XMC1000, XMC4000

32-bit Microcontroller Series for Industrial Applications

Position Interface (POSIF)

AP32289

Application Note

About this document

Scope and purpose

This application note provides a brief introduction to the key features of the Position Interface (POSIF) modules, some typical application examples, and some usage hints.

Intended audience

Experienced engineers who wish to develop motor control applications with the XMC microcontroller family.

Applicable Products

- XMC1000
- XMC4000
- DAVE ™

References

The User's Manual can be downloaded from http://www.infineon.com/XMC.

DAVE™ and its resources can be downloaded from http://www.infineon.com/DAVE

Table of Contents

Table of Contents

1 Introduction to POSIF

A POSIF is a universal Position Interface unit. When used in conjunction with the CAPCOM units, CCU4 and CCU8, POSIF offers powerful solutions for motion control systems that use various position sensors or rotary encoders in the feedback loop. This enables the building of both simple and complex control feedback loops for industrial and automotive motor applications, targeting high-performance motion and position monitoring.

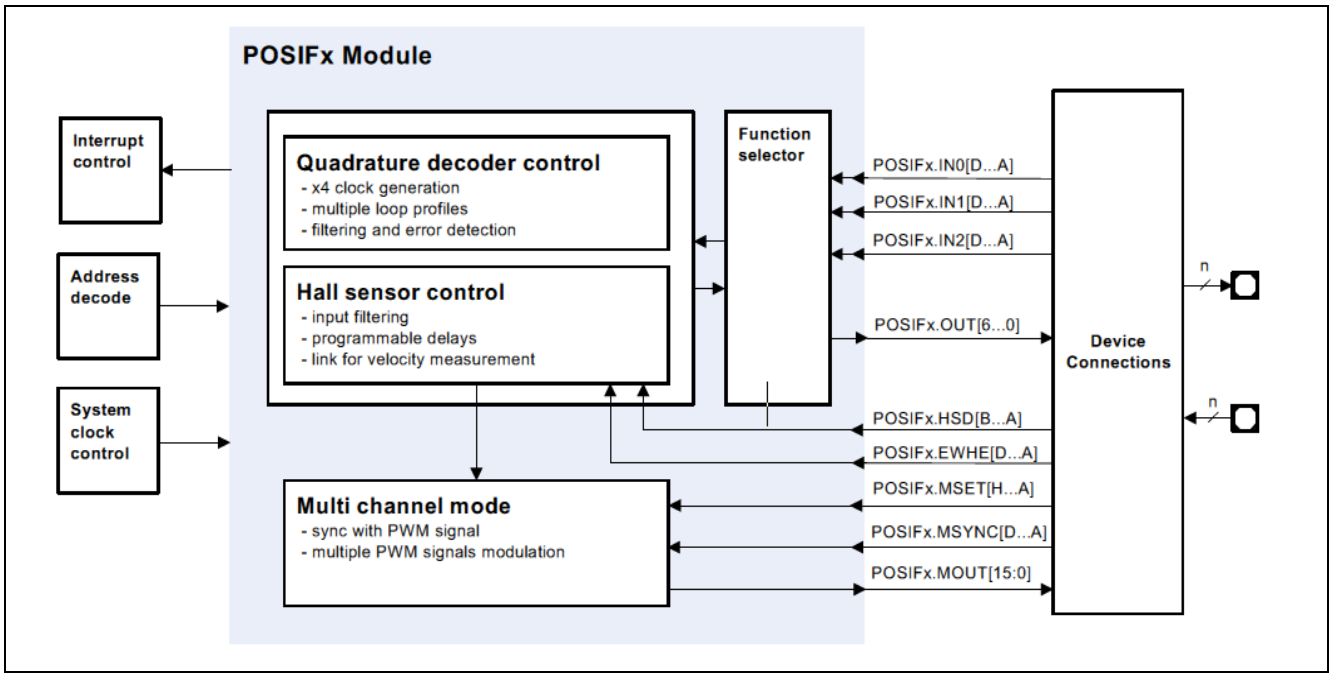


Figure 1 POSIF module block diagram

1.1 Function selector

The POSIF module has flexible configuration schemes. In order to cover the complete range of motion control applications, the Pin Connections have alternative functions which are activated dependent on the bitfield PCONF.FSEL setup:

- 00_B: Hall Sensor Mode
- 01_B: Quadrature Decoder Mode
- 10_B: Stand-alone Multi-Channel Mode
- 11_B: Stand-alone Multi-Channel & Quadrature Decoder Mode

1.2 Hall Sensor mode

The Hall Sensor Control Unit is used for direct control of brushless DC motors. Features include a simple built-in mode for brushless DC motor control, a shadow register for the multi-channel pattern, and complete synchronization with the PWM signals and the multi-channel pattern update.

The control unit provides an easy plug-in for Motor Control application using Hall Sensors:

- 2 or 3 Hall sensor topologies.
- Extended input filtering to avoid unwanted pattern switch due to noisy input signals.
- Synchronization with the PWM signals of the Capture/Compare Unit.
- Active freewheeling/synchronous rectification with dead time support (link with Capture/Compare Unit).
- Easy velocity measurement by using a Capture/Compare unit Timer Slice.

1.3 Quadrature Decoder mode

The Quadrature Decoder Unit is used for position control linked with a rotary incremental encoder. It has interfaces for position measurement, motor revolution, and velocity measurement. It provides an easy plug-in for rotary encoders:

- With or without index/top marker signal.
- Gear-slip or shaft winding compensation.
- Separate outputs for position, velocity and revolution control, matching different system requirements.
- Extended profile for position tracking, with revolution measurement and multiple position triggers for each revolution.
- Support for high dynamic speed changes due to tick-to-tick and tick-to-sync capturing method.

1.4 Multi-Channel mode

The Multi-Channel Mode unit is used in conjunction with the Hall Sensor mode to output the required motor control pattern, or alternatively it can be used stand-alone to perform a simple multi-channel control of several control units.

The mode provides modulation of multiple PWM signals:

- Parallel modulation controlled via software for N PWM signals , for systems with multiple power converters.
- Generating proprietary PWM modulations.
- Parallel and synchronous shut down of N PWM signals due to system feedback.

2 Triple-Hall Commutation Control for 3-Phase motor

When used in conjunction with the CAPCOM units CCU4 or CCU8, POSIF offers powerful solutions for motion control systems that use position sensors in the feedback loop. The POSIF Hall Sensors Decoder is one of its main sub-units, dedicated for 3-Phase motors.

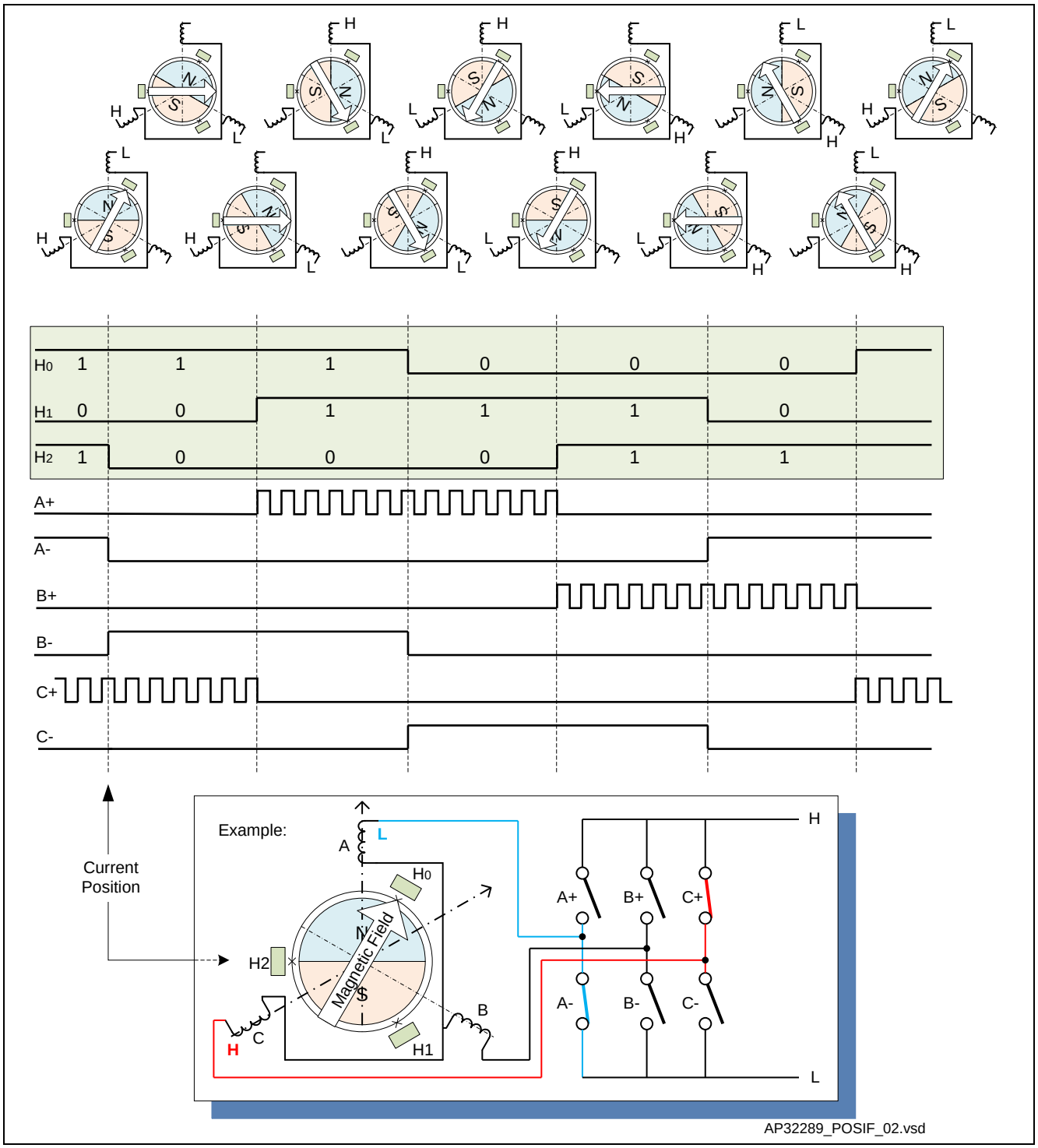


Figure 2 BLDC Motor Control with Triple Hall Commutation

2.1 Triple Hall Input Pattern

The Hall Sensor Decoder mode operates in four successive stages:

- The Hall Input Samples stage for the commutation signals.
- The Detection and Delay stage for input sampling after any edge transition.
- The Verification and Administration stage of expected Hall Patterns.
- The Output Pattern Control and hardware protocol.

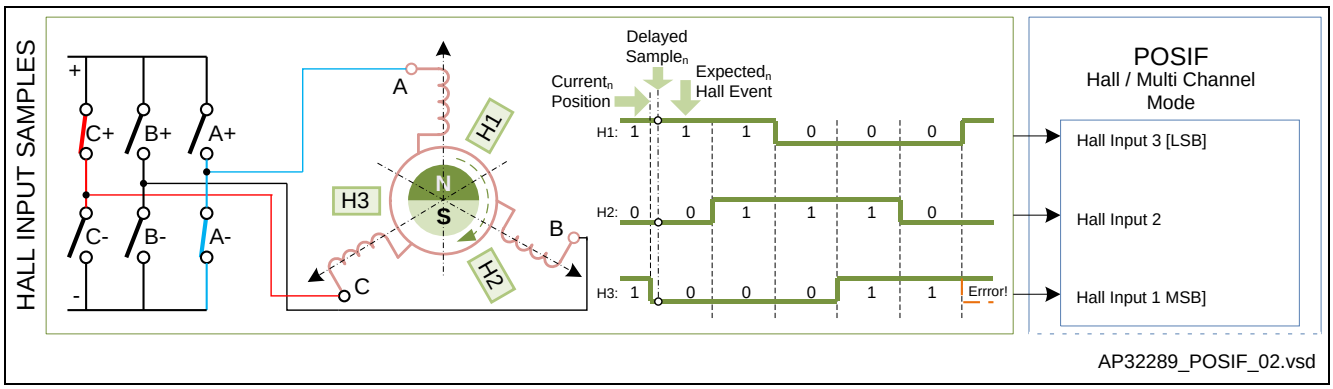


Figure 3 Triple-Hall Input Samples stage for a BLDC Motor Commutation Control

2.2 Delayed Hall Input Sampling

The sampling of each Hall Input pattern can be delayed a certain time after an input transition edge is detected, in order to reject noise that might appear at these positions.

The Input Edge Detection output POSIFx.OUT0 has to start a CCU4/8 timer that after a certain time triggers the sampling via a selected input: POSIFx.HSD[B..A].

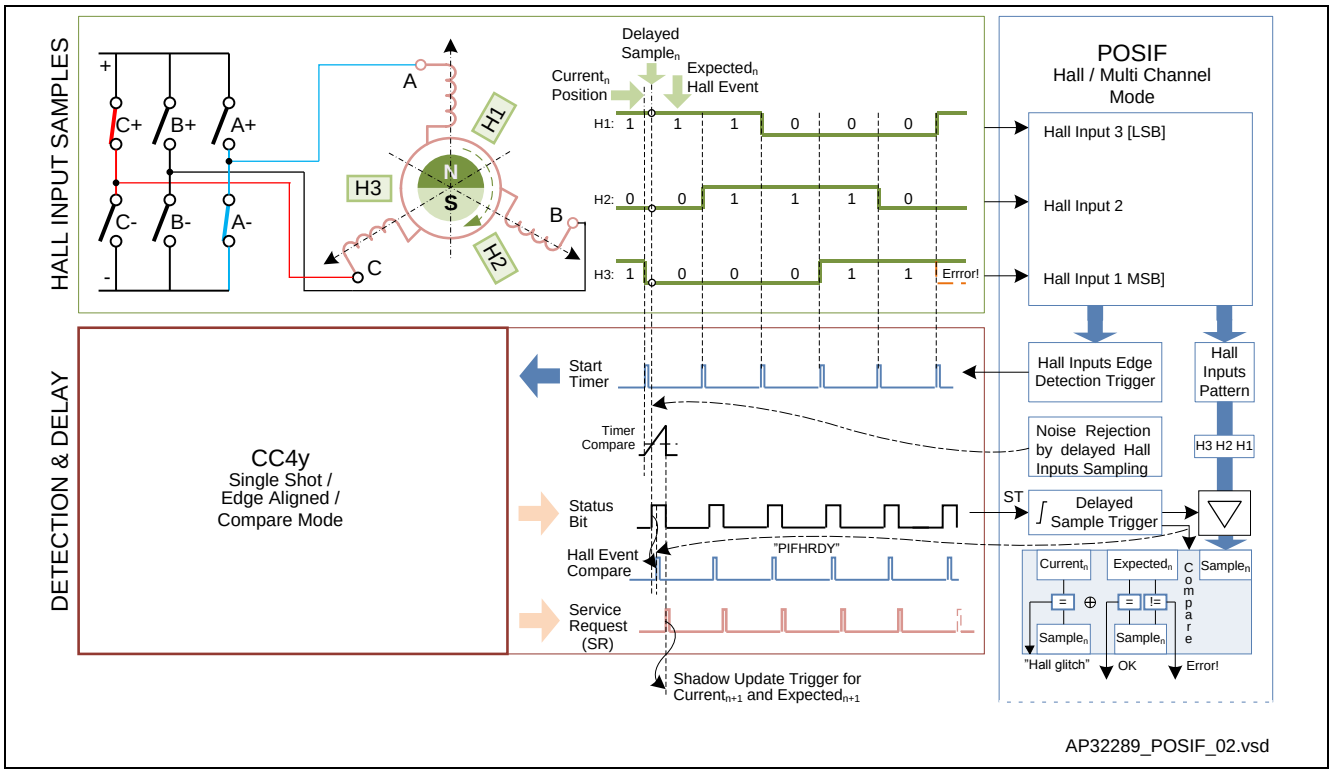


Figure 4 Triple-Hall Input Samples Edge Detection

2.2.1 Shadow Update of Expected Patterns

After each occurrence and sampling of a new pattern (Sample n) by the delayed Sample Trigger (ST) and an event verification by a Hall Event Compare, there should be a Service Request (SR) to Shadow Update the next (Expected$_{n+1}$) pattern. An update path could be a DMA transfer, on the falling edge of the Sample Trigger (ST).

2.3 Verification and Administration

2.3.1 Verification of Hall Event Input Patterns

Each Hall Input pattern ($Sample_n$) is compared to the current pattern ($Current_n$) and the expected pattern ($Expected_n$), by the PIFHRDY trigger.

The comparison results will be interpreted: Hall Glitch (==$Current_n$), Correct Hall Event (CHE) (==$Expected_n$) or Wrong Hall Event (WHE), if ($Sample_n$) is not equal to any of those.

