Microsemi Ethernet Board API (MEBA) Programmers guide

User Guide

UG1069
Rev. 4.3.0
2017-12-15

Confidential

1. Introduction

This document is a programmers guide for the MEBA layer. MEBA is an API that allows the WebStaX application suite to be used on a variety of different boards, without the need for changing the WebStaX application itself.

Starting from the 4.1.0 release, the WebStaX software stack is using the MEBA layer. As such, all Microsemi reference boards have a MEBA layer.

As a MEBA layer is needed for all target boards, this guide introduces the MEBA layer, and how to implement this for a new board.

For more information about the WebStaX package, refer to the [ug1068] document.

Table of Contents

1.1. Audience

The intended audience for this document is software developers who need to create a MEBA layer for a new board.

1.2. Prerequisites

This document assumes the reader possesses the following skills/resources.

1. Fluent in "C".
2. Some knowledge of CMake.
3. MSCC WebStaX source package (version 4.1 or newer) and the corresponding binary BSP.
4. A target board for running the WebStaX package. The board is expected to have at least the bootloader running.

1.3. Terminology

Throughout the document the following terms are used.

MEBA API

The "C" language application programming interface defined by the header `<mscc/ethernet/board/api.h>`, being part of the Microsemi Switch API. This is located in the `meba/include` directory of the Microsemi Switch API source tree.

MEBA user

An application using the MEBA API. This will typically be the WebStaX switch application.

MEBA implementation
An implementation of the MEBA API for one or more specific hardware targets. The MEBA implementations for the Microsemi reference boards are located in the `meba/src` directory of the Microsemi Switch API source tree.

MESA
Microsemi Ethernet Switch API. See [MESA].

application
Another term for MEBA user.

2. MEBA Reference

2.1. Overview

The MEBA layer is typically implemented as a library, with one single exposed symbol `meba_initialize`, which is defined in the next section.

The MEBA library can either be linked statically with the application, or it can be loaded dynamically. The WebStaX application currently load MEBA dynamically, and has knowledge of which MEBA library to use for a specific board.

All other MEBA entry points are accessed *indirectly* through the `meba_inst_t.api` function pointers. All MEBA entry points have the MEBA instance pointer as the first parameter.

The MEBA layer will be executing in Linux userspace with the same privileges as the MEBA user. It will have access to the normal Linux usermode API's, as well as to MESA. In addition, the caller export a number of function pointers via the `meba_board_interface_t` structure.

The `meba_board_interface_t` structure defines API's for the following functions.

- Low-level switch register access
- I2C access (for a specific port)
- Trace output
- Configuration data access

The image below illustrates the overall system architecture. The *MEBA Callouts* are the function pointers offered by the application during MEBA initialization.

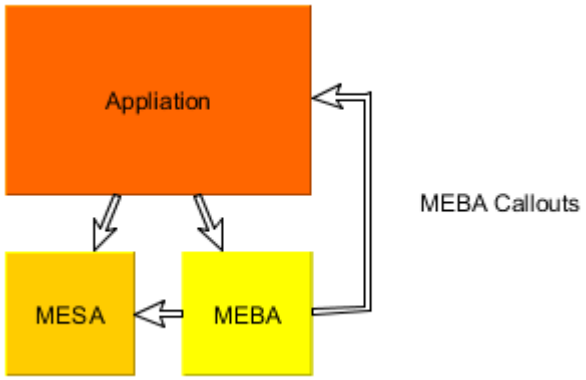

Figure 1. Overall system architecture

2.2. MEBA Initialization

The initialization sequnce for an application using MEBA must perform the following steps.

1. Use the `dlopen()` C library call to load the desired MEBA implementation shared library.
2. Use `dlsym()` to locate the `meba_initialize()`.
3. Initialize the *MEBA Callouts* structure and call the function pointer obtained in the previous step.
4. The meba_initialize() will now execute, performing the operations as describe in [meba_initialize()].
5. The `meba_initialize()` will return a MEBA instance handle. In case of errors, this handle will be `NULL`.
6. In case of successful execution, the handle will contain information to instantiate the MESA API. After MESA instantiation, both layers are operational. Specifically, the `meba_port_entry_get()` API can be used to establish a port map for MESA.
7. Should the application wish to perform graceful shutdown, the `meba_deinitialize()` can be used before exiting.

2.2.1. meba_initialize()

As described previously, this API is the only one exposed as a public symbol.

This function has the type `meba_initialize_t` and is defined as below.

```
typedef meba_inst_t (*meba_initialize_t)(size_t callouts_size,
                                         const meba_board_interface_t *callouts);
```

The function must allocate and initialize the instance structure - the callouts are copied to the allocated instance.

The return value should convey whether the expected hardware is present. During execution of this function the MESA layer is not (yet) initialized and must **not** be used.

If local state is required by the board implementation, the instance member pointer `inst.private_data` may be used to store this.

At the time of returning from this function, the `meba_inst_t.props` structure must be initialized in order for the MESA API to be instantiated. This implies initializing:

- Board name - textual representation of the target system
- Target chip (the MSCC switch 4-digit hexadecimal chip number)
- mux_mode (depending on target)

It is allowed that a specific board does not support all the features that the MEBA API defines. As such, a MEBA API function pointer may be set as `NULL`. This implies that the corresponding functionality is not available on the board in question. I.e. if a board does not support SFP's, the function pointers for SFP functions can simply be left `NULL`. Other non-NULL MEBA API's should be hooked up in the `meba_inst_t.api` in this function.

If a MEBA implementation support more than one board type, the appropriate board must be determined before returning from this function. This may involve direct access to chip registers via the `meba_board_interface_t` register access functions.

2.2.2. meba_deinitialize()

```
1 typedef void (*meba_deinitialize_t)(meba_inst_t inst);
```

The function must de-allocate the board instance and any other resources. Where required, the board hardware may also be brought to a given state.

