UTMC Commercial Product Line

eCard Distributed Query Processor Technical Specification

Providing Ultra-high performance
for
lookup intensive applications

Draft

Revision 1.1

26-Dec-00

TABLE OF CONTENTS

eCard Distributed Query Processor Technical Specification

Ultra-high performance for lookup intensive applications

Product Overview

The eCard is a PCI card designed to accelerate any computing task that must organize and retrieve information. In many systems, this activity represents a significant performance bottleneck.

Target Applications

Systems requiring extremely fast queries or handling a very high volume of query dependant traffic are candidates for a high performance distributed query processor. While higher speed CPUs can satisfy the demand in some applications, there is a growing set of applications which can not satisfy their need for performance with a general purpose processor. These applications generally fall into three categories:

1. **High speed transaction oriented processes.** For transaction oriented applications, performance demand is driven by the number of simultaneous or near-simultaneous transactions. A transaction might involve a query in a search engine, the recording of billing information, or an update of inventory records. With the growth in the web and the number of people and systems interconnected with one another, astounding transaction rates are being experienced.

2. **Executive information systems.** These systems are generally built around a large data warehouse. A high level query in such a system can result in millions of elemental database queries. The time required to resolve many such queries, when an expert wants to explore the data and perform what-if scenarios, has lead to the ruin of many of these systems. Yet, the ability to extract information from massive amounts of data is increasingly becoming one of the more significant strategic advantages in business.

3. **Artificial intelligence applications.** Artificial Intelligence (AI) applications have long suffered from an inability of computers to quickly do what comes naturally to humans. The ability to quickly perform associative memory operations and to find close matching patterns is key to the success of this promising field.

In many cases, performance objectives are not being met with current technology. In other cases, performance objectives are being satisfied at considerable cost. The eCard is designed to eliminate the memory bottleneck and provide a low cost path to ultra-high performance.

Distributed Architectures

Performance can be further enhanced by employing multiple eCards to further distribute the processing load. This is shown in figure 1.

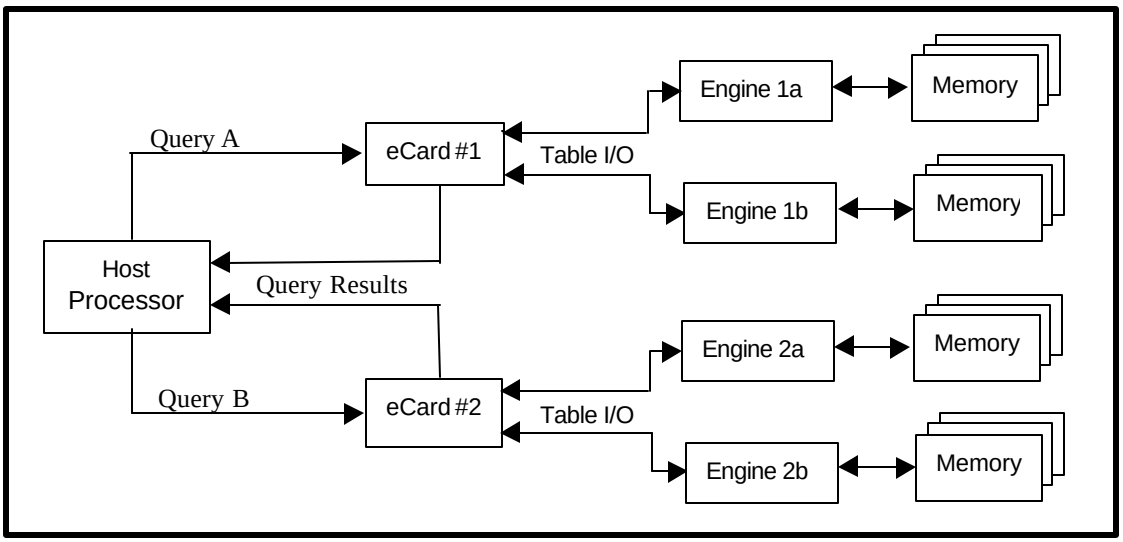


Figure 1 - Distributed Processing Environment

The distributed query processor approach frees up CPU cycles on the host processor while a query is being resolved. Typically these cycles can be used to perform needed high level application tasks which otherwise would have to wait for the CPU to complete its query processing activity.

The eCard advantages over traditional processing approaches include:

1. **The Pipelined Parallel Processing Advantage.** In a typical PC, multiple process threads must be performed serially. In a multiple processor architecture, multiple threads are handled by different CPUs with arbitration for shared resources. With an eCard, however, a single process thread can be expedited using the eCard's built in FIFOs. True parallel processing can be performed on the query processor's MIPS CPU and the two UTCAM-Engine chips.

2. **The Memory Capacity Advantage.** The eCard dramatically extends the memory capacity of the host system by supporting up to 2 gigabytes of memory on each card. Future generations will support up to 32 gigabytes of memory on each card, providing a path for system extendibility.

3. **The Cost / Performance Advantage.** The eCard is specifically designed for associative memory performance. Each of an eCard's two engines is capable of performing elemental database work at rates of about 5 times that of a state of the art processor. When the alternative is a multiple processor system and/or expensive software licenses, the use of an eCard can result in considerable cost savings.

eCard Features

The eCard distributed query processor, shown in figure 2 below, has the following capabilities:

- Supports up to 2GB of content addressable memory (based on present DIMM technology)
  - Uses normal PC100 SDRAM DIMMs
  - Configurable with as little as 16MB of memory
- Supports up to 128MB of system memory
- Contains two UTCAM-Engine chips on a 64-bit, 100 MHz data bus
- Fully programmable query operation via an on-board 260 MHz MIPS processor
- Query control code is downloaded from the host upon power-up
- Plug and play PCI interfaces with applications via simple device drivers
- Supports any combination of 32 or 64 bit and 33 or 66 MHz PCI
- Performs up to 10 million elemental database seeks per second
- Capable of comparing up to 200 Million patterns per second
- RS-232 serial port for debug monitor access

The figure below shows a block diagram of the eCard:

