I. The QEP-A signal should be routed into INPUTXBAR2, the QEP-B signal should be routed into INPUTXBAR1, and the QEP-I signal should be routed into INPUTXBAR3.

Table 6-13 lists the connections that need to be made to route the EPWM signals to the CLB X-BARs.

Table 6-13. QepOnClb EPWM to CLB INPUTXBAR Connections

Board	EPWMA to INPUTXBAR2 (QEP-A)	EPWMB to INPUTXBAR1 (QEP-B)	GPIO to INPUTXBAR3 (QEP-I)
LAUNCHXL-F280025C	GPIO0 (J4-40) to GPIO8 (J2-12)	GPIO1 (J4-39) to GPIO9 (J1-7)	GPIO2 (J4-38) to GPIO27 (J2-11)
LAUNCHXL-F280039C	GPIO0 (J4-40) to GPIO8 (J2-15)	GPIO1 (J4-39) to GPIO9 (J1-7)	GPIO2 (J4-38) to GPIO27 (J6-59)
LAUNCHXL-F280049C	GPIO10 (J4-40) to GPIO39 (J2-13)	GPIO11 (J4-39) to GPIO40 (J1-4)	GPIO8 (J4-38) to GPIO27 (J6-59)
LAUNCHXL-F28379D	GPIO0 (J4-40) to GPIO18 (J1-4)	GPIO1 (J4-39) to GPIO40 (J5-50)	GPIO2 (J4-38) to GPIO27 (J6-52)
TMDSCNCD28388D	GPIO0 (Pin 49) to GPIO18 (Pin 71)	GPIO1 (Pin 51) to GPIO40 (Pin 89)	GPIO2 (Pin 53) to GPIO27 (Pin 81)

The EPWM signals are routed to the eQEP peripheral for loopback testing. This is done in order to provide a comparison with the CLB-based QEP module.

Table 6-13 lists the connections that need to be made to route the EPWM signals to the eQEP peripheral.

Table 6-14. QepOnClb EPWM to eQEP Connections

Board	EPWMA to eQEP-A	EPWMB to eQEP-B	GPIO to eQEP-I
LAUNCHXL-F280025C	GPIO0 (J4-40) to GPIO25 (J4-31)	GPIO1 (J4-39) to GPIO29 (J1-4/5)	GPIO2 (J4-38) to GPIO23 (J2-13)
LAUNCHXL-F280039C	GPIO0 (J4-40) to GPIO25 (J6-51)	GPIO1 (J4-39) to GPIO29 (J1-4/5)	GPIO2 (J4-38) to GPIO23 (J2-11)
LAUNCHXL-F280049C	GPIO10 (J4-40) to GPIO35 (J1-10)	GPIO11 (J4-39) to GPIO37 (J1-9)	GPIO8 (J4-38) to GPIO59 (J2-11)
LAUNCHXL-F28379D	GPIO0 (J4-40) to GPIO20 (QEP1A)	GPIO1 (J4-39) to GPIO21 (QEP1B)	GPIO2 (J4-38) to GPIO23 (QEP1I)
TMDSCNCD28388D	GPIO0 (Pin 49) to GPIO20 (Pin 68)	GPIO1 (Pin 51) to GPIO21 (Pin 70)	GPIO2 (Pin 53) to GPIO23 (Pin 74)

7 Library Source and Projects

This section describes how to import and rebuild the libraries and provides a description of each API function. Each Code Composer Studio library project includes configuration information for the Configurable Logic Block (CLB). To learn how to modify the CLB's configuration, see the *CLB Tool User's Guide*.

Note

Some APIs work with Pulse Train Inputs (PTI) and others with Pulse Train Outputs (PTO). For simplicity, the examples, libraries, and directory structure make use of the suffix "pto" to identify content belonging to this library.

7.1 Locating the Library Source Code

The PTO APIs and source code can be found in the location shown in Table 7-1.

Table 7-1. Location of PTO Libraries

`C:\ti\c2000\C2000Ware_MotorControl_SDK_[version]`	Default install location for the SDK. (`[SDK]`)
`[SDK]\libraries\position_sensing\pto`	Library base install directory (`[lib_base]`)
`[lib_base]\ccs\[device]`	Code Composer projectspec file for the reference library. Use these projects to re-build the library for each device.
`[lib_base]\lib`	Generated library object files (.lib)
`[lib_base]\source`	Library source code (.c) and CLB configuration (.syscfg) files.
`[lib_base]\include`	Library header file. #include this file in the application project that calls the library.
`[lib_build_dir]\RELEASE\syscfg`	Location of the CLB Tile Diagram. By rebuilding the compiled object, CCS it will regenerate the CLB tile diagram (clb.svg or clb.html). and object (.lib). You can access the .svg or html file through CCS Project Explorer.

7.2 Import and Build the Library Project

To rebuild the API library object, follow this procedure:

1. If not already done, install the required software tools described in Section 6.2.
2. In CCS or higher, click '*Project -> Import CCS Projects…*'.
3. Navigate to the Code Composer Studio (CCS) projectspec directory for your device, see Table 7-1.
4. Select the library project of choice and click '*Finish*'.
5. In the CCS Project Explorer window, expand the selected project and open the file corresponding SysConfig file (for example, “pto_pulsegen.syscfg”).
6. Inspect the configuration of the tile(s) and observe the logical expressions in the LUTs and FSMs, and output LUTs.
7. From the CCS menu, select '*Project -> Build Project*'.
8. At this point, an output object (.lib) is generated and located in the `[lib_base]\lib` folder. This file will be included in PTO example projects. This object is a compiled object of the PTO source files.
9. [Optional] – for instructions on how to run a simulation of the CLB based project, see the *Running the Simulation* section in the *CLB Tool User’s Guide*.

