

QOS Scheduler Family Firmware Specification

Version B
Jul 17 2015

Document License

This work is licensed under the Creative Commons Attribution-NoDerivs 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nd/3.0/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

Contributors to this document

Copyright (C) 2015 Texas Instruments Incorporated - <http://www.ti.com/>

Revision Record	
Document Title:	Software Design Specification
Revision	Description of Change
A	Initial Release (Firmware v 2.1.0.9)
B	Add simultaneous packets and bytes to the narrow qos scheduler; add group stat query to narrow qos + wide qos (but not qos+drop) (Firmware v 2.1.0.10)

Contents

1. OVERVIEW.....	4
1.1 RELATED DOCUMENTS.....	4
1.2 OVERVIEW.....	4
1.2.1 What this Specification Provides	4
2. FUNCTIONAL DESCRIPTION.....	5
2.1 BASIC OPERATION.....	5
2.1.1 Drop Scheduler.....	5
2.1.2 Push Proxy	7
2.1.3 QoS Scheduler	7
2.1.4 Congestion Management	12
3. QOS ALGORITHM DESCRIPTION	12
3.1 SOFTWARE OVERVIEW.....	12
3.1.1 Pseudocode Configuration and State Data Structures	12
3.1.2 Foreground Task Pseudocode	15
3.1.3 Port Scheduler Pseudocode.....	15
3.1.4 Group Scheduler Pseudocode	20
3.1.5 Queue Scheduler Pseudocode	21
3.1.6 Drop Scheduler Pseudocode	21
3.1.7 Background Task (Congestion) Pseudocode.....	26
3.2 QoS SCHEDULER SHADOW CONFIGURATION SPECIFICATION	28
3.2.1 QoS Scheduler Queue	28
3.2.2 QoS Scheduler Group (Bytes or Packets)	28
3.2.3 QoS Scheduler Group (Bytes And Packets)	29
3.2.4 QoS Scheduler Physical Port (Bytes Or Packets).....	30
3.2.5 QoS Scheduler Physical Port (Bytes And Packets)	31
3.2.6 Complete Shadow Configuration Spec (QoS Scheduler Full/Lite Ports supporting Bytes or Packets) 32	
3.2.7 Complete Shadow Configuration Spec (QoS Scheduler Full/Lite Ports supporting Bytes And Packets) 33	
3.2.8 Drop Scheduler Queue Configuration	33
3.2.9 Drop Scheduler Config Profile Configuration in Shadow	34
3.2.10 Drop Scheduler Top Level Config in Shadow	35
3.2.11 Drop Scheduler Output Profile Config in Shadow	35
3.2.12 Dedicated Query Statistics Shadow Area	36
3.2.13 Group Statistics in Common Shadow Area	37
3.2.14 Push Statistics	38
3.2.15 Push Proxy	38
3.2.16 Input Queue Map for QoS Scheduler	39
3.2.17 Input Queue Map for Drop Scheduler	40
4. FIRMWARE COMMAND INTERFACE	41
4.1 FIRMWARE COMMAND HANDSHAKE	41
4.1.1 Command Handshake.....	41
4.1.2 Command Buffer	41
4.1.3 QoS Scheduler Queue Region Base	42
4.1.4 Timer Configuration.....	43
4.1.5 Enable / Disable QoS Scheduler Physical Port.....	43
4.1.6 Copy Configuration To/From Shadow.....	44
4.1.7 Stats Request.....	44
4.2 INTERNAL MEMORY ALLOCATION	45
4.2.1 PDSP / QMSS Scratch RAM Allocation	45

1. Overview

1.1 Related Documents

Document	Link

1.2 Overview

This document specifies the PDSP firmware operation and command interface of a “Quality of Service” (QoS) functionality designed to run on the Multicore Navigator of Keystone I/II devices. This document covers 3 different firmware builds which are “qos_sched”, “qos_sched_drop_sched”, and “qos_sched_wide”. These differ based on whether they have the drop scheduler, and the number and size of ports.

1.2.1 What this Specification Provides

- Functional Description
 - Basic Operation
 - Algorithm Details
- Host processor interface
 - Firmware command interface
 - Scratchpad memory usage

2. Functional Description

2.1 Basic Operation

The quality of service (QoS) PDSP is charged with the job of policing all packet flows in the system, and verifying that neither the peripherals nor the host CPU are overwhelmed with packets.

The key to the functionality of the QoS system is the arrangement of packet queues. There are two distinct scheduling blocks each of which has two sets of packet queues, the QoS ingress queues, and the final destination queues.

There is a high priority block called the drop scheduler which implements a model of tail drop and RED (Random Early Detect/Drop). It has 80 input queues each of which can be mapped to an output queue. Its goal is to run as fast as possible to keep the input queues empty, while examining the depth of the output queues. It therefore models tail drop or RED on the output queues when it transfers packets from the input queues. (It is a model of tail drop/RED, not actual tail drop/RED because it is not able to atomically act as part of a push operation. Thus a series of fast pushes will allow more queued descriptors than a pure tail drop/RED would allow implemented as part of the push).

There is a medium priority block which is the QoS Scheduler. Each QoS scheduler physical port maps to one destination queue that can be queue supported by the QMSS whether it is processed by peripherals, by one of the host processors, or even an input queue to another QoS scheduler port.

Finally there is a low priority block which enforces queue depth on the QoS scheduler ports.

2.1.1 Drop Scheduler

The drop scheduler runs once per timer tick.

If the drop scheduler is enabled in the build, then the build also only supports 20 lite ports. These lite ports do not support any of the group thresholds (cir/pir/wrr) because only the port's CIR is functionally necessary when there is one group.

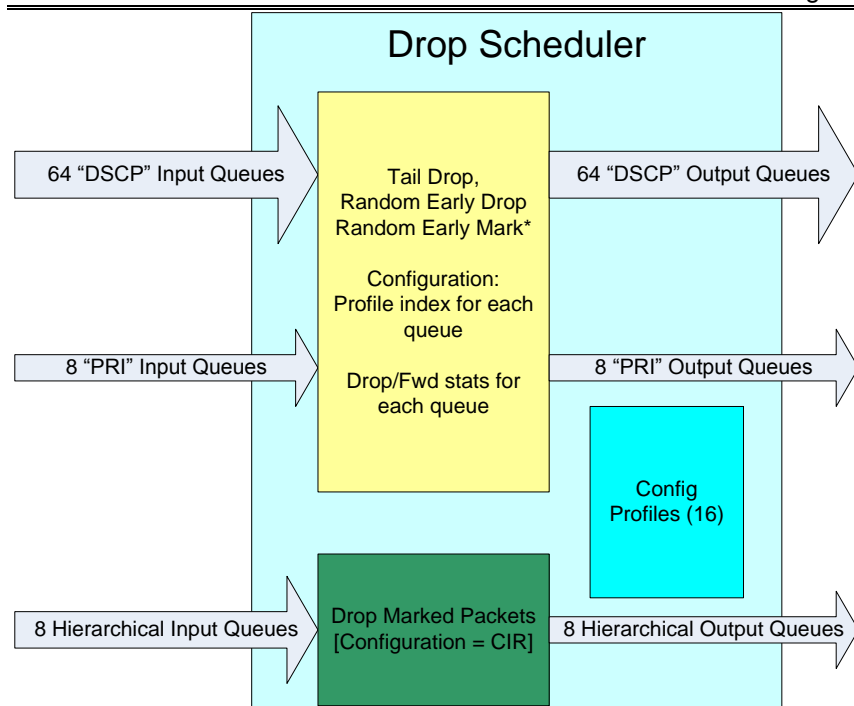


Figure 1: Drop Scheduler Block Diagram

2.1.1.1 Input queues

There are 80 regular input queues. There are no internal QoS assumptions based on these queues. However, it is dimensioned such that there is a queue for each of the 64 possible DSCP levels, plus 8 queues for each of the 8 possible 802.1p priorities. Each queue supports statistics, a mode (red mark, red drop, and tail drop), min/max average thresholds, absolute maximum threshold, a drop probability, and an output queue assignment.

There may be 8 additional hierarchical queues. These go together with RED mark. The details for these queues are a future feature.

2.1.1.2 Tail Drop

In tail drop mode, only the absolute maximum threshold (L_{abs}) in bytes or packets is used to drop packets. If the number of packets (or bytes) in the output queue exceeds the threshold then all input packets are dropped until the output queue falls below the threshold. The average thresholds are not used, nor is the drop probability used.

2.1.1.3 Fixed probability RED

A simple RED algorithm is supported. It can be configured to mark or drop packets. In this version, only drop is supported, but mark may be implemented as a future feature.

RED defines an upper threshold (L_{max}) and a lower threshold (L_{min}). These can be configured in bytes or packets units. There is also a drop probability P_d that is configured.

RED computes an average depth of the associated output queue. Its time constant (t_c) is configurable power of 2. Once per interval, the average is updated using (where D_i is instantaneous depth for averaging, input d_i is instantaneous depth of input queue, and output d_i is instantaneous depth of output queue. D_i , d_i are integers. d_a is a fixed point number (Q format) with the binary point at location t_c .

$$D_i = (\text{input } d_i / 2) + \text{output } d_i$$

$$d_a = d_a - (d_a \gg t_c) + D_i$$

If the average depth d_a of the output queue is:

$d_a \leq L_{min}$: no packets are marked or dropped

$d_a \geq L_{\max}$: 100% of the packets are marked or dropped

$L_{\min} < d_a < L_{\max}$: Packets are dropped/marked with probability = $P_d \cdot (d_a / (L_{\max} - L_{\min}))$

The absolute maximum threshold (L_{abs}) may be used together with RED. This allows a hard maximum of packets when desired, even if the average d_a hasn't converged.

If the instantaneous depth of the output queue (d_i) is:

$d_i \geq L_{\text{abs}}$: all input packets are dropped

If the instantaneous total packets in the output queue exceeds the absolute max threshold, all the packets are dropped from the input queue until the instantaneous total packets falls below the threshold.

2.1.1.4 Statistics

Each input queue has a set of associated statistics which are bytes forwarded, bytes dropped, packets forwarded, and packets dropped. Stats can be requested atomically so input statistics can be calculated from forward+drop. When operating in RED mark or RED drop mode, the average queue depth is also available as a statistic.

2.1.2 Push Proxy

There is a Push Proxy feature which enables pushing the C+D register of any queue. This can be used to workaround HW issues in the HW proxy on some devices. The FW waits until both size and pointer in the proxy become nonzero, then implements a 64-bit push, then sets the input parameters to 0. The queue number and size register shall be written together; the pointer register can be written separately in any order. See section 3.2.15.

2.1.3 QoS Scheduler

When the drop scheduler is not present, the QoS scheduler is arranged with a total of 12 physical ports. The first two physical ports are "full" physical ports that each supports 5 groups of 8 queues (40 total ingress queues per port). The last 10 physical ports are "lite" physical ports that each supports one group of four queues.

When the drop scheduler is present, the QoS scheduler supports 20 lite physical ports.

In the "wide" build, the QoS scheduler supports 1 full physical port with 17 groups of 8 queues (136 total ingress queues). Note that the "wide" build has a different interface/memory map due to the size of a shadow supporting 136 queues.

The QoS scheduler with 2 full ports and 10 lite ports supports CIR/PIR shaping by both bytes and packets simultaneously (as well as shaping either bytes or packets). The drop scheduler and wide builds do NOT support simultaneous bytes and packets.

The wide build and QoS with 2 full ports and 10 lite ports support querying an entire group of queue stats at a time. The build with drop scheduler only supports querying queue stats one queue at a time.

The QoS scheduler operates on a configurable timer. On each tick of the timer, each physical port will have an opportunity to schedule its CIR (committed information rate).

Each port will select group(s) to schedule using a weighted round robin algorithm (WRR). Each group also has a CIR separate from that port. Once a group is selected using WRR it will schedule packets from its queues up to its CIR. Packets are selected from queues using a combination of strict priority (SP) and weighted round robin (WRR).

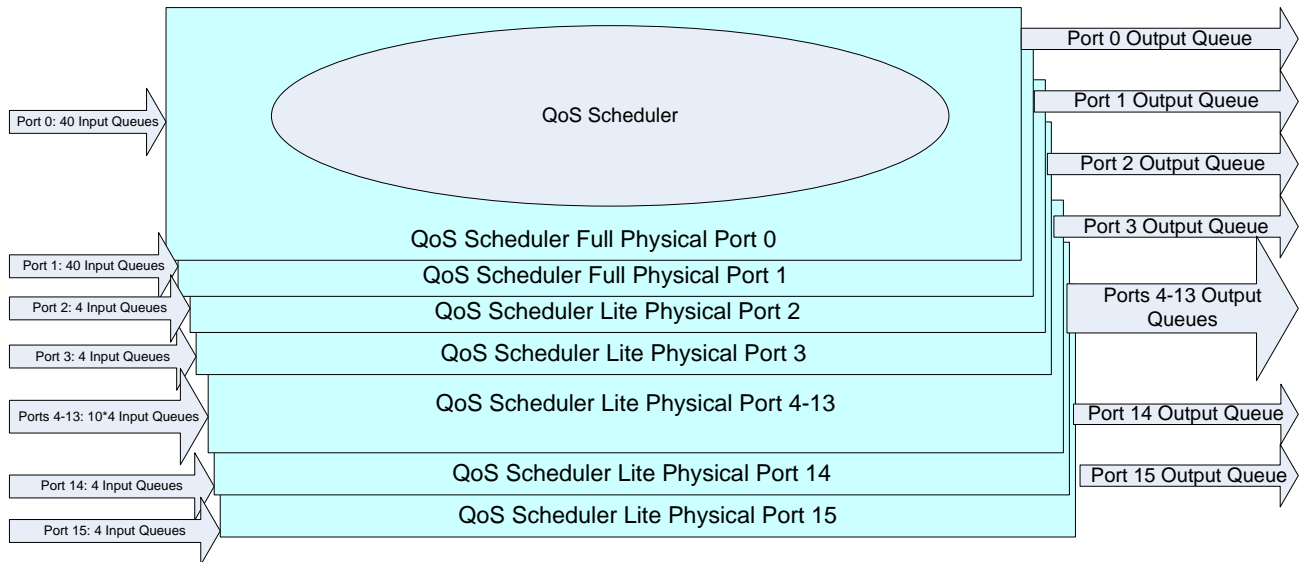


Figure 2: QoS Scheduler Block Diagram

2.1.3.1 QOS Ingress Queues

There is a designated set of queues in the system that feed into the QOS PDSP. These are called QOS queues. The QOS queues are simply queues that are controlled by the firmware running on the PDSP. There are no inherent properties of the queues that fix them to a specific purpose. Any queue aligned to a multiple of 32 queues can be configured to be the ingress queue base.