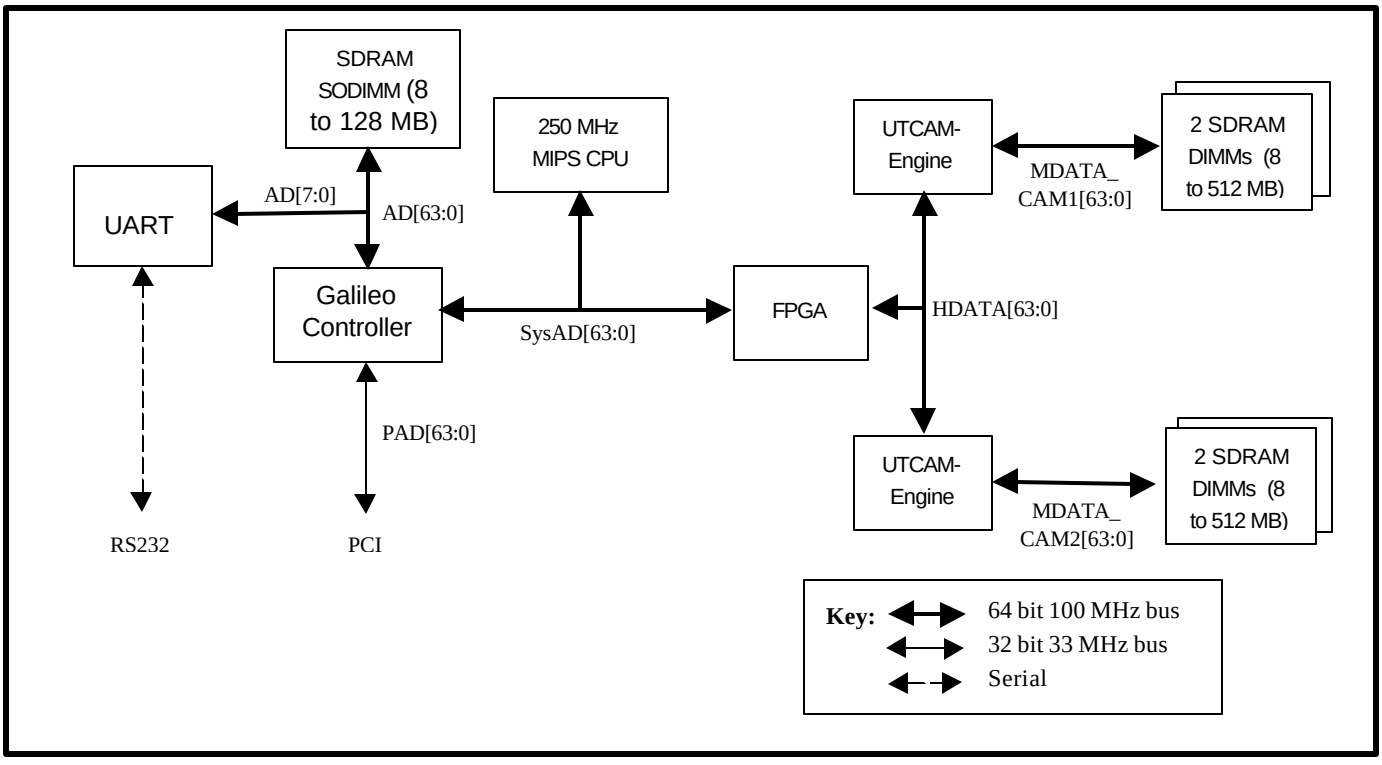


Figure 2 - eCard Block Diagram

The workhorses of the eCard are two state of the art Content Addressable Memory Engines. These chips, custom engineered by UTMC Microelectronic Systems, operate at 100 MHz and have the following capabilities:

CAM Engine Features

- Performs lookups in as little as 100 nanoseconds
- Partitions memory into as many as 8,192 tables sized from 256 to 30 million records
- Key widths, for a given table, can be programmed from 1 to 32 bytes in width
- Association widths can be up to 8 megabytes
- Performs exact matches, hierarchical matches, prefix matches and proximity matches
- Supports pipelined operation with separate I/O FIFOs
- Performs bulk table load, unload, and count functions
- Handles table overflows

eCard Layout

The figure below shows the layout of the eCard. The identified components are described in later sections of this document.