The Hall Event Verification results in the following outputs:

- Correct Hall Event (CHE), linked to POSIFx.OUT1, triggers Shadow Transfer of $Current_{n+1}$ and $Expected_{n+1}$.
- Wrong Hall Event (WHE), linked to POSIFx.OUT2, can be used for IDLE control or STOP Run Bit/PWM.
- Glitch Hall Event or Wrong Hall Event, both events linked to POSIFx.OUT3, will generate a STOP signals.

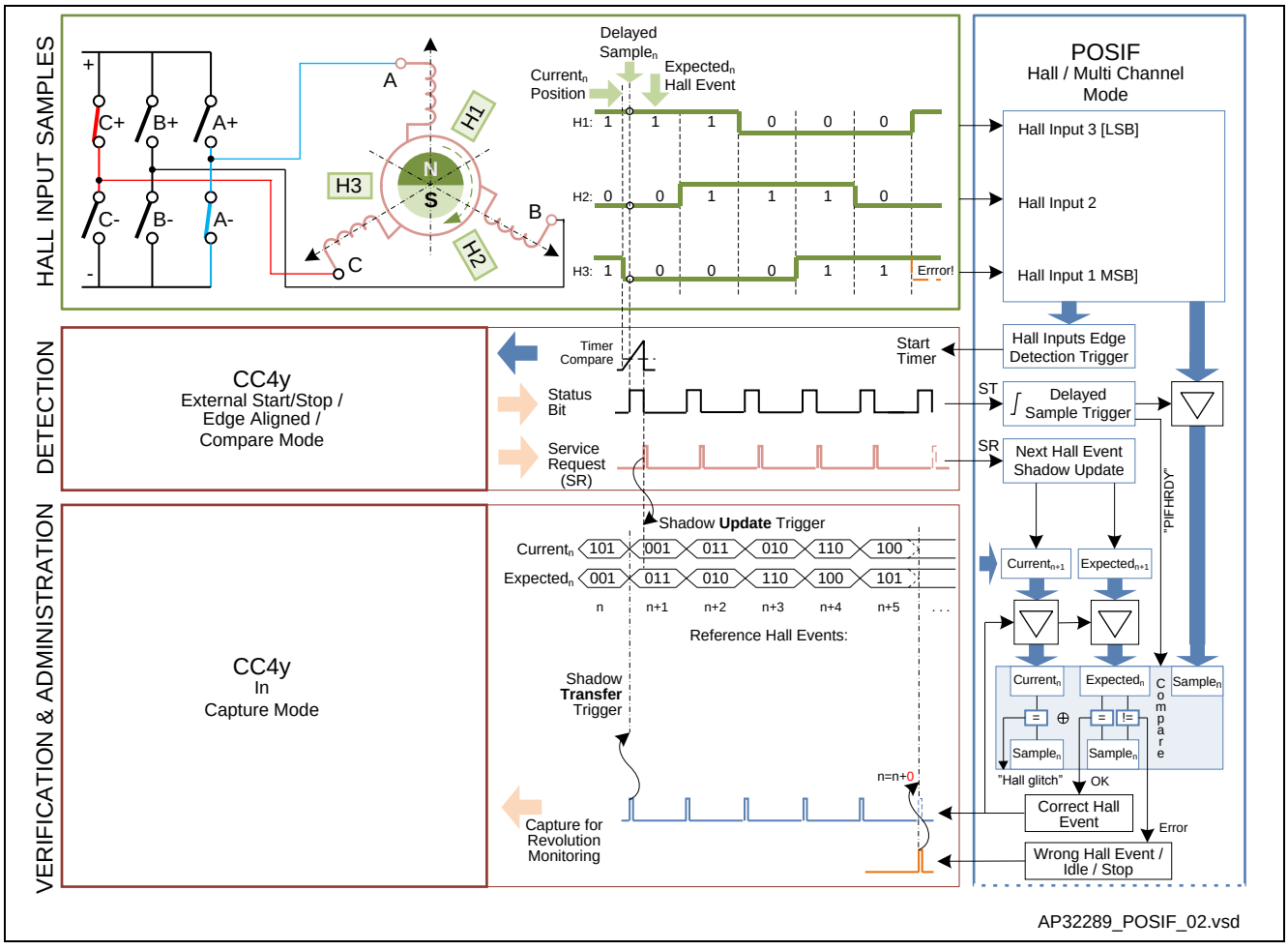


Figure 5 Triple-Hall Input Samples Verification and Administration of Expected Samples

2.3.2 Administration (Shadow Transfer) of Pattern Compare Values

On each successful Correct Hall Event (CHE) there is a Shadow Transfer of the next pattern compare values. The compare registers *Current*$_n$ Pattern (HALP.HCP) and *Expected*$_n$ Pattern (HALP.HEP) are reloaded from the *Current*$_{n+1}$ / *Expected*$_{n+1}$ shadow registers (HALP.HCPS / HALP.HEPS) that are shadow updated by software.

2.4 Output Pattern Control

The Multi-Channel-Mode Pattern register MCM.MCMP is linked to the output POSIFx.MOUT[15:0]. Its value will be updated from the shadow register MCSM.MCMPS by shadow transfer on the Multi-Channel update trigger PIFMST; i.e. the Multi-Channel Pattern Update Request POSIFx.OUT6 and Pattern Update Sync POSIFx.MSYNC[D…A].

2.4.1 Triple-Hall Output Pattern for 3-Phase Motor Control

The POSIF module in conjunction with the CAPCOM units in multi-channel mode (or in external modulation mode) enables complete synchronicity between the output state update and the application of a new pattern. This can be used in Hall Sensor Mode for the direct drive of brushless DC motors with the required motor control commutation patterns.

2.4.2 Output Pattern Control by CAPCOM Slices in Multi-Channel Mode

A CCU4/8OUT-pin output level of any slice that has been set in multi-channel mode (CC4/8yTC**.**MCME = 1_B), can be controlled in parallel by an external level that is present on its Multi-Channel Input MCI-pin. For synchronization, a Set Shadow Transfer Enable MCSS and a Pattern Set PSy request complete the hardware protocol.

Triple-Hall Commutation Control for 3-Phase motor

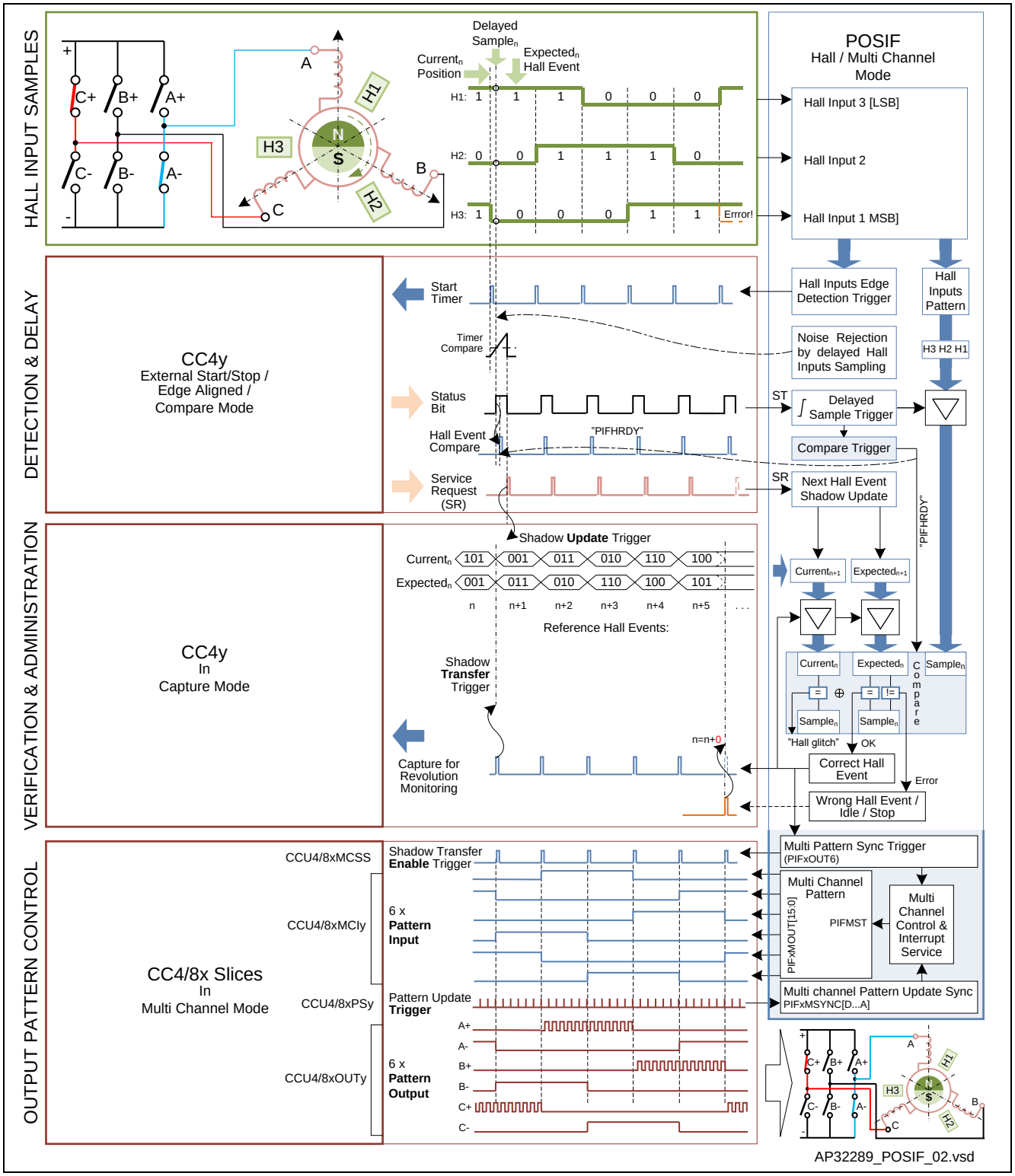


Figure 6 Triple-Hall BLDC Motor Commutation Control using CAPCOM units in Multi-Channel mode

2.5 Example Use Case: Using POSIF in Hall Sensor Mode to measure the speed of the motor

This example uses the POSIF in Hall Sensor Mode to measure the speed of the connected 3-Phase motor.

The 3-Phase motor is driven using a selected motor control algorithm (in this example, a Space Vector Modulation algorithm using a DAVE™ PWM_SVM APP). The hall inputs (Hall1/P0.13, Hall2/P1.1, and Hall3/P1.0) are connected to the POSIF module.

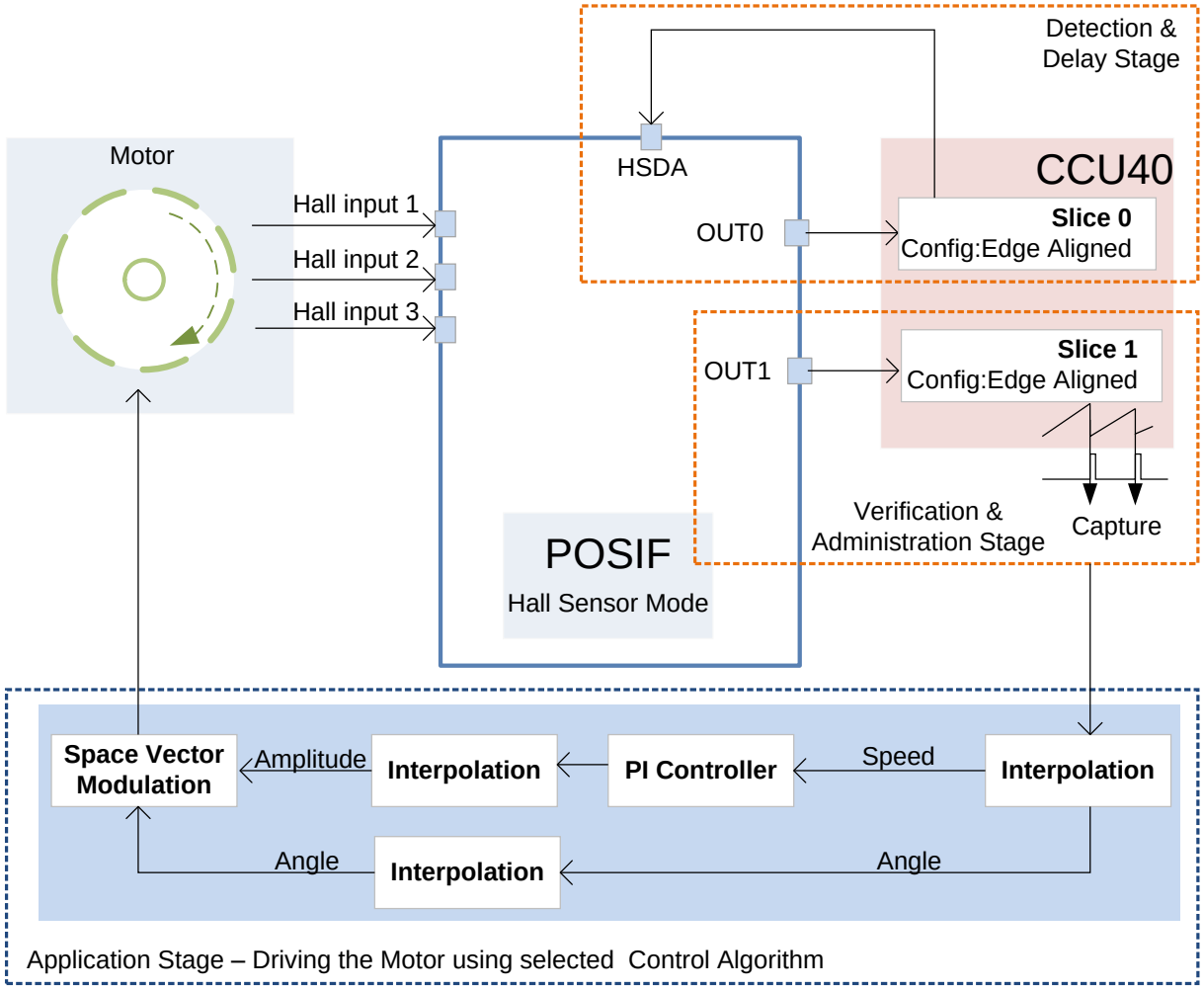
POSIF.OUT0 provides a detection and delay stage for the inputs sampling after any edge detection. This signal is connected to a CCU40 slice to start a (1us) timer, configured in single-shot mode. The status signal is connected to POSIF.HSDA to delay the input sampling to reject noise that might appear at those positions.

POSIF.OUT1 generates an output on a correct Hall Event that is used to trigger a shadow transfer of the $Current_{N+1}$ and $Expected_{N+1}$ pattern. This signal is also connected to a CCU40 slice to capture the speed of the motor (i.e. the timing between 2 correct Hall events). Each time a correct Hall event is detected, an interrupt routine is entered where the next set of current and expected Hall pattern are updated, and the captured time between the Hall events are read from the CCU4 capture slice.

The captured time is the speed of the motor and can be used by the application for feedback into the system. This input can be used in the selected motor control algorithm for controlling the motor.

Note:

1. *The details of the implementation for the application stage are not covered in this example.*
2. *This example is based on the XMC1300.*



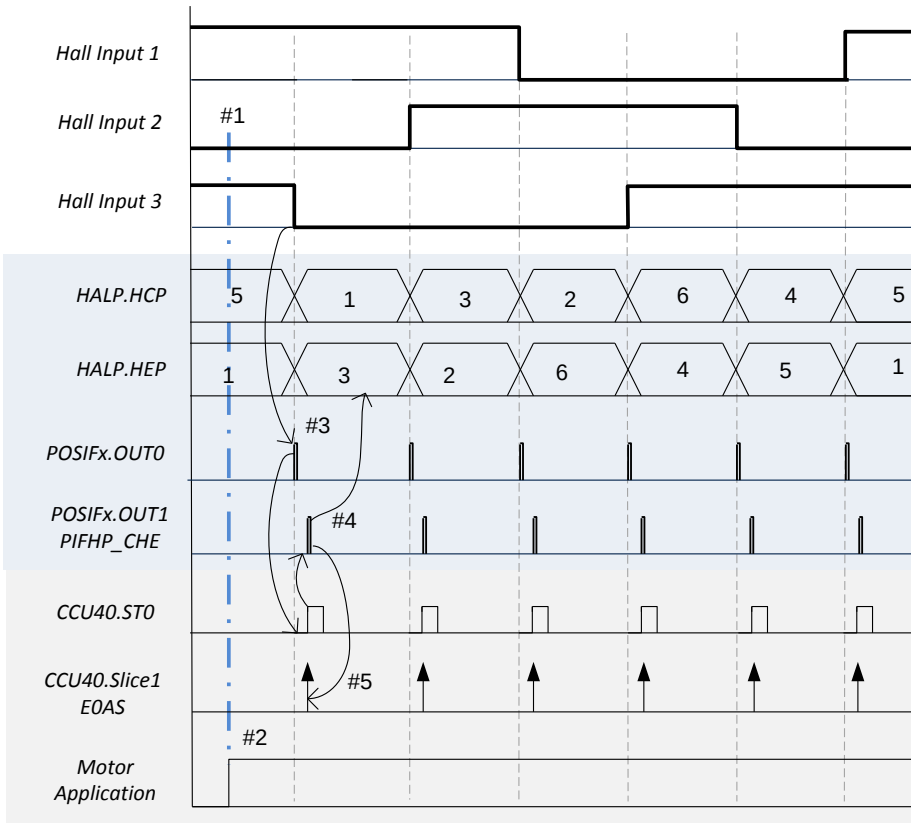
In a motor control application, the POSIF module can be used to determine the speed of the motor by detecting a correct Hall event generated from the motor. Once the current speed is determined, it can be adjusted based on an appropriate motor control algorithm.

