Flash Data Integrator (FDI)
User's Guide

1997

Intel Corporation
P.O. Box 7641
Mt. Prospect, IL 60056-7641

or call 1-800-879-4683

CG-041493

1

Technical Overview

CHAPTER 1
TECHNICAL OVERVIEW

Many personal communications devices (e.g., cellular telephones and pagers) have a need to store both data and code. This problem has been solved using multiple memory devices (Figure 1-1). Although viable, this approach is neither cost-effective nor sensitive to critical power constraints. System integration is necessary not only to reduce cost and form factor but also to foster product differentiation. Furthermore, consumer demand for ease-of-use and convenience is pushing data requirements to new levels. These factors, and others, have fueled the need for a monolithic flash memory device capable of simultaneous read while write (RWW) operation.

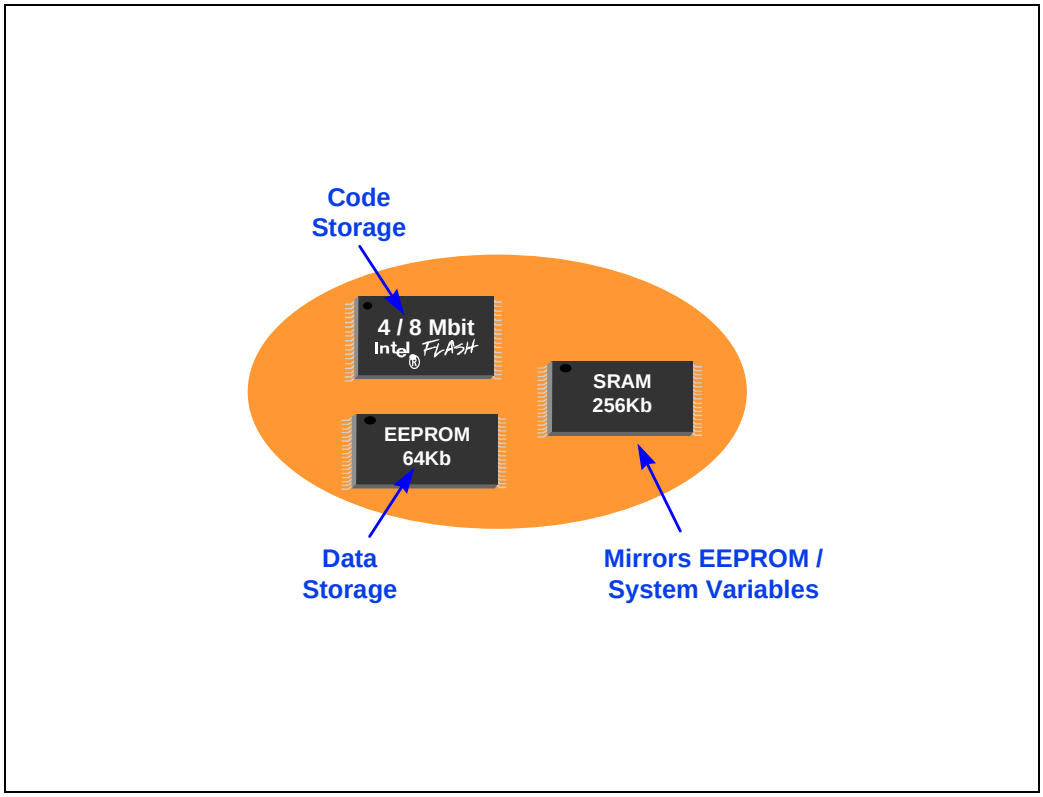


Figure 1-1. Typical Memory Subsystem in Today's Designs

The market need for read-while-write flash memory is clear; however it is questionable whether a hardware partitioned flash device is the best choice to satisfy this need. There are numerous hardware RWW flash devices available today (Figure 1-2). These hardware devices lack the flexibility to adapt as code and data needs change. Moreover, when these devices are used in a real-time operating environment, the flash device still needs to be managed (since it is writeable on a byte basis but eraseable only on a block basis) In addition, a hardware flash device can be up to 20% more costly than standard flash memory.

Market analysis shows there is a superior alternative solution which meets code and data storage requirements without additional flash memory cost. This solution incorporates the management of flash memory within a real-time environment. The Intel Flash Data Integrator provides the most cost-effective and flexible solution for code plus data storage applications.

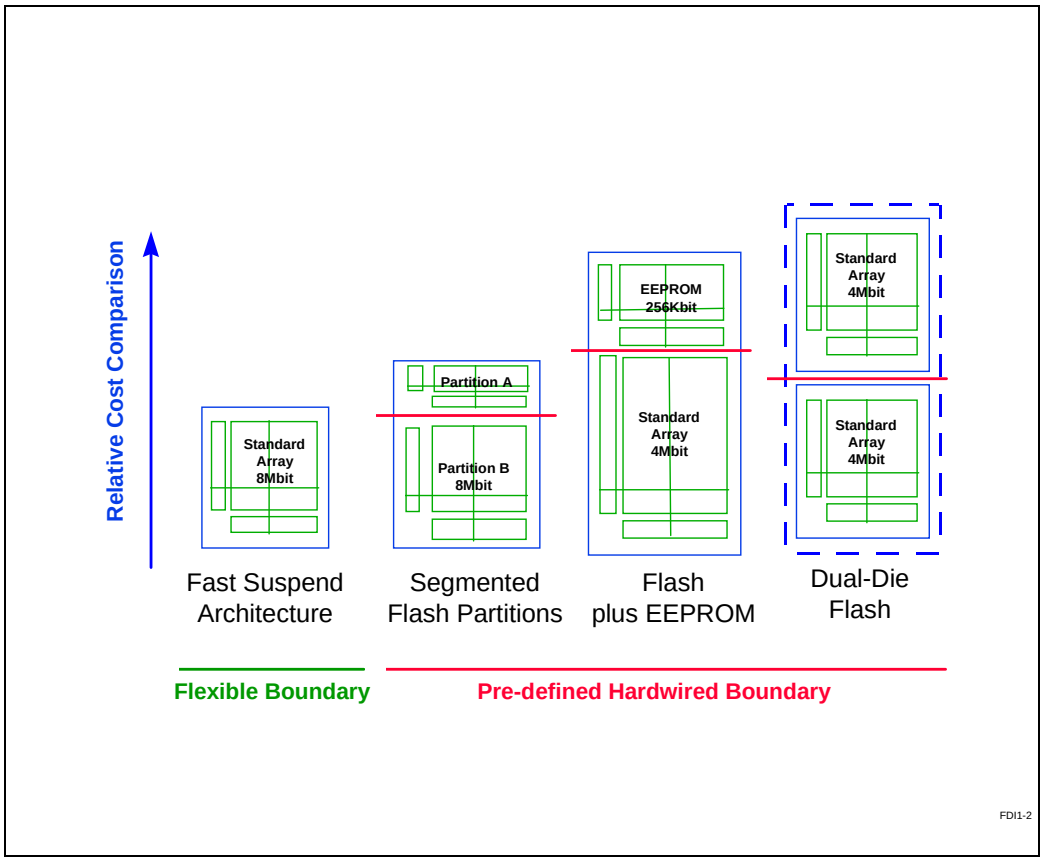


Figure 1-2. Relative Cost and Die Sizes of Differing Approaches to Code Plus Data Storage

The Flash Data Integrator (FDI) is a royalty-free flash media manager that enables code reads from flash while data writes are occurring. This is accomplished with a deterministic program and erase suspend (a feature available on Intel's new Smart 3 and Smart 5 flash memory products). If an application can manage a maximum 20 µs context switch latency, it can realize a code plus data storage solution within a single flash memory device with FDI and standard flash—without the expense of a hardware RWW flash device. FDI background flash management can be "held-off" during the execution of time-critical code segments.

To evaluate whether FDI meets critical system requirements, Chapter 2 of this manual presents a technical analysis of how FDI with standard flash can be used to replace a EEPROM in a GSM cellular phone application. Over 84% of the processing time available for doing background flash management; therefore an erase operation that typically takes 1 second would take about 1.2 seconds during a call. Chapter 2 gives an excellent overview of the FDI architecture and how it is applicable to many environments. The commands and structures that make up FDI are explained in Chapter 3. Chapter 4 describes Intel's FDI test platform for GSM cellular phone applications.

Appendix A contains a list of frequently asked questions (FAQs) for FDI. Information on technical support for FDI is available in Appendix B. Appendix C provides an overview of the system requirements for implementing FDI. A full description of the FDI license agreement is given in Appendix D.

2

EEPROM Replacement with Flash Memory

CHAPTER 2
EEPROM REPLACEMENT WITH FLASH MEMORY

2.1 INTRODUCTION

Flash memory is used in a wide range of applications for embedded control code storage· Many of these applications including cellular phones, modems, automobile engine control and others also use a separate EEPROM to store factory, system, and/or user data. With ever-increasing pressure to eliminate components and reduce system cost, designers are looking to use flash memory to emulate EEPROM for simultaneous code and data storage.

Intel introduced an EEPROM emulation methodology based on linked data list structures that was successful in applications such as automobile engine control. However, in time-synchronized applications like the cellular phone, the inability of flash memory to write during an erase suspend operation and the undeterministic maximum write and erase flash timing may have prevented EEPROM emulation using standard flash components in certain market segments. Time-critical applications, such as cellular telephones, must service system interrupts by providing access to processor code stored in flash while simultaneously supporting data writes to flash. For this reason, research has been undertaken by Intel and others to develop a simultaneous read operation while writing or erasing another flash memory partition (block). Specialized components have been proposed to support simultaneous read and write operations, but they incur from 10- to 20-percent increase in silicon die area due to redundant circuitry, and have not been manufactured in volume production. Although specialized circuits enable simultaneous read-while-write (RWW) operation, the added cost is less attractive in cost-critical, high-volume manufactured applications. New hardware-assisted suspend/resume circuitry with fast latency offer a technically-feasible approach to emulate simultaneous RWW operation without the cost impact of specialized circuits.

Regardless of specialized flash circuits, flash media management software is required to manage the larger (8 or 64 Kbytes) flash memory block partitions. This is true, since flash memory cannot be erased on the byte level common to memory such as EEPROM, but must be erased on a block granularity. The development of a flash memory manager requires a keen understanding of flash technology and data management methods. Fortunately, Intel has designed the necessary flash media manager, known as Flash Data Integrator (FDI), which handles variable length parameter storage, while utilizing hardware-assisted circuitry to emulate simultaneous code execution. This new method handles power loss recovery in the case of battery removal during data storage, providing a reliable EEPROM replacement.

This paper describes the hardware and software architecture necessary to emulate EEPROM memory in flash. Section 2.2 reviews the fundamentals of flash and EEPROM technology. Critical new timing parameters and hardware limitations are examined, along with a description of new hardware suspend to read/write capabilities common to many standard flash architectures. The software architecture for EEPROM emulation in flash memory is reviewed

in Section 2.3. The software modules and features are also discussed in Section 2.3. Resource and system requirements are presented in Sections 2.4 and 2.5. Parameter cycling is characterized in Section 2.6, and power loss recovery techniques are described in Section 2.7. Section 2.8 reviews the flexibility of extending FDI to support enriched data storage and remote code updates.

2.2 MEMORY FUNDAMENTALS

2.2.1 Memory Architecture

Flash memory technology offers the electrical erasability of random access memory (RAM), and nonvolatility of read only memory (ROM) to retain information after power is removed. Unlike RAM, flash cannot be erased on a byte basis. Flash memory supports writing (programming), the processes of changing a logic “1” to a “0,” on a byte or word [double byte] basis. Certain flash memory components, including those from Intel, have the added capability to be programmed one bit (or multiple bits) at a time.

Erasing flash is the process of changing a logical “0” to a “1” on a block-by-block basis. Physical block partitioning is set by a fixed address range of the component (see Figure 2-1). Typical block sizes range from 8 Kbytes to 64 Kbytes for parameter and main blocks, respectively.

Flash memory stores data as charge on the floating gate of a single transistor as compared to other memory types that require additional components to hold a charge or the state of a latch circuit. As a result, flash has significant silicon area and cost advantages when compared with other memory types (see Table 2-1), offering a cost-effective means of storing data.

2.2.2 Program/Erase Timing

EEPROM supports byte alterability by rewriting a page, typically 16, 32 or 64 bytes. The system must wait 10 ms to allow time for the data to be written to the EEPROM cell in the background. This limits EEPROM write times from 157 µs to 625 µs/byte or 12.5 Kb/s to 49.7 Kb/s. Flash memory, on the other hand, supports data writes at a continuous 17 µs/byte (22 µs/word) typical 2.7 volt program time in the foreground, thereby supporting data write rates up to 710 Kb/s and reducing the amount of time a system spends writing data from 93% to 98%. Continuous data programming may be essential for streaming data packets such as short message service (SMS), fax, or digitized voice recording. This also doesn’t account for any overhead time lost rewriting an entire page in EEPROM when only a single byte update is required—providing even a further reduction in data write time overhead.

Unlike flash, EEPROM does not require a block erase operation to free up space before data can be rewritten. This means that some form of software management is required to store data in flash. However, EEPROM technology is also limited to a maximum number of data writes [cycles] between 10,000 and 100,000. Flash memory, on the other hand, does not experience a device cycle until the block is erased. This means flash improves cycling reliability on the order of hundreds of times better than EEPROM technology. The details of parameter cycling is discussed in Section 2.5.

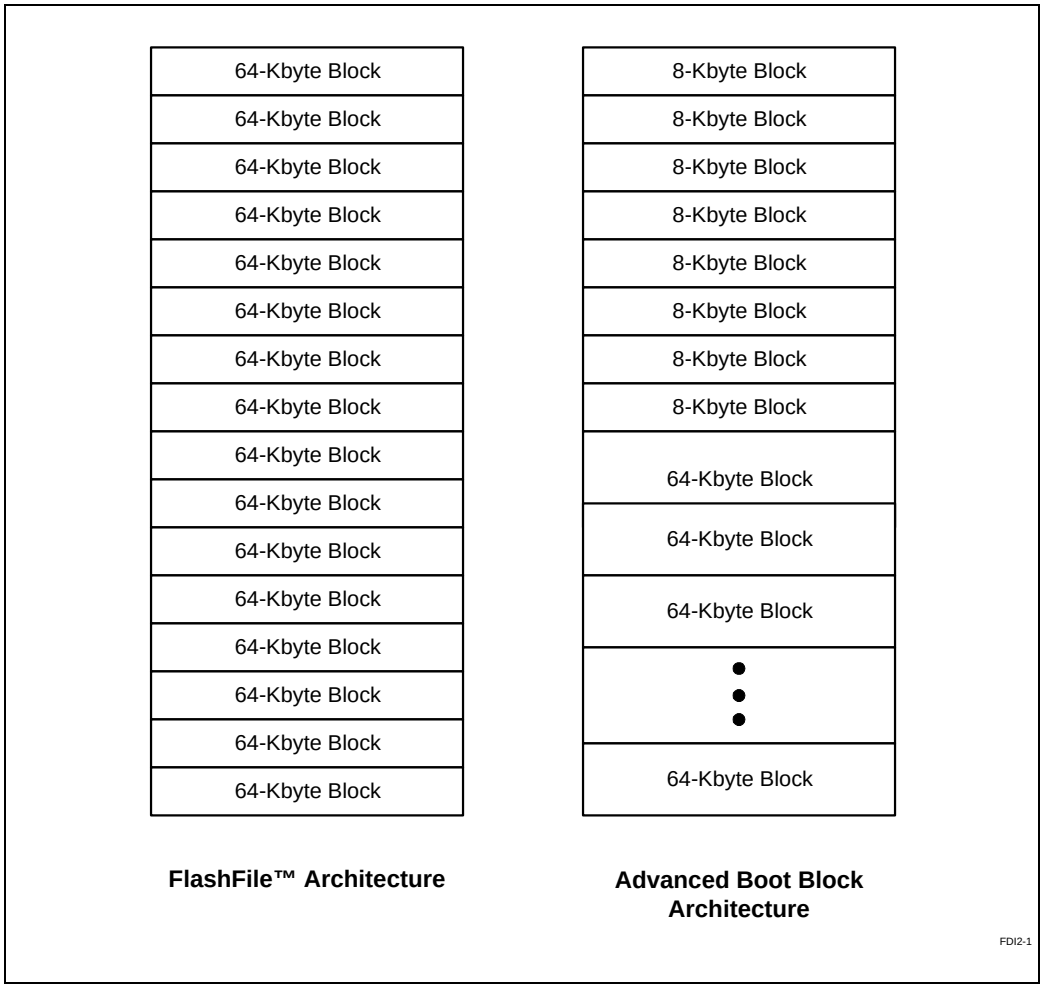


Figure 2-1. Intel's Flash Memory Architectures

Table 2-2 compares the timing specifications of flash and EEPROM memory. Although the write performance of flash technology is fast compared to EEPROM, it is important to consider maximum program time. The time it takes to reliably store charge on the floating gate in flash memory is a function of process variation, temperature, voltage, and electron storage susceptibility. Under worst case conditions it may take as long as 170 µs to store a byte (200 µs for a word), as given by the specification t_{WHQV1} and t_{WHQV2}, respectively (see Table 2-3). The maximum time, however, does not occur across each of the flash cells, and is only realized in a single or few cells within a given address range. When writing a single byte or word, one should account for this maximum time t_{WHQV1} and t_{WHQV2}, respectively. However, when writing a page of data to flash memory the maximum write time is dependent on the page size and is given by the graph in Figure 2-2.

The erase time of the flash parameter and main blocks are given by the specification t_{WHQV3} and t_{WHQV4}, respectively (see Table 2-3). The distribution of erase time is similar to that of write times and are dependent on operating conditions and cycling. Erase times remain semi-constant for erase cycles less than 10,000. Above 10,000 cycles the erase time increases as illustrated in Figure 2-3. The manufacturer's specified cycling parameter is based on a given erase and program time. Flash can be reliably cycled beyond the specified value provided the design accommodates an increase in the erase and program time. For example, a flash device with specified 10K erase cycles can operate with 100K cycles with a typical erase time of 1.5 sec.

Fortunately, engineers need not design systems to wait the maximum specified values. Instead flash components commonly contain internal Status Registers that indicate when a program or erase operation is complete. By polling the internal register, the designer can determine when an operation is completed and the memory is available for another operation such as read.

