intel

CHAPTER 3
FDI ARCHITECTURE AND API SPECIFICATION

3.1 INTRODUCTION

3.1.1 Scope

This chapter provides a detailed design description of the Flash Data Integrator (FDI) which enables code plus data storage in a single flash component.

3.1.2 Purpose

The purpose of this chapter is to provide a general and detailed design description of the FDI.

3.1.3 System Overview

Many of today's systems use flash for code storage and execution. These same systems use EEPROMs to provide nonvolatile memory for system data and parameter storage even though the flash device often has unused space. The FDI code plus data solution removes the EEPROM from the system, thus reducing board space and product cost.

FDI allows the storage of data, and the execution of code from the same flash device. FDI also allows for future code updates, upgrades, and extensions in flash. While the current FDI effort is GSM-centric, many different system architectures can utilize the FDI software.

Figure 3-1 provides a highly simplified diagram of a system software architecture which is a good candidate for FDI. In this system, many tasks, such as protocol control, run on top of a real- time operating system kernel. An EEPROM data storage request may be prompted by a user pressing a key or a service routine which generates a system interrupt. An EEPROM manager task handles the data storage request, and may queue the data in RAM until time is available to write the data to EEPROM.

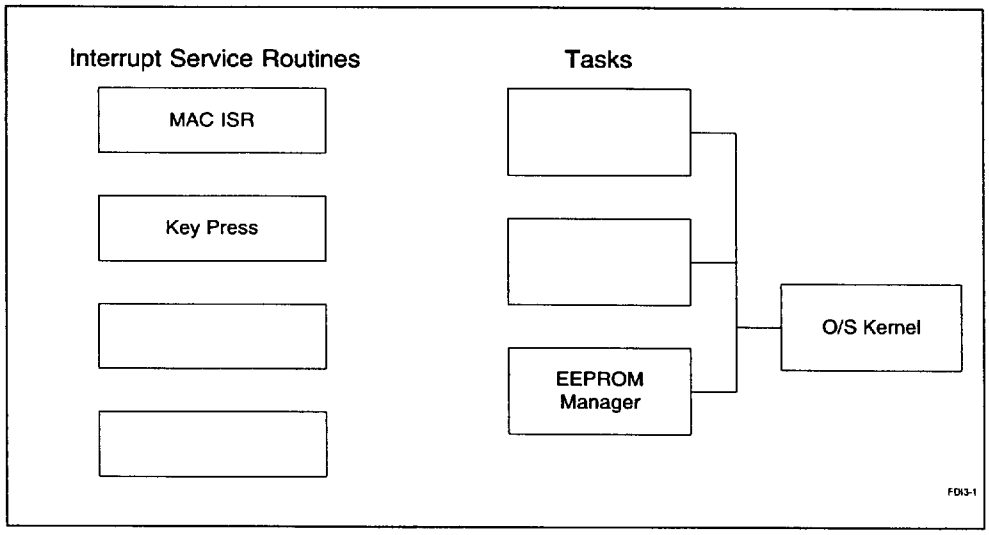

Figure 3-1. Simplified System Software Architecture

FDI is a complete flash media manager which includes an EEPROM manager type API. FDI handles all aspects of storing and retrieving variable length parameters into flash memory. FDI allows system designers to remove EEPROM and use existing flash for parameter data and code storage.

Intel's Advanced Boot Block Flash Memory products have been designed to maximize data efficiency when using FDI. FDI takes advantage of Advanced Boot Block features including small data blocks and program/erase suspend features.

3.1.4 Document Overview

The General Description in Section 3.3 contains the FDI APIs, and general implementation considerations.

The Detailed Design in Section 3.4 contains the FDI software requirements to a level of detail sufficient to enable system and firmware designers to design a system with FDI as a component of their system.

3.2 FLASH DATA INTEGRATOR REFERENCES

3.2.1 Glossary

3.2.1.1 DEFINITIONS

Bitmask	Set of bits
BYTE	8-bit value
DWORD	32-bit value
Granularity	Minimum allocation unit size
Init	Initialization
NIBBLE	4-bit value
NULL	Zero
Unit	A section of a flash block taking up one or more granular sizes
WORD	16-bit value
data parameters	Information such as system variables, small arrays, etc.
data streams	Data such as SMS, voice messages, large arrays, etc.
data fragment	A unit containing one of multiple data pieces of a data parameter or stream
Instance	One occurrence of a data parameter in a unit which can hold multiple occurrences of the data parameter

3.2.1.2 ACRONYMS

APC	Advanced Personal Communication
API	Application Program Interface
EEPROM	Electrically Erasable Programmable Read Only Memory
FDI	Flash Data Integrator
GSM	Global System for Mobile communications
ISR	Interrupt Service Routines
RAM	Random Access Memory

3.2.1.3 ABBREVIATIONS

blk	block
blknum	block number

3.2.2 References

European Digital Cellular Telecommunications System (Phase 2); Specification of the Subscriber Identity Module–Mobile Equipment (SIM–ME) Interface (GSM 11.11) ETS 300 608 January 1995.

3.3 FLASH DATA INTEGRATOR GENERAL DESCRIPTION

3.3.1 Flash Data Integrator Product Perspective

The Flash Data Integrator (FDI) enables embedded systems to use flash memory for code storage and execution as well as data storage FDI will replace EEPROM management software in current systems. Figure 3-2 provides an overview of the software components necessary to accomplish this and how these components interact with the existing system software.

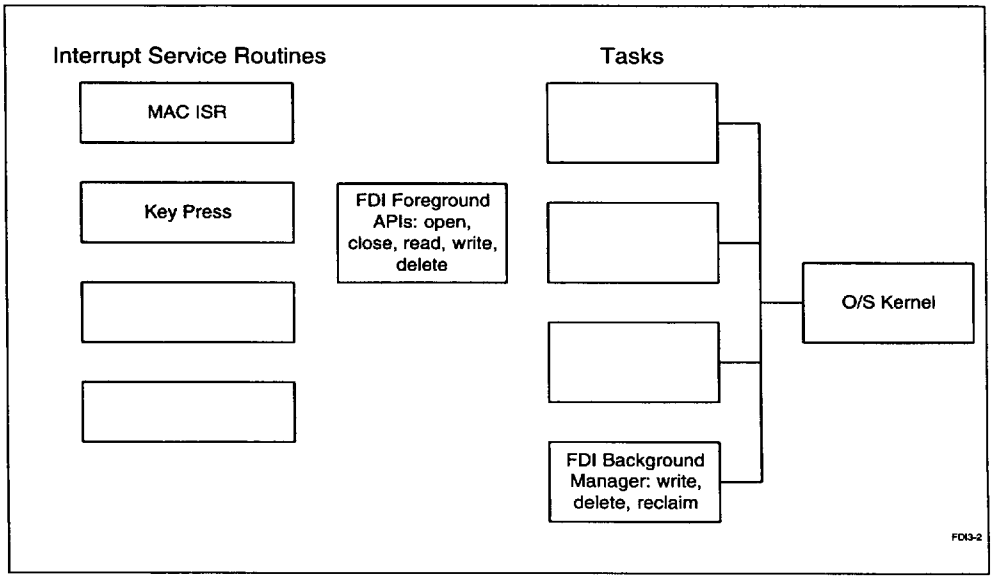

Figure 3-2. Simplified System Software Architecture Using FDI

All tasks and interrupts that need to store data into the flash, interface to functions provided in the FDI Foreground API. These functions (such as open/close/read/write) allow the command and corresponding data to be queued in memory for the FDI Background Manager.

The FDI Background Manager is responsible for executing pending data writes. When a write is queued, and background processing time is available, the Background Manager manages updating or creation of data in flash. The Background Manager also manages any reclaim of invalid (deleted) data areas in flash for reuse.

The FDI Background Manager utilizes a small low level flash erase and programming routine which resides in RAM. This low level routine responds to interrupts that occur during the flash program/erase times by suspending the program/erase, and allowing the interrupt to then be executed from flash. Worst case latencies from program and erase suspend are 7 µs and 20 µs respectively.

FDI implements robust power loss recovery mechanisms to protect the valuable data stored in flash media.

Figure 3-3 provides a diagram of the information flow between flash and RAM. The system calls the Foreground API function (1), with a command and data. The Foreground API function either, stuffs the command and data into a RAM queue for operations which modify flash (1a), or executes the command directly for commands which do not modify flash (1b). The Background Manager (2), executing out of flash, manages the queued tasks during available processor time. During a flash write or erase, interrupts with vectors in flash are disabled and control is turned over to a small routine in RAM (3). This routine polls interrupts while monitoring progress of the program or erase operation. If a higher priority interrupt occurs, the polling routine suspends the program or erase operation, and allows the interrupt handler to then execute from flash. Upon completion of the interrupt routine, the flash program or erase operation is resumed by the RAM polling routine.

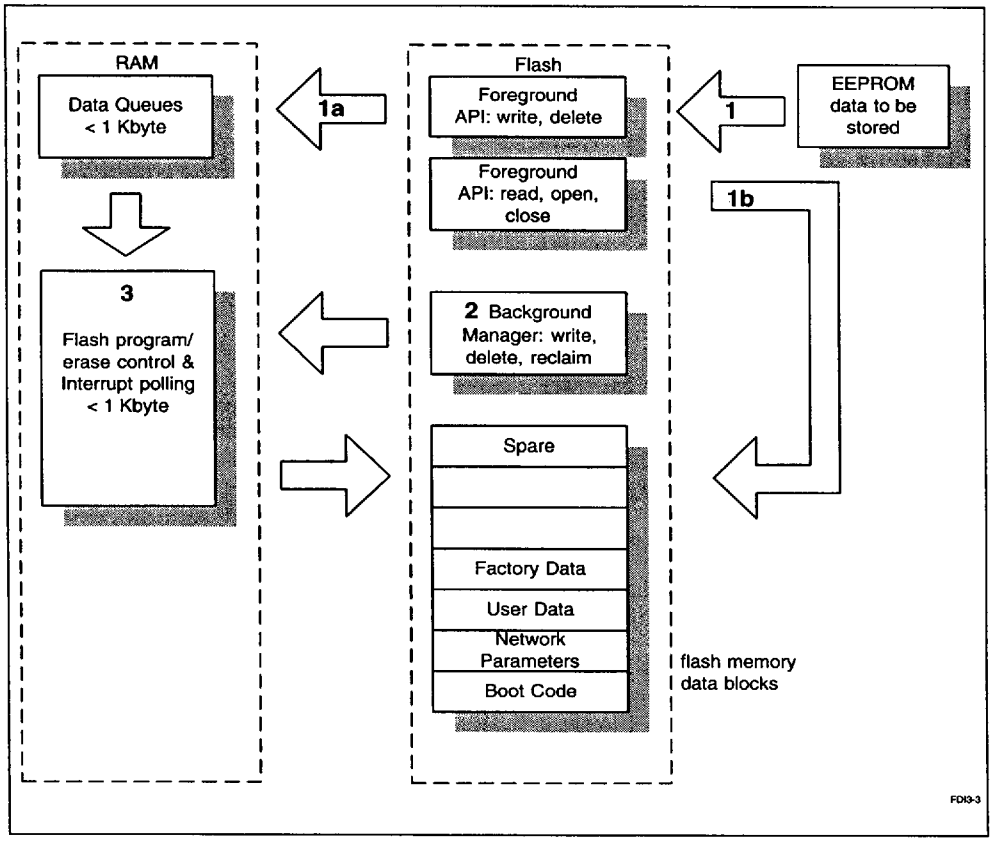

Figure 3-3. Software Flash Data Integrator Data Flow Diagram

3.3.2 Foreground APIs

The Foreground API functions receive storage and read commands from other tasks in the system The system calls the Foreground API function with a command and data. The Foreground API function then either stuffs the command and data into a RAM queue for operations which modify flash, or executes the command directly for commands which do not modify flash. The Foreground API functions are the application interface for storing EEPROM data types, factory data, network parameters and user alterable data to the flash media.

3.3.3 Background Manager

The FDI Background Manager controls the actual writes into the flash media. The Background Manager awaits the arrival of items into its Data Queue and then extracts the highest priority item to act upon. When there is CPU time available, the Background Manager reads from the queue and determines the information's location in flash. During flash programming and erase operations the Background Manager disables and polls interrupts. If an interrupt occurs, the Background Manager suspends the program/erase in progress, and then relinquishes the flash to the interrupt task. Once the interrupt handling is complete, the Background Manager continues until it completes, or until interrupted again by other interrupts. The Background Manager resumes the write/erase which then continues in this fashion until the RAM queue is empty.

The Background Manager is also responsible for reclaiming invalid (deleted or superseded) data. Reclamation occurs upon a predetermined free space trigger, or a user-defined percentage (default value of INVALID_PER_BLOCK is 70%) of a block that has been dirtied, or when there is not enough free space to store a new piece of data. The Background Manager asks for permission from the system, and when granted, executes the reclamation process to free up invalid regions of flash memory and makes them available for reuse.