7.3 PTO - PulseGen API

This section details the PulseGen Library functions. For information on locating the source code and rebuilding the library, see Section 7.

PulseGen include file: *pto_pulsegen.h*

Table 7-2. PTO-PulseGen API Functions

Name	Description	Type
pto_pulsegen_reset	Used to reset the pulsegen parameters set by earlier configuration and start a fresh setup. This function needs to be called in case the pulse generation needs to be reset and started again at a later stage.	Initialization time
pto_pulsegen _setupPeriph	Setup for CLB and other interconnect XBARs is performed with this function during system initialization. This function needed to be called after every system reset. No transactions will be performed until the setup peripheral function is called.	Initialization time
pto_pulsegen _startOperation	This function will initiate the pulse generation on the interface. To be called after pto_pulsegen_setupPeriph. Performs the transaction set up by earlier function. Note that the setup up and start operation are separate function calls. You can setup the peripherals when needed and start the actual pulse generation using this function call, as needed, at a different time.	Run time
pto_pulsegen _runPulseGen	A runtime function to be called periodically for dynamically configuring and changing the pulse generation requirements as needed by the application. This function needs to be called periodically with appropriate parameters like the number of pulses, period, duration, and so forth. Details in the later section.	Run time

7.3.1 pto_pulsegen_runPulseGen

Description

A runtime function to be called periodically for dynamically configuring and changing the pulse generation requirements as required by the application. This function must be called periodically with appropriate parameters like the number of pulses, period, duration, and more.

Definition

```
uint16_t pto_pulsegen_runPulseGen(
        uint32_t pulseLo,
        uint32_t pulseHi,
        uint32_t ptoActivePeriod,
        uint32_t ptoFullPeriod,
        uint32_t ptoInterruptTime,
        uint16_t ptoDirection,
        uint16_t run
        );
```

Parameters

Input:

- pulseLo – Low pulse width
- pulseHi – High pulse width
- ptoActivePeriod – Period the pulses are sent out; less than ptoFullPeriod
- ptoFullPeriod – Full PTO period
- ptoInterruptTime – Time when that the interrupt is generated to the CPU
- ptoDirection – Direction output; latched as it is on direction output at the beginning of new period
- run – Value indicting 1-run and 0-stop. Sampled at the beginning of the new period to determine to continue or halt the pulse generation

Return:

- Val – If the function is executed successfully, the function will return ptoFullPeriod as the return value

Usage

In pto_pulsegen.c, a sample configuration function called pto_setOptions is provided as an example to assist with the pto_pulsegen_runPulseGen function and to perform the intermediate calculations. See the following code sample calculation that illustrates how various parameters for this function can be generated. For more details, see the pto_pulsegen.c and pto_pulsegen.h files.

```
uint32_t pto_setOptions(
        uint32_t numPulses, //number of pulses needed to be generated in next period
        uint32_t Period,        // PTO period in clock cycles
        uint32_t ptoInterruptTime, // Interrupt generation time
        uint16_t ptoDirection,    // Direction output
        uint16_t run)        //run-stop condition.
{
    uint32_t pulseFreq, reminder;
    uint32_t pulseLo;
    uint32_t pulseHi;
    uint32_t ptoActivePeriod;
    uint32_t ptoFullPeriod;
    pulseFreq = Period / numPulses;
    reminder = Period - (pulseFreq * numPulses);
    pulseLo = (pulseFreq/2 );
    pulseHi = pulseFreq;
    ptoActivePeriod = (pulseFreq * numPulses);
    ptoFullPeriod = Period;
    pto_pulsegen_runPulseGen(
            pulseLo,
            pulseHi,
            ptoActivePeriod,
            ptoFullPeriod,
            ptoInterruptTime,
            ptoDirection,
            run);
    return(reminder);
}
```

7.3.2 pto_startOperation

Description

This function initiates the pulse generation. This function must be called after pto_pulsegen_setupPeriph. Hence, the pto_pulsegen_startOperation kick starts the pulse generation that was set up earlier.

Note

The setup and start operations are separate function calls. Users can set up the transfer and start the pulse generation by using this function call, as required, at a different time.

Definition

```
void pto_pulsegen_startOperation(void);
```

Parameters

Input: none

Return: none

Usage

Example code:

```
pto_initPulsegen ();
SysCtl_delay (800L);
pto_pulsegen_startOperation ();
retval1 = pto_pulsegen_runPulseGen (7, 15, 960, 990, 500, 1, 1);
```

7.3.3 pto_pulsegen_setupPeriph

Description

This function performs the setup for the CLB and other interconnect XBARs during system initialization. This function must be called after every system reset. No transactions will be performed until the setup peripheral function is called.

Definition

```
void pto_pulsegen_setupPeriph (void);
```

Parameters

Input: none

Return: none

Usage

Example code:

```
pto_pulsegen_setupPeriph();
```

7.3.4 pto_pulsegen_reset

Description

This function resets the pulsegen parameters set by the earlier configuration (PulseGen function calls) and starts a new setup. This function must be called in case the pulse generation must be reset and started again at a later stage.