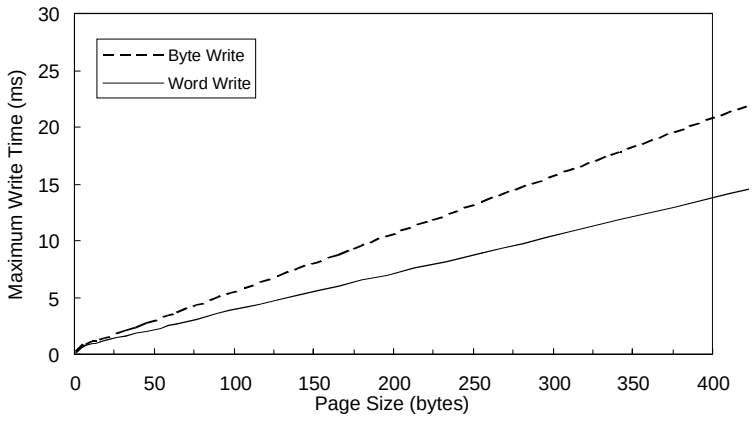
Table 2-1. Die Area Comparison of Memory Technology

Features	Flash	DRAM	EEPROM	SRAM
Cell Components	1 Transistor	1 Transistor + 1 Capacitor	2 Transistor	4 Transistor + 2 Resistor
Cell Area (µm²) [0.4µ lithography]	2.0	3.2	4.2	22
Chip Area (mm²) (16-Mbit density)	61	98	107	59 (1-Mbit Density)
Read Speed (ns)	80 (5V) 120 (3V)	60	150	<60

Table 2-2. Comparison of Flash and EEPROM

Features	Flash	EEPROM
Write Time (Typical)	10 µs / Byte (5V) 17 µs / Byte (3V)	10 ms / 16, 32 or 64 Byte Page [157–625 µs/Byte]
Erase Time (Typical)	800 ms/8-KB Block (5V) 1000 ms/8-KB Block (3V)	NA
Internal Program/Erase Voltage	5V/12V (PSE) 5V/–10V (NGE)	5V/21V
Cycling	10–100K **Erase** Cycle/Block 10–300M Write Cycle/Byte[1]	10–100K **Write** Cycle/Byte

NOTE: 1. See Section 2.6.



NOTE:
Initial characterization; subject to change based on device validation.

Figure 2-2. Maximum Write Timing
(V_{CC} = V_{PP} = 2.7V–3.6V, T_A = –40°C to +85° C, 10K Cycles)

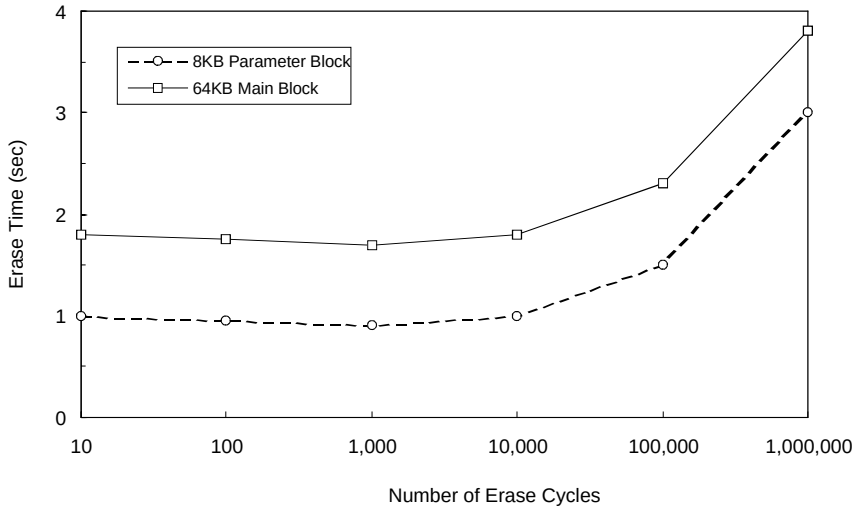


Figure 2-3. Cycling Effects on Erase Time
(V_{CC} = V_{PP} = 2.7V–3.6V, T_A = –40°C to +85°C)

Table 2-3. Flash Memory Erase and Program Timings(3,4)

Sym	Parameter	Notes	V_{PP} = 2.7V		V_{PP} = 12V		Unit
			Typ[1]	Max[5]	Typ[1]	Max[5]	
t_{WHQV1} t_{EHQV1}	Word Program Time	2	22	200	8	185	µs
t_{WHQV2} t_{EHQV2}	Byte Program Time	2	17	170	8	155	µs
t_{BWPB1}	Block Program Time (Word) (Parameter)	2	0.10	0.30	0.03	0.10	sec
t_{BWPB1}	Block Program Time (Byte) (Parameter)	2	0.16	0.48	0.07	0.21	sec
t_{BWMB2}	Block Program Time (Word) (Main)	2	0.80	2.40	0.24	0.80	sec
t_{BWMB2}	Block Program Time (Byte) (Main)	2	1.28	3.84	0.56	1.7	sec
t_{WHQV3} t_{EHQV3}	Block Erase Time (Parameter)	2	1	5.0	0.8	4.8	sec
t_{WHQV4} t_{EHQV4}	Block Erase Time (Main)	2	1.8	8.0	1.1	7.0	sec
t_{WHRH1} t_{EHRH1}	Word/Byte Program Suspend Latency Time to Read		6	10	5	6	µs
t_{WHRH2} t_{EHRH2}	Erase Suspend Latency Time to Read		13	20	10	12	µs

NOTES:

1. Typical values measured at T_A = +25°C and nominal voltages. Subject to change based on device characterization.
2. Excludes external system-level overhead.
3. These performance numbers are valid for all speed versions.
4. Characterized but not 100% tested.
5. Maximum values are based on typical process skews. Subject to change based on device characterization.

2.2.3 Specialized Flash RWW Circuits

Most currently available flash technology must complete a program or erase operation before code can be read from another memory block. Based on the maximum program/erase timing specifications of flash, there is a common misconception that EEPROM emulation can be supported only when the application can mask interrupts and allow a write or erase operation to complete. In time-synchronized applications with maximum latencies in the range of microseconds, such as a cellular phone, simultaneous operation may be difficult without some form of hardware assistance.

Many approaches to hardware assisted read-while-write (RWW) operation have been proposed for flash architecture (see Figure 2-4). One approach is to segment the standard memory array into separate physical partitions by duplicating the row and column (x/y) decoders, sense amplifiers and charge pump circuits—adding 12% to 17% to the silicon die area, and component cost. This form of hardware-assisted flash memory allows code to be read from one memory block, while a program or erase operation executes simultaneously in another block in the opposite physical partition.

Simultaneous read with background program/erase has higher peak power dissipation, but the total energy may be the same as the standard architecture. Segmented flash partitions further require that the data and code fit completely within the fixed partitions—making the selection of the partition size critical. If either code or data requirements exceed the maximum partition limit then simultaneous operation is no longer possible when data and code reside in the same partition, and the maximum suspend latency timing of the component must be considered. Although this method is attractive for EEPROM emulation, it does not offer the flexibility to support growing data needs.

On the other hand, a segmented architecture does minimize program/erase to read latency (see Table 2-5). This reduces the effect on system timing and may reduce testing if the design is time-critical.

An alternative approach is based on packaging two standard flash die into a single “dual-die” package. The “dual-die” approach supports simultaneous operation between the two die with the added flexibility that the size of the data or code partition can be changed to meet the needs of the application. Unfortunately, total peek memory system power is twice the power of a single standard flash component. Dual-die packaging, or two separate flash components, is attractive when the data requirements exceed 2 Mbits to 4 Mbits (256 KB to 512 KB). EEPROM emulation alone requires far less parameter storage needs, 8 Kbytes to 32 Kbytes, making a dual-die solution less cost-attractive.

A third approach to RWW is to combine EEPROM technology onto a two-transistor (2T) flash memory process. This approach eliminates the need for media management software, but has the disadvantage of increased memory system power and cost. Memory power dissipation can be as high as 200 mW compared to standard flash memory at 60 mW. The increase in die area necessary to support both memory technologies has an adverse impact on die yield and in turn product cost.

Yet another approach to hardware assistance is enhanced suspend circuits that allow program/erase operations to be suspended temporarily to read code from another partition. Suspend circuits allow time critical operations to be serviced without stalling the microprocessor (CPU). Unlike the other specialized RWW approaches, suspend circuits do not place limits on the code/data partition size, thereby increasing the flexibility and offering support as data storage needs grow. Suspend circuits do not increase the flash die size (cost), nor do they increase memory system power. The following section describes the suspend-resume operation in more detail.

Table 2-4 summarizes the comparison of the various RWW memory architectures.

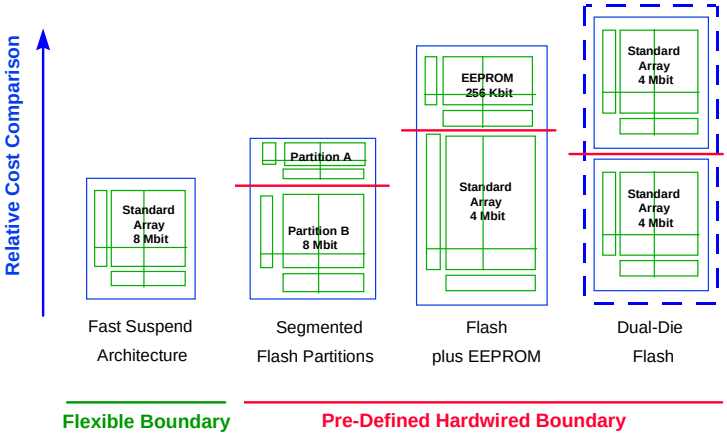


Figure 2-4. Comparison of Standard and Specialized Flash Memory Architectures for RWW

Table 2-4. Comparison of Hardware Assisted Flash Memory Architectures

Attribute	Advanced Boot Block Flash	Segmented Partitions Flash	Dual-Die Flash	2T Flash Plus EEPROM
Die Size (mil/side)	254 (8 Mb)	315 (8 Mb)	380 (two, 4-Mbit die)	345 (4 Mb + 256-Kbit EEPROM)
Process Lithography	0.4 µm	0.5 µm	0.4 µm	0.6 µm
Min. Operating Voltage (Read/Write)	2.7V / 2.7V	2.7V / 4.5V	2.7V / 2.7V	4.5V / 4.5V
Max. Read Pwr.[1]	60 mW	<90 mW	105 mW	200 mW
Max. Program/Erase Pwr.[1]	120/82.5 mW	120/1220 mW	235/260 mW	200 mW
Max. Standby Pwr.[1]	150 µW	300 µW	60 µW	1,500 µW
Max. Latency to Read	10 to 20 µs	1 µs	120 ns	300 ns

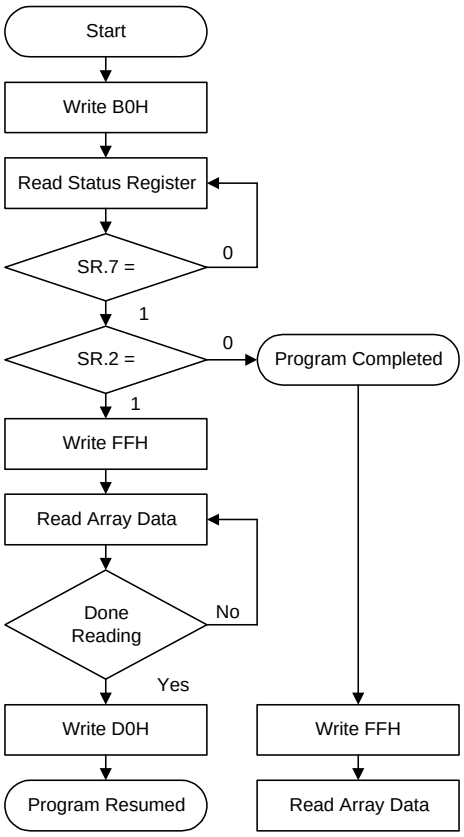
NOTE:

1. Assumes nominal 3.0V or 5.0V read or program voltage, and 5 MHz data rate.

2.2.4 New Hardware Assisted Suspend to Read/Write

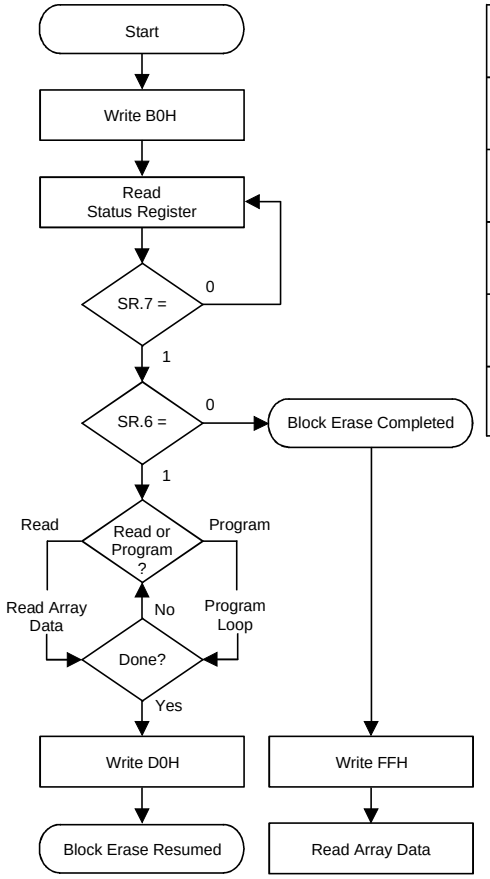
Intel's 0.4µ ETOX™ V flash process technology components include two suspend commands; **Program and Erase Suspend**. Program and erase suspend mode allows system software to suspend both the word/byte program or block erase command in order to read from or write data to another block. Commands are written to the Command User Interface (CUI), connecting the microprocessor and the internal chip controller, using standard microprocessor write timings. Issuing a program or erase suspend command will begin to suspend a program/erase operation. The flash internal Status Register will indicate when the device reaches program/erase suspend mode. In this mode, the CUI will respond only to the Read Array, Read Status Register, Program Resume, and Erase Resume commands. Flash specification t_{WHRH1} and t_{WHRH2} define the program and erase suspend latency, respectively (Table 2-3).

After a Program or Erase Resume is written to the flash memory, the flash device will continue with the program or erase process, respectively, (see Figures 2-5 and 2-6). The flash continues from the point where the suspend command was issued, eliminating the need to repeat the program or erase operation. The suspend to read/write operation provides a maximum latency of 10/20 µs, respectively, and allows system designers to emulate simultaneous RWW operation within the time constraints of the systems.



Bus Operation	Command	Comments
Write	Program Suspend	Data = B0H Addr = X
Read		Status Register Data Toggle CE# or OE# to Update Status Register Data Addr = X
Standby		Check SR.7 1 = WSM Ready 0 = WSM Busy
Standby		Check SR.2 1 = Program Suspended 0 = Program Completed
Write	Read Array	Data = FFH Addr = X
Read		Read array data from block other than the one being programmed.
Write	Program Resume	Data = D0H Addr = X

Figure 2-5. Program Suspend/Resume Flowchart



Bus Operation	Command	Comments
Write	Erase Suspend	Data = B0H Addr = X
Read		Status Register Data Addr = X
Standby		Check SR.7 1 = WSM Ready 0 = WSM Busy
Standby		Check SR.6 1 = Block Erase Suspended 0 = Block Erase Completed
Write	Erase Resume	Data = D0H Addr = X

FDI2-6

Figure 2-6. Block Erase Suspend/Resume Flowchart

2.3 FLASH DATA INTEGRATOR SOFTWARE STRUCTURE

Software is required to make all flash memory components, regardless of RWW circuitry, emulate an EEPROM. The approach of storing data on a nonvolatile media is well understood, but intricate due to the need to overcome the conditions described previously.

Intel has developed an open software architecture, known as the Flash Data Integrator (FDI), that enhances the flash technology. FDI allows the system designer to use a single low-cost flash memory component as a storage medium for both system code and data in real-time systems. This section describes the FDI flash media management software and reviews basic flash data management techniques.

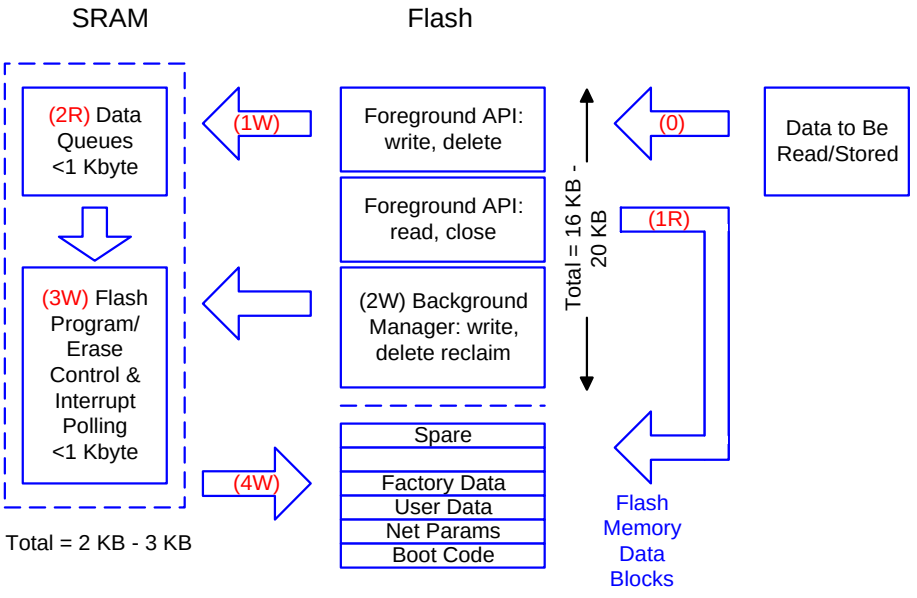
2.3.1 Flash Data Integrator Functional Overview

The FDI architecture consists of three major subsystems; the Foreground Application Programming Interface (API), Background Manager, and Boot Code Manager.

System tasks and interrupts that need to store data, interface to the Foreground API functions. Through the API interface, commands that modify flash, and their corresponding data are queued by the system. The API commands such as open, close, read, write, and query are the interface between FDI and the system. The Background Manager reads commands/parameters from the queue, determines where the information should be stored, and performs any parameter storage or clean-up necessary while monitoring for interrupts. During initial system power-on the Boot Code Manager initializes the FDI control structures and performs any necessary power loss recovery.

Figure 2-7 illustrates the information flow between the system flash memory and SRAM. The system calls the foreground API function (0), with a command and data. The API function either, queues the command and data into SRAM for operations which modify flash (1W), or executes the command directly for commands which do not modify flash (1R). The FDI Background Manager (2W), executing out of flash memory, manages the queued tasks during available processor time. During a flash program or erase operation (4W), interrupts with vectors in flash are disabled and control is turned over to a small routine (less than 1 Kbyte) in SRAM (3W). This routine polls interrupts while monitoring progress of the program or erase operation. If a higher priority interrupt occurs, the polling routine suspends the flash memory program or erase operation, and allows the interrupt handler to then execute directly from flash memory. Upon completion of the interrupt routine, the flash program or erase operation is resumed by the SRAM polling routine. The Background Manager continues in this fashion until all events are handled and all necessary cleanup is complete.

A complete description of the FDI subsystems is provided in Chapter 3, *FDI Architecture and API Specification.*



NOTES:
R = Read
W = Write

Figure 2-7. FDI Information Flow between Flash and SRAM Memory

2.3.2 EEPROM Parameter Types

Parameters stored in the EEPROM in a cellular phone can be characterized as either **factory, network or end-user data**. These parameter records vary in size and frequency of updates. For example, factory tuning data may be a long record (few hundred bytes) that is written to the EEPROM during the manufacturing process and may only be updated on an infrequent basis when the user brings the phone into a service center. On the other hand, the call timer parameter keeps track of the duration of a call and may be updated as often as every couple seconds during the process of the call. Table 2-5 lists the data parameter types commonly stored in the EEPROM of a cellular phone.