The input queues are statically assigned to the physical ports. Queues 0-39 are assigned to the first port, 40-79 to the second port, 80-83 to the third, with four queues assigned to each of the remaining ports.

2.1.3.2 Physical Ports

Each physical port has a configurable committed information rate (CIR) that is specified as a fraction of packets or bytes that are granted for each timer tick. It also has a maximum allowed CIR that prevents excessive credit from accumulating when there is traffic below the CIR.

Each “full” port supports up to 5 groups but software can configure fewer groups. Each “lite” port only supports a single group. The arrangement of full and lite ports and their connection to queues in the system are shown in Figure 3 and Figure 4.

The function of a port is to grant itself CIR credit each timer tick, then to select groups to schedule packets using weighted round robin. Each group has a configurable weight (in bytes or packets) for this purpose.

Each physical port also supports an output throttle threshold. This prevents the port from forwarding packets if the output queue is not draining. However, credits are granted and capped like normal even when throttled.

Two adjacent physical lite ports can be combined into a “joint” port that supports 8 inputs. This must be an even/odd pair, where if port 0 is even, the odd port is 1. This works either with or without drop scheduler.

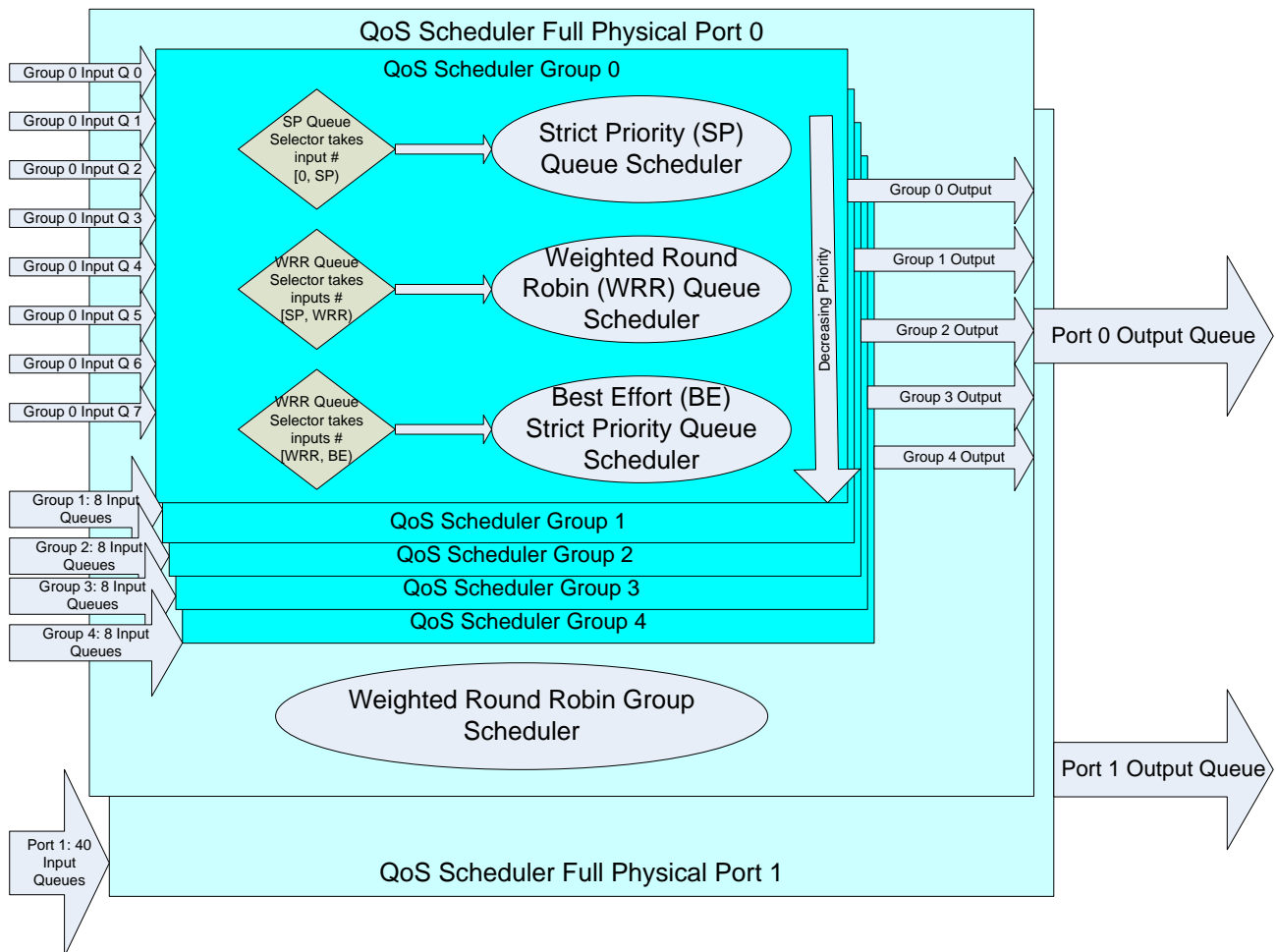


Figure 3: QoS Scheduler Full Port Block Diagram

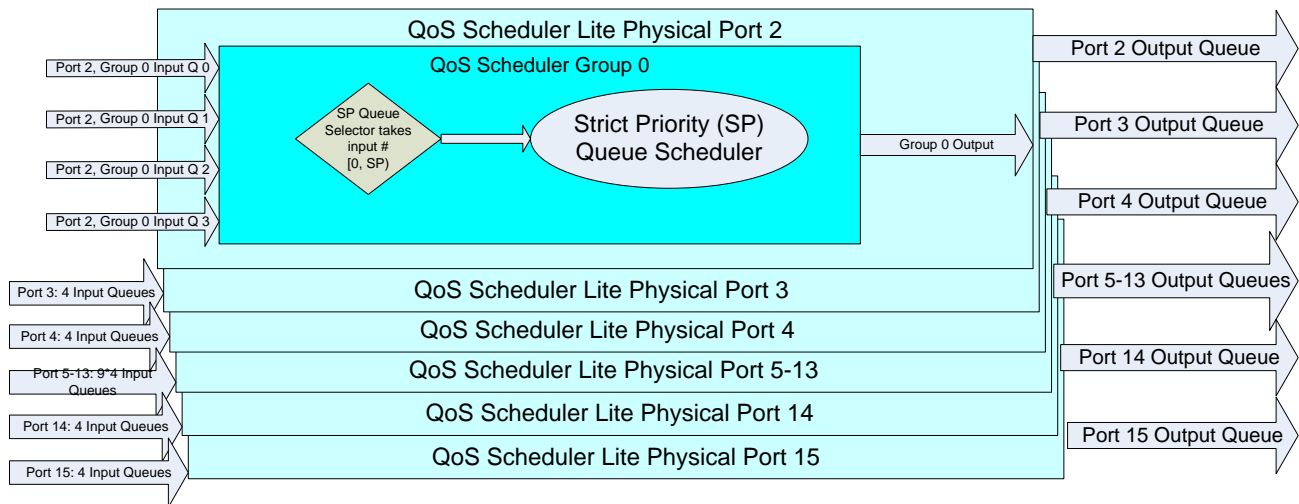


Figure 4: QoS Scheduler Lite Port Block Diagram

2.1.3.3 Groups

Each group contains up to 8 queues (4 on a lite port). Each group has a weight so the port can perform weighted round robin scheduling across all the groups. Each group has a configurable CIR and a peak information rate (PIR) associated with it. The port first gives each group an opportunity to use its CIR. If all groups have a chance at their respective CIR, then each group has an opportunity to use the rest of the port's CIR using the group's PIR.

The queues can be divided among strict priority queues (SP), weighted round robin (WRR), and best effort (BE) queues.

The queues are arranged in each group in strict priority order such that queue 0 is highest priority and queue 7 is the lowest priority. The strict priority queues must start at 0, while the WRR queues must follow the SP queues, and the BE queues must follow the WRR. It is legal to have 0 to 8 queues of each type as long as the total is ≤ 8 .

This is configured by specifying the total number of queues, the number of SP queues, and the number of WRR queues. The number of BE queues is $BE = total - wrr - sp$.

2.1.3.4 Queues within Groups

2.1.3.4.1 Strict Priority Queues

The first queues can be strict priority queues. This means that all the packets from queue 0 must be drained before any packets can be drained from queue 1. Packets from queue 2 will only be drained if there are no packets on queues 0 and 1, and so on.

2.1.3.4.2 Weighted Round Robin Queues

The weighted round robin queues are only drained after all the SP queues are drained. Each queue has an associated weight specified in bytes or packets. Packets are scheduled in a round robin fashion unless the queue has no remaining weight credit.

2.1.3.4.3 Best Effort Queues

Best effort queues follow the WRR queues. These are additional strict priority queues, but they are lower priority than the WRR queues.

2.1.3.5 Port configuration rules

The ports are configured using a shadow configuration in the PDSP's scratch memory. There are commands to copy one of the port's current configuration to the shadow area, to copy the shadow area to one of the port's active configuration. This enables reconfiguration of credits without having potentially inconsistent configurations actively in use. Note that the number of groups or queues per group should not be changed (especially decreased) otherwise descriptors can be left (leaked) in the newly disabled queues. The firmware will recycle all the descriptors on all the queues of a port when the port is disabled.

2.1.4 Congestion Management

Each QoS scheduler queue can be configured with an optional congestion threshold (a value of 0 disabled congestion dropping). Whenever the firmware is waiting for a timer tick, it will check all the queues with configured congestion thresholds to see if the number of bytes or number of packets on the queue exceeds the threshold. It will drop packets from the head of the queue until the number of bytes or packets is below the threshold.

For "normal" operation, the QoS firmware does not examine anything inside the descriptors or associated buffers (if any). However, in order to drop descriptors, it examines four fields within the descriptor which presumes the descriptor is formatted as a CPPI descriptor. The "Packet Return Queue Mgr #" and "Packet Return Queue #" fields are used to return the descriptor as if the packet were consumed by a packet DMA. The "Return Push Policy" field is honored. Finally the "Packet Id" must be set to monolithic (2) or host (0).

If the congestion threshold is disabled, then the addresses pointed to be the descriptors are not touched, and therefore, do not even have to point to real memory.

3. QOS Algorithm Description

3.1 Software Overview

The firmware assumes 104 QOS queues are allocated to the QOS PDSP. They are physically located at a fixed base (most likely not zero), but are referred to as QOS queues 0 through 103 in configuration. The base queue should be configured by the application after allocating a block of 104 contiguous queues aligned to a multiple of 32 queues.

The algorithm is specified by the following pseudocode. An executable version of the foreground task is used as part of the QMSS LLD's unit test for the QoS Scheduler Firmware. An executable version of the background task is not used because a cycle-exact model would be needed to demonstrate correct operation.