2.3. Board capabilities

```
1 typedef uint32_t (*meba_capability_t)(meba_inst_t inst, int cap);
```

This function retrieve the board capabilities. The individual capabilities that can be queried are:

MEBA_CAP_POE	Power Over Ethernet
MEBA_CAP_1588_CLK_ADJ_DAC	AD5667 DAC used to Adjust the 1588 ref clock
MEBA_CAP_1588_REF_CLK_SEL	Set 1588 ref clock to different frequencies
MEBA_CAP_TEMP_SENSORS	Number of board temperature sensors
MEBA_CAP_BOARD_PORT_COUNT	Number of ports on board

MEBA_CAP_BOARD_PORT_MAP_COUNT	Number of ports in the port map (may be > MEBA_CAP_BOARD_PORT_COUNT if having MEP or Mirror loop ports
MEBA_CAP_LED_MODES	Number of alternate (port) LED modes
MEBA_CAP_DYING_GASP	Dying gasp power loss hardware support
MEBA_CAP_FAN_SUPPORT	Fan controller support
MEBA_CAP_LED_DIM_SUPPORT	LED dimming support
MEBA_CAP_BOARD_HAS_PCB107_CPLD	The PCB107 CPLD design was originally made specifically for PCB107 but is now being used as a general component on other boards as well
MEBA_CAP_PCB107_CPLD_CS_VIA_MUX	Some boards drive CS to PCB107 CPLD via a mux
MEBA_CAP_SYNCE_CLOCK_DPLL	DPLL number for the DPLL used for SYNCE
MEBA_CAP_SYNCE_CLOCK_OUTPUT_CNT	Number of clock output references, including 10G ports, which must be connected to the controller outputs
MEBA_CAP_SYNCE_PTP_CLOCK_OUTPUT	Clock output used for PTP independent Phase/Frequency adjustment
MEBA_CAP_SYNCE_HO_POST_FILTERING_BW	Default holdover post filtering

Number of ... denotes quantities, other are boolean values (true/false).

This API is mandatory to implement, as it disclose required information about the target board.

2.4. Board reset

The board reset API will bring certain board components to a well-defined state at command.

```
1 typedef mesa_rc (*meba_reset_t)(meba_inst_t inst, meba_reset_point_t reset);
```

The different reset points are:

MEBA_BOARD_INITIALIZE	Initialize Board
MEBA_PORT_RESET	Global Port Reset
MEBA_PORT_RESET_POST	Global Port Post Reset
MEBA_STATUS_LED_INITIALIZE	Status LED Initialize
MEBA_PORT_LED_INITIALIZE	Port LED Initialize
MEBA_FAN_INITIALIZE	Fan Initialize
MEBA_SENSOR_INITIALIZE	Sensors Initialize
MEBA_INTERRUPT_INITIALIZE	Interrupts Initialize
MEBA_SYNCE_DPLL_INITIALIZE	Initialize the SyncE DPLL i.e. setup dividers, references, monitors etc.

The MEBA implementation may at its discretion perform the reset operations all under `MEBA_BOARD_INITIALIZE`, or under the individual reset points. Its the responsibility of the MEBA user to issue the corresponding MEBA reset before calling MEBA functions for the associated functionality.

2.5. Port table

The port table entries describe capabilities of the switch ports, as well of how to control and access these and associated PHY's.

```
1  typedef struct {
2      mesa_port_map_t        map;           /**< Port map */
3      mesa_port_interface_t  mac_if;        /**< MAC interface */
4      meba_port_cap_t        cap;           /**< Port capabilities */
5      mesa_chip_no_t         poe_chip_port; /**< Chip port number (may be
different than poe channel number) */
6      mesa_bool_t            poe_support;   /**< PoE support for this port */
7  } meba_port_entry_t;
8
9  typedef mesa_rc (*meba_port_entry_get_t)(meba_inst_t inst,
10                                          mesa_port_no_t port_no,
11                                          meba_port_entry_t *entry);
```

After the switch API has been instantiated, the switch application will call this function to retrieve information of how to construct the switch API port map, and to know what the physical characteristics of the port is.

In the `map` structure you typically define these members:

chip_port
The physical chip port for the port. -1 if not used.

miim_controller
MII management controller (`MESA_MIIM_CONTROLLER_NONE` for SFP).

miim_addr
If MIIM is used (see above).

2.5.1. MAC Interface

The `mac_if` value is typically set to one of these values:

MESA_PORT_INTERFACE_SERDES
MESA_PORT_INTERFACE_SGMII
MESA_PORT_INTERFACE_QSGMII
MESA_PORT_INTERFACE_SFI

Refer to [MESA] for details on the referenced MESA types.

2.5.2. Port Capabilities

The `cap` value reflect the port capabilities, which is defined by these bitmasks:

MEBA_PORT_CAP_NONE	0x00000000	No capabilities
MEBA_PORT_CAP_AUTONEG	0x00000001	Auto negotiation
MEBA_PORT_CAP_10M_HDX	0x00000002	10 Mbps, half duplex
MEBA_PORT_CAP_10M_FDX	0x00000004	10 Mbps, full duplex
MEBA_PORT_CAP_100M_HDX	0x00000008	100 Mbps, half duplex
MEBA_PORT_CAP_100M_FDX	0x00000010	100 Mbps, full duplex
MEBA_PORT_CAP_1G_FDX	0x00000020	1 Gbps, full duplex
MEBA_PORT_CAP_2_5G_FDX	0x00000040	2.5 Gbps, full duplex
MEBA_PORT_CAP_5G_FDX	0x00000080	5Gbps, full duplex
MEBA_PORT_CAP_10G_FDX	0x00000100	10Gbps, full duplex
MEBA_PORT_CAP_FLOW_CTRL	0x00001000	Flow control

MEBA_PORT_CAP_COPPER	0x00002000	Copper media
MEBA_PORT_CAP_FIBER	0x00004000	Fiber media
MEBA_PORT_CAP_DUAL_COPPER	0x00008000	Dual media, copper preferred
MEBA_PORT_CAP_DUAL_FIBER	0x00010000	Dual media, fiber preferred
MEBA_PORT_CAP_SD_ENABLE	0x00020000	Signal Detect enabled
MEBA_PORT_CAP_SD_HIGH	0x00040000	Signal Detect active high
MEBA_PORT_CAP_SD_INTERNAL	0x00080000	Signal Detect select internal
MEBA_PORT_CAP_XAUI_LANE_FLIP	0x00200000	Flip the XAUI lanes
MEBA_PORT_CAP_VTSS_10G_PHY	0x00400000	Connected to VTSS 10G PHY
MEBA_PORT_CAP_SFP_DETECT	0x00800000	Auto detect the SFP module
MEBA_PORT_CAP_STACKING	0x01000000	Stack port candidate
MEBA_PORT_CAP_DUAL_SFP_DETECT	0x02000000	Auto detect the SFP module for dual media
MEBA_PORT_CAP_SFP_ONLY	0x04000000	SFP only port (not dual media)
MEBA_PORT_CAP_SERDES_RX_INVERT	0x10000000	Serdes RX signal is inverted
MEBA_PORT_CAP_SERDES_TX_INVERT	0x20000000	Serdes TX signal is inverted
MEBA_PORT_CAP_INT_PHY	0x40000000	Connected to internal PHY