The frequency of parameter updates determines how often parameter blocks must be cleaned up [erased], to ensure free space is always available in flash for data writes. The write occurrence combined with the system time allocated for flash management and the timing parameters of the flash memory should be evaluated. Based on the low data write rate of cellular phone EEPROM data, flash memory with hardware assisted suspend/resume circuitry provides adequate timing to emulate an EEPROM and respond to system interrupts. It should be feasible

to manage all flash memory program and erase operations during normal phone operation (e.g., during “dead” time while on a control channel or during the a phone call). This maximizes the time the CPU remains in sleep mode.

Table 2-5. EEPROM Data Parameters

Parameter Type	Size (Bytes)	Number	Amount (Bytes)	Occurrence
Factory	1–300	<10	~1,024	1–2 times/year
Network	5–20	25–50	~1,024	< few times/day
End-User	20–250	30–250	~6,144	Every few seconds during call

2.3.3 Parameter Storage and Management

Unlike EEPROM, flash memory cannot be erased on a byte basis. By using software management techniques, data can be stored on a byte or variable length basis and flash erase operations can be completed using a suspend command to emulate byte alterability.

Data parameters are stored and tracked by software as virtual units within the physical boundaries of the flash block (see Figure 2-8). This is required whether specialized RWW circuits are available or not. Since a byte in flash may not be overwritten, an old occurrence of a parameter is marked “dirty” when the parameter is updated. The valid parameter is written to the next available memory location. The software media manager tracks the valid occurrences and controls access when requested by the system.

Parameters are stored until there is not enough “clean” space available in the block to insure new records can be written without over flowing the block. When this point is reached, the latest occurrence of each parameter is transferred to a clean [erased] block. Block header records associated with each parameter block indicates the status of the block. That is, information such as if the block is active [containing valid data], if the block is transferring data, or if the block is erased. After the valid parameters are transferred, the original block is marked for clean-up [erasing]. The parameter storage and management process if handled fully by FDI, and may be suspended by the system to write data provided free space is available.

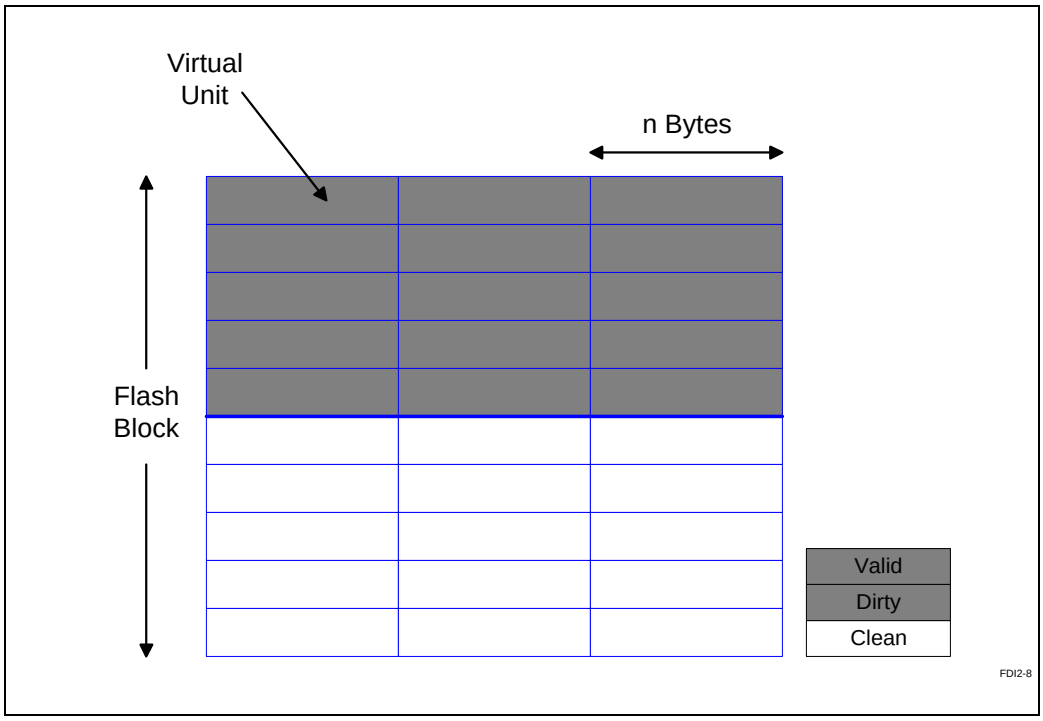


Figure 2-8. Flash Media Manager

2.3.4 Read Latency

Unlike previous EEPROM emulation techniques that were based on a linked list approach, FDI uses a look up pointer to the parameter header to access the data. This provides uniform latency and simplifies system timing issues. (See *AP-604 Using Intel's Boot Block Flash Memory Parameter Blocks to Replace EEPROM*.)

2.3.5 Real Time Interrupt Support

To support real time interrupts, the flash management operations are suspended before servicing an interrupt. Upon completion of the interrupt routine, the flash operation is resumed until complete.

During a flash program or erase operation, a system interrupt hardware register is polled while waiting for the flash command to complete. If an interrupt occurs, the program or erase command is suspended and the interrupt handler is executed directly from flash memory after the maximum latency of the flash component (20 µs in the case of Intel's Advanced Boot Block Flash). This eliminates the need to store interrupt handlers in SRAM. Upon completion of the interrupt, control is returned to flash program/erase routine to resume the operation.

Since non-specialized RWW flash components cannot be read from during a program or erase operation, a small (less than 1 Kbyte) software handler in the system's static RAM (SRAM) is required. The SRAM and other system requirements are described in Section 2.5.

2.3.6 FDI Features

- Ability to easily integrate data and code into a variety of digital cellular environments. All areas of the code that require porting should be very limited and the code does not depend on the existence of non-ANSI 'C' libraries.
- Ability to suspend all data management activity when requested to execute code.
- Ability to resume data management activity following code execution.
- Ability to migrate the SW developed for standard flash architecture with hardware suspend/resume capability and flash components that support specialized RWW circuitry.
- Ability to support all EEPROM data storage requirements in the initial release.
- Ability to support enriched (larger) data types in future releases. These could include phone directories, audio recordings, or code updates. The initial release will not support these data types, however, the architecture is planned with this in mind.
- Ability to trade-off the features with the flash/RAM requirement.
- Low latency parameter/file read access.
- Support for variable parameter sizes without large overhead in media.
- Power-off recovery capabilities. An unexpected power loss should never corrupt or lose data. Replacement of old data with new should always provide the old data as a back-up until the new data can be guaranteed.
- Supports symmetrical block sizes through a portion of the flash component, but allows the block size to be definable at compile time.
- Flexible through use of defines, compile time options, or parameter options.

2.4 DEVELOPMENT RESOURCES

Although basic flash data management techniques may be well understood by the system software engineer, the work necessary to develop a reliable system takes significant time and resources. This effort has delayed many OEMs from fully emulating the EEPROM in flash. Fortunately, Intel's FDI solution greatly reduces the OEM's development effort.

Table 2-6 provides an estimate of the development resources required to integrate Intel's FDI into an existing system compared to developing an internal media manager for a flash memory component using a specialized RWW flash component. Intel's FDI may reduce the development time by 82%, allowing the OEM to bring the product to market faster.

Although additional time may be necessary to test future software changes that effect system timing. This may or may not be significant depending on the latency requirements of the system.

Table 2-6. Projected EEPROM Emulation Development Time

Feature	Intel FDI	Internal Media Manager
FDI Definition	included	500 devl.-hours
Flash Parameter Storage Management	included	600 devl.-hours
Flash Storage Management Reclaim	included	600 devl.-hours
EEPROM Interface	included	80 devl.-hours
Power Loss Recovery	included	480 devl.-hours
Flash Suspend/Resume Interface and Testing	included	N/A
API Integration	400 devl.-hours	N/A
Total	**400 devl.-hours**	**2,260 devl.-hours**

2.5 SYSTEM REQUIREMENTS

2.5.1 Random Access Memory Requirements

Some amount of system RAM (SRAM) is required to provide instructions during flash program and erase operation. The amount of RAM usage is dependent on the specific features needed. The size of this code is expected to be 2 Kbyte, 1 Kbyte for queue storage and less than 1 Kbyte of code. RAM should be used for queuing of events/data. The goal is to create a modular set of reference code, where the cellular phone OEM can pick and choose the various features needed in their product, and thus tailor the software to their specific product needs. Flash memory with specialized RWW circuitry does not have the requirement for available RAM as the component can provide instructions to control operation within the separate partition.

2.5.2 Flash Memory Requirements

Flash memory space will be necessary to store the Foreground and Background media manager program code. This is required for all flash memory types, standard and specialized. Intel's FDI media manager should require 16 Kbytes to 20 Kbytes of flash memory.

In addition, the flash memory component must include program and erase suspend commands (such as those in Intel's Advanced Boot Block components) or include specialized RWW circuitry as described in Section 2.2.3.

2.6 PARAMETER CYCLING

Intel's flash memory is specified to work over 100,000 erase cycles when operating over 0°C to +70°C, between 20,000 and 30,000 over the range of –25°C to +85°C, and 10,000 over the extended temperature range of -40°C to +85°C.

A cycle is defined as an erase operation, and not the number of data writes the device can support. For example, an 8-Kbyte block supports 8,192 byte writes before a single erase operation, one cycle, has completed. Therefore, parameter cycling is a function of the parameter size. This is important when determining how many parameter updates can be supported. Today, many OEMs limit writes to EEPROM due to 100K write cycling limit of EEPROM technology. Parameter updates in EEPROM over write the previous instance. The maximum life of the EEPROM is thereby limited to the update rate of the most frequently written parameter (e.g., call timer in the case of a digital cellular phone. Using flash for EEPROM emulation extends the effective number of data cycles.

The effective number of write cycles is dependent on the number of available parameter blocks and size of the parameter record and is given by:

$$Eff.\ Write\ Cycles = \frac{available\ bytes/block\ x\ no.\ blocks}{parameter\ record\ size} x Max.\ erase\ cycles/block$$

Assuming two 8-KB flash blocks are used to store a 5-byte record over an extended temperature range, and further assuming 5 Bytes/block status and 512 Bytes/block overhead results in:

$$Eff.\ Write\ Cycles = \frac{(8{,}192 - 5 - 512\,Bytes/blk)x2\,Blks}{5Bytes/\ parameter} x10{,}000 cycles/blk$$
$$= 30{,}700{,}000$$

This is a 300 times improvement over EEPROM memory that is limited to 100,000 write cycles. The same approach works for variable size records, where the effective write cycles are determined by the summation of the occurrences of the various records.

2.7 POWER LOSS RECOVERY

Power loss is handled in a reliable manner by adding a status field to the header of each data parameter block, as well as each parameter. The status field indicates that a parameter update has been initiated or the write was complete. If power is lost during a parameter update, the status is known when power is restored. Upon power recovery, the initiation process should check the status of each parameter. If the status indicates that a parameter update began but did not complete successfully, then the record can be marked invalid. The same process is used during clean-up operations when valid data is moved to a clean block. Because of the fast write

capability of flash, critical parameters can be stored to flash sooner than an EEPROM component, thereby improving the robustness of the system.

To improve system power-on performance, the initiation process may be suspended, provided free space in flash is maintained.

2.8 ENRICHED DATA STORAGE AND REMOTE CODE UPDATES

Specialized RWW flash architectures with fixed size data partitions are limited in their ability to manage data that exceeds the partition size. Intel's FDI software is designed to manage a multiple number of memory blocks, offering a more flexible solution when combined with a component that is not limited by a physical partition, such as the Intel Advanced Boot Block flash memory.

FDI architecture supports extensions to manage enriched, streaming data types such as digitized voice, fax, company phone directories, and more. The architecture also enables the ability to remotely manage code modules stored in the main memory blocks.

2.9 CONCLUSION

A low-cost, flexible and reliable approach to EEPROM emulation in flash memory was presented for real time applications such as cellular phones. The approach is based on flash memory management software, known as the Flash Data Integrator (FDI), that emulates EEPROM functionality while enabling the flexibility for future data sotrage needs. This method reduces system cost, improves system write times by as much as 98%, supports data write rates up to 710 Kb/s, reduces memory system power by as much as 140 mW (compared with specialized RWW components), reduces development time by as much as 82%, and can increase parameter cycling 300 times over EEPROM memory. Power loss recovery techniques ensure data is not lost or corrupted in the event of power loss, eliminating the need for battery backed SRAM. EEPROM emulation in flash requires limited system resources depending on the needs and selected flash technology. Intel's FDI flash media management software and Advanced Boot Block flash memory offer a cost-effective, robust, reliable, and flexible solution to EEPROM replacement.

3

FDI Architecture and API Specification

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.

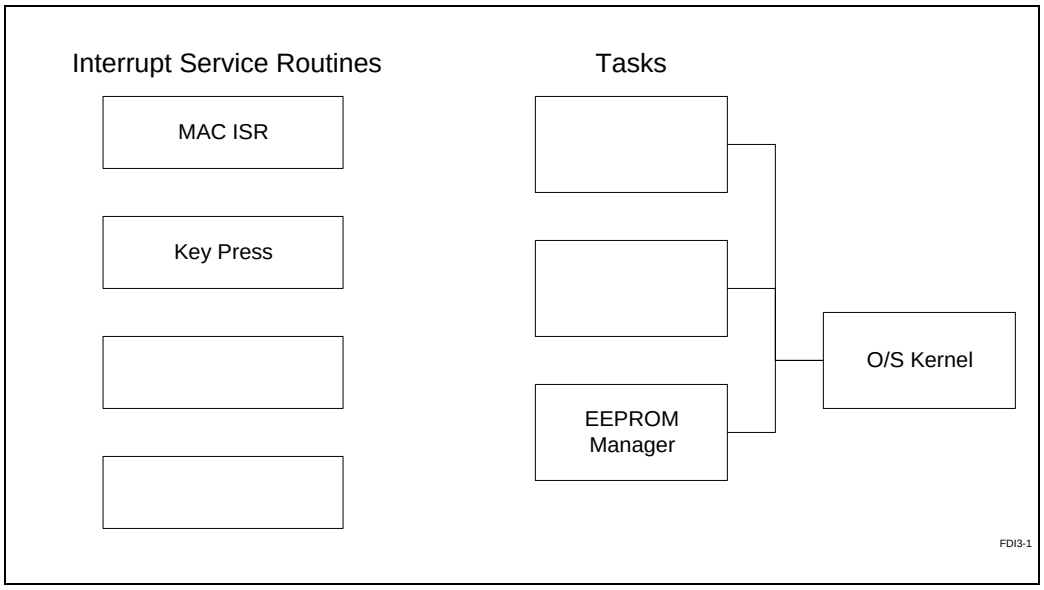


Figure 3-1. Simplified System Software Architecture

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.

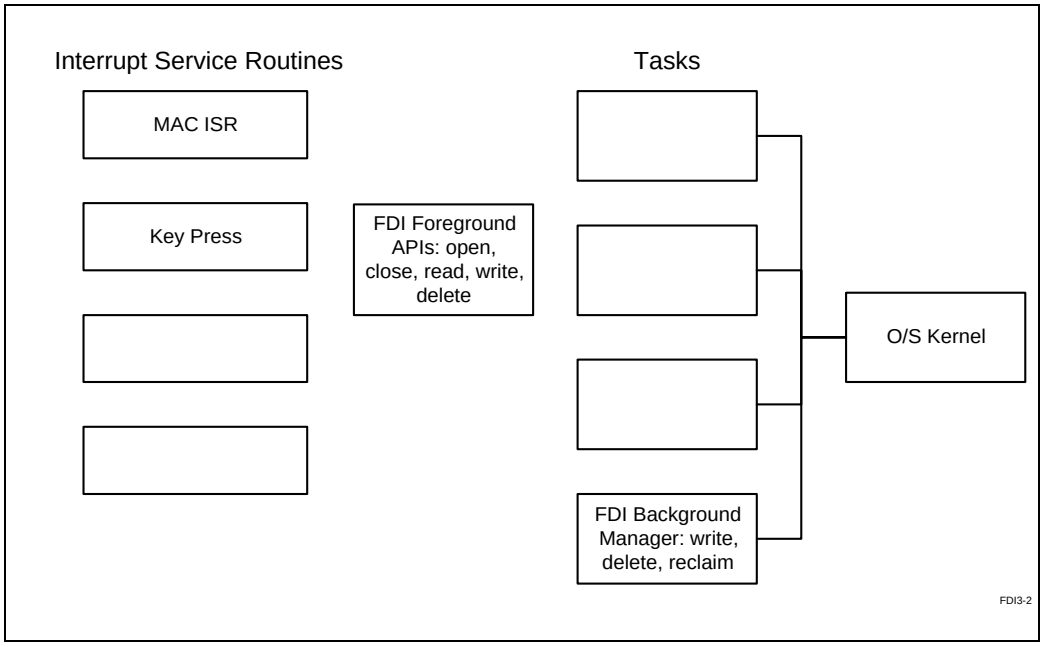


Figure 3-2. Simplified System Software Architecture Using FDI

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.

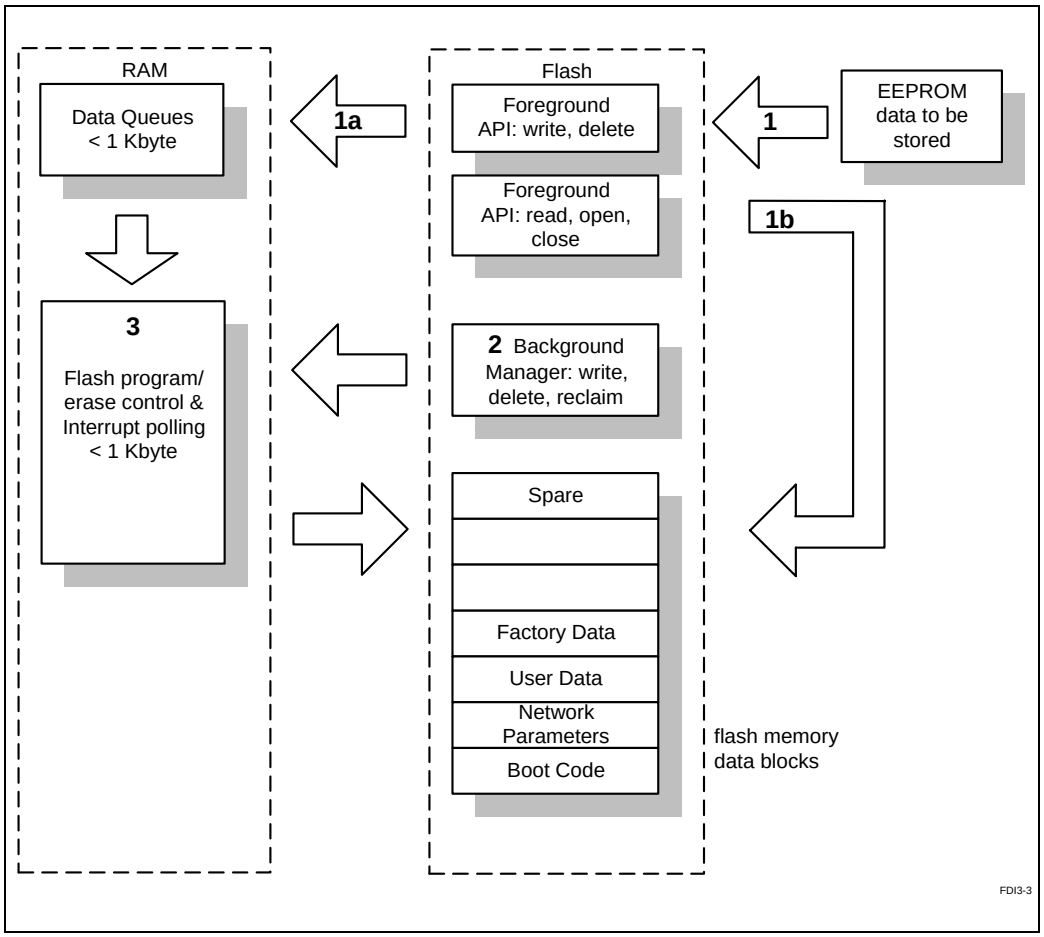


Figure 3-3. Software Flash Data Integrator Data Flow Diagram

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.

