Technical Note

Micron® Flash Data Integrator (FDI) Support for Java™ Applications

Introduction

This document provides an instructional reference to customers of the Micron® Flash Data Integrator (FDI). An OEM or ODM utilizing FDI in their design can use this document to construct a comprehensive Java™-enabled cellular/embedded system. Beginning with the basic market perspective on Java, this document describes in detail a generic implementation of Java in the targeted environment, focusing on the implications of using FDI as the system's Flash storage media manager.

Definitions

Table 1: Definitions and Acronyms

Term	Definition
Application Program Interface (API)	A collection of standard protocols, functions, and tools to assist a developer with building software.
CLDC	J2ME Connected Limited Device Configuration. Defines the minimum set of Java virtual machine features and Java class libraries available on a connected device (wired or wireless) that has limited hardware features.
Code Manager (CM)	The portion of the FDI API concerned with contiguous storage of contiguous objects (such as code) that supports the requirements of J2ME application storage.
Direct Access Volume (DAV)	A partition within Flash that is managed by FDI for the purpose of supporting XIP.
Flash Data Integrator (FDI)	Flash media management software. FDI handles the complex task of managing Flash so that an application can read and write from Flash simultaneously.
Java2 Micro Edition (J2ME)	The version of the Java 2 platform that caters to consumer electronics and embedded devices such as cellular phones.
Java Application Manager (JAM)	Serves as the interface between host RTOS, FDI, and KVM. Responsible for interfacing with FDI to store and retrieve files to and from Flash memory on behalf of the KVM. JAM can be used to retrieve Java files from the network in lieu of retrieving Java files from a nonvolatile storage device. Also performs classfile verification.
Kilobyte Java (K-Java)	A subset of the Java language specifically targeting the needs of an embedded environment.
Kilobyte Java Virtual Machine (KVM)	A compact, portable Java virtual machine designed for small, resource-constrained devices.
Mobile Information Device (MID)	A cellular phone, pager, or similar mobile/embedded device.
J2ME Mobile Information Device Profile (MIDP)	A targeted API that allows third-party developers to create wireless Java applications that run on a mobile phone or pager that includes a J2ME-compliant KVM.
Real-Time Operating System (RTOS)	An operating system with a deterministic response time (meaning the system is guaranteed to respond to inputs within a fixed delay time). Embedded and cellular systems typically run an RTOS.

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

Products and specifications discussed herein are for evaluation and reference purposes only and are subject to change by Micron without notice. Products are only warranted by Micron to meet Micron's production data sheet specifications. All information discussed herein is provided on an "as is" basis, without warranties of any kind.

Table 1: Definitions and Acronyms

Term	Definition
eXecute in Place (XiP)	A Flash storage mechanism that stores files in a contiguous fashion that facilitates code execution (as opposed to store-and-download, in which a fragmented application is reconstructed in volatile memory prior to execution).
Profile Layer (Example: MIDP)	Defines the minimum set of APIs available on a particular family of devices representing a particular vertical market segment.
Configuration Layer (Example: CLDC)	Defines the minimum set of Java virtual machine features and Java class libraries available on a particular category of devices representing a particular horizontal market.

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

Market Perspectives

Java Community Perspective

Java Historical Perspective

Historically speaking, Java was originally described by Sun Microsystems, Inc. with the following statement: “Java is a simple, object-oriented, distributed, interpreted, robust, secure, architecture-neutral, portable, multi-threaded, dynamic, buzzword-compliant, general-purpose programming language developed by Sun Microsystems in 1995.” Sun later saddled Java with the slogan “write once, run anywhere.” Though this lofty promise has not entirely been fulfilled, the major objectives of Java for mobile and embedded devices might be filtered into the following statement: to develop secure, platform-independent applications that can be downloaded to a device by the user and executed in a constrained environment. The Java community quickly realized that the one-size-fits-all goal was not achievable. This realization led to the derivation of three different versions of Java:

- **Java 2 Enterprise Edition** targets servers and high-end workstations.
- **Java 2 Standard Edition** targets workstations, consumer PCs, and laptops.
- **Java 2 Micro Edition** targets set-top boxes, PADs, mobile phones, and other mobile/ embedded devices and consumer electronics.

The Java environment is essentially an open one; applications are written to the specific Java API and ultimately run on any device that contains the particular virtual machine. In theory, any Java application written in the Java language for the J2ME CLDC configuration will run on any device supporting that configuration.

On a final note, one basic assumption that is necessary for most Java functionality is the existence of a network connection. The network connection is the umbilical cord that supplies a source for the user to obtain Java applications.

Embedded Java (J2ME) Project Goals

- As stated above, the objective of J2ME is to develop secure, platform-independent applications that can be downloaded to a device by the user and executed in a constrained environment. This statement consists of a number of separate goals. How does the J2ME project address each of these goals?
- **Security** is addressed by:
  - Running the Java application through a verification process before execution, where the verifier checks for many known methods of gaining control of the device.
  - Providing a sandbox in which the application can execute, and preventing any reference outside of the sandbox.
  - Preventing an application from modifying any installed Java library.
  - Preventing an application from overriding any class in a protected system package.
- **Platform independence** is addressed by the virtual machine. The Java application is compiled into byte codes that the virtual machine interprets. The virtual machine essentially translates Java byte codes into instructions for the system’s specific processor. This enables software development without regard to the underlying processor. The catch is that someone must develop the virtual machine for the target processor.
- **Download-and-execute** is addressed by the Java Application Manager, which retrieves the Java application and initiates the execution of the file on the virtual machine.
- **Resource-constrained environment** is addressed by the KVM, CLDC and MIDP:

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

- The KVM is the virtual machine that caters to mobile and embedded devices. The KVM supports a small subset of the requirements of the much broader and resource-plentiful desktop or server environment.
- The CLDC is the configuration specific to cellular phones and pagers. This configuration supports, via the KVM, much of the consistent functionality of any mobile phone and/or pager application.
- The MIDP is the profile that supports, via an API, much of the consistent functionality of any phone and/or pager application.
- By limiting the KVM to operations, and the API to functionality that a generic mobile phone or pager would use, the size of the package can be greatly reduced.

Mobile Device Manufacturer Perspective

Current Mobile Device Trends

The typical mobile device manufacturer objective is to develop an embedded device that consumers will need to have. The trend has been to develop mobile/embedded devices of ever-increasing complexity. As the devices become more complex, the manufacturer has a decision to make. They can either:

- Create all of the applications for the device in a closed environment
  or
- Create an open environment and allow users to take advantage of third-party software that is capable of enhancing the user experience.

The former approach demands that the manufacturer create everything and is prohibitive in today's rapid development environment. The latter approach allows the manufacturer to get a phone in the hands of a user, and then push for development of value-added applications that enhance the user experience. The trade-off is developing everything in-house versus developing the framework in which third-party software developers can work.

J2ME Mobile Device Environment

With the creation of the J2ME environment, several mobile phone manufacturers have taken the latter approach and created the framework based on the Java language and execution environment. This approach provides a blank canvas on which anyone can create Java applications, as long as they follow the requirements of the mobile/embedded platform (CLDC configuration, MIDP profile, KVM virtual machine).

Mobile/Embedded Device Manufacturer Responsibilities

The mobile device manufacturer has the responsibility to provide or acquire the following mechanisms to support the Java framework:

- A kernel to control the target hardware and provide a thread for execution of the KVM and JAM
- KVM for target hardware/RTOS/processor
- JAM for target hardware (the application manager is responsible for storing, retrieving, loading, and verifying the Java application on target without a file system)
- MIDP profile Java class libraries
- A mechanism to receive Java applications from the network
- Nonvolatile storage mechanism for Java applications

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

Third-party Software Developer Perspective

Third-party Software Developer Approach

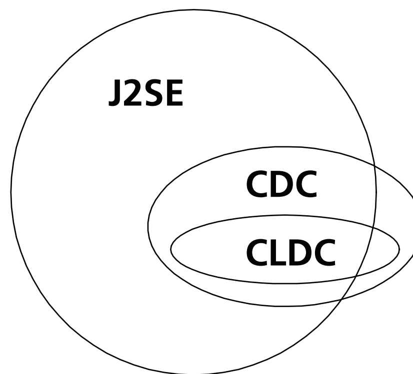
From the perspective of the third-party developer, the ideal Java application is developed a single time and is capable of running on any J2ME-compatible mobile/ embedded device. The third-party developer simply creates the application and provides it to the network carrier, who subsequently provides the application to the user on request.

J2ME Development Environment

The J2ME development environment allows a developer to compile, debug, test, and run a J2ME applet on a desktop PC. The development environment consists of a compiler, a set of class libraries, a bytecode verifier, and a generic J2ME emulator. The J2ME emulator runs any application developed using the J2ME class libraries, simulating the user interaction and outputs of the application as they would appear on a real mobile/ embedded device.

The developer of a J2ME application can safely rely on a consistent set of APIs across all J2ME-compliant mobile/embedded devices. Developing an application based on the minimum set of common class libraries ensures compatibility across devices. However, the mobile device manufacturers are free to extend the J2ME specification and include class libraries of functionality outside of the minimum set. For the third-party developer, this limits the number of platforms that are capable of running their application. The relationship between the standard Java class libraries and the CLDC class libraries is illustrated in:

Figure 1: Relationship Between J2ME Configurations and Java 2 Standard Edition



Future of J2ME

It is anticipated that eventually the third-party applications will drive the adoption of the J2ME-compliant mobile/embedded devices, in much the same way that a standard API drove the adoption and success of the PC. The J2ME environment overcomes the limitations introduced by the lack of platform and OS standardization in the mobile/ embedded device environment – an environment where third-party developers can choose their role in the value-added infrastructure, be it mobile commerce, games, information, or otherwise.

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

Micron Flash Data Integrator (FDI) Perspective

J2ME Application Storage Needs

The generic specification of Java describes a scenario whereby the user downloads an application every time he/she wishes to use the application. The mobile/embedded device manufacturers saw this as inefficient and proposed an alternative: the user downloads the application the first time, then stores it on the mobile/embedded device for future use. This type of functionality requires nonvolatile storage of the applications. With nonvolatile storage now a need, the mobile/embedded device manufacturers started to investigate methods for storing applications in Flash memory.

Micron Flash memory has become a widespread standard for cellular phone storage needs. Many cellular phone manufacturers use Micron Flash Data Integrator to manage Micron Flash memory. The combination of Micron Flash memory and FDI provide a cost-effective means for mobile/embedded device OEM's to include reliable nonvolatile storage for their device without dedicating precious development resources to implementing a Flash media manager.

Current FDI Implementation

Micron aspires to provide Flash memory to mobile/embedded devices for nonvolatile storage of code and data. In conjunction, Micron offers FDI software to mobile/embedded device OEMs to simplify the integration process necessary to utilize Micron Flash memory. FDI has internal mechanisms that read and write data to and from specific managed blocks within Flash memory. FDI breaks large data items into smaller fragments to simplify storage management and optimize Flash utilization. On top of that functionality, FDI provides a robust power-loss recovery system that protects data integrity through power interruptions and fluctuations. From a user's point of view, FDI provides a simple API to manage data in Flash memory.

J2ME FDI Implementation Requirements

The available FDI storage mechanisms alone are not sufficient for the needs of executing J2ME applications. The Java Application Manager (JAM) requires direct access (a pointer address) to the storage location and the Java application object must be stored in a contiguous fashion. To meet the storage requirements of the J2ME applications, the FDI software team at Micron has developed the Code Manager, a Direct Access Volume storage mechanism. The CM stores the object in a contiguous fashion while providing the JAM direct access to the object.

The high-level requirements for K-Java applet storage are:

- Direct Flash memory access
- Reclaim-locking
- Unrestricted file size
- Simultaneous KVM and FDI access
- Java application data access
- Power-loss recovery
- File management
- Pointer fix-ups
- Distinct J2ME and FDI volumes

Throughout this document, each of the high-level requirements are described in detail.

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

FDI/J2ME Integration

The remainder of this document is dedicated to describing:

- J2ME application storage requirements in detail
- How FDI meets the J2ME application storage requirements
- The steps necessary to integrate FDI with the J2ME environment from a mobile/embedded device manufacturer's perspective

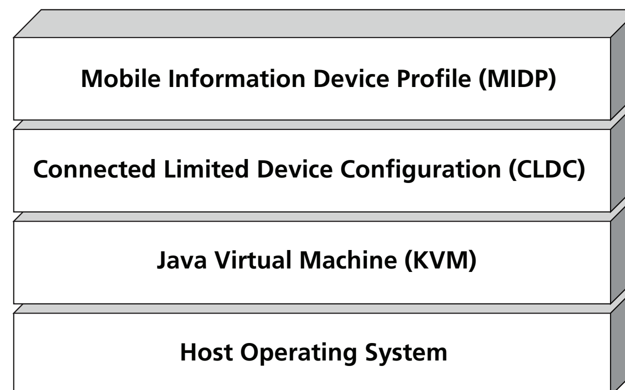
Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

Developing J2ME Compatible Mobile Devices

Java 2 Micro Edition Platform

The Java 2 platform specifies several implementations, configurations, and profiles for specific market segments. For the mobile market, several industry participants have defined the J2ME platform catering to mobile devices. This platform consists of the Mobile Information Device Profile (MIDP), the Connected Limited Device Configuration (CLDC), the Java 2 Micro Edition (J2ME), the Java programming language, and the Kilobyte Virtual Machine (KVM).

