March 4, 2021

Document Number APN00097

Revision A

****This Application Note consists of the same content as the Application Note originally published by Inphi Corporation on its website on March 4, 2021****

Revision History

Version	Date	Comment
A	March 4, 2021	Initial release

Contents

Figures

1. Overhead Access (OHA) Interface

1.1 Top Level Overview

Figure 1: INDx400 Top-level Block Diagram

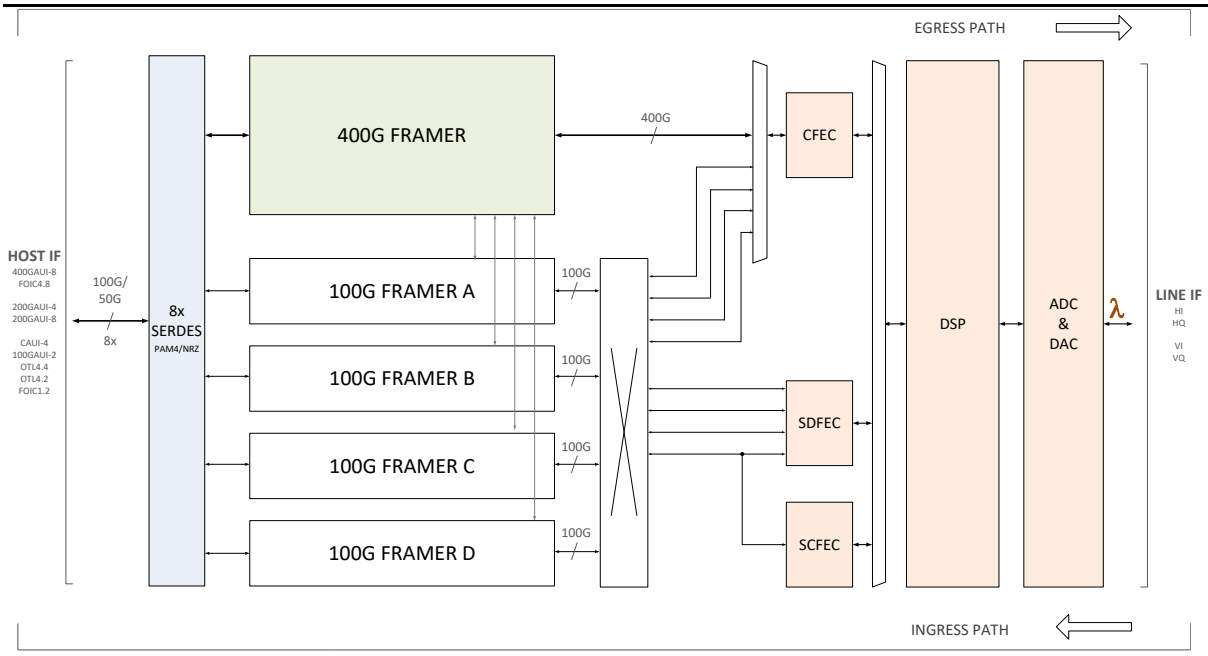
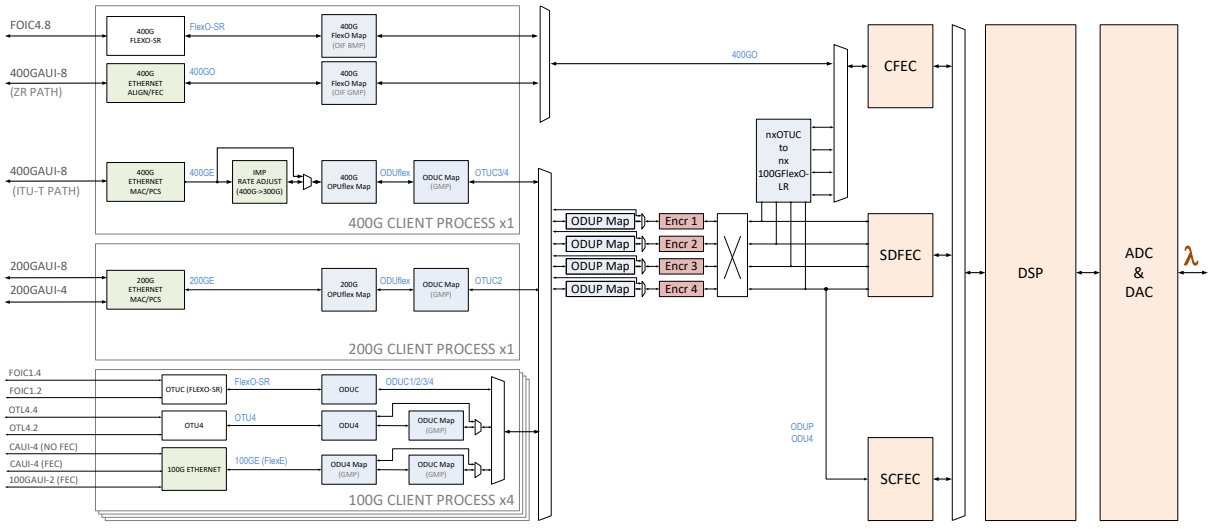


Figure 2: Client Mapping Modes



Figures 1 and 2 show high level INDx400 block diagram and client mapping modes INDx400 supports, respectively. For details, please see INDx400 Datasheet.

The Overhead Access interface (OHA) provides access – extraction and insertion – for both OTN and FlexE overhead bytes through two identical OHA, SGMII packet-based interfaces. See Figure 3.

The OHA “SA / SS” (**S**ignal **A**ggregator and **S**ignal **S**plitter) block connects physically to the four INDx400 100G framers and also to the 400G framer. All framers include “DROP” and “ADD” points in the data-path where overhead at the various framing levels can be dropped or added (overwritten). As depicted in Figure 3., this overhead can be dropped/added from/to OxUflex, OxU, OxUC, FlexO or FlexE streams depending on the mapping structure.

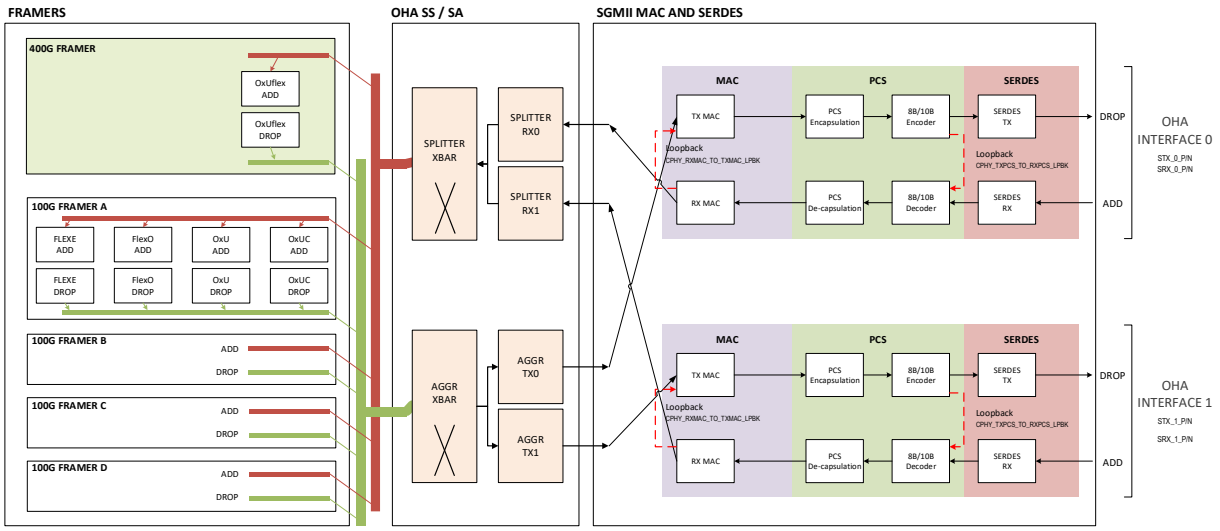
The OHA SGMII Aggregator (OHA SA), aggregates the overhead information from the ‘OHA Drop’ blocks in the framers and sends it towards the two SGMII MAC TX devices and the Transmit/Drop interfaces.

The OHA SGMII Splitter (OHA SS), receives overhead data from the two RX MACs (connected to the Receive/Add Interfaces) and distributes it to the appropriate ‘OHA Add’ blocks for insertion into the respective frame overhead.

Programmable crossbars at both the Splitter and the Aggregator allow for flexible allocation of overhead bandwidth between the two SGMII interfaces.

The PCS blocks encapsulate the MAC frames using standard 8B/10G PCS encoding. Finally, the Tx SerDes serializes the PCS encoded stream towards the DROP (output) interface. Identical but reverse operations are performed in the opposite (ADD) direction. As illustrated in Figure 1 two loopbacks are available for test and diagnostics purposes.

Figure 3: OHA Interface Top Level Connections



The two OHA SGMII interface ports can operate in two distinct modes: A reduced speed mode supporting up to 16 OTN overhead bytes per interface, or a full-speed mode that can carry up to 58 OTN overhead bytes per interface. The reduced speed mode requires an SGMII/1000BASE-X operating at 1.25Gb/s (1G Ethernet data rate), whereas the full-speed mode requires a 3.125Gb/s SGMII/2500BASE-X interface (2.5G Ethernet data rate). The serial (SerDes) rates of 1.25Gb/s and 3.125 Gb/s include the 8b/10b encoding overhead.

Each OHA SGMII interface can add/drop a configurable number of Over Head Block Unit (OHBU)s (1~12 OHBUs)

- 4 OHBUs from 400GE framer in 1x400G mode

- Or 3 OHBUs from each 100G framer (Max 4 framers x 3 OHBU = 12 OHBUs).

By using 2 SGMII interfaces, user can add/drop up to 12 total OHBUs (maximum accessible OHBUs remain same as using 1 SGMII interface) but overhead data for add/drop will be double compared to using 1 SGMII interface.

Each OHBU is independent so each OHBU can deliver different OH bytes of multiple framers (400GE framer in 1x400G mode OR 4x100G framers).

At 3.125 Gbps speed, each OHA SGMII interface can deliver up to total 256 bytes including headers.

At 1.25 Gbps speed, each OHA SGMII interface can deliver up to total 64 bytes including headers.

1.2 The OHA Port Drop Interface

1.2.1 OTU Drop Port

With OTN clients, the overhead access interface exports the selected OxU Frame Overhead Fields to the SGMII/1000BASE-X SS as soon as they become available. The number of OH bytes transported by the Drop OH block units (D-OHBU) can be selected from 1 to 58. Each byte exported can carry any of the 64 OH bytes. The D-OHBU includes

- Port ID (ID)
- Block size (SZ). This refers only to the number of OH bytes and excludes control bytes.
- Processing reference byte (RF). This byte is provided by the Add port and indicates the current MFAS being processed.
- Add port Grant (AG) byte. This number reports the number of buffers used in the Add port as explained in the OHA Flow Control section.

