Application Note

Advanced Timer Techniques in MSPM0

Gaurang Gupta, Sal Ye, Baibhav Tripathy, Luke Ledbetter

ABSTRACT

Advanced Timer (TIMA) peripherals in MSPM0 Devices are designed to meet the demands of modern embedded applications. This application note explores the multifaceted functionalities of the Advanced Timer modules. It delves into their application in scenarios ranging from software-based bit-banging communication protocols to sophisticated feedback-controlled PWM systems. By leveraging features such as Input Capture, PWM output, and synchronization with analog peripherals like ADCs and comparators, developers can implement responsive and efficient control loops. Through detailed examples and use-case analyses, this note demonstrates how the Advanced Timer architecture facilitates the development of advanced embedded solutions, adaptive power regulation, and real-time signal processing. The insights provided aim to assist engineers in harnessing the full potential of the Timer peripherals for innovative application designs. Project collateral detailed in this application note can be downloaded from Advanced Timer Techniques in MSPM0.

Table of Contents

Trademarks

1 Introduction

This application note explores the versatile capabilities of the Advanced Timer (TIMA) module in MSPM0 microcontrollers, demonstrating advanced timing and control functionalities crucial for modern embedded applications. The document presents comprehensive implementation strategies for various timer-based solutions, ranging from motor control to communication protocol emulation.

Key applications detailed in this guide include motor-centric PWM configurations with idle-low state maintenance, and sophisticated PWM generation techniques featuring synchronized outputs with precise phase control. The note demonstrates two distinct approaches to achieving phase-shifted PWM: utilizing the phase load functionality and leveraging secondary capture-compare channels. For applications requiring custom communication protocols, bit-banging implementations have been explored, specifically focusing on UART transmitter and receiver emulation using TIMA resources.

Additionally, the document covers critical timing control aspects such as coordinated delayed starts between multiple timer instances, feedback-based PWM generation for closed-loop systems, and hardware-triggered timer control. The guide concludes with techniques for dynamic PWM parameter modifications, enabling real-time adjustments of duty cycle and period values. Each implementation is presented with practical examples, timing diagrams, and optimized configuration settings specific to MSPM0 devices.

2 Idle-Low State: PWM Output Channel Low-state Configuration

TIMA provides a mechanism where all Capture/Compare (CC) channels have a complementary channel. This complementary channel can generate an output that is an inversion of the main CC channel. For Motor applications both the output channels must be low initially.

When CTRCTL.EN bit is zero, the CC chanel output depends on OCTL.CCPIV. If both the CC and the Complementary channels are to be kept low, the solution is to keep the IOMUX disabled for the Complementary channel and to be enabled with below cases:

- In edge-align mode, keep IOMUX disable for Complementary CC channel until CTRCTL.EN is enabled
- In central-aligned mode, keep IOMUX disable for Complementary CC channel until the CTRCTL.EN is enabled and CC channel met the first CC event

```
/* Configure Complementart Channel's IOMUX in PULL_DOWN state initially */
SYSCONFIG_WEAK void SYSCFG_DL_GPIO_init(void)
{
    DL_GPIO_initPeripheralOutputFunction(GPIO_PWM_0_C0_IOMUX,GPIO_PWM_0_C0_IOMUX_FUNC);
    DL_GPIO_enableOutput(GPIO_PWM_0_C0_PORT, GPIO_PWM_0_C0_PIN);
    DL_GPIO_setDigitalInternalResistor(GPIO_PWM_0_C0_CMPL_IOMUX,DL_GPIO_RESISTOR_PULL_DOWN);
}

/* Enable Counter with CTRCL.EN bit and configure the IOMUX back to PWM mode */
int main(void)
{
    SYSCFG_DL_init();
    DL_TimerA_startCounter(PWM_0_INST);
    DL_GPIO_initPeripheralOutputFunction(GPIO_PWM_0_C0_CMPL_IOMUX,GPIO_PWM_0_C0_CMPL_IOMUX_FUNC);
    while (1) {
        __WFI();
    }
}
```

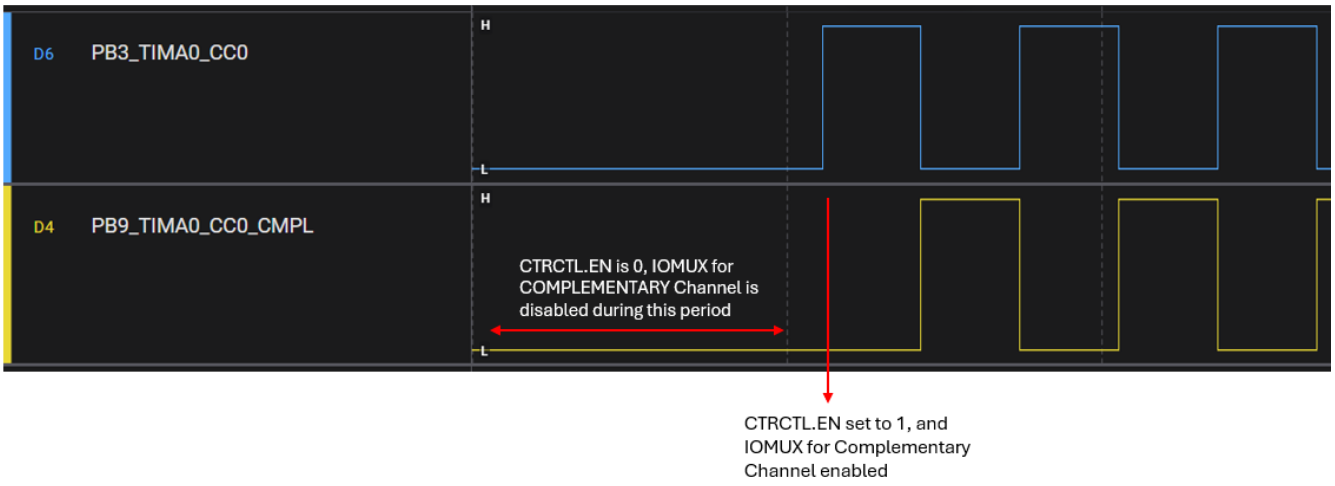

Figure 2-1. PWM Waveform with CC Outputs Set as Idle-Low

Additionally, TIMA supports a feature where Software based configuration can be used to over-ride the output. SWFRCACT_CMPL (Software Force on Complimentary channel) field in CCACT can be used to override the CC Output as LOW. As per the application the SWFRCACT_CMPL field can be cleared using external events or in the interrupt service routine.

Note

SWFRCACT_CMPL takes effect after CTRCTL.EN is set.

3 Asymmetric PWM: Dual Synchronized PWM Generation with Phase Shift Control

TIMA can generate two synchronized center-aligned PWM signals with a controlled phase shift for

- Motor Control to minimize torque ripple and to improve smoothness in low-speed operations
- Digital Power Supplies/DC-DC converters to reduce switching losses and EMI
- AC Phase Controlled Circuits for enabling precise phase-angle-based power control

3.1 Using Phase Load Functionality

To generate asymmetric PWM signals, the phase load feature in TIMA can be used. In this methodology, two timer instances are used, one of which must be TIMA (only TIMA supports the Phase Load feature). Both the TIMER instances are synchronized using the Cross-Triggering feature. When a Timer is Cross-triggered, if Phase Load functionality is enabled, Phase Load comes into action, and the timer starts counting from the Phase Load value. This configurable Phase Load value enables a controlled phase shift between the two PWMs. The configuration is described in the following sections.

3.1.1 Configuration for the Primary Timer (Main Timer)

The primary timer is the one that generates the cross-trigger signal to synchronize other timers:

- Enable the Cross-Trigger Output functionality by setting the TIMx.CTTRIGCTL.CTEN bit.
- Configure the cross-trigger generation event by selecting the appropriate trigger source.
  - For a Software based Cross-Trigger, set the TIMx.CTTRIG.TRIG bit.
  - For a Hardware based Cross-Trigger, select the source for the trigger using the TIMx.CTTRIGCTL.EVTCTTRIGSEL and enable the hardware trigger by setting the TIMx.CTTRIGCTL.EVTCTEN.
- Timer also offers the feature to self-trigger itself

```
/* Configuration for Main Timer to generate Cross Trigger */
static const DL_TimerA_ClockConfig gPWM_0ClockConfig = {
    .clockSel = DL_TIMER_CLOCK_BUSCLK,
    .divideRatio = DL_TIMER_CLOCK_DIVIDE_1,
    .prescale = 0U
};

static const DL_TimerA_PWMConfig gPWM_0Config = {
    .pwmMode = DL_TIMER_PWM_MODE_CENTER_ALIGN,
    .period = 1600,
    .isTimerWithFourCC = false,
    .startTimer = DL_TIMER_STOP,
};

SYSCONFIG_WEAK void SYSCFG_DL_PWM_0_init(void) {

    DL_TimerA_setClockConfig(
        PWM_0_INST, (DL_TimerA_ClockConfig *) &gPWM_0ClockConfig);

    DL_TimerA_initPWMMode(
        PWM_0_INST, (DL_TimerA_PWMConfig *) &gPWM_0Config);

    // Set Counter control to the smallest CC index being used

DL_TimerA_setCounterControl(PWM_0_INST,DL_TIMER_CZC_CCCTL0_ZCOND,DL_TIMER_CAC_CCCTL0_ACOND,DL_TIMER_
CLC_CCCTL0_LCOND);

    DL_TimerA_setCaptureCompareOutCtl(PWM_0_INST, DL_TIMER_CC_OCTL_INIT_VAL_LOW,
        DL_TIMER_CC_OCTL_INV_OUT_DISABLED, DL_TIMER_CC_OCTL_SRC_FUNCVAL,
        DL_TIMERA_CAPTURE_COMPARE_0_INDEX);

    DL_TimerA_setCaptCompUpdateMethod(PWM_0_INST, DL_TIMER_CC_UPDATE_METHOD_IMMEDIATE,
DL_TIMERA_CAPTURE_COMPARE_0_INDEX);
    DL_TimerA_setCaptureCompareValue(PWM_0_INST, 500, DL_TIMER_CC_0_INDEX);

    DL_TimerA_setCaptureCompareOutCtl(PWM_0_INST, DL_TIMER_CC_OCTL_INIT_VAL_LOW,
        DL_TIMER_CC_OCTL_INV_OUT_DISABLED, DL_TIMER_CC_OCTL_SRC_FUNCVAL,
        DL_TIMERA_CAPTURE_COMPARE_1_INDEX);

    DL_TimerA_setCaptCompUpdateMethod(PWM_0_INST, DL_TIMER_CC_UPDATE_METHOD_IMMEDIATE,
DL_TIMERA_CAPTURE_COMPARE_1_INDEX);
    DL_TimerA_setCaptureCompareValue(PWM_0_INST, 500, DL_TIMER_CC_1_INDEX);
```

```
    DL_TimerA_enableClock(PWM_0_INST);

    DL_TimerA_setCCPDirection(PWM_0_INST , DL_TIMER_CC0_OUTPUT | DL_TIMER_CC1_OUTPUT );

    DL_TimerA_configCrossTrigger(PWM_0_INST, DL_TIMER_CROSS_TRIG_SRC_ZERO,
                                 DL_TIMER_CROSS_TRIGGER_INPUT_ENABLED,
DL_TIMER_CROSS_TRIGGER_MODE_ENABLED
        );//Configuration to Generate Hardware Based Cross Trigger on Zero Event

    DL_TimerA_setCaptureCompareInput(PWM_0_INST, DL_TIMER_CC_INPUT_INV_NOINVERT,
DL_TIMER_CC_IN_SEL_TRIG, DL_TIMER_CC_0_INDEX);

    DL_TimerA_setCaptureCompareInput(PWM_0_INST, DL_TIMER_CC_INPUT_INV_NOINVERT,
DL_TIMER_CC_IN_SEL_TRIG, DL_TIMER_CC_1_INDEX);

    /*
     * Determines the external triggering event to trigger the module (self-triggered in main
configuration)
     * and triggered by specific timer in secondary configuration
     */
    DL_TimerA_setExternalTriggerEvent(PWM_0_INST,DL_TIMER_EXT_TRIG_SEL_TRIG_1);
    DL_TimerA_enableExternalTrigger(PWM_0_INST);
    uint32_t temp;
    temp = DL_TimerA_getCaptureCompareCtl(PWM_0_INST, DL_TIMER_CC_0_INDEX);
    DL_TimerA_setCaptureCompareCtl(PWM_0_INST, DL_TIMER_CC_MODE_COMPARE, temp | (uint32_t)
DL_TIMER_CC_ZCOND_TRIG_RISE, DL_TIMER_CC_0_INDEX);

    temp = DL_TimerA_getCaptureCompareCtl(PWM_0_INST, DL_TIMER_CC_1_INDEX);
    DL_TimerA_setCaptureCompareCtl(PWM_0_INST, DL_TIMER_CC_MODE_COMPARE, temp | (uint32_t)
DL_TIMER_CC_ZCOND_TRIG_RISE, DL_TIMER_CC_1_INDEX);

}
```

3.1.2 Configuration for the Secondary Timer

The secondary timer receives the cross-trigger signal and synchronizes its operation with the primary timer.

- Configure the TIMx.TSEL.ETSEL field as per the Cross-Trigger mapping (see device-specific data sheet) and the primary and secondary timer instances being used.
- Enable the input trigger function by setting the TIMA.TSEL.TE bit to 1.
- Set TIMAx.IFCTL_01.ISEL bit to 3 to select the cross-trigger as the input source.
- If counter is counting in the down counting mode set CCCTL.LCOND as 1 to use rising edge to trigger a load event this will set the counter value as load value when the cross-trigger is received and the counter will start down counting, similarly for up counting mode set CCCTL.ZCOND as 1
- In case of a self-cross trigger scenario and up-down counting mode, if, for example, TIMx.CTTRIGCTL.EVTCTTRIGSEL is set as zero, event then CCCTL.ZCOND should be set as 1
- The TIMx.CTRCTL.EN bit is set as the result of an LCOND or ZCOND condition being met, and the counter value changes to the load value or zero value, respectively.

