



FreeBSD on

CAVIUM THUNDERX SYSTEM ON A CHIP

By Zbigniew Bodek and Wojciech Macek

THIS ARTICLE DESCRIBES THE FREEBSD OPERATING SYSTEM PORT FOR THE CAVIUM THUNDERX CN88XX SYSTEM ON A CHIP. THUNDERX IS A NEWLY INTRODUCED ARM64 (ARMV8) SOC DESIGNED FOR THE HIGH-PERFORMANCE AND SERVER MARKETS. IT IS CURRENTLY THE ONLY ONE IN THE ARM WORLD TO INCORPORATE UP TO 96 CPU CORES IN THE SYSTEM ALONG WITH THE TECHNOLOGY TO MAKE IT POSSIBLE.

ThunderX is up to date with the latest trends in the computer architecture industry, including those that are relatively new to FreeBSD, like SR-IOV (Single Root I/O Virtualization), or completely unique, such as ARM GICv3 and ITS.

The main focus here is to provide a bottom-up view of how the FreeBSD platform support for ThunderX was implemented and to depict the benefits and pitfalls of the newly introduced ARMv8 technology in terms of the OS development. The

article also describes key components of the ThunderX system and explains how they were supported in FreeBSD. Finally, possible areas of further improvements are pointed out briefly.

INTRODUCTION

FreeBSD is undoubtedly the most recognizable Unix-like operating system available. One of the main areas of its deployment is the server market, which is still dominated by Intel and AMD-based computers. However, recently in the mobility industry, the highly successful ARM architecture is gaining interest as a foundation for high-performance server SoCs. A turning point in that field was the emergence of a 64-bit ARM implementation (ARMv8) including improved technology to obtain insane multi-core capabilities. Thus, it is important for the FreeBSD community (developers and users) to keep up with the growing interest in ARM-based servers and supply the ecosystem with an ARM64 BSD OS that will be on par with the available Tier-1 x86 platforms.

The motivation behind the work on ThunderX was driven by the actual market need for the FreeBSD OS on that platform, the aim of which is

to become a real alternative for energy-intensive solutions. ThunderX is also the only ARM-based chip in the world to scale up to 48 CPU cores per socket with a possible dual socket configuration (up to 96 CPUs). With the included hardware accelerators for networking, storage, and security, as well as powerful peripheral devices, ThunderX is a perfect base for the server-oriented OS such as FreeBSD and a great engineering challenge for a kernel developer.

The support introduced here is based on foundational work with emulation as a primary target (ARM Foundation Model). Hence the platform support for ThunderX was partly carried out in parallel to the FreeBSD base system development and occasionally was intertwined with it. ThunderX was also the first hardware platform to switch from the ARM emulator. The result is a fully functional OS that supports key chip features such as:

- ◆ PCI Express
- ◆ GICv3 + ITS
- ◆ SMP (single and dual socket)
- ◆ SATA
- ◆ Virtualized Network Interfaces

The description of the general FreeBSD kernel implementation for ARMv8 architecture is beyond the scope of this paper and is discussed only with respect to the work on ThunderX support.

HARDWARE OVERVIEW

Contemporary ARM-based chips very much follow current trends in the computer industry, incorporating multiprocessor capabilities; high-performance, peripheral devices; buses and extensions, as well as various hardware accelerators and virtualization technologies. The 8th architecture revision (ARMv8) moved those concepts to a new level by overcoming previous architectural limitations. ThunderX is a good representative of the new ARM chips generation, as it is the first to utilize majority of the newly introduced features.

Core Complex and CCPI

The heart of the CN88XX SoC is a set of Cavium proprietary ARMv8-compliant CPUs. Each CPU has its own I-cache and D-cache but they share the common 16MB L2 cache. A single package can contain up to 48 CPUs organized in three clusters of 16 CPUs each. The core complex can be connected to another CN88XX

processor using Cavium Coherent Processor Interconnect fabric (CCPI). All CPUs in the system are cache coherent in respect to L1, L2 cache, and DMA accesses. The system coherency capabilities are also extended across CCPI-connected entities. Therefore, the dual socket configuration yields a fully coherent system containing 96 ThunderX CPUs. The potential multi-socket configuration can incorporate four nodes with a total number of 192 CPUs [1].

PORTING

The efforts of porting the FreeBSD operating system to a new platform can usually be divided into a few general stages:

- ◆ Toolchain and build environment support.
- ◆ Kernel loading and bootstrapping.
- ◆ `locore.S` and low-level operations. Kernel start code and low-level machine bring-up as well as basic cache maintenance, atomic operations, synchronization primitives, etc.
- ◆ Elementary system operations. These include virtual memory management in `pmap.c`, exceptions handling, context switching, and interrupts as well as system timers and console support.
- ◆ Drivers for peripheral devices.
- ◆ Userland support.

