PM7324

S/UNI-ATLAS

Software Programmer's Guide
&
Example Software

Released

Issue 3: Feb. 2002

Legal Information

Copyright

PMC-1980585 (P3), ref PMC-1971154 (P7)

Disclaimer

Trademarks

Patents

The technology discussed is protected by one or more of the following Patents:

U.S. Patent No. 5,668,797; 5,815,737; 6,108,303; 6,128,766

Canadian Patent No. 2,164,546; 2,167,757; 2,209,887

UK Patent No. 2,301,913

Relevant patent applications and other patents may also exist.

Contacting PMC-Sierra

PMC-Sierra
8555 Baxter Place Burnaby, BC
Canada V5A 4V7

Tel: (604) 415-6000
Fax: (604) 415-6200

Document Information: document@pmc-sierra.com
Corporate Information: info@pmc-sierra.com
Technical Support: apps@pmc-sierra.com
Web Site: http://www.pmc-sierra.com

Revision History

Issue No.	Issue Date	Details of Change
3	Feb. 2002	fixed typos Add sections 5.2.2.1 and 7.1
2	January 1999	Add section for programming OAM functionality.
1	September 4,	Document created

Table of Contents

List of Figures

List of Tables

1 References

- PMC-1971154, "S/UNI-ATLAS Data sheet", Issue 7, February 1998
- PMC-960345, "RCMP Software Driver Detailed Design", Issue 3, July 1996
- ITU-T Recommendation I.610, "B-ISDN Operation and Maintenance Principles and Functions", 1998 Living List
- ATM Forum, "User Network Interface Specification", V3.1

2 Glossary

Term	Definition
AIS	Alarm Indication Signal. An OAM cell that indicates to downstream entities that the connection is not receiving cells.
CC	Continuity Check. An OAM cell sent through the network so that downstream entities may differentiate between a failure and periods of low user cell traffic.
FM	Fault Management. The mechanism used by the network to inform management entities and other network equipment of faults within the network.
OAM	Operations, Administration, and Maintenance. The maintenance of VCs within the network.
PM	Performance Management. The mechanism used by the network to monitor the performance parameters of a particular VC.
RDI	Remote Defect Indication. An OAM cell sent to an upstream entity at the OAM flow endpoint to indicate that a connection in the flow is not receiveing cells.
VC	Virtual Connection. This refers to either a Virtual Path Connection (VPC) or a Virtual Channel Connection (VCC) within a physical link.
VCC	Virtual Channel Connection. A virtual connection between two network elements. A virtual channel connection is normally a constituent member of a virtual path connection, where the VPC consists of one or more VCCs. This is sometimes known as an F5 connection.
VPC	Virtual Path Connection. A virtual connection between two network elements. A virtual path connection may span one or more physical links. This is sometimes known as an F4 connection.

3 Introduction

This document provides software routines that may be used to ease the integration of the PM7234 S/UNI-ATLAS into an application, such as a switch or add/drop multiplexer. It is intended for software and system designers who are using or planning to use the ATLAS, as well as for audiences in general who would like to have a better understanding of the VC identification algorithm that the ATLAS uses, or the powerful OAM capabilities of the ATLAS.

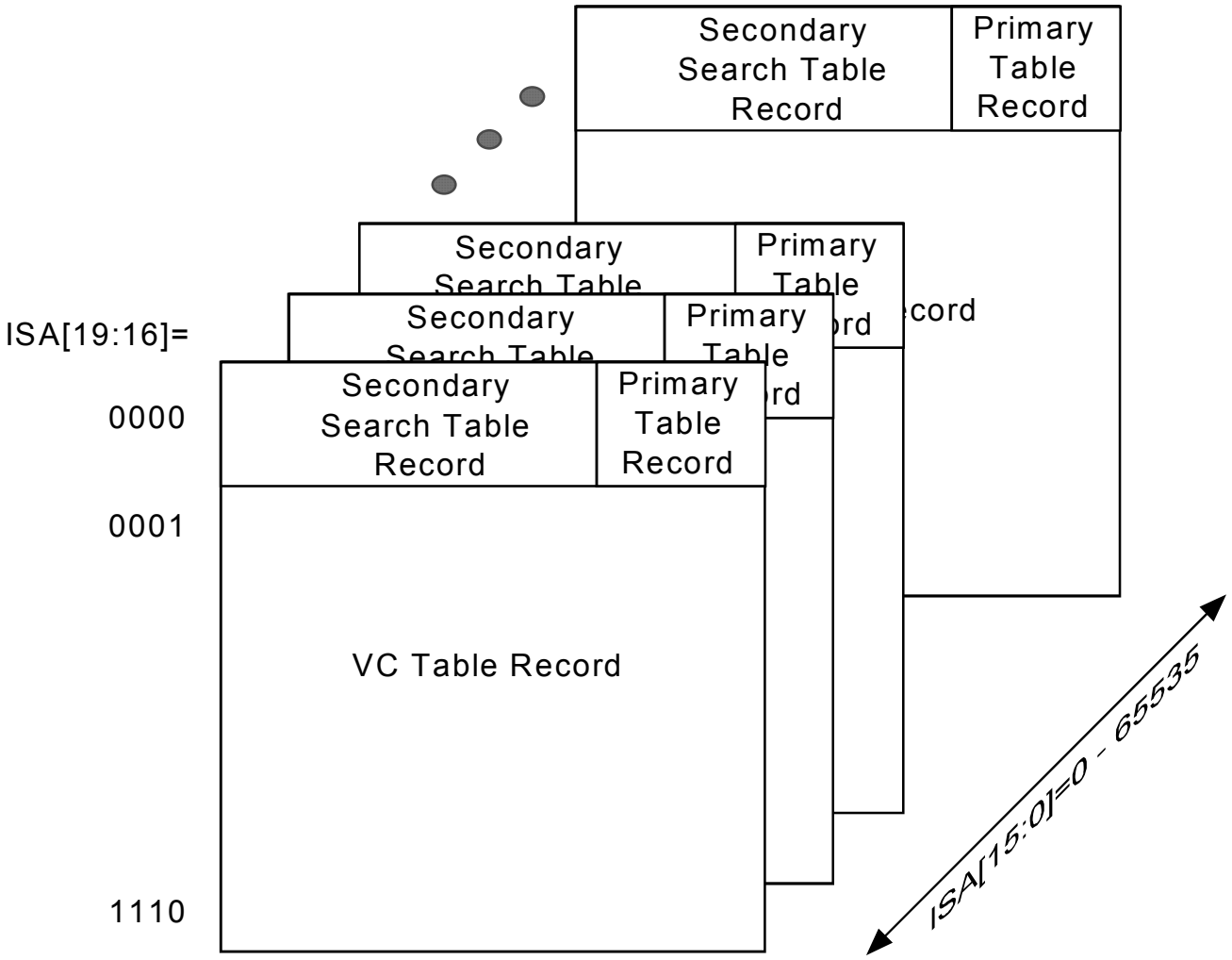
It is assumed that the reader has a basic understanding of the ATLAS. Please refer to document PMC-971154, "PM7324 S/UNI-ATLAS Datasheet" for a detailed description. In addition, basic knowledge of the ATM protocol would be helpful.

The routines provided are of a basic nature, written in ANSI C. Therefore, it is assumed that the reader has a reasonable understanding of the C programming language. To maintain operating system and processor independence, the routines rely on low level routines to access ATLAS registers. The intent of this document is not to implement a software driver, but to illustrate the operation of various ATLAS operations and how they may interact with software. As such, there is no pretense that the software presented here is exhaustive or optimal. It is expected that modifications shall be required to customize the example code for a specific system.

4 Programming the VC Search Tables

Search tables and context tables for the ATLAS are stored in external SRAM. There is one SRAM interface for the ingress, and one for the egress. Figure 1 below illustrates the organization of the ingress SRAM. The Primary Search Table is made up of the set of Primary Table Records. Similarly, the Secondary Search Table is made up of the set of Secondary Search Records, and the VC Table is made up of the set of VC Table Records. The egress SRAM is organized in a similar fashion, except that there is no search table since the egress side of the ATLAS uses a direct lookup for VC identification.

Figure 1 SRAM Organization



To identify the VC associated with a particular cell, the ATLAS performs a search in the VC Search Tables using the information in the PHY ID, cell header, prepend, and postpend. Once the VC associated with the cell has been identified, the cell may be processed according to the VC's properties, which are stored in the VC Table. Both the VC Search Tables and VC Tables are stored in external SRAM.

It is important to note that although the VC Tables and Search Tables occupy the same physical memory space, the Search Table Records are not necessarily associated with the VC Table Records at the same SRAM location.

4.1 SRAM Access

SRAM access is provided indirectly through the microprocessor interface. Table 1 below lists the registers that are relevant to SRAM accesses in the ATLAS. For a detailed description of these registers, refer to the S/UNI-ATLAS Datasheet.

Table 1 SRAM Access Registers

Address	Register
0x181	Ingress VC Table External RAM Address (LSB)
0x182	Ingress VC Table External RAM Access Control
0x183	Ingress VC Table External RAM Row Select
0x184	Ingress VC Table Write Mask
0x190 – 0x1CB	Ingress VC Table External RAM Data
0x2AB	Egress VC Table External RAM Address (LSB)
0x2AC	Egress VC Table Write Mask and Access Control
0x2AD	Egress VC Table External RAM Row Select
0x2AE – 0x2CD	Egress VC Table External RAM Data

An entire VC Table's data can be cached in the ATLAS, to reduce the amount of attention required by the microprocessor in performing accesses. To perform a write, execute the following steps:

1. Check that the BUSY bit in the Access Control Register is deasserted. Do not proceed if the BUSY bit is asserted.

2. Write data to the External RAM Data Registers for the fields required.

3. Write the appropriate information to the External RAM Row Select Register.

4. Write the SRAM address to be accessed to the External RAM Address Register.

5. Perform a write to the Access Control Register with the RWB bit set to 0. This will initiate the access and assert the BUSY bit. When the BUSY bit is deasserted, the SRAM access is complete.

To perform a read, execute the following steps:

1. Check that the BUSY bit in the Access Control Register is deasserted. Do not proceed if the BUSY bit is asserted.

2. Write the appropriate information to the External RAM Row Select Register.

3. Write the SRAM address to be accessed to the External RAM Address Register.

4. Perform a write to the Access Control Register with the RWB bit set to 1. This will initiate the access and assert the BUSY bit. When the BUSY bit is deasserted, the SRAM access is complete.

5. Read the data from the External RAM Data Registers for the fields required.

As an alternative to polling the BUSY bit, the BUSYB pin may be polled or used as an interrupt by a microprocessor. The BUSYB pin includes the status of both the egress and ingress BUSY bits.

4.2 ATLAS VC Identification

In the Ingress, the search consists of two stages: the Primary Search, and the Secondary Search. The first is a direct lookup using a key based upon the PHYID and up to 16 contiguous bits of the cell prepend, postpend, and header. The result of the Primary Search is the address of the root node of a binary tree that is used in the Secondary Search. The binary tree is traversed using a key based upon the cell's VPI/VCI and up to 11 contiguous bits of the cell prepend, postpend, and header. The termination of the binary search provides the address of the VC Table Record for the connection.

In the egress, the search consists of a direct lookup using a 16 bit key based upon the PHY ID, and 2 sets of up to 16 contiguous bits of the cell prepend, postpend, and header. The result of the lookup provides the address of the VC Table Record for the connection.

Table 2 below lists the registers associated with VC identification in the ATLAS. For a detailed description, refer to the S/UNI-ATLAS datasheet.

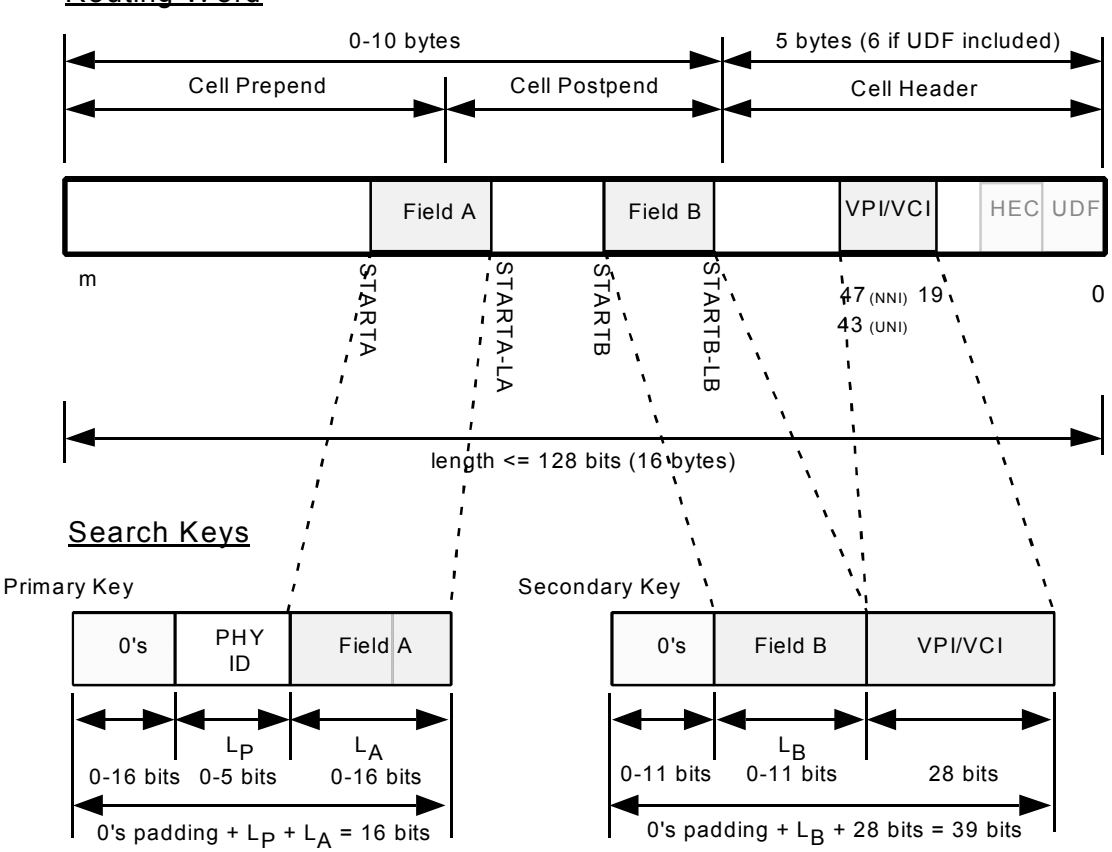
Table 2 VC Identification Registers

Address	Register
0x180	Ingress Search Engine Configuration
0x185	Field A Location and Length
0x186	Field B Location and Length
0x282	Egress Cell Processor Direct Lookup Configuration 1
0x283	Egress Cell Processor Direct Lookup Configuration 2

4.2.1 Ingress VC Search

When a cell enters the ATLAS, a routing word is constructed from the cell header, prepend and postpend. A Primary Key and Secondary Key are extracted from the routing word based upon the settings of STARTA[6:0], STARTB[6:0], LA[4:0], and LB[4:0] in registers 0x185 and 0x186. This is illustrated in Figure 2 below.

Figure 2 Ingress Search Key Composition



The ATLAS first uses the Primary Key. The Primary Key is the SRAM address of the Primary Search Table record for the particular connection. The contents of the Primary Search Table record is the address of Secondary Search Table entry which is the root node of the binary search tree for the particular connection. The Secondary Search Table Record contains the information for the next stage of the binary search. Figure 3 below illustrates the contents of the Secondary Search Table record.

Figure 3 Secondary Search Table Record

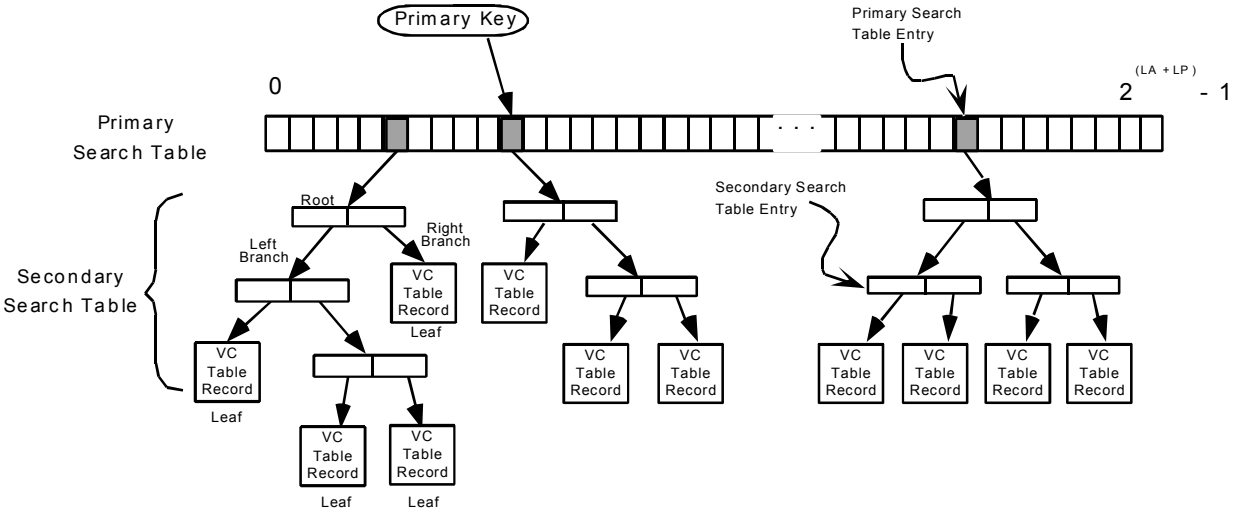
ISD[55:50]	ISD[49]	ISD[48:33]	ISD[32]	ISD[31:16]
Selector	Left Leaf	Left Branch	Right Leaf	Right Branch

Notes

1. Selector. The index of the Secondary Search Key bit upon which the branching decision is based. An index of zero represents the LSB. If the selected bit is a logic one, the "Left Leaf" and "Left Branch" fields are subsequently used. Likewise, if the selected bit is a logic zero, the "Right Leaf" and "Right Branch" are subsequently used. Typically, the Select value decreases monotonically with the depth of the tree, but other search sequences are supported by the flexibility of this bit. (i.e. typically, one starts from the most significant bit side and heads towards the least significant bit when selecting the bits to be used for branching decisions)
2. Left Leaf. This flag indicates if this node is a leaf. If "Left Leaf" is a logic one, the left branch is a leaf and the binary search terminates if the decision bit is a logic one. If "Left Leaf" is a logic zero, "Left Branch" value points to another node in the binary tree.
3. Left Branch. The 16-bit SRAM address pointing to the node accessed if the decision bit is a logic one. If "Left Leaf" is a logic one, "Left Branch" contains the SA[15:0] address identifying the VC Table Record for the incoming cell. If "Left Leaf" is a logic zero, "Left Branch" contains the SA[15:0] value pointing to another Secondary Search Table entry.
4. Right Leaf. This flag indicates if this node is a leaf. If "Right Leaf" is a logic one, the Right branch is a leaf and the binary search terminates if the decision bit is a logic zero. If "Right Leaf" is a logic zero, "Right Branch" value points to another node in the binary tree.
5. Right Branch. The pointer to the node accessed if the decision bit is a logic zero. If "Right Leaf" is a logic one, "Right Branch" contains the SA[15:0] address identifying the VC Table Record for the incoming cell. If "Right Leaf" is a logic zero, "Right Branch" contains the SA[15:0] value pointing to another Secondary Search Table entry.

