VITESSE®

Vitesse API Software

Application Note

VPPD-02870 / AN1007
Revision 1.8
March 2015

Vitesse

www.vitesse.com

TERMS OF USE

The information provided by Vitesse Semiconductor Corporation ("Vitesse") in this document pursuant to these terms ("Agreement") is intended for illustrative purposes only. All information provided herein is subject to change at any time without notice. The information provided, including but not limited to Sample Code ("Software"), is protected by United States and other applicable copyright laws and international treaties. Vitesse does not grant You any license, explicitly or implicitly, under any trademark, patent, copyright, mask work protection right, trade secret or any other intellectual property right.

ALL INFORMATION, INCLUDING BUT NOT LIMITED TO THE SAMPLE CODE ("CODE ") SUPPLIED, IS PROVIDED STRICTLY "AS-IS" WITH NO WARRANTIES OF ANY KIND, EXPRESS, IMPLIED OR STATUTORY, MADE WITH RESPECT TO THE INFORMATION TO INCLUDE THE CODE AND ALL ACCOMPANYING WRITTEN MATERIALS, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. YOU ASSUME THE ENTIRE RISK AS TO THE QUALITY, ACCURACY, AND PERFORMANCE OF THE INFORMATION, AND YOU ASSUME ANY AND ALL RISK AND LIABILITY FOR ANY ACTIONS TAKEN BY YOU ON THE BASIS OF ITS ANALYSIS OR OTHER USE OF THE INFORMATION, INCLUDING BUT NOT LIMITED TO MODIFICATIONS TO YOUR PRODUCTS IN LIGHT OF SUCH USE OF INFORMATION, AND YOU HEREBY ACKNOWLEDGE THAT VITESSE SHALL HAVE NO RESPONSIBILITY OR LIABILITY AS A RESULT OF YOUR USE OF INFORMATION PROVIDED HEREUNDER.

IN NO EVENT SHALL VITESSE BE LIABLE FOR ANY DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR OTHER PECUNIARY LOSS) ARISING OUT OF USE OR INABILITY TO USE THE CODE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This Agreement is governed by the laws of the State of California, without regard to principles of conflicts of laws. Each provision of this Agreement is severable. If a provision is found to be unenforceable, this finding does not affect the enforceability of the remaining provisions of this Agreement. This Agreement is binding on successors and assigns. By accessing the information contained in or referenced by this document, You acknowledge that You have read this Agreement, that You understand it, that You agree to be bound by its terms, and that this is the complete and exclusive statement of the Agreement between You and Vitesse regarding the information and Code.

Contents

Figures

Tables

1 Introduction

The Vitesse unified Application Programming Interface (API) provides a comprehensive, user friendly, and robust function library that supports all Vitesse Ethernet switch, PHY, and Optical Transport Network (OTN) Mapper products. The unified API is portable to any Operating System (OS) and was developed with 32-bit and 64-bit CPUs as intended targets. The driver software was developed in standard C and supports multi-instance device targets. This document explains the code structure and porting process along with application examples to assist in rapid development and system deployment.

Additional application notes, device collateral, and API release notes are available on the Vitesse website at www.vitesse.com.

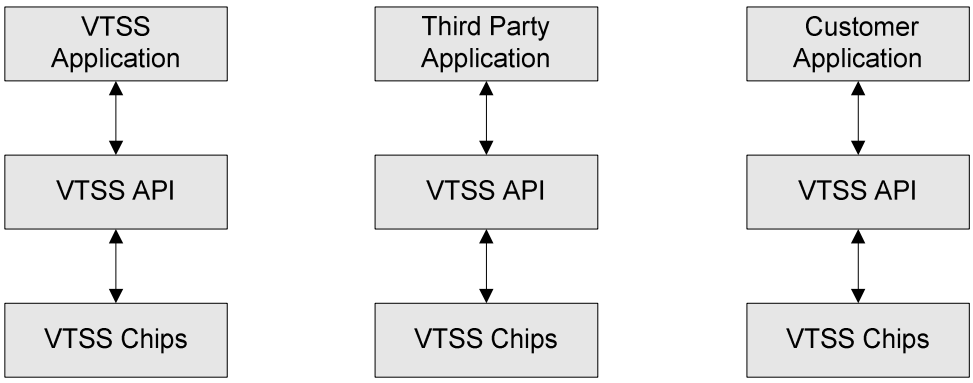
2 Overview

2.1 Applications

The Vitesse API provides portable driver software for Vitesse switch, PHY, and OTN Mapper products. It can be used as a basis for application software solutions such as the following.

- Vitesse application software used for production and demonstration
- Third party application software provided by a partner company
- Customer application software developed by customers using the Vitesse API