Most of the listed steps had already been implemented in FreeBSD, as a lot of work has been put into running it on Qemu and ARM Foundation Model. However, the real hardware was still a huge unexplored area. Existing pieces of the code provided a good starting point for ThunderX support. The missing parts of the development mainly concerned:

- ◆ System bootstrap. Every real hardware device has its own quirks, assumptions, and, most of all, custom firmware. Even though ThunderX is an ARMv8-compliant platform, it's in fact a custom ARM implementation, which requires some more care than generic Cortex-A53/A57 tested on Qemu and ARM emulators.
- ◆ Interrupts handling. A massive number of CPU cores and peripheral devices, as well as the PCI-centric architecture demanded a completely new approach to interrupts signaling and a number of changes to both generic ARM64- and ThunderX-specific code.

CONTINUES NEXT PAGE

- ◆ PCI-express.
The connectivity between the ThunderX core complex and peripheral coprocessors is almost entirely based on the PCIe bus. Support for the on-chip PCIe bridges hierarchy was critical in relation to other integrated interfaces such as SATA, network or USB.
- ◆ A complex network controller.
A powerful networking card with IO virtualization and modular architecture.
- ◆ SMP operation with 48/96 CPU cores. Running FreeBSD on a real hardware and in a multi-core environment revealed a list of issues that needed to be investigated and resolved.

SYSTEM BOOTSTRAP

A typical CN88XX boot scenario starts with the on-chip Boot ROM. This stage is supposed to load Boot-level 1 Firmware (BL1) from the SPI-connected FLASH, which will then load further BL2 and BL3 Firmware, and, finally, the control over the system is passed to the Cavium Unified Extensible Firmware Interface (UEFI) bootloader. At that point the ThunderX system and its interfaces are initialized and the boot CPU core is in EL2 exception level (Hypervisor). In order to run FreeBSD, ThunderX needs to load the kernel ELF file and supply it with a machine description such as the DTB (Device Tree Blob), available memory regions, as well as information about the kernel location in DRAM, etc. Hence the next step in the FreeBSD boot process is the native BSD *loader(8)* which in that case is executed in a UEFI runtime environment. *loader(8)* handles kernel acquisition and jumps into its code. ThunderX uses genuine ARM64 *loader(8)*, so almost no modifications needed to be made.

EARLY SYSTEM INITIALIZATION

The very first kernel code being executed is the one in `locore.s`. It performs three fundamental actions:

- ◆ Puts CPU into a well-defined state.
- ◆ Prepares an execution environment for the C code.
- ◆ Forwards information about the modules obtained from the bootloader.

The ARMv8 processors (in AArch64 state) operate in one of the four maximum Exception Levels: EL0-EL3, from which EL0 has the lowest software execution privilege that corresponds to the *User Mode*; EL1 can be called a *Kernel Mode*, EL2 is a *Hypervisor*, and EL3 is used for ARM's

TrustZone Secure Monitor Mode [3]. The start code usually drops the exception level to EL1 (after performing an EL2-specific configuration). At this point, the initial and identity (1:1) kernel mappings are created, which will allow changing the context to the kernel virtual address space when the Memory Management Unit is enabled. Before that happens, all settings related to address translation, caching, and initial system behavior (such as exceptions reception) are applied. Finally, the kernel stack is configured and CPU jumps to the early machine initialization in C code.

ThunderX requires more settings during this stage than the ARM Foundation Model. These include:

- ◆ Enabling EL1 access to the Generic Interrupt Controller's CPU Interface.
By default, the CPU interface cannot be accessed through System Registers from EL1. Access permission has to be granted while still in EL2.
- ◆ Enlargement of the virtual address space in `TCR_EL1`.
On some variants of ThunderX, the physical memory is mapped beyond 512GB. Therefore, if the programmed address space is not big enough, it is impossible to create an identity mapping required to jump from the physical to the virtual address space.

The changes, however, are not strictly ThunderX-specific and will apply to any platform with similar requirements.

INTERRUPTS DELIVERY

Exceptions whose purpose is to indicate to CPU that a certain action took place are called interrupts. There would be no reasonable multi-threading OS without the interrupts support; therefore, they are one of the most fundamental elements of the system. Previous generations of ARM processors often incorporated a so-called ARM Generic Interrupt Controller (GIC) or other proprietary implementation.

Exemplary ARM GIC consists of two main components: Distributor and CPU Interfaces. Typically, interrupt lines from the on-chip devices are wired to the distributor interface, which then, according to its configuration, routes the interrupt signals to the appropriate CPU interfaces. If the interrupt signaling is enabled, the CPU will receive a notification on the appropriate interrupt line (IRQ or FIQ). Finally, the CPU can acquire the interrupt information, such as its number, from the CPU Interface registers and change the pending state

of the interrupt. This architecture works fine for ARMv6/v7 processors but has some serious limitations in terms of:

- ◆ Scalability. Can route interrupts up to 8 CPUcores.
- ◆ Maximum number of interrupts. Each interrupt requires a physical connection to the distributor.
- ◆ No Message Signaled Interrupts support. Cannot use in-band PCI interrupt signaling.
- ◆ Slow access to the CPU Interface registers. Each interrupt requires at least a few read/write sequences to the memory-mapped registers (slow access to device memory and possible TLB misses).