3.3.4 Initialization

The initialization process performs power loss recovery, and initializes all FDI hardware and RAM variables. FDI implements robust power loss recovery mechanisms throughout the code. This safeguards the valuable data stored in the flash media. By utilizing the unique characteristics of flash media, the integrity of the existing data can be assured if power fails while writing to flash.

The worst case power loss situations for FDI are:

- All data in the RAM queue (not yet written to flash) will be lost if a power failure occurs.
- Any operation in progress is considered not done.

If a power loss occurs during the operation, the partial data that has been written prior to power loss is discarded. A reclamation in progress is identified and completed during the initialization process at the next power on. Refer to the *Power Loss Recovery Process* section below.

3.3.4.1 POWER LOSS RECOVERY PROCESS

Power loss recovery is done during initialization to guarantee all internal structures and data are in a valid state. To validate all blocks, power loss recovery reads internal structures maintained within each block. If a power loss has occurred during the reclaim of a block, the reclaim is restarted and completed. To validate all data, power loss recovery reads the header structures. If a power loss has occurred during a data modification, power loss recovery will restore the original data.

3.3.5 Low Level Code and Interrupt Handling

FDI performs all programs and erases to flash media through a RAM based low-level flash driver.

The basic functions provided by the low-level flash driver are: program, erase, program/erase suspend, and interrupt polling. During a flash program or erase, interrupts with vectors in flash are disabled and control is turned over to a RAM based low-level flash driver. This routine polls interrupts while monitoring the progress of the flash program or erase operation. Upon the occurrence of an interrupt, the RAM based routine suspends the flash program or erase operation, and allows the interrupt handler to then execute from flash. Upon completion of the interrupt routine, the flash program or erase operation is resumed by the RAM flash driver.

Intel's Advanced Boot Block product family provides Erase Suspend to Read (ESR) in 20 µs maximum and Program Suspend to Read (PSR) in 10 µs maximum. This allows FDI to suspend program or erase and re-enabling interrupts with minimal latency.

3.3.6 Implementation Constraints

FDI is coded in ANSI Standard C. Assembly language is used only for those processes that require greater speed and optimization than a C compiler could provide.

Documentation to assist users in porting processor specific areas will be provided with the software.

3.3.7 Assumptions, Dependencies and Limitations

- Only one open Read/Write stream will be supported at any given time.
- Deletions of portions of stored data is not allowed.
- Data reads will not be interrupted.

3.4 FLASH DATA INTEGRATOR DETAILED DESIGN

3.4.1 Media Control Structures

FDI manages code and data separately to enable code execution and data storage in the same flash device. FDI provides a movable data/code partition, however, it must be on flash memory block boundary. A movable partition allows the ratio of code vs. data throughout the life of a system to evolve to match the systems needs. A symmetrically-blocked part is required to allow a movable data/code partition. If using an asymmetrically-blocked flash device, the data/code partition is fixed. This is a small block cannot be used as a spare block for a large block. FDI requires a spare block in the Data Storage area for reclamation.

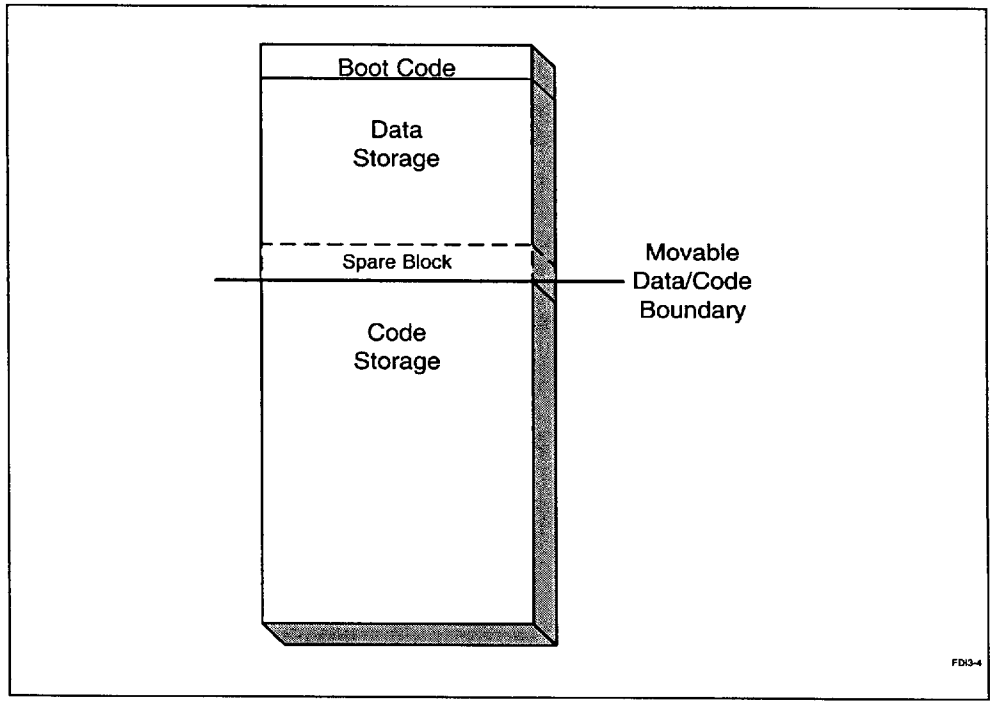

Figure 3-4. Code + Data Storage Arrangement in Flash

FDI supports boot code block separate from Code and Data storage which can contain the initialization code needed at startup. Some systems may have boot code stored external to the flash device, and would then use the entire flash device for code and data storage.

The following control structures are used by FDI to manage data. FDI provides flash read/write/modify capabilities with limited overhead and improved performance over EEPROM.

3.4.1.1 CONTROL STRUCTURES USED BY FDI

Command Control structure–Interface between the system and the FDI. A pointer to this structure enables the system and FDI to communicate and share information for reading, writing, and managing flash.

Data Lookup table–FDI indexes into this table to provide quick access to header location. During initialization, FDI recreates this array in RAM.

Unit Header structure–Describes the contents of the unit it points to with name, type, size, and attribute fields.

Multiple Instance structure–Describes the number of instances of the data parameter which can be contained within this unit, and the current valid instance.

Block Information structure–Used to track the logical block number, and power loss information.

Sequence Table structure–Describes the location of multiple Unit Headers describing multiple fragments of a data parameter of data stream.

Logical Block Table–Translation of logical block numbers from physical block numbers.

Data Location structure–Physical location of append data in the flash media.

Command structure–Contains the information needed to write data to or delete data from the flash media.

3.4.1.2 HOW FDI USES CONTROL STRUCTURES

Each flash data block consists of a list of Unit Headers addressed from the top of the block, and the data they describe addressed from the bottom of the block. Figure 3-5 shows the arrangement of Unit Headers and data within a physical block. At the bottom of each block is the Block Information structure. The Block Information structure maintains the logical block number, and tracks reclamation status for the block.

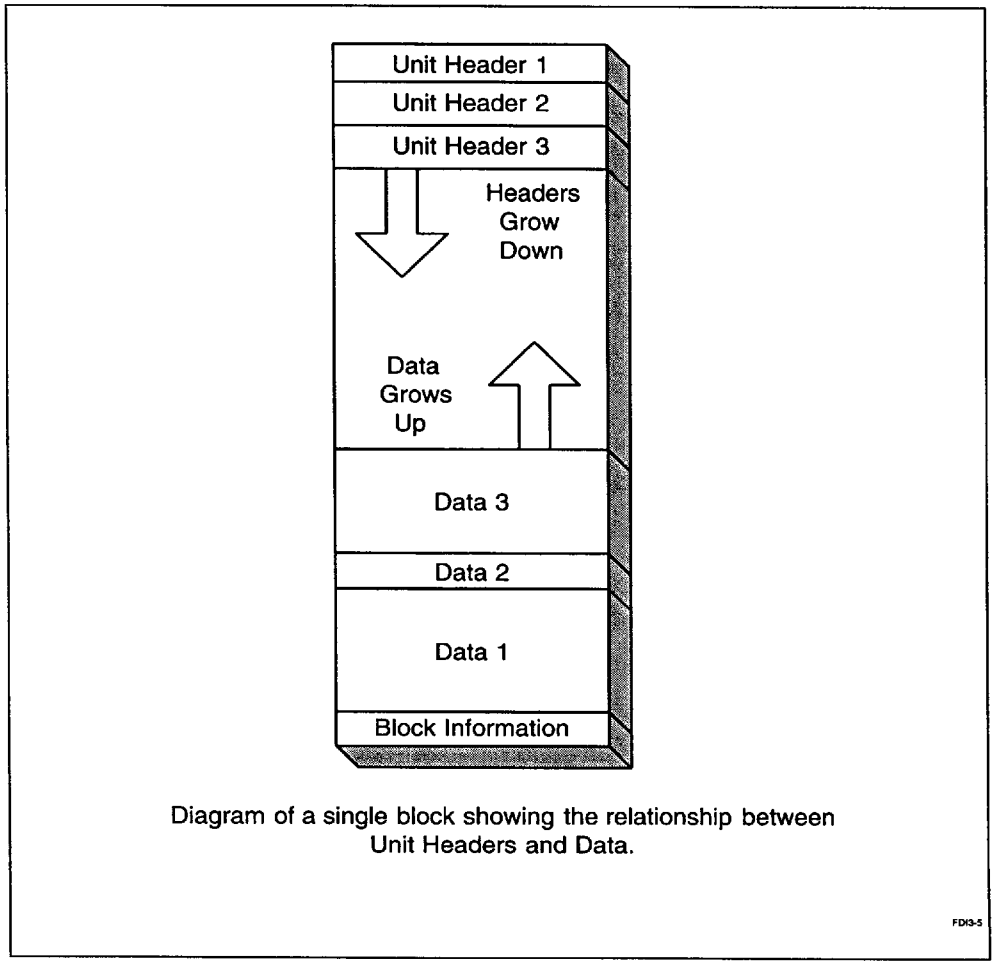

Diagram of a single block showing the relationship between Unit Headers and Data.

Figure 3-5. Data Block Arrangement

For ease in data management, physical blocks are segmented into sections called units. A unit is a segment of a block whose contents are described by a Unit Header. Units vary in size but always have the same granularity. Granularity is the minimum allocation unit size defined at compile time.

3.4.1.2.1 Command Control Structure

The Command Control structure is the interface between the system and FDI. A pointer to Command Control enables the system and FDI to communicate with each other, and share information for reading, writing, and managing flash.

Command Control Structure Fields

```
typedef struct command_control {
        DWORD  buffer;        /* buffer address */
        DWORD  count;         /* number of bytes desired */
        DWORD offset;         /* beginning offset into the data */
        DWORD  actual;        /* number of actual bytes acted on */
        DWORD  sub_cmd;       /* sub-command to expand functionality */
        DWORD  aux;           /* supplementary field */
        WORD   identifier;    /* unique identity for each data or code */
        BYTE   type;          /* command type: either data or code type */
        BYTE   priority;      /* each identifier is assigned a priority */
} COMMAND_CONTROL;
```

buffer–Pointer to a buffer to read data from, or write data to flash.

count–The number of bytes to read or write.

offset–The number of bytes into the Unit (offset) to begin reading or writing.

actual–The number of bytes FDI was able to read or write.

sub_cmd–Used for future or extended commands.

aux–Allows additional information to be passed between the application and FDI.

identifier–Unique identifier.

type–Used to define unique classes or types of information.

priority–The priority of the data determines the order data is written if data is queued for write.

3.4.1.2.2 Data Look-Up Table Structure

This look-up table increases the speed of accessing data. Since this table is located in RAM, it must be recreated from the Unit Header structures at initialization. The index into this table is based on the type and identifier value of each data parameter or stream.

Data Look-Up Table Structure Fields

```
typedef struct data_lookup {
        BYTE            ptrUnitHeader;      /* logical block and offset */
} DATA_LOOKUP;
```

ptrUnitHeader–Logical block and offset of the Unit Header whose name and type match the offset into this table.

3.4.1.2.3 Unit Header Structure

The Unit Header describes the data unit within the physical block. It also tracks the status bits of the data for reclamation and power loss recovery. Bit fields within the unit header structure indicate whether the data unit is active, being transferred, or invalid. Unit size indicates multiples of the base granularity.

Unit Header Structure Fields

```
typedef struct unit_header {
        WORD    identifier;   /* unique identifier per parameter */
        BYTE    status;       /* power-off recovery */
        BYTE    type;         /* data parameter, data stream, phone #,
                               *  fax #, SMS, etc. */
        WORD    size;         /* in multiples of granularity */
        WORD    ptrUnit;      /* offset from the bottom of the block */
} UNIT_HEADER;
```

identifier–A unique identity.

status–A bit-mapped field that indicates the current status of the data parameter, data stream, etc.

