MUSIC SEMICONDUCTORS

Epoch API Software Users Manual

Revision 1

December 1, 2000

http://www.music-ic.com

email: info@music-ic.com

Contents

Index

Overview

INTRODUCTION

This document describes how to use the functions in the Epoch Application Programming Interface (API) to implement a wire-speed Router system using the MUSIC Semiconductors Epoch chipset.

Applications

The Epoch chipset allows the construction of a MultiLayer Switch or Router system that performs Layer 3 and Layer 4 routing functions at wire-speed and consists of the following:

- Epoch MultiLayer Switch
- MUSIC MUAC Routing Co-Processor (RCP)
- Generic SRAM and SDRAM

In a system using the Epoch MultiLayer Switch, the Routing Table operations are mostly performed in hardware (using RCPs and SRAM). Although some operations would be performed in software, the RCPs and SRAM would allow the storage of the following:

- IP Unicast, IP Multicast, and IPX entries
- Behavior Aggregate Classification (BAC) and Microflow entries for Layer 4 classification

Features and Benefits

The Epoch API facilitates fast system development by handling complex initialization and Routing Table maintenance routines. Functions are provided with the following capabilities:

- Entries may be searched for, added, and removed
- Users can initialize the necessary pointers for proper system operation

Note: *The customer may use the API code as provided or modify it for use in their Epoch application. There are several patents pending for RCP table maintenance and searching algorithms, however, and the customer is prohibited from using these in non-Epoch applications.*

API FUNCTIONS

The Epoch API provides a comprehensive set of functions. Many are lower-level functions called by the top-level functions. The user may facilitate fast system development by using only the higher-level code. Lower-level code may never be required but is provided as a reference to allow maximum flexibility. Users may develop their own higher-level code by using or altering the lower-level code. Each function is marked to indicate if it is higher- or lower-level. Table 1-1 describes the three files used by the API functions.

Table 1-1: API Function Files

FIle	Description
epoch.h	Structure and enumerated type definitions. Defines for Routing Table manipulation such as RCP op-codes and function declarations.
epochLib.c	Initialization, SRAM read and write functions
epochTable.c	IP Unicast Table, IP Multicast, Layer 4, and IPX functions

RCP ADDRESS SPACE

This document refers to the following expressions when describing the RCP address space:

- Physical address
- Page address
- Virtual address or contiguous virtual address

Physical Address

This is the actual physical address location in a single RCP entry. It ranges from 0 through 4095 for a 4k RCP device and 0 through 8191 for an 8k RCP device. When a result of a search of a device returns an address on the RCP Active Address output bus, it is the physical address of the matching entry. If an address location of a particular device is accessed using the RCP hardware or software mode, as described above, it is this physical address that is required.

Page Address

When multiple RCP devices are cascaded or chained to provide a larger database, each device is given a unique page address value. Usually the values would be 0 through 3 with highest priority device being given the page address 0. When a result of a search of the whole database is given, the page address is given on the RCP Page Address output bus.

Virtual Address or Contiguous Virtual Address

Both of these expressions refer to the address used by the Epoch API. To give the RCP database maximum flexibility, the RCP database may be built from any combination of devices (up to a maximum of four). However, by allowing both 4k and 8k devices to be "mixed and matched", the usual RCP Page Address and Physical Address structure is slightly inadequate. If a combination of 4k and 8k devices are used unaltered, the address values normally used may be misleading and may direct the user to an address location that does not physically exist.

Example: *If a system is using two 4k devices and the user wishes to access RCP physical location 4096, the first RCP may be incorrectly accessed on the assumption that the first device has 8192 locations. The correct action is to access the second device, as its first location is the virtual address 4096. Therefore, the virtual address is the address of any location in a multiple RCP database when it has been translated to one large contiguous block.*

epochRCPInit

The **epochRCPInit** function initializes the RCPs into one contiguous block. When using **epochRCPInit**, the following occurs:

1. The RCPs are initialized with Page Address values and the ***epoch_t*** structure is initialized with the appropriate values required for translating between physical addresses and virtual addresses.
2. When **epochRCPInit** has initialized the database, the other RCP functions operate using contiguous virtual addresses.

Low-Level Functions

The API provides high-level functions that allow initialization and Routing Table operations. There are no functions supplied that perform the low-level access to the Epoch register space. The user *must* provide these functions. The Epoch register space can be either memory-mapped or I/O mapped. Both mapping schemes are supported in the Epoch API and are implemented to allow the user to choose any offset or any number of Epoch devices. Refer to Lower-Level API on page 3-67 for more information on the API low-level functions.

Function Calls

Throughout the API code, the following two function calls are used to access an Epoch register for reading or writing:

```
epochMlsRegRead (pEpoch, regAddress) ;
```

These are the low-level functions used to access the Epoch registers for reading data. The user should provide a function that takes the register address and returns the 32-bit data found in that register. A pointer to the ***epoch_t*** data structure should be passed as an input parameter.

```
epochMlsRegWrite (pEpoch, regAddress, regData) ;
```

These are the low-level functions that access the Epoch registers for writing data. The user should provide a function that takes the 32-data and writes it into the provided register address. A pointer to the ***epoch_t*** data structure should be passed as an input parameter.

DOCUMENT STRUCTURE AND CONVENTIONS

Structure

This section details the contents and organization of each section in this manual.

Table 1-2: Document Contents and Organization

Section	Contents and Organization
Section 1, Overview	This section provides a general overview of the Epoch API including applications, API functions, and RCP address space.
Section 2, Structures	This section describes the seven Epoch API data structures. The syntax and purpose of each structure is described along with any members belonging to that structure. The RCP database and Epoch routing table are also described.
Section 3, Epoch API	This section describes each of the API functions including syntax, input parameters, return values, any structures used, and usage. Each function is grouped with similar functions and is listed in alphabetical order for easy reference. Tables are provided that allow the user to quickly locate any particular function in this manual and in which file each function can be found.
Index	An index listing functions, structures, and other important features.

Conventions

Table 1-3 describes the font conventions used for identification of file types and features.

Table 1-3: Font Conventions

Feature/File Type	Convention	Example
Structure	Helvetica, italic, bold	***rcp_t***
Function	Helvetica, bold	**epochIPUAddEntry**
Block Member	Times New Roman, italic, bold	***ipmblk***
Status Code	Times New Roman, capitals, bold	**RCP_NO_MATCH**
Input Parameters	Times New Roman, italic	**rword*

Structures

INTRODUCTION

The Epoch Application Programming Interface (API) uses seven data structures to store data and pointer information. Table 2-1 shows the seven data structures and their purpose. ***epoch_t*** is the main Epoch API data structure and has the following functions and characteristics:

- Contains all information for maintaining the IP Unicast, IP Multicast, Layer 4 Microflow, and IPX tables. This allows the API to keep track of how the Routing Table is partitioned to store each of the different entry types.
- Stores translation information used when functions have to convert physical addresses to virtual addresses (and vice versa)
- Contains a pointer to the starting address of the memory-mapped or I/O mapped Epoch register space
- Contains initialized fields (by the **epochRCPInit** function) used by Epoch API functions when making the RCP address space contiguous

Most of the Epoch API functions take a pointer to the ***epoch_t*** data structure as an argument. This offers the advantage of portability and the ability to maintain and access several Epoch RCP databases with a relatively small data structure.

Table 2-1: API Data Structures

Structure	Purpose	Page
blockSort_t	Stores RCP page and physical address to virtual address translation data	2-7
blockPtr_t	Stores specific pointer information for the IP Unicast, IP Multicast and IPX blocks	2-10
flowBlock_t	Stores pointer information for the Layer 4 Microflow "parent" and "child" block	2-11
microflow_t	Stores Layer 4 Microflow information	2-12
rcp_t	Stores a 64-bit RCP entry	2-13
rcpResultStatus_t	Stores the result of RCP operations	2-14
epoch_t	Stores block pointer information for the Routing Table set up	2-15

RCP DATABASE

The Epoch Routing Table is built using RCPs and associated data SRAM. The total memory RCP space is divided into the following blocks or partitions for each type of entry:

- IP Unicast
- IP Multicast
- Layer 4 parent
- Layer 4 child
- IPX

The Epoch API uses the ***epoch_t*** and the ***blockPtr_t*** data structures to store the block control information.

RCP Physical Address Translation

To allow the user the flexibility to use any combination of RCP sizes when building the system Routing Table, the API *must* translate normal RCP physical addresses into some form of virtual address scheme. This is because the normal RCP operation produces a large contiguous address block *only* when RCPs of the same size are used. The API provides functions that easily translate between the two formats. The ***blockSort_t*** data structure stores the required data for performing a translation.

Contiguous Virtual Block Formation

The following is a brief explanation of how the Epoch Routing Table is formed into a contiguous virtual block. The Epoch uses a shared RCP address bus to access the RCP memory locations and four chip select outputs to enable it to access individual devices. During the RCP chain configuration process, each RCP is assigned a unique page address starting from 0 and incrementing. In the example shown in Section , the page addresses ranges from 0 through 3. When all 8k-location devices are used, the physical address space is naturally contiguous, and the physical address space is identical to the virtual address space.

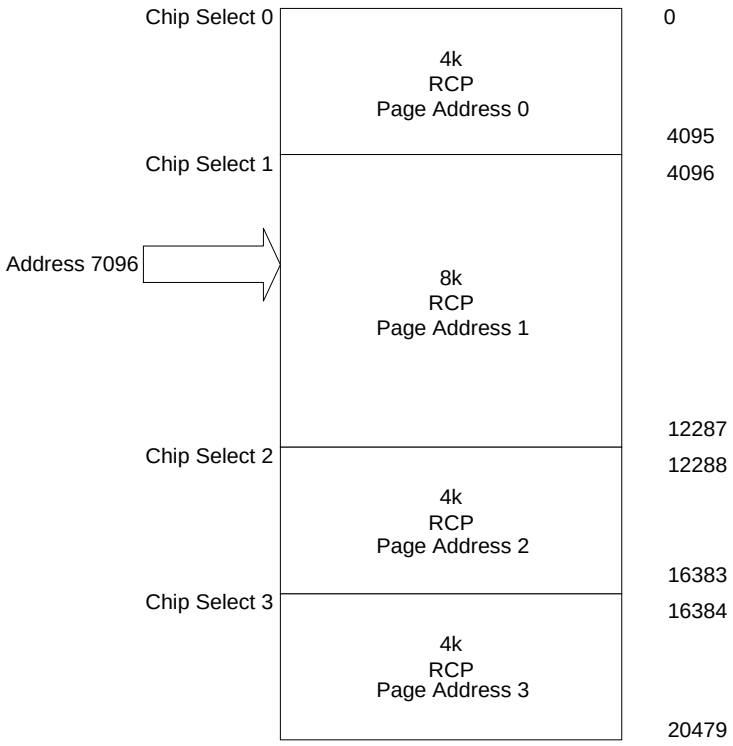

Figure 2-1: RCP Page Address and Virtual Address Example

Non-8K Devices

When all the devices used are not 8k devices, the page address format *must* be translated to the virtual address. This mapping process is handled by the Epoch API functions, making the RCP space transparent to the user. The ***blockSort_t*** structure keeps track of the address information, allowing the use of any combination of RCP size and still use contiguous addresses. The user should use the **epochRCPInit** function. This determines the size of each device used, and automatically initialize the relevant members of the ***blockSort_t*** structure with the correct translation data.

RCP Memory Space

Because the RCP database may not be built using more than four individual devices, the memory space available to the Epoch can range from 4096 locations up to maximum 32,768 locations. Table 2-2 shows the direct relationship between the used RCPs and available memory space. Because of the flexibility allowed by using different size RCP devices in this way, the Epoch is unable to know how many RCPs are used or of what size. Instead, the Epoch API initializes the RCP memory space into one large contiguous block (using the **epochRCPInit** function).

Table 2-2: RCP Memory Space

4096 Location RCPs	8192 Location RCPs	Address Space Available (locations)
0	1	8,192
0	2	16,384
0	3	24,576
0	4	32,768
1	0	4,096
1	1	12,288
1	2	20,480
1	3	28,672
2	0	8,192
2	1	16,384
2	2	24,576
3	0	12,288
3	1	20,480
4	0	16,364

Device Size Tracking

The translation process involves keeping track of the size of each RCP used in the Routing Table and where it is located in the chain. The ***blockSort_t*** structure keeps track of the size of each device so that the device chip selects can be mapped to the appropriate device when the relevant device memory location is accessed.

Example: *Section shows an RCP database that is built using four devices. It is initialized (using* the **epochRCPInit** function*) so that a contiguous block of 20,480 virtual memory locations is available to the user. If a write is attempted to virtual address 7096 (in the second device), the API uses the structure to determine the Page address, Physical address, and chip select number of the location. The structure would contain the information that showed that chip select 1 should be used when a virtual memory location between 4,096 and 12,287 is accessed. It would also show that this virtual address is actually located at Page address 1 and Physical address 3000. However, if the first device had 8,192 locations instead of 4,096 it should be accessed using chip select 0. The Page address would be 0 and the Physical address would be 7096.*

EPOCH ROUTING TABLE

The Epoch System Routing Table consists of the contiguous virtual RCP database (refer to RCP Database on page 2-2) and a 128K-deep SRAM for storing the Port Bitmaps and Layer 4 Flow Handles. The Routing Table may use any number of RCPs up to a maximum of four devices. This gives the system from 4092 to 32,768 entries in which the user partitions into block sizes best suiting the specific application. The associated data is stored in a 16-bit wide SRAM. It *must* be a 128K deep SRAM to accommodate all the possible entries. Section shows how the Routing Table is constructed using RCPs and associated data SRAM.

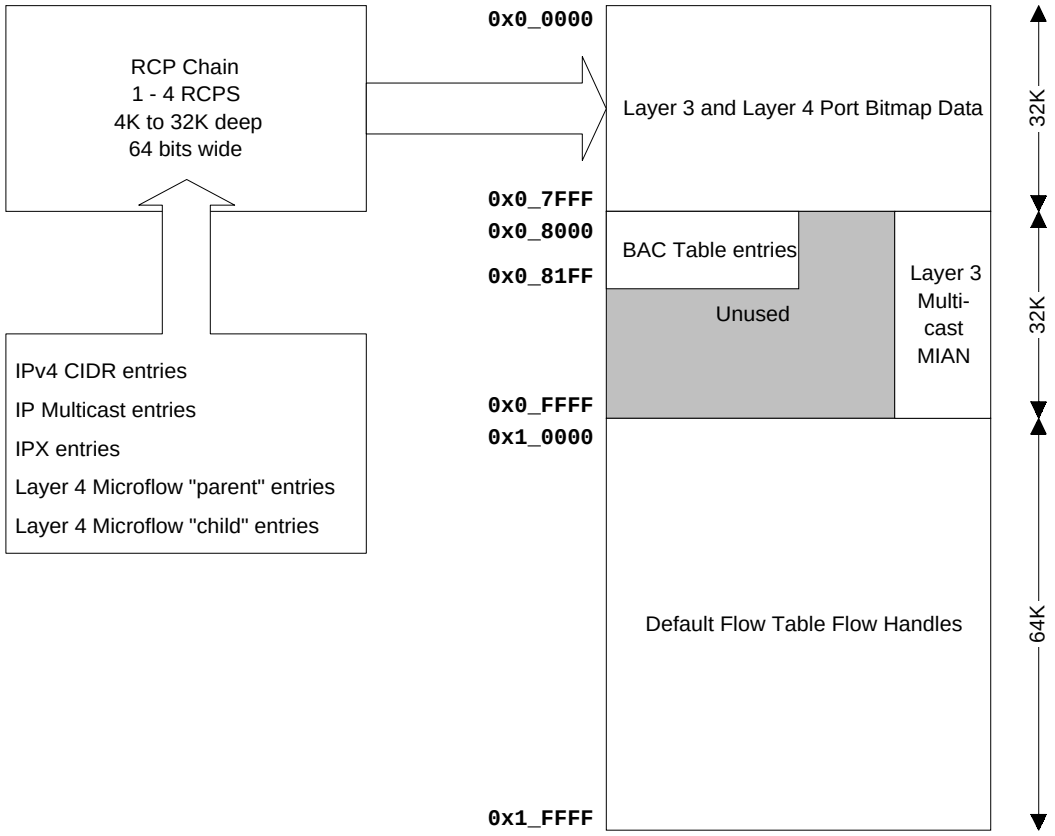

Figure 2-2: Epoch System Routing Table

Layer 3 and Layer 4 Port Bitmap Data

When the system uses the maximum possible 32K of RCP address space, the RCP entries are mapped directly to the corresponding entry in the associated data SRAM. This means that each entry in the RCP database is associated with a corresponding entry in the SRAM in locations 0x0000 through 0x7FFF. The associated data SRAM stores the 16-bit port bitmap value indicating to which physical port a packet should be forwarded. The Page Address and Physical Address of each RCP entry is used as an index to locate each SRAM entry.

Note: *In cases where the user may have a smaller RCP database than the maximum possible size, it is highly recommended that the Routing Table is constructed using the full 128K deep SRAM. This is because the Epoch API uses the Page Address and Physical Address values, which may indicate any possible value in the full 32K-address range depending how many RCPs are used.*

BAC Table Flow Handles

The next 32K deep block of associated data SRAM is the memory locations 0x8000 through 0xFFFF. This block stores the layer 4 BAC Table entries and IP Multicast Interface Authorization Number (MIAN) values. The BAC table block *must* start at memory location 0x8000 and uses 512 locations to store the possible 256 16-bit Flow Handles. Each entry is stored in two 8-bit slices of each individual SRAM location. The BAC Table functions allow the user to add and remove entries by specifying the 8-bit Type of Service (or DS) field that directly relates the Flow Handle. The API positions the BAC Table entries accordingly, so the user does have to place the entries in the appropriate SRAM section. The file "epoch.h" declares the starting address using the line:

```
#define BAC_TABLE_START 0x8000
```

IP Multicast MIAN Values

The IP Multicast block *must* start at location 0x8000. The stored MIAN data is directly related to the location of the IP Multicast entry in the RCP database shifted to start at 0x8000. The provided IP Multicast functions allow the user to add and remove entries by specifying the appropriate IP Multicast data. The API positions the MIAN Table entries accordingly, so the user does not have to place the entries in the appropriate SRAM section. The file "epoch.h" declares the starting address using the line:

```
#define MIAN_START 0x8000
```

Default Flow Table Flow Handles

The last 32K deep block of associated data SRAM (memory location 0x10000 through 0x1FFFF) stores the Layer 4 Default Flow Table entries. The Default Flow Table block *must* start at location 0x10000 and uses 32,768 locations to store all possible 16-bit default Flow Handles. The Layer 4 Microflow functions provided allow the user to add and remove entries by specifying the 16-bit UDP or TCP port number that directly relates to the Flow Handle. The API positions the Default Flow Table entries accordingly, so the user does not have to place the entries in the appropriate SRAM section. The file "epoch.h" declares the starting address using the line:

```
#define DEF_F_START 0x10000L
```

RCP Entries

The RCP database stores the five different entry types. Therefore, the Epoch API partitions the database into four distinct blocks. The Layer 4 Microflow "parent" and "child" entries are both stored in the same block. The four blocks store each of the following entry types:

- Layer 4 Microflow "parent" and "child" entries
- IPv4 CIDR entries
- IP Multicast entries
- IPX entries

RCP Blocks

Full control over how large each block should be allows the user to allocate space best suiting the application. The block partitioning system is allowable due to the encoding method used for each type of entry. Each type of entry has its own distinct encoding method and is described in full in the Epoch data sheet. The only restriction placed on the RCP database is that the user *must* place the block used for the Layer 4 Microflow entries at the beginning of the virtual address space.

