Texas Instruments

VLYNQ Device Driver

VLYNQ Architecture/Design

December 2007 ***Platform Software Group***

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		Applications	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address:
Texas Instruments
Post Office Box 655303, Dallas, Texas 75265

About This Document

This document is VLYNQ PAL API(s) specification. This is also intended to serve as a programming and users guide for PSP development team and BU(s) alike.

This document assume, knowledge of Base PSP and its internals and is intended to present technical thought process that has gone into the design.

This specification replaces all the previous Base PSP implementation for VLYNQ.

Trademarks

Related Documents

- Platform Framework: http://www.software.ti.com/cdronline/focuszones/psp/methodology.htm
- VLYNQ Refer to www.nbu.sc.ti.com for latest version.

Notations

Explain any special notations or typefaces used (such as for API guides, special typefaces for functions, variables, etc.)

Terms and Abbreviations

Term	Description
API	Application Programmer's Interface
DDC	Device driver core.
OS	Operating System
SOC	System On Chip
PSP	Platform Support Package

Revision History

Date	Author	Comments	Version
May 26, 2006	Saloni Shah	Document created	1.0
August 7, 2006	Saloni Shah	Changes for the Release 0.1.1	1.1
October 8, 2006	Rinkal Shah	Review comments closed	1.2
December 1. 2006	Ankur Verma	Modified for the release 0.3.0	1.3
March 24, 2007	Anuj Aggarwal	Modifying according to the VLYNQ latest code base	1.4
June 5, 2007	Anuj Aggarwal	Modifying according to the VLYNQ latest code base taken from DM64LC release 1.0.0.1	1.5
November 15, 2007	Nagarjuna K	Updated for DM6437/C6424/DM648/C6452 package	1.6

Table of Contents

1 System Context 1

1.1 Hardware 1

1.2 Software 1

1.2.1 Operating Environment and dependencies 1

2 Design Considerations 1

2.1 Design Constraints 2

3 Design Goals 2

3.1 Nomenclature 3

4 Design Goals 4

4.1 Initialization 4

4.2 Remote Devices 4

4.3 Interrupts 4

5 Application Programming Interface(s) 7

5.1 Configuration API(s) 7

5.1.1 PAL_sysVlynqInit( ) - Initialize the VLYNQ control module. 7

5.1.2 PAL_sysVlynqInitSoc( ) - Initialize the VLYNQ control module. 8

5.1.3 PAL_sysVlynqCleanUp( ) – Un-Initialize the VLYNQ control module. 9

5.1.4 PAL_sysVlynqDevCreate( ) – Creates a device reference. 10

5.1.5 PAL_sysVlynqDevDestroy( ) – Destroys the device reference. 11

5.1.6 PAL_sysVlynqMapRegion( ) – Map the memory regions of the device. 12

5.1.7 PAL_sysVlynqMappedRegion ( ) – Return the Mapped Region configuration for Local/Peer. 13

5.1.8 PAL_sysVlynqUnMapRegion( ) – UnMap the memory regions of the device. 14

5.1.9 PAL_sysVlynqMapIrq( ) – Maps the IRQ hardware line onto the VLYNQ. 15

5.1.10 PAL_sysVlynqUnMapIrq( ) – UnMaps the IRQ hardware line. 16

5.1.11 PAL_sysVlynqChainAppend( ) –Append to the VLYNQ chain. 17

5.1.12 PAL_sysVlynqAddDevice( ) –Add the device reference into VLYNQ. 18

5.1.13 PAL_sysVlynqRemoveDevice( ) – Removes the device reference from VLYNQ. 19

5.1.14 PAL_sysVlynqChainUnAppend( ) – Remove (the tail) from the VLYNQ chain. 20

5.1.15 PAL_sysVlynqRootIsr( ) – The Root ISR; register it with the system. 21

5.2 For Drivers et al. 22
5.2.1 PAL_sysVlynqDevFind( ) – Get the handle for the device. 22
5.2.2 PAL_sysVlynqDevGetVlynq( ) – Get the VLYNQ for this device. 23
5.2.3 PAL_sysVlynqGetDevBase( ) – Get the physical base address of the device. 24
5.2.4 PAL_sysVlynqDevFindIrq( ) – Get the mapped interrupts of the device. 25
5.2.5 PAL_sysVlynqDevGetResetBit( ) – Get the reset bit of the device. 26
5.2.6 PAL_sysVlynqAddIsr( ) – Install the ISR for the device. 27
5.2.7 PAL_sysVlynqRemoveIsr( ) – Uinstall the previously installed ISR. 28
5.2.8 PAL_sysVlynqDevCbRegister( ) – Register for the callbacks. 29
5.2.9 PAL_sysVlynqDevCbUnregister( ) – Unregister the callbacks. 30
5.3 Control API(s) 31
5.3.1 PAL_sysVlynqIoctl( ) – Read/Write register of the VLYNQ module. 31
5.3.2 PAL_sysVlynqClockConfig( ) – Configures the Clock for the VLYNQ bridge. 32
5.4 Interrupts Galore 33
5.4.1 PAL_sysVlynqGetForIrq( ) – Get the VLYNQ for the IRQ. 33
5.4.2 PAL_sysVlynqSetIrqPol( ) – Set the polarity of the hardware IRQ line. 34
5.4.3 PAL_sysVlynqSetIrqType( )- Set the type of the hardware IRQ line. 35
5.4.4 PAL_sysVlynqGetIrqPol ( ) – Get the polarity of the hardware IRQ line. 36
5.4.5 PAL_sysVlynqGetIrqType( ) – Get the type of the hardware IRQ type. 37
5.4.6 PAL_sysVlynqGetIrqCount( ) – Get the number of times this IRQ occurred. 38
5.4.7 PAL_sysVlynqDisableIrq( ) – Disable the IRQ. 39
5.4.8 PAL_sysVlynqEnableIrq( ) – Enable the IRQ. 40
5.5 Status and Utility 41
5.5.1 PAL_sysVlynqGetLinkStatus( ) – Get the status of the of the VLYNQ module. 41
5.5.2 PAL_sysVlynqGetNumRoot( ) – Get the number of the root VLYNQ(s). 42
5.5.3 PAL_sysVlynqGetRoot( ) – Get the handle to the specified root VLYNQ. 43
5.5.4 PAL_sysVlynqGetRootVLYNQ( ) – Get root for the given VLYNQ. 44
5.5.5 PAL_sysVlynqGetRootAtBase( ) – Get the root VLYNQ at the base address. 45
5.5.6 PAL_sysVlynqGetBaseAddr( ) – Returns the base address of the VLYNQ. 46
5.5.7 PAL_sysVlynqGetNext( ) – Get the next VLYNQ module in the chain. 47
5.5.8 PAL_sysVlynqIsLast( ) – Is this VLYNQ module the last one in the chain. 48
5.5.9 PAL_sysVlynqGetChainLength( ) – Get the length of the VLYNQ chain. 49
5.5.10 PAL_sysVlynqDump( ) – Dumps vital VLYNQ information into the buffer. 50
6 Appendix A 51
6.1 Enumerators and typedefs 51
6.2 PAL_VLYNQ_CONFIG_T (pal_sysvlynq.h) 52
6.3 User defines 54
7 Appendix B 55

List of Figures

Figure 1 VLYNQ Block 1

List of Tables

Error! No table of figures entries found.