1.2.2 FlexE Drop Port

In the case of FlexE clients, the overhead access interface exports always 66 bytes of FlexE overhead fields to the SGMII SS as soon as they become available. The D-OHBU includes

- Port ID (ID)
- Block size (SZ). This refers only to the number of OH bytes and excludes control bytes.
- Processing Reference (RF) byte for compatibility. This byte is ignored by OHA SS.
- Add port Grant (AG) byte for compatibility. This number reports the number of buffers used in the Add port as explained in the OHA Flow Control section.

After adding all the OH bytes and the control bytes, the supported mode is 70B D-OHDUs. The interface drops only one single FlexE PHY overhead (66bit x 8 rows) along with each Ethernet frame (there can be up to 4 FlexE PHYs in INDx400). Within OH DATA, FlexE overhead row 1 is transmitted first and row 8 last.

1.2.3 FlexO Drop Port

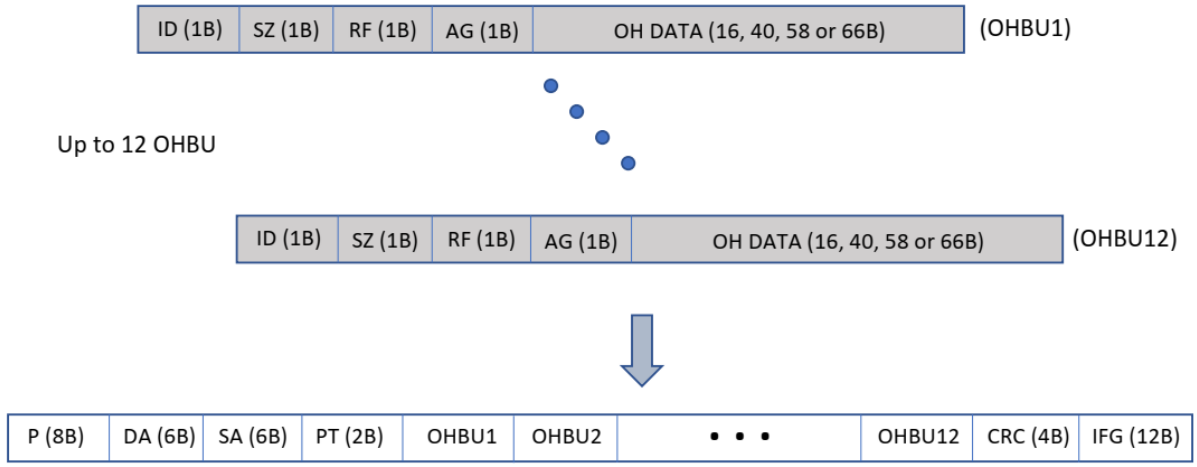
In the case of FlexO clients, the overhead access interface exports selected FlexO overhead fields to the OHA SS as soon as they become available. The number of OH bytes transported by the D-OHBU can be selected from 1 to 40. Each byte exported can carry any of the 40 OH bytes. The D-OHBU includes

- Port ID (ID)
- Block size (SZ). This refers only to the number of OH bytes, excludes control bytes.

- Processing Reference byte (RF). This byte is provided by the Add port and indicates the current MFAS being processed.
- Add port Grant byte (AG). This number reports the number of buffers used in the Add port as explained in the OHA Flow Control section.

After adding all the OH bytes and the control bytes, the supported mode is 44B D-OHBUs.

Figure 4: Drop OHA Interface Structure



As shown in Figure 4, multiple D-OHB data blocks are encapsulated into the data portion of a standard Ethernet frame (Preamble, SA, DA, PT, [OHBs = DATA], CRC, IFG). Each Ethernet frame carries 1~12 overhead data blocks covering all supporting framers (400GE, 4x100G framers).

- 4 OHBUs from 400GE framer
- Or 3 OHUs from each 100G framer (total 12 OHBUs from 4x100G framers)

Occasionally, there will be frames where one of the OHBs is invalid and the device connected to the SGMII Interface (typically an FPGA) must discard it. A special ID code (0xAA) identifies invalid OHB blocks.

1.2.4 Port ID

Port ID is NOT static and depends on number of framers enabled, mapping level configuration or client signal type. Each 100G framer is assigned to max 3 IDs. User can enable min 1 and max 3 framers (e.g. **HOST/MAP1/MAP2**, **MAP2/LINE ……** or **LINE**).

nx100GE (e.g. 2x100GE, 4x100GE)

- When "**map_level**" in "SetOtnOhaConfig" API = **HOST**
  - HOST framer is NOT accessible.
- When "**map_level**" in "SetOtnOhaConfig" API = **MAP1**
  - 100G MAP1 framer A, B, C, D to get port ID = 1, 4, 7, 10 respectively

- When "**map_level**" in "SetOtnOhaConfig" API = **MAP2**
  - 100G MAP2 framer A, B, C, D to get port ID = 2, 5, 8, 11 respectively
- When "**map_level**" in "SetOtnOhaConfig" API = **LINE**
  - 100G LINE framer A, B, C, D to get port ID = 3, 6, 9, 12 respectively

nxOTU4 (e.g. 2xOTU4, 4xOTU4)

100G "**HOST**" framer A, B, C, D will always have 1,4,7,10, respectively.
100G "**LINE**" framer A, B, C, D will always have 3,6,9,12, respectively.
100G "**MAP1**" or "**MAP2**" ID depends on how many framers are enabled or "HOST" or "LINE" framer is used.

IF "**LINE**" framer is not enabled (this is base default ID),
- 100G "**HOST**" framer A, B, C, D will have 1,4,7,10, respectively.
- 100G "**MAP1**" framer A, B, C, D will have 2,5,8,11, respectively.
- 100G "**MAP2**" framer A, B, C, D will have 3,6,9,12, respectively.

IF "**HOST**" framer is not enabled,
- 100G "**MAP1**" framer A, B, C, D will have 1,4,7,10, respectively.
- 100G "**MAP2**" framer A, B, C, D will have 2,5,8,11, respectively.
- 100G "**LINE**" framer A, B, C, D will have 3,6,9,12, respectively.

case 1 (IF HOST, MAP1, MAP2 are enabled)
- When "**map_level**" in "SetOtnOhaConfig" API = **HOST**
  - 100G HOST framer A, B, C, D to get port ID = 1, 4, 7, 10 respectively
- When "**map_level**" in "SetOtnOhaConfig" API = **MAP1**
  - 100G MAP1 framer A, B, C, D to get port ID = 2, 5, 8, 11 respectively
- When "**map_level**" in "SetOtnOhaConfig" API = **MAP2**
  - 100G MAP2 framer A, B, C, D to get port ID = 3, 6, 9, 12 respectively

case 2 (IF MAP1, MAP2, LINE are enabled)
- When "**map_level**" in "SetOtnOhaConfig" API = **MAP1**
  - 100G MAP1 framer A, B, C, D to get port ID = 1, 4, 7, 10 respectively
- When "**map_level**" in "SetOtnOhaConfig" API = **MAP2**
  - 100G MAP2 framer A, B, C, D to get port ID = 2, 5, 8, 11 respectively
- When "**map_level**" in "SetOtnOhaConfig" API = **LINE**
  - 100G LINE framer A, B, C, D to get port ID = 3, 6, 9, 12 respectively

case 3 (IF HOST, MAP2, LINE are enabled)
- When "**map_level**" in "SetOtnOhaConfig" API = **HOST**
  - 100G HOST framer A, B, C, D to get port ID = 1, 4, 7, 10 respectively
- When "**map_level**" in "SetOtnOhaConfig" API = **MAP2**
  - 100G MAP2 framer A, B, C, D to get port ID = 2, 5, 8, 11 respectively
- When "**map_level**" in "SetOtnOhaConfig" API = **LINE**
  - 100G LINE framer A, B, C, D to get port ID = 3, 6, 9, 12 respectively

case 4 (IF HOST, LINE are enabled)
- When "**map_level**" in "SetOtnOhaConfig" API = **HOST**
  - 100G HOST framer A, B, C, D to get port ID = 1, 4, 7, 10 respectively
- When "**map_level**" in "SetOtnOhaConfig" API = **LINE**

- 100G LINE framer A, B, C, D to get port ID = 3, 6, 9, 12 respectively

case 5 (IF MAP2, LINE are enabled)

- When "**map_level**" in "SetOtnOhaConfig" API = **MAP2**
  - 100G MAP2 framer A, B, C, D to get port ID = 2, 5, 8, 11 respectively
- When "**map_level**" in "SetOtnOhaConfig" API = **LINE**
  - 100G LINE framer A, B, C, D to get port ID = 3, 6, 9, 12 respectively

case 6 (IF LINE is enabled)

- When "**map_level**" in "SetOtnOhaConfig" API = **LINE**
  - 100G LINE framer A, B, C, D to get port ID = 3, 6, 9, 12 respectively

FlexO

- FlexO framer will get following port ID
  - 100G FlexO framer A, B, C, D to get port ID = 1, 4, 7, 10 respectively

1x400G

- For 400G framer, only LINE framer is used. LINE framer is assigned to port ID 13.

1.3 OHA Port Add Interface

Figure 5 shows ADD OHBU structure.

1.3.1 OTU Add Port

With OTN clients, the Add port Overhead Access interface imports from 1 up to 64 OTU Frame Overhead Fields from an external device. The bytes are imported in Add OH block units (A-OHBU), which are different from the D-OHBU. Each byte imported can replace any of the 64 OTN OH bytes.

The A-OHBU includes

- port ID (ID)
- size byte (SZ). This refers only to the number of OH bytes and excludes control bytes.
- Insertion reference byte (RF). When reference-driven insertion is enabled, this byte is compared to the current MFAS being processed by the Add port and the OH byte insertion takes place only if the MFAS value equals the reference byte value.
- A 0's reserved byte to match the length of the D-OHBU.

For each of the overhead bytes added by the "OHA Add" block, there are corresponding 8-bit mask registers that enable individual per-bit insertion. The bi-mask register feature allows the user to insert overhead fields with bit level granularity if necessary (e.g. BDI, IAE and STAT field insertion).

1.3.2 FlexE Add Port

In the case of FlexE, clients each OHB Imports 66 FlexE Overhead Fields from an external device. Bytes are imported in Add OH block units (A-OHBU), these are different than the D-OHBU. The number of OH bytes transported by the A-OHBU is a fixed 66B number. The A-OHBU includes

- Port ID (ID)
- Size byte (SZ). This refers only to the number of OH bytes, excludes control bytes.

- Insertion reference byte (RF) for compatibility only. The Add port ignores this byte.
- A 0's reserved byte to match the length of the D-OHBU

After adding all the OH bytes and the Control bytes, the mode supported is 70B. The A-OHBU delivers the 66B to the FlexE OHP source block which will take care of inserting them into the FlexE Frame OH.

