SLAC **Stanford Linear Accelerator Center**

Flash Core Set

Electronics group

design specification

Document Version:	0.3
Document Issue:	1
Document Edition:	English
Document Status:	Draft - for internal distribution only
Document ID:	XXX-TD-00860
Document Date:	July 20, 2006

SLAC Stanford Linear Accelerator Center (SLAC)
2575 Sandhill Road
Menlo Park California, 94025 USA

This document has been prepared using the Software Documentation Layout Templates that have been prepared by the IPT Group (Information, Process and Technology), IT Division, CERN (The European Laboratory for Particle Physics). For more information, go to http://framemaker.cern.ch/.

Abstract

The Flash Memory Controller (FMC) is Intellectual Property (core IP) designed to provide an interface between a set of flash memory devices and either a PCI bus or PCI *Express* interface. The FMC is designed to operate with the *Samsung K9XXG08UXM* family of flash devices and can manage as much as 128 Gigabytes of flash storage. The interface is strictly divided into data and control side interfaces. The data side interface supports any variety of PCI bus configurations, including speeds of either 33 or 66 MHZ, and widths of either 32 or 64 bits. The FMC contains a generic DMA engine (bus master) interface which allows for split-transactions (concurrent reads and writes). The control side interface is designed to operate as a slave (*target*) on a PCI bus and includes support for notification of I/O activity and completion through PCI interrupts. Although the control interface is designed with PCI in mind, the interface is generic enough to allow many different control implementations.

Hardware compatibility

This document assumes the following hardware revision:

FMC: Version TBD

Intended audience

This document is intended principally as a guide for the *developer* and *users* of the Flash Memory Controller (FMC).

Conventions used in this document

Certain special typographical conventions are used in this document. They are documented here for the convenience of the reader:

- Field names are shown in bold and italics (*e.g.*, ***respond*** or ***parity***).
- Acronyms are shown in small caps (*e.g.*, SLAC or TEM).
- Hardware signal or register names are shown in Courier bold (*e.g.*, `RIGHT_FIRST` or `LAYER_MASK_1`)

References

1 *Xilinix* Virtex-4 Family Overview. Dated February 10, 2006

2 *Actel*, CorePCI Target, Master, and Master/Target Product specification (v3.0), October 2000.

3 *Xilinix* Spartan-3 FPGA Family: Complete Data Sheet. Dated April 03, 2006

4

Document Control Sheet

Table 1 Document Control Sheet

Document	**Title:**	Flash Core Set design specification		
	Version:	0.3		
	Issue:	1		
	Edition:	English		
	ID:	XXX-TD-00860		
	Status:	Draft - for internal distribution only		
	Created:	February 9, 2002		
	Date:	July 20, 2006		
	Access:	Z:\Private\pcp\fcs\v2.1\frontmatter.fm		
	Keywords:	Flash Memory Controller		
Tools	**DTP System:**	Adobe FrameMaker	**Version:**	6.0
	Layout Template:	Software Documentation Layout Templates	**Version:**	V2.0 - 5 July 1999
	Content Template:	--	**Version:**	--
Authorship	**Coordinator:**	Michael Huffer		
	Written by:	Michael Huffer		

Document Status Sheet

Table 2 Document Status Sheet

Title:	Flash Core Set design specification		
ID:	XXX-TD-00860		
Version	**Issue**	**Date**	**Reason for change**
1.0	1	4/24/2006	Initial draft

Table of Contents

List of Figures

List of Tables

Chapter 1
Principals of operation

1.1 Introduction

The Flash Core Set (FCS) consists of two types of Intellectual Property (IP) macros, which when instantiated in a design, provide an interface between an arbitrary sized set of flash memory and an external communication bus or serial protocol. The central component of the core set is the Flash Memory Control (FMC). The FMC manages four flash devices and presents two 32-bit transfer buses, capable of moving data at up to 40 Mbytes/secs. One bus is used to transfer information *to* the FMC (the *Inbound* bus) and the other transfers information *from* the FMC (the *Outbound* bus). This allows the FMC to be incorporated naturally into a design which interfaces to a split I/O bus. Depending on the specific flash device used, one FMC is capable of managing from 4 to 16 Gigabytes of storage.

Operations are initiated through a separate *Initiator* interface. The FMC *queues* operations, allowing the act of operation initiation to be decoupled from its completion. The set of operations supported by the FMC can be summarized as follows:

- Randomly *read* flash memory at a granularity of a *code block* (132 bytes).
- Randomly *write* flash memory at a granularity of a *page* (8 Kilobytes).
- Randomly *erase* flash memory at a granularity of a *flash block* (512 Kilobytes).
- Access flash device attributes and FMC performance counters.

The FMC is described in additional detail, starting in Section 1.3. In order to increase the amount of memory in a design, while continuing to scale its performance, a design is expected to instantiate many FMCs. The Initiator side of such a design would take on the form illustrated in Figure 2:

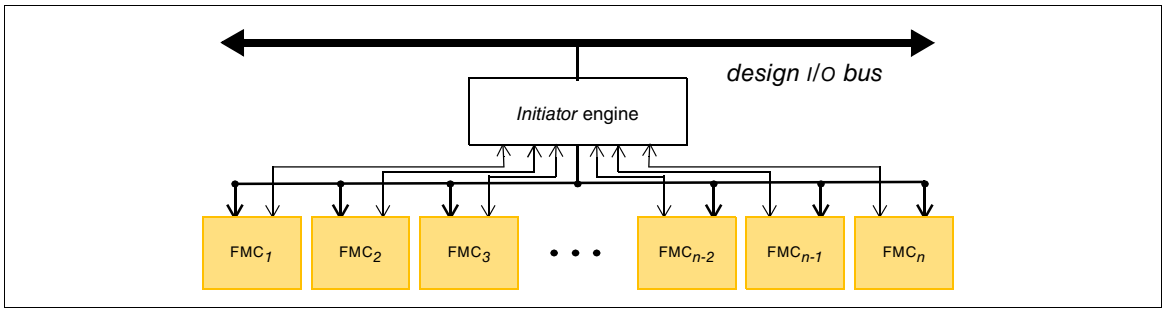

Figure 1 Abstract design of a flash memory system using the FCS

Here, the design's Initiator engine responds to requests from the external bus in order to initiate transactions on any of its FMCs. The case of the transfer side of the design is somewhat more complicated. As each FMC operates autonomously with respect to its peers they must, of a necessity, compete with each other for the central resource represented by the design bus. In order to mediate this competition in a fashion appropriate to the architecture of the FMC, the core set provides an Arbiter macro. The abstract model of a design is shown in Figure 2:

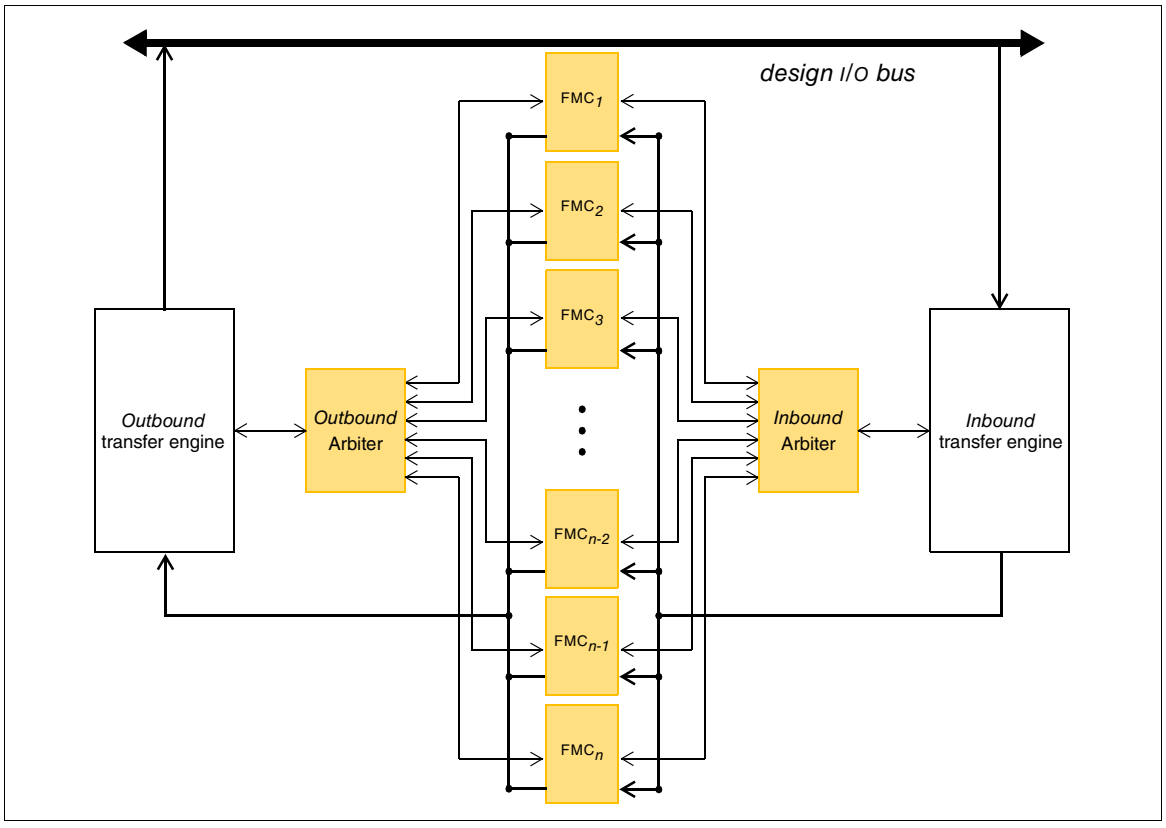

Figure 2 Abstract design of a flash memory system using the FCS

The coloured boxes represent the functionality provide by the core set: a variable number of FMCs and two arbiters. One arbiter to mediate inbound transfers and the other to mediate outbound transfers. One side of each arbiter is connected to the set of FMCs and the other side

to the appropriate in-going or out-going DMA engine. These engines transfer information to and from the design specific external I/O bus. One final box

1.2 The Flash Devices used by the FMC

The FMC is designed around the *Samsung K9XXG08UXM* family of flash devices [2]. Devices from this family provide an 8-bit I/O interface which operates at a maximum rate of 10 Mbytes/sec and therefore, to gain a reasonable match between device and external I/O rates, four such devices are operated by the FMC in parallel. The largest such device from this family is the The *K9NBG08U5M* and is a 4G *x* 8 bit part and therefore, any one FMC is capable of managing as many as 16 Gigabytes of flash storage.

Each member of the *K9XXG08UXM* family is constructed by stacking a number of 1G *x* 8 bit chips[1]. For example, the K9NBG08U5M stacks four such chips. Each chip contains a (2K + 64) byte cache in order to buffer over the (relatively) long times to read and write its actual storage cells. This implies accessing information from a flash device is a two-stage operation. For example, to read information, data is first fetched from cell to cache (a relatively long operation) and then from cache to user (a relatively short operation). *Samsung* refers to the maximum amount of information which can be buffered in the cache as a *page*. Information is *committed* (written) to a device in units of *pages*, but *erased* in units of *blocks*. A device block contains 64 pages or (128K + 4K) bytes and one chip contains 8K blocks or 512K pages.

As each chip contains its own cache, a device contains as many caches as is does chips. For example, the K9NBG08U5M contains four. Each chip of a device is controlled independently through separate chip enables (hereafter referred to as CE_0 - CE_3). These enables allow each chip to be programmed and erased simultaneously. However, unfortunately, these enables are not sufficient to allow simultaneous memory access, as all chips of a device share a common I/O port. For the FMC, similar chip enables are simply ganged together and used solely to extend addressing range.