Example: *If the total address space available is 4096 entries and each of the four blocks are given 1024 entries, the Layer 4 block must be placed in RCP locations 0 through 1023. The other three blocks may be placed anywhere in the remaining address space.*

The Epoch API provides functions that create the RCP entry from the appropriate information (IP addresses, CIDR mask, IPX net, etc.) and places it in the proper partition of the database. The user *must* initialize the ***epoch_t*** data structure with the relevant information for each block of memory space. The block storing the IPv4 CIDR entries is further divided into 32 blocks that directly relate to the 32 possible CIDR mask values. A function is provided that allocates and partitions the 32 possible IPv4 CIDR blocks.

epoch_t Member Initialization

The ***epoch_t*** data structure *must* be initialized with all relevant information relating to each of the four RCP blocks. The members that require initialization are as follows:

Layer 4 Microflow Block

L4blk. maddr	This should be initialized with the last virtual address of the block that stores the Layer 4 "parent" and "child" entries. The first virtual address of the block does not need to be initialized because it is always location 0.

IPv4 CIDR Block

ipublk[n].floor	This should be initialized with the first virtual address of the block storing the IPv4 CIDR entries relating to the CIDR block specified by **n**. There are 32 possible CIDR blocks from 0 through 31.
ipublk[n].bs	This should be initialized with the block size storing the IPv4 CIDR entries relating to the CIDR block specified by **n**. There are 32 possible CIDR blocks from 0 through 31.
ipublk[n].nextFree	This should be initialized with the first virtual address of the first empty location of the block storing the IPv4 CIDR entries relating to the CIDR block specified by **n**. At initialization, this should be the same as the floor member. There are 32 possible CIDR blocks from 0 through 31.
ipublk[31].threshold	This should be initialized with the last possible virtual address of the complete CIDR area. This may be greater than ipublk.[31].floor + ipublk[31].bs. However, it should never encroach the IP Multicast block or IPX block if they are supported in an application.

IP Multicast Block

ipmblk.floor	This should be initialized with the first virtual address of the block storing the IP Multicast entries.
ipmblk.bs	This should be initialized with the size of the block storing the IP Multicast entries (in RCP entries).
ipmblk.nextFree	This should be initialized with the virtual address of the first empty location of the block storing the IP Multicast entries. At initialization, this should be the same as the floor member.

IPX Block

ipxblk.floor	This should be initialized with the first virtual address of the block storing the IPX entries.
ipxblk.bs	This should be initialized with the size of the block storing the IPX entries (in RCP entries).
ipxblk.nextFree	This should be initialized with the virtual address of the first empty location of the block storing the IPX entries. At initialization, this should be the same as the floor member.

STRUCTURES

blockSort_t

```
#define EPOCH_NUMPAGES 4
typedef struct {
U32 pageSize[EPOCH_NUMPAGES];
U16 pageAddr[EPOCH_NUMPAGES];
U32 chipSelectLkup[EPOCH_NUMPAGES * 2];
U32 pageAddrLkup[EPOCH_NUMPAGES * 2];
U32 pktPtr;
} blockSort_t;
```

Purpose

This structure stores the control information that initializes and maintains the RCP database as one contiguous virtual memory space.

Members

pageSize[EPOCH_NUMPAGES]

This is a four-element array that holds four 32-bit values. It is intended to store the size of each RCP in an RCP chain. The member is used by the address translation functions when determining the appropriate address mask. ***pageSize[0]*** stores the size of the highest priority device, ***pageSize[1]*** holds the second device size and so on. The size should be the number of entries the RCP can hold.

Example: *If the RCP database consists of three 4K devices and one 8K device where the second device is the 8K device, the elements are as follows:*

pageSize[0] = 4096
pageSize[1] = 8192
pageSize[2] = 4096
pageSize[3] = 4096

If the chain has less than four devices, unused elements should contain zero. This member is automatically initialized by the function **epochRCPInit()**.

pageAddr[EPOCH_NUMPAGES]

This is a four-element array that holds four 16-bit values. It is intended to store RCP page address information that is used by the function PAtoA when translating the Physical Address and Page Address given by the RCP database to a contiguous virtual address. They should be initialized to store the virtual page address values for each device using the function ***epochRCPInit()***.

Example: *pageAddr[0] should be initialized with 0, pageAddr[1] should be initialized with the first address of the second device assuming all devices produce a contiguous address space.*

The same action is performed for all devices in the chain. The elements of the array that corresponds to the actual number of devices in the user's application need only be initialized.

Example: *If the application is only using two devices, there is no need to initialize the elements with the indexes 2 and 3. An RCP chain using an 8k device and two 4k devices, with the 8k device being the highest priority device (lowest virtual addresses) would have the elements set as follows:*

pageAddr[0] = 0 (0x0)
pageAddr[1] = 8192 (0x2000)
pageAddr[2] = 12288 (0x3000)
pageAddr[3] = doesn't need to be initialized.

This member is automatically initialized by the functions **epochRCPInitAddrTrans()** and **epochRCPInit()**.

chipSelectLkup[EPOCH_NUMPAGES * 2]

This is an eight-element array that holds eight 32-bit values. It is intended to store the chip select data required for determining which of the four possible chip selects should be used. They should be initialized to store the correct op-code used by the Epoch when accessing an individual RCP using InitRCP. The four possible op-codes that should be stored are:

Chip select for RCP connected to RCPCS20b = 0x0001C000
Chip select for RCP connected to RCPCS21b = 0x0001A000
Chip select for RCP connected to RCPCS22b = 0x00016000
Chip select for RCP connected to RCPCS23b = 0x0000E000

There are two sizes of RCP available: 4k and 8k location devices. Assuming that a system could use any combination of four RCPs to create up to the maximum 32k RCP address space available, every time an RCP virtual address is to be accessed, the chip select to use must be found. Each of the eight array elements correspond to a virtual contiguous address in each of the following ranges:

00000 – 04095 = ***chipSelectLkup[0]***	16384 – 20479 = ***chipSelectLkup[4]***
04096 – 08191 = ***chipSelectLkup[1]***	20480 – 24575 = ***chipSelectLkup[5]***
08192 – 12287 = ***chipSelectLkup[2]***	24474 – 28671 = ***chipSelectLkup[6]***
12288 – 16383 = ***chipSelectLkup[3]***	28672 – 32767 = ***chipSelectLkup[7]***

This element array provides the following functions and characteristics:

- Eight elements correspond to the eight possible changes in virtual address size from 0 through 32k.
- Each element (up to the total address space being used) should be initialized with the appropriate op-code.
- When the Epoch reads ***chipSelectLkup[3]*** to determine which op-code to use for a virtual contiguous address = 14004, it finds the appropriate value (0x1C000, 0x1A000, 0x16000, or 0x0E000). Unused elements (above contiguous address range) should be initialized with 0x0001E000.

This member is initialized automatically by the functions **epochRCPInitAddrTrans()** and **epochRCPInit()**.

pageAddrLkup[EPOCH_NUMPAGES * 2]

This is an eight-element array that holds eight 32-bit values. It is intended to store the page address data required for a virtual address to a RCP Page Address and Physical Address format translation. They should be initialized to store the correct translation for each block of 4096 entries. The translation data required is as follows:

Page Address 0 = 0x00000000
Page Address 1 = 0x00010000
Page Address 2 = 0x00020000
Page Address 3 = 0x00030000

There are 2 sizes of RCP available: 4k and 8k location devices. Assuming that a system could use any combination of four RCPs to create up to the maximum 32k RCP address space available, every time an RCP virtual address is to be provided for translation, we must find the RCP Page Address. Each of the eight array elements correspond to a contiguous virtual address in each of the following ranges:

00000 – 04095 = ***pageAddrLkup[0]***	16384 – 20479 = ***pageAddrLkup[4]***
04096 – 08191 = ***pageAddrLkup[1]***	20480 – 24575 = ***pageAddrLkup[5]***
08192 – 12287 = ***pageAddrLkup[2]***	24474 – 28671 = ***pageAddrLkup[6]***
12288 – 16383 = ***pageAddrLkup[3]***	28672 – 32767 = ***pageAddrLkup[7]***

This element array provides the following functions and characteristics:

- Eight elements correspond to the eight possible changes in virtual address size from 0 through 32k.
- Each element (up to the total address space being used) should be initialized with the appropriate Page Address code.
- The op-code that corresponds to the appropriate Page Address should be placed in each element of the array.

Example: *If the first device is a 4k device, pageAddrLkup[0] is always 0x00000000, pageAddrLkup[1] = 0x0001000. If the first device is an 8k device, pageAddrLkup[1] = 0x00000000. After initialization, the pageAddrLkup[n] elements should contain the Page Address codes that show the appropriate Page Addresses for the contiguous virtual address given. Unused elements (above contiguous address range) do not have to be initialized. The following are two examples of how the elements would be initialized:*

4k, 8k, 8k, 4k devices: *pageAddrLkup[0] = 0x00000, pageAddrLkup[1] = 0x10000, pageAddrLkup[2] = 0x10000, pageAddrLkup[3] = 0x20000, pageAddrLkup[4] = 0x20000, pageAddrLkup[5] = 0x30000.*

8k, 4k, 4k, 4k devices: *pageAddrLkup[0] = 0x00000, pageAddrLkup[1] = 0x00000, pageAddrLkup[2] = 0x10000, pageAddrLkup[3] = 0x20000, pageAddrLkup[4] = 0x30000.*

This member is automatically initialized by the functions **epochRCPInitAddrTrans()** and **epochRCPInit()**.

init_ok

This Flag indicates if the RCP initialization is successful. 1 = successful initialization, 0 = unsuccessful initialization.

blockPtr_t

```
typedef struct {
U16 floor;
U16 nextFree;
U16 bs;
U16 mask;
U16 bkp;
U16 threshold;
}blockPtr_t;
```

Purpose

This structure has the following functions and characteristics:

- Stores the IPv4 CIDR pointers for sorting the table. IPv4 CIDR uses a maximum 32 blocks to store entries of similar network mask values. One block for each mask value from 0x00000000 to 0xFFFFFFFF. The Epoch associates block 0 with network mask = 32, block 1 with mask = 31 and so on.
- Stores pointers for a single block, therefore an array of 32 structures is required for IPv4 CIDR information
- Stores the IPX pointers
- Stores the IP Multicast pointers

Members

floor

This is the pointer to the first RCP location of this block.

nextFree

This is the next free RCP location in this block and is where the next entry may be placed.

bs

This is the block size (the number of RCP entries in this block).

mask

This is the mask value in bits that are set to 1. Used only when the structure stores IPv4 CIDR information.

Example: *Stores 32 if the block is for a 32-bit mask value of all 1s (0xFFFFFFFF). 28 = 0xFFFFFFF0, 24 = 0xFFFFFF00, 23 = 0xFFFFFE00 and so on.*

bkp

This is the BackPressure Flag. This is set to FULL if all blocks below this block are filled and are unable to store any more RCP entries. If there is space in any of the blocks below, it is set to NFULL. Used only when the structure stores IPv4 CIDR information.

threshold

This is the threshold. It is the last RCP address location of the complete IPv4 CIDR partition of the Routing Table. This is required, when blocks are being resized, so that the user can keep track of where the IPv4 blocks end. Used only when the structure stores IPv4 CIDR information, specifically for block 31.

flowBlock_t

```
typedef struct {
U16 maxAddr;
U16 nParent;
U16 nChild;
} flowBlock_t;
```

Purpose

This structure stores the number of "parent" and "child" entries in the Layer 4 Microflow partition of the Routing Table. Every time a "parent" or "child" is removed from the Table, the user should decrement the appropriate member. Alternatively, the member should be incremented if the appropriate entry is added.

Members

maxAddr

This is the maximum address. It is the last virtual RCP address location of the Layer 4 Microflow partition of the Routing Table.

nParent

This is the number of "parent" entries presently stored in the RCP database.

nChild

This is the number of "child" entries presently stored in the RCP database.

microflow_t

```
typedef struct {
U32 sa;
U32 da;
U16 sp;
U16 dp;
U16 ifc;
} microflow_t;
```

Purpose

This structure stores all relevant information that constitutes a Layer 4 Microflow entry in the RCP database.

Members

sa

This is the IP Source Address.

da

This is the IP Destination Address.

sp

This is the TCP or UDP Source Port number.

dp

This is the TCP or UDP Destination Port number.

ifc

This is the physical port number that the flow was received by Epoch and should have a value from 0 through 15.

rcp_t

```
typedef struct {
U32 lo;
U32 hi;
} rcp_t;
```

Purpose

This structure stores a 64-bit RCP entry. The 64-bit entry is split into two 32-bit sections.

Members

lo

This is the 32-bit low word of the 64-bit RCP entry. It relates directly to bits [31:0] of the RCP location.

hi

This is the 32-bit high word of the 64-bit RCP entry. It relates directly to bits [63:32] of the RCP location.

rcpResultStatus_t

```
typedef struct {
U32 addr;
U16 code;
U16 data;
} rcpResultStatus_t;
```

Purpose

Structure used to store the status information regarding RCP and associated SRAM operations. When any RCP or Associated SRAM operation is performed, this structure is updated with the results of an operation.

Members

addr

This is the resulting address of any RCP or SRAM operation that should yield an address. It is stored as a contiguous virtual address.

code

The result of the operation is returned as a status code. The following codes are possible:

RCP_FAIL
The attempt to write an entry to the RCP or associated data SRAM was unsuccessful.

RCP_OK
The attempt to write an entry to the RCP or associated data SRAM was successful.

RCP_CMP_MATCH
The RCP search operation yielded a match.

RCP_NO_MATCH
The RCP search operation did not yield a match.

RCP_CMP_MMATCH
The RCP search operation yielded a multiple-match.

RCP_BLOCK_FULL
An attempt is made to write an entry to an RCP block or partition and it is full.

RCP_OUT_OF_RANGE
An attempt is made to write an entry to an RCP block or partition and the specified block is out of range.

RCP_CHILD_DELETE
The attempt to delete an "child" entry is successful.

RCP_PARENT_DELETE
The attempt to delete an "parent" entry is successful.

RCP_PARENT_CREATED
A "parent" entry was successfully added to the Routing Table.

RCP_CHILD_CREATED
A "child" entry was successfully added to the Routing Table.

data

This is the resulting data of any associated SRAM read.

epoch_t

```
typedef struct {
U32             epochMemBase;
blockSort_t     blockSort;
blockPtr_t      ipmblk;
blockPtr_t      ipxblk;
blockPtr_t      ipublk[32];
flowBlock_t     14blk;
} epoch_t;
```

Purpose

This structure stores the control information used to initialize and maintain the RCP database as one contiguous virtual memory space (***blockSort*** member). It also stores the block pointers for the IP Unicast, IP Multicast, IPX partitions of the database, and the number of Layer 4 Microflow entries.

Members

epochMemBase

This is the Epoch base address. This should be initialized with the base address of where Epoch is located in the system address space. Any register access routines would subsequently use this address.

blockSort

This is a ***blockSort_t*** structure that stores and retrieves the address translation information used to maintain the RCP database.

ipmblk

This is a ***blockPtr_t*** structure that stores and retrieves the block pointer for the IP Multicast partition of the RCP database.

ipxblk

This is a ***blockPtr_t*** structure that stores and retrieves the block pointer for the IPX partition of the RCP database.

ipublk[32]

This is an array of 32 ***blockPtr_t*** structures. Each of the array elements stores and retrieves the block pointers for the IPv4 CIDR block index. For example, if ***ipublk[0]*** is accessed, it should store and retrieve the information for block number 0.

14blk

This is a ***flowBlock_t*** structure that stores and retrieves Layer 4 Microflow information.

Epoch API

INTRODUCTION

The Epoch Application Programming Interface (API) is split into the following seven categories:

- IP Unicast
- IP Multicast
- IPX
- Layer 4
- Initialization
- Lower-level API
- Debug API

The lower-level API functions perform address translation, RCP and SRAM data tasks, and other data manipulation functions used by the higher-level API functions.Refer to

Note: *If the user wishes to perform Routing Table initialization and maintenance, the low-level API functions do not need to be used individually as they are called by the higher-level API.*

Table 3-1 shows each of the Epoch API in alphabetical order. The page where the function is located in the manual is shown. Low- and high-level functions are identified by (H) = high-level, (L) = low-level.