Detection & Delay Stage
POSIF.OUT0 provides a detection and delay stage for the inputs sampling after any edge detection. This signal is connected to a CCU40 slice to start a (1us) timer, configured in single-shot mode. The status signal is connected to POSIF.HSDA to delay the input sampling to reject noise that might appear at those positions.

Verification & Administration Stage
Each Hall input pattern is compared with the current pattern and the expected pattern. A correct Hall Event generates an output linked to POSIF.OUT1 and it triggers a shadow transfer of the $Current_{N+1}$ and $Expected_{N+1}$ pattern. In addition, this signal is connected to a CCU40 slice to capture the speed of the motor (i.e. the timing between 2 correct Hall events).

Application Stage
A space vector modulation algorithm is selected for driving the 3-Phase motor. The capture timing can be used by the selected motor control algorithm, as a feedback to the system, to update the system at run-time. This stage is not covered in this document.

Figure 7 Example: Setup for POSIF in Hall Sensor mode to measure motor speed



#1: Read the initial motor position. Once the position is determined, set up the current and expected hall patterns into the POSIF module.

#2: Once the Hall pattern is setup, start the motor application. (If the motor application is not directly controlled via the MCU, a signal could be given to the external motor application to synchronize the starting.

#3: POSIFx.OUT0 is generated on detecting an input transition edge from the Hall input. To reject noise at these points, OUT0 is connected to the CCU40 slice to start a timer. The ST signal of this slice is connected to the HSDA to delay the POSIF sampling.

#4: A correct Hall event is detected and generates a shadow update of the next expected pattern. A CHE event is triggered to update the Hall pattern.

#5: The OUT1 signal is used to trigger a capture event on CCU40 slice 1. This captures the time between 2 correct Hall event (i.e. speed).

Figure 8 Example: Setup for POSIF in Hall Sensor mode to measure motor speed

2.5.1 Theory of Operation

The setup of the POSIF module makes the assumption that the Hall pattern of the motor is available. The motor application is not started before the initial position of the motor is set up. Once the starting Hall pattern is determined, the motor application can be started. Thereafter, the control of the motor is based on the selected motor control algorithm.

In our example we used a Space Vector Modulation algorithm. The captured speed of the motor is interpolated into speed and angle inputs, to be used in a PI controller. The output is then interpolated before using space vector modulation to drive the motor.

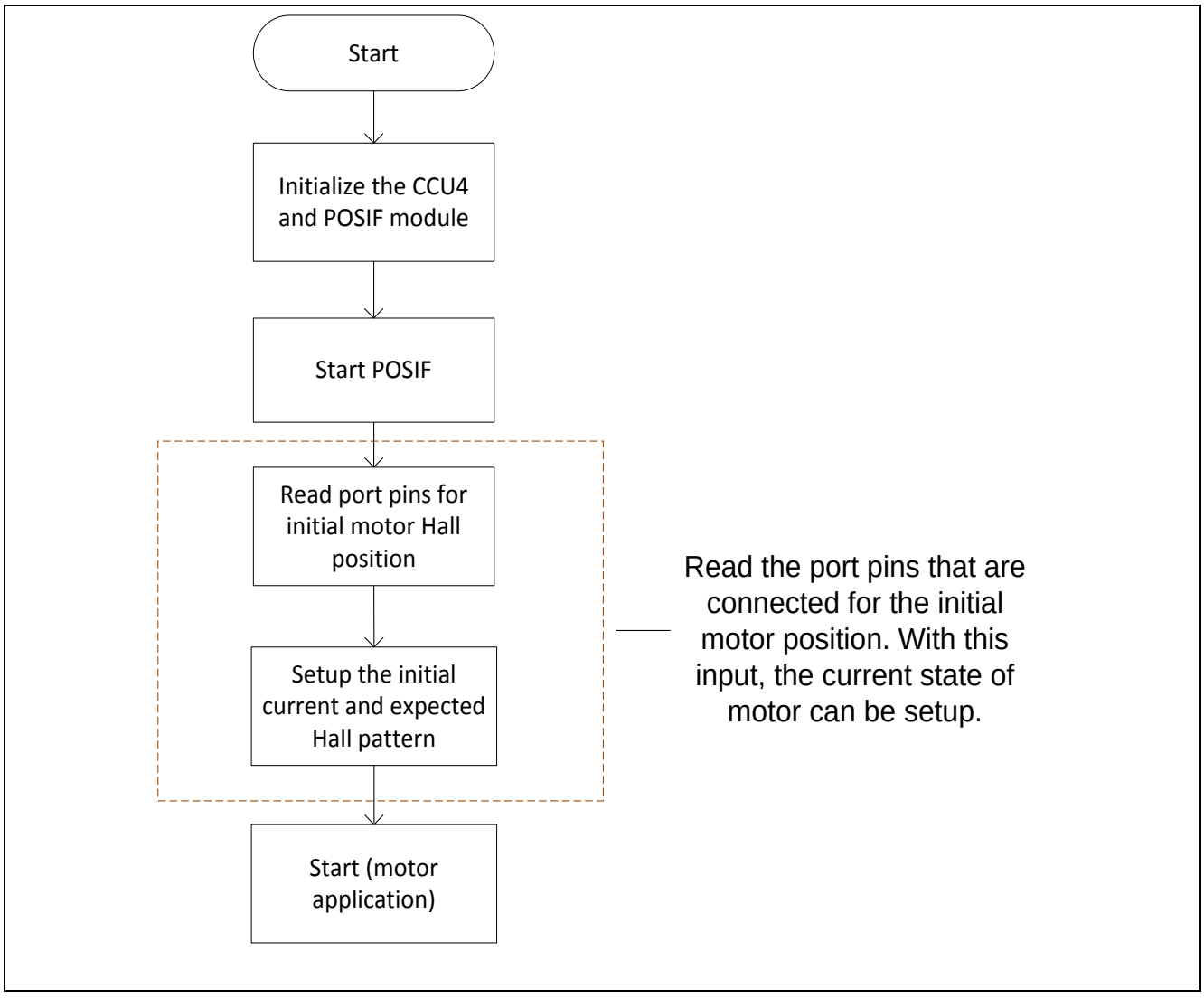


Figure 9 Program Flow: Initialization

2.5.2 Macro and variable settings

XMC Lib Project includes

```
#include <xmc_ccu4.h>
#include <xmc_posif.h>
#include <xmc_gpio.h>
```

Project Macro definitions

```
/* CCU4 Macros */
#define MODULE_PTR              CCU40
#define MODULE_NUMBER           (0U)

#define DELAY_SLICE_PTR         CCU40_CC40
#define DELAY_SLICE_NUMBER      (0U)

#define CAPTURE_SLICE_PTR       CCU40_CC41
#define CAPTURE_SLICE_NUMBER    (1U)

/* POSIF Macros */
#define POSIF_PTR POSIF0

/* Application Related Macros */
#define HALL_POSIF_MCM(EP,CP)   (((uint32_t)EP<< 3)|(uint32_t)CP)
#define ANGLE_ONE_DEGREE        (46603U)
#define ANGLE_SEVEN_DEGREE      (ANGLE_ONE_DEGREE * 7.2)
```

Project Variables Definition

```
/* Hall pattern of the motor. This depends on the type and make of the motor selected */
uint8_t hall_pattern[] =
{
        (uint8_t)HALL_POSIF_MCM(0,0),(uint8_t)HALL_POSIF_MCM(3,1),
        (uint8_t)HALL_POSIF_MCM(6,2),(uint8_t)HALL_POSIF_MCM(2,3),
        (uint8_t)HALL_POSIF_MCM(5,4),(uint8_t)HALL_POSIF_MCM(1,5),
        (uint8_t)HALL_POSIF_MCM(4,6),(uint8_t)HALL_POSIF_MCM(0,0),

        (uint8_t)HALL_POSIF_MCM(0,0),(uint8_t)HALL_POSIF_MCM(5,1),
        (uint8_t)HALL_POSIF_MCM(3,2),(uint8_t)HALL_POSIF_MCM(1,3),
        (uint8_t)HALL_POSIF_MCM(6,4),(uint8_t)HALL_POSIF_MCM(4,5),
        (uint8_t)HALL_POSIF_MCM(2,6),(uint8_t)HALL_POSIF_MCM(0,0)
};

uint32_t hall[3] = {0,0,0};
uint32_t initialhallposition = 0;
uint32_t hallposition = 0;
uint32_t motorspeed_onerev =0;
```

2.5.3 XMC Lib Peripheral Configuration structure

XMC Compare Unit 4 (CCU4) Configuration

```
//XMC Capture/Compare Unit 4 (CCU4) Configuration for Capture:
XMC_CCU4_SLICE_COMPARE_CONFIG_t delay_config =
{
  .timer_mode = (uint32_t) XMC_CCU4_SLICE_TIMER_COUNT_MODE_EA,
  .monoshot = (uint32_t) true,
  .shadow_xfer_clear = (uint32_t) 0,
  .dither_timer_period = (uint32_t) 0,
  .dither_duty_cycle = (uint32_t) 0,
  .prescaler_mode = (uint32_t) XMC_CCU4_SLICE_PRESCALER_MODE_NORMAL,
  .mcm_enable = (uint32_t) 0,
  .prescaler_initval = (uint32_t) 0,    /* in this case, prescaler = 2^10  */
  .float_limit = (uint32_t) 0,
  .dither_limit = (uint32_t) 0,
  .passive_level = (uint32_t) XMC_CCU4_SLICE_OUTPUT_PASSIVE_LEVEL_LOW,
  .timer_concatenation = (uint32_t) 0
};

/* Capture Slice configuration */
XMC_CCU4_SLICE_CAPTURE_CONFIG_t capture_config =
{
   .fifo_enable = false,
   .timer_clear_mode = XMC_CCU4_SLICE_TIMER_CLEAR_MODE_ALWAYS,
   .same_event = false,
   .ignore_full_flag = true,
   .prescaler_mode = XMC_CCU4_SLICE_PRESCALER_MODE_NORMAL,
   .prescaler_initval = (uint32_t) 5, /* in this case, prescaler = 2^5  */
   .float_limit = (uint32_t) 0,
   .timer_concatenation = (uint32_t) 0
};

XMC_CCU4_SLICE_EVENT_CONFIG_t start_event0_config =    //off time capture
{
  .mapped_input = XMC_CCU4_SLICE_INPUT_E,              //CAPTURE on POSIF0.OUT0
  .edge         = XMC_CCU4_SLICE_EVENT_EDGE_SENSITIVITY_RISING_EDGE,
  .level        = XMC_CCU4_SLICE_EVENT_LEVEL_SENSITIVITY_ACTIVE_HIGH,
  .duration     = XMC_CCU4_SLICE_EVENT_FILTER_DISABLED
};

XMC_CCU4_SLICE_EVENT_CONFIG_t capture_event0_config =  //off time capture
{
  .mapped_input = XMC_CCU4_SLICE_INPUT_F,              //CAPTURE on POSIF0.OUT1
  .edge         = XMC_CCU4_SLICE_EVENT_EDGE_SENSITIVITY_RISING_EDGE,
  .level        = XMC_CCU4_SLICE_EVENT_LEVEL_SENSITIVITY_ACTIVE_HIGH,
  .duration     = XMC_CCU4_SLICE_EVENT_FILTER_DISABLED
};
```

XMC Position Interface Unit (POSIF) Configuration

```
/* Configuration for POSIF - Hall Sensor Mode */
```

```
XMC_POSIF_CONFIG_t POSIF_HALL_config =
{
  .mode   = XMC_POSIF_MODE_HALL_SENSOR,    /**< POSIF Operational mode */
  .input0 = XMC_POSIF_INPUT_PORT_B,    /**< Choice of input for Input-1 */
  .input1 = XMC_POSIF_INPUT_PORT_A,    /**< Choice of input for Input-2 */
  .input2 = XMC_POSIF_INPUT_PORT_A,   /**< Choice of input for Input-3 */
  .filter = XMC_POSIF_FILTER_DISABLED   /**< Input filter configuration */
};

XMC_POSIF_HSC_CONFIG_t POSIF_HSC_config =
{
  .disable_idle_signal= 1,
  .sampling_trigger = 0,        //HSDA
  .sampling_trigger_edge = 0    //Rising edge

};
```

XMC GPIO Configuration

```
XMC_GPIO_CONFIG_t HALL_POSIF_0_Hall_PadConfig =
{
  .mode            = (XMC_GPIO_MODE_t)XMC_GPIO_MODE_INPUT_TRISTATE,
  .output_level    = (XMC_GPIO_OUTPUT_LEVEL_t)XMC_GPIO_OUTPUT_LEVEL_LOW,
};
```

2.5.4 Interrupt Service Routine Function implementation

The Correct Hall Event (CHE) interrupt handler function updates the new Hall pattern and reads the captured speed of the motor from the CCU40 Slice 1.

Note: The processing of the captured speed is not discussed here, as this depends on the implementation of the selected motor control algorithm. For a typical closed loop system, interpolation stages to convert the captured data in to the required data format of the motor control algorithm are required.

```
/* Interrupt handler - to setup the next hall pattern and read the captured value */

void POSIF0_0_IRQHandler(void)
{
   uint32_t capturedvalue0;
   uint32_t capturedvalue1;
   uint32_t motorspeed;

   /* Read the captured registered */
   capturedvalue0 = XMC_CCU4_SLICE_GetCaptureRegisterValue(CAPTURE_SLICE_PTR,0U);
   capturedvalue1 = XMC_CCU4_SLICE_GetCaptureRegisterValue(CAPTURE_SLICE_PTR,1U);

   /* Check if a new value is captured, store value to Speed variable */
   if ( capturedvalue0 & CCU4_CC4_CV_FFL_Msk )
   {
        motorspeed = capturedvalue0 & CCU4_CC4_CV_CAPTV_Msk;
   }
   else
```

```
  {
       motorspeed = capturedvalue1 & CCU4_CC4_CV_CAPTV_Msk;
  }

  /* Clear pending event  */
  XMC_POSIF_ClearEvent(POSIF_PTR,XMC_POSIF_IRQ_EVENT_CHE);

  /* Set the new Hall pattern */
  hallposition = XMC_POSIF_HSC_GetExpectedPattern(POSIF_PTR);
  XMC_POSIF_HSC_SetHallPatterns(POSIF_PTR,hall_pattern[hallposition]);

  /* Add up the captured timing for each hall event detected */
  motorspeed_onerev+=motorspeed;

  /* Process the captured speed for the motor update when if one complete motor */
  /* revolution is detected.*/
  if (hallposition==initialhallposition)
  {
       /* Application code for speed PI to determine the new angle for the */
       /* motor update */
       /* ... ... ... */

       /* Reset the motorspeed variable for next revolution capture */
       motorspeed_onerev = 0;
  }
}
```

2.5.5 Main Function implementation

Before the first start and execution of timer slice software, the CCU4 must have been initialized using the following sequence:

- Initialize the HALL GPIO

```
XMC_GPIO_Init(P0_13, &HALL_POSIF_0_Hall_PadConfig);
XMC_GPIO_Init(P1_0, &HALL_POSIF_0_Hall_PadConfig);
XMC_GPIO_Init(P1_1, &HALL_POSIF_0_Hall_PadConfig);
```