Figure 1. Vitesse API Applications



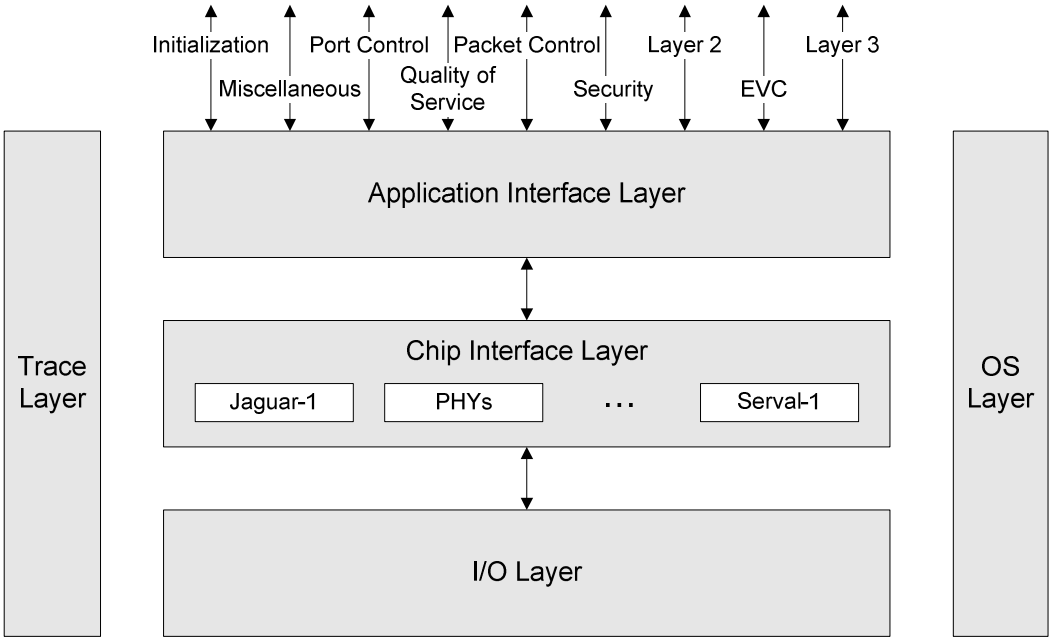
2.2 API Architecture

The API Architecture consists of the following layers.

- **Application Interface Layer** provides a C interface to the application layer. Functions are arranged in groups.

- **Chip Interface Layer** maps function parameters to chip-specific register accesses.
- **I/O Layer** provides register access. This layer is platform dependent and is implemented outside the API.
- **OS Layer** adapts the API to a given OS.
- **Trace Layer** maps code trace macros for debugging purposes.

Figure 2. Vitesse API Architecture



2.3 Directory Structure

The following table shows the subdirectories into which the API source is organized. The source code includes application example code in a separate directory.

Table 1. Directory Structure

Directory	Description
vtss_api/doc	Automatically generated documentation of the public header files. The directory includes a document for each product family.
vtss_api/include	Public (external) header files. The application must include only vtss_api.h, which includes all other required header files.
vtss_api/base	API implementation and private (internal) header files arranged in a number of sub-directories. The application build system (e.g., Makefile) should compile all C files in this directory, including sub-directories.
vtss_api/boards	Reference implementations of board specific code. Used by the Linux kernel module example.

Directory	Description
vtss_api/appl	Application example implementations, running mainly as userspace Linux applications.
vtss_api/linux_support	Contain example user-mode IO (UIO) Linux drivers for supporting running the switch API on external CPU systems via PCIe etc.

2.4 Targets

The API must be compiled for one or more target chips by defining one or more of the symbols in the following table. This selection has the following effects:

- The external header files will include the functions and data types supported by the selected targets.
- The Application Interface Layer will include the functions supported by the selected targets.
- The Chip Interface Layer will be included for the selected targets.

The VTSS_CHIP_CU_PHY target may be used when compiling the API for a PHY-only application.

When compiling the API for switch, MAC, or OTN Mapper targets, the PHY part is always included. For example, if the VTSS_CHIP_JAGUAR_1 target is selected, PHYs connected to the MII management controller of the switch chip can be controlled using the PHY portion of the API.

Table 2. API Targets

Target	Part Number	
VTSS_CHIP_CU_PHY	VSC8211	VSC8641
	VSC8221	VSC8664
	VSC8224	VSC8502
	VSC8234	VSC8504
	VSC8244	VSC8514
	VSC8512	VSC8552
	VSC8522	VSC8572
	VSC8538	VSC8574
	VSC8558	VSC8582
	VSC8601	VSC8584
	VSC8634	
VTSS_CHIP_10G_PHY	VSC8256	VSC8487-15
	VSC8257	VSC8488-15
	VSC8258	VSC8489
	VSC8484	VSC8490
	VSC8486	VSC8491
	VSC8488	
VTSS_CHIP_SPARX_III_11	VSC7414	
VTSS_CHIP_SERVAL_LITE	VSC7416	
VTSS_CHIP_SERVAL	VSC7418	

Target	Part Number
VTSS_CHIP_SPARX_III_10_UM	VSC7420
VTSS_CHIP_SPARX_III_17_UM	VSC7421
VTSS_CHIP_SPARX_III_25_UM	VSC7422
VTSS_CHIP_CARACAL_LITE	VSC7423
VTSS_CHIP_SPARX_III_10	VSC7424
VTSS_CHIP_SPARX_III_18	VSC7425
VTSS_CHIP_SPARX_III_24	VSC7426
VTSS_CHIP_SPARX_III_26	VSC7427
VTSS_CHIP_CARACAL_1	VSC7428
VTSS_CHIP_CARACAL_2	VSC7429
VTSS_CHIP_E_STAX_III_48	VSC7432
VTSS_CHIP_E_STAX_III_24_DUAL	Dual VSC7431
VTSS_CHIP_E_STAX_III_68	VSC7434
VTSS_CHIP_E_STAX_III_68_DUAL	Dual VSC7434
VTSS_CHIP_SERVAL_2	VSC7438
VTSS_CHIP_SPARX_IV_52	VSC7442
VTSS_CHIP_SPARX_IV_44	VSC7444
VTSS_CHIP_SPARX_IV_80	VSC7448
VTSS_CHIP_LYNX_1	VSC7462
VTSS_CHIP_JAGUAR_1	VSC7460
VTSS_CHIP_LYNX_2	VSC7464
VTSS_CHIP_JAGUAR_2	VSC7468
VTSS_CHIP_DAYTONA	VSC8492
VTSS_CHIP_TALLADEGA	VSC8494