For convenience, these shorthands are defined using the above:

MEBA_PORT_CAP_HDX	Half duplex

MEBA_PORT_CAP_TRI_SPEED_FDX	Tri-speed port full duplex only
MEBA_PORT_CAP_TRI_SPEED	Tri-speed port, both full and half duplex
MEBA_PORT_CAP_1G_PHY	1G PHY present
MEBA_PORT_CAP_TRI_SPEED_COPPER	Tri-speed port copper only
MEBA_PORT_CAP_TRI_SPEED_FIBER	Tri-speed port fiber only
MEBA_PORT_CAP_TRI_SPEED_DUAL_COPPER	Tri-speed port both fiber and copper. Copper preferred
MEBA_PORT_CAP_TRI_SPEED_DUAL_FIBER	Tri-speed port both fiber and copper. Fiber preferred
MEBA_PORT_CAP_ANY_FIBER	Any fiber mode
MEBA_PORT_CAP_SPEED_DUAL_ANY_FIBER_FIXED_SPEED	Any fiber mode, but auto detection not supported
MEBA_PORT_CAP_SPEED_DUAL_ANY_FIBER	Any fiber mode, auto detection supported

MEBA_PORT_CAP_TRI_SPEED_DUAL_ANY_FIBER	Copper 5 Fiber mode, auto detection supported
MEBA_PORT_CAP_TRI_SPEED_DUAL_ANY_FIBER_FIXED_SFP_SPEED	Copper & Fiber mode, but SFP auto detection not supported
MEBA_PORT_CAP_DUAL_FIBER_1000X	1000Base-X fiber mode
MEBA_PORT_CAP_SFP_1G	SFP fiber port 100FX/1G with auto negotiation and flow control
MEBA_PORT_CAP_SFP_2_5G	SFP fiber port 100FX/1G/2.5G with auto negotiation and flow control
MEBA_PORT_CAP_SFP_SD_HIGH	SFP fiber port 100FX/1G/2.5G with auto negotiation and flow control, signal detect high
MEBA_PORT_CAP_2_5G_TRI_SPEED_FDX	100M/1G/2.5G Tri-speed port full duplex only

MEBA_PORT_CAP_2_5G_TRI_SPEED	100M/1G/2.5G Tri-speed port, all full duplex and 100M half duplex
MEBA_PORT_CAP_2_5G_TRI_SPEED_COPPER	100M/1G/2.5G Tri-speed port copper only

This API is mandatory to implement, as it disclose required information about the target board.

2.6. Sensor support

The MEBA sensor support 2 types of sensors.

- Board temperature sensor (`MEBA_CAP_TEMP_SENSORS` capability)
- Port temperature sensor (`MEBA_CAP_BOARD_PORT_COUNT` capability)

Access to the sensor data is handled by a single API with the following signature:

```
typedef enum {
    MEBA_SENSOR_PORT_TEMP,          /**< Port temperature sensor */
    MEBA_SENSOR_BOARD_TEMP,         /**< Board/chassis temperature sensor */
} meba_sensor_t;

typedef mesa_rc (*meba_sensor_get_t)(meba_inst_t inst,
                                     meba_sensor_t type,
                                     int six,
                                     int *value);
```

The `six` parameter is the sensor index for the given type. Upon a successful call, the `value` contains the value of the sensor.

The unit for temperature sensors is in Celsius (signed).

2.7. SFP Support

If the board does not support SFP's, all API's in this section can be omitted.

The support for SFP's span the following areas.

- Generic I2C access to a SFP on a given port
- SFP insertion state
- Detailed SFP status
- Port administrative control

2.7.1. SFP I2C access

The generic I2C is implemented by the following MEBA entrypoint.

```
typedef mesa_rc (*meba_sfp_i2c_xfer_t)(meba_inst_t inst,
                                       mesa_port_no_t port_no,
                                       mesa_bool_t write,
                                       uint8_t i2c_addr,
                                       uint8_t addr,
                                       uint8_t *data,
                                       uint8_t cnt,
                                       mesa_bool_t word_access);
```

The I2C transfer can either be handled by MESA or by calling the `meba_inst_t.api.i2c_read/write` functions, depending on which I2C controller the SFP is attached to.

2.7.2. SFP insertion state

SFP insertion state is returned by the following MEBA API.

```
typedef mesa_rc (*meba_sfp_insertion_status_get_t)(meba_inst_t inst,
                                                   mesa_port_list_t *present);
```

The `present` port list will contain all (SFP) ports where a SFP has been detected.

2.7.3. SFP detailed state

SFP detailed state for a specific port is returned by the following MEBA API.

```
typedef struct {
    mesa_bool_t  tx_fault;         /**< TxFault */
    mesa_bool_t  los;              /**< Loss Of Signal  */
    mesa_bool_t  present;          /**< SFP module present  */
} meba_sfp_status_t;

typedef mesa_rc (*meba_sfp_status_get_t)(meba_inst_t inst,
                                         mesa_port_no_t port_no,
                                         meba_sfp_status_t *status);
```

If a port does not support SFP the API should return an error code. Generally, the API can only be expected to succeed if the port has previously signaled a SFP being inserted. (See the previous section).

2.7.4. Port administrative control

In order to control any additional operations necessary when enabling/disabling a port, such as controlling transmit on an SFP, or port specific operations, the following MEBA API is defined:

```
typedef struct {
    mesa_bool_t             enable; /**< Admin enable/disable */
} meba_port_admin_state_t;

typedef mesa_rc (*meba_port_admin_state_set_t)(meba_inst_t inst,
                                               mesa_port_no_t port_no,
                                               const meba_port_admin_state_t
*state);
```

The API should should be called by the application when a port changes administrative state (being enabled or disabled), and the MEBA implementation should perform any operations needed to enable/disable the physical interface beyond what is controlled by normal MESA port control.

2.8. LED support

Each of the API's in this section can be omitted on an individual basis.

The LED support in MEBA is defined by the following functions.