1 System Context

1.1 Hardware

The VLYNQ module core used has the following blocks:

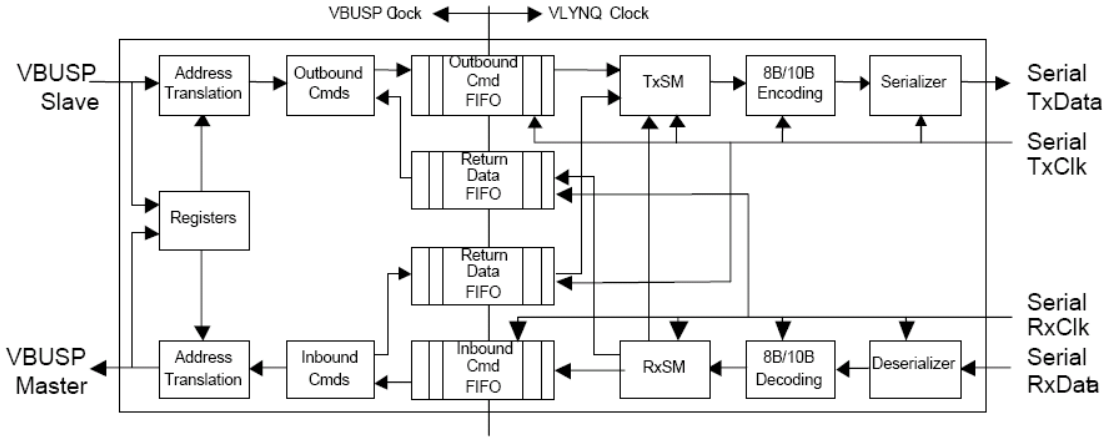


Figure 1 VLYNQ Block

1.2 Software

The document provides an overall understanding of the TI VLYNQ device driver architecture.

1.2.1 Operating Environment and dependencies

Refer system level release notes for tools and BIOS versions.

2 Design Considerations

VLYNQ is serial (i.e. low pin count) communications interface that enables the extension of an internal CBA bus segment to one or more external physical devices. VLYNQ accomplishes this function by serializing bus transactions in one device, transferring the serialized transaction between devices via a VLYNQ port, and de-serializing the transaction in the external device. VLYNQ hardware module has been used in DaVinci, Jacinto, Avalanche, Puma, Sangam, Titan, APEX and other TI communication processors.

Since the same hardware module is used in various SOC(s), there is a need for common VLYNQ software that can be used in the PSP software for these SOC(s). The design goal for this software implementation is to produce configurable VLYNQ software that can be ported / used in the PSP software for any SOC (Jacinto, Avalanche, Puma, Sangam, APEX and Titan as of now).

Configuration mechanisms, storage and mitigation are outside the scope of this design endeavor.

2.1 Design Constraints

The design constraints for the implementation are documented here upfront to set the expectations of the design:

Dynamic run time discovery / enumeration of VLYNQ devices - VLYNQ does not perform an enumeration operation on the VLYNQ bus. It just configures the host configuration (and its associated peer VLYNQ) based upon configured parameters. The application can enumerate the bus, discover devices and call the VLYNQ init function with the required parameters.

Since the VLYNQ module offers flexibility in terms of interrupt configuration and other aspects, the implementation interface API has to be as flexible as possible. This is to allow the application to dictate the working of the module. Thus wrapper code that uses this implementation is very critical. This implementation cannot be “on its own”. It needs to be used as per the design of the SOC and its applications.

VLYNQ hardware does not allow generating local interrupts (intLocal=1) and also sending them to the remote via interrupt packets. Thus, either the host or remote handles VLYNQ interrupts. For this version of implementation, the host handles the interrupts.

The configuration storage and persistence aspects are outside the scope of this discussion.

Note: Even though there are few VLYNQ registers in hardware, VLYNQ module configuration is a bit complex considering all the flexibility offered by the hardware. The implementation does not provide a readymade solution for all the scenarios in which this hardware can be used. Instead, it assists the developer to program the hardware with the implementation API.

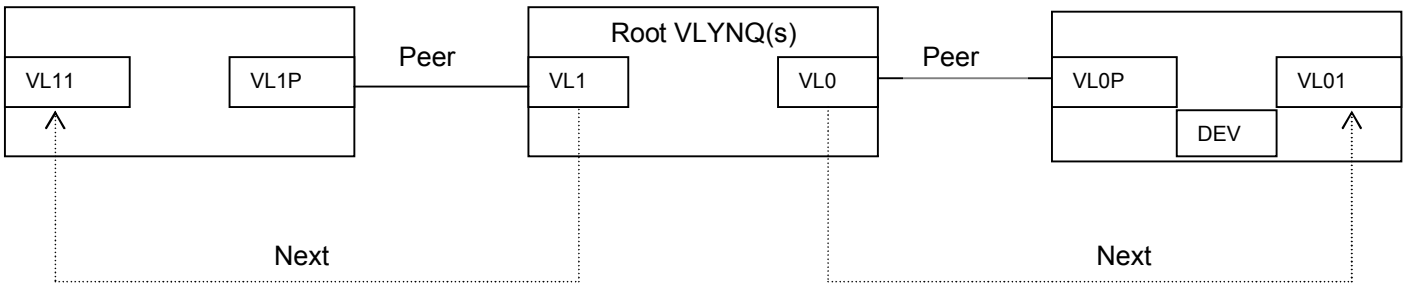
It shall be possible to chain multiple downstream entities.

3 Design Goals

- Keep it simple; do not add new concepts/aspects in VLYNQ software.
- Implementation to re-usable code for DSP/BIOS as an immediate requirement.
- Make the implementation independent of SOC.
- Flexible configuration for the VLYNQ interrupts mechanism.

- Supports up to 2.6 VLYNQ starting from 1.1 VLYNQ on a same code base.

3.1 Nomenclature



Root: The VLYNQ control module on the avalanche family SoC is called Root. Typically, there are two roots. For this version of BasePSP 7.0, there is one root for VLYNQ. By convention, the first VLYNQ control module to be initialized on the avalanche SoC is assigned root 0.

Bridge: It is the Link between two VLYNQ control modules over the VLYNQ serial bus. The usual convention of identifying a bridge is <root#><hop#>. The hop# as well as root# are 0 based. The first hop on root 0 shall be then identified as bridge00; the second hop shall be bridge01 and so on. A notation such as bridge<root#> in general and specifically bridge0 shall mean everything pertaining to multiple hops over the cascaded serial VLYNQ(s) associated with root 0.

Chain: A serial cascade of VLYNQ bridges. It has one root and one or more bridges.

Device (Dev): An instance of peripheral on the system.

4 Design Goals

4.1 Initialization

A SOC shall have typically one or two VLYNQ interfaces or control modules. They are called root VLYNQ modules or simply root(s). Each (root) VLYNQ module shall have a peer VLYNQ module. Typically, each VLYNQ module shall be initialized along with its peer.

The software initialization procedure involves specifying the virtual base address of the VLYNQ module and encompasses setting up of clock, interrupt and Endianess information as applicable for the VLYNQ module. The initialization kick starts the activities in the hardware and returning a software handle on successful completion.

The initialization sequence for a chained VLYNQ setup goes in out; meaning first root VLYNQ module or interface is initialized; it is ascertained for the hardware state and then next VLYNQ module is taken up for initialization. Of course, it is responsibility of the user to ensure that next VLYNQ module is mapped into the memory map of the root VLYNQ module prior to initializing the next VLYNQ module.

4.2 Remote Devices

Once the VLYNQ hardware with its peer has been initialized, it shall be possible to add information about the device(s) connected to peer VLYNQ, such devices can be TI DSP (DM64vx), Serial ATA, Customer FPGA, vdsp et al, which are being accessed through serial bus into the VLYNQ modules. Even the next VLYNQ module should be specified this way.

It is left to the discretion of the user and need of the system to specify a single block (RX Size and RX Offset) for a set of multiple devices associated with the peer VLYNQ or individual block for each device. User shall be required to specify the VLYNQ interrupt numbers (to map the hardware interrupt lines) that should to be used for a given device. Users shall ensure that the mapping of interrupt vectors to the interrupt numbers are unique with a VLYNQ module and with in a chain, if applicable.

For a chained VLYNQ setup, user should ensure that the next VLYNQ module has been mapped in the context of the current VLYNQ module. User shall specify the mapped virtual address of the next VLYNQ at creation time.

4.3 Interrupts

Interrupts under VLYNQ assume significance given the fact that there are many possibilities to configure the interrupts over VLYNQ.