Generic Interrupt Controller v3

Platforms such as ThunderX require improved interrupts handling to provide better SMP utilization, support for PCIe devices, and minimal time penalty per interrupt. These features were introduced with ARM Generic Interrupt Controller v3 [2] in cooperation with the Interrupt Translation Service. The contributed work includes full FreeBSD support for ARM GICv3 and ITS along with the ThunderX-specific quirks, but excluding virtualization extensions.

This article describes the support for the crucial GIC components only and does not cover the implementation of the machine-dependent part of the interrupts handling code that needed to be redesigned for the purpose of this port.

Affinity-based Routing

Unlike earlier GIC architectures, GICv3 incorporates an additional, third component in the form of Re-Distributors, which are memory-mapped entities associated with every CPU in the system. Moreover, the CPU Interface can now be accessed through the CPU's System Registers to speed up interrupts handling after the core gets notification. To overcome the interrupt-to-CPU delivery limitations, the interrupt is now routed based on the Affinity Hierarchy. This means that the interrupt destination is now addressed by the 4-level CPU affinity number in the system. The GICv3 driver configures all SPIs (Shared Peripheral Interrupts) in the global Distributor, but PPIs (Private Peripheral Interrupts), SGIs (Software Generated Interrupts), and a new class of LPIs (Local Peripheral Interrupts) are managed through per-CPU Re-Distributors. Each Re-Distributor needs to be enabled or woken up before it can be used. Fortunately, GICv3 provides an auto-configuration scheme that allows the OS driver to

iterate through a device's memory-mapped region, match the configurator CPU to a correct Re-Distributor (based on the affinity), and perform appropriate actions. Once configured, the Re-Distributor interface can be used in a similar manner to the global Distributor.

Inter Processor Interrupts (IPI) can also be delivered based on the CPU Affinity Hierarchy. Because the FreeBSD kernel does not enumerate CPUs in SMP according to their hardware affinity, it is required to save and match each CPU address with the requested CPU group on every IPI. Software Generated Interrupts (triggered by the write to the CPU Interface register) are used to perform IPI exchange.

ITS and Message-Signaled Interrupts

In a typical scenario the peripheral device requests an interrupt in the Distributor that then forwards it to the appropriate Re-Distributor. Finally, the interrupt is signaled to the CPU Interface. The alternative behavior, for which Re-Distributors are used in GICv3, is interrupt routing that bypasses the Distributor. This is used by MSIs and requires Interrupt Translation Service (ITS) assist. Both ways are depicted in Figure 1.

ITS is a GICv3 extension that manages routing and migration of LPIs generated by any device that can send Message-Signaled Interrupts. The unique approach presented by the ITS controller is that all MSI-capable devices can use a single memory location (GITS_TRANSLATER register address) to generate an interrupt. The interrupt request number and appropriate routing is deduced based on the interrupting device's identifier and translation information programmed into the ITS. The interrupt controller works closely with the PCIe bus and IOMMU because unique device IDs used in the interrupt translation process are passed within the bus transaction itself. The programming of the ITS is performed via commands sent to the special Command Queue and is required to set up the relation between the PCI endpoint, LPI numbers (a.k.a.

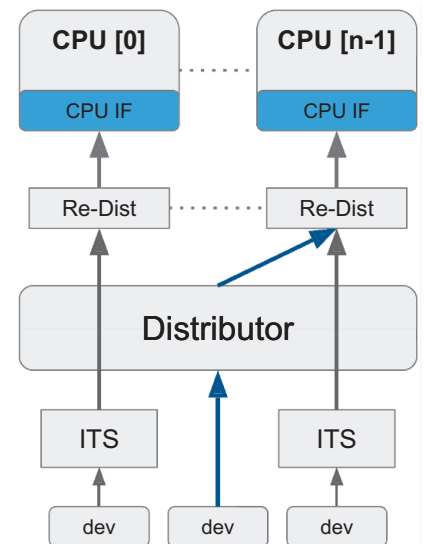


Fig. 1: Interrupts distribution using GICv3 and ITS.

interrupts collection), and the target Re-Distributor.

The FreeBSD support for ITS consists of the GICv3 subordinate driver (`gic_v3_its.c`) and device methods for allocating and mapping MSIs (`pic_alloc_{msi, msix}()`, `pic_map_msi()`). The controller requires some portion of the system memory to operate and this needs to be provided by the OS. The variety of possible ITS implementations implies the existence of auto-detection features that need to be revised when configuring the device. This includes not only the amount of RAM to reserve for the controller, but also various cacheability/shareability attributes, as well as page sizes used by the ITS memory system.