As four devices are managed by the FMC, this implies FMC page and block sizes are four times the size of any one flash device. That is, a FMC:

— *page* is 8K + 256 (8448) bytes

— *block* is 512K + 16K (540672) bytes

The metrics for each member of the family when used within a FMC are summarized in Table 3:

1. Starting from the K9WAG08U1M.

Table 3 Relationship between device type and memory size

device type	chip size	number of chips	(data) capacity
K9*WAGO8U1M*	*1 Gigabyte*	*1*	*4 Gigabytes*
K9*K8G08U0M*	*2 Gigabytes*	*2*	*8 Gigabytes*
K9*NBG08U5M*	*4 Gigabytes*	*4*	*16 Gigabytes*

As described below (see Section 1.4.2), the FMC defines the unit of read quanta as the *Code Block*. One code block is 132 bytes long and therefore a FMC cache contains sixty-four (64) code blocks. Whenever the initiator requests access to a code block within a page, the FMC reads (simultaneously) 33 bytes from each of its four devices. In other words, a code block is accessed by asserting the *same* chip enable on all four of its devices. These relationships are all illustrated in Figure 3:

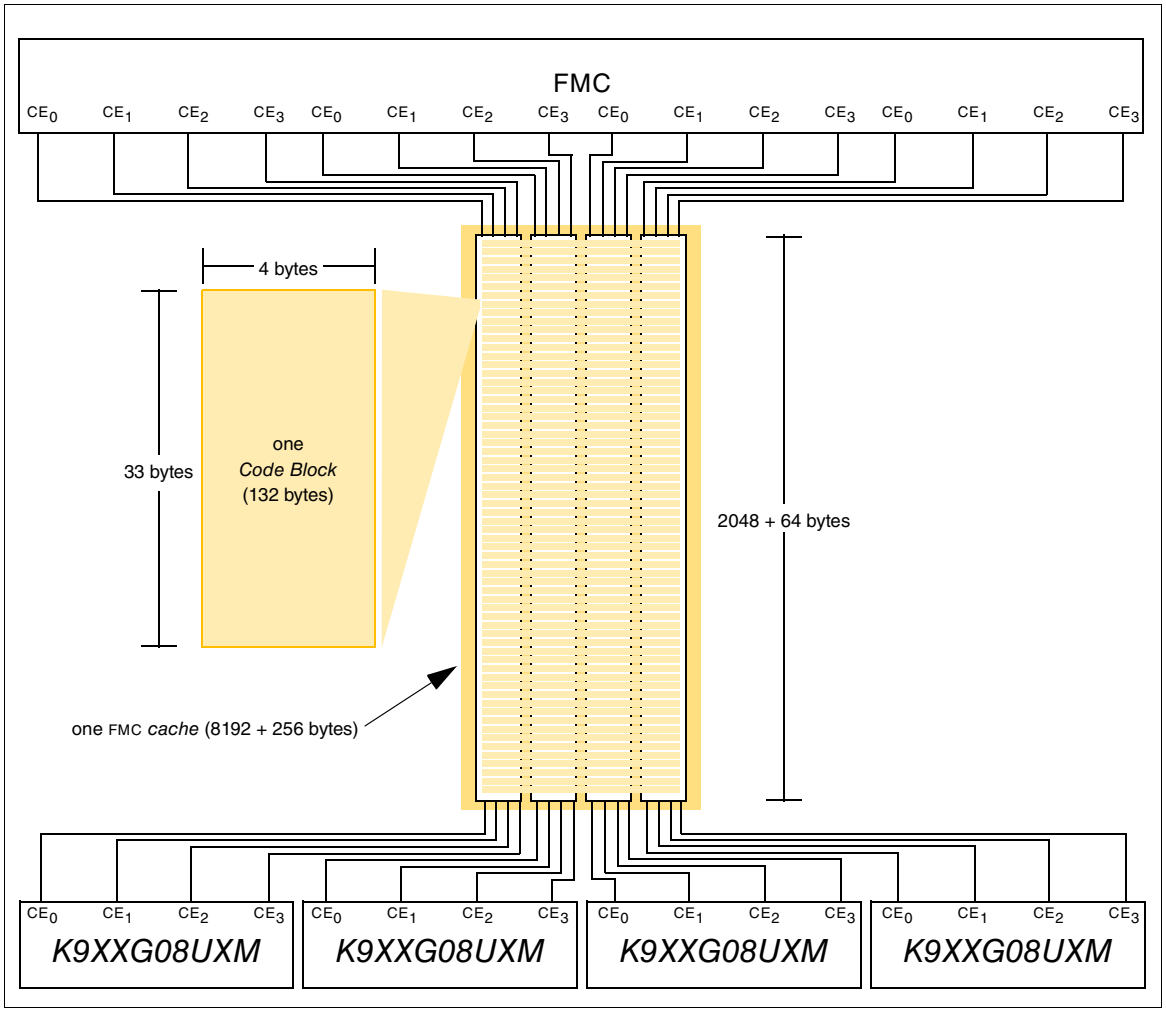

Figure 3 Interleaving of code blocks within the FMC

1.2.1 Page data structure and data encoding

The design of the FMC assumes the user stores their data Reed-Solomon (R-S) encoded. Therefore, the organization of a page and access to the information in that page are biased to maximize

Note, that there is no *a-priori* way to know whether or not the data in a code block is actually R-S encoded, This knowledge must be held externally. For this reason, if inbound engines include decoding their must be a mechanisms to suppress such.

The quanta of R-S processing is the *symbol*, where typically, one symbol is contained in one byte (8 bits). A *Code block* is defined as the sum of the encoded symbols plus the information necessary to correct these symbols. R-S codes are normally referred to as (*n*,*k*) codes, where *n* is the total number of symbols in one code block and *k* is its number of information (or *data*) symbols. The difference (*n* - *k*), is the number of *check* symbols and the maximum number of corrections per code block is half this number or (*n* - *k*)/2.

The FMC assumes a (*132*, *128*) code block and therefore, one page contains sixty-four of these blocks. This encoding corrects up to two symbols in each block, which implies up to 128 corrections are possible in each page. This amount of data correction is somewhat arbitrary and is designed to maximize the usage of the flash device's supplemental storage. As we gain experience in these devices and more fully understand their failure rates this scheme could change. Encoding adds about *1/4 us* of latency to a *Write Page* transaction and decoding adds about *1 us* of latency to a *Read Blocks* operation.

These relationships are illustrated in Figure 4:

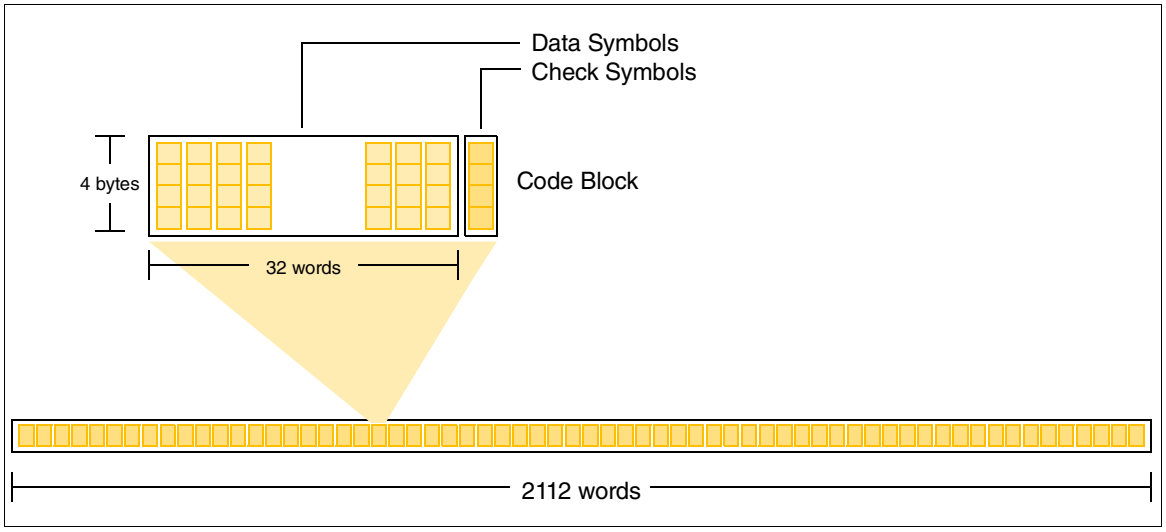

Figure 4 Page Organization

1.2.2 Flash Attributes

This parameter contains the information returned in the last four bytes of the device's ID register. The structure of this parameter is illustrated in Figure 5:

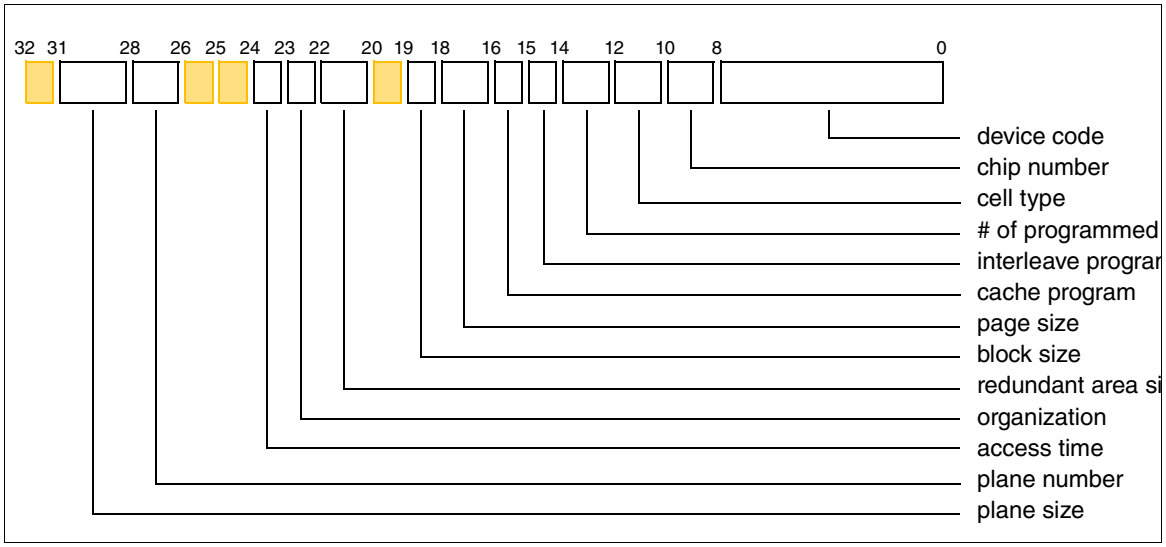

Figure 5 Structure of the returned word for the "Get Flash Attributes" command

Where:

Device code: TBD

1.3 The FMC

The FMC is designed around the *Samsung* K9XXG08UXM family of flash devices [2]. Devices from this family provide an 8-bit I/O interface which operates at a maximum rate of 10 Mbytes/sec and therefore, to gain a reasonable match between device and external I/O rates, four such devices are operated by the FMC in parallel. The FMC implements the following functions:

Read one to sixty-three blocks: Read one to sixty-three blocks of information from a specified device address. Fetch the page from block to cache and then read the appropriate data from cache. Transfer the read data to the specified remote address. Increment the appropriate performance counters (see Section 1.5).

Write page: Write one page (8192 bytes) to a specified device address. Transfer the data for the page from the specified remote address. Increment the appropriate performance counters (see Section 1.5).

Move page: Move the data from one specified page address to another specified page address. This operation entails *no* activity on the remote memory bus. Increment the appropriate performance counters (see Section 1.5).

Erase block: Erase the data from the specified block address. This operation entails *no* activity on the remote memory bus. Increment the appropriate performance counters (see Section 1.5).