- Setup the CCU40 slices
  - Slice 0 for single shot
  - Slice 1 for capture time

```
/* Enable clock, enable prescaler block and configure global control */
XMC_CCU4_Init(MODULE_PTR, XMC_CCU4_SLICE_MCMS_ACTION_TRANSFER_PR_CR);

/* Start the prescaler and restore clocks to slices */
XMC_CCU4_StartPrescaler(MODULE_PTR);

/* Ensure fCCU reaches CCU40, CCU80 */
XMC_CCU4_SetModuleClock(MODULE_PTR, XMC_CCU4_CLOCK_SCU);

   /* Configure CCU4 slices as monoshot and capture slice */
XMC_CCU4_SLICE_CompareInit(DELAY_SLICE_PTR, &delay_config);
```

```
XMC_CCU4_SLICE_CaptureInit(CAPTURE_SLICE_PTR, &capture_config);

/* Configure CCU4 delay as 1us - CCU40.ST0 is connected to POSIF.HSDA*/
XMC_CCU4_SLICE_SetTimerPeriodMatch(DELAY_SLICE_PTR, 127U);
XMC_CCU4_SLICE_SetTimerCompareMatch(DELAY_SLICE_PTR, 64U);
XMC_CCU4_SLICE_SetTimerPeriodMatch(CAPTURE_SLICE_PTR, 65535U);

/* Transfer value from shadow timer registers to actual timer registers */
XMC_CCU4_EnableShadowTransfer(MODULE_PTR, \
(uint32_t) (XMC_CCU4_SHADOW_TRANSFER_SLICE_0 | XMC_CCU4_SHADOW_TRANSFER_SLICE_1));

/* Configure and enable events */
XMC_CCU4_SLICE_StartConfig(DELAY_SLICE_PTR, \
XMC_CCU4_SLICE_EVENT_0, XMC_CCU4_SLICE_START_MODE_TIMER_START_CLEAR);
XMC_CCU4_SLICE_Capture0Config(CAPTURE_SLICE_PTR, XMC_CCU4_SLICE_EVENT_0);

XMC_CCU4_SLICE_ConfigureEvent(CAPTURE_SLICE_PTR, \
XMC_CCU4_SLICE_EVENT_0, &capture_event0_config);
XMC_CCU4_SLICE_ConfigureEvent(DELAY_SLICE_PTR, \
     XMC_CCU4_SLICE_EVENT_0, &start_event0_config);

/* Get the slice out of idle mode */
XMC_CCU4_EnableClock(MODULE_PTR, DELAY_SLICE_NUMBER);
XMC_CCU4_EnableClock(MODULE_PTR, CAPTURE_SLICE_NUMBER);
```

- Setup the POSIF in Hall Sensor Mode

```
/* POSIF Configuration */
XMC_POSIF_Init(POSIF_PTR, &POSIF_HALL_config);

XMC_POSIF_HSC_Init(POSIF_PTR, &POSIF_HSC_config);

XMC_POSIF_EnableEvent(POSIF_PTR, XMC_POSIF_IRQ_EVENT_CHE);

/* Connect correct hall event to SR0 */
XMC_POSIF_SetInterruptNode(POSIF_PTR, XMC_POSIF_IRQ_EVENT_CHE, XMC_POSIF_SR_ID_0);

/* Configure NVIC */
/* Set priority */
NVIC_SetPriority(POSIF0_0_IRQn, 0U);

/* Enable IRQ */
NVIC_EnableIRQ(POSIF0_0_IRQn);
```

- Start the CCU4 slice for the capture timing

```
/* Start Timer Running */
XMC_CCU4_SLICE_StartTimer(CAPTURE_SLICE_PTR);
```

- Start the POSIF module

```
/* Start the POSIF module*/
XMC_POSIF_Start(POSIF_PTR);
```

- Set the initial motor position

```
/* read the IO on P0.13, 1.0. 1.1 */
hall[0]=XMC_GPIO_GetInput(P0_13);
hall[1]=XMC_GPIO_GetInput(P1_0);
hall[2]=XMC_GPIO_GetInput(P1_1);
hallposition = (uint32_t)((hall[0] | (hall[1] << 1) | (hall[2] << 2)));
initialhallposition = hallposition;

/* use the hall pattern */
XMC_POSIF_HSC_SetHallPatterns(POSIF_PTR,hall_pattern[hallposition]);
XMC_POSIF_HSC_UpdateHallPattern(POSIF_PTR);

hallposition = XMC_POSIF_HSC_GetExpectedPattern(POSIF_PTR);
XMC_POSIF_HSC_SetHallPatterns(POSIF_PTR,hall_pattern[hallposition]);
```

Once the initial motor position is determined, the motor application can be started. If a DAVE APP is used, the appropriate API can be used to start the motor application. Otherwise, depending on the application, a hand-shaking signal (such as a GPIO or UART command) can be issued to the motor to initiate the starting sequence.

3 Quadrature Decoder

3.1 Principle of Operation

Inside the Quadrature Decoder Mode, there are two different sub-sets available:

- Standard Quadrature Mode.
  - The standard mode is used when the external rotary encoder provides two phase signals and an index/marker signal that is generated once per shaft revolution.
- Direction Count Mode.
  - The Direction Count Mode is used when the external encoder only provides a clock and a direction signal.

3.1.1 Quadrature Clock Generation Basics

Each Quadrature Clock Cycle is a 4-state, 4 clocks, pulse train that is generated by two phase-shifted signals from a sensor-pair, (A) and (B). Since the state order determines which sensor is in the sense position of the leading phase, each state transition is unique for the motion direction: clockwise or counterclockwise.

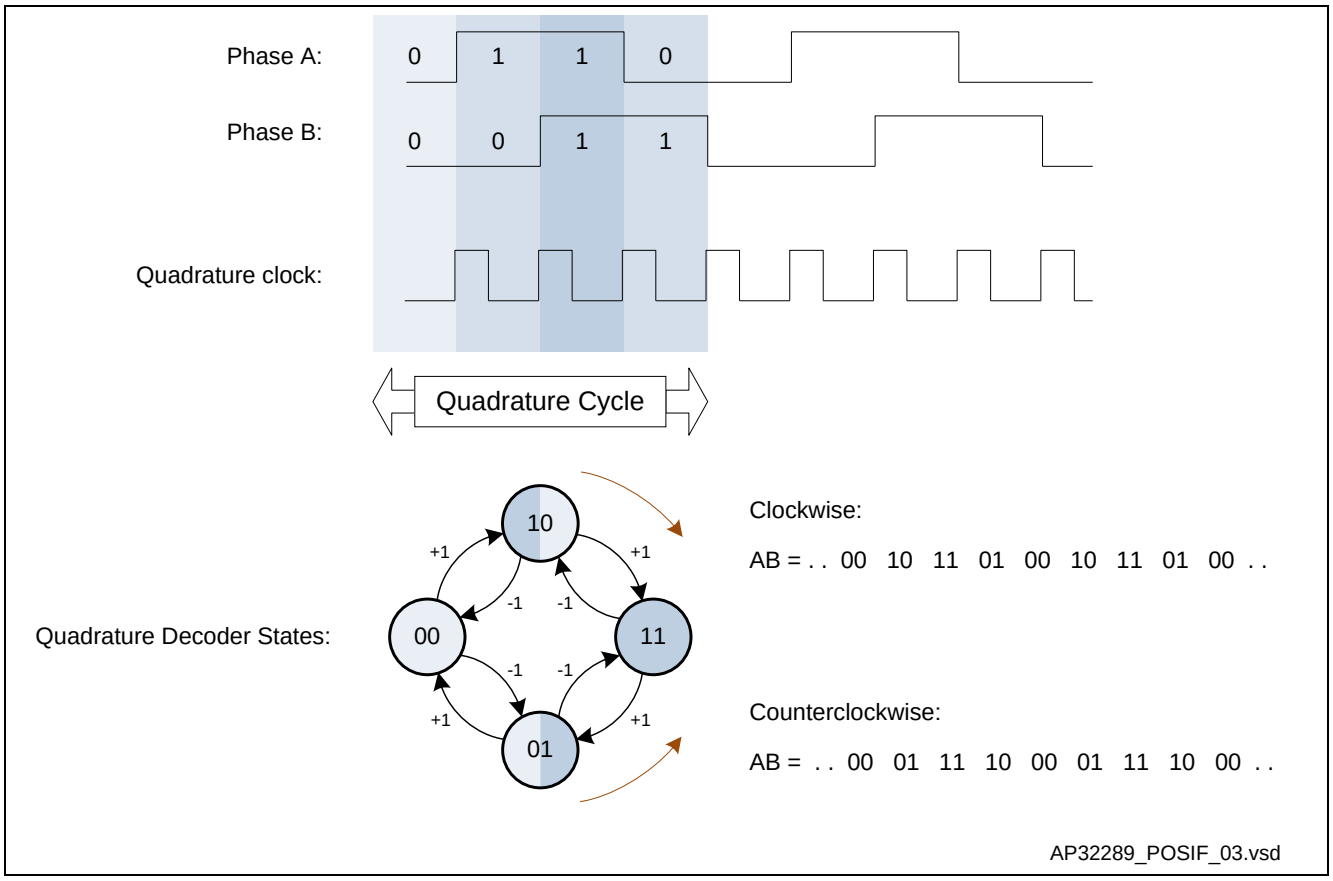


Figure 10 Standard Quadrate Clock Decoding

3.1.2 Standard Quadrature Mode

Position encoders that work in Standard Quadrature Mode provide two phase signals (Phase A, Phase B), and encoders with or without an index/top marker signal.

- The inputs should be mapped to:
  - POSIFx.IN0[D...A]
  - POSIFx.IN1[D...A]
  - POSIFx.IN2[D...A]
- The output pins are POSIFx.OUT0 to POSIFx.OUT5 for Quadrature Clock, Direction, Period Clock, Index Occurrence, Index (revolution) and Synchronous Start.
  - The Quadrature clock gives position up/down count information in conjunction with the Direction signal.
  - The Period clock contains sequences of (AB) or (BA) pulse-pairs for velocity detection.
  - The Index Occurrence pin asserts Z-mark/Index (on QDC.ICM conditions), whilst the Index revolution asserts only once per revolution.
  - Synchronous Start output, POSIFx.OUT5, is linked with the module run bit can be used together with the CAPCOM units for a complete synchronous start, if those CAPCOM units have been preset for External Events Control (External Start and Extended Start for Start for example, or Flush/Start).

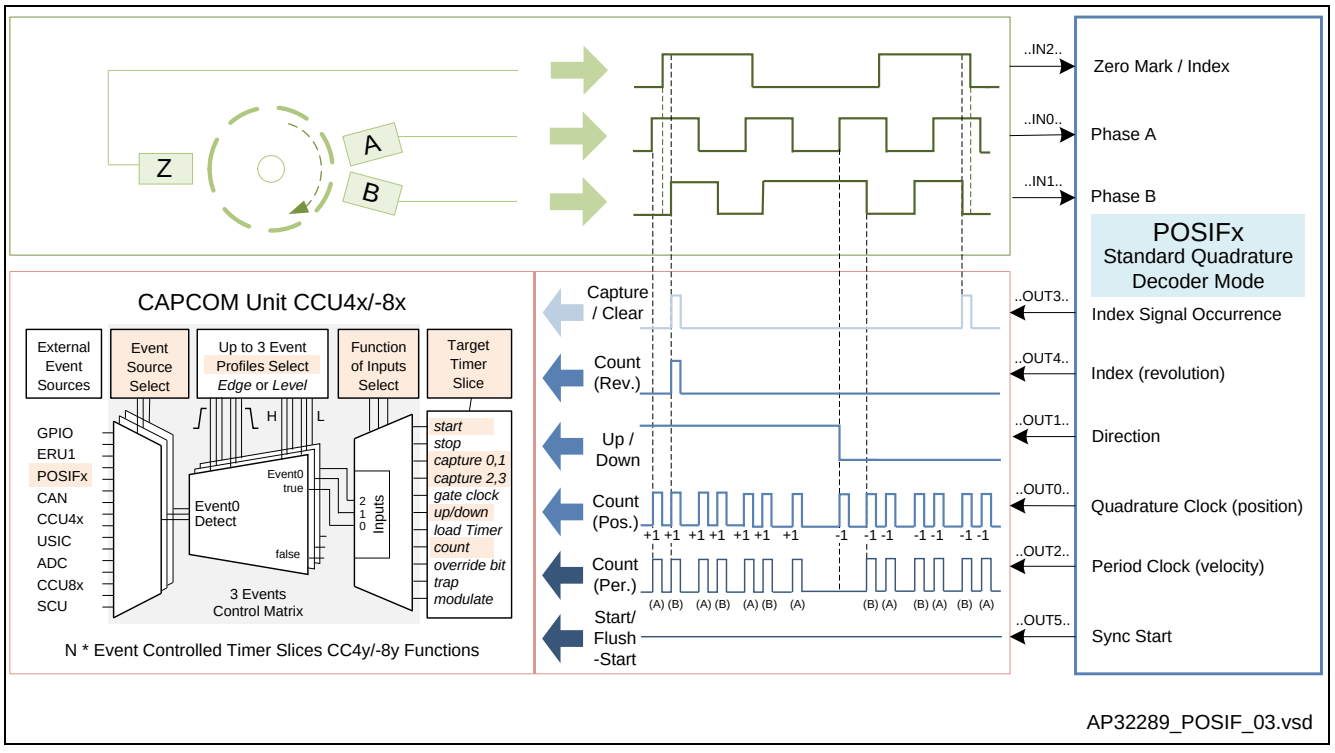


Figure 11 POSIF in Standard Quadrature mode

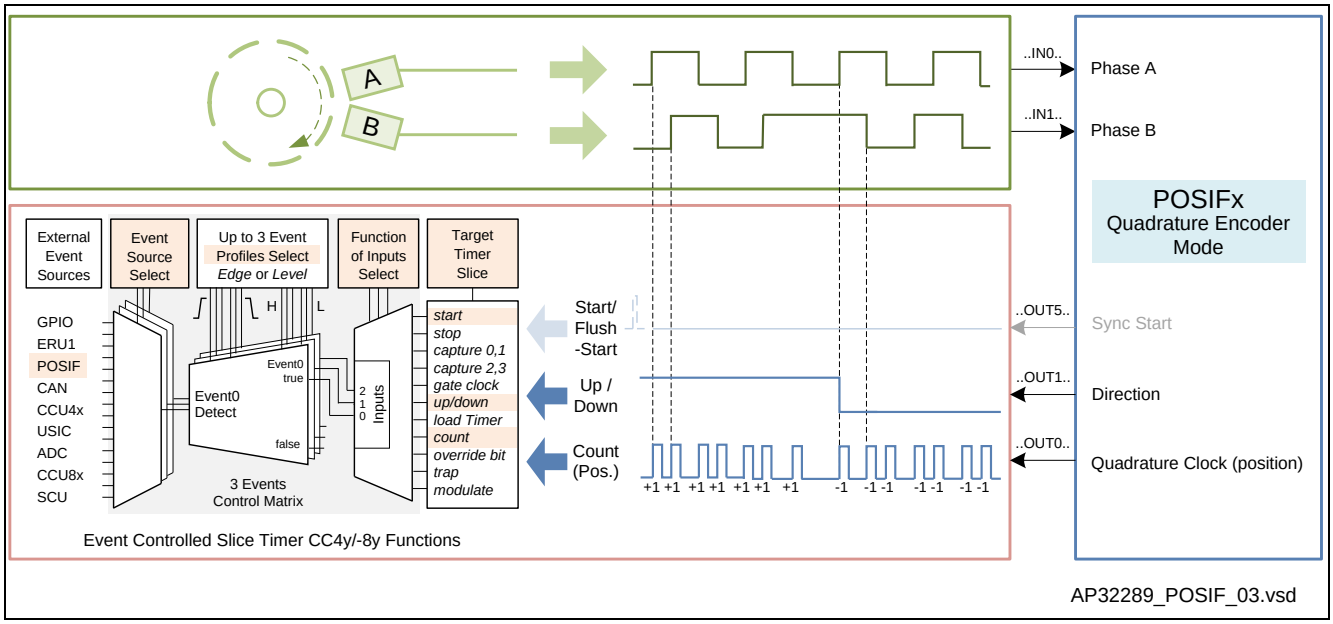


Figure 12 POSIF in Standard Quadrature mode (when using position monitoring)

3.1.3 Direction Count mode

Position encoders that work in Direction Count mode provide just two signals; Clock and Direction information.

- The POSIFx gets the Direction Count mode by setting PCONF.QDCM =1_B.
- The inputs should be mapped to POSIFx.IN0[D...A]/ POSIFx.IN1[D...A]).
- The output pins are POSIFx.OUT0/-1/-5 for Clock/Direction/Sync.-Start.