Figure 2: Java 2 Micro Edition Software Stack



Java Virtual Machine (KVM)

At the lowest layer, the mobile embedded device manufacturer must have a minimal kernel to manage the hardware and to provide a mechanism to schedule and run the Java virtual machine. While a full-featured operating system is not required, many commercial RTOS operating systems currently support, or plan to support, the J2ME KVM. Choosing a commercial RTOS that implements the J2ME KVM relieves the manufacturer of the responsibility of developing the KVM.

Additional Java KVM information may be found at the URL: http://java.sun.com/products/kvm/

Connected Limited Device Configuration (CLDC)

The next layer up is the Connected Limited Device Configuration. A configuration is the minimum set of Java virtual machine features and Java class libraries available for the device. The specific requirements of the CLDC configuration can be found in Sun specification JSR-30. The specification is summarized in the following list:

- Goals:
  - Define a standard, minimum-footprint Java platform for small, resource-constrained, connected devices
  - Network connectivity
  - Dynamic delivery of Java applications
  - Open environment for third-party application developers
- Scope:
  - Define Java language and virtual machine features
  - Core Java libraries

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

- Input/output
- Networking
- Security
- Internationalizations

Additional Java CLDC information may be found at the URL: http://java.sun.com/products/cldc

Mobile Information Device Profile (MIDP)

The next layer up is the Mobile Information Device Profile. A profile defines the minimum set of APIs available for the device. The specific requirements of the MIDP specification can be found in Sun specification JSR-37. The specification is summarized in the following list:

- Goals:
  - Define the architecture and the associated APIs required to enable an open, third-party application development environment for mobile information devices (MIDs)
- Scope:
  - Application life-cycle management
  - User interface functionality
  - Event handling
  - High-level application model

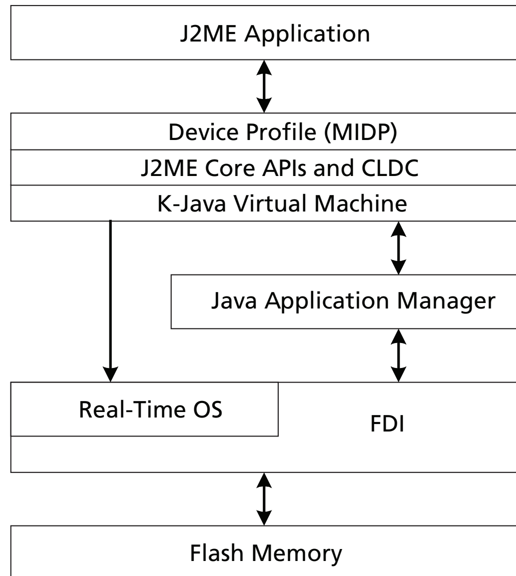
Additional Java MIDP information may be found at the URL: http://java.sun.com/products/midp/

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

Generic J2ME Architecture

A generic implementation of the J2ME/CLDC/MIDP specifications could be created as shown in Figure 3. Each component is described below from a top-down perspective.

Figure 3: Generic J2ME KVM Architecture



J2ME Application

The J2ME application is the driving force behind implementing the architecture shown above. The application can be anything from a game to a personal information manager to a video player. The open infrastructure potentially allows an infinite number of applications to run on the device. The remainder of the architecture layers exists to support the open J2ME application development environment.

Mobile Information Device Profile

The mobile information device profile is the layer that requires the most effort on the part of the mobile/embedded device manufacturer. The MIDP is built on top of the CLDC and focuses the generic functionality of the CLDC specification to a target application, such as a cellular phone. Several implementation issues must be resolved at the MIDP layer, specifically Java application storage and Java application management.

Connected Limited Device Configuration

The CLDC specification imposes the minimum set of requirements and class libraries that are necessary to develop the MIDP layer. At the heart of a CLDC implementation is the virtual machine. Together with the virtual machine and the class libraries, the CLDC forms the software development foundation for an array of mobile/embedded devices. Typically, the CLDC layer is a component of a commercial RTOS that supports Java.

K-Java Virtual Machine

The K-Java virtual machine is typically an application or thread of execution that reads a Java classfile and interprets the Java byte codes. The interpretation process consists of mapping the generic Java byte codes onto the native instruction set of the target

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

processor, or mapping the generic Java OS resource to the native RTOS resource for execution. The KVM is tightly coupled with the CLDC configuration and is typically a component of a commercial RTOS that supports Java.

Real-Time OS

The real-time operating system (RTOS) is responsible for tying the entire software subsystem to the system hardware. This layer is responsible for executing the interpreted Java byte codes on the system processor. The RTOS is a significant design decision to be made by the mobile/embedded device developer. Most commercial RTOS vendors support a Java VM on many popular target processors.

Target Hardware & Flash Memory

At the core of the target hardware is the target processor. The target hardware also commonly includes RAM (or a similar volatile memory subsystem), Flash memory, timer device, I/O devices, interrupt mechanism, and interface logic. For many mobile/embedded device targets, Flash memory provides nonvolatile storage of code and data. This medium is well-suited for storage of Java applications.

Flash Data Integrator (FDI)

Mobile/embedded devices that use Flash memory for nonvolatile storage of code and data need a low-level software component to control and manage the Flash media. Micron Flash Data Integrator (FDI) is an off-the-shelf software package that simplifies the development of Flash media management in a mobile/embedded device. This package includes mechanisms that simplify nonvolatile storage management and execution of J2ME applications.

Java Application Manager

The Java Application Manager (JAM) is responsible for managing the life cycle of the J2ME application. The management functions associated with the JAM are J2ME application storage, retrieval, de-allocation, and modification. The JAM relies on FDI for managing the nonvolatile storage of J2ME applications. The JAM is also responsible for initiating J2ME application execution on the virtual machine.

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

Generic J2ME Implementation

Platform-Specific Implementation

The platform-specific implementation consists of the target hardware, target processor and the RTOS. These items are specific to the mobile/embedded device and will not be discussed further.

J2ME-Specific Implementation

The J2ME CLDC core and K-Java virtual machine are typically supported in the chosen RTOS. If the chosen RTOS does not support the CLDC core and K-Java virtual machine, then the mobile/embedded device manufacturer must develop them, or obtain them from Sun Microsystems and integrate them with the target platform. This decision is specific to the mobile/embedded device and will not be discussed further.

Mobile Information Device Profile Implementation

The mobile information device profile is the layer that requires the most effort on the part of the mobile/embedded device manufacturer. The following minimum requirements have been specified for the MIDP:

- Display:
  - Screen-size: 96x54
  - Display depth: 1-bit
  - Pixel shape: approximately 1:1
- Input:
  - One or more user-input mechanism: one handed keyboard, two handed keyboard, or touch-screen
- Memory:
  - 128KB of nonvolatile memory for MIDP components
  - 8KB of nonvolatile memory for application-created data
  - 32KB of volatile memory for Java runtime
- Network:
  - Two-way wireless, possible intermittent, with limited bandwidth
- APIs
  - Application APIs (defining the semantics of a MIDP application and how it is controlled)
  - User-interface APIs
  - Nonvolatile storage APIs
  - Networking APIs
  - Timer API
- Implementation:

The mobile/embedded device manufacturer must implement all of the APIs outlined above to be compliant with the MIDP specification. In addition to the items specified in the MIDP, the device manufacturer must also develop the Java Application Manager, which is responsible for:

- Application download mechanism
- Application nonvolatile storage mechanism
- Application life-cycle management mechanism

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

Java Application Manager Implementation

The Java Application Manager (JAM) also requires a significant amount of development on the part of the mobile/embedded device manufacturer. The JAM is responsible for life-cycle management of the Java application.

Life-cycle management consists of:

- Downloading the Java application
- Uncompressing the Java application
- Verifying the Java application classfile
- Installing the Java application (nonvolatile Java application storage)
- Upgrading the Java application
- Uninstalling the Java application
- Launching the Java application
- Controlling the Java application execution

Java Application nonvolatile Storage Need

The Java Application Manager requires a software package that manages the nonvolatile storage mechanism for Java applications. The mobile/embedded device manufacturer can develop this capability in-house, or use FDI (version 3.01 or higher), which supports the Java application storage requirements.

The following section covers the specific requirements for the nonvolatile storage of Java applications in the J2ME/CLDC/MIDP environment.

©2011 Micron Technology, Inc. All rights reserved.

J2ME Application Storage Requirements

Mobile Information Device Profile (JSR-37)

The J2ME Connected Limited Device Configuration calls for nonvolatile storage of J2ME applications on the mobile/embedded device. Several high-level requirements have been specified for this storage mechanism. FDI has the capability to support nonvolatile storage of J2ME applications. However, FDI design goals conflict with a few of these high-level requirements. This section discusses each of the J2ME application storage requirements individually. First, each requirement is described, including the reason that it is a requirement. Following that is an explanation of how FDI currently implements the requirement. Last is a discussion of how the Code Manager addition to FDI supports the requirement.

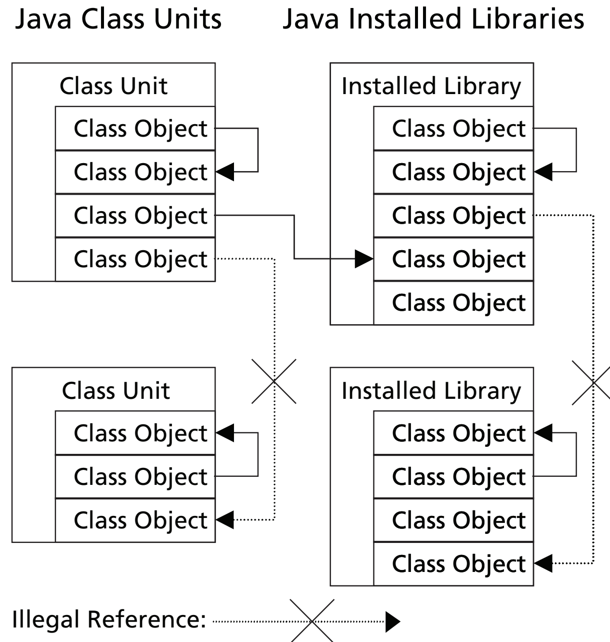
Detailed J2ME Application Storage Requirements

Nonvolatile storage of J2ME applications requires eXecute In Place (XIP) capability. XIP capability is defined as a mechanism that provides pointer access to a contiguous array of bytes in the storage medium. In this case, the array of bytes represents the Java application. The J2ME Class Units and Installed Libraries require XIP capability. The J2ME application files are not truly executed in place. Instead, the KVM fetches the byte codes to be interpreted directly from Flash memory via pointer access. Because of the direct fetch, each J2ME application file must be represented contiguously in memory; this way, fetching a subsequent byte code is as simple as incrementing the pointer to Flash memory. Flash reclamation must also be synchronized with J2ME application interpretation.

A Class Unit and Installed Library may contain many Class Objects. Within a Class Unit or Installed Library, Class Objects may reference one another. Class Objects within a Class Unit can also reference Class Objects within Installed Libraries, but not Class Objects in other Class Units. Class objects within an Installed Library cannot reference any Class Objects outside the Installed Library.

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

Figure 4: J2ME Java Class Object Associations



A J2ME application file may require internal pointers to absolute addresses in order to meet performance requirements. This introduces an issue with managed nonvolatile storage. If the application is moved to another location in Flash memory, such as during reclaim, the internal absolute address pointers will require patching.

The requirements for J2ME application nonvolatile storage are as follows:

- Direct Flash Access: The KVM interpreter must be capable of accessing the J2ME application file via a pointer to the file. Each file must be stored contiguously in Flash memory.
- Reclaim Locking: Reclamation is the restoration free storage space by rearranging data in the storage medium. Reclaim locking is a function of FDI that prevents reclamation. During file interpretation, the file contents cannot be moved. Thus, Reclaim must be locked until it is granted permission to unlock by the KVM. If reclamation is needed in the middle of file interpretation, Reclaim requests permission to unlock from the KVM. The KVM can choose when to unlock Reclaim. Once reclamation is complete, the KVM can re-fetch the file location and recommence interpretation.
- Unrestricted File Size: A single J2ME application file must be able to span several Flash blocks. An upper limit to the size of the file has not been identified.
- Flash Storage During Interpretation: During J2ME application interpretation, another J2ME application file may be stored to Flash memory, as well as other standard FDI storage.
- Java Data Access: A J2ME application may create and access application data. The data size is not known in advance. The application data is stored in the standard FDI data volume, not in the XIP direct access volume.
- Inter-file Access: A Class Object of a Class Unit needs to be able to reference a Class Object within itself or of an Installed Library.
- File Position: A Class Unit file must be able to be deleted or moved. An Installed Library file should not be moved or deleted if a Class Unit reference to it exists.

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

- Pointer Fix-Ups: The internal pointers of a Class Unit require patching after Reclaim moves the unit. The pointer fix-ups are performed by the KVM in RAM on the entire Class Unit. After internal pointer fix-ups, the Class Unit in RAM must be written to the Class Unit destination in Flash memory determined by Reclaim.
- Reclaim Timing: When the J2ME application uninstaller deletes a Class Unit, all free space in Flash memory must be made available prior to the next installation.
- J2ME Storage Volumes: Two storage volumes are needed to separate Installed Libraries versus Class Units, so that the Class Units can be moved without moving Installed Libraries.
- FDI Volumes: The J2ME application files must be stored in a different volume than the critical system data that FDI manages via standard parameters.
- Power-Loss Recovery: Complete power-loss recovery is required to prevent fatal data corruption caused by fluctuations or interruptions of the power source.