Read Flash attributes: Retrieve on board chip information for the four devices of a slice. Return the sampled value in the result.

Read FMC Attributes: Establish the conditions under which the specified FMC will assert an interrupt. Return the sampled value in the result.

Read Counter: Sample the current value of the specified performance counter (see Section 1.5). Return the sampled value in the result.

The interfaces are described in Chapters 2 and 4. A block diagram expressing these interfaces is illustrated in Figure 6:

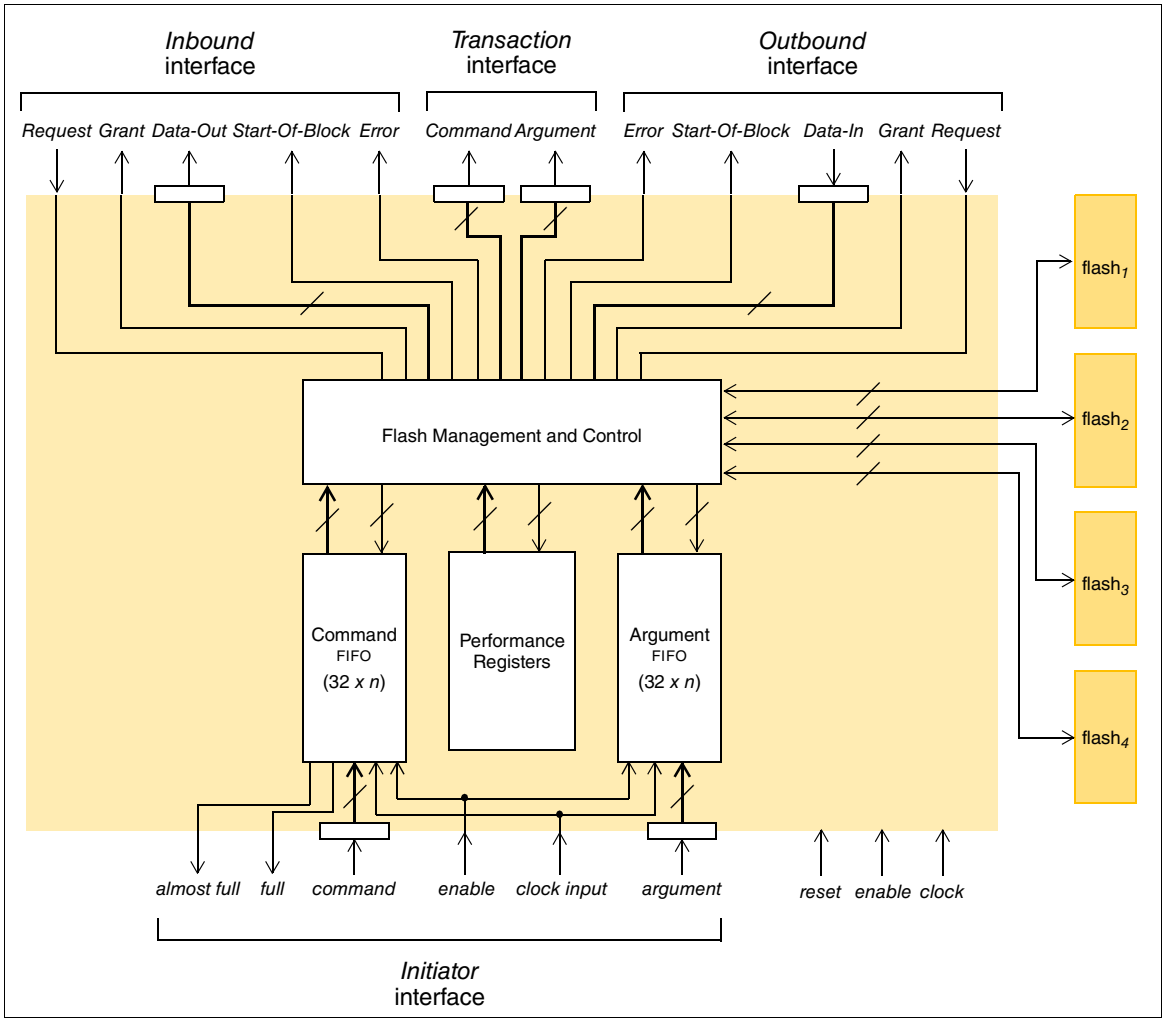

Figure 6 Block diagram and Interfaces of the FMC

1.3.1 FMC Attributes

this section is not complete.

The FMC has three parameters:

— inbound/outbound arbitration timeout

— definition of almost full (same for both FIFOs)

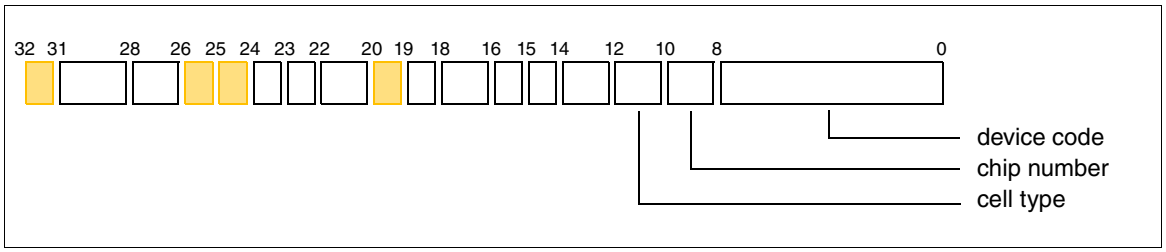

Figure 7 Structure of word for the “Get FMC Attributes” command

1.3.2 FMC Resets

TBD

1.4 Transactions

The FMC process requests in units of *transaction*. Transactions originate with an *Initiator*. The specification of a transaction request is called a *command*. The FMC is single threaded, i.,e., it can only perform a single transaction at a time. However, it contains two FIFOs which are used to buffer transaction requests while busy and therefore, the FMC is capable of processing requests from multiple initiators. In order to allow the initiator to operate in a clock domain independent of the FMC these are *asynchronous* FIFOs.

While of course, different transactions perform different functions, all transactions follow the same three phases:

— Initiator constructs a command. The command includes the type of operation to perform and the parameters for that specific operation. For example, to *read* data from a flash device requires a specification of the chip, block and page to read as well as the address where the read data is to be returned.

— Using the FMC’s Initiator Interface, the initiator queues the command to the FMC. This interface is specified in Section 1.3.

— Whenever the FMC is both idle and its command request FIFOs are *not empty*, a command is dequeued, decoded, and the necessary transaction initiated. The FMC is now *busy*.

— At some point, while busy, the FMC must either fetch or send data. requires the services of either an inbound outbound transfer engine. An inbound engine is used to fetch data and an outbound engine to send data. The transaction continues until the appropriate engine is actually required and at that point the FMC arbitrated for the appropriate engine.

— When granted, the FMC performs the necessary transfer by exchanging data between engine and FMC in units of 32-bit words. For such an eventuality, the FMC specifies both an inbound and outbound interfaces (see Section 1.6).

— Once the transaction is complete, the FMC (conditionally) increments the appropriate performance counter(s).

1.4.1 Commands

Commands have the structure illustrated in Figure 8:

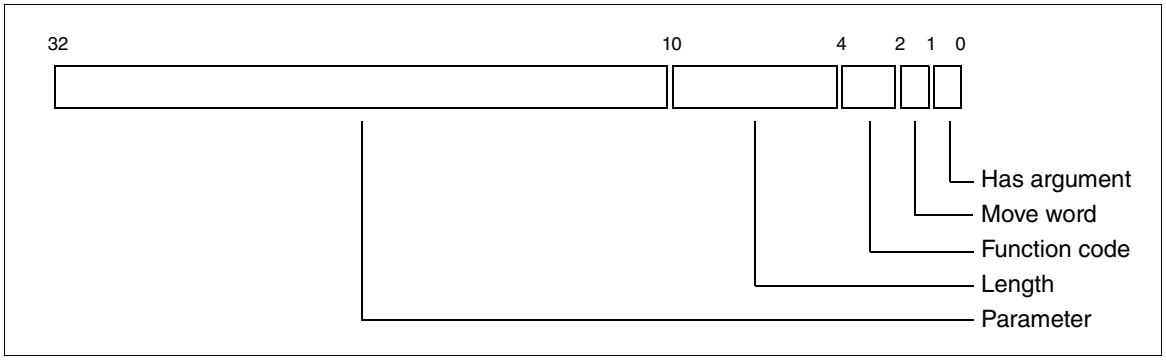

Figure 8 Generic structure of a transaction command

Where:

Has argument: Commands may have either *zero* or *one* argument. This field determines the number of command parameters. If the field is asserted, the command has an argument. If the command does specify an argument, the value of this argument is registered into argument port specified in xxx.

Move word: This field determines whether or not the command implies a transfer of *words* or *code blocks* (see Section 1.4.3). If the field is *not asserted*, the command transfers code blocks either to or from the FMC. If this field is *asserted*, the command transfers zero, one, or more 32-bit words. In addition to specifying the structure of the data transferred, this field determines the interpretation of the ***Length*** field (see below).

Function code: This field contains a small integer which enumerates the *function* to be performed by the command. The possible values for this field are enumerated in Table 4.

Length: This field specifies the transfer length. Its interpretation depends on the value of the ***Move word*** field (see above). If the ***Move word*** field is asserted, the length field is interpreted as the number of code blocks to transfer, where a value of *zero* (0) corresponds to *one* block, a value of *one* (1) to *two* blocks, a value of *two* (2) to *three* blocks, and so forth. If the ***Move word*** field is *not* asserted, this field is interpreted as the number of 32-bit words to transfer, where a value of *zero* (0) corresponds to *none*, a value of *one* (1) to *one* word, a value of *two (2)* to *two* words, and so forth.

Parameter: This field determines the command's parameterization. Its interpretation is function specific. For example, commands which target the FMC's flash devices (for example: ***Set Page*** or ***Erase Block***) interpret the this field as a flash ***Device address*** (see Section 1.4.2). See the specific description for the corresponding command in order to determine the structure of this field.

The command set is enumerated within Table 4:

Table 4 FMC commands

Command	Has argument?	Move word?	Function code	Length	described in:
Get Flash Attributes	*Yes*	*Yes*	0	*1*	Section 3.6
Get FMC Attributes	*Yes*	*Yes*	1	*1*	Section 3.7
Get Counter	*Yes*	*Yes*	2	*1*	Section 3.8
Erase Block	*No*	*Yes*	3	*0*	Section 3.5
Get Blocks	*Yes*	*No*	0	n[1]	Section 3.2
Set Page	*Yes*	*No*	2	*63*	Section 3.3
Move Page	*Yes*	*No*	3	*0*	Section 3.4

1. where *n* can vary from *zero* (0) to *sixty-three* (63).

1.4.2 Addressing

Commands which target the FMC's flash devices (for example, read, write, or erase), interpret the command parameter as a flash ***Device address***. The structure of a device address is illustrated in Figure 9:

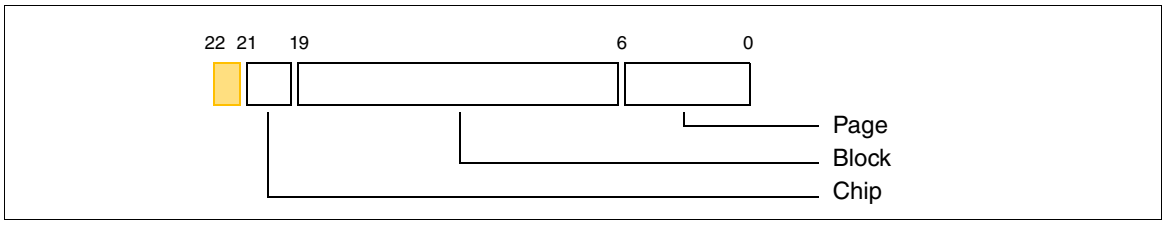

Figure 9 Structure of a Device Address

Where the device address is composed of:

Page: This field specifies the page number whose data is to be accessed. Page numbers vary from *zero* (0) to *63* (decimal).