Since the VLYNQ hardware module permits either local interrupt generation or interrupt packets to peer (remote) (intLocal bit in the configuration register), it is not possible in hardware to map the interrupt two ways i.e. generate a local interrupt to the host and also propagate the same to the remote device. **This is a hardware limitation.**

Thus, special care needs to be taken by the user when programming intLocal and int2Cfg bits of the VLYNQ device; user discretion is advised.

IntLocal bit in the control register determines if pending interrupts (from the Int Pending/Set register) are propagated to the peer over the link. Thus to propagate local interrupts to the peer, intLocal should be set to 0. Effectively local VLYNQ module status interrupts are not reported (as interrupts) to the local host. This is a hardware limitation (feature) and not software limitation. **Typically the host handles interrupts from the remote device. In this scenario, the host configures local intLocal=1 and remote intLocal=0.** In this case, the host local VLYNQ interrupt vectors should not be enabled.

For example: Jacinto SOC has HECC, MLB/IIS, MiBSPI, McASP, GPIO interrupts as interrupt vectors on the VLYNQ module. If the local intLocal=1, then these vectors should not be enabled, otherwise these will result in unwanted interrupts in the system. Only when local intLocal=0, these interrupt vectors should be enabled so that these interrupts are propagated to the peer.

Interrupt packets from the peer can be handled in two ways. If Int2CFG = 1, then Int Ptr register should point to Int Pending/Set register and then peer interrupts bits are written to the local Int Pending/Set register thereby causing local interrupt to the host (only if IntLocal = 1). Else Int Ptr register can point to the host device Interrupt Set register in which case, the remote interrupts will directly be handled by the host device interrupt controller. ***For the sake of simplicity at this time, the current implementation mandates the user to configure the int2CFG for the root (local) VLYNQ as 1. The root VLYNQ shall handle all the interrupts at this time.***

Depending upon the type of remote VLYNQ interface, a mechanism is usually provided to interrupt the remote side (e.g. in V2PCI module, there is a register to generate interrupts to the PCI based upon setting of some bits). This mechanism is generally used to generate an interrupt on the remote VLYNQ by the host software. If such a mechanism is not present and peer interrupts are required, and this has to be achieved by via the VLYNQ, then set the local intLocal=0 and remote intLocal=1 on the host VLYNQ interface. ***For the sake of simplicity, the implementation assumes that there is no impending and immediate requirement to post interrupts from the host to its peer; this limits the user to configure local intLocal=1 and peer intLocal = 0.***

If two way interrupts are desired, then this can be achieved by setting intLocal=0 of both SOC local VLYNQ and remote VLYNQ device. In this mechanism the int2Cfg should be set to 0 and Int Ptr register pointing to the device interrupt set register. The only problem with this mechanism is that local VLYNQ error/status interrupt has to be handled by the peer and vice versa. ***This is an advanced configuration concept and requires careful configuration efforts and hence for the sake of simplicity this configuration aspect is not allowed in the implementation.***

A local interrupt handler is installed to handle local VLYNQ status interrupts. A similar interrupt handler is required to handle remote VLYNQ status interrupts if remote is passing its interrupts to local VLYNQ device. Since VLYNQ interrupt has 32 bits, bit vector is passed in the initialization function. The handlers are automatically installed based upon local and remote VLYNQ interrupt configuration.

Further to this discussion, it is the onus on the user to ensure that, unique interrupt vector to number mapping happens. It is found that typically PCI devices could use sharing, so interrupt sharing is supported by this implementation; this means that while it is possible to install multiple interrupt service routines for a given interrupt number but still vector to interrupt number mapping for interrupt should be unique.

5 Application Programming Interface(s)

For information on data structures please refer to Appendix A.

5.1 Configuration API(s)

5.1.1 PAL_sysVlynqInit( ) - Initialize the VLYNQ control module.

PAL_Result	PAL_sysVlynqInit	(	void	)

5.1.2 PAL_sysVlynqInitSoc( ) - Initialize the VLYNQ control module.

PAL_VLYNQ_HND* PAL_sysVlynqInitSoc (PAL_VLYNQ_CONFIG_T* pal_vlynq_config)		
	config	The collection of configuration parameters; to initialize the VLYNQ interface. Refer to Appendix A for details. Note: **If the VLYNQ control module being initialized happens to be a non-root entity, then, it is necessary and MUST that this VLYNQ (identified by PAL_VLYNQ_HND) has to be appended to the chain before this handle can be used for any purpose.**
	Returns	On failure, it returns a NULL, otherwise, it returns a valid handle to the instance so initialized. Refer to Appendix A for details.

5.1.3 PAL_sysVlynqCleanUp( ) – Un-Initialize the VLYNQ control module.

PAL_Result	PAL_sysVlynqCleanUp	(	PAL_VLYNQ_HND*	vlynq)
		vlynq	Handle to the VLYNQ module instance; would have been returned by PAL_sysVlynqInit( ). Consequence of using this handle after a call to this function is undefined.	
		Returns	0 – on success. -1 – on failure. Refer to Appendix A for details.	

5.1.4 PAL_sysVlynqDevCreate( ) – Creates a device reference.

```
PAL_VLYNQ_DEV_HND*  PAL_sysVlynqDevCreate  (  PAL_VLYNQ_HND*   vlynq,
                                              char*            name,
                                              Uint32           instance,
                                              Int32            reset_bit
                                              Bool             peer   )
```

vlynq	The VLYNQ module instance to which a device has to be added; this value would have been returned by `PAL_sysVlynqInitSoc( )`.
name	Name of the device; shall not exceed 30 characters and shall be NULL terminated.
instance	The instance of the device; it is 0 based.
reset_bit	The reset bit for the VLYNQ device in the perspective of the system. Valid only for the on-board device. If not applicable, a value of –1 shall be set.
peer	Identifies whether the device is connected to the local or peer VLYNQ module.
Returns	On failure, returns NULL otherwise, returns a valid handle to the device instance. Refer to Appendix A for details.

5.1.5 PAL_sysVlynqDevDestroy( ) – Destroys the device reference.

PAL_Result	PAL_sysVlynqDevDestroy	(	PAL_VLYNQ_DEV_HND* vlynq_dev_destroy)
	vlynq_dev		The VLYNQ device instance which would have been returned by `PAL_sysVlynqDevCreate( )`. Use of this handle after calling this function is undefined.
	Returns		0 - On success -1 – On failure.

5.1.6 PAL_sysVlynqMapRegion( ) – Map the memory regions of the device.

```
PAL_Result    PAL_sysVlynqMapRegion    (  PAL_VLYNQ_HND*        vlynq,
                                          Bool                  remote,
                                          Uint32                region_id,
                                          Uint32                rx_offset,
                                          Uint32                rx_size,
                                          PAL_VLYNQ_DEV_HND*    vlynq_dev  )
```

vlynq	The VLYNQ instance, which on which regions have to be mapped. This value would have been returned by `PAL_sysVlynqInitSoc( ).`
remote	Identifies whether the region to be mapped is on remote/peer or local VLYNQ. A device connected to peer VLYNQ **exports** its memory by programming the RX Region registers of the peer VLYNQ and **imports** the memory regions by programming the RX Registers of the local VLYNQ. So, for a "peer" device, it should set the remote flag as true (1) to export its regions. Similarly, a device connected to local VLYNQ **exports** its memory by programming the RX Region registers of the local VLYNQ and **imports** region by programming the peer VLYNQ. So, setting remote as true (1) shall mean that the device wants to imports memory regions. The parameter has to be viewed in the context of the device for which memory mapping is being envisaged. 0 (false) => local 1 (true) => remote
region_id	Identifier of the RX region; valid values are from 0 to 3.
rx_offset	The offset of the region in the physical memory map of the local or remote SOC.
rx_size	The size of the region to be mapped.
vlynq_dev	The device instance for which certain regions have to be mapped. This value would have been returned by `PAL_sysVlynqDevCreate( ).`
Returns	0 – on success. -1 – on failure

