AN5760

Configuring Event Settings for VSC8254/56/57/58

Author: *John Haechten*
Microchip Technology, Inc.

1.0 INTRODUCTION

This application note provides information on how to configure events using the API for the VSC8258/57/56/54 PHY Hardware. This family of devices consists of the following PHYs:

- VSC8254 – Dual Port, 10 Gbps SFI to XFI PHY with IEEE-1588 and MACSec support
- VSC8256 – Quad Port, 10 Gbps SFI to XFI Repeater
- VSC8257 – Quad Port, 10 Gbps SFI to XFI PHY with IEEE-1588
- VSC8258 – Quad Port, 10 Gbps SFI to XFI PHY with IEEE-1588 and MACSec support

This application note provides a step-by-step procedure essential to configure the PHY Hardware. This document also explains which polling routines and which registers are being configured or polled by the API used with the event processing.

There are two types of events supported by the API for the VSC8254/56/57/58 family namely, channel-level events (interrupts) and aggregated events (interrupts). This application note describes the steps required to propagate an interrupt generated from within a specific PHY Block to a GPIO OUTPUT pin.

1.1 Sections

This application note covers the following topics:

1.2 References

Consult the following documents when using this application note:

- *VSC8254-01 Data Sheet*
- *VSC8256-01 Data Sheet*
- *VSC8257-01 Data Sheet*
- *VSC8258-01 Data Sheet*
- *AN3576 Software for VSC PH Application Note*

Note: The *VSC8257-01 Data Sheet* contains Register Descriptions embedded in Section 4. The user needs to open this document with Adobe Acrobat to access the embedded register definitions. Therefore, if register definitions are needed, the user should download the *VSC8257-01 PHY Data Sheet* and go to Section 4 and click on the paper-clip icon which will pull in the embedded Register definitions if using Adobe Acrobat.

2.0 BLOCK-LEVEL INTERRUPTS

Each block within the PHY has a set of registers dedicated to specific interrupt events generated within that particular block. The individual Interrupt events may be masked at this level or they may be enabled or routed up to the block level within the chip. At the block level, the lower-level interrupts are aggregated together into the two interrupt generators per channel, INTR[0] and INTR[1].

An example of the Individual Interrupts within a particular block can be found for the blocks described below:

- LINE_PMA:PMA_INTR:PMA_INTR_MASK Dev: 0x1, Reg: 0xA203
  - TX_LOL_INTR_EN – bit 2
  - RX_LOL_INTR_EN – bit 1
  - RX_LOS_INTR_EN – bit 0
- LINE_PMA:PMA_INTR:PMA_INTR_STAT Dev: 0x1, Reg: 0xA204
  - TX_LOL_STICKY – bit 2
  - RX_LOL_STICKY – bit 1
  - RX_LOS_STICKY – bit 0
- HOST_PMA:PMA_INTR:PMA_INTR_MASK Dev: 0x9, Reg: 0xA203
- HOST_PMA:PMA_INTR:PMA_INTR_STAT Dev: 0x9, Reg: 0xA204
- LINE_PCS10G:PCS_INTR_PEND1:PCS_INTR_PEND1 Dev: 0x3, Reg: 0x8E00
- LINE_PCS10G:PCS_INTR_MASK1:PCS_INTR_MASK1 Dev: 0x3, Reg: 0x8E01
- LINE_PCS10G:PCS_INTR_STATUS:PCS_INTR_STATUS Dev: 0x3, Reg: 0x8E03
- HOST_PCS10G:PCS_INTR_PEND1:PCS_INTR_PEND1 Dev: 0xB, Reg: 0x8E00
- HOST_PCS10G:PCS_INTR_MASK1:PCS_INTR_MASK1 Dev: 0xB, Reg: 0x8E01
- HOST_PCS10G:PCS_INTR_STATUS:PCS_INTR_STATUS Dev: 0xB, Reg: 0x8E03

There is an exception for the HOST/LINE PCS for 1G mode, as these two blocks do not have dedicated registers for interrupt masking and status. While HOST_PCS1G and LINE_PCS1G do not have a dedicated set of registers for individual interrupts, there are STICKY bits which are used in a similar fashion, but these STICKY bits must be polled, and the signals from within the block do not propagate in the same manner as with the dedicated interrupt block.

These blocks (HOST/LINE 1G PCS) do not have dedicated registers for specific Interrupt events generated within that particular block. In addition, these blocks cannot be used to drive an external event, such as a GPIO pin.

When a particular functional block does have an INTR sub-block of registers within the block, these are considered Hardware INTR as a signal is generated from the Hardware and can be routed to an external GPIO pin. The hardware INTR exceptions, the blocks which do not have an INTR sub-block, are listed below:

- HOST MAC TX MONITOR
- LINE MAC TX MONITOR
- HOST PCS1G
- LINE PCSC1G
- FC BUFFER STATUS

These blocks do not have a dedicated INTR sub-block of registers for interrupt masking and status. These blocks contain STICKY bits which are used in a similar fashion. However, these STICKY bits have different properties and must be polled to obtain the status. In addition, these STICKY bit indicators do not propagate from within the block in the same manner as with the dedicated interrupt sub-block. This results in the inability to aggregate some of the lower-level interrupts together into the two interrupt generators per channel, INTR[0] and INTR[1], which can be used to drive a hardware generated output like a GPIO signal. Only those blocks within the chip that have an interrupt sub-block for the events may be used to drive external signals.

As an example, the LINE_PCS10G block can be used to generate an output on a GPIO pin for a loss-of-link when in 10G mode, and that indicator can be routed to a FPGA/Processor. However, the LINE_PCS1G block does not have this capability in the hardware. Therefore, it cannot generate an output on a GPIO pin for a loss-of-link indicator when in 1G mode which could be routed to an FPGA/Processor. Therefore, the FPGA/processor would need to poll the Link Status STICKY bits in the HOST/LINE PCS blocks when in 1G mode to determine the link status. The reading of the appropriate registers is handled by the API.

3.0 BLOCK-LEVEL INTERRUPT AGGREGATION AND ROUTING

There are two interrupt generators available per channel. These interrupt generators are maskable as there are interrupt enables for each contributing block. When enabled, these interrupt generators (INTR_0 and INTR_1) propagate interrupts from the selected blocks, and these interrupt signals may be propagated through a virtual GPIO channel to a GPIO output pin.

These two interrupt generators are located in the following registers:

- GPIO_INTR_CTRL:GPIO_INTR:INTR[0] Dev: 0x1, Reg: 0xC018
- GPIO_INTR_CTRL:GPIO_INTR:INTR[1] Dev: 0x1, Reg: 0xC019

Corresponding to these two interrupt generators, there is an Interrupt status for each in the following registers:

- GPIO_INTR_CTRL:GPIO_INTR:INTR_STAT[0] Dev: 0x1, Reg: 0xC01A
- GPIO_INTR_CTRL:GPIO_INTR:INTR_STAT[1] Dev: 0x1, Reg: 0xC01B

Each block is represented by a bit within the previously described INTR and INTR_STAT registers. The list of available blocks within the PHY is shown in Table 1.

