Order this document
by AN1296/D

AN1296

Using the MCM69C232/MCM69C432 Content–Addressable Memory on an ATM Line Card

Prepared by: Mike Parks

Asynchronous Transfer Mode (ATM) switches, due to their connection–based protocol, must translate each cell's address at every point along the routing path. The cell's address information is carried in two fields within the five byte header. The Virtual Path Identifier (VPI) is an 8 or 12 bit value, depending on the protocol being run, while the Virtual Circuit Identifier (VCI) is a 16 bit value. Figure 1 shows a small network of switches (S1 – S5) connecting six computers (C1 – C6).

A connection has been established between C1 and C6, through switches S1, S3, and S5. Conceptually, the connection consists of a list of VPI/VCI translation pairs. For example, C1 sends cells to S1 using a VPI of $7AC_{16}$ (hexadecimal notation) and a VCI of $5AB0_{16}$. When S1 receives a cell on port 1 with those values, it changes the VPI to 902_{16} and the VCI to $A77F_{16}$. The cell is then transmitted to S3 by outputting it on port 3. Similarly, when S3 received the cell, it translates the VPI/VCI and forwards the cell to S5. Switch S5 performs a final VPI/VCI translation and forwards the cell to its end point, C6. This type of connection, in which both the VPI and the VCI are examined and (usually) modified, is known as a Virtual Circuit Connection (VCC). A second connection type, the Virtual Path Connection (VPC) only requires that the VPI be translated. The VPC is used when many connections follow the same path through a sequence of switches.

The speed at which VPI/VCI pairs must be translated is a function of several variables, including the line speed, the number of lines connected to a single line card, and the speed of the other circuitry on the line card. ATM switch designers typically require that a VPI/VCI translation is completed within one quarter to one half the time it takes to receive a cell. For OC12 rates (622 megabits/second) this corresponds to 160 to 320 ns. Note that the translation must be completed in this interval whether the cell belongs to a VPC or a VCC.

Switch designers use one of several approaches to solve the VPI/VCI translation problem. The simplest approach simply restricts the address space. While the 28 bits of the VPI/VCI define a space of 26.8 billion possible values, typically only a few thousand are active in a switch at a time. By assigning an invariant value to most of the VPI/VCI bits and only considering 12 – 14 bits when creating new connections, the switch can maintain a table in memory of the outbound VPI, VCI, and port values that correspond to an incoming cell's address. See Figure 2 for an example. This approach is somewhat risky due to its inflexibility and various regional standards that are still in a state of flux.

Other designers use a hashing technique to translate the VPI/VCI into an appropriately sized table offset. For example, if a switch needs to maintain 4096 connections, i.e., 2^{12}, a logic network can be designed with 28 inputs and 12 outputs. The logic will reduce the VPI/VCI value so that it addresses a table of manageable size. However, the output of the logic network can be duplicated for more than one active connection. Thus, the output values of VPI, VCI, and port number must be maintained in a data structure such as a linked list that allows them to be accessed sequentially until the correct one is found. Figure 3 shows an example of this approach.

The maximum time required to translate a VPI/VCI pair increases with the length of the linked list. In a worst case scenario, the allowable time can be exceeded, unless the switch's processor devotes resources to modification of the hashing function in response to changing active VPI/VCI values.

In yet another approach, the connections can be kept in a table that is sorted by VPI/VCI value. Then, when a cell arrives, a binary search can be performed to find a matching value of VPI/VCI in the table. This approach has the advantage of a deterministic upper bound on search time. For example, a 4K table can be searched in 12 steps. However, the stringent translation time specifications necessitate a high performance hardware implementation.

The best approach uses a Content–Addressable Memory (CAM). The CAM turns the normal memory access "inside out". A *datum* is applied to the CAM, and it outputs the *address* where a matching value is stored. Therefore, if 4K connections need to be active in a switch, their VPI/VCI values can be stored in a 4K x 28 CAM. The 12 bit address output can be used as an index into a RAM table where the translated VPI/VCIs are stored. While the CAM–based solution is simple, deterministic, and unrestricted, it has been relatively expensive. In mid–1995, a CAM with 1024 entries sold for approximately $40.

Motorola's introduction of the 4K MCM69C232 and the 16K MCM69C432 CAMs has changed the balance of the competing VPI/VCI translation methods. Due to the use of a standard 4–transistor bit cell, the Freescale products are both higher capacity and lower cost than past solutions. Like competing CAMs, they are very well suited to this application.

