NSP_GMACSW

Driver Design Document

Document #	Author(s)	Approval(s)
	Daniel Allred	

Template Version 0.1

TABLE OF CONTENTS

1 Introduction

1.1 Purpose & Scope

This document details the design and architecture for the NSP_GMACSW software deliverable. The NSP_GMACSW software is a Network Support Package (NSP) for the GMACSW peripheral, supporting integration with the TI Network Development Kit (NDK) on the TDA2xx SoC (Vayu), the TI814x/VisionMid SoC (Centaurus), and the TI811x SoC (J5Eco). The TI NDK provides a TCP/IP networking stack and various other features to enable networking applications on TI embedded processors. The NSP_GMACSW package provides the bridge between the generic NDK stack and the specific networking hardware of the supported devices.

The package itself is divided into two distinct libraries. The first is the low-level hardware driver itself, which exposes software interfaces to establish networking communication channels and configure other aspects of the GMACSW subsystem. This first library, simply called the GMACSW driver library, is device-dependent and OS-independent. The second library is the adaptation layer that bridges the gap between the TI NDK and the low-level driver. It provides Network Interface Management Unit (NIMU) APIs to satisfy the NDK requirement and converts between the NDK data structures and driver data structures on transmit and receive. This second library, called the NDK2NSP library, is device-independent and OS-dependent, and is specifically written to support the NDK stack and the SYS/BIOS RTOS from Texas Instruments.

The library separation is important as it would allow a user to put the low-level driver library into service with a different networking stack and/or OS (or possibly in a no-OS environment).

1.2 Terms & Abbreviations

CPDMA	Communication Port Direct Memory Access
CPPI	Communications Port Programming Interface
CPTS	Common Platform Time Stamp
Ethernet	Collection of IEEE standards defining a family of computer network technologies (formalized in IEEE 802.3)
MAC	Media Access Control – logical control layer for networking
NDK	TI Network Development Kit
NIMU	Network Interface Management Unit
NSP	Network Support Package
PHY	Physical layer device – physical layer for networking

1.3 References

1.	TDA2x TRM, Section 22.8	Device Technical Reference Manual

2 Software Module Overview

This section provides detailed information about each software module of the driver. It also notes the interdependencies between the modules. This section is divided into two parts, corresponding to the library separation described in the introduction. Figure 1 can be used as a reference when reading this section.

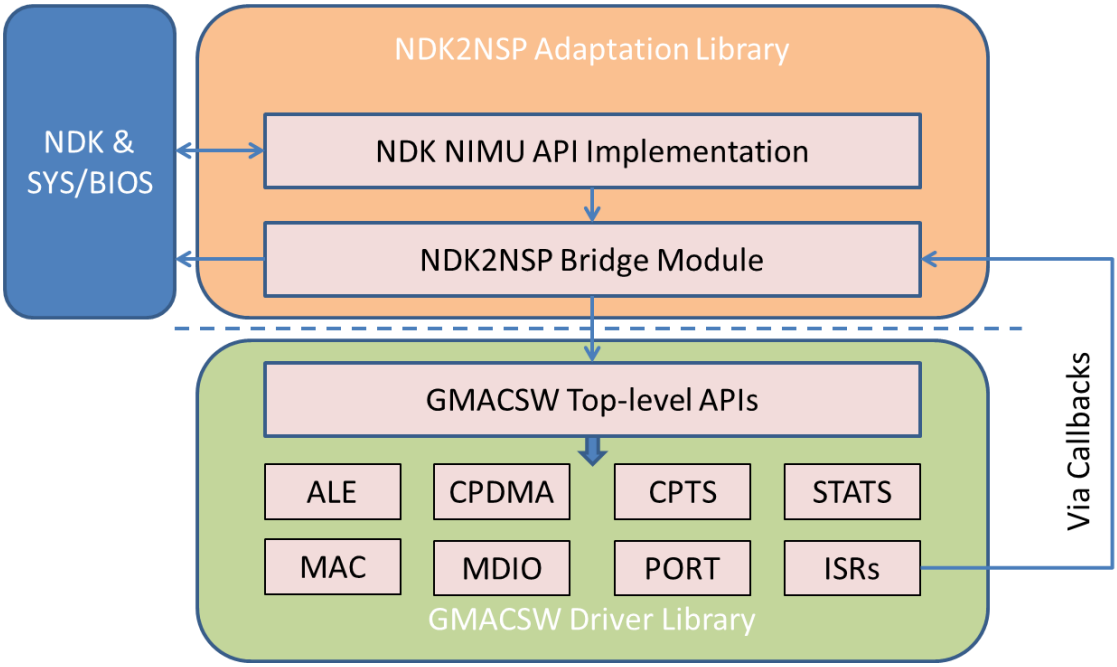

Figure 1. Graphical Overview of NSP_GMACSW Architecture

2.1 Low-level GMACW Driver Library

The low-level driver library is device-dependent, as each device has potentially different memory maps for the GMACSW register set and different supported features. The modules of this library are written in pure C, utilizing standard C header files and features, with C99 standard data types. There are no direct dependencies on operating system features or other frameworks (like XDC). Any feature which might need to depend on the OS or platform is abstracted through an abstraction layer defined in gmacsw_al.h. The implementation of that abstraction layer must be done in some other scope (the application or another library). For this package it is done in the NDK2NSP library, which is specific to the SYS/BIOS RTOS. See section 2.2.3 for more detail.

For further information about the files described below, the user can examine the source in the packages/ti/nsp/drv path. The public header files are in the packages/ti/nsp/drv/inc folder and the source files can be found in the packages/ti/nsp/drv/gmacsw folder. There is also doxygen-generated documentation in HTML format located at docs/gmacsw/html/index.html (it is also linked from the release notes). All source files are released as open source under a 3-clause BSD license.

2.1.1 gmacsw.c & gmacsw.h

This module implements the top-level driver interface, defining methods for opening and closing the driver, as well as interacting with it via IOCTL calls. It calls into all the other modules of the driver to open those modules as part of its own open function.

The first time the open function is called, a pointer to a valid GMACSW_Config structure must be provided as an input parameter. Subsequent calls will ignore the config parameter and will simply return a handle to the already open driver. The returned handles are reference counted in the driver and the driver close function will not perform the close operation until the reference count returns back to zero.

The GMACSW_ioctl() API is intended to be used as the entry point to the driver for all driver operations. The sub modules of the driver should not generally be called directly. The top-level IOCTL API will delegate to the sub modules as appropriate

2.1.2 gmacsw_config.c & gmacsw_config.h

The gmacsw_config.h file defines the GMACSW_Config structure needed for initial opening of the driver. The gmacsw_config.c file is not a critical component of the driver but rather defines memory for a single GMACSW_Config structure and provides routines for populating and returning that structure with appropriate default values.

These functions are not required to be used, as the user who opens the driver could create and populate a configuration structure on their own, but they do provide the default configuration used for testing and development, so their use is recommended.

2.1.3 ale.c & ale.h

These files provide the interface to manipulate the address lookup engine (ALE) of the GMACSW peripheral. The ALE is central to the switch operation (the SW part of GMACSW). The default driver configuration used for testing puts the ALE in bypass mode, so this module will not be discussed further. In this configuration, the driver configures the peripheral to operate in MAC-only mode, where data arriving at the MAC ports is forwarded exclusively to the internal host port for processing by the driver. The switching capability of the GMACSW is essentially not used with this software driver.