3.1.1 Pseudocode Configuration and State Data Structures

```
#define QMSS_QOS_SCHED_BYTES_SCALE_SHIFT 11
#define QMSS_QOS_SCHED_PACKETS_SCALE_SHIFT 20
#define QMSS_QOS_WRR_BYTES_SCALE_SHIFT (QMSS_QOS_SCHED_BYTES_SCALE_SHIFT - 3)
#define QMSS_QOS_WRR_PACKETS_SCALE_SHIFT (QMSS_QOS_SCHED_PACKETS_SCALE_SHIFT - 3)
#define NUM_PHYS_PORTS 16
#define NUM_DROP_CFG_PROFILES 16
#define NUM_DROP_DSCP_QUEUES 64
#define NUM_DROP_PRI_QUEUES 8
#define NUM_DROP_INPUT_QUEUES (NUM_DROP_DSCP_QUEUES + NUM_DROP_PRI_QUEUES + 8)
#define NUM_DROP_STATS_BLOCKS 48
```

```
#define NUM_DROP_OUTPUT_PROFILES 36
#define NUM_QOS_QUEUES (40+40+14*4)

typedef struct _QOSQUEUE_PROC {
    int32_t WrrCurrentCredit; // Current Queue WRR credit
    uint32_t PacketsForwarded; // Number of packets forwarded
    uint32_t PacketsDropped; // Number of packets dropped
    uint64_t BytesForwarded; // Number of bytes forwarded
    uint64_t BytesDropped; // Number of bytes dropped
    uint16_t QueueNumber; // Input queue
} QOSQUEUE_PROC;

typedef struct _QOSQUEUE_CFG {
    int32_t WrrInitialCredit; // Initial Queue WRR credit on a "new" schedule
    uint32_t CongestionThresh; // The max amount of congestion before drop
} QOSQUEUE_CFG;

typedef struct _LOGICAL_GRP_PROC {
    int32_t CirCurrentByByte; // Current CIR credit
    int32_t CirCurrentByPkt; // Current CIR credit
    int32_t PirCurrentByByte; // Current PIR credit
    int32_t PirCurrentByPkt; // Current PIR credit
    uint8_t NextQueue; // The next RR queue to examine in the group
    uint8_t WrrCreditMask; // Flag mask of WRR queues that have WRR credit
    remaining
    int32_t WrrCurrentCredit; // Current Group WRR credit
    QOSQUEUE_PROC Queue[8]; // Up to eight queues per logical group
} LOGICAL_GRP_PROC;

typedef struct _LOGICAL_GRP_CFG {
    bool fIsSupportByteShaping; // scheduling using *ByByte is enabled
    bool fIsSupportPacketShaping; // scheduling using *ByPacket is enabled
    int32_t CirIterationByByte; // CIR credit per iteration
    int32_t CirIterationByPkt; // CIR credit per iteration
    int32_t PirIterationByByte; // PIR credit per iteration
    int32_t PirIterationByPkt; // PIR credit per iteration
    int32_t CirMaxByByte; // Max total CIR credit
    int32_t PirMaxByByte; // Max total PIR credit
    int32_t CirMaxByPkt; // Max total CIR credit
    int32_t PirMaxByPkt; // Max total PIR credit
    int32_t WrrInitialCredit; // Initial Group WRR credit on a "new" schedule
    uint8_t QueueCount; // Total number of active QoS queues (up to 8)
    uint8_t SPCount; // The number of SP queues (usually 2 or 3)
    uint8_t RRCount; // The number of RR queues (usually QueueCount-SPCount)
    QOSQUEUE_CFG Queue[8]; // Up to eight queues per logical group
} LOGICAL_GRP_CFG;

typedef struct _PHYS_PORT_PROC {
    bool fEnabled; // port enable flag
    int32_t CirCurrentByByte; // Current CIR credit
    int32_t CirCurrentByPkt; // Current CIR credit
    uint8_t WrrCreditMask; // Flag mask of WRR groups that have WRR credit
    remaining
    uint8_t NextGroup; // The next RR group to examine
    uint8_t LastTimerTicks; // Used to schedule missed interrupts. Initialized
    // to TimerTicks when port is turned on.
    LOGICAL_GRP_PROC Group[5]; // Up to 5 logical groups
} PHYS_PORT_PROC;

typedef struct _PHYS_PORT_CFG {
    bool fByteWrrCredits; // When set, WRR credits are always in bytes
    bool fByteCongest; // When set, congestion is in bytes, else packets
    bool fByteDestThrottle; // dest throttle is bytes, else packets
    bool fIsJoint; // When set, even/odd pair of ports behaves as one
    bool fIsSupportByteShaping; // scheduling using *ByByte is enabled
    bool fIsSupportPacketShaping; // scheduling using *ByPacket is enabled
    int32_t CirIterationByByte; // CIR credit per iteration
    int32_t CirIterationByPkt; // CIR credit per iteration
    int32_t CirMaxByByte; // Max total CIR credit
    int32_t CirMaxByPkt; // Max total CIR credit (always in bytes)
    uint8_t GroupCount; // The number of logical groups
    uint8_t OverheadBytes; // Number of bytes of wire overhead to account, beyond
    packet size in QM.
```

```

deducted, // This is often set to 24. This only affects credits

configured // not statistics. It also only has effect on credits

// as bytes, not packets.
uint8_t RemoveBytes; // Number of bytes to remove from each packet size
uint16_t DestThrottleThresh;
uint16_t DestQueueNumber; // Output queue
LOGICAL_GRP_CFG Group[5]; // Up to 5 logical groups
} PHYS_PORT_CFG;

typedef struct _DROP_CFG_PROFILE
{
    bool fByteTailThresh; // Units for tail drop are bytes
#define DROP_MODE_TAIL_ONLY 0 // Tail drop only
#define DROP_MODE_RED 1 // Random Early Drop
#define DROP_MODE_REM 2 // Random Early Mark (Not currently supported)
    uint8_t Mode;
    uint8_t TC; // Time constant as shift
    uint32_t RedThreshLow; // Avg Below this threshold, no packets are dropped/marked
// Between these thresholds, packets are dropped/marked
// randomly with probability
// Formula below assumes floating point so q format shifts
are omitted
- redThreshLow) // pscale = (avg queue depth - redThreshLow) / (redThreshHigh
// prob = pscale * redProb
    uint32_t RedThreshHigh; // Avg Above this threshold all packets are dropped/marked
    uint32_t RedHighMLowRecip; // 1/((RedThreshHigh - RedThreshLow)<<TC) in Q16
    uint32_t TailThresh; // Above this threshold all packets are dropped (0 disables)
// configuration ends, state begins
} DROP_CFG_PROFILE;

typedef struct _DROP_OUTPUT_PROFILE
{
    uint16_t DestQueueNumber; // Output queue
    uint16_t RedProb; // Drop/mark probability in Q16 (0x8000 = 0.5).
    uint8_t CfgProfIdx; // Configuration of thresholds
    bool fEnabled; // Profile is valid/enabled
// configuration ends, state begins
    uint32_t QAvg; // Average Q depth in bytes in Q DROP_PROFILE.TC
} DROP_OUTPUT_PROFILE;

typedef struct _DROP_STATS
{
    uint32_t BytesForwarded; // Bytes Forwarded
    uint32_t BytesDropped; // Bytes Dropped (or marked)
    uint32_t PacketsForwarded; // Packets Forwarded
    uint32_t PacketsDropped; // Packets dropped
} DROP_STATS;

typedef struct _DROP_QUEUE
{
    bool fEnabled; // queue is enabled/valid
    uint8_t StatsBlockIdx; // Stats block to update
    uint8_t OutProfIdx; // Output que profile index
// configuration ends, state begins
} DROP_QUEUE;

typedef struct _DROP_SCHED
{
    bool fEnabled; // Drop Sched Enabled?
    uint8_t Interrupt; // Interrupt number in INTD to use to signal stats
overflow
    uint16_t BaseQueue;
    uint32_t rng_s1; // Random seed for deciding whether to drop packets
    uint32_t rng_s2; // Random seed for deciding whether to drop packets
    uint32_t rng_s3; // Random seed for deciding whether to drop packets
    DROP_QUEUE Queues[NUM_DROP_INPUT_QUEUES]; // drop queues
    DROP_CFG_PROFILE CfgProfiles[NUM_DROP_CFG_PROFILES];
    DROP_OUTPUT_PROFILE OutProfiles[NUM_DROP_OUTPUT_PROFILES];
    DROP_STATS StatsBlocks[NUM_DROP_STATS_BLOCKS];
    uint32_t queEnBits[(NUM_DROP_INPUT_QUEUES + 31) / 32];

```

```
// configuration ends, state begins
} DROP_SCHED;

typedef struct _QOS_SCHED
{
    uint8_t      PortCount;
    uint8_t      TimerTicks;
    uint16_t     BaseQueue;
    PHYS_PORT_CFG PortsCfg[NUM_PHYS_PORTS];
    PHYS_PORT_PROC PortsProc[NUM_PHYS_PORTS];
} QOS_SCHED;

DROP_SCHED DropSched;
QOS_SCHED  QoSSched;
```

3.1.2 Foreground Task Pseudocode

```
void ForegroundTask(QOS_SCHED *sched)
{
    uint32_t i;

    // Process one control message
    check_for_cmd();

    // Run Drop Scheduler
    if (DropSched.fEnabled)
    {
        DropScheduler (&DropSched);
    }

    // Schedule packets from all active physical ports
    for(i=0; i<sched->PortCount; i++)
        if (sched->PortsProc[i].fEnabled)
            PhysPortScheduler(sched, &sched->PortsCfg[i], &sched->PortsProc[i]);
}
```

3.1.3 Port Scheduler Pseudocode

```
// Returns 0 if no space left, else 1
int32_t PhysPortUpdateOutputSpace (PHYS_PORT_CFG *pPort, int32_t *OutputSpaceAvail, int32_t BytesUsed)
{
    if (*OutputSpaceAvail)
    {
        if (pPort->fByteDestThrottle)
        {
            *OutputSpaceAvail -= BytesUsed + pPort->OverheadBytes - pPort->RemoveBytes;
        }
        else
        {
            (*OutputSpaceAvail)--;
        }

        if (*OutputSpaceAvail <= 0)
        {
            return 0;
        }
    }

    return 1;
}

int isCreditAvail (int32_t credit1, int32_t credit2, uint8_t flag1, uint8_t flag2)
{
    int creditAvail = 1; /* flag1 || flag2 if do care above invalid config */
    if((credit1 <= 0) && (flag1)) {
        creditAvail = 0;
    }
    if((credit2 <= 0) && (flag2)) {
        creditAvail = 0;
    }
    // don't care if neither fIsSupportPacketShaping nor fIsSupportByteShaping is set; invalid config
```

```

    return creditAvail;
}

//
// This is the function that schedules packets on a physical port
//
void PhysPortScheduler(QOS_SCHED *sched, PHYS_PORT_CFG *pPortCfg, PHYS_PORT_PROC *pPortProc)
{
    int32_t BytesUsed;           // Bytes used is returned from the Logical Scheduler
    uint8_t PacketPendingMask;   // Flag mask of RR groups that are not empty
#ifdef MULTIGROUP
    int32_t WrrCreditUsed;       // Wrr Credit used (in packets or bytes as configured)
    uint8_t PirCreditMask = 0;   // Flag set when more PIR credit remains
#endif
    int     fPacketsSent;
    int     i;
    int     OutputSpaceAvail = 0;

    /* Add credits for all TimerTicks that occurred since last time this port ran */
    while ((uint8_t)(sched->TimerTicks - pPortProc->LastTimerTicks) > 0)
    {
        pPortProc->LastTimerTicks++;
        //
        // Add credits for all time based credit counters
        //

        // Credit for the main port
        if (pPortCfg->fIsSupportPacketShaping)
        {
            pPortProc->CirCurrentByPkt += pPortCfg->CirIterationByPkt;
            if( pPortProc->CirCurrentByPkt > pPortCfg->CirMaxByPkt )
                pPortProc->CirCurrentByPkt = pPortCfg->CirMaxByPkt;
        }
        if (pPortCfg->fIsSupportByteShaping)
        {
            pPortProc->CirCurrentByByte += pPortCfg->CirIterationByByte;
            if( pPortProc->CirCurrentByByte > pPortCfg->CirMaxByByte )
                pPortProc->CirCurrentByByte = pPortCfg->CirMaxByByte;
        }

        // Credit for the port's logical groups
#ifdef MULTIGROUP
        for( i=0; i<pPortCfg->GroupCount; i++ )
        {
            if (pPortCfg->Group[i].fIsSupportPacketShaping)
            {
                pPortProc->Group[i].CirCurrentByPkt += pPortCfg->Group[i].CirIterationByPkt;
                // Cap CIR credit at its max level
                if( pPortProc->Group[i].CirCurrentByPkt > pPortCfg->Group[i].CirMaxByPkt )
                    pPortProc->Group[i].CirCurrentByPkt = pPortCfg->Group[i].CirMaxByPkt;
                pPortProc->Group[i].PirCurrentByPkt += pPortCfg->Group[i].PirIterationByPkt;
                if( pPortProc->Group[i].PirCurrentByPkt > 0 )
                {
                    // Track every group with PIR credit for later
                    PirCreditMask |= (1<<i);
                    // Cap PIR credit at its max level
                    if( pPortProc->Group[i].PirCurrentByPkt > pPortCfg->Group[i].PirMaxByPkt )
                        pPortProc->Group[i].PirCurrentByPkt = pPortCfg->Group[i].PirMaxByPkt;
                }
            }

            if (pPortCfg->Group[i].fIsSupportByteShaping)
            {
                pPortProc->Group[i].CirCurrentByByte += pPortCfg->Group[i].CirIterationByByte;
                // Cap CIR credit at its max level
                if( pPortProc->Group[i].CirCurrentByByte > pPortCfg->Group[i].CirMaxByByte )
                    pPortProc->Group[i].CirCurrentByByte = pPortCfg->Group[i].CirMaxByByte;
                pPortProc->Group[i].PirCurrentByByte += pPortCfg->Group[i].PirIterationByByte;

                PirCreditMask &= ~(1<<i);

                if( pPortProc->Group[i].PirCurrentByByte > 0 )

```



```

        {
            // Track every group with PIR credit for later
            PirCreditMask |= (1<<i);
            // Cap PIR credit at its max level
            if( pPortProc->Group[i].PirCurrentByByte > pPortCfg->Group[i].PirMaxByByte )
                pPortProc->Group[i].PirCurrentByByte = pPortCfg->Group[i].PirMaxByByte;
        }
    }
}

#endif
}

/* Find out how much room is left in output queue */
if (pPortCfg->DestThrottleThresh)
{
    OutputSpaceAvail = pPortCfg->DestThrottleThresh;
    OutputSpaceAvail -= getQueueLength (pPortCfg->DestQueueNumber, pPortCfg->fByteDestThrottle);
    // No room in output queue */
    if (OutputSpaceAvail <= 0)
    {
        return;
    }
}

// Assume all groups have packets pending until we find out otherwise
PacketPendingMask = 0xFFFFF;

//
// Schedule each logic group's CIR, while also ensuring that the
// physical port's CIR is not violated.
// If the physical port has no credit quit out of the scheduler entirely
if (!isCreditAvail (pPortProc->CirCurrentByPkt, pPortProc->CirCurrentByByte,
                    pPortCfg->fIsSupportPacketShaping, pPortCfg->fIsSupportByteShaping))
    return;

// Foreground task can exit once all packets are sent either because
// the input queues are empty, or we ran out of group CIR, or we run
// out of port CIR.
do
{
    fPacketsSent = 0;

    for( i=0; i<pPortCfg->GroupCount; i++ )
    {
#ifdef MULTIGROUP
        if (isCreditAvail (pPortProc->Group[i].CirCurrentByPkt, pPortProc->
>Group[i].CirCurrentByByte,
                        pPortCfg->Group[i].fIsSupportPacketShaping, pPortCfg->
>Group[i].fIsSupportByteShaping))
#endif
        {
            // Attempt to schedule a packet
            BytesUsed = LogicalGroupScheduler( sched, pPortCfg, &pPortCfg->Group[i], &pPortProc->
>Group[i] );

            // If no packet scheduled, clear the pending mask
            if( !BytesUsed )
            {
                PacketPendingMask &= ~(1<<i);
            }
            else
            {
                uint32_t bytes = (uint32_t)BytesUsed & ~0x40000000;
                uint32_t bytesAdjusted = (bytes + pPortCfg->OverheadBytes - pPortCfg->
>RemoveBytes) << QMSS_QOS_SCHED_BYTES_SCALE_SHIFT;
                uint32_t packetsAdjusted = 1 << QMSS_QOS_SCHED_PACKETS_SCALE_SHIFT;

                if (pPortCfg->fIsSupportByteShaping)
                {
                    pPortProc->CirCurrentByByte -= bytesAdjusted;
                }
            }
#ifdef MULTIGROUP
            if (pPortCfg->Group[i].fIsSupportByteShaping)