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

FDI Code Manager (CM) Description

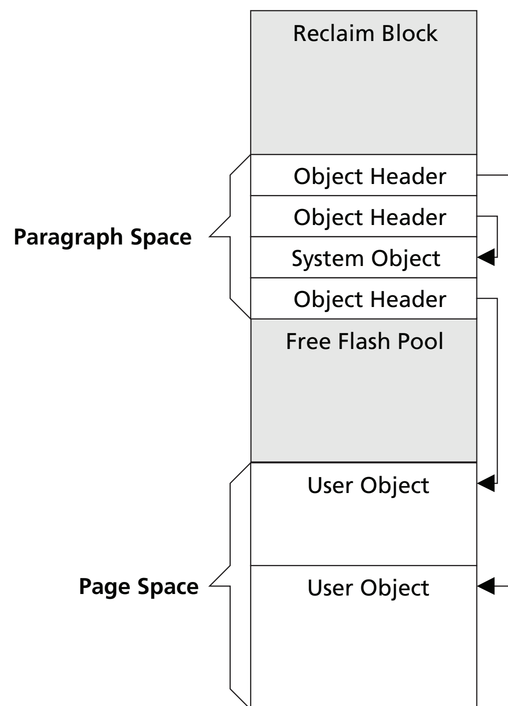
Code Manager Background

In response to the requirements for nonvolatile storage of J2ME applications, FDI now features the Code Manager software that manages the Direct Access Volume (DAV) of the Flash memory subsystem.

Code Manager Organization in Flash

The DAV partition managed by the Code Manager is divided into sections, each of which has individual requirements and responsibilities.

Figure 5: DAV Organization in Flash



- **Reclaim Block:** Unlike DRAM, the very nature of Flash memory requires that physical blocks be erased before data is programmed into them. Due to this limitation, when data in Flash memory is no longer useful, it is de-allocated (marked invalid). To write new data over de-allocated data, the file system must erase the physical block in which the de-allocated data resides. Since FDI manages Flash memory in virtual, software-managed blocks, it is possible a physical block contains both de-allocated data and valid data still in use. To write new data over the de-allocated space, the valid data must first be moved out of the way to initiate the physical block erase. FDI reserves the reclaim block to hold the valid data while it erases a physical block. Again, Reclaim is the process by which FDI removes de-allocated data from Flash memory, thus freeing up the de-allocated space for more data storage. The reclaim block is reserved for use solely by FDI and cannot be used by applications. The reclaim block has a fixed size equal to the largest virtual block in the object space.
- **Paragraph Space:** Paragraph space consists of a list of header entries for the page and paragraph objects. Paragraph objects consist of small files. The paragraph object header entries are immediately followed by the paragraph object data.

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

- **Free Flash Pool:** The free Flash pool is the area of unused Flash space that resides between the page object space and the paragraph object space. The free Flash pool space begins and ends on virtual block boundaries. When either object space needs more Flash memory space, it takes an entire virtual block from the free Flash pool.
- **Page Space:** Page space consists of large data objects. However, the header entries for page objects are stored in the paragraph space.
- **Object Header:** Object headers are paragraph objects that contain information regarding a page or paragraph object. Header information consists of object type, name, size, miscellaneous attributes, and status.
- **System/User Object:** Other data objects, such as user data or J2ME applications, can be stored in a contiguous fashion for direct access.

Code Manager Functional Overview

The Code Manager software consists of five main functional concepts: allocation, reallocation, object management, reclamation, and recovery. Each of these functions is described in the following sections.

Allocation

Allocation is the process by which FDI stores a new file to the DAV partition managed by the Code Manager. The allocation process is as follows:

- FDI confirms that enough free space is available to store the object.
- FDI creates the header associated with the object.
- FDI determines in which space (page versus paragraph) to store the object.
- FDI writes the object data to Flash memory.

The allocation management functions provided by FDI are:

- FDI_AllocateFlash( ): Creates a file referenced by filename.
  - Input Elements:
    Object name
    Name size
    Object type
    Object size
  - Output Elements:
    Base address of object
    Error status
  - Processing Characteristics:

  The filename can be up to 255 bytes long. The file data is stored in a contiguous fashion. This function builds the management structures in Flash memory, reserves the file space, and may reclaim Flash memory space if necessary.
- FDI_WriteObject( ): Writes the file data to the files data field. This function is repeatedly called as necessary to write the file data.
  - Input Elements:
    Object name
    Name size
    Object type
    Data buffer
    Number of bytes to write
    Offset from base address to write bytes
  - Output Elements:
    Error status

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

– Processing Characteristics:

This function writes the specified number of bytes to the location specified for the object.

- FDI_WriteComplete( ): Marks a file as valid, and is called after all file data has been written.
  - Input Elements:
    Object name
    Name size
    Object type
  - Output Elements:
    Error status
  - Processing Characteristics:

  This is a power loss recovery step. This step validates the file data in Flash memory.

Reallocation

Reallocation is the process by which FDI updates an existing object with new data. As a requirement, the new data must be written in the same physical address space as the original object. Thus, the space associated with the original object must be erased. Reallocation provides two options, use of reserved Flash memory and saving of the original object. Flash memory can be reserved for an object to be updated even during full memory conditions. This allows the Code Manager to guarantee that an object may be updated. Reallocation can also save a copy of the original object until the user indicates to the Code Manager that it is finished updating the new data. Once the user specifies it is done updating the new data, the Code Manager will delete the original object (which is stored in a temporary space).

The re-allocation management function is:

- FDI_ReAllocateFlash( ): Prepares an existing object to be updated.
  - Input Elements:
    Object name
    Name size
    Object type
  - Output Elements:
    Base address of object
    Object size
    Error status
  - Processing Characteristics:

This function effectively deletes the data field of the object and replaces it with the new data. The new data is written via the FDI_WriteObject( ) and FDI_WriteComplete( ) functions.

Object Management

Object management involves maintaining an object that already exists in the Flash media. The generic functions supported by object management are: object de-allocation, reading an object, getting the header of an object, memory statistics, and moving an object in the managed partition.

The de-allocation management function is:

- FDI_DeAllocateFlash( )
  Deletes an existing object in Flash memory.
  - Input Elements:

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

Object name
Name size
Object type

– Output Elements:

Error status

– Processing Characteristics:
This function marks a file as invalid, or no longer in use. During reclamation, invalid files are erased from Flash memory and free space is recovered.

The read function is:

- FDI_ReadObject( )
Reads an existing file's data field.
  – Input Elements:
  Object name
  Name size
  Object type
  Data buffer to place the bytes
  Number of bytes to write
  Offset from base address to read bytes from
  – Output Elements:
  Buffer containing bytes read from Flash memory
  Error status
  – Processing Characteristics:
  Copies bytes from Flash memory to the buffer provided.
The management function that searches for a file in Flash memory is:
- FDI_GetObjectHeader( )
Searches for the file specified and returns the header information.
  – Input Elements:
  Object name
  Name size
  Object type
  Search method
  Search starting point
  – Output Elements:
  Object name
  Name size
  Object type
  Object address
  Object size
  Object write status
  Error status
  – Processing Characteristics:
  This function locates the complete file information based on a set of criteria inputs, and provides the following search capabilities:
  Find the file specified by name and type
  Find the first file matching specified name
  Find the first file matching specified type
  Find the next file matching the criteria

The management function that provides memory statistics is:

- FDI_GetMemoryStatistics( )
Provides memory statistical information for the DAV partition.

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

  - Input elements:
    Pointer to statistics buffer
  - Output elements:
    Memory statistics to buffer
    Error status

Reclamation

There are three types of reclaim discussed in this document: object reclaim, reclaim in place, and configuration header reclaim. **Object reclaim** refers to the process by which FDI erases space associated with page and paragraph objects that have been de-allocated. **Reclaim in place** is the process by which FDI erases space associated with any single object.

Both forms of reclaim have failure-recovery mechanisms in place, in which a reclaim table is used to store reclaim-recovery information. The configuration header is used to store the location of the reclaim table. This allows the recovery mechanisms to locate the reclaim table no matter in what state the system is. Eventually the configuration header will run out of room to store the location of the reclaim table. When this happens, FDI invokes a configuration-header reclaim. A special reclaim is performed because the configuration header must always reside in the object list as the first object, and no reclaim table can be used to track failure-recovery mechanisms. Currently, reclaim is forced every time an object is deleted. This has the effect of maintaining a clean storage space to accelerate the allocation process.

Reclaim must be locked while the KVM is executing a J2ME application. The FDI/Code Manager API is semaphore-protected to prevent reclaim during J2ME application execution.

The reclamation management functions provided by the FDI/Code Manager are:

- FDI_ReclLock( )
  Informs the Code Manager that it must request a reclaim before proceeding.
  - Input Elements:
    None
  - Output elements:
    None
- FDI_ReclUnlock( )
  Informs the Code Manager that it does not need to request a reclaim before proceeding.
  - Input Elements:
    None
  - Output elements:
    None

Also necessary is a management function that supports the relocation and subsequent internal pointer fix-ups. This function is not implemented as part of FDI. The mobile/embedded device manufacturer has the responsibility to provide a function to update the internal J2ME application pointers as a J2ME application object is relocated in the Flash media. Only the prototype has been specified as follows.

The object move/modify function prototyped for the FDI/Code Manager is:

- FDI_ModifyObject( )
  Pointer fix-up feature supported by implementing an object-tracking mechanism during reclaim.
  - Input elements:

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

New address
Object size
Old address

– Output elements:
Error status

Recovery

Internal to FDI, objects are stored with additional information that allows FDI to maintain the integrity of the object through an interruption or fluctuation of power. If an object is in the middle of modification and a power loss occurs, FDI guarantees that the original object will be preserved or the new object will be stored successfully. The details of recovery are very complex and are explained in detail in the Direct Access Volume Architecture Definition.

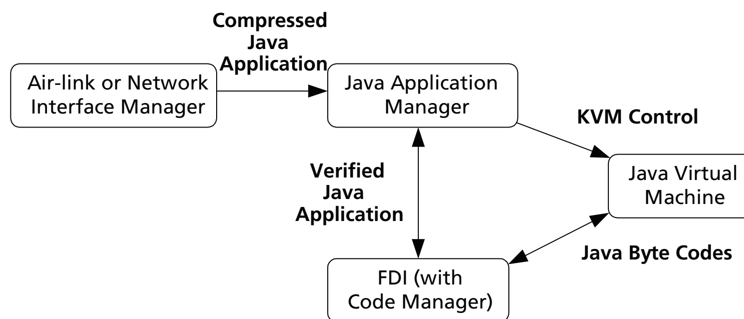
Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

Integration of J2ME Application Support in FDI

Introduction

Code Manager and the Direct Access Volume were specifically added to Micron Flash Data Integrator to assist the mobile/embedded device manufacturer in the creation of J2ME-compliant devices. By 1) utilizing a commercial RTOS that contains a J2ME-compliant Java virtual machine, and 2) integrating FDI as the Flash media manager, the remaining task for the OEM is to develop the Java Application Manager. There are several scenarios that the J2ME environment presents to the JAM developer. The following thread-interaction diagram will be used in describing the operational scenarios:

Figure 6: MIDP Thread Interaction Diagram



- The **Air-link or network interface manager** is the means for Java applications to be delivered to the device.
- The **Java virtual machine** is the interpreter that executes the Java byte codes.
- **FDI** handles the nonvolatile storage requirements of Java applications in the Flash media.
- The **Java Application Manager** is responsible for receiving the compressed Java application, verifying the Java application, storing the Java application, and initiating execution of the application on the Java virtual machine.

To facilitate porting of the KVM to small, resource-constrained platforms, the KVM implementation contains an optional component called Java Application Manager (JAM) that can be used as the starting point for machine specific implementations.

The KVM porting guide provides a reference implementation of the JAM provided with the KVM. The JAM is a native C application that is responsible for downloading, installing, inspecting, launching, and uninstalling Java applications. Due to the large range of potential implementations, the Java Application Manager must be developed by the OEM.

The reference implementation provided with the KVM porting guide is targeted to a desktop PC, and as such, relies heavily on the file management functions provided by the desktop OS. In a resource-constrained environment that lacks a file system, the Micron Flash Data Integrator can be used to support the application lifecycle management functions specified for the JAM. Specifically, FDI can be used to support the installing, inspecting, launching, and uninstalling J2ME Java applications.

The OEM (the reader of this document) must take the responsibility to develop the Java Application Manager. The Java Application Manager specified in this document is *not* identical to the Java Application Manager reference design. An FDI-optimized Java Application Manager is an extension of the reference design, designed to specifically

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

utilize the FDI API to enhance the storage mechanism of a J2ME-compliant system. The following sections specify various scenarios in the J2ME environment, and how each scenario can take advantage of FDI. Included in each scenario specification is a description of the scenario, detailed step-by-step instructions on how to implement the necessary code, and a flow diagram to graphically represent the interaction between the threads involved.

Implementation Task Checklist

The following is a list of scenarios that must be implemented in the Java Application Manager by the OEM. This checklist can be used to keep track of which functions have been completed in the development of the Java Application Manager.