Table 3-1: API Descriptions

API Function Name	Purpose	File Name	Page
AtoCS (L)	Converts an input RCP physical address to the appropriate RCP output Chip Select.	epochLib.c	3-68
AtoPA (L)	Converts an input RCP physical address to the appropriate RCP page address value.	epochLib.c	3-70
DecodeMicroflow (L)	Decodes the Layer 4 “parent” and “child” RCP entries to produce the Microflow information.	epochTable.c	3-39
epochBACReadTableEntry (H)	Reads a BAC Table Flow Handle from the BAC partition of the Routing Table.	epochLib.c	3-55
epochBACWriteTableEntry (H)	Writes a BAC Table Flow Handle into the BAC partition of the Routing Table.	epochLib.c	3-56
epochCheckLeakedPacketPointers (H)	Checks that all Packet pointers are still linked properly.	epochLib.c	3-90
epochIPMAddEntry (H)	Adds an IP Multicast entry to the IP Multicast partition of the Routing Table.	epochTable.c	3-21
epochIPMCreateEntry (L)	Creates the 64-bit RCP entry from the input IP Multicast data (IP SA and DA).	epochTable.c	3-23
epochIPMRemoveEntry (H)	Removes an IP Multicast entry from the IP Multicast partition of the Routing Table.	epochTable.c	3-24
epochIPMSearch (H)	Searches the Routing Table for an IP Multicast entry.	epochTable.c	3-26

Table 3-1: API Descriptions (continued)

API Function Name	Purpose	File Name	Page
epochIPMSearchMian (H)	Searches the Routing Table for an IP Multicast entry and the MIAN.	epochTable.c	3-28
epochIPUAddEntry (H)	Adds an IP Unicast entry to the IP Unicast partition of the Routing Table.	epochTable.c	3-6
epochIPUCreateEntry (L)	Creates the 64-bit ternary RCP entry from the input Unicast IP data (IP DA and mask).	epochTable.c	3-8
epochIPUDumpBlockPointers (H)	Dumps all the IPv4 CIDR block information to a file called "block.dmp".	epochTable.c	3-9
epochIPURemoveEntry (H)	Removes an IP Unicast entry from the IP Unicast partition of the Routing Table.	epochTable.c	3-10
epochIPUSearch (H)	Searches the Routing Table for an IP Unicast entry.	epochTable.c	3-12
epochIPXAddEntry (H)	Adds an IPX entry to the IPX partition of the Routing Table.	epochTable.c	3-31
epochIPXCreateEntry (L)	Creates the 64-bit RCP entry from input IPX net.	epochTable.c	3-33
epochIPXRemoveEntry (H)	Removes an IPX entry from the IPX partition of the Routing Table.	epochTable.c	3-34
epochIPXSearch (H)	Searches the Routing Table for an IPX entry.	epochTable.c	3-36
epochL4FlowAddEntry (H)	Adds a Layer 4 Microflow entry to the Layer 4 (Microflow) partition of the Routing Table.	epochTable.c	3-48
epochL4FlowAge (H)	Ages (or deletes) all Layer 4 entries with the “touch” bit set to 0.	epochTable.c	3-50
epochL4FlowRemoveEntry (H)	Removes a Layer 4 Microflow entry from the Layer 4 (Microflow) partition of the Routing Table.	epochTable.c	3-51
epochL4FlowSearch (H)	Searches the Routing Table for a Layer 4 Microflow.	epochTable.c	3-53
epochPacketPtrClear (L)	Clears the external 64K (65,536) location Packet Pointer SRAM, setting the entries to 0.	epochLib.c	3-66
epochPacketPtrInit (L)	Initializes the external 64K (65,536) location Packet Pointer SRAM.	epochLib.c	3-65
epochPKMInit (H)	Initializes the Packet Manager SRAM. This involves initializing the internal Queue pointers and the external Packet pointers.	epochLib.c	3-58
epochQueuePtrMemInit (L)	Initializes the 128-entry internal Queue Pointer SRAM.	epochLib.c	3-64
epochRCPBinarySearch (L)	Performs a binary search in the RCP for the 64-bit input data.	epochTable.c	3-74
epochRCPDeleteByAddr (L)	Deletes a 64-bit entry from the RCP memory location specified by an address input parameter.	epochLib.c	3-76
epochRCPInit (H)	Initializes the RCP(s).	epochLib.c	3-60

Table 3-1: API Descriptions (continued)

API Function Name	Purpose	File Name	Page
epochRCPInitAddrTrans (L)	Initializes the address translation pointers.	epochLib.c	3-77
epochRCPMoveEntry (L)	Moves a 64-bit RCP entry from a source RCP physical location to a destination location.	epochLib.c	3-79
epochRCPNextFree (L)	Finds the next free RCP location.	epochTable.c	3-80
epochRCPReadEntry (L)	Reads a 64-bit entry from the RCP memory location specified by an address input parameter.	epochLib.c	3-81
epochRCPSRAMRead (L)	Reads the 16-bit RCP associated data SRAM entry specified by an address input parameter. Used for SRAM locations 0xFFFF or less.	epochLib.c	3-82
epochRCPSRAMReadDir (L)	Reads the 16-bit RCP associated data SRAM entry specified by an address input parameter. Used for SRAM locations 0x10000 or greater.	epochLib.c	3-83
epochRCPSRAMWrite (L)	Writes the 16-bit RCP associated data SRAM entry specified by an address input parameter. Used for SRAM locations 0xFFFF or less.	epochLib.c	3-84
epochRCPSRAMWriteDir (L)	Writes the 16-bit RCP associated data SRAM entry specified by an address input parameter. Used for SRAM locations 0x10000 or greater.	epochLib.c	3-85
epochRCPWrite32 (L)	Function for writing instructions in the form of op-code and 32-bit data to RCP.	epochLib.c	3-87
epochRCPWriteEntry (L)	Writes a 64-bit entry into the RCP at the memory location specified by an address input parameter.	epochLib.c	3-86
epochSDRAMInit (H)	Initializes the control SDRAM and tests the data SDRAM.	epochLib.c	3-62
ExtractPind (L)	Extracts the RCP physical address of a Layer 4 “parent” entry from the corresponding “child” entry.	epochTable.c	3-41
FindMask (L)	Finds the IPv4 CIDR mask value from the 64-bit RCP word input parameter.	epochLib.c	3-5
L4CreateChild (L)	Creates the 64-bit RCP “child” entry from an input Microflow and “parent” index.	epochTable.c	3-42
L4CreateParent (L)	Creates the 64-bit RCP “parent” entry from the input Microflow.	epochTable.c	3-44
L4FindSibling (L)	Finds Layer 4 “child” entries that have the same “parent”.	epochTable.c	3-46
PAtoA (L)	Converts an input page address value to the appropriate RCP physical address.	epochLib.c	3-72
SetBackPressure (L)	Sets the BackPressure “flag” for IPv4 block manipulation.	epochTable.c	3-14
SortDown (L)	Recursive sorting function used to insert IP Unicast entry by sorting downwards.	epochTable.c	3-16
SortUp (L)	Recursive sorting function used to insert IP Unicast entry by sorting upwards.	epochTable.c	3-18

IP UNICAST API

The functions in this section are responsible for maintaining the IPv4 CIDR longest match database table (IP Unicast) in the RCP. The IP Unicast table has 32 blocks that represent the 32 possible masks. The user has control over the initial block sizes and start address, subject to the following conditions.

- First Condition–Each block in use *must* have at least one entry reserved at initialization time. This is to ensure that the sorting algorithm, as it is currently implemented, works properly. The worst case causes up to 32 RCP entries to be wasted.
- Second Condition–Blocks are adjacent to each other, i.e., if the last entry in block 2 is at address x, the first entry in block 3 is at address x+1. This condition is beneficial to the user because it ensures that no limited RCP address space is lost.

Table 3-2 shows all the IP Unicast functions, the files in which they may be found and the page in this manual where they are located. Low- and high-level functions are identified by (H) = high-level, (L) = low-level.

Table 3-2: IP Unicast Functions

API Function Name	Purpose	File Name	Page
FindMask (L)	Finds the IPv4 CIDR mask value from the 64-bit RCP word input parameter.	epochLib.c	3-5
epochIPUAddEntry (H)	Adds an IP Unicast entry to the IP Unicast partition of the Routing Table.	epochTable.c	3-6
epochIPUCreateEntry (L)	Creates the 64-bit ternary RCP entry from the input Unicast IP data (IP DA and mask).	epochTable.c	3-8
epochIPUDumpBlockPointers (H)	Dumps all the IPv4 CIDR block information to a file called “block.dmp”.	epochTable.c	3-9
epochIPURemoveEntry (H)	Removes an IP Unicast entry from the IP Unicast partition of the Routing Table.	epochTable.c	3-10
epochIPUSearch (H)	Searches the Routing Table for an IP Unicast entry.	epochTable.c	3-12
SetBackPressure (L)	Sets the BackPressure “flag” for IPv4 block manipulation.	epochTable.c	3-14
SortDown (L)	Recursive sorting function used to insert IP Unicast entry by sorting downwards.	epochTable.c	3-16
SortUp (L)	Recursive sorting function used to insert IP Unicast entry by sorting upwards.	epochTable.c	3-18

FindMask

Syntax

```
U16 FindMask(rcp_t *rword);
```

Description

This function finds the IPv4 CIDR mask value from the input 64-bit RCP word. This function is useful if the user wishes to inspect all entries found in the IPv4 partition of the Routing Table as it allows finding the mask value for each entry located during the inspection.

Input Parameters

**rword*	The 64-bit RCP word that is to be inspected to locate the mask value. The parameter should be a variable in the form of the structure ***rcp_t***. The ***rcp_t*** variable should hold the following information: rcp_t.hi = Bits 63:32 of the RCP entry rcp_t.lo = Bits 31:0 of the RCP entry

Return Value

A 16-bit unsigned integer is returned. This is the mask value of the input ***rcp_t*** variable. It is a value in the range 1 through 32 and signifies the number of contiguous 1s in the IPv4 CIDR mask. For example:

32 = 0xFFFFFFFF

31 = 0xFFFFFFFE

30 = 0xFFFFFFFC

4 = 0xF0000000

3 = 0xE0000000

2 = 0xC0000000

1 = 0x80000000

Pre-requisites

None.

Usage

```
rcp_t rword;
ipaddr_t IP_address = 0xbbccaa03;
U16 mask = 32;
epoch_t pEpoch;
rword = epochIPUCreateEntry(&pEpoch, IP_address, mask);
Printf("\nThe mask value of the entry is: %d", FindMask(&rword));
```

epochIPUAddEntry

Syntax

```
rcpResultStatus_t epochIPUAddEntry(epoch_t *pEpoch,
                                   ipaddr_t addr,
                                   U16 mask,
                                   U16 data);
```

Description

This function adds the input IP address and mask combination to the appropriate IPv4 CIDR block in the Routing Table.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
addr	The 32-bit IP address to be added. This can be any class other than class D. The customer is responsible for ensuring class D addresses are not added to the IP Unicast table.
mask	The mask number of the entry. It should be a number from 1 through 32. The number determines how many contiguous 1 are in the mask that will be used to encode the entry. Please refer to MUSIC Application Note AN-N22 for a full description of how masks are used in adding IPv4 CIDR entries.
data	The 16-bit associated data that will be added to the associated data SRAM. This is the port bitmap that indicates where any corresponding packets should be routed. The appropriate bit is set if the packet is to be routed to the port associated with it. For example, bit 0 = port 0, bit 1 = port 1, bit 2 = port 2 and so on.

Return Value

A variable in the form of the structure ***rcpResultStatus_t*** is returned. This indicates any relevant information regarding the function's success or failure.

If the IPv4 CIDR partition of the Routing Table is full the "code" member is **RCP_FAIL**

Pre-requisites

The IPv4 block structure *must* be initialized prior to using this function.

Usage

```
ipaddr_t IP_address = 0xbbccaa03;
U16 mask = 32, associated_data = 0x0002;
epoch_t pEpoch;
rcpResultStatus_t write_status;
int i_stat;

i_stat = epochRCPInit(&pEpoch);
if(i_stat == INIT_OK){
      write_status = epochIPUAddEntry(&pEpoch, IP_address, mask,
                                      associated_data);
      if(write_status.code & RCP_FAIL){
            printf("\nThere is no memory available for RCP write.");
      }
      else{
            printf("\nThe IP address/mask/associated data was added.");
      }
}
else{
      printf("\nERROR : Unable to initialize RCP chain.");
}
```

epochIPUCreateEntry

Syntax

```
rcp_t epochIPUCreateEntry(epoch_t *pEpoch,
                          ipaddr_t addr,
                          U16 mask);
```

Description

This function creates a 64-bit RCP entry from the input 32-bit IP address and mask number. The RCP entry is returned in the form of the ***rcp_t*** structure. The format of the IPv4 CIDR entry created is in the form shown the Epoch data sheet. The structure member "lo" stores the lower 32-bit value and the member "hi" stores the upper 32-bit value.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
addr	The 32-bit IP address to be added. This can be any class other than class D. The customer is responsible for ensuring class D addresses are not added to the IP Unicast table.
mask	The mask number of the entry. It should be a number from 1 through 32. The number determines how many contiguous 1s are in the mask that will be used to encode the entry. Please refer to MUSIC Application Note AN-N22 for a full description of how masks are used in adding IPv4 CIDR entries.

Return Value

A variable in the form of the structure ***rcp_t*** is returned. This is the RCP entry that is created from the input address and mask values.

rcp_t.hi = Bits 63:32 of the RCP entry.

rcp_t.lo = Bits 31:0 of the RCP entry.

Pre-requisites

None.

Usage

```
rcp_t rword;
ipaddr_t IP_address = 0xbbccaa03;
U16 mask = 32;
epoch_t pEpoch;
rword = epochIPUCreateEntry(&pEpoch, IP_address, mask);
printf("\nAn entry was created. It is: 0x%lx (high word)
        0x%lx (low word)",rword.hi, rword.lo);
```

epochIPUDumpBlockPointers

Syntax

```
void epochIPUDumpBlockPointers(epoch_t *pEpoch);
```

Description

This function writes information regarding the IPv4 block structure to the screen for diagnostic purposes. The function writes the block size (bs), the mask level (or number), the floor RCP virtual address and the next free virtual address for each of the 32 blocks. The status of the BackPressure flag is also written. The function writes in the information that it finds in the ***epoch_t*** data structure.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.

Return Value

None.

Pre-requisites

The IPv4 block structure *must* be initialized prior to using this function. If this is not done, the function writes the pre-initialized data it finds in the ***epoch_t*** data structure.

Usage

```
epoch_t pEpoch;
/*assume block pointers have been intialized*/
epochIPUDumpBlockPointers(&pEpoch);
```

epochIPURemoveEntry

Syntax

```
rcpResultStatus_t epochIPURemoveEntry(epoch_t *pEpoch,
                                      ipaddr_t ipAddr,
                                      U16 mask);
```

Description

This function removes the input IP address and mask combination from the appropriate IPv4 CIDR block in the Routing Table. It also sorts the remaining entries, in order to keep the block contiguous.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
ipAddr	The 32-bit IP address to be deleted. This can be any class other than class D.
mask	The mask number of the entry. It should be a number from 1 through 32. The number determines how many contiguous 1 are in the mask that will be used to locate the entry. Please refer to MUSIC Application Note AN-N22 for a full description of how masks are used in adding IPv4 CIDR entries.

Return Value

A variable in the form of the structure ***rcpResultStatus_t*** is returned. This indicates any relevant information regarding the function's success or failure. The RCP virtual address of where the IPv4 CIDR entry was located (if found) is returned in the "addr" member.

Pre-requisites

None.

Usage

```
ipaddr_t IP_address = 0xbbccaa03;
U16 mask = 32, associated_data = 0x0002;
epoch_t pEpoch;
rcpResultStatus_t write_status , delete_status;
int i_stat;

i_stat = epochRCPInit(&pEpoch);
if(i_stat == INIT_OK){
      write_status = epochIPUAddEntry(&pEpoch, IP_address, mask,
                                      associated_data);
      if(write_status.code & RCP_FAIL){
            printf("\nThere is no memory available for RCP write.");
      }
      else{
            printf("\nThe IP address/mask/associated data was added.");
      }
      delete_status = epochIPURemoveEntry(&pEpoch, IP_address, mask);
      if(delete_status.code & RCP_CMP_NMATCH){
            printf("\nUnable to find IP address in Routing Table.");
      }
      else{
            printf("\nThe IP address was deleted.");
      }
}
else{
      printf("\nERROR : Unable to initialize RCP chain.");
}
```

epochIPUSearch

Syntax

```
rcpResultStatus_t epochIPUSearch(epoch_t *pEpoch,
                                 ipaddr_t ipAddr);
```

Description

This function performs a longest match search for the input IPv4 non-class D address. If the address produces a match during the search, the port bitmap and virtual adjacency pointer/RCP address are returned.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
ipAddr	The 32-bit IP address to be searched for. This can be any class other than class D.

Return Value:

A variable in the form of the structure ***rcpResultStatus_t*** is returned. This indicates any relevant information regarding the function's success or failure.

- **Successful–**The RCP virtual address of where the IPv4 CIDR entry was located (if found) is returned in the "addr" member. The associated data found in the associated data SRAM is returned in the "data" member. **RCP_CMP_MATCH** or **RCP_CMP_MMATCH** are returned in the "code" member to indicate if there was a match or multiple matches.
- **Unsuccessful–RCP_NO_MATCH** is returned in the "code" member to indicate that there was no match found.

Pre-requisites

None.

Usage

```
ipaddr_t IP_address = 0xbbccaa03;
U16 mask = 32, associated_data = 0x0002;
epoch_t pEpoch;
rcpResultStatus_t write_status , search_status;
int i_stat;

i_stat = epochRCPInit(&pEpoch);
if(i_stat == INIT_OK){
      write_status = epochIPUAddEntry(&pEpoch, IP_address, mask,
                                      associated_data);
      if(write_status.code & RCP_FAIL){
            printf("\nThere is no memory available for RCP write.");
      }
      else{
            printf("\nThe IP address/mask/associated data was added.");
      }
      search_status = epochIPUSearch(&pEpoch, IP_address);
      if(search_status.code & RCP_CMP_NMATCH){
            printf("\nUnable to find IP address in Routing Table.");
      }
      else{
            printf("\nThe IP address was found at virtual address
                    0x%x with assoc. data = 0x%lx",
                    search_status.addr, search_status.data);
      }
}
else{
      printf("\nERROR : Unable to initialize RCP chain.");
}
```

SetBackPressure

Syntax

```
static int SetBackPressure(epoch_t *pEpoch);
```

Description

This function sets the BackPressure Flag used in the IPv4 CIDR sorting algorithm. The function checks the state of each CIDR block to determine the status of available free entry locations. It sets the bkp member of each block to NFULL or FULL as it finds if there are entries available in the blocks below. Finally, it returns the status of the complete IPv4 CIDR Table (FULL or NFULL).

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.

Return Value

An integer is returned that indicates if the IPv4 CIDR portion of the Routing Table is completely full and can not accept any more entries.

- **FULL** = All possible Table locations have valid IPv4 CIDR entries.
- **NFULL** = There is at least one empty location.

Pre-requisites

None.

Usage

```
#define L4_STOPADDR 0x1000
#define IPU_STOPADDR 0x3000
#define IPM_STOPADDR 0x3800
#define IPX_STOPADDR 0x3C00
epoch_t pEpoch;
int i_stat, x;
blockPtr_t *bp;
struct {
        U16     size;
        U16     mask;
       } IPUInfo[] = {
     { 200, 32 }, { 100, 31 }, { 50, 30 }, { 25, 29 }, { 20, 28 },
     { 20, 27 }, { 20, 26 }, { 20, 25 }, { 20, 24 }, { 20, 23 },
     { 20, 22 }, { 20, 21 }, { 20, 20 }, { 15, 19 }, { 15, 18 },
     { 15, 17 }, { 15, 16 }, { 15, 15 }, { 15, 14 }, { 15, 13 },
     { 15, 12 }, { 15, 11 }, { 15, 10 }, { 15, 9 }, { 15, 8 },
     { 10, 7 }, { 10, 6 }, { 5, 5 }, { 5, 4 }, { 4, 3 },
     { 2, 2 }, { 2, 1 },
    };

i_stat = epochRCPInit(&pEpoch);
if(i_stat == INIT_OK){
    //initialiaze the IPU block pointers
    for (x = 0; x < 32; x++){
        pEpoch.ipublk[x].bs = IPUInfo[x].size;
        pEpoch.ipublk[x].mask = IPUInfo[x].mask;
        if (x == 0){
            pEpoch.ipublk[x].floor = L4_STOPADDR + 1;
            pEpoch.ipublk[x].nextFree = L4_STOPADDR + 1;
        }
        else{
            bp = &(pEpoch.ipublk[x - 1]);
            pEpoch.ipublk[x].floor = bp->floor + bp->bs;
            pEpoch.ipublk[x].nextFree = bp->floor + bp->bs;
        }
        pEpoch.ipublk[x].bkp = NFULL;
    }
    pEpoch.ipublk[31].threshold = IPU_STOPADDR;
    //initialize other pointers
    pEpoch.ipmblk.floor = IPU_STOPADDR + 1;
    pEpoch.ipmblk.bs = IPM_STOPADDR - IPU_STOPADDR;
    pEpoch.ipmblk.nextFree = pEpoch.ipmblk.floor;
    pEpoch.ipxblk.floor = IPM_STOPADDR + 1;
    pEpoch.ipxblk.bs = IPX_STOPADDR - IPM_STOPADDR;
    pEpoch.ipxblk.nextFree = pEpoch.ipxblk.floor;
}
else{
    printf("\nERROR: Unable to initialize RCP chain.");
}
//now set Back pressure Flags
if(SetBackPressure(&pEpoch) == FULL){
    printf("\nThe IPU partition is FULL.");
}
else{
    printf("\nThe IPU partition is NOT FULL.");
}
```

SortDown

Syntax

```
static RCP_Result_Status SortDown(epoch_t *pEpoch,
                                  U16 ei,
                                  U16 bi);
```

Description

This function uses the IPv4 CIDR sorting algorithm (downward) to allocate a free location in a previously full block. The function recursively checks the next block down to see if it has at least one free location. When it finds a free location it moves the appropriate pointers to create an empty location in the desired block.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.

ei	The last RCP virtual address in the block that the user wishes to free. For example, if the user wishes to place an entry into block 3, but it is full, the user would set the parameter ***ei*** to **`userepoch_t->ipublk[3]->nextFree.`** This assumes that the user has already tested to see if the block is full. This could be done in more that one way. One example would be to compare the next free location in the block with the floor pointer added to the block size:
	`if((userepoch_t->ipublk[3]->floor+userepoch_t->ipublk[3]->bs)`
	`== userepoch_t->ipublk[3]->nextFree)`
	If this comparison is true, then the block is full.

bi	The block from which the user wishes to attempt to free an empty location. This always should be the block where ***ei*** is located plus 1 (one down). For example, if the user wishes to place an entry into block 3, but it were full, the user would set the parameter ***bi*** to 4. The function attempts to free the location from the block specified by the parameter ***bi***. If this block is also full, the function recursively attempts to free a location from the next block down until it is successful.