- Board status LED
- Port status LED
- LED intensity control

2.8.1. Board status LED

Th API to control board LED's are defined as follows.

```
typedef enum {
    MEBA_LED_TYPE_FRONT,         /**< Front LED main state */
} meba_led_type_t;

typedef enum {
    MEBA_LED_COLOR_OFF,          /**< No LED */
    MEBA_LED_COLOR_GREEN,        /**< Green LED */
    MEBA_LED_COLOR_RED,          /**< Red LED */
    MEBA_LED_COLOR_YELLOW,       /**< Yellow LED */
    MEBA_LED_COLOR_COUNT,        /**< Number of LED colors */
} meba_led_color_t;

typedef mesa_rc (*meba_status_led_set_t)(meba_inst_t inst,
                                         meba_led_type_t type,
                                         meba_led_color_t color);
```

Currently the only board LED supported is `MEBA_LED_TYPE_FRONT`. Blinking and alternating different colors should be done by the MEBA user by calling the API at regular intervals.

If a color is unsupported by the board, the MEBA implementation may choose to return an error or select an alternate color.

2.8.2. Port status LED

The following MEBA API should be called by the application to update the port LED state. The application should either update at regular intervals, or when it detects at change in port state.

```
typedef mesa_rc (*meba_port_led_update_t)(meba_inst_t inst,
                                          mesa_port_no_t port_no,
                                          const mesa_port_status_t *status,
                                          const mesa_port_counters_t *counters,
                                          const meba_port_admin_state_t *state);
```

The MEBA implementation should update the port LED according to current state. If it supports visualizing collision counters or other activity data, it can use the counters provided, as well as other state data provided.

The port LED displayed may also depend on the current *port LED mode*, if the board supports this. (The `MEBA_CAP_LED_MODES` capability) The port LED mode is controlled by the following MEBA API:

```
typedef mesa_rc (*meba_led_mode_set_t)(meba_inst_t inst, uint32_t mode);
```

This MEBA API will be called when management operations change the LED mode, and the MEBA implementation should update the port LED's to reflect current state.

If `MEBA_CAP_LED_MODES` is zero or 1, this API should neither be implemented nor called.

2.8.3. LED intensity control

If the board support LED intensity control (LED dimming), the following API can be used to control it. The API should only be called if the MEBA_CAP_LED_DIM_SUPPORT is non-zero.

```
typedef mesa_rc (*meba_led_intensity_set_t)(meba_inst_t inst,
                                            mesa_phy_led_intensity intensity);
```

The API is supposed to control all LED's capable of dimming (as a whole). The `intensity` parameter is a percentage, ranging from 0 to 100.

2.9. Fan support

If the board does not support FAN control, all API's in this section can be omitted, and they should be assumed to be available only is the MEBA_CAP_FAN_SUPPORT capabilty is non-zero.

MEBA fan support specify two API's:

- Fan parameter retrieval
- Fan configuration retrieval

2.9.1. FAN parameter retrieval

To expose the characteristics of the board FAN, the following API is used.

```
typedef struct {
    /** The duration of time to be before going to low level (seconds) */
    uint8_t start_time;

    /** The level to be at before going to low level (pct) */
    uint8_t start_level;

    /** The min level supported by fan (pct) */
    uint8_t min_pwm;
} meba_fan_param_t;

typedef mesa_rc (*meba_fan_param_get_t)(meba_inst_t inst,
                                        meba_fan_param_t *param);
```

The application can use this information to spin up, and drive the board fan according to the fan operational requirements.

2.9.2. Fan configuration retrieval

The fan is assumed to be controlled by MESA. In order to initialize the fan controller, the MESA fan configuration must be obtained. This MEBA API is designed to provide the MESA fan configuration structure.

```
typedef mesa_rc (*meba_fan_conf_get_t)(meba_inst_t inst,
                                       mesa_fan_conf_t *conf);
```

The application can use MESA to configure and control the fan speed. The `meba_fan_param_t` data provide guides on how the fan should be operated. Fan speed control normally is coupled with temperature sensor support.

2.10. Interrupt support

If the board does not support interrupts, all API's in this section can be omitted.

Interrupt support use three API's:

- Interrupt event enable
- Interrupt handler
- Interrupt requestor

The interrupt events are defined by the `meba_event_t` enumeration. These values denote the **decoded** interrupts.

The interrupt events are signalled to the application by the interrupt handler. The interrupt handler is invoked by the application when a specific interrupt signal - `mesa_irq_t` is detected.

The interrupt requestor convey to the application which interrupt signals MEBA wants to handle (decode). It is the responsibility of the application to configure interrupts and call the MEBA interrupt when the OS signal the interrupt.

2.10.1. Interrupt event enable

The interrupt enable API is use to enable or disable one specific interrupt event, for example the `MEBA_EVENT_PUSH_BUTTON` event. The event is enabled or disabled at the hardware level. If a specific event is not supported by the particular MEBA implementation `MESA_RC_NOT_IMPLEMENTED` should be returned.

```
typedef mesa_rc (*meba_event_enable_t)(meba_inst_t inst,
                                       meba_event_t event,
                                       mesa_bool_t enable);
```

2.10.2. Interrupt handler

The interrupt handler must decode all the supported events for a given interrupt signal as per the following steps.

1. Identify all interrupt events for a given interrupt signal.
2. Disable the interrupt event (source).
3. Determine a possible instance number(s) (port, etc.)
4. Call the event sink `signal_notifier`, signalling the interrupt event type and (all possible) instance numbers.
5. If no interrupt sources were seen an error must be returned.

```
typedef mesa_rc (*meba_irq_handler_t)(meba_inst_t inst,
                                      mesa_irq_t chip_irq,
                                      meba_event_signal_t signal_notifier);
```

2.10.3. Interrupt requestor

The interrupt requestor API must must return `MESA_RC_OK` if the interrupt specified by the `chip_irq` parameter is handled by the MEBA layer, otherwise `MESA_RC_NOT_IMPLEMENTED`.

The application can use the API (at startup) to determine which interrupt signals should be claimed from the OS.

```
typedef mesa_rc (*meba_irq_requested_t)(meba_inst_t inst,
                                        mesa_irq_t chip_irq);
```

2.11. SyncE support

The support for SyncE in MEBA is at this time limited to the following functions:

- Detecting if a DPLL is present in the system and in case a DPLL is present also detecting the type of the DPLL.
- In the case that a DPLL is present and it is controlled via SPI, MEBA supplies a function for accessing the DPLL via SPI.
- Supplying the application with a graph describing the topology of the SyncE hardware. The SyncE board graph is read from MEBA by the application which then uses it for deriving how to set up muxes, dividers, DPLL references etc.