Definition

```
void pto_pulsegen_reset (void);
```

Parameters

Input: none

Return: none

Usage

Example code:

```
pto_pulsegen_reset();
```

7.4 PTO - QepDiv API

This section details the QepDiv Library functions. For information on locating the source code and rebuilding the library, see Section 7.

QepDiv include file: *pto_qepdiv.h*

Table 7-3. PTO-QepDiv API Functions

Name	Description	Type
pto_qepdiv_reset	Used to reset the qepdiv parameters set by earlier configuration and start a fresh setup. This function needs to be called in case the pulse generation needs to be reset and started again at a later stage.	Initialization time
pto_qepdiv _setupPeriph	Setup for CLB and other interconnect XBARs is performed with this function during system initialization. This function needed to be called after every system reset. No transactions will be performed until the setup peripheral function is called.	Initialization time
pto_qepdiv _startOperation	This function will initiate the pulse generation on the interface. To be called after pto_qepdiv_setupPeriph. Performs the transaction set up by earlier function. Note that the setup up and start operation are separate function calls. User can setup the peripherals when needed and start the actual pulse generation using this function call, as needed, at a different time.	Run time
pto_qepdiv_config	This function configures the divider, the divider value cannot be changed dynamically. User needs to reset the module using pto_qepdiv_reset before reconfiguring the functionality.	Run time

7.4.1 pto_qepdiv_config

Description

This function configures the divider. The divider value cannot be changed dynamically. Users must reset the module using pto_qepdiv_reset before reconfiguring the functionality.

Definition

```
pto_qepdiv_config(uint16_t divider, uint16_t indexWidth);
```

Parameters

Input:

- Divider – Value of the divider
- Index width – Number of cycles for which the index pulse output is kept on

Return:

- Val – If the function is executed successfully, it will return ptoFullPeriod as the return value

Usage

Example code:

```
retval1 = pto_qepdiv_config(4, 10);
    pto_qepdiv_startOperation(1);
```

7.4.2 pto_startOperation

Description

This function initiates the pulse generation. This function must only be called after pto_qepdiv_setupPeriph. Hence, the pto_qepdiv_startOperation function kick starts the pulse generation that was set up earlier.

Note

The setup and start operations are separate function calls. Users can set up the transfer and start the pulse generation by using this function call, as required, at a different time.

Definition

```
void pto_qepdiv_startOperation(uint16_t run);
```

Parameters

Input (parameters to be passed to start or stop the function):

- Start = 1
- Stop = 0

Return: none

Usage

Example code:

```
retval1 = pto_qepdiv_config(4, 10);
    pto_qepdiv_startOperation(1);
```

7.4.3 pto_qepdiv_setupPeriph

Description

Setup for the CLB and other interconnect XBARs is performed with the pto_qepdiv_setupPeriph function during system initialization. This function must be called after every system reset. No transactions will be performed until the setup peripheral function is called.

Definition

```
void pto_qepdiv_setupPeriph (void);
```

Parameters

Input: none

Return: none

Usage

Example code:

```
pto_qepdiv_setupPeriph();
```

7.4.4 pto_qepdiv_reset

Description

Used to reset the qepdiv parameters set by earlier configurations and to begin a fresh setup. This function must be called in case the pulse generation must be reset and started again at a later stage.

Definition

```
void pto_qepdiv_reset (void);
```

Parameters

Input: none

Return: none

Usage

Example code:

```
pto_qepdiv_reset();
```

7.5 PTO - Abs2Qep API

This section details the Abs2Qep Library functions. For information on locating the source code and rebuilding the library, see Section 7.

Abs2Qep include file: *pto_abs2qep.h*

Table 7-4. PTO-Abs2Qep API Functions

Name	Description	Type
pto_abs2eqep _setupPeriph	Setup for CLB and other interconnect XBARs is performed with this function during system initialization. This function to be called after every system reset. No transactions will be performed until the setup peripheral function is called.	Initialization time
pto_abs2qep _translatePosition	Translates a change in absolute position into an equivalent change in incremental position. This function configures the CLB to generate: the number of QCLKs, the QCLK frequency, and the QEP-I pulse. The parameters used for the translation can be configured in the library header file. This function loads the configuration into the HLC FIFO.	Run Time
pto_abs2qep _runPulseGen	This function will initiate the pulse generation on the interface. To be called after pto_abs2qep_translatePosition has setup the CLB HLC FIFO. Note that the setup up and start operation are separate function calls.	Run Time

7.5.1 Abs2Qep API Configuration

The library header file, `pto_abs2qep.h`, contains parameters which can be modified to configure the library for different encoders and position sample rates. Parameters include:

- Position sampling period
- Maximum motor revolutions per minute (RPM)
- Absolute encoder resolution
- Incremental encoder lines per revolution

7.5.2 pto_abs2qep_runPulseGen

Description

Function called during runtime to start a new PTO. This function checks that the previous PTO has completed before starting a new PTO.

Note

The setup and start operations are separate function calls. The setup (pto_abs2qep_translatePosition) must be called before this function. You can set up the transfer and start the pulse generation by using this function call, as required, at a different time.

Definition

```
    void
    pto_abs2qep_runPulseGen(
        uint16_t ptoDirection
    );
```

Parameters

Input:

- ptoDirection: Direction of the PTO. This determines which signal leads QEP-A or QEP-B.