Return Value

A variable in the form of the structure ***rcpResultStatus_t*** is returned. This indicates any relevant information regarding the function's success or failure.

- **Successful**–The RCP virtual address of where the new location was found is returned in the "addr" member.
- **Unsuccessful**–**RCP_BLOCK_FULL** is returned in the "code" member to indicate that it is not possible to sort down because there are no free entries in or below the block specified in ***bi.*** All of the current IPv4 CIDR block pointers are written to a file called "block.dmp" so the user may inspect them.

Pre-requisites

The user should check the BackPressure Flag of the block specified in ***bi*** to ensure there is at least on free location in or below the block.

Usage

```
#define NUMBLOCKS 32
epoch_t pEpoch;
U16 This_block = 5;
rcpResultStatus_t status;
blockPtr_t *bp;

//set block pointer for block 5
bp = &(pEpoch.ipublk[This_block]);

//assuming the RCP has been initialized and contains IPU entries
if ((bp->floor + bp->bs) == bp->nextFree){
//This Block is FULL, so sort down or sort up
      if (This_block != NUMBLOCKS-1){ /* we are not in last block*/
            if ((bp+1)->bkp == NFULL){ // we can sort down
                  bp->nextFree++; //create the new entry
                  bp->bs++;
                  status = SortDown(&pEpoch, bp->nextFree-1,
                                    This_block+1);
                  //new entry in block 5 created
            }
            else { // we must sort up
                  bp->bs++;
                  bp->floor--;
                  if ((bp-1)->nextFree == bp->floor+1){
                        status = SortUp(&pEpoch, (bp-1)->nextFreee-1,
                                        This_block-1);
                  }
                  else {
                        (bp -1)->bs--;
                  }
            }
      }
}
```

SortUp

Syntax

```
static rcpResultStatus_t SortUp(epoch_t *pEpoch,
                                U16 ei,
                                U16 bi);
```

Description

This function uses the IPv4 CIDR sorting algorithm (upward) to allocate a free location in a previously full block. The function recursively checks the next block up to see if it has at least one free location. Once it finds a free location it moves the appropriate pointers to create an empty location in the desired block.

Input Parameters

pEpoch* — A pointer to the ***epoch_t* structure. This structure *must* contain information about the Epoch and the Routing Table.

ei — The first RCP virtual address in the block that the user wishes to free (floor). For example, if the user wishes to place an entry into block 3, but it is full (and all blocks below are full), the user would set the parameter ***ei*** to:

`userepoch_t->ipublk[3]->floor.` This assumes that the user has already tested to see if the block is full. This could be done in more that one way. One example would be to compare the next free location in the block with the floor pointer added to the block size:

```
if((userepoch_t->ipublk[3]->floor+userepoch_t->ipublk[3]->bs)
== userepoch_t->ipublk[3]->nextFree)
```

If this comparison is true, then the block is full.

bi — The block that the user wishes to attempt to free an empty location from. This should always be the block where ***ei*** is located minus 1 (one up). For example, if the user wishes to place an entry into block 3, but it were full (and all blocks below are full), the user would set the parameter ***bi*** to 2. The function attempts to free the location from the block specified by the parameter ***bi***. If this block is also full, the function recursively attempts to free a location from the next block up until it is successful.

Return Value

A variable in the form of the structure ***rcpResultStatus_t*** is returned. This indicates any relevant information regarding the function's success or failure.

- **Successful–**The RCP virtual address of where the new location was found is returned in the "addr" member.
- **Unsuccessful–RCP_BLOCK_FULL** is returned in the "code" member to indicate that it is not possible to sort up because there are no free entries in or above the block specified in ***bi.*** All of the current IPv4 CIDR block pointers are written to the screen so the user may inspect them.

Pre-requisites

The user should use the function `SetBackPressure()` to check if there is at least one free entry in the Table. It is recommended that the user attempt to sort down first.

Usage

```
#define NUMBLOCKS 32
epoch_t pEpoch;
U16 This_block = 5;
rcpResultStatus_t status;
blockPtr_t *bp;

//set block pointer for block 5
bp = &(pEpoch.ipublk[This_block]);

//assuming the RCP has been initialized and contains IPU entries
if ((bp->floor + bp->bs) == bp->nextFree){
//This Block is FULL, so sort down or sort up
      if (This_block != NUMBLOCKS-1){ /* we are not in last block*/
            if ((bp+1)->bkp == NFULL){ // we can sort down
                  bp->nextFree++; //create the new entry
                  bp->bs++;
                  status = SortDown(&pEpoch, bp->nextFree-1,
                                    This_block+1);
                  //new entry in block 5 created
            }
            else { // we must sort up
                  bp->bs++;
                  bp->floor--;
                  if ((bp-1)->nxtfree == bp->floor+1){
                        status = SortUp(&pEpoch, (bp-1)->nextFree-1,
                                        This_block-1);
                  }
                  else {
                        (bp -1)->bs--;
                  }
            }
      }
}
```

IP MULTICAST API

The functions in this section are responsible for maintaining the IP Multicast database table in the RCP. The files in which they may be found and the page in this manual where they can be located is shown for easy reference. Low- and high-level functions are identified by (H) = high-level, (L) = low-level.

Table 3-3: IP Multicast Functions

API Function Name	Purpose	File Name	Page
epochIPMAddEntry (H)	Adds an IP Multicast entry to the IP Multicast partition of the Routing Table.	epochTable.c	3-21
epochIPMCreateEntry (L)	Creates the 64-bit RCP entry from the input IP Multicast data (IP SA and DA).	epochTable.c	3-23
epochIPMRemoveEntry (H)	Removes an IP Multicast entry from the IP Multicast partition of the Routing Table.	epochTable.c	3-24
epochIPMSearch (H)	Searches the Routing Table for an IP Multicast entry.	epochTable.c	3-26
epochIPMSearchMian (H)	Searches the Routing Table for an IP Multicast entry and the MIAN.	epochTable.c	3-28

epochIPMAddEntry

Syntax

```
rcpResultStatus_t epochIPMAddEntry(epoch_t *pEpoch,
                                   U32 srcAddr,
                                   U32 dstAddr,
                                   U16 srcPort,
                                   U16 bitmap);
```

Description

This function adds the input IP Multicast address information to the RCP and associated data SRAM. The IP source and destination address are encoded and added to the IP multicast partition of the Routing Table. The destination ports for the Multicast group is added to the associated data SRAM. The Multicast Interface Authentication Number (MIAN) is encoded from the physical source port and added to the MIAN partition of the Associated Data SRAM.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
srcAddr	The 32-bit IP Source Address to be encoded with the IP Destination Group Address and added to the Routing Table. This can be any class other than class D. The customer is responsible for ensuring class D Source addresses are not added to the IP Multicast table. The Epoch forbids the Source Address from being class D and punts all such packets with the DROPSAM puntcode (0x3).
dstAddr	The 32-bit IP Destination Group Address to be encoded with the IP Source Address and added to the Routing Table.
srcPort	The physical source port that the packet is received on. This should be a value from 0 through 15. This is added to the MIAN partition of the associated data SRAM.
bitmap	The 16-bit associated data that will be added to the associated data SRAM. This is the port bitmap that indicates where any corresponding packets should be routed. The appropriate bit is set if the packet is to be routed to the port associated with it. For example, bit 0 = port 0, bit 1 = port 1, bit 2 = port 2 and so on.

Return Value

A variable in the form of the structure ***rcpResultStatus_t*** is returned. This indicates any relevant information regarding the function's success or failure.

If the IP Multicast partition of the Routing Table is full the "code" member is **RCP_BLOCK_FULL.**

Pre-requisites

The IP Multicast pointers (***blockPtr_t***) should be initialized prior to using this routine.

Usage

```
#define L4_STOPADDR 0x1000
#define IPU_STOPADDR 0x1800
#define IPM_STOPADDR 0x1C00
#define IPX_STOPADDR 0x2000

U32 IP_SA = 0xbbccaa03;
U32 IP_DA = 0xebccaa04;
U16 src_port = 2, bitmap = 0x000F;
epoch_t pEpoch;
rcpResultStatus_t write_status;
int i_stat;

i_stat = epochRCPInit(&pEpoch);
//set block pointers
pEpoch.ipmblk.floor = IPU_STOPADDR+1;
pEpoch.ipmblk.bs = IPM_STOPADDR - IPU_STOPADDR;
pEpoch.ipmblk.nextFree = pEpoch.ipmblk.floor;
pEpoch.ipxblk.floor = IPM_STOPADDR+1;
pEpoch.ipxblk.bs = IPX_STOPADDR - IPM_STOPADDR;
pEpoch.ipxblk.nextFree = pEpoch.ipxblk.floor;

if(i_stat == INIT_OK){
      write_status = epochIPMAddEntry(&pEpoch, IP_SA, IP_DA, src_port,
                                      bitmap);
      if(write_status.code & RCP_BLOCK_FULL){
            printf("\nThere is no memory available for RCP write.");
      }
      else{
            printf("\nThe IP addresses/source port/bitmap was added.");
      }
}
else{
      printf("\nERROR : Unable to initialize RCP chain.");
}
```

epochIPMCreateEntry

Syntax

```
static rcp_t epochIPMCreateEntry(epoch_t *pEpoch,
                                 ipaddr_t sa,
                                 ipaddr_t da);
```

Description

This function creates a 64-bit RCP entry from the input 32-bit IP Source and Destination group addresses. The RCP entry is returned in the form of the ***rcp_t*** structure. The format of the binary RCP entry created is in the form shown the Epoch data-sheet. The structure member "lo" stores the lower 32-bit value and the member "hi" stores the upper 32-bit value.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
sa	The 32-bit IP Source Address to be encoded with the IP Destination Group Address and added to the Routing Table. This can be any class other than class D. The customer is responsible for ensuring class D Source addresses are not added to the IP Multicast table. The Epoch forbids the Source Address from being class D and punts all such packets with the DROPSAM puntcode (0x3).
da	The 32-bit IP Destination Group Address to be encoded with the IP Source Address and added to the Routing Table.

Return Value

A variable in the form of the structure ***rcp_t*** is returned. This is the RCP entry that is created from the input address and mask values.

- rcp_t.hi = Bits 63:32 of the RCP entry
- rcp_t.lo = Bits 31:0 of the RCP entry

Pre-requisites

None.

Usage

```
rcp_t rword;
ipaddr_t IP_SA = 0xbbccaa03, IP_DA = 0xebccaa04;

rword = epochIPMCreateEntry(&pEpoch, IP_SA, IP_DA);
printf("\nAn entry was created. It is: 0x%lx (high word)
  0x%lx (low word)",rword.hi, rword.lo);
```

epochIPMRemoveEntry

Syntax

```
rcpResultStatus_t epochIPMRemoveEntry(epoch_t *pEpoch,
                                      ipaddr_t srcAddr,
                                      ipaddr_t dstAddr);
```

Description

This function removes the input IP Source and Destination address combination from the IP Multicast partition of the Routing Table.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
srcAddr	The 32-bit IP Source Address to be deleted. This can be any class other than class D.
dstAddr	The 32-bit IP Destination Group Address to be deleted.

Return Value

A variable in the form of the structure ***rcpResultStatus_t*** is returned. This indicates any relevant information regarding the function's success or failure. The RCP virtual address of where the IPv4 CIDR entry was located (if found) is returned in the "addr" member.

Pre-requisites

The IP Multicast pointers (***blockPtr_t***) should be initialized prior to using this routine.

Usage

```
#define L4_STOPADDR 0x1000
#define IPU_STOPADDR 0x1800
#define IPM_STOPADDR 0x1C00
#define IPX_STOPADDR 0x2000

ipaddr_t IP_SA = 0xbbccaa03;
ipaddr_t IP_DA = 0xebccaa04;
U16 src_port = 2, bitmap = 0x000F;
epoch_t pEpoch;
rcpResultStatus_t write_status, delete_status;
int i_stat;

i_stat = epochRCPInit(&pEpoch);
//set block pointers
pEpoch.ipmblk.floor = IPU_STOPADDR+1;
pEpoch.ipmblk.bs = IPM_STOPADDR - IPU_STOPADDR;
pEpoch.ipmblk.nextFree = pEpoch.ipmblk.floor;
pEpoch.ipxblk.floor = IPM_STOPADDR+1;
pEpoch.ipxblk.bs = IPX_STOPADDR - IPM_STOPADDR;
pEpoch.ipxblk.nextFree = pEpoch.ipxblk.floor;

if(i_stat == INIT_OK){
      write_status = epochIPMAddEntry(&pEpoch, IP_SA, IP_DA, src_port,
                                      bitmap);
      if(write_status.code & RCP_BLOCK_FULL){
            printf("\nThere is no memory available for RCP write.");
      }
      else{
            printf("\nThe IP addresses/source port/bitmap was added.");
      }
      delete_status = epochIPMRemoveEntry(&pEpoch, IP_SA, IP_DA);
      if(delete_status.code & RCP_CMP_NMATCH){
            printf("\nUnable to find IP Multicast entry in Routing
                    Table.");
      }
      else{
            printf("\nThe IP Multicast entry was deleted.");
      }
}
else{
      printf("\nERROR : Unable to initialize RCP chain.");
}
```

epochIPMSearch

Syntax

```
rcpResultStatus_t epochIPMSearch(epoch_t *pEpoch,
                                 U32 srcAddr,
                                 U32 dstAddr);
```

Description

This function performs a binary search for the input IP Source and Destination addresses. The function does not require the MIAN value associated with the Multicast entry. Therefore the function only searches for the IP address combination in the RCP database. If the address combination produces a match during the search, the port bitmap and virtual adjacency pointer/RCP address are returned.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
scrAddr	The 32-bit IP Source Address to be searched for along with the Destination address. This can be any class other than class D.
dstAddr	The 32-bit IP Destination Group Address to be searched for along with the Source address.

Return Value

A variable in the form of the structure ***rcpResultStatus_t*** is returned. This indicates any relevant information regarding the function's success or failure.

- **Successful–**The RCP virtual address of where the IP Multicast entry was located (if found) is returned in the "addr" member. The associated data found in the associated data SRAM is returned in the "data" member. **RCP_CMP_MATCH** or **RCP_CMP_MMATCH** are returned in the "code" member to indicate if there was a match or multiple matches.
- **Unsuccessful–RCP_NO_MATCH** is returned in the "code" member to indicate that there was no match found.

Pre-requisites

None.

Usage

```
ipaddr_t IP_SA = 0xbbccaa03;
ipaddr_t IP_DA = 0xebccaa04;
U16 src_port = 2, bitmap = 0x000F;
epoch_t pEpoch;
rcpResultStatus_t write_status, search_status;
int i_stat;

i_stat = epochRCPInit(&pEpoch);
if(i_stat == INIT_OK){
      write_status = epochIPMAddEntry(&pEpoch, IP_SA, IP_DA, src_port,
                                      bitmap);
      if(write_status.code & RCP_BLOCK_FULL){
            printf("\nThere is no memory available for RCP write.");
      }
      else{
            printf("\nThe IP addresses/source port/bitmap was added.");
      }
      search_status = epochIPMSearch(&pEpoch, IP_SA, IP_DA);
      if(search_status.code & RCP_NO_MATCH){
            printf("\nUnable to find IP Multicast entry in Routing
                    Table.");
      }
      else{
            printf("\nThe IP multicast entry was found at virtual
                    address 0x%x with assoc. data = 0x%lx",
                    search_status.addr, search_status.data);
      }

}
else{
      printf("\nERROR : Unable to initialize RCP chain.");
}
```

epochIPMSearchMian

Syntax

```
rcpResultStatus_t epochIPMSearchMian(epoch_t *pEpoch,
                                     ipaddr_t srcAddr,
                                     ipaddr_t dstAddr,
                                     U16 mian);
```

Description

This function performs a binary search for the input IP Source and Destination addresses. If the address combination produces a match during the search, the function compares the input MIAN value with the MIAN value found during the search. If they are not the same, the function returns **RCP_NO_MATCH**. If the two values are the same, the port bitmap and virtual adjacency pointer/RCP address are returned.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
scrAddr	The 32-bit IP Source Address to be searched for along with the Destination address. This can be any class other than class D.
dstAddr	The 32-bit IP Destination Group Address to be searched for along with the Source address.
mian	The MIAN value to be compared if the function finds a match during the binary search for the IP address combination. This is the physical port that the packet should be received on.

Return Value

A variable in the form of the structure ***rcpResultStatus_t*** is returned. This indicates any relevant information regarding the function's success or failure.

- **Successful**–The RCP virtual address of where the IP Multicast entry was located (if found) is returned in the "addr" member. The associated data found in the associated data SRAM is returned in the "data" member. **RCP_CMP_MATCH** or **RCP_CMP_MMATCH** are returned in the "code" member to indicate if there was a match or multiple matches.
- **Unsuccessful–RCP_NO_MATCH** is returned in the "code" member to indicate that there was no match found.

Pre-requisites

None.

Usage

```
ipaddr_t IP_SA = 0xbbccaa03;
ipaddr_t IP_DA = 0xebccaa04;
U16 mian = 1, bitmap = 0x000F;
epoch_t *pEpoch;
rcpResultStatus_t write_status, search_status;
int i_stat;

i_stat = epochRCPInit(&pEpoch);
if(i_stat == INIT_OK){
      write_status = epochIPMAddEntry(&pEpoch, IP_SA, IP_DA, mian,
                                      bitmap);
      if(write_status.code & RCP_BLOCK_FULL){
            printf("\nThere is no memory available for RCP write.");
      }
      else{
            printf("\nThe IP addresses/source port/bitmap was added.");
      }
      search_status = epochIPMSearchMian(&pEpoch, IP_SA, IP_DA, mian);
      if(search_status.code & RCP_NO_MATCH){
            printf("\nUnable to find IP Multicast entry in Routing
                    Table.");
      }
      else{
            printf("\nThe IP multicast entry was found at virtual
                    address 0x%x with assoc. data = 0x%lx",
                    search_status.addr, search_status.data);
      }

}
else{
      printf("\nERROR : Unable to initialize RCP chain.");
}
```

IPX API

The functions in this section are responsible for maintaining the IPX database table in the RCP. The files in which they may be found and the page in this manual where they can be located is shown for easy reference. Low- and high-level functions are identified by (H) = high-level, (L) = low-level.