2.5 Instance References

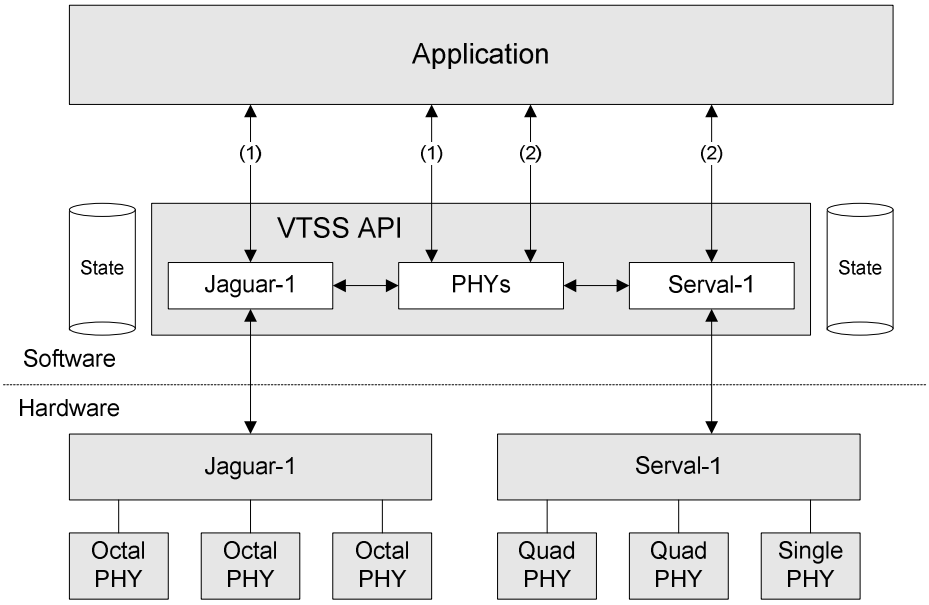
The API supports control of multiple target instances from one application. To facilitate this, the API provides a create function, **vtss_inst_create**, which must be called for each target instance during initialization. The create function returns an instance reference that must be used for subsequent API calls on the same target.

For applications controlling a single target instance, the create function may be called with a NULL instance reference pointer and subsequent API calls on the target may use a NULL instance reference.

The following illustration shows how the instance references are used to select target instances for an application controlling a Jaguar-1 switch and a Serval-1 switch, both of which control multiple PHYs. The API internally associates a state block with each created target. In the figure, interface call references are indicated using numbers.

1. The first instance created is the Jaguar-1 target. Subsequent API function calls using the returned instance reference are directed to the Jaguar-1 chip for switch and PHY control.
2. The second instance created is the Serval-1 target. Subsequent API function calls using the returned instance reference are directed to the Serval-1 chip for switch and PHY control.

Figure 3. Instance References



2.6 Initialization

The following initialization sequence must be used for each target instance controlled by the application. Note the differences between switch and OTN Mapper targets as compared to PHY-only targets. The application example includes initialization code demonstrating this sequence.

1. Create a target instance using **vtss_inst_get** and **vtss_inst_create**.
2. Initialize a target instance using **vtss_init_conf_get** and **vtss_init_conf_set**.
   a. For switch and OTN Mapper targets, register read/write callback functions must be provided by the application.
   b. For PHY-only targets, MII management read/write callback functions must be provided by the application.
3. For switch targets, set up the port map table using **vtss_port_map_set**.

When the sequence has been completed, the application can start controlling ports, Quality of Service (QoS), and other functions on the target.

2.7 API Protection

The API functions are not reentrant due to an internal state in the targets and the API. To protect API accesses for multi-threaded applications, two callback functions must be implemented by the application using a semaphore.

- **vtss_callout_lock** This function is called by the API when entering an API function.
- **vtss_callout_unlock** This function is called by the API when exiting an API function.

For single-threaded applications, these functions may be left empty.

2.8 OS Layer

The OS layer implements timer functions required by the API.

An OS-specific header file is needed for each OS type. The API supports Linux and eCos, but can be extended to support any OS with basic timer functionality.

The OS-specific header defines the following entities:

- **vtss_mtimer_t**(*typedef*) Data type used to implement timers.
- **VTSS_NSLEEP**(*nsec*) Macro to sleep for *nsec* nanoseconds.
- **VTSS_MSLEEP**(*msec*) Macro to sleep for *msec* microseconds.
- **VTSS_MTIMER_START**(*timer*, *msec*) Macro to start a timer by initializing the timeout to *msec* microseconds, where *timer* is a variable declared of the type **vtss_mtimer_t**. If the timer has already been started it must be restarted with the new timeout value.
- **VTSS_MTIMER_TIMEOUT**(*timer*) Macro used to determine if the timer, *timer,* has expired, where *timer* is a variable declared of type **vtss_mtimer_t**.

2.9 Trace Layer

The API code includes trace macros, in the style of the C function printf, for debugging and troubleshooting. The application must implement the following callback function for mapping the trace macros:

1. **vtss_callout_trace_printf** Print or log a trace message.
2. **vtss_callout_trace_hex_dump** Print or log a dump of data in hexadecimal.

The trace macros are organized in groups with levels that may be changed at run-time. By default, only the error trace is enabled.

The trace system may be disabled by setting *VTSS_OPT_TRACE* to zero at compile-time. The callback function must not be implemented if trace is disabled.

2.10 Recommended API calling sequence

The following general sequence is recommended to be used by anyone using the PHY API. This sequence will help prevent the passing of structures from the application to the PHY API with uninitialized or incorrectly initialized variables. In general, the PHY

API does not perform boundary checking of elements within a given structure passed into the API.