© Freescale Semiconductor, Inc., 2004. All rights reserved.

REV 1

freescale semiconductor

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

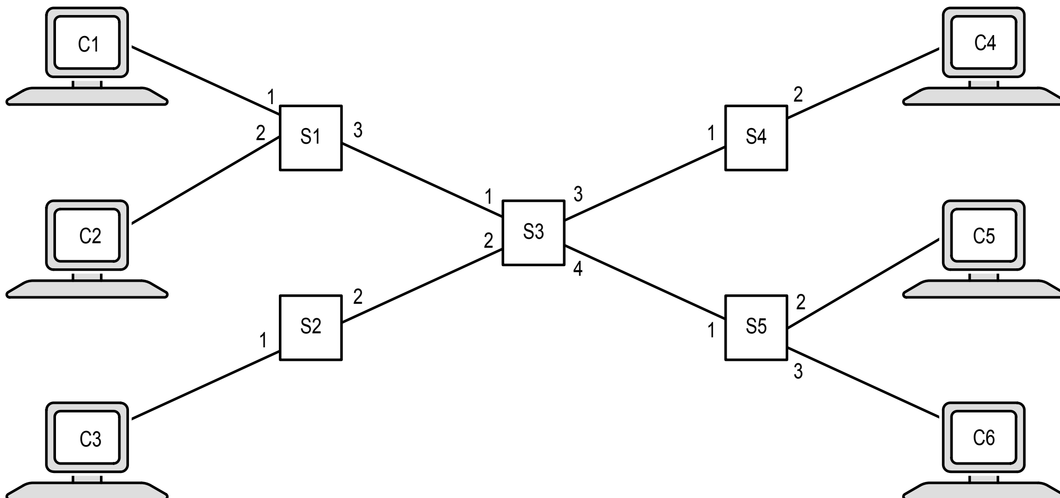

Figure 1.