Table 3-1. Unit Header Structure Status Field Definitions

Name	Condition Status Value	Definition
"empty"	1111 111X Binary	This is an empty granular unit. The system can use this for the next unit header.
"allocating"	0111 111X Binary	The unit header is in the process of being written.
"allocated"	0011 111X Binary	The unit data is in the process of being written.
"valid"	0001 111X Binary	This unit header describes valid data.
"invalid"	0000 111X Binary	This unit header describes invalid data.

type–This field distinguishes the information associated with this header. The currently defined types in the first nibble of this field are attributes: Multiple Instance, Single Instance, Data Fragment and Sequence Table. The last nibble defines the associated data type: data parameter, data stream, phone number, etc.

size–Size is a multiple of granularity in this unit. Granularity is defined at compile time as the minimum number of bytes taken up by any unit.

ptrUnit–The offset from the beginning of the block to the start of the Unit data. PtrUnit is in multiples of granularity.

Table 3-2. Unit Header Structure Type Field Definitions

Type Value	Definition
XXXX 1110 Binary	This header points to Multiple Instance unit.
XXXX 1100 Binary	This header points to a sequence table.
XXXX 1000 Binary	This header points to a Single Instance unit.
XXXX 0000 Binary	This header points to a Data Fragment unit.
1110 XXXX Binary	The data type is data parameter.
1101 XXXX Binary	The data type is data stream.
1100 XXXX Binary	The data type is phone number.
1010 XXXX Binary	The data type is SMS.

3.4.1.2.4 Multiple Instance Structure

This structure describes multiple instances of small parameter data within a unit. Grouping the data into multiple instances limits the overhead in managing small data parameters, and improves the performance of updates. Figure 3-6 provides an example of a small data parameter with four available instances.

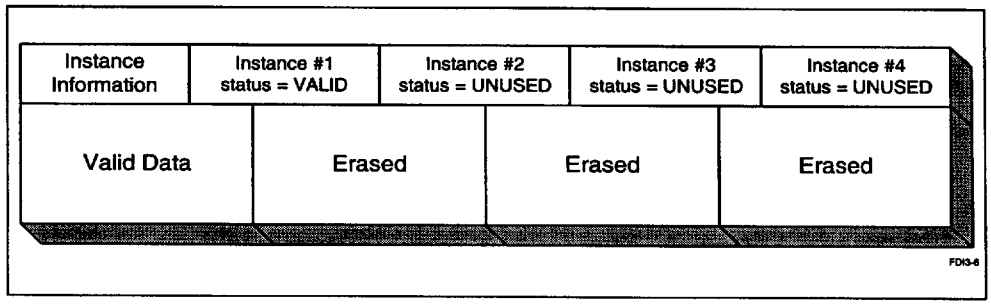

Figure 3-6. Example of Multiple Instances

Each instance has a corresponding status. New instances added have the status of "allocating," "allocated," and "valid." The old instances have the status of "invalid." Figure 3-7 displays a data parameter updated with a new instance. Note the status of the old instance is set to "invalid."

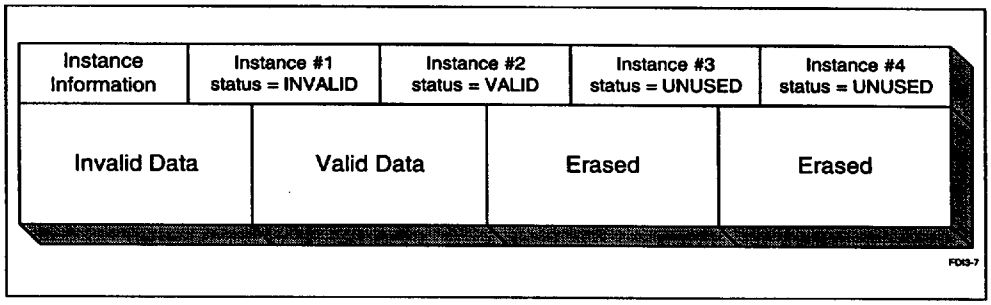

Figure 3-7. Parameter Update with New Instance

The number of multiple instances of the parameters is based on the size of the parameter and the size of the containing unit.

Multiple Instance Structure Fields

```
typedef struct data_info {
      WORD          sizeOfInst;          /* size of this instance of *
                                          * the data */
      BYTE          numOfInst;           /* number of data instances
                                          * available in this unit */
      Bitmask       validOfInst[];       /* 4 bits of validation for
                                          * each instance used */
} DATA_INFO;
```

sizeOfInst–The size in bytes of each data instance.

numOfInst–The number of instances available in this unit.

validOfInst[]–Contains four status bits for each instance in this unit.

Table 3-3. Multiple Instance Structure ValidOfInst Field Definitions

Name	Condition Status Value	Definition
"empty"	1111 Binary	This is an unused data instance.
"allocated"	0011 Binary	The instance is in the process of being written.
"valid"	0001 Binary	The instance holds valid data.
"invalid"	0000 Binary	The instance no longer holds valid data.

3.4.1.2.5 Block Information Structure

The Block Information Structure is located at the bottom of each physical block used for data storage. It contains the logical block number, reclamation status, and current state of the block. Figure 3-8 depicts the block information structure in a physical block.

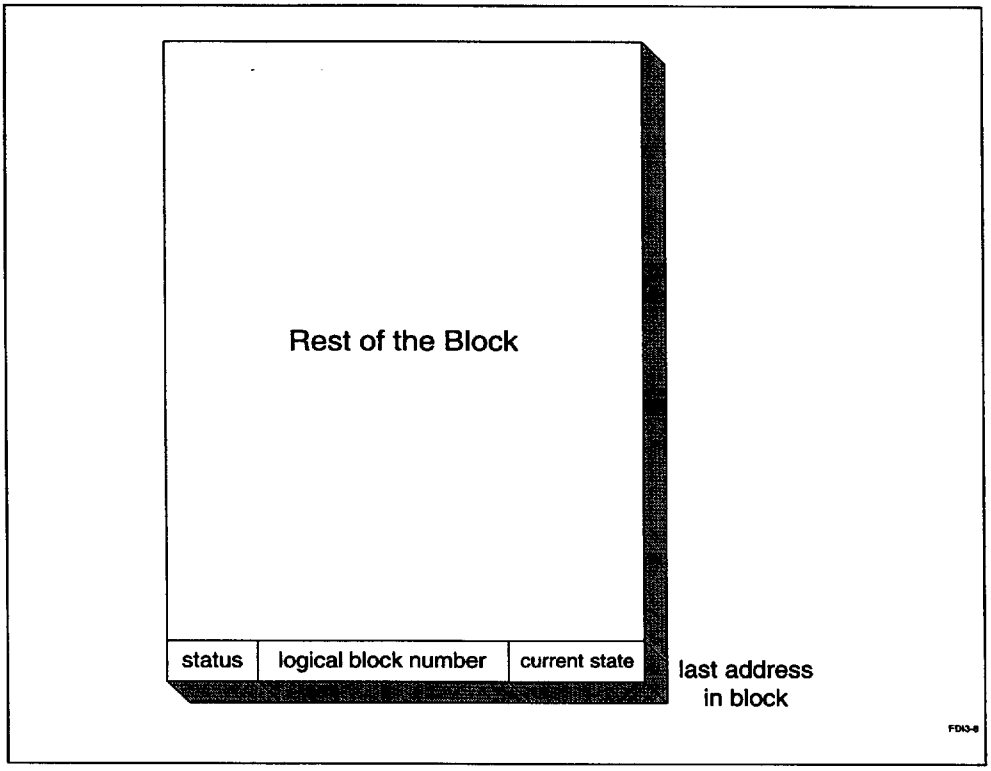

Figure 3-8. Placement of Block Information

Block Information Structure Fields

```
typedef struct block_info {
      BYTE   status;              /* includes: reclamation info */
      BYTE   logicalBlkNum;       /* allows for movable spare block */
      BYTE   physicalCopy;        /* physical block being copied during
                                   *  reclamation */
      WORD   currentState;        /* this will demonstrate block
                                   * integrity: F0F0H */
} BLOCK_INFO;
```

status–This field contains the information needed for the reclamation process.

Table 3-4. Block Information Structure Status Field Definitions

Name	Condition Status Value	Definition
"erased"	1111 1111 Binary	Indicates block has not been written to and all bits are in the erased state.
"recover"	1111 1110 Binary	Indicates that the process of placing data into this block from a block being reclaimed has begun.
"erasing"	1111 1100 Binary	All data has been transferred from a block being reclaimed to this block and the block indicated is undergoing erase.
"write"	1111 1000 Binary	This block is available for writing.

logicalBlkNum-Contains the logical block number.

physicalCopy-Contains the physical block number of the block being copied during reclaim.

currentState-Enables the FDI software to verify the block's integrity. CurrentState is checked to see if a block erase was interrupted by a power loss.

3.4.1.2.6 Sequence Table Structure

If data spans physical block boundaries, a sequence table is used to list each data fragment. Figure 3-9 provides an example of using a sequence table with three separate fragments across two physical blocks. Sequence tables contain an ordered list of data fragments. Unit Headers for each data fragment are described by logical block number and occurrence in the block. Notice in Figure 3-9 that the second fragment in the sequence table is associated to the second instance in block 1.

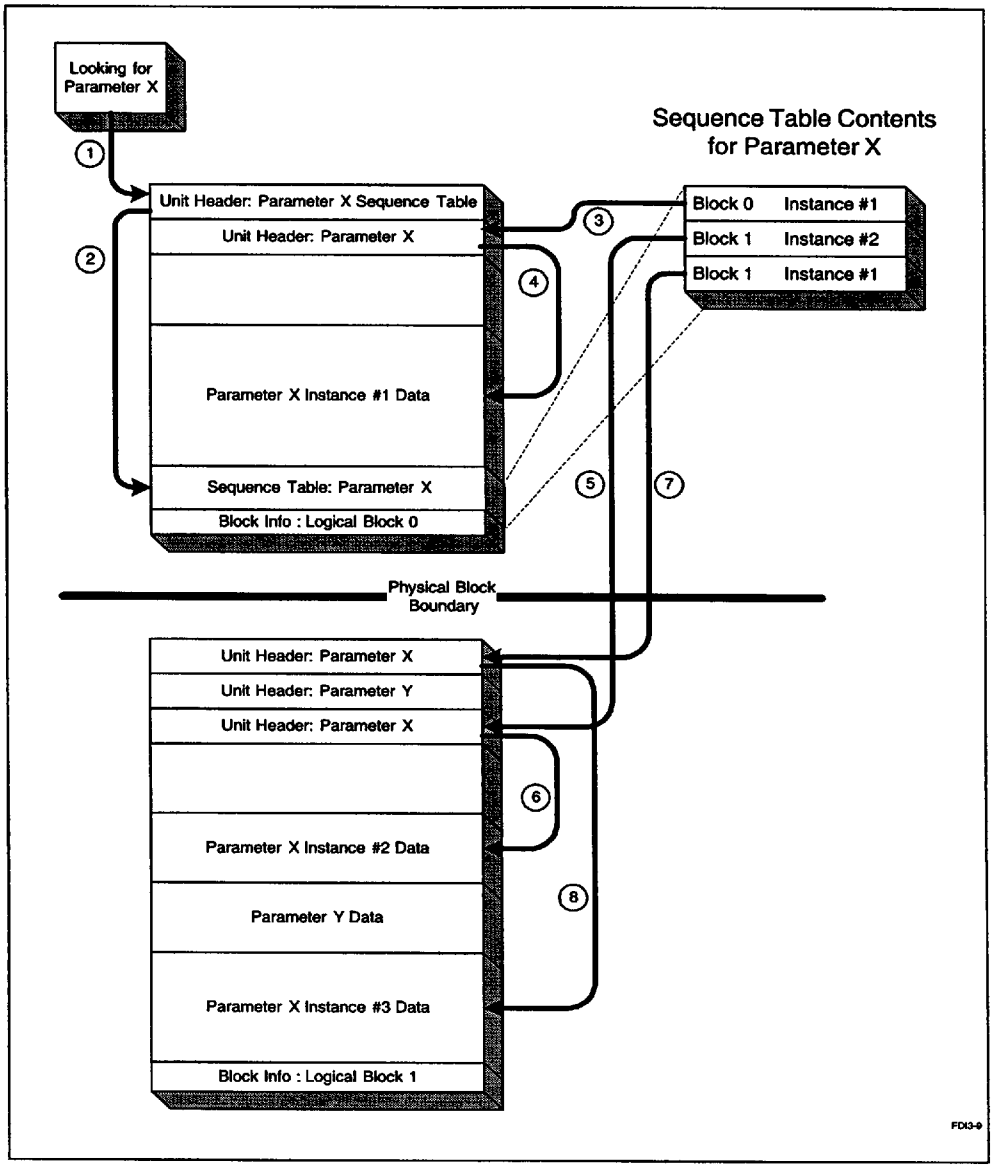

Figure 3-9. A Sequence Table Example

Sequence Table Structure Fields

```
typedef struct sequence_table {
        BYTE    blockNum;           /* virtual block number */
        BYTE    instance;           /* the instance of this fragment */
        WORD    size;               /* in multiples of granularity */
} SEQUENCE_TABLE;
```

blockNum–This is the block number of the parameter data.

instance–Because there can be multiple fragments in a single block, this field contains the instance of this fragment in the unit header table.

size–This field describes each fragment's size in multiples of granularity.