- Library Installation
- Library Modification (Addition)
- Library Modification (Remove/Replace)
- Managed Library Installation
- Managed Library Modification (Addition)
- Managed Library Modification (Remove/Replace)
- Direct Access Volume Initialization
- Application Storage
- Application Removal
- Application Reclamation
- J2ME Application Listing
- Application Initiation
- Compressed Application Storage
- Application Resolution

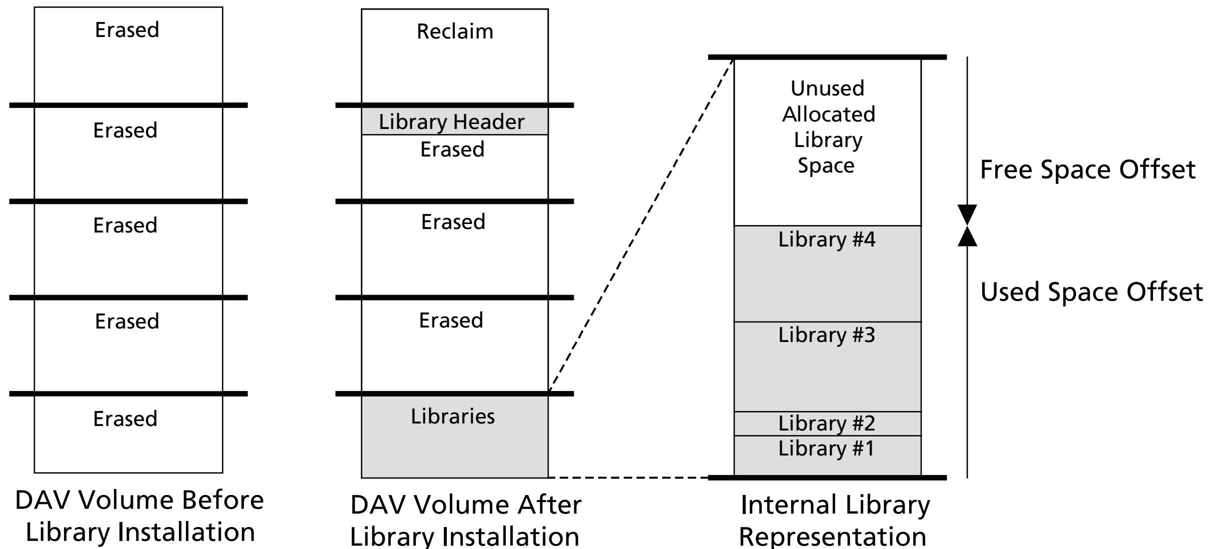
Library Installation

The basis of a Java execution environment is the installed libraries that form the foundation upon which third-party applications rely. This operational scenario describes the steps necessary to install the Java foundation libraries into the execution environment. The Java specification states that the Java application should be downloaded to the device and then executed. The Java application, however, inherits a large amount of functionality from the installed libraries. To avoid repeatedly downloading the libraries required by the application, the libraries can be stored in Flash memory. To simplify the library management and the ability to upgrade them, the libraries should be installed before any applications are installed, and should pre-allocate a complete Flash memory block for the libraries. Note that the installed libraries consume less than a complete block. This is intended, and will simplify future management of the libraries. The Java Application Manager must maintain the current size of the installed libraries and the available library space. Library versions can change and new libraries may become necessary in the future.

Figure 7 represents the physical structure and management of the installed libraries:

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

Figure 7: Physical Representation of DAV during Library Installation



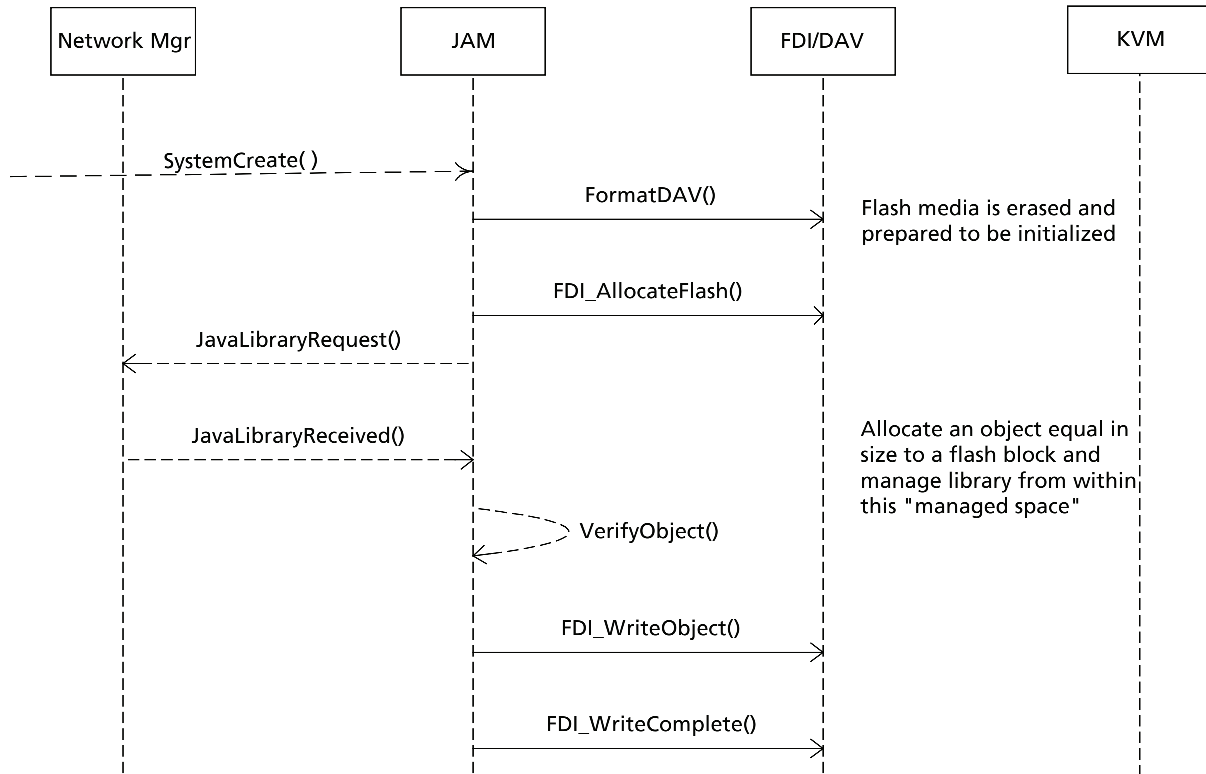
Initially, the Flash volume is completely erased (or should be). The JAM pre-allocates a complete Flash memory block for the libraries. The JAM also maintains a pointer or counter to the end of the used space. This pointer/counter will be used to append libraries and to determine the amount of space available. Installing the libraries first is the only way to guarantee that the libraries will be installed at the lowest physical address. Pre-allocating more space than necessary is important because this guarantees space for the libraries to grow and maintain their physical location at the low address. Adding a library to the existing libraries and updating an existing library are discussed in later sections.

To install a library:

1. Make a system create call to the JAM. This may be a manufacturing step.
2. The JAM formats (erases) the DAV and prepares the volume for installation of Java libraries.
3. Pre-allocate a complete Flash memory block for the libraries by the FDI call FDI_AllocateFLash( ).
4. The JAM requests the libraries from the network. Alternately, during the manufacturing process, the request is made to a manufacturing network connection.
5. The protocol for the library request includes a mechanism to determine the library size.
6. Decompress the received libraries.
7. Scan the received libraries for errors.
8. Verify the received libraries.
9. Call the FDI function FDI_WriteObject( ) as necessary to copy the libraries to Flash memory.
10. After all of the libraries have been copied to Flash memory, call the function FDI_WriteComplete( ) to inform FDI that the libraries have been successfully installed. This causes internal Code Manager status information to latch, locking the libraries into the DAV.

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

Figure 8: Library Installation Operational Scenario



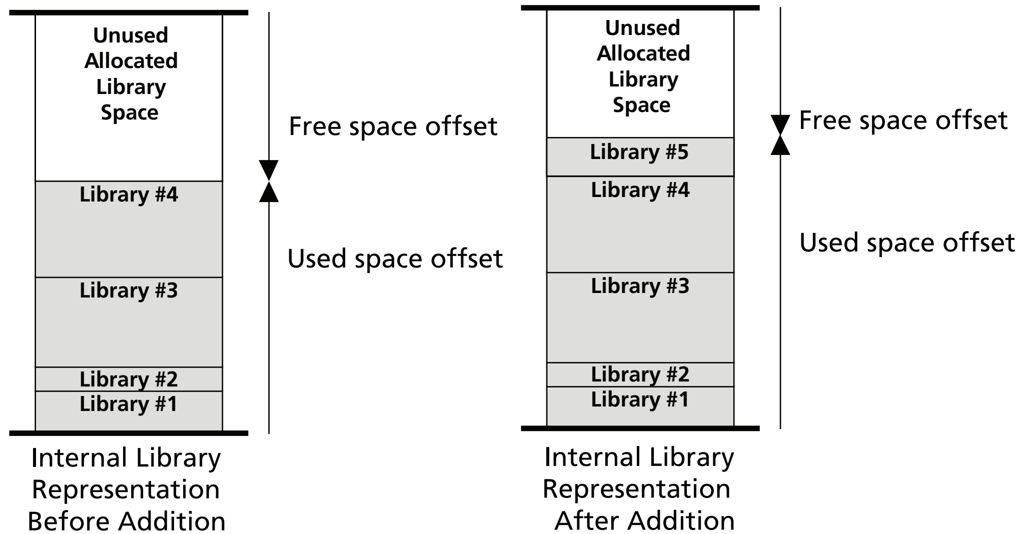
Library Modification - Addition

During the life of a mobile/embedded device, the Java library foundation is likely to change. This may require maintenance of the libraries that are already installed. The simplest form of library maintenance is to add a library. This modification simply appends the currently installed libraries with the new library. This is acceptable because excess space is reserved during the original library installation. The JAM is responsible for updating the pointer/counter to the end of the used space, to reflect that additional libraries have been installed. This scenario assumes that the library installation (“Library Installation” on page 24) and the DAV initialization (“Direct Access Volume Initialization” on page 36) have previously executed without error.

Figure 9 represents the physical structure and management of the internal libraries before and after adding a library:

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

Figure 9: Physical Representation of DAV during Library Addition

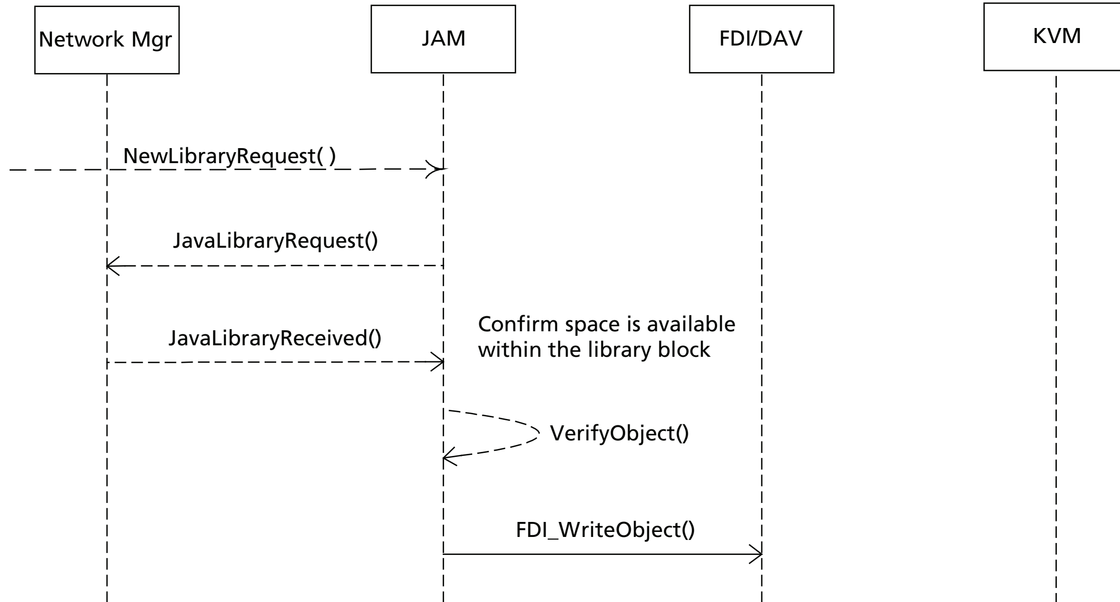


To add a library:

1. Make a system call to the JAM to inform it that a new library is needed.
2. The JAM requests the new library from the network.
3. The protocol for the library request includes a mechanism to determine the library size.
4. The Java Application Manager confirms that there is enough space available for the new library.
5. Decompress the received library.
6. Scan the received library for errors.
7. Verify the received library.
8. Call the FDI function FDI_WriteObject( ) as necessary to copy the library into Flash memory.

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

Figure 10: Library Addition Operational Scenario

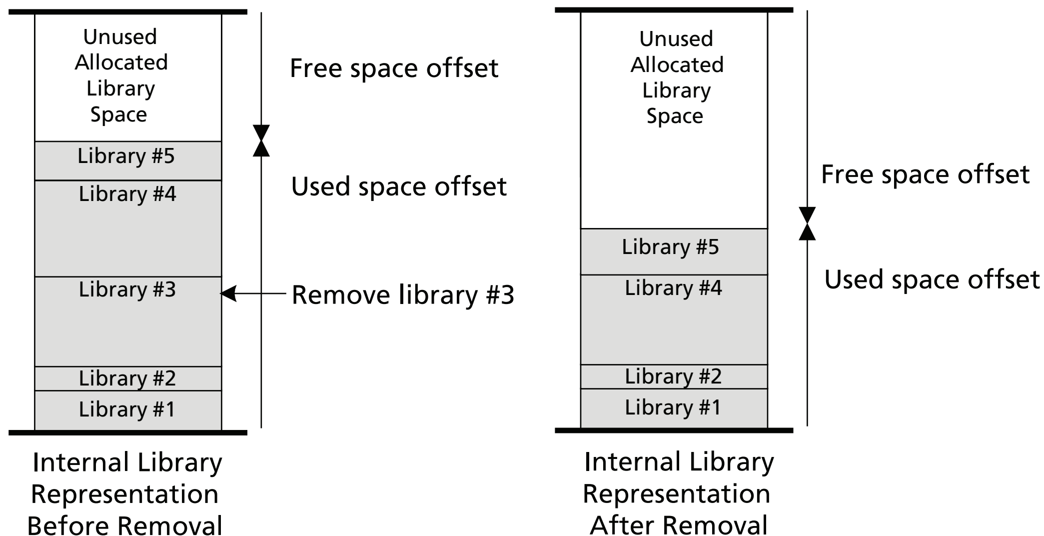


Library Modification - Removal/Replacement

A more difficult form of library modification is to replace or remove a library within the existing library space. This requires the JAM to copy all of the libraries to RAM, reassemble them in RAM, then copy the reassembled library array back into Flash memory. This operation relies heavily on the JAM. This scenario assumes the library installation ("Library Installation" on page 24) and the DAV initialization ("Direct Access Volume Initialization" on page 36) have previously executed without error.