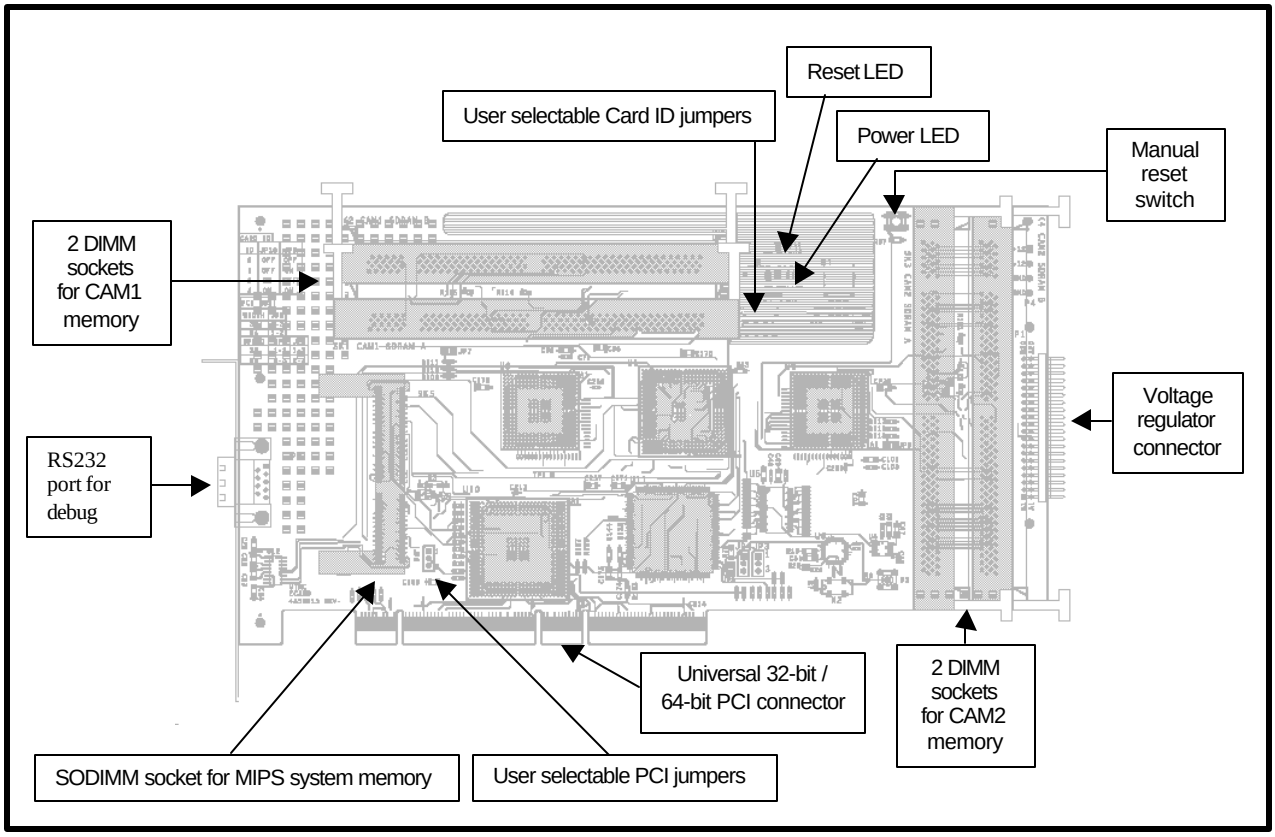


Figure 3 - eCard Layout

System Requirements

The eCard is designed to run in any PCI slot of a Windows 95, 98, NT, or Linux host. There are, however, a few special system requirements:

- The eCard, with the voltage regulator attached, measures 6" by 12.6". This exceeds normal PCI card dimensions and may be too large to some trim-line enclosures.

- The DIMMs and the voltage regulator extend beyond the normal half inch thickness allowed for a PCI card. While a normal PCI card will fit next to an eCard, it is not possible to insert two eCards in adjacent PCI sockets. Generally, it is possible to install an eCard in the last PCI socket without impacting the availability of other sockets.

- Depending upon the airflow in a given enclosure, it may be advisable to install extra cooling fans. One excellent way to accomplish this is by installing an auxiliary fan card in the adjacent PCI socket. One inexpensive source for such a fan card is:

  http://www.pcpowercooling.com/products/cooling/index_cooling.htm

- Like a disk drive or CDROM, the eCard is powered directly from the power supply. Thus, a power supply cable must be available. A six inch pigtail is attached to the eCard to facilitate making this connection.

- Each eCard draws from 27 to 73 Watts of power, depending upon the amount of memory installed. It is critical that the power supply be sized to supply this power in addition to the power required by the rest of the system. In some cases it may be necessary to install a second power supply.

Hardware Installation

WARNING: Static electricity can severely damage electronic parts. Take these precautions:

- Before touching any electronic parts, discharge the static electricity from your body by touching the internal metal frame of your computer while it is unplugged.
- Don't remove the eCard from the anti-static container it is shipped in until you're ready to install it. When you remove it from the computer, place it back in its container.
- Don't let your clothes touch any electronic parts.
- When handling the eCard, hold it by its edges and avoid touching its circuitry.

1. Prepare your computer

Turn off the computer and any peripherals. Open the enclosure and choose a PCI expansion slot that can accommodate the high profile of the eCard with the least impact on cards you may wish to add in the future. See your system manual if you need help identifying the PCI slots. Plugging the eCard into a non-PCI slot could damage the eCard, your computer, or both. Remove the cover for the slot you intend to use. Save the screw for securing the eCard mounting bracket.

2. Set the Card ID

Up to four eCards can be installed in a given system. Jumpers 9 and 10, located to the right of the CAM1 DIMMs, must be properly installed to distinguish between the different cards. This CardID will be used by the eCard Library Functions. If eCard software is provided by a third party, then the CardID will be specified by the developer. As long as the CardIDs are unique, any ID can be used. For example, it is legal for a single card system to have an ID of zero or three. The following table shows how to configure these jumpers.

Card ID	JP10	JP9
0	Off	Off
1	Off	On
2	On	Off
3	On	On

3. Set the card for 32 or 64 bit PCI

The eCard can support either 32 or 64 bit PCI. If the eCard card edge connector is roughly twice the length of your PCI socket, then your system has a 32 bit PCI bus. Jumper 6, located just right of the SO DIMM socket, must be set properly for your card to operate. The following table shows how to configure your system for the correct bus width.

PCI Bus Width	JP6
32 bits	2-3
64 bits	1-2

4. Set the card for 33 or 66 MHz PCI

The eCards can support either a 33 or a 66 MHz PCI interface. Check your motherboard specification if you are not sure which your system supports. Jumpers 3 and 4, located just right of the SO DIMM socket, must be set properly for your card to operate. The following table shows how to configure your system for the appropriate bus speed.

PCI Bus Frequency	JP4	JP3
33 MHz	1-2	1-2
66 MHz	2-3	2-3

5. Install the voltage regulator

Remove the eCard and the voltage regulator from their anti-static containers, paying careful attention to the warning above. Slide the voltage regulator into the 40 pin connector at the end of the card.

6. Install SODIMM

The MIPS system memory is contained on the small outline DIMM (SODIMM) on the left side of the card. Install this memory by sliding the SODIMM into the socket and snapping it into place.

7. Install CAM DIMMs

There are four CAM DIMMs on the card. Locate the two DIMM sockets labeled:

- SK1 CAM1 SDRAM A
- SK2 CAM1 SDRAM B

If CAM1 is to be used, at least socket A must be populated with memory. If both DIMM sockets are populated, they must be populated with identical DIMMs. To install a DIMM, slide it into the socket and press the two locking wings into place.

Repeat this process for CAM2.