3.4.1.2.7 Logical Block Table

This table is a logical to physical block jump table where the index is the logical block number and the physical block number is the element.

Logical Block Table Structure Fields

```
typedef struct logical_block_tag {
        WORD    freeSpace;      /* the amount of free space in block */
        WORD    dirtySpace;     /* invalid space in block */
        BYTE    physical;       /* physical block identifier */
} LOGICAL_BLOCK;
```

freeSpace–The amount of free space available in the block measured in multiples of granularity.

dirtySpace–The amount of invalid space used in the block measured in multiples of granularity.

physical–This field is the physical block number of the logical block used as the index into this table.

3.4.1.2.8 Data Location Structure

This structure contains information about a data parameter or stream. It is used internally by FDI to assist with tracking information and improving performance.

Data Location Structure Fields

```
typedef struct data_location_tag {
      DWORD        ptrUnit;      /* ptr to unit accessed by header */
      IDTYPE identifier;         /* identity of data accessed */
      WORD         size;         /* the size of the data in
                                  * multiples of granularity */
      BYTE         type;         /* data and unit type accessed */
} DATA_LOCATION;
```

ptrUnit–The physical address of this unit's information structure.

identifier–A unique identity to validate stream open/close operations.

size–Size is a multiple of granularity in this unit.

type–This field distinguishes the information associated with this header. The currently defined types in the first nibble of this field are attributes: Multiple Instance, Single Instance, Data Fragment and Sequence Table. The last nibble defines the associated data type: data parameter, data stream, phone number, etc.

Table 3-5. Data Location Structure Type Field Definitions

Type Value	Definition
XXXX 1110 Binary	This header points to Multiple Instance unit.
XXXX 1100 Binary	This header points to a sequence table.
XXXX 1000 Binary	This header points to a Single Instance unit.
XXXX 0000 Binary	This header points to a Data Fragment unit.
1110 XXXX Binary	The data type is data parameter.
1101 XXXX Binary	The data type is data stream.
1100 XXXX Binary	The data type is phone number.
1010 XXXX Binary	The data type is SMS.

3.4.1.2.9 Data Queue Structure

The system writes to the Data Queue using FDI API functions to request flash reads, writes, and modifications. Figure 3-10 depicts the Data Queue Structure.

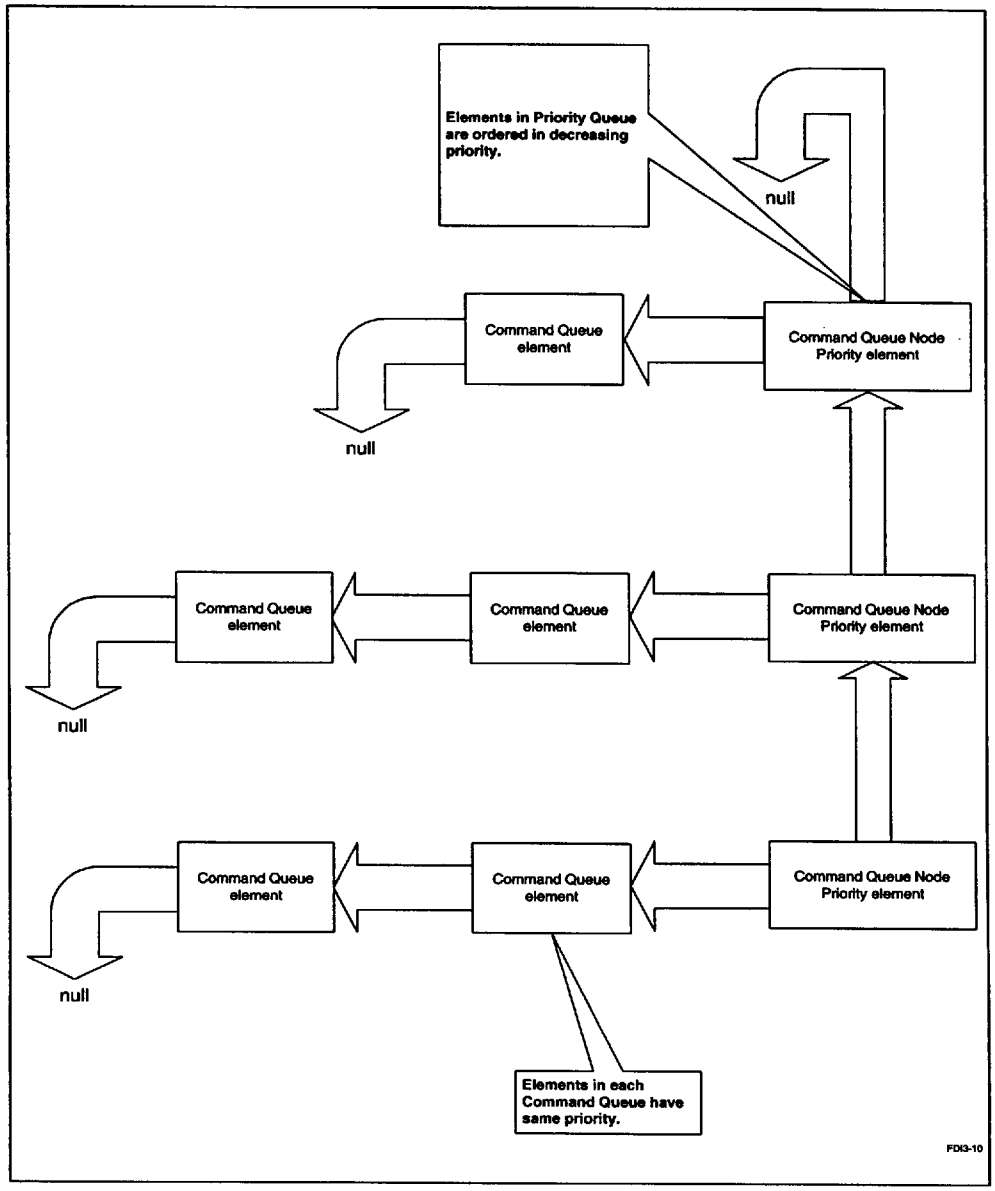

Figure 3-10. Data Queue Structure

Command Queue Node Fields

```
typedef struct priority_element_tag {
        PELEMENT_PTR ptrNextPriority;     /* points to lower priority
                                           * queue */
        CELEMENT_PTR ptrFirsttCmd;        /* points to command queue */
        BYTE         priority;            /* priority of queue. */
} PELEMENT;
```

ptrNextPriority–Points to the next lower priority element in the Command Queue Node priority queue. If there is no lower priority element, this field points to NULL.

ptrFirstCmd–Points to the first command element in the command queue of priority "priority."

priority–Data is accessed from the command queue in high to low priority order.

```
typedef struct command_element_tag {
        CELEMENT_PTR ptrNextCmd;    /* points to next command queue
                                     * element */
        WORD         dataSize;      /* size of data to read/write. */
        COMMAND      commandData;   /* command data structure */
} CELEMENT;
```

ptrNextCmd–Points to the next element in the command queue. If the current element is the last element, then ptrNextCmd is set to NULL.

dataSize–Indicates the size in bytes of the data buffer to be written/modified.

commandData–Contains the command, Id, offset into data unit, data unit type, and RAM data pointer, as described below.

```
typedef struct command_data_tag {
        IDTYPE     identifier;      /* identity of data accessed. */
        WORD       dataOffset;      /* beginning offset into the data. */
        BYTE       sub_cmd;         /* task execution sub commands. */
        BYTE       type;            /* command type: either data
                                     * parameter or a data stream. */
        BYTE_PTR   ptrContainer;    /* array of data follows. */
} COMMAND;
```

identifier–Unique identifier of a Unit to write/modify.

dataOffset–The number of bytes from the beginning of the Unit to begin operation.

sub_cmd–The type of data modification function requested.

type–Defines unique classes of data storage information.

ptrContainer–The ptrContainer field is a pointer to an allocated buffer which contains the data to be written to flash.

3.4.1.3 RAM USAGE AND CONTROL STRUCTURES

Variable Name	Data Type	Description	Size in Bytes
last_data_found	DATA_LOCATION	Global loaded by the dataFind routine	9
get_data_found	DATA_LOCATION	A copy of last_data_found used by the FDI_get function	9
open_stream	DATA_LOCATION	A copy of last_data_found used by the FDI_open and FDI_close routines	9
command_element	CELEMENT	The information for each write or delete function loaded in the command queue. This includes command_data structure also.	16 + data size
priority_element	PELEMENT	The priority information common for each write or delete function of same priority is loaded in this structure. This is a dummy element acting as a node for the command queue of that priority	9
qhead_ptr	PELEMENT	Points to the highest priority element in the data Queue	4
data_q_node	COMMAND	The information for each write or delete function loaded in the Data Queue	18 + data size
logical_block	LOGICAL_BLOCK	The table contains the logical block number, free space and dirty space for each physical block	5 / block

3.4.2 MODULES

3.4.2.1 FOREGROUND API SUB-SYSTEM

3.4.2.1.1 Opening a Data Parameter or Stream

FDI_open opens a data parameter or stream for reading, editing or creates a data stream for writing. Only one data parameter or stream can be opened at any given time.

Open Data Stream Format

```
int FDI_open(COMMAND_CONTROL *cmd_cntrl);
```

Input Elements

Identifier	Description	Data Type	Data Rep.	Limit /Range	Validity Check Performed ?	Input Method
cmd_cntrl->sub_cmd	This field contains a command to indicate the data stream opening method: OPEN_READ: Open for read only. OPEN_MODIFY: Open for parameter data modification. OPEN_CREATE: Open for writing a new parameter.	DWORD	flag	 OPEN READ OPEN MODIFY OPEN CREATE	yes	by reference
cmd_cntrl->aux	unused					
cmd_cntrl->identifier	This is a unique data parameter or stream identifier.	WORD	integer	na	yes	by reference
cmd_cntrl->type	This field indicates a data type. The options are: DATA_ PARAMETER or DATA_STREAM	BYTE	id_type	na	yes	by reference
cmd_cntrl->ptrBuffer	unused					

Identifier	Description	Data Type	Data Rep.	Limit /Range	Validity Check Performed ?	Input Method
cmd_cntrl->count	unused					
cmd_cntrl->offset	unused					
cmd_cntrl->actual	unused					
cmd_cntrl->priority	unused					

Processes Characteristics

FDI_open improves performance when accessing a data stream multiple times by maintaining location information in RAM. FDI_open stores the data location information in the global open_stream structure (Section 3.4.1.2.8). FDI_open returns an error if the system attempts to open multiple data parameters or streams. FDI_open calls dataFind to determine the existence of data with a matching identifier and type. If data already exists and is being opened for reading or modification, FDI_open gets the data size in multiples of granularity and updates the open_stream structure size field. Successive calls to FDI_read or FDI_write by pass the call to dataFind and the size update, thus reducing overhead.

Error Handling

If during the open process there is an error, the return value will contain a descriptive error code.

Utilization of Other Elements

FDI_open uses the dataFind routine defined in Section 3.4.2.5.1.

The global structure open_stream is from the Data Location Structure defined in Section 3.4.1.2.8.

Limitations

Only one data parameter or stream can be opened at any given time.

Output Elements

Identifier	Description	Data Type	Data Representation	Limit / Range	Validity Check Performed ?	Output Method
cmd_cntrl->ptrBuffer	Points to the buffer which has a copy of the open_stream structure information	DWORD	DATA_LOCATION pointer		no	by reference
error	Refer to the return codes in Section 3.5.	BYTE	count		no	function call
cmd_cntrl->identifier	reserved					
cmd_cntrl->type	reserved					
cmd_cntrl->count	unused					
cmd_cntrl->offset	unused					
cmd_cntrl->actual	unused					
cmd_cntrl->aux	unused					
cmd_cntrl->priority	unused					
cmd_cntrl->sub_cmd	reserved					

3.4.2.1.2 Closing a Data Parameter or Stream

FDI_close closes an open data stream or parameter.

Close Call Format

```
int FDI_close(COMMAND_CONTROL *cmd_cntrl);
```

Input Elements

Identifier	Description	Data Type	Data Rep.	Limit / Range	Validity Check Performed ?	Input Method
cmd_cntrl->identifier	Unique data parameter identifier	IDTYPE	integer	na	no	by reference
cmd_cntrl->type	This field indicates a data type. The options are: DATA_PARAMETER or DATA_STREAM	BYTE	integer	ENUM	yes	by reference
cmd_cntrl->sub_cmd	unused					
cmd_cntrl->count	unused					
cmd_cntrl->offset	unused					
cmd_cntrl->actual	unused					
cmd_cntrl->aux	unused					
cmd_cntrl->priority	unused					
cmd_cntrl->ptrBuffer	reserved					

Processing Characteristics

FDI_close determines if a data stream or parameter is open. FDI_close returns an error if no data stream or parameter is open. FDI_close clears the open_stream structure to indicate closing a data stream or parameter.

Error Handling

If during the close process there is an error, the return value will contain a descriptive error code.

Utilization of Other Elements

The global structure open_stream is from the Data Location Structure defined in Section 3.4.1.2.8.