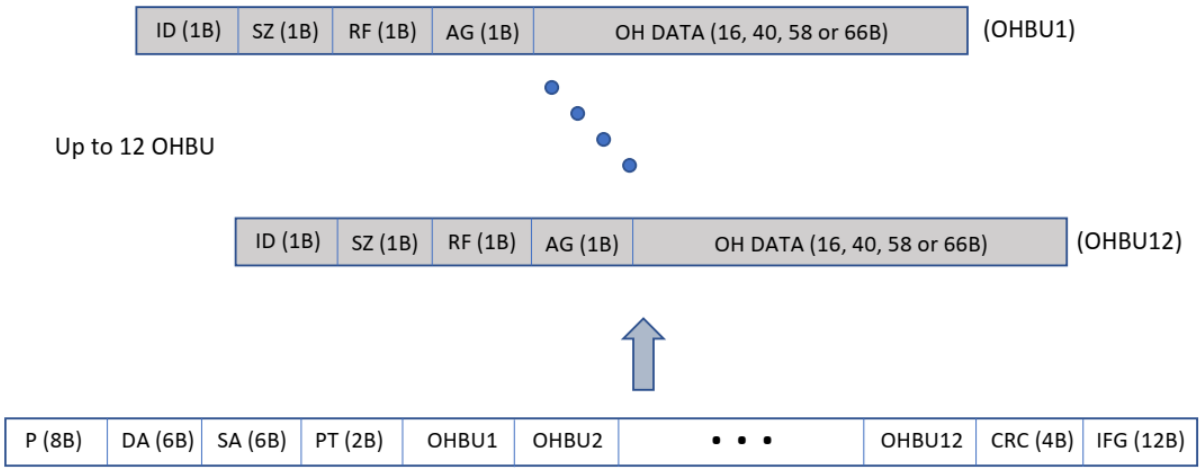
1.3.3 FlexO Add Port

In the case of FlexO clients, the number of OH bytes transported by the A-OHBU is 40 Bytes. Import from 1 up to 40 OTU Frame Overhead Fields from an external device. Bytes are imported in Add OH block units (A-OHBU), these are different than the D-OHBU. The number of OH bytes transported by the A-OHBU can be selected from 1 to 40. Each byte imported can replace any of the 40 OH bytes. The A-OHBU includes

- Port ID (ID)
- Size byte (SZ). This refers only to the number of OH bytes, excludes control bytes.
- Insertion reference byte (RF). When reference-driven insertion is enabled, this byte is compared to the current MFAS being processed by the Add port and the OH byte insertion takes place only if MFAS == reference byte.
- A 0's reserved byte to match the length of the D-OHBU.

After adding all the OH bytes and the Control bytes, the main mode supported is 44B OHBU.

Figure 5: ADD OHA Interface Structure



As shown in Figure 5, multiple A-OHB data blocks are encapsulated into the data portion of a standard Ethernet frame (Preamble, SA, DA, PT, [OHBs = DATA] , CRC, IFG). Each Ethernet frame carries up to 12 OHBU overhead data blocks

- 4 OHBUs for 400GE framer
- Or 3 OHUs for each 100G framer (max 12 OHBUs from 4x100G framers)

For reference, the OH Fields processed by Drop and Add ports are illustrated in Figure 6 through Figure 9 below. The OHA can also access all the OTU(C) overhead bytes, including GCC0.

1.3.4 Port ID

Refer to Section 1.2.4 Port ID. The same rule as in Section 1.2.4 applies to ADD port operation.

Figure 6: ODUC/OPUC Overhead Fields

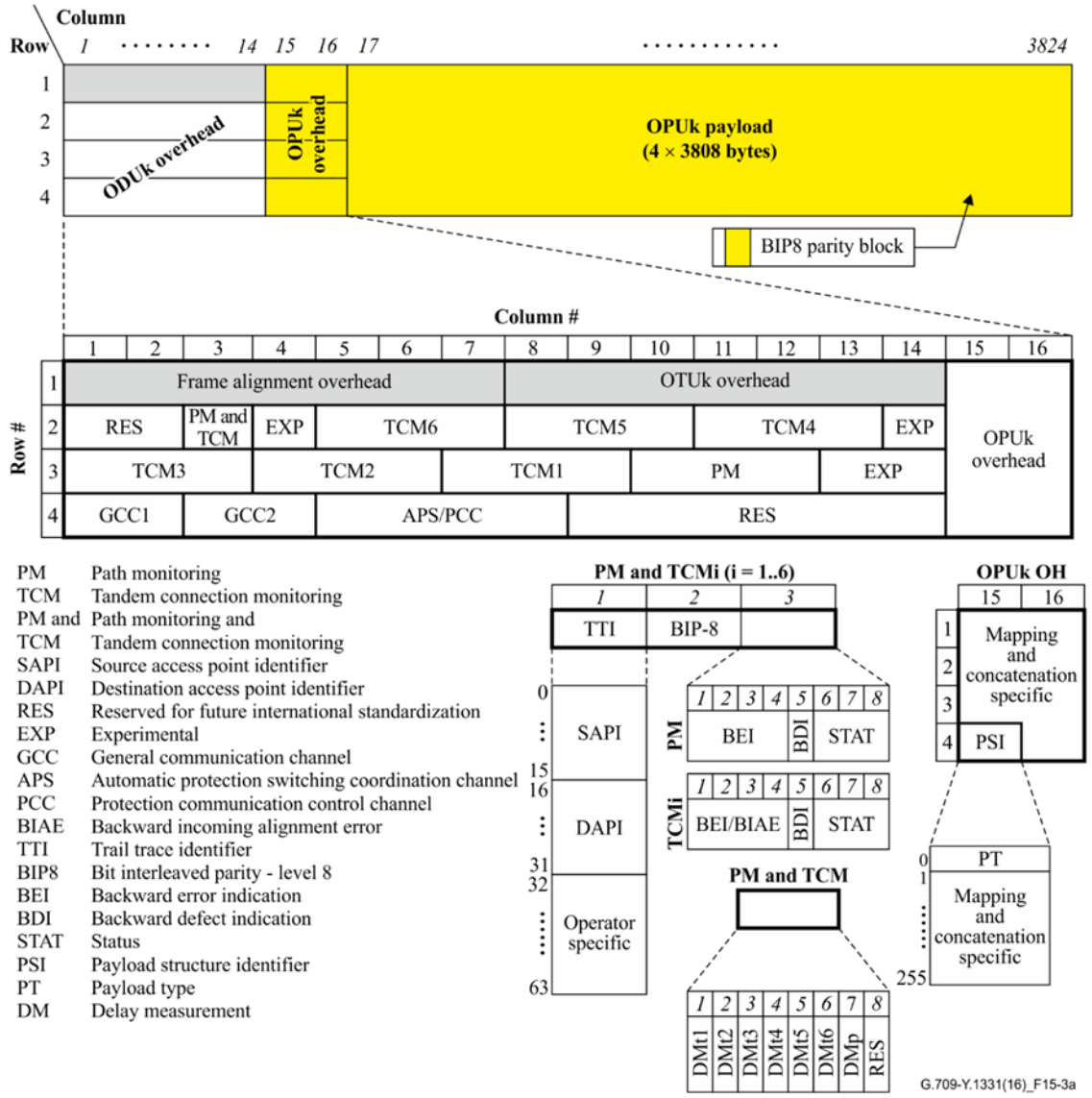


Figure 7: ODUC.OPUC Overhead Fields

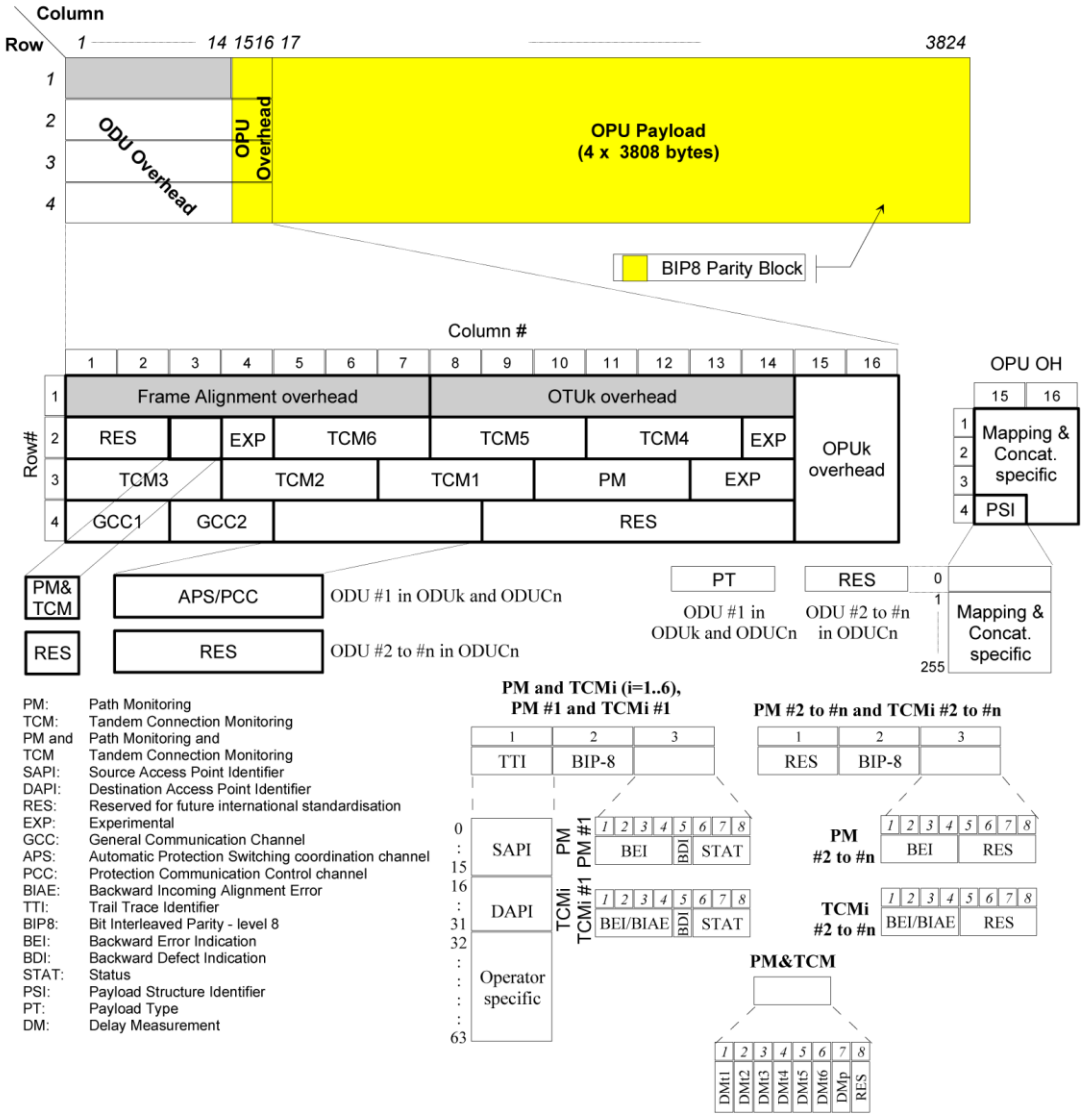
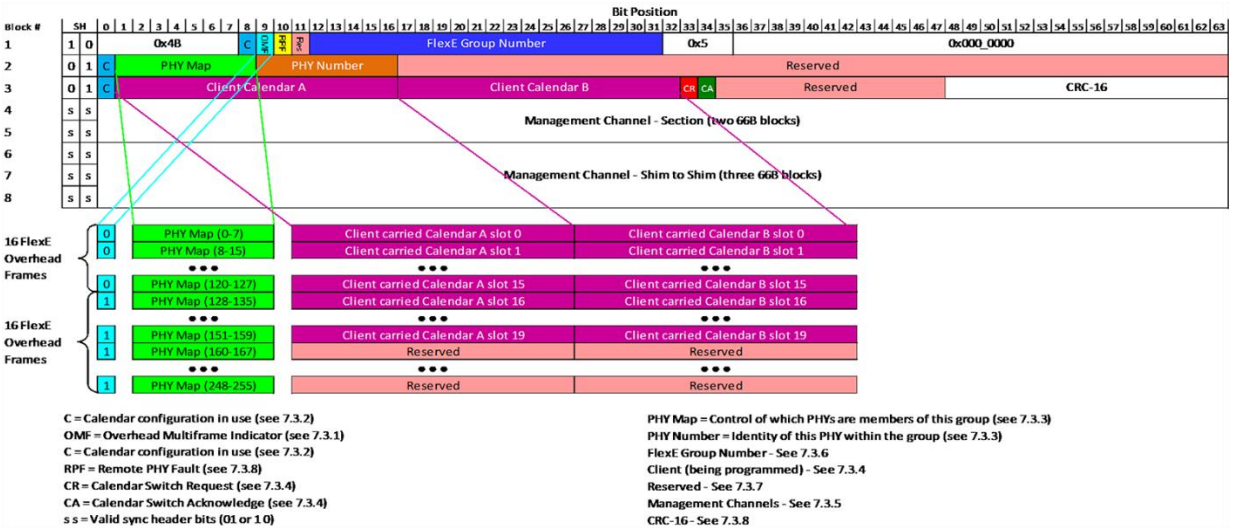