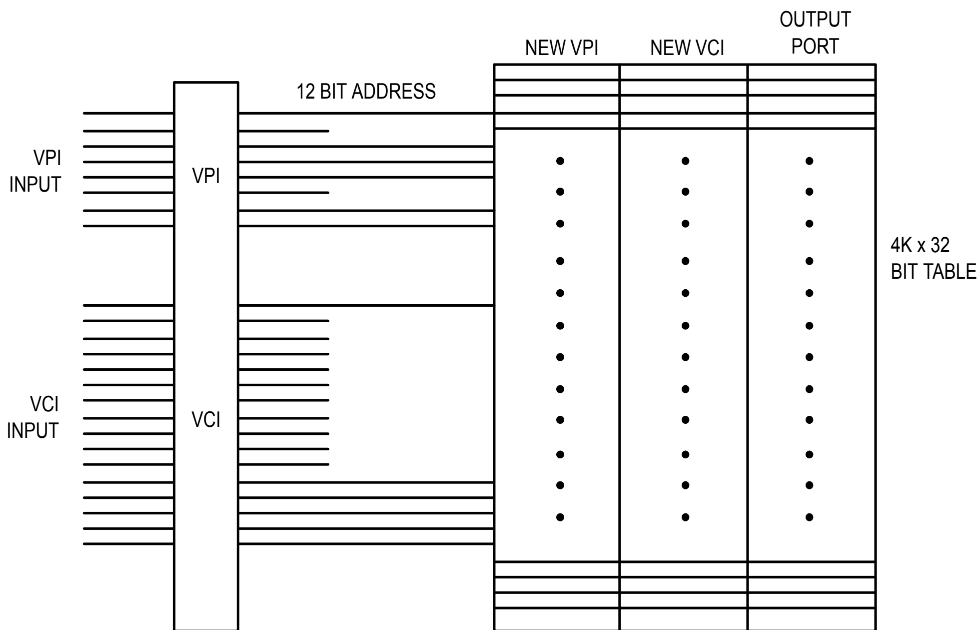

Figure 2. “Brute Force” VPI/VCI Compression

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

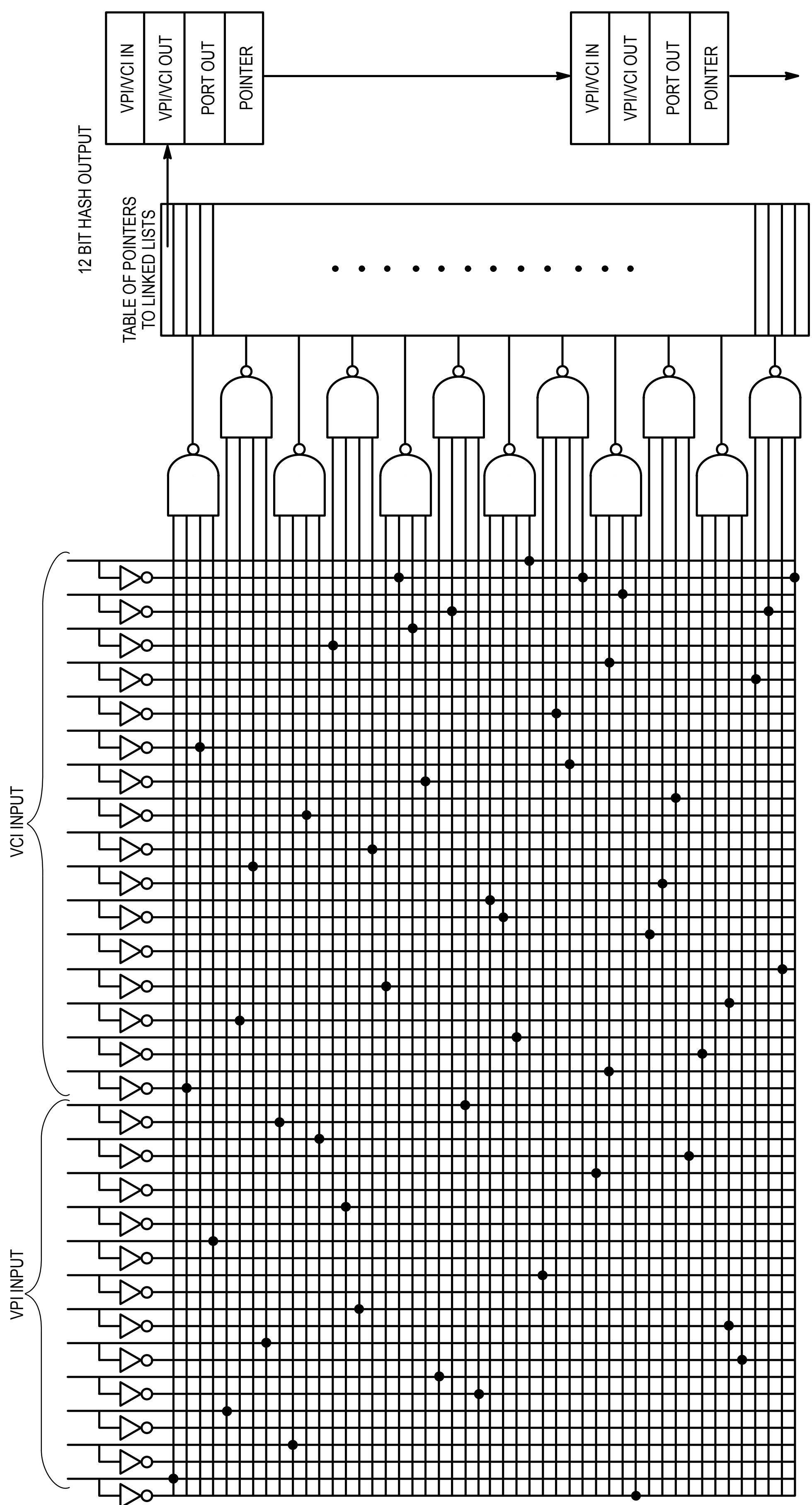

Figure 3.

Freescale Semiconductor, Inc.

Freescale Semiconductor, Inc.

GENERIC LINE CARD ARCHITECTURE

In ATM applications, a line card consists of a physical layer interface, a framer function, a cell processor that includes some method of VPI/VCI translation, an interface to the switch fabric, and a controller. Typically, the card also includes a context memory, accessed by the controller and the cell processor, that holds information about each connection.

The physical layer device provides clock recovery and data synchronization functions, and passes a serial data stream on to the framer. The framer chip finds packet and cell boundaries, and passes cells on to the cell processor, typically on 1– or 2–byte–wide busses. VPI/VCI translation is performed by the cell processor and the CAM.

In this architecture, the controller is connected to the CAM's control port and is used to initialize the part and to perform table maintenance as connections are added and dropped. The cell processor is connected to the CAM's match port. Match port operations are the highest priority, ensuring that VPI/VCI translation can be done in a deterministic fashion.