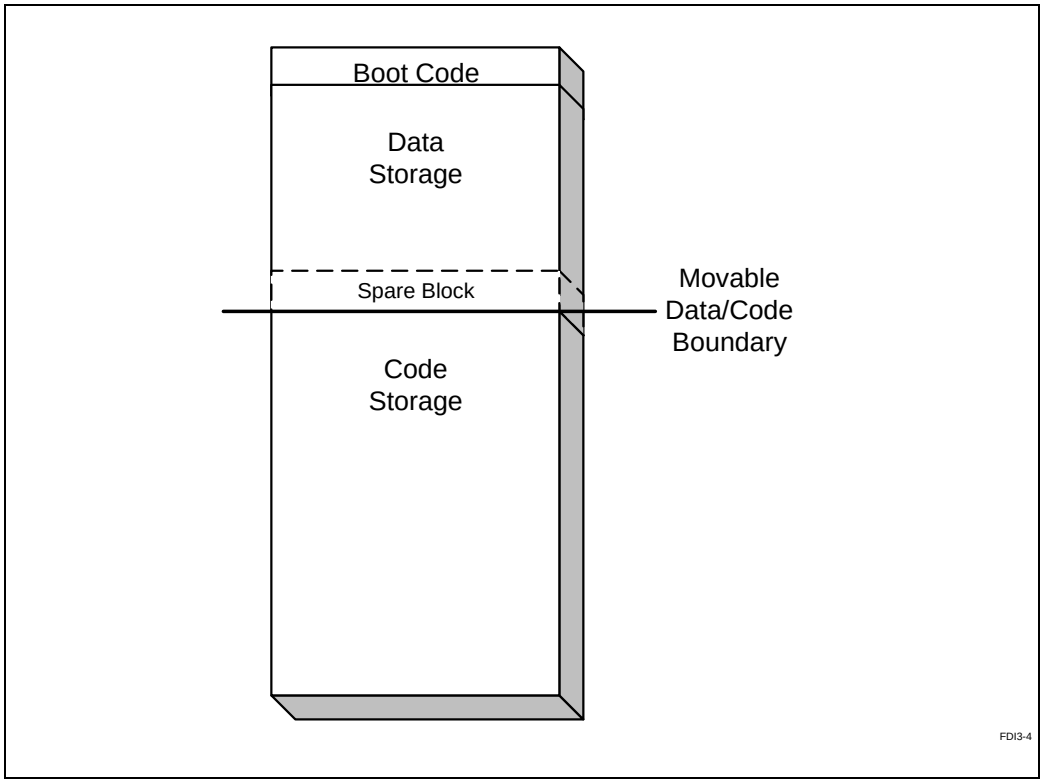


Figure 3-4. Code + Data Storage Arrangement in Flash

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.

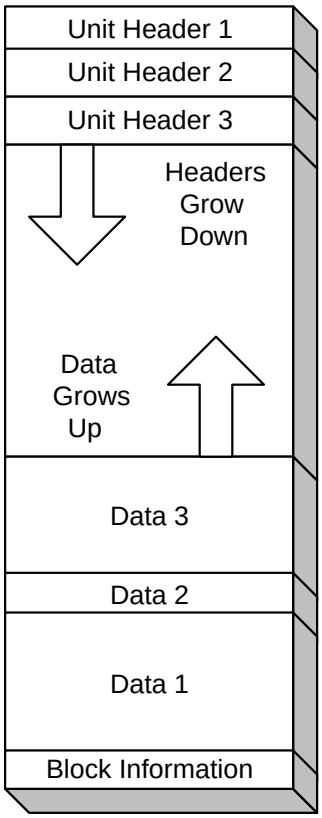


Diagram of a single block showing the relationship between Unit Headers and Data.

FDI3-5

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.

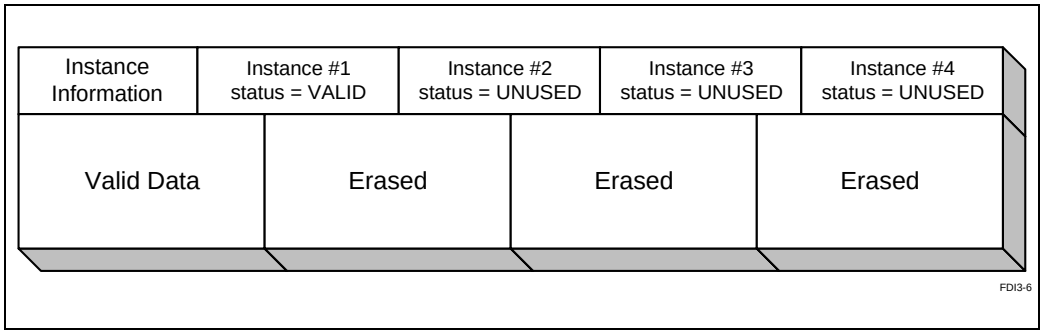


Figure 3-6. Example of Multiple Instances

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.”

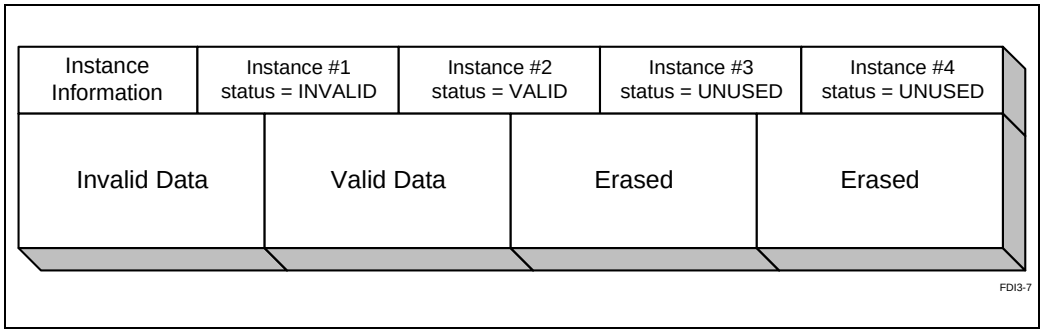


Figure 3-7. Parameter Update with New Instance

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.

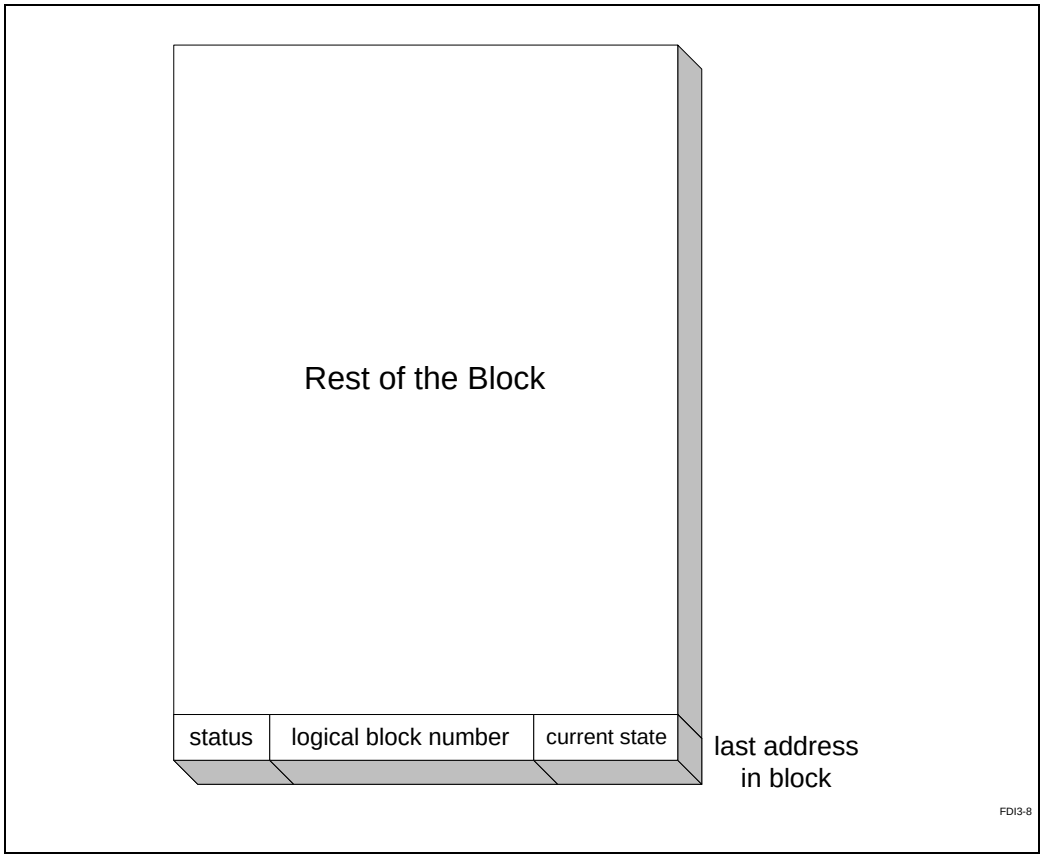


Figure 3-8. Placement of Block Information

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.

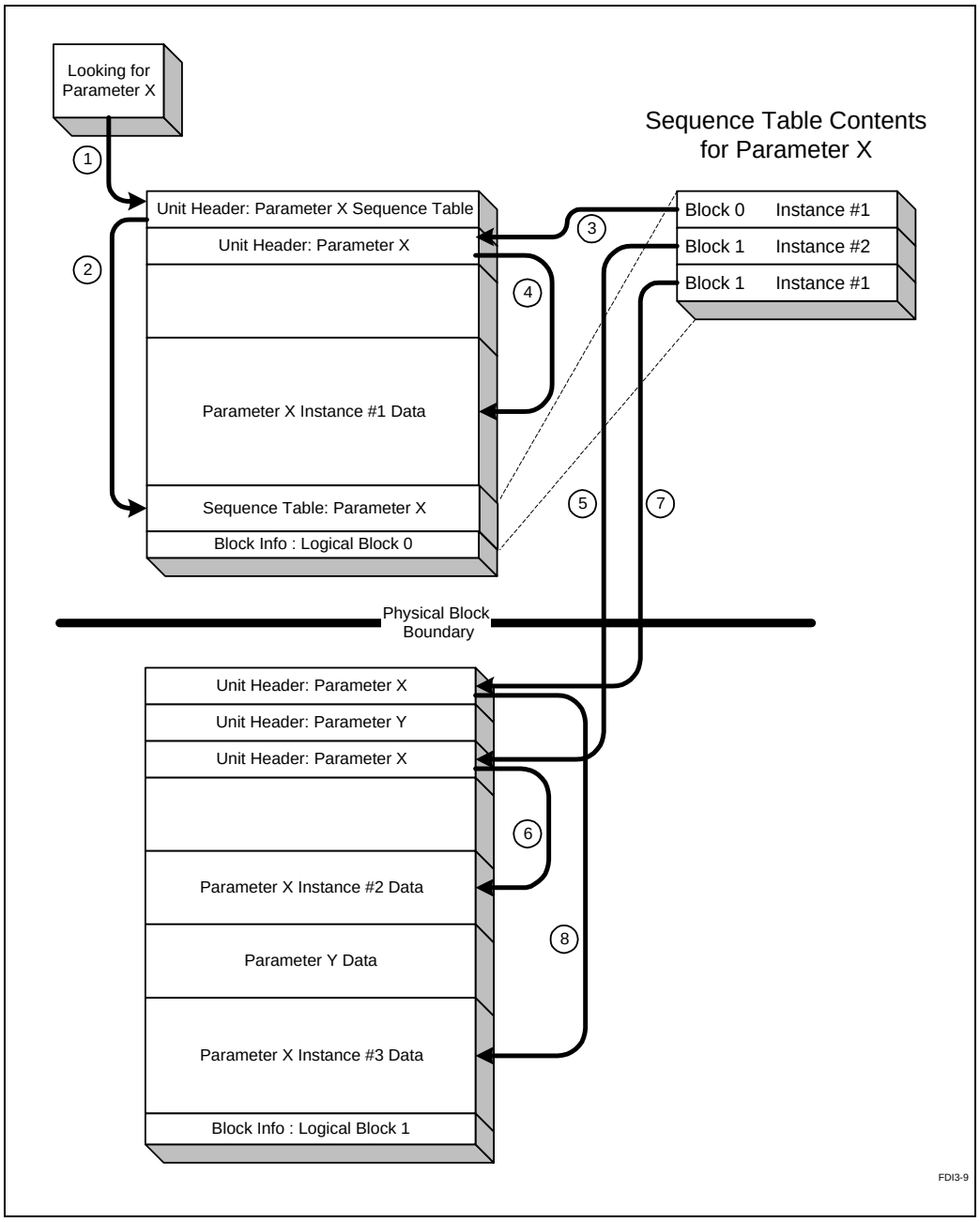


Figure 3-9. A Sequence Table Example

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.

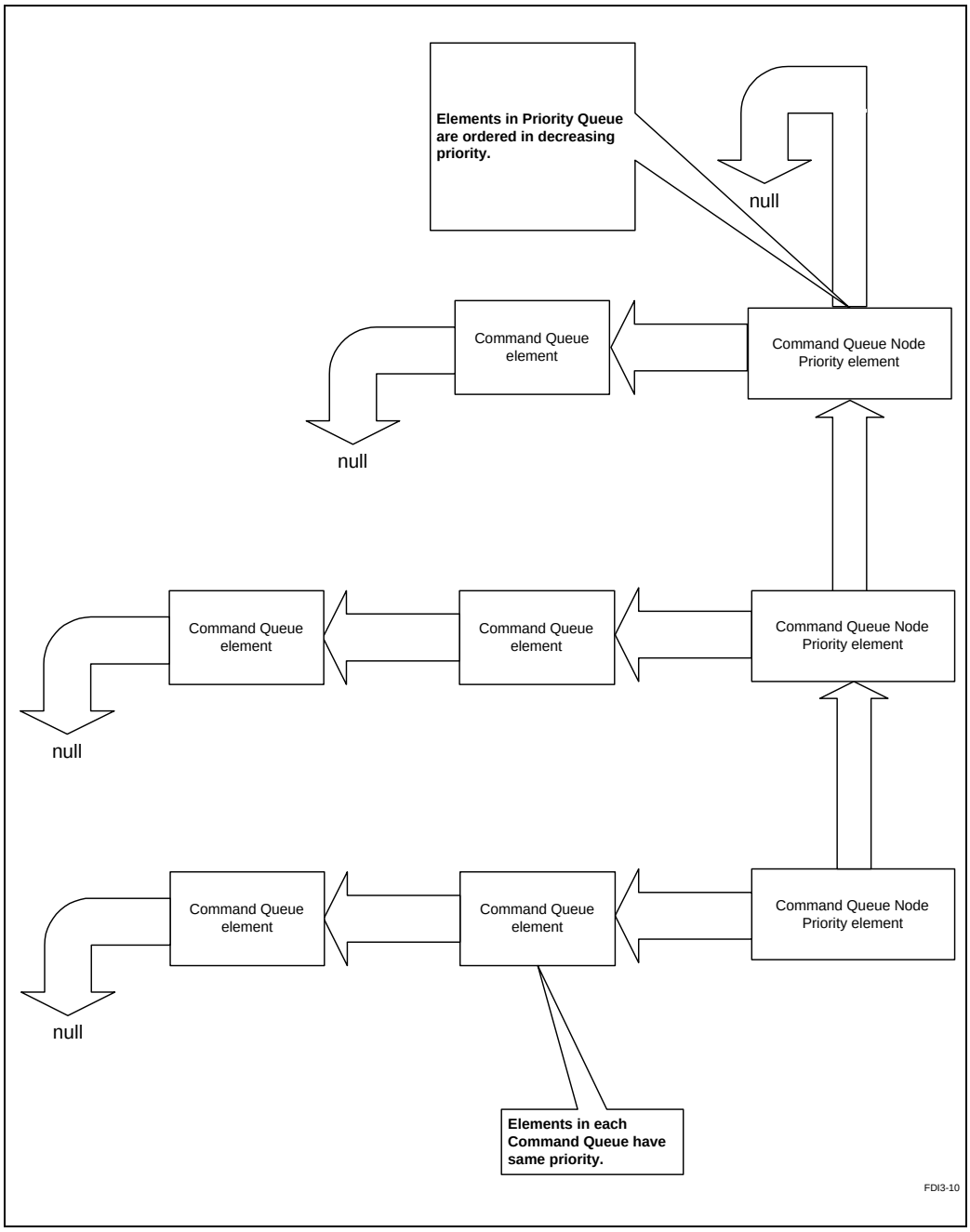


Figure 3-10. Data Queue Structure

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_LO-CATION 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.

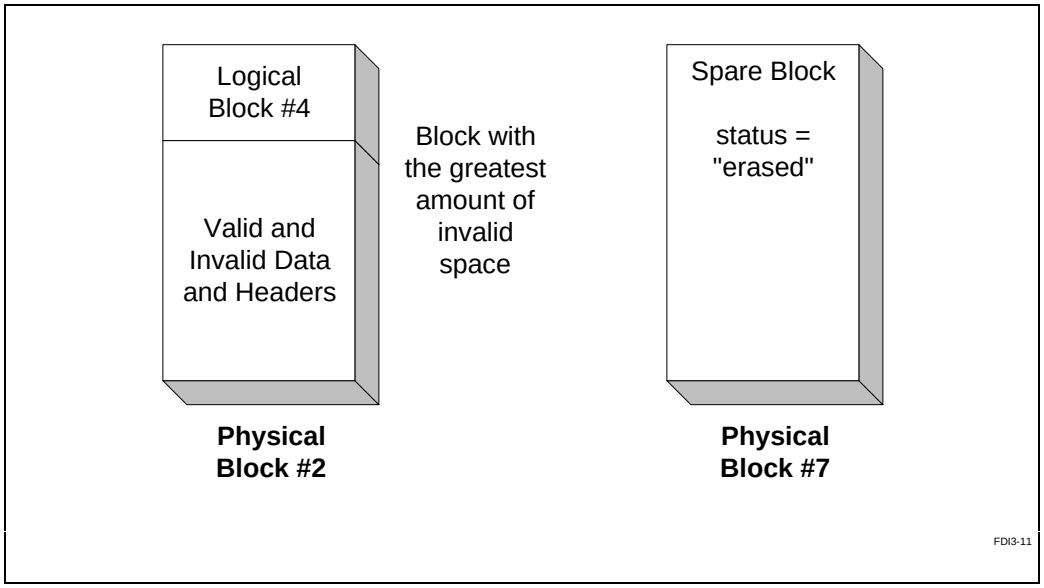


Figure 3-11. Reclaim Candidate and Spare

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.