The absolute basic configuration requires:

- ◆ **Memory for ITS Private Tables**
Private controller's tables that are not accessible to the programmer.
- ◆ **Memory for the Command Queue**
A ring buffer for the ITS commands. Apart from the memory reservation, OS software must set the queue write pointer (`GITS_CWRITER`) to the start of the queue.
- ◆ **Memory for the LPI Configuration and Pending Tables**
The LPIs pending status is visible through the bitmap in the Pending Tables. A particular LPI can be configured (i.e., masked/unmasked) using an array of bytes in the Configuration Table.

Interrupt mapping is created per device and per Re-Distributor (so in fact per CPU). The implementation-defined device identifiers are usually based on the PCIe `Bus:Device:Function` address. For ThunderX, this ID is more complicated due to possible multi-socket configuration and requires the additional `Node:ECAM.number` address. Device IDs differ for the internal and external PCIe units and this needs to be taken into consideration by the ITS code. CPUs on the other hand are matched based on their CPU affinity or Re-Distributor physical address.

In order to create an LPI interrupt route the software has to:

- ◆ **Map interrupts collection to a Re-Distributor (`MAPC` command)** Managing interrupts via collections and not stand-alone entities is useful when migrating LPIs from one CPU to another. This, however, does not occur very often on FreeBSD, but still is required as a part of interrupts routing. An ITS driver assigns collection IDs based on the

destination CPU IDs, so the subsequent interrupt-to-collection mapping can be easily associated with the interrupt-to-CPU assignment.

- ◆ **Allocate and assign an Interrupt Translation Table (ITT) to a device (`MAPD` command)** The software has to provision memory for an array of Translation Table Entries; each mapping interrupts for a given device. ITT will be associated with the device ID after issuing `MAPD` command to the ITS.
- ◆ **Reserve a range of LPI numbers** Each MSI vector is signaled to the CPU as an LPI. The pool of available LPI *physical* numbers starts with 8192 and is configurable through an array of bytes in memory. Usually MSI-capable endpoints will request a range of vectors rather than a single interrupt. For that reason, the driver needs to find a free range of LPIs and allocate it for a particular device. LPI bookkeeping is achieved by a bitmap in which each bit represents an interrupt number. This maps directly to the pool of free LPIs and portions of the bitmap are assigned for each MSI-capable device.
- ◆ **Map MSI vector to the LPI, device ID, and a collection (`MAPVI` command)** The benefit of the interrupt translation is that any virtual interrupt number that is being sent by the device can be mapped to a given *physical* interrupt vector (LPI). Therefore, each endpoint can simply use any convenient combination of messages. The `MAPVI` command inserts an interrupt translation and route to the appropriate CPU into the ITT of the selected device. This final step provides a full set of information that allows for MSI delivery.

The ITS driver's PIC methods provide the implementation of the described steps.

`PIC_ALLOC_MSI/MSIX` allocates the abstract *ITS device*, which is in fact an Interrupt Translation Table with a set of pre-allocated LPI numbers. The main difference between MSI and MSIX allocation is that MSIs are requested as a range of vectors, whereas MSIX vectors are requested one at a time. Common `PIC_MAP_MSI` callback does exactly that: it maps an MSI vector to the LPI for a given device. Finally, the LPI needs to be unmasked in a Configuration Table; however, this has to be initiated from the top level PIC, which is, in that case, the GICv3 driver.

Although very extensive, this design provides a

CONTINUES NEXT COLUMN

flexible way of managing Message-Signaled Interrupts and simplifies the interrupt resources' assignment for each device by providing a single MSI/MSIX triggering address, arbitrary selection of MSI data, hardware translation to a *physical* interrupt identifier, and a large number of supported devices and interrupt vectors.

SYMMETRIC MULTIPROCESSING

The standard ARMv7-MP specification limits the number of supported CPU cores to eight. Each four cores are logically connected to create a cluster. Then, up to two clusters can be combined using the CoreLink interface that provides a fully coherent 8-CPU core system. Some vendors' implementations of ARMv7 cores can provide up to 16-cluster scalability, which seemed to be the theoretical limit for the architecture. ARMv8 is a huge step forward. The interconnects now are able to address each CPU by its logical location using four-level CPU *Affinity* Address (A 3:A 2:A 1:A 0) that allows a significant growth in total core number.

SMP Bring-up

ThunderX CPU cores are managed through a standard ARM Power State Coordination Interface (PSCI). The relevant code was already available in FreeBSD sources and was used as is. The main work done to support SMP operation on ThunderX was focused on resolving problems in areas such as:

- ◆ System and TLB cache management
- ◆ IPI and interrupts handling
- ◆ Context switching
- ◆ Memory ordering
- ◆ Operations atomicity

For example, the system maintains cache coherence between CPU cores, but only within their shareability domain. If the common memory mappings are not marked as shareable in that domain, data copies seen by the CPUs may differ. Similar results can be observed when one CPU modifies shared Translation Table entries and appropriate TLB maintenance operation is not issued and propagated to the secondary cores. In that case, CPUs can potentially see different physical frames with different access permissions at the same virtual addresses.