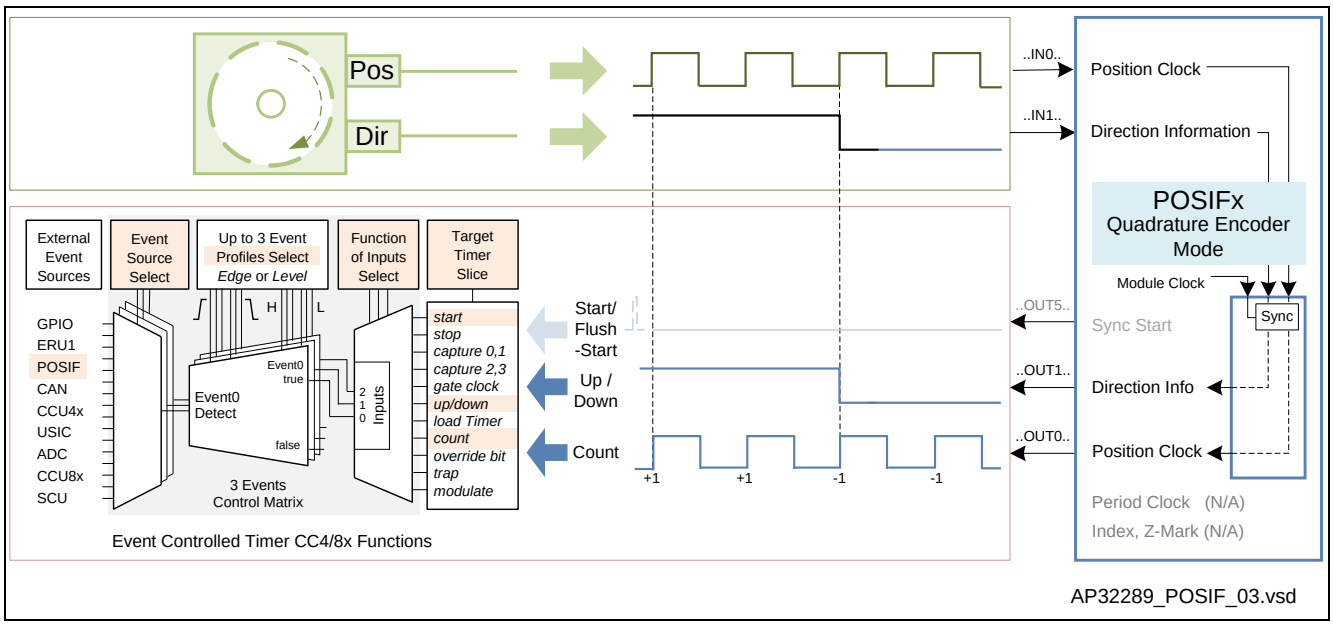


Figure 13 POSIF in Direction Count mode

3.2 Motion Monitoring profiles

The Quadrature Decoder is dedicated for motion profile monitoring. Assume 2 different connection profiles with a POSIFx unit and 4 Timer Slices CC40/-41/-42/-43 of a CAPCOM4. Several motion monitoring functions can be performed in parallel in each profile to cover a variety of challenging scenarios, including:

- Motion Tracking plus Velocity Monitoring based on Time within N Ticks.
- Motion Tracking plus Velocity Monitoring based on Ticks within Time T.

3.2.1 Motion Tracking plus Velocity Monitoring based on Time within N Ticks

This is Motion Monitoring “Profile 1”. “N Ticks” should be understood as a certain number of Period clocks.

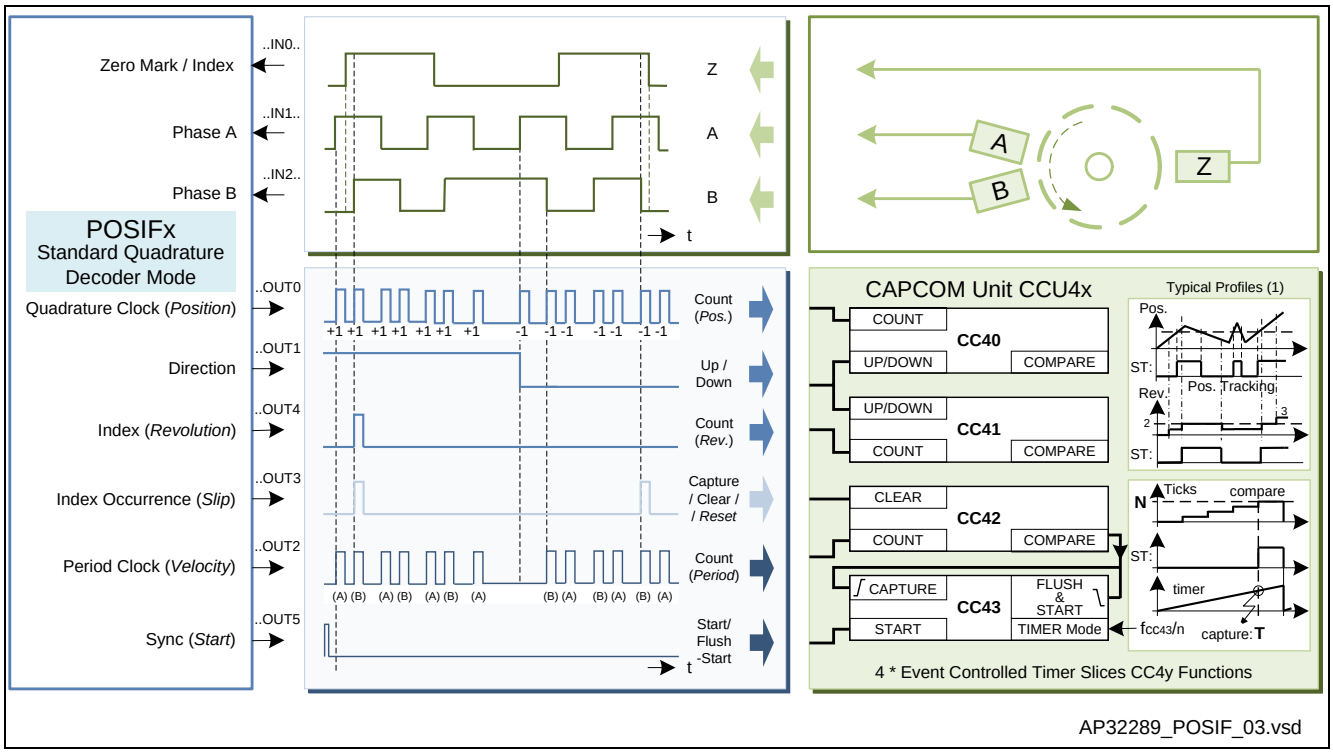


Figure 14 Quadrature Decoder in Profile 1; Velocity based on Time within a certain number of Ticks

3.2.1.1 Position Tracking

Timer Slice CC40.

The position information of an object in motion is asserted respectively by the:

- External Events Control functions COUNT and UP/DOWN on the Quadrature Clock edge events.
- Direction level events.

Position tracking is achieved by using the COMPARE facilities and watching the status flag ST.

3.2.1.2 Revolution Tracking

Timer Slice CC41.

The update of each entirely elapsed revolution is asserted respectively by the:

- External Event Control functions COUNT and UP/DOWN on the Index (Revolution) edge events.
- Direction level events.

A trigger for a certain number of revolutions can be achieved with a present COMPARE register value.

3.2.1.3 Velocity based on elapsed Time (T/N)

Timer Slices CC42/-43.

The Ticks are asserted to CC42 by the External Events Control function COUNT on the Period Clock edges.

On COMPARE match (= N counts) an event request for CAPTURE (of Time T) is linked to Timer Slice CC43, which will FLUSH/START on the falling ST edge event request by each CC42 period match.

3.2.1.4 Index / Z-Mark (or Top-Mark) Detection

Timer Slice CC42.

By using the Index Occurrence Signal a CLEAR request can be asserted at the 1st (or each time) this signal occurs, to for example reset the time and-period measurements for new velocity calculations.

3.2.1.5 Synchronous Start

Timer Slice CC43.

By using the Synchronous Start signal, a START (or FLUSH/START) request can be asserted to synchronize a timer start with the POSIF start. This signal occurs when the POSIF Run Bit is set.

The CC43 should be set for the External Events Control function START or (FLUSH/START by extended start).

3.2.2 Motion Tracking plus Velocity Monitoring based on Ticks within Time T

This is Motion Monitoring “Profile 2”.

‘Ticks’ should be understood as Period clocks and T as “certain Time T intervals” by Time Stamp.

This velocity monitoring method assumes that the time between the Ticks is negligible, due to the high speed level. (Now the CC42/-43 has exchanged CAPTURE COMPARE tasks).

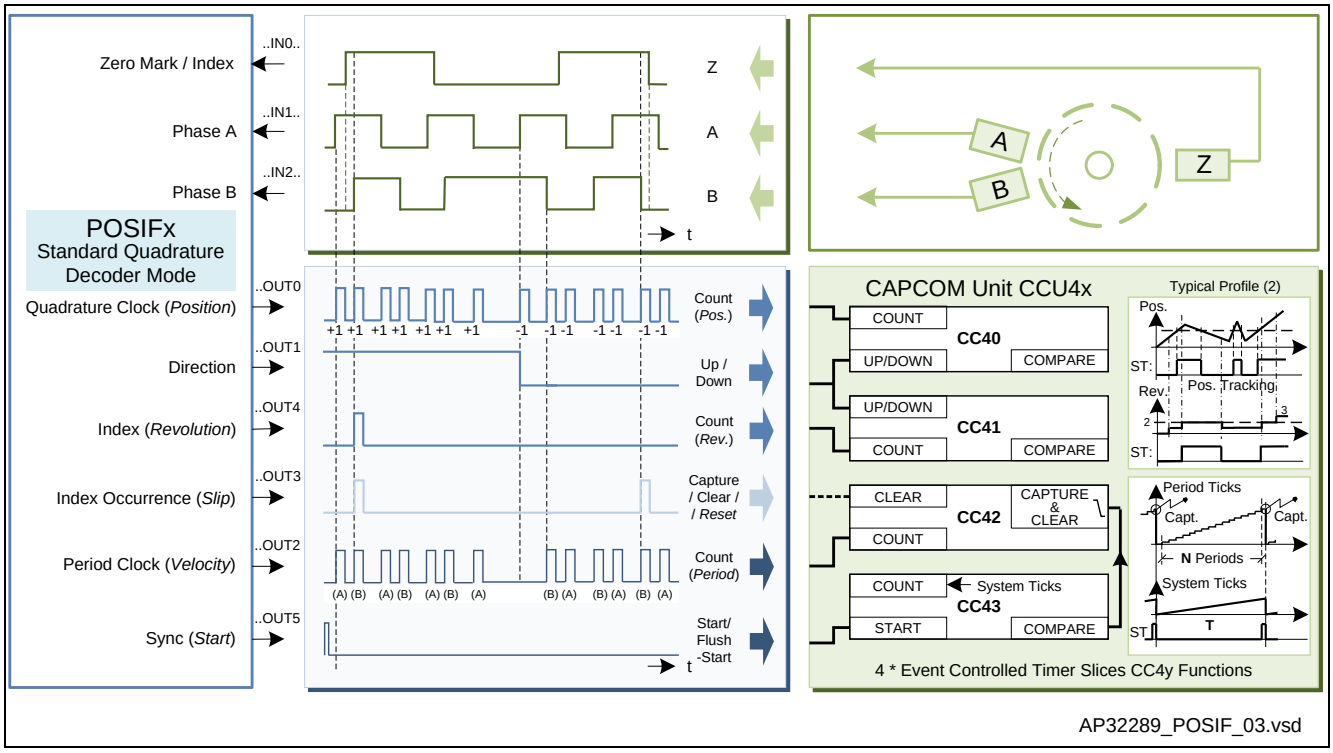


Figure 15 Quadrature Decoder in Profile 2: Velocity based on Number of Ticks within a certain time T

3.2.2.1 Position / Revolution Tracking

Timer Slices CC40/-41.

Position and revolutions are tracked by COMPARE events, as in the “Profile 1” case.

3.2.2.2 Velocity based on elapsed Tick (N/T) – Profile 2

Timer Slices CC42/-43.

The Ticks are counted by CC42 on the External Events Control function COUNT, asserted by the Period clock events.

On CC43 period match (= Time T) an event request is linked to Timer Slice CC42, which will CAPTURE and CLEAR on every negative ST edge, so that the capture of N/T counts is achieved.

4 Multi-Channel Multi-Phase Control

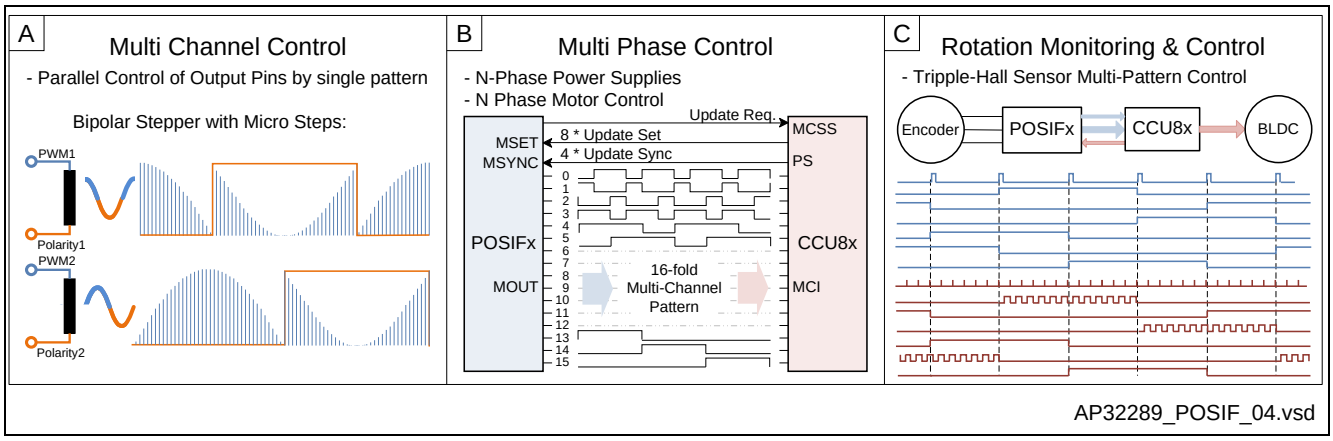


Figure 16 Multi-Channel mode Use Cases

4.1 Principle of Operation

The Multi-Channel mode sub-unit has an input and output port-pair. The input port-pair is used for the Next Pattern Update Set and Synchronization. The output port-pair is used for the Pattern Update Request (MCSS) and Pattern Out for up to 16 CAPCOM slices. This Hardware protocol provides the multi-pattern flow that is required to support a maximum 16-fold channel/phase control loop.

4.1.1 Multi-Channel Next Pattern Update Set Input

An Update Set request by software or from a selected CAPCOM slice to one of the POSIFx.MSET[H...A] inputs will set, if enabled, a Flip-Flop (F/F), the MCMF.MSS bit. This bit is a Multi-Channel Update Trigger that enables a Set Shadow Transfer Request for the POSIF Next Pattern Out, and a ‘MCSS’ for each CAPCOM slice involved.

4.1.2 Multi-Channel Pattern Update Synchronization Input

After a Next Pattern Update Set has been asserted by setting the Multi-Channel Update Trigger Flip-Flop (F/F), the pattern update can be synchronized with for example a PWM signal. This signal should be mapped to one of the POSIFx.MSYNC[D...A] inputs, which will assert “synchronization event = true” upon a falling edge on this input.

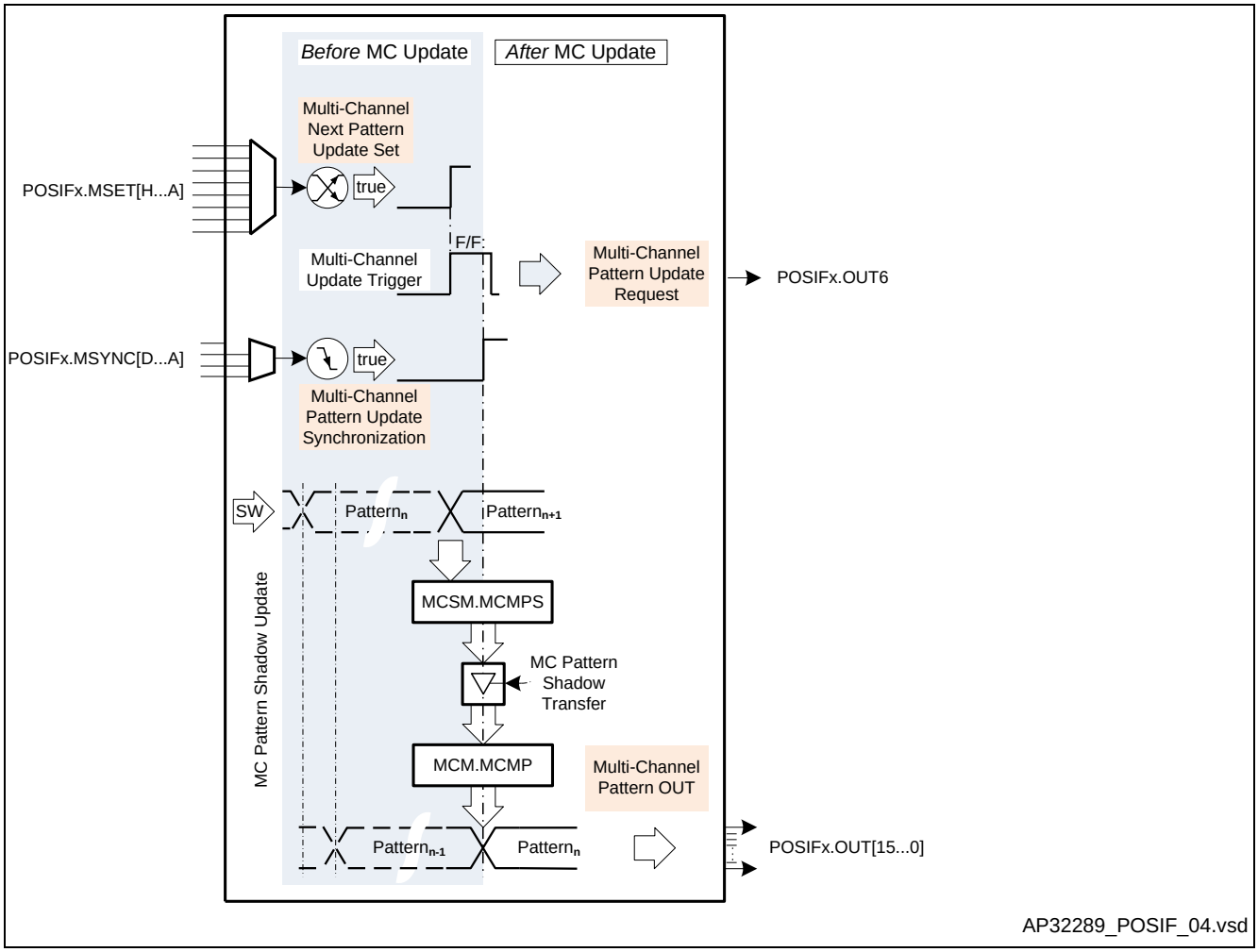


Figure 17 Multi-Channel mode sub-unit

4.1.3 Multi-Channel Pattern Update Request Output

The POSIFx.OUT6 output pin asserts a Multi-Pattern Set Shadow Transfer Request Enable by its pulse to the MCSS inputs of all CAPCOM slices that are selected for Multi-Channel Pattern Update in parallel. This pulse is the Multi-Channel Update Trigger Flip-Flop (F/F) status and will be cleared after a multi-pattern update.