Note

Timers which are working in the cross trigger mode should have the same TIMCLK frequency. In case the generated cross trigger pulse-width is less than one TIMCLK cycle of the Timer receiving the trigger, the cross trigger will not enable the counter.

```
/* Configuration for Secondary Timer to Receive a Cross Trigger */
static const DL_TimerA_ClockConfig gPWM_1ClockConfig = {
    .clockSel = DL_TIMER_CLOCK_BUSCLK,
    .divideRatio = DL_TIMER_CLOCK_DIVIDE_1,
    .prescale = 0U
};

static const DL_TimerA_PWMConfig gPWM_1Config = {
    .pwmMode = DL_TIMER_PWM_MODE_CENTER_ALIGN,
    .period = 1600,
    .isTimerWithFourCC = true,
    .startTimer = DL_TIMER_STOP,
};
```

```
SYSCONFIG_WEAK void SYSCFG_DL_PWM_1_init(void) {

    DL_TimerA_setClockConfig(
        PWM_1_INST, (DL_TimerA_ClockConfig *) &gPWM_1ClockConfig);

    DL_TimerA_initPWMMode(
        PWM_1_INST, (DL_TimerA_PWMConfig *) &gPWM_1Config);

    // Set Counter control to the smallest CC index being used

DL_TimerA_setCounterControl(PWM_1_INST,DL_TIMER_CZC_CCCTL0_ZCOND,DL_TIMER_CAC_CCCTL0_ACOND,DL_TIMER_
CLC_CCCTL0_LCOND);

    DL_TimerA_setCaptureCompareOutCtl(PWM_1_INST, DL_TIMER_CC_OCTL_INIT_VAL_LOW,
        DL_TIMER_CC_OCTL_INV_OUT_DISABLED, DL_TIMER_CC_OCTL_SRC_FUNCVAL,
        DL_TIMERA_CAPTURE_COMPARE_0_INDEX);

    DL_TimerA_setCaptCompUpdateMethod(PWM_1_INST, DL_TIMER_CC_UPDATE_METHOD_IMMEDIATE,
DL_TIMERA_CAPTURE_COMPARE_0_INDEX);
    DL_TimerA_setCaptureCompareValue(PWM_1_INST, 500, DL_TIMER_CC_0_INDEX);

    DL_TimerA_setCaptureCompareOutCtl(PWM_1_INST, DL_TIMER_CC_OCTL_INIT_VAL_LOW,
        DL_TIMER_CC_OCTL_INV_OUT_DISABLED, DL_TIMER_CC_OCTL_SRC_FUNCVAL,
        DL_TIMERA_CAPTURE_COMPARE_1_INDEX);

    DL_TimerA_setCaptCompUpdateMethod(PWM_1_INST, DL_TIMER_CC_UPDATE_METHOD_IMMEDIATE,
DL_TIMERA_CAPTURE_COMPARE_1_INDEX);
    DL_TimerA_setCaptureCompareValue(PWM_1_INST, 500, DL_TIMER_CC_1_INDEX);

    DL_Timer_enablePhaseLoad(PWM_1_INST);

    DL_TimerA_enableClock(PWM_1_INST);

    DL_Timer_setPhaseLoadValue(PWM_1_INST, 200);

    DL_TimerA_setCCPDirection(PWM_1_INST , DL_TIMER_CC0_OUTPUT | DL_TIMER_CC1_OUTPUT );

    DL_TimerA_setCaptureCompareInput(PWM_1_INST, DL_TIMER_CC_INPUT_INV_NOINVERT,
DL_TIMER_CC_IN_SEL_TRIG, DL_TIMER_CC_0_INDEX);

    DL_TimerA_setCaptureCompareInput(PWM_1_INST, DL_TIMER_CC_INPUT_INV_NOINVERT,
DL_TIMER_CC_IN_SEL_TRIG, DL_TIMER_CC_1_INDEX);

    /*
     * Determines the external triggering event to trigger the module (self-triggered in main
configuration)
     * and triggered by specific timer in secondary configuration
     */
    DL_TimerA_setExternalTriggerEvent(PWM_1_INST,DL_TIMER_EXT_TRIG_SEL_TRIG_1);
    DL_TimerA_enableExternalTrigger(PWM_1_INST);
    uint32_t temp;
    temp = DL_TimerA_getCaptureCompareCtl(PWM_1_INST, DL_TIMER_CC_0_INDEX);
    DL_TimerA_setCaptureCompareCtl(PWM_1_INST, DL_TIMER_CC_MODE_COMPARE, temp | (uint32_t)
DL_TIMER_CC_ZCOND_TRIG_RISE, DL_TIMER_CC_0_INDEX);

    temp = DL_TimerA_getCaptureCompareCtl(PWM_1_INST, DL_TIMER_CC_1_INDEX);
    DL_TimerA_setCaptureCompareCtl(PWM_1_INST, DL_TIMER_CC_MODE_COMPARE, temp | (uint32_t)
DL_TIMER_CC_ZCOND_TRIG_RISE, DL_TIMER_CC_1_INDEX);
}
```

3.1.3 Implementation for Cross Trigger Function

Table 3-1 introduces two cross trigger generation methods.

Table 3-1. Cross Trigger

S.No.	Software Cross Trigger	Hardware Cross Trigger
1	Software cross triggers are essentially generated when the application code writes 1 to the TIMx.CTTRIG.TRIG bit.	Based on the event configured in the TIMx.CTTRIGCTL.EVTCTTRIGSEL field, hardware cross triggers are generated by the hardware at every configured event without any involvement from the software(after the initial configuration has been done), for example- if the EVTCTTRIGSEL has been set as Zero Event, hardware cross trigger would be generated at every zero event of the primary timer.
2	The software cross trigger would synchronize the primary and the secondary timer only when the application requires to do so. **Note** The synchronization due to software might lose effect in case of changes in Load/CC values.	The hardware cross triggering mechanism synchronizes the primary and secondary timers repeatedly. Thus even if there's a change in Load/CC value of the primary timer, the secondary timers would get synchronized with the primary timer as the hardware cross trigger would be generated at every event.

Mechanism to Generate a Software-based Cross-Trigger Signal

```
DL_TimerA_generateCrossTrigger(PWM_0_INST);//Mechanism to generate Software based Cross-Trigger,
this software cross trigger will enable TIMA0 and TIMA1
```

Configuration to Generate Hardware based Cross-Trigger on Zero Event

```
    DL_TimerA_configCrossTrigger(PWM_0_INST, DL_TIMER_CROSS_TRIG_SRC_ZERO,
    DL_TIMER_CROSS_TRIGGER_INPUT_ENABLED, DL_TIMER_CROSS_TRIGGER_MODE_ENABLED
        );  // Configuration to Generate Hardware based Cross-Trigger on Zero Event
```

Phase Shifted PWM Generation

Figure 3-1 shows the test waveform for phase shifted PWM generation.

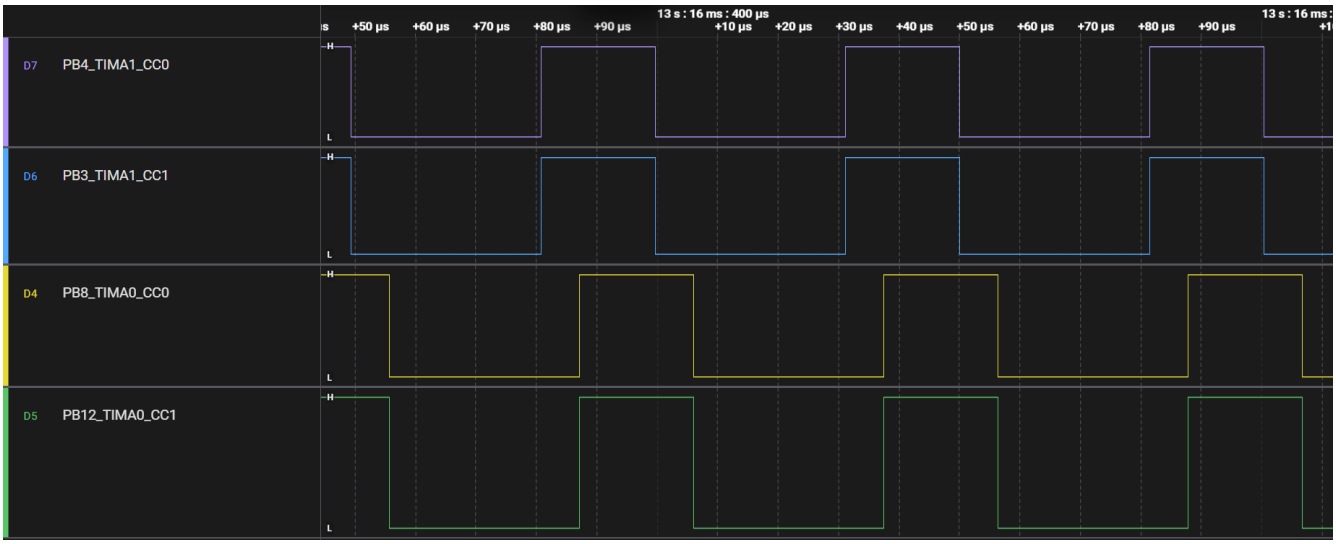

Figure 3-1. Phase Shifted PWM Generation Using Cross-Trigger and Phase Load

3.2 Using Secondary Capture-Compare Channels

TIMA0 with its 4 external and 2 internal CC channels can be independently used to generate 3 phase shifted PWMs. This can be achieved by utilizing the internal CC channels' SECONDARY CC event. GPTIMER provides the feature of choosing the secondary CC channel by configuring the CCCTL.CC2SELD or CCCTL.CC2SELU, depending on the counting mode(down, up, up-down).

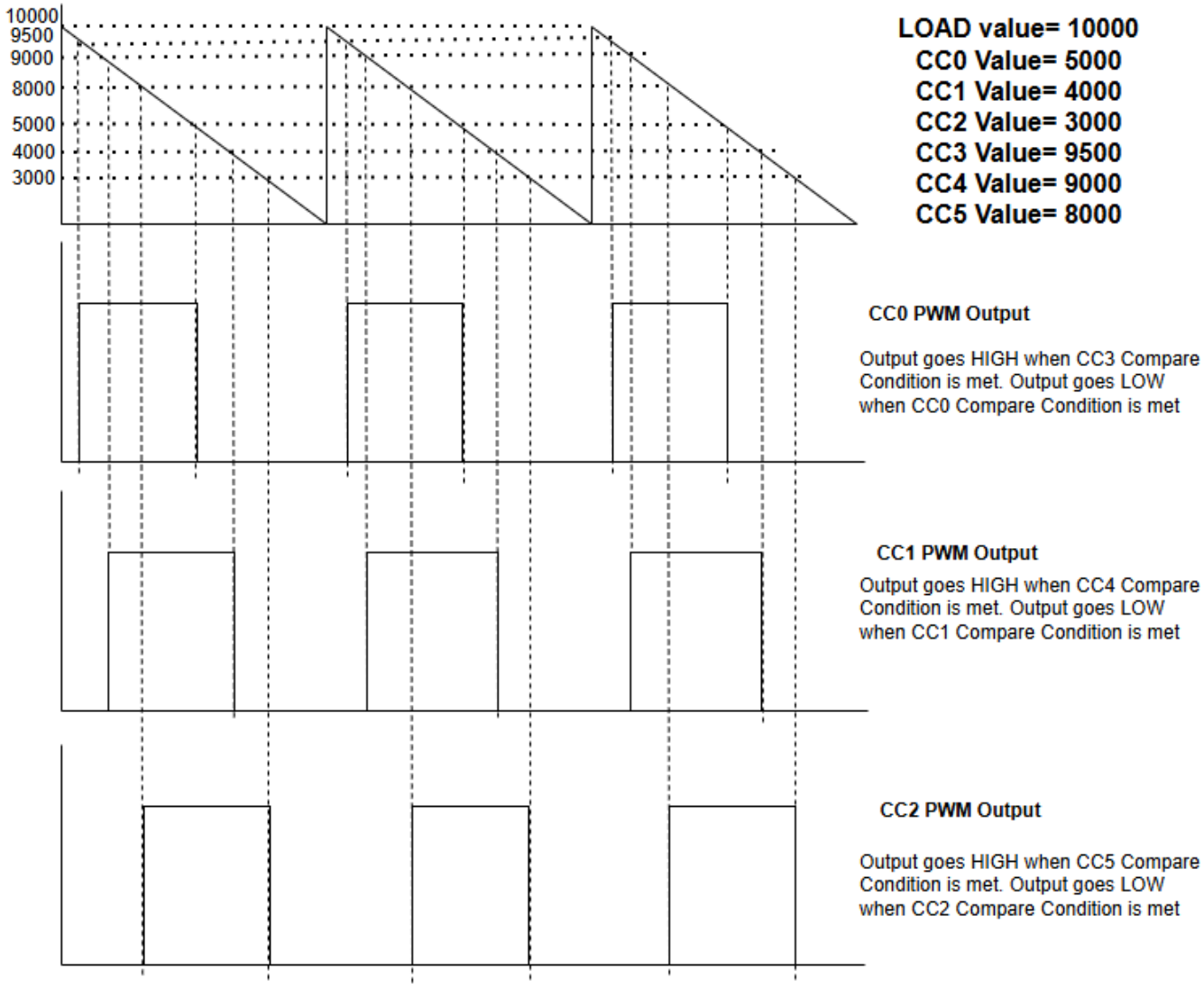

Figure 3-2. Diagrammatic Representation of Asymmetric PWM generation using TIMA0 with Secondary Channel Approach

Based on the secondary channel selected, output generation can be done by configuring the CCACT.CC2UACT or CCACT.CC2DACT depending on the counting mode(down, up, up-down).

For example, if counter Load value is 10000 and counter is in down counting mode and CC4 value is 9000.

CCCTL[1].CC2SELD is configured as 0x4 meaning CC4 is chosen as the secondary CC channel for the main channel CC1.

CCACT[1].CC2DACT is configured as 0x1 meaning CC output value would be set as HIGH whenever counter reaches 9000 in value, which is the current CC4 value. CC4 will act as the **pseudo** load value for CC1.

It would be observed that the output of CC1 would go HIGH every time the counter reaches 9000 because of the CCACT[1].CC2DACT. PWM output will go LOW when the counter reaches 4000 (CC1 value).

Implementing this approach to generate three phase shifted PWMs can be useful. Using cross-trigger function to generate more sync shifted PWM output channels with multiple Timer instances.