1. Declare the structure to be passed into the PHY API.
2. Initialize all parameters in the declared structure to 0 using **memset** or equivalent OS function.
3. If the PHY API has a "get" function, execute the "get" function call so that the structure that was previously declared and passed in gets set to the existing values in the PHY API. Example: **vtss_init_conf_get**
4. Modify only the values within the structure where a change is desired.
5. The PHY API has "set" functions, execute the "set" function call so that the structure passed in gets set in the Hardware configuration registers by the PHY API. Example: **vtss_init_conf_set**

Performing the above sequence will minimize issues that can be caused by either uninitialized elements or incorrect default settings for elements within any given data structure passed into the PHY API.

2.11 Checklist for API configuration

The following items are suggested areas that a user porting the PHY API may want to pay additional attention. Some initializations can be done on the compilation line.

1. Initialize VTSS_OPT_PORT_COUNT macro, Ex: `-DVTSS_OPT_PORT_COUNT=4`
2. Initialize Operating System type, Ex: `-DVTSS_OPSYS_LINUX=1`
3. Initialize for PHY Chip Type, Ex: `-DVTSS_CHIP_CU_PHY`
4. Initialize the VTSS_MSLEEP macro properly

2.12 Sample API Demo Applications

Demo applications for the PHY API can be found in the vtss_api/appl directory.

vtss_appl_cu_phy.c - Sample Demo program for 1G PHY

3 Product Families

This section briefly describes the API functionality for each product family.

3.1 1GE PHY Family

The 1G PHY family includes products listed for VTSS_CHIP_CU_PHY in the API Targets table.

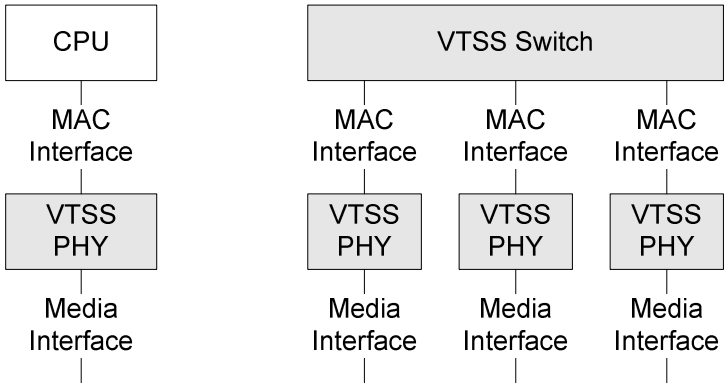
3.1.1 PHY Applications

The API supports a number of Vitesse PHY products. Each chip may include multiple PHYs (ports). For each target instance, the individual PHYs are identified using a port number. The following illustration shows two PHY applications:

1. One single PHY connected to a MAC inside a CPU. Both the MAC interface and the MII Management interface of the PHY are connected to the CPU.
2. Three octal PHYs connected to a VTSS switch chip. Both the MAC interfaces and the MII management interfaces of the PHYs are connected to the switch chip.

The API contains examples of both application types described in the Application Examples section.

Figure 4. PHY Applications



3.1.2 PHY Initialization

PHY initialization is performed in steps.

1. Initialize the chip by calling the **vtss_phy_pre_reset** function. This will perform initialization needed for the entire chip (e.g., load the internal 8051 CPU). The **vtss_phy_pre_reset** function MUST be called with the lowest PHY (port) number within the chip.
2. Initialize each PHY within the chip by calling the **vtss_phy_reset** function for each PHY (port).

3. Startup the chip by calling the **vtss_phy_post_reset** function. This will perform setup of the chip which is needed after the first time that all PHYs within the chip have been reset (e.g., set the coma pin). The **vtss_phy_post_reset** function can be called with any PHY (port) number within the chip.

3.1.3 PHY Control after Initialization

After the main initialization, the normal PHY control sequence for each port is as follows:

1. Configure the PHY using **vtss_phy_conf_set**.
2. Periodically poll the PHY status to detect link change events:
    a. For PHY-only applications, use **vtss_phy_status_get**
    b. For switch applications, use **vtss_port_status_get**
3. If link state events are detected, the application must take appropriate action. If auto-negotiation is enabled, the switch must normally be configured on link-up events, such as speed, duplex, and flow control.

3.1.4 PHY API Documentation

The 1G PHY API functions and structures are described within the chip API documentation included in the API package (e.g., /doc/vtss_serval.pdf).

3.1.5 PHY Debug Print Functions

The API contains two functions to assist the debug process.

1. Calling **vtss_phy_debug_stat_print** will print the PHY statistics.
2. Calling **vtss_phy_debug_phyinfo_print** will print the internal PHY information.

3.2 10G PHY Family

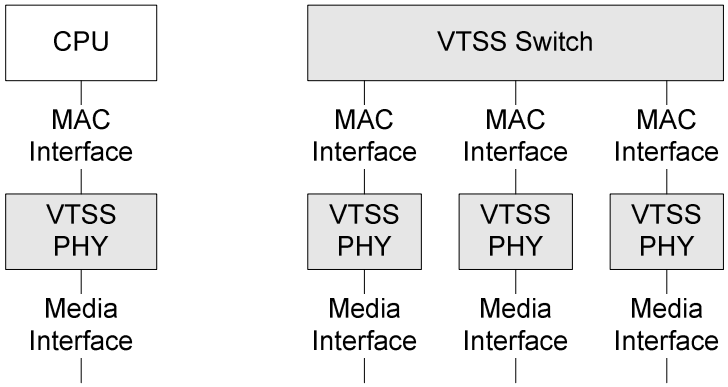
The 10G PHY family includes products listed for VTSS_CHIP_10G_PHY in the API Targets table.