2.11.1. DPLL Detection

For the detection of the DPLL present in the system, MEBA provides the following function:

```
mesa_rc meba_synce_spi_if_get_dpll_type(meba_inst_t inst,
                                        meba_synce_clock_hw_id_t *dpll_type);
```

This function is called from the synce_dpll module in the application as part of its initialization. In some cases, the function is also used by MEBA itself to determine which version of a board graph to return when differences exist for different DPLLs.

If no DPLL is present, the function will return MESA_RC_ERROR.

2.11.2. DPLL SPI Interface

At present, the actual setup and control of the DPLL is done by the application. To facilitate this MEBA provides the following function for accessing the DPLL via SPI:

```
mesa_rc meba_synce_spi_if_spi_transfer(meba_inst_t inst,
                                       uint32_t buflen,
                                       const uint8_t *tx_data,
                                       uint8_t *rx_data);
```

Note: In the case of the ServalT built-in DPLL, the DPLL is controlled directly via the Omega API as in this case the DPLL is not connected via SPI.

2.11.3. SyncE Board Graph

A board graph consists of nodes of various types (e.g. ports, muxes, dplls) connected to each other by means of edges. See the figure named Serval2 lite SyncE Board Graph below for an example of a graph for a Serval2 Lite board:

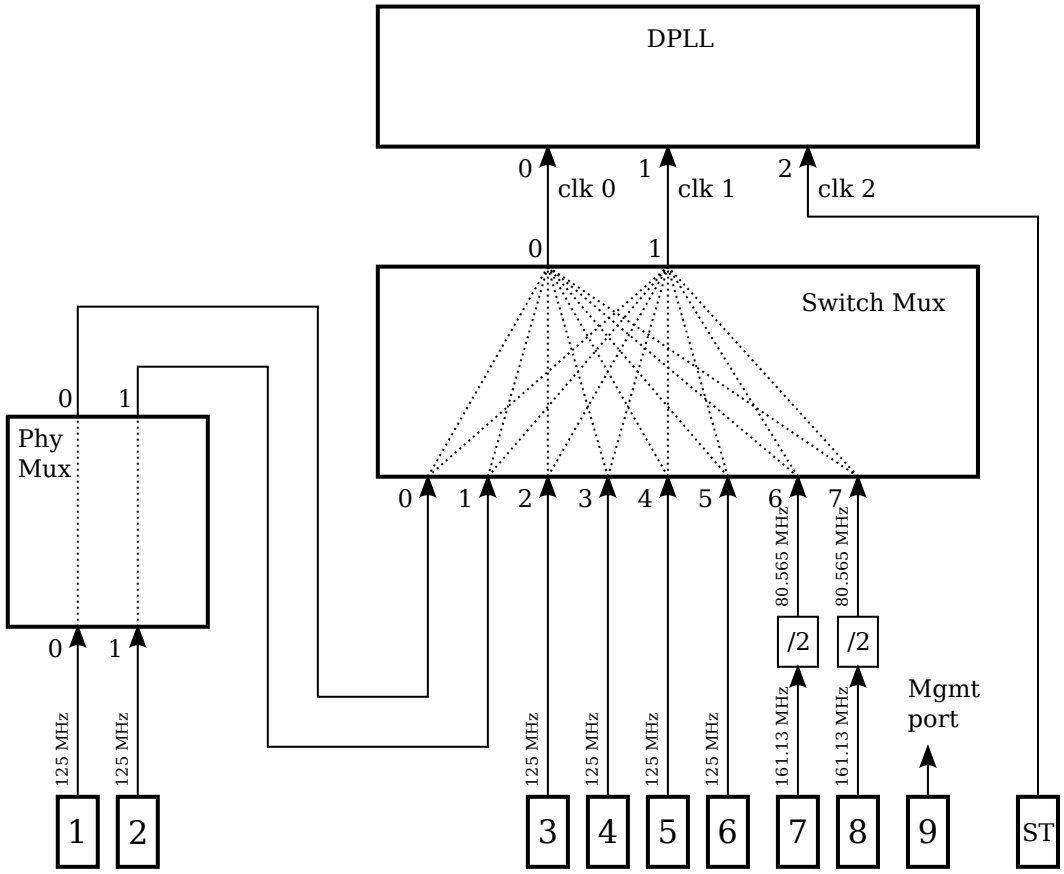

Figure 2. Serval2 lite SyncE Board Graph

The data types used for specifying the board graph are defined in the file (within the mesa source tree):

```
./meba/include/mscc/ethernet/board/api/synce.h
```

The graph itself (the array synce_graph_elements_serval2_lite_board) is defined in the file:

```
./meba/src/servalt/synce.c
```

Basically, an edge is just a connection from an output of a source node to an input of a destination node. All edges connecting nodes generally have the same type (defined by the structure meba_synce_graph_element_t with the type member set to MEBA_SYNCE_GRAPH_ELEMENT_TYPE_CONNECTION). The src and dst members specify the start and end points of the edge.

A node is characterized by the following features:

- It's type (represented by the type member of meba_synce_terminal_t)
- It's device ID (represented by the dev_id member of meba_synce_terminal_t)
- A number of input ports
- A number of output ports

All nodes irrespective of their type are represented by one or more structures of the type meba_synce_terminal_t (one per input/output terminal of the node).

For each input/output port a seperate meba_synce_terminal_t must be defined with the idx value representing the port number. For output ports, idx is generally the port number. For input ports, idx must have the constant MESA_SYNCE_DEV_INPUT added/OR'ed to the port number to signify that it is an input port.