INITIALIZING THE CAM FOR VPI/VCI TRANSLATION

Initialization of the MCM69C232 and MCM69C432 is performed via the control port. In the general case, 16 bit data values are written to each of the 4 I/O registers. An instruction word is then written to the operation register that describes the action to be performed. Some of the instructions do not use one or more of the I/O registers. The following sections of the paper describe the process of initializing the CAM. Appendix A contains a C–language implementation of the initialization sequence.

For VPI/VCI translation applications, the first step is to enter ATM mode. This instruction does not require any data values. Therefore, it is executed by writing the "Enter ATM Mode" op code (0008_{16}) to the operation register at address offset 4 (the values of A2 – A0 should be 100_2 to write to the operation register).

The mask register can then be loaded by writing 0s to the bits that should be included in a match operation. The 64 bit value is constructed by concatenating the values in I/O registers 3 – 0. Bit 15 of I/O register 3 is the most significant bit, and bit 0 of I/O register 0 is the least significant bit. For ATM applications, the lower 32 bits are normally not included in the match operation. The VPI bits must correspond to bits 59 – 48 of each entry, while the VCI corresponds to bits 47 – 32. (If a User–Network Interface protocol is being run, the VPI bits are 55 – 48.) Assuming that the four uppermost bits are not needed for some reason proprietary to the switch designer, the values written to I/O registers are as follows: I/O register 3 = $F000_{16}$ ($FF00_{16}$ for UNI), I/O register 2 = 0000_{16}, I/O register 1 = $FFFF_{16}$, and I/O register 0 = $FFFF_{16}$. The mask register is then loaded with these values by writing the "Set Mask Register" op code (0002_{16}) to the operation register.

The next step in the initialization sequence is to execute the "Enter Fast Entry Mode" instruction by writing the op code (0004_{16}) to the operation register. This instruction does not require any data values. The fast entry mode is preferred when large numbers of entries are to be added to the table.

If desired, the almost–full point can be set by writing the desired value to I/O register 0, then writing the "Set Almost–Full Point" op code (0007_{16}) to the operation register. Note that only the lower 12 bits are used for the MCM69C232, and the lower 14 bits for the MCM69C432. As an example, if an interrupt is desired when the '232 becomes half full, the value written to I/O register 0 should be 0800_{16}.

At this point, the initial connections can be loaded into the CAM. The most significant 16 bits are written to I/O register 3. Similarly, the three other I/O registers are loaded with the remaining bits of the CAM entry. Note that the VPI and VCI bits must be aligned as explained in the "Set Mask Register" section. The VPI is loaded into the lower 12 bits of I/O register 3 (the lower 8 bits for UNI), while the VCI is loaded into I/O register 2. A VPC is established by using a value of $FFFF_{16}$ for the VCI. This means that $FFFF_{16}$ cannot be used as the VCI of a VCC. I/O registers 1 and 0 are loaded with the 32 bit value that should be output on the MQ bus when this entry is matched. Bit 15 of I/O register 1 corresponds to MQ31, while bit 0 of I/O register 0 corresponds to MQ0. After the four I/O register values are written, the "Insert Value" op code (0000_{16}) is written to the operation register. This process is repeated until all of the initial entries have been loaded.

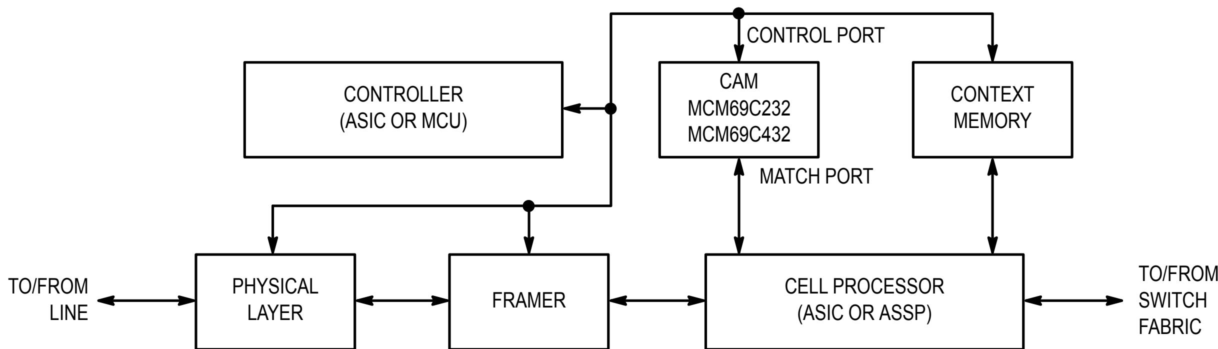

Figure 4. ATM Line Card

Next, the CAM table must be initialized. This instruction, which does not require any I/O register values, sorts the table in preparation for matching. It is executed by writing the "Initialize Table" op code ($000B_{16}$) to the operation register. Execution of this instruction automatically switches the CAM to its buffered entry mode, which allows matching to continue while new values are being added. The CAM also switches to buffered entry mode when the next to last entry is added to the table.