Block: This field specifies the block number which contains the page whose data is to be accessed. Block numbers vary from *zero* (0) to *8,191* (decimal).

Chip: This field specifies the block number which contains the page whose data is to be accessed. Chip numbers vary from *zero* (0) to *3*.

1.4.3 Units of data transfer

A transaction moves data in units of words or code blocks.

— words

— code blocks

Note, that there is no *a-priori* way to know whether or not the data in a code block is actually R-S encoded, This knowledge must be held externally. For this reason, if inbound engines include decoding their must be a mechanisms to suppress such.

Engine and FMC transfer information in units of *code blocks*. A block contains 132 *symbols* of R-S encoded data where a symbol is contained in one byte. The block is further divided into 128 *Data* symbols and four (4) *Check* symbols. As this data is exchanged between engine and FMC through a 32-bit interface port, data symbols are transferred in thirty-two (32) clocks and check symbols in a single (1) clock. The first four symbols are transferred in the *zeroth* clock and the last four symbols in the *thirty-second* clock. Within any one clock, symbols are found in increasing order with respect to the thirty-two fields of the data port. For example, on the first clock of a block transfer, the *fifth* symbol is contained in fields *0-7*, the *sixth* symbol in fields *8-15*, the *seventh* symbol in fields *16-23*, the *eight* symbol in fields *24-31*, and so-forth. The check symbols are always the last symbols of a block (increasing clock order). The number of blocks in any one transaction can vary from one (1) to sixty-four (64) and is specified by the FMC to the transfer engine as a field through the CMND port of the appropriate transfer interface. These relationships are illustrated in Figure 10:

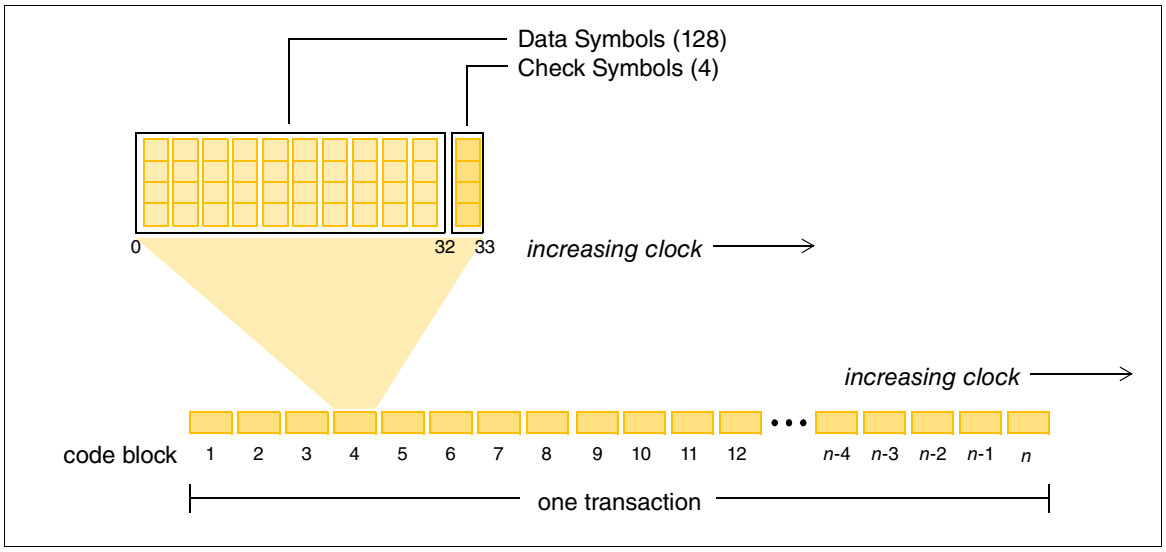

Figure 10 Code block transfers

1.5 Performance counters

Each FMC maintains a set of counters to monitor its health. These are 32-bit, non-saturating counters[1]. These counters are (and may, only be) *re-zeroed* when the FMC is *reset* (see Section 1.3.2). An enumeration of these counters is given in Table 5:

Table 5 FMC performance counters

Name	Number	described in...
Reads	0	Section 1.5.1
Writes	1	Section 1.5.2
Moves	2	Section 1.5.3
Erasures	3	Section 1.5.4
Device Errors	4	Section 1.5.5
Arbitration time	5	Section 1.5.6
Busy time	6	Section 1.5.7
Arbitration timeouts	7	Section 1.5.8
Command Congestion	8	Section 1.5.9

1. The counter simply rolls over when it overflows.

The ***Get Counter*** command (see Section 3.8) is used to sample the current value of any one of these counters. The parameter for this command is the *number* of the counter to be sampled. The correspondence between counter name and number is enumerated in Table 5. In order to understand *what* is being counted, there are two time intervals which must be defined:

Busy: The interval of time (for any one transaction) from when the FMC dequeues a pending transaction request until the FMC could potentially begin a new transaction. I.e., the time spent in a transaction. Time is measured in units of *clock tics*.

Arbitrating: The interval of time (for any one transaction) from which the FMC requests the services of a transfer engine until the arbiter grants that request. The FMC constrains how long in will wait for arbitration (the arbitration *timeout*). Note, that arbitration time is one component of a transactions's *busy* time. Time is measured in units of *clock tics*.

1.5.1 The *Reads* counter

This counter contains the total number of ***Get Blocks*** transactions since the FMC was *reset* and is incremented at the conclusion (successful or otherwise) of each ***Get Blocks*** transaction (see Section 3.2).

1.5.2 The *Writes* counter

This counter contains the total number of ***Set Page*** transactions since the FMC was *reset* and is incremented at the conclusion (successful or otherwise) of each ***Set Page*** transaction (see Section 3.3).

1.5.3 The *Moves* counter

This counter contains the total number of ***Move Page*** transactions since the FMC was *reset* and is incremented at the conclusion (successful or otherwise) of each ***Move Page*** transaction (see Section 3.4).

1.5.4 The *Erasures* counter

This counter contains the total number of ***Erase Block*** transactions since the FMC was *reset* and is incremented at the conclusion (successful or otherwise) of each ***Erase Block*** transaction (see Section 3.5).

1.5.5 The *Device Errors* counter

This counter contains the total number of flash device errors since the FMC was *reset.* This counter can be (potentially) incremented for that any transaction which accesses the flash. However, it may increment at most once and only once per transaction.

1.5.6 The *Arbitration time* counter

The total *amount* of time (in clock tics), since the FMC was *reset*, spent in *arbitration*.

1.5.7 The *Busy time* counter

The total *amount* of time (in clock tics), since the FMC was *reset*, spent *busy.*

1.5.8 The *Arbitration timeouts* counter

The total *number* of times, since the FMC was *reset*, an arbitration request timed out.

1.5.9 The *Command Congestion* counter

The total *number* of times, since the FMC was *reset*, the FMc's request FIFO was *not empty*, while the FMC was busy.

1.6 The Arbiter

Because FMCs operate autonomously, they potentially require the services of any one engine simultaneously. The arbitrator contains a state machine responsible for deciding which FMC is granted an engine in such an eventuality. Note, that each read and write engine has their own arbitration engine, however, the behaviour of both engines are identical. The arbitrator's states and transitions are illustrated in Figure 12. An arbitrator takes input from both its corresponding engine and a set of FMCs when making its decisions. The arbitrator sees its paired engine as either *idle* or *busy.* A engine is considered busy under any one of the following conditions:

- A transaction is in progress
- Its *data* FIFO is more then 3/4 *full*
- Its *pending* FIFO is more then 3/4 *full*

A block diagram expressing these functions and the interface between arbitrator and FMC is illustrated in Figure 11:

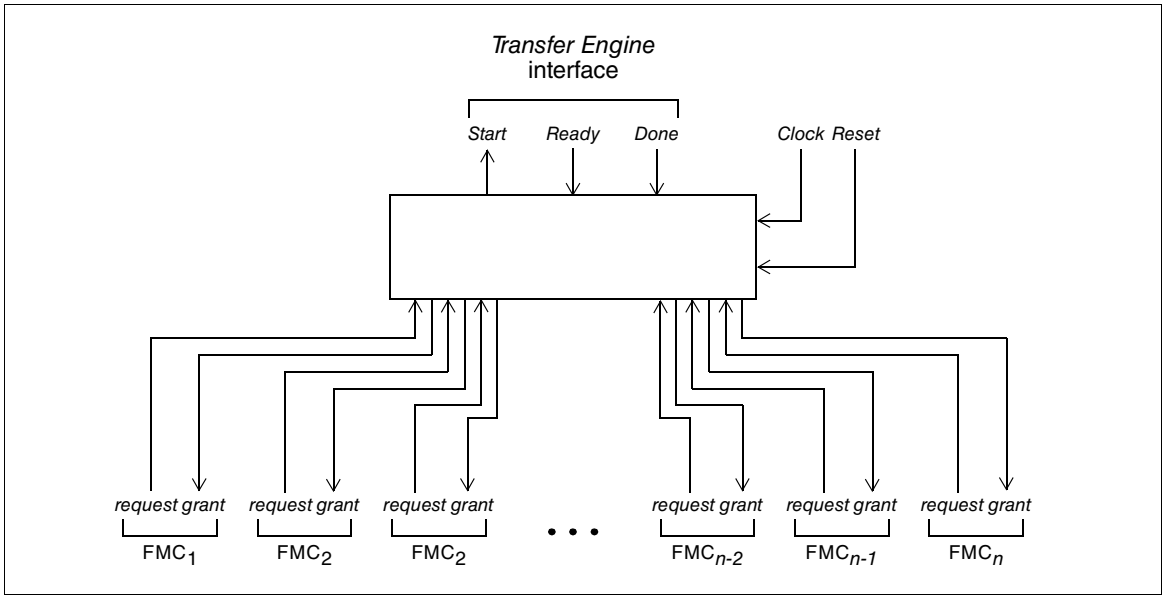

Figure 11 Arbiter Interfaces

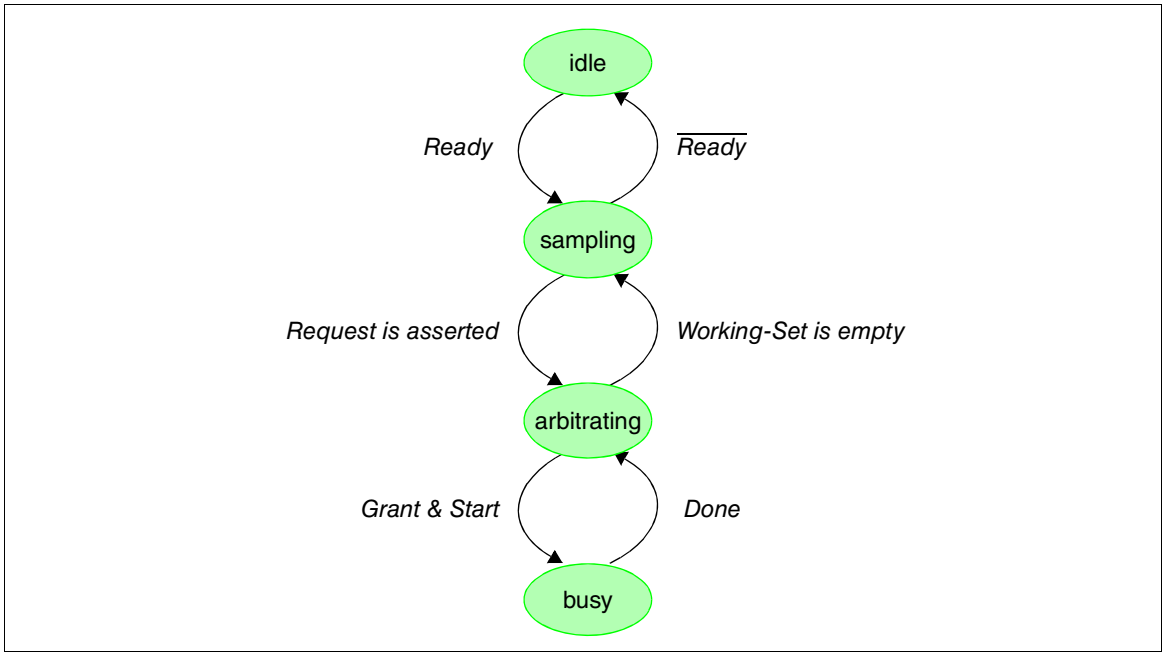

Figure 12 Arbitration State Machine

Asserted *request* signals are ignored by the arbitrator while it is busy. The flow of work through the arbitrator is as follows:

- A FMC spontaneously requests the services of an engine by asserting its *request* signal.
- Whenever the engine is ready and when one or more FMC *request* signals are asserted, the arbitrator is carried from its "idle" to its "sampling" state.
- In its "sampling" state, the arbitrator samples the state of all request signals, harvesting the set of *asserted* signals. This set is called the arbitrator's *working* set. If the working set is *empty*, the arbitrator is carried back to its "idle" state. If the working set is *not* empty, the arbitrator passes from its "sampling" to its "arbitrating" state.
- In its "arbitrating" state, the arbitrator first determines whether its working set is *empty*. If the working set is empty, the arbitrator is carried back to its "sampling" state. If the working set is *not* empty, the arbitrator picks one member *randomly* from the set. It strikes this member off the set and asserts the *grant* signal to the corresponding FMC. This carries the arbitrator from its "arbitrating" to its "busy" state.
- After the granted FMC finishes its transfer it de-asserts its *pending* signal. Typically, this results in the engine becoming *not busy*. However, if the engine in partnership with the transfer engine cannot keep up, the transfer engine's FIFOs could back up and even though the transaction between FMC and transfer engine is complete, the engine may continue to assert busy. In any case, once the transfer engine goes not busy, the arbitrator is carried from its "busy" state back to its "arbitrating" state.

1.6.1 Arbiter Attributes

TBD

1.6.2 Arbiter Resets

TBD

1.7 Performance

Ignore this section. Its a work in progress and is not correct.

The design is targeted to sustain an access rate of up to 100 Megabytes/sec (either read or write) and an erase rate of up to 500 Megabytes/sec.

To understand how well such a box could perform, assume data is distributed randomly within the box and performance is measured as the time to read one sector (512 bytes) of data and present this data at

the external interface. Call this operation a transaction. Once inside the PCI bus, for any one FMC in isolation, the transaction time has the following components:

i. queue and start transaction. Estimated at 200 *ns*.

ii. fetch time from flash cell to flash cache estimated at 20 *us*.

iii. transfer time on the PCI bus. Estimated at 512/4 x 30 ns = 4 *us*.

iv. Reed-Solomon decode. Estimated at 1 *us*.

v. process completion interrupt. Estimated at 200 *ns*.

Naively this corresponds to a transaction time of around 25.4 us. However, each FMC can perform up to 32 operations concurrently which allows flash transfer time to effectively overlap flash fetch time. Therefore, the irreducible transaction time is more like 5.4 us. As the box has 16 buses which operate essentially in parallel, this gives a transaction rate of around 3 million transactions/second. If access is random, there is some real possibility of achieving this number. Of course, this number does not take into account software, interface and management overhead, as well as the fact that any one transaction may require more then one sector of data. I. e., your mileage may vary.

Chapter 2
The Initiator Interface

2.1 Conventions

The interface consists of a number of signals and *ports*. Any entity which accesses the interface is called an *initiator*. A port consists of a data register and the signals necessary for the initiator to access that register. The width of a data register is specified in *bits* and register width is port dependent. The contents of a data register are specified in units of *field*, where a field is specified as (bit) offset and length within the corresponding register. The behaviour of a field fits into one of three classes:

Not defined: Undefined fields are identified as Must Be Zero (MBZ) and are illustrated *greyed out*. An MBZ field will:

- Read back as *zero*
- Ignore writes
- Reset to *zero*

Read/Write: A *Reset* will set a read/write field to *zero*.

Read-only: Read-only fields are illustrated *lightly* grayed-out along with their value. Any *read-only* field will:

- Ignore writes
- Reset to *zero*, unless otherwise documented

Any field used as a boolean has a width of one bit. A value of one (1) is used to indicate its *set* or *true* sense and a value of *zero* (0) to indicate its *clear* or *false* sense. Field numbering for registers is such that offset *zero* (0) corresponds to a register's Least Significant Bit (LSB) and offset *width - 1* is the register's Most Significant Bit (MSB).

Note: The signal descriptions given below assume that the device enable for the FMC is asserted.

2.2 Initiator Interface

TBD. The signal definitions for this port are specified in Table 6:

Table 6 Signal definitions for the *Initiator* interface.

Signal name	I/O	Description
$\overline{\text{ENW}}$	In	The interface write enable. An active low signal that enables writing to the interface on the rising edge of CKW while $\overline{\text{ENW}}$ is active.
CKW	In	The interface write clock. The rising edge clocks data into both CMND and PARM while $\overline{\text{ENW}}$ is active. On the rising edge this signal also updates the FULL and ALMOST_FULL flags (see below).
CMND[0:31]	In	Command data inputs. Inputs are sampled on the rising edge of CKW while $\overline{\text{ENW}}$ is active. See Section 1.4.1 for the specification of this structure of the data written to this port.
PARM[0:31]	In	Parameter data Inputs. Inputs are sampled on the rising edge of CKW while $\overline{\text{ENW}}$ is active. See Section 1.4.1 for the specification of this structure of the data written to this port.
ALMOST_FULL	Out	FMC's command FIFO *Almost Full* flag. This signal remains asserted while this condition remains true. T The definition of this signal is an attribute of the FMC. See xxx for a description of the timing of this signal with respect to changes in the FIFO's state.
FULL	Out	FMC's command FIFO *Full* flag. This signal remains asserted while this condition remains true. See xxx for a description of the timing of this signal with respect to changes in the FIFO's state.

2.3 Timing

TBD.

Chapter 3
Initiator Commands

3.1 Conventions

The interface consists of a number of separately addressable *Ports*. Any entity which accesses the interface is called an *initiator*. A port consists of a data register and the signals necessary for the initiator to access that register. The width of a data register is specified in *bits* and register width is port dependent. The contents of a data register are specified in units of *field*, where a field is specified as (bit) offset and length within the corresponding register. The behaviour of a field fits into one of three classes:

Not defined: Undefined fields are identified as Must Be Zero (MBZ) and are illustrated *greyed out*. An MBZ field will:

- Read back as *zero*
- Ignore writes
- Reset to *zero*

Read/Write: A *Reset* will set a read/write field to *zero*.

Read-only: Read-only fields are illustrated *lightly* grayed-out along with their value. Any *read-only* field will:

- Ignore writes
- Reset to *zero*, unless otherwise documented

Any field used as a boolean has a width of one bit. A value of one (1) is used to indicate its *set* or *true* sense and a value of *zero* (0) to indicate its *clear* or *false* sense. Field numbering for registers is such that offset *zero* (0) corresponds to a register's Least Significant Bit (LSB) and offset *width - 1* is the register's Most Significant Bit (MSB).

Note: The signal descriptions given below assume that the device enable for the FMC is asserted.

3.2 Get Blocks

Read one to sixty-four Code Blocks worth of information from flash memory. The specification of which blocks are accessed is determined by the parameter whose value is a ***Device address***. See Figure 9 within Section 1.2 for the specification of a device address. Transfer the read data to the ***Remote address*** specified by the command argument. When the transaction is complete, increment the appropriate statistics counters (see Section 3.2.2).

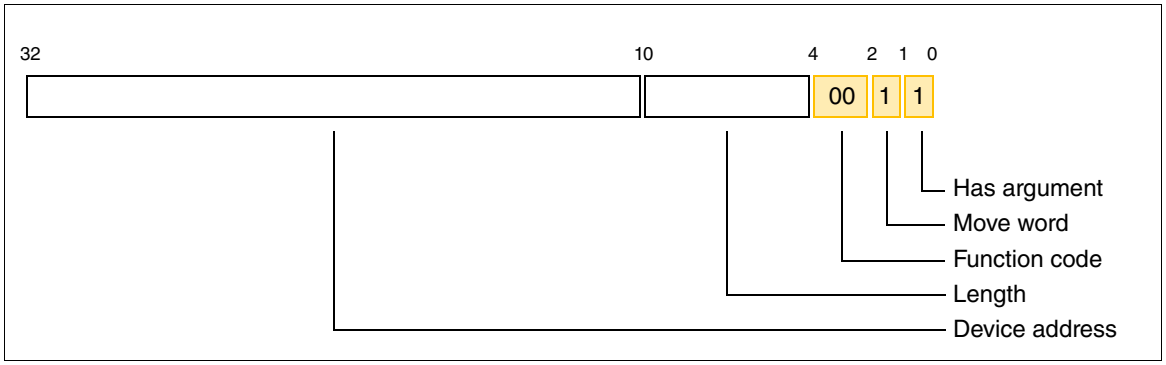

Figure 13 "Get blocks" command

The external interfaces for which this command requires arbitration by the FMC are enumerated in Table 7:

Table 7 External interfaces used by the "Get Blocks" command

Interface	Used?
Flash	*Yes*
Outbound	*Yes*
Inbound	*No*

3.2.1 Argument

The argument for this transaction is the ***Remote address*** which specifies the location where the read information are to be returned. A remote address is simply a 32-bit value whose interpretation is agreed on between *Initiator* and the FMC's Outbound engine.

3.2.2 Performance counters incremented

Each transaction will increment the *reads* counter. See Section 1.5 for more information on these counters and how they may be accessed.

3.3 Set Page

Write one page of information to flash memory. The specification of which page to write is determined by the command parameter whose value is a ***Device address***. See Figure 9 within Section 1.2 for the specification of a device address. Transfer the data to be written from the ***local address*** specified by the command argument. When the transaction is complete, increment the appropriate statistics counters (see Section 3.2.2).

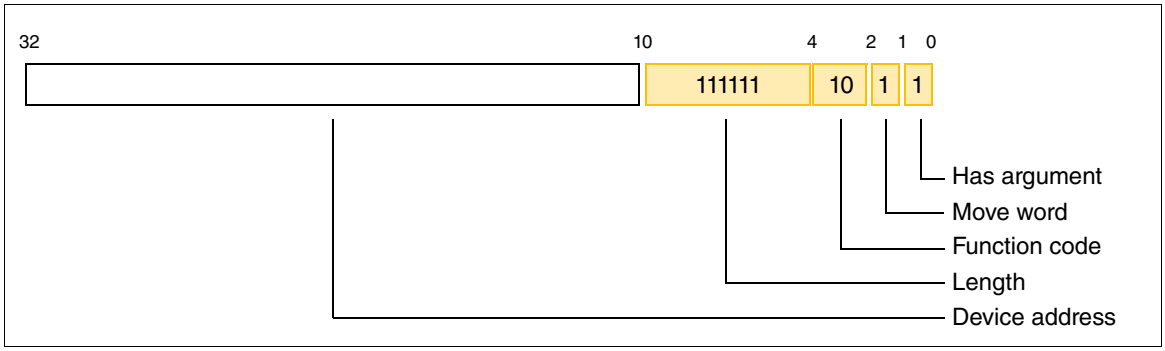

Figure 14 "Set Page" command

The external interfaces for which this command requires arbitration by the FMC are enumerated in Table 8:

Table 8 External interfaces used by the "Set Page" command

Interface	Used?
Flash	*Yes*
Outbound	*No*
Inbound	*Yes*

3.3.1 Argument

The argument for this transaction is the ***Local address*** which specifies the location where the information to be written is found. A local address is simply a 32-bit value whose interpretation is agreed on between Initiator and the FMC's *Inbound* engine.