5.1.7 PAL_sysVlynqMappedRegion ( ) – Return the Mapped Region configuration for Local/Peer.

void	PAL_sysVlynqMappedRegion (	PAL_VLYNQ_T *	p_vlynq_mapped,
		Uint32	address,
		Uint32	peer)

p_vlynq_mapped	Handle to the opened Resource Manager.
address	Return values of TxMap,RxMap.size[4] & RxMap.offset[4]
peer	The flag whether the device is associated with the local VLYNQ or the peer.
Returns	None

5.1.8 PAL_sysVlynqUnMapRegion( ) – UnMap the memory regions of the device.

PAL_Result	PAL_sysVlynqUnMapRegion (	PAL_VLYNQ_HND*	vlynq,
		Bool	remote,
		Uint32	region_id,
		PAL_VLYNQ_DEV_HND*	vlynq_dev)

vlynq	The VLYNQ instance, on which previously mapped regions have to be un-mapped.
remote	Identifies whether the region to be mapped is on remote/peer or local VLYNQ. The parameter has to be viewed in the context of the device for which memory mapping is being envisaged. 0 (false) => local 1 (true) => remote
region_id	Identifier of the RX region; valid values are from 0 to 3.
vlynq_dev	The device instance for which certain regions have been previously mapped; and now are being un-mapped.
Returns	0 – on success. -1 – on failure

5.1.9 PAL_sysVlynqMapIrq( ) – Maps the IRQ hardware line onto the VLYNQ.

```
PAL_Result  PAL_sysVlynqMapIrq  (  PAL_VLYNQ_HND*      vlynq,
                                   Uint32             irq_hw_line,
                                   Uint32             irq,
                                   PAL_VLYNQ_DEV_HND* vlynq_dev  )
```

vlynq	The vlynq instance on which the IRQ mapping is to be carried out. . This value would have been returned by `PAL_sysVlynqInitSoc( ).`
irq_hw_line	Identifies the hardware vector line (in the perspective of the VLYNQ module), which runs from the device to the VLYNQ module. The valid values are 0 to 7 (both inclusive).
irq	Identifies the IRQ number to which the interrupt hardware line shall be mapped. The valid values are 0 to 31 (both inclusive). It is the responsibility of the user to ensure that there is a unique irq_hw_line to irq mapping within a VLYNQ chain.
vlynq_dev	The device instance for which certain hardware interrupts have to be mapped on the VLYNQ; this value would have been returned by `PAL_sysVlynqDevCreate().`
Returns	0 – on success. -1 – on failure.

5.1.10 PAL_sysVlynqUnMapIrq( ) – UnMaps the IRQ hardware line.

PAL_Result	PAL_sysVlynqUnMapIrq	(	PAL_VLYNQ_HND*	vlynq,
			Uint32	irq,
			PAL_VLYNQ_DEV_HND*	vlynq_dev)

vlynq	The VLYNQ instance on which the un-mapping has to be carried out.
irq	Identifies the IRQ number for which the interrupt hardware line shall be un-mapped. The valid values are 0 to 31 (both inclusive).
vlynq_dev	The device instance for which certain hardware interrupts are required to be un-mapped on VLYNQ; this value would have been returned by `PAL_sysVlynqDevCreate()`.
Returns	0 – on success. -1 – on failure.

5.1.11 PAL_sysVlynqChainAppend( ) –Append to the VLYNQ chain.

PAL_Result	PAL_sysVlynqChainAppend	(	PAL_VLYNQ_HND* this, PAL_VLYNQ_HND* to)
		this	The VLYNQ module instance, which has to be added to the chain (away from the root). This value would have been returned by `PAL_sysVlynqInitSoc( ).`
		to	The VLYNQ module instance, to which "this" has to be chained (away from the root). . This value would have been returned by `PAL_sysVlynqInitSoc( ).`
		Returns	0 – on success. -1 – on failure.

5.1.12 PAL_sysVlynqAddDevice( ) –Add the device reference into VLYNQ.

PAL_Result	PAL_sysVlynqAddDevice	(	PAL_VLYNQ_HND* vlynq, PAL_VLYNQ_DEV_HND* vlynq_dev, Bool peer)
	vlynq		The VLYNQ module instance, to which the device instance has to be added. This value would have been returned by `PAL_sysVlynqInitSoc( ).`
	vlynq_dev		The device instance, which has to be added into the VLYNQ. This value would have been returned by `PAL_sysVlynqDevCreate().`
	peer		The flag whether the device is associated with the local VLYNQ or the peer.
	Returns		0 – on success. -1 – on failure.

5.1.13 PAL_sysVlynqRemoveDevice( ) – Removes the device reference from VLYNQ.

PAL_Result	PAL_sysVlynqRemoveDevice	(	PAL_VLYNQ_HND* vlynq, PAL_VLYNQ_DEV_HND* vlynq_dev)
		vlynq	The VLYNQ module instance, from which the device instance has to be removed.
		vlynq_dev	The device instance, which is to be removed from the VLYNQ.
		Returns	0 – on success. -1 – on failure.

5.1.14 PAL_sysVlynqChainUnAppend( ) – Remove (the tail) from the VLYNQ chain.

PAL_Result	PAL_sysVlynqChainUnAppend	(	PAL_VLYNQ_HND* this, PAL_VLYNQ_HND* from)
		this	The VLYNQ module instance, which has to be removed from the chain (away from the root).
		from	The VLYNQ module instance, from which "this" has to be removed (away from the root).
		Returns	0 – on success. -1 – on failure.

5.1.15 PAL_sysVlynqRootIsr( ) – The Root ISR; register it with the system.

PAL_Result	PAL_sysVlynqRootIsr	(Int	*p_vlynq)
	p_vlynq	Pass this parameter along while attaching the HWI to the PAL_sysVlynqRootIsr(). It is the address of the VLYNQ handle of the root VLYNQ module running.	
	Returns	0 – on success. -1 – error.	

5.2 For Drivers et al.

5.2.1 PAL_sysVlynqDevFind( ) – Get the handle for the device.

PAL_VLYNQ_DEV_HND* PAL_sysVlynqDevFind	(const char *name, Uint8 instance)	
	name	The name of the device whose information is being sought. This should same as that used in the `PAL_sysVlynqDevCreate( ).`
	instance	The instance of the remote device. This should same as that used in the `PAL_sysVlynqDevCreate( ).`
	Returns	NULL, if no device could be found or a valid handle.

5.2.2 PAL_sysVlynqDevGetVlynq( ) – Get the VLYNQ for this device.

PAL_VLYNQ_HND*	PAL_sysVlynqDevGetVlynq	(	PAL_VLYNQ_DEV_HND* vlynq_dev_get)
	vlynq_dev		The device whose associated VLYNQ control module is being sought. This would have been returned by `PAL_sysVlynqDevFind( )`.
	Returns		NULL on error or a valid VLYNQ handle.

5.2.3 PAL_sysVlynqGetDevBase( ) – Get the physical base address of the device.

```
PAL_Result  PAL_sysVlynqGetDevBase  (  PAL_VLYNQ_ HND*       vlynq,
                                       Uint32               offset,
                                       Uint32               *base_addr
                                       PAL_VLYNQ_DEV_HND*   dev    )
```

vlynq	The instance on which the physical base address is being sought.
offset	The offset (in bytes) into the remote device memory map.
base_addr	The placeholder for the evaluated physical base address of the remote device in the context of the root SOC. The caller manages the memory for this.
dev	The handle to remote device instance; would have been returned by `PAL_sysVlynqDevFind( )`. The offset is being sought for this device.
Returns	0 – on success. -1 – on failure.

5.2.4 PAL_sysVlynqDevFindIrq( ) – Get the mapped interrupts of the device..

PAL_Result	PAL_sysVlynqDevFindIrq	(	PAL_VLYNQ_DEV_HND* vlynq_dev_irq, Uint8 *irq, Uint32 num_irq)
		dev	The handle to remote device instance; would have been returned by `PAL_sysVlynqDevFind( ).`
		irqs	The placeholder for the identifiers of the IRQ(s) vectors in the perspective of the chain. The caller is responsible for managing the memory. Typically, the caller shall allocate memory for atleast “num_irqs”.
		num_irqs	The number of interrupts to be read; this should be same as the number of interrupts for which placeholder has been provided.
		Returns	0 – on success. -1 – if the placeholder is inadequate. -2 – other errors.