8. Insert the eCard

Position the eCard over the expansion slot you have chosen. If your computer only supports 32 bit PCI, you will notice that the card edge connector will extend beyond the PCI socket. This is not a problem. Push the card in firmly and evenly until it is fully seated in the slot. Replace the screw to secure the eCard bracket to your computer chassis.

9. Connect the power cable

Connect that power cable to a spare power connector from your power supply.

10. Connect the debug monitor cable (optional)

If your plan on developing custom code to run on your eCard, you will want to connect the DB9 connector on the back of the eCard to one of your computer's COM ports using a standard 9 pin cable.

11. Restart your computer

Secure the cover on your computer and power it on. Windows will report that new hardware has been detected.

TBD: Screen image of New Hardware dialog… Specific next steps… 95 / 98 / NT differences…

Software Installation

1. For Windows 95, Windows 98 or Windows NT 4.0 systems

Install eCard software drivers, insert the eCard Driver CD in your CD–ROM drive. The setup program automatically starts the installation process. Remainder TBD…

2. **For Linux systems** TBD…

3. Configuring serial debug interface

If the system is to be used for debug, a serial port must be configured as follows:

- 19200 baud
- data bits = 8
- stop bits = 1
- parity = none
- flow control = hardware (RTS/CTS)

Resetting the eCard

It may become necessary to reset the eCard if MIPS software errors occur. Normally, a cold reset of the MIPS is performed. This will return the MIPS to its original operating state and eliminate all CAM data. For debug purposes, a warm MIPS reset is also provided. This will leave the MIPS registers and CAM data intact while debug code examines registers and data. Data and registers are generally corrupted before a worm reset, so after debug data has been gathered, the user will want to perform a cold reset. The eCard can be reset by three means:

- The host can be reset (via power or reset buttons) which will automatically reset the eCard. When this occurs, the host must once again boot the eCard. See the Host Driver Functions for further details.
- The eCard's reset push button can be pressed to reset the eCard's MIPS processor. This function is provided for convenience during software development.
- The host (or MIPS) software can perform a cold reset of the MIPS processor. The Host Driver BootCard driver function performs a cold reset.
- The host (or MIPS) software can perform a warm reset of the MIPS processor. This will preserve registers and CAM data for debug purposes. The Host Driver WarmReset function performs a warm reset.

The table below shows the behavior of each of these reset methods.

Reset Type	MIPS Reaction	PCI Controller Reaction	Host Reaction	Notes
Host reset	**cold reset**	**reset**	**reset**	Hardware reset is performed on power-up. Host software will release the MIPS from reset via the BootCard function.
Button push	**cold reset**	**no effect**	**no effect**	The Galileo PCI controller is not reset. No PCI reconfiguration is required.
Software generated (by MIPS or Host)	**cold reset**	**no effect**	**no effect**	Used by the host BootCard function to re-boot the eCard.
Software generated (by MIPS or Host)	**warm reset**	**no effect**	**no effect**	Used by the host WarmReset function to reset the eCard for debug data gathering.

Table 1 - Reset Conditions

Note: While it is possible for the MIPS processor to place itself into reset, it is dependant upon the host to remove it from reset.

MIPS 5261 Memory Map

The following table describes the memory map of the MIPS processor.

Area	Physical Start Address	Physical End Address	Size / Notes
SDRAM Stack 0	0x0000.0000	0x0FFF.FFFF	256 Mbytes*
SDRAM Stack 1	0x1000.0000	0x1FFF.FFFF	256 Mbytes*
PCI 0 memory 0	0x2000.0000	0x21FF.FFFF	32 Mbytes
PCI 0 memory 1	0x2200.0000	0x23FF.FFFF	32 Mbytes
PCI 0 I/O	0x2400.0000	0x25FF.FFFF	32 Mbytes
Internal registers	0x2600.0000	0x2600.0FFF	4 Kbytes
Device 0 (CS0)	0x2700.0000	0x277F.FFFF	8 Mbytes - UART
Device 2 (CS2)	0x2800.0000	0x287F.FFFF	8 Mbytes - Reset
Boot Area Mapped to GT SDRAM	0x2900.0000	0x2900.0FFF	4 Mbytes
FPGA Status register	0x4.0000.0000	0x7.FFFF.FFFF	Bits [35:34] = 01
CAM 1	0x8.0000.0000	0xB.FFFF.FFFF	Bits [35:34] = 10
CAM 2	0xC.0000.0000	0xF.FFFF.FFFF	Bits [35:34] = 11

Table 2 - MIPS Memory Map

*Dynamically adjusted by boot code based on DIMM configuration

Host Driver Functions

The following host driver routines are provided as public function calls in the eCardHost class library. These drivers allow a host C++ routine to interact with one or more eCards.

CeCardHost()

Prototype: **CeCardHost(void);**
Purpose: This is the constructor. It creates a eCardHost object and initializes its internal variables.
Arguments: None
Returns: Nothing

~CeCardHost()

Prototype: **~CeCard(void);**
Purpose: This is the destructor. It closes the eCardHost object if it has been opened.
Arguments: None
Returns: Nothing

BootCard()

Prototype: **int BootCard(int card, char* bootFile);**
Purpose: This function boots one of the eCards in the system. All data serviced by this card is discarded when BootCard is called. Initialize must be called before Engine functions can be performed on the booted card.
Arguments: **int card**, contains the number of the eCard to be initialized. This is a number between 0 and 3 which is set via jumpers 9 and 10. See Hardware Installation for more details.
char* bootFile, contains the full file path for the boot file and executable code to be downloaded to the eCard.
Returns: int: 0 = Success
1 = Invalid card number
2 = Invalid file name

WritePacketToCard()

Prototype: **int WritePacketToCard(int card, unsigned__int32* eCardPacketPtr, unsigned__int32* hostPacketPtr, int packetSize);**
Purpose: This function writes a packet (often containing a query) to one of the eCards in the system.
Arguments: **int card**, contains the number of the eCard that the packet is being sent to.
unsigned__int32* eCardPacketPtr, contains a pointer to an array in the eCard's system memory space where the packet will be written.
unsigned__int32* hostPacketPtr, contains a pointer to the packet source array on the host.
Int packetSize, contains the number of 32 bit words in the packet.
Returns: int: 0 = Success
1 = Invalid card number
4 = Invalid host packet address
5 = Invalid packet size
6 = Invalid eCard packet address

ReadPacketFromCard()

Prototype: **int ReadPacketFromCard(int card, unsigned__int32* eCardPacketPtr, unsigned__int32* hostPacketPtr, int packetSize);**

Purpose: This function reads a packet (often containing query results) from one of the eCards in the system.