Following the **configuration sequence** below to set the secondary CC channel:

- In the TIMA.CTRCTL, set the desired counter control settings for:
  - Counting Mode(CM) and Count Value After Enable(CVAE).
  - CLC, CZC, CAC to specify what condition controls zeroing, advancing or loading the counter
- Set the TIMA.LOAD value to configure the PWM period
- Set TIMA.CCCTL_xy[0/1].COC=0 for Compare Mode
- Configure CCP as an output for the CC block by setting respective bit in the CCPD registers.
- In TIMA.OCTL_xy[0/1], set CCPO = 0 to select the signal generator output.
- Enable the corresponding CCP output by setting ODIS.C0CCPn to 0 for the corresponding counter n.
- Configure polarity of the signal using the CCPOINV bit and configure CCPIV to specify the CCP output state while counter is disabled.
- Configure CCCTL.CC2SELD field for CC0, as 3 meaning CC3 is used as the secondary CC block.
- Configure CCCTL.CC2SELD field for CC1, as 4 meaning CC4 is used as the secondary CC block.
- Configure CCCTL.CC2SELD field for CC2, as 5 meaning CC5 is used as the secondary CC block.
- Configure CC3 depending on the phase shift required for example, if the phase shift is needed for 500 TIMCLK cycles configure CC3= Load value – 500=9500.
- Configure CC4 depending on the phase shift required for example, if the phase shift is needed for 1000 TIMCLK cycles configure CC4= Load value – 1000=9000.
- Configure CC5 depending on the phase shift required for example, if the phase shift is needed for 2000 TIMCLK cycles configure CC5= Load value – 2000=8000.
- Configure CC0, CC1, CC2 based on the required duty cycle with CC3, CC4 and CC5 as the reference respectively.
- CC3, CC4 and CC5 values will act as the **pseudo** load values for CC0, CC1 and CC2 respectively.
- Configure CCACT.CC2DACT for CC0, CC1 and CC2 as 0x1 to set the output high whenever counter reaches the CC3, CC4 and CC5 values respectively.
- Configure CCACT.CDACT for CC0, CC1 and CC2 as 0x2 to clear the output low whenever counter reaches the CC0, CC1 and CC2 values respectively.
- Enable the counter by setting TIMA.CTRCTL.EN = 1.

```
/* TIMA0 Configuration for Generating Phase Shifted PWMs Using Secondary CC events */
static const DL_TimerA_ClockConfig gPWM_0ClockConfig = {
    .clockSel = DL_TIMER_CLOCK_BUSCLK,
    .divideRatio = DL_TIMER_CLOCK_DIVIDE_1,
    .prescale = 255U
};

static const DL_TimerA_PWMConfig gPWM_0Config = {
    .pwmMode = DL_TIMER_PWM_MODE_EDGE_ALIGN,
    .period = 10000,
    .isTimerWithFourCC = true,
    .startTimer = DL_TIMER_STOP,
};

SYSCONFIG_WEAK void SYSCFG_DL_PWM_0_init(void) {

    DL_TimerA_setClockConfig(
        PWM_0_INST, (DL_TimerA_ClockConfig *) &gPWM_0ClockConfig);

    DL_TimerA_initPWMMode(
        PWM_0_INST, (DL_TimerA_PWMConfig *) &gPWM_0Config);

    // Set Counter control to the smallest CC index being used

DL_TimerA_setCounterControl(PWM_0_INST,DL_TIMER_CZC_CCCTL0_ZCOND,DL_TIMER_CAC_CCCTL0_ACOND,DL_TIMER_
CLC_CCCTL0_LCOND);

    DL_TimerA_setCaptureCompareOutCtl(PWM_0_INST, DL_TIMER_CC_OCTL_INIT_VAL_LOW,
        DL_TIMER_CC_OCTL_INV_OUT_DISABLED, DL_TIMER_CC_OCTL_SRC_FUNCVAL,
        DL_TIMERA_CAPTURE_COMPARE_0_INDEX);

    DL_TimerA_setCaptCompUpdateMethod(PWM_0_INST, DL_TIMER_CC_UPDATE_METHOD_IMMEDIATE,
DL_TIMERA_CAPTURE_COMPARE_0_INDEX);
```

```
    DL_TimerA_setCaptureCompareOutCtl(PWM_0_INST, DL_TIMER_CC_OCTL_INIT_VAL_LOW,
        DL_TIMER_CC_OCTL_INV_OUT_DISABLED, DL_TIMER_CC_OCTL_SRC_FUNCVAL,
        DL_TIMERA_CAPTURE_COMPARE_1_INDEX);

    DL_TimerA_setCaptCompUpdateMethod(PWM_0_INST, DL_TIMER_CC_UPDATE_METHOD_IMMEDIATE,
DL_TIMERA_CAPTURE_COMPARE_1_INDEX);
    PWM_0_INST->COUNTERREGS.CCCTL_01[0]=0x60000000;//Configuring CC3 as the secondary event for CC0
    PWM_0_INST->COUNTERREGS.CCCTL_01[1]=0x80000000;//Configuring CC4 as the secondary event for CC1
    PWM_0_INST->COUNTERREGS.CCCTL_23[0]=0xA0000000;//Configuring CC5 as the secondary event for CC2

    PWM_0_INST->COUNTERREGS.CCACT_01[0]=0x00001080;//Configuring CC2DACT to drive PWM High and
CDACT to drive PWM Low
    PWM_0_INST->COUNTERREGS.CCACT_01[1]=0x00001080;//Configuring CC2DACT to drive PWM High and
CDACT to drive PWM Low
    PWM_0_INST->COUNTERREGS.CCACT_23[0]=0x00001080;//Configuring CC2DACT to drive PWM High and
CDACT to drive PWM Low

    DL_TimerA_setCaptureCompareValue(PWM_0_INST, 5000, DL_TIMER_CC_0_INDEX);
    DL_TimerA_setCaptureCompareValue(PWM_0_INST, 4000, DL_TIMER_CC_1_INDEX);
    DL_TimerA_setCaptureCompareValue(PWM_0_INST, 3000, DL_TIMER_CC_2_INDEX);
    DL_TimerA_setCaptureCompareValue(PWM_0_INST, 9500, DL_TIMER_CC_3_INDEX);
    DL_TimerA_setCaptureCompareValue(PWM_0_INST, 9000, DL_TIMER_CC_4_INDEX);
    DL_TimerA_setCaptureCompareValue(PWM_0_INST, 8000, DL_TIMER_CC_5_INDEX);

    DL_TimerA_enableClock(PWM_0_INST);
    DL_TimerA_setCCPDirection(PWM_0_INST , DL_TIMER_CC0_OUTPUT | DL_TIMER_CC1_OUTPUT |
DL_TIMER_CC2_OUTPUT);
}
```

Figure 3-3 shows the shifted PWM output waveforms. To dynamically change the period/duty cycle, refer to Section 8 .

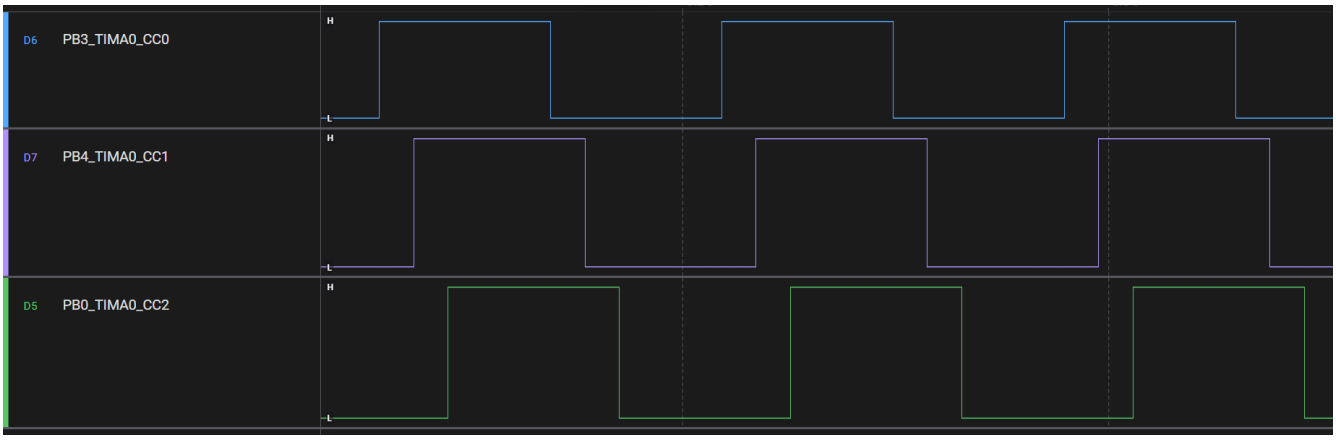

Figure 3-3. Phase Shifted PWM Generation Using TIMA0 Secondary Events

4 Bit-Banging Emulation: Software-based Communication Protocol Implementation

TIMA can be used to emulate a standard communication protocol like the Universal Asynchronous Receiver Transmitter (UART). TIMA also allows the implementation of non-standard or custom protocols, where high security encryption is required. With TIMA developers can precisely schedule output toggles for clock or data lines, capture input edges for read operations and compare events to generate consistent baud rates or clock cycles.

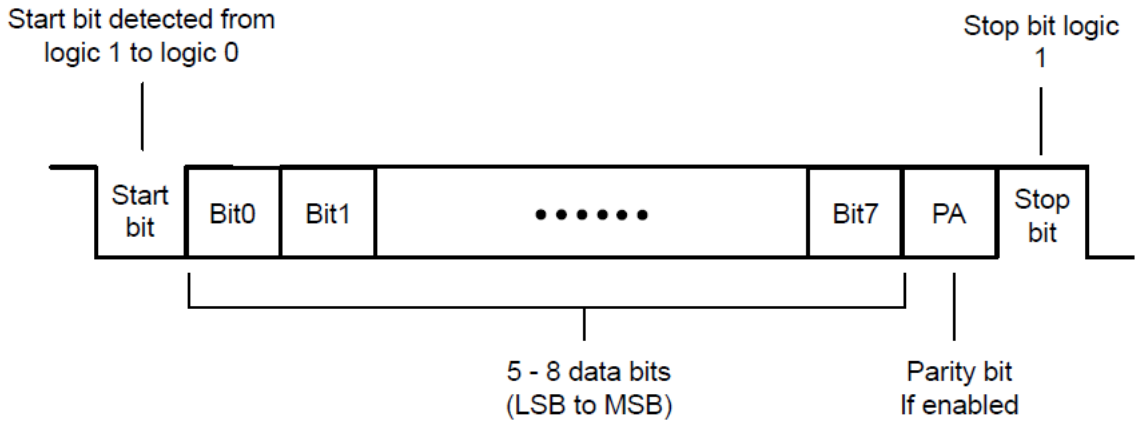

Figure 4-1. UART Protocol Standard Frame

4.1 Emulation of UART Rx Using TIMA

TIMA provides all the necessary features to configure it to emulate the UART protocol. Figure 4-2 shows the flow chart for using TIMA as UART Rx.

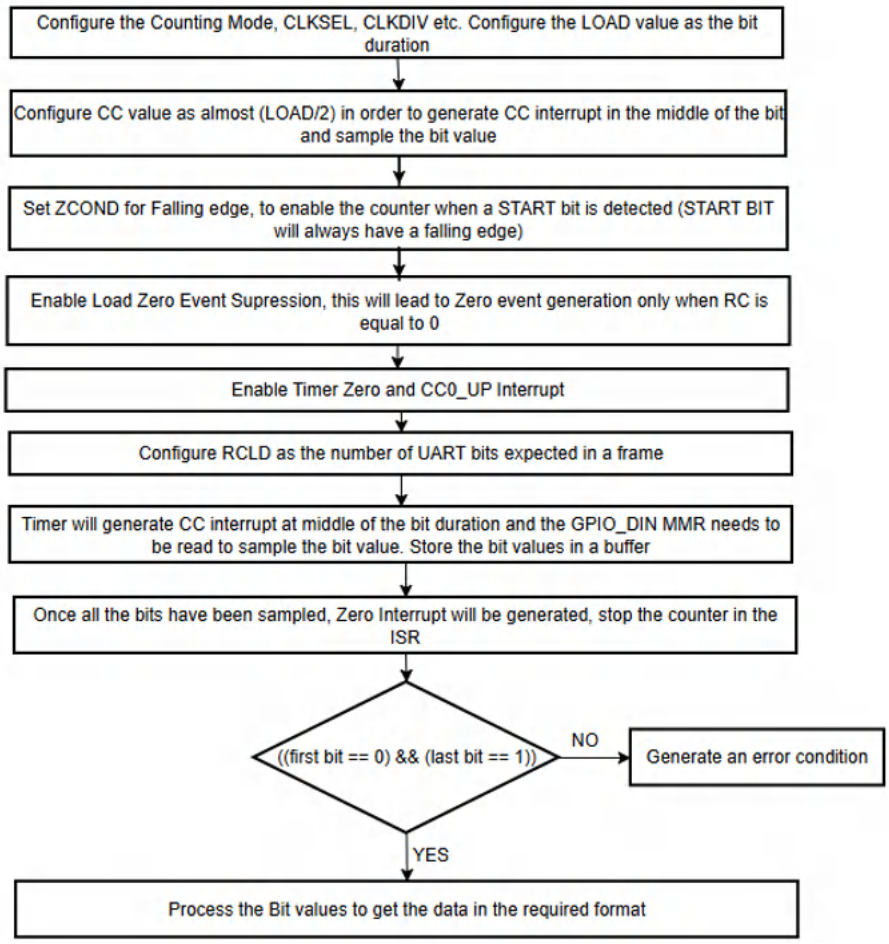

Figure 4-2. Flowchart Demonstrating UART Rx

The configuration sequence is described below:

- Configure TIMA in Up-Counting Mode and configure CCCTL.ZCOND for falling edge detection, this is because in UART, start of a frame is indicated with a falling edge.
- Configure the Load value of the TIMER as the bit time depending on the Baud Rate.
- Configure the timer to run in the repeat mode for n times (where n= number of bits expected in the UART frame) by configuring the RCLD register.Configure CC value as (LOAD value/2 or LOAD value/2 +1), to generate CC event in the middle of the bit and sample the corresponding bit.
- Enable Load and Zero event suppression, this will not generate zero events until RC is zero. Zero event is not required unless all bits have been sampled.
- In the Timer's Interrupt service routine, in case of CC event read the GPIO_DIN register, this will give the input value of signal
- In the Timer's Interrupt service routine, in case of Zero event stop the counter.

```
/* Timer Configuration for Emulating UART Rx */
//9600 UART baud rates equates to bit duration of 104.167us, hence configuring LOAD Value as 3333
with 32MHz Clock
static const DL_TimerA_ClockConfig gCAPTURE_0ClockConfig = {
    .clockSel    = DL_TIMER_CLOCK_BUSCLK,
    .divideRatio = DL_TIMER_CLOCK_DIVIDE_1,
    .prescale    = 0U
};

static const DL_TimerA_TimerConfig gTimer_Config = {
    .timerMode      = DL_TIMER_TIMER_MODE_PERIODIC_UP,
    .period         = 3332,
    .startTimer     = DL_TIMER_STOP,
    .genIntermInt   = DL_TIMER_INTERM_INT_ENABLED,
```

```
    .counterVal     = 1666,
};

SYSCONFIG_WEAK void SYSCFG_DL_CAPTURE_0_init(void) {

    DL_TimerA_setClockConfig(CAPTURE_0_INST,
        (DL_TimerA_ClockConfig *) &gCAPTURE_0ClockConfig);
    DL_TimerA_initTimerMode(CAPTURE_0_INST,
        (DL_TimerA_TimerConfig *) &gTimer_Config);

DL_TimerA_setCounterControl(CAPTURE_0_INST,DL_TIMER_CZC_CCCTL0_ZCOND,DL_TIMER_CAC_CCCTL0_ACOND,DL_TI
MER_CLC_CCCTL0_LCOND);
    DL_TimerA_setCaptCompUpdateMethod(CAPTURE_0_INST, DL_TIMER_CC_UPDATE_METHOD_IMMEDIATE,
DL_TIMERA_CAPTURE_COMPARE_0_INDEX);

    DL_TimerA_setCaptureCompareValue(CAPTURE_0_INST, 1666, DL_TIMER_CC_0_INDEX);//Configuring CC1
as almost half the bit width, to sample the bit in the middle
    uint32_t temp;
    temp = DL_TimerA_getCaptureCompareCtl(CAPTURE_0_INST, DL_TIMER_CC_0_INDEX);
    DL_TimerA_setCaptureCompareCtl(CAPTURE_0_INST, DL_TIMER_CC_MODE_COMPARE, temp | (uint32_t)
DL_TIMER_CC_ZCOND_TRIG_FALL, DL_TIMER_CC_0_INDEX);//Enable Counter on a falling edge, this will
detect the start bit

    DL_TimerA_enableInterrupt(CAPTURE_0_INST , DL_TIMERA_INTERRUPT_CC0_UP_EVENT |
        DL_TIMERA_INTERRUPT_ZERO_EVENT);
    DL_Timer_enableLZEventSuppression(CAPTURE_0_INST);//This will suppress zero events until RC is
equal to 0
    DL_TimerA_enableClock(CAPTURE_0_INST);
}
```

```
/* Application Code for Emulating UART Rx Using Timer */
#define GPIO_CAPTURE_0_C0_PIN_BIT 3
volatile uint32_t gCaptureCnt;
volatile bool gSynced;
volatile bool gCheckCaptures;
uint32_t gLoadValue;
uint32_t uart_data_frame_received[10];
uint32_t number_of_zero_interrupts=0;
uint32_t number_of_CC_interrupts=0;
uint32_t data_bit_value;
uint8_t number_of_bits_in_UART_frame=0;
bool uart_frame_received=0;
uint8_t hex_value=0;
uint8_t uart_data_received=0;
bool uart_frame_erroneous=0;
int main(void)
{
    SYSCFG_DL_init();
    NVIC_EnableIRQ(CAPTURE_0_INST_INT_IRQN);
    number_of_bits_in_UART_frame=sizeof(uart_data_frame_received)/
sizeof(uart_data_frame_received[0]);
    DL_Timer_setRepeatCounter(CAPTURE_0_INST, number_of_bits_in_UART_frame);//Set the repeat
counter= No. of bits to be received in the UART Frame

    while(1){
        while (uart_frame_received==0){};

        if(uart_data_frame_received[0]!=0){
            uart_frame_erroneous=true;//If 0th Bit i.e. the START bit is not 0, raise an ERROR Flag
        }
        if(uart_data_frame_received[9]!=1){
            uart_frame_erroneous=true;//If 9th Bit i.e. the STOP bit is not 1, raise an ERROR Flag
        }
        for(int i=1;i<number_of_bits_in_UART_frame-1;i++){
            hex_value|= uart_data_frame_received[i]<<(i-1);//Convert the values to HEX
        }
        uart_data_received=hex_value;
        uart_frame_received=0;
    }
}
void CAPTURE_0_INST_IRQHandler(void)
{
    switch (DL_TimerA_getPendingInterrupt(CAPTURE_0_INST)) {
        case DL_TIMERA_IIDX_CC0_UP:
            data_bit_value=(DL_GPIO_readPins(GPIO_CAPTURE_0_C0_PORT,
GPIO_CAPTURE_0_C0_PIN))>>GPIO_CAPTURE_0_C0_PIN_BIT;//Read the GPIO value, to sample the input state
and the Bit value
```

```
            uart_data_frame_received[number_of_CC_interrupts++]=data_bit_value;//Store the bit
value in an array
            break;
        case DL_TIMERG_IIDX_ZERO:
                uart_frame_received=1;
                DL_TimerA_stopCounter(CAPTURE_0_INST);//Zero interrupt will come once all the UART
bits have been sampled, stop the counter when Zero Interrupt comes
                number_of_CC_interrupts=0;
                number_of_zero_interrupts=0;
            break;
        default:
            break;
    }
}
```

GPTIMER also supports input filtering mechanism, this feature can be utilized to prevent unwanted glitches from being incorrectly sampled as START bits.

Note

1. Prioritize reading the GPIO value first in ISR, so that sampling happens the correct time
2. Take care of setting TIMER ISR as the highest priority interrupt, otherwise this might mess up samping time point
3. Multiple combinations of CLK, CLKDIV and PRESCALER can be used to achieve the baud rate. Lower Timer function clock might reduce capture resolution.

The application can alternatively be implemented using TIMG in place of TIMA. However, there is an important limitation to consider: TIMG lacks support for the repeat counter functionality. As a result, when implementing with TIMG, the counter must be manually reloaded via software within the ISR to achieve the same functionality.

Note

To optimize application throughput for higher baud rates while reducing CPU bandwidth consumption, the application can be implemented using DMA. In this configuration, DMA can be configured to read the GPIO_DIN MMRs when triggered by Timer events, eliminating the need for CPU intervention. Refer to device-specific data sheet to confirm DMA accessibility to the GPIO_DIN MMRs.

4.2 Emulation of UART Tx Using TIMA

TIMA provides a robust solution for UART transmitter emulation by leveraging its output generation capabilities, shadow CCACT functionality, and repeat counter mechanism. The performance of this implementation can be significantly enhanced by incorporating DMA transfers, creating an end-to-end solution that optimizes system resources. With DMA handling the data movement, CPU intervention is minimized while ensuring precise bit timing and reliable data transmission.

This integrated Timer-DMA approach eliminates dependencies on interrupt priorities and guarantees consistent timing accuracy. The solution ensures deterministic behavior, making it ideal for bit-bang emuylation applications. Figure 4-3 shows the flow chart for using TIMA as UART Tx.

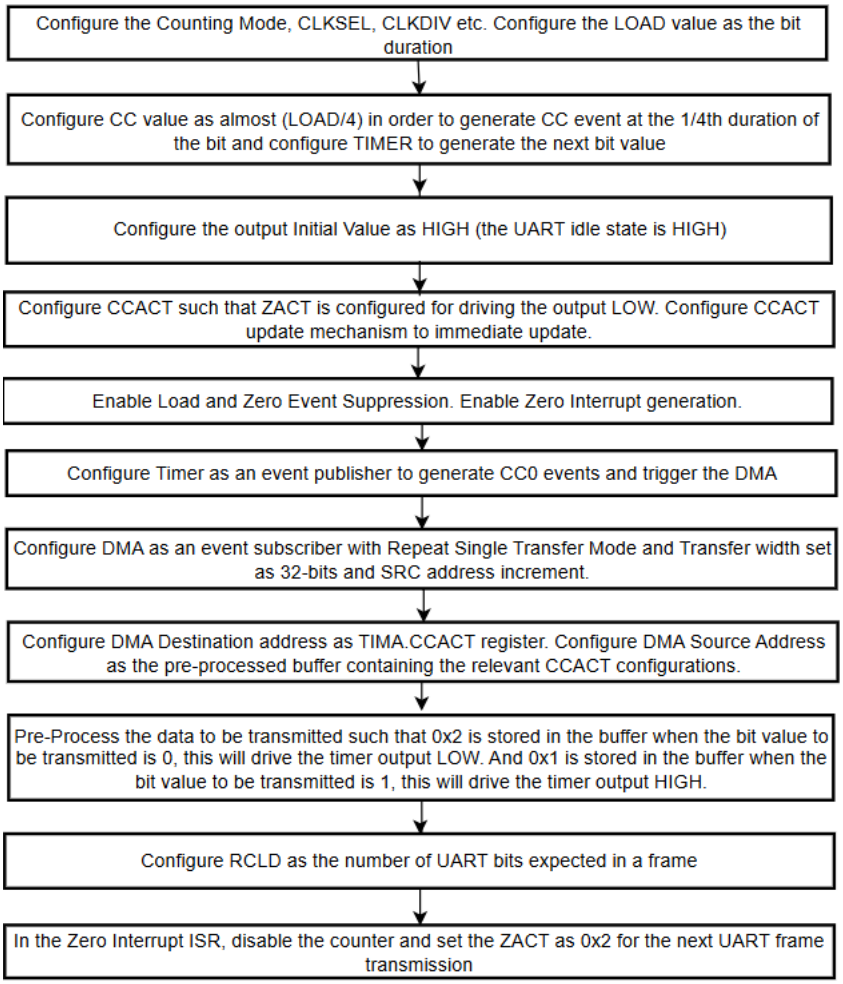

Figure 4-3. Flowchart Demonstrating UART Tx

The configuration sequence is described in detail below:

- Configure the Counting Mode, CLKSEL, CLKDIV etc. Configure the load value as per the bit duration.
- Configure the CC value as 'LOAD/4' or 'LOAD/4+1', this will generate CC events at the one-fourth of the total bit duration, this event is needed so that timer can be configured to transmit the next bit value.
- Since the UART Idle state is HIGH, configure the output initial value OCTL.CCPIV as HIGH.
- Configure CCACT such that ZACT is configured for driving the output LOW, this is because a LOW START bit should be generated when counter is enabled.
- Configure CCACT update mechanism to immediate update.
- Enable Load and Zero Event Suppression, this will help us generate zero events only when RC=0.
- Enable the Timer Zero Interrupt
- TIMA can generate events and trigger other peripherals like the DMA via the event fabric. TIMA needs to be configured as an event publisher so that DMA can be triggered to initiate data transfers when CC0 event is generated.
- DMA needs to be configured as an Event Subscriber, so that the DMA can carry out the data transfers as per TIMA events.
- Configure the DMA in Repeat Single Transfer mode, with 32-bit width and source address increment
- Configure DMA Destination Address as the TIMA.CCACT Register. At every CC0 event, CCACT register needs to be updated to transmit the next bit value. This updating mechanism will be carried out by DMA.
- Pre-process the data to be transmitted such that 0x2 is stored in the buffer when the bit value to be transmitted is 0, this will drive the timer output LOW. And 0x1 is stored in the buffer when the bit value to be transmitted is 1, this will drive the timer output HIGH.