5.2.5 PAL_sysVlynqDevGetResetBit( ) – Get the reset bit of the device.

PAL_Result PAL_sysVlynqDevGetResetBit (PAL_VLYNQ_DEV_HND* vlynq_dev_bit,
Uint32 *reset_bit)

dev	The handle to remote device instance; would have been returned by `PAL_sysVlynqDevFind( ).`
*reset_bit	The placeholder for the identifiers of reset bit. **Only for on board remote devices for which hardware reset provisions have been made.**
Returns	0 – on success. -1 – error.

5.2.6 PAL_sysVlynqAddIsr( ) – Install the ISR for the device.

```
PAL_Result  PAL_sysVlynqAddIsr  (  PAL_VLYNQ_HND*                  vlynq,
                                   Uint32                         irq,
                                   PAL_VLYNQ_DEV_ISR_FN*          dev_isr,
                                   PAL_VLYNQ_DEV_ISR_PARAM_GRP    isr_param  )
                                   _T*
```

vlynq	The VLYNQ module instance for whose associated remote device, an isr is being installed.
irq	One of the identifier of the IRQ as returned by `PAL_sysVlynqGetDevIrq( )or` VLYNQ module specific interrupt..
dev_isr	The OS specific function pointer (ISR handler); the type of the function PAL_VLYNQ_DEV_ISR_FN shall be defined for each of the OS by the user. Driver for the remote device shall load appropriate function instance. Refer to Appendix A for details. The number of parameters in the signature of this function should be exactly same as specified by PAL_VLYNQ_DEV_ISR_PARM_NUM.
isr_param	It is a collection of parameters for each instance of the remote device driver. The user shall define the type PAL_VLYNQ_DEV_ISR_PARAM_GRP_T for the collection of the parameters for each OS. The only caveat here is that the name of the members of the PAL_VLYNQ_DEV_ISR_PARAM_GRP_T should start (i.e. first member) with arg0 (inclusive) and should continue in sequence of arg1, arg2… up to arg9 (inclusive); it is assumed that no OS implementation shall require more than 10 callback ISR parameters. User shall define and specify the number of ISR parameter, PAL_VLYNQ_DEV_ISR_PARM_NUM as required for a specific OS implementation; this should not exceed 10. For a value, less than 10, the member names shall start as arg0, arg1… up to arg< PAL_VLYNQ_DEV_ISR_PARM_NUM– 1>. The keywords for member names here are arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8 and arg9. Refer to Appendix A for details.
Returns	0 – on success. -1 – error.

5.2.7 PAL_sysVlynqRemoveIsr( ) – Uinstall the previously installed ISR.

PAL_Result	PAL_sysVlynqRemoveIsr	(	PAL_VLYNQ_HND* vlynq, Uint32 irq, PAL_VLYNQ_DEV_ISR_PARAM_GRP_T * isr_param)
	vlynq		The VLYNQ module instance for whose associated remote device, an isr is being removed.
	irq		The identifier of the IRQ as provided in the `PAL_sysVlynqAddIsr( )`.
	isr_param		Same as that provided in the `PAL_sysVlynqAddIsr( )`.
	Returns		0 – on success. -1 – error.

5.2.8 PAL_sysVlynqDevCbRegister( ) – Register for the callbacks.

PAL_Result	PAL_sysVlynqDevCbRegister	(	PAL_VLYNQ_DEV_HND*	vlynq_dev_cb,
			PAL_VLYNQ_DEV_DRV_CB_FN	cb_fn,
			void*	this_driver)

dev	The handle to remote device instance; as returned by `PAL_sysVlynqDevFind( ).`
func	typedef int (*PAL_VLYNQ_DEV_CB_FN)(void* this_driver, Uint32 condition, Uint32 condition value); The callback function that is be called by implementation for local error, or remote error update conditions. Refer to Appendix A for details.
this_driver	The driver instance, which is registering for the callbacks from the VLYNQ implementation.
Returns	0 – on success. -1 – error.

5.2.9 PAL_sysVlynqDevCbUnregister( ) – Unregister the callbacks.

PAL_Result	PAL_sysVlynqDevCbUnregister	(	PAL_VLYNQ_DEV_HND*	vlynq_dev_uncb,
			void*	this_driver)

dev	The handle to remote device instance; as returned by `PAL_sysVlynqDevFind( ).`
this_driver	The driver instance, which is registered for the callbacks from the VLYNQ implementation.
Returns	0 – on success. -1 – error.

5.3 Control API(s)

5.3.1 PAL_sysVlynqIoctl( ) – Read/Write register of the VLYNQ module.

PAL_Result	PAL_sysVlynqIoctl	(	PAL_VLYNQ_HND*	vlynq,
			Uint32	cmd,
			Uint32*	command_val)
		vlynq	The VLYNQ module instance, whose register has to be controlled; this value would have been returned by `PAL_sysVlynqInitSoc()`.	
		cmd	Various read and write commands to be carried out. Refer below for the 32 bit break up of the command.	
		Command_val	For write command(s) this provides the value to be written and for read operations it provides the placeholder for the value to be read.	
		Returns	0 – on success. -1 – on failure.	

32 bit Command :

31:Bit Op	30: R/W*	29: Peer	28-24 Reserved	23-16 Major id	15-8 Reserved	7-0 Minor id

If Bit Op is not set, the major id identifies the commands for raw 32-bit accesses or any specific operation. For now, if major command is 32 bit accesses, then register id refer to minor id otherwise minor id are don't care.

If Bit Op is set, the major id identifies a specific register for a select bit operation. Minor id then shall identify the registers.

Note: Refer to Appendix B for details.

5.3.2 PAL_sysVlynqClockConfig( ) – Configures the Clock for the VLYNQ bridge.

```
PAL_Result  PAL_sysVlynqConfigClock  (  PAL_VLYNQ_HND*                 vlynq,
                                        PAL_VLYNQ_CLOCK_DIR_ENUM_T    local_clock_dir,
                                        PAL_VLYNQ_CLOCK_DIR_ENUM_T    peer_clock_dir,
                                        Uint8                         local_clock_div,
                                        Uint8                         peer_clock_div,  )
```

vlynq	The VLYNQ module instance, whose register has to be controlled; this value would have been returned by `PAL_sysVlynqInitSoc()`.
local_clock_dir	The clock direction for the local VLYNQ. Refer to Appendix A for details.
peer_clock_dir	The clock direction for the peer VLYNQ. Refer to Appendix A for details.
local_clock_div	The divisor for the local clock. Valid values are 1 to 255.
peer_clock_div	The divisor for the peer clock. Valid values are 1 to 255.
Returns	0 – on success. -1 – on failure.

5.4 Interrupts Galore

5.4.1 PAL_sysVlynqGetForIrq( ) – Get the VLYNQ for the IRQ.

PAL_VLYNQ_HND*	PAL_sysVlynqGetForIrq	(	PAL_VLYNQ_HND*	root,	
			Uint32	irq	)

root	The VLYNQ chain identifier (the root VLYNQ); the interrupt numbers are unique with in a chain.
irq	The interrupt number whose association with a VLYNQ module is being sought. The valid values are 0 to 31 (both inclusive).
Returns	0 – on success. -1 – error.

5.4.2 PAL_sysVlynqSetIrqPol( ) – Set the polarity of the hardware IRQ line.

PAL_Result	PAL_sysVlynqSetIrqPol	(	PAL_VLYNQ_HND*	vlynq,
			Uint32	irq,
			PAL_VLYNQ_IRQ_POL_ENUM_T	*polarity)

vlynq	The VLYNQ instance whose hardware interrupt line is to be set for polarity.
irq	The interrupt number representing the mapped interrupt hardware line on the VLYNQ module. This should be unique for a given VLYNQ chain.
polarity	The polarity of the line. Refer to the data structure in Appendix A.
Returns	0 – on success. -1 – error.