Figure 8: FlexO Overhead Fields

MFAS bits		1	2	3	4	5	6	7	8	9	10	11	12	13	·····	26	27	28	29	·····	40
[678] frame																					
000	1	MFAS	STAT	GID	GID	GID	RES	PID	MAP					CRC		FCC		OSMC	RES		
001	2	MFAS	STAT	AVAIL	RES				MAP					CRC		FCC		OSMC			
010	3	MFAS	STAT						MAP					CRC		FCC		OSMC			
011	4	MFAS	STAT						MAP					CRC		FCC		OSMC			
100	5	MFAS	STAT						MAP					CRC		FCC		OSMC			
101	6	MFAS	STAT						MAP					CRC		FCC		OSMC			
110	7	MFAS	STAT						MAP					CRC		FCC		OSMC			
110	8	MFAS	STAT						MAP					CRC		FCC		OSMC			

Figure 9: FlexE Overhead Fields



1.4 Examples

Example 1 (4x100G OTU mode)

- One OHA SGMII interface can add/drop GCC1 of framer 1 with OHB1, GCC1 of framer 2 with OHB2, GCC1 of framer 3 with OHB3, GCC1 of framer 4 with OHB4.
- One OHA SGMII interface can add/drop GCC1 of framer 1 with OHB1, GCC2 of framer 2 with OHB2, PM of framer 3 with OHB3, TCMs (TCM1~TCM6) of framer 4 with OHB4.

Example 2 (FlexE mode)

- At full speed (3.125 Gbps mode), for 4x100GE, one SGMII interface can deliver fixed 66 FlexE OH bytes of framer 1 with OHB1, fixed 66 FlexE OH bytes of framer 2 with OHB2, fixed 66 FlexE OH bytes of framer 3 with OHB3, and no FlexE OH bytes from framer 4 due to limited bandwidth (max 256 Bytes available).
- For 4x100GE, one SGMII interface can deliver fixed 66 FlexE OH bytes of framer 1 with OHB1, any OxU OH bytes of framer 2 with OHB2, any OxUC OH bytes of framer 3 with OHB3, fixed 66 FlexE OH bytes of framer 4 with OHB4 at 3.125 Gbps speed.

- For 4x100GE, one SGMII interface can deliver fixed 66 FlexE OH bytes of framer 1 with OHB1, any FlexO OH bytes of framer 1 with OHB2, any OxU OH bytes of framer 1 with OHB3, any OxUC OH bytes of framer 1 with OHB4.
- For 1x400GE (ITU path), one SGMII interface can deliver any OxUFlex OH bytes with OHB1, any OxUC OH bytes with OHB2.

NOTE: 1x400GE (ZR path) does not support OHA interface

1.5 OHA Flow Control

This section describes the overhead flow-control mechanism for any given pair of “OHA Add” and “OHA Drop” blocks inside the INDx400 Framer/Mapper. As an example, Figure 10 depicts such a pair of OHA blocks within the detailed data path diagram of the FRAMER_400GE; in this case at the ODUflex level. Similar associations exist at the various mapping levels in the 100G framers.

Figure 10: Add and Drop Pair - 400G ODUflex Example

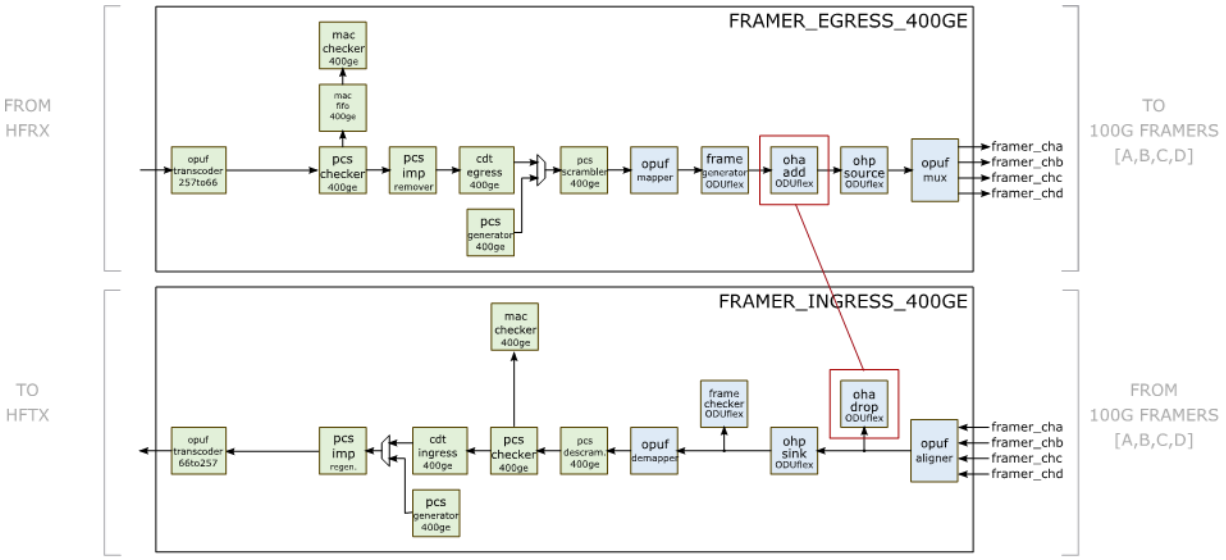


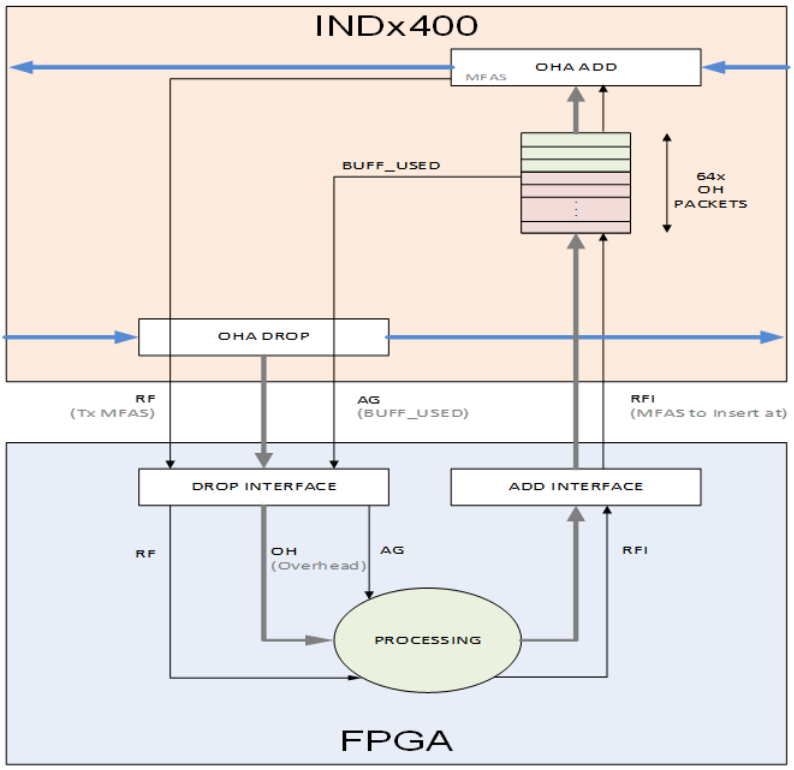
Figure 11 shows a conceptual view of the flow-control mechanism. In this example, INDx400 interfaces to an external FPGA through the SGMII/1000BASE-X interface. The FPGA processes the dropped overhead packets and generates overhead packets for insertion in the corresponding “OHA Add” block.

The “OHA Add” block drains a set of internal overhead packet buffers by consuming one overhead packet for each OTN frame transmitted. There are a total of 64 overhead packet buffers per “OHA Add” blocks in the device. Each buffer contains one overhead ADD block (58 bytes or 16 bytes of overhead depending on the operation mode). The number of overhead packet buffers used (BUFF_USED) is conveyed to the associated “OHA Drop” block through the AG byte. The BUFF_USED number is a measure of the “OHA Add” buffer fill-level that the FPGA can use to determine how many overhead ADD packets it can post towards the INDx400 “OHA Add” block.