2.1.4 cpdma.c & cpdma.h

These files define and provide an interface to the internal host port of the GMACSW peripheral. This host port, called the CPDMA, is a multichannel DMA engine that implements the Communications Port Programming Interface (CPPI) v3.1. This module is at the heart of the movement of data from the external network into the host CPU and the networking stack that is running there. The CPDMA port makes up one of the three ports of the GMACSW three-port switch. The structures, definitions, and APIs provided in this module are laid out in more detail in section 3.2.

2.1.5 cpts.c & cpts.h

This module is used for interacting with the CPTS block of the GMACSW peripheral. The CPTS block is used for attaching timestamps to arriving packets that meet certain criteria, such as adhering to the IEEE 1588v2 Precision Time Protocol (PTP). This should be used as the basis for implementing a PTP software stack. The hardware timestamping provides a very accurate and high-resolution time basis that can be maintained and updated as part of a PTP implementation

2.1.6 interrupts.c

This module centralizes all the interrupt service routines defined by the driver. The GMACSW peripheral defines four interrupts: receive pending, transmit pending, receive threshold pending, and miscellaneous. The description of the ISRs for each of these interrupts is given in section 3.2.3.

2.1.7 mac.c & mac.h

This module provides an interface to the external MAC ports of the GMACSW peripheral. The external MAC ports, which can support a few different media independent interface (MII) standards (such as RGMII, RMII – refer to the device reference manual for details on which interfaces are supported), are the I/O blocks through which network data moves in and out of the device. Typically the MAC ports are connected to a PHY device to receive and transmit the data bits over the physical wires. There are two MAC ports, which make up two of the three ports of the GMACSW three-port switch.

2.1.8 mdio.c & mdio.h

This module implements a driver to send and receive data via the MDIO (management data input/output) interface. This interface is the standard way to communicate with and configure Ethernet PHY devices. The MDIO interface, which is used for configuration and management, is distinct from the MAC interface (in MII, RMII, or RGMII modes), which is used for data movement.

2.1.9 port.c & port.h

The port module touches upon aspects of the switch and the switch ports (the host CPDMA port and the MAC ports). The GMACSW hardware separates the port registers into their own distinct blocks, apart from either the CPDMA or MAC modules, and this software organization reflects that separation (though an argument could be made to have this code be merged into the cpdma.c and mac.c modules). This module provides capabilities and features related to operating modes, prioritization and priority mapping, switch queue management, etc. This module provides IOCTLs to add and remove Rx priority mappings and to add and remove Tx rate limits.

2.1.10 stats.c & stats.h

The stats module is simply a front-end for the hardware statistics registers included in the GMACSW. These registers record octet counts, packet counts by type, error counts, etc. The module maintains a cumulative software mirror of the hardware registers which are updated when the statistics are queried by a user or when a statistics interrupt is generated (which is a type of the miscellaneous interrupt). The module provides IOCTLs to reset/clear the statistics and to read the current statistics.

2.1.11 gmacsw_al.h

This file defines a simple abstraction layer that the driver relies on for certain activities, but which are not defined in the low-level driver in order to preserve the independence from the OS and networking stack. The functions defined in it should be implemented in a separate library or in the application. In this package, the implementation is provided in the NDK adaptation layer.

2.2 NDK2NSP Adaptation Library

This adaptation layer library is device-independent, but it is OS- and stack-dependent. It serves as a bridge between the TI NDK TCP/IP stack and the GMACSW low-level driver described in section 2.1. This library is included in the SYS/BIOS application build if the NDK is specified to be used (which is done via the SYS/BIOS configuration file), but it will not be included if the NDK is not used. This is an important point, since this layer is the one that calls the low-level driver's GMACSW_open() function. If the NDK is not used, some other entity is responsible for the initial opening of the driver.

For further information about the files described below, the user can examine the source in the packages/ti/nsp/drv/ndk2nsp path.

2.2.1 nimu_ndk.c & nimu_ndk.h

This module provides a definition of the NIMUDeviceTable array. This array contains a NULL-terminated list of initialization routines which will be called as part of the NDK initialization. The array defined here has one element, indicating that a single interface is defined for this platform. The initialization routine in the list is defined in this module as well.

When called, the initialization routine populates a NETIF_DEVICE structure and then calls the NIMURegister API of the NDK in order to register the interface with the NDK. The NETIF_DEVICE structure contains a table of function pointers that all NIMU compliant interfaces must implement. These are provided in this module to satisfy the NIMU requirements of the NDK. The function implementations call into the functions provided as part of the ndk2nsp module.

2.2.2 ndk2nsp.c & ndk2nsp.h

This module is the true glue that bridges the NDK software to the GMACSW low-level driver. When the NIMU start API, defined in the nimu_ndk module, is called, it calls the NDK2NSP_open() API of this module. It attempts to retrieve an application-provided driver configuration by calling an extern function, GMACSW_getConfig(). This function should return a proper configuration structure, which should include the MAC addresses for the MAC port(s). How the MAC addresses are determined could be device or application specific. They included examples should be consulted for more details.

With a proper configuration structure in place, this module then calls the driver's GMACSW_open() API. This will initialize the driver and reset the GMACSW peripheral and all of the sub modules which comprise it. This module then creates and configures receive and transmit channels in the host CPDMA port. These channels are setup to be used for best effort traffic (priority is given as NONE). This module creates OS tasks (threads, one for Rx and one for Tx) to be used for processing packets returned by the driver. Finally the CPDMA channels are started so that data movement can begin.

Details of how the Rx and Tx packet processing tasks are used to move data from the driver and into the stack are given in section 0.

2.2.3 gmacsw_al.c

This module provides an abstraction layer implementation of the APIs defined in the gmacsw_al.h header file. Currently, the only APIs implemented are those to globally disable and enable interrupts, which are primarily used to make the manipulation of queues and linked-lists thread safe.

3 Principle Features and Design

This section describes, in more detail, the features and design of the driver, as well as the reasons for the design decisions that were made.

3.1 Driver Initialization

In an application that makes use of the NDK, the NDK adaptation layer (see section 2.2) is responsible for initializing the driver and opening channels for data movement for the NDK stack. The startup steps include:

1. Opening the driver with an appropriate configuration structure.
2. Opening Rx and Tx channels in the CPDMA module.
3. Setting up the port configuration.
4. Starting the Rx and Tx channels.

After the above steps are complete, the driver is active and data can move in and out of the hardware.

3.1.1 GMACSW_open()

The driver open command begins by resetting the particular sub-modules of the GMACSW peripheral in a specific order.

1. Reset CPDMA host port by setting soft reset bit.
2. Reset MAC external ports (the SL modules) by setting soft reset bit.
3. Reset the 3-port switch subsystem by setting the soft reset bit.
4. Reset the subsystem wrapper (the WR module) by setting the soft reset bit.

The driver then performs open operations on the various modules. The configurations used for the open commands comes from the global configuration parameter given to the top-level GMACSW_open() call. There are many options available to the different modules of the peripheral, so it is important that the configuration structures are correctly filled in. When all of the sub modules have been opened, the driver is ready to receive IOCTL calls to further configure and utilize the driver.

3.1.2 Opening channels in the CPDMA

The CPDMA is the host port of the 3-port gigabit switch at the heart of the GMACSW peripheral. The CPDMA implements the CPPI v3.1 standard. In the CPPI terminology, a channel refers to the sub-division of information (flows) that is transported across the CPDMA host port. There are eight receive (Rx) channels and eight transmit (Tx) channels. Each channel has associated state information, which is accessible in the STATERAM register block of the GMACSW peripheral MMR space. The CPDMA is a DMA engine that, as a bus master, will copy data from memory buffers to the hardware on Tx (and vice versa on Rx).