CCPI and Dual Socket Operation

ThunderX chips provide even more sophisticated scalability. Based on the Cavium Coherent

Processor Interconnect (CCPI), two processors can be connected creating a shared memory space. The typical two-socket configuration supports 96 cores and up to 1 TB of system memory. From the operating-system perspective, the complete machine looks like it has two separate NUMA (Non-Uniform Memory Access) nodes, each made of 48 CPUs and half of the memory. The I/O interfaces (e.g., PCIe, SATA) are accessible by both nodes, but it is strongly suggested that all I/O accesses be done by the socket owning the corresponding interface. In that scenario, the entire system performance and peripherals are doubled. The dual-socket machine offers twice as many PCIe links, Ethernet interfaces, etc., and the interrupts are distributed among all CPUs using two separate ITS units. For even bigger workloads, the two-socket Cavium systems can be connected using a low-latency Ethernet fabric. This allows for hundreds of gigabits per second of aggregated network bandwidth.

PCIe

The Cavium ThunderX machine provides a standardized interface to which all peripherals are attached. The only I/O the CPU provides is a modern PCIe 3.0 bus. All other devices (SATA, Ethernet NICs, etc.) are typical PCIe endpoints that can be easily detected by the operating system and do not require any machine-specific resource management code except for a single PCIe controller driver. ThunderX provides two distinct PCIe interface types: internal and external. The internal one is a reduced version of the generic PCIe standard providing PCI-like logical access to all peripherals inside the ThunderX SoC. The external one, on the other hand, is a fully compatible PCIe 3.0 link allowing easy connection of generic PCIe cards of up to x8 lanes.

The ThunderX driver is divided into three parts: generic PCIe hardware accessors, FDT-configured internal PCIe controller, and, finally, an internal PCIe device representing an external PCIe controller. The FreeBSD PCI subsystem takes care of almost every aspect of PCI operation except for some very hardware-dependent, low-level functionalities. In order to fulfill these requirements, the driver needs to provide three things: access to a device's configuration space, resource (bus addresses) allocations, and interrupt mapping. On the CN88XX platform, any access to the internal configuration space is done using a generic mechanism called ECAM (i.e., all configuration headers of devices are mapped into the host memory space; each memory access to that location causes the controller to automatically generate all PCIe requests for the user). External

PCIe configuration space is accessed using indirect addressing supported by an external controller. Second, the functionality that the driver needs to provide is a resource assignment. Fortunately, the Cavium UEFI configures all PCIe trees and fills in every BAR with appropriate values. The only thing the driver needs to do is read those (bus) addresses, mark them as used in the Resource Manager (*rman(9)*), and return a result. If a driver is not initialized (this happens for example for NIC's Virtual Functions), the controller manually allocates the necessary bus space and properly configures the BAR. The last thing the driver provides is interrupt mapping. Currently, the only supported interrupt types are MSI or MSI-X, which require some quirks in ITS as well.

The advanced architecture of ThunderX offers a huge amount of internal PCIe devices (more than 200 endpoints). To avoid creation of enormous PCIe device trees, these devices are separated into three different zones, each of which is governed by a separate internal PCIe controller. It is also worth noting that the external PCIe controller is the PCIe device attached to the internal PCIe bus. Although not intuitive, this solution provides an easy way to support various hardware versions of the device. Let's imagine one ThunderX chip has only one external PCIe available, whereas another might have three. Using a conventional approach, each version of the hardware would require a different machine description (i.e., DTB) file to make all the controllers get detected and configured properly. But when the external controller is an internal PCIe device, a number of them are gathered on-the-fly using a standard PCIe enumeration technique.

VNIC

The CN88XX chip has powerful Ethernet capabilities that include 40 Gbps, 20/10 Gbps, and 1 Gbps interfaces. ThunderX introduces a flexible and highly programmable design of the network subsystem that allows for efficient hardware resource virtualization and node-to-node connectivity without using external switches. The main objective of the presented work was to provide a basic networking support for all types of available interfaces.

The networking subsystem in ThunderX is partitioned into a few, core components

- ◆ BGX - Common Ethernet Interface
- ◆ NIC - Network Interface Controller
- ◆ TNS - Traffic Network Switch

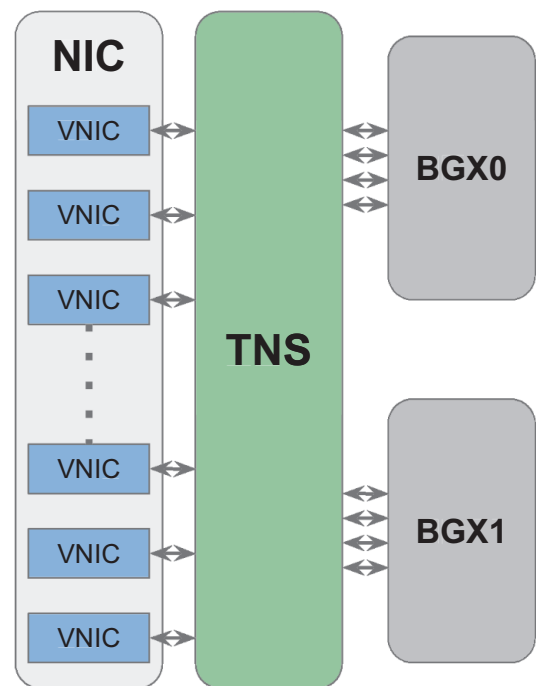


Fig. 2: Network subsystem architecture.