Table 3-4: IPX API Functions

API Function Name	Purpose	File Name	Page
epochIPXAddEntry (H)	Adds an IPX entry to the IPX partition of the Routing Table.	epochTable.c	3-31
epochIPXCreateEntry (L)	Creates the 64-bit RCP entry from input IPX net.	epochTable.c	3-33
epochIPXRemoveEntry (H)	Removes an IPX entry from the IPX partition of the Routing Table.	epochTable.c	3-34
epochIPXSearch (H)	Searches the Routing Table for an IPX entry.	epochTable.c	3-36

epochIPXAddEntry

Syntax

```
rcpResultStatus_t epochIPXAddEntry(epoch_t *pEpoch,
                                   U32 ipxDn,
                                   U16 bitmap);
```

Description

This function adds the input IPX net information to the RCP and associated data SRAM. The IPX destination net is encoded and added to the IPX partition of the Routing Table. The destination port for the net is added to the associated data SRAM.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
ipxDn	The 32-bit IPX destination net be encoded and added to the Routing Table. This net is encoded as described in the Epoch data sheet and added to the RCP.
bitmap	The 16-bit associated data that will be added to the associated data SRAM. This is the port bitmap that indicates where any corresponding packets should be routed. The appropriate bit is set if the packet is to be routed to the port associated with it. For example, bit 0 = port 0, bit 1 = port 1, bit 2 = port 2 and so on.

Return Value

A variable in the form of the structure ***rcpResultStatus_t*** is returned. This indicates any relevant information regarding the function's success or failure.

If the IP Multicast partition of the Routing Table is full the "code" member is **RCP_BLOCK_FULL.**

Pre-requisites

The IPX pointers (***blockPtr_t***) should be initialized prior to using this routine.

Usage

```
#define L4_STOPADDR 0x1000
#define IPU_STOPADDR 0x1800
#define IPM_STOPADDR 0x1C00
#define IPX_STOPADDR 0x2000

U32 IPX_net = 0x4024;
U16 associated_data = 0x0002;
epoch_t pEpoch;
rcpResultStatus_t write_status;
int i_stat;

i_stat = epochRCPInit(&pEpoch);
//set block pointers
pEpoch.ipmblk.floor = IPU_STOPADDR+1;
pEpoch.ipmblk.bs = IPM_STOPADDR - IPU_STOPADDR;
pEpoch.ipmblk.nextFree = pEpoch.ipmblk.floor;
pEpoch.ipxblk.floor = IPM_STOPADDR+1;
pEpoch.ipxblk.bs = IPX_STOPADDR - IPM_STOPADDR;
pEpoch.ipxblk.nextFree = pEpoch.ipxblk.floor;

if(i_stat == INIT_OK){
      write_status = epochIPXAddEntry(&pEpoch, IPX_net, associated_data);
      if(write_status.code & RCP_BLOCK_FULL){
            printf("\nThere is no memory available for RCP write.");
      }
      else{
            printf("\nThe IPX net/associated data was added.");
      }
}
else{
      printf("\nERROR : Unable to initialize RCP chain.");
}
```

epochIPXCreate Entry

Syntax

```
static rcp_t IPXCreateEntry(epoch_t *pEpoch,
                            U32 network);
```

Description

This function creates a 64-bit RCP entry from the input 32-bit IPX net. The RCP entry is returned in the form of the ***rcp_t*** structure. The format of the binary RCP entry created is in the form shown the Epoch data-sheet. The structure member "lo" stores the lower 32-bit value and the member "hi" stores the upper 32-bit value.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
network	The 32-bit IPX net to be encoded as a 64-bit RCP entry.

Return Value

A variable in the form of the structure ***rcp_t*** is returned. This is the RCP entry that is created from the input address and mask values.

- rcp_t.hi = Bits 63:32 of the RCP entry.
- rcp_t.lo = Bits 31:0 of the RCP entry.

Pre-requisites

None.

Usage

```
epoch_t pEpoch,
rcp_t rword;
U32 IPX_net = 0x4042;

//assume IPX pointers have been initialized
rword = epochIPXCreateEntry(&pEpoch, IPX_net);
printf("\nAn entry was created. It is: 0x%lx (high word)
  0x%lx (low word)",rword.hi, rword.lo);
```

epochIPXRemoveEntry

Syntax

```
rcpResultStatus_t epochIPXRemoveEntry(epoch_t *pEpoch,
                                      U32 ipxDn);
```

Description

This function removes the input IPX net from the IPX partition of the Routing Table.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
ipxDn	The 32-bit IPX destination net to be deleted.

Return Value

A variable in the form of the structure ***rcpResultStatus_t*** is returned. This indicates any relevant information regarding the function's success or failure. The RCP virtual address of where the IPX entry was located (if found) is returned in the "addr" member.

Pre-requisites

The IPX pointers (***blockPtr_t***) should be initialized prior to using this routine.

Usage

```
#define L4_STOPADDR 0x1000
#define IPU_STOPADDR 0x1800
#define IPM_STOPADDR 0x1C00
#define IPX_STOPADDR 0x2000

U32 IPX_net = 0x4024;
U16 associated_data = 0x0002;
epoch_t pEpoch;
rcpResultStatus_t write_status, delete_status;
int i_stat;

i_stat = epochRCPInit(&pEpoch);
//set block pointers
pEpoch.ipmblk.floor = IPU_STOPADDR+1;
pEpoch.ipmblk.bs = IPM_STOPADDR - IPU_STOPADDR;
pEpoch.ipmblk.nextFree = pEpoch.ipmblk.floor;
pEpoch.ipxblk.floor = IPM_STOPADDR+1;
pEpoch.ipxblk.bs = IPX_STOPADDR - IPM_STOPADDR;
pEpoch.ipxblk.nextFree = pEpoch.ipxblk.floor;

if(i_stat == INIT_OK){
      write_status = epochIPXAddEntry(&pEpoch, IPX_net, associated_data);
      if(write_status.code & RCP_BLOCK_FULL){
            printf("\nThere is no memory available for RCP write.");
      }
      else{
            printf("\nThe IPX net/associated data was added.");
      }
      delete_status = epochIPXRemoveEntry(&pEpoch, IPX_net);
      if(delete_status.code & RCP_NO_MATCH){
            printf("\nUnable to find IPX entry in Routing Table.");
      }
      else{
            printf("\nThe IPX entry was deleted.");
      }

}
else{
      printf("\nERROR : Unable to initialize RCP chain.");
}
```

epochIPXSearch

Syntax

```
rcpResultStatus_t epochIPXSearch(epoch_t *pEpoch,
                                 U32 ipxDn);
```

Description

This function performs a binary search for the input IPX net. If the net produces a match during the search, the port bitmap and virtual adjacency pointer/RCP address are returned.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
ipxDn	The 32-bit IPX destination net to be searched for.

Return Value

A variable in the form of the structure ***rcpResultStatus_t*** is returned. This indicates any relevant information regarding the function's success or failure.

- **Successful–**The RCP virtual address of where the IPX entry was located (if found) is returned in the "addr" member. The associated data found in the associated data SRAM is returned in the "data" member. **RCP_CMP_MATCH** or **RCP_CMP_MMATCH** are returned in the "code" member to indicate if there was a match or multiple matches.
- **Unsuccessful–RCP_NO_MATCH** is returned in the "code" member to indicate that there was no match found.

Pre-requisites

None.

Usage

```
U32 IPX_net = 0x4024;
U16 associated_data = 0x0002;
epoch_t pEpoch;
rcpResultStatus_t write_status, search_status;
int i_stat;

i_stat = epochRCPInit(&pEpoch);
//assuming the block pointers have been initialized

if(i_stat == INIT_OK){
      write_status = epochIPXAddEntry(&pEpoch, IPX_net, associated_data);
      if(write_status.code & RCP_BLOCK_FULL){
            printf("\nThere is no memory available for RCP write.");
      }
      else{
            printf("\nThe IPX net/associated data was added.");
      }
      search_status = epochIPXSearch(&pEpoch, IPX_net);
      if(search_status.code & RCP_NO_MATCH){
            printf("\nUnable to find IPX entry in Routing
                  Table.");
      }
      else{
            printf("\nThe IPX entry was found at virtual
                  address 0x%x with assoc. data = 0x%lx",
                  search_status.addr, search_status.data);
      }

}
else{
      printf("\nERROR : Unable to initialize RCP chain.");
}
```

LAYER 4 API

The functions in this section are responsible for maintaining the Layer 4 database table in the RCP. The files in which they may be found and the page in this manual where they can be located is shown for easy reference. Low- and high-level functions are identified by (H) = high-level, (L) = low-level.

Table 3-5: Layer 4 API Functions

API Function Name	Purpose	File Name	Page
DecodeMicroflow (L)	Decodes the Layer 4 “parent” and “child” RCP entries to produce the Microflow information.	epochTable.c	3-39
ExtractPind (L)	Extracts the RCP physical address of a Layer 4 “parent” entry from the corresponding “child” entry.	epochTable.c	3-41
L4CreateChild (L)	Creates the 64-bit RCP “child” entry from an input Microflow and “parent” index.	epochTable.c	3-42
L4CreateParent (L)	Creates the 64-bit RCP “parent” entry from the input Microflow.	epochTable.c	3-44
L4FindSibling (L)	Finds Layer 4 “child” entries that have the same “parent”.	epochTable.c	3-46
epochL4FlowAddEntry (H)	Adds a Layer 4 Microflow entry to the Layer 4 (Microflow) partition of the Routing Table.	epochTable.c	3-48
epochL4FlowAge (H)	Ages (or deletes) all Layer 4 entries with the “touch” bit set to 0.	epochTable.c	3-50
epochL4FlowRemoveEntry (H)	Removes a Layer 4 Microflow entry from the Layer 4 (Microflow) partition of the Routing Table.	epochTable.c	3-51
epochL4FlowSearch (H)	Searches the Routing Table for a Layer 4 Microflow.	epochTable.c	3-53
epochBACReadTableEntry (H)	Reads a BAC Table Flow Handle from the BAC partition of the Routing Table.	epochLib.c	3-55
epochBACWriteTableEntry (H)	Writes a BAC Table Flow Handle into the BAC partition of the Routing Table.	epochLib.c	3-56

DecodeMicroflow

Syntax

```
static microflow_t DecodeMicroflow(epoch_t *pEpoch,
                                   rcp_t *parentRcp,
                                   rcp_t *childRcp);
```

Description

This function performs a Layer 4 Microflow conversion from the input RCP words. The user inputs the 64-bit RCP "parent" and "child" entries and is returned the decoded Layer 4 Microflow. The encoding method of the RCP entries is described in the Epoch data sheet. The Layer 4 Microflow is returned as the IP Source and Destination addresses, TCP/UDP Source and Destination port numbers and the input physical port number.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
**parentRcp*	A pointer to the 64-bit RCP "parent" entry. The parameter should be a pointer to a variable of the structure ***rcp_t***. The variable has two members: rcp_t.hi = Bits 63:32 of the RCP entry rcp_t.lo = Bits 31:0 of the RCP entry
**childRcp*	A pointer to the 64-bit RCP "child" entry. The parameter should be a pointer to a variable of the structure ***rcp_t***. The variable has two members: rcp_t.hi =Bits 63:32 of the RCP entry rcp_t.lo =Bits 31:0 of the RCP entry

Return Value

A variable in the form of the ***microflow_t*** structure is returned. This shows each individual element that was used to encode the two RCP entries. The members are as follows:
microflow_t.sa: The IP Source Address.
microflow_t.da: The IP Destination Address.
microflow_t.sp: The TCP or UDP Source Port.
microflow_t.dp: The TCP or UDP Destination Port.
microflow_t.ifc: The physical port number that Epoch received the flow.

Pre-requisites

None.

Usage

```
#define CMP_ADDR_MASK 0x7FFFL
#define MAGIC_AGE_NUM 0xFFBF

microflow_t this_flow, read_flow;
rcp_t parent, child;
U32 cmp_result;
U16 pind, cind, flowhandle;
epoch_t pEpoch;

this_flow.sa = 0xbbccaa04;
this_flow.da = 0xbbccaa05;
this_flow.sp = 23;
this_flow.dp = 1000;
this_flow.ifc = 3; flow_handle = 0x0007;

//assuming the RCPs and the block pointers have been initialized
parent = L4CreateParent(&pEpoch, &this_flow);
cmp_result = epochRCPBinarySearch(&pEpoch, &parent, 0);
if (!(cmp_result & CMP_M)){  // no parent match
       pind = epochRCPNextFree(&pEpoch);
       if (pind > pEpoch.L4blk.maxAddr){ //error, address out of range
              printf("\nCant write Parent. The Layer 4 block is full.");
       }
       else {  //write entry to Routing Table
              epochRCPWriteEntry(&pEpoch, pind, &parent);
              epochRCPSRAMWrite(&pEpoch, pind, MAGIC_AGE_NUM);
              pEpoch.L4blk.nParent++; //increment parent count
       }
}
else { //parent match
       pind = PAtoA(&pEpoch, (U16)(cmp_result & CMP_ADDR_MASK));
}
child = L4CreateChild(&pEpoch, this_flow, pind);
//search for child
cmp_result = epochRCPBinarySearch(&pEpoch, &child, 0);
if (!(cmp_result & CMP_M)){// no child match
       cind = epochRCPNextFree(&pEpoch);
}
else { //child match --just overwrite it
       cind = PAtoA(&pEpoch, (U16)(cmp_result & CMP_ADDR_MASK));
}
if (cind > pEpoch.L4blk.maxAddr){
       //error, address out of range
       printf("\nCant write Child. The Layer 4 block is full.");
}
else {
       epochRCPWriteEntry(&pEpoch, cind, &child);
       epochRCPSRAMWrite(&pEpoch, cind, flowhandle);
       pEpoch.L4blk.nChild++; //increment child count
}
read_flow = DecodeMicroflow(&pEpoch, &parent, &child);
printf("\nThe parent and child entries were decoded.");
printf("\nSA: %lx DA: %lx", read_flow.sa, read_flow.da);
printf("\nSP: %x DP: %x", read_flow.sp, read_flow.dp);
printf("\nIncoming Port #: %lx", read_flow.ifc);
```

ExtractPind

Syntax

```
static U16 ExtractPind(rcp_t *childRcp);
```

Description

This function extracts the "parent" index from the input "child" RCP entry. That means that the function finds the page and physical address of where the "parent" entry is located in the RCP.

Input Parameters

**childRcp*	A pointer to the 64-bit RCP "child" entry. The parameter should be a pointer to a variable of the structure ***rcp_t***. The variable has two members: rcp_t.hi = Bits 63:32 of the RCP entry. rcp_t.lo = Bits 31:0 of the RCP entry.

Return Value

The page and virtual address of where the "parent" entry is located in the RCP.

Pre-requisites

None.

Usage

```
#define CMP_MATCH 6
rcpResultStatus_t rcp_status;
rcp_t child;
U16 pind;
microflow_t this_flow;
epoch_t pEpoch;

this_flow.sa = 0xbbccaa04;   this_flow.da = 0xbbccaa05;
this_flow.sp = 23;           this_flow.dp = 1000;
this_flow.ifc = 3;
//assuming the RCPs and block pointers have been initialized.
//find a L4 entry
rcp_status = epochL4FlowSearch(&pEpoch, &this_flow);
if (rcp_status.code == RCP_CMP_MATCH){
      child = epochRCPReadEntry(&pEpoch, (U16) rcp_status.addr);
      pind = ExtractPind(&child);
      printf("\nThe parent was located at 0x%x", pind);
      printf("\n(in Page Address / Physical Address format)");
}
else{
      printf("\nUnable to locate parent.");
}
```

L4CreateChild

Syntax

```
static rcp_t L4CreateChild(epoch_t *pEpoch,
                           microflow_t *pFlow,
                           U16 pInd);
```

Description

This function creates a 64-bit "child" RCP entry from the input microflow and "parent" index. The RCP entry is returned in the form of the ***rcp_t*** structure. The format of the binary RCP entry created is in the form shown the Epoch data-sheet. The structure member "lo" stores the lower 32-bit value and the member "hi" stores the upper 32-bit value.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
**pFlow*	A pointer to the Layer 4 Microflow to be used when creating the "child" entry. The parameter should be pointer to a variable of the ***microflow_t*** structure. The variable has five members: microflow_t.sa–The IP Source Address. microflow_t.da–The IP Destination Address. microflow_t.sp–The TCP or UDP Source Port. microflow_t.dp–The TCP or UDP Destination Port. microflow_t.ifc–The physical port number that Epoch received the flow.
pInd	The parent Index. That is the contiguous physical address of the corresponding Layer 4 "parent" entry.

Return Value

A variable in the form of the structure ***rcp_t*** is returned. This is the RCP entry that is created from the microflow and index values.

- rcp_t.hi = Bits 63:32 of the RCP entry
- rcp_t.lo = Bits 31:0 of the RCP entry

Pre-requisites

None.

Usage

```
#define CMP_ADDR_MASK 0x7FFFL
#define MAGIC_AGE_NUM 0xFFBF

microflow_t this_flow;
rcp_t parent, child;
U32 cmp_result;
U16 pind, cind, flowhandle;
epoch_t pEpoch;

this_flow.sa = 0xbbccaa04;
this_flow.da = 0xbbccaa05;
this_flow.sp = 23;
this_flow.dp = 1000;
this_flow.ifc = 3; flow_handle = 0x0007;

//assuming the RCPs and the block pointers have been initialized
parent = L4CreateParent(&pEpoch, &this_flow);
cmp_result = epochRCPBinarySearch(&pEpoch, &parent, 0);
if (!(cmp_result & CMP_M)){  // no parent match
      pind = epochRCPNextFree(&pEpoch);
      if (pind > pEpoch.L4blk.maxAddr){ //error, address out of range
            printf("\nCant write Parent. The Layer 4 block is full.");
      }
      else {  //write entry to Routing Table
            epochRCPWriteEntry(&pEpoch, pind, &parent);
            epochRCPSRAMWrite(&pEpoch, pind, MAGIC_AGE_NUM);
            pEpoch.L4blk.nParent++; //increment parent count
      }
}
else { //parent match
      pind = PAtoA(&pEpoch, (U16)(cmp_result & CMP_ADDR_MASK));
}
child = L4CreateChild(&pEpoch, this_flow, pind);
//search for child
cmp_result = epochRCPBinarySearch(&pEpoch, &child, 0);
if (!(cmp_result & CMP_M)){// no child match
      cind = epochRCPNextFree(&pEpoch);
}
else { //child match --just overwrite it
      cind = PAtoA(&pEpoch, (U16)(cmp_result & CMP_ADDR_MASK));
}
if (cind > pEpoch.L4blk.maxAddr){
      //error, address out of range
      printf("\nCant write Child. The Layer 4 block is full.");
}
else {
      epochRCPWriteEntry(&pEpoch, cind, &child);
      epochRCPSRAMWrite(&pEpoch, cind, flowhandle);
      pEpoch.L4blk.nChild++; //increment child count
}
```

L4CreateParent

Syntax

```
static rcp_t L4CreateParent(epoch_t *pEpoch,
                            microflow *pFlow);
```

Description

This function creates a 64-bit "parent" RCP entry from the input microflow. The RCP entry is returned in the form of the ***rcp_t*** structure. The format of the binary RCP entry created is in the form shown the Epoch data-sheet. The structure member "lo" stores the lower 32-bit value and the member "hi" stores the upper 32-bit value.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
**pFlow*	A pointer to the Layer 4 Microflow to be used when creating the "parent" entry. The parameter should be pointer to a variable of the ***microflow_t*** structure. The variable has five members: microflow_t.sa–The IP Source Address. microflow_t.da–The IP Destination Address. microflow_t.sp–The TCP or UDP Source Port. microflow_t.dp–The TCP or UDP Destination Port. microflow_t.ifc–The physical port number that Epoch received the flow.