Some nodes (generally ethernet ports or station clocks) only have a single output port. Examples of such nodes are the nodes labeled 1 to 9 and ST in the figure named Serval2 lite SyncE Board Graph above. In the synce.c source file those nodes are:

```
#define eth_port_0           MESA_SYNCE_DEV_PORT(    0, 0)
#define eth_port_1           MESA_SYNCE_DEV_PORT(    1, 0)
#define eth_port_2           MESA_SYNCE_DEV_PORT(    2, 0)
#define eth_port_3           MESA_SYNCE_DEV_PORT(    3, 0)
#define eth_port_4           MESA_SYNCE_DEV_PORT(    4, 0)
#define eth_port_5           MESA_SYNCE_DEV_PORT(    5, 0)
#define eth_port_6           MESA_SYNCE_DEV_PORT(    6, 0)
#define eth_port_7           MESA_SYNCE_DEV_PORT(    7, 0)
#define station_clock_port_0 MESA_SYNCE_DEV_CLOCK_IN(9, 0)
```

Other nodes (generally DPLLs) only have inputs. An example of such a node is the node labeled DPLL in the figure named Serval2 lite SyncE Board Graph above. In the synce.c source file this corresponds to:

```
#define dpll_port_0 MESA_SYNCE_DEV_DPLL(500, MESA_SYNCE_DEV_INPUT | 0)
#define dpll_port_1 MESA_SYNCE_DEV_DPLL(500, MESA_SYNCE_DEV_INPUT | 1)
#define dpll_port_7 MESA_SYNCE_DEV_DPLL(500, MESA_SYNCE_DEV_INPUT | 7)
```

The remaining nodes that have both inputs and outputs are either buffers or muxes. When more inputs and outputs exist, any input can connect to any output of the same node. That is, the node can be considered a mux with full connectivity. A buffer can be considered a special case of mux with only one input and one output.

In cases where a mux has less than full connectivity, this can be modelled using edges with the type member of meba_synce_graph_element_t set to MESA_SYNCE_GRAPH_INVALID_CONNECTION. If such an invalid connection is defined from an input of a node to an output of the same node, the connection will be considered invalid.

In the synce.c source two muxes are defined. The Phy mux is restricted so the only connectivity allowed is from input 0 to output 0 and from input 1 to output 1. This is not expressed in the definition of the Phy mux itself. Rather this is expressed in the declaration of edges (see further below). In the synce.c source file the Phy Mux is defined as in the following:

```
#define phy_mux_port_in_0  MESA_SYNCE_DEV_MUX_PHY(300, MESA_SYNCE_DEV_INPUT | 0)
#define phy_mux_port_in_1  MESA_SYNCE_DEV_MUX_PHY(300, MESA_SYNCE_DEV_INPUT | 1)
#define phy_mux_port_out_0 MESA_SYNCE_DEV_MUX_PHY(300, 0)
#define phy_mux_port_out_1 MESA_SYNCE_DEV_MUX_PHY(300, 1)
```

The switch mux that has full connectivity is defined in the synce.c source file as follows:

```
#define switch_mux_port_in_0  \
    MESA_SYNCE_DEV_MUX_SWITCH(400, MESA_SYNCE_DEV_INPUT | 0)
#define switch_mux_port_in_1  \
    MESA_SYNCE_DEV_MUX_SWITCH(400, MESA_SYNCE_DEV_INPUT | 1)
#define switch_mux_port_in_2  \
    MESA_SYNCE_DEV_MUX_SWITCH(400, MESA_SYNCE_DEV_INPUT | 2)
#define switch_mux_port_in_3  \
    MESA_SYNCE_DEV_MUX_SWITCH(400, MESA_SYNCE_DEV_INPUT | 3)
#define switch_mux_port_in_4  \
    MESA_SYNCE_DEV_MUX_SWITCH(400, MESA_SYNCE_DEV_INPUT | 4)
#define switch_mux_port_in_5  \
    MESA_SYNCE_DEV_MUX_SWITCH(400, MESA_SYNCE_DEV_INPUT | 5)
#define switch_mux_port_in_6  \
    MESA_SYNCE_DEV_MUX_SWITCH(400, MESA_SYNCE_DEV_INPUT | 6)
#define switch_mux_port_in_7  \
    MESA_SYNCE_DEV_MUX_SWITCH(400, MESA_SYNCE_DEV_INPUT | 7)
#define switch_mux_port_in_8  \
    MESA_SYNCE_DEV_MUX_SWITCH(400, MESA_SYNCE_DEV_INPUT | 8)
#define switch_mux_port_out_0 \
    MESA_SYNCE_DEV_MUX_SWITCH(400, 0)
#define switch_mux_port_out_1 \
    MESA_SYNCE_DEV_MUX_SWITCH(400, 1)
```

The two elements labeled **/2** in the board graph figure are "virtual" buffers that do not actually exist in reality. They have been inserted to make it possible to specify that the recovered frequency of 161.13 MHz from ports 7 and 8 should be divided by 2 to make 80.565 MHz before reaching the input of the DPLL.

These virtual buffers are declared as follows in the synce.c file:

```
#define divider_eth_port_6_in  \
    MESA_SYNCE_DEV_DIVIDER(100, MESA_SYNCE_DEV_INPUT | 0)
#define divider_eth_port_6_out \
    MESA_SYNCE_DEV_DIVIDER(100, 0)
#define divider_eth_port_7_in  \
    MESA_SYNCE_DEV_DIVIDER(200, MESA_SYNCE_DEV_INPUT | 0)
#define divider_eth_port_7_out \
    MESA_SYNCE_DEV_DIVIDER(200, 0)
```

This then leads to the board graph array:

```
static const meba_synce_graph_element_t
synce_graph_elements_serval2_lite_board[] = {
    // type                     source                  destination
    MESA_SYNCE_GRAPH_CONNECTION(eth_port_0,             phy_mux_port_in_0),
    MESA_SYNCE_GRAPH_CONNECTION(eth_port_1,             phy_mux_port_in_1),
    MESA_SYNCE_GRAPH_CONNECTION(phy_mux_port_out_0,     switch_mux_port_in_0),
    MESA_SYNCE_GRAPH_CONNECTION(phy_mux_port_out_1,     switch_mux_port_in_1),
    MESA_SYNCE_GRAPH_CONNECTION(eth_port_2,             switch_mux_port_in_2),
    MESA_SYNCE_GRAPH_CONNECTION(eth_port_3,             switch_mux_port_in_3),
    MESA_SYNCE_GRAPH_CONNECTION(eth_port_4,             switch_mux_port_in_4),
    MESA_SYNCE_GRAPH_CONNECTION(eth_port_5,             switch_mux_port_in_5),
    MESA_SYNCE_GRAPH_CONNECTION(eth_port_6,             divider_eth_port_6_in),
    MESA_SYNCE_GRAPH_CONNECTION(eth_port_7,             divider_eth_port_7_in),
    MESA_SYNCE_GRAPH_CONNECTION(divider_eth_port_6_out, switch_mux_port_in_6),
    MESA_SYNCE_GRAPH_CONNECTION(divider_eth_port_7_out, switch_mux_port_in_7),
    MESA_SYNCE_GRAPH_CONNECTION(switch_mux_port_out_0,  dpll_port_0),
    MESA_SYNCE_GRAPH_CONNECTION(switch_mux_port_out_1,  dpll_port_1),
    MESA_SYNCE_GRAPH_CONNECTION(station_clock_port_0,   dpll_port_2),
    MESA_SYNCE_GRAPH_INVALID_CONNECTION(phy_mux_port_in_0, phy_mux_port_out_1),
    MESA_SYNCE_GRAPH_INVALID_CONNECTION(phy_mux_port_in_1, phy_mux_port_out_0)
};
```