- Configure the DMA Source Address as the pre-processed buffer containing the relevant CCACT configurations
- Configure RCLD as the number of UART bits expected in a frame
- Enable the Counter when data is to be transmitted.
- In the Zero Interrupt ISR, disable the counter and set the ZACT as 0x2 for the next UART frame transmission.

```
/* Timer Configuration for Emulating UART Tx with DMA */
//For 9600 baud rate, single bit duration is 104.167us, hence configuring LOAD value as 3333, with
32MHz clock
static const DL_TimerA_ClockConfig gPWM_0ClockConfig = {
    .clockSel = DL_TIMER_CLOCK_BUSCLK,
    .divideRatio = DL_TIMER_CLOCK_DIVIDE_1,
    .prescale = 0U
};
static const DL_TimerA_PWMConfig gPWM_0Config = {
    .pwmMode = DL_TIMER_PWM_MODE_EDGE_ALIGN_UP,
    .period = 3333,
    .isTimerWithFourCC = false,
    .startTimer = DL_TIMER_STOP,
};

SYSCONFIG_WEAK void SYSCFG_DL_PWM_0_init(void) {
    DL_TimerA_setClockConfig(
        PWM_0_INST, (DL_TimerA_ClockConfig *) &gPWM_0ClockConfig);
    DL_TimerA_initPWMMode(
        PWM_0_INST, (DL_TimerA_PWMConfig *) &gPWM_0Config);

DL_TimerA_setCounterControl(PWM_0_INST,DL_TIMER_CZC_CCCTL0_ZCOND,DL_TIMER_CAC_CCCTL0_ACOND,DL_TIMER_
CLC_CCCTL0_LCOND);
    DL_TimerA_setCaptureCompareOutCtl(PWM_0_INST, DL_TIMER_CC_OCTL_INIT_VAL_HIGH, //Keeping CC0
initial value as HIGH as UART idle state should be HIGH
        DL_TIMER_CC_OCTL_INV_OUT_DISABLED, DL_TIMER_CC_OCTL_SRC_FUNCVAL,
        DL_TIMERA_CAPTURE_COMPARE_0_INDEX);
    PWM_0_INST->COUNTERREGS.CCACT_01[0]=0x2;//Configuring Zero action as CC Output LOW, as START
bit in UART is always LOW
    DL_TimerA_setCaptCompUpdateMethod(PWM_0_INST, DL_TIMER_CC_UPDATE_METHOD_IMMEDIATE,
DL_TIMERA_CAPTURE_COMPARE_0_INDEX);
    DL_Timer_setCaptCompActUpdateMethod(PWM_0_INST,DL_TIMER_CCACT_UPDATE_METHOD_IMMEDIATE,
DL_TIMERA_CAPTURE_COMPARE_0_INDEX);
    DL_TimerA_setCaptureCompareValue(PWM_0_INST, 834, DL_TIMER_CC_0_INDEX);//Configuring CC0 as
almost 1/4th the bit width
    PWM_0_INST->COUNTERREGS.CCACT_01[0]=0x2;//Configuring Zero action as CC Output LOW, as START
bit in UART is always LOW
    DL_Timer_enableInterrupt(PWM_0_INST, DL_TIMER_INTERRUPT_ZERO_EVENT);
    DL_Timer_enableLZEventSuppression(PWM_0_INST);//This will suppress zero events until RC is
equal to 0

    DL_Timer_enableEvent(PWM_0_INST, DL_TIMER_EVENT_ROUTE_1, DL_TIMER_INTERRUPT_CC0_UP_EVENT);//
CC0_UP Event triggers the DMA CHAN0
    DL_Timer_setPublisherChanID(PWM_0_INST, DL_TIMER_PUBLISHER_INDEX_0, 1);//Timer publishing event
to DMA to carry out CCACT update
    DL_TimerA_enableClock(PWM_0_INST);
    DL_TimerA_setCCPDirection(PWM_0_INST , DL_TIMER_CC0_OUTPUT );
}
// Below is DMA configuration
static const DL_DMA_Config gDMA_CH0Config = {
    .transferMode   = DL_DMA_FULL_CH_REPEAT_SINGLE_TRANSFER_MODE,
    .extendedMode   = DL_DMA_NORMAL_MODE,
    .destIncrement  = DL_DMA_ADDR_UNCHANGED,
    .srcIncrement   = DL_DMA_ADDR_INCREMENT,
    .destWidth      = DL_DMA_WIDTH_WORD,
    .srcWidth       = DL_DMA_WIDTH_WORD,
    .trigger        = DMA_GENERIC_SUB0_TRIG,
    .triggerType    = DL_DMA_TRIGGER_TYPE_EXTERNAL,
};
void SYSCFG_DL_DMA_CH0_init(void)
{
    DL_DMA_initChannel(DMA, DMA_CH0_CHAN_ID , (DL_DMA_Config *) &gDMA_CH0Config);
    DL_DMA_clearInterruptStatus(DMA, DL_DMA_INTERRUPT_CHANNEL0);
        DL_DMA_enableInterrupt(DMA, DL_DMA_INTERRUPT_CHANNEL0);
        DL_DMA_setSubscriberChanID(DMA, DL_DMA_SUBSCRIBER_INDEX_0, 0x01);//DMA subscribing to Timer
event
}

void SYSCFG_DL_DMA_init(void){
```

```
    SYSCFG_DL_DMA_CH0_init();
}
```

```
/* Application Code for Emulating UART Tx Using Timer */
uint8_t data_to_transmit[8]={1,1,0,1,0,1,0,0};//this equals to 0x2B
uint8_t uart_frame_to_transmit[9];//8 data bits + STOP Bit
uint8_t number_of_bits_in_each_frame=0;
uint8_t number_of_bits_in_uart_frame=0;
uint32_t DMA_frame_to_transmit[9];
int number_of_CC_interrupts=0;
int number_of_zero_int=0;
//int number_of_load_int=0;
volatile bool gIsTimerExpired;
int main(void)
{
    bool isRetentionError;

    NVIC_EnableIRQ(PWM_0_INST_INT_IRQN);

    SYSCFG_DL_init();
    DL_DMA_setDestAddr(DMA, DMA_CH0_CHAN_ID, (uint32_t)&PWM_0_INST->COUNTERREGS.CCACT_01[0]);//Set
DMA destination address as TIM_CCACT MMR
    DL_DMA_setSrcAddr(DMA, DMA_CH0_CHAN_ID, (uint32_t)&DMA_frame_to_transmit );//Set DMA source
address as the array which contains CCACT config for each bit
    DL_DMA_setTransferSize( DMA, DMA_CH0_CHAN_ID,number_of_bits_in_uart_frame);//Configure DMA SZ
for the number of bits in each frame

    DL_DMA_enableChannel(DMA, DMA_CH0_CHAN_ID);//Enable DMA channel

    number_of_bits_in_each_frame=sizeof(data_to_transmit);
    uart_frame_to_transmit[number_of_bits_in_each_frame+1]=1;//STOP bit is always 1

    for(int i=0;i<number_of_bits_in_each_frame;i++){
        uart_frame_to_transmit[i]=data_to_transmit[i];
        }

    number_of_bits_in_uart_frame=sizeof(uart_frame_to_transmit); //Adding 1 Stop bit Frame

    DL_Timer_setRepeatCounter(PWM_0_INST,number_of_bits_in_uart_frame);//Configuring RCLD as the
number_of_bits_in_uart_frame
/////////PRE-PROCESS THE ARRAY WHICH CONTAINS THE CORRESPONDING CCACT CONFIGURATION FOR EACH
BIT//////////////////////////

for( number_of_CC_interrupts=0;number_of_CC_interrupts<(number_of_bits_in_uart_frame+1);number_of_CC
_interrupts++){
    if(uart_frame_to_transmit[number_of_CC_interrupts]==0){
        DMA_frame_to_transmit[number_of_CC_interrupts]=0x2;//if bit value is 0 configure ZACT as CC
Output LOW
    }
    else if(uart_frame_to_transmit[number_of_CC_interrupts]==1){
        DMA_frame_to_transmit[number_of_CC_interrupts]=0x1;//if bit value is 1 configure ZACT as CC
Output HIGH
    }
    else{
    }
    if(number_of_CC_interrupts==number_of_bits_in_uart_frame-1){
        DMA_frame_to_transmit[number_of_CC_interrupts]=0x1;//Always generate a HIGH stop bit
    }
}
////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////
    DL_TimerA_startCounter(PWM_0_INST);//Start Counter
    while(1);
}

void PWM_0_INST_IRQHandler(void)
{
    switch (DL_TimerA_getPendingInterrupt(PWM_0_INST)) {
        case DL_TIMER_IIDX_ZERO:
            number_of_zero_int++;
            DL_TimerA_stopCounter(PWM_0_INST);//Stop the counter
            PWM_0_INST->COUNTERREGS.CCACT_01[0]=0x2;//Configure ZACT for the next UART Transmission
            delay_cycles(3333);//Adding 1 bit of delay between 2 UART frames
            number_of_CC_interrupts=0;
            DL_TimerA_startCounter(PWM_0_INST);//Start counter for the next UART Transmission
            break;
        default:
```

```
            break;
    }
}
```

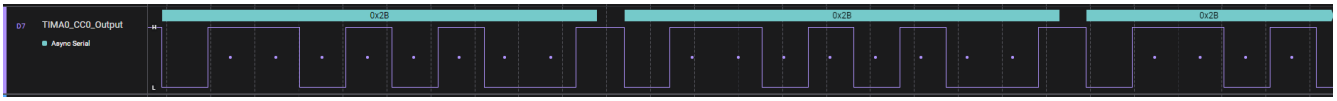

Figure 4-4. Transmitting 0x2B in UART Format Using TIMA

Note

For the Application described above, an alternative implementation using TIMG instead of TIMA is possible. However, TIMG does not support the repeat counter feature. Therefore, when using TIMG, the counter would need to be manually reloaded through software intervention within the ISR.

5 Feedback-Based PWM Generation

Feedback-based Pulse Width Modulation (PWM) signal generation is used to achieve precise control and regulation in various applications by dynamically adjusting the pulse width based on system feedback. This is particularly useful for applications requiring continuous monitoring and adaptation to changing conditions, such as motor speed control, power supply regulation, and battery management systems.

5.1 Feedback-Based PWM Signal Replication

The ability to generate an exact copy of an input PWM signal serves critical functions in modern electronic systems, particularly in applications requiring safety redundancy validation. This methodology finds extensive implementation in dual-channel motor control systems, where signal validation and fault detection are paramount for operational safety.

In industrial and safety-critical applications, PWM signal replication enables robust redundant control paths and continuous system monitoring. The technique involves capturing input PWM signal's characteristics and generating an identical output signal, maintaining precise timing relationships.

This capability proves especially valuable in power electronics applications, where synchronized switch control and multiple phase-aligned outputs are essential. The implementation allows for signal buffering, regeneration, and fan-out to multiple subsystems while maintaining signal integrity and noise immunity.

In motor control applications, this approach enables redundant gate drive signals and dual-channel safety systems, critical for fault-tolerant operation. The system continuously monitors and validates signals, providing real-time verification and performance monitoring capabilities. This is particularly important in industrial equipment where safety and reliability are paramount.

The implementation is straightforward yet powerful, utilizing timer capture features to read input PWM parameters and generate identical output signals. This ensures precise replication of both period and duty cycle values, maintaining signal integrity throughout the system. The resulting solution offers comprehensive benefits including signal validation, safety redundancy, distributed control capabilities, and robust system monitoring functions.

A Timer instance can be configured to generate feedback based PWM using the following steps:

- Configure Timer in up-counting mode.
- Configure CC0 channel for Edge Capture for both rising and falling edges.
- Input the reference PWM signal on the CC0 channel.
- Configure CC1 in output mode to generate feedback-based PWM.
- Configure CCCTL.CC2SELU for CC1 channel as CC0 and CCACT.CC2UACT field as CCP output toggle. This will toggle the CC1 output based on CC0 capture events.
- Capture event on CC0 channel will be generated by the input reference signal.

```
/* Configuration Sequence to Generate Feedback-based PWM */
static const DL_TimerA_ClockConfig gCAPTURE_0ClockConfig = {
    .clockSel    = DL_TIMER_CLOCK_BUSCLK,
    .divideRatio = DL_TIMER_CLOCK_DIVIDE_1,
    .prescale = 0U
};

static const DL_TimerA_CaptureConfig gCAPTURE_0CaptureConfig = {
    .captureMode    = DL_TIMER_CAPTURE_MODE_EDGE_TIME_UP,
    .period         = CAPTURE_0_INST_LOAD_VALUE,
    .startTimer     = DL_TIMER_STOP,
    .edgeCaptMode   = DL_TIMER_CAPTURE_EDGE_DETECTION_MODE_EDGE,//Enable Edge Capture on both
rising and falling edges for CC0
    .inputChan      = DL_TIMER_INPUT_CHAN_0,
    .inputInvMode   = DL_TIMER_CC_INPUT_INV_NOINVERT,
};

SYSCONFIG_WEAK void SYSCFG_DL_CAPTURE_0_init(void) {

    DL_TimerA_setClockConfig(CAPTURE_0_INST,
        (DL_TimerA_ClockConfig *) &gCAPTURE_0ClockConfig);

    DL_TimerA_initCaptureMode(CAPTURE_0_INST,
        (DL_TimerA_CaptureConfig *) &gCAPTURE_0CaptureConfig);
```

```
DL_TimerA_setCounterControl(CAPTURE_0_INST,DL_TIMER_CZC_CCCTL0_ZCOND,DL_TIMER_CAC_CCCTL0_ACOND,DL_TI
MER_CLC_CCCTL0_LCOND);

    DL_TimerA_enableInterrupt(CAPTURE_0_INST , DL_TIMERA_INTERRUPT_CC0_UP_EVENT);

    CAPTURE_0_INST->COUNTERREGS.CCACT_01[1]=0x00018000;//Enable CC output Toggle for Secondary CC
event. Secondary Event for CC1 will be generated when CC0 captures an edge
    DL_TimerA_setCaptureCompareValue(CAPTURE_0_INST, 0, DL_TIMER_CC_1_INDEX);
    DL_TimerA_enableClock(CAPTURE_0_INST);
    DL_TimerA_setCCPDirection(CAPTURE_0_INST , DL_TIMER_CC1_OUTPUT);
}
```

The configurations above are for a scenario where the period as well as the duty cycle of the reference PWM is changing.

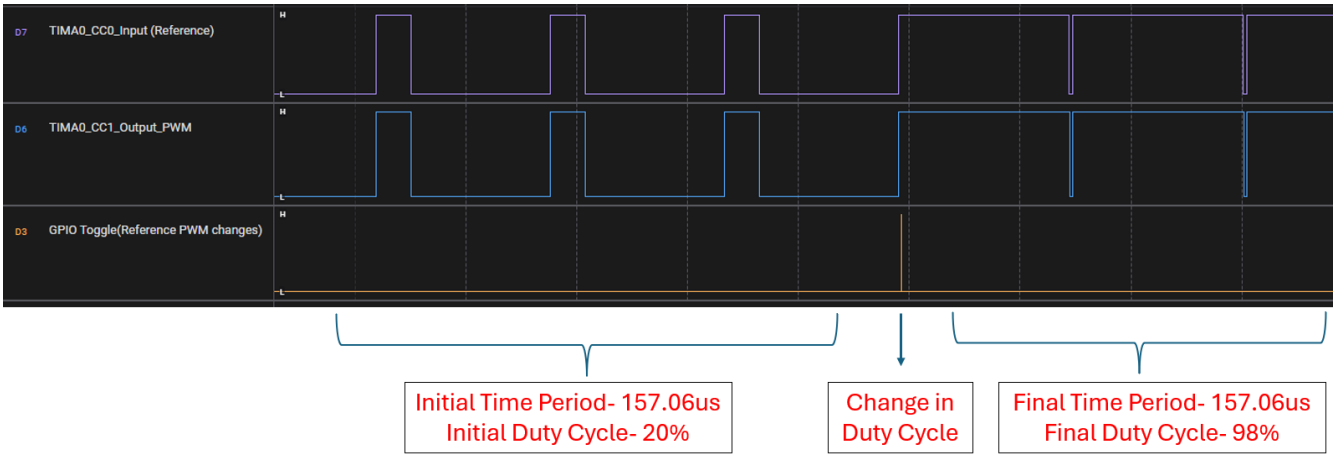

Figure 5-1. Only Duty Cycle of the Reference PWM Is Changing

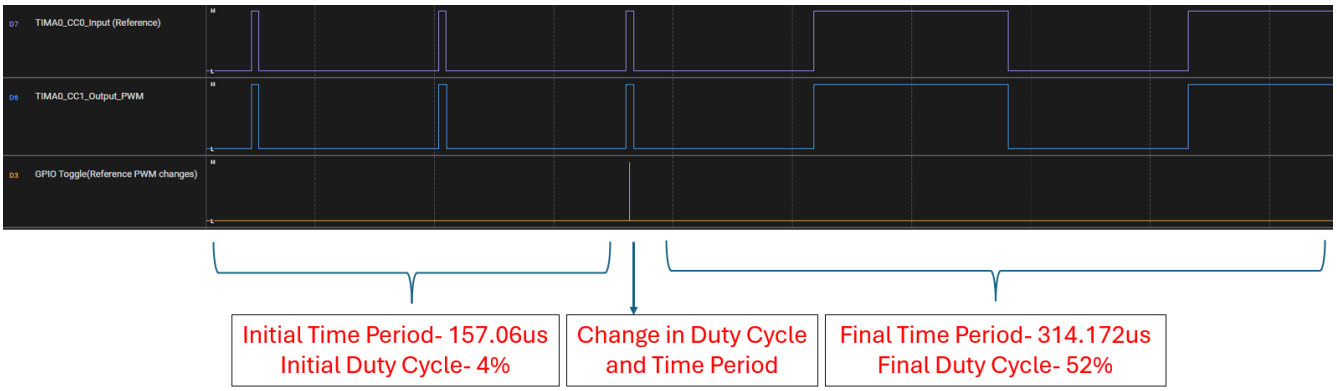

Figure 5-2. Both Duty Cycle and the Time Period of the Reference PWM Is Changing

5.2 Delayed PWM Signal Generation Using an Input Reference

Feedback-based PWM generation with controlled delay offers versatile solutions across multiple application domains, particularly in sophisticated motor control and power electronics systems. This technique enables precise phase-shifted PWM generation, critical for dual motor synchronization and dead-time insertion in complementary signals. The implementation allows for accurate timing control, essential in LED display application and multi-phase motor drive systems.