5.4.3 PAL_sysVlynqSetIrqType( )- Set the type of the hardware IRQ line.

```
PAL_Result  PAL_sysVlynqSetIrqType  (  PAL_VLYNQ_HND*              vlynq,
                                       Uint32                     irq,
                                       PAL_VLYNQ_IRQ_TYPE_ENUM_T  type  )
```

vlynq	The VLYNQ instance whose hardware interrupt line is to be set for trigger type.
irq	The interrupt number representing the mapped interrupt hardware line on the VLYNQ module. This should be unique for a given VLYNQ chain.
type	The type of the interrupt line. Refer to the data structure in the Appendix A.
Returns	0 – on success. -1 – error.

5.4.4 PAL_sysVlynqGetIrqPol ( ) – Get the polarity of the hardware IRQ line.

PAL_Result	PAL_sysVlynqGetIrqPol	(	PAL_VLYNQ_HND*	vlynq,	
			Uint32	irq,	
			PAL_VLYNQ_IRQ_POL_ENUM_T*	polarity	)

vlynq	The VLYNQ instance whose hardware interrupt line is to being looked up for polarity setting.
Irq	The interrupt number representing the mapped interrupt hardware line on the VLYNQ module. This should be unique for a given VLYNQ chain.
polarity	The polarity of the line. The caller manages memory. Refer to the data structure in the Appendix A.
Returns	0 – on success. -1 – error.

5.4.5 PAL_sysVlynqGetIrqType( ) – Get the type of the hardware IRQ type.

```
PAL_Result  PAL_sysVlynqGetIrqType  (  PAL_VLYNQ_HND*              vlynq,
                                       Uint32                    irq,
                                       PAL_VLYNQ_IRQ_POL_ENUM_T  type  )
                                       *
```

vlynq	The VLYNQ instance whose hardware interrupt line is being looked up for trigger type.
Irq	The interrupt number representing the mapped interrupt hardware line on the VLYNQ module. This should be unique for a given VLYNQ chain.
type	The type of the interrupt line. The caller manages memory. Refer to the data structure in the Appendix A.
Returns	0 – on success. -1 – error.

5.4.6 PAL_sysVlynqGetIrqCount( ) – Get the number of times this IRQ occurred.

Uint32	PAL_sysVlynqGetIrqCount	(	PAL_VLYNQ_HND*	vlynq,
			Uint32	irq
			Uint32	*count)

vlynq	The VLYNQ module instance for whose associated “irq” has to be queried for number of dispatches so far.
irq	The interrupt number representing the mapped interrupt hardware line on the VLYNQ module. This should be unique for a given VLYNQ chain.
count	Placeholder to store the number of times the interrupt “irq” has been raised. The value in the placeholder is valid only if the function returns success.
Returns	0 – on success. -1 – error.

5.4.7 PAL_sysVlynqDisableIrq( ) – Disable the IRQ.

Uint32	PAL_sysVlynqDisableIrq	(	PAL_VLYNQ_HND*	vlynq,
			Uint32	irq)

vlynq	The VLYNQ module instance for whose associated “irq” hardware line is to be disabled for generating interrupts.
irq	The interrupt number representing the mapped interrupt hardware line on the VLYNQ module. This should be unique for a given VLYNQ chain.
Returns	0 – on success. -1 – error.

5.4.8 PAL_sysVlynqEnableIrq( ) – Enable the IRQ.

Uint32	PAL_sysVlynqEnableIrq	(	PAL_VLYNQ_HND*	vlynq,
			Uint32	irq)

vlynq	The VLYNQ module instance for whose associated “irq” hardware line is to be enabled for generating interrupts.
irq	The interrupt number representing the mapped interrupt hardware line on the VLYNQ module. This should be unique for a given VLYNQ chain.
Returns	0 – on success. -1 – error.

5.5 Status and Utility

5.5.1 PAL_sysVlynqGetLinkStatus( ) – Get the status of the of the VLYNQ module.

Bool	PAL_sysVlynqGetLinkStatus	(	PAL_VLYNQ_HND*	vlynq)
		vlynq	The VLYNQ module instance, whose link status with the peer is being requested.	
		Returns	1 – on link. 0 – on link failure.	

5.5.2 PAL_sysVlynqGetNumRoot( ) – Get the number of the root VLYNQ(s).

Int32	PAL_sysVlynqGetNumRoot	(	void	)
	Returns		The number of root VLYNQ modules on the SOC. A value of 0 means no VLYNQ. 1 means an root index of 0, 2 mean indices0 and 1.	

5.5.3 PAL_sysVlynqGetRoot( ) – Get the handle to the specified root VLYNQ.

PAL_VLYNQ_HND*	PAL_sysVlynqGetRoot	(	Int32	index)
		index	0 based. Starts at 0 and extends to (inclusive of) one less number of roots returned by `PAL_sysVlynqGetNumRoot ( )`	
		Returns	NULL, on failure otherwise a valid handle to access the root VLYNQ module.	

5.5.4 PAL_sysVlynqGetRootVLYNQ( ) – Get root for the given VLYNQ.

PAL_VLYNQ_HND*	PAL_sysVlynqGetRoot	(	PAL_VLYNQ_HND*	vlynq)

vlynq	Handle to the VLYNQ whose root is being sought.
Returns	NULL, on failure otherwise a valid handle, to access the root VLYNQ module.

5.5.5 PAL_sysVlynqGetRootAtBase( ) – Get the root VLYNQ at the base address.

PAL_VLYNQ_HND*	PAL_sysVlynqGetRootAtBase	(	Uint32	base_addr)
		base_addr		The virtual base address of the VLYNQ module on the SOC.
		Returns		NULL, on failure otherwise a valid handle to access the root VLYNQ module.

5.5.6 PAL_sysVlynqGetBaseAddr( ) – Returns the base address of the VLYNQ.

PAL_Result	PAL_sysVlynqGetBaseAddr	(	PAL_VLYNQ_HND* vlynq, Uint32 *base_addr)
		vlynq	The Vlynq module whose base address is being sought.
		base_addr	The placeholder for the virtual base address of the VLYNQ module on the SOC.
		Returns	0 on success -1 on failure.

5.5.7 PAL_sysVlynqGetNext( ) – Get the next VLYNQ module in the chain.

PAL_VLYNQ_HND*	PAL_sysVlynqGetNext	(	PAL_VLYNQ_HND* this	)
	this		The VLYNQ module instance whose next chained entity (away from the root) is being sought.	
	Returns		NULL, on failure otherwise a valid handle to access the neighbor module.	

5.5.8 PAL_sysVlynqIsLast( ) – Is this VLYNQ module the last one in the chain.

Int32	PAL_sysVlynqIsLast	(	PAL_VLYNQ_HND*	this)
	this		The VLYNQ module instance to be ascertained whether last in the chain (away from the root).	
	Returns		0 – on false. 1 – on true.	

5.5.9 PAL_sysVlynqGetChainLength( ) – Get the length of the VLYNQ chain.

Int32	PAL_sysVlynqGetChainLength	(	PAL_VLYNQ_HND*	this)
		this	The starting VLYNQ module instance inclusive of which, the number of VLYNQ modules existing in the chain (away from root) is being sought. If this happens to be the root VLYNQ module, then the length of the entire chain is ascertained.	
		Returns	The number of VLYNQ modules in the chain (away from the root).	

5.5.10 PAL_sysVlynqDump( ) – Dumps vital VLYNQ information into the buffer.

Int32	PAL_sysVlynqDump	(	PAL_VLYNQ_HND*	vlynq,
			Uint32	dump_type,
			char*	buf,
			Int32	limit,
			Int32*	eof)
		vlynq	The vlynq instance for which the dump has to be provided.	
		dump_type	Identifies the information being sought. Some of the commands can be raw byte dump of the hardware, complete chain dump, raw register values or specific register value with enumeration such as status register or control register. Refer Appendix B for details.	
		buf	The placeholder for the buffer. After the function, the buffer can be printed out for reading and information.	
		limit	The size of the buffer. It is strongly recommended to provide a buffer of at least 4096 bytes.	
		eof	Whether the buffer was insufficient.	
		Returns	The number of bytes that have been formatted and placed in the buffer. -1 on error.	