The board graph array lists all the edges making up the board graph. The information that still needs to be specified is the assignment of clocks to the input references of the DPLL and the frequencies as specified in the graph. This is done by means of a seperate array of attributes in synce.c as follows:

```
static const meba_synce_terminal_attr_t attr_serval2_lite_board[] = {
    //              device                attr-type           attr-value
    MESA_SYNCE_ATTR(dpll_port_0,          MEBA_ATTR_CLOCK_ID, 1),
    MESA_SYNCE_ATTR(dpll_port_1,          MEBA_ATTR_CLOCK_ID, 2),
    MESA_SYNCE_ATTR(dpll_port_7,          MEBA_ATTR_CLOCK_ID, 3),
    MESA_SYNCE_ATTR(switch_mux_port_in_6, MEBA_ATTR_FREQ, \
                    MEBA_SYNCE_CLOCK_FREQ_80_565MHZ),
    MESA_SYNCE_ATTR(switch_mux_port_in_7, MEBA_ATTR_FREQ, \
                    MEBA_SYNCE_CLOCK_FREQ_80_565MHZ),
    MESA_SYNCE_ATTR(eth_port_0,           MEBA_ATTR_FREQ, \
                    MEBA_SYNCE_CLOCK_FREQ_125MHZ),
    MESA_SYNCE_ATTR(eth_port_1,           MEBA_ATTR_FREQ, \
                    MEBA_SYNCE_CLOCK_FREQ_125MHZ),
    MESA_SYNCE_ATTR(eth_port_2,           MEBA_ATTR_FREQ, \
                    MEBA_SYNCE_CLOCK_FREQ_125MHZ),
    MESA_SYNCE_ATTR(eth_port_3,           MEBA_ATTR_FREQ, \
                    MEBA_SYNCE_CLOCK_FREQ_125MHZ),
    MESA_SYNCE_ATTR(eth_port_4,           MEBA_ATTR_FREQ, \
                    MEBA_SYNCE_CLOCK_FREQ_125MHZ),
    MESA_SYNCE_ATTR(eth_port_5,           MEBA_ATTR_FREQ, \
                    MEBA_SYNCE_CLOCK_FREQ_125MHZ),
    MESA_SYNCE_ATTR(eth_port_6,           MEBA_ATTR_FREQ, \
                    MEBA_SYNCE_CLOCK_FREQ_161_13MHZ),
    MESA_SYNCE_ATTR(eth_port_7,           MEBA_ATTR_FREQ, \
                    MEBA_SYNCE_CLOCK_FREQ_161_13MHZ),
};
```

The board graph array and the attributes array are wrapped together in one data structure defined as:

```
/** A data structure for representing the "clock" graph on the board. */
typedef struct {
    /** Number of elements in the graph. */
    uint32_t  graph_length;

    /** Array of graph elements. */
    const meba_synce_graph_element_t *graph;

    /** Number of attributes */
    uint32_t  attr_length;

    /** Array of attribues */
    const meba_synce_terminal_attr_t *attr;

} meba_synce_graph_t;
```

The application can fetch a pointer to this structure by means of the following function:

```
mesa_rc meba_synce_graph_get(meba_inst_t inst,
                             const meba_synce_graph_t **const g);
```

2.11.4. Advanced Example of a Board Graph

The figure named Jr2 (24 ports + 4*10GB) SyncE Board Graph below shows a more advanced example of a graph for a Jaguar2 (24 ports) board:

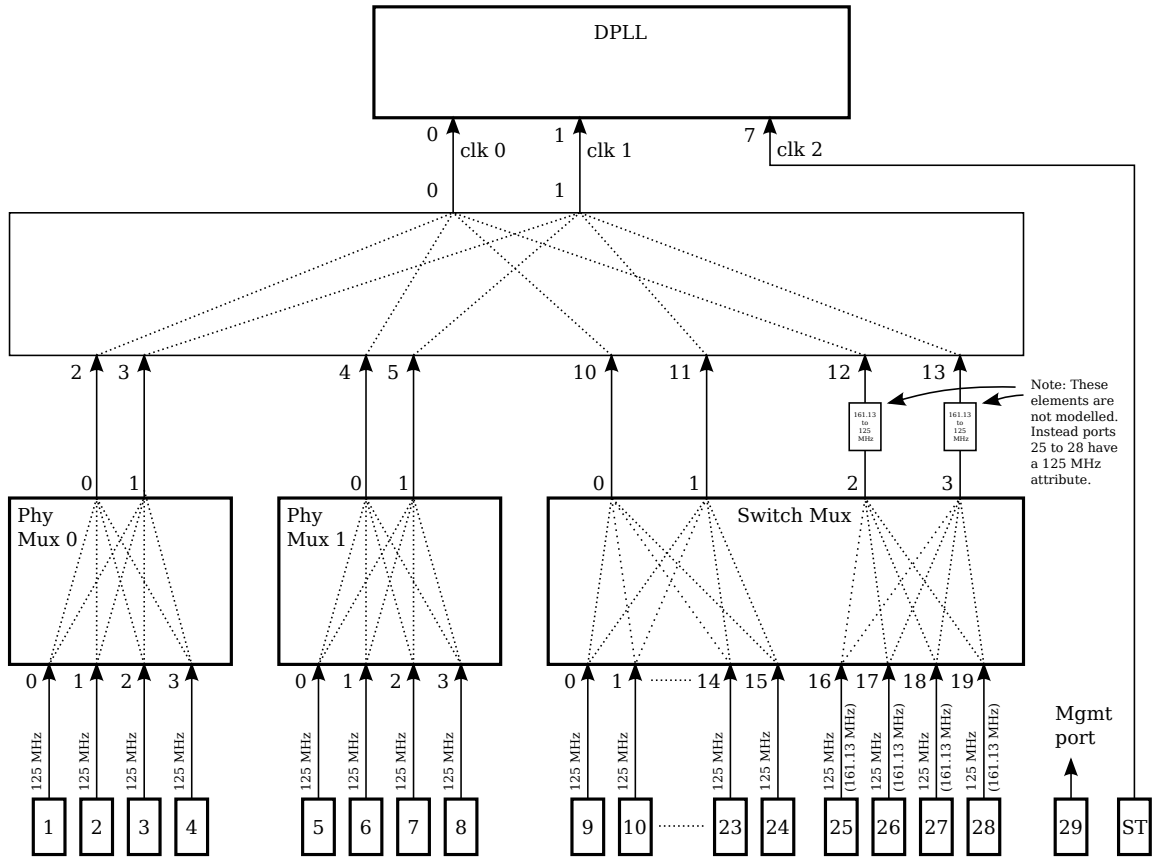

*Figure 3. Jr2 (24 ports + 4*10GB) SyncE Board Graph*

The graph (the array synce_graph_elements_jr2_24_board) and the associated attributes (the array attr_jr2_24_board together with either attr_defaults_dpll or attr_zarlink_dpll) are defined in the file (within the mesa source tree):

```
./meba/src/jr2/synce.c
```

Although this graph is larger, has more muxes and more connections, it is actually quite straight forward to understand. What does need a little bit of explanation though are the two elements sitting between the ports 2 and 3 of the switch mux and ports 12 and 13 of the board mux.

These two elements are frequency converters that exist on the Jr2 (24 ports) board in order to convert the recovered clock frequencies from Ethernet ports 25-28 to 125 MHz. These frequency converters are not modelled in the board graph. Rather a 125 MHz frequency attribute has been set on ports 25-28 although these ports are actually delivering a 161.13 MHz recovered clock.

3. MEBA Cookbook

3.1. Adding MEBA Support to a Board.

When adding MEBA support for a new board type, there are generally two different approaches:

- Change an existing MEBA implementation for a board resembling your own.
- Add a new MEBA implementation.

It is a matter of style personal preferences whether you choose one or the other. Normally it makes sense to modify an existing MEBA implementation if your board only is a *little* different. If there is a bigger difference in features and how the board works, you will probably be better off by having separate implementations.

If you decide to extend an existing MEBA implementation, you should also consider whether you:

- Abandon the original board support, i.e. just change the code to only support your board.
- *Add* support for your board by extending the code to perform different operations where needed. This will require the code to keep track of which board it is operating on. This will require the code to either be able to tell the boards apart by probing the hardware, or by means of configuration data (See [Using board configuration]). Examples of both can be seen in the MEBA support for the Microsemi reference boards.

It should be noted that if you are aiming to support more than one of your own (similar) boards, you should use the latter approach. And if you design your boards so they are easy to tell apart, the board probing will be simple to implement. (Notice the constraint that MESA is *not* available during the probing - `meba_initialize()`).

3.2. Adding MEBA a library

If you end up deciding to add a separate, new MEBA implementation file, you will first need to add this library to the MEBA main CMake build make file.

This file is `…/meba/CMakeLists.txt`. Around line 50 you will find a line starting with `MEBA_LIB`.

```
MEBA_LIB(serval1 caracal jr2 servalt ocelot)
```

Between the parentheses, you should add the name of your board, for example `myname` - like below.

```
MEBA_LIB(serval1 caracal jr2 servalt ocelot myname)
```

This will add a CMake build option called `BUILD_MEBA_myname`. When you are invoking the MESA top level build for your target, you can now add a `-DBUILD_MEBA_myname=on` argument. This will trigger building your MEBA shared library. (If you are adding this

for use by the WebStaX application, the build configuration option `Custom/Meba` can be set to the name of your MEBA library, which will cause the above CMake build option to be enabled.)

3.3. Development workflow

Then adding MEBA support for your board, you can follow the outline below.

1. Decide whether you are starting a new MEBA implementation file - or extending an existing implementation.
2. If you are adding a new implementation, make the changes described in the previous section.
3. Create a new directory, `src/myname` with an empty source file, `meba.c`. (If you are using WebStaX, also make use of the `Custom/Meba` setting in your build configuration file.)
4. Verify you can build your MEBA library.
5. Extend the empty MEBA file to include the following (use one of the MEBA libraries for the Microsemi boards as a reference):
   a. A `meba_initialize` function, setting up target switch and the MEBA API function pointers below:
   b. A `myname_capability` function. Be sure the `MEBA_CAP_BOARD_PORT_COUNT` value is accurate.
   c. A `meba_reset` function, handling only the most basic `MEBA_BOARD_INITIALIZE` code your board needs.
   d. A `meba_port_entry_get` function, capable of setting up at least some valid port entries. If your board support different port types, setup only the basic ones to start with.
6. Ensure the *barebone* MEBA library compiles.
7. Try getting the MEBA library running on your target system. You can make use of the debug output function and the `T_D()` / `T_I()` etc. macros in your code. With WebStaX, the MEBA output is controlled by the `main.board` trace module level.

Once you have the basic port functions running, you can continue adding support for the complete port map, LED's, temperature sensors, etc. according to the capabilities of your board.

If you are implementing synce support, add a separate file for this functional group, and setup the API group pointer using the `meba_synce_get()` function. Refer to the MEBA synce documentation and the reference implemetations.

4. Advanced topics

4.1. Using board configuration.

In some cases, it case be useful to be able to retrieve configuration from the embodying software application to control features in MEBA. For example your board could have a feature to use an alternate port table layout, which are not possible to detect by probing the board hardware.

To support this or similar feature, the `api.meba_conf_get` function can be used to querying configuration data. The application can then implement access to configuration values for a specific name. The MEBA library is free to define the naming, for example `"myname.port_cfg"`.

The Microsemi MEBA implementations use `"target"` to define the target API switch type, and `"type"` in some cases where a MEBA library support more than one board type, but probing to distinguish them is not feasible.

When using board configuration, it is recommended to have some sane default values where possible, since the configuration retreival can fail (return failure).

5. References

[]
ug1070 Microsemi Ethernet Switch API

[]
ug1068 SW Introduction to WebStaX under Linux

[]
AN1163 Linux Customizations