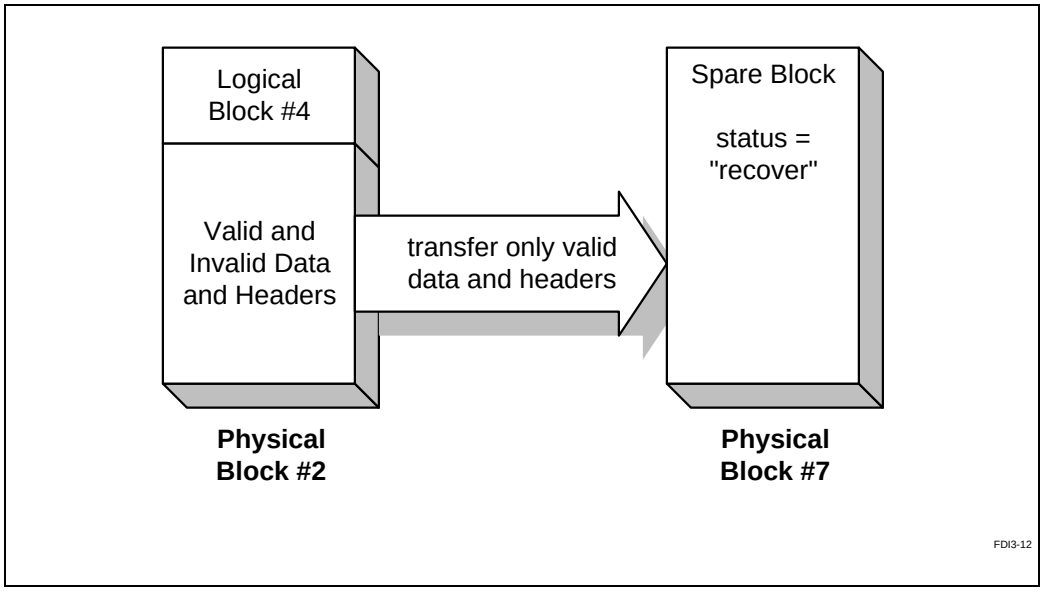


Figure 3-12. Transferring Valid Data to the Spare Block

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.

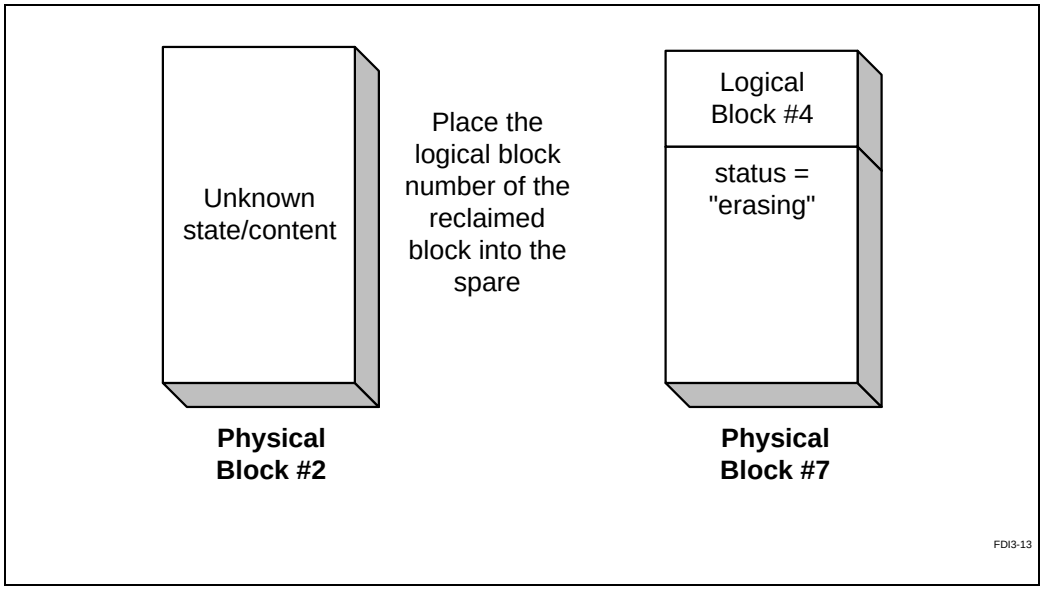


Figure 3-13. Erasing the Reclaim Block

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.

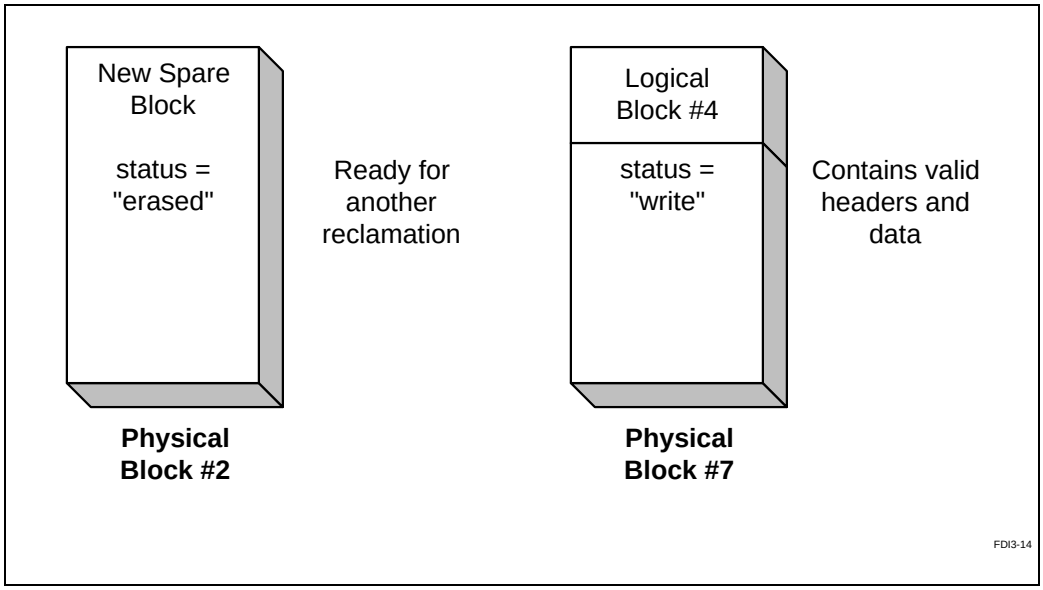


Figure 3-14. Status after Reclamation

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.

4

FDI Test Specification

CHAPTER 4
FDI TEST SPECIFICATION

4.1 INTRODUCTION

The Flash Data Integrator (FDI) software allows users to store system parameters into flash while performing code execution from the same flash device. The software is designed to add future capabilities to update code stored in flash, as well as storing larger data files. The software is initially being directed toward the digital cellular market. This document outlines the test platform and software being developed to allow testing the FDI software in a system and environment similar to a GSM cellular system. While the current system test environment concentrates on the GSM cellular system, many different cellular system architectures (such as PCN, or CDMA based architectures) can utilize the FDI software.

The FDI test system consists of an embedded platform on which the software is tested, a PC to perform remote debug of the embedded platform, and a second PC at which automatic test scripts are run. The embedded platform provides an external interface to receive asynchronous communication from the Test Script Interface and user interface on the PC. The embedded platform consists of a Motorola 48340-based evaluation board with a flash add-on board. The platform utilizes the Wind River Systems VxWorks Operating System and development environment to provide a real-time multi-tasking environment.

On the Test Script Interface PC, test scripts are generated and run. Commands in the test script are verified/translated and downloaded to the embedded controller through an “asynchronous” interface. Once the embedded platform receives a test script, it waits for control on the test PC to indicate that the test should be started. The script is run and all commands are run on the embedded platform to emulate communication similar to a digital cellular phone. In addition, the test script PC can cause asynchronous events, such as a key press, to be simulated and communicated to the embedded platform through the asynchronous interface. The goal of this setup is to simulate realistic cellular phone activity to guarantee applicability of the FDI software. As commands are completed on the embedded platform, status is sent to the test script PC to be placed into a log file. This allows test scenarios to be repeated for reproduction of problems. Figure 4-1 provides an overview of the necessary components.

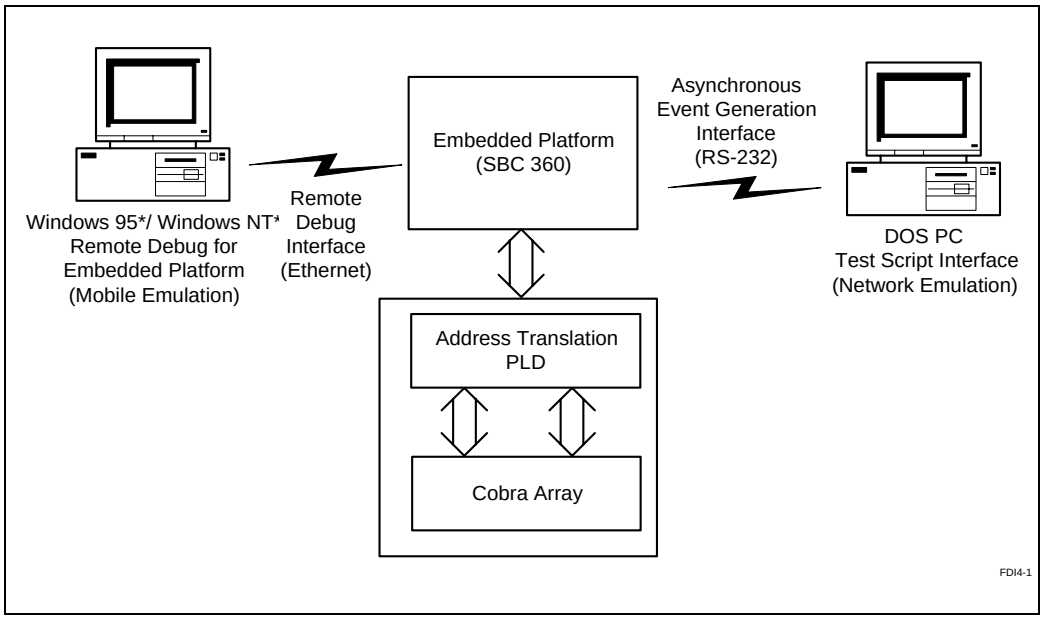


Figure 4-1. System Overview

4.2 DEFINITIONS AND CONVENTIONS

ACC	Access Control Class
ACM	Accumulated Call Meter
ACM Max	Accumulated Call Meter Maximum; the maximum value that the ACM field can store
ADN	Abbreviated Dialing Numbers
AD	Administrative Data
BCCH	Broadcast Control Channel
BYTE	8-bit value
CCP	Capability Configuration Parameters
CBMI	Cell Broadcast Message Id selection
CDMA	Code Division Multiple Access (IS-95)
DAMPS	Digital American Mobile Phone System (IS-54)
DWORD	32-bit value

EXT1	Extension of LND/ADN/SSC/MSISDN parameters
EXT2	Extension of FDN/SSC parameters
FDI	Flash Data Integrator software
FDN	Fixed Dialing Numbers
FNULL	0xFFFFFFFFh; used to refer to flash null media pointers
FPLMN	Forbidden Public Land Mobile Network
GSM	Global System for Mobile communications, a digital cellular radio system
HPLMN	Home Public Land Mobile Network
ICCID	SIM identification
IMSI	International Mobile Subscriber Identification
KC	Ciphering Key
LND	Last Number Dialed
LOCI	Location Information
LP	Language Preference
NULL	Zero
Media Control Structure	A structure of data used as a controlling structure in any of the drivers; a Media Control Structure can exist on the flash card or in RAM
MSISDN	Mobile Station Integrated Services Digital Network Number
PCCP	PC Consumer Peripheral; a consumer product that is being used as a PC Peripheral
PCN	Pacific Digital Cellular
PHASE	Phase Identification
PLMN	Public Land Mobile Network
PLMN Sel	Public Land Mobile Network Selector
PUCT	Price per Unit and Currency Table
SIM	Subscriber Identity Module. This is a removable memory card used in GSM mobile stations. Many parameters are stored in the SIM or in nonvolatile memory on the mobile station or both memories.
SMS	Short Message Service; a feature in GSM cellular networks in which mobile phones may send or receive text messages of a maximum length of 140 bytes
SMSP	Short Message Service Parameters
SMS	Short Message Service Status

SST	SIM Service Table
TDMA	Time Division Multiplexed Access. Each 4.415 ms in time is broken down into eight time slots or logical channels. Each of these logical channels allow a different user to access the shared frequency at a regularly defined interval.
TDMA frame	A 4.415 ms period of time that provides eight separate time slots or logical channels for cellular communication.
TDMA hyper-frame	A large scale timing interval in the GSM system that consists of 2048 super frames which is equivalent to 2H : 28M : 53S : 740 ms of time
TDMA multi-frame	A large scale timing interval consisting of 24 frames for a traffic channel or 51 frames for a signaling channel.
TDMA super-frame	A large scale timing interval consisting of 51 instances of a 24 frame traffic channel multi-frame or consisting of 24 instances of a 51 frame signaling channel multi-frame
TDMA time slot	One of eight 577 ms pieces of time that combine to make a TDMA frame
WORD:	14-bit value

4.3 SYSTEM ENVIRONMENT SOFTWARE ARCHITECTURE

This document covers the system environment software on the embedded platform and the Test Script Interface PC. This software creates the system environment in which the FDI software is verified. The remote debug software for the embedded platform consists of third party vendor software purchased through Wind River Systems. The Test Script Interface (TSI) Software on the PC provides commands that allow the embedded test platform to emulate the cellular network interface. The software on the embedded platform uses these commands to simulate the timing and activity of the mobile station.

On the Test Script Interface PC, a test script is passed to the test executable through the user interface. The test script is parsed through a layer of software that validates all commands and parameters in the script. The test script is then transferred to the embedded platform through a download message which is communicated over the asynchronous interface. Asynchronous messages are received by the embedded platform. The asynchronous interface interprets the messages and passes each message to the appropriate handling mechanism. A download message is passed to the GSM emulation software which breaks the test script down into distinct commands. Each command is treated as a series of GSM events that require simulation. This software is designed to emulate the behavior of a real-time multi-tasking environment which exists in most cellular phones. Other messages from the user interface on the Test PC are used to simulate asynchronous interrupts (such as keypad interrupts) seen in the mobile station environment. Figure 4-2 provides an overview of the software layers and their interaction.

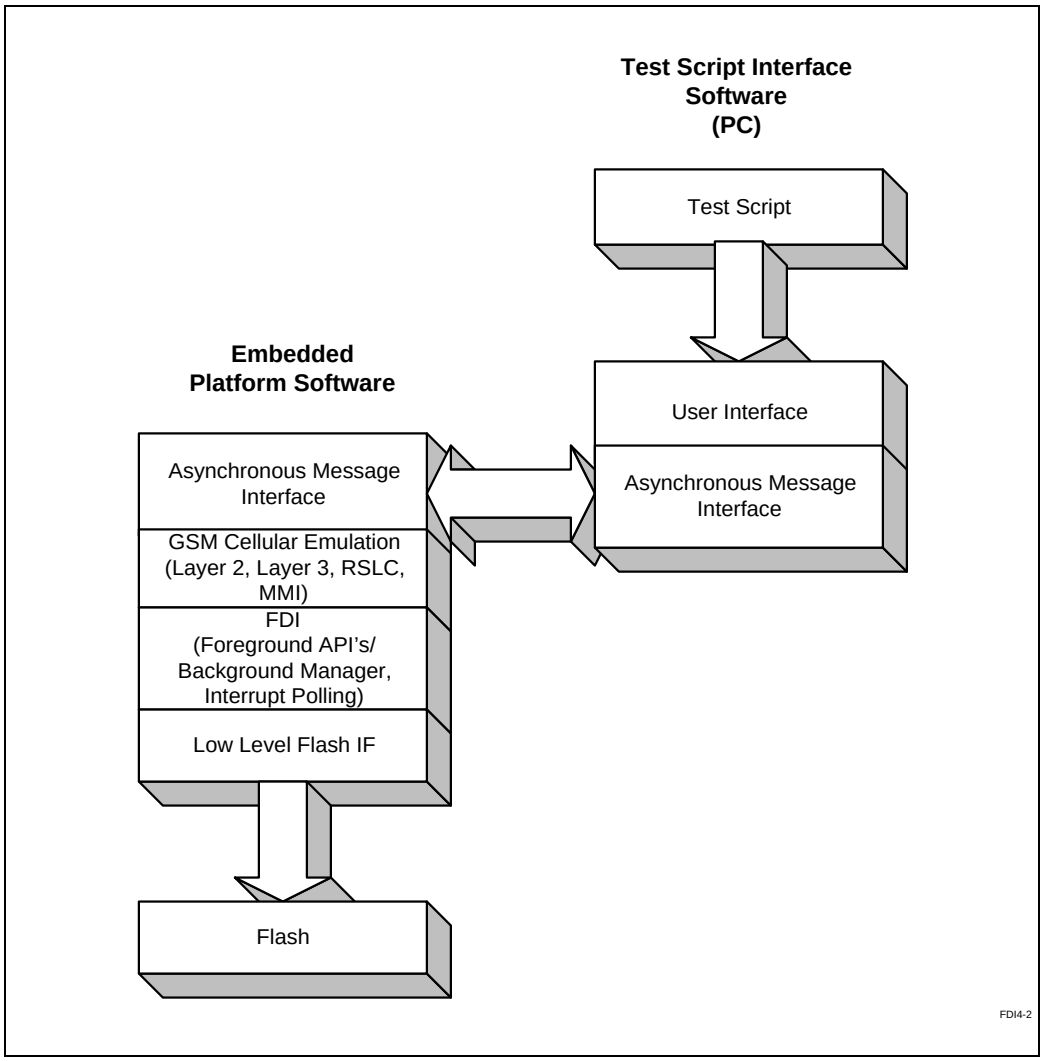


Figure 4-2. Software Overview

4.3.1 PC User Interface

The user interface allows users to enter commands to download test scripts to the embedded platform, control execution of the command scripts on the embedded platform (start, pause, stop, etc.), and send asynchronous commands to the embedded platform while test scripts are executing. It also polls for the receipt of messages sent by the embedded platform and allows reading of stored parameters in flash. These messages contain status updates to be stored in log files to indicate the progress of command scripts and provide feedback information to the user.