```

```

        {
            pPortProc->Group[i].CirCurrentByByte -= bytesAdjusted;
            pPortProc->Group[i].PirCurrentByByte -= bytesAdjusted;
        }
#endif

        if (pPortCfg->fIsSupportPacketShaping)
        {
            pPortProc->CirCurrentByPkt          -= packetsAdjusted;
        }

#ifdef MULTIGROUP
        if (pPortCfg->Group[i].fIsSupportPacketShaping)
        {
            pPortProc->Group[i].CirCurrentByPkt -= packetsAdjusted;
            pPortProc->Group[i].PirCurrentByPkt -= packetsAdjusted;
        }
#endif

        fPacketsSent = 1;

        // If the physical port has no credit quit out of the scheduler entirely
        if (!isCreditAvail (pPortProc->CirCurrentByPkt, pPortProc->CirCurrentByByte,
            pPortCfg->fIsSupportPacketShaping, pPortCfg->fIsSupportByteShaping))
        {
            return;
        }

        // See if we used up output space
        if (PhysPortUpdateOutputSpace (pPortCfg, &OutputSpaceAvail, bytes) == 0)
        {
            return;
        }
    }
#ifdef MULTIGROUP
}
#endif
}
} while (fPacketsSent);
//
// Schedule each logic group's PIR in a WRR fashion while the
// physical port's CIR is not violated.
//
#ifdef MULTIGROUP
do
{
    // If there are no groups left with PIR group credit and packets, then we're done
    if( !(PirCreditMask & PacketPendingMask) )
        return;

    // If all groups with WRR credit remaining are empty, add WRR credit
    while( !( (PirCreditMask & pPortProc->WrrCreditMask & PacketPendingMask) )
    {
        // Reset credits
        for(i=0; i<pPortCfg->GroupCount; i++)
        {
            pPortProc->Group[i].WrrCurrentCredit += pPortCfg->Group[i].WrrInitialCredit;

            if (pPortProc->Group[i].WrrCurrentCredit > (pPortCfg->Group[i].WrrInitialCredit <<
1))
                pPortProc->Group[i].WrrCurrentCredit = (pPortCfg->Group[i].WrrInitialCredit <<
1);

            if (pPortProc->Group[i].WrrCurrentCredit > 0 || (! pPortCfg->
>Group[i].WrrInitialCredit))
                pPortProc->WrrCreditMask |= (1<<i);
        }
        // while loop will always terminate because PirCreditMask & PacketPendingMask check
    }

    // If this group has PIR credit, WRR credit, and packets pending, then schedule a packet
    if( (PirCreditMask & pPortProc->WrrCreditMask & PacketPendingMask) & (1<<pPortProc->
>NextGroup) )
    {
        // Attempt to schedule a packet
    }
}

```

```

    BytesUsed = LogicalGroupScheduler( sched, pPortCfg, &pPortCfg->Group[pPortProc-
>NextGroup], &pPortProc->Group[pPortProc->NextGroup]);

    // If no packet scheduled, clear the pending mask
    if( !BytesUsed )
        PacketPendingMask &= ~(1<<pPortProc->NextGroup);
    else
    {
        uint32_t bytes = (uint32_t)BytesUsed & ~0x40000000;
        uint32_t bytesAdjusted = (bytes + pPortCfg->OverheadBytes - pPortCfg->RemoveBytes);
        uint32_t packetsAdjusted = 1 << QMSS_QOS_SCHED_PACKETS_SCALE_SHIFT;

        // Use packet or byte count, depending on configuration
        if( pPortCfg->fByteWrrCredits )
            WrrCreditUsed = bytesAdjusted << QMSS_QOS_WRR_BYTES_SCALE_SHIFT;
        else
            WrrCreditUsed = 1 << QMSS_QOS_WRR_PACKETS_SCALE_SHIFT;

        // We also deduct the WRR credit
        pPortProc->Group[pPortProc->NextGroup].WrrCurrentCredit -= WrrCreditUsed;
        bytesAdjusted <=< QMSS_QOS_SCHED_BYTES_SCALE_SHIFT;

        // Deduct the PIR/CIR credit
        if (pPortCfg->fIsSupportPacketShaping)
        {
            pPortProc->CirCurrentByPkt -= packetsAdjusted;
        }

        if (pPortCfg->Group[pPortProc->NextGroup].fIsSupportPacketShaping)
        {
            pPortProc->Group[pPortProc->NextGroup].PirCurrentByPkt -= packetsAdjusted;
        }

        // Deduct the PIR/CIR credit
        if (pPortCfg->fIsSupportByteShaping)
        {
            pPortProc->CirCurrentByByte -= bytesAdjusted;
        }

        if (pPortCfg->Group[pPortProc->NextGroup].fIsSupportByteShaping)
        {
            pPortProc->Group[pPortProc->NextGroup].PirCurrentByByte -= bytesAdjusted;
        }

        // Clear the group's PIR credit mask if we depleted the PIR credit
        if (!isCreditAvail (pPortProc->Group[pPortProc->NextGroup].PirCurrentByPkt,
pPortProc->Group[pPortProc->NextGroup].PirCurrentByByte,
pPortCfg->Group[pPortProc->NextGroup].fIsSupportPacketShaping,
pPortCfg->Group[pPortProc->NextGroup].fIsSupportByteShaping))
            PirCreditMask &= ~(1<<pPortProc->NextGroup);

        // Clear the group's WRR credit mask if we depleted the WRR credit
        if( pPortProc->Group[pPortProc->NextGroup].WrrCurrentCredit <= 0 )
            pPortProc->WrrCreditMask &= ~(1<<pPortProc->NextGroup);

        // See if we used up output space
        if (PhysPortUpdateOutputSpace (pPortCfg, &OutputSpaceAvail, bytes) == 0)
        {
            return;
        }
    }
}

// Move on to the next group
pPortProc->NextGroup++;
if( pPortProc->NextGroup == pPortCfg->GroupCount )
    pPortProc->NextGroup = 0;

} while (isCreditAvail (pPortProc->CirCurrentByPkt, pPortProc->CirCurrentByByte,
pPortCfg->fIsSupportPacketShaping, pPortCfg->fIsSupportByteShaping));
#endif // MULTIGROUP
}

```

3.1.4 Group Scheduler Pseudocode

```
//
// This is the function that schedules a single packet from queues on a logical group
// The function returns the packet size of the packet selected
//
int32_t LogicalGroupScheduler(QOS_SCHED *sched, PHYS_PORT_CFG *pPortCfg, LOGICAL_GRP_CFG *pGroupCfg,
LOGICAL_GRP_PROC *pGroupProc)
{
    int32_t    BytesUsed;
    int32_t    packetSent = 0;
    uint8_t    PacketPendingMask;
    int        i, j;

    PollProxy();
    if (timerExpired()) // this costs 1 PDSP cycle if timer didn't expire
    {
        clearTimer();
        sched->TimerTicks++;
    }
    // With queues, we can directly read the pending status
    PacketPendingMask = ReadQosQueuePendingBits(pGroupCfg, pGroupProc);

    // If no packets, nothing to do
    if(!PacketPendingMask)
        return 0;

    //
    // Try to take a high priority queue first
    //
    for( i=0; i<pGroupCfg->SPCount; i++ )
    {
        if( PacketPendingMask & (1<<i) )
            return( QosQueueScheduler(pPortCfg, &pGroupProc->Queue[i]));
    }

    //
    // Next try to pick a round robin queue
    //
    if (PacketPendingMask & (((1 << pGroupCfg->RRCount) - 1) << pGroupCfg->SPCount))
    {
        // There are RR packets pending
        for( i=0; i<pGroupCfg->RRCount; i++ )
        {
            // If all queues with WRR credit remaining are empty, reset the credit
            while ( !(pGroupProc->WrrCreditMask & PacketPendingMask) )
            {
                // Reset credits
                for(j=pGroupCfg->SPCount; j<(pGroupCfg->SPCount+pGroupCfg->RRCount); j++)
                {
                    pGroupProc->Queue[j].WrrCurrentCredit += pGroupCfg->Queue[j].WrrInitialCredit;

                    if (pGroupProc->Queue[j].WrrCurrentCredit > (pGroupCfg->Queue[j].WrrInitialCredit
<< 1))
                        pGroupProc->Queue[j].WrrCurrentCredit = (pGroupCfg->Queue[j].WrrInitialCredit
<< 1);

                    if (pGroupProc->Queue[j].WrrCurrentCredit > 0 || (! pGroupCfg->
>Queue[j].WrrInitialCredit))
                        pGroupProc->WrrCreditMask |= (1<<j);
                }

                // While loop must terminate given
                // (PacketPendingMask & (((1 << pGroup->RRCount) - 1) << pGroup->SPCount))
            }

            // If the next queue has WRR credit and packets, then schedule a packet
            if( (pGroupProc->WrrCreditMask & PacketPendingMask) & (1<<pGroupProc->NextQueue) )
            {
                // Attempt to schedule a packet
                BytesUsed = QosQueueScheduler( pPortCfg, &pGroupProc->Queue[pGroupProc->NextQueue] );
                // If 0x40000000, will "fall off" in shift below
                // Deduct the WRR credit
                if( pPortCfg->fByteWrrCredits )

```

```
        pGroupProc->Queue[pGroupProc->NextQueue].WrrCurrentCredit -= (BytesUsed +
pPortCfg->OverheadBytes - pPortCfg->RemoveBytes) << QMSS_QOS_WRR_BYTES_SCALE_SHIFT;
        else
            pGroupProc->Queue[pGroupProc->NextQueue].WrrCurrentCredit -= 1 <<
QMSS_QOS_WRR_PACKETS_SCALE_SHIFT;

        // Clear the queues's WRR credit mask if we depleted the WRR credit
        if( pGroupProc->Queue[pGroupProc->NextQueue].WrrCurrentCredit <= 0 )
            pGroupProc->WrrCreditMask &= ~(1<<pGroupProc->NextQueue);

        packetSent = 1;
    }

    // Move on to the next group
    pGroupProc->NextQueue++;
    if( pGroupProc->NextQueue == pGroupCfg->SPCount+pGroupCfg->RRCount )
        pGroupProc->NextQueue = pGroupCfg->SPCount;

    // Quit now if we moved a packet
    if(packetSent)
        return(BytesUsed);
    }
}

//
// Finally, try to get a packet from the OPTIONAL best effort queues
//
for( i=pGroupCfg->SPCount+pGroupCfg->RRCount; i<pGroupCfg->QueueCount; i++ )
{
    if( PacketPendingMask & (1<<i) )
        return( QosQueueScheduler(pPortCfg, &pGroupProc->Queue[i] ));
}

// No packet was transferred
return(0);
}
```

3.1.5 Queue Scheduler Pseudocode

```
//
// This is the function that moves a packet from the QOS queue to the egress
//
int32_t QosQueueScheduler(PHYS_PORT_CFG *pPortCfg, QOSQUEUE_PROC *pQueueProc)
{
    int32_t    ByteSize;

    ByteSize = TransferPacket( pPortCfg->DestQueueNumber, pQueueProc->QueueNumber );

    if (ByteSize != -1)
    {
        pQueueProc->PacketsForwarded += 1;
        pQueueProc->BytesForwarded += ByteSize;
        return(ByteSize | 0x40000000);
    }
    return(0);
}
```