6 Appendix A

6.1 Enumerators and typedefs

```
typedef enum pal_vlynq_clock_dir_enum
{
    pal_vlynq_clk_in = 0,
    /**< Sink the clock.         */
    pal_vlynq_clk_out
    /**< Source the clock         */
} PAL_VLYNQ_CLOCK_DIR_ENUM_T;

typedef enum pal_vlynq_endian_enum
{
    pal_vlynq_ignore_en = 0,
    /**< Ignore the endianess    */
    pal_vlynq_little_en = (Int32)LITTLE_ENDIAN,
       /**< For little endianess    */
    pal_vlynq_big_en    = (Int32)BIG_ENDIAN
     /**< For big endianess        */
} PAL_VLYNQ_ENDIAN_ENUM_T;

typedef enum
{
   pal_vlynq_high_irq_pol = 0,
   /**< IRQ polarity high. */
   pal_vlynq_low_irq_pol
    /**< IRQ polarity low. */
}PAL_VLYNQ_IRQ_POL_ENUM_T;

typedef enum
{
   pal_vlynq_level_irq_type = 0,
   /**< IRQ level triggered. */
   pal_vlynq_edge_irq_type
    /**< IRQ edge triggered.  */
}PAL_VLYNQ_IRQ_TYPE_ENUM_T;


typedef Int32 (*PAL_VLYNQ_DEV_CB_FN)(void* this_driver , Uint32 condition, Uint32 value);

typedef void PAL_VLYNQ_HND;
typedef void PAL_VLYNQ_DEV_HND;
```

6.2 PAL_VLYNQ_CONFIG_T (pal_sysvlynq.h)

```
typedef struct
{
    Uint8                   on_soc;
    /**< 1 => the VLYNQ module is on the SOC running the code, 0 => otherwise.    */

    PAL_VLYNQ_INIT_ERR_ENUM_T error_status;
    /**<   status of error     */

    char                    error_msg[50];
    /**<  holds error message string      */

    Uint32                  base_addr;
    /**< Virtual Base Address of the module     */

    Uint32                  init_timeout_in_ms;
    /**< The number of millsecs that the software should allow for initialization
      *  to complete     */
    /**<  functionality not implemented  */

    Uint8                   local_clock_div;
    /**< The clock divisor for the local VLYNQ module      */

    PAL_VLYNQ_CLOCK_DIR_ENUM_T local_clock_dir;
     /**< The clock direction; sink or source for the local VLYNQ module     */

    Uint8                   local_intr_local;
    /**< 1 => interrupts are being handled locally or 0 => sent interrupt packate over the
interface     */

    Uint8                   local_intr_vector;
    /**<    The IRQ vector# to be used on the local VLYNQ module. Valid values are
      *    0 to 31. Should be unique      */

    Uint8                   local_intr_enable;
    /**<     1 => enable the local irq vector or => disable      */

    Uint8                   local_int2cfg;
    /**<   Valid only if intr_local is set.
      *            1 => write to the local VLYNQ interrupt pending register;
      *            0 => write to the location refered by local address pointer       */

    Uint32                  local_intr_pointer;

#ifdef INCLUDE_ENDIAN_REGISTER
    /**<     Address to which the irq should be written to; valid only if int2cfg is not
set */
    PAL_VLYNQ_ENDIAN_ENUM_T  local_endianness;
#endif

    /**<    Endianess of the local VLYNQ module      */
    Uint32                  local_tx_addr;
    /**< The physical portal address of the local VLYNQ      */

    PAL_VLYNQ_RTM_CFG_ENUM_T local_rtm_cfg_type;
     /**< The RTM configuration for the local VLYNQ      */

    Uint8                   local_rtm_sample_value;
    /**< The RTM sample value for the local VLYNQ, valid only if forced rtm cfg
      *  type is selected     */

    Bool                    local_tx_fast_path;
```

```
    /**<    TX Fast path 0 => slow path, 1 => fast path.     */

    Uint8                   peer_clock_div;
    /**<    The clock divisor for the peer VLYNQ module     */

    PAL_VLYNQ_CLOCK_DIR_ENUM_T peer_clock_dir;
    /**<    The clock direction; sink or source for the peer VLYNQ module    */

    Uint8                   peer_intr_local;
    /**< 1 => interrupts are being handled by peer VLYNQ or 0 => sent over the bus      */

    Uint8                   peer_intr_enable;
    /**< 1 => enable the peer irq vector or => disable      */

    Uint8                   peer_intr_vector;
    /**<  The IRQ vector# to be used on the peer VLYNQ module. Valid values are 0 to 31.
      *   Should be unique                                        */

    Uint8                   peer_int2cfg;
    /**<         Valid only if intr_local is set.
     *            1 => write to the local VLYNQ interrupt pending register;
     *           0 => write to the location refered by local address pointer   */

    Uint32                  peer_intr_pointer;

#ifdef INCLUDE_ENDIAN_REGISTER
    /**< Address to which the irq should be written to; valid only if int2cfg is not set
*/
    PAL_VLYNQ_ENDIAN_ENUM_T  peer_endianness;
#endif
    /**<    Endianess of the local VLYNQ module      */

    Uint32                  peer_tx_addr;
    /**<  The physical portal address of the peer VLYNQ      */

    PAL_VLYNQ_RTM_CFG_ENUM_T peer_rtm_cfg_type;
    /**<  The RTM configuration for the peer VLYNQ      */

    Uint8                   peer_rtm_sample_value;
    /**< The RTM sample value for the local VLYNQ, valid only if forced rtm cfg type is
selected   */

    Bool                    peer_tx_fast_path;
    /**<    TX Fast path 0 => slow path, 1 => fast path     */

    Bool                    init_swap_flag;
    /**<  Is the Vlynq module in Endian swapped state to begin with      */

} PAL_VLYNQ_CONFIG_T;
```

6.3 User defines

```
/* This sample code should be realized in the pal_sysvlynqOs.h  for each
OS */

/** Number of params with ISR. */
#define PAL_VLYNQ_DEV_ISR_PARM_NUM 3

/* To keep the ISR dispatch simple we align the signature of the ISR
 * to that stipulated by the OS.
 */
typedef void (*PAL_VLYNQ_DEV_ISR_FN)(int, void*, void*);


typedef struct
{
   /* types same as defined in the signature and in same strict order. The caveat here is
        keywords for the name of the members of this structure; they start with arg0 ...
        arg9. The name should always start with arg0 and shall continue up to the arg
        <number of variables - 1>.

        At this time it is assumed that there shall be no requirement to define more than
        10 arguments for the ISR routine. Sample:
*/

    Uint32 arg0,
    Uint8* arg1,
    void*  arg2

} PAL_VLYNQ_DEV_ISR_PARAM_GRP_T;


/* Now include the pal_sysvlynq.h */
```