At this point, the CAM is initialized, and match port operations can begin. To initiate a match, the cell processor places the VPI from an incoming cell on MQ27 – MQ16 (MQ23 – MQ16 for UNI) and the VCI on MQ15 – MQ0. The match is started by asserting the LH/SM signal. The completion of this operation is signaled by the assertion of the MC (Match Complete) pin. If the match is successful, MS is asserted, while the VPC signal is asserted if the match is a virtual path circuit.

While the cell processor uses the match port to translate the incoming VPI/VCI values, the controller can continue to use the control port for table maintenance and administrative functions. The C code in Appendix A continues to loop through a sequence of adding any new entries that are needed, deleting any stale entries, and periodically reading the entire CAM table to verify that all the desired connections are still active.

While VPCs and VCCs are typically segregated into different ranges of VPI values; occasionally, it may be necessary to expand the range of values used for VPCs. If this occurs, all VCCs that use that VPI must first be deleted. They can be located by using "Fast Read" instructions, then deleted one by one. Note that they will typically be located in a contiguous group in the CAM. However, the last entry should always be checked because it will hold the last value added to a completely full CAM, no matter what its match value might be.

Freescale Semiconductor, Inc.

```
/****************************************************************************/
/* initialization of the mcm69c232 content addressable memory              */
/* for vpi/vci translation in an atm application                            */
/* August 1996                                                              */
/* Author: Mike Parks                                                       */
/****************************************************************************/
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>

#define  CAMSIZE 4096

/* define flag register's bits */
#define INT_ENABLE                   1
#define LAST_MATCH_SUCCESSFUL        2
#define TABLE_INITIALIZED            4
#define BUFFERED_ENTRY_MODE          8
#define ENTRY_QUEUE_EMPTY            16
#define ENTRY_QUEUE_FULL             32
#define CAM_TABLE_FULL               64
#define ERROR_CONDITION_SET          128
#define TABLE_ALMOST_FULL            256
#define ATM_MODE                     512
#define LAST_OP_COMPLETE             1024

/* define op code values */
#define INSERT_VALUE                 0x0
#define DELETE_VALUE                 0x1
#define CHECK_FOR_VALUE              0x6
#define INITIALIZE_TABLE             0xb
#define ENTER_FAST_ENTRY_MODE        0x4
#define ENTER_BUFFERED_ENTRY_MODE    0x5
#define SET_ATM_MODE                 0x8
#define RETURN_ENTRY_COUNT           0x3
#define SET_GLOBAL_MASK_REGISTER     0x2
#define SET_ALMOST_FULL_POINT        0x7
#define SET_FAST_READ_REGISTER       0x9
#define FAST_READ                    0xa

/* define registers as contents of pointer variables */
#define IO_REG0    *io_reg0_ptr
#define IO_REG1    *io_reg1_ptr
#define IO_REG2    *io_reg2_ptr
#define IO_REG3    *io_reg3_ptr
#define OP_REG     *op_reg_ptr
#define FLAG_REG   *flag_reg_ptr
#define ERR_REG    *err_reg_ptr
#define INT_REG    *int_reg_ptr

/* function template declarations */
void output_connections();
void verify_connections();
void reclaim_output_values(int);
void next_output();

/* global variable declarations */
unsigned int *io_reg0_ptr;     /* pointer to I/O register 0 */
unsigned int *io_reg1_ptr;     /* pointer to I/O register 1 */
unsigned int *io_reg2_ptr;     /* pointer to I/O register 2 */
unsigned int *io_reg3_ptr;     /* pointer to I/O register 3 */
unsigned int *op_reg_ptr;      /* pointer to operation register */
unsigned int *flag_reg_ptr;    /* pointer to flag register */
unsigned int *err_reg_ptr;     /* pointer to error register */
unsigned int *int_reg_ptr;     /* pointer to interrupt register */
```