Return Value

A variable in the form of the structure ***rcp_t*** is returned. This is the RCP entry that is created from the microflow.

- rcp_t.hi = Bits 63:32 of the RCP entry
- rcp_t.lo = Bits 31:0 of the RCP entry

Pre-requisites

None.

Usage

```
#define CMP_ADDR_MASK 0x7FFFL
#define MAGIC_AGE_NUM 0xFFBF

microflow_t this_flow;
rcp_t parent, child;
U32 cmp_result;
U16 pind, cind, flowhandle;
epoch_t pEpoch;

this_flow.sa = 0xbbccaa04;
this_flow.da = 0xbbccaa05;
this_flow.sp = 23;
this_flow.dp = 1000;
this_flow.ifc = 3; flow_handle = 0x0007;

//assuming the RCPs and the block pointers have been initialized
parent = L4CreateParent(&pEpoch, &this_flow);
cmp_result = epochRCPBinarySearch(&pEpoch, &parent, 0);
if (!(cmp_result & CMP_M)){  // no parent match
      pind = epochRCPNextFree(&pEpoch);
      if (pind > pEpoch.L4blk.maxAddr){ //error, address out of range
            printf("\nCant write Parent. The Layer 4 block is full.");
      }
      else {  //write entry to Routing Table
            epochRCPWriteEntry(&pEpoch, pind, &parent);
            epochRCPSRAMWrite(&pEpoch, pind, MAGIC_AGE_NUM);
            pEpoch.L4blk.nParent++; //increment parent count
      }
}
else { //parent match
      pind = PAtoA(&pEpoch, (U16)(cmp_result & CMP_ADDR_MASK));
}
child = L4CreateChild(&pEpoch, this_flow, pind);
//search for child
cmp_result = epochRCPBinarySearch(&pEpoch, &child, 0);
if (!(cmp_result & CMP_M)){// no child match
      cind = epochRCPNextFree(&pEpoch);
}
else { //child match --just overwrite it
      cind = PAtoA(&pEpoch, (U16)(cmp_result & CMP_ADDR_MASK));
}
if (cind > pEpoch.L4blk.maxAddr){
      //error, address out of range
      printf("\nCant write Child. The Layer 4 block is full.");
}
else {
      epochRCPWriteEntry(&pEpoch, cind, &child);
      epochRCPSRAMWrite(&pEpoch, cind, flowhandle);
      pEpoch.L4blk.nChild++; //increment child count
}
```

L4FindSibling

Syntax

```
static U32 L4FindSibling(epoch_t *pEpoch,
                         U16 pInd);
```

Description

This function searches the Layer 4 RCP database for "child" entries that have the input "parent" index encoded. If any RCP entries that constitute a "child" of the "parent" index that is given, relevant information is returned.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
pInd	The parent Index. That is the contiguous physical address of the corresponding Layer 4 "parent" entry.

Return Value

The contents of the Epoch CRI register 0x134 is returned. This shows relevant information regarding the search for any "child" entries. The information is as follows:

Bits	Explanation
[12:0]	This is the Index of any matching entries found during the search. It uses the standard Epoch format and shows physical address within a specific RCP. Only valid if bit 15 = 1 or bit 16 = 1.
[14:13]	Page address of any matching entries found during the search. This corresponds to the Page Address values set during initialization. The usual format is: 00 = First device 01 = Second device 10 = Third device 11 = Fourth device
15	Match Flag. If this bit = 1, there was a MATCH and therefore a "child" entry was found.
16	Multiple Match Flag. If this bit = 1, there was a Multiple MATCH and therefore more than one "child" entry was found. Bits [14:0] specify the highest priority entry matched.

Pre-requisites

None.

Usage

```
U32 search_result;
epoch_t pEpoch;
U16 PageAddress, PhysicalAddress, parent_index = 23;
//assuming the RCPs and the block pointers have been initialized
search_result = L4FindSibling(&pEpoch, (U16) parent_index);
PageAddress = (U16)(((search_result & 0x0000600) >> 13) & 0x3);
PhysicalAddress = (U16) search_result & 0x00001FFF;
if(seach_result & 0x8000){
      printf("\nOnly one child entry was found.");
      printf("\nRCP number : 0x%x", PageAddress);
      printf("\nPhysical address within that device: 0x%x",
                PhysicalAddress);
}
else if(seach_result & 0x10000){
      printf("\nMultiple child entries were found.\nHighest Priority
              entry is located at following address. ");
      printf("\nRCP number : 0x%x", PageAddress);
      printf("\nPhysical address within that device: 0x%x",
                PhysicalAddress);
}
else if((search_result & 0x18000) == 0){
      printf("\nNo child entries were found.");
}
```

epochL4FlowAddEntry

Syntax

```
rcpResultStatus_t epochL4FlowAddEntry(epoch_t *pEpoch,
                                      microflow_t *pFlow,
                                      U16 flowhandle);
```

Description

This function adds the input Layer 4 Microflow information to the RCP and associated data. The microflow input is encoded as "parent" and "child" RCP entries and added to the Layer 4 partition of the Routing Table. The input Flow Handle is added to the associated data SRAM.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
**pFlow*	A pointer to the Layer 4 Microflow to be added to the Routing Table. The parameter should be pointer to a variable of the ***microflow_t*** structure. The variable has five members: microflow_t.sa–The IP Source Address. microflow_t.da–The IP Destination Address. microflow_t.sp–The TCP or UDP Source Port. microflow_t.dp–The TCP or UDP Destination Port. microflow_t.ifc–The physical port number that Epoch received the flow.
flowhandle	The 16-bit Flow Handle to be added to the associated data SRAM. This is described in the Epoch data sheet.

Return Value

A variable in the form of the structure ***rcpResultStatus_t*** is returned. This indicates any relevant information regarding the function's success or failure.

If the Layer 4 partition of the Routing Table is full the "code" member is **RCP_BLOCK_FULL** and the "addr" member is the next free location in the RCP**.**

Pre-requisites

The Layer 4 pointers (***blockPtr_t***) should be initialized prior to using this routine.

Usage

```
#define L4_STOPADDR 0x1000
#define IPU_STOPADDR 0x1800
#define IPM_STOPADDR 0x1C00
#define IPX_STOPADDR 0x2000

rcpResultStatus_t write_status;
U16 flowhandle;
microflow_t this_flow;
epoch_t pEpoch;
int i_stat;

this_flow.sa = 0xbbccaa04;
this_flow.da = 0xbbccaa05;
this_flow.sp = 23;
this_flow.dp = 1000;
this_flow.ifc = 3;
flow_handle = 0x0007;

i_stat = epochRCPInit(&pEpoch);
//set block pointers
pEpoch.ipmblk.floor = IPU_STOPADDR+1;
pEpoch.ipmblk.bs = IPM_STOPADDR - IPU_STOPADDR;
pEpoch.ipmblk.nextFree = pEpoch.ipmblk.floor;
pEpoch.ipxblk.floor = IPM_STOPADDR+1;
pEpoch.ipxblk.bs = IPX_STOPADDR - IPM_STOPADDR;
pEpoch.ipxblk.nextFree = pEpoch.ipxblk.floor;

if(i_stat == INIT_OK){
      //add a L4 entry to the Routing Table
      write_status = epochL4FlowAddEntry(&pEpoch, &this_flow, flow_handle);
      if(write_status.code == RCP_BLOCK_FULL){
            printf("\nThere is no memory available for RCP write.");
      }
      else{
            printf("\nThe Layer 4 flow was added.");
      }
}
else{
      printf("\nERROR : Unable to initialize RCP chain.");
}
```

epochL4FlowAge

Syntax

```
rcpResultStatus_t L4FlowAge(epoch_t *pEpoch);
```

Description

If an aging scheme is used, bit 5 of the Flow Handle (the "touch" bit) is set to 1 every time the Epoch receives a particular flow. This function can be used to remove all the flows from the Routing Table that have not had their "touch" bits set to 1. The function ages the Layer 4 partition of the Routing Table. That means that all Layer 4 entries in RCP and their corresponding SRAM information are deleted if the "touch" bit of the Flow Handle is 0. All existing entries with their "touch" bits set to 1 have the bit reset back to 0.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.

Return Value

A variable in the form of the structure ***rcpResultStatus_t*** is returned. This indicates how many Layer 4 microflows were deleted from the Table. This information is returned in the "data" member.

Pre-requisites

The Layer 4 pointers (***blockPtr_t***) should be initialized prior to using this routine.

Usage

```
epoch_t pEpoch;
rcpResultStatus_t age_status;

//Assuming the RCPs and the block pointers have bben initialized
//Also assuming that there is some L4 entries in Table.
age_status = epochL4FlowAge(&pEpoch);
printf("\nThe layer 4 microflow Table was aged.");
printf("\n%d entries were deleted from the Table.", age_status.data);
```

epochL4FlowRemoveEntry

Syntax

```
rcpResultStatus_t epochL4FlowRemoveEntry(epoch_t *pEpoch,
                                         microflow_t *pFlow);
```

Description

This function removes the input Layer 4 Microflow from the Layer 4 partition of the Routing Table.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
**pFlow*	A pointer to the Layer 4 Microflow to be added to the Routing Table. The parameter should be pointer to a variable of the ***microflow_t*** structure. The variable has five members: microflow_t.sa–The IP Source Address. microflow_t.da–The IP Destination Address. microflow_t.sp–The TCP or UDP Source Port. microflow_t.dp–The TCP or UDP Destination Port. microflow_t.ifc–The physical port number that Epoch received the flow.

Return Value

A variable in the form of the structure ***rcpResultStatus_t*** is returned. This indicates any relevant information regarding the function's success or failure. The RCP virtual address of where the "child" entry was located is returned in the "addr" member. The "code" member indicates if the "child" and "parent" were successfully deleted.

Pre-requisites

The Layer 4 pointers (***blockPtr_t***) should be initialized prior to using this routine.

Usage

```
#define L4_STOPADDR 0x1000
#define IPU_STOPADDR 0x1800
#define IPM_STOPADDR 0x1C00
#define IPX_STOPADDR 0x2000

rcpResultStatus_t write_status, delete_status;
U16 flowhandle;
microflow_t this_flow;
epoch_t pEpoch;
int i_stat;

this_flow.sa = 0xbbccaa04;
this_flow.da = 0xbbccaa05;
this_flow.sp = 23;
this_flow.dp = 1000;
this_flow.ifc = 3;
flow_handle = 0x0007;

i_stat = epochRCPInit(&pEpoch);
//set block pointers
pEpoch.ipmblk.floor = IPU_STOPADDR+1;
pEpoch.ipmblk.bs = IPM_STOPADDR - IPU_STOPADDR;
pEpoch.ipmblk.nextFree = pEpoch.ipmblk.floor;
pEpoch.ipxblk.floor = IPM_STOPADDR+1;
pEpoch.ipxblk.bs = IPX_STOPADDR - IPM_STOPADDR;
pEpoch.ipxblk.nextFree = pEpoch.ipxblk.floor;

if(i_stat == INIT_OK){
      //add a L4 entry to the Routing Table
      write_status = epochL4FlowAddEntry(&pEpoch, &this_flow, flow_handle);
      if(write_status.code == RCP_BLOCK_FULL){
            printf("\nThere is no memory available for RCP write.");
      }
      else{
            printf("\nThe Layer 4 flow was added.");
      }
      delete_status = epochL4FlowRemoveEntry(&pEpoch, &this_flow);
      if(delete_status.code & RCP_NO_MATCH){
            printf("\nUnable to find L4 entry in Routing Table.");
      }
      else{
            printf("\nThe L4 entry was deleted.");
      }
}
else{
      printf("\nERROR : Unable to initialize RCP chain.");
}
```

epochL4FlowSearch

Syntax

```
rcpResultStatus_t epochL4FlowSearch(epoch_t *pEpoch,
                                    microflow_t *pFlow);
```

Description

This function performs a two-phase binary search for the input Layer 4 Microflow. The function encodes the microflow and searches for the "parent" entry. If successful, it also search for the associated "child" entry. If the microflow produces a match during the search, the Flow Handle and virtual adjacency pointer/RCP address of the "child" entry are returned.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
**pFlow*	A pointer to the Layer 4 Microflow to be added to the Routing Table. The parameter should be pointer to a variable of the ***microflow_t*** structure. The variable has five members: microflow_t.sa–The IP Source Address. microflow_t.da–The IP Destination Address. microflow_t.sp–The TCP or UDP Source Port. microflow_t.dp–The TCP or UDP Destination Port. microflow_t.ifc–The physical port number that Epoch received the flow.

Return Value

A variable in the form of the structure ***rcpResultStatus_t*** is returned. This indicates any relevant information regarding the function's success or failure.

- **Successful–**The RCP virtual address of where the "child" entry was located (if found) is returned in the "addr" member. The Flow Handle found in the associated data SRAM is returned in the "data" member. **RCP_CMP_MATCH** is returned in the "code" member to indicate that there was a match.
- **Unsuccessful–RCP_NO_MATCH** is returned in the "code" member to indicate that there was no match found.

Pre-requisites

None.

Usage

```
#define L4_STOPADDR 0x1000
#define IPU_STOPADDR 0x1800
#define IPM_STOPADDR 0x1C00
#define IPX_STOPADDR 0x2000

rcpResultStatus_t write_status, search_status;
U16 flowhandle;
microflow_t this_flow;
epoch_t pEpoch;
int i_stat;

this_flow.sa = 0xbbccaa04;
this_flow.da = 0xbbccaa05;
this_flow.sp = 23;
this_flow.dp = 1000;
this_flow.ifc = 3;
flow_handle = 0x0007;

i_stat = epochRCPInit(&pEpoch);
//set block pointers
pEpoch.ipmblk.floor = IPU_STOPADDR+1;
pEpoch.ipmblk.bs = IPM_STOPADDR - IPU_STOPADDR;
pEpoch.ipmblk.nextFree = pEpoch.ipmblk.floor;
pEpoch.ipxblk.floor = IPM_STOPADDR+1;
pEpoch.ipxblk.bs = IPX_STOPADDR - IPM_STOPADDR;
pEpoch.ipxblk.nextFree = pEpoch.ipxblk.floor;

if(i_stat == INIT_OK){
      //add a L4 entry to the Routing Table
      write_status = epochL4FlowAddEntry(&pEpoch, this_flow, flow_handle);
      if(write_status.code == RCP_BLOCK_FULL){
            printf("\nThere is no memory available for RCP write.");
      }
      else{
            printf("\nThe Layer 4 flow was added.");
      }
      search_status = epochL4FlowSearch(&pEpoch, &this_flow);
      if(search_status.code & RCP_NO_MATCH){
            printf("\nUnable to find L4 entry in Routing Table.");
      }
      else{
            printf("\nThe L4 entry (child) was found at virtual
                  address 0x%x with assoc. data = 0x%lx",
                  search_status.addr, search_status.data);
      }
}
else{
      printf("\nERROR : Unable to initialize RCP chain.");
}
```

epochBACReadTableEntry

Syntax

```
U16 epochBACReadTableEntry(epoch_t *pEpoch,
                           U16 addr);
```

Description

This function reads the BAC Table entry specified by the input DS field. The input DS field should be any value between 0 and 0xFF. The function locates and returns the corresponding 16-bit Flow Handle from the BAC Table.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
addr	The DS field for which the user wishes to locate a Flow Handle. The input should be any valid DS field value between 0 and 0xFF.

Return Value

The 16-bit Flow Handle found in the BAC Table partition of the associated data SRAM is returned.

Pre-requisites

None.

Usage

```
U16 DSField, FlowHandle;
epoch_t pEpoch;

//assuming RCPs and block pointers have been initialized.
FlowHandle = epochBACReadTableEntry(&pEpoch, addr);

Printf("\nThe Flow Handle for and entry with a DS field =
        0x%x is 0x%x", DSField, FlowHandle);
```

epochBACWriteTableEntry

Syntax

```
int epochBACWriteTableEntry(epoch_t *pEpoch,
                            U16 addr,
                            U16 data);
```

Description

This function writes the 16-bit BAC Table entry specified by the input flowhandle field into the appropriate location. The input addr specifies the DS field that is used when placing the entry. The input DS field should be any value between 0 and 0xFF.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
addr	The DS field for which the user wishes to write a Flow Handle into the BAC Table. The input should be any valid DS field value between 0 and 0xFF.
data	The 16-bit Flow Handle that is to be written into the appropriate location in the BAC Table.

Return Value

An integer is returned to relay the status of the attempted write operation. If the write was successful, 0 is returned. If the write failed, -1 is returned.

Pre-requisites

None.

Usage

```
U16 DSField = 0x23, FlowHandle = 0x0007;
epoch_t pEpoch;
int i_stat;

//assuming RCPs and block pointers have been initialized.
i_stat = epochBACWriteTableEntry(&pEpoch, addr, FlowHandle);
if(i_stat == -1){
      printf("\nThere was an error while writing entry.");
}
else{
      printf("\nThe entry with Flow Handle = 0x%x and DS field = 0x%x
              was added", FlowHandle, DSField);
}
```

INITIALIZATION API

The functions in this section are responsible for initializing the Packet pointers, Queue pointers, the Packet Control pointers and the RCP database. The files in which they may be found and the page in this manual where they can be located is shown for easy reference. Low- and high-level functions are identified by (H) = high-level, (L) = low-level.

Table 3-6: API Initialization Functions

API Function Name	Purpose	File Name	Page
epochPKMInit (H)	Initializes the Packet Manager SRAM. This involves initializing the internal Queue pointers and the external Packet pointers.	epochLib.c	3-58
epochRCPInit (H)	Initializes the RCP(s).	epochLib.c	3-60
epochSDRAMInit (H)	Initializes the control SDRAM and tests the data SDRAM.	epochLib.c	3-62
epochQueuePtrMemInit (L)	Initializes the 128-entry internal Queue Pointer SRAM.	epochLib.c	3-64
epochPacketPtrInit (L)	Initializes the external 64K (65,536) location Packet Pointer SRAM.	epochLib.c	3-65
epochPacketPtrClear (L)	Clears the external 64K (65,536) location Packet Pointer SRAM, setting the entries to 0.	epochLib.c	3-66

epochPKMInit

Syntax

```
int epochInitPKM(epoch_t *pEpoch,
                 U32 numBufs,
                 U32 usePort);
```

Description

This function initializes the Epoch Packet pointers and Queue pointers as described in Packet Manager section of the Epoch data sheet. There are 128 Queue pointers that are stored in an internal and 64K (65,536) Packet pointers that are stored in an external SRAM that both require initialization. The user may select the number of physical ports and the number of pointers to initialize.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure is initialized with information required for RCP address to RCP virtual address translation.
numBufs	This is the number of Packet pointers to be initialized. The Epoch has 65,536 Packet pointers that allow it to store 65,536 64-byte packets. This parameter indicates how many of the 65,536 Packet pointers are to be initialized. The parameter should be a value from 0 through 65,535 (0x0000 through 0xFFFF). The relationship between the Queue pointers and Packet pointers *must* be maintained or an error occurs. This means that there *must* be enough packet pointers initialized to satisfy the number of Queue pointers that are initialized. Therefore *numBufs must* always be greater than the number of ports specified by *usePort* times eight.
usePort	This is the 16-bit port bitmap. There are 128 Queue pointers that are used by Epoch for sixteen ports each having eight queues. Each physical port (0 – 15) may have its pointers initialized. The appropriate bit is set to 1 if the port that is indicated is to be initialized. If the bit is set to 0, the port is not affected by the function. For example, bit 0 = port 0, bit 1 = port 1, bit 2 = port 2, and so on. For normal initialization of sixteen ports this parameter should be set to 0xFFFF.