A stack that wishes to make use of the GMACSW driver to send data must open a transmit channel, which is done by calling the CPDMA_OPEN_TX_CHANNEL IOCTL. A configuration structure, of type CPDMA_TxChannelConfig, must be provided to the IOCTL. The configuration structure includes fields that define how many buffers are used per CPDMA packet, how many CPDMA packets must be available for use by the channel, a pointer to memory in which the CPDMA packet instances will be allocated (as the driver does not do any memory allocation), and callback function pointers that will be used to notify the stack (or adaptation layer) when the channel is done with a packet (because the data in the packet has been delivered to the hardware). Note that a CPDMA packet is not the same as network packet, but rather describes a single data unit for movement via the CPDMA.

A stack that wishes to make use of the GMACSW driver to receive data must open a receive (Rx) channel, which is done by calling the CPDMA_OPEN_RX_CHANNEL IOCTL. The same info provided for the Tx channel configuration must be provided for the Rx channel configuration (of type CPDMA_RxChannelConfig), with the addition of a field to specify the Rx buffer descriptor threshold. This threshold value is used by the receive channel to know when to generate an Rx Threshold interrupt (as opposed to a standard Rx Pulse interrupt). If the threshold value given is zero, Rx threshold

interrupt generation is disabled. The threshold feature should be used in order to prevent overruns when the Rx interrupt pacing feature is enabled.

Opening the channels does not start the channels (i.e. no data movements take place, nor are any interrupts generated). While a stack could open more than one channel, for managing different types of data flows, it is anticipated that a single channel of each type will be used. The NDK adaptation layer opens only a single Rx and Tx channel.

3.1.3 Setting up the Port Configuration

One important thing to do, particularly when multiple channels will be used by multiple stacks, is to configure the ports and the switch to correctly map priorities and other options. This allows implementation of Quality of Service (QoS) features and can help prevent Denial of Service (DoS) attacks. The driver currently supports Rx priority mapping, which allows the user to specify the mapping of certain traffic flows to the switch priorities and finally to CPDMA channels at the host CPDMA port. The priority selection can be based on VLAN tags in Ethernet frames or IPv4 diffserv values in the IP header of a packet. You can also establish a mapping for all other traffic, allowing one to distinguish between best effort traffic and other higher priority traffic. The Rx priority mapping is done using the PORT_ADD_RX_PRIORITY_MAPPING IOCTL. Details on this topic can be found in section 3.3.

A future version of the driver will also support Tx priority mapping and rate limitations based on the CPDMA channel used to send the data.

3.1.4 Starting the CPDMA channels

After all port and channel configuration is complete, the channels must be enabled (started) before data can be received or transmitted. This is done by calling the CPDMA_START_RX_CHANNEL and/or CPDMA_START_TX_CHANNEL ioctls as appropriate.

After the channels are started, packets arriving at the external port will be directed through the switch to the host CPDMA port and the appropriate CPDMA channel, filling in a CPDMA packet and the packet's associated buffers, which will result in the generation of an interrupt. For transmitted packets, the data will be placed into a CPDMA packet of the CPDMA Tx channel and will be transmitted via the switch to the external port and then sent out of the device. When the data in the CPDMA packet has been copied from memory and submitted into the switch's host port Rx FIFO (data entering the switch is considered Rx, even if the host sees this data as Tx destined to leave the device via the external MAC port), an interrupt is generated to tell the host that it can now reclaim the CPDMA packet. Many more details about the movement of data can be found in section 3.2.

3.2 Data Movement

This section goes into further details of the CPDMA host port of the three port gigabit switch peripheral and describes the hardware and software flow for data moving through that host port.

3.2.1 CPDMA Overview

The CPDMA host port is a DMA engine that implements the Communications Port Programming Interface v3.1. The movement of data via the CPDMA requires four primary components:

1. A collection of logical channels, some for transmit and some for receive.

2. State information for each channel.
3. A collection of buffer descriptors in memory that can be linked together and submitted to the hardware.
4. Data buffers in memory, pointed to by the buffer descriptors.

A distinction must be made between hardware constructs and software constructs in order to adequately explain how the driver works.

The CPDMA channels are hardware within the CPDMA engine, with various registers acting as a control interface to the channels in the CPDMA module's register space. The DMA engine of the CPDMA is hardware that can independently move data from memory into the host port of the 3-port switch. The software driver also maintains a CPDMA channel structure for each opened channel, and a handle to this structure is returned from the CPDMA_OPEN_RX_CHANNEL and CPDMA_OPEN_TX_CHANNEL IOCTL. A description of this structure is given in section 3.2.2.

The state information for each channel is held in hardware in a set of memory mapped registers, called the STATERAM. These registers are updated by both hardware and software and contain information about the queue of buffer descriptors (and therefore the associated data buffers) that composes the active channel. There is also state information for each channel maintained in the software CPDMA channel structure.

The buffer descriptors are structures in memory whose format and contents are dictated by the CPDMA hardware, as they are used directly by the hardware DMA engine to determine what data to move and how to move it. These descriptors are setup by the software and include a pointer to the data buffer to be sent (for Tx) or filled up (for Rx). When these descriptors are handed over to the hardware and become active in the data movement, they should not be touched by software in any way. When their use in the DMA data movement is complete, the descriptors (and the associated buffers) are returned to the software via an interrupt. The buffer descriptors can exist in any memory region accessible by the GMACSW peripheral. However, the peripheral also contains a local RAM memory of 8KB for the express purpose of holding buffer descriptors. Each buffer descriptor is 16 bytes, so this local memory can hold 512 of them. The current driver implementation hardcodes the use of this memory region for allocating buffer descriptors. Because the region is local to the peripheral, it should be more efficient for the DMA engine to use.

The data buffers associated with each buffer descriptor are simply regions of memory that have been allocated for holding packet data that is to be moved in or out via the CPDMA. While the CPPI interface allows buffers of up to 64KB in length, the GMACSW peripheral only allows packets up to 2K (and furthermore, the typical Ethernet packet is limited to a max of 1522 bytes on the wire). A software data packet can be distributed over multiple buffers, but each buffer must be associated with a buffer descriptor. As a result, a CPDMA packet software construct is used which can contain multiple buffer descriptors. With the current implementation, the number of buffer descriptors and buffers used per channel is a configuration option at channel open time and is presumed to remain constant for the lifetime of the channel. The data buffers are filled by the stack or other software for Tx, and the buffers are filled by the CPDMA hardware for Rx.

3.2.2 CPDMA Data Structures

This section describes the software data structures in use with the hardware to move data in and out of the CPDMA.

3.2.2.1 CPDMA Descriptor

The CPDMA buffer descriptor software structure is defined as follows:

```
typedef struct CPDMA_BUFF_DESC_
{
    struct CPDMA_BUFF_DESC_ *pNextBufferDescriptor;
    uint8_t *pDataBuffer;
    uint32_t bufferOffsetAndLength;
    uint32_t flagsAndPacketLength;
}
CPDMA_BuffDesc, *CPDMA_BuffDescHandle;
```

The pNextBufferDescriptor field is a pointer to another buffer descriptor, allowing the formation of a linked-list, or queue, of buffer descriptors. This field is filled in by software for both Tx and Rx use cases.