Return: none

Usage

```
// Call to sample a new absolute position
    ....
// Translate change from previous position to PTO configuration
    ptoDirection = pto_abs2qep_translatePosition(absolutePosition);
    ....
// Start the last configuration
    pto_abs2qep_runPulseGen(ptoDirection);
```

7.5.3 pto_abs2qep_setupPeriph

Description

This function performs the setup for the CLB and XBAR interconnect during system initialization. This function must be called after every system reset. No transactions will be performed until the setup peripheral function is called.

Definition

```
void pto_abs2qep_setupPeriph(void);
```

Parameters

Input: none

Return: none

Usage

```
pto_abs2qep_setupPeriph();
...
//
// GPIO and other system peripheral configuration
//
```

7.5.4 pto_abs2qep_translatePosition

Description

This function translates a change in absolute position into an equivalent PTO configuration to be loaded into the CLB FIFO. The information includes:

- Number of QCLKs required to generate the QEP-A and QEP-B pulses
- If crossing zero, the QCLK edge where QEP-I should be driven high and low
- The number of CLB clocks between each QCLK
- The direction of the position change.

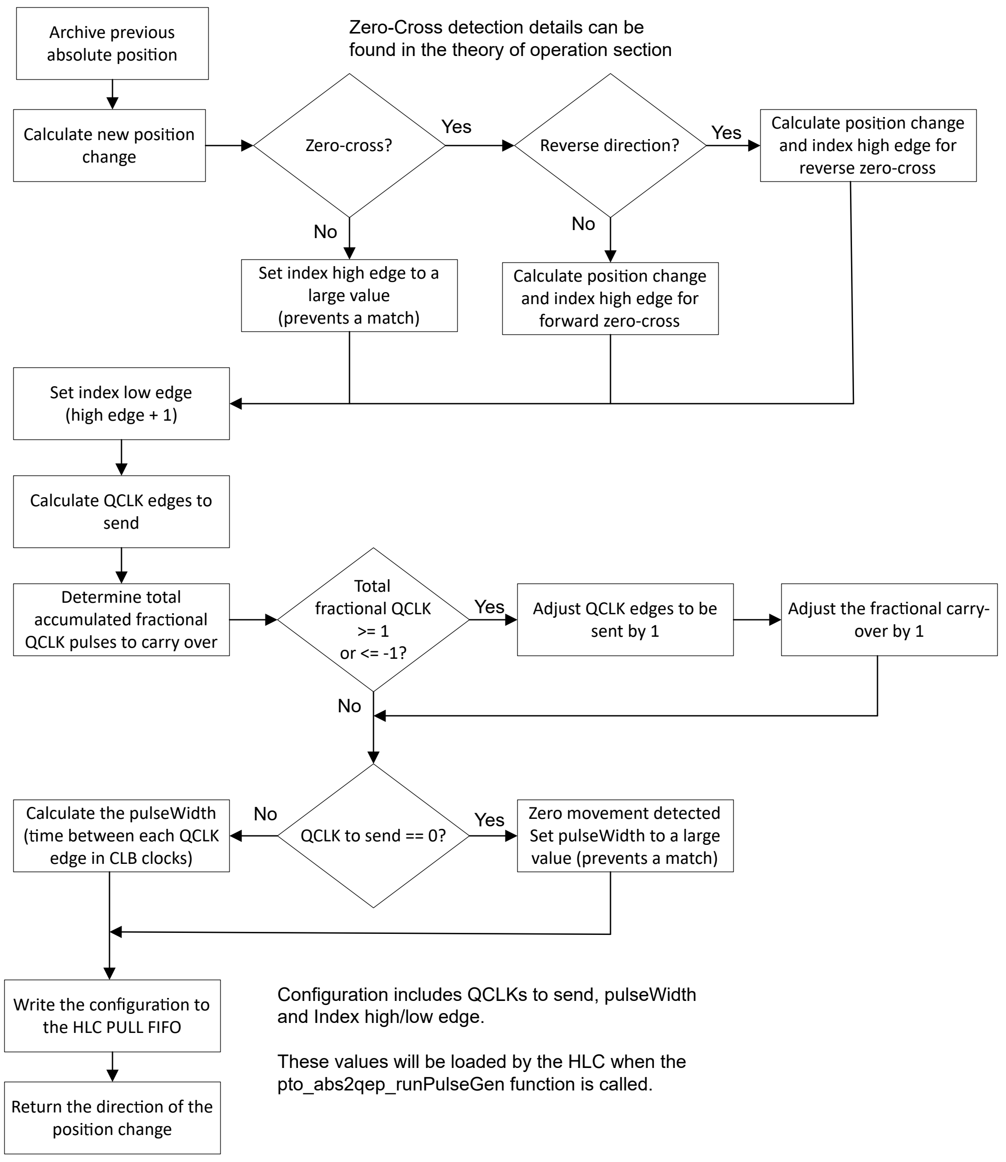

Archive previous absolute position
Zero-Cross detection details can be found in the theory of operation section
Calculate new position change
Zero-cross?
Yes
Reverse direction?
Yes
Calculate position change and index high edge for reverse zero-cross
No
No
Set index high edge to a large value (prevents a match)
Calculate position change and index high edge for forward zero-cross
Set index low edge (high edge + 1)
Calculate QCLK edges to send
Determine total accumulated fractional QCLK pulses to carry over
Total fractional QCLK >= 1 or <= -1?
Yes
Adjust QCLK edges to be sent by 1
Adjust the fractional carry-over by 1
No
Calculate the pulseWidth (time between each QCLK edge in CLB clocks)
No
QCLK to send == 0?
Yes
Zero movement detected Set pulseWidth to a large value (prevents a match)
Write the configuration to the HLC PULL FIFO
Configuration includes QCLKs to send, pulseWidth and Index high/low edge.
These values will be loaded by the HLC when the pto_abs2qep_runPulseGen function is called.
Return the direction of the position change

Figure 7-1. Abs2Qep Translate Function

Definition

```
    uint16_t
        pto_abs2qep_translatePosition(
            uint32_t positionNew
        );
```

Parameters

Input:

- positionNew - The new absolute position as sampled by the system. The translate function compares this value to the previous sample to determine the change in position.