Return Value

An integer is returned if the initialization was successful. If successful, **INIT_OK** is returned. If not successful, **INIT_FAIL** is returned.

Pre-requisites

None.

Usage

```
U16 useport = 0xFFFF;
U32 num_of_bufs = 0xFFFF;
epoch_t pEpoch;
int i_stat;

//assuming RCPs and block pointers have been initialized.

//initialize all ports and all buffers
i_stat = epochPKMInit(&pEpoch, num_of_bufs, useport);
      if(i_stat == INIT_OK){
            printf("\nThe PKM was initialized.");
}
else {
            print("\nThere was an error during initialization.
                  \nPlease investigate.");
}
```

epochRCPInit

Syntax

```
int epochRCPInit(epoch_t *pEpoch);
```

Description

This function initializes the Epoch RCP chain into one contiguous virtual address space and initializes the page address conversion data in the ***blockSort_t*** structure. The function performs the following tasks:

1. Detects how many RCPs are being used in the system (maximum = 4).
2. Detects the size of each RCP being used.
3. Places all RCPs in hardware mode.
4. Initializes all RCPs with correct Page Address values.
5. Initializes the ***pageAddr[n]***, ***pageSize[n]***, ***chipSelectLkup[n]***, and ***pageAddrLkup[n]*** members of the ***blockSort_t*** structure. This allows the conversion between RCP address and Page address values and virtual contiguous address values.
6. Sets the RCP Mask registers to the values required for Layer 4 searches and binary searches that return a match for any valid entry.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure is initialized with information required for RCP address to RCP virtual address translation.

Return Value

An integer is returned if the intialization was successful. If successful, **INIT_OK** is returned. If not successful, **INIT_FAIL** is returned.

Pre-requisites

None.

Usage

```
epoch_t pEpoch;
int i_stat;

//assuming block pointers have been initialized.

i_stat = epochRCPInit(&pEpoch);
if(i_stat == INIT_OK){
      printf(“\nThe RCP chain was initialized.”);
}
else{
      printf(“\nERROR : Unable to initialize RCP chain.”);
}
```

epochSDRAMInit

Syntax

```
int epochSDRAMInit(epoch_t *pEpoch,
                   U32 refreshPeriod,
                   U32 progSync);
```

Description

This function initializes the Epoch SDRAM memories. This involves initializing the Buffer Control pointer SDRAM and testing the Buffer Data SDRAM.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
refreshPeriod	The refresh period for the system SDRAM. This is the number of system clock cycles between Buffer and Buffer control refresh cycles. This number is calculated by the following formula: **refresh_period = system_clock x required_SDRAM_period** system_clock is the Epoch and the SDRAM clock frequency. Required_SDRAM_period is the refresh period required by the SDRAM being used. For a 32ms, 2K-refresh period, this is usually 15.6 us. For example, a system operating at 50 MHz using typical 15.6 us SDRAM, the refresh_period would be 50,000,000 x 0.0000156 = 780.
progSync	The number of clock cycles between the system SYNC pulse and the first word of TDM Rx data. This is the value that is written into Epoch register 0x054. For normal operation, set to three.

Return Value

An integer is returned if the intialization was successful. If successful, **INIT_OK** is returned. If not successful, **INIT_FAIL** is returned.

Pre-requisites

The system sync pulse *must* be off prior to using this function. If the sync pulse is on, the Epoch attempts to access the SDRAM for refresh and maintenance tasks. The Epoch sync pulse is switched on and off using register 0x03c. The user *must* ensure that the sync pulse is switched back on prior to receiving network traffic.

Usage

```
U32 refresh_period, prog_sync;
epoch_t pEpoch;
int i_stat;

//assuming RCPs and block pointers have been initialized.
//set progSync for normal operation
prog_sync = 3;

//initialize SDRAM operating at 50MHz
refresh_period = 780;
i_stat = epochSDRAMInit(&pEpoch, refresh_period, prog_sync);
if(i_stat == INIT_FAIL){
      printf("\nThere was an error while initializing the Queue SDRAM control
                pointers or testing the buffer data SDRAM. \nPlease
                Investigate.");
}
else{
            printf("\nThe SDRAM was initialized.");
}
```

epochQueuePtrMemInit

Syntax

```
int epochQueuePtrMemInit(epoch_t *pEpoch,
                         U16 usePort);
```

Description

This function initializes the 128-entry internal Queue Pointer SRAM.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure is initialized with information required for RCP address to RCP virtual address translation.
usePort	This is the 16-bit port bitmap. There are 128 Queue pointers that are used by Epoch for sixteen ports each having eight queues. Each physical port (0 – 15) may have its pointers initialized. The appropriate bit is set to 1 if the port that is indicated is to be initialized. If the bit is set to 0, the port is not affected by the function. For example, bit 0 = port 0, bit 1 = port 1, bit 2 = port 2, and so on. For normal initialization of sixteen ports this parameter should be set to 0xFFFF.

Return Value

An integer is returned if the intialization was successful. If successful, **INIT_OK** is returned. If not successful, **INIT_FAIL** is returned.

Pre-requisites

None.

Usage

```
U32 useport = 0xFFFF;
epoch_t pEpoch;
int i_stat;

//initialize all ports
i_stat = epochQueuePtrMemInit(&pEpoch, useport);
if(i_stat == INIT_FAIL){
      print("\nThere was an error while initializing the Queue
               pointers.\nPlease investigate.");
}
else {
      printf("\nThe Queue Pointers were initialized.");
}
```

epochPacketPtrInit

Syntax

```
int epochPacketPtrInit(epoch_t *pEpoch,
                       U32 numBufs);
```

Description

This function initializes the external 64K (65,536) location Packet Pointer SRAM.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure is initialized with information required for RCP address to RCP virtual address translation.
numBufs	This is the number of Packet pointers to be initialized. The Epoch has 65,536 Packet pointers that allow it to store 65,536 64-byte packets. This parameter indicates how many of the 65,536 Packet pointers are to be initialized. The parameter should be a value from 0 through 65,535 (0x0000 through 0xFFFF). The relationship between the Queue pointers and Packet pointers *must* be maintained or an error occurs. This means that there *must* be enough packet pointers initialized to satisfy the number of Queue pointers that are initialized. Therefore *numBufs must* always be greater than the number of ports specified by *usePort* times eight.

Return Value

An integer is returned if the intialization was successful. If successful, **INIT_OK** is returned. If not successful, **INIT_FAIL** is returned.

Pre-requisites

The ***epochQueuePtrMem()*** function must be used prior to using this function in order to set the pktPtr member of the ***blockSort_t*** structure. If the ***epochQueuePtrMem()*** function is not called prior to using this function, then an error may result.

Usage

```
U32 numbufs = 0xFFFF, useport = 0xFFFF;
epoch_t pEpoch;
int i_stat;

epochQueuePtrMemInit(&pEpoch, useport);
//initialize all buffers
i_stat = epochPacketPtrMemInit(&pEpoch, numbufs);
if(i_stat == INIT_FAIL){
      print("\nThere was an error while intializing the packet
               pointers.\nPlease investigate.");
}
else {
      printf("\nThe packet pointers were initialized.");
}
```

epochPacketPtrClear

Syntax

```
int epochPacketPtrClear(epoch_t *pEpoch,
                        U32 numBufs);
```

Description

This function clears the external 64K (65,536) location Packet Pointer SRAM, setting the entries to 0.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure is initialized with information required for RCP address to RCP virtual address translation.
numBufs	This is the number of Packet pointers to be cleared. The Epoch has 65,536 Packet pointers that allow it to store 65,536 64-byte packets. This parameter indicates how many of the 65,536 Packet pointers are to be set to 0. The parameter should be a value from 0 through 65,535 (0x0000 through 0xFFFF). The relationship between the Queue pointers and Packet pointers *must* be maintained or an error occurs. This means that there *must* be enough packet pointers chosen with this parameter to satisfy the number of Queue pointers that are initialized. Therefore *numBufs must* always be greater than the number of ports specified by *usePort* times eight.

Return Value

An integer is returned if the operation was successful. If successful, **INIT_OK** is returned. If not successful, **INIT_FAIL** is returned.

Pre-requisites

The ***epochQueuePtrMem()*** function must be used prior to using this function in order to set the pktPtr member of the ***blockSort_t*** structure. If the ***epochQueuePtrMem()*** function is not called prior to using this function, then an error may result.

Usage

```
U32 numbufs = 0xFFFF, useport = 0xFFFF;
epoch_t pEpoch;
int i_stat;

epochQueuePtrMemInit(&pEpoch, useport);
//initialize all buffers
i_stat = epochPacketPtrMemClear(&pEpoch, numbufs);
if(i_stat == INIT_FAIL){
      print("\nThere was an error while clearing the packet
               pointers.\nPlease investigate.");
}
else {
      printf("\nThe packet pointers were set to 0.");
}
```

LOWER-LEVEL API

The functions in this section are responsible for the lower-level operations on the Routing Table entries. This involves adding and removing RCP and SRAM entries. Routines are also provided for conversion from RCP page address and physical address format to RCP virtual contiguous address format. The files in which they may be found and the page in this manual where they can be located is shown for easy reference. Low- and high-level functions are identified by (H) = high-level, (L) = low-level.

Table 3-7: Lower-Level API Functions

API Function Name	Purpose	File Name	Page
AtoCS (L)	Converts an input RCP physical address to the appropriate RCP output Chip Select.	epochLib.c	3-68
AtoPA (L)	Converts an input RCP physical address to the appropriate RCP page address value.	epochLib.c	3-70
PAtoA (L)	Converts an input page address value to the appropriate RCP physical address.	epochLib.c	3-72
epochRCPBinarySearch (L)	Performs a binary search in the RCP for the 64-bit input data.	epochTable.c	3-74
epochRCPDeleteByAddr (L)	Deletes a 64-bit entry from the RCP memory location specified by an address input parameter.	epochLib.c	3-76
epochRCPInitAddrTrans (L)	Initializes the address translation pointers.	epochLib.c	3-77
epochRCPMoveEntry (L)	Moves a 64-bit RCP entry from a source RCP physical location to a destination location.	lpusort.c	3-79
epochRCPNextFree (L)	Finds the next free RCP location.	epochTable.c	3-80
epochRCPReadEntry (L)	Reads a 64-bit entry from the RCP memory location specified by an address input parameter.	epochLib.c	3-81
epochRCPSRAMRead (L)	Reads the 16-bit RCP associated data SRAM entry specified by an address input parameter. Used for SRAM locations 0xFFFF or less.	epochLib.c	3-82
epochRCPSRAMReadDir (L)	Reads the 16-bit RCP associated data SRAM entry specified by an address input parameter. Used for SRAM locations 0x10000 or greater.	epochLib.c	3-83
epochRCPSRAMWrite (L)	Writes the 16-bit RCP associated data SRAM entry specified by an address input parameter. Used for SRAM locations 0xFFFF or less.	epochLib.c	3-84
epochRCPSRAMWriteDir (L)	Writes the 16-bit RCP associated data SRAM entry specified by an address input parameter. Used for SRAM locations 0x10000 or greater.	epochLib.c	3-85
epochRCPWriteEntry (L)	Writes a 64-bit entry into the RCP at the memory location specified by an address input parameter.	epochLib.c	3-86
epochRCPWrite32 (L)	Function for writing instructions in the form of op-code and 32-bit data to RCP.	epochLib.c	3-87

AtoCS

Syntax

```
U32 AtoCS(epoch_t *pEpoch,
          U16 addr);
```

Description

This function converts the input RCP virtual contiguous address to the appropriate RCP chip select op-code and physical address combination. The physical address is the address within the device specified by the chip select op-code.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
addr	The RCP virtual contiguous address of the entry that is to be accessed. This location is found in one of up to four devices when using an RCP chain. The function take this input and returns which of the four devices the address is located along with physical address within the device specified.

Return Value

The contents of the appropriate ***chipSelectLkup[n]*** member of the ***blockSort_t*** structure converts the input *addr.* If the members were initialized properly prior to using this function, the chip select information for RCP 0, 1, 2 or 3 should be returned. This indicates which of the four possible RCPs in a chain should be accessed when attempting to read or write the RCP address specified by *addr.* The physical address within the device specified is also returned. The return value is decoded as follows:

Bits	Explanation
[12:0]	This is the RCP Physical address of the virtual address given as an input to the function. It uses the standard Epoch format and shows physical address within a specific RCP.
[16:13]	This is the RCP chip select code that is required to select one of a chain of up to four devices. It is decoded as follows: 1110b = First device 1101b = Second device 1011b = Third device 0111b = Fourth device

Pre-requisites

The ***chipSelectLkup[n]***, ***pageAddrLkup[n]***, and ***pageSize[n]***, members of the ***blockSort_t*** structure *must* be initialized to the correct values. **epochRCPInit()** automatically does this if it is executed prior to using this function.

Usage

```
U16 vir_addr = 5000;
U32 RCP_opcode, cs_addr;
epoch_t pEpoch;
int i_stat;

//assuming block pointers have been initialized.

i_stat = epochRCPInit(&pEpoch);
if(i_stat == INIT_OK){
      printf(“\nThe RCP chain was initialized.”);
      printf(“\nAdding an entry to RCP at virtual address: 0x%x”,
              vir_addr);
      cs_addr = AtoCS(&pEpoch, vir_addr);
      RCP_opcode = 0x80000 | cs_addr; // /av low, dsc low, /vb hi
      //write low 32 bits to RCP
      epochRCPWrite32(&pEpoch, 0x12345678, RCP_opcode);
      RCP_opcode = 0x20000 | cs_addr; // /av low, dsc hi, /vb low
      //write high 32 bits to RCP
      epochRCPWrite32(&pEpoch, 0x87654321, RCP_opcode);
}
else{
      printf(“\nERROR : Unable to initialize RCP chain.”);
}
```

AtoPA

Syntax

```
U16 AtoPA(epoch_t *pEpoch,
          U16 addr);
```

Description

This function converts an input RCP virtual contiguous address to the corresponding RCP page address and physical address combination. This function is primarily used by higher-level functions that read and write entries from and to the associated data SRAM. The function is required, because the SRAM entries *must* be accessed using the standard page and physical address format.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
addr	The RCP virtual contiguous address of the entry that is to be converted. This location is found in one of up to 4 devices when using an RCP chain. The function takes this input and returns the physical address and page address.

Return Value

The contents of the appropriate ***pageAddrLkup[n]*** member of the ***blockSort_t*** structure converts the input *addr*. The page address and the physical address of the entry are returned. If the members of the structure were initialized properly prior to using this function, the page address are bits [14:13] and the physical address within that device are bits [12:0].

Pre-requisites

The ***pageAddrLkup[n]*** and ***pageSize[n]*** members of the ***blockSort_t*** structure *must* be initialized to the correct values. ***epochRCPInit()*** automatically does this if it is executed prior to using this function.

Usage

```
U16 p_addr, vir_addr = 5000;
U16 page_addr;
U32 RCP_opcode;
epoch_t pEpoch;
int i_stat;

//assuming block pointers have been initialized.

i_stat = epochRCPInit(&pEpoch);
if(i_stat & INIT_OK){
      printf("\nThe RCP chain was initialized.");
      printf("\nAdding an entry to RCP at virtual address: 0x%x",
              vir_addr);
      p_addr = AtoPA(&pEpoch, vir_addr);
      page_addr = p_addr >> 13;
      printf("\nThe page address of virtual address : 0x%x is %d",
              vir_addr, page_addr);
}
else{
      printf("\nERROR : Unable to initialize RCP chain.");
}
```

PAtoA

Syntax

```
U16 PAtoA(epoch_t *pEpoch,
          U16 addr);
```

Description

This function converts an input RCP page address and physical address combination to the corresponding RCP virtual contiguous address. This function is primarily used by higher-level functions that read and write entries from and to the RCP database.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
addr	The RCP page address and physical address of the entry that is to be converted. The page address are bits [14:13] and the physical address within that device are bits [12:0]. The function takes this input and returns the RCP contiguous virtual address.

Return Value

The contents of the appropriate ***pageAddrLkup[n]*** member of the ***blockSort_t*** structure converts the input *addr*. If the members were initialized properly prior to using this function, the RCP contiguous virtual address is returned.

Pre-requisites

The ***pageAddrLkup[n]*** and ***pageSize[n]*** members of the ***blockSort_t*** structure *must* be initialized to the correct values. **epochRCPInit()** automatically does this if it is executed prior to using this function.

Usage

```
#define CMP_M (0x1L<<15)
#define CMP_ADDR_MASK 0x7FFFL

rcp_t parent, child;
microflow_t this_flow;
U32 cmp_result;
U16 vir_addr, pa_addr;
epoch_t pEpoch;
int i_stat;

//assuming block pointers have been initialized.
this_flow.sa = 0xbbccaa04;
this_flow.da = 0xbbccaa05;
this_flow.sp = 23;
this_flow.dp = 1000;
this_flow.ifc = 3;

i_stat = epochRCPInit(&pEpoch);
//assuming the L4 Table has entries
if(i_stat == INIT_OK){
      parent = L4CreateParent(&pEpoch, &this_flow);
      cmp_result = epochRCPBinarySearch(&pEpoch, &parent, 0);
      if (cmp_result & CMP_M){
            pa_addr = (U16)(cmp_result & CMP_ADDR_MASK);
            vir_addr = PAtoA(&pEpoch, pa_addr);
            child = L4CreateChild(&pEpoch, &this_flow, vir_addr);
            cmp_result = epochRCPBinarySearch(&pEpoch, &child, 0);
            if (cmp_result & CMP_M){ //child matches
                  printf("\nThere was a child MATCH at virtual address
                        0x%x", vir_addr;
            }
            else{
                  printf("\nThere was no child MATCH.");
            }
      }
      else {
            printf("\nThere was no parent MATCH.");
      }
}
else{
      printf("\nERROR : Unable to initialize RCP chain.");
}
```

epochRCPBinarySearch

Syntax

```
static U32 epochRCPBinarySearch(epoch_t *pEpoch,
                                rcp_t *pWord,
                                U16 mask);
```

Description

This function performs a RCP binary search for the input ***rcp_t*** through the RCP mask register specified by the input mask.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
**pWord*	A pointer to the 64-bit RCP entry. The parameter should be a pointer to a variable of the structure ***rcp_t***. The variable has two members: rcp_t.hi = Bits 63:32 of the RCP entry rcp_t.lo = Bits 31:0 of the RCP entry
mask	The mask register number that is to be used if the search is to be done using a mask. If no mask register is to be used, this input should be 0.