The pDataBuffer is a 32-bit pointer to the data buffer memory. It is filled in by the software for both Tx and Rx use cases. Note that the driver does not make any assumptions about the cacheability of the buffer pointer to, or the current cache status if that buffer is cacheable. The tasks which submit and retrieve packets must appropriately deal with cache concerns so that the view of memory from the CPDMA and host CPU remain consistent. In this NSP_GMACSW package, the NDK2NSP library handles this for packets meant for the NDK.

The bufferOffsetAndLength value contains a 16-bit offset field from the start of the buffer and the length field, or count, of valid data bytes contained in the buffer. These two fields are filled-in by software in the Tx use case, and are filled in by the hardware in the Rx use case. In the case of Rx, the offset value is determined by the value stored in a hardware register.

The flagsAndPacketLength value contains a 16-bit field of the packet length (which should be the sum of the buffer length fields of the buffer descriptors that make up the packet). In the upper 16-bits it has flags reserved for different uses in the Rx and Tx cases. The important flags are the OWNER, SOP, and EOP flags. The OWNER flag is set by the software just before giving the buffer descriptor to the hardware CPDMA for use. The OWNER flag indicates that the buffer descriptor is "owned" by the hardware and should not be manipulated in any way by the software. This flag will be cleared by the hardware when the buffer descriptor is done being used and can be reclaimed by the software. The SOP flag is used to indicate that a buffer descriptor is the start of a packet (the first buffer descriptor in a chain of descriptors that comprise the packet). The EOP flag is used to indicate that a buffer descriptor is the end of a packet (the last buffer descriptor in a chain of descriptors that comprise the packet). Certain fields, such as the packet length, only have meaning for the SOP buffer descriptor. The SOP and EOP flags must be set by software for Tx packets, but will be set by the CPDMA hardware for Rx packets.

This structure, which is 16-bytes in length, is used directly by the hardware. Therefore its organization and fields are defined by the CPPI standard, which the CPDMA implements. However, as mentioned above, the software must manipulate some elements the buffer descriptors before they are given to hardware. This manipulation is not done by the driver, but is rather expected to be done by the software which calls the CPDMA IOCTL to submit packets (see section 3.2.4).

3.2.2.2 CPDMA Descriptor Queue

The CPDMA descriptor queue software structure is defined as follows:

```
typedef struct CPDMA_BUFF_DESC_QUEUE_
{
    uint32_t count;
    CPDMA_BuffDesc *pHead;
    CPDMA_BuffDesc *pTail;
}
CPDMA_BuffDescQueue;
```

This is a software structure that represents a collection of CPDMA buffer descriptors. This structure is used by the CPDMA packet structure to keep track of all descriptors used to create a packet.

3.2.2.3 CPDMA Packet

The CPDMA packet software structure is defined as follows:

```
typedef struct CPDMA_PACKET_
{
    struct CPDMA_PACKET_    *pNextPacket;
    void                    *hPrivate;
    CPDMA_BuffDescQueue     buffDescQueue;
}
CPDMA_Packet;
```

This software structure is basically the software representation of a CPDMA hardware packet. In the hardware, a packet is a collection of one or more buffer descriptors linked together, with a SOP buffer descriptor to start the packet and an EOP buffer descriptor to end the packet.

For every channel that is opened, a collection of CPDMA_Packets are created in memory. The number of packets and the number of buffer descriptors per packet is fixed for the channel based on the configuration options used in the channel open. The number of buffer descriptors per packet is put into the count field of the buffDescQueue member, and that number of descriptors is added to that queue.

The pNextPacket field is a pointer to another CPDMA_Packet and allows the creation a linked-list, or queue, of packets. This field is updated by the software as the packet is moved among the various packet queues associated with a CPDMA channel.

The hPrivate field is a generic pointer that should be filled in by the stack to allow the stack to be able to identify this packet. This is required because the software networking stack is typically responsible for allocating buffers and formatting data, and this field allows an association between a stack-allocated data structures and the CPDMA packets that are given to the hardware CPDMA. That association is critical when a packet is returned by the hardware to be freed or processed by the software networking stack.

3.2.2.4 CPDMA Packet Queue

The CPDMA packet queue software structure is defined as follows:

```
typedef struct CPDMA_PACKET_QUEUE_
{
    uint32_t            count;
    CPDMA_Packet        *pHead;
    CPDMA_Packet        *pTail;
}
CPDMA_PacketQueue;
```

This is a software structure that represents a collection of CPDMA packets. This structure is used by the CPDMA channel structure to keep track of all packets as they move between hardware and software ownership.

The packet queue is an important structure for feeding data to and retrieving data from the low-level driver. The CPDMA_RETRIEVE_PACKETS IOCTL is used to retrieve a queue of packets from the driver and the CPDMA_SUBMIT_PACKETS IOCTL is used to submit a queue of packets to the driver. When data rates are very high, moving groups of packets is much more efficient than moving packets one at a time.

3.2.2.5 CPDMA Channels

The CPDMA channel software structure is a private structure used by the internal implementation of the driver. The user of the driver receives an opaque handle to a channel when it is opened. The software structure backing that opaque handle is as follows:

```
typedef struct CPDMA_CHANNEL_
{
    uint32_t            channelNum;
    uint32_t            channelDir;
    uint32_t            isOpened;
    uint32_t            isStarted;
    uint32_t            packetCount;
    CPDMA_Packet        *packetMem;
    CPDMA_Callbacks     callbacks;
    CPDMA_PacketQueue   inUsePacketQueue;
    CPDMA_PacketQueue   fromHardwarePacketQueue;
    CPDMA_PacketQueue   toHardwarePacketQueue;
}
CPDMA_Channel;
```

The channelNum field contains the hardware number of the channel. The lowest channel number available is used when a channel is opened. If all eight channels are in use, the channel open will fail.

The channelDir field indicates if the channel is an Rx or Tx channel.

The isOpened field is a flag that indicates if the channel is opened already.

The isStarted field is a flag that indicates if the channel is started already.

The packetCount field contains the count of packets associated with this channel. It is determined at channel open time.

The packetMem field is a pointer to a memory region that was provided by the stack/application at channel open time. That memory region contains packetCount CPDMA_Packet structures after the channel open completes.

The callbacks field is a structure of two function callback pointers, the first used to notify the stack/application of used packets ready to be processed and the second used to ensure proper cleanup during the channel shutdown. These function callback pointers are provided by the stack/application during the channel open call.

The inUsePacketQueue is a queue of CPDMA packets that are currently owned by the hardware. That is, the OWNER bit of the SOP buffer descriptor of the packet is set and the packet is part of the hardware queue of linked buffer descriptors that the hardware can use. The software should never touch packets in this queue. Only the ISRs should manipulate this queue.

The fromHardwarePacketQueue is a queue of CPDMA packets that the hardware has finished using and that are waiting to be retrieved from the driver via the CPDMA_RETRIEVE_PACKETS IOCTL.

The toHardwarePacketQueue is a queue of CPDMA packets that the software has submitted to the driver via the CPDMA_SUBMIT_PACKETS IOCTL, and which are waiting to be given to the hardware.

Figure 2 shows an example representation of the various CPDMA structures which were described above. Not all details are shown.