At each step in the binary search, the ATLAS will look at the bit in the secondary key identified by the selector field. Based upon the value of the bit, the left branch or right branch in the tree will be taken. The left or right branch will be either a VC Table Record or another node in the tree, depending on the setting of the left or right leaf bits. If a node, the search continues. If a leaf, then the search terminates. The ATLAS will then compare the secondary key to the values for Field B, , VPI, and VCI found in the VC Table where the search terminated. If they match, the cell is processed. Otherwise, the cell is discarded. This process is illustrated in below.

Figure 4 VC Search Data Structure



4.2.2 Detailed Example

This example illustrates how the search could be constructed and what happens when a cell enters the ATLAS ingress. For this example, we will setup the ATLAS for eight connections. Table 2 shows the connection characteristics. We will assume that there is no prepend or postpend, so that the connection can be uniquely identified by the cell's header contents and PHY ID. For simplicity, the PHY ID and VPI for each of the eight connections is the same.

Table 2 VC Characteristics

VC #	VPI	VCI	PHY ID
1	5	00B5	0
2	5	00B0	0
3	5	008F	0
4	5	0023	0
5	5	0020	0
6	5	0017	0
7	5	0016	0
8	5	0010	0

To setup the ATLAS to support these connections, the ATLAS is configured to use the PHY ID and VPI for the Primary Search, and the VCI for the Secondary search. The register settings for this configuration are summarized below in Table 3.

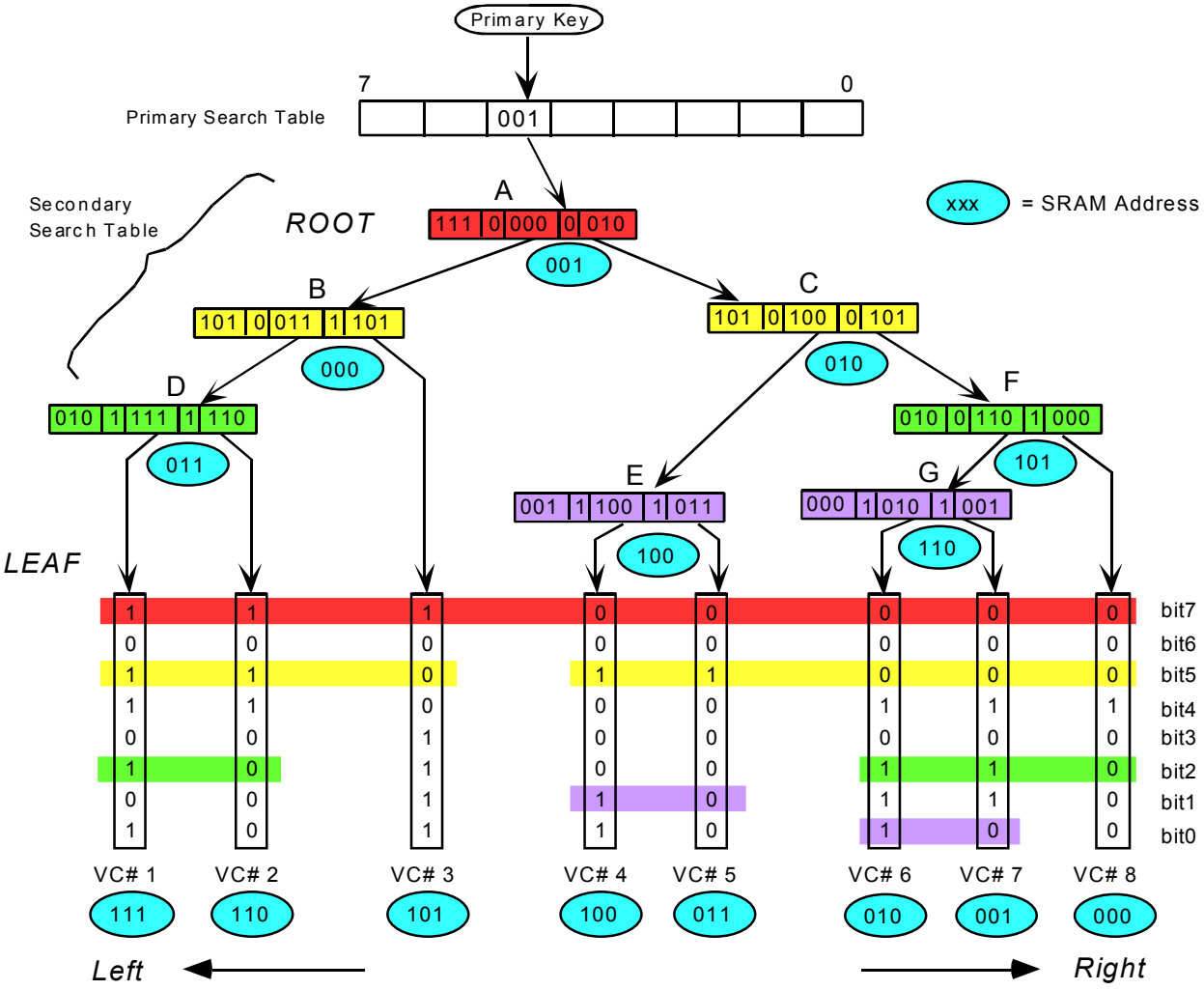
Table 3 ATLAS Search Configuration

Register	Setting	Effect	Description
0x185	0x0326	STARTA[6:0]= 0x26 LA[4:0]=0x03	Field A is the last 3 bits of the VPI
0x186	0x081B	STARTB[6:0]= 0x1B LB[3:0]=0x8	Field B is the last 8 bits of the VCI
0x180	0x1000	PHY[2:0]=0x0 BCIFHECUDF=1	Only using one PHY Cells from the Backward OAM Cell Interface have connection number encoded into HECUDF word.

Figure 5 below illustrates the search setup. The Secondary Search Table entries are shown at each node. For simplicity, *leading zeros are not included in each field*. These entries are arranged such that their vertical positions correspond to which bit is used to make the branching decision.

The 3-bit SRAM addresses are shown for the search table entries and the VC Table Records themselves. The SRAM addresses for the Secondary Search Table entries can be assigned arbitrarily. The SRAM addresses for the VC Table Records can also be assigned arbitrarily. In this case, they are assigned in descending order with the VC#'s.

Figure 5 Detailed Binary Search Tree



Suppose a cell comes in corresponding to VC# 2 in Table 2 above. The primary key will be 0x5. The address of the root node (node A) of the Secondary Search Tree will be in the 5th record of the Primary Table. The Secondary key will be 0xB0 (1011000). At the root (node A), the Secondary Search Table entry is read from the SRAM. The *Selector* field indicates that bit 111, or bit 7, should be examined. It is a one, which means the left branch should be taken. The *leaf* indicator is 0, meaning the leaf has not been found yet. Thus, the *left branch* address, 000, is used to read the Secondary Search Table entry of the next node, node B. Here, the *Selector* field is 101, meaning bit 5 should be used. Bit 5 is a one, meaning the left branch should be taken. The *leaf* indicator is 0, meaning the leaf has not been found yet. Thus, the *left branch* address, 011, is used to read the Secondary Search Table entry of the next node, node D. Here, the *Selector* field is 010, meaning bit 2 should be used. Bit 2 is a zero, meaning the right branch should be taken. The *leaf* indicator is 1, meaning the leaf has been found. The *right branch* address is used to read the 8-bit identifier of the leaf, which is used to compare to the incoming cell. They are the same,

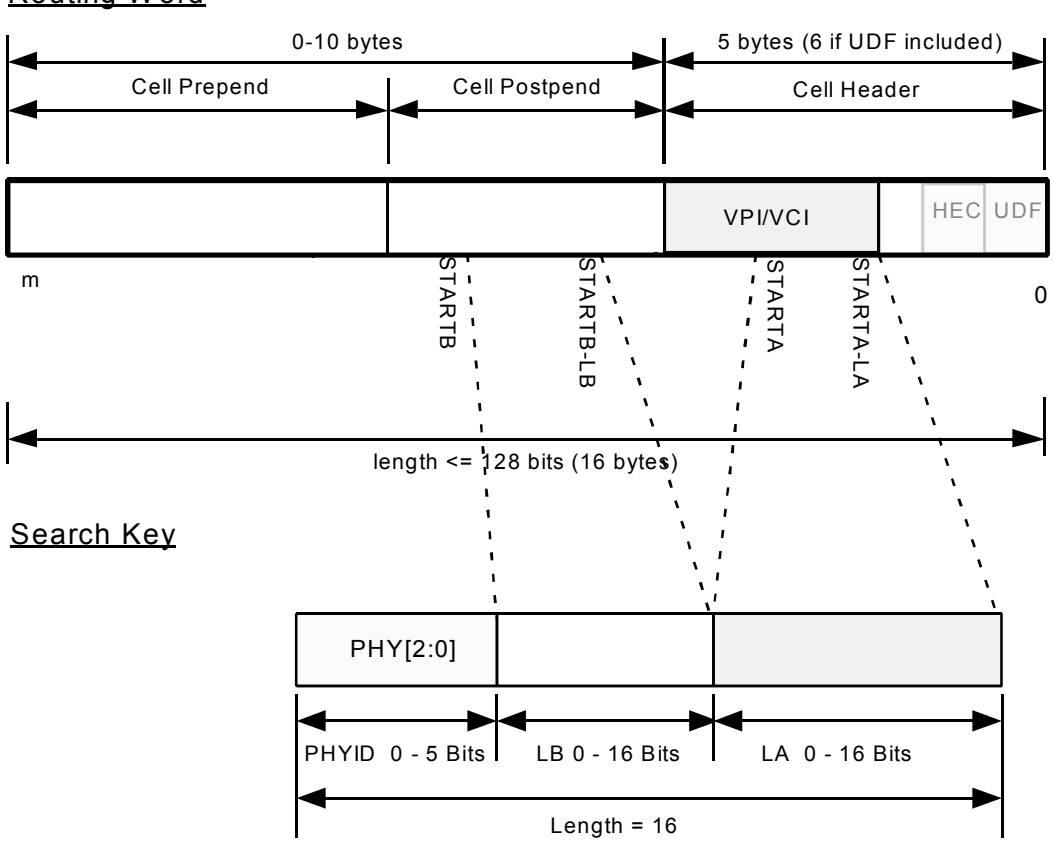
and thus the 8-bit address is found that points to the correct VC Table Record that corresponds to VC# 2.

In this example, the depth of the binary search is 4, which is equal to the number of bits in the secondary key as specified by the *Selector* field. On one extreme, the 8 VC's can be uniquely identified using only 3 bits, giving a minimum depth of 3 for the binary tree. On the other extreme, a maximum of 7 bits are used, giving a maximum depth of 7 for the binary tree.

4.2.3 Egress VC Search

When a cell enters the ATLAS at the egress, a routing word is constructed from the cell header, PHY ID, prepend, and postpend. A direct lookup key is extracted from the routing word based upon the settings of STARTA[6:0], LA[4:0], PHY[2:0], STARTB[6:0], and LB[4:0] in registers 0x282 and 0x283. This is illustrated in Figure 6 below. It is important to note that for a given bidirectional connection, the Ingress and Egress VC Tables must be related in one of three ways, as outlined in the ATLAS datasheet, Section 8.5.1. This is necessary for OAM cell generation to work correctly.

Figure 6 Egress Search Key Composition



4.2.4 Detailed Example

This example illustrates how the VC Record Address is determined for a cell entering the egress of the ATLAS. For OAM processing to work correctly, the Ingress and Egress VC Tables must be related. One of the ways that they can be related is by having the same VC Record address for both the Ingress and Egress Connection. In this case, generated OAM cells may have their HEC and UDF bytes overwritten with a 16 bit VC Record Address. Therefore, it will be assumed that cells entering the egress have their HEC and UDF bytes encoded with the VC Record Address (this is easily performed on the other side of the switch fabric, using header translation). Table 4 below illustrates the necessary configuration to extract the lookup key from the routing word.

Table 4 ATLAS Search Configuration

Register	Setting	Effect	Description
0x282	0x108F	PHY[2:0]=0x0 STARTA[6:0]= 0x0F BCIFHECUDF=1 LA[4:0]=0x10	Only one PHY selected Lookup key is equal to the HEC+UDF, being 16 bits starting with the 15th bit.
0x283	0x0000	STARTB[6:0]= 0x00 LB[3:0]=8=0x0	Field B is not used.

With this configuration, a cell entering the ATLAS will have the HEC and UDF extracted from the cell and used as the location of the VC Record in SRAM.

4.3 Operations

This section describes how to initialize and build up the Primary and Secondary Search Tables. VCs can be added or removed on the fly without corrupting a binary search in progress. It is assumed that there is a replica of the VC Table structure kept by the microprocessor, such that the microprocessor can determine how to add or remove VC's based on this replica. *Any modification to the actual VC Table Records (through the ATLAS) should be duplicated in the replica structure*.

4.3.1 Initialization

The following are the microprocessor actions required to initialize the Search Tables and VC Table Records:

1. Set the ISTANDBY and ESTANDBY bits of the Master Configuration register (0x01). These bits default to a logic 1 after a reset or upon power-up.

2. Write zeros (null pointer) to every Primary Search Table location (ISA[19:16] =0000).

3. Write zeros to the third word (ISA[19:16]=0010) of all Ingress VC Table Records. This clears the *Active* bit in the Configuration field.

4. Write zeros to the first word (ESA[19:16]=0000) of all Egress VC Table Records. This clears the *Active* bit in the Activation Field[2:0].

5. Clear the ISTANDBY and ESTANDBY bits of the Master Configuration register (0x01).

The remaining SRAM locations can be initialized as required for adding or removing connections.

4.3.2 Adding Connections

The following are the microprocessor actions required to provision a connection. Note that *the steps apply to ingress connections only*. For egress connections, one need only initialize the egress VC Table Record corresponding to the egress search key.

1. Determine the next available Ingress VC Table Record address. This address can simply be one higher than the highest existing VC Table Record address, or it can be the address of an VC Table Record that has been removed. Initialize the contents of the VC Table Record via ingress SRAM access registers in Table 1.

2. Perform a binary search (using the replica VC Table structure) to determine the insertion point. The last pointer accessed in the search shall be the one modified, be it a Primary Search Table entry, left branch or right branch.

3. Find a free Secondary Search Table entry and initialize it. The only exception to this is when a single VC Table Record exists in a tree, in which case the solitary Secondary Search Table entry is modified.

4. Perform a single SRAM write (via the Microprocessor RAM Address and Data registers) to incorporate the new Secondary Search Table entry in the existing tree structure. This step must be performed last to ensure a binary search in progress is not corrupted.

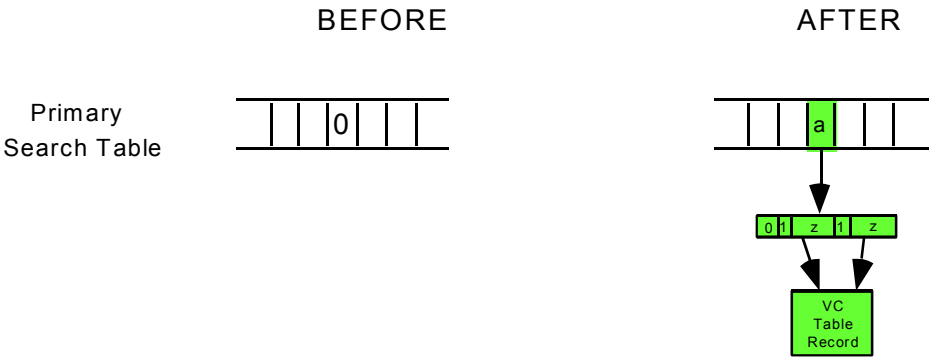
Five distinct types of insertions are possible based on the existing tree structure, and are detailed in the sections that follow. In the accompanying diagrams, the following key is used:

- a, b, c: Pointers to Secondary Search Table records
- w, x, y, z: Pointers to VC Table Records
- k, m, n: *Selector* field contents
- Shaded: Fields which have been modified in the process.

Case 1: Insertion Into an Empty Tree

The binary tree is empty. In this case, the null Primary Search Table pointer is modified to point to a newly created Secondary Search Table entry. Because no bits within the Secondary Search Key are required, both the left and right branches of the Secondary Search Table entry point to the same VC Table Record. The *selector* is a 'don't care'.