Arguments: **int card**, contains the number of the eCard to read from.
unsigned__int32* eCardPacketPtr, contains a pointer to an array in the eCard's system memory space containing the packet to be read.
unsigned__int32* hostPacketPtr, contains a pointer to a destination array on the host.
Int packetSize, contains the number of 32 bit words in the packet.

Returns: int: 0 = Success
1 = Invalid card number
3 = Invalid eCard packet address
4 = Invalid host packet address
5 = Invalid packet size

WriteMailbox()

Prototype: **int WriteMailbox(int card, unsigned__int32* mailboxEntry);**

Purpose: This function writes a 32 bit PCI mailbox entry to one of the eCards in the system.

Arguments: **int card**, contains the number of the eCard that the packet is being sent to.
unsigned__int32* mailboxEntry, contains the 32 bit mailbox entry to be written.

Returns: int: 0 = Success
1 = Invalid card number

ReadMailbox()

Prototype: **int ReadMailbox(int card, unsigned__int32* mailboxEntry);**

Purpose: This function reads a 32 bit PCI mailbox entry from one of the eCards in the system.

Arguments: **int card**, contains the number of the eCard to be read from.
unsigned__int32* mailboxEntry, contains the 32 bit mailbox entry to be written.

Returns: int: 0 = Success
1 = Invalid card number

WarmReset()

Prototype: **int ReadMailbox(int card);**

Purpose: This function performs a warm reset of an eCard. It stops processing on the card while preserving the registers and CAM data for debug purposes. After debug data has been gathered, it should generally be followed by a BootCard() call.

Arguments: **int card**, contains the number of the eCard to be reset.

Returns: int: 0 = Success
1 = Invalid card number

eCard Library Functions

The following eCard driver routines are provided as public function calls in the eCard class library. These drivers allow an eCard C++ routine to interact with the host and the CAMs. This library is intended for use with a cross-compiler targeting the MIPS 5261 processor.

CeCard()

Prototype: **CeCardHost(void);**
Purpose: This is the constructor. It creates a eCard object and initializes its internal variables.
Arguments: None
Returns: Nothing

~CeCard()

Prototype: **~CeCard(void);**
Purpose: This is the destructor. It closes the eCard object if it has been opened.
Arguments: None
Returns: Nothing

WritePacketToHost()

Prototype: **int WritePacketToHost (unsigned__int32* eCardPacketPtr, unsigned__int32* hostPacketPtr, int packetSize);**
Purpose: This function writes a packet to an array in the host's system memory.
Arguments: **unsigned__int32* eCardPacketPtr**, contains a pointer to a source array in the eCard's system memory space.
unsigned__int32* hostPacketPtr, contains a pointer to the destination array on the host.
Int packetSize, contains the number of 32 bit words in the packet.
Returns: int: 0 = Success
3 = Invalid eCard packet address
4 = Invalid host packet address
5 = Invalid packet size

ReadPacketFromHost()

Prototype: **int ReadPacketFromHost(unsigned__int32* eCardPacketPtr, unsigned__int32* hostPacketPtr, int packetSize);**
Purpose: This function reads a packet from the host's system memory.
Arguments: **unsigned__int32* eCardPacketPtr**, contains a pointer to a receiving array in the eCard's system memory space.
unsigned__int32* hostPacketPtr, contains a pointer to the source array on the host.
Int packetSize, contains the number of 32 bit words in the packet.
Returns: int: 0 = Success
3 = Invalid eCard packet address
4 = Invalid host packet address
5 = Invalid packet size

WriteMailbox()

Prototype: **void WriteMailbox(unsigned__int32* mailboxEntry);**
Purpose: This function writes a 32 bit PCI mailbox entry to the host.
Arguments: **unsigned__int32* mailboxEntry**, contains the 32 bit mailbox entry to be written.
Returns: nothing

ReadMailbox()

Prototype: **void ReadMailbox(unsigned__int32* mailboxEntry);**
Purpose: This function reads a 32 bit PCI mailbox entry from the host.
Arguments: **unsigned__int32* mailboxEntry**, contains the 32 bit mailbox entry to be written.
Returns: nothing

Initialize()

Prototype: **void Initialize(int engine, int memSize, int tablesAllowed);**
Purpose: This function initializes one of the eCard's CAM Engines. All data serviced by this Engine is discarded when Initialize is called. Initialize must be called before other functions can be performed.
Arguments: **int engine**, contains the number of the Engine to be initialized (1 or 2).
int memSize, contains a power of 2 indicating the size (in 64 bit words) of the total eCard memory for the specified Engine.
int tablesAllowed, contains a power of 2 indicating the maximum number of tables to be managed by the Engine. The value must be a number between 0 and 13 inclusive.
Returns: nothing
FIFO Status: 0 = Success
10 = Invalid engine number
11 = Invalid memory size
12 = Invalid number of tables

ConfigNewDiscreteTable()

Prototype: **void ConfigNewDiscreteTable(int engine, int tableNumber, int packMode, int keyWidth, int assocWidth, int tableDepth, int tableType);**

Purpose: This function configures a new discrete table.

Arguments: **int engine**, contains the Engine number (1 or 2).
int tableNumber, contains the number of the table to be configured.
int packMode, 0 if standard record packing.
1 if special key and association packing.
int keyWidth, number of bytes in key (2 to 32).
int assocWidth, if normal width table; number of bytes in association (0 to 32). If extra wide table; power of 2 indicating number of association bytes (0 to 23) (note: virtual chip supports 0 to 8).
int tableDepth, exponent of 2 indicating number of target records in table.
int tableType, 0 for Discrete normal width table.
1 for Discrete extra wide table.