TABLE 1: BLOCK-LEVEL INTERRUPT OPTIONS

Interrupt	API Designation for Interrupt Generators Aggregated Interrupt: “c_intrpt” Options	Register INTR[0] (Reg: 0xC018) or INTR[1] (Reg: 0xC019) Bit
WIS Interrupt 0	VTSS_10G_GPIO_INTRPT_WIS0	0
WIS Interrupt 1	VTSS_10G_GPIO_INTRPT_WIS1	1
LPCS 10G Interrupt	VTSS_10G_GPIO_INTRPT_LPCS10G	2
HPCS 10G Interrupt	VTSS_10G_GPIO_INTRPT_HPCS10G	3
LPCS 1G Interrupt	VTSS_10G_GPIO_INTRPT_LPCS1G	4
HPCS 1G Interrupt	VTSS_10G_GPIO_INTRPT_HPCS1G	5
MACSEC EGR Interrupt	VTSS_10G_GPIO_INTRPT_MSEC_EGR	6
MACSEC INGR Interrupt	VTSS_10G_GPIO_INTRPT_MSEC_IGR	7
Line MAC Interrupt	VTSS_10G_GPIO_INTRPT_LMAC	8
Host MAC Interrupt	VTSS_10G_GPIO_INTRPT_HMAC	9
FC Buffer Interrupt	VTSS_10G_GPIO_INTRPT_FCBUF	10
Line INGR FIFO Interrupt	VTSS_10G_GPIO_INTRPT_LIGR_FIFO	11
Line EGR FIFO Interrupt	VTSS_10G_GPIO_INTRPT_LEGR_FIFO	12
Host EGR FIFO Interrupt	VTSS_10G_GPIO_INTRPT_HEGR_FIFO	13
Line PMA Interrupt	VTSS_10G_GPIO_INTRPT_LPMA	14
Host PMA Interrupt	VTSS_10G_GPIO_INTRPT_HPMA	15

4.0 VIRTUAL GPIO CHANNELS

There are eight virtual GPIO Channels that have been dedicated by the API per PHY port. Refer to the GPIO section of the data sheet for the recommended GPIO layout for the board, which matches the layout shown in Table 2.

GPIOs can be set for input or output. When configured as an output, a set of eight multiplexers in each channel of the PHY selects the per-channel output function routed to each of the eight per-channel virtual GPIO Outputs. A second level of multiplexing occurs at the GPIO pin itself where the individual per-channel virtual outputs as well as the chip-level output functions are associated to a particular GPIO output. When configuring GPIOs for output, many times they are propagating an interrupt signal from within a block in the PHY. The Virtual GPIO Output Channel column in Table 2 indicates the recommended mapping for GPIOx_OUT functions.

TABLE 2: MAPPING GPIO AND VIRTUAL GPIO OUTPUTS

Channel Number	GPIO Number	DS Recommended GPIO Configuration	Virtual GPIO Output Channels GPIO_INTR_CTRL:GPIO_INTR:GPIOx_OUT Dev: 0x1, Reg: 0xC010 -> 0xC017
0	0	CH0_RATESEL0	GPIO0_OUT
0	1	CH0_MOD_ABS	GPIO1_OUT
0	2	CH0_I2C_MST_SCL	GPIO2_OUT
0	3	CH0_I2C_MST_SDA	GPIO3_OUT
0	4	CH0_TX_DIS	GPIO4_OUT
0	5	CH0_TX_FAULT	GPIO5_OUT
0	6	CH0_RX_LOS	GPIO6_OUT
0	7	CH0_LINK_UP	GPIO7_OUT
1	8	CH1_RATESEL0	GPIO0_OUT
1	9	CH1_MOD_ABS	GPIO1_OUT
1	10	CH1_I2C_MST_SCL	GPIO2_OUT
1	11	CH1_I2C_MST_SDA	GPIO3_OUT
1	12	CH1_TX_DIS	GPIO4_OUT
1	13	CH1_TX_FAULT	GPIO5_OUT
1	14	CH1_RX_LOS	GPIO6_OUT
1	15	CH1_LINK_UP	GPIO7_OUT
2	16	CH2_RATESEL0	GPIO0_OUT
2	17	CH2_MOD_ABS	GPIO1_OUT
2	18	CH2_I2C_MST_SCL	GPIO2_OUT
2	19	CH2_I2C_MST_SDA	GPIO3_OUT
2	20	CH2_TX_DIS	GPIO4_OUT
2	21	CH2_TX_FAULT	GPIO5_OUT
2	22	CH2_RX_LOS	GPIO6_OUT
2	23	CH2_LINK_UP	GPIO7_OUT
3	24	CH3_RATESEL0	GPIO0_OUT
3	25	CH3_MOD_ABS	GPIO1_OUT
3	26	CH3_I2C_MST_SCL	GPIO2_OUT
3	27	CH3_I2C_MST_SDA	GPIO3_OUT
3	28	CH3_TX_DIS	GPIO4_OUT
3	29	CH3_TX_FAULT	GPIO5_OUT
3	30	CH3_RX_LOS	GPIO6_OUT
3	31	CH3_LINK_UP	GPIO7_OUT
—	32	I2C_SLAVE_SDA	—
—	33	I2C_SLAVE_SCL	—

TABLE 2: MAPPING GPIO AND VIRTUAL GPIO OUTPUTS (CONTINUED)

Channel Number	GPIO Number	DS Recommended GPIO Configuration	Virtual GPIO Output Channels GPIO_INTR_CTRL:GPIO_INTR:GPIOx_OUT Dev: 0x1, Reg: 0xC010 -> 0xC017
—	34	INTR_A	—
—	35	INTR_B	—
—	36	CH0_ACTIVITY	—
—	37	CH1_ACTIVITY	—
—	38	CH2_ACTIVITY	—
—	39	CH3_ACTIVITY	—

There are eight GPIOs dedicated to each channel (Port) of the PHY chip and the software API is written to accommodate the mapping of Channel_no to GPIO_no for each channel as shown in Table 2.

It is not recommended to use GPIO pins 32 and above for Virtual Channel GPIOs as these have been mapped within the API. It is a possible but not a recommended configuration since special sequences may be required to overwrite the API register configuration.

There are eight virtual GPIO outputs per channel. These virtual GPIOs are described in Table 2 and are designated as GPIO_INTR_CTRL:GPIO_INTR:GPIOx_OUT, where x=0->7. The GPIOx_OUT can be configured by selecting the internal signal source (Option for SEL0) to be routed to the selected GPIOx_OUT line, that is, the interrupting signal from the early part of this section located in the following registers:

- GPIO_INTR_CTRL:GPIO_INTR:GPIO0_OUT.SEL0 Dev: 0x1, Reg: 0xC010
- GPIO_INTR_CTRL:GPIO_INTR:GPIO1_OUT.SEL0 Dev: 0x1, Reg: 0xC011
- ….
- GPIO_INTR_CTRL:GPIO_INTR:GPIO7_OUT.SEL0 Dev: 0x1, Reg: 0xC017

5.0 GPIO OUTPUT PINS

There are 40 GPIO pins which have configurable functionality. Any function can be mapped to any GPIO pin. The only restriction is that only eight of the per-channel GPIO output functions may be used at any one time. However, there are recommendations of how the GPIOs are configured based upon the API implementation.

For the specific case of propagating interrupts to a GPIO **OUTPUT** pin, the previous paragraphs have described the following:

- Individual interrupts are enabled (Example: PMA RX_LOL).
- The individual interrupt signal is aggregated into a block-level interrupt (Host/Line PMA Block, Host/Line PCS10G Block, and so on.)
- The block-level interrupt is routed to the two interrupt generators per channel, INTR[0], INTR[1].
- The interrupt generator is associated to a virtual GPIO output (GPIOx_OUT).
- The virtual GPIO is connected to a designated GPIO output pin for signal output.