Based on AG (Buffer fill-level) and overhead received through the OHA Drop interface, the FPGA posts overhead ADD packets towards the INDx400. Since the latency through the SGMII/1000BASE-X interface is expected to be small compared to the OTN frame period, the FPGA may run ahead of the INDx400 “OHA Add” and post several overhead packet blocks in advance. The FPGA uses the buffer fill

level information fed back through the AG byte to adjust how many overhead packet blocks to post with the goal of maintaining the fill level centered at about half of the total number of buffers (64/2 = 32), thus avoiding overflow and underflow situations.

Figure 11: OHA Flow Control



1.6 OHA Clocking

Clocking for the SGMII/1000BASE-X interface is derived internally from the common reference clock (REFCLK) with electrical characteristics listed under the Reference Clock (REFCLK) Input Specifications section. Internally, INDx400 derives the SGMII/1000BASE-X transmit reference clock (timing the SGMII/1000BASE-X output) from REFCLK. No other external reference clock is required for the SGMII/1000BASE-X operation.

1.7 APIs

Please refer to INDx400 API manual for more details. This section will be further updated in future revision when the API manual update is complete.

1.7.1 LoadOhaFw

Place OHA firmware image (say “oha_image.bin”) in MCU memory. This API must be called to load the OHA firmware before utilizing OHA interface and OHA APIs. Figure 16 shows example code to load OHA FW.

Note: For DSP FW 0.17.x or lower, OHA FW is provided as independent image. Starting from DSP FW 0.18.x, OHA image will be combined into regular DSP FW. This change will result in an increase of the

regular binary size from 1548KBytes to 1724KByes (additional 176KB due to addition of OHA binary). The OHA image will be on top of the regular image (new image = DSP image + OHA image).

The regular DSP image loading process won't be changed – no S/W change is required for the standard DSP image. However, the process of loading the OHA binary needs minor change – only starting memory position is changed from 0x0 to 0xC1800 and other OHA loading sequence remains same using same API (LoadOhaFw).
The OHA image is loaded in blocks of 176 16-bit words. The number of bytes to be copied in KB is equal to 128.

For DSP FW 0.17.x or lower (see Figure 16 script)

```
for i in range(0, len(s), 176):
    chip.api.LoadOhaFw(data=s[i:i+176])
```

For DSP FW 0.18.x or higher

```
for i in xrange(0xC1800, len(s), 176):
    chip.api.LoadOhaFw(data=s[i:i+176])
```

1.7.2 GetOhaFw

After loading the OHA firmware, call this API to make sure the OHA image is loaded to DSP. If the firmware loading is successful, this API returns

{'Status': 0, 'Info': 0, 'Length': 4}

If the firmware loading is NOT successful, this API will return following error code.

{'Status': 3, 'Info': 45, 'Length': 4}

1.7.3 SetOhaGlobalConfig

This API must be executed to configure and initialize the OHA interface as follows:

- **sgmii_channel**: Select OHA interface 0 or 1.
  - **OHA_SGMII_CHANNEL_0** selects OHA interface 0
  - **OHA_SGMII_CHANNEL_1** selects OHA interface 1
- **sgmii_rate:**
  - **OHA_SGMII_OFF** to turn off SGMII interface
  - **OHA_SGMII_HALF_SPEED** to select 1.25Gbps OHA data processing
  - **OHA_SGMII_FULL_SPEED** to select 3.125Gbps OHA data processing
  - **OHA_SGMII_CLOCKS:** RESERVED
- **lpbk_mod**: Select DSP internal loopback mode
  - **OHA_LPBK_DISABLED** to disable internal loopback
  - **OHA_LPBK_TXPCS_TO_RXPCS** to loopback DSP Drop to DSP Add path. See Figure 12.
  - **OHA_LPBK_RXMAC_TO_TXMAC** to loopback customer FPGA Add port to customer FPGA Drop path. See Figure 13.

- **rx_discard_on_address_mismatch**: Frame discard mode. User can select “Accept” or “Discard” when MAC source and destination addresses do not match.
- **tx_mac_src_address**: Source address of TX MAC.
- **tx_mac_dst_address**: Destination address of RX MAC.

Please see Figure 16 for example code.

Figure 12: TXPCS to RXPCS Loopback

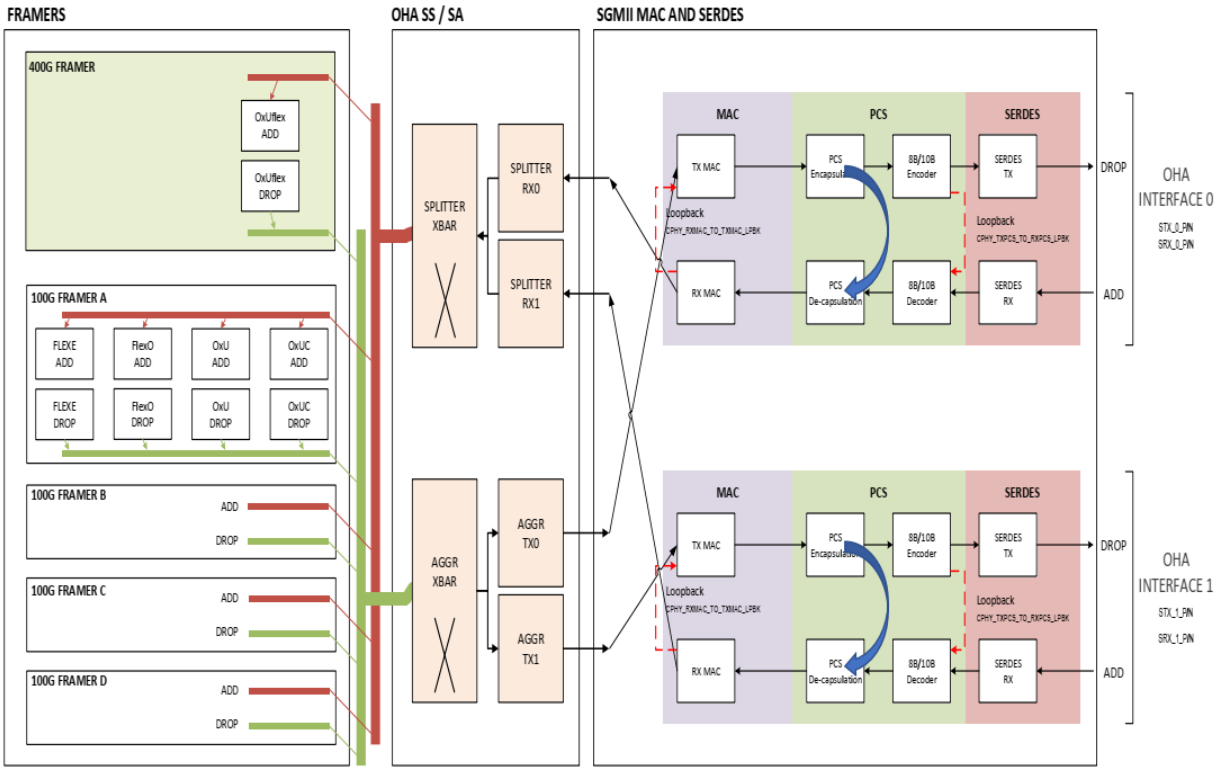
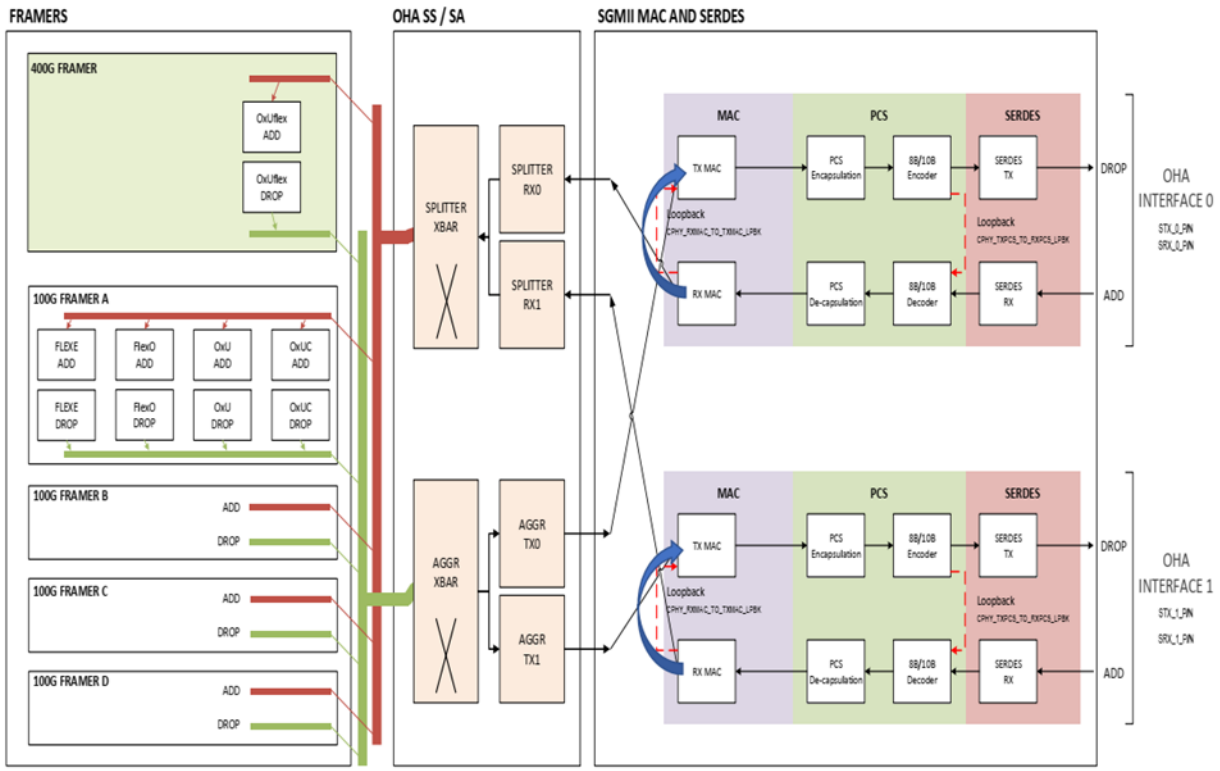


Figure 13: RXMAC to TXMAC Loopback



1.7.4 GetOhaGlobalConfig

Returns OHA configuration set by **SetOhaGlobalConfig** API.

1.7.5 GetOhaGlobalStatus

Returns OHA status from selected OHA channel.

1.7.6 SetOtnOhaConfig

This API configures following OTN OHA interface parameters.

- **channel**: Select one of 4x100G framers or 1x400GE framer.
- **map_level**: Select Framer OH mapping level (see Figure1**4**, 1**5**)
  - Host, MAP1, MAP2 or Line
- **sgmii_channel**: Select SGMII channel0 or 1
- **add_byte_select[64]**: Select OTN overhead field for Add. This contains 64 x 8-bit fields corresponding to each one of the 64 OH DATA Bytes in the OTU OHA Frame and selects which OH bytes to be added.