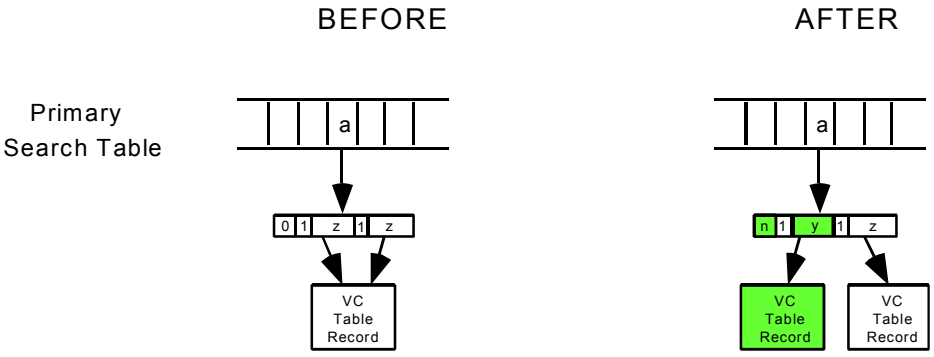
Figure 7 Case 1



Case 2: Insertion Into a Single Record Tree

The binary tree contains only a single VC Table Record. Modify the *selector* field to index the most significant bit of the Secondary Search Key which differs between the new and existing connection. Modify the left or right branch, as appropriate, to point to the newly created VC Table Record.

Figure 8 Case 2

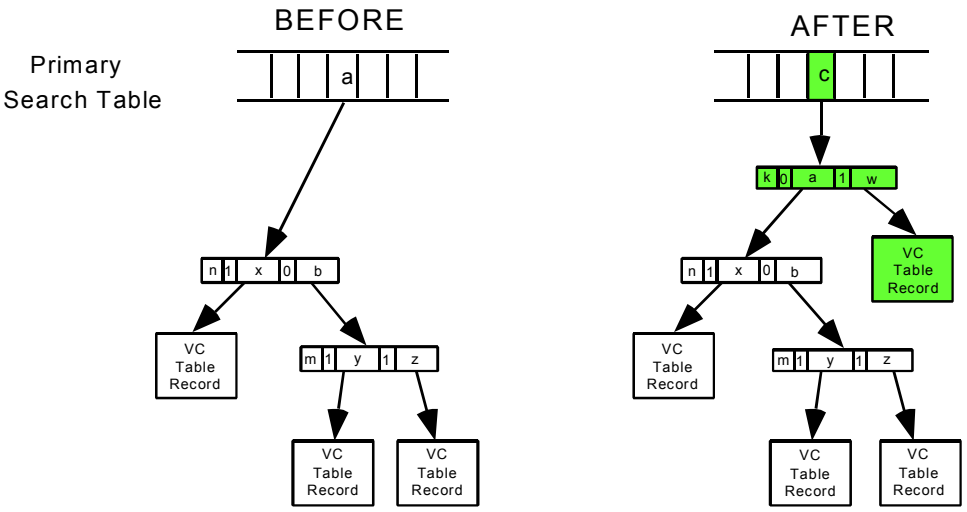


The diagram illustrates the case where the new VC has a one in the decision bit position and the existing VC has a zero in the same bit position. If the new VC had a zero in the decision bit position, the right branch would have been modified instead.

Case 3: Insertion at the Root of a Tree

The insertion point is at the root of the tree. This occurs when the new decision bit index is greater any of the indices currently in the search tree. In this case, the Primary Search Table entry is modified to point to the newly created Secondary Search Table entry. The New Secondary Search Table entry points to the new VC Table Record and the old tree root.

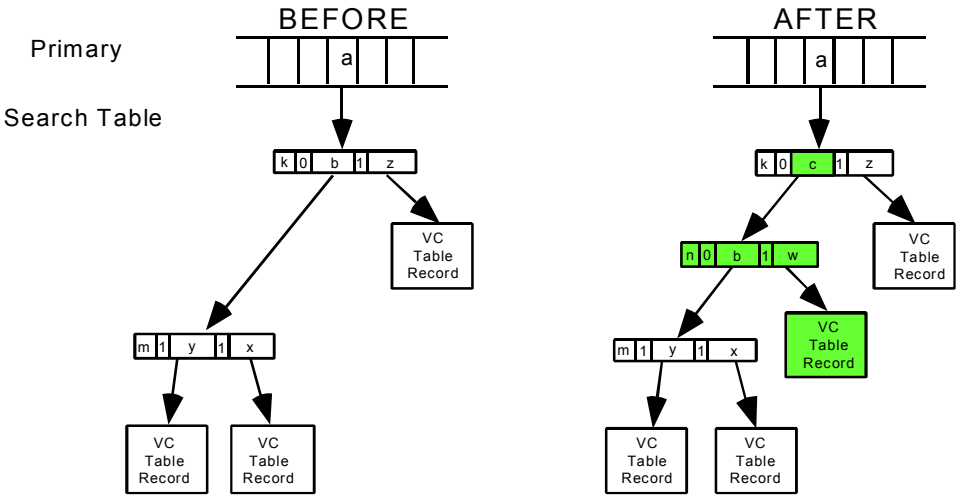
Figure 9 Case 3



Case 4: Insertion at Middle of a Tree

The insertion point is in the middle of the binary tree. The new Secondary Search Table entry points to the new VC Table Record and an existing node in the tree. The parent of the existing node is modified to point to the new Secondary Search Table entry in the final step of the insertion.

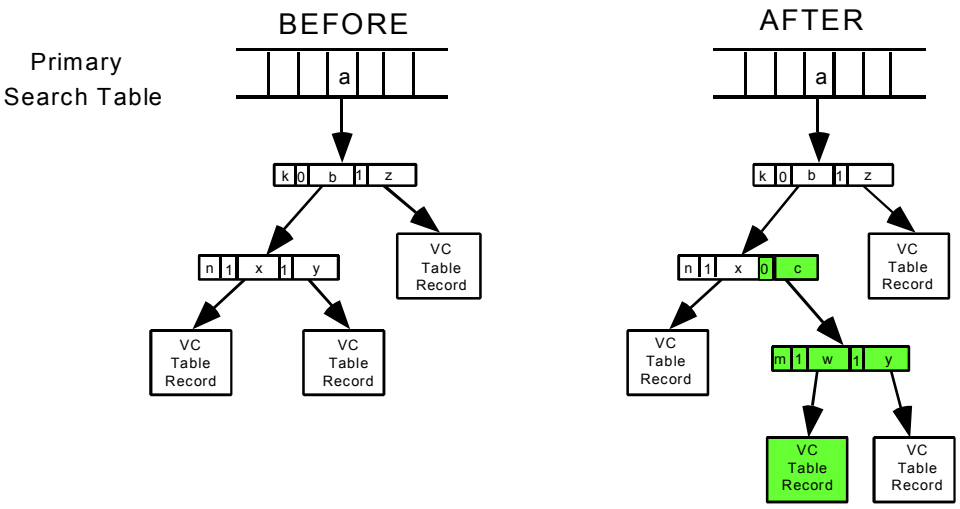
Figure 10 Case 4



Case 5: Insertion at a Leaf

The new Secondary Search Table entry is inserted at a leaf. The search for a candidate insertion point ends on a node which already points to a VC Table Record. The new Secondary Search Table entry points to the existing VC Table Record and the new VC Table Record. The existing Secondary Search Table entry is modified to point to the new Secondary Search Table entry in the final step of the insertion.

Figure 11 Case 5



4.3.3 Removing Connections

The following are the microprocessor actions required to remove a connection:

1. Find the location of the Secondary Search Table entry pointing to the connection's VC Table Record.

2. Perform and SRAM write to modify the parent node (be it the Primary Table entry or another Secondary Search Table entry) of the Secondary Search Table entry being removed to point to the node remaining after the connection removal. The only exception to this is when only two VC Table Records exist in a tree, in which case the solitary Secondary Search Table entry is modified. The VC is now considered unprovisioned and any cells belonging to the VC will be discarded.

3. Tag in software the removed Secondary Search Table entry as free.

4. Read the final statistics for the connection from the VC Table Record and tag in software the VC Table Record address as free. Also, clear the "Active" bit in the VC Table Record.

The connection removal process examples are not illustrated because the results are exactly the reverse of the connection provisioning process. Note that the above steps are for ingress connections only. For egress connections, one need only turn off the active bit in the egress VC Table Record and setup the appropriate parameters.

4.4 Example Routines

The services provided are detailed in the sections that follow. Note that in all cases, other embodiments of the routines are possible, and the method presented here represents only one possible implementation.

4.4.1 Accessing the ATLAS

All operations on the ATLAS are performed through the microprocessor interface. All example software in this document relies on low level service routines for the 16-bit read and write access in the ATLAS. The sections that follow detail the routines that need to be defined.

Type Definitions

```
char BYTE; /* signed 8 bits, range [-128,127] */
unsigned char UBYTE; /* unsigned 8 bits, range [0,255] */
short WORD; /* signed 16 bits, range [-32768,32767] */
unsigned short UWORD; /* unsigned 16 bits, range [0,65535] */
long DWORD; /* signed 32 bits, range [-2147483648,2147483647] */
unsigned long UDWORD; /* unsigned 32 bits, range [0,4294967295] */
UWORD REG_TYPE; /* ATLAS registers are 16 bit */
```

Global Variables

```
UDWORD DEVICE_BASE_ADDR[MAX_DEVICE_NUMBER+1]  /* array of base
addresses */
UDWORD MAX_REG_ADDR[MAX_DEVICE_NUMBER+1]  /* array of devices' max
register */
UBYTE REG_BYTE_SIZE[MAX_DEVICE_NUMBER+1]  /* array of devices'
register size*/
```

ATLAS Read

This routine performs a read of the specified register of the specified ATLAS device. The specific implementation of this routine will rely entirely on the microprocessor and supporting hardware.

Pseudocode:

```
BYTE devRead(
      UBYTE device,
```

```
      UDWORD regAddr,
      REG_TYPE *readData)
{
address = DEVICE_BASE_ADDR[device] + regAddr*REG_SIZE
if REG_BYTE_SIZE[device] = 1 then
      *readData = (UBYTE) contents of address
else if REG_BYTE_SIZE[device] = 2 then
      *readData = (UWORD) contents of address
if data is valid (check via exception handling) then
      return SUCCESS
else return ERROR
}
```

Parameters:

- device Device to access
- regAddr Register address
- readData Stores the value read

ATLAS Write

This routine performs a write to the specified register of the specified ATLAS device with the specified data. The specific implementation of this routine will rely entirely on the microprocessor and supporting hardware.

Pseudocode:

```
BYTE devWrite(
      UBYTE device,
      UDWORD regAddr,
      REG_TYPE wrData)
{
address = DEVICE_BASE_ADDR[device] + regAddr*REG_SIZE
if REG_SIZE[device] is 1 then
      contents of address = (UBYTE) wrData
else if REG_SIZE[device] is 2 then
      contents of address = (UWORD) wrData
else return ERROR
}
```

Parameters:

- `device` Device to access
- `regAddr` Register address
- `wrData` Value to write into the register

ATLAS Write with Mask

This function modifies selected bits of a device register. The register is read, the desired bits are modified, then the register is written with the new data. Note that since this routine performs a read, bits that clear upon read, such as interrupt indication bits, will be lost.

Pseudocode:

```
BYTE devWriteMask(
      UBYTE device,
      UDWORD regAddr,
      REG_TYPE wrData,
      REG_TYPE wrMask)
{
devRead(device, regAddr, &data)
data = data & (~wrMask)
data = data | (wrData & wrMask)
devWrite(device, regAddr, data)
}
```

Parameters:

- `device` device number to access
- `regAddr` register address
- `wrData` value to write into the register(s)
- `wrMask` mask for bits to write into register, bits set = bits to modify

Poll ATLAS Bit High

This function polls a particular bit in a register until it is a logic 1 or a timeout occurs.

Pseudocode:

```
BYTE devBitHiPoll(
      UBYTE device,
      UDWORD regAddr,
      UBYTE bitPosition,
      UWORD numCheck)
```

```
{
Loop numCheck times
      devRead(device, regAddr, &data)
   If read error then return error code from devRead()
      If (data >> bitPosition) & 0x01 = 1 then return SUCCESS
   return ERROR_TIMED_OUT
end loop
}
```

Parameters:

- `device` device number to access
- `regAddr` register address
- `bitPosition` bit to poll (0 = least significant bit)
- `numCheck` number of times to check the bit before giving up

Poll ATLAS Bit Low

This function polls a particular bit in a register until it is a logic 0 or a timeout occurs.

Pseudocode:

```
BYTE devBitLowPoll(
      UBYTE device,
      UDWORD regAddr,
      UBYTE bitPosition,
      UWORD numCheck)
{
Loop numCheck times
   devRead(device, regAddr, &data)
   If read error then return error code from devRead()
   If (data >> bitPosition) & 0x01 = 0 then return SUCCESS
End loop
return ERROR_TIMED_OUT
}
```

Parameters:

- `device` device number to access
- `regAddr` register address

- `bitPosition` bit to poll (0 = least significant bit))
- `numCheck` number of times to check the bit before giving up

4.4.2 SRAM Access Routines

SRAM access routines are used by the SRAM diagnostic function and the table maintenance routines.

Type Definitions

```
typedef struct {
    UWORD addr; /* Ingress VCRA (16 bits) */
    UWORD row; /* row(s) of the SRAM to access, bit n=row n */
    UBYTE data[IVCRAM_NUM_ROWS][IVCRAM_NUM_BYTES]; /* data as 2D
array */
    UBYTE wrMask;  /* bytemask for single row access only */
    } IVCRAM_TYPE;


typedef struct {
    UWORD addr; /*  Egress VCRA (16 bits) */
    UWORD row;  /*  rows of the egress SRAM to access (bit n = row n)
*/
    UBYTE data[EVCRAM_NUM_ROWS][EVCRAM_NUM_BYTES]; /* data as 2D
array */
    UBYTE wrMask; /* bytemask for single row access only */
    } EVCRAM_TYPE;
```

Ingress SRAM Read

This function reads a record from the Ingress VC SRAM.

Pseudocode:

```
BYTE ivramRead(
      UBYTE device,
      IVC_TYPE *ivcData)
{
(*ivcData).row = ((*ivcData).row & IVCRAM_ROWMASK) to mask out unused
rows
devBitLowPoll() /* to ensure BUSY is low */
devWrite(device, REG_IVC_ADDR, (*ivcData).addr) /* to setup address
*/
devWrite(device, REG_IVC_ROW, (*ivcData).row) /* to setup row mask */
```

```
devWriteMask(device, REG_IVC_CONTROL, 0x8000, 0x8000) /* to initiate
read*/
devBitLowPoll() /* to check if access is complete */
for each row being accessed (0 - 14)
   for each word in the row
         devRead(appropriate register, based on offset using row,
word)
         *ivcData.data[row][byte]=result of devRead
         *ivcData.data[row][byte+1]=result of devRead>>8
}
```

Parameters:

- `device` Device to access
- `*ivcData` Pointer to a record of IVC_TYPE

Ingress SRAM Write

This function writes a record to the Ingress VC SRAM.

Pseudocode:

```
BYTE ivcramWrite(
      UBYTE device,
      IVC_TYPE ivcData)
{
ivcData.row = (ivcData.row & IVCRAM_ROWMASK); /* mask out unused rows
*/
devBitLowPoll() /* to ensure BUSY is low-*/
devWrite(device, REG_IVC_ADDR, ivcData.addr) /* to setup address*/
devWrite(device, REG_IVC_ROW, ivcData.row) /* to setup row*/
for each row not masked out
      for each word in the row
      regData = ivcData.data[row][byte+1]
      regData = (ATLAS register << 8) | ivcData.data[row][byte]
      regAddr = appropriate offset from row, word
      devWrite(device, regAddr, regData)

devWrite(device, REG_IVC_WRMASK, ivcData.wrMask) /* setup bytemask if
used*/
devWriteMask(device, REG_IVC_CONTROL, 0x0000, 0x8000) /*initiate
write*/
```

```
return devBitLowPoll() /* to ensure access is complete */
}
```

Parameters:

- `Device` Device to access
- `IvcData` Record of IVC_TYPE

Egress SRAM Read

This function reads a record from the Egress VC SRAM.

Pseudocode:

```
BYTE evcramRead(
      UBYTE device,
      EVC_TYPE *evcData)
{
(*evcData).row = ((*evcData).row & EVCRAM_ROWMASK) /* maskout unused rows*/
devBitLowPoll() to ensure BUSY is low
devWrite(device, REG_EVC_ADDR, (*evcData).addr) /*setup address */
devWrite(device, REG_EVC_ROW, (*evcData).row)/* setup row*/
devWriteMask(device, REG_EVC_CONTROL, 0x8000, 0x8000) /* initiate read*/
devBitLowPoll() /* check access is complete*/
for each row accessed
   for each word
         devRead(appropriate register)
         *evcData.data[row][byte]=regData
         *evcData.data[row][byte+1]=regData>>8
return SUCCESS;
}
```

Parameters:

- `device` Device to access
- `*evcData` Pointer to a record of EVC_TYPE

Egress SRAM Write

This function writes a record to the Egress VC SRAM.

Pseudocode

```
BYTE evcramWrite(
      UBYTE device,
      EVC_TYPE evcData)
{
evcData.row = (evcData.row & EVCRAM_ROWMASK);
devBitLowPoll()/*ensure BUSY is low*/
devWrite(device, REG_EVC_ADDR, evcData.addr) /* setup address*/
devWrite(device, REG_EVC_ROW, evcData.row); /* setup row*/
for each row accessed
   for each word
      regData = ecData.data[row][byte+1]
      regData = (ATLAS register << 8) | evcData.data[row][byte]
      regAddr = appropriate offset from row, word
      devWrite(device, regAddr, regData)

regData = (0x0000 | evcData.wrMask);
devWriteMask(device, REG_EVC_CONTROL, regData, 0x8008) /*initiate the
write*/
return devBitLowPoll() /* check access is complete*/
}
```

Parameters:

- `device` Device to access
- `evcData` A record of EVC_TYPE

4.4.3 SRAM Diagnostic Routines

SRAM diagnostics should be performed at initialization to ensure data integrity at the interface between the ATLAS and the SRAM.

Global Variables

```
UBYTE pattern[MAXPATTERN]={0x00,0xFF,0x55,0xAA,0xCD,0x32} /* test
patterns */
```

Ingress SRAM Diagnostics

This function tests the external Ingress SRAM and the interface to it. The test algorithm is based on "Fault Modelling and Test Algorithm Development for Static Random Access Memories" in the references. The test is performed as follows:

1. Write "pattern" to all locations with an incrementing address.

2. Read "pattern" and write inverse "pattern" to all locations with an incrementing address.

3. Read inverse "pattern" and write "pattern" to all locations with an incrementing address.

4. Read "pattern" and write inverse "pattern" to all locations with a decrementing address.

5. Read inverse "pattern" and write "pattern" to all locations with a decrementing address.

The byte "patterns" used are 0x00, 0x55 and 0xCD, repeated over the eight byte width of the SRAM bus. The last pattern verifies the parity bits. The global constant, `IVC_NUM_RECORDS`, determines the depth of RAM tested. Depending on the depth of the SRAM, this test may take several seconds. Missing SRAM fields may be masked out by use of the global constant `IVC_ROWMASK`. Note that the test overwrites the contents of SRAM.