For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

```
int vpi[CAMSIZE];              /* array to hold VPIs during verification */
int vci[CAMSIZE];              /* array to hold VCIs during verification */
int out_msw[CAMSIZE];          /* array to hold upper 16 bits of output words */
int out_lsw[CAMSIZE];          /* array to hold lower 16 bits of output words */
int output_values[CAMSIZE/16]; /* array to designate available output values */
int insert_value_time = 0;     /* set by switch when a new connection is needed */
int delete_value_time = 0;     /* set by switch when time to drop a connection */
int check_conn_time = 0;       /* set by switch to trigger connection check */
int new_vpi = 0;               /* switch provides new connection's VPI here */
int new_vci = 0;               /* switch provides new connection's VCI here */
int del_vpi = 0;               /* switch provides VPI to delete here */
int del_vci = 0;               /* switch provides VCI to delete here */
FILE *f_conn;                  /* pointer to connections file */
int conn_count;                /* number of active connections in CAM */

void main()
{
  int a, b;

 // assume CAM is decoded at address F000 hex
 io_reg0_ptr  = (unsigned int *)0xF000; /* address offset 0 */
 io_reg1_ptr  = io_reg0_ptr + 1;        /* address offset 1 */
 io_reg2_ptr  = io_reg0_ptr + 2;        /* address offset 2 */
 io_reg3_ptr  = io_reg0_ptr + 3;        /* address offset 3 */
 op_reg_ptr   = io_reg0_ptr + 4;        /* address offset 4 */
 flag_reg_ptr = io_reg0_ptr + 5;        /* address offset 5 */
 err_reg_ptr  = io_reg0_ptr + 6;        /* address offset 6 */
 int_reg_ptr  = io_reg0_ptr + 7;        /* address offset 7 */

 // Initialize output_values[] array to indicate all output values are available
  for(a = 0; a < CAMSIZE/16; a++)
      output_values[a] = 0xFFFF;        /* set all the bits */

 // For VPI/VCI translation, the ATM mode must be used to match virtual
 // path connections and virtual circuit connections simultaneously.
 OP_REG = SET_ATM_MODE;        /* writes 0x0008 to operation register */
 while((FLAG_REG & LAST_OP_COMPLETE) == 0)
      ;    /* null statement - loop until last operation is completed */

 // Now set the mask register for VPI/VCI translation. Assuming the upper
 // four bits are not used for any proprietary purpose, the mask register
 // should be set to F0000000FFFFFFFF for Network-Network Interface protocols.
 IO_REG3 = 0xF000; /* 0xFF00 for UNI */
 IO_REG2 = 0x0000;
 IO_REG1 = 0xFFFF;
 IO_REG0 = 0xFFFF;
 OP_REG = SET_GLOBAL_MASK_REGISTER;   /* write opcode to operation register */
 while((FLAG_REG & LAST_OP_COMPLETE) == 0)
      ;    /* null statement - loop until last operation is completed */

 // Set the CAM to fast entry mode, which is the preferred method of
 // entering large numbers of new entries.
 OP_REG = ENTER_FAST_ENTRY_MODE;
 while((FLAG_REG & LAST_OP_COMPLETE) == 0)
      ;    /* null statement - loop until last operation is completed */

 // Setting the almost-full point is optional.  This example sets a
 // value of 2048, which represents the half-full point of the MCM69C232.
 IO_REG0 = 0x0800;    /* hex equivalent of 2048 */
 OP_REG = SET_ALMOST_FULL_POINT;
 while((FLAG_REG & LAST_OP_COMPLETE) == 0)
      ;    /* null statement - loop until last operation is completed */

 // Now the initial set of connections can be loaded into the CAM.  Let's
 // assume that the first two connections are the same every time the switch
 // is rebooted, so we can examine how a VPC and a VCC are established. The
 // rest of the connections will be loaded from a file called "connect.dat"
```