4.1.4 Multi-Channel Pattern Shadow Transfer and Pattern Output

On a POSIFx.MSYNC[D...A] input falling edge, a Multi-Channel Pattern Shadow Transfer is performed to the POSIFx.MOUT[15…0] output. This pattern may control, via the input (MCIy), the output (OUTy) of all slices that are Multi-Channel Mode Enabled (by MCME), and Slice Shadow Transfer Enabled (globally by GCTRL.MSEy).

4.1.5 POSIF-to-CCU-Slices Multi-Channel Pattern Transfer Synchronization

A slice in Multi-Channel Mode prolongs its status bit (CCST) by 3 module clocks, to be valid even after a Pattern Set (PS) request has been asserted. Therefore the OUTy operation of such a slice will include the requested Multi-Channel Input Pattern (MCIy) and (CCST), and so the POSIF-to-CCU-Slices pattern transfer will be provided.

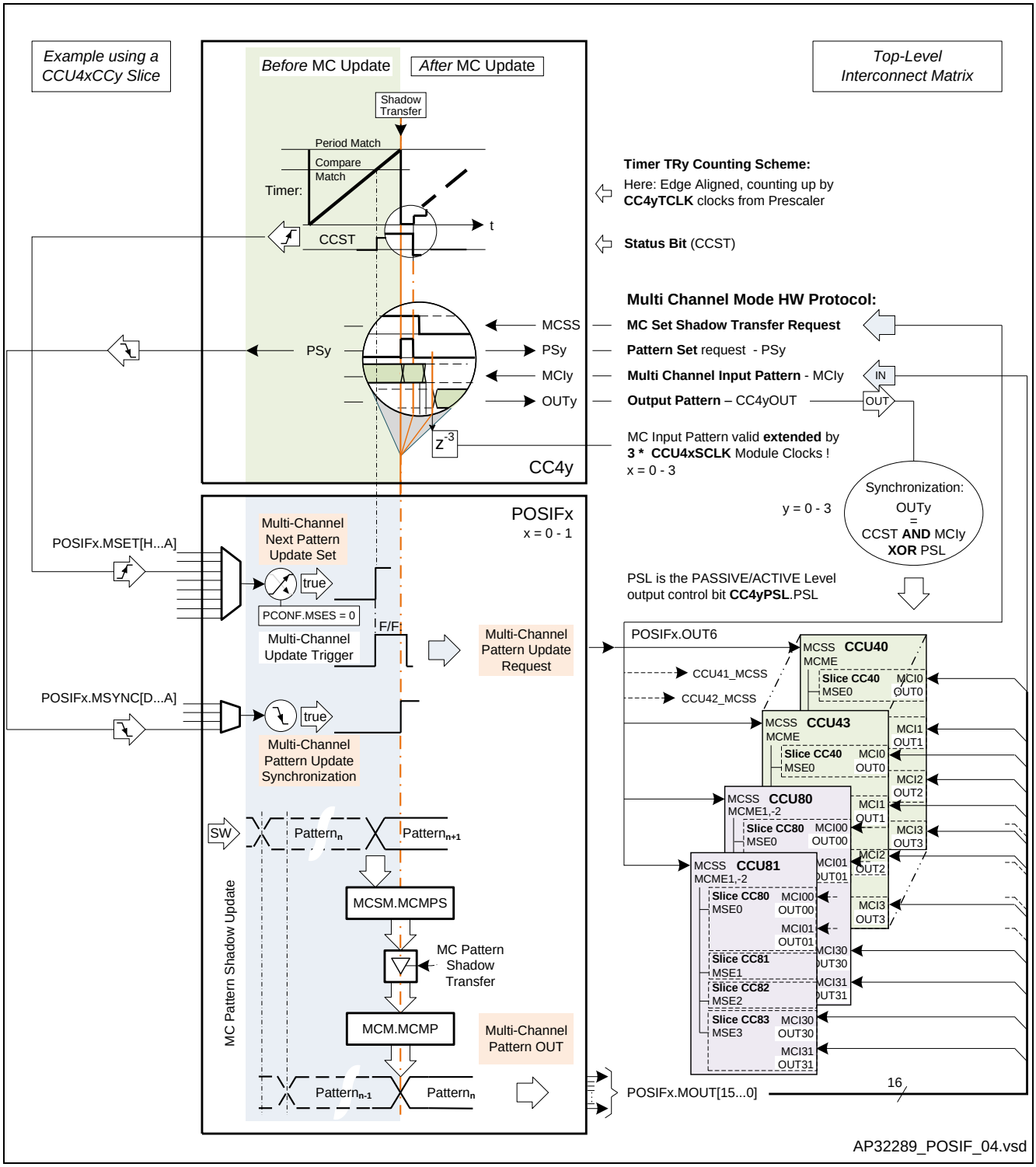


Figure 18 The POSIF-to-CCU-Slices Multi-Channel Pattern Transfer : Profile Example

4.2 Multi-Channel Unit in Stand-Alone Mode

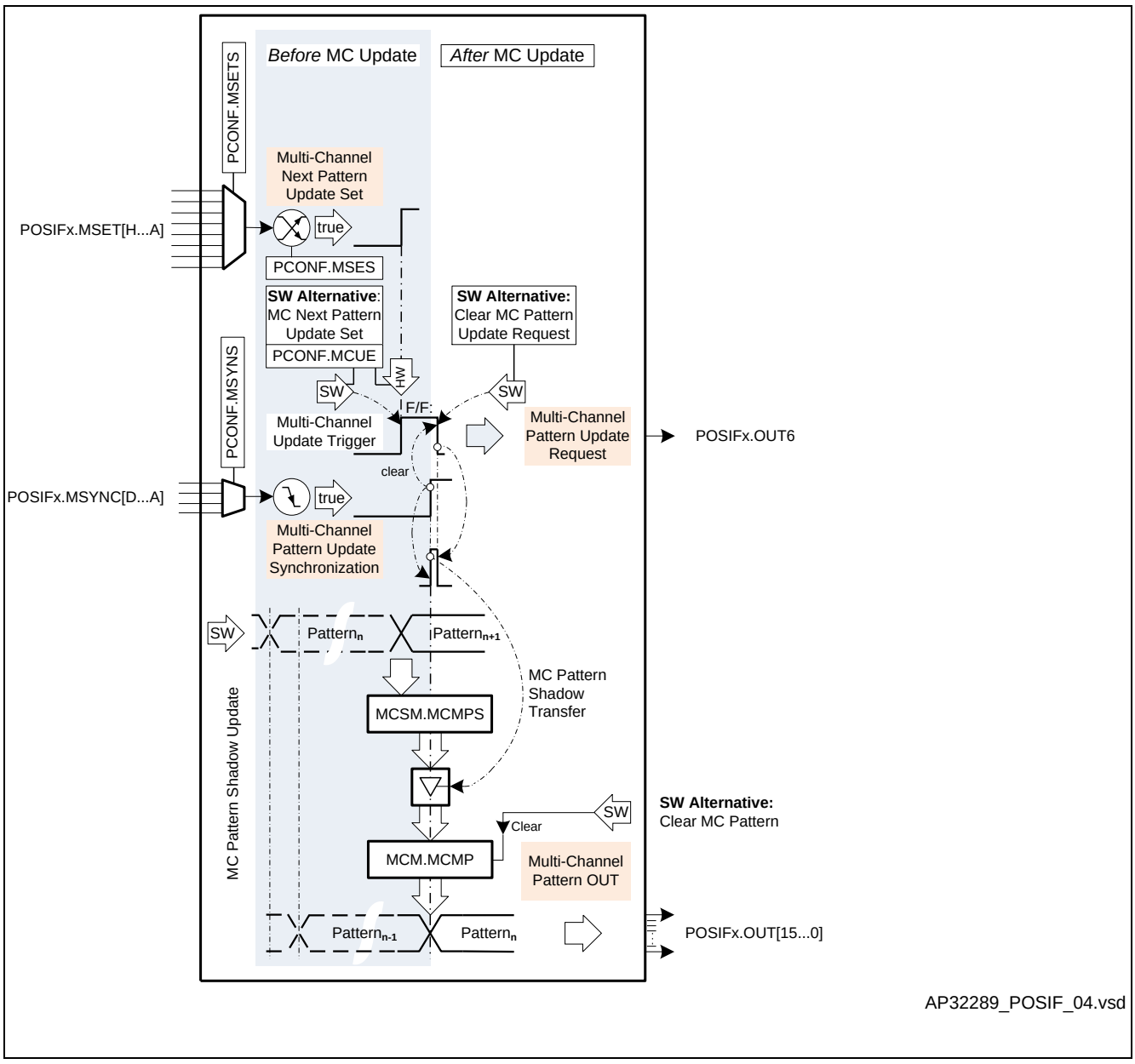


Figure 19 POSIFx Multi-Channel Unit in stand-alone mode

4.2.1 Next Pattern Update Set Inputs in stand-alone mode

A Multi-Channel Next Update Set can be asserted either by software (selected by PCONF.MCUE=1_B), or via one of the POSIFx.MSET[H...A] inputs (selected by the PCONF.MSETS bitfield) on a signal transition (selected by PCONF.MSES). The MCMF.MSS bit (F/F) will be set and cleared by “synchronization event = true”, or by software.

4.2.2 Pattern Update Synchronization Inputs in stand-alone mode

After the Multi-Channel Update Trigger request into Flip-Flop (F/F) has been set, the pattern update can be synchronized with for example a PWM signal. This signal should be mapped to one of the POSIFx.MSYNC[D...A] inputs (selected by the PCONF.MSYNS bitfield), which will assert a “synchronization event = true” upon a falling edge.

4.2.3 Multi-Channel Pattern Update Request Output in stand-alone mode

The POSIFx.OUT6 output pin asserts a Multi-Pattern Set Shadow Transfer Request Enable by its pulse to the MCSS inputs of all CAPCOM slices that are selected for Multi-Channel Pattern Update in parallel. This pulse is the Multi-Channel Update Trigger Flip-Flop (F/F) status that will be cleared by hardware (or software) on Update Done.

4.2.4 Multi-Channel Pattern Shadow Transfer in stand-alone mode

Upon a POSIFx.MSYNC[D...A] input falling edge, a Multi-Channel Pattern Shadow Transfer is performed from the MCSM.MCMPS shadow register to the POSIFx.MOUT[15…0] register output pins. This output register should be maintained by software and may be cleared at any time by software to stop for example PWM generating units.

4.3 Multi-Channel Unit in Hall-Sensor mode

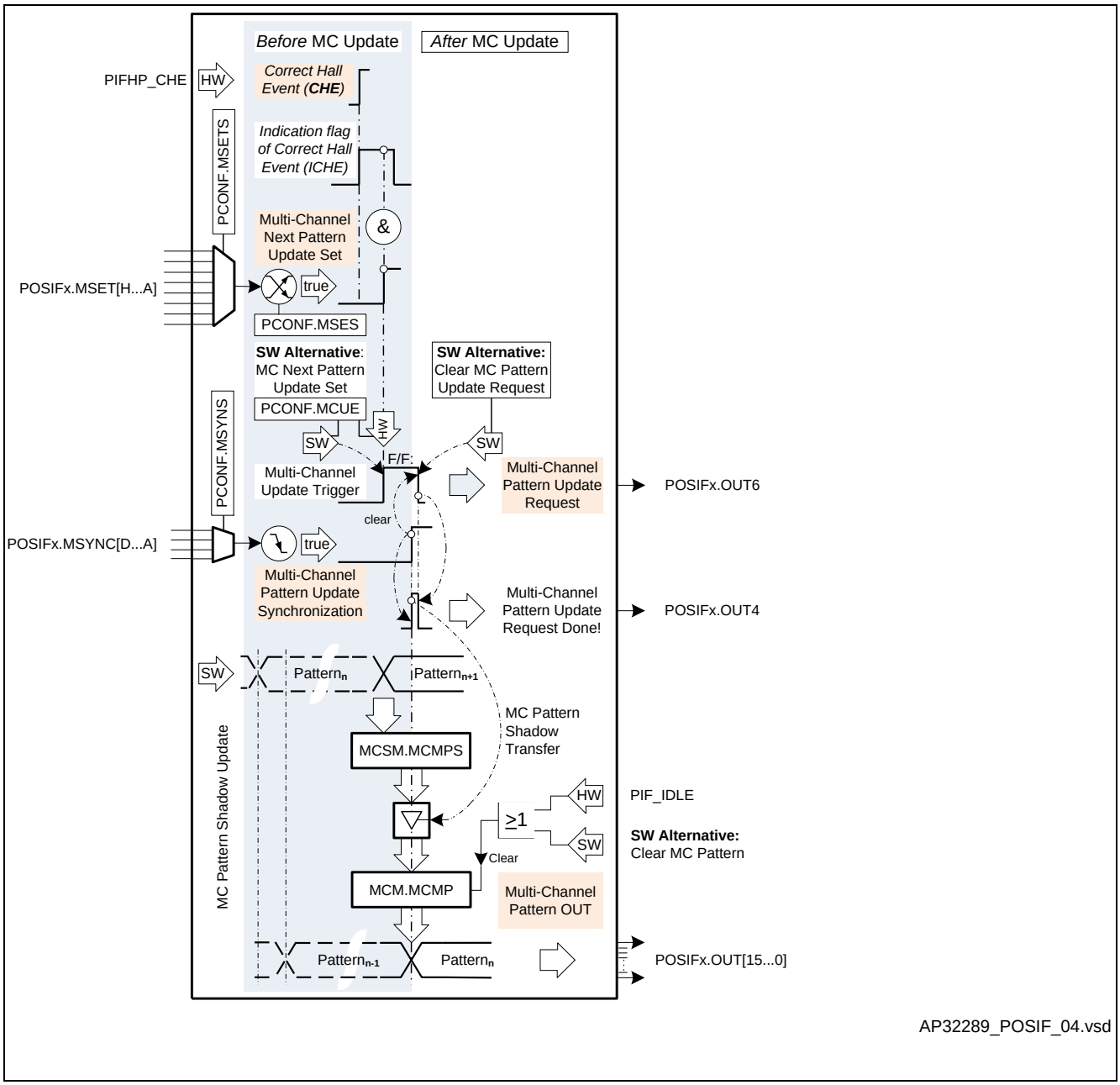


Figure 20 POSIFx Multi-Channel Unit in Hall Sensor mode

4.3.1 Next Pattern Update Set Inputs in Hall-Sensor mode

A Multi-Channel Next Update Set can be asserted either by software (selected by PCONF.MCUE=1_B) or via one of the POSIFx.MSET[H...A] inputs (selected by the PCONF.MSETS bitfield) on a signal transition (selected by PCONF.MSES). However, the MC Update Trigger MCMF.MSS bit (F/F) will only be set at Correct Hall Events.