In power electronics applications, this methodology proves invaluable for multi-phase DC-DC converters and phase-shifted bridge converters. The ability to introduce controlled delays enables efficient dead-time insertion for half-bridge drivers and facilitates interleaved power supply operation. The system can precisely generate output PWM signals with specific timing offsets.

This capability extends to signal processing applications, where controlled-delay signal repeaters and timing compensation in distributed systems are essential. The technique supports sophisticated clock signal distribution with phase control and enables robust cascaded control systems. Specific implementations include synchronous buck converters with phase shifting and multi-channel LED dimming applications requiring precise phase control.

The implementation utilizes timer capture features to measure input signal timing and generates delayed outputs with precise phase relationships. This approach ensures accurate timing delays while maintaining synchronized operation across the system. The resulting solution offers significant benefits including precise phase control, deterministic timing delays, and enhanced noise immunity through signal retiming.

The solution is particularly valuable in synchronized power stage control and applications requiring precise phase relationships between multiple PWM signals.

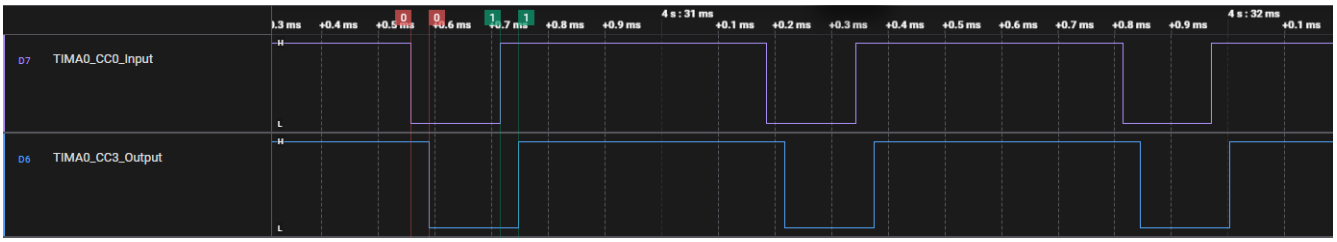

Figure 5-3. PWM Signal Generated with a Fixed Delay with an Input Reference

The configuration sequence is described below-

- Configure the Load value and CM as Up-Counting Mode.
- Configure CC0 for rising edge detection and CC1 for falling edge detection in Capture Mode. Configure Zero Condition (CCCTL.ZCOND) such that CC0 rising edge generates zero condition.
- Configure IFCTL.ISEL for CC1 as 0x2 so that the input on CC0 is also fed to the CC1 channel.
- The reference input PWM is fed to the CC0 channel. Internally CC0 input is also fed to the CC1 channel
- Configure CC4 as the secondary CC channel for CC3, this means that a CC4 compare event will generate a secondary compare event for CC3.
- Enable CC0_UP and CC1_UP interrupts.
- Configure CCACT for CC3 such that CUACT will drive the CC output HIGH and CC2UACT drives the CC Output LOW.
- Configure CC3 as output. The final output signal with a fixed delay will be generated by CC3.
- In the Interrupt Service Routine, use CC0_UP interrupt to update the CC3 value with a delay as per application requirement. Use CC1_UP interrupt to update the CC4 value with a delay as per application.

Note

In applications with multiple ISRs, higher priority interrupts, or high PWM frequencies, care has to be taken to increase the priority of Timer ISR and optimize the ISR code so that CC value can be updated as fast as possible.

```
/* Configuration Sequence for Delayed Timer Generation */

static const DL_TimerA_ClockConfig gCAPTURE_0ClockConfig = {
    .clockSel    = DL_TIMER_CLOCK_BUSCLK,
    .divideRatio = DL_TIMER_CLOCK_DIVIDE_1,
    .prescale = 0U
};

static const DL_TimerA_CaptureConfig gCAPTURE_0CaptureConfig = {
    .captureMode    = DL_TIMER_CAPTURE_MODE_EDGE_TIME_UP,
    .period         = 65535,
    .startTimer     = DL_TIMER_STOP,
    .edgeCaptMode   = DL_TIMER_CAPTURE_EDGE_DETECTION_MODE_RISING,//Enable Rising Edge Capture for
CC0
    .inputChan      = DL_TIMER_INPUT_CHAN_0,
    .inputInvMode   = DL_TIMER_CC_INPUT_INV_NOINVERT,
};
```

```
SYSCONFIG_WEAK void SYSCFG_DL_CAPTURE_0_init(void) {

    DL_TimerA_setClockConfig(CAPTURE_0_INST,
        (DL_TimerA_ClockConfig *) &gCAPTURE_0ClockConfig);

    DL_TimerA_initCaptureMode(CAPTURE_0_INST,
        (DL_TimerA_CaptureConfig *) &gCAPTURE_0CaptureConfig);

DL_TimerA_setCounterControl(CAPTURE_0_INST,DL_TIMER_CZC_CCCTL0_ZCOND,DL_TIMER_CAC_CCCTL0_ACOND,DL_TI
MER_CLC_CCCTL0_LCOND);

    DL_TimerA_enableInterrupt(CAPTURE_0_INST , DL_TIMERA_INTERRUPT_CC0_UP_EVENT|
DL_TIMERA_INTERRUPT_CC1_UP_EVENT);
    CAPTURE_0_INST->COUNTERREGS.CCCTL_01[0]=0x21001;
    CAPTURE_0_INST->COUNTERREGS.CCCTL_01[1]=0x20002;
    CAPTURE_0_INST->COUNTERREGS.CCCTL_23[1]=0x01000000;
    DL_TimerA_setCaptureCompareInput(CAPTURE_0_INST, DL_TIMER_CC_INPUT_INV_NOINVERT,
DL_TIMER_CC_IN_SEL_CCP0, DL_TIMER_CC_1_INDEX);
    CAPTURE_0_INST->COUNTERREGS.CCACT_23[1]=0x10200;
    DL_TimerA_enableClock(CAPTURE_0_INST);
    DL_TimerA_setCCPDirection(CAPTURE_0_INST , DL_TIMER_CC3_OUTPUT);
}
```

```
/* Interrupt Service Routine to Update the CC Values with a Fixed Delay of 1000 Cycles as an
Example */
void CAPTURE_0_INST_IRQHandler(void)
{
    switch (DL_TimerA_getPendingInterrupt(CAPTURE_0_INST)) {
        case DL_TIMERA_IIDX_CC0_UP:
            CC0_value=DL_Timer_getCaptureCompareValue(CAPTURE_0_INST, DL_TIMER_CC_0_INDEX);
            DL_Timer_setCaptureCompareValue(CAPTURE_0_INST, CC0_value+1000, DL_TIMER_CC_3_INDEX);
            CC0_int++;
            break;
        case DL_TIMERA_IIDX_CC1_UP:
            CC1_value=DL_Timer_getCaptureCompareValue(CAPTURE_0_INST, DL_TIMER_CC_1_INDEX);
            DL_Timer_setCaptureCompareValue(CAPTURE_0_INST, CC1_value+1000, DL_TIMER_CC_4_INDEX);
            CC1_int++;
            break;
        default:
            break;
    }
}
```

6 Delayed Timer Start: Synchronized Timer Instances Initiation with Configurable Delays

Timers with intentional start-time offsets enable controlled phase relationships that are critical for efficiency, stability and signal integrity in some applications involving motor control, and sensor data polling or sampling. This can be achieved when using two or more timers by using them in synchronized fashion using the cross-triggering mechanism, as described below:

- Configure the timers in a software based cross-triggering mechanism, as per Section 3 .
- Based on the application the timer that is to be enabled first can be configured as the primary timer.
- Configure CC value depending on the required start-time offsets.
- Generate a Software Cross-Trigger within the CC event ISR to enable the Secondary Timer.

Note

Due to delay limitations that prevent setting delays on the second Timer when specific duty cycles are required for the first PWM output, using the CC4/5 event of the first PWM would provide the necessary flexibility for configuring the delay.

For example, an offset of 10ms is needed between enabling TIMA0 and TIMA1. If TIMA1 is required to be enabled first, configure TIMA1 as the primary timer and TIMA0 as the secondary timer. Configure the CC value for TIMA1 such that CC event is generated exactly 10ms after enable. Within this CC event a Software Cross Trigger can be generated to enable TIMA0 exactly after 10ms. This means, with CLKSEL as LFCLK (32KHz) TIMA1's CC value can be configured as 320 with Timer in Up Counting mode, this will generate a CC event after ~10ms (320 cycles × (1/32000) seconds = 0.01 seconds = 10ms). This CC event will enable TIMA0 via the software based cross-triggering mechanism.

```
/* TIMA Interrupt Service Routine */
void PWM_0_INST_IRQHandler(void)
{
    switch (DL_TimerA_getPendingInterrupt(PWM_0_INST)) {
        case DL_TIMERA_IIDX_CC0_DN:
            if(interrupt_counter==0){
            DL_TimerA_generateCrossTrigger(PWM_0_INST);//Mechanism to generate Software based Cross-
Trigger
            }
            else{
                ;
            }
            interrupt_counter++;
            break;
        default:
            break;
    }
}
```

Note

While the Hardware Cross-Triggering functionality can also be utilized for this application, it is important to recognize that this mechanism will generate Cross-Trigger on every CC event. Therefore, if developers intend to utilize the hardware cross-trigger capability, it must be explicitly disabled after one cross-trigger for this application to function as intended.

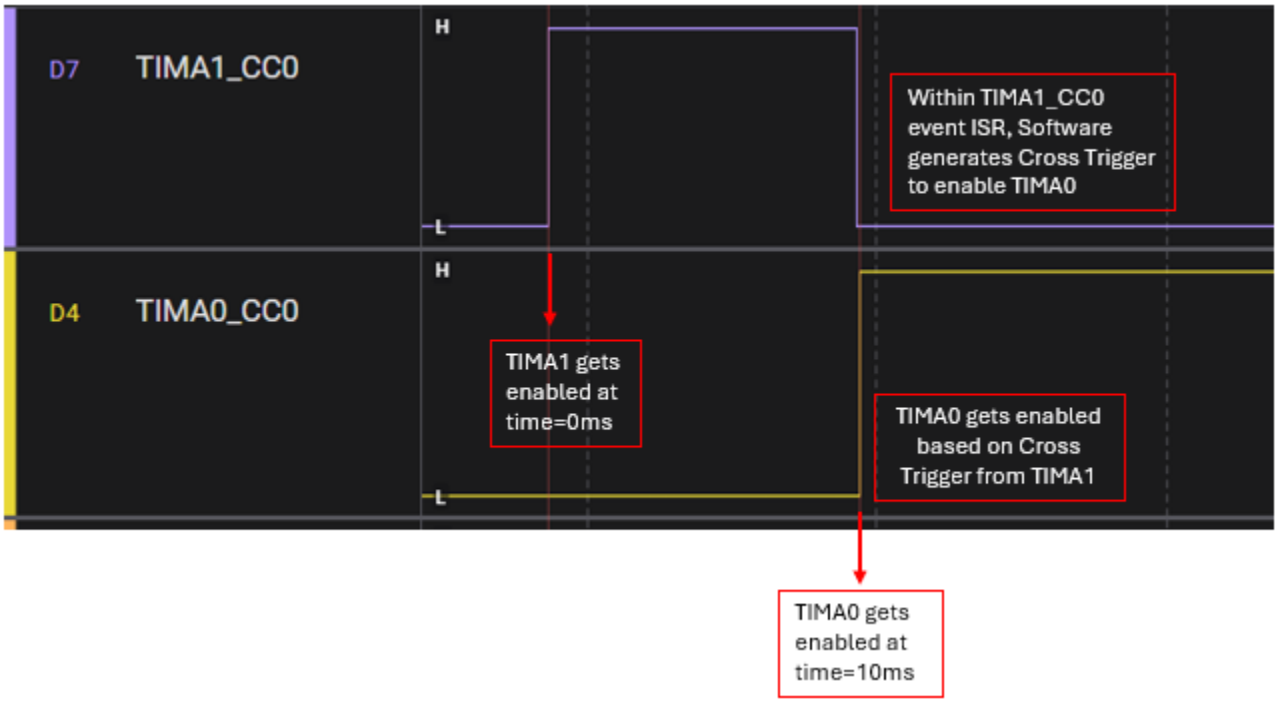

Figure 6-1. Configurable Timer Start Time Offset

7 Stopping a Running Timer Based on Hardware Events

Using Hardware Events to stop a running timer provides a **low-latency**, precise and autonomous control over timing functions. This enables real-time measurement and response, crucial in sensing, safety and control systems. Figure 7-1 shows the flowchar to stop the running timer.