Freescale Semiconductor, Inc.

```
// When the CAM is in ATM mode, establishing a connection with a VCI of
// 0xFFFF defines a VPC.  Any input value with the corresponding VPI will
// result in a match, regardless of the VCI input value. When inserting
// a value, the VPI is loaded (right-justified) into I/O register 3, the
// VCI into I/O register 2, and the 32 bit value to be output on a match
// is loaded into I/O registers 1 & 0.
IO_REG3 = 0x07AC;              /* lower 12 bits are VPI (lower 8 bits for UNI) */
IO_REG2 = 0xFFFF;              /* VCI of FFFF defines this as a VPC */
                               /* Any cell with VPI of 7AC will match this entry */
next_output();                 /* Sets I/O registers 1 & 0 with output value */
OP_REG = INSERT_VALUE;         /* execute insert-value instruction */
while((FLAG_REG & LAST_OP_COMPLETE) == 0)
     ;    /* null statement - loop until last operation is completed */

// now load the constant VCC */
IO_REG3 = 0x0EB7;              /* lower 12 bits are VPI (lower 8 bits for UNI) */
IO_REG2 = 0x4000;              /* VCI not FFFF defines this as a VCC */
                               /* Both VPI and VCI of incoming cell must match */
next_output();                 /* Sets I/O registers 1 & 0 with output value */
OP_REG = INSERT_VALUE;         /* execute insert-value instruction */
while((FLAG_REG & LAST_OP_COMPLETE) == 0)
     ;    /* null statement - loop until last operation is completed */

// Open the file that holds the rest of the connection information.  Each
// entry consists of two integers, corresponding to the VPI and VCI values
// to be written to I/O registers 3 & 2.  The 32 bit output value is returned
// in I/O registers 1 & 0 by a call to the function "next_output".
f_conn = fopen("connect.dat", "r");
if(f_conn == NULL)
     {printf("Error opening connections file. \n");
      exit(-1);}

// Now loop until the end of file is reached, loading values from the file
// into the I/O registers, generating the output value, then executing the
// insert-value instruction. To prevent overflow, include a counter variable
// in the for statement.
for(a = 2; a < CAMSIZE; a++)
        {
         b = fscanf(f_conn, "%04x%04x", io_reg3_ptr, io_reg2_ptr);
         if(b != EOF)
              {
               next_output();
               OP_REG = INSERT_VALUE;
               while((FLAG_REG & LAST_OP_COMPLETE) == 0)
                      ;    /* null statement - loop until last operation is completed */
              }
        }

// Now that the initial connection values are loaded, the table must be
// sorted to prepare the CAM for match operations. The initialize-table
// instruction requires as much as 12 milliseconds to complete.  If desired,
// an interrupt can be enabled to occur upon completion so the controller
// can do other things.  In this example, we will wait for completion.
OP_REG = INITIALIZE_TABLE;
while((FLAG_REG & LAST_OP_COMPLETE) == 0)
      ;    /* null statement - loop until last operation is completed */

// At this point, the CAM is initialized and match port operations can
// begin.  Execution of the initialize-table instruction automatically
// switched us to buffered entry mode, so we can continue to add new values
// as needed without running initialize-table again.  We now will loop
// continuously, checking for a new connection to add, an old one to delete,
// and periodically reading the table to verify the existing connections.
while(1)  /* continue in this loop until hardware reset */
 {
      /* insert a new value if switch has set flag and CAM isn't full */
```


For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

```
        if(insert_value_time && ((FLAG_REG & CAM_TABLE_FULL) == 0))
         {
          while(FLAG_REG & ENTRY_QUEUE_FULL)
                      ; /* wait until queue isn't full */
          IO_REG3 = new_vpi;  /* switch provides new VPI in this variable */
          IO_REG2 = new_vci;  /* and new VCI in this one */
          next_output();      /* call function that generates output value */
          /* Output values of dropped connections are only reclaimed by the
             check_connections process.  Call these functions if next_output
             returned a value of 0xFFFF in I/O register 0. */
          if(IO_REG0 == 0xFFFF)
               {
                output_connections(); /* copy connection data from CAM to arrays */
                reclaim_output_values(conn_count); /* update connection bit array */
                next_output();     /* guaranteed to work this time, since number of
                                   available output values = number of CAM entries */
               }
          OP_REG = INSERT_VALUE; /* execute insert-value now that inputs are ready */
          while((FLAG_REG & LAST_OP_COMPLETE) == 0)
                      ;    /* null statement - loop until last operation is completed */
         }

        // Next delete any connections that are no longer needed.  The output
        // values could be reclaimed as the connection is dismantled, but we
        // will wait to do that during the process of verifying connections.
        if(delete_value_time) /*flag set by switch to delete a connection */
               {
                while(FLAG_REG & ENTRY_QUEUE_FULL)
                      ; /* loop until queue is not full */
                IO_REG3 = del_vpi;     /* switch provides VPI of connection to kill */
                IO_REG2 = del_vci;     /* and the VCI  */
                OP_REG = DELETE_VALUE; /* execute the delete operation */
                while((FLAG_REG & LAST_OP_COMPLETE) == 0)
                      ; /* null statement - loop until last operation is completed */
               }

        // The switch sets the flag check_conn_time when the table of active
        // connections needs to be verified. The array of bits that tracks
        // available output values is also updated at this time, reclaiming
        // any values that belonged to now-deleted connections.
        if(check_conn_time)
         {
          output_connections();  /* copy values from CAM to data arrays */
          verify_connections();  /* compare data in arrays to expected values */
          reclaim_output_values(conn_count); /* reset bits that represent available
                                         output values */
         }
  }
 }

void output_connections()
 {
  int b;
  while((FLAG_REG & ENTRY_QUEUE_EMPTY) != 0)
             ; /* loop until queue is empty, a condition for fast-read instruction */
  OP_REG = RETURN_ENTRY_COUNT;
  while((FLAG_REG & LAST_OP_COMPLETE) == 0)
             ;    /* null statement - loop until last operation is completed */
  conn_count = IO_REG0; /* save # of connections before overwriting I/O reg 0 */
  /* set the fast-read register to 0 */
  IO_REG0 = 0x0000;
  OP_REG = SET_FAST_READ_REGISTER;
  while((FLAG_REG & LAST_OP_COMPLETE) == 0)
             ;    /* null statement - loop until last operation is completed */
```