The individual PHYs are identified using a port number. Figure 5 shows two PHY applications:

1. One single 10G PHY connected to a XAUI interface of a MAC inside a CPU. Both the MAC interface and the MDIO Management interface of the PHY are connected to the CPU.
2. Multiple 10G PHYs connected to a VTSS switch chip. Both the MAC interfaces and the MDIO management interfaces of the PHYs are connected to the switch chip.

The API contains examples of both application types as described in the Application Examples section.

Figure 5. 10G PHY Applications

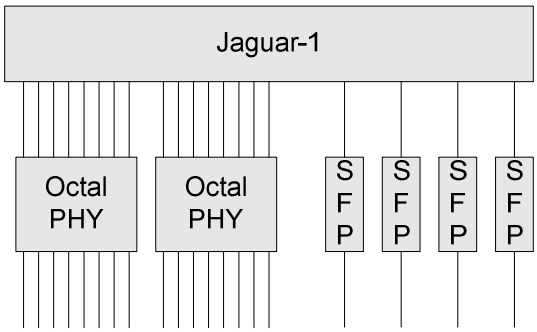


The normal 10G PHY control sequence for each port is to set the PHY operating mode using **vtss_phy_10g_phy_mode**. This function detects, resets, and sets the operating mode of the PHY type.

3.3 Switch Families

The following illustration shows an example of a switch based on the Jaguar-1 product. The system is a Layer 2 switch with 16 Cu ports and 4 SFP ports.

Figure 6. VSC7460 Jaguar-1 Application



For switch families, the initialization sequence is as follows:

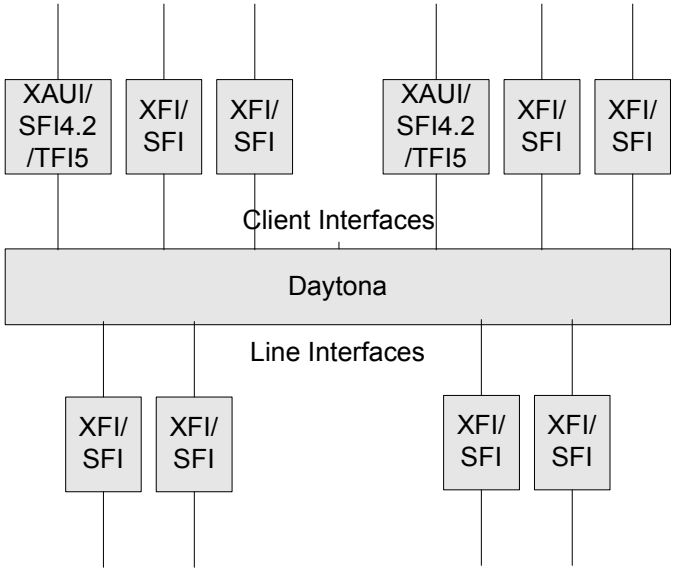
1. Create a target using **vtss_inst_get** and **vtss_inst_create**
2. Initialize a target instance using **vtss_init_conf_get** and **vtss_init_conf_set**
3. Set up the port map table using **vtss_port_map_set**
4. Configure the switch ports
    a. If PHYs are present, reset and configure PHYs
    b. Configure ports using **vtss_port_conf_get** and **vtss_port_conf_set**

3.4 OTN Mapper Families

The following illustration shows a mapper application example based on the VSC8492 Daytona product. The line interfaces operate in XFI/SFI mode. The client interfaces

may operate in XAUI/SFI-4.2/TFI-5 host interface mode or XFI/SFI mode, depending on the configuration mode of the device.

Figure 7. VSC8492 Daytona Application



For Mapper families, the initialization sequence is as follows:

1. Create a target using **vtss_inst_get** and **vtss_inst_create**
2. Initialize a target instance using **vtss_init_conf_get** and **vtss_init_conf_set**
    a. Set the mode of operation for each of the two channels.

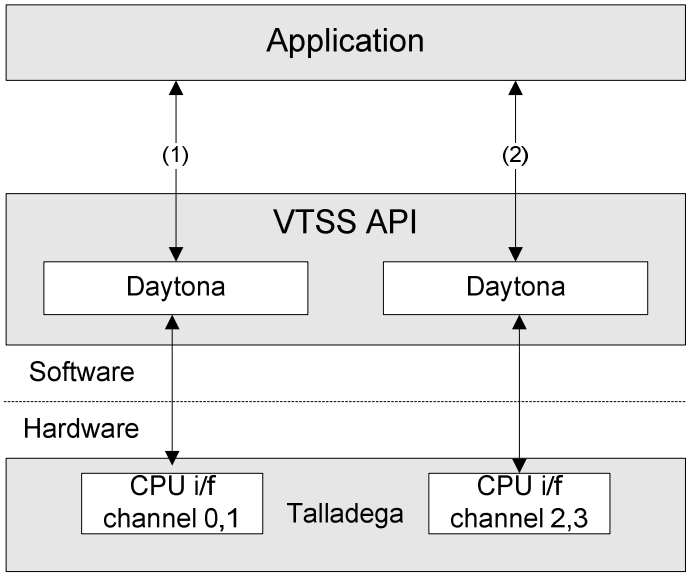
    Note: The Daytona mapper family can operate in one of many different modes. Changing this mode will cause a re-initialization of the channel.
3. Setup the port map table using **vtss_port_map_set**

For Mapper families, a warm-start initialization sequence is as follows:

1. Implemented in a later release

The VSC8494 Talladega product is actually two Daytona dies in one chip. Therefore this chip is treated as two separate chips, and as a consequence is also implemented using two instances of the API as shown in the following figure.