that, in the presented order, implement: the MAC layer, the Network Interface layer, and hardware switching between the mentioned components and other CN88XX devices. The high-level view of network subsystem architecture is depicted in Figure 2. Moreover, NIC is a SR-IOV [4] capable device, that can incorporate up to 128 Virtual Functions, each providing full network interface features. ThunderX contains 2 BGX instances that are connected to the NIC and its VFs via a TNS unit. However, the described FreeBSD implementation of the ThunderX networking drivers does not include support for the TNS and its features. TNS Bypass logic is used instead, which results in the direct connection of BGX units to the corresponding NIC TNS interfaces as well as freedom of Rx/Tx queues assignment to Logical MACs provided by the BGX.

The contributed work is located in the `sys/dev/vnic/` directory in the sources tree and consists of the group of drivers, each implementing the BGX, NIC, VNIC, and MDIO interface. The MDIO setup and partially the BGX configuration are based on the *FDT* description, currently the only method available.

BGX

The Programmable MAC layer is implemented in the BGX controller. It is seen as a regular PCIe endpoint and can be configured without the explicit machine description provided; however, BGX-to-Ethernet PHYs assignment needs to be

presented to the driver. Each of the two built-in BGX controllers can provide up to 4 Logical MACs (LMACs) with maximum rate of 10 Gbps each, or a single 40 Gbps LMAC. Any LMAC can be connected to the arbitrary NIC's Virtual Function. LMAC types are selected by the low-level firmware and FreeBSD driver retrieves this information to proceed with the appropriate setup.

The core BGX code, located in the `thunder_bgx.c` file, is responsible for the general interface bring-up, such as SerDes configuration, detection of the LMAC type, and final Ethernet interface configuration. Since BGX can be attached to a variety of Ethernet PHYs, different media connection types need to be supported. High-speed interfaces, such as XLAUI, XAU, DXAU, or XFI, are managed from the BGX driver; however, low-speed SGMII uses a dedicated MDIO driver located in the `thunder_mdio.c` file. The latter exports `KOBJ(9)` methods for PHY connection management (`LMAC_PHY_{CONNECT, DISCONNECT}`) as well as link state polling (`LMAC_MEDIA_STATUS`).

During a normal BGX driver operation, software polls the MAC layer status to keep the NIC Physical Function up to date. The polling itself is done from the `callout(9)` context and is executed periodically (once per two system ticks). BGX/LMAC re-configuration is performed upon link status change, such as speed transition, etc. Current link status is stored in the driver's software context for each LMAC and is exported to the PF driver on demand.

Physical Function

The Physical Function driver, located in `nic_main.c`, cooperates with BGXes and TNS to create a highly programmable network interface. Unlike other popular NIC cards, the CN88XX Physical Function does not provide networking capabilities, but rather is a resource manager for subordinate Virtual Functions (VFs) and an interface between the MAC layer (BGX) and the networking interface layer (VNIC). PF supports up to 128 Virtual Functions using PCI SR-IOV [4] technology. The communication between PF and VFs is held using private *Mailboxes* for each VF. Any changes in the MAC layer or configuration requests are signaled to the VFs using *Mailbox* interrupt. The Physical Function receives requests from the VFs in a similar manner.

The introduced FreeBSD driver uses a generic PCI IOV subsystem to create and configure Virtual Functions. The key step of the VF bring-

up is the `pci_iov_attach()` invocation. This is done in PF's `nic_sriov_init()` function, called during the device attach procedure. At this point, PF's and VF's so-called configuration schemes (high-level interface options, such as MAC address selection capabilities) are specified and passed to the PCI IOV subsystem. The code also needs to supply the PCI layer with the `IOV KOBJ(9)` methods, such as:

- ◆ `PCI_IOV_INIT(9)`
Implemented by `nicpf_iov_init()`, validates the number of requested VFs and saves this value for later use when the actual bring-up occurs.
- ◆ `PCI_IOV_UNINIT(9)`
Is supposed to disable previously enabled VFs.
- ◆ `PCI_IOV_ADD_VF(9)`
Is called when SR-IOV infrastructure is initializing a new VF. Options requested by the `pci_iov_attach()`, can be used to set up the VF's parameters based on the configuration schema.

During a normal driver's operation, the PF's code periodically polls LMACs' links through the BGX interface and handles VF↔PF requests, which may relate to the VF's Queues configuration, link/MAC status changing, etc.