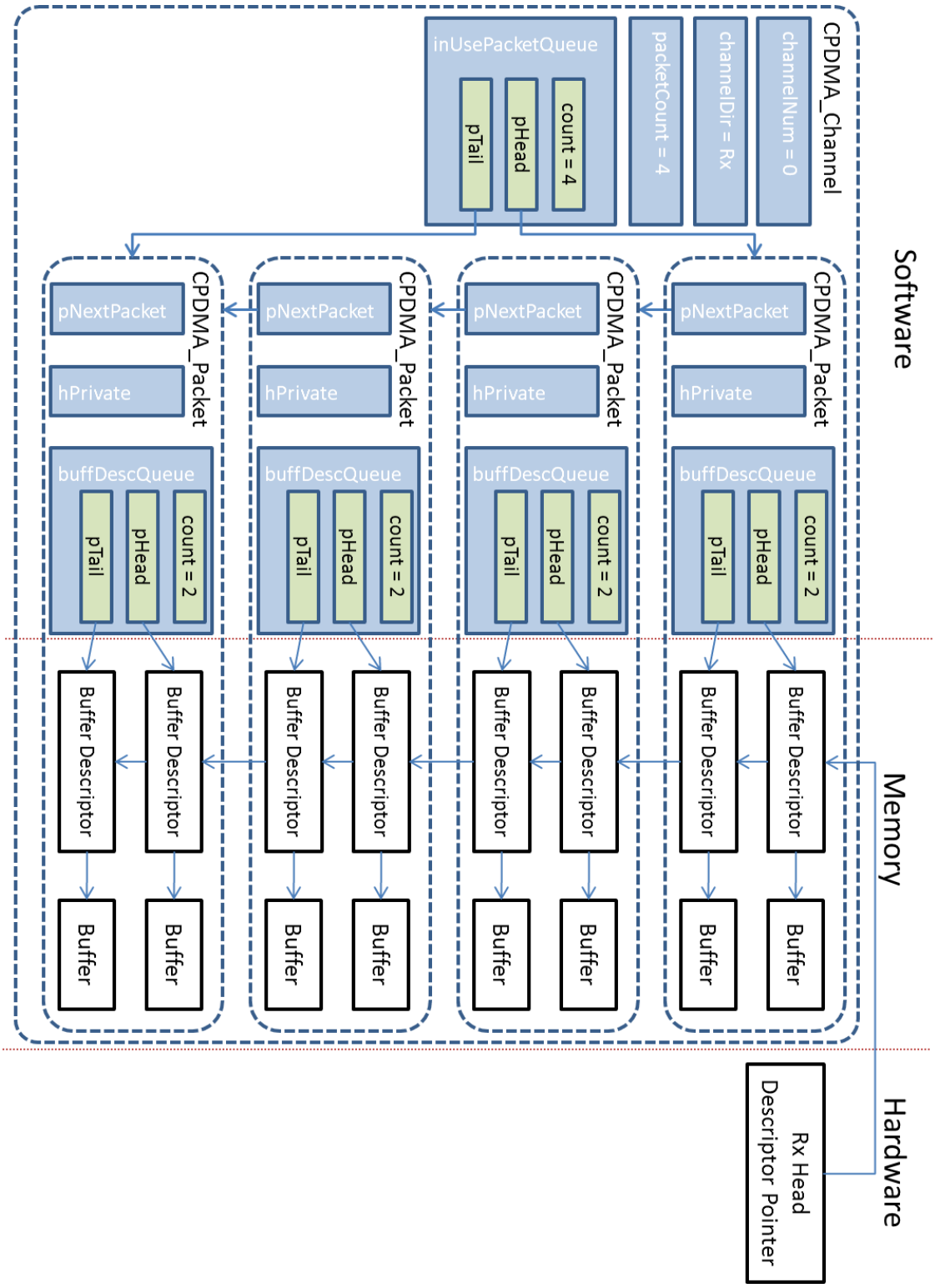

Figure 2. Graphical representation of the various CPDMA structures showing the relationship with hardware and software for an Rx channel with 4 CPDMA packets, each of which contains 2 buffer descriptors.

3.2.3 Interrupt Handling and Interrupt Pacing

The GMACSW peripheral supports four interrupts: receive pending, transmit pending, receive threshold pending, and miscellaneous. In the hardware documentation, these are referred to as the RX_PULSE, TX_PULSE, RX_THRESH_PULSE, and MISC_PULSE interrupts. How these interrupts are mapped to the host core running this driver software, and through the interrupt controller associated with that core, is outside the scope of the driver itself. For examples in the NSP_GMACSW package, all interrupt mappings are done in the example application's SYS/BIOS configuration files. This snippet shows the configuration for the TDA2x (Vayu) device.

```
/* GMAC_SW RX_THRESH_PULSE Interrupt */
IntXbar.connectIRQMeta(57, 334);
var hwi_param_0 = new Hwi.Params;
hwi_param_0.arg = 0;
Hwi.create(57, '&HwIntRxThresh', hwi_param_0);

/* GMAC_SW RX_PULSE Interrupt */
IntXbar.connectIRQMeta(58, 335);
var hwi_param_1 = new Hwi.Params;
hwi_param_1.arg = 0;
Hwi.create(58, '&HwIntRx', hwi_param_1);

/* GMAC_SW TX_PULSE Interrupt */
IntXbar.connectIRQMeta(59, 336);
var hwi_param_2 = new Hwi.Params;
hwi_param_2.arg = 0;
Hwi.create(59, '&HwIntTx', hwi_param_2);

/* GMAC_SW MISC_PULSE Interrupt */
IntXbar.connectIRQMeta(60, 337);
var hwi_param_3 = new Hwi.Params;
hwi_param_3.arg = 0;
Hwi.create(60, '&HwIntMisc', hwi_param_3);
```

Figure 3. Example interrupt configuration for SYS/BIOS & NDK example.

One very important aspect of the interrupt generation of the GMACSW peripheral is the concept of pacing, particular for reception. In a simplistic implementation of networking hardware, one interrupt would be generated per arrived packet. With high bit rate connections, however this can quickly lead the system getting bogged down with interrupts and the associated overhead of context switching. To avoid this, most networking solutions today offer some sort of pacing feature, allowing the hardware to defer interrupts for a period of time until there are more packets ready to process. This feature is implemented in the GMACSW hardware, and is configurable to allow anywhere between 2 interrupts per millisecond (every 500us) and 64 interrupts per millisecond (every ~16us). The default configuration that the driver provides in the gmacsw_config module uses 2 interrupts per millisecond. This has the best performance, but the longest latency.

The pacing feature is important to improve overall performance, but it opens up a potential problem. If the time delay between paced interrupts is too large, there is a real possibility of a DMA overrun condition on an Rx channel, where all Rx buffers owned by the hardware are exhausted and have not yet been replenished due to the interrupt pacing delay. To avoid this problem, the GMACSW peripheral implements

the RX_THRESH_PEND interrupt. It acts as an "emergency" interrupt that will fire, regardless of the time elapsed from the previous RX_PULSE interrupt, when the number of available buffer descriptors in a channel drops beneath a certain, configurable threshold. In other words, if the number of available buffers gets too low, the threshold interrupt will kick in immediately to help ensure that the supply of buffers gets more quickly replenished. There is no threshold interrupt for Tx, since the transmit operation can simply block in the host software until buffer descriptors are made available.