Returns: nothing

FIFO Status:
0 = Success
1 = Invalid table depth
2 = Out of memory
3 = CAM not initialized
5 = Invalid table number
6 = Table already configured
7 = Illegal association width
8 = Illegal key width
22 = Invalid table type

ConfigNewPrefixTable()

Prototype: **void ConfigNewPrefixTable(int engine, int tableNumber, int groupEnabled, int maxSigLen, int minSigLen, int tableDepth, int prefixScratch);**

Purpose: This function configures a new Prefix table.

Arguments: **int engine**, contains the Engine number (1 or 2).
int tableNumber, contains the number of the table to be configured.
int groupEnabled, 0 if not group enabled, 1 if group enabled.
int maxSigLen, 0 if 32 bits, 1 if 23 bits, 2 if 15 bits, 3 if 7 bits.
int tableDepth, exponent of 2 indicating number of target records in table.
int prefixScratch, exponent of 2 indicating number of words of prefix scratch to allocate

Returns: nothing

FIFO Status:
0 = Success
1 = Invalid table depth
2 = Out of memory
3 = CAM not initialized
5 = Invalid table number
6 = Table already configured

setHierarchy()

Prototype: **void SetHierarchy(int engine, int parent, int child);**
Purpose: This function establishes a hierarchy between a child table and a parent table.
Arguments: **int engine**, contains the Engine number (1 or 2).
int parent, number of parent table in hierarchy.
int child, number of child table in hierarchy.
Returns: nothing
FIFO Status: 0 = Success
3 = CAM not initialized
5 = Invalid table number
13 = Invalid hierarchy
20 = Hierarchical association width different

setContext()

Prototype: **void SetContext(int engine, int tableNumber, int proximityBoundary, int manhattan, int returnKey, int modifyMode);**
Purpose: This function sets context to a previously configured table.
Arguments: **int engine**, contains the Engine number (1 or 2).
int tableNumber, number of table to set context to.
int proximityBoundary, exponent of 2 indicating number of nibbles in each proximity element.
int manhattan, 0 if Euclidean proximity match, 1 if Manhattan.
int returnKey, 1 if key is to be returned on proximity search,
0 if only the association should be returned on a proximity search.
int modifyMode, 0 if unique key mode (report duplicate key error)
1 if the association is to be replaced when an add of an existing key occurs.
Returns: nothing.
FIFO Status: No FIFO entry made.

UnloadTable()

Prototype: **int UnloadTable(int engine, unsigned __int64* unloadPtr, unsigned __int32 unloadLimit, unsigned __int32 &unloadCount);**
Purpose: This function unloads the table which currently has context. The output FIFO and the Engine's processing queue must be empty when this function is called, since this function does not operate in a pipelined mode.
Arguments: **int engine**, contains the Engine number (1 or 2).
unsigned __int64* unloadPtr, pointer to array where unloaded table is to be stored
unsigned __int32 unloadLimit, max number of words that can be unloaded into the array (array size)
Returns: **unsigned __int32 &unloadCount**, number of words unloaded into array.
status, 0 = Success
3 = CAM not initialized
14 = No table has context
120 = Unloaded words exceeded unloadLimit
FIFO Status: none.

UnloadMemory()

Prototype: **int UnloadMemory(int engine, unsigned __int32 copyMemoryAddr, unsigned __int32 copyMemorySize, unsigned __int64* unloadPtr, unsigned __int32 unloadLimit, unsigned __int32 &unloadCount);**

Purpose: This function unloads CAM memory into an array. The output FIFO and the Engine's processing queue must be empty when this function is called, since this function does not operate in a pipelined mode.

Arguments: **int engine**, contains the Engine number (1 or 2).
unsigned __int32 copyMemoryAddr, Starting address in physical CAM memory to unload
unsigned __int32 copyMemorySize, Exponent of 2 indicating number of words to unload (valid values = 0 to 32)
unsigned __int64* unloadPtr, pointer to array where unloaded memory is to be stored
unsigned __int32 unloadLimit, max number of words that can be unloaded into the array (array size)

Returns: **unsigned __int32 &unloadCount**, number of words unloaded into array.
status, 0 = Success
3 = CAM not initialized
14 = No table has context
120 = Unloaded words exceeded unloadLimit

FIFO Status: none.

Count()

Prototype: **void Count(int engine);**

Purpose: This function counts the records in the table that has context.

Arguments: **int engine**, contains the Engine number (1 or 2).

Returns: nothing.

FIFO Status: 0 = Success
3 = CAM not initialized
14 = No table has context

FIFO Data: If success, least significant 32 data bits = number of records stored in table.

LoadTable()

Prototype: **int LoadTable(int engine, unsigned __int64* loadPtr, unsigned __int32 loadLimit, unsigned __int32 &loadCount);**

Purpose: This function loads the table which currently has context from data in memory that was either previously unloaded from a table of the same configuration or formatted for the particular configuration. The output FIFO and the Engine's processing queue must be empty when this function is called, since this function does not operate in a pipelined mode.

Arguments: **int engine**, contains the Engine number (1 or 2).
unsigned __int64* loadPtr, pointer to array where unloaded table is to be stored
unsigned __int32 loadLimit, max number of words that can be loaded into the array (array size)

Returns: **unsigned __int32 &loadCount**, number of words loaded into table
status, 0 = Success
3 = CAM not initialized
14 = No table has context
16 = Configurations different on table load
120 = Unloaded words exceeded unloadLimit

FIFO Status: none.

LoadMemory()

Prototype: **int LoadMemory(int engine, unsigned __int32 startAddress, int wordCountExponent, unsigned __int64* loadPtr, unsigned __int32 loadLimit, unsigned __int32 &loadCount);**

Purpose: This function loads CAM memory directly from an array. This function can be very destructive and must be used with care. The output FIFO and the Engine's processing queue must be empty when this function is called, since this function does not operate in a pipelined mode.