Virtual Function

Virtual Functions implement the networking capabilities of VNIC. VFs are able to perform DMA transactions to and from the main memory in order to transfer packet traffic. The presented VNIC driver consists of the two logically separated parts: `nicvf_main.c` and `nicvf_queues.c`. The first one performs the typical FreeBSD's network interface configuration and provides `ifnet(9)` callbacks, such as `if_init`, `if_ioctl`, etc. As the controller can handle more than one transmitting queue, the driver implements a multi-queue variant of the Tx path, which uses the `nicvf_if_transmit()` function for the outgoing traffic. The Ethernet media status is updated through the messages from the PF, so a stub `nicvf_media_status()` function just exports this information to the upper layers. The driver also supports hardware and interface statistics that are refreshed periodically in the `nicvf_tick_stats()` callout.

The second part of the driver, located in the `nicvf_queues.c`, is responsible for the controller's resource allocation as well as packets transmission and reception. In particular `nicvf_config_data_transfer()` is the

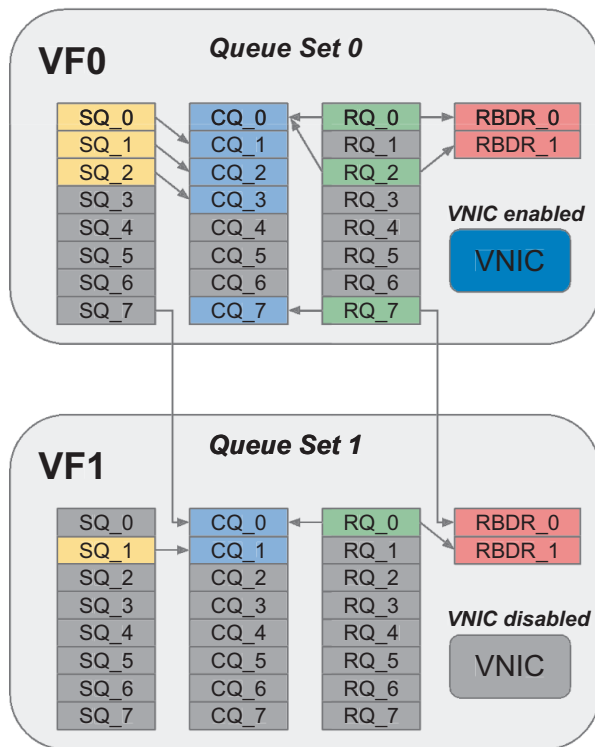


Fig. 3: Exemplary NIC's QS configuration

focal point of the VF's queues configuration.

Each Virtual Function contains a Queue Set (QS) that consists of:

- ◆ 8 x Completion Queue (CQ)
- ◆ 8 x Send Queue (SQ)
- ◆ 8 x Receive Queue (RQ)
- ◆ 2 x Receive Buffer Ring (RBDR)

The Completion Queue contains data describing all completed actions, such as packet transmission or reception. Other queues are assigned by the Physical Function to the selected CQs, potentially many-to-one.

The transmitter side uses SQ to describe the egress data and actions that need to be performed on the packet before it can be sent (e.g., L3, L4 checksums calculation). Each SQ is subscribed to a target Completion Queue.

Receive Queues are VNIC's internal structures that describe how to receive packets. They need CQ and RBDR assignments. More than one RQ can be attached to both CQ and RBDR. RBDRs on the other hand, describe free buffers in the main memory that can be used to store received data. Upon packet reception, VNIC moves the incoming data to the free buffer provided by the RBDR descriptor and returns its physical addresses to the assigned Completion Queue. This could possibly be a drawback, as it is much more difficult to locate a received buffer in the memory using

physical addressing rather than virtual. However, this was resolved by allocating memory exclusively from the *Direct Map*, which is a continuous region of linearly mapped memory. Thanks to that, it is fairly simple to find the received buffer in the virtual address space. Moreover, parts of the ingress buffers, used by the RBDR queues, are adapted to store a small descriptor containing *mbuf(9)* information as well as *bus_dma(9)* related data. This approach can significantly simplify *mbufs* bookkeeping and reduce *mbuf* access latency on Rx.

An interesting capability of the NIC's Queue Sets is that if the VNIC on a particular VF is not enabled, its queues can still be used by another operational VNIC. This makes it possible to assign one Queue Set to a single VNIC, all Queue Sets to a single VNIC, or anything in between. An exemplary, yet possible QS configuration is depicted in Figure 3. Currently, the VF driver utilizes single Queue Set per VNIC by default.

AVAILABILITY

Support has been integrated into the mainline FreeBSD 11.0-CURRENT and is publicly available starting with SVN revision no. 289550. As of the writing of this article, work on improving FreeBSD on ThunderX is still in progress.