Freescale Semiconductor, Inc.

```
  // Now perform fast-read operations from location 0 thru (conn_count - 1),
  // saving the vpi, vci, and output data into arrays for verification.
  for(b = 0; b < conn_count; b++)
       {
        OP_REG = FAST_READ;
        while((FLAG_REG & LAST_OP_COMPLETE) == 0)
                     ;    /* null statement - loop until last operation is completed */
        vpi[b] = IO_REG3;
        vci[b] = IO_REG2;
        out_msw[b] = IO_REG1;
        out_lsw[b] = IO_REG0;
       }
  }

// This function compares the values read from the CAM into arrays vpi[], vci[],
// out_msw[], and out_lsw[] to the values of the active connections. The imple-
// mentation will be dependent on the method the switch uses to hold those
// expected values.
void verify_connections()
 {
 }

// This function manages an array of bits equal in size to the number of bits
// in the CAM.  It assumes 16 bits per integer.  A bit value of 1 means the
// corresponding output value is available, while a 0 means it is in use. Bit 0
// of word 0 corresponds to an output value of 0x0000, bit 1 of word 0 -->
// 0x0001, bit 14 of word 0 --> 0x000E, bit 0 of word 1 --> 0x0010, etc.
void next_output()
 {
  int a, b;
  for(a = 0; a < CAMSIZE/16 && (output_values[a] == 0); a++)
              ;  /* loop until a non-zero value is found */
  if(a == CAMSIZE/16)
       IO_REG0 = 0xFFFF;   /* indicates no output values are available */
  else
       {
       /* find the first non-zero bit within word a */
        for(b = 0; b < 16 && ((output_values[a] & (2^b)) == 0); b++)
                     ;  /* loop until a bit is found that is a 1,
                           corresponding to an available value */
        IO_REG0 = a*16 + b; /* put the output value in I/O register 0 */
        IO_REG1 = 0x0000;   /* output ranges from 0-CAMSIZE, so upper 16 bits = 0 */
        output_values[a] = output_values[a] & ~(2^b); /* clear bit we assigned */
       }
 }

// This function manipulates the array output_values[].  Each bit represents a
// possible output value. A value of 0 means the value is in use, while 1 means
// the value is available.  First we set all bits, then clear bits that correspond
// to output values that are being used by an active connection.
void reclaim_output_values(int count)
 {
  int a, b, c, d;
  // Set all the bits to signify all output values are available. Those that
  // are in use will be cleared one by one.
  for(a = 0; a < CAMSIZE/16; a++)
       output_values[a] = 0xFFFF;  /* set all the bits */
```

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

```
 for(a = 0; a < count; a++)
      {
       b = out_lsw[a];           /* integer value of an output */
       c = b/16;                 /* calculates which 16 bit word of output_values[]
                                             corresponds to the value of b */
       d = b % 16;               /* calculates which bit in that word
                                             corresponds to the value of b */
       /* now clear bit d of word c */
       output_values[c] = output_values[c] & ~(2^d);
      }
}
```

Freescale Semiconductor, Inc.

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.
Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters which may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals” must be validated for each customer application by customer’s technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

freescale™ semiconductor

Freescale Semiconductor, Inc.