Return: ptoDirection - Indicates the direction of the PTO.

- PTO_ABS2QEP_CLOCKWISE_PTO
- PTO_ABS2QEP_COUNTERCLOCKWISE_PTO

Note

This function loads the PTO configuration directly into the HLC PULL FIFO.

Usage

```
// Call to sample a new absolute position
    ....
// Translate change from previous position to PTO configuration
    ptoDirection = pto_abs2qep_translatePosition(absolutePosition);
    ....
// Start the last configuration
    pto_abs2qep_runPulseGen(ptoDirection);
```

7.6 PTO - QepOnClb API

This section details the QepOnClb Library functions. For information on locating the source code and rebuilding the library, see Section 7.

QepOnClb include file: *pto_qeponclb.h*

Table 7-5. PTO-QepOnClb API Functions

Name	Description	Type
pto_qeponclb_setupPeriph	Setup for CLB and other interconnect XBARs is performed with this function during system initialization. This function needs to be called after every system reset. No decoding can be done until the setup peripheral function is called.	Initialization Time
pto_qeponclb_initCLBQEP	Helper function that initializes the CLB peripheral to simulate a QEP decoder. This function is used in the pto_qeponclb_setupPeriph function as a part of the CLBQEP system initialization.	Initialization Time
pto_qeponclb_configMaxCounterPos	This function configures the maximum counter value for the CLBQEP and loads it as a parameter in the CLB tile.	Initialization Time
pto_qeponclb_enableCLBQEP	Used to enable the CLBQEP counter to begin decoding. The CLBQEP counter needs to be initialized prior to enabling.	Run Time
pto_qeponclb_resetCLBQEP	Used to reset the CLBQEP parameters set by earlier configuration. This function needs to be called in case the counter needs to be reset and started again at a later instance.	Run Time
pto_qeponclb_getCounterVal	This function captures the current value of counter 0 from the CLB.	Run Time
pto_qeponclb_getCLBQEPPos	This function is to be called when wanting to capture the current position of the CLB-based QEP peripheral from the CLB.	Run Time
pto_qeponclb_clearFIFOptr	This function is to be called when needing to clear the FIFO pointer from the CLB. This function needs to be called subsequently after reading a FIFO value pushed by the HLC so that the next PUSH will be in the correct position.	Run Time

7.6.1 pto_qeponclb_setupPeriph

Description

This function sets up the CLB and other interconnect XBARs during system initialization to allow for using the CLBQEP peripheral. This function needs to be called after every system reset. No QEP decoding can be done until the setup peripheral function is called.

Definition

```
void pto_qeponclb_setupPeriph(uint32_t maxPosition);
```

Parameters

Input:

- maxPosition - maximum counter value for the CLBQEP

Return: none

Usage

Example code:

```
uint32_t max_position;
max_position = 500;
pto_qeponclb_setupPeriph(max_position);
```

7.6.2 pto_qeponclb_initCLBQEP

Description

This function is a helper function that initializes the CLB peripheral to simulate a QEP decoder. This function is used in the pto_qeponclb_setupPeriph function as a part of the CLBQEP system initialization.

Definition

```
void pto_qeponclb_initCLBQEP(uint32_t maxPosition);
```

Parameters

Input:

- maxPosition - maximum counter value for the CLBQEP

Return: none

Usage

Example code:

```
uint32_t max_position;
max_position = 500;
pto_qeponclb_initCLBQEP(max_position);
```

7.6.3 pto_qeponclb_configMaxCounterPos

Description

This function configures the maximum counter value for the CLBQEP and loads this value as a parameter in the CLB tile.

Definition

```
void pto_qeponclb_configMaxCounterPos(uint32_t clbBase, uint32_t maxPosition);
```

Parameters

Input:

- clbBase – Base address of CLB tile
- maxPosition - maximum counter value for the CLBQEP

Return: none

Usage

Example code:

```
uint32_t max_position;
max_position = 500;
pto_qeponclb_configMaxCounterPos(max_position);
```

7.6.4 pto_qeponclb_enableCLBQEP

Description

This function is used to enable the CLBQEP counter to begin decoding. The CLBQEP counter needs to be initialized prior to enabling.

Definition

```
void pto_qeponclb_enableCLBQEP(uint32_t clbBase, uint32_t enableCapture);
```

Parameters

Input:

- clbBase – Base address of CLB tile
- enableCapture - GPREG bit corresponding to enable CLBQEP

Return: none

Usage

Example code:

```
pto_qeponclb_enableCLBQEP(0x00003000, (1 << 2));
```

7.6.5 pto_qeponclb_resetCLBQEP

Description

This function is used to reset the CLBQEP parameters set by earlier configuration. This function needs to be called in case the counter needs to be reset and started again at a later instance.