Future Development

Even though FreeBSD support for ThunderX is in decent shape, there are a number of areas that could benefit from further development. There are also other functional issues that need to be resolved before dual-socket configuration can be fully supported in mainline FreeBSD. Some more interesting areas of development are briefly described below:

◆ Dual-socket Support in Mainline FreeBSD

To improve kernel performance, all physical addresses from `DMAP_MIN_PHYSADDR` to `DMAP_MAX_PHYSADDR` are mapped statically into a continuous VA region called DMAP (Direct Map). When the PA space is fragmented (as it is for ThunderX in dual-socket configuration), the VA DMAP range is huge and exceeds the limit that the CPU core is able to address in 3-level Translation Table mode. Semihalf implemented a patch that resolves those issues by creating multiple regions of DMAP range to save space and allow for dual-socket device operation. However, the lookups in DMAP arrays are likely to reduce the performance, and hence the 4-level Translation Table solution is the preferred one. However, the current 4-level Page Tables implementation does not allow for successful dual-socket boot. This is

CONTINUES NEXT PAGE

an area that needs to be reworked in order to support two ThunderX nodes in the mainline.

Multiple ITS Support

Currently, only the ITS attached to socket-0 is operational, but in the case of a dual socket system, this resource is doubled. FreeBSD/arm64 lacks support for multiple and chained interrupt controllers; therefore, changes in the interrupts handling code for ARM64 architecture need to be applied.

ACKNOWLEDGMENTS

Special thanks to the following people:


- Dominik Ermel, Michał Mazur, Tomasz Nowicki, and Michał Stanek (Semihalf) for their work and commitment to this project.
- Ed Maste (The FreeBSD Foundation), Andrew Wafaa (ARM), and Larry Wikelius (Cavium) for their successful cooperation.
- Andrew Turner (FreeBSD Project) for insightful reviews and advice.
- Rafał Jaworowski (Semihalf) for organizing and managing this project.
- The success of this project is a joint work of the Semihalf team, Andrew Turner, ARM, Cavium, and The FreeBSD Foundation. The project was sponsored by ARM, Cavium, The FreeBSD Foundation, and Semihalf. ●

Zbigniew Bodek, a Software Engineer at Semihalf, focuses on BSD and Linux operating systems development for ARM and PowerPC-based computers. His main areas of interest are computer science, microprocessor technology, embedded systems and kernel development. He has been a FreeBSD committer since 2013. For the last 1.5 years Zbigniew has been involved in FreeBSD development and optimizations for multi-core ARMv8 chips.

Wojciech Macek is a Software Engineer at Semihalf, a small company in a frosty part of the world. He is primarily interested in ARM architecture, kernel internals and ultra-fast storage solutions. His recent work has focused on enhancing and optimizing FreeBSD for multi-core ARMv8 systems.


REFERENCES



- [1] Cavium, *Cavium ThunderX CN88XX Hardware Reference Manual*. (2015)
- [2] ARM Ltd. Processor Division, *GIC Architecture Specification PRD03-GENC- 010745 12.0*. (2013)
- [3] ARM Ltd., *ARM Architecture Reference Manual for ARMv8-A architecture profile*. (2013–2015)
- [4] Intel, *PCI-SIG SR-IOV Primer, 321211-002*. (2011)




Rack-mount networking server


Designed for BSD and Linux Systems
Up to **5.5Gbit/s** routing power!

Made for  FreeBSD





 PERFECT FOR


- ▶ BGP & OSPF routing
- ▶ Firewall & UTM Security Appliances
- ▶ Intrusion Detection & WAF
- ▶ CDN & Web Cache / Proxy
- ▶ E-mail Server & SMTP Filtering
- ▶ Anti-DDoS and clean pipe filtering

 KEY FEATURES


- ▶ 6 NICs w/ Intel igb(4) driver w/ bypass
- ▶ Hand-picked server chipsets
- ▶ Netmap Ready (FreeBSD & pfSense)
- ▶ Up to 14 Gigabit expansion ports
- ▶ Up to 4x10GbE SFP+ expansion




DESIGNED FOR
FreeBSD



DESIGNED FOR
GNU / Linux



DESIGNED FOR
FreeBSD
ThunderX Reference



DESIGNED FOR
Sense

Designed. Certified. Supported

1Gbit/s Copper	Ports	Chipset
L800-G808-1	8x Gbe RJ-45 ports	8x Intel i210 AT; PEX8618
L800-G808-2	8x Gbe RJ-45 ports	8x Intel i210 AT; PEX8618
L800-G428-1	4x Gbe RJ-45 ports	1x Intel i350 AM4
L800-G428-2	4x Gbe RJ-45 ports	1x Intel i350 AM4
1Gbit/s SFP (Fiber)	Ports	Chipset
L800-S406-1	4x Gbe SFP ports	i350-AM4
10GbE Copper	Ports	Chipset
L800-T202-1	2x 10GbE RJ-45 ports	Intel X540
L800-T203-1	2x 10GbE RJ-45 ports	Intel X540
10GbE SFP+ (Fiber)	Ports	Chipset
L800-X204-1	2x 10GbE SFP+	Intel 82599ES
L800-X205-1	2x 10GbE SFP+	Intel 82599ES
L800-X405-1	4x 10GbE SFP+	Intel 82599ES; PEX8724

contactus@serveru.us | www.serveru.us | 8001 NW 64th St. Miami, LF 33166 | +1 (305) 421-9956