3.2.3.1 RX_PULSE Interrupt

The RX_PULSE interrupt is generated when one or more channels have packets that have arrived and, if pacing is enabled, the pacing interval has expired. The interrupt service routine for handling the receive packet interrupt will also handle any outstanding receive threshold interrupts. It will cycle through all Rx channels and process any that have their interrupt flag set. The steps to handle the channel are as follows:

1. Read channel completion pointer from the CPDMA STATERAM registers. If the channel completion pointer is 0xFFFFFFFC, skip to step 7. This 0xFFFFFFFC value is a magic number that indicates that the channel is in teardown mode, and therefore no processing should take place.
2. Start from the head of the channel's InUseQueue packet queue and cycle through each packet in the queue until finding the SOP buffer descriptor of the packet is still marked as owned by the hardware (OWNER bit set).
3. For each packet not owned by the hardware, check for either of the following conditions:
    a. The current packet is the last packet in the InUseQueue
    b. Or, a packet is encountered whose EOP buffer descriptor is marked with the end-of-queue (EOQ) flag, but whose next pointer is not NULL (misqueue condition)
4. Move all completed packets (and their associated buffer descriptors) out of the InUseQueue and into the channel's FromHardwareQueue.
5. Call the channel's notify callback to let the stack/application know that there are packets to be processed. For optimal performance, the callback should not directly process the FromHardwareQueue and replenish the channel's InUseQueue, as this can be a time-consuming process. Rather, the call back should be used as a mechanism to tell a high priority task in the stack/application/adaptation layer that packets should be processed outside of the ISR context. The ndk2nsp adaptation layer, for example, simply posts a semaphore to another receive processing task.
6. Set the channel's receive completion pointer in the STATERAM to the address of the last buffer descriptor processed (which should be the EOP buffer descriptor of the last packet processed or 0xFFFFFFFC in the case of teardown).
7. If handling the threshold interrupt for the channel, clear the PENDTHRESH value for the channel to zero to ensure that the threshold interrupt won't immediately re-fire as soon as exiting the ISR.

3.2.3.2 TX_PULSE Interrupt

The TX_PULSE interrupt is generated when one or more channels have old packets that have already been used to send data and, if pacing is enabled, the pacing interval has expired. The transmit interrupt will cycle through all Tx channels and process any that have their interrupt flag set. The steps to handle the channel are as follows:

1. Read channel completion pointer from the CPDMA STATERAM registers. If the channel completion pointer is 0xFFFFFFFC, skip to step 7. This 0xFFFFFFFC value is a magic number that indicates that the channel is in teardown mode, and therefore no processing should take place.
2. Start from the head of the channel's InUseQueue packet queue and cycle through each packet in the queue until finding the SOP buffer descriptor of the packet is still marked as owned by the hardware (OWNER bit set).
3. For each packet not owned by the hardware, check for either of the following conditions:
   a. The current packet is the last packet in the InUseQueue
   b. Or, a packet is encountered whose EOP buffer descriptor is marked with the end-of-queue (EOQ) flag, but whose next pointer is not NULL (misqueue condition)
4. Move all completed packets (and their associated buffer descriptors) out of the InUseQueue and into the channel's FromHardwareQueue.
5. Call the channel's notify callback to let the stack/application know that there are packets to be processed. For optimal performance, the callback should not directly process the FromHardwareQueue, as this can be a time-consuming process. Rather, the call back should be used as a mechanism to tell a high priority task in the stack/application/adaptation layer that packets should be processed outside of the ISR context. The ndk2nsp adaptation layer, for example, simply posts a semaphore to another processing task.
6. Set the channel's receive completion pointer in the STATERAM to the address of the last buffer descriptor processed (which should be the EOP buffer descriptor of the last packet processed or 0xFFFFFFFC in the case of teardown).

3.2.3.3 RX_THRESH_PULSE Interrupt

The processing of the receive threshold interrupt is handled the same way as specified in section 3.2.3.1, except that ONLY receive threshold interrupts are dealt with (and not the standard, paced RX_PULSE interrupts). An important step that the interrupt handler for threshold interrupts also does is to reduce the threshold value for a processed channel down to zero. This is necessary to prevent the threshold interrupt from immediately firing again. This would happen since no packets are processed and submitted back to the driver within the ISR context (only notifications should occur). When the receive task does submit new packets to the channel at a later time, the threshold value for that channel is reset to the value specified when the channel was originally opened.

3.2.3.4 MISC_PULSE Interrupt

The miscellaneous interrupt is used as a sort of catch-all for other various events of interest that the GMACSW peripheral can generate. These are intended to be very low-frequency events.

In the current driver implementation, the events associated with the miscellaneous interrupt that are handled are:

1. CPTS events – The arrival of PTP Ethernet packets, for example. More details in section 3.4.1.
2. Statistics events – Occurs before a 32-bit statistics register rolls over, allowing the driver to accumulate the current value into a wider variable in the STATS software module. See more details in section 3.4.2.
3. Host Error – Detection of a catastrophic CPDMA failure which cannot be recovered from.

3.2.4 CPDMA IOCTLs

This section details the two driver IOCTLs that are actually used for moving data in and out of the driver. The intended calling context for these IOCTLs is from within a task/thread, and not within an interrupt context. Every effort was made to make the driver IOCTLs be thread-safe, but testing for this was limited.

3.2.4.1 Submit Packets

The CPDMA_SUBMIT_PACKETS IOCTL is used to move a queue of CPDMA_Packets from the software and into the hardware CPDMA channel. For Rx channels, the packets that are submitted will be filled in with arriving data via the switch from an external port that is mapped to the CPDMA channel. For Tx channels, the packets already contain data that will be sent out of the CPDMA channel and will enter the switch and make their way to the external port to go onto the Ethernet network. In both cases, a submit IOCTL must be completed before any data movement can happen.

The buffers that make up each packet will be added to the active hardware buffer queue, by adding the buffer descriptors associated with each buffer to the hardware buffer descriptor queue. This hardware buffer descriptor queue is defined by the channel's head descriptor pointer in the STATERAM, which points to the first buffer descriptor in a singly linked list of descriptors (with the next descriptor pointer of each descriptor providing the linking). As Rx data packets come into the CPDMA channel, the hardware can traverse the linked list to fill in the buffers. As Tx data is sent out of the CPDMA channel, the hardware will traverse the linked list, sending the buffers that comprise the packets until a the linked list is terminated by a NULL next descriptor pointer

From a software point of view, the queue of packets (and therefore the buffer descriptors associated with those packets) is maintained in the channel's InUseQueue packet queue. This software queue is important since it maintains a tail pointer to the last hardware-owned packet. With this tail pointer, the queue of packets being submitted to the hardware can quickly be attached to the currently active queue by simply linking the tail of the active queue to the head of the submitted queue and making the tail of the active queue point to the tail of the submitted queue. Therefore no copying or moving of descriptors is required.

One important point to be emphasized is that the queue of packets to be submitted must be completely and properly formatted by the entity calling the CPDMA_SUBMIT_PACKETS IOCTL. In particular, the buffer descriptors of each packet must be filled in completely. For Rx packets:

1) Write the next buffer descriptor pointer to each descriptor to create the linked list of descriptors. All buffer descriptors in the packet queue to be submitted MUST be linked together.
2) Write the buffer pointer for the buffer to be filled in.
3) Clear the offset field.
4) Write the buffer length with the number of bytes available in the buffer.
5) Clear the SOP, EOP, and EOQ bits.
6) Set the OWNER bit.

For Tx packets:

1) Write the next buffer descriptor pointer to each descriptor to create the linked list of descriptors. All buffer descriptors in the packet queue to be submitted MUST be linked together.
2) Write the buffer pointer for the buffer to be filled in.

3) For SOP descriptors, set the offset field as appropriate.
4) Write the buffer length with the number of bytes in the buffer.
5) Set the SOP and EOP bits as appropriate.
6) Clear the EOQ bits.
7) Set the OWNER bit.