Pseudocode:

```
BYTE ivcramdiag(
   UBYTE device,
   UDWORD *faultLoc)
{
IVCRAM_TYPE ivcDiag
devWriteMask(device, REG_MASTER_CONFIG, 0x0001, 0x0001) /*set
STANDBY*/
for each pattern
      for each vcra up to the maximum populated
         ivcData.addr=vcra
         for each row populated
         ivcData.row=row
               for each byte
                     ivcDiag.data[row][byte]=pattern
         ivcramWrite(device,ivcDiag) to write to SRAM
         for each row populated
               for each byte
                     if ivcDiag.data[row][byte]!=pattern then return
fail
devWrite(device, REG_MASTER_CONFIG, 0x0000, 0x0001) /* clear STANDBY
*/
return SUCCESS
}
```

Parameters:

- `Device` Pointer to the base address of the ATLAS device being accessed.

- *FaultLoc A pointer to an integer which stores the SRAM address value at which the failure was detected.

Egress SRAM Diagnostics

The egress SRAM diagnostic is conducted in the same way as the ingress test, with the exception that the bus width is four bytes rather than eight.

Pseudocode:

```
BYTE evcramdiag(
   UBYTE device,
   UDWORD *faultLoc)
{
EVCRAM_TYPE evcDiag
devWriteMask(device, REG_MASTER_CONFIG, 0x0002, 0x0002) /*set
STANDBY*/
for each pattern
      for each vcra up to the maximum populated
         evcData.addr=vcra
         for each row populated
         evcData.row=row
               for each byte
                     evcDiag.data[row][byte]=pattern
         ivcramWrite(device,evcDiag) to write to SRAM
         for each row populated
               for each byte
                     if evcDiag.data[row][byte]!=pattern then return
fail
devWrite(device, REG_MASTER_CONFIG, 0x0000, 0x0002) /* clear STANDBY
*/
return SUCCESS
}
```

Parameters:

- Device Pointer to the base address of the ATLAS device being accessed.
- *faultLoc A pointer to an integer which stores the SRAM address value at which the failure was detected.

4.4.4 Table Maintenance Routines

This section details routines for adding and removing ingress and egress connections within the ATLAS. A shadow copy of the configured connections is maintained in the microprocessor memory.

Type Definitions

```
typedef struct {    /* type definition for ingress VC records */
    UWORD fieldA;   /* this isn't part of the VC rec, but is req'd
for search
    UBYTE f4toF5AIS;    /* fieldA + phyID = primary search Key */
    UBYTE phyID;
    UBYTE pmActive2;
    UBYTE pmAddr2;
    UBYTE pmActive1;
    UBYTE pmAddr1;
    UBYTE nni;
    UWORD fieldB;
    UWORD vpi;
    UWORD vci;
   .
   .
   .
UBYTE receivedSegmentAisDefect[16];
    } IVC_RECORD_TYPE;

typedef struct {     /* type definition for secondary tree nodes */
    UBYTE selector;
    UBYTE leftLeaf;
    UWORD leftBranch;
    UBYTE rightLeaf;
    UWORD rightBranch;
    } NODE_TYPE;

typedef struct {  /* type definition for egress VC records */
    UBYTE inPhyID;
    UWORD fieldA;   /* inPhyID + fieldA + fieldB = address */
    UWORD fieldB;
    UBYTE active;
```

```
    .
    .
    .
     UBYTE receivedSegmentAisDefectLocation[16];
     } EVC_RECORD_TYPE;


typedef struct {  /* type definition for search results */
    UWORD primary; /* - primary key used for the search */
    UBYTE secondary[SECONDARY_KEY_SIZE]; /* - secondary key
    UBYTE matched;  /* Success of search: 1 = success; 0 = not found
*/
    UBYTE root; /* search terminated on root node */
    UBYTE emptyTree; /* VC table is empty: 1 = empty; 0 otherwise */
    UBYTE singleBranch; /* the secondary tree has only 1 connection
*/
    UWORD leafAddr; /* Address of secondary key found for comparison
*/
    UBYTE leafValue[SECONDARY_KEY_SIZE]; /* secondary key found
    UBYTE finalDirection; /* last search direction: 1 = left; 0 =
right */
    UWORD finalNodeAddr; /* Address of final node searched */
    NODE_TYPE finalNode; /* the node where the search terminated */
    UBYTE previousDirection; /* the previous branch direction */
    UWORD previousAddr; /* - Address of previous node searched */
    UBYTE newSelector; /* - selector bit of the secondary key of the
new VC */
    } IVC_SEARCH_RESULTS_TYPE;
```

Global Variables

```
UWORD gmIvcPrimaryTable[] /* local copy of the primary search table
table */
NODE_TYPE gmIvcSecondaryTable[]/*  local copy of the secondary search
table */
UBYTE gmIvcSecondaryKeys[][] /* local copy of the secondary search
keys */
UWORD gmIvcSecondaryAddrList[] /* list of free secondary search nodes
*/
UWORD gmIvcSecondaryAddrIndex = 0; /* points to next free node */
UBYTE gmIvcRecordAddrFree[] /* list of free VC addresses */
UWORD gmIvcRecordAddrFreeIndex = 0; /* points to gmIvcRecordAddrFree[
]  */
```

```
UWORD gmIvcRecordNumFree = IVC_NUM_RECORDS;
UBYTE gIvcPrimaryTableUsed; /* boolean, 1 if primary table is in use
*/
UBYTE gIvcFieldALength; /* primary key length */
UWORD gIvcFieldAMask; /* mask off unwanted field A bits  */
UWORD gIvcPhyIDMask; /* mask off unwanted PhyID bits  */
UWORD gIvcFieldBMask; /* mask off unwanted field B bits */
UBYTE gIvcFieldBLength; /* length of field B */

UWORD gEvcFieldALength; /*  set up the egress direct lookup */
UWORD gEvcFieldAMask; /*  mask off field A bits */
UWORD gEvcFieldBLength; /*  set up the primary key  */
UWORD gEvcFieldBMask; /*  mask off unwanted field B bits, */
UWORD gEvcPhyIDMask; /*  mask off unwanted inPhyID bits */
UBYTE gmEvcRecordAddrExists[] /*   table to indicate which used
egress VCs */
```

Add Ingress VC

The functions presented in this section set up an ingress connection. This is accomplished through the steps below.

1. Perform a search using the primary and secondary keys of the VC being added. This will determine the structure of the associated search tree. If the search terminates on a node (as opposed to an empty tree), then choose a selector value that corresponds to the most significant bit that differs between the new secondary key and the old secondary key.
   - If the binary tree is empty, add a root node with both left and right branches pointing to the single VC Table Record (Case 1). This completes the addition of the VC.
   - If the binary tree has only one connection, modify the root node so that the appropriate branch points to the new VC Table Record (Case 2). This completes the addition of the VC.
2. Determine the correct insertion point for the new node that will point to the new VC Table Record (Case 3, 4 or 5). The correct insertion point will be based on the selector value at each of the existing nodes and the selector value chosen for the new node.
   - If the correct insertion point is at the end of a tree (Case 5), add a node with the appropriate branch pointing to the new VC Table record, and the other branch pointing to the old VC Table Record. Modify the node above to point to the new node rather than the old VC Table Record.
   - If the correct insertion point is at the root (Case 3), add a node with the appropriate branch pointing to the old root node, and the other branch pointing to the new VC Table Record.

- If the correct insertion point is between two nodes (Case 4), add a node with the appropriate branch pointing to the node below, and the other branch pointing to the new VC Table Record. Modify the node above to point to new node rather than the node below.

4.4.5 Initialization

Prior to use, the VC Tables and their internal copies must be initialized.

Pseudocode:

```
BYTE ivcInitialize(UBYTE device)
{
for each vcra
   gmIvcPrimaryTable[]=0 /*set the table to the null pointer*/
   ivcData.data[0][0]=0
   ivcData.addr=vcra
   ivcData.row=0
   ivcData.wrMask=0xFFFC
   ivcramWrite(ivcData) /*clear primary table in SRAM*/

gmIvcRecordNumFree = IVC_NUM_RECORDS;
gmIvcRecordAddrFreeIndex = 0;

ivcData.data[2][7] = 0; /* status field */
ivcData.row = 0x0040; /* row 0010 */
ivcData.wrMask = 0;
for each vcra
   ivcData.addr=vcra
   ivcramWrite(ivcData) /*clear active bit in each vc record*/

/* Initialize list of free secondary node addresses */
for (addr = 1; (addr <= IVC_MAX_SECONDARY_ADDR); addr++)
    gmIvcSecondaryAddrList[addr-1] = addr;
return SUCCESS;
}
```

Parameters:

None.

4.4.6 Add Connection

Pseudocode:

```
BYTE ivcAddVC(
      UBYTE device,
      IVC_RECORD_TYPE ivcRecord
      UBYTE addrForce
      UWORD *ivcAddr)
{
IVC_SEARCH_RESULTS_TYPE searchResults; /* results from ivcSearch() */
IVC_SEARCH_RESULTS_TYPE insertResults; /* results from
ivcFindInsertionPoint()
UWORD primaryKey; /* primary Key of the connection being added */
UBYTE secondaryKey[SECONDARY_KEY_SIZE]; /* secondary key of the vc
being added


primaryKey = (ivcRecord.phyID +ivcRecord.fieldA)
secondaryKey = (ivcRecord.fieldB+ivcRecord.vpi+ivcRecord.vci)
ivcSearch(primaryKey, secondaryKey, &searchResults);
if (searchResults.matched) return ERROR_IVC_ADD_VC_EXISTS;
if (searchResults.emptyTree)  /* case 1 */
   ivcAddToEmptyTree()
if (searchResults.singleBranch)  /* case 2 */
   ivcAddToSingleBranch()
if (searchResults.newSelector < searchResults.finalNode.selector) /*
case 5 */
      ivcAddToEndNode()
ivcFindInsertionPoint()
if (insertResults.root) /* case 3 */
    return ivcAddToRoot()
else /* case 4 */
      ivcAddToMiddleBranch()
return ERROR_IVC_ADD_FAIL; /* no other options */
}
```

Parameters:

- `device` Device to access
- `ivcRecord` The VC Table Record to be added

- addrForce If TRUE, then add the VC to the address specified in ivcAddr. If FALSE, then an address will be selected automatically and returned in ivcAddr
- ivcAddr SRAM address of the VC record

4.4.7 VC Search

This routine performs a binary search to help determine how the tree structure must be changed in order to add or remove a connection on the tree.

Pseudocode:

```
BYTE  ivcSearch(
   UWORD primaryKey,
      UBYTE secondaryKey[SECONDARY_KEY_SIZE],
      IVC_SEARCH_RESULTS_TYPE *searchResults)
{
ivcInitSearchResults(searchResults) /*initialize result values*/
currentNodeAddr = gmIvcPrimaryTable[primaryKey];
if currentNodeAddr=0 { /* case 1 - the search tree is empty */
    (*searchResults).emptyTree = TRUE;
    (*searchResults).secondary = secondaryKey;
    (*searchResults).primary = primaryKey;
    return SUCCESS;
    }

previousNodeAddr = currentNodeAddr;
foundLeaf = FALSE;
while not foundLeaf {
    if bit#currentNode.selector of secondaryKey=1 (go left)
         if currentNode.leftLeaf=1 foundLeaf=TRUE
         nextNodeAddr =
gmIvcSecondaryTable[currentNodeAddr].leftBranch;
    else (go right)
      if currentNode.rightLeaf=1 foundLeaf=TRUE
      nextNodeAddr =
gmIvcSecondaryTable[currentNodeAddr].rightBranch;
    if not foundLeaf (continue search)
        (*searchResults).previousDirection = currentDirection;
        previousNodeAddr = currentNodeAddr;
        currentNodeAddr = nextNodeAddr;
```

```
} /* end while loop */


(*searchResults).previousAddr = previousNodeAddr;
(*searchResults).finalDirection = currentDirection;
(*searchResults).finalNodeAddr = currentNodeAddr;
(*searchResults).finalNode = gmIvcSecondaryTable[currentNodeAddr];
(*searchResults).leafAddr = nextNodeAddr;
(*searchResults).leafValue = gmIvcSecondaryKeys[nextNodeAddr]
(*searchResults).secondary = secondaryKey;
(*searchResults).primary = primaryKey;
if (previousNodeAddr == currentNodeAddr) (*searchResults).root =
TRUE;
if (((*searchResults).finalNode.leftBranch ==
(*searchResults).finalNode.rightBranch) &&
(*searchResults).finalNode.leftLeaf &&
(*searchResults).finalNode.rightLeaf &&
(*searchResults).root) (*searchResults).singleBranch = TRUE;


/*if the secondary key of the new VC matches the secondaryKey
(leafValue)*/ /*the VC that the search terminated on, then the VC
already exists.*/
(*searchResults.newSelector)=the MSB that differs between the 2 keys
/* the new selector is the most significant bit that differs between
them */
return SUCCESS;
}
```

Parameters:

- `PrimaryKey` Key to search in the primary table
- `SecondaryKey` Key to search in the secondary table
- `SearchResults` Pointer to a record of IVC_SEARCH_RESULTS_TYPE that will hold results of this search

4.4.8 Case 1: Insertion Into an Empty Tree

Pseudocode:

```
BYTE ivcAddToEmptyTree(
      UBYTE device,
      IVC_SEARCH_RESULTS_TYPE searchResults,
      IVC_RECORD_TYPE ivcRecord,
```

```
      UBYTE addrForce,
      UWORD ivcAddr)
{
if addrForce=1 ivcMarkRecordAddr(recordAddr); /*go mark the record as
used*/
else ivcGetFreeRecordAddr(&recordAddr); /*allocate a record*/
*ivcAddr = recordAddr;
ivcGetFreeSecondaryAddr(&rootNodeAddr); /*allocate a new secondary
node*/

rootNode.selector = 0;  /*set l&r to point to leaves at recorAddr*/
rootNode.leftLeaf = 1;
rootNode.leftBranch = recordAddr;
rootNode.rightLeaf = 1;
rootNode.rightBranch = recordAddr;

ivcUpdateRecord() /*map ivcRecord into SRAM*/

ivcUpdateSecondaryTable() /*modify both the local and external sec
tables*/
ivcUpdatePrimaryTable() /*point to the new node*/
}
```

Parameters:

- `Device` Device to be accessed
- `SearchResults` Pointer to a record of IVC_SEARCH_RESULTS_TYPE that will hold results of this search
- `ivcRecord` The VC Table Record to be added
- `addrForce` If `TRUE`, then add the VC to the address specified in `ivcAddr`. If `FALSE`, then an address will be selected automatically and returned in `ivcAddr`
- `ivcAddr` SRAM address of the VC record

4.4.9 Case 2: Insertion Into a Single Record Tree

Pseudocode:

```
BYTE ivcAddToSingleBranch(
      UBYTE device,
      IVC_SEARCH_RESULTS_TYPE searchResults,
      IVC_RECORD_TYPE ivcRecord,
```

```
      UBYTE addrForce,
      UWORD *ivcAddr
{
if addrForce=1 then mark the vc record as used
else ivcGetFreeRecordAddr(&recordAddr) /* allocate a new record*/

newRootNode = searchResults.finalNode; (the new=(old +
modifications))
newRootNode.selector = searchResults.newSelector;
if bit#newSelector of secondaryKey=1 /* new record goes on left */
    newRootNode.leftBranch = recordAddr;
else /* new record goes on right */
    newRootNode.rightBranch = recordAddr;

ivcUpdateRecord()/* setup the new vc record*/
ivcUpdateSecondaryTable() /*modify the secondary table (local and
ext)*/
}
```

Parameters:

- `Device` Device to access
- `SearchResults` Contains results from a VC binary tree search
- `ivcRecord` The VC Table Record to be added
- `addrForce` If `TRUE`, then add the VC to the address specified in `ivcAddr`. If `FALSE`, then an address will be selected automatically and returned in `ivcAddr`
- `ivcAddr` SRAM address of the VC record

4.4.10 Case 3: Insertion at the Root of a Tree

Pseudocode:

```
BYTE ivcAddToRoot(
      UBYTE device,
      IVC_SEARCH_RESULTS_TYPE searchResults,
      IVC_RECORD_TYPE ivcRecord,
      UBYTE addrForce,
      UWORD *ivcAddr)

{
```

```
if addrForce=1 then mark vc record as used
else allocate a new vc record
ivcGetFreeSecondaryAddr() allocate a new secondary node
newRootNode.selector = searchResults.newSelector;
if bit#newSelector of secondaryKey =1 (new record goes on left)
    newRootNode.leftLeaf = 1;
    newRootNode.leftBranch = recordAddr;
    newRootNode.rightLeaf = 0;
    newRootNode.rightBranch = searchResults.finalNodeAddr;

else ( new record goes on right)
    newRootNode.rightLeaf = 1;
    newRootNode.rightBranch = recordAddr;
    newRootNode.leftLeaf = 0;
    newRootNode.leftBranch = searchResults.finalNodeAddr;

ivcUpdateRecord() /*setup new vc record*/
ivcUpdateSecondaryTable() /*setup new node (local and external)*/
ivcUpdatePrimaryTable() /*link in the new node (local and external)*/
}
```

Parameters:

- `Device` Device to access
- `SearchResults` Contains results from a VC binary tree search
- `IvcRecord` Record containing all fields of a vc record
- `AddrForce` If TRUE, then add the VC to the address specified in ivcAddr
- `IvcAddr` SRAM address of the vc record