This section details the steps required to connect a virtual GPIO output to the physical GPIO pin. The 40 GPIO pins available are shown in Table 2. There is a recommended mapping of the GPIO pins, and the suggested mapping is derived to optimize the usage of the eight virtual GPIO outputs per PHY channel. These virtual GPIOs are described above and designated as GPIO_INTR_CTRL:GPIO_INTR:GPIOx_OUT, where x=0->7. The GPIOx_OUT can be configured by selecting the internal signal source (Option Selection for SEL0) to be routed to the selected GPIOx_OUT line. The GPIOx_OUT Virtual GPIO is then connected to a GPIO pin which has been configured for OUTPUT, with the Input Signal being the Virtual GPIO.

This is accomplished by setting the proper GPIO bit designation to enable the GPIO output in one of the following registers:

- GPIO_CTRL:GPIO_CFG_STAT:GPIO_CFG1 Dev: 0x1E, Reg: 0xF200
  [Enable bits for GPIO 39:32] GPIO33 = bit 1 set
- GPIO_CTRL:GPIO_CFG_STAT:GPIO_CFG0 Dev: 0x1E, Reg: 0xF201
  [Enable bits for GPIO 31:0] GPIO7 = bit 7 set

Once the GPIO pin has been enabled for output, it must be connected to a source, which in this case will be the virtual GPIO output configured previously.

Each GPIO pin has a dedicated register associated to it for the selection of the source for the GPIO output.

- GPIO_CTRL:GPIO_CFG_STAT:GPIO_OUT_CFG_0 Dev: 0x1E, Reg: 0xF20C
- GPIO_CTRL:GPIO_CFG_STAT:GPIO_OUT_CFG_1 Dev: 0x1E, Reg: 0xF20D
- ….
- GPIO_CTRL:GPIO_CFG_STAT:GPIO_OUT_CFG_39 Dev: 0x1E, Reg: 0xF233

The software API has functions to configure the GPIO pins for different settings. This can be accomplished by calling API: vtss_phy_10g_gpio_mode_set w/gpio_mode structure configured as shown later in this document. As an example, to configure a GPIO for the OUTPUT mode, the gpio_mode.mode = VTSS_10G_PHY_GPIO_OUT.

Note that the LOPC default functionality in the chip has defaulted to GPIO 34->37:

- CH0: GPIO 34 GPIO_CTRL:GPIO_CFG_STAT:CH0_LINE_LOPC_CFG
  Dev: 0x1E, Reg: 0xF234 Default Value: 34
- CH1: GPIO 35 GPIO_CTRL:GPIO_CFG_STAT:CH1_LINE_LOPC_CFG
  Dev: 0x1E, Reg: 0xF235 Default Value: 35
- CH2: GPIO 36 GPIO_CTRL:GPIO_CFG_STAT:CH2_LINE_LOPC_CFG
  Dev: 0x1E, Reg: 0xF236 Default Value: 36
- CH3: GPIO 37 GPIO_CTRL:GPIO_CFG_STAT:CH3_LINE_LOPC_CFG
  Dev: 0x1E, Reg: 0xF237 Default Value: 37

Therefore, if GPIO 34->37 are intended to be used for other functionality than LOPC, the application should change the above registers and select a value of 63 as default.

6.0 AGGREGATED EVENTS

There are four aggregated events available per PHY.

1. VTSS_10G_PHY_GPIO_AGG_INT_0
2. VTSS_10G_PHY_GPIO_AGG_INT_1
3. VTSS_10G_PHY_GPIO_AGG_INT_2
4. VTSS_10G_PHY_GPIO_AGG_INT_3

The API was designed to process or use the VTSS_10G_PHY_GPIO_AGG_INT_0 for events set by the API. The application developer needs to keep this in mind, especially for the 10G Link events, because these events are polled by the API on VTSS_10G_PHY_GPIO_AGG_INT_0 and not on VTSS_10G_PHY_GPIO_AGG_INT_1, VTSS_10G_PHY_GPIO_AGG_INT_2, VTSS_10G_PHY_GPIO_AGG_INT_3. Furthermore, there are two interrupt generators (INTR[0] and INTR[1]) and many times the API only checks one of these, primarily the INTR[1] for 10G events.

There is limited processing for VTSS_10G_PHY_GPIO_AGG_INT_1, VTSS_10G_PHY_GPIO_AGG_INT_2, and VTSS_10G_PHY_GPIO_AGG_INT_3. The extended event and extended2 event polling functions handle LINE PCS1G Block and HOST PCS1G Block for both INTR[0] and INTR[1]. However, the extended event handler only checks for HOST/LINE ANEG Restart Events. Therefore, the general configuration overview of the usage of INTR[0] and INTR[1] by the API is:

- INTR[0] Events are from the WIS Block, Line PCS1G ANEG Restart, and Host PCS1G ANEG Restart.
- INTR[1] Events are from all the other Blocks within the PHY and include Line PCS1G, Host PCS1G.

When enabling the interrupt, ensure that the interrupt event selected will be processed by the polling function. Make sure that the application only selects the proper INT_x (highly likely INTR[1] for most applications) which is processed by the event_polling functions. See Table 3.

TABLE 3: MAPPING AGGREGATED INTR DEFINITIONS TO API SOFTWARE DEFINITIONS

Register INTR[0] or INTR[1]bit	API Designation	Description
0	VTSS_10G_GPIO_AGGR_INTRPT_CH0_INTR0_EN	Aggregated Interrupt Ch0 INTR[0]
1	VTSS_10G_GPIO_AGGR_INTRPT_CH0_INTR1_EN	Aggregated Interrupt Ch0, INTR[1]
2	VTSS_10G_GPIO_AGGR_INTRPT_CH1_INTR0_EN	Aggregated Interrupt Ch1, INTR[0]
3	VTSS_10G_GPIO_AGGR_INTRPT_CH1_INTR1_EN	Aggregated Interrupt Ch1, INTR[1]
4	VTSS_10G_GPIO_AGGR_INTRPT_CH2_INTR0_EN	Aggregated Interrupt Ch2, INTR[0]
5	VTSS_10G_GPIO_AGGR_INTRPT_CH2_INTR1_EN	Aggregated Interrupt Ch2, INTR[1]
6	VTSS_10G_GPIO_AGGR_INTRPT_CH3_INTR0_EN	Aggregated Interrupt Ch3, INTR[0]
7	VTSS_10G_GPIO_AGGR_INTRPT_CH3_INTR1_EN	Aggregated Interrupt Ch3, INTR[1]
8	VTSS_10G_GPIO_AGGR_INTRPT_IP1588_0_INTR0_EN	Aggregated Interrupt 1588 Ch0, INTR0
9	VTSS_10G_GPIO_AGGR_INTRPT_IP1588_0_INTR1_EN	Aggregated Interrupt 1588 Ch0, INTR1
10	VTSS_10G_GPIO_AGGR_INTRPT_IP1588_0_INTR2_EN	Aggregated Interrupt 1588 Ch0, INTR2
11	VTSS_10G_GPIO_AGGR_INTRPT_IP1588_0_INTR3_EN	Aggregated Interrupt 1588 Ch0, INTR3
12	VTSS_10G_GPIO_AGGR_INTRPT_IP1588_1_INTR0_EN	Aggregated Interrupt 1588 Ch1, INTR0
13	VTSS_10G_GPIO_AGGR_INTRPT_IP1588_1_INTR1_EN	Aggregated Interrupt 1588 Ch1, INTR1
14	VTSS_10G_GPIO_AGGR_INTRPT_IP1588_1_INTR2_EN	Aggregated Interrupt 1588 Ch1, INTR2
15	VTSS_10G_GPIO_AGGR_INTRPT_IP1588_1_INTR3_EN	Aggregated Interrupt 1588 Ch1, INTR3
16	VTSS_10G_GPIO_AGGR_INTRPT_LCPLL_0_INTR_EN	Aggregated Interrupt LCPLL_0
17	VTSS_10G_GPIO_AGGR_INTRPT_LCPLL_1_INTR_EN	Aggregated Interrupt LCPLL_1
18	VTSS_10G_GPIO_AGGR_INTRPT_EXP4_INTR_EN	Aggregated Interrupt EXP4
19	VTSS_10G_GPIO_AGGR_INTRPT_CLK_MUX_INTR_EN	Aggregated Interrupt CLK_MUX
20	VTSS_10G_GPIO_AGGR_INTRPT_GPIO_INTR_EN	Aggregated Interrupt GPIO