Figure 8. Talladega API



4 Function Groups

The following sections briefly list the most important API functions. The header files include detailed descriptions on the data types and functions while the documentation directory includes a document for each product family. These resources can be used to study the API for a specific product.

Some general definitions are listed in vtss_api/include/vtss/api/options.h:

- **Feature defines** The selection of one or more targets at compile time causes a number of symbols (VTSS_FEATURE_*) to be defined. These are used to indicate which functions and data fields are available.
- **Options** Default values are assigned to compile time options (VTSS_OPT_*). These default values may be overridden when building the API.

4.1 Initialization

Initialization functions are defined in include/vtss_init_api.h:

- Target creation
- Target initialization

These functions must be called before calling other functions for the target.

For targets supporting warm start, restart functions are also defined in include/vtss_init_api.h.

4.2 Miscellaneous

Miscellaneous functions are defined in include/vtss_misc_api.h:

- Trace configuration and callback functions
- Debug print functions (for showing API internal information)
- API lock/unlock callback functions
- Register read/write (for debugging switch and OTN Mapper targets)
- Chip ID access (for switch and OTN Mapper targets)
- GPIO control (for switch and OTN Mapper targets)
- Interrupt control (switch and OTN Mapper targets)

4.3 Port Control

The file include/vtss_port_api.h defines port control functions for switch, MAC, and OTN Mapper targets. Defined functions include the following:

- Port map (must be called after initializing switch and OTN Mapper targets)
- MMD management for 10G PHYs
- Auto negotiation, IEEE 802.3 clause 37

- Port configuration (speed, duplex, flow control, etc.) (for switch targets)
- Port status (for switch targets)
- Port statistics (for switch targets)

er targets, the configuration, status, and statics are organized per layer because the relevant layer depends on the mode of operation.

The file include/vtss_phy_api.h defines PHY control functions that include the following:

- PHY reset
- PHY configuration and status
- PHY power configuration and status
- Recovered clock configuration
- PHY register read/write functions
- VeriPHY (cable diagnostics)

The file include/vtss_phy_10g_api.h defines 10G PHY control functions that include the following:

- 10G PHY operating mode and status
- 10G PHY sublayer status
- 10G PHY reset
- 10G PHY power on/off
- 10G PHY loopback
- 10G PHY counters

4.4 Quality of Service

The file include/vtss_qos_api.h defines QoS functions for switch targets. Defined functions control the following:

- QCL: QoS classification rules
- Bandwidth control (policing and shaping)
- Egress scheduler

4.5 Packet Control

The file include/vtss_packet_api.h defines packet control functions for switch targets. Defined functions control the following:

- CPU Rx packet registration and CPU queue mappings
- CPU Rx functions
- CPU Tx functions

Frame DMA functions are defined in the file include/vtss_fdma_api.h. These can only be used from the internal CPU of the switch.

- FDMA initialization and channel configuration.
- CPU Rx functions (frame extraction)
- CPU Tx functions (frame injection)
- FDMA statistics

4.6 Security

The file include/vtss_security_api.h defines security functions for switch targets. Defined functions control the following:

- 802.1X state (switch targets)
- Access Control List (switch targets)

4.7 Layer 2

Layer 2 functions are defined in vtss_l2_api.h, which is used with switch targets.

- MAC address table (switch targets)
- Learning mode (switch targets)
- RSTP and MSTP state (switch targets)
- Virtual LAN (VLAN) membership and port configuration (switch targets)
- VCL: Advanced VLAN classification (switch targets)
- VLAN translation (switch targets)
- Port isolation (switch targets)
- Private VLANs (switch targets)
- Link aggregation (switch targets)
- Port Mirroring (switch targets)
- IPv4 multicast control (switch targets)
- IPv6 multicast control (switch targets)
- Port protection switching (switch targets)
- Ring protection switching (switch targets)
- Port forwarding state (switch targets)
- VStaX stacking configuration (switch targets)

4.8 EVC

Ethernet Virtual Connection functions are defined in the file vtss_evc_api.h.

4.9 Synchronization

Synchronization functions are defined in the file vtss_sync_api.h.

4.10 Time Stamping

The file vtss_ts_api.h defines time stamping functions at the MAC layer for the Caracal and Jaguar-1 switch families.

The file vtss_phy_ts_api.h defines time stamping functions at the PHY layer for devices that support IEEE1588v2 such as the VSC8487-15, VSC8488-15, VSC8574, and VSC8492. Functions control the following:

- 1-step and 2-step timestamping feature.
- Tx timestamp FIFO interface.
- Set/Get/Synchronize Time of Day.
- Adjust clock rate.

4.11 OTN Mapper Layers (OTN Mappers Only)

The API defines functions to control OTN Mappers such as the following:

- Interface control and monitoring (XFI and UPI layers)
- MAC control and monitoring(MAC and PCS layers)
- OTU/FEC control and monitoring
- WIS/SONET/SDH layer control and monitoring
- GFP mapping control and monitoring
- Backplane auto negotiation (AN), IEEE 802.3 clause 73
- Backplane adaptive equalization (AE), IEEE 802.3 clause 72

5 Application Examples

5.1 Platforms

The vtss_api/appl directory includes application examples demonstrating how to use the API. The applications can be built to run the Vitesse platforms shown in the following table.

The application can be built using the Linux "cmake" tool, or by compiling the files below with the listed defines.

Refer also to chapter 6, if you are using the API running Linux on the internal CPU of a switch chipset.