4.4.11 Case 4: Insertion at Middle of a Tree

Pseudocode:

```
BYTE ivcAddToMiddleBranch(
      UBYTE device,
      IVC_SEARCH_RESULTS_TYPE searchResults,
      IVC_RECORD_TYPE ivcRecord,
      UBYTE addrForce,
      UWORD *ivcAddr)
```

```
{
if addrForce=1 then mark the record as used
else allocate a record
allocate a new secondary node

newNode.selector = searchResults.newSelector;
if bit#newSelector=1 /*goes on left*/
    newNode.leftLeaf = 1;
    newNode.leftBranch = recordAddr;
    newNode.rightBranch = searchResults.finalNodeAddr;
    newNode.rightLeaf = 0;
else /*goes on right*/
    newNode.rightLeaf = 1;
    newNode.rightBranch = recordAddr;
    newNode.leftBranch = searchResults.finalNodeAddr;
    newNode.leftLeaf = 0;

previousNode = gmIvcSecondaryTable[searchResults.previousAddr];
if previous direction=1 then go left
    previousNode.leftBranch = newNodeAddr;
else go right
    previousNode.rightBranch = newNodeAddr;

ivcUpdateRecord() /*setup the new record*/
ivcUpdateSecondaryTable() /*setup the new node    */
ivcUpdateSecondaryTable() /* update the parent node*/
}
```

Parameters:

- `Device` device to access
- `SearchResults` contains results from a VC binary tree search
- `IvcRecord` record of type IVC_RECORD_TYPE containing all fields of a vc record
- `addrForce` if TRUE, then add the VC to the address specified in ivcAddr
- `ivcAddr` SRAM address of the vc record

4.4.12 Case 5: Insertion at a Leaf

Pseudocode:

```
BYTE ivcAddToEndNode(
      UBYTE device,
      IVC_SEARCH_RESULTS_TYPE searchResults,
      IVC_RECORD_TYPE ivcRecord,
UBYTE addrForce,
UWORD *ivcAddr
{
if addrForce=1 then mark the record as used
else allocate a record
allocate a new secondary node

modify the final node to point to the new node:
modifiedNode = searchResults.finalNode;
if (searchResults.finalDirection)
    modifiedNode.leftLeaf = 0;
    modifiedNode.leftBranch = nextNodeAddr;
    }
else {
    modifiedNode.rightLeaf = 0;
    modifiedNode.rightBranch = nextNodeAddr;
    }

Set up the new end node:
nextNode.selector = searchResults.newSelector;
nextNode.leftLeaf = 1;
nextNode.rightLeaf = 1;
if bit#newSelector of secondaryKey=1 then /*new record goes on the
left*/
    nextNode.leftBranch = recordAddr;
    nextNode.rightBranch = searchResults.leafAddr;
else /*goes on the right*/
    nextNode.rightBranch = recordAddr;
    nextNode.leftBranch = searchResults.leafAddr;

ivcUpdateRecord() /*setup the new vc record*/
ivcUpdateSecondaryTable() /*add the setup the new node return*/
ivcUpdateSecondaryTable() /*modify the old node to link in the new*/
}
```

Parameters:

- `Device` device to access
- `SearchResults` contains results from a VC binary tree search
- `IvcRecord` record containing all fields of a vc record
- `AddrForce` if TRUE, then add the VC to the address specified in ivcAddr
- `IvcAddr` pointer, SRAM address of the vc record

4.4.13 Insertion Point

Pseudocode:

```
BYTES ivcFindInsertionPoint(
      UBYTE decisionBit,
      IVC_SEARCH_RESULTS_TYPE *searchResults)
{
while not foundPoint
   if selecor(newNode)>selector(thisNode)
         insertionPoint=thisNode
         foundPoint=TRUE
   thisNode=nextNode(left or right branch)
}
```

Parameters:

- `DecisionBit` Value to compare the node selector values to
- `SearchResults` Pointer to a record of IVC_SEARCH_RESULTS_TYPE

Remove Ingress VC

This function removes an ingress connection. This is performed through the following steps:

1. Search the associated binary tree to find the location of the Record within the tree.
   - If the VC Table Record is the only Record in the tree (Case 1), simply free up the memory associated with the VC Table Record and Secondary Tree node.
   - If there is only one node in the tree pointing to 2 VC Table records (Case 2), free up the memory associated with the VC Table Record being deleted. Modify the root node so that both left and right branches point to the other VC Table Record.
   - If the node is a root node (Case 3), modify the Primary Table to point to the next node down instead. Free up the memory associated with the VC Table Record and old root node.

- If the node is in the middle of the tree (Case 4), modify the previous node to point to the next node down. Free up the memory associated with the VC Table Record and old node.
- If the node is last in the tree (Case 5), replace the address of the node in the previous node with the address of the VC Table Record that is not being removed. Change the previous node's appropriate 'Leaf' bit to a 1.

4.4.14 Remove Connection

Pseudocode:

```
BYTE ivcRemoveVC(
      UBYTE device,
      IVC_RECORD_TYPE ivcRecord,
      UWORD *ivcAddr)
{
primaryKey=extracted from ivcRecord
SecondaryKey=extracted from ivcRecord

ivcSearch(device, primaryKey, secondaryKey, &searchResults)
if not searchResults.matched then return ERROR_IVC_VC_NOT_EXIST
if not searchResults.root then /* case 4&5 */
   ivcRemoveEndNode(device, searchResults)
else remove some part of the root node
   if (searchResults.singleBranch = TRUE) then  /* case 1 */
         call ivcRemoveSingleBranch(device, searchResults)
         if not SUCCESS then return error code from
ivcRemoveSingleBranch()
   else if (searchResults.finalNode.leftLeaf =1) and
                              (searchResult.finalNode.rightLeaf=1)
                        then
         call ivcRemoveDoubleBranch(device, searchResults)  /* case 2
*/
         else (must remove the root node) /* Case 3 */
            call ivcRemoveRootNode(device, searchResults)
return SUCCESS
}
```

Parameters:

- `device` Device to access
- `ivcRecord` The VC Table Record to be removed

- ivcAddr The address of the VC Table record to be removed

4.4.15 Case 1: Removing a Single Branch

Pseudocode:

```
BYTE ivcRemoveSingleBranch(
      UBYTE device,
      IVC_SEARCH_RESULTS_TYPE searchResults)
{
ivcUpdatePrimaryTable() /*unlink the root node*/
release the vcRecord
release the secondaryNode
return SUCCESS
}
```

Parameters:

Device Device to access

PrimaryKey Key to search in the primary table

4.4.16 Case 2: Removing From a Double Branch

Pseudocode:

```
BYTE ivcRemoveDoubleBranch(
      UBYTE device,
      UWORD primaryKey,
      IVC_SEARCH_RESULTS_TYPE searchResults)
{
release the vc record
modifiedNode = searchResults.lastNode (modify the root node)
   if searchResults.lastDirection = 1 then /*left*/
         modifiedNode.rightBranch = modifiedNode.leftBranch
   else (right)
         modifiedNode.leftBranch = modifiedNode.rightBranch
Call ivcUpdateSecondaryTable() /*unlink the old VC record*/
}
```

Parameters:

`Device` Device to access

`PrimaryKey` Key to search in the primary table

`SearchResults` Results from ivcSearch

4.4.17 Case 3 Removing the Root Node

Pseudocode:

```
BYTE ivcRemoveRootNode(
      UBYTE device,
      IVC_SEARCH_RESULTS_TYPE searchResults)
{
if searchResults.lastDirection = 1 then
newRoot = searchResults.lastNode.rightBranch
   else newRoot = searchResults.lastNode.leftBranch

ivcUpdatePrimaryTable() /*point to the new root node*/
release the VC record
release the secondary node
return SUCCESS
}
```

Parameters:

`Device` Device to access

`SearchResults` Results from ivcSearch

4.4.18 Case 4 & 5: Removing a Node From the End of a Tree

Pseudocode:

```
BYTE ivcRemoveEndNode(
      UBYTE device,
      IVC_SEARCH_RESULTS_TYPE searchResults)
{
if the parent node is a left child of the grandparent /*left*/
   newNode.leftLeaf = oldLeaf
      newNode.leftBranch = oldBranch
else /*right*/
```

```
   newNode.rightLeaf = oldLeaf
      newNode.rightBranch = oldBranch

ivcUpdateSecondaryTable() /* modify the grandparent node*/
release the vc record
release the old secondary node
return SUCCESS
}
```

Parameters:

`Device`	Device to access
`SearchResults`	Results from ivcSearch

Add Egress VC

4.4.19 Initialization

Pseudocode:

```
BYTE ivcInitialize(void)
{
evcramWrite() /*clear the active bit of each VC record*/
for each vcra
    gmEvcRecordAddrExists[loopIndex] = FALSE;
    evcramWrite() to clear the active bit of each VC record
return SUCCESS
}
```

4.4.20 Add VC

```
BYTE evcAddVC(
      UBYTE device,
      EVC_RECORD_TYPE evcRecord,
      UWORD *evcAddr)
{
*evcAddr=(inPhyID+fieldA+fieldB)
gmevcRecordAddrExists[*evcAddr]=TRUE
evctableUpdateRecord() to map evcRecord into SRAM
return SUCCESS
}
```

Parameters:

- `device` Device to access
- `evcAddr` The address of the VC Record
- `evcRecord` The VC Table Record to be added

Remove Egress VC

Pseudocode:

```
BYTE evcRemoveVC(
      UBYTE device,
      IVC_RECORD_TYPE evcRecord,
      UWORD *evcAddr)
{
*evcAddr=(inPhyID+fieldA+fieldB)
evcRamWrite() /*clear active bit*/
gmEvcRecordAddrExists[*evcAddr] = FALSE
return SUCCESS
}
```

Parameters:

- `device` Device to access
- `evcAddr` The address of the VC Table removed
- `evcRecord` The VC Table Record to be removed

5 Programming OAM

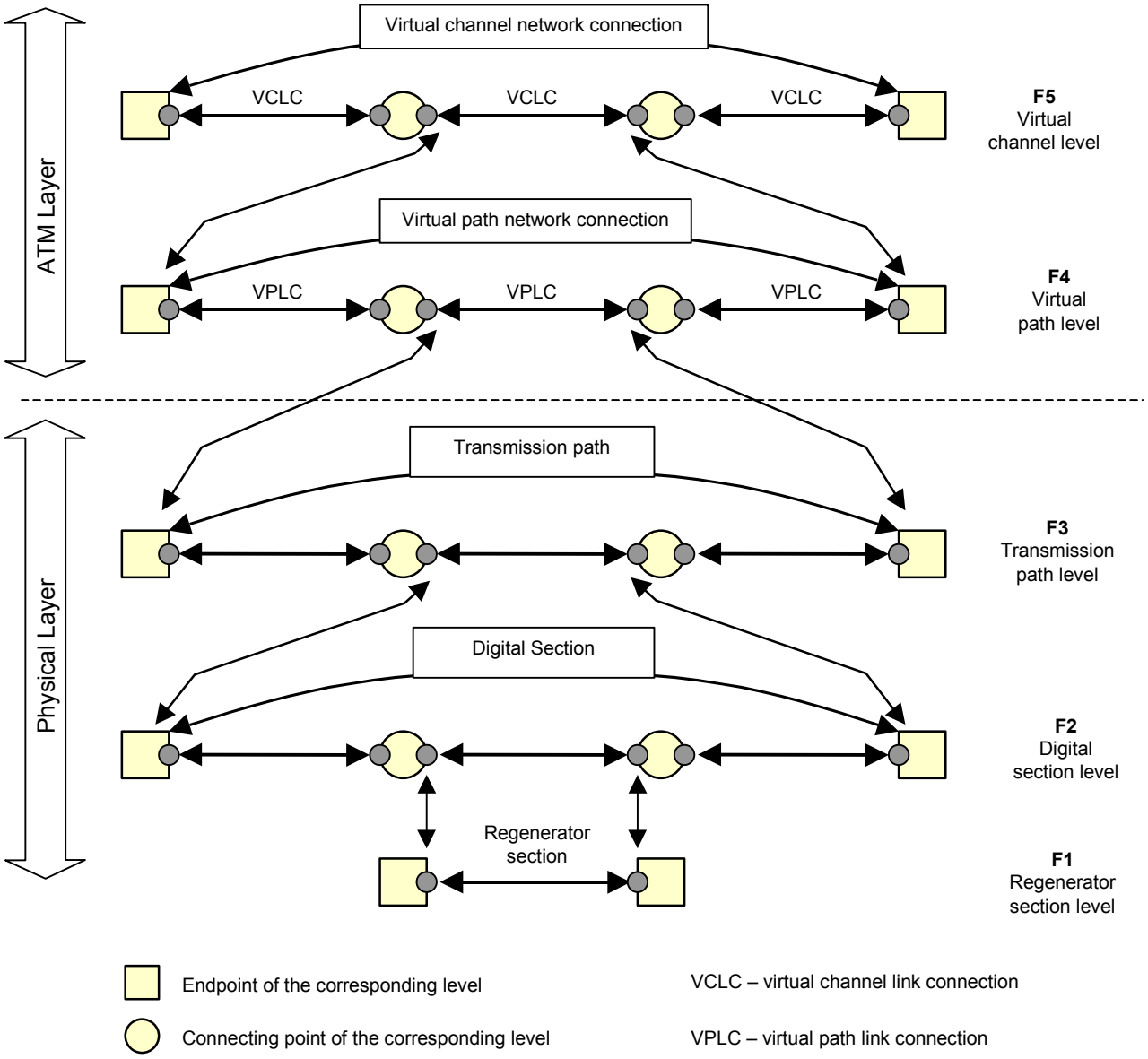
This section introduces the Operations and Maintenance (OAM) ATM flow and how to enable these functions using the PM7324 S/UNI-ATLAS device.

5.1 OAM Flows

The OAM functions in the network are performed on five OAM hierarchical levels associated with the ATM and physical layers of the protocol reference model. The functions result in corresponding bidirectional information flows on five different levels. A pictorial explanation of the levels is shown in Figure 12 below. The levels are as follows:

- F5, Virtual channel (VCC) level: Extends between network elements performing virtual channel connection termination and is shown extending through one or more virtual paths.
- F4, Virtual path (VPC) level: Extends between network elements performing virtual path connection termination and is shown extending through one or more transmission path.
- F3, Transmission path level: Extends between network elements assembling/disassembling the payload of a transmission system and associating it with its OAM functions.
- F2, Digital section level: Extends between section endpoints and comprises a maintenance entity.
- F1, Regenerator section level: A regeneration section (e.g. usually SONET optical interface) is a portion of a digital section and as such is a maintenance sub-entity.

Figure 12 ATM OAM Hierarchical Levels



Note

1. The F5 flow does not necessarily extend beyond the end point of F4 flow.

Levels F1, F2 and F3 are physical layer (SONET) OAM flows, while F4 and F5 flows are the ATM layer OAM flows. ATM Layer OAM flows are necessary, since a virtual connection may extend beyond the SONET network.

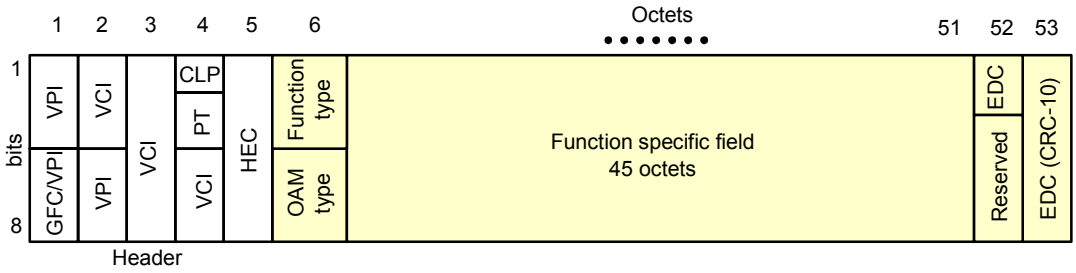
F4 flows refer to OAM flows over a VPC. F5 flows refer to OAM flows over a VCC. An OAM flow may extetend over the entire connection. This is called an end-to-end flow. An OAM flow may also cover a portion of the end to end connection. This is called a segment flow. Therefore, there are four main types of OAM flows:

- F4 End-to-end
- F4 Segment
- F5 End-to-end
- F5 Segment

5.1.1 OAM Cell Format

Figure 13 below shows an ATM OAM Cell. The total size of any ATM Cell (including OAM cell) is 53 octets. In the OAM cell, the forty-eight octets of the user payload are replaced with the OAM specific data.

Figure 13 OAM Cell Structure



The OAM cell is distinguished from a user cell by the VCI (in the case of F4 OAM cells), and the PT (in the case of F5 OAM cells). Table 5 below shows the use of the VCI and PT fields for F4 and F5 ATM cells. Table fields related to the OAM cells are highlighted.

Table 5 Cells at F4 and F5 Flow Level

F4		F5	
VCI	**Interpretation**	**PT**	**Interpretation**
0	Unassigned cell (VPI=0)	000	User data cell, congestion not experienced
0	Unused (VPI>0)	001	
1	Meta-signaling cell (UNI)	010	User data cell, congestion experienced
2	General broadcast signaling cell (UNI)	011	
3	Segment OAM F4 flow call	100	Segment OAM F5 flow call
4	End-to-end OAM F4 flow call	101	End-to-end OAM F5 flow call
5	Point-to-point signaling cell	110	Resource management cell
6	Resource management cell	111	Reserved for future use
7-15	Reserved for future use.		
16-31	Reserved for future use.		

F4		F5
>31	Available for user data transmission	

The first byte of the cell payload indicates the OAM cell type and function type. There are 4 cell types defined: Fault Management (FM), Performance Management (PM), Activation/Deactivation (A/D), and System Management (SM). For each cell type, there are several Function Types. For instance, a FM cell may be AIS, RDI, CC, or LB. The various OAM cell types and function types are detailed in below.