7.0 SOFTWARE REQUIREMENTS

For the VSC6803 code line, as of the MEPA 2023.12 API code release, the event/interrupts processing described in this document are supported in the MEPA API.

For the VSC6802 code line, as of the latest release of the API 4.67.05 code line, the event/interrupts processing described in this document are not included in the API code, and a software patch is required. API 4.67.05 API with the patch: vtss_api_4_67_05_jira_851_patch_p30.txt is applied.

Perform the following steps to apply the patch:

1. Set the directory to the Top Level vtss_api Directory.
2. Copy the patch file (.txt file) into the directory.
3. Apply the patch using the cmd: patch -p1 < vtss_api_4_67_05_jira_851_patch_p30.txt.
4. Check if the patch applies without errors. If there are errors, contact customer support for guidance.

Customers with software API versions other than API 4.67.05 may contact support and provide the API version in use as the patch file may be different based upon the software version.

8.0 ENABLING EVENTS

There are three different APIs used to enable events. The application developer needs to ensure that the events being enabled correspond to the events that the event handler calls on for processing. The following is a list of events:

- vtss_phy_10g_event_enable_set – See Table 5.
- vtss_phy_10g_extended_event_enable_set – See Table 6.
- vtss_phy_10g_extended2_event_enable_set – See Table 7.

9.0 PROCESSING EVENTS

Due to the large number of events, there are three different APIs used to process events. The application developer needs to keep this in mind when events are being enabled, so that the corresponding event handler is called for the processing of events. Refer to Table 4.

TABLE 4: API FOR EVENT PROCESSING

Event Enablement	Top-level Event Processing API	PHY-specific Event Handler
vtss_phy_10g_event_enable_set	vtss_phy_10g_event_poll	malibu_phy_10g_event_poll
vtss_phy_10g_extend-ed_event_enable_set	vtss_phy_10g_extended_event_poll	malibu_phy_10g_extended_event_poll
vtss_phy_10g_extend-ed2_event_enable_set	vtss_phy_10g_extended2_event_poll	malibu_phy_10g_extended2_event_poll

9.1 Steps for Processing Events

9.1.1 API: VTSS_PHY_10G_EVENT_POLL

For API vtss_phy_10g_event_poll, the following events are processed by malibu_phy_10g_event_poll:

1. Reading the INTR[0] indicator for each channel from GPIO_CTRL (Device 0x1E global registers):INTR_CFG_STAT:INTR_STAT[0-3] Dev: 0x1E, Reg: 0xF260-0xF264. Checking the following:
   - CH0_INTR0_STAT – Bit 0
   - CH1_INTR0_STAT – Bit 2
   - CH2_INTR0_STAT – Bit 4
   - CH3_INTR0_STAT – Bit 6
   - GPIO_INTR_STAT – Bit 20
2. Once interrupting channel is determined, it reads the Interrupt status from the Interrupt Generator INTR[0], Blocks WIS0 and WIS1:
   - INTR[0] and INTR[1] are available in the hardware, but the software only processes INTR[0] for blocks WIS0 and WIS1.
   - GPIO_INTR_CTRL:GPIO_INTR:INTR_STAT[0] Dev: 0x1, Reg: 0xC01A
   - bit 0: WIS0
   - bit1: WIS1
   - Checks WIS Block, VTSS_10G_GPIO_INTRPT_WIS0, and VTSS_10G_GPIO_INTRPT_WIS1.
   - Checks GPIO_INTR_STAT; used to propagate interrupts and clears
3. Checking GPIO_INTR_STAT for indications that interrupts were propagated. The following indicators are cleared:
   - GPIO_CTRL:INTR_CFG_STAT:INTR_SRC_EN[0].GPIO_INTR_EN Dev: 0x1E, Reg: 0xF25C
   - GPIO_CTRL:INTR_CFG_STAT:INTR_SRC_EN[1].GPIO_INTR_EN Dev: 0x1E, Reg: 0xF25D
   - GPIO_CTRL:INTR_CFG_STAT:INTR_SRC_EN[2].GPIO_INTR_EN Dev: 0x1E, Reg: 0xF25E
   - GPIO_CTRL:INTR_CFG_STAT:INTR_SRC_EN[3].GPIO_INTR_EN Dev: 0x1E, Reg: 0xF25F

   The WIS0 and WIS1 events are read and processed/cleared and re-enabled for the next event. The PHY Registers which are read for WIS0 and WIS1 Events are:
   - WIS:EWIS_Interrupt_Pending_1:EWIS_INTR_PEND1 Dev: 02, Reg:0xEE00
   - WIS:EWIS_Interrupt_Pending_2:EWIS_INTR_PEND2 Dev: 02, Reg:0xEE04
   - WIS:EWIS_Interrupt_Pending_3:EWIS_INTR_PEND3 Dev: 02, Reg:0xEE08

Therefore, when using the API **vtss_phy_10g_event_poll** which calls **malibu_phy_10g_event_poll**, only the events describe in Table 5 will be processed. These are the only events that should be enabled using **API vtss_-phy_10g_event_enable_set**.