Definition

```
void pto_qeponclb_resetCLBQEP(uint32_t clbBase, uint32_t resetCounter);
```

Parameters

Input:

- clbBase – Base address of CLB tile
- resetCounter - GPREG bit corresponding to reset CLBQEP

Return: none

Usage

Example code:

```
pto_qeponclb_resetCLBQEP(0x00003000, (1 << 0));
```

7.6.6 pto_qeponclb_getCounterVal

Description

This function captures the current value of counter 0 from the CLB.

Definition

```
uint32_t pto_qeponclb_getCounterVal(uint32_t clbBase);
```

Parameters

Input:

- clbBase – Base address of CLB tile

Return:

- Val - The function will return counterVal as the return value

Usage

Example code:

```
#Define CLB1_BASE 0x00003000U
uint32_t retVal1;
retVal1 = pto_qeponclb_getCounterVal(CLB1_BASE);
```

7.6.7 pto_qeponclb_getCLBQEPPos

Description

This function is to be called when wanting to capture the current position of the CLB-based QEP peripheral from the CLB.

Definition

```
uint32_t pto_qeponclb_getCLBQEPPos(uint32_t clbBase);
```

Parameters

Input:

- clbBase – Base address of CLB tile

Return:

- Val - The function will return clbqepPos as the return value

Usage

Example code:

```
#Define CLB1_BASE 0x00003000U
uint32_t retVal1;
retVal1 = pto_qeponclb_getCLBQEPPos(CLB1_BASE);
```

7.6.8 pto_qeponclb_clearFIFOptr

Description

This function is to be called when needing to clear the FIFO pointer from the CLB. This function needs to be called subsequently after reading a FIFO value pushed by the HLC so that the next PUSH will be in the correct position.

Definition

```
void pto_qeponclb_clearFIFOptr(uint32_t clbBase);
```

Parameters

Input:

- clbBase – Base address of CLB tile

Return: none

Usage

Example code:

```
pto_qeponclb_clearFIFOptr(0x00003000);
```

8 Using the Reference APIs in Projects

The following sections describe the steps required to include one or more of the libraries into your project. These include:

- Adding the library header file
- Updating Code Composer Studio options to link in the library
- Modification of the internal routing to, or from, the CLB
- Initialization steps required to invoke the functionality of the API

Note

Some APIs work with Pulse Train Inputs (PTI) and others with Pulse Train Outputs (PTO). For simplicity, the examples, libraries, and directory structure make use of the suffix "pto" to identify content belonging to this library.

8.1 Adding PTO Support to a Project

Use the following instructions to add the PTO APIs to a project.

Note

The exact location may vary depending on where C2000Ware_MotorControl_SDK is installed and which other libraries the project is using.

1. Include the PTO header file in the application i.e. `{ProjectName}.h`.

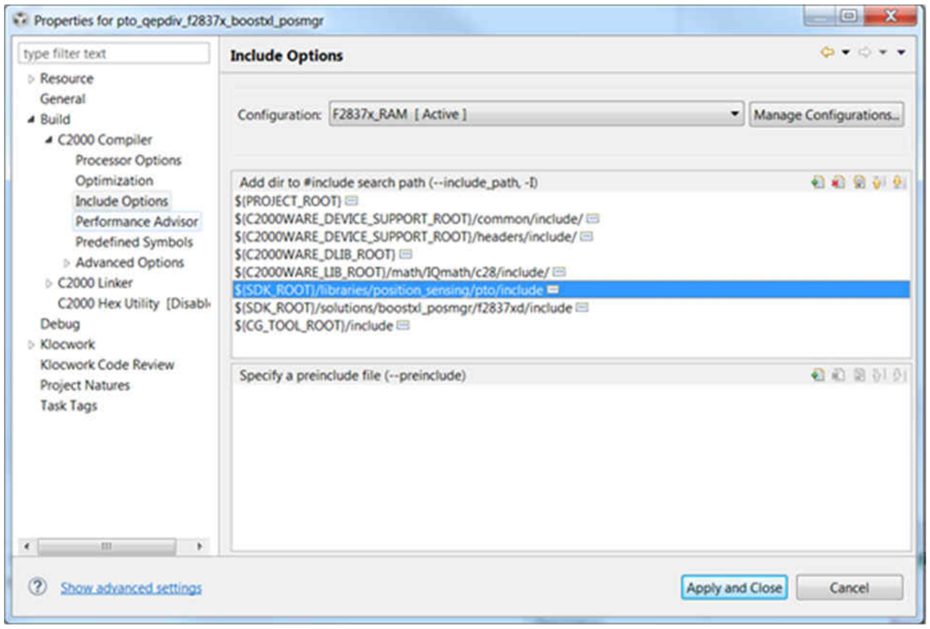
```
#include "pto_pulsegen.h"
#include "pto_qepdiv.h"
#include "pto_abs2qep.h"
#include "pto_qeponclb.h"
```

2. In Code Composer Studio (CCS), right click on the project and navigate to *Project Properties → Build → C2000 Compiler → Include Options*
   a. Add the header file directory to the '*#include search path*' (see Figure 8-1)

      The path for PTO header files is `${SDK_ROOT}\libraries\position_sensing\pto\include`.

Note

${SDK_ROOT} is a varaible used by CCS to indicate the install location of the SDK. Its definition can be viewed under *Project Properties → Resource → Linked Resources*.