3.1.6 Drop Scheduler Pseudocode

```
// PDSP can do this in 1 cycle
uint32_t lmbd32 (uint32_t val, int bit)
{
    int i;

    for (i = 31; i >= 0; i--)
    {
        if (!((val >> i) ^ (bit & 1)))
        {
            return (uint32_t)i;
        }
    }
    return 32;
}
```

```

void DropSchedSnapQueue (
    uint16_t  thisQueueNum,
    uint8_t   *depth_p,
    uint16_t  *depth_b32
)
{
    uint32_t bytes = QueueByteLength (thisQueueNum);
    uint32_t packets = QueuePacketCount (thisQueueNum);
    if (packets > 255)
    {
        packets = 255;
    }
    // Scaled such that 255 8K packets doesn't overflow
    bytes = ((bytes + 31) >> 5);
    if (bytes > 65535)
    {
        bytes = 65535;
    }
    If (depth_b32)
    {
        *depth_b32 = (uint16_t)bytes;
    }
    *depth_p = (uint8_t)packets;
}

// Snapshot input queue depth and assign to output profile
void DropSchedSnapInput (
    DROP_SCHED *dSched,
    uint8_t     *depth_p,
    uint32_t     *packets_present
)
{
    int         bf;
    uint16_t queueBlockIdx = 0;

    // Read qpend bits for each of the queues
    for (bf = 0; bf < (NUM_DROP_INPUT_QUEUES+31)/32; bf++)
    {
        uint32_t pending = ReadQosQueuePendingBits(dSched->BaseQueue + queueBlockIdx);
        uint32_t lmbdval;
        if ( (NUM_DROP_INPUT_QUEUES - queueBlockIdx) < 32 )
        {
            // Ignore unintended queues
            PacketPendingMask &= (1 << (NUM_DROP_INPUT_QUEUES - queueBlockIdx + 1)) - 1;
        }

        packets_present[bf] = pending;
        while ((lmbdval = lmbd32(pending, 1)) < 32)
        {
            uint16_t thisQueueIdx = queueBlockIdx + lmbdval;
            uint16_t thisQueueNum = thisQueueIdx + dSched->BaseQueue;
            if (dSched->Queues[thisQueueIdx].fEnabled)
            {
                DropSchedSnapQueue (thisQueueNum, depth_p + thisQueueIdx, NULL);
            }
            pending &= ~(1 << lmbdval);
        }
        queueBlockIdx += 32;
    }
}

void DropSchedSnapOutput (
    DROP_SCHED *dSched,
    uint8_t     *depth_p,
    uint16_t     *depth_b32
)
{
    uint8_t thisOutProfIdx;
    for (thisOutProfIdx = 0; thisOutProfIdx < NUM_DROP_OUTPUT_PROFILES; thisOutProfIdx++)
    {
        if (dSched->OutProfiles[thisOutProfIdx].fEnabled)
        {

```

```

        uint16_t thisQueueNum = dSched->OutProfiles[thisOutProfIdx].DestQueueNumber;
        DropSchedSnapQueue (thisQueueNum, depth_p + thisOutProfIdx, depth_b32 + thisOutProfIdx);
    }
}

// Discard disabled queues
void DropSchedDropDisabled (
    DROP_SCHED *dSched,
    uint32_t *packets_present
)
{
    int bf;
    uint16_t queueNum = dSched->BaseQueue;
    for (bf = 0; bf < (NUM_DROP_INPUT_QUEUES + 31) / 32; bf++)
    {
        uint32_t lmbdval;
        uint32_t pending = packets_present[bf];
        uint32_t enabled = dSched->queEnBits[bf];
        uint32_t disabled = ~enabled;
        uint32_t pendingAndDisabled = pending & disabled;
        uint32_t updatedPending = pending & ~pendingAndDisabled;
        packets_present[bf] = updatedPending;
        while ((lmbdval = lmbd32(pendingAndDisabled, 1)) < 32)
        {
            uint32_t thisQueueNum = queueNum + lmbdval;

            /* Drop without stats */
            while(DropPacket (thisQueueNum));

            pendingAndDisabled &= ~(1 << lmbdval);
        }
        queueNum += 32;
    }
}

// This is from http://www.iro.umontreal.ca/~lecuyer/myftp/papers/tausme.ps
uint32_t taus88 (uint32_t *seeds)
{
    uint32_t b;
    b = (((seeds[0] << 13) ^ seeds[0]) >> 19);
    seeds[0] = (((seeds[0] & 4294967294u) << 12) ^ b);
    b = (((seeds[1] << 2) ^ seeds[1]) >> 25);
    seeds[1] = (((seeds[1] & 4294967288u) << 4) ^ b);
    b = (((seeds[2] << 3) ^ seeds[2]) >> 11);
    seeds[2] = (((seeds[2] & 4294967280u) << 17) ^ b);
    return (seeds[0] ^ seeds[1] ^ seeds[2]);
}

// Return a draw between 0 and 1 in Q16
uint16_t Rand16 (DROP_SCHED *dSched)
{
    static uint32_t last_rand;
    static int pos = 0;
    uint16_t this_result;

    if (pos < 12)
    {
        last_rand = taus88 (&dSched->rng_s1);
        pos = 32;
    }

    this_result = (uint16_t)(last_rand << 4);
    last_rand >>= 8;
    pos -= 8;

    return this_result;
}

uint16_t mulProb (uint32_t dif, uint32_t recip, uint16_t prob)
{
    uint32_t lmbdval;
    uint32_t res32;

```

```

uint32_t a,b;
uint64_t res64;

// Calculates
// res32 = dif*recip
// res64 = res32*prob
// return res64 >> 32
//
if (dif > recip)
{
    a = recip;
    b = dif;
}
else
{
    a = dif;
    b = recip;
}

res32 = 0;
while ( (lmbdval = lmbd32 (a, 1) ) != 32 )
{
    res32 += b << lmbdval;
    a &= ~ (1<<lmbdval);
}
res64 = 0;
while ( (lmbdval = lmbd32 (prob, 1)) != 32)
{
    res64 += ((uint64_t)res32) << lmbdval;
    prob &= ~ (1<<lmbdval);
}

return (uint16_t)(res64 >> 32);
}

void DropSchedSched (
    DROP_SCHED *dSched,
    uint8_t     in_depth_p,
    uint8_t     out_depth_p,
    uint16_t    out_depth_b32,
    uint32_t    in_packets_present
)
{
    int         que;
    int         bf;
    uint16_t    queueBlockIdx = 0;
    uint16_t    dropProb[NUM_DROP_OUTPUT_PROFILES];
    uint8_t     thisOutProfIdx;

    // Step through each pending bitfield and process all
    // queues with input packets
    for (bf = 0; bf < (NUM_DROP_INPUT_QUEUES + 31) / 32; bf++)
    {
        uint32_t lmbdval;
        uint32_t pending = in_packets_present[bf];
        int needInt = 0;
        while ((lmbdval = lmbd32(pending, 1)) < 32)
        {
            uint16_t thisQueueIdx = queueBlockIdx + lmbdval;
            uint16_t thisQueueNum = thisQueueIdx + dSched->BaseQueue;
            uint8_t thisStatsIdx = dSched->Queues[thisQueueIdx].StatsBlockIdx;
            uint8_t thisOutIdx = dSched->Queues[thisQueueIdx].OutProfIdx;
            uint8_t thisProfIdx = dSched->OutProfiles[thisOutIdx].CfgProfIdx;
            uint8_t fwdPkts;
            for (fwdPkts = in_depth_p[thisQueueIdx]; fwdPkts; fwdPkts--)
            {
                // Don't need to check enable since in_packets_present already
                // compensated for disabled queues
                bool    dropPacket = 0;

                // Tail drop block
                if (dSched->CfgProfiles[thisProfIdx].TailThresh)
                {

```

```

        if (dSched->CfgProfiles[thisProfIdx].fByteTailThresh)
        {
            if ((out_depth_b32[thisOutIdx] << 5) >=
                dSched->CfgProfiles[thisProfIdx].TailThresh)
            {
                dropPacket = 1;
            }
        }
        else
        {
            if (out_depth_p[thisOutIdx] >=
                dSched->CfgProfiles[thisProfIdx].TailThresh)
            {
                dropPacket = 1;
            }
        }
    }

    // RED drop block
    if (!dropPacket && (dSched->CfgProfiles[thisProfIdx].Mode != DROP_MODE_TAIL_ONLY))
    {
        if (dropProb[thisOutIdx] == 0xffff)
        {
            dropPacket = 1;
        }
        else if (dropProb[thisOutIdx] == 0)
        {
            dropPacket = 0;
        }
        else
        {
            if (dropProb[thisOutIdx] <= Rand16(dSched))
            {
                dropPacket = 1;
            }
            else
            {
                dropPacket = 0;
            }
        }
    }

    // Execute drop block
    if (dropPacket)
    {
        // drop 1 packet and count it
        uint32_t bytes = DropPacket (thisQueueNum);
        dSched->StatsBlocks[thisStatsIdx].PacketsDropped ++;
        needInt |= dSched->StatsBlocks[thisStatsIdx].PacketsDropped >> 31;
        dSched->StatsBlocks[thisStatsIdx].BytesDropped += bytes;
        needInt |= dSched->StatsBlocks[thisStatsIdx].BytesDropped >> 31;
    }
    else
    {
        // Forward 1 packet and count it
        uint32_t bytes = TransferPacket(dSched->OutProfiles[thisOutIdx].DestQueueNumber,
thisQueueNum);

        dSched->StatsBlocks[thisStatsIdx].PacketsForwarded ++;
        needInt |= dSched->StatsBlocks[thisStatsIdx].PacketsForwarded >> 31;
        dSched->StatsBlocks[thisStatsIdx].BytesForwarded += bytes;
        needInt |= dSched->StatsBlocks[thisStatsIdx].BytesForwarded >> 31;

        // Update output queue depth so tail drop "sees it".
        out_depth_b32[thisQueueIdx] += (bytes + 31) >> 5;
        out_depth_p[thisQueueIdx] ++;
    }

    if (needInt)
    {
        GenInt (dSched, thisProfIdx);
    }
}

pending &= ~(1 << lmbdval);

```

```

    }
    queueBlockIdx += 32;
}
// Update averages every time, so 0's propagate and compute drop prob
for (thisOutProfIdx = 0; thisOutProfIdx < NUM_DROP_OUTPUT_PROFILES; thisOutProfIdx++)
{
    if (dSched->OutProfiles[thisOutProfIdx].fEnabled)
    {
        // Find new average, presuming all the packets are forwarded
        uint32_t QAvg = dSched->OutProfiles[thisOutProfIdx].QAvg;
        // bytes_pend = in / 2 + out;
        uint32_t bytes_pend = (out_depth_b32[thisOutProfIdx] << 5);
        uint16_t thisProb;
        uint8_t thisCfgProfIdx = dSched->OutProfiles[thisOutProfIdx].CfgProfIdx;
        DROP_CFG_PROFILE *cfg_p = &dSched->CfgProfiles[thisCfgProfIdx];
        DROP_OUTPUT_PROFILE *out_p = &dSched->OutProfiles[thisOutProfIdx];

        if (cfg_p->Mode != DROP_MODE_TAIL_ONLY)
        {
            QAvg += bytes_pend - (QAvg >> cfg_p->TC);
            dSched->OutProfiles[thisOutProfIdx].QAvg = QAvg;

            // Determine drop probability
            if (QAvg <= cfg_p->RedThreshLow)
            {
                thisProb = 0;
            }
            else if (QAvg < cfg_p->RedThreshHigh)
            {
                uint32_t threshDiff = (QAvg - cfg_p->RedThreshLow) >> cfg_p->TC;
                thisProb = mulProb (threshDiff, cfg_p->RedHighMLowRecip, out_p->RedProb);
            }
            else
            {
                thisProb = 0xffff;
            }
            dropProb[que] = thisProb;
        }
    }
}

// Drop Scheduler Main
void DropScheduler (DROP_SCHED *dSched)
{
    uint8_t in_depth_p[NUM_DROP_INPUT_QUEUES]; // input packets
    uint32_t in_packets_present[(NUM_DROP_INPUT_QUEUES + 31)/32];
    uint8_t out_depth_p[NUM_DROP_OUTPUT_PROFILES];
    uint16_t out_depth_b32[NUM_DROP_OUTPUT_PROFILES];

    // Snapshot instantaneous queue depth of input queues.
    // If there are more than 255 packets, then only 255 are processed this tick
    DropSchedSnapInput (dSched, in_depth_p, in_packets_present);
    DropSchedSnapOutput (dSched, out_depth_p, out_depth_b32);

    DropSchedDropDisabled (dSched, in_packets_present);

    DropSchedSched (dSched, in_depth_p, out_depth_p, out_depth_b32, in_packets_present);
}

```

3.1.7 Background Task (Congestion) Pseudocode

```

//
// This is the background task that checks for queue congestion. Note that it is not
// run to completion, but constantly yields to the ForegroundTask() function.
//
{
    int32_t   ByteSize;
    uint32_t  i,j,k;
    PHYS_PORT_CFG *PortsCfg  = sched->PortsCfg;
    PHYS_PORT_PROC *PortsProc = sched->PortsProc;

```

```

// Do this forever
while(1)
{
    // Look at enabled Physical Ports
    for(i=0; i<sched->PortCount; i++)
    {
        if (PortsProc[i].fEnabled)
        {
            // Look at all groups within a port
            for(j=0; j<PortsCfg[i].GroupCount; j++)
            {
                // Look at all queues without a group
                for( k=0; k<PortsCfg[i].Group[j].QueueCount; k++ )
                {
                    if( PortsCfg[i].Group[j].Queue[k].CongestionThresh > 0 )
                    {
                        while( getQueueLength (PortsProc[i].Group[j].Queue[k].QueueNumber,
PortsCfg[i].fByteCongest) >
                                PortsCfg[i].Group[j].Queue[k].CongestionThresh )
                        {
                            ByteSize = DropPacket( PortsProc[i].Group[j].Queue[k].QueueNumber);
                            PortsProc[i].Group[j].Queue[k].PacketsDropped += 1;
                            PortsProc[i].Group[j].Queue[k].BytesDropped += ByteSize;
                            PollProxy();
                            if (timerExpired()) // this costs 1 PDSP cycle if timer didn't
expire
                                {
                                    clearTimer();
                                    sched->TimerTicks++;
                                    ForegroundTask(sched);
                                }
                        }
                    }
                }
            }
        }
    }

    PollProxy();
    if (timerExpired()) // this costs 1 PDSP cycle if timer didn't expire
    {
        clearTimer();
        sched->TimerTicks++;
        ForegroundTask(sched);
    }
}

```

3.2 QoS Scheduler Shadow Configuration Specification

The QoS Scheduler is configured using messages defined in section 4, and a shadow configuration region as defined below. They directly map to the physical port, group, and queue schedulers summarized in the pseudocode data structure in section 3.1.1.