Table 3. Application Example Platforms

Platform	Defines	Application C Files
Switch application for SparX-III-26 evaluation board. Vitesse Genie board is running Linux.	VTSS_CHIP_SPARX_III_26 VTSS_OPSYS_LINUX BOARD_LUTON26_EVAL	vtss_appl.c vtss_appl_cli.c vtss_appl_board_l26_eval.c
Switch application for Jaguar-1 evaluation board Vitesse Genie board is running Linux	VTSS_CHIP_JAGUAR_1 VTSS_OPSYS_LINUX BOARD_JAGUAR1_EVAL	vtss_appl.c vtss_appl_cli.c vtss_appl_board_jr1_eval.c
Switch application for Serval reference board using VRAP. Vitesse Genie board is running Linux.	VTSS_CHIP_SERVAL VTSS_OPSYS_LINUX BOARD_SERVAL_REF	vtss_appl.c vtss_appl_cli.c vtss_appl_board_serval_ref.c
Cu PHY application for VSC8512 evaluation board. The application can run on a Linux PC.	VTSS_CHIP_CU_PHY VTSS_OPSYS_LINUX VTSS_OPT_PORT_COUNT=12	vtss_appl_cu_phy.c vtss_appl_board_atom12_eval.c
10G PHY application example currently not targeting a specific board.	VTSS_CHIP_10G_PHY VTSS_CHIP_CU_PHY VTSS_OPSYS_LINUX VTSS_OPT_PORT_COUNT=12	vtss_appl_cu_phy.c vtss_appl_board_atom12_eval.c
10G OTN Mapper application example for Daytona. Vitesse Genie board is running Linux	VTSS_CHIP_DAYTONA VTSS_OPSYS_LINUX BOARD_DAYTONA	vtss_appl.c vtss_appl_cli.c vtss_appl_board_daytona.c
10G OTN Mapper application example for Talladega. Vitesse Genie board is running Linux	VTSS_CHIP_TALLADEGA VTSS_OPSYS_LINUX BOARD_TALLADEGA	vtss_appl.c vtss_appl_cli.c vtss_appl_board_daytona.c

5.2 Switch Application Functionality

The switch portion of the application file, appl/vtss_appl.c, demonstrates basic functionality such as the following:

- Trace system integration
- Initialization
- Port map setup
- Port reset and configuration
- Port status polling and configuration based on auto-negotiation

The board specific part of the application appl/vtss_appl_board_*.c includes the following:

- Register access
- Port map

The application includes a Command Line Interface (CLI) found in appl/vtss_appl_cli.c for configuration and monitoring of the system.

5.3 PHY Application Functionality

The PHY application file appl/vtss_appl_cu_phy.c initializes the PHYs on the target board and demonstrates their use.

Use the following steps to set up the ATOM12 PHY device from a PC (either Linux or Windows using CyWin) for use on the Vitesse Atom12 evaluation board.

1) Compile the API with the ATOM12 evaluation hardware platform and Linux OS files.

   Example:

```
gcc -g -o vtss_api.exe  -I include \
appl/vtss_appl_board_atom12_eval.c \
appl/vtss_appl_cu_phy.c \
`find base/phy/ -name "*.c"`
-DVTSS_OPSYS_LINUX=1 \
-DVTSS_OPT_PORT_COUNT=12 \
-DVTSS_CHIP_CU_PHY
```

2) The Atom12 evaluation board uses a Vitesse add-on "Rabbit CPU board" which is used to communicate with the PC and the PHY. Communication between the Rabbit board and the PC is done using a socket connection, so the user must specify the Rabbit board's IP address. The IP address is set as an argument when running the API program.

   Example:

```
vtss_api.exe 10.10.132.59.
```

5.4 OTN Mapper Application Functionality

The mapper application portion of the file, appl/vtss_appl.c, demonstrates basic functions such as the following:

- Trace system integration
- Initialization
- Port map setup
- Port reset and configuration

The board specific portion of the file appl/vtss_appl_board_*.c includes the following:

- Register access
- Access to external components on the evaluation board through the MDIO interface (i.e., VSC8486 PHY), and I2C (SFP+).
- Access to other external components on the evaluation board through the Rabbit module.

The backplane AE (IEEE 802.3 clause 72) file appl/ae/*.c includes the following:

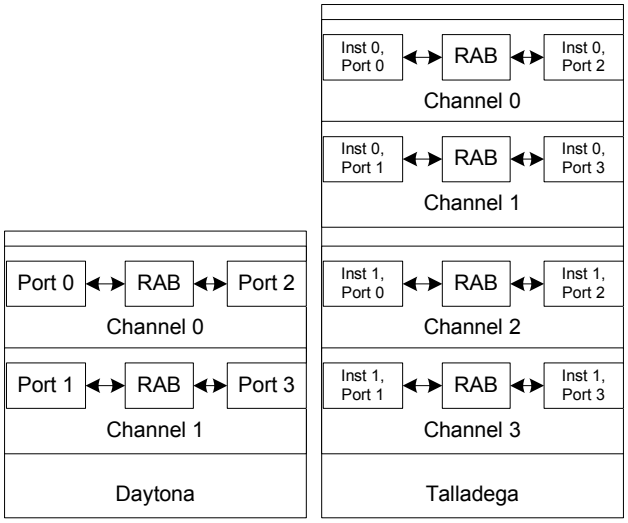
- Process for controlling the AE, using the API interface primitives for AE.

The application includes a CLI from the file appl/vtss_appl_cli.c for configuration and monitoring of the system. CLI commands are implemented for all the API functions so all API functions can be exercised from external scripts.

5.4.1 Port and Channel Numbering

The Daytona chip has two channels that connect a client side port with a line side port. Each channel is nearly symmetric as the same features are available towards both client and line side, centered about the Rate Adaptation buffer (RAB). Therefore, the API treats the Daytona as 4 ports numbered 0 through 3 as shown in the following illustration.