Arguments:
int engine, contains the Engine number (1 or 2).
unsigned __ int32 startAddress, first address in memory to write to
int wordCountExponent, exponent of 2 indicating number of words to be written
unsigned __int64* loadPtr, pointer to array where unloaded table is to be stored
unsigned __int32 loadLimit, max number of words that can be loaded from the array (array size)

Returns:
unsigned __int32 &loadCount, number of words loaded into memory
status, 0 = Success
3 = CAM not initialized
21 = Load would exceed memory bounds
16 = Configurations different on table load
120 = Unloaded words exceeded unloadLimit

FIFO Status: none.

AddDiscreteRecord()

Prototype: **void AddDiscreteRecord(int engine, unsigned __int64* key, unsigned __int64* association);**

Purpose: This function adds a record to the discrete table that has context.

Arguments:
int engine, contains the Engine number (1 or 2).
unsigned __int64* key, pointer to array containing record key (4 words minimum)
unsigned __int64* association, pointer to array containing association (must be sized for table)

Returns: nothing.

FIFO Status:
0 = Success
3 = CAM not initialized
9 = Duplicate key
14 = No table has context
15 = Table full, record not added

Note: Both the key and association arrays are filled starting at array element zero. If the value is not a multiple of 8 bytes, the remaining bytes are stored in the least significant bits of the word. For example, to store the ASCII value "0123456789" in the key, the following two assignments must be made:

```
key[1] = 0x3938;
key[0] = 0x3736353433323130;
```

AddPrefixRecord()

Prototype: **void AddPrefixRecord(int engine, unsigned __int32 prefixWord, unsigned __int32association, unsigned int significanceLength, unsigned int group);**

Purpose: This function adds a record to the prefix table that has context.

Arguments: **int engine**, contains the Engine number (1 or 2).
unsigned __int32 prefixWord, 32 bits containing prefix in most significant bits
unsigned __int32 association, 32 bit association
unsigned int significanceLength, number of significant bits in prefixWord
unsigned int group, group ID (zero to 127) if group enabled - otherwise zero

Returns: nothing.

FIFO Status:
0 = Success
3 = CAM not initialized
9 = Duplicate key
14 = No table has context
15 = Table full, record not added

DeleteDiscreteRecord()

Prototype: **void DeleteDiscreteRecord(int engine, unsigned __int64* key);**

Purpose: This function deletes a record to the discrete table that has context.

Arguments: **int engine**, contains the Engine number (1 or 2).
unsigned __int64* key, pointer to array containing record key (4 words minimum)

Returns: nothing.

FIFO Status:
0 = Success
3 = CAM not initialized
9 = Duplicate key
10 = Key not found - could not delete
14 = No table has context

Note: The key array is filled starting at array element zero. If the value is not a multiple of 8 bytes, the remaining bytes are stored in the least significant bits of the word. For example, to store the ASCII value "0123456789" in the key, the following two assignments must be made:

```
key[1] = 0x3938;
key[0] = 0x3736353433323130;
```

DeletePrefixRecord()

Prototype: **void DeletePrefixRecord(int engine, unsigned __int32 prefixWord, unsigned int significanceLength, unsigned int group);**

Purpose: This function deletes a record to the prefix table that has context.

Arguments: **int engine**, contains the Engine number (1 or 2).
unsigned __int32 prefixWord, 32 bits containing prefix in most significant bits
unsigned int significanceLength, number of significant bits in prefixWord
unsigned int group, group ID (zero to 127) if group enabled - otherwise zero

Returns: nothing.

FIFO Status:
0 = Success
3 = CAM not initialized
10 = Key not found - could not delete
14 = No table has context

SeekDiscreteRecord()

Prototype: **void int SeekDiscreteRecord(int engine, unsigned __int64* key);**
Purpose: This function seeks a record in the discrete table that has context.
Arguments: **int engine**, contains the Engine number (1 or 2).
unsigned __int64* key, pointer to array containing record key (4 words minimum)
Returns: nothing.
FIFO Status: 0 = Success
3 = CAM not initialized
10 = Key not found – no association returned
14 = No table has context
19 = Key not found, hierarchy broken
FIFO Data: if status = 0, FIFO data contains association

Note: The key array is filled starting at array element zero. If the key is not a multiple of 8 bytes, the remaining bytes are stored in the least significant bits of the word. For example, to store the ASCII value "0123456789" in the key, the following two assignments must be made:

```
key[1] = 0x3938;
key[0] = 0x3736353433323130;
```

SeekPrefixRecord()

Prototype: **void SeekPrefixRecord(int engine, unsigned __int32 prefixWord, unsigned int group);**
Purpose: This function seeks a record in the prefix table that has context.
Arguments: **int engine**, contains the Engine number (1 or 2).
unsigned __int32 prefixWord, 32 bits containing prefix in most significant bits
unsigned int group, group ID (zero to 127) if group enabled - otherwise zero
Returns: nothing.
FIFO Status: 0 = Success
3 = CAM not initialized
10 = Key not found – no association returned
14 = No table has context
FIFO Data: if status = 0, FIFO data contains association

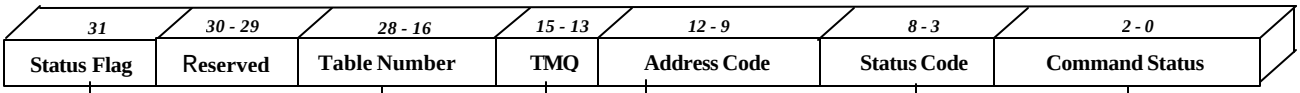
SeekProximity()

Prototype: **void SeekProximity(int engine, unsigned __int64* key);**
Purpose: This function seeks the closest record from the discrete table that has context using the proximity parameters (proximityBoundary, manhattan, and returnKey) that were established when context was set.
Arguments: **int engine**, contains the Engine number (1 or 2).
unsigned __int64* key, pointer to array containing record key (4 words minimum)
Returns: nothing.
FIFO Status: 0 = Success
3 = CAM not initialized
10 = Key not found – no association returned
14 = No table has context
25 = illegal proximity seek against non-discrete table
FIFO Data: if status = 0, FIFO data contains association (and the key if “return key” was specified when context was set).