4.3.2 PC Asynchronous Communication

This layer of software controls all asynchronous communication on the PC. Asynchronous communication is used to allow emulation of events external to the TDMA emulation. This interface allows for simulation of events such as keypad input, and other events or status to assist with the debug of software on the embedded platform. This interface consists of an ethernet port.

4.3.3 Embedded Asynchronous Communication

This layer of software receives asynchronous commands from the PC Asynchronous Interface, interprets, and acts upon the command. Commands include test script downloads, script control commands, keypad input commands, parameter storage commands, and any future debug messages necessary to successfully test the system.

All messages on this interface return an ack/nack to indicate if the command/message has been accepted as valid. Keypad input allows user input, such as phone numbers, to occur. Once a valid keypad sequence has been received, the information is passed to the parameter storage system.

4.3.4 Embedded GSM Emulation Software

The GSM emulation software is used to emulate the timing involved for each of these commands and the behavior of the system. This layer of software consists of several tasks that simulate the multi-tasking environment of a GSM mobile phone. Each of the tasks track the states similar to those used in a mobile phone. Test scripts sent from the PC are used to simulate parameter storage and the overall CPU availability that is seen in cellular phones. The GSM Emulation software receives the commands, and responds by saving parameters, simulating the activity that would occur in a normal phone.

4.4 PARAMETER STORAGE EMULATION

The test software must allow for testing of storage, update, and retrieval of a set of parameters similar to that used in GSM telephones. The Table 4-1 summarizes the parameters that are currently stored to the SIM card or nonvolatile memory in a GSM phone that could be stored to flash memory. This table forms the base set of parameters used for testing and emulation for the FDI software. Any parameters marked as NV or both in Table 4-1 are candidates for flash and are emulated in the test software.

Table 4-1. GSM Parameter Storage Summary

Memory (S = SIM, NV = Nonvolatile)	Param. No.	File ID	Description	Size (Bytes)	Updated	Update Occurrences
S	2FF2	ICCID	SIM Id	10	Never	
NV	6F05	LP	Language Preference	1–N	Low	User
S	6F07	IMSI	International Mobile Subscriber Id	9	Never	
S	6F20	KC	Ciphering Key	9	High	Init, Termination, Authentication Request
S/NV	6F30	PLMN Sel	PLMN Selector	3N	Low	Init, User
S/NV	6F31	HPLMN	HPLMN Search period	1	Never	Service Provider
S/NV	6F37	ACM Max	ACM max value	3	Never	Service Provider
S	6F38	SST	SIM service table	4	Never	Service Provider
S/NV	6F39	ACM	Accumulated call meter	3	High	Init, periodically during call
S/NV	6F41	PUCT	Price per unit & currency table	5	Never	Service Provider
Both	6F45	CBMI	Cell Broadcast message id selection	2N	Low	User
Both	6F74	BCCH	Broadcast control channel list	16	High	Init, Termination, Location Updates
S	6F78	ACC	Access Control Class	2	Never	Service Provider
S	6F7B	FPLMN	Forbidden PLMN	12	Low	Init, Termination, Location Update Rejection
S	6F7E	LOCI	Location Info	11	High	Init, Termination, Handover (p73: 4.08)
S	6FAD	AD	Administrative Data	3 + X	Never	Service Provider
S	6FAE	PHASE	Phase Identification	1	Never	Service Provider
Both	6F3A	ADN	Abbreviated Dialing Numbers	X + 14	Low	User
S	6F3B	FDN	Fixed Dialing Numbers	X + 14	Never	Service Provider
Both	6F3C	SMS	Short Message Service	176	High	Network SMS procedure

Table 4-1. GSM Parameter Storage Summary (Continued)

Memory (S = SIM, NV = Nonvolatile)	Param. No.	File ID	Description	Size (Bytes)	Updated	Update Occurrences
Both	6F3D	CCP	Capability Configuration Parameters	14	Never	Service Provider
S	6F40	MSISDN	MS - ISDN	X + 14	Never	Service Provider
Both	6F42	SMSP	Short Message Service Parameters	28 + Y	Low	When sending SMS
Both	6F43	SMSS	Short Message Service Status	2 + X	Low	Network SMS procedure
Both	6F44	LND	Last Number Dialed	X + 14	High	Call Termination
Both	6F4A	EXT1	Extension data of LND, ADN/SSC, MSISDN	13	Low	User
S	6F4B	EXT2	Extension data of FDN/SSC	13	Low	User
Both	XXXX	MISC	Miscellaneous OEM specific data (volume control, ring type)	variable 1–300	Low	User

NOTES:

N = >=8

X = Ranges from 0–241

Y =Y may be zero, uses GetResponse function of SIM Card to determine length

4.5 TIMING ENVIRONMENT

GSM cellular phones share frequency band-width through use of Time Division Multiple Access, or TDMA. TDMA allows multiple users to share a common frequency channel on a time schedule. All users sharing the frequency resource have their own assigned repeating time slot within a group of time slots called a frame. Figure 4-3 shows an overview of the timing in a GSM system. Each of the defined time periods shown in this diagram is further explained below. This timing environment must be simulated by the GSM emulation software on the embedded platform.

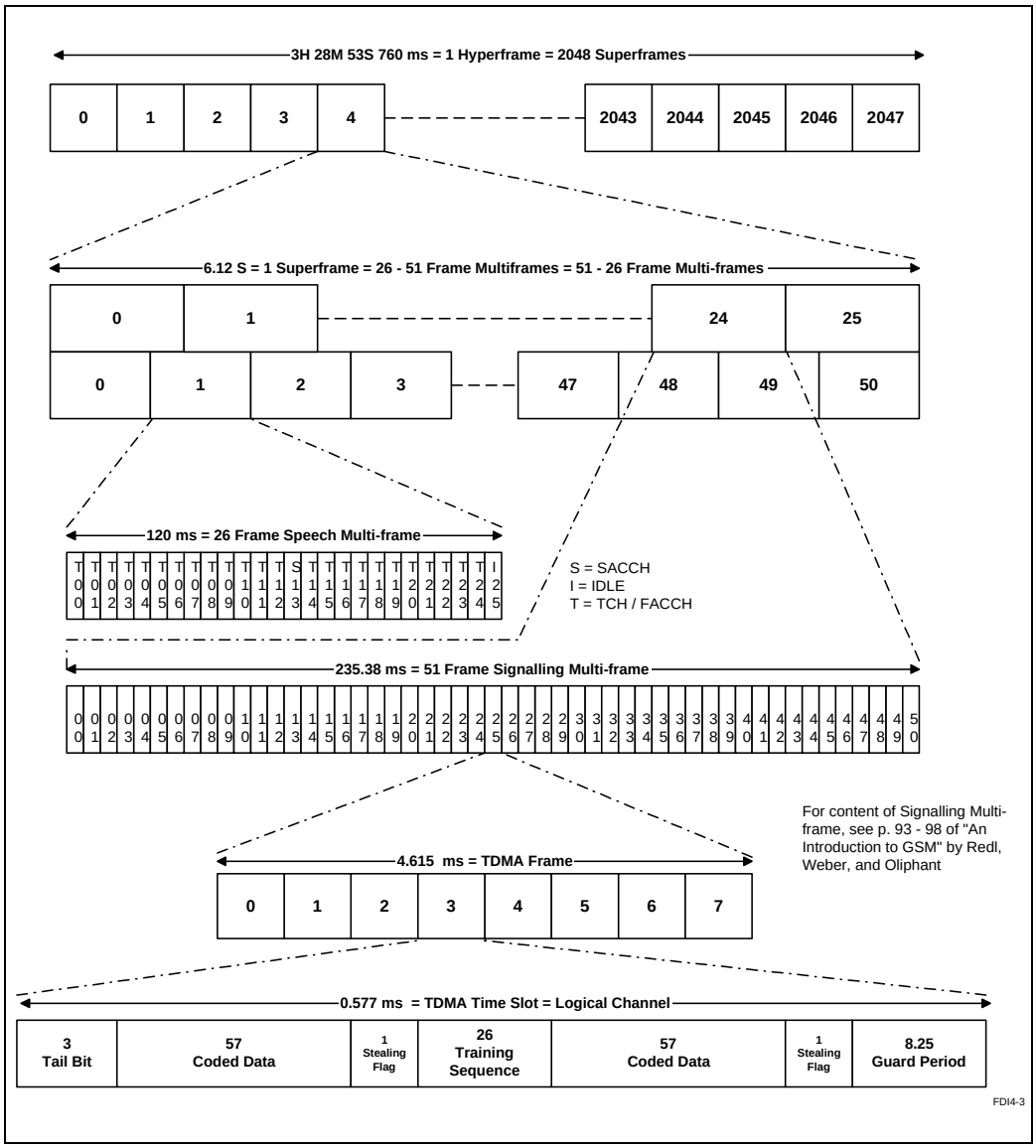


Figure 4-3. Time Division Multiple Access Overview

4.5.1 TDMA Time Slots

In GSM cellular phones, all frequencies are divided into a minimum common repeating interval called a time slot. Each time slot is 577 µs in length. The data transferred during this time follows the structure defined in Figure 4-3.

4.5.2 TDMA Frames

Eight TDMA time slots are combined to create a TDMA frame. Each of eight users are assigned one time slot within a frame. One TDMA frame lasts 4.615 ms (8 * 577 µs). This provides each user a repeating time slot in which to communicate to the cellular network. Each user is assigned a time slot within a time frame and this is called their logical channel. Each logical channel consists of a time slot to receive data from the network, and a time slot to transmit data. Transmissions and receiving frames occur on separate frequencies. The GSM specification requires the time difference between receive and transmit functions to be three time slots, although the time slot numbering (logical channel) remains the same as if both were using the same time slot at the same time. Figure 4-4 shows how time-division duplex occurs in the GSM system.

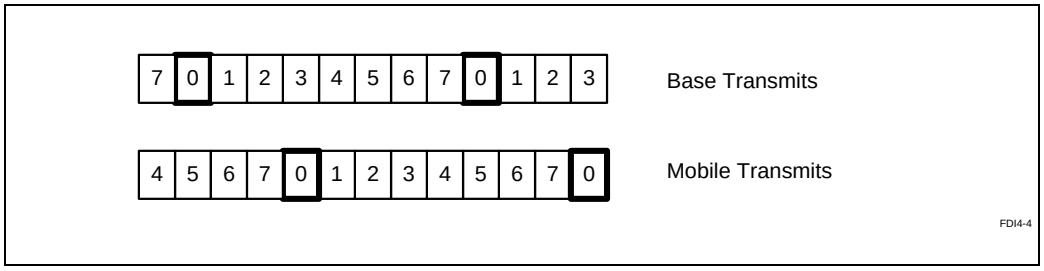


Figure 4-4. Time Division Duplex in the GSM System

4.5.3 TDMA Multi-Frames

A multi-frame is a structure that defines logical channels for a multiple number of frames. This allows different types of signaling opportunities for the variety of equipment in the cellular network. Due to the limitation of bits transferred in each time slot, user-level messages often take several TDMA frames to transmit. Each channel in the multi-frames defined below is configured to take the number of consecutive frames typical messages on the channel take (for example typical messages for the SDCCH channel take four frames to transmit, therefore the channel takes four consecutive frames). In some situations, several instances of a multi-frame must be received before an entire message can be received (for example, normal SACCH messages take four frames, however, the traffic multi-frame only allows one frame to be used for SACCH per multi-frame. This means it would take four traffic multi-frames to receive one SACCH message). Section 4.5.7 defines the uses for each of the channels defined in the multi-frames defined below.

4.5.3.1 SIGNALING MULTI-FRAME

On logical channels that are being used for phone control, 51 consecutive occurrences of a logical channel combine to create what is called a signaling multi-frame. This multi-frame defines certain repeating intervals on a higher level to create a variety of logical control channels. These control channels do not move users data (such as voice or fax), but move the

data the network and mobile phones need to make sure all events in the system are handled properly. All control messaging flows in the test environment mirror the signaling multi-frame defined in Figure 4-5 as closely as possible. Figure 4-3 also contains a high level overview of the signaling multi-frame. All channels and their purposes are described in Section 4.5.7.

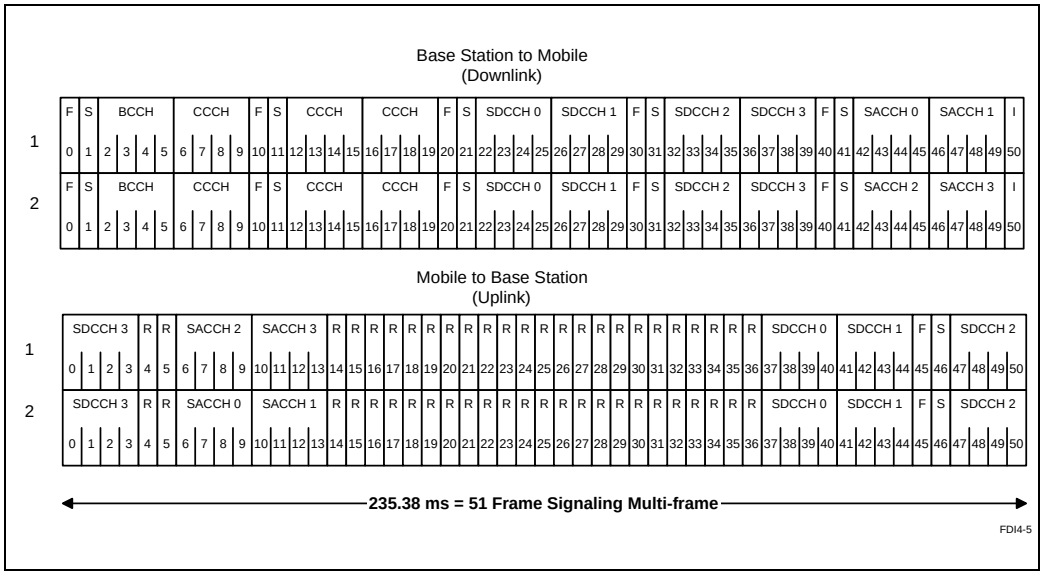


Figure 4-5. Signaling Multi-Frame

4.5.3.2 TRAFFIC MULTI-FRAME

Logical channels that are being used for speech or data are considered a logical traffic channel. Traffic channels follow the communication structure shown in Figure 4-3. The traffic channel structure consists of 26 consecutive occurrences of a logical channel. The first 12 occurrences are used for traffic, one occurrence is used for control, then the last 12 are used for traffic. When urgent control data must be communicated, some of the slices reserved for traffic may be changed to control. This is indicated through use of the stealing flag in each packet. All traffic messaging flows in the test environment mirror the traffic multi-frame defined in Figure 4-3 as closely as possible.

4.5.4 TDMA Super-Frame

The super-frame provides a mechanism for the signaling multi-frame and traffic multi-frame to meet up at regular intervals. This frame type is not used in the test software emulation defined by this document because this time interval is not critical in the evaluation of using FDI software in GSM systems. A super-frame is defined as 26 instances of the 51 frame signaling multi-frame, or 51 instances of a 26 frame traffic multi-frame.

4.5.5 TDMA Hyper-Frame

The hyper-frame provides a timing interval on a larger scale. This frame type is not used in the test software emulation defined by this document because this time interval is not critical in the evaluation of using FDI software in GSM systems.

4.5.6 Background CPU Availability

Each OEM's background CPU availability is different due to architectural differences in each implementation. The background CPU availability created by the test software is modifiable through several different variables that determine the length of time it takes for a system to do several tasks that occur on a regular basis such as transmit, receive, monitoring other frequencies, and power level adjustments.

4.5.7 Communication Channels

Several pre-defined channels exist to provide a guaranteed repeating time slot upon which certain messages are transmitted. These channels are pre-defined by the signaling and traffic multi-frames as certain portions of the multi-frame. This provides mechanisms to allow the mobile phone to originate messages, as well as to locate information about each cell as it moves about a network. Below is an overview of the channels that exist and the type of information sent on these channels. The GSM emulation software emulates the timing associated with using the appropriate channels for the appropriate messages.

There are several types of channels. These are defined as follows:

Traffic Channel (TCH)–Reserved for voice and data messaging.

Broadcast Channel (BCH)–Transmitted only by the base station. Intended to provide information to the mobile station to allow it to synchronize with the network.

Common Control Channels (CCCH)–Support the establishment of a dedicated link between a mobile and a base station.

Dedicated Control Channels (DCCH)–Used for message transfers between the network and mobile station, not for traffic.

Associated Control Channels (ACCH)–Always used in association with a traffic channel (TCH) or Standard Dedicated Control Channel (SDCCH).

Channel	Name	Type	Purpose
TCH	Traffic Channel	TCH	Used to transmit users voice or data.
BCCH	Broadcast Control Channel	BCH	Informs mobile station about system parameters it needs to identify the network (cell options, neighboring cell frequencies, access parameters, location area code, etc.).
FCCH	Frequency Correction Channel	BCH	Provides the mobile station with the frequency reference of the system and helps in initial synchronization.
SCH	Synchronization Channel	BCH	Provides the training sequence the mobile station needs to demodulate information coming from the network and indicates current frame number.
RACH	Random Access Channel	CCCH	Used by the mobile station to request a dedicated control channel from the network.
PCH	Paging Channel	CCCH	Used by the base station to call an individual mobile station.
AGCH	Access Grant Channel	CCCH	Used by the base station to inform the mobile which dedicated channel it should use for its signaling needs and provides a channel description.
SDCCH	Standalone Dedicated Control Channel	DCCH	Used for transfer of information between the mobile and the network.
SACCH	Slow Associated Control Channel	ACCH	Used for control and measurement parameters or routine data needed to maintain a link between the mobile station and the network.
FACCH	Fast associated Control Channel	ACCH	Used to replace all or part of a traffic channel to transfer information similar to that carried by the SDCCH.

4.6 TEST ENVIRONMENT COMMANDS

There are two types of commands that exist in the test environment. Test script commands are sent in a script to the embedded platform and are executed on the embedded platform to simulate GSM activity and parameters. User commands are sent from a user interface to control execution of the test script, to create asynchronous events, and to provide user feedback.