Configuration: F2837x_RAM [Active]

Add dir to #include search path (--include_path, -I)

```
${PROJECT_ROOT}
${C2000WARE_DEVICE_SUPPORT_ROOT}/common/include/
${C2000WARE_DEVICE_SUPPORT_ROOT}/headers/include/
${C2000WARE_DLIB_ROOT}
${C2000WARE_LIB_ROOT}/math/IQmath/c28/include/
${SDK_ROOT}/libraries/position_sensing/pto/include
${SDK_ROOT}/solutions/boostxl_posmgr/f2837xd/include
${CG_TOOL_ROOT}/include
```

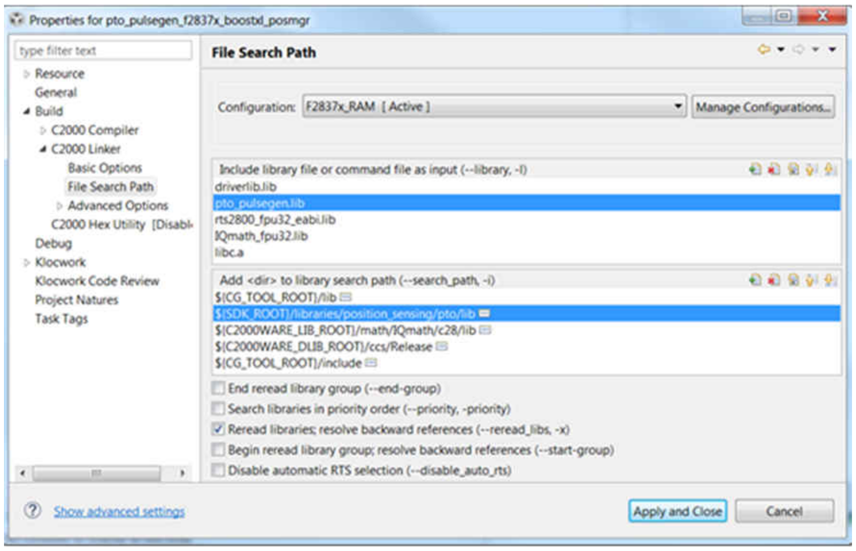
Specify a preinclude file (--preinclude)

Figure 8-1. Compiler Include Options for Projects Using PTO Reference APIs

3. Add the compiled library file to the project:
   a. Right click on the project name in the Project Explorer window
   b. Navigate to '*Project Properties → Build → C2000 Linker → File Search Path*'
   c. Add the library directory to the '*library search path*'
   d. Add the name of the library to the '*Include library file*'
   e. Click '*Apply and Close*'

   The PTO compiled library object files are located at: `[C2000Ware_MotorControl_SDK]\libraries\position_sensing\pto\lib`.

An example is shown in Figure 8-2 and Figure 8-3 show the changes to the linker options that are required to include the PTO APIs compiled object file.



File Search Path

Configuration: F2837x_RAM [Active]

Include library file or command file as input (--library, -l)

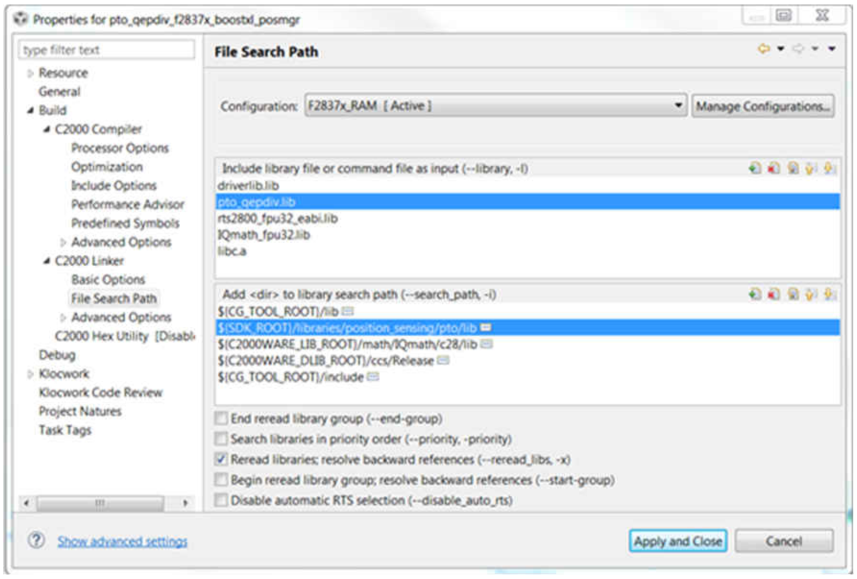
```
driverlib.lib
pto_pulsegen.lib
rts2800_fpu32_eabi.lib
IQmath_fpu32.lib
libc.a
```

Add <dir> to library search path (--search_path, -i)

```
${CG_TOOL_ROOT}/lib
${SDK_ROOT}/libraries/position_sensing/pto/lib
${C2000WARE_LIB_ROOT}/math/IQmath/c28/lib
${C2000WARE_DLIB_ROOT}/ccs/Release
${CG_TOOL_ROOT}/include
```

- [ ] End reread library group (--end-group)
- [ ] Search libraries in priority order (--priority, -priority)
- [x] Reread libraries; resolve backward references (--reread_libs, -x)
- [ ] Begin reread library group; resolve backward references (--start-group)
- [ ] Disable automatic RTS selection (--disable_auto_rts)