- **add_byte_mask[64]:** This contains 64 x 8-bit fields corresponding to each one of the 64 OH DATA Bytes in the OTU OHA Frame and is to mask the selected OH bytes by Add Byte Select. See Figure 16, 17 for example.
- **add_mfas_align:** Enable or disable MFAS based insertion. If ENABLED, bytes are inserted based on the MFAS Reference (RFI byte) from the Add Port.
- **drop_byte_select[64]:** Select OTN overhead field for Drop. This contains 64 x 8-bit fields corresponding to each one of the 64 OH DATA Bytes in the OTU OHA Frame and selects which OH bytes to drop. The length of the OHBU also depends on the number of SGMII lanes used and the number and type of ADD/DROP ports enabled per framer.
- **drop_ohbu_length**: Selects the length in bytes of the OH DATA portion of the OHBU for the Add or Drop port. If **sgmii_rate** in **SetOhaGlobalConfig** is set to **OHA_SGMII_HALF_SPEED** (1) then the max length is 16. If **sgmii_rate** is set to **OHA_SGMII_FULL_SPEED** (2) then the max length is 58. The length of the OHBU also depends on the number of SGMII lanes used and the number and type of ADD/DROP ports enabled per framer.
- **port_enable**: enable or disable SGMII port

Figure 16 script (associated with Figure 17 for OTN OH structure) explains how to add GCC0[12] byte through SGMII. **sgmii_rate** in **SetOhaGlobalConfig** is set to **OHA_SGMII_HALF_SPEED** (1) so INDx400 is handling max 16 OTN OH byte to OHBU1:

OH DATA [byte 1] = GCC0 [byte 12]

OH DATA [byte 2:6] = Frame Alignment [byte 2:6]

OH DATA [byte 7] = MFAS [byte 7]

OH DATA [byte 8:10] = SM [byte 8:10]

OH DATA [byte 11] = GCC0[byte 11]

OH DATA [byte 12] = Frame Alignment [byte 1]

OH DATA [byte 13:14] = RES [byte 13:14]

OH DATA [byte 15] = RES [byte 15]

OH DATA [byte 16] = JC [byte16]

INDx400 will expect GCC0(byte 12) to come in Byte 1 instead of byte 12. And Byte 12 would be replaced with 1st Byte of the Frame Alignment.

1.7.7 GetOtnOhaConfig

Returns OHA configuration set by **SetOtnOhaConfig** API.

1.7.8 SetFlexeOhaConfig

This API configures following FlexE OHA interface parameters.

- Framer channel: Select one of 4x100G framers.
- SGMII channel: Select SGMII channel0 or 1
- Drop OHBU length: Select number of OHBU for Drop. Normally 66.
- Add PCS Mask
- Port control: enable or disable SGMII port

1.7.9 GetFlexeOhaConfig

Returns OHA configuration set by **SetFlexeOhaConfig** API.

1.7.10 SetFlexoOhaConfig

This API configures following FlexO OHA interface parameters.

- **channel**: Select one of 4x100G framers or 1x400GE framer.
- **sgmii_channel**: Select SGMII channel0 or 1
- **add_byte_select[40]**: Select FlexO overhead field for “add” operation. This contains 40 x 8-bit fields corresponding to each one of the 40 OH DATA Bytes in the FlexO frame and selects which OH bytes to be added.
- **add_byte_mask[40]:** This contains 40 x 8-bit fields corresponding to each one of the 40 OH DATA Bytes in the FlexO Frame and is to mask the selected OH bytes by **add_byte_select**. See Figure 16 for example. The length of the OHBU also depends on the number of SGMII lanes used and the number and type of ADD/DROP ports enabled per framer.
- **drop_ohbu_length**: Selects the length in bytes of the OH DATA portion of the OHBU for the Add or Drop port. If **sgmii_rate** in **SetOhaGlobalConfig** is set to **OHA_SGMII_HALF_SPEED** (1) then the max length is 16. If **sgmii_rate** is set to **OHA_SGMII_FULL_SPEED** (2) then the max length is 40. The length of the OHBU also depends on the number of SGMII lanes used and the number and type of ADD/DROP ports enabled per framer.
- **mfas_insertion_enable:** Reserved for future use. DO not use.
- **port_enable**: enable or disable SGMII port

1.7.11 GetFlexoOhaConfig

Returns OHA configuration set by **SetFlexoOhaConfig** API

Figure 14: OHP Source Blocks – 100G Framer Slices

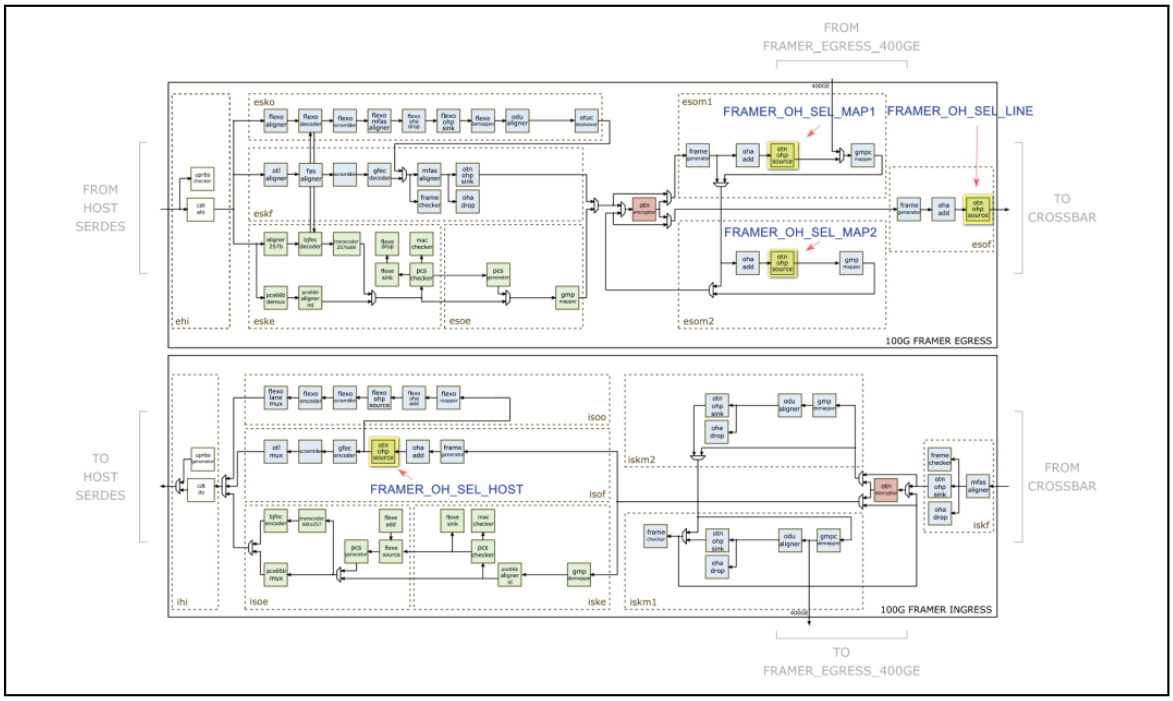
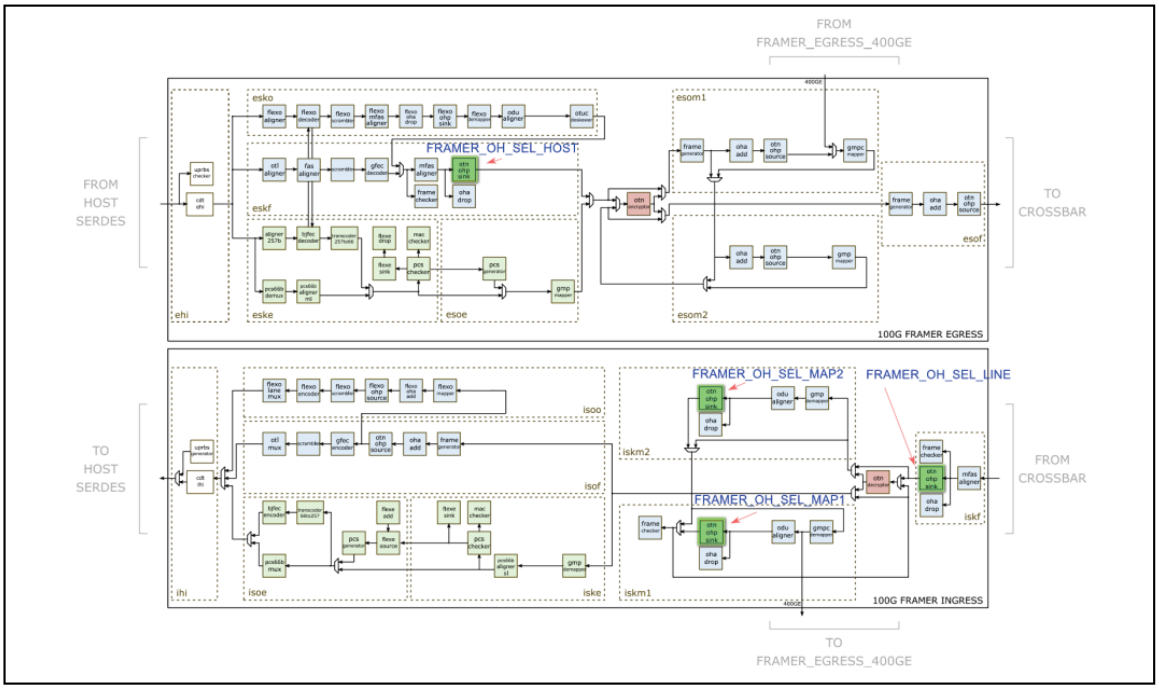