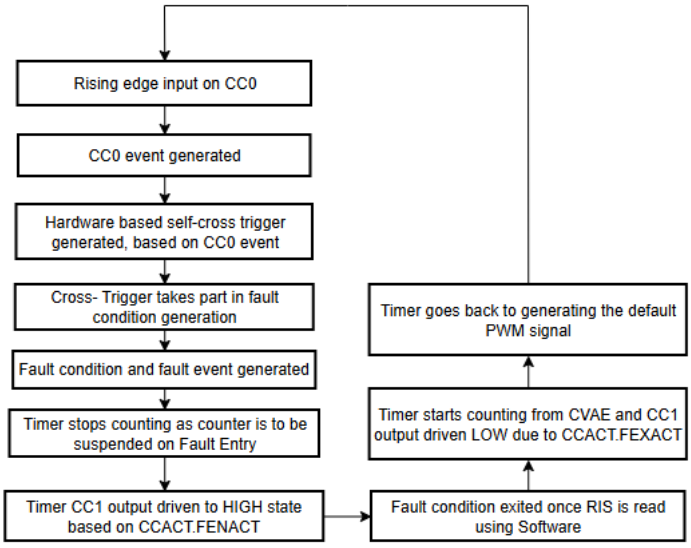

Figure 7-1. Flowchart Representing How Hardware Stops a Running Timer

The following configurations in Advanced Timer (TIMA) can be done to achieve this requirement:

- Configure TIMA to generate a self-cross trigger on CC event.
- Configure CC0 for rising edge capture. An external rising edge on the CC0 pin will generate a CC0 event and a self-cross trigger.
- Configure the hardware-based fault-sensing mechanism for the TIMA instance being used.
- Configure the FCTL.TFIM bit to '1' meaning the selected trigger would participate in fault condition generation, this would lead to the cross-trigger generating a fault condition to stop the timer.
- Configure the FCTL.FL to '1' to enable Fault Latch Mode, this would mean the Overall fault condition is dependent on the CPU_INT.RIS.F bit.
- Configure fault entry to suspend the counter, this would stop a running counter.

```
/* Configuration Sequence to Stop a Running Timer Using Hardware Events */
static const DL_TimerA_ClockConfig gPWM_0ClockConfig = {
    .clockSel = DL_TIMER_CLOCK_BUSCLK,
    .divideRatio = DL_TIMER_CLOCK_DIVIDE_1,
    .prescale = 0U
};

static const DL_TimerA_PWMConfig gPWM_0Config = {
    .pwmMode = DL_TIMER_PWM_MODE_EDGE_ALIGN_UP,
    .period = 1600,
    .isTimerWithFourCC = false,
    .startTimer = DL_TIMER_STOP,
};
SYSCONFIG_WEAK void SYSCFG_DL_PWM_0_init(void) {

    DL_TimerA_setClockConfig(
        PWM_0_INST, (DL_TimerA_ClockConfig *) &gPWM_0ClockConfig);
    DL_TimerA_initPWMMode(
        PWM_0_INST, (DL_TimerA_PWMConfig *) &gPWM_0Config);
    // Set Counter control to the smallest CC index being used

DL_TimerA_setCounterControl(PWM_0_INST,DL_TIMER_CZC_CCCTL0_ZCOND,DL_TIMER_CAC_CCCTL0_ACOND,DL_TIMER_
CLC_CCCTL0_LCOND);

    DL_TimerA_setCaptCompUpdateMethod(PWM_0_INST, DL_TIMER_CC_UPDATE_METHOD_IMMEDIATE,
DL_TIMERA_CAPTURE_COMPARE_0_INDEX);
```

```
    DL_TimerA_setCaptureCompareOutCtl(PWM_0_INST, DL_TIMER_CC_OCTL_INIT_VAL_LOW,
        DL_TIMER_CC_OCTL_INV_OUT_DISABLED, DL_TIMER_CC_OCTL_SRC_FUNCVAL,
        DL_TIMERA_CAPTURE_COMPARE_1_INDEX);

    DL_TimerA_setCaptCompUpdateMethod(PWM_0_INST, DL_TIMER_CC_UPDATE_METHOD_IMMEDIATE,
DL_TIMERA_CAPTURE_COMPARE_1_INDEX);
    DL_TimerA_setCaptureCompareValue(PWM_0_INST, 500, DL_TIMER_CC_1_INDEX);//Set CC1 value, this is
to generate PWM using CC1

//Configure Fault such that Cross Trigger generates a fault condition, which will be latched unless
the Fault RIS is cleared
    DL_TimerA_setFaultConfig(PWM_0_INST, DL_TIMER_FAULT_CONFIG_TFIM_ENABLED|
DL_TIMER_FAULT_CONFIG_FL_LATCH_SW_CLR|
                                        DL_TIMER_FAULT_CONFIG_FI_DEPENDENT|
DL_TIMER_FAULT_CONFIG_FIEN_ENABLED);

    DL_TimerA_configFaultOutputAction(PWM_0_INST,
                                      DL_TIMER_FAULT_ENTRY_CCP_HIGH,
                                      DL_TIMER_FAULT_EXIT_CCP_LOW, DL_TIMER_CC_1_INDEX);
//Generating a Fault condition will stop the Counter if Fault Entry Action is set as Fault Counter
Suspend Counting
    DL_TimerA_configFaultCounter(PWM_0_INST,
                                 DL_TIMERA_FAULT_ENTRY_CTR_SUSP_COUNT,
                                 DL_TIMERA_FAULT_EXIT_CTR_CVAE_ACTION);
    DL_TimerA_setFaultSourceConfig(
            PWM_0_INST, DL_TIMERA_FAULT_SOURCE_EXTERNAL_0_SENSE_HIGH);
    PWM_0_INST->COUNTERREGS.CCCTL_01[0]=0x20001;//CC0 configured for rising edge capture
    DL_TimerA_enableClock(PWM_0_INST);
    DL_TimerA_setCCPDirection(PWM_0_INST ,  DL_TIMER_CC1_OUTPUT );
    DL_TimerA_configCrossTrigger(PWM_0_INST, DL_TIMER_CROSS_TRIG_SRC_CCU0,
                                 DL_TIMER_CROSS_TRIGGER_INPUT_ENABLED,
DL_TIMER_CROSS_TRIGGER_MODE_ENABLED
        );//Configuration to Generate Hardware Based Cross Trigger on CC0 Event
    DL_TimerA_setCaptureCompareInput(PWM_0_INST, DL_TIMER_CC_INPUT_INV_NOINVERT,
DL_TIMER_CC_IN_SEL_TRIG, DL_TIMER_CC_1_INDEX);
    DL_TimerA_setExternalTriggerEvent(PWM_0_INST,DL_TIMER_EXT_TRIG_SEL_TRIG_1);//This is to receive
the self cross trigger generated by TIMA1
    DL_TimerA_enableExternalTrigger(PWM_0_INST);
}
```

Note

The application described above utilizes a rising edge on CC0 to generate a self-cross-trigger and consequently a fault condition to stop the timer. Similarly, events originating from other peripherals received by TIMA via the event fabric, for example, GPIO events, can also be utilized to generate a self-cross-trigger and consequently a fault condition to stop the timer using hardware.

8 Dynamic PWM Update: Duty Cycle and Time Period Adjustment

The ability to modify PWM characteristics during runtime enables precise control across diverse applications, offering flexibility in both frequency and duty cycle adjustments. By dynamically updating Timer Compare (CC) and Load values, systems can achieve real-time control while maintaining stable operation. This capability becomes particularly powerful when implemented using shadow register features, ensuring glitch-free transitions at predetermined update points.

In motor control systems, this functionality enables smooth speed ramping through gradual frequency adjustments and precise torque control via duty cycle modulation. The same mechanism proves invaluable in LED and lighting applications, where smooth brightness transitions and complex dimming patterns are essential for quality user experience. Power supply applications benefit from adaptive voltage regulation and dynamic frequency adjustment for optimal efficiency, while temperature control systems utilize this capability for precise thermal management through PWM-based heater control and fan speed modulation.

8.1 Shadow Load and Shadow Compare Features

TIMA provides shadow registers for configuring CC and LOAD value to support dynamically changing PWM duty and period, as shown in Figure 8-1. The shadow register mechanism plays a crucial role in maintaining system stability by ensuring parameter updates occur at safe transition points, typically at period boundaries. This systematic approach prevents PWM output glitches, irregular pulse widths, and unexpected frequency variations that could otherwise occur with direct register updates. The result is a robust solution for applications requiring smooth transitions and precise timing control, particularly valuable in systems where predictable behavior and stable operation are paramount.

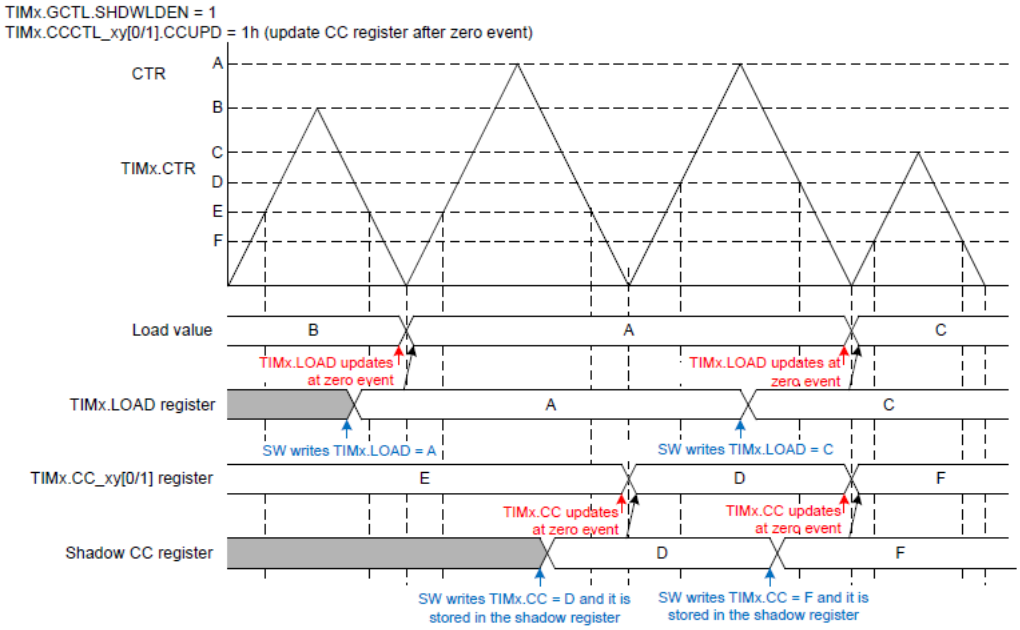

Figure 8-1. Shadow Load and Shadow Compare Taking Effect at Zero Event in Up/Down Mode

Note

Without shadow registers, unexpected PWM outputs (0% or 100% duty cycle) can occur randomly when applications dynamically update the PWM duty cycle. To mitigate this issue, enable an interrupt that triggers on load or zero events for this timer, then perform duty cycle updates within the ISR while implementing appropriate max/min duty cycle constraints to account for interrupt processing delays. Additionally, consider using DMA update CC value to reduce CPU overhead and interrupt processing delays.

The implementation offers remarkable flexibility while maintaining system integrity, making it ideal for applications ranging from simple brightness control to complex motor drive systems. Whether implementing soft-start sequences, thermal management systems, or sophisticated PID control loops, the ability to dynamically modify PWM parameters while ensuring glitch-free operation provides a powerful tool for modern embedded system design.

Shadow features can be used to update the CC and load values at fixed reference points. When the Load value needs to be changed, TIMx.LOAD value can be written directly. Internally the Load value will only get updated on the Zero event irrespective of when it is updated by software. When the CC value need to be changed, write to the TIMx.CC value directly. Internally the CC value will only get updated on the event configured in CC update method irrespective of when it is updated by software. Shadow Features are enabled by writing to TIMx.GCTL.SHDWLDEN (=1) register.

```
/* Configuration Sequence for Shadow Load and Shadow CC Update */
static const DL_TimerA_ClockConfig gPWM_0ClockConfig = {
    .clockSel = DL_TIMER_CLOCK_BUSCLK,
    .divideRatio = DL_TIMER_CLOCK_DIVIDE_1,
    .prescale = 0U
};

static const DL_TimerA_PWMConfig gPWM_0Config = {
    .pwmMode = DL_TIMER_PWM_MODE_EDGE_ALIGN_UP,
    .period = 5000,
    .isTimerWithFourCC = false,
    .startTimer = DL_TIMER_STOP,
};

SYSCONFIG_WEAK void SYSCFG_DL_PWM_0_init(void) {

    DL_TimerA_setClockConfig(
        PWM_0_INST, (DL_TimerA_ClockConfig *) &gPWM_0ClockConfig);

    DL_TimerA_initPWMMode(
        PWM_0_INST, (DL_TimerA_PWMConfig *) &gPWM_0Config);

    // Set Counter control to the smallest CC index being used

DL_TimerA_setCounterControl(PWM_0_INST,DL_TIMER_CZC_CCCTL0_ZCOND,DL_TIMER_CAC_CCCTL0_ACOND,DL_TIMER_
CLC_CCCTL0_LCOND);

    DL_TimerA_setCaptureCompareOutCtl(PWM_0_INST, DL_TIMER_CC_OCTL_INIT_VAL_LOW,
        DL_TIMER_CC_OCTL_INV_OUT_DISABLED, DL_TIMER_CC_OCTL_SRC_FUNCVAL,
        DL_TIMERA_CAPTURE_COMPARE_0_INDEX);

    DL_TimerA_setCaptureCompareOutCtl(PWM_0_INST, DL_TIMER_CC_OCTL_INIT_VAL_LOW,
        DL_TIMER_CC_OCTL_INV_OUT_DISABLED, DL_TIMER_CC_OCTL_SRC_ZERO,
        DL_TIMERA_CAPTURE_COMPARE_1_INDEX);//CC1 output will go high for 1 TIMCLK cycle when
Counter value is Zero, indicating a Zero Event

    DL_TimerA_setCaptCompUpdateMethod(PWM_0_INST, DL_TIMER_CC_UPDATE_METHOD_ZERO_EVT,
DL_TIMERA_CAPTURE_COMPARE_0_INDEX);//Update the CC value on the subsequent Zero Event

    DL_TimerA_setCaptureCompareValue(PWM_0_INST, 2000, DL_TIMER_CC_0_INDEX);

    DL_TimerA_enableClock(PWM_0_INST);

    DL_TimerA_setCCPDirection(PWM_0_INST , DL_TIMER_CC0_OUTPUT);
}
```

```
/* SDK API Used to Update Load and CC Values in Main Application */
 DL_TimerA_setLoadValue(PWM_0_INST, 10000);
 DL_TimerA_setCaptureCompareValue(PWM_0_INST, 2000, DL_TIMER_CC_0_INDEX);
```

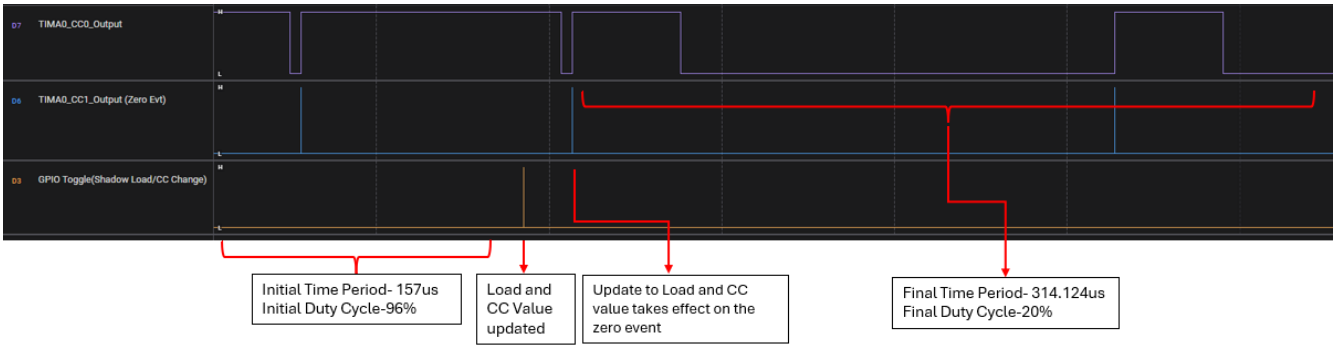

Figure 8-2. Waveform Showing Change in PWM Period and Duty Cycle

User can configure CC and CCACT Update events as per the application requirement as shown in the Figure 8-3.