7 Appendix B

```
#ifndef __PAL_VLYNQ_IOCTL_H__
#define __PAL_VLYNQ_IOCTL_H__

#define PAL_VLYNQ_IOCTL_BIT_CMD             (1 << 31)
#define PAL_VLYNQ_IOCTL_READ_CMD            (1 << 30)
#define PAL_VLYNQ_IOCTL_REMOTE_CMD          (1 << 29)


#define PAL_VLYNQ_IOCTL_MAJOR_VAL(val)          ((val & 0xff) << 16)
#define PAL_VLYNQ_IOCTL_MAJOR_DE_VAL(cmd)  ((cmd >> 16) & 0xff)

#define PAL_VLYNQ_IOCTL_MINOR_VAL(val)          (val & 0xff)
#define PAL_VLYNQ_IOCTL_MINOR_DE_VAL(cmd)  (cmd & 0xff)

/* Major commands; if bit option is not selected. */
#define PAL_VLYNQ_IOCTL_REG_CMD             (0x00) /* Shall use vlynq regs as minor cmd*/
#define PAL_VLYNQ_IOCTL_PREP_LINK_DOWN     (0x01)  /* Prepare to teardown the link. */
#define PAL_VLYNQ_IOCTL_PREP_LINK_UP          (0x02)  /* Setup   now the link is up. */
#define PAL_VLYNQ_IOCTL_CLEAR_INTERN_ERR  (0x03)  /* Clear internal interrupt errors. */

/* Control Register parameters, valid for bit operations. */
/**< Control register RESET command */
#define PAL_VLYNQ_IOCTL_CNT_RESET_CMD      (0x00u)
/**< Control register ILOOP command */
#define PAL_VLYNQ_IOCTL_CNT_ILOOP_CMD      (0x01u)
/**< Control register AOPT command :  Write */
#define PAL_VLYNQ_IOCTL_CNT_AOPT_CMD       (0x02u)
/**< Control register INT2CFG command */
#define PAL_VLYNQ_IOCTL_CNT_INT2CFG_CMD    (0x07u)
/**< Control register INTVEC command */
#define PAL_VLYNQ_IOCTL_CNT_INTVEC_CMD     (0x08u)
/**< Control register INT_EN command */
#define PAL_VLYNQ_IOCTL_CNT_INT_EN_CMD     (0x0du)
/**< Control register INT_LOC command */
#define PAL_VLYNQ_IOCTL_CNT_INT_LOC_CMD    (0x0eu)
/**< Control register CLK_DIR command */
#define PAL_VLYNQ_IOCTL_CNT_CLK_DIR_CMD    (0x0fu)
/**< Control register CLK_DIV command : Write */
#define PAL_VLYNQ_IOCTL_CNT_CLK_DIV_CMD    (0x10u)
/**< Control register CLK_MOD command */
#define PAL_VLYNQ_IOCTL_CNT_CLK_MOD_CMD    (0x15u)
/**< Control register TX_FAST command */
#define PAL_VLYNQ_IOCTL_CNT_TX_FAST_CMD    (0x15u)
/**< Control register RTM_SELECT command */
#define PAL_VLYNQ_IOCTL_CNT_RTM_SELECT_CMD (0x16u)
/**< Control register RTM_SAMPLE command */
```

```
#define PAL_VLYNQ_IOCTL_CNT_RTM_VALIDWR_CMD (0x17u)
/**< Control register RTM_SAMPLE command */
#define PAL_VLYNQ_IOCTL_CNT_RTM_SAMPLE_CMD (0x18u)
/**< Control register SLKPU command : Write */
#define PAL_VLYNQ_IOCTL_CNT_CLK_SLKPU_CMD  (0x1eu)
/**< Control register PMEM command  : Write */
#define PAL_VLYNQ_IOCTL_CNT_PMEM_CMD       (0x1fu)

/* Status Register parameters, valid for bit operations. */

/**< Status register LINK bit in status register */
#define PAL_VLYNQ_IOCTL_STS_LINK          (0x00u)
/**< Status register MPEND bit in status register */
#define PAL_VLYNQ_IOCTL_STS_MPEND         (0x01u)
/**< Status register SPEND bit in status register */
#define PAL_VLYNQ_IOCTL_STS_SPEND         (0x02u)
/**< Status register NFEMP0 bit in status register */
#define PAL_VLYNQ_IOCTL_STS_NFEMP0        (0x03u)
/**< Status register NFEMP1 bit in status register */
#define PAL_VLYNQ_IOCTL_STS_NFEMP1        (0x04u)
/**< Status register NFEMP2 bit in status register */
#define PAL_VLYNQ_IOCTL_STS_NFEMP2        (0x05u)
/**< Status register NFEMP3 bit in status register */
#define PAL_VLYNQ_IOCTL_STS_NFEMP3        (0x06u)
/**< Status register LERR bit in status register */
#define PAL_VLYNQ_IOCTL_STS_LERR          (0x07u)
/**< Status register RERR bit in status register */
#define PAL_VLYNQ_IOCTL_STS_RERR          (0x08u)
/**< Status register OFLOW bit in status register */
#define PAL_VLYNQ_IOCTL_STS_OFLOW         (0x09u)
/**< Status register IFLOW bit in status register */
#define PAL_VLYNQ_IOCTL_STS_IFLOW         (0x0Au)
/**< Status register RTM bit in status register */
#define PAL_VLYNQ_IOCTL_STS_RTM           (0x0Bu)
/**< Status register RTM_VAL bit in status register */
#define PAL_VLYNQ_IOCTL_STS_RTM_VAL       (0x0Cu)
/**< Status register SWIDOUT bit in status register */
#define PAL_VLYNQ_IOCTL_STS_SWIDOUT       (0x14u)
/**< Status register MODESUP bit in status register */
#define PAL_VLYNQ_IOCTL_STS_MODESUP       (0x15u)
/**< Status register SWIDIN bit in status register */
#define PAL_VLYNQ_IOCTL_STS_SWIDIN        (0x18u)
/**< Status register SWIDTH bit in status register */
#define PAL_VLYNQ_IOCTL_STS_SWIDTH        (0x18u)
/**< Status register DEBUG bit in status register */
#define PAL_VLYNQ_IOCTL_STS_DEBUG         (0x1du)

/* VLYNQ Registers */

#define PAL_VLYNQ_IOCTL_REVSION_REG         (0x00)
#define PAL_VLYNQ_IOCTL_CONTROL_REG         (0x04)
#define PAL_VLYNQ_IOCTL_STATUS_REG          (0x08)
```

```
#define PAL_VLYNQ_IOCTL_INT_PRIR_REG        (0x0c)
#define PAL_VLYNQ_IOCTL_INT_STS_REG         (0x10)
#define PAL_VLYNQ_IOCTL_INT_PEND_REG        (0x14)
#define PAL_VLYNQ_IOCTL_INT_PTR_REG         (0x18)
#define PAL_VLYNQ_IOCTL_TX_MAP_REG          (0x1c)
#define PAL_VLYNQ_IOCTL_RX1_SZ_REG          (0x20)
#define PAL_VLYNQ_IOCTL_RX1_OFF_REG         (0x24)
#define PAL_VLYNQ_IOCTL_RX2_SZ_REG          (0x28)
#define PAL_VLYNQ_IOCTL_RX2_OFF_REG         (0x2c)
#define PAL_VLYNQ_IOCTL_RX3_SZ_REG          (0x30)
#define PAL_VLYNQ_IOCTL_RX3_OFF_REG         (0x34)
#define PAL_VLYNQ_IOCTL_RX4_SZ_REG          (0x38)
#define PAL_VLYNQ_IOCTL_RX4_OFF_REG         (0x3c)
#define PAL_VLYNQ_IOCTL_CVR_REG             (0x40)
#define PAL_VLYNQ_IOCTL_AUTO_NEG_REG        (0x44)
#define PAL_VLYNQ_IOCTL_MAN_NEG_REG         (0x48)
#define PAL_VLYNQ_IOCTL_NEG_STS_REG         (0x4c)
#define PAL_VLYNQ_IOCTL_ENDIAN_REG          (0x4c)
#define PAL_VLYNQ_IOCTL_IVR30_REG           (0x60)
#define PAL_VLYNQ_IOCTL_IVR74_REG           (0x64)

/* Dumping options, not part of ioctl options. */

#define PAL_VLYNQ_DUMP_ALL_ROOT         (0x10000)
#define PAL_VLYNQ_DUMP_RAW_DATA         (0x20000)
#define PAL_VLYNQ_DUMP_ALL_REGS         (0x30000)
#define PAL_VLYNQ_DUMP_STS_REG          (0x00008)
#define PAL_VLYNQ_DUMP_CNTL_REG         (0x00004)

#endif
```