TABLE 5: MAPPING EVENT REGISTERS TO API SOFTWARE DEFINITIONS

Event Register	API Software Event Definition
EWIS_INTR_PEND_1	VTSS_PHY_10G_LINK_LOS_EV
EWIS_INTR_PEND_1	VTSS_PHY_EWIS_SEF_EV
EWIS_INTR_PEND_1	VTSS_PHY_EWIS_SEF_EV
EWIS_INTR_PEND_1	VTSS_PHY_EWIS_FPLM_EV
EWIS_INTR_PEND_1	VTSS_PHY_EWIS_FAIS_EV
EWIS_INTR_PEND_1	VTSS_PHY_EWIS_LOF_EV
EWIS_INTR_PEND_1	VTSS_PHY_EWIS_RDIL_EV
EWIS_INTR_PEND_1	VTSS_PHY_EWIS_AISL_EV
EWIS_INTR_PEND_1	VTSS_PHY_EWIS_LCDP_EV
EWIS_INTR_PEND_1	VTSS_PHY_EWIS_PLMP_EV
EWIS_INTR_PEND_1	VTSS_PHY_EWIS_AISP_EV
EWIS_INTR_PEND_1	VTSS_PHY_EWIS_LOPP_EV
EWIS_INTR_PEND_1	VTSS_PHY_EWIS_UNEQP_EV
EWIS_INTR_PEND_2	VTSS_PHY_EWIS_FEUNEQP_EV
EWIS_INTR_PEND_2	VTSS_PHY_EWIS_FERDIP_EV
EWIS_INTR_PEND_2	VTSS_PHY_EWIS_REIL_EV
EWIS_INTR_PEND_2	VTSS_PHY_EWIS_REIP_EV
EWIS_INTR_PEND_2	VTSS_PHY_EWIS_B1_NZ_EV
EWIS_INTR_PEND_2	VTSS_PHY_EWIS_B2_NZ_EV
EWIS_INTR_PEND_2	VTSS_PHY_EWIS_B3_NZ_EV
EWIS_INTR_PEND_2	VTSS_PHY_EWIS_REIL_NZ_EV
EWIS_INTR_PEND_2	VTSS_PHY_EWIS_REIP_NZ_EV
EWIS_INTR_PEND_3	VTSS_PHY_EWIS_B1_THRESH_EV
EWIS_INTR_PEND_3	VTSS_PHY_EWIS_B2_THRESH_EV
EWIS_INTR_PEND_3	VTSS_PHY_EWIS_B3_THRESH_EV
EWIS_INTR_PEND_3	VTSS_PHY_EWIS_REIL_THRESH_EV
EWIS_INTR_PEND_3	VTSS_PHY_EWIS_REIP_THRESH_EV

9.1.2 API: VTSS_PHY_10G_EXTENDED_EVENT_POLL

For API vtss_phy_10g_extended_event_poll, the following events are processed by malibu_phy_10g_extended_event_poll:

1. Reading VTSS_10G_PHY_GPIO_AGG_INT_0 status registers
2. Determining the state of INTR[1] Indicator for each channel from GPIO_CTRL:INTR_CFG_STAT:INTR_STAT[0-3] Dev: 0x1E, Reg: 0xF260-0xF264, checking the following:
   - CH0_INTR1_STAT – Bit 1
   - CH1_INTR1_STAT – Bit 3
   - CH2_INTR1_STAT – Bit 5
   - CH3_INTR1_STAT – Bit 7
3. Once interrupting channel is determined, it reads the interrupt status from the Interrupt Generator INTR[1], reads the interrupt status from the Interrupt Generator INTR[1] for 10G, and reads INTR[0] and INTR[1] for 1G processing:
   - INTR[0]: GPIO_INTR_CTRL:GPIO_INTR:INTR_STAT[0] Dev: 0x1, Reg: 0xC01A
   - INTR[1]: GPIO_INTR_CTRL:GPIO_INTR:INTR_STAT[1] Dev: 0x1, Reg: 0xC01B

The blocks checked are:

- INTR[1]
  - LINE PCS10G Block – Bit 2: LPCS10G
  - LMAC Block – Bit 8: LMAC
  - HMAC Block – Bit 9: HMAC
  - HOST PMA Block – Bit 15: HPMA
- INTR[0] and INTR[1]:
  - LINE PCS1G Block – Bit 4: LPCS1G
  - HOST PCS1G Block – Bit 5: HPCS1G

4. Reading VTSS_10G_PHY_GPIO_AGG_INT_1, VTSS_10G_PHY_GPIO_AGG_INT_2, and VTSS_10G_-PHY_GPIO_AGG_INT_3 status registers. Reads INTR[0] and INTR[1] for 1G processing:
   - INTR[0]: GPIO_INTR_CTRL:GPIO_INTR:INTR_STAT[0] Dev: 0x1, Reg: 0xC01A
   - INTR[1]: GPIO_INTR_CTRL:GPIO_INTR:INTR_STAT[1] Dev: 0x1, Reg: 0xC01B

The blocks checked are:

- INTR[0] and INTR[1]:
  - LINE PCS1G Block – Bit 4: LPCS1G checks VTSS_PHY_1G_LINE_AUTONEG_RESTART_EV
  - HOST PCS1G Block – Bit 5: HPCS1G checks for VTSS_PHY_1G_HOST_AUTONEG_RESTART_EV

Therefore, when using the API **vtss_phy_10g_extended_event_poll** which calls **malibu_phy_10g_extended_event_poll**, only the events describe in Table 6 will be processed. Therefore, these are the only events that should be enabled using API **vtss_phy_10g_extended_event_enable_set.**

TABLE 6: MAPPING HW INTR BLOCK TO API SOFTWARE EXTENDED EVENTS

Event	INTR Generator	Block	API Software Event Definition
LINE PCS10G Block Events	INTR[1]	LPCS10G	VTSS_PHY_10G_RX_CHAR_DEC_CNT_THRESH_EV
		LPCS10G	VTSS_PHY_10G_TX_CHAR_ENC_CNT_THRESH_EV
		LPCS10G	VTSS_PHY_10G_RX_BLK_DEC_CNT_THRESH_EV
		LPCS10G	VTSS_PHY_10G_TX_BLK_ENC_CNT_THRESH_EV
		LPCS10G	VTSS_PHY_10G_RX_SEQ_CNT_THRESH_EV
		LPCS10G	VTSS_PHY_10G_TX_SEQ_CNT_THRESH_EV
		LPCS10G	VTSS_PHY_10G_FEC_UNFIXED_CNT_THRESH_EV
		LPCS10G	VTSS_PHY_10G_FEC_FIXED_CNT_THRESH_EV
		LPCS10G	VTSS_PHY_10G_HIGHBER_EV
		LPCS10G	VTSS_PHY_10G_RX_LINK_STAT_EV
HOST PMA Block Events	INTR[1]	HPMA	VTSS_PHY_10G_RX_LOS_EV
		HPMA	VTSS_PHY_10G_RX_LOL_EV
		HPMA	VTSS_PHY_10G_TX_LOL_EV
HMAC Events	INTR[1]	HMAC	VTSS_PHY_10G_HOST_MAC_LOCAL_FAULT_EV
		HMAC	VTSS_PHY_10G_HOST_MAC_REMOTE_FAULT_EV
LMAC Events	INTR[1]	LMAC	VTSS_PHY_10G_LINE_MAC_LOCAL_FAULT_EV
		LMAC	VTSS_PHY_10G_LINE_MAC_REMOTE_FAULT_EV
LPCS1G Events	INTR[0 / 1]	LPCS1G	VTSS_PHY_1G_LINE_AUTONEG_RESTART_EV
HPCS1G Events	INTR[0 / 1]	HPCS1G	VTSS_PHY_1G_HOST_AUTONEG_RESTART_EV

Additionally, this API is the only API processes VTSS_10G_PHY_GPIO_AGG_INT_1, VTSS_10G_PHY_GPIO_AGG_INT_2, and VTSS_10G_PHY_GPIO_AGG_INT_3. The event handler processes these interrupts for the following cases:

- LINE PCS1G events on INTR[0] or INTR[1] – VTSS_PHY_1G_LINE_AUTONEG_RESTART_EV
- HOST PCS1G events on INTR[0] or INTR[1] – VTSS_PHY_1G_HOST_AUTONEG_RESTART_E

9.1.3 API: VTSS_PHY_10G_EXTENDED2_EVENT_POLL

The following events are processed by malibu_phy_10g_extended2_event_poll:

1. Reading VTSS_10G_PHY_GPIO_AGG_INT_0 status registers
2. Determining the state of INTR[1] Indicator for each channel from GPIO_CTRL:INTR_CFG_STAT:INTR_STAT[0-3] Dev: 0x1E, Reg: 0xF260-0xF264, checking the following:
   - CH0_INTR1_STAT – Bit 1
   - CH1_INTR1_STAT – Bit 3
   - CH2_INTR1_STAT – Bit 5
   - CH3_INTR1_STAT – Bit 7
   - GPIO_INTR_STAT – Bit 20
3. Once Interrupting channel is determined, it reads the Interrupt status from the Interrupt Generator INTR[1] for 10G:
   - INTR[0]: GPIO_INTR_CTRL:GPIO_INTR:INTR_STAT[0] Dev: 0x1, Reg: 0xC01A
   - INTR[1]: GPIO_INTR_CTRL:GPIO_INTR:INTR_STAT[1] Dev: 0x1, Reg: 0xC01B

Blocks checked are:

- INTR[1]
  - HOST PCS10G Block – Bit 3: HPCS10G
  - LINE PCS1G Block – Bit 4: LPCS1G
  - HOST PCS1G Block – Bit 5: HPCS1G
  - FC BUFF Block – Bit 10: FCBUF
  - LINE INGR FIFO Block – Bit 11: LIGR_FIFO
  - LINE EGR FIFO Block – Bit 12: LEGR_FIFO
  - HOST EGR FIFO Block – Bit 13: HEGR_FIFO
  - LINE PMA Block – Bit 14: LPMA

Therefore, when using the API **vtss_phy_10g_extended2_event_poll** which calls **malibu_phy_10g_extended2_event_poll**, only the events describe in Table 7 will be processed. Therefore, these are the only events that should be enabled using API **vtss_phy_10g_extended2_event_enable_set**.

TABLE 7: MAPPING HW INTR BLOCK TO API SOFTWARE EXTENDED2 EVENTS

Event	INT Generator	Block	API Software Event Definition
HOST PCS10G Events	INTR[1]	HPCS10G	VTSS_PHY_HOST_10G_FEC_UNFIXED_CNT_THRESH_EV
		HPCS10G	VTSS_PHY_HOST_10G_FEC_FIXED_CNT_THRESH_EV
		HPCS10G	VTSS_PHY_HOST_10G_HIGHBER_EV
		HPCS10G	VTSS_PHY_HOST_10G_RX_LINK_STAT_EV
LINE PMA Events	INTR[1]	LPMA	VTSS_PHY_LINE_10G_RX_LOS_EV
		LPMA	VTSS_PHY_LINE_10G_RX_LOL_EV
		LPMA	VTSS_PHY_LINE_10G_TX_LOL_EV
LINE PCS1G Events	INTR[1]	LPCS1G	VTSS_PHY_LINE_1G_XGMII_MASK_LINK_DOWN_MASK
		LPCS1G	VTSS_PHY_LINE_1G_XGMII_MASK_OUT_OF_SYNC_MASK
HOST PCS1G Events	INTR[1]	HPCS1G	VTSS_PHY_HOST_1G_XGMII_MASK_LINK_DOWN_MASK
		HPCS1G	VTSS_PHY_HOST_1G_XGMII_MASK_OUT_OF_SYNC_MASK

TABLE 7: MAPPING HW INTR BLOCK TO API SOFTWARE EXTENDED2 EVENTS (CONTINUED)

Event	INT Generator	Block	API Software Event Definition
FC_BUFF	INTR[1]	FCBUF	VTSS_MAC_FC_BUFFER_STATUS_MASK_XOFF_PAUSE_-GEN_STICKY_MASK
		FCBUF	VTSS_MAC_FC_BUFFER_STATUS_MASK_XON_PAUSE_-GEN_STICKY_MASK
		FCBUF	VTSS_MAC_FC_BUFFER_STATUS_MASK_TX_UNCORRECTED_FRM_DROP_STICKY_MASK
		FCBUF	VTSS_MAC_FC_BUFFER_STATUS_MASK_RX_UNCORRECTED_FRM_DROP_STICKY_MASK
		FCBUF	VTSS_MAC_FC_BUFFER_STATUS_MASK_TX_CTRL_QUEUE_OVERFLOW_DROP_STICKY_MASK
		FCBUF	VTSS_MAC_FC_BUFFER_STATUS_MASK_TX_CTRL_QUEUE_UNDERFLOW_DROP_STICKY_MASK
		FCBUF	VTSS_MAC_FC_BUFFER_STATUS_MASK_TX_DATA_QUEUE_OVERFLOW_DROP_STICKY_MASK
		FCBUF	VTSS_MAC_FC_BUFFER_STATUS_MASK_TX_DATA_QUEUE_UNDERFLOW_DROP_STICKY_MASK
		FCBUF	VTSS_MAC_FC_BUFFER_STATUS_MASK_RX_OVERFLOW_-DROP_STICKY_MASK
		FCBUF	VTSS_MAC_FC_BUFFER_STATUS_MASK_RX_UNDER-FLOW_DROP_STICKY_MASK
LIGR_FIFO, LEG-R_FIFO, HEGR_FIFO	INTR[1]	FIFO_BIST	VTSS_PHY_10G_TX_FIFO_UNDERFLOW_EV
		FIFO_BIST	VTSS_PHY_10G_TX_FIFO_OVERFLOW_EV
		FIFO_BIST	VTSS_PHY_10G_RX_FIFO_UNDERFLOW_EV
		FIFO_BIST	VTSS_PHY_10G_RX_FIFO_OVERFLOW_EV
		FIFO_BIST	VTSS_PHY_10G_TX_FIFO2_UNDERFLOW_EV
		FIFO_BIST	VTSS_PHY_10G_TX_FIFO2_OVERFLOW_EV (Note that interrupts are defined in FIFO_BIST:RATE_COMP for LIGR_FIFO, LEGR_FIFO, HEGR_FIFO)
GPIO_INTR_STAT	—	—	VTSS_PHY_10G_HOST_LOPC_EV (Note that GPIO_INTR_STAT is used to propagate Interrupts.)

10.0 SOFTWARE RESOLUTION

The software API provides APIs to make it easier for the application to enable, disable, and poll for events. There are also APIs for configuring GPIOs such that internal signals can be routed to an external GPIO pin.

There are 3 different APIs used to enable events. The application developer needs to keep this in mind when events are being enabled so that the corresponding event handler will be called for processing of the events.

- vtss_phy_10g_event_enable_set
- vtss_phy_10g_extended_event_enable_set
- vtss_phy_10g_extended2_event_enable_set

To process the interrupts, three (3) different APIs are used to process these events. The mapping to the VSC8254/56/57/58 PHY-specific internal APIs are also shown in Table 4.

- vtss_phy_10g_event_poll
- malibu_phy_10g_event_poll
- vtss_phy_10g_extended_event_poll
- malibu_phy_10g_extended_event_poll
- vtss_phy_10g_extended2_event_poll
- malibu_phy_10g_extended2_event_poll

11.0 CONFIGURING CHANNEL LEVEL EVENTS

The software API, vtss_phy_10g_gpio_mode_set( ), can be used to configure the GPIO functionality.