Limitations

Not applicable.

Output Elements

Identifier	Description	Data Type	Data Rep.	Limit / Range	Validity Check Performed ?	Output Method
error	Refer to the return codes in Section 3.5.	BYTE	count		no	function call
cmd_cntrl->identifier	reserved					
cmd_cntrl->type	reserved					
cmd_cntrl->sub_cmd	reserved					
cmd_cntrl->ptrBuffer	unused					
cmd_cntrl->count	unused					
cmd_cntrl->offset	unused					
cmd_cntrl->actual	unused					
cmd_cntrl->priority	unused					
cmd_cntrl->aux	unused					

3.4.2.1.3 Delete Parameter Data

The delete process invalidates a data parameter or stream in the flash media.

Delete Call Format

```
int FDI_delete(COMMAND_CONTROL *cmd_cntrl);
```

Input Elements

Identifier	Description	Data Type	Data Rep.	Limit / Range	Validity Check Performed ?	Input Method
cmd_cntrl->identifier	Unique data parameter or stream identifier	IDTYPE	integer	na	no	by reference
cmd_cntrl->type	This field indicates a data type. The options are: DATA_PARAMETER or DATA_STREAM	BYTE	integer	ENUM	yes	by reference
cmd_cntrl->priority	Used to indicate the priority of the data parameter or stream	BYTE	integer	na	yes	by reference
cmd_cntrl->sub_cmd	unused					
cmd_cntrl->count	unused					
cmd_cntrl->offset	unused					
cmd_cntrl->actual	unused					
cmd_cntrl->aux	unused					
cmd_cntrl->ptrBuffer	reserved					

Processing Characteristics

FDI_delete loads all requests to modify the flash data into the Data Queue for the Background Manager to execute in priority order. FDI_delete returns an error if the data parameter or stream is open. Using the input identifier and type as parameters, a call to dataFind verifies the data's existence. FDI_delete fills a Data Queue entry structure buffer and calls dataQSend to add the entry to the queue.

Error Handling

If during the delete process there is an error, the return value will contain a descriptive error code.

Utilization of Other Elements

FDI_delete uses the dataFind routine defined in Section 3.4.2.5.1 and the dataQSend routine defined in Section 3.4.2.4.2.

Limitations

Not Applicable.

Output Elements

Identifier	Description	Data Type	Data Rep.	Limit / Range	Validity Check Performed ?	Output Method
cmd_cntrl->ptrBuffer	unused					
error	Refer to the return codes in Section 3.5.	BYTE	count		no	function call
cmd_cntrl->identifier	reserved					
cmd_cntrl->type	reserved					
cmd_cntrl->priority	reserved					
cmd_cntrl->count	unused					
cmd_cntrl->offset	unused					
cmd_cntrl->actual	unused					
cmd_cntrl->aux	unused					
cmd_cntrl->sub_cmd	reserved					

3.4.2.1.4 Getting Parameter Data

The get process locates the first or next data parameter or stream of the specified type or it will find a matched data parameter or stream using the type and identifier fields. FDI_get places information from the global last_data_found structure into the callers buffer if the ptrBuffer parameter is non-zero.

Get Call Format

```
int FDI_get(COMMAND_CONTROL *cmd_cntrl);
```

Input Elements

Identifier	Description	Data Type	Data Rep.	Limit /Range	Validity Check Perf.?	Input Method
cmd_cntrl->sub_cmd	Contains a flag to modify the functions action: GET_FIRST: finds the first data parameter of a given type GET_NEXT: finds the matching data parameter of a given type and identifier GET_MATCHED: finds the matching data parameter of a given type and identifier	DWORD	flag	commands listed below: GET_FIRST GET_NEXT GET_MATCHED	yes	by ref.
cmd_cntrl->identifier	Unique data parameter identifier	IDTYPE	integer	na	no	by ref.
cmd_cntrl->type	This field indicates a data type. The options are: DATA_PARAMETER or DATA_STREAM	BYTE	integer	ENUM	yes	by ref.
cmd_cntrl->ptrBuffer	A pointer to a buffer to contain the get_data_found information	DWORD	DATA_LOCATION structure pointer			
cmd_cntrl->count	unused					
cmd_cntrl->offset	unused					
cmd_cntrl->actual	unused					
cmd_cntrl->priority	unused					
cmd_cntrl->aux	unused					

Processing Characteristics

FDI_get calls the dataFind routine and saves the data location into the structure get_data_found defined from the Data Location structure type. FDI_get uses get_data_found to find the next data item of the same type if the input sub-command is GET_NEXT. FDI_get places information from the global last_data_found structure into the callers buffer if the ptrBuffer parameter is non-zero.

Error Handling

If the dataFind routine returns an error, the function sets the input parameter buffer to zero and returns the error. The error ERR_NOTEXISTS indicates the last id of a particular type has been found or if the type is GET_MATCHED, indicates that the data does not exist.

Utilization of Other Elements

FDI_get uses the dataFind routine defined in Section 3.4.2.5.1.

The local structure get_data_found is from the Data Location Structure defined in Section 3.4.1.2.8.

Limitations

Not applicable.

Output Elements

Identifier	Description	Data Type	Data Representation	Limit / Range	Validity Check Performed ?	Output Method
cmd_cntrl->ptrBuffer	A pointer to a buffer to contain the get_data_found information	DWORD	DATA_LOCATION structure pointer	na	no	by reference
error	Refer to the return codes in Section 3.5.	BYTE	count		no	function call
cmd_cntrl->identifier	reserved					
cmd_cntrl->type	reserved					
cmd_cntrl->count	unused					
cmd_cntrl->offset	unused					
cmd_cntrl->actual	unused					
cmd_cntrl->aux	unused					
cmd_cntrl->priority	unused					
cmd_cntrl->sub_cmd	reserved					

3.4.2.1.5 Reading Parameter Data

The read process returns the specified portion of a data parameter or stream's content into a calling routine's data buffer.

Read Call Format

```
int FDI_read(COMMAND_CONTROL *cmd_cntrl);
```

Input Elements

Identifier	Description	Data Type	Data Rep.	Limit / Range	Validity Check Performed ?	Input Method
cmd_cntrl->ptrBuffer	This is a pointer to a buffer in which the data content is placed.	DWORD	pointer	na	no	by reference
cmd_cntrl->count	This field contains the number of bytes to read.	DWORD	count	na	yes	by reference
cmd_cntrl->offset	This is the number of bytes into the data parameter or stream to begin reading.	DWORD	index	na	yes	by reference
cmd_cntrl->actual	This field returns the actual number of bytes read.	DWORD	count	na	no	by reference
cmd_cntrl->sub_cmd	unused					
cmd_cntrl->aux	unused					
cmd_cntrl->identifier	This field indicates a data type. The options are: DATA_PARAMETER or DATA_STREAM	WORD	integer	na	yes	by reference
cmd_cntrl->type	This field indicates a parameter data type.	BYTE	id_type	na	yes	by reference
cmd_cntrl->priority	Priority of the data.	BYTE	integer	na	yes	by reference

Processing Characteristics

FDI_read reads from the starting offset location in flash and writes count bytes of data into the input buffer. Using the identifier and type input parameters, FDI_read verifies the data's existence in the open_stream structure. FDI_read calls dataFind to update the open_stream structure if the data is not currently open. FDI_read gets the data size in multiples of granularity and updates the open_stream structure size field. FDI_read validates the count and offset input parameters against the data size to ensure the read does not go beyond the end of the data. Otherwise FDI_read returns only the size of data stored in flash.

If the data stream or parameter is fragmented, the data fragments are read unit by unit for each fragment in the sequence table until done reading. If no fragmentation exists, a single unit

needs to be read. To read information from any unit, a starting offset and data size are set up to indicate the amount of data read within the current unit. FDI_read calls readUnit to read from the starting offset location in flash and writes size bytes of data into the input buffer. FDI_read calls readUnit for each unit of data requested.

Finally, FDI_read determines if more data writes of the same identifier and type are pending in the Data Queue by calling dataQPeek. If matching data is in the queue, this routine returns a pointer to the queue buffer. FDI_read updates the input buffer data if commands in the Data Queue overwrite the same data. FDI_read repeatedly calls dataQPeek until there are no more matching data modification items in the Data Queue.

Error Handling

If during the read process there is an error, the return value will contain a descriptive error code.

Utilization of Other Elements

FDI_read uses the dataFind routine defined in Section 3.4.2.5.1, the dataQPeek routine defined in Section 3.4.2.4.5 and the readUnit routine defined in Section 3.4.2.5.2.

The global structure open_stream is from the Data Location Structure defined in Section 3.4.1.2.8.

Limitations

Not Applicable.

Output Elements

Identifier	Description	Data Type	Data Rep.	Limit / Range	Validity Check Performed?	Output Method
cmd_cntrl->ptrBuffer	Points to the buffer into which the data is read	DWORD	pointer	na	no	by reference
error	Refer to the return codes in Section 3.5.	BYTE	count	na	no	function call
cmd_cntrl->identifier	reserved					
cmd_cntrl->type	reserved					
cmd_cntrl->count	unused					
cmd_cntrl->offset	unused					

Identifier	Description	Data Type	Data Rep.	Limit / Range	Validity Check Performed?	Output Method
cmd_cntrl->actual	The actual number of bytes read	DWORD	count	na	no	by reference
cmd_cntrl->aux	unused					
cmd_cntrl->priority	unused					
cmd_cntrl->sub_cmd	reserved					

3.4.2.1.6 Writing Parameters and Streams

The write process queues data to be written or replaced by the Background Manager.

Write Call Format

```
int FDI_write(COMMAND_CONTROL *cmd_cntrl);
```

Input Elements

Identifier	Description	Data Type	Data Rep.	Limit /Range	Validity Check Perf.?	Input Method
cmd_cntrl->ptrBuffer	This points to the data to be written	BYTE_PTR	byte pointer	1006 bytes	no	by ref.
cmd_cntrl->count	The number of bytes to write.	DWORD	integer	1006 (queue limit)	yes	by ref.
cmd_cntrl->offset	The index into the data at which the new data is written	DWORD	integer	na	yes	by ref.
cmd_cntrl->sub_cmd	Contains a flag to modify the function's action: WRITE_REPLACE: replaces existing data in flash WRITE_APPEND: writes after existing or creates new data WRITE_MODIFY: modifies existing data in place	DWORD	flag	commands listed below: WRITE_REPLACE WRITE_APPEND WRITE_MODIFY	yes	by ref.
cmd_cntrl->identifier	Unique data parameter or stream identifier.	IDTYPE	integer	na	no	by ref.

Identifier	Description	Data Type	Data Rep.	Limit /Range	Validity Check Perf.?	Input Method
cmd_cntrl->type	This field indicates a data type. The options are: DATA_PARAM-ETER or DATA_STREAM	BYTE	integer	ENUM	yes	by ref.
cmd_cntrl->priority	Priority value of the data.	BYTE	integer	0-255	yes	by ref.
cmd_cntrl->aux	unused					
cmd_cntrl->actual	unused					

Processing Characteristics

FDI_write loads all requests to modify the flash data into the Data Queue for the Background Manager to execute in priority order. FDI_write allocates memory space for each command element and the Background Manager will free the space upon completion of executing the request.

FDI_write checks the open_stream structure to see if the data is already opened. Otherwise FDI_write calls dataFind to see if the data exists in the flash media. FDI_write creates the data if the sub-command type is WRITE_APPEND and dataFind returns error ERR_NOTEXIST.

A call to dataQSend loads the command element information into the Data Queue.

Error Handling

If during the write process there is an error, the return value will contain a descriptive error code.

Utilization of Other Elements

The malloc library function is needed to dynamically allocate memory for the command element and the data container. DataQSend (Section 3.4.2.4.2) loads the command element information into the Data Queue. FDI_write uses dataFind defined in Section 3.4.2.5.1.

Limitations

Not applicable.

Output Elements

Identifier	Description	Data Type	Data Representation	Limit / Range	Validity Check Performed ?	Output Method
error	Refer to the return codes in Section 3.5.	BYTE	count		no	function call

3.4.2.1.7 Reclaim Request Enable

FDI_reclaimEnable unblocks the reclaim process pending on the reclaimEnable Semaphore. A reclaim can be invoked by the system at any time.

Reclaim Call Format

```
void FDI_reclaimEnable(void);
```

Input Elements

None.

Processing Characteristics

FDI_reclaimEnable grants reclaim permission by asserting the reclaimEnable semaphore.

reclaimEnable: Used to control flash reclamation. When reclaimEnable is asserted, reclamation is unblocked and proceeds to reclaim invalid (written to but superseded) areas of flash.

Error Handling

None.

Utilization of Other Elements

FDI_reclaimEnable uses the reclaimEnable semaphore.

Limitations

Not applicable.

Output Elements

None.

3.4.2.1.8 Memory Statistics

This memory statistics process calculates the memory statistics of the media and returns the values to the calling function.