Figure 15: OHP Sink Blocks – 100G Framer Slices



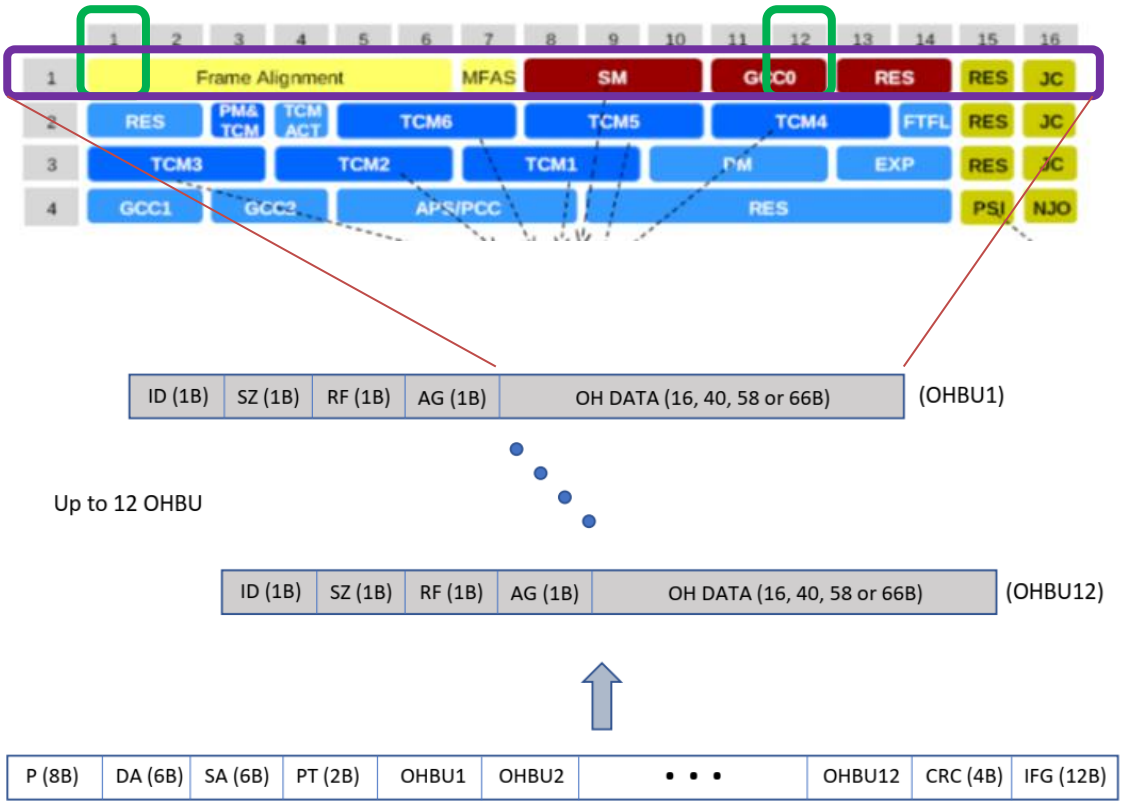
1.7.12 Example Script

```
from struct import unpack
import datetime

SET_OHA_FW_ACTION_START = 0
SET_OHA_FW_ACTION_END = 1

FW_OHA = r"C:\device\fw\bin\oha_image.bin"

def LoadOHA_FW():
    # Load OHA FW
    fo = open(FW_OHA, 'rb')
    s = fo.read()
    s = unpack('<'+str(176 * 512)+'H', s)  # little endian, two bytes swapped


    a = datetime.datetime.now()

    # Step 1: API Start
    chip.api.SetOhaFw(action=SET_OHA_FW_ACTION_START)

    # Step 2: Loading FW
    print 'Loading FW ...',
    for i in range(0, len(s), 176):
        chip.api.LoadOhaFw(data=s[i:i+176])
    print 'done'
    print 'time=', datetime.datetime.now() - a

    # Step 3: API end
    chip.api.SetOhaFw(action=SET_OHA_FW_ACTION_END)
    print 'time=', datetime.datetime.now() - a

LoadOHA_FW()
print chip.api.GetOhaFw()

# SetOhaGlobalConfig
chip.api.SetOhaGlobalConfig(
        sgmii_channel=Oha_Sgmii_Channel.OHA_SGMII_CHANNEL_0,
        sgmii_rate=Oha_Sgmii_Interface_Mode.OHA_SGMII_HALF_SPEED,
        lpbk_mode=Oha_Loopback_Mode.OHA_LPBK_DISABLED,
        rx_discard_on_address_mismatch=Oha_Frame_Discard_Mode.OHA_FRAME_ACCEPTED,
        tx_mac_src_address=0x0000FFFFFFFFFFFF,
        tx_mac_dst_address=0x0000FFFFFFFFFFFF
    )

# SetOtnOhaConfig

add_byte_select = [
        12 , 2  , 3  , 4  , 5  , 6  , 7  , 8  , 9 , 10 , 11 ,  1 , 13 , 14 , 15 , 16 ,
        17 , 18 , 19 , 20 , 21 , 22 , 23 , 24 , 25 , 26 , 27 , 28 , 29 , 30 , 31 , 32 ,
        33 , 34 , 35 , 36 , 37 , 38 , 39 , 40 , 41 , 42 , 43 , 44 , 45 , 46 , 47 , 48 ,
        49 , 50 , 51 , 52 , 53 , 54 , 55 , 56 , 57 , 58 , 59 , 60 , 61 , 62 , 63 , 64 ]
```



```
#REPLACE GCC0
add_byte_mask = [
      0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 255, 0 , 0 , 0 , 0 ,
      0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 ,
      0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 ,
      0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 ];

drop_byte_select  = [
      12 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 , 11 , 1 , 13 , 14 , 15 , 16 ,
      17 , 18 , 19 , 20 , 21 , 22 , 23 , 24 , 25 , 26 , 27 , 28 , 29 , 30 , 31 , 32 ,
      33 , 34 , 35 , 36 , 37 , 38 , 39 , 40 , 41 , 42 , 43 , 44 , 45 , 46 , 47 , 48 ,
      49 , 50 , 51 , 52 , 53 , 54 , 55 , 56 , 57 , 58 , 59 , 60 , 61 , 62 , 63 , 64 ]

chip.api.SetOtnOhaConfig(
   channel=Framer_Channel.FRAMER_CH_A,
   map_level=Oh_Map_Level.FRAMER_OH_SEL_MAP1,
   port_enable = Enable.ENABLE,
   sgmii_channel = Oha_Sgmii_Channel.OHA_SGMII_CHANNEL_0,
   add_mfas_align = Enable.DISABLE,
   drop_ohbu_length = 16,
   add_byte_select = add_byte_select,
   drop_byte_select = drop_byte_select,
   add_byte_mask = add_byte_mask
   )

# SetFlexoOhaConfig
# Configure OHA for MFAS/STAT test
add_byte_select = [
   1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10,  11 , 12 , 13 , 14 , 15 , 16 ,
   17 , 18 , 19 , 20 , 21 , 22 , 23 , 24 , 25 , 26 , 27 , 28 , 29 , 30 , 31 , 32 ,
   33 , 34 , 35 , 36 , 37 , 38 , 39 , 40 ];

#REPLACE BYTE RIGHT AFTER MFAS
add_byte_mask = [
   0 , 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
   0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 ,
   0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 ];

drop_byte_select  = [
   1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 , 11 , 12 , 13 , 14 , 15 , 16 ,
   17 , 18 , 19 , 20 , 21 , 22 , 23 , 24 , 25 , 26 , 27 , 28 , 29 , 30 , 31 , 32 ,
   33 , 34 , 35 , 36 , 37 , 38 , 39 , 40 ];

chip.api.SetFlexoOhaConfig(
   channel=Framer_Channel.FRAMER_CH_A,
   sgmii_channel = Oha_Sgmii_Channel.OHA_SGMII_CHANNEL_0,
   add_byte_select = add_byte_select,
   add_byte_mask = add_byte_mask,
   drop_byte_select = drop_byte_select,
   drop_ohbu_length = 40,
   mfas_insertion_enable = Enable.DISABLE,
   port_enable = Enable.ENABLE
   )
```


Figure 16: OHA Example (GCC0[byte 12] byte)



2. How to Debug OHA Issues

2.1 SGMII Interface Debug

When OHA interface bring-up fails (no overhead data traffic works), try following debugging check points.

Check point #1

Turn off **autonegotiation** in the PCS Phy (FPGA). In INDx400 **autonegotiation** is off as default.

Check point #2

1) Host FPGA internal loopback test (refer to Figure 11. OHA flow control)
    a. Set FPGA to internal loopback
    b. Make sure overhead data communication works.
2) FPGA external loopback test (refer to Figure 11. OHA flow control)
    a. Connect FPGA TX an FPGA RX with external cable if available.
    b. Make sure overhead traffic works.

Check point #3 (Internal loopback)

1) INDx400 SGMII internal loopback test (refer to Fig 12. TXPCS to RXPCS loopback) to debug DROP operation
    a. Set INDx400 to internal loopback (TXPCS to RXPCS). See example code below.
    b. Make sure overhead traffic works.

Example code

```
chip.api.SetOhaGlobalConfig(
        sgmii_channel=Oha_Sgmii_Channel.OHA_SGMII_CHANNEL_0,
        sgmii_rate=Oha_Sgmii_Interface_Mode.OHA_SGMII_HALF_SPEED,
       lpbk_mode=Oha_Loopback_Mode. OHA_LPBK_TXPCS_TO_RXPCS,
        rx_discard_on_address_mismatch=Oha_Frame_Discard_Mode.OHA_FRAME_ACCEPTED,
        tx_mac_src_address=0x0000FFFFFFFFFFFF,
        tx_mac_dst_address=0x0000FFFFFFFFFFFF
    )
```

2) To debug ADD operation,
    a. Set DSP to internal loopback (Fig 13. RXMAC to TXMAC loopback). See example code below.
    b. FPGA to send packet and to check the traffic.

Example code

```
chip.api.SetOhaGlobalConfig(
        sgmii_channel=Oha_Sgmii_Channel.OHA_SGMII_CHANNEL_0,
        sgmii_rate=Oha_Sgmii_Interface_Mode.OHA_SGMII_HALF_SPEED,
        lpbk_mode=Oha_Loopback_Mode.OHA_LPBK_RXMAC_TO_TXMAC,
        rx_discard_on_address_mismatch=Oha_Frame_Discard_Mode.OHA_FRAME_ACCEPTED,
        tx_mac_src_address=0x0000FFFFFFFFFFFF,
        tx_mac_dst_address=0x0000FFFFFFFFFFFF
    )
```


Check point #4

1) INDx400 OHA API configuration

   a. Check if following INDx400 APIs are properly set and called.

      - SetTransceiverMode
      - SetOhaGlobalConfig
      - SetOtnOhaConfig (OTU4 case)
      - Or SetFlexoOhaConfig (FlexO case)

   b. Example #1

      - In 4x100GE (e.g. SetTransceiverMode API selects "ST_100GEBJLR"), "HOST" framers can't be accessible for overhead. If "HOST" framers are selected in "SetOtnOhaConfig", no overhead traffic will be established. See client mapping table in datasheet (table 9).