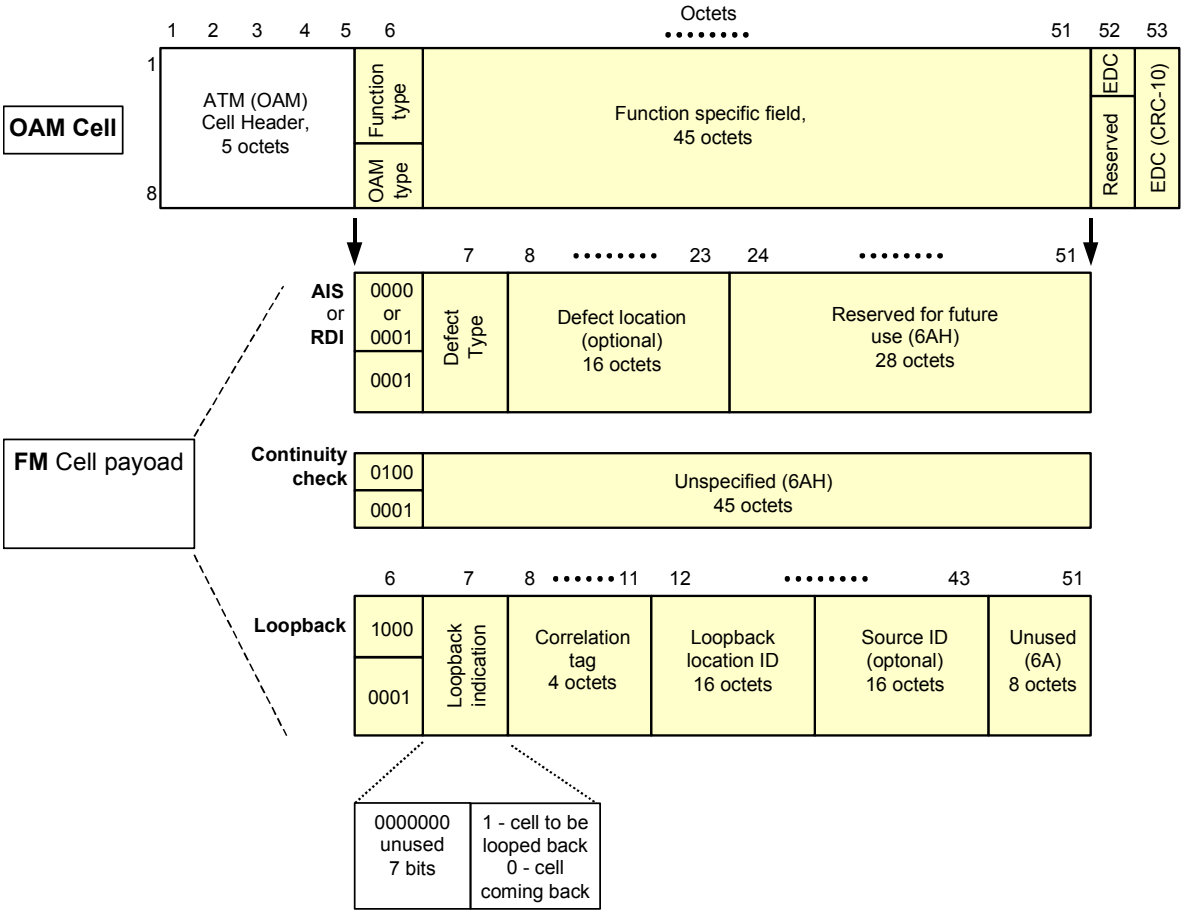
Table 6 OAM Cell Types and Functions

OAM Cell Type		OAM Function Type		Description
Fault Management	0001	AIS (alarm indication signal)	0000	For reporting defect indications in the forward direction
		RDI (remote defect indication)	0001	For reporting remote defect indications in the backward direction
		CC (Continuity check)	0100	For continuously monitoring the availability of a link
		LB (Loopback)	1000	For on-demand connectivity monitoring, fault localization, and pre-service connectivity verification
Performance Management	0010	Forward monitoring	0000	For estimating performance over a link or segment of a link
		Backward reporting	0001	For reporting performance estimations on the backward direction
Activation/ Deactivation	1000	Performance Management A/D Forward Monitoring and backward Reporting	0000	For activation/deactivation of PM functionality in a standard way
		Continuity check A/D	0001	For activation/deactivation of CC functionality in a standard way
		Forward monitoring A/D	0010	For activation/deactivation of FM functionality in a standard way
System Management	1111	Not specified in I.610 (1998)		

10 bits at the end of the OAM cell are dedicated to the EDC. The EDC is calculated over the cell payload. The EDC is used protect against erroneous decisions based on corrupted OAM cell data.

5.1.2 Fault Management Cells

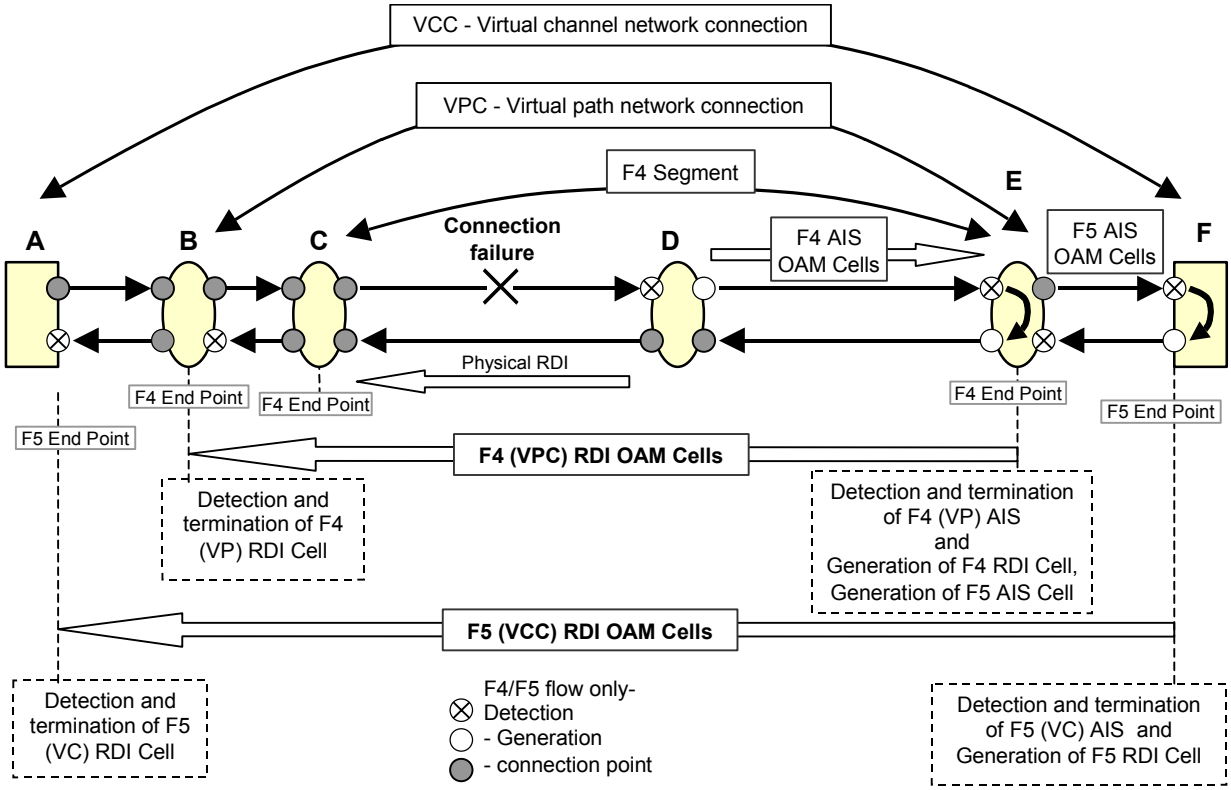
Figure 14 FM Cell Function Specific Fields



AIS and RDI Cells

AIS cells are used to report defects in the downstream direction. RDI cells are used to report defects in the upstream direction. An example of the AIS and RDI flow is shown in Figure 15 below.

Figure 15 AIS and RDI Flow



The example shown above has the OAM flows set as follows:

- "A" to "F" is an F5 End-to-End flow
- "B" to "E" is an F4 End-to-End flow
- "C" to "E" is a Segment F4 flow
- "D" is a connection point (non-end point to either flow).

Consider a failure between "C" and "D". The network will react as follows:

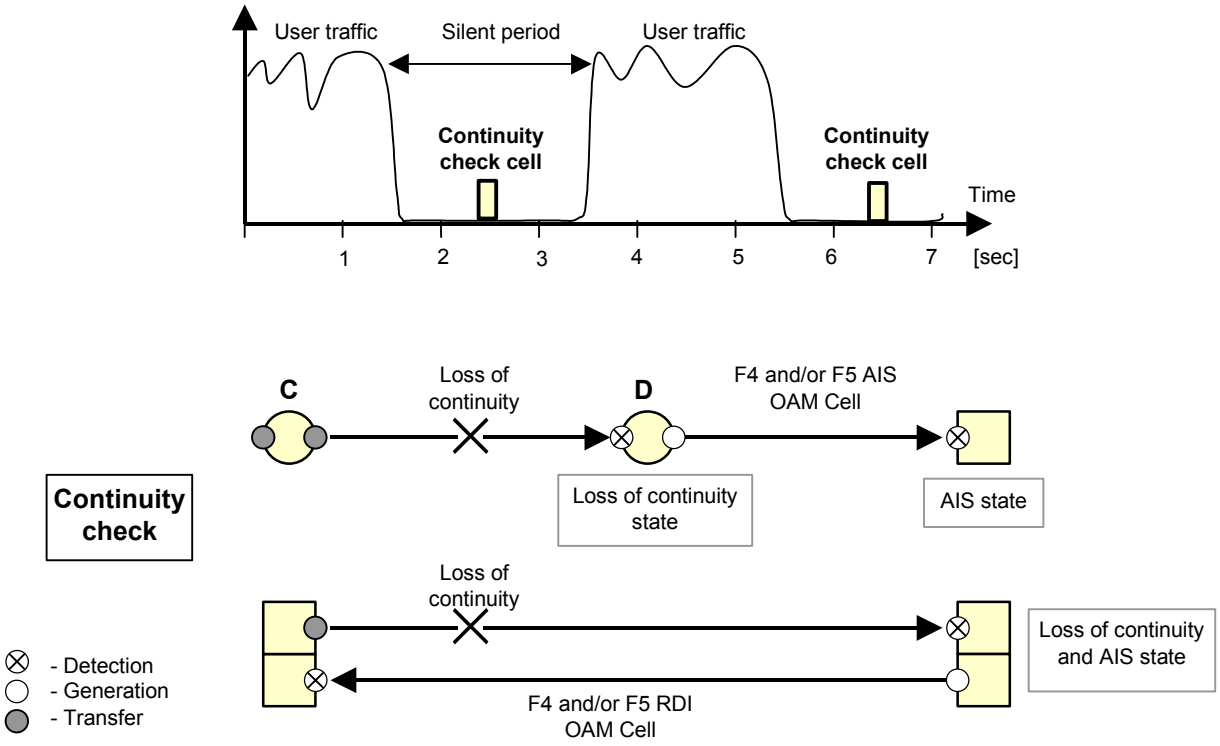
- The failure is detected at "D" . It may be detected at the physical layer or at the ATM layer.
- "D" generates Segment and End-to-End F4 AIS cells downstream once per second. "D" does not generate RDI nor F5 AIS, since it is not an end-point for the F4 or F5 flows. "D" may generate physical layer RDI, if appropriate.
- "E" is both a segment and end-to-end flow end point, and therefore terminates both F4 AIS cell flows. In response to the AIS cells, "E" generates F4 Segment and End-to-End RDI cells upstream. Additionally, since E is also a connecting point for the F5 flow, F5 End-to-End AIS cells are sent downstream once per second.

- "F" is end-to-end flow end point, and therefore terminates the F5 End-to-End AIS. In response to the AIS, "F" gernerates F5 End-to-End RDI cells upstream.
- "E" does not terminate the F5 End-to-End RDI generated at "F", since it is not an end point for the F5 OAM flow.
- "D" does not terminate OAM cells since it is not an end point.
- "C" terminates the F4 Segment RDI generated by "E".
- "B" terminates the F4 End-to-End RDI generated by "E".
- "A" terminates the F5 End-to-End RDI cells generated by "E".
- "A", "B" and "C" do not generate any OAM cells related to this connection failure.

CC Cells

Continuity failures at the ATM layer are detected using Continuity Check (CC) cells. Continuity check cells are inserted into a connection so the downstream entity may differentiate between a loss of continuity and a period of low cell flow.

Figure 16 CC Flow

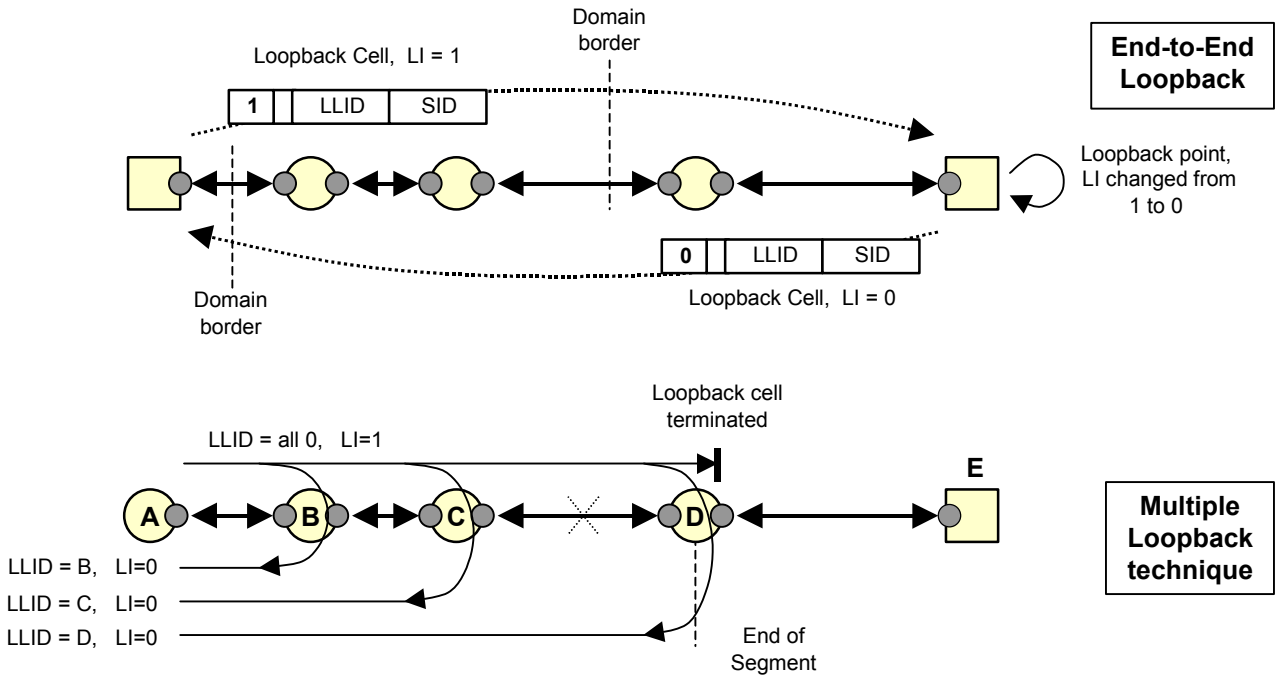


Connection points (intermediate nodes) in a segment and/or end-to-end OAM flow (F4 and F5) can be set to look for the presence of CC cells. If a lack of user cells and CC cells is detected over 3.5 ±0.5sec, a network entity may trigger CC alarm, which in turn may activate sending of AIS cells downstream. The CC alarm triggered at the flow end-points may also activate sending RDI cells in upstream direction.

Loopback Cells

Another type of fault management cell is the loopback cell. The intent of the loopback cell is to determine continuity in a connection after it has been setup, to isolate misconfiguration problems. This is a demand service used by network operators. Examples of loopback flows are shown in Figure 17 below.

Figure 17 Loopback Flow Examples



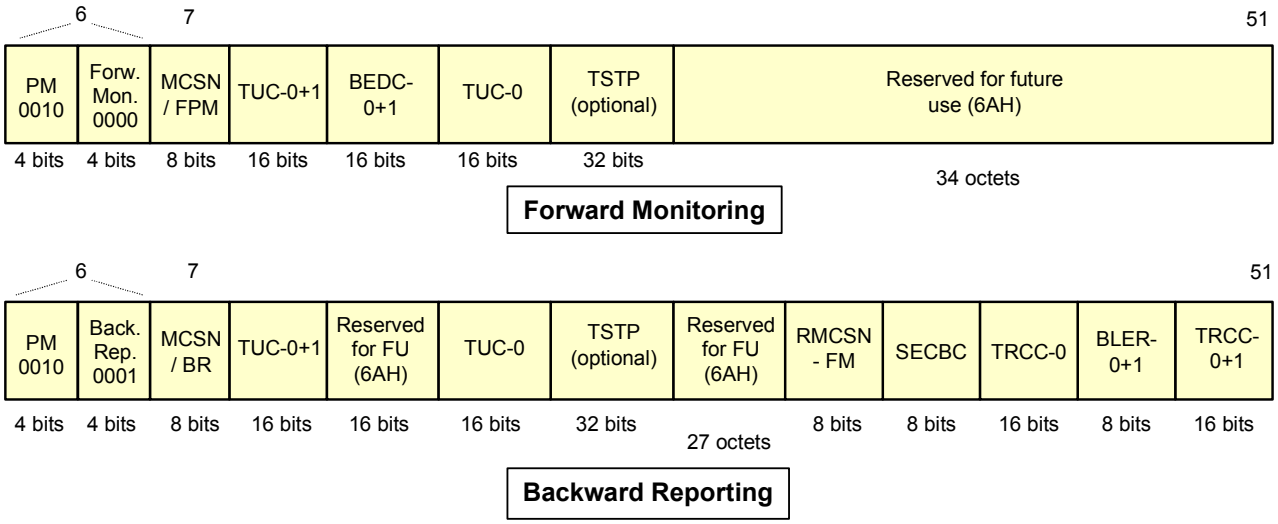
The simplest use of the loopback cell is shown in the top half of Figure 17. The loopback cell is sent from one end of the connection and looped back at the end of the connection.

A multiple loopback technique can be used to simplify the segment diagnostic process (as shown above). In this case the Loopback Location ID (LLID) field of a cell generated at point "A" is filled with all 0's. Each connection point in the segment sends the loopback cell back with the LLID changed to that node's ID and loopback indicator (LI) changed to 0. The loopback cell originating point "A" can receive, for example, three cells. If connection C←→D is broken, point A receives only two loopback cells back (B anad C). The segment loopback cell is terminated at the segment end point (D).