3.3.2 Performance counters incremented

Each transaction will increment the *write* counter. It may also increment the *device error* counter. See Section 1.5 for more information on these counters and how they may be accessed.

3.4 Move Page

Move one page of information from one location in flash memory to another. The specification of which page to move is determined by the command parameter whose value is a ***Device Address***. The destination address is specified by the command argument whose value is a ***Device Address***. See Figure 9 within Section 1.2 for the specification of a device address. When the transaction is complete, increment the appropriate statistics counters (see Section 3.2.2).

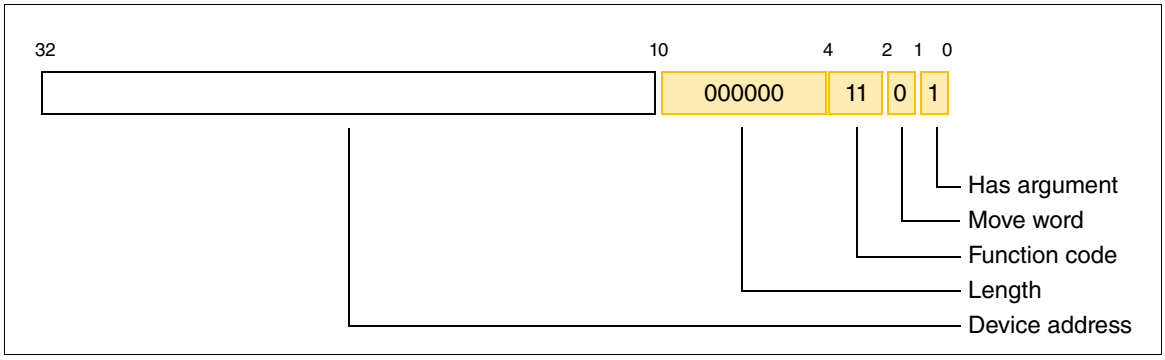

Figure 15 "Move Page" command

The external interfaces for which this command requires arbitration by the FMC are enumerated in Table 9:

Table 9 External interfaces used by the "Move Page" command

Interface	Used?
Flash	*Yes*
Outbound	*Yes*
Inbound	*No*

3.4.1 Argument

The low-order 22 bits of the argument specify the destination address. The high-order 10 bits will be *zero*. The destination address is specified as a ***Device Address***. See Figure 9 within Section 1.2 for the specification of a device address.

3.4.2 Performance counters incremented

Each transaction will increment the *move* counter. It may also increment the *device error* counter. See Section 1.5 for more information on these counters and how they may be accessed.

3.5 Erase Block

Erase one block of information (512 Kilobytes) at the specified device address. When the transaction is complete, increment the appropriate statistics counters (see Section 3.2.2). The specification of which block to erase is determined by the command parameter whose value is a ***Device address***. See Figure 9 within Section 1.2 for the specification of a device address.

Note: For this command, the page field of the device address is ignored and must be MBZ (0).

When the transaction is complete, increment the appropriate statistics counters (see Section 3.2.2).

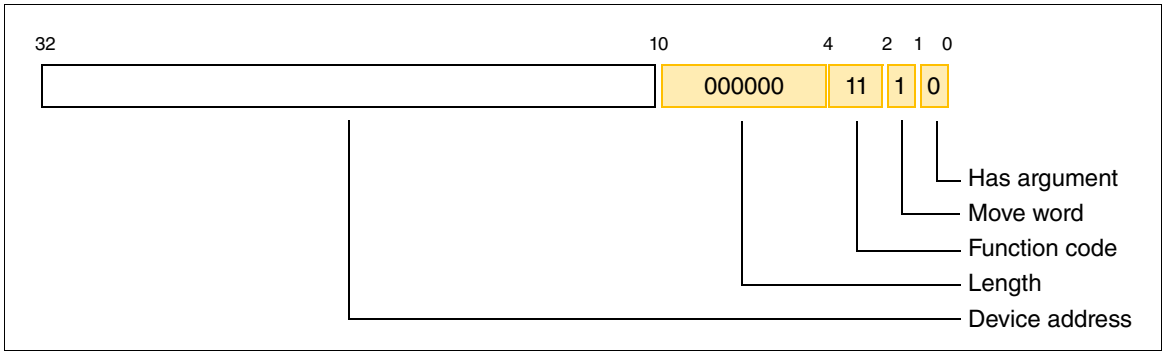

Figure 16 "Erase Block" command

The external interfaces for which this command requires arbitration by the FMC are enumerated in Table 10:

Table 10 External interfaces used by the "Erase Block" command

Interface	Used?
Flash	*Yes*
Outbound	*No*
Inbound	*No*

3.5.1 Argument

None.

3.5.2 Performance counters incremented

Each transaction will increment the *erase* counter. It may also increment the *device error* counter. See Section 1.5 for more information on these counters and how they may be accessed.

3.6 Get Flash Attributes

Read the ID register of one of the four flash devices managed by the FMC. This register is called the *Flash Device Attributes*. The attributes are contained in a word whose structure is described in Section 1.2.2. Return this word to the ***Remote address*** specified by the command argument.

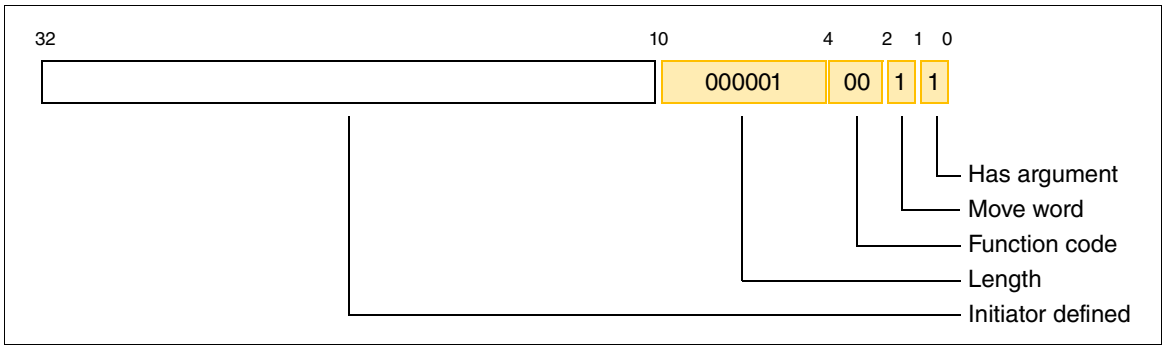

Figure 17 "Get Flash Attributes" command

Usage of the command parameter is completely user determined. Its value is specified by the initiator of the command and is both unused and is uninterpreted by the FMC. The external interfaces for which this command requires arbitration by the FMC are enumerated in Table 11:

Table 11 External interfaces used by the "Get Flash Attributes" command

Interface	Used?
Flash	*Yes*
Outbound	*Yes*
Inbound	*No*

3.6.1 Argument

The argument for this transaction is the ***Remote address*** which specifies the location where the word containing the attributes are to be returned. A remote address is simply a 32-bit value whose interpretation is agreed on between *Initiator* and the FMC's Outbound engine.

3.6.2 Performance counters incremented

None.

3.7 Get FMC Attributes

Sample and return the FMC's attributes. The attributes are contained in a word whose structure is described in Section 1.3.1. Return this word to the ***Remote address*** specified by the command argument.

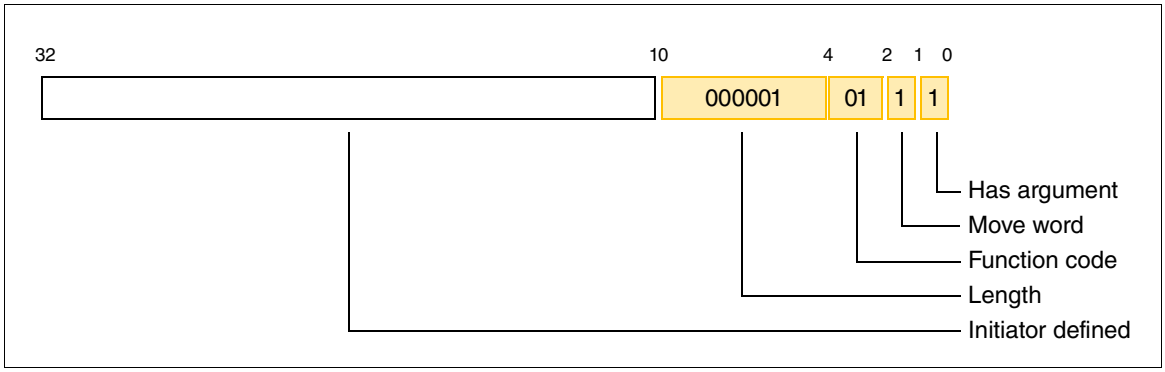

Figure 18 "Get FMC Attributes command

Usage of the command parameter is completely user determined. Its value is specified by the initiator of the command and is both unused and is uninterpreted by the FMC. The external interfaces for which this command requires arbitration by the FMC are enumerated in Table 12:

Table 12 External interfaces used by the "Get FMC Attributes" command

Interface	Used?
Flash	*No*
Outbound	*Yes*
Inbound	*No*

3.7.1 Argument

The argument for this transaction is the ***Remote address*** which specifies the location where the word containing the attributes are to be returned. A remote address is simply a 32-bit value whose interpretation is agreed on between *Initiator* and the FMC's Outbound engine.

3.7.2 Performance counters incremented

None.

3.8 Get Counter

Sample the value of a specified performance counter. Return the value of the read counter to the ***Remote address*** specified by the command argument. The command's parameter field contains a small value which enumerates which particular counter to sample and return. The correspondence between value and counter is described in Section 1.5.

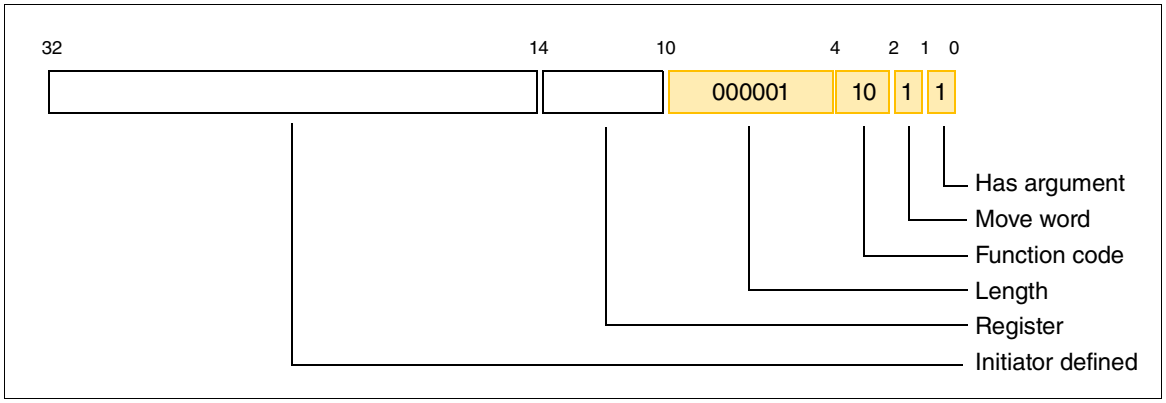

Figure 19 "Get Counter" command

Usage of the remainder of the command parameter is completely user determined. Its value is specified by the initiator of the command and is both unused and is uninterpreted by the FMC. The external interfaces for which this command requires arbitration by the FMC are enumerated in Table 13*:*

Table 13 External interfaces used by the "Get Counter" command

Interface	Used?
Flash	*No*
Outbound	*Yes*
Inbound	*No*

3.8.1 Argument

The argument for this transaction is the ***Remote address*** which specifies the location where the word containing the read counter is to be returned. A remote address is simply a 32-bit value whose interpretation is agreed on between *Initiator* and the FMC's Outbound engine.