3.2.1 QoS Scheduler Queue

Each queue has a weighted round robin credit which is the weight used to schedule the WRR queues within the group. This value is not used for strict priority and best effort queues. The congestion threshold is also specified per queue and is used for all enabled queues.

Offset	Byte Field			
	Byte 3	Byte 2	Byte 1	Byte 0
0x0000	WRR Initial Credit (for queue in group)			
0x0004	Congestion Threshold			

3.2.1.1 QoS Queue Record Fields

Name	Description
WRR Initial Credit	Each time all the WRR credits are consumed, they are redistributed using this value. The units are bytes << 8, or packets << 17, depending on the value of fWrrlsBytes in the unit flags of the port. This value is only used for queues that are WRR queues within the group.
Congestion Threshold	When the background task detects more than this amount of bytes or packets as specified by fConglsBytes in the unit flags of the port, excess packets will be dropped from head of queue. A value of 0 disabled congestion dropping.

3.2.2 QoS Scheduler Group (Bytes or Packets)

Each group has a CIR credit, a PIR credit as well as accumulated CIR/PIR maximums. A WRR credit is specified to allow the groups to be scheduled in a WRR fashion within the port. Finally the breakdown of SP, WRR, and BE queues is specified.

Offset	Byte Field			
	Byte 3	Byte 2	Byte 1	Byte 0
0x0000	CIR Iteration Credit			
0x0004	PIR Iteration Credit			
0x0008	Maximum accumulated CIR			
0x000C	Maximum accumulated PIR			
0x0010	WRR Initial Credit (for group in port)			
0x0014	Reserved	WRR Queue Cnt	SP Queue Cnt	Total Queue Cnt

3.2.2.1 QOS Queue Record Fields

Name	Description
CIR Iteration Credit	Committed Information Rate credit granted to the group for each timer interval. The units are either packets << 20, or bytes << 11, as specified by fCirlsBytes in the unit flags of the port. This value isn't used on the lite ports defined together with the drop scheduler.
PIR Iteration Credit	Peak Information Rate credit granted to the group for each timer interval. The units are either packets << 20, or bytes << 11, as specified by fCirlsBytes in the unit flags of the port. This value isn't used on the lite ports defined together with the drop scheduler.
Maximum accumulated CIR	Limit on CIR credit for the group in the same units as CIR Iteration Credit. This value isn't used on the lite ports defined together with the drop scheduler.

Maximum accumulated PIR	Limit on PIR credit for the group in the same units as PIR Iteration Credit. This value isn't used on the lite ports defined together with the drop scheduler.
WRR Initial Credit	Each time all the group WRR credits within a port are consumed, they are redistributed using this value. The units are bytes << 8, or packets << 17, depending on the value of as fWrrlsBytes in the unit flags of the port. This value is only used for queues that are WRR queues within the group. This value isn't used on the lite ports defined together with the drop scheduler.
WRR Queue Cnt	Number of WRR queues in the group. These are the "middle" queues.
SP Queue Cnt	Number of SP queues in the group. These are the first queues.
Total Queue Cnt	Total number of queues. BE queues = Total Queue Cnt – WRR Queue Cnt – SP Queue Cnt

3.2.3 QoS Scheduler Group (Bytes And Packets)

This format is used to build that supports simultaneous bytes and packets (build containing 2 full ports and 10 lite ports)

Offset	Byte Field			
	Byte 3	Byte 2	Byte 1	Byte 0
0x0000	CIR Iteration Credit (bytes)			
0x0004	PIR Iteration Credit (bytes)			
0x0008	Maximum accumulated CIR (bytes)			
0x000C	Maximum accumulated PIR (bytes)			
0x0010	WRR Initial Credit (for group in port)			
0x0014	Flags	WRR Queue Cnt	SP Queue Cnt	Total Queue Cnt
0x0018	CIR Iteration Credit (packets)			
0x001c	PIR Iteration Credit (packets)			
0x0020	Maximum accumulated CIR (packets)			
0x0024	Maximum accumulated PIR (packets)			

3.2.3.1 QOS Queue Record Fields

Name	Description
CIR Iteration Credit (bytes)	Committed Information Rate credit granted to the group for each timer interval. The units are bytes << 11. Value is used when flags & 0x20 is set.
PIR Iteration Credit (bytes)	Peak Information Rate credit granted to the group for each timer interval. The units bytes << 11. Value is used when flags & 0x20 is set.
Maximum accumulated CIR (bytes)	Limit on CIR credit for the group in the same units as CIR Iteration Credit. Value is used when flags & 0x20 is set.
Maximum accumulated PIR (bytes)	Limit on PIR credit for the group in the same units as PIR Iteration Credit. Value is used when flags & 0x20 is set.
WRR Initial Credit	Each time all the group WRR credits within a port are consumed, they are redistributed using this value. The units are bytes << 8, or packets << 17, depending on the value of as fWrrlsBytes in the unit flags of the port. This value is only used for queues that are WRR queues within the group.
Flags	0x80: Inherited (sw use in LLD to retain inherited config). Does not affect fw operation; bytes and packets are always selected with 0x40 and 0x20. 0x40: Schedule by packets 0x20: Schedule by bytes
WRR Queue Cnt	Number of WRR queues in the group. These are the "middle" queues.
SP Queue Cnt	Number of SP queues in the group. These are the first queues.
Total Queue Cnt	Total number of queues. BE queues = Total Queue Cnt – WRR Queue Cnt – SP Queue Cnt

CIR Iteration Credit (packets)	Committed Information Rate credit granted to the group for each timer interval. The units are packets << 20. Value is used when flags & 0x40 is set.
PIR Iteration Credit (packets)	Peak Information Rate credit granted to the group for each timer interval. The units are packets << 20. Value is used when flags & 0x40 is set.
Maximum accumulated CIR (packets)	Limit on CIR credit for the group in the same units as CIR Iteration Credit. Value is used when flags & 0x40 is set.
Maximum accumulated PIR (packets)	Limit on PIR credit for the group in the same units as PIR Iteration Credit. Value is used when flags & 0x40 is set.

3.2.4 QoS Scheduler Physical Port (Bytes Or Packets)

Each physical port enables configuration of a port CIR, its egress queue, and specifies the units of all the credit parameters for the port, group, and queues.

Offset	Byte Field			
	Byte 3	Byte 2	Byte 1	Byte 0
0x0000	Egress Queue Number		Group Count	Unit Flags
0x0004	Output Throttle Threshold		Reserved	Overhead Bytes
0x0008	CIR Iteration Credit			
0x000C	Maximum Accumulated CIR			

3.2.4.1 QOS Queue Record Fields

Name	Description
Egress Queue Number	<p>Queue Manager and Queue Number for output queue. This can be any queue supported by the QMSS, whether it is serviced by hardware, host software, or firmware.</p> <p>When chaining output queues to other input queues processed by same QoS PDSP, it is suggested to chain to ascending port numbers. Since ports are processed in ascending order, this reduces unnecessary latency compared to linking to lower numbered ports.</p>
Group Count	Number of groups in use on this port (1-5 for full ports, must be 1 on a lite port)
Unit Flags	<p>0x0001: fWrrlsBytes - WRR credits are specified in bytes 0x0002: fCirlsBytes - CIR credits are specified in bytes 0x0004: fConglsBytes - Congestion Threshold is specified in bytes 0x0008: fByteDestThrottleBytes – Output throttle is specified in bytes 0x0010: flsJoint – combine even/odd lite ports into joint port, Even port is configured where its group WRR Queue Cnt, SP Queue Cnt, and Total queue count are set for both ports. For example, for 2SP, 4WRR, and 2BE queues, these would be set to (4, 2, 8). On the odd port, which must be disabled with flsJointSet, the queue cnts would be set to (2, 0, 4) indicating the portion of queues serviced by the second port. The queues are configured with the first 4 set through the even port, and the last 4 set through the odd port. Statistics are all queried through the even port (for all 8 queues).</p> <p>This is a bit field where 0 or more of the 3 bits can be set.</p>

Output Throttle Threshold	<p>Limit on pending packets in output queue. Once output queue contains this many packets/bytes, no further packets will be forwarded this iteration. This is intended to be used with hierarchical configurations, where only the outermost level should drop. By limiting packets in inner levels, it makes the entire backlog visible to the outer (drop) unit.</p> <p>In order to allow "single path" max rate through the next scheduling block, this throttle should be set to at least the number of bytes that can be scheduled per tick in that block. For example, if QoS runs at 50us, and the next port can forward 100mbit, then this should be set to 5000 bits.</p>
Overhead Bytes	Number of bytes to add to each packet before charging cir/pir. This represents bytes not included in the QMSS C register, such as Ethernet headers and trailers. For example, a common value is 24. These bytes are NOT included in statistics, but are only deducted from credits.
CIR Iteration Credit	Committed Information Rate credit granted to the group for each timer interval. The units are either packets < 20, or bytes < 11, as specified by fCirlsBytes in the unit flags of the port.
Maximum accumulated CIR	Limit on CIR credit for the group in the same units as CIR Iteration Credit.

3.2.5 QoS Scheduler Physical Port (Bytes And Packets)

Each physical port enables configuration of a port CIR, its egress queue, and specifies the units of all the credit parameters for the port, group, and queues.

This is only used with build that supports bytes and packets at the same time.

Offset	Byte Field			
	Byte 3	Byte 2	Byte 1	Byte 0
0x0000	Egress Queue Number		Group Count	Unit Flags
0x0004	Output Throttle Threshold		Reserved	Overhead Bytes
0x0008	CIR Iteration Credit (bytes)			
0x000C	Maximum Accumulated CIR (bytes)			
0x0010	CIR Iteration Credit (packets)			
0x0014	Maximum Accumulated CIR (packets)			

3.2.5.1 QOS Queue Record Fields

Name	Description
Egress Queue Number	<p>Queue Manager and Queue Number for output queue. This can be any queue supported by the QMSS, whether it is serviced by hardware, host software, or firmware.</p> <p>When chaining output queues to other input queues processed by same QoS PDSP, it is suggested to chain to ascending port numbers. Since ports are processed in ascending order, this reduces unnecessary latency compared to linking to lower numbered ports.</p>
Group Count	Number of groups in use on this port (1-5 for full ports, must be 1 on a lite port)

Unit Flags	<p>0x0001: fWrrlsBytes - WRR credits are specified in bytes 0x0002: fCirByBytes – Scheduling using byte credits enabled 0x0004: fConglsBytes - Congestion Threshold is specified in bytes 0x0008: fByteDestThrottleBytes – Output throttle is specified in bytes 0x0010: flsJoint – combine even/odd lite ports into joint port, 0x0020: fCirByPackets – Scheduling using packet credits enabled.</p> <p>Even port is configured where its group WRR Queue Cnt, SP Queue Cnt, and Total queue count are set for both ports. For example, for 2SP, 4WRR, and 2BE queues, these would be set to (4, 2, 8). On the odd port, which must be disabled with flsJointSet, the queue cnts would be set to (2, 0, 4) indicating the portion of queues serviced by the second port. The queues are configured with the first 4 set through the even port, and the last 4 set through the odd port. Statistics are all queried through the even port (for all 8 queues).</p> <p>This is a bit field where 0 or more of the 3 bits can be set.</p>
Output Throttle Threshold	<p>Limit on pending packets in output queue. Once output queue contains this many packets/bytes, no further packets will be forwarded this iteration. This is intended to be used with hierarchical configurations, where only the outermost level should drop. By limiting packets in inner levels, it makes the entire backlog visible to the outer (drop) unit.</p> <p>In order to allow “single path” max rate through the next scheduling block, this throttle should be set to at least the number of bytes that can be scheduled per tick in that block. For example, if QoS runs at 50us, and the next port can forward 100mbit, then this should be set to 5000 bits.</p>
Overhead Bytes	Number of bytes to add to each packet before charging cir/pir. This represents bytes not included in the QMSS C register, such as Ethernet headers and trailers. For example, a common value is 24. These bytes are NOT included in statistics, but are only deducted from credits.
CIR Iteration Credit (bytes)	Committed Information Rate credit granted to the group for each timer interval. The units are bytes << 11. This field is used if UnitFlags.fCirByBytes is set.
Maximum accumulated CIR (bytes)	Limit on CIR credit for the group in the same units as CIR Iteration Credit. This field is used if UnitFlags.fCirByBytes is set.
CIR Iteration Credit (bytes)	Committed Information Rate credit granted to the group for each timer interval. The units are packets << 20. This field is used if UnitFlags.fCirByPackets is set.
Maximum accumulated CIR (bytes)	Limit on CIR credit for the group in the same units as CIR Iteration Credit. This field is used if UnitFlags.fCirByBytes is set.

3.2.6 Complete Shadow Configuration Spec (QoS Scheduler Full/Lite Ports supporting Bytes or Packets)

This table shows the relationship between the port, group, and queue configuration structures to their placement in the memory map shown in “complete shadow configuration area” of section 4.2.1. The same structure format is used for both full and lite ports except that the limit for the number of groups and number of each type of queue is lower on a lite port.