The structure gpio_mode if of type, vtss_gpio_10g_gpio_mode_t, and is used by the API vtss_phy_10g_gpio_-mode_set( ) to configure the GPIO as an example of configuring a channel level event to drive a GPIO pin output based upon the link status of a particular port. In this case, the PHY would be configured in 10G mode of operation and the line side PCS RX status is routed and used to trigger/generate the GPIO output. Perform the following steps:

1. Clear the structure, initializing it to all zeros.
   Example: memset(&gpio_mode, 0, sizeof(vtss_gpio_10g_gpio_mode_t));
2. Select the GPIO pin to be used.
   Example: gpio_no = 38; // GPIO38 will be configured for Output
3. Start by setting up the gpio_mode to be an Output.
   Example: gpio_mode.mode = VTSS_10G_PHY_GPIO_OUT
4. Choose one of the 8 Virtual GPIO channels available GPIOx_OUT (where x=0->7) to be used for this Output.
   Example: gpio_mode.p_gpio = 4 indicates Virtual GPIO4_OUT
5. Choose whether the signal is inverted.
   Example: gpio_mode.invert_output = TRUE; // 10G Works Inverted, 1G Not Inverted
6. Choose the input Signal which is coming from the particular block:

   Example 1:
   gpio_mode.in_sig = VTSS_10G_GPIO_INTR_SGNL_LINE_PCS_RX_STAT; /** Line PCS RX link status: INV 10G only */

   Example 2:
   gpio_mode.in_sig = VTSS_10G_GPIO_INTR_SGNL_LINE_LINK; /** 10G: Line Link status*/
7. Configure the GPIO by calling API: vtss_phy_10g_gpio_mode_set w/gpio_mode structure configured as shown in steps 1 to 6. Set up the events. Choose the event(s) to be enabled in the Individual Block.
   Example: ex_ev = VTSS_PHY_10G_RX_LOS_EV | VTSS_PHY_10G_RX_LOL_EV |
   VTSS_PHY_10G_RX_LINK_STAT_EV; // Worked for 10G
8. Configure the Interrupts by calling API: vtss_phy_10g_extended_event_enable_set

   rc = vtss_phy_10g_extended_event_enable_set( )
   or

   rc = vtss_phy_10g_extended2_event_enable_set( )

   The board is now configured for link event. When the link event occurs, it will be indicated on the OUTPUT of GPIO38 and Indicated when polling for events.
9. Poll for the events by calling APIs:
   - vtss_phy_10g_extended_event_poll
   - vtss_phy_10g_extended2_event_poll

EXAMPLE 1: CONFIGURATION AND CALLING SEQUENCE FOR 10G OPTICAL SFP

```
memset(&gpio_mode, 0, sizeof(vtss_gpio_10g_gpio_mode_t));
gpio_no = 38;
rc = vtss_phy_10g_extended_event_enable_get( )
gpio_mode.mode = VTSS_10G_PHY_GPIO_OUT;
gpio_mode.p_gpio = 4;  // range 0-7, for gpio_intr:gpio0_out-gpio7_out
gpio_mode.invert_output = TRUE;  // 10G Works Inverted, 1G Not Inverted so this is FALSE for 1G
gpio_mode.in_sig = VTSS_10G_GPIO_INTR_SGNL_LINE_PCS_RX_STAT; /** Line PCS RX link status: INV 10G; not working 1G */
rc = vtss_phy_10g_gpio_mode_set( )
ex_ev = VTSS_PHY_10G_RX_LOS_EV | VTSS_PHY_10G_RX_LOL_EV | VTSS_PHY_10G_RX_LINK_STAT_EV;  // Worked for 10G
rc = vtss_phy_10g_extended2_event_enable_set( )
```

The board is now configured for a 10G Port Link Event.

When the link event occurs, it will be indicated on the OUTPUT of GPIO38 and indicated when polling for events.
rc = vtss_phy_10g_extended2_event_poll

EXAMPLE 2: CONFIGURATION AND CALLING SEQUENCE FOR 1G OPTICAL SFP

```
memset(&gpio_mode, 0, sizeof(vtss_gpio_10g_gpio_mode_t));
gpio_no = 38;
rc = vtss_phy_10g_extended_event_enable_get( )
gpio_mode.mode = VTSS_10G_PHY_GPIO_OUT;
gpio_mode.p_gpio = 4;  // range 0-7, for gpio_intr:gpio0_out-gpio7_out
gpio_mode.invert_output = FALSE;    // 10G Works Inverted, 1G Not Inverted so this is FALSE for 1G
gpio_mode.in_sig = VTSS_10G_GPIO_INTR_SGNL_LINE_LINK;        /** Line Link status used for 1G */
vtss_phy_10g_gpio_mode_set( )

ex_ev = VTSS_PHY_LINE_1G_XGMII_MASK_LINK_DOWN_MASK |
 VTSS_PHY_LINE_1G_XGMII_MASK_OUT_OF_SYNC_MASK|
 VTSS_PHY_HOST_1G_XGMII_MASK_LINK_DOWN_MASK |
 VTSS_PHY_LINE_1G_XGMII_MASK_OUT_OF_SYNC_MASK
rc = vtss_phy_10g_extended2_event_enable_set( )
```

The board is now configured for a 1G Port Link Event.

When the link event occurs, it will be indicated on the OUTPUT of GPIO38 and Indicated when polling for events.

Poll for the events by calling APIs:

- vtss_phy_10g_extended2_event_poll

12.0 CONFIGURING AGGREGATED EVENTS

The software API: vtss_phy_10g_gpio_mode_set( ) can be used to configure the GPIO functionality.

The structure gpio_mode if of type: vtss_gpio_10g_gpio_mode_t and is used by the API vtss_phy_10g_gpio_-mode_set() to configure the GPIO as an example of configuring a aggregated event to drive a GPIO pin output based upon the link status of a particular port. In this case, the PHY would be configured in 10G mode of operation, and the line side PCS RX Status is routed and used to trigger/generate the GPIO output. Perform the following steps:

1. Clear the structure, initializing it to all zeros.
   Example: memset(&gpio_mode, 0, sizeof(vtss_gpio_10g_gpio_mode_t))
2. Select the GPIO pin to be used.
   Example: gpio_no = 38; // GPIO38 will be configured for Output
3. Begin by setting up the gpio_mode to be: VTSS_10G_PHY_GPIO_AGG_INT_0, as the API processes this option.

   Example: gpio_mode.mode = VTSS_10G_PHY_GPIO_AGG_INT_0
4. Choose the Interrupt and the Channel:
   gpio_mode.aggr_intrpt = 0;

   // Set the Aggr INT based upon Ch and the INT_x where x=0,1
   // Note: API only processes INT_1 in the polling function, so setting INT_0 doesn't do anything

   ```
   switch (port_no) {
      case 0:
        gpio_mode.aggr_intrpt |= 1 << VTSS_10G_GPIO_AGGR_INTRPT_CH0_INTR1_EN;   // Ch0: INT_1:
        break;
      case 1:
        gpio_mode.aggr_intrpt |= 1 << VTSS_10G_GPIO_AGGR_INTRPT_CH1_INTR1_EN; // Ch1: INT_1:
        break;
     case 2:
        gpio_mode.aggr_intrpt |= 1 << VTSS_10G_GPIO_AGGR_INTRPT_CH2_INTR1_EN; // Ch2: INT_1:
        break;
      case 3:
        gpio_mode.aggr_intrpt |= 1 << VTSS_10G_GPIO_AGGR_INTRPT_CH3_INTR1_EN; // Ch3: INT_1:
         break;
    default:
    }
   ```
5. Choose the Channel Level Block INTR where the interrupt is being generated or originated