5.1.3 Performance Management Cells

Performance Management (PM) cells are used to determine the performance of a particular VC. PM is generally a demand service initiated by the network operator. Two types of PM cells are defined: forward monitoring and backward reporting. The function specific fields of the PM cell are shown in Figure 18 below.

Figure 18 PM Cell Function Specific Fields



A full description of each field is presented in I.610. A brief summary is presented below.

Forward Monitoring

- MCSN/FPM Monitoring Cell Sequence Niumber, forward PM
- TUC0+1 Total User Cells, CLP0+1
- BEDC0+1 Block Error Detection Code, CLP0+1
- TUC0 Total User Cells, CLP0
- TSTP Timestamp

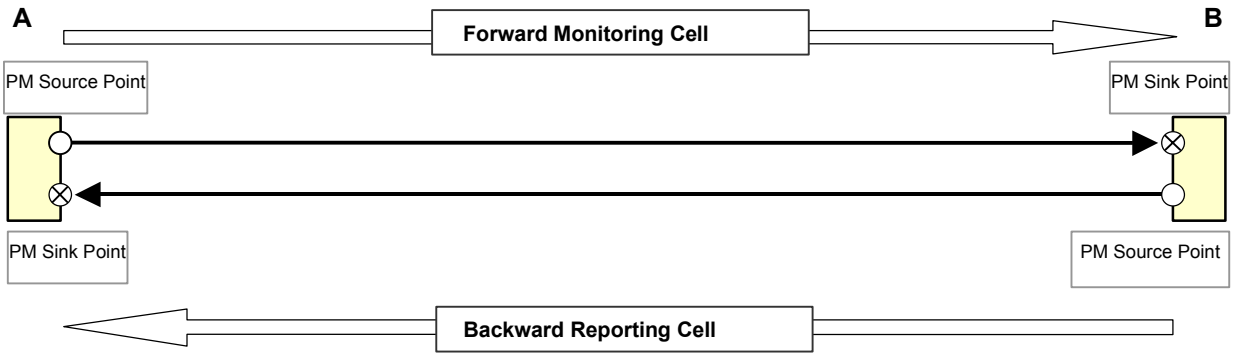
Backward Reporting

- MCSN/BR Monitoring Cell Sequence Niumber, Backward Monitoring Cell
- TUC0+1 Total User Cells, CLP0+1
- TUC0 Total User Cells, CLP0
- TSTP Timestamp
- RMCSN-FM Reported Monitoring Cell Sequence Number, Forward Monitoring Cell
- SECBC Severely Errored Cell Block Count

- TRCC0 Total Received Cell Count, CLP0
- BLER0+1 Block Error Result, CLP0+1
- TRCC0+1 Total Received Cell Count, CLP0+1

The PM cells are sent at the source point, and terminated and processed at the sink points for segment and end-to-end flows on VPCs and VCCs. The PM cells may also be monitored at intermediate nodes. Unlike FM flows, there is no interaction between the F4 and F5 levels for PM flows. Figure 19 below shows how the PM flow works. The figure applies to F4 segment, F4 end-to-end, F5 segment, and F5 end-to-end flows.

Figure 19 Example of PM Cell Flow



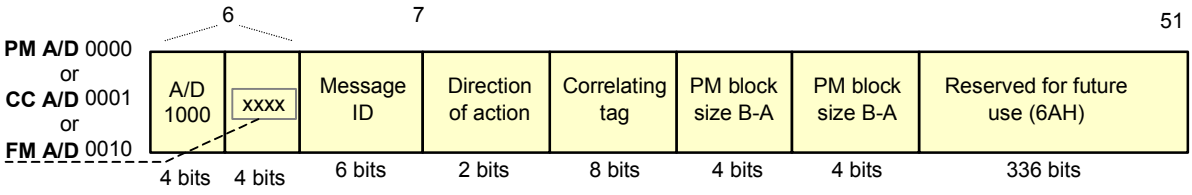
Every *n* user cells (where *n* is the block size, defined by I.610), a forward PM cell will be generated at the PM source point. The end point for the PM flow (the "sink") will terminate the PM cell, process the contents, and generate a backward PM cell in response.

5.1.4 Activation/Deactivation Cells

The Activation/Deactivation (A/D) process is use to activate and deactivate PM or CC flow sessions through the ATM network. The advantage of the A/D process is that it uses the same network that carries user traffic. At the same time, it may be a major disadvantage. For instance, if a particular connection is not performing well, it may be difficult to reliably pass A/D cells to setup a PM session.

The function specific fields of the A/D cell is shown in Figure 20 below.

Figure 20 A/D Cell Function Specific Fields



A full description of each field is presented in I.610. A brief summary is presented below.

- Message ID Specifies the function of the cell. Refer to I.610 for the use of the message ID field.
- Direction of Action Identifies direction(s) of transmission.
- Correlation Tag A tag generated for each message, used by nodes to correlate commands with responses.
- PM Block Size A-B Specifies the A-B block size for forward monitoring.
- PM Block Size B-A Specifies the B-A block size required for backward reporting. The PM block size is always 2^N, where $7 \leq N \leq 15$.

5.2 Configuring the ATLAS for OAM processing

The ATLAS is capable of supporting all the functions shown in section 5.1. The sections that follow illustrate how to configure the ATLAS. Where possible, specific examples are given. No attempt is made to perform any type of interrupt processing, as interrupt handling is system specific.

The ATLAS is capable of terminating F4 and F5, segment and end-to-end, OAM flows. The hierarchical level of the cells terminated is determined by the connection type (VPC or VCC). The flow type is determined by the configuration of the connection. This is configured by the bits in Table 7 below.

Table 7 OAM Cell Termination Bits

VC Table Record Field	Bit	Function
OAM Configuration (ingress)	Segment_Point	Use this to terminate segment OAM cells
	End-to-End_Point	Use this to terminate end-to-end OAM cells
OAM Configuration (egress)	Segment_Point	Use this to terminate segment OAM cells
	End-to-End_Point	Use this to terminate end-to-end OAM cells

5.2.1 Fault Management Cell Processing

The ATLAS can generate segment or end-to-end, AIS, RDI, and CC cells on a per-connection basis. The ATLAS also supports automatic generation of RDI in response to received AIS. Additionally, automatic extraction and insertion of defect location and type are supported. The VC table record and normal mode register bits in below are used to configure the fault management functions. AIS and CC cells generated at the ingress are sent to the ingress output cell interface, while RDI cells generated at the ingress are sent to the egress output cell interface. Conversely, AIS and CC cells generated at the egress are sent to the egress output cell interface, while RDI cells generated at the egress are sent to the ingress output cell interface.

Table 8 Fault Management Configuration for Ingress

VC Table Record Field or Normal Mode Register	Bit	Function
OAM Configuration	Send_AIS_segment	Set these bits to send segment or end-to-end AIS or RDI cells. The cells will be sent once per second until the appropriate bit is turned off. Send_RDI is not necessary if the AUTORDI and CC_RDI functions are enabled. Send_AIS would normally be used at non-end points to indicate failures downstream.
	Send_AIS_end-to-end	
	Send_RDI_segment	
	Send_RDI_end-to-end	
	CC_RDI	Use this to automatically generate RDI upstream upon declaration of a continuity check alarm. RDI cells are only generated at flow end points.
	CC_Activate_segment	Activates segment continuity checking.
	CC_Activate_end-to-end	Activates end-to-end continuity checking.
Configuration	Defect_Type[3:0]	Selects of one of 16 defect types (see registers 0x226 – 0x22D)
Miscellaneous	PHYID[4:0]	This should be set to reflect the PHYID of the connection.
	Received end-to-end AIS defect Location[127:0] Received end-to-end AIS defect type[7:0] Received segment AIS defect Location[127:0] Received segment AIS defect type[7:0]	When the ATLAS terminates an AIS cell, the defect location and type are extracted and placed in the VC table. This information may be used to determine the location and nature of the fault. Additionally, the information is used for the defect location and type in RDI cells generated by the ATLAS using the AUTORDI function.
0x200	AUTORDI	Set this bit to automatically generate RDI upon termination of an AIS cell.
0x221	AISCCCP[15:0]	AIS and CC cell pacing limits the effects of cell generation on the switch fabric. The setting of this register is system specific.

VC Table Record Field or Normal Mode Register	Bit	Function
0x222 – 0x225	AIS31 – AIS16 AIS15 – AIS0 RDI31 – RDI16 RDI15 – RDI0	AIS and RDI cells may be generated on a per-PHY basis. This would normally be done in the event of a physical layer failure. Rather than set Send_AIS_end-to-end or Send_RDI_end-to-end for each affected connection, the ATLAS will automatically generate AIS or RDI cells on PHYx if AISx or RDIx is set.
0x226 – 0x22D	DT0[7:0] – DT15[7:0]	The ATLAS can store up to 16 defect types that may be used for non-automatic (ie, not as a result of AUTORDI) cell generation. The setting of these bits is system specific.
0x22E – 0x235	DL[127:0]	The defect location is inserted into non-automatic (ie, not as a result of AUTORDI) cell generation. The setting of these registers is system specific. It is expected that this would be set to some unique value within the network.
0x238	ForceCC	This controls the mode of operation of continuity checking. If CC cells are to be inserted only in periods of low user bandwidth, then ForceCC should be set to 0. If CC cells are to be inserted regardless of user bandwidth, then this bit should be set to 1.
	IAISCOPY	This should be set if ingress SRAM at ISA[19:16]=1010 – 1110 is populated.
0x23F	MAX	This should be set to the maximum depth of SRAM. For instance, if 8K VCs are supported, this should be set to 0x1FFF.
0x240 – 0x241	APS31 – APS0	The ATLAS can automatically propogate a segment AIS flow into an end-to-end AIS flow at a segment end point. This is performed only if APSx=0, and if end-to-end AIS cells are not currently being generated.

Table 9 Fault Management Configuration for Egress

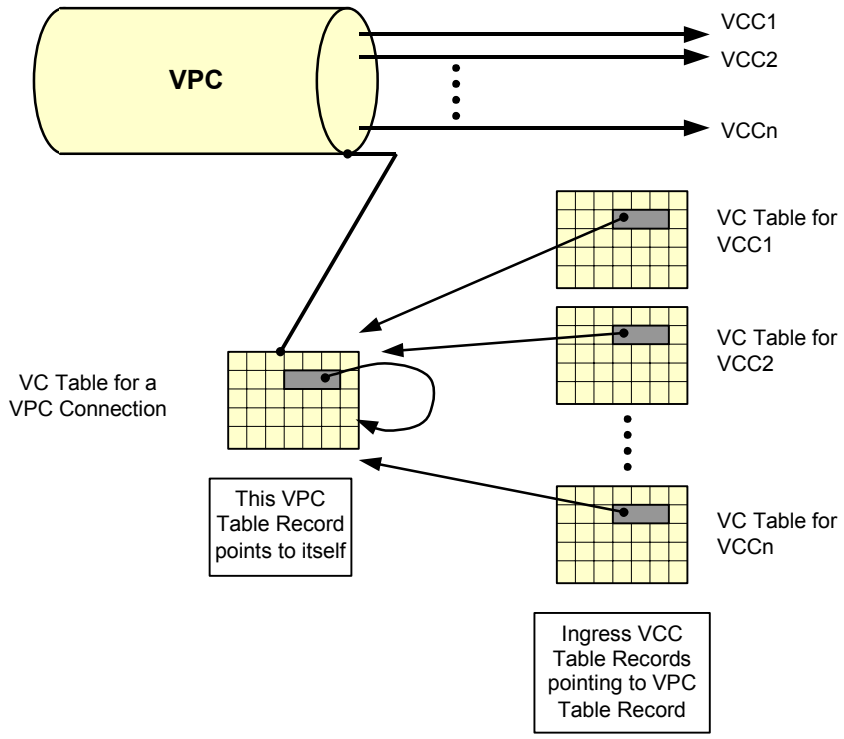
VC Table Record Field or Normal Mode Register	Bit	Function
0x280	AUTORDI	Set this bit to automatically generate RDI upon termination of an AIS cell.
0x283	ForceCC	This controls the mode of operation of continuity checking. If CC cells are to be inserted only in periods of low user bandwidth, then ForceCC should be set to 0. If CC cells are to be inserted regardless of user bandwidth, then this bit should be set to 1.
	EAISCOPY	This should be set if egress SRAM at ESA[19:16]=1xxx is populated.

VC Table Record Field or Normal Mode Register	Bit	Function
0x286	BCP[15:0]	Backward cell interface pacing limits the effects of cells from the egress backward cell interface on the switch fabric. The setting of this register is system specific.
0x287	HBTO[15:0]	If a PHY at the egress output cell interface fails, OAM cells in the egress backward cell interface destined for that PHY may cause head of line blocking in that FIFO. The timeout function allows those cells to be discarded after spending a certain amount of time in the FIFO.
0x288	AISCCCP[15:0]	AIS and CC cell pacing limits the effects of cell generation on the PHY. The setting of this register is system specific.
0x28A - 0x28D	AIS15 – AIS0 AIS31 – AIS16 RDI15 – RDI0 RDI31 – RDI16	AIS and RDI cells may be generated on a per-PHY basis. This would normally be done in the event of a physical layer failure. Rather than set Send_AIS_end-to-end or Send_RDI_end-to-end for each affected connection, the ATLAS will automatically generate AIS or RDI cells on PHYx if AISx or RDIx is set. Note that since this is a symetric feature in the ATLAS, this could equivelently be done at the ingress side of an ATLAS on the other side of the switch fabric.
0x28E – 0x28F	APS15 – APS0 APS31 – APS16	The ATLAS can automatically propogate a segment AIS flow into an end-to-end AIS flow at a segment end point. This is performed only if APSx=0, and if end-to-end AIS cells are not currently being generated.
0x292 – 0x299	DT0 – DT15	The ATLAS can store up to 16 defect types that may be used for non-automatic (ie, not as a result of AUTORDI) cell generation. The setting of these bits is system specific.
0x29A - 0x2A1	DL[15:0]	The defect location is inserted into non-automatic (ie, not as a result of AUTORDI) cell generation. The setting of these registers is system specific. It is expected that this would be set to some unique value within the network.
0x2AA	MAX[15:0]	This should be set to the maximum depth of SRAM. For instance, if 8K VCs are supported, this should be set to 0x1FFF.
OAM Configurattion	Send_AIS_segment	Set these bits to send segment or end-to-end AIS or RDI cells. The cells will be sent once per second until the appropriate bit is turned off. Send_RDI is not necessary if the AUTORDI function is enabled. Send_AIS would normally be used at non-end points to indicate failures downstream.
	Send_AIS_end-to-end	
	Send_RDI_segment	
	Send_RDI_end-to-end	

VC Table Record Field or Normal Mode Register	Bit	Function
	CC_RDI	Use this to automatically generate RDI upstream upon declaration of a continuity check alarm. RDI cells are only generated at flow end points.
	CC_Activate_segment	Activates segment continuity checking.
	CC_Activate_end-to-end	Activates end-to-end continuity checking
Configuration	Defect_Type[3:0]	Selects of one of 16 defect types (see registers 0x292 – 0x299)
Miscellaneous	PHYID[4:0]	This should be set to reflect the PHYID of the connection.
Miscellaneous	Received end-to-end AIS defect Location[127:0] Received end-to-end AIS defect type[7:0] Received segment AIS defect Location[127:0] Received segment AIS defect type[7:0]	When the ATLAS terminates an AIS cell, the defect location and type are extracted and placed in the VC table. This information may be used to determine the location and nature of the fault. Additionally, the information is used for the defect location and type in RDI cells generated by the ATLAS using the AUTORDI function.

When a VPC is terminated and broken into its constituent VCCs, the ATLAS can terminate the F4 fault management flow and create an F5 flow for each of the VCCs.. Each VCC of the VPC must be setup as a separate connection in the Ingress VC Table. An additional F4 connection must also be setup. Each of the VCCs is configured to have its VPC Pointer[15:0] (in the ingress VC table record) point to the SRAM address of the parent VPC. The parent VPC does not carry any user traffic, it only terminates and monitors the F4 OAM flow. below illustrates the process.

Figure 21 F4 to F5 Processing (Ingress only)



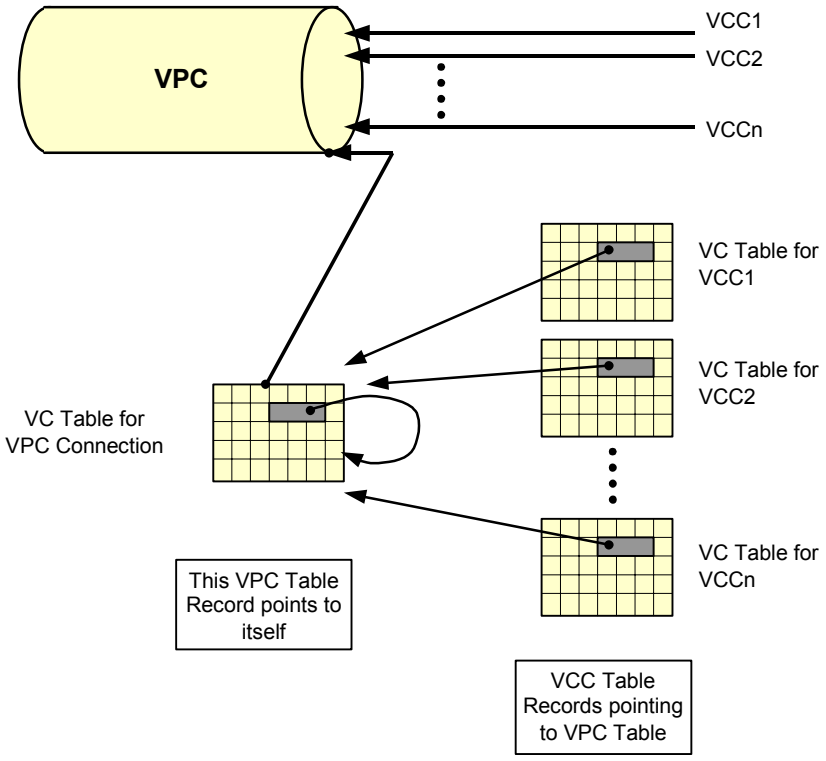
The VC Table Record bits related to the F4-to-F5 AIS cell process are shown in Table 10 below.