4.3.2 Pattern Update Synchronization Inputs in Hall-Sensor mode

After the Multi-Channel Update Trigger request into Flip-Flop (F/F) has been set, the pattern update can be synchronized with for example a PWM signal. This signal should be mapped to one of the POSIFx.MSYNC[D...A] inputs (selected by the PCONF.MSYNS bitfield), which will assert a “synchronization event = true” on a falling edge.

4.3.3 Multi-Channel Pattern Update Request Output in Hall-Sensor mode

The POSIFx.OUT6 output pin asserts a Multi-Pattern Set Shadow Transfer Request Enable by its pulse to the MCSS inputs of all CAPCOM slices that are selected for Multi-Channel Pattern Update in parallel. The POSIFx.OUT4 output pin asserts “synchronization event = true” and claims “Update done”, when it is cleared.

4.3.4 Multi-Channel Pattern Shadow Transfer in Hall-Sensor mode

Upon a POSIFx.MSYNC[D...A] input falling edge, a Multi-Channel Pattern Shadow Transfer is performed from the MCSM.MCMPS shadow register to the POSIFx.MOUT[15…0] register output pins. The shadow register should be updated by software. The output can be reset by software or hardware (“POSIF Idle”), to stop PWM generating units.

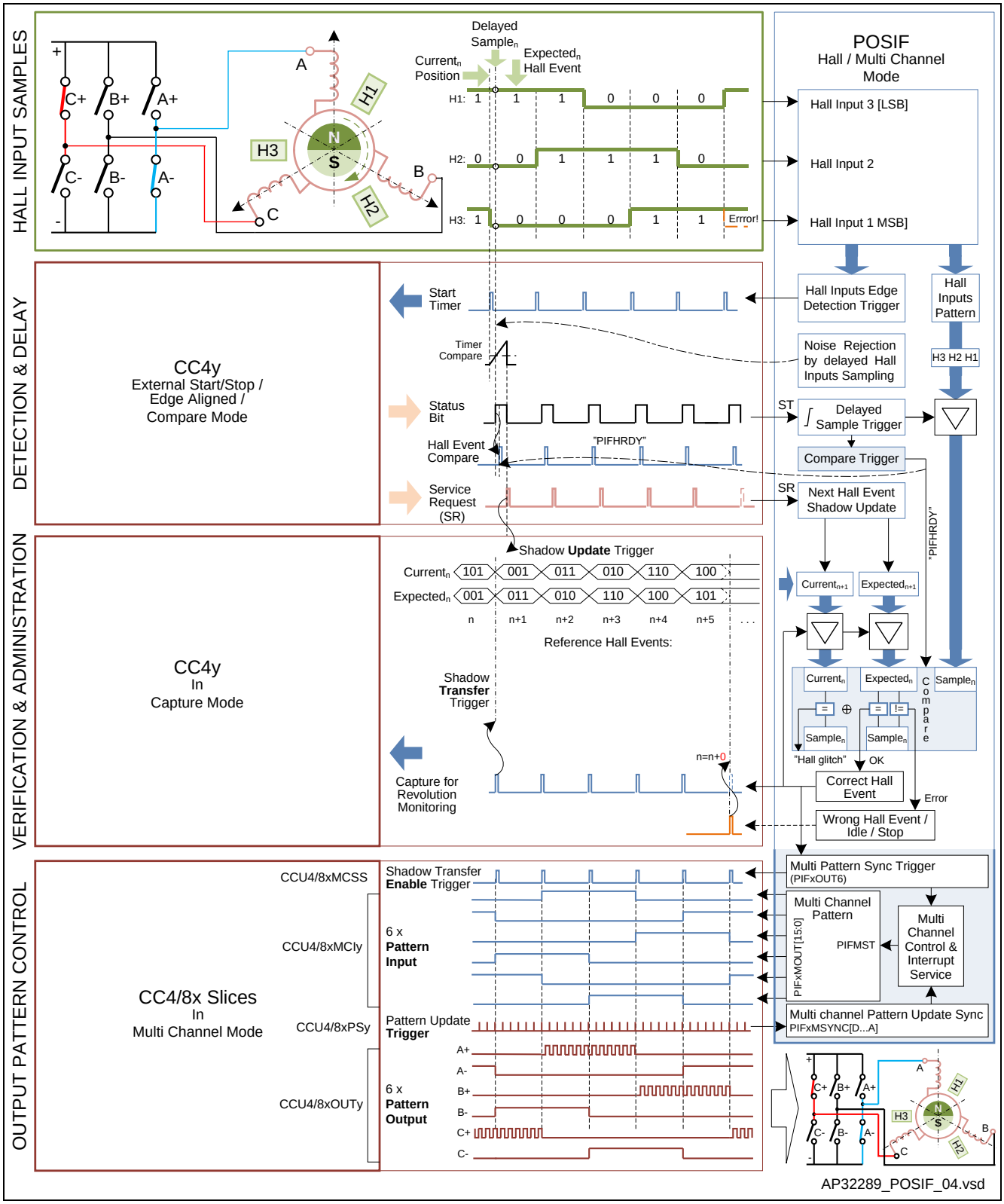


Figure 21 Triple-Hall BLDC Motor Commutation Control using CAPCOM units in Multi-Channel mode

4.4 Example Use Case: Using Stand-alone Multi-Channel mode to modulate PWM signals

This example uses the POSIF in stand-alone Multi-Channel mode to modulate PWM signals. It uses a CCU4 slice (CCU40.CC40) to update a new pattern on compare match interrupt. The pattern is incremented by 1 each time a compare match is entered and written to the POSIF Multi-Channel mode shadow pattern register. The update is not immediate, but is triggered from CCU80.CC81 PS1 for the update. This synchronizes the update pattern with the modulated PWM signals. The CCU4 and CCU8 slices are configured to synchronously start on an external event 1 from SCU Global Start Control signal. The PWM frequency and duty cycle on the CCU4 and CCU8 slices are 20 kHz and set at 50%. It is targeted for the XMC1300 device.

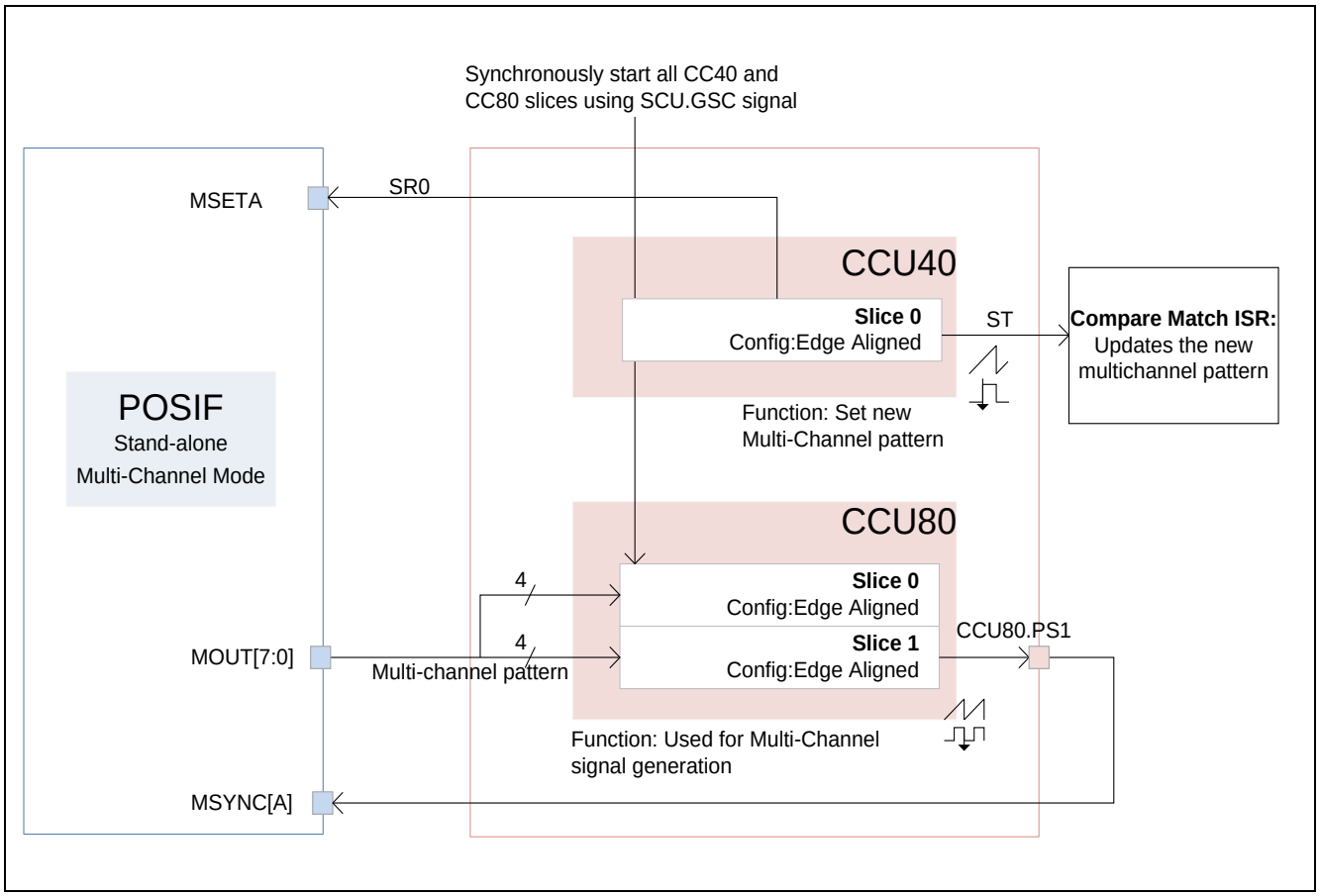
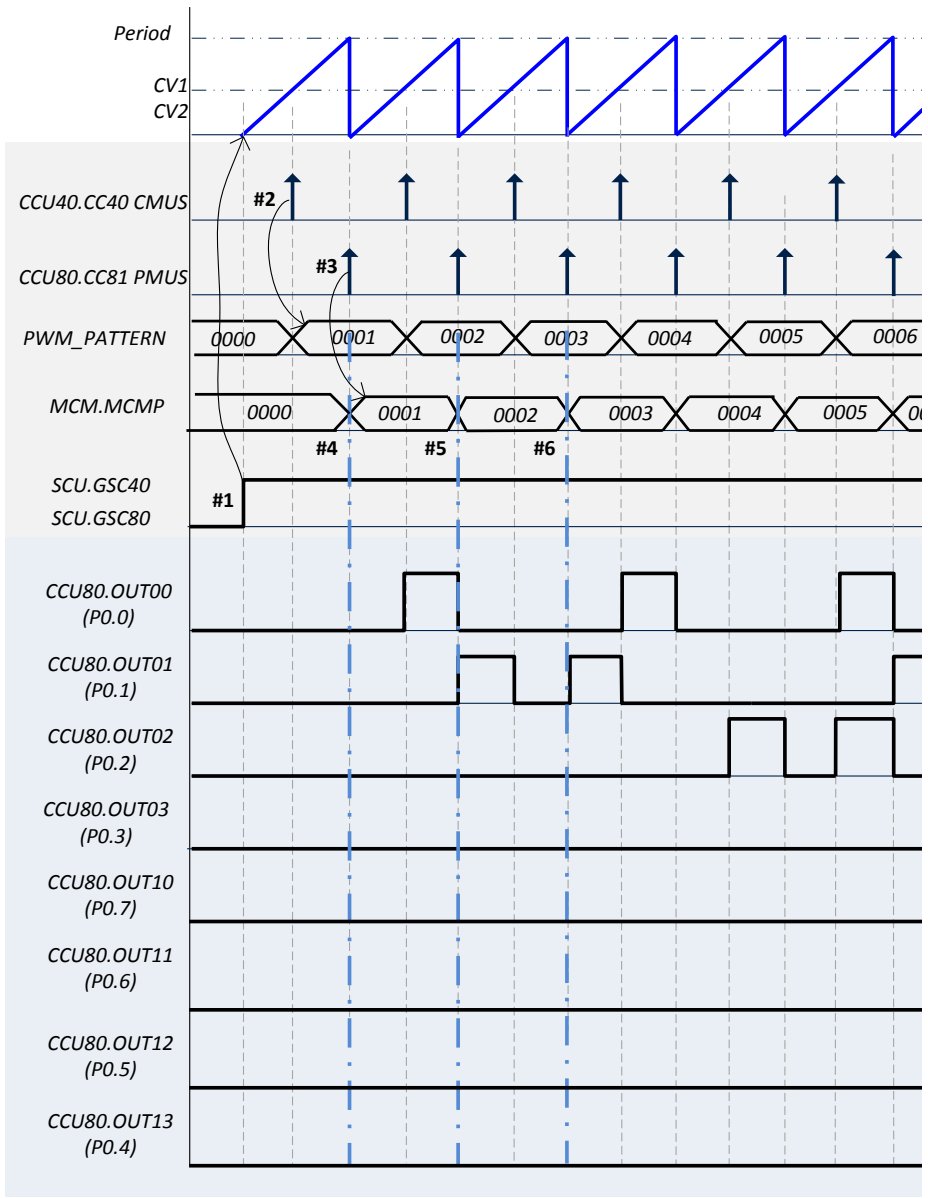


Figure 22 Example: Setup for Stand-alone Multi-Channel mode to modulate PWM signals



SLICE Configuration:
XMC1300
System Clock = 64 MHz

CCU4 and CCU8 are configured with the same configuration:
Frequency = 20 kHz
CV1, CV2 = 50% Duty Cycle
Mode = Edge-aligned, Counting up

#1: CCU4 and CCU8 slices are started synchronously on external event 1 from a rising edge of SCU.GSC40, SCU.GSC80.

#2: CCU4 compare match interrupt on SR0 is triggered. A new PWM_PATTERN is written to the Shadow Multi-Channel Pattern. A Multi-Channel Pattern Update Enable Set (MNPS) is set to synchronize the update with the PWM (CCU80.CC81).

#3: At CCU81 period match, the new Multi-Channel Pattern is loaded. This is used to synchronize the update of the multi-channel pattern with the shadow transfer. The new Multi-Channel Pattern is used to modulate with the CCU8 output.

#4: Only MOUT[0] is 1 and CCU80.OUT00 output is available.

#5: Only MOUT[1] is 1 and CCU80.OUT01 output is available.

#6: Only MOUT[0] and MOUT[1] are 1. CCU80.OUT00 and CCU80.OUT01 outputs are available.