   Example: gpio_conf.c_intrpt =VTSS_10G_GPIO_INTRPT_LPCS10G;
6. Configure the GPIO by calling API: vtss_phy_10g_gpio_mode_set w/gpio_mode structure configure as shown above
   vtss_phy_10g_gpio_mode_set( )
7. Set up the events - Choose the event(s) to be enabled in the Individual Block.
   ```
   extEvent = VTSS_PHY_LINE_1G_XGMII_MASK_LINK_DOWN_MASK |
              VTSS_PHY_LINE_1G_XGMII_MASK_OUT_OF_SYNC_MASK;
   ```
8. Configure the interrupt by calling API: vtss_phy_10g_extended2_event_enable_set
   rc = vtss_phy_10g_extended2_event_enable_set( )

EXAMPLE 3: EXAMPLE_3: CONFIGURATION AND CALLING SEQUENCE FOR 1G OPTICAL SFP – AGGREGATED INTS

```
memset(&gpio_mode, 0, sizeof(vtss_gpio_10g_gpio_mode_t));
gpio_no = 38;
rc = vtss_phy_10g_extended2_event_enable_get( )
gpio_mode.mode = VTSS_10G_PHY_GPIO_AGG_INT_0;
gpio_mode.aggr_intrpt = 0;
gpio_mode.aggr_intrpt |= 1 << VTSS_10G_GPIO_AGGR_INTRPT_CH0_INTR1_EN;   // Ch0: INT_1:
gpio_mode.aggr_intrpt |= 1 << VTSS_10G_GPIO_AGGR_INTRPT_CH1_INTR1_EN;   // Ch1: INT_1:
gpio_mode.aggr_intrpt |= 1 << VTSS_10G_GPIO_AGGR_INTRPT_CH2_INTR1_EN;   // Ch2: INT_1:
gpio_mode.aggr_intrpt |= 1 << VTSS_10G_GPIO_AGGR_INTRPT_CH3_INTR1_EN;   // Ch3: INT_1:
gpio_conf.c_intrpt =VTSS_10G_GPIO_INTRPT_LPCS1G;

vtss_phy_10g_gpio_mode_set( )

ex_ev = VTSS_PHY_LINE_1G_XGMII_MASK_LINK_DOWN_MASK | VTSS_PHY_LINE_1G_XG-
MII_MASK_OUT_OF_SYNC_MASK | VTSS_PHY_HOST_1G_XGMII_MASK_LINK_DOWN_MASK  VTSS_-
PHY_LINE_1G_XGMII_MASK_OUT_OF_SYNC_MASK;

rc = vtss_phy_10g_extended2_event_enable_set( )
```

The board is now configured for link event.

When the link event occurs, it will be indicated on the OUTPUT of GPIO38 and shown when polling for events.

Poll for the events by calling APIs:

```
vtss_phy_10g_extended2_event_poll
```

EXAMPLE 4: CONFIGURATION AND CALLING SEQUENCE FOR 1G CUSFP – AGGREGATED INTS

```
memset(&gpio_mode, 0, sizeof(vtss_gpio_10g_gpio_mode_t));
gpio_no = 38;
rc = vtss_phy_10g_extended_event_enable_get( )
gpio_mode.mode = VTSS_10G_PHY_GPIO_AGG_INT_0;
gpio_mode.aggr_intrpt = 0;
gpio_mode.aggr_intrpt |= 1 << VTSS_10G_GPIO_AGGR_INTRPT_CH0_INTR1_EN;   // Ch0: INT_1:
gpio_mode.aggr_intrpt |= 1 << VTSS_10G_GPIO_AGGR_INTRPT_CH1_INTR1_EN;   // Ch1: INT_1:
gpio_mode.aggr_intrpt |= 1 << VTSS_10G_GPIO_AGGR_INTRPT_CH2_INTR1_EN;   // Ch2: INT_1:
gpio_mode.aggr_intrpt |= 1 << VTSS_10G_GPIO_AGGR_INTRPT_CH3_INTR1_EN;   // Ch3: INT_1:

gpio_conf.c_intrpt = VTSS_10G_GPIO_INTRPT_LMAC;;

vtss_phy_10g_gpio_mode_set( )

ex_ev = VTSS_PHY_10G_LINE_MAC_LOCAL_FAULT_EV | VTSS_PHY_10G_LINE_MAC_REMOTE_FAULT_EV;
rc = vtss_phy_10g_extended_event_enable_set( )
```

The board is now configured. A link event on the CuSFP should generate a Fault which would propagate back to the VSC PHY. The LINE MAC should capture this event.

When the link event occurs, it will be indicated on the OUTPUT of GPIO38 and shown when polling for events.

Poll for the Events by calling APIs:
vtss_phy_10g_extended_event_poll

APPENDIX A: REVISION HISTORY

TABLE A-1: REVISION HISTORY

Revision Level & Date	Section/Figure/Entry	Correction
DS00005760A (02-14-25)	Initial release	

NOTES:

Microchip Information

Trademarks

The “Microchip” name and logo, the “M” logo, and other names, logos, and brands are registered and unregistered trademarks of Microchip Technology Incorporated or its affiliates and/or subsidiaries in the United States and/or other countries (“Microchip Trademarks”). Information regarding Microchip Trademarks can be found at https://www.microchip.com/en-us/about/legal-information/microchip-trademarks.

ISBN: 979-8-3371-0675-5

Legal Notice

This publication and the information herein may be used only with Microchip products, including to design, test, and integrate Microchip products with your application. Use of this information in any other manner violates these terms. Information regarding device applications is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. Contact your local Microchip sales office for additional support or, obtain additional support at www.microchip.com/en-us/support/design-help/client-support-services.

THIS INFORMATION IS PROVIDED BY MICROCHIP "AS IS". MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE, OR WARRANTIES RELATED TO ITS CONDITION, QUALITY, OR PERFORMANCE.

IN NO EVENT WILL MICROCHIP BE LIABLE FOR ANY INDIRECT, SPECIAL, PUNITIVE, INCIDENTAL, OR CONSEQUENTIAL LOSS, DAMAGE, COST, OR EXPENSE OF ANY KIND WHATSOEVER RELATED TO THE INFORMATION OR ITS USE, HOWEVER CAUSED, EVEN IF MICROCHIP HAS BEEN ADVISED OF THE POSSIBILITY OR THE DAMAGES ARE FORESEEABLE. TO THE FULLEST EXTENT ALLOWED BY LAW, MICROCHIP'S TOTAL LIABILITY ON ALL CLAIMS IN ANY WAY RELATED TO THE INFORMATION OR ITS USE WILL NOT EXCEED THE AMOUNT OF FEES, IF ANY, THAT YOU HAVE PAID DIRECTLY TO MICROCHIP FOR THE INFORMATION.

Use of Microchip devices in life support and/or safety applications is entirely at the buyer's risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights unless otherwise stated.

Microchip Devices Code Protection Feature

Note the following details of the code protection feature on Microchip products:

- Microchip products meet the specifications contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is secure when used in the intended manner, within operating specifications, and under normal conditions.
- Microchip values and aggressively protects its intellectual property rights. Attempts to breach the code protection features of Microchip product is strictly prohibited and may violate the Digital Millennium Copyright Act.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of its code. Code protection does not mean that we are guaranteeing the product is “unbreakable”. Code protection is constantly evolving. Microchip is committed to continuously improving the code protection features of our products.