Table 10 VC Table Fields for F4-toF5 Processing

VC Table Record Field	Bit	Function
Miscellaneous	F4toF5AIS	Set this bit to enable the F4 to F5 process
	VPC Ponter[15:0]	For each connection associated with the VPC, the VPC Pointer should be set to the VCRA of the parent VPC. If F4 to F5 processing is not required, then the VPC pointer should be set to the address of the VCC.
Configuration (ingress)	SegmentFlow	This bit should be set if this point (where the VPC terminates) is between F5 segment end points for this VCC. That is, the point where the VPC terminates is not the origin of the VCC. This is the case where the VCC extends beyond the end-to-end end points of the VPC.

At the egress side of the ATLAS, the opposite process is performed. Cells received on each of the child VCCs update the parent VPC so that continuity checking is correctly performed. The F5 to F4 process is illustrated in Figure 22 below.

Figure 22 F5 to F4 Processing (Egress only)



The ATLAS does not automatically support loopback cell operations. Loopback cells may be dropped to the microprocessor cell interface, processed externally, and inserted through the microprocessor interface.

5.2.2 Performance Management Cell Processing

Figure 19 showed an example of performance management flow between two points. The flow provides coverage of the link between A and B using the forward monitoring flow, with feedback over the link from B to A using the backward reporting flow. In practice, this type of flow is duplicated so that there is a forward flow from B to A with a backward reporting flow from A to B. The ATLAS supports this type of operation. Further, the ATLAS may support this type of operation upstream and downstream *simultaneously.* That is, the ATLAS may simultaneously terminate forward monitoring and backward reporting PM cells at both the ingress and egress input cell interfaces, *and generate* forward monitoring and backward reporting PM cells toward both the ingress and egress output cell interfaces. This can be performed on any 128 of the possible 64K connections that the ATLAS supports. The parameters for each PM session are stored in on-chip SRAM.

PM RAM RAM Access

PM RAM is accessed in much the same way as VC Table SRAM. There are 2 banks of PM RAM at the ingress and 2 banks of PM RAM at the egress. PM RAM access is provided indirectly through the microprocessor interface. Table 11 below lists the registers that are relevant to PM RAM accesses in the ATLAS. For a detailed description of these registers, refer to the S/UNI-ATLAS Datasheet.

Table 11 PM RAM Access Registers

Address	Register
0x211	Ingress Performance Monitoring RAM Record Address
0x212	Ingress Performance Monitoring RAM Word Select and Access Control
0x213 - 217	Ingress Performance Monitoring RAM Data
0x2CE	Egress Performance Monitoring RAM Record Address
0x2CF	Egress Performance Monitoring RAM Row Select and Access Control
0x2D0	Egress Performance Monitoring RAM Write Mask
0x2D1 – 2D5	Egress Performance Monitoring RAM Data Word

Unlike the VC Table SRAM, each row of the PM RAM must be accessed individually. To perform a write, execute the following steps:

1. Check that the BUSY bit in the PM RAM Access Control Register is deasserted. Do not proceed if the BUSY bit is asserted.

2. Write data to the PM RAM Data Registers for the fields required.

3. Write the PM RAM address and bank to be accessed to the PM RAM Record Address Register.

4. Perform a write to the PM RAM Access Control Register with the RWB bit set to 0 and PM Row[2:0] set to the appropriate value. This will initiate the access and assert the BUSY bit. When the BUSY bit is deasserted, the SRAM access is complete.

To perform a read, execute the following steps:

1. Check that the BUSY bit in the PM RAM Access Control Register is deasserted. Do not proceed if the BUSY bit is asserted.

2. Write the PM RAM address and bank to be accessed to the PM RAM Address Register.

3. Perform a write to the PM RAM Access Control Register with the RWB bit set to 1 and PM Row[2:0] set to the appropriate value. This will initiate the access and assert the BUSY bit. When the BUSY bit is deasserted, the SRAM access is complete.

4. Read the data from the External RAM Data Registers for the fields required.

PM Configuration

Below VC Table fields and normal mode registers for configuration of PM functions.

Table 12 PM Configuration

VC Table Record Field or Normal Mode Register	Bit	Function
Miscellaneous	PM Addr1[6:0]	These fields should be programmed with the PM RAM address(s) related to this connection..
	PM Addr2[6:0]	
	PM Active 1	These bits indicate whether the PM Addr1 and PM Addr2 are active for this connection. These bits should not be set until all the parameters for the PM session are configured. These fields apply equally to both ingress and egress VC Table records.
	PM Active2	
0x218, 0x21A, 0x21C, 0x21E	MERROR[3:0]	These registers contain the thresholds required to count the errors in PM RAM. For instance, if MMISINS[11:0]=0x10A, then 266 misinserted cells would be required in a single PM block to increment the SECB Misinserted count by one. The setting of these registers is system specific.
	MMISINS[11:0]	
0x219, 0x21B, 0x21C, 0x21F	MLOST[11:0]	
0x2A2, 0x2A4, 0x2A6, 0x2A8	MERROR[3:0]	These registers serve the same function as the ingress registers 0x218 – 0x21F.
	MLOST[11:0]	
0x2A3, 0x2A5, 0x2A7, 0x2A9	MMISINS[11:0]	
0x23E, 0x289	FWDPMP[15:0]	As with generated FM cells, forward PM cells may also be paced to relieve the switch fabric and PHY of the effects of PM cell generation.
0x23C, 0x23D, 0x284, 0x285	F4 and F5 PM Flow PTI and VCI Map	These registers should be left at their default values

The PM Configuration & Status Field in the PM RAM Table must be initialized the appropriate values. Table 13 below describes the purpose of each bit in the PM Table configuration field. Prior to use, all the fields in the PM Table should be initialized to 0.

Table 13 PM Table Configuration and Status Field

Bit	Function
Source_FwdPM	Bits 15:14 have the following mapping:

Bit	Function
Generate_BwdPM	00: Sink forward and backward PM cells 01: Source backward PM cells upon receipt of forward PM cells. 10: Source forward PM cells 11: Reserved
F4_F5B	Set to 1 if this session is for a VPC. Set to 0 otherwise.
ETE_SegB	Set to 1 if this is an end-to-end PM flow. Set to 0 if this is a segment PM flow.
Force_FwdPM	This bit allows this PM session to ignore the forward PM cell pacing register 0x23E & 0x289.
Threshold_Select	There are four sets of threshold registers (A1/A2 – D1/D2). The setting of these bits selects which of the four sets is applicable to this PM session.
Blocksize	This should be set to the desired PM block size, using the encoding provided in the datasheet. For instance, this would be set to 0100 for a block size of 2048 cells. Small block sizes are not recommended for active connections, as the PM cells could then amount to a significant portion of the available user bandwidth.
Reserved	This should be set to 0 initially. Thereafter, it should not be overwritten. This can be done using the write mask function when performing writes to the PM RAM
Bwd_PM_Pending	These bits should be initially set to a logic 1. Thereafter, they should not be overwritten. This can be done using the write mask function when performing writes to the PM RAM
Fwd_PM0	
Bwd_PM0	

5.2.2.1 Setting a PM Flow on an F4 Level to Monitor F5 Levels

A separate PM flow must be set up for an F4. This PM flow must be pointed to by the F4 OAM connection, but in addition it needs to be pointed to by all constituent F5s as well. This means that if the constituent F5s are also set up to have PM sessions, the F5 PM sessions are going to need to all be in one bank and the F4 PM session in another.

The following gives an illustration:

Assume an F4 OAM connection, VCRA d, with three constituent F5s, VCRA a, b, c.

Each F5 needs its own PM, and also a PM at the F4 level is needed (i.e., total four PM sessions.)

Set up the F5 PM sessions: bank 1, address 0,1, and 2. Each has F4_F5b = 0.

Set up the F4 PM session: bank 2, address 0. It has F4_F5b = 1.

VCRA d, the F4, has PM1 inactive. PM 2 active, PM 2 address 0.

VCRA a, an F5, has PM1 active, PM1 address 0, PM2 active, PM2 address 0.

VCRA b, an F5, has PM1 active, PM1 address 1, PM2 active, PM2 address 0.

VCRA c, an F5, has PM1 active, PM1 address 2, PM2 active, PM2 address 0.

5.2.3 System Management Cells

System management cells are not directly supported by the ATLAS. Rather, they may be dropped to the microprocessor cell interface and processed externally.

5.2.4 Activate and Deactivate Cells

Activate and Deactivate cells are not directly supported by the ATLAS. Rather, they may be dropped to the microprocessor cell interface and processed externally.

6 Modifying and Running the Example Code

The following files are packaged in atlas_software_a1.zip:

Table 14 List of Source Files

File Name	Description
devRegAccess.h	Header file for devRegAccess.c, as described in section 4.4.1. devRegAccess.c is not provided.
atlasRamAccess.h	Header file for atlasRamAccess.c, as described in section 4.4.2.
atlasRamAccess.c	Routines for accessing ingress and egress SRAM, as described in section 4.4.2.
sramDiag.h	Header file for sramDiag.c, as described in section 4.4.3.
sramDiag.c	Routines for performing diagnostics on ingress and egress SRAM, as described in section 4.4.3.
searchModule.h	Header file for searchModule.c, as described in section 4.4.4.
searchModule.c	Routines for adding and removing connections, as described in section 4.4.4.

6.1 Modifications Required

6.1.1 devRegAccess.h

The type definitions in this file allow the system independence of the software. The type definitions in this file should be modified to correspond to data types allowed in the system. In addition, error code constants should be modified as desired.

6.1.2 devRegAccess.c

Routines will need to be written to allow microprocessor access to the ATLAS. The routines should use the prototypes provided in devRegAccess.h.

6.1.3 atlasRamAccess.h

The following constants should be modified to match the particular target system:

- IVCRAM_NUM_BUSY_POLL This should be set to the ceiling of 1200ns / minimum uP read access time. For instance, if the shortest microprocessor read access time is 160ns, this should be set to 8.
- IVCRAM_ROWMASK This should be set to reflect the populated SRAM rows in the ingress VC Table Record. If all rows are populated, then this should be set to 0x7FFF.
- EVCRAM_NUM_BUSY_POLL Use the same value as IVCRAM_NUM_BUSY_POLL
- EVCRAM_ROWMASK This should be set to reflect the populated SRAM rows in the egress VC Table Record. If all rows are populated, then this should be set to 0xFFFF.

In addition, error code constants should be modified as desired.

6.1.4 sramDiag.h

The following constants should be modified to match the target system:

- IVC_MAX_VC This should reflect the depth of the ingress SRAM. For 64K VCs, this should be set to 0xFFFF.
- EVC_MAX_VC This should reflect the depth of the egress SRAM. For 64K VCs, this should be set to 0xFFFF.

In addition, error code constants should be modified as desired.

6.1.5 searchModule.h

The following constants should be modified to match the target system:

- IVC_NUM_PRIMARY The number of primary table locations. This is equal to the number of VCs. For 64K ingress VCs, this should be set to 65536.
- IVC_NUM_SECONDARY The number of secondary table locations. This is equal to the number of VCs. For 64K ingress VCs, this should be set to 65536.
- IVC_NUM_RECORDS The number of ingress VCs supported. This should reflect the depth of physical SRAM. For 64K VCs, this should be set to 65536.
- IVC_MAX_PRIMARY_ADDR The maximum primary table address. For 64K ingress VCs, this should be set to 0xFFFF
- IVC_MAX_SECONDARY_ADDR The maximum secondary table address. For 64K ingress VCs, this should be set to 0xFFFF
- IVC_MAX_RECORD_ADDR The maximum ingress VC table record address. For 64K VCs, this should be set to 0xFFFF
- EVC_NUM_RECORDS The maximum number of egress VCs supported. For 64K VCs, this should be set to 0xFFFF
- EVC_MAX_RECORD_ADDR The maximum egress VC table record address. For 64K VCs, this should be set to 0xFFFF

In addition, error code constants should be modified as desired.

6.1.6 searchModule.c

The following global variables should be configured to match the register configuration of the ATLAS:

- gIvcPrimaryTableUsed Set to 1 if the primary search table is used.
- gIvcFieldALength Set to L_A, the length of field A for the primary search.
- gIvcFieldAMask Mask out unused field A bits. For instance, if L_A=8, then this should be set to 0xFF.

- gIvcPhyIDMask Use this to mask out unused PHY ID bits in the primary key. If L_P=2, then this should be set to 0x03
- gIvcFieldBLength Set to L_B, the length of field B for the secondary search.
- gIvcFieldBMask Mask out unused field B bits.
- gEvcFieldALength The length of field A for the egress lookup.
- gEvcFieldAMask Mask out unused field A bits.
- gEvcFieldBLength The length of field B for the egress lookup.
- gEvcFieldBMask Mask out unused field B bits.
- gEvcPhyIDMask Mask out unused PhyID bits for egress lookup.

6.2 Using the Code

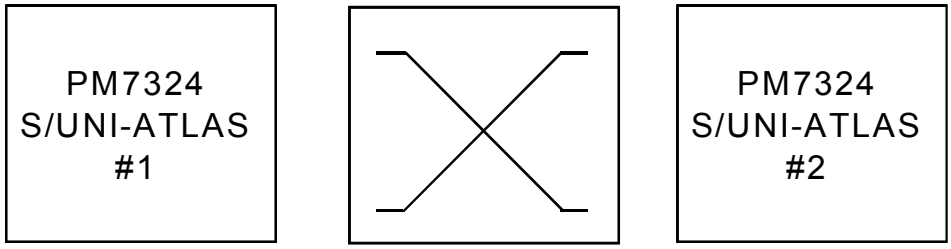
The software is organized in a hierachical manner, as illustrated in below in Figure 23. All higher layers may make calls to lower layers, but may not access higher layers.

Figure 23 Software Organization

<table>
<tr><td colspan="4">User Application</td></tr>
<tr><td colspan="2">SRAM Diagnostics (sramDiag.x)</td><td colspan="2">Table Maintenance (searchModule.x)</td></tr>
<tr><td>ivcSramDiag()</td><td>evcSramDiag()</td><td>ivcInitialize()
ivcReadVC()
ivcAddVC()
ivcRemoveVC()</td><td>evcInitialize()
evctableReadRecord()
evcAddVC()
evcRemoveVC()</td></tr>
<tr><td colspan="4">ATLAS SRAM Access (atlasRamAccess.x)</td></tr>
<tr><td colspan="2">ivcramRead()
ivcramWrite()</td><td colspan="2">evcramRead()
evcramWrite()</td></tr>
<tr><td colspan="4">ATLAS register Access (devRegAccess.x)</td></tr>
<tr><td colspan="2">devRead()
devWrite()
devWriteMask()</td><td colspan="2">devBitHiPoll()
devBitLowPoll()</td></tr>
</table>

In a typical application, there will be several instances of the ATLAS. There needs to be one instance of the table maintenance routines for each instance of the ATLAS, to be controlled by the User Application. For instance, consider the system in Figure 24.

Figure 24 System Using Two ATLASES



To provision a bidirectional connection through the system, ivcAddVC() and evcAddVC() will need to be used to provision the ingress and egress connections on each device. In order for backward OAM cell insertion to work correctly, the ingress and egress VC table record addresses must be related as outlined in the ATLAS datasheet. To accomadate this, the routine ivcAddVC() may, as an option, force the ingress VC table record to be located at a particular address. Since the egress cell identification is a direct lookup, the egress VC table address will be fixed by the cell contents. One way to relate the ingress and egress VC table addresses is to make them equal. Assuming the HEC and UDF bytes are used as the lookup address on each ATLAS, the following steps would be performed by the user application to provision the bidirectional system.

1. Use ivcAddVC() to provision an ingress connection at address X on ATLAS #1. Use header translation write Y to the HEC and UDF bytes on the outgoing cell.

2. Use evcAddVC() to provision an egress connection at address Y on ATLAS #2. Cells incoming will use address Y from the HEC and UDF bytes as the direct lookup.

3. Use ivcAddVC() to provision an ingress connection at address Y on ATLAS #2. Use header translation to write X to the HEC and UDF bytes on the outgoing cell.

4. Use evcAddVC() to provision an egress connection at address X on ATLAS #1. Cells incoming will use address X from the HEC and UDF bytes as the direct lookup.

7 Policing Configuration

7.1 Setting I and L Parameters

Example: Assume the following traffic descriptors are required for setting I and L GCRA parameters of a VC table:

PCR = 353207 cells/s, SCR = 235849 cells/s, MBS = 210 cells, CDVT = 35.390 us, Δt = 20 ns

GCRA1 (PCR):

I1 = 1/(PCR * Δt) = 1/(353207*20e-9) = 142

Since the most significant 5 bits are exponent (e) and least significant 8 bits are mantissa (m), it needs to be mapped to the floating point fields.

e1 = int[ln(I1)/ln(2)] = int[ln(142)/ln(2)] = 7 = 00111 B

m1 = [(I1/2^e)-1]*512 = [(142/2^7)-1]*512 = 56 = 000111000 B

L1 = CDVT/(Δt) = 35.390e-9/20e-9 = 1770

e1 = int[ln(L1)/ln(2)] = int[ln(1770)/ln(2)] = 10 = 01010 B

m1 = [(L1/2^e)-1]*512 = [(1770/2^10)-1]*512 = 373 = 101110101 B

GCRA 2 (SCR):

I2 = 1/(SCR * Δt) = 1/(235849*20e-9) = 212

Since the most significant 5 bits are exponent (e) and least significant 8 bits are mantissa (m), it needs to be mapped to the floating point fields.

e2 = int[ln(I2)/ln(2)] = int[ln(212)/ln(2)] = 7 = 00111 B

m2 = [(I2/2^e)-1]*512 = [(212/2^7)-1]*512 = 336 = 101010000 B

L2 = [(MBS - 1)(1/SCR - 1/PCR)]/(Δt) = [(210 - 1)(1/235849 - 1/353207)]/(20e-9) = 14722

e2 = int[ln(L2)/ln(2)] = int[ln(14722)/ln(2)] = 13 = 01101 B

m2 = [(L2/2^e)-1]*512 = [(14722/2^13)-1]*512 = 408 = 110011000 B

In summary:

I1 = 00111000111000 Binary = E38 Hex

L1 = 01010101110101 Binary = 1575 Hex

I2 = 00111101010000 Binary = F50 Hex

L2 = 1101110011000 Binary = 1B98 Hex

Notes