Figure 23 Example: Using Stand-alone Multi-Channel mode to modulate PWM signals

4.4.1 Macro and variable Settings

XMC Lib Project includes

```
#include <xmc_ccu4.h>
#include <xmc_ccu8.h>
#include <xmc_gpio.h>
#include <xmc_posif.h>
#include <xmc_scu.h>
```

Project Macro definitions

```
/* POSIF Macros */
#define POSIF_PTR POSIF0

/* CCU4 Macros */
#define CCU40_MODULE_PTR        CCU40
#define CCU40_MODULE_NUMBER     (0U)

#define CCU40_SLICE0_PTR        CCU40_CC40
#define CCU40_SLICE0_NUMBER     (0U)
#define CCU40_SLICE0_OUTPUT     P1_0

/* CCU8 Macros */
#define CCU80_MODULE_PTR        CCU80
#define CCU80_MODULE_NUMBER     (0U)

#define CCU80_SLICE0_PTR        CCU80_CC80
#define CCU80_SLICE0_NUMBER     (0U)
#define CCU80_SLICE0_OUTPUT00   P0_0
#define CCU80_SLICE0_OUTPUT01   P0_1
#define CCU80_SLICE0_OUTPUT02   P0_2
#define CCU80_SLICE0_OUTPUT03   P0_3

#define CCU80_SLICE1_PTR        CCU80_CC81
#define CCU80_SLICE1_NUMBER     (1U)
#define CCU80_SLICE1_OUTPUT10   P0_7
#define CCU80_SLICE1_OUTPUT11   P0_6
#define CCU80_SLICE1_OUTPUT12   P0_5
#define CCU80_SLICE1_OUTPUT13   P0_4
```

Project Variables Definition

```
volatile uint16_t PWMPATTERN=0;
```

4.4.2 XMC Lib Peripheral Configuration Structure

XMC System Clock Unit (SCU) Configuration

PWM period is calculated based on PCLK which is equivalent to 64 MHz.

```
/* XMC System Clock Unit (SCU) Configuration: */
/* PWM period is calculated based on PCLK which is equivalent to 64 MHz. */
XMC_SCU_CLOCK_CONFIG_t clock_config =
{
  .pclk_src = XMC_SCU_CLOCK_PCLKSRC_DOUBLE_MCLK,
  .rtc_src = XMC_SCU_CLOCK_RTCCLKSRC_DCO2,
  .fdiv = 0,
  .idiv = 1,
};
```

XMC Compare Unit 4 (CCU4) Configuration

```
/* XMC Compare Unit 4 (CCU4) Configuration: */
XMC_CCU4_SLICE_COMPARE_CONFIG_t CCU40_SLICE_config =
{
  .timer_mode          = (uint32_t) XMC_CCU4_SLICE_TIMER_COUNT_MODE_EA,
  .monoshot            = (uint32_t) false,
  .shadow_xfer_clear   = (uint32_t) 0,
  .dither_timer_period = (uint32_t) 0,
  .dither_duty_cycle   = (uint32_t) 0,
  .prescaler_mode      = (uint32_t) XMC_CCU4_SLICE_PRESCALER_MODE_NORMAL,
  .mcm_enable          = (uint32_t) 0,
  .prescaler_initval   = (uint32_t) 0,
  .float_limit         = (uint32_t) 0,
  .dither_limit        = (uint32_t) 0,
  .passive_level       = (uint32_t) XMC_CCU4_SLICE_OUTPUT_PASSIVE_LEVEL_LOW,
  .timer_concatenation = (uint32_t) 0
};

XMC_CCU4_SLICE_EVENT_CONFIG_t CCU40_SLICE_event0_config =
{
  .mapped_input = XMC_CCU4_SLICE_INPUT_I,      /* mapped to SCU.GSC40 */
  .edge         = XMC_CCU4_SLICE_EVENT_EDGE_SENSITIVITY_RISING_EDGE,
  .level        = XMC_CCU4_SLICE_EVENT_LEVEL_SENSITIVITY_ACTIVE_HIGH,
  .duration     = XMC_CCU4_SLICE_EVENT_FILTER_3_CYCLES
};
```

XMC Compare Unit 8 (CCU8) Configuration

```
/* XMC CCU8 configuration structure */

XMC_CCU8_SLICE_COMPARE_CONFIG_t  CCU80_SLICE_config     =
{
   .timer_mode           = (uint32_t)XMC_CCU8_SLICE_TIMER_COUNT_MODE_EA,
   .monoshot             = (uint32_t)XMC_CCU8_SLICE_TIMER_REPEAT_MODE_REPEAT,
   .shadow_xfer_clear    = 0U,
   .dither_timer_period  = 0U,
   .dither_duty_cycle    = 0U,
```

```
    .prescaler_mode        = (uint32_t)XMC_CCU8_SLICE_PRESCALER_MODE_NORMAL,
    .mcm_ch1_enable        = 1U,
    .mcm_ch2_enable        = 1U,
    .slice_status          = (uint32_t)XMC_CCU8_SLICE_STATUS_CHANNEL_1,
    .passive_level_out0    = (uint32_t)XMC_CCU8_SLICE_OUTPUT_PASSIVE_LEVEL_LOW,
    .passive_level_out1    = (uint32_t)XMC_CCU8_SLICE_OUTPUT_PASSIVE_LEVEL_LOW,
    .passive_level_out2    = (uint32_t)XMC_CCU8_SLICE_OUTPUT_PASSIVE_LEVEL_LOW,
    .passive_level_out3    = (uint32_t)XMC_CCU8_SLICE_OUTPUT_PASSIVE_LEVEL_LOW,
    .asymmetric_pwm        = 0U,
    .invert_out0           = 0U,
    .invert_out1           = 1U,
    .invert_out2           = 0U,
    .invert_out3           = 1U,
    .prescaler_initval     = 0U,
    .float_limit           = 0U,
    .dither_limit          = 0U,
    .timer_concatenation   = 0U,
};

XMC_CCU8_SLICE_EVENT_CONFIG_t CCU80_SLICE_event0_config =
{
    .mapped_input        = XMC_CCU8_SLICE_INPUT_H,      //Connected to SCU.GSC80
    .edge                = XMC_CCU8_SLICE_EVENT_EDGE_SENSITIVITY_RISING_EDGE,
    .level               = XMC_CCU8_SLICE_EVENT_LEVEL_SENSITIVITY_ACTIVE_LOW,
    .duration            = XMC_CCU8_SLICE_EVENT_FILTER_DISABLED,
};
```

XMC Position Interface Unit (POSIF) Configuration

```
/* Configuration for POSIF - Multi-Channel Mode */
XMC_POSIF_CONFIG_t POSIF_config =
{
  .mode   = XMC_POSIF_MODE_MCM,    /**< POSIF Operational mode */
  .input0 = XMC_POSIF_INPUT_PORT_A,    /**< Choice of input for Input-1 */
  .input1 = XMC_POSIF_INPUT_PORT_A,    /**< Choice of input for Input-2 */
  .input2 = XMC_POSIF_INPUT_PORT_A,   /**< Choice of input for Input-3 */
  .filter = XMC_POSIF_FILTER_DISABLED   /**< Input filter configuration */
};

/* Configuration for POSIF - Multi-Channel Mode update settings */
XMC_POSIF_MCM_CONFIG_t POSIF_MCM_config =
{
 .pattern_sw_update            = (uint8_t)false,
 .pattern_update_trigger       = XMC_POSIF_INPUT_PORT_A,              /* CCU40.SR0 */
 .pattern_trigger_edge         = XMC_POSIF_HSC_TRIGGER_EDGE_RISING,
 .pwm_sync                     = (uint8_t)XMC_POSIF_INPUT_PORT_A      /* CCU80.PS1 */
};
```

XMC GPIO Configuration

```
/* XMC GPIO Configuration: P1_0 */
XMC_GPIO_CONFIG_t CCU40_SLICE_OUTPUT_config =
{
```

```
    .mode = XMC_GPIO_MODE_OUTPUT_PUSH_PULL_ALT2,
    .input_hysteresis = XMC_GPIO_INPUT_HYSTERESIS_STANDARD,
    .output_level = XMC_GPIO_OUTPUT_LEVEL_LOW,
};
/* Configuration for standard pads: Port0;[0:7] */
XMC_GPIO_CONFIG_t CCU80_SLICE_OUTPUT_config =
{
    .mode = XMC_GPIO_MODE_OUTPUT_PUSH_PULL_ALT5,
    .input_hysteresis = XMC_GPIO_INPUT_HYSTERESIS_STANDARD,
    .output_level = XMC_GPIO_OUTPUT_LEVEL_LOW,
};
```

4.4.3 Interrupt Service Routine Function Implementation

The CCU40 interrupt handler function updates the new multi-channel pattern.

```
/* CCU40 Compare Match ISR for the new multi channel pattern. */
void CCU40_0_IRQHandler(void)
{
   /* Acknowledge CCU4 compare match event*/
   XMC_CCU4_SLICE_ClearEvent(CCU40_SLICE0_PTR, \
        XMC_CCU4_SLICE_IRQ_ID_COMPARE_MATCH_UP);

   /* Increment and prepare for the next PWM pattern */
   PWMPATTERN++;

   /* Write new multichannel pattern */
   XMC_POSIF_MCM_SetMultiChannelPattern(POSIF_PTR, PWMPATTERN);

}
```

4.4.4 Main Function Implementation

Before the start and execution of timer slice software for the first time, the CCU4 must have been initialized appropriately in the following sequence:

- Clock setup

```
/* Ensure clock frequency is set at 64MHz (2*MCLK) */
XMC_SCU_CLOCK_Init(&clock_config);
```

- Enable clock, enable pre-scaler block and configure global control

```
/* Enable clock, enable prescaler block and configure global control */
XMC_CCU4_Init(CCU40_MODULE_PTR, XMC_CCU4_SLICE_MCMS_ACTION_TRANSFER_PR_CR);
XMC_CCU8_Init(CCU80_MODULE_PTR, XMC_CCU8_SLICE_MCMS_ACTION_TRANSFER_PR_CR);

/* Start the prescaler and restore clocks to slices */
XMC_CCU4_StartPrescaler(CCU40_MODULE_PTR);
XMC_CCU8_StartPrescaler(CCU80_MODULE_PTR);

/* Ensure fCCU reaches CCU40, CCU80 */
XMC_CCU4_SetModuleClock(CCU40_MODULE_PTR, XMC_CCU4_CLOCK_SCU);
XMC_CCU8_SetModuleClock(CCU80_MODULE_PTR, XMC_CCU8_CLOCK_SCU);
```

- Configure Slice(s) Functions, Interrupts and Start-up

```
/* Configure CCU8x_CC8y slice as timer */
XMC_CCU4_SLICE_CompareInit(CCU40_SLICE0_PTR, &CCU40_SLICE_config);
XMC_CCU8_SLICE_CompareInit(CCU80_SLICE0_PTR, &CCU80_SLICE_config);
XMC_CCU8_SLICE_CompareInit(CCU80_SLICE1_PTR, &CCU80_SLICE_config);

/* Set period match value of the timer  */
XMC_CCU4_SLICE_SetTimerPeriodMatch(CCU40_SLICE0_PTR, 3199U);
XMC_CCU8_SLICE_SetTimerPeriodMatch(CCU80_SLICE0_PTR, 3199U);
XMC_CCU8_SLICE_SetTimerPeriodMatch(CCU80_SLICE1_PTR, 3199U);

/* Set timer compare match value for channel (50%) of period */
XMC_CCU4_SLICE_SetTimerCompareMatch(CCU40_SLICE0_PTR, 1600U);
XMC_CCU8_SLICE_SetTimerCompareMatch(CCU80_SLICE0_PTR, \
   XMC_CCU8_SLICE_COMPARE_CHANNEL_1, 1600U);
XMC_CCU8_SLICE_SetTimerCompareMatch(CCU80_SLICE0_PTR, \
   XMC_CCU8_SLICE_COMPARE_CHANNEL_2, 1600U);
XMC_CCU8_SLICE_SetTimerCompareMatch(CCU80_SLICE1_PTR, \
   XMC_CCU8_SLICE_COMPARE_CHANNEL_1, 1600U);
XMC_CCU8_SLICE_SetTimerCompareMatch(CCU80_SLICE1_PTR, \
   XMC_CCU8_SLICE_COMPARE_CHANNEL_2, 1600U);

/* Transfer value from shadow timer registers to actual timer registers */
XMC_CCU4_EnableShadowTransfer(CCU40_MODULE_PTR, \
(uint32_t) XMC_CCU4_SHADOW_TRANSFER_SLICE_0);
XMC_CCU8_EnableShadowTransfer(CCU80_MODULE_PTR, (uint32_t)\
   (XMC_CCU8_SHADOW_TRANSFER_SLICE_0|XMC_CCU8_SHADOW_TRANSFER_SLICE_1));

/* Configure events */
XMC_CCU4_SLICE_ConfigureEvent(CCU40_SLICE0_PTR, \
   XMC_CCU4_SLICE_EVENT_0, &CCU40_SLICE_event0_config);
XMC_CCU8_SLICE_ConfigureEvent(CCU80_SLICE0_PTR, \
   XMC_CCU8_SLICE_EVENT_0, &CCU80_SLICE_event0_config);
XMC_CCU8_SLICE_ConfigureEvent(CCU80_SLICE1_PTR, \
   XMC_CCU8_SLICE_EVENT_0, &CCU80_SLICE_event0_config);

XMC_CCU4_SLICE_StartConfig(CCU40_SLICE0_PTR, \
   XMC_CCU4_SLICE_EVENT_0, XMC_CCU4_SLICE_START_MODE_TIMER_START_CLEAR);
XMC_CCU8_SLICE_StartConfig(CCU80_SLICE0_PTR, \
   XMC_CCU8_SLICE_EVENT_0, XMC_CCU8_SLICE_START_MODE_TIMER_START_CLEAR);
XMC_CCU8_SLICE_StartConfig(CCU80_SLICE1_PTR, \
   XMC_CCU8_SLICE_EVENT_0, XMC_CCU8_SLICE_START_MODE_TIMER_START_CLEAR);

/* Enable events */
/*Enable Compare match event for pattern update*/
XMC_CCU4_SLICE_EnableEvent(CCU40_SLICE0_PTR, XMC_CCU4_SLICE_IRQ_ID_COMPARE_MATCH_UP);

/*Enable Period match event for synchronizing the pattern update with the PWM */
XMC_CCU8_SLICE_EnableEvent(CCU80_SLICE1_PTR, XMC_CCU8_SLICE_IRQ_ID_PERIOD_MATCH);
```

```
 /* Connect compare match event to SR0 - MSETA*/
XMC_CCU4_SLICE_SetInterruptNode(CCU40_SLICE0_PTR, \
XMC_CCU4_SLICE_IRQ_ID_COMPARE_MATCH_UP, XMC_CCU4_SLICE_SR_ID_0);

/* Connect period match event to SR1 - connect as input to MSYNC*/
XMC_CCU8_SLICE_SetInterruptNode(CCU80_SLICE1_PTR, \
XMC_CCU8_SLICE_IRQ_ID_PERIOD_MATCH, XMC_CCU8_SLICE_SR_ID_1);

/* Set NVIC priority */
NVIC_SetPriority(CCU40_0_IRQn, 3U);

/* Enable IRQ */
NVIC_EnableIRQ(CCU40_0_IRQn);

/*Initializes the GPIO*/
XMC_GPIO_Init(CCU40_SLICE0_OUTPUT, &CCU40_SLICE_OUTPUT_config);

XMC_GPIO_Init(CCU80_SLICE0_OUTPUT00, &CCU80_SLICE_OUTPUT_config);
XMC_GPIO_Init(CCU80_SLICE0_OUTPUT01, &CCU80_SLICE_OUTPUT_config);
XMC_GPIO_Init(CCU80_SLICE0_OUTPUT02, &CCU80_SLICE_OUTPUT_config);
XMC_GPIO_Init(CCU80_SLICE0_OUTPUT03, &CCU80_SLICE_OUTPUT_config);

XMC_GPIO_Init(CCU80_SLICE1_OUTPUT10, &CCU80_SLICE_OUTPUT_config);
XMC_GPIO_Init(CCU80_SLICE1_OUTPUT11, &CCU80_SLICE_OUTPUT_config);
XMC_GPIO_Init(CCU80_SLICE1_OUTPUT12, &CCU80_SLICE_OUTPUT_config);
XMC_GPIO_Init(CCU80_SLICE1_OUTPUT13, &CCU80_SLICE_OUTPUT_config);
```

- Configure POSIF in standalone Multi-Channel Mode

```
/* POSIF Configuration - Standalone mode */
XMC_POSIF_Init(POSIF_PTR, &POSIF_config);
XMC_POSIF_MCM_Init(POSIF_PTR, &POSIF_MCM_config);

/* Start the POSIF module*/
XMC_POSIF_Start(POSIF_PTR);
```

- Enable the CCU80 Slices and start the timers

```
/* Get the slice out of idle mode */
XMC_CCU8_EnableClock(CCU80_MODULE_PTR, CCU80_SLICE0_NUMBER);
XMC_CCU8_EnableClock(CCU80_MODULE_PTR, CCU80_SLICE1_NUMBER);
XMC_CCU4_EnableClock(CCU40_MODULE_PTR, CCU40_SLICE0_NUMBER);

/* Start the PWM on a rising edge on SCU.GSC40 and GSC80 */
XMC_SCU_SetCcuTriggerHigh((uint32_t)\
   (XMC_SCU_CCU_TRIGGER_CCU40|XMC_SCU_CCU_TRIGGER_CCU80));
```

5 Revision History

Current Version is V1.0, 2015-07

Page or Reference	Description of change
V1.0, 2015-07	
	Initial Version

www.infineon.com

Edition 2015-07

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about any aspect of this document?

Email: erratum@infineon.com

Document reference

AP32289

Legal Disclaimer

THE INFORMATION GIVEN IN THIS APPLICATION NOTE (INCLUDING BUT NOT LIMITED TO CONTENTS OF REFERENCED WEBSITES) IS GIVEN AS A HINT FOR THE IMPLEMENTATION OF THE INFINEON TECHNOLOGIES COMPONENT ONLY AND SHALL NOT BE REGARDED AS ANY DESCRIPTION OR WARRANTY OF A CERTAIN FUNCTIONALITY, CONDITION OR QUALITY OF THE INFINEON TECHNOLOGIES COMPONENT. THE RECIPIENT OF THIS APPLICATION NOTE MUST VERIFY ANY FUNCTION DESCRIBED HEREIN IN THE REAL APPLICATION. INFINEON TECHNOLOGIES HEREBY DISCLAIMS ANY AND ALL WARRANTIES AND LIABILITIES OF ANY KIND (INCLUDING WITHOUT LIMITATION WARRANTIES OF NON-INFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS OF ANY THIRD PARTY) WITH RESPECT TO ANY AND ALL INFORMATION GIVEN IN THIS APPLICATION NOTE.

Information

For further information on technology, delivery terms and conditions and prices, please contact the nearest Infineon Technologies Office (**www.infineon.com**).

Warnings

Due to technical requirements, components may contain dangerous substances. For information on the types in question, please contact the nearest Infineon Technologies Office. Infineon Technologies components may be used in life-support devices or systems only with the express written approval of Infineon Technologies, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.