Figure 11 represents the physical structure and management the internal libraries before and after removal or replacement of a library:

Figure 11: Physical Representation of DAV during Library Removal/Replacement



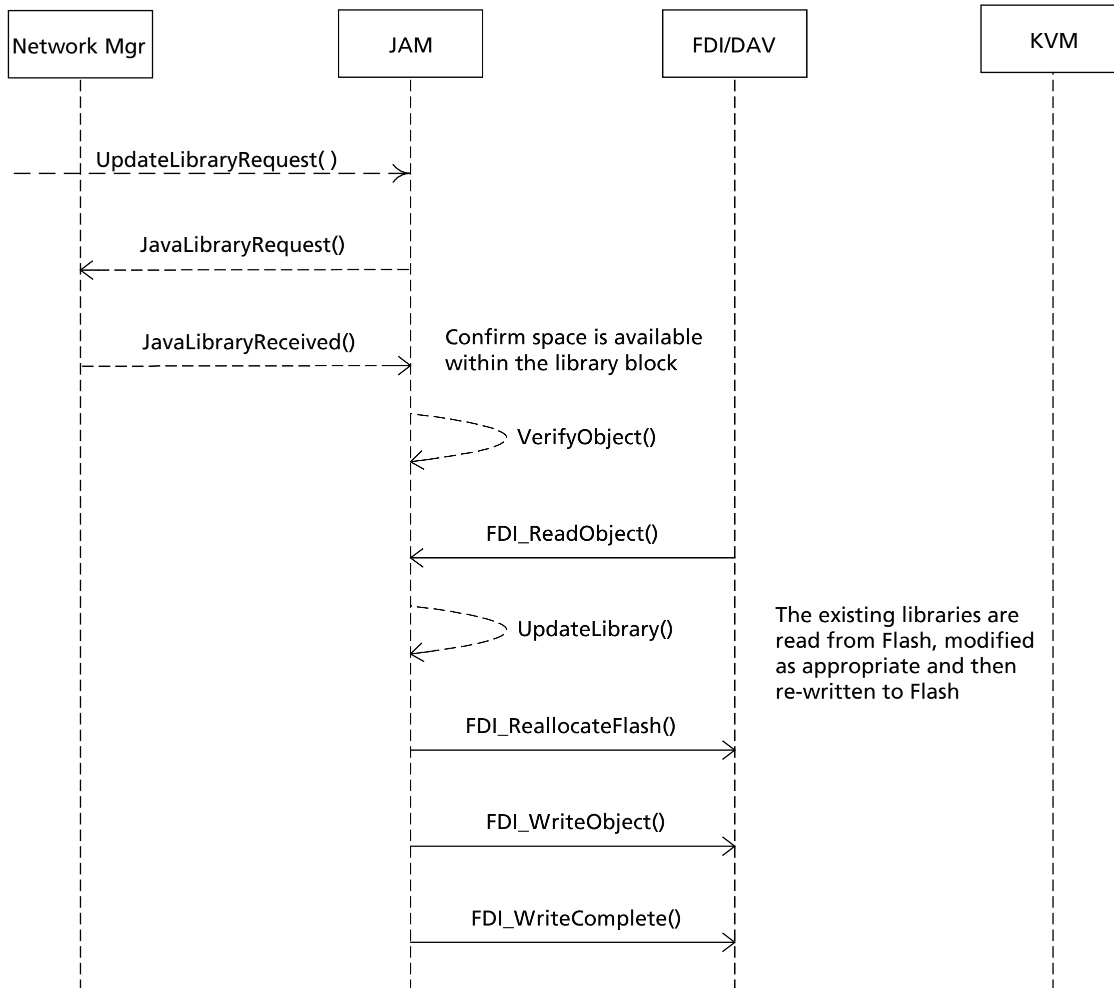
Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

To remove/replace a library:

1. Make a system call to the JAM to inform it that a library must be updated.
2. The JAM requests the new library from the network.
3. The protocol for the library request includes a mechanism to determine the library size.
4. The JAM confirms that there is enough free space available for the new library.
5. Decompress the received library.
6. Scan the received library for errors.
7. Verify the received library.
8. Copy the existing libraries from Flash memory to RAM by calling the FDI_ReadObject( ) function to obtain the libraries.
9. Replace the old library with the new library in RAM and reassemble the libraries as necessary.
10. Call the FDI function FDI_ReAllocateFlash( ) to erase the existing libraries in Flash memory. If a power failure occurs while the JAM is updating the libraries, then FDI will restore the old libraries on power-up initialization.
11. Call the FDI function FDI_WriteObject( ) as necessary to copy the updated libraries into Flash memory.
12. After all of the libraries have been copied to the Flash, call the function FDI_WriteComplete( ) to inform FDI that the libraries have been successfully updated.

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

Figure 12: Library Removal/Replacement Operational Scenario



Managed Library Installation

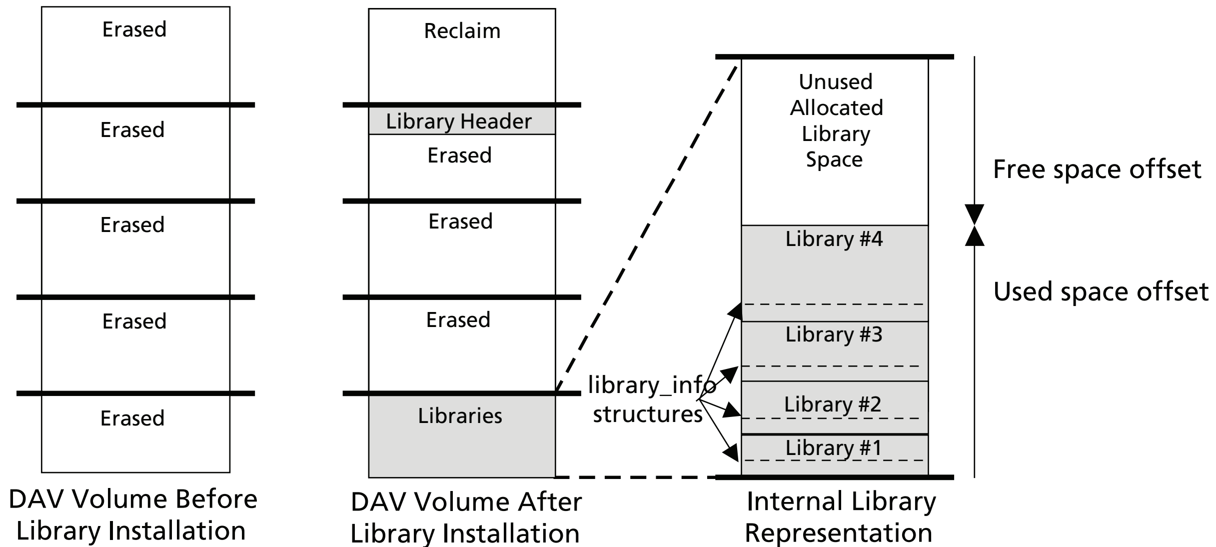
Managed libraries include all of the information necessary to manage the library within the library block. Additional information is stored along with each Java library to aid the management of the library block. Essentially, a library information structure is inserted at the beginning of each library class that contains the name, version, and size of the following library classfile, along with a pointer to the next library structure location. This creates a linked list of structures that maintain all of the information necessary to manage the libraries within the library block.

Note: Use of the managed library space is necessary only if the JAM does not maintain the configuration information for the installed libraries.

Figure 13 represents the physical structure and management of the installed managed libraries:

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

Figure 13: Physical Representation of DAV during Library Installation



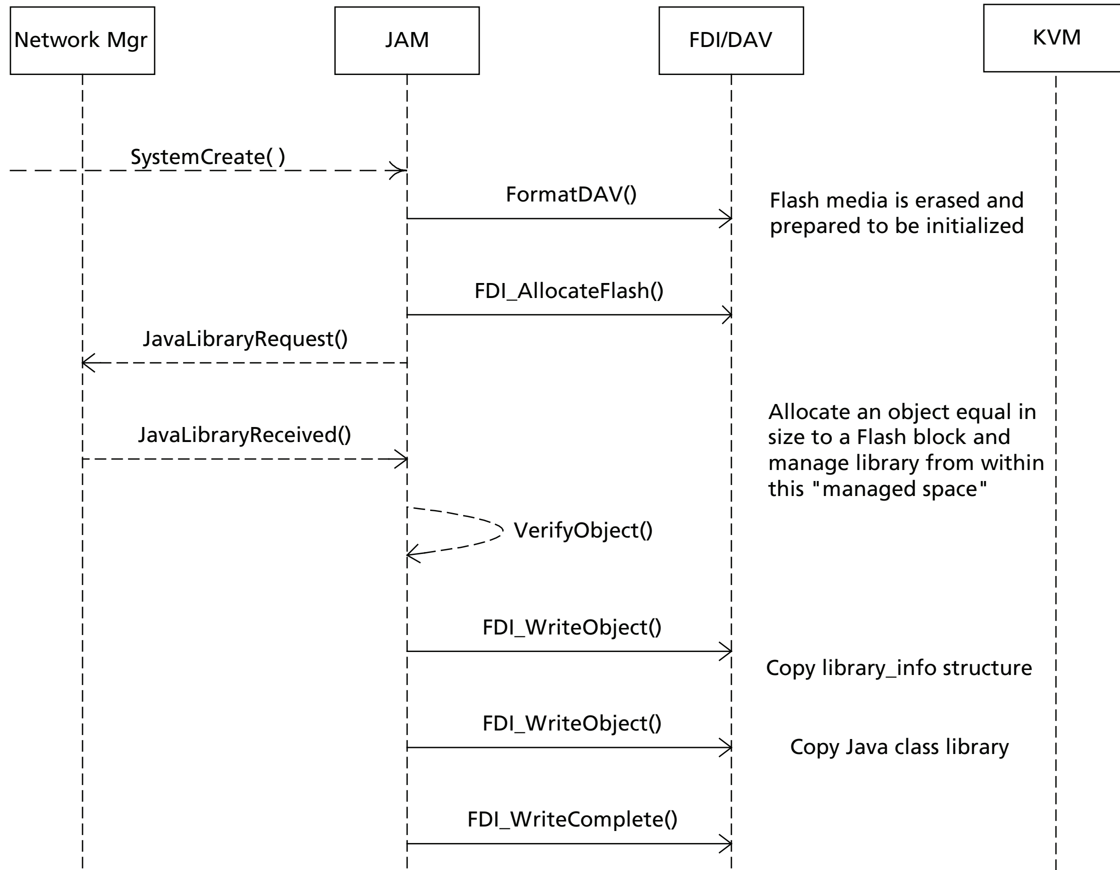
Initially, the Flash volume is completely erased (or should be). The JAM pre-allocates a complete Flash memory block for the libraries. For managed libraries, the JAM will not be required to maintain a pointer or count to the end of the used space. The information necessary to obtain library block configuration is stored within the library block. Pre-allocating more space than necessary is important because this guarantees space for the libraries to grow and maintain their physical location at the low address. Adding a library to the existing managed libraries and updating an existing managed library are discussed in later sections.

To install a managed library:

1. Make a system create call to the JAM.
2. The JAM formats the DAV and prepares the volume for installation of Java libraries.
3. Pre-allocate a complete Flash block for the libraries using the FDI function FDI_AllocateFLash( ).
4. The JAM requests the libraries from the network. Alternately, during the manufacturing process, the request is made to a manufacturing network connection.
5. The protocol for the library request includes a mechanism to determine the library size.
6. Decompress the received libraries.
7. Scan the received libraries for errors.
8. Verify the received libraries.
9. For managed libraries, call the FDI function FDI_WriteObject( ) to write the library_info structure immediately before the Java Class Library.
10. Call the FDI function FDI_WriteObject( ) as necessary to copy the libraries into Flash memory.
11. Repeat steps 9 and 10 as necessary.
12. After all of the libraries have been copied to the Flash, call the function FDI_WriteComplete( ) to inform FDI that the libraries have been successfully installed. This will cause internal status information to latch, locking the libraries into the DAV.