Bit Field	Value	Description/Comment
CCUPD / CCACTUPD	0	The value written to TIMx.CC register take effect immediately.
	1	The value written to the TIMx.CC register is stored in a shadow compare register and gets transferred to the TIMx.CC register in the TIMCLK cycle following a zero event (TIMx.CTR value equals 0).
	2	The value written to the TIMx.CC register is stored in a shadow compare register and gets transferred to the TIMx.CC register in the TIMCLK cycle following a compare (down) event (TIMx.CTR value equals TIMx.CC)
	3	The value written to the TIMx.CC register is stored in a shadow compare register and gets transferred to the TIMx.CC register in the TIMCLK cycle following a compare (up) event (TIMx.CTR value equals TIMx.CC)
	4	The value written to the TIMx.CC register is stored in a shadow compare register and gets transferred to the TIMx.CC register in the TIMCLK cycle following a zero or load event (TIMx.CTR value equals 0 or TIMx.CTR equals TIMx.LOAD). **Note: this update mechanism is defined for use only in up/down counting mode.**
	5	The value written to the TIMx.CC register is stored in a shadow compare register and gets transferred to the TIMx.CC register in the TIMCLK cycle following a zero event and the repeat count equaling zero (TIMx.CTR value equals 0 and TIMx.RC equals 0)
	6	The value written to the TIMx.CC register is stored in a shadow compare register, and gets transferred to the TIMx.CC register in the TIMCLK cycle following a trigger pulse. See Section 27.2.7.

Figure 8-3. Shadow Compare and Action Update Behavior

Shadow Registers Operational Characteristics

- Default State: The shadow register initializes to 0x0 by default when no user-defined initial value is specified to CC or LOAD register after shadow function enabled

Note

If users set CC value before Shadow CC function enabled, and not set a second CC value after Shadow CC enabled, then after next user-defined EVENT occurs, the CC value will come back to zero and keep zero.

- Runtime Update Logic: CC or LOAD value modifications made after shadow feature enabled remain in a pending state until the next occurrence of the user-defined EVENT triggers

Note

If users set initial CC value after Shadow CC function enabled, then before next user-defined EVENT occurs, the CC value will keep zero.

8.2 Arbitrary Signal Generation with DMA

The timer module can generate DMA requests via the event fabric. This feature allows the user to modify the waveform output by a timer peripheral on the fly, by adjusting the timer peripheral registers content. For example, it allows the user to update the TIMx.LOAD register content to adjust the waveform frequency or update the TIMx.CC register to adjust the signal duty-cycle.

Standard PWM generation typically relies on fixed frequency and duty cycle parameters defined in the timer's load and capture-compare registers respectively. While these registers can be manually updated through software intervention, this approach introduces significant CPU overhead and potential timing uncertainties.

This section demonstrates an optimized approach that leverages DMA (Direct Memory Access) capabilities in conjunction with timer peripherals to achieve:

- Deterministic waveform generation
- Minimal CPU intervention
- Precise timing control
- Automatic parameter updates

Rather than relying on CPU-driven updates at each timer iteration, the solution utilizes hardware-triggered DMA transfers to automatically update both frequency and duty cycle parameters. This methodology ensures:

1. Precise timing relationships between successive PWM cycles
2. Predictable and jitter-free waveform generation
3. Efficient system resource utilization
4. Dynamically updating PWM duty and period without shadow features.

This section details the implementation strategy, showcasing how the timer's update events trigger automated DMA transfers for seamless register updates, eliminating CPU overhead while maintaining precise waveform control.

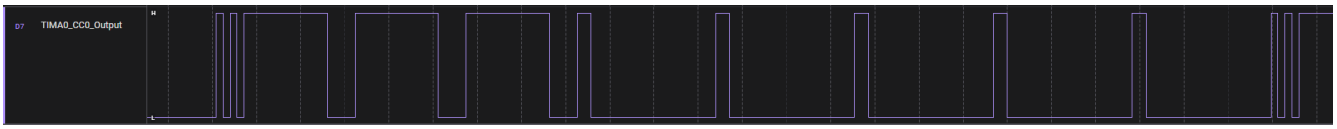

Figure 8-4. Arbitrary Signal Generation Using DMA

The application flow is described below:

- Configure TIMER as an event publisher to publish events to trigger DMA, use FPUB0 to trigger DMA_CHAN0 and FPUB1 to trigger DMA_CHAN1.
- DMA_CHAN0 will be used to write to the LOAD Register, whereas DMA_CHAN1 will be used to write to the CC Register.
- Changing the LOAD value will vary the Period, whereas changing the CC value will vary the Duty Cycle.
- Configure CC update event as Zero event, meaning the CC value will get updated on the next zero event.
- Configure DMA as a subscriber in Repeated single transfer mode with 16-bit as the width.
- Configure DMA source address as increment and destination address as unchanged.
- For DMA_CHAN0 configure source address as the buffer containing the LOAD values and destination address as the TIMx.LOAD register address.
- For DMA_CHAN1 configure source address as the buffer containing the CC values and destination address as the TIMx.CC register address.
- For CC1, OCTL.CCPO has been configured as 0x4. Hence CC1 output signal will toggle and set to a HIGH value for 1 TIMCLK cycle when counter value is zero, after that CC1 output will be cleared by hardware. This has been used to check the Load and CC update timing with respect to the zero event. This feature can also be used for debugging purposes in various other applications.

```
/* Configuration Sequence for Timer */
static const DL_TimerA_ClockConfig gPWM_0ClockConfig = {
    .clockSel = DL_TIMER_CLOCK_BUSCLK,
    .divideRatio = DL_TIMER_CLOCK_DIVIDE_1,
    .prescale = 0U
```

```
};

static const DL_TimerA_PWMConfig gPWM_0Config = {
    .pwmMode = DL_TIMER_PWM_MODE_EDGE_ALIGN_UP,
    .period = 5000,
    .isTimerWithFourCC = false,
    .startTimer = DL_TIMER_STOP,
};

SYSCONFIG_WEAK void SYSCFG_DL_PWM_0_init(void) {

    DL_TimerA_setClockConfig(
        PWM_0_INST, (DL_TimerA_ClockConfig *) &gPWM_0ClockConfig);

    DL_TimerA_initPWMMode(
        PWM_0_INST, (DL_TimerA_PWMConfig *) &gPWM_0Config);

    // Set Counter control to the smallest CC index being used

DL_TimerA_setCounterControl(PWM_0_INST,DL_TIMER_CZC_CCCTL0_ZCOND,DL_TIMER_CAC_CCCTL0_ACOND,DL_TIMER_
CLC_CCCTL0_LCOND);

    DL_TimerA_setCaptureCompareOutCtl(PWM_0_INST, DL_TIMER_CC_OCTL_INIT_VAL_LOW,
        DL_TIMER_CC_OCTL_INV_OUT_DISABLED, DL_TIMER_CC_OCTL_SRC_FUNCVAL,
        DL_TIMERA_CAPTURE_COMPARE_0_INDEX);
    DL_TimerA_setCaptureCompareOutCtl(PWM_0_INST, DL_TIMER_CC_OCTL_INIT_VAL_LOW,
        DL_TIMER_CC_OCTL_INV_OUT_DISABLED, DL_TIMER_CC_OCTL_SRC_ZERO,
        DL_TIMERA_CAPTURE_COMPARE_1_INDEX);

    DL_TimerA_setCaptCompUpdateMethod(PWM_0_INST, DL_TIMER_CC_UPDATE_METHOD_ZERO_EVT,
DL_TIMERA_CAPTURE_COMPARE_0_INDEX);//Update CC on Zero Event

    DL_TimerA_setCaptureCompareValue(PWM_0_INST, 2000, DL_TIMER_CC_0_INDEX);
    DL_TimerA_setCaptureCompareValue(PWM_0_INST, 4000, DL_TIMER_CC_1_INDEX);

    DL_TimerA_enableClock(PWM_0_INST);

    DL_Timer_enableEvent(PWM_0_INST, DL_TIMER_EVENT_ROUTE_1, DL_TIMERA_INTERRUPT_ZERO_EVENT);//Zero
Event triggers the DMA CHAN0
    DL_Timer_setPublisherChanID(PWM_0_INST, DL_TIMER_PUBLISHER_INDEX_0, 1);//Timer publishing event
to DMA to carry out Load Value update

    DL_Timer_enableEvent(PWM_0_INST, DL_TIMER_EVENT_ROUTE_2, DL_TIMERA_INTERRUPT_ZERO_EVENT);//Zero
Event triggers the DMA CHAN1
    DL_Timer_setPublisherChanID(PWM_0_INST, DL_TIMER_PUBLISHER_INDEX_1, 12);//Timer publishing
event to DMA to carry out CC value update

    DL_TimerA_setCCPDirection(PWM_0_INST , DL_TIMER_CC0_OUTPUT|DL_TIMER_CC1_OUTPUT);
}
```

```
/* Configuration Sequence for DMA */
static const DL_DMA_Config gDMA_CH0Config = {
    .transferMode   = DL_DMA_FULL_CH_REPEAT_SINGLE_TRANSFER_MODE,
    .extendedMode   = DL_DMA_NORMAL_MODE,
    .destIncrement  = DL_DMA_ADDR_UNCHANGED,
    .srcIncrement   = DL_DMA_ADDR_INCREMENT,
    .destWidth      = DL_DMA_WIDTH_HALF_WORD,
    .srcWidth       = DL_DMA_WIDTH_HALF_WORD,
    .trigger        = DMA_GENERIC_SUB0_TRIG,
    .triggerType    = DL_DMA_TRIGGER_TYPE_EXTERNAL,
};
static const DL_DMA_Config gDMA_CH1Config = {
    .transferMode   = DL_DMA_FULL_CH_REPEAT_SINGLE_TRANSFER_MODE,
    .extendedMode   = DL_DMA_NORMAL_MODE,
    .destIncrement  = DL_DMA_ADDR_UNCHANGED,
    .srcIncrement   = DL_DMA_ADDR_INCREMENT,
    .destWidth      = DL_DMA_WIDTH_HALF_WORD,
    .srcWidth       = DL_DMA_WIDTH_HALF_WORD,
    .trigger        = DMA_GENERIC_SUB1_TRIG,
    .triggerType    = DL_DMA_TRIGGER_TYPE_EXTERNAL,
};

void SYSCFG_DL_DMA_CH0_init(void)
{
    DL_DMA_initChannel(DMA, DMA_CH0_CHAN_ID , (DL_DMA_Config *) &gDMA_CH0Config);
    DL_DMA_initChannel(DMA, DMA_CH1_CHAN_ID , (DL_DMA_Config *) &gDMA_CH1Config);
    DL_DMA_clearInterruptStatus(DMA, DL_DMA_INTERRUPT_CHANNEL0);
```

```
        DL_DMA_enableInterrupt(DMA, DL_DMA_INTERRUPT_CHANNEL0);
        DL_DMA_setSubscriberChanID(DMA, DL_DMA_SUBSCRIBER_INDEX_0, 0x01);//DMA subscribing to Timer
event
        DL_DMA_setSubscriberChanID(DMA, DL_DMA_SUBSCRIBER_INDEX_1, 12);//DMA subscribing to Timer
event
}

void SYSCFG_DL_DMA_init(void){
    SYSCFG_DL_DMA_CH0_init();
}
```

```
/* Application Code for Timer and DMA */
uint16_t Load_vals[10] = {
    0x3E8,    // 1000
    0x3E8,    // 1000
    0x1F40, // 8000
    0x1F40, // 8000
    0x1F40, // 8000
    0x2710,   // 10000
    0x2710,   // 10000
    0x2710,   // 10000
    0x2710,   // 10000
    0x2710,   // 10000


};
uint16_t CC_vals[10] = { 0x1F4,    // 500
                         0x1F4,    // 500
                         0x1770,   // 6000
                         0x1770,   // 6000
                         0x1770,   // 6000
                         0x3E8,    // 1000
                         0x3E8,    // 1000
                         0x3E8,    // 1000
                         0x3E8,    // 1000
                         0x3E8,    // 1000
};
int main(void)
{
    SYSCFG_DL_init();

    DL_DMA_setDestAddr(DMA, DMA_CH0_CHAN_ID, (uint32_t)&PWM_0_INST->COUNTERREGS.LOAD);
    DL_DMA_setSrcAddr(DMA, DMA_CH0_CHAN_ID, (uint32_t)&Load_vals );
    DL_DMA_setTransferSize( DMA, DMA_CH0_CHAN_ID,10);

    DL_DMA_setDestAddr(DMA, DMA_CH1_CHAN_ID, (uint32_t)&PWM_0_INST->COUNTERREGS.CC_01[0]);
    DL_DMA_setSrcAddr(DMA, DMA_CH1_CHAN_ID, (uint32_t)&CC_vals );
    DL_DMA_setTransferSize( DMA, DMA_CH1_CHAN_ID,10);

    DL_DMA_enableChannel(DMA, DMA_CH1_CHAN_ID);
    DL_DMA_enableChannel(DMA, DMA_CH0_CHAN_ID);

    DL_TimerA_startCounter(PWM_0_INST);

    while (1) {
        ;
    }
}
```

9 Summary

This application note explores Advanced Timer (TIMA) techniques in MSPM0 microcontrollers, focusing on the TIMA module's versatile capabilities. This document covers PWM idle-low configurations, phase-shifted PWM generation, and bit-banging for UART emulation. The document demonstrates feedback-based PWM generation, synchronized timer initiation with configurable delays, and hardware-triggered timer stopping control. Key features include shadow registers for dynamic PWM updates, and DMA-based arbitrary signal generation. Practical examples with code snippets illustrate implementation strategies for modern embedded systems.

10 References

- Texas Instruments, *Advanced Timer Techniques in MSPM0 Example Projects*, FAQs.
- Texas Instruments, *MSPM0Gx51x Mixed-Signal Microcontrollers With CAN-FD Interface*, datasheet.
- Texas Instruments, *MSPM0 G-Series 80-MHz Microcontrollers Technical Reference Manual*, technical reference manual.

IMPORTANT NOTICE AND DISCLAIMER

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

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

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

TI’s products are provided subject to TI’s Terms of Sale, TI’s General Quality Guidelines, or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products. Unless TI explicitly designates a product as custom or customer-specified, TI products are standard, catalog, general purpose devices.

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

Last updated 10/2025