Memory Statistics Call Format

```
void FDI_statistics(WORD *freeUnits, WORD *invalidUnits);
```

Input Elements

Identifier	Description	Data Type	Data Representation	Limit / Range	Validity Check Performed ?	Input Method
freeUnits	Pointer to number of free units.	WORD	pointer		no	by reference
invalidUnits	Pointer to number of invalid units.	WORD	pointer		no	by reference

Processing Characteristics

FDI_statistics provides a way to monitor the FDI usage of the entire media. These values represent FDI usage for the Flash Data Integrator structures and application data. The total number of clean units remaining does not include those units held in reserve (system_threshold and FDI_threshold). Each entry of the Logical Block table tracks the statistics of individual blocks. FDI_statistics adds up the free_space field of the Logical Block table to get the total_free_space and the dirty_space field to get the total_invalid_space. FDI_statistics returns these values by filling in the variables pointed to by the pointers that were passed in by the calling function.

Error Handling

Not applicable.

Utilization of Other Elements

FDI_statistics uses the Logical Block table defined in Section 3.4.1.2.7.

Limitations

Not applicable.

Output Elements

Identifier	Description	Data Type	Data Representation	Limit / Range	Validity Check Performed ?	Output Method
freeUnits	Number of free units in terms of granularity.	WORD	count		no	by reference
invalidUnits	Number of invalid units in terms of granularity.	WORD	count		no	by reference

3.4.2.1.9 Data Queue Status

FDI_status returns the queue status and the reclamation status to the calling function.

Status Call Format

```
BYTE FDI_status(void);
```

Input Elements

None.

Processing Characteristics

FDI_status returns whether reclamation is in progress and whether the data queue is empty.

The table below indicates all possible values that can be returned by this function and its interpretation.

Bit mask	Interpretation
0000	No pending reclamation. Queue is empty.
0001	Reclamation is pending. Queue is empty.
0010	No pending reclamation. Message pending in the queue.
0011	Reclamation is pending. Message pending in the queue.

Error Handling

None.

Utilization of Other Elements

None.

Limitations

Not applicable.

Output Elements

Identifier	Description	Data Type	Data Representation	Limit / Range	Validity Check Performed ?	Output Method
value	Return value indicates the status of the queue and reclamation.	BYTE	integer	0–3	no	function call

3.4.2.2 BACKGROUND MANAGER SUB-SYSTEM

3.4.2.2.1 Background Task

BkgrdTask is the FDI Background Manager which performs any modifications/writes to flash. BkgrdTask pends on data available in the Data Queue, and then reads the highest priority Command Queue element. BkgrdTask disables system interrupts before issuing program or erase commands. BkgrdTask then polls interrupts while waiting for the program or erase to complete. If an interrupt is asserted during this time, bkgrdTask suspends the flash program or erase in progress, and then relinquishes the system to the interrupting task. Once the interrupt handler has completed executing, bkgrdTask continues until it completes, or until it is interrupted again by another interrupt. BkgrdTask continues in this fashion until the Data Queue is empty. BkgrdTask also determines when a reclaim is necessary by checking predetermined thresholds.

Background Task Call Format

```
int bkgndTask(void);
```

Input Elements

Not applicable.

Processing Characteristics

When pending on the queueCount semaphore, bkgndTask looks for the highest priority item in the Data Queue. When an item arrives in the Data Queue, bkgndTask gets a pointer to the item by calling dataQReceive. BkgndTask uses the Data Queue command information pointed to by dataQReceive to modify the flash media. BkgndTask determines the location of the data in flash by calling dataFind if the data is not already opened. BkgndTask modifies the Data

Header status field if the operation is a DELETE sub-command. For the cases of WRITE_REPLACE and WRITE_MODIFY commands bkgndTask determines the storage method of the data, either a Multiple Instance unit, a Single Instance unit, or a fragmented data unit.

If the data is in a Multiple Instance unit and there is available space, bkgndTask creates a new instance of the data within the existing Multiple Instance unit. The old data instance in the unit is marked invalid. If there is not enough space in the Multiple Instance unit, bkgndTask creates a new unit with corresponding Unit Header with the replacement or modified data. The old Multiple Instance unit and associated Unit Header is invalidated.

If the storage method is a Single Instance unit and there is available space, bkgndTask creates a new unit with corresponding Unit Header with the replacement or modified data. The old Single Instance unit and associated Unit Header is invalidated.

If the storage method is a fragmented data unit, this requires writing the replacement or modified data to a new unit and corresponding Unit Header. BkgndTask recreates the Sequence Table and associated Unit Header to reflect the changed data. In the case of the WRITE_APPEND command, bkgndTask determines the storage method of the data. If the method is a Multiple Instance unit, bkgndTask creates a new unit with the additional data added. The old unit and associated Unit Header is invalidated. If the storage method is a fragmented data unit, bkgndTask creates a new unit and Unit Header with the additional data and updates the Sequence Table.

Before a data write takes place, it is necessary to determine if enough space exists on the flash media. If the data size is greater than the available space above the system_threshold, bkgndTask checks the condition of the reclaimDone semaphore. If reclaimDone semaphore is asserted, bkgndTask asserts the reclaimRequest semaphore and resets the reclaimDone semaphore. The write command completes if the data size fits within the available space and that headroom exists between the system_threshold and the FDI_threshold. Otherwise, bkgndTask is pending on reclaimDone semaphore. Once the reclamation function asserts the reclaimDone semaphore, bkgndTask continues with the write or delete operation.

Before modifying flash media, bkgndTask determines if there is enough time to execute the command. BkgndTask delays if there is not enough time until the next interrupt occurs. A call to the flashLowLevel function executes the command from within RAM. If an error occurs during this low-level function, bkgndTask gives an error semaphore to indicate the error to the system. If the low-level operation completes, a call to dataQDelete removes the item from the queue and bkgndTask is again pending on items in the Data Queue.

Error Handling

If an error is returned the semaphore describing the error and location is given to the system.

Utilization of Other Elements

BkgndTask uses dataQReceive defined in Section 3.4.2.4.3, dataQDelete defined in Section 3.4.2.4.4 and flashLowLevel defined in Section 3.4.2.5.5. The semaphores used are queueCount, reclaimRequest and reclaimDone.

Limitations

Not applicable.

Output Elements

Not applicable.

3.4.2.2.2 Low-Level Interrupt and Status Polling

RAM_flashModify exists in RAM and allows a real-time multi-tasking system flash "read while write" capabilities.

Low-Level Polling Call Format

```
int RAM_flashModify(LOWLVL_INFO_PTR ptrLowlvlInfo);
```

Input Elements

The input parameter structure is defined as follows:

```
typedef struct lowlvl_info_tag {
        DWORD  address;      /* identity of data accessed */
        DWORD  ptrBuffer;    /* pointer to actual data */
        WORD   count;        /* number of bytes to write to flash */
        BYTE   command;      /* task execution sub-commands */
} LOWLVL_INFO;
```

Identifier	Description	Data Type	Data Rep.	Limit / Range	Validity Check Perf.?	Input Method
ptrLowlvlInfo->address	Beginning address of flash media data.	DWORD	address pointer		no	by reference
ptrLowlvlInfo->count	Number of bytes if programming.	WORD	byte counter	0–0XFFFF	yes	by reference
ptrLowlvlInfo->command	Either program or erase.	BYTE	command flag	0– 1	yes	by reference
ptrLowlvlInfo->ptrBuffer	Pointer to buffer containing data for programming.	BYTE pointer	array of bytes		no	by reference

Processing Characteristics

RAM_flashModify disables the system Task Scheduling so the scheduler does not interrupt the write process. The Data Queue still contains the current data element being acted upon. Next, RAM_flashModify disables the interrupts. This is the point of worst case interrupt latency, after the interrupts have been disabled. RAM_flashModify calculates the "time until next interrupt" using the last interrupt time-stamp and the current time. There must be available time for a minimum run, overhead and command suspend time. If there is not enough time RAM_flashModify re-enables the interrupts and the task scheduler. RAM_flashModify then delays until the next interrupt occurs and the process begins again.

If enough time exists, RAM_flashModify gives the program or erase command to start or continue the operation. Checking the status register verifies the command is complete.

If the operation is a programming command the byte counter is decrements and the address pointer increments to the next location. RAM_flashModify checks the status register for errors if there are no more bytes to write and sets the status variable to indicate correct command completion or error. Verification of the status register ensures the completion of the operation. RAM_flashModify analyzes the available time if the command has not completed. RAM_flashModify sets the status variable to the suspended state if there is not enough time to poll the status register or an interrupt has occurred. This is the point of best case interrupt latency, after the interrupt polling. RAM_flashModify suspends the program or erase command and waits for the operation to complete. RAM_flashModify re-enables the interrupts and the task scheduler. The system will vector to the address of the interrupt handler. After the system interrupt completes and the Background Manager is allowed CPU time, the process is continued until interrupted again or until complete.

If the status variable indicates that the program/erase command was suspended, RAM_flashModify disables the Task Scheduling, disables the interrupts and verifies the available time. RAM_flashModify resumes the previously interrupted command until the variable status indicates completion or error.

Error Handling

RAM_flashModify returns the status value to the calling function.

Utilization of Other Elements

None.

Limitations

Not applicable.

Output Elements

Identifier	Description	Data Type	Data Rep.	Limit / Range	Validity Check Perf.?	Output Method
status	Returns the status of the command upon completion.	BYTE	unsigned char	0 - okay 1 - suspended 2 - an error	no	function call

3.4.2.2.3 Reclamation Management

Reclamation is the process used to make invalid regions of flash memory available for reuse for parameter storage. The FDI system uses a spare block for reclaiming valid headers and data. This spare is not used for parameter storage but is used for reclamation only. When there is no more room to write updated or new data, the system finds a block with the greatest amount of invalid space and finds the spare block.

Figure 3-11 displays the reclaim candidate and the spare block prior to the reclaim process.

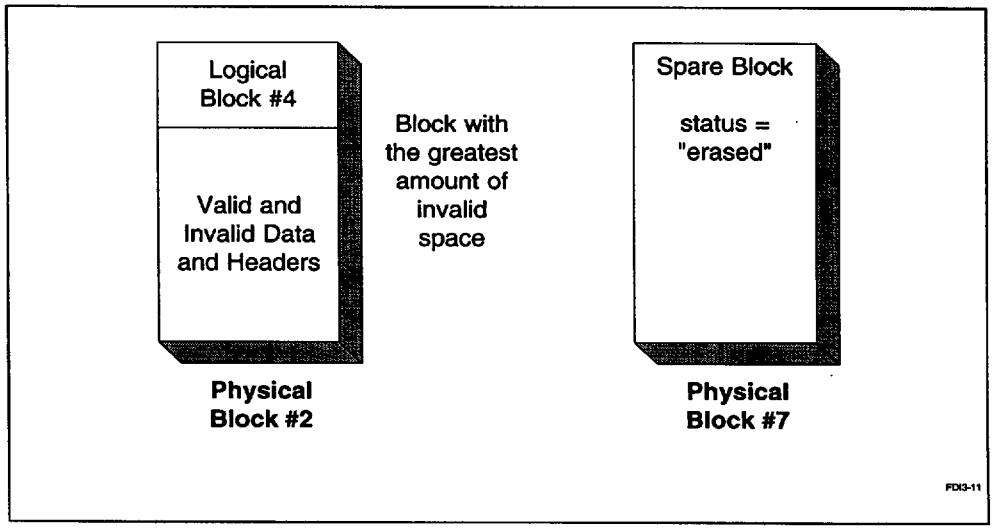

Figure 3-11. Reclaim Candidate and Spare

Transfer of valid data and headers from the block being reclaimed to the spare block is the next step. The status of the spare block indicates the valid data is being transferred. Figure 3-12 demonstrates this process. Notice invalid data and headers do not transfer to the spare block.

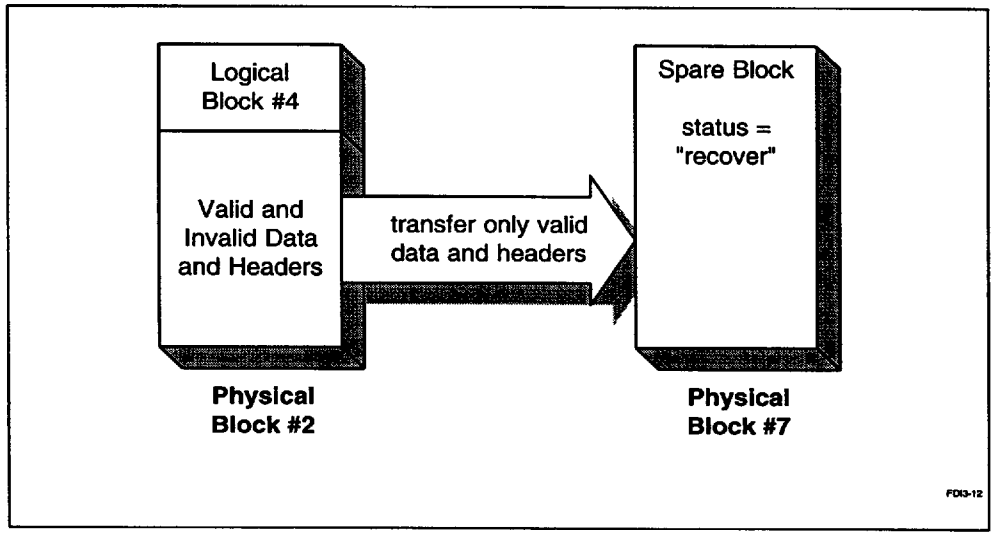

Figure 3-12. Transferring Valid Data to the Spare Block

Upon completion of the data transfer the next step is to prepare the spare block to logically take the original block's place. The FDI writes the original block's logical block number to the spare block's block information structure. For power off recovery purposes the status of the spare block indicates the original block is about to be erased. The old block begins to erase after all valid data exists in the spare block. Figure 3-13 shows the status of the spare and reclaim blocks at this time.