Return Value

The contents of the Epoch CRI register 0x134 is returned. This shows relevant information regarding the search for any entries. The information is as follows:

Bits	Explanation
[12:0]	This is the Index of any matching entries found during the search. It uses the standard Epoch format and shows physical address within a specific RCP. Only valid if bit 15 = 1 or bit 16 = 1.
[14:13]	Page address of any matching entries found during the search. This corresponds to the Page Address values set during initialization. The usual format is: 00 = First device 01 = Second device 10 = Third device 11 = Fourth device.
15	Match Flag. If this bit = 1, there was a MATCH and therefore an entry matching the input was found.
16	Multiple Match Flag. If this bit = 1, there was a Multiple MATCH and therefore more than one matching entry was found. Bits [14:0] specify the highest priority entry matched.

Pre-requisites

None.

Usage

```
#define CMP_M (0x1L<<15)
#define CMP_ADDR_MASK 0x7FFFL

rcp_t parent, child;
microflow_t this_flow;
U32 cmp_result;
U16 vir_addr, pa_addr;
epoch_t pEpoch;
int i_stat;

//assuming block pointers have been initialized.
this_flow.sa = 0xbbccaa04;
this_flow.da = 0xbbccaa05;
this_flow.sp = 23;
this_flow.dp = 1000;
this_flow.ifc = 3;

i_stat = epochRCPInit(&pEpoch);
//assuming the L4 Table has entries
if(i_stat & INIT_OK){
      parent = L4CreateParent(&pEpoch, &this_flow);
      cmp_result = epochRCPBinarySearch(&pEpoch, &parent, 0);
      if (cmp_result & CMP_M){
            pa_addr = (U16)(cmp_result & CMP_ADDR_MASK);
            vir_addr = PAtoA(&pEpoch, pa_addr);
            child = L4CreateChild(&pEpoch, &this_flow, vir_addr);
            cmp_result = epochRCPBinarySearch(&pEpoch, &child, 0);
            if (cmp_result & CMP_M){ //child matches
                  printf("\nThere was a child MATCH at virtual address
                        0x%x", vir_addr;
            }
            else{
                  printf("\nThere was no child MATCH.");
            }
      }
      else {
            printf("\nThere was no parent MATCH.");
      }
}
else{
      printf("\nERROR : Unable to initialize RCP chain.");
}
```

epochRCPDeletebyAddr

Syntax

```
rcpResultStatus_t epochRCPDeleteByAddr(epoch_t *pEpoch,
                                       U16 addr);
```

Description

This function deletes the 64-bit RCP entry specified by the input contiguous virtual address.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
addr	The RCP contiguous virtual address of the entry that is to be deleted from the RCP database.

Return Value

A variable in the form of the structure ***rcpResultStatus_t*** is returned. This indicates any relevant information regarding the function's success or failure. The RCP physical address of the entry deleted is returned in the ***addr*** member.

Pre-requisites

The ***chipSelectLkup[n]*** members of the ***blockSort_t*** structure *must* be initialized to the correct values. **epochRCPInit()** automatically does this if it is executed prior to using this function.

Usage

```
U16 vir_addr = 5000;
epoch_t pEpoch;
rcpResultStatus_t r_stat;

//assuming the RCPs and block pointers have been initialized
rstat = epochRCPDeleteByAddr(&pEpoch, vir_addr);
printf("\nThe entry at 0x%x has been deleted.", rstat.addr);
```

epochRCPInitAddrTrans

Syntax

```
static int epochRCPInitAddrTrans(epoch_t *pEpoch,
                                 U16 numRcp);
```

Description

This function initializes the ***blockSort_t*** members ***pageAddr[n]***, ***pageAddrLkup[n]***, and ***chipSelectLkup[n]*** that are used by the functions **epochRCPDeleteByAddr**, **AtoCS**, **AtoPA**, and **PAtoA**. It uses the ***pageSize*** information found in the ***blockSort*** structure to determine page address boundaries and chip select values, etc.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
numRcp	The number of RCPs that are chained together to form one contiguous virtual address space.

Return Value

The number of RCP entries found in the contiguous virtual address space.

Pre-requisites

The ***pageSize[n]*** members of the ***blockSort_t*** structure *must* be initialized with the size of each RCP in the RCP chain.

Usage

```
#define CS_NULL 0xFL<<13
#define DEC_AR 0x25L
#define ALL_RCPS 0x00000000L
#define AV_HIGH  ((0x1L<<18))
#define AV_LOW  (0x0L<<18)
#define RCPWrDataReg 0x120
#define RCPWrOpReg 0x124
#define RCPRdDataReg 0x11C

U16 num_rcp, address_space;
U32 chip_select = CS_NULL;
U32 i, ThisRCPSize ;
U32 rcp_ar[4];
epoch_t pEpoch;

//assuming block pointers have been initialized.

num_rcp = 4;
//broadcast decrement address to determine RCP sizes
epochMlsRegWrite(&pEpoch, RCPWrDataReg, DEC_AR);
epochMlsRegWrite(&pEpoch, RCPWrOpReg,(ALL_RCPS | AV_HIGH));
for (i=0; i< num_rcp; i++){
      chip_select = (CS_NULL) ^ (1L << (i+13));
      //read address register to read out RCP size
      epochMlsRegWrite(&pEpoch, RCPWrDataReg,RD_AR | (1L<<13));
      epochMlsRegWrite(&pEpoch, RCPWrOpReg,(chip_select | AV_HIGH));
      epochMlsRegWrite(&pEpoch, RCPRdOpReg, (chip_select | AV_LOW  ));
      ThisRCPSize = epochMLsRegRead(&pEpoch, RCPRdDataReg);
      if(ThisRCPSize == 0xFFFFFFFFL){//no RCP found
            pEpoch.pageSize[i] = 0L;
      }
      else{
            pEpoch.pageSize[i] = ThisRCPSize + 1;
      }
}

//initialize address conversion
address_space = epochRCPInitAddrTrans(&pEpoch, num_rcp);
printf("\nThe Address translation info has been initialized.");
printf("\nThe total address space found is %d entries",address_space);
```

epochRCPMoveEntry

Syntax

```
rcpResultStatus_t epochRCPMoveEntry(epoch_t *pEpoch,
                                    U16 scrAddr,
                                    U16 dstAddr);
```

Description

This function moves the 64-bit RCP entry specified by the input contiguous virtual address "scrAddr" to the input contiguous virtual address "dstAddr".

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
scrAddr	The RCP contiguous virtual address of the entry that is to be moved.
dstAddr	The RCP contiguous virtual address of the location where the entry specified by "fraddr" is to be moved to.

Return Value

A variable in the form of the structure ***rcpResultStatus_t*** is returned. This indicates any relevant information regarding the function's success or failure. The RCP physical address of the where the entry was moved from is returned in the ***addr*** member.

Pre-requisites

The ***chipSelectLkup[n]*** members of the ***blockSort_t*** structure *must* be initialized to the correct values. **epochRCPInit()** automatically does this if it is executed prior to using this function.

Usage

```
U16 from_addr = 5000, to_addr = 5005;
epoch_t pEpoch;

//assuming the RCPs and block pointers have been initialized
epochRCPMoveEntry(&pEpoch, from_addr, to_addr);
printf("\nThe entry at 0x%x has been moved to 0x%x.",
         from_addr, to_addr);
```

epochRCPNextFree

Syntax

```
static U16 epochRCPNextFree(epoch_t *pEpoch);
```

Description

This function returns the contiguous virtual address of the next free location in the RCP database.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.

Return Value

The RCP contiguous virtual address of the next free location in the RCP database.

Pre-requisites

The ***pageAddrLkup[n]*** members of the ***blockSort_t*** structure *must* be initialized to the correct values. **epochRCPInit()** automatically does this if it is executed prior to using this function.

Usage

```
U16 nf_addr;
epoch_t pEpoch;

//assuming the RCPs and block pointers have been initialized
nf_addr = epochRCPNextFree(&pEpoch);
printf("\nThe next free location is at 0x%x", nf_addr);
```

epochRCPReadEntry

Syntax

```
rcp_t epochRCPReadEntry(epoch_t *pEpoch,
                        U16 addr);
```

Description

This function returns the 64-bit RCP entry specified by the input contiguous virtual address.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
addr	The RCP contiguous virtual address of the entry that is to be read from the RCP database.

Return Value

A variable in the form of the structure ***rcp_t*** is returned. This is the RCP entry that is read from the input address.

- rcp_t.hi = Bits 63:32 of the RCP entry
- rcp_t.lo = Bits 31:0 of the RCP entry

Pre-requisites

The ***chipSelectLkup[n]*** members of the ***blockSort_t*** structure *must* be initialized to the correct values. **epochRCPInit()** automatically does this if it is executed prior to using this function.

Usage

```
rcp_t this_word;
U16 addr = 5000;
epoch_t pEpoch;

//assuming the RCPs and block pointers have been initialized
epochRCPReadEntry(&pEpoch, addr);
printf("\nThe RCP entry at 0x%x is:", addr);
printf("\n[63:32] : 0x%lx [31:0] : 0x%lx", this_word.hi, this_word.lo);
```

epochRCPSRAMRead

Syntax

```
U16 epochRCPSRAMRead(epoch_t *pEpoch,
                     U16 addr);
```

Description

This function returns the 16-bit associated data SRAM entry specified by the input address. This function is used when the user wishes to access the lower 32K portion of the associated data SRAM.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
addr	The contiguous virtual address of the entry that is to be read from the associated data SRAM database. The input can be any valid value from 0x0000 through 0xFFFF.

Return Value

The 16-bit value found in the associated data SRAM at the location specified by the input parameter *addr.*

Pre-requisites

None.

Usage

```
rcp_t this_word;
U16 addr = 5000;
U16 SRAM_entry;
epoch_t pEpoch;

//assuming the RCPs and block pointers have been initialized
this.word = epochRCPReadEntry(&pEpoch, addr);
printf("\nThe RCP entry at 0x%x is:", addr);
printf("\n[63:32] : 0x%lx [31:0] : 0x%lx", this_word.hi, this_word.lo);
SRAM_entry = epochRCPSRAMRead(&pEpoch, addr);
printf("\nThe SRAM entry at 0x%x is 0x%x:", addr, SRAM_entry);
```

epochRCPSRAMReadDir

Syntax

```
U16 epochRCPSRAMReadDir(epoch_t *pEpoch,
                        U32 addr);
```

Description

This function returns the 16-bit associated data SRAM entry specified by the input address. This function is used when the user wishes to access the upper 64K portion of the associated data SRAM.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
addr	The contiguous virtual address of the entry that is to be read from the associated data SRAM database. The input can be any valid value from 0x10000 through 0x1FFFF.

Return Value

The 16-bit value found in the associated data SRAM at the location specified by the input parameter *addr.*

Pre-requisites

None.

Usage

```
#define DEF_F_START 0x10000

U16 flowhandle;
U32 addr = 0x0234;
epoch_t pEpoch;

//look up default flow handle
flowhandle =  epochRCPSRAMReadDir(&pEpoch,(DEF_F_START | addr));
printf("\nThe flow handle for UDP / TCP port number 0x%x is :
             0x%x", addr, flowhandle);
```

epochRCPSRAMWrite

Syntax

```
rcpResultStatus_t epochRCPSRAMWrite(epoch_t *pEpoch,
                                    U16 addr,
                                    U16 data);
```

Description

This function writes the 16-bit associated data SRAM entry specified by the input "data" in the location specified by "addr". This function is used when the user wishes to write an entry into the lower 64K portion of the associated data SRAM.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
addr	The contiguous virtual address of the entry that is to be written into the associated data SRAM database. The input can be any valid value from 0x0000 through 0xFFFF.
data	The 16-bit associated data entry that is to be written into the associated data SRAM database.

Return Value

A variable in the form of the structure ***rcpResultStatus_t*** is returned. This indicates any relevant information regarding the function's success or failure. The SRAM address of the where the entry was written is returned in the "addr" member. The data that was written is returned in the "data" member. If the data was successfully written into the location specified, **RCP_OK** is returned in the "status" member. If the function was unsuccessful, **RCP_FAIL** is returned.

Pre-requisites

None.

Usage

```
U16 addr = 0x0234;
U16 SRAM_entry 0x1234;
epoch_t pEpoch;
rcpResultStatus_t r_stat;

//assuming the RCPs and block pointers have been initialized
r_stat = epochRCPSRAMWrite(&pEpoch, addr, SRAM_entry);
if(r_stat.status == RCP_OK){
      printf("\nThe SRAM entry was written to address : 0x%x", addr);
}
else{
      printf("\nERROR: Unable to write entry to SRAM");
}
```

epochRCPSRAMWriteDir

Syntax

```
rcpResultStatus_t epochRCPSRAMWriteDir(epoch_t *pEpoch,
                                       U32 addr);
                                       U16 data,
```

Description

This function writes the 16-bit associated data SRAM entry specified by the input "data" in the location specified by "addr". This function is used when the user wishes to write an entry into the upper 64K portion of the associated data SRAM.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
addr	The contiguous virtual address of the entry that is to be written into the associated data SRAM database. The input can be any valid value from 0x10000 through 0x1FFFF.
data	The 16-bit associated data entry that is to be written into the assoc. data SRAM database.

Return Value

A variable in the form of the structure ***rcpResultStatus_t*** is returned. This indicates any relevant information regarding the function's success or failure. The SRAM address of the where the entry was written is returned in the ***addr*** member. The data that was written is returned in the ***data*** member. If the data was successfully written into the location specified, **RCP_OK** is returned in the ***status*** member. If the function was unsuccessful, **RCP_FAIL** is returned.

Pre-requisites

None.

Usage

```
#define DEF_F_START 0x10000
U16 flowhandle = 0x0007;
U32 addr = 0x0234;
epoch_t pEpoch;
rcpResultStatus_t r_stat;
//assuming the RCPs and block pointers have been initialized
r_stat = epochRCPSRAMWriteDir(&pEpoch, DEF_F_START | addr, flowhandle);
if(r_stat.status == RCP_OK){
      printf("\nThe SRAM entry was written to address : 0x%lx",
              DEF_F_START | addr);
}
else{
      printf("\nERROR: Unable to write entry to SRAM");
}
```

epochRCPWriteEntry

Syntax

```
rcpResultStatus_t epochRCPWriteEntry(epoch_t *pEpoch,
                                     U16 addr,
                                     rcp *pEntry);
```

Description

This function writes the 64-bit RCP entry pointed to by the input "*pEntry" into the location specified by the input contiguous virtual address "addr".

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
addr	The RCP contiguous virtual address of the entry that is to be written into the RCP database.
**pEntry*	A pointer to the 64-bit RCP entry that is to be written. The parameter should be a pointer to a variable of the structure ***rcp_t***. The variable has two members: rcp_t.hi = Bits 63:32 of the RCP entry rcp_t.lo = Bits 31:0 of the RCP entry

Return Value

A variable in the form of the structure ***rcpResultStatus_t*** is returned. This indicates any relevant information regarding the function's success or failure. The RCP contiguous virtual address of the where the entry was written is returned in the ***addr*** member. If the data was successfully written into the location specified, **RCP_OK** is returned in the ***status*** member. If the function was unsuccessful, **RCP_FAIL** is returned.

Pre-requisites

None.

Usage

```
rcp_t this_word;
U16 addr = 5000;
epoch_t *pEpoch;
rcpResultStatus_t r_stat;

//assuming the RCPs and block pointers have been initialized
this_word.lo = 0x12345678;
this_word.hi = 0x87654321;
r_stat = epochRCPWriteEntry(&pEpoch, addr, &this_word);
if(r_stat.status == RCP_OK){
      printf("\nThe RCP entry was written to address : 0x%x", addr);
}
else{
      printf("\nERROR: Unable to write entry to RCP");
}
```

epochRCPWrite32

Syntax

```
int epochRCPWrite32(epoch_t *pEpoch,
                    U32 data,
                    U32 opCode);
```

Description

This function writes a 32-bit data word to the RCP chain along with an instruction specified by the input op-code. The op-code instructs the RCP chain to act accordingly when dealing with the 32-bit word. The op-code is written to the Epoch RCP op-code register 0x124 and is described in the Epoch data sheet. The MUAC RCP Data Sheet contains a list and description of all the possible op-codes.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.
data	The 32-bit data word that is to be written to the RCP database. This data is dealt with according to the instruction specified by the op-code.
opCode	The op-code that is to be written to the RCP database. The op-code is the value that will be written into Epoch register 0x124. This op-code causes the RCP chain to write the 32-bit data word accordingly.

Return Value

The function returns 0.

Pre-requisites

None.

Usage

```
U16 cs_addr, vir_addr = 5000;
U32 RCP_opcode;
epoch_t *pEpoch;
int i_stat;

//assuming block pointers have been initialized.

i_stat = epochRCPInit(&pEpoch);
if(i_stat == INIT_OK){
      printf("\nThe RCP chain was initialized.");
      printf("\nAdding an entry to RCP at virtual address: 0x%x",
              vir_addr);
      cs_addr = AtoCS(&pEpoch, vir_addr);
      RCP_ocode = 0x80000 | cs_addr; // /av low, dsc low, /vb hi
      //write low 32 bits to RCP
      epochRCPWrite32(&pEpoch, 0x12345678, RCP_opcode);
      RCP_ocode = 0x20000 | cs_addr; // /av low, dsc hi, /vb low
      //write high 32 bits to RCP
      epochRCPWrite32(&pEpoch, 0x87654321, RCP_opcode);
}
else{
      printf("\nERROR : Unable to initialize RCP chain.");
}
```

DEBUG API

The functions in this section are responsible for debug operations on the Routing Table entries. This involves dumping all valid entries from the Routing Table into a file, and testing for leaking packet pointers. The files in which they may be found and the page in this manual where they can be located is shown for easy reference. Low- and high-level functions are identified by (H) = high-level, (L) = low-level.

Table 3-8: Debug API Functions

API Function Name	Purpose	File Name	Page
epochCheckLeakedPacketPointers (H)	Checks that all Packet pointers are still linked properly.	epochLib.c	3-90

epochCheckLeakedPacketPointers

Syntax

```
int epochCheckLeakedPacketPointers(epoch_t *pEpoch);
```

Description

This function reads the next free packet pointer and uses this to check that all the packet pointers are linked together properly. This function also checks that all the queue pointers are linked together properly.

Input Parameters

**pEpoch*	A pointer to the ***epoch_t*** structure. This structure *must* contain information about the Epoch and the Routing Table.

Return Value

If the packet pointers and queue pointers are okay the function returns 0, otherwise –1 is returned.

Pre-requisites

None.

Usage

```
epoch_t pEpoch;

//assuming the RCPs and block pointers have been initialized
if(epochCheckLeakedPacketPointers(&pEpoch) == 0){
      printf(“\nThe packet pointers are okay.”);
}
else{
      printf(“\nThe packet pointers have a leak.”);
}
```

Index

F

I

L

M

O

MUSIC SEMICONDUCTORS

http://www.music-ic.com
email: info@music-ic.com

Worldwide Headquarters
MUSIC Semiconductors
2290 N. First St., Suite 201
San Jose, CA 95131
USA
Tel: 408 232-9060
Fax: 408 232-9201
USA Only: 800 933-1550 Tech Support
888 226-6874 Product Info

Asian Headquarters
MUSIC Semiconductors
Special Export Processing Zone
Carmelray Industrial Park
Canlubang, Calamba, Laguna
Philippines
Tel: +63 49 549-1480
Fax: +63 49 549-1024
Sales Tel/Fax: +632 723-6215

European Headquarters
MUSIC Semiconductors
P. O. Box 184
6470 ED Eygelshoven
The Netherlands
Tel: +31 43 455-2675
Fax: +31 43 455-1573