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

Figure 14: Managed Library Installation Operational Scenario

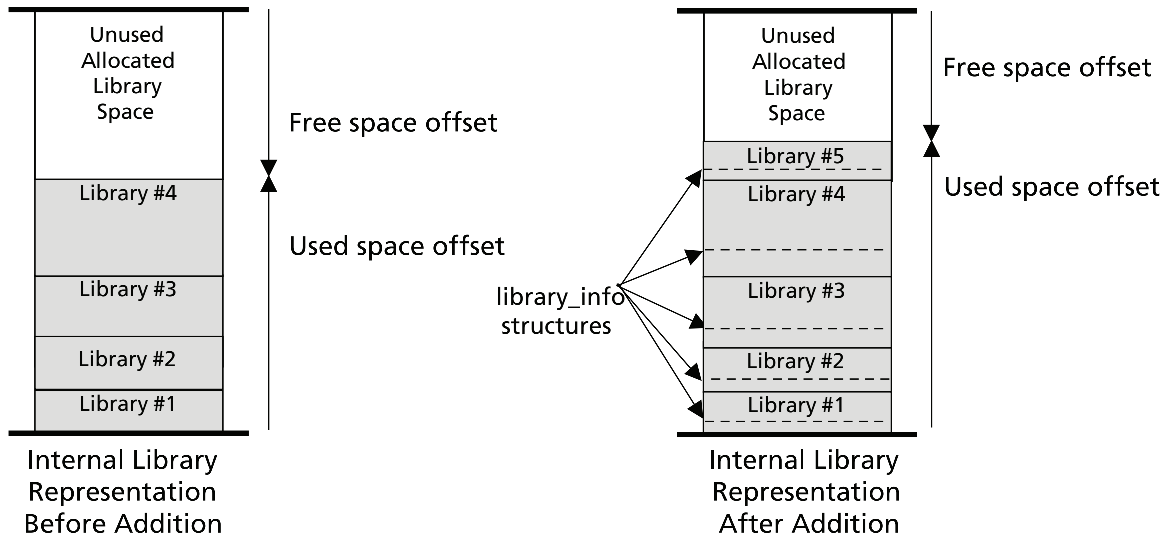


Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

Managed Library Modification - Addition

Figure 15 represents the physical structure and management the internal libraries before and after addition of a library to the managed library space:

Figure 15: Physical Representation of DAV during Managed Library Addition

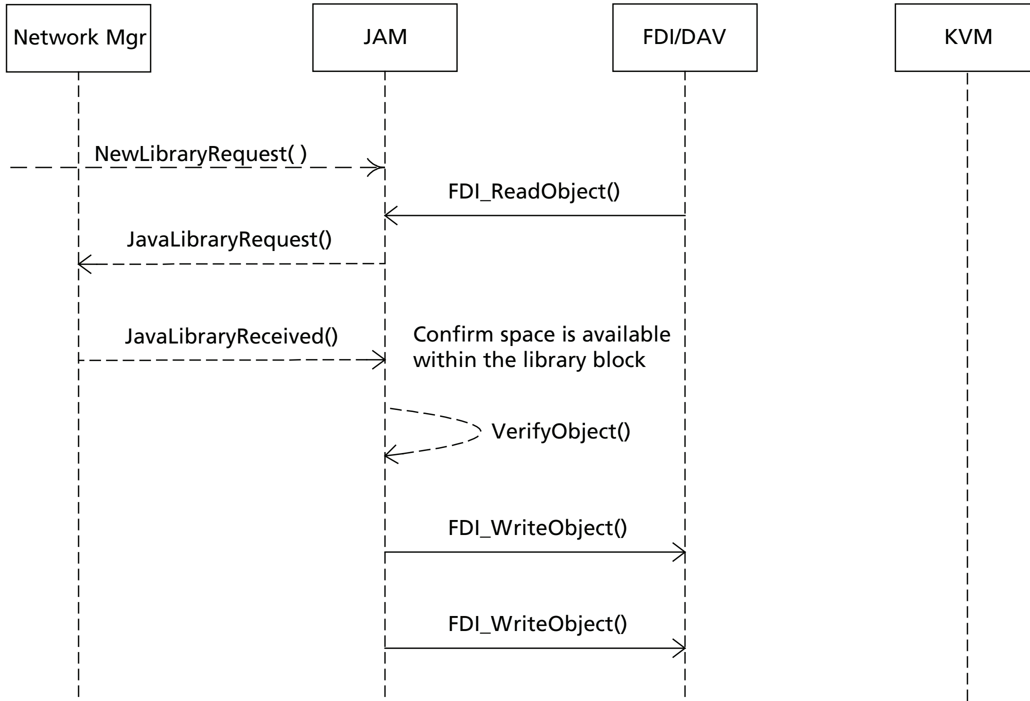


To add a library:

1. Make a system call to the JAM to inform it that a new library is needed.
2. The JAM scans the installed libraries, by indexing through the library_info structures, to determine the current state of the library block.
3. The JAM requests the new library from the network.
4. The protocol for the library request includes a mechanism to determine the library size.
5. The JAM confirms that there is enough space available for the new library.
6. Decompress the received library.
7. Scan the received library for errors.
8. Verify the received library.
9. For managed libraries, call the FDI function FDI_WriteObject( ) to write the library_info structure immediately before the Java Class Library.
10. Call the FDI function FDI_WriteObject( ) as necessary to copy the library into Flash memory.
11. Repeat steps 9 and 10 as necessary.

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

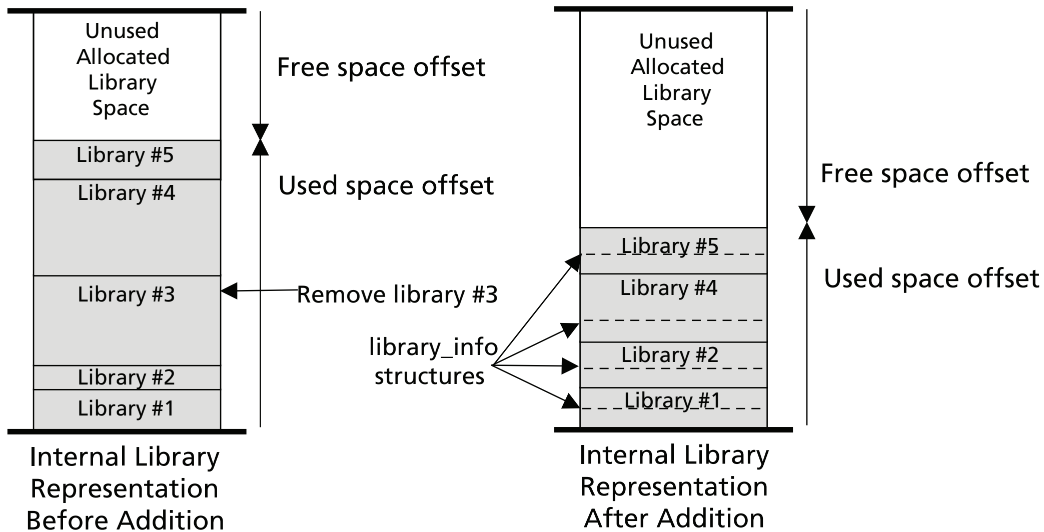
Figure 16: Managed Library Addition Operational Scenario



Managed Library Modification - Remove/Replace

Figure 17 represents the physical structure and management of the internal libraries before and after removal or replacement of a library within the managed library space:

Figure 17: Physical Representation of DAV during Library Removal/Replacement



To remove/replace a library:

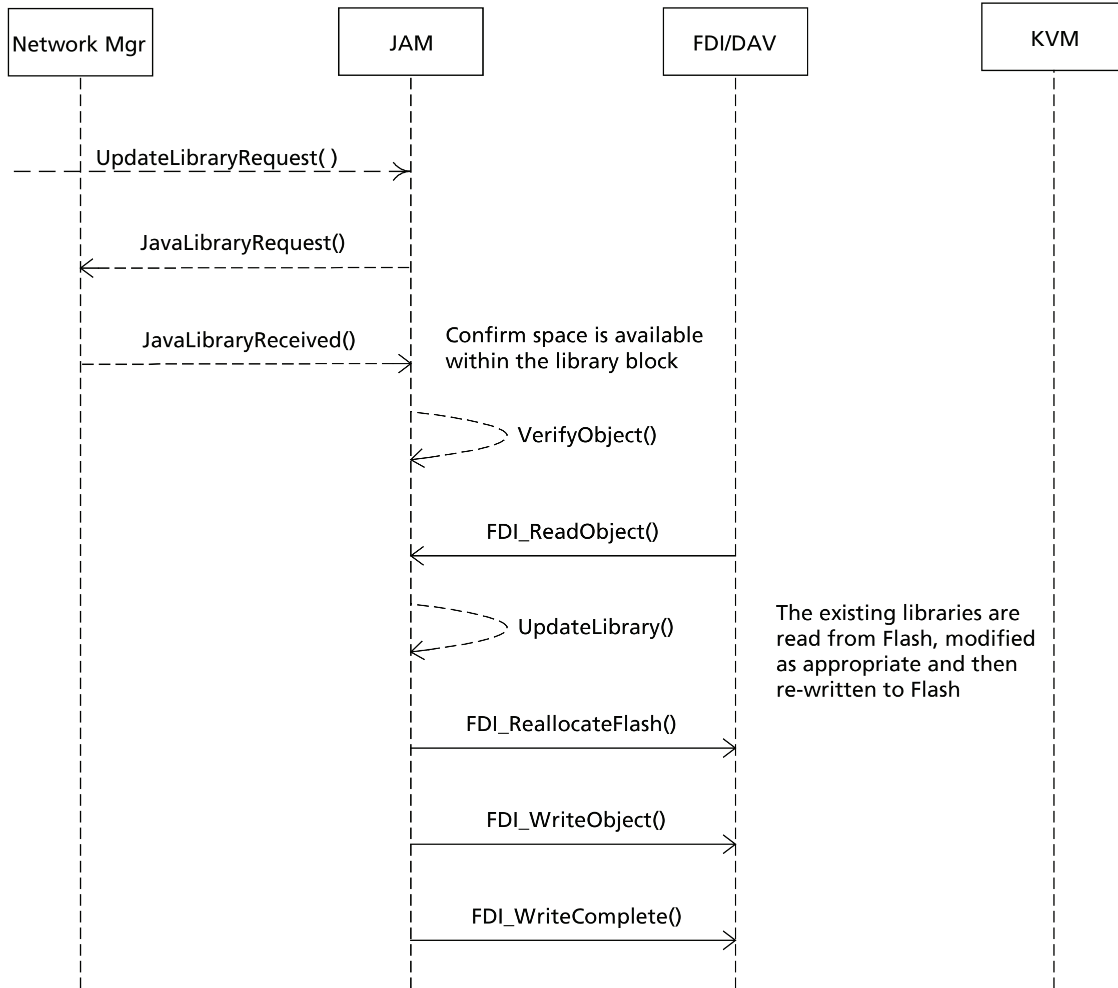
1. Make a system call to the JAM to inform it that a library must be updated.

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

2. The JAM scans the installed libraries by indexing through the library_info structures to determine the current state of the library block.
3. The JAM requests the new library from the network.
4. The protocol for the library request includes a mechanism to determine the library size.
5. The JAM confirms that there is enough space available for the new library.
6. Decompress the received library.
7. Scan the received library for errors.
8. Verify the received library.
9. Copy the existing libraries from Flash memory to RAM by calling the FDI function FDI_ReadObject( ) to obtain the libraries.
10. Replace the old library with the new library in RAM and reassemble the libraries as necessary.
11. Call the FDI function FDI_ReAllocateFlash( ) to erase the existing libraries in Flash memory. If a power failure occurs while the JAM is updating the libraries, then FDI will restore the old libraries on power-up initialization.
12. For managed libraries, call the FDI function FDI_WriteObject( ) to write the library_info structure immediately before the Java Class Library.
13. Call the FDI function FDI_WriteObject( ) as necessary to copy the updated libraries into Flash memory.
14. Repeat steps 12 and 13 as necessary.
15. After all of the libraries have been copied to Flash memory, call the function FDI_WriteComplete( ) to inform FDI that the libraries have been successfully updated.

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

Figure 18: Managed Library Removal/Replacement Operational Scenario



Direct Access Volume Initialization

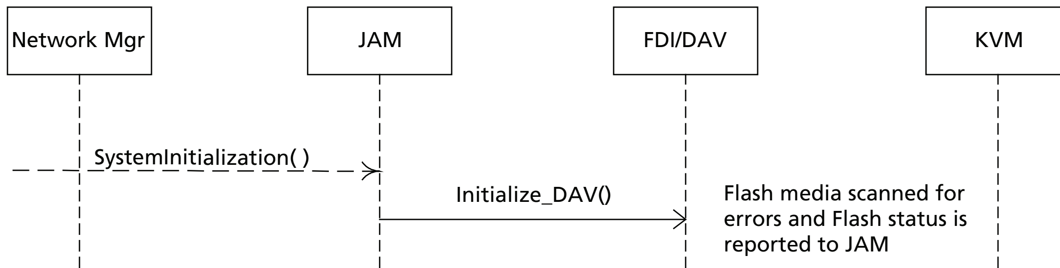
After the Java libraries have been installed, and any time the mobile/embedded device is powered-on, the DAV must be scanned for errors. An error could be present if there was power loss in the process a write or reclaim. Internally, FDI can handle any error caused by power loss by scanning the Flash memory and fixing corruptions. This scenario illustrates the process steps necessary for the JAM to inform FDI that power has been applied and to perform the power-on scan for errors.

To initialize the DAV volume:

1. Make a system initialization call to the JAM.
2. The JAM performs any necessary internal power-up processing.
3. The JAM calls the FDI function Initialize_DAV( ) to allow FDI to initialize and scan the Flash memory for errors. All errors are fixed during the scan.