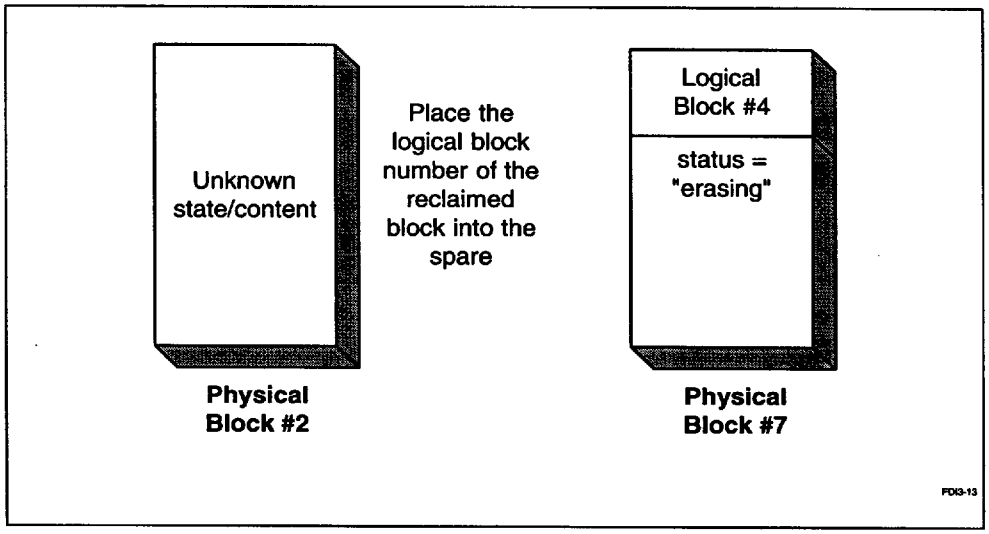

Figure 3-13. Erasing the Reclaim Block

When the erase of the original block completes, the status of the spare block changes to indicate that the block is now a writable block. All of the valid data and headers have been transferred successfully and the old locations have been erased. The new status of the blocks is shown in Figure 3-14.

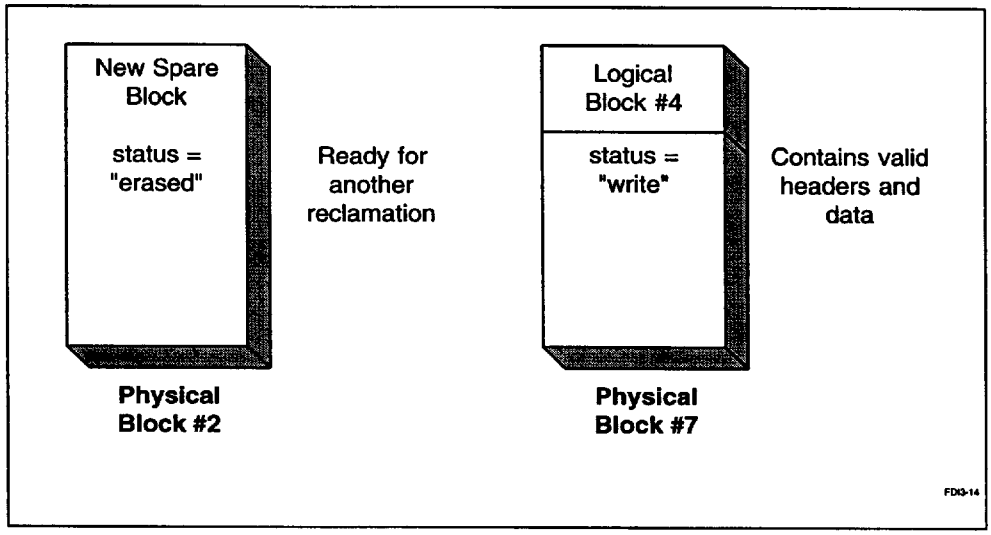

Figure 3-14. Status after Reclamation

Reclamation Call Format

```
void reclamation(void);
```

Input Elements

None.

Processing Characteristics

Flash is byte alterable. However, to rewrite a flash location which has already been written requires a block erase. Flash blocks are typically large – several Kbytes. Any valid information in a block to be erased must be moved from the block before "reclaiming" invalid flash memory by erasing the block. Reclamation copies all the valid data from a block with invalid data to reclaim to an empty "spare"block and then erases the block.

On Initialization installs reclamation as a task which runs in the background. The reclamation task pends on the reclaimEnable semaphore. When this semaphore is asserted by the system

through the Foreground API FDI_reclaimEnable function, the reclamation process begins. The following semaphores are used to control reclamation:

reclaimRequest:	BbkgndTask uses this semaphore to request the system to enable reclaim at the next available instant.
reclaimEnable:	The system asserts this semaphore to grant reclaim permission. The reclamation function awaits this binary semaphore.
reclaimDone:	The reclamation function uses this semaphore to indicate the completion of the reclamation process. If bkgndTask is pending on the reclaimDone semaphore, this indicates that memory is full and that it cannot continue the write or delete operation until reclaim is complete.

Reclamation selects the reclaim block by finding the block with the least amount of erase counts or the highest percentage of dirty space. Reclaiming the block with the least amount of erase counts distributes the erases across the blocks for maximum reliability. This process is called wear-leveling.

Reclamation first reads all the erase count information stored in each block and stores them in a local array and calculates the difference between the fewest erase counts and the most erase counts. If the difference is greater than a pre-defined value, reclamation reclaims the block with smallest erase count. If the difference is less than or equal to a pre-defined value, reclamation reads the invalid count of each block from the Logical Block table and selects the block with most invalid data.

Once reclamation selects the block to reclaim, reclamation writes the erase count value of that block as a data parameter with a unique identifier directly bypassing the Data Queue.

Then reclamation finds the spare block from the Logical Block table and marks the spare block to indicate the start of data transfer.

After finding the spare block, reclamation transfers the valid units from the reclaim block to the spare block. Once the data transfer is complete, reclamation writes the logical block number and the physical block number of the reclaimed block into the block information area of the spare block.

After completion of the data transfer, reclamation marks the spare block to indicate that the erase of the old block has begun.

Reclamation then erases the old block. Upon erase completion, reclamation retrieves the cycle count by doing a parameter read, increments it by one and writes this value into the block information area of the reclaimed block. The reclamation function then writes the complete block information into the spare block and marks it as a valid block. Then reclamation deletes the parameter cycle count to free up its RAM space. Finally, reclamation updates the logical block table and asserts the reclaimDone semaphore.

Error Handling

If during the process there is an error, a special semaphore that indicates that an error has occurred in the system is set by reclamation.

Utilization of Other Elements

None.

Limitations

Not applicable.

Output Elements

Identifier	Description	Data Type	Data Rep.	Limit / Range	Validity Check Perf.?	Output Method
reclaimDone	Indicate the completion of reclamation to the pending background write task if any.	semaphore			no	by reference
reclaimError	Indicate an error occurs during the reclamation process.	semaphore			no	by reference

3.4.2.3 BOOT CODE MANAGER SUB-SYSTEM

3.4.2.3.1 Power On Initialization

The initialization process initializes all the FDI control structures and performs necessary power loss recovery.

Initialization Call Format

```
int FDI_init(void);
```

Input Elements

None.

Processing Characteristics

FDI_init calls the initUnit routine to validate all the blocks and to validate all the units of each block. FDI_init also updates all the control structures and the global variables used by the FDI functions. FDI_init scans through all the Unit Header entries to build the Data lookup table by

scanning through all the Unit Header entries. FDI_init also builds the Logical Block Table scanning through the Block Information entries at the end of each block. Finally, FDI_init installs the Background Manager task as well as the reclamation task.

Error Handling

Any error during the initialization process will be returned to the application layer.

Utilization of Other Elements

FDI_init uses initUnit defined in Section 3.4.2.5.3, and dataQCreate in Section 3.4.2.4.1.

Limitations

Care must be taken to avoid repeated calls to FDI_init from the application or OS interface. Multiple calls to FDI_init can create undesired results, such as trashing existing global variables.

Output Elements

Identifier	Description	Data Type	Data Rep.	Limit / Range	Validity Check Perf.?	Output Method
error	Refer to the return codes in Section 3.5.	BYTE	count		no	function call

3.4.2.4 COMMAND DATA QUEUE SUB-PROGRAMS

The queue is implemented as a counting semaphore as well as a mutual exclusion semaphore. The queue functions implemented in this library are:

- dataQCreate–initializes the queue structure and necessary variables.
- dataQSend–adds the new command information to the equal priority queue.
- dataQReceive–reads the highest priority item from the queue.
- dataQDelete–removes the highest priority item from the queue.
- dataQPeek–scans through the queue and finds the matching command of same identifier and offset range.

3.4.2.4.1 Creating a Queue

This queue creating process initializes the queue element structure and the necessary variables.

dataQCreate Call Format

```
void dataQCreate(void);
```

Input Elements

None.

Processing Characteristics

DataQCreate initializes the queue structure and the necessary variables. This includes the variables that are used to maintain the semaphore protection for the queue, variable Q_Size which is used to limit the queue size to a user defined size. The system defined variable maxQSize is defaulted to 1K. The queue size includes the memory occupied by the queue structure as well as the data pointed by its elements. DataQCreate also sets the global pointer qhead_ptr to null. DataQCreate should be called once by the FDI_init function only during the initialization process.

Error Handling

Not applicable.

Utilization of Other Elements

None.

Limitations

None.

Output Elements

None.

3.4.2.4.2 Sending Information to the Queue

DataQSend puts the information into the data queue.

Queue Send Call Format

```
int dataQSend(CLEMENT_PTR Queue, BYTE priority);
```

Input Elements

Identifier	Description	Data Type	Data Rep.	Limit / Range	Validity Check Perf.?	Input Method
CELEMENT->commandData	Actual command structure.	COMMAND	structure	na	yes	
CELEMENT->ptrNextCmd	Pointer to the next command element on the command queue.	CELEMENT_PTR	pointer	na	yes	
CELEMENT->dataSize	The data container size.	WORD	count	na	no	
priority	Data priority.	BYTE	integer	na	yes	

Processing Characteristics

DataQSend puts the data command information into the data queue. DataQSend makes sure that the queue has been created successfully during the initialization process. This is done by verifying if the global variables are properly initialized. Then dataQSend makes sure that the node count has not yet reached the maximum count. DataQSend checks if adding this element will exceed the maxQSize, which is a pre-defined value. If so, returns an error code to indicate that the queue is full.

If the data and the command element will fit into the limit, dataQSend scans the queue in the order of highest to lowest priority looking for a match. If dataQSend does not find a matching priority element, it creates a priority element which also acts as a node to the same priority command queue. To create a new priority element, dataQSend allocates and fills the memory with the appropriate information from the buffer whose pointer is passed by the calling function. Then dataQSend adds a command element to the command queue of that priority. If dataQSend finds a match, it scans through the same priority looking for the last command element. Once dataQSend hits the last command element, it adds the new command element to the last command element. To add a new command element, dataQSend fills the allocated memory with the appropriate information from the buffer whose pointer is passed by the calling function.

queueProtect: DataQSend uses this semaphore to protect the queue from being accessed by any other task at the same time the queue pointers are being redirected.

queueCount: DataQSend uses the counting semaphore queueCount to keep track of the number of entries in the queue.

Error Handling

If any error occurs during this process, dataQSend returns an descriptive error code.

Utilization of Other Elements

Not applicable.

Limitations

Not applicable.

Output Elements

Identifier	Description	Data Type	Data Rep.	Limit / Range	Validity Check Perf.?	Output Method
error	Refer to the return code in Section 3.5.	BYTE	count		no	function call

3.4.2.4.3 Receiving Information from the Queue

This queue data read process receives the information from the queue.

Queue Receive Call Format

```
void dataQReceive(BYTE_PTR pbuffer);
```

Input Elements

Identifier	Description	Data Type	Data Rep.	Limit / Range	Validity Check Perf.?	Input Method
pbuffer	Pointer to the buffer in which data is read.	BYTE_PTR	pointer		no	by reference

Processing Characteristics

DataQReceive reads the information from the queue. The calling function passes the pointer to the buffer into which dataQReceive reads. DataQReceive returns a pointer to the Data Queue's qhead_ptr.

Error Handling

None.

Utilization of Other Elements

None.