3.2.4.2 Retrieve Packets

The CPDMA_RETRIEVE_PACKETS IOCTL is used to get packets that were previously owned by the hardware (given to the hardware via the CPDMA_SUBMIT_PACKETS IOCTL) back into the software domain. The channel's FromHardwareQueue holds all these packets that have already been used by the driver and are ready to be returned to be handled by the software. This IOCTL will simply copy the elements in the FromHardwareQueue to the queue provided by the caller of the IOCTL (the copy is efficient) and clear the FromHardareQueue.

This IOCTL is intended to be called by the task which has been notified via channel callback functions from the ISR that packets are ready. Once the queue of packets is returned to the software task via the CPDMA_RETRIEVE_PACKETS IOCTL, the packet contents must be dealt with.

For receive packets, the buffers must be given to the stack/application for which they are destined. This will typically require using the hPrivate pointer and casting it back to some data structure relevant to the stack/application. The size of received data will be set by the hardware in the buffer descriptors. The buffer descriptors of the CPDMA packets can be associated with new empty buffers and then be recycled back to the hardware (using the CPDMA_SUBMIT_PACKETS IOCTL). If splitting packets across multiple buffers, the software must have some a priori knowledge about the length of packets and how many buffers will be occupied, since the number of descriptors associated with each packet is a fixed value. If this can't be guaranteed, then the software must use a single packet of sufficient length and parse or split the buffer contents in the stack/application. This latter approach is taken with the NDK stack.

For transmit packets, the buffers that are being returned from the hardware have already been used for sending their contents, so typically these packets will simply be freed in the context of the stack or application. As with Rx packets, the hPrivate handle will be cast to a type relevant to the stack/application.

One unusual usage of the CPDMA_RETRIEVE_PACKETS IOCTL is following the opening of the channel. When the channel is opened, a collection of CPDMA_Packets is initialized in the memory region specified by the caller. All of these packets, which haven't technically been used by the driver, are put into the FromHardwareQueue of the channel so that they can be retrieved initially by the stack/application. The contents of the packets won't be meaningful at this stage. The packets for an Rx channel can be populated with buffers and then submitted to the driver so that receive operations can begin. The initial returned packets for a Tx channel can be set aside in a free queue so that when the stack or application sends content, the packets can be used to submit data to the hardware.

3.3 Prioritization and Queue Management

This section details the software interface and methods employed to make use of the various hardware features regarding prioritization and queue management

3.3.1 Rx Prioritization

The prioritization of receive packets begins when packets arrive at the external MAC port and continue all the way through the switch and into the CPDMA host port. The prioritization of packets also allows data flow partitioning by mapping different priority streams to different host CPDMA Rx channels. From a software point of view, the prioritization and partitioning is configured using the PORT_ADD_RX_PRIORITY_MAPPING IOCTL. The configuration structure that must be passed to this command (via an IOCTL command structure is as follows:

```
typedef struct PORT_RX_PRIORITY_CONFIG_
{
    CPDMA_ChannelHandle     hCpdmaRxChannel;
    PORT_SwitchPriority     switchPriority;
    PORT_RxSource           rxSource;
    PORT_RxPriorityType     rxPriorityType;
    PORT_RxPriorityOption   rxPriorityOption;
}
CPDMA_Channel;
```

The hCpdmaRxChannel is simply the handle to the opened Rx channel to which this prioritized traffic flow will be directed. The channel handle must be valid, meaning that the channel should be opened, but not necessarily started, before creating a priority mapping.

The switchPriority field is one of four values indicating the relative priority of the mapping we are creating. This value determines the internal Tx queue used in the egress from the switch's host port (as data moves out of the switch and into the CPDMA Rx engine). There are four logical Tx queues for data leaving the switch at each port, with the queue priority determining the chance of data being dropped at the 20KB FIFO buffers. The intention is that lower priority packets are dropped before higher priority packets. The highest priority (switch priority 3) packets are allowed to fill the entire FIFO, the second highest priority (switch priority 2) packets are allowed to fill all but the last 2KB of the FIFO, the next priority (switch priority 1) packets are allows to fill all but the last 4KB of the FIFO, and the lowest priority (switch priority 0) packets can fill all but the lowest 6KB of the FIFO. The switchPriority passed in to the IOCTL via the configuration structure can be HIGH, MEDIUM, LOW, or NONE. If all priorities were in use (involved in active mappings), these would correspond one-to-one to the switch priorities described above. But if less than four mappings are in use, the highest switch priorities are used. For example, if only NONE is in use by any active mappings, which is the case for best effort NDK traffic in the current implementation, then it would map to switch priority 3. But as soon as a new mapping with higher priority is introduced, such as adding an AVB stack at LOW priority, the mapping with LOW priority would take over switch priority 3, and the NONE priority would move down to switch priority 2. In this way, we make best use of the switch's FIFO memory.

The rxSource variable in the configuration structure is used to specify which of the two external ports is involved in the mapping. Mapping from switch priorities to the CPDMA channel can be done on a per port basis. The rxSource field can specify port 1, port 2, or both ports.

There are up to four different types of priority mapping (not all devices support all mapping types – in particular, older devices lack the IPv4 DiffServ mapping capability). The priority mapping type, in field rxPriorityType, can be ALL_UNTAGGED, IPV4_DIFFSERV, PRI_TAGGED, and VLAN_TAGGED. The first two

types are related to packets without VLAN tags, the last two types are for packets with VLAN tags. The ALL_UNTAGGED type is the default mapping when the other mappings don't apply and should be used with the NONE switch priority to provide mapping for all best effort traffic. The other types are for IPv4 differentiated services, described in RFC 2474, VLAN tagged packets with VLAN ID of zero (PRI_TAGGED, for priority tagging), and VLAN tagged packets with non-zero VLAN ID (VLAN_TAGGED). The selection in this field is used to setup the external MAC port to properly map to the packet priority to the proper switch priority. Though the external ports support up to eight priority values, the current implementation only uses four for each external port since the switch priority can only be one of four values.

The final configuration structure field, rxPriorityOption, is used to provide the required options for the type of mapping specified in the rxPriorityType field. This variable is a union of different types, one for each valid rxPriorityType. For each type, at least a priority field is required, which indicates the priority given to the packet at the external MAC port. For PRI_TAGGED and VLAN_TAGGED packets, the packet priority is derived from the PCP field of the VLAN tag, and the value should be set to match the expected value in the packets. For the ALL_UNTAGGED type, the priority value is actually set in hardware to be given to packets that don't match the other criteria. For IPV4_DIFFSERV mappings, the priority, along with the DSCP value is used to create a mapping in the hardware between the 6-bit DSCP field in the IPv4 packet to one of the eight priority values (which are further mapped to the four switch priorities).

Figure 4 shows the switch hardware and the priority, queue, and channel mappings after adding two mappings for both ports, one for best effort traffic (port packet priority 0, switch priority NONE, CPDMA channel 0) and one for VLAN-tagged traffic with PCP field equal to 5 (port packet priority 5, switch priority HIGH, CPDMA channel 1).