4.6.1 Test Script Commands

The test script commands allow a multi-tasking environment similar to a GSM cellular phone to be simulated. The test script is sent as a whole entity (using the download command in the user commands) to the embedded environment and the GSM simulation on the embedded platform executes each command and simulates the activity that would normally be seen in

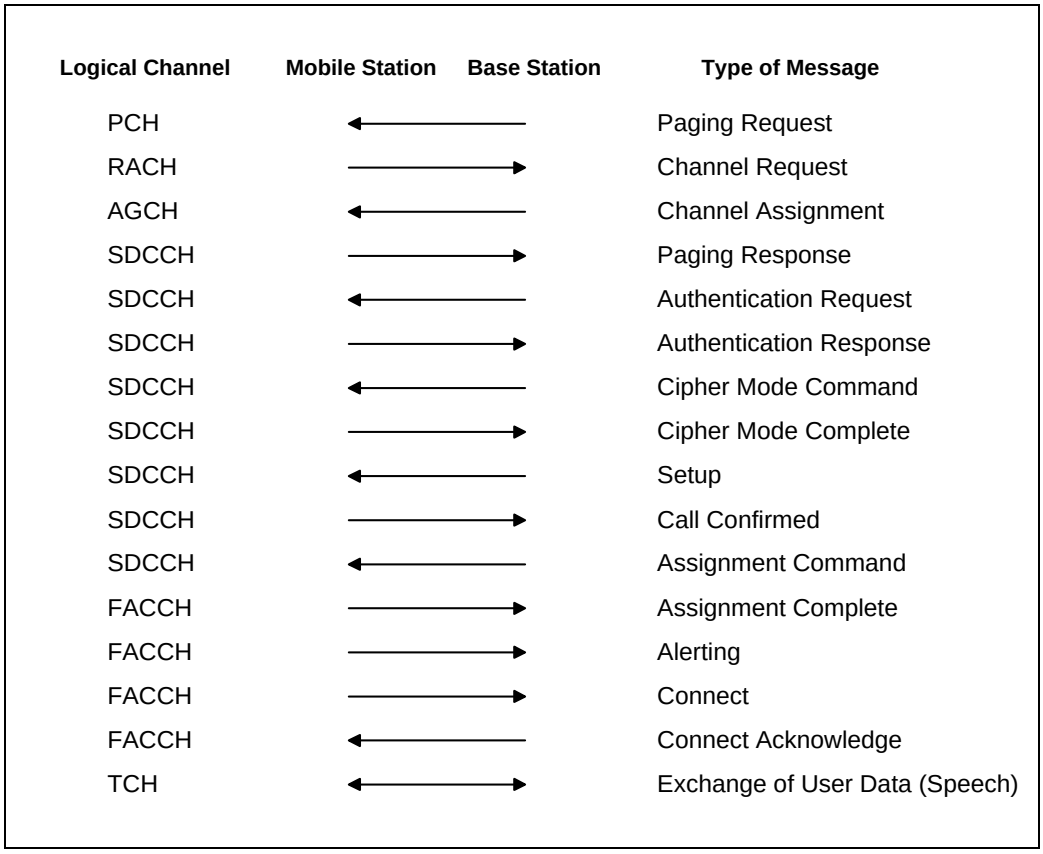
messaging as well as any other background activity and timing that would normally occur on a GSM cellular phone.

The following sections describe each of the call flows to be simulated for the commands summarized in the table below. Each flow shows which logical channel is used for communication. This can be used along with the signaling/traffic multi-frame drawings to determine the GSM simulation behavior. Each command overview below also defines which parameters are possibly updated and in which situations they are updated.

Command	Inputs	Purpose	SIM Params. Affected	NV Params. Affected
Call Establishment		Allow storage/reading of parameters	ACM, KC, LOCI (status)	ACM
Location Update	Success, Failure	Allow storage/reading of parameters	FPLMN, KC, LOCI	
Loop	Label, Count	Allows looping of various command sequences	None	None
Wait	Time	Allows time between commands to be altered	None	None
Call Clearing		Allow storage/reading of parameters	ACM, LND	ACM, LND
Send SMS		Allows storage/reading of SMS parameters	KC, LOCI, SMS, SMSS	SMS
Determine Random Parameter Number	Number Range	Allow random testing to occur	None	None
Fill buffer with data	Random or Pre-Determined	Allows random data testing to occur	None	None

4.6.1.1 CALL ESTABLISHMENT

Call establishment is used by the GSM network to set up all necessary channels to establish a phone call connection. Normally, calls can be originated by the network or by the mobile station. To simplify the design, the test software allows network-originated call establishment only. All messages in the following scenario are simulated through the GSM emulation software. The following flow is for the network originated (mobile terminated) call establishment in a normal channel assignment situation.

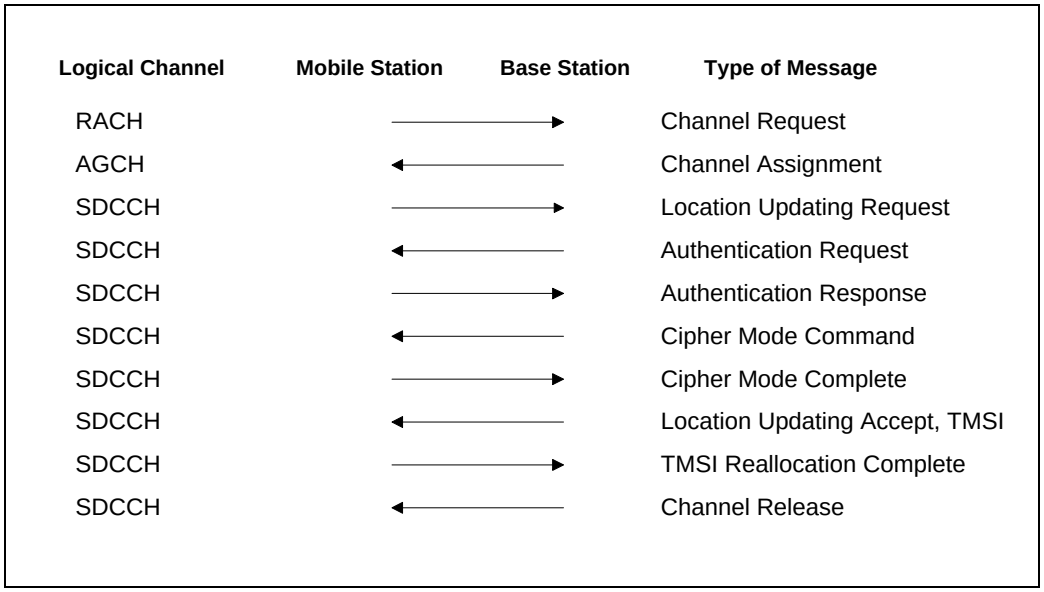


The parameters which may be updated with relation to the call establishment are as follows:

Parameter	Cause for Update
ACM	Update begins once call is established, is updated to Flash once every X seconds where X is defined at compile time.
KC	Authentication Request
LOCI	When channel assignment is received, TMSI portion when Cipher Mode command is received

4.6.1.2 LOCATION UPDATE

The location update is used by the GSM network to keep the network informed about the location of the mobile station. All messages in the following scenario are simulated by the GSM emulation software. The following flow is for the successful location update.



The parameters which may be updated with relation to the location update are as follows:

Parameter	Cause for Update
FPLMN	Location Update rejection
KC	Authentication Response
LOCI	TMSI Reallocation Command

4.6.1.3 LOOP

No call flow exists for this command. This command is used to force the command translation software in the GSM emulation portion of the embedded system to automatically repeat portions of the test script.

4.6.1.4 WAIT

No call flow exists for this command. This command is used by the command translation software in the GSM emulation portion of the embedded system to allow the phone to remain in its current state for a pre-defined amount of time allowing the call to continue or the phone to remain in its current state.

4.6.1.5 SEND SMS

Short Message Service (SMS) is a special feature in the GSM network to allow short messages to be sent to/from a mobile. To simplify the design, the test software simulates network originated SMS only. This command allows an SMS command to be sent asynchronously to the embedded platform. Several parameters may be updated with relation to the SMS command. They are as follows:

Parameter	Cause for Update
KC	
LOCI	
SMS	SMS message received
SMSS	Status of message received/sent
SMSP	

4.6.1.6 CALL CLEARING

Call clearing is used by the GSM network to release all channels used in a phone call connection. Normally, calls can be cleared by the network or by the mobile station. To simplify the design, the test software allows network originated call clearing only. All messages in the following scenario are simulated by the GSM emulation software. The following flow is for the network originated call clearing.

The parameters which may be updated with relation to call clearing are as follows:

Parameter	Cause for Update
ACM	Update to this parameter ceases when disconnect is received
LND	Update to this parameter upon call established or disconnect

4.6.1.7 DETERMINE RANDOM PARAMETER

This command allows random parameters in a certain range to be chosen. This allows random variability in parameter updates.

4.6.1.8 FILL BUFFER WITH DATA

This command allows a buffer to be filled with random data or pre-determined data. This buffer is used to provide data for parameter updates. This information is determined on the test PC and transferred to the embedded system. This allows verification of data to occur.

4.6.2 User Commands

The user commands assist in creating the GSM environment by allowing asynchronous events to occur as well as controlling the test script execution and receiving status information to indicate progress. Status information is displayed for the user as well as stored into a log file. All commands in this section consist of a command message followed by a command response message. Message details are defined in the Detailed Definition documentation for this software.

Command	Origination	Inputs	Purpose	SIM Params Affected	NV Params Affected
Download Script	PC	File Input	Download a script to run on the embedded platform	None	None
Run Script	PC	None	Begin execution of a test script	None	None
Pause Script	PC	None	Temporarily halt execution of script at next possible location	None	None
Stop Script	PC	None	Halt/abort execution of script at next possible location	None	None
Keypad Input	PC	File Input	Allow asynchronous interrupts as in cellular environment, allow update of user parameters	ADN, EXT1, EXT2, FDN, LP, PLMN Sel	ADN, EXT1, EXT2, FDN, LP, PLMN Sel, OEM defined params
Parameter Update	PC	File Input, # Bytes, Parameter ID	Allows basic storage/update of parameters	None	All
Parameter Read	PC	# Bytes, Parameter ID	Allows read verification of parameters	None	All
Status Update	Embedded System		Indicates information about command completion to provide user feedback	None	None
Misc Debug commands	Either	Currently Undefined	Currently undefined	None	None

4.6.2.1 DOWNLOAD SCRIPT

This command allows the user to input a test script file. The test script file is then parsed, validated, and sent to the embedded system. The test script is stored on the embedded system in a RAM area. The script is run at a later time when given the “run script” command by the user.

4.6.2.2 RUN SCRIPT

This command allows the user to begin execution of a script on the embedded platform. The embedded platform begins execution of the most recent script that has been downloaded. If no script has been downloaded, the embedded platform returns an error in the response. If the execution of a script was in progress and no new script has been downloaded, this command functions as a “resume execution” command.

4.6.2.3 PAUSE SCRIPT

This command allows the user to pause execution of a script on the embedded platform. The embedded platform stops execution of the script when the command in progress completes and retains the current command. The “run script” command is used as a method to resume execution. This command allows execution to stop to allow various structures on the embedded platform to be evaluated and allow execution to continue.

4.6.2.4 STOP SCRIPT

This command allows the user to stop execution of a script on the embedded platform. The embedded platform stops execution of the script when the command in progress completes. The current command is not retained in this situation.

4.6.2.5 KEYPAD INPUT

Keypad input is performed by a user on a GSM cellular mobile station. This creates asynchronous interrupts in a mostly synchronous environment. This command allows simulation of key presses to occur. Each key press generates a keypad input command. Keypad commands from the user interface may take input from a file and generate several keypad input message sequences.

Several parameters may be updated with relation to the keypad input. They are as follows:

Parameter	Cause for Update
ADN	User enters ADN key sequence
EXT1	User enters ADN key sequence
EXT2	User enters FDN key sequence
FDN	User enters FDN key sequence
LP	User enters LP key sequence selection
PLMN Sel	User enters PLMN key sequence selection
User Defined Parameters	

4.6.2.6 PARAMETER UPDATE COMMAND

This command includes the identification of the parameter to update, the offset and number of bytes to update in the parameter, followed by the data with which to update the parameter. This command, along with the parameter read command, allows basic parameter storage to be tested without waiting for the GSM emulation software to be complete.

4.6.2.7 PARAMETER READ COMMAND

This command includes the identification of the parameter to update, the offset and number of bytes to read from the parameter. This command expects a response message that includes the parameter identification, offset, number of bytes, and the actual parameter data. This command, along with the parameter update command, allows basic parameter storage to be tested without waiting for the GSM emulation software to be complete.

4.6.2.8 STATUS UPDATE

This command is initiated by the embedded platform and is sent at the completion of each command in the test script. This information is used by the PC to update the user interface and to update the log file that tracks test script execution.

4.6.2.9 MISCELLANEOUS

Future messages may be added as debug aids or additional simulation tools. The design should be easily modifiable to allow command additions to be simple.

4.7 PC TEST FUNCTIONALITY

4.7.1 Functionality Overview

The PC Test Script Interface software allows automated testing of the parameter storage database to occur. It provides a list of script commands that are interpreted and turned into a command file that is sent to the embedded platform. Many of these commands simulate the GSM environment, while others are used for debug purposes only. It also provides a user interface at which user commands can be sent to the embedded system while a test script is executing.

4.7.2 Asynchronous Interface

This interface is responsible for sending commands over the asynchronous interface and receiving any appropriate responses. This interface consists of the following software modules:

1. Communication hardware control
2. Messaging software that allows users to send or receive messages on this interface

The Communication hardware control performs initialization for all aspects of the communication device for the channel being used and sets up any interrupt vectoring necessary for the control chip. The receive message interrupt first handles any device specific needs, then calls a function to handle any message specific needs. Once a full message is received, the message is placed into a message queue.

The messaging software provides the capability of sending or receiving full messages. This software is responsible for using the hardware interface to transmit multiple byte messages, as well as receiving multiple byte messages. The receive function monitors the incoming message queue for a pre-determined length of time. If the message queue is empty and no response is received in the pre-determined length of time, an error is returned to higher layer levels of software.

4.7.3 User Interface

This layer of functionality is the main control of the PC test interface. Once all initialization is complete (variables initialized, log file set up, etc.), control is passed to this layer of software which watches for user input as well as received messages from the embedded platform. If messages are received from the embedded platform, they are either command responses or status updates. Each message is handled in the appropriate manner. If user input is received, the command is validated and the appropriate message is sent to the embedded platform.

When a test script download is requested through the user interface, a layer of test script validation and parsing software will exist. This layer of software provides functions for reading a file that contains ASCII commands, validating each of the commands, and sending the

compiled commands to the embedded system. The test script parsing software should support all commands specified in the table which summarizes all Test Script Commands, in Section 4.6.1. The test script parsing software should allow expandability in the command set and should not be limited to this pre-defined set of commands. These functions are used by the user interface software when a user command to download a test script has been entered.

4.8 EMBEDDED GSM EMULATION FUNCTIONALITY

4.8.1 Functionality Overview

The GSM emulation software provides a real-time multi-tasking environment in which parameters are updated and provides a variety of test scenario situations. Tasks similar to those found in a cellular mobile station exist which provide similar communication scenarios, however, the tasks do not perform tasks necessary for cellular communication. Instead, the CPU overhead is simulated and must be modifiable to allow different scenarios to be tested. Figure 4-6 outlines the tasks involved and their high level interaction.

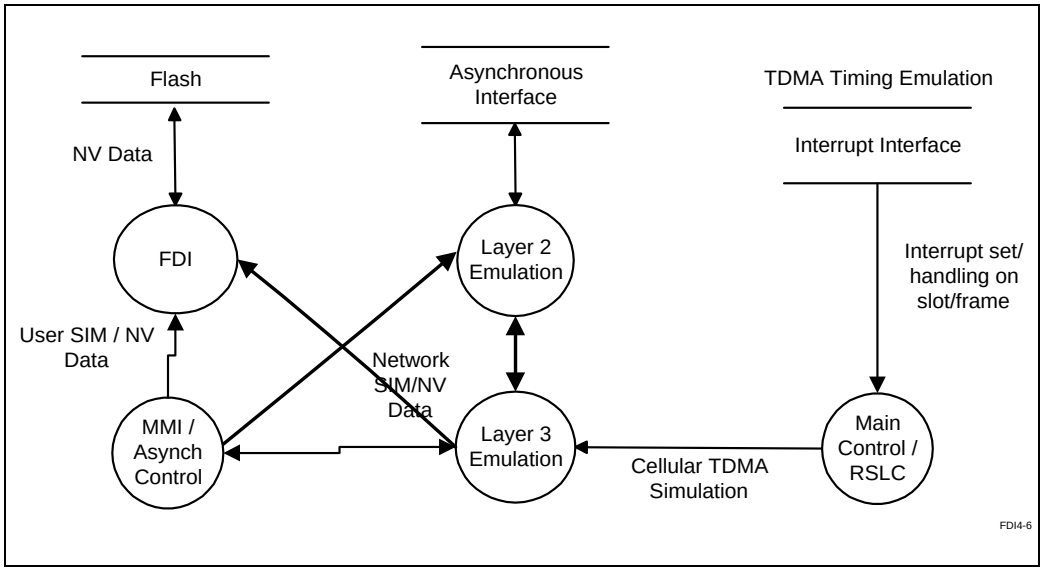


Figure 4-6. High-Level Simulation Task Interaction

4.8.2 Asynchronous Interface (MMI)

This interface is responsible for receiving commands over the asynchronous interface, validating the key combinations or command, sending the appropriate response, passing commands on to the appropriate task when necessary, and storing any appropriate parameters

through use of the FDI interface. This interface consists of several software modules. These are:

1. Communication hardware control
2. Asynchronous interface tasks to act on messages received/send the appropriate response

The communication hardware control performs initialization for all aspects of the communication device for the channel being used and sets up any interrupt vectoring necessary for the control chip.

The asynchronous receive task pends on a message queue that indicates the receipt of a message. Once a message is received, this task verifies the message and sends the appropriate response. Any keypad input combinations are evaluated for validity and once a full keypad combination is received, the appropriate parameter is updated through use of the FDI interface. The asynchronous send task pends on a message queue to perform a send.

4.8.3 Layer 2 Emulation

This interface is responsible for simulating activity of a synchronous TDMA interface. This interface is also responsible for providing services to layer 3 that allow the simulation of layer 2 activity. This interface contains a function to emulate sending TDMA messages and a function to emulate receiving TDMA messages.

4.8.4 Layer 3 Emulation

This layer of software emulates the messaging and call control at its highest level. This software consists of a task that pends on messages from the network and modules for call and resource control emulation. This layer parses through test scripts to determine which GSM call flows to emulate.

4.8.4.1 RADIO RESOURCE CONTROL

This software module is responsible for emulating the establishment, maintenance, and termination of Radio Resource connections for use by the mobility management and call management software. This software deals with the changing radio environment. Specific procedures to be emulated include:

1. Channel assignment and release function
2. Channel frequency changes
3. Frequency hopping sequences (if necessary)
4. Measurement reports from the mobile (performed approximately 1 time per multi-frame)
5. Power control and timing (performed approximately 1 time per multi-frame)
6. Cipher mode setting

4.8.4.2 MOBILITY MANAGEMENT

This software module is responsible for emulating the authorization and connection of the mobile user, as well as emulating supporting mobility functions. Specific procedures to be emulated include:

1. Location and periodic updating
2. Authentication
3. IMSI attach and detach, TMSI re-allocation
4. Identification

4.8.4.3 CALL CONTROL

This software module is responsible for emulating call control, short message service, and any special services. Specific procedures to be emulated include:

1. Call establishment and call control
2. Call termination

4.8.5 RSLC Emulation/Main Control

This task is responsible for emulating cell selection and for emulating the behavior of the mobile station while in idle mode. This function emulates the following:

1. Measures signal strength of each carrier
2. Looks for 30 strongest signals, reads BCCH data
3. When camped on a cell, listens to BCCH and PCH
4. Decodes BCCH data every 30 seconds
5. Decodes BCCH of six strongest non-serving cells at least every five minutes

A

FAQs

CHAPTER A
FAQs

OVERVIEW

This document contains questions and answers relating to Intel's Flash Data Integrator (FDI) software used for code plus data storage in a single flash memory component. This document assumes the reader is familiar with the information in Chapter 3, *FDI Architecture and API Specification.*

Q & As are organized in the following categories:

- Hardware
- Software
- Reclaim
- General

HARDWARE QUESTIONS

1. What if the system cannot withstand a 20 µs latency for erase suspend or 10 µs for program suspend?

   The 20 µs latency is only associated with Erase suspend. An erase operation occurs only during a reclaim process. FDI includes a Reclaim_Enable command which can be used to disable the reclaim. With reclaim disabled the maximum latency is based on program suspend, which is a maximum 10 µs. If the system cannot support this, a portion of the interrupt handler would need to be loaded into system RAM to allow code execution for the 10 µs or 20 µs period. Intel continues to evaluate improvement of suspend latency performance.