Example #1

```
stm_args = {'reserved_0': Path.BOTH,
            'line_fec': Line_Fec.SDFEC_LP_DC,
            'pilot_symbol_separation': 20,
            'line_shaping': 0,
            'line_modulation': Line_Modulation.LM_QAM16,
            'bcd_mode': 40000, # default: 7200, max 40000
            'ltx_osr': Osr_Line.THREE_OVER_TWO,
            'lrx_osr': Osr_Line.THREE_OVER_TWO,
            'reserved_1': 0x0,
            'signal_type': [Signal_Type.ST_100GEBJLR]*4,
            'line_mapping': [Line_Mapping.GMP_4CP]*4,
            'host_modulation': Host_Modulation.HM_PAM4}

hip.api.SetTransceiverMode(**stm_args)
chip.api.SetOtnOhaConfig(
   channel=Framer_Channel.FRAMER_CH_A,
   map_level=Oh_Map_Level.FRAMER_OH_SEL_HOST,
   port_enable = Enable.ENABLE,
   sgmii_channel = Oha_Sgmii_Channel.OHA_SGMII_CHANNEL_0,
   add_mfas_align = Enable.DISABLE,
   drop_ohbu_length = 16,
   add_byte_select = add_byte_select,
   drop_byte_select = drop_byte_select,
   add_byte_mask = add_byte_mask
   )
```


Table 9 Client Mapping Stages (ITU-T Path)

Client Mapping	GMP Stages			FRAMER Configuration					M200 Compatible
	GMP4	GMPC	GMPP	400G FRAMER	HOST 100G FRAMER	MAP1 100G FRAMER	MAP2 100G FRAMER	LINE 100G FRAMER	
OTU4	-	-	-	-	OTU4	-	-	OTU4	YES
OTU4_C	-	ON	-	-	OTU4	OTU4	-	OTUC	NO
OTU4_P	-	-	ON	-	OTU4	-	OTU4	OTUP	YES
OTU4_CP	-	ON	ON	-	OTU4	OTU4	OTUC	OTUP	NO
FOIC	-	-	-	-	OTUC	-	-	OTUC	NO
FOIC_P	-	-	ON	-	OTUC	-	OTUC	OTUP	NO
100GE_4	ON	-	-	-	-			OTU4	YES
100GE_4C	ON	ON	-	-	-	OTU4	-	OTUC	NO
100GE_4P	ON	-	ON	-	-		OTU4	OTUP	YES
100GE_4CP	ON	ON	ON	-	-	OTU4	OTUC	OTUP	NO
200GE_C	-	ON	-	ODUflex	-	-	-	OTUC	NO
200GE_CP	-	ON	ON	ODUflex	-	-	OTUC	OTUP	NO
400GE_C	-	ON	-	ODUflex	-	-	-	OTUC	NO
400GE_CP	-	ON	ON	ODUflex	-	-	OTUC	OTUP	NO

c. Example #2

- “add_byte_select” and “drop_byte_select” must select all 64 bytes for SetOtnOhaConfig API. See example #2 below. When any byte = 0 in “add_byte_select”or drop_byte_select” as in configuration #2, API will return error code. It is required to assign a value to each of the 64 bytes to make sure there are no double selections. Also 0 is not allowed.

Example #2

Configuration #1

add_byte_select = [1,2,3,4, 5,6,7,8, 9,10,11,12, 13,14,15,16,
17,18,19,20, 21,22,23,24, 25,26,27,28, 29,30,31,32,
33,34,35,36, 37,38,39,40, 41,42,43,44, 45,46,47,48,
49,50,51,52, 53,54,55,56, 57,58,59,60, 61,62,63,64]

add_byte_mask = [0,0,0,0, 0,0,0,0, 0,0,0,255, 255,0,0,0,
0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0]

drop_byte_select = [1,2,3,4, 5,6,7,8, 9,10,11,12, 13,14,15,16,
17,18,19,20, 21,22,23,24, 25,26,27,28, 29,30,31,32,
33,34,35,36, 37,38,39,40, 41,42,43,44, 45,46,47,48,
49,50,51,52, 53,54,55,56, 57,58,59,60, 61,62,63,64]

chip.api.SetOtnOhaConfig(channel=0, map_level=0, sgmii_channel=0,
add_byte_select=add_byte_select,
add_byte_mask=add_byte_mask, add_mfas_align=0,
drop_byte_select=drop_byte_select, drop_ohbu_length=10, port_enable=1)

('Response: ', ['0x4'], '(Exe Time: ', 1, 'unit , 0.2 s)')

Configuration #2

add_byte_select = [1,2,3,4, 5,6,7,8, 9,10,11,12, 13,14,15,16,
17,18,19,20, 21,22,23,24, 25,26,27,28, 29,30,31,32,
33,34,35,36, 37,38,39,40, 41,42,43,44, 45,46,47,48,
49,50,51,52, 53,54,55,56, 57,58,59,60, 61,62,63,0]

add_byte_mask = [0,0,0,0, 0,0,0,0, 0,0,0,255, 255,0,0,0,
0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0]

drop_byte_select = [1,2,3,4, 5,6,7,8, 9,10,11,12, 13,14,15,16,
17,18,19,20, 21,22,23,24, 25,26,27,28, 29,30,31,32,
33,34,35,36, 37,38,39,40, 41,42,43,44, 45,46,47,48,
49,50,51,52, 53,54,55,56, 57,58,59,60, 61,62,63,0]

chip.api.SetOtnOhaConfig(channel=0, map_level=0, sgmii_channel=0,
add_byte_select=add_byte_select,
add_byte_mask=add_byte_mask, add_mfas_align=0,
drop_byte_select=drop_byte_select, drop_ohbu_length=10, port_enable=1)

('Response: ', ['0x7020004'], '(Exe Time: ', 1, 'unit , 0.2 s)')
INVALID FIELD IN COMMAND: drop_byte_select

Check point #5

IF ADD operation does not work, try ET (Ethernet Type) = 0x0. User may want to reboot FPGA after ET update if ET change itself does not make ADD operation work.

2.2 Reboot OHA Interface

In case rebooting SGMII interface is necessary, please consider following reboot procedure.

a) Turn down SGMII traffic by setting “sgmii_rate” of **SetOhaGlobalConfig** API to “OFF”.
b) Power down FPGA
c) Power up FPGA but don’t configure/bring-up FPGA (PCS/MAC) and DSP OHA SGMII yet.
d) Download OHA FW
e) Setup INDx400 OHA configuration with “sgmii_rate” of **SetOhaGlobalConfig** API to HALF or FULL SPEED.
f) Configure/bring-up FPGA (PCS/MAC) and INDx400 OHA SGMII.

chip.api.SetOhaGlobalConfig(
sgmii_channel=Oha_Sgmii_Channel.OHA_SGMII_CHANNEL_0,
sgmii_rate=Oha_Sgmii_Interface_Mode.OHA_SGMII_HALF_SPEED,
lpbk_mode=Oha_Loopback_Mode.OHA_LPBK_DISABLED,
rx_discard_on_address_mismatch=Oha_Frame_Discard_Mode.OHA_FRAME_ACCEPTED,


```
    tx_mac_src_address=0x0000FFFFFFFFFFFF,
    tx_mac_dst_address=0x0000FFFFFFFFFFFF
)
```

2.3 INDx400 OHA Debug Register

Check following DSP registers if OHA overhead traffic does not work.

- 0x87001068 (MAC_TX_FRAMES)
  - Non-0 indicates DSP OHA RX receiving data (good state)
  - 0 means no packet traffic from FPGA to DSP (bad state)
- 0x8700106C (MAC_RX_FRAMES)
  - Non-0 indicates DSP OHA TX sending data (good state)
  - 0 means no packet traffic from DSP OHA TX (bad state)
- 0x87001070 (MAC_CRC_ERR): Make sure no error for good state.
- 0x87001074 (MAC_ALIGNMENT_ERR): Make sure no error for good state.
- 0x87001088 (MAC_IF_IN_ERROR, MAC interface input error): Make sure no error for good state.
- 0x87000154 (SPLITTER SZ ERR. These are received OHBU size errors, they happened if the RX ethernet MAC packets are good but some of the symbols are corrupted, not that many, and the size field is a good indication of that corruption. It can also happen if the user sends a bad packet size, that is less common.)

IF DSP does not receive overhead packet from FPGA (ADD operation),

Check and make sure following registers are NOT 0 for good state. 0 means DSP OHA input does not receive packet from FPGA.

- 0x848024b0 (FIFO entries. Framer Ch A)
- 0x850024b0 (FIFO entries. Framer Ch B)
- 0x858024b0 (FIFO entries. Framer Ch C)
- 0x860024b0 (FIFO entries. Framer Ch D)

3. Flow Control Sample Verilog

Figure 17: Sample Verilog code for SGMII Interface Flow Control

```
generate
    genvar ii ;
    for (ii  = 0; ii  < 4; ii  = ii  + 1)
    begin:gen_data_gens

        reg [8 -1 :0]           grant_ii;
        reg [8 -1 :0]           grant_ii_q;
        reg                     stop_ii;
        wire                    up_ii, dwn_ii;

        ohbu_gen_32
        u_ohbu_gen
        (
            .i_enable                       ( rf_static_enabled_ports[ii]               ) ,
            .i_valid                        ( ports_request[ii]                         ) ,
            .i_stop                         ( stop_ii                                   ) ,
            .i_start                        ( timer_start | rf_static_disable_timer[0]  ) ,
            .i_rf_static_payload_lenght     ( rf_static_payload_lenght[8-1:0]           ) ,
            .i_rf_static_id                 ( rf_static_ids[ii*8 +:8]                   ) ,
            .i_rf_static_pattern            ( 32'hCCDDEEFF                              ) ,
            .i_rf_static_enabled_counters   ( rf_static_enabled_counters[8-1:0]         ) ,
            .i_rf_static_insert_fixed_byte  ( rf_static_insert_fixed_byte[4-1:0]        ) ,
            .o_sop                          ( gen_sop[ii]                               ) ,
            .o_eop                          ( gen_eop[ii]                               ) ,
            .o_dval                         ( gen_dval[ii]                              ) ,
            .o_data                         ( gen_data[ii]                              ) ,
            .i_reset                        ( reset_sync | rf_reset_sync[16+ii]         ) ,
            .i_clock                        ( mac_clk                                   )
        );

        always @(posedge mac_clk)
        if(reset_sync | rf_reset_sync[2])
            grant_ii <= 8'b0;
        else if(i_grant_valid_bus[ii])
            grant_ii <= i_grant_bus[ii*8 +: 8];

        always @(posedge mac_clk)
        if(reset_sync | rf_reset_sync[2])
            grant_ii_q <= 8'b0;
        else if(i_grant_valid_bus[ii])
            grant_ii_q <= grant_ii;

        assign up_ii = grant_ii > grant_ii_q;

        assign dwn_ii = grant_ii < grant_ii_q;

        always @(posedge mac_clk)
        if(reset_sync | rf_reset_sync[2] )
         stop_ii <= 1'b1;
        else if ( up_ii & ( grant_ii > rf_static_flow_threshold[ii*8 +: 8] ))
         stop_ii <= 1'b0;
        else if ( dwn_ii & ( grant_ii < rf_static_on_flow_threshold[ii*8 +: 8] ))
         stop_ii <= 1'b1;

        assign stop_stat[ii] = stop_ii;

    end

endgenerate
```