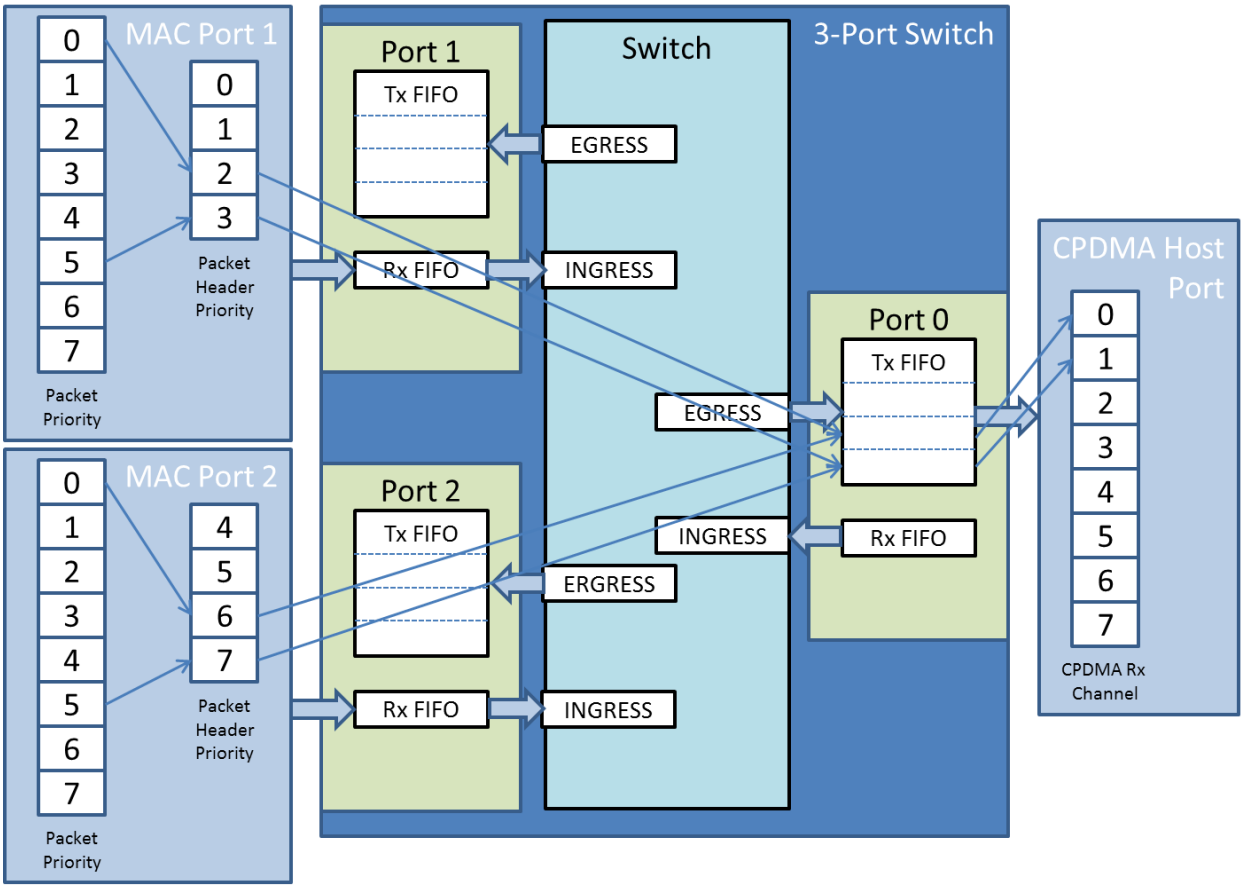

Figure 4. Receive priority mapping example for best effort traffic to CPDMA channel 0 and VLAN tagged packets with PCP=5 mapped to CPDMA channel 1.

3.3.2 Tx Rate Limiting

This feature is not currently implemented. This section will be updated when it is implemented.

3.4 Other Features

3.4.1 Time stamping

The time stamping capability of the GMACSW peripheral is embedded within the CPTS module. With the default driver configuration, the CPTS module is enabled and the CPTS counter is incrementing. Internal CPTS events (such as half-rollover and full-rollover) are handled via the MISC interrupt service routine. However, no external events (PTP packet Rx or Tx) are processed until a stack registers with the GMACSW driver using the CPTS_REGISTER_STACK IOCTL. This IOCTL registers a notification callback so that the stack can be notified of arriving PTP packets. These arriving external events are stored in a software queue. The hardware contains a queue of all events that have not been serviced by the ISR. The software queue, by comparison, is only utilized for external events that need to be processed by a PTP stack. The size of the software queue is determined by the application/stack that registers itself. Only one entity can be registered with the driver to be notified of events (there should be only one PTP stack running).

The CPTS module also implements IOCTLs for getting events from the software queue, clearing all events (in both the hardware and software queues), looking up events based on submitted criteria, and releasing used events back to the driver.

The frequency of the CPTS clock is stored in the CPTS configuration, but the CPTS driver does not actually configure any clock sources to achieve the configured value. Some other software is responsible for setting up the clocks to the value specified in the CPTS configuration. The driver default configuration contains reasonable/correct defaults for the supported platforms. The CPTS module also implements IOCTLs to set and get the CPTS frequency value, and these should be called if the system ever updates the CPTS clock to operate at some other value. Any such change should not be performed while a stack is registered and making use of the CPTS module.

3.4.2 Statistics Gathering

The STATS software module is a simple wrapper around the STATS hardware module, which is simply a collection of 32-bit registers which the hardware uses to accumulate byte counts and error counts for different aspects of the GMACSW peripheral.

The software module provides a mirrored structure of those registers in memory, but with 64-bits per statistic item instead of 32-bits. This allows the software to accumulate the statistics from the hardware for a long period of time without overflowing. To prevent overflow in hardware and to give notification of the statistics status, one of the MISC interrupts is for servicing the STATS module. When any register in the STATS module rolls over the half way mark (i.e. the uppermost bit of the 32-bit register changes from 0 to 1), the statistics MISC interrupt will trigger. The ISR can then perform a read of the registers and accumulate the value in the 64-bit wide software structure. The statistics registers are write-to-decrement. After reading the values from the registers, those same values are written back into the registers they came from. The new register value is the difference between the current register value and the value written into the register. This prevents the loss of any increments that happened between the read and the write, and allows proper accounting in software.

The software module supports two IOCTLs: STATS_CLEAR_STATS and STATS_GET_STATS. The first is used to clear the gathered statistics, both in the hardware registers and in the accumulating software structure. The second is used to read the current statistics, which causes an immediate update of the accumulating software structure of statistics and then a copy of that structure back to the calling application.

The statistics module can be enabled or disabled by the driver configuration given to the GMACSW_open() call. The default driver configuration enables the statistics gathering. The statistics module gathers data across both external ports, and there is no way to distinguish between the counts generated by the two ports.

3.4.3 MDIO Communication to PHYs

For the purpose of detecting external link status when connecting the MAC ports to external PHYs, the driver contains a PHY status state machine that should be driven by a periodic timer. The default configuration recommends a 100ms period. The MDIO state machine will perform auto-negotiation and auto-detection if configured to do so. Because the state machine is driven by a periodic call, it can take up to 500ms for the link status change to be noticed and a notification to take place. The GMACSW driver allows the user to supply a link status notification callback, so that the application can detect the changes in status. The driver also allows the

application or stack to register functions with the driver that will be called, in order, when the 100ms periodic timer expires.

The driver can also be configured to operate in a NOPHY mode by setting the MAC port initial configuration's mdioModeFlags field to MDIO_MODEFLG_NOPHY. In this mode, the MDIO state machine to talk to the PHYs is bypassed and the link status callback is unused. The driver assumes that the link is always active. This mode is intended to be used when connecting the MII/RMII/GMII/RGMII interfaces of two devices directly together.

4 Revision History

Version #	Date	Author Name	Revision History
v1.0	25 June 2014	Daniel Allred	Initial design document for NSP_GMACSW driver.