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

Figure 19: DAV Initialization Operational Scenario

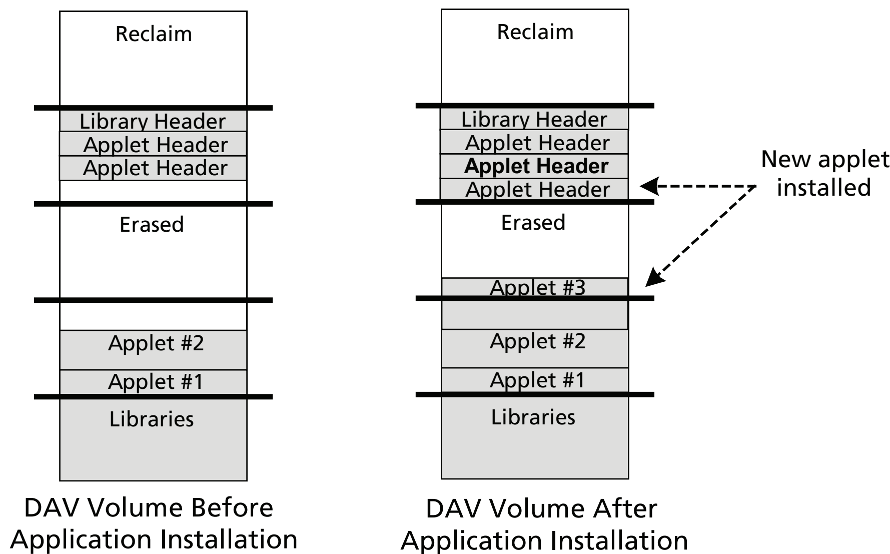


Application Storage

The application storage scenario executes whenever the mobile/embedded device owner requests an application to be downloaded to the device. This scenario assumes the library installation ("Library Installation" on page 24) and the DAV initialization ("Direct Access Volume Initialization" on page 36) have executed without error.

Figure 20 represents the physical structure and management of the application storage in DAV:

Figure 20: Physical Representation of DAV during Application Storage



To store an applet:

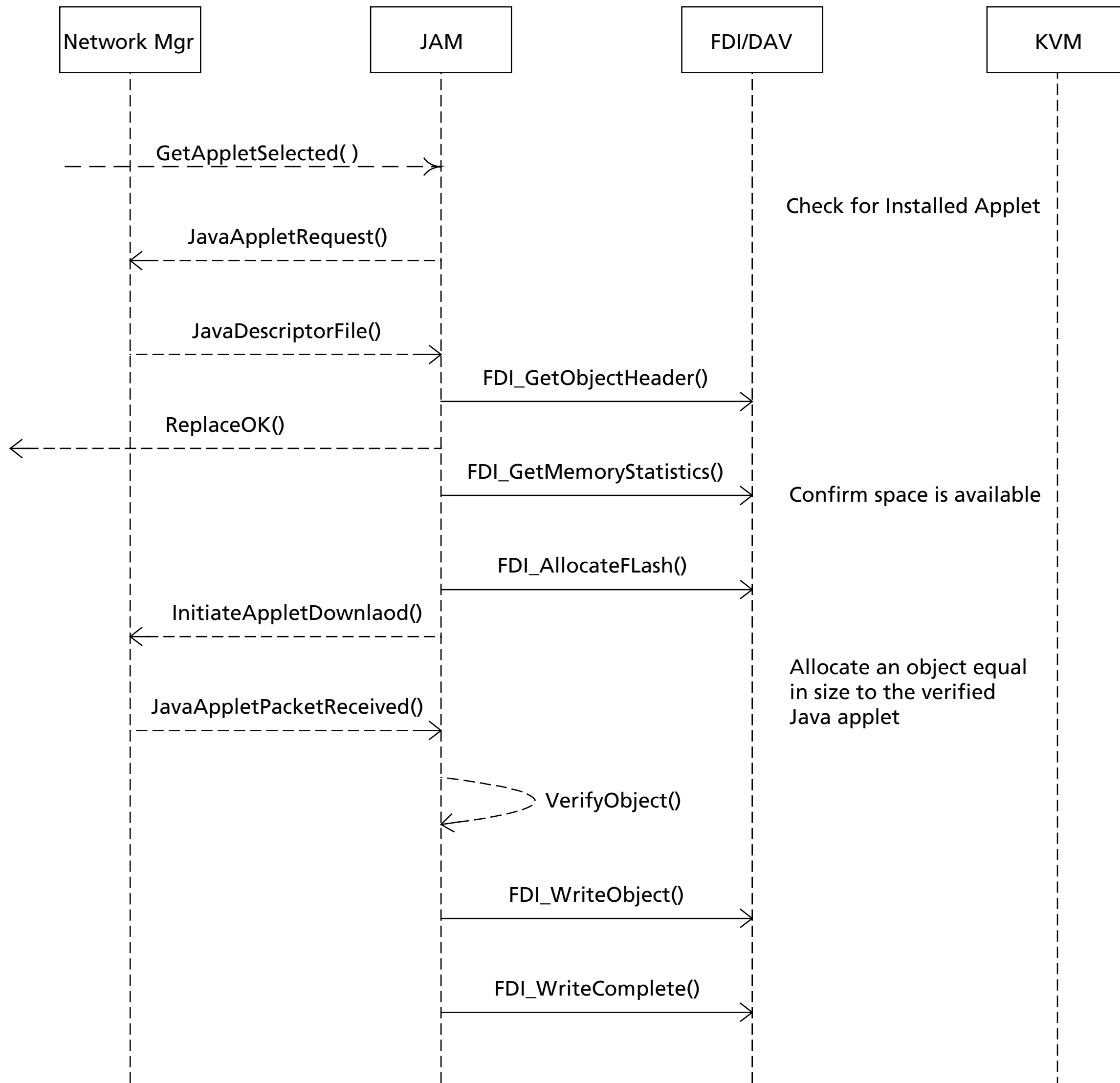
1. Make a system call to the JAM to inform it that the user has selected an application for download.
2. The JAM requests the Java descriptor file for the Java application from the network.
3. The JAM receives the Java descriptor file.
4. From the Java descriptor file, the JAM will obtain the file name, file version, and file size of the application. The JAM scans the Flash memory for an existing file with the same name as the requested file.
5. If a file exists in Flash memory and is newer than the requested file, the JAM ignores the download and continues with the application initiation ("Application Initiation" on page 43).

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

6. If the file exists in Flash memory and is older than the requested fie, the JAM prompts the user that an existing file will be replaced. If the user confirms, the download process proceeds. If the user denies, then the download process terminates.
7. If the file is not currently installed or the user has confirmed the replace prompt, the JAM confirms that there exists enough free Flash memory space for the requested application by calling the FDI function FDI_GetMemoryStatistics( ).
8. If enough space exists, FDI pre-allocates the necessary Flash memory space for the application through the call FDI_AllocateFlash( ).
9. Initiate download of the Java application.
10. Decompress the received application.
11. Scan the received application for errors.
12. Verify the received application.
13. Call the FDI function FDI_WriteObject( ) as necessary to copy the application download packets into Flash memory.
14. After the application has been copied to Flash memory, call the function FDI_WriteComplete( ) to inform FDI that the application has been successfully installed. This will cause internal Code Manager status information to latch, locking the application into the DAV.
15. Application execution continues with application initiation (“Application Initiation” on page 43).

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

Figure 21: Application Storage Operational Scenario



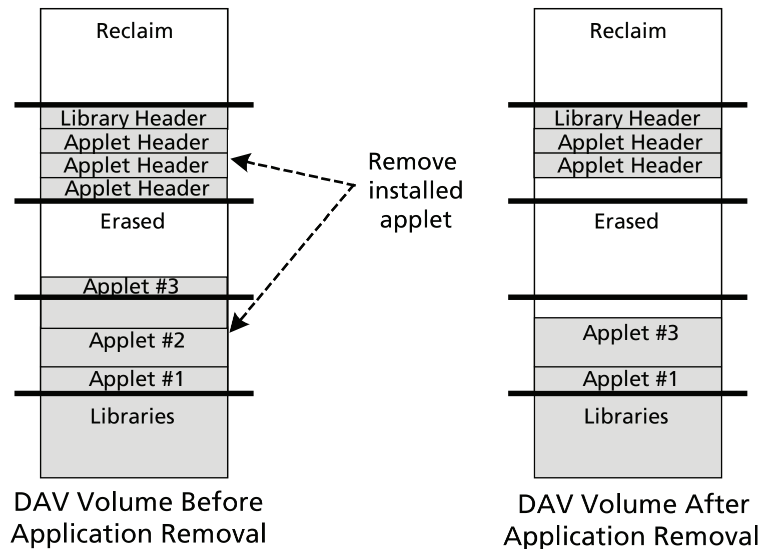
Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

Application Removal

The application removal scenario executes whenever the mobile/embedded device owner requests that an application be removed from the device. This scenario assumes the library installation (“Library Installation” on page 24) and the DAV initialization (“Direct Access Volume Initialization” on page 36) have executed without error.

Figure 22 represents the physical structure and management of the DAV before and after the application removal.

Figure 22: Physical Representation of DAV during Application Removal



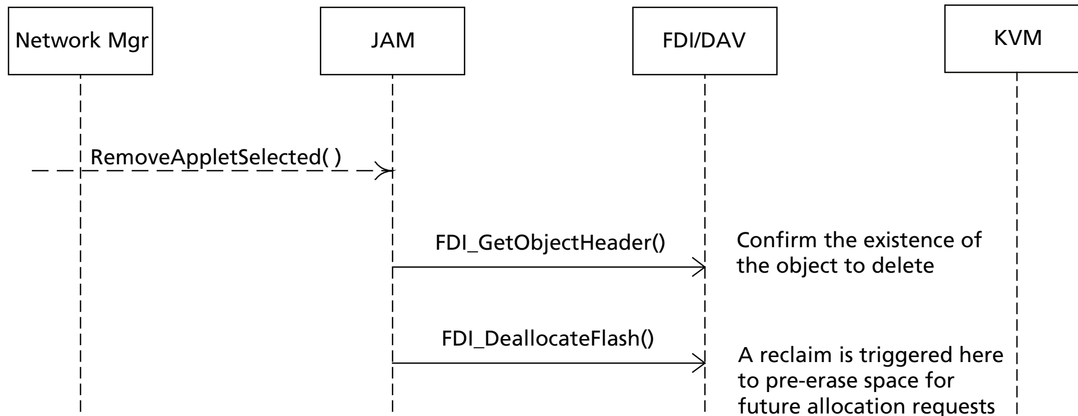
To remove an application:

1. Make a system call to the JAM to inform it that the user has selected an application for removal.
2. The JAM then confirms that the application still exists in the DAV by calling the FDI function FDI_GetObjectHeader( ).
3. If the application still exists, then the JAM calls the FDI function FDI_DeallocateFlash( ) to remove the application from the DAV.

Removing an application will trigger a reclaim. This reclaim is performed to clean up the Flash memory and to create space for future application downloads. As the reclaim proceeds, the application reclamation scenario (“Application Reclamation” on page 41) may execute if an application must be relocated in the DAV. If the KVM still has a reference to any classfile, then the reclaim will be postponed until no further references exist. The JAM is responsible for maintaining the number of current references to class units in the DAV.

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

Figure 23: Application Removal Operational Scenario



Note: Sequence continues from reclaim scenario if no objects are still referenced by the KVM.

Application Reclamation

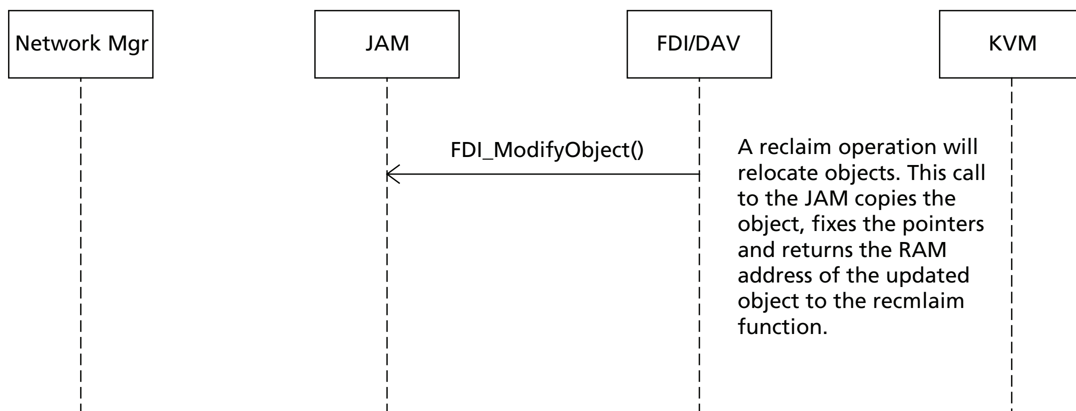
The application reclamation scenario executes whenever an application is removed from Flash memory or a reclaim has been triggered. This scenario assumes the library installation (“Library Installation” on page 24) and the DAV initialization (“Direct Access Volume Initialization” on page 36) have executed without error.

To reclaim an application:

1. If an object is to be moved during reclaim, FDI calls the function FDI_ModifyObject( ) to the JAM. The JAM updates all pointers in the application based on the new storage results (i.e. the application references will be re-snapped to account for the relocated application).
2. FDI then replaces the application containing the old pointer references (now incorrect), with the modified application containing the new pointer references (not fixed-up).
3. On completion of reclaim, applications are shifted to lower addresses, and snapped references are fixed. The significant details and complexity associated with reclamation are handled internally to FDI.

Note: For more information, please see the Code Manager Description (Section 8.2) in the Micron Flash Data Integrator (FDI) User’s Guide, version 3.0.