Figure 9. Daytona and Talladega Channels and Ports



The API calls are converted to one or more register reads or writes implemented as callouts from the API.

Example read callout function declaration:

```
/**
 * \brief Read value from target register.
 *
 * \param chip_no [IN]  Chip number (if multi-chip instance).
 * \param addr [IN]     Address to read. Byte offset from chip address base.
 * \param value [OUT]   Register value.
 *
 * \return Return code.
 **/
static vtss_rc reg_read(const vtss_chip_no_t chip_no,
                        const u32 addr,
                        u32 *const value);
```

Where:

`chip_no` is not used (always 0)

`addr` is the register offset within the chip

The Talladega chip is treated as two Daytona chips, that is two API instances are created and separate callouts are defined per instance. It has one callout for the channel 0 and 1 portion of the chip and another callout for the channel 2 and 3 portion of the chip.

The port_number from the API call is reflected in the *addr* parameter. Stated another way, the API converts the port_number and function block to the proper register address.

5.4.2 Register Access

The Mapper test application is made for both the Daytona evaluation board and the Talladega evaluation board. The Talladega application creates two instances of the API, one for channel 0 and 1 (die 0) and one for channel 2 and 3 (die 1). In the

reg_read and **reg_write** callout functions, the functions handling die 0 uses the *addr* parameter directly. The callout functions handling die 1 sets bit 22 in the *addr* parameter and uses the same register access function.

Example read callout function implementation for die 1:

```
static vtss_rc reg_read_die_1(const vtss_chip_no_t chip_no,
                              const u32 addr,
                              u32 *const value)
{
    /* chip die 1 is accessed by setting bit 22 in the address */
    addr |= 0x400000;
    reg_read(chip_no, addr, value);
    return rc;
}
```

6 Linux Support

6.1 External CPU configuration

The directory, `linux_support/misc`, contains example UIO drivers showcasing providing register and interrupt access for running the API and the example application described in chapter 5.

The drivers can be used to run the API on a Linux system with PCIe connections, for example from a x86 Ubuntu 12.04 LTS system.

Other systems configurations and can be constructed in a similar fashion, replacing PCIe with other bus options available on your target platforms.

6.2 Internal CPU configuration

If you are using the internal processor to run Linux, you may wish to use the Yocto-based BSP that Vitesse provides. Refer to Application note AN1125, available from your local Vitesse representative, for further description of the BSP.

Once you have the basic BSP compiling, you should add the `meta-vtss-switch` layer, available from https://github.com/vtss/meta-vtss-switch. You will also need to obtain a suitable API release tarball, which should be placed in recipes-applications/vtss-api/files/. You will need release 4.60a or later.

After downloading the `meta-vtss-switch` layer and placing the tar archive, you must add the layer to the `<build-directory>/conf/bblayers.conf` as below:

```
BBLAYERS ?= " \
  /home/<user>/project/poky/meta \
  /home/<user>/project/poky/meta-yocto \
  /home/<user>/project/poky/meta-yocto-bsp \
  /home/<user>/project/poky/meta-vtss-vcoreiii \
  /home/<user>/project/poky/meta-vtss-switch \
  "
```

With the new layer added, you can now build (bitbake) the "`vtss-core-minimal`" target, and produce a flash image containing the Vitesse Unified Switch API shared library as well as the "`vtss_miniapp`" example application.

Refer to AN1125 and https://www.yoctoproject.org/ for more information for more information on Yocto and how to deploy the image on a target system.

7 Porting Guide

This section summarizes the required steps to port the API to a given platform.

7.1 Board Support Package

For a given CPU system and OS, a board support package (BSP) must be developed. Vitesse currently provides the following BSPs:

- VCore-III/eCos for SparX-III/Caracal/Jaguar reference systems
- VCore-III/Linux for SparX-III/Caracal/Jaguar reference systems

7.2 Build System

Create a build system to compile the OS, application, and Vitesse API. All C-files in the vtss_api/base directory, including sub directories, must be compiled. Also set the appropriate OS (`VTSS_OPSYS_ECOS`, `VTSS_OPSYS_LINUX` or `VTSS_OS_CUSTOM`) and target (`VTSS_CHIP_*`) defines for the platform.

The file vtss_api/CMakeLists.txt is available to support the cmake tool.

7.3 OS Layer

When compiling for other than supported OS types, the Makefile must define the `VTSS_OS_CUSTOM` C preprocessor symbol.

By doing so, the include file <vtss_os_custom.h> will be included to implement the OS timer functions as described in Section 2.8. This file is not present in the distributed source, but is reserved for OS porting purpose.

7.4 Register Access

For switch and OTN Mapper targets, register read/write functions must be implemented.

For PHY targets the MII management read/write functions must be implemented.

7.5 API Protection

The **vtss_callout_lock** and **vtss_callout_unlock** functions must be implemented. For single-threaded applications, these functions may be left empty.

7.6 Trace Layer

The **vtss_callout_trace_printf** and **vtss_callout_trace_hex_dump** functions must be implemented if the application wants to use the API trace. Alternatively, set `VTSS_OPT_TRACE=0` in your build system to exclude the trace.

As the API is split into different groupings, it is possible to enable the trace feature individually for each group. This is done with the **vtss_trace_conf_set** function.

Refer to the specific chip API documentation included in the API package (e.g., doc/vtss_jaguar1.pdf).

7.7 Application

The application code must be developed, possibly using the Vitesse application example code for reference. Include the file vtss_api/include/vtss_api.h to access the Vitesse API. For switch targets, ensure that the port map is setup correctly.