2. Are nested erase and program suspends supported? In other words, can you suspend a write to read after you have suspended an erase?

   The architecture supports erase suspend to program and subsequent program suspend to read.

SOFTWARE QUESTIONS

1. What SW effort is there in integrating FDI?

Feature	Intel FDI	Other OEM Solution
Flash Data Integrator (FDI) Architecture	included	500 devl.-hrs
Flash Parameter Storage Management	included	600 devl.-hrs
Flash Storage Management Reclaim	included	600 devl.-hrs
EEPROM Interface	included	80 devl.-hrs
Power Loss Recovery	included	480 devl.-hrs
Flash Suspend/Resume Interface and Testing	included	N/A
API Integration	400 devl.-hrs	N/A
Total	**400 devl.-hrs**	**2,260 devl.-hrs**

2. What is the FDI code size? What is the RAM size requirement for FDI?

   The estimated code size will equal 16 KB–20 KB and the RAM will equal 2 KB–3 KB for the data queue and flash memory drivers.

3. What variables affect the RAM buffer size when using FDI? And, how does the FDI RAM buffer size compare to that required for RWW solutions with specialized circuits (H/W RWW) and for EEPROM?

   The estimated data throughput affects the RAM buffer size. The available MIPS from the processor also affects the RAM buffer size. The FDI RAM buffer size is identical to what would be needed for H/W RWW. The RAM buffer for FDI is actually less, in most cases, than what OEMs are using today with EEPROM. Many OEMs mirror all parameters in RAM and EEPROM. Using a RAM buffer limits the usage of RAM for current systems.

4. What happens if a higher priority read occurs during a lower priority read?

   Read operations are not queued. The lower priority read will complete and then the higher priority read will be executed.

5. Why are unit headers independent of data?

   Having unit headers independent of data allows any algorithms that scan information in a block to automatically index through the headers instead of having to calculate the size of the information it must skip over to reach the next unit header.

6. Can you stream data over a block boundary?

   A sequence table is included in the FDI architecture that supports the management of fragmented data. The sequence table is used to point to instances across block boundaries.

7. Is the sequence table supported in the initial release?

   The architecture supports it, but this is not enabled in the initial release. Future releases of FDI will include full support of a sequence table which enables data storage across block boundaries.

8. Does FDI support data re-packing?

   Data re-packing consumes power to rewrite data that already exists in flash. Rather than re-packing data, FDI uses an instance table to track data.

9. Is there flexibility in FDI to limit EEPROM replacement across a smaller number of parameter blocks (e.g., use three or four blocks instead of six or eight)? Can code be stored in the non-used parameter blocks?

   Two or more blocks can be used for data storage. There are compile time options to identify the data block size (8 KB or 64 KB) and which blocks are used for data storage.

10. Will stale data be read if a read occurs prior to queued data being written to flash?

    FDI uses a look aside technique: the valid instance is first read from flash then FDI looks in the RAM queue to see if the data has been superseded. The most current instance of the data is returned.

11. Why is cycle count contained in the block information?

    The cycle count is incremented if the block is reclaimed. This information is used by FDI for wear leveling. Wear leveling is a compile option that will be available and will allow users to determine if they would like this information to be used and to influence which block is chosen during reclaim.

12. What are the benefits of wear leveling?

    Wear leveling allows all blocks to be used at an equal rate. If wear leveling is not used, this can cause some blocks to reach maximum erase cycling before other blocks. For example, without wear leveling, blocks with fixed data (does not get updated) are not cycled as much as blocks with changing data. If wear leveling is used, it may require the reclaim of a block that has little or no invalid data in the block. This causes an occasional extra reclaim, however, the block with a lower cycling count can now be cycled with changing data.

13. How are variable size parameters handled by FDI?

    The FDI architecture includes a multiple instance data structure that is user- programmable. It can be adjusted to support multiple instance of a small parameter or single instance of a large parameter (e.g., a fixed factory tuning parameter such as a trim level on a D/A converter).

14. Why did Intel select the multiple instance data structure? What is the “expense” of managing smaller granularity data?

    The multiple instance data structure allows a parameter or small data object to be updated with minimal modifications to overhead. Typically an update to data that is being tracked in a multiple instance structure entails updating three status bits in addition to the data. When the multiple instance structure is filled, it entails updating an 8-byte unit header along with the size and status information in the multiple instance structure. In comparison, using a single instance structure would waste several bytes in the allocated data unit, as well as cost an 8-byte unit header update and a single bit status update with each update to the parameter.

15. How does FDI handle high priority data writes if data is pending in the RAM queue?

    The FDI data queue is prioritized. The most important parameter is written first.

16. Is FDI as safe as using an EEPROM in the event of power failure?

    FDI includes status fields within the data parameter structure as well as the physical block information field. This information is used during the initialization process to detect if data was not successfully written or if a reclaim process did not complete. EEPROM management software does not normally include similar levels of protection. If power is removed during an EEPROM write, the data may be corrupted. FDI offers an equal or increased level of data protection.

RECLAIM QUESTIONS

1. What is reclaim?

    When a flash block is full (or near full), the valid data within that block must be copied to another block. Data writes are then done to the new block. The reclaim process (a) copies valid data from a full or near full block to a spare block and (b) erases the full or near full block. Reclaim is sometimes referred to as “garbage collection.”

2. How do you suspend a reclaim?

    A reclaim is suspended through system interrupts. This can be a hardware or software interrupt.

3. What feedback does FDI provide following a reclaim?

    An API function (FDI_Statistics) exists to allow the application to monitor the state of memory. An API function (FDI_Query) exists to allow the application to monitor the status of reclaim as well as the status of the RAM data queue.

4. How long does it take to reclaim as a function of data reclaimed?

    Reclaim time = erase time + (write time * bytes copied in reclaim) + software overhead time. Benchmark examples will be available when the software is released to assist in estimating software overhead time.

5. What if RP# is asserted during a reclaim? Is the reclaim operation aborted? If it is, how is this handled since power was not removed and the initialization routine will not be executed?

   If RP# is connected to the system reset, power loss recovery algorithms will deal with any issues during the normal FDI_Init call.

6. What statistics are available on how often is it necessary to do a reclaim?

   The FDI_Statistics function provides the total free space available as well as the total invalid space. There is one threshold that is user configurable that allows the user to determine when a reclaim should be initiated by the file system. Once the file system determines that a reclaim should occur, it can either handle the reclaim automatically, or request permission from the application (through the use of a semaphore). Permission to reclaim can be granted through the FDI_Reclaim function. During reclaim, new data can be written until a second threshold is reached that indicates a full memory.

7. Can you write data during a reclaim suspend?

   The Advanced Boot Block architecture supports erase suspend to read or program.

GENERAL QUESTIONS

1. Can I implement FDI on a device that does not have program suspend? Can I implement FDI on a device that does not have erase suspend?

   This depends on the latencies that are allowable on the system. If the system can accept a latency as large as the device's maximum program time, then FDI can be used on a device without program suspend. If the system can accept a latency as large as the device's maximum erase time, then FDI can be used on a device without erase suspend. This may also require that the system has latched or "sticky" interrupts.

2. If the CPU does not support "sticky" interrupts, how can FDI work?

   Any interrupt line is raised high for a minimum period of time. If this time is greater than the maximum interrupt detection time in the flash interrupt polling algorithm, then FDI will detect the interrupt before the interrupt line transitions.

3. Can data be stored in the main blocks as well?

   The current assumption in the software is that all blocks being used for data are symmetrical in size. This means that any number of 8-KB blocks or 64-KB blocks can be used for data storage, but not simultaneously.

4. Will FDI support data storage in both the parameter and main blocks simultaneously?

   Combining 8-KB and 64-KB blocks together is not impossible, however, the current software algorithms do not support this option. The spare block must remain the size of the largest data block size. Extra algorithms are required to allow a combination file system.

5. Does FDI have any “security” built into it to ensure that my code is safe when writing data?

   The flash program and erase algorithms perform boundary checks on the flash addresses passed in to the algorithm. This boundary check ensures that the address being accessed falls within the flash data area being managed.

6. Has FDI been verified on other digital cellular standards like CDMA, DAMPS, or PDC?

   In addition to GSM cellular phone validation, verification of the FDI software on other digital cellular standards will be done in the future.

7. Does Intel have any intentions to standardize this software?

   A file system standard is really only necessary when media will be exchanged between multiple OEM systems. Since this is not the normal case for FDI systems, standardization is currently not being pursued.

8. What are the licensing rights?

   OEMs have royalty-free derivative rights to all source code.

9. What if I need a multi-chip solution? Will FDI still work?

   As long as the memory map for the area being managed by FDI is contiguous.

10. Is FDI affected by voltage?

    Reclaim and programming performance are directly impacted by voltage. Lower voltages result in slower reclaim performance and program throughput.

11. Should fixed parameters be handled in the same manner as parameters that are updateable?

    Fixed parameters should be grouped together into larger parameters. Any of the individual portions of the larger parameter can still be read using an offset with the FDI_Read command. Grouping fixed parameters in this manner allows less media waste due to granular memory allocations and prevents the data from being placed into multiple instance structures in which the extra instances will never get used.

B

Technical Support

APPENDIX B
TECHNICAL SUPPORT

Intel provides extensive customer support for all of its products. For technical assistance on flash memory products, please contact Intel's Customer Support (ICS) team at 1-800-628-8686. You may also send electronic mail to the ICS team [e-mail* address: ICS_Flash@ccm.fm.intel.com]. The e-mail* must be of the following form (Note: the colon following each keyword is required):

Company: *[Enter your company's name]*

Name: *[Enter your name]*

Product: *[Enter the specific flash product name, e.g. TE28F160B3-T120]*

Question: *[Enter your question]*

For FDI support, check the FAQ section of this manual. Documentation is available from Intel's Flash memory WWW (http://www.intel.com/design/flcomp) page or Bulletin Board System (BBS). For specific software questions, send e-mail to the flash software team [e-mail address: flash@inside.intel.com]. Also the flash software team can be reached at 1-916-356-8922.

C

System/Flash Hardware Requirements

APPENDIX C
SYSTEM/FLASH HARDWARE REQUIREMENTS

This chapter outlines the system and hardware requirements for implementing the Flash Data Integrator (FDI): a real-time O/S, interrupt polling, flash, and RAM.

C-1 REAL-TIME O/S

A real-time operating system is required to allow certain FDI tasks to be time sliced with other system processes. During a reclaim operation for example, the Background Manager (see Chapter 5, *FDI API Specification*) shares processing time with other system tasks. The real-time operating system requirements include multi-tasking and the ability to communicate between tasks via semaphores or flags.

C-2 INTERRUPTS

Interrupt polling is required to handle real-time interrupts that may occur during flash erase and programming operations. FDI disables interrupts before initiating a program or erase command. If a higher level interrupt occurs, a key being pressed on the keypad for example, the program or erase command is suspended and the required interrupt service routine is then allowed to execute.

C-3 FLASH

In order to implement FDI, the flash device must support deterministic program and erase suspend to read. The suspend capability allows the servicing of real-time interrupts (Figure C-1 shows a program suspend example). A spare flash block is also required. This spare block is used for reclaim (garbage collection). Lastly, the FDI code is estimated to be between 16 KB–20 KB in size (both the Foreground and Background Manager).

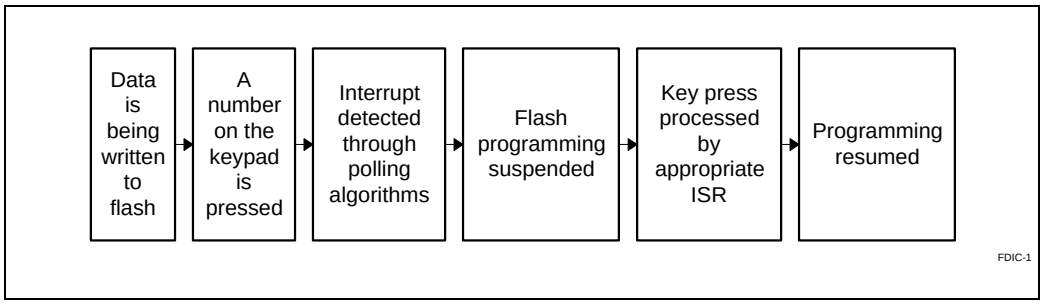


Figure C-1. Example of a Program Suspend

Any flash device which satisfies the above conditions may also use FDI to achieve code plus data storage within a single flash memory device. Intel's Smart 3 Advanced Boot Block and Smart 3 FlashFile™ memory products provide deterministic suspend capability: program can be suspended in 10 µs maximum (5 µs typical) and erase can be suspended in 20 µs maximum (10 µs typical).

C-4 RAM

RAM is required to assist FDI in managing data storage to flash. Current estimates indicate 2 Kbytes–3 Kbytes total is required. One Kbyte is used to store the flash program and erase routines (executed by the Background Manager from flash); the remaining RAM is used as a data queue and for system variables.

The Foreground Manager stores pending data to be written to flash in the RAM queue; when spawned, the Background Manager writes the data to flash.

D

FDI Licensing Agreement

APPENDIX D
FDI LICENSING AGREEMENT

INTEL OEM

SOFTWARE LICENSE AGREEMENT

BY USING THIS SOFTWARE, YOU ARE AGREEING TO BE BOUND BY THE TERMS OF THIS AGREEMENT. DO NOT USE THE SOFTWARE UNTIL YOU HAVE CAREFULLY READ AND AGREED TO THE FOLLOWING TERMS AND CONDITIONS. IF YOU DO NOT AGREE TO THE TERMS OF THIS AGREEMENT, PROMPTLY RETURN THE SOFTWARE PACKAGE AND ANY ACCOMPANYING ITEMS.

IF YOU USE THIS SOFTWARE, YOU WILL BE BOUND BY THE TERMS OF THIS AGREEMENT

LICENSE: Intel Corporation ("Intel") grants you the non-exclusive and royalty-free right to use the enclosed software program ("Software"). You will not use, copy, modify, rent, sell or transfer the Software or any portion thereof, except as provided in this Agreement.

OEM System Developers may:

1. Copy the Software for support, backup or archival purposes;
2. Install, use, or distribute Intel owned Software in object code form only;
3. Modify and/or use Software source code that Intel directly ships to you as an OEM;
4. Install, use, modify, distribute, and/or make or have made derivatives ("Derivatives") of Intel owned Software under the terms and conditions in this Agreement, ONLY if you are an OEM system developer and NOT an end-user.

RESTRICTIONS:

YOU WILL NOT:

1. Copy the Software, in whole or in part, except as provided for in this Agreement;
2. Decompile or reverse engineer Software provided in object code format;
3. Remove or modify the "Compatibility" module, if any, in the Software or in any Derivative work.

TRANSFER: You may transfer the Software to another party if the receiving party agrees to the terms of this Agreement at the sole risk of any receiving party.

OWNERSHIP AND COPYRIGHT OF SOFTWARE: Title to the Software and all copies thereof remain with Intel or its vendors. The Software is copyrighted and is protected by United States and international copyright

laws. You will not remove the copyright notice from the Software. You agree to prevent any unauthorized copying of the Software.

DERIVATIVE WORK: OEMs that make or have made Derivatives will not be required to provide Intel with a copy of the source or object code. OEMs shall be authorized to use, market, sell, and/or distribute Derivatives at their own risk and expense. Title to Derivatives and all copies thereof shall be in the particular OEM creating the Derivative. OEMs shall remove the Intel copyright notice from all Derivatives if such notice is contained in the Software source code.

DUAL MEDIA SOFTWARE: If the Software package contains multiple media, you may only use the medium appropriate for your system.

WARRANTY: The Software is provided "AS IS". Intel warrants that the media on which the Software is furnished will be free from defects in material and workmanship for a period of **one (1) year** from the date of purchase. Upon return of such defective media, Intel's entire liability and your exclusive remedy shall be the replacement of the Software.

THE ABOVE WARRANTIES ARE THE ONLY WARRANTIES OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE.

LIMITATION OF LIABILITY: NEITHER INTEL NOR ITS VENDORS OR AGENTS SHALL BE LIABLE FOR ANY LOSS OF PROFITS, LOSS OF USE, LOSS OF DATA, INTERRUPTION OF BUSINESS, NOR FOR INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES OF ANY KIND WHETHER UNDER THIS AGREEMENT OR OTHERWISE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

TERMINATION OF THIS LICENSE: Intel reserves the right to conduct or have conducted audits to verify your compliance with this Agreement. Intel may terminate this Agreement at any time if you are in breach of any of its terms and conditions. Upon termination, you will immediately destroy, and certify in writing the destruction of, the Software or return all copies of the Software and documentation to Intel.

U.S. GOVERNMENT RESTRICTED RIGHTS: The Software and documentation were developed at private expense and are provided with "RESTRICTED RIGHTS". Use, duplication or disclosure by the Government is subject to restrictions as set forth in FAR52.227-14 and DFAR252.227-7013 et seq. or its successor.

EXPORT LAWS: You agree that the distribution and export/re-export of the Software is in compliance with the laws, regulations, orders or other restrictions of the U.S. Export Administration Regulations.

APPLICABLE LAW: This Agreement is governed by the laws of the State of California and the United States, including patent and copyright laws. Any claim arising out of this Agreement will be brought in Santa Clara County, California.

E

Additional Information

APPENDIX E
ADDITIONAL INFORMATION

Order Number	Document/Tool
210830	*1997 Flash Memory Databook*
290580	*Smart 3 Advanced Boot Block 4-Mbit, 8-Mbit, 16-Mbit Flash Memory Family* Datasheet
292148	*AP-604 604 Using Intel's Boot Block Flash Memory Parameter Blocks to Replace EEPROM*
292199	*AP-641 Achieving Low Power with Advanced Boot Block Flash Memory*
292200	*AP-642 Designing for Upgrade to Smart 3 Advanced Boot Block Flash Memory*

NOTE:

1. Please call the Intel Literature Center at (800) 548-4725 to request Intel documentation. International customers should contact their local Intel or distribution sales office.
2. Visit Intel's World Wide Web home page at http://www.Intel.com for technical documentation and tools.