Limitations

None.

Output Elements

None.

3.4.2.4.4 Deleting Information from the Queue

This queue delete process deletes the information from the queue.

Queue Delete Call Format

```
void dataQDelete(void);
```

Input Elements

None.

Processing Characteristics

DataQDelete deletes the first command element from the highest priority command queue from the data queue.

> queueProtect: DataQDelete uses this semaphore to protect the queue from being accessed by any other task at the same time the queue pointers are being redirected.

DataQDelete redirects the pointers to proper elements pulling out the first command element from the highest priority queue. If this is the only command element left in the queue before deletion, dataQDelete pulls out the first priority element also. After this pointer redirection the protection semaphore can be released. Then it frees the memory of the data container pointed by this command element. At last, it frees the memory occupied by the pulled out command element and the priority element if pulled out.

> queueCount: DataQDelete uses this counting semaphore to keep track of the number of existing elements in the queue. The semaphore count decrements automatically when this semaphore is given.

DataQDelete decrements the Q_Size by a total of the element size and the data container size.

Error Handling

None.

Utilization of Other Elements

None.

Limitations

None.

Output Elements

None.

3.4.2.4.5 Peeking through the Queue

DataQPeek scans the queue looking for matching type and identifier. If the command is WRITE, dataQPeek also looks for the matching offset range. DataQPeek returns a null pointer if it did not find a match. Otherwise, dataQPeek returns the pointer to the queue buffer of the matching element to the calling function.

Queue Peek Calling Format

```
int dataQPeek(CELEMENT_PTR Node, BYTE priority);
```

Input Elements

Identifier	Description	Data Type	Data Rep.	Limit / Range	Validity Check Perf.?	Input Method
Node	A pointer to a queue element from which peek should start peeking through or a null pointer if the scanning should start from the head of the queue.	CELEMENT_PTR	pointer	n.a.	yes	by reference
priority	Data priority.	BYTE	integer	n.a.	yes	by reference

Processing Characteristics

The calling function of dataQPeek function passes in two pointers. If the Node pointer passed in is a null pointer, dataQPeek scans the queue from the qhead_ptr in the order highest to lowest priority looking for a priority match that was passed in through the Queue pointer and finds a match. If the Node pointer is pointing to a same priority element, dataQPeek scans from that element. DataQPeek scans over the same priority command elements in the queue looking for a matching identifier field. DataQPeek returns the pointer to the queue buffer if it finds a match. Otherwise, dataQPeek returns a null pointer.

Error Handling

None.

Utilization of Other Elements

None.

Limitations

None.

Output Elements

Identifier	Description	Data Type	Data Rep.	Limit / Range	Validity Check Perf.?	Output Method
pointer	Pointer to the queue buffer or a null pointer is returned.	pointer	DWORD		yes	function call

3.4.2.5 SUPPORTING SUB-PROGRAMS

3.4.2.5.1 Finding Data

DataFind locates the first or next data parameter of the specified type or it will find a matched parameter using the type and identifier fields. DataFind fills the ptrBuffer with the last_data_found global structure if the ptrBuffer parameter is a non-zero.

Data Find Call Format

```
int dataFind(COMMAND_CONTROL *cmd_cntrl);
```

Input Elements

Identifier	Description	Data Type	Data Rep.	Limit /Range	Validity Check Perf.?	Input Method
cmd_cntrl->sub_cmd	Contains a flag to modify the functions action: GET_FIRST: finds the first data parameter of a given type. GET_NEXT: finds the matching data parameter of a given type and identifier. GET_MATCHED: finds the matching data parameter of a given type and identifier.	DWORD	flag	commands listed below: GET_FIRST GET_NEXT GET_MATCHED	yes	by ref.
cmd_cntrl->identifier	Unique data parameter identifier.	IDTYPE	integer	n.a.	no	by ref.
cmd_cntrl->type	Indicates a data type.	BYTE	integer	n.a.	yes	by ref.
cmd_cntrl->count	unused					
cmd_cntrl->offset	unused					
cmd_cntrl->actual	unused					
cmd_cntrl->aux	unused					
cmd_cntrl->ptrBuffer	reserved					

Processing Characteristics

DataFind handles three different cases of sub-commands.

If the sub_cmd is FIND_FIRST dataFind looks in the Data Lookup Table (Section 3.4.1.2.2) for the first type listed that matches the input field. If the cmd_cntrl->type parameter is an illegal value, dataFind returns ERR_NOTEXISTS.

DataFind uses the last_data_found structure's identifier field as the starting index into the Data Lookup table if the sub_cmd is FIND_NEXT. The Data Location structure in Section 3.4.1.2.8 defines the global last_data_found structure. From there dataFind looks for the next type listed that matches the input type. If the type parameter is an illegal value, dataFind returns ERR_NOTEXISTS.

If the sub_cmd is FIND_MATCHED dataFind indexes into the Parameter Lookup Table by the identifier input field. If there are no matching identifiers or types dataFind returns error ERR_NOTEXISTS.

The following is done in all cases: The Data Lookup Table locates the logical block and the Unit Header offset for the data. The physical block is located using the logical block number as an index into the Logical Block Table.

The physical block number and the offset define the location of the Unit Header within the flash memory. The Unit Header (Section 3.4.1.2.3) provides the location of the corresponding data within the block. DataFind fills the structure last_data_found with the information provided by the Unit Header structure.

Error Handling

If an error is returned by the dataFind function, the input parameter ptrBuffer is set to zero and the error is returned.

Utilization of Other Elements

The global structure last_data_found is from the Data Location Structure defined in Section 3.4.1.2.8.

Limitations

The FIND_FIRST sub-command must be executed before the FIND_NEXT sub-command otherwise an error will be returned.

Output Elements

Identifier	Description	Data Type	Data Rep.	Limit / Range	Validity Check Perf.?	Output Method
cmd_cntrl->ptrBuffer	Points to the last_data_found structure.	DWORD	DATA_LOCATION pointer	n.a.	no	by ref.
error	Refer to the return codes in Section 3.5.	BYTE	count		no	function call
cmd_cntrl->identifier	reserved					
cmd_cntrl->type	reserved					
cmd_cntrl->count	unused					
cmd_cntrl->offset	unused					

Identifier	Description	Data Type	Data Rep.	Limit / Range	Validity Check Perf.?	Output Method
cmd_cntrl->actual	unused					
cmd_cntrl->aux	unused					
cmd_cntrl->sub_cmd	reserved					

3.4.2.5.2 Read Unit

The readUnit function reads the data from the media into the input buffer.

Read Unit Call Format

```
BYTE readUnit(DWORD startloc, DWORD size, BYTE *buffer);
```

Input Elements

Identifier	Description	Data Type	Data Rep.	Limit / Range	Validity Check Perf.?	Input Method
startloc	Indicates the start location of the unit from which the data is to be read	DWORD	block offset location		no	by reference
size	Indicates the size in bytes to be read	DWORD	number of bytes		no	by reference
buffer	Indicates the address of the buffer into which the data is read	BYTE *	pointer		no	by reference

Processing Characteristics

ReadUnit goes to the starting location in flash and writes size bytes of data into the input buffer.

Error Handling

A descriptive error is returned if there is any problem during execution of this function.

Utilization of Other Elements

Not applicable.

Limitations

Not applicable.

Output Elements

Identifier	Description	Data Type	Data Rep.	Limit / Range	Validity Check Perf.?	Output Method
buffer	Data is read into this buffer.	BYTE *	pointer		no	by reference
error	Error while reading this unit cause this value to be returned.	BYTE	count		no	function call

3.4.2.5.3 Unit Initialization

The initUnit validates all the blocks and all the units in each block.

Unit Initialization Call Format

```
int initUnit(void);
```

Input Elements

None.

Processing Characteristics

The initUnit function scans all the available blocks in the media to verify the validity of each block by reading the Block Information structure. If initUnit encounters a block in the process of being erased, initUnit completes the erase and prepares the blocks for reuse. InitUnit also scans all internal data management structures and performs any necessary power loss recovery.

Error Handling

Any error during the initialization process will be returned to the API layer.

Utilization of Other Elements

None.

Limitations

None.

Output Elements

Identifier	Description	Data Type	Data Rep.	Limit / Range	Validity Check Perf.?	Output Method
error	Refer to the return codes in Section 3.5.	BYTE	count		no	function call

3.4.2.5.4 Downloading RAM Code

The downloadCode function loads the Interrupt and Status Polling code into RAM at Initialization. The low-level function RAM_flashModify is literally copied directly from its code location in flash and copied to RAM.

Downloading RAM Code Call Format

```
void downloadCode(void);
```

Input Elements

Identifier	Description	Data Type	Data Representation	Limit / Range	Validity Check Perf.?	Output Method
RAM_START	Pointer to RAM destination.	defined address	address pointer		n.a.	defined at compile time

Processing Characteristics

The downloadCode function marks the beginning and end of the low-level function by storing the RAM_flashModify function pointer as the begin pointer and the next function pointer as the end pointer. The low-level function RAM_flashModify is copied byte by byte from the begin pointer to the end pointer, into the RAM destination address.

Error Handling

Not applicable.

Utilization of Other Elements

The function RAM_flashModify and the next function in memory.

Limitations

None.

Output Elements

None.

3.4.2.5.5 Interface to Low Level Functions

The flashLowLevel function executes the low-level polling function located in RAM.

RAM Interface Call Format

```
BYTE flashLowLevel(BYTE function, BYTE_PTR ptrAddr,
                   WORD length, WORD offset, BYTE_PTR ptrData);
```

Input Elements

Identifier	Description	Data Type	Data Representation	Limit / Range	Validity Check Perf.?	Input Method
function	Type of command erase or program.	BYTE	integer value	0, 1	yes	by value
ptrAddr	Pointer to the address in flash.	BYTE *	address pointer	n.a.	no	by reference
length	Size of data to write.	WORD	integer value		no	by value
offset	Offset into data to write.	WORD	integer value		no	by value
ptrData	Pointer to the data in RAM.	BYTE *	address pointer		no	by reference

Processing Characteristics

The input parameters for flashLowLevel describe the data item and the flash location for the polling function. FlashLowLevel places the input parameters into a LOWLVL_INFO structure (Section 3.4.2.2.2) and calls the RAM_flashModify function located at the RAM_START address in RAM to perform the command.

Error Handling

Returns forwarding errors to the calling function.

Utilization of Other Elements

None.

Limitations

None.

Output Elements

Identifier	Description	Data Type	Data Rep.	Limit / Range	Validity Check Perf.?	Output Method
status	Returns the status of the command upon completion.	BYTE	unsigned char	0 - okay 1 - suspended 2 - an error	no	function call

3.5 RETURN ERROR CODES

The following list of error codes may possible be returned from any interface function of the FDI system

Return Error	Code	Meaning
ERR_NONE	0	Indicates command was successful
ERR_READ	1	Indicates an error reading the flash component
ERR_WRITE	2	Indicates an error in the status register when writing to the flash component. Potentially due to a locked block if the flash component has this capability.
ERR_PARAM	3	Indicates an incorrect parameter to a function.
	4	Reserved for future use.
ERR_OPEN	5	Indicates an operation is attempted on an open file when the file should be closed.
ERR_EXISTS	6	Indicates the attempt to create a file or directory that already exists.
ERR_NOTEXISTS	7	Indicates an error in attempting to perform an operation on a file that does not exist.
ERR_QFULL	8	Indicates the Data Queue is full and cannot accept additional elements.
ERR_SPACE	9	Indicates that there is no available clean flash space left. This indicates that a manual reclaim needs to occur before re-attempting the command.
	10	Reserved for future use.
ERR_NOTOPEN	11	Indicates an error in performing an operation on a file that is not open but must be to complete the operation.
ERR_ERASE	12	Indicates an error erasing the flash block. Potentially due to a locked block if using flash with this capability.
	13	Reserved for future use.
	14	Reserved for future use.
ERR_MAX_PARAMS	15	Indicates that the maximum number of objects has been reached.
	16	Reserved for future use.
	17	Reserved for future use.
	18	Reserved for future use.
	19	Reserved for future use.
	20	Reserved for future use.
	21	Reserved for future use.

Return Error	Code	Meaning
ERR_FORMAT	22	Indicates the detection of an error in the card format structures.
ERR_MEDIA _TYPE	23	Indicates the media type is identified as a media that is unsupported (such as RAM if RAM_SUPPORT is disabled)
ERR_NOT_DONE	24	Indicates that a function was aborted before completion due to an abort capability such as that used in the REVERSE_SEEK_SETUP.
	25	Reserved for future use.
	26	Reserved for future use.
	27	Reserved for future use.
	28	Reserved for future use.
	29	Reserved for future use.
ERR_WRITE_PROTECT	30	Indicates the media is write protected. Returned at in initialization (which should not be treated as an error, only a flag) and when a write is attempted.
ERR_DRV_FULL	31	Indicates that the flash media is full.
ERR_MAX_OPEN	32	Indicates that the maximum number of objects open consecutively has already been reached. This is determined by the MAX_OBJECTS define in type.h.
	33	Reserved for future use.