Note: The key array is filled in the same manner as it is for SeekDiscreteRecord().

ReadStatus()

Prototype: **unsigned __int32 ReadStatus(int engine,);**
Purpose: This function returns the status from the oldest entry in the UTCAM-Engine FIFO.
Arguments: int engine, contains the Engine number (1 or 2).
Returns: unsigned __int32 status, indicates status of oldest FIFO entry as indicated below:



Field	Description
Status Flag	Always set to one. (see note 1)

Field	Description
Table Number	Specifies the number of the table that had context when the command that generated the status word began execution.

Field	Description
Table Management Qualifier	Indicates the table management qualifier of the operation that generated the status word if the Address Code of that operation was 2. Otherwise, this field is set to zero.

Field	Description
Address Code	Indicates the address code of the operation that generated the status word

Field	Description
Command Status	Indicates the status of the oldest command in the output FIFO: (see note 2) 0 = No operation complete (output FIFO empty) 1 = Command completed with non-zero status code. 2 = Command completed successfully, no data generated 3 = Command completed successfully, one data word remains in output FIFO 7 = Command completed successfully, more than one data word remains in output FIFO 4, 5, 6 = Reserved

Field	Description
Status Code	Indicates the status of the oldest command in the output FIFO. If the status code is zero and if the command status indicates that a terminated command is in the output FIFO, then the command terminated successfully. Otherwise, the status code indicates the reason for an abnormal termination. (see *section* ***Error! Reference source not found.***)

Notes: **1)** If the returned status = 80000000 hex, then there is no pending output in the output FIFO.

2) If the command that produced the FIFO entry did not produce any FIFO data, either because it never does or because of a special condition, then the call to ReadStatus() will have the side effect of incrementing the FIFO to the next oldest command. Otherwise, the FIFO won't be incremented and subsequent ReadStatus() calls will yield the same result.

3) See the command that generated the FIFO entry for a description of the valid status codes.

ReadData()

Prototype: unsigned __int64 ReadData(**int engine,**);
Purpose: This function reads a 64 bit word, containing either status or data, from the oldest FIFO entry.
Arguments: **int engine**, contains the Engine number (1 or 2).
Returns: **unsigned __int64 value**, contains either status or data. If the command that produced the FIFO entry did not produce any FIFO data, either because it never does or because of a special condition, then the call to ReadData() will return status in the least significant 32 bits of the return value and increment the output FIFO. See the ReadStatus() command to interpret status results. Otherwise, the call to ReadData() will return the next available 64 bit data word. When the last data word is read, the FIFO will be incremented.

Working with FIFOs

Each CAM Engine contains four 256-bit input and four 256-bit output FIFO entries. This allows the programmer of the eCard to pipeline CAM commands and achieve true parallel processing with the MIPS processor and the CAM Engines.

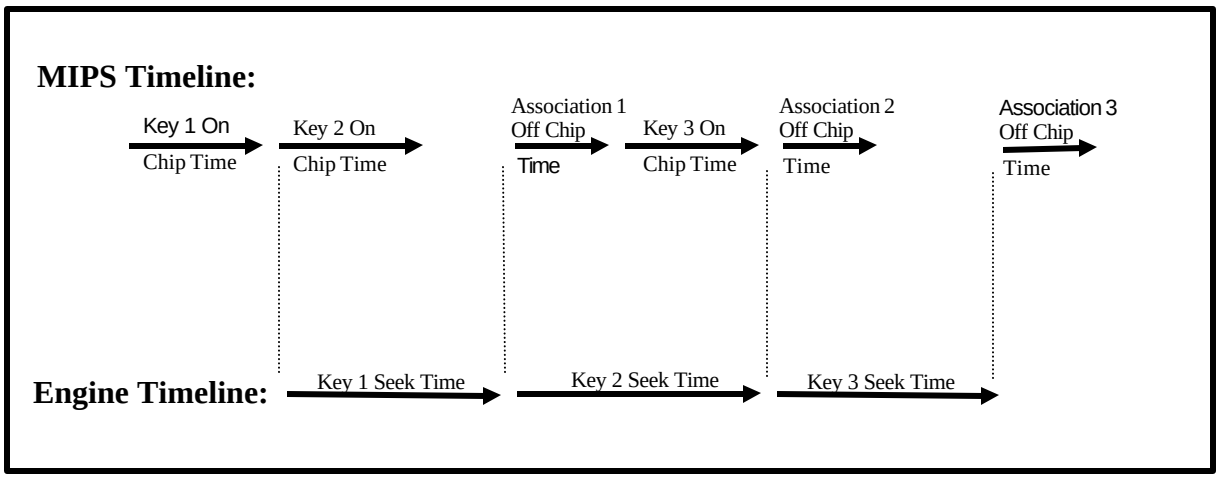


Figure 4 - Timeline Example Showing Pipelined Seeks

It is important not to overflow the FIFO capacity which can result in degraded performance or system lockup.

The recommended procedure is to be no more than three commands ahead at any point in time. This is accomplished by writing two or three commands to one of the CAM Engines and then relieving the output FIFO with the results of the first before writing another command to the input FIFO.

Debugging eCard Applications

The eCard has been designed for ease of programming and debug.

Error Trapping

Memory range violations, as well as parity errors, are automatically trapped by the eCard. Error trapping code is built into the eCard's MIPS kernel. When such an event occurs, the MIPS kernel automatically reports the error to the host via a PCI doorbell interrupt and places the MIPS processor in a warm reset. The eCard's debug features can then be used to examine registers, system memory, and CAM memory.

Debug Monitoring

The debug aware MIPS kernel, provided with the eCard, provides users with the ability to perform normal software debug functions from an application running on the host. Communication with the eCard is performed via a serial connection between the eCard and the host.

Trouble Shooting

TBD.

Support and Training

UTMC provides complete documentation for the eCard on our web site at http://www.utmc.com/. This includes schematics, FPGA design code, and other reference design materials.

For technical hardware or software support, call 1-800-645-UTMC.

Training classes are also available upon request.

Revision History

None.