Figure 8-2. C2000™ Linker Options – PulseGen



File Search Path

Configuration: F2837x_RAM [Active]

Include library file or command file as input (--library, -l)

```
driverlib.lib
pto_qepdiv.lib
rts2800_fpu32_eabi.lib
IQmath_fpu32.lib
libc.a
```

Add <dir> to library search path (--search_path, -i)

```
${CG_TOOL_ROOT}/lib
${SDK_ROOT}/libraries/position_sensing/pto/lib
${C2000WARE_LIB_ROOT}/math/IQmath/c28/lib
${C2000WARE_DLIB_ROOT}/ccs/Release
${CG_TOOL_ROOT}/include
```

- [ ] End reread library group (--end-group)
- [ ] Search libraries in priority order (--priority, -priority)
- [x] Reread libraries; resolve backward references (--reread_libs, -x)
- [ ] Begin reread library group; resolve backward references (--start-group)
- [ ] Disable automatic RTS selection (--disable_auto_rts)

Figure 8-3. C2000 Linker Options – QepDiv

Note

The exact location may vary depending on where C2000Ware_MotorControl_SDK is installed and which other libraries the project is using.

8.2 Routing To and From the CLB

The next step is to understand the routing to/from the CLB and how it will integrate into your project's requirements.

The provided examples route signals between the CLB and specific GPIO pins. The input/output routing is described in the Section 6 Your application may, however, require routing signals to different pins or a different XBAR or a different peripheral. The specific modifications differ from case-to-case.

In some cases, a simple change to the application code is all that is required.

The routing is through INPUTXBARx to the CLB AUXSIGx global MUX. You want to change the input to a different GPIO. This can be accomplished by changing which GPIO is connected to INPUTXBARx.

In other cases, the CLB library may need to be changed or the design moved to another tile. For example:

If a Tile output is not able to use the OUTPUTXBAR it may instead override a peripheral output. Which peripherals are available to each tile differs. For example ePWM1 can be overridden by Tile 1, ePWM2 by Tile2, and so forth. A change here may require routing the output to a different OUTLUT or to a different tile.

8.3 Initialization Steps

The following section describes the specific initialization steps required to use the PTO library. The examples should be used as a reference.

8.3.1 PTO-PulseGen API Initalization

The following steps are required for initialization and proper function of the PTO PulseGen API functions.

1. Initialize and set up the peripheral configuration by calling the `pto_pulsegen_setupPeriph()` function.
2. Set up the GPIOs required for configuration. For more information, see `pto_setupGPIO`.
3. To set the pulse generation configuration, see the `pto_setOptions` function.
4. `ptoISR` is used as the primary interrupt service routine (ISR). To see how to update the PTO configuration, see this ISR.

8.3.2 PTO-QepDiv API Initialization

The following steps are required for initialization and proper function of the PTO QepDiv API functions.

1. Initialize and set up the peripheral configuration by calling the `pto_qepdiv_setupPeriph()` function.
2. Set up the GPIOs required for configuration.
3. To set the configuration, see the `pto_qepdiv_config()` function.
4. Call `pto_qepdiv_startOperation()` to start the QepDiv configuration.

8.3.3 PTO-Abs2Qep API Initialization

The following steps are required to initialize and configure the PTO Abs2Qep API functions.

1. Review the library headerfile, `pto_abs2qep.h`, and update any configuration information required to match your system. If modifications are made, then rebuild the library as described in Section 7. The configuration includes:
    a. Resolution of the drive
    b. Max RPM of the drive
    c. Lines per revolution of the incremental encoder
    d. Position sampling rate
2. Setup the GPIO and routing to/from the CLB as described in Section 8.2.
3. Setup the CLB, see `pto_abs2qep_setupPeriph()`. Note at the end of this function, a call is made to setup the CLB with position 0.

```
    pto_abs2qep_translatePosition(0);
    pto_abs2qep_runPulseGen(PTO_ABS2QEP_CLOCKWISE_PTO);
```

4. Configure a timer to start an ISR when the absolute position is to be sampled. The example uses ePWM3.
5. In the sampling ISR do the following (see `pto_EPWM3ISR`):
    a. Start the previous PTO translation: `pto_abs2qep_runPulseGen(ptoDirection)`
    b. Sample a new absolute position `AbsolutePositionNext = <application dependent function>()`
    c. Translate the next PTO. `ptoDirection = pto_abs2qep_translatePosition(absolutePositionNext)` This translation will be run the next time the ISR is run.

8.3.4 PTO-QepOnClb API Initialization

The following steps are required to initialize and configure the PTO QepOnDiv API functions.

1. Initialize and set up the peripheral configuration by calling the `pto_qeponclb_setupPeriph()` function.
2. Configure the maximum counter position by passing the value as an argument into both the `pto_qeponclb_setupPeriph()` and `pto_qeponclb_configMaxCounterPos()` functions.
3. Set up the GPIOs required for configuration.
4. `epwmISR` is used as the primary interrupt service routine (ISR). To see how to update the PTO configuration, see this ISR.
5. Call `pto_qeponclb_enableCLBQEP()` to enable the CLBQEP for decoding.

9 References

- Texas Instruments: *CLB Tool User's Guide*
- Texas Instruments: *C2000™ Piccolo™ F28004x Series LaunchPad™ Development Kit*

Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from Revision C (May 2021) to Revision D (January 2022)	Page
• Added QEP implementation	2
• Add new Section 5	21
• Added new Section 5.1	22