Chapter 4
The Transfer Interface

4.1 Conventions

The interface consists of a number of separately addressable *Ports*. Any entity which accesses the interface is called a *transfer engine*. A port consists of a data register and the signals necessary for the transfer engine to access that register. The width of a data register is specified in *bits* and data register width is port dependent. The contents of a data register are specified in units of *field*, where a field is specified as (bit) offset and length within the corresponding register. The behaviour of a field fits into one of three classes:

Not defined: Undefined fields are identified as Must Be Zero (MBZ) and are illustrated *greyed out*. An MBZ field will:

- Read back as *zero*
- Ignore writes
- Reset to *zero*

Read/Write: A *Reset* will set a read/write field to *zero*.

Read-only: Read-only fields are illustrated *lightly* grayed-out along with their value. Any *read-only* field will:

- Ignore writes
- Reset to *zero*, unless otherwise documented

Any field used as a boolean has a width of one bit. A value of one (1) is used to indicate its *set* or *true* sense and a value of *zero* (0) to indicate its *clear* or *false* sense. Field numbering for

registers is such that offset *zero* (0) corresponds to a register's Least Significant Bit (LSB) and offset *width - 1* is the register's Most Significant Bit (MSB).

Note: The signal descriptions given below assume that the device enable for the FMC is asserted.

4.2 Transaction Interface

TBD.

Table 14 Signal definition for the Transaction interface.

Signal name	I/O	Description
CMND[0:31]	Out	The transaction command. The specification of a command is found in Section 1.4.1. Its value becomes valid at the same time RQST is asserted (see above) and remains valid for the duration of the transaction.
ARG[0:31]	Out	The transaction command argument. The specification of an argument is found in xxx. Its value becomes valid at the same time RQST is asserted (see above) and remains valid for the duration of the transaction.

4.3 Inbound Interface

The signals of this interface are connected to both the inbound arbiter and transfer engine. The FMC uses this interface for each transaction which requires moving data *to* the FMC. See, for example, the ***Set Page*** transaction described in Section 3.3 which *writes* code blocks to the FMC's flash memory. The FMC signifies its interest in using the transfer engine by asserting its RQST signal. This signal is asserted once, and only once, for each transaction. The specification of the transaction corresponding to the request is found in the ***Transaction Interface*** described in Section 4.2. In particular, the tenure of any one transaction on this interface is determined by the values of the ***Move Word*** and ***Length*** fields of the command registered within the CMND port (see Section 1.4.1).

Table 15 Signal definition for the Inbound interface.

Signal name	I/O	Description
RQST	Out	This signal is asserted by the FMC whenever it requires the services of its corresponding inbound transfer engine. This signal is connected to the inbound *Arbiter* (see Section 1.6). It remains asserted until the GRNT signal (see below) is asserted by the arbiter. Once granted (see below), the FMC will not re-assert this signal until the transfer corresponding to the request has completed. The signals on the ***Transaction Interface*** (see Section 4.2) become valid when RQST is asserted and remain valid until the transfer is finished.
GRNT	In	This signal is asserted for one clock by the inbound *Arbiter* (see Section 1.6) in response to the FMC's assertion of its RQST signal (see above). It grants the FMC the services of its corresponding inbound transfer engine for one transfer. This signal is connected to the inbound arbiter. In response to receiving this signal the FMC will de-assert its RQST signal (see above). If the arbiter does not assert GRNT within a specified amount of time, the FMC will abort the outstanding request.
SOB	In	Start of Code Block or Word. This signal is asserted for one clock for each block or word in the transfer. It is always asserted at the beginning of the block. It specifies that either the word or first four symbols of any one block are valid (see Section 1.4.3 for a discussion of transfer structure). The value of either a word or four symbols of a code block are presented on the DIN port described below.
DIN[0:31]	In	Either the value of a word or the value of the "next" four symbols of a code block. A word or the first four symbols of a code block are registered on this port whenever the SOB signal is asserted (see above). See Section 1.4.3 for a discussion of transfer blocks.
ERR	Out	Specifies whether or not the transaction completed successfully. This signal is asserted for one clock at the end of the transfer.

4.4 Outbound Interface

The signals of this interface are connected to both the outbound arbiter and transfer engine. The FMC uses this interface for each transaction which requires moving data *from* the FMC. See, for example, the ***Get Blocks*** transaction described in Section 3.2 which *reads* code blocks from the FMC's flash memory. The FMC signifies its interest in using the transfer engine by asserting its RQST signal. This signal is asserted once, and only once, for each transaction. The specification of the transaction corresponding to the request is found in the ***Transaction Interface*** described in Section 4.2. In particular, the tenure of any one transaction on this interface is determined by the values of the ***Move Word*** and ***Length*** fields of the command registered within the CMND port (see Section 1.4.1).

Table 16 Signal definition for the Outbound interface.

Signal name	I/O	Description
RQST	Out	This signal is asserted by the FMC whenever it requires the services of its corresponding outbound transfer engine. This signal is connected to the outbound *Arbiter* (see Section 1.6). It remains asserted until the GRNT signal (see below) is asserted by the arbiter. Once granted (see below), the FMC will not re-assert this signal until the transfer corresponding to the request has completed. The signals on the ***Transaction Interface*** (see Section 4.2) become valid when RQST is asserted and remain valid until the transfer is finished.
GRNT	In	This signal is asserted for one clock by the outbound *Arbiter* (see Section 1.6) in response to the FMC's assertion of its RQST signal (see above). It grants the FMC the services of its corresponding outbound transfer engine for one transfer. This signal is connected to the outbound arbiter. In response to receiving this signal the FMC will de-assert its RQST signal (see above). If the arbiter does not assert GRNT within a specified amount of time, the FMC will abort the outstanding request.
SOB	Out	Start of Code Block or Word. This signal is asserted for one clock for each block or word in the transfer. It is always asserted at the beginning of the block. It specifies that either the word or first four symbols of any one block are valid (see Section 1.4.3 for a discussion of transfer structure). The value of either a word or four symbols of a code block are presented on the DOUT port described below.
DOUT[0:31]	Out	Either the value of a word or the value of the "next" four symbols of a code block. A word or the first four symbols of a code block are registered on this port whenever the SOB signal is asserted (see above). See Section 1.4.3 for a discussion of transfer blocks.
ERR	Out	Specifies whether or not the transaction completed successfully. This signal is asserted for one clock at the end of the transfer.

4.5 Transfer Timing

Relative timing is independent of whether or not the transfer is inbound or outbound. It is however, dependent on the length and type of transfer. The length and type of the transfer are determined by the values of the ***Move Word*** and ***Length*** fields of the command registered within the CMND port (see Section 1.4.1).There are three cases to consider, each of which are described below. Note that each example (for pedagogical reasons) shows the ERR flag asserted. Of course, in reality, this flag will be asserted at most infrequently and hopefully not at all.

4.5.1 Transfer one or more code blocks

Code block transfers are done by both the inbound and outbound interfaces. As an example, see the ***Set Page*** transaction discussed in Section 3.3. The number of code blocks to transfer is determined by the ***Length*** field of the transaction's command. Each code block takes thirty-three (33) clock cycles to transfer. The first 32 clocks provide the 128 data symbols for the block and the last clock provides its four check symbols. An example, showing the transfer of three code blocks, is illustrated in Figure 20:

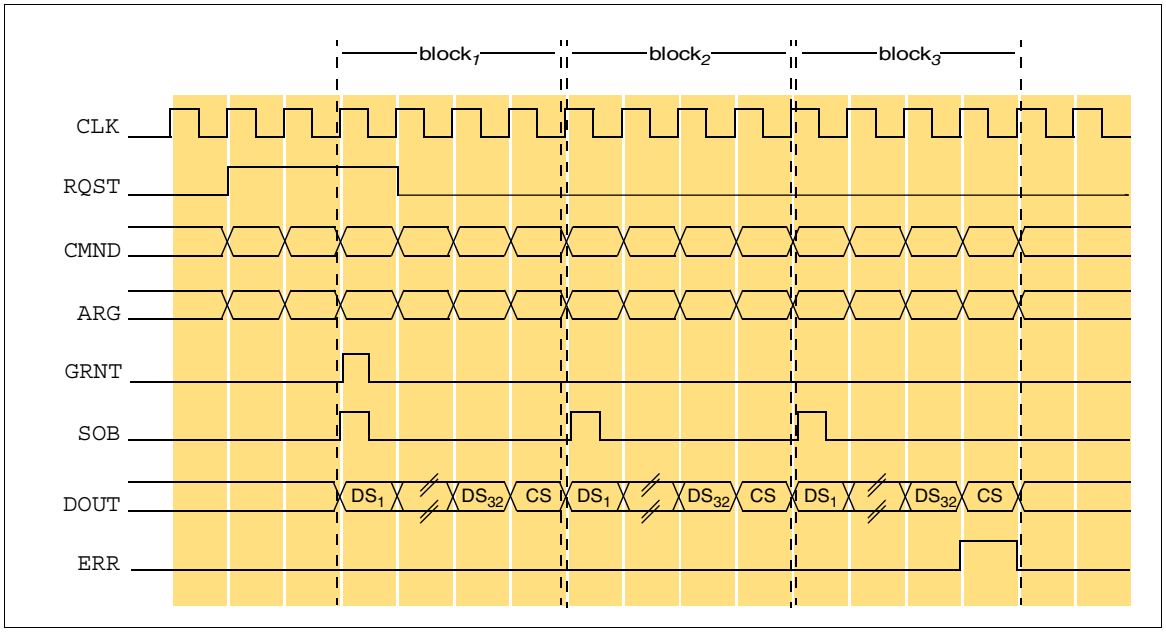

Figure 20 Timing diagram of code block transfer

4.5.2 Transfer one word

Word transfers are currently done by only the outbound interface (but, this could change). As an example, see the ***Get Counter*** transaction discussed in Section 3.8. The number of words to transfer is determined by the ***Length*** field of the transaction's command. An example is illustrated in Figure 21:

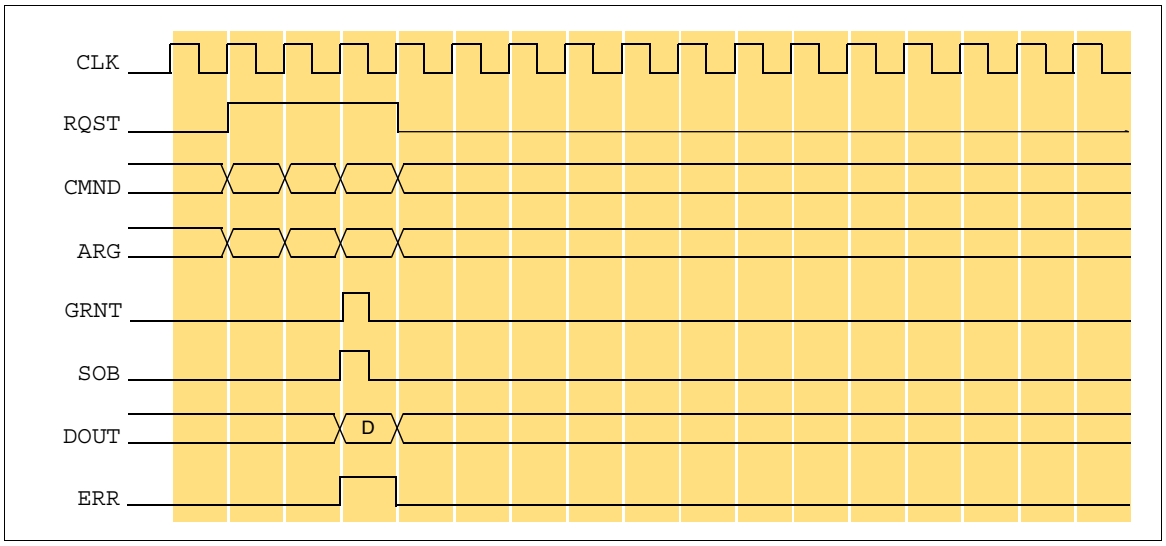

Figure 21 Timing diagram for word transfer

4.5.3 Zero-Length Transfers

Zero-Length transfers are currently done by only the outbound interface. As an example, see the ***Erase Block*** transaction discussed in Section 3.5. For transactions of this type the ***Move Word*** field of the command will be *asserted* and the ***Length*** field will be *zero*. An example is illustrated in Figure 22:

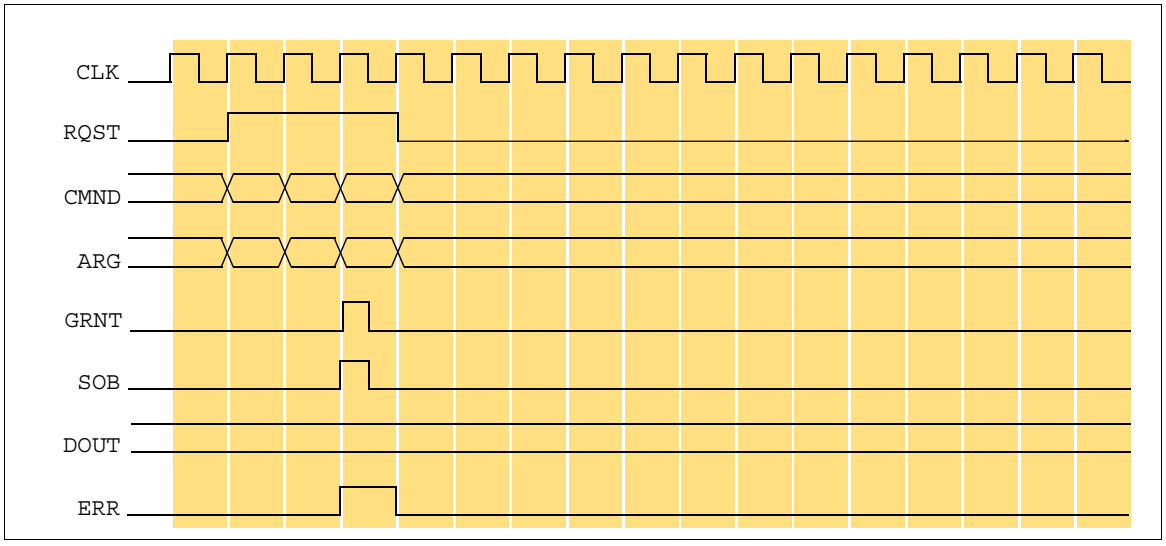

Figure 22 Timing diagram for zero length transfer

Chapter 5
The Arbiter Interface

5.1 Conventions

The interface consists of a number of separately addressable *Ports*. Any entity which accesses the interface is called a *transfer engine*. A port consists of a data register and the signals necessary for the transfer engine to access that register. The width of a data register is specified in *bits* and data register width is port dependent. The contents of a data register are specified in units of *field*, where a field is specified as (bit) offset and length within the corresponding register. The behaviour of a field fits into one of three classes:

Not defined: Undefined fields are identified as Must Be Zero (MBZ) and are illustrated *greyed out*. An MBZ field will:

- Read back as *zero*
- Ignore writes
- Reset to *zero*

Read/Write: A *Reset* will set a read/write field to *zero*.

Read-only: Read-only fields are illustrated *lightly* grayed-out along with their value. Any *read-only* field will:

- Ignore writes
- Reset to *zero*, unless otherwise documented

Any field used as a boolean has a width of one bit. A value of one (1) is used to indicate its *set* or *true* sense and a value of *zero* (0) to indicate its *clear* or *false* sense. Field numbering for

registers is such that offset *zero* (0) corresponds to a register's Least Significant Bit (LSB) and offset *width - 1* is the register's Most Significant Bit (MSB).

Note: The signal descriptions given below assume that the device enable for the FMC is asserted.

5.2 Transfer Engine Interface

The signals of this interface are connected to both the inbound arbiter and transfer engine. The FMC uses this interface for each transaction which requires moving data *to* the FMC. See, for example, the ***Set Page*** transaction described in Section 3.3 which *writes* code blocks to the FMC's flash memory. The FMC signifies its interest in using the transfer engine by asserting its RQST signal. This signal is asserted once, and only once, for each transaction. The specification of the transaction corresponding to the request is found in the ***Transaction Interface*** described in xxx. In particular, the tenure of any one transaction on this interface is determined by the values of the ***Move Word*** and ***Length*** fields of the command registered within the CMND port (see Section 1.4.1).

Table 17 Signal definition for the Arbiter Transfer Engine interface.

Signal name	I/O	Description
READY	In	This signal is asserted for one clock by the *Transfer Engine* in response to the FMC's assertion of its RQST signal (see above). It grants the FMC the services of its corresponding inbound transfer engine for one transfer. This signal is connected to the inbound arbiter. In response to receiving this signal the FMC will de-assert its RQST signal (see above). If the arbiter does not assert GRNT within a specified amount of time, the FMC will abort the outstanding request.
START	Out	This signal is asserted by the *Arbiter* whenever it requires the services of its corresponding inbound transfer engine. This signal is connected to the inbound *Arbiter*. It remains asserted until the GRNT signal (see below) is asserted by the arbiter. Once granted (see below), the FMC will not re-assert this signal until the transfer corresponding to the request has completed. The signals on the ***Transaction Interface*** (see xxx) become valid when RQST is asserted and remain valid until the transfer is finished.
DONE	In	This signal is asserted for one clock by the *Transfer Engine* in response to the FMC's assertion of its RQST signal (see above). It grants the FMC the services of its corresponding inbound transfer engine for one transfer. This signal is connected to the inbound arbiter. In response to receiving this signal the FMC will de-assert its RQST signal (see above). If the arbiter does not assert GRNT within a specified amount of time, the FMC will abort the outstanding request.

5.3 FMC Interface

The signals of this interface are connected to both the inbound arbiter and transfer engine. The FMC uses this interface for each transaction which requires moving data *to* the FMC. See, for example, the ***Set Page*** transaction described in Section 3.3 which *writes* code blocks to the FMC's flash memory. The FMC signifies its interest in using the transfer engine by asserting its RQST signal. This signal is asserted once, and only once, for each transaction. The specification of the transaction corresponding to the request is found in the ***Transaction Interface*** described in xxx. In particular, the tenure of any one transaction on this interface is determined by the values of the ***Move Word*** and ***Length*** fields of the command registered within the CMND port (see Section 1.4.1).

Table 18 Signal definition for the Arbiter FMC interface.

Signal name	I/O	Description
RQST _n	Out	This signal is asserted by the FMC whenever it requires the services of its corresponding inbound transfer engine. This signal is connected to the inbound *Arbiter* (see Section 1.6). It remains asserted until the GRNT signal (see below) is asserted by the arbiter. Once granted (see below), the FMC will not re-assert this signal until the transfer corresponding to the request has completed. The signals on the ***Transaction Interface*** (see xxx) become valid when RQST is asserted and remain valid until the transfer is finished.
GRNT_n	In	This signal is asserted for one clock by the inbound *Arbiter* (see Section 1.6) in response to the FMC's assertion of its RQST signal (see above). It grants the FMC the services of its corresponding inbound transfer engine for one transfer. This signal is connected to the inbound arbiter. In response to receiving this signal the FMC will de-assert its RQST signal (see above). If the arbiter does not assert GRNT within a specified amount of time, the FMC will abort the outstanding request.

5.4 Arbiter Timing

Relative timing is independent of whether or not the transfer is inbound or outbound. It is however, dependent on the length and type of transfer. The length and type of the transfer are determined by the values of the ***Move Word*** and ***Length*** fields of the command registered within the CMND port (see Section 1.4.1).There are three cases to consider, each of which are described below. Note that each example (for pedagogical reasons) shows the ERR flag asserted. Of course, in reality, this flag will be asserted at most infrequently and hopefully not at all.

5.4.1 Transfer one or more code blocks

Code block transfers are done by both the inbound and outbound interfaces. As an example, see the ***Set Page*** transaction discussed in Section 3.3. The number of code blocks to transfer is determined by the ***Length*** field of the transaction's command. Each code block takes thirty-three (33) clock cycles to transfer. The first 32 clocks provide the 128 data symbols for the block and the last clock provides its four check symbols. An example, showing the transfer of three code blocks, is illustrated in Figure 23:

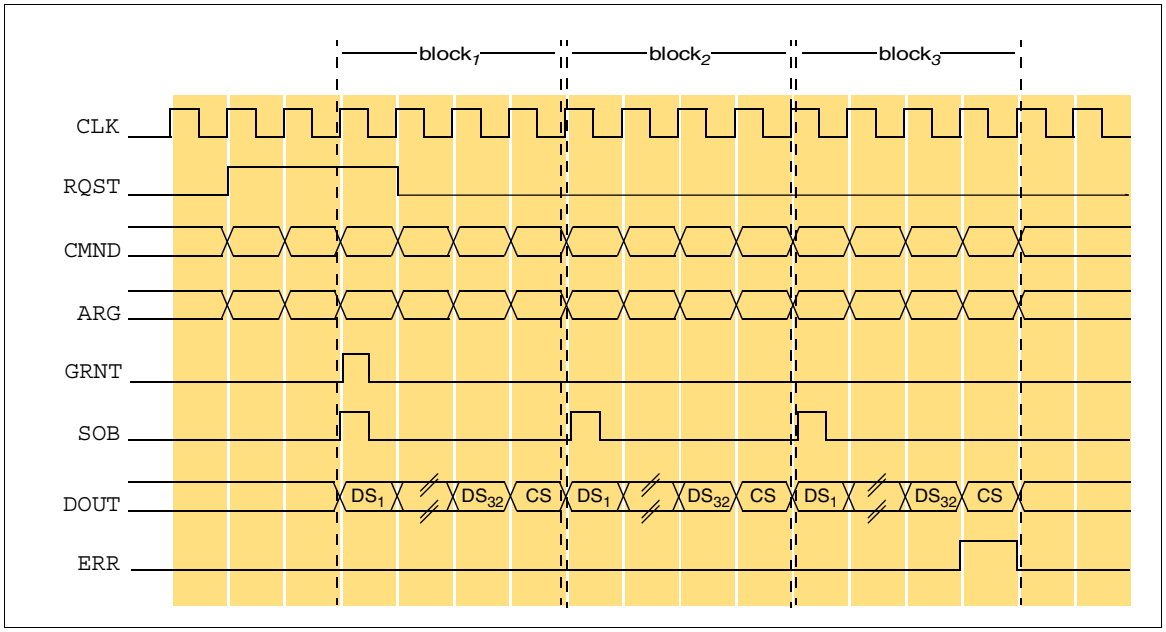

Figure 23 Timing diagram of code block transfer