Figure 24: Application Reclamation Operational Scenario



Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

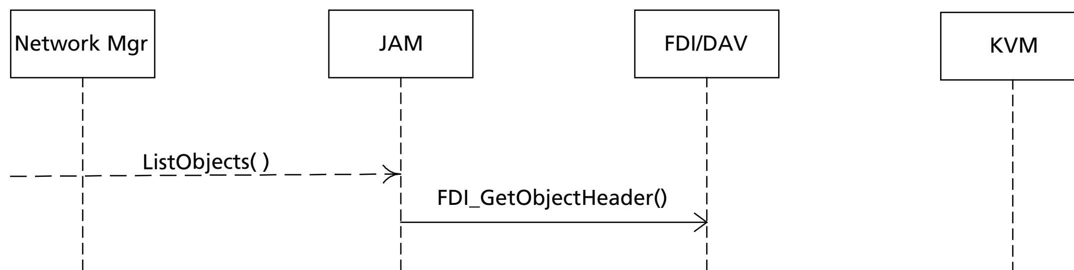
J2ME Application Listing

To obtain a listing of the application currently installed, the JAM simply queries FDI for the installed applications. This scenario assumes the library installation ("Library Installation" on page 24) and the DAV initialization ("Direct Access Volume Initialization" on page 36) have executed without error.

To obtain an application listing:

1. Make a system call to the JAM to inform it that a listing is requested.
2. The JAM will posts requests to FDI for installed objects via the function FDI_GetObjectHeader( ) until no further applications are found in the DAV.

Figure 25: Application Listing Operational Scenario



Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

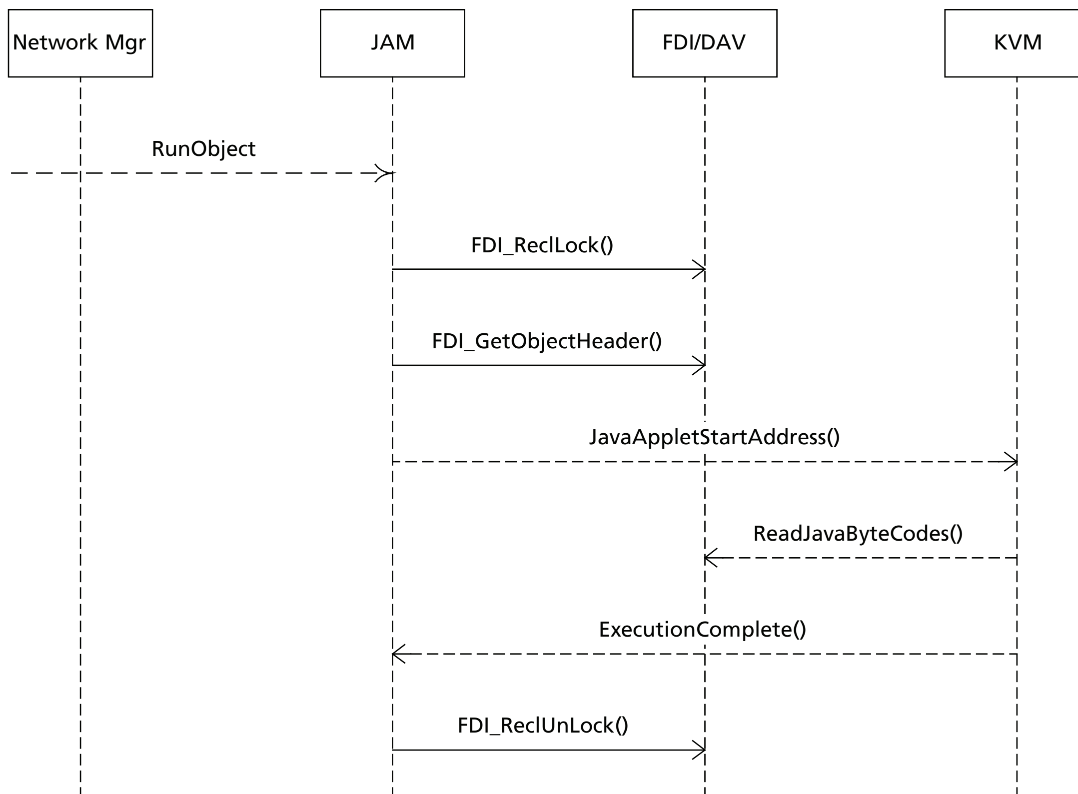
Application Initiation

At some point, the user will request that an application be executed. This scenario assumes the library installation (“Library Installation” on page 24) and the DAV initialization (“Direct Access Volume Initialization” on page 36) have executed without error, and an application has been installed via the application storage scenario (“Application Storage” on page 37).

To initiate an application:

1. Make a system call to the JAM to inform it that an application execution has been requested.
2. The JAM posts the FDI_ReclLock( ) semaphore to FDI. This semaphore blocks reclaim while the KVM is running an application.
3. The JAM confirms that the application exists in the DAV via the function FDI_GetObjectHeader( ).
4. The JAM is the responsible for providing the KVM with the physical start address for the Java application. The KVM starts executing the Java application via XIP capability.
5. The KVM reads the Java classfile and the Java byte codes directly from Flash memory and interprets them.
6. When the Java application has completed execution, the KVM informs the JAM that the application has terminated.
7. If no further applications are executing, or there are no further references to Flash memory by the KVM, then the JAM may post the FDI_ReclUnLock( ) semaphore to FDI. This allows any pending reclaims to start.

Figure 26: Application Initiation Operational Scenario



Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

Compressed Application Storage

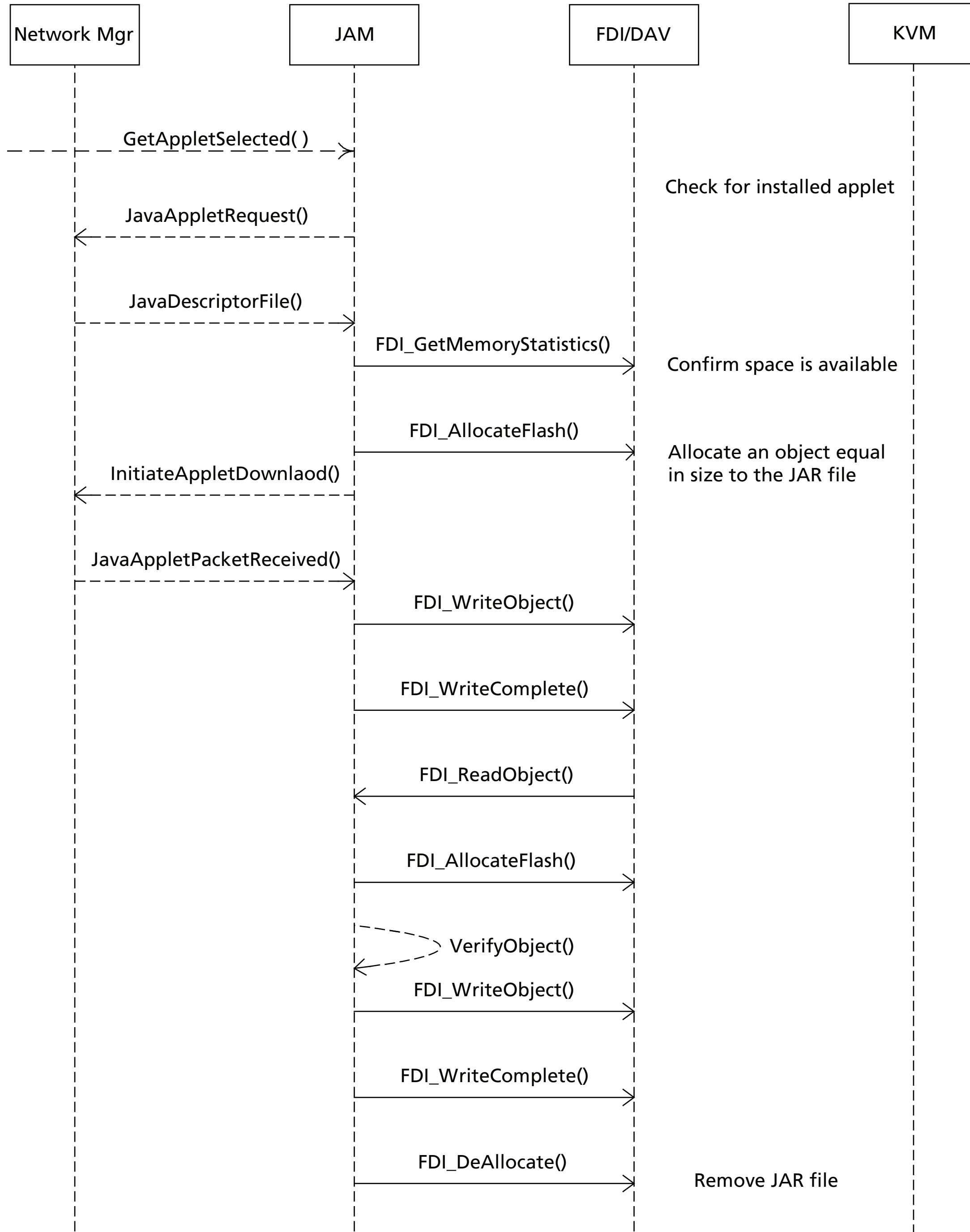
The compressed application scenario executes whenever the user requests to run an application that has been stored in a compressed JAR-formatted file. The JAR file contains the classfile to execute and any other files necessary to execute the application. The JAR file follows a format that allows the JAM to query the JAR file for the number, size, and version of the files that are included in the compressed file. The JAM can then expand the JAR file into a series of Java applications, store the Java applications in Flash memory, and remove the JAR file. This scenario assumes the library installation (“Library Installation” on page 24) and the DAV initialization (“Direct Access Volume Initialization” on page 36) have executed without error.

To install a compressed application:

1. Make a system call to the JAM to inform it that the user has selected an application for download.
2. The JAM then requests the Java descriptor file for the Java application from the network.
3. The JAM receives the Java descriptor file.
4. From the Java descriptor file, the JAM obtains the file name, file version, and file size. The JAM also determines that the file is a compressed JAR file.
5. The compressed file is stored in Flash memory, so the JAM confirms that enough space for the compressed file exists in Flash memory by calling the FDI function FDI_GetMemoryStatistics( ).
6. If enough space exists, FDI pre-allocates the necessary Flash memory space for the application via the FDI function FDI_AllocateFlash( ).
7. Initiate the Java application download.
8. Receive the JAR file.
9. Call the FDI function FDI_WriteObject( ) as necessary to copy the JAR file download packets into Flash memory.
10. After the JAR file has been copied to the Flash memory, call the function FDI_WriteComplete( ) to inform FDI that the application has successfully installed. This will cause internal Code Manager status information to latch, locking the application into the DAV.
11. Read the JAR file from Flash memory to determine the number of files in the JAR file and total decompressed size.
12. If enough free space exists in Flash memory, pre-allocate the necessary Flash memory space for the first file compressed in the JAR file.
13. Decompress the first file compressed in the JAR file.
14. Scan the first file for errors.
15. Verify the first file.
16. Call the FDI function FDI_WriteObject( ) to write the decompressed file to Flash memory. Repeat this step as necessary to copy the expanded file to Flash memory.
17. After the expanded file is copied to Flash memory, call the function FDI_WriteComplete( ) to inform FDI that the file has been successfully installed.
18. Repeat steps 11 to 17 for each file compressed in the JAR file.
19. After every file compressed in the JAR file has been decompressed and copied to Flash memory, call the function FDI_DeAllocate( ) to delete the original JAR file.
20. Continue execution of the decompressed and installed application via the application initiation scenario (“Application Initiation” on page 43).

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

Figure 27: Compressed Application Storage Operational Scenario



Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.

Application Resolution

During execution, Java applications make several calls to other class objects within the class unit or class libraries. This process is called resolution. The nature of a Java classfile is that there are no direct references to other class units or class libraries. Instead, the references are symbolic. A symbolic reference is a fully qualified name. The fully qualified name contains a relative file path, a file name, and an object name. If the object is a function, then the object name also contains encoded function parameters. The Java virtual machine, when it encounters a symbolic reference, must convert the symbolic reference to a physical reference. Converted symbolic names are maintained in a method table that is created for each class object as it executes. The method table is maintained in RAM, while the location referenced by the table entry is stored in Flash memory. If the resolution process proceeds without errors and passes security checks, then the object access or function call is allowed to proceed. If not, a security fault is trapped by the KVM.

To resolve a symbolic reference:

1. Scan the classfile object pool for the symbolic reference.
2. Build the symbolic reference by following the rules for the Java classfile format.
3. Check the method table to see if the symbolic reference has previously been resolved (i.e., the method table contains a valid symbolic-reference / object-pointer pair).
4. If the reference is external to the currently executing Java classfile, scan the installed libraries for a filename that matches the relative path/file name by using the FDI function FDI_GetObjectHeader( ).

Note: Filenames are stored in the Code Manager with the filename concatenated to the relative path. This produces file names that include the relative path. This greatly simplifies the resolution process in an environment that does not include a hierarchical file system.

5. If a matching file is found, complete the resolution process by scanning its object pool for the symbolic reference.
6. If the object is found and the resolution completes without an error or security fault, then a symbolic-reference / object-pointer pair is added to the method table. This pointer is used to access the object whenever the specific symbolic reference is encountered.

Note: A Symbolic Reference is defined as: Java/IO/PrintStream (I)I println - This decodes as a function called println, that takes an integer as a parameter, returns an integer, and is a public member of the PrintStream class. The method table should store pairs of encoded fully-qualified names and a physical pointer to the function.

8000 S. Federal Way, P.O. Box 6, Boise, ID 83707-0006, Tel: 208-368-3900
www.micron.com/productsupport Customer Comment Line: 800-932-4992
Micron and the Micron logo are trademarks of Micron Technology, Inc. All other trademarks are the property of their respective owners.

Micron Technology, Inc., reserves the right to change products or specifications without notice.
©2011 Micron Technology, Inc. All rights reserved.