Offset	Size	Structure Definition
0x0000	0x0010	Physical Port Configuration (section 3.2.4)
0x0010	0x0018	Group 0 Configuration (section 3.2.2)
0x0028	0x0008	Group 0, Queue 0 Configuration (section 3.2.1)
0x0030	0x0008	Group 0, Queue 1 Configuration (section 3.2.1)

0x0038	0x0008	Group 0, Queue 2 Configuration (section 3.2.1)
0x0040	0x0008	Group 0, Queue 3 Configuration (section 3.2.1)
0x0048	0x0008	Group 0, Queue 4 Configuration (section 3.2.1)
0x0050	0x0008	Group 0, Queue 5 Configuration (section 3.2.1)
0x0058	0x0008	Group 0, Queue 6 Configuration (section 3.2.1)
0x0060	0x0008	Group 0, Queue 7 Configuration (section 3.2.1)
0x0068	0x0018	Group 1 Configuration (section 3.2.2)
0x0080	0x0040	Group 1, 8 Queue Configurations (section 3.2.1)
0x00C0	0x0018	Group 2 Configuration (section 3.2.2)
0x00D8	0x0040	Group 2, 8 Queue Configurations (section 3.2.1)
0x0118	0x0018	Group 3 Configuration (section 3.2.2)
0x0130	0x0040	Group 3, 8 Queue Configurations (section 3.2.1)
0x0170	0x0018	Group 4 Configuration (section 3.2.2)
0x0188	0x0040	Group 4, 8 Queue Configurations (section 3.2.1)

3.2.7 Complete Shadow Configuration Spec (QoS Scheduler Full/Lite Ports supporting Bytes And Packets)

This table shows the relationship between the port, group, and queue configuration structures to their placement in the memory map shown in “complete shadow configuration area” of section 4.2.1. The same structure format is used for both full and lite ports except that the limit for the number of groups and number of each type of queue is lower on a lite port.

Offset	Size	Structure Definition
0x0000	0x0018	Physical Port Configuration (section 3.2.5)
0x0018	0x0028	Group 0 Configuration (section 3.2.2)
0x0040	0x0008	Group 0, Queue 0 Configuration (section 3.2.1)
0x0048	0x0008	Group 0, Queue 1 Configuration (section 3.2.1)
0x0050	0x0008	Group 0, Queue 2 Configuration (section 3.2.1)
0x0058	0x0008	Group 0, Queue 3 Configuration (section 3.2.1)
0x0060	0x0008	Group 0, Queue 4 Configuration (section 3.2.1)
0x0068	0x0008	Group 0, Queue 5 Configuration (section 3.2.1)
0x0070	0x0008	Group 0, Queue 6 Configuration (section 3.2.1)
0x0078	0x0008	Group 0, Queue 7 Configuration (section 3.2.1)
0x0080	0x0028	Group 1 Configuration (section 3.2.2)
0x00A8	0x0040	Group 1, 8 Queue Configurations (section 3.2.1)
0x00E8	0x0028	Group 2 Configuration (section 3.2.2)
0x0110	0x0040	Group 2, 8 Queue Configurations (section 3.2.1)
0x0150	0x0028	Group 3 Configuration (section 3.2.2)
0x0178	0x0040	Group 3, 8 Queue Configurations (section 3.2.1)
0x01B8	0x0028	Group 4 Configuration (section 3.2.2)
0x01E0	0x0040	Group 4, 8 Queue Configurations (section 3.2.1)

3.2.8 Drop Scheduler Queue Configuration

The shadow area can simultaneously configure all 80 drop scheduler input queues as follows.

Offset	Byte Field			
	Byte 3	Byte 2	Byte 1	Byte 0
0x0000	valid	Stats Push profile index queue 0	Statistics Block Index [0,47] for input	Output Profile Index [0-35] for input queue 0

			queue 0	
0x0004	valid	Stats Push profile index queue 0	Statistics Block Index [0,47] for input queue 0	Output Profile Index [0-35] for input queue 0
0x013C	valid	Stats Push profile index queue 0	Statistics Block Index [0,47] for input queue 79	Output Profile Index [0-35] for input queue 79

3.2.8.1 Drop Scheduler Queue Configuration Fields

There are sufficient rows in the table to simultaneously configure all 80 queues.

Name	Description
Valid	0: profile is invalid/queue is disabled 1: profile is valid/queue is enabled
Stats Push profile index	0: no push stats for this queue 1-4: use stats push profile pair 0-3 from 3.2.10.1. Whenever the MSB of a stat associated with a packet on this queue becomes set, the queue pair will be used to push the stats to host software before they would overflow. The stats internally auto-reset to 0 atomically with the push stats.
Statistics Block Index	0-47: Statistics Block to use for packets forwarded/dropped from this queue The QoS Drop Scheduler supports 48 distinct sets of statistics.
Output Profile Index	0-35:Output Profile The Output Profile specifies the output queue. It also tracks the average of each input+output queue pair. Some implementations call this a "CoS" (class of service).

3.2.9 Drop Scheduler Config Profile Configuration in Shadow

While the drop scheduler supports 80 input queues, they are mapped into CoS (class of service) profiles. Each class of service supports a profile that configures the thresholds and drop probability as well as the egress queue.

Offset	Byte Field			
	Byte 3	Byte 2	Byte 1	Byte 0
0x0000	Reserved	Time Constant	Mode	Unit Flags
0x0004	Tail Drop Threshold			
0x0008	RED low threshold			
0x000C	RED high threshold			
0x0010	Thresh Recip			

3.2.9.1 Drop Scheduler Config Profile Fields

Name	Description
Time Constant	Time constant used for computing average queue depth. This is expressed as a $2^{-(\text{TimeConstant})}$ value. For example for 1/512, use 9.
Mode	0x0000: tail drop only 0x0001: Random Early Drop enabled 0x0002: Random Early Mark enabled
Unit Flags	0x0001: fTailThreshBytes : Tail drop thresholds are in bytes [red thresholds are always bytes]

Tail Drop Threshold	Tail Drop Threshold in fTailThreshBytes units. This is used with instantaneous queue depth. A value of 0 disables tail drop.
RED low threshold	If average depth below this threshold, then no packets are marked/dropped (in fRedThreshBytes units)
RED high threshold	If average depth above this threshold, then all packets are marked/dropped (in fRedThreshBytes units)
Thresh Recip	In Q32, $1/((\text{red high thresh} - \text{red low thresh}) \ll \text{time constant})$

3.2.10 Drop Scheduler Top Level Config in Shadow

The top level configuration for the drop scheduler configures parameters that apply to all of drop scheduler (as well as timer relationship to QoS scheduler)

Offset	Byte Field			
	Byte 3	Byte 2	Byte 1	Byte 0
0x0000	Reserved			
0x0004	Random Seed 1			
0x0008	Random Seed 2			
0x000C	Random Seed 3			
0x0010	Push Stats Source Queue 1		Push Stats Destination Queue 1	
0x0014	Push Stats Source Queue 2		Push Stats Destination Queue 2	
0x0018	Push Stats Source Queue 3		Push Stats Destination Queue 3	
0x001C	Push Stats Source Queue 4		Push Stats Destination Queue 4	

3.2.10.1 Drop Scheduler Top Level Config Fields

Name	Description
Random Seed 1	Used to seed random number generator used to drop/mark packets. Normally this is a don't care. 0 on write means don't change (0 is an illegal configuration for a Tausworthe). Default is 0xfef1. This value will change (representing s1) as RNG runs.
Random Seed 2	Used to seed random number generator used to drop/mark packets. Normally this is a don't care. 0 on write means don't change (0 is an illegal configuration for a Tausworthe). Default is 0xdead. This value will change (representing s2) as RNG runs.
Random Seed 3	Used to seed random number generator used to drop/mark packets. Normally this is a don't care. 0 on write means don't change (0 is an illegal configuration for a Tausworthe). Default is 0xbeef. This value will change (representing s3) as RNG runs.
Push Stats Source Queue N	Queue number to pop a descriptor (32 byte or larger) that will be used to store stats. Stats are directly placed in descriptor, then it is pushed into Push Stats Destination Queue N
Push Stats Destination Queue N	Queue number to send filled stats descriptors to SW.

3.2.11 Drop Scheduler Output Profile Config in Shadow

There are 36 output profiles in the drop scheduler. Each profile specifies an output queue number and tracks the average queue depth between the input and output queues.

Offset	Byte Field
--------	------------

	Byte 3	Byte 2	Byte 1	Byte 0
0x0000	RED drop probability		Output Queue Number	
0x0004	Reserved		Enabled	Config Profile Index
0x0008	Average Queue Depth			

3.2.11.1 Drop Scheduler Output Profile Fields

Name	Description
RED drop probability	Fraction of packets that will be dropped in Q16 units. For example, a value of 0x8000 would set drop probability to 0.5, and 0x51F will set it to 0.02. This is the drop probability when the average queue depth is equal to the RED high threshold.
Output Queue Number	Queue Manager and Queue Number for output queue. This can be any queue supported by the QMSS, whether it is serviced by hardware, host software, or firmware. When chaining output queues to other input queues processed by same QoS PDSP, it is suggested to chain to ascending port numbers. Since ports are processed in ascending order, this reduces unnecessary latency compared to linking to lower numbered ports.
Enabled	0: profile is disabled/not valid 1: profile is enabled/valid
Config Profile Index	Index to threshold defined in section 3.2.9.
Average Queue Depth	Average queue depth, measured over time constant TC. The binary point is at the location configured by scaling factor TC. TC is located in the config profile (see 3.2.9). This can be read for statistics purposes or to compute new RED drop probability if it is too big/small. Value on write isn't used.

3.2.12 Dedicated Query Statistics Shadow Area

The statistics are queried via a message. This ensures they are atomically queried and reset (if the host software were to directly read and reset the stats, packets could be processed between the reads/writes, leading to inconsistent stats). After the message is issued the stats will be in the statistics shadow area in the following format.

Both QoS scheduler and Drop Scheduler can use the Query Statistics. However, the drop scheduler will not report any MSW statistics. Rollover of the Drop Scheduler statistics is handled via Push Statistics (section 3.2.14)

When group statistics are queried by setting queue number to 0xff this area is not used.

Offset	Byte Field			
	Byte 3	Byte 2	Byte 1	Byte 0
0x0000	Bytes Forwarded LSW			
0x0004	Bytes Forwarded MSW			
0x0008	Bytes Discarded LSW			
0x000C	Bytes Discarded MSW			
0x0010	Packets Forwarded			
0x0014	Packets Discarded			

3.2.12.1 Query Statistics Shadow Record Fields

Name	Description
Bytes Forwarded LSW	Bytes forwarded least significant word. Must read LSW and MSW with two separate 32 bit reads, do not issue 64 bit read.

Bytes Forwarded MSW	Bytes forwarded most significant word. Must read LSW and MSW with two separate 32 bit reads, do not issue 64 bit read. Not used for Drop Scheduler statistics (always 0).
Bytes Discarded LSW	Bytes discarded least significant word. Must read LSW and MSW with two separate 32 bit reads, do not issue 64 bit read. This only includes packets which were discarded due to the congestion dropping, not due to port disable.
Bytes Discarded MSW	Bytes discarded most significant word. Must read LSW and MSW with two separate 32 bit reads, do not issue 64 bit read. This only includes packets which were discarded due to the congestion dropping, not due to port disable. Not used for Drop Scheduler statistics (always 0).
Packets Forwarded	Number of packets forwarded to the egress queue.
Packets Discarded	Number of packets discarded. This only includes packets which were discarded due to the congestion dropping, not due to port disable.
Average Queue Depth	

3.2.13 Group Statistics in Common Shadow Area

This is format of stats for entire group when queried with queue=0xff.

Offset	Byte Field			
	Byte 3	Byte 2	Byte 1	Byte 0
0x0000	Reserved			# Of Queues
0x0004	Queue 0 Bytes Forwarded LSW			
0x0008	Queue 0 Bytes Forwarded MSW			
0x000C	Queue 0 Bytes Discarded LSW			
0x0010	Queue 0 Bytes Discarded MSW			
0x0014	Queue 0 Packets Forwarded			
0x0018	Queue 0 Packets Discarded			
0x001c	Queue 1 Bytes Forwarded LSW			
0x0020	Queue 1 Bytes Forwarded MSW			
0x0024	Queue 1 Bytes Discarded LSW			
0x0028	Queue 1 Bytes Discarded MSW			
0x002c	Queue 1 Packets Forwarded			
0x0030	Queue 1 Packets Discarded			
0x0034	Queue 2 Bytes Forwarded LSW			
0x0038	Queue 2 Bytes Forwarded MSW			
0x003c	Queue 2 Bytes Discarded LSW			
0x0040	Queue 2 Bytes Discarded MSW			
0x0044	Queue 2 Packets Forwarded			
0x0048	Queue 2 Packets Discarded			
0x004c	Queue 3 Bytes Forwarded LSW			
0x0050	Queue 3 Bytes Forwarded MSW			
0x0054	Queue 3 Bytes Discarded LSW			
0x0058	Queue 3 Bytes Discarded MSW			
0x005c	Queue 3 Packets Forwarded			
0x0060	Queue 3 Packets Discarded			
0x0064	Queue 4 Bytes Forwarded LSW			
0x0068	Queue 4 Bytes Forwarded MSW			
0x006c	Queue 4 Bytes Discarded LSW			
0x0070	Queue 4 Bytes Discarded MSW			
0x0074	Queue 4 Packets Forwarded			
0x0078	Queue 4 Packets Discarded			
0x007c	Queue 5 Bytes Forwarded LSW			
0x0080	Queue 5 Bytes Forwarded MSW			

0x0084	Queue 5 Bytes Discarded LSW
0x0088	Queue 5 Bytes Discarded MSW
0x008c	Queue 5 Packets Forwarded
0x0090	Queue 5 Packets Discarded
0x0094	Queue 6 Bytes Forwarded LSW
0x0098	Queue 6 Bytes Forwarded MSW
0x009c	Queue 6 Bytes Discarded LSW
0x00a0	Queue 6 Bytes Discarded MSW
0x00a4	Queue 6 Packets Forwarded
0x00a8	Queue 6 Packets Discarded
0x00ac	Queue 7 Bytes Forwarded LSW
0x00b0	Queue 7 Bytes Forwarded MSW
0x00b4	Queue 7 Bytes Discarded LSW
0x00b8	Queue 7 Bytes Discarded MSW
0x00bc	Queue 7 Packets Forwarded
0x00c0	Queue 7 Packets Discarded

3.2.14 Push Statistics

The Drop Scheduler requires the use of Push Statistics. Push Statistics work as follows: if the MSB of one of the statistics becomes set, the firmware will push out the statistics using the Push Stats Destination Queue N in 3.2.11. This section documents the format of those statistics within the descriptor.

Since each of the stats is 32 bits the MSB gets set after either 2G packets or 2Gbytes flow through the queue. At gigabit rate, it takes about 20 minutes for the packet counters to reach this point using minimum size packets (2G/1.5Mbps). It takes about (2G/100MBs) 20 seconds for the byte counters to set the MSB. Once the MSB is set, there is another 20 minutes/ 20 seconds before the counter would roll over. This gives plenty of time for the host to service all the interrupts for 80 queues.

The internal statistics are auto reset when they are copied to the push stats shadow area.

Offset	Byte Field			
	Byte 3	Byte 2	Byte 1	Byte 0
0x0000	Reserved	Reserved	Size	Block Number
0x0000	Bytes Forwarded			
0x0004	Bytes Discarded			
0x0008	Packets Forwarded			
0x000C	Packets Discarded			

3.2.14.1 Push Statistics Descriptor Fields

Name	Description
Size	Size of stats descriptor (actually used). This should be 20.
Block Number	Statistic Block number for associated stats.
Bytes Forwarded	Bytes forwarded.
Bytes Discarded	Bytes discarded or marked for discard eligible. This only includes packets which were discarded due to the congestion dropping, not due to port disable.
Packets Forwarded	Number of packets forwarded to the egress queue.
Packets Discarded	Number of packets discarded. This only includes packets which were discarded due to the congestion dropping, not due to port disable.

3.2.15 Push Proxy

This feature is only implemented together with drop scheduler. The QoS scheduler only build doesn't support this feature. The requests will be forwarded in less than 5us and have an overall throughput >=250K pushes per second.

The host code should use the following pseudocode to do a push:

```
void proxy_push(uint32_t queue, void *ptr, uint32_t size);
{
    lock(); // (stop other cores and threads
    while (*desc_ptr);
    *desc_ptr = ptr;
    queueNum_size = (queue << 16) | size;
    unlock(); // other cores/threads are OK now, they will spin until this push done
}
```

Offset	Byte Field			
	Byte 3	Byte 2	Byte 1	Byte 0
0x0000	queueNum		size	
0x0004	desc_ptr			

3.2.15.1 Push Statistics Descriptor Fields

Name	Description
queueNum	Queue number (0-8191) that proxy will push to (must write atomically with size)
size	Size of packet (to be written to queue's C register) (must write atomically with queueNum)
desc_ptr	Pointer to write to queue's D register. Hint bits can be encoded as needed.

3.2.16 Input Queue Map for QoS Scheduler

The base queue is set with message defined in section 4.1.3.2. The functions of each queue are listed below.

3.2.16.1 Drop Scheduler not present

Queue	Description
0-39	Full port 0
40-79	Full port 1
80-83	Lite port 2
84-87	Lite port 3
88-91	Lite port 4
92-95	Lite port 5
96-99	Lite port 6
100-103	Lite port 7
104-107	Lite port 8
108-111	Lite port 9
112-115	Lite port 10
116-119	Lite port 11

3.2.16.2 Wide Port (Drop Scheduler not present)

Queue	Description
0-135	Full port 0

3.2.16.3 If the Drop Scheduler is present

Queue	Description
0-3	Lite Port 0
4-7	Lite port 1

8-11	Lite port 2
12-15	Lite port 3
16-19	Lite port 4
20-23	Lite port 5
24-27	Lite port 6
28-31	Lite port 7
32-35	Lite port 8
36-39	Lite port 9
40-43	Lite port 10
44-47	Lite port 11
48-51	Lite port 12
52-55	Lite port 13
56-59	Lite port 14
60-63	Lite port 15
64-67	Lite port 16
68-71	Lite port 17
72-75	Lite port 18
76-79	Lite port 19

3.2.17 Input Queue Map for Drop Scheduler

The base queue is set with message defined in section 4.1.3.2. The functions of each queue are listed below.

Queue	Description
0-63	"DSCP" queues
64-72	"PRI" queues
73-80	Linux Queues

4. Firmware Command Interface

4.1 Firmware Command Handshake

4.1.1 Command Handshake

The process of writing a command is to check to see if the command buffer is free, then write the command parameters, and finally write the command. Optionally, the caller can wait for command completion.

The command buffer is free when the “command” field of the first work in the command buffer is set to 0x00.

When a command is written, the host CPU must write the word containing the command byte *last*. The command buffer is in internal RAM and should not be marked as cacheable by the host CPU. If the RAM is cached on the host CPU, then the host must perform two separate writes and cache flushes; the first for writing the parameters, and then a second independent write and cache flush for writing the command word. All writes should be performed as 32 bit quantities.

Note that the first word of the command buffer appears in a noncontiguous memory region as the remaining fields in the buffer.

After the command is written, the PDSP will clear the “command” field upon command completion. The command results can then be read from the Return Code field.

4.1.2 Command Buffer

The session router is programmed using a shared memory command buffer. The command buffer consists of a command word, followed by several parameters. The format of the buffer is as follows:

Command Buffer Address	Field			
	Byte 3	Byte 2	Byte 1	Byte 0
0x000B:C000	Index		Option	Command
0x000B:C004	Return Code			

The following is the breakdown of each field:

Field	Byte Width	Notes
Command	1	QOS Command
Option	1	Command Option
Index	2	Command Index
Return Code	4	Used to return status to the caller: QOSSCHED_CMD_RETCODE_SUCCESS 0x01 QOSSCHED_CMD_RETCODE_INVALID_COMMAND 0x02 QOSSCHED_CMD_RETCODE_INVALID_INDEX 0x03 QOSSCHED_CMD_RETCODE_INVALID_OPTION 0x04

4.1.3 QoS Scheduler Queue Region Base

Egress queues can be located anywhere in the system, but the QoS ingress queues are restricted to a set of 140 starting at a fixed base (which is a multiple of 32). Having a fixed base is not an issue since QoS queues must be allocated out of a general use pool in any case.

4.1.3.1 QOSSCHED_CMD_GET_QUEUE_BASE

The QOSSCHED_CMD_GET_QUEUE_BASE command is used to read the queue number index of the base queue of the QoS scheduler or the Drop Scheduler.

Calling Parameters:

Command	QOSSCHED_CMD_GET_QUEUE_BASE (0x80)
Option	0: Get base queue of the QoS Scheduler (120 if drop scheduler not present, 80 if drop scheduler is present queues) 1: Get base queue of the Drop Scheduler (80 queues)
Index	Not used

Returns:

Index	Queue index of the requested base queue
Return Code	Success or Error Code

4.1.3.2 QOSSCHED_CMD_SET_QUEUE_BASE

The QOSSCHED_CMD_SET_QUEUE_BASE command is used to set the queue number index of the base queue for each of the QoS Scheduler and the Drop Scheduler.

Calling Parameters:

Command	QOSSCHED_CMD_SET_QUEUE_BASE (0x81)
Option	0: Set base queue of the QoS Scheduler (120 if drop scheduler not present, 80 if drop scheduler is present queues) 1: Set base queue of the Drop Scheduler (80 queues)
Index	Queue index of the base queue of the specified QoS queue region. Must be aligned to a multiple of 32 queues.

Returns:

Return Code	Success or Error Code
-------------	-----------------------

4.1.4 Timer Configuration

The PDSP timer determines when credit is passed out. The recommended interval is 100us. If the interval is set too low, the credit “resolution” becomes an issue (you don’t want to doll out one byte at a time), and the firmware performance may not be able to keep up with the interval requested.

The timer is configured by supplying a timer constant. The constant is computed as follows:

$$\text{Constant} = (\text{QMSS_Clock_Frequency} * \text{Desired_Interval}) / 2$$

For example, if the QMSS is running at 350MHz, and the desired credit interval is 100us, the constant value to program would be:

$$\text{Constant} = (350,000,000 * 0.000100) / 2 = 17500$$

4.1.4.1 QOSSCHED_CMD_TIMER_CONFIG

The QOSSCHED_CMD_TIMER_CONFIG command is used to configure QoS credit interval timer.

Calling Parameters:

Command	QOSSCHED_CMD_TIMER_CONFIG (0x82)
Option	not used
Index	Timer Constant

Returns:

Return Code	Success or Error Code
-------------	-----------------------

4.1.5 Enable / Disable QoS Scheduler Physical Port

4.1.5.1 QOSSCHED_CMD_PORT_ENABLE

The QOSSCHED_CMD_PORT_ENABLE command is used to enable or disable a QoS Scheduler physical port. The configuration should be performed by the host before executing this command. All parameters can be changed on a disabled port, while the number of queues and groups should not be changed on a running port (but the credits and units can be changed).

When a port is disabled, all packets on QoS queues contained in that port are discarded.

Calling Parameters:

Command	QOSSCHED_CMD_PORT_ENABLE (0x90)
Option	Set to 1 to enable the port Set to 0 to disable the port
Index	Index is split into a MSB and LSB MSB = 0, LSB=0-19 : Enable/disable QoS scheduler port MSB = 1, LSB=0 : Enable/disable Drop Scheduler

Returns:

Return Code	Success or Error Code
-------------	-----------------------

4.1.6 Copy Configuration To/From Shadow

4.1.6.1 QOSSCHED_CMD_PORT_SHADOW

The QOSSCHED_CMD_PORT_SHADOW command is used copy the configuration between the active area and the shadow area.

Calling Parameters:

Command	QOSSCHED_CMD_PORT_SHADOW (0x91)
Option	Set to 0 to copy from an active port to the shadow area Set to 1 to copy from shadow area to an active (or disabled) port.
Index	Index is split into a MSB and LSB MSB = 0, LSB=0-19 : Copy QoS scheduler port config MSB = 1, LSB=0: Copy drop scheduler profile config MSB = 2, LSB=0 : Copy all drop scheduler queues MSB = 3, LSB=0 : Copy all drop scheduler output config profiles MSB = 4, LSB=0 : Copy drop scheduler top global config

Returns:

Return Code	Success or Error Code
-------------	-----------------------

4.1.7 Stats Request

4.1.7.1 QOSSCHED_CMD_REQ_STATS

The QOSSCHED_CMD_REQ_STATS command is used to atomically copy and optionally reset the stats from a single queue to the statistics shadow area in section 4.2.1.

Calling Parameters:

Command	QOSSCHED_CMD_REQ_STATS (0x92)
Option	Used as a bit field where: 0x0001: reset the forwarded bytes stat 0x0002: reset the forwarded packets stats 0x0004: reset the discarded bytes stats 0x0008: reset the discarded packets stats. 0x0080: request drop scheduler stats instead of QoS scheduler stats
Index	QoS Scheduler: Used as a bit field to index a specific queue when option bit 0x80 is not set. Mapping for QoS scheduler and QoS scheduler + drop scheduler Bits 0-4: physical port Bits 5-7: logical group Bits 8-15: queue within group. Mapping for wide QoS scheduler Bits 0-2: physical port Bits 3-7: logical group Bits 8-15: queue within group. For “wide” QoS scheduler or QoS scheduler without drop scheduler, setting “queue within group” to 0xff will transfer all of the stats for the queues in the group to the shadow area (instead of the stats area). Drop Scheduler: Specifies stats profile 0-47 when option bit 0x80 is set.

Returns:

Return Code	Success or Error Code
	The statistics are copied to the shadow area in section 3.2.12.

4.2 Internal Memory Allocation

4.2.1 PDSP / QMSS Scratch RAM Allocation

The firmware assumes that 8K bytes of RAM are available. No base address is assumed, it is taken from constant register c9 (which is 0xbc000 on keystone 1 devices). The following addresses are relative to that base.

Mapping for QoS Scheduler (Narrow)

Address	Length	Field
0x0000	0x0040	Command Buffer (public)
0x0040	0x0220	Shadow Configuration Area for one port (public)
0x0208	0x0080	-free-
0x02E0	0x0008	Push Proxy (public)
0x02E8	0x0018	-free-
0x0300	0x0020	Statistics shadow area (public)
0x0320	0x0800	Port Configurations (private)
0x0B20	0x0FF0	Port dynamic state (private)
0x1B10	0x02F0	-free-
0x1E00	0x01F8	Scratch (private)
0x1FF8	0x0008	Copy of firmware's version key (public)

QoS Scheduler + Drop Scheduler

Address	Length	Field
0x0000	0x0040	Command Buffer (public)
0x0040	0x01C8	Shadow Configuration Area for one port (public)
0x0208	0x00D8	-free-
0x02E0	0x0008	Push Proxy (public)
0x02E8	0x0018	-free-
0x0300	0x0020	Statistics shadow area (public)
0x0320	0x05E0	Port Configurations (private)
0x0900	0x0D00	Port dynamic state (private)
0x1600	0x0800	-free-
0x1E00	0x01F8	Scratch (private)
0x1FF8	0x0008	Copy of firmware's version key (public)

Mapping for Wide QoS Scheduler

Address	Length	Field
0x0000	0x0040	Command Buffer (public)
0x0040	0x05E8	Shadow Configuration Area for one port (public)
0x0628	0x0018	-free-
0x0640	0x0008	Reserved For Push Proxy
0x0648	0x0018	-free-
0x0660	0x0020	Statistics shadow area (public)
0x0680	0x05E8	Port Configurations (private)
0x0C68	0x1000	Port dynamic state (private)
0x1C68	0x0198	-free-
0x1E00	0x01F8	Scratch (private)
0x1FF8	